Порубльов І. М.

ДИСКРЕТНА МАТЕМАТИКА

НАВЧАЛЬНИЙ ПОСІБНИК

Черкаський національний університет імені Богдана Хмельницького

Порубльов І. М.

Дискретна математика

Навчальний посібник
для студентів 1-го курсу бакалаврату
галузі знань «Інформаційні технології»
та споріднених

Черкаси, 2018

УДК 519.1
П60

Рекомендовано до друку вченою радою
Черкаського національного університету імені Богдана Хмельницького
(протокол № 8 від 15.06.2017)

Рецензенти:
доктор фізико-математичних наук Слинько В. І.
доктор фізико-математичних наук Запорожець Т. В.

Порубльов І. М.
Дискретна математика.
Навчальний посібник для студентів 1-го курсу бакалаврату галузі знань «Інформаційні технології» та споріднених. – Черкаси: видавець ФОП Гордієнко Є. І., 2018. – 220 с.

ISBN 978-966-9730-48-3

Посібник містить матеріали з основних галузей дискретної математики, а саме: вступ до математичної логіки; множини та відношення; індуктивні засоби доведення; комбінаторика; графи та алгоритми на графах; мови, регулярні вирази та автомати. Викладення матеріалу поєднує класичний математичний підхід з орієнтованістю на галузі дискретної математики, найбільш потрібні спеціалістам з інформаційних технологій та комп'ютерних наук.

Посібник розрахований на студентів першого курсу бакалаврату, у першу чергу галузі знань «Інформаційні технології» та споріднених.

ISBN 978-966-9730-48-3

Зміст

Вступне слово

Ми починаємо вивчати курс «Дискретна математика». Тож природнє бажання з'ясувати смисл сло́ва «*дискретний*» (рос. «*дискретный*», англ. «*discrete*»). Іноді кажуть, що «дискретність» — це протилежність до «неперервності». Це частково правильно; але висновок, ніби «дискретний» означає «розривчастий», більш заплутує, ніж прояснює ситуацію. Тож наведемо кілька прикладів доречного вживання цього сло́ва.

Розглянемо сукупності (множи́ни) натуральних чисел $\mathbb{N}$ та дійсних чисел $\mathbb{R}$. $\mathbb{N}$ складається з *окремих, чітко відділених одне від одного* натуральних чисел, для яких має смисл поняття «наступне число»; $\mathbb{R}$ — зі «щільно упакованих» чисел, для яких поняття «наступне число» не має смислу, зате має смисл поняття «дуже близькі чи́сла».

Ще розглянемо електронний цифровий та механічний стрілочний годинники, причому нехай вони обидва показують лише години і хвилини. Покази електронного цифрового годинника змінюються «ривками» раз на хвилину, тоді як положення стрілок механічного годинника змінюються потихеньку.

Так от: множина натуральних чисел ($\mathbb{N}$) та покази електронного цифрового годинника є безсумнівно дискретними, множина дійсних чисел ($\mathbb{R}$) — безсумнівно неперервною; покази стрілочного годинника — якщо стрілки рухаються потихеньку й непомітно, то неперервними, а якщо окремі тіки та скачки все-таки помітні, то дискретними.

У реальному світі є і неперевні, і дискретні процеси. Але у комп'ютерній техніці майже всі процеси дискретні. Це вказує на велику важливість дискретної математики для студентів програмістських напрямків підготовки.

Звичайно, як і інші галузі математики, дискретна математика включає в себе різні підрозділи. Зокрема, у посібнику розглянуто такі великі теми:

1. Вступ до математичної логіки
2. Множи́ни та відношення
3. Індуктивні засоби доведення
4. Комбінаторика
5. Гра́фи та алгоритми на графах
6. Мови, регулярні вирази та автомати

Всі ці теми мають практичне значення для програмування.

Ситуація з підручниками з дискретної математики не дуже приємна. У книжках, написаних науковцями найпотужніших ВНЗ, зазвичай розглядають значно ширший (ніж у цьому посібнику) обсяг матеріалу, і часто не досить детально пояснюють базовий матеріал, вважаючи, що він і так зрозумілий. Це спонукало автора приділити значну увагу в тому числі й детальним поясненням базових понять. (Що, втім, не зводить цей посібник до *лише* простого, бо він містить також і аналіз деяких неочевидних «тонких» питань, і велику кількість додаткових задач підвищеної складності.)

Переважна більшість наведених у посібнику означень супроводжуються при́кладами. Наполегливо рекомендується використовувати їх для того, щоб *перевіряти правильність розуміння* означень, співвідносячи приклад(и) з означенням, що сформульоване у загальних поняттях. І *ні в якому разі* ***не*** *рекомендується* намагатися *підмінятися* означення прикладами. Просто тому́, що вони (приклади) його (означення) не замінюють. І з точки зору математики (байдуже, дискретної чи якоїсь іншої), і з точки зору програмування. Скажімо, програма мовою програмування, призначена, щоб додавати введені з клавітури два числа́, повинна містити якусь дію у стилі "`c=a+b`" або "*вивести*`(a+b)`", й тоді вона зовсім елементарна. Але якщо спробувати лише понаписувати величезну кількість прикладів (у стилі «1+1=2», «2+2=4», «17+25=42»), не пишучи правило — навряд чи таке взагалі запрацює.

Аналогічно, деякі означення та теореми сформульовані і словесно, і формулами. У таких випадках теж рекомендується перевіряти правильність розуміння, співвідносячи вміст, поданий цими різними способами, і привчатися розуміти формули. Хоч програми мовою програмування й не є класично-математичними формулами, але все-таки ближчі до них, ніж до розмовної мови. Так що навичка розуміти формули не лише потрібна в рамках математики (не лише дискре-

тної), а ще й частково корисна для розвитку навички читання та написання програм мовами програмування.

Інша велика складність з літературою спричинена тим, що у багатьох розділах дискретної математики ще не встановилися єдині загальноприйняті позначення. Тобто, одне й те са́ме може позначатися у різних книжках по-різному, а іноді й ще гірше — одна й та сама назва або один і той самий значок у різних книгах мають різне значення.

У рамках цього посібника позначення та термінологія, звісно, узгоджені. Але для відмінної підготовки все-таки варто користуватися різними джерелами інформації, так що ця проблема цілком може проявитися. Са́ме тому́ в багатьох місцях оголошується окремо, які терміни чи позначення використовуються у рамках цього посібника, і окремо, які є поширені альтернативи. Так само з метою полегшення користування альтернативними джерелами інформації для більшості термінів наведено їхні переклади російською та англійською мовами.

Насамкінець: при вивченні нового матеріалу трапляються сумніви та/або неправильне розуміння. Це нормально. Ненормально не намагатися зрозуміти, не задумуватися, чи утворює вивчений матеріал цілісну непротирічиву теорію. Коли виникають питання, які можна задати викладачеві — не стидайтеся справді задавати.

1 Вступ до математичної логіки

1.1 Базові поняття математичної логіки

1.1.1 Значення, операції, вирази

Перш за все, зафіксуємо можливі *логічні* (або *бу́леві*[1]) *значення*. Це *істина* і *хиба* (рос. «*истина*» / «*ложь*», англ. «*true*» / «*false*»). Інших значень у класичній логіці не буває.[2] Істину будемо позначати символом "1", хибу — символом "0". Коли треба підкреслити, що йдеться са́ме про логічне значення, а не число, істину позначають «`true`», хибу — «`false`». Значення `true` та `false` є аналогом чисел (1, 2, 3, ...) «звичайної» (числової) математики.

Логічна *змінна* задає логічне значення (або 0, або 1) — аналогічно змінній «шкільної» алгебри, невідомо яке, або таке, що може змінитися.

Логічна *операція* задає правило перетворення логічних значень або змінних, аналогічне арифметичній дії (додавання, множення, ...).

Заперечення Інші назви: "*не*", "`not`" "*negation*". Позначення: "$\neg x$", "$\overline{x}$",[3] "x'". Результат заперечення протилежний аргументу: $\neg 0 = 1$, $\neg 1 = 0$.

Кон'юнкція Інші назви: "*логічне множення*", "*і*", "*та*", "`and`", "*conjunction*". Позначення: "$x \wedge y$", "$x \cdot y$", "xy",[4] "$x\&y$". Результат кон'юнкції істинний, коли обидва аргументи істинні (у решті випадків хибний).

Диз'юнкція Інші назви: "*або*", "*логічне додавання*"[5], "`or`", "*disjunction*". Позначення: "$x \vee y$". Результат диз'юнкції істинний, коли хоча б один аргумент істинний (і хибний коли обидва аргументи хибні).

Ксор Інші назви: "*виключне або*", "*або ..., або ...*", "*сума Жегалкіна*", "*додавання за модулем 2*", "`xor`", "*exclusive or*".[6] Позначення: "$x \oplus y$", "$x \not\equiv y$", "$x \triangle y$". Результат ксора істинний, коли істинний *рівно один* з двох аргументів (а коли обидва хибні чи обидва істинні, то хибний).

Імплікація Інші назви: "*якщо ..., то ...*", "*з ... випливає ...*", "*слідування*", "*implication*", "*... implies ...*". Позначення: "$x \to y$", "$x \Rightarrow y$", "$x \supset y$". Результат імплікації хибний, коли перший аргумент істинний, а другий хибний; у решті випадків результат істинний.

Еквіваленція Інші назви: "*рівносильність*", "*... тоді й тільки тоді, коли ...*", "*biconditional*", "*if and only if*", "*iff*"[7]. Позначення: "$x \leftrightarrow y$", "$x \equiv y$", "$x \Leftrightarrow y$", "$x \sim y$". Результат еквіваленції істинний, коли значення аргументів однакові, і хибний, коли різні.

Переклади назв операцій рос. мовою: отрицание; конъюнкция; дизъюнкция; ксор (исключающее или); импликация; эквиваленция.

Наведені означення операцій зводяться до пояснень, яким є результат такої-то операції при таких-то значеннях її аргументів. Це можна зручніше й коротше записати за допомогою т. зв. *таблиць істинності* (укр. «таблиця істинності»; рос. «*таблица истинности*»; англ. «*truth table*»; і укр., і рос. мовами часто кажуть також «табличка», це те са́мо).

---

[1] На честь англ. математика XIX ст. Дж. Бу́ля (George Boole), завдяки роботам якого і з'явилися перші результати *математичної логіки*. До того, ще з часів Аристотеля, була відома *філософська логіка*.

[2] Але бувають у деяких некласичних узагальненнях логіки. Зокрема, в цьому посібнику коротко згадана (розд. 1.8) некласична тризначна логіка.

[3] Позначення заперечення як "$\overline{x}$" використовується у багатьох джерелах, красиве, дуже компактне... Але біда в тому, що $\overline{x}\,\overline{y}$ означає $(\neg x) \wedge (\neg y)$, $\overline{xy}$ означає $\neg(x \wedge y)$, тобто зовсім іншу формулу, а візуальна відмінність між $\overline{x}\,\overline{y}$ та $\overline{xy}$ надто мала́. Тому автор наполегливо радить або (як у цьому посібнику) взагалі уникати позначення "$\overline{x}$", або при спільному використанні заперечень та кон'юнкцій писати кон'юнкції явно: відмінність між $\overline{x} \cdot \overline{y}$ та $\overline{x \cdot y}$ видно значно краще, ніж між $\overline{x}\,\overline{y}$ та $\overline{xy}$.

[4] Тобто, у багатьох випадках значок кон'юнкції просто пропускають — так само, як пропускають значок множення числових змінних.

[5] Ми будемо уникати цього варіанта назви, щоб не плутати диз'юнкцію з ксором.

[6] Тобто, «ксор» — транслітерація від «хог», котре є скороченням від «exclusive or», що якраз і є «виключне або». Але казати «виключне або» досить довго й не дуже красиво, тому в цьому посібнику перевага надається варіанту «ксор».

[7] Са́ме з двома "f"; таке слово не притаманне загальній лексиці англійської мови, але має певне поширення як термін матлогіки.

Кожен рядок таблиці істинності містить один набір аргументів та значення операції на цьому наборі. Таблиця повинна містити всі можливі набори. Значить, для унарної[8] операції таблиця істинності повинна складатися з 2 рядків (0 та 1), а для бінарних[8] операцій — з 4 рядків (0 0, 0 1, 1 0 та 1 1).

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \to y$	$x \leftrightarrow y$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1

"$\oplus$" (ксор, він же «виключне або») відрізняється від "$\vee$" (диз'юнкції, вона ж «або») на єдиному наборі 1 1, а "$\to$" (імплікація) від "$\leftrightarrow$" (еквіваленції) — на єдиному наборі 0 1.

Ще раз зверніть увагу, що фразі «якщо ..., то ...» відповідає операція "$\to$" (для якої $0 \to 1 = 1$), а не "$\leftrightarrow$" (для якої $0 \leftrightarrow 1 = 0$). Нехай, наприклад, батько пообіцяв сину: *«Якщо поступиш на бюджет — куплю тобі мопед»*. У яких випадках можна сказати, що батько порушив свою обіцянку? *Лише* якби син на бюджетну форму навчання поступив, а батько мопеда не купив. У випадках «поступив — купив» та «не поступив — не купив» обіцянка очевидно дотримана, але вона не порушена також і у випадку, коли син не поступив, а батько все ж купив мопед.

Ще один приклад: ніхто не стане сумніватися в істинності фрази *«якщо забредеш у Дніпро, то замочиш ноги»* на тій підставі, що ноги можна замочити й у якомусь іншому місці. (*Якби* була "$\leftrightarrow$", то вийшло *б* «забрів у Дніпро» = `false`, «замочив ноги» = `true`, `false` $\leftrightarrow$ `true` дає `false`, тобто випадок коли замочив ноги в якомусь іншому місці суперечив *би* фразі. Але насправді маємо $0 \to 1 = 1$, і ніякої хибності фрази в цілому не виникає.)

Правила побудови логічних виразів (формул) Зі значень, змінних та операцій можна «збирати» складніші вирази (вони ж *формули*), що можуть містити по кілька операцій — наприклад, "$(x \vee y) \to (x \oplus y)$". Математично строго правила побудови виразів можна записати так:

- константа 0 є логічним виразом;
- константа 1 є логічним виразом;
- логічна змінна є логічним виразом;
- якщо A є логічним виразом, то $(\neg A)$ теж є логічним виразом;
- якщо A і B є логічними виразами, то кожен з записів $(A \vee B)$, $(A \wedge B)$, $(A \oplus B)$, $(A \to B)$, $(A \leftrightarrow B)$ теж є логічним виразом.

(В цих правилах насправді нема нічого нового — тільки ще раз наголошено, що логічний вираз або є елементарним (єдине значення або єдина змінна), або складається з менших виразів (*підвиразів*), до яких застосована одна з операцій; при цьому, будь-яка з операцій "$\vee$", "$\wedge$", "$\oplus$", "$\to$" або "$\leftrightarrow$" має з'єднувати два підвирази, а операція "$\neg$" застосовується до одного підвиразу.)

Пріоритети логічних операцій Як відомо, у «числовій» математиці порядок виконання дій у виразі визначається порядком само́їх операцій, дужками та т. зв. *пріоритетами* (наприклад, $2 + 3 \cdot 4$ рахується як $2 + 12 = 14$, а не як $5 \cdot 4 = 20$). (Назва: укр. — *пріоритет*; рос. — *приоритет*; англ. — *precedence*; са́ме «precedence», а не «priority».)

Порядок виконання операцій у логічному виразі теж визначається по-перше дужками, по-друге пріоритетами, по-третє порядком зліва направо. Пріоритети логічних операцій такі:

(найвищий) $\neg$ $\wedge$ $\vee$ $\to$ (найнижчий)

(З операціями "$\oplus$" та "$\leftrightarrow$" ситуація гірша. Загальноприйнято, що пріоритет "$\oplus$" десь між "$\wedge$" та "$\to$", але єдиного стандарта щодо співвідношення пріоритетів "$\oplus$" та "$\vee$" нема. Так само загальноприйнято, що пріоритет "$\leftrightarrow$" нижчий за "$\vee$", але єдиного стандарта щодо співвідношення "$\leftrightarrow$" з "$\oplus$" та "$\to$" нема. Тому́ скрізь, де виникає загроза неоднозначності, будемо писати дужки.)

Наприклад, знаходити значення виразу $1 \wedge 0 \to 1 \vee 0$ слід так: спочатку $1 \wedge 0 = 0$, вираз спрощується до $0 \to 1 \vee 0$; потім $1 \vee 0 = 1$, вираз спрощується до $0 \to 1$; нарешті, $0 \to 1 = 1$ (отримуємо остато́чний результат 1).

[8] *унарний* — залежний від одного аргумента; *бінарний* — залежний від двох аргументів; *тернарний* — від трьох; для інших кількостей аргументів теж використовують термін «арність», але ці кількості зазвичай позначають вже не латинськими префіксами, а просто числами (0-арний, 8-арний, n-арний)

1.1.2 Логічні операції у мовах програмування

(Цей розд. 1.1.2 містить деяку інформацію щодо застосування логічних операцій у мовах програмування (головним чином, C++). Його матеріал майже не використовується у подальших розділах цього посібника. Тому ті читачі, котрі ще не почали програмувати та/або працювати з двійковою системою числення, цілком можуть при першому прочитанні обмежитися лише зрозумілим їм матеріалом, а до решти розд. 1.1.2 повернутися пізніше. Водночас, прочитати бажано, бо матеріал дуже корисний у програмуванні.)

Більшість логічних операцій наявні також у мовах програмування. І конкретні позначення операцій, і навіть їх точний перелік у різних мовах програмування можуть відрізнятися. Найбільш масово у програмуванні застосовують такі операції:

	заперечення	кон'юнкція	диз'юнкція	ксор
математичне позначення	$\neg$	$\wedge$	$\vee$	$\oplus$
мова Pascal	`not`	`and`	`or`	`xor`
мова C++, логічна операція	`!`	`&&`	`||`	(див. далі)
мова C++, побітова операція	`~` (тильда)	`&`	`|`	`^` (кришка)

У C++ є два чинники, що ускладнюють ситуацію: (1) логічні значення можуть подаватися як у типі `bool`, так і цілочисельних типах (зазвичай, `int`); (2) є «логічні» та «побітові» операції. Розглянемо все це детальніше.

Щодо bool та int Мова C++ має окремий тип `bool` для подання (лише) значень `false` та `true`. Але мова C++ розроблена так, щоб компілювати також і більшість програм, написаних старими версіями мови C (без плюсів), де нема типу `bool` і для подання логічних значень використовується той самий `int`, що для цілих чисел (`0` трактується як хиба, *будь-яке інше* число — як істина). Тому, логічні операції "`!`", "`&&`", "`||`" можуть застосовуватися не лише до `false` та/або `true`, а також і до чисел. Наприклад, кожен з виразів "`42 && 7`" та "`17 || 0`" дає результат `true` (при присвоєнні у `bool`) або 1 (при присвоєнні в `int`), бо це `true`∧`true` та `true`∨`false` відповідно.

Пишучи нові програми мовою C++, рекомендується вживати `bool` і *не* користуватися тим, що цілі чи́сла сприймаються як логічні значення. Але бувають виключення: (А) код треба написати так, щоб працював як у C++, так і в C; (Б) значна частина коду вже написана з використанням са́ме `int`-ового подання логічних значень.

Дуже сумним наслідком змішування `bool`-ів та `int`-ів є те, що мовами C/C++ вирази, подібні до "`a<x<b`", компілюються, але означають *зовсім не те*, що символ-у-символ такі са́мі математичні. Математичний вираз "$1<x<2$" є подвійною нерівністю й означає «x строго більший 1, але строго менший 2», або, що те са́ме, «x перебуває у проміжку від 1 до 2, ме́жі (одиниця та двійка) не включаються». Вираз "`1<x<2`" мовою C/C++ обчислюється зовсім інакше: результат порівняння `1<x` перетворюється або в 1 (якщо x більший 1), або в 0 (якщо x менший або рівний 1); потім отримане 1 або 0 порівнюється із 2; причому, остато́чний результат завжди, яким би не був `x`, буде істинним, бо хоч `1<2`, хоч `0<2` дають істину. Це вельми неприємна поведінка, тому́ в більшості сучасніших C-подібних мов (як-то Java, C#) таких автоматичних переходів між логічними та числовими значеннями просто нема. Але у C/C++ маємо те, що маємо.

(Як все-таки виразити математичну подвійну нерівність мовами C/C++? Врахувати, що це дві умови, накладені одночасно, й виразити через "`&&`". Наприклад, "$1<x<2$" можна правильно записати як "`1<x && x<2`". Додаткові дужки, на відміну від Паскаля, не є обов'язковими; хоча, при бажанні їх можна й поставити: "`(1<x) && (x<2)`".)

Логічні та побітові операції Логічні операції ("`!`", "`&&`", "`||`") ставляться до кожного зі своїх аргументів як до одного логічного значення, й формують як результат одне логічне значення. Побітові операції ("`~`", "`&`", "`|`", "`^`") застосовуються до окремих бітів цілих чисел.

Наприклад, праворуч зображені молодші 8 бітів двійкових записів чисел $42_{Dec}=101010_{Bin}$ та $7_{Dec}=111_{Bin}$.

`a`	00101010
`b`	00000111
`a & b`	00000010
`a | b`	00101111
`a ^ b`	00101101

Якщо застосувати до них операцію `&` (побітовий `and`), виявиться, що є лише один біт (передостанній), в якому обидва числа́ $42_{Dec}=101010_{Bin}$ та $7_{Dec}=111_{Bin}$ містять одиниці, тому результатом буде $10_{Bin}=2_{Dec}$. Так і отримується твердження «результатом "`42 & 7`" є число `2`».

Аналогічно можна застосувати `|` (побітовий `or`, результат $101111_{Bin}=47_{Dec}$, бо са́ме в цих бітах хоча б одне з чисел-аргументів містить 1) або `^` (побітовий `xor`, результат $101101_{Bin}=45_{Dec}$).

Тепер замінимо $7_{Dec}=111_{Bin}$ на $5_{Dec}=101_{Bin}$ (відрізняється лише нулем замість одиниці у передостанньому біті). Легко бачити, що `42 & 5` дає результат 0, бо нема жодного *спільного* біта, який був би одиницею в обох числах $42_{Dec}=101010_{Bin}$ та $5_{Dec}=101_{Bin}$ одночасно.

Таким чином, `42 & 5` дає результат 0 (що відповідає `false`), хоча `42 && 5` дає результат `true` (як `true`$\wedge$`true`). Тобто, бувають випадки (їх насправді небагато, але вони є), коли результат застосування "`&`" відрізняється від результату застосування "`&&`" до тих самих аргументів найістотнішим чином: виходять різні не лише чи́сла, а й відповідні їм логічні значення.

Тому́, бажано чітко розрізняти логічні операції від побітових, й застосовувати потрібні, не плутаючи подвійні значки з одинарними.

Скорочене обчислення "`&&`" та "`||`" Ще одна відмінність логічних та побітових операцій — для логічних "`&&`" та "`||`" діє (а для побітових "`&`" та "`|`" не діє) *скорочене обчислення* (рос. «*сокращённое вычисление*», англ. «*short-circuit evaluation*»). Його суть така: спочатку знаходиться значення лівого аргументу; ***якщо** за його значенням **вже видно**, яким буде **загальний результат, то правий аргумент не аналізується***. Обчислення "`&&`" можна обривати, якщо лівий аргумент=`false` (яким би не був дру́гий аргумент, результат усього виразу все одно `false`); якщо ж лівий аргумент=`true`, то для знаходження усього виразу треба знаходити правий аргумент. З обчисленням "`||`" усе симетрично: якщо лівий аргумент=`true`, результат `true`; інакше, треба знаходити правий аргумент.

(Наприклад, вираз "`x>=0 && sqrt(x)<=7.5`" гарантовано не зіткнеться з добуванням кореня з від'ємного числа́,[9] бо `sqrt` обчислюватиметься *лише* якщо $x\geqslant 0$. А вираз "`sqrt(x)<=7.5 && x>=0`" може й видобувати корінь з від'ємного числа́. Тобто, у програмуванні "`a&&b`" не завжди цілком рівносильне "`b&&a`". Незважаючи на те, що у математиці $a\wedge b=b\wedge a$ є стандартною тотожністю. З "`||`" ("$\vee$") аналогічно.)

Якщо вираз містить кілька однотипних операцій підряд (наприклад, "`0<=x && x<=5 && 3<=y && y<=4`"), усе аналогічно: значення підвиразів знаходять по порядку зліва направо; якщо в якийсь момент результат стає і так зрозумілим, подальші підвирази не знаходяться.

Еквіваленція Еквіваленція $a\leftrightarrow b$ зазвичай може бути виражена як порівняння "`a==b`". Тільки треба переконатися, що відповідні логічні значення подані `bool`-ами, а не цілочисельно (бо коли потрібна еквіваленція, а однакові логічні значення `true` подані *різними ненульовими* цілими числами, результат вийде `false` замість очікуваного `true`). Якщо є сумніви, можна привести аргументи до `bool`-ів або присвоєннями у проміжні змінні типу `bool`, або приведенням типу вигляду "`(bool)a == (bool)b`".

Логічний ксор Така операція на практиці буває дуже потрібною (і її містять деякі інші мови, включаючи Паскаль), а готової її у C/C++ нема. В принципі можна виражати логічний ксор $a\oplus b$ через побітовий "`(bool)a ^ (bool)b`", або як "`(bool)a != (bool)b`". Щоправда, це багато ким сприймається як небажане відхилення від стандартів.

Імплікація Ця операція дуже рідко потрібна у програмуванні. В принципі можна виражати $a\to b$ як `(bool)a <= (bool)b` (де "`<=`" означає "$\leqslant$"). Але це набагато більш небажане відхилення від стандартів.

1.1.3 Таблиці істинності складених логічних виразів

Для складених логічних виразів, що залежать від змінних, природньо з'являється операція ***побудови** таблиці істинності*. Адже для довільного логічного виразу можна записати всі можливі набори значень логічних змінних, для кожного набору знайти значення виразу...

Якщо вираз залежить від n логічних змінних, таблиця істинності містить 2^n рядків.

Доведення. При $n=1$ таблиця істинності справді складається з $2^1=2$ рядків.

Тепер (вважаючи гарантованим, що для k логічних змінних таблиця істинності містить 2^k рядків), розглянемо усі можливі набори при кількості логічних змінних $k+1$. Розділимо їх на дві групи: набори, де «нова́» $(k+1)$-а змінна набуває значення 0, та набори, де вона набуває значення 1. У кожній з цих груп решта k змінних можуть утворювати всі можливі набори — отже, кількість наборів у кожній з цих груп дорівнює 2^k; звідси, загальна кількість наборів $k+1$ логічних змінних дорівнює $2^k+2^k=2\cdot 2^k=2^{k+1}$. ■

[9] до чого са́ме призводить спроба добути корінь з від'ємного числа́ — чи до аварійного завершення програми, чи до тихого отримання значення `NaN` (not a number), чи ще до чогось — може залежати від мови програмування (навіть версії та налаштувань компілятора), й детально тут не обговорюється

Рядки таблиць істинності слід записувати у правильному порядку: 1-й стовпчик змінних містить у верхній половині рядків лише 0, у нижній — лише 1; для 2-го стовпчика, у кожній з половин виділяються свої половини, і теж верхня заповнюється 0, нижня 1; і т. д. Правильний порядок рядків для $n = 2$, $n = 3$ та $n = 4$ має вигляд

0	0
0	1
1	0
1	1

,

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

та

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

(Цей порядок можна отримати і з інших міркувань: якщо розглянути кожен набір як двійковий запис числá, то правильний порядок наборів відповідає послідовним числам від 0 до $2^n - 1$.)

Щоб отримати результат (стовпчик значень досліджуваного виразу на всіх можливих наборах змінних) просто виділяють підвирази (згідно порядку виконання операцій) і знаходять значення для кожного рядка.

Наприклад, побудуємо таблицю істинності для виразу $(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$. Вираз залежить від двох змінних, тож перш за все запишемо відповідні чотири рядки, потім виділяємо «найбільш внутрішні» підвирази, значення яких слід знайти першими, і заповнюємо стовпчики значень цих підвиразів, потім переходимо до «більших» підвиразів, будуючи значення в їхніх стовпчиках на основі значень у раніше побудованих стовпчиках — доки не побудуємо стовпчик значень виразу в цілому.

x_1	x_2	$x_1 \vee x_2$	$x_1 \wedge x_2$	$\neg(x_1 \wedge x_2)$	увесь вираз
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	0

Якщо вираз великий, виникає технічна незручність: «заголовки» стовпчиків (якими досі були підвирази) займають надто багато місця, і таблиця стає неадекватно широченною. Це можна вирішувати по-різному; на думку автора посібника, найзручніше так: повиділяти у виразі нижніми та/або верхніми дужками підвирази (див. приклад), попозначати їх (1), (2), (3), ..., і саме ці позначки (1), (2), (3), ... використати у самій таблиці. Тоді і таблиця компактна, і досить легко бачити, що звідки береться, і ясно, яку частину початкового виразу містить стовпчик. Приклад наведено нагорі стор. 13.

Дослідження тотожньої рівності логічних виразів за допомогою таблиць істинності Якщо побудовані таблиці істинності логічних виразів виявилися однаковими (по *всім* рядкам), це *доводить тотожню рівність* (вона ж еквівалентність або рівносильність) цих виразів. Якщо ж таблиці відрізняються (*хоча б одним* рядком), це *доводить* їхню *не*тотожність. Тобто, цим способом гарантовано можна взнати результат; не буває ситуацій «таблиці побудували, але зрозуміти з них, чи вирази тотожні, не зуміли».

(Наприклад, раніше побудована таблиця істинності виразу "$(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$", а ще раніше задані стандартні таблиці ксора та диз'юнкції. Тим, що таблиця $(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$ виявилася цілком однаковою з таблицею ксора, доведено, що цей вираз рівносильний з $x_1 \oplus x_2$. А тим, що однаковість з таблицею диз'юнкції виконується лише на 3-х рядках з 4-х, доведено нерівносильність того ж виразу з $x_1 \vee x_2$.)

Такі перевірки мають смисл ***лише** для таблиць, побудованих для **одних і тих са́мих змінних в одному й тому са́мому порядку!***

(Наприклад, "$x \vee y$" та "$a \vee b$" *не* еквівалентні, не зважаючи на «однаковий» вигляд 0 1 1 1 (згори донизу); толку з такої «однаковості», коли в одному випадку «набір "0 0"» означає $x = y = 0$, в іншому $a = b = 0$, і це різні речі?)

Попередній абзац може схилити до думки, ніби вирази з різними наборами змінних не бувають тотожньо рівними. Це не зовсім так.

$$x_2 \wedge \Big(x_4 \leftrightarrow \big(x_4 \oplus \underbrace{(x_1 \to x_2)}_{(1)}\big)\Big)_{(2)}\Big)_{(3)} \wedge \Big(x_3 \leftrightarrow \Big(\neg\big(x_2 \to \underbrace{(x_2 \leftrightarrow x_3)}_{(4)}\big)_{(5)}\big)_{(6)} \vee x_4 \vee \underbrace{\neg x_1}_{(7)}\Big)_{(8)}\Big)_{(9)}$$

x_1	x_2	x_3	x_4	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	увесь вираз
0	0	0	0	1	1	0	1	1	0	1	1	0	0
0	0	0	1	1	0	0	1	1	0	1	1	0	0
0	0	1	0	1	1	0	0	1	0	1	1	1	0
0	0	1	1	1	0	0	0	1	0	1	1	1	0
0	1	0	0	1	1	0	0	0	1	1	1	0	0
0	1	0	1	1	0	0	0	0	1	1	1	0	0
0	1	1	0	1	1	0	1	1	0	1	1	1	0
0	1	1	1	1	0	0	1	1	0	1	1	1	0
1	0	0	0	0	0	1	1	1	0	0	0	1	0
1	0	0	1	0	1	1	1	1	0	0	1	0	0
1	0	1	0	0	0	1	0	1	0	0	0	0	0
1	0	1	1	0	1	1	0	1	0	0	1	1	0
1	1	0	0	1	1	0	0	0	1	0	1	0	0
1	1	0	1	1	0	0	0	0	1	0	1	0	0
1	1	1	0	1	1	0	1	1	0	0	0	0	0
1	1	1	1	1	0	0	1	1	0	0	1	1	0

Розглянемо вирази "$(x \vee y) \leftrightarrow (\neg xy)$" та "$x \oplus 1$". Побудуємо для них таблиці істинності від змінних x, y (усіх, що згадуються хоча б у одному з виразів; формально кажучи, береться об'єднання (див. розд. 2.1.4) наборів змінних).

Як бачимо, таблиці все ж однакові; отже, вирази еквівалентні.

x	y	$\neg x$	$\neg xy$	$x \vee y$	$(x \vee y) \leftrightarrow (\neg xy)$	$x \oplus 1$
0	0	1	0	0	1	1
0	1	1	1	1	1	1
1	0	0	0	1	0	0
1	1	0	0	1	0	0

Може здатися дивним: "$(x \vee y) \leftrightarrow (\neg xy)$" містить y, а "$x \oplus 1$" ні; як вони можуть бути тотожньо рівними? Виявляється, 1-й вираз *містить* y, але *не залежить* від нього: при x=0 завжди (не зважаючи на значення y) виходить 1, при x=1 завжди 0. Такі ситуації (формально присутньої змінної, від якої насправді нічого не залежить) називають «*фіктивна* змінна» (рос. «*фиктивная* переменная», англ. «*non-essential* variable»).

Можна і не таблицю $x \oplus 1$ будувати в наборі x, y, а, навпаки, побудувати звичайну 4-рядкову таблицю "$(x \vee y) \leftrightarrow (\neg xy)$", помітити там фіктивність y і скоротити таблицю до вигляду, наведеного праворуч. Очевидно, "$x \oplus 1$" має ту саму таблицю істинності; отже, вирази тотожньо рівні.

x	$(\neg x \vee y) \leftrightarrow (xy)$
0	1
1	0

Будувати таблиці різних виразів окремо й викреслювати фіктивні змінні краще тим, що менший сумарний обсяг таблиць; але процес визначення й вилучення фіктивних змінних може бути й помітно складнішим, ніж тут.

1.2 Аналітичні перетворення логічних виразів. Стандартні логічні тотожності (закони)

Таблиці істинності — потужний і простий засіб роботи з логічними виразами. Але треба вміти робити й аналітичні перетворення — розкривати дужки, зводити подібні, тощо. Особливо, коли *не* задані 2 вирази, і потрібно сказати, чи тотожні вони, *а* просять «спростити» вираз.

Отже, запишемо перелік законів (вони ж «стандартні тотожності») для логічних операцій.

1. Комутативність (commutativity; у школі комутативність додавання та множення називають «переставний закон»):
$$a \vee b = b \vee a, \quad a \wedge b = b \wedge a.$$

2. Асоціативність (associativity; «сполучний закон»):
$$(a \vee b) \vee c = a \vee (b \vee c), \quad (a \wedge b) \wedge c = a \wedge (b \wedge c).$$

3. Дистрибутивність (distributivity; «розподільний закон»):
$$(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c), \quad (a \wedge b) \vee c = (a \vee c) \wedge (b \vee c).$$
(Перша тотожність називається *дистрибутивність кон'юнкції відносно диз'юнкції* (англ. «distributivity of conjunction *over* disjunction»); відповідно дру́га — *"$\vee$"-ії відносно "$\wedge$"-ії* ("$\vee$" over "$\wedge$"). Зверніть увагу, що для "$\wedge$" та "$\vee$" виконуються обидві дистрибутивності, тоді як для числових "$+$" та "$\times$" — лише множення відносно додавання, $(a{+}b)c{=}ac + bc$.)

4. Закони нуля, одиниці і заперечення (bound and negation rules):
$$a \vee \neg a = 1, \quad a \wedge \neg a = 0, \quad a \vee 0 = a, \quad a \wedge 1 = a.$$
(Закон "$a \vee \neg a = 1$" називають ще *законом виключення третього* (*закон исключения третьего*, *law of the excluded middle*); "$a \wedge \neg a = 0$" — *законом суперечності*; "$a \vee 0 = a$" та "$a \wedge 1 = a$" іноді групують з законами поглинання (див. далі).)

Перелічених досі законів достатньо, щоб вивести будь-які (правильні) тотожності для операцій "$\vee$", "$\wedge$" та "$\neg$" — в т. ч. і наведені далі. Але на практиці зручніше користуватися ширшим набором законів, тому продовжимо.

5. Ідемпотентність (idempotence):
$$a \vee a = a, \quad a \wedge a = a.$$

6. Закони де Морга́на (DeMorgan's laws):
$$\neg(a \vee b) = \neg a \wedge \neg b, \quad \neg(a \wedge b) = \neg a \vee \neg b.$$

7. Закон подвійного заперечення (double negation):
$$\neg(\neg a) = a.$$

8. Закони поглинання:
$$a \vee 1 = 1, \quad a \wedge 0 = 0.$$
(У деяких книгах ці тотожності ніяк не називають, а назву «закони поглинання» використовують для тотожностей "$a \vee (a \wedge b) = a$" та "$a \wedge (a \vee b) = a$".)

На цьому перелік часто вживаних законів для операцій "$\vee$", "$\wedge$" та "$\neg$" закінчується. Але є ще інші логічні операції...

9. Вираження імплікації:
$$a \to b = \neg a \vee b.$$

10. Вираження заперечення імплікації:
$$\neg(a \to b) = a \wedge \neg b.$$

11. Закон контрапозиції імплікації (contrapositive law):
$$a \to b = \neg b \to \neg a.$$
(А вирази "$a \to b$" і "$\neg a \to \neg b$" <u>не</u> тотожньо рівні!)

12. Зв'язок між "$\leftrightarrow$" та "$\to$":
$$(a \leftrightarrow b) = (a \to b) \wedge (b \to a).$$

13. Комутативності "$\oplus$" та "$\leftrightarrow$":
$$a \oplus b = b \oplus a, \quad a \leftrightarrow b = b \leftrightarrow a.$$
("$\to$" не комутативна!)

14. Асоціативності "$\oplus$" та "$\leftrightarrow$":
$$(a \oplus b) \oplus c = a \oplus (b \oplus c), \quad (a \leftrightarrow b) \leftrightarrow c = a \leftrightarrow (b \leftrightarrow c),$$
("$\to$" не асоціативна!)
15. Дистрибутивність "$\wedge$" відносно "$\oplus$":
$$(a \oplus b) \wedge c = ac \oplus bc.$$
(Дистрибутивність "$\oplus$" відносно "$\wedge$" не виконується, тобто $(a \wedge b) \oplus c$ не завжди рівне $(a \oplus c) \wedge (b \oplus c)$)
16. Властивості ксора:
$$a \oplus a = 0, \quad a \oplus 0 = a.$$
17. Зв'язок між ксором та запереченням:
$$a \oplus 1 = \neg a.$$
18. Зв'язок між ксором та еквіваленцією:
$$(a \oplus b) = \neg(a \leftrightarrow b), \quad (a \leftrightarrow b) = \neg(a \oplus b) = a \oplus b \oplus 1.$$
(Саме тому для ксора іноді використовують значок "$\not\equiv$" (де "$\equiv$" — еквіваленція).)

При доведенні тотожностей логічних виразів *не можна* «викреслювати однакові шматочки» з обох частин, як часто роблять у «числовій» алгебрі (наприклад, "$A + c = B + c$" перетворюють у "$A = B$"). Закони логічних операцій відрізняються від законів арифметичних операцій, і, наприклад, "$A \vee c = B \vee c$" та "$A = B$" — не рівносильні рівності: при $A = 0$, $B = 1$, $c = 1$ перша рівність виконується, дру́га — ні.

Тому́ основний спосіб аналітичного доведення тотожньої рівності логічних виразів — спростити кожен вираз окремо до одного й того ж вигляду. Якщо вдалося отримати однакові вирази, то цим самим доведення тотожньої рівності початкових виразів успішно завершене. Але якщо отримати однакові вирази не вдалося, це нічого не означає: можливо, не вдалося, бо не можна; можливо, не вдалося, бо погано старалися...

Приклад 1. Довести тотожність $p \to (q \to r) = (p \wedge q) \to r$.

Перетворення лівої частини:

$p \to (q \to r) =$	виражаємо імплікацію
$= \neg p \vee (q \to r) = \neg p \vee (\neg q \vee r)$	виражаємо імплікацію

Перетворення правої частини:

$(p \wedge q) \to r =$	виражаємо імплікацію
$= \neg(p \wedge q) \vee r =$	заносимо "$\neg$" за законом де Моргана
$= (\neg p \vee \neg q) \vee r = \neg p \vee (\neg q \vee r)$	переставляємо дужки за асоціативністю "$\vee$"

Ліва та права частини успішно перетворені до однакового вигляду; отже, вирази справді тотожньо рівні.

Приклад 2. Спростити $(a \wedge b) \vee (\neg b \wedge c) \vee (\neg a \wedge b) \vee (\neg b \wedge \neg c)$.

$(a \wedge b) \vee (\neg b \wedge c) \vee (\neg a \wedge b) \vee (\neg b \wedge \neg c) =$	переставимо внутрішні підвирази (за асоціативністю "$\vee$")
$= \big((a \wedge b) \vee (\neg a \wedge b)\big) \vee \big((\neg b \wedge c) \vee (\neg b \wedge \neg c)\big) =$	винесемо (згідно дистрибутивності "$\wedge$" відносно "$\vee$") "b" та "$\neg b$"
$= \big((a \vee \neg a) \wedge b\big) \vee \big(\neg b \wedge (c \vee \neg c)\big) =$	за законом виключення третього
$= (1 \wedge b) \vee (\neg b \wedge 1) =$	за властивістю одиниці
$= b \vee \neg b = 1.$	за законом виключення третього

1.3 Нормальні форми

Для логічних виразів характерно, що може бути багато різних способів записати по суті одну й ту саму залежність. Інакше кажучи, багато різних формул можуть мати одну й ту саму таблицю істинності. Тому іноді з'являється потреба зменшити це розмаїття, наклавши додаткові обмеження. Са́ме для цього і використовують т. зв. *нормальні форми*.

1.3.1 ДНФ, КНФ, ДДНФ, ДКНФ

ДНФ, КНФ *Елементарна кон'юнкція* (*ЕК*; рос. «*элементарная конъюнкция*», «*ЭК*»; англ. «*elementary conjunction*», «*EC*») — це вираз, який може містити лише операції "$\wedge$" (кон'юнкції) та "$\neg$" (заперечення), причому "$\neg$" можуть стосуватися лише окремих змінних.

(*Не* вимагається, щоб ЕК містили обидві операції "$\neg$" та "$\wedge$"; може бути лише одна з них і навіть жодної. Вимагається, щоб не було ніяких інших операцій, крім "$\neg$" та "$\wedge$".)

Приклади ЕК: "$\neg x_1 x_2 x_3$"; "$x_1 x_2 x_3$"; "$\neg x_1$"; "x_1".

Аналогічно, *елементарна диз'юнкція* (*ЕД*; рос. «*элементарная дизъюнкция*», «*ЭД*»; англ. «*elementary disjunction*», «*ED*») — це вираз, який може містити лише операції "$\vee$" (диз'юнкції) та "$\neg$" (заперечення), причому "$\neg$" можуть стосуватися лише окремих змінних.

(Аналогічно, вимагається не присутність обох операцій, а відсутність інших.)

Диз'юнктивна нормальна форма (*ДНФ*; рос. «*дизъюнктивная нормальная форма*», «*ДНФ*»; англ. «*disjunctive normal form*», «*DNF*») — це вираз, який являє собою або ЕК, або результат з'єднання кількох ЕК диз'юнкціями.

Приклади ДНФ: $\underline{x_1} \vee \underline{x_2}$; $\underline{x_1 x_2 x_3}$; $\underline{x_1 \neg x_2} \vee \underline{x_2 \neg x_3}$; $\underline{x_1} \vee \underline{x_2 x_3} \vee \underline{\neg x_1 \neg x_2 \neg x_3}$ (знизу підкреслено окремі ЕК, з яких складаються ці ДНФ; ніякого іншого смислу підкреслення не мають, і зроблені суто заради ілюстрації означення).

(Значок "$\wedge$" при записі ДНФ прийнято пропускати, подібно до того, як пропускають значок множення у «звичайній» числовій алгебрі.)

Приклади логічних виразів, котрі не є ДНФ:

1. $\neg(x_1 x_2)$ — не ДНФ, бо "$\neg$" стосується не змінної, а підвиразу "$x_1 \wedge x_2$";
2. $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$ — не ДНФ, бо диз'юнкції та кон'юнкція виконуються не в тому порядку, який дозволений правилами ДНФ.

Аналогічно, *кон'юнктивна нормальна форми* (*КНФ*; рос. «*конъюнктивная нормальная форма*», «*КНФ*»; англ. «*conjunctive normal form*», «*CNF*») — це вираз, який являє собою або ЕД, або результат з'єднання кількох ЕД кон'юнкціями.

Приклади КНФ: $\underline{x_1 \vee x_2}$; $\underline{x_1} \wedge \underline{x_2} \wedge \underline{x_3}$; $\underline{(x_1 \vee \neg x_2)} \wedge \underline{(x_2 \vee \neg x_3)}$; $\underline{x_1} \wedge \underline{x_2} \wedge \underline{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}$.

(Причому, $x_1 \vee x_2$ та $x_1 \wedge x_2 \wedge x_3$ — і ДНФ, і КНФ одночасно; перший з них як ДНФ складається з окремих ЕК, а як КНФ являє собою єдину ЕД; дру́гий — навпаки.

При записі КНФ значок "$\wedge$" прийнято записувати явно.)

Приклади логічних виразів, котрі не є КНФ:

1. $\neg(x_1 x_2)$ — не КНФ, бо "$\neg$" стосується не змінної, а підвиразу "$x_1 \wedge x_2$";
2. $x_1 \wedge (\neg x_2 \vee \neg x_1 x_3)$ — не КНФ, бо він міг би виявитися КНФ лише як кон'юнкція двох ЕД ("x_1" та решти); ця решта, тобто "$\neg x_2 \vee \neg x_1 x_3$", не є ЕД, бо містить "$\neg x_1 x_3$", що насправді являє собою "$\neg x_1 \wedge x_3$", тобто містить заборонену всере́дині ЕД кон'юнкцію.

ДДНФ, ДКНФ Елементарна кон'юнкція або елементарна диз'юнкція називається *повною*, якщо до неї входять всі змінні, в точності по одному разу (або у вигляді самих змінних, або у вигляді заперечень).

Досконала диз'юнктивна нормальна форма (*ДДНФ*, *досконала ДНФ*; рос. «*совершенная дизъюнктивная нормальная форма*», «*СДНФ*», «*совершенная ДНФ*»; англ. «*canonical disjunctive normal form*», «*CDNF*» або «*perfect disjunctive normal form*», «*PDNF*») — це вираз, який, по-перше, є ДНФ, і, по-друге, всі його ЕК повні.

Досконала кон'юнктивна нормальна форма (*ДКНФ*, *досконала КНФ*; рос. «*совершенная конъюнктивная нормальная форма*», «*СКНФ*», «*совершенная КНФ*»; англ. «*canonical conjunctive normal form*», «*CCNF*» або «*perfect conjunctive normal form*», «*PCNF*») — це вираз, який, по-перше, є КНФ, і, по-друге, всі його ЕД повні.

Приклади ДДНФ: $x_1 \neg x_2 \vee \neg x_1 x_2$; $\neg x_1 x_2 x_3 \vee x_1 \neg x_2 x_3 \vee \neg x_1 \neg x_2 \neg x_3$.

Приклад ДНФ, яка не є ДДНФ: $\neg x_1 \vee x_1 x_2 x_3$ — не ДДНФ, бо функція в цілому залежить від трьох змінних, а перша елементарна кон'юнкція містить лише змінну x_1 (тобто не повна).

Крім того, ДДНФ чи ДКНФ не може багатократно містити ті самі підвирази. Наприклад, $x_1 x_2 \vee \neg x_1 \neg x_2 \vee x_1 x_2$ не є ДДНФ (а $x_1 x_2 \vee \neg x_1 \neg x_2$, де вилучено повторне входження — є).

Таблиці істинності для ДДНФ та ДКНФ Побудуємо таблицю істинності для ДДНФ ($x_1 \neg x_2 \vee \vee \neg x_1 x_2$) звичайними (як для всіх виразів) засобами, а потім проаналізуємо її особливості.

У стовпчиках $x_1\neg x_2$ та $\neg x_1 x_2$ в точності по одній 1. Щоб кон'юнкція $x_1\wedge\neg x_2$ дорівнювала 1, потрібно, щоб одиницями були одночасно і x_1, і $\neg x_2$, а це можливо в точності на одному наборі x_1=1, x_2=0. Аналогічно, $\neg x_1 x_2$ буде одиничкою (лише) при $\neg x_1$=1 та x_2=1, тобто на наборі x_1=0, x_2=1. Са́ме в цих двох рядках ДДНФ в цілому =1, бо зовнішня операція ДДНФ (яка з'єднує ЕК) — диз'юнкція.

Це не збіг обставин, а властивості, спричинені структурою ДДНФ. *Для будь-якої ДДНФ,* кожен стовпчик, відповідний повній ЕК, містить рівно одну 1, а вся ДДНФ =1 в тих і тільки тих рядках, де одна з ЕК =1.

x_1	x_2	$\neg x_1$	$\neg x_2$	$x_1\wedge\neg x_2$	$\neg x_1\wedge x_2$	$x_1\neg x_2\vee\neg x_1 x_2$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

Будувати таблицю істинності ДДНФ значно легше, ніж таблицю довільної функції. Наприклад, $\neg x_1 x_2 x_3\vee x_1\neg x_2 x_3\vee$ $\vee\neg x_1\neg x_2\neg x_3$: ЕК $\neg x_1 x_2 x_3$ =1 на наборі 0 1 1, ЕК $x_1\neg x_2 x_3$ на наборі 1 0 1, ЕК $\neg x_1\neg x_2\neg x_3$ на 0 0 0. Отже, уся ДДНФ=1 рівно на трьох згаданих рядках.

$x_1x_2x_3$	$\neg x_1 x_2 x_3$	$x_1\neg x_2 x_3$	$\neg x_1\neg x_2\neg x_3$	уся ДДНФ
0 0 0	0	0	1	1
0 0 1	0	0	0	0
0 1 0	0	0	0	0
0 1 1	1	0	0	1
1 0 0	0	0	0	0
1 0 1	0	1	0	1
1 1 0	0	0	0	0
1 1 1	0	0	0	0

Для ДКНФ аналогічно, але роль 1 виконує 0: стовпчики, відповідні повним ЕД, містять по одному 0, вся ДКНФ=0 в тих рядках, де одна з ЕД=0.

Побудуємо таблицю істинності ДКНФ $(\neg x_1\vee\neg x_2\vee x_3)\wedge(x_1\vee\neg x_2\vee x_3)$. ЕД $\neg x_1\vee\neg x_2\vee$ $\vee\, x_3=0$ лише при $\neg x_1=\neg x_2=x_3=0$, тобто на наборі 1 1 0, а $x_1\vee\neg x_2\vee x_3=0$ лише при $x_1=\neg x_2=x_3=0$, тобто на 0 1 0. Отже, уся ДКНФ=0 рівно на цих двох рядках.

$x_1\,x_2\,x_3$	$\neg x_1\vee\neg x_2\vee x_3$	$x_1\vee\neg x_2\vee x_3$	уся ДКНФ
0 0 0	1	1	1
0 0 1	1	1	1
0 1 0	1	0	0
0 1 1	1	1	1
1 0 0	1	1	1
1 0 1	1	1	1
1 1 0	0	1	0
1 1 1	1	1	1

Усі ці властивості дають ключ ще й до оберненої задачі — побудови аналітичного вигляду (формул) ДДНФ та ДКНФ за заданою таблицею істинності. Щоб побудувати ДДНФ за таблицею істинності, розглядаємо всі рядки зі значенням 1, знаходимо відповідні їм повні ЕК та записуємо їх усі через "$\vee$". Аналогічно, щоб побудувати ДКНФ — розглядаємо рядки зі значенням 0, знаходимо відповідні повні ЕД та записуємо через "$\wedge$".

Приклад. Побудуємо ДДНФ та ДКНФ для $f(x,y,z)$, що має таблицю 00001101 (згори донизу). Для цього випишемо навпроти кожної одиниці повну ЕК, котра набуває значення 1 лише на цьому наборі, навпроти кожного ноля — повну ЕД, що набуває значення 0 лише на цьому наборі.

$x\ y\ z$	$f(x,y,z)$	ЕК	ЕД
0 0 0	0		$x\vee y\vee z$
0 0 1	0		$x\vee y\vee\neg z$
0 1 0	0		$x\vee\neg y\vee z$
0 1 1	0		$x\vee\neg y\vee\neg z$
1 0 0	1	$x\neg y\neg z$	
1 0 1	1	$x\neg yz$	
1 1 0	0		$\neg x\vee\neg y\vee z$
1 1 1	1	xyz	

Наприклад, $f(0,0,0)=0$ — значить, для набору 0 0 0 треба будувати ЕД; щоб 0 виходив лише при $x=y=z=0$, це повинна бути $x\vee y\vee z$. Для набору 0 0 1, де $f(0,0,1)$ теж =0, по z треба зробити заперечення, бо са́ме тоді набір 0 0 1 перетвориться у $x=y=\neg z=0$ та у повну ЕД $x\vee y\vee\neg z$. І так далі, тільки для одиниць будуємо ЕК.

Диз'юнкція всіх виписаних повних ЕК дає ДДНФ $f(x,y,z)=x\neg y\neg z\vee xyz\vee x\neg yz$, а кон'юнкція всіх виписаних повних ЕД дає ДКНФ $f(x,y,z)=(x\vee y\vee z)\wedge(x\vee\neg y\vee z)\wedge(x\vee\neg y\vee\neg z)\wedge(x\vee\neg y\vee z)\wedge$ $\wedge(\neg x\vee\neg y\vee z)$.

(Можливість легко переходити як від виразу до таблиці істинності, так і від таблиці до виразу — одна з причин, чому ДДНФ та ДКНФ називають «досконалими». Але *не* слід вважати, ніби ДДНФ чи ДКНФ справді досконалі абсолютно в усіх смислах. Наприклад, щойно побудовані ДДНФ та ДКНФ виявилися досить громіздкими, тоді як значно коротша «не досконала» КНФ $x\wedge(\neg y\vee z)$ має ту саму таблицю істинності…)

Єдиність ДДНФ та ДКНФ З попереднього розділу можна зробити очевидний висновок: *для будь-якої функції існує єдина ДДНФ і єдина ДКНФ (з точністю до порядку елементарних кон'юнкцій чи диз'юнкцій).*

(Пояснимо словосполучення «з точністю до порядку». Наприклад, останню згадану ДДНФ можна записати у вигляді $xyz \vee x\neg yz \vee x\neg y\neg z$; це не співпадає з попереднім записом буква в букву, але відрізняється *лише* порядком елементарних кон'юнкцій.)

Якщо функція на *всіх* наборах змінних набуває значення 0, умовно вважають, що її ДДНФ має вигляд "0" (хоча це й суперечить основному означенню). Аналогічно, ДКНФ-ом тотожньої одиниці умовно вважають "1".

1.3.2 Полінóм Жегалкіна

Полінóм Жегалкіна (рос. «*полинóм Жегалкина*», англ. «*Zhegalkin polynomial*») — це спосіб запису булевих функцій, де дозволяються лише дії "$\wedge$" (кон'юнкція, вона ж множення) та $\oplus$ (ксор, він же додавання за модулем 2); крім того, дозволяється використання константи 1 (лише у вигляді "$\ldots \oplus 1$"). Як і у «звичайних» поліномах (многочленах), додавання (ксори) повинні бути зовнішніми операціями, множення (кон'юнкції) — внутрішніми.

Приклади поліномів Жегалкіна: $x_1 \oplus x_2$; $x_1 \oplus x_2 \oplus 1$; $x_1x_2 \oplus x_1 \oplus x_2$.

Приклади формул, що не є поліномами Жегалкіна:

1. $\neg x_1$ — не поліном Жегалкіна, бо використане заперечення, котре не входить до переліку допустимих операцій;
2. $(x_1 \oplus 1)x_2$ — не поліном Жегалкіна, бо "$\oplus$" не зовнішня операція (після неї виконується кон'юнкція).

(Аналогічно ДДНФ, для функції, яка на всіх наборах змінних набуває значення 0, умовно вважають, що її поліном Жегалкіна має вигляд "0".)

Тотожності для "$\oplus$" та "$\wedge$" (Всі вони відомі з розд. 1.2.)

комутативності (обидві)	$a \oplus b = b \oplus a, \qquad ab = ba;$
асоціативності (обидві)	$(a \oplus b) \oplus c = a \oplus (b \oplus c), \quad (ab)c = a(bc);$
дистрибутивність (одна, як і для числової алгебри)	$(a \oplus b)c = ac \oplus bc;$
	$a \oplus 0 = a;$
	$a \oplus a = 0.$

Побудова полінома Жегалкіна методом перетворень ДДНФ У ДДНФ використовують операції "$\wedge$", "$\vee$" та "$\neg$"; потрібно перейти до полінома Жегалкіна, де використовують операції "$\wedge$", "$\oplus$" та константу 1. Отже, потрібно виразити "$\vee$" та "$\neg$" через "$\wedge$", "$\oplus$" та "1". "$\neg$" можна виразити як $\neg x = x \oplus 1$; отже, лишається тільки розібратися з операцією "$\vee$".

Завдяки особливій структурі ДДНФ, на будь-якому наборі змінних або абсолютно всі повні елементарні кон'юнкції набувають значення 0, або якась одна набуває значення 1, а решта — значення 0; різні повні ЕК не можуть дорівнювати 1 одночасно. Водночас, $a \oplus b$ відрізняється від $a \vee b$ *лише* при $a = b = 1$, а на інших наборах вони однакові.

Отже, в ДДНФ можна позамінятиˆ всі операції "$\vee$" на "$\oplus$".

(Повторюємо: це правильно *завдяки* особливій структурі ДДНФ.)

Таким чином вдається отримати вираз, де є лише операції "$\wedge$", "$\oplus$" та константи 1. Потім його потрібно перетворити до вигляду полінома Жегалкіна, розкриваючи дужки (згідно тотожності $(a \oplus b)c = ac \oplus bc$) та зводячи однакові доданки (згідно тотожності $a \oplus a = 0$).

Наприклад, процес перетворення ДДНФ $\neg x_1\neg x_2\neg x_3 \vee \neg x_1x_2x_3 \vee x_1\neg x_2x_3$ буде таким. Замінивши усі $\neg x_i$ як $(x_i \oplus 1)$, отримаємо $(x_1 \oplus 1)(x_2 \oplus 1)(x_3 \oplus 1) \oplus (x_1 \oplus 1)x_2x_3 \oplus x_1(x_2 \oplus 1)x_3$. Розкривши дужки, отримаємо $(x_1x_2x_3 \oplus x_1x_2 \oplus x_1x_3 \oplus x_1 \oplus x_2x_3 \oplus x_2 \oplus x_3 \oplus 1) \oplus (x_1x_2x_3 \oplus x_2x_3) \oplus (x_1x_2x_3 \oplus x_1x_3)$. Повиділяємо пари однакових доданків: $\underline{x_1x_2x_3} \oplus x_1x_2 \oplus \underline{x_1x_3} \oplus x_1 \oplus \underline{\underline{x_2x_3}} \oplus x_2 \oplus x_3 \oplus 1 \oplus \underline{x_1x_2x_3} \oplus \oplus \underline{\underline{x_2x_3}} \oplus x_1x_2x_3 \oplus \underline{x_1x_3}$. Вони позводяться (завдяки комутативності, асоціативності та тотожностям $a \oplus a = 0$, $a \oplus 0 = a$), і залишиться остаточна відповідь: $x_1x_2x_3 \oplus x_1x_2 \oplus x_1 \oplus x_2 \oplus x_3 \oplus 1$.

Побудова полінома Жегалкіна методом невизначених коефіцієнтів *Метод невизначених коефіцієнтів* (рос. «*метод неопределённых коэффициентов*», англ. «*method of undetermined coefficients*») — загальна ідея, що використовується у різних розділах математики, включаючи

алгебру та диференціальні рівняння. Суть така: щоб знайти аналітичний вигляд функції, вона спочатку записується із використанням якихось, поки що невідомих, коефіцієнтів; потім ці коефіцієнти підбирають так, щоб вийшла потрібна функція. Для більшості галузей математики, підбір коефіцієнтів зводиться до розв'язування системи рівнянь. І конкретно для полінома Жегалкіна ці рівняння дуже прості, й аналіз кожного рядка таблиці істинності (в порядку згори донизу) дозволяє знаходити значення одного нового коефіцієнта.

Побудуємо поліном Жегалкіна для імплікації $x_1 \to x_2$.

x_1	x_2	$x_1 \to x_2$
0	0	1
0	1	1
1	0	0
1	1	1

Оскільки маємо функцію від двох змінних, загальним виглядом функції (з ще не визначеними коефіцієнтами) є $f(x_1,x_2)=ax_1x_2\oplus b_1x_1\oplus b_2x_2\oplus c$, де кожен із коефіцієнтів a, b_1, b_2, c буде замінено або на 1, або на 0.

У са́мому верхньому рядку таблиці істинності записано, що при $x_1=x_2=0$ функція набуває значення $f(0,0)=1$. Підставивши це у загальний вигляд, отримаємо: $a\cdot0\cdot0\oplus b_1\cdot0\oplus b_2\cdot0\oplus c=1$. Ми (ще) не знаємо ні a, ні b_1, ні b_2, але вони все одно множаться на 0, тож у лівій частині рівності лишається тільки c. Тобто, приходимо до рівності $c=1$, і надалі будемо замість c підставляти знайдене значення 1.

Далі написано, що $f(0,1)=1$. Підставивши це у загальний вигляд, отримаємо: $a\cdot0\cdot1\oplus b_1\cdot0\oplus\oplus b_2\cdot1\oplus1=1$, тобто $b_2\oplus1=1$. Легко бачити, що єдиним розв'язком цього рівняння є $b_2=0$.

Далі написано$f(1,0)=0$. Підставивши це (а також відомі $c=1$, $b_2=0$) у загальний вигляд, отримаємо: $a\cdot1\cdot0\oplus b_1\cdot1\oplus0\cdot0\oplus1=0$, тобто $b_1\oplus1=0$. Легко бачити, що єдиним розв'язком цього рівняння є $b_1=1$.

У останньому рядку записано, що $f(1,1)=1$. Підставивши це та все вже відоме у загальний вигляд, отримаємо: $a\cdot1\cdot1\oplus1\cdot1\oplus0\cdot1\oplus1=1$, тобто $a\oplus1\oplus1=1$. Звідси за тотожністю $1\oplus1=0$ отримуємо $a\oplus0=1$, тобто $a=1$.

Остаточно, підставивши знайдені значення коефіцієнтів, отримуємо поліном Жегалкіна $x_1x_2\oplus x_1\oplus1$.

Ще один приклад: функція 10010100. Вона від трьох змінних, тому загальний вигляд полінома Жегалкіна такий:

$ax_1x_2x_3\oplus b_{12}x_1x_2\oplus b_{13}x_1x_3\oplus b_{23}x_2x_3\oplus c_1x_1\oplus c_2x_2\oplus c_3x_3\oplus d.$

x_1	x_2	x_3		a	b_{12}	b_{13}	b_{23}	c_1	c_2	c_3	d			
0	0	0	1	0⊕	0 ⊕	0 ⊕	0 ⊕	0 ⊕	0 ⊕	0 ⊕	d	=	1	$d=1$
0	0	1	0	0⊕	0 ⊕	0 ⊕	0 ⊕	0 ⊕	0 ⊕	c_3 ⊕	1	=	0	$c_3=1$
0	1	0	0	0⊕	0 ⊕	0 ⊕	0 ⊕	0 ⊕	c_2 ⊕	0 ⊕	1	=	0	$c_2=1$
0	1	1	1	0⊕	0 ⊕	0 ⊕	b_{23} ⊕	0 ⊕	1 ⊕	1 ⊕	1	=	1	$b_{23}=0$
1	0	0	0	0⊕	0 ⊕	0 ⊕	0 ⊕	c_1 ⊕	0 ⊕	0 ⊕	1	=	0	$c_1=1$
1	0	1	1	0⊕	0 ⊕	b_{13} ⊕	0 ⊕	1 ⊕	0 ⊕	1 ⊕	1	=	1	$b_{13}=0$
1	1	0	0	0⊕	b_{12} ⊕	0 ⊕	0 ⊕	1 ⊕	1 ⊕	0 ⊕	1	=	0	$b_{12}=1$
1	1	1	0	a ⊕	1 ⊕	0 ⊕	0 ⊕	1 ⊕	1 ⊕	1 ⊕	1	=	0	$a=1$

Відповідь: $x_1x_2x_3\oplus x_1x_2\oplus x_1\oplus x_2\oplus x_3\oplus1$.

Єдиність полінома Жегалкіна Ми розглянули два методи побудови полінома Жегалкіна. Кожен з них чіткий і однозначний — отже, внаслідок (правильного) застосування конкретного методу до конкретної таблиці істинності може бути отриманий тільки один поліном Жегалкіна. А чи можливо, щоб поліноми, отримані з однієї таблиці істинності цими різними методами, були різними? А якщо застосувати ще якийсь третій метод?.. Виявляється (що зовсім не очевидно), *для будь-якої бу́левої функції існує* ***єдиний*** *поліном Жегалкіна (з точністю до порядку доданків).*

Доведення. (При першому прочитанні рекомендується переглянути побіжно, а потім ознайомитися ще раз після вивчення розд. 4.1 та 4.2.5.)

Кількість можливих доданків n-арного полінома Жегалкіна (таких, як "x_2", "$x_1x_3x_4$", тощо, включно з доданком "1") становить 2^n (бо кожна з n змінних може або входити (як множник) до доданку, або ні; коли жодна не входить, утворюється "1"). Кожен з цих 2^n доданків може або входити до конкретного полінома, або ні. Значить, кількість різних n-арних поліномів Жегалкіна становить 2^{2^n}.

З іншого боку, кількість різних n-арних булевих функцій (таблиць істинності) теж становить 2^{2^n} (бо є 2^n рядків таблиці істинності, кожен може набувати одне з двох значень). Отже (враховуючи, що для кожної функції існує поліном Жегалкіна), нема «зайвих» поліномів, що могли б, по кілька різних, відповідати одній і тій самій функції. ■

1.4 Мінімізація булевих функцій

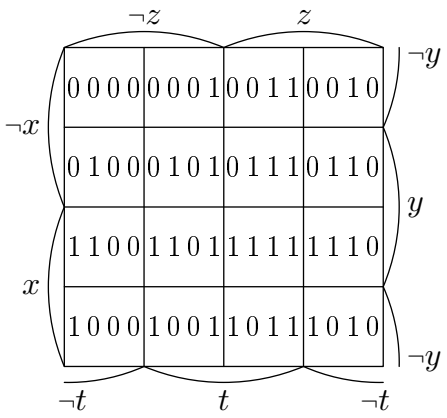
Взагалі кажучи, «*мінімізація*» (рос. «*минимизация*», англ. «*minimization*») означає «зменшення». У застосуванні са́ме до булевих функцій, задача мінімізації ставиться так. Є деяка булева функція (задана формулою або таблицею істинності). Потрібно знайти таку формулу, щоб вона, по-перше, задавала ту са́му функцію, і, по-дру́ге, містила якомога менше операцій.

(Іноді розглядають «задачу мінімізації у широкому смислі», коли для кожного примірника задачі окремо задають і функцію, і перелік операцій, через які виразити цю функцію. Але для такої широкої задачі не відомі хоч скільки-небудь зручні методи.)

Зазвичай розглядають *мінімізацію у класі ДНФ*, тобто шукають якнайкоротшу ДНФ (не хвилюючись з приводу того, що якась формула, котра не є ДНФ, може виявитися ще коротшою). Як правило, при підрахунку довжини́ ДНФ рахують тільки сумарну кількість кон'юнкцій та диз'юнкцій (а заперечення не рахують).

1.4.1 Карти Карно

Карта Карно (або *Карнау*, вона ж *діаграма Вейча*; рос. «*карта Карно*» (*Карнау*), «*диаграмма Вейча*», англ. «*Karnaugh map*», «*K-map*») є таблицею зі спеціальним чином розміщеними $2^4 = 16 = 4\times4$ клітинками.

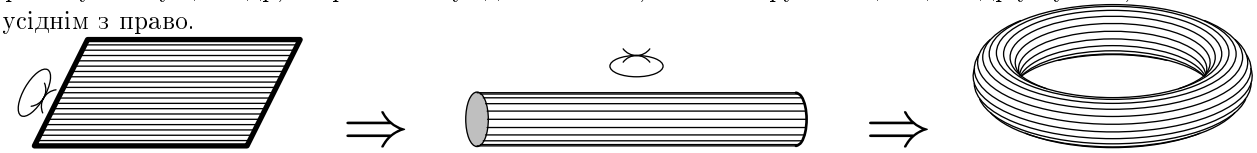


(Не плутати діаграми Вейча з діаграмами Венна, котрі іноді називають також кругами Ейлера (розд. 2.1.4).)

Верхні два рядки підписані "$\neg x$", нижні два "x". Це означає, що у верхніх двох рядках розміщені ті клітинки, в яких x (1-а змінна) дорівнює 0, у нижніх двох $x = 1$. Аналогічно, у лівих двох стовпчиках z (3-я змінна) дорівнює 0, у правих двох $z = 1$. Аналогічно, у крайніх (рядках/стовпчиках) (y/t) ((2-а/4-а) змінна) дорівнює нулю, у внутрішніх — одиниці.

Будемо називати клітинки карти Карно *сусідніми*, коли вони або мають спільну сторону (а не лише вершину), або це верхня та нижня клітинки одного стовпчика, або це ліва та права клітинки одного рядка.

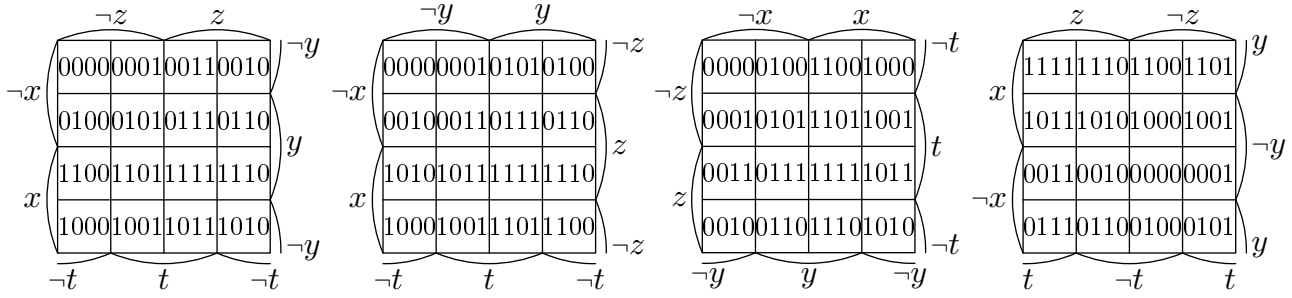
(Таке сусідство можливе, але не на площині чи сфері, а на *то́рі* (поверхні бублика). Зігнемо плоский прямокутник у циліндр, і верх стане сусіднім з низом; потім закрутимо цей циліндр у бублик, і ліво стане сусіднім з право.



Але це був так, ліричний відступ. Для задачі мінімізації це не суттєво...)

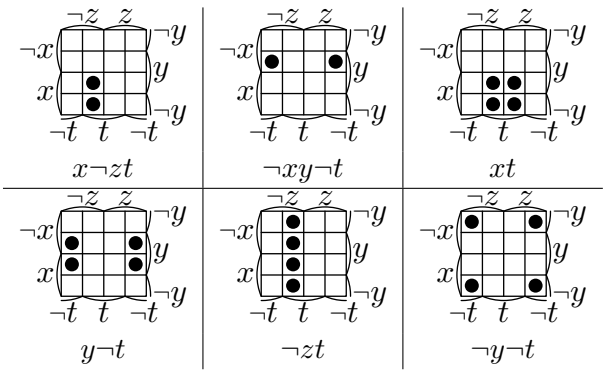
Основна властивість карти Карно: *клітинки сусідні тоді й тільки тоді, коли записані у них набори відрізняються в точності в одному розряді.*

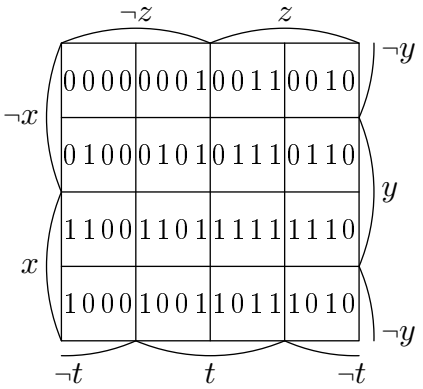
(Таку властивість має *не лише* вищенаведене заповнення таблиці, а й деякі інші: можна переставляти місцями змінні та/або міняти місцями, в яких рядках/стовпчиках є заперечення і в яких нема; всього $4! \times 2^4 = 24\times16 = 384$ варіанти, 4 з яких наведено тут:



Щоб не створювати непорозуміння, рекомендуємо дотримуватися того (поширеного у багатьох джерелах) варіанта, який був наведений раніше і тут повторений як перший; але інші все ж бувають.)

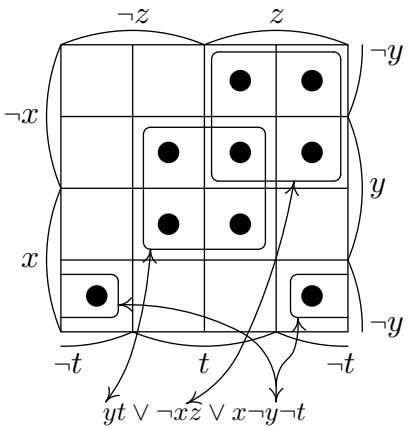
Розглянемо ЕК (елементарну кон'юнкцію) від 3-х змінних $x\neg zt$. Вона=1, коли $x=\neg z=t=1$, а змінна y може набувати будь-яке значення. Тобто, одиниця буде на наборах 1 0 0 1 та 1 1 0 1, які відрізняються лише одним розрядом, і тому́ розміщені у двох сусідніх клітинках. Аналогічно, ЕК $\neg xy\neg t$ набуває значення 1 при x=0, y=1, t=0 та довільному z; ці набори теж відповідають сусіднім («через край») клітинкам.





ЕК, що містить дві змінні, істинна при фіксованих значеннях цих двох змінних і довільних значеннях решти двох; тобто, на 4-х наборах; отже, така ЕК утворює на карті Карно область розміром 4 (2×2, або 4×1, або 1×4).

Нарешті, якщо (раптом) зустрінеться ЕК, що містить єдину змінну, то їй відповідають вже згадані два сусідні рядки або два сусідні стовпчики.



Тепер зобразимо на карті Карно область істинності ДНФ (диз'юнкції кількох ЕК). Кожній ЕК відповідає прямокутна область; їхній диз'юнкції — об'єднання цих областей.

Наприклад, для ДНФ $yt\vee\neg xz\vee x\neg y\neg t$ виходить дві області 2×2 (які частково перекриваються) та одна область 1×2 (з'єднана через край).

Повернемось до основної задачі — знаходження мінімальної ДНФ, що задає потрібну функцію. Для її розв'язання, перш за все позначають на карті Карно клітинки з наборами, на яких функція набуває значення 1. А потім підбирають таку сукупність прямокутних областей (елементарних кон'юнкцій), що їхнє об'єднання (диз'юнкція) покриває (дає одиницю) в точності потрібні клітинки (на потрібних наборах).

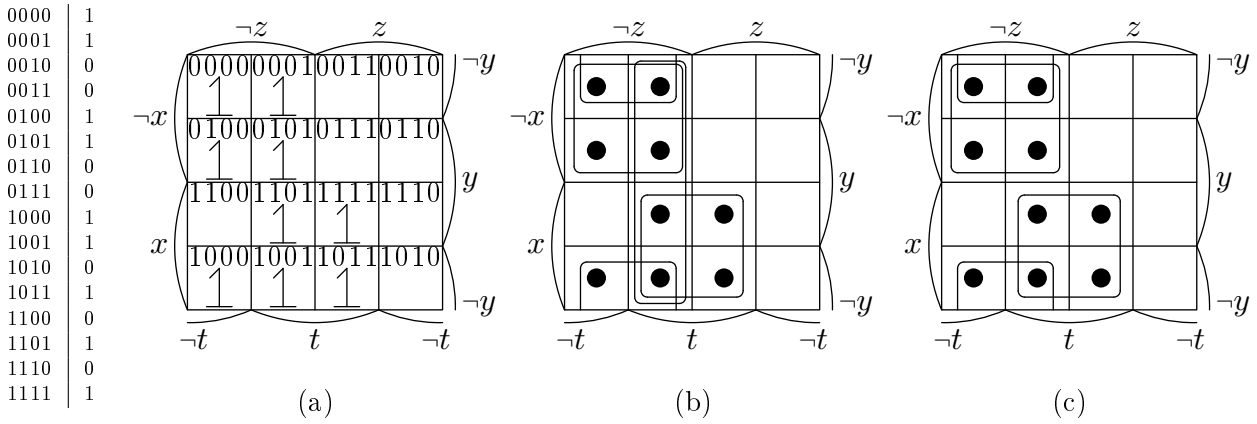
Як правило, при побудові сукупності областей слід міркувати так:

1. У першу чергу слід намагатися виділити якнайбільші області.
   (Тобто, спочатку 4×2 або 2×4 (якщо такі є), потім 2×2, 4×1 або 1×4, і т. д. Це доцільно, бо більша область одночасно і покриває більше клітинок, і відповідає коротшій елементарній кон'юнкції.)
2. Ситуація, коли одна й та сама клітинка потрапляє відразу до кількох областей, допустима і часто навіть корисна.
   (Скажімо, у прикладі області yt та $\neg xz$ перекривалися. Якби ці са́мі 7 клітинок покривали областями, що не перекриваються, вийшло би три області, ро́змірами 2×2, 2×1 та 1×1, і сумарна кількість операцій збільшилась би з 3-х до 8-ми.)
3. У відповіді не повинно бути областей, які покривають *лише* клітинки, що і так покриті іншими областями.
   (Такі області можна просто вилучити, внаслідок чого з ДНФ зникне відповідна ЕК; ДНФ від цього стане гарантовано меншою.)

Але лише «як правило». Хоча б тому́, що останні два пункти дещо суперечливі й ніяк не є чітким алгоритмом.

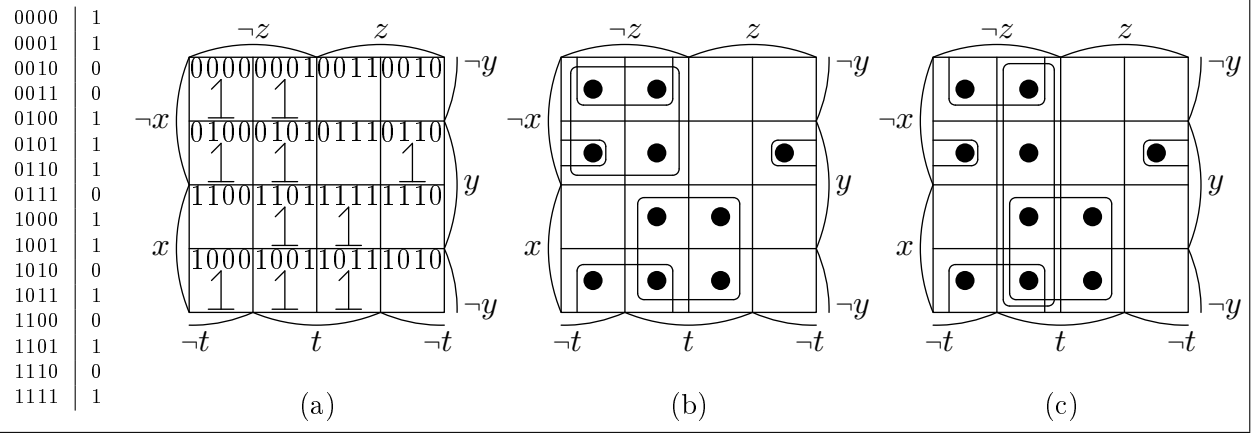
Тобто, у частині випадків все ж доводиться займатися перебором: розглядати різні покриття й дивитися, яке з них забезпечує меншу кількість операцій у ДНФ. Але *такий* перебір, порівнюючи з іншими методами, все-таки досить зручний і більш-менш швидкий.

Розглянемо мінімізацію функції 1100110011010101 (тобто функції, що має таблицю істинності, наведену на рис.; вжитий скорочений запис містить стовпчик значень у порядку згори донизу).

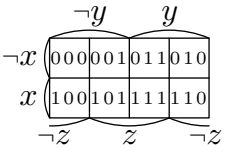


Спочатку розмістимо одиничи на карті Карно (рис. (a)). Бачимо, що областей розміром 8 нема, тож переходимо до виділення областей розміром 4. Такими областями є (рис. (b)): $\neg zt$ (2-ий стовпчик); $\neg x\neg z$ (зліва вгорі); $\neg y\neg z$ (ліворуч, сусідні через край верх та низ); xt (внизу посере́дині). Цим самим ми покрили всі (потрібні) клітинки, на кожному кроці покривали хоча б одну нову... і, не зважаючи на це, найперша область ($\neg zt$, вона ж 2-ий стовпчик) покриває лише ті клітинки, які й так покриті іншими областями. Отже, остато́чне покриття зображене на рис. (c), і йому відповідає мінімальна ДНФ $\neg x\neg z \vee xt \vee \neg y\neg z$.

Задача мінімізації в принципі може мати різні правильні відповіді.
Наприклад, для функції 1100111011010101 маємо дві різні ДНФ однакової мінімальної довжини́ 8 (нагадаємо, що заперечення не рахуються): $\neg x\neg z \vee xt \vee \neg y\neg z \vee \neg xy\neg t$ (рис. (b)) та $\neg zt \vee \vee xt \vee \neg y\neg z \vee \neg xy\neg t$ (рис. (c)).



Узагальнення карт Карно на іншу кількість змінних На основі тих самих ідей легко побудувати карту Карно для функцій від трьох змінних (див. рис.).



А узагальнювати карти Карно на функції від більшої кількості змінних незручно. Для 5-ти та 6-ти змінних є вибір: або втратити основну властивість цих карт «комірки сусідні ⇔ набори відрізняються по одній змінній», або зберегти її за рахунок виходу у третій вимір: розмістити два (для функцій від 5 змінних) або чотири (від 6) примірники квадратів 4×4 оди́н над о́дним. Для ще більшої кількості — без варіантів, лише втратити основну властивість. А коли втрачається властивість, втрачається і смисл карти.

1.4.2 Метод Квайна

Метод Квайна[10] (точніше кажучи, «методу Квайна мінімізації у класі ДНФ», бо Квайн розробив також інші методи для інших задач; один з них розглянутий у розд. 1.5.1) — інший метод розв'язування тієї ж задачі мінімізації у класі ДНФ. З причин, пояснених далі, його називають також методом Квайна–Мак-Класкі. Для ручного застосування на папері він менш зручний, чим карти Карно. Але він і зручніший для програмування, і може бути застосований до булевих функцій від довільної кількості змінних. Метод складається з двох етапів.

Перший етап, формулювання На першому етапі методу Квайна багатократно застосовують у спеціальному порядку т. зв. *склеювання ЕК* (рос. «*склеивание ЭК*», англ. «*combining EC*»). Склеювання ЕК базується на тотожності $AB \vee A\neg B = A$ (її доведення очевидне: винести A за дужки і використати $B \vee \neg B = 1$), де в якості B завжди береться окрема змінна, а в якості A, як правило, використовуються складніші вирази. Наприклад, $x_1\neg x_2x_3x_4 \vee x_1\neg x_2\neg x_3x_4$ можна склеїти, якщо в якості B взяти x_3, відповідно в якості A буде $x_1\neg x_2x_4$; отримаємо $x_1\neg x_2x_3x_4 \vee \vee x_1\neg x_2\neg x_3x_4 = x_1\neg x_2x_4$.

Легко бачити (за ідемпотентністю $x \vee x = x$), що $AB \vee A\neg B \vee A$ теж дорівнює A, тому неважливо, чи зразу замінювати $AB \vee A\neg B$ на A, чи спочатку на $AB \vee A\neg B \vee A$, а вже потім на A.

Починати застосування методу Квайна треба з функції, *записаної у вигляді* ***ДДНФ***. (Якщо порушити цю вимогу, застосовувавши метод до не досконалої ДНФ, результат матиме правильну таблицю істинності, але зовсім не обов'язково буде мінімальним.)

Спочатку треба провести можливі склеювання повних ЕК (від усіх n змінних); при цьому, якщо одна й та ж ЕК може взяти участь у кількох різних склеюваннях — вона *повинна* взяти участь в них *усіх*. Коли етап склеювань повних ЕК завершено, ті ЕК, які взяли участь хоча б у одному склеюванні, включені у результат(и) своїх склейок, тому їх можна викреслити, а ті, які не взяли участі в жодному, треба залишити. (Тут є взаємозв'язок з тим, що коли у картах Карно є дві сусідні одиночки, і їх можна обвести областю 1×2 чи 2×1, то обводити кожну з таких одиничок окремою областю 1×1 не треба; а коли одиничка не має жодного сусіда (ні у звичайному смислі, ні через край) й ні з чим не поєднується, її мусимо виділити окремою областю 1×1.)

Потім треба аналогічно пробувати клеїти ЕК від $n-1$ змінних у ЕК від $n-2$ змінних, теж так, щоб кожна ЕК взяла участь в усіх можливих для неї склеюваннях; завершивши ці склеювання, теж треба позакреслювати ті ЕК від $n-1$ змінних, які взяли участь хоча б у одному склеюванні, й залишити ті, які не взяли в жодному. Потім слід аналогічно повторювати спроби склеювань ЕК від $n-2$ змінних у ЕК від $n-3$ змінних, потім спроби склеювань ЕК від $n-3$ змінних у ЕК від $n-4$ змінних, і т. д. Процес обов'язково завершиться, хоча б тому, що довжи́ни окремо взятих ЕК від склеювань зменшуються, що не може тривати вічно. Але зазвичай процес завершується, коли жодна пара поточних ЕК не склеюється.

Мак-Класкі (E. J. McCluskey) помітив, що раз склеювання можливе лише між такими ЕК, одна з яких містить рівно на одне заперечення більше, ніж інша, то варто починати метод Квайна з групування ЕК за кількостями заперечень (група №0 або порожня, або містить єдину повну ЕК без заперечень $x_1x_2\ldots x_n$; група №1 або порожня, або містить усі наявні у ДДНФ повні ЕК з рівно одним запереченням; і т. д.). Якщо це зроблено, то достатньо пробувати склеювати лише всі можливі пари ЕК з сусідніх груп; якщо ні — слід пробувати склеювати взагалі кожну ЕК з кожною ЕК, що призводить до того ж результату, але з більшою кількістю невдалих спроб.

Перший етап, пояснення на прикладі Повторимо всю суть першого етапу ще раз, іншими словами, з ілюструванням конкретним прикладом: функцією, ДДНФ якої дорівнює $\neg x_1x_2\neg x_3\neg x_4 \vee \vee \neg x_1x_2\neg x_3x_4 \vee \neg x_1x_2x_3\neg x_4 \vee \neg x_1x_2x_3x_4 \vee x_1x_2\neg x_3\neg x_4 \vee x_1x_2x_3\neg x_4 \vee x_1x_2x_3x_4$.

Щоб користуватися надалі оптимізацію Мак-Класкі, розіб'ємо на групи згідно кількості запереченень (група 0 — ЕК без заперечень, група 1 — ЕК з 1-им "¬", група 2 — ЕК з 2-ма "¬", і т. д.).	$(x_1x_2x_3x_4) \vee (x_1x_2x_3\neg x_4 \vee \vee \neg x_1x_2x_3x_4) \vee (\neg x_1x_2\neg x_3x_4 \vee \vee \neg x_1x_2x_3\neg x_4 \vee x_1x_2\neg x_3\neg x_4) \vee \vee (\neg x_1x_2\neg x_3\neg x_4)$.

[10] це прізвище в оригіналі пишеться Quine і читається [kwain]; в україно- та російськомовній літературі зустрічається і транскрипція «*Квайн*», і транскрипція «*Куайн*»

Беремо ЕК сусідніх груп (спочатку групи 0 і групи 1) і намагаємося їх склеїти. Записуємо результат склейки, але не прибираємо також і аргументи, до яких було застосоване склеювання.	$(\underline{x_1x_2x_3x_4}) \vee (\underline{x_1x_2x_3\neg x_4} \vee$ $\vee \neg x_1x_2x_3x_4) \vee (\neg x_1x_2\neg x_3x_4 \vee$ $\vee \neg x_1x_2x_3\neg x_4 \vee x_1x_2\neg x_3\neg x_4) \vee$ $\vee (\neg x_1x_2\neg x_3\neg x_4) \vee (\underline{x_1x_2x_3})$.
Продовжуємо склеювання між кон'юнкціями групи 0 і групи 1. Особливу увагу звертаємо на те, що хоча $x_1x_2x_3x_4$ вже було використано в одній склейці, ми намагаємося склеїти його також з іншими кон'юнкціями групи 1.	$(\underline{x_1x_2x_3x_4}) \vee (x_1x_2x_3\neg x_4 \vee$ $\vee \underline{\neg x_1x_2x_3x_4}) \vee (\neg x_1x_2\neg x_3x_4 \vee$ $\vee \neg x_1x_2x_3\neg x_4 \vee x_1x_2\neg x_3\neg x_4) \vee$ $\vee (\neg x_1x_2\neg x_3\neg x_4) \vee (x_1x_2x_3 \vee$ $\vee \underline{x_2x_3x_4})$.
Продовжуємо склеювання, перейшовши до групи 1 і групи 2. Перевіряємо кожну можливу пару: $x_1x_2x_3\neg x_4$ та $\neg x_1x_2\neg x_3x_4$ (їх склеїти не можна), $x_1x_2x_3\neg x_4$ та $\neg x_1x_2x_3\neg x_4$ (склеюються по x_1), $x_1x_2x_3\neg x_4$ та $x_1x_2\neg x_3\neg x_4$ (склеюються по x_3), $\neg x_1x_2x_3x_4$ та $\neg x_1x_2\neg x_3x_4$ (склеюються по x_3), $\neg x_1x_2x_3x_4$ та $\neg x_1x_2x_3\neg x_4$ (склеюються по x_4), $\neg x_1x_2x_3x_4$ та $x_1x_2\neg x_3\neg x_4$ (їх склеїти не можна). Підкреслено кон'юнкції, що взяли участь у *якому-небудь* склеюванні (не лише останньому).	$(\underline{x_1x_2x_3x_4}) \vee (\underline{x_1x_2x_3\neg x_4} \vee$ $\vee \underline{\neg x_1x_2x_3x_4}) \vee (\underline{\neg x_1x_2\neg x_3x_4} \vee$ $\vee \underline{\neg x_1x_2x_3\neg x_4} \vee \underline{x_1x_2\neg x_3\neg x_4}) \vee$ $\vee (\neg x_1x_2\neg x_3\neg x_4) \vee (x_1x_2x_3 \vee$ $\vee x_2x_3x_4) \vee (x_2x_3\neg x_4 \vee$ $\vee x_1x_2\neg x_4 \vee \neg x_1x_2x_4 \vee \neg x_1x_2x_3)$.
Продовжуємо склеювання, перейшовши до групи 2 і групи 3. Перевіряємо кожну можливу пару: $\neg x_1x_2\neg x_3x_4$ та $\neg x_1x_2\neg x_3\neg x_4$ (склеюються по x_4), $\neg x_1x_2x_3\neg x_4$ та $\neg x_1x_2\neg x_3\neg x_4$ (склеюються по x_3), $x_1x_2\neg x_3\neg x_4$ та $\neg x_1x_2\neg x_3\neg x_4$ (склеюються по x_1).	$(\underline{x_1x_2x_3x_4}) \vee (\underline{x_1x_2x_3\neg x_4} \vee$ $\vee \underline{\neg x_1x_2x_3x_4}) \vee (\underline{\neg x_1x_2\neg x_3x_4} \vee$ $\vee \underline{\neg x_1x_2x_3\neg x_4} \vee \underline{x_1x_2\neg x_3\neg x_4}) \vee$ $\vee (\underline{\neg x_1x_2\neg x_3\neg x_4}) \vee (x_1x_2x_3 \vee$ $\vee x_2x_3x_4) \vee (x_2x_3\neg x_4 \vee$ $\vee x_1x_2\neg x_4 \vee \neg x_1x_2x_4 \vee$ $\vee \neg x_1x_2x_3) \vee (\neg x_1x_2\neg x_3 \vee$ $\vee \neg x_1x_2\neg x_4 \vee x_2\neg x_3\neg x_4)$.
На цьому завершено склеювання ЕК від 4 змінних. Тепер потрібно видалити всі підкреслені ЕК (кожна з них взяла участь у склеюванн(і/ях), отже, покривається створеною ЕК від 3 змінних). Далі слід перейти до склеювання ЕК від 3 змінних.	$x_1x_2x_3 \vee x_2x_3x_4) \vee (x_2x_3\neg x_4 \vee$ $\vee x_1x_2\neg x_4 \vee \neg x_1x_2x_4 \vee$ $\vee \neg x_1x_2x_3) \vee (\neg x_1x_2\neg x_3 \vee$ $\vee \neg x_1x_2\neg x_4 \vee x_2\neg x_3\neg x_4)$.
Продовжуємо склеювання, перейшовши до групи 0 і групи 1 серед ЕК від 3 змінних. Перевіряємо кожну можливу пару: $x_1x_2x_3$ та $x_2x_3\neg x_4$ (їх склеїти не можна); $x_1x_2x_3$ та $x_1x_2\neg x_4$ (їх склеїти не можна); $x_1x_2x_3$ та $\neg x_1x_2x_4$ (їх склеїти не можна); $x_1x_2x_3$ та $\neg x_1x_2x_3$ (склеюються по x_1); $x_2x_3x_4$ та $x_2x_3\neg x_4$ (склеюються по x_4, але отримується ЕК x_2x_3, яку вже отримували раніше; отже, ЕК $x_2x_3x_4$ та $x_2x_3\neg x_4$ позначаємо як використані, а результат склейки повторно не пишемо; $x_2x_3x_4$ та $x_1x_2\neg x_4$ (їх склеїти не можна); $x_2x_3x_4$ та $\neg x_1x_2x_4$ (їх склеїти не можна); $x_2x_3x_4$ та $\neg x_1x_2x_3$ (їх склеїти не можна). Нагадаємо, що ми підкреслюємо кон'юнкції, котрі взяли участь в якому-небудь склеюванні.	$(\underline{x_1x_2x_3} \vee \underline{x_2x_3x_4}) \vee (\underline{x_2x_3\neg x_4} \vee$ $\vee x_1x_2\neg x_4 \vee \neg x_1x_2x_4 \vee$ $\vee \underline{\neg x_1x_2x_3}) \vee (\neg x_1x_2\neg x_3 \vee$ $\vee \neg x_1x_2\neg x_4 \vee x_2\neg x_3\neg x_4) \vee$ $\vee (x_2x_3)$.
Продовжуємо склеювання, перейшовши до групи 1 і групи 2 серед ЕК від 3 змінних. Перевіряємо кожну можливу пару: $x_2x_3\neg x_4$ та $\neg x_1x_2\neg x_3$ (їх склеїти не можна); $x_2x_3\neg x_4$ та $\neg x_1x_2\neg x_4$ (їх склеїти не можна); $x_2x_3\neg x_4$ та $x_2\neg x_3\neg x_4$ (склеюються по x_3); $x_1x_2\neg x_4$ та $\neg x_1x_2\neg x_3$ (їх склеїти не можна); $x_1x_2\neg x_4$ та $\neg x_1x_2\neg x_4$ (склеюються по x_1, але ЕК $x_2\neg x_4$ вже отримана раніше; $x_1x_2\neg x_4$ та $x_2\neg x_3\neg x_4$ (їх склеїти не можна); $\neg x_1x_2x_4$ та $\neg x_1x_2\neg x_3$ (їх склеїти не можна); $\neg x_1x_2x_4$ та $\neg x_1x_2\neg x_4$ (склеюються по x_4); $\neg x_1x_2x_4$ та $x_2\neg x_3\neg x_4$ (їх склеїти не можна); $\neg x_1x_2x_3$ та $\neg x_1x_2\neg x_3$ (склеюються по x_3), але ЕК $\neg x_1x_2$ вже отримана раніше; $\neg x_1x_2x_3$ та $\neg x_1x_2\neg x_4$ (їх склеїти не можна); $\neg x_1x_2x_3$ та $x_2\neg x_3\neg x_4$ (їх склеїти не можна).	$(\underline{x_1x_2x_3} \vee \underline{x_2x_3x_4}) \vee (\underline{x_2x_3\neg x_4} \vee$ $\vee \underline{x_1x_2\neg x_4} \vee \underline{\neg x_1x_2x_4} \vee$ $\vee \underline{\neg x_1x_2x_3}) \vee (\underline{\neg x_1x_2\neg x_3} \vee$ $\vee \underline{\neg x_1x_2\neg x_4} \vee \underline{x_2\neg x_3\neg x_4}) \vee$ $\vee (x_2x_3) \vee (x_2\neg x_4 \vee \neg x_1x_2)$.
На цьому завершено склеювання ЕК від 3 змінних. Вилучаємо всі підкреслені ЕК.	$(x_2x_3) \vee (x_2\neg x_4 \vee \neg x_1x_2)$.

Далі слід переходити до склеювання ЕК від 2 змінних, але серед них нема жодної пари, яку можна було б склеїти. Тому процес склеювань завершується.

Оскільки серед ЕК від 4 та від 3 змінних не було жодної невикресленої, результатом склеювання є $x_2x_3 \vee x_2\neg x_4 \vee \neg x_1x_2$ (але якби були невикреслені ЕК від більшої кількості змінних, їх теж треба було б включити у результат першого етапу).

Дру́гий етап Внаслідок першого етапу (склеювань) отримують *скорочену ДНФ* (рос. «*сокращённая ДНФ*», англ. «*prime implicants*»). Часто вона і є мінімальною, але не завжди.

Щоб з'ясувати це, будують таблицю, стовпчики якої позначають ЕК початкової ДДНФ, рядки — ЕК побудованої скороченої ДНФ. На перетинах тих рядків і стовпчиків, де ЕК скороченої ДНФ *покриває* ЕК ДДНФ (тобто ЕК скороченої ДНФ набуває значення 1 в т. ч. і на то́му наборі, де повна ЕК ДДНФ), ставлять позначки (наприклад, галочки).

На жаль, у цієї таблиці забагато різних назв: «*таблиця поглинань*», «*таблиця перекриттів*», «*імплікантна таблиця*», «*імплікантна матриця*»; російськомовні варіанти аналогічні; англійською більш-менш прийнято називати її «*prime implicant chart*».

	$\neg x_1x_2\neg x_3\neg x_4$	$\neg x_1x_2\neg x_3x_4$	$\neg x_1x_2x_3\neg x_4$	$\neg x_1x_2x_3x_4$	$x_1x_2\neg x_3\neg x_4$	$x_1x_2x_3\neg x_4$	$x_1x_2x_3x_4$
$\neg x_1x_2$	√	√	√	√			
x_2x_3			√	√		√	√
$x_2\neg x_4$	√		√		√	√	

Якщо взяти скорочену ДНФ $x_2x_3 \vee x_2\neg x_4 \vee \neg x_1x_2$, отриману в процесі склеювань, виходить таблиця поглинань з рис. праворуч. Конкретно у ній, будь-яка спроба викреслити якийсь рядок (вилучити ЕК зі скороченої ДНФ) призведе до того, що в одному зі стовпчиків не лишиться галочок (на одному з потрібних наборів функція перестане набувати значення `true`). Отже, ця скорочена ДНФ є мінімальною.

Але в інших випадках може бути й не так. Розглянемо ще два приклади: функцію 1011000011111100 та функцію 1100111011010101. Виконавши (не наведені тут) етапи склеювань, отримаємо для першої з них скорочену ДНФ $\neg x_2\neg x_4 \vee x_1\neg x_3 \vee x_1\neg x_2 \vee \neg x_2x_3$, а для другої $\neg x_1x_2\neg x_4 \vee x_1x_4 \vee \neg x_3x_4 \vee \neg x_1\neg x_3 \vee \neg x_2\neg x_3$. У вигляді таблиць поглинань це буде:

Функція 1011000011111100

	$x_1\neg x_2x_3x_4$	$x_1x_2\neg x_3x_4$	$\neg x_1\neg x_2x_3x_4$	$x_1\neg x_2\neg x_3x_4$	$x_1\neg x_2x_3\neg x_4$	$x_1x_2\neg x_3\neg x_4$	$\neg x_1\neg x_2x_3\neg x_4$	$x_1\neg x_2\neg x_3\neg x_4$	$\neg x_1x_2\neg x_3\neg x_4$
$\neg x_2\neg x_4$				√		√	√	√	
$x_1\neg x_3$		√		√		√		√	
$x_1\neg x_2$	√			√	√			√	
$\neg x_2x_3$	√		√		√		√		

Функція 1100111011010101

	$x_1x_2x_3x_4$	$x_1\neg x_2x_3x_4$	$x_1x_2\neg x_3x_4$	$\neg x_1x_2\neg x_3x_4$	$\neg x_1x_2x_3\neg x_4$	$x_1\neg x_2\neg x_3x_4$	$\neg x_1\neg x_2x_3x_4$	$\neg x_1x_2\neg x_3\neg x_4$	$x_1\neg x_2x_3\neg x_4$	$\neg x_1\neg x_2\neg x_3\neg x_4$
$\neg x_2\neg x_3$						√	√		√	√
$\neg x_1\neg x_3$				√			√	√		√
$\neg x_3x_4$			√	√		√	√			
x_1x_4	√	√	√			√				
$\neg x_1x_2\neg x_4$					√			√		

Розглядаючи першу (ліву) з них, бачимо, що 3-й рядок (ЕК $x_1\neg x_2$) можна викреслити, бо він покриває лише повні елементарні кон'юнкції $x_1\neg x_2x_3x_4$ (яка і так покривається $\neg x_2x_3$), $x_1\neg x_2\neg x_3x_4$ (і так покривається $x_1\neg x_3$), $x_1\neg x_2x_3\neg x_4$ (покривається відразу двома іншими ЕК $\neg x_2x_3$ і $\neg x_2\neg x_4$) та $x_1\neg x_2\neg x_3\neg x_4$ (покривається $x_1\neg x_3$ і $\neg x_2\neg x_4$). Спроби ж викреслити будь-який інший рядок призводять до того, що в як(ому/ихо)сь стовпчик(у/ах) не лишається галочок. Отже, мінімальна ДНФ функції 1011000011111100 має вигляд $x_1\neg x_3 \vee \neg x_2x_3 \vee \neg x_2\neg x_4$.

У дру́гому (правому) все трохи складніше. Очевидно, не можна викреслювати ні 1-й ($\neg x_2\neg x_3$), ні 4-й (x_1x_4), ні 5-й ($\neg x_1x_2\neg x_4$) рядок — це призвело б до не покритості стовпчиків $x_1\neg x_2\neg x_3\neg x_4$, $x_1x_2x_3x_4$ чи $\neg x_1x_2x_3\neg x_4$ відповідно. А серед 2-го ($\neg x_1\neg x_3$) та 3-го ($\neg x_3x_4$) рядків можна викреслити будь-який один (всі стовпчики лишаються покритими), але не обидва одночасно (виявився б не покритим $\neg x_1x_2\neg x_3x_4$). У цьому випадку, функція 1100111011010101 має дві різні ДНФ $\neg x_1x_2\neg x_4 \vee x_1x_4 \vee \neg x_3x_4 \vee \neg x_2\neg x_3$ та $\neg x_1x_2\neg x_4 \vee x_1x_4 \vee \neg x_1\neg x_3 \vee \neg x_2\neg x_3$ однакової мінімальної довжини́ (що природньо, бо це та са́ма функція, для якої були отримані дві різні мінімальні ДНФ картами Карно).

Але біда в тім, що у складніших випадках може виявитися, що після різних викреслень виходять ДНФ різних довжин. Цілком можливо, що якусь ЕК вилучити *можна, але не варто*, бо якби не викреслили її, то можна було б викреслити щось інше й отримати коротший остáточний результат. І *наука не знає* простого й чіткого критерію, що дозволяв би гарантовано швидко і гарантовано правильно вирішити, які з таких ЕК краще вилучити, а які залишити. Метод Квайна, як і карти Карно, теж не дозволяє повністю уникнути перебору. В найгіршому випадку, при великій кількості змінних, це може бути *дуже* довго (наприклад, при ≈15 змінних склеювання можуть зайняти кілька секунд роботи комп'ютера, подальший перебір ЕК скороченої ДНФ — непередбачувано, від кількох секунд до кількох років).

Не виключені ситуації, коли не дуже важливо, чи отримана ДНФ справді мінімальна, чи її довжина лише близька до мінімальної; *тоді* цей перебір можна істотно спрощувати, пришвидшуючи за рахунок ризику можливо не мінімальної відповіді.

1.5 Методи перевірки тавтологічності логічних виразів

Тавтологія (рос. «*тавтология*», англ. «*tautology*») — це логічний вираз, який завжди (на всіх можливих наборах змінних) набуває значення `true`.

Звісно, є очевидний метод перевірки тавтологічності: побудувати таблицю істинності й подивитися, чи складається вона з самúх лише одиниць. Буває, що сáме так і доцільно робити. Але буває й так, що інші методи дозволяють отримати результат значно швидше. (Адже питають не повний вміст усієї таблиці істинності, а лише досить просту її властивість; «видобути» часткову інформацію може бути легше, ніж повну...)

1.5.1 Метод Квайна

Основна ідея методу Квайна (точніше кажучи, «методу Квайна перевірки тавтологічності булевих виразів», бо Квайн розробив також інші методи для інших задач; один з них розглянутий у розд. 1.4.2) — по можливості поєднати переваги й обійти недоліки таких «крайніх» підходів, як аналітичні перетворення та побудова таблиць істинності.

Суть методу в тому, щоб застосовувати аналітичні спрощення скрізь, де це легко й зручно, а там, де це важко й незручно, надавати якій-небудь логічній змінній конкретне значення. Як правило, після такої підстановки вираз знов стає придатним до зручних аналітичних спрощень. Звичайно, при підстановці значення замість змінної слід розглянути (послідовно, одúн після óдного) два випадки (підставити 0 і підставити 1).

Якщо в *усіх* випадках кінець кінцем отримуємо константу 1, то логічний вираз є тавтологією. А якщо хоча б у одному з випадків вийшов 0 — вираз не тавтологічний, решту випадків можна не перевіряти.

Для аналітичних спрощень при застосуванні методу Квайна можна використовувати будь-які логічні закони. Але у методі Квайна значно частіше, ніж у інших ситуаціях, застосовні такі тотожності:

$$\begin{array}{llll} 1 \vee x = x \vee 1 = 1; & 1 \wedge x = x \wedge 1 = x; & 1 \to x = x; & x \to 1 = 1; \\ 0 \vee x = x \vee 0 = x; & 0 \wedge x = x \wedge 0 = 0; & 0 \to x = 1; & x \to 0 = \neg x; \\ x \vee x = x \wedge x = x; & x \to x = \neg x \vee x = 1; & \neg x \wedge x = 0. & \end{array}$$

Приклад. Проаналізуємо (методом Квайна) на тавтологічність вираз $\big(((p \wedge q) \to r) \wedge (p \to q)\big) \to (p \to r)$.

Не схоже, щоб цей вираз можна було легко й зручно спростити аналітично. Так що спробуємо зафіксувати яку-небудь (довільну) змінну. Схоже, що найкраще фіксувати змінну p (бо вона має найбільше входжень).

1. Підставимо $p = 0$. Вираз набуває вигляду $\big(((0 \wedge q) \to r) \wedge (0 \to q)\big) \to (0 \to r)$. Очевидно, "$0 \wedge q$" спрощується до "0", і отримуємо $\big((0 \to r) \wedge (0 \to q)\big) \to (0 \to r)$. Тепер помітимо, що тотожність $0 \to x = 1$ дає можливість спростити *кожну* з дужок до 1, тобто увесь вираз до $(1 \wedge 1) \to 1$, тобто $1 \to 1$, тобто 1.

2. Підставимо $p=1$. Вираз набуває вигляду $\big(((1\wedge q)\to r)\wedge(1\to q)\big)\to(1\to r)$. Помітимо, що "$1\wedge q=q$", а тотожність $1\to x=x$ дає можливість спростити "$(1\to q)$" та "$(1\to r)$". Таким чином, увесь вираз спрощується до $\big((q\to r)\wedge q\big)\to r$.

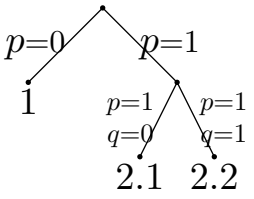
   Тепер можна або думати, як спростити вираз аналітично, або фіксувати ще якусь змінну. Щоб допоказати метод, підемо дру́гим шляхом.

   2.1. Підставимо $q=0$.
   (Важливо підставляти $q=0$ *не* у першопочатковий вираз $\big(((p\wedge q)\to r)\wedge(p\to q)\big)\to(p\to r)$, а у вираз $\big((q\to r)\wedge q\big)\to r$, отриманий внаслідок підстановки $p=1$. Томý це не пункт 3, а підпункт 2.1 всере́дині пункту 2.)
   Вираз набуває вигляду $\big((0\to r)\wedge 0\big)\to r$, тобто $(\ldots\wedge 0)\to r$, тобто $0\to r$, тобто 1.

   2.2. Підставимо $q=1$. Вираз набуває вигляду $\big((1\to r)\wedge 1\big)\to r=(r\wedge 1)\to r=r\to r=1$.

Тобто, в *усіх* випадках (1, 2.1, 2.2) вдалося спростити вираз до одиниці. Значить, він тавтологічний.

(Метод Квайна *не* є чітким алгоритмом. Яку змінну першою підставляти — правил нема, можна будь-яку. Коли у п. 1 «помітили, що $0\to x=1$ дає можливість ...» — могли ж і не помітити, й перетворювати значно довше. У п. 2 можна було не розглядати випадки 2.1, 2.2, а провести аналітичне спрощення, котре не таке й складне́. Тому хід застосування методу може істотно залежати від того, хто його застосовує. Але в ньому не буває так, щоб було геть не ясно, що робити: нема справді добрих ідей — можна хоча б розглядати випадки (підставляти значення змінної), і кінець кінцем гарантовано отримати результат. Правда, якщо випадків забагато, легше зразу будувати таблицю істинності...)



Цей метод Квайна іноді називають «*метод семантичних дерев*», бо процес перебору варіантів значень змінних зручно зображати у вигляді т. зв. кореневого де́рева.

Початок (корінь) де́рева — початкова ситуація; необхідність розглянути два варіанти p=0 та p=1 відповідає двом гілкам, що виходять з кореня; у випадку p=0 все вдалося встановити без подальшого перебору, а при p=1 доводиться розглядати випадки q=0 та q=1, томý права гілка знов розгалужується, а ліва — ні.

Подібні дере́ва дуже широко використовуються у computer science.

Іноді доводиться досліджувати вирази, в яких багатократно зустрічається деякий скла́дений підвираз. (Наприклад, "$a\vee cd$" у виразі $(a\vee cd)\to\big(bd\to(a\vee cd)\big)$.) Чи можна для пришвидшення розглядати два можливі варіанти значень *підвиразу*?

В самóму методі такого не написано. Але, хоч «по уставу» і «не положено», *спробуємо дослідити* правильність такого «модифіката Квайна».

Усе, ніби, так са́мо: якщо в *кожному* з випадків «підвираз = 0» і «підвираз = 1» отримали, що весь вираз істинний, то він таки буде завжди істинний... Підвираз же не може набувати інших значень, крім 0 та 1; значить, всі можливі випадки розглянуті...

Але розглянемо вираз $(a\to(b\to a))\vee(a\to(b\to a))$ і «випадки» згідно значення $a\to(b\to a)$. У «випадку» $a\to(b\to a)=1$ весь вираз спрощується до $1\vee 1=1$; у «випадку» $a\to(b\to a)=0$ — до $0\vee 0=0$. Так можна (*помилково!*) зробити висновок, ніби вираз $(a\to(b\to a))\vee(a\to(b\to a))$ не тавтологічний, бо хибний при $a\to(b\to a)=0$. А насправді «випадок $a\to(b\to a)=0$» неможливий, бо підвираз $a\to(b\to a)$ сам тавтологічний.

Міркування позаминулого абзацу *правильні в один бік* (на рівні «якщо–то», але не «тоді й тільки тоді»). Так що запропонована «модифікація» прискорює в деяких випадках отримання результату «так, тавтологічний», але коли дає результат «ні, не тавтологічний», треба ще перевіряти, чи точно той «випадок», при якому увесь вираз=0, можливий. А «справжній» метод Квайна такої проблеми не має.

1.5.2 Метод редукції

Основна ідея методу редукції — припустити, ніби досліджуваний вираз може (хоча б на деяких наборах логічних змінних) бути хибним, і подивитися, чи призведе таке припущення до протиріччя. Якщо призведе — значить, припущення було неправильне і насправді вираз тавтологічний.

Приклад 1. Проаналізуємо (методом редукції) на тавтологічність той самий вираз $\big(((p\wedge q)\to r)\wedge(p\to q)\big)\to(p\to r)$.

Припускаємо, ніби він може бути хибним:

(1) $\big(((p\wedge q)\to r)\wedge(p\to q)\big)\to(p\to r)=0$
Зовнішня операція — імплікація; "$\to$" хибна *лише* коли лівий її аргумент істинний, а правий хибний. Значить, з (1) випливають (2) та (3):

(2) $((p\wedge q)\to r)\wedge(p\to q)=1$ (з (1))

(3) $p\to r=0$ (з (1))
З (2) випливають (4) та (5), бо "$\wedge$" істинна *лише* на наборі 1 1; з (3) випливають (6) та (7), бо "$\to$" хибна *лише* на наборі 1 0:

(4) $(p\wedge q)\to r=1$ (з (2))

(5) $p\to q=1$ (з (2))

(6) $p=1$ (з (3))

(7) $r=0$ (з (3))
Підставимо (6) у (5):

(8) $1\to q=1$ (з (5), (6))
Оскільки $1\to 1=1$, але $1\to 0\neq 1$,

(9) $q=1$ (з (8))

А тепер розглянемо **(4)**, **(6)**, **(7)** та **(9)** разом.

При підстановці **(6)** $p=1$, **(7)** $r=0$ та **(9)** $q=1$ у **(4)** $(p\wedge q)\to r=1$ виходить $(1\wedge 1)\to 0=1$. *Оце і є* ***протиріччя*** (насправді ж $(1\wedge 1)\to 0=1\to 0=0$).

А звідки це протиріччя взялося? «Хто» у ньому «винуватий»? «Винувате» припущення (1), ніби вираз, який досліджують на тавтологічність, може дорівнювати 0. *Лише* це припущення заявлене голослівно, а всі інші твердження (від (2) до (9) і до власне протиріччя) є прямими чи опосередкованими наслідками цього припущення.

Раз із припущення отримали протиріччя — значить, це припущення хибне. А що припускали? Що вираз *може* (хоча б іноді) набувати значення 0. Отже, вираз не може (ніколи) набувати значення 0. Тобто, тавтологічний.

Приклад 2. Проаналізуємо (методом редукції) на тавтологічність вираз $(a\to b)\to((a\to c)\to(b\to c))$.

Припускаємо, ніби він може бути хибним:

(1) $(a\to b)\to\big((a\to c)\to(b\to c)\big)=0$
Зовнішня операція — імплікація; "$\to$" хибна *лише* коли лівий її аргумент істинний, а правий хибний. Значить, з (1) випливають (2) та (3):

(2) $(a\to b)=1$ (з (1))

(3) $(a\to c)\to(b\to c)=0$ (з (1))
З (3) випливають (4) та (5), бо "$\to$" хибна *лише* на наборі 1 0:

(4) $a\to c=1$ (з (3))

(5) $b\to c=0$ (з (3))
З (5) випливають (6) та (7), бо "$\to$" хибна *лише* на наборі 1 0:

(6) $b=1$ (з (5))

(7) $c=0$ (з (5))
Підставимо (7) у (4):

(8) $a\to 0=1$ (з (4), (7))
Це можливо *лише* при a=0 (бо $1\to0\neq1$, $0\to0=1$, інших варіантів нема):

(9) $a=0$ (з (8))
Підставимо (9) та (6) у (2):

(10) $0\to 1=1$ (з (2), (6), (9)), в чому нема ніякого протиріччя.

Тільки тут треба акуратно перевіряти, чи точно протиріччя нема *ніде*, чи не «загубили» вираз, де протиріччя все-таки з'являється. Цю перевірку можна робити по-різному. Можна так: «значення a, b, c отримані як наслідки **(4)** та **(5)** (що насправді означає «як наслідки **(3)**») й не протирічать **(2)**, а разом узяті **(2)** та **(3)** дають **(1)**, тобто вираз у цілому таки набуває значення хиба». Або, можна взяти **(6)** $b=1$, **(7)** $c=0$ та **(9)** $a=0$ й підставити у початковий вираз $(a\to b)\to((a\to c)\to(b\to c))$. Виходить $\underbrace{(0\to 1)}_{=1}\to(\underbrace{(0\to 0)}_{=1}\to\underbrace{(1\to 0)}_{=0})=1\to(1\to 0)=$ $=1\to0=0$, що явно демонструє його не тавтологічність.

Що робити у методі редукції, коли нема ні імплікацій чи диз'юнкцій, що рівні 0, ні кон'юнкцій, що рівні 1, ні констант, які можна підставити?

Універсальних, чітких і зручних рекомендацій просто нема. Можна придумувати щось специфічне для конкретного прикладу. Можна розглядати випадки. Наприклад: нехай відомо, що $A \oplus B = 0$ (де A та B — складені вирази). З цього не можна зробити гарантованого висновку про значення окремо взятого A і окремо взятого B. Зате можна розглянути окремо випадок $A{=}B{=}0$, окремо $A{=}B{=}1$, і знати, що інші випадки неможливі.

Але при розгляді випадків ***зникає*** *право заявляти «Як тільки виникло протиріччя, так і довели тавтологічність».* Кожен випадок (як-то «$A{=}B{=}0$») теж є в деякому смислі припущенням. І коли всере́дині такого випадку отримали протиріччя, то ще невідомо, що його спричинило: чи досліджуваний вираз тавтологічний, чи він не може бути нулем лише у випадку $A{=}B{=}0$, а в інших випадках може... І тавтологічність стає доведеною лише після виникнення протиріч *у кожному з можливих* випадків.

А розгляд сукупності *таких* випадків (A та B — не змінні, а складені вирази!) нерідко виявляється настільки заплутаним, що легше все кинути і зробити заново методом Квайна або побудовою таблиці істинності...

Тому́, для деяких виразів метод редукції дозволяє перевірити тавтологічність швидко й зручно, а для деяких інших з нього нема ніякого толку.

1.6 Предикати

1.6.1 Означення предиката

Досі ми вважали, що у логічних (бу́левих) виразах використовуються наперед задані бу́леві константи та/або змінні, для кожної з яких треба розглянути обидва можливі варіанти значень.

Але у програмуванні бу́леві значення найчастіше з'являються у ситуаціях на зразок "`if(x>0 && y!=0) ...`" — виконуються арифметичні *порівняння*, бу́левий результат яких *залежить від* чисел `x` та `y`... Очевидно, що подібні «умови» потрібні не лише у практичному програмуванні, а й у більшості галузей математики. Це формалізується за допомогою т. зв. *предикатів* (укр., рос. «*предика́т*», англ. «*predicate*»).

Предика́т — це функція, що може залежати від будь-якої (яка потрібна) кількості параметрів (вони ж аргументи)[11] будь-яких (які потрібні) типів, результатом якої обов'язково є логічне значення.

(У літературі можна знайти інші означення предиката, й деякі з них на перший погляд геть не схожі на щойно згадане. Але насправді всі вони мають один і той са́мий смисл, просто підходять до цього поняття «з різних боків».)

Наприклад, предикатом є твердження *«n — просте число»*. Цей предикат залежить від одного параметра n, тому його можна позначити як $P(n)$. Щоб отримати конкретне бу́леве значення, слід підставити замість "n" конкретне натуральне число: $P(3) = 1$, $P(17) = 1$, $P(35) = 0$, тощо.

Предикатами є усі математичні «порівняння» в широкому смислі цього сло́ва — наприклад, "$x > 0$" (дійсне число x строго додатнє); "$n \vdots k$" (натуральне число n ділиться націло на натуральне число k); "$x \perp y$" (пряма x перпендикулярна прямій y); ...

Предикат може складатися з простіших предикатів, зв'язаних логічними операціями. Наприклад, згаданий вираз "`(x>0 && y!=0)`" (у «більш математичному» вигляді — "$x > 0 \wedge y \neq 0$"). Якщо підставити замість параметрів x та y конкретні чи́сла, підвирази "$x > 0$" та "$y \neq 0$" набудуть конкретних бу́левих значень, і вже до цих бу́левих значень застосовується операція "$\wedge$".

Предикати не зобов'язані бути класично-математичними. Наприклад, можна розглянути предикат $T(x, y, z)$, котрий означає «z лежить на найкращому шляху з x до y», де в якості x, y та z можна підставляти населені пункти. Скажімо, T(Черкаси, Шпола, Сміла)$=1$, T(Умань, Київ, Василькі́в)$=1$, T(Чигирин, Кам'янка, Канів)$=0$.

(Умова «лежить на найкращому шляху» не математично строга, тому́ з'являються певні неоднозначності. Як бути з тим, що одним подобається їздити зі Сміли до Києва через Корсунь–Обухів, іншим — через Золотоношу–Бориспіль? Чи вважати, що Золотоноша «лежить на шляху», якщо усі їдуть по об'їзній,

[11] У програмуванні більш-менш прийнято розрізняти поняття «параметр» та «аргумент»: *параметр* — назва в заголовку функції (як підпрограми), яка потребу́є, щоб замість неї підставили щось конкретне, а *аргумент* — те конкретне, яке підставляють на місце параметра. Але на математику ця домовленість не поширюється, тому в тексті посібника можливі відхилення від неї. Тим паче можливі такі відхилення в інших книжках з математики.

а не через місто? Усі ці питання *не* дурні й непотрібні. Такого роду уточнення при переході від неформальних фраз до математичної постановки задачі можуть бути складовою хоч математичного розв'язування словесно сформульованої задачі, хоч реальної роботи програміста. Вибір найкращого уточнення може з'являтися, наприклад, внаслідок аналізу тексту задачі (чи вимог замовника). Зараз, коли ні детального тексту задачі, ні замовника нема, приймемо рішення: в усіх згаданих випадках вважати, що умова «лежить на найкращому шляху» виконується (предикат дорівнює `true`).)

Ще один «не класично-математичний» приклад предиката — «людина x побувала у місті y». У цьому предикаті параметри різнотипні: в якості x слід піставляти конкретну людину, в якості y — конкретне місто.

1.6.2 Способи подання предикатів

«*Подання́*» (воно ж «*предста́влення*»; рос. «*представле́ние*», англ. «*presentation*» або «*representation*») означає приблизно те ж, що «засоби запису». У першу чергу нас цікавить машинне (комп'ютерне) подання́.

Пряма аналітична формула Умова, яку виражає предикат, записується виразом типу `bool`.

(Наприклад, предикат «n — па́рне число» можна подати виразом "`n%2==0`".)

Прямі аналітичні формули найлегше будувати для такого роду арифметичних порівнянь; але вони бувають застосовні і до інших предикатів. Наприклад, геометричний предикат «$a \perp b$» (a, b — прямі на площині) теж можна задати прямою аналітичною формулою: узяти напрямні вектори прямих, обчислити їхній скалярний добуток і подивитися, чи рівний він нулю.

Але цим способом можна подати *далеко не всі* предикати...

Алгоритмічна перевірка Передбачає наявність алгоритму, вхідними даними якого є аргумент(и) предиката, результатом — значення предиката.

(Наприклад, предикат «n — просте число» не можна записати одним виразом типу `bool`. Зате його можна виразити за допомогою циклічного алгоритму, який перебере всі чи́сла від 2 до $[\sqrt{n}]$ і для кожного подивиться, чи справді воно не є дільником n.)

Очевидно, обидва вищезгадані способи подання незастосовні, коли предикат такої природи, що його значення не можна отримати шляхом міркувань, а можна лише знати або не знати. Якщо, наприклад, спитати в когось, хто *не* знає м. Львів, *«чи проходить трамвай №3 через площу Ринок?»*, то, яким алгоритмом не аналізуй слова́ «трамвай», «Ринок» та число 3, навряд чи можна отримати відповідь, не звернувшися до карти, або людини, котра знає, або списку зупинок, або ще якогось джерела́ інформації. В таких випадках слід спиратися на деякі переліки окремих фактів. Звісно, переліки реально використати лише там, де кількість можливих наборів аргументів предиката не просто скінче́нна, а ще й не дуже велика.

Таблиця істинності Передбачає, що для всіх можливих значень параметр(а/ів) предиката готова відповідь ("0" чи "1") береться з таблиці. Для унарних предикатів таблиця лінійна (одновимірна), для бінарних — прямокутна (двовимірна), де рядки́ відповідають значенням одного параметра, стовпчики — значенням іншого параметра, тощо.

(Наприклад, табличний спосіб дуже зручний для подання предиката $G(x, y)$ «тролейбусом x можна доїхати до y», де x може набувати лише значення 1, 4, 7, 10, а y — лише значення "драмтеатр", "ПЗР", "пл. Б. Хм.", "трол. парк", "ЧНУ".

	драмтеатр	ПЗР	пл. Б. Хм.	трол. парк	ЧНУ
1	1	0	1	0	1
4	1	0	0	1	0
7	1	0	1	1	1
10	1	1	1	0	0

Ми казали, що нас цікавить у першу чергу комп'ютерне подання. Таблиці зазвичай реалізують масивами, індексами яких є послідовні цілі чи́сла, а не слова́. У таких випадках треба використати яке-небудь перетворення назв у номери́, котрі й будуть індексами масиву... Але це вже суто програмістська, а не математична робота.)

Перелік наборів, на яких предикат істинний Все сказано назвою. (Наприклад, цей самий «тролейбусний» предикат буде поданий у вигляді переліка (1, драмтеатр), (1, пл. Б. Хм.), (1, ЧНУ), (4, драмтеатр), (4, трол. парк), (7, драмтеатр), (7, пл. Б. Хм.), (7, трол. парк), (7, ЧНУ), (10, драмтеатр), (10, ПЗР), (10, пл. Б. Хм.).)

Перевага цього способу: *якщо* значення предиката дуже рідко дорівнює істині, *то* такий перелік значно економніший, ніж таблиця, за об'ємом пам'яті. Але це так лише для деяких предикатів.

Недоліки способу такі: (1) дещо ускладнений пошук значення предиката за заданими аргументами; (2) не зрозуміло, чи варто зберігати перелік всіх можливих значень кожного параметра (= область визначення) предиката. З одного боку, збереження суперечить ідеї економії пам'яті. З іншого — не збереження може призвести до втрати інформації про справжню область визначення (а як наслідок — до неправильного результату застосування квантора; див. також початок розд. 1.6.4).)

Наведені способи подання — *не* «єдино дозволені»; можуть бути інші, можна модифікувати та комбінувати ці. Власне, значна частина реальних програм цим і займається: бере щось з готових баз даних (що є аналогом останніх двох способів), і перетворює його згідно якогось алгоритму (що є аналогом перших двох)...

1.6.3 Означення кванторів

І у математиці, і у фразах природною мовою є поняття «кожен», «завжди», «для всіх», «для будь-якого», а також «існує», «хоча б іноді». Їх формалізують за допомогою т. зв. *ква́нторів* (укр., рос. *ква́нтор*; англ. *quantifier*). Як правило, розглядають два види кванторів — *квантор загальності* та *квантор існування*.

Квантор загальності *Квантор загальності* (рос. «*квантор (все)общности*», англ. «*universal quantifier*»; позначається "$\forall$" (походить від перевернутої "A" з «for All»); читається «*для всіх*», «*для будь-якого*», «*для кожного*», «*для довільного*», «*для всех*», «*для любого*», «*для произвольного*», «*для каждого*», «*for all*», «*for any*», «*for arbitrary*», «*for each*»)[12] позначає, що предикат істинний для всіх можливих значень змінної.

Тобто, стверджувати, що предикат з квантором загальності істинний ("$(\forall x P(x)) = \texttt{true}$"), можна лише після того, як або перебрали всі можливі значення параметра x і для кожного з них переконалися, що предикат набуває значення `true` (все це, зрозуміло, можливо *лише* якщо кількість можливих значень параметра не просто скінче́нна, а й досить мала́), або побудували (правильне) математичне доведення.

(Наприклад, $\forall x(x \cdot 0 = 0)$ буде `true`, бо рівність $x \cdot 0 = 0$ справді виконується абсолютно для всіх x.)

Щоб стверджувати "$(\forall x P(x)) = \texttt{false}$", досить знайти якесь одне значення аргумента, при якому предикат набуває значення "`false`". Це називають *контрприклад* (рос. «*контрпример*», англ. «*counterexample*»).

(Скажімо, $\forall x(x^2 > 10)$ буде `false`, і щоб це показати, досить пред'явити контрприклад $x=2$, при якому $(2^2 > 10) = \texttt{false}$. Для цього предиката є й інші контрприклади, скажімо, $x=-0{,}12$, враховуючи $(-0{,}12)^2 = 0{,}0144 < 10$, теж дає $((-0{,}12)^2 > 10) = \texttt{false}$.)

Контрприклади широко застосовують у самих різних галузях математики та філософії, тому повторимо суть цього поняття іншими словами. Є твердження, що деяка властивість (чи то математично сформульований предикат, чи то філософська думка, тощо) виконується *завжди*. У відповідь наводять деякий конкретний приклад, коли ця властивість не дотримана. Якщо сформулювати такий контрприклад-відповідь вдалося — значить, початкове твердження (що ця властивість завжди виконується) було хибним.

Контрприклад не має доказової сили, коли у початковому твердженні говориться про «зазвичай», «у більшості випадків», тощо; але квантору загальності відповідає «завжди», і проти нього застосовувати контрприклад можна. (Для повноти картини варто врахувати також розд. 1.6.4.)

Невдачі при пошуку контрприкладу на побутовому рівні часто сприймаються за доведення правильності твердження; але у математичній логіці це так *лише* у вже згаданому випадку «перебрали абсолютно всі можливі значення параметра», а в інших випадках невдачі при пошуку контрприкладу нічого не доводять і не спростовують: можливо, невдачі спричинені тим, що предикат всюди істинний, але можливо й тим, що погано шукали.

[12] Цікаво відзначити, що з точки зору психології, соціології та ін. подібних дисциплін «всі» і «кожен» — різні поняття; але з точки зору класичної математичної логіки це одне й те саме.

Квантор існування *Квантор існування* (рос. «*квантор существования*», англ. «*existential quantifier*»; позначається "$\exists$" (походить від перевернутої "E" з «there Exists»); читається «*існує*», «*для деякого*», «*існує хоча б один*», «*существует*», «*для некоторого*», «*существует хотя бы один*», «*there exists*», «*for some*», «*for at least one*») позначає, що є *хоча б одне* значення змінної, при якому предикат істинний.

(*Не* «*рівно* одне», а *хоча б* одне; тобто, можна одне, можна п'ять, можна всі...[13])

Отже, щоб стверджувати, що предикат з квантором існування істинний ("$(\exists x P(x))=$`true`"), достатньо знайти яке-небудь значення змінної, при якому предикат набуває значення `true`; щоб стверджувати протилежне ("$(\exists x P(x))=$`false`"), потрібно або перебрати всі можливі значення параметра x, і для кожного з них переконатися, що предикат набуває значення `false`, або будувати математичне доведення.

(Наприклад, $\exists x(x^2{>}5)=$`true`, і щоб це показати, досить пред'явити $x{=}4$, при якому $(4^2>5)=$`true`.

І, наприклад, $\exists x(x{+}2=x{+}5)$ буде `false`, бо як не шукай, а не знайдеш такого x, щоб вийшло $x{+}2=$ $=x{+}5$. Або, більш строго: зі шкільного курсу алгебри відомо, що якщо з обох частин рівності відняти один і той самий вираз, рівність не перестане бути правильною, якщо була, і не стане правильною, якщо не була. Тож віднімемо з обох частин x, і отримаємо $2{=}5$, а така рівність очевидно ніколи (ні при якому значенні x) не виконується.)

Значення аргумента, яким показується істинність предиката з квантором існування, нерідко називають *приклад* (*без(!)* префікса «контр», який означає «проти».) Цілком «симетрично» до квантора загальності, невдачі при пошуку прикладу (підтверджувального) на побутовому рівні часто сприймаються за спростування (доведення хибності) твердження; але у математичній логіці це так *лише* у вже згаданому випадку «перебрали абсолютно всі можливі значення параметра», а в інших випадках відсутність прикладу нічого не спростовує і не доводить.

Синтаксис використання кванторів *За квантором мусить слідувати спочатку змінна, потім предикат.* Як правило, предикат залежить від цієї змінної; наприклад "$\forall x(x^2 \geqslant 0)$" — маємо змінну x при кванторі, і вона ж потім використовується у "$x^2 \geqslant 0$". Як виключення, можливі й записи у стилі "$\exists t(y{<}z)$" ("t" не має стосунку до "$y<z$", і квантор нічого не дає, але формально так можна; ну, як можна писати $x+0$, хоча толку додавати $0\ldots$).

Записи ж вигляду "$\exists$" або "$\forall x$" або "$\exists P(x)$" взагалі не є завершеними виразами, як не є осмисленим завершеним виразом "$\sqrt{\ }$": щоб застосувати корінь, треба вказати, з чого його добувати. А щоб застосувати квантор, треба і сам значок квантора, і змінна, і предикат.

Вільні та зв'язані змінні. Перейменування змінних *Зв'язана змінна* (рос. «*связанная переменная*», англ. «*bound variable*») — це параметр предиката, по якому взятий квантор. *Вільна змінна* (рос. «*свободная переменная*», англ. «*free variable*») — це параметр предиката, по якому не взятий квантор. Наприклад, для предиката "$\exists x P(x)$", x є зв'язаною змінною, а для предиката "$P(y)$", y є вільною змінною. А в предикаті "$\exists a T(a,b,c)$" є одночасно і зв'язана змінна a, і вільні змінні b та c.

Вільні змінні «видимі ззовні»: щоб предикат набув конкретного значення, їх ще треба підставити. Зв'язані змінні «невидимі», їх можливі значення «вже перебрані» квантором, підставити вже нічого не можна.

Зв'язані змінні можна перейменовувати. Наприклад, "$\exists x P(x)$" та "$\exists z P(z)$" мають абсолютно однаковий смисл — предикат $P(\cdot)$ виконується хоча б для одного значення аргумента. При перейменуванні зв'язаної змінної, треба заміняти *всі її входження* — і безпосередньо при кванторі, і скрізь у предикаті, до якого цей квантор застосований (в *області дії* квантора).

Водночас, перейменовувати вільні змінні не можна. Скажімо, "$x>5$" та "$y>5$" — різні предикати, і вони набудуть різних значень, якщо підставити з зовнішнього світу $x=3$ та $y=7$.

(Можна провести таку аналогію. Просто так замінити лише в одному місці програми "`if(x>0) ...`" на "`if(y>0) ...`" неправильно, бо `x` та `y` — різні змінні, й толку порівнювати не ту, яка потрібна? Так само неправильно і свавільно перейменувати вільну змінну. А от заміна змінної як у заголовку циклу `for`, так і в його тілі (наприклад, цикл "`for(int i=0; i<N; i++) cout << a[i] << " ";`" на цикл "`for(int k=0; k<N; k++) cout << a[k] << " ";`") не впливає на смисл: наведені цикли виконують абсолютно однакову

[13] У деяких *інших* книжках з математики іноді використовують також квантор "$\exists!$"; *він* означає «існує рівно од(ин/на/не)» й називається «квантор *існування та єдиності*». Але у рамках цього посібника такий квантор (зі знаком оклику) більше не згадуватиметься, а "$\exists$" (без знака оклику) має цілком загальноприйнятий смисл «*хоча б* од(ин/на/не)».

роботу — виводять значення елементів масиву `a`, й неважливо, яка змінна (`i` чи `k`) використовується як лічильник. Тут ситуація подібна до заміни зв'язаної змінної як при кванторі, так і в області його дії.

Крім того, міняти "`for(int i...`" на "`for(int k...`" можна не завжди, а лише якщо змінна `k` не використовується всере́дині цього циклу ні для яких інших потреб. Те са́ме стосується і перейменувань зв'язаних змінних у квантифікованих предикатах: нове ім'я може бути будь-яким, але таким, що не має в поточному контексті ніякого іншого смислу.)

Квантори застосовують частіше, ніж здається на перший погляд Розглянемо твердження *«Якщо натуральне число закінчується на 0, то воно па́рне»*. До нього ставляться, як до істинного, а не як до предиката, істинного для одних чисел і хибного для інших. Хоча «шматочки» *«Натуральне число закінчується на 0»* та *«Натуральне число па́рне»* явно залежать від того, яке число підставити. То залежність від числа́ все-таки є, чи нема?

Причина непорозуміння — у твердженні квантор не вказаний явно, але мається на увазі. Воно є (стандартним) скороченням такого: *«Для кожного натурального числа́ справедливо: якщо його десятковий запис закінчується на 0, то воно па́рне»*. А тут уже чітко видно і квантор загальності, і те, що якщо згадані «шматочки» позначити $P(n) =$ *«Натуральне число n закінчується на 0»* та $Q(n) =$ *«Натуральне число n па́рне»*, то цілком нормально, що $P(n)$ та $Q(n)$ залежать від вільної змінної n, а усе твердження "$\forall n(P(n) \to Q(n))$" не залежить від зв'язаної змінної n.

1.6.4 Обмежені квантори

Розглянемо квантифікований предикат "$\exists k(k < 1)$". Його значення залежить від того, ***де*** *заданий* предикат "$k < 1$". Якщо в якості k можна підставляти лише натуральні чи́сла, то жодне натуральне число не є строго меншим одиниці — отже, увесь (квантифікований) вираз дорівнює хибі. А якщо розглянути той са́мий запис "$\exists k(k < 1)$", але вважаючи, що k може бути довільним дійсним числом, то увесь (квантифікований) вираз дорівнює істині, бо можна знайти, наприклад, $k = \frac{3}{4}$, при якому $(\frac{3}{4} < 1) =$`true`. Отже, *щоб правильно розуміти смисл квантифікованого предиката, треба чітко знати область можливих значень змінної, по якій береться квантор.*

Один із способів задати цю область — т. зв. *обмежені квантори* (рос. «*ограниченные кванторы*», англ. «*bounded quantifiers*»). Це квантори, при яких записано не лише змінну, а ще й додаткову умову, яка накладається на цю змінну. Наприклад, неоднозначний запис "$\exists k(k < 1)$" можна уточнити як "$\exists k_{k\in\mathbb{N}}(k < 1)$" (що дорівнює `false`) або як "$\exists k_{k\in\mathbb{R}}(k < 1)$" (що дорівнює `true`). Або, наприклад, «$Q(x)$ виконується для всіх іксів з інтервалу від 0 до 1» можна записати як "$\forall x_{0<x<1} Q(x)$".

Додаткова умова, накладена на змінну, сама є предикатом (причому предикатом з єдиною вільною змінною — тією са́мою, по якій береться обмежений квантор). Тому можна побудувати предикати зі «звичайними» кванторами, еквівалентні предикатам з обмеженим "$\forall x_{S(x)} P(x)$" та "$\exists x_{S(x)} P(x)$", і вони матимуть вигляд "$\forall x\big(S(x) \dots P(x)\big)$" та "$\exists x\big(S(x) \dots P(x)\big)$" (де замість "$\dots$" потрібно підставити правильну логічну операцію).

Обмежений квантор "$\exists x_{S(x)} P(x)$" означає «серед тих іксів, для яких $S(x)$, існує такий, що $\dots$». Тобто, має знайтися x, який задовольняє одночасно і додаткову умову $S(x)$, і предикат $P(x)$:

$$\exists x_{S(x)} P(x) = \exists x\big(S(x) \wedge P(x)\big). \tag{1}$$

Це ***не*** *означає*, ніби "$\forall x_{S(x)} P(x)$" перетворюється у $\forall x\big(S(x) \wedge P(x)\big)$. Такий вираз ставить надто жорсткі вимоги: щоб абсолютно для всіх іксів виконувалися і $S(x)$, і $P(x)$. Хоча "$\forall x_{S(x)} P(x)$" нічого не каже про ті ікси, для яких не виконується додаткова умова $S(x)$ — для них $P(x)$ може хоч виконуватися, хоч ні. А от для всіх тих іксів, де $S(x)$ виконується, вимагається, щоб виконувалося і $P(x)$. Отже, імплікація:

$$\forall x_{S(x)} P(x) = \forall x\big(S(x) \to P(x)\big). \tag{2}$$

(Приклад про па́рні чи́сла зі стор. 33 теж можна і звести до обмеженого квантора загальності, і використати в аргументації того, що (2) повинна містити са́ме імплікацію.)

1.6.5 Застосування кванторів до багатоарних предикатів

Тут з'являються різні випадки. Можна зв'язати кванторами усі змінні, а можна лише частину; можна узяти по різним змінним однотипні квантори, а можна різнотипні.

Усі змінні, від яких залежить предикат, зв'язуються однотипними кванторами Тобто, або по всім змінним однаково беруться "$\forall$", або по всім змінним однаково беруться "$\exists$". Це найпростіший випадок, аналогічний розглянутому випадку унарних предикатів.

(Наприклад, $\forall x\forall y\forall z(x+y=z)=$`false`, бо можна підібрати контрприклад, скажімо, x=2, y=3, z=1, при яких рівність не виконується; а $\exists x\exists y\exists z(x+y=z)=$`true`, бо можна підібрати і, наприклад, x=1, y=1, z=2, при підстановці яких рівність виконується.

Тепер щодо перевірки шляхом перебору значень парамерів: перебирати слід всі можливі пари/трійки/.... Само собою, хибність $\forall x_1\ldots\forall x_n P(x_1,\ldots,x_n)$ може бути встановлена першим же контрприкладом, а істинність $\exists x_1\ldots\exists x_n P(x_1,\ldots,x_n)$ — першим же підтверджувальним прикладом; але для встановлення істинності $\forall x_1\ldots\forall x_n P(x_1,\ldots,x_n)$ чи хибності $\exists x_1\ldots\exists x_n P(x_1,\ldots,x_n)$ таки треба перебирати всі можливі поєднання кожного значення кожного параметра з усіма значеннями решти параметрів.)

Зв'язування квантором однієї зі змінних *Ця* змінна перестає бути вільною, а решта лишаються. Томý утворюється предикат, залежний від решти змінних. Наприклад, $\exists x P(x,y)$ залежить від y. (Залежність може бути фіктивною, тобто значення предиката може бути однаковим при всіх наборах змінних, що лишилися вільними. А може бути й справжня залежність.)

Знайдемо $\forall x G(x,y)$, $\exists x G(x,y)$, $\forall y G(x,y)$ та $\exists y G(x,y)$ для предиката «тролейбус ... проїжджає через ...» (стор. 30).

	драмтеатр	ПЗР	пл. Б. Хм.	трол. парк	ЧНУ	$\forall y G(x,y)$	$\exists y G(x,y)$
1	1	0	1	0	1	0	1
4	1	0	0	1	0	0	1
7	1	0	1	1	1	0	1
10	1	1	1	0	0	0	1
$\forall x G(x,y)$	1	0	0	0	0		
$\exists x G(x,y)$	1	1	1	1	1		

При $y=$ПЗР, $\forall x G(x,\text{ПЗР})=0$, бо не всі тролейбуси проходять через ПЗР (у стовпчику «ПЗР» є хоча б один нуль). Так само з $\forall x G(x,\text{пл. Б. Хм.})=0$, $\forall x G(x,\text{трол. парк})=0$, $\forall x G(x,\text{ЧНУ})=0$. А $\forall x G(x,\text{драмтеатр})=1$, бо всі наведені тролейбуси проходять повз драмтеатр (стовпчик «драмтеатр» складається з самих одиниць). Оці чотири нулі й одна одиниця і формують предикат $\forall x G(x,y)$, залежний від y (істинний при $y=$ драмтеатр, і хибний при $y=$ пл. Б. Хм., $y=$ трол. парк, $y=$ ЧНУ та $y=$ ПЗР).

Для $\exists x G(x,y)$ усе аналогічно, але перевіряємо, чи є хоча б одна 1 у стовпчику. Конкретно для цієї таблиці, $\exists x G(x,y)$ істинний при всіх y (залежність від y фіктивна).

Для $\forall y G(x,y)$ та $\exists y G(x,y)$ усе аналогічно, але перевіряємо, чи всі одиниці та чи є хоч одна одиниця по рядкам, а не стовпчикам. Для цієї таблиці виходить, що стовпчик $\forall y G(x,y)$ складається з самих лише нулів, стовпчик $\exists y G(x,y)$ — з самих лише одиниць.

Тепер дослідимо предикат $\exists y(x+y=z)$ (x, y, z — дійсні). Його не можна подати скінчéнною таблицею, отже деталі попереднього способу не застосовні. Але лишається незмінною ідея «"існує y" — от і спробуємо підібрати». $x+y=z$ рівносильно $y=(z-x)$. Дію $z-x$ можна виконати завжди (які б не були дійсні x, z). Отже, нам вдалося знайти такий y, щоб предикат "$x+y=z$" був істинним. Значить, $\exists y(x+y=z)=$`true` (змінні x та z фіктивні).

Тепер дослідимо той самий предикат "$\exists y(x+y=z)$", але x, y, z — натуральні. При $x<z$ попередній розв'язок «візьмемо $y=(z-x)$ й отримаємо тотожнє `true`» правильний, але при $x\geqslant z$ дія $z-x$ виводить за межі натуральних чисел, а в першопочатковому формулюванні $(x+y=z)$ неможливо додати до і так більшого ікса натуральне число й отримати менший зет. Тому тут $\exists y(x+y=z)=\begin{cases}\texttt{true}, & \text{при } x<z,\\ \texttt{false}, & \text{при } x\geqslant z\end{cases}=(x<z)$. Був предикат $(x+y=z)$ від трьох змінних x, y, z, застосували до нього квантор $\exists y$, отримали предикат $(x<z)$ від решти змінних x, z.

Якщо початковий предикат має n ($n\geqslant 3$) параметрів, і однотипні квантори беруться відразу по кільком параметрам, але не всім — ситуація повністю аналогічна. Наприклад, $\exists x\exists t P(x,y,z,t)$ залежить від y та z, причому істинний тоді й тільки тоді, коли початковий $P(x,y,z,t)$ при вказаних y та z істинний хоча б при деяких x та t.

До предиката застосовано різнотипні квантори Це найскладніша, але й найпотрібніша ситуація. Почнемо з випадків застосування двох (обох) різнотипних кванторів до двомісного предиката. Тобто, випадків $\exists x\forall y P(x,y)$, $\forall x\exists y P(x,y)$, $\exists y\forall x P(x,y)$ та $\forall y\exists x P(x,y)$. Їх можна пояснити двома способами, які призводять до однакового результату, і взагалі взаємозамінні.

Спосіб перший. Квантифікація $\exists x\forall y P(x,y)$ читається «існує такий ікс, що для всіх ігреків виконується пе від ікс, ігрек» і означає «можна підібрати одне таке значення $x{=}x^\star$, що при цьому значенні $x^\star$ предикат істинний абсолютно при всіх значеннях y». Аналогічно, $\exists y\forall x P(x,y)$ читається «існує такий ігрек, що для всіх іксів виконується пе від ікс, ігрек» і означає «можна підібрати одне таке значення $y{=}y^\star$, що при цьому значенні $y^\star$ предикат істинний абсолютно при всіх значеннях x».

Квантифікація $\forall y\exists x P(x,y)$ читається «для кожного ігрека існує такий ікс, що виконується пе від ікс, ігрек» і означає «для кожного y можна підібрати таке значення $x^{\star\star}(y)$, що при цих $x^{\star\star}(y)$ та y предикат істинний». Аналогічно, $\forall x\exists y P(x,y)$ читається «для кожного ікса існує такий ігрек, що виконується пе від ікс, ігрек» і означає «для кожного x можна підібрати таке значення $y^{\star\star}(x)$, що при цих x та $y^{\star\star}(x)$ предикат істинний».

Зверніть увагу, що у першій парі випадків ($\exists x\forall y$ та $\exists y\forall x$) квантифікований предикат істинний лише коли можливо пред'явити *один і той самий* x чи y для всіх можливих значень іншої змінної. Тоді як $\forall y\exists x$ дозволяє для різних y підбирати різні x (аналогічно, $\forall x\exists y$ дозволяє для різних x підбирати різні y). Причому, са́ме дозволяє, а не вимагає. Тобто, $\forall y\exists x$ дозволяє і ситуацію, коли для різних ігреків підбираються різні ікси, і ситуацію, коли для всіх ігреків береться один і той самий ікс, і ситуацію, коли для різних ігреків ікси частково повторюються й частково різні.

Ситуації $\begin{cases}\exists x\forall y P(x,y)=\texttt{true},\\ \forall y\exists x P(x,y)=\texttt{false}\end{cases}$ *чи* $\begin{cases}\exists y\forall x P(x,y)=\texttt{true},\\ \forall x\exists y P(x,y)=\texttt{false}\end{cases}$ *неможливі.*

Доведення. $\exists x\forall y P(x,y)=\texttt{true}$ означає, що вдалося вибрати $x^\star$, один і той са́мий для всіх y. Тоді ніщо не заважає щоразу пред'являти са́ме цей $x^\star$ для кожного з y і так отримати $\forall y\exists x P(x,y)=\texttt{true}$. З другою ситуацією все аналогічно. ■

А от ситуації $\begin{cases}\exists x\forall y P(x,y)=\texttt{false},\\ \forall y\exists x P(x,y)=\texttt{true}\end{cases}$ чи $\begin{cases}\exists y\forall x P(x,y)=\texttt{false},\\ \forall x\exists y P(x,y)=\texttt{true}\end{cases}$ цілком можливі. Але са́ме можливі, а не гарантовані. Обмін місцями різнотипних кванторів може вплинути на результат, а може й не вплинути.

Наприклад, розглянемо квантифікації $\forall y\exists x G(x,y)$ та $\exists x\forall y G(x,y)$ все того ж «тролейбусного» предиката. Хоча в обох квантифікаціях йдеться про хоча б один тролейбус і про абсолютно всі (згадані у таблиці) місця, значення вийдуть різні: $\forall y\exists x G(x,y)=\texttt{true}$, бо для $y=$ драмтеатр можна узяти $x{=}1$, для $y=$ ПЗР можна узяти $x{=}10$, для $y=$ пл. Б. Хм. можна узяти $x{=}7$, для $y=$ трол. парк можна узяти $x{=}4$ та для $y=$ ЧНУ можна знов узяти $x{=}1$. А $\exists x\forall y G(x,y)=\texttt{false}$, бо нема жодного такого тролейбусного маршруту, щоб цей конкретний маршрут проходив через усі наведені у таблиці місця.

А якщо розглянути квантифікації $\forall x\exists y G(x,y)$ та $\exists y\forall x G(x,y)$ того ж предиката, то в обох випадках вийде `true`, бо і кожен тролейбус проїжджає повз хоча б одне місце, і за тією таблицею існує таке місце $y=$ драмтеатр, повз яке проїжджають усі тролейбуси.

Спосіб дру́гий. Квантори слід застосовувати зсередини назовні:

$$\begin{array}{rclcrcl}\exists x\forall y P(x,y) &=& \exists x\big(\forall y P(x,y)\big); & & \forall y\exists x P(x,y) &=& \forall y\big(\exists x P(x,y)\big);\\ \exists y\forall x P(x,y) &=& \exists y\big(\forall x P(x,y)\big); & & \forall x\exists y P(x,y) &=& \forall x\big(\exists y P(x,y)\big).\end{array}$$

Наприклад, щоб знайти $\forall y\exists x P(x,y)$, можна спочатку знайти $\exists x P(x,y)$, потім застосувати до $\exists x P(x,y)$ квантор $\forall y$; в інших випадках аналогічно.

Для ще складніших конструкцій, ситуація повністю аналогічна. Наприклад, розглянемо $\exists x\forall z\exists y\exists t P(x,y,z,t)$. Для нього можна побудувати зсередини назовні спочатку $\exists t P(x,y,z,t)$, потім $\exists y\big(\exists t P(x,y,z,t)\big)$, потім $\forall z\big(\exists y(\exists t P(x,y,z,t))\big)$, і насамкінець $\exists x\big(\forall z\big(\exists y(\exists t P(x,y,z,t))\big)\big)$. Або можна сказати, що «$\exists x\forall z\exists y\exists t P(x,y,z,t)$ істинний» рівносильно «існує такий x, щоб при ньому (одному й тому ж) предикат виконувався для всіх z і хоча б для деяких y та t (де y та t можуть підбиратися залежно від z та x)».

1.6.6 Аналітичні перетворення предикатів

Якщо у виразі з предикатами можна виділити частину, яка в точності відповідає якомусь із законів (тотожностей) розд. 1.2, то цей закон можна застосувати, і в результаті отримаємо тотожньо рівний предикат.

Наприклад, $\neg\big(P(x)\wedge\exists tQ(t)\big)$ за законом де Моргана можна перетворити до $\neg P(x)\vee\neg(\exists tQ(t))$. Хоча спочатку закон $\neg(a\wedge b)=\neg a\vee\neg b$ вводився тільки для булевих значень/змінних, можна взяти в якості a предикат $P(x)$ і в якості b квантифікований предикат "$\exists tQ(t)$".

Аналогічно, для предиката $\exists x\big(\neg(P(x)\to Q(x))\big)$ можна виразити заперечення імплікації $\neg(a\to b)=a\wedge\neg b$ й отримати рівносильний предикат $\exists x\big(P(x)\wedge\neg Q(x)\big)$. Те, що перетворення відбулося всере́дині області дії квантора, не створює ніяких проблем.

Але жоден з раніше вивчених (розд. 1.2) логічних законів не дозволяє спростити, наприклад, $\neg\big(\exists x(\neg P(x))\big)$. Тому зараз розглянемо додаткові закони, які мають справу з кванторами.

1. Перейменування зв'язаних змінних
$$\exists xP(x)=\exists yP(y),\quad \forall xP(x)=\forall yP(y).$$
2. Внесення заперечення у квантор
$$\neg(\forall xP(x))=\exists x(\neg P(x)),\quad \neg(\exists xP(x))=\forall x(\neg P(x)).$$
3. Можливість перестановки однотипних кванторів
$$\forall x\forall yP(x,y)=\forall y\forall xP(x,y),\quad \exists x\exists yP(x,y)=\exists y\exists xP(x,y).$$
(Раніше показано, що різнотипні квантори переставляти <u>не</u> можна!)
4. Дистрибутивність "$\forall$" відносно "$\wedge$", "$\exists$" відносно "$\vee$"
$$\big(\forall xP(x)\big)\wedge\big(\forall xQ(x)\big)=\forall x\big(P(x)\wedge Q(x)\big),$$
$$\big(\exists xP(x)\big)\vee\big(\exists xQ(x)\big)=\exists x\big(P(x)\vee Q(x)\big).$$
(Ні дистрибутивність "$\exists$" відносно "$\wedge$", ні дистрибутивність "$\forall$" відносно "$\vee$" <u>не</u> виконуються! Тобто, $\big(\forall xP(x)\big)\vee\big(\forall xQ(x)\big)$ не завжди дорівнює $\forall x\big(P(x)\vee Q(x)\big)$; аналогічно, $\big(\exists xP(x)\big)\wedge\big(\exists xQ(x)\big)$ не завжди дорівнює $\exists x\big(P(x)\wedge Q(x)\big)$.)
5. Занесення під квантор виразу, *не залежного від змінної*
$$C=\exists xC,\quad C=\forall xC;$$
$$\big(\exists xP(x)\big)\wedge C=\exists x\big(P(x)\wedge C\big),\quad \big(\exists xP(x)\big)\vee C=\exists x\big(P(x)\vee C\big);$$
$$\big(\forall xP(x)\big)\wedge C=\forall x\big(P(x)\wedge C\big),\quad \big(\forall xP(x)\big)\vee C=\forall x\big(P(x)\vee C\big).$$
(де C *не залежить* від x; на відміну від п. 4, тут правильні в т. ч. й перетворення для "$\exists$" та "$\wedge$", а також "$\forall$" та "$\vee$".)

Перелічені тотожності записані для унарних та бінарних предикатів. Але вони справедливі також і для багатоарних предикатів, що мають інші змінні. Головне, щоб були ті змінні, які справді задіяні у тотожності.

1.7 Повнота систем булевих функцій

1.7.1 Означення функціональної повноти

Система (набір, множина) функцій називається *функціонально повною*[14] (рос. «*функционально полная*», англ. «*functionally complete*»), якщо абсолютно будь-яку функцію можна подати у вигляді формули, в котрій містяться лише функції цієї системи.

Наприклад, будь-яку булеву функцію можна подати у вигляді таблиці істинності, і для будь-якої таблиці істинності можна побудувати ДДНФ. ДДНФ містить лише операції "$\wedge$", "$\vee$" та "$\neg$". *Отже, система булевих функцій* $\{\wedge,\vee,\neg\}$ *є функціонально повною*, бо через них справді можна виразити абсолютно будь-яку булеву функцію.

[14] Якщо в найближчому контексті і так згадується слово «функція», «функціонально повна» можна скорочувати до «повна». Наприклад, «повна система функцій».

(Формально, це міркування не охоплює особливий випадок — функцію–константу 0. Але тотожній нуль можна подати як "$x_1 \wedge \neg x_1$", тож це (єдине) виключення не порушує повноту системи $\{\wedge, \vee, \neg\}$.)

Система $\{\vee, \neg\}$ теж функціонально повна: можна взяти ДДНФ, і повиключати всі кон'юнкції за допомогою закона де Моргана $x \wedge y = \neg(\neg x \vee \neg y)$. В результаті вийде досить громіздка формула, найімовірніше не ДДНФ; але це буде формула, в якій використано лише дві дії — $\vee$ та $\neg$. І таку формулу можна побудувати для будь-якої таблиці істинності, тобто для будь-якої булевої функції. Отже, система функцій $\{\vee, \neg\}$ повна.

Абсолютно аналогічно показується повнота системи $\{\wedge, \neg\}$ (через застосування іншого закону де Моргана $x \vee y = \neg(\neg x \wedge \neg y)$).

Ще один приклад функціонально повної системи — $\{\oplus, \wedge, 1\}$. Операції "$\oplus$", "$\wedge$" та константа 1 (й лише вони) використовуються в поліномах Жегалкіна, а ними теж можна задати абсолютно будь-яку булеву функцію. Важливо, що йдеться са́ме про систему $\{\oplus, \wedge, 1\}$, а не $\{\oplus, \wedge\}$: як не комбінуй самі́ лише "$\oplus$" та "$\wedge$" (без константи 1), виразити $\neg x$, неможливо.

1.7.2 Класи Поста та теорема Поста

Класи Поста та теорема Поста надають спосіб, що дозволяє для будь-якої системи булевих функцій перевірити, чи є вона повною, не будуючи сам спосіб вираження інших функцій через функції цієї системи. Що з одного боку буває зручно, з іншого — *не конструктивно*: коли за цим способом взнаємо, що *якось* можна, це *не* дає ніякої інформації про те, *як са́ме* це зробити.

Спочатку, означення класів Поста. Їх п'ять:

1. Булева функція належить до класу Поста T_0 (*зберігає константу 0*), якщо $f(0, \dots, 0) = 0$ (при підстановці замість усіх змінних нулів результатом буде 0).
2. Булева функція належить до класу Поста T_1 (*зберігає константу 1*), якщо $f(1, \dots, 1) = 1$ (при підстановці замість усіх змінних одиниць результатом буде 1).
3. Булева функція належить до класу Поста S (*самодвоїста*), якщо на (*всіх!*) протилежних наборах вона набуває протилежні значення: $\forall x_1 \dots \forall x_n (f(\neg x_1, \dots, \neg x_n) = \neg f(x_1, \dots, x_n))$.
4. Булева функція належить до класу Поста L (*лінійна*), якщо її поліном Жегалкіна не містить жодної кон'юнкції.
5. Булева функція належить до класу Поста M (*монотонна*), якщо на більших наборах аргументів функція приймає не менші значення.

(Схоже з означенням монотонно неспадної функції зі шкільного курсу алгебри. Але ця схожість лише часткова, бо у шкільному курсі алгебри поняття монотонності було лише для функцій, які приймали один числовий аргумент і вертали числовий результат. А за́раз розглядаються булеві функції, що приймають кілька логічних аргументів і вертають логічний результат. Тому тепер треба ще з'ясувати, що таке «не менше значення» і що таке «більший набір аргументів».

Поняття «не менше значення» вводиться природньо: уявимо, ніби 0 — не `false`, а числовий 0, а 1 — не `true`, а числова 1, і порівняємо ці чи́сла. Інакше кажучи, $1 \geqslant 1$, $1 \geqslant 0$ та $0 \geqslant 0$ виконуються, а "$0 \geqslant 1$" не виконується.

Тепер про «більший набір аргументів»: набір $(x_1, \dots, x_n)$ вважається більшим за набір $(y_1, \dots, y_n)$ тоді й тільки тоді, коли $(\forall i (x_i \geqslant y_i)) \wedge (\exists i (x_i > y_i))$, тобто усі (1-й, 2-й, …, n-й) компоненти набора x-ів не менші відповідних компонентів набора y-ів, і при цьому хоча б по одному компоненту x_i строго більше y_i.. Наприклад, $(1, 1, 0, 1) \succ (0, 1, 0, 0)$, бо по першій компоненті $1 > 0$, по другій $1 \geqslant 1$, по третій $0 \geqslant 0$, і по четвертій $1 > 0$.

І лише за́раз, коли сформульовано означення «не меншого значення» та «більшого набору аргументів», можна вважати, що означення класу M задане чітко й математично правильно.)

Для прикладу, дослідимо на належність класам Поста функцію $f_{\to}(x_1, x_2) = x_1 \to x_2$.

x_1	x_2	$f_{\to}(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

$f_{\to}(0, 0) = 1 \neq 0$, тому $f_{\to} \notin T_0$.

$f_{\to}(1, 1) = 1$, тому $f_{\to} \in T_1$.

Можна підібрати протилежні набори 0 0 та 1 1, на котрих $f_{\to}$ набуває однакові (не протилежні) значення. Отже, $f_{\to} \notin S$.

Ми вже будували (на стор. 19) поліном Жегалкіна для $f_{\to}$, він має вигляд $x_1x_2 \oplus x_1 \oplus 1$, тобто містить кон'юнкцію (у підвиразі "x_1x_2"). Отже, $f_{\to} \notin L$.

Існує пара наборів 0 0 та 1 0, для якої $(0,0) \prec (1,0)$, але $f_{\to}(0,0) = 1 > 0 = f_{\to}(1,0)$. Отже, $f_{\to} \notin M$.

(Причому, це — *єдина* пара наборів, для якої порушується монотонність. Те, що $f_{\to}(0,1) = 1 > 0 = f_{\to}(1,0)$, не рахується, бо за вказаним вище означенням «більшого набору» набори $(0,1)$ та $(1,0)$ взагалі не порівнювані: набір $(0,1)$ не більший, бо має менший x_1, а $(1,0)$ не більший, бо має менший x_2. Це є прикладом *нелінійного відношення порядку*, що детальніше буде розглянуто у розд. 2.6.1 та 2.6.3.)

Тепер дослідимо функцію $f_{\neg}(x) = \neg x$.

x	$f_{\neg}(x)$
0	1
1	0

$f_{\neg}(0) = 1 \neq 0$, тому $f_{\neg} \notin T_0$.

$f_{\neg}(1) = 0 \neq 1$, тому $f_{\neg} \notin T_1$.

Для цієї функції є лише одна пара протилежних наборів: набір "0" та набір "1". Для них умова $f_{\neg}(\neg x) = \neg(\neg x) = \neg f_{\neg}(x)$ виконується. Отже, $f_{\neg} \in S$.

Як відомо, поліном Жегалкіна для $\neg x$ має вигляд $x \oplus 1$. Тут нема кон'юнкцій. Отже, $f_{\neg} \in L$.

Можна підібрати пару наборів (набір "0" та набір "1"), для якої $(0) \prec (1)$, але $f_{\neg}(0) = 1 > 0 = f_{\neg}(1)$. Отже, $f_{\neg} \notin M$.

Теорема Поста *Для того, щоб система булевих функцій була функціонально повною, необхідно й достатньо, щоб система містила хоча б одну функцію, яка не зберігає 0, хоча б одну, що не зберігає 1, хоча б одну не самодвоїсту, хоча б одну не лінійну, хоча б одну не монотонну.*

(Повне доведення досить складне́, і ми його пропускаємо. Втім, в одну сторону воно відносно просте: ретельно обґрунтовуються твердження «якщо абсолютно всі функції системи належать T_0, то, як їх не поєднуй, однаково вийде функція, що належить T_0», і ще чотири аналогічних для решти класів Поста.)

Наприклад, спираючись на цю теорему і раніше проведене дослідження функцій $f_{\to}$ та $f_{\neg}$, можна зробити висновок, що $\{\to, \neg\}$ є функціонально повною системою: тут є функція, що не зберігає 0 (будь-яка "$\to$" чи "$\neg$"), є функція, що не зберігає 1 ("$\neg$"), є не самодвоїста ("$\to$"), є не лінійна ("$\to$"), є не монотонна (наприклад, "$\to$").

	T_0	T_1	S	L	M
$\to$	$-$	$+$	$-$	$-$	$-$
$\neg$	$-$	$-$	$+$	$+$	$-$

Є дві бінарні логічні операції, кожна з яких сама́ утворює функціонально повну систему: стрілка Пірса та штрих Шеффера.

x_1	x_2	стрілка Пірса $x_1 \downarrow x_2$	штрих Шеффера $x_1 \mid x_2$
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0

Їх можна навіть *знайти*, спираючись на теорему Поста: $f \notin T_0$ вимагає $f(0,0) = 1$; $f \notin T_1$ вимагає $f(1,1) = 0$; тоді з $f(0,0) = 1$, $f(1,1) = 0$ та наявності лише двох пар протилежних наборів $(\neg 0, \neg 0) = (1,1)$ і $(\neg 0, \neg 1) = (1,0)$ випливає, що для $f \notin S$ необхідно, щоб $f(1,0) \neq \neg f(0,1)$, тобто $f(1,0) = f(0,1)$; отже, *лише* дві бінарні операції, з таблицями істинності 1 0 0 0 та 1 1 1 0, мають ту властивість, що кожна окремо взята одночасно $f \notin T_0$, $f \notin T_1$, $f \notin S$.

З $f(0,0) = 1$ та $f(1,1) = 0$ для кожної з них автоматично слідує $f \notin M$. Для перевірки ж нелінійності слід побудувати будь-яким способом поліном Жегалкіна кожної з них: $x_1 \downarrow x_2 = x_1x_2 \oplus \oplus x_1 \oplus x_2 \oplus 1$; $\quad x_1 \mid x_2 = x_1x_2 \oplus 1$; отже, обидві операції справді $\notin M$, що завершує доведення як того, що будь-яка окремо взята з цих операцій утворює функціонально повну систему, так і того, що інших таких бінарних операцій нема.

Заодно, вкажемо, як виразити через кожну з цих операцій $\neg$, $\vee$ та $\wedge$:

$$\begin{array}{ll} \neg x = x \downarrow x & \neg x = x \mid x \\ x \vee y = {}_{\neg(x \downarrow y)} = (x \downarrow y) \downarrow (x \downarrow y) & x \vee y = {}_{(\neg x) \mid (\neg y)} = (x \mid x) \mid (y \mid y) \\ x \wedge y = {}_{(\neg x) \downarrow (\neg y)} = (x \downarrow x) \downarrow (y \downarrow y) & x \wedge y = {}_{\neg(x \mid y)} = (x \mid y) \mid (x \mid y) \end{array}$$

1.8 Тризначна логіка (вступ)

Класична логіка оперує виключно зі значеннями `true` та `false`, але бувають ситуації, коли цих значень недостатньо. *Одна з* таких відносно простих і масових ситуацій — банальне незнання.

Наприклад, для питання *«Чи був дощ у Черкасах 5 травня 1555 року (за старим стилем)?»*, формально можливі лише варіанти відповіді "так, був" та "ні, не було" (особливо, якщо уточнити межі Черкас на той час та інші подібні деталі). Але достеменно встановити, все-таки "так" чи "ні", нереально. *Невідомо*, чи такий дощ був, чи ні. І це *не* предикат: тут нема параметра, залежно від якого виходитиме `true` чи `false`. Тут са́ме невідомість.

Сучасніший приклад: нехай при реєстрації на сайті вказувати дату народження не обов'язково, але для тих, хто вказав, сайт її використовує. Нехай адміністрація сайту бажає показати одне повідомлення всім користувачам, молодшим 21 року, та інше всім, хто досяг віку 21 рік чи більше. Навіть якщо є чітке означення, в яку секунду людині виповнюється 21 рік — для тих, хто не вказав дату народження, результатом перевірки «чи досяг віку 21 рік» на основі самих лише реєстраційних даних буде «невідомо».

Відомі кілька різних спроб розширити класичну логіку так, щоб вона враховувала такі ситуації. Їх са́ме кілька різних, «конкуруючих» і не в усьому узгоджених одна́ з о́дною. Тому́ розглянемо лише деякі базові моменти, спільні для кількох (все одно не всіх) із таких математичних узагальнень і деяких спеціалізованих мов, що реально використовують тризначну логіку у програмуванні — наприклад, SQL (structured query language, дослівно «мова структурованих запитів»; використовується для роботи з базами даних).

Будемо вважати, що у тризначній логіці можливі такі значення:

- `false`, воно ж "0", воно ж "F", воно ж «хиба»;
- `unknown`, воно ж "$\frac{1}{2}$", воно ж "?", воно ж "U", воно ж «невідомо»;
- `true`, воно ж "1", воно ж "T", воно ж «істина».

(На жаль, у деяких джерелах пропонують ще інші, несумісні, системи позначень. Одна з них позначає `false` як "−", `unknown` як "0", `true` як "+", тобто надає зовсім інший смисл символу "0". Ще одна позначає `false` як "0", `unknown` як "1", `true` як "2", тобто надає зовсім інший смисл символу "1". На жаль, так буває, коли різні люди пропонують свої системи позначень...)

Заперечення вводиться так:

x	$\neg x$
0	1
$\frac{1}{2}$	$\frac{1}{2}$
1	0

Для заперечення `true` та `false` результат той самий, що й у класичній двозначній логіці. $\neg\frac{1}{2}=\frac{1}{2}$, бо неможливість гарантувати ні істинність, ні хибність x якраз і є неможливістю гарантувати ні хибність, ні істинність $\neg x$.

x	y	$x\vee y$	$x\wedge y$
0	0	0	0
0	$\frac{1}{2}$	$\frac{1}{2}$	0
0	1	1	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	1	1	$\frac{1}{2}$
1	0	1	0
1	$\frac{1}{2}$	1	$\frac{1}{2}$
1	1	1	1

Диз'юнкція та кон'юнкція вводяться згідно таблиці істинності, наведеної ліворуч (рядків 3^n замість 2^n, бо кожен аргумент може набувати одне з трьох значень). Праворуч записане те саме, просто аналогічно таблиці множення (результат розміщено на перетині рядка та стовпчика, відповідних аргументам). Ще один поширений спосіб задати ці самі означення: якщо вважати, ніби `false` відповідає *числу* 0 (са́ме числу́), `unknown` — числу $\frac{1}{2}$, `true` — числу 1, виходить, що "$x\vee y$" відповідає "$\max(x,y)$", а "$x\wedge y$" — "$\min(x,y)$".

$\vee$	0	$\frac{1}{2}$	1
0	0	$\frac{1}{2}$	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
1	1	1	1

$\wedge$	0	$\frac{1}{2}$	1
0	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$
1	0	$\frac{1}{2}$	1

При цих означеннях лишаються справедливими рівності $0\vee0=0$, $0\vee1=1$, $1\vee0=1$, $1\vee1=1$, $0\wedge0=0$, $0\wedge1=0$, $1\wedge0=0$, $1\wedge1=1$, а також значна частина інших властивостей: «для істинності диз'юнкції досить істинності одного з аргументів, незалежно від значення іншого», «якщо хоча б один з аргументів хибний, кон'юнкція хибна незалежно від значення іншого», тощо.

Але все-таки лише частина. Деякі з законів класичної логіки порушуються у тризначній. Найфундаментальніший з них — закон виключення третього: $x\vee\neg x$ дає результат 1 лише при x=0 та x=1, але не при x=$\frac{1}{2}$.

x	$\neg x$	$x\vee\neg x$
0	1	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
1	0	1

Інші формули, які є тотожностями класичної логіки, пропонується перевірити у тризначній шляхом побудови таблиць істинності.

Ми не будемо вводити інші логічні операції (імплікацію, ксор, тощо). По-перше, є забагато різних варіантів цих операцій: у літературі можна знайти *п'ять різних (!)* варіантів імплікації у тризначній логіці (які відрізняються такими деталями, як, наприклад, «чи вважати $\frac{1}{2}\to 0=0$, чи $\frac{1}{2}\to 0=\frac{1}{2}$»), і властивості цих імплікацій частково відрізняються одна від одної. По-друге, са́ме такий мінімальний набір тризначних операцій $\{\neg,\vee,\wedge\}$ є спільною частиною більшості математичних теорій та SQL.

У SQL є перевірка "`... IS NOT NULL`", яка забезпечує самі лише значення `TRUE`/`FALSE`, без `UNKNOWN`. Але двозначної логіки все одно не виходить. Наприклад, якщо ціна може бути невизначеною, умову «ціна до 100 (включно)» слід виражати як "`(PRICE IS NOT NULL) AND (PRICE <= 100)`", «дорожче 100» — як "`(PRICE IS NOT NULL) AND (PRICE > 100)`". В результаті, умова «ціна до 100 (включно)» не є запереченням умови «дорожче 100», бо "`NOT((PRICE IS NOT NULL) AND (PRICE > 100))`" дає `TRUE` не лише для цін до 100 (включно), а також і для невідомих цін. Та само проблема з `ELSE`-гілкою розгалуження. Відповідно, практичне застосування тризначної логіки є, але обмежене: більшість програмістів ставляться до тризначної логіки SQL як до малоприємного способу хоч якось працювати в умовах неповних даних, а не як до нової потужної можливості.

1.9 Завдання до розділу 1

1. Побудувати таблицю істинності для $\neg(\neg x_1\vee\neg x_2)$.
2. Побудувати таблицю істинності для $(x\vee\neg y)\wedge(\neg x\vee y)$.
3. Побудувати таблицю істинності для $\neg(x\vee y)\vee(xy)$.
4. Побудувати таблицю істинності для $(x\to y)\to z$.
5. Переконатися за допомогою таблиць істинності, що еквіваленція та ксор асоціативні, а імплікація — ні.
6. З'ясувати (за допомогою таблиць істинності), які з чотирьох виразів
   (а) $x\to y$, (б) $y\to x$, (в) $\neg x\to\neg y$, (г) $\neg y\to\neg x$
   рівносильні між собою, а які — ні.
7. Чи є тотожньо рівними вирази $A=xy\vee x\neg z\vee y\neg z$, $B=(y\vee z)\to x(z\to y)$?
8. Чи є тотожньо рівними вирази $A=xy\neg z\vee xy\vee z\neg z$, $B=xy$? (з'ясувати за допомогою таблиць істинності).
9. Істинна чи хибна з точки зору формальної логіки фраза *«Якщо $2\times 2=5$, то Місяць зроблений з зеленого сиру»*?
10. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Побудувати таблицю істинності логічного виразу, <u>*не*</u> спрощуючи аналітично. Обов'язково виділяти у виразі підвирази, позначати їх (1), (2), ..., і підписувати стовпчики таблиці цими позначками.
    (1) $(x_2\to\neg x_3)\oplus\neg\big(x_2\leftrightarrow(x_3\to x_4)\big)\oplus\big((x_1\vee x_3\vee x_4\vee x_2)\wedge x_1\wedge x_2\big)$;
    (2) $\big(\neg(x_1\leftrightarrow\neg x_3)\vee(x_3\oplus x_2)\big)\wedge\neg(x_3\to\neg x_1)\wedge x_4\wedge(x_2\vee x_4)$;
    (3) $(x_4\wedge x_3\wedge x_2\wedge x_1)\oplus(\neg x_3\to x_1)\oplus(x_4\vee x_3)\oplus\big((x_2\to\neg x_4)\leftrightarrow x_1\big)$;
    (4) $(\neg x_2\to x_4)\oplus\neg(x_3\vee x_4\vee\neg x_1\vee x_2)\oplus\big(x_1\wedge(x_1\leftrightarrow x_3)\wedge x_4\big)$;
    (5) $(x_3\to\neg x_4)\oplus(x_2\wedge\neg x_1)\oplus\big((\neg x_2\leftrightarrow x_3)\to x_4\big)\oplus\big(x_4\to(x_2\vee x_1)\big)$;
    (6) $(x_4\oplus x_1\oplus x_2\oplus x_3)\wedge\big((x_4\leftrightarrow x_1)\to(x_3\to x_2)\big)\wedge x_2\wedge\big(x_3\leftrightarrow(x_1\vee\neg x_3)\big)$;
    (7) $\big((x_2\vee x_4\vee x_3\vee x_1)\wedge(x_1\to x_3)\wedge(x_4\oplus x_2\oplus\neg x_3)\big)\leftrightarrow\big((x_3\leftrightarrow x_1)\to x_2\big)$;
    (8) $\neg\big((\neg x_1\leftrightarrow x_2)\to x_1\big)\vee\big((\neg x_4\wedge x_3)\leftrightarrow(x_2\oplus x_1)\big)\vee x_1\vee\neg x_3$;
    (9) $(x_1\to x_4)\oplus\big((\neg x_2\to x_4)\leftrightarrow(x_2\to x_1)\big)\oplus x_2\oplus\big(\neg x_1\wedge(x_1\vee x_2)\wedge x_3\big)$;
    (10) $\big((x_2\to x_3)\vee\neg(x_1\leftrightarrow x_2)\vee x_3\big)\wedge(x_4\oplus x_1)\wedge(x_3\vee\neg x_4)\wedge\neg x_2$;
    (11) $(x_3\oplus x_4\oplus\neg x_2)\leftrightarrow\big((x_1\vee x_2\vee x_3\vee\neg x_4)\to(x_3\wedge\neg x_1\wedge x_4)\big)$;
    (12) $\big((\neg x_1\wedge x_4)\leftrightarrow x_2\big)\to\neg\big((x_4\to x_3)\oplus x_4\oplus(x_4\vee\neg x_1\vee x_3)\oplus x_3\big)$;
    (13) $\big(\neg(x_4\to x_1)\leftrightarrow(\neg x_4\vee x_3)\big)\wedge\big(x_2\oplus(x_4\wedge x_2)\oplus x_4\oplus(x_1\to x_3)\big)\wedge x_4$;
    (14) $(\neg x_1\oplus x_3\oplus x_2)\wedge\neg\big(x_3\oplus(x_4\vee x_1)\oplus x_4\big)\wedge\big(\neg x_4\to(x_3\leftrightarrow x_2)\big)$;
    (15) $\big((x_1\to x_2)\wedge x_1\wedge\neg(x_4\leftrightarrow x_2)\wedge x_3\big)\vee\big((x_4\oplus x_1)\to(x_1\wedge\neg x_3\wedge x_2)\big)$.

11. Спростити $xy\neg z \vee xy \vee z\neg z$.
12. Спростити $(x \vee y) \wedge (x \vee y \vee t)$.
13. Спростити $x \vee \neg xy$.
14. Довести $(b \to c) \to a = (a \vee b) \wedge (c \to a)$.
15. Довести $\neg(a \vee b \vee c) = \neg a \neg b \neg c$.
16. Спростити вираз $\neg((x \to y) \vee (x \to z)y)$ так, щоб у ньому лишилося дві логічні операції.
17. Спростити вираз $x \wedge y \vee \neg y \wedge z \vee y$ так, щоб у ньому лишилася одна логічна операція. *Обов'язково* почати зі з'ясування порядку дій.
18. Спростити $(a \vee b) \wedge (b \vee \neg c) \wedge (\neg a \vee b) \wedge (\neg b \vee \neg c)$.
19. Спростити вираз $(a \to b) \to b$ так, щоб лишилася одна операція.
20. Спростити вираз $(a \to b) \to \neg b$ так, щоб лишилася одна операція.
21. Спростити вираз $b \to (a \to b)$.
22. Спростити $\neg ab \vee a\neg b \vee ab$.
23. Спростити $(x \vee (y \to z)) \wedge (x \vee y \vee z) \wedge (x \vee y \vee t)$. (Цей вираз *не* скорочується до *гетъ* простого. Але скоротити з 8 операцій до 3–4 можна.)
24. Спростити $(x \vee (y \to z)) \wedge (x \vee y \vee z) \vee (x \vee z \vee t)$.
25. Спростити вираз $(a \to b) \wedge (a \vee bc) \wedge (a \to c) \vee \neg c$ так, щоб у ньому лишилася одна логічна операція.
26. Знайти для функції $x_1 \vee x_2$ ДДНФ, ДКНФ та поліном Жегалкіна (перетворенням ДДНФ та методом невизначених коефіцієнтів).
27. Знайти для булевої функції від двох змінних, що має таблицю істинності 0100, ДДНФ, ДКНФ та поліном Жегалкіна двома способами (перетворенням ДДНФ та методом невизначених коефіцієнтів).
28. Побудувати для функції $(x \to y) \to z$, на основі таблиці істинності, ДДНФ, ДКНФ та поліном Жегалкіна двома способами (перетворенням ДДНФ та методом невизначених коефіцієнтів).
29. Знайти ДДНФ, ДКНФ та поліном Жегалкіна для виразів:

    (а) $(x \leftrightarrow y) \leftrightarrow z$; (б) $(x \oplus y) \oplus z$; (в) $x \leftrightarrow (y \leftrightarrow z)$; (г) $x \oplus (y \oplus z)$.

    При цьому слід постаратися вибрати для кожної складової завдання найбільш доречний саме для неї спосіб (інакше кажучи, вітається варіант «замість робити зайву роботу, пояснити, чому її можна не робити», але пояснення повинні бути чіткі й математично правильні).
30. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Побудувати поліном Жегалкіна, обов'язково методом перетворення ДДНФ (варіанти підібрані так, щоб побудова цим методом була приблизно однакової громіздкості та складності для різних варіантів; при розв'язуванні методом невизначених коефіцієнтів це не так).

    (1) 01100100; (4) 00110011; (7) 01110000; (10) 01011000;
    (2) 00101001; (5) 00001111; (8) 01100010; (11) 00100111;
    (3) 01010101; (6) 00011101; (9) 00011011; (12) 01100001.

31. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Побудувати поліном Жегалкіна, обов'язково методом невизначених коефіцієнтів (варіанти підібрані так, щоб побудова цим методом була приблизно однакової громіздкості та складності для різних варіантів; при розв'язуванні методом перетворення ДДНФ це не так).

    (1) 10111000; (4) 01001010; (7) 10101100; (10) 00101100;
    (2) 11000110; (5) 00111000; (8) 01010010; (11) 11010010;
    (3) 01011110; (6) 10100000; (9) 10110100; (12) 01000110.

32. Зобразити на карті Карно область, що відповідає формулі:
    (а) $\neg x\neg y\neg zt$; (в) $x\neg z$; (д) $\neg x\neg y$; (ж) $\neg x\neg y\neg zt\vee x\neg z$;
    (б) $xy\neg t$; (г) yz; (е) $\neg y\neg z$; (и) $yz\vee\neg xy\vee\neg y\neg zt$.
33. Мінімізувати (картами Карно) булеву функцію від чотирьох змінних:
    (а) 1111111100001110; (б) 1111001011010101.
34. Виконавши попереднє завдання картами Карно, розв'язати ті самі приклади методом Квайна–Мак-Класкі й переконатися в узгодженості результатів.
35. Дослідити, чи тавтологічний вираз
$$(a\to(b\to c))\to\big((d\to b)\to(a\to(d\to c))\big).$$
36. Дослідити, чи тавтологічний вираз
$$(a\to(b\to c))\to\big((b\to d)\to(a\to(d\to c))\big).$$
37. Дослідити, чи тавтологічні вирази:
    (а) $(a\to b)\wedge(b\to c)\to(a\to bc)$;
    (б) $(a\to bc)\to(a\to b)\wedge(b\to c)$;
    (в) $(a\vee b)\to\big((a\to c)\wedge(b\to d)\to(c\vee d)\big)$.

    Всі три вирази повинні бути досліджені методом Квайна, деякі два досліджені методом редукції, а щодо третього пояснено, чому досліджувати його методом редукції значно важче й менш ефективно, ніж інші два. (Який з них є отим третім, визначити самостійно.)
38. Пояснити, які з наведених фраз можна вважати предикатами (і які ні):
    (а) $x^2+7x+1=0$;
    (б) x^2+7x+1;
    (в) Розв'яжіть рівняння $x^2+7x+1=0$;
    (г) Вчитель сказав: «Розв'яжіть рівняння $x^2+7x+1=0$».
39. Знайти (використавши засоби «числової» алгебри та/або арифметики) значення предикатів. Вважати $x\in\mathbb{R}$.
    (а) $\exists x(x^2=9)$; (г) $\neg(\forall x(x^2=9))$; (ж) $\forall x(x^2>0)$;
    (б) $\forall x(x^2=9)$; (д) $\neg(\forall x(x^2\neq 9))$; (и) $\exists x(x^2\geqslant 0)$;
    (в) $\forall x(x^2\neq 9)$; (е) $\forall x(x^2\geqslant 0)$; (к) $\exists x(x^2>0)$.

40. Нехай $W(x,y)$ означає *«людина x побувала у місті y»*. Що тоді означають формули:
    (а) $W(x,\text{Черкаси})$;
    (б) $\neg W(\text{дядько Степан},\text{Токіо})$;
    (в) $\forall y(\neg W(\text{дядько Степан},y))$;
    (г) $\neg(\forall yW(\text{дядько Степан},y))$; чи рівносильне це твердження з попереднім? якщо ні, то пояснити суть відмінності;
    (д) $\exists y\exists xW(x,y)$;
    (е) $\exists x\exists yW(x,y)$; чи рівносильне це твердження з попереднім? якщо ні, то пояснити суть відмінності;
    (ж) $\forall y\exists xW(x,y)$;
    (и) $\exists x\forall yW(x,y)$; чи рівносильне це твердження з попереднім? якщо ні, то пояснити суть відмінності.

41. Записати у вигляді квантифікованого предиката твердження *«Жодна людина не побувала на Марсі»* (виразивши через $W(x,y)$, де y вже не міста, а місця).
42. Розглянемо предикат $W(x,y)$, звужений на малу область визначення $x\in\{$Дмитро, дядько Степан, Назар, Мирослава$\}$, $y\in\{$Черкаси, Львів, Київ, Торонто$\}$ і заданий таблицею істинності

	Черкаси	Львів	Київ	Торонто
Дмитро	1	1	1	1
дядько Степан	0	0	1	0
Назар	1	1	1	0
Мирослава	0	1	0	1

    Прочитати словами та знайти значення усіх можливих квантифікацій (як повних, так і часткових; всього їх існує 12 штук).

43. Знайти (використавши засоби «числової» алгебри та/або арифметики) значення предикатів. Вважати $x, y \in \mathbb{R}$.

(а) $\forall x \exists y (y = x)$;
(б) $\exists x \forall y (y = x)$;
(в) $\forall x \exists y (y = x^2)$;
(г) $\forall y \exists x (y = x^2)$;
(д) $\forall x \exists y (y \geqslant x^2)$;
(е) $\exists y \forall x (y \geqslant x^2)$;
(ж) $\exists y \forall x (y \leqslant x^2)$.

Відповіді пояснити.

44. Записати у вигляді квантифікованого предиката твердження *«Всі ромби — паралелограми, але бувають паралелограми, які не ромби»*, *обов'язково* виразивши цю фразу через базові предикати $P(x)$ = *«фігура x є паралелограмом»* та $R(x)$ = *«фігура x є ромбом»*).
Вказівка. Записати кожну з «половин» твердження у вигляді предиката з обмеженим квантором, потім перетворити до «звичайного», потім з'єднати ці «половини» у цілісний вираз.

45. Записати у вигляді квантифікованого предиката (спочатку з обмеженим квантором, а потім «звичайним») твердження *«Корінь з натурального числа́ або цілий, або не раціональний»*.
У *цьому* завданні вводити «місцеві» позначення заборонено. Все треба виразити через позначення "$\in$", "$\notin$", "$\sqrt{\ }$", "$\mathbb{N}$", "$\mathbb{Z}$", "$\mathbb{Q}$", логічні операції та квантори.

46. Рекламно-нечітка фраза *«У нашому бюро працюють перекладачі, що вільно володіють англійською, німецькою, іспанською!»* насправді може описувати будь-яке з тверджень:

(а) $\forall x_{B(x)} \big(E(x) \wedge D(x) \wedge S(x)\big)$;
(б) $\forall x_{B(x)} E(x) \wedge \forall x_{B(x)} D(x) \wedge \forall x_{B(x)} S(x)$;
(в) $\exists x_{B(x)} \big(E(x) \wedge D(x) \wedge S(x)\big)$;
(г) $\exists x_{B(x)} E(x) \wedge \exists x_{B(x)} D(x) \wedge \exists x_{B(x)} S(x)$

(де $B(x)$ — *«x працює в бюро»*, $E(x)$ — *«x вільно володіє англійською»*, $D(x)$ — *«x вільно володіє німецькою»*, $S(x)$ — *«x вільно володіє іспанською»*). Які з цих тверджень мають однаковий смисл, і які різний? Пояснити відмінності між різними.

47. Нехай $P(a)$ означає *«a — вірш»*, $N(a)$ означає *«a — повість»*, $W(a, b)$ означає *«a є автором, що написав b»*, $L(a)$ означає *«a є літератором»*. Прочитати словами предикати:
(а) $\forall x (P(x) \vee N(x) \to W(\text{Франко}, x))$;
(б) $\forall x (W(\text{Франко}, x) \to P(x) \vee N(x))$;
(в) $\exists x \exists y (P(x) \wedge N(y) \wedge W(\text{Франко}, x) \wedge W(\text{Франко}, y))$;
(г) $\forall x \exists y \big(P(x) \to (L(y) \wedge W(y, x))\big)$;
(д) $\exists x \forall y \big(L(x) \wedge (P(y) \to W(x, y))\big)$;
(е) $\exists x \forall y \big(L(x) \wedge (P(y) \to \neg W(x, y))\big)$.
Примітка. Відповіддю цього завдання має бути твердження українською мовою, а не Ваше ставлення до того, істинне воно чи хибне. Навіть якщо отримане твердження явно дурне, не виключено, що формулою саме ця дурниця і записана.

48. Спростити аналітично $\neg\big(\exists x (\neg P(x))\big)$.

49. Довести за теоремою Поста функціональну <u>*не*</u>повноту системи $\{\to\}$ та функціональну повноту кожної з систем

(а) $\{\to, \oplus\}$;
(б) $\{\to, 0\}$.

50. Показати (таблицями істинності), що тотожність $(x \vee \neg y) \wedge (\neg x \vee y) = \neg(x \vee y) \vee (xy)$ виконується як у класичній логіці, так і в тризначній.

51. Показати (таблицями істинності), що тотожність $x \vee \neg xy = x \vee y$ виконується у класичній логіці, але <u>*не*</u> виконується у тризначній.

Додаткові завдання підвищеного рівня складності

1*. Довести тотожність $(cf \to ab) \to abcf = \big(c \to (df \to c)\big) \to cf$ чотирма способами:
   (а) аналітично;
   (б) побудувавши таблиці істинності в об'єднаному наборі змінних;
   (в) побудувавши окремі таблиці істинності в початкових наборах змінних і відкинувши фіктивні змінні;
   (г) замінивши рівність на "$\leftrightarrow$" і довівши методом Квайна тавтологічність отриманого виразу.

   У способах 1*б, 1*в заборонено використовувати які б не було елементи аналітичних спрощень. У способі 1*г придумати, як найкраще вибрати, які змінні підставляти, щоб і розв'язок вийшов короткий, і переважна більшість аналітичних перетворень зводилася до тотожностей з нулями й одиницями зі стор. 26.

2*. Довести $(x \vee y) \wedge (y \vee z) \wedge (z \vee x) = xy \vee yz \vee zx$, ретельно вказуючи крок за кроком кожну використану стандартну тотожність.

3*. Реалізувати алгоритм побудови ДДНФ та ДКНФ у вигляді програми мовою програмування. Вибір мови (C++, C#, Pascal, Python, тощо) — на розсуд студента. Вхідними даними алгоритму повинна бути таблиця істинності функції.

   Простіший варіант: зробити окремі програми для n=2 та для n=3. Складніший: програму, що працюватиме для всіх $1 \leqslant n \leqslant 20$.

4*. Запропонувати й аргументувати спосіб, яким можна легко й швидко отримати поліном Жегалкіна функції $z \vee \neg xy \neg z$. (Цей спосіб не зобов'язаний бути універсальним, достатньо, щоб він був правильним для са́ме цієї функції).

5*. Реалізувати алгоритм побудови ДДНФ та перетворення її до полінома Жегалкіна у вигляді програми мовою програмування. Вибір мови (C++, C#, Pascal, Python, тощо) — на розсуд студента. Вхідними даними алгоритму повинна бути таблиця істинності функції.

   Простіший варіант: зробити окремі програми для n=2 та для n=3. Складніший: програму, що працюватиме для всіх $1 \leqslant n \leqslant 20$.

6*. Реалізувати алгоритм побудови полінома Жегалкіна методом невизначених коефіцієнтів у вигляді програми мовою програмування. Вибір мови (C++, C#, Pascal, Python, тощо) — на розсуд студента. Вхідними даними алгоритму повинна бути таблиця істинності функції.

   Простіший варіант: зробити окремі програми для n=2 та для n=3. Складніший: програму, що працюватиме для всіх $1 \leqslant n \leqslant 20$.

7*. Показати суперечність (= тотожню хибність) виразу $(a \to (b \to c)) \wedge (a \to b) \wedge a \wedge \neg c$ двома способами: спираючись на ідеї методу Квайна та спираючись на ідеї методу редукції. Детально пояснити, що саме треба переробити у порівнянні зі стандартним методом Квайна і стандартним методом редукції.

8*. Нехай є масив `int a[99]`, причому всі 99 елементів (з 0-го по 98-й) реально використані. Записати у вигляді предикатів такі твердження.
   (а) «Елемент `a[0]` є максимальним у масиві» (тобто, нема елементів, строго більших за `a[0]`; при цьому не заборонено, щоб деякі інші елементи теж мали це саме максимальне значення).
   (б) «У масиві є різні елементи з однаковим значенням».
   (в) «Хоча б два елементи масиву мають однакове максимальне значення».
   (г) «Всі елементи зі значенням 0 знаходяться у серéдній третúні масиву» (іншими словами, елементи зі значенням 0 можуть розміщуватися лише в діапазоні індексів від 33 включно до 66 не включно).
   (д) «В масиві є хоча б один *опорний* елемент». Нагадаємо, що *опорним* (у алгоритмі QuickSort) називають такий елемент, що всі лівіші елементи менші-рівні за нього, а всі правіші — більші-рівні». Наприклад, у масиві `30 70 100 170 120 200 500 400` таких елементів два (`a[2]=100` та `a[5]=200`), а у масиві `10 5 7 2` опорних елементів нема.

В усіх пунктах цього завдання *вводити додаткові позначення* (аналогічні «нехай $M(x)$ означає, що x максимальний») *заборонено*. Треба виразити твердження, користуючись *лише* зверненнями до елементів масиву, стандартними математичними порівняннями ($=$, $<$, …), логічними операціями ($\wedge$, $\to$, …) та кванторами.

9*. Спираючись на означення «*функція* $f : \mathbb{R} \to \mathbb{R}$ опукла, *коли*

$$\forall x_1 \forall x_2 \left(f\Big(\frac{x_1 + x_2}{2}\Big) \leqslant \frac{f(x_1) + f(x_2)}{2} \right) \text{»},$$

перевірте, чи є опуклими функції $f_1(x) = x^2$ та $f_2(x) = x^3$.
Слід користуватися *лише* наведеним означенням, знаннями з логіки та предикатів, і знаннями шкільного курсу алгебри за 7–9 класи.

10*. Спираючись на означення «*число* A *являє собою границю нескінченної послідовності* $a_1, a_2, \ldots, a_n, \ldots$, *коли* $\forall \varepsilon_{\varepsilon>0} \exists N^\star \forall n_{n \geqslant N^\star} (|x_n - A| < \varepsilon)$», довести, що:
(а) число 2 є границею послідовності $a_n = \frac{2n+3}{n+2}$;
(б) число 0 *не* є границею послідовності $a_n = \sin \frac{\pi \cdot n}{180}$.
Слід користуватися *лише* наведеним означенням, знаннями з логіки та предикатів, і знаннями шкільного курсу алгебри за 7–9 класи.

11*. Довести: якщо таблиця істинності булевої функції від $n \geqslant 2$ змінних містить непарну кількість одиниць, така функція не лінійна (не належить класу Поста L).

12*. Виразити стандартні операції "$\neg$", "$\vee$", "$\wedge$" через (функціонально повні згідно аналізу зі стор. 38 та завдання 49 зі стор. 43) системи

(а) $\{\to, \neg\}$; (б) $\{\to, \oplus\}$; (в) $\{\to, 0\}$.

13*. (Потребу́є знань розд. 2.1, але лише дуже базових.)
(а) Довести, що $M \setminus (T_0 \cup T_1) = \varnothing$;
(б) з'ясувати, які функції входять до $M \setminus (T_0 \cap T_1)$ (детально пояснити).

14*. Враховуючи результати завд. 51 зі стор. 43, запропонувати спосіб, як легко й швидко знайти той набір тризначної логіки, на якому тотожність $xy\neg z \vee xy \vee z\neg z = xy$ (правильна у класичній логіці) не виконується у тризначній логіці.

2 Множини та відношення

2.1 Основні поняття теорії множин

2.1.1 Описове означення множини. Способи запису множини

Множина́ (рос. «*мно́жество*», англ. «*set*», рідше «*collection*», «*class*»)[15] — одне з пе́рвісних понять математики, для нього *нема* строгого математичного означення. Так са́мо, як нема строгого математичного означення числа́, точки, прямої... Але є не строгі описові означення.

Множина — це сукупність (набір, зібрання) певних відрізнюваних об'єктів (предметів, понять), що розглядається як єдине ціле. Самі́ об'єкти (предмети, поняття) є *елементами* (англ. «*elements*», «*members*») множини́.

Приклади множин: сукупність тролейбусів міста; зв'язка ключів; сукупність літер алфавіту; група студентів; сукупність студентів першого курсу факультету ОТІУС; сукупність груп першого курсу факультету ОТІУС (останні дві множи́ни різні: одна складається з ≈ 60 елементів-людей, інша — з шести елементів-груп (котрі в свою чергу теж є множинами)).

Щоб записати абстрактне позначення множини́, зазвичай пишуть велику латинську літеру, як-от «множина A». Крім абстрактних позначень (як «число x»), мусять бути і способи запису конкретних значень (як «число 3»). Є два основні способи записати конкретне значення множини́.

1. *Явний перелік* (рос. «*явный перечень*», англ. «*explicit enumeration*») елементів у фігурних дужках. Наприклад, множина голосних літер української абетки {*а*, *е*, *є*, *и*, *і*, *ї*, *о*, *у*, *ю*, *я*}; множина цифр {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; множина міст України з населенням більше мільйона {Київ, Харків, Одеса, Дніпро}.
   (Такий спосіб задання множин дуже простий, але зі зростанням кількості елементів множини́ стає не дуже зручним, а для нескінче́нних множин — взагалі неможливим. Звісно, можна ставити багатокрапку "..."; але раптом хтось зрозуміє цю багатокрапку не так, як мав на увазі автор запису?
2. за допомогою *характеристичного предиката*[16]: всере́дині фігурних дужок пишуть змінну, потім знак " | " (або " : "), потім предикат, вільна змінна якого збігається зі змінною, записаною перед " | ". (Тому що пишуть змінну та умову, що накладається на цю змінну.)

Так, множину непарних чисел можна записати як $\{n \mid n \bmod 2 \neq 0\}$; множину чисел з відрізка від 3 до 7 — як $\{x \mid x \geqslant 3 \wedge x \leqslant 7\}$; множину точних квадратів $\{1, 4, 9, 16, 25, \ldots\}$ — як $\{n \mid \sqrt{n} \in \mathbb{N}\}$.

У перших двох прикладах має місце вже відома проблема: неясно, про які чи́сла йде мова (натуральні чи цілі (т. ч. від'ємні) для першого прикладу? натуральні, раціональні чи дійсні для дру́гого?). Тому́, для більшої строгості можна записати ще й надмножину, з якої слід брати змінні для «перевірки» характеристичним предикатом. Наприклад, $\{x \mid x \in \mathbb{R} \wedge x \geqslant 3 \wedge x \leqslant 7\}$ (або, що те са́ме, $\{x \in \mathbb{R} \mid x \geqslant 3 \wedge x \leqslant 7\}$). Заодно, раз уже мова зайшла про стандартні числові множи́ни, нагадаємо їхні позначення:

$\mathbb{N}$ — множина натуральних чисел: $\{1, 2, 3, \ldots\}$;

$\mathbb{N}^+$, $\mathbb{N}_0$, $\mathbb{Z}^+$ — різні позначення множини́ цілих невід'ємних чисел, що містить всі натуральні чи́сла, а також нуль ($\{0, 1, 2, 3, \ldots\}$); (В англомовній термінології часто вважають, що 0 є натуральним числом; враховуючи, що останнім часом все більша частина джерел є перекладами з англійської, краще в разі найменшого сумніву уточняти, чи точно $0 \notin \mathbb{N}$.)

$\mathbb{Z}$ — множина цілих чисел: $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$;

$\mathbb{Q}$ — множина раціональних чисел (котрі можна подати у вигляді дробу $\frac{p}{q}$);

$\mathbb{R}$ — множина дійсних чисел;

$\mathbb{C}$ — множина комплексних чисел.

У школі, як правило, пишуть «прості» варіанти літер (N, Z, тощо); але у теорії множин більш прийнятий саме наведений запис, з «широкими» частинами літер.

Запис через характеристичний предикат можна застосовувати до будь-якої множини́ (не лише числової). Просто для чисел найлегше записати короткий аналітичний характеристичний предикат...

---

[15] І «множество», і «set» — багатозначні слова́, тому комп'ютерні програми-перекладачі часто перекладають ці слова́ *на* українську неправильно (як «set→набір» або як «множество→безліч»). Називати множину «набором» трохи некрасиво, але це не є грубою помилкою; називати ж множину «безліччю» — цілковита дурниця, така ж, як переклад «произведение чисел 7 и 5» у «твір чисел 7 та 5».

[16] рос. «*характеристический предикат*»; англ. мовою цей предикат рідко називають якимсь терміном, а весь спосіб задання множини́ називають «*implicit set specification*» або «*set builder notation*»

Запис через характеристичний предикат можна узагальнити, дозволивши писати перед знаком " | " не просто змінну, а вираз від вільної змінної (вільних змінних) характеристичного предиката. Наприклад, множину па́рних натуральних чисел тоді можна записати як $\{2k \mid k \in \mathbb{N}\}$; множину раціональних чисел — як $\mathbb{Q}=\{\frac{m}{n} \mid m \in \mathbb{Z} \wedge n \in \mathbb{N}\}$.

2.1.2 Порожня множина та універсум

Порожня множина (позначається "$\varnothing$", іноді "$\emptyset$"; рос. «*пустое множество*», англ. «*empty set*») — це множина, у якій нема жодного елемента.

(Наприклад, множина людей, прописаних на Місяці. Або множина трамваїв, що беруть участь у «Формулі–1». Або множина дійсних розв'язків рівняння $x^2 + 1 = 0$.)

Універсум (позначається "U" або "$\mathcal{U}$", іноді "E"; рос. «*универсум*», англ. «*universe*») — це множина, у якій є (котрій належать) усі елементи.

(«Усі» зазвичай розуміють як «усі, що можуть знадобитися», а не «взагалі всі на світі». Наприклад, розглядаючи таку «серію множин», як множи́ни студентів певної групи, що були присутні на різних парах, за універсум доцільно взяти множину всіх студентів цієї групи, але не варто включати в цей універсум стільці, ві́дра, тролейбуси та їхніх кондукторів. Якщо розглядати таку «серію множин», як всі орфографічно правильні слова́ деякої природньої мови, ввічливі офіційно-ділові слова́ цієї ж мови, діалектизми цієї ж мови — в якості універсуму можна взяти, наприклад, сукупність всіх можливих послідовностей літер алфавіту цієї мови, що мають розумну довжину.

Тобто, універсум для різних задач може бути різним; більш того, для деяких задач визначити універсум досить складно.)

2.1.3 Основні співвідношення для множин

Між об'єктом (предметом, поняттям) з одного боку, та множиною з іншого боку може виконуватися (або не виконуватися) відношення «*належить*» (рос. «*принадлежит*»). Належність позначається символом "$\in$", не належність — "$\notin$". Наприклад, $\neg \in \{\vee, \wedge, \neg\}$; $7 \in \mathbb{N}$; $\sqrt{2} \in \mathbb{R}$. А також $\sqrt{2} \notin \mathbb{Q}$; $ы \notin \{є, і, ї\}$; $\mathbb{N} \notin \mathbb{Z}$ (будь-яке натуральне число є цілим, але *множина* натуральних чисел *не* є цілим *числом*!).

(Відношенню належності важко дати чітке означення, бо це — суттєва складова поняття множи́ни. Англійською мовою взагалі нема відповідного терміна, кажуть просто «*... is an element of ...*» або «*... is not an element of ...*».)

Множи́ни *рі́вні* (рос. «*равны́*», англ. «*equal*»; позначається символом "="), коли складаються з одних і тих самих елементів:

$$A = B \overset{\text{def}}{\iff} \forall x\big((x \in A) \leftrightarrow (x \in B)\big). \tag{3}$$

(Формулою записане те са́ме означення, що й словами. Ліворуч від символу "$\overset{\text{def}}{\iff}$" записують поняття, якому дається означення. Праворуч — вираз, який виражає це поняття через відомі. Очевидно, "$(x \in A) \leftrightarrow (x \in B)$" якраз і означає, що x або одночасно належить обом множинам, або не належить жодній.

Це означення рівності множин називають також *аксіомою екстенсіональності* (рос. «*аксиома экстенсиональности*», англ. «*axiom of extensionality*»). Таке страшне словосполучення не означає нічого дуже мудрого, це просто одна з назв досить простого означення «множи́ни рі́вні, коли складаються з одних і тих самих елементів»...)

Зокрема, це означає, що *нема* такого поняття, як «порядок елементів множи́ни». Наприклад, $\{\alpha, \beta, \gamma\}$, $\{\alpha, \gamma, \beta\}$, $\{\gamma, \beta, \alpha\}$ — різні записи *однієї й тієї ж* множи́ни. Так само (класична) множина не може багатократно містити один і той самий елемент; наприклад, $\{h, e, l, l, o, w, o, r, l, d\}$ — некрасивий запис множи́ни $\{h, e, l, o, w, r, d\}$.

(Коли потрібно враховувати кратності елементів, користуються т. зв. *мультимножинами* (рос. «*мультимножество*», англ. «*multiset*»); в межах цього посібника вони детально не розглядаються.)

Множина A є *підмножиною* (рос. «*подмножество*», англ. «*subset*») множи́ни B, коли кожен елемент множи́ни A є також елементом множи́ни B:

$$A \subseteq B \overset{\text{def}}{\iff} \forall x\big((x \in A) \to (x \in B)\big). \tag{4}$$

Наприклад, $\{2, 5\} \subseteq \{2, 3, 5, 7\}$; $\mathbb{N} \subseteq \mathbb{R}$. Іноді знак "$\subseteq$" «розвертають», пишучи "$\supseteq$" (наприклад, $\mathbb{R} \supseteq \mathbb{Z}$), але це не дуже рекомендується.

Є багато термінів–синонімів до «A є підмножиною B»: «A входить до B», «A включається у B», «B включає A», «B є надмножиною A»; рос. — «A входит в B», «A содержится в B», «B содержит A», «B является надмножеством A»; англ. — «A is a subset of B», «A is contained in B», «B contains A», «B is a superset of A». Але *ні в якому разі не можна* читати "$A \subseteq B$" як «A належить B», бо "$\in$" та "$\subseteq$" — різні співвідношення!

Будь-яка множина включає порожню множину і будь-яка множина включає саму себе:

$$\varnothing \subseteq A, \quad A \subseteq A.$$

(Са́ме «*включає* (як підмножину)», а *не* «*містить* (як елемент)». Щодо належності як елемента, $\varnothing$ нічим не краща, за, скажімо, число 12345, яке теж деяким множинам належить, а решті (більшості) — ні.)

Множи́ни рівні тоді й тільки тоді, коли кожна є підмножиною іншої:

$$(A = B) \;=\; (A \subseteq B) \wedge (B \subseteq A).$$

Наведені формули "$\varnothing \subseteq A$", "$A \subseteq A$", "$(A = B) = (A \subseteq B) \wedge (B \subseteq A)$" — не означення, а властивості; строге викладення цього матеріалу для «чистих» математиків повинно було б містити доведення, що кожна з цих формул справді слідує з раніше зроблених означень. Оскільки цей посібник призначений для інженерних спеціальностей, а не математичних, відповідні доведення все ж пропущені; але це робить викладення менш строгим.

A — *власна підмножина* (рос. «*собственное подмножество*», англ. «*proper subset* of») B, коли A — підмножина B і при цьому $A \neq B$. Ми будемо записувати це як $A \subset B$.

(На жаль, значок "$\subset$" у різних книгах має різне значення: в одних — власна підмножина, в інших — просто підмножина, тобто те саме, що у нас "$\subseteq$". Іноді власну підмножину позначають як "$\subsetneq$", але цей значок ще менш поширений. «Розвертати» значок "$\subset$" до вигляду "$\supset$" (власна *над*множина) тим паче не рекомендується, бо до всіх щойно згаданих проблем додається ще й та, що у деяких джерелах так позначають імплікацію.)

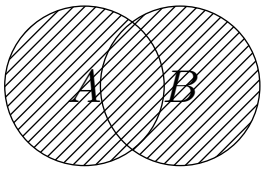
Множинні порівняння "$\subseteq$" та "$\subset$" дещо схожі на числові порівняння "$\leqslant$" та "$<$". Але є суттєва відмінність: бувають множи́ни, жодна з яких не входить до іншої (наприклад, $\{a, b, c, d\}$ та $\{b, p\}$).

2.1.4 Основні операції над множинами

Наведені словесні фрази та формули слід вважати власне означеннями; малюнки (котрі називаються *діаграми Венна*)[17] — ілюстраціями до них. Адже рисунки менш строгі й мають меншу математичну цінність, ніж формули.

(Не плутати діаграми Ве́нна з діаграмами Ве́йча, вони ж карти Карно (розд. 1.4.1).)

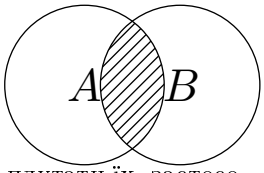
Об'єднання (рос. «*объединение*», англ. «*union*»; позначається "$\cup$") множин A і B — це множина, що складається з усіх елементів, які належать хоча б одній з множин A, B:



$$A \cup B \stackrel{\text{def}}{=} \{x \mid x \in A \vee x \in B\}. \tag{5}$$

(Значок "$\stackrel{\text{def}}{=}$", як і "$\stackrel{\text{def}}{\Longleftrightarrow}$", використовують, щоб підкреслити: це означення, тут вперше вводиться те, що записане зліва від цього значка; відмінність між "$\stackrel{\text{def}}{\Longleftrightarrow}$" та "$\stackrel{\text{def}}{=}$" та, що праворуч від "$\stackrel{\text{def}}{\Longleftrightarrow}$" пишуть *умову, коли виконується* те, що зліва від нього, а праворуч від "$\stackrel{\text{def}}{=}$" пишуть *вираз, результату якого дорівнює* те, що зліва від нього.)

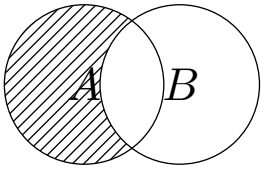
Перетин (рос. «*пересечение*», англ. «*intersection*»; позначається "$\cap$") множин A і B — це множина, що складається лише з тих елементів, які належать одночасно обом множинам A, B:



$$A \cap B \stackrel{\text{def}}{=} \{x \mid x \in A \wedge x \in B\}. \tag{6}$$

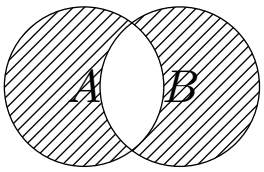
(Отже, між значками "$\cup$" та "$\vee$" (а також "$\cap$" та "$\wedge$") є глибокий взаємозв'язок; але плутати їх, застосовуючи "$\wedge$" до множин або "$\cup$" до булевих значень — груба помилка.)

[17] Їх називають також *діаграмами Ейлера–Венна*, а іноді й *кругами Ейлера*, але це не зовсім одне й те само: круги Ейлера, залежно від обставин, можна зображати, як тут, а можна так, що вони не перетинаються, або якийсь з них повністю всере́дині іншого; для діаграм Венна обов'язково, щоб були всі 2^n областей (див. також розд. 2.2.2).



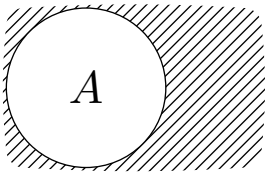
Різниця (рос. «*разность*», англ. «*difference*»; позначається "$\setminus$", іноді "$-$") множин A і B — це множина, що складається з тих елементів A, які не належать B:

$$A \setminus B \stackrel{\text{def}}{=} \{x \mid x \in A \wedge \neg(x \in B)\}. \tag{7}$$



Симетрична різниця (рос. «*симметрическая разность*», англ. «*symmetric difference*»; позначається "$\div$", нерідко "$\triangle$", іноді "$\oplus$") множин A і B — це множина, що складається з елементів, які належать в точності одній з множин A, B:

$$\begin{aligned} A \div B &\stackrel{\text{def}}{=} \{x \mid x \in A \oplus x \in B\} = \\ &= \{x \mid \big(x \in A \wedge \neg(x \in B)\big) \vee \big(\neg(x \in A) \wedge x \in B\big)\}. \end{aligned} \tag{8}$$



Доповнення (рос. «*дополнение*», англ. «*complement*»; позначається "A'", іноді "$\overline{A}$", "C_A") множи́ни A — це множина всіх елементів, які не належать A:

$$A' \stackrel{\text{def}}{=} \{x \mid \neg(x \in A)\}; \tag{9}$$

операція доповнення осмислена лише коли є чітко заданий універсум.

(Різниця та доповнення взаємопов'язані тотожністю $A' = U \setminus A$ (де U — універсум). Через це різницю $A \setminus B$ навіть називають «*відносне доповнення*» або «*доповнення B до A*» («*relative complement of B with respect to A*»); відповідно, унарне доповнення називають *абсолютним* («*absolute complement*»).)

(В означенні різниці, замість "$\neg(x{\in}B)$" можна написати "$x{\notin}B$", що виглядає коротшим. Але слід розуміти, що у коротшому записі "$x{\notin}B$" теж мається на увазі заперечення. Наприклад, $x{\notin}(C{\cap}D)$ ні в якому разі *не* рівносильне $(x{\notin}C){\wedge}(x{\notin}D)$; при бажанні, його можна перетворити як $x{\notin}(C{\cap}D) = \neg(x{\in}(C{\cap}D)) = \neg\big((x{\in}C) \wedge (x{\in}D)\big) = \neg(x{\in}C) \vee \neg(x{\in}D) = (x{\notin}C) \vee (x{\notin}D)$, використавши закон де Моргана.)

Згадаємо ще одну множинну операцію, яка по смислу сильно відрізняється від вищезгаданих.

Булеа́н (*множина всіх підмножин*, англ. «*power set*»; позначається 2^A, $\beta(A)$, $\wp(A)$) множи́н A — це множина, елементами якої є всі можливі підмножи́ни множи́н-аргумента.

Наприклад, $2^{\{a,b,c\}} = \{\varnothing,\ \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$.

Неважко переконатися, що коли множина A скінче́нна і складається з n елементів, то 2^A — скінче́нна множина, що складається з 2^n елементів.

2.2 Доведення тотожностей і включень. Аналітичні перетворення виразів над множинами

2.2.1 Перетворення характеристичних предикатів

Безпосередньо в означеннях операцій над множинами вказані характеристичні предикати множин–результатів. Тому́, коли є скла́дений вираз над множинами, зовсім не важко побудувати його характеристичний предикат у вигляді логічного виразу від предикатів вигляду "$x \in \ldots$" (замість "$\ldots$" підставляються множи́ни, що згадуються у виразах). А щоб проаналізувати, чи тотожні логічні вирази, є дуже простий спосіб — таблиці істинності.

Приклад 1. Доведемо тотожність $(A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$.

Ліва частина досліджуваної тотожності згідно означень перетворюється як $(A \cup B) \setminus (A \cap B) = \{x \mid x \in (A \cup B) \wedge \neg(x \in (A \cap B))\} = \{x \mid \underbrace{(x \in A \vee x \in B)}_{\text{Л1}} \wedge \underbrace{\neg(\underbrace{x \in A \wedge x \in B}_{\text{Л2}})}_{\text{Л3}}\}$. Права — як

$(A \setminus B) \cup (B \setminus A) = \{x \mid x \in (A \setminus B) \vee (x \in (B \setminus A))\} = \{x \mid \underbrace{(x \in A \wedge \underbrace{\neg(x \in B)}_{\text{П1}})}_{\text{П2}} \vee \underbrace{(x \in B \wedge \underbrace{\neg(x \in A)}_{\text{П3}})}_{\text{П4}}\}$.

(Тут скрізь пишемо "$=$", а не "$\stackrel{\text{def}}{=}$", бо *використовуємо* означення, а не *даємо* їх. І взагалі, якщо є сумнів, краще писати "$=$".)

Тепер обидва характеристичні предикати залежать від однакових базових предикатів "$x \in A$" та "$x \in B$". Тому́ можна побудувати таблиці істинності:

$x{\in}A$	$x{\in}B$	Л1	Л2	Л3	уся ліва	П1	П2	П3	П4	уся права
0	0	0	0	1	0	1	0	1	0	0
0	1	1	0	1	1	0	0	1	1	1
1	0	1	0	1	1	1	1	0	0	1
1	1	1	1	0	0	0	0	0	0	0

Стовпчики істинності для лівої частини та правої частини співпали — отже, характеристичні предикати однакові. А значить, однакові й множи́ни.

Коли потрібно дослідити включення (“$\subseteq$”) скла́дених виразів над множинами, це теж досить просто зробити через перетворення характеристичних предикатів. Адже $S_1 \subseteq S_2$ означає $\forall x\big((x \in S_1) \to (x \in S_2)\big)$; значить, достатньо розписати всере́дині цього виразу “$x \in S_1$” та “$x \in S_2$” і подивитися, чи завжди істинна імплікація.

Приклад 2. Доведемо, що $(A \cap B) \subseteq (A \div B)'$. За означенням, це включення правильне тоді й тільки тоді, коли вираз $(x \in (A \cap B)) \to (x \in (A \div B)')$ завжди істинний. Перетворимо його.

$$(x \in (A \cap B)) \to (x \in (A \div B)') =$$
$$= (\underbrace{x \in A \wedge x \in B}_{1}) \to \underbrace{\neg(\underbrace{x \in A \oplus x \in B}_{2})}_{3}.$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{4}$$

$x{\in}A$	$x{\in}B$	1	2	3	4
0	0	0	0	1	1
0	1	0	1	0	1
1	0	0	1	0	1
1	1	1	0	1	1

Отримали, що “вираз 1” $\to$ “вираз 3” виконується завжди. Отже, включення правильне.

(Строго кажучи, розглянуто лише 4 випадки $(x \notin A,\ x \notin B)$, $(x \notin A,\ x \in B)$, $(x \in A,\ x \notin B)$ та $(x \in A,\ x \in B)$. А в означенні включення стоїть квантор “$\forall x$”, що задає перебір усіх можливих елементів універсуму (можливо, нескінче́нного). Тому́, для більшої строгості, висновок можна переформулювати приблизно так: «Отримали, що вираз “$(x \in (A \cap B)) \to (x \in (A \div B)')$” істинний в усіх чотирьох перелічених випадках; кожен елемент x універсуму потрапляє до одного з цих чотирьох випадків; таким чином, згадана імплікація виконується для будь-якого елемента універсуму. Отже, досліджуване включення справді правильне.»)

2.2.2 Побудова діаграм Венна

Побудову діаграм багато хто не вважає математичним доведенням: мовляв, один розмістить області діаграми так, інший інакше; де гарантія, що вони обов’язково прийдуть до однакових результатів?..

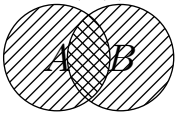
Але побудова діаграм може бути цілком строгим способом аналізу виразів над множинами, нітрохи не гіршим за таблиці істинності — *якщо* дотримуватися двох простих правил:

1. Щоб порівняти вирази, їхні області будують на різних примірниках однаково влаштованих діаграм (на діаграмах повинні бути зображені одні й ті самі множи́ни, причому однаково розміщені).
2. На діаграмі, яка містить n множин, повинні бути всі області, відповідні усім 2^n випадкам належності до частини з цих множин (аналогічно 2^n рядкам таблиці істинності).

2-ге правило, по суті, вимагає, щоб діаграми були са́ме діаграмами Венна, а не кругами Ейлера (див. також прим. 17 на стор. 48). Якщо порушувати це правило, можна (помилково) «доводити» неправильні твердження. Наприклад, розглянемо «діаграму», де множи́ни A і B позначені круга́ми, що дотикаються (зовнішнім чином), але не перетинаються. Зобразимо на них “$A \cup B$” та “$A \div B$”. В обох випадках виходить однаковий результат “  ”. Звідси можна зробити «висновок» про нібито рівність цих виразів, і причина помилки — у то́му, що на «діаграмі» пропущена якраз та область, де і виявляється відмінність між виразами.

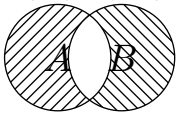
Приклад 1. Доведемо через побудову діаграм (вже доведену через характеристичні предикати) тотожність $(A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$.

$(A \cup B) \setminus (A \cap B)$



Замалюємо $(A \cup B)$ «правою» штриховкою, $(A \cap B)$ — «лівою». До них треба застосувати "\", отже остаточний результат — область, де є «права» штриховка і нема «лівої».

$(A \setminus B) \cup (B \setminus A)$

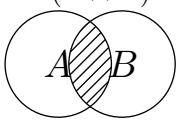


Замалюємо $(A \setminus B)$ «правою» штриховкою, $(B \setminus A)$ — «лівою». До них треба застосувати "∪", отже остаточний результат — уся область, де є хоч якась штриховка.

Бачимо, що області однакові. Отже, тотожність справді виконується.

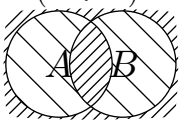
Приклад 2. Доведемо через побудову діаграм (вже доведене через характеристичні предикати) включення $(A \cap B) \subseteq (A \div B)'$.

$(A \cap B)$



Замалюємо $(A \cap B)$.

$(A \div B)'$



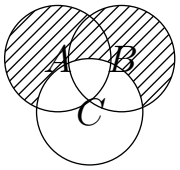
Замалюємо $(A \div B)$ ріденькою «лівою» штриховкою. До нього слід застосувати доповнення, тож замальовуємо все інше густою «правою» штриховкою.

Бачимо, що область лівого виразу є частиною області правого виразу. Що очевидно відповідає співвідношенню "⊆".

Більш того: при наявності певного досвіду зовсім не важко будувати діаграми значно складні́ших виразів, залежних від трьох множин.

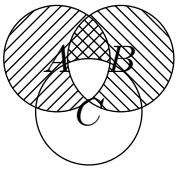
Приклад 3. Доведемо $(A \cup B) \setminus C \subseteq \big(A \setminus (B \cap C)\big) \cup \big(B \setminus (A \cap C)\big)$.

$(A \cup B) \setminus C$



$(A \cup B)$ зовсім не важко уявити, дивлячись на діаграму, але не штрихуючи. Тож заштрихуємо зразу $(A \cup B) \setminus C$.

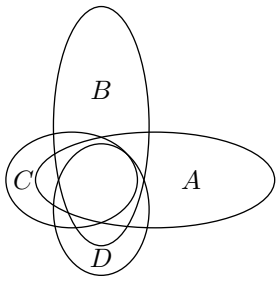
$\big(A \setminus (B \cap C)\big) \cup \big(B \setminus (A \cap C)\big)$

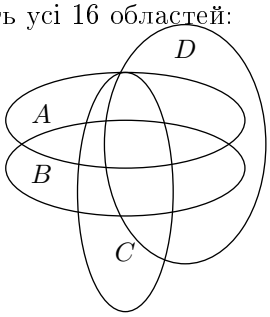


Аналогічно, $B \cap C$ уявимо, а заштрихуємо (////) зразу $A \setminus (B \cap C)$; так само $A \cap C$ уявимо, а заштрихуємо (\\\\) $B \setminus (A \cap C)$. До них треба застосувати "∪", отже результат — область, де є (хоч якась) штриховка.

Бачимо, що область лівого виразу є частиною області правого виразу. Що очевидно відповідає співвідношенню "⊆".

Але коли множин більше трьох, будувати діаграми, як правило, недоцільно. На рис. зображено два варіанти «кістяка» діаграм для 4-х множин, що містять усі 16 областей:





Конструкції вельми складні́ (а при більшій кількості множин ще набагато складніші), в них легко помили́тися й пропустити якусь область чи навіть кілька. А як вже показано, при пропуску хоча б однієї області діаграми можуть втрачати доказову силу. Тому й виходить, що для n=2 та n=3 діаграми досить зручні, а при $n \geqslant 4$ зазвичай недоцільні.

2.2.3 Стандартні тотожності для множин

Враховуючи описаний зв'язок між операціями над множинами та операціями над їхніми характеристичними предикатами, природньо, що мають місце стандартні тотожності множинних виразів — аналоги стандартних тотожностей для логічних виразів:

1. Комутативність
$$A \cup B = B \cup A, \quad A \cap B = B \cap A.$$
2. Асоціативність
$$(A \cup B) \cup C = A \cup (B \cup C), \quad (A \cap B) \cap C = A \cap (B \cap C).$$
3. Дистрибутивність
$$(A \cup B) \cap C = (A \cap C) \cup (B \cap C), \quad (A \cap B) \cup C = (A \cup C) \cap (B \cup C).$$
4. Закони порожньої множини, універсуму і доповнення
$$A \cup A' = U, \quad A \cap A' = \varnothing, \quad A \cup \varnothing = A, \quad A \cap U = A.$$

Якщо додати ще способи вираження різниці та симетричної різниці
$$A \setminus B = A \cap B', \quad A \div B = (A \cap B') \cup (A' \cap B),$$
отримаємо набір, достатній для аналітичних перетворення виразів над множинами. Як і для логічних операцій, мінімальний набір зручно розширити, ввівши ідемпотентність, аналоги законів де Морга́на, інволютивність (аналог закона подвійного заперечення), закони поглинання, тощо.

2.3 Способи подання множин

Зупинимося лише на способах явного переліка («explicit-них») — адже «implicit-ними» можна вважати хіба що способи аналітико-алгоритмічного подання характеристичних предикатів, про які не можна сказати нічого нового порівняно з розд. 1.6.2.

2.3.1 Бітові вектори

Нехай універсум чітко відомий і досить малий. Тоді можна занумерувати всі елементи універсуму (взагалі всі можливі елементи) числами проміжку $0 \leqslant i <$ `MAXN`, і працювати з масивами вигляду `bool a[MAXN]`.

Значення множини́ визначається тим, які елементи їй належать. Нехай кожен `i`-ий елемент масиву зберігає інформацію про те, чи належить відповідний елемент універсуму a_i цій множині, чи не належить:
$$\texttt{a[i]} = (a_i \in A) = \begin{cases} \texttt{true}, & a_i \in A; \\ \texttt{false}, & a_i \notin A. \end{cases}$$

Наприклад, якщо $U = \{0,1,2,3,4,5\}$, то множина $\{2,3,5\}$ подається як `{false, false, true, true, false, true}`, тобто

0	1	2	3	4	5
–	–	+	+	–	+

.

Операції над множинами, поданими у вигляді бітових векторів, реалізуються просто й однотипно:

$C = A \cup B$	`for(int i=0; i<MAXN; i++) c[i] = a[i] || b[i];`
$C = A \cap B$	`for(int i=0; i<MAXN; i++) c[i] = a[i] && b[i];`
$C = A \setminus B$	`for(int i=0; i<MAXN; i++) c[i] = a[i] && !b[i];`
$C = A \div B$	`for(int i=0; i<MAXN; i++) c[i] = a[i] ^ b[i];`
$C = A'$	`for(int i=0; i<MAXN; i++) c[i] = !a[i];`

(Якщо не обмежуватися класичними масивами, можна зменшити витрати пам'яті, виділяючи на кожен елемент універсуму всього один біт. (Звідси й походить назва «бітовий вектор», яку умовно поширюють і на подання у масиві.) Причому, мовою C++ це можна робити як низькорівневими засобами доступу до окремих бітів, так і простим використанням `vector<bool>`. Правда, на службову інформацію `vector`-а теж витрачається пам'ять, але коли елементів універсуму (масиву) досить багато (хоча б сотні) — економія пам'яті є. Щоправда, за рахунок сповільнення виконання.)

2.3.2 Подання відсортованим переліком. Злиття

Дещо несподівано, але факт: ***впорядкованими переліками*** *можна користуватися* ***завдяки*** *тому, що* ***множи́ни не впорядковані***. Наприклад, $\{2,7,5\}$, $\{5,2,7\}$, $\{5,7,2\}$, ... — різні записи *однієї й тієї ж* множини́; *тому́* можна вважати «умовно найправильнішим» запис (`2`, `5`, `7`). І так для кожної множини́: оскільки порядок елементів не впливає на значення множини́, можна вважати «умовно найправильнішим» запис у відсортованому порядку.

При відсортованому поданні можна виконувати основні операції над множинами на основі т. зв. *злиття* (рос. «*слияние*», англ. «*merge*»). Опишемо модифікацію алгоритму злиття, що виконує об'єднання множин $A \cup B$.

Нехай множи́ни–вхідні дані задані у відсортованих `vector`-ах `a` та `b`, результат треба повернути як `vector`–результат функції. (де `vector` — це така сучасна продвинута версія масиву, яка сама знає свій розмір (метод `size()`) і вміє додавати елемент собі в кінець (метод `push_back(...)`)). Надалі називатимемо `vector` масивом, бо відмінності суто технічні.

Результат (`res`) спочатку порожній. Елемент `a[0]` найменший серед усіх елементів масиву `a`, елемент `b[0]` — серед масиву `b`. Значить, менший з цих двох елементів є найменшим серед взагалі всіх елементів $A \cup B$, тобто його і треба розмістити у масив `res` як 0-й (найменший) елемент об'єднання. Цей найменший елемент стає розглянутим, і до нього вже не треба повертатися. Тож можна продовжити робити те са́мо для решти елементів. А щоб перейти до решти, зсуваємо поточну позицію (індекс) у тому з масивів, звідки взятий цей мінімальний елемент, і далі проводимо аналогічні порівняння.

Можлива ситуація, коли у множинах A і B є однакові елементи; для об'єднання ("$\cup$") їх треба включити у результат, але один раз. Найприродніше врахувати це так: при порівнянні поточних елементів `a[i]` та `b[j]` розрізняти не два випадки, а три: `a[i]`$<$`b[j]`, `a[i]`$>$`b[j]`, `a[i]`$=$`b[j]`.

(Коли потрібне не "$\cup$", а таке поєднання, де наявність одного й того ж значення і в `a`, і в `b` означає, що його слід записати у результат двічі — можна обійтися простішим `if`-ом "`if(a[i]<=b[j]) res.push_back(a[i++]); else res.push_back(b[j++]);`"; три гілки спричинені якраз потребою враховувати однакове лише один раз.)

Завершується злиття, коли вже нема чого зливати, бо дійшли до кінця. Зазвичай, якийсь один зі вхідних масивів закінчуватиметься, коли в іншому ще є не оброблені елементи (можливо, багато). Томý «основну частину» злиття треба робити, поки ще є вхідні дані в обох масивах; після «основної частини» треба ще переписати «хвіст» недообробленого масиву (якщо є).

Реалізації "$\cap$", "$\div$", "$\setminus$" відрізняються головним чином тим, в яких са́ме гілках потрібно заносити поточний елемент до результуючої множини́:

	`a[i]<b[j]`	`a[i]>b[j]`	`a[i]==b[j]`	дописування хвоста `a`	дописування хвоста `b`
$A \cup B$	так	так	так	так	так
$A \cap B$	ні	ні	так	ні	ні
$A \setminus B$	так	ні	ні	так	ні
$A \div B$	так	так	ні	так	так

У наведеному переліку *нема і не повинно бути операції доповнення*. Адже при поданні впорядкованими послідовностями та при злитті універсум взагалі не фіксується. (Точніше, він фіксується при оголошенні типу, чи це `vector<short int>`, чи `vector<double>`, тощо); але це *далеко не* така жорстка фіксація, як при поданні бітовими векторами.) Якщо все-таки виникає потреба в операції доповнення, можна зберігати універсум U «на правах звичайної множини́», і знаходити A' як $U \setminus A$.

Тепер про перевірку належності ("$\in$"). Якщо важливі лише простота й очевидність, можна просто переглядати масив, порівнюючи кожен елемент з шуканим значенням. Якщо важлива ефективність, то, знов-таки завдяки відсортованості, слід застосувати *бінарний пошук*. Тобто, починаємо з елемента посереди́ні масиву. Можливо, він якраз дорівнює шуканому значенню, тоді пошук успішно завершується. Інакше, перевіряємо, що менше. Якщо середній елемент більший шуканого, цим з'ясовано (завдяки відсортованості), що й уся права половина більша, тож значення варто шукати *лише* у лівій половині. Аналогічно, якщо середній елемент менший шуканого, надалі слід шукати лише у правій половині. Розмір проміжку, який треба розглядати, щоразу зменшується приблизно удвічі — отже, щоб знайти елемент (або переконатися у його відсутності)

Алгоритм може бути реалізований, наприклад, так:

```
```

Ні `vector`-и, ні `template`-и не є обов'язковими засобами реалізації злиття. Цілком корисно й доступно сильному студенту 1-го курсу написати злиття самостійно, на основі лише раніше наведених словесних описів та користуючись добре відомими засобами (`vector`-и можна замінити простими масивами, можна без `template`-ів вказати конкретний тип, наприклад, `int`). Просто `vector` дозволяє зручно додавати елемент у кінець (`push_back`), а `template` дозволяє зробити зразу для всіх типів, а не лише для `int`...

Приклад об'єднання злиттям $\{2,3,4,6,7\}$ та $\{1,6,8,9,12\}$.

Вхід 1	Вхід 2	Результат	Примітка
2 3 4 6 7	[1] 6 8 9 12	[1]	$2 > 1$, вибираємо 1
[2] 3 4 6 7	1 | 6 8 9 12	1 [2]	$2 < 6$, вибираємо 2
2 [3] 4 6 7	1 | 6 8 9 12	1 2 [3]	$3 < 6$, вибираємо 3
2 3 [4] 6 7	1 | 6 8 9 12	1 2 3 [4]	$4 < 6$, вибираємо 4
2 3 4 [6] 7	1 [6] 8 9 12	1 2 3 4 [6]	6=6, беремо спільне значення 6 (один раз) та зсуваємо обидві поточні позиції
2 3 4 6 [7]	1 6 | 8 9 12	1 2 3 4 6 [7]	$7 < 8$, вибираємо 7
Основний етап злиття завершено, бо закінчилася одна з послідовностей. Розпочинаємо дописування хвоста.			
2 3 4 6 7 |	1 6 [8] 9 12	1 2 3 4 6 7 [8]	просто дописуємо хвіст
2 3 4 6 7 |	1 6 8 [9] 12	1 2 3 4 6 7 8 [9]	просто дописуємо хвіст
2 3 4 6 7 |	1 6 8 9 [12]	1 2 3 4 6 7 8 9 [12]	просто дописуємо хвіст

бінарним пошуком, потрібно приблизно $\log_2 n$ порівнянь. Наприклад, для мільйона елементів — всього $\approx$20.

2.3.3 Порівняння розглянутих способів подання

Фрагменти програм, які стосуються бітових векторів, незрівнянно коротші й простіші, ніж злиття. Але бітові вектори мають ряд недоліків.

Якщо універсум зарані не відомий (нові елементи можуть з'являтися під час роботи програми) або дуже великий, то бітові вектори взагалі незастосовні. Якщо універсум відомий і малий, але «незручний» (наприклад, елементами є слова́), застосування бітових векторів потребу́є постійного «перекладу» незручних назв у номери і навпаки, а робити це більш-менш ефективно — складніше, ніж реалізувати злиття і бінарний пошук.

Тепер розглянемо ситуацію, коли універсум великий, а конкретні множи́ни невеликі. За таких умов застосування бітових векторів можливе, але навряд чи доцільне. По-перше, об'єм пам'яті, необхідний для зберігання кожної множини́ у вигляді бітового вектора, пропорційний кількості елементів усього універсуму, а у вигляді переліку — лише кількості реально використаних у множині елементів. По-друге, операції "$\cup$", "$\cap$", ... через злиття вимагають перегляду лише реально існуючих елементів і в результаті будуть виконуватися швидше, ніж цикл `for(int i=0; i<MAXN; i++) ...`, який без толку переглядатиме величезну кількість незадіяних елементів універсуму.

Але якщо доводиться мати справу з множинами, яким належить значна частина елементів універсуму, кращими виявляються бітові вектори. В такому разі зберігання `MAXN` бітів імовірно вимагатиме менше пам'яті, ніж зберігання ненабагато меншої кількості значень елементів (кожне з яких може займати кілька байтів). Крім того, дуже простий цикл `for(int i=0; i<MAXN; i++) ...` працюватиме мабуть швидше, ніж злиття, яке має виконати ненабагато менше складніших ітерацій.

Незалежно від «щільності» використання універсуму, перевірка належності елемента множині робиться ефективніше у бітових векторах (подивитися прямим доступом значення `a[i]` швидше, ніж запускати бінарний пошук, не кажучи вже про повний перегляд масиву). Але коли треба виконати якусь дію для кожного елемента множини́, то подання переліком надає ці елементи у

готовому вигляді один за одним, тоді як по бітовому вектору треба «бігти», перевіряючи абсолютно кожен елемент універсуму.

(Бітові вектори мають багато спільного з розглянутим у розд. 1.6.2 табличним поданням предикатів, а подання відсортованими масивами — з розглянутим там са́мо переліком наборів. Більш того: ці са́мі ідеї в новому ракурсі з'являться ще й при поданні графів, у розд. 5.2.2 та 5.2.3. Що ж, свіжих ідей мало, областей застосування гарних ідей багато...

Але *не слід* робити висновок, ніби відомі *лише* два розглянуті способи подання множин. Є ще багато способів, теж основаних на зберіганні переліку елементів множи́н, але в деякій технічно складнішій структурі даних — наприклад, `set` бібліотеки STL зберігає у збалансованому бінарному дереві пошуку. Але ці деталі вже стосуються аналізу алгоритмів, а не математики.)

2.4 Декартів добуток

(Термін «дека́ртів» (або «дека́ртовий») походить від прізвища видатного математика та філософа Рене Дека́рта[18]. Строго кажучи, Декарт лише запропонував основну ідею, застосувавши її до декартової системи координат на площині (кожній точці площини співставляються *два* числа́) та до нумерації місць у театрі («ряд ..., крісло ...»). Сучасний смисл множинної операції «декартів добуток» ввели пізніше.)

Декартів добуток (рос. «*декартово произведение*», англ. «*Cartesian product*») множин A та B — це множина всіх можливих упорядкованих пар, де перший елемент пари береться з множи́н A, дру́гий — з множи́н B:

$$A \times B \stackrel{\text{def}}{=} \{(a,b) \mid a \in A \wedge b \in B\}. \tag{10}$$

(Наприклад, $\{a,b,c\} \times \{1,2\} = \{(a,1),\ (a,2),\ (b,1),\ (b,2),\ (c,1),\ (c,2)\}$. Декартів добуток є множиною, отже пари можна переставляти (наприклад, $\{(b,1),\ (a,1),\ (c,2),\ (a,2),\ (b,2),\ (c,1)\}$ — некрасивий запис того са́мого добутку). Але переставляти елементи *всере́дині* пари *не можна* (наприклад, $(2,c) \notin \{a,b,c\} \times \times \{1,2\}$, а 2-ге місце в 19-му ряду — *не* те са́ме, що 19-те місце у 2-му ряду).

І це означає *некомутативність* декартового добутку: $A \times B \neq B \times A$.)

Декартів добуток очевидно узагальнюється на випадок кількох (більш ніж двох) множин. *Декартів до́буток множин* A_1, A_2, ..., A_n — це множина всіх можливих упорядкованих n-ок[19], де 1-ий елемент n-ки береться з множи́н A_1, 2-ий — з множи́н A_2, ..., n-ий — з множи́н A_n:

$$A_1 \times A_2 \times \ldots \times A_n \stackrel{\text{def}}{=} \{(a_1, a_2, \ldots, a_n) \mid a_1 \in A_1 \wedge a_2 \in A_2 \wedge \ldots \wedge a_n \in A_n\}. \tag{11}$$

Приклад: $\{a,b\} \times \{2,5\} \times \{a,o\} = \{(a,2,a),\ (a,2,o),\ (a,5,a),\ (a,5,o),\ (b,2,a),\ (b,2,o),\ (b,5,a),\ (b,5,o)\}$.

n-им декартовим степенем множи́н A називають частковий випадок (узагальненого) декартового добутку, коли $A_1 = A_2 = \ldots = A_n$:

$$A^n \stackrel{\text{def}}{=} \underbrace{A \times A \times \ldots \times A}_{n \text{ множників}} \tag{12}$$

(У деяких книгах стверджують, буцімто, як і для переважної більшості різноманітних «добутків» дискретної математики,[20] ніякої окремої операції «узагальнений декартів добуток» не треба, а степінь можна ввести як $A^1 = A$, $A^n = \underbrace{(\underbrace{\ldots (A \times A) \times \ldots \times A}_{A^{n-1}\ (n-1 \text{ штук } A)}) \times A}_{n \text{ штук } A}$.

Автор посібника підтримує іншу поширену думку, за якою елементами $A \times B \times C$ та $(A \times B) \times C$ є просто різні сутності: $A \times B \times C$ складається з трійок (a_i, b_j, c_k), тоді як $(A \times B) \times C$ — з $((a_i, b_j), c_k)$, тобто пар, першими елементами яких є пари. При такому підході, декартів добуток не лише не комутативний, але й не асоціативний ($(A \times B) \times C \neq A \times (B \times C)$, бо в одному випадку утворюються послідовності вигляду $((a_i, b_j), c_k)$, в іншому — вигляду $(a_i, (b_j, c_k))$); через це, пропонується уникати послідовних декартових домножень, натомість виражаючи як степінь, так і інші поєднання зразу кількох елементів у єдину послідовність однією операцією узагальненого декартового добутку.)

---

[18] він же де́ Карте́с, тому кажуть також «картезіа́нський»; за радянською традицією більш прийнято писати «декартів», за західною — «Cartesian»

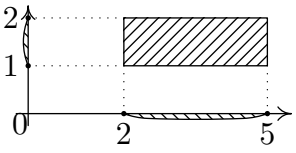
[19] читається «е́нок»; «е́нка» означає просто «послідовність з n елементів»; англ. — «*n-tuple*»

[20] у цьому посібнику, крім декартового добутку, розглянуто ще деякі операції, які відносять до «добутків»: кон'юнкцію (розд. 1.1), композицію відношень (розд. 2.5.4) та конкатенацію мов (розд. 6.2), а також використано відомий з алгебри добуток матриць; все це — ще не повний перелік «добутків», бо дискретна математика цим посібником не обмежується

Оскільки декартів добуток є множиною, до декартових добутків застосовні множинні операції (“$\cup$”, “$\cap$”, “$\setminus$”, “$\div$”) та відношення (“$\in$”, “$\subseteq$”, “$\subset$”).

Іноді (не часто) буває зручно зображати декартів добуток графічно (звичайно, *лише двоаргументний добуток!*). Для цього треба множи́ни, котрі є лівими аргументами добутку, зобразити вздовж однієї осі, праві аргументи — вздовж іншої осі. Зображенням декартового добутку буде прямокутник або сукупність прямокутників.

Наприклад, зобразимо добуток $[2;5]\times[1;2]$ (де $[2;5]$ та $[1;2]$ — відрізки всіх дійсних чисел від … до …, обидві межі́ включно):



Доведення тотожностей і включень У означенні декартового добутку заданий характеристичний предикат. Значить, можна переходити до логічних виразів від простіших характеристичних предикатів, досліджувати їх за допомогою таблиць істинності, тощо.

Приклад 1. Доведемо $(A\times B)\cap(C\times D)=(A\cap C)\times(B\cap D)$.

Ліва частина: $(A\times B)\cap(C\times D)=\{(x,y)\mid(x,y)\in(A\times B)\wedge(x,y)\in(C\times D)\}=\{(x,y)\mid\big(x\in A\wedge y\in B\big)\wedge\big(x\in C\wedge y\in D\big)\}$.

Права частина: $(A\cap C)\times(B\cap D)=\{(x,y)\mid x\in(A\cap C)\wedge y\in(B\cap D)\}=\{(x,y)\mid\big(x\in A\wedge x\in C\big)\wedge\big(y\in B\wedge y\in D\big)\}$.

Можна побудувати таблиці істинності (на $2^4=16$ рядків). А можна побачити, що у виразах використовується лише “$\wedge$”, та довести тотожню рівність цих виразів згідно з асоціативністю та комутативністю кон’юнкції.

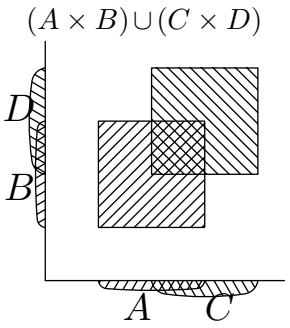
Приклад 2. Доведемо, що $(A\times B)\cup(C\times D)\subseteq(A\cup C)\times(B\cup D)$, причому ці вирази *не* тотожньо рівні. Ліва частина: $(x,y)\in(A\times B)\cup(C\times D)=\big((x,y)\in(A\times B)\big)\vee\big((x,y)\in(C\times D)\big)=\big((x\in A)\wedge(y\in B)\big)\vee\big((x\in C)\wedge(y\in D)\big)$; права частина: $(x,y)\in(A\cup C)\times(B\cup D)=(x\in(A\cup C))\wedge(y\in(B\cup D))=\big((x\in A)\vee(x\in C)\big)\wedge\big((y\in B)\vee(y\in D)\big)$; “$\subseteq$”, як завжди, перетворюється у “$\to$” (з подальшим аналізом, чи всюди ця імпілікація істинна).

$$\underbrace{\underbrace{\big(\underbrace{(x\in A)\wedge(y\in B)}_{(1)}\big)\vee\big(\underbrace{(x\in C)\wedge(y\in D)}_{(2)}\big)}_{(3)}\to\underbrace{\big(\underbrace{(x\in A)\vee(x\in C)}_{(4)}\big)\wedge\big(\underbrace{(y\in B)\vee(y\in D)}_{(5)}\big)}_{(6)}}$$

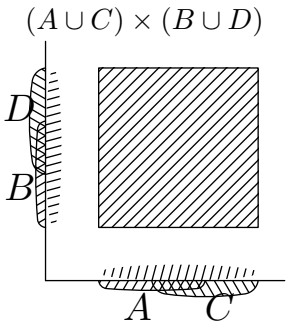
$x\in A$	$y\in B$	$x\in C$	$y\in D$	(1)	(2)	(3)	(4)	(5)	(6)	(3)→(6)
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0	1
0	0	1	0	0	0	0	1	0	0	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0	1	0	1
0	1	0	1	0	0	0	0	1	0	1
0	1	1	0	0	0	0	1	1	1	1
0	1	1	1	0	1	1	1	1	1	1
1	0	0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	1	1	1	1
1	0	1	0	0	0	0	1	0	0	1
1	0	1	1	0	1	1	1	1	1	1
1	1	0	0	1	0	1	1	1	1	1
1	1	0	1	1	0	1	1	1	1	1
1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1

Як бачимо, імплікація істинна завжди, але стовпчики (3) і (6) не однакові. Значить, для виразів справді виконується “$\subseteq$”, але не “$=$”.

Покажемо це са́ме графічно. A і C є лівими аргументами декартового добутку, B і D — правими. Тож зобразимо A і C відрізками вздовж горизонтальної осі, B і D — уздовж вертикальної.



Будуємо $A \times B$ («права» штриховка) та $C \times D$ («ліва»). Їх потрібно об'єднати; отже, результат — усе, що заштриховане (хоча б якось).



Спочатку потрібно провести об'єднання $A \cup C$ (результат на горизонтальній осі) та $B \cup D$ (на вертикальній), потім будуємо добуток.

Бачимо, що вирази не рівні. Більш того, перший вираз є підмножиною дру́гого. (Останнє твердження аргументоване, бо на діаграмі справді є області, відповідні усім 16 випадкам.)

Тепер стає ще важливішим, ніж досі, що логічні операції застосовують ***до базових предикатів, а не до множин****.* Розглянемо (неправильне) буцімто доведення: *«Доведемо, що* $(A \times B) \cap (C \times D) = (A \cap B) \times (C \cap D)$*. Позначимо твердження, пов'язані з множиною A, як "a", пов'язані з B — як "b", з C — як "c", з D — як "d". І декартів добуток, і перетин виражаються через кон'юнкцію, тому обидві частини перетворюються до однакових виразів "*$(a \wedge b) \wedge (c \wedge d)$*", що й доводить тотожність.»*. Протиставимо йому приклад: $A=\{1,2\}$, $B=\{2,3\}$, $C=\{3,4\}$, $D=\{4,5\}$.

$$\underbrace{(A \times B)}_{\{(1,2),(1,3),(2,2),(2,3)\}} \cap \underbrace{(C \times D)}_{\{(3,4),(3,5),(4,4),(4,5)\}} = \varnothing, \qquad \underbrace{(A \cap B)}_{\{2\}} \times \underbrace{(C \cap D)}_{\{4\}} = \{(2,4)\}.$$

Твердження неправильне! Воно не може мати правильного доведення!!!

Конкретна помилка цього «доведення» полягає в тому, що, зокрема, "$x \in B$" і "$y \in B$" — *різні* предикати, і їх не можна позначати однаковим b. Правильні перетворення $(A \times B) \cap (C \times D) = (A \cap B) \times (C \cap D)$ дають вираз $\big((x \in A \wedge y \in B) \wedge (x \in C \wedge y \in D)\big) \leftrightarrow \big((x \in A \wedge x \in B) \wedge (y \in C \wedge y \in D)\big)$, залежний від *шести* різних базових предикатів "$x \in A$", "$x \in B$", "$y \in B$", "$x \in C$", "$y \in C$", "$y \in D$", і серед $2^6 = 64$ рядків його таблиці істинності є 7 штук нулів.

Відзначимо (довести охочі можуть самостійно), що декартів добуток дистрибутивний відносно стандартних бінарних множинних операцій:

$$\begin{aligned} A \times (B \cap C) &= (A \times B) \cap (A \times C), & (A \cap B) \times C &= (A \times C) \cap (B \times C), \\ A \times (B \cup C) &= (A \times B) \cup (A \times C), & (A \cup B) \times C &= (A \times C) \cup (B \times C), \\ A \times (B \setminus C) &= (A \times B) \setminus (A \times C), & (A \setminus B) \times C &= (A \times C) \setminus (B \times C), \\ A \times (B \div C) &= (A \times B) \div (A \times C), & (A \div B) \times C &= (A \times C) \div (B \times C). \end{aligned}$$

(У кожному рядку записані два варіанти дистрибутивності *однієї й тієї ж* пари операцій. Досі (для інших операцій) ми не потребува́ли окремих «лівої» та «правої» дистрибутивності, бо одну з них можна було одержати з іншої через комутативність. А тепер, коли "$\times$" не комутативний, ці тотожності стають «незалежними».)

2.5 Відношення

2.5.1 Загальне означення відношення

Відношенням (рос. «*отношение*», англ. «*relation*») між множинами A_1, A_2, …, A_n називають підмножину декартового добутку $A_1 \times A_2 \times \ldots \times A_n$.

Але це означення *аж надто* загальне. Томý наведемо деякі пояснення.

Відношення вказують факт наявності зв'язкі́в між елементами множин. Якщо пара (енка) належить відношенню, це означає, що між елементами цієї пари (енки) є зв'язок, якщо не належить — зв'язку нема.

Відношення задають те саме, що й предикати, але в іншому форматі. Приклади предикатів («n — просте число», «$x \perp y$», «z лежить на найкращому шляху з x до y», тощо) можна вважати також і прикладами відношень. Але предикатами були умови, а відношеннями є множи́ни усіх тих (чисел / пар прямих / трійок міст / …), для яких умова виконується.

Наприклад, у розділі 1.6.2 «Способи подання предикатів» згадувався предикат «тролейбусом x можна доїхати до y». При цьому «перелік наборів, на яких предикат істинний» був таким: (1, драмтеатр), (1, пл. Б. Хм.), (1, ЧНУ), (4, драмтеатр), (4, трол. парк), (7, драмтеатр), (7, пл. Б. Хм.), (7, трол. парк), (7, ЧНУ), (10, драмтеатр), (10, ПЗР), (10, пл. Б. Хм.). А це якраз і є *підмножиною* декартового добутку $\{1,4,7,10\} \times \{$драмтеатр, ПЗР, пл. Б. Хм., трол. парк, ЧНУ$\}$: перелічені пари туди належать, а, наприклад, (1, ПЗР) — не належить. Отже, перелічені пари можна вважати також і відношенням. Предикат має значення `true` — пара належить відношенню, значення `false` — не належить.

(Це можна і формально узагальнити. Нехай є предикат $P(a_{(1)}, \ldots, a_{(n)})$, причому значення параметра $a_{(1)}$ можуть братися з множин S_1, …, параметра $a_{(n)}$ — з множин S_n. Тоді за цим предикатом можна побудувати відношення — підмножину $S_1 \times \ldots \times S_n$:

$$R = \{(x_1, \ldots, x_n) \mid x_1 \in S_1 \wedge \ldots \wedge x_n \in S_n \wedge P(x_1, \ldots, x_n)\}.$$

Аналогічно, коли є відношення $R \subseteq S_1 \times \ldots \times S_n$, за ним (по ньому) завжди можна побудувати предикат

$$P(x_1, \ldots, x_n) = (x_1, \ldots, x_n) \in R = \begin{cases} \texttt{true}, & (x_1, \ldots, x_n) \in R, \\ \texttt{false}, & (x_1, \ldots, x_n) \notin R, \end{cases}$$

залежний від n параметрів, де 1-ий параметр може братися з множин S_1, …, n-ий параметр — з S_n.)

(“R” — абстрактне позначення (якогось) відношення, аналогічно як букву “f” використовують для абстрактного позначення (якоїсь) функції. Відношення “R”, як правило, *не має* ніякого стосунку до множин дійсних чисел “$\mathbb{R}$”.)

Відношення у загальному вигляді зазвичай детально розглядають у дисципліні «Бази даних». Ми ж перейдемо до розгляду часткового випадку — т. зв. *бінарних відношень*.

2.5.2 Бінарні відношення. Бінарність «у вузькому» та «у широкому» смислах

Природньо, щоб *бінарним відношенням* (рос. «*бинарное отношение*», англ. «*binary relation*») називали двоаргументне відношення, тобто підмножину декартового добутку 2-х множин $A \times B$.

Але часто розглядають відношення, *бінарні у вузькому смислі* — підмножини *декартового квадрата* деякої множини. Відповідно, відношення-підмножини $A \times B$ (де множини A і B можуть, але не зобов’язані дорівнювати одна одній) бінарні «у широкому смислі». Таким чином, кожне відношення, бінарне «у вузькому смислі», буде також і бінарним «у широкому»; але далеко не кожне бінарне «у широкому» буде бінарним «у вузькому».

(Наприклад, обидва відношення «пряма x перпендикулярна прямій y» та «тролейбусом x можна доїхати до y» бінарні «у широкому смислі», але лише перше з них бінарне «у вузькому смислі». Адже у першому відношенні x і y беруться з однієї множини — множини прямих у площині, а у другому x береться з множини тролейбусних маршрутів, y — з множини визначних місць.

Бінарне «у широкому смислі» відношення (підмножину $A \times B$) називають також «*відношення в $A \times B$*» (рос. «*отношение в $A \times B$*», англ. «*relation between A and B*»); бінарне «у вузькому смислі» відношення (підмножину $A^2 = A \times A$) — «*відношення на A*» (рос. «*отношение на A*», англ. «*relation on A*»).)

Прикладами бінарних «у вузькому смислі» відношень є множини істинності більшості математичних «порівнянь» (“$>$”, “$\neq$”, “$\subseteq$”, “$\vdots$”, “$\perp$”, тощо).

Щоб вказати, що для елементів x і y виконується бінарне відношення R, використовуються дві форми запису: “$(x, y) \in R$” (як для всіх відношень) та “$x\,R\,y$” (символ бінарного відношення “R” ставиться між аргументами, подібно до “$x < y$”; така форма запису називається *інфіксна*).

2.5.3 Подання бінарного відношення матрицею

Коли бінарне відношення скінченне й невелике (як за кількістю пар, так і за кількістю елементів тих множин, в добутку яких воно задане), буває зручно подавати його матрицею. Рядки такої матриці відповідають можливим елементам першої множини відношення, стовпчики — можливим елементам другої, і на перетині рядка та стовпчика ставиться або 1, якщо пара відповідних елементів належить відношенню, або 0, якщо не належить.

Якщо відношення бінарне «у широкому смислі» (в $A \times B$, де A та B можуть бути різні), матриця прямокутна (можливо, квадратна, але не обов’язково). Якщо бінарне «у вузькому смислі» (на A), то гарантовано квадратна. При цьому, навіть для квадратних матриць, що подають бінарні «у вузькому смислі» відношення, важливо, що рядки відповідають першим елементам відношення, стовпчики — другим (а не навпаки).

Матриця бінарного відношення виявляється такою са́мою, як табличне подання предиката від двох параметрів (див. розд. 1.6.2), так що наведений на стор. 30 приклад поданого таблицею «тролейбусного» предиката є також і прикладом матриці відношення.

2.5.4 Операції над бінарними відношеннями

Бінарні відношення є множинами, тому для них визначені операції "$\cup$", "$\cap$", "$\setminus$", "$\div$" та "$'$" (універсумом вважається $A \times B$ чи A^2, для «широкого» та «вузького» смислів відповідно).

Але для бінарних відношень визначені ще деякі специфічні операції.

Область визначення (рос. «*область определения*», англ. «*domain*»; позначається "$\mathrm{dom}R$", або "$D(R)$", іноді "R_A") бінарного відношення — це множина усіх можливих перших аргументів відношення:

$$D(R) \stackrel{\text{def}}{=} \{a \mid \exists b \;\; a\, R\, b\}. \tag{13}$$

(Наприклад, якщо $R = \{(1,a),\ (1,b),\ (2,b),\ (4,z)\}$, то $D(R) = \{1,2,4\}$.)

Область значень (рос. «*область значений*», англ. «*codomain*»; позначається "$E(R)$", іноді "R_B") бінарного відношення — це множина усіх можливих дру́гих аргументів відношення:

$$E(R) \stackrel{\text{def}}{=} \{b \mid \exists a \;\; a\, R\, b\}. \tag{14}$$

(Наприклад, якщо $R = \{(1,a),\ (1,b),\ (2,b),\ (4,z)\}$, то $E(R) = \{a,b,z\}$.)

Відношення, *обернене* до (рос. «*обратное* к», англ. «*inverse* of») відношення R в $A \times B$ — це відношення в $B \times A$, якому належать усі ті самі пари, але у зворотньому порядку:

$$R^{-1} \stackrel{\text{def}}{=} \{(b,a) \mid a\, R\, b\}. \tag{15}$$

(Наприклад, якщо $R = \{(1,a),(1,b),(2,b),(4,z)\}$, то $R^{-1} = \{(a,1),(b,1),(b,2),(z,4)\}$.)

Очевидно, $(R^{-1})^{-1} = R$.

Композиція (вона ж *суперпозиція*, вона ж *добуто́к*; рос. «*композиция*»; англ. «*composition*») відношень R_1 та R_2 — це відношення

$$R_1 \circ R_2 \stackrel{\text{def}}{=} \{(a,c) \mid \exists b (a\, R_1\, b \wedge b\, R_2\, c)\}, \tag{16}$$

тобто пара (a,c) перебуває у відношенні $R_1 \circ R_2$ тоді й тільки тоді, коли можна знайти такий елемент b, щоб a перебувало у відношенні R_1 з цим b, і (те са́ме) b перебувало у відношенні R_2 з елементом c. При цьому R_1 та R_2 мають бути визначені в $A \times B$ та у $B \times C$ (дру́га множина першого відношення має дорівнювати першій множині дру́гого відношення; решта множин можуть бути хоч різними, хоч однаковими).

Приклад 1. Нехай $R_1 = \{(a,1),\ (b,1),\ (b,2),\ (z,4)\}$, $R_2 = \{(1,\beta),\ (2,\beta),\ (2,\gamma)\}$. Тоді $R_1 \circ R_2 = \{(a,\beta),\ (b,\beta),\ (b,\gamma)\}$. Пара (a,β) належить $R_1 \circ R_2$, бо є елемент 1, такий що $(a,1) \in R_1 \wedge (1,\beta) \in R_2$; аналогічно, $(b,\gamma) \in R_1 \circ R_2$, бо $(b,2) \in R_1 \wedge (2,\gamma) \in R_2$; для (b,β) роль «проміжного» елемента може відіграти хоч 1, хоч 2. Інші пари (такі, як (a,γ), (z,β), тощо) не належать $R_1 \circ R_2$, бо для них неможливо підібрати спільний «проміжний» елемент.

Приклад 2. Нехай $x\, R_1\, y$ означає «x — брат y», $y\, R_2\, z$ означає «y — батько z». Тоді $x\, R_1 \circ R_2\, z$ означає «x — дядько (батьків брат) z». А відношення «x — дядько z (включаючи дядьків і по матері, і по батьку)» можна виразити як $R_1 \circ (R_2 \cup R_3)$ (де $y\, R_3\, z$ означає «y — мати z»).

Очевидно, композиція не комутативна, тобто $R_1 \circ R_2$ може не дорівнювати $R_2 \circ R_1$ (і навіть один з виразів може мати смисл, а інший не мати).

Але композиція асоціативна, тобто коли один з виразів $(R_1 \circ R_2) \circ R_3$, $R_1 \circ (R_2 \circ R_3)$ має смисл, то інший теж має смисл і задає те са́ме відношення.

Доведення. Безпосередньо з означень,

$$(R_1 \circ R_2) \circ R_3 = \{(a,d) \mid \exists c (\exists b (a\, R_1\, b \wedge b\, R_2\, c) \wedge c\, R_3\, d)\},$$
$$R_1 \circ (R_2 \circ R_3) = \{(a,d) \mid \exists b (a\, R_1\, b \wedge \exists c (b\, R_2\, c \wedge c\, R_3\, d))\};$$

ці характеристичні предикати рівні, тому що: 1) квантор можна переносити через підпредикат, що не містить вільних входжень відповідної змінної; 2) однотипні квантори можна переставляти місцями; 3) кон'юнкція асоціативна. ■

(Не комутативність, але асоціативність — та са́ма ситуація, яка має місце і для добутку матриць. Виявляється, цей збіг не випадковий, а обумовлений глибшим зв'язком. Як вже пояснено у розд. 2.5.3, рядки́ матриці відношення відповідають першій (лівій) множині́ відношення, стовпчики — дру́гій (правій). Таким чином, вимога, що дру́га множина першого відношення має дорівнювати першій множині дру́гого відношення, має тісний зв'язок з вимогою, що кількість стовпчиків першого матричного множника має дорівнювати кількості рядків дру́гого.

Запишемо означення композиції відношень, але замість квантора "$\exists b$" поставимо диз'юнкцію по всім можливим значенням b (що, очевидно, має той са́мий смисл). Паралельно запишемо стандартне означення добутку матриць. Вийде

Композиція відношень $R_1 \circ R_2$	Добуток матриць $C = A \times B$
$a_i\,(R_1 \circ R_2)\,c_j \overset{\text{def}}{\Longleftrightarrow} \bigvee\limits_k (a_i\,R_1\,b_k \wedge b_k\,R_2\,c_j)$	$c_{ij} = \sum\limits_k (a_{ik} \cdot b_{kj})$

При стандартному (розд. 2.5.3) переході від відношення до його матриці, (a_i, b_k) відповідає a_{ik}, (b_k, c_j) відповідає b_{kj}, (a_i, c_j) відповідає c_{ij}; елементи матриці можуть набувати лише значень 0 та 1, тому́ операції множення та кон'юнкція в певному смислі однакові: і та, і та дають ненульовий результат тільки коли обидва аргументи — одиниці. Отже, *єдина* відмінність між добутком матриць відношень та матрицею композиції відношень — це відмінність між додаванням та диз'юнкцією (додавання враховує кількість одиничних добутків, диз'юнкція не враховує кількість одиничних кон'юнкцій).)

Сформулюємо і доведемо кілька основних тотожностей, що стосуються операцій над відношеннями.

$$(R_1 \cup R_2)^{-1} = {R_1}^{-1} \cup {R_2}^{-1}, \tag{17}$$

$$(R_1 \cap R_2)^{-1} = {R_1}^{-1} \cap {R_2}^{-1}, \tag{18}$$

$$(R')^{-1} = (R^{-1})'. \tag{19}$$

Доведення. Для (17): $(a,b) \in (R_1 \cup R_2)^{-1} = (b,a) \in (R_1 \cup R_2) = \big((b,a) \in R_1\big) \vee \big((b,a) \in R_2\big) = \big((a,b) \in {R_1}^{-1}\big) \vee \big((a,b) \in {R_2}^{-1}\big) = (a,b) \in {R_1}^{-1} \cup {R_2}^{-1}$; для (18) аналогічно. Для (19): $(a,b) \in (R')^{-1} = (b,a) \in R' = (b,a) \notin R = (a,b) \notin R^{-1} = (a,b) \in (R^{-1})'$. ■

$$(R_1 \circ R_2)^{-1} = R_2^{-1} \circ R_1^{-1} \tag{20}$$

(коли один з цих виразів має смисл, інший теж має смисл і задає те са́ме відношення).

Доведення. $(a,c) \in (R_1 \circ R_2)^{-1} = (c,a) \in R_1 \circ R_2 = \exists b (c\,R_1\,b \wedge b\,R_2\,a) = \exists b (b\,{R_1}^{-1}\,c \wedge a\,{R_2}^{-1}\,b) = \exists b (a\,{R_2}^{-1}\,b \wedge b\,{R_1}^{-1}\,c) = (a,c) \in R_2^{-1} \circ R_1^{-1}$. ■

$$D(R^{-1}) = E(R), \qquad E(R^{-1}) = D(R); \tag{21}$$

$$D(R_1 \circ R_2) \subseteq D(R_1), \qquad E(R_1 \circ R_2) \subseteq E(R_2); \tag{22}$$

$$\text{якщо } E(R_1){=}D(R_2), \text{ то} \quad \begin{cases} D(R_1 \circ R_2) = D(R_1), \\ E(R_1 \circ R_2) = E(R_2); \end{cases} \tag{23}$$

(Без доведень; див. також завд. 7* на стор. 79.)

Все вищесказане стосувалося усіх бінарних («у широкому смислі») відношень. Для відношень, бінарних «у вузькому смислі» (*тільки* для них), можливо ввести ще два означення.

Тотожне відношення, воно ж *відношення рівності* на множині A — це відношення, якому належать усі пари однакових елементів і лише вони:

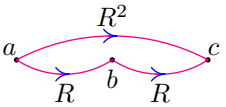
$$I_A \overset{\text{def}}{=} \{(a,a) \mid a \in A\}. \tag{24}$$

(Тобто, $a\,I_A\,b$ по суті те саме, що $a = b$; просто іноді зручно скористатися са́ме способом запису "I_A". Перекладі: рос. «*тождественное отношение*», «*отношение равенства*»; англ. «*equality relation*» (не плутати з «equivalence relation» з розд. 2.6.2).)

Степінь відношення — це відповідну кількість разів використана композиція відношення з самим собою:

$$R^n \overset{\text{def}}{=} \underbrace{R \circ \ldots \circ R}_{\substack{n \text{ відношень,} \\ n-1 \text{ композицій}}} \quad ; \quad R^1 = R; \quad R^0 \overset{\text{def}}{=} I_A \text{ (де } I_A \text{ — тотожне відношення).} \tag{25}$$

Згідно цього означення та асоціативності композиції, степінь бінарного відношення має властивості $R^{n+m} = R^n \circ R^m$ та $R^{n \cdot m} = (R^n)^m$ (при $n \in \mathbb{Z}^+ \wedge m \in \mathbb{Z}^+$). Але поширювати ці тотожності на усі цілі чи́сла (вважаючи, ніби обернене відношення R^{-1} є (-1)-им степенем) *не можна*.



Відношенню R^2 належать пари елементів, для яких відношення R «виконується через ланцюжок, що містить один проміжний елемент і два переходи».

Аналогічно, відношенню R^n (при $n \geqslant 2$) належать такі пари елементів, для яких відношення R «виконується через ланцюжок, що містить $n-1$ проміжних елементів і n переходів». «Ланцюжок» може багатократно містити одні й ті ж елементи; але сумарна кількість (враховуючи кратність) переходів повинна становити *в точності* n.)

2.6 Важливі класи бінарних «у вузькому смислі» відношень

2.6.1 Рефлексивність, іррефлексивність, симетричність, антисиметричність, транзитивність, повнота (класичні означення)

Для бінарних «у вузькому смислі» відношень вводять важливі класи, або спеціальні властивості. Це *не* властивості, яким обов'язково задовольняє кожне бінарне відношення. Навпаки: для кожної з них (властивостей) є одні бінарні відношення, які мають цю властивість, і є інші бінарні відношення, для яких вона не виконується.

Бінарне відношення *рефлексивне* (рос. «*рефлексивное*», англ. «*reflexive*»), коли кожен елемент перебуває у (цьому) відношенні сам з собою.

$$R \text{ рефлексивне} \quad \stackrel{\text{def}}{\iff} \quad \forall a\ (a\,R\,a). \tag{26}$$

Бінарне відношення *іррефлексивне* (рос. «*иррефлексивное*», англ. «*irreflexive*»), коли жоден елемент не перебуває у (цьому) відношенні сам з собою.

$$R \text{ іррефлексивне} \quad \stackrel{\text{def}}{\iff} \quad \forall a(\neg(a\,R\,a)). \tag{27}$$

Бінарне відношення *симетричне* (рос. «*симметричное*», англ. «*symmetric*»), коли з того, що пара елементів перебуває у (цьому) відношенні, завжди випливає, що пара, де ці елементи переставлені місцями, теж перебуває у (цьому) відношенні:

$$R \text{ симетричне} \quad \stackrel{\text{def}}{\iff} \quad \forall a \forall b(a\,R\,b \to b\,R\,a). \tag{28}$$

Бінарне відношення *антисиметричне* (рос.«*антисимметричное*», англ. «*antisymmetric*»), коли з того, що елементи перебувають у (цьому) відношенні в обох можливих порядках, завжди випливає, що ці елементи рівні:

$$R \text{ антисиметричне} \quad \stackrel{\text{def}}{\iff} \quad \forall a \forall b(a\,R\,b \wedge b\,R\,a \to a = b). \tag{29}$$

Якщо «перевернути» імплікацію згідно контрапозиції, стає трохи зрозумілішим, в чому ж тут полягає протилежність до симетричності: бінарне відношення *антисиметричне*, коли заборонено, щоб різні елементи задовольняли (цьому) відношенню в обох можливих порядках:

$$R \text{ антисиметричне} \quad \stackrel{\text{def}}{\iff} \quad \forall a \forall b(a \neq b \to \neg(a\,R\,b \wedge b\,R\,a)).$$

Але у більшості випадків перше формулювання зручніше для доведень.

Бінарне відношення *транзитивне* (рос. «*транзитивное*», англ. «*transitive*»), коли з того, що елемент перебуває у (цьому) відношенні з яким-небудь іншим, а той інший — з яким-небудь ще іншим, завжди випливає, що перший згаданий елемент перебуває у відношенні з третім згаданим:

$$R \text{ транзитивне} \quad \stackrel{\text{def}}{\iff} \quad \forall a \forall b \forall c(a\,R\,b \wedge b\,R\,c \to a\,R\,c). \tag{30}$$

Бінарне відношення *повне* (або, що те са́ме, *лінійне*; рос. «*полное*», «*линейное*»; англ. «*total*», «*linear*»), коли будь-яка пара різних елементів перебуває у (цьому) відношенні — хоча б у одному з двох можливих порядків:

$$R \text{ повне} \quad \stackrel{\text{def}}{\iff} \quad \forall a \forall b_{b \neq a}(a\,R\,b \vee b\,R\,a). \tag{31}$$

Наведемо приклади. Скрізь, де зручно, будемо наводити приклади трьох типів: 1) відношення, основані на математичних порівняннях; 2) відношення, задані осмисленими, але не вузько математичними, фразами; 3) відношення, подані матрицею (розд. 2.5.3).

Означення рефлексивності та іррефлексивності очевидно суперечать одне́ о́дному, тож не буває відношень, одночасно рефлексивних та іррефлексивних. Але бувають ні рефлексивні, ні іррефлексивні відношення.

Рефл.	Іррефл.	Матем. порівняння	Приклади-фрази	Приклади-матриці
+	+	— *не буває* —		
+	−	$x = y$ (числова рівність) $x \leqslant y$ (числове) $X \subseteq Y$ (включення множин) $x \vdots y$ (кратність $\mathbb{N}$ чисел)	людина x знає ім'я людини y (вважаючи, що будь-хто знає своє ім'я)	$\begin{array}{c\|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & 1 & 1 & 1 & 0 \\ a_2 & 0 & 1 & 0 & 0 \\ a_3 & 0 & 1 & 1 & 1 \\ a_4 & 0 & 1 & 0 & 1 \end{array}$

Рефл.	Ірефл.	Матем. порівняння	Приклади-фрази	Приклади-матриці
−	+	$x \neq y$ (числова нерівність) $x < y$ (числове менше) $x \perp y$ (перпендикулярність прямих)	від міста x до міста y більше 500 км	$\begin{array}{c\|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & 0 & 1 & 1 & 0 \\ a_2 & 0 & 0 & 0 & 1 \\ a_3 & 0 & 1 & 0 & 0 \\ a_4 & 1 & 1 & 0 & 0 \end{array}$
−	−	$y = \lvert x\rvert$ (на $\mathbb{R}$) (див. далі)	людина x знає номер телефона людини y (враховуючи, що у деяких людей нема телефона)	$\begin{array}{c\|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & 0 & 1 & 1 & 0 \\ a_2 & 0 & 1 & 0 & 1 \\ a_3 & 1 & 1 & 0 & 0 \\ a_4 & 1 & 0 & 0 & 1 \end{array}$

Перейдемо до симетричності та антисиметричності. Природньо, коли одна з них виконується, а інша — ні; але взагалі-то можливі всі 4 комбінації.

Сим.	Антисим.	Матем. порівняння	Приклади-фрази	Приклади-матриці
+	−	$x \neq y$ (числова нерівність) $x \perp y$ (перпендикулярність) $x \parallel y$ (паралельність) $x \cdot y > 0$	студенти x та y сидять за однією партою	$\begin{array}{c\|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & 1 & 1 & 1 & 0 \\ a_2 & 1 & 1 & 0 & 0 \\ a_3 & 1 & 0 & 1 & 1 \\ a_4 & 0 & 0 & 1 & 1 \end{array}$
−	+	$x < y$ (числове менше) $X \subseteq Y$ (включення множин) $x \vdots y$ (кратність $\mathbb{N}$ чисел)	людина x значно старша за людину y	$\begin{array}{c\|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & 1 & 1 & 1 & 0 \\ a_2 & 0 & 0 & 0 & 0 \\ a_3 & 0 & 0 & 0 & 0 \\ a_4 & 1 & 0 & 0 & 1 \end{array}$
−	−	$x \vdots y$ (кратність $\mathbb{Z}$ чисел). не симетричне, бо $6 \vdots 3 \nrightarrow 3 \vdots 6$, не антисиметричне, бо $(2 \vdots (-2)) \wedge ((-2) \vdots 2) \wedge (2 \neq -2)$.		$\begin{array}{c\|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & 0 & 1 & 1 & 1 \\ a_2 & 1 & 1 & 0 & 0 \\ a_3 & 0 & 0 & 0 & 0 \\ a_4 & 0 & 0 & 0 & 0 \end{array}$
+	+	$x = y$ (числова рівність)		$\begin{array}{c\|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & 0 & 0 & 0 & 0 \\ a_2 & 0 & 1 & 0 & 0 \\ a_3 & 0 & 0 & 1 & 0 \\ a_4 & 0 & 0 & 0 & 1 \end{array}$

Перейдемо до транзитивності.

Транзитивність виконується для відношень: $x = y$ (числова рівність); $x < y$ (числове строго менше); $x \leqslant y$ (числове менше-рівне); $x \subseteq y$ (включення множин); $x \vdots y$ (кратність, причому байдуже, чи на $\mathbb{N}$, чи на $\mathbb{Z}$); паралельність прямих, але вважаючи, що кожна пряма паралельна сама собі; «людина x значно старша за людину y».

Транзитивність не виконується для відношень $x \neq y$ (числова нерівність), $x \perp y$ (перпендикулярність прямих); «людина x добре знає людину y», «від міста x до міста y менше 500 км».

(Прикладів матриць транзитивних та нетранзитивних відношень не наводимо, бо досліджувати транзитивність на матрицях незручно. Хоча можна: перебирати всі можливі можливі трійки (їх скінче́нна кількість, бо сама́ матриця скінче́нна), й для кожної трійки дивитися, чи утворює вона контрприклад (див. розд. 1.6.3), чи ні.

Власне, решта властивостей теж перевіряються на матрицях таким само пошуком контрикладів; мова про те, що такі умови, як «чи всі елементи головної діагоналі .....» та «чи симетрична матриця» формулювати й перевіряти більш-менш зручно, а з транзитивністю це технічно (але не ідейно!) складніше. Див. також приклад аналізу відношення, заданого матрицею, на стор. 64.)

Щодо повноти (лінійності) краще навести один-єдиний приклад: порівняти числове менше-рівне ($x \leqslant y$) та включення множин ($X \subseteq Y$). Обидва ці відношення рефлексивні, не ірефлексивні, не симетричні, антисиметричні та транзитивні. *Єдина* з точки зору стандартної класифікації бінарних відношень відмінність — що відношення "$x \leqslant y$" повне, бо для будь-яких двох чисел одне з них буде менше або рівне за інше, тоді як "$X \subseteq Y$" неповне, бо можна підібрати такі множи́ни (наприклад, $\{a, b, c, d\}$ та $\{b, p\}$), що жодна з них не входить до іншої.

У більшості вищезгаданих прикладів виконання/не виконання властивостей досить очевидне. Але можливі й відношення, для яких така перевірка значно складніша. Тож наведемо також приклад детального, з усіма доведеннями, дослідження відношення на виконання/не виконання цих властивостей. А саме, дослідимо відношення $R \in \mathbb{R}^2$: $xRy \Leftrightarrow y = |x|$.

Рефлексивність вимагає $\forall a\,(a\ R\ a)$. При підстановці замість абстрактного позначення "R" конкретного відношення, отримуємо $\forall a\,(a = |a|)$. Для від'ємних a така рівність не виконується. Отже, відношення не рефлексивне.

Іррефлексивність вимагає $\forall a(\neg(a\,R\,a))$. При підстановці замість абстрактного позначення "R" конкретного досліджуваного відношення, отримуємо $\forall a(\neg(a = |a|))$, тобто $\forall a\ (a \neq |a|)$. Але для додатних a маємо $a = |a|$, тобто для них <u>не</u>рівність *не* виконується. Отже, відношення не іррефлексивне.

Симетричність вимагає $\forall a \forall b(a\,R\,b \to b\,R\,a)$. При підстановці замість абстрактного позначення "R" конкретного досліджуваного відношення, отримуємо $\forall a \forall b((b = |a|) \to (a = |b|))$ (справді, запис "$a\,R\,b$" означає, що замість ікса підставляється a, замість ігрека — b; запис $b\,R\,a$ — навпаки).

Підставивши $a = -2$, $b = 2$, отримуємо $\underbrace{2 = |-2|}_{\texttt{true}} \to \underbrace{-2 = |2|}_{\texttt{false}} = \texttt{false}$, тобто, контрприклад (див. розд. 1.6.3), бо за означенням симетричності імплікація має виконуватися *для всіх* a і b. Отже, відношення не симетричне.

Антисиметричність вимагає $\forall a \forall b(a\ R\ b \wedge b\ R\ a\ \to\ a = b)$. При підстановці замість абстрактного позначення "R" конкретного досліджуваного відношення, отримуємо $\forall a \forall b((b = |a|) \wedge (a = |b|) \to (a = b))$.

Доведемо, що це твердження істинне (думка рухатися в цьому напрямку може з'явитися зразу, може — після невдалих спроб знайти контрприклад). Треба довести твердження, яке є імплікацією — отже, можна вважати, що записане ліворуч від "$\to$" нам *дано*, а записане праворуч — *треба довести*. Побудувати доведення вдалося. Значить, відношення антисиметричне.

Дано: $b = |a|$ $(1')$ та $a = |b|$ $(2')$.
Довести: $a = b$.
Доведення:

$$\left.\begin{array}{l} b = |a| \ (1') \\ |a| \geqslant 0 \end{array}\right| \Rightarrow b \geqslant 0 \Rightarrow |b| = b \ (3')$$

$$\left.\begin{array}{l} a = |b| \ (2') \\ |b| = b \ (3') \end{array}\right| \Rightarrow a = b \qquad \blacksquare$$

Транзитивність вимагає $\forall a \forall b \forall c(a\,R\,b \wedge b\,R\,c \to a\,R\,c)$. При підстановці замість абстрактного позначення "R" конкретного відношення, отримуємо $\forall a \forall b \forall c((b = |a|) \wedge (c = |b|) \to (c = |a|))$. Знов побудуємо доведення.

Дано: $b = |a|$ $(1')$ та $c = |b|$ $(2')$.
Довести: $c = |a|$.
Доведення:

$$\left.\begin{array}{l} b = |a| \ (1') \\ c = |b| \ (2') \end{array}\right| \Rightarrow c = \big||a|\big| = |a| \text{ (що й треба довести).}$$

Побудувати доведення вдалося. Значить, відношення транзитивне.

Повнота вимагає $\forall a \forall b_{b \neq a}(a\,R\,b \vee b\,R\,a)$. При підстановці замість позначення "R" конкретного відношення, отримуємо $\forall a \forall b_{b \neq a}((b = |a|) \vee (a = |b|))$. Узявши a=2, b=3 (для яких виконується $a \neq b$), отримаємо контрприклад: $\underbrace{3 = |2|}_{\texttt{false}} \vee \underbrace{2 = |3|}_{\texttt{false}} = \texttt{false}$. Отже, відношення не повне.

2.6.2 Відношення еквівалентності

Бінарне відношення називається *відношення еквівалентності* (рос. «*отношение эквивалентности*», англ. «*equivalence relation*»), якщо воно (одночасно) рефлексивне, симетричне та транзитивне.

(Слово «еквівалентні» означає приблизно те са́ме, що «рівноцінні» або «взаємозамінні». Природньо, щоб кожен об'єкт був рівноцінним сам із собою — отже, еквівалентність передбачає рефлексивність. Природньо, щоб два об'єкти були або рівноцінними, або ні, незалежно від порядку — отже, еквівалентність передбачає симетричність. Природньо, що коли (об'єкти a і b) та (об'єкти b і c) рівноцінні між собою, то a і c теж рівноцінні між собою — отже, еквівалентність передбачає транзитивність.)

Найпростішим прикладом відношення еквівалентності є "$x = y$" (на $\mathbb{N}$). Але цей приклад надто простий і мало корисний. Тож розглянемо ще кілька прикладів відношень еквівалентності.

<u>*Приклад 1.*</u> Розглянемо відношення паралельності прямих на площині, але вважаючи, що пряма сама собі теж паралельна. Це відношення рефлексивне (згідно зауваження), симетричне (якщо пряма x не перетинає пряму y, то пряма y теж не перетинає пряму x) та транзитивне (довести можна через транзитивність рівності кутів між цими прямими та спільною січною). Отже, це відношення є відношенням еквівалентності.

Приклад 2. Розглянемо відношення "$x \equiv y \pmod{16}$", задане на $\mathbb{Z}$.

("$x \equiv y \pmod{16}$" читається «x та y *еквівалентні за модулем* 16», або «x та y *порівнювані за модулем* 16», або «x та y *конгруентні за модулем* 16», рос. «x и y *(эквивалентны/сравнимы/конгруэнтны) по модулю* 16», англ. «x and y are *equivalent mod* 16».

Смисл же цього поняття такий: $(x-y) \vdots 16$ (де ":" означає кратність, причому на $\mathbb{Z}$). Іноді "$x \equiv y \pmod{16}$" означають як «чи́сла x і y при діленні на 16 дають однакову остачу»; така дефініція теж правильна, і навіть природніша... але, на жаль, допускає різні трактування. Одні вважають, що остачею від ділення, наприклад, (-25) на 16 є (-9), а інші, що $(+7)$. Й означення через остачу правильне лише при дру́гому трактуванні.

Замість 16 можна взяти будь-яке натуральне число m, це не впливає на суть міркувань; але говорити про *бі*нарне відношення з третім параметром m якось некрасиво...)

Перейдемо до власне доведення.

Коли замість обох параметрів x та y підставити однакове значення a, умова $(a-a) \vdots 16$ виконається, бо $0 \vdots 16$. Отже, $a \equiv a \pmod{16}$ виконується завжди, тобто відношення рефлексивне.

Нехай $a \equiv b \pmod{16}$ виконується. Це означає $(a-b) \vdots 16$, тобто частка $\frac{a-b}{16}$ дорівнює деякому *цілому* k. Тоді $\frac{b-a}{16} = -\frac{a-b}{16} = -k$ теж ціле число, отже $(b-a) \vdots 16$, тобто $b \equiv a \pmod{16}$. Отже, з $a \equiv b \pmod{16}$ випливає $b \equiv a \pmod{16}$, тобто відношення симетричне.

Нехай виконуються твердження $a \equiv b \pmod{16}$ та $b \equiv c \pmod{16}$. Тоді $(a-b) \vdots 16$ та $(b-c) \vdots 16$, тобто $\frac{a-b}{16} = k_1$ та $\frac{b-c}{16} = k_2$ є цілими чи́слами; в такому разі $\frac{a-c}{16} = \frac{(a-b)+(b-c)}{16} = k_1 + k_2$ теж є цілим числом, значить $(a-c) \vdots 16$, тобто $a \equiv c \pmod{16}$. Отже, відношення транзитивне.

Отже, це відношення теж є відношенням еквівалентності.

Приклад 3. Розглянемо скінче́нне відношення, задане наведеною матрицею. Усі пари вигляду (a, a) (головна діагональ матриці) належать відношенню, отже воно рефлексивне.

	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8
z_1	1	0	0	0	0	0	0	0
z_2	0	1	1	1	1	0	0	1
z_3	0	1	1	1	1	0	0	1
z_4	0	1	1	1	1	0	0	1
z_5	0	1	1	1	1	0	0	1
z_6	0	0	0	0	0	1	1	0
z_7	0	0	0	0	0	1	1	0
z_8	0	1	1	1	1	0	0	1

Усі пари вигляду (a, b) і (b, a) (розміщені симетрично відносно головної діагоналі матриці) або одночасно належать відношенню, або одночасно не належать — отже, воно симетричне.

Транзитивність теж виконується, але встановити це можна лише прямим перебором: знайшовши пари (z_2, z_3) та (z_3, z_4), переконуємось, що (z_2, z_4) теж належить відношенню — і так для всіх пар (a, b) та (b, c) зі спільним елементом b.

Отже, це відношення теж є відношенням еквівалентності.

Якщо не звертати увагу на останній рядок та останній стовпчик, матриця має спеціальну структуру: вздовж головної діагоналі йдуть квадратні блоки, що містять лише одиниці, а решта елементів матриці нульові. Не віриться, щоб це було випадково. Але якщо це не випадок, а характерна властивість, то чому ж елемент z_8 порушує її?..

Причина в то́му, що всі множинні означення не залежать від порядку елементів, а «блочність» матриці залежить. Бінарне відношення на (скінче́нній) множині A є відношенням еквівалентності тоді й тільки тоді, коли елементи A *можна переставити* так, щоб матриця ма́ла щойно описаний блочний вигляд.

Приклад 4. Розглянемо відношення тотожньої рівності булевих виразів.

Наприклад, цьому відношенню належать пара ("$x \to y$", "$\neg x \vee y$"), пара ("$z \wedge (x \oplus y)$", "$zy \oplus xz$"), пара ("$\neg t$", "$t \oplus 1$"), пара ("$\neg xy \vee xy$", "y"); *не* належать пара ("$x \vee y$", "$z \vee t$") (залежать від різних змінних), пара ("$x \vee y$", "$x \oplus y$") (мають різні таблиці істинності).

Рефлексивність, симетричність та транзитивність цього відношення легко нестрого аргументувати тим, що йдеться про *рівність* таблиць істинності; але коректне врахування можливої тотожності виразів, які мають однакові суттєві змінні, але різні фіктивні, дещо ускладнює повне доведення. Тож *повіримо*, що це теж відношення еквівалентності.

Розглянемо кілька понять, застосовних *лише* до відношень еквівалентності. (Довільне) відношення еквівалентності будемо позначати "$R_{\equiv}$".

Клас еквівалентності елемента x за відношенням $R_{\equiv}$ (рос. «*класс эквивалентности элемента x по отношению $R_{\equiv}$*», англ. «*equivalence class of x with respect to $R_{\equiv}$*»; позначається "$[x]_{R_{\equiv}}$") — це множина усіх елементів, що перебувають у відношенні еквівалентності $R_{\equiv}$ з елементом x:

$$[x]_{R_{\equiv}} \stackrel{\text{def}}{=} \{y \mid y \, R_{\equiv} \, x\}. \tag{32}$$

Коли і так ясно, про яке відношення йдеться, кажуть і позначають коротше: «*клас еквівалентності x*» («*класс эквивалентности x*», «*equivalence class of x*»); "$[x]$".

Приклади класів еквівалентності для всіх згаданих відношень еквівалентності: 1) клас еквівалентності для осі Ox — сукупність усіх горизонтальних прямих; 2) клас еквівалентності числá 3 — множина $\{16m+3 \mid m \in \mathbb{Z}\} = \{\ldots, -29, -13, 3, 19, 35, 51, \ldots\}$; 3) клас еквівалентності елемента z_3 — множина $\{z_2, z_3, z_4, z_5, z_8\}$; 4) клас еквівалентності виразу "$x \vee y$" — нескінчéнна множина виразів, якій належать, зокрема, "$x \vee y$", "$\neg(\neg x \wedge \neg y)$", "$x \oplus y \oplus xy$".

Класи еквівалентності еквівалентних елементів — рівні множúни:

$$(x \mathrel{R_{\equiv}} y) \Rightarrow ([x]_{R_{\equiv}} = [y]_{R_{\equiv}}). \tag{33}$$

Доведення. За означенням рівності множин, "$[x]_{R_{\equiv}} = [y]_{R_{\equiv}}$" значить

$$\forall z\big((z \in [x]_{R_{\equiv}}) \to (z \in [y]_{R_{\equiv}})\big) \wedge \forall z\big((z \in [y]_{R_{\equiv}}) \to (z \in [x]_{R_{\equiv}})\big).$$

За означенням класу еквівалентності, це означає

$$\forall z\big((z \mathrel{R_{\equiv}} x) \to (z \mathrel{R_{\equiv}} y)\big) \wedge \forall z\big((z \mathrel{R_{\equiv}} y) \to (z \mathrel{R_{\equiv}} x)\big).$$

Нам треба довести це твердження, спираючись на «головну» гіпотезу "$x \mathrel{R_{\equiv}} y$". Отже, достатньо довести "$z \mathrel{R_{\equiv}} y$", спираючись на дві гіпотези "$z \mathrel{R_{\equiv}} x$" та "$x \mathrel{R_{\equiv}} y$", а також довести "$z \mathrel{R_{\equiv}} x$", спираючись на "$z \mathrel{R_{\equiv}} y$" та "$x \mathrel{R_{\equiv}} y$":

$$\text{потрібно довести } \begin{cases} (z \mathrel{R_{\equiv}} x) \wedge (x \mathrel{R_{\equiv}} y) \to (z \mathrel{R_{\equiv}} y), \\ (z \mathrel{R_{\equiv}} y) \wedge (x \mathrel{R_{\equiv}} y) \to (z \mathrel{R_{\equiv}} x). \end{cases}$$

У першому випадку просто застосуємо транзитивність відношення $R_{\equiv}$, у дрýгому — спочатку перетворимо "$x \mathrel{R_{\equiv}} y$" в "$y \mathrel{R_{\equiv}} x$" згідно симетричності $R_{\equiv}$, а вже потім застосуємо транзитивність $R_{\equiv}$. ■

Зворотнє твердження, $([x]_{R_{\equiv}} = [y]_{R_{\equiv}}) \Rightarrow (x \mathrel{R_{\equiv}} y)$, випливає безпосередньо з означення класа еквівалентності.

Доведення. Кожен елемент належить власному класу еквівалентності ($x \in [x]_{R_{\equiv}}$); замінивши $[x]_{R_{\equiv}}$ на рівну (за гіпотезою теореми) множину $[y]_{R_{\equiv}}$, отримуємо $x \in [y]_{R_{\equiv}}$, що якраз і означає $x \mathrel{R_{\equiv}} y$. ■

Засобами мат. логіки можна отримати рівносильне твердження «Якщо елементи не еквівалентні, то їхні класи еквівалентності не рівні». Але детальніший аналіз засобами предметної області дозволяє отримати значно «потужніший» висновок:

Класи еквівалентності не еквівалентних елементів не перетинаються:

$$(x, y) \notin R_{\equiv} \Rightarrow [x]_{R_{\equiv}} \cap [y]_{R_{\equiv}} = \varnothing. \tag{34}$$

Доведення. Припустимо, ніби це слідування не виконується. Тобто, припускаємо, ніби можна знайти таке відношення еквівалентності $R_{\equiv}$ і такі елементи x, y, що елементи не еквівалентні, але, не зважаючи на це, $[x]_{R_{\equiv}} \cap [y]_{R_{\equiv}} \neq \varnothing$. «Множина не порожня» означає, що їй належить деякий (хоча б один) елемент c. Тоді $c \in ([x]_{R_{\equiv}} \cap [y]_{R_{\equiv}})$ означає $(c \in [x]_{R_{\equiv}}) \wedge (c \in [y]_{R_{\equiv}})$, тобто $(c \mathrel{R_{\equiv}} x) \wedge (c \mathrel{R_{\equiv}} y)$; в такому разі, перетворивши $(c \mathrel{R_{\equiv}} x)$ у $(x \mathrel{R_{\equiv}} c)$ згідно симетричності $R_{\equiv}$ і використавши транзитивність $R_{\equiv}$, отримаємо $(x \mathrel{R_{\equiv}} y)$, що суперечить припущенню. Ми припустили *в точності* протилежне до того, що потрібно було довести; отже, отримане протиріччя доводить початкове твердження. ■

Тобто, класи еквівалентності або однакові (є одним класом), або не перетинаються. Інакше кажучи, ***не однакові класи еквівалентності не перетинаються***.

Згадаємо, що клас еквівалентності можна будувати для будь-якого елемента. Звідси можна зробити такі (насправді рівносильні) висновки:

1. *Кожен елемент належить* ***в точності одному*** *класу еквівалентності.*
2. *Об'єднання всіх класів еквівалентності дорівнює множині, на якій визначене відношення еквівалентності.*

Це сáме записують також у вигляді співвідношень

$$S_i \cap S_j = \varnothing \text{ (при } i \neq j), \qquad \bigcup_i S_i = A \tag{35}$$

(де $S_{\ldots}$ — сукупність усіх різних класів еквівалентності, A — множина, на якій задане відношення).

Набір підмножин, що має властивості: 1) різні підмножúни не перетинаються; 2) об'єднання всіх підмножин дорівнює початковій надмножині — називають *розбиття* (рос. «*разбиение*», англ. «*partition*»).

(Розбиття множúни може з'явитися і не внаслідок побудови класів еквівалентності. Але коли на множині задане відношення еквівалентності, класи еквівалентності завжди утворюють розбиття; так само, коли є розбиття, завжди можна утворити відношення еквівалентності «належать одній підмножині».)

Ще одне поняття, тісно пов'язане з класами еквівалентності — *фактор-множина* (рос. «*фактор-множество*», англ. «*quotient set*»). Так називають множину всіх класів еквівалентності.

(Для наведеного вище 3-го прикладу відношення еквівалентності, фактор-множина має вигляд $\{\{z_1\}, \{z_2, z_3, z_4, z_5, z_8\}, \{z_6, z_7\}\}$, тобто складається з трьох елементів, одним з яких є одноелементна множина $\{z_1\}$, іншим — п'ятиелементна множина $\{z_2, z_3, z_4, z_5, z_8\}$, ще іншим — двоелементна множина $\{z_6, z_7\}$.

Для відношення $x \equiv y \pmod{16}$ фактор-множина складається з 16 елементів: множи́ни всіх чисел, кратних 16; множи́ни всіх чисел, що дають остачу 1 при діленні на 16; множи́ни всіх чисел, що дають остачу 2 при діленні на 16; і т. д. (до 15 включно).)

Ще розглянемо поняття «*представник класу еквівавлентності*» (рос. «*представитель класса эквивалентности*», англ. «*representative of equivalence class*»). Це означає, що з кожного класу еквівавлентності вибирається деякий елемент і оголошується представником. *Важливо чітко зафіксувати* ***одно́го*** *представника кожного класу.*

Наприклад, у розд. 1.3.1 відзначено, що часом незручно, що багато різних формул можуть задавати одну й ту са́му залежність (таблицю істинності), і введено додаткові обмеження, які створили нормальні форми. ДДНФ якраз і є природним прикладом представників класів (конкретний вираз-ДДНФ є представником класу всіх логічних виразів, що мають таку ж, як ця ДДНФ, таблицю істинності).

Ще доречно згадати початок розд. 2.3.2. Розглядалося питання, як подати множину. Серед стандартних типів мови програмування є масив, який зручно використовувати для збереження (скінче́нних) послідовностей. Конкретна послідовність задає собою (єдину) множину, але різні послідовності можуть задавати одну й ту са́му множину (наприклад, $(2, 7, 5)$, $(5, 2, 7)$, $(5, 7, 2)$, ...). Введемо серед послідовностей бінарне відношення «відповідають одній і тій са́мій множині»; очевидно, воно є відношенням еквівалентності. Звідси, класи еквівалентності послідовностей і множи́ни *взаємно*-однозначно відповідають одні́ о́дним. Тому́ кожну множину можна подати, вибравши деякого представника відповідного класу еквівалентності. Найзручнішими представниками класів виявляються відсортовані послідовності, бо до них застосовні злиття та бінпошук.

У деяких випадках може бути зручно використати представників класів відношення еквівалентності, заданого на невеликій множині, як особливий спосіб подання, придатний лише для такого різновиду відношень. Для цього слід завести *одно*вимірний масив, індексами якого є елементи, значеннями — представник того класу еквівалентності, куди належить цей елемент. Так, для відношення еквівалентності, фактор-множина якого має вигляд $\{\{0, 1\}, \{2, 3, 4, 5, 8\}, \{6, 7\}\}$, таким масивом може бути, наприклад, $\begin{array}{c|c|c|c|c|c|c|c|c} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 0 & 0 & 2 & 2 & 2 & 2 & 6 & 6 & 2 \end{array}$ (якщо представником класу $\{0, 1\}$ взяти 0, представником класу $\{2, 3, 4, 5, 8\}$ взяти 2, і представником класу $\{6, 7\}$ взяти 6). Причому, в принципі можна взяти й інших представників, наприклад, 1 від класу $\{0, 1\}$, 5 від класу $\{2, 3, 4, 5, 8\}$, і так са́мо 6 від класу $\{6, 7\}$, тоді вийде інше правильне подання цього самого скінче́нного відношення еквівалентності $\begin{array}{c|c|c|c|c|c|c|c|c} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & 1 & 5 & 5 & 5 & 5 & 6 & 6 & 5 \end{array}$. Але важливо один раз прийняти деяке рішення щодо того, які елементи будуть представниками своїх класів, й потім його неухильно дотримуватися.

2.6.3 Відношення порядку

Бінарне відношення називається *відношення порядку* (рос. «*отношение порядка*», англ. «*order*», «*order relation*», «*ordering relation*»), коли воно антисиметричне і транзитивне.

(Відношення порядку формалізують співвідношення на зразок «більший», «кращий», «смачніший», тощо. Отже, якщо є два різні предмети, не повинно б бути такого, що кожен кращий за інший — тому́ антисиметричність; якщо 1-ий предмет кращий за 2-ий, 2-ий кращий за 3-ій, то природньо, щоб 1-ий був кращий за 3-ій — тому́ транзитивність.)

Виділяють різні типи відношень порядку.

Якщо відношення порядку рефлексивне, його називають *відношення нестрогого порядку*; якщо іррефлексивне — *відношення строгого порядку.* (Переклади: нестрогого порядку — рос. «*отношение нестрогого порядка*», англ. «*non-strict order*», «*weak order*»; строгого порядку — рос. «*отношение строгого порядка*», англ. «*strict order*».)

(Наприклад, числове "$<$" (котре так і називають «*строго* менше») є відношенням строгого порядку, числове "$\leqslant$" («менше-або-рівне») — відношенням нестрогого порядку.)

Бувають відношення, одночасно і не рефлексивні, і не іррефлексивні. Як наслідок, при бажанні можна побудувати відношення порядку, які не є ні строгими, ні нестрогими. Але більшість осмислених відношень порядку — або строгі, або нестрогі.

Якщо відношення порядку лінійне (повне), його називають *відношення лінійного порядку* (рос. «*отношение линейного порядка*», англ. «*linear order*», «*total order*»).

Наприклад, числове "$<$" є відношенням лінійного порядку, бо будь-які два не рівні між собою числá можна порівняти і сказати, яке менше.

А тепер розглянемо множину тар (сумок, валіз, ящиків, ...) і введемо відношення «менше» як «тара x менша за тару y, коли x можна (не складаючи) засунути всере́дину y». Це відношення порядку, але не лінійне. Наприклад, розглянемо майже плоский дипломат $40 \times 25 \times 10$ см і майже кубічний ящик $25 \times 20 \times 20$ см. Жоден об'єкт не поміщається всере́дину іншого, значить — жоден не менший. Такі елементи *не порівнювані* (рос. «*не сравнимы*», англ. «*non-comparable*»).

Ще один приклад нелінійного відношення порядку розглядався в означенні класу Поста M (розд. 1.7.2, стор. 37). Там наголошувалося, що з двох пар $(0,1)$ та $(1,0)$ жодна не є більшою за іншу в тому смислі, який потрібен в означенні класу M — тобто, вони не порівнювані.

Наведемо приклади відношень порядку різних типів:

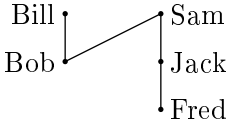
Тип	Приклади
Відношення строгого лінійного порядку	$x < y$ (числове менше)
Відношення нестрогого лінійного порядку	$x \leqslant y$ (числове менше-рівне)
Відношення строгого не лінійного порядку	$X \subset Y$ (включення множин як *власних* підмножин); «тара x поміщається всере́дині тари y»
Відношення нестрогого не лінійного порядку	$X \subseteq Y$ (включення множин); $x \vdots y$ (кратність) на $\mathbb{N}$

(На жаль, у класифікації відношень порядку поки що нема єдиної чітко встановленої термінології. Зокрема, майже в усіх джерелах вводять термін «відношення часткóвого порядку» (рос. «частичного», англ. «partial») — але в одних книжках це те са́ме, що у нас просто «відношення порядку» (не обов'язково лінійне, але може бути й лінійним), в інших — обов'язково нелінійне відношення порядку... Так що в цьому посібнику такого терміну уникатимемо, незважаючи на його відносну поширеність.)

Будемо позначати відношення порядку (байдуже, лінійні чи ні) значками "$\prec$" (для строгих відношень) та "$\preccurlyeq$" (для нестрогих). Відношення лінійного порядку будемо позначати "$<$" та "$\leqslant$" відповідно. Записи "$x \prec y$" та "$x < y$" будемо (*умовно!*) читати як «x менше y» або «y більше x».

За будь-яким відношенням нестрогого порядку можна побудувати відповідне йому відношення строгого порядку (поклавши $x \prec y \overset{\text{def}}{\Longleftrightarrow} (x \preccurlyeq y) \wedge (x \neq y)$); аналогічно можна ввести $x \preccurlyeq y$ як $(x \prec y) \vee (x = y)$. Завдяки цьому, подальші поняття формально вводяться лише для "$\prec$", а застосовні до різних типів порядку.

Для подання скінче́нних відношень порядку буває зручно використати т. зв. *діаграми Гассе* (*Hasse diagrams*). Вершинами зображаються елементи, ре́брами (лініями) — факти наявності зв'язкý; при цьому, менші елементи розміщують нижче за більші, а ре́бра проводять, коли елементи порівнювані, але це не можна встановити за транзитивністю через інші елементи.



(Наприклад, нехай на множині спортсменів {Bill, Bob, Fred, Jack, Sam} введене відношення «гарантовано кращий»: Bill кращий за Bob-a, Sam кращий і за Fred-a, і за Jack-a, і за Bob-a, Jack кращий за Fred-a; решта пар спортсменів не порівнювані. Перевіривши, що транзитивність та антисиметричність виконуються, бачимо, що це справді відношення порядку.

Щоб не образити кращих спортсменів, вважаємо, що $x \prec y$, коли y кращий за x.

Тоді вершина Sam природньо потрапляє на самий верх, вершина Fred — на самий низ, тощо. (те, що вершини Bill і Sam та вершини Bob і Jack потрапили на один рівень, не суттєво). Лінії проводимо для пар вершин (Bill; Bob), (Sam; Bob), (Sam; Jack) та (Jack; Fred); окремої лінії для пари (Sam; Fred) проводити не треба, бо факт, що Sam кращий за Fred-a, і так видний з того, що є маршрут Sam – Jack – Fred, всі переміщення по якому відбуваються згори донизу.)

Елемент x — *максимальний* (рос. «*максимальный*», англ. «*a maximal*»), коли жоден елемент не більший за нього:

$$x \text{ максимальний} \overset{\text{def}}{\Longleftrightarrow} \neg(\exists y(x \prec y)). \tag{36}$$

Елемент x — *мінімальний* (рос. «*минимальный*», англ. «*a minimal*»), коли жоден елемент не менший за нього:

$$x \text{ мінімальний} \overset{\text{def}}{\iff} \neg(\exists y(y \prec x)). \tag{37}$$

Елемент x — *найбільший* (рос. «*наибольший*», англ. «*the greatest*»), коли він більший за будь-який інший елемент:

$$x \text{ найбільший} \overset{\text{def}}{\iff} \forall y_{y\neq x}(y \prec x). \tag{38}$$

Елемент x — *найменший* (рос. «*наименьший*», англ. «*the least*»), коли він менший за будь-який інший елемент:

$$x \text{ найменший} \overset{\text{def}}{\iff} \forall y_{y\neq x}(x \prec y). \tag{39}$$

(Наприклад, для відношення «про спортсменів» є два максимальні елементи (Bill і Sam) та два мінімальні (Bob і Fred), а найбільшого і найменшого нема.)

Наведемо основні твердження про мінімальні та найменші елементи. Звичайно, їх можна «симетрично» переформулювати і довести також для максимальних та найбільших елементів.

Будь-яка непорожня скінче́нна множина, на якій задане відношення порядку, має мінімальний елемент.

(Не виключено, що мінімальних елементів буде кілька різних.)

Доведення. Беремо довільний елемент $x_{(0)}$.[21] Пробуємо знайти елемент $x_{(1)}$, для якого $x_{(1)} \prec x_{(0)}$. Якщо це не вдається, мінімальним є сам $x_{(0)}$. Якщо ж знайти $x_{(1)}$ вдається, то пробуємо знайти $x_{(2)}$, для якого $x_{(2)} \prec x_{(1)}$. І т. д.

Розглянемо елементи $x_{(i)}$ та $x_{(j)}$ при $j > i$. За побудовою, $x_{(j)} \prec x_{(j-1)}$, $x_{(j-1)} \prec x_{(j-2)}$, …, $x_{(i+1)} \prec x_{(i)}$; значить, за транзитивністю, $x_{(j)} \prec x_{(i)}$. Ми розглядаємо відношення строгого порядку, а воно іррефлексивне. Отже, $x_{(j)} \neq x_{(i)}$. Наведені міркування застосовні до будь-яких i та j — отже, *всі значення* $x_{(0)}$, $x_{(1)}$, $x_{(2)}$, … *різні.*

З іншого боку, множина за умовою *скінче́нна.*

Ці два спостереження разом узяті доводять, що рано чи пізно (гарантовано не пізніше, ніж через $|A|$ кроків, де $|A|$ — кількість елементів множини́) не вдасться знайти $x_{(k+1)}$ такого, щоб $x_{(k+1)} \prec x_{(k)}$, а значить — $x_{(k)}$ буде мінімальним. ■

Якщо найменший елемент існує, то він — єдиний мінімальний.

Доведення. Спочатку покажемо, що найменший елемент є мінімальним. Припустимо, ніби x найменший ($\forall y_{y\neq x}(x \prec y)$), і водночас цей самий x не мінімальний ($\exists z(z \prec x)$). Розглянемо те значення $z_{(0)}$, при котрому виконалося $z_{(0)} \prec x$. Порядок "$\prec$" строгий, значить $z_{(0)} \neq x$ і $z_{(0)}$ входить до множини́, що перебирається квантором "$\forall y_{y\neq x}\ldots$"; отже, $x \prec z_{(0)}$. Але ж одночасне виконання $z_{(0)} \prec x$, $x \prec z_{(0)}$ та $z_{(0)} \neq x$ суперечить антисиметричності відношення порядку. Значить, маємо протиріччя, джерело якого — у припущенні, ніби x найменший, але не мінімальний.

Тепер покажемо, що не може бути інших мінімальних елементів. Припустимо, ніби x найменший і водночас z ($z \neq x$) мінімальний. «x найменший» означає "$\forall y_{y\neq x}(x \prec y)$"; це виконується в т. ч. і для елемента z. Значить, $x \prec z$. Отже, $\neg\big(\exists y(y \prec z)\big) = \texttt{false}$ (бо в якості y можна узяти x, сам квантифікований предикат буде істиною, його заперечення — хибою), тобто z не може бути мінімальним. ■

Для відношень ***лінійного*** *порядку поняття «мінімальний» і «найменший» рівносильні.*

Доведення. З урахуванням вже доведеного твердження «найменший елемент є єдиним мінімальним», лишається тільки показати, що мінімальний елемент лінійного порядку обов'язково є найменшим.

Тож нехай x мінімальний, тобто $\neg(\exists y(y<x))$.[22] Занесемо заперечення всере́дину квантора: $\forall y(\neg(y<x))$. З іншого боку (за означенням лінійності), $\forall y_{y\neq x}(x < y \vee y < x)$. Тобто, для кожного y (крім x) виконується щось одне з "$x < y$" або "$y < x$", і при цьому не виконується "$y < x$". Значить, лишається тільки варіант "$x < y$". Отже, для всіх y (не рівних x) виконується "$x < y$" (що і є означенням найменшого елемента). ■

2.6.4 Розширення відношень порядку та топологічне сортування відношень

Розширення відношення порядку *до лінійного* (рос. «*расширение* отношения порядка *до линейного*», англ. «*linear extention* of order») — це таке відношення, яке: (1) є надмножиною початкового відношення порядку; (2) є відношенням лінійного порядку.

[21] індекс пишемо у дужках, щоб підкреслити: ця нумерація ніяк не зв'язана з «основною» нумерацією елементів множини́ (якщо така є)

[22] Значок "$<$", а не "$\prec$" обумовлений тим, що за умовою відношення порядку лінійне; але це не конкретне числове менше, а позначення *довільного* лінійного порядку.

Іншими словами: (1) для всіх пар елементів, що були порівнювані, лишається те са́ме співвідношення, який з них менший і який більший (в цьому й полягає «надмножина»); (2) для тих пар, що були не порівнюваними, приймається вольове рішення, який елемент вважати меншим, який більшим. Як правило — довільним чином, аби вийшло відношення лінійного порядку.

(Наприклад, нехай є вже згадане не лінійне відношення «кращий» на множині спортсменів; нехай виникає потреба нагородити їх послідовно, оди́н за о́дним, причому так, щоб спочатку швиденько пооголошувати слабших, а потім особливо урочисто і докладно вшанувати сильніших. Тобто, лінійний (бо «послідовно, оди́н за о́дним») порядок нагородження слід побудувати так, щоб слабші за досягненнями (менші у смислі нелінійного відношення) викликалися раніше; якщо спортсмени не порівнювані між собою, то їх взаємне розміщення може бути яким завгодно.

Зверніть увагу, що для одного й того ж (наведеного на стор. 67 діаграмою Гассе та словами) відношення можуть бути кілька різних правильних розширень: Bob, Bill, Fred, Jack, Sam; Fred, Jack, Bob, Sam, Bill; Fred, Bob, Jack, Sam, Bill та ще багато інших.

Тобто, результат не визначається однозначно початковим відношенням.

Очевидно також, що смисл розширеного відношення може істотно відрізнятися від смислу початкового. Якщо в одному розширеному відношенні Bob 1-й, а в іншому 3-й — це не означає, що він відповідно там найслабший, а там середній, а лише те, що не такий уже й тісний зв'язок між «раніше у списку» та «слабший спортсмен». А якщо спочатку вводили нелінійний порядок на множині ящиків та валіз, що має смисл «менше поміщається всере́дину більшого», і ящик $25 \times 20 \times 20$ був непорівнюваним із дипломатом $40 \times 25 \times 10$, а потім побудували його розширення, за яким цей ящик оголошений меншим за цей дипломат — неможливість засунути такий ящик всере́дину такого дипломата нікуди не ділась, просто розширене відношення до цього вже ставиться інакше.)

Будь-яке відношення порядку, задане на скінченній множині, можна розширити до відношення лінійного порядку.

Доведення. Щоб побудувати лінійний порядок — розширення заданого порядку, будемо діяти згідно такого алгоритму:

```
X = A; // кладемо у X копію множини́ A, на якій задане відношення порядку
while(X≠ ∅) {
  x_curr = (який-небудь) мінімальний (згідно заданого порядку) елемент множини́ X;
         // алгоритм пошуку мінімального елемента описаний у доведенні твердження
         // "будь-яке скінче́нне відношення порядку має мінімальний елемент"
  вивести(x_curr);
  X \= {x_curr} // вилучаємо елемент x_curr з множини́ X
}
```

Порядок, у якому елементи потраплятимуть у зовнішній світ за допомогою "`вивести(x_curr)`", лінійний. І цей лінійний порядок є розширенням заданого, бо щоразу вибирається мінімальний серед ще не розглянутих елементів (а отже — якщо $x \prec y$, то x буде вибраний і виведений раніше за y). ■

Процес розширення не лінійного порядку до лінійного у випадку скінче́нних множин називають також *топологі́чним сортуванням* (рос. «*топологическая сортировка*», англ. «*topological sorting*»).

(Наголосимо (хоч це й видно з означення), що топологічне сортування *не є* ще одним алгоритмом сортування масивів чи інших послідовностей. Задача істотно відрізняється від класичної задачі сортування — щонайменше, наявністю порівнюваних і непорівнюваних пар елементів.

Про топологічне сортування ще згадуватиметься наприкінці розд. 5.6.2, де буде запропоновано схожий, але інший варіант постановки задачі, а також запропоновано більш ефективний алгоритм на основі пошуку в глибину.)

Приклад топологічного сортування. Нехай відношення порядку задане матрицею, наведеною праворуч (при бажанні можна перевірити, що це відношення порядку, тобто що матриця справді задає антисиметричне транзитивне відношення).

	α	β	◯	△	□	δ	ε
α	1	1	0	0	0	0	1
β	0	1	0	0	0	0	0
◯	0	1	0	0	0	0	1
△	0	0	0	0	0	0	0
□	0	0	0	0	1	0	0
δ	1	1	1	0	0	0	1
ε	0	1	0	0	0	0	0

Згідно алгоритму зі стор. 69, слід багатократно шукати і виводити мінімальний елемент. Щоб шукати мінімальний елемент алгоритмічно, варто користуватися міркуваннями доведення леми про існування мінімального елементу (стор. 68). Для ручного ж виконання зручніше діяти напряму через означення.

Наприклад, елемент α не є мінімальним, бо згідно матриці $(\delta, \alpha) \in R$, тобто $\delta \prec \alpha$. А, наприклад, для елемента △ нема жодного елемента x, щоб було $x \prec$ △ (бо відповідний △ стовпчик складається лише з нулів); те саме і для елемента δ (одиничкиу *рядку* δ є, але вони означають $\delta \prec \alpha$, $\delta \prec \beta$, тощо, отже не заважають δ бути мінімальним); для елемента □ одиничка у стовпчику є, але вона в рядку □ (на головній діагоналі), так що жоден *інший* елемент не менший □. Отже, в якості першого мінімального елементу можна взяти будь-який один з △, □ або δ. Візьмемо △.[23]

Після цього елемент △ слід вилучити з відношення й повторити увесь процес. Внаслідок вилучення △ з початкової матриці отримуємо матрицю відношення порядку

	α	β	◯	□	δ	ε
α	1	1	0	0	0	1
β	0	1	0	0	0	0
◯	0	1	0	0	0	1
□	0	0	0	1	0	0
δ	1	1	1	0	0	1
ε	0	1	0	0	0	0

.

Для неї одним з мінімальних елементів є □. Видаємо його на вихід і вилучаємо з відношення, отримуючи матрицю

	α	β	◯	δ	ε
α	1	1	0	0	1
β	0	1	0	0	0
◯	0	1	0	0	1
δ	1	1	1	0	1
ε	0	1	0	0	0

.

Для неї мінімальним елементом (на цьому кроці єдиним) є δ. Видаємо його на вихід і вилучаємо з відношення, отримуючи матрицю

	α	β	◯	ε
α	1	1	0	1
β	0	1	0	0
◯	0	1	0	1
ε	0	1	0	0

.

Для неї одним з мінімальних елементів є α. Видаємо його на вихід і вилучаємо з відношення, отримуючи матрицю, наведену праворуч.

	β	◯	ε
β	1	0	0
◯	1	0	1
ε	1	0	0

Для неї мінімальним елементом (на цьому кроці єдиним) є ◯. Видаємо його на вихід і вилучаємо з відношення, отримуючи матрицю, наведену праворуч.

	β	ε
β	1	0
ε	1	0

Для неї мінімальним елементом (на цьому кроці єдиним) є ε. Видаємо його на вихід і вилучаємо з відношення, отримуючи відношення, задане на множині, що містить єдиний елемент β. Він і буде черговим мінімальним і теж буде поданий на вихід.

Множина, на якій задане відношення порядку, стала порожньою, роботу алгоритму завершено. Результатом роботи алгоритму (відповіддю задачі) є та послідовність, в якій елементи подавалися на вихід: △, □, δ, α, ◯, ε, β.

2.6.5 Рефлексивність, ірефлексивність, симетричність, антисиметричність, транзитивність, повнота (альтернативні означення). Замикання відношень

Усі означення стандартних класів бінарних відношень (розд. 2.6.1) в принципі можна сформулювати так:

$$R \text{ рефлексивне} \quad \Leftrightarrow \quad I_A \subseteq R \tag{40}$$

$$R \text{ ірефлексивне} \quad \Leftrightarrow \quad I_A \cap R = \varnothing \tag{41}$$

$$R \text{ симетричне} \quad \Leftrightarrow \quad R^{-1} = R \tag{42}$$

$$R \text{ антисиметричне} \quad \Leftrightarrow \quad R^{-1} \cap R \subseteq I_A \tag{43}$$

[23] насправді не варто шукати всі мінімальні елементи, достатньо якийсь один; всі наведені лише заради пояснення, чому топологічне сортування цього ж відношення може мати інші правильні відповіді

$$R \text{ транзитивне} \quad \Leftrightarrow \quad R \circ R \subseteq R \tag{44}$$

$$R \text{ лінійне (повне)} \quad \Leftrightarrow \quad R \cup R^{-1} \cup I_A = A^2 \tag{45}$$

(Від викладення *всіх* доведень утримаємось, бо більшість з них однотипні. Наведемо лише (дуже типове) доведення для іррефлексивності та (досить не типове) доведення для транзитивності.)

Доведення. Іррефлексивність. Покажемо, що з нового означення іррефлексивності (41) випливає старе (27). "$R \cap I_A = \varnothing$" означає «жодна пара не належить одночасно і R, і I_A», тобто $\neg\big(\exists x_{x\in A^2}(x \in R \wedge x \in I_A)\big) = \neg(\exists x_{x\in I_A}(x \in R))$; враховуючи, що "$x \in I_A$" означає «в якості x можна брати лише пáри $(a,a)_{a\in A}$», "$(x \in R)$" означає "$a\,R\,a$", а твердження в цілому перетворюється у $\neg(\exists a\; a\,R\,a) = \forall a(\neg(a\,R\,a))$, що і є старим означенням (27).

Тепер покажемо, що зі старого означення (27) випливає нове (41). Розглянемо *рівносильне* (за контрапозицією) твердження «якщо не виконується нове означення, то не виконується і старе». $R \cap I_A \neq \varnothing$ означає $\exists x_{x\in A^2}(x \in R \wedge x \in I_A)$, тобто $\exists a \exists b(a\,R\,b \wedge (a = b))$, тобто $\exists a(a\,R\,a)$, що суперечить старому означенню іррефлексивності. ■

Доведення. Транзитивність. Перш за все, розпишемо нове означення транзитивності (44) $R \circ R \subseteq R$ за означеннями композиції відношень та включення множин. Виходить $\forall a \forall c\big(\underbrace{(\exists b(a\,R\,b \wedge b\,R\,c))}_{(a,c)\in(R\circ R)} \to a\,R\,c\big)$.

$$\begin{aligned}
&\forall a \forall c\big((\exists b(a\,R\,b \wedge b\,R\,c)) \to a\,R\,c\big) = && \text{виражаємо імплікацію як "}\neg x \vee y\text{"}\\
&= \forall a \forall c\big(\neg(\exists b(a\,R\,b \wedge b\,R\,c)) \vee a\,R\,c\big) = && \text{заносимо заперечення всередину квантора}\\
&= \forall a \forall c\big(\forall b(\neg(a\,R\,b \wedge b\,R\,c)) \vee a\,R\,c\big) = && \text{переносимо "}\forall b\text{" згідно правила}\\
& && (\forall x P(x)) \vee A = \forall x(P(x) \vee A)) \text{ (бо "}a\,R\,c\text{" не містить}\\
& && \text{вільних входжень } b)\\
&= \forall a \forall c \forall b\big(\neg(a\,R\,b \wedge b\,R\,c) \vee a\,R\,c\big) = && \text{вертаємо "}\neg x \vee y\text{" до вигляду імплікації, переставля-}\\
& && \text{ємо місцями однойменні квантори}\\
&= \forall a \forall b \forall c\big(a\,R\,b \wedge b\,R\,c \to a\,R\,c\big) && \text{маємо старе означення транзитивності (30)}
\end{aligned}$$

Ці перетворення правильні при виконанні в будь-якому напрямку, тому вони доводять не лише слідування, а і рівносильність. ■

Замикання (*Рефлексивне / симетричне / транзитивне*) *замикання* (рос. «*замыкание*», англ. «*closure*») відношення R — це найменше серед відношень, які є надмножинами R і при цьому (рефлексивні / симетричні / транзитивні).

(Тобто, тут задані три незалежні, але слово у слово однакові означення. «Найменше серед відношень» тут слід розуміти у смислі означення найменшого елемента зі стор. 68, при застосуванні до нелінійного порядку "$\subset$" на множині бінарних відношень.

Ці означення *не конструктивні*: вони не вказують, *як* будувати відповідні замикання. Більш того, з них навіть не очевидно, чи завжди існують ці замикання. Томý наведемо конструктивні способи побудови цих замикань. Тільки, раз тепер з'являються окремо означення й окремо способи побудови, то треба ще доводити, що наведені далі конструктивні способи справді будують сáме те, що описане у вищезгаданих означеннях. Заодно, цим буде ще й доведено, що замикання існують для всіх відношень.)

Рефлексивне замикання R можна побудувати як

$$r(R) = R \cup I_A. \tag{46}$$

Доведення. Перше означення рефлексивного замикання вимагає, щоб $R \subseteq r(R)$, тому $r(R)$ мусить містити усі пари-елементи з R; те перше означення також вимагає, щоб відношення $r(R)$ було рефлексивним, тобто згідно (40) вимагає, щоб $I_A \subseteq r(R)$, тому $r(R)$ мусить містити усі пари-елементи з I_A. Очевидно, одночасно обидві ці вимоги виконують лише $R \cup I_A$ та її надмножúни; отже, $R \cup I_A$ і є найменшим таким відношенням. ■

Симетричне замикання R можна побудувати як

$$s(R) = R \cup R^{-1}. \tag{47}$$

Доведення. Позначимо довільне (не обов'язково найменше) симетричне відношення — надмножину R — як R_2. Розглянемо довільну пару (a,b) таку, що $a\,R^{-1}\,b$. За означенням оберненого відношення, це те сáме, що $b\,R\,a$. Оскільки $R \subseteq R_2$, то $b\,R\,a \to b\,R_2\,a$. Оскільки R_2 симетричне, то (за старим означенням) $b\,R_2\,a \to a\,R_2\,b$. Поєднавши всі ці перетворення, маємо $\forall a \forall b(a\,R^{-1}\,b \to a\,R_2\,b)$, тобто $R^{-1} \subseteq R_2$.

Отже, для будь-якої надмножúни R, що є симетричним відношенням, $(R \subseteq R_2) \wedge (R^{-1} \subseteq R_2)$; як уже згадувалося, це рівносильно $(R \cup R^{-1}) \subseteq R_2$.

На жаль, доведення цим не закінчується: ми довели слідування в один бік (якщо (R_2 симетричне) і (R_2 надмножина R), то (R_2 надмножина $R \cup R^{-1}$)), але не у зворотній.

Той факт, що $R \cup R^{-1}$ є надмножиною R (як вимагає означення замикання), очевидний. Значить, лишилося довести, що $R \cup R^{-1}$ симетричне. Перетворимо $(R \cup R^{-1})^{-1} = (R^{-1} \cup (R^{-1})^{-1}) = (R^{-1} \cup R) = (R \cup R^{-1})$. Тобто, $(R \cup R^{-1})^{-1} = (R \cup R^{-1})$, що якраз і є новим означенням симетричності.

Нарешті, маємо доведення всіх трьох умов (надмножина; симетрична; найменша). ■

Транзитивне замикання R можна побудувати як

$$t(R) = R \cup R^2 \cup R^3 \cup \ldots; \tag{48}$$

якщо R задане на скінче́нній множині з кількістю елементів $|A| = n$, то $t(R) = R \cup R^2 \cup \cup R^3 \cup \ldots \cup R^{n-1}$.

На стор. 60 сказано, що R^i містить в точності ті пари елементів, для яких відношення R «виконується через ланцюжок, що містить i переходів». А для транзитивного замикання якраз і потрібно, щоб йому належали i пари, що самі́ перебувають у відношенні R, i пари, що зв'язані через «ланцюжок довжино́ї 2», i через «ланцюжок довжино́ї 3», і т. д. Так що лишилося довести тільки те, що для відношень на скінче́нній множині можна завершувати цей процес на $(n-1)$-му степеню відношення. Цей момент поки що залишимо не з'ясованим (повернемось до нього у розд. 5.3, стор. 121). ■

Втім, цей спосіб, хоч і конструктивний, *не* є найкращим алгоритмом пошуку транзитивного замикання; і легшим для написання, і трохи швидшим є алгоритм Воршалла (розд. 5.7.2).

Часто буває потрібне *рефлексивно-транзитивне* замикання R^*, котре являє собою рефлексивне замикання транзитивного замикання, а отже — може бути побудоване як $R^* = I_A \cup R \cup \cup R^2 \cup R^3 \cup \ldots \cup R^{n-1}$; це може бути виражене також простішою формулою $R^* = (I_A \cup R)^{n-1}$.

2.7 Функції (огляд)

(Частина означень цього розділу загальновідома. Але все ж запишемо їх формально.)

(Унарна) функція — це бінарне («у широкому смислі») відношення, для якого виконується додаткова умова $\forall a \forall b \forall c \big(a\, R\, b \wedge a\, R\, c \to (b = c)\big)$, тобто не може бути різних дру́гих елементів, що перебувають у відношенні з одним і тим са́мим першим елементом.

(Наприклад, відношення $\{(1,a),\ (2,a),\ (3,a),\ (4,b)\}$ є функцією, а відношення $\{(1,a),\ (1,b),\ (2,a),\ (3,b)\}$) — не функція, бо є дві різні пари з 1-им елементом "1".)

Узагальнювальне означення n-арної функції аналогічне: *(n-арна) функція* — це $(n+1)$-арне відношення, для якого виконується додаткова умова $\forall a_1 \forall a_2 \ldots \forall a_n \forall b \forall c \big((a_1, a_2, \ldots, a_n, b) \in R \wedge \wedge (a_1, a_2, \ldots, a_n, c) \in R \to (b = c)\big)$, тобто не може бути різних останніх елементів, що перебувають у відношенні з однією й тією са́мою енкою попередніх елементів.

Ми в основному обмежимося розглядом унарних функцій.

Для унарних функцій, 1-ий елемент пари $(a \in A)$ називають *аргументом* або *параметром*;[24] останній елемент $(b \in B)$ — *значенням* або *результатом*.

Є три способи позначити, що елементи a і b перебувають у зв'язку́, що задається функцією f. Ці способи перелічені у табличці праворуч.

$(a,b) \in f$	(як для всіх множин)
$a\ f\ b$	(як для всіх відношень)
$b = f(a)$	(суто функціональне позначення)

До функцій застосовні (і навіть природніші, ніж для решти відношень) введені на стор. 59 поняття області визначення $D(f) = \{a \mid \exists b (f(a) = b)\}$ та області значень $E(f) = \{b \mid \exists a (f(a) = b)\}$.

Функції, що є відношеннями в $A \times B$, називають також *функції* ***з*** A ***у*** B, позначається $f : A \to B$. (А функції, що є відношеннями в $A_1 \times A_2 \times \ldots \times A_n \times B$, називають також *функції* ***з*** $A_1 \times A_2 \times \ldots \times A_n$ ***у*** B, позначається $f : A_1 \times A_2 \times \ldots \times A_n \to B$.)

(При цьому, запис $f : A \to B$ *не* означає *ні* $D(f) = A$, *ні* $E(f) = B$. Наприклад, для функції $f_1(x) = \sqrt{25 - x^2}$ область визначення $D(f_1) = [-5; 5] \subset \mathbb{R}$, область значень $E(f_1) = [0; 5] \subset \mathbb{R}$, але це все ж функція вигляду $\mathbb{R} \to \mathbb{R}$.)

Якщо для функції $f : A \to B$ виконується $D(f) = A$ (тобто вона може бути застосована до всіх можливих значень з A), така функція називається *всюди визначеною*, або *тотальною*.

(Будь-яку унарну функцію можна штучно (искусственно) оголосити тотальною. Наприклад, для $f_1(x) = \sqrt{25 - x^2}$ можна заднім числом заявити, ніби ми і не хотіли розглядати її як $\mathbb{R} \to \mathbb{R}$, а лише як $f_1 : [-5; 5] \to [0; 5]$, і тоді f_1 всюди визначена. Іноді навіть вводять систему означень, де не тотальні функції взагалі не вважаються функціями. Але такий прийом можна застосувати далеко не до всіх багатоарних функцій. Наприклад, для $f_2 : \mathbb{R}^2 \to \mathbb{R}$, $f_2(x_1, x_2) = (|x_1| + |x_2| + 1) \,/\, \sqrt{x_1^2 + x_2^2}$ область визначення $D(f_2) = \mathbb{R}^2 \setminus \{(0;0)\}$, що не можна виразити як результат декартового добутку.)

[24] див. також примітку 11 на стор. 29

Сюр'єкція Якщо для функції $f : A \to B$ виконується додаткова умова $E(f) = B$ (тобто f набуває *всіх* можливих значень з B), вона називається *сюр'єктивною функцією*, або *сюр'єкцією*, або *функцією з A **на** B* (рос. «*сюръективная функция*», «*сюръекция*», *функция из A **на** B*, англ. «*surjective function*», «*surjection*», *function from A **onto** B*).

(Наприклад, функція $f_3(x) = x^2$ (якщо розглядати її як $\mathbb{R} \to \mathbb{R}$) не є сюр'єкцією, бо не набуває від'ємних значень; а $f_4(x) = x^3$ — сюр'єкція.)

Ін'єкція Функцію називають *ін'єктивною функцією*, або *ін'єкцією* (рос. «*инъективная функция*», «*инъекция*», англ. «*injective function*», «*injection*»), якщо $\forall a \forall b\big((a \in D(f)) \wedge (b \in D(f)) \wedge \wedge (a \neq b) \to (f(a) \neq f(b))\big)$, тобто «різним аргументам відповідає різне значення». Те са́ме можна записати також як $\forall a \forall b\big((f(a_1) = f(a_2)) \to (a_1 = a_2)\big)$, тобто «однакові значення можуть досягатися *лише* на однакових аргументах»).

(Наприклад, функція $f_3(x) = x^2$ не є ін'єкцією, бо набуває значення 4 і при $x = 2$, і при $x = -2$; а $f_4(x) = x^3$ — ін'єкція, бо може набути значення y_i *лише* при $x_i = \sqrt[3]{y_i}$.)

Бієкція Функція називається *бієктивною функцією*, або *бієкцією* (рос. «*биективная функция*», «*биекция*», англ. «*bijective function*», «*bijection*»), якщо вона ін'єктивна і сюр'єктивна одночасно.

Обернена функція Унарні функції є бінарними відношеннями, тому до них можна застосувати операцію «обернене відношення» (формула (15) зі стор. 59). Очевидно, що відношення, обернене до відношення-функції, не завжди є функцією: наприклад, вищезгадане відношення-функція $\{(1, a),\ (2, a),\ (3, a),\ (4, b)\}$ перетворюється у $\{(a, 1),\ (a, 2),\ (a, 3),\ (b, 4)\}$, тобто відношення, але не функцію. Легко переконатися, що результат застосування до функції операції «обернене відношення» є функцією тоді й тільки тоді, коли початкова функція є ін'єкцією. Якщо ця умова виконується, то результат називають *обернена функція* і позначають "f^{-1}": $x = f^{-1}(y) \overset{\text{def}}{\iff} y = f(x)$.

Суперпозиція функцій Унарні функції є бінарними відношеннями, тому до них можна застосувати операцію «композиція відношень» зі стор. 59:

$c = (f_1 \circ f_2)(a) \ \Leftrightarrow$	перейшли від функціонального позначення до позначення «пара належить відношенню як підмножині декартового добутку»
$\Leftrightarrow\ (a, c) \in (f_1 \circ f_2) \ \Leftrightarrow$	означення "$\circ$" (формула (16) зі стор. 59)
$\Leftrightarrow\ \exists b\big((a, b) \in f_1 \wedge (b, c) \in f_2\big) \ \Leftrightarrow$	перейшли від позначень «пара належить відношенню як підмножині декартового добутку» до функціональних позначень
$\Leftrightarrow\ \exists b\big(b = f_1(a) \wedge c = f_2(b)\big) \ \Leftrightarrow$	використали відомий з інший галузей математики запис функції від функції, врахувавши, що в якості b все одно може бути *лише* $f_1(a)$ (*якщо* $f_1(a) \in D(f_2)$, тобто від $f_1(a)$ можна бра́ти f_2)
$\Leftrightarrow\ c = f_2(f_1(a))$.	

Таким чином, *суперпозиція* (вона ж *композиція*)[25] унарних функцій по смислу являє собою $y = (f_1 \circ f_2)(x) \overset{\text{def}}{\iff} y = f_2(f_1(x))$.

2.8 Рівнопотужні та не рівнопотужні множини (огляд)

Множи́ни A та B називають *рівнопотужними* (рос. «*равномощные*», англ. «*equipotent*»), якщо між ними можна встановити бієкцію.

(Наприклад, множина латинських літер рівнопотужна $\{1, 2, \ldots, 26\}$, бо між ними є бієкція, що встановлюється порядком латинського алфавіту.)

Абсолютно очевидно, що скінче́нні множи́ни рівнопотужні тоді й тільки тоді, коли містять однакову кількість елементів.

Більш того, для скінче́нних множин має місце т. зв. *принцип Дирихле*. У «популярній» формі він звучить так: якщо потрібно розсадити $n + 1$ кроликів по n кліткам, то, як би їх не розсаджували, гарантовано знайдеться (хоча б одна) клітка, у яку потраплять (щонайменше) два кролики.

(«*Хоча б* одна клітка» і «*щонайменше* два кролики», бо розглядаються *всі* «розподіли» кроликів по кліткам, включно з тими, де майже всіх кроликів садовлять в одну клітку, а більшість кліток порожні. Звісно, це спостереження очевидне. Але воно носить ім'я Дирихле, бо він вперше застосував його у строгих математичних доведеннях.

У англомовній літературі цей самий принцип частіше називають «*pigeonhole principle*», не згадуючи Дирихле безпосередньо. Слово «pigeon» означає «голуб» (птах), бо в західній традиції прийнято формулювати його не про кроликів, а про голубів.)

[25] тобто, і для унарних функцій, і для бінарних відношень в принципі можна вживати будь-який з цих термінів, але для відношень частіше кажуть «композиція», а для функцій «суперпозиція»

Але ми недаремно щоразу приговорювали «для скінче́нних». Виявляється, для нескінче́нних множин усе *зовсім не так!*

Наприклад, розглянемо множину всіх натуральних чисел ($\mathbb{N}$) та множину всіх невід'ємних па́рних чисел $E = \{2m \mid m \in \mathbb{Z}^+\}$. І спробуємо з'ясувати, яка з них «більша за розміром».

Ніби й є певні підстави вважати, що $\mathbb{N}$ «більша», бо $\mathbb{N} \setminus E = \{1, 3, 5, \ldots\} = \{2n-1 \mid n \in \mathbb{N}\}$ значно більше за $E \setminus \mathbb{N} = \{0\}$.

Але, з іншого боку, E та $\mathbb{N}$ рівнопотужні, бо існує бієкція $0 \leftrightarrow 1$, $2 \leftrightarrow 2$, $4 \leftrightarrow 3$, $6 \leftrightarrow 4$, $8 \leftrightarrow 5$, $\ldots$, $2 \cdot (i-1) \leftrightarrow i$, $\ldots$

І загальноприйнятою є якраз точка зору, що раз множи́ни $\mathbb{N}$ і E рівнопотужні, то вони у деякому смислі «однакові за розміром»!

Аналогічно можна встановити бієкцію і між множинами $\mathbb{Z}^+$ і $\mathbb{Z}$: $0 \leftrightarrow 0$, $1 \leftrightarrow +1$, $2 \leftrightarrow -1$, $3 \leftrightarrow +2$, $4 \leftrightarrow -2$, $\ldots$, $(2i-1) \leftrightarrow +i$, $2i \leftrightarrow -i$, $\ldots$ Отже, одночасно виконуються i твердження «$\mathbb{Z}^+$ — *власна* підмножина $\mathbb{Z}$», i твердження «$\mathbb{Z}^+$ рівнопотужна $\mathbb{Z}$».

Ще можна встановити бієкцію між інтервалом дійсних чисел $(0;1)$ та множиною всіх дійсних чисел $\mathbb{R}$. Найлегше це зробити через суперпозицію двох бієкцій: спочатку встановити бієкцію $x \leftrightarrow \leftrightarrow (x - \frac{1}{2}) \cdot \pi$ між інтервалом $(0;1)$ та інтервалом $(-\frac{\pi}{2}; \frac{\pi}{2})$, потім бієкцію $x \leftrightarrow \operatorname{tg} x$ між інтервалом $(-\frac{\pi}{2}; \frac{\pi}{2})$ та $\mathbb{R}$. Знов одночасно виконуються i твердження «інтервал $(0;1)$ — *власна* підмножина $\mathbb{R}$», i твердження «інтервал $(0;1)$ рівнопотужний $\mathbb{R}$».

Власне, це навіть вважають характеристичною ознакою нескінче́нної множи́ни: множина нескінче́нна тоді й тільки тоді, коли можна встановити бієкцію між усією множиною та деякою її власною підмножиною.

Після ознайомлення з такими фактами може спа́сти на думку, ніби всі нескінче́нні множи́ни рівнопотужні. *Але це не так.*

Теорема Кантора *Інтервал дійсних чисел* $(0;1)$ ***не*** *рівнопотужний* $\mathbb{N}$.

Доведення цієї теореми спирається на т. зв. *діагональний метод* (рос. «*диагональный метод*», англ. «*diagonal argument*»).

Корені діагонального методу сягають Давньої Греції. Там був острів Крит, і критський філософ Епіменід якось заявив: *«Усі критяни брехуни»*. Він це сказав, сам будучи критянином. Значить, сам брехун... Традиційно це називається «парадоксом Епіменіда». Хоча насправді *цей* «парадокс» не є парадоксом: якщо припустити, що сам Епіменід — брехун, а частина решти критян — чесні люди, то ніяких внутрішніх протиріч не виникає.

Час ішов, знання людства зростали, і Сервантес у «Дон Кіхоті» зумів побудувати справжній парадокс. При одному мості поставили сторо́жу, яка допитувала кожного перехожого, куди він іде, і перевіряла правдивість його слів. Якщо перехожий казав правду, йому дозволяли йти далі, а якщо виявлялося, що він бреше, його вішали на шибениці. Одного разу з'явився перехожий, що заявив: прийшов єдино заради того, щоб його повісили. Сторожа не змогла вирішити, що ж робити. Адже якщо пропустити цього перехожого, то виявиться, що він брехав, і його треба було повісити; а якщо повісити, то він казав правду, і його слід було пропустити.)

Доведення теореми Кантора використовує протиріччя на основі цієї самої ідеї, але при математично строгих припущеннях та міркуваннях.

Припустимо, ніби інтервал $(0;1)$ рівнопотужний $\mathbb{N}$, тобто між цими множинами існує (якась) бієкція. Зафіксуємо цю бієкцію, розмістивши всі дійсні чи́сла інтервалу $(0;1)$ послідовно рядок за рядком: спочатку дійсне число, якому за вибраною бієкцією відповідає натуральне число 1, потім дійсне число, якому відповідає 2, і т. д.

Як відомо, кожне дійсне число з інтервалу $(0;1)$ задається нескінче́нною послідовністю цифр десяткового запису (кількість десятих, кількість сотих, кількість тисячних, і т. д.; у випадках, коли дійсне число записується скінче́нною кількістю десяткових цифр, допишемо в кінці нескінче́нну кількість нулів).[26]

Можна вважати, що отримали «таблицю розміром $\infty \times \infty$», елементами якої є десяткові цифри. Позначимо j-ий стовпчик (j-ту цифру) i-го рядка (числа́) як $c_{i,j}$.

[26] Для повної строгості варто згадати, що слід пропускати десяткові записи з дев'ятками у періоді (бо, наприклад, $0{,}1199999999\ldots$ (з *нескінче́нною* кількістю дев'яток) — те са́ме, що $0{,}1200000000\ldots = 0{,}12$); тож очевидна бієкція існує *не* між (усіма дійсними числами інтервалу $(0;1)$) та (усіма нескінче́нними послідовностями десяткових цифр), *а* між (усіма дійсними числами інтервалу $(0;1)$) та (усіма нескінче́нними послідовностями десяткових цифр, які *не* закінчуються на дев'ятку в періоді).

Втім, це спостереження, хоча й потрібне для повної строгості доведення, не є вирішальним.

А тепер побудуємо нескінче́нну послідовність γ десяткових цифр за правилом

$$\gamma_k = \begin{cases} 1, & \text{якщо } c_{k,k} \in \{0,2,3,4,5,6,7,8,9\}, \\ 2, & \text{якщо } c_{k,k} = 1. \end{cases}$$

Послідовність $\gamma_1, \gamma_2, \ldots$ задає десятковий запис числа́ $\overline{0{,}\gamma_1\gamma_2\ldots}$ з інтервала $(0;1)$. І цього числа́ у цій таблиці нема. (Справді, кожне число записане під певним номером; отже, якщо $\overline{0{,}\gamma_1\gamma_2\ldots}$ є у таблиці, то воно записане під деяким номером k^*; але цього не може бути, бо за побудовою $\gamma_{k^*} \neq c_{k^*,k^*}$, отже k^*-ий рядок містить якесь інше число.)

Отже, така «бієкція» насправді не бієкція (бо не співставила ніякого номера рядка числу $\overline{0{,}\gamma_1\gamma_2\ldots}$). Усі ці міркування застосовні до будь-якої спроби встановити бієкцію. Отже, побудувати бієкцію неможливо, тобто інтервал $(0;1)$ не рівнопотужний $\mathbb{N}$. ■

Згадавши побудовану вище бієкцію між інтервалом $(0;1)$ та $\mathbb{R}$, отримуємо очевидний наслідок «$\mathbb{R}$ не рівнопотужна $\mathbb{N}$». Разом узяті факти «$\mathbb{N} \subset \mathbb{R}$» та «$\mathbb{R}$ не рівнопотужна $\mathbb{N}$» природньо спробувати поєднати у «потужність $\mathbb{R}$ строго більша за потужність $\mathbb{N}$».

Поняття «більшої потужності» справді має математичний смисл, але в загальному випадку його слід ввести трохи інакше: потужність множи́ни A строго більша за потужність множи́ни B, коли (самі́ A і B не рівнопотужні) $\wedge$ (B рівнопотужна деякій власній підмножині A).

Виявляється, поняття «більшої потужності» до того ж є відношенням строгого лінійного порядку; але і строге доведення лінійності, і строге доведення антисиметричності *дуже* складні.

Очевидно, найменшою є потужність порожньої множи́ни, потім ідуть потужності 1-, 2-, 3-, …, i-, … -елементних множин (для всіх $i \in \mathbb{N}$), а потім є ще нескінче́нна кількість різних значень потужності нескінче́нних множин.

(Те, що їх справді нескінче́нна кількість, випливає хоча б з теореми «Булеа́н будь-якої множи́ни має потужність, строго більшу за потужність само́ї множи́ни»; доводиться ця теорема аналогічно, за допомогою діагонального методу.)

«Найменшими» серед нескінче́нних множин є множи́ни, рівнопотужні $\mathbb{N}$. Їх називають *зліче́нні* (рос. «*счётные*», англ. «*countable*». Враховуючи все вищесказане, очевидно, що зліче́нними є, зокрема: $\mathbb{Z}$; множина па́рних чисел; множина непа́рних чисел. Можна також довести, що зліче́нною є множина простих чисел.

Множи́ни, рівнопотужні $\mathbb{R}$, називають *континуа́льними*, або *конти́нуум-множинами* (*continuum*). Континуальними є, зокрема, множина дійсних чисел довільного інтервала $(a;b)$ (при $a<b$), $\mathbb{C}$ (множина комплексних чисел).

Питання, чи континуум іде у згаданому порядку потужностей безпосередньо після зліченності, чи існують «проміжні» множи́ни (потужність яких строго більша зліче́нних і строго менша континуальних) дуже складне́. Сам Ґ. Кантор висунув *континуум-гіпотезу*, що «проміжних» множин не буває. На сучасному етапі розвитку науки вважається, що цю гіпотезу неможливо ні довести, ні спростувати, її можна лише або приймати за ще одну аксіому, або не приймати.

На перший погляд, зліченність відповідає дискретності, а континуальність — неперервності. Але це неправда. І не лише тому, що континуальну множину легко зробити розривчастою, а ще й тому, що множина $\mathbb{Q}$ (раціональних чисел) виявляється неперервною зліче́нною.

2.9 Завдання до розділу 2

1. Нехай $U = \{0,1,2,3,4,5,6,7,8,9\}$, $A = \{1,2,4\}$, $B = \{2,3,7\}$, $C = \{2,3,8\}$. Для кожного з трьох пунктів

   (а) $(A \cup B) \setminus C$; (б) $(A \setminus C) \cup B$; (в) $(A \cup B \cup C)'$

   виконати такі два завдання: (1) "обчислити" (знайти значення при вказаних A, B, C); (2) побудувати діаграми Венна, вважаючи A, B, C довільними (ігноруючи наведені значення).
   (Зверніть увагу, що, для вказаних у цьому завданні A, B, C, значення $(A \cup B) \setminus C$ виявляється *не* рівним значенню $(A \setminus C) \cup B$. Це одна з причин, які тривалий час гальмували становлення теорії множин. Математикам здавалося, що раз для чисел $(a+b)-c = (a-c)+b$, то треба і для інших сутностей винаходити операції з такими самими властивостями; зручність розвитку інших теорій, в яких операції мають інші властивості, була достатньо усвідомлена лише у XIX ст.)

2. Нехай $A=\{1,2,4\}$, $B=\{2,3,7\}$. "Обчислити" (знайти значення):

   (а) $A\cap B$; (б) $A\cup B$; (в) $A\setminus B$; (г) $B\setminus A$; (д) $A\div B$.

3. З'ясувати, які твердження правильні, а які — ні:

   (а) $2\in\{2\}$; (ж) $2,3\in\{2,3,5\}$; (п) $\{1,2,3\}=\{1,3,2\}$;
   (б) $2=\{2\}$; (и) $\{2,3\}\subseteq\{2,3,5\}$; (р) $\varnothing\subset\{1,3,7\}$;
   (в) $2\subseteq\{2\}$; (к) $\{2,3\}=\{2,3,5\}$; (с) $\varnothing\in\{1,3,7\}$;
   (г) $\{2\}\subseteq\{2\}$; (л) $\{2,3\}\in\{2,3,5\}$; (т) $\varnothing\subseteq\varnothing$;
   (д) $\{2\}=\{2\}$; (м) $\{2,3\}\subset\{2,3,5\}$; (у) $\varnothing\in\varnothing$;
   (е) $\{2\}\subset\{2\}$; (н) $\{2,3,5\}\subseteq\{2,3\}$; (ф) $\varnothing\subset\varnothing$.

4. Чи виконується твердження "$(A\cap B)\in C$", де $A=\{1,2,4\}$, $B=\{2,3,7\}$, $C=\{2,3,8\}$?
5. Найчастіше, елементи конкретної множи́ни однотипні (всі — чи́сла, або всі — слова́, або всі — люди, тощо). Але буває й інакше. Наприклад, деякі відзнаки (ордени, премії, ...) присуджують і окремим людям, і колективам (множинам людей). Тому іноді розглядають множи́ни, елементами яких можуть бути і «звичайні» елементи, і множи́ни. З'ясувати, які твердження правильні, а які — ні:

   (а) $\{2,3\}\in\{2,3,5\}$; (в) $3\in\{\{1,2\},3\}$; (д) $\{1,2,3\}=\{\{1,2,3\}\}$;
   (б) $1\in\{\{1,2\},3\}$; (г) $\{1,2,3\}=\{\{1,2\},3\}$; (е) $\{1,2,3\}\in\{\{1,2,3\}\}$.

6. Продовжимо розгляд множин, елементами яких можуть бути і «звичайні» елементи, і множи́ни. Нехай $A=\{\{a,b,c\},\ \{a,d,e\},\ \{c,d\},\ c,\ d,\ e\}$. З'ясувати, які твердження правильні, а які — ні:

   (а) $c\in A$; (г) $\{c,d\}\in A$; (ж) $\{c,d,e\}\subseteq A$;
   (б) $\{c\}\in A$; (д) $\{c,d\}\subseteq A$; (и) $\{a\}\in A$;
   (в) $\{c\}\subseteq A$; (е) $\{c,d,e\}\in A$; (к) $\{a\}\subseteq A$.

7. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Знайти значення виразу, що містить операції над множинами.

   (1) $\Big(\big(\{a,b,c\}\cup\{a,c,f\}\big)\setminus\{c,g\}\Big)\times\Big(\big(\{0,2,5\}\div\{1,2,4\}\big)\cap\{2,3,5\}\Big)$;
   (2) $\Big(\{c,g,r\}\setminus\big(\{a,b,c\}\cap\{a,c,f\}\big)\Big)\times\Big(\big(\{0,2,5\}\cap\{1,2,4\}\big)\div\{2,3,5\}\Big)$;
   (3) $\Big(\big(\{a,c,f\}\setminus\{a,b,c\}\big)\cap\{c,f\}\Big)\times\Big(\big(\{2,3,8\}\div\{1,2,5\}\big)\setminus\{2,5,7\}\Big)$;
   (4) $\Big(\{0,2,5\}\div\big(\{1,2,4\}\cap\{2,3,5\}\big)\Big)\times\Big(\{a,b,c\}\cup\big(\{a,c,f\}\setminus\{c,g\}\big)\Big)$;
   (5) $\Big(\{2,3,5\}\setminus\big(\{1,2,4\}\div\{2,3,5\}\big)\Big)\times\Big(\{a,b,c\}\cup\big(\{a,c,f\}\cap\{c,g\}\big)\Big)$;
   (6) $\Big(\{2,3,5\}\div\big(\{1,2,4\}\setminus\{0,3,5\}\big)\Big)\times\Big(\big(\{a,b,c\}\cup\{a,c,f\}\big)\cap\{c,g\}\Big)$;
   (7) $\Big(\{2,3,8\}\div\big(\{1,2,5\}\setminus\{2,5,7\}\big)\Big)\times\Big(\{a,c,f\}\setminus\big(\{a,b,c\}\cap\{c,f\}\big)\Big)$.

8. Проілюструвати тотожність $A\cup(B\cap C)=(A\cup B)\cap(A\cup C)$ за допомогою діаграм Венна та за допомогою перетворення характеристичних предикатів з подальшою побудовою таблиць істинності. (Доводити її аналітично не варто, бо це і є одна зі стандартних тотожностей.)
9. Довести тотожність $(A\cup B)\setminus(B\setminus A)=A$ через аналітичні перетворення (без переходу до предикатів).
10. Довести тотожність $A\setminus(B\cap C)=(A\setminus B)\cup(A\setminus C)$ всіма способами.
11. Довести аналітично тотожність $(A\cap B)\cup(B\setminus A)=B$.
12. Чи є правильними співвідношення:

    (а) $(A\div B)\subseteq(A\cup B)$? (б) $(A\div B)\subset(A\cup B)$?

13. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Проімітувати покроково виконання дій над множинами на основі злиття.
    (1) $\{2,3,5,7,12,17,25\} \div \{1,3,5,11,17,19,22\}$;
    (2) $\{1,3,5,6,11,17,19,22,24\} \setminus \{2,3,5,7,12,17,25,27\}$;
    (3) $\{1,3,5,6,11,17,19,22,24\} \cap \{2,3,5,7,11,12,17\}$;
    (4) $\{4,6,13,14,15,17,18\} \div \{1,4,11,17,19,27\}$;
    (5) $\{2,3,6,8,9,10,13,25\} \cap \{5,6,8,10,13,19,29\}$;
    (6) $\{0,4,10,13,20,23,24\} \setminus \{5,10,12,20,22,28,29\}$;
    (7) $\{2,3,8,15,21,25,27\} \div \{0,2,7,10,12,15,24\}$.
14. (а) Перелічити всі елементи $\{1,2,3\} \times \{\alpha,\beta\}$.
    (б) Перелічити всі елементи $\{1,2\} \times \{a,b\} \times \{2,5\}$.
    (в) Зобразити на площині $[2;5] \times [3;4)$.
    (г) Зобразити на площині $[2;5] \cap (4;7) \times [3;4) \cup (1;2)$.
15. Довести включення $(A \setminus B) \times (C \setminus B) \subseteq (A \times C) \setminus (B \times B)$ за допомогою перетворення характеристичних предикатів з подальшою побудовою таблиць істинності.
16. Записати «відношення на $\mathbb{N}$ "$x+y \leqslant 5$"» у вигляді явного переліку пар.
17. Нехай $R \subseteq A \times B$, $P \subseteq B \times C$, $A=\{x,y,z\}$, $B=\{1,2,3\}$, $C=\{\alpha,\beta,\gamma\}$, $R=\{(x,1),\ (x,2),\ (y,1),\ (z,1),\ (z,2)\}$, $P=\{(1,\gamma),\ (2,\alpha),\ (2,\beta),\ (3,\alpha),\ (3,\beta),\ (3,\gamma)\}$. Побудувати $R \circ P$, $P^{-1} \circ R^{-1}$, $P \circ P^{-1}$.
18. Нехай задані відношення R_1: $x\, R_1\, y \Leftrightarrow$ x та y — подружжя, та R_2: $x\, R_2\, y \Leftrightarrow$ x є мамою y. Виразити словами смисл відношень $R_2 \circ R_2$, $R_1 \circ R_2$, $R_2 \circ R_1$.
19. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Знайти значення таких чотирьох композицій: $R_1 \circ R_1^{-1}$, $R_1^{-1} \circ R_1$, $R_1 \circ R_2$, $R_2^{-1} \circ R_1^{-1}$. Для яких-небудь (на вибір студента) 2–3 з пар, що належать якомусь із відношень-відповідей, дати пояснення, чому вони туди належать.

    (1) $R_1 = \{(b,\tau),(c,\pi),(c,\rho),(c,\tau),(d,\rho)\}$;
        $R_2 = \{(\pi,4),(\rho,2),(\rho,3),(\rho,4),(\tau,1)\}$.
    (2) $R_1 = \{(b,\pi),(b,\rho),(c,\pi),(c,\tau),(d,\tau)\}$;
        $R_2 = \{(\pi,1),(\rho,3),(\tau,1),(\tau,2),(\tau,3)\}$.
    (3) $R_1 = \{(a,\pi),(a,\tau),(b,\pi),(b,\rho),(d,\pi)\}$;
        $R_2 = \{(\pi,2),(\pi,4),(\rho,2),(\rho,3),(\tau,4)\}$.
    (4) $R_1 = \{(a,\pi),(a,\tau),(c,\rho),(d,\rho),(d,\tau)\}$;
        $R_2 = \{(\pi,1),(\rho,2),(\rho,4),(\tau,2),(\tau,4)\}$.
    (5) $R_1 = \{(a,\pi),(a,\tau),(b,\rho),(b,\tau),(c,\tau)\}$;
        $R_2 = \{(\pi,1),(\pi,3),(\pi,4),(\rho,1),(\tau,2)\}$.
    (6) $R_1 = \{(a,\rho),(b,\pi),(b,\tau),(d,\rho),(d,\tau)\}$;
        $R_2 = \{(\pi,1),(\pi,3),(\rho,3),(\rho,4),(\tau,1)\}$.
    (7) $R_1 = \{(a,\tau),(b,\rho),(b,\tau),(c,\rho),(d,\pi)\}$;
        $R_2 = \{(\rho,1),(\rho,3),(\rho,4),(\tau,3),(\tau,4)\}$.
    (8) $R_1 = \{(a,\rho),(a,\tau),(b,\pi),(b,\tau),(c,\rho)\}$;
        $R_2 = \{(\pi,1),(\rho,1),(\rho,3),(\rho,4),(\tau,4)\}$.
    (9) $R_1 = \{(a,\tau),(b,\rho),(b,\tau),(c,\pi),(c,\rho)\}$;
        $R_2 = \{(\pi,2),(\rho,2),(\rho,3),(\rho,4),(\tau,3)\}$.
    (10) $R_1 = \{(a,\rho),(b,\pi),(b,\rho),(b,\tau),(c,\tau)\}$;
        $R_2 = \{(\pi,3),(\rho,1),(\rho,4),(\tau,1),(\tau,3)\}$.
    (11) $R_1 = \{(a,\pi),(a,\rho),(b,\pi),(b,\tau),(d,\pi)\}$;
        $R_2 = \{(\pi,3),(\pi,4),(\rho,3),(\tau,1),(\tau,2)\}$.
    (12) $R_1 = \{(a,\pi),(a,\rho),(c,\tau),(d,\pi),(d,\tau)\}$;
        $R_2 = \{(\pi,1),(\pi,3),(\pi,4),(\rho,2),(\tau,2)\}$.
    (13) $R_1 = \{(a,\pi),(a,\rho),(c,\pi),(c,\tau),(d,\rho)\}$;
        $R_2 = \{(\pi,1),(\pi,2),(\pi,3),(\rho,2),(\tau,2)\}$.
    (14) $R_1 = \{(a,\pi),(a,\rho),(a,\tau),(b,\rho),(d,\tau)\}$;
        $R_2 = \{(\pi,1),(\rho,4),(\tau,1),(\tau,2),(\tau,3)\}$.

20. Нехай на множині людей (як нині живих, так і померлих, але не ще не народжених) задані відношення I, $R_{\text{СМ}}$, $R_{\text{СБ}}$, $R_{\text{ДМ}}$ та $R_{\text{ДБ}}$: $x\, I\, y \Leftrightarrow x = y$, тобто це одна й та ж людина; $x R_{\text{СМ}} y \Leftrightarrow$ x — син y, а y — мати x; $x R_{\text{СБ}} y \Leftrightarrow$ x — син y, а y — батько (отец) x; $x R_{\text{ДМ}} y \Leftrightarrow$ x — донька y, а y — мати x; $x R_{\text{ДБ}} y \Leftrightarrow$ x — донька y, а y — батько (отец) x. Пояснити словами смисл:
    (а) $D(R_{\text{СМ}})$; (б) $E(R_{\text{СМ}})$; (в) $D({R_{\text{СМ}}}^{-1})$;
    (г) $\big((R_{\text{СМ}} \circ {R_{\text{СМ}}}^{-1}) \cap (R_{\text{СБ}} \circ {R_{\text{СБ}}}^{-1})\big) \setminus I$;
    (д) $\big((R_{\text{СМ}} \circ {R_{\text{СМ}}}^{-1}) \cup (R_{\text{СБ}} \circ {R_{\text{СБ}}}^{-1})\big) \setminus I$;
    (е) $\big((R_{\text{СМ}} \circ {R_{\text{СМ}}}^{-1}) \cup (R_{\text{СБ}} \circ {R_{\text{СБ}}}^{-1})\big)$.
    Повинно бути чітко сформульовано, *який* родинний зв'язок описується кожним з виразів.

21. Дослідити належність / не належність бінарного відношення

    (а) $x\,R\,y \Leftrightarrow x-y \leqslant 100$ (на $\mathbb{R}$);
    (б) $x\,R\,y \Leftrightarrow x-y \geqslant 100$ (на $\mathbb{R}$);
    (в) $x\,R\,y \Leftrightarrow |x-y| \leqslant 100$ (на $\mathbb{R}$);
    (г) $x\,R\,y \Leftrightarrow |x-y| \geqslant 100$ (на $\mathbb{R}$);
    (д) $x\,R\,y \Leftrightarrow x+y \leqslant 100$ (на $\mathbb{R}$);
    (е) $x\,R\,y \Leftrightarrow x \leqslant y+3$ (на $\mathbb{R}$);
    (ж) $x\,R\,y \Leftrightarrow (x+y) \vdots 12$ (на $\mathbb{Z}$) (де "$\vdots$" — кратне);
    (и) $x\,R\,y \Leftrightarrow 60 \vdots (x+y)$ (на $\mathbb{Z}$);
    (к) $x\,R\,y \Leftrightarrow (xy) \vdots 2$ (на $\mathbb{Z}$);
    (л) $x\,R\,y \Leftrightarrow (xy) \,\overline{\vdots}\, 2$ (на $\mathbb{Z}$) (де "$\overline{\vdots}$" — *не* кратне).

    до стандартних класів (рефлексивість, іррефлексивність, симетричність, антисиметричність, транзитивність, повнота).
22. З'ясувати належність/неналежність до стандартних класів (рефлексивість, іррефлексивність, симетричність, антисиметричність, транзитивність, повнота) відношення, заданого на множині $\{a,b,c,d,e\}$, повний перелік елементів відношення — $\{(a,a), (a,b), (a,d), (b,a), (b,b), (d,a), (d,d), (e,e)\}$.
23. Довести, що відношення є відношенням еквівалентності, знайти класи еквівалентності та фактор-множину (в т. ч. вказати, скільки всього є класів еквівалентності); вказати класи еквівалентності для наведених елементів і чи є серед них однакові (якщо є, які?).

    (а) $R \subseteq \mathbb{Z}^2$, $x\,R\,y \Leftrightarrow (x+y) \vdots 2$; знайти $[1]_R$, $[2]_R$, $[3]_R$;
    (б) $R \subseteq \mathbb{Z}^2$, $x\,R\,y \Leftrightarrow (x^2-y^2) \vdots 5$; знайти $[2]_R$, $[3]_R$, $[4]_R$, $[5]_R$;
    (в) $R \subseteq \mathbb{R}^2$, $x\,R\,y \Leftrightarrow (x-y) \in \mathbb{Z}$; знайти $[2]_R$, $[2{,}5]_R$, $[3]_R$, $[\sqrt{2}]_R$;
    (г) $R \subseteq \mathbb{N}^2$, $x\,R\,y \Leftrightarrow \exists m_{m\in\mathbb{Z}}(\frac{x}{y} = 2^m)$ (інакше кажучи, $\frac{x}{y} \in \{\ldots, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 8, \ldots\}$); знайти $[2]_R$, $[3]_R$, $[4]_R$, $[5]_R$.
24. Довести, що $x \vdots y$ (на $\mathbb{N}$) є відношенням порядку.
25. Побудувати діаграму Гассе, визначити максимальні, мінімальні, найбільші, найменші елементи для відношення $x \vdots y$, звуженого на $\{2,3,6,10,12,15,25,36,60\}$.
26. Додати до розв'язку попереднього завдання елемент 0.
    Підказка. Ретельно перевірте, що Ви трактуєте поняття «менше», «більше», «мінімальний», «максимальний», «найменший», «найбільший» са́ме згідно з їх означеннями з розд. 2.6.3 та формулюванням попереднього завдання, а не за звичкою.
27. Довести, що відношення є відношенням порядку (вважаючи, що відношення задане на $\mathbb{N}$); вказати, яким са́ме відношенням порядку (строгим/нестрогим/ні строгим, ні нестрогим; лінійним/нелінійним); для звуження відношення на скінче́нну множину B, побудувати діаграму Гассе, знайти мінімальні та максимальні елементи

    (а) $x\,R\,y \Leftrightarrow (y-2x) \in \mathbb{N}$, $B = \{1, 2, 3, 5, 7, 12, 25, 36, 60\}$.
    (б) $x\,R\,y \Leftrightarrow \dfrac{y}{x}$ є непа́рним натуральним, $B = \{1, 2, 3, 5, 6, 10, 12, 15, 36, 60\}$.
    (в) $x\,R\,y \Leftrightarrow \dfrac{y}{x}$ є па́рним натуральним, $B = \{1, 2, 3, 5, 6, 10, 12, 15, 36, 60\}$.
28. Для кожного з відношень

    (а) $A = \{1,2,3\}$, $R = \{(1,1),(2,2),(3,3),(1,3),(3,1)\}$;
    (б) $A = \{1,2,3\}$, $R = \{(1,1),(2,2),(3,3),(1,2),(2,3)\}$;
    (в) $A = \{1,2,3\}$, $R = \{(1,1),(2,2),(3,3),(1,2),(2,3),(1,3)\}$;
    (г) $A = \{1,2,3\}$, $R = \{(1,1),(2,2),(3,3)\}$;
    (д) $A = \{1,2,3,4\}$, $R = \{(1,1),(2,2),(3,3),(1,3),(3,2),(1,2),(4,2)\}$

    визначити, чи є воно відношенням еквівалентності та чи є відношенням порядку. Для відношень еквівалентності, знайти фактор-множи́ни. Для відношень порядку, побудувати діаграми Гассе та вказати максимальні, мінімальні, найбільші, найменші елементи.
29. Навести приклад відношень еквівалентності, об'єднання яких не є відношенням еквівалентності.
30. Навести приклад відношень порядку, об'єднання яких не є відношенням порядку.

31. Реалізувати функції:
    (а) `bool is_reflexive(const vvi &A);`
    (б) `bool is_irreflexive(const vvi &A);`
    (в) `bool is_symmetric(const vvi &A);`
    (г) `bool is_antisymmetric(const vvi &A);`
    (д) `bool is_transitive(const vvi &A);`
    (е) `bool is_linear(const vvi &A)`

    вважаючи, що вище дане означення `typedef vector<vector<int> > vvi;`, тобто тип `vvi` є реалізацією `vector`-ами двовимірного масиву.
    Ці функції, отримуючи як параметр матрицю бінарного відношення, повинні визначати (повертати), чи є воно рефлексивним/іррефлексивним/симетричним/антисиметричним/транзитивним/лінійним (повним). По тексту функцій мають бути коментарі-пояснення.
    Примітка. Див. зауваження на стор. 62 та приклад на стор. 64.
32. З'ясувати, чи є всюди визначеними, ін'єктивними, сюр'єктивними, бієктивними функції (які розглядаються як $\mathbb{R} \to \mathbb{R}$)
    (а) $y = x^3$;
    (б) $y = x^2$;
    (в) $y = \operatorname{tg} x$;
    (г) $y = x^3 + x^2$;
    (д) $y = 2^x$;
    (е) $y = \log_2 x$.

Додаткові завдання підвищеного рівня складності

1*. Реалізувати операції (написати програму мовою програмування, де операції реалізовані як підпрограми) $\cup$, $\cap$, $\div$, $\setminus$, $\in$, $\subseteq$, «ініціалізувати як $\varnothing$», «додати один елемент», при поданні множин як:
    (а) бітових векторів (універсум — невеликий перелічуваний діапазон);
    (б) впорядкованих послідовностей (універсум довільний лінійно впорядкований, наприклад «дійсні» (з рухомою комою) чи́сла), «вартість» "$\in$" — $O(\log n)$ (на основі бінарного пошуку), операцій "$\cup$", "$\cap$", "$\div$", "$\setminus$" та відношення "$\subseteq$" — $O(m+n)$ (на основі злиття). *Обов'язково* для пункту (b), щоб програма, що містить такі реалізації дій над множинами, успішно пройшла усі тести задачі A змагання № 90 «Деякі задачі дискретної математики» сайту `https://ejudge.ckipo.edu.ua`.
2*. З завд. 1 (стор. 75) можна, зокрема, зробити висновок, що «$(A \cup B) \setminus C = (A \setminus C) \cup B$» *не є* правильною тотожністю. Але ж хоча б іноді (хоча б для деяких множин) така рівність виконується. Тож *за яких умов* вона виконується? (Слід сформулювати *критерій*, тобто умову, одночасно і необхідну, й достатню.)
3*. Показати, що на $\mathbb{N}$ $\leqslant \circ \leqslant = \leqslant$, але $< \circ < \neq <$, тобто, композиція відношення $\leqslant$ (на $\mathbb{N}$) з самим собою дає знову це са́ме відношення $\leqslant$ (на $\mathbb{N}$), а от композиція відношення $<$ (на $\mathbb{N}$) з самим собою дає якесь інше відношення, а не те са́ме.
4*. Нехай $R = \{(x, y) \mid x^2 + y^2 = 1\}$ $(x, y \in \mathbb{R})$. Побудувати R^2, R^3, R^4 (навести і формули, і пояснення, і графіки на координатній площині).
5*. Нехай $R, P \subseteq \mathbb{R}^2$, $R = \{(x, y) \mid y = x + 1\}$, $P = \{(x, y) \mid y = |x|\}$. Побудувати $R \circ P$ та $P \circ R$ (навести і формули, і пояснення, і графіки на координатній площині).
6*. На стор. 60 стверджується: *«Степінь бінарного відношення має властивості $R^{n+m} = R^n \circ \circ R^m$ та $R^{n \cdot m} = (R^n)^m$ (при $n \in \mathbb{Z}^+ \wedge m \in \mathbb{Z}^+$). Але поширювати ці тотожності на усі цілі чи́сла (вважаючи, ніби обернене відношення R^{-1} є (-1)-им степенем) не можна.»*. Показати (шляхом наведення відповідного контрприкладу), що справді не можна.
7*. Довести (21)–(23) зі стор. 60, а також навести приклади, які показують: при $E(R_1) \neq D(R_2)$ можливо як $D(R_1 \circ R_2) = D(R_1)$, так і $D(R_1 \circ R_2) \subset D(R_1)$ (а також як $E(R_1 \circ R_2) = E(R_2)$, так і $E(R_1 \circ R_2) \subset E(R_2)$).
8*. Реалізувати функцію `vvi compose(vvi R1, vvi R2)`, яка за заданими відношеннями `R1`, `R2` (бінарними у широкому смислі) будує (і повертає як результат функції) композицію `R1` $\circ$ `R2`. Смисл `vvi` див. у завд. 31 (стор. 79).
9*. Довести, що якщо непорожнє відношення симетричне та транзитивне, воно не може бути іррефлексивним.
10*. Довести, що (одночасно) рефлексивним, симетричним та лінійним бінарним відношенням може бути лише увесь декартів квадрат.

11*. Довести, що перетин відношень еквівалентності обов'язково є відношенням еквівалентності.

12*. Довести, що перетин відношень порядку обов'язково є відношенням порядку.

13*. Побудувати рефлексивне замикання відношення $y = x + 1$ (на $\mathbb{N}$). Тобто, описати словами і/або формулою, яке відношення отримується внаслідок застосування рефлексивного замикання до відношення $y{=}x{+}1$.

14*. Побудувати симетричне замикання відношення $y = x + 1$ (на $\mathbb{N}$). Тобто, описати словами і/або формулою, яке відношення отримується внаслідок застосування симетричного замикання до відношення $y{=}x{+}1$.

15*. Побудувати транзитивне замикання відношення $y = x + 1$ (на $\mathbb{N}$). Тобто, описати словами і/або формулою, яке відношення отримується внаслідок застосування транзитивного замикання до відношення $y{=}x{+}1$.

16*. Показати, що рефлексивно-симетричне замикання будь-якого лінійного відношення являє собою увесь декартів квадрат.

17*. В обчислювальній геометрії на площині досить важливу роль відіграє т. зв. «косий добуток». Він, для вільних векторів $\vec{a}(a_x, a_y)$ та $\vec{b}(b_x, b_y)$, дорівнює $|\vec{a}| \cdot |\vec{b}| \cdot \sin\varphi$, де φ — тригонометричний (включаючи знак, додатний проти годинникової стрілки й від'ємний за стрілкою) кут найкоротшого повороту від $\vec{a}$ до $\vec{b}$; цей са́мий косий добуток може бути обчислений також як $a_x b_y - a_y b_x$. Тобто, обчисливши цю досить просту формулу, легко встановити, що якщо результат=0, то $\vec{a}$ та $\vec{b}$ колінеарні (співнапрямлені чи протинапрямлені), якщо додатний, то найкоротший поворот від $\vec{a}$ до $\vec{b}$ відбувається проти годинникової стрілки, а якщо від'ємний, то за стрілкою. У деяких джерелах стверджують, ніби це дає можливість використовувати косий добуток як компаратор[27], щоб відсортувати вектори за їхніми кутами нахилу (наприклад, проти годинникової стрілки). Чому, тим не менш, використовувати косий добуток в якості компаратора векторів за кутом нахилу — дуже, дуже погана ідея? За яких додаткових умов, накладених на порівнювані вектори, косий добуток все-таки можна надійно використовувати як компаратор?

Автору відомо, що це завдання містить багато фактажу, що виходить за рамки посібника. Тим цінніше, що пояснення суті проблеми майже повністю лежить у межах цього розділу.

18*. Провести формальне доведення еквівалентності означень іррефлексивності, симетричності, антисиметричності та повноти, даних у розд. 2.6.1 та даних у розд. 2.6.5.

19*. Реалізувати функцію `vvi reflexive_closure(vvi in_R)`, яка за заданим відношенням `in_R` будує (і повертає як результат функції) його рефлексивне замикання. Смисл `vvi` див. у завд. 31 (стор. 79). Вважати, що `in_R` гарантовано є бінарним у вузькому смислі.

20*. Реалізувати функцію `vvi symmetric_closure(vvi in_R)`, яка за заданим відношенням `in_R` будує (і повертає як результат функції) його симетричне замикання. Смисл `vvi` див. у завд. 31 (стор. 79). Вважати, що `in_R` гарантовано є бінарним у вузькому смислі.

21*. Нехай і R_1, і R_2 задаються умовою $x\,R_i\,y \Leftrightarrow |x - y| \leqslant 100$, але R_1 задане на $\mathbb{Z}$, а R_2 на $\{-1000,\ -888,\ -100,\ -20,\ 20,\ 30,\ 40,\ 70,\ 150,\ 256,\ 300,\ 333,\ 555,\ 567,\ 666\}^2$. Побудувати рефлексивно-транзитивні замикання R_1 та R_2 (кожного окремо).

22*. Довести, що $\mathbb{Q}$ неперервна, але злічéнна.

23*. Раціональні чи́сла теж можна записувати у вигляді нескінчéнних десяткових дробів. Чому тоді міркування, використані при доведенні теореми Кантора про незлічéнність $\mathbb{R}$, *не* доводять незлічéнності $\mathbb{Q}$?

[27] спосіб порівняння, що може передаватися як аргумент функції сортування в сучасних мовах програмування; подальші деталі — у документації з програмування

3 Індуктивні засоби доведення

3.1 Метод математичної індукції

Метод математичної індукції (скорочено «*ММІ*», «*матіндукція*») є способом міркувань, який може бути корисним при доведенні тверджень, що залежать від натурального n. Суть методу:
якщо *доведені*

1. ***база індукції*** *(твердження виконується при $n=1$)*

   та

2. ***крок індукції*** *(з істинності твердження при $n=k$ випливає істинність твердження також і при $n=k+1$)*

то *твердження виконується для всіх натуральних n.*

Переклади: рос. «*метод математической индукции*» («*ММИ*», «*матиндукция*»), «*базис индукции*» («*база индукции*»), «*шаг индукции*»; англ. «*mathematical induction*», «*base case*» («*basis*»), «*inductive step*».

За потреби, в якості бази можна взяти не n=1, а деяке інше число. Наприклад, якщо твердження $\left(1-\frac{1}{4}\right)\cdot\left(1-\frac{1}{9}\right)\cdot\left(1-\frac{1}{16}\right)\cdot\ldots\cdot\left(1-\frac{1}{n^2}\right)=\frac{n+1}{2n}$ осмислене лише при $n\geqslant 2$, то за базу доцільно взяти n=2. Аналогічно, якщо $2^0+2^1+2^2+\ldots+2^n=2^{n+1}-1$ осмислене для $n\in\mathbb{Z}^+$, тобто також і при n=0, то за базу можна взяти n=0. І так далі.

Правильність ММІ як методу міркувань не доводиться, а приймається за аксіому. І все ж спробуємо пояснити, як він працює. Правильність досліджуваного твердження при $n=1$ з'ясовується безпосередньо (як база індукції); правильність при $n=2$ випливає з бази та правильності кроку індукції; правильність при $n=3$ — з бази та правильності двократного застосування кроку індукції; і так далі. «Накручуючи» крок потрібну кількість разів, отримуємо правильність досліджуваного твердження при будь-якому натуральному n.

Приклад №1 доведення за допомогою ММІ. Доведемо, що для всіх натуральних n

$$1^2+2^2+\ldots+n^2=\frac{n(n+1)(2n+1)}{6}. \tag{49}$$

База індукції. При n=1 ліва сторона набуває вигляду $1^2=1$, права $\frac{1\cdot(1+1)\cdot(2+1)}{6}=\frac{1\cdot2\cdot3}{6}=1$. Отже, при $n=1$ твердження виконується. (База Ok)

Крок індукції. Спираючись на рівність

$$1^2+2^2+\ldots+k^2=\frac{k(k+1)(2k+1)}{6} \tag{50}$$

як на гарантовано правильну, доведемо рівність

$$1^2+2^2+\ldots+k^2+(k+1)^2=\frac{(k+1)(k+2)(2k+3)}{6} \tag{51}$$

(де (50) являє собою частковий випадок (49) при $n=k$, (51) — при $n=k+1$).

Перетворимо ліву частину (51).

(Оскільки маємо справу з виразами числової алгебри (а не логічної чи множинної), тут *можна* застосувати поширене у шкільному курсі алгебри «паралельне» спрощення лівої та правої частин. На думку автора посібника, перетворення лівої частини до правої все одно краще, бо чіткіше видно, коли і як використане припущення $P(k)$; якщо треба використати *не*еквівалентні перетворення (як у доведенні *не*рівності з наступного прикладу), легше переконатись, що вони використані у правильному напрямку. Але принципової заборони на «паралельні» спрощення матіндукція все ж *не* накладає.)

$\underbrace{\left(1^2+2^2+\ldots+k^2\right)}_{\text{дорівнює лівій частині (50)}}+(k+1)^2=$ — ми вважаємо рівність (50) гарантовано правильною; отже, можна замінити виділений підвираз на праву частину (50)

$=\frac{k(k+1)(2k+1)}{6}+(k+1)^2=$ — приводимо до спільного знаменника і додаємо

$=\frac{k(k+1)(2k+1)+6(k+1)^2}{6}=$ — виносимо спільний множник $(k+1)$

$=(k+1)\cdot\frac{k(2k+1)+6(k+1)}{6}=$ — розкриваємо дужки та зводимо подібні

$= (k+1)\cdot\dfrac{2k^2+7k+6}{6} =$ розкладаємо чисельник на множники

$= \dfrac{(k+1)\cdot(k+2)(2k+3)}{6}.$ отримали праву частину (51).

Тобто, нам вдалося довести твердження (51), користуючись лише правильними перетвореннями, а також твердженням (50). (Крок Ok)

Отже, раз доведено і базу, і крок, твердження (49) виконується для всіх натуральних n. ■

Приклад №2 доведення за допомогою методу математичної індукції. Доведемо, що для всіх натуральних n та для всіх дійсних $a\geqslant -1$

$$(1+a)^n \geqslant 1+an \tag{52}$$

Очевидно (хоча б з того, що n натуральне, а a дійсне), що індукцію слід проводити по n, розглядаючи $a\geqslant -1$ як константу (параметр).

База індукції. При $n=1$ ліва сторона набуває вигляду $(1+a)^1 = 1+a$, права — вигляду $1+a\cdot 1 = 1+a$. Отже, при $n=1$ твердження виконується. (База Ok)

Крок індукції. Спираючись на нерівність

$$(1+a)^k \geqslant 1+ak \tag{53}$$

як на гарантовано правильну, доведемо нерівність

$$(1+a)^{k+1} \geqslant 1+a(k+1) \tag{54}$$

(де (53) являє собою частковий випадок (52) при $n=k$, (54) — при $n=k+1$).

$(1+a)^{k+1} =$ беремо ліву частину (54); розкладаємо x^{k+1} як $x^k\cdot x$

$= \underbrace{(1+a)^k}_{\text{=лівій частині (53)}} \times(1+a) \geqslant$ замінюємо ліву частину (53) на меншу або рівну за неї праву частину (53); оскільки замінюється не сам вираз $(1+a)^k$, а домножений на $(1+a)$, тут *використовується*, що $a\geqslant -1$, тож правильна нерівність (53) домножається *на невід'ємне* $(1+a)\geqslant 0$.

$\geqslant (1+an)\times(1+a) =$ розкриваємо дужки

$= 1+an+a+a^2n =$ виносимо за дужки a з суми $an+a$.

$= 1+a(n+1)+a^2n \geqslant$ відкидаємо a^2n (невід'ємне, бо $a^2\geqslant 0$, $n\geqslant 1$)

$\geqslant 1+a(n+1)$. отримали праву частину (54).

Тобто, нам вдалося довести твердження (54), користуючись лише правильними перетвореннями, а також твердженням (53). (Крок Ok)

Отже, раз доведено і базу, і крок, твердження (52) доведене. ■

ММІ задає «загальний напрям» доведення, але ні в якому разі *не* є «повністю автоматизованим» методом. Головним чином тому́, що крок індукції потрібно щоразу (для кожного твердження) доводити по-своєму.

Відзначимо, що неправильність бази означає неправильність досліджуваного твердження (знайшли, що воно хибне для $n=1$ — значить, воно точно не є істинним *для всіх* n). Тим не менш, невдача при спробі доведення *кроку* індукції *не* гарантує неправильності твердження.

Наприклад, якщо спробувати довести са́ме матіндукцією твердження $\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^n}<1$, запросто може вийти щось у стилі «база $\frac{1}{2}<1$ виконується, але ж з того, що $\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^k}<1$ не слідує, що $(\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^k})+\frac{1}{2^{k+1}}<1$, адже коли до числа́, меншого 1, щось додати, воно цілком може вийти й більшим 1, так що крок не доведено».

Але це нічого не означає! Хоча б тому, що не довели, але й не довели протилежного. Якщо додати *щось* до *деякого* числа́, меншого 1, то може й вийде більше 1; але як щодо конкретного додавання $\frac{1}{2^{k+1}}$ до $(\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^k})$?

Сама́ нерівність *правильна*, її *можна* довести (наприклад, спочатку довести, чи то матіндукцією, чи то якимсь іншим способом, що $\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^n}=1-\frac{1}{2^n}$, потім сказати, що $1-\frac{1}{2^n}<1$). Але у вигляді нерівності, котра не згадує про $1-\frac{1}{2^n}$, твердження виявилося незручним для ММІ.

3.2 Інваріант циклу

Інваріант циклу (мається на увазі цикл мови програмування — `for`, `while`, тощо) є засобом теоретичного обґрунтування правильності програм. Інваріант не потрібен компілятору; він *лише* допомагає розуміти та доводити, чому виконання циклу призводить са́ме до таких наслідків.

Інваріант циклу (рос. «*инвариант цикла*», англ. «*loop invariant*») — це предикат, залежний від змінних, значення яких можуть змінюватися у циклі, але істинний у деякий чітко визначений момент *кожної ітерації* циклу.

Стандартним для інваріанта моментом вважатимемо той момент, коли приймається рішення, чи продовжувати цикл, чи завершувати.

Зокрема, для часто вживаного циклу "`for(int i=0; i<N; i++)`" рішення щодо продовження чи завершення приймається: вперше — коли змінна `i` щойно ініціалізована нулем; подальші рази — коли змінна `i` щойно збільшена дією `i++`. При $N \geqslant 0$ така ситуація настає $N+1$ разів, вперше перед ітерацією при $i=0$, востаннє при завершенні циклу, після збільшення `i` з $N-1$ до `N`.

З циклом `while` усе аналогічно: стандартним є момент, коли перевіряється умова `while`-а, тобто теж і перед початком кожної ітерації, і коли після останньої ітерації приймається рішення завершити цикл.

Можливі й не стандартні ситуації, коли істинність інваріанта розглядають у деякий інший момент. Але це теж повинен бути чітко визначений для цього циклу момент (а *не* так, що для одного й того ж циклу на одних ітераціях предикат істинний після виконання одного оператора, на інших ітераціях — іншого, й усе це хаотично змінюється).

Схема доведення правильності циклу із застосуванням інваріанта така:

1. Довести, що інваріант виконується на першій ітерації циклу.
2. Довести, що інваріант зберігає істинність після виконання ітерації.
3. Довести, що цикл завершиться, а не працюватиме вічно, і що після завершення циклу змінні набудуть саме тих значень, які потрібні.

Ці формулювання дуже вже короткі, тож спочатку розглянемо приклад простого застосування інваріанта, потім детальні коментарі до пунктів 2–3, потім ще кілька прикладів.

Приклад 1 (степінь, простий алгоритм). Розглянемо наведений алгоритм обчислення a^N (`a` — типу `double`, `N` — невід'ємне типу `int`). Він загальновідомий, але його можна й довести, використавши інваріант *«у стандартний для інваріантів момент, виконується* $\texttt{deg} = \texttt{a}^{\texttt{i}}$*»*.

```
double deg = 1.0;
for(int i=0; i<N; i++)
  deg *= a;
```

П. 1: це справді так перед початковою ітерацією, бо $\texttt{deg} = 1$, $a^0 = 1$.

(Є різні думки щодо того, чи включає це міркування випадок $\texttt{a}=0$. З одного боку, 0^0 часто вважають невизначеністю, бо $\lim\limits_{x\to 0} x^0 = 1 \neq 0 = \lim\limits_{y\to 0} 0^y$. Тоді доведення не охоплює випадок $\texttt{a}=0$, але можна побудувати окреме доведення правильності при $\texttt{a}=0$, $\texttt{N}>0$ (не окремого пункту 1, а окреме доведення всього алгоритму). З іншого боку, у програмуванні (на відміну від математичного аналізу) більш-менш прийнято вважати $0^0 = 1$, тоді це взагалі не потребу́є окремого розгляду.)

П. 2: це лишається правильним після кожної чергової пари дій `deg*=a` та `i++`, бо домноження на `a` якраз і є збільшенням показника степеню на 1.

П. 3: якщо вхідні дані коректні ($\texttt{N} \geqslant 0$), то цикл завершується завжди, при $\texttt{i}=\texttt{N}$ (остання ітерація відбувається при $\texttt{i}=\texttt{N}-1$, при $\texttt{i}=\texttt{N}$ цикл *лише* завершується). Враховуючи інваріант $\texttt{deg} = \texttt{a}^{\texttt{i}}$, це означає, що цикл завершується при $\texttt{deg} = \texttt{a}^{\texttt{N}}$ (що й треба). ■

Зверніть увагу, що рівність $\texttt{deg} = \texttt{a}^{\texttt{i}}$, як правило, не виконується, коли чергове `deg*=a` вже виконано, а чергове `i++` — ще ні. Це підкреслює, чому треба вказувати чітко визначений момент істинності інваріанта.

Коментарі до загальних правил застосування інваріанта П. 2 має доводити *не лише* істинність для *дру́гої* ітерації; доведення мусить бути таким, щоб його можна було застосовувати як крок матіндукції, тобто, провівши теоретичні міркування один раз, могти «накручувати» *«раз є факт істинності інваріанта на першій ітерації та це доведення, то інваріант істинний також і на дру́гій»*; *«раз є факт істинності інваріанта на дру́гій ітерації та це доведення, то інваріант істинний також і на третій»*; і так далі.

Якщо побудувати доведення п. 2 не вдалося — як і у «звичайній» матіндукції, це нічого не значить. Таке може бути і коли інваріант порушується, і коли не порушується, але це незручно доводити са́ме такими засобами. Буває й так, що проблеми доведення виявляють проблеми алгоритму або допомагають виявити справжні ме́жі його застосовності (як-то «алгоритм правильний *лише* для вхідних даних з такого-то діапазону»).

П. 3 нерідко дуже простий, як-то: для "`for(int i=0; i<N; i++)`", при $\texttt{N}>0$, відсутності інших змін `i` та відсутності `break`-ів чи інших засобів дострокового завершення, елементарно отримуємо, що цикл обов'язково завершиться, причому са́ме при $\texttt{i}=\texttt{N}$; щоб взнати значення інших змінних, підставимо в інваріант `N` замість `i`.

Але бувають і такі цикли, для яких складо́ві «довести, що завершиться» та «встановити значення змінних наприкінці виконання» дуже складні́.

```
cin >> n;
int steps=0;
while(n!=1) {
  if(n%2==0)
    n/=2;
  else
    n=3*n+1;
  steps++;
}
```

Наприклад, наведений цикл намагалися досліджувати чимало серйозних математиків (його математичні назви — «*гіпотеза Коллатца*», «*гіпотеза* $3n{+}1$», «*Сіракузька проблема*»). І все одно наука не знає навіть того, чи гарантовано він завершиться для будь-якого $n\in\mathbb{N}$, не кажучи вже про зв'язок між початковим `n` і остато́чним `steps`.

Просто запустити програму й подивитися теж пробували. Станом на початок 2016 р. шляхом перебору встановлено, що цикл завершується для всіх $1\leqslant n\leqslant 2{,}7\cdot 10^{15}$. Але ж це ніяк не доводить скінче́нності при більших n...

Звісно, «проблема $3n{+}1$» — штучна й теоретична задача. Але все одно вона показує: *бувають* цикли, для яких аналіз п. 3 дуже складни́й.

Тобто, є зра́зу кілька причин, чому інваріант циклу не є всемогутнім засобом: його треба ще зуміти побудувати так, щоб він правильно відповідав циклу (він же не складова програми, а будується окремо); бувають цикли, для яких вельми важко аналізувати п. 2 та/або п. 3...

Але дослідження інваріанта все ж буває корисним. Зокрема, коли пропонуються/розробляються алгоритми, які начебто ефективні й елегантні, але їхня правильність не очевидна (до таких алгоритмів можна віднести, зокрема, алгоритм Евкліда (приклад 6) та ефективне піднесення до степеню (приклад 7); насправді їх багато, але ж не можна вмістити тут усі корисні алгоритми...).

Приклад 2 (максимум). Доведемо, що наведений відомий алгоритм справді знаходить максимальне значення. Сформулюємо інваріант так: *«у стандартний для інваріантів момент, змінна* `max_value` *дорівнює максимуму серед перших* `i` *елементів, з 0-го по* $(\texttt{i}-1)$*-й»*.

```
auto max_value = a[0];
for(int i=1; i<N; i++)
  if(a[i] > max_value)
    max_value = a[i];
```

П. 1: перед циклом відбувається ініціалізація `max_value=a[0]`, цикл починається з `i=1`, отже «перші `i` елементів, з 0-го по $(\texttt{i}-1)$-й» являють собою єдиний елемент `a[0]`, а раз елемент єдиний, то він максимальний.

П. 2: оскільки наступний стандартний для інваріантів момент настане після `i++`, слід показати, що після завершення `if`-а, але перед `i++`, змінна `max_value` дорівнює максимальному серед елементів з 0-го по `i`-й включно. Якщо `a[i]>max_value`, то, враховуючи смисл `max_value`, `a[i]` більший за усі елементи з 0-го по $(\texttt{i}-1)$-й. Тож після присвоєння `max_value=a[i]` змінна `max_value` дорівнюватиме максимальному серед елементів з 0-го по `i`-й включно, бо у цьому випадку `a[i]` і є максимальним. Інакше (при `a[i]<=max_value`), максимум переглянутої частини не змінюється (і не повинен), тож інваріант правильний тому, що був правильним раніше.

П. 3 елементарний: цикл `for` гарантовано завершується при $\texttt{i}=\texttt{N}$, тож «перші `N` елементів, з 0-го по $(\texttt{N}-1)$-й» являють собою увесь масив, тобто `max_value` містить шукане максимальне значення усього масиву. ■

Приклад 3 (сортування вибором). Сортування вибором містить два цикли, тож повне доведення мало би містити окремо дослідження інваріанта внутрішнього циклу, окремо зовнішнього.

Але внутрішній цикл аналогічний попередньому прикладу (хоч і має низку відмінностей: шукається не максимум, а мінімум; не значення, а індекс у масиві; мінімум не усього масиву, а починаючи з `i`-го елементу). Тому дозволимо собі лише чітко сформулювати твердження *«після виконання рядків 2–5 змінна* `imin` *містить індекс елемента, мінімального на проміжку від* `i`*-го елементу включно до кінця масиву»* і сказати, що його доведення відрізняється від попереднього

```
/*1*/  for(int i=0; i<N; i++) {
/*2*/    int imin = i;
/*3*/    for(int j=i+1; j<N; j++)
/*4*/      if(a[j] < a[imin])
/*5*/        imin = j;
/*6*/    swap(a[i], a[imin]);
/*7*/  }
```

До початкової ітерації							
зі значенням i=0	3	1	4	1	5	9	2
i щойно стало 1	1	3	4	1	5	9	2
i щойно стало 2	1	1	4	3	5	9	2
i щойно стало 3	1	1	2	3	5	9	4
i щойно стало 4	1	1	2	3	5	9	4
i щойно стало 5	1	1	2	3	4	9	5
i щойно стало 6	1	1	2	3	4	5	9
i щойно стало 7	1	1	2	3	4	5	9

прикладу лише очевидними змінами. Тобто, вважатимемо його правильним і зосередимося на зовнішньому циклі.

Сформулюємо інваріант зовнішнього циклу як *«у стандартний для інваріантів момент, щонайменше перші* `i` *елементів, з 0-го по* (`i`−1)*-й, вже містять остато́чні значення (ті, які повинні бути в остато́чно відсортованому масиві)»*. Тобто, `a[0]` мінімальне серед усіх значень, `a[1]` — мінімальне серед решти (не враховуючи `a[0]`), `a[2]` — мінімальне серед решти (не враховуючи `a[0]` та `a[1]`), тощо. Все це гарантовано лише у проміжку з 0-го по (`i`−1)-й елементи (а у проміжку з `i`-го по (`N`−1)-й якщо й виконується, то, мабуть, випадково). Са́ме проміжки, на яких властивість виконується гарантовано, підкреслені у табличці праворуч від коду алгоритму.

П. 1: перед початкової ітерацією, «щонайменше перші 0 елементів вже містять остато́чні значення»: ще нічого не зробили, ще нема гарантованого результату. У подібних випадках зазвичай вважають, що властивість виконується, бо у межах порожнього проміжку нема кому її порушити.

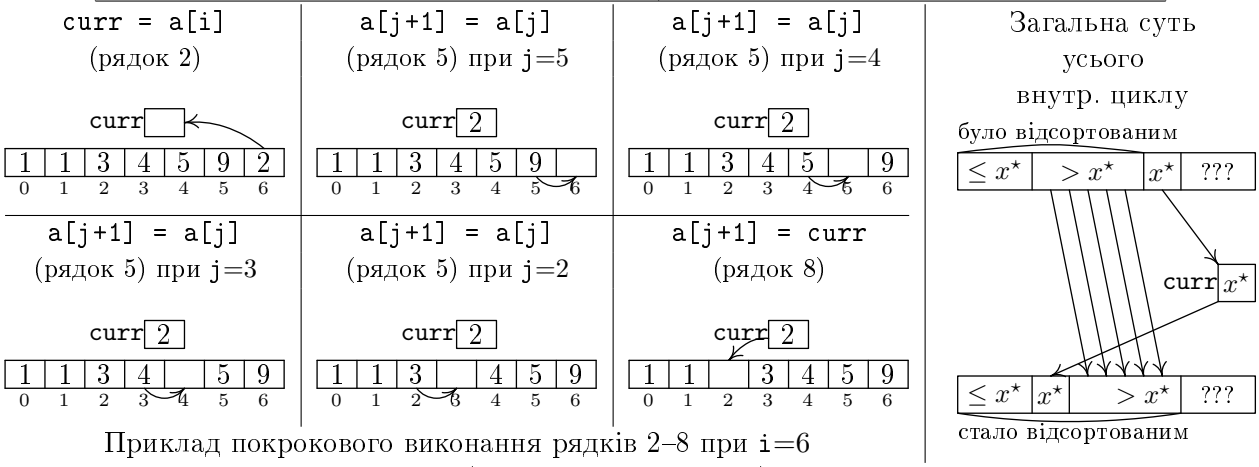
П. 2: раніше виконувалося, що елементи з 0-го по (`i`−1)-й вже містять остато́чні значення, тож `a[i]` повинен ста́ти мінімальним серед решти (проміжку з `i`-го по (`N`−1)-й). Рядки 2–5 са́ме такий елемент і знаходять, після чого рядок 6 ставить його на потрібне `i`-те місце. Так що перед початком наступної ітерації інваріант теж виконується.

П. 3 традиційний: цикл `for` завершується при `i` = `N`, після його завершення «перші `N` елементів» охоплюють увесь масив, тобто усі елементи містять ті значення, які повинні. ■

Приклад 4 (сортування вставками). Алгоритм сортування вставками працює за іншим принципом, ніж сортування вибором.

```
/*1*/  for(int i=1; i<N; i++) {
/*2*/    auto curr = a[i];
/*3*/    int j = i-1;
/*4*/    while(j>=0 && a[j] > curr) {
/*5*/      a[j+1] = a[j];
/*6*/      j-;
/*7*/    }
/*8*/    a[j+1] = curr;
/*9*/  }
```

До початкової ітерації							
зі значенням i=0	3	1	4	1	5	9	2
i щойно стало 2	1	3	4	1	5	9	2
i щойно стало 3	1	3	4	1	5	9	2
i щойно стало 4	1	1	3	4	5	9	2
i щойно стало 5	1	1	3	4	5	9	2
i щойно стало 6	1	1	3	4	5	9	2
i щойно стало 7	1	1	2	3	4	5	9



Приклад покрокового виконання рядків 2–8 при `i`=6

Одна ітерація зовнішнього циклу (рядки 2–8 алгоритму) збільшує розмір вже відсортованого початку на 1

Інваріант зовнішнього циклу — *«у стандартний для інваріантів момент, перші* `i` *елементів являють собою переставлені у відсортованому порядку перші* `i` *елементів початкового масиву»*. Тобто, на відміну від сортування вибором, ці елементи не обов'язково містять остато́чні значення; цілком можливо, що ті значення ще будуть зсунуті праворуч, щоб вставити подальші менші значення. Але тут виконується інше (що не виконується для сортування вставками): у будь-який стандартний для інваріантів момент, значення перших `i` елементів одні й ті самі, лише переставлені між собою.

Строге доведення усього сортування повинно було б включати у себе строге доведення (із використанням свого інваріанта) твердження «виконання рядків 2–8 збільшує розмір вже відсортованого початку на 1». На жаль, це виходить дещо громіздко й заплутано, тому пропонується сформулювати не зовсім строгу аргументацію, вважаючи це твердження очевидним на основі тексту програми та рисунків.

При такому підході, не строга аргументація сортування дуже проста:

П. 1: при `i`$=1$, проміжок з єдиного елемента `a[0]` неминуче відсортований (для порушення відсортованості потрібні хоча б 2 елементи).

П. 2 якраз і є твердженням, сприйнятим за очевидне.

П. 3: стандартний `for` гарантовано завершується при `i`$=$`N`, «відсортовані перші `N` елементів» являють собою відсортований увесь масив. ■

Приклад 5 ((2i+5)!). У розглянутих досі прикладах, інваріант обґрунтовував правильність вже готового циклу, і це типове його використання. Але іноді зручно змістити ролі, використавши інваріант для *побудови* циклу.

Нехай потрібно написати програму, яка послідовно для всіх $i = 0, 1, 2, 3, \ldots, 50$ виведе значення i, $2i{+}5$ та $(2i{+}5)!$; факторіал рахувати в типі `double`, бо в цілочисельні все одно не поміститься.

```
double fact(int n) {
  double res = 1.0;
  for(int i=1; i<=n; i++)
    res *= i;
  return res;
}
. . . . . .
  cout << i << "\t"
     << 2*i+5 << "\t"
     << fact(2*i+5)<<endl;
```

i	2i+5	(2i+5)!
0	5	120
1	7	5040
2	9	362880
3	11	3.99168e+007
4	13	6.22702e+009
5	15	1.30767e+012
6	17	3.55687e+014
7	19	1.21645e+017
8	21	5.10909e+019

У правій стороні таблички вказано кілька перших рядків тих результатів, які слід отримати. У лівій — два різні фрагменти програми (окремо функція `fact`, окремо, десь у іншій функції, виклики функції `fact`), які взагалі-то розв'язують цю задачу. Якщо на перше місце ставити читабельність (зазвичай са́ме так і є), то якраз це і є найкращим розв'язком.

Але *якщо* ефективність важливіша за читабельність, погано те, що кожен факторіал рахується заново, хоча взагалі-то включає в себе попередній, як-то $7! = \big(1{\times}2{\times}3{\times}4{\times}5\big) \times 6{\times}7$. За рахунок цього можна зменшити об'єм обчислень.

Тож побудуємо цикл, що перебирає `i` від 0 до 50, спочатку із ще не дописаними виразами. Потім допишемо ті вирази так, щоб вийшов інваріант *«у стандартний для інваріантів момент,* `f`*=(2*`i`*+5)!»*.

```
double f = /*треба дописати*/;
for(int i=0; i<=50; i++) {
  f *= /*треба дописати*/;
  cout << i << "\t"
       << 2*i+5 << "\t"
       << f << endl;
}
```

Поєднавши інваріант та формулу, що виражає $7!$ через $5!$, легко здогадатися, що $(2i{+}5)!$ при черговому `i` відрізняється від $(2i{+}5)!$ при попередньому `i` в точності множниками $(2i{+}4) \cdot (2i{+}5)$. Отже, перший рядок *тіла* циклу набуває вигляду `f*=(2*i+4)*(2*i+5)`.

Після того, як сформовано оператор `f*=(2*i+4)*(2*i+5)`, бачимо, що значення $(2{\cdot}0 + 5)! = 5! = 120$ має досягтися після домноження на $(2{\cdot}0 + 4) \times (2{\cdot}0 + 5) = 4{\times}5$, значить, до циклу треба ініціалізувати `f` як $1{\times}2{\times}3 = 6$.

(Са́ме так. Са́ме спочатку підбирається, яку дію виконувати на кожній ітерації, а потім залежно від того підбирається ініціалізація. Хоча б тому, що дію з ітерації виконувати багато разів, і варто подбати в першу чергу про неї. А ініціалізацію, яка відбувається один раз, можна й якось підлаштувати.)

Приклад 6 (алгоритм Евкліда). *Спільний дільник* чисел a та b — число, на яке діляться (націло, без остачі) одночасно і a, і b. Термін *найбільший спільний дільник* a та b, скорочено *НСД* (a,b), вводиться очевидним чином, як найбільший серед спільних дільників. Наприклад: НСД$(7,9)$=1, НСД$(8,10)$=2, НСД$(20,30)$=10, НСД$(4,8)$=4.

Коли треба писати код, що шукає НСД, зручно користуватися алгоритмом Евкліда, який і легко писати, і швидко виконується. Класична версія алгоритму Евкліда: *«Поки чи́сла не рівні між собою, віднімати з більшого менше»*. Наприклад, для 8 і 10: $(8,10)\to(8,2)\to(6,2)\to(4,2)\to$ $\to(2,2)$. Недолік цієї версії: якщо одне з чисел *значно* більше іншого, можна дуже багато разів віднімати з великого числá одне й те са́ме малé.

Тому́ *сучасна версія алгоритму Евкліда* замінює віднімання на взяття залишку; внаслідок цього, умову «чи́сла стали рівними» доводиться змінити на «одне з чисел стало нулем». Наприклад, для тих самих 8 і 10 це буде $(8,10)\to(8,2)\to(0,2)$.

Вхідними даними алгоритму Евкліда можуть бути лише цілі невід'ємні чи́сла. Крім того, для вхідних даних $a{=}b{=}0$ він вертає як результат 0, що навряд чи правильно (чи може 0 взагалі бути дільником? а чи не краще взяти 100, для якого $100{>}0$, $0\vdots 100$? а чи не краще не 100, а ще більше число? то це тоді так збільшувати аж до нескінче́нності включно?).

(Тут можна згадати задачу 26 зі стор. 78. Там елемент 0 виявився максимальним та найбільшим (у смислі означень зі стор. 67) щодо нелінійного відношення порядку $(x\preccurlyeq y)\Leftrightarrow(y\vdots x)$. Якщо уявити, що взяли са́ме таке нелінійне відношення порядку, і провели топологічне сортування так, що для додатних чисел порядок відповідає звичайному числовому порядку, а 0 так і лишили найбільшим — *в таких умовах* результат НСД$(0,0)=0$ стає осмисленим. Само собою, при такій підміні поняття «найбільший» виникають деякі інші проблеми, тому питання, чи варто вважати НСД$(0,0)=0$, лишається відкритим. Тут лише показано, що можлива ситуація, коли це доречно.)

Тож просто запам'ятаємо, що випадок $a{=}b{=}0$ може потребува́ти окремого розгляду, а коли потрібно шукати НСД чисел довільних знаків — варто взяти їх по модулю ще до початку робо́ти алгоритму.

```
unsigned long long gcd(
        unsigned long long a,
        unsigned long long b) {
  while(a>0 && b>0) {
    if(a>b) a%=b;
    else    b%=a;
  }
  return a+b;//одне (невідомо яке) 0
} // тож вертається значення іншого
```

Алгоритм Евкліда (сучасна версія)

Правильність алгоритму Евкліда зовсім не очевидна; са́ме це й робить цінним його доведення.

Інваріант циклу (теж не очевидний) має вигляд *«перелік спільних дільників поточних значень a та b такий самий, як і перелік спільних дільників початкових значень a та b»*. Цей інваріант виконується завжди (не лише в якісь окремі моменти).

П. 1: перелік спільних дільників не змінився, бо не змінилися самі́ a та b.

П. 2: цикл ще продовжується — значить, на початку ітерації $a{>}0$, $b{>}0$.

Розглянемо випадок $a{>}b{>}0$. Позначимо $a \bmod b$ як c (можливо як $c{>}0$, так і $c{=}0$). Тоді a можна подати як $a=b\times d+c$, де $d=a \operatorname{div} b$ є натуральним. (Мова *не* йде про включення додаткових змінних у алгоритм. Величи́ни c, d, p, a_1, b_1, q, b_2, c_2 розглядаються *лише* у доведенні.) Нехай деяке p є спільним дільником a та b (не обов'язково найбільшим). Тобто, $a_1{=}a/p$ та $b_1{=}b/p$ натуральні. Тоді, $c=a-b\times d=a_1\times p-b_1\times p\times d=(a_1-b_1\times d)\times p$, тобто $c\vdots p$. Цим показано, що якщо число було спільним дільником a та b (старих значень змінних алгоритму Евкліда `a` та `b`), воно лишається спільним дільником і для c та b (нових значень змінних `a` та `b`).

Тепер нехай деяке q (можливо як $q{=}p$, так і $q{\neq}p$) є спільним дільником b та c, тобто $b_2{=}b/q$ та $c_2{=}c/q$ є цілими. Тоді $a=b\times d+c=b_2\times q\times d+c_2\times q=(b_2\times d+c_2)\times q$, тобто $a\vdots q$. Цим показано, що серед спільних дільників c та b (нових значень змінних `a` та `b`) не з'явилося ніяких нових значень, які не є спільними дільниками a та b (старих значень змінних `a` та `b`).

Формально кажучи, у попередніх двох абзацах розглядався лише випадок $a{>}b{>}0$, тож правильність інваріанта циклу поки що доведена лише частково, для $a{>}b{>}0$. Але при $0{<}a{\leqslant}b$ можна провести всі ті ж міркування для b, a та $c=b \bmod a$, а інших, крім цих двох, випадків у рамках $a{>}0$, $b{>}0$ не буває. Тому висновки двох попередніх абзаців можна посилити до «виконання усього циклу `while(a>0&&b>0)...` не змінює сукупності спільних дільників», тобто п. 2 аналізу інваріанта упішно завершений.

Починаємо аналізувати п. 3. Вище вже згадано про особливість вхідних даних $a{=}b{=}0$, тож розглядаємо лише випадок, коли хоча б одне з початкових значень a, b (як правило, обидва)

строго більше 0. За одну ітерацію змінюється значення лише однієї зі змінних `a` або `b`, тож умова продовження циклу `a>0 && b>0` порушиться са́ме у ситуації, коли *одне* зі значень `a` або `b` дорівнює 0, інше$\neq$0. Функція вертає як відповідь «інше» (ненульове) значення. Це справді НСД, бо абсолютно всі дільники будь-якого додатного числа́ є дільниками нуля, тож перелік спільних дільників додатного числа́ і нуля дорівнює просто переліку дільників цього додатного числа́, і саме́ це число є найбільшим зі своїх дільників. А враховуючи інваріант «перелік спільних дільників незмінний», це число є найбільшим також і серед усіх спільних дільників початкових значень `a` та `b`, тобто шуканим НСД.

Тільки це ще не кінець п. 3, бо досі доведено *лише* те, що *якщо* алгоритм завершує роботу, то результат правильний. Те, що він при будь-яких $a>0$, $b>0$ справді завершить роботу, а не зациклиться, треба доводити окремо. Це робиться так: при $a>b>0$, дія `a%=b`, не чіпаючи натурального `b`, замінює значення `a` на ціле невід'ємне, строго менше старого `a`; інакше (при $b\geqslant a>0$), дія `b%=a`, не чіпаючи натурального `a`, замінює значення `b` на ціле невід'ємне, строго менше старого `b`. Тобто, на кожній ітерації циклу, сума `a+b`, лишаючись натуральною, стає строго меншою. А це не може тривати вічно. ■

Приклад 7 (швидке піднесення до степеню, ітеративне). Підносити до степеню можна й швидше, ніж циклом з прикладу 1. Часто найкращим є спосіб *«використати формулу $a^n=e^{n\cdot\ln a}$ та розгалуження, щоб розібратися з випадком, коли $a\leqslant 0$, але, хоч $\ln a$ й не існує, піднести до степеню все-таки можна, бо n ціле»*; більшість компіляторів, в яких є стандартна (бібліотечна) функція `pow(double a, double n)`, са́ме так її й реалізують.

Але до степеню підносять не лише чи́сла, а також квадратні матриці. Або цілі чи́сла у кільці за модулем p (вхідними даними є *три* натуральні числа́ a, N, p, і треба знайти $a^N \bmod p$). Причому, якщо років з 50 тому це здавалося дивною забаганкою математиків, за́раз вираз $a^N \bmod p$ використовується в багатьох методах шифрування. Причому, його обчислюють для, наприклад, 1024-бітових a, N та p. Застосувати тут стандартні exp та ln просто неможливо. А простий алгоритм з прикладу 1 працював би *дуже* довго, значно довше, ніж існує планета Земля.

Алгоритм швидкого піднесення до степеню одночасно і значно швидший за цикл з прикладу 1, і придатний для ситуацій, коли формула $a^n=e^{n\cdot\ln a}$ втратила смисл/зручність (правда, накладає своє обмеження: показник n мусить бути цілим невід'ємним).

(На жаль, у цього алгоритму чимало різних назв, і жодна з них не є загальноприйнятою. Укр. та рос. мовами досить поширена назва *«індійський алгоритм»*, але, наскільки відомо автору посібника, ніякої схожої назви англійською нема; англійською поширені на́зви *«exponentiation by squaring»*, *«repeating squaring»* та *«Russian peasant method»*.)

У табличці наведено зразу дві версії цього алгоритму; ще одна (рекурсивна) наведена у розд. 3.3.

```
double b = a;
int m = n;
double res = 1.0;
while (m > 0) {
  if (m % 2 == 1)
    res *= b;
  m /= 2;
  b *= b;
}
```

```
double res = 1.0;
while (true) {
  if (n % 2 == 1)
    res *= a;
  n /= 2;
  if (n == 0)
    break;
  a *= a;
}
```

Чому це швидше, ніж цикл з прикладу 1? Тому́, що зменшення показника степеню удвічі (для па́рних) супроводжується лише одним множенням, або зменшення трохи більш, як удвічі (для непарних) — лише двома. Наприклад, від n=9876 за одне множення переходимо до 4938, за ще одне до 2469, за ще два до 1234, ще одне до 617, ще два до 308, ще одне до 154, ще одне до 77, ще два до 38, ще два до 19, ще два до 9, ще два до 4, ще одне до 2, ще одне до 1, ще одне–два (залежно від версії) зменшують показник до 0 і завершують алгоритм. Тобто, для обчислення a^{9876}, замість 9876 чи 9875 множень, досить 20–21. Говорячи загально, сумарна кількість множень цього алгоритму не перевищує $1+2\log_2 n$ (що для $n\approx 1000$ становить до ≈ 20, для $n\approx 10^9$ — до ≈ 60, для $n\approx 10^{18}$ — до ≈ 120).

Цей спосіб дуже неочевидний. Тим важливіше його довести! Доводити зручніше 1-шу (лівішу) версію. Інваріант: *«у стандартний для інваріантів момент, $\mathtt{a}^{\mathtt{n}}=\mathtt{res}\times\mathtt{b}^{\mathtt{m}}$»*.

П. 1 очевидний: зразу після `b=a`, `m=n`, `res=1` має місце $\mathtt{res}\times\mathtt{b}^{\mathtt{m}}=1\times\mathtt{a}^{\mathtt{n}}$.

При доведенні п. 2, розглянемо випадки $\mathtt{m}=2k$ (`m` па́рне) та $\mathtt{m}=2k+1$ (`m` непарне). *Не* змінимо програму, а розглянемо теоретично у рамках доведення. Так само теоретично скажемо, що змінна `b` на початку ітерації циклу містить значення $b^\star$, а дія `b*=b` перетворює його у $(b^\star)^2$.

У випадку $\mathtt{m}=2k$, до ітерації $\mathtt{res}\times\mathtt{b}^{\mathtt{m}}$ дорівнювало $\mathtt{res}\times(b^\star)^{2k}$, після ітерації стало дорівнювати $\mathtt{res}\times((b^\star)^2)^k$, причому значення `res` незмінне. Отже, враховуючи $(b^\star)^{2k}=((b^\star)^2)^k$, увесь добуток $\mathtt{res}\times\mathtt{b}^{\mathtt{m}}$ не змінився.

У випадку $\mathtt{m}=2k+1$, значення `res` змінюється. Позначимо значення `res` до ітерації за r. Тоді до ітерації $\mathtt{res}\times\mathtt{b}^{\mathtt{m}}$ дорівнювало $r\times(b^\star)^{2k+1}$, після ітерації стало дорівнювати $\underbrace{(r\times b^\star)}_{\text{нове }\mathtt{res}}\times\underbrace{((b^\star)^2)^k}_{\text{нове }\mathtt{b}^{\mathtt{m}}}=$ $=r\times b^\star\times(b^\star)^{2k}=r\times(b^\star)^{2k+1}$. Тобто, увесь добуток $\mathtt{res}\times\mathtt{b}^{\mathtt{m}}$ знов не змінився.

Що й закінчує формальне доведення п. 2, адже інших випадків, крім $\mathtt{m}=2k$ та $\mathtt{m}=2k+1$, для цілих невід'ємних `m` не буває.

Доведення п. 3 елементарне: якщо початкове значення `m` ціле невід'ємне, при багатократних цілочисельних діленнях на 2 змінене значення `m` обов'язково кінець кінцем стане рівним 0, бо на кожній ітерації зменшується принаймні удвічі, лишаючись цілим. Підставивши кінцеве значення $\mathtt{m}=0$ у інваріант «$\mathtt{a}^{\mathtt{n}}=\mathtt{res}\times\mathtt{b}^{\mathtt{m}}$», отримаємо $a^n=\mathtt{res}\times\underbrace{b^0}_{=1}$, тобто $\mathtt{res}=a^n$. ■

(До останнього кроку можна поставитися як до універсального, а можна сказати, що він лише для $\mathtt{b}\neq0$; тоді випадок $\mathtt{b}=0$ (фактично, $a=0$) можна проаналізувати окремо й побачити, що для нього, при $\mathtt{n}>0$, теж правильно знаходитиметься $\mathtt{res}=0$.)

Для неформального розуміння корисно подивитися на хід виконання алгоритму, уявляючи `n` та `m` у двійковій системі числення. (Міняти щось у програмі не треба; са́ме уявити теоретично.)

Наприклад, див. наведене піднесення до степеню 50, що у двійковому записі має вигляд 110010. Кожна дія `m/=2` забирає останню двійкову цифру з `m`, а виконана щойно перед тим дія `if(m%2==1) res*=b` додає таку саму цифру спереду показника степеню a в `res`. Тож наприкінці циклу все число біт за бітом переписане з `m` у показник степеню a в `res`.

`res`	`m`	`b`
1.0	$50_{Dec}=110010_{Bin}$	a
$1.0=a^{0_{Dec}}=a^{0_{Bin}}$	$25_{Dec}=11001_{Bin}$	$a^{2_{Dec}=10_{Bin}}$
$a^{2_{Dec}}=a^{10_{Bin}}$	$12_{Dec}=1100_{Bin}$	$a^{4_{Dec}=100_{Bin}}$
$a^{2_{Dec}}=a^{010_{Bin}}$	$6_{Dec}=110_{Bin}$	$a^{8_{Dec}=1000_{Bin}}$
$a^{2_{Dec}}=a^{0010_{Bin}}$	$3_{Dec}=11_{Bin}$	$a^{16_{Dec}=10000_{Bin}}$
$a^{18_{Dec}}=a^{10010_{Bin}}$	$1_{Dec}=1_{Bin}$	$a^{32_{Dec}=100000_{Bin}}$
$a^{50_{Dec}}=a^{110010_{Bin}}$	$0_{Dec}=0_{Bin}$	$a^{64_{Dec}=1000000_{Bin}}$

Інша наведена версія відрізняється лише двома дрібними оптимізаціями: (1) уникає додаткових змінних `b` та `m`, замість цього «псуючи» (безповоротньо змінюючи) самі́ змінні `a` та `n`; (2) уникає останнього непотрібного множення `a*=a`, коли щойно стало `n==0`. *Якщо* множення фактично є викликом дуже складно́ї та повільної функції, такі дрібниці бувають варті уваги.

3.3 Доведення правильності рекурсивних підпрограм (вступ)

З рекурсією все ще складніше (чим з циклами): і за рахунок того, що рекурсія буває і пряма (підпрограма викликає себе), і непряма (підпрограми взаємно викликають одна о́дну); і за рахунок того, що навіть у прямій рекурсії кількість викликів само́ї себе може бути різною; і за рахунок того, що у загальному вигляді одні виклики можуть впливати на інші і через повернуті (`return`-ом) значення, і через зміни значень глобальних змінних. Тому наведений тут матеріал зовсім не претендує на вичерпність, будучи лише вступним оглядом лише одного часткового випадку: коли рекурсія лише пряма та зсере́дини цієї рекурсії не змінюються ніякі глобальні змінні (це називають також «функція не має побічних ефектів»). При таких обмеженнях, загальну схему доведення правильності рекурсивної підпрограми можна сформулювати так:

1. Довести скінченність
   - 1.1. Показати, що аргумент(и) рекурсивн(ого/их) виклик(у/ів) більшої вкладеності завжди менш(ий/і), ніж поточного виклику.
   - 1.2. Показати, що будь-яка можлива послідовність аргумент(у/ів) завершується деяким мінімальним елементом, який означений не рекурсивно.
     (Це забезпечує, що будь-яке заглиблення у рекурсію рано чи пізно завершується виходом з рекурсії.)

   (Хоча часто можна брати звичайне числове «менше», буває й так, що для нього ці умови не виконуються, але виконуються, якщо спеціально підібрати деяке (можливо, нелінійне) відношення порядку (див. розд. 2.6.3 та наступну сторінку).)

2. Показати, що умови виходу з рекурсії повертають правильні значення.
   (Це є певним аналогом бази матіндукції.)
3. Показати, що рекурсивна функція правильно збирає результати більших підзадач з результатів менших.
   (Це є певним аналогом кроку матіндукції; але, на відміну від кроку класичного ММІ, тут не завжди слід спиратися на $P(k)$ і доводити $P(k{+}1)$. Конкретний вигляд того, що дано і що треба довести, визначається конкретним виглядом тієї рекурисвної функції, правильність якої доводять.)

(Потреби вводити якийсь аналог інваріанта циклу нема, але це лише тому, що ми розглядаємо частковий випадок, коли рекурсивна функція не має побічних ефектів. Коли побічні ефекти допускаються, може з'являтися потреба поєднувати засоби цього підрозділу із засобами попереднього. Так що все може виходити дуже, дуже складно.)

Приклад 1 (факторіал). (Повторимо класичне зауваження, що цей код є прикладом правильного, але не доцільного використання рекурсії, й наведений тут лише як приклад простої рекурсії, яку легко проаналізувати. А коли треба рахувати факторіал на практиці, краще робити це циклом, без рекурсії.)

```
double fact(int n) {
  if (n==0)
    return 1.0;
  else
    return n * fact(n-1);
}
```

П. 1.1: `fact(n)` викликає `fact(n-1)`, тож аргумент більшої вкладеності $(n{-}1)$ справді менший за поточний аргумент (n).

П. 1.2: *якщо* початковий виклик з іншої функції відбувається з (цілим) аргументом $\texttt{n} \geqslant 0$, то чи то зразу (при `n`=0), чи то після рівно `n` викликів (кожен наступний отримує значення рівно на 1 менше) отримаємо виклик, де $n = 0$, а це умова виходу.

П. 2: єдиною умовою виходу є `n==0`, у цьому випадку відбувається `return 1.0`. Це відповідає властивості факторіала $0! = 1$.

П. 3: якщо виклик `fact(n-1)` повернув правильне значення $(n{-}1)!$, то внаслідок його домноження на n отримається правильне значення $(n{-}1)! \times n = n!$, це одна з властивостей факторіала.

Приклад 2 (швидке піднесення до степеню, рекурсивне). Очевидно правильні з точки зору алгебри формули

$$a^0 = 1, \qquad a^{2k} = \left(a^2\right)^k, \qquad a^{2k+1} = \left(a^2\right)^k \cdot a \tag{55}$$

дають ключ до іншої, ніж на стор. 88–89, версії алгоритму швидкого піднесення до степеню, яку значно легше запам'ятовувати й доводити.

Внаслідок накладних витрат на рекурсію, ця версія трохи менш ефективна, ніж раніше розглянута ітеративна, але са́ме «*трохи*»; залежно від особливостей компілятора та «заліза», відмінність у швидкодії може становити від кількох відсотків до, щонайгірше, 2–3 разів. А принциповий виграш «не більше $1{+}2\log_2 n$ множень замість n» зберігається.

```
double pow_rs(double a, int n)
{
  if (n == 0)
    return 1.0;
  if (n % 2 == 0)
    return pow_rs(a*a, n/2);
  else
    return pow_rs(a*a, n/2) * a;
}
```

П. 1.1: `pow_rs(n)` викликає `pow_rs(a*a, n/2)`, тож *дру́гий* аргумент більшої вкладеності (`n/2`, ділення цілочисельне) справді менший (враховуючи невід'ємність) за поточний аргумент (n).

(Те, що 1-й аргумент більшої вкладеності `a*a` виявляється (при $|\texttt{a}|{>}1$) більшим за `a`, несуттєво, бо у доведенні можна використовувати будь-яке відношення порядку у смислі розд. 2.6.3; наприклад, *«порівнювати n за звичайним числовим порівнянням та ігнорувати a»*; це відношення порядку, бо його антисиметричність та транзитивність слідують з антисиметричності та транзитивності звичайного числового порівняння.)

П. 1.2: оскільки при кожному рекурсивному виклику відбувається *цілочисельне* ділення на 2 невід'ємного цілого числа́, обов'язково отримаємо виклик, де $\texttt{n} = 0$, а це умова виходу.

П. 2: єдиною умовою виходу є `n==0`, у цьому випадку відбувається `return 1.0`. Це відповідає властивості степеню $a^0 = 1$.

(Див. також міркування щодо 0^0 у доведенні ітеративної версії.)

П. 3: якщо при па́рному `n` (де цілочисельне ділення на 2 дає такий самий результат, як звичайне) виклик `pow_rs(a*a, n/2)` повернув правильне значення степеню $(a^2)^{n/2}$, то це і є правильний результат, бо $(a^2)^{n/2} = a^n$. Інакше, тобто при непа́рному n, можна подати (не у програмі, а суто

в рамках доведення) непа́рне n як $n=2k+1$, причому са́ме це k й буде результатом цілочисельного ділення `n/2`. Тоді рекурсивний виклик у гілці, відповідній непа́рному n, поверне (правильне за індуктивним припущенням) значення $(a^2)^k$, а оскільки в цій гілці розгалуження відбувається ще домноження результату на a, то вийде $(a^2)^k \cdot a = a^{2k+1}$, що якраз і є a^n. Оскільки інших гілок розгалуження, крім вже розглянутих, в алгоритмі нема, виходить, що `pow_rs(a, n)` завжди правильно повертає a^n.

Для рекурсивної та ітеративної версій швидкого піднесення до степеню потрібні окремі незалежні доведення, причому не лише тому, що для рекурсії інші засоби доведення, ніж для циклу, а ще й тому, що версії лише схожі, але не еквівалентні. Наприклад, при обчисленні a^{50} рекурсивна версія буде обчислювати проміжне значення a^{48}, якого нема в ітеративній версії, але не буде обчислювати a^{18}, яке там є.

3.4 Завдання до розділу 3

1. Довести (методом матіндукції) $1+2+\ldots+n=\frac{n(n+1)}{2}$.
2. Довести (методом матіндукції) $(n+1)! \geqslant 2^n$ (де "!" означає факторіал).
3. Довести (методом матіндукції) $1+3+5+\ldots+2n-1=n^2$.
4. Довести (методом матіндукції) $2^0+2^1+2^2+\ldots+2^n=2^{n+1}-1$.
5. Довести (методом матіндукції) $(4^n+2) \vdots 3$ (де "$\vdots$" — «кратне»).
6. Довести (методом матіндукції) $\left(11 \cdot 3^{2n}+10 \cdot 2^n\right) \vdots 7$.
7. Довести (методом матіндукції) $(4^n+15n-1) \vdots 9$.
8. Знайти у теоретичних викладках розд. 1 «Вступ до математичної логіки» доведення, яке по суті являло собою застосування матіндукції (просто там не вживався цей термін).
9. Довести (методом матіндукції), що число, десятковий запис якого складається з 3^n одиниць, ділиться націло на 3^n. (Зокрема, при $n=1$ отримуємо $111 \vdots 3$, а при $n=2$ отримуємо $\underbrace{111111111}_{9} \vdots 9$.)

   Застереження. Переконатися, що Ваше доведення не намагається доводити чи використовувати твердження «аналогічно до 3 та 9, число ділиться на 3^n тоді й тільки тоді коли на 3^n ділиться сума цифр». *Це неправда.* Наприклад, 1899: $(1+8+9+9)=27$, але $1899=27 \cdot 70++9$, тобто $1899 \,\overline{\vdots}\, 27$. (Що не заважає бути правильним тому твердженню, яке просять довести, бо там йдеться *не* про суму цифр.)
10. Довести (методом матіндукції), що у матриці
$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$
елементи 1-го рядка являють собою n-е та $(n-1)$-е чи́сла Фібоначчі.
11. З яким са́ме матеріалом цього розд. 3 можна поєднати попереднє завдання, щоб отримати неочевидний ефективний алгоритм знаходження чисел Фібоначчі?
12. Провести формальне доведення правильності стандартного простого ітеративного способу побудови чисел Фібоначчі (суть способу згадати або знайти у літературі з програмування).
13. Провести формальне доведення правильності стандартного простого рекурсивного способу побудови чисел Фібоначчі (суть способу згадати або знайти у літературі з програмування). Звернути увагу на п. 1.2, котрий тут виявляється складнішим, ніж у досі розглянутих прикладах. Пояснити, чому такий спосіб має суто теоретичне значення, а з практичної точки зору він жахливий.
14. Чому засоби цього розділу непридатні для доведення правильності пошуку вглиб, що розглядається у розд. 5.6.2?

    (Це *просте* питання, для відповіді на яке не обов'язково розуміти отой пошук углиб, досить розуміння цього розділу і *побіжного* перегляду коду функції `dfs` та коментарів до неї на стор. 134.)

Додаткові завдання підвищеного рівня складності

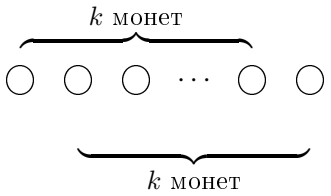
1*. Довести нерівність $2^n \geqslant n^2 - 1$, використавши ідеї методу математичної індукції, але дещо модифікувавши їх.

2*. Вказати помилку в «доведенні», ніби всі натуральні чи́сла рівні:
«Доведемо $n = n + 1$ для $n \in \mathbb{N}$. Спираючись на $k = k + 1$, дуже легко (додавши по одиничці до обох сторін рівності) отримати $k + 1 = (k + 1) + 1$, тобто $P(k) \to P(k + 1)$; отже, будь-які два «сусідні» числа́ рівні між собою, а звідси — всі натуральні чи́сла рівні між собою.»
Це «доведення» *містить* помилку, воно *неправильно* використовує метод математичної індукції. Суть завдання — пояснити, в чому са́ме полягає помилка.

3*. Довести (методом матіндукції) $\sqrt{n + \sqrt{n - 1 + \ldots + \sqrt{3 + \sqrt{2 + \sqrt{1}}}}} < \sqrt{n} + 2 \quad (n \in \mathbb{N})$.

4*. У самому кінці розд. 2.6.5 (стор. 72) заявлено, що рефлексивно-транзитивне замикання відношення, заданого матрицею R, можна будувати хоч як $I_A \cup R \cup R^2 \cup R^3 \cup \ldots \cup R^{n-1}$, хоч як $(I_A \cup R)^{n-1}$. Довести еквівалентність цих формул методом матіндукції.

5*. Знайти помилку в «доведенні», ніби всі монети мають однаковий номінал.
«Переформулюємо твердження, щоб воно залежало від n: «яку б не взяли сукупність з n монет, усі вони мають однаковий номінал». База індукції ($n = 1$) виконується: якщо взяти групу з єдиної монети, її номінал однаковий сам із собою.
Крок індукції. Як завжди, вважаємо, що при $n = k$ твердження виконується, тобто будь-які k монет мають однаковий номінал. Тоді можна розглянути будь-яку групу з $(k + 1)$ монет, і виділити у ній підгрупи з k монет двома способами (див. рис.).
Перша й дру́га монети мають однаковий номінал, бо належать одній групі з k монет (виділено згори); дру́га й остання монети теж мають однаковий номінал, бо належать одній групі з k монет (виділено знизу). Отже, перша й остання монети теж мають однаковий номінал, а звідси — всі монети групи розміром $k + 1$ мають однаковий номінал.



Таким чином (раз вдалося побудувати і базу, і крок) твердження правильне для всіх $n \in \mathbb{N}$; загальна кількість монет, які існують у світі, є натуральним числом; звідси — всі монети мають однаковий номінал.»
Це «доведення» *містить* помилку, воно *неправильно* використовує метод матіндукції. Але чітко вказати цю помилку не так просто; в цьому й полягає суть завдання...

6*. Провести формальне доведення правильності алгоритму рекурсивної версії принципу включень та виключень (розд. 4.3, стор. 99).

7*. Придумати ефективний (час роботи лінійно проопорційний розміру вхідних даних) алгоритм вирішення такої задачі *та довести його правильність*.
«Солдати інопланетної армії перед походом шикуються у шеренгу, повертаючись до командира або правим, або лівим боком. За командою вони починають готуватися до руху. Якщо двоє сусідніх солдатів повернуті обличчями один до одного, обоє розвертаються на 180° за 1 с. Розвороти різних пар солдатів відбуваються одночасно. За початковим розташуванням солдатів потрібно визначити, чи зможе армія коли-небудь вирушити у похід, і якщо так, то через скільки секунд і яку загальну кількість розворотів виконають усі солдати. Початкове розташування задається зі стандартного входу в єдиному рядку послідовністю символів "<" і ">": "<" означає, що солдат стоїть обличчям ліворуч, ">" — праворуч. Результати вивести на стандартний вихід у одному рядку: якщо армія зможе вирушити у похід, то (через пробіл) час і загальну кількість розворотів, якщо не зможе — слово `infinite`.»

Приклади:

Вхід	*Вихід*
>><<	*3 4*
<>	*0 0*
>><><	*3 5*

(Задача також доступна для online-перевірки: `www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=729`*)*

4 Комбінаторика

Під терміном «комбінаторика» у різних джерелах мають на увазі дещо різні речі. Будемо вважати, що *комбінаторика* — це розділ дискретної математики, що вивчає способи підрахунку кількостей різних наборів, які задовольняють певним правилам (підрахунок мусить бути швидшим та/або простішим за перебір всіх конкретних значень наборів). У ширшому розумінні, комбінаторика може включати також і способи генерації таких наборів та дослідження ще деяких їх властивостей. Але цей посібник зосереджується на вищезгаданому вужчому розумінні.

Приклад класичної задачі комбінаторики: *«У змаганнях беруть участь 8 команд. Скільки може бути різних результатів, якщо істотні лише призові місця (1-е, 2-е, 3-є), але для призових місць суттєво, хто 1-ий, хто 2-ий і хто 3-ій?»*. Відповідь на цю задачу — кількість розміщень з 8 по 3, тобто $A_8^3 = 8 \cdot 7 \cdot 6 = 336$.

Але це «занадто класична» задача, що зводиться до застосування стандартної формули A_n^k. Комбінаторика *не* зводиться до кількох стандартних формул. І для розуміння суті «усієї» комбінаторики варто почати з основних правил — *правила суми* та *правила добутку*. Са́ме на них (та ще на деякі прийоми) спирається дуже значна частина комбінаторики, в т. ч. й стандартні формули кількостей перестановок P_n, розміщень A_n^k та сполучень C_n^k.

4.1 Правила суми та добутку

Правило суми (рос. «*правило суммы*», англ. «*rule of sum*», «*addition principle*»). Розглянемо два еквівалентні формулювання.

А) Нехай усі способи можна розподілити по n групам так, що кожен спосіб належить до рівно однієї з цих груп (тобто, нема ні способу, не належного жодній з груп, ні способу, належного відразу кільком групам). Нехай відомі кількості способів у кожній групі: k_1, k_2, …, k_n. Тоді загальна кількість способів дорівнює $k_1 + k_2 + \ldots + k_n$.

Б) Якщо деяка множина розбита (див. стор. 65) на скіче́нну кількість скіче́нних класів та відомі кількість класів n та кількості елементів кожного з цих класів k_1, k_2, …, k_n, то загальна кількість елементів усієї множин́и дорівнює $k_1 + k_2 + \ldots + k_n$.

Приклад. Умова: *«Є 3 зелених чашки і 4 червоних; скількома способами можна вибрати одну з цих чашок?»* Розв'язок: чашка не може бути одночасно зеленою й червоною, інших чашок нема, значить діє правило суми й кількість таких способів 3+4=7.

Правило добутку (рос. «*правило произведения*», англ. «*rule of product*», «*multiplication principle*»). Розглянемо три еквівалентні формулювання.

А) Нехай скла́дена дія містить n послідовних етапів, перший етап можна виконати k_1 способами, другий — k_2 способами, …, і для виконання дії в цілому слід виконати будь-яким способом 1-ий етап, потім будь-яким способом 2-ий етап, і т. д. Тоді дію в цілому можна виконати $k_1 \cdot k_2 \cdot \ldots \cdot k_n$ способами.

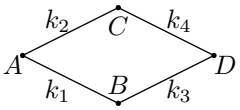
Б) Нехай набір складається з n частин, причому в якості першої частини можна взяти будь-який з k_1 предметів 1-ого типу, другої — будь-який з k_2 предметів 2-ого типу, і т. д. Нехай до того ж у наборі можна поєднувати будь-який предмет 1-ого типу з будь-яким предметом 2-ого і т. д. Тоді набір в цілому можна зібрати $k_1 \cdot k_2 \cdot \ldots \cdot k_n$ способами.

В) Нехай є скінче́нні множи́ни A_1, A_2, …, A_n, причому A_1 містить k_1 елементів, A_2 містить k_2 елементів, і т. д., A_n містить k_n елементів. Тоді декартів добуток $A_1 \times A_2 \times \ldots \times A_n$ містить $k_1 \cdot k_2 \cdot \ldots \cdot k_n$ елементів-енок.

Приклад. Нехай є 3 тарілки і 4 ложки (причому, як то зазвичай буває в гуртожитках, серед них нема однакових). Тоді є $3 \cdot 4 = 12$ різних способів скласти набір з тарілки та ложки:

$$\begin{array}{llll} \langle\text{тарілка } 1, \text{ложка } 1\rangle, & \langle\text{тарілка } 1, \text{ложка } 2\rangle, & \langle\text{тарілка } 1, \text{ложка } 3\rangle, & \langle\text{тарілка } 1, \text{ложка } 4\rangle, \\ \langle\text{тарілка } 2, \text{ложка } 1\rangle, & \langle\text{тарілка } 2, \text{ложка } 2\rangle, & \langle\text{тарілка } 2, \text{ложка } 3\rangle, & \langle\text{тарілка } 2, \text{ложка } 4\rangle, \\ \langle\text{тарілка } 3, \text{ложка } 1\rangle, & \langle\text{тарілка } 3, \text{ложка } 2\rangle, & \langle\text{тарілка } 3, \text{ложка } 3\rangle, & \langle\text{тарілка } 3, \text{ложка } 4\rangle. \end{array}$$

Приклад. Нехай з міста A до міста D можна дістатися або через місто B, або через місто C. Причому, дістатися з A до B можна k_1 способами, з A до $C-k_2$ способами, з B до $D-k_3$ способами, з C до $D-k_4$ способами. Скільки є різних способів дістатися з A до D?



Оскільки всі дороги проходять або через B, або через C, тут справедливе правило суми: треба порахувати кількість шляхів через B, кількість шляхів через C і результати додати. Кількість різних шляхів з A до D через B можна визначити за правилом добутку, бо кожен спосіб дістатися з A до B можна комбінувати з кожним способом дістатися з B до D. Аналогічно зі шляхами з A до D через C. Отже, остаточна відповідь задачі: $k_1k_3+k_2k_4$.

Приклад. «АТП має у розпорядженні k_1 водіїв однакової кваліфікації, k_2 сідельних тягачів, k_3 напівпричепів, k_4 вантажних фургонів.

У документації на конкретне перевезення (яке відбуватиметься одним транспортним засобом) мають бути вказані всі дані про цей Т.З. та про водія. Скільки всього може бути різних варіантів такої документації?»

Спочатку розберемося з транспортними засобами (Т.З.). Це *або* фургон, *або* тягач *із* напівпричепом. Всі способи сформувати «тягач *із* напівпричепом» передбачають, що можна поєднувати будь-який тягач з будь-яким напівпричепом, тобто діє правило добутку й виходить $k_2{\cdot}k_3$ способів. До них *додається* k_4, бо *або* фургон, *або* тягач із напівпричепом. Отже, варіантів вибору одного Т.З. є $k_2{\cdot}k_3+k_4$. Водіїв k_1, і вони однакової кваліфікації — за відсутності додаткової інформації слід трактувати, що будь-якого водія можна призначити на будь-який Т.З., а це правило добутку. Таким чином, остато́чна відповідь: $k_1\cdot(k_2{\cdot}k_3+k_4)$.

Різниця й частка використовуються в комбінаториці рідше, ніж сума й добуток, і їх не прийнято виносити у стандартні правила. Тим не менш, іноді вони все ж використовуються. Приклади використання частки можна знайти, зокрема, у доведеннях з розд. 4.2.3 та 4.2.4, і в цьому посібнику це називається «правилом частки», але в лапках, бо такий термін все ж *не* є загальноприйнятим. Віднімання використовується, зокрема, у розд. 4.3.

Що з цими правилами робити далі? Перш за все, запам'ятати *умови, коли* слід додавати, й *умови, коли* слід множити (а не лише «правило суми додає, а правило добутку множить»). При розв'язуванні задач підбирати правило, *перевіряючи, чи виконуються* ці умови.

Чотири при́клади застосувань цих правил вже наведено вище. Тепер розглянемо при́клади, де застосовувати ці правила не варто.

Наприклад, «задача» *«У магазині є у продажу 4 телефони фірми Samsung та 6 телефонів розміру 5 дюймів. Скільки всього є способів купити телефон у цьому магазині?»* не є коректною. Деяка схожість на правило суми тут є (бувають такі телефони, бувають сякі телефони, скільки є способів вибрати серед них усіх один телефон?). Але якщо деякі з телефонів одночасно і фірми Samsung, і розміру 5 дюймів, вони будуть враховані двічі, хоча треба один раз. А телефони одночасно іншого виробника та іншого розміру взагалі не будуть враховані. Тобто, умови застосовності правила суми тут порушені, причому аж двічі.

Сумнівною є також коректність «задачі» *«У селі 20 неодружених парубків та 25 незаміжніх дівчат. Скількома способами можна вибрати пару, що може взяти шлюб?»*. Поєднання в парі чоловіка та жінки створює деяку схожість на правило добутку. Але такий добуток передбачає «кожного можна поєднувати з кожною», чим ігноруються і бажання цих людей, і законодавча заборона укладання шлюбу між близькими родичами. Що якраз і є порушенням умов застосовності правила добутку.

На жаль, деякі задачі з комбінаторики якраз «грішать» нехтуванням такого роду додаткових обмежень... Наприклад, у розглянутій задачі, де діставалися з A у D або через B, або через C, ігнорувалося, що таке оті k_1 способів з A в B, k_3 способів з B у D, тощо. Якщо це регулярні рейси

громадського транспорту, то цілком можливо, що між деякими з них пересадки зручні, а між деякими — жахливі (наприклад, потрібно надто довго чекати, або, навпаки, завеликий ризик не встигнути на пересадку). Коли застосовується правило добутку, мається на увазі поєднання кожного рейсу з кожним, і питання зручності пересадок ігнорується.

Аналогічно, в реальних АТП можуть не любити перечеплювати напівпричепи чи пересаджувати водіїв на іншу машину; це може навіть порушувати якісь технічні регламенти, трудові угоди, тощо. Але там питали про *всі* можливі варіанти документації, не вказуючи ніяких додаткових обмежень; з цих міркувань, розв'язок правильний.

Питання про відповідність між математичною моделлю і практикою взагалі складне́ і філософське. У комбінаториці часто доводиться мати справу з текстовими задачами (сформульованими фразами природньою мовою), тому тут це проявляється відносно часто. Але при бажанні можна шукати невідповідності у застосуваннях самих різних галузей математики. Навіть до рівності «1+1=2» можна висловити претензії. Претензія №1: *«якщо одна з "1" означає купівлю цілого одного автомобіля, а інша"1" — купівлю всього-навсього одного блокнотика, хіба нормальна людина казатиме, що зроблено дві покупки?»*. Претензія №2: *«коли один плюс один, то двоє, а коли один плюс одна, то може ж вийти й більше!»*.

Але все це — не привід скасовувати дію додавання й оголошувати неправильною рівність «1+1=2»... Так і з комбінаторикою — треба просто вивчити, в яких (насправді, дуже багатьох) ситуаціях її результати доречні, а в яких (насправді, окремих) — не дуже.

4.2 Основні стандартні типи виборок

4.2.1 Перестановки

Перестановками (рос. «*перестановки*», англ. «*permutations*») називають послідовності, що відрізняються порядком, складаючись з одного й того ж набору елементів. Повний перелік перестановок трьох елементів: $\langle 1,2,3\rangle$; $\langle 1,3,2\rangle$; $\langle 2,1,3\rangle$; $\langle 2,3,1\rangle$; $\langle 3,1,2\rangle$; $\langle 3,2,1\rangle$. Класичний приклад задачі на кількість перестановок: *«Скількома різними способами можна розставити книжки на полиці?»*

Кількість перестановок n-елементної множини́ позначається P_n і визначається формулою

$$P_n = n\cdot(n-1)\cdot\ldots\cdot 2\cdot 1 = n! \tag{56}$$

(Позначення P_n виявилося довшим за його значення $n!$, тому на практиці зазвичай зразу пишуть $n!$)

Доведення. Зазвичай, цю формулу доводять так. Будемо отримувати перестановки, спочатку обираючи, який елемент займе перше місце, потім елемент на дру́ге місце, і т. д. На перше місце можна поставити будь-який з n елементів, на дру́ге — будь-який з $(n-1)$ решти (крім уже використаного на першому місці), і т. д., доки не дійдемо до останнього місця, для якого лишається єдиний елемент. Оскільки послідовність в цілому визначається значеннями всіх елементів, то, за правилом добутку, отримуємо (56).

(Строго кажучи, класичне формулювання правила добутку передбачає, що у різних наборах (на різних етапах) предмети (дії) обираються з незалежних і заранє відомих множин. А зараз набір предметів, які можна використати на дру́гому місці послідовності, залежить від того, який предмет розмістили на першому, і т. д. Але вибір конкретних елементів для початкових місць не впливає на *кількість* решти елементів. Тому використання добутку все ж виправдане, і формула (56) все-таки правильна.) ■

4.2.2 Розміщення

Розміщеннями з n по k (рос. «*размещения из n по k*») називають виборки по k елементів з n можливих, причому порядок елементів у виборці суттєвий. Повний перелік розміщень з 4 по 2: $\langle 1,2\rangle$; $\langle 1,3\rangle$; $\langle 1,4\rangle$; $\langle 2,1\rangle$; $\langle 2,3\rangle$; $\langle 2,4\rangle$; $\langle 3,1\rangle$; $\langle 3,2\rangle$; $\langle 3,4\rangle$; $\langle 4,1\rangle$; $\langle 4,2\rangle$; $\langle 4,3\rangle$. Класичний приклад задачі на кількість розміщень уже згадувався: *«У змаганнях беруть участь 8 команд. Скільки може бути різних результатів, якщо істотні лише призові місця (1-е, 2-е, 3-є), але для призових місць суттєво, хто 1-й, хто 2-й і хто 3-й?»*.

(Англ. мовою це перекладають як «*k-permutations of n-element sequence*», тобто (враховуючи, що «permutations» — перестановки) окремого терміну для цього поняття нема.)

Кількість розміщень з n по k за (пост)радянською традицією позначається A_n^k, а за американською $[n]_k$, і визначається формулою

$$A_n^k = n\cdot(n-1)\cdot\ldots\cdot(n-k+1) = \frac{n!}{(n-k)!}. \tag{57}$$

Другий вигляд формули (частка факторіалів) зручніший для аналітичних перетворень, а для обчислень безумовно краще перший (добуток).

Доведення. Аналогічно попередньому, але спиняємося на k-ому місці, де можна розмістити будь-який з $(n-k+1)$ решти предметів. ■

4.2.3 Сполучення

Сполученнями (або *комбінаціями*) з n по k (рос. «*сочетания из n по k*»; англ. «*k-combinations of n-element set*») називають виборки по k елементів з n можливих, якщо порядок елементів у виборці не істотний (тобто виборки, що складаються з однакових елементів, вважаються однаковими незалежно від порядку елементів). Повний перелік різних сполучень з 4 по 2: $\{1,2\}$; $\{1,3\}$; $\{1,4\}$; $\{2,3\}$; $\{2,4\}$; $\{3,4\}$. Класичний приклад задачі на кількість сполучень: «*З n учнів треба вибрати k учнів у екскурсійну групу. Скількома різними способами це можна зробити, якщо суттєво тільки те, хто потрапляє до групи, а хто ні?*».

Кількість сполучень з n по k за (пост)радянською традицією позначається C_n^k, а за американською — $\binom{n}{k}$ і визначається формулою[28]

$$C_n^k = \frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{1\cdot 2\cdot\ldots\cdot k} = \frac{n!}{k!\cdot(n-k)!}. \tag{58}$$

І знову, другий вигляд формули зручніший для аналітичних перетворень, а для обчислень безумовно краще перший. Причому, зазвичай доцільно спочатку порахувати `(n*(n-1))/2`, потім помножити на `(n-2)`, потім поділити цілочисельно на `3`, і т. д. Це дозволяє, лишаючись у рамках цілочисельної арифметики, у значній мірі уникнути великих проміжних значень при не дуже великому остаточному результаті.

Доведення. Враховуючи (57), достатньо довести $C_n^k = \frac{A_n^k}{k!}$.

Розглянемо всі сполучення й усі розміщення для деяких (довільно вибраних, але однакових протягом усього доведення) n і k. Кожному сполученню відповідає група розміщень (наприклад, сполученню $\{2,3\}$ — група розміщень $\langle 2,3\rangle$ і $\langle 3,2\rangle$). Різних розміщень, відповідних одному сполученню, $k!$, бо всі можливі перестановки розміщення є іншими розміщеннями (перестановки міняють порядок $\Rightarrow$ розміщення інше), відповідаючи тому са́мому сполученню (перестановки не міняють набір $\Rightarrow$ сполучення те са́ме).

Кожне окремо взяте розміщення мусить потрапити в деяку групу, причому лише в одну (адже розміщення складається з деяких k елементів, отже вони однозначно визначають його групу). Тобто, відомо, що в кожній групі однаково по $k!$ розміщень, а сумарна по всім групам кількість розміщень становить A_n^k. Значить, кількість груп (котрі якраз і відповідають сполученнями) дорівнює $A_n^k/k!$. ■

Схожі міркування час від часу використовуються у комбінаториці. Ми навіть будемо називати їх «*правилом частки*», але пишучи в лапках, бо це *не* є загальноприйнятим терміном.

Виразити формулювання «правила частки» більш загально можна, наприклад, так: «*Нехай відома загальна кількість деяких виборок a, і відомо, що всі ці виборки утворюють розбиття (у смислі стор. 65) на класи однакового розміру b. Тоді кількість таких класів дорівнює a/b.*». Можна сформулювати й без посилань на матеріал розд. 2.6.2, але тоді треба пояснювати не лише те, що класи повинні бути одного розміру, а й те, що кожен елемент мусить належати рівно одному класу (не більше й не менше).

4.2.4 Перестановки з повтореннями

Перестановками з повтореннями складу $(k_1,k_2,\ldots,k_n)$ (рос. «*перестановки с повторениями состава* $(k_1,k_2,\ldots,k_n)$») називають послідовності довжиной $k_1+k_2+\ldots+k_n$, які містять k_1 елементів першого типу, k_2 — другого, …, k_n — n-го типу і відрізняються одна від одної порядком слідування елементів різних типів. Повний перелік перестановок з повтореннями складу $(2,3)$ (де 2 і 3 — кратності елементів a і b відповідно): $\langle a,a,b,b,b\rangle$, $\langle a,b,a,b,b\rangle$, $\langle a,b,b,a,b\rangle$, $\langle a,b,b,b,a\rangle$, $\langle b,a,a,b,b\rangle$, $\langle b,a,b,a,b\rangle$, $\langle b,a,b,b,a\rangle$, $\langle b,b,a,a,b\rangle$, $\langle b,b,a,b,a\rangle$, $\langle b,b,b,a,a\rangle$.

(Англійською мовою нема стандартного відповідника словосполученню «складу $(k_1,k_2,\ldots,k_n)$», а перестановки з повтореннями називають «*permutations of multiset*», вказуючи, якої мультимножини́ це перестановки: такої, що містить k_1 елементів a_1, k_2 елементів a_2, …, k_n елементів a_n. Що кінець кінцем те са́ме, навіть зрозуміліше.)

[28] зверніть увагу, що у (пост)радянському записі загальна кількість елементів n знизу, кількість обраних елементів k згори, а в американському навпаки

Класичний приклад задачі на кількість перестановок з повтореннями: *«Скільки різних "слів" (послідовностей букв) можна отримати, переставляючи (але не видаляючи і не додаючи) букви у слові "математика"?»*;

Кількість перестановок з повтореннями складу $(k_1, k_2, \ldots, k_n)$ позначається $\overline{P}(k_1, k_2, \ldots, k_n)$ і визначається формулою

$$\overline{P}(k_1, k_2, \ldots, k_n) = \frac{(k_1 + k_2 + \ldots + k_n)!}{k_1! \cdot k_2! \cdot \ldots \cdot k_n!}. \tag{59}$$

Доведення. «Понавішуємо» на невідрізнювані об'єкти додаткові «мітки», щоб вони стали відрізнюваними; тоді кількість різних виборок суть кількість («звичайних») перестановок, тобто $(k_1 + k_2 + \ldots + k_n)!$.

Кожній перестановці з повтореннями відповідають $k_1! \cdot k_2! \cdot \ldots \cdot k_n!$ («звичайних») перестановок. (Наприклад, на рис. наведені всі $3! \cdot 2!$ перестановок, різних з урахуванням «міток», але однакових з точки зору перестановок з повтореннями (при $k_1 = 3$, $k_2 = 2$).)

$$3! \left\{ \begin{array}{ccccc|ccccc} a_1 & b_1 & a_2 & a_3 & b_2 & a_1 & b_2 & a_2 & a_3 & b_1 \\ a_1 & b_1 & a_3 & a_2 & b_2 & a_1 & b_2 & a_3 & a_2 & b_1 \\ a_2 & b_1 & a_1 & a_3 & b_2 & a_2 & b_2 & a_1 & a_3 & b_1 \\ a_2 & b_1 & a_3 & a_1 & b_2 & a_2 & b_2 & a_3 & a_1 & b_1 \\ a_3 & b_1 & a_1 & a_2 & b_2 & a_3 & b_2 & a_1 & a_2 & b_1 \\ a_3 & b_1 & a_2 & a_1 & b_2 & a_3 & b_2 & a_2 & a_1 & b_1 \end{array} \right. \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{2!}$$

Отже, тут застосовне «правило частки» (стор. 96), що й доводить формулу (59). ■

4.2.5 Розміщення з повтореннями

Розміщеннями з повтореннями з n по k (рос. «*размещения с повторениями из n по k*») називають виборки по k елементів з n можливих, причому: один і той самий елемент дозволяється вибирати багатократно; порядок елементів у виборці істотний. Повний перелік розміщень з повтореннями з 3 по 2: $\langle 1,1\rangle$; $\langle 1,2\rangle$; $\langle 1,3\rangle$; $\langle 2,1\rangle$; $\langle 2,2\rangle$; $\langle 2,3\rangle$; $\langle 3,1\rangle$; $\langle 3,2\rangle$; $\langle 3,3\rangle$. Легко бачити, що це по суті сукупність взагалі всіх можливих послідовностей. Томý, англійською мовою прийнято називати цей тип виборки «*strings*» (рядки). Приклад задачі на кількість розміщень з повтореннями: *«Скільки може бути різних серій заводського номера деякого виробу, якщо серія складається з трьох латинських літер (які можуть повторюватися)?»*. (Тут розмір виборки $k = 3$, кількість можливих елементів $n = 26$.)

Кількість розміщень з повтореннями позначається $\overline{A}_n^k$ і визначається формулою

$$\overline{A}_n^k = n^k. \tag{60}$$

Доведення. Формула випливає безпосередньо з правила добутку: є k позицій, у кожній з яких може перебувати будь-яке з n можливих значень. ■

4.2.6 Сполучення з повтореннями

Сполученнями з повтореннями з n по k (рос. «*сочетания с повторениями из n по k*»; англ. «*k-combinations with repetitions of n-element set*», «*k-multicombinations of n-element set*») називають виборки по k елементів з n можливих, причому: один і той самий елемент дозволяється вибирати багатократно; порядок елементів у виборці не істотний, тобто виборки, які відрізняються лише порядком елементів, вважаються однаковими. Повний перелік сполучень із повтореннями з 3 по 2: $\{1,1\}$, $\{1,2\}$, $\{1,3\}$, $\{2,2\}$, $\{2,3\}$, $\{3,3\}$. Приклад задачі на кількість сполучень із повтореннями: *«Скількома способами можна купити 5 штук булочок у магазині, де продають 8 видів булочок?»* (вибираються k=5 елементів з n=8 можливих).

(Для виборок із повтореннями осмислений також і випадок $k > n$: ніщо не заважає купити 3 штуки булочок у магазині, що торгує лише 2-ма видами; тільки й того, що повтори виявляться не просто можливі, а неминучі. Способами такої покупки будуть сполучення з повтореннями з 2 по 3: $\{1,1,1\}$, $\{1,1,2\}$, $\{1,2,2\}$, $\{2,2,2\}$.)

Кількість сполучень із повтореннями позначається за (пост)радянською традицією $\overline{C}_n^k$, за американською $\left(\!\binom{n}{k}\!\right)$ і визначається формулою

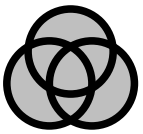
$$\overline{C}_n^k = C_{n+k-1}^k = \frac{(n+k-1)!}{k!(n-1)!}. \tag{61}$$

(Доведення пропустимо.)

(Для сполучень з повтореннями та розміщень з повтореннями $\overline{C}_n^k \neq \overline{A}_n^k/k!$; аналогічної частки з яким би не було іншим знаменником теж *нема*. Наприклад, сполученню з повтореннями $\{1,1,2,3,4\}$ відповідають $\overline{P}(2,1,1,1)=60$ різних розміщень з повтореннями, а наприклад, сполученню з повтореннями $\{1,1,2,2,2\}$ лише $\overline{P}(2,3)=10$ розміщень з повтореннями, тобто групи (класи еквівалентності) виходять різних розмірів. А це значить, що умови застосовності «правила частки» (стор. 96) *не* виконуються.)

Повторимо, що у прийнятій у посібнику (пост)радянській системі позначень кількості виборок без повторень позначаються P, A, C, а аналогічних їм виборок з повтореннями — $\overline{P}$, $\overline{A}$, $\overline{C}$. Тобто, наявність чи відсутність верхньої риски впливає на смисл та числове значення.

4.3 Принцип включень та виключень; його рекурсивна реалізація



Нехай є три круги A, B, C, які можуть перетинатися. Все, що потрапило до цих кругів, замальовується (причому, щільність замальовування однакова і для просто кругів, і для їхніх перетинів). Тоді площу замальованої частини площини можна порахувати таким чином:

1) Спочатку порахуємо суму площ кругів: $S = S(A)+S(B)+S(C)$.

2) Перетини кругів були пораховані двічі, тому їх треба відняти: $S -= \big(S(A\cap B)+S(A\cap C)+$ $+S(B\cap C)\big)$;[29]

3) Перетин усіх трьох кругів спочатку тричі додали, а потім тричі відняли, тобто не врахували; отже, його треба додати: $S += S(A\cap B\cap C)$.

Або, все разом і у чисто-математичних позначеннях, $S(A\cup B\cup C) = S(A)+S(B)+S(C)-$ $-S(A\cap B)-S(A\cap C)-S(B\cap C)+S(A\cap B\cap C)$.

Ці міркування являють собою частковий випадок так званого *принципу включень та виключень* (рос. «*принцип включений и исключений*», англ. «*inclusion–exclusion principle*»).

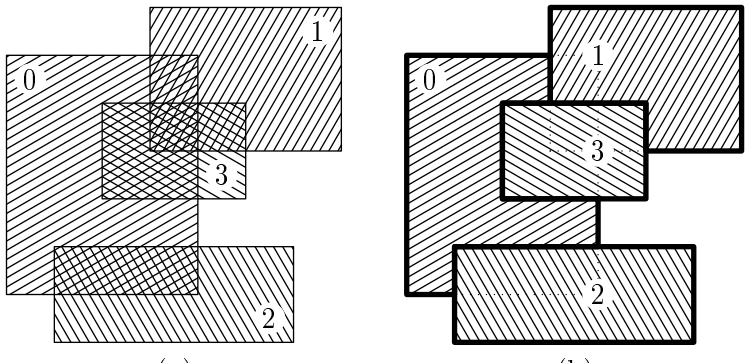
Більш повне формулювання принципу таке. Якщо є n множин A_1, A_2, …, A_n, то *міра* (кількість елементів, площа, …) об'єднання усіх цих множин $A_1\cup A_2\cup\ldots\cup A_n$ дорівнює сумі мір усіх окремо взятих множин, мінус міри всіх можливих попарних перетинів, плюс міри всіх можливих потрійних перетинів, мінус міри всіх можливих перетинів групами по чотири, і т. д., до міри перетину всіх n множин включно. Знаки так і продовжують чергуватися (+, −, +, −, …), так що міра перетину всіх n множин додається, якщо n непарне, і віднімається, якщо па́рне.

(Цей принцип можна строго довести, користуючись матіндукцією.)

Рекурсивна реалізація принципу включень та виключень. Ми навели словесне формулювання принципу включень та виключень для довільного n, але не стали писати відповідну аналітичну формулу, бо вона *дуже* складна і неприємна. Але виявляється, що цю неприємну формулу *можна* запрограмувати рекурсивною функцією.

Нехай кількість множин подається у (глобальній) змінній `int N`, самі́ множи́ни — у (глобальному) масиві `TMySet A[]` (де `TMySet` — власний тип, котрий якось подає множину і для котрого якось реалізована підпрограма знаходження перетину). Побудуємо функцію `double CountM(TMySet X, int i)`, котра рахує міру частини множи́ни `X`, що не перетинається з множи́нами, номери яких строго більші `i`.

```
res = 0;
for(int i=0; i<N; i++)
  res += countM(A[i],i);
```



(a) (b)

[29] Зверніть увагу, що тут не рівність, а `-=`, у смислі C/C++-подібних мов програмування

Тоді наведені на рис. багатократні виклики цієї функції призведуть до обчислення міри $A_0 \cup \cup A_1 \cup \ldots \cup A_{N-1}$. Справді, `countM(A[0],0)` обчислить міру всього того, що належить лише множині A_0 і жодній іншій множині (див. область 0 на рис. (b)); `countM(A[1],1)` — міру того, що належить A_1 і не належить $A_2, \ldots, A_{N-1}$ (не зважаючи на належність чи не належність до A_0) (див. область 1 на рис. (b)); і т. д.

Самá функція може мати приблизно такий вигляд:

```
double CountM(TMySet X, int i)
{
  if(X==∅)                    /* якщо множина порожня, то */
    return 0;         /* порожні будуть і подальші перетини */
  double res = |X|;           /* рахуємо міру самóї множинú */
  for(int j=i+1; j<N; j++)        /* віднімаємо перетини з */
    res -= CountM(X∩A[j], j); /* ‘‘подальшими’’ множинами */
  return res;
}
```

(Звичайно, це все не очевидно; але й безмежно складного нічого нема, цей алгоритм *можна* формально довести засобами розд. 3.3.

Цікаво відзначити, що в наведеній реалізації нема явного чергування знаків (+, −, +, −, ...), але воно (чергування) з'являється самé внаслідок того, що значення, які вертаються рекурсивними викликами, віднімаються: $a-(b-(c-\ldots))=a-b+c-\ldots$)

4.4 Біноміальні коефіцієнти та бінóм Ньютона

Біноміальними коефіцієнтами будемо називати кількості сполучень (без повторень) $C_n^k = \frac{n!}{k!(n-k)!}$. (Пізніше буде пояснено, чому.) Для них відомо багато тотожностей. Зокрема:

$$C_n^{n-k} = C_n^k, \tag{62}$$

$$C_{n-1}^{k-1} + C_{n-1}^k = C_n^k. \tag{63}$$

Доведення. Наведемо два доведення кожної тотожності — через аналіз комбінаторного смислу та через формальні перетворення формул.

1-А) Права частина формули (62) задає кількість способів вибрати k об'єктів серед n можливих. Але кожен спосіб вибрати k об'єктів взаємнооднозначно (бієктивно) відповідає способу вибрати $n-k$ об'єктів — узяти *решту* (все, що не було вибране). Значить, кількості таких виборок однакові.

1-Б) $C_n^{n-k} = \dfrac{n!}{(n-k)!(n-(n-k))!} = \dfrac{n!}{(n-k)!k!} = C_n^k$.

2-А) Виділимо серед n можливих об'єктів один, доля (судьба) котрого нас особливо цікавить. Кожну виборку k об'єктів серед (усіх) n можливих (їх C_n^k) можна віднести до одного з двох взаємовиключних випадків: або виділений об'єкт потрапляє до виборки, або не потрапляє. У першому випадку серед решти $n-1$ можливих об'єктів треба вибрати решту $k-1$ (C_{n-1}^{k-1} способів); у дрýгому всі k об'єктів треба вибирати серед решти $n-1$ можливих (C_{n-1}^k способів).

2-Б) $C_{n-1}^{k-1} + C_{n-1}^k =$ — стандартна формула (58)

$= \dfrac{(n-1)!}{(k-1)!((n-1)-(k-1))!} + \dfrac{(n-1)!}{k!((n-1)-k))!} =$ — $(n\underline{-1}) - (k\underline{-1}) = (n-k)$

$= \dfrac{(n-1)!}{(k-1)!(n-k)!} + \dfrac{(n-1)!}{k!(n-k-1)!} =$ — $(n-k)! = (n-k)\cdot(n-k-1)!$, $k! = k\cdot(k-1)!$

$= \dfrac{(n-1)!}{(k-1)!\cdot(n-k)\cdot(n-k-1)!} +$ — винесли спільне (великий дріб) за дужки

$+ \dfrac{(n-1)!}{k\cdot(k-1)!\cdot(n-k-1)!} =$

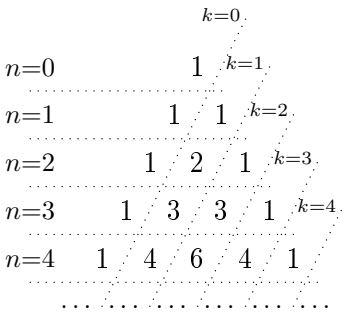
$= \dfrac{(n-1)!}{(k-1)!(n-k-1)!}\cdot\left(\dfrac{1}{n-k}+\dfrac{1}{k}\right) =$ — у дужках привели до спільного знаменника і додали

$= \dfrac{(n-1)!}{(k-1)!(n-k-1)!}\cdot\dfrac{k+(n-k)}{(n-k)k} =$ — $k+(n-k)=n$; попереносили окремі множники n, k, $n-k$ туди, де вони потрібніші

$= \dfrac{(n-1)!\cdot n}{(k-1)!\cdot k\cdot(n-k-1)!\cdot(n-k)} =$ — $n\cdot(n-1)! = n!$, $(n-k)\cdot(n-k-1)! = (n-k)!$

$= \dfrac{n!}{k!(n-k)!} = C_n^k$. — стандартна формула (58) ■

Тотожність (63) дає ключ до т. зв. *трикутника Паскаля* (рос. «*треугольник Паскаля*», англ. «*Pascal's triangle*»; названий на честь само́го Блеза Паскаля, а не мови). Це числовий трикутник (див. рис.), по краям якого записані одиниці, а решта елементів визначається як сума двох верхніх сусідніх. Якщо розпочати нумерацію рядків та чисел всере́дині рядка з нуля, то вийде, що k-е число у n-му рядку $= C_n^k$.



Біно́м Ньютона — це формула

$$(a+b)^n = \sum_{k=0}^{n} C_n^k a^k b^{n-k} \qquad (n \in \mathbb{N}). \tag{64}$$

Са́ме тому C_n^k і називають біноміальними коефіцієнтами: вони виявляються коефіцієнтами у правій частині цієї формули. Зокрема, шкільні «формули скороченого множення» $(a+b)^2 = a^2 + 2ab + b^2$ та $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$ є частковими випадками бінома Ньютона при $n=2$ та $n=3$. Щоб показати це чіткіше, перепишемо їх, наголошуючи на коефіцієнтах:

$$(a+b)^2 = \underbrace{1\cdot b^2}_{C_2^0=1} + \underbrace{2\cdot ab}_{C_2^1=2} + \underbrace{1\cdot a^2}_{C_2^2=1}; \quad (a+b)^3 = \underbrace{1\cdot b^3}_{C_3^0=1} + \underbrace{3\cdot ab^2}_{C_3^1=3} + \underbrace{3\cdot a^2b}_{C_3^2=3} + \underbrace{1\cdot a^3}_{C_3^3=1}.$$

Саме́ слово «біном» означає «сума чи різниця рівно двох величин», тобто біномом (просто біномом, а не біномом Ньютона) є вираз у дужках «$a+b$». Взагалі кажучи, формула (64) була відома й до Ньютона; Ньютон лише запропонував нові способи її доведення та узагальнення. Так що назва «біном Ньютона» традиційна, широко вживана, але не зовсім правильна. До речі, англійською мовою цю формулу зазвичай називають «*binomial expansion*» або «*binomial theorem*», без прямої згадки Ньютона.

Нарешті, наведемо доведення формули (64), знов двома різними незалежними способами.

Доведення. А) Очевидно, кожен доданок многочлену містить n співмножників — деяку кількість k множників a та $n-k$ множників b, тобто має вигляд $a^k b^{n-k}$, де $0 \leqslant k \leqslant n$. Кожен такий доданок взаємно однозначно відповідає підмножині номерів дужок, з яких для утворення доданку взятий співмножник a. Тому доданків $a^k b^{n-k}$ стільки, скільки таких підмножин, тобто C_n^k.

Б) Доведемо формулу (64) через метод математичної індукції.

При $n=1$ формула набуває вигляду $(a+b)^1 = C_1^0 \cdot a^0 \cdot b^1 + C_1^1 \cdot a^1 \cdot b^0$; оскільки $C_1^0 = C_1^1 = 1$ та $a^0 = b^0 = 1$, то обидві частини рівності перетворюються до $a+b$. Отже, при $n=1$ формула правильна. (База ok.)

Тепер, вважаючи рівність $(a+b)^k = \sum_{i=0}^{k} C_k^i a^i b^{k-i}$ гарантовано правильною, доведемо $(a+b)^{k+1} = \sum_{i=0}^{k+1} C_{k+1}^i a^i b^{k+1-i}$.

$(a+b)^{k+1} = (a+b)^k \cdot (a+b) =$ — замінимо $(a+b)^k$ на рівну (за припущенням кроку індукції) суму

$= \left(\sum_{i=0}^{k} C_k^i a^i b^{k-i}\right) \cdot (a+b) =$ — "$(a+b)$" не залежить від i, тому добуток суми на $(a+b)$ дорівнює сумі, в якій кожен доданок окремо помножений на $(a+b)$

$= \sum_{i=0}^{k} \left(C_k^i a^i b^{k-i}(a+b)\right) =$ — розкриємо дужки в добутку $a^i b^{k-i}(a+b)$

$= \sum_{i=0}^{k} \left(C_k^i (a^{i+1} b^{k-i} + a^i b^{k+1-i})\right) =$ — перегрупуємо доданки (аналогічно заміні $(x_1+y_1) + (x_2+y_2) + \ldots + (x_m+y_m)$ на $(x_1+x_2+\ldots+x_m) + (y_1+y_2+\ldots+y_m)$)

$= \sum_{i=0}^{k} C_k^i a^{i+1} b^{k-i} + \sum_{i=0}^{k} C_k^i a^i b^{k+1-i} =$ — у 1-ій сумі перейдемо до сумування по $j = i+1$; у 2-ій — перейменуємо i на j

$= \sum_{j=1}^{k+1} C_k^{j-1} a^j b^{k-(j-1)} + \sum_{j=0}^{k} C_k^j a^j b^{k+1-j} =$ — «відщепляємо» від 1-ої суми доданок при $j = k+1$, від 2-ої — при $j=0$

помічаємо, що $k-(j-1) = k+1-j$

$= C_k^k a^{k+1} b^0 + \sum_{j=1}^{k} C_k^{j-1} a^j b^{k+1-j} +$ $+ \sum_{j=1}^{k} C_k^j a^j b^{k+1-j} + C_k^0 a^0 b^{k+1} =$ — для $1 \leqslant j \leqslant k$ перегруповуємо доданки у спільну суму та виносимо за дужки спільний множник $a^j b^{(k+1)-j}$

$= C_k^k a^{k+1} b^0 + C_k^0 a^0 b^{k+1} +$ $+ \sum_{j=1}^{k} \left((C_k^{j-1} + C_k^j) a^j b^{k+1-j}\right) =$ — користуємося тотожністю $C_k^{j-1} + C_k^j = C_{k+1}^j$

помічаємо, що $C_k^k = 1 = C_{k+1}^{k+1}$, $C_k^0 = 1 = C_{k+1}^0$

$= C_{k+1}^{k+1} a^{k+1} b^0 + C_{k+1}^0 a^0 b^{k+1} +$ $+ \sum_{j=1}^{k} C_{k+1}^j a^j b^{k+1-j} =$ — помічаємо, що доданки, не включені в суму, рівні доданкам суми при $j = k+1$ та $j = 0$

$= \sum_{i=0}^{k+1} C_{k+1}^i a^i b^{k+1-i}.$ — що й треба було отримати (Крок ok) ■

На основі бінома Ньютона можна доводити інші тотожності з біноміальними коефіцієнтами. Наприклад,

$$C_n^0 + C_n^1 + \ldots + C_n^n = 2^n. \tag{65}$$

Доведення. Отримується з (64) при $a=b=1$. ■

(Формула (65) має також доведення, незалежне від бінома Ньютона: C_n^k являє собою кількість різних k-елементних підмножин; тоді $\sum_{k=0}^{n} C_n^k$ — кількість 0-елементних підмножин плюс кількість 1-елементних підмножин плюс і т. д. — тобто, кількість взагалі всіх підмножин. А кількість взагалі всіх підмножин 2^n.)

Відомо дуже багато інших тотожностей з біноміальними коефіцієнтами. Але ми обмежимося вже розглянутими — найбільш цікавими та корисними.

4.5 Рекурентні співвідношення

Розглянуті досі засоби комбінаторики не вичерпують можливі комбінаторні об'єкти. Тобто, всі ті засоби корисні, бо чимало задач якраз і слід розв'язувати, якимсь способом поєднуючи якісь вже розглянуті стандартні засоби.

Але буває корисно також і знати додаткові засоби.

Тому́, за́раз перейдемо до розгляду ще одного засобу комбінаторики: побудови та застосування *рекурентних співвідношень* (рос. «*рекуррентные соотношения*», англ. «*recurrence relations*»), тобто формул, де кількість об'єктів виражається через кількості об'єктів аналогічної структури менших розмірів. «Аналогічна структура менших розмірів» передбачає деяку *серію підзадач*, коли є допоміжна задача, що формулюється однаковими фразами, але для різних «частин» початкової задачі; наприклад, але зовсім не обов'язково — різних клітинок деякої таблиці.

Далі це буде частково пояснено на при́кладах, і настійливо рекомендується перечитати попередній абзац ще кілька разів по мірі ознайомлення з тими при́кладами. Але деталі все одно треба продумувати для кожної задачі окремо, бо правила цього підходу все одно надто загальні.

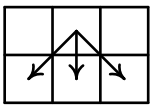
У деяких книжках, особливо орієнтованих на «чистих» математиків, до рекурентних співвідношень ставляться як до чогось проміжного, яке «треба» перетворити до аналітичного вигляду, а коли це не вдається, кажуть, що «розв'язати співвідношення не вдалося». Ми ж питання перетворення до аналітичного вигляду взагалі не будемо досліджувати, а ситуацію, коли для початкової текстової задачі комбінаторики зуміли побудувати рекурентне співвідношення, яке можна обчислити на комп'ютері (запрограмувавши або зробивши формули в електронних таблицях), вважатимемо цілком повноцінним розв'язком. І якщо таким чином вдаватиметься знайти розв'язок задачі, яку не зуміли розв'язати через аналітичні формули — вважатимемо це прикладами ситуацій, в яких рекурентні співвідношення виявилися кращими.

У деяких джерелах, особливо орієнтованих на олімпіадне програмування, рекурентні співвідношення комбінаторики називають «динамічним програмуванням» або «табличною технікою». На думку автора посібника, ті терміни гірші, бо назва «таблична техніка» надто загальна (за допомогою таблиць роблять дуже багато дуже різних дій), а «динамічне програмування» за першопочатковими уявленнями передбачало не підрахунок кількості, а максимізацію чи мінімізацію деякої цільової функції. Але такі назви цього підходу все ж досить поширені.

Розглянемо кілька задач комбінаторики, які зручно розв'язувати са́ме рекурентними співвідношеннями. Як вже сказано, деталі інших задачі все одно треба продумувати окремо. Втім, потреба ретельно думати над кожною задачею окремо взагалі характе́рна для комбінаторики...

4.5.1 Шляхи у таблиці

«Є прямокутна таблиця N рядків на M стовпчиків. Скільки є способів пройти по ній згори донизу, починаючи з будь-якої клітинки верхнього рядка, далі переходячи щоразу в одну з «нижньо-сусідніх» і закінчити маршрут у якій-небудь клітинці нижнього рядка?



Інакше кажучи, при нумерації рядків згори донизу, з клітинки (i,j) можна перейти у $(i{+}1,j{-}1)$, або у $(i{+}1,j)$, або у $(i{+}1,j{+}1)$, але не виходячи за межі таблиці, тобто при $j{=}1$ перший з цих варіантів стає неможливим, а при $j{=}M$ — останній.

Наприклад, для $N{=}2$, $M{=}3$ відповіддю є 7. Ці сім способів мають такий вигляд: 1)  ; *2)*  ; *3)*  ; *4)*  ; *5)*  ; *6)*  ; *7)*  .»

(Як розв'язати цю задачу іншими (крім рекурентних співвідношень) засобами, не ясно. При $M{=}2$ результат рівний 2^N, бо у кожному з N рядків можна бра́ти будь-який з 2 стовпчиків. Але це не узагальнюється на потрібну задачу, бо при більших значеннях M вже не можна перескакувати з будь-якого стовпчика у будь-який. Якби прямокутник був «закручений у циліндр», щоб крок праворуч з найправішого стовпчика приводив у найлівіший стовпчик (і, симетрично, ліворуч з найлівішого — у найправіший), результатом було б $M{\times}3^{N-1}$ (M варіантів, з якого стовпчика почати; далі, на кожному з $N{-}1$ кроків, 3 варіанти, до якої з нижньо-сусідніх перейти). Але для звичайного прямокутника так не виходить: з крайніх клітинок є два можливі продовження маршруту, з проміжних — три, і неясно, як це врахувати.)

Поставимо серію підзадач *«Скільки є різних способів дістатися від верхнього рядка до клітинки номер (i,j)?»* і позначимо ці кількості як $T(i,j)$.

Для кожної з клітинок 1-го (крайнього верхнього) рядка відповідь рівна 1 ($T(1,j)=1$), бо це одноклітинкові маршрути, без переходів, котрі закінчуються тут же, де почалися; таких маршрутів по одному для кожної клітинки.

Для клітинок подальших рядків, ці кількості можна обчислювати як

$$T(i,j) = T(i{-}1,j{-}1) + T(i{-}1,j) + T(i{-}1,j{+}1) \quad \begin{matrix}\text{(пропускаючи варіанти, які}\\ \text{виводять за межі таблиці)}\end{matrix} \tag{66}$$

тобто знаходити значення в усіх рядках, починаючи з 2-го, як суму двох чи трьох верхньо-сусідніх. Це правильно (правило суми застосовне), бо всі можливі шляхи до кожної такої клітинки приходять у неї з однієї з верхньо-сусідніх, ніякий шлях не може бути ні пропущений (не можна прийти нізвідки, крім них), ні врахований неоднократно (різні верхньо-сусідні гарантують, що шляхи відрізняються щонайменше тим попереднім рядком).

Остато́чна відповідь (кількість *усіх* шляхів) визначається як сума чисел останнього рядка. Адже шлях може закінчуватись або у 1-й клітинці останнього рядка, або у 2-й, ..., або у M-й, так що тут теж діє правило суми.

Приклади (чотири окремі):

$N{=}2$, $M{=}3$

1	1	1
2	3	2

Відповідь: $2{+}3{+}2{=}7$.

$N{=}4$, $M{=}2$

1	1
2	2
4	4
8	8

Відповідь: $8{+}8{=}16$.

$N{=}5$, $M{=}4$

1	1	1	1
2	3	3	2
5	8	8	5
13	21	21	13
34	55	55	34

Відповідь: $34{+}55{+}55{+}34{=}178$.

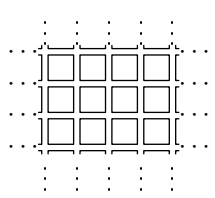
$N{=}4$, $M{=}5$

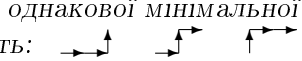
1	1	1	1	1
2	3	3	3	2
5	8	9	8	5
13	22	25	22	13

Відповідь: $13{+}22{+}25{+}22{+}13{=}95$.

4.5.2 Шляхи у квадратних кварталах

(Задача присвячується генеральному плану забудови Черкас 1826 року.)



«Є місто з квадратними кварталами. У такому місті зазвичай є багато різних шляхів однакової мінімальної довжини, які з'єднують одну й ту саму пару перехресть:  *Занумеруємо вертикальні на карті вулиці зліва направо, горизонтальні — знизу догори; щоб задати координати перехрестя, будемо вказувати спочатку номер горизонтальної на карті вулиці, потім вертикальної. Відомо, що деякі перехрестя перекриті (через них проходити не можна). Сформулювати алгоритм, як за початковим і кінцевим перехрестями та переліком перекритих перехресть знайти кількість дозволених найкоротших шляхів. Гарантовано, що початкове і кінцеве перехрестя не перекриті. У випадку, якщо всі найкоротші шляхи перекриті, відповідь=0 (навіть якщо існують довші шляхи, які проходять через не перекриті перехрестя). Вхідні дані: координати початкового та кінцевого перехресть, кількість та координати перекритих перехресть. Результат: кількість шляхів.»*

Ключ до розв'язку дає спостереження, що найкоротшими є ті й лише ті шляхи, в яких *усі* одноквартальні переміщення відбуваються у «правильних» напрямках. Наприклад, якщо кінцева клітинка вища й правіша початкової, кожен «крок» має бути або праворуч, або догори.

(Са́ме для випадку «праворуч-догори» сформульовані подальші міркування, але в остаточному варіанті програми треба застосувати їх до усіх можливих взаємних розташувань старту та фінішу.)

Так що будемо шукати для кожного перехрестя (всере́дині прямокутника, утвореного початковим та кінцевим перехрестями) кількість способів потрапити на нього, рухаючись лише праворуч або догори. Сукупність усіх шляхів, якими можна дійти до перехрестя, рухаючись лише праворуч або догори, природньо розбивається на дві групи: шляхи, що приходять на останньому кроці з сусіднього зліва і шляхи, що приходять на останньому кроці з сусіднього знизу перехрестя.

Отже, застосовне правило суми: кількість способів прийти до будь-якого проміжного перехрестя дорівнює кількості способів прийти до сусіднього зліва плюс кількість способів прийти до сусіднього знизу.

5	1	3	7	16	36	77	77
4	1	2	4	9	20	41	×
3	1	1	2	5	11	21	36
2	1	×	1	3	6	10	15
1	1	×	1	2	3	4	5
0	1	1	1	1	1	1	1
	0	1	2	3	4	5	6

Якщо для деякого перехрестя лівий або нижній сусід виявляється лівіше/нижче початкового, вважаємо відповідний доданок рівним нулю; якщо сусід виявляється перекритим перехрестям, теж вважаємо доданок рівним нулю. Для початкового ж перехрестя кількість способів дорівнює одиниці.

Приклад застосування описаних правил наведено у табличці (хрестиками позначені перекриті перехрестя). Відповіддю є 77 (різних найкоротших шляхів дістатися з $(0;0)$ до $(5;6)$ при трьох перекритих $(1;1)$, $(2;1)$ та $(4;6)$).

(У частковому випадку, коли нема перекритих перехресть, така табличка виявляється повернутим трикутником Паскаля, і відповідь можна виразити простою формулою $C^{\Delta x}_{\Delta x+\Delta y}$ (де Δx і Δy — відстані між початковим і кінцевим перехрестями по горизонталі і по вертикалі). Формулу $C^{\Delta x}_{\Delta x+\Delta y}$ можна вивести й без трикутника Паскаля: загальна кількість переміщень становить $\Delta x+\Delta y$, з них треба вибрати Δx штук горизонтальних. Тільки формулу $C^{\Delta x}_{\Delta x+\Delta y}$ дуже важко, майже неможливо, узагальнити, коли з'являються перекриті перехрестя. А для рекурентних співвідношень це очевидне додавання простого `if`-а.)

4.5.3 Кількість правильних дужкових виразів

«Скільки всього є правильних дужкових виразів, що містять $2n$ символів (тобто n пар дужок)? Дужковий вираз правильний, якщо кількість "`(`" рівна кількості "`)`" і у кожній парі дужок спочатку йде "`(`", а потім "`)`". Вхід: кількість пар дужок. Вихід: кількість правильних виразів. Наприклад, при n=1 такий вираз єдиний: "`()`"; при n=3 таких виразів 5: "`((()))`", "`(()())`", "`(())()`", "`()(())`", "`()()()`".»

Будемо користуватись термінами «позиція у слові» (тобто номер літери; нумерацію починаємо з одиниці) та «$[i\,..\,j]$-підслово» (тобто слово, куди входять усі підряд літери від i-ої до j-ої включно). Термін «правильний дужковий вираз» будемо скорочувати до «п. д. в.».

Очевидно, на 1-ій позиції п. д. в. може бути лише "`(`". Розглянемо, на яких позиціях у слові може опинитися парна до неї закривна дужка. Вона може бути або зразу ж у 2-ій позиції, або десь далі. Якщо вона опиняється десь далі, то між цими дужками щось є. *Причому, раз дужки парні, це «щось» обов'язково мусить бути п. д. в.* Якщо після згаданої "`)`" є деякий «залишок» рядка, він теж мусить бути п. д. в. А якщо покласти, що порожній рядок — теж п. д. в. (причому єдиний п. д. в. довжини́ 0), то будь-який непорожній п. д. в. можна записати у вигляді

$$\underset{\uparrow\\1}{(}\ \underbrace{\cdots\cdots}_{\substack{\text{п. д. в.}\\\text{довжини́}\\2(i-1)}}\ \underset{\uparrow\\2i}{)}\ \underbrace{\cdots\cdots}_{\substack{\text{п. д. в.}\\\text{довжини́}\\2(n-i)}}\quad,\qquad 1\leqslant i\leqslant n. \tag{67}$$

Очевидно, що значення i зручно використати для розбиття всіх п. д. в. на n груп. Для цих груп виконується правило суми, а всере́дині кожної групи виконується правило добутку, бо кожен «лівий» п. д. в. (довжини́ $2\cdot(i-1)$) можна поєднувати з кожним «правим» п. д. в. (довжини́ $2\cdot(n-i)$). Так отримуємо формулу

$$K(n)=\sum_{i=1}^{n}K(i-1)\cdot K(n-i). \tag{68}$$

(З точки зору ефективності, її слід програмувати з використанням циклу та одновимірного масиву, а не рекурсивно.)

Кількість правильних дужкових виразів довжиной $2n$ називають також *n-им числом Каталана*. Наведемо без доведень іншу формулу точного обчислення чисел Каталана

$$K(n) = \frac{C_{2n}^{n}}{n+1} \tag{69}$$

і формулу, яка дозволяє швидко оцінити приблизне значення

$$K(n) \approx 0{,}564 \cdot \frac{4^n}{n\sqrt{n}}. \tag{70}$$

4.5.4 Щасливі квитки

«У місті Глупові прийнята p-кова система числення (замість десяткової), а номери тролейбусних квитків складаються з $2k$ розрядів (кожен розряд — одна p-кова цифра). Квиток вважається щасливим, якщо сума перших k розрядів дорівнює сумі останніх k розрядів. Розробити алгоритм, який за вхідними даними p і k знаходитиме кількість щасливих квитків.»

За умовою, сума лівих k розрядів і сума правих k розрядів мають дорівнювати одна одній, але обидві ці суми разом узяті можуть набувати будь-яке значення (у діапазоні від 0, коли всі цифри нулі, до $k(p-1)$, коли всі цифри максимальні можливі в p-ковій системі числення). Отже, доцільно розбити всі можливі щасливі квитки на групи відповідно значенням цих сум. Очевидно, що для цього розбиття застосовне правило суми.

Розглянемо для прикладу групу з сумами $s=2$ при $k=3$, $p=10$. Повний перелік щасливих квитків можна подати як

$$\left.\begin{array}{l}002\\011\\020\\101\\110\\200\end{array}\right\} \times \left\{\begin{array}{l}002\\011\\020\\101\\110\\200\end{array}\right.$$

Ліворуч і праворуч записані всі трицифрові послідовності, які мають суму 2, поєднання кожної трицифрової послідовності ліворуч з кожною трицифровою послідовністю праворуч дає (єдиний) номер щасливого квитка, суми якого дорівнюють 2, і кожен щасливий квиток, суми якого дорівнюють 2, є поєднанням (однією парою) якихось з цих послідовностей. Тобто, всере́дині кожної такої групи застосовне правило добутку.

Так приходимо до формули

$$N_{total} = \sum_{s=0}^{k(p-1)} (N(s))^2, \tag{71}$$

де $N(s)$ — кількість таких послідовностей (з k цифр p-кової системи числення), що їхня сума дорівнює s.

Практично застосувати (71) до розв'язування задачі можна буде тоді, коли побудуємо алгоритм знаходження $N(s)$.

Можна перебрати всі послідовності з k цифр p-кової системи числення. Кожна з них має якусь суму, от і будемо щоразу збільшувати на один кількість послідовностей, які мають цю суму. Але оцінимо кількість дій такого алгоритму. Він вимагає розглянути p^k чисел — «половинок» номеру квитка. Це суттєво менше, ніж найтупіший лобовий метод (перебір взагалі всіх номерів від 0 до $p^{2k}-1$ і перевірка кожного на щасливість), але теж забагато. А головне, при розв'язанні комбінаторної задачі бажано взагалі уникнути такого етапу, як перебори і додавання одиничок. А поки що такий етап є.

Покажемо, як знайти $N(s)$ (яке краще називати його справжнім позначенням $N(s,p,k)$, бо $N(s)$ залежить від p і k) комбінаторно. Будь-яка послідовність p-кових цифр закінчується якоюсь із цифр 0, 1, …, $p-1$. Цю ознаку можна використати для розбиття на групи, до яких застосовне

правило суми. k'-значна послідовність ($k' \geqslant 2$) може мати суму s, якщо вона закінчується нулем, а решта $k'-1$ розрядів мають суму s, або ж якщо вона закінчується одиницею, а решта $k'-1$ розрядів мають суму цифр $s-1$, і так далі до $p-1$:

$$N(s,\, p,\, k') = \sum_{i=0}^{p-1} N(s-i,\, p,\, k'-1). \tag{72}$$

При k'=1, для всіх s значення $N(s,p,1)$=1; спираючись на них і на формулу (72), можна знайти значення $N(s,p,2)$ для всіх s від 0 до $2(p-1)$, спираючись на них — значення $N(s,p,3)$, і так далі до заданого на вході k. Так що треба просто завести масив і заповнити його.

Само собою, після заповнення останнього рядка масиву N треба використати формулу (71).

(Треба ще врахувати, що при обчисленнях за формулою (72) будуть звернення до вельми дивних елементів: наприклад, при обчисленні $N(17,10,2)$ треба використати значення $N(17,10,1)$ (тобто кількість послідовностей десяткових цифр довжиной 1, які мають суму 17), а при обчисленні $N(7,10,2)$ — значення $N(-2,10,1)$. Аналогічні ситуації можливі і в інших задачах комбінаторики — наприклад, C_3^5, тобто кількість способів вибрати 5 (*різних!*) елементів з 3-х можливих. У таких випадках слід не дивуватися, а просто відзначати, що раз такого не буває, то кількість способів рівна нулю. Можна, наприклад, написати додаткові `if`-и, щоб перевіряти, чи знаходиться індекс у межах розумних значень.)

4.6 Завдання до розділу 4

1. У старшій групі дитячого садочка 15 хлопчиків та 12 дівчаток.
   (а) Скільки є різних способів вибрати одне дитя, яке має прочитати віршика?
   (б) Скільки є різних способів вибрати пару, що має складатися з дівчинки та хлопчика?
2. *Чому* стандартний 8-бітовий байт може набувати са́ме 256 значень?
3. Міський телефонний номер мусить складатися з 6 цифр і не може починатися з 0, з 1 та з 8. Яка максимально можлива кількість номерів?
4. Яка максимально можлива кількість різних автомобільних номерів, якщо брати до уваги лише українські номери, лише сучасний наймасовіший варіант?
5. Любитель мітингувати має k_1 табличок, які слід носити у лівій руці, k_2 табличок, які слід носити у правій руці, та k_3 плакатів, які потрібно носити обома руками (отже, якщо узяти плакат, доведеться не взяти жодної таблички). Зі скількома різними наборами гасел він може вийти на мітинг...
   (а) якщо необхідно, щоб обидві його руки були зайняті?
   (б) якщо такої вимоги не ставиться?
6. Нехай відомо, що пароль складається рівно з 8 символів. Нехай є програма, яка може автоматично перевіряти правильність/неправильність паролів зі швидкістю 10 тис. паролів за секунду. За скільки часу така програма зможе гарантовано підібрати пароль (бо перебере *всі* можливі варіанти), якщо пароль може містити...
   (а) ...лише десяткові цифри;
   (б) лише малі латинські літери;
   (в) будь-яким чином перемішані малі латинські літери та цифри (в т. ч. й лише літери або лише цифри);
   (г) будь-яким чином перемішані малі латинські літери та цифри, причому щонайменше один із символів — цифра, і щонайменше один — літера;
   (д) будь-яким чином перемішані малі та/або великі латинські літери, цифри та символи “`!`”, “`@`”, “`#`”, “`$`”, “`%`”, “`^`”, “`&`”, “`*`”, “`(`”, “`)`”, “`|`”, “`\`”, “`-`”, “`_`”, “`+`”, “`=`”, “`{`”, “`}`”, “`[`”, “`]`”, “`<`”, “`>`”, “`:`”, “`;`”, “`'`”, “`"`”, “`,`”, “`.`”, “`/`”, “`?`”

   Відповіді і виразити формулами, і довести до зручного для сприйняття людиною вигляду (як-то «стільки-то годин, стільки-то хвилин», «стільки-то років», тощо).

7. Пан Мирослав бажає придбати записи композицій “Червона рута”, “Yesterday” та “В последнюю осень”. Скількома різними способами він може це зробити, якщо у музичній крамниці є 5 збірок, до яких входить лише “Червона рута”, 7 збірок — лише “Yesterday”, 4 збірки — лише “В последнюю осень”, 2 збірки містять одночасно “Yesterday” та “В последнюю осень”, 1 збірка — одночасно “Червону руту” та “В последнюю осень”? Враховувати лише ті способи, в результаті яких пан Мирослав купує по одному примірнику кожної з названих композицій. Увесь набір купується одночасно.

8. Скількома способами можна розподілити 6 (різних) листів між трьома кур’єрами? Будь-який лист можна віддати будь-якому кур’єру, розглядаються всі способи, включно з тими, де з усіма листами бігає один кур’єр, а решта б’ють байдики.

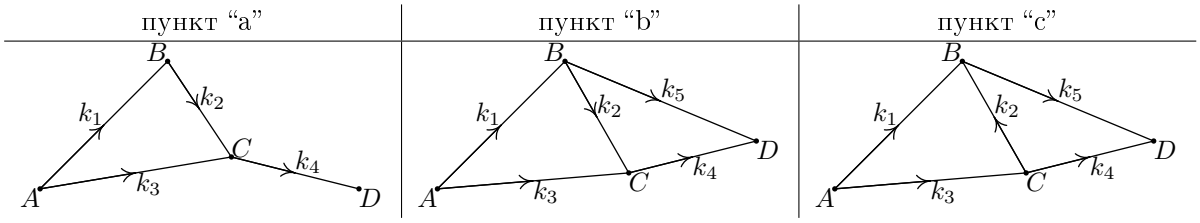
9. Є необмежений запас однакових предметів. Є k різних нерухомих ящиків, у кожен з яких може поміститися не більше, ніж m предметів. Скільки є способів заповнення ящиків?
Підказка. Наприклад, при $n{=}k{=}2$ таких способів є 9: (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), де 1-е число у дужках — кількість предметів у 1-му ящику, 2-е — у 2-му.

10. Нехай число n розкладається на прості множники як $p_1^{m_1} \cdot p_2^{m_2} \cdot \ldots \cdot p_k^{m_k}$. Скільки різних дільників у числá n? Відповіді виразити як формули, залежні від змінних k, m_1, …, m_k, p_1, …, p_k. Для перевірки: при $12 = 2^2 \cdot 3^1$ ($k = 2$, $m_1{=}2$, $m_2{=}1$, $p_1{=}2$, $p_2{=}3$) відповіддю є 6, і цими шістьма дільниками є 1, 2, 3, 4, 6 та 12.
Вказівка. Завдання має зв’язок із попереднім (про не більше, ніж m предметів у кожен з k різних нерухомих ящиків). Щоб зрозуміти, який сáме, варто подати кожен з перелічених дільників 1, 2, 3, 4, 6 та 12 у вигляді розкладення на прості множники $2^a{\cdot}3^b$.

11. Скільки є різних способів поставити дві різні фігури на два різні пóля (клітинки) шахової дошки (стандартної, 8×8)? А якщо треба, щоб ці полéй були різних кольорів (одне чорне, інше біле)?

12. Заводський номер деякого виробу містить спочатку 3 латинські літери, потім 6 цифр. Як літери, так і цифри *можуть* повторюватися. Скільки може існувати різних номерів?

13. На рисунках зображено, через які проміжні пункти можливо дістатися з пункту A до пункту D. Підписи ребер означають «між цими населеними пунктами є стільки-то різних рейсів». Скільки всього є способів дістатися з пункту A до пункту D? Рахувати треба всі способи, не намагаючись вибирати найкоротші чи ще якісь; переходи дозволені лише у відповідності з наведеними напрямками ребер; відповіді виразити як формули від k_1, k_2, …



14. (а) Скільки всього існує різних булевих функцій від n змінних?
(б) Скільки серед них зберігають константу 0 (у смислі розд. 1.7.2)?
(в) Скільки константу 1?
(г) Скільки зберігають і константу 0, і константу 1?
(д) Скільки є самодвоїстими?

15. Ідентифікатор може містити лише латинські літери (великі та маленькі вважаються різними), знак підкреслення (“_”) та цифри, причому цифра не може бути першим символом ідентифікатора.
(а) Скільки існує різних ідентифікаторів довжиною рівно n символів?
(б) Скільки з них є паліндромами? (Тобто, однаково читаються зліва направо і справа наліво, наприклад `azza`, `rotor`.)

16. Скількома способами можна розсадити N людей уздовж довгої лавки?

17. Скількома способами можна розсадити N людей ($N \geqslant 2$) за круглим столом, що повільно обертається? (Способи, що відрізняються лише поворотом стола, вважаються однаковими.)

18. Скільки діагоналей містить опуклий 17-кутник?

19. Заводський номер деякого виробу містить спочатку 3 латинські літери, потім 6 цифр, причому літери мусять бути всі різні, а цифри можуть повторюватися. Скільки може існувати різних номерів?

20. У вищій лізі грають 17 команд. Скільки може бути різних результатів чемпіонату, якщо важливо, які команди вибороли золоту, срібну та бронзову медалі, та які 4 найслабші команди вибули з вищої ліги?

21. У класі навчаються 25 учнів. Скількома різними способами можна сформувати з них екскурсійну групу з 16 учнів...
    (а) ...при відсутності додаткових обмежень
    (б) ...якщо найліпші подруги Марічка, Тетянка та Оксанка або поїдуть всі разом, або не поїде ніхто з них
    (в) ...якщо злійших ворогів Грицька і Федька не можна одночасно включати до групи (але можна обох залишити вдома)
    (г) ...якщо слід врахувати обмеження обох попередніх пунктів

22. У класі навчаються 30 учнів, причому хлопців та дівчат порівну (по 15). Скількома різними способами можна сформувати екскурсійну групу з 16 учнів так, щоб у групі дівчат і хлопців теж було порівну?

23. Нехай у колоді є 52 карти (4 масті, у кожній Т, 2, 3, 4, 5, 6, 7, 8, 9, 10, В, Д, К), гравець отримує 5 (різних) карт. Яка ймовірність, що цей набір виявиться “full house”-ом, тобто в ньому будуть три карти якогось одного номінала і дві карти іншого (наприклад, три сімки і дві дами)? Гравець має право перевпорядкувати свої карти, тобто, наприклад, послідовність «дві сімки, дама, ще одна сімка, ще одна дама» теж являє собою три сімки і дві дами.

24. Скільки є ідентифікаторів довжиńй 7, які ...
    (а) ... складаються лише з малих латинських букв?
    (б) ... складаються лише з малих латинських букв і в яких усі букви різні?
    (в) ... складаються лише з малих латинських букв і в яких *хоча б* дві букви однакові?
    (г) ... складаються лише з малих латинських букв і в яких *точно* дві букви однакові?
    (д) ... складаються лише з малих латинських букв і в яких жодна літера не повторюється на сусідніх позиціях (наприклад, “`azzaone`” та “`abbbcde`” не враховуємо, тоді як “`alalala`” враховуємо)?

25. Якщо буквально врахувати стандарт мови C++ (а також міркування з кінця розд. 4.1 щодо відповідності між математичною моделлю та практикою), розв’язок попереднього завд. 24, ідеальний з точки зору загальнокомбінаторного підходу, потребу́є деяких не те щоб складни́х, але марудних уточнень. Чому потребу́є? Яких уточнень?

26. Є 2 лейтенанти, 5 сержантів та 25 рядових. Скількома способами можна сформувати з них патрульну групу, котра повинна складатися з командира та 2 підлеглих, якщо командир має бути старшим у групі за званням, а підлеглі — мати однакові звання?

27. З 10 викладачів та 12 студентів треба утворити штаб конференції, який має складатися з голови, трьох заступників (з наукових, поліграфічних та організаційних питань), а також 10 технічних секретарів. Скількома способами можна утворити цей штаб (голова та заступники — викладачі, технічні секретарі — студенти)?

28. У спортивній секції навчаються 22 хлопчики. На змагання з бігу слід виставити команду, що складається з 4 дітей. Секція виставляє одну команду, яка буде змагатися з командами, виставленими іншими секціями. Скільки є способів сформувати команду, якщо...
    (а) ... під час змагання вся команда одночасно біжить 2000 м;
    (б) ... змагання являє собою естафету, де спочатку 1-й учасник команди отримує естафетну паличку й біжить з нею 100 м; коли прибіжить, передає естафетну паличку 2-му учаснику команди, який біжить з нею 200 м; коли прибіжить, передає паличку 3-му, який біжить з нею 400 м; коли прибіжить, передає 4-му, який біжить з нею 800 м.
29. (а) Обчислити кількість способів, якими з 7-и кольорів можна утворити 3-кольоровий прапор з горизонтальних смуг однакової ширини.
    (б) Обчислити кількість способів, якими з 7-и кольорів можна утворити прапор з 3-х кольорових горизонтальних смуг однакової ширини.
    (в) Розв'язати варіант задачі з 3-кольоровим прапором, якщо одним із кольорів обов'язково повинен бути зелений (де зелений *є* одним із цих 7-и кольорів).
30. В іспаномовних країнах прийнято давати людям по багато імен. Будемо казати, що "повне" ім'я людини складається з кількох "простих".
    (а) Скільки можна утворити "повних" чоловічих імен довжиною від трьох до п'яти "простих", якщо прості імена не повинні повторюватися, а загальна кількість простих чоловічих імен складає 40?
    (б) Якщо в якості 2-го імені (але не інших) можна використати *також* якесь із 50 жіночих? («також» вводить *додаткові* варіанти, *не* скасовуючи основних.)
31. Є 5 різних книжок з вищої математики, 3 різні книжки з дискретної математики та 4 різні книжки з програмування (вважаємо, що кожна книжка стосується *лише одного* з предметів). Скількома різними способами можна розставити їх уздовж полиці, якщо потрібно щоб книжки з кожної дисципліни стояли підряд?
32. Є 7 книжок, причому 3 із цих 7 однакові (невідрізнювані). Скількома способами можна розставити їх уздовж полиці?
33. Скільки різних ідентифікаторів можна утворити, переставляючи (але не викидаючи і не додаючи) символи в ідентифікаторах:
    (а) `univ`,
    (б) `student`,
    (в) `www_cdu_edu_ua`,
    (г) `o_elzy_911` (починати ідентифікатори з цифри не можна)?
34. Знайти кількість k-значних десяткових чисел, послідовності цифр яких монотонно не зростають (наприклад, `4321`, `4220`, `7777` враховувати треба, а `2008` — ні, бо $0 < 8$).
35. У грі доміно використовуються «дощечки» (вони ж «плитки», вони ж «кості»), що складаються з двох половинок, на кожній з яких крапками зображено ціле число від нуля («пусто») до 6. На обох половинках однієї дощечки може бути двічі зображено одне й те саме число (як-то $\boxed{1}\boxed{1}$; це називається «дупль» або «дубль»). Але нема, наприклад, двох різних дощечок $\boxed{2}\boxed{5}$ та $\boxed{5}\boxed{2}$ — є лише одна дощечка, яку можна перевернути будь-яким з цих способів. Скільки дощечок у наборі?
    Навести два способи розв'язання: в одному спертися на одну з виборок з повтореннями, в іншому — на загальні міркування. Переконатися, що відповіді виявилися однаковими.
36. Скільки є різних способів початкової роздачі при грі в доміно двох гравців? Скільки всього в наборі дощечок — з'ясовано у попередньому завданні; кожен гравець спочатку отримує по 7 дощечок; суттєво, кому які дощечки дісталися, але порядок дощечок у гравця не суттєвий, бо він сам вирішує (у рамках правил), коли якою ходити.
    Навести два способи розв'язання: в одному спертися на одну з виборок з повтореннями, в іншому — на виборки без повторення та загальні міркування. Переконатися, що відповіді виявилися однаковими.

37. У групі з 10 студентів відбувається full random екзамен, на якому кожен може отримати будь-яку з оцінок 2, 3, 4 або 5. Скільки може бути різних результатів екзамену:
    (а) рахуючи з точки зору студентів, котрим суттєво, хто яку оцінку отримав;
    (б) рахуючи з точки зору статистичної звітності, для якої важливі лише кількості двійок, трійок, четвірок та п'ятірок.

    Відповіді подати як у вигляді формул, так і у вигляді чисел.

38. В будинку нелегально проживають k мишей. Одного разу кіт вирішив навести порядок, виловивши їх усіх за n захóдів. Скількома способами він може це зробити:
    (а) рахуючи з точки зору бідолашних мишок?
    (б) рахуючи з точки зору кота, котрому всі миші сірі?

    Між різними захóдами проходить досить тривалий час, а в межах одного захóду миші ловляться одночасно.

39. Знайти кількість розв'язків нерівності $x_1+x_2+x_3+x_4 \leqslant 17$, якщо x_1, x_2, x_3, x_4 — невід'ємні цілі чи́сла. Отримати відповідь, не користуючись технічними засобами, а потім обчислити за допомогою комп'ютера й порівняти результати.

40. У групі 60% студентів знають дискретну математику, 70% студентів — програмування, 50% — обидва предмети. Яка частина студентів групи знають хоча б один з предметів?

41. В університеті випускають три місцевих газети (A, B та C). 60% студентів читають газету A, 50% — газету B, 50% — газету C, 30% — газети A і B, 20% — B і C, 40% — A і C, 10% — читають усі три газети. Скільки відсотків студентів не читають жодної з цих газет?

42. Скільки чисел на проміжку від 1 до 10^9 не кратні ні 2, ні 5? Отримати відповідь, не користуючись технічними засобами, а потім обчислити за допомогою комп'ютера й порівняти результати.

43. Скільки чисел на проміжку від 1 до 10^9 не кратні ні 2, ні 3, ні 5, ні 7? Отримати відповідь, не користуючись технічними засобами, а потім обчислити за допомогою комп'ютера й порівняти результати.

44. Чому дорівнює 1002^5? Отримати відповідь, не користуючись технічними засобами, а потім обчислити за допомогою комп'ютера й порівняти результати.

45. Чому дорівнює $C_n^0 - C_n^1 + C_n^2 - C_n^3 + \ldots + (-1)^n C_n^n$? Отримати відповідь, не користуючись технічними засобами, а потім обчислити за допомогою комп'ютера й порівняти результати.

Додаткові завдання підвищеного рівня складності

1*. До роздоріжжя під'їхав загін із n лицарів, причому l із них ні в якому разі не поїдуть далі праворуч, r — ліворуч, s — прямо, а решті байдуже ($l+r+s<n$). Скількома різними способами лицарі можуть розбитися на загони для подальшої подорожі?

2*. Скільки існує точок перетину діагоналей опуклого n-кутника? (Рахуються лише перетини всере́дині n-кутника, а сходження діагоналей в одну вершину перетином не вважається; вважати, що більш ніж дві діагоналі в одній точці не перетинаються.)

3*. Нехай число n розкладається на прості множники як $p_1^{m_1} \cdot p_2^{m_2} \cdot \ldots \cdot p_k^{m_k}$. Скільки у числа́ n різних дільників, які є точними квадратами? Відповіді виразити як формули, залежні від змінних k, m_1, …, m_k, p_1, …, p_k. Для перевірки: при $12=2^2\cdot 3^1$ ($k=2$, $m_1=2$, $m_2=1$, $p_1=2$, $p_2=3$) відповіддю є 2, і цими двома дільниками й одночасно точними квадратами є $1=1^2$ та $4=2^2$.)

Вказівка. Це завдання є відносно нескладним узагальненням завдання 10 (стор. 106).

4*. За одним із варіантів лотереї «Кєно», грає 80 номерів, гравець вказує довільні 10 номерів, під час розігришу вибираються (випадають) довільні 20 номерів. Яка ймовірність...

(а) ... отримати виграш, встановлений за те, що «вгадав 10 номерів», тобто всі 10 вказаних гравцем номерів виявляються серед 20 вибраних?

(б) ... отримати виграш, встановлений за те, що «не вгадав жодного номеру», тобто жоден з 10 вказаних гравцем номерів не виявився серед 20 вибраних?

(в) ... «вгадати 2 номери», тобто серед 10 вказаних гравцем і 20 вибраних під час розігришу буде рівно 2 спільні? (За це виграшу не передбачено, ймовірність такого результату значно *вища*, ніж «не вгадати жодного номеру».)

Приблизні числові відповіді можна знайти готові в багатьох місцях, зокрема, на сторінці `ru.wikipedia.org/wiki/Кено_(игра)#Вероятность_выигрыша` Але ж треба отримати їх точнішими, і з поясненнями, *звідки* вони беруться...

5*. Задача «Dividers» (`www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=1364`)

- повністю, повний розв'язок передбачає і зарахування програми сайтом, і написання мікрореферату та його захист.
- частково: є програма, яка працює правильно і швидко хоча б для $N \leqslant 10^7$ (мають проходити тести по 12-й включно, а на подальших вердикт або OK, або перевищення часу роботи); мікрореферату не треба, усний захист треба.

Слід захищати або перше, або дру́ге. Обов'язкова вимога — програма повинна чесно знаходити результати, *не* користуючись якими б не було готовими табличками.

6*. Є m хлопців і n дівчат. Скількома способами можна утворити k пар для танцю ($k \leqslant \leqslant \min(m,n)$)? Пари невпорядковані, тобто способи «Іван танцює з Марічкою, а Гнат з Ївгою» та «Гнат танцює з Ївгою, а Іван з Марічкою» вважаються за один.

7*. Для завдання 41 стор. 109 (про читачів місцевих газет), знайти, скільки студентів читають рівно дві газети, та скільки рівно одну.

8*. Реалізувати підпрограми (6 штук), які явно перераховують (виводять на екран або у файл) всі можливі варіанти вивчених стандартних виборок (перестановки, розміщення, сполучення, перестановки з повтореннями, розміщення з повтореннями, сполучення з повтореннями).

Наприклад, підпрограма генерації сполучень при аргументах `(4, 2)` має вивести 6 рядків, наведені праворуч.
(Це завдання не зовсім стосується «комбінаторики у вузькому смислі», що займається методами швидкого підрахунку кількостей, але цілком належить до «комбінаторики у широкому смислі»; див. також стор. 93.)

```
1 2
1 3
1 4
2 3
2 4
3 4
```

9*. На стор. 11 розказано про ситуації, коли `a & b` $= 0$, хоча `a && b` $=$ `true`, і сказано, що їх небагато. Знайти конкретні ймовірності, припускаючи, що `int` (a) 8-бітовий; (b) 16-бітовий; (c) 32-бітовий; (d) 64-бітовий і вважаючи, ніби `a` та `b` можуть рівноймовірно набувати будь-яких значень з діапазону свого типу (що насправді не відповідає практиці використання `int` для подання логічних значень, тому практична значимість цієї задачі невелика; тим не менш, розв'язати слід са́ме таку задачу).

Для 8-бітових та 16-бітових чисел провести також прямий обчислювальний експеримент (знайти й порівняти значення `a & b` та `a && b` для всіх можливих пар `a`, `b`) і переконатися, що відповіді обчислювального експерименту та комбінаторних розрахунків зійшлися.

10*. Задача «Dictionary» (`www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=127`) — повний розв'язок передбачає і зарахування програми сайтом, і написання мікрореферату та його захист.

11*. Написати та захистити мікрореферат на тему «Доведення формули $\overline{C}_n^k = C_{n+k-1}^k$».

12*. Написати та захистити мікрореферат на тему «Доведення формули включень та виключень для довільного n».

13*. Довести тотожність $C_{m+n}^k = \sum_{i=0}^k C_m^i \cdot C_n^{k-i}$, для $0 \leqslant k \leqslant \min(m,n)$ (достатньо будь-яким одним способом, рекомендується через аналіз комбінаторного смислу).

14*. Реалізувати програму, котра працює за тим же принципом, що ручне розв'язання завдання 43 зі стор. 109 (про некратні 2, 3, 5, 7). Можливі міри виконання завдання:
   (а) межі діапазону та значення простих дільників вводяться, але кількість різних простих дільників константно дорівнює 4;
   (б) вводяться і межі діапазону, і довільна кількість довільних простих значень дільників;
   (в) вводяться і межі діапазону, і довільна кількість довільних значень дільників, і ці дільники не зобов'язані бути простими.

15*. Задача «AreaPlus» (`www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=133`) — повний розв'язок передбачає і зарахування програми сайтом, і написання мікрореферату та його захист. (*Вказівка.* Застосувати рекурсивну реалізацію формули включень та виключень.)

16*. Довести співвідношення $\sum_{k=2}^{n-2}(k-1)(n-k-1)=C_{n-1}^3$ (при $n\geqslant 4$) двома способами: методом математичної індукції та шляхом аналізу комбінаторного смислу частин співвідношення. (*Вказівка.* Комбінаторний смисл обох частин пов'язаний з діагоналями многокутника; як са́ме — слід придумати.)

17*. Одного разу двоє випадкових знайомих, котрих звати A_1 та A_2, розговорилися про далеке місто Н-ськ, у котрому проживає N жителів. З'ясувалося: A_1 особисто знає m_1 жителів Н-ська, а A_2 — m_2 жителів.
   (а) Яка ймовірність того, що A_1 та A_2 обидва знають са́ме того жителя Н-ська, що виграв у міській лотереї? Хто саме виграв, не знаємо ні ми, ні A_1 з A_2, але це одна чітко визначена особа.
   (б) Яка ймовірність того, що A_1 та A_2 мають серед жителів Н-ська хоча б одного спільного знайомого?

18*. Довести, що при $n\geqslant 2$, для всіх $0<k<n$ виконується $C_n^k\leqslant\dfrac{n^n}{k^k(n-k)^{n-k}}$. (*Вказівка.* Один з можливих способів доведення включає етап «довести для $k\leqslant n/2$ модифікованим методом математичної індукції».)

19*. Довести, що $C_n^{j+k}\leqslant C_n^j\cdot C_{n-j}^k$ (двома способами: (1) перетворенням формул та матіндукцією; (2) аналізом комбінаторного смислу).

20*. Обчислити $C_n^0-C_n^2+C_n^4-C_n^6+\ldots+(-1)^{(n\operatorname{div}2)\bmod 2}C_n^{2(n\operatorname{div}2)}$.
Вказівка. Придумати нестандартний спосіб поєднати біном Ньютона та більш-менш відому формулу, що вивчається в іншій галузі математики (не дискретної).

21*. Задача «Pavement» (`www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=205`) — повний розв'язок передбачає і зарахування програми сайтом, і написання мікрореферату та його захист.

22*. Задача «Racing» (`www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=183`) — повний розв'язок передбачає і зарахування програми сайтом, і написання мікрореферату та його захист.

5 Теорія графів та алгоритми на графах

5.1 Основні означення теорії графів

5.1.1 Поняття графа. Основні види графів

Іноді графи означають як «сукупність точок площини, деякі пари яких з'єднані лініями». Але нам як програмістам таке означення незручне, бо незручно подавати у комп'ютері площину, точки і лінії[30]. Тим паче, що *головне у графі — сукупність зв'язків, а не геометричне розміщення.*

(Наприклад, задача *«Спортивний турнір проходить за круговою системою (кожен грає з кожним іншим учасником по одній партії, яка закінчується перемогою одного з гравців). Знаючи всі результати, з'ясувати, чи утворилося хоча б одне «зачароване коло» (i_1-й гравець виграв у i_2-го, i_2-й виграв у i_3-го, …, i_{k-1}-й виграв у i_k-го, і при цьому i_k-й виграв у i_1-го)?»* геть не геометрична, але типово «гра́фова».)

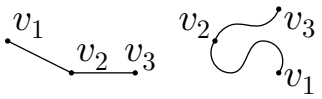
Тож будемо активно користуватися графічними зображеннями (*діаграмами*) графів, але вважати їх лише ілюстраціями; «справжнє» ж означення графа спирається не на геометрію чи топологію, а на теорію множин.

Граф — це пара $\langle V, E\rangle$, де $V \neq \varnothing$ — множина *вершин*, E — множина *ре́бер*, причому кожне ребро́ є парою вершин. Позначення G, V та E походять від англомовних термінів *graph* (граф), *vertex*[31] (вершина) та *edge* (ребро).

Кількість вершин графа ($|V|$) традиційно позначають як n, кількість ребер ($|E|$) — як m. Вершини графа найчастіше позначають v_1, v_2, …, v_n (або $v[1]$, $v[2]$, …, $v[n]$). В разі потреби можуть бути використані й інші позначення (наприклад, номери, але не по порядку, або навіть словесні назви) — аби лише позначення усіх вершин були попарно різними. Ре́бра графа часто не позначають, а задають безпосередньо парами вершин.

Орієнтовані та неорієнтовані Для *неорієнтованих графів* (геометрично — лініям, які з'єднують вершини, не задані напрямки) ре́бра є невпорядкованими парами вершин, тобто $\{v_i, v_j\}$ — те саме, що $\{v_j, v_i\}$.

Наприклад, граф, який можна зобразити будь-яким з наведених рисунків, складається з множи́н вершин $V = \{v_1, v_2, v_3\}$ та множи́н ребер $E = \{\{v_1, v_2\}, \{v_2, v_3\}\}$.



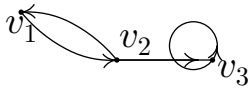
Для *орієнтованих графів* (геометрично — лініям, що з'єднують вершини, приписані напрямки, і наявність зв'язку в один бік не залежить від наявності зв'язку в інший бік) ре́бра є впорядкованими парами. Ре́бра орграфів називають також *ду́гами* (укр., рос. — «*дуга́*», англ. — «*arc*»).

Переклади інших термінів: «неорієнтований граф» — рос. «*неориентированный граф*», англ. «*undirected graph*»; «орієнтований граф» — рос. «*ориентированный граф*», англ. «*directed graph*»; укр. та рос. мовами «орієнтований граф» дуже часто скорочують до «*орграф*», аналогічне скорочення англ. мовою «*digraph*» існує, але не настільки широко вживане.

Ми будемо записувати пари вершин неорієнтованих ребер у фігурних дужках (наприклад, $\{v_i, v_j\}$) і казати, що таке ребро має *два кінці* v_i та v_j, або ж *з'єднує* v_i та v_j; пари вершин дуг — у кутових дужках (наприклад, $\langle v_i, v_j\rangle$)[32] і казати, що v_i — *початок* дуги́, v_j — *кінець* (або що дуга *виходить з* v_i і *заходить у* v_j).

Рос. переклади цих термінів: «*два конца*»; «*соединяет* v_i и v_j»; «*начало*»; «*конец*»; «*выходит из* v_i»; «*заходит в* v_j». Англ. переклади: «*two endpoints*»; «*joins* v_i and v_j», or «*connects* v_i and v_j», or «is *incident on* v_i and v_j»; «*initial vertex*» or «*tail*»; «*terminal vertex*» or «*head*»; «*leaves* v_i» or «is *incident from* v_i»; «*enters* v_j» or «is *incident to* v_j».

Пе́тлі *Петля́* (рос. «*петля́*», англ. «*self-loop*») — це ребро, що заходить у ту ж вершину, з якої вийшло. Часто вважають, що орграфи можуть містити пе́тлі, а неорієнтовані графи — ні; але це обмеження може й порушуватися.



Наприклад, орграф з петлями (див. рис.) складається з множи́н вершин $V = \{v_1, v_2, v_3\}$ та множи́н дуг $E = \{\langle v_1, v_2\rangle, \langle v_2, v_1\rangle, \langle v_2, v_3\rangle, \langle v_3, v_3\rangle\}$.

[30] причому лінії не зобов'язані бути відрізками прямих!

[31] множи́нна форма (у граматичному смислі, рос. «множественное число», англ. «plural») сло́ва vertex утворюється не за правилами англійської мови, а сумішшю правил латинської та англійської мов: *vertices*

[32] У багатьох книжках використовується інша система позначень: і неорієнтовані ре́бра, і ду́ги записують у круглих дужках: (v_i, v_j). Але ж це менш інформативно…

(Таким чином, для орграфів з петлями дуга може бути будь-якою впорядкованою парою вершин, тобто будь-яким елементом декартового квадрата V^2; відповідно, множина дуг E може бути довільною підмножиною V^2. А підмножина V^2 є бінарним («у вузькому смислі») відношенням на множині V. Отже, орграфи з петлями являють собою бінарні відношення, неорієнтовані безпетельні графи можна вважати ірефлексивними симетричними відношеннями, тощо. Ми не будемо сильно заглиблюватися у цей взаємозв'язок (бо історично склалося, що теорію графів розглядають як окремий розділ), але іноді все ж використовуватимемо його.)

Мультиграфи та графи без кратних ребер У *мультиграфах* (рос. «*мультиграф*», англ. «*multigraph*») дозволені різні ре́бра, що з'єднують одну й ту ж пару вершин; формально, E — не класична множина, а мультимножина (пари вершин належать E *стільки разів*, скільки є відповідних ребер).

(Наприклад, мультиграфи потрібні, коли досліджують питання «яку мінімальну кількість доріг потрібно перекрити, щоб стало неможливим доїхати з пункту A до пункту B», а деякі пари пунктів зв'язані більш ніж однією дорогою.

Властивість «мульти-» незалежна від попередніх розглянутих. Тобто, можна розглядати і неорієнтовані мультиграфи, і мультиорграфи...)

Для мультиграфів слід нумерувати ре́бра, бо коли задана лише пара вершин, може бути не ясно, про яке са́ме ребро йде мова.

Зважені та не зважені *Зва́жені* (рос. «*взве́шенные*», англ. «*weighted*») графи передбачають, що кожному ребру приписане дійсне число, яке називають *довжиною* або *вагою* (рос. «*длина́*», «*вес*», англ. «*weight*»).

(Наприклад, для автомобільної дороги суттєвий не лише факт її наявності, а ще й довжина (у кілометрах); при плануванні поїздки міжміським громадським транспортом суттєва не лише наявність рейсу, а й вартість проїзду; тощо.

Властивість зваженості теж незалежна від попередніх розглянутих.

Зважений граф називають також «*мережа*» (рос. «*сеть*», англ. «*network*»); ми не вживатимемо цей термін, бо він має надто багато інших значень.)

За такою класифікацією, в принципі можливі $2\times2\times2\times2=16$ різновидів графів. Деякі з них розглядають значно частіше за інші. Зокрема, часто розглядають *прості* (рос. «*простые*», англ. «*simple*») — неорієнтовані, не зважені, без кратних ребер («не мульти-»), без петель.

Степені вершин *Степінь* (рос. «*степень*», англ. «*degree*») вершини неорієнтованого графа — це кількість ребер, для яких ця вершина є одним з кінців. Степінь вершини v_i позначається $d(v_i)$. Якщо розглядають неорієнтовані графи з петлями, вважають, що петля $\{v_i, v_i\}$ збільшує $d(v_i)$ *на 2*.

Для вершин орграфів рахують окремо *напівстепінь ви́ходу* (кількість дуг, для яких вершина є початком; рос. «*полустепень исхо́да*», англ. «*out-degree*»; позначатимемо $d_{out}(v_i)$) та *напівстепінь захо́ду* (він же *напівстепінь входу* — кількість дуг, для яких вершина є кінцем; рос. «*полустепень захо́да*», англ. «*in-degree*»; $d_{in}(v_i)$). Петля додає по 1 в обидва напівстепені.

(Поширена також красива й коротка нотація, де напівстепені входу та виходу позначають d^+ та d^- відповідно. Але, на жаль, у деяких книгах d^- і d^+ мають в точності протилежний смисл: $d^-(v_i)$ — кількість дуг, що заходять у v_i, $d^+(v_i)$ — кількість дуг, що виходять з v_i. Тому́ в цьому посібнику вибрано позначення d_{in} та d_{out}.)

Для будь-якого орграфа сума напівстепенів виходу всіх вершин дорівнює сумі напівстепенів заходу всіх вершин; для будь-якого неорієнтованого графа сума степенів усіх вершин па́рна.

Доведення. Напівстепені ви́ходу та захо́ду «з'являються» лише «завдяки» ду́гам; кожна дуга вносить по одиничці і в напівстепінь ви́ходу (свого початку), і в напівстепінь захо́ду (свого кінця). Для неорієнтованих графів, кожне ребро збільшує суму степенів вершин на 2 (по одиничці для кожного зі своїх кінців, якщо не петля, або згідно примітки в означенні, якщо петля). ■

(Частину цього твердження, що стосується неорієнтованих графів без петель, називають *лемою про рукостискання* (рос. «*лемма о рукопожатиях*», англ. «*handshaking lemma*»). Ця назва пов'язана з тим, що її можна сформулювати як *«нехай зустрілася група людей; частина з них привіталися, потиснувши один о́дному руки; порахуємо для кожної людини, зі скількома іншими людьми вона привіталась; якщо додати ці кількості (для всіх людей), гарантовано вийде па́рне число»*.)

У будь-якому простому графі (крім одновершинного) є щонайменше дві вершини однакового степеню.

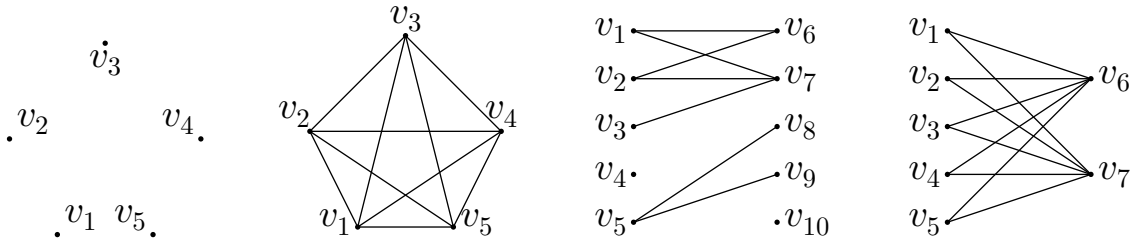
Доведення. Максимально можливе значення степеню дорівнює $n-1$.

А) Нехай у графі є (хоча б одна) ізольована вершина. Тоді не може бути вершини зі степенем $n-1$ (з неї ре́бра мали б іти в усі інші вершини, в т. ч. в ізольовану, а такого не може бути). Значить, степені вершин такого графа можуть набувати лише значення 0, 1, ..., $n-3$, $n-2$ — всього $n-1$ варіант.

Б) Нехай у графі нема ізольованих вершин. Тоді степені можуть набувати значення 1, 2, ..., $n-2$, $n-1$ — теж $n-1$ варіант.

У кожному з випадків n «примірників значень» степеню (для кожної вершини), і вони мусять розподілитися по $n-1$ «типам значень». Значить (за принципом Дирихле, див. також стор. 73), хоча б два «примірники» потраплять до одного й того ж «типу». ■

Спеціальні означення для простих графів Вершину, степінь якої=0, називають *ізольованою* (рос. «*изолированная*», англ. «*isolated*»). Вершину, степінь якої=1 — *висячою*, або *кінцевою* (рос. «*висячая*», «*концева́я*»; англ. «*leaf*», рідше «*pendant*»)[33]. На рис. (c) v_4 та v_{10} ізольовані, v_3, v_8 та v_9 висячі.



(a) (b) (c) (d)

Граф, усі вершини якого мають однакові (рівні) степені, називають *регулярним* (або *однорідним*; рос. «*регулярный*», «*однородный*», англ. «*regular*»). Графи з рис. (a) та (b) регулярні, з рис. (c) та (d) — ні.

Граф, у якому кожна пара вершин з'єднана ребром, називають *повним* (рос. «*полный*», англ. «*complete*») і позначають K_n. Такий граф регулярний, степені його вершин $= n-1$ (бо ре́бра йдуть в *усі інші* вершини). На рис. (b) зображено K_5.

Граф, множина вершин якого V може бути розбита[34] на дві підмножини́ V_1 та V_2 так, що для кожного ребра́ один з кінців належить V_1, а інший — V_2, називають *дводольним* (іноді *двочастковим*; рос. «*двудольный*», англ. «*bipartite*»). Якщо кожна вершина V_1 зв'язана ребром з кожною вершиною V_2, такий граф називають *повним дводольним* і позначають $K_{p,q}$ (де $p = |V_1|$, $q = |V_2|$). На рис. (d) зображено $K_{5,2}$. На рис. (c) зображено неповний дводольний граф. Очевидне з діаграми розбиття його вершин на «ліву» та «праву» долі ($\{v_1, v_2, v_3, v_4, v_5\}$ та $\{v_6, v_7, v_8, v_9, v_{10}\}$) не єдине; приклад іншого розбиття — $\{v_1, v_2, v_3, v_4, v_8, v_9, v_{10}\}$ та $\{v_5, v_6, v_7\}$.

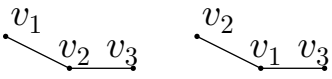
Спеціальні означення для орграфів Вершину, обидва напівстепені якої $= 0$, називають *ізольованою*. Вершину, в яку не заходить жодна дуга ($d_{in}(v_i) = 0$), називають *недосяжною*. Вершину, з якої не виходить жодна дуга ($d_{out}(v_i) = 0$), називають *тупиковою*.

(Поширена також термінологія, де не ізольовану недосяжну вершину називають *джерело* (рос. «*источник*», англ. «*source*»), не ізольовану тупикову — *стік* (рос. «*сток*», англ. «*sink*»). Але під цими ж словами часто мають на увазі інші поняття...)

5.1.2 Ізоморфізм

Ми вже відзначали, що граф визначається сукупністю зв'язків (а не геометричними співвідношеннями), і наводили приклади геометрично різних зображень одного й того ж графа.

А графи з рис. праворуч вважаються різними: перший містить ребро $\{v_2, v_3\}$ й не містить ребра́ $\{v_1, v_3\}$, дру́гий — навпаки.



Але ж природньо вважати їх якщо не «однаковими», то хоча б «схожими» або «рівноцінними». Са́ме такого роду «схожість» і формалізується за допомогою поняття ізоморфізму графів.

[33] слово «pendant» відповідає слову «висяча», тож після «pendant» слід писати «vertex»; слово ж «leaf» само по собі відповідає всьому словосполученню «висяча вершина»

[34] у смислі теоретико-множинного розбиття: $V_1 \cap V_2 = \varnothing$, $V_1 \cup V_2 = V$; див. також стор. 65

Графи $G_1 = \langle V_1, E_1\rangle$ та $G_2 = \langle V_2, E_2\rangle$ *ізоморфні* (рос. «*изоморфны*», англ. «*isomorphic*»), коли існує бієктивна функція $\varphi : V_1 \to V_2$ така, що $\{\varphi(v_i), \varphi(v_j)\} \in E_2 \Leftrightarrow \{v_i, v_j\} \in E_1$.

(Зокрема, для щойно згаданих графів можна розглянути бієкцію $\varphi(v_1) = v_2$, $\varphi(v_2) = v_1$, $\varphi(v_3) = v_3$; тоді ребро $\{v_1, v_2\}$ 1-го графа відповідає ребру $\{v_2, v_1\} = \{v_1, v_2\}$ дру́гого, ребро $\{v_2, v_3\}$ — ребру $\{v_1, v_3\}$.)

Простіше кажучи, два графи *ізоморфні*, коли можна провести таке перенумерування вершин одного з них, що внаслідок нього стануть однаковими і множи́ни вершин, і множи́ни ребер.

Звичайно, ізоморфними можуть бути лише графи однакового виду. Причому, для орграфів ду́ги мають стати однаковими са́ме як ду́ги (з урахуванням напрямку); для мультиграфів мають стати однаковими кількості ребер; для зважених графів — довжи́ни ребер.

(Поняття ізоморфізму (як невідрізнюваності після деякого перетворення) використовується не тільки у теорії графів, а й у багатьох інших галузях математики (не лише дискретної). А дослівний переклад сло́ва «ізоморфізм» — «однаковоформність».)

Перевірка ізоморфності графів — досить складна́ задача. Причому, в теорії все просто: досить перебрати можливі бієкції вершин і для кожної перевірити, чи однакові вийшли множи́ни ребер... Але кількість усіх можливих бієкцій дорівнює $n!$ (де n — кількість вершин графа), а це забагато навіть для комп'ютера. Програма, що перебирає *всі можливі* бієкції, при $n = 11$ працюватиме на сучасному (тактова частота кілька гігагерців) комп'ютері кілька секунд, при $n = 14$ — кілька годин, при $n = 17$ — кілька років. Причому, прогрес «заліза» майже не вирішує проблему: якщо взяти суперкомп'ютер (швидший у тисячі разів), при $n = 17$ така програма працюватиме багато годин (замість кількох років), але варто узяти $n \geqslant 20$, як рахунок знову піде на роки, сторіччя, тощо.

На щастя, насправді нема потреби перебирати абсолютно всі $n!$ бієкцій. З наведеного далі прикладу видно, що об'єм перебору можна зменшити за рахунок простої ідеї: як тільки вибрана якась частина бієкції (співставлені лише деякі пари вершин), зразу ж перевіряти, чи не утворює такий «початок» суперечності; якщо утворює, то доуточняти відповідність по іншим вершинам нема потреби, бо потрібного результату все одно не буде. Більш детально про цю ідею див. у літературі з програмування та аналізу алгоритмів за ключовими словами «*back-tracking*» та «*відтинання при переборі*».

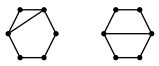
Але відтинання дозволяють лише дещо оптимізувати перебір; способів, що дозволили б повністю позбутися перебору при дослідженні ізоморфізму, наука не знає. До того ж, вплив відтинань на скорочення обсягу перебору дуже сильно залежить від конкретних вхідних даних. Трапляються і пари великих (наприклад, $n \geqslant 100$) графів, ізоморфізм яких можна перевірити досить швидко, і «особливо незручні» пари відносно невеликих (наприклад, $n \approx 40$) графів, дослідження ізоморфізму яких потребу́є нереально великого часу роботи сучасного комп'ютера.

Перш ніж запускати перебір можливих бієкцій, варто перевірити деякі *інваріанти*, тобто властивості, які для ізоморфних графів гарантовано однакові.

Найпростішим інваріантом є кількість вершин та кількість ребер. Трохи «потужніший» інваріант — сукупність (мультимножина) степенів вершин графа (для орграфів — сукупність пар $\langle d_{in}, d_{out}\rangle$).

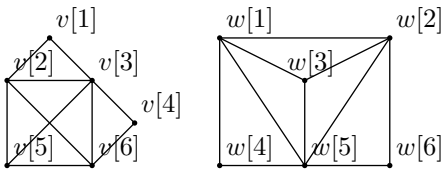
Якщо у графах різна кількість вершин або ребер, графи точно не ізоморфні. Так само, якщо у графів різна сукупність степенів вершин, вони теж не можуть бути ізоморфними. Але ні однаковість кількостей вершин та ребер, ні однаковість мультимножин степенів *не гарантує* ізоморфності.

Наприклад, графи, наведені на рис. праворуч, *не* ізоморфні, хоча вищезгадані інваріанти для них однакові.



Відомі ще кілька (трохи складніших) інваріантів ізоморфізму. Тільки всі вони теж є обов'язковими, але не достатніми умовами. Тому́ й виходить, що аналіз інваріантів допомагає сильно скоротити перебір, але не позбутися його повністю.

Наведемо приклад дослідження ізоморфності двох графів (див. рис.). Для 1-го графа сукупність степенів має вигляд $\{2, 4, 5, 2, 3, 4\}$, для 2-го — $\{4, 4, 3, 2, 5, 2\}$. Вони однакові (з точністю до порядку), отже поки що є шанс, що графи ізоморфні.



Степінь 5 мають лише вершина $v[3]$ (1-го графа) та $w[5]$ (2-го). Значить, якщо бієкція взагалі існує, то має бути $\varphi(v[3]) = w[5]$. Аналогічно, $\varphi(v[5]) = w[3]$ (лише вони мають степінь 3). Зразу ж перевіряємо, що $\{v[3], v[5]\} \in E_1$ і $\{w[3], w[5]\} \in E_2$, тобто цей «початок» бієкції не порушує вимогу про відповідність ребер.

На цьому завершуються «гарантовані» висновки, і починається перебір.

Почнемо з вершин зі степенем 4. Спробуємо співставити $v[2] \leftrightarrow w[1]$. Це припущення правдоподібне, бо $\{v[2], v[3]\} \in E_1 \Leftrightarrow \{w[1], w[5]\} \in E_2$ і $\{v[2], v[5]\} \in E_1 \Leftrightarrow \{w[1], w[3]\} \in E_2$ виконуються. Тоді $v[6] \leftrightarrow$ $\leftrightarrow w[2]$ (бо зі степенем 4 лишилися тільки вони); знов перевіряємо відповідність ребер: $\{v[2], v[6]\} \in E_1 \Leftrightarrow$ $\Leftrightarrow \{w[2], w[1]\} \in E_2$, $\{v[3], v[6]\} \in E_1 \Leftrightarrow \{w[2], w[5]\} \in E_2$ і $\{v[5], v[6]\} \in E_1 \Leftrightarrow \{w[2], w[3]\} \in E_2$ виконуються.

Перейдемо до степеню 2. Спроба співставити $v[4] \leftrightarrow w[4]$ не призводить до успіху, бо при такій бієкції ребру $\{v[4], v[6]\} \in E_1$ мало б відповідати ребро $\{w[4], v[2]\}$ (а такого ребра́ у E_2 нема). Але є ще варіант $v[1] \leftrightarrow w[4]$, $v[4] \leftrightarrow w[6]$. Для нього відповідності ребер $\{v[1], v[2]\} \in E_1 \Leftrightarrow \{w[4], w[1]\} \in E_2$, $\{v[1], v[3]\} \in E_1 \Leftrightarrow \Leftrightarrow \{w[4], w[5]\} \in E_2$, $\{v[4], v[3]\} \in E_1 \Leftrightarrow \{w[6], w[5]\} \in E_2$ і $\{v[4], v[6]\} \in E_1 \Leftrightarrow \{w[6], w[2]\} \in E_2$ виконуються.

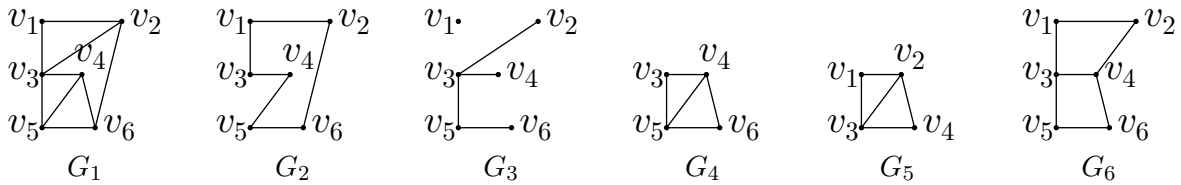
Отже, досліджувані графи ізоморфні; однією з бієкцій вершин, що забезпечують відповідність ребер, є $\begin{array}{c|c|c|c|c|c} v[1] & v[2] & v[3] & v[4] & v[5] & v[6] \\ \hline w[4] & w[1] & w[5] & w[6] & w[3] & w[2] \end{array}$.

Якби остання перевірка відповідності ребер завершилася невдало, це ***не означало б***, що графи не ізоморфні; в такому разі слід було б повернутися на один рівень вгору по дереву перебору, розглянути іншу спробу бієкції для вершин степеню 4, для неї заново розглянути можливі відповідності ще не співставлених вершин...

5.1.3 Підграфи

Граф $G_2 = \langle V_2, E_2 \rangle$ є *підграфом* (рос. «*подграф*», англ. «*subgraph*») графа $G_1 = \langle V_1, E_1 \rangle$, коли $(V_2 \subseteq V_1) \wedge (E_2 \subseteq E_1)$.

(Іншими словами, G_2 є підграфом G_1 тоді й тільки тоді, коли G_2 можна отримати з G_1 виламуванням деяких ребер та/або вилученням деяких вершин (коли вилучається не ізольована вершина, разом з нею вилучаються і пов'язані з нею ре́бра).



Наприклад, серед графів, наведених на рис., G_2, G_3 та G_4 є підграфами графа G_1. Кожен з графів G_5, G_6 містить ребро $\{v[2], v[4]\}$ (якого нема у G_1), тому вони не є підграфами G_1 (не зважаючи на ізоморфність G_5 з G_4, котрий є підграфом G_1).

Увесь граф теж є підграфом само́го себе. Щоб сказати «підграф, не рівний усьому графу», кажуть *власний підграф* (аналогічно власній підмножині).)

Підграф називають *остовним*[35] (або *каркасним*, або *стяжним*, або *кістяковим*; рос. «*о́стовный*», рідше «*скелетный*», «*каркасный*»; англ. «*spanning*»), коли $(V_2 = V_1) \wedge (E_2 \subseteq E_1)$, тобто множина вершин підграфа дорівнює множині вершин початкового графа.

(Серед наведених на рис., остовними підграфами графа G_1 є графи G_2 та G_3 (G_6 не є, бо він взагалі не підграф).)

Підграф *правильний* (рос. «*правильный*», англ. «*full*»), коли $(V_2 \subseteq V_1) \wedge (E_2 = E_1 \cap V_2^2)$, тобто коли підграф містить *усі можливі* ре́бра, обидва кінці яких належать вибраній підмножині вершин. Правильний підграф з множиною вершин V_2 називають також «*підграф, породжений* (множиною) V_2» (рос. «*подграф, порождённый* V_2», англ. «*subgraph, induced by* V_2»).

(На рис. наведений лише один правильний підграф G_1 — граф G_4 (породжений $\{v_3, v_4, v_5, v_6\}$). Терміни «правильний» і «породжений» зазвичай не поєднують, бо саме́ вживання терміну «породжений» означає, що мова йде са́ме про правильний підграф.)

(На жаль, у цьому питанні є різні варіанти термінології. У багатьох книжках (особливо радянських) те, що ми назвали підграфом, називають «частиною графа», тоді як «підграфом» називають лише те, що ми назвали правильним підграфом.)

5.2 Способи подання графів

Ми розглянемо чотири основні способи поданн́я графів. Звичайно, це лише найбільш вживані способи, й ніхто не забороняє в разі потреби придумувати й застосовувати інші структури.

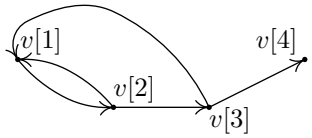
[35] Са́ме ос*т*овний, ні в якому разі не «ос*н*овий». Укр. мовою поширені два варіанти наголосу — «ості́вний» і «о́стовний», рос. — лише «о́стовный». Поширена також думка, за якою українською слід говорити або «карка́сний», або «кістяко́вий», або «стяжни́й», але не «остовний». Але надто вже призвичаїлося використання абревіатур «ОДМВ» та «МОД» (розд. 5.8.2), а деякі словники української мови все ж містять слово «о́стов»...

5.2.1 Матриця інциденцій

Історично перший спосіб поданнá графів — *матриця інциденцій* (рос. «*матрица инциденций*», англ. «*incidence matrix*»). Це матриця розміром $n \times m$ (тобто рядки відповідають вершинам, стовпчики — ребрам (дугам)), елементами якої можуть бути:

1. для неорієнтованих графів:
   (а) 0 — ребро не має стосунку до вершини,
   (б) 1 — вершина є одним із кінців ребрá;
2. для орграфів:
   (а) 0 — дуга не має стосунку до вершини,
   (б) -1 — дуга виходить з вершини,
   (в) $+1$ — дуга заходить у вершину;
3. для (ор)графів з петлями, додатково до вказаних з'являється значення
   - 2 — ребро є петлею при вершині.

Приклад:



	$e[1]=\langle v[1],v[2]\rangle$	$e[2]=\langle v[2],v[1]\rangle$	$e[3]=\langle v[2],v[3]\rangle$	$e[4]=\langle v[3],v[1]\rangle$	$e[5]=\langle v[3],v[4]\rangle$
$v[1]$	-1	1	0	1	0
$v[2]$	1	-1	-1	0	0
$v[3]$	0	0	1	-1	-1
$v[4]$	0	0	0	0	1

У переважній більшості задач матриця інциденцій не зручна і дуже неекономна за обсягом потрібної пам'яті. Але бувають випадки, коли вона все ж доцільна.

(Зокрема, випадок, заради якого і придумували цю матрицю. Придумав її Кірхгоф — той сáмий, котрий сформулював закони електричних кіл (контурів); дрýгий закон Кірхгофа звучить «для кожного замкненого кóла сума падінь напруги дорівнює сумі е. р. с.». Щоб скласти за цим законом систему рівнянь для конкретної електричної схеми, слід вибрати кілька замкнених кіл цієї схеми; причому, вибрати треба такі кола, щоб утворені рівняння не були лінійно залежними. Все це досить зручно робити сáме через матрицю інциденцій. Втім, це був лише ліричний відступ, смисл якого — нагадати, що дискретна математика має зв'язок з багатьма технічними дисциплінами. Для подальшого вивчення теорії графів закони Кірхгофа не потрібні.)

Ще аналоги матриці інциденцій бувають зручні для подання деяких об'єктів, які схожі на графи, але, строго кажучи, не є графами. Зокрема, т. зв. *гіперграфів* (рос. «*гиперграф*», англ. «*hypergraph*»), де кожне ребро являє собою не обов'язково пару вершин, а підмножину з довільною кількістю вершин. Прикладом може бути відношення «тролейбус ... проїжджає через ... » та його матриця (див. розд. 1.6.2, 2.5.1, 2.5.3): маршрути зв'язують місця́, і цим схожі на рéбра; маршрут може проходити через кілька місць (не обов'язково два), тому не є класичним ребром. Матрицю такого відношення (особливо, транспоновану відносно вигляду з розд. 1.6.2) можна трактувати як аналог матриці інциденцій: де маршрут (гіперребро) проходить через визначне місце (вершину), там 1, де не проходить, там 0.

5.2.2 Матриця суміжності

Матриця суміжності (рос. «*матрица смежности*», англ. «*adjacency matrix*») — матриця розміром $n \times n$, тобто і рядки, і стовпчики її відповідають вершинам. На перетині i-го рядка і j-го стовпчика ставиться 1, якщо є ребро (дуга) із вершини $v[i]$ у вершину $v[j]$, або 0, якщо такого ребрá (дугú) немає: $a_{ij} = \begin{cases} 1, & \text{є ребро } v_i \to v_j; \\ 0, & \text{інакше.} \end{cases}$

Розглянемо той самий орграф, для якого будували матрицю інциденцій. Його матриця суміжності має вигляд:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

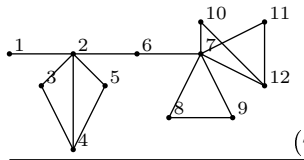
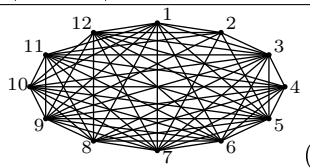
Матрицю суміжності однаково легко застосувати і до неорієнтованих графів, і до орграфів. При цьому, для неорієнтованих графів матриця обов'язково симетрична, а для орграфів, як правило, ні. Щоб подавати (ор)графи з петлями, достатньо дозволити, щоб на головній діагоналі могли бути і нулі (петлí нема), і одиниці (петля є). При поданні мультиграфів, пишуть не одиничку, а кількість ребер (дуг); відсутність ребер (дуг) позначають нулем. При поданні зважених

графів, замість одиниці пишуть довжину відповідного ребра́. З відсутністю ребра́ трохи гірше: *якщо* гарантовано не буває ребер довжиною 0, відсутність можна позначати так само нулем; але поширений також варіант «ставити на місця, де нема ребра́, $+\infty$».

(З тих міркувань, що для типової для зважених графів задачі пошуку найкоротших (найдешевших) шляхів $+\infty$ рівноцінна відсутності ребра́. При комп'ютерному поданні, за $+\infty$ можна бра́ти або дуже велике число (як-то `const double INFTY=1e+300`), або нескінче́нність засобами самих типу з рухомою комою (floating point) та мови програмування: якщо написати `double a = exp(1e6)` і виконати з цією `a` арифметичні дії — у більшості сучасних компіляторів це не лише скомпілюється, а й виведе осмилені результати; але, на жаль, все ще можливі й ситуації, коли це призводить до помилки.)

Єдина комбінація видів графів, для якої практично нереально пристосувати матрицю суміжності — зважені мультиграфи.

Чи ефективно використовує пам'ять матриця суміжності? Це істотно залежить від т. зв. *щільності* графа. Взагалі існує строге означення щільності, яке виражає її парою чисел; але ми його розглядати не будемо, обмежившись приблизними описовими означеннями крайніх випадків:

Розріджений граф	*Щільний* граф
(рос. «*разре́женный*», англ. «*sparse*»)	(рос. «*плотный*», англ. «*dense*»)
кількість ребер співмірна з кількістю вершин ($m \sim n$)	присутня значна частина з усіх можливих ребер ($m \sim n^2$)
 (n=12, m=16)	 (n=12, m=60)
дуже неефективне використання пам'яті — велика кількість комірок без толку зайнята нулями; деякі інші способи уникають цього	непогана ефективність використання пам'яті матрицею суміжності[36] — ре́бра все одно треба десь тримати, й іншими способами краще не буде

(Автору посібника відомо, що на малюнку колонки «щільний граф» дуже тяжко щось розібрати. Але са́ме цей граф взагалі важко намалювати якось зручніше! Тому й вивчаємо способи подання, що зображення графів малюночками має недоліки й обмеження...)

5.2.3 Списки суміжності

У *списках суміжності* (рос. «*списки смежности*», англ. «*adjacency lists*») для кожної вершини вказують перелік вершин, куди йдуть ре́бра (ду́ги) з неї.

Наприклад, списки суміжності (все того ж орграфа) наведені праворуч. Причому, якби список вершини 2 був "`3, 1`", це був би *інший правильний запис того ж* орграфа. Але переставляти цілі рядки́ (вершина і її список), як-то спочатку `3 : 1, 4`, потім `1 : 2`, тощо — погана ідея, бо швидко знаходити потрібний рядок значно легше, коли вони по порядку і без пропусків.

вершина	перелік вершин, куди йдуть ду́ги з неї
1	2
2	1, 3
3	1, 4
4	∅

Для неорієнтованих графів кожне ребро $\{v_i, v_j\}$ (при $i \neq j$) записується двічі: значенням j у списку i-ої вершини та значенням i у списку j-ої.

Списки суміжності легко модифікуються під різні види графів. Зокрема, для мультиграфів вершина може багатократно зустрічатися в одному переліку; для зважених графів елементами списків є не самі́ номери вершин, а па́ри ⟨номер вершини, куди йде ребро; довжина цього ребра́⟩.

(До речі, при такому підході можливе і подання зважених мультиграфів.)

Для використання всіх переваг списків суміжності, пам'ять під переліки вершин бажано виділяти динамічно і незалежно один від о́дного. Якщо зберігати ці переліки у масивах, розміри яких задані «з запасом», не вийде ніякої економії пам'яті порівняно з матрицею суміжності.

Не можна і робити їх надто короткими, бо бувають розріджені графи, де *деякі* вершини є кінцями багатьох ребер. 

[36] Для не мульти- незважених графів (де елементи матриці лише 0 та 1), мовою C++ можна додатково зекономити пам'ять, використавши `vector<vector<bool> >`, який пакує `true` та `false` у окремі біти, тобто до 8 значень у байт. Звісно, присутні також накладні витрати на самі `vector`-и, так що виграш є лише для досить великих n (дуже приблизно — починаючи з багатьох сотень).

Назва «списки суміжності» пов'язана з динамічною структурою даних «зв'я́зний список», бо історично перша програмна реалізація, яка забезпечувала можливість переліків різних довжин — масив вказівників на зв'язні списки. При використанні сучасних версій C++ це не актуально, бо списки суміжності *можна* подавати як `vector<vector<int> >`, тобто майже так само, як і двовимірні масиви, бо `vector<vector<int> >` *дозволяє*, щоб різні рядки мали різну довжину.

Готовий код, який, читаючи перелік ребер, формує списки суміжності:

```
cin >> N >> M;
vector<vector<int> > graph(N);
for(int k=0; k<M; k++) {
  int u, v;
  cin >> u >> v; // u та v -- кінці поточного ребрА
  graph[u].push_back(v); // завжди
  graph[v].push_back(u); // ЛИШЕ для неорієнтованих
}
```

Щоправда, цей код правильний лише у (найпростішому) випадку, коли вершини у вхідних даних занумеровані числами від 0 до $N-1$. Як бути в інших випадках (чи перенумеровувати у діапазон від 0 до $N-1$, чи робити `vector` більшої довжи́ни, чи ще якось) — залежить від ситуації.

Списки суміжності часто кращі за матрицю суміжності ***і*** за економією пам'яті, ***і*** за швидкістю роботи. Чому тоді матрицю все ж використовують? (1) Прямокутні масиви наявні у більшості мов програмування ще з 1960-х років, а *зручно* працювати з масивом переліків різних довжин можна лише використовуючи відносно нові мови / бібліотеки. (2) З точки зору математики, матриця — стандартний засіб, а списки — ні. (3) Списки суміжності значно економніші *лише* для розріджених графів, а для щільних економнішою може виявитися матриця (особливо подана як `vector<vector<bool> >`).

5.2.4 Список ребер

Список ребер (рос. «*список рёбер*», англ. «*edge list*») — це просто перелік усіх елементів множи́н вершин V та всіх елементів множи́н ребер E. Наприклад, ми вже подавали граф як $V = \{v_1, v_2, v_3\}$, $E = \{\{v_1, v_2\}, \{v_2, v_3\}\}$.

Цей спосіб найекономніший за пам'яттю, але часто незручний та неефективний за часом роботи (надто довго шукати потрібне ребро). Втім, іноді виявляється найкращим — наприклад, у алгоритмі Краскала (розд. 5.8.2).

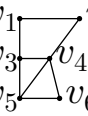
Насамкінець, розглянуті способи — стандартні, але *не* «єдино дозволені». Можливі як різноманітні модифікації цих способів, так і способи, специфічні для якоїсь конкретної задачі.

Наприклад, задачу знаходження найкоротшого шляху в лабіринті, заданому прямокутною табличкою нулів та одиниць ("`1`" — стіна, "`0`" — прохід) варто розв'язувати пошуком у ширину (розд. 5.6.1), але *замість* подання якимсь із способів цього розділу краще використати само́ту табличку: нулі є вершинами графа, пари сусідніх у табличці нулів — ребрами.)

5.3 Маршрути (шляхи) у графі

5.3.1 Основні означення. Досяжність. Довжина маршруту

Маршрут (він же *шлях*; рос. «*маршрут*», «*путь*», англ. «*route*», «*walk*») у графі — це послідовність вершин, де переходи від вершини до наступної відбуваються по ребрам (для орграфів — з урахуванням напрямку).



Наприклад, для графа з рисунка $v_4 - v_6 - v_5$ та $v_1 - v_3 - v_4 - v_2 - v_1 - v_3 - v_5 - v_3 - v_4 - v_2$ є маршрутами (шляхами), а $v_1 - v_2 - v_3 - v_4$ не є, бо нема ребра́, по якому можна було б перейти з v_2 безпосередньо у v_3.

Маршрут (шлях) являє собою са́ме *послідовність* («як їхати», через які проміжні пункти), *а не число* (сумарний кілометраж чи щось подібне). Це дещо протирічить означенню терміна «шлях» з кінематики (розділу фізики). Але *у теорії графів* шлях є послідовністю. Так буває. Само́ слово «граф» для дворянина XIX ст. теж мало зовсім інший смисл, ніж у цьому посібнику...

На жаль, попереднє означення недостатньо строге. Зокрема, у випадку мультиграфів: хоч і вказано, з якої вершини в яку переходити, але не ясно, по якому ребру. Тому більш строгим є таке (складніше) означення:

Маршрут (він же *шлях*) у графі — це послідовність вигляду:

1) вершина $v[i_1]$;
2) ребро $e[j_1]$, що виходить з $v[i_1]$;
3) вершина $v[i_2]$, куди заходить $e[j_1]$;
4) ребро $e[j_2]$, що виходить з $v[i_2]$;

. . . ;

$2k$) ребро $e[j_k]$, що виходить з $v[i_k]$;
$2k{+}1$) вершина $v[i_{k+1}]$, куди заходить $e[j_k]$.

(Наприклад, маршруту $v_4 — v_6 — v_5$ відповідають $k = 2$ та $i_1 = 4$, $i_2 = 6$, $i_3 = 5$.)

Вершина $v[i_1]$ називається *початком* маршруту, вершина $v[i_{k+1}]$ — *кінцем* маршруту. Решта вершин маршруту — *проміжні*, або *внутрішні*.

Випадок $k{=}0$ (маршрут складається лише з вершини $v[k_1]$) дозволений, але маршрут не може взагалі не містити жодної вершини.

Ланцюг (рос. «*цепь*»; англ. «*trail*») — це маршрут, в якому всі ре́бра попарно різні. (Тобто, у ланцюгу ре́бра не можуть повторюватися.)

Простий ланцюг (рос. «*простая цепь*»; англ. «*simple path*») — це ланцюг, в якому всі вершини попарно різні. (Але якщо є *лише* повтор «початкова вершина дорівнює кінце́вій», такий ланцюг теж вважається простим. Звісно, перша й остання вершини не зобов'язані бути однаковими, а лише можуть.)

(Наприклад, маршрут $v[1]-v[2]-v[4]-v[3]-v[1]$ є простим ланцюгом. А маршрути $v[2]-v[4]-v[3]-v[5]-v[4]-v[6]$ та $v[2]-v[4]-v[3]-v[5]-v[4]$ — ланцюги, але не прості.)

Замкнений маршрут (або *циклічний маршрут*; рос. «*замкнутый маршрут*», англ. «*closed walk*») — це маршрут, у якого однакові початок та кінець ($v[i_1]=v[i_{k+1}]$).

Цикл (рос. «*цикл*», англ. «*tour*») — це замкнений маршрут, що одночасно є ланцюгом. (Еквівалентне означення: цикл — це маршрут, у якого кінець дорівнює початку і котрий не містить повторень одного й того ж ребра́.)

Простий цикл (рос. «*простой цикл*», «*simple cycle*») — це маршрут, у якого кінець дорівнює початку і котрий не містить ні ніяких інших повторень вершин, ні повторень ребер. (Еквівалентне означення: простий цикл — це цикл, що одночасно є простим ланцюгом.)

(Дослідимо, чи можна в означенні простого ланцюга замінити слова́ «це ланцюг, у якого . . . » на «це маршрут, у якого . . . »? Здавалося б, можна (й іноді так і роблять), бо вимога неповторюваності вершин ніби передбачає неповторюваність ребер (якщо маршрут містить повторення ребра́, то вершини–кінці цього ребра́ теж повторюються). Але при детальнішому розгляді виявляється, що це справедливо для *майже* всіх маршрутів, *за виключенням* $v[i]-v[j]-v[i]$ (у неорієнтованому графі), який проходить по ребру і тут же повертається. Тут є повторення ребра́, але нема повторень проміжних вершин.

Аналогічно, зауваження щодо першої та останньої вершин у означенні простого ланцюга зроблено заради того, щоб зберегти правильність твердження «цикл є спеціальним випадком ланцюга, а простий цикл — спеціальним випадком простого ланцюга». На жаль, у теорії графів чимало «тонких моментів», коли доводиться або штучно ускладнювати означення, або миритися з тим, що деякі твердження виконуються «майже завжди, крім таких-то виключних випадків».

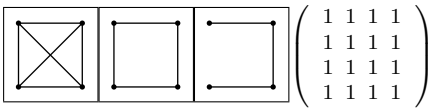
Надалі ми не будемо щоразу загострювати увагу на аналогічних «тонких моментах», бо якщо намагатися розбиратися з усіма дрібницями, можна забути про головне... Тим паче, що відносно деяких ситуацій загальноприйнятої думки просто не існує: автори різних книжок вважають по-різному. Наприклад, англійською мовою широко вживають терміни «path» та «cycle»; але згідно одних джерел вони означають маршрут та замкнений маршрут, згідно інших — простий ланцюг та простий цикл.

Ще один неприємний момент — у багатьох класичних книжках означення, пов'язані з маршрутами, вводять окремо для простих графів, окремо для орграфів, окремо для зважених, тощо. У цьому посібнику зроблена спроба дати означення так, щоб вони стосувалися відразу всіх різновидів графів. У деяких окремих випадках це призвело до використання рідко вживаної термінології (наприклад, лише в малій частині книжок застосовують термін «ланцюг» до орграфів). Але розбиратися з усіма наявними у різних джерелах варіантами термінології було б надто довго й нецікаво.)

Досяжність Вершина $v[j]$ називається *досяжною* (рос. «*достижи́мая*», англ. «*reachable*») з вершини $v[i]$, якщо існує маршрут (хоч який-небудь) із вершини $v[i]$ у вершину $v[j]$.

Матриця досяжності графа — це матриця розміром $n \times n$, де j-й елемент i-го рядка дорівнює або 1 (якщо є маршрут (якої-небудь довжини́) з $v[i]$ у $v[j]$), або 0 (якщо такого маршрута нема): $r_{ij} = \begin{cases} 1, & \text{є маршрут з } v_i \text{ до } v_j; \\ 0, & \text{інакше.} \end{cases}$

(Тобто, матриця суміжності задає, чи є ребро, яке «безпосередньо» з'єднує вершини, а матриця досяжності — чи є між вершинами хоч який-небудь маршрут. Переклади очевидні: рос. «*матрица достижимости*», англ. «*reachability matrix*».)



Матриця досяжності <u>*не є*</u> способом *подання* графа, бо буває (див. рис.), що різні графи мають однакову матрицю досяжності.

(Можна встановити взаємозв'язок між поняттям досяжності та поняттями з розд. 2.6.5. Орграф з петлями є бінарним («у вузькому смислі») відношенням. З точки зору досяжності, будь-який вид графа еквівалентний орграфу з петлями: граф без петель можна вважати графом, що може містити пе́тлі, але са́ме цей примірник їх не містить; неорієнтований граф можна замінити орграфом, де або є обидві дуги́ $\langle v_i, v_j\rangle$ та $\langle v_j, v_i\rangle$, або нема жодної; кратні ре́бра мультиграфів та довжи́ни ребер зважених графів не впливають на досяжність. І якщо поглянути на означення досяжності з точки зору бінарних відношень, видно, що досяжність є рефлексивно-транзитивним замиканням графа.)

(У книжках, призначених для «чистих» математиків, люблять доводити, що матрицю досяжності можна виразити через матрицю суміжності як

$$R = B\big[E + A + A^2 + \ \dots\ + A^{n-1}\big], \tag{73}$$

де E — одинична матриця (по головній діагоналі одиниці, решта елементів нулі), $B[\,\cdot\,]$ — поелементне «булеве перетворення» $B(n) = \begin{cases} 0, & n = 0; \\ 1, & \text{інакше} \end{cases}$, яке нулі лишає нулями, а інші значення перетворює в одиниці. Але програмістам все це мало корисно, бо краще скористатися алгоритмом Воршалла (розд. 5.7.2) або деяким пошуком у графі (розд. 5.6).)

Довжина маршруту *Довжиною* маршруту *для незважених графів* називають кількість ребер цього маршруту. (Тобто, k з дру́гого означення маршруту.) *Довжиною* маршруту *для зважених графів* називають суму довжин його ре́бер $l(e[j_1]) + l(e[j_2]) + \ldots + l(e[j_k])$. В обох випадках, якщо маршрут складається з однієї лише вершини $v[i_1]$, його довжина=0.

(Переклади: рос. «*длина́* маршрута для невзве́шенных/взве́шенных графов», англ. «walk *length* in unweighted/weighted graphs».)

Маршрут може містити повторення вершин і навіть ребер, тому можливі маршрути дуже великої довжини́. Але наступна теорема показує, що маршрути надто великої довжини́ часто можна просто не розглядати.

Для будь-якого маршруту в будь-якому незваженому графі існує маршрут довжини́ $<n$, який має ті са́мі початок та кінець.

(Де n, як завжди, позначає кількість вершин.)

Доведення. За означенням, маршрут довжиною k містить $k+1$ вершин; отже, довільний маршрут довжиною $\geqslant n$ містить $\geqslant n+1$ вершин. За принципом Дирихле, це можливо лише за умови, що у маршруті повторюється одна й та са́ма вершина, тобто $v[i_A] = v[i_B]$ (при $A < B$). «Виріжемо» з маршруту «шматок» $e[i_A]$, $v[i_{A+1}]$, $e[i_{A+1}]$, $\ldots$, $e[i_{B-1}]$, $v[i_B]$ (замінимо замкнений маршрут, що починається й закінчується у вершині $v[i_A]$, на однократне входження $v[i_A]$); отримаємо маршрут, що має ті самі початок і кінець, але меншу кількість вершин та ребер. Якщо знов отримали маршрут довжиною $\geqslant n$, то він знов містить у собі замкнений маршрут, який можна вирізати. Це можна повторювати, доки не прийдемо до маршруту, довжина якого строго менша n. ■

[Наслідок. Якщо з'ясовано, що з вершини $v[i]$ до вершини $v[j]$ нема ні маршрутів довжини́ 0, ні довжини́ 1, $\ldots$, ні довжини́ $n-1$, то маршрутів з $v[i]$ до $v[j]$ взагалі нема. **]**

5.3.2 Ві́дстань. Діаметр, радіус, центр

Ві́дстань (рос. «*расстояние*», англ. «*distance*») у графі від вершини $v[i]$ до вершини $v[j]$ (позначають $d(v_i, v_j)$) — це мінімальна серед довжин усіх можливих маршрутів з $v[i]$ у $v[j]$. (Іншими словами, відстань — це довжина найкоротшого маршрута.) Якщо $v[j]$ не досяжна з $v[i]$, то або кажуть, що $d(v[i], v[j])$ невизначена, або покладають $d(v[i], v[j]) = +\infty$.

Проаналізуємо, чи виконуються для введеного таким чином поняття відстані деякі природні властивості відстані (вони ж «аксіоми метрики»).

Перша «природня властивість» відстані: $(d(v,w)\geqslant 0)\wedge\big((d(v,w)=0)\Leftrightarrow(v=w)\big)$, тобто відстань завжди невід'ємна, причому рівною 0 буває лише відстань до само́ї себе (а для різних вершин строго додатна). Ця властивість виконується для незважених графів та для зважених зі строго додатними довжинами ребер; якщо можливі ре́бра нульово́ї довжини́, від властивості лишається тільки "$(d(v,w)\geqslant 0)$".

Але бувають задачі з довжинами ре́бер довільного знаку. От тоді з'являються проблеми. Найменшою з проблем є те, що перестає виконуватися ця властивість; найбільшою — що може взагалі втратитися смисл поняття «відстань». Якщо існує циклічний маршрут від'ємної довжини́, то можна, багатократно «накручуючи» його, отримати як завгодно малу (від'ємну, велику за модулем) довжину маршруту; і що тоді вважати найкоротшим?..

Дру́га «природня властивість» відстані: $d(v,w)\leqslant d(v,u)+d(u,w)$ (нерівність трикутника). Ця властивість виконується для всіх графів, для яких саме́ поняття відстані має смисл.

Доведення. Коли хоча б один доданок правої частини $=+\infty$, нерівність виконується, бо ліва частина не може бути строго більшою нескінче́нності. Лишається випадок, коли (u досяжна з v) та (w досяжна з u). «Спочатку прийти по найкоротшому маршруту з v до u, потім по найкоротшому маршруту з u до w» є одним із маршрутів з v до w, довжини́ $d(v,u)+d(u,w)$. А $d(v,w)$ — довжина найкоротшого серед *усіх* маршрутів з v до w. Найкоротший серед усіх маршрутів не може бути довшим за один із маршрутів. ■

Мова йшла про нерівність трикутника для відстаней, а не для довжин ре́бер; для довжин ре́бер зваженого графа порушення нерівності трикутника якраз можливі (приклад: вартість безпересадкового проїзду в таксі більша за сумарну вартість проїзду тролейбусами з пересадками). Просто, при виборі найдешевшого маршруту, замість «безпосередніх, але дороги́х» ребер будуть вибиратися інші маршрути, тому, навіть якщо нерівність трикутників порушується для ребер, вона виконується для відстаней.

Третя «природня властивість» відстані: $d(v,w)=d(w,v)$ (симетричність). Очевидно, вона виконується для будь-яких неорієнтованих графів, але, як правило, не виконується для орієнтованих. Са́ме тому́ ми говоримо про «відстань *від ... до ...*» (а не «відстань *між ... та ...*»).

Матриця відстаней (рос. «*матрица расстояний*», англ. «*distance matrix*» — це матриця розміром $n\times n$, де (i,j)-й елемент дорівнює відстані від $v[i]$ до $v[j]$ (якщо $v[j]$ недосяжна з $v[i]$, то $d_{ij}=+\infty$).

У деяких випадках, зокрема для простих графів, одинички матриці відстаней розміщені там са́мо, де й одинички матриці суміжності, і матриця відстаней однозначно задає граф. Але якщо говорити про графи взагалі, включаючи графи з петлями та зважені графи, то матриця відстаней, як і матриця досяжності, не є способом подання графа.

(«Чисті» математики часто розглядають такий спосіб побудови матриці відстаней незваженого графа шляхом аналізу степенів матриці суміжності. На головній діагоналі матриці відстаней D зразу ставимо нулі (0 кроків, щоб дійти з вершини в саму себе); решта клітинок поки що незаповнені. Потім переносимо одинички матриці суміжності A (є ребро) у відповідні клітинки матриці відстаней (можна дійти за 1 крок); але якщо в графі є пе́тлі, у відповідних клітинках (головної діагоналі) матриці D лишаються нулі (бо відстань — довжина *найкоротшого* маршруту, а $0<1$). Потім будуємо матрицю A^2 (яка характеризує маршрути довжиною 2)[37], перебираємо всі ненульові значення A^2, і, якщо відповідна клітинка D усе ще порожня, ставимо туди двійку. І т. д. Якщо після аналізу всіх степенів по A^{n-1} включно якісь із клітинок D все ще не заповнені, пишемо в них ∞. До матриці D будуть занесені відстані (довжи́ни *найкоротших* маршрутів), бо спочатку заносяться всі можливі 0, потім всі можливі 1, потім всі можливі 2, і т. д.

Втім, програмістам краще використати алгоритмічні методи знаходження відстаней (деякі з них розглянуті в розд. 5.6.1 та 5.7).)

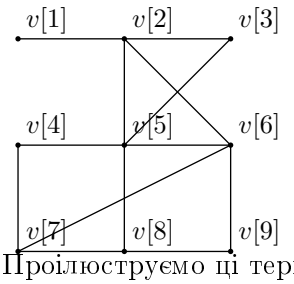
Діаметр, радіус та центр (Ці поняття стосуються *лише* неорієнтованих графів, матриці відстаней яких не містять "∞" ($\Leftrightarrow$ зв'я́зних, див. розд. 5.4.1).)

Ексцентрисите́т вершини — це максимальна з відстаней між цією вершиною та іншими вершинами графа. (Отже, ексцентриситет вершини дорівнює максимальному значенню у відповідному рядку матриці відстаней.)

Діаметр графа $D(G)$ — це максимальний ексцентриситет його вершин.

[37] це майже очевидно, якщо згадати рис. зі стор. 60 та міркування про порівняння композиції відношень та множення матриць зі стор. 60

діаграма



матриця суміжності

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

матриця відстаней

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 2 & 2 & 3 & 3 & 3 \\ 1 & 0 & 1 & 2 & 1 & 1 & 2 & 2 & 2 \\ 2 & 1 & 0 & 2 & 1 & 2 & 3 & 2 & 3 \\ 3 & 2 & 2 & 0 & 1 & 2 & 1 & 2 & 3 \\ 2 & 1 & 1 & 1 & 0 & 1 & 2 & 1 & 2 \\ 2 & 1 & 2 & 2 & 1 & 0 & 1 & 2 & 1 \\ 3 & 2 & 3 & 1 & 2 & 1 & 0 & 1 & 2 \\ 3 & 2 & 2 & 2 & 1 & 2 & 1 & 0 & 1 \\ 3 & 2 & 3 & 3 & 2 & 1 & 2 & 1 & 0 \end{pmatrix}$$

Проілюструємо ці терміни на прикладі. Матрицю відстаней розглядаємо як задану, не спиняючись на процесі її побудови.

У цьому графі ексцентриситет вершин $v[1]$, $v[3]$, $v[4]$, $v[7]$, $v[8]$ та $v[9]$ рівний 3, а ексцентриситет вершин $v[2]$, $v[5]$ та $v[6]$ рівний 2.

Для цього графа діаметр дорівнює 3, причому є досить багато пар вершин, відстані між якими рівні 3, наприклад $v[1]$ та $v[7]$, або $v[3]$ та $v[9]$, тощо.

Радіус цього графа дорівнює 2.

Для цього графа, центральними є три вершини: $v[2]$, $v[5]$ та $v[6]$. Відповідно, центр цього графа — множина $\{v[2], v[5], v[6]\}$.

Отже, діаметр графа не зобов'язаний бути рівно удвічі більшим за його радіус. Але для будь-якого неорієнтованого графа $R(G) \leqslant D(G) \leqslant 2R(G)$.

Оскільки діаметр — максимальний ексцентриситет, а ексцентриситет — максимальна з відстаней між цією вершиною та іншими, то еквівалентним буде інше означення: *діаметр графа* — це максимальна з усіх відстаней між вершинами: $D(G) = \max\limits_{i,j} d(v[i], v[j])$. Тобто, діаметр дорівнює максимальному значенню серед усіх елементів матриці відстаней.

Радіус графа $R(G)$ — це мінімальний ексцентриситет його вершин.

Вершина називається *центральною*, якщо її ексцентриситет дорівнює радіусу графа.

Центр графа — це множина усіх його центральних вершин.

Переклади згаданих термінів: рос. «*эксцентриситéт*», «*диаметр*», «*радиус*», «*центральная*», «*центр*»; англ. «*eccentricity*», «*diameter*», «*radius*», «*central*», «*center*».

Означення ексцентриситету, радіусу, діаметру та центру без проблем узагальнюються на зважені неорієнтовані графи.

Якщо застосувати задачу знаходження центру до зваженого неорієнтованого графа, що задає сукупність доріг (довжина ребра́ — час проїзду), отримуємо один з найпростіших способів наближеного розв'язування т. зв. *мінімаксної задачі розміщення*, смисл якої — «*де розмістити об'єкт, щоб ні для кого він не виявився занадто далеким?*».

(Наприклад, нехай охоронна структура планує утримувати одну базу, з якої охоронці виїздитимуть на об'єкти, де спрацювала сигналізація; місце для цієї бази ще потрібно вибрати — так, щоб прибувати на об'єкти якомога швидше. *Якщо* усі об'єкти, що охороняються, вважаються важливими (*не можна* міркувати «ну то й що як до тих 3% об'єктів довго добиратися, їх же тільки 3%») — тоді найкращі місця для бази знаходяться навколо центральних вершин графа. Звісно, на практиці треба враховувати багато інших обмежень; але аналіз ексцентриситетів все ж є важливою складовою.)

5.4 Зв'язність

5.4.1 Зв'язність та компоненти зв'язності неорієнтованих графів

(У межах цього розділу (5.4.1), «граф» означає «неорієнтований граф».)

Граф *зв'я́зний* (рос. «*свя́зный*», англ. «*connected*»), якщо будь-яка пара його вершин може бути з'єднана яким-небудь маршрутом. (Еквівалентне означення: граф *зв'язний*, коли кожна вершина досяжна з кожної вершини.)

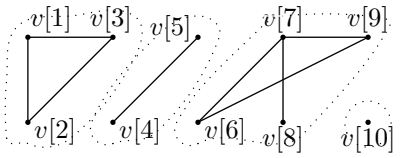
(Зверніть увагу: «зв'я́*зн*ий», а не «зв'яз*д*ний» чи «зв'язни́й»; «зв'я́*зн*ість».)

Граф, що не задовольняє цьому означенню, називають *незв'язним*. Неформально кажучи, незв'язний граф «складається з кількох не пов'язаних шматочків». Ці шматочки називають *компонентами зв'я́зності*, або ж *зв'я́зними компонентами*.

Більш строго, *компонента зв'язності* (рос. «*компонента связности*», англ. «*connected component*») — це такий (правильний) підграф, що: 1) сам підграф зв'язний; 2) до нього не можна додати інші вершини або ре́бра початкового графа так, щоб не порушилася зв'язність.

(Якщо граф в цілому зв'язний, у ньому є одна компонента зв'язності — весь граф.)

І у теорії графів, і у деяких інших галузях математики, як українською, так і російською мовами, є традиція вживати са́ме форму «компоне*нта*» (вона), не зважаючи на те, що в інших галузях «компоне*нт*» (він).



Наприклад, у цьому графі 4 компоненти зв'язності. Це підграфи, породжені (див. стор. 116) такими наборами вершин: 1) $\{v[1], v[2], v[3]\}$; 2) $\{v[4], v[5]\}$; 3) $\{v[6],\ v[7],\ v[8],\ v[9]\}$; 4) $v[10]$.

(А, наприклад, підграф, породжений $\{v[6],\ v[7],\ v[9]\}$ (без $v[8]$) не є компонентою, бо порушує вимогу «не можна додати інші вершини або ре́бра початкового графа так, щоб не порушилася зв'язність» (можна додати вершину $v[8]$ і ребро $\{v[7], v[8]\}$); така са́ма невідповідність означенню виникне і якщо взяти вершини $\{v[1], v[2], v[3]\}$ та ре́бра $\{\{v[1], v[2]\},\ \{v[1], v[3]\}\}$ без ребра́ $\{v[2], v[3]\}$.)

Як визначати компоненти зв'язності довільного графа? З «чисто»-математичної точки зору, можна дослідити, як переставити рядки та стовпчики матриці досяжності, щоб вийшла матриця діагонально-блочної структури. Але з програмістської точки зору зручніше та конструктивніше модифікувати який-небудь пошук у графі (розд. 5.6).

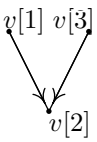
(Якщо згадати, що неорієнтований граф є симетричним відношенням, досяжність — рефлексивно-транзитивним замиканням цього відношення, належність вершин одній компонентні зв'язності є відношенням еквівалентності. Тож компоненти зв'язності «заодно» є ще й класами еквівалентності та утворюють розбиття множи́н вершин графа.)

5.4.2 Зв'язність та компоненти зв'язності орграфів

Розглянемо орграф з двох вершин та єдиної дуги́ $\{\langle v_1, v_2\rangle\}$. Його не варто оголошувати зв'язним, бо з v_2 не можна дістатися до v_1. Але він все ж «більш зв'язний», ніж орграф, який не містить жодної дуги́. Тому́, для орграфів поняття зв'язності виявляється складнішим, ніж для неорієнтованих. А са́ме, для орграфів вводять різні «рівні зв'язності».

1. Орграф *сильно зв'язний* (рос. «*сильно связный*», англ. «*strongly connected*»), якщо для будь-якої пари вершин існують (орієнтовані) маршрути в обох напрямках.
2. Орграф *односторонньо зв'язний* (рос. «*односторонне связный*», рідше «*полусвязный*»; англ. «*semiconnected*»), якщо для будь-якої пари вершин існує (орієнтований) маршрут хоча б в одному напрямку.
3. Орграф *слабко зв'язний* (рос. «*слабо связный*», англ. «*weakly connected*»), якщо при заміні усіх дуг на неорієнтовані ре́бра отримується зв'язний неорієнтований граф.

- Орграф *незв'язний*, якщо він не є навіть слабко зв'язним.



Наприклад, цей орграф не є ні сильно зв'язним, ні односторонньо зв'язним, бо нема маршруту ні з $v[3]$ у $v[1]$, ні з $v[1]$ у $v[3]$. Але він слабко зв'язний, бо при заміні дуг $\langle v[1], v[2]\rangle$ та $\langle v[3], v[2]\rangle$ на ре́бра $\{v[1], v[2]\}$ та $\{v[2], v[3]\}$ отримується зв'язний неорієнтований граф.

А, наприклад, вищезгаданий орграф $V = \{v_1, v_2\}$, $E = \{\langle v_1, v_2\rangle\}$ не є сильно зв'язним, але він односторонньо зв'язний та слабко зв'язний.

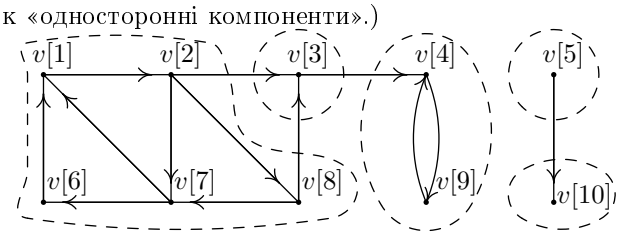
Неважко переконатися, що:

1. якщо орграф сильно зв'язний, то він також і односторонньо зв'язний;
2. якщо орграф односторонньо зв'язний, то він також і слабко зв'язний.

(Причому, односторонньо зв'язний не зобов'язаний бути сильно зв'язним, слабко зв'язний не зобов'язаний бути односторонньо зв'язним.)

Оскільки для орграфів є різні зв'язності, можна говорити і про різні компоненти зв'язності: *сильні компоненти* (компоненти сильної зв'язності) та *слабкі компоненти* (компоненти слабкої зв'язності). Обидва означення аналогічні означенню компоненти неорієнтованої зв'язності: (правильний) підграф, який сам (сильно/слабко) зв'язний, і до нього не можна додати інші вершини або ду́ги початкового графа, не порушивши відповідну зв'язність.

(Чому і сильні, і слабкі є, а односторонніх нема? Розглянемо ще раз недавно згаданий орграф, де дýги з $v[1]$ та $v[3]$ сходяться у $v[2]$. Міркуючи аналогічно «вибрати максимальну групу, в межах якої виконується така-то зв'язність», варто об'єднати і $v[1]$ з $v[2]$, і $v[3]$ з $v[2]$, і при цьому не можна об'єднувати $v[1]$ з $v[3]$. Тобто, якби і спробували ввести «компоненти односторонньої зв'язності», виявилося б, що такі компоненти не утворюють розбиття вершин (деякі вершини належали б відразу багатьом компонентам). Тому було прийнято рішення просто не вводити такого поняття, як «односторонні компоненти».)



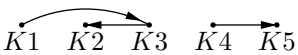
Наприклад, у орграфі з рис. дві слабкі компоненти: 1) породжена $\{v[1],\ v[2],\ v[3],\ v[4],\ v[6],\ v[7],\ v[8],\ v[9]\}$); 2) породжена $\{v[5], v[10]\}$. Водночас, у ньому 5 сильних компонент: К1) породжена $\{v[1], v[2], v[6], v[7], v[8]\}$; К2) породжена $\{v[4], v[9]\}$; К3) вершина $v[3]$; К4) вершина $v[5]$; К5) вершина $v[10]$. Адже лише всере́дині цих підграфів з будь-якої вершини в будь-яку є маршрут.

Сукупність компонент сильної зв'язності завжди задає розбиття множини́ вершин, але деякі дýги можуть не потрапляти до жодної з сильних компонент (скажімо, $\langle v[2], v[3]\rangle$ і ще деякі дýги наведеного прикладу).

Іноді потрібно виділити якраз інформацію про зв'язки́ *між* сильними компонентами. Для цього використовують т. зв. *граф конденсації* (або *конденсацію*; рос. «*граф конденсации*», «*конденсация*», англ. «*condensation*»). Це орграф, вершини якого відповідають сильним компонентам, дýги — дугам між вершинами різних сильних компонент.

Тобто, конденсація є операцією (дією), яка отримує деякий орграф як аргумент, і вертає деякий (можливо, той са́мий) орграф як результат. Тому́ доречно говорити про «конденсацію такого-то орграфа» (або, що те са́мо, «граф конденсації такого-то орграфа»).

(Наприклад, праворуч зображено граф конденсації орграфа, розглянутого три абзаци тому; $K1$, …, $K5$ мають рівно той самий смисл «$K1$ — підграф, породжений $\{v[1], v[2], v[6], v[7], v[8]\}$; $K2$ — породжений $\{v[4], v[9]\}$; $K3$ — $v[3]$; $K4$ — $v[5]$; $K5$ — $v[10]$».



У цьому графі конденсації $K2$ досяжна з $K1$, але дугá $\langle K1, K2\rangle$ нема — адже в початковому графі нема жодної дугú, початок якої належав би $K1$, а кінець $K2$. Граф конденсації розглядають як «не мульти-», тому різні дýги $\langle v[2], v[3]\rangle$ та $\langle v[8], v[3]\rangle$ перетворилися в одну й ту са́му дугу $\langle K1, K3\rangle$.)

У графі конденсації не може бути пар вершин, між якими є ормаршрути в обидва боки (якби така пара вершин була, відповідні сильні компоненти слід було б об'єднати в одну).

Алгоритм виділення компонент слабкої зв'язності очевидно складається з двох великих кроків: 1) перетворити орграф у неорієнтований; 2) виділити компоненти зв'язності отриманого неорієнтованого графа.

Для виділення компонент сильної зв'язності найлегше побудувати матрицю досяжності R, а потім діяти безпосередньо за означенням: $v[i]$ та $v[j]$ належать одній сильній компоненті $\Leftrightarrow$ $\Leftrightarrow (R[i][j] \neq 0) \wedge (R[j][i] \neq 0)$.

(Відомий також алгоритм на основі DFS (розд. 5.6.2), який знаходить сильні компоненти «безпосередньо» (не будуючи матрицю досяжності) і томý ефективніший для розріджених графів; але цей алгоритм складніший, і ми його розглядати не будемо.)

5.4.3 Двозв'язність та k-зв'язність неорієнтованих графів

(У межах цього розділу (5.4.3), «граф» означає «неорієнтований граф».)

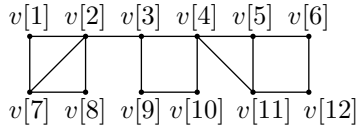
У багатьох ситуаціях, наприклад для мережі автодоріг (або мережі проводового електрозв'язку), зв'язність графа є абсолютно необхідною. Більш того: дуже бажано, щоб перекриття будь-якої однієї дороги або будь-якого одного перехрестя не робило неможливим проїзд між іншими перехрестями, щоб завжди існував інший (можливо, довший) «об'їзний» маршрут.

Точка зчленува́ння (вона ж «*точка з'єднання*»; рос. «*точка сочлене́ния*», англ. «*articulation point*») — це така вершина неорієнтованого графа, що її вилучення (разом з ребрами, для яких вона є одним з кінців) призводить до збільшення кількості компонент зв'язності графа. (Еквівалентне означення: вершина a називається точкою зчленування, якщо існують вершини v і u ($v{\neq}a$, $u{\neq}a$), такі, що *всі* маршрути з u до v проходять через a.)

Міст (рос. «*мост*», англ. «*bridge*») — це таке ребро, що його вилучення (без вилучення вершин) призводить до збільшення кількості компонент зв'язності. (Еквівалентне означення: ребро e — міст, якщо існують вершини v і u, такі, що *всі* маршрути між u та v проходять через e.)

«Збільшення кількості компонент» в обох цих випадках означає, що компонента розпадається на 2 (або 3, 4, ...). Схожим за смислом є коротший вираз «порушення зв'язності», але в цих означеннях точніше вживати «збільшення кількості компонент», бо точки зчленування та мости можуть бути й у графах, які з самого початку не зв'язні.

Наприклад, у зображеному на рис. графі є один міст $\{v[2], v[3]\}$ та три точки зчленування $v[2]$, $v[3]$ та $v[4]$.



Неорієнтований граф *вершинно двозв'я́зний* (рос. «*вершинно двусвя́зный*», англ. «*vertex biconnected*»), якщо він зв'язний, містить щонайменше три вершини і не містить точок зчленування.

Неорієнтований граф *ре́берно двозв'я́зний* (рос. «*рёберно двусвя́зный*», англ. «*edge biconnected*»), якщо він зв'язний, містить щонайменше три вершини і не містить мостів.

Вершинна двозв'язність — сильніша умова, ніж реберна двозв'язність.

(тобто, має місце одностороння імплікація "верш. двозв. → реб. двозв.")

Доведення. Правильність імплікації покажемо через еквівалентну їй імплікацію "¬(реб. двозв.) → → (¬(верш. двозв.))". З яких причин граф може бути не реберно двозв'язним? З означення та закону де Моргана очевидно: (n<3) ∨ (граф не зв'язний) ∨ (існує міст). Якщо справа у 1-ій або 2-ій причині, то з цієї ж причини граф не є вершинно двозв'язним; отже лишається розглянути тільки випадок «є міст». Якщо вилучення само́го лише моста призводить до збільшення кількості компонент зв'язності, то вилучення кінця цього моста призведе до вилучення кількох ребер, в т. ч. і цього са́мого моста; отже — знов-таки до збільшення кількості компонент зв'язності.

Для повної строгості, слід окремо розглянути випадок висячих вершин, бо вилучення висячої вершини разом з ребром *не* призводить до збільшення кількості компонент зв'язності. Але у зв'язному графі при $n \geqslant 3$ не може бути ребер, обидва кінці яких висячі, тож можна просто вилучати не висячий кінець моста (якщо обидва не висячі, будь-який).

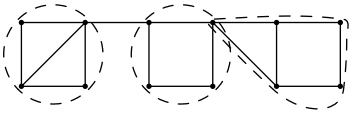
Те, що ця імплікація виконується лише в один бік, демонструється точкою зчленування $v[4]$ останнього наведеного прикладу. ■

Тому́, хоча й є дві різні двозв'язності (вершинна і реберна), часто говорять просто про «двозв'язність», маючи на увазі вершинну двозв'язність.

Компонента двозв'я́зності (вона ж *двозв'я́зна компонента*, вона ж *блок*; рос. «*компонента двусвязности*», «*двусвязная компонента*», «*блок*», англ. «*biconnected component*», «*block*») графа — це такий підграф, що: 1) сам підграф (вершинно) двозв'язний; 2) до нього не можна додати інші вершини або ре́бра початкового графа так, щоб не порушилася двозв'язність.

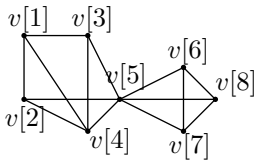
(Якщо граф в цілому двозв'язний, у ньому є один блок — увесь граф.)

Праворуч наведено приклад не двозв'язного графа з виділеними блоками. Точки зчленування можуть (але не зобов'язані) належати відразу кільком блокам. Мости не належать жодному блоку. Таким чином, блоки (компоненти вершинної двозв'язності) не утворюють розбиття ні множини́ ребер, ні множини́ вершин.



Двозв'язність узагальнюють до 3-зв'я́зності, 4-зв'я́зності, тощо — взагалі k-зв'я́зності при довільному k (в межах $1 \leqslant k < n$). Граф *вершинно k-зв'я́зний* (рос. «*вершинно k-связный*», англ. «*k-vertex-connected*»), коли він зв'язний, містить більше k вершин, і одночасне вилучення будь-яких $k-1$ вершин не призводить до незв'язності. Граф *ре́берно k-зв'я́зний* (рос. «*рёберно k-связный*», англ. «*k-edge-connected*»), коли він зв'язний, містить більше k вершин, і одночасне вилучення будь-яких $k-1$ ребер не призводить до незв'язності. Аналогічно двозв'язності, коли говорять просто «k-зв'язний», мають на увазі вершинну k-зв'язність.

(Наприклад, граф з рис. вершинно 1-зв'язний і не є вершинно 2-зв'язним ($v[5]$ — точка зчленування). Водночас, він реберно 2-зв'язний і навіть реберно 3-зв'язний. А реберно 4-зв'язним вже не є, бо внаслідок одночасного вилучення, наприклад, $\{v[5], v[6]\}$, $\{v[5], v[7]\}$ і $\{v[5], v[8]\}$ (у цьому графі є й інші такі трійки) деякі пари вершин (наприклад, $v[2]$ і $v[8]$) перестають бути досяжними одна з одної.)



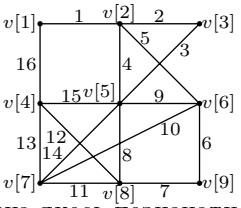
Для перевірки двозв'язності є алгоритм, оснований на пошуку вглиб. Для перевірки k-зв'язності можна застосувати алгоритм пошуку максимального потоку. Але ці алгоритми залишаться за межами цього посібника.

5.5 Ейлерові та гамільтонові шляхи

5.5.1 Ейлерів цикл та ейлерів шлях

Ейлерів шлях (або *ейлерів ланцюг*; рос. «*эйлеров путь*», «*эйлерова цепь*», англ. «*Eulerian path*», «*Eulerian trail*») — це маршрут, що проходить по усім ребрам графа в точності по одному разу. Він може бути *ейлеровим циклом* (якщо, пройшовши по всім ребрам по одному разу, повертається у початкову вершину) або *нециклічним ейлеровим маршрутом* (якщо не повертається).

Побудова одного з ейлерових шляхів — суттєва складова вирішення задачі *«Як намалювати фігуру, не відриваючи олівця від паперу й не малюючи жодну лінію двічі?»*.



Зокрема, у цьому простому графі можна знайти ейлерів цикл. Наприклад, це може бути $v[1] - v[2] - v[3] - v[5] - v[2] - v[6] - v[9] - v[8] - v[5] - v[6] - v[7] - v[8] - v[4] - v[7] - v[5] - v[4] - v[1]$ (є й багато інших).

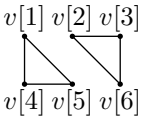
(Як бачимо, ейлерів шлях можна записати послідовністю вершин (принаймні, якщо граф «не мульти-»), але це не наочно: перевіряти вручну, чи справді записаний шлях проходить по всім ребрам по одному разу, незручно (особливо, якщо не можна якось позначати вже пройдені ре́бра). Тому́, якщо граф зображений діаграмою, рекомендуємо на цій самій діаграмі нумерувати ре́бра в порядку їх проходження.)

Коли потрібно не знаходити сам ейлерів шлях (як послідовність вершин та ребер), а лише *з'ясувати, чи існує* у графі ейлерів цикл, слід користуватися дуже простою теоремою Ейлера.

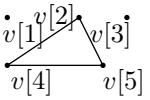
Теорема Ейлера (про ейлерів цикл). *Щоб у графі можна було виділити ейлерів цикл, необхідно і достатньо, щоб граф був зв'язним і степені всіх його вершин були па́рними.*

(Ця теорема стосується лише неорієнтованих графів, зате будь-яких неорієнтованих (в т. ч. мультиграфів та неорієнтованих з петлями).)

(У теоремі Ейлера є неприємний тонкий момент. Перевіряти зв'язність необхідно, бо, скажімо, у графа з рис. (a) степені всіх вершин $= 2$ ($\Rightarrow$ па́рні), а ейлерового циклу нема, бо маршрут не може «перескочити» з однієї компоненти зв'язності в іншу. Але граф з рис. (b) теж незв'язний, а цикл $v[2] - v[5] - v[4] - v[2]$ тим не менш проходить по всім ребрам. Тому таке означення ейлерового циклу та таке формулювання теореми Ейлера не зовсім відповідають одне́ о́дному (хо́ча й записані са́ме так у багатьох книжках*!*).



(a)



(b)

Очевидно, ця неузгодженість проявляється *лише* для: (А) графів, у яких є одна «велика» компонента зв'язності та одна або кілька ізольованих вершин; (Б) графів, які містять *лише* ізольовані вершини, тобто ребер зовсім нема. Можна вирішувати, як краще трактувати ці випадки з точки зору конкретної задачі, де потрібно досліджувати ейлерові цикли, та (в разі потреби) «підправляти» перевірку зв'язності графа.)

Наведемо істотний фрагмент доведення теореми Ейлера.

Якщо ейлерів цикл існує, то граф зв'язний (скрізь у цьому доведенні йдеться про зв'язність з урахуванням зауважень трьома абзацами вище) і для всіх вершин, крім початкової (вона ж кінцева), треба або зайти і вийти, або зайти, вийти, знову зайти, знову вийти, або повторити «зайти і вийти» ще більше разів; якщо це вдалося і всі ре́бра використані по одному разу, то степінь вершини парний, бо кожне «зайти і вийти» потребу́є рівно двох ребер. Для початкової (вона ж кінцева) вершини аналогічно, лише одна з пар «зайти і вийти» перетворюється у «насамперед вийти» та «насамкінець зайти».

Якщо відомо, що степені всіх вершин па́рні, виберемо довільну вершину за початкову й почнемо ходити по графу випадковим чином, звертаючи увагу *лише* на те, щоб не проходити ні по якому ребру повторно. Колись обов'язково виявиться, що прийшли у вершину, в якої нема непройдених ребер. Це *мусить* бути початкова вершина, бо для будь-якої іншої вершини ре́бра витрачалися парами «зайшли, вийшли», й кількість ще не використаних ребер завжди лишалася парною (можливо, сягаючи 0 *після* того, як вийшли). А з початковою вершиною інакше: на самому початку одне з її ребер використане, й у неї цілком можна зайти по останньому ребру, так, що вийти вже не можна.

Можливо, нам пощастило, й так були використані взагалі всі ре́бра графа; тоді все чудово, бо якраз ейлерів цикл і побудований. Але більш імовірно, що десь (*не* у початковій вершині) якісь ре́бра лишилися невикористані. Тоді можна повторювати всі ті самі дії з довільною початковою вершиною і випадковими ходіннями по ще не використаним ребрам, доки не будуть використані взагалі всі ре́бра. (Адже якщо всі степені були парними, то всі кількості ще не використаних ребер лишатимуться парними.)

Твердження, що для будь-якого зв'язного графа такі різні блукання можна з'єднати в один маршрут, потребу́є окремого доведення, і його пропустимо (що, власне, й є причиною, чому це не все доведення, а лише істотний фрагмент).

Ідею про довільні блукання без повторів ребер та з'єднання отриманих циклів у один можна використати і для *побудови циклу як послідовності*.

(У багатьох джерелах стверджують, ніби для цього слід користуватися *алгоритмом Фльо́рі* (рос. *Флёри*, англ. *Fleury*). Автор посібника пропонує охочим ознайомитися з ним за іншими джерелами, задуматися, *як* робити потрібну там перевірку «чи є ребро мостом?», і зрозуміти, що *легше* спертися на блукання та з'єднання циклів у один.)

Умова існування нециклічного ейлерового маршруту теж досить проста: він існує *тоді й тільки тоді, коли граф зв'язний і рівно дві вершини мають непарний степінь* (вони й будуть кінцями маршрута).

Узагальнення теореми Ейлера на орграфи *Цикл, який проходить по всім дугам рівно по одному разу, існує тоді й тільки тоді, коли орграф сильно зв'язний і для кожної його вершини d_{in} (напівстепінь входу) дорівнює d_{out} (напівстепеню виходу). Нециклічний шлях, який проходить по всім дугам рівно по одному разу, існує тоді й тільки тоді, коли орграф односторонньо зв'язний і для всіх вершин, крім двох, $d_{in}(v)=d_{out}(v)$, а для решти двох вершин $d_{in}(start)=$ $=d_{out}(start)-1$, $d_{in}(finish)=d_{out}(finish)+1$.*

5.5.2 Гамільтонів цикл та гамільтонів шлях

Гамільтонів цикл (рос. «*гамильтонов цикл*», англ. «*Hamiltonian cycle*») — це замкнений маршрут, що проходить через усі вершини графа в точності по одному разу. (Нециклічні гамільтонові маршрути теж розглядають, але рідко.)

(Граф з попереднього розд. містить не лише ейлерів цикл, а й гамільтонів. Наприклад, $v[1]$ — $v[2]$ — $v[3]$ — $v[5]$ — $v[6]$ — $v[9]$ — $v[8]$ — $v[7]$ — $v[4]$ — $v[1]$. При графічному зображенні, гамільтонові шляхи зручно виділяти жирним.)

Незважаючи на схожість формулювань, пошук гамільтонових циклів *значно складніший* за пошук ейлерових. Для з'ясування, *чи* містить граф гамільтонів цикл, досі не відомо ніякого простого й однозначного критерію (на зразок теореми Ейлера щодо ейлерового шляху) — взагалі жодного способу, істотно кращого за back-tracking'ові оптимізації перебору (див. книжки з програмування та аналізу алгоритмів, а також розд. 5.1.2).

І це при тому, що перевірка існування гамільтонового циклу — одна з найвідоміших т. зв. NP-повних задач. На межі 1960/1970-х рр. було доведено, що якби вдалося побудувати ефективний алгоритм вирішення хоч якоїсь із NP-повних задач, на основі нього можна було б легко отримати багато нових ефективних алгоритмів для багатьох практично потрібних задач.

Ясно, що на спроби побудувати такий алгоритм були витрачені величезні зусилля — але всі вони скінчилися невдачею. Це схиляє до думки, що ефективного алгоритму, мабуть, просто не існує. Тому ще в тих самих 1960/1970-х рр. розпочали пошук строгого доведення неіснування ефективного алгоритму — й такого доведення теж досі нема!

Так що перевірка існування гамільтонового циклу (та інші NP-повні задачі) — одне з найважливіших відкритих питань сучасної математики.

Звичайно, *деякі* математичні результати щодо умов існування гамільтонових циклів все ж є. Коротко згадаємо кілька найпростіших.

При $n\geqslant 3$, якщо у графі є висяча вершина, цей граф не містить гамільтонового циклу.

Доведення. Якщо вершина v_i висяча, то до неї йде лише одне ребро; позначимо його $\{v_i, v_j\}$. Тоді будь-який замкнений маршрут, що містить v_i, міститиме v_j щонайменше двічі (перед та після v_i). ■

Теорема Дірака стверджує: *якщо у неорієнтованому не-мульти графі без петель, при $n\geqslant 3$, степінь кожної вершини не менший за $n/2$, то граф містить гамільтонів цикл.*

(Без доведення.)

Перевірити, чи є у графі висячі вершини, просто. Якщо є, можна видати відповідь *«Гамільтонового циклу нема»*. Порівняти степені з $n/2$ (і, якщо усі більші, видати відповідь *«Гамільтонів цикл є»*) — теж. Але перша з цих теорем задає *лише необхідну* (не достатню) умову, дру́га — *лише достатню* (не необхідну). І, наприклад, для 7-вершинного графа зі степенями (3, 3, 3, 3, 3, 3, 6) жодна з них не дає остаточної відповіді, все одно доводиться запускати перебір... Більш того: нерідко (в т.ч. для щойно згаданого переліку степенів) існують різні графи з однаковим переліком степенів, одні з яких містять гамільтонів цикл, а інші не містять.

(У «чисто-математичних» книжках можна знайти критерії (тобто умови, необхідні й достатні одночасно) існування гамільтонового циклу. Але всі відомі критерії нічого не дають на практиці, бо їх теж невідомо як перевіряти швидше, ніж повним перебором.)

5.6 Пошуки у графах (обхóди графів)

Пройти по графу, переглянувши його вершини та рéбра — не елементарна задача: шляхи можуть «розповзатися», знову «сходитися», тощо. Тому, щоб уміти правильно переглянути довільний граф, варто знати методи. Їх називають *пошуками* або *обходами*. Історично склалося, що більш поширений варіант «пошук», хоча взагалі-то він менш вдалий: ці методи сáме переглядають (обходять) графи, а не шукають у них щось конкретне.

Пошук у грáфі (рос. «*поиск в грáфе*», англ. «*graph search*»), він же *обхід графа* (рос. «*обход графа*», англ. «*graph traversal*») — це алгоритмічний метод перегляду графа, який гарантує, що всі[38] вершини та рéбра будуть розглянуті і не будуть розглядатися надто багато разів.

Найбільш стандартними є два пошуки: *пошук у ширину* (він же «*пошук уширь*», рос. «*поиск в ширину*», англ. «*breadth-first search*», скорочено «*BFS*») та *пошук у глибину* (він же «*пошук углиб*», рос. «*поиск в глубину*», англ. «*depth-first search*», скорочено «*DFS*»).

5.6.1 Пошук у ширину (BFS) та найкоротші маршрути в незважених графах

Основна ідея пошука в ширину: спочатку досліджуються всі вершини — сусіди початкової (тобто ті, куди йдуть рéбра/дýги з початкової); потім — усі вершини, що знаходяться на відстані 2 від початкової; потім усі, що на відстані 3, і т. д. *Завдяки цьому, кожна вершина* ***зразу*** *досягається за* ***найкоротшим*** *шляхом від початкової вершини.*

(Тут і скрізь у розд. 5.6.1 мається на увазі відстань у незваженому графі, тобто кількість ребер; алгоритми пошуку відстаней у зважених графах див. у розд. 5.7.)

Для реалізації BFS використовують структуру даних *черга* (рос. «*очередь*», англ. «*queue*»). Елементи черги «вишикувані в лінію», яка має два кінці: *голову* («*head*») і *хвіст* («*tail*»). Додавати елементи можна лише у хвіст черги (елемент, що досі був крайнім з хвоста, зміщується «всередúну» черги), брáти (виймати) — лише з голови. Тобто, звичайна черга в яку-небудь касу, в яку ніхто не лізе поза чергою і котру ніхто не кидає не достоявши.

Тоді пошук у ширину можна реалізувати приблизно так:

```
void bfs(start) {
  покласти у чергу єдиний елемент - вершину start;
  st[start] = 1;
  while (у черзі є елементи) {
    узяти елемент з голови черги і розмістити у змінній curr;
    for (next : curr→next) /* перебираємо у змінній next
        усі вершини, куди йдуть рéбра (дýги) з вершини curr */
      if (st[next]==0) {
        додати вершину next в кінець черги;
        st[next] = 1; // позначити як відвідану
      }
  }
}
```

Параметр `start` задає вершину, з якої слід починати пошук у ширину; змінна `curr` зберігає поточну вершину; `next` використовується для перебирання сусідів `curr`. Як правило, вершини нумеруються, і змінні `start`, `curr` та `next` — цілі чúсла (номери вершин); але може бути й інакше.

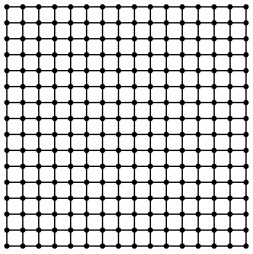
Масив `st` зберігає т. зв. *стáтуси* вершин: елемент `st[next]` містить або 0 (якщо вершина v[`next`] ще не була додана у чергу), або 1 (вже була додана; можливо, вже була й вийнята, але це несуттєво). Це поняття, яке тут назване *статус* (рос. «*статус*», англ. «*status*»), можуть також називати *стан* (рос. «*состояние*», англ. «*state*»), і навіть взагалі ніяк не називати, оперуючи лише поняттями «відвідана», «не відвідана», «чи відвідана?», тощо. Пошук углиб (розд. 5.6.2) та алгоритм Дейкстри (розд. 5.7.1) теж мають справу зі схожими статусами, і це зауваження стосується також і їх.

[38] «Всі» може означати або «абсолютно всі», або «всі досяжні». Іноді вважають, що у пошуку в ширину маються на увазі всі досяжні, у пошуку в глибину — абсолютно всі.

Перед викликом `bfs` *слід ініціалізувати всі елементи* `st` *нулями!*

(Ми не стали робити це в самій підпрограмі `bfs`, бо іноді її використовують як складову частину інших алгоритмів, і деяким з них ініціалізація всере́дині BFS заважала б. Тож будемо вважати, що масив `st` глобальний і ініціалізація його елементів нулями відбувається перед викликом підпрограми `bfs`.)

Надання вершині статусу 1 слід робити при занесенні її у хвіст черги (а не, наприклад, при вийманні з голови). Якщо порушувати це правило, деякі вершини можуть додаватися в чергу по кілька разів.



(А у деяких графах — по *дуже багато* разів. Розглянемо граф, відповідний місту з 15×15 квадратними кварталами, вершинами є перехрестя, ребрами — фрагменти вулиць між перехрестями (див. рис.). Такий граф має $16{\times}16 = 256$ вершин і 480 ребер, що для комп'ютера зовсім не багато.

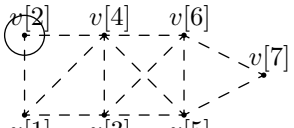
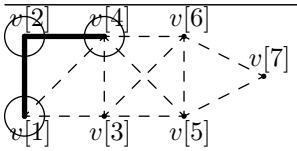
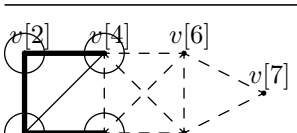
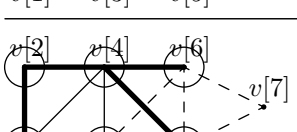
Якщо запустити в такому графі погано реалізований пошук ушир (з невчасною заміною статусу вершин), узявши в якості початкової вершини крайнє кутове перехрестя, діагонально-протилежне кутове перехрестя буде додано в чергу 155 117 520 разів, бо са́ме стільки є різних шляхів однакової мінімальної довжини́ (див. також розд. 4.5.2). Це може потребува́ти *гігабайт (!)* пам'яті для черги. А при правильній реалізації черга не перевищує 16 елементів.)

Цикл перебору ребер, що виходять із поточної вершини `curr`, спеціально записано синтаксично неправильно (`for (next : curr→next) ...`), щоб підкреслити: BFS можна реалізовувати при різних поданнях графа. Якщо використовується матриця суміжності, слід поєднати звичайний цикл від 0 до $N-1$ та `if`; якщо списки суміжності, то `if` не потрібен, бо список суміжності вершини `curr` якраз і містить перелік лише потрібних `next`.

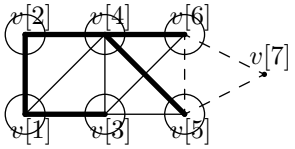
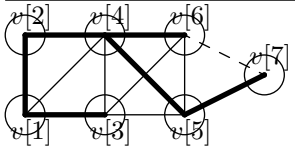
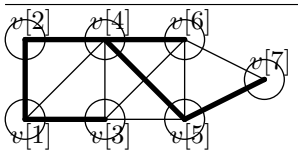
Приклад застосування BFS. Нехай хочемо знайти мінімальні шляхи від $v[2]$; з неї й запускаємо пошук. Нехай цикл `for (next:curr→next) ...` перебирає вершини `next` у порядку зростання номерів.[39]

На серії рисунків, де зображений процес виконання BFS, вершина, обведена кружечком, позначає, що вона вже розглянута (не обведена — не розглянута); ребро, проведене пунктиром — не розглянуте; проведене товстою лінією — розглянуте, причому са́ме по ньому відбувся перехід до розгляду нових вершин; проведене лінією звичайної товщини — розглянуте, але по ньому фактично не переходили (бо вершина `next` вже розглянута).

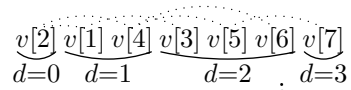
Колонка «черга» показуватиме вміст черги (голова згори, хвіст знизу); елемент, уже вийнятий з голови (він же — поточна вершина, що зберігається в `curr`), та елемент, щойно доданий у хвіст, відділятимемо рисочками.

	черга	`curr`	`next`	коментар
	2	??	??	Перед початком основного циклу, щойно додали у чергу початкову вершину $v[2]$
	2 1 4	2	1 4	Вийняли з черги $v[2]$, розглядаємо ре́бра, що виходять з неї ($v[2] \to v[1]$ та $v[2] \to v[4]$). Ні $v[1]$, ні $v[4]$ ще не розглянуті, тому використовуємо обидва ці ребра й додаємо обидві ці вершини у чергу
	1 4 3	1	2 3 4	Вийняли з черги $v[1]$, розглядаємо спочатку ребро $v[1] \to v[2]$ (яке пропускаємо, бо $v[2]$ вже відвідана), потім $v[1] \to v[3]$ (яке використовуємо й додаємо $v[3]$ у чергу), потім $v[1] \to v[4]$ (яке пропускаємо, бо $v[4]$ вже відвідана)
	4 3 5 6	4	1 2 3 6	Вийняли з черги $v[4]$, розглядаємо $v[4] \to v[1]$, $v[4] \to v[2]$, $v[4] \to v[3]$ (кожне з них пропускаємо, бо вершини $v[1]$, $v[2]$ та $v[3]$ вже відвідані), $v[4] \to v[5]$ та $v[4] \to v[6]$ (кожне з них використовуємо й додаємо $v[5]$ та $v[6]$ у чергу)

[39] Порядок розгляду сусідів може бути довільним, аби жоден з них не був пропущений і не розглядався надто багато разів. Якщо так склалося, що списки суміжності записані не у порядку зростання, нема ніякої потреби сортувати їх перед застосуванням BFS. Див. також примітку 40 на стор. 132

	черга	curr	next	коментар
	3 5 6	3	1 4 5 6	Вийняли з черги $v[3]$, розглядаємо $v[3] \to v[1]$, $v[3] \to v[4]$, $v[3] \to v[5]$ та $v[3] \to v[6]$. Усі вершини $v[1]$, $v[4]$, $v[5]$ та $v[6]$ вже відвідані, пропускаємо
	5 6 7	5	3 4 6 7	Вийняли з черги $v[5]$, розглядаємо $v[5] \to v[3]$, $v[5] \to v[4]$, $v[5] \to v[6]$ (кожне з них пропускаємо, бо $v[3]$, $v[4]$ та $v[6]$ вже відвідані), та $v[5] \to v[7]$ (використовуємо й додаємо $v[7]$ у чергу)
	6 7	6	3 4 5 7	Вийняли з черги $v[6]$, розглядаємо $v[6] \to v[3]$, $v[6] \to v[4]$, $v[6] \to v[5]$ та $v[6] \to v[7]$. Усі вершини $v[3]$, $v[4]$, $v[6]$ та $v[7]$ вже відвідані, пропускаємо
(без змін)	7	7	5 6	Вийняли з черги $v[7]$, розглядаємо $v[7] \to v[5]$ та $v[7] \to v[6]$. Обидві вершини $v[5]$ та $v[6]$ вже відвідані, пропускаємо
(Черга порожня, основний цикл пошуку в ширину закінчується)				

Чому черга забезпечує перегляд спочатку всіх вершин на відстані 1, потім всіх на відстані 2, і т. д.?



Спочатку в чергу кладеться (й тут же виймається) початкова вершина — єдина на відстані 0. У хвіст черги додаються всі її сусіди — тобто, *всі* вершини, що знаходяться на відстані 1. Потім починається обробка вершин відстані 1. Їхні сусіди (точніше, ті з сусідів, які досі не додані в чергу) перебувають на відстані 2, й са́ме вони кладуться у хвіст черги, тож новий вміст черги має вигляд: починаючи з голови йдуть вершини відстані 1, потім, починаючи з якогось місця і до хвоста — вершини відстані 2. Не зважаючи на відсутність явного роздільника, ці групи гарантовано не перемішуються. Так са́мо не перемішуються вони і при більших значеннях відстані.

Модифікація BFS, що дозволяє явно знаходити найкоротші шляхи Наведений на стор. 129 алгоритм — найпростіший варіант BFS. Він-то відбувається по найкоротшим шляхам, але не дає можливості вивести ці шляхи у зручному для користувача вигляді. В цьому смислі він не робить безпосередньо корисної конкретної робо́ти. Так само, як не робить конкретної робо́ти код "`for(int i=0; i<n; i++) a[i];`". Але такий цикл є «кістяком» (основою) більшості засобів обробки масиву — треба лише доповнювати `a[i]` більш осмисленими діями з ним. Аналогічно і вищенаведений код `bfs` є «кістяком», який можна доповнити до корисних алгоритмів.

Введемо масив `dist`, який міститиме відстані вершин від початкової, та масив `previous`, який вказуватиме, з якої вершини ми прийшли у поточну. Заповнення цих масивів відбуватиметься: для початкової вершини — на початку пошуку, для всіх інших досяжних — у момент додавання вершини у хвіст черги. Результат див. нагорі стор. 132; відмінності від попередньої версії виділені позначками "`//!!!`".

Аргументуємо, що масив `dist` справді містить відстані від початкової вершини до всіх досяжних. Для початкової вершини `start` відбувається ініціалізація `dist[start]=0`, і це правильно; для всіх інших досяжних, значення присвоюється як `dist[next] = dist[curr] + 1`, причому за умови, що вершина `next` була додана у чергу як сусід вершини `curr`. А це означає, що до неї можна дійти по шляху довжиною `dist[curr]` + 1: спочатку дійти до `curr`, потім зробити ще один крок. З іншого боку, завдяки дотриманню основної ідеї пошуку в ширину, якщо вершина `next` тільки зараз додається до черги, то коротшого шляху із початкової для неї не існує. Значить, раз до `next` за `dist[curr] + 1` дійти можна, а за меншу кількість кроків ні, то це і є відстань (як кількість ребер) до вершини `next`.

Масив `previous`, який містить попередників вершин на шляхах, використовують, щоб відновлювати самі́ найкоротші шляхи (як послідовності вершин). Це робиться за допомогою т. зв. *зворотнього ходу*. Нехай потрібно знайти найкоротший шлях з початкової вершини `start` до якої-небудь вершини `i`; тоді перш за все встановлюють передостанню вершину шляху (це буде `previous[i]`), потім передпередостанню (`previous[previous[i]]`), і т. д., доки, «задкуючи», не дойдемо до початкової вершини `start`.

```
void bfs_2(start) {
  покласти у чергу єдиний елемент - вершину v;
  st[start] = 1;
  dist[start] = 0;                                              //!!!
  previous[start] = (якесь спеціальне значення "не-вершина");  //!!!
  while (у черзі є елементи) {
    узяти елемент з голови черги і розмістити у змінній curr;
    for (next : curr→next) /* перебираємо у змінній next
        усі вершини, куди йдуть ре́бра (ду́ги) з вершини curr */
      if (st[next]==0) {
        додати вершину next в кінець черги;
        dist[next] = dist[curr]+1;                              //!!!
        previous[next] = curr;                                  //!!!
        st[next] = 1; // позначити як відвідану
      }
  }
}
```

(Звісно, реалізовувати «задкування» у програмі слід не зверненнями у стилі `previous[previous[i]]`, а циклом у стилі `curr=finish; while(curr!=start) { ...; curr = previous[curr];}`.)

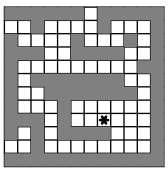
Якщо пошук у ширину може застосовуватися до графів, у яких не всі вершини досяжні з початкової, може бути доцільним ініціалізувати елементи масиву `dist` дуже великими значеннями («машинними ∞»), масиву `previous` — спеціальним значенням «недосяжна» (наприклад, "-2"); хоча досяжність/недосяжність вершини й можна визначати за масивом `st`, буває доцільно продублювати інформацію про те, які вершини вже досягнуті й які ні, у інших масивах. А можна й навпаки: тримаючи інформацію про досягнуті вершини в інших масивах, позбутися масиву `st`. Подібного роду варіацій на тему пошуку вшир насправді досить багато. Але, звісно, в рамках однієї програми слід вибирати щось одне.

Якщо до деяких вершин є різні шляхи однакової мінімальної довжини́, розглянуті пошук у ширину і відновлення шляху зворотнім ходом знаходять якийсь один[40] з них і не зважають на решту. Так, у детально розглянутому прикладі застосування BFS можливий альтернативний найкоротший шлях до $v[7]$, який проходить через $v[6]$ замість $v[5]$, але це ігнорується.

Зазвичай, це нормально, бо достатньо знати якийсь один мінімальний шлях, а всі не потрібні (тим паче, що у деяких графах їх *дуже* багато; див. приклад на стор. 130). А якщо все-таки треба шукати *всі* мінімальні шляхи, це можна зробити, внісши до алгоритму такі, наприклад, зміни.

1. Зробимо `previous` не одновимірним масивом чисел, а `vector`-ом `vector`-ів, щоб `previous[i]` містив *перелік усіх попередників* $v[i]$.
2. Перебираючи сусідів у циклі "`for (next : curr→next)` ...", для тих вершин `next`, які *вже мають статус 1*, будемо перевіряти умову `dist[next] == dist[curr] + 1` — вона означає, що новий шлях до вершини `next` має таку ж довжину, як вже знайдений мінімальний. Якщо так, додаємо `curr` до переліку `previous[next]`.
3. Зворотній хід організуємо рекурсивно: кожен перехід до попередника — рекурсивний виклик; якщо попередників кілька — кілька рекурсивних викликів у циклі.

Пошук у ширину на прямокутній табличці Розглянемо особливий випадок пошуку вшир у застосуванні до такої задачі.



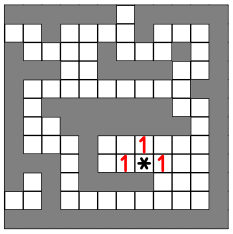
Дано прямокутний лабіринт, сформований з квадратних клітинок: сірі — стіни, білі — вільні (проходи). Потрібно знайти шлях мінімальної довжини́ від початкової позиції, позначеної зірочкою, до будь-якого з виходів (виходами вважаються усі вільні клітинки у зовнішніх стінах). Переходити за 1 хід можна лише у вільну клітинку зі спільною стороною.

Багато дрібних деталей істотно відрізняються від раніше описаного стандартного варіанта BFS, але загальна суть та са́ма. Якщо добре усвідомити спільну суть, зазвичай стає яснішим і сам алгоритм пошуку вшир.

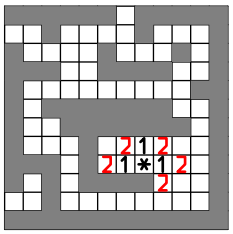
[40] Перший, який трапиться; а який са́ме трапиться першим, залежить від того, в якому порядку розглядаються сусідні вершини у циклі "`for (next : curr→next)` ...". Див. також примітку 39 на стор. 130.

При відображенні процесу, звертатимемо увагу, *що* робиться на відповідному кроці, не заглиблюючись у технічні деталі того, *як* це зробити.

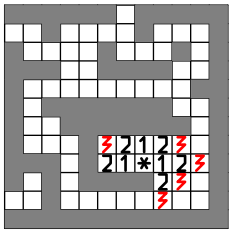
Позначимо всі вільні клітинки навколо старту одиничками



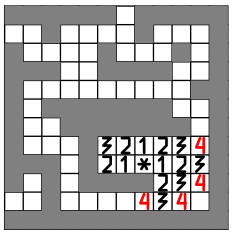
Позначимо всі вільні клітинки навколо всіх одиничок двійками



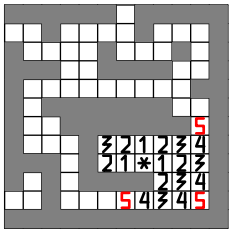
Позначимо всі вільні клітинки навколо всіх двійок трійками



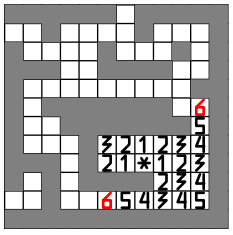
Позначимо всі вільні клітинки навколо всіх трійок четвірками



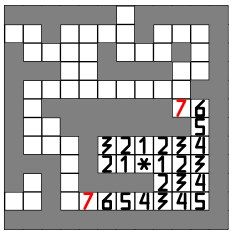
Позначимо всі вільні клітинки навколо всіх четвірок п'ятірками



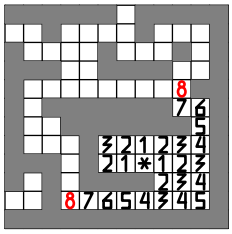
Позначимо всі вільні клітинки навколо всіх п'ятірок шістками



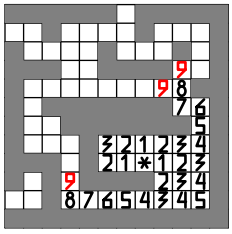
Позначимо всі вільні клітинки навколо всіх шісток сімками



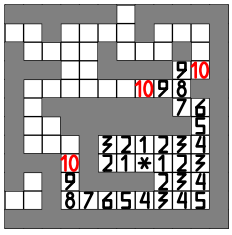
Позначимо всі вільні клітинки навколо всіх сімок вісімками



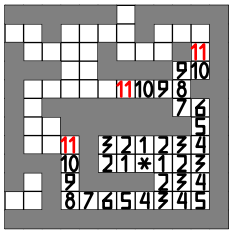
Позначимо всі вільні клітинки навколо всіх вісімок дев'ятками



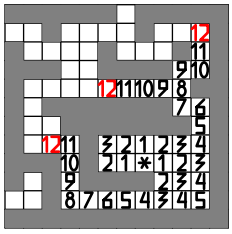
Позначимо всі вільні клітинки навколо всіх дев'яток десятками



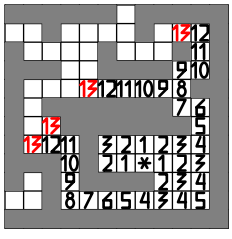
Позначимо всі вільні клітинки навколо всіх десяток одинадцятками



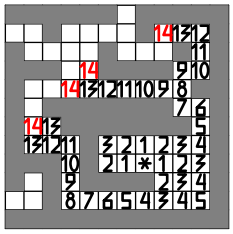
Позначимо всі вільні клітинки навколо всіх одинадцяток дванадцятками



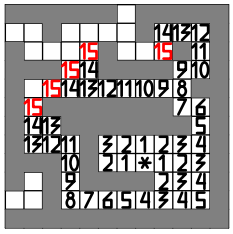
Позначимо всі вільні клітинки навколо всіх дванадцяток тринадцятками



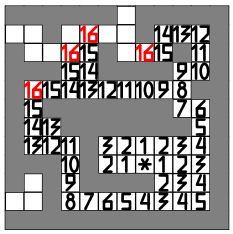
Позначимо всі вільні клітинки навколо всіх тринадцяток чотирнадцятками



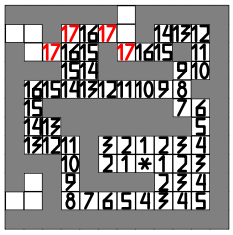
Позначимо всі вільні клітинки навколо всіх чотирнадцяток п'ятнадцятками



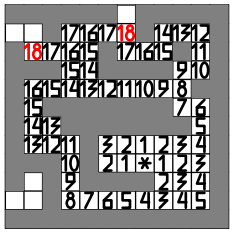
Позначимо всі вільні клітинки навколо всіх п'ятнадцяток шістнадцятками



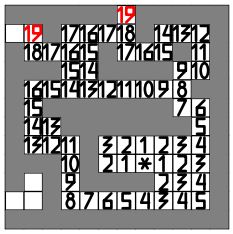
Позначимо всі вільні клітинки навколо всіх шістнадцяток сімнадцятками



Позначимо всі вільні клітинки навколо всіх сімнадцяток вісімнадцятками



Позначимо всі вільні клітинки навколо всіх вісімнадцяток дев'ятнадцятками

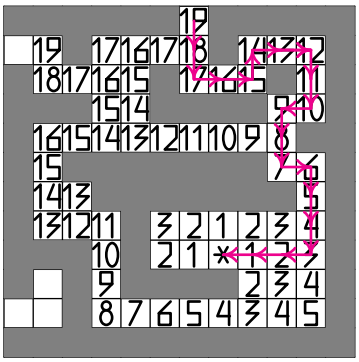


Дістатися до виходу (нагорі прямокутника), можна завершувати основний етап.

Виконувати наведені дії на прямокутній табличці можна різними засобами; пошук у ширину, з його чергою та статусами, цілком годиться, більш-менш вдало поєднуючи простоту́ та ефективність. Без черги (чи іншого подібного контейнеру), для виконання дій, подібних до *«Позначимо всі вільні клітинки навколо всіх шісток сімками»*, треба переглядати усе поле (навіть якщо таких

шісток всього дві). Якщо ж складати клітинки (наприклад, як пари чисел «№ рядка, № стовпчика») у чергу, це будуть клітинки, що виймаються з черги підряд. «Клітинка вільна» відповідає «клітинка не є стіною та `st[цієї клітинки]==0`». Тобто, цю вільність (невідвіданість) клітинки можна позначати і якимсь спеціальним значенням у то́му са́мому масиві, де пишуться чи́сла-відстані. Але можна й статусами. І так далі.

Щоб відновити маршрут як послідовність клітинок, слід застосувати зворотній хід. Тут можна застосувати в т. ч. й вищеописану схему із запам'ятовуванням попередників у окремий масив `previous`. Правда, са́ме в цьому випадку це не дуже зручно: кожна клітинка тоді повинна містити і число-відстань, і пару чисел–координати клітинки-попередника. Тут зручніше відновлювати відновлювати попередників «на ходу».



Клітинка, де вперше досягли виходу, має значення 19; отже, в неї прийшли з сусідньої зі значенням 18. У ту, що має значення 18, прийшли з сусідньої зі значенням 17, і таких сусідніх зі значенням 17 є дві різні; значить, є різні маршрути однакової мінімальної довжини́. Якщо хочемо знайти будь-який один з маршрутів, то вибираємо будь-яку одну з цих двох сусідніх зі значенням 17. Наприклад, нижню. У неї, що має значення 17, прийшли з сусідньої зі значенням 16. І так «задкуємо» до самого старту.

5.6.2 Пошук у глибину (DFS), перевірка ациклічності орграфа та топологічне сортування

Основна ідея пошуку в глибину: коли з'являється розгалуження, треба спочатку повністю дослідити одну його гілку і лише потім перейти до розгляду інших (якщо вони все ще будуть не розглянуті).

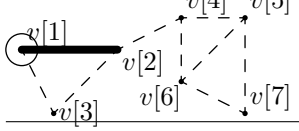
Одна зі стандартних реалізацій DFS — рекурсивна:

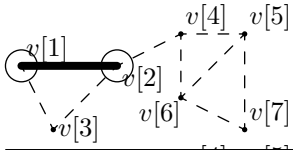
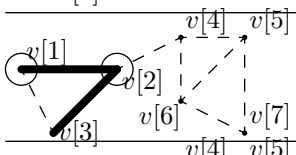
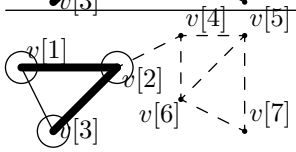
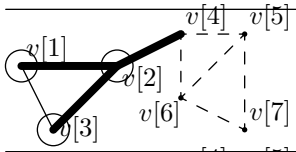
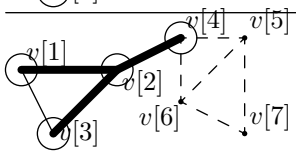
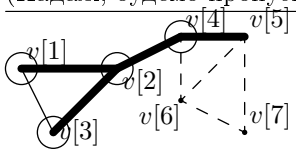
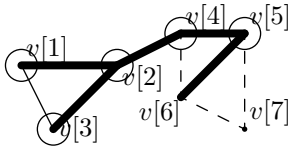
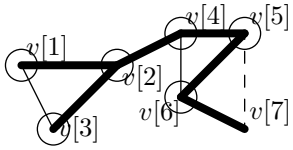
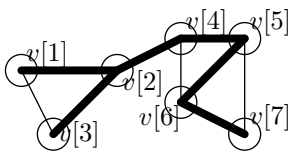
```
void dfs(curr) {
  st[curr] = 1;
  for (next : curr→next) /* перебираємо у змінній next
       усі вершини, куди йдуть ре́бра (ду́ги) з вершини curr.
       Змінна next обов'язково ЛОКАЛЬНА */
    if(st[next]==0)
      dfs(next);
}
```

Параметр `curr` означає поточну вершину (оскільки підпрограма викликає себе рекурсивно, значення `curr` постійно змінюється); змінна `next`, що перебирає вершини, мусить бути локальною. Смисл масиву `st` такий самий, як для `bfs`; тепер ще суттєвіше, що він глобальний.

Підкреслимо відомий факт, що рекурсивні виклики циклу "`for(next:curr→next) if(...) dfs(next)`" працюють так: спочатку повністю завершується один виклик, і лише потім починається інший (або не починається, якщо відповідна вершина вже розглянута). Що якраз і відповідає ідеї «спочатку повністю дослідити одну гілку, лише потім перейти до інших».

Приклад застосування DFS. Більшість умовних позначень і додаткових домовленостей ті самі, що для BFS. Колонка «стек» показує вміст програмного стеку у відповідні моменти: кількість рядків вказує глибину занурення у рекурсію, самі́ чи́сла — значення аргументу `curr` та локальної змінної `next` для кожного з цих викликів. У прикладі пошук починається з $v[1]$, але, взагалі кажучи, з якої вершини треба, з тієї й починаємо.

граф	стек	коментар
	curr | next 1 | 2	перший виклик `dfs` з головної програми; всередині нього, перша ітерація циклу `for next...` розглядає перше по порядку ребро, що виходить з $v[1]$ ($v[1]\to v[2]$).

граф	стек	коментар
	curr | next 1 | 2 2 | 1	відбувся рекурсивний виклик `dfs`, тепер поточною вершиною `curr` є $v[2]$; перша ітерація циклу `for next...` розглядає перше по порядку ребро, що виходить з $v[2]$; це ребро $v[2]\to v[1]$, у вершині $v[1]$ ми вже побували, тому переходимо до наступного значення змінної `next`.
	curr | next 1 | 2 2 | 3	продовжуємо цикл `for next...`, переходимо до ребра́ $v[2]\to v[3]$, у вершині $v[3]$ ще не бували, тож переходимо в неї.
	curr | next 1 | 2 2 | 3 3 | 1	поточна вершина $v[3]$, дивимося ребро $v[3]\to v[1]$, воно веде в уже розглянуту вершину
(без змін)	curr | next 1 | 2 2 | 3 3 | 2	поточна вершина $v[3]$, дивимося ребро $v[3]\to v[2]$, воно веде в уже розглянуту вершину; на цьому перелік ребер, що виходять з $v[3]$, закінчено — отже, цей виклик завершується, виконання програми повертається на попередній рівень рекурсії (той, де $\texttt{curr} = 2$).
	curr | next 1 | 2 2 | 4	продовжуємо цикл `for next...`, переходимо до ребра́ $v[2]\to v[4]$, у вершині $v[4]$ ще не бували, тож переходимо в неї.
	curr | next 1 | 2 2 | 4 4 | 2	поточна вершина $v[4]$, дивимося ребро $v[4]\to v[2]$; воно веде в уже розглянуту вершину, продовжуємо цикл `for next...`
(Надалі, будемо пропускати подібні ситуації «не йдемо в уже розглянуту вершину», не записуючи їх)		
	curr | next 1 | 2 2 | 4 4 | 5	поточна вершина $v[4]$, дивимося ребро $v[4]\to v[5]$, бачимо, що по ньому потрібно перейти.
	curr | next 1 | 2 2 | 4 4 | 5 5 | 6	поточна вершина $v[5]$, ребро $v[5]\to v[4]$ пропускаємо, по ребру $v[5]\to v[6]$ переходимо.
	curr | next 1 | 2 2 | 4 4 | 5 5 | 6 6 | 7	поточна вершина $v[6]$, ре́бра $v[6]\to v[4]$ та $v[6]\to v[5]$ пропускаємо, по ребру $v[6]\to v[7]$ переходимо.
	curr | next 1 | 2 2 | 4 4 | 5 5 | 6 6 | 7 7 | ??	поточна вершина $v[7]$; жодне з ребер, котрі виходять з неї, не веде до нової вершини, тож на цьому відповідний рекурсивний виклик завершується.
(без змін)	curr | next 1 | 2 2 | 4 4 | 5 5 | 6 6 | ??	повернувшися на попередній рівень рекурсії (поточна вершина $v[6]$), бачимо, що вже всі ре́бра, котрі виходять з неї, розглянуті; отже, завершуємо і цей виклик, повертаючись на попередній рівень рекурсивної вкладності

(Далі виклики теж лише завершуються: для $v[5]$ ще є сусід $v[7]$, але ця вершина вже розглянута; для $v[4]$ ще є сусід $v[6]$, але ця вершина вже розглянута; для $v[2]$ цикл перебору сусідів і так вже завершився; для $v[1]$ ще є сусід $v[3]$, але ця вершина вже розглянута. Отже, увесь пошук у глибину остаточно завершується.)

Аналогічно BFS, сам по собі DFS не робить нічого безпосередньо корисного, але чимало графових задач можна розв'язати *на його основі*. Для цього (знов-таки аналогічно BFS) потрібно додати до наведеної вище основної частини тексту `dfs` інші дії, потрібні для відповідної задачі.

Наприклад, DFS-ом можна побудувати матрицю досяжності або виділити компоненти зв'язності неорієнтованого графа. Але са́ме для цих задач придатні хоч $\underline{D}$FS, хоч $\underline{B}$FS; це може створити хибне враження, ніби пошук ушир «кращий» (він шукає мінімальні шляхи, а пошук углиб не шукає). Це не так. Ніякий з цих пошуків не кращий і не гірший, вони просто трохи схожі й трохи різні: є задачі, які можна вирішити будь-яким з них, є задачі, які можні вирішити лише одним, і є задачі, які можні вирішити лише іншим.

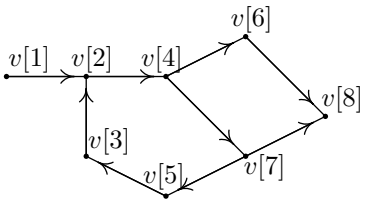
Прикладом задачі, яку легко вирішити модифікацією DFS, але не BFS, є перевірка ациклічності орграфа (згадана на стор. 112 як «геть не геометрична, але типово гра́фова» задачі).

Перевірка ациклічності орграфа рекурсивним DFS-ом Детальніша постановка задачі така: задано орграф, і потрібно з'ясувати, чи містить він хоча б один орієнтований цикл (у найпростішій версії цієї задачі потрібна лише відповідь «"так"/"ні"»; у трохи складнішій версії потрібен ще й сам цикл як послідовність вершин). От і розглянемо, як вирішити цю задачу (найпростішу версію) за допомогою рекурсивного DFS.

(Взагалі кажучи, існують інші, крім розглянутої в цьому посібнику, реалізації пошуку вглиб. Наприклад, *не*рекурсивний пошук углиб, котрий майже як BFS, тільки черга замінена на стек, що працює як контейнер (структура даних): не внаслідок входу у функцію/виходу з функції, а явними ви́кликами дій «покласти у стек» та «вийняти зі стеку». То для того нерекурсивного варіанта всі подальші міркування неправильні, і він не вміє перевіряти ациклічність. Тому́ зараз і говориться са́ме про рекурсивний DFS.)

Твердження. *Орграф містить цикл, що проходить через $v[i_0]$, тоді й тільки тоді, коли рекурсивний DFS, запущений з $v[i_0]$, розглядає серед інших в т. ч. й дугу, яка заходить у $v[i_0]$.*

Наприклад, на орграфі з рис. DFS із $v[2]$ піде у $v[4]$, потім у $v[6]$, у $v[8]$, потім повернеться у $v[6]$, повернеться у $v[4]$, піде у $v[7]$, у $v[5]$, у $v[3]$, і *перевірить дугу* $v[3] \to v[2]$ (звісно, не переходячи по ній, бо $v[2]$ вже відвідана).



Це може призвести до такої ідеї: всере́дині циклу `for (next : curr→next)` замінити «одногілкове» розгалуження «якщо вершина ще не розглянута, запустити пошук із неї» на «двогілкове»: «якщо ще не розглянута, то запустити пошук, а якщо розглянута, то запам'ятати, що знайшли цикл».

На жаль, рівно у такому вигляді це неправда. У тому ж орграфі є вершина $v[8]$, куди ведуть різні орієнтовані шляхи $v[2] \to v[4] \to v[6] \to v[8]$ та $v[2] \to v[4] \to v[7] \to v[8]$, які *не* утворюють циклу. Тому потрібно розрізняти ситуацію повторного розгляду $v[8]$ від повторного розгляду $v[2]$.

Вімінність полягає у тому, що повторний розгляд вершини $v[8]$ відбувається *після відкату* рекурсії з $v[8]$ у $v[6]$, а повторний розгляд $v[2]$ — *всере́дині* виконання пошуку з вершини $v[2]$. Тому, елементи масиву статусів `st` тепер мають зберігати не одне з двох значень, а одне з трьох:

- "`0`" — вершина ще не розглянута;
- "`1`" — сама вершина розглянута, а рекурсивний розгляд досяжних з неї частин орграфа ще триває;
- "`2`" — вже завершено розгляд і самої вершини, і досяжних з неї частин орграфа.

Неважко переконатися, що для підтримання таких значень достатньо міняти елементи `st` у такі моменти:

1. перед першим запуском `dfs` ініціалізувати всі елементи `st` нулями;
2. потрапивши всере́дину рекурсивного виклику `dfs(curr)`, відразу робити присвоєння `st[curr]=1`;
3. перед самим завершенням рекурсивного виклику `dfs(curr)` робити присвоєння `st[curr]=2`.

Отримуємо псевдокод алгоритму перевірки ациклічності:

```
void dfs_cycle(curr) {
  st[curr] = 1;
  for (next : curr→next) /* перебираємо у локальній змінній next
       всі вершини, куди йдуть дýги з вершини curr */
    if(st[next]==0)
      dfs_cycle(next);
    else
      if(st[next]==1)   //!!!
        CYCLE!!! ;
  st[curr] = 2;         //!!!
}
```

На місце фрагмента "`CYCLE!!!`" слід написати дії, які слід робити при виявленні циклу. Якщо ці дії хоч одного разу були виконані, орграф містить цикл; якщо не були — ациклічний.

Ще один тонкий момент — як запускати таку підпрограму?

На все тому ж прикладі, бачимо: якщо запустити її з головної програми однократно, починаючи з вершини $v[8]$ (або $v[6]$), цикл не буде знайдений, бо ні з $v[8]$, ні з $v[6]$ не можна дістатися до циклу. Тому, `dfs_cycle` слід запускати багатократно, починаючи з різних вершин.

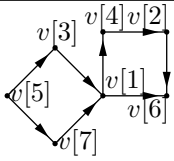
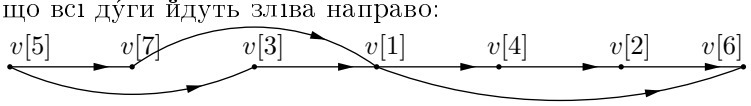
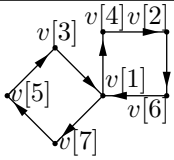
Можна запускати `dfs_cycle` із усіх вершин орграфа, «чистячи» перед кожним запуском масив статусів `st`.

```
for(int v=0; v<n; v++) {
  st.assign(n, 0);
  dfs_cycle(v);
}
```

Але ці запуски можна організувати *значно ефективніше*, уникнувши повторних розглядів вже розглянутих «шматків» графа:

```
st.assign(n, 0);
for(int v=0; v<n; v++)
  if(st[v]==0)
    dfs_cycle(v);
```

Топологічне сортування (topsort) *Задача топологíчного сортування орграфа* (рос. «*задача топологи́ческой сортировки орграфа*», англ. «*digraph topological sorting problem*») полягає в тому, щоб переставити вершини орграфа в такому порядку, щоб усі дýги вели від раніших вершин до пізніших. Цю назву часто скорочують: укр., рос. — «*топсорт*», англ. — «*topsort*».

	Вхідні дані	Результат
Приклад 1		Порядок $v[5]$, $v[7]$, $v[3]$, $v[1]$, $v[4]$, $v[2]$, $v[6]$ забезпечує, що всі дýги йдуть зліва направо:  Якщо переставити $v[7]$ і $v[3]$, вийде інша правильна відповідь
Приклад 2		Топологічне сортування незастосовне до орграфів з циклами

Наголосимо (хоч це й видно з означення), що топологічне сортування *не є* ще одним алгоритмом сортування масивів чи інших послідовностей. Це не алгоритм, а задача, яка істотно відрізняється (щонайменше, наявністю окремо вершин, окремо дуг) від того сортування, яке виконують бульбашкове сортування, quickSort, тощо.

Як співвідносяться означення топологічного сортування, дане за́раз і дане у розд. 2.6.3? Вони дуже схожі, бо між «розмістити менші елементи раніше за більші» та «розмістити вершини, звідки виходять дýги, раніше за вершини, куди вони заходять» дуже глибокий зв'зок, це майже те са́ме. Головна відмінність — для відношень прийнято спочатку переконуватися, що відношення є відношенням порядку (антисиметричним транзитивним), і без цього вважається, що топологічне сортування не має смислу; для орграфів же ніякий аналог транзитивності взагалі не потрібен, замість антисиметричності вживається трохи схожа (але не рівносильна) ациклічність, яку прийнято перевіряти вже під час виконання топсорта, а не зарані.

Не зважаючи на ці відмінності у деталях, цілком правильним є алгоритм топологічного сортування, аналогічний описаному в розд. 2.6.3: *«Поки ще є невибрані вершини, повторювати: вибрати будь-яку вершину, в яку не входить жодна дуга; оголосити, що вона є черговою у поряд-*

Реалізація топологічного сортування рекурсивною функцією, що є черговою модифікацією DFS:

```
void dfs_topsort(curr) {
  st[curr] = 1;
  for (next : curr→next) { /* перебираємо у локальній змінній next
       всі вершини, куди йдуть дуги з вершини curr. */
    if(st[next]==0)
      dfs_topsort(next);
    else
      if(st[next]==1)
        cycle_found = true;
    if(cycle_found)
      return;
  }
  st[curr] = 2;
  res.push_back(curr);
}
```

Виклики цієї рекурсивної функції слід організувати приблизно так:

```
st.assign(n, 0);
res.clear();
cycle_found = false;
for(int v=0; v<n; v++)
  if(st[v]==0) {
    dfs_topsort(v);
    if(cycle_found)
      break;
  };
if(cycle_found) {
  ...(виводимо чи вертаємо, що в орграфі є цикл)...
} else {
  reverse(res.begin(), res.end());
  ...(виводимо чи вертаємо, що res є результатом топсорта)...
}
```

ку зліва направо; вилучити її саму і всі дуги, що з неї виходять. Якщо хоч раз настає ситуація, коли вершини ще є, але вибрати вершину за цим правилом неможливо — орграф містить цикл, топологічне сортування неможливе».

При бажанні, цей самий алгоритм можна використовувати також і для перевірки ациклічності, замість раніше описаної модифікації DFS: орієнтований цикл існує тоді й тільки тоді, коли вершини ще є, але серед них неможливо (навіть після раніше зроблених видалень дуг) вибрати таку, куди не заходить жодна дуга.

Але для розріджених графів це значно менш ефективно, ніж DFS.

І можна, навпаки, доробити DFS-ну перевірку ациклічності, щоб вона і перевіряла ациклічність, і, якщо жодного циклу нема, виконувала топсорт.

Зручно організувати алгоритм так, щоб він намагався будувати результат топсорта (послідовність вершин), «сподіваючись» на ациклічність орграфа; якщо орцикл все-таки знайдеться, можна просто «викинути» (проігнорувати) цю недобудовану послідовність. Тому́ в подальших міркуваннях будемо умовно припускати, ніби орграф ациклічний, хоч це й не гарантовано.

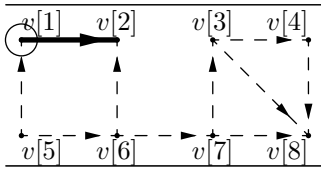
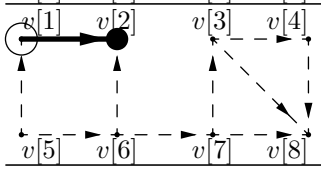
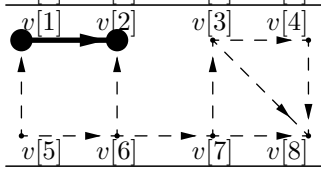
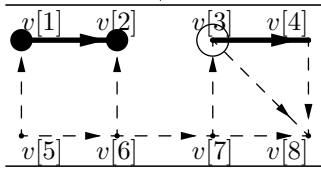
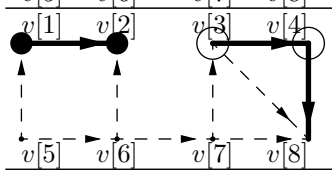
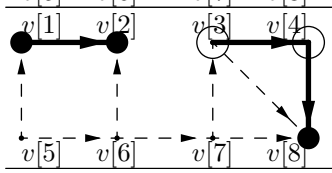
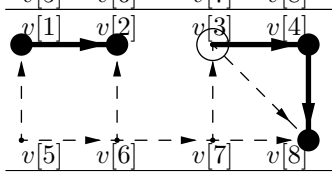
Рекурсивний DFS припиняє рекурсивні заглиблення й починає підйом (відкат) з рекурсії в одній з двох ситуацій: або з вершини не виходить жодна дуга, або всі дуги, що виходять з неї, ведуть у вже відвідані вершини. (Адже, якщо не перше і не дру́ге, то значить є дуги, які ведуть у ще не відвідані вершини, і згідно алгоритму треба в них піти, тобто продовжити рекурсивне заглиблення.) Отже, кожна неможливість заглиблюватись далі в рекурсію обов'язково знаходить щось одне з двох: або цикл (який робить топсорт неможливим), або вершину, всі дуги з якої ведуть лише до вершин статусу 2 (нагадаємо, ми модифікуємо не початковий DFS, а ту версію, яка перевіряє ациклічність і має три, а не два, значення статусу).

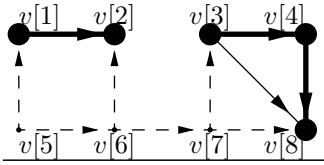
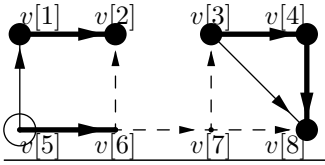
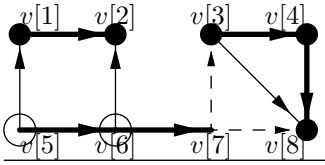
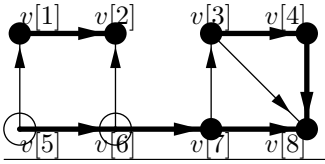
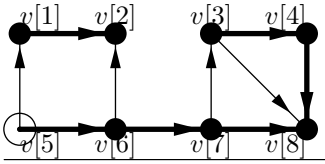
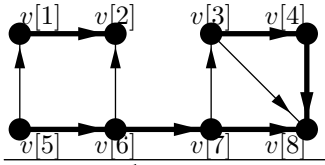
А це значить, що хронологічний порядок підйомів з рекурсії в точності протилежний порядку–результату топсорта: найперший підйом відбудеться у вершині, з якої взагалі не виходять дýги, наступний — у вершині, з якої або взагалі не виходять, або виходить лише дуга у вершину статусу 2, знайдену на попередньому кроці, і т. д. (Повторюємо: все це у припущенні, що орграф ациклічний.) Тобто, можна при кожному підйомі з рекурсії дописувати поточну вершину в деяку глобальну послідовність, а наприкінці розвернути ту послідовність (якщо, звісно, так і не знайдеться цикл).

У наведеній нагорі стор. 138 реалізації, `vector<int> st` (на початку заповнений нулями) є статусом, що має рівно той самий смисл, що й у перевірці ациклічності; `bool cycle_found` зберігає, чи було знайдено цикл; `vector<int> res` (на початку порожній) використовується, щоб побудувати (у зворотньому порядку) послідовність–відповідь.

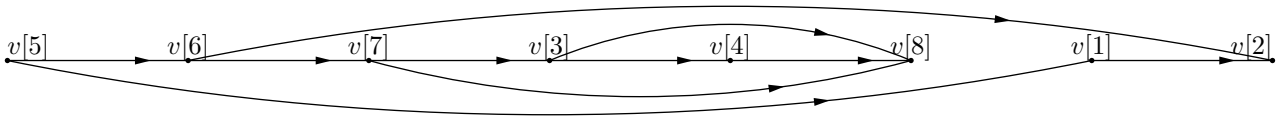
Тобто, такий алгоритм вміє і перевірити, чи містить орграф цикл, і (якщо не містить) знайти який-небудь результат топологічного сортування. І все це — кодом, не набагато складнішим за просто пошук углиб, причому як у плані написання, так і у плані швидкості виконання.

Приклад застосування `dfs_topsort`.

граф	стек	поточний стан відповіді	коментар
	curr | next 1 | 2	порожньо	первинний (а не рекурсивний) виклик `dfs_topsort`($v[1]$); всередині нього, перша ітерація циклу `for next...` розглядає першу по порядку дугу $v[1]\to v[2]$; по ній треба перейти
	curr | next 1 | 2 ~~2~~ | ??	$v[2]$	з поточної вершини $v[2]$ не виходить жодна дуга; відбувається повернення у $v[1]$ (підйом з $v[2]$), тому $v[2]$ набуває статусу 2 й записується у відповідь
	curr | next ~~1~~ | ??	$v[2]$, $v[1]$	з поточної вершини $v[1]$ не виходить жодна *ще не розглянута* дуга; томý $v[1]$ набуває статусу 2 й дописується у відповідь і відбувається остатóчний вихід з рекурсії...
...але продовжується зовнішній цикл; $v[2]$ пропускається як відвідана, томý аргументом наступного первинного (а не рекурсивного) виклику `dfs_topsort` є $v[3]$			
	curr | next 3 | 4	$v[2]$, $v[1]$	поточна вершина $v[3]$, розглядається перша по порядку дуга $v[3]\to v[4]$; по ній треба перейти
	curr | next 3 | 4 4 | 8	$v[2]$, $v[1]$	поточна вершина $v[4]$, розглядається перша по порядку дуга $v[4]\to v[8]$; по ній треба перейти
	curr | next 3 | 4 4 | 8 ~~8~~ | ??	$v[2]$, $v[1]$, $v[8]$	з поточної вершини $v[8]$ не виходить жодна дуга; відбувається повернення у $v[4]$ (підйом з $v[8]$), тому $v[8]$ набуває статусу 2 й дописується у відповідь
	curr | next 3 | 4 ~~4~~ | ??	$v[2]$, $v[1]$, $v[8]$, $v[4]$	з поточної вершини $v[4]$ не виходить жодна ще не розглянута дуга; відбувається повернення з $v[4]$ у $v[3]$, томý $v[4]$ набуває статусу 2 й дописується у відповідь

граф	стек	поточний стан відповіді	коментар
	curr | next ~~3~~ | ??	v[2], v[1], v[8], v[4], v[3]	спочатку, розглядається й пропускається v[3]→v[8], що веде до розглянутої v[8]; потім не лишається не розглянутих дуг, v[3] набуває статусу 2 й дописується у відповідь, відбувається вихід з рекурсії...
...але продовжується зовнішній цикл; v[4] пропускається як відвідана, томý аргументом наступного первинного (а не рекурсивного) виклику `dfs_topsort` є v[5]			
	curr | next 5 | ~~1~~ 6	v[2], v[1], v[8], v[4], v[3]	спочатку, розглядається й пропускається v[5]→v[1], що веде до розглянутої v[1]; потім розглядається наступна по порядку дуга v[5]→v[6], по ній треба перейти
	curr | next 5 | 6 6 | ~~2~~ 7	v[2], v[1], v[8], v[4], v[3]	спочатку, розглядається й пропускається v[6]→v[2], що веде до розглянутої v[2]; потім розглядається наступна по порядку дуга v[6]→v[7], по ній треба перейти
	curr | next 5 | 6 6 | 7 ~~7~~ | ??	v[2], v[1], v[8], v[4], v[3], v[7]	спочатку, розглядаються й пропускаються v[7]→v[3] та v[7]→v[8], що ведуть до розглянутих v[3] та v[8]; потім не лишається не розглянутих дуг, томý v[7] набуває статусу 2 й дописується у відповідь і відбувається повернення з v[7] у v[6]
	curr | next 5 | 6 ~~6~~ | ??	v[2], v[1], v[8], v[4], v[3], v[7], v[6]	з поточної вершини v[6] не виходить жодна ще не розглянута дуга; томý v[6] набуває статусу 2 й дописується у відповідь і відбувається повернення з v[6] у v[5]
	curr | next ~~5~~ | ??	v[2], v[1], v[8], v[4], v[3], v[7], v[6], v[5]	з поточної вершини v[5] не виходить жодна ще не розглянута дуга; томý v[5] набуває статусу 2 й дописується у відповідь і відбувається остатóчний вихід з рекурсії
цикл перебору не відвіданих вершин (ззовні `dfs_topsort`) перевіряє, що v[6], v[7] та v[8] вже відвідані, і все разом узяте топологічне сортування остаточно завершується; результатом є «розвернута» послідовність «поточний стан відповіді».			

Тобто, остаточним результатом топологічного сортування є послідовність $v[5]$, $v[6]$, $v[7]$, $v[3]$, $v[4]$, $v[8]$, $v[1]$, $v[2]$. Для неї всі дýги справді напрямлені зліва направо:



Пошук углиб має й інші корисні застосування — перевірка сильної зв'язності (ефективніша, ніж побудова матриці зв'язності); перевірка двозв'язності, пошук мостів та точок зчленування; тощо. Але ті алгоритми складніші й явно лежать за межами дискретної *математики*.

5.7 Алгоритми пошуку найкоротших маршрутів у зважених графах

Пошук ушир знаходить маршрут мінімальної довжини́, вважаючи, що довжиною маршрута є кількість ребер. Але на практиці пошук найкоротших маршрутів частіше стосується зважених графів, де ре́бра мають довжину, і довжиною маршрута вважається сума довжин пройдених ребер.

5.7.1 Алгоритм Дейкстри

Алгоритм Дейкстри[41] дозволяє знайти найкоротші маршрути від вказаної початкової вершини зваженого графа до всіх інших. Цей алгоритм може бути застосований як до орграфів, так і до неорієнтованих; він *не* вимагає виконання нерівності трикутника для довжин ребер[42]. Але одне обмеження накладається: ***довжи́ни ребер мають бути невід'ємними!***

Згідно алгоритму Дейкстри, кожній вершині співставляються оцінка відстані від початкової (масив `d`) та *статус* (масив `st`). Статус може набувати одне з трьох значень:

- статус 0 (або «*біла*» вершина, або «*спляча*») — до цієї вершини ще не дісталися, про її ві́дстань від початкової ще нічого не відомо;
- статус 1 (або «*сіра*» вершина, або «*розбуджена*») — до цієї вершини вже дісталися, отже — є *деяка оцінка* її відстані від початкової; але *ще не відомо, чи* ця оцінка і буде остато́чною відстанню, чи вона покращиться (буде знайдено інший, коротший, маршрут);
- статус 2 (або «*чорна*» вершина, або «*вбита*») — для цієї вершини вже відомі гарантовано мінімальний маршрут та остато́чна відстань.

Довжину ребра́ від i-ої вершини до j-ої позначатимемо як $l(i,j)$; залежно від структури даних, це може бути `[i][j]`-ий елемент матриці суміжності, чи частина структури елемента списків суміжності, тощо.

Сам алгоритм має такий вигляд:

Спочатку, всім вершинам надається статус 0, а стартовій (тій, маршрути від якої шукаємо) — статус 2 та відстань 0. Стартова вершина запам'ятовується як «поточна». Далі відбувається великий цикл, кожна ітерація якого складається з двох етапів:

1. Досліджуються усі сусіди поточної вершини, і, в разі потреби, оновлюються їхні (сусідніх вершин) оцінки.
   Потреба оновити оцінки виникає в одному з двох випадків: або коли щойно вперше знайдено хоч якийсь маршрут до цієї вершини, або коли щойно розглянутий маршрут кращий (коротший) за раніше відомий. Такі «оновлення в разі потреби» називають також *релаксаціями* (рос. «*релаксация*», англ. «*relaxation*»).
2. Серед вершин, які мають статус 1, вибирається та, оцінка відстані якої мінімальна; ця вершина переводиться до статусу 2 і робиться поточною для наступної ітерації великого циклу.
   (При цьому *не* враховується, «наскільки давно» вершини перейшли зі статусу 0 до статусу 1; якщо є різні вершини з однаковими мінімальними оцінками, береться будь-яка одна з них.)

Завершувати великий цикл слід тоді, коли не лишається вершин статусу 1.

Сформульовано словами, бо реалізовувати суть пунктів 1 та 2 можна дуже, дуже по-різному.

(Найпростіша реалізація — подати граф матрицею суміжності і написати для кожного з пунктів свій цикл перегляду всіх вершин: для п. 1 — проходження по рядку матриці суміжності, для п. 2 — вибір мінімума з оцінок відстаней через цикл та (`if(st[i]==1 && d[i]<d_min) ...`). Тоді кількість дій алгоритму становить $O(n^2)$: $O(n)$ разів зовнішній цикл, всере́дині нього послідовно два цикли по $O(n)$ кожен.

Для великих розріджених графів, значно ефективніше (складність $O(m \log n)$) вчинити інакше: 1) використати на першому великому кроці списки суміжності; 2) замінити очевидний пошук мінімума серед усіх елементів `d` на пошук мінімума за допомогою спеціальної структури даних (піраміди або `set`-а; детальніше — у дисципліні «Алгоритми та структури даних»).

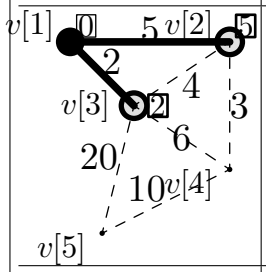
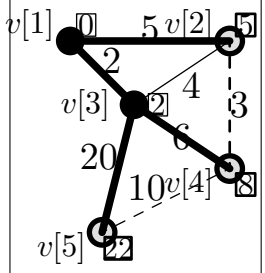
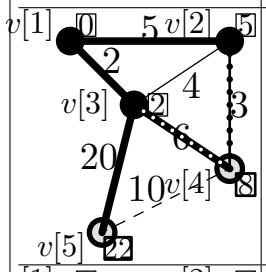
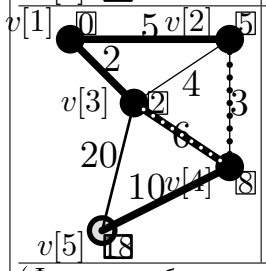
І спільного між цими двома реалізаціями дуже ма́ло, хоча обидві вони в точності відповідають наведеному словесному формулюванню.)

Для знаходження маршрутів як послідовностей вершин можна, як і в BFS, ввести масив попередників `previous` та скористатися зворотнім ходом.

[41] Е́дсгер Де́йкстра (Edsger Dijkstra; 1930–2002) — один з найвидатніших вчених у галузі computer science, лауреат премії Тьюрінга, професор Ейндховенського університету (Нідерланди) та Техаського університету (США)

[42] тобто, працює правильно в тому числі й у випадку, коли коли у графі можуть бути ре́бра $\{v[i],v[j]\}$, $\{v[j],v[k]\}$ та $\{v[i],v[k]\}$ такі, що $l(\{v[i],v[k]\}) > l(\{v[i],v[j]\}) + l(\{v[j],v[k]\})$, тобто маршрут через проміжну вершину $v[j]$ коротший за «пряме» ребро

Вершини статусу 0 не обведені; статусу 1 обведені ко́лом; статусу 2 залиті чорним. Не розглянуті ре́бра проведені пунктиром; розглянуті — суцільними лініями (тонка лінія — ребро не використовується у найкоротших маршрутах, жирна — використовується). Чи́сла посере́дині ребер — довжи́ни ребер; чи́сла, обведені прямокутниками — оцінки відстаней вершин від початкової. "`curr` (стара)" — яка вершина поточна у пункті 1; "`curr` (нова)" — яка вершина стає новою поточною у пункті 2.

граф	`curr` (стара)	`curr` (нова)	коментар
	1	3	У першій великій частині циклу, $v[2]$ та $v[3]$ переходять зі статусу 0 до статусу 1, їм надаються оцінки $d[2]=5$ та $d[3]=2$. У дру́гій, $v[3]$ переходить до статусу 2.
	3	2	У першій великій частині циклу, досліджуються маршрути через $v[3]$ до $v[2]$, $v[4]$ та $v[5]$. Новий маршрут до $v[2]$ довший, ніж уже відомий ($d[3]+l(3,2)=$ $=2+4>5=d[2]$), тому оцінка $d[2]$ не змінюється. Для $v[4]$ та $v[5]$ маршрути через $v[3]$ є першими знайденими, тому ці вершини переходять до статусу 1, їм надаються оцінки $d[4]=8$, $d[5]=22$. У дру́гій великій частині циклу, серед $v[2]$, $v[4]$, $v[5]$ вибирається $v[2]$ і переводиться до статусу 2.
	2	4	У першій великій частині циклу, досліджуються маршрути через $v[2]$. Є́диний її сусід, що перебуває не у статусі 2 — $v[4]$. Довжина нового маршруту дорівнює довжині старого ($d[2]+l(2,4)=$ $=5+3=8=d[4]$), тому оцінка не змінюється. Вершину-попередника можна хоч лишити незмінною ($v[3]$), хоч змінити на $v[2]$ — і так, і так правильно. У дру́гій великій частині циклу, серед $v[4]$ та $v[5]$ вибирається $v[4]$ і переводиться до статусу 2.
	4	5	У першій великій частині циклу, досліджуються маршрути через $v[4]$. І виявляється, що досить давно знайдена оцінка $d[5]$ неоптимальна: новий маршрут коротший ($d[4]+l(4,5)=8+10=$ $=18<22=d[5]$), тому оцінка замінюється на нову ($d[5]=18$). У дру́гій великій частині циклу $v[5]$ переводиться до статусу 2.

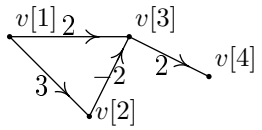
(Фактично буде ще одна ітерація великого циклу, на якій будуть (невдалі) спроби побудувати нові маршрути через вершину 5 та (невдала) спроба знайти мінімальну оцінку серед вершин зі статусом 1, але її малювати не будемо.)

Якщо потрібно шукати найкоротші маршрути *не в усі* вершини, а лише *між двома вказаними* вершинами `v_start` та `v_finish`, це *не* дає можливості *значно* оптимізувати алгоритм: чи не є́дине, що можна зробити — замінити умову виходу з великого циклу на «не лишилося вершин статусу 1 *або вершина* `v_finish` *вже у статусі 2*», а така оптимізація не є принциповою.

Приклад застосування алгоритму Дейкстри див. нагорі стор. 142.

Доведення правильності алгоритму Дейкстри для зважених графів з невід'є́мними довжинами ребер охочі можуть знайти у літературі.

А от приклад, коли спроба застосувати алгоритм Дейкстри до графа з ребрами від'є́мної довжини́ призводить до помилкової відповіді, наведемо.



$v[1]$	$v[2]$	$v[3]$	$v[4]$
0	3	1	3

У орграфі з рис. нема *циклів* від'є́мної довжини́, тож відстані як довжи́ни найкоротших маршрутів мають смисл; значення відстаней від $v[1]$ наведені у табличці (можна перевірити перебором).

А при застосуванні алгоритму Дейкстри (починаючи з $v[1]$), серед $v[2]$ і $v[3]$ до статусу 2 переводиться спочатку $v[3]$, і буде вважатися $d[3]=2$. Як наслідок, неправильно визначається також і $d[4]$.

5.7.2 Алгоритми Флойда та Воршалла

Алгоритм Флойда	Алгоритм Воршалла

`for(int k=0; k<n; k++)`
`  for(int i=0; i<n; i++)`
`  for(int j=0; j<n; j++)`
`  d[i][j] = min(d[i][j],`
`  d[i][k] + d[k][j]);`
	`for(int k=0; k<n; k++)`
	`  for(int i=0; i<n; i++)`
	`  for(int j=0; j<n; j++)`
	`  d[i][j] |= d[i][k] & d[k][j];`

Цикли не можна переставляти довільно: *зовнішнім мусить бути цикл, що перебирає проміжні вершини* k суми `d[i][k]+d[k][j]` чи кон'юнкції `d[i][k]&d[k][j]`.

(Оскільки розглядаємо мінімальні маршрути, основну увагу приділимо алгоритму Флойда (Floyd's). Але заодно побіжно згадаємо й алгоритм Воршалла (Warshall's); з цього і почнемо.

Алгоритм Воршалла будує транзитивне замикання відношення: якщо перед запуском алгоритму покласти у масив `d` матрицю бінарного «у вузькому смислі» відношення, то після виконання цих циклів той самий масив `d` міститиме вже матрицю транзитивного замикання цього відношення.

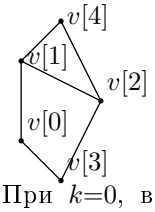
Це майже те са́ме, що побудова матриці досяжності за відомою матрицею суміжності (у варіанті, коли матриця суміжності може містити лише 0 (нема ребра́) та 1 (ребро є)), але не враховується досяжність кожної вершини самої з се́бе за маршрутом довжини́ 0. Якщо треба, можна дописати цикл, що замінює всі елементи головної діагоналі на одиниці; тоді це буде в точності побудова матриці досяжності.)

Тепер зосередимося на алгоритмі Флойда. На початку (перед запуском циклів) масив `d` має містити спеціальну форму матриці суміжності, де всі діагональні елементи `d[i][i]`$=0$, а відсутність ребер позначається ∞ (див. стор. 118). По завершенню роботи алгоритму цей самий масив містить матрицю відстаней.

Доведення правильності алгоритму Флойда відбувається за такою схемою. Перед початком робо́ти враховано тривіальні маршрути, які або взагалі не містять ребер, або містять єдине ребро. Після ітерації зовнішнього циклу при k=0, враховано ще й маршрути, які проходять через єдину проміжну вершину $v[0]$. Після ітерації при k=1 — маршрути, які можуть проходити через $v[0]$ та/або $v[1]$. Причому, з варіантів

не проходити ні через $v[0]$, ні через $v[1]$	`d[i][j]` не було змінене ні при k=0, ні при k=1
проходити через $v[0]$, але не проходити через $v[1]$	`d[i][j]` змінене при k=0, але не змінене при k=1
проходити через $v[1]$, але не проходити через $v[0]$	`d[i][j]` змінене при k=1, причому ні `d[i][1]`, ні `d[1][j]` не мінялися при k=0
проходити спочатку через $v[0]$, потім через $v[1]$	`d[i][j]` змінене при k=1, причому `d[i][1]` змінене при k=0, а `d[1][j]` — ні
проходити спочатку через $v[1]$, потім через $v[0]$	`d[i][j]` змінене при k=1, причому `d[1][j]` змінене при k=0, а `d[i][1]` — ні
проходити через $v[0]$, потім через $v[1]$, потім знов через $v[0]$	`d[i][j]` змінене при k=1, причому і `d[i][1]`, і `d[1][j]` змінені при k=0

вибрано найкращий. Аналогічно, після ітерації при k=2 вибрано найкращі зі маршрутів, які можуть (але не зобов'язані) проходити (в будь-якому порядку) через $v[0]$, $v[1]$ та $v[2]$. І так далі. Тому, після виконання усіх ітерацій, масив `d` містить ві́дстані. ■



Застосуємо алгоритм Флойда до наведеного графа. На початку, треба підготувати матрицю суміжності.

0	1	∞	1	∞
1	0	1	∞	1
∞	1	0	1	1
1	∞	1	0	∞
∞	1	1	∞	0

При k=0, враховано маршрут довжиною 2 між $v[1]$ та $v[3]$ *через* $v[0]$ (тут і далі — в обох напрямках)

0	1	∞	1	∞
1	0	1	***2***	1
∞	1	0	1	1
1	***2***	1	0	∞
∞	1	1	∞	0

При k=1, враховано маршрути *через* $v[1]$: між $v[0]$ та $v[2]$, між $v[0]$ та $v[4]$, між $v[3]$ та $v[4]$

0	1	***2***	1	***2***
1	0	1	2	1
2	1	0	1	1
1	2	1	0	***3***
2	1	1	***3***	0

При k=2, враховано маршрут між $v[3]$ та $v[4]$ *через* $v[2]$, який виявляється коротшим, ніж через $v[0]$ та $v[1]$

0	1	2	1	2
1	0	1	2	1
2	1	0	1	1
1	2	1	0	***2***
2	1	1	***2***	0

Далі будуть ще ітерації при k=3 та k=4, які не змінюють жодного елементу. Але це — випадковий збіг обставин са́ме для цього графа, а не загальна властивість алгоритму. Наприклад, для графа $v[0]$ $v[2]$ $v[1]$ спочатку відбувається дві ітерації, на яких масив не міняється, а вже потім при k=2 враховується маршрут через $v[2]$.

Як бачимо, алгоритм Флойда будує відразу всю матрицю відстаней. Якщо са́ме матриця й потрібна, алгоритм Флойда безсумнівно найлегший для написання, і при цьому ще й оптимальний за часом виконання. Але якщо потрібні лише деякі відстані, нема ніякої можливості пришвидшити цей алгоритм, не шукаючи не потрібні.

(*Якщо* граф розріджений і не містить ребер від'ємної довжини́, то запускати n разів (з кожної вершини) *ефективний* варіант алгоритму Дейкстри (про який дуже побіжно згадано на стор. 141) швидше. Але якщо граф щільний, або якщо користуватися найпростішою версією алгоритму Дейкстри, то швидший алгоритм Флойда.)

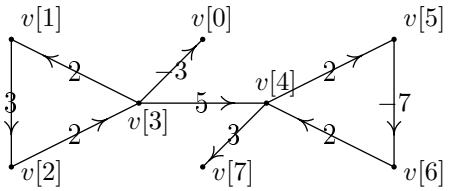
Алгоритм Флойда коректно працює з ребрами від'ємної довжини́. Згадаємо (див. також стор. 122), що головна проблема з від'ємними довжинами ребер полягає в тому, що в разі наявності *циклічних* маршрутів від'ємної довжини́ поняття відстані втрачає смисл, «уходячи в $-\infty$». А коректність алгоритму Флойда полягає в таких двох властивостях.

1) Якщо у графі нема циклічних маршрутів від'ємної довжини́, то алгоритм Флойда знаходить найкоротші маршрути правильно — навіть якщо орграф містить ду́ги від'ємної довжини́ (не буває проблем, подібних до проблеми з алгоритмом Дейкстри, описаної на стор. 142).

(Наведені вище пояснення того, як працює алгоритм Флойда, ніяк не змінюються, коли є ду́ги (але нема циклів) від'ємної довжини́.) ■

2) Орграф містить циклічні маршрути від'ємної довжини́ тоді й тільки тоді, коли після виконання алгоритму Флойда на головній діагоналі з'являється хоча б одне від'ємне значення. (Таким чином, алгоритм Флойда може діагностувати таку ситуацію.)

Як розуміти, якщо від'ємні значення з'являються у деяких елементах головної діагоналі, але не в усіх? На жаль, тут можливі різні варіанти.

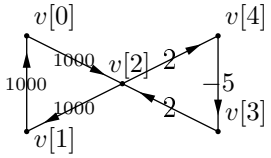


Для орграфів, що не є сильно зв'язними, можлива ситуація, коли поняття відстані втратило смисл для деяких пар вершин, але не всіх.

Наприклад, на рисунку зображено орграф, де є від'ємний цикл $v[4] \to v[5] \to v[6] \to v[4]$, але в деяких його частинах (зокрема, підграфі, породженому $\{v[0],\ v[1],\ v[2],\ v[3]\}$) діє «звичайна» відстань, бо звідти можна вийти, але, «накрутивши ($-\infty$) уздовж $v[4] \to v[5] \to v[6] \to \to v[4]$», неможливо повернутися.

І внаслідок виконання на цьому орграфі алгоритму Флойда від'ємними на головної діагоналі стають лише `d[4][4]`, `d[5][5]`, `d[6][6]`.

Це може схилити до думки, ніби якщо лише частина елементів головної діагоналі стали від'ємними, то лише у підграфі, породженому цими вершинами, поняття відстані втратило смисл. *Але це неправда.* Хоча б тому, що на все тому ж графі є ще вершина $v[7]$, для якої значення `d[7][7]` як було 0, так і лишилось, а відстані до неї (з усіх вершин, крім самої себе та $v[0]$) пішли у $-\infty$ і втратили смисл.



І це не «виключення лише для тупикових вершин». Праворуч наведено орграф, для якого від'ємними на головній діагоналі стають лише `d[2][2]`, `d[3][3]` та `d[4][4]`.

При цьому *існують* цикли від'ємної довжини́, що проходять через $v[0]$ та $v[1]$ — вигляду $v[0] \to \Big(\underbrace{v[2] \to v[4] \to v[3] \to v[2]}_{\text{повторюється } \geqslant 3001 \text{ разів}}\Big) \to v[1] \to v[0]$. Настільки багатьох повторів $v[2]$, $v[4]$, $v[3]$ алгоритм Флойда не знаходить...

Сформулюємо узагальнену задачу пошуку шляхів мінімальної довжини́ так. *«Знайти матрицю відстаней заданого зваженого (довжи́ни дуг довільного знаку) орграфа, так, щоб матриця могла містити і значення "$+\infty$" (недосяжна) і "$-\infty$" (можливий маршрут, що «накручує» цикл від'ємної довжини́), і конкретні звичайні чи́сла у решті випадків.»*

Бездоказово заявимо, що її можна правильно розв'язати так.

(1) Вибрати таке поєднання компілятора та ОС, щоб стандартний числовий тип підтримував значення "$+\infty$" та "$-\infty$" (за́раз таких більшість, див. також стор. 118).

(2) Ініціалізацію елементів, відповідних парам вершин, між якими нема дуги́, робити са́ме тими значеннями "$+\infty$", які забезпечує тип, а не `1e+300` чи якимсь схожим чином.

(3) Переконатися, що у ситуаціях, коли `d[i][j]` є звичайним числом у розумних межах, а сума `d[i][k] + d[k][j]` являє собою $(+\infty)+(-\infty)$ чи $(-\infty)+(+\infty)$, дія `d[i][j] = min(d[i][j], d[i][k] + d[k][j])` не змінює значення `d[i][j]`. Хоч функція `min` і готова бібліотечна, і є надія, що вона правильна, але деталі реалізації бібліотек різних компіляторів все-таки різні, а ми намагаємось використати функцію у нестандартній ситуації; тому треба ретельно перевірити, і, в разі потреби, переписати.

(Наприклад, при виконанні у Microsoft Visual Studio Express 2013 for Windows Desktop присвоєння `d[i][j] = min(d[i][j], d[i][k] + d[k][j])` все працює правильно, а присвоєння `d[i][j] = min(d[i][k] + d[k][j], d[i][j])` (де *лише* обміняні місцями аргументи `min`) — неправильно (значення `d[i][j]` «псується»). Причому, корисна у багатьох подібних випадках порада «запускати у Debug-режимі» са́ме тут нічого не дає — програма все одно мовчки працює неправильно.

Причина у тому, що порівняння звичайного числа́ з невизначеністю $(+\infty)+(-\infty)$ чи $(-\infty)+(+\infty)$ осмислене хіба що з результатом `unknown` у тризначній логіці (розд. 1.8), а бібліотечна функція `min` нічого не знає ні про тризначну логіку взагалі, ні про те, що кокретно цій задачі потрібно у ситуації `unknown`. Так що наявність готових "$+\infty$" та "$-\infty$" приносить не лише зручності, а також і проблеми.

Наскільки відомо автору посібника, спосіб «написати руками (без функції `min`) `dist_new = d[i][k] + d[k][j]; if(dist_new < d[i][j]) d[i][j] = dist_new;`» начебто повинен працювати завжди.)

(4) Запустити стандартні вкладені цикли `for for for` алгоритму Флойда (реалізовані з урахуванням всього вищесказаного).

(5) Пройти по головній діагоналі матриці (й лише по ній), замінюючи всі від'ємні (але не нульові́!) значення на "$-\infty$". Так би мовити, посилити переконливість того, що ця вершина належить циклу від'ємної довжини́...

(6) *Ще раз* запустити такі самі стандартні вкладені цикли `for for for` алгоритму Флойда, реалізовані з урахуванням всього вищесказаного.

Результатом буде матриця відстаней, яка правильно містить і значення "$+\infty$" (недосяжна) і "$-\infty$" (можна «накручувати» цикл від'ємної довжини́), і правильні конкретні звичайні чи́сла-відстані у решті випадків.

Відновлення маршрутів як послідовностей вершин для алгоритму Флойда можливе, але організовується зовсім інакше, ніж для BFS чи алгоритму Дейкстри. У масиві `next` розміром $n \times n$, `next[i][j]` має смисл *«Яка вершина є першою проміжною на найкоротшому маршруті з $v[i]$ до $v[j]$?»*.

Основна частина алгоритму набуває вигляду:

```
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    next[i][j] = j;                 //!!!
for(int k=0; k<n; k++)
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      if(d[i][k] + d[k][j] < d[i][j]) {
        d[i][j] = d[i][k] + d[k][j];
        next[i][j] = next[i][k];    //!!!
      }
```

Відновлення маршруту з `st` до `fin` навіть простіше, ніж для BFS чи Дейкстри:

```
cout << st;
c = st;
while(c!=fin) {
  c = next[c][fin];
  cout << " " << c;
}
```

5.8 Дере́ва (неорієнтовані)

Ми розглянемо лише неорієнтовані дере́ва. Коренéві дере́ва, що є основою деяких структур даних у програмуванні, мають дещо спільне з цими деревами, але насправді небагато.

Скрізь у цьому розділі 5.8, де вживається слово «граф» і не вказується його вид, мається на увазі неорієнтований, без петель, без кратних ребер ("не мульти-") граф; а от зваженість скрізь дозволяється, а в деяких місцях навіть вимагається.

5.8.1 Означення та властивості дерев

(Неорієнтоване) дерево — це...

- ... ациклічний зв'язний граф (де *ациклічний* — такий, що не містить жодного циклу);
- ... зв'язний граф, у якого $m = n - 1$ (як завжди, n — кількість вершин, m — ребер);
- ... ациклічний граф, у якого $m = n - 1$;
- ... граф, у якому для будь-яких (різних) вершин v та w існує єдиний простий ланцюг, що з'єднує v і w;
- ... такий ациклічний граф, що коли будь-які дві несуміжні вершини v і w з'єднати ребром $\{v, w\}$, то одержаний граф міститиме в точності один цикл.

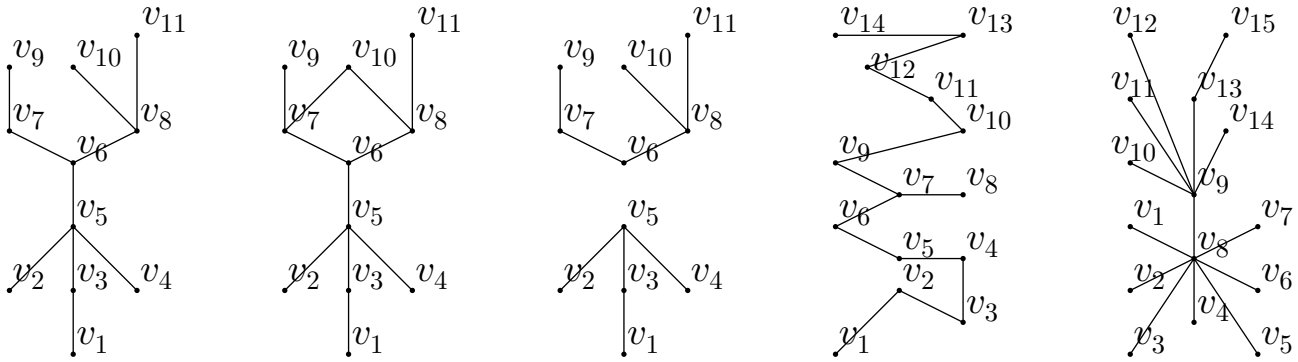
Всі наведені означення рівносильні, тобто якщо граф є деревом за будь-яким одним з цих означень, то він є деревом і за будь-яким іншим; по-чесному це слід було б довести...

Ліс — це ациклічний граф (не обов'язково зв'язний). Очевидно, ліс «складається з дерев»; більш строго, дере́ва є компонентами зв'язності лісу.

(За цими означеннями виявляється правильною досить «дика» з «житейської» точки зору фраза *«якщо у дереві виламати кілька гілок, то вийде ліс»* ☺).

На рис. (a), (d) та (e) зображено дере́ва; рис. (b) та (c) не є деревами: (b) має цикл $v_6 - v_7 - v_{10} - v_8 - v_6$, граф (c) не зв'язний.

Ліс зображено на рис. (a), (c), (d), (e). При цьому ліс (c) складається з двох дерев, а кожен з (a), (d) та (e) — з одного.



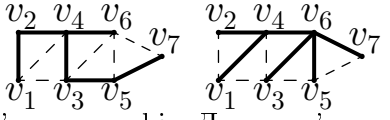
Відзначимо (без доведення), що у будь-якому дереві (за виключенням одновершинного графа K_1) є щонайменше дві висячі вершини.

Як правило, висячих вершин значно більше (наприклад, у де́ревi з рис. (e) висячими є 12 вершин (усі крім v_8, v_9 та v_{13})); але, скажімо, у дереві з рис. (d) лише три висячі вершини (v_1, v_8 та v_{14}). А якщо дерево зовсім «витягнуте у ланцюжок», то рівно 2.

5.8.2 Задача побудови остовного де́рева мінімальної ваги. Алгоритми Краскала та Пріма

Остовне[43] (терміни-синоніми: *каркасне*, *кістякове*, *стяжне*; рос. «*о́стовное*», «*каркасное*», «*покрывающее*», «*скелетное*»; англ. «*spanning*») *дерево* неорієнтованого графа — це такий підграф[44], що: 1) множина вершин підграфа дорівнює множині вершин початкового графа; 2) підграф є деревом.

(*Приклад.* Два різні остовні дерева одного й того ж графа. Ре́бра остовного де́рева зображені жирними лініями, ре́бра початкового графа, які не увійшли до остовного де́рева — пунктиром.)



Зрозуміло, що поняття остовного де́рева має смисл лише для зв'язних графів. Для незв'язних графів можливий *остовний ліс* — ліс, кожне дерево якого є остовним деревом відповідної компоненти зв'язності початкового графа.

[43] див. також стор. 116 (означення остовного підграфа та примітку 35)

[44] майже завжди не правильний

Кількість ребер будь-якого остовного де́рева дорівнює $n-1$. Але для зважених графів сума довжин ребер різних остовних дерев може бути різною. Це можна бачити хоч би й на прикладі того самого рисунку: якщо поміряти довжи́ни відрізків–ребер лінійкою, то в першому дереві (де п'ять з шести відрізків або вертикальні, або горизонтальні) сума довжин буде менша.

Так виникає задача побудови *остовного де́рева мінімальної ваги* (*ОДМВ*): серед усіх можливих остовних дерев знайти те, сума довжин ребер якого якнайменша. ОДМВ називають також *мінімальним остовним деревом* (*МОД*). Тобто, назви МОД і ОДМВ означають одне й те само.

(Рос. — «*остовное дерево минимального веса*» (*ОДМВ*), «*минимальное остовное дерево*» (*МОД*); англ. — «*minimum spanning tree*» (*MST*).)

ОДМВ не має стосунку до найкоротших шляхів. Це просто інша задача, з іншими вимогами. Тут є вимога задіяти всі вершини графа. Тут *нема* ні вказаних старту та фінішу, ні навіть чогось одного з них. Тут мінімізується сума довжин ребер *де́рева, а не маршруту*.

(Та і в ОДМВ, зображеному на першому з рисунків 6 абзаців тому́, неможливо пройти з $v[6]$ у $v[7]$ коротше, ніж через $v[4]$, $v[3]$, $v[5]$; це не має нічого спільного з маршрутом мінімальної довжини́.)

А до чого має стосунок? Наприклад, нехай є аудиторії, між якими треба прокласти кабелі зв'язку (без використання бездротових технологій); для кожної пари аудиторій відомо, чи можливо прокласти кабель між ними, і, якщо можливо, то скільки це коштуватиме. Якщо метою є *«прокласти кабелі так, щоб від будь-якої аудиторії був зв'язок із будь-якою іншою (годиться як безпосередній через один кабель, так і через будь-який ланцюжок з довільною кількістю проміжних аудиторій); з усіх варіантів вибрати той, де сумарні витрати мінімальні»* — тоді це задача побудови ОДМВ.

Практичну цінність ОДМВ зменшує те, що ігнорується питання, чи зручно потім буде користуватися мережею, чи не призводитиме завелика кількість проміжних пунктів (чи завелика кількість розгалужень) до істотних проблем. Але якщо не призводитиме, а мінімізувати витрати на розбудову мережі важливо, то це виходить приклад практичного використання ОДМВ. І якраз із кабелями шанси на таку ситуацію є. А якщо аналогічно вибирати, які з можливих доріг між містами побудувати — шансів значно менше, бо від того, які дороги вибрати, залежить, скільки забиратиме часу й коштуватиме грошей майбутній проїзд, а ОДМВ це ігнорує.

Задачу побудови ОДМВ найчастіше розв'язують одним з двох алгоритмів — алгоритмом Краскала (Kruskal's algorithm; зустрічаються і транскрипція «Кра́скал», і «Кру́скал») або алгоритмом Пріма (Prim's algorithm).

Алгоритм Краскала На початку в алгоритмі Краскала розглядають ліс, який містить усі вершини початкового графа і не містить жодного ребра́; таким чином, кожна вершина ізольована і являє собою окрему компоненту зв'язності. Потім до цих вершин додають ре́бра (описаним далі способом).

Усі ре́бра початкового графа впорядковують за неспаданням довжин, щоб розглядати їх у порядку від найкоротшого до найдовшого. При розгляді кожного ребра́ перевіряють, чи різним компонентам зв'язності належать його кінці: якщо різним, то це ребро додають до поточного лісу (а якщо одній і тій самій — пропускають, щоб не утворити цикл).

Час роботи алгоритму Краскала сильно залежить від того, *як* перевіря́ти, чи одній компоненті зв'язності належать вершини. *Не варто* робити це за допомогою пошуку в графі, бо це *і* неефективно, *і* не найлегший для програмування спосіб.

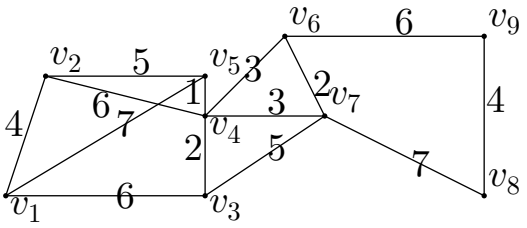
Можна підтримувати масив `c`, індекси якого відповідають вершинам графа, значення — умовному номеру компоненти зв'язності, якій належить відповідна вершина (ці уявні номери іноді називають також «кольорами»). Тоді «$v[j]$ та $v[k]$ належать різним компонентам» виражається як "`c[j]≠c[k]`" («різнокольорові»). Звісно, щоб масив `c` містив правильні значення, його потрібно правильно модифікувати при кожному додаванні ребра́ до графа (щоб злиті в одну компоненту вершини ставали «однокольоровими»).

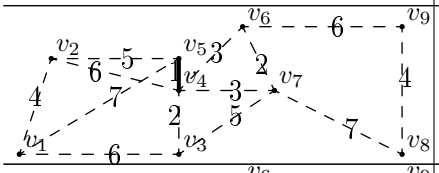
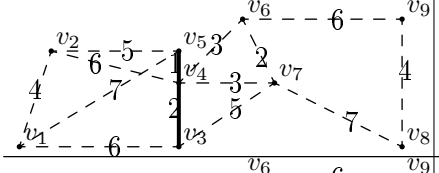
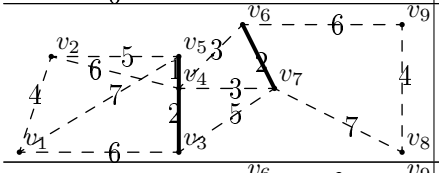
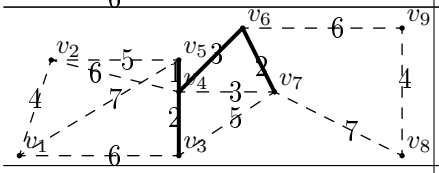
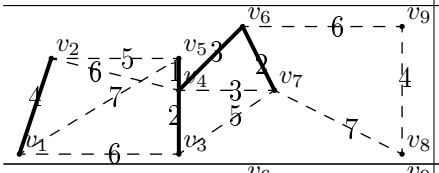
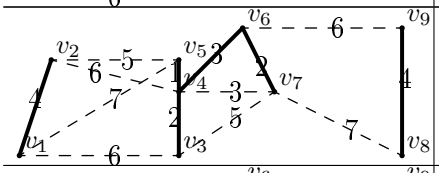
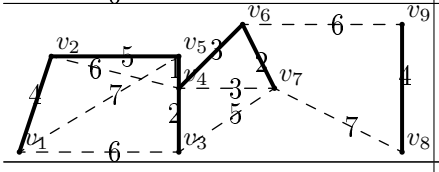
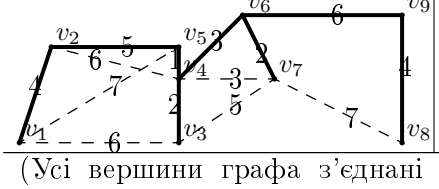
Очевидні способи такої модифікації потребу́ють $O(n)$ дій при кожному додаванні ребра́, сумарно — $O(n^2)$. Виявляється, не дуже складно проводити ці модифікації й так[45], щоб сумарна оцінка виявилася $O(n \log n)$. В такому разі, найдовшим етапом ($\Omega(m \log m)$) виявляється сортування ребер за неспаданням довжин. Це не очевидно (враховуючи, скільки місця у наведеному прикладі застосування алгоритму Краскала зайняло сортування відносно всього іншого), але це так.

[45] Як са́ме, в цьому посібнику не розглядається; охочі можуть пошукати в додаткових джерелах інформації з алгоритмів та структур даних, наприклад, структуру disjoint sets.

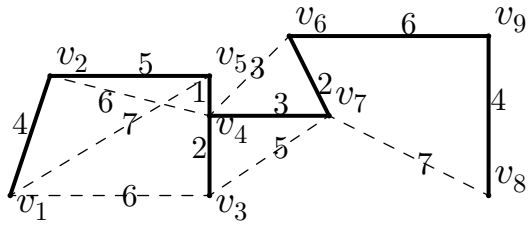
Приклад покрокового виконання алгоритму Краскала.

Якщо впорядкувати ре́бра цього зваженого графа за неспаданням довжин, вийде послідовність $\langle\{v_4, v_5\},\ 1\rangle$, $\langle\{v_3, v_4\},\ 2\rangle$, $\langle\{v_6, v_7\},\ 2\rangle$, $\langle\{v_4, v_6\},\ 3\rangle$, $\langle\{v_4, v_7\},\ 3\rangle$, $\langle\{v_1, v_2\},\ 4\rangle$, $\langle\{v_8, v_9\},\ 4\rangle$, $\langle\{v_2, v_5\},\ 5\rangle$, $\langle\{v_3, v_7\},\ 5\rangle$, $\langle\{v_1, v_3\},\ 6\rangle$, $\langle\{v_2, v_4\},\ 6\rangle$, $\langle\{v_6, v_9\},\ 6\rangle$, $\langle\{v_1, v_5\},\ 7\rangle$, $\langle\{v_7, v_8\},\ 7\rangle$.



граф	коментар
	Розглядаємо ребро $\langle\{v_4, v_5\}, 1\rangle$. Його кінці v_4 і v_5 належать різним компонентам зв'язності, тому додаємо це ребро.
	Розглядаємо ребро $\langle\{v_3, v_4\}, 2\rangle$. Його кінці v_3 і v_4 належать різним компонентам зв'язності, тому додаємо це ребро.
	Розглядаємо ребро $\langle\{v_6, v_7\}, 2\rangle$. Його кінці v_6 і v_7 належать різним компонентам зв'язності, тому додаємо це ребро.
	Розглядаємо ребро $\langle\{v_4, v_6\}, 3\rangle$. Його кінці v_4 і v_6 належать різним компонентам зв'язності, тому додаємо це ребро.
(без змін)	Розглядаємо ребро $\langle\{v_4, v_7\}, 3\rangle$. Його кінці v_4 та v_7 і так належать одній і тій самій компоненті зв'язності, тому це ребро пропускаємо.
	Розглядаємо ребро $\langle\{v_1, v_2\}, 4\rangle$. Його кінці v_1 і v_2 належать різним компонентам зв'язності, тому додаємо це ребро.
	Розглядаємо ребро $\langle\{v_8, v_9\}, 4\rangle$. Його кінці v_8 і v_9 належать різним компонентам зв'язності, тому додаємо це ребро.
	Розглядаємо ребро $\langle\{v_2, v_5\}, 5\rangle$. Його кінці v_2 і v_5 належать різним компонентам зв'язності, тому додаємо це ребро.
(без змін)	Розглядаємо ребро $\langle\{v_3, v_7\}, 5\rangle$. Його кінці v_3 та v_7 і так належать одній і тій самій компоненті зв'язності, тому це ребро пропускаємо.
(без змін)	І з ребром $\langle\{v_1, v_3\}, 6\rangle$ та само історія.
	Розглядаємо ребро $\langle\{v_6, v_9\}, 6\rangle$. Його кінці v_6 і v_9 належать різним компонентам зв'язності, тому додаємо це ребро.
(Усі вершини графа з'єднані в одну компоненту зв'язності, тому подальші ре́бра можна не розглядати)	

Для розглянутого графа можливе й інше ОДМВ, теж з сумарною довжиною ребер 27. Його теж можна було б отримати за алгоритмом Краскала, якби ре́бра $\{v_4, v_6\}$ та $\{v_4, v_7\}$ однакової довжи́ни 3 при сортуванні були розставлені в іншому порядку.



Алгоритм Пріма Алгоритм Пріма теж починає з графа, в якому нема жодного ребра́, і додає ре́бра одне за одним. Але в алгоритмі Пріма ре́бра додають так, щоб вже побудована частина постійно була деревом, тобто зв'язним графом. На початку це дерево складається з єдиної вершини (*якої-небудь*; ми візьмемо вершину v_1, але це не обов'язково), потім містить дві вершини й одне ребро, потім три вершини і два ребра, і так доки не будуть задіяні всі n вершин. При цьому щоразу додається найкоротше серед усіх ребер, один кінець яких належить дереву, а інший не належить.

Тобто, у загальному вигляді алгоритм можна записати так:

```
створити дерево, що складається з єдиної вершини v[1];
while (не всі вершини включені до де́рева) {
  знайти найкоротше серед ребер {v[i],v[j]}, таких, що
         v[i] належить дереву, а v[j] не належить;
  додати до де́рева вершину v[j] і ребро {v[i],v[j]};
}
```

Алгоритм Пріма має дещо спільне з алгоритмом Дейкстри. При бажанні вишукувати спільне й ігнорувати відмінне, можна навіть сказати, що єдина відмінність — що в алгоритмі Дейкстри вибирається мінімум серед оцінок відстаней від старту (*сум* довжин ребер), а в алгоритмі Пріма — мінімум серед довжин ре́бер (*окремих*). Що, звісно, не скасовує того факту, що постановки задач мінімальних шляхів і ОДМВ зовсім різні, й тому результати цих алгоритмів різні.

При найнаївнішій реалізації алгоритму Пріма (для вибору найкоротшого ребра́, яке з'єднує поточне дерево з новими вершинами, щоразу переглядати усі ре́бра графа) загальна кількість дій виявляється порядку $O(n^3)$ або $O(mn)$. Іншою крайністю є найефективніша для розріджених графів реалізація складності $O(m \log m)$, що вимагає засобів, аналогічних тим, які дозволяють досягти $O(m \log m)$ для алгоритму Дейкстри.

Часто «золотою серединою» (і не дуже складно писати, і не дуже довго виконується) є така організація алгоритму Пріма зі складністю $O(n^2)$. Будемо підтримувати три масиви, індекси яких відповідають вершинам графа: `in_tree` (типу `bool`), `d` (`double` або `int`, залежно які довжи́ни ребер) та `previous` (типу `int`). Значення `in_tree[i]` виражає умову «вершина $v[i]$ вже включена до де́рева». Значення `d[i]` (осмислене при `in_tree[i]==false`, тобто для вершин, ще не включених до де́рева) дорівнює довжині найкоротшого з ребер, які з'єднують цю вершину $v[i]$ з якою-небудь із вершин поточного де́рева; якщо таких ребер нема, `d[i]` $= \infty$. (Са́ме масив `d` і наведений у останньому стовпчику прикладу покрокового виконання алгоритму Пріма.) Значення `previous[i]` містить номер попередника, тобто вершини, через ребро з якої $v[i]$ була приєднана до де́рева (аналогічно відновленню шляху в BFS).

Отже, якщо є масив `d`, зрозуміло, як за один його перегляд вибирати, яку вершину слід додати до де́рева (звичайний пошук мінімального значення в `d`, враховуючи лише ті індекси, для яких `in_tree` хибне). Тепер розглянемо, як ефективно будувати і змінювати такий масив.

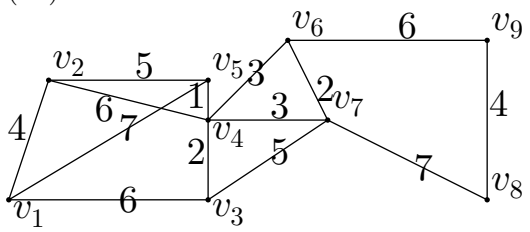
На першій ітерації основного циклу відбувається приєднання до де́рева однієї з вершин, сусідніх з $v[1]$.[46] Отже, значення масиву `d` для першої ітерації можна отримати, розглянувши *лише* ре́бра, що виходять з $v[1]$.

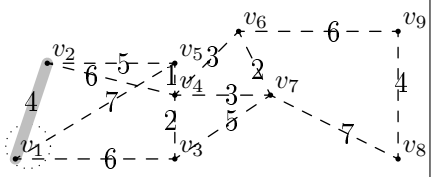
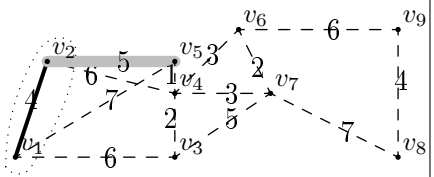
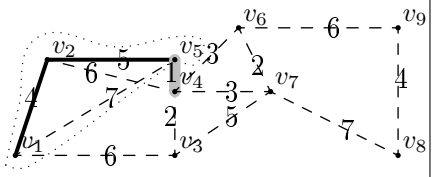
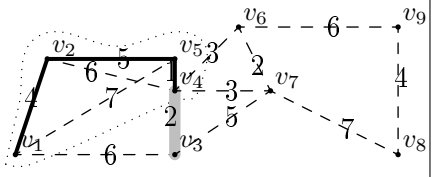
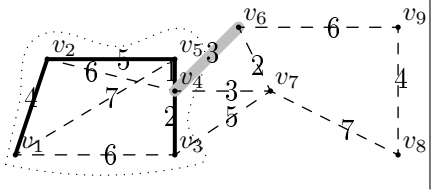
На дру́гій ітерації основного циклу в дереві вже дві вершини ($v[1]$ та вершина, приєднана на першому кроці; позначимо її як $v[i_1]$). Отже, на значення елементів `d` тепер впливають довжи́ни не лише ребер, що виходять з $v[1]$, а ще й ребер, що виходять з $v[i_1]$. Причому, масив `d` *вже* містить значення, пов'язані з $v[1]$; отже, потрібно передивитися лише ре́бра, що виходять з $v[i_1]$, і, якщо довжи́ни якихось із цих ре́бер менші за старі значення відповідних елементів `d`, замінити значення цих елементів `d`.

[46] точніше кажучи — з тією вершиною, з якої починають будувати дерево

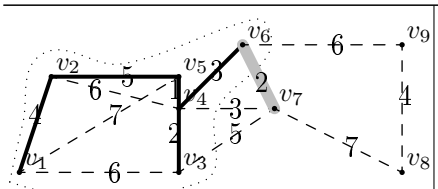
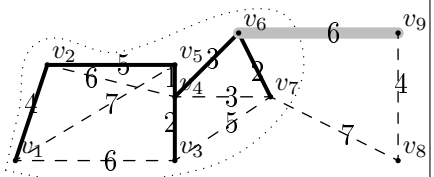
Так само буде й надалі: щоб отримати нове значення `d[k]` для вершини $v[k]$, достатньо вибрати мінімум зі старого значення `d[k]` і довжини́ ребра́ між цією вершиною $v[k]$ і вершиною $v[i_j]$, щойно доданою до де́рева. Таким чином, кожне оновлення значень усього масиву `d` потребує $O(n)$ операцій, а всього таких оновлень $n-1$. Сумарно, $O(n^2)$.

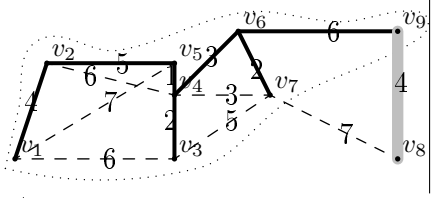
Приклад роботи алгоритму Пріма, для того ж графа, що й алгоритм Краскала.



граф	коментар	d
	Серед ребер, які з'єднують v_1 з іншими вершинами, найкоротшим є ребро $\langle\{v_1, v_2\}, 4\rangle$. Його і додаємо до де́рева.	1 – 2 **4** 3 6 4 ∞ 5 7 6 ∞ 7 ∞ 8 ∞ 9 ∞
	Серед ребер, які з'єднують поточне (побудоване на вершинах v_1 і v_2) дерево з іншими вершинами, найкоротшим є ребро $\langle\{v_2, v_5\}, 5\rangle$. Його і додаємо до де́рева.	1 – 2 – 3 6 4 6 5 **5** 6 ∞ 7 ∞ 8 ∞ 9 ∞
	Серед ребер, які з'єднують поточне (побудоване на вершинах v_1, v_2 і v_5) дерево з іншими вершинами, найкоротшим є ребро $\langle\{v_4, v_5\}, 1\rangle$. Його і додаємо до де́рева.	1 – 2 – 3 6 4 **1** 5 – 6 ∞ 7 ∞ 8 ∞ 9 ∞
	Серед ребер, які з'єднують поточне (побудоване на вершинах v_1, v_2, v_4 і v_5) дерево з іншими вершинами, найкоротшим є ребро $\langle\{v_4, v_3\}, 2\rangle$. Його і додаємо до де́рева.	1 – 2 – 3 **2** 4 – 5 – 6 3 7 3 8 ∞ 9 ∞
	Серед ребер, які з'єднують поточне (побудоване на вершинах v_1, v_2, v_3, v_4 і v_5) дерево з іншими вершинами, одним з найкоротших є ребро $\langle\{v_4, v_6\}, 3\rangle$. Його і додаємо до де́рева.	1 – 2 – 3 – 4 – 5 – 6 **3** 7 3 8 ∞ 9 ∞

(Замість ребра́ $\{v[4], v[6]\}$ на цьому кроці можна було вибрати і ребро $\{v[4], v[7]\}$, у результаті отримали б інше ОДМВ (див. також зауваження після прикладу виконання алгоритму Краскала).

	Серед ребер, які з'єднують поточне (побудоване на вершинах v_1, v_2, v_3, v_4, v_5 і v_6) дерево з іншими вершинами, найкоротшим є ребро $\langle\{v_6, v_7\}, 2\rangle$. Його і додаємо до де́рева.	1 – 2 – 3 – 4 – 5 – 6 – 7 **2** 8 ∞ 9 6
	Серед ребер, які з'єднують поточне (побудоване на вершинах v_1, v_2, v_3, v_4, v_5, v_6 і v_7) дерево з іншими вершинами (v_8 та v_9), найкоротшим є ребро $\langle\{v_6, v_9\}, 6\rangle$. Його і додаємо до де́рева.	1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 7 9 **6**

граф	коментар	d
	Серед ребер, які з'єднують поточне дерево з іншими вершинами (єдина вершина v_8), найкоротшим є ребро $\langle\{v_8, v_9\}, 4\rangle$. Його і додаємо до де́рева.	1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 4 9 –

(Усі вершини включені до де́рева, робота алгоритму завершена.)

Якщо зважений граф заданий тим варіантом матриці суміжності, де елемент `A[i][j]` містить довжину ребра́ $\{v_i, v_j\}$, відсутність ребра́ позначається нескінче́нністю, нумерація вершин починається з 0, то алгоритм Пріма можна реалізувати так.

```
in_tree[0] = true;
for(int i=1; i<n; i++) in_tree[i] = false;
for(int i=0; i<n; i++) {
  d[i] = A[0][i]; previous[i] = 1;
}
for(int k=1; k<n; n++) {/*одна вершина вже є, треба додати ще n-1*/
  min_d = INFTY; min_i = 0;
  for(int i=1; i<n; i++)/*шукаємо, яку вершину будемо додавати зараз*/
    if(!in_tree[i]  &&  (d[i] < min_d)) {
      min_d = d[i]; min_i = i;
    }
  add_edge(previous[min_i], min_i);/*додаємо ребро і вершину, див.далі*/
  in_tree[min_i] = true;
  for(int i=1; i<n; i++)/*поправляємо (якщо треба) масиви d і previous*/
    if(A[min_i][i] < d[i]) {
      d[i] = A[min_i][i];
      previous[i] = min_i;
    }
}
```

Вміст підпрограми `add_edge(int u, int v)` визначається виключно тим, у якому вигляді потрібно отримати шукане ОДМВ. Якщо, наприклад, треба вивести його на екран у вигляді переліку ребер, функція `add_edge` містить єдину дію `cout << u << " " << v << endl;`. Якщо, наприклад, потрібно отримати ОДМВ у вигляді списків суміжності, то дві дії `graph[u].push_back(v);` та `graph[v].push_back(u);`. І так далі.

Доведення правильності алгоритмів Краскала та Пріма І алгоритм Краскала, і алгоритм Пріма належать до т. зв. *жадібних* (рос. «*жадные*», англ. «*greedy*»): кожен з них щоразу додає до графа, який будує, якнайкоротше можливе ребро і надалі не переглядає (не змінює) цього вибору. Далеко не кожна задача може бути (правильно) розв'язана жадібним алгоритмом. Наприклад, якщо шукати найкоротший шлях між об'єктами у реальному місті, стратегія «намагатися постійно йти у потрібному напрямку» цілком може замість найкоротшого шляху завести у тупик.

Тим не менш, жадібні алгоритм Краскала й алгоритм Пріма гарантовано правильно розв'язують задачу побудови ОДМВ.

Сума довжин ребер остовного де́рева, побудованого за алгоритмом Краскала, мінімальна (не перевищує суми довжин ребер будь-якого іншого остовного де́рева того ж графа).

Доведення. Розглянемо будь-яке остовне дерево T_0, яке відрізняється від де́рева $T_{(K)}$, отриманого згідно алгоритму Краскала. Відсортуємо ре́бра T_0 так само, як при роботі алгоритму Краскала були відсортовані ре́бра всього заданого у вхідних даних графа,[47] і порівняємо послідовності ребер T_0 і $T_{(K)}$. *Перша* відмінність не може полягати в тому, що T_0 містить ребро, якого нема в $T_{(K)}$ (бо алгоритм Краскала пропускає *лише* ре́бра, які утворюють цикл, а T_0 за припущенням є деревом). Отже, перша відмінність — T_0 не містить деякого ребра́ $\{v[i], v[j]\}$, яке алгоритм Краскала включає до графа.

[47] Враховуючи не лише неспадання довжин, а й порядок ребер з однаковою довжиною; якщо цього не зробити, можливість переставляти ре́бра однакової довжини́ зробить деякі моменти доведення неправильними. Але мова йде лише про те, що однаковість порядків потрібна для доведення; ускладнювати додатковими правилами фактичну реалізацію сортування у алгоритмі Краскала не варто.

Тепер розглянемо граф G_1, що містить усі ре́бра T_0 плюс ребро $\{v[i], v[j]\}$. G_1 містить цикл, який включає в себе $\{v[i], v[j]\}$ (граф T_0 зв'язний, отже між $v[i]$ та $v[j]$ існує інший шлях). Цей цикл включає деяке інше ребро $\{v[k], v[l]\}$, не коротше за $\{v[i], v[j]\}$ (якби всі інші ре́бра циклу були строго коротші за $\{v[i], v[j]\}$, то всі вони розглядалися б алгоритмом Краскала раніше за ребро $\{v[i], v[j]\}$; тоді на момент аналізу $\{v[i], v[j]\}$ вершини $v[i]$ і $v[j]$ вже належали б одній компоненті зв'язності, а це не так).

Замінимо остовне дерево T_0 на граф T_1, отриманий з T_0 додаванням ребра́ $\{v[i], v[j]\}$ і вилученням ребра́ $\{v[k], v[l]\}$. T_1 теж є остовним деревом, бо за побудовою він містить $n-1$ ребро (T_0 — дерево, отже містить $n-1$ ребро; при побудові T_1 одне ребро додали й одне вилучили) і зв'язний (вилучення ребра́ $\{v[k], v[l]\}$ *з циклу* не порушує зв'язності).

Дерево T_1 «більш схоже» на $T_{(K)}$, ніж T_0: або множи́ни ребер T_1 і $T_{(K)}$ стали однакові, або перша (згідно все того ж сортування) відмінність між ними починається строго пізніше, ніж це було для T_0 і $T_{(K)}$ (ребро $\{v[i], v[j]\}$ більше не створює відміності, а ре́бра, які йдуть раніше за нього, не вставлялися і не виламувалися). І при цьому сума довжин ребер остовного де́рева T_1 не більша, ніж де́рева T_0.

В разі потреби, можна аналогічним чином перетворити T_1 у $T_2, T_3, \ldots$ — доки чергове дерево не стане рівним $T_{(K)}$. І при цьому сума довжин ребер кожного наступного де́рева не більша, ніж попереднього.

Значить, остовне дерево, побудоване за алгоритмом Краскала, справді мінімальне. ■

Доведення правильності алгоритму Пріма аналогічне: для довільного остовного де́рева T_0 знаходимо першу (згідно з порядком побудови де́рева $T_{(P)}$ за алгоритмом Пріма) відмінність T_0 від $T_{(P)}$, додаємо до T_0 відповідне ребро з $T_{(P)}$, показуємо (трохи іншими, але схожими міркуваннями), що в утвореному циклі можна знайти і вирізати ребро, не коротше доданого, і т. д.

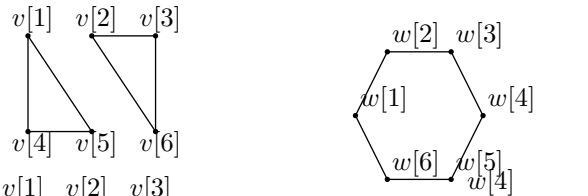
5.9 Завдання до розділу 5

1. Для вказаних пар графів, з'ясувати, чи є вони ізоморфними.

   В усіх пунктах, де відповідь «так, ізоморфні», навести табличку перенумерацій або малюнок з виконаною перенумерацією (досить щось одне); див. також п. (ж).

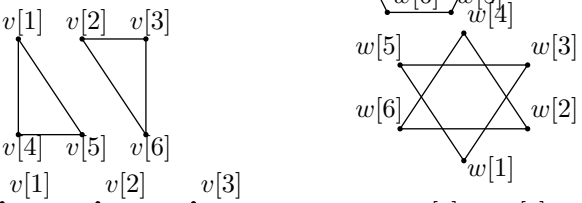
   В усіх пунктах, де відповідь «ні, не ізоморфні», навести детальні пояснення, *чому* графи не ізоморфні; див. також п. (а).

(а)

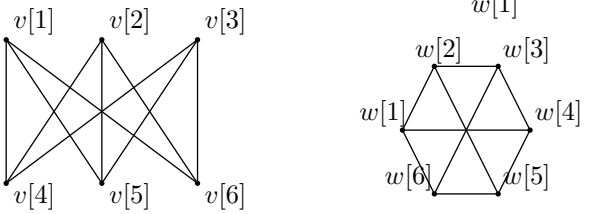


Не ізоморфні, правий зв'язний, лівий ні

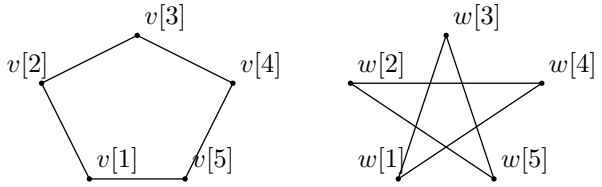
(б)



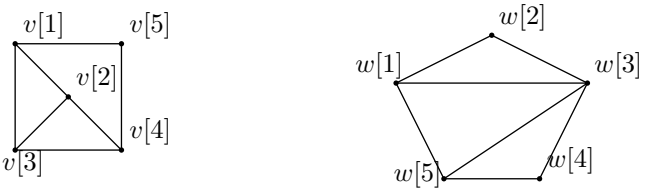
(в)



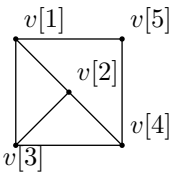
(г)

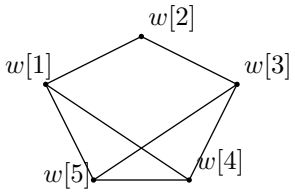


(д)

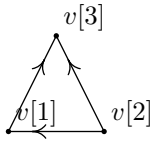


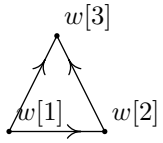
(е)





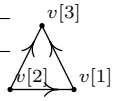
(ж)



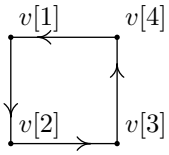


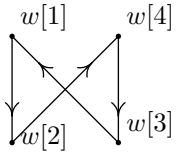
Так, ізоморфні

$v[1]$	$w[2]$
$v[2]$	$w[1]$
$v[3]$	$w[3]$

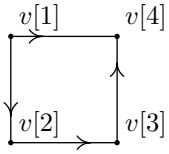


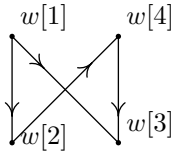
(и)



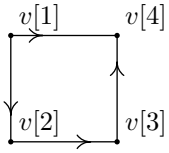


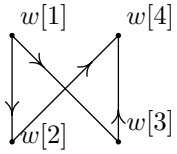
(к)



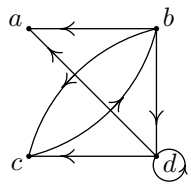


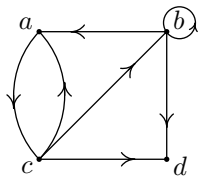
(л)





(м)





2. Побудувати простий граф з указаним переліком степенів вершин (або пояснити, чому такого простого графа не існує):

(а) 1, 2, 2, 2, 3, 3;
(б) 1, 1, 2, 2, 3, 3;
(в) 2, 3, 3, 4, 4, 4;
(г) 0, 0, 1, 3, 3, 3;
(д) 2, 2, 3, 4, 5, 6;
(е) 0, 2, 3, 3, 3, 3.

3. Побудувати орграф (обов'язково без кратних ребер, але пе́тлі дозволені) з указаним переліком пар напівстепенів виходу та напівстепенів захо́ду вершин (або пояснити, чому такого орграфа не існує):

(а)

i	$d_{out}(v_i)$	$d_{in}(v_i)$
1	1	0
2	1	1
3	1	2

(б)

i	$d_{out}(v_i)$	$d_{in}(v_i)$
1	0	1
2	1	1
3	1	2

(в)

i	$d_{out}(v_i)$	$d_{in}(v_i)$
1	0	2
2	2	1
3	2	1

(г)

i	$d_{out}(v_i)$	$d_{in}(v_i)$
1	0	1
2	2	1
3	2	2

4. (а) Простий граф заданий списками суміжності:

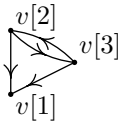
1 : 2
2 : 1 3 5
3 : 2 5
4 : ∅
5 : 2 3

Намалювати діаграму графа, записати матрицю суміжності та матрицю інциденцій.

(б) Орграф заданий матрицею інциденцій

$$\begin{matrix} 2 & -1 & 0 & +1 & 0 & 0 \\ 0 & +1 & -1 & 0 & +1 & 0 \\ 0 & 0 & +1 & -1 & -1 & +1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{matrix}$$

Намалювати діаграму графа, записати матрицю суміжності та списки суміжності.

5. Для орграфа 

вказати:
   (а) усі можливі *ланцюги*, що ведуть з $v[3]$ у $v[1]$;
   (б) усі можливі *прості ланцюги*, що ведуть з $v[3]$ у $v[1]$;
   (в) усі *маршрути* довжиной $\leqslant 5$, що ведуть з $v[3]$ у $v[1]$.

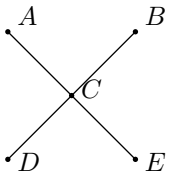
   Відповіді (та причини, чому відповіді різних пунктів різні) пояснити.

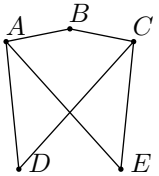
6. Намалювати діаграму графа K_4, в якій ребра зображаються відрізками прямих і не перетинаються.

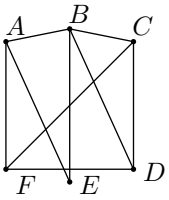
7. Скільки ребер у повному простому графі K_n? (Відповідь пояснити.)

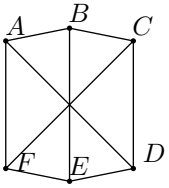
8. Скільки вершин та скільки ребер у повному дводольному графі $K_{p,q}$? (Відповідь пояснити.)

9. Для вказаних графів, з'ясувати, чи є вони дводольними.

(а) 

(б) 

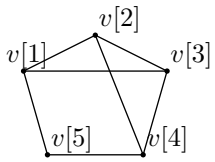
(в) 

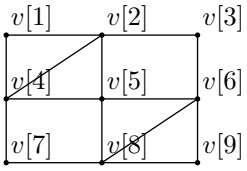
(г) 

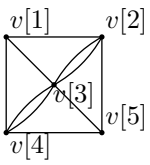
10. Написати підпрограму, яка за матрицею суміжності незваженого неорієнтованого графа знаходить степені його вершин.

11. Аналогічно попередньому завданню, але для незваженого орграфа (які деталі все-таки істотно відрязняються, з'ясувати самостійно й описати у словесних поясненнях до написаної підпрограми).

12. Дослідити графи на наявність: ейлерових циклів та нециклічних ейлерових маршрутів; гамільтонових циклів та нециклічних гамільтонових маршрутів. Якщо є — навести приклад маршруту; якщо нема — *пояснити*, *чому* нема. Пояснення — *суттєва* складова завдання.

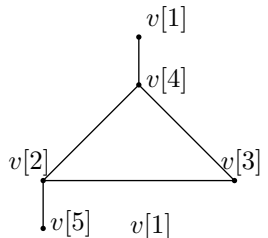
(а) 

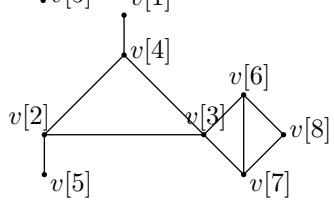
(б) 

(в) 

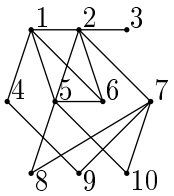
(г)
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(д)
$$\begin{array}{ll} 1: & 2,\,5 \\ 2: & 1,\,5 \\ 3: & 4,\,6 \\ 4: & 3,\,6 \\ 5: & 1,\,2 \\ 6: & 3,\,4 \end{array}$$

(е) 

(ж) 

13. Якщо даний конкретний, відомий в усіх деталях, неорієнтований мультиграф з петлями, як все-таки скористатися теоремою Дірака? (Не зважаючи на те, що її сформульовано лише для простих графів.)

14. Даний граф 

(а) Розписати детально застосування BFS, починаючи з вершини 1.

(б) Аналогічно від вершини 2.

(в) Знайти *всі* різні шляхи однакової мінімальної довжини́ від вершини 2. Відповідь аргументувати.
(Беручи за старт завжди вершину 2, за фініш послідовно кожну з решти вершин, познаходити всі різні шляхи однакової мінімальної довжини́ між одним і тим самим стартом та одним і тим самим фінішем.)

(г) Як, виконуючи пункти по порядку, легко і просто знайти відстані до всіх вершин від вершини 3 цього графа?

(д) Користуючись BFS (не розписуючи детально), знайти відстані до всіх вершин від вершин 4, 5 і 6.

(е) Як, виконуючи пункти по порядку, легко й просто знайти відстані до всіх вершин від вершин 7, 8, 9 і 10 цього графа?

(ж) Знайти діаметр, радіус та центр цього графа.

15. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). «Лабіринт» задано прямокутником, в якому кожна клітинка містить або "`.`" (прохід), або "`*`" (стіна). Знайти (обов'язково користуючись пошуком у ширину) найкоротший маршрут від клітинки, позначеної "`S`", до будь-якого з виходів (виходами називаємо усі вільні клітинки у зовнішніх рядках/стовпчиках). Вказати якийсь один конкретний маршрут (як *послідовність* клітинок). Дослідити питання єдиності мінімального маршруту (чи є різні маршрути однакової мінімальної довжини́, чи маршрут мінімальної довжини́ лише один).

(1)
```
************
********.***
****.*..S***
***....**...
*..*.**...**
**.*....*..*
**.*.**...*.
**.*....*...
**.**......*
******.*.***
******.*****
************
```

(2)
```
************
********.***
***...**.***
***.*.*..***
***.*.....**
......***.**
***.**....**
*.*.....*..**
*.*.****.*.*
*.*.*....*..
*.******.S**
************
```

(3)
```
************
************
*******.S***
*......*..**
**.****.*.**
..*.....*.**
****.*....**
**.....*.***
**.***.*...*
**........**
**..**.*****
**.*********
```

(4)
```
************
*******.****
*******S****
***..**..***
*....***..**
*****...*..*
*..........*
**.*.*.*.*.*
**.*.*...*.*
**.*..*...**
**....******
***.********
```

(5)
```
************
*******.****
***...**.***
**.*...*.***
**S..*.*..**
*****..*.***
*.**..*.****
*........****
**..**.*....
****.*.*..**
**.....*.***
*****.**.***
```

(6)
```
************
*.*.**.*****
**.S.*..****
*..***..****
**........***
*..**.**.***
**.........**
**.**..*..**
***..*.*.*.*
***.*..*....
********.***
************
```

(7)
```
************
*.**********
*S****...***
*.....*..***
***.*..**.**
**..*.*...**
**..*.*.*.**
***...*..***
***.*..*..**
****.*.*.***
*........***
***.********
```

(8)
```
************
******.*****
****...***.*
***.*...*..*
***...*.*..*
**.*.*...*.*
**S...**...*
**.***..*..*
***...*.*...
***..*.....*
************
************
```

(9)
```
************
********.***
*****....***
***..*.**.**
*...*.*...**
**..*...*.**
**.*..*.*S**
*........*..*
**.****...**
**.....*****
*****..*****
*****..*****
```

(10)
```
************
**..*.******
***.***.****
***.......**
*...**...***
*.S...*.****
****.*...***
*...*..*.***
**.*...*****
***..*..****
****.*....**
****..******
************
```

(11)
```
************
*******.****
****S....***
****.**.****
**..*...****
*..*...*..**
***..*...***
*.....*...**
**..*...****
**..*.*.****
**.....*****
***.**.*****
************
```

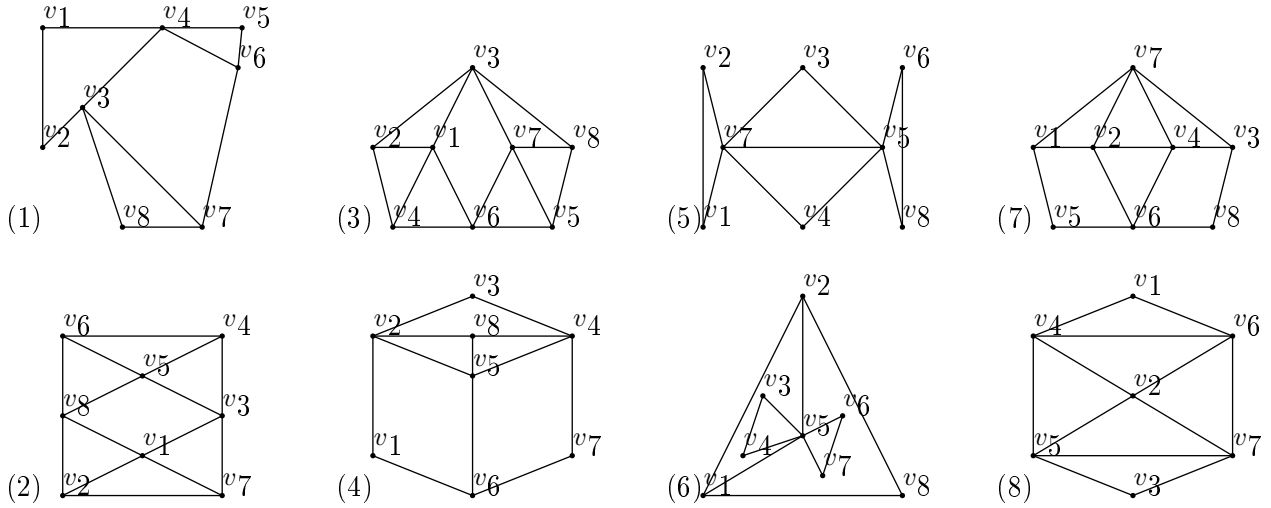
(12)
```
************
*****..*****
****.*...***
***....*...*
***.**..**S*
**....*..*.*
***.**..*.**
***.....*.**
*******...**
***....**...
*****..**...
************
```

(13)
```
************
********.*.*
***.S*...*.*
**.*....*..*
**..***..*.*
***......***
****.*.*.***
**....*..***
***.*..*.***
***..*....**
***..**.****
****.********
```

(14)
```
************
***..*.*****
**..*..*****
*.**.....***
*S...**..***
***.*.*.**.*
*..........*
**...*..*.**
********...*
*******..*.*
*******..*.*
********.***
```

(15)
```
*******..***
********.***
*****....***
******.**.**
*****..*...*
****....****
*.***.*....*
**.....*.*.*
*....**..*.*
*.***...*..*
*****.*...S*
************
```

16. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Для простого графа



(а) Провести BFS з вершини $v[6]$, розписавши детально.

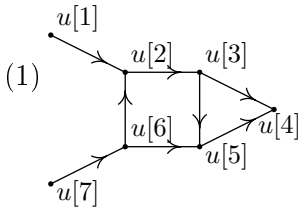
(б) Для кожної з решти вершин, навести дерево найкоротших маршрутів з цієї вершини (повиділявши на діаграмі рéбра, по яким фактично відбувся перехід під час BFS).

(в) Записати матрицю відстаней; знайти діаметр, радіус та центр.

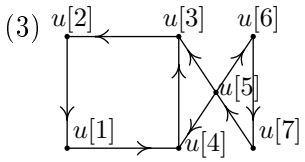
17. Знайти значення діаметра та радіуса будь-якого повного простого графа (як формулу від n). Відповідь аргументувати.

18. Навести який-небудь не повний простий граф, у якому $D(G) = R(G)$.

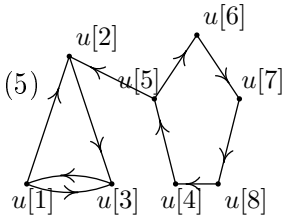
19. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Для вказаного орієнтованого незваженого графа



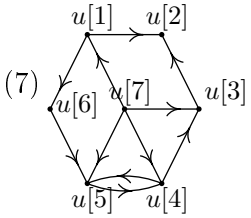
пошук починати від вершини $u[7]$;



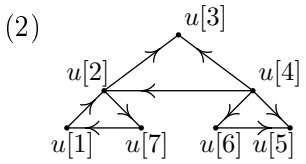
пошук починати від вершини $u[5]$;



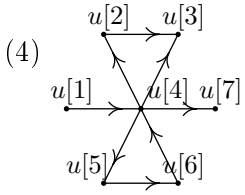
пошук починати від вершини $u[7]$;



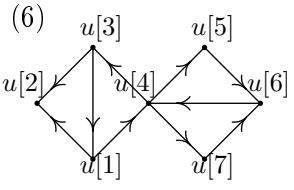
пошук починати від вершини $u[1]$;



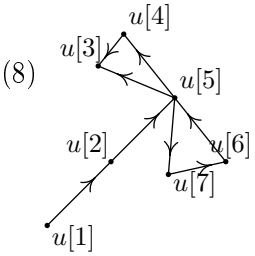
пошук починати від вершини $u[4]$;



пошук починати від вершини $u[1]$;



пошук починати від вершини $u[7]$;



пошук починати від вершини $u[2]$.

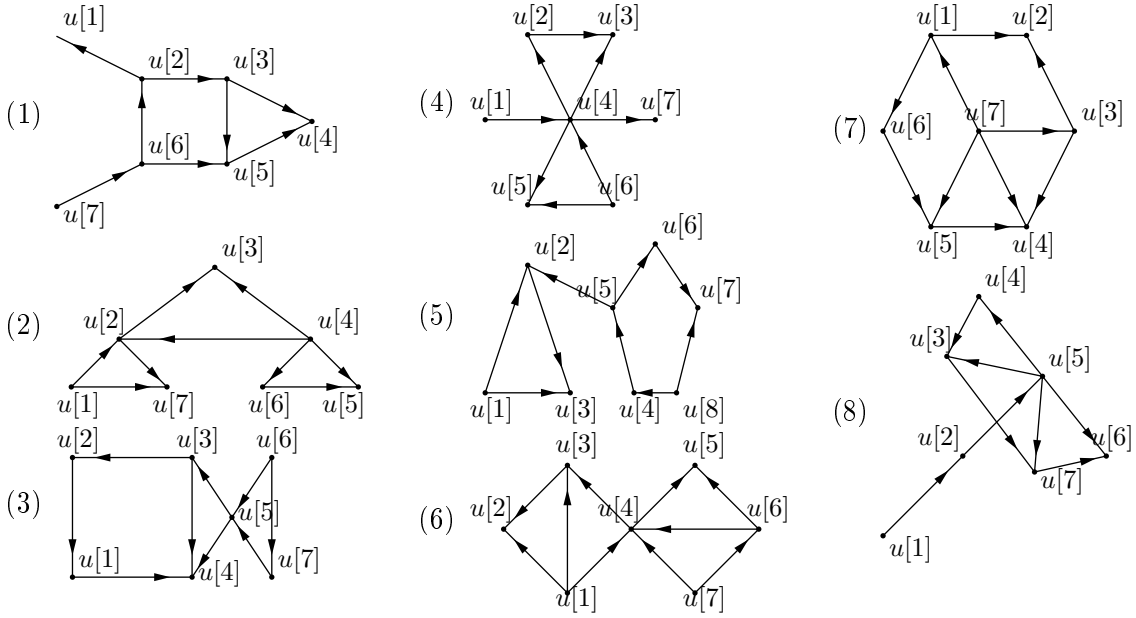
(а) Провести DFS, починаючи від указаної у варіанті вершини.

(б) Записати матрицю досяжності цього орграфа.

(в) Встановити (аргументовано) тип орієнтованої зв'язності цього орграфа.

(г) Побудувати граф конденсації цього орграфа.

20. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Провести топологічне сортування, причому двічі: один раз, скрізь перебираючи вершини у порядку зростання номерів ($v[1]$, $v[2]$, …, $v[n]$), інший раз — у порядку спадання ($v[n]$, $v[n-1]$, …, $v[2]$, $v[1]$).

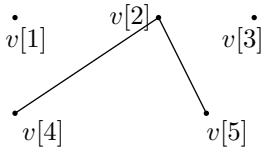


21. Вважаючи, що матриця досяжності орграфа вже побудована у двовимірному масиві `R`, написати підпрограму, яка видавала б одну з трьох відповідей: «орграф сильно зв'язний», «орграф односторонньо зв'язний (але не сильно зв'язний)», «орграф не є односторонньо зв'язним» (розрізняти, чи він слабко зв'язний, чи не зв'язний, не треба).

22. Написати програму, яка читатиме кількість вершин простого графа N, потім його матрицю суміжності, і виводитиме, якій компоненті зв'язності яка вершина належить (нумерація компонент довільна, аби всередині однієї й тієї ж компоненти вершини отримували однакові позначки, а у різних компонентах різні).

Приклад:

Вхідні дані	Результат	Графічне зображення

`5`
`0 0 0 0 0`
`0 0 0 1 1`
`0 0 0 0 0`
`0 1 0 0 0`
`0 1 0 0 0`
	`v1 -- компонента 1`
	`v2 -- компонента 2`
	`v3 -- компонента 3`
	`v4 -- компонента 2`
	`v5 -- компонента 2`
		

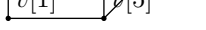
(Графічне зображення дається лише для кращого розуміння прикладу, програма його малювати не повинна).

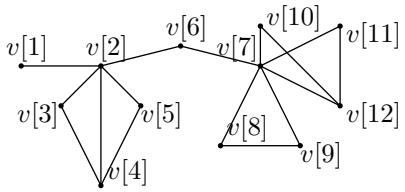
Вказівка: виділення однієї компоненти зв'язності повинно відбуватися одним первинним (а не рекурсивним) викликом `dfs`. Тобто, викликом або з головної програми, або з іншої (нерекурсивної) функції.

23. Описати (словами та малюночком), як відбуватиметься DFS, якщо його застосувати до повного простого графа. А якщо застосувати BFS?

24. Намалювати діаграми всіх можливих не ізморфних між собою дерев з п'ятьма вершинами. Аргументувати, що цей перелік вичерпний (що нема ніяких інших).

25. Скільки ребер у лісі, що містить n вершин та k компонент зв'язності? Чому?

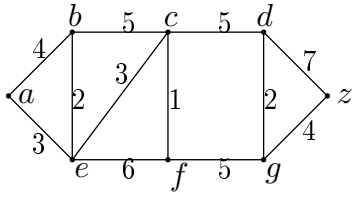
26. Вказати всі можливі остовні дерéва для графа  . Аргументувати, що цей перелік є вичерпним (нема ніяких інших остовних дерев).



27. Яка загальна кількість остовних дерев такого графа?

28. Побудувати ОДМВ для графа з рисунку
    (а) алгоритмом Краскала;
    (б) алгоритмом Пріма.

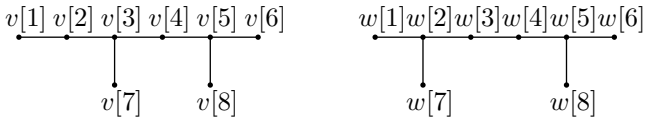
    Вказати усі можливі ОДМВ цього графа.



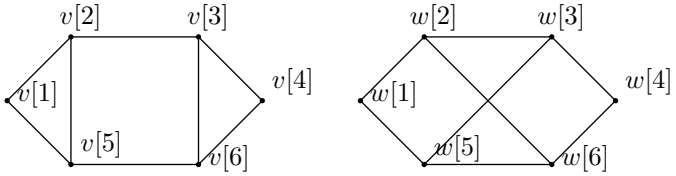
Додаткові завдання підвищеного рівня складності

1*. Для вказаних пар графів, з'ясувати, чи є вони ізоморфними. Вимоги щодо оформлення див. у завд. 1 (стор. 152).

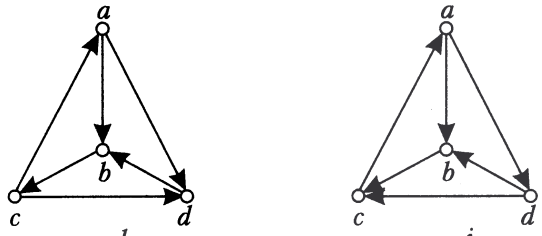
(н)



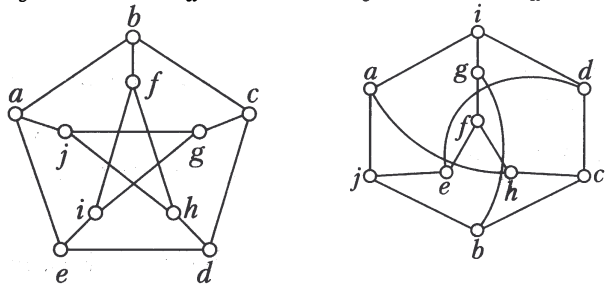
(п)



(р)



(с)



2*. `informatics.mccme.ru/moodle/mod/statements/view.php?id=359` (група з 22 задач).

3*. Змагання «5 — Тренування, що стартувало 18.01.2013» сайту `https://ejudge.ckipo.edu.ua`

4*. Написати підпрограму, яка перетворюватиме матрицю інциденцій незваженого не мульти графа у матрицю суміжності (цього ж графа). Підпрограма має перевіряти, чи справді заданий масив є правильною матрицею інциденцій (якщо ні — видавати повідомлення про конкретне розміщення конкретної помилки).
    (а) Додатково відомо, що граф до того ж неорієнтований і без петель.
    (б) Підпрограма автоматично визначає, чи орієнтований граф і чи містить пе́тлі. Дотримуватися класичної точки зору, що граф або орієнтований, або ні (хоча реальна мережа автодоріг і являє собою приклад порушення цієї вимоги, бо *лише деякі* фрагменти вулиць односторонні).

5*. Написати функцію, яка за заданими матрицею суміжності орграфа (не мульти, не зваженого, пе́тлі можуть бути, може й не бути) та послідовністю вершин визначатиме
   (а) чи задає вказана послідовність маршрут; результатом повинно бути «так»/«ні»;
   (б) чи задає вказана послідовність простий ланцюг; результатом повинно бути одне з трьох: «так, є простим ланцюгом», або «послідовність є маршрутом, але не є простим ланцюгом», або «послідовність не є маршрутом».

   (*Підказка.* У пункті (b) функція повинна використовувати додатковий масив; у пункті (а) він не потрібен.)

6*. Написати програму перевірки ізоморфності двох простих графів, заданих матрицями суміжності, методом перебору можливих бієкцій (способів перенумерації).
   (а) Завжди перебираючи абсолютно всі $n!$ способів.
   Підказка. Цей пункт, на відміну від наступного, насправді досить простий. Дослідивши засоби бібліотеки `algorithm` (STL мови C++), можна знайти функцію, на яку можна перекласти значну частину робо́ти.
   (б) Скорочуючи перебір за рахунок врахування степенів вершин.

7*. Довести формулу (73) зі стор. 121.

8*. У розд. 5.3.2 на стор. 122 описано спосіб знаходження матриці відстаней незваженого графа за його матрицею суміжності, який часто згадується у підручниках для «чистих» математиків, але рідко доцільний з точки зору програміста.

   Запропонувати додаткову умову для обривання обчислення степенів матриці суміжності й записування у не заповнені клітинки "∞". Таку, щоб якщо са́ме цей спосіб чомусь все ж реалізований програмно, то можна було б дописати `if(...) break;`, і це (будучи гарантовано правильним) не завжди, але нерідко прискорювало виконання.

9*. Довести, що (i,j)-ий елемент k-го степеню матриці суміжності дорівнює *кількості* всіх можливих маршрутів з $v[i]$ до $v[j]$.

   Примітка. «Всі можливі» включають в т. ч. маршрути з багатократними проходженнями вершин/ребер; універсальних простих формул для кількостей ланцюгів або простих ланцюгів наука не знає.

10*. Аргументувати згадане в попередньому завданні 9* «наука не знає», пов'язавши це твердження із гамільтоновими циклами.

11*. Довести нерівність $R(G) \leqslant D(G) \leqslant 2R(G)$ (див. стор. 123).

12*. `www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=134`, тобто задача «Conflict»; повний розв'язок передбачає і зарахування програми сайтом, і написання мікрореферату та його захист.

13*. `www.olymp.vinnica.ua/index_ua.php?lng=ua&cid=1366`, тобто задача «`Dyvaknet`». Суть — реалізувати описаний на стор. 132 варіант відновлення шляхів, що знаходить *усі* найкоротші шляхи; повний розв'язок передбачає і зарахування програми сайтом, і написання мікрореферату та його захист.

14*. Пред'явіть два різні прості графи зі степенями вершин 6, 3, 3, 3, 3, 3, 3, такі, щоб один з них містив гамільтонів цикл, а інший ні.

15*. Перетворити описану в розд. 5.5.1 схему доведення теореми Ейлера до формально строгого вигляду; на основі цього побудувати також строге доведення критерію існування ейлерового циклу в орграфі. Оформити все це як реферат та захистити усно.

16*. Припустимо, ніби попереднє завдання виконане; як тоді строго довести наведені в розд. 5.5.1 критерії існування нециклічних ейлерових шляхів?

17*. Довести: «*Якщо (як того потребу́є аналог теореми Ейлера для орграфів) для кожної вершини виконується умова рівності напівстепенів виходу і захо́ду, то перевірку сильної зв'язності можна реалізувати значно простішим чином, ніж згадано наприкінці розд. 5.4.2: достатньо узяти будь-яку одну вершину, і перевірити, чи усі вершини досяжні з неї.*». Також показати, що якщо умова рівності напівстепенів виходу і захо́ду не виконується, то перевіряти таким чином сильну зв'язність *не* завжди правильно.

18*. Реалізувати програму, яка перевірятиме *факт існування* (з відповіддю «так»/«ні») ейлерового шляху в неорієнтованому графі (який може містити кратні ре́бра та пе́тлі, а може й не містити). Граф подавати матрицею суміжності (вводити готову).

19*. Реалізувати програму, яка не лише перевірятиме факт існування ейлерового шляху в неорієнтованому графі, але також і знаходитиме сам шлях як послідовність вершин. Вибір способу подання графа — складова завдання. Рівень складності дуже істотно залежить від того, наскільки ефективну програму писати. Дуже приблизно, виконання на недорогому сучасному ПК вкладається у 1 сек при розмірах графа:

$n\leqslant 10^2$, $m\leqslant 10^3$	$n\leqslant 10^3$, $m\leqslant 10^4$	$n\leqslant 10^4$, $m\leqslant 10^5$	$n\leqslant 10^5$, $m\leqslant 10^6$
неважко	середньо	складно	дуже складно

20*. Довести, що при $n\geqslant 3$ для існування гамільтонового циклу необхідна (але не достатня) вершинна двозв'язність.

21*. Як неодноразово наголошено, коли у зваженому графі є цикл від'ємної довжи́ни, задача знаходження найкоротших шляхів втрачає смисл.

Здавалося б, можна не оголошувати про втрату смислу, а переформулювати задачу: шукати маршрут мінімальної довжи́ни лише серед несамоперетинних (наприклад, простих ланцюгів). Це дозволило б уникнути прямування до $(-\infty)$ за рахунок накручування від'ємних циклів. А для графів, що не містять від'ємних циклів, нічого не змінилося б, бо для них самоперетини все одно невигідні.

Виявляється, так зазвичай не роблять тому́, що хоч поставити таку модифіковану задачу й можна, але неясно, що з нею робити далі: за сучасними уявленнями науки, для розв'язання такої задачі, мабуть, не існує ефективного алгоритму.

Показати це.

Вказівка. Слід спертися на те, що за сучасними уявленнями, мабуть, не існує ефективного алгоритму пошуку гамільтонових шляхів. В чому полягає зв'язок між задачею про мінімальні несамоперетинні шляхи та гамільтоновими шляхами, і як його тут використати — слід придумати самостійно, або знайти десь за межами цього посібника.

22*. Розглянемо таку задачу на шляхи в лабіринті. Нехай потрібно знайти абсолютно *всі несамоперетинні* шляхи (наприклад, у зображеному лабіринті — чотири штуки, включно з явно кружним «піднятися праворуч, спуститися посередині й піднятися ліворуч»). Як модифікувати DFS для розв'язання такої задачі? Чому така модифікація вже не може називатися пошуком у глибину?

```
**.*****
*......*
*.*.**.*
*...**.*
***..X.*
********
```

23*. У завданні 21 (стор. 157) говорилося «розрізняти, чи він слабко зв'язний, чи не зв'язний, не треба». Розрізнити це за матрицею досяжності справді важче, ніж інші види орієнтованої зв'язності. Запропонувати спосіб, як це все-таки зробити. Реалізовувати не обов'язково, досить пояснити алгоритм словесно.

24*. Розглянемо дві (помилкові!) пропозиції, як, нібито, можна (насправді ні) зробити, щоб алгоритм Дейкстри працював на графах з від'ємними довжинами ребер.

(А) Перед тим, як застосувати алгоритм Дейкстри, додамо до довжи́ни кожного ребра́ графа одне й те саме число (достатньо велике, щоб *усі* довжи́ни стали невід'ємними). Застосуємо алгоритм Дейкстри до отриманого графа з невід'ємними довжинами ребер, а тоді повиправляємо довжи́ни ребер знайденого найкоротшого маршруту.

(Б) Перевіряти, чи менша довжина нового маршруту `d[curr]`+l(`curr`, `i`) за довжину відомого маршруту `d[i]`, яким би не був статус $v[i]$. Якщо оцінка справді зменшується, то переводити $v[i]$ до статусу 1, навіть якщо вона вже перебувала у статусі 2.

Детально описати недоліки цих пропозицій (кожної окремо; можна лише однієї).

25*. Розглянемо нестандартну графову модель, де вершини позначають населені пункти, зважені ре́бра — дороги між ними, але вага ребра́ є не довжиною дороги, а обмеженням на масу вантажівок, які можуть там їздити (наприклад, по ребру, позначеному «9», можуть їздити вантажівки масою від 0 до 9 включно, а масою 9,0000000001 уже не можуть).

Як треба модифікувати алгоритм Дейкстри, щоб розв'язувати задачу «*вантажівка якої максимальної маси може доїхати від вершини $v[i]$ до вершини $v[j]$?*»?

26*. Реалізувати алгоритм Краскала. Достатньо зі складністю $O(n^2)$, тобто перевірку, чи одній компоненті зв'язності належать вершини, можна робити через очевидний варіант описаної на стор. 147 системи «кольорів». Повинні бути і рукописний текст з коментарями, і зарахування задачі №1377 на `https://informatics.mccme.ru`.

27*. *«Заданий простий граф; з'ясувати, чи містить він хоча б один цикл;*

(а) потрібна лише відповідь "так/ні";

(б) потрібен сам цикл як послідовність вершин.»

Детально сформулювати словами алгоритми розв'язання обох версій цієї задачі, причому алгоритм для (а) повинен бути значно простішим, ніж для (b).

28*. Розглянемо задачу: *«Вхід у скарбницю закрито прямокутною кам'яною плитою, яка від старості покрита тріщинами. Кожна тріщина має форму прямолінійного відрізку. Тріщини не можуть ні торкатися одна до одної чи до меж плити, ні перетинатися. Щоб увійти до підземелля, плиту треба розламати уздовж деякої ламаної, що з'єднує* ПРОТИЛЕЖНІ *сто́рони плити́. Ця ламана може включати в себе деякі (можливо, всі) вже наявні тріщини та додаткові, які шукачі скарбів мають пробити власноруч. Пробивати додаткові тріщини можна де завгодно. Кожну наявну тріщину можна не використовувати зовсім, можна використати по всій довжині, можна використати частково. Знайти такий спосіб розламати плиту, який вимагатиме пробивання якомога меншої (за сумарною довжиною) сукупності додаткових тріщин».* (Координати вершин плити, кількість наявних тріщин та координати їхніх кінців є вхідними даними задачі; система координат введена так, щоб сто́рони плити́ були паралельні вісям.)

Пояснити, де тут з'являється граф, що є вершинами графа, що ребрами; яка тема з посібника має стосунок до цієї задачі, та який; детально описати словами алгоритм розв'язання. Навести який-небудь приклад рисунку плити з тріщинами та відповідний їй граф.

29*. Розглянемо задачу: *«Слова́ складаються з маленьких букв латинського алфавіту; довжина кожного окремо взятого сло́ва не перевищує 50. Чи можна з заданного (як вхідні дані) набору слів скласти чайнворд (перша буква кожного наступного сло́ва однакова з останньою буквою попереднього)? Треба використати всі слова́ по одному разу.»* Вхідними даними є набір слів; результатом — відповідь «так»/«ні».

Пояснити, де тут з'являється граф, що є вершинами графа, що ребрами; яка тема з посібника має стосунок до цієї задачі, та який; детально описати словами алгоритм розв'язання.

30*. «Три задачі про доміно». Їх можна виконувати в будь-якому порядку, в т. ч. лише частину з них. Геометрією нехтувати: якщо «можна, але криво і з не щільними приляганнями» — значить, можна. Ці задачі *мають* близький стосунок до однієї з розглянутих у посібнику тем. Який стосунок і до якої теми — слід придумати самостійно.

(а) Чи можна скласти цикл з сукупності дощечок стандартного доміно, беручи в точності увесь набір, нічого не викидаючи і не додаючи? Дощечки слід притуляти за правилами, тобто однаковими номерами. Достатньо дати лише відповідь «так»/«ні», але обов'язково її довести.

(б) Аналогічно попередньому пункту, але для зменшеного доміно: на половинках дощечок можна писати чи́сла не від 0 до 6, а: (1) від 0 до 2; (2) від 0 до 3. На відміну від попереднього пункту, в разі позитивної відповіді, навести сам цикл з дощечок.

(в) Розглянемо ще більш узагальнене доміно: у вхідних даних задаються не тільки N (максимальне значення на половинках дощечок), а ще й вказується, яка саме пара чисел написана на кожній дощечці набору. Наприклад, може бути набір, у якому $N = 3$ і є лише дощечки $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,1\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$.

Детально сформулювати словами алгоритм розв'язання задачі *«Яку* мінімальну *кількість ланцюжків можна викласти з усіх дощечок заданого (у вхідних даних) набору?»* Наприклад, вищезгаданий набір неможливо викласти в один ланцюжок, але можна викласти у два: $\langle\ \{0,1\}, \{1,1\}, \{1,2\}, \{2,3\}\ \rangle$ та $\langle\ \{1,3\}, \{3,0\}, \{0,2\}\ \rangle$. Отже, для цього набору результатом є число 2.

6 Мови, регулярні вирази та автомати

6.1 Неформальний вступ до формальних мов

Формальна мова (рос. «*формальный язык*», англ. «*formal language*») — це множина слів. *Словом* (рос. «*слово*», англ. «*string*») вважається будь-яка послідовність символів. *Символами* (вони ж *літери*, *букви*; рос. «*символы*», «*буквы*»; англ. «*symbols*», «*characters*», значно рідше «*letters*») можуть бути будь-які відрізнювані об'єкти.

Як правило, ми будемо розглядати в якості символів латинські букви та/або десяткові цифри, але це лише заради зручності; в принципі можливе й слово "⇑⊥ɾ'ξ7ўz♠†", а на стор. 170 наведено приклад символів та алфавіту, ще менш схожих на традиційний смисл цих слів.

Як завжди, послідовність передбачає впорядкованість (*cat* та *act* — різні слова́), а множина не передбачає (мова $\{hello, 42\}$ дорівнює мові $\{42, hello\}$).

Довжина́ сло́ва (рос. «*длина́* сло́ва», англ. «string's *length*») — це кількість символів у ньому. Наприклад, довжина сло́ва "*aaabacaba*" дорівнює 9.

Приклад 1. Формальна мова цілих чисел L_{int} — множина всіх можливих послідовностей символів "`-`", "`+`", "`0`", "`1`", "`2`", "`3`", "`4`", "`5`", "`6`", "`7`", "`8`", "`9`", для яких виконуються правила: "`-`" та "`+`" можуть зустрічатися лише у першій позиції сло́ва; "`0`" не може бути першим серед цифр послідовності (але коли він — єдиний символ послідовності, то може).

Приклад 2. Формальна мова ідентифікаторів L_{iden} — множина всіх можливих послідовностей латинських букв, десяткових цифр, знаків "`_`" (підкреслень), які починаються не з цифри.

Аналогічно можна ввести формальну мову чисел з рухомою комою (floating point), формальну мову `string`-ових констант (де "`string`" — тип мови програмування), формальну мову арифметичних виразів, формальну мову ДНФ (де диз'юнкції та заперечення є символами, а змінні можуть бути символами, а можуть і складатися з кількох символів, як-то букви і числового індекса). Можна розглянути і формальну мову ДДНФ, яка є підмножиною мови ДНФ. І так далі.

Задача перевірки належності сло́ва мові полягає у тому, що задана формальна мова, задане слово, і потрібно визначити, чи «це слово належить цій мові», чи «... *не* належить ...».

Порожнє слово *Порожнє слово* (рос. «*пустое слово*», англ. «*empty string*») — це слово, що не містить жодного символу (інакше кажучи, довжина́ 0). Ми будемо умовно позначати його як "ε" (у деяких джерелах порожнє слово позначають як "Λ", або "λ", або "e").

Порожнє слово *не* є пропуском (пробілом): пробіл — символ, а три пробіли — три символи; водночас, три порожні слова підряд дорівнюють одному порожньому слову й не містять жодного символу.

Порожнє слово реально використовується у програмуванні: `string s=""` (між лапками *нема* пробіла) ініціалізує `s` порожнім словом.

6.2 Регулярні мови та регулярні вирази (regex-и)

Регулярні мови (рос. «*регулярные языки*», англ. «*regular languages*»; іноді скорочують як «рег. мови», «рег. языки», «reg. lang.-s») є одним з важливих часткових випадків формальних мов. Щоб дати означення регулярної мови, спочатку треба дати означення деяких операцій над мовами.

Об'єднання *Об'єднання* (рос. «*объединение*», англ. «*union*») мов A та B — множина всіх слів, що належать хоча б одній з мов A або B:

$$A \cup B \stackrel{\text{def}}{=} \{p \mid p \in A \vee p \in B\}. \tag{74}$$

(Це нічим не відрізняється від об'єднання, введеного формулою (5) на стор. 48 у модулі «Множи́ни та відношення». Але це *не* означає, ніби тепер будуть розглянуті також перетин, симетрична різниця, тощо — вони за́раз не потрібні.)

Конкатенація Конкатенація слів — це утворення сло́ва послідовним дописуванням двох чи кількох слів. Наприклад, мовою C++ "`string s1="bath", s2="room"; string s3=s1+s2;`" надає змінній `s3` значення `"bathroom"`, отримане дописуванням 2-го сло́ва `room` після 1-го сло́ва `bath`. Конкатенація *не* комутативна: `bath`+`room`=`bathroom`≠`roombath`=`room`+`bath`.

Тепер, на основі щойно розглянутої операції «конкатенація слів», вважаючи її відомою, введемо нову операція «конкатенація мов».

Конкатенація (вона ж *добуток*; рос. *«конкатенация»*, англ. *«concatenation»*) мов A та B — множина всіх слів, які можуть бути утворені як конкатенація сло́ва мови A і сло́ва мови B (са́ме в такому порядку):

$$A \cdot B \stackrel{\text{def}}{=} \{pq \mid p \in A \wedge q \in B\} \tag{75}$$

(де між "{" та "|" відбувається конкатенація слів p та q).

(Крім "$A \cdot B$", конкатенація може позначатися також як "AB" (знак операції пропускають, аналогічно множенню чи кон'юнкції), або як "$A * B$". Позначення зірочкою незручне, бо зірочка меншого розміру й вище розміщена позначає іншу операцію «ітерація» (див. далі); але у деяких джерелах так все ж пишуть.)

Конкатенація мов дещо нагадує декартів добуток — розглядаються па́ри кожного сло́ва 1-ої мови з кожним словом 2-ої. Наприклад, $\{a, b\} \cdot \{a, c, d\} = \{aa,\ ac,\ ad,\ ba,\ bc,\ bd\}$. Але аналогія з декартовим добутком не повна, бо утворюються не пари, а слова́. Наприклад, $\{a, ab\} \cdot \{bc, c\} = \{abc,\ ac,\ abbc\}$ — результат містить три сло́ва, а не чотири, бо як поєднання a з bc, так і поєднання ab з c дають одне й те ж слово abc (а результатом конкатенації мов є мова, яка є класичною множиною, а не мультимножиною, слів).

Відзначимо (хоч це й випливає з решти означень), що $a\varepsilon = \varepsilon a = a$, тобто конкатенація будь-якого сло́ва з порожнім словом, незалежно від того, чи ε виступає правим аргументом конкатенації, чи лівим, дає саме́ це слово. За рахунок цього, якщо відбувається конкатенація мов, які містять у собі ε (як елемент), то конкатенація включає у себе (як підмножину) об'єднання. Але са́ме *якщо* мови-аргументи містять у собі ε, а не завжди. Наприклад:

$$\begin{aligned}
\{a, ab\} \cdot \{bc, c\} &= \{abc, ac, abbc\};\\
\{\varepsilon, a, ab\} \cdot \{bc, c\} &= \{bc, c, abc, ac, abbc\};\\
\{a, ab\} \cdot \{\varepsilon, bc, c\} &= \{a, abc, ac, ab, abbc\};\\
\{\varepsilon, a, ab\} \cdot \{\varepsilon, bc, c\} &= \{\varepsilon, bc, c, a, abc, ac, ab, abbc\}.
\end{aligned}$$

Степінь На основі конкатенації (добутку) мов вводять операцію *степінь* (рос. «*степень*», англ. «*power*») мови

$$A^n \stackrel{\text{def}}{=} \underbrace{A \cdot A \cdot \ldots \cdot A}_{n \text{ штук "}A\text{"}} \tag{76}$$

(тут A — мова, n — невід'ємне ціле число); для будь-якої мови A, вважається $A^0 = \{\varepsilon\}$ (не $\varnothing$, а са́ме множина, яка містить ε як елемент).

Наприклад, $\{a, z\}^3 = \{aaa,\ aaz,\ aza,\ azz,\ zaa,\ zaz,\ zza,\ zzz\}$; $\{\varepsilon,\ a,\ ab,\ b\}^2 = \{\varepsilon,\ a,\ ab,\ b,\ aa,\ aab,\ aba,\ abab,\ abb,\ ba,\ bab,\ bb\}$.

Ітерація (зірочка Кліні) *Ітерація* (вона ж *зірочка Кліні*; рос. «*итерация*», «*звёздочка Клини*», англ. «*Kleene star*») мови A (позначається A^*) — об'єднання всіх можливих степенів мови A:

$$A^* \stackrel{\text{def}}{=} \{\varepsilon\} \cup A \cup A^2 \cup A^3 \cup \ldots \tag{77}$$

(показник степеню перебирає всі чи́сла з $\mathbb{N}^+$, прямуючи до $+\infty$).

Ітерація скіче́нної (і за кількістю слів, і за максимальною довжиною сло́ва) мови майже завжди є нескінче́нною (і за кількістю слів, і за максимальною довжиною сло́ва) мовою. Наприклад, $\{0, 1\}^* = \{\varepsilon,\ 0,\ 1,\ 00,\ 01,\ 10,\ 11,\ 000,\ 001,\ 010,\ 011,\ 100,\ 101,\ 110,\ 111,\ \ldots\}$ (до нескінченності).

Мов, ітерація яких скінче́нна, є в точності дві: $\varnothing$ та $\{\varepsilon\}$. Це різні мови: $\{\varepsilon\}$ містить (як елемент) хоча б порожнє слово, а $\varnothing$ не містить взагалі нічого. А ітерації цих мов однакові: $\varnothing^* = \{\varepsilon\}^* = \{\varepsilon\}$.

Означення регулярної мови Саме́ поняття «*регулярна мова*» вводиться за допомогою такого рекурсивного означення.

1. Будь-яка мова, що складається зі скінче́нної (в т. ч. порожньої) сукупності скінче́нних слів (в т. ч. порожнього сло́ва), регулярна;
2. Якщо A та B — регулярні мови, то регулярними є також і мови $A \cup B$ та $A \cdot B$;
3. Якщо A — регулярна мова, то регулярною є також і мова A^*.

   (Якщо ж формальна мова не може бути отримана за допомогою скінче́нної кількості застосувань цих операцій, то вона не є регулярною.)

Регулярні вирази (regex-и) Регулярні мови найчастіше записують *регулярними виразами* (рос. «*регулярные выражения*», англ. «*regular expression*», скорочено «*regex*»; ще є скорочення "reg. exp.", "рег. вираз", "РВ"; ми найчастіше вживатимемо скорочення «regex», бо воно найпопулярніше у практичних застосуваннях).

Дії над regex-ами відповідають таким діям над регулярними мовами:

1. regex, що задає одне скінче́нне (в т. ч. порожнє) слово, співпадає з само́м цим словом;
2. якщо рег. мови A та B задаються regex-ами R_A та R_B, то рег. мова $A \cup B$ задається regex-ом $(R_A \mid R_B)$;
3. якщо рег. мови A та B задаються regex-ами R_A та R_B, то рег. мова $A \cdot B$ задається regex-ом $R_A R_B$;
4. якщо рег. мова A задається regex-ом R_A, то рег. мова A^* задається regex-ом $(R_A)^*$ (якщо regex R_A односимвольний або і так взятий у дужки, то ще одні дужки не ставляться).

   (Все, що не може бути отримане за допомогою скінче́ннної кількості застосувань вказаних операцій, не є регулярними виразами.)

Тобто, регулярні мови та регулярні вирази взаємопов'язані, але являють собою різні сутності (мають різні типи): рег. мова — це множина слів (часто нескінче́нна), а рег. вираз — це скінче́нна формула, яка задає рег. мову, вказуючи, за допомогою яких операцій ця мова побудована.

Операції над регулярними виразами називають так само, як відповідні операції над регулярними мовами (*об'єднання*, *конкатенація* та *ітерація*); але для операції $(R_A|R_B)$ дуже поширена ще одна назва — *альтернатива* (причому так називають лише операцію над regex-ами, не над рег. мовами).

Приклади регулярних виразів та відповідних їм регулярних мов

1. Regex «ba^*» задає усі слова́, що починаються з b і далі містять довільну кількість (в т. ч. 0) символів a; рег. мова: $\{b, ba, baa, baaa, baaaa, \ldots\}$.
2. Regex «a^*b^*» задає усі слова́, що містять спочатку довільну кількість (в т. ч. 0) символів a, потім довільну кількість (в т. ч. 0) символів b; рег. мова: $\{\varepsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \ldots\}$.
3. Regex «$(ab)^*$» задає усі слова́, що довільну (в т. ч. 0) кількість разів повторюють підслово ab (тобто повторюватися можуть лише неподільні пари ab, а не окремі a чи b); рег. мова: $\{\varepsilon, ab, abab, ababab, \ldots\}$.
4. Regex «$(a \mid b)^*$» задає взагалі всі слова́ в алфавіті $\{a, b\}$, тобто символи a, b йдуть у будь-яких кількостях та як завгодно перемішані; рег. мова: $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\}$.
5. Regex «$(a^*|b^*)$» задає слова́, що складаються або з довільної кількості a, або з довільної кількості b, але не можуть містити і a, і b одночасно; рег. мова: $\{\varepsilon, a, b, aa, bb, aaa, bbb, aaaa, bbbb, \ldots\}$.
6. Regex «$(a \mid bb)^*$» задає послідовності з a та b, куди b входять лише парами bb; рег. мова: $\{\varepsilon, a, aa, bb, aaa, abb, bba, aaaa, aabb, abba, bbaa, bbbb, aaaaa, aaabb, aabba, abbaa, abbbb, bbabb, bbbba, \ldots\}$.
7. Формальна мова ідентифікаторів з розд. 6.1 є регулярною і її можна задати regex-ом «$(A \mid B \mid \ldots \mid Z \mid \_)\ (A \mid B \mid \ldots \mid Z \mid \_ \mid 0 \mid 1 \mid \ldots \mid 9)^*$».
   (У цьому та наступному прикладах для скорочення запису вжито "…"; взагалі-то більш правильно явно перелічувати всі можливі варіанти.)
8. Формальна мова цілих чисел з розд. 6.1 є регулярною і її можна задати regex-ом «$(0 \mid \mid (+ \mid - \mid \varepsilon)\ (1 \mid 2 \mid \ldots \mid 9)\ (0 \mid 1 \mid \ldots \mid 9)^*$)».

Чи існують формальні мови, які не є регулярними? Скільки завгодно. Наприклад, не є регулярною (занадто складна́, щоб задати її регулярним виразом) формальна мова всіх осмислених речень природньою мовою (українською, англійською, тощо). І навіть значно простіші формальні мови, придатні для комп'ютерної обробки (іншими засобами): мова синтаксично правильних програм мовою C++ (і не лише C++, а й переважною більшістю реальних мов програмування); і навіть такий простіший варіант, як мова синтаксично правильних арифметичних виразів, які

можуть мати дужки довільної вкладеності; і навіть такий простіший варіант, як $\{({}^k a)^k \mid k{\in}\mathbb{Z}^+\} = \{a, (a), ((a)), (((a))), ((((a)))), \ldots\}$, тобто множина всіх слів, де спочатку йдуть відкривні дужки, потім буква a, потім закривні дужки — *рівно стільки ж*, скільки раніше було відкривних. Ця неможливість навіть доведена далі по тексту (див. розд. 6.5.3, 6.5.6, 6.5.7).

6.3 Практичне застосування regex-ів

Regex-и, крім того, що мають велике теоретичне значення, є ще й потужним засобом практичної обробки текстів. У багатьох текстових редакторах та багатьох мовах програмування вбудована підтримка regex-ів. При практичній реалізації виявляється доцільним дещо відступати від описаних вище математичних правил; як наслідок, regex-и з різних редакторів чи мов програмування мають різний синтаксис. Тож відзначимо лише найголовніші й найтиповіші особливості більшості таких реалізацій, а для повноти картини все одно треба користуватися більш технічними описами конкретної реалізації. Наприклад, для мови java — `https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html`

1. Для деяких символів та груп символів є спеціальні позначення, як-то:
   (а) `.` (крапка) — будь-який один символ;
   (б) `\d` — будь-яка цифра;
   (в) `\D` — будь-який символ, крім цифр;
   (г) `\w` — будь-який «символ сло́ва» (цифра, буква або "`_`"); стандартно, маються на увазі лише латинські букви, але в деяких реалізаціях це може залежати від налаштувань;
   (д) `\W` — будь-який *не* «символ сло́ва»;
   (е) `\s` — будь-який «порожній» символ (пробіл, або табуляція, або переведення рядка);
   (і т. д.)
2. Символи, які не можна писати безпосередньо, бо вони мають спеціальний смисл, пишуть через "`\`": сам бекслеш пишеться "`\\`", крапка — "`\.`", круглі дужки — "`\(`" та "`\)`", тощо.
3. Якщо потрібно вказати «будь-який символ переліку», перелік записується у квадратних дужках (наприклад, "`[аєиіїоуюя]`" позначає будь-яку (одну!) голосну букву української абетки в нижньому регістрі); перелік може містити символ "`-`", котрий трактується як тире, тобто вказує діапазон (наприклад, "`[1-7]`" позначає будь-яку цифру в діапазоні від 1 до 7); «внутрішні» символи діапазону визначаються згідно порядку ASCII (або іншої кодової таблиці, яку використовує реалізація); зразу після дужки "`[`" перед переліком може стояти «кришка» "`^`" — це означає, що потрібно шукати всі символи, *крім* вказаних у переліку.
4. Об'єднання (альтернатива) позначається символом "`|`", але у правилах розстановки круглих дужок (чи згідно математичного означення, чи брати в дужки кожен аргумент окремо, чи такі дужки не обов'язкові) різні реалізації можуть відрізнятися одна від о́дної.
5. Конкатенація ніяк спеціально не позначається, вирази просто записують один після одного.
6. Ітерація реалізована у багатьох «версіях»:
   (а) «Стандартна» ітерація (повторити 0 разів, або 1 раз, або 2 рази, і т. д.) позначається зірочкою "`*`"; наприклад, "`м[ие]*р`" задає слова́ «мр», «мир», «мер», «миир», «миер», «меир», «меер», ...;
   (б) «непорожня» ітерація (повторити 1 раз, або 2 рази, і т. д.) позначається плюсом "`+`"; наприклад, "`м[ие]+р`" задає слова́ «мир», «мер», «миир», «миер», «меир», «меер», ... — але не слово «мр»;
   (в) необов'язкова частина (включити один раз або не включати взагалі) позначається знаком питання "`?`"; наприклад, "`м[ие]?р`" задає (лише) слова́ «мр», «мир», «мер»;
   (г) ітерація з обмеженою кількістю повторів: після regex-а у фігурних дужках вказується число (`(...){n}`), або число і кома (`(...){n,}`), або кома і число (`(...){,m}`), або число, кома і число (`(...){n,m}`) — в усіх випадках конкретний regex повинен містити не змінну, а конкретне число, як-то `\d{3}` (рівно три цифри підряд), а позначки `n` та `m` вжиті лише у тому смислі, що в різних regex-ах ці чи́сла можуть бути різними.

Якщо коми нема — число вказує точну кількість повторів; інакше, число перед комою вказує мінімальну кількість повторів, після коми — максимальну; якщо кома є, і вказане лише одне число, то з іншого боку кількість повторів не обмежується. Наприклад, вираз `\d{2,5}` задає дво-, три-, чотири- та п'ятизначні чи́сла; вираз `\d{,4}` — чи́сла з не більш ніж чотирма цифрами, а також порожнє слово; вираз `\d{3,}` — чи́сла з не менш ніж трьома цифрами.

(Значна частина реалізацій regex-ів містять ще більше різновидів ітерацій, бо вона ще може бути або *жадібною* (*greedy*), або *лінивою* (*reluctant*), або *ревнивою* (*possessive*); але це вже не включено до посібника, бо такі особливості можуть бути різними для різних програм–обробників regex-ів. Такі деталі пропонується вивчати окремо, у прив'язці до конкретної програми–обробника regex-ів.)

Один з *най*потужніших засобів застосування regex-ів до обробки текстів (придуманий програмістами-практиками, а не вченими, які вивчали регулярні вирази теоретично) — можливість проводити не лише пошук відповідностей regex-зразку́, а ще й *заміну* знайдених входжень regex-ів. Можна запам'ятовувати частини рядка, що відповідають частинам знайденої у тексті відповідності regex-у, і підставляти у вираз, на який замінюють. («Частина рядка» може бути й усім рядком; важливо, що є вибір, чи використати весь рядок, чи частину, і якщо частину, то яку са́ме.) Для цього частину, яку треба запам'ятати, у виразі пошуку беруть у круглі дужки, а у виразі заміни записують `$1` (або `$2`, або `$3` — відповідно до того, який по порядку з запам'ятованих виразів потрібно підставити; номер визначається як послідовний (1, 2, 3, ...) номер відкривної круглої дужки у виразі пошуку).

Тут реалізації теж можуть відрізнятися одна від о́дної: по-перше, деякі замість `$1`, `$2`, `$3`, ... вживають `\1`, `\2`, `\3`, ...; по-друге, значна частина реалізацій (але не всі) передбачають, що завжди є нульова́ група `$0`, яка означає відповідність усьому виразу пошуку (навіть якщо він не взятий у дужки); по-третє, по-різному трактуються багатоцифрові чи́сла після долара (наприклад, `$17` деякі реалізації сприймають як «запам'ятоване сімнадцятими дужками», деякі — як «запам'ятоване першими дужками, після чого цифра 7», а деякі автоматично вибрають з цих варіантів: «якщо regex пошуку містить 17-ті дужки, то запам'ятоване цими 17-ми дужками, а якщо не містить, то запам'ятоване 1-ми дужками, після чого цифра 7»).

Як уже згадано, є кілька різних варіантів використання всіх цих засобів. Можна використати їх як складову програми мовою програмування (викликáти, де треба, бібліотечні функції, які перетворюють відповідним чином змінні типу `string` — якщо, звісно, є такі бібліотеки). Можна прямо у рядку пошуку й рядку заміни віконечка «Знайти і замінити» текстового редактора — якщо, звісно, редактор таке підтримує. Можна користуватися утилітою `grep` або подібною. Якщо говорити про мову C++, такі засоби (бібліотека `regex`) існують, починаючи зі стандарту C++11, наприклад, вони *є* у Microsoft Visual Studio Express 2012; але вони поки що не досить кросплатформенні (те, що працює в одному компіляторі C++, може й не працювати в іншому, навіть якщо вони обидва стандарту C++11). У деяких інших мовах (Java, Perl, тощо) підтримка regex-ів з'явилася значно раніше, тому там вони більш стандартні та надійні. Якщо говорити про текстовий редактор — наприклад, у тому самому Microsoft Visual Studio Express 2012 регулярні вирази можна використовувати не лише у програмі, яку пишемо, щоб з нею працювали компілятор та бібліотека, а ще й під час редагування програми у цій самій Studio, у віконечку `Quick Replace` (`Ctrl+H`) є область із зображенням "`.*`" та підказкою (hint-ом) «`Use regular expressions (Alt+E)`», і якщо натиснути на цю область — можна робити пошук і заміну з використанням regex-ів. Є й багато інших редакторів: Notepad++ (але там дещо інший синтаксис regex-ів), вбудований редактор оболонки FAR manager, тощо.

Microsoft Word надає багато інших, в тому числі й напрочуд корисних, засобів пошуку та заміни; але, наскільки відомо автору посібника, са́ме регулярні вирази там усе ще реалізовані лише частково. У старіших версіях, були лише спеціальні знаки (як-то «будь-яка цифра», «будь-який з пропускових символів (пробіл, табуляція, тощо)») та можливість підставляти у вираз заміни *увесь* знайдений рядок. У новіших версіях Word, можливість підставити у вираз заміни також і виділену частину знайденого з'явилась (з відставанням років на 15–20 від деяких інших текстових редакторів), але, наскільки відомо автору посібника, повноцінних ітерації та об'єднання все ще нема.

Наведемо кілька прикладів, які показують: заміни з використанням запам'ятованих частин regex-ів — дуже потужний і дуже корисний на практиці, але не всемогутній засіб.

Приклад 1. Нехай є великий текст, у якому багато чисел – десяткових дробів (ціла частина, кома, дробова частина). Причому, в якості роздільника використана саме кома ("`,`"), згідно радянського стандарту. Нехай треба перетворити всі ці десяткові дроби до західного стандарту, де ціла й дробова частини розділяються крапкою ("`.`"). Нехай до того ж просто позаміняти всі коми на крапки не можна, бо в тексті дуже багато також інших ком (які, наприклад, розділяють частини речення), і їх треба лишити без змін.

Для виконання такої заміни достатньо шукати фрагменти "`(\d),(\d)`" і заміняти їх на "`$1.$2`". Тут "`\d`" означає десяткову цифру, дужки — вказівку запам'ятати цю цифру (щоб потім підставити). Якщо застосувати таку заміну, наприклад, до тексту *«збільшення, з 42,17 до 123,456»*, то зразок буде знайдено двічі: *«збільшення, з 42,17 до 123,456»*, у першому входженні буде запам'ятовано "2" у `$1`, "1" у `$2`, у дру́гому — "3" у `$1`, "4" у `$2`. Потім перше входження буде замінено на *"2.1"*, дру́ге — на *"3.4"*, і весь текст набуде вигляду *«збільшення, з 42.17 до 123.456»* (що й треба).

Причому, це працює навіть якщо в одному тексті є і десяткові дроби через коми, і переліки чисел через коми та пробіли, як-то "*2, 3, 5, 7, 11*": після цифри й коми йде пробіл, а не цифра, тож ці фрагменти не відповідають зразку "`(\d),(\d)`" й тому лишаться незмінними.

Щоправда, якщо текст містить також і переліки чисел, записані через коми без пробілів — така заміна вчинить неправильно, змінивши, наприклад, "*2,3,5,7,11,13,17*" на "*2.3,5.7,11.13.17*". Але, коли із запису "*2,3*" не ясно, чи то дві цілі три десяті, чи перелік двох чисел 2 та 3 — це, мабуть, проблеми не регулярних виразів, а початкового тексту...

Приклад 2. Нехай у тексті є дати у форматі `mm/dd/yyyy`, тобто через похилі риски місяць, день, рік (наприклад, 24 серпня 1991 року має вигляд "*08/24/1991*"), і треба перевести їх у формат `dd.mm.yyyy`, тобто день, місяць, рік через крапки (та са́ма дата набуде вигляду "*24.08.1991*").

Якщо гарантовано, що день та місяць містять рівно по 2 цифри, рік рівно 4, і нема інших, крім дат, послідовностей з цифр та похилих рисок, то можна шукати "`(\d{2})\/(\d{2})\/(\d{4})`" і замінювати на "`$2.$1.$3`" (де `$1` містить запам'ятований місяць, як 1-ше число, `$2` — запам'ятований день, як 2-ге число, `$3` — запам'ятований рік).

Якщо ж таких гарантій нема, то ці regex-и пошуку й заміни можуть наробити багато украй недоречних замін, як-то "`99/88/7777/66/55/4444`" на "`88.99.7777/55.66.4444`". Що з цим можна (і що майже не реально) зробити, частково описано у наступному при́кладі.

Приклад 3. Нехай треба знайти IP-адреси формату IPv4, тобто 4 числа́, розділені крапками, кожне у проміжку від 0 до 255 (зарезервованістю деяких таких послідовностей знехтуємо). У першому наближенні це просто: "`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`" (де "`\.`" означає, що у тексті має бути са́ме крапка; "`.`" у regex-і пошуку означало б «будь-який символ»).

Якщо regex шукає у тексті, в якому все, що схоже на IPv4-адреси, є ними — це годиться. Але якщо мета інша, як-то *перевірити, чи* введений користувачем рядок являє собою коректну IPv4-адресу, це не годиться, хоча б тому, що такий regex сприймає як правильний рядок "*999.666.1.555*" (хоча всі чотири числа́ повинні бути від 0 до 255).

Ні «математичні» рег. вирази, ні більшість програмних реалізацій regex-ів просто не оперують поняттям «значення числа́», а отже й поняттям «число належить проміжку». Якщо в таких умовах все-таки треба перевірити належність проміжку, доводиться *кожен з чотирьох* фрагментів "`\d{1,3}`" замінити, наприклад, на "`((2((5[0-5])|([0-4]\d)))|(1?\d{1,2}))`". Тут фрагмент "`2((5[0-5])|([0-4]\d))`" означає «починається з цифри 2, далі є два варіанти: або цифра 5, після якої цифра від 0 до 5, або цифра від 0 до 4, після якої будь-яка цифра», що разом виражає «число від 200 до 255». Фрагмент "`1?\d{1,2}`" означає «одно- чи двоцифрове число, перед яким спереду може бути або не бути цифра 1», що виражає «число від 0 до 199». У конкретних реалізаціях regex-ів частину з цих круглих дужок можна пропускати, але біда в тім, що в різних реалізаціях це можуть бути різні дужки.

Якщо таку перевірку відповідності рядка формату IPv4-адреси треба зробити у тексті програми (мовою програмування, що підтримує regex-и), може бути доцільнішим інший спосіб: перевірити відповідність regex-у "`^(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})$`"; *якщо* відповідає,

то перетворити кожен з цих 4-х фрагментів у число і перевірити значення всіх цих чисел. (Бібліотека роботи з regex-ами дозволяє отримати, яка са́ме послідовність символів була виділена 1-ми круглими дужками, яка 2-ми, тощо, а перетворення цих послідовностей цифр з рядкового подання у числове — функція з іншої стандартної бібліотеки.) Причому, раз ці фрагменти перетворюватимуть у чи́сла лише якщо текст відповідає regex-у, значить перетворення їх з рядкового подання у чисельне не призведе до помилок формату.

Вжиті у попередньому абзаці "`^`" («кришка») та "`$`" (долар) означають «входження зразка мусить починатися від самого початку тексту/рядка» та «входження мусить закінчуватися наприкінці тексту/рядка» відповідно. Щоб не казати «правильна IPv4-адреса» на, наприклад, *увесь* рядок "*zzz12.123.0.12345*" чи "*1.2.3.4.5.6.7*". Іншими словами це ще називають «рядок *являє собою* потрібний зразок», а не «*містить* зразок, і, можливо, ще щось». У деяких бібліотеках це виражається ще й тим, що для цих різних перевірок «являє собою» і «містить» використовують функції з різними іменами. Але згадані "`^`" та "`$`" — теж широко вживаний спосіб.

Сумно, що це така само «кришка», як позначення «будь-який символ, *крім* переліку» та такий само долар, як «запам'ятоване у дужках № ...», але так вирішили розробники програмних реалізацій regex-ів. Розрізнити один смисл «кришки» від іншого можна за тим, що смисл «початок тексту/рядка» — на початку зразка, «символ крім переліку» — у квадратних дужках; один смисл долара від іншого — за тим, що смисл «кінець тексту/рядка» — наприкінці зразка пошуку, «підставити запам'ятоване» — у зразку заміни перед цифрою. Більшість конкретних реалізацій мають ще й інші подібні сумні ситуації, коли один символ може мати різний смисл залежно від контексту (обставин, в яких він ужитий).

Приклад 4. Знайти (нічого не замінюючи) усі місця́, де одна й та сама послідовність з трьох або більше «символів слів» (`\w`) повторюється два рази «майже підряд», а са́ме: між входженнями цієї послідовності $\leqslant 5$ інших символів. Наприклад: "*барбарис*", "*відповідь*", "*матрица достижимости*".

Засобами *«математичних»* регулярних виразів (означених на стор. 164), такий пошук *принципово неможливий* (див. також розд. 6.5.3), бо тими засобами не можна задати умову, що оброблюваний текст містить *один і той самий* (нетривіальний) підрядок *кілька разів*. Іншими словами: «математичними» рег. виразами можна задати, наприклад, «в одному місці зустрілося конкретно "*abc*", в другому теж зустрілося конкретно "*abc*"». Також можна задати, наприклад, «в одному місці зустрілася відповідність зразку $a^*b^*c^*$, в іншому теж зустрілася відповідність зразку $a^*b^*c^*$» — але тоді ці відповідності можуть бути різними, наприклад "*abbbbc*" та "*abccc*". А щоб одночасно і шукати не конкретне слово, а відовідність regex-у (який може містити альтернативи та/або ітерації), і щоб друге входження дорівнювало першому — цього «математичними» регулярними виразами задати не можна.

У *більшості програмних реалізацій* regex-ів така можливість все ж є (завдяки тому, що ці реалізації використовують в т. ч. й засоби, «заборонені» з точки зору математиків, які вперше придумали регулярні вирази). Regex для конкретно згаданого пошуку має вигляд "`(\w{3,}).{,5}\1`" (якщо вважати, що між двома входженнями однакового фрагменту можуть бути будь-які символи; інакше, замість крапки слід вжити щось конкретніше). А якщо говорити взагалі, то "`\1`", або "`\2`", або "`\3`", ... у рядку пошуку виражає «знов таке са́ме, як було в 1-х/2-х/3-х/... дужках».

Зловживання складни́ми засобами ("`\1`", "`\2`", ... — один з них, але не єдиний) може призводити до того, що обробка таких regex-ів програмою чи редактором триватиме *дуже* довго.

Приклад 5. Нехай є велика Паскаль-програма, у якій багато операторів `read` та `readln`, що читають дані з клавіатури; нехай треба, щоб вони читали ті самі дані з текстового файла `f`.

На перший погляд, достатньо виконати дві звичайні (без усяких регулярних виразів) заміни: "`read(`" на "`read(f,`" та "`readln(`" на "`readln(f,`".

Але у не акуратно написаних програмах цілком реальна ситуація, коли частина таких операторів містить непотрібні, але допустимі пробіли між словом `read` чи `readln` та дужкою "`(`", і для повного щастя у різних операторах кількості цих пробілів можуть бути різними. Якщо не користуватися regex-ами, з усім цим доводиться розбиратися окремо.

Водночас, потрібне перетворення тексту програми можна зробити за допомогою *однієї* заміни з використанням regex: шукаємо "`(read(ln)?\s*\()`" і замінюємо на "`$1f,`". Або, якщо хочемо заодно ще й поприбирати непотрібні пробіли, шукаємо "`(read(ln)?)\s*\(`" і замінюємо на "`$1(f,`".

Ці заміни не чіпають оператори `read`/`readln` без параметрів; якщо їх теж потрібно перетворити, це краще зробити окремим проходом, замінюючи "`(read(ln)?)(\s*[^\(])`" на "`$1(f)$2`". (Може здатися природнішим міняти "`(read(ln)?)\s*\;`" на "`$1(f);`"; але це погана ідея, бо така заміна «не помічатиме» оператори, які не завершені крапкою з комою (у мові Паскаль оператор не завжди завершуватися крапкою з комою — наприклад, вона не ставиться перед `else`).)

Приклад 6. Нехай у Паскаль-програмі є багато викликів бінарного варіанта процедури `inc`, тобто «збільшити таку-то змінну на стільки-то». Нехай треба замінити всі `inc`-и на присвоєння (наприклад, "`inc(p,2)`" →"`p:=p+2`"). Якщо хотіти однією парою regex-ів виконати перетворення не лише для конкретної змінної `p`, а зразу для всіх можливих аргументів `inc`-а, то абсолютно правильного розв'язку така задача насправді не має: регулярними виразами не завжди можливо правильно визначити аргументи.

Спроба шукати "`inc\((.*)\,(.*)\)`" і заміняти на "`$1:=$1+$2`", тобто шукати фрагмент "`inc(`", після нього що завгодно (його вважатимемо першим аргументом `inc`-а), після нього кому, після нього що завгодно (його вважатимемо дру́гим аргументом `inc`-а), після нього закривну фігурну дужку — погана ідея, таке перетворення більше зашкодить, ніж допоможе, бо часто спрацьовуватиме там, де не треба. Наприклад, замінюючи[48] "`inc(i); write('i=',i)`" на "`i); write('i=':=i); write('i='+i`" (підкреслений фрагмент вибирається в якості `$1`). Вираз "`.*`" «занадто потужний», а точніше — занадто нерозбірливий...

Якщо замінювати лише те, в чому цілком упевнені — можна, наприклад, шукати "`inc\(([A-Za-z_][A-Za-z_0-9]*), ([A-Za-z_0-9+-*/\s]+)\)`" й заміняти на таке само "`$1:=$1+$2`". Тоді точно не буде хибних (непотрібних) спрацювань: у перших дужках "`[A-Za-z_][A-Za-z_0-9]*`" задає ідентифікатор (починається з букви або знаку підкреслення, потім ще скільки завгодно букв або знаків підкреслення або цифр); у дру́гих дужках "`[A-Za-z_0-9+-*/\s]+`" задає якесь наближення до поняття допустимого виразу і точно не захопить нічого зайвого, бо не захопить дужки, якою закінчується перелік аргументів `inc`-а. Але такі пошук та заміна не перетворять, наприклад, ні "`inc(a[i], 1)`", ні "`inc(sum, a[i])`".

Можна шукати "`inc\(([^\,\)]*)\,([^\)]*)\)`" й замінювати на той самий "`$1:=$1+$2`": тут вказано, що перший аргумент `inc`-а не може містити всере́дині себе ні коми, ні закривної дужки, другий аргумент — закривної дужки. Така пара правильно розбереться із вищезгаданими випадками, але знов не є абсолютно правильною, бо, наприклад, "`inc(mass[i,j])`" буде помилково перетворено у "`mass[i:=mass[i+j]`"; у виразі "`inc(s,f(a)+f(b))`" закривна дужка з підвиразу "`f(a)`" буде помилково сприйнята за закривну дужку усього `inc`-а, що теж неправильно.

І так далі. *Слід дбати і про те, щоб* ***знаходити все, що треба****, і про те, щоб* ***не знаходити, чого не треба****, і це не завжди легко поєднати.* Особливо, якщо хотіти знайти зразки згідно формальної мови, яка взагалі-то не регулярна (див. кінець розд. 6.2 та наведені там посилання).

Тим не менш, при використанні regex-ів у редакторі (а не у програмі) буває дуже зручно виконати перетворення у напівавтоматичному режимі (задавши парою regex-ів правило виконання замін, вручну підтверджувати або пропускати кожну конкретну заміну), а зі складними випадками, що лишилися, розбиратися суто вручну.

6.4 Коротко про автомати взагалі

Існує багато різновидів автоматів, і «автомат взагалі» можна лише приблизно описати словами; строгі означення різновидів будуть дані пізніше.

Автомат (він же *абстрактний автомат*; рос. «*автомат*», «*абстрактный автомат*»; англ. мовою поширені як варіант «*automaton*»[49], так і варіанти «*state machine*», «*finite-state machine*») є математичною моделлю пристрою, що працює потактово, читаючи на кожному такті один вхідний символ і якось його обробляючи.

[48] така неправильна заміна точно виконається, якщо фрагменти будуть в одному рядку (через пробіл або табуляцію); якщо вони будуть у різних рядках (через Enter), то виконання чи невиконання залежить від налаштувань програми–обробника regex-ів

[49] множина (множественное число, plural) утворюється за правилами латинської мови, як «*automata*»; казати/писати automatons неправильно

Для різних автоматів символи можуть братися з різних наборів, але для кожного конкретного автомата цей набір (*вхідний алфавіт*; рос. «*входной алфавит*», англ. «*input alphabet*») повинен бути відомий заранє.

Ми розглядатимемо переважно автомати, вхідні алфавіти яких — деякі з букв, цифр, розділових знаків, тощо. Але можливі й принципово інші алфавіти. Вже згадувалося, що теоретично ніщо не заважає розглядати, наприклад, алфавіт $\{\Uparrow, \perp, \not\prime, \xi, 7, \check{y}, z, \spadesuit, \dagger\}$. А насправді все ще цікавіше, бо символи алфавіту автомата взагалі не зобов'язані бути значками, які можна зобразити. Наприклад, реальний пристрій, що керує ліфтом, можна описати абстрактним автоматом, вхідний алфавіт якого містить символи *«натиснена кнопка ззовні ліфта на . . .-му поверсі»* (для кожного поверху свій символ), *«у ліфті натиснена кнопка . . .-го поверху»* (теж для кожного поверху), *«перешкода при зачиненні дверей»*, тощо.

Якщо ліфт зайнятий, він реагує на натискання кнопок інакше, ніж коли він вільний. Це про́яв іншої характерної властивості автоматів — результат залежить *і* від того, який символ отриманий на вході, *і* від того, в якому *стані* (рос. «*состояние*», англ. «*state*») перебуває автомат.

Стани для того й потрібні автомату, щоб розрізнятн ситуації, коли треба по-різному реагувати на один і той самий вхідний символ.

На кожному проміжку між тактами автомат перебуває в якомусь одному стані[50]; це називають також «*поточний* стан» (рос. «*текущее* состояние», англ. «*current* state»). Під час такту стан може змінитися на якийсь інший або лишитися тим самим. Точніше кажучи, кожен новий стан залежить від попереднього стану і від символу, отриманого на вході — й ні від чого іншого.

Як правило, розглядають *ініціальні* (рос. «*инициальные*», англ. «*initial*») автомати, для яких один зі станів виділений як *початковий* (рос. «*начальное* состояние», англ. «*start* state»). Це стан, в якому перебуває автомат перед початком робо́ти, ще не прочитавши жодного вхідного символу.

Вхідним словом (рос. «*входное слово*», англ. «*input string*») будемо називати повну, від самого початку, послідовність вхідних символів. Неважко переконатися, що *для будь-якого ініціального автомата поточний стан повністю визначається вхідним словом.*

Доведення. Автомат ініціальний ⇒ початковий стан відомий. Якщо вхідне слово порожнє, автомат залишається у початковому стані. Інакше, після обробки першого символу, автомат переходить до стану, який залежить лише від відомого початкового стану й відомого символу. Тобто, цей стан теж відомий. Це міркування можна повторювати, доки не закінчиться вхідне слово. Значить, і вся послідовність станів ініціального автомата, і, зокрема, останній стан цієї послідовності, визначаються лише вхідним словом. ■

Отже, *стан автомата* ***зберігає часткову інформацію*** *про початок вхідного сло́ва* (цей початок ще називають *передісторією*; рос. «*предыстория*», англ. «*prehistory*»).

6.5 Автомати-розпізнавачі

6.5.1 Означення та способи подання автомата-розпізнавача

Автомат-*розпізнавач* (рос. «автомат-*распознаватель*», англ. «*acceptor*», «*recognizer*») — це п'ятірка $\langle S, s_{start}, X, F, f\rangle$, де

1. S — множина станів;
2. s_{start} — початковий стан;
3. X — множина (алфавіт) вхідних символів;
4. $F \subseteq S$ — множина *допуско́вих* станів;
5. $f : S \times X \to S$ — функція переходів.

З множинами S та X повинно бути більш-менш ясно (з загального опису для всіх різновидів автоматів). З початковим станом s_{start} теж, варто лише додати, що s_{start} — один з елементів S. У деяких (далеко не всіх) джерелах стверджується, ніби початковим завжди повинен бути стан s_0; ми вважатимемо, що початковим може бути будь-який із станів.

[50] за виключенням недетермінованих автоматів, що розглядаються у розд. 6.5.4–6.5.7

Перейдемо до функції переходів (рос. «*функция переходов*», англ. «*state-transition function*»). Запис "$f : S \times X \to S$" означає «функція f приймає два параметри (1-ий з множини́ S, 2-ий з множини́ X) і повертає результат, що належить множині S». Тобто, функція переходів f визначає, до якого са́ме стану має перейти автомат, якщо він, перебуваючи у такому-то стані (1-ий аргумент функції), прочитав такий-то вхідний символ (2-ий аргумент).

Перейдемо до множини́ F ($F \subseteq S$) — допуско́вих станів (вони ж *допускаючі*; рос. «*допускающие* состояния», англ. «*accept* states»).

Автомат-розпізнавач *допускає* (або «*приймає*», «*розпізнає*»; рос. «*допускает*», «*принимает*», «*распознаёт*»; англ. «*accepts*», «*recognizes*») слово, коли після прочитання його (сло́ва) останнього символу переходить у стан, що належить множині допускових станів F. Якщо ж автомат після прочитання останнього символу сло́ва перейшов у не допусковий стан — він не допускає (*відхиляє*; рос. «*отклоняет*»; англ. «*declines*») це слово. Особливим випадком цього ж правила є допусковість чи не допусковість початкового стану s_{start}: коли автомат на початку робо́ти ще не прочитав жодного символу, він вже обробив порожнє слово ε. Так що, якщо початковий стан допусковий ($s_{start} \in F$), то автомат допускає ε, інакше ($s_{start} \in (S \backslash F)$) відхиляє.

Один з двох вердиктів (або допуск, або відхилення) робиться і перед першим вхідним символом, і після абсолютно кожного подальшого вхідного символу. Наприклад, нормальною є ситуація, коли автомат, читаючи слово `abbaaabab`, видає «допуск» перед 1-им та після 4-го, 6-го й 7-го символів (на слова́ ε, `abba`, `abbaaa` і `abbaaab`) та відхилення в усі інші моменти (на слова́ `a`, `ab`, `abb`, `abbaa`, `abbaaaba`, `abbaaabab`).

Перехід автомата-розпізнавача до допускового стану ***не*** *означає* завершення роботи: автомат просто видає вердикт про допуск поточного сло́ва, й продовжує чекати на наступний вхідний символ, щоб після нього так са́мо перейти у новий стан (чи лишитися у старому) згідно з функцією переходів і видати новий вердикт. Са́ме тому́ в цьому посібнику надана перевага терміну «допусковий», хоча поширеними є також варіанти «*фінальний*» або «*заключний*» стан (рос. «*финальное*», «*заключительное*» состояние, англ. «*final* state»). Навіщо називати стан, який не передбачає нічого схожого на завершення робо́ти, «фінальним» чи «заключним» — питання філософське, яке мусить бути адресоване не автору цього посібника, а прихильникам тієї термінології.

Автомат *розпізнає формальну мову* L, якщо він допускає усі слова́, які належать цій мові, й не допускає жодного сло́ва, яке не належить L. Звідси й походить назва «розпізнавач»; щодо мов, на відміну від слів, вживають лише термін «розпізнає мову» (рос. «*распознаёт* язык», англ. «*recognizes* language»), а не «приймає» чи «допускає».

Очевидно, що далеко не для кожної формальної мови можна побудувати автомат-розпізнавач. Зокрема, формальними мовами є множи́ни всіх можливих синтаксично правильних програм реальними мовами програмування високого рівня. Такі мови, хоча й простіші за природні, все ж занадто складні для такого простого об'єкта, як автомат. Більш того, у розд. 6.5.3 наведено приклади порівняно простих формальних мов, які не можна розпізнати за допомогою автоматів[51].

Щоб автомат-розпізнавач був осмислений, він має деякі слова́ допускати, а деякі (решту) відхиляти. Тому́ потрібно $F \neq \varnothing$ і $S \setminus F \neq \varnothing$ (щоб серед станів були і допускові, і не допускові)[52].

Табличний спосіб подання — задається таблиця, рядки якої відповідають символам вхідного алфавіту, стовпчики — станам, і в кожній комірці вказується, в який стан має перейти автомат, якщо, перебуваючи у відповідному стані, прочитав відповідний вхідний символ. До речі, коли задана така таблиця, самі собою виявляються заданими і вхідний алфавіт X (позначення рядків), і множина станів S (позначення стовпчиків).

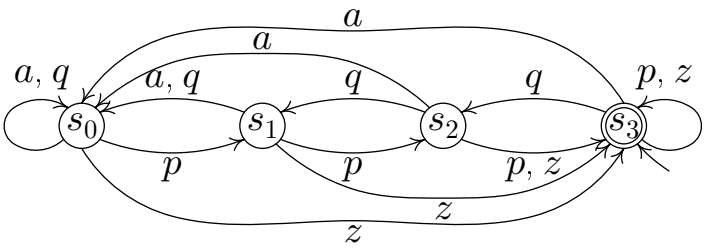
Множину допускових станів та початковий стан треба задавати окремо. Ми будемо зображати під функцією переходів ще один рядок, в якому писати для кожного стану або "доп." чи "Д", якщо стан допусковий, або "не доп." чи "н/д", якщо не допусковий. Наприклад:

[51] точніше, за допомогою *скінче́нних* автоматів, у яких скінче́нні всі множи́ни, якими він задається

[52] неповні (стор. 173) та недетерміновані (розд. 6.5.4–6.5.7) автомати можуть відхиляти слова́ й іншими, ніж явні не допускові стани, засобами, тому на них це зауваження не поширюється

Початковий стан: s_3.

	s_0	s_1	s_2	s_3
a	s_0	s_0	s_0	s_0
p	s_1	s_2	s_3	s_3
q	s_0	s_0	s_1	s_2
z	s_3	s_3	s_3	s_3
	н/д	н/д	н/д	доп.



Граф переходів зображено праворуч від таблиці переходів. Слово «граф» тут слід сприймати у старому смислі (сукупність вершин, деякі з яких з'єднані лініями), а не згідно сучасного означення. Стани автомата зображають кружéчками, причому не допускові — одинарними, допускові — подвійними. Початковий стан ініціального автомата виділяють окремою стрілочкою, що заходить у початковий стан нізвідки.

Дýгами (орієнтованими ребрами) графа є переходи (початок дуги́ — старий стан, кінець — новий стан; якщо новий стан дорівнює старому, дуга виявляється петлею). Кожна така дуга повинна бути підписана символом вхідного алфавіту, який вказує, коли переходити по цій дузі (лише якщо перебуваємо у тому стані, де дуга починається, і прочитали зі входу якраз той символ, яким підписана дуга). Конкретно у випадку автоматів-розпізнавачів не малюють по багато дуг з однаковими як початком, так і кінцем, пишучи натомість по кілька символів вхідного алфавіту на одній дузі (дугою слід користуватися при прочитанні будь-якого з цих символів).

Повторюємо: щойно наведено два різні способи подання (табличний та графічний) *одного й того са́мого* автомата.

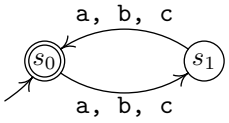
Приклад застосування автомата. Розглянемо детально процес застосування вищезгаданого автомата до вхідного сло́ва *qqazzappp*.

вхідний символ	стан до переходу	стан після переходу	поточний вердикт
Перед початком роботи автомата, s_3 доп. $\Rightarrow$ порожнє слово допущене			
q	s_3	$f(s_3,q)=s_2$	s_2 н/д $\Rightarrow$ слово "q" відхилене
q	s_2	$f(s_2,q)=s_1$	s_1 н/д $\Rightarrow$ слово "qq" відхилене
a	s_1	$f(s_1,a)=s_0$	s_0 н/д $\Rightarrow$ слово "qqa" відхилене
z	s_0	$f(s_0,z)=s_3$	s_3 доп. $\Rightarrow$ слово "$qqaz$" допущене
z	s_3	$f(s_3,z)=s_3$	s_3 доп. $\Rightarrow$ слово "$qqazz$" допущене
a	s_3	$f(s_3,a)=s_0$	s_0 н/д $\Rightarrow$ слово "$qqazza$" відхилене
p	s_0	$f(s_0,p)=s_1$	s_1 н/д $\Rightarrow$ слово "$qqazzap$" відхилене
p	s_1	$f(s_1,p)=s_2$	s_2 н/д $\Rightarrow$ слово "$qqazzapp$" відхилене
p	s_2	$f(s_2,p)=s_3$	s_3 доп. $\Rightarrow$ слово "$qqazzappp$" допущене

Приклад побудови 1. Побудуємо автомат, який допускатиме слова́ в алфавіті $\{a,b,c\}$ па́рної довжи́ни (порожнє слово теж вважатимемо словом парної довжини́, бо 0 — парне число). Виходить дуже простий автомат, з двома станами s_0 (досі оброблена частина сло́ва має парну довжину) та s_1 (оброблена частина має непарну довжину); кожен введений символ завжди переводить автомат до іншого стану.

Початковий стан: s_0.

	s_0	s_1
a	s_1	s_0
b	s_1	s_0
c	s_1	s_0
	доп.	н/д



Приклад побудови 2. Нехай потрібно розпізнавати слова́ в алфавіті $\{0,1\}$, куди нулі входять парну кількість разів, а одиниці — непарну (1, 001, 010, 100, 111, 00001, 00010, 00100, 00111, 01000, 01011, 01101, 01110, 10000, 10011, 10101, 10110, 11001, 11010, 11100, 11111, ...).

Вибір множи́ни станів S — процес творчий (він же погано алгоритмізований); часто він і є найскладнішим етапом побудови автомата, і нерідко вибрану множину станів доводиться «поправляти» у процесі побудови автомата. Але як правило все ж варто записати хоча б деяке «наближення» S, описавши кожен стан словами (*що означає* перебування у цьому стані).

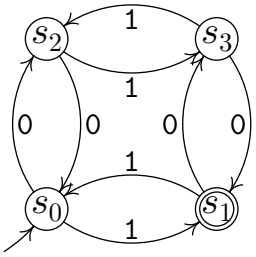
В цій задачі потрібно вміти розрізняти чотири ситуації:

1. парна кількість нулів та парна кількість одиниць;
2. парна кількість нулів та непарна кількість одиниць;
3. непарна кількість нулів та парна кількість одиниць;
4. непарна кількість нулів та непарна кількість одиниць.

Перед початком роботи автомата маємо першу ситуацію; це і буде початковий стан s_0. Решту станів (в порядку переліку) позначимо як s_1, s_2 та s_3. Очевидно, що допусковим має бути лише стан s_1. Переходи теж будуються досить очевидно: обробка символу "0" перемикає автомат між станами s_0 та s_2 або між s_1 та s_3; символу "1" — між станами s_0 та s_1 або між s_2 та s_3.

Початковий стан: s_0.

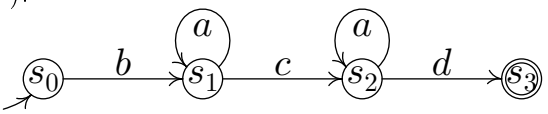
	s_0	s_1	s_2	s_3
0	s_2	s_3	s_0	s_1
1	s_1	s_0	s_3	s_2
	не доп.	доп.	не доп.	не доп.



Цей автомат має особливу структуру, яку побачити у графовому поданні: «верхня й нижня половини» приблизно однакові (крім початкового стану та допускового стану), всі переходи між цими «половини» лише за символом 0; аналогічно з «лівою та правою половинами» і символом 1. Це не випадково, а тому́, що задача схожа на те, ніби треба двічі розв'язати попередню задачу: один раз лише для вхідних символів 0, інший — окремо, «в іншому вимірі» та замінивши парну кількість на непарну, лише для вхідних символів 1, і з'єднати результати.

Для такої ситуації (поєднання роботи двох автоматів шляхом того, що утворюється новий автомат із станами, відповідними усім можливим парам станів тих двох автоматів) навіть є стандартна назва: *добуток автоматів* (рос. «*произведение автоматов*», англ. «*automata product*»).

Приклад побудови 3. Нехай потрібно розпізнавати регулярну мову, задану regex-ом ba^*ca^*d (ітерація в обох випадках стосується лише символу "a").

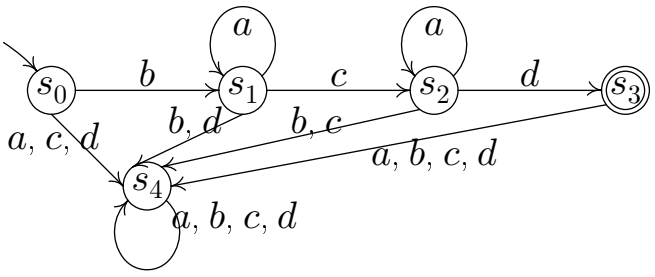


В цьому випадку, стани мають відповідати «мірі просування» по зразку: важливо розрізнити, чи вдалося знайти перший символ b; символи a (яких може бути скільки завгодно, в т. ч. 0 штук), залишають автомат у тому ж стані; потім важливо розрізнити, чи вдалося знайти символ c, і т. д.

Остаточно, приходимо до автомата, зображеного праворуч. Строго кажучи, він не задовольняє раніше даному означенню: вхідний алфавіт автомата очевидно $\{a, b, c, d\}$, а у гра́фі не вказується, куди переходити, якщо, наприклад, поточний стан s_0 і на вхід подається d.

Але це не помилка, а зображення автомата-розпізнавача у т. зв. *неповному* вигляді (рос. «автомат в *неполном виде*», англ. «*partial* automaton»): якщо з деякої вершини не вказано перехід по деякому символу, це означає, що після прочитання такого символу в такому стані автомат гарантовано ніколи не потрапить до допускового стану (зокрема, слово, що починається на "d", гарантовано не може бути описане regex-ом ba^*ca^*d).

Щоб подати такий автомат у повному вигляді (рос. «представить в *полном* виде», англ. «represent as *total* automaton»), вводять ще один не допусковий стан — «чорну діру», куди можна зайти, але звідки не можна вийти, і спрямовують всі відсутні переходи туди.



Вживання фрази «чорна діра» (рос. «чёрная дыра», англ. «black hole») у смислі не допускового стану, всі переходи якого напрямлені у себе, має певне поширення, але не дуже велике. Ще одна, теж не дуже поширена, назва того ж поняття — «диявольський стан» (рос. «дьявольское состояние», англ. «diablo state»).

6.5.2 Основна теорема теорії скінченних автоматів

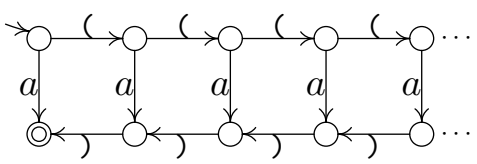
Клас регулярних мов дорівнює класу мов, які можна розпізнати за допомогою скінче́нних автоматів-розпізнавачів.

Цю теорему називають також *теоремою Кліні* (Кліні — той самий, на честь кого ітерацію називають також «зірочкою Кліні»). Поки що залишимо цю теорему без доведення; потім (у розд. 6.5.6) розглянемо алгоритм побудови (недетермінованого) автомата-розпізнавача за довільним regex-ом — і цим проведемо конструктивне доведення теореми в один бік.

У теоремі Кліні маються на увазі «математичні» регулярні вирази. Для значної частини програмних реалізацій regex-ів ця рівність класів мов порушується. Як у бік, що програмні реалізації можуть розпізнавати трохи «складніші» мови, які «не по силам» «математичним» регулярним виразам та автоматам, так і навпаки, коли програмні реалізації накладають свої обмеження (максимальної довжини́, максимальної вкладеності, тощо).

6.5.3 Лема про накачку та доведення неавтоматності деяких мов

Чи можна побудувати скінче́нний автомат, що розпізнаватиме правильні дужкові вирази? Виявляється, ні. Більш того: виявляється, що скінче́нний автомат-розпізнавач не може розпізнати навіть таку (простішу) формальну мову, як $\{({}^k a)^k \mid k{\in}\mathbb{Z}^+\} = \{a, (a), ((a)), (((a))), ((((a)))), \ldots\}$, тобто множину всіх слів, де спочатку йдуть відкривні дужки, потім буква a, потім закривні дужки — *рівно стільки ж*, скільки раніше було відкривних.



Інтуїтивно неможливість побудувати скінче́нний автомат-розпізнавач цієї мови (це й називають *неавтоматністю* мови) можна аргументувати тим, що очевидний автомат має вигляд, зображений праворуч — отже, щоб обробити 3-кратну вкладеність дужок, потрібно 8 станів; 20-кратну — 42 стани; довільну кратність — нескінче́нну (злічéнну) кількість станів.

Але це не доведення: а раптом якимось іншим способом побудувати автомат все-таки можна? Тож почнемо строге формальне доведення неавтоматності мови $\{({}^k a)^k \mid k{\in}\mathbb{Z}^+\}$.

Припустимо, ніби вдалося побудувати скінче́нний автомат, який розпізнає цю мову (при довільних вкладеностях дужок). Значить, зафіксована деяка скінче́нна множина станів цього автомата. Позначимо кількість станів як n і розглянемо вхідний рядок вигляду

$$\underbrace{(((\ldots((}_{n+1\text{ штук}}\ a\ \underbrace{))\ldots)))}_{n+1\text{ штук}}. \tag{78}$$

У процесі обробки "$(((\ldots((}$" автомат мусить (щонайменше раз) перейти до стану, в якому вже побував раніше — по тій причині, що кількість станів n, а кількість дужок $n{+}1$. Позначимо стан, що повторюється, як $\tilde{s}$, номер дужки, перед обробкою якої автомат вперше перебував у стані $\tilde{s}$, як m_1, номер дужки, перед обробкою якої автомат знов перейшов у стан $\tilde{s}$, як m_2.

За означенням, «поведінка» скінче́нного автомата залежить *лише* від стану та від вхідного символу. Значить, автомат ніяк не може розрізнити, чи він обробляє m_1-й символ, чи m_2-й. Значить, підслово "$((\ldots($" довжини́ $m_2{-}m_1$, яке займало позиції з m_1-ї включно по m_2-у не включно, «не лишає по собі ніякого сліду», і його можна «вирізати», отримавши слово

$$\underbrace{(\,(\,(\ \ldots\ (\,(}_{\substack{n+1-\\ -(m_2-m_1)\text{ шт.}}}\ a\ \underbrace{)\,)\ldots)\,)\,)}_{n+1\text{ штук}},$$

або, навпаки, повторювати кілька разів, отримуючи слова́

$$\underbrace{(\,(\,(\ \ldots\ (\,(}_{\substack{n+1+\\ +(m_2-m_1)\text{ шт.}}}\ a\ \underbrace{)\,)\ldots)\,)\,)}_{n+1\text{ штук}},\qquad \underbrace{(\,(\,(\ \ldots\ (\,(}_{\substack{n+1+\\ +2\cdot(m_2-m_1)\text{ шт.}}}\ a\ \underbrace{)\,)\ldots)\,)\,)}_{n+1\text{ штук}},\quad \ldots$$

І автомат сприйматиме всі ці слова́ *так само*, як слово (78). Хоча їх треба сприймати по-різному, бо слово (78) належить мові $\{({}^k a)^k \mid k{\in}\mathbb{Z}^+\}$, а інші наведені не належать. Що й доводить неавтоматність цієї мови (неможливість розпізнавання її скінче́нним автоматом).

Проведене доведення неавтоматності мови $\{({}^k a)^k \mid k{\in}\mathbb{Z}^+\}$ *зовсім не* унікальне, а являє собою частковий випадок ідеї, яку називають «*лема про накачку*» (рос. «*лемма о накачке*», англ. «*the pumping lemma*»). Від наведення математично строгого формулювання цієї леми утримаємось, бо воно досить складне́ і при цьому не дає нічого принципово нового. А стандартні етапи доведень, що проводяться згідно з лемою про накачку, повторимо ще раз:

1. припустивши, що мова автоматна, підібрати таке слово α, щоб при його обробці автомат мусив неоднократно пройти через деякий стан;
2. подати це слово як $\alpha = uvw$ (де $v{\neq}\varepsilon$ — підслово, перед обробкою та після обробки якого автомат потрапляє в один і той самий стан);
3. якщо серед слів uw, uvw, $uvvw$, $uvvvw$, … (котрі обробляються автоматом однаково) вдається знайти такі, які мають оброблятися по-різному, то цим доводиться неавтоматність досліджуваної мови.

Звичайно, тут описана лише загальна схема, а конкретний зміст кроків (як підбирати $\alpha = uvw$, як аналізувати слова́ uw, uvw, $uvvw$, …) потрібно робити для кожної досліджуваної мови окремо.

Наголосимо, що такі доведення стверджують *тільки* те, що задача розпізнавання дослідженої мови не може бути *повністю* розв'язана використанням *лише скінче́нних* автоматів-розпізнавачів. Але ніщо не забороняє будувати автомати, що розв'язують цю задачу при додаткових обмеженнях (наприклад, якщо вкладеність дужок гарантовано не перевищуватиме 5, можна побудувати автомат з дванадцятьма станами). Так само ніщо не забороняє розпізнавати слова́ вказаної мови за допомогою інших засобів — автоматів з лічильниками, нескінче́нних автоматів, магазинних автоматів, машин Тьюрінга, «звичайних» мов програмування (як-то C++), тощо.

Враховуючи задекларовану в розд. 6.5.2 еквівалентність автоматів-розпізнавачів та регулярних виразів, логічно очікувати, що якщо для деякої формальної мови вдалося застосувати лему про накачку і довести неможливість автомата, то цим доведено і неможливість регулярного виразу. І це справді так, коли мова йде про *«математичні»* регулярні вирази.

Коли ж ідеться про програмні реалізації regex-ів, ситуація ускладнюється. Деякі засоби програмних реалізацій regex-ів, (зокрема, можливість запам'ятовувати фрагменти і підставляти їх у рядку пошуку; див. про `\1`, `\2`, … на стор. 168) іноді роблять можливими regex-и для програмних реалізацій там, де для «математичних» регулярних виразів це неможливо.

Але са́ме іноді. Наскільки відомо автору цього посібника, переважна більшість програмних реалізацій regex-ів все ж не можуть правильно задати $({}^k a)^k$.

Хоча є деякі мови програмування (зокрема, Perl), в яких дозволено оголошувати regex-ові функції і робити їх рекурсивними. Але тут уже виникає питання, чи варто продовжувати називати «регулярними виразами» те, що аж настільки далеко відійшло від початкової ідеї «константні слова́, конкатенація, альтернатива, ітерація — і все».

6.5.4 Недетерміновані переходи

Слово «детермінований» буквально перекладається «визначений». Усі автомати, які ми розглядали досі, були детермінованими. Конкретніше, детермінованість проявляється у тому, що у кожен момент автомат перебуває в одному, чітко визначеному, стані; коли на вхід подається символ, функція переходів чітко й однозначно вказує новий стан, куди має перейти автомат.

Для *недетермінованих* (рос. «*недетерминированных*», англ. «*nondeterministic*») автоматів це не так. Є різні формальні означення:

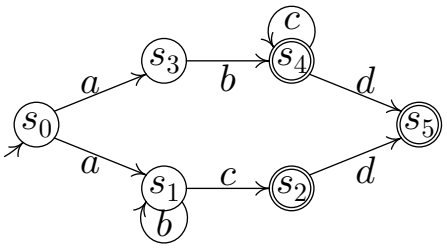
- у недетермінованих автоматах замість функції переходів $f : S \times X \to S$ задається відношення переходів, що є підмножиною $S \times X \times S$;
- у недетермінованих автоматах функція переходів багатозначна;
- у недетермінованих автоматах функція переходів замість вигляду $f : S \times X \to S$ має вигляд $f : S \times X \to 2^S$ (де 2^S — булеа́н S, див. стор. 49).

(Решта означення, де йдеться про S, s_{start}, X та F, лишається такою ж, як на початку розд. 6.5.1; частина, що стосується f, змінюється одним із щойно згаданих способів.)

Суть цих модифікацій одна: дозволяються[53] різні переходи з одного й того ж стану при прочитанні одного й того ж вхідного символу. *Недетермінований автомат* ***не «вибирає» один*** *із цих переходів (випадково чи ще якось),* ***а*** *використовує* ***одночасно всі*** *такі переходи.* Як наслідок, недетермінований автомат може перебувати одночасно у кількох станах. Тому́ замість поточного стану говорять про *поточну множину станів* (рос. «*текущее множество состояний*», англ. «*current set of states*»).

Недетермінований автомат-розпізнавач допускає слово тоді й тільки тоді, коли після прочитання останнього символу цього сло́ва поточна множина станів ***містить хоча б один допусковий***. Чи перебуває він також і в не допускових, не впливає на результат.

Приклад. Застосуємо ініціальний неповний недетермінований автомат-розпізнавач (див. граф переходів трохи нижче) до сло́ва *abccdd*.



1) Прочитавши a, автомат переходить з початкового стану s_0 до двох станів s_1 та s_3. Жоден з них не допусковий — слово a відхилене.

2) Прочитавши b, автомат переходить (з s_1 у s_1) та (з s_3 у s_4); нова поточна множина станів $\{s_1, s_4\}$; один з них (s_4) допусковий — слово ab допущене.

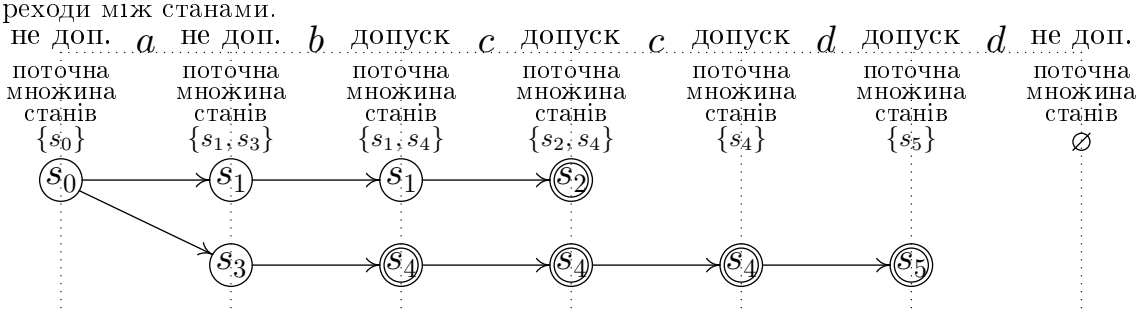
3) Прочитавши c, автомат переходить (з s_1 у s_2) та (з s_4 у s_4); нова поточна множина станів $\{s_2, s_4\}$; у цій множині є допусковий (навіть два, але це не суттєво) — слово abc допущене.

4) Прочитавши дру́гу c, з s_2 переходити нема куди — отже, ця «гілка» «обривається»; з s_4 можна перейти у s_4; поточна множина станів $\{s_4\}$; s_4 допусковий — слово $abcc$ допущене.

5) Прочитавши d, можна перейти з s_4 у s_5; поточна множина станів $\{s_5\}$; s_5 допусковий — слово $abccd$ допущене.

6) Прочитавши дру́гу d, переходити нема куди, множина поточних станів стає $\varnothing$, тож, не містячи ніяких станів, не містить допускових — слово $abccdd$ відхилене. Так са́мо буде відхилене будь-яке продовження сло́ва $abccdd$.

Зобразимо те са́мо графічно. Вертикальні «зрізи» містять поточні множи́ни станів; позначки згори вказують, допущене чи відхилене слово; горизонтальні та похилі стрілочки позначають переходи між станами.



6.5.5 ε-переходи

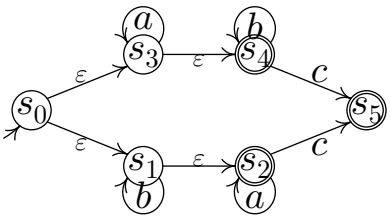
ε-перехід — це перехід, що відбувається *не* внаслідок обробки прочитаного символу, а внаслідок обробки порожнього сло́ва ε.

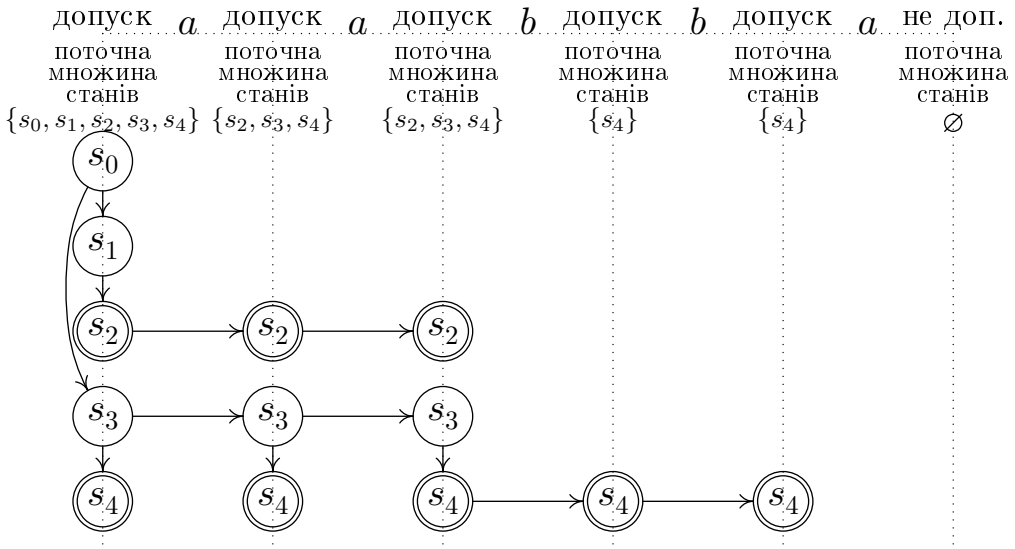
Порожнє слово можна знайти де завгодно (перед будь-яким символом, між будь-якими сусідніми символами). Тому ε-переходи вводять *лише* для недетермінованих автоматів. Коли автомат, що містить перехід $s_j \xrightarrow{\varepsilon} s_k$, потрапляє у s_j, він одночасно і лишається у s_j, і переходить у s_k (бо використовує одночасно всі можливі переходи); якби з s_k виходили ще інші ε-переходи, вони теж були б задіяні. Бо між будь-якими символами можна знайти не лише одне ε, а скільки завгодно.

Надалі будемо користуватися скороченнями "НСА" та "ε-НСА". "НСА" означає «недетермінований скінче́нний автомат-розпізнавач (без ε-переходів)»; "ε-НСА" означає «недетермінований скінче́нний автомат-розпізнавач, що може містити ε-переходи» (а може і не містити; отже, НСА є частковим випадком ε-НСА). Російською, ці абревіатури мають вигляд "НКА" та "ε-НКА" (від «недетерминированный конечный автомат»); англійською — "NFA" (або "NFSM") та "ε-NFA" (або "ε-NFSM") (від «nondeterministic finite automaton» або «nondeterministic finite state machine»).

[53] але не вимагаються; тому́, «звичайні» детерміновані автомати виявляються частковим випадком недетермінованих (аналогічно тому, як натуральні чи́сла виявляються частковим випадком дійсних)

Приклад. Побудуємо схему переходів (нижче) при застосуванні ε-НСА (праворуч) до вхідного слóва *aabba*. Тепер у схемі переходів, крім горизонтальних і похилих стрілочок (які, як і раніше, відповідають переходам при обробці вхідних символів) з'являються ще вертикальні, які відповідають ε-переходам.



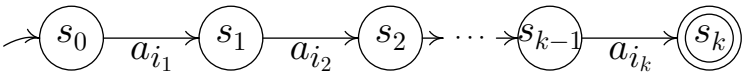


6.5.6 Побудова ε-НСА за (будь-яким) регулярним виразом

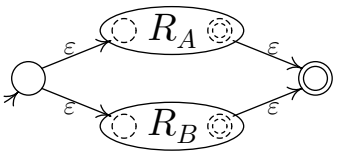
За означенням, регулярний вираз є або константним словом, або має структуру "$(R_A \mid R_B)$", або "$R_A \cdot R_B$", або "$(R_A)^*$". От і розглянемо ці чотири випадки, і побудуємо автомат-розпізнавач для кожного з них.

(При дослідженні випадків 2–3 доводиться вважати, що автомати для розпізнавання R_A та R_B (у випадку 4 — автомат для розпізнавання R_A) вже побудовані. Це стандартна практика доведень для рекурсивних означень: всере́дині R_A та R_B теж має місце один з тих самих чотирьох випадків, котрі кінець кінцем усі розглянуті. І така практика має багато спільного з тим, як у розд. 3.3 п. 3 спирається на те, що всі рекурсивні виклики поповертали правильні значення (хоча це все ще у процесі доведення), а також тим, як у розд. 3.1 крок «класичної» матіндукції спирається на правильність $P(k)$.)

1. Будь-яке константне слово $a_{i_1} a_{i_2} \dots a_{i_k}$ можна розпізнати автоматом

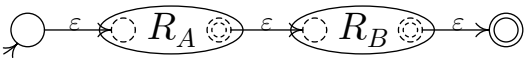


2. Якщо зовнішньою операцією regex-а є альтернатива, і автомати для мов, заданих виразами R_A та R_B, вже побудовані, то мову, задану виразом "$(R_A \mid R_B)$", можна розпізнати таким ε-НСА:

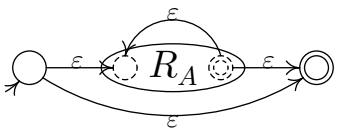


   Тобто, вводиться новий початковий стан, з нього пускаються ε-переходи у стани, які були початковими для автоматів, відповідних R_A та R_B (а після цієї дії перестають бути початковими). Аналогічно, вводиться новий допусковий стан, у нього пускаються ε-переходи зі станів, які були допусковими для автоматів, відповідних R_A та R_B (а після цієї дії перестають бути допусковими).

3. Якщо зовнішньою операцією regex-а є "$R_A \cdot R_B$", то задану ним мову можна розпізнати таким ε-НСА:

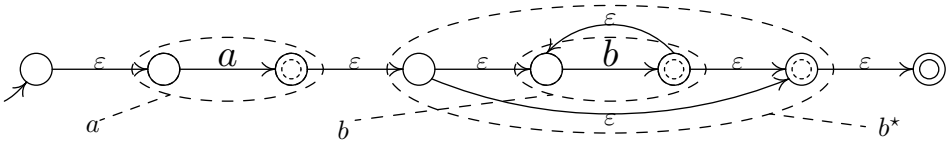


4. Якщо зовнішньою операцією regex-а є "$(R_A)^*$", то задану ним мову можна розпізнати таким ε-НСА:



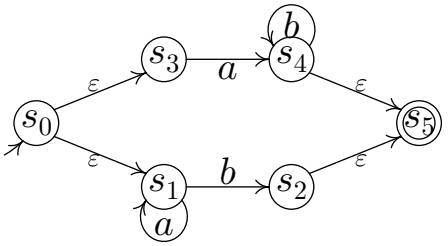
Побудуваний за цими правилами автомат може виявитися не найкращим.

Наприклад, мова, задана regex-ом "ab^*", може бути розпізнана дуже простим детермінованим автоматом, зображеним праворуч, тоді як буквальне дотримання наведених правил призводить до значно складнішого ε-НСА, наведеного нижче:



Але зате ці правила застосовні *завжди*, до будь-яких регулярних виразів. (А отже — являють собою обіцяне конструктивне доведення в один бік теореми Кліні з розд. 6.5.2.)

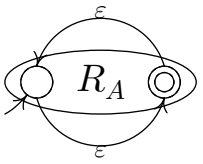
Скажімо, якщо треба побудувати автомат-розпізнавач мови, що задається виразом $(a^*b|ab^*)$, то «шматочки» для розпізнавання окремо a^*b, окремо ab^* будуються легко, а от «з'єднати» їх проблематично: у першому автоматі a призводить до повернення у початковий стан, у другому — до переходу в новий, і як це поєднати, геть не ясно. А застосування стандартного правила для $(R_A \mid R_B)$ легко призводить до результату (див. праворуч).



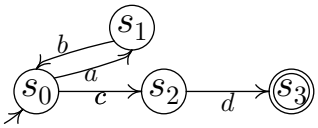
Головна причина і того, що розглянуті засоби працюють тоді, коли без них неясно, що робити, і того, що ці засоби продукують автомати з великою кількістю станів — ці засоби підтримують автомати у т. зв. *нормалізованій формі*, а са́ме: єдиний початковий стан, у який ніколи не повертаються з інших станів, та єдиний допусковий стан, із якого ніколи не переходять у інші стани.

Тобто, ε-переходи виявляються корисними головним чином можливістю та зручністю підтримання автомата у нормалізованій формі.

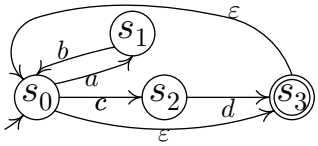
(При вивченні цієї теми багато в кого виникає думка, ніби правила можна змінити на простіші, які призводили б до автоматів з меншою кількістю станів. Наприклад, нерідко виникає пропозиція реалізовувати ітерацію просто за рахунок ε-переходів зі старту до фінішу та з фінішу до старту, як зображено праворуч.



Що ж, у значній частині випадків це правильно. Можна навіть довести, що при умові нормалізованості «вкладеного» автомата для R_A — завжди. Але якщо автомат для R_A не є нормалізованим, це може бути й неправдою:



Автомат не є нормалізованим, але правильно розпізнає мову, задану рег. виразом $(ab)^*cd$.



Автомат не розпізнає мову, задану $\big((ab)^*cd\big)^*$, бо допускає слово $abab$, яке не задається цим рег. виразом.

Томý традиційні правила й зроблені са́ме такими, як вказано у переліку вище, а до спроб їх «оптимізацій» слід ставитися дуже обережно.)

6.5.7 Детермінізація недетермінованих автоматів

З того, що у попередньому розділі не зуміли побудувати детермінований розпізнавач мови $(a^*b|ab^*)$, але зуміли побудувати ε-НСА, *не слід* робити висновок, ніби ε-НСА принципово потужніші за детерміновані. Насправді, не зуміли виключно з тієї причини, що погано старалися («геть не ясно, як» далеко не завжди означає «неможливо»), а класи автоматів еквівалентні.

Якщо відомий деякий скінче́нний ініціальний ε-НСА, то завжди можна побудувати детермінований скінче́нний ініціальний автомат-розпізнавач, що розпізнаєту саму мову.

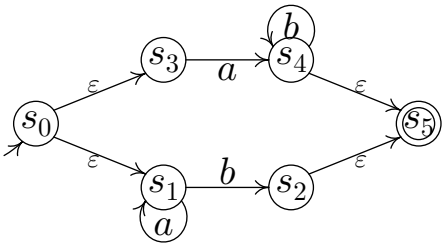
(Еквівалентність передбачає також симетричне твердження *«якщо відомий деякий скінче́нний ініціальний детермінований автомат-розпізнавач, то завжди можна побудувати скінче́нний ініціальний ε-НСА, що розпізнає ту саму мову»*. Але це твердження тривіальне, бо з усього вищезгаданого очевидно, що будь-який детермінований автомат відразу, без перетворень, є ε-НСА (просто са́ме він не містить ні ε-переходів,

ні різних переходів з одного стану по одному символу). А твердження з попереднього абзацу є неочевидною складовою еквівалентності. Тому́ приділимо увагу цьому неочевидному твердженню і запропонуємо алгоритм побудови детермінованого автомата, еквівалентного ε-НСА.)

Побудову детермінованого автомата, еквівалентного заданому ε-НСА, називають *детермінізацією*.[54] Основна ідея — вибрати «потрібні» *множи́ни* станів ε-НСА (які реально можуть бути поточними), і співставити їм стани нового детермінованого автомата. Як наслідок, кількість станів детермінованого автомата може виявитися значно більшою за кількість станів ε-НСА. У найгіршому випадку, 2^n (де n — кількість станів ε-НСА). Але на практиці так майже не трапляється; значно частіше, кількість станів нового детермінованого автомата приблизно така ж, як у початковому ε-НСА. Це тому́, що для переважної більшості ε-НСА лише мала́ частина з усіх можливих 2^n підмножин станів виявляється «потрібними» множинами.

Формально записати алгоритм детермінізації можна, наприклад, так: *«Виділити за початковий стан нового детермінованого автомата множину станів ε-НСА, куди можна дійти по ε-переходам з початкового стану. Далі виділяти всі можливі «потрібні» множи́ни станів ε-НСА, за допомогою деякого пошуку в графі, вершинами якого є множи́ни станів ε-НСА, ду́гами — можливості перейти від множи́н станів до множи́н станів, обробляючи один вхідний символ. Кожну нову таку множину станів співставляти новому стану детермінованого автомата, який будується, доки не будуть перебрані всі досяжні множи́ни станів ε-НСА.»*

Але цей запис все-таки не дуже чіткий та зрозумілий, тому детально розглянемо також приклад. Нехай потрібно детермінізувати наведений вище ε-НСА розпізнавання мови, що задається regex-ом $(ab^*\,|\,a^*b)$.



Перш за все, з'ясуємо початкову множину станів. Хоча формально початковим станом ε-НСА є лише s_0, але з урахуванням ε-переходів перед обробкою першого символу автомат перебуватиме у множині станів $\{s_0, s_1, s_3\}$. От і запам'ятаємо цю множину, відзначивши, що вона відповідатиме початковому стану q_0 детермінованого автомата.

В разі прочитання символу a цей ε-НСА піде (з s_1 у s_1) та (з s_3 у s_4), а з s_0 по символу a йти нема куди. Здавалося б, це означає, що автомат приходить у множину станів $\{s_1, s_4\}\ldots$ але треба ще врахувати ε-перехід (з s_4 у s_5). Остаточно, при прочитанні символу a у початковій множині станів $\{s_0, s_1, s_3\}$ цей ε-НСА потрапляє у множину станів $\{s_1, s_4, s_5\}$. Аналогічно, в разі прочитання b він потрапить у множину станів $\{s_2, s_5\}$. Запам'ятовуємо ці множи́ни, співставляючи їм «нові» стани q_1 та q_2 відповідно.

Тепер треба розглянути, куди далі може потрапити ε-НСА. Вже за́раз є дві не досліджені множини станів, а при їхньому розгляді можливі подальші розгалуження, тож, щоб нічого не загубити, варто скористатись якимсь пошуком в орграфі, де вершинами є множи́ни станів ε-НСА, дугами — можливості переходу з множи́н станів у множину станів (за один такт ε-НСА).

BFS передбачає, що треба пам'ятати чергу і стани вершин. Будемо підтримувати всю цю інформацію, але в іншому, ніж у розд. 5.6.1, форматі. Знайдені множи́ни станів ε-НСА все одно запам'ятовуються у переліку, що задає відповідність цих множин новим станам q_i. Цей са́мий перелік можна використати і в якості черги, і для з'ясування, які вершини–множи́ни вже зустрічалися, а які ще ні (що замінює стани; щоправда, замість прямого звернення до `st[next]` доводиться переглядати раніше запам'ятовані множи́ни й порівнювати кожну з поточною, це знижує ефективність). Ду́ги будемо обробляти відразу по мірі знаходження, ніяк не запам'ятовуючи.

Тепер дослідимо, куди можна перейти зі співставленої q_1 множи́н $\{s_1, s_4, s_5\}$. При обробці a потрапляємо у $\{s_1\}$, при обробці b — у $\{s_2, s_4, s_5\}$. Отримані множи́ни станів $\{s_1\}$ та $\{s_2, s_4, s_5\}$ є новими (са́ме як множи́ни), тому́ співставляються станам q_3 та q_4 відповідно, й додаються у чергу.

Ще не відомо, скільки буде станів у детермінованому автоматі. Але для частини станів уже відома остаточна таблиця переходів, зображена праворуч. Таблиця ще буде дописуватися (причому її стовпчики $q_2 = \{s_2,\ s_5\}$, $q_3 = \{s_1\}$ та

	$\{s_0,s_1,s_3\}$	$\{s_1,s_4,s_5\}$	$\{s_2,s_5\}$	$\{s_1\}$	$\{s_2,s_4,s_5\}\ldots$
	q_0	q_1	q_2	q_3	q_4	$\ldots$
a	q_1	q_3	???	???	???	$\ldots$
b	q_2	q_4	???	???	???	$\ldots$

[54] укр. та рос. мовами — «детермінізація» («детерминизация»), що наголошує на меті; англійською — «powerset construction», що наголошує на використаному способі

$q_4 = \{s_2,\ s_4,\ s_5\}$ являють собою чергу пошуку вшир), але *вже точно відомо*, який вигляд (зокрема, переходи для q_0 та q_1) має її початок.

Далі виймаємо з черги $q_2 = \{s_2, s_5\}$. Ні з s_2, ні з s_5, ні по a, ні по b нікуди потрапити не можна. Значить, стани нового автомата поки що без змін, а у відповідні клітинки таблиці переходів пишемо "$-$".

Далі виймаємо з черги $\{s_1\}$. По вхідному символу a потрапляємо знов у ту саму $\{s_1\}$, по b — у (теж вже досліджену) $\{s_2, s_5\}$. Жодної нової множини́ станів не отримали, значить додавати до переліку нічого не треба.

Виймаємо з голови черги $\{s_2, s_4, s_5\}$, по a отримуємо $\varnothing$, по b нову множину $q_5 = \{s_4, s_5\}$ (записуємо в кінець черги і тут же виймаємо).

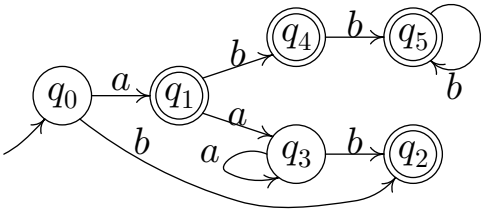
З $\{s_4, s_5\}$ приходимо лише в $\varnothing$ та $\{s_4, s_5\}$ — нічого нового.

Всі підмножи́ни досліджені, отже черга порожня і пошук завершено: ми дослідили всі досяжні множи́ни станів заданого ε-НСА.

Перепишемо таблицю переходів «начисто» і визначимо допускові стани (згідно стандартного правила: множина станів недетермінованого автомата допускова тоді й тільки тоді, коли містить хоча б один допусковий стан).

	$\{s_0,s_1,s_3\}$	$\{s_1,s_4,s_5\}$	$\{s_2,s_5\}$	$\{s_1\}$	$\{s_2,s_4,s_5\}$	$\{s_4,s_5\}$
	q_0	q_1	q_2	q_3	q_4	q_5
a	q_1	q_3	$-$	q_3	$-$	$-$
b	q_2	q_4	$-$	q_2	q_5	q_5
	н/д	доп.	доп.	н/д	доп.	доп.

Початковий стан: q_0.



(«Заднім числом», маючи цей автомат перед очима, можна і зрозуміти його логіку, і навіть знайти неоптимальність.

Логіка така: коли вже знайдено початок aa, це може відповідати виразу $(ab^* \,|\, a^*b)$ лише через відповідність частині a^*b, стани q_0, q_1, q_3, q_2 якраз правильно розпізнають ті зі слів, що починаються на aa й далі відповідають a^*b; якщо ж знайдено початок abb, то це може відповідати виразу $(ab^* \,|\, a^*b)$ лише через відповідність частині ab^*, і стани q_0, q_1, q_4, q_5 якраз правильно розпізнають ті зі слів, що починаються на abb й далі відповідають ab^*; мова, задана виразом $(ab^* \,|\, a^*b)$, містить лише три сло́ва, які не містять на початку ні aa, ні abb: це слова́ a, b, ab, вони теж допускаються.

Неоптимальність полягає в тому, що стан q_4 насправді не розрізняє ніякої ситуації, яка була б істотно відмінна від ситуації стану q_5: що там, що там слово можна продовжувати лише символами b у довільній кількості. Тому́ q_4 та q_5 насправді можна і треба з'єднати в один стан (див. також розд. 6.7).

Але це не скасовує того факту, що придумати такий автомат відразу в детермінованому вигляді досить важко, тоді як і побудова для того ж рег. виразу ε-НСА, і подальша детермінізація потребу́ють просто акуратно застосувати стандартні правила.)

6.6 Автомати-перетворювачі

Автомат-*перетворювач* (рос. «автомат-*преобразователь*», англ. «finite-state *transducer*») на кожному такті, прочитавши один вхідний символ, видає один вихідний символ (з *вихідного алфавіту*, рос. «*выходной алфавит*», англ. «*output alphabet*»). Вихідний алфавіт зазвичай не дорівнює вхідному (хоча може й дорівнювати).

Наприклад, у розд. 6.4 згадувалося, що пристрій, який керує ліфтом, можна описати абстрактним автоматом. Це точно не автомат-розпізнавач, бо керування ліфтом не має нічого спільного з допуском/відхиленням слів. А на автомат-перетворювач це більш-менш схоже (хоча теж не описується в усіх деталях тими найпростішими перетворювачами, які ми розглянемо).

Вхідний алфавіт такого автомата містить символи *«натиснена кнопка ззовні ліфта на ...-му поверсі»* (для кожного поверху свій символ), *«у ліфті натиснена кнопка ...-го поверху»* (теж для кожного поверху), тощо. (Насправді у вхідний алфавіт треба додати ще багато чого, зокрема покази датчиків, завдяки яким керувальний пристрій «взнає», чи вдалося зачинити двері, «взнає», що кабіна ліфта прибула на ...-й поверх, тощо.)

Вихідним алфавітом цього автомата будуть *«увімкнути мотор для відчинення дверей»*, *«увімкнути мотор для зачинення дверей»*, *«увімкнути мотор для руху кабіни вгору»*, *«увімкнути мотор для руху кабіни вниз»*, тощо.

Можливі й простіші для математичного розгляду ситуації, як-то «обидва алфавіти — десяткові цифри», «вхідний алфавіт — цифри, вихідний — латинські букви», тощо.

Вихідне слово означається аналогічно вхідному, як повна, від самого початку, послідовність вихідних символів. Ми вже з'ясували, що для будь-якого автомата вхідне слово визначає послідовність станів; якщо задані і послідовність вхідних символів, і послідовність станів, однозначно отримується також і послідовність вихідних символів (вихідне слово).

Отже, *для будь-яких ініціальних автоматів-перетворювачів вихідне слово повністю визначається вхідним словом.*

Ми розглянемо лише дві прості моделі автоматів-перетворювачів — автомати Му́ра та автомати Мі́лі — для яких *не* вводять поняття допускових і недопускових станів; вихідні символи та утворені з них вихідні слова́ є єдиним результатом їхньої робо́ти, *замість* допусків/відхилень. Відповідно, в рамках цього розд. 6.6 втрачають смисл раніше введені поняття неповного автомата, «чорної діри», а також недетермінізм: усі вони істотно спиралися на допускові/недопускові ста́ни. Також втрачає смисл поняття ε-переходу, бо воно спиралось на недетермінізм.

6.6.1 Автомати Му́ра

Автомат Му́ра (рос. «*автомат Му́ра*», англ. типово «*Moore machine*», значно рідше «*Moore automaton*») — це автомат-перетворювач, що задається п'ятіркою $\langle S, X, Y, f, g\rangle$ або шісткою $\langle S, s_{start}, X, Y, f, g\rangle$, де

- S — множина станів;
- s_{start} (якщо є) — початковий стан, один з елементів S;
- X — вхідний алфавіт;
- Y — вихідний алфавіт;
- $f : S \times X \to S$ — функція переходів;
- $g : S \to Y$ — функція виходів.

Суть множини́ станів, вхідного алфавіту та функції переходів можна перенести з (детермінованих) автоматів-розпізнавачів. Суть вихідного алфавіту обговорено на початку розд. 6.6.

Функція виходів (рос. «*функция выходов*», англ. «*output function*») визначає, який са́ме символ вихідного алфавіту виводиться (подається на вихід) на поточному кроці. Для автоматів Мура — залежно від стану, в який щойно перейшов автомат. Аналогічно тому, як автомат-розпізнавач видає допуск/відхилення залежно від стану, в який щойно перейшов, тільки результатом є не допуск і не відхилення, а символ з вихідного алфавіту.

Ще одна відмінність — автомат-розпізнавач виносить перший вердикт допуск/відхилення на порожнє слово, ще нічого не прочитавши. Автомат Мура так не робить (перший вихідний символ — лише після того, як прочитав вхідний символ і перейшов до чергового стану).

Приклад. Нехай треба читати (поцифрово, зліва направо) десятковий запис числа́ й після кожної прочитаної цифри виводити залишок від ділення прочитаної частини числа́ на 3. Наприклад, вхідному слову `7685` має відповідати вихідне слово `1102` (бо $7\%3=1$, $76\%3=1$, $768\%3=0$, $7685\%3=2$).

Почнемо з алфавітів: вхідний $X=\{$`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`$\}$, вихідний $Y=\{$`0`, `1`, `2`$\}$.

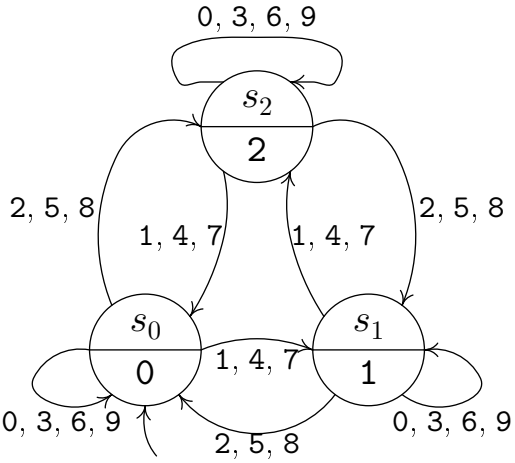
Спробуємо взяти три стани:

1. стан s_0 — сума вже оброблених цифр дає залишок 0 при діленні на 3; якраз таку ситуацію маємо, коли ще жодна цифра не оброблена — отже, цей стан слід взяти за початковий;
2. стан s_1 — сума вже оброблених цифр дає залишок 1 при діленні на 3;
3. стан s_2 — сума вже оброблених цифр дає залишок 2 при діленні на 3.

Враховуючи, що залишок від ділення десяткового числа́ на 3 дорівнює залишку від ділення на 3 суми його цифр, читання символу "`0`", "`3`", "`6`" або "`9`" не змінює стану; символу "`1`", "`4`" або "`7`" — замінює (s_0 на s_1), (s_1 на s_2), (s_2 на s_0); символу "`2`", "`5`" або "`8`" — замінює (s_0 на s_2), (s_1 на s_0), (s_2 на s_1).

Початковий стан: s_0.

	s_0	s_1	s_2
0	s_0	s_1	s_2
1	s_1	s_2	s_0
2	s_2	s_0	s_1
3	s_0	s_1	s_2
4	s_1	s_2	s_0
5	s_2	s_0	s_1
6	s_0	s_1	s_2
7	s_1	s_2	s_0
8	s_2	s_0	s_1
9	s_0	s_1	s_2
	0	1	2



Один і той самий автомат знов наведений двічі: таблицею та графом переходів. Для автоматів Мура, таблиця виходів складається з єдиного рядка — він і зображений після всіх рядків таблиці переходів, замість рядка з «доп.»/«не доп.». У графі переходів, у вершинах записують і стан, і вихідний символ, але не буває подвійних кружечків (по причині відсутності поняття допускових станів); решта аналогічне автоматам-розпізнавачам.

6.6.2 Автомати Мілі

Автомат Мі́лі (рос. «*автомат Ми́ли*», англ. типово «*Mealy machine*», значно рідше «*Mealy automaton*») — це автомат-перетворювач, заданий п'ятіркою $\langle S, X, Y, f, g\rangle$ або шісткою $\langle S, s_{start}, X, Y, f, g\rangle$, де

- S — множина станів;
- s_{start} (якщо є) — початковий стан, один з елементів S;
- X — вхідний алфавіт;
- Y — вихідний алфавіт;
- $f : S \times X \to S$ — функція переходів;
- $g : S \times X \to Y$ — функція виходів.

Ситуація з S, s_{start}, X, Y та $f : S \times X \to S$ повністю аналогічна автоматам Му́ра. Функція ж виходів відрізняється: в автоматі Мура вона має формат $g : S \to Y$, а тут $g : S \times X \to Y$. Тобто, результат залежить і від стану, і від прочитаного вхідного символу. Але у цієї відмінності насправді є ще одна сторона: і для автоматів-розпізнавачів, і для автоматів Мура аргументом функції переходів був «новий» стан, куди щойно перейшов автомат. Для автоматів Мілі, спочатку виводиться вихідний символ, залежний від «старого» (до переходу) стану та вхідного символу, а потім змінюється стан.

Приклад. Побудуємо автомат Мілі, що виконує таке перетворення: на вхід подається слово — послідовність символів "`a`" та "`b`", автомат має перетворити його за правилом: усі символи "`b`" копіюються зі входу на вихід без змін, а символи "`a`" замінюються: непарні на "`x`", парні на "`y`". Наприклад, слово "`abaaabba`" має бути перетворене у "`xbyxybbx`".

Почнемо з алфавітів: вхідний $X = \{\texttt{a}, \texttt{b}\}$, вихідний $Y = \{\texttt{b}, \texttt{x}, \texttt{y}\}$.

Тепер проведемо творчий, він же погано алгоритмізований, процес вибору множини́ станів. У цьому випадку потрібно лише два стани:

- стан s_0, який означає «вже оброблено па́рну (в т. ч. 0) кількість символів "`a`"»; са́ме ця ситуація має місце, коли ще не оброблено жодного вхідного символу, отже — цей стан розумно взяти за початковий;
- стан s_1, який означає «вже оброблено непа́рну кількість символів "`a`"».

Тепер неважко побудувати і функцію переходів та функцію виходів. Читання символу "`a`" повинно призводити до видачі на виході або символу "`x`" (якщо поточний стан (до переходу) був s_0), або символу "`y`" (якщо був стан s_1). Читання ж символу "`b`" завжди має призводити до видачі

на виході символу "b". Кожен символ "a" повинен змінювати стан на інший («протилежний»); читання ж символу "b" має не впливати на стан. Отже, функція переходів f та функція виходів g виходять такі:

Функція виходів автомата Мілі задається не одним рядком, а окремою прямокутною таблицею. Що, втім, прямо слідує з означення.

f	s_0	s_1
a	s_1	s_0
b	s_0	s_1

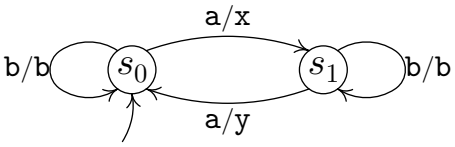
g	s_0	s_1
a	x	y
b	b	b

Початковий стан: s_0.

Граф переходів автомата Мілі в цілому аналогічний графам переходів інших розглянутих різновидів автоматів. Головна відмінність — вершини тепер не можуть *ні* обводитися подвійними кружéчками, *ні* містити вихідних символів. Натомість, вихідними символами тепер позначають дýги-переходи: кожну дугу підписують двома символами через риску дробу: символом вхідного алфавіту (вказує, коли переходити по цій дузі), та символом вихідного алфавіту (вказує, що при цьому видавати на вихід).

Якщо є зразу кілька переходів з однаковою парою станів, конкретно для автоматів Мілі зазвичай не пишуть по багато пар символів на одній стрілочці, а малюють для кожного переходу окрему стрілочку (з однією парою).

Зокрема, для щойно побудованого і поданого таблицями автомата, граф переходів такий:



6.6.3 Еквівалентність автоматів Мілі та автоматів Мура

Клас автоматів Мілі та клас автоматів Мура еквівалентні в тому смислі, що якщо є автомат одно́го типу, то завжди можна побудувати автомат іншого типу, який виконує в точності таке ж перетворення вхідних слів у вихідні.

Справді, якщо є автомат Мура, то еквівалентний йому автомат Мілі будується дуже просто: S, X, Y та f лишаються в точності такими ж, а функція виходів будується як $g_{\text{Мілі}}(s,x)=g_{\text{Мура}}(f(s,x))$. Тобто, для кожної пари $\langle s,x\rangle$ можна подивитися, в який новий стан $\tilde{s}=f(s,x)$ переходить заданий автомат Мура, який символ $g_{\text{Мура}}(\tilde{s})$ при цьому видається на вихід, і саме його й узяти в якості $g_{\text{Мілі}}(s,x)$.

Якщо ж є автомат Мілі й потрібно побудувати еквівалентний автомат Мура, то певні складнощі створює той факт, що в автоматі Мілі при різних переходах, котрі приводять у один і той самий стан, можуть видаватися на вихід різні символи — отже, їх не можна подавати переходами в один і той самий стан автомата Мура.

Тут можна вчинити громіздко, але просто: оголосити, що стани новопобудованого автомата Мура відповідатимуть не станам заданого автомата Мілі, а парам ⟨стан автомата Мілі, вихідний символ⟩. Отже, множи́ни вхідних (X) та вихідних (Y) символів не змінюються (чого і слід чекати від автоматів, що виконують однакові перетворення над словами); нова множина станів $S_{\text{Мура}}=\{\tilde{s}_{(s,y)} \mid s\in S_{\text{Мілі}}, y\in Y\}$, де $\tilde{s}_{(s,y)}$ — новий стан, відповідний парі ⟨стан (Мілі) s, вихідний символ y⟩. Тоді функція виходу очевидно має вигляд $g_{\text{Мура}}(\tilde{s}_{(s,y)})=y$ (адже y — якраз той символ, що видається на вихід), а функція переходів $f_{\text{Мура}}(\tilde{s}_{(s,y)},x)=\tilde{s}_{(f_{\text{Мілі}}(s,x),g_{\text{Мілі}}(s,x))}$.

Крім того (*якщо* початковий автомат Мілі був ініціальним) потрібно вказати початковий стан отриманого автомата Мура. Логічно, що це повинен бути якийсь зі станів $\tilde{s}_{(s_0,y)}$ (де s_0 — початковий стан заданого автомата Мілі, y перебирає елементи множини́ Y). Причому, всі ці стани за побудовою мають однакові рядки таблиці переходів — отже, в якості початкового можна взяти будь-який з них.

Легко бачити, що описаний процес застосовний до *будь-якого* автомата Мілі. Отже, цим завершене *доведення* еквівалентності класів автоматів Мілі та автоматів Мура. Це доведення конструктивне, тобто в ньому не лише доводиться, що перетворення якось можна зробити, а ще й дається явний алгоритм такого перетворення. ■

6.7 Мінімізація автоматів. Алгоритм Ауфенкампа–Хона

Мінімізація скінче́нних автоматів полягає у тому, щоб побудувати автомат, еквівалентний вказаному, але «якнайменший». Тобто, постановка задачі мінімізації автоматів стандартна, аналогічна, наприклад, постановці задачі мінімізації булевих функцій. Лишилось уточнити, в якому смислі автомати мають бути «еквівалентними» і що са́ме слід зробити «якнайменшим».

Автомати-розпізнавачі еквівалентні, коли розпізнають однакові мови; автомати-перетворювачі еквівалентні, коли для однакових вхідних слів виводять однакові вихідні. Гіпотетично можливо, щоб символи були формально присутні в алфавіт(і/ах), а фактично не використовувались; але це якісь дивні ситуації, на них зупинятись не будемо. В усіх інших випадках, зміна алфавіт(у/ів) відразу робить автомати не еквівалентними. Отже, *пробувати зменшити можна* ***лише кількість станів***. Основний спосіб такого зменшення — об'єднання кількох станів у один.

Щоб не повторювати майже однакові міркування для кожного різновиду автоматів окремо, введемо поліморфний[55] термін «*реакція* автомата на вхідне слово»: реакція розпізнавача — допуск/відхилення цього сло́ва; реакція перетворювача — вихідне слово, що утворюється у відповідь на це вхідне.

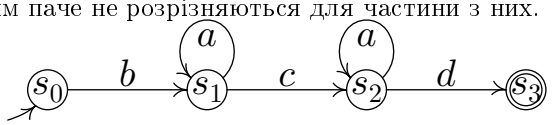
Згадаємо, що призначення станів автомата — розрізняти «типи ситуацій», у яких автомат має по-різному реагувати на однакові продовження вхідного сло́ва. Отже, якщо різні стани справді призводять до різної реакції на (хоча б деякі) вхідні слова́, їх слід залишити різними, бо інакше не вийде еквівалентний автомат. А от якщо є така група з кількох станів, що стани цієї групи не можна розрізнити за реакціями (тобто, для всіх можливих продовжень вхідного сло́ва реакція не залежить від того, в якому зі станів цієї групи перебував автомат), таку групу варто об'єднати в один стан. Тільки *як* це перевіряти? Не перебирати ж усі можливі слова́ (нескінче́нну кількість)...

Ключ до такої перевірки — у понятті k-еквівалентних станів. Стани називають *k-еквівалентними* (рос. «*k-эквивалентные* состояния», англ. «*k-equivalent* states»), коли їх не можна розрізнити за реакціями на вхідні слова́ довжиною $\leqslant k$ символів. Очевидно, повна нерозрізнюваність за реакціями (у смислі попередніх абзаців) — те саме, що ∞-еквівалентність.

Лема. *При $k_1 \leqslant k_2$, k_2-еквівалентні стани обов'язково є також і k_1-еквівалентними, а k_1-еквівалентні можуть бути k_2-еквівалентними, а можуть і не бути.*

Доведення. Всі слова́, довжина яких $\leqslant k_1$, є *частиною* всіх слів, довжина яких $\leqslant k_2$. Тому, якщо реакції не розрізняються для всіх слів довжи́ни $\leqslant k_2$, то тим паче не розрізняються для частини з них.

А щоб показати, що при однаковій реакції для всіх коротших слів цілком можлива різна реакція для довших слів, наведемо приклад.



Повторимо автомат зі стор. 173. Його стани s_0 та s_1 1-еквівалентні, бо ні ε, ні яке б не було слово з єдиного символу однаково не можуть дати реакції «слово допущене», хоч починаючи зі стану s_0, хоч зі стану s_1. Але ці са́ме стани s_0 та s_1 не є 2-еквівалентними, бо слово cd довжи́ни 2 починаючи з s_1 призводить до допуску, а починаючи з s_0 — до відхилення. ■

[Наслідок. При зростанні k згадані групи (вони ж класи еквівалентності) можуть розщеплюватися (стани старої групи «розподіляються» по новим групам), але не можуть об'єднуватися. **]**

Решта важливих властивостей задається такими лемами.

Лема. *Якщо розбиття на групи k-еквівалентних та $(k{+}1)$-еквівалентних станів однакові, це са́ме розбиття задає також і групи $(k{+}2)$-еквівалентних, $(k{+}3)$-еквівалентних, ..., ∞-еквівалентних станів.*

(без доведення)

Лема. *k-еквівалентні стани s_i та s_j є також і $(k{+}1)$-еквівалентними тоді й тільки тоді, коли для всіх символів x вхідного алфавіту стани $f(s_i, x)$ та $f(s_j, x)$ (куди переходить автомат по цьому x) k-еквівалентні.*

(без доведення)

Все сказане дає можливість сформулювати такий алгоритм мінімізації (*Ауфенкампа–Хона*):

1. Розбити множину станів на групи
   (а) для автоматів Мілі — 1-еквівалентних, тобто таких, що стовпчики таблиці виходів усіх станів групи співпадають;
   (б) для автоматів Мура — 1-еквівалентних, тобто таких, що при переході в будь-який стан групи на вихід подається один і той самий символ;
   (в) для автоматів-розпізнавачів — 0-еквівалентних, це будуть множина всіх допускових та множина всіх недопускових станів.

[55] у смислі об'єктно-орієнтованого програмування

2. Повторювати
   - побудову груп $(k{+}1)$-еквівалентних станів за відомими групами k-еквівалентних та останньою лемою

   доти, доки не виявиться, що розбиття співпали.
3. Записати мінімізований автомат, стани якого відповідають групам ∞-еквівалентних станів.

При розгляді ініціальних автоматів можливе ще одне «джерело» немінімальності: *недосяжні* стани (ті, куди не можна прийти з початкового ні при яких вхідних словах). Очевидно, недосяжні стани можна просто вилучити (разом з переходами із них), і це ніяк не вплине на роботу ініціального автомата. Алгоритм Ауфенкампа–Хона не аналізує досяжність — отже, може залишити недосяжні стани у «мінімізованому» автоматі. Томý, якщо нема впевненості, що всі стани початкового автомата досяжні, варто запускати ще перевірку досяжності — наприклад, пошуком у графі (DFS, BFS, ...). Причому, розумно написаний пошук у графі виконується швидше за алгоритм Ауфенкампа–Хона, тому перевіряти досяжність станів краще перед застосуванням алгоритму мінімізації. Хоча, можна й після; це питання лише ефективності, а не правильності.

У випадку мінімізації неповного автомата-розпізнавача, доцільно перетворити його до вигляду повного (додавши «чорну діру», див. стор. 173), а потім вже провести самý мінімізацію. Це вирішує зразу дві проблеми. По-перше, не треба придумувати спеціальні засоби для обробки відсутності переходів. По-друге, можливий (не дуже розумний) випадок, коли заданий на вході автомат і неповний, і містить явно зазначені стани, з яких нема жодного шляху до жодного допускового стану. Що одне, що інше має смисл «якщо потрапили сюди, ніяке продовження вхідного слóва вже не буде допущене». Томý такі ситуації доцільно поєднувати (не розрізняти). Що й буде забезпечено само собою, звичайним застосуванням алгоритму Ауфенкампа–Хона без будь-яких спеціальних перевірок.

Приклад 1 (мінімізація автомата-розпізнавача). Мінімізуємо неповний ініціальний детермінований автомат-розпізнавач: Початковий стан: q_0.

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_9	q_0	q_0	q_0	q_5	q_9	q_9	$-$	q_3	q_5	q_5	$-$
b	q_2	q_8	q_8	q_8	q_6	q_9	$-$	$-$	$-$	q_6	q_8	$-$
c	q_{11}	q_3	q_1	q_{10}	q_9	q_{11}	$-$	q_5	$-$	q_2	q_4	q_5
	н/д	Д	Д	Д	Д	н/д	н/д	Д	н/д	Д	Д	Д

Перед початком мінімізації, проведемо пошук недосяжних станів:

з q_0 є переходи у q_9, q_2, q_{11}; черга q_9, q_2, q_{11};

з q_9 є переходи у q_5, q_6, q_2 (вже був); черга q_2, q_{11}, q_5, q_6;

з q_2 є переходи у q_0 (вже був (початковий)), q_8, q_1; черга q_{11}, q_5, q_6, q_8, q_1;

з q_{11} є переходи у «чорну діру» та q_5 (вже був); черга q_5, q_6, q_8, q_1, $-$;

з q_5 є переходи у q_9 та q_{11} (обидва вже були); черга q_6, q_8, q_1, $-$;

з q_6 є переходи у q_9 та «чорну діру» (обидва вже були); черга q_8, q_1, $-$;

з q_8 є переходи у q_3 (новий) та «чорну діру» (вже була); черга q_1, $-$, q_3;

з q_1 є переходи у q_0, q_8, q_3 (усі вже були); черга $-$, q_3;

«чорна діра» на те й «чорна діра», що з неї переходи лише в саму себе, нових станів нема; черга q_3;

з q_3 є переходи у q_0, q_8 (обидва вже були) та q_{10} (новий); черга q_{10};

з q_{10} є переходи у q_5, q_8 (обидва вже були) та q_4 (новий); черга q_4;

з q_4 є переходи у q_5, q_6, q_9 (усі вже були); черга порожня, кінець пошуку.

Стан q_7 не був досліджений$\Rightarrow$він недосяжний$\Rightarrow$його треба позбутися.

Розпочинаємо власне алгоритм Ауфенкампа–Хона. Формуємо два класи 0-еквівалентних станів: всі досяжні не допускові (I$=\{q_0, q_5, q_6, q_8, -\}$) та всі досяжні допускові (II$=\{q_1, q_2, q_3, q_4, q_9, q_{10}, q_{11}\}$).

З'ясуємо, чи будуть вони також і 1-еквівалентними. Для цього перепишемо таблицю переходів таким чином. Стовпчики в цілому відповідають станам вказаного в умові автомата, лише вилучений недосяжний стан q_7 і додана «чорна діра». Вхідний алфавіт береться з початкового автомата без змін (це завжди так, а не лише в цьому прикладі).

У елементах таблиці замість конкретного стану будемо писати клас: наприклад, у рядку “a”, стовпчику “q_0” буде “II”, бо q_9 віднесений до групи II; у рядку “a”, стовпчику “q_1” буде “I”, бо q_0 віднесений до групи I; і так далі.

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_8	q_9	q_{10}	q_{11}	$-$
a	II	I	I	I	I	II	II	II	I	I	I	I
b	II	I	I	I	I	II	I	I	I	I	I	I
c	II	II	II	II	II	II	I	I	II	II	I	I

Бачимо, що серед станів групи $\{q_0, q_5, q_6, q_8, -\}$ не всі стовпчики виявилися однаково заповненими: q_0 та q_5 містять переходи до станів груп II, II, II; q_6 та q_8 — до станів груп II, I, I; “$-$” — до станів груп I, I, I. Значить, групу $\{q_0, q_5, q_6, q_8, -\}$ треба розщепити на три. Наприклад, q_0 та q_5 залишимо у групі I, q_6 та q_8 віднесемо до новоствореної групи III, “$-$” — до новоствореної групи IV (можна й переставити місцями вміст цих груп I, III, IV, це на суть не впливає; важливо, щоб їх було три, і q_0 та q_5 потрапили в якусь одну, q_6 та q_8 — в якусь іншу, “$-$” — у останню з трьох).

Аналогічно досліджуючи групу $\{q_1, q_2, q_3, q_4, q_9, q_{10}, q_{11}\}$, бачимо, що для q_1, q_2, q_3, q_4, q_9 та q_{10} стовпчики однакові (I, I, II), а для q_{11} стовпчик інший (I, I, I). Отже, цю групу треба розщепити на 2, наприклад, залишити $\{q_1, q_2, q_3, q_4, q_9, q_{10}\}$ у групі II, а q_{11} віднести до новоствореної групи V.

Стовпчики для станів q_{11} та “$-$” виявилися однаковими (I, I, I), але їх не можна об’єднувати в одну групу, бо після початкового об’єднання станів у групи, ці групи в принципі ніколи не можуть об’єднуватися, лише або розщеплюватись, або лишатися незмінними (див. також стор. 184).

(Конкретно в цьому випадку це видно ще й з того, що стан q_{11} допусковий, а «чорна діра» — ні, й тому ніяк не можуть бути еквівалентними.)

Отже, поточне розбиття на групи (класи 1-еквівалентних станів) таке: $\mathrm{I}=\{q_0, q_5\}$; $\mathrm{II}=\{q_1, q_2, q_3, q_4, q_9, q_{10}\}$; $\mathrm{III}=\{q_6, q_8\}$; $\mathrm{IV}=\{-\}$; $\mathrm{V}=\{q_{11}\}$.

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_8	q_9	q_{10}	q_{11}	$-$
a	II	I	I	I	I	II	II	II	I	I	IV	IV
b	II	III	III	III	III	II	IV	IV	III	III	IV	IV
c	V	II	II	II	II	V	IV	IV	II	II	I	IV

Щоб з’ясувати, чи будуть вони також і 2-еквівалентними, потрібно знову побудувати таблицю на основі початкової таблиці переходів та теперішнього розбиття на групи (див. праворуч).

Тепер виявилося, що в обох станах групи I переходи до однакових груп (II, II, V), усіх шести станах групи II переходи до однакових груп (I, III, II), та в обох станах групи III переходи до однакових груп (II, IV, IV), а групи IV та V перевіряти нема потреби, бо розщепити і так одноелементу групу неможливо. Це означає, що подальших розщеплень не буде, ці групи остато́чні.

(Слід розуміти, що тут *може* бути й інша ситуація. Хоч групи і розщеплювали так, щоб стовпчики однієї групи були однаковими, але однаковість згідно попереднього розбиття на групи не гарантує однаковості згідно наступного розбиття на групи.

Наприклад, *якби* автомат був майже таким са́мим, лише $f(q_0, \mathrm{a})=q_{11}$ (замість q_9), це ніяк не змінило б усі попередні кроки: ні щодо первинного об’єднання у дві групи $\{q_0, q_5, q_6, q_8, -\}$ та $\{q_1, q_2, q_3, q_4, q_9, q_{10}, q_{11}\}$, ні щодо їх розщеплення на п’ять груп $\{q_0, q_5\}$, $\{q_1, q_2, q_3, q_4, q_9, q_{10}\}$, $\{q_6, q_8\}$, $\{-\}$ та $\{q_{11}\}$. *Перша* відмінність у процесі мінімізації цих автоматів — са́ме на цьому етапі: для автомата, де $f(q_0, \mathrm{a})=q_9$, можна припиняти основний цикл, а якби було $f(q_0, \mathrm{a})=q_{11}$, стовпчик q_0 набув би вигляду (V, II, V), стовпчик q_5 — вигляду (II, II, V), групу $\{q_0, q_5\}$ треба було б розщепити на дві $\{q_0\}$ і $\{q_5\}$, після чого заново будувати всю таблицю й дивитися, чи не виникло потреби розщепити ще якісь групи. Продовжувати далі аналіз зміненого автомата не будемо, повернемось до початкового варіанта з $f(q_0, \mathrm{a})=q_9$. Це був лише відступ, щоб показати, що в алгоритмі недаремно написано «повторювати ..., доки не ...».)

Коли групи перестали змінюватися (стали ∞-еквівалентними), слід сформувати з цих груп стани мінімізованого автомата. Наприклад, співставимо групі I стан (нового, мінімізованого автомата) s_1, групі II стан s_2, групі III стан s_3, групі IV стан s_4, групі V стан s_5 (можна й якось переставити місцями, або трохи інакше назвати стани, як-то почавши нумерацію з 0; то деталі, важливо лише, щоб стани співставлялися групам).

Таблиця переходів береться із останньої побудованої таблиці з групами: вона якраз виражала, що який би не взяли зі станів групи $\mathrm{I}=\{q_0, q_5\}$, однаково отримуємо переходи по a у групу II, по b у групу II, по c у групу V, і т. д. Так отримуємо таблицю, наведену праворуч.

	s_1	s_2	s_3	s_4	s_5
a	s_2	s_1	s_2	s_4	s_4
b	s_2	s_3	s_4	s_4	s_4
c	s_5	s_2	s_4	s_4	s_1

Далі треба з’ясувати, які стани нового автомата допускові, які ні. Тут дивимось, чи допусковий будь-який представник $\mathrm{I}=\{q_0, q_5\}$, будь-який представник $\mathrm{II}=\{q_1, q_2,$

q_3, q_4, q_9, $q_{10}\}$, і т. д. Оскільки спочатку об'єднували лише допускові з допусковими і не допускові з не допусковими, а потім групи лише розщеплювались, всере́дині кожної групи це однаково.

Початковий стан з'ясовується аналогічно — згідно того, в яку групу потрапив початковий стан початкового автомата: $q_0 \in \{q_0, q_5\} = \mathrm{I}$, тож початковим станом мінімізованого автомата слід взяти s_1.

	s_1	s_2	s_3	s_4	s_5
a	s_2	s_1	s_2	s_4	s_4
b	s_2	s_3	s_4	s_4	s_4
c	s_5	s_2	s_4	s_4	s_1
	н/д	Д	н/д	н/д	Д

Початковий стан: s_1.

Початковий автомат був неповним, і перед мінімізацією була штучно введена «чорна діра». Так що один зі станів мінімізованого автомата по суті й є цією «чорною дірою». При бажанні, можна повернути автомат до неповного вигляду, чим зменшити кількість «явних» станів ще на 1.

«Чорною дірою» автомата є s_4 (це можна побачити хоч із $\mathrm{IV} = \{-\}$, хоч із того, що s_4 не допусковий і містить переходи лише у себе).

	s_1	s_2	s_3	s_4
a	s_2	s_1	s_2	$-$
b	s_2	s_3	$-$	$-$
c	s_4	s_2	$-$	s_1
	н/д	Д	н/д	Д

Поч. стан: s_1.

Щоб не було «дірки» в нумерації, можна також перейменувати s_5 на s_4 (як у заголовку останнього стовпчика, так і в значенні $f(\mathrm{c}, s_1)$). Але чи варто робити такі перейменування, і чи варто взагалі вертати автомат до неповного вигляду — не очевидно й залежить від ситуації. Тут лише показано, як це робити в разі потреби.

Приклад 2 (мінімізація автомата Мілі). Мінімізуємо такий автомат:

f	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0	s_1	s_3	s_4	s_6	s_6	s_6	s_6
1	s_2	s_4	s_5	s_4	s_5	s_3	s_2

g	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0	a	b	b	a	a	a	a
1	a	b	b	b	b	b	a

Початковий стан не заданий — значить, шукати недосяжні стани неможливо, а у мінімізованому автоматі теж не буде початкового стану. На практиці може бути доцільним повернутися і розібратися, чому це раптом в автомата нема початкового стану; але припустимо, що з'ясувати неможливо. Тим паче, що не ініціальні автомати-перетворювачі, поведінка яких залежить від того, з якого стану починати робо́ту, в принципі можливі.

Бачимо, що для станів s_0 та s_6 і вхідний символ "0", і вхідний символ "1" призводять до реакції (видачі вихідного символу) "a"; аналогічно, для станів s_1 та s_2 і "0", і "1" призводять до реакції "b"; для s_3, s_4 та s_5 "0" призводить до "a", "1" — до "b". Тобто, маємо три групи 1-еквівалентних станів: $\mathrm{I} = \{s_0, s_6\}$, $\mathrm{II} = \{s_1, s_2\}$, $\mathrm{III} = \{s_3, s_4, s_5\}$.

З'ясуємо, чи будуть вони також і 2-еквівалентними. Для цього перепишемо таблицю переходів, пишучи замість конкретного стану групу (як-то «$f(s_0, 0) = s_1$, $s_1 \in \{s_1, s_2\} = \mathrm{II}$, томý на перетині рядка 0 та стовпчика s_0 пишемо II» — цілком аналогічно прикладу 1; процес розщеплення груп для детермінованих повних розпізнавачів, автоматів Мура та автоматів Мілі відбувається абсолютно однаково).

f	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0	II	III	III	I	I	I	I
1	II	III	III	III	III	III	II

Бачимо, що для станів I-ої групи переходи по вхідному символу "0" відрізняються: $f(s_0, 0)$ потрапляє до II-ої групи, тоді як $f(s_6, 0)$ — до I-ої. Отже, ці стани *не* 2-еквівалентні, і їх доведеться «розвести» по різним групам. Для станів же II-ої та III-ої груп всі стовпчики таблиці переходів всере́дині кожної групи однакові ($\langle \mathrm{III}, \mathrm{III} \rangle$ для II-ої, $\langle \mathrm{I}, \mathrm{III} \rangle$ для III-ої). Отже, II-а та III-а група лишаються без змін. Остаточно, маємо чотири групи 2-еквівалентних станів: $\mathrm{I} = \{s_0\}$, $\mathrm{II} = \{s_1, s_2\}$, $\mathrm{III} = \{s_3, s_4, s_5\}$, $\mathrm{IV} = \{s_6\}$.

Чергова таблиця має вигляд, наведений праворуч. У ній всі стовпчики II-ої групи однакові між собою і всі стовпчики III-ої групи однакові між собою (розбиття на 3-еквівалентні групи дорівнює розбиттю на 2-еквівалентні). Отже, стани кожної групи ∞-еквівалентні, і робити подальші розщеплення не треба.

f	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0	II	III	III	IV	IV	IV	IV
1	II	III	III	III	III	III	II

Остато́чно мінімізований автомат:

f	q_0	q_1	q_2	q_3
0	q_1	q_2	q_3	q_3
1	q_1	q_2	q_2	q_1

g	q_0	q_1	q_2	q_3
0	a	b	a	a
1	a	b	b	a

6.8 Завдання до розділу 6

1. Перелічити всі слова́ довжини́ $\leqslant 7$ мови, заданої рег. виразом $(ab|xyz)^*$. (Вибір лише слів певної довжини́ — украй не типова для регулярних виразів дія. Але автор посібника вважає, що цінність завдання «перелічити *всі*» досить велика, щоб змиритися з цим.)
2. Перелічити всі слова́ довжини́ $\leqslant 6$ мови, заданої $\big(abc^*|xy^*|(abc)^*|(xy)^*\big)$.
3. Заповнити всі комірки таблички знаками "+" (означає: слово, вказане у стовпчику, може бути задане регулярним виразом, вказаним у рядку) та "−" (не може):

	ad	ada	abd	$abcd$	$acbd$	$abcbd$	$abcbcd$	$abbd$	$abbcd$	$accd$
$a(b\|c)^*d$										
$a(b^*\|c^*)d$										
$a(bc)^*d$										
ab^*c^*d										
$a(b^*\|c)d$										
$a(b\|c)d$										

4. Комплект задач на практичне застосування regex-ів, доступний за будь-яким з посилань
   (а) `https://ejudge.ckipo.edu.ua`, змагання №81
   (б) `https://informatics.mccme.ru/mod/statements/view.php?id=3700`
5. Розглянемо мову рядків вигляду $a^m b^n c^k$ ($m\geqslant 1$, $n\geqslant 2$, $k\geqslant 1$), або, що те само, $\underbrace{a\ldots a}_{\geqslant 1}\underbrace{b\ldots b}_{\geqslant 2}\underbrace{c\ldots c}_{\geqslant 1}$.
   (а) Побудувати відповідний цій мові *математичний* регулярний вираз;
   (б) побудувати автомат-розпізнавач цієї мови;
   (в) зобразити цей автомат також в іншому вигляді (табличному, якщо був побудований графом, чи графом, якщо був таблично);
   (г) застосувати цей автомат до вхідних слів:
      i. $aabc$;
      ii. $abbbcc$;
   (д) яка з'явиться відмінність, якщо замінити, наприклад, bb^* на b^*b?
6. Побудувати (детерміновані) автомати-розпізнавачі для мов, заданих рег. виразами:
   (а) $a(b|c)^*d$ (б) $a(b^*|c^*)d$ (в) $a(bc)^*d$ (г) ab^*c^*d
7. Для алфавіту $\{a, b, c\}$, побудувати автомат-розпізнавач рядків...
   (а) ... парної довжини́.
   (б) ..., які містять непарну кількість літер a (кількості інших літер не впливають на результат).
   (в) ..., які містять парну кількість літер a та непарну кількість літер b (кількість літер c не впливає на результат).
8. Будемо вважати узагальненим словом непорожню послідовність або лише кирилічних букв, або лише цифр, або таку, де є буквенні та цифрові непорожні частини, розділені дефісами. Наприклад, "`2й7`" не задовольняє це означення, а "`Щ`", "`854`", "`Щ-854`", "`3-14-15-92-шість`" та "`Цис-2-Хлор-9-3-ди-метил-амін`" задовольняють. Побудувати для мови узагальнених слів regex (згідно синтаксису Java або аналогічного) та детермінований розпізнавач.
9. (а) Побудувати ε-НСА для розпізнавання чисел, кратних 25 (чи́сла подаються на вхід поцифрово, зліва направо), вважаючи, що кратні 25 чи́сла — це в точності послідовності цифр, що закінчуються на 25, або 50, або 75, або 00. Автомат повинен бути значно простішим за детермінований (графічне подання має містити всього $\approx$6 стрілочок).
   (б) Побудувати ε-НСА для розпізнавання чисел, кратних 25 (чи́сла подаються на вхід поцифрово, зліва направо), «згадавши», що може бути (а може і не бути) ще знак "`+`" чи "`-`" спереду, а також може бути одноцифровий 0.
   (в) Застосувати автомат з п. 9а до вхідних слів:
      i. 250752,
      ii. 3550425,

причому оформити кожне з застосувань двома способами:

i. у вигляді таблиці, як у розд. 6.5.1;

ii. у вигляді діаграми переходів, як у розд. 6.5.4.

(г) Детермінімувати (засобами powerset construction, розд. 6.5.7) автомат з п. 9б.

10. (а) Побудувати ε-НСА для розпізнавання рядків, що *закінчуються на* "`колокол`" або "`локон`" (один і той самий автомат, повинен видавати допусковий вердикт і тоді, коли слово закінчилося на "`колокол`", і тоді, коли слово закінчилося на "`локон`").

(б) Детермінізувати отриманий ε-НСА, подавши результат і таблично, і графічно.

11. На стор. 185 пропонувалося відкидати недосяжні стани, запускаючи пошук у гра́фі. Може здатися, ніби можна вчинити й значно простіше: повилучати як недосяжні ті й тільки ті стани, в які нема жодного переходу. Чому застосування щойно описаного спрощеного підходу не завжди призводить до правильного вилучення недосяжних станів?

12. Побудувати автомат Мілі зі вхідним та вихідним алфавітами $X = Y = \{0, 1\}$, який виконує «затримку» входу: на перших двох тактах завжди (незалежно від входу) видає нулі, далі на кожному i-му такті виводить символ, прочитаний на $(i-2)$-му (наприклад, слово "`1011101001`" перетворюється у "`0010111010`").

13. Персонаж комп'ютерної гри перебуває на вузькій платформі. Ухиляючись від падінь усіляких предметів, він на кожному кроці або пересувається на 1 м ліворуч, або на 1 м праворуч, або лишається на місці. Відходити від початкового положення далі, як на 3 м, не бажано, бо тоді доведеться впасти з платформи у болото, а там рухатися зовсім неможливо.

Побудуйте автомат Мура, який за послідовністю вказівок щодо переміщень персонажа (закодованих літерами L, R, S) на кожному кроці повідомлятиме, де він зараз знаходиться: у початковій позиції (символ "0"), лівіше за неї (символ "$-$"), правіше неї (символ "$+$") чи впав додолу (символ "$\downarrow$").

14. Персонажі попередньої гри рішенням своєї профспілки домоглися встановлення загорож по краям платформи, так що тепер замість прикрих падінь вони просто нікуди не рухатимуться. Перебудуйте відповідним чином автомат.

15. Побудувати автомат (Мілі або Мура), який виконує додавання двійкових чисел. Вхідними символами є пари двійкових цифр, що знаходяться в однакових розрядах доданків, починаючи справа. Вихідними — символи 0 та 1. Наприклад, $101000 + 11010$ буде подаватися на вхід як $\boxed{00}$, $\boxed{01}$, $\boxed{00}$, $\boxed{11}$, $\boxed{01}$, $\boxed{10}$, $\boxed{00}$; автомат повинен видати 0, 1, 0, 0, 0, 0, 1 (з тим, що виведення останньої 1 потребу́є введення «зайвої» пари $\boxed{00}$, змиритися, бо у класичній автоматній моделі з цим нічого не вдіяти). Автомат зобразити і таблично, і графічно.

16. Аналогічно попередній задачі 15, побудувати автомат, який виконує віднімання невід'ємних двійкових чисел (вважаючи, що зменшуване не менше за від'ємник).

17. Це завдання пропонується давати по варіантам (кожен студент виконує один варіант, варіанти розподілені рівномірно). Мінімізувати детермінований неповний автомат-розпізнавач (вилучення недосяжних станів, алгоритм Ауфенкампа–Хона).

(1) Початковий стан: q_0

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_9	q_0	q_0	q_0	q_5	q_9	q_9	$-$	q_3	q_5	q_5	$-$
b	q_2	q_8	q_8	q_8	q_6	q_9	$-$	$-$	$-$	q_6	q_8	$-$
c	q_{11}	q_3	q_1	q_{10}	q_9	q_{11}	$-$	q_5	$-$	q_2	q_4	q_5
	н/д	Д	Д	Д	Д	н/д	н/д	Д	н/д	Д	Д	Д

(2) Початковий стан: q_1

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_8	q_4	q_8	q_{10}	q_1	q_4	q_8	q_4	q_4	q_8	q_3	q_8
b	q_7	$-$	q_7	$-$	$-$	$-$	q_7	$-$	q_9	q_5	$-$	q_1
c	q_7	q_6	q_5	q_0	$-$	q_0	q_3	q_2	$-$	q_5	$-$	q_3
	н/д	Д	н/д	Д	н/д	Д	н/д	Д	Д	н/д	н/д	н/д

(3) Початковий стан: q_{11}

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_3	–	q_8	q_6	–	q_0	q_5	q_0	q_2	–	–	–
b	–	q_{10}	–	–	q_4	–	–	–	–	q_{10}	q_{10}	q_{10}
c	q_9	q_9	q_7	q_7	q_9	q_7	q_{11}	q_7	q_{11}	q_8	q_9	q_6
	н/д	Д	Д	Д	Д	Д	н/д	Д	н/д	н/д	Д	н/д

(4) Початковий стан: q_1

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_7	q_6	–	–	–	q_6	q_{11}	q_9	–	–	q_3	–
b	–	–	–	–	q_3	–	q_4	q_4	–	–	q_4	–
c	q_8	q_3	q_1	q_5	q_3	q_3	q_{11}	q_9	q_0	q_0	q_9	q_0
	н/д	н/д	н/д	н/д	Д	н/д	Д	Д	н/д	н/д	Д	н/д

(5) Початковий стан: q_{11}

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_0	q_{10}	q_{10}	q_3	q_3	q_3	q_7	q_7	q_3	q_{10}	q_{10}	q_{10}
b	q_8	q_7	q_6	q_4	q_0	q_4	q_1	q_4	q_0	q_7	q_9	q_9
c	–	q_3	q_3	q_2	q_3	q_{10}	–	–	q_{10}	q_3	q_8	q_3
	н/д	Д	Д	н/д	Д	Д	н/д	н/д	Д	Д	н/д	Д

(6) Початковий стан: q_1

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_4	q_3	q_3	–	q_5	–	q_5	q_6	q_5	q_{10}	q_3	q_6
b	q_{10}	–	–	–	q_7	–	q_0	q_4	–	q_{10}	q_9	q_4
c	q_{11}	q_{10}	q_4	q_3	q_9	q_5	q_7	q_7	q_{10}	q_{11}	q_9	q_0
	н/д	Д	Д	н/д	Д	н/д	Д	н/д	Д	н/д	Д	н/д

(7) Початковий стан: q_{10}

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_{10}	q_2	q_0	–	–	–	–	q_2	q_{11}	–	q_7	q_0
b	q_3	q_9	q_8	q_7	q_1	q_1	q_7	q_3	q_3	q_1	q_7	q_7
c	q_2	q_{10}	–	q_4	–	q_4	q_4	q_2	q_2	q_4	–	–
	н/д	н/д	Д	н/д	Д	н/д	н/д	н/д	н/д	н/д	Д	Д

(8) Початковий стан: q_3

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	–	–	–	q_1	q_0	–	–	–	–	q_4	q_0	–
b	q_9	q_9	–	–	–	q_9	–	–	–	q_2	–	–
c	–	–	q_4	q_6	q_6	–	q_{10}	q_{10}	q_4	q_5	q_{11}	q_4
	н/д	н/д	н/д	Д	Д	н/д	н/д	н/д	н/д	Д	Д	н/д

(9) Початковий стан: q_{10}

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_8	q_8	q_3	q_2	q_0	–	q_7	q_2	q_1	q_2	–	q_8
b	q_6	q_4	–	q_8	–	q_{11}	–	q_2	–	–	q_{11}	–
c	–	–	q_7	–	q_1	q_7	q_7	–	q_7	–	q_7	–
	н/д	н/д	Д	н/д	Д	Д	Д	н/д	Д	н/д	Д	н/д

(10) Початковий стан: q_6

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_{11}	–	–	–	q_0	–	q_8	q_6	q_{11}	q_8	–	q_4
b	q_2	–	q_0	–	q_2	–	q_2	q_3	q_2	q_1	–	q_2
c	q_0	q_7	q_{10}	q_9	q_8	q_7	q_8	q_8	q_4	q_0	q_7	q_0
	н/д	Д	Д	Д	н/д	Д	н/д	н/д	н/д	н/д	Д	н/д

(11) Початковий стан: q_8

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	q_1	q_1	q_4	q_0	q_{10}	q_4	–	q_{11}	–	q_2	q_{10}	q_1
b	q_6	q_6	–	–	q_6	q_8	q_{11}	–	q_3	–	q_6	–
c	q_6	q_8	–	–	q_6	q_8	q_7	q_2	q_7	q_2	q_6	–
	н/д	н/д	н/д	н/д	н/д	н/д	Д	Д	Д	Д	н/д	н/д

(12) Початковий стан: q_{11}

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
a	–	–	–	–	–	–	q_4	–	–	q_8	–	–
b	q_2	q_1	q_8	q_5	q_{10}	q_1	–	q_5	q_2	–	q_0	q_5
c	–	q_9	–	q_4	–	q_6	–	q_0	–	–	–	q_0
	н/д	н/д	н/д	Д	н/д	н/д	Д	Д	н/д	Д	н/д	Д

Додаткові завдання підвищеного рівня складності

1*. Розв'язати «регулярні кросворди» з сайту `https://regexcrossword.com`

2*. Написати (обов'язково використовуючи бібліотеку роботи з regex-ами) програму, що читає файл з вихідним кодом (іншої) програми мовою C++ і створює файл з аналогічним вихідним кодом, де в усіх двовимірних масивах зроблено обмін індексів — наприклад, "`a[x][y+1]`" → "`a[y+1][x]`"; програма повинна допускати також однократну вкладеність квадратних дужок, наприклад "`arr[x+dx[k]][y+dy[k]]`" → "`arr[y+dy[k]][x+dx[k]]`".

Достатньо, щоб програма вміла працювати лише для масивів, оголошених зі статичними розмірами (писати аналогічну програму для `vector`-ів, які мають багато різних засобів зміни свого розміру — об'єктивно набагато складніше, і про це *не* йдеться). Достатньо вважати, що у програмі можливі лише одно- та двовимірні масиви (гарантовано нема 3-, 4-, ...-вимірних), та що для двовимірного масиву завжди вказуються обидва індекси.

3*. Створити автомат-розпізнавач, котрий допускає лише слова́ вигляду:

(а) $\{\mathtt{a}^{2k} \mid k \in \mathbb{N}\}$, тобто `aa`, `aaaa`, $\underbrace{\mathtt{aaaaaa}}_{6}$, $\underbrace{\mathtt{aaaaaaaa}}_{8}$, $\underbrace{\mathtt{aaaaaaaaaa}}_{10}$, ... (або довести, що це неможливо);

(б) $\{\mathtt{a}^{2^k} \mid k \in \mathbb{N}\}$, тобто `aa`, `aaaa`, $\underbrace{\mathtt{aaaaaaaa}}_{8}$, $\underbrace{\mathtt{aaaaaaaaaaaaaaaa}}_{16}$, ... (або довести, що це неможливо).

Слід зробити *обидва* пункти.

4*. Написати програму, що емулює детермінований автомат-розпізнавач: задаються таблиця переходів, множина допускових станів, початковий стан та вхідне слово, програма виводить покроковий протокол робо́ти. Достатньо виводити все просто текстом, без зображень.

5*. Написати програму, що емулює роботу ε-НСА:

(а) у вигляді таблиці, як у розд. 6.5.1;

(б) у вигляді діаграми переходів, як у розд. 6.5.4.

6*. Реалізувати мовою програмування алгоритм мінімізації автоматів (Ауфенкампа–Хона). Можливі варіанти постановки задачі (в порядку зростання складності):

- тільки для детермінованих повних розпізнавачів;
- і для детермінованих повних розпізнавачів, і для Мілі, і для Мура;
- і для детермінованих повних розпізнавачів, і для Мілі, і для Мура, і при цьому максимально використано code reusing — власне мінімізація написана *один раз*, тип автомата визначається за значенням параметра;
- і для детермінованих повних розпізнавачів, і для Мілі, і для Мура, власне мінімізація написана лише один раз, і при цьому підтримка різних типів автоматів робиться не за рахунок значень параметра, а за рахунок того, що мінімізація працює з абстрактним класом-предком, а дії, які для різних автоматів треба робити по-різному, реалізовані у поліморфних перевантаженнях нащадків.

Додатково можна зробити виключення недосяжних станів, але лише якщо власне мінімізація хоч якось реалізована.

7*. Реалізувати програму, що розв'язує таку задачу:

У деяких ВНЗ діє система оцінювання, згідно з якою студент може набрати за предмет від 0 до 100 балів, з них від 0 до 75 — протягом семестру, від 0 до 25 — протягом підсумкового іспиту. Остаточна оцінка визначається залежно від суми семестрових та екзаменаційних балів згідно такої таблиці:

Сума балів	*Оцінка європейська*	*Оцінка національна*
90–100	*A*	*відмінно*
82–89	*B*	*добре*
75–81	*C*	*— " —*
68–74	*D*	*задовільно*
60–67	*E*	*— " —*
35–59	*FX*	*незадовільно*

В рамках цієї задачі вважаємо, що якщо студент протягом семестру набрав строго менше 35 балів, він/вона не допускається до складання іспиту, і що таких студентів заранні викреслюють зі списків.

При прочитанні згори донизу стовпчику європейських оцінок можуть утворюватися різноманітні «слова́». Наприклад, якщо суми балів трьох підряд по списку студентів становлять 92, 75 та 66, вони отримують оцінки A, C та E відповідно, і утворюється «слово» ACE. У випадку оцінки FX, у «слово» додаються обидві літери (спочатку F, потім X).

Іспит — штука ризикована, і знати його результати наперед неможливо. Але викладач приблизно знає рівень знань кожного студента та перелік завдань. Тому він може оцінити ймовірність того, що такий-то студент набере стільки-то балів. Тож нехай для кожного студента відомі ймовірність (у відсотках), що цей студент набере 0 балів, 1 бал, ..., 25 балів — разом 26 цілих невід'ємних чисел, сума яких дорівнює 100. Бали, набрані студентом протягом семестру, теж відомі (як конкретне ціле число від 35 до 75, без яких би не було ймовірностей).

Викладач, що приймає іспит — великий естет, і йому дуже неприємно, коли у «слові», утвореному оцінками, трапляються деякі «некрасиві» підрядки (са́ме підрядки, тобто літери йдуть підряд).

Напишіть програму, яка знаходитиме ймовірність, що викладач-естет буде задоволеним, бо у «слові» оцінок не трапиться жодного «некрасивого» підрядка.

Перший рядок вхідних даних містить кількість тестових блоків $TEST\_NUM$. У кожному тестовому блоці, перший рядок блоку містить кількість студентів у групі N. Кожен з подальших N рядків містить по 27 цілих чисел, розділених пропусками (пробілами) — кількість семестрових балів (від 35 до 75), та 26 ймовірностей, відповідних 0, 1, 2, ..., 25 екзаменаційним балам (кожна ймовірність невід'ємна, сума дорівнює 100). Наступний рядок містить число K — кількість «некрасивих» з точки зору викладача-естета підрядків, кожен з подальших K рядків тестового блоку — черговий «некрасивий» підрядок. Гарантовано, що кожен з цих K рядків містить лише великі латинські літери (від 2 до 15 штук) і завершується символом переведення рядка.

Ваша програма має вивести на стандартний вихід єдине дійсне число в єдиному рядку — знайдену ймовірність (у відсотках) того, що викладач-естет буде задоволеним. Формат виведення дійсного числа́ довільний, але із використанням десяткової точки (а не коми). Відповідь зараховуватиметься, якщо відносна похибка не перевищуватиме 10^{-6}.

Вхідні дані	*Результат*
1 *3* *72 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 2 3 5 7 9 14 16 21 12 5 1* *55 0 0 0 1 2 3 4 5 6 7 8 8 9 8 8 7 6 5 4 3 2 2 1 1 0 0* *55 0 0 0 1 2 3 4 5 6 7 8 8 9 8 8 7 6 5 4 3 2 2 1 1 0 0* *2* *DE* *WAW*	*79.5*

У оцінках не може з'явитися літера W, тому підрядок WAW можна відкинути, лишивши тільки DE. 1-й студент набере щонайменше 72+10=82 балів, тобто не зможе отримати D. Отже, «некрасивий» підрядок DE з'явиться тоді й тільки тоді, коли 2-й студент донабере від 13 до 19 балів (ймовірність $8\%+8\%+7\%+6\%+5\%+4\%+3\%=41\%$), і, водночас, 3-й — від 5 до 12 (ймовірність $3\%+4\%+5\%+6\%+7\%+8\%+8\%+9\%=50\%$). Отже, слово DE з'явиться з імовірністю $0{,}41\times 0{,}5=0{,}205$, а з імовірністю $1-0{,}205=0{,}795$ (інакше кажучи, 79,5%) не з'явиться.

Вказівки. Пояснення з умови, хоч і пояснює конкретну відповідь 79,5%, *не дає* ключа до розв'язання задачі при довільних вхідних даних.

Один зі способів розв'язання (не найефективніший, але достатній для обмежень $K, N \leqslant 100$) такий. Побудувати (аналогічно завд. 10а зі стор. 189, тільки не на папері для відомих слів, а у програмі для слів, заданих у вхідних даних) недетермінований автомат-розпізнавач,

який видає допуск тоді й тільки тоді, коли щойно закінчилося будь-яке з "неприємних" слів. Потім детермінізувати цей автомат. Потім перетворити цей автомат на ймовірнісний (відомі шість ймовірностей: що поточна вхідна літера, тобто оцінка поточного студента, буде A, що B, тощо; в цьому посібнику такого нема, деталі знайти або придумати самостійно), і зробити, щоб при потраплянні у допусковий стани ймовірність скидувалася на 0. Далі лишається поєднати звичайне застосування автомата з формулою повної ймовірності.

Більш ефективні способи (які можна використовувати не лише при $K, N \leqslant 100$, а й при $K, N \leqslant 5000$) — наприклад, алгоритм Ахо–Корасик — теж спираються на побудову детермінованого автомата-розпізнавача та перетворення його у ймовірнісний, але іншим способом, без проміжного недетермінованого автомата (а отже, й без детермінізації).

Ця задача (без умови, лише перевірка) доступна як задача «G Exam» змагання №43 «Всякі різні дорішування» сайту `https://ejudge.ckipo.edu.ua` Реалізація описаного алгоритму (для $K, N \leqslant 100$), повинна проходити тести 1–26 і давати перевищення часу (тривалості) роботи на тестах 27–31. Розв'язок з більш ефективним способом побудови автомата (наприклад, алгоритм Ахо–Корасик) повинен проходити усі тести 1–31.

Список літератури

Основна література

1. Нікольський Ю. В., Пасічник В. В., Щербина Ю. М. «Дискретна математика», К., BHV, 2007.
2. Міхайленко В. М., Федоренко Н. Д., Демченко В. В. «Дискретна математика», К., видавництво Європейського університету, 2003.

Додаткова література

3. Андерсон Дж., «Дискретная математика и комбинаторика», М.–СПБ–К., Вильямс, 2004.
4. Ахо А., Сети́ Р., Ульман Дж. «Компиляторы. Принципы, технологии, инструменты» М.–СПБ–К., Вильямс, 2001.
5. Виленкин Н. Я., Виленкин А. Н., Виленкин П. А., «Комбинаторика» М., ФИМА–МЦНМО, 2007.
6. Виленкин Н. Я. «Рассказы о множествах» М., МЦНМО, 2007.
7. Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И. «Лекции по теории графов» М., Наука, 1990.
8. Капітонова Ю. В., Кривий С. Л., Летичевський О. А., Луцький Г. М., Печурін М. К. «Основи дискретної математики». К., LitSoft, 2000.
9. Карпов Ю. Г. «Теория автоматов» СПБ, Питер, 2002.
10. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. «Алгоритмы: построение и анализ» (второе издание) М.–СПБ–К., Вильямс, 2005.
11. Коршунов Ю. М. «Математические основы кибернетики» М., Энергия, 1980.
12. Липски В. «Комбинаторика для программистов» М., Мир, 1988.
13. Михайленко В. М., Федоренко Н. Д. «Спеціальні розділи математики» К., Вища школа, 1992.
14. Новиков О. А. «Дискретная математика для программистов» СПБ, Питер, 2000.
15. Рейнгольд Э., Нивергельт Ю., Део Н. «Комбинаторные алгоритмы. Теория и практика» М., Мир, 1980.
16. Сигорский В. П. «Математический аппарат инженера» К., Техніка, 1977.
17. Бен Форта «Освой самостоятельно регулярные выражения. 10 минут на урок» М.–СПБ–К., Вильямс, 2005.
18. Харари Ф. «Теория графов» М., Мир, 1973.
19. Яблонский С. В. «Введение в дискретную математику» М., Наука, 1979.

Покажчики

Покажчики термінів, інтегровані зі словниками

Українсько-російсько-англійський словник та покажчик

алгоритм Евкліда	алгоритм Евклида	Euclid's *algorithm*	87
алгоритм Флойда	алгоритм Флойда	Floyd's *algorithm*	143
вхідний *алфавіт*	входной *алфавит*	input *alphabet*	170
антисиметричне відношення	антисимметричное отношение	antisymmetric relation	61
асоціативність (“$\vee$”, “$\wedge$”)	ассоциативность (“$\vee$”, “$\wedge$”)	associativity (“$\vee$”, “$\wedge$”)	14
база індукції	базис (база) индукции	base case (basis)	81
бектрекінг (відтинання при переборі)	бэктрэкинг (отсечения при переборе)	back-tracking	115
бієкція (бієктивна функція)	биекция (биективная функция)	bijection (injective function)	73
бінарне відношення	бинарное отношение	binary relation	58
бінарне відношення в $A \times B$	бинарное отношение в $A \times B$	binary relation between A and B	58
бінарне відношення на A	бинарное отношение на A	binary relation on A	58
бінарне «у вузькому смислі» відношення	бинарное «в узком смысле» отношение	binary relation on A	58
бінарне «у широкому смислі» відношення	бинарное «в широком смысле» отношение	binary relation between A and B	58
бінарний	бинарный	binary	9
біном Ньютона	бином Ньютона	binomial expansion	100
біноміальні коефіцієнти	биномиальные коэффициенты	binomial coefficients	99
блок	блок	block	126
булеа́н множини	булеа́н множества	power set of set	49
діаграма *Вейча*	диаграмма *Вейча*	Karnaugh map (K-map)	20
діаграми *Венна*	диаграммы *Венна*	Venn's diagrams	48
вершина	вершина	vertex	112
вершинна двозв'я́зність	вершинная двусвя́зность	vertex biconnectedness	126
вершинно k-зв'я́зний граф	вершинно k-свя́зный граф	k-vertex-connected graph	126
сортування *вибором*	сортировка *выбором*	selection sort	84
відновлення шляху (зворотній хід)	восстановление пути (обратный ход)	restoring path (backtrace)	131
відношення	отношение	relation	57
бінарне *відношення* в $A \times B$	бинарное *отношение* в $A \times B$	binary *relation* between A and B	58
бінарне *відношення* на A	бинарное *отношение* на A	binary *relation* on A	58
відношення еквівалентності	отношение эквивалентности	equivalence *relation*	63
відношення порядку	отношение порядка	order *relation*	66
матриця *відстаней*	матрица *расстояний*	distance matrix	122
ві́дстань	расстоя́ние	distance	121
відтинання при переборі	отсечения при переборе	back-tracking	115
відхиляти слово	отклонять слово	to *decline* string	171
виключне або	исключающее или	exclusive or	8
вільна змінна	свободная переменная	free variable	32
регулярний *вираз*	регулярное *выражение*	regular *expression*	164
висяча (кінцева́) вершина	висячая (концева́я) вершина	leaf (pendant vertex)	114
принцип *включень та виключень*	принцип *включений и исключений*	inclusion–exclusion principle	98
власна підмножина	собственное подмножество	proper subset	48
власний підграф	собственный подграф	proper subgraph	116
алгоритм *Воршалла*	алгоритм *Уоршалла*	Warshall's algorithm	143
сортування *вставками*	сортировка *вставками*	insertion sort	85
вхідне слово	входное слово	input string	170
вхідний алфавіт	входной алфавит	input alphabet	170
гамільтонів цикл	гамильтонов цикл	Hamiltonian cycle	128
діаграма *Гассе*	диаграмма *Хассе*	Hasse diagram	67
гіперграф	гиперграф	hypergraph	117
пошук у *глибину* (вглиб)	поиск в *глубину*	depth-first search	134
голова черги	голова очереди	head of queue	129
граф	граф	graph	112

граф конденсації	граф конденсации	condense of graph	125
дводольний (двочастковий) граф	двудольный граф	bipartite graph	114
двозв'я́зна компонента	двусвя́зная компонента	biconnected component	126
двозв'я́зність графа	двусвя́зность графа	biconnectedness of graph	126
ДДНФ	СДНФ	CDNF	16
закони *де Моргана*	законы *де Моргана*	DeMorgan's laws	14
алгоритм *Дейкстри*	алгоритм *Дейкстры*	Dijkstra's algorithm	141
декартів добуток	декартово произведение	Cartesian product	55
декартів степінь	декартова степень	Cartesian power	55
метод семантичних *дерев*	метод семантических *деревьев*	semantic tree method	27
дерево (неорієнтоване)	дерево (неориентированное)	tree (undirected)	146
детермінізація	детерминизация	powerset construction	179
діагональний метод	диагональный метод	diagonal argument	74
діаграма Вейча	диаграмма Вейча	Karnaugh map (K-map)	20
діаграма Гассе	диаграмма Хассе	Hasse *diagram*	67
діаграми Венна	диаграммы Венна	Venn's *diagrams*	48
діаметр (графа)	диаметр (графа)	diameter (of graph)	122
диз'юнктивна нормальна форма	дизъюнктивная нормальная форма	disjunctive normal form	16
диз'юнкція	дизъюнкция	disjunction	8
принцип *Дирихле*	принцип *Дирихле*	pigeonhole principle	73
дискретний	дискретный	discrete	6
дистрибутивність ("∧", "∨")	дистрибутивность ("∧", "∨")	distributivity ("∧", "∨")	14
диявольський стан	дьявольское состояние	diablo state	173
ДКНФ	СКНФ	CCNF	16
ДНФ	ДНФ	DNF	16
правило *добутку*	правило *произведения*	rule of *product*	93
добуток автоматів	произведение автоматов	automata *product*	173
довжина маршруту	длина́ маршрута	walk *length*	121
довжина ребра́	длина́ ребра́	edge's weight	113
довжина сло́ва	длина́ сло́ва	string's *length*	162
доповнення	дополнение	complement	49
допускати слово	допускать слово	to *accept* string	171
допусковий стан	допускающее состояние	accept state	171
досконала диз'юнктивна нормальна форма	совершенная дизъюнктивная нормальная форма	canonical disjunctive normal form	16
досконала кон'юнктивна нормальна форма	совершенная конъюнктивная нормальная форма	canonical conjunctive normal form	16
дося́жність	достижи́мость	reachability	121
матриця *досяжності*	матрица *достижимости*	reachability matrix	121
дуга (орграфа)	дуга (орграфа)	arc (of digraph)	112
дужкові вирази (підрахунок кількості)	скобочные выражения (подсчёт количеств)	parenthis expresession (counting quantity)	103
ЕД	ЭД	ED	16
круги *Ейлера*	круги *Эйлера*	Euler's diagrams	48
теорема *Ейлера* (про ейлерів цикл)	теорема *Эйлера* (об эйлеровом цикле)	Euler's theorem (about Eulerian cycle)	127
ейлерів шлях (ланцюг)	эйлеров путь (эйлерова цепь)	Eulerian path (trail)	127
ЕК	ЭК	EC	16
еквівалентні за модулем p	эквивалентны по модулю p	equivalent mod p	64
k-еквівалентні стани	*k-эквивалентные* состояния	*k-equivalent* states	184
відношення *еквівалентності*	отношение *эквивалентности*	equivalence relation	63
клас *еквівалентності*	класс *эквивалентности*	equivalence class	64
еквіваленція	эквиваленция	iff	8
алгоритм *Евкліда*	алгоритм *Евклида*	Euclid's algorithm	87
ексцентриси́тет (вершини)	эксцентриси́тет (вершины)	eccentricity (of vertex)	122

елемент	элемент	element 46
елементарна диз'юнкція	элементарная дизъюнкция	elementary disjunction 16
елементарна кон'юнкція	элементарная конъюнкция	elementary conjunction 16
енка	энка	*n-tuple* 55
ε-перехід	ε-переход	ε-move 176
полінóм *Жегалкіна*	полинóм *Жегалкина*	Zhegalkin polynomial 18
квантор *загальності*	квантор *всеобщности*	universal quantifier 31
заключний стан	заключительное состояние	final state 171
закон виключення третього	закон исключения третьего	law of the excluded middle 14
закон заперечення заперечення	закон отрицания отрицания	law of double negation 14
закон контрапозиції імплікації	закон контрапозиции импликации	law of contraposition 14
закон подвійного заперечення	закон двойного отрицания	law of double negation 14
закони де Моргана	законы де Моргана	DeMorgan's *laws* 14
замикання відношення	замыкание отношения	closure of relation 71
замкнений маршрут	замкнутый маршрут	closed walk 120
заперечення	отрицание	negation 8
закон *заперечення заперечення*	закон *отрицания отрицания*	double negation law 14
зважений граф	взвешенный граф	weighted graph 113
зворотній хід (відновлення шляху)	обратный ход (восстановление пути)	backtrace (restoring path) 131
зв'язана змінна	связанная переменная	bound variable 32
зв'я́зний граф	связный граф	connected graph 123
вершинно *k-зв'я́зний* граф	вершинно *k-свя́зный* граф	*k*-vertex-*connected* graph 126
ре́берно *k-зв'я́зний* граф	рёберно *k-свя́зный* граф	*k*-edge-*connected* graph 126
компонента *зв'я́зності*	компонента *связности*	connected component 124
зірочка Кліні	звёздочка Клини	Kleene star 163
злиття	слияние	merge 53
зліче́нна множина	счётное множество	countable set 75
точка *зчленува́ння*	точка *сочлене́ния*	articulation point 125
і	и	and 8
ідемпотентність ("$\vee$", "$\wedge$")	идемпотентность ("$\vee$", "$\wedge$")	idempotence ("$\vee$", "$\wedge$") 14
ізольована вершина	изолированная вершина	isolated vertex 114
ізоморфні графи	изоморфные графы	isomorphic graphs 115
імплікантна таблиця	импликантная таблица	prime implicant chart 25
імплікація	импликация	implication 8
інваріант графа	инвариант графа	graph *invariant* 115
інваріант циклу	инвариант цикла	loop *invariant* 83
індійський алгоритм піднесення до степеню	индийский алгоритм возведения в степень	repeating squaring 88
метод математичної *індукції*	метод математической *индукции*	mathematical *induction* 81
ініціальний автомат	инициальный автомат	initial automaton (FSM) 170
інфіксний запис відношень	инфиксная запись отношений	infix notation for relations 58
матриця *інциденцій*	матрица *инциденций*	incidence matrix 117
ін'єкція (ін'єктивна функція)	инъекция (инъективная функция)	injection (injective function) 73
іррефлексивне відношення	иррефлексивное отношение	irreflexive relation 61
квантор *існування*	квантор *существования*	existential quantifier 32
істина	истина	true 8
таблиця *істинності* (побудова)	таблица *истинности* (построение)	truth table (constructing) 12
таблиця *істинності* (стандартна)	таблица *истинности* (стандартная)	truth table (standard) 8
ітерація	итерация	Kleene star 163
теорема *Кантора*	теорема *Кантора*	Cantor's theorem 74
каркасне (остовне) дерево	о́стовное дерево	spanning tree 146

каркасне (остовне) дерево мінімальної ваги	о́стовное дерево минимального веса	minimum weight *spanning tree*	147
каркасний (остовний) підграф	о́стовный подграф	spanning subgraph	116
карта Карно	карта Карно	Karnaugh *map*	20
числа *Каталана*	числа *Каталана*	Catalan numbers	104
метод *Квайна* (перевірки тавтологічності)	метод *Квайна* (проверки тавтологичности)	Quine's method (tautology test)	26
метод *Квайна–Мак-Класкі*	метод *Квайна–Мак-Класки*	Quine–McCluskey method	23
квантор загальності	квантор (все)общности	universal *quantifier*	31
квантор існування	квантор существования	existential *quantifier*	32
кінець дуги	конец дуги	head of arc	112
кінцева́ (висяча) вершина	концева́я (висячая) вершина	pendant vertex (leaf)	114
кінці неорієнтованого ребра	концы неориентированного ребра	endpoints of undirected edge	112
кістяковий (остовний, каркасний, стяжний) підграф	о́стовный подграф	spanning subgraph	116
клас еквівалентності	класс эквивалентности	equivalence *class*	64
класи Поста (T_0, T_1, S, L, M)	классы Поста (T_0, T_1, S, L, M)	Post's classes (T_0, T_1, S, L, M)	37
зірочка *Кліні*	звёздочка *Клини*	Kleene star	163
КНФ	КНФ	CNF	16
метод невизначених *коефіцієнтів*	метод неопределённых *коэффициентов*	method of undetermined *coefficients*	19
комбінації з n по k	сочетания из n по k	k-combinations of n-element set	96
композиція відношень	композиция отношений	composition of relations	59
компонента двозв'я́зності	компонента двусвя́зности	biconnected component	126
компонента зв'я́зності	компонента связности	connected *component*	124
компонента, сильна (сильної зв'я́зності)	компонента, сильная (сильной свя́зности)	component, strong (strongly connected)	124
компонента, слабка (слабкої зв'я́зності)	компонента, слабая (слабой свя́зности)	component, weak (weakly connected)	124
комутативність ("$\vee$", "$\wedge$")	коммутативность ("$\vee$", "$\wedge$")	commutativity ("$\vee$", "$\wedge$")	14
конденсація графа	конденсация графа	condense of graph	125
конкатенація	конкатенация	concatenation	163
континуа́льна множина	континуа́льное множество	continuum	75
закон *контрапозиції* імплікації	закон *контрапозиции* импликации	contrapositive law	14
контрприклад	контрпример	counterexample	31
кон'юнктивна нормальна форма	конъюнктивная нормальная форма	conjunctive normal form	16
кон'юнкція	конъюнкция	conjunction	8
крок індукції	шаг индукции	inductive *step*	81
круги Ейлера	круги Ейлера	Euler's *diagrams*	48
ксор	ксор	xor	8
ланцюг	цепь	trail	120
лема про накачку	лемма о накачке	pumping lemma	175
лема про рукостискання	лемма о рукопожатиях	handshaking lemma	113
лінійне відношення	линейное отношение	linear relation	61
відношення *лінійного порядку*	отношение *линейного порядка*	total order	67
ліс	лес	forest	146
максимальний елемент	максимальный элемент	a *maximal* element	67
маршрут	маршрут	route	119
матіндукція	матиндукция	math induction	81
матриця бінарного відношення	матрица бинарного отношения	binary relation matrix	58
матриця відстаней	матрица расстояний	distance *matrix*	122
матриця досяжності	матрица достижимости	reachability *matrix*	121
матриця інциденцій	матрица инциденций	incidence *matrix*	117

матриця суміжності	матрица смежности	adjacency *matrix*	117
мережа (зважений граф)	сеть (взвешенный граф)	network (weighted graph)	113
метод Квайна (перевірки тавтологічності)	метод Квайна (проверки тавтологичности)	Quine's *method* (tautology test)	26
метод Квайна–Мак-Класкі	метод Квайна–Мак-Класки	Quine–McCluskey *method*	23
метод математичної індукції	метод математической индукции	mathematical induction	81
метод невизначених коефіцієнтів	метод неопределённых коэффициентов	method of undetermined coefficients	19
метод редукції	метод редукции	tautology test by contradiction	27
метод семантичних дерев	метод семантических деревьев	semantic tree *method*	27
автомат *Мілі*	автомат *Мили*	Mealy machine	182
мінімальний елемент	минимальный элемент	a *minimal* element	68
мінімізація автоматів	минимизация автоматов	minimization of automata	184
мінімізація булевих функцій	минимизация булевых функций	minimization of boolean functions	20
міра множини	мера множества	set *cardinality*	98
міст	мост	bridge	125
множина́	мно́жество	set	46
регулярна *мова*	регулярный *язык*	regular *language*	163
формальна *мова*	формальный *язык*	formal *language*	162
МОД	МОД	MST	147
закони де *Моргана*	законы де *Моргана*	De*Morgan*'s laws	14
мультиграф	мультиграф	multigraph	113
мультимножина	мультимножество	multiset	47
автомат *Му́ра*	автомат *Му́ра*	Moore machine	181
функція *на*	функция *на*	function *onto*	73
найбільший елемент	наибольший элемент	the *greatest* element	68
найменший елемент	наименьший элемент	the *least* element	68
лема про *накачку*	лемма о *накачке*	pumping lemma	175
напівстепінь виходу	полустепень исхо́да	out-degree	113
напівстепінь захо́ду (входу)	полустепень захо́да	in-degree	113
не	не	not	8
не порі́внювані	не сравни́мы	non-comparable	67
метод *невизначених коефіцієнтів*	метод *неопределённых коэффициентов*	method of *undetermined coefficients*	19
недетермінований автомат	недетерминированный автомат	nondeterministic automaton	175
незважений граф	невзвешенный граф	unweighted graph	113
неорієнтований граф	неориентированный граф	undirected graph	112
неповний вигляд автомата	неполный вид автомата	partial automaton	173
відношення *нестрогого порядку*	отношение *нестрогого порядка*	weak order	66
ε-НСА	ε-НКА	ε-NFA	176
НСА	НКА	NFA	176
біном *Ньютона*	бином *Ньютона*	binomial expansion	100
обернена функція	обратная функция	inverse function	73
обернене відношення	обратное отношение	inverse relation	59
область визначення	область определения	domain	59
область значень	область значений	codomain	59
обхід графа	обход гра́фа	graph traversal	129
обхід у глибину (вглиб)	обход в глубину	depth-first *traversal*	134
обхід у ширину (вшир)	обход в ширину	breadth-first *traversal*	129
об'єднання	объединение	union	48
об'єднання	объединение	union	162
ОДМВ	ОДМВ	MWST	147
односторонньо зв'язний орграф	односторонне связный орграф	semiconnected digraph	124
орграф (орієнтований граф)	орграф (ориентированный граф)	digraph (directed graph)	112
орграф, односторонньо зв'язний	орграф, односторонне связный	digraph, semiconnected	124

орграф, сильно зв'язний	орграф, сильно связный	digraph, strongly connected	124
орграф, слабко зв'язний	орграф, слабо связный	digraph, weakly connected	124
орієнтований граф	ориентированный граф	directed graph	112
остовне (каркасне) дерево	о́стовное дерево	spanning tree	146
остовне (каркасне) дерево мінімальної ваги	о́стовное дерево минимального веса	minimum weight *spanning tree*	147
остовний (каркасний) підграф	о́стовный подграф	spanning subgraph	116
трикутник *Паскаля*	треугольник *Паскаля*	Pascal's triangle	100
передісторія	предыстория	prehistory	170
перестановки	перестановки	permutations	95
перестановки з повтореннями	перестановки с повторениями	permutations of multiset	96
автомат-*перетворювач*	автомат-*преобразователь*	finite-state *transducer*	180
перетин	пересечение	intersection	48
ε-перехід	*ε-переход*	*ε-move*	176
функція *переходів*	функция *переходов*	state-*transition* function	171
петля	петля	self-loop	112
підграф	подграф	subgraph	116
підграф, породжений множиною вершин	подграф, порождённый множеством вершин	subgraph induced by vertices set	116
підмножина	подмножество	subset	47
власна *підмножина*	собственное *подмножество*	proper *subset*	48
стрілка *Пірса*	стрелка *Пирса*	Peirce arrow	38
побітові операції	побитовые операции	bitwise operators	10
функціонально *повна* система	функционально *полная* система	functionally *complete* set	36
повне відношення	полное отношение	total relation	61
повний вигляд автомата	полный вид автомата	total automaton	173
повний граф	полный граф	complete graph	114
повний дводольний граф	полный двудольный граф	complete bipartite graph	114
подання бінарного відношення матрицею	представление бинарного отношения матрицей	matrix *presentation* of binary relation	58
подання множини характеристичним предикатом	задание множества характеристическим предикатом	set-builder notation with predicate	46
подання предикатів	представление предикатов	presentation of predicates	30
закон *подвійного заперечення*	закон *двойного отрицания*	double negation law	14
полінóм Жегалкіна	полинóм Жегалкина	Zhegalkin *polynomial*	18
порі́внювані та не порівнювані елементи	сравни́мые и не сравнимые элементы	comparable and non-comparable elements	67
порі́внювані за модулем p	сравни́мы по модулю p	equivalent mod p	64
підграф, *породжений* множиною вершин	подграф, *порождённый* множеством вершин	subgraph *induced* by vertices set	116
порожнє слово	пустое слово	empty string	162
порожня множина	пустое множество	empty set	47
відношення *порядку*	отношение *порядка*	order relation	66
класи *Поста* (T_0, T_1, S, L, M)	классы *Поста* (T_0, T_1, S, L, M)	Post's classes (T_0, T_1, S, L, M)	37
поточна множина станів (НСА)	текущее множество состояний (НКА)	current set of states (NFA)	176
поточний стан автомата	текущее состояние автомата	current state of automaton	170
початковий стан	начальное состояние	initial state	170
початок дуги	начало дуги	tail of arc	112
пошук у глибину (вглиб)	поиск в глубину	depth-first *search*	134
пошук у гра́фі	поиск в гра́фе	graph search	129
пошук у ширину (вшир)	поиск в ширину	breadth-first *search*	129
правило суми	правило суммы	rule of sum	93
правило добутку	правило произведения	rule of product	93
«правило частки»	«правило частного»	«rule of division»	96
правильний підграф	правильный подграф	full subgraph	116

предикат	предикат	predicate 29
представник класу еквівалентності	представитель класса эквивалентности	representative of equivalence class 66
принцип включень та виключень	принцип включений и исключений	inclusion–exclusion *principle* 98
принцип Дирихле	принцип Дирихле	pigeonhole *principle* 73
пріоритети логічних операцій	приоритеты логических операций	precedence of logical connectives 9
простий граф	простой граф	simple graph 113
простий ланцюг	простая цепь	simple path 120
простий цикл	простой цикл	simple cycle 120
радіус (графа)	радиус (графа)	radius (of graph) 123
список *ребер*	список *рёбер*	edge list 119
ре́берна двозв'я́зність	рёберная двусвя́зность	edge biconnectedness 126
ре́берно k-зв'я́зний граф	рёберно k-свя́зный граф	k-edge-connected graph 126
ребро	ребро	edge 112
регулярна мова	регулярный язык	regular language 163
регулярний вираз	регулярное выражение	regular expression 164
регулярний (однорідний) граф	регулярный (однородный) граф	regular graph 114
метод *редукції*	метод *редукции*	reduction method 27
рекурентні співвідношень	рекуррентные соотношения	recurrence relations 101
релаксація оцінки відстані	релаксация оценки расстояния	relaxation of shortest-path estimate 141
рефлексивне відношення	рефлексивное отношение	reflexive relation 61
рівнопотужні множи́ни	равномощные множества	equipotent sets 73
різниця	разность	difference 49
розбиття	разбиение	partition 65
розбиття	разбиение	partition 124
розміщення з n по k	размещения из n по k	k-permutations of n-element sequence 95
розміщення з повтореннями з n по k	размещения с повторениями из n по k	strings 97
розпізнавати мову	распознавать язык	to *recognize* language 171
автомат-*розпізнавач*	автомат-*распознаватель*	recognizer 170
розріджений граф	разреженный граф	sparse graph 118
розширення відношення порядку до лінійного	расширение отношения порядка до линейного	linear *extention* of order 68
лема про *рукостискання*	лемма о *рукопожатиях*	handshaking lemma 113
метод *семантичних дерев*	метод *семантических деревьев*	semantic tree method 27
сильна компонента	сильна компонента	strong component 124
сильно зв'язний орграф	сильно связный орграф	strongly connected digraph 124
символ	символ	symbol 162
симетрична різниця	симметрическая разность	symmetric difference 49
симетричне відношення	симметричное отношение	symmetric relation 61
склеювання ЕК	склеивание ЭК	combining EC 23
скорочена ДНФ	сокращённая ДНФ	prime implicants 25
скорочене обчислення логічних виразів	сокращённое вычисление логических выражений	short-circuit evaluation of boolean expressions 11
слабка компонента	слабая компонента	weak component 124
слабко зв'язний орграф	слабо связный орграф	weakly connected digraph 124
слово	слово	string 162
слово вхідне	слово входное	input *string* 170
порожнє *слово*	пустое *слово*	empty *string* 162
сортування вибором	сортировка выбором	selection *sort* 84
сортування вставками	сортировка вставками	insertion *sort* 85
списки суміжності	списки смежности	adjacency *lists* 118

список ребер	список рёбер	edge *list* 119
сполучення з n по k	сочетания из n по k	k-combinations of n-element set 96
сполучення з повтореннями з n по k	сочетания с повторениями из n по k	k-combinations with repetitions of n-element set 97
стан (автомата)	состояние (автомата)	state (of automaton) 170
статус (стан) вершини	статус (состояние) вершины	vertex *status (state)* 129
степінь вершини	степень вершины	vertex *degree* 113
степінь відношення	степень відношення	power of relation 60
степінь мови	степень языка	power of language 163
стрілка Пірса	стрелка Пирса	Peirce *arrow* 38
відношення *строгого порядку*	отношение *строгого порядка*	strict order 66
стяжний (остовний, каркасний, кістяковий) підграф	о́стовный подграф	spanning subgraph 116
правило *суми*	правило *суммы*	rule of *sum* 93
матриця *суміжності*	матрица *смежности*	adjacency matrix 117
списки *суміжності*	списки *смежности*	adjacency lists 118
суперпозиція відношень	суперпозиция отношений	composition of relations 59
сюр'єкція (сюр'єктивна функція)	сюръекция (сюръективная функция)	surjection (surjective function) 73
таблиця істинності (побудова)	таблица истинности (построение)	truth *table* (constructing) 12
таблиця істинності (стандартна)	таблица истинности (стандартная)	truth *table* (standard) 8
таблиця поглинань	таблица поглощений	prime implicant *chart* 25
тавтологія	тавтология	tautology 26
теорема Дірака	теорема Дирака	Dirac's theorem 128
теорема Ейлера (про ейлерів цикл)	теорема Эйлера (об эйлеровом цикле)	Euler's *theorem* (about Eulerian cycle) 127
теорема Кантора	теорема Кантора	Cantor's *theorem* 74
теорема Кліні	теорема Клини	Kleene theorem 174
теорема Поста про функціональну повноту	теорема Поста о функциональной полноте	Post's functional completeness theorem 38
тернарний	тернарный	ternary 9
топологі́чне сортування відношення порядку	топологи́ческая сортировка отношения порядка	topological sorting of order 69
топологі́чне сортування орграфа	топологи́ческая сортировка орграфа	topological sort (topsort) of digraph 137
тор	тор	torus 20
тотожнє відношення	тождественное отношение	equality relation 60
точка зчленува́ння	точка сочлене́ния	articulation *point* 125
транзитивне відношення	транзитивное отношение	transitive relation 61
тризначна логіка	трёхзначная логика	three-value logic 39
трикутник Паскаля	треугольник Паскаля	Pascal's *triangle* 100
унарний	унарный	unary 9
універсум	универсум	universe 47
фактор-множина	фактор-множество	quotient set 66
фіктивна змінна	фиктивная переменная	non-essential variable 13
алгоритм *Флойда*	алгоритм *Флойда*	Floyd's algorithm 143
формальна мова	формальный язык	formal language 162
функціонально повна система	функционально полная система	functionally complete set 36
функція	функция	function 72
функція переходів	функция переходов	state-transition function 171
подання множини *характеристичним предикатом*	задание множества *характеристическим предикатом*	set-builder notation with predicate 46
хвіст черги	хвост очереди	tail of queue 129
хиба	ложь	false 8
центр (графа)	центр (графа)	center (of graph) 123

центральна вершина	центральная вершина	central vertex	123
цикл	цикл	tour	120
черга	очередь	queue	129
числа Каталана	числа Каталана	Catalan *numbers*	104
чорна діра	чёрная дыра	black hole	173
швидке піднесення до степеню (ітеративне)	быстрое возведение в степень (итерационное)	exponentiation by squaring (iterative)	88
швидке піднесення до степеню (рекурсивне)	быстрое возведение в степень (рекурсивное)	exponentiation by squaring (recursive)	90
пошук у *ширину* (вшир)	поиск в *ширину*	breadth-first search	129
шлях	путь	walk	119
штрих Шеффера	штрих Шеффера	Sheffer *stroke*	38
щільний граф	плотный граф	dense graph	118
явний перелік	явный перечень	explicit enumeration	46
ε-НСА	ε-НКА	ε-NFA	176
ε-перехід	ε-переход	ε-move	176
$\overline{C}_n^k$	$\overline{C}_n^k$	$\left(\!\binom{n}{k}\!\right)$	97
A_n^k	A_n^k	$[n]_k$	95
C_n^k	C_n^k	$\binom{n}{k}$	96
k-еквівалентні стани	k-эквивалентные состояния	k-equivalent states	184
k-зв'я́зний (вершинно) граф	k-свя́зный (вершинно) граф	k-vertex-connected graph	126
k-зв'я́зний (реберно) граф	k-свя́зный (реберно) граф	k-edge-connected graph	126
n-арний	n-арный	n-ary	9
n-ка	n-ка	n-tuple	55

Російсько-український-англійський словник та покажчик

автомат Ми́ли	автомат Мі́лі	Mealy machine	182
автомат Му́ра	автомат Му́ра	Moore machine	181
автомат-преобразователь	автомат-перетворювач	finite-state transducer	180
автомат-распознаватель	автомат-розпізнавач	recognizer	170
алгоритм Ауфенкампа–Хона	алгоритм Ауфенкампа–Хона	Aufenkamp–Hohn algorithm	184
алгоритм Дейкстры	алгоритм Дейкстри	Dijkstra's *algorithm*	141
алгоритм Евклида	алгоритм Евкліда	Euclid's *algorithm*	87
алгоритм Уоршалла	алгоритм Воршалла	Warshall's *algorithm*	143
алгоритм Флойда	алгоритм Флойда	Floyd's *algorithm*	143
входной *алфавит*	вхідний *алфавіт*	input *alphabet*	170
антисимметричное отношение	антисиметричне відношення	antisymmetric relation	61
ассоциативность ("$\vee$", "$\wedge$")	асоціативність ("$\vee$", "$\wedge$")	associativity ("$\vee$", "$\wedge$")	14
базис (база) индукции	база індукції	base case (basis)	81
биекция (биективная функция)	бієкція (бієктивна функція)	bijection (injective function)	73
бинарное «в узком смысле» отношение	бінарне «у вузькому смислі» відношення	binary relation on A	58
бинарное «в широком смысле» отношение	бінарне «у широкому смислі» відношення	binary relation between A and B	58
бинарное отношение	бінарне відношення	binary relation	58
бинарное отношение в $A \times B$	бінарне відношення в $A \times B$	binary relation between A and B	58
бинарное отношение на A	бінарне відношення на A	binary relation on A	58
бинарный	бінарний	binary	9
бином Ньютона	біном Ньютона	binomial expansion	100
биномиальные коэффициенты	біноміальні коефіцієнти	binomial coefficients	99
блок	блок	block	126
булеа́н множества	булеа́н множини	power set of set	49

быстрое возведение в степень (итерационное)	швидке піднесення до степеню (ітеративне)	exponentiation by squaring (iterative)	88
быстрое возведение в степень (рекурсивное)	швидке піднесення до степеню (рекурсивне)	exponentiation by squaring (recursive)	90
бэктрэкинг (отсечения при переборе)	бектрекінг (відтинання при переборі)	back-tracking	115
диаграмма *Вейча*	діаграма *Вейча*	Karnaugh map (K-map)	20
диаграммы *Венна*	діаграми *Венна*	Venn's diagrams	48
вершина	вершина	vertex	112
вершинная двусвя́зность	вершинна двозв'я́зність	vertex biconnectedness	126
вершинно k-свя́зный граф	вершинно k-зв'я́зний граф	k-vertex-connected graph	126
взвешенный граф	зважений граф	weighted graph	113
висячая (концева́я) вершина	висяча (кінцева́) вершина	leaf (pendant vertex)	114
принцип *включений и исключений*	принцип *включень та виключень*	inclusion–exclusion principle	98
восстановление пути (обратный ход)	відновлення шляху (зворотній хід)	restoring path (backtrace)	131
квантор *всеобщности*	квантор *загальності*	universal quantifier	31
сортировка *вставками*	сортування *вставками*	insertion sort	85
входное слово	вхідне слово	input string	170
входной алфавит	вхідний алфавіт	input alphabet	170
сортировка *выбором*	сортування *вибором*	selection sort	84
регулярное *выражение*	регулярний *вираз*	regular *expression*	164
гамильтонов цикл	гамільтонів цикл	Hamiltonian cycle	128
гиперграф	гіперграф	hypergraph	117
поиск в *глубину*	пошук у *глибину* (вглиб)	depth-first search	134
голова очереди	голова черги	head of queue	129
граф	граф	graph	112
граф конденсации	граф конденсації	condense of graph	125
закон *двойного отрицания*	закон *подвійного заперечення*	double negation law	14
двудольный граф	дводольний (двочастковий) граф	bipartite graph	114
двусвя́зная компонента	двозв'я́зна компонента	biconnected component	126
двусвя́зность графа	двозв'я́зність графа	biconnectedness of graph	126
законы *де Моргана*	закони *де Моргана*	DeMorgan's laws	14
алгоритм *Дейкстры*	алгоритм *Дейкстри*	Dijkstra's algorithm	141
декартова степень	декартів степінь	Cartesian power	55
декартово произведение	декартів добуток	Cartesian product	55
дерево (неориентированное)	дерево (неорієнтоване)	tree (undirected)	146
метод семантических *деревьев*	метод семантичних *дерев*	semantic tree method	27
детерминизация	детермінізація	powerset construction	179
диагональный метод	діагональний метод	diagonal argument	74
диаграмма Вейча	діаграма Вейча	Karnaugh map (K-map)	20
диаграмма Хассе	діаграма Гассе	Hasse *diagram*	67
диаграммы Венна	діаграми Венна	Venn's *diagrams*	48
диаметр (графа)	діаметр (графа)	diameter (of graph)	122
дизъюнктивная нормальная форма	диз'юнктивна нормальна форма	disjunctive normal form	16
дизъюнкция	диз'юнкція	disjunction	8
принцип *Дирихле*	принцип *Дирихле*	pigeonhole principle	73
дискретный	дискретний	discrete	6
дистрибутивность ("$\wedge$", "$\vee$")	дистрибутивність ("$\wedge$", "$\vee$")	distributivity ("$\wedge$", "$\vee$")	14
длина́ маршрута	довжина маршруту	walk *length*	121
длина́ ребра́	довжина ребра́	edge's weight	113
длина́ сло́ва	довжина сло́ва	string's *length*	162
ДНФ	ДНФ	DNF	16
дополнение	доповнення	complement	49

допускать слово	допускати слово	to *accept* string	171
допускающее состояние	допусковий стан	accept state	171
матрица *достижимости*	матриця *досяжності*	reachability matrix	121
достижи́мость	дося́жність	reachability	121
дуга (орграфа)	дуга (орграфа)	arc (of digraph)	112
дьявольское состояние	диявольський стан	diablo state	173
алгоритм *Евклида*	алгоритм *Евкліда*	Euclid's algorithm	87
полино́м *Жегалкина*	полі́ном *Жегалкіна*	Zhegalkin polynomial	18
задание множества характеристическим предикатом	подання множини характеристичним предикатом	set-builder notation with predicate	46
заключительное состояние	заключний стан	final state	171
закон двойного отрицания	закон подвійного заперечення	law of double negation	14
закон исключения третьего	закон виключення третього	law of the excluded middle	14
закон контрапозиции импликации	закон контрапозиції імплікації	law of contraposition	14
закон отрицания отрицания	закон заперечення заперечення	law of double negation	14
законы де Моргана	закони де Моргана	DeMorgan's *laws*	14
замкнутый маршрут	замкнений маршрут	closed walk	120
замыкание отношения	замикання відношення	closure of relation	71
звёздочка Клини	зірочка Кліні	Kleene star	163
и	і	and	8
идемпотентность (“$\vee$”, “$\wedge$”)	ідемпотентність (“$\vee$”, “$\wedge$”)	idempotence (“$\vee$”, “$\wedge$”)	14
изолированная вершина	ізольована вершина	isolated vertex	114
изоморфные графы	ізоморфні графи	isomorphic graphs	115
или	або	or	8
импликантная таблица	імплікантна таблиця	prime implicant chart	25
импликация	імплікація	implication	8
инвариант графа	інваріант графа	graph *invariant*	115
инвариант цикла	інваріант циклу	loop *invariant*	83
индийский алгоритм возведения в степень	індійський алгоритм піднесення до степеню	repeating squaring	88
метод математической *индукции*	метод математичної *індукції*	mathematical *induction*	81
инициальный автомат	ініціальний автомат	initial automaton (FSM)	170
инфиксная запись отношений	інфіксний запис відношень	infix notation for relations	58
матрица *инциденций*	матриця *інциденцій*	incidence matrix	117
инъекция (инъективная функция)	ін'єкція (ін'єктивна функція)	injection (injective function)	73
иррефлексивное отношение	іррефлексивне відношення	irreflexive relation	61
исключающее или	виключне або	exclusive or	8
истина	істина	true	8
таблица *истинности* (построение)	таблиця *істинності* (побудова)	truth table (constructing)	12
таблица *истинности* (стандартная)	таблиця *істинності* (стандартна)	truth table (standard)	8
итерация	ітерація	Kleene star	163
теорема *Кантора*	теорема *Кантора*	Cantor's theorem	74
карта Карно	карта Карно	Karnaugh *map*	20
числа *Каталана*	числа *Каталана*	Catalan numbers	104
метод *Квайна* (проверки тавтологичности)	метод *Квайна* (перевірки тавтологічності)	Quine's method (tautology test)	26
метод *Квайна–Мак-Класки*	метод *Квайна–Мак-Класкі*	Quine–McCluskey method	23
квантор (все)общности	квантор загальності	universal *quantifier*	31
квантор существования	квантор існування	existential *quantifier*	32
класс эквивалентности	клас еквівалентності	equivalence *class*	64
классы Поста (T_0, T_1, S, L, M)	класи Поста (T_0, T_1, S, L, M)	Post's classes (T_0, T_1, S, L, M)	37
звёздочка *Клини*	зірочка *Кліні*	Kleene star	163

КНФ	КНФ	CNF 16
коммутативность (“∨”, “∧”)	комутативність (“∨”, “∧”)	commutativity (“∨”, “∧”) 14
композиция отношений	композиція відношень	composition of relations 59
компонента двусвя́зности	компонента двозв'я́зності	biconnected component 126
компонента связности	компонента зв'я́зності	connected *component* 124
компонента, сильная (сильной свя́зности)	компонента, сильна (сильної зв'я́зності)	component, strong (strongly connected) 124
компонента, слабая (слабой свя́зности)	компонента, слабка (слабкої зв'я́зності)	component, weak (weakly connected) 124
конденсация графа	конденсація графа	condense of graph 125
конец дуги	кінець дуги	head of arc 112
конкатенация	конкатенація	concatenation 163
континуа́льное множество	континуа́льна множина	continuum 75
закон *контрапозиции* импликации	закон *контрапозиції* імплікації	contrapositive law 14
контрпример	контрприклад	counterexample 31
концева́я (висячая) вершина	кінцева́ (висяча) вершина	pendant vertex (leaf) 114
концы неориентированного ребра	кінці неорієнтованого ребра	endpoints of undirected edge 112
конъюнктивная нормальная форма	кон'юнктивна нормальна форма	conjunctive normal form 16
конъюнкция	кон'юнкція	conjunction 8
метод неопределённых *коэффициентов*	метод невизначених *коефіцієнтів*	method of undetermined *coefficients* 19
круги Ейлера	круги Ейлера	Euler's *diagrams* 48
ксор	ксор	xor 8
лемма о накачке	лема про накачку	pumping lemma 175
лемма о рукопожатиях	лема про рукостискання	handshaking lemma 113
лес	ліс	forest 146
отношение *линейного порядка*	відношення *лінійного порядку*	total order 67
линейное отношение	лінійне відношення	linear relation 61
ложь	хиба	false 8
максимальный элемент	максимальний елемент	a *maximal* element 67
маршрут	маршрут	route 119
матиндукция	матіндукція	math induction 81
матрица бинарного отношения	матриця бінарного відношення	binary relation matrix 58
матрица достижимости	матриця досяжності	reachability *matrix* 121
матрица инциденций	матриця інциденцій	incidence *matrix* 117
матрица расстояний	матриця відстаней	distance *matrix* 122
матрица смежности	матриця суміжності	adjacency *matrix* 117
мера множества	міра множини	set *cardinality* 98
метод Квайна (проверки тавтологичности)	метод Квайна (перевірки тавтологічності)	Quine's *method* (tautology test) 26
метод Квайна–Мак-Класки	метод Квайна–Мак-Класкі	Quine–McCluskey *method* 23
метод математической индукции	метод математичної індукції	mathematical induction 81
метод неопределённых коэффициентов	метод невизначених коефіцієнтів	method of undetermined coefficients 19
метод редукции	метод редукції	tautology test by contradiction 27
метод семантических деревьев	метод семантичних дерев	semantic tree *method* 27
автомат *Ми́ли*	автомат *Мі́лі*	Mealy machine 182
минимальный элемент	мінімальний елемент	a *minimal* element 68
минимизация автоматов	мінімізація автоматів	minimization of automata 184
минимизация булевых функций	мінімізація булевих функцій	minimization of boolean functions 20
мно́жество	множина́	set 46
МОД	МОД	MST 147
законы де *Моргана*	закони де *Моргана*	De*Morgan*'s laws 14

мост	міст	bridge 125
мультиграф	мультиграф	multigraph 113
мультимножество	мультимножина	multiset 47
автомат *Му́ра*	автомат *Му́ра*	Moore machine 181
функция *на*	функція *на*	function *onto* 73
наибольший элемент	найбільший елемент	the *greatest* element 68
наименьший элемент	найменший елемент	the *least* element 68
лемма о *накачке*	лема про *накачку*	pumping lemma 175
начало дуги	початок дуги	tail of arc 112
начальное состояние	початковий стан	initial state 170
не	не	not 8
не сравни́мы	не пори́внювані	non-comparable 67
невзвешенный граф	незважений граф	unweighted graph 113
недетерминированный автомат	недетермінований автомат	nondeterministic automaton 175
метод *неопределённых коэффициентов*	метод *невизначених коефіцієнтів*	method of *undetermined coefficients* 19
неориентированный граф	неорієнтований граф	undirected graph 112
неполный вид автомата	неповний вигляд автомата	partial automaton 173
отношение *нестрогого порядка*	відношення *нестрогого порядку*	weak order 66
ε-НКА	ε-НСА	ε-NFA 176
НКА	НСА	NFA 176
бином *Ньютона*	біном *Ньютона*	binomial expansion 100
область значений	область значень	codomain 59
область определения	область визначення	domain 59
обратная функция	обернена функція	inverse function 73
обратное отношение	обернене відношення	inverse relation 59
обратный ход (восстановление пути)	зворотній хід (відновлення шляху)	backtrace (restoring path) 131
обход в глубину	обхід у глибину (вглиб)	depth-first *traversal* 134
обход в ширину	обхід у ширину (вшир)	breadth-first *traversal* 129
обход гра́фа	обхід графа	graph traversal 129
квантор *общности*	квантор *загальності*	universal quantifier 31
объединение	об'єднання	union 48
объединение	об'єднання	union 162
ОДМВ	ОДМВ	MWST 147
односторонне связный орграф	односторонньо зв'язний орграф	semiconnected digraph 124
орграф (ориентированный граф)	орграф (орієнтований граф)	digraph (directed graph) 112
орграф, односторонне связный	орграф, односторонньо зв'язний	digraph, semiconnected 124
орграф, сильно связный	орграф, сильно зв'язний	digraph, strongly connected 124
орграф, слабо связный	орграф, слабко зв'язний	digraph, weakly connected 124
ориентированный граф	орієнтований граф	directed graph 112
о́стовное дерево	остовне (каркасне) дерево	spanning tree 146
о́стовное дерево минимального веса	остовне (каркасне) дерево мінімальної ваги	minimum weight *spanning tree* 147
о́стовный подграф	остовний (каркасний) підграф	spanning subgraph 116
отклонять слово	відхиляти слово	to *decline* string 171
отношение	відношення	relation 57
бинарное *отношение* в $A \times B$	бінарне *відношення* в $A \times B$	binary *relation* between A and B 58
бинарное *отношение* на A	бінарне *відношення* на A	binary *relation* on A 58
отношение порядка	відношення порядку	order *relation* 66
отношение эквивалентности	відношення еквівалентності	equivalence *relation* 63
отрицание	заперечення	negation 8
закон *отрицания отрицания*	закон *заперечення заперечення*	double negation law 14
отсечения при переборе	відтинання при переборі	back-tracking 115
очередь	черга	queue 129

треугольник *Паскаля*	трикутник *Паскаля*	Pascal's triangle	100
пересечение	перетин	intersection	48
перестановки	перестановки	permutations	95
перестановки с повторениями	перестановки з повтореннями	permutations of multiset	96
ε-переход	*ε-перехід*	*ε-move*	176
функция *переходов*	функція *переходів*	state-*transition* function	171
петля	петля	self-loop	112
стрелка *Пирса*	стрілка *Пірса*	Peirce arrow	38
плотный граф	щільний граф	dense graph	118
побитовые операции	побітові операції	bitwise operators	10
подграф	підграф	subgraph	116
подграф, порождённый множеством вершин	підграф, породжений множиною вершин	subgraph induced by vertices set	116
подмножество	підмножина	subset	47
собственное *подмножество*	власна *підмножина*	proper *subset*	48
поиск в глубину	пошук у глибину (вглиб)	depth-first *search*	134
поиск в гра́фе	пошук у гра́фі	graph search	129
поиск в ширину	пошук у ширину (вшир)	breadth-first *search*	129
полино́м Жегалкина	полі́ном Жегалкіна	Zhegalkin *polynomial*	18
функционально *полная* система	функціонально *повна* система	functionally *complete* set	36
полное отношение	повне відношення	total relation	61
полный вид автомата	повний вигляд автомата	total automaton	173
полный граф	повний граф	complete graph	114
полный двудольный граф	повний дводольний граф	complete bipartite graph	114
полустепень захо́да	напівстепінь захо́ду (входу)	in-degree	113
полустепень исхо́да	напівстепінь виходу	out-degree	113
подграф, *порождённый* множеством вершин	підграф, *породжений* множиною вершин	subgraph *induced* by vertices set	116
отношение *порядка*	відношення *порядку*	order relation	66
классы *Поста* (T_0, T_1, S, L, M)	класи *Поста* (T_0, T_1, S, L, M)	Post's classes (T_0, T_1, S, L, M)	37
правило произведения	правило добутку	rule of product	93
правило суммы	правило суми	rule of sum	93
«правило частного»	«правило частки»	«rule of division»	96
правильный подграф	правильний підграф	full subgraph	116
предикат	предикат	predicate	29
представитель класса эквивалентности	представник класу еквівалентності	representative of equivalence class	66
представление бинарного отношения матрицей	подання бінарного відношення матрицею	matrix *presentation* of binary relation	58
представление предикатов	подання предикатів	presentation of predicates	30
предыстория	передісторія	prehistory	170
автомат-*преобразователь*	автомат-*перетворювач*	finite-state *transducer*	180
принцип включений и исключений	принцип включень та виключень	inclusion–exclusion *principle*	98
принцип Дирихле	принцип Дирихле	pigeonhole *principle*	73
приоритеты логических операций	пріоритети логічних операцій	precedence of logical connectives	9
произведение автоматов	добуток автоматів	automata *product*	173
правило *произведения*	правило *добутку*	rule of *product*	93
простая цепь	простий ланцюг	simple path	120
простой граф	простий граф	simple graph	113
простой цикл	простий цикл	simple cycle	120
пустое множество	порожня множина	empty set	47
пустое слово	порожнє слово	empty string	162
путь	шлях	walk	119
равномощные множества	рівнопотужні множи́ни	equipotent sets	73

радиус (графа)	радіус (графа)	radius (of graph)	123
разбиение	розбиття	partition	65
разбиение	розбиття	partition	124
размещения из n по k	розміщення з n по k	k-permutations of n-element sequence	95
размещения с повторениями из n по k	розміщення з повтореннями з n по k	strings	97
разность	різниця	difference	49
разреженный граф	розріджений граф	sparse graph	118
автомат-*распознаватель*	автомат-*розпізнавач*	recognizer	170
распознавать язык	розпізнавати мову	to *recognize* language	171
расстоя́ние	ві́дстань	distance	121
матрица *расстояний*	матриця *відстаней*	distance matrix	122
расширение отношения порядка до линейного	розширення відношення порядку до лінійного	linear *extention* of order	68
список *рёбер*	список *ребер*	edge list	119
рёберная двусвя́зность	ре́берна двозв'я́зність	edge biconnectedness	126
рёберно k-свя́зный граф	ре́берно k-зв'я́зний граф	k-edge-connected graph	126
ребро	ребро	edge	112
регулярное выражение	регулярний вираз	regular expression	164
регулярный (однородный) граф	регулярний (однорідний) граф	regular graph	114
регулярный язык	регулярна мова	regular language	163
метод *редукции*	метод *редукції*	reduction method	27
рекуррентные соотношения	рекурентні співвідношень	recurrence relations	101
релаксация оценки расстояния	релаксація оцінки відстані	relaxation of shortest-path estimate	141
рефлексивное отношение	рефлексивне відношення	reflexive relation	61
лемма о *рукопожатиях*	лема про *рукостискання*	handshaking lemma	113
свободная переменная	вільна змінна	free variable	32
связанная переменная	зв'язана змінна	bound variable	32
компонента *связности*	компонента *зв'я́зності*	connected component	124
связный граф	зв'я́зний граф	connected graph	123
вершинно k-*свя́зный* граф	вершинно k-*зв'я́зний* граф	k-vertex-*connected* graph	126
рёберно k-*свя́зный* граф	ре́берно k-*зв'я́зний* граф	k-edge-*connected* graph	126
СДНФ	ДДНФ	CDNF	16
метод *семантических деревьев*	метод *семантичних дерев*	semantic tree method	27
сеть (взвешенный граф)	мережа (зважений граф)	network (weighted graph)	113
сильна компонента	сильна компонента	strong component	124
сильно связный орграф	сильно зв'язний орграф	strongly connected digraph	124
символ	символ	symbol	162
симметрическая разность	симетрична різниця	symmetric difference	49
симметричное отношение	симетричне відношення	symmetric relation	61
склеивание ЭК	склеювання ЕК	combining EC	23
СКНФ	ДКНФ	CCNF	16
скобочные выражения (подсчёт количеств)	дужкові вирази (підрахунок кількості)	parenthis expresession (counting quantity)	103
слабая компонента	слабка компонента	weak component	124
слабо связный орграф	слабко зв'язний орграф	weakly connected digraph	124
слияние	злиття	merge	53
слово	слово	string	162
слово входное	слово вхідне	input *string*	170
пустое *слово*	порожнє *слово*	empty *string*	162
матрица *смежности*	матриця *суміжності*	adjacency matrix	117
списки *смежности*	списки *суміжності*	adjacency lists	118
собственное подмножество	власна підмножина	proper subset	48
собственный подграф	власний підграф	proper subgraph	116

совершенная дизъюнктивная нормальная форма	досконала диз'юнктивна нормальна форма	canonical disjunctive normal form	16
совершенная конъюнктивная нормальная форма	досконала кон'юнктивна нормальна форма	canonical conjunctive normal form	16
сокращённая ДНФ	скорочена ДНФ	prime implicants	25
сокращённое вычисление логических выражений	скорочене обчислення логічних виразів	short-circuit evaluation of boolean expressions	11
сортировка вставками	сортування вставками	insertion *sort*	85
сортировка выбором	сортування вибором	selection *sort*	84
состояние (автомата)	стан (автомата)	state (of automaton)	170
сочетания из n по k	сполучення з n по k	k-combinations of n-element set	96
сочетания с повторениями из n по k	сполучення з повтореннями з n по k	k-combinations with repetitions of n-element set	97
точка *сочленéния*	точка *зчленува́ння*	articulation point	125
списки смежности	списки суміжності	adjacency *lists*	118
список рёбер	список ребер	edge *list*	119
сравни́мые и не сравнимые элементы	пори́внювані та не порівнювані елементи	comparable and non-comparable elements	67
сравни́мы по модулю p	пори́внювані за модулем p	equivalent mod p	64
статус (состояние) вершины	статус (стан) вершини	vertex *status (state)*	129
степень вершины	степінь вершини	vertex *degree*	113
степень відношення	степінь відношення	power of relation	60
степень языка	степінь мови	power of language	163
стрелка Пирса	стрілка Пірса	Peirce *arrow*	38
отношение *строгого порядка*	відношення *строгого порядку*	strict order	66
правило *суммы*	правило *суми*	rule of *sum*	93
суперпозиция отношений	суперпозиція відношень	composition of relations	59
квантор *существования*	квантор *існування*	existential quantifier	32
счётное множество	зліче́нна множина	countable set	75
сюръекция (сюръективная функция)	сюр'єкція (сюр'єктивна функція)	surjection (surjective function)	73
таблица истинности (построение)	таблиця істинності (побудова)	truth *table* (constructing)	12
таблица истинности (стандартная)	таблиця істинності (стандартна)	truth *table* (standard)	8
таблица поглощений	таблиця поглинань	prime implicant *chart*	25
тавтология	тавтологія	tautology	26
текущее множество состояний (НКА)	поточна множина станів (НСА)	current set of states (NFA)	176
текущее состояние автомата	поточний стан автомата	current state of automaton	170
теорема Дирака	теорема Дірака	Dirac's theorem	128
теорема Кантора	теорема Кантора	Cantor's *theorem*	74
теорема Клини	теорема Кліні	Kleene theorem	174
теорема Поста о функциональной полноте	теорема Поста про функціональну повноту	Post's functional completeness theorem	38
теорема Эйлера (об эйлеровом цикле)	теорема Ейлера (про ейлерів цикл)	Euler's *theorem* (about Eulerian cycle)	127
тернарный	тернарний	ternary	9
тождественное отношение	тотожнє відношення	equality relation	60
топологи́ческая сортировка орграфа	топологі́чне сортування орграфа	topological sort (topsort) of digraph	137
топологи́ческая сортировка отношения порядка	топологі́чне сортування відношення порядку	topological sorting of order	69
тор	тор	torus	20
точка сочлене́ния	точка зчленува́ння	articulation *point*	125
транзитивное отношение	транзитивне відношення	transitive relation	61
треугольник Паскаля	трикутник Паскаля	Pascal's *triangle*	100

трёхзначная логика	тризначна логіка	three-value logic	39
унарный	унарний	unary	9
универсум	універсум	universe	47
алгоритм *Уоршалла*	алгоритм *Воршалла*	Warshall's algorithm	143
фактор-множество	фактор-множина	quotient set	66
фиктивная переменная	фіктивна змінна	non-essential variable	13
алгоритм *Флойда*	алгоритм *Флойда*	Floyd's algorithm	143
формальный язык	формальна мова	formal language	162
функционально полная система	функціонально повна система	functionally complete set	36
функция	функція	function	72
функция переходов	функція переходів	state-transition function	171
задание множества *характеристическим предикатом*	подання множини *характеристичним предикатом*	set-builder notation with predicate	46
диаграмма *Хассе*	діаграма *Гассе*	Hasse diagram	67
хвост очереди	хвіст черги	tail of queue	129
центр (графа)	центр (графа)	center (of graph)	123
центральная вершина	центральна вершина	central vertex	123
цепь	ланцюг	trail	120
цикл	цикл	tour	120
чёрная дыра	чорна діра	black hole	173
числа Каталана	числа Каталана	Catalan *numbers*	104
шаг индукции	крок індукції	inductive *step*	81
поиск в *ширину*	пошук у *ширину* (вшир)	breadth-first search	129
штрих Шеффера	штрих Шеффера	Sheffer *stroke*	38
ЭД	ЕД	ED	16
круги *Эйлера*	круги *Ейлера*	Euler's diagrams	48
теорема *Эйлера* (об эйлеровом цикле)	теорема *Ейлера* (про ейлерів цикл)	Euler's theorem (about Eulerian cycle)	127
эйлеров путь (эйлерова цепь)	ейлерів шлях (ланцюг)	Eulerian path (trail)	127
ЭК	ЕК	EC	16
класс *эквивалентности*	клас *еквівалентності*	equivalence class	64
отношение *эквивалентности*	відношення *еквівалентності*	equivalence relation	63
эквивалентны по модулю p	еквівалентні за модулем p	equivalent mod p	64
k-*эквивалентные* состояния	k-*еквівалентні* стани	k-*equivalent* states	184
эквиваленция	еквіваленція	iff	8
эксцентриситет (вершины)	ексцентриситет (вершини)	eccentricity (of vertex)	122
элемент	елемент	element	46
элементарная дизъюнкция	елементарна диз'юнкція	elementary disjunction	16
элементарная конъюнкция	елементарна кон'юнкція	elementary conjunction	16
энка	енка	n-*tuple*	55
ε-переход	ε-перехід	ε-move	176
явный перечень	явний перелік	explicit enumeration	46
регулярный *язык*	регулярна *мова*	regular *language*	163
формальный *язык*	формальна *мова*	formal *language*	162
ε-НКА	ε-НСА	ε-NFA	176
ε-переход	ε-перехід	ε-move	176
$\overline{C}_n^k$	$\overline{C}_n^k$	$\left(\!\binom{n}{k}\!\right)$	97
A_n^k	A_n^k	$[n]_k$	95
C_n^k	C_n^k	$\binom{n}{k}$	96
k-свя́зный (вершинно) граф	k-зв'я́зний (вершинно) граф	k-vertex-connected graph	126
k-свя́зный (реберно) граф	k-зв'я́зний (реберно) граф	k-edge-connected graph	126
k-эквивалентные состояния	k-еквівалентні стани	k-equivalent states	184
n-арный	n-арний	n-ary	9
n-ка	n-ка	n-tuple	55

Англо-українсько-російський словник та покажчик

accept state	допусковий стан	допускающее состояние	171
to *accept* string	допускати слово	допускать слово	171
addition principle	правило суми	правило суммы	93
adjacency lists	списки *суміжності*	списки *смежности*	118
adjacency matrix	матриця *суміжності*	матрица *смежности*	117
Dijkstra's *algorithm*	алгоритм Дейкстри	алгоритм Дейкстры	141
Euclid's *algorithm*	алгоритм Евкліда	алгоритм Евклида	87
Floyd's *algorithm*	алгоритм Флойда	алгоритм Флойда	143
Warshall's *algorithm*	алгоритм Воршалла	алгоритм Уоршалла	143
input *alphabet*	вхідний *алфавіт*	входной *алфавит*	170
and	і	и	8
antisymmetric relation	антисиметричне відношення	антисимметричное отношение	61
arc (of digraph)	дуга (орграфа)	дуга (орграфа)	112
Peirce *arrow*	стрілка Пірса	стрелка Пирса	38
articulation point	точка *зчленува́ння*	точка *сочлене́ния*	125
associativity (“$\vee$”, “$\wedge$”)	асоціативність (“$\vee$”, “$\wedge$”)	ассоциативность (“$\vee$”, “$\wedge$”)	14
Aufenkamp–Hohn algorithm	алгоритм Ауфенкампа–Хона	алгоритм Ауфенкампа–Хона	184
backtrace (restoring path)	зворотній хід (відновлення шляху)	обратный ход (восстановление пути)	131
back-tracking	відтинання при переборі	отсечения при переборе	115
base case (basis)	база індукції	базис (база) индукции	81
BFS	пошук у ширину (вшир)	поиск в ширину	129
biconditional	еквіваленція	эквиваленция	8
biconnected component	двозв'я́зна компонента	двусвя́зная компонента	126
biconnectedness of graph	двозв'я́зність графа	двусвя́зность графа	126
bijection (injective function)	бієкція (бієктивна функція)	биекция (биективная функция)	73
binary	бінарний	бинарный	9
binary relation	бінарне відношення	бинарное отношение	58
binary relation between A and B	бінарне відношення в $A \times B$	бинарное отношение в $A \times B$	58
binary relation matrix	матриця бінарного відношення	матрица бинарного отношения	58
binary relation on A	бінарне відношення на A	бинарное отношение на A	58
binomial coefficients	біноміальні коефіцієнти	биномиальные коэффициенты	99
binomial expansion	біном Ньютона	бином Ньютона	100
bipartite graph	дводольний (двочастковий) граф	двудольный граф	114
bitwise operators	побітові операції	побитовые операции	10
black hole	чорна діра	чёрная дыра	173
block	блок	блок	126
bound variable	зв'язана змінна	связанная переменная	32
breadth-first search	пошук у *ширину* (вшир)	поиск в *ширину*	129
bridge	міст	мост	125
canonical conjunctive normal form	досконала кон'юнктивна нормальна форма	совершенная конъюнктивная нормальная форма	16
canonical disjunctive normal form	досконала диз'юнктивна нормальна форма	совершенная дизъюнктивная нормальная форма	16
Cantor's theorem	теорема *Кантора*	теорема *Кантора*	74
set *cardinality*	міра множини	мера множества	98
Cartesian power	декартів степінь	декартова степень	55
Cartesian product	декартів добуток	декартово произведение	55
Catalan numbers	числа *Каталана*	числа *Каталана*	104
CCNF	ДКНФ	СКНФ	16
CDNF	ДДНФ	СДНФ	16
center (of graph)	центр (графа)	центр (графа)	123
central vertex	центральна вершина	центральная вершина	123
prime implicant *chart*	таблиця поглинань	таблица поглощений	25

equivalence *class*	клас еквівалентності	класс эквивалентности	64
closed walk	замкнений маршрут	замкнутый маршрут	120
closure of relation	замикання відношення	замыкание отношения	71
CNF	КНФ	КНФ	16
codomain	область значень	область значений	59
method of undetermined *coefficients*	метод невизначених *коефіцієнтів*	метод неопределённых *коэффициентов*	19
k-*combinations* with *repetitions* of n-element set	сполучення з повтореннями з n по k	сочетания с повторениями из n по k	97
k-*combinations* of n-element set	сполучення з n по k	сочетания из n по k	96
combining EC	склеювання ЕК	склеивание ЭК	23
commutativity ("$\vee$", "$\wedge$")	комутативність ("$\vee$", "$\wedge$")	коммутативность ("$\vee$", "$\wedge$")	14
comparable and non-comparable elements	порі́внювані та не порівнювані елементи	сравни́мые и не сравнимые элементы	67
complement	доповнення	дополнение	49
complete bipartite graph	повний дводольний граф	полный двудольный граф	114
complete graph	повний граф	полный граф	114
functionally *complete* set	функціонально *повна* система	функционально *полная* система	36
connected *component*	компонента зв'я́зності	компонента связности	124
component, strong (strongly connected)	компонента, сильна (сильної зв'я́зності)	компонента, сильная (сильной свя́зности)	124
component, weak (weakly connected)	компонента, слабка (слабкої зв'я́зності)	компонента, слабая (слабой свя́зности)	124
composition of relations	композиція відношень	композиция отношений	59
concatenation	конкатенація	конкатенация	163
condense of graph	конденсація графа	конденсация графа	125
conjunction	кон'юнкція	конъюнкция	8
conjunctive normal form	кон'юнктивна нормальна форма	конъюнктивная нормальная форма	16
connected component	компонента *зв'я́зності*	компонента *связности*	124
connected graph	зв'я́зний граф	связный граф	123
k-edge-*connected* graph	ре́берно k-*зв'я́зний* граф	рёберно k-*свя́зный* граф	126
k-vertex-*connected* graph	вершинно k-*зв'я́зний* граф	вершинно k-*свя́зный* граф	126
continuum	континуа́льна множина	континуа́льное множество	75
tautology test by *contradiction*	метод *редукції*	метод *редукции*	27
contrapositive law	закон *контрапозиції* імплікації	закон *контрапозиции* импликации	14
countable set	злічéнна множина	счётное множество	75
counterexample	контрприклад	контрпример	31
current set of states (NFA)	поточна множина станів (НСА)	текущее множество состояний (НКА)	176
current state of automaton	поточний стан автомата	текущее состояние автомата	170
to *decline* string	відхиляти слово	отклонять слово	171
vertex *degree*	степінь вершини	степень вершины	113
DeMorgan's laws	закони *де Моргана*	законы *де Моргана*	14
dense graph	щільний граф	плотный граф	118
depth-first search	пошук у *глибину* (вглиб)	поиск в *глубину*	134
DFS	пошук у глибину (вглиб)	поиск в глибину	134
diablo state	диявольський стан	дьявольское состояние	173
diagonal argument	діагональний метод	диагональный метод	74
Hasse *diagram*	діаграма Гассе	диаграмма Хассе	67
Euler's *diagrams*	круги Ейлера	круги Ейлера	48
Venn's *diagrams*	діаграми Венна	диаграммы Венна	48
diameter (of graph)	діаметр (графа)	диаметр (графа)	122
difference	різниця	разность	49
digraph (directed graph)	орграф (орієнтований граф)	орграф (ориентированный граф)	112

digraph, semiconnected	орграф, односторонньо зв'язний	орграф, односторонне связный	124
digraph, strongly connected	орграф, сильно зв'язний	орграф, сильно связный	124
digraph, weakly connected	орграф, слабко зв'язний	орграф, слабо связный	124
Dijkstra's algorithm	алгоритм *Дейкстри*	алгоритм *Дейкстры*	141
Dirac's theorem	теорема Дірака	теорема Дирака	128
directed graph	орієнтований граф	ориентированный граф	112
discrete	дискретний	дискретный	6
disjunction	диз'юнкція	дизъюнкция	8
disjunctive normal form	диз'юнктивна нормальна форма	дизъюнктивная нормальная форма	16
distance	ві́дстань	расстоя́ние	121
distance matrix	матриця *відстаней*	матрица *расстояний*	122
distributivity ("∧", "∨")	дистрибутивність ("∧", "∨")	дистрибутивность ("∧", "∨")	14
DNF	ДНФ	ДНФ	16
domain	область визначення	область определения	59
double negation law	закон *подвійного заперечення*	закон *двойного отрицания*	14
EC	ЕК	ЭК	16
eccentricity (of vertex)	ексцентриси́тет (вершини)	эксцентриси́тет (вершины)	122
ED	ЕД	ЭД	16
edge	ребро	ребро	112
edge biconnectedness	ре́берна двозв'я́зність	рёберная двусвя́зность	126
edge list	список *ребер*	список *рёбер*	119
element	елемент	элемент	46
elementary conjunction	елементарна кон'юнкція	элементарная конъюнкция	16
elementary disjunction	елементарна диз'юнкція	элементарная дизъюнкция	16
empty set	порожня множина	пустое множество	47
empty string	порожнє слово	пустое слово	162
endpoints of undirected edge	кінці неорієнтованого ребра	концы неориентированного ребра	112
ε-move	ε-перехід	ε-переход	176
equality relation	тотожнє відношення	тождественное отношение	60
equipotent sets	рівнопотужні множи́ни	равномощные множества	73
equivalence class	клас *еквівалентності*	класс *эквивалентности*	64
equivalence relation	відношення *еквівалентності*	отношение *эквивалентности*	63
equivalent mod p	пори́внювані за модулем p	сравни́мы по модулю p	64
k-equivalent states	*k*-*еквівалентні* стани	*k*-*эквивалентные* состояния	184
Euclid's algorithm	алгоритм *Евкліда*	алгоритм *Евклида*	87
Euler's diagrams	круги *Ейлера*	круги *Эйлера*	48
Eulerian path (trail)	ейлерів шлях (ланцюг)	эйлеров путь (эйлерова цепь)	127
Euler's theorem (about Eulerian cycle)	теорема *Ейлера* (про ейлерів цикл)	теорема *Эйлера* (об эйлеровом цикле)	127
exclusive or	виключне або	исключающее или	8
existential quantifier	квантор *існування*	квантор *существования*	32
explicit enumeration	явний перелік	явный перечень	46
exponentiation by squaring (iterative)	швидке піднесення до степеню (ітеративне)	быстрое возведение в степень (итерационное)	88
exponentiation by squaring (recursive)	швидке піднесення до степеню (рекурсивне)	быстрое возведение в степень (рекурсивное)	90
regular *expression*	регулярний *вираз*	регулярное *выражение*	164
linear *extention* of order	розширення відношення порядку до лінійного	расширение отношения порядка до линейного	68
false	хиба	ложь	8
final state	заключний стан	заключительное состояние	171
finite-state transducer	автомат-перетворювач	автомат-преобразователь	180
Floyd's algorithm	алгоритм *Флойда*	алгоритм *Флойда*	143
forest	ліс	лес	146

formal language	формальна мова	формальный язык 162
free variable	вільна змінна	свободная переменная 32
full subgraph	правильний підграф	правильный подграф 116
function	функція	функция 72
functionally complete set	функціонально повна система	функционально полная система 36
graph	граф	граф 112
graph search	пошук у гра́фі	поиск в гра́фе 129
graph traversal	обхід графа	обход гра́фа 129
the *greatest* element	найбільший елемент	наибольший элемент 68
Hamiltonian cycle	гамільтонів цикл	гамильтонов цикл 128
handshaking lemma	лема про рукостискання	лемма о рукопожатиях 113
Hasse diagram	діаграма *Гассе*	диаграмма *Хассе* 67
head of arc	кінець дуги	конец дуги 112
head of queue	голова черги	голова очереди 129
hypergraph	гіперграф	гиперграф 117
idempotence (“$\vee$”, “$\wedge$”)	ідемпотентність (“$\vee$”, “$\wedge$”)	идемпотентность (“$\vee$”, “$\wedge$”) 14
iff	еквіваленція	эквиваленция 8
implication	імплікація	импликация 8
in-degree	напівстепінь захо́ду (входу)	полустепень захо́да 113
incidence matrix	матриця *інциденцій*	матрица *инциденций* 117
inclusion–exclusion principle	принцип *включень та виключень*	принцип *включений и исключений* 98
subgraph *induced* by vertices set	підграф, *породжений* множиною вершин	подграф, *порождённый* множеством вершин 116
mathematical *induction*	метод математичної *індукції*	метод математической *индукции* 81
inductive step	крок індукції	шаг индукции 81
infix notation for relations	інфіксний запис відношень	инфиксная запись отношений 58
initial automaton (FSM)	ініціальний автомат	инициальный автомат 170
initial state	початковий стан	начальное состояние 170
injection (injective function)	ін’єкція (ін’єктивна функція)	инъекция (инъективная функция) 73
input alphabet	вхідний алфавіт	входной алфавит 170
input string	вхідне слово	входное слово 170
insertion sort	сортування *вставками*	сортировка *вставками* 85
intersection	перетин	пересечение 48
graph *invariant*	інваріант графа	инвариант графа 115
loop *invariant*	інваріант циклу	инвариант цикла 83
inverse function	обернена функція	обратная функция 73
inverse relation	обернене відношення	обратное отношение 59
irreflexive relation	іррефлексивне відношення	иррефлексивное отношение 61
isolated vertex	ізольована вершина	изолированная вершина 114
isomorphic graphs	ізоморфні графи	изоморфные графы 115
Karnaugh map (K-map)	карта Карно	карта Карно 20
Kleene star	ітерація	итерация 163
Kleene theorem	теорема Кліні	теорема Клини 174
formal *language*	формальна *мова*	формальный *язык* 162
regular *language*	регулярна *мова*	регулярный *язык* 163
law of contraposition	закон контрапозиції імплікації	закон контрапозиции импликации 14
law of double negation	закон подвійного заперечення	закон двойного отрицания 14
law of the excluded middle	закон виключення третього	закон исключения третьего 14
DeMorgan’s *laws*	закони де Моргана	законы де Моргана 14
leaf (pendant vertex)	висяча (кінцева́) вершина	висячая (концева́я) вершина 114
the *least* element	найменший елемент	наименьший элемент 68
walk *length*	довжина маршруту	длина́ маршрута 121
string’s *length*	довжина сло́ва	длина́ сло́ва 162

linear order	відношення *лінійного порядку*	отношение *линейного порядка*	67
linear relation	лінійне відношення	линейное отношение	61
edge *list*	список ребер	список рёбер	119
adjacency *lists*	списки суміжності	списки смежности	118
loop invariant	інваріант циклу	инвариант цикла	83
Karnaugh *map*	карта Карно	карта Карно	20
mathematical induction	метод математичної індукції	метод математической индукции	81
adjacency *matrix*	матриця суміжності	матрица смежности	117
distance *matrix*	матриця відстаней	матрица расстояний	122
incidence *matrix*	матриця інциденцій	матрица инциденций	117
binary relation *matrix*	матриця бінарного відношення	матрица бинарного отношения	58
reachability *matrix*	матриця досяжності	матрица достижимости	121
a *maximal* element	максимальний елемент	максимальный элемент	67
Mealy machine	автомат Мі́лі	автомат Ми́ли	182
merge	злиття	слияние	53
reduction *method*	метод редукції	метод редукции	27
semantic tree *method*	метод семантичних дерев	метод семантических деревьев	27
method of undetermined coefficients	метод невизначених коефіцієнтів	метод неопределённых коэффициентов	19
Quine's *method* (tautology test)	метод Квайна (перевірки тавтологічності)	метод Квайна (проверки тавтологичности)	26
Quine–McCluskey *method*	метод Квайна–Мак-Класкі	метод Квайна–Мак-Класки	23
a *minimal* element	мінімальний елемент	минимальный элемент	68
minimization of automata	мінімізація автоматів	минимизация автоматов	184
minimization of boolean functions	мінімізація булевих функцій	минимизация булевых функций	20
Moore machine	автомат Му́ра	автомат Му́ра	181
De*Morgan*'s laws	закони де *Моргана*	законы де *Моргана*	14
MST	МОД	МОД	147
multigraph	мультиграф	мультиграф	113
multiplication principle	правило добутку	правило произведения	93
multiset	мультимножина	мультимножество	47
MWST	ОДМВ	ОДМВ	147
negation	заперечення	отрицание	8
network (weighted graph)	мережа (зважений граф)	сеть (взвешенный граф)	113
ε-NFA	ε-НСА	ε-НКА	176
NFA	НСА	НКА	176
non-comparable	не порі́внювані	не сравни́мы	67
non-strict order	відношення *нестрогого порядку*	отношение *нестрогого порядка*	66
nondeterministic automaton	недетермінований автомат	недетерминированный автомат	175
non-essential variable	фіктивна змінна	фиктивная переменная	13
not	не	не	8
Catalan *numbers*	числа Каталана	числа Каталана	104
function *onto*	функція *на*	функция *на*	73
or	або	или	8
order relation	відношення *порядку*	отношение *порядка*	66
out-degree	напівстепінь виходу	полустепень исхо́да	113
parenthis expresession (counting quantity)	дужкові вирази (підрахунок кількості)	скобочные выражения (подсчёт количеств)	103
partial automaton	неповний вигляд автомата	неполный вид автомата	173
partition	розбиття	разбиение	65
partition	розбиття	разбиение	124
Pascal's triangle	трикутник *Паскаля*	треугольник *Паскаля*	100
PCNF	ДКНФ	СКНФ	16
PDNF	ДДНФ	СДНФ	16
Peirce arrow	стрілка *Пірса*	стрелка *Пирса*	38

pendant vertex (leaf)	кінцева́ (висяча) вершина	концева́я (висячая) вершина	114
perfect conjunctive normal form	досконала кон'юнктивна нормальна форма	совершенная конъюнктивная нормальная форма	16
perfect disjunctive normal form	досконала диз'юнктивна нормальна форма	совершенная дизъюнктивная нормальная форма	16
permutations	перестановки	перестановки	95
permutations of multiset	перестановки з повтореннями	перестановки с повторениями	96
k-*permutations* of n-element sequence	розміщення з n по k	размещения из n по k	95
pigeonhole principle	принцип *Дирихле*	принцип *Дирихле*	73
articulation *point*	точка зчленува́ння	точка сочлене́ния	125
Zhegalkin *polynomial*	поліно́м Жегалкіна	полино́м Жегалкина	18
Post's classes (T_0, T_1, S, L, M)	класи Поста (T_0, T_1, S, L, M)	классы Поста (T_0, T_1, S, L, M)	37
Post's functional completeness theorem	теорема Поста про функціональну повноту	теорема Поста о функциональной полноте	38
power of language	степінь мови	степень языка	163
power of relation	степінь відношення	степень відношення	60
power set of set	булеа́н множини	булеа́н множества	49
powerset construction	детермінізація	детерминизация	179
precedence of logical connectives	пріоритети логічних операцій	приоритеты логических операций	9
predicate	предикат	предикат	29
prehistory	передісторія	предыстория	170
matrix *presentation* of binary relation	подання бінарного відношення матрицею	представление бинарного отношения матрицей	58
presentation of predicates	подання предикатів	представление предикатов	30
prime implicant chart	імплікантна таблиця	импликантная таблица	25
prime implicants	скорочена ДНФ	сокращённая ДНФ	25
addition *principle*	правило суми	правило суммы	93
inclusion–exclusion *principle*	принцип включень та виключень	принцип включений и исключений	98
multiplication *principle*	правило добутку	правило произведения	93
pigeonhole *principle*	принцип Дирихле	принцип Дирихле	73
automata *product*	добуток автоматів	произведение автоматов	173
rule of *product*	правило *добутку*	правило *произведения*	93
proper subgraph	власний підграф	собственный подграф	116
proper subset	власна підмножина	собственное подмножество	48
pumping lemma	лема про накачку	лемма о накачке	175
existential *quantifier*	квантор існування	квантор существования	32
universal *quantifier*	квантор загальності	квантор (все)общности	31
queue	черга	очередь	129
Quine's method (tautology test)	метод *Квайна* (перевірки тавтологічності)	метод *Квайна* (проверки тавтологичности)	26
Quine–McCluskey method	метод *Квайна–Мак-Класкі*	метод *Квайна–Мак-Класки*	23
quotient set	фактор-множина	фактор-множество	66
radius (of graph)	радіус (графа)	радиус (графа)	123
reachability	дося́жність	достижи́мость	121
reachability matrix	матриця *досяжності*	матрица *достижимости*	121
to *recognize* language	розпізнавати мову	распознавать язык	171
recognizer	автомат-розпізнавач	автомат-распознаватель	170
recognizer	автомат-*розпізнавач*	автомат-*распознаватель*	170
recurrence relations	рекурентні співвідношень	рекуррентные соотношения	101
reduction method	метод *редукції*	метод *редукции*	27
reflexive relation	рефлексивне відношення	рефлексивное отношение	61
regular expression	регулярний вираз	регулярное выражение	164
regular graph	регулярний (однорідний) граф	регулярный (однородный) граф	114

regular language	регулярна мова	регулярный язык 163
relation	відношення	отношение 57
binary *relation* between A and B	бінарне *відношення* в $A \times B$	бинарное *отношение* в $A \times B$ 58
binary *relation* on A	бінарне *відношення* на A	бинарное *отношение* на A 58
equivalence *relation*	відношення еквівалентності	отношение эквивалентности 63
order *relation*	відношення порядку	отношение порядка 66
relaxation of shortest-path estimate	релаксація оцінки відстані	релаксация оценки расстояния 141
repeating squaring	індійський алгоритм піднесення до степеню	индийский алгоритм возведения в степень 88
representative of equivalence class	представник класу еквівавлентності	представитель класса эквивалентности 66
restoring path (backtrace)	відновлення шляху (зворотній хід)	восстановление пути (обратный ход) 131
route	маршрут	маршрут 119
«rule of division»	«правило частки»	«правило частного» 96
rule of product	правило добутку	правило произведения 93
rule of sum	правило суми	правило суммы 93
Russian peasant method	індійський алгоритм піднесення до степеню	индийский алгоритм возведения в степень 88
breadth-first *search*	пошук у ширину (вшир)	поиск в ширину 129
depth-first *search*	пошук у глибину (вглиб)	поиск в глубину 134
graph *search*	пошук у гра́фі	поиск в гра́фе 129
selection sort	сортування *вибором*	сортировка *выбором* 84
self-loop	петля	петля 112
semantic tree method	метод *семантичних дерев*	метод *семантических деревьев* 27
semiconnected digraph	односторонньо зв'язний орграф	односторонне связный орграф 124
set	множина́	мно́жество 46
set-builder notation with predicate	подання множини характеристичним предикатом	задание множества характеристическим предикатом 46
short-circuit evaluation of boolean expressions	скорочене обчислення логічних виразів	сокращённое вычисление логических выражений 11
simple cycle	простий цикл	простой цикл 120
simple graph	простий граф	простой граф 113
simple path	простий ланцюг	простая цепь 120
insertion *sort*	сортування вставками	сортировка вставками 85
selection *sort*	сортування вибором	сортировка выбором 84
spanning subgraph	остовний (каркасний) підграф	о́стовный подграф 116
spanning tree	остовне (каркасне) дерево	о́стовное дерево 146
minimum weight *spanning tree*	остовне (каркасне) дерево мінімальної ваги	о́стовное дерево минимального веса 147
sparse graph	розріджений граф	разреженный граф 118
Kleene *star*	ітерація	итерация 163
state (of automaton)	стан (автомата)	состояние (автомата) 170
state-transition function	функція переходів	функция переходов 171
vertex *status (state)*	статус (стан) вершини	статус (состояние) вершины 129
inductive *step*	крок індукції	шаг индукции 81
strict order	відношення *строгого порядку*	отношение *строгого порядка* 66
string	слово	слово 162
empty *string*	порожнє *слово*	пустое *слово* 162
input *string*	слово вхідне	слово входное 170
strings	розміщення з повтореннями з n по k	размещения с повторениями из n по k 97
Sheffer *stroke*	штрих Шеффера	штрих Шеффера 38
strong component	сильна компонента	сильна компонента 124
strongly connected digraph	сильно зв'язний орграф	сильно связный орграф 124

subgraph	підграф	подграф	116
subgraph induced by vertices set	підграф, породжений множиною вершин	подграф, порождённый множеством вершин	116
subset	підмножина	подмножество	47
proper *subset*	власна *підмножина*	собственное *подмножество*	48
rule of *sum*	правило *суми*	правило *суммы*	93
surjection (surjective function)	сюр'єкція (сюр'єктивна функція)	сюръекция (сюръективная функция)	73
symbol	символ	символ	162
symmetric difference	симетрична різниця	симметрическая разность	49
symmetric relation	симетричне відношення	симметричное отношение	61
tail of arc	початок дуги	начало дуги	112
tail of queue	хвіст черги	хвост очереди	129
tautology	тавтологія	тавтология	26
tautology test by contradiction	метод редукції	метод редукции	27
ternary	тернарний	тернарный	9
Euler's *theorem* (about Eulerian cycle)	теорема Ейлера (про ейлерів цикл)	теорема Эйлера (об эйлеровом цикле)	127
Cantor's *theorem*	теорема Кантора	теорема Кантора	74
three-value logic	тризначна логіка	трёхзначная логика	39
topological sort (topsort) of digraph	топологі́чне сортування орграфа	топологи́ческая сортировка орграфа	137
topological sorting of order	топологі́чне сортування відношення порядку	топологи́ческая сортировка отношения порядка	69
torus	тор	тор	20
total automaton	повний вигляд автомата	полный вид автомата	173
total order	відношення *лінійного порядку*	отношение *линейного порядка*	67
total relation	повне відношення	полное отношение	61
tour	цикл	цикл	120
trail	ланцюг	цепь	120
finite-state *transducer*	автомат-*перетворювач*	автомат-*преобразователь*	180
state-*transition* function	функція *переходів*	функция *переходов*	171
transitive relation	транзитивне відношення	транзитивное отношение	61
breadth-first *traversal*	обхід у ширину (вшир)	обход в ширину	129
depth-first *traversal*	обхід у глибину (вглиб)	обход в глубину	134
graph *traversal*	обхід графа	обход гра́фа	129
tree (undirected)	дерево (неорієнтоване)	дерево (неориентированное)	146
Pascal's *triangle*	трикутник Паскаля	треугольник Паскаля	100
true	істина	истина	8
truth table (constructing)	таблиця *істинності* (побудова)	таблица *истинности* (построение)	12
truth table (standard)	таблиця *істинності* (стандартна)	таблица *истинности* (стандартная)	8
n-tuple	енка	энка	55
unary	унарний	унарный	9
method of *undetermined coefficients*	метод *невизначених коефіцієнтів*	метод *неопределённых коэффициентов*	19
undirected graph	неорієнтований граф	неориентированный граф	112
union	об'єднання	объединение	48
union	об'єднання	объединение	162
universal quantifier	квантор *загальності*	квантор *всеобщности*	31
universe	універсум	универсум	47
unweighted graph	незважений граф	невзвешенный граф	113
Venn's diagrams	діаграми *Венна*	диаграммы *Венна*	48
vertex	вершина	вершина	112
vertex biconnectedness	вершинна двозв'я́зність	вершинная двусвя́зность	126

walk	шлях	путь	119
Warshall's algorithm	алгоритм *Воршалла*	алгоритм *Уоршалла*	143
weak component	слабка компонента	слабая компонента	124
weak order	відношення *нестрогого порядку*	отношение *нестрогого порядка*	66
weakly connected digraph	слабко зв'язний орграф	слабо связный орграф	124
edge's *weight*	довжина ребра́	длина́ ребра́	113
weighted graph	зважений граф	взвешенный граф	113
xor	ксор	ксор	8
Zhegalkin polynomial	полінóм *Жегалкіна*	полинóм *Жегалкина*	18
k-combinations of n-element set	сполучення з n по k	сочетания из n по k	96
k-combinations with repetitions of n-element set	сполучення з повтореннями з n по k	сочетания с повторениями из n по k	97
k-edge-connected graph	ре́берно k-зв'я́зний граф	рёберно k-свя́зный граф	126
k-equivalent states	k-еквівалентні стани	k-эквивалентные состояния	184
k-permutations of n-element sequence	розміщення з n по k	размещения из n по k	95
k-vertex-connected graph	вершинно k-зв'я́зний граф	вершинно k-свя́зный граф	126
n-ary	n-арний	n-арный	9
n-tuple	n-ка	n-ка	55
$[n]_k$	A_n^k	A_n^k	95
ε-move	ε-перехід	ε-переход	176
ε-NFA	ε-НСА	ε-НКА	176
$\left(\!\binom{n}{k}\!\right)$	$\overline{C}_n^k$	$\overline{C}_n^k$	97
$\binom{n}{k}$	C_n^k	C_n^k	96

Покажчик значків та інших позначок

Навчальне видання

Порубльов І. М.

Дискретна математика

Навчальний посібник
для студентів 1-го курсу бакалаврату
галузі знань «Інформаційні технології»
та споріднених

Здано до набору 14.06.2018.
Підписано до друку 15.08.2018.
Формат 60x84/8. Папір офсет. Гарнітура Times.
Ум. др. арк 25,57. Наклад 100 прим.

Це видання надруковано на папері із деревини відповідної нормам екологічного лісовикористання

Видавець ФОП Гордієнко Є.І.
Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготовників і розповсюджувачів видавничої продукції
Серія ДК № 4518 від 04.04.2013 р.
Україна, 18000, м. Черкаси
тел./факс: (0472) 56-56-12, (067) 444-28-94
e-mail: book.druk@gmail.com