Міністерство освіти і науки України
Львівський фізико-математичний ліцей
при Львівському національному університеті
імені Івана Франка

Міжнародний конкурс з інформатики та комп'ютерного мислення «Бебрас-2018»

Матеріали школи програмування (Львів-2018)

Кам'янець-Подільський
«Аксіома»
2019

УДК 519.683
М58

Упорядник:
Ростислав Шпакович

Схвалено для використання у загальноосвітніх навчальних закладах
Науково-методичною радою з питань освіти
Міністерства освіти і науки України
(лист ІМЗО від 15.07.2019 № 22.1/12-Г-665)

М58 Міжнародний конкурс з інформатики та комп'ютерного мислення «Бебрас–2018». Матеріали школи програмування : навчально-методичний посібник / С. С. Жуковський, І. М. Порубльов, І. В. Скляр, Р. С. Шпакович – Кам'янець-Подільський : Аксіома, 2019. – 104 с.

ISBN 978-966-496-495-8

У посібнику подано теоретичний матеріал та задачі, які опрацьовувались слухачами Школи програмування переможців конкурсу Бебрас, яка проходила у жовтні 2018 року в м. Львові. Ці матеріали будуть корисними вчителям інформатики та учням при вивченні основ програмування та підготовці до олімпіад.

Подано інформацію про проведення конкурсу у світі та Україні, вказівки до розв'язування задач конкурсу 2018 року.

Видання здійснено на благодійницьких засадах
і розповсюджується безкоштовно

УДК 519.683

ISBN 978-966-496-495-8

© Львівський фізико-математичний ліцей, 2019
© «Аксіома», видання, 2019

Передмова

Все більш масовим у світі стає Міжнародний конкурс з інформатики та комп'ютерного мислення «Бебрас» (у перекладі з литовської – «Бобер»).

У минулому році до нього приєднались ще тринадцять країн. Всього у конкурсі «Бебрас-2018» взяли участь понад 2 мільйони 780 тисяч учнів з 57-ми країн світу. В Україні конкурс пройшов одинадцятий раз, детальніше – у статті «Конкурс «Бебрас-2018» у світі та Україні».

Конкурс є одним з перших етапів участі учнів у цікавих змаганнях з інформатики та спортивного програмування. Метою більшості завдань конкурсу є ілюстрація ефективного застосування класичних та сучасних алгоритмів до розв'язування прикладних та олімпіадних задач. Приклади таких задач, які використовувались у минулорічному конкурсі, – у статті «Вибрані задачі конкурсу».

Вже п'ять років у м. Львові проводиться Осіння школа програмування для переможців конкурсу «Бебрас». Остання школа відбулась 2–8 жовтня 2018 року. Цього разу у ній взяли участь 30 учнів з 10 областей України (Волинської, Дніпропетровської, Донецької, Запорізької, Львівської, Одеської, Харківської, Херсонської, Черкаської, Чернівецької).

Упродовж 7 днів школи учасники навчалися та удосконалювали свої знання з програмування. Заняття проводили досвідчені викладачі, автори відомих книг з програмування та задач Всеукраїнських олімпіад та турнірів:

Сергій Жуковський, доцент Житомирського державного університету;

Ілля Порубльов, старший викладач Черкаського національного університету;

Ірина Скляр, викладач Київського природничо-наукового ліцею.

Заняття проводились паралельно у двох групах:

1 група – переможці обласних та Всеукраїнських олімпіад.

Теми занять:

1 день – Ілля Порубльов. Динамічне програмування.

2 день – Ілля Порубльов. «Коли динамічне програмування недоречне».

3 день – Сергій Жуковський. Структури даних.

2 група – учні, які роблять перші кроки в олімпіадному програмуванні.

Теми занять:

1 день – Сергій Жуковський. Бітова арифметика.
2 день – Ірина Скляр. Двовимірні масиви.
3 день – Ірина Скляр. Динамічне програмування.

У останній день була проведена олімпіада.

Її переможцями стали:

11 клас

1. Денис Матяш (обласна школа-інтернат «Обдарованість», м. Харків)
2. Богдан Євтушенко («Обдарованість», м. Харків)
3. Михайло Таланцев («Обдарованість», м. Харків)

10 клас

1. Данило Рясний (гімназія № 28, м. Запоріжжя)
2. Кирило Караваєв (Бердянська СШ № 16, Запорізька обл.)
3. Владислав Сосунович (Скадовська академічна гімназія, Херсонська обл.)

9 клас

1. Кирило Стрельбицький (Маріупольський технічний ліцей, Донецька обл..)
2. Вадим Пастушок (Луцька гімназія № 14.)
3. Олег Брозинський (Чернівецький ліцей № 1)

Вісім учнів цьогорічної школи стали учасниками 4-го туру Всеукраїнської олімпіади з інформатики 2019 року.

У статтях авторів цієї книжки подано теоретичний матеріал та задачі, які опрацьовувались слухачами школи. Ці матеріали будуть корисними вчителям інформатики та учням при вивченні основ програмування та підготовці до олімпіад.

Ростислав Шпакович. Вибрані задачі конкурсу «Бебрас-2018» та вказівки до їх розв'язування

Демоверсія всіх завдань конкурсу та вказівки до їх розв'язування розміщені на сайті конкурсу у архіві завдань: http://bober.net.ua/page.php?name=archive&

У даній статті розглянуті лише деякі типи завдань для вибраних вікових груп. Спробуйте розв'язати аналогічні завдання інших вікових груп самостійно.

1. Задача «Олімпіада» (2–11 класи)

Бобренята Богдан, Божен, Борис, Боян взяли участь в олімпіаді з інформатики. У кожному з трьох наступних повідомлень одне твердження правильне, а інше – неправильне:

Боян – перше місце, Борис – друге місце;

Боян – друге місце, Богдан – третє місце;

Божен – друге місце, Богдан – четверте місце.

Вкажіть, яке місце посіло кожне бобреня.

Розв'язування:

Найпростіший спосіб розв'язування – перевірити, яке з двох тверджень першої стрічки правильне.

1) Нехай правильне твердження: ***Борис посів друге місце.***

Тоді у другій стрічці неправильне перше твердження, і повинне бути правильним твердження: ***Богдан посів третє місце.*** Але після цього ***обидва твердження третьої стрічки стають неправильними.***

Тому залишається припустити, що правильне друге твердження першої стрічки:

2) Нехай правильне твердження: ***Боян посів перше місце.***

Тоді у другій стрічці правильне твердження: ***Богдан посів третє місце.***

У третій стрічці правильне твердження: ***Божен посів друге місце.***

Борису залишилось четверте місце.

2. Задача «Горішки» (6–7 класи)

Є 14 горішків. Бобренята Аліса та Боб по черзі беруть з правого краю або 1, або 4, або 9 горішків. Перемагає той, хто забирає останній горішок. Розпочинає Аліса. Допоможіть їй перемогти.

Розв'язування:

Задача є класичною грою на виграшні та програшні позиції. Кожен гравець старається, щоб після його ходу суперник опинився у програшній позиції.

Очевидно, що позиція виграшна, якщо залишилось або один, або чотири, або дев'ять горішків.

Відповідно, коли залишилось два або п'ять горішків, позиція програшна – гравець може зробити хід у лише виграшні для суперника позиції з одним або чотирма горішками.

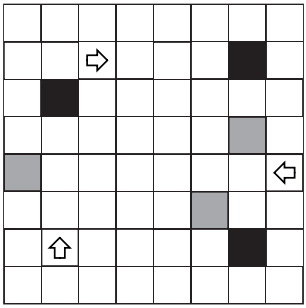
Внесемо ці дані у табличку. У першому рядку таблиці вказана кількість горішків, що залишились, у другому рядку літера В означає, що дана позиція виграшна, літера П означає, що позиція програшна:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
В	П		В	П				В					

Попробуйте заповнити табличку до кінця. Після цього неважко знайти серію виграшних ходів.

3. Задача «Робот» (6-9 класи)

Три роботи неперервно рухаються вперед. Якщо перед роботом є перешкода, він повертає праворуч і продовжує свій рух. Початкові перешкоди позначені чорним кольором.



Поставте якнайменше додаткових перешкод, щоб роботи не змогли вийти за межі поля.

Відповідь: три перешкоди. Вони позначені сірим кольором

4. Задача «Повітряні кулі» (8–9 класи)

Десять бобрів-мандрівників хочуть оглянути вершину кіпрської гори Олімпус на п'ятьох повітряних кулях. У кожну кулю можуть сісти лише по два бобри. Вантажопідйомність кулі A 11 кг, кулі B – 14 кг, кулі C – 15 кг, кулі D – 18 кг, кулі E – 23 кілограми. Імена та ваги бобрів: Джеррі – 14, Аїша – 12, Рембо – 11, Артур – 10, Джессі – 9, Арчі – 8, Ксена – 7, Том – 5, Боб – 3, Мікі – 2 кг.

Розподіліть всіх бобрів по повітряних кулях.

Один з алгоритмів розв'язування:

Сумарна вага бобрів та загальна вантажопідйомність всіх куль однакові (по 81 кг). Отже кожна куля повинна бути завантажена повністю.

1) Складемо список всіх можливих способів повного завантаження кожної кулі.

Куля A (11 кг): 2+9 або 3+8 (Мікі і Джессі, або Боб і Арчі).

Куля B (14 кг): 2+12, 3+11, 9+5.

Куля C (15 кг): 12+3, 10+5, 8+7.

Куля D (18 кг): 11+7, 10+8.

Куля E (23 кг): 14+9, 12+11.

2) З цього списка бачимо, що Джеррі (14 кг) можна посадити лише в кулю E разом з Джессі (9 кг).

Після цього Джессі можна викреслити, як кандидата на посадку в інші кулі:

Куля A (11 кг): ~~2+9~~ або 3+8

Куля B (14 кг): 2+12, 3+11, ~~9+5~~.

Куля C (15 кг): 12+3, 10+5, 8+7.

Куля D (18 кг): 11+7, 10+8.

Куля E (23 кг): **14+9**, ~~12+11~~.

3) Тепер бачимо, що у кулю A повинні сісти Боб (3 кг) та Арчі (8 кг). Після цього знову викреслюємо їх, як кандидатів на посадку в інші кулі.

Попробуйте завершити розв'язання цієї задачі самостійно.

5. Задача «Замок» (8–11 класи)

На валізі бобра Захара є трьохцифровий кодовий замок. Щоб не забути код, Захар записав на своєму мобільнику такі чотири підказки.

875 – у цьому коді лише одна цифра правильна і вона знаходиться на своєму місці.

523 – у цьому коді дві цифри правильні, аде жодна не знаходиться на своєму місці.

163 – у цьому коді жодної правильної цифри.

164 – у цьому коді лише одна цифра правильна.

Розв'язування:

Послідовність логічних висновків:

1) З третього та четвертого тверджень робимо висновок, що цифра 4 входить у код, а цифри 1, 3 та 6 – не входять.

2) З другого твердження робимо висновок, що цифри 2 і 5 входять у код, але зараз не на своїх місцях.

3) З першого твердження робимо висновок, що цифра 5 повинна бути на третій позиції. Після цього цифри 2 залишається лише перша позиція.

Отже відповідь – 245.

6. Задача «Монети» (10–11 класи)

Бобренята знайшли 4 мішки з монетами. У деяких мішках лише справжні срібні монети вагою по 10 грамів, у деяких лише легші фальшиві монети вагою по 9 грамів, у деяких — важчі фальшиві монети вагою по 11 грамів. Бобренята можуть зробити лише одне зважування.

Вони поклали на вагу 1 монету з першого мішка, 3 монети з другого, 9 монет з третього та 27 монет з четвертого мішка. Сумарна вага цих монет виявилась рівною 415 грамам.

Визначіть, які монети у кожному мішку.

Розв'язування:

Бобренята поклали на вагу 40 монет. Сорок справжніх монет повинні важити 400 грамів. Тому важчих фальшивих монет на п'ятнадцять більше, ніж легших фальшивих монет.

Отже, ми повинні, використовуючи лише числа 1, 3, 9 та 27, отримати алгебраїчну суму, що дорівнює 15. Це можна зробити єдиним способом:

$$27 - 9 - 3 = 15.$$

Отже, у першому мішку – справжні монети, у другому та третьому – легші фальшиві, а у четвертому – важчі фальшиві монети.

Отримати число 15, як алгебраїчну суму чисел 27, 9, 3 і 1, неважко. Для великих чисел і великої кількості доданків це зробити набагато важче.

Розглянемо інший підхід до цієї задачі:

Переведемо число 15 у трійкову систему числення:

$$\mathbf{15_{10}= 120_3}, \text{ або } \mathbf{1*9+2*3+0*1=15}$$

Після цього трійкове число 120 переведемо у зважену трійкову систему числення (коли коефіцієнти при степенях трійки можуть дорівнювати лише 0, або 1, або –1):

$\mathbf{120_3}$ = 1*9+2*3+0*1= 1*9+ (1*9–1*3)+0*1= 1*27–1*9–1*3+0*1= **[1,–1,–1, 0]** – (у зваженій трійковій системі числення).

Тобто значення 0 у відповідному розряді означає, що монети справжні, –1 означає, що монети легші, +1 означає, що монети важчі.

Попробуйте розв'язати цю ж задачу з такими змінами:

Бобренята знайшли 7 мішків з монетами. Вони поклали на вагу 1 монету з першого мішка, 3 монети з другого, 9 монет з третього, 27 монет з четвертого, 81 монету з п'ятого, 243 монети з шостого та 729 монет з сьомого мішка. Після зважування виявилось, що вага цих монет менша за відповідну вагу такої ж кількості справжніх монет на 600 грамів. Визначіть, які монети у кожному мішку.

7. Задача «Впорядкування» (10–11 класи)

Розташуйте картки зліва направо у цій же стрічці по зростанню номерів за найменшу кількість ходів. За один хід можна довільну картку перемістити у сусідню вільну (по горизонталі або вертикалі) клітинку:

4	1	2	7	5	6	3

Розв'язування:

У верхній стрічці числа 1, 2, 5, 6 утворюють найдовшу зростаючу підпослідовність.

Тому у другу стрічку потрібно опустити лише картки з цифрами 4, 3 і 7.

Причому картку з цифрою 7 потрібно опускати у другу стрічку лише після того, як картка з цифрою 3 стане на своє місце у верхній стрічці.

Відповідь: 18 ходів.

Ірина Скляр. Сортування лінійних масивів

Поняття сортування

Загальне значення слова «сортування» — це розподіл елементів на групи за деякою ознакою, наприклад, розподіл цукерок за вартістю, товарів за якістю, тощо. Проте у програмуванні всі дані представлені у числовому вигляді, а тому сортування фактично є упорядкуванням чисел за зростанням або за спаданням. Наприклад, розташування слів у словнику в алфавітному порядку, учнів за середнім балом, тощо.

Як бачимо, об'єктами впорядкування можуть бути найрізноманітніші елементи. Головне, щоб їх можна було ***порівнювати*** за деякою ознакою («Об'єкт A має стояти після об'єкту B або, навпаки, перед ним»).

Ознаку, за якою сортуються елементи, називають ***ключем сортування***. Це може бути число (вартість мобільного телефона), послідовність символів (слово у словнику) або складніша величина.

Сортування з'явилося у практичній діяльності задовго до появи комп'ютерів і є необхідним у багатьох практичних задачах. Так, очевидно, що у відсортованому наборі можна значно прискорити подальший пошук та обробку даних. Наприклад, якщо ми шукаємо слово, коли нам невідомо, де воно розташоване в словнику, ми розкриваємо словник приблизно посередині. Якщо слово починається з букви, яка за алфавітом знаходиться далі, ніж перші букви слів на сторінці, шукаємо тільки в другій половині словника. Далі розкриваємо словник на середині його другої половини. Щоразу, заглядаючи в словник, ми поділяємо «простір пошуку» навпіл, зменшуючи його приблизно вдвічі. Описаний пошук називається ***бінарним***. Він дозволяє нам швидко знайти слово завдяки саме тому, що словник упорядковано за алфавітом. Якби цього не було, довелося б перебирати всі слова підряд.

Отже, упорядкування даних має дуже велике значення: саме тому було створено десятки алгоритмів сортування та їх численні реалізації.

Традиційно розрізняють ***зовнішні*** та ***внутрішні*** сортування. Перші з них використовуються для сортування дуже великих обсягів даних, які не можна помістити цілком в оперативну пам'ять. Для цього використовують зовнішню пам'ять, таку як стример, він-

честер, флеш-пам'ять тощо. А другі – упорядковують масиви даних в оперативній (внутрішній) пам'яті комп'ютера.

У шкільному курсі ми будемо розглядати тільки методи внутрішнього сортування. Вони також класифікуються за кількома ознаками, найголовнішими з яких є такі:

- швидкість виконання алгоритму;
- об'єм додаткової пам'яті (додаткового масиву).

За швидкістю виконання алгоритми сортування поділяють на ***базові*** та ***швидкі***. Базові працюють порівняно повільно, проте їх легко записати та налагодити. Швидкі, переважно, є значно складнішими для розуміння, але сортують набагато швидше, особливо великі масиви даних. Щоб уточнити, що означає «швидкий алгоритм», нам знадобиться поняття складності алгоритму, представлене нижче.

Поняття складності алгоритму

Складність алгоритму зазвичай оцінюють або за часом виконання, або за використаною пам'яттю. В обох випадках складність залежить від розмірів вхідних даних: адже очевидно, що масив, який містить 100 елементів, буде оброблений швидше, ніж масив з кількістю елементів 10^6. При цьому точний час роботи алгоритму, як правило, мало кого цікавить, адже він залежить від тактової частоти процесора, мови програмування, методу представлення даних та інших параметрів. Як правило, важливою є асимптотична складність, тобто складність, яка залежить від розміру вхідних даних.

Порядок росту описує, як складність алгоритму зростає зі збільшенням розміру вхідних даних. Найчастіше він представлений у вигляді О-нотації (від нім. «*Ordnung*» – порядок): $O(f(x))$, де $f(x)$ – формула, що виражає складність алгоритму. Формально $O(f(x))$ означає, що час роботи алгоритму (або обсяг використовуваної пам'яті) росте в залежності від розміру вхідних даних не швидше, ніж деяка константа, що помножена на $f(x)$.

В формулі $f(x)$ може бути присутня змінна n, що представляє розмір вхідних даних. Найчастіше зустрічаються такі варіанти складності.

Константний – $O(1)$. Такий порядок означає, що обчислювальна складність алгоритму не залежить від розміру вхідних даних.

Лінійний – $O(n)$. Лінійний порядок - означає, що складність алгоритму лінійно росте зі збільшенням розміру вхідних даних. Так,

якщо лінійний алгоритм обробляє один елемент даних за одну мілісекунду, тисячу елементів він опрацює за одну секунду. Таку складність, наприклад, має алгоритм пошуку максимального елементу у невідсортованому масиві, адже потрібно перевірити усі елементи масиву, щоб зрозуміти, який з них найбільший.

Логарифмічний – $O(\log\ n)$. Такий порядок росту означає, що час виконання алгоритму росте логарифмічно зі збільшенням розміру вхідного масиву. При аналізі алгоритму за замовчанням використовується логарифм за основою 2. Найпростіший приклад – бінарний пошук. Якщо масив відсортовано, пошук конкретного числа можна виконувати методом ділення навпіл. Для цього спочатку перевіряється середній елемент. Якщо він більше шуканого, відкидається друга половина масиву, оскільки там його точно немає. Якщо ж менше, то навпаки – відкидається перша половина. Так продовжується ділення навпіл, поки не буде знайдено шуканий елемент, або чергова частина масиву буде містити всього один елемент і він не є шуканим. Очевидно, що в результаті буде перевірено $\log\ n$ елементів.

Лінійно-логарифмічний – $O(n\ \log\ n)$. У цю категорію попадають деякі алгоритми «розділяй та володарюй», наприклад, сортування злиттям або швидке сортування, які буде розглянуто пізніше.

Квадратичний – $O(n^2)$. Час роботи алгоритму з порядком росту $O(n^2)$ залежить від квадрату розміру вхідного масиву. Таку складність має, наприклад, алгоритм сортування вставками. Він являє собою два вкладених цикли: один для проходу по всьому масиву, а другий – для знаходження місця черговому елементу у вже відсортованій частині. Таким чином, кількість операцій буде залежати від розміру масиву як $n*n$.

Не дивлячись на те, що такої ситуації іноді не уникнути, квадратична складність – це привід переглянути алгоритм або структуру даних. Дійсно, якщо масив із тисячі елементів потребує 10^6 операцій, то масив з 10^6 елементів буде потребувати 10^{12} операцій. Якщо одна операція вимагала би одну мілісекунду для свого виконання, квадратичний алгоритм обробляв би 10^6 елементів 32 роки! Навіть пришвидшений у сто разів, такий алгоритм виконувався б 84 дні. На жаль, усі базові алгоритми сортування мають квадратичну складність.

При вимірюванні складності алгоритмів та структур даних ми зазвичай говоримо про дві речі: кількість операцій, що потрібні

для завершення роботи (обчислювальна складність), та об'єм ресурсів, зокрема, пам'яті, який необхідний алгоритму (просторова складність). Алгоритм, що виконується в десять разів швидше, але використовує у десять разів більше пам'яті, цілком підходить для серверної машини з великим обсягом пам'яті. Але на вбудованих системах, де кількість пам'яті обмежена, такий алгоритм використовувати не можна.

У нашому випадку ми будемо говорити переважно про обчислювальну складність, але при розгляданні алгоритмів сортування торкнемося також питань ресурсів. Операції, кількість яких ми будемо вимірювати, включають в себе:

- порівняння («більше», «менше», «дорівнює»);
- присвоєння;
- виділення пам'яті.

Те, які операції ми враховуємо, зазвичай ясно з контексту. Так, при описі алгоритму пошуку елементу в структурі даних ми маємо на увазі операції порівняння. Пошук – це переважно процес читання, тому нема сенсу робити присвоєння або виділення пам'яті. Коли ж ми говоримо про сортування, ми можемо враховувати як порівняння, так і виділення пам'яті або присвоєння. У таких випадках ми будемо явно вказувати, які операції ми розглядаємо.

А тепер повернемося до розгляду алгоритмів різних сортувань.

Обмінне сортування

Першим розглянемо алгоритм обмінного сортування, яке традиційно вважається найпростішим. За назвою алгоритму зрозуміло, що його основною операцією буде обмін двох елементів місцями. Однак очевидно, що обміни не можуть бути хаотичними: потрібно мати якусь ознаку. Оскільки сортуванню будуть підлягати числа, домовимося упорядковувати їх за зростанням (або за неспаданням при наявності однакових елементів). Тоді очевидно, що два будь-яких елементи масиву повинні мати таке розташування, щоб лівий елемент пари був меншим (не більшим) за правий. Теоретично, переглянувши усі можливі пари елементів і переставивши їх згідно наведеної вище ознаки, ми отримаємо відсортований набір даних. Але як організувати перебір усіх можливих пар, не повторюючись та не пропускаючи деякі з них? Для цього пропонується розглядати усі пари послідовно, починаючи, наприклад, з початку масиву і рухаючись поступово до його кінця.

Розглянемо для прикладу послідовність значень 13, 5, –4, 2, 1. Після порівняння першої пари елементів, вони поміняються місцями згідно нашої домовленості. Потім будуть порівняні елементи другої пари і так далі. В результаті поступово у масиві відбудуться такі зміни (на кожному кроці підкреслено елементи, що будуть порівнюватися):

13 5 –4 2 1
5 13 –4 2 1
5 –4 **13 2** 1
5 –4 **2 13 1**
5 –4 **2 1 13**

Цей метод ще називається ***бульбашковим сортуванням***.

Як видно з прикладу, після завершення першого проходу по масиву елементи ще будуть не повністю відсортовані, а тому знову прийдеться почати з початку. Черговий прохід по масиву «виштовхне» другий за величиною елемент на передостаннє місце. Однак, і після цього масив повністю впорядкований не буде, а тому починати з початку буде необхідно не один раз.

Спробуємо логічними міркуваннями підрахувати, скільки разів у найгіршому випадку прийдеться починати прохід масиву з початку. Очевидно, що найгіршим буде випадок, коли початковий масив упорядковано навпаки, тобто за спаданням (незростанням). Тоді після кожного проходу рівно один елемент буде займати своє місце, тобто починати спочатку прийдеться **N–1** раз.

Наведемо функцію бульбашкового сортування **bubbleSort** (bubble – бульбашка).

```
void bubbleSort(int m[], int N)
{
  for(int i=0; i<N-1; i++)
    for(int j=0; j<N-1; j++)
      if(m[j]>m[j+1]) swap(m[j],m[j+1]);
}
```

Очевидно, що запропонована реалізація має основний недолік: рух виконується завжди до останньої пари, хоча остання пара вже обов'язково впорядкована. Для усунення цього недоліку потрібно зменшувати кількість пар, що переглядаються. Вкладений цикл можна оформити таким чином:

```
for (int i=1; i<N; i++)
  for (int j=0; j<N-i; j++)
    //Вже впорядковані елементи правої частини масиву
    //не порівнюються}
    if (m[j]>m[j+1]) swap(m[j],m[j+1]);
```

Найбільша кількість елементарних обмінів останнього алгоритму у найгіршому випадку дорівнює загальній кількості порівнянь, яких: $(N-1)+(N-2)+\ldots+1=\frac{N(N-1)}{2}$. Звідси складність сортування N-елементного масиву описаним способом має оцінку $O(N^2)$.

Тепер спробуємо зменшити кількість проходів по масиву, модифікувавши зовнішній цикл. За словесним описом алгоритму повторювати проходи по масиву потрібно, *доки масив не буде відсортовано*. А тому зовнішній цикл повинен бути циклом «поки» (**while**). Виникає питання, за якою умовою він буде працювати, адже поняття «відсортовано – не відсортовано» не має відображення в операторах мови програмування. Для реалізації цієї перевірки пропонуємо застосувати метод, що має аналогію з методом доказу «від супротивного» у математиці.

У нашому випадку в якості припущення беремо наступне: «нехай масив вже відсортовано». Щоб перевірити припущення, потрібно пройти по масиву і порівняти усі пари сусідніх елементів. Якщо всі пари стоять правильно, масив відсортовано. Якщо ж хоча б раз виявиться, що пара елементів не задовольняє ключу сортування, то масив ще не відсортовано і потрібно почати перегляд спочатку.

Для фіксації ситуації «відсортовано – не відсортовано» візьмемо змінну логічного типу, причому домовимося, що значення цієї змінної **true** відповідає відсортованому масиву і значення **false** – не відсортованому. Початкове значення змінної буде **false**, оскільки масив містить довільні елементи і принаймні один раз потрібно запустити цикл для перевірки усіх його пар сусідніх елементів. Але після входу в цикл **while** змінимо значення цієї змінної на true – припускаємо, що масив відсортовано. Якщо в результаті порівняння хоча б один раз відбулася переміна місцями елементів масиву, припущення було хибним, тобто міняємо значення змінної на **false**. Якщо в результаті чергового проходу всі елементи виявилися на своїх місцях, змінна залишиться у стані **true** і цикл **while** припинить свою роботу. Отже фрагмент програмної реалізації має вигляд:

```
bool flag=false;
while (!flag)
{
  flag=true; // Припускаємо, що масив відсортовано
  for (int j=0; j<N-1; j++)
    if (m[j]>m[j+1])
    {
      flag=false; // Припущення було хибним
      swap(m[j], m[j+1]);
    }
}
```

Дана реалізація дозволяє зменшити кількість проходів по масиву до необхідного мінімуму. Так, якщо масив відразу відсортовано, перший прохід буде єдиним, оскільки в результаті його виконання змінна **flag** не змінить свого стану (всі пари елементів розміщені згідно ключа сортування). Однак, при несприятливому стані масиву (відсортовано навпаки), кількість проходів зменшити не вдасться, адже елементи, що знаходяться в кінці масиву, потрібно перемістити на початок.

Можна дещо зменшити кількість порівнянь у вкладеному циклі для довільного масиву, якщо кінцеве значення змінної циклу коригувати, як і в попередньому випадку, з урахуванням того, що при кожному проході принаймні один елемент займає свою позицію. Тоді заголовок вкладеного циклу **for** буде мати наступний вигляд:

```
for (int j=0; j<N-i; j++) ...
```

Як при цьому повинна змінюватися змінна **i** подумайте самостійно.

При сортуванні на деякому кроці виконання алгоритму може трапитися така ситуація, коли ліва частина масиву ще не відсортована, а права вже впорядкована, причому всі значення впорядкованої частини перевищують значення невпорядкованої частини. Наприклад (впорядкована частина підкреслена):

3 12 –4 –5 2 <u>100 101 200 240 241</u> ...

У цьому випадку навіть зменшення кінцевого значення циклу на i суттєво не покращить ситуацію і кількість зайвих перевірок буде досить великою. З метою урахування вказаної ситуації пропонуємо ввести поняття «правої границі перестановки». Під цією величиною будемо розуміти індекс найправішого елементу масиву, який на попередньому проході брав участь у перестановці.

На початку алгоритму індекс найправішого елементу перестановки буде **N–1**, оскільки стан випадково заданого масиву не відомий і можлива ситуація, що в його правому кінці елементи ще не впорядковані. На кожному проході будемо запам'ятовувати поточну границю перестановки, яка після завершення роботи вкладеного циклу буде дорівнювати індексу лівого елементу останньої переставленої пари. Очевидно, що всі елементи, які розташовані правіше цього індексу, вже впорядковано і на наступному проході їх порівнювати нема сенсу. Фрагмент функції, що реалізовує описаний алгоритм, має наступний вигляд:

```
bool flag=false;
int i;
int right=N-1; // Початкове значення останньої праворуч перестановки
while (!flag)
{
  flag=true; // Припускаємо, що масив відсортовано
  for (int j=0; j<right; j++)
    if (m[j]>m[j+1])
    {
      flag=false; // Припущення було хибним
      swap(m[j],m[j+1]);
      i=j; // Запам'ятовування індексу поточної перестановки
    }
  right=i; // Запам'ятовування індексу останньої
           // праворуч перестановки
}
```

Запропонована реалізація дозволяє максимально швидко переміщувати максимальні елементи з лівого краю масиву у правий без виконання зайвих перевірок та проходів по масиву. Але якщо з правого боку масиву будуть знаходитися мінімальні значення, цей метод ситуацію не покращить, адже мінімум за повний прохід по масиву переміститься тільки на одну позицію. Наприклад, для такого набору даних:

12 23 45 234 235 700 1000 1001 2034 1

після завершення проходу по масиву мінімальний елемент з правого боку займе замість останнього передостаннє місце:

12 23 45 234 235 700 1000 1001 1 2034

Щоб прискорити просування маленьких чисел справа наліво до початку масиву, оптимальною буде організація зворотного проходу

по масиву – від останнього елемента до першого. Цей прохід дозволить мінімальному елементу швидко стати на своє місце. Крім цього, можна використати той самий прийом, що й раніше, для зменшення кількості проходів внутрішнього циклу: запам'ятовувати номер останнього елементу (найлівішого), що переставлявся, для використання його індексу в якості границі невідсортованої частини.

Однак, просто замінити проходи по масиву зліва направо на зворотні (справа наліво) не має сенсу, оскільки у загальному випадку початковий стан елементів невідомий. А тому пропонується чергувати проходи зліва направо зі зворотніми проходами – справа наліво. У такому випадку масив буде відсортований не більше, ніж за N/2 таких «подвійних» проходів. Використовуючи прапорець, можна ще покращити алгоритм, завершуючи процес впорядкування, коли перестановки вже не виконуються.

Такий різновид вдосконалення базового алгоритму обмінного сортування називається методом «шейкера». Він має особливий виграш, коли багато елементів вхідного масиву даних стоять далеко від правильних своїх позицій. Програмна реалізація алгоритму шейкерного сортування має наступний вигляд:

```
void ShakerSort (int m[], int N)
{
  int Left=0, Right=N-1, i;
  bool flag=false;
  while (!flag)
  {
    flag=true;
    for (int j=Left; j<Right; j++)
      if (m[j]>m[j+1])
      {
        swap(m[j],m[j+1]);
        i=j;
        flag=false;
      }
    Right=i;
    if (!flag)
    {
      for (int j=Right; j>=Left; j--)
        if (m[j]>m[j+1])
        {
          swap(m[j],m[j+1]);
          i=j;
          flag=false;
        }
```

```
        Left=i;
      }
    }
  }
```

Текст програми, як бачимо, став значно більшим, але швидкість сортування помітно зростає, якщо на вхід подано достатньо великий обсяг даних.

При розгляді запропонованих алгоритмів потрібно чітко розуміти, що наявність двох циклів для сортування одновимірного масиву обгрунтована необхідністю організувати кілька (зовнішній цикл) проходів по масиву (внутрішній цикл). Кожен з них не залежить від іншого і тому запропоновані оптимізації можна використовувати як разом, так і окремо. Наприклад, оптимізація з використанням прапорця може не передбачати зменшення кількості проходів внутрішнього циклу, тобто внутрішній цикл може виконуватись N–1 разів, але при черговому проході без перестановок елементів прапорець буде набувати значення **false** і завершувати виконання алгоритму.

Сортування вибором

Даний метод сортування, виходячи з назви, основною операцією має операцію вибору. Очевидно, що при сортуванні вибирати потрібно максимальні або мінімальні елементи, адже чітко відомо, де вони повинні міститися у відсортованому масиві. Розглянемо спочатку алгоритм, у якому будуть шукатися максимуми. Оскільки сортувати ми будемо за зростанням (неспаданням), найбільші елементи будуть міститься з правого боку масиву.

Почнемо з пошуку максимального елементу заданого масиву. У відсортованому наборі цей елемент повинен знаходитися на останньому місці, однак там знаходиться інший елемент. А тому потрібно поміняти місцями ці два елементи – максимальний та останній. Далі аналогічні дії (пошук максимуму та виставлення його у правий бік) повторюємо без урахування вже виставленого елементу. Поступово кількість елементів, що знаходяться на своїх місцях, збільшується, а кількість невідсортованих елементів – зменшується. Процес відбувається N–1 разів і завершується, коли у невідсортованій частині залишиться тільки один елемент.

Промоделюємо алгоритм на конкретному наборі даних: **10 –6 13 24 0 –67 100 75**. Будемо знаходити на кожному кроці максимум (підкреслений елемент) і міняти його місцями з останнім елементом

розглядуваної частини масиву:

10 –6 13 24 0 –67 100 –75
10 –6 13 24 0 –67 –75 100
10 –6 13 –75 0 –67 24 100
10 –6 –67 –75 0 13 24 100
0 –6 –67 –75 10 13 24 100
–75 –6 –67 0 10 13 24 100
–75 –67 –6 0 10 13 24 100
–75 –67 –6 0 10 13 24 100

Функція, яка реалізує описаний алгоритм, має вигляд:

```
void selectSort(int m[], int N)
{
  int iMax; //індекс максимального значення
  for (int k=N-1; k>0; k--)
  {
    iMax=0;
    for (int i=1; i<=k; i++)
      if (m[i]>m[iMax]) iMax=i;
    if (iMax != k) swap(m[k],m[iMax]);
  }
}
```

Перший цикл реалізації організовує N–1 раз пошук максимального елементу масиву, причому він організований таким чином, щоб його параметр дорівнював індексу останнього розглядуваного елементу масиву. Другий цикл призначений безпосередньо для пошуку максимуму у розглядуваному підмасиві.

Очевидно, що складність цього алгоритму, як і алгоритму обмінного сортування, має оцінку $O(N^2)$ – два вкладених цикли. У порівнянні з бульбашковим сортуванням він потребує меншої кількості обмінів – не більше N–1. Однак, на відміну від обмінного, складність цього сортування завжди однакова, оскільки пошук максимуму – це повноперебірна задача, і здійснювати цей пошук потрібно N–1 разів.

Наведений алгоритм має один непомітний недолік, який може бути важливим у деяких реальних задачах. Однакові значення після сортування змінюють свій взаємний порядок, тому цей алгоритм називають ***нестійким***. На відміну від нього, бульбашкове сортування упорядковує однакові значення, зберігаючи їхнє взаємне роз-

ташування, тому є ***стійким***. У деяких задачах це буває дуже суттєвим, а тому потрібно це враховувати.

Сортування вставками

Наступні алгоритми сортування – вставками – відрізняються від наведених вище алгоритмів тим, що шукаються не «елементи для місць», а «місця для елементів». Спочатку розглянемо так званий алгоритм ***прямих вставок***.

У масиві виділяємо дві частини: відсортовану та невідсортовану. Оскільки нам нічого не відомо про початковий стан елементів, у відсортовану частину спочатку потрапляє тільки один елемент, адже один елемент сам із собою відсортований. Далі щоразу беремо чергове значення з невідсортованої частини та вставляємо його до відсортованої так, щоб вона не втратила впорядкованості. Для цього зсуваємо всі елементи, що стоять лівіше розглядуваного та більші за нього вправо. У певний момент виявляється, що всі елементи зсунуті і на звільнене місце можна поставити розглядуваний елемент. Отже, елемент «вставляється» у відсортовану частину масиву (звідки й назва методу). Функція, що реалізує описаний алгоритм, має такий вигляд:

```
void insertSort(int m[], int N)
{
  int etalon,i;
  for (int j=1; j<N; j++)
  {
    etalon=m[j]; // Елемент, для якого буде шукатися місце
    i=j-1;
    while (m[i]>etalon && i>=0)
    {
      // Зсув усіх елементів, що більші розглядуваного
      m[i+1]=m[i]; i--;
    }
    // Вставка елементу на звільнене місце
    m[i+1]=etalon;
  }
}
```

Можна реалізувати алгоритм вставки дещо по-іншому, але для цього потрібні допоміжні витрати пам'яті. Щоб став зрозумілий підхід, наведемо аналогію. Припустимо є такий набір даних: **23 45 12 7 -4 100**. Очевидно, що число **23** у відсортованому масиві буде на

четвертому місці, оскільки менше за нього три елементи, число **45** – на п'ятому з тієї ж причини, число **12** – на третьому, і так далі.

Оскільки визначені місця для елементів у масиві зайняті, доведеться ставити їх не у початковий масив, а у інший – додатковий. Таким чином, цей алгоритм вимагає додаткових витрат пам'яті для своєї реалізації. Крім того, певна проблема виникає з однаковими елементами – адже за наведеним алгоритмом вони будуть потрапляти в одну ту саму позицію. Можна знайти кілька методів боротьби з цим недоліком. Та ми пропонуємо такий: для визначення місця розташування розглядуваного елементу у відсортованому масиві потрібно порахувати не тільки кількість елементів, що менші за даний, але й ті, що рівні даному та розташовані раніше нього. Тоді для кожного елементу початкового масиву буде визначатися унікальне місце в результуючому. Програмна реалізація описаного методу має такий вигляд:

```
void counter(int m[], int M[], int N)
{
  int cnt;
  for (int i=0; i<N; i++)
  {
    cnt=0;
    for (int j=0; j<N; j++)
      if (m[j]<m[i] || m[i]==m[j] && j<i)
        cnt++;
    M[cnt]=m[i];
  }
}
```

Зверніть увагу, що ця функція на відміну від попередніх має три аргументи, два з яких масиви:

- **m** – не відсортований масив;
- **M** – відсортований масив.

Після виконання функції початковий масив залишиться без змін, а новий масив буде містити відсортовані елементи вихідного масиву. Складність запропонованих методів прямої вставки також має оцінку $O(N^2)$, оскільки кожен з них містить два вкладених цикли.

У 1959 році Д. Шелл запропонував удосконалення алгоритму прямих вставок. Його ідея — порівнювати елементи, що знаходяться на певній відстані один від одного, і покроково зменшувати цю відстань.

Розглянемо приклад. Нехай масив має такий вигляд.

44 55 12 42 94 18 06 67

Спочатку окремо згрупуємо й упорядкуємо за допомогою прямих вставок елементи, які знаходяться на відстані 4 (четверне впорядкування). У цьому прикладі вісім елементів, тому групи містять по два елементи.

44 18 06 42 94 55 12 67

Тепер згрупуємо елементи на відстані 2 (подвійне впорядкування). Маємо дві групи по чотири елементи.

06 18 12 42 44 55 94 67

Нарешті, виконаємо звичайне (одинарне) впорядкування, порівнюючи сусідні елементи.

06 12 18 42 44 55 67 94

З описаного зрозуміло, що початкова відстань $step = N/2$. Після кожного проходу вона зменшується вдвічі.

```
void ShellSort(int m[], int N)
{
  int etalon, j;
  for (int step=N/2; step>0; step/=2)
    for (int i=step; i<N; i++)
    {
      etalon = m[i];
      for (j=i-step; j>=0 && m[j]>temp; j-=step)
        m[j+step] = m[j];
      m[j+step] = etalon;
    }
}
```

Ідея методу Шелла: змінити масив так, щоб кожна група елементів на відстані *step* була впорядкованою. Це дозволяє за деякого ряду відстаней порівняння *step*, остання з яких дорівнює одиниці, отримати упорядкований масив. Ефективність сортування Шелла забезпечується тим, що елементи «швидше» стають на свої місця, адже вони «перескакують» на більшу за одиницю відстань від початкового стану до кінцевого розташування у відсортованому масиві. Найголовнішим питанням є тільки питання, яку послідовність кроків слід обрати?

Запропоновані Шеллом проміжки, які обчислювалися методом ділення навпіл, у найгіршому випадку дають складність $O(N^2)$. Ба-

гато математиків досліджували представлений алгоритм, вибираючи різні відстані та доводячи їх складність для найгіршого випадку. Один з найкращих результатів був отриманий Седжвіком: за його дослідженнями при певних кроках можна отримати складність вказаного алгоритму $O(N^{\frac{7}{6}})$.

По суті, єдиною перевагою наведених вище алгоритмів є простота й швидкість програмування та наладки. Проте майже всі вони мають оцінку складності $O(N^2)$ і за великих N працюють надто повільно.

Швидкі алгоритми сортування

Розглянемо тепер алгоритми, які мають оцінку складності $O(N \log N)$ і за великих N працюють значно швидше. Проте їх важче зрозуміти й налагодити, деяким з них потрібна додаткова пам'ять великого розміру.

Розглянемо такі методи швидкого сортування:

- індексне сортування;
- пірамідальне сортування;
- швидке сортування;
- порозрядне сортування;
- сортування «злиттям».

Індексне сортування

Цей алгоритм сортування є найшвидшим, але він має один суттєвий недолік: за його допомогою можна сортувати тільки значення обмеженого діапазону. Для пояснення алгоритму розглянемо набір даних, що є натуральними числами, які не перевищують значення 100. Для таких даних можна виконати наступне: спочатку лінійним проходом підрахувати кількість елементів кожного номіналу, а потім, за необхідності, другим лінійним проходом заповнити вхідний масив. Наприклад, для масиву **1 5 2 1 5 3 4 2** за описаним алгоритмом спочатку буде пораховано, що він містить дві одиниці, дві двійки, одну трійку, одну четвірку та дві п'ятірки. Далі записати потрібну кількість кожного з чисел поверх вхідних даних не виявляється складним.

Для реалізації алгоритму, очевидно, знадобляться додаткові витрати пам'яті для збереження кількості елементів кожного номіналу. Це повинен бути масив з кількістю елементів, рівною кількості

різних номіналів вихідного масиву. Функція, що реалізує описаний алгоритм, має такий вигляд:

```
void IndexSort(int m[], int N)
{
  int cnt[101]={0}; // Масив для підрахунку кількості елементів
                    // кожного номіналу
  for (int i=0; i<N; i++)
    cnt[m[i]]++; // Підрахунок кількості
  // Заповнення вихідного масиву даними у відсортованому порядку
  for (int i=100; i>0; i--)
    for (int j=0; j<cnt[i]; j++)
    {
      N--;
      m[N]=i;
    }
}
```

Як відомо, розмір масиву обмежений можливостями системи, а тому набори даних діапазоном [–2*10^9..2*10^9] або більше відсортувати цим методом не вдасться. Крім того, оскільки при виконанні підрахунку кількості кожного номіналу початкового набору, елементи цього масиву є індексами допоміжного масиву, сортувати дійсні числа таким чином також не вдасться.

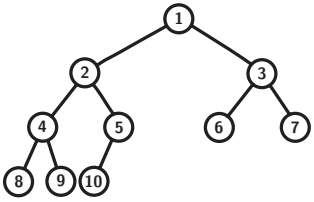
Аналіз алгоритму показує, що він має складність $O(2*N)$, тобто є лінійним з точністю до константи. Другий прохід по масиву потрібен тільки для того, щоб заповнити вихідний масив відсортованими числами. Для розв'язання деяких задач отримання відсортованого масиву необов'язково і тоді складність описаного алгоритму є строго лінійною.

Пірамідальне сортування

Пірамідальне сортування (або ***сортування за допомогою купи***) використовує ***бінарне дерево*** (***піраміду*** або ***купу***). А тому спочатку розглянемо це поняття.

Деревом називається ієрархічна структура, що має зв'язки «батько-син». Кожен елемент дерева називається ***вузлом***, а зв'язки між вузлами – ***дугами***. Вузол, що не залежить від інших (вузол з номером 1), називається ***коренем*** дерева. Вузли, що є залежними від інших, називаються ***синами***. Наприклад, вузли 2 та 3 – сини для вузла 1. Вузли, що мають підлеглих, називаються ***батьками***. Так, вузол 1 – батько для вузлів 2 та 3, вузол 4 – батько для вузлів

8 та 9. Вузли, що не мають синів, називаються ***листами*** (вузли 6, 7, 8, 9 та 10).



Якщо у кожного батька не більше двох синів, дерево називається ***двійковим*** (дивись малюнок).

Побудуємо двійкове дерево таким чином. Перший вузол – корінь. Далі будемо додавати вузли рядками, подвоюючи кількість елементів у кожному рядку: якщо у першому рядку перший елемент, то у другому — елементи 2 і 3, у третьому – елементи 4–7, далі – 8–15 тощо. Останній рядок може залишитися неповним. Наприклад, при n=10 буде утворено дерево, як на рисунку праворуч. Таке дерево називається збалансованим і дійсно нагадує піраміду або купу.

Якщо розташовані у вузлах зображеного дерева числа вважати індексами, то фактично воно є нетрадиційним відображенням лінійного масиву з індексацією від 1 до N, де N – кількість елементів масиву. Очевидно, що в такому масиві елемент з індексом 1 є коренем, $N/2$ (цілочисельне ділення) елементів є батьками. Кожен батько має не більше двох синів (дерево двійкове), причому у кожного i-го батька сини мають індекси 2*i та 2*i+1. Перший з них називається лівим, а другий – правим синами.

А тепер розглянемо безпосередньо сам алгоритм пірамідального сортування. Припустимо, масив двійкового збалансованого дерева має таку властивість: усі його батьки є не меншими за обох своїх синів. Дерево з такою властивістю називається ***правильним***, або ***правильною купою***.

Розглянемо приклад правильного дерева, яке відповідає послідовності значень масиву **30, 12, 30, 12, 5, 29, 2, 11, 10, 3**.

30

12 **30**

12 **5** **29** **2**

11 **10** **3**

Сортування за допомогою купи використовує те, що корінь правильної купи містить її максимальне значення. Поміняємо місцями значення першого елемента масиву (кореня дерева) та останнього. Найбільше значення після цього займе «своє» останнє місце в масиві, і далі про нього можна забути. Серед решти елементів масиву

тільки перший елемент втратив властивість правильної купи – батьки не менше своїх синів. Якщо відновити цю властивість, корінь знову стане максимумом і його потрібно буде обміняти місцями з передостаннім. Так потрібно діяти, поки дерево не скоротиться до одного елемента.

Щоб довільний масив перетворити на правильну купу, а також відновлювати властивість купи після кожної переміни місцями потрібно мати функцію, яка відновлює цю властивість для заданого елемента-батька. Очевидно, що якщо нижче деякого вузла дерева властивість правильної купи вже досягнута, то даний вузол можна «полагодити» шляхом порівняння з його синами. Якщо виявиться, що найбільший з двох синів не перевищує батька, властивість купи для даного вузла справджується. Якщо ж батько виявися меншим за більшого зі своїх синів, їх потрібно поміняти місцями. Далі потрібно перевірити піддерево нижчого рівня, у якого змінився вузол, на збереження властивості правильної купи і за необхідності відновити цю властивість. Функція, яка виконує цей процес, має такий вигляд:

```
void rebuild(int m[], int father, int last_ind)
{
  int maxSon;
  while (father<=last_ind/2) // Перевірка, що елемент ще є батьком
  {
    // Пошук максимального сина у батька
    maxSon=2*father; // Лівий син є завжди
    if (2*father+1<=last_ind // Перевірка наявності правого сина
        && m[maxSon]<m[maxSon+1])
      maxSon++;
    if (m[maxSon]>m[father])
    // Порівняння максимального сина з батьком
    {
      swap(m[maxSon],m[father]);
      father=maxSon; // Оголошення батьком «зіпсованого» сина
    }
    else father=last_ind; // Завершення роботи циклу
  }
}
```

Тепер пояснимо, як за допомогою функції можна зробити перебудову довільного масиву у правильну купу з властивістю «батьки не менше своїх синів». Очевидно, що всі елементи-листя є «пра-

вильними», оскільки їх немає з чим порівнювати. Найпростіше відновити властивість купи у «наймолодшого» (з найбільшим номером) батька, тобто батька з номером $N/2$. Для цього достатньо поміняти його з найбільшим із синів. Піднімаючись поступово до кореня дерева, будемо утворювати правильні піддерева для батьків $N/2$–1, $N/2$–2, ..., 1. Це дозволить стверджувати, що побудований масив представляє правильну купу.

Нарешті, підведемо підсумки. Пірамідальне сортування складається з двох етапів:

- перебудова довільного масиву у правильну купу, в якій батьки не менше своїх синів (при сортуванні за зростанням);
- безпосереднє сортування методом прямого вибору. Для цього максимальний елемент (корінь) міняється місцями з останнім елементом підмасиву, що сортується. Далі вже виставлений елемент викидається з розгляду, а для решти елементів відновлюється властивість правильної купи. Процес триває N–1 разів.

Функція пірамідального сортування має такий вигляд:

```
void heapsort(int m[], int N)
{
  for (int i=N/2; i>0; i--)
    rebuild(m,i,N);
  for (int i=N; i>1; i--)
  {
    swap(m[1],m[i]);
    rebuild(m,1,i-1);
  }
}
```

Оцінимо складність алгоритму. Очевидно, що вона прямо пропорційна загальній кількості викликів функції **rebuild**. На першому етапі при перебудові масиву у правильну купу ця функція викликається N разів, а при сортуванні – ще N–1 разів. Тепер оцінимо складність виконання одного виклику процедури **rebuild**. Помітимо, що в циклі значення змінної *father* не менш ніж подвоюється, а цикл не буде продовжуватися, якщо це значення стане більше `las_index`, яке не більше N. Отже, таких подвоювань не може бути більше ніж $[\log_2 N]$. Кількість дій у тілі циклу є константою, тому загальна складність має оцінку $O(N \log N)$.

Висотою вузла дерева називають кількість ребер у найдовшому шляху з цієї вершини вниз до листків; висоту кореня називають висотою дерева. Дерево, яке утворено як піраміда з n вузлів, має висоту $[\log_2 n]$.

Швидке сортування

Перший алгоритм швидкого сортування було розроблено в 1960 році Ч.А.Р.Хоаром (C.A.R.Hoare). Цей алгоритм є одним із найпопулярніших, оскільки порівняно нескладний у реалізації, добре працює на різноманітних видах вхідних даних і не вимагає величезних обсягів додаткової пам'яті (окрім відносно невеликого стека).

Ідея швидкого сортування така. У деякий спосіб вибирається ***еталонне значення etalon***. Значення елементів масиву m обмінюються так, що масив розбивається на дві ділянки – ліву та праву. Елементи в лівій ділянці мають значення не більше `etalon`, а у правій – не менше. Після цього достатньо окремо відсортувати ці дві ділянки.

Існує простий, але досить ефективний спосіб вибору *etalon* на ділянці масиву **m[Left]**, ..., **m[Right]**: **etalon=m[(Left+Right)/2]**. Для розбиття використовуються два індекси — «лівий курсор» **i** та «правий курсор» **j**. Спочатку **i=Left**, **j=Right**; далі вони рухаються назустріч один одному в такий спосіб. Значення менше `etalon` («хороші») в лівій частині пропускаються, а на «поганому» рух зупиняється. Аналогічно після цього у правій частині пропускаються значення, що більші `etalon`. Якщо **i** ще не став більше **j**, то це означає, що обидва вони вказують на «погані» значення. Ці значення обмінюються місцями, а курсори зсуваються назустріч один одному. Рух курсорів продовжується, поки **i** не стане більше **j**. Тоді всі елементи від **m[Left]** до **m[j]** мають значення не більше `etalon`, а всі від **m[i]** до **m[Right]** – не менше. Ці дві ділянки, якщо їх довжина більша 1, поділяються й сортуються рекурсивно. Якщо ж ділянка має довжину 1, то її вже відсортовано.

Оформимо сортування частини масиву **m[Left]**, ..., **m[Right]** такою функцією.

```
void QuickSort(int m[], int Left, int Right)
{
  // Встановлення значень «лівого» та «правого» курсорів
  int i=Left, j=Right;
  // Знаходження еталону
  int etalon=m[(Left+Right)/2];
```

```
  while (i<j)
  {
    // Пошук елементів, що підлягають переміні
    while (m[i]<etalon) i++;
    while (m[j]>etalon) j--;
    if (i<=j)
    {
      // Переміна місцями «поганих» елементів
      swap(m[i],m[j]);
      i++; j--;
    }
  }
  // Сортування лівої частини масиву
  if (Left<j) QuickSort(m,Left,j);
  // Сортування правої частини масиву
  if (i<Right) QuickSort(m,i,Right);
}
```

Оцінимо складність наведеної реалізації швидкого сортування. Нехай масив m має N елементів. Припустимо, що в кожному виклику функції **QuickSort** еталонним значенням буде найменше (або найбільше) на ділянці від **m[Left]** до **m[Right]**. Тоді розбиття ділянки масиву довжиною *Len* дає ділянки довжиною 1 та *Len*–1. Оскільки поступово $Len = N$, N–1, ..., 2, глибина рекурсії досягає $O(N)$, а сумарна складність має оцінку $O(N)+O(N–1)+...+O(2)=O(N^2)$.

Проте описаний найгірший випадок у реальних даних практично ніколи не трапляється. Для переважної більшості даних розбиття ділянки масиву дає дві ділянки з приблизно рівними довжинами. Тому розміри масивів, які сортуються, при переході на наступний рівень рекурсії зменшуються приблизно вдвічі. Отже, ***в середньому*** кількість рівнів рекурсії оцінюється як $O(\log N)$. Очевидно, що на кожному рівні рекурсії сумарна складність є $O(N)$, тому загальна складність ***в середньому*** має оцінку N*log(N). Численні практичні дослідження свідчать, що наведений алгоритм в середньому працює швидше, ніж інші алгоритми сортування, які мають складність найгіршого випадку N*log(N).

Для вибору еталонного значення існує багато способів, а не тільки вибір середнього елементу масиву. Теоретично, у якості еталонного значення можна брати взагалі будь-яке довільне значення, хоча очевидно, що найкращий результат буде отриманий, якщо еталонним буде ***медіана*** масиву (елемент, що у відсортованому масиві

буде знаходитись посередині). Дійсно, у цьому випадку масив буде розбиватися чітко на дві рівні частини і складність алгоритму буде максимально наближена до N*log(N). Дуже гарно, крім того, якщо еталонним буде одне зі значень сортованої частини масиву, адже тоді не потрібно турбуватися про запобігання виходу за межі сортованої частини й перевіряти додаткові умови.

Вибір «золотої середини» (значення, яке має бути посередині, або ***медіана***) вимагає чималих додаткових зусиль і може взагалі погіршити оцінку складності. Проте досить ефективним є вибір середнього з трьох значень — першого (з індексом *Left*), останнього (*Right*) та серединного ((*Left*+*Right*)/2). Для цього перед вибором еталона їх можна обміняти місцями у такий спосіб.

```
if m[Left] > m[(Left+Right)/2]
  swap(m[Left],m[(Left+Right)/2]);
if m[(Left+Right)/2] > m[Right]
  swap (m[(Left+Right)/2],m[Right]);
if m[Left] > m[(Left+Right)/2]
  swap(m[Left],m[(Left+Right)/2]);
etalon := m[(Left+Right)/2];
```

Швидке сортування не зберігає взаємного порядку однакових значень, тобто є ***нестійким***.

Ітеративна версія алгоритму. Як відомо, рекурсивні версії підпрограм працюють повільніше, ніж їх ітеративні аналоги. А тому запишемо ітеративну функцію, в якій явно реалізуємо обробку стека, необхідну для реалізації алгоритму.

Помітимо, що при заглибленні в рекурсію швидкого сортування, параметрами, що нею керують, є ліва та права границі сортованих підмасивів. В рекурсивній реалізації ці значення зберігаються у апаратному стеку (частині оперативної пам'яті). Тому в ітеративній реалізації потрібно моделювати роботу апаратного стеку за допомогою масиву **Stack**, в якому будемо зберігати ліві та праві межі сортованих підмасивів. Щоб масив **Stack** опрацьовувався за принципом стеку (перший прийшов останній пішов), уведемо поняття **top** (вершини стеку) – індекс цього масиву, з якого можна читати елемент. Якщо є потреба записати нове значення у масив **Stack**, то спочатку індекс **top** збільшується, а потім записується нове значення (імітація покладання нового елементу «поверх» попередніх). Якщо ж треба забрати елемент зі стеку, він береться з верхівки (ін-

декс **top**), після чого індекс **top** зменшується на 1. Коли цей індекс стане від'ємним – стек буде вважатися пустим.

```
void QuickSort_it(int m[], int N)
{
  int stack[1000][2];
  //Перший елемент стеку буде містити початкові границі
  int top=0,Left,Right,i,j,etalon;
  //Ліва границя початкового масиву - 0, права - N-1;
  stack[0][0]=0; stack[0][1]=N-1;
  while(top>=0)
  {
    Left=stack[top][0];
    Right=stack[top][1];
    top--;
    //Встановлення значень «лівого» та «правого» курсорів
    i=Left; j=Right;
    //Знаходження еталону
    etalon=m[(Left+Right)/2];
    while (i<j)
    {
      //Пошук елементів, що підлягають обміну
      while (m[i]<etalon) i++;
      while (m[j]>etalon) j--;
      if (i<=j)
      {
        //Обмін місцями «поганих» елементів
        swap(m[i],m[j]);
        i++; j--;
      }
    }
    //Запам'ятовування границь лівої частини масиву
    if (Left<j)
    {
      top++;
      stack[top][0]=Left;
      stack[top][1]=j;
    }
    //Запам'ятовування границь правої частини масиву
    if (i<Right)
    {
      top++;
      stack[top][0]=i;
      stack[top][1]=Right;
    }
  }
}
```

Сортування злиттям

В основі алгоритмів сортування «злиттям» лежить об'єднання двох упорядкованих послідовностей в одну. Це схоже на перебудову двох колон учнів, вишикуваних за зростом, в одну. Учні, перші у своїх колонах, порівнюються; вищий стає у нову колону, інший залишається першим у своїй. Після цього в колоні, з якої пішов учень, наступний за зростом учень стає першим. Знову порівнюються перші, і так вони діють, поки в одній з колон нікого не залишиться — тоді решта іншої колони перейде у хвіст нової без зміни порядку.

Нехай у масиві m з елемента **m[Left]** починається упорядкована ділянка довжиною **Middle**–**Left**+**1**, а з елемента **m[Middle+1]** – ділянка довжиною **Right**–**Middle**. Під упорядкованою ділянкою (відрізком або серією) розуміємо ділянку, яка не є частиною іншої упорядкованої ділянки. Наприклад, довжина таких упорядкованих ділянок дорівнює **Middle**–**Left**+**1** = **3** і **Right**–**Middle** = **3**.

1	**3**	**13**	**2**	**5**	**19**
Left		**Middle**	**Middle+1**		**Right**

Вони об'єднуються в таку ділянку довжиною **Right**–**Left**+**1** у допоміжному масиві **temp**.

1	**2**	**3**	**5**	**13**	**19**
Left		**Middle**	**Middle+1**		**Right**

Розглянемо процедуру злиття в масив temp пари суміжних ділянок масиву m, в якому ліва містить індекси від **Left** до **Middle**, а права — від **Middle**+**1** до **Right**.

```
void Merge(int m[],int temp[],int Left,int Middle,int Right)
{
  int i=Left, j=Middle+1;
  for (int k=Left; k<=Right; k++)
  {
    if(i>Middle)
    {
      //Перша ділянка закінчилася
      temp[k]=m[j]; j++;
    }
    else if (j>Right)
    {
      //Друга ділянка закінчилася
      temp[k]=m[i]; i++;
    }
```

```
      else if(m[i]>m[j])
      {
        //Елемент першої ділянки більший за елемент другої
        temp[k]=m[j]; j++;
      }
      else
      {
        //Елемент другої ділянки більший за елемент першої
        temp[k]=m[i]; i++;
      }
    }
  }
```

Очевидно, тіло циклу виконується **Right**–**Left**+**1** разів, і щоразу виконується $O(1)$ елементарних дій. Отже, складність виконання виклику функції **merge** є $O($**right**–**left**$+1)$.

Тепер можна використати цей алгоритм для сортування довільного масиву. Для цього у заданому масиві шукається пара суміжних відсортованих ділянок, які зливаються у допоміжний масив за допомогою наведеної функції **merge**. Потім відбувається пошук та об'єднується наступна пара тощо. Наприкінці масиву може залишитися ділянка, яка не має пари, тоді її потрібно скопіювати без змін.

Після першого проходження масивом може трапитися, що він ще не повністю буде відсортований, лише відсортовані ділянки стануть довшими. Тоді на наступному кроці потрібно виконати аналогічне злиття пар ділянок допоміжного масиву в основний. Кроки будуть повторюватися, доки якийсь із масивів не перетвориться на одну упорядковану ділянку. Якщо це допоміжний масив (на непарному кроці), він копіюється в основний.

Продемонструємо виконання алгоритму на масиві $m =$ **11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1** довжиною $N = 11$. Упорядковані послідовності будемо виділяти дужками < >, пари ділянок, які об'єднуються, відокремлювати «;», **temp** — ім'я допоміжного масиву.

A = <**11**><**10**>; <**9**><**8**>; <**7**><**6**>; <**5**><**4**>; <**3**><**2**>; <**1**>
temp = <**10, 11**><**8, 9**>; <**6, 7**><**4, 5**>; <**2, 3**><**1**>
A = <**8, 9, 10, 11**><**4, 5, 6, 7**>;<**1, 2, 3**>
temp = <**4, 5, 6, 7, 8, 9, 10, 11**><**1, 2, 3**>
A = <**1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11**>

Як бачите, масив відсортовано за чотири кроки злиття.

Після першого кроку злиття довжина упорядкованих ділянок не менше 2 (за винятком, можливо, «хвоста» довжиною 1), після другого – не менше 4 (крім, можливо, «хвоста» меншої довжини) тощо. Отже, після i-го кроку довжина упорядкованих ділянок, крім, можливо, «хвоста», не менше 2^i. Нехай N – кількість елементів масиву. На останньому, k-му, кроці об'єднуються дві ділянки; перша з них має довжину не менше 2^{k-1}, причому $2^{k-1}<N$. Отже, кількість кроків $k<\log N+1$. За рахунок можливого додаткового копіювання кількість кроків треба збільшити на 1, але оцінка $O(\log N)$ збережеться. На кожному кроці загальна кількість елементарних дій є $O(N)$, тому складність алгоритму – $O(N*\log N)$.

Реалізуємо алгоритм сортування злиттям за допомогою функції **sortByMrg**. Крок злиття ділянок одного масиву в інший оформимо функцією **mergeStep**, причому вона буде повертати ознаку того, що на кроці злиття було знайдено хоча б одну пару упорядкованих ділянок, а границі відсортованих ділянок будуть передаватися з функції за посиланням. Якщо пару не знайдено, то масив, початковий у виклику функції, відсортовано. На непарних кроках злиття функція **mergeStep** виконує злиття ділянок основного масиву m у допоміжний масив **temp**, на парних – навпаки. Якщо масив, початковий у виклику **mergeStep**, виявився відсортованим на парному кроці злиття, значить, це масив **temp**, і його треба скопіювати в основний масив **m**. А якщо на непарному кроці, то це масив **m**, і більше нічого робити не треба.

Пару суміжних упорядкованих ділянок N-елементного масиву (перший із них починається індексом **Left**) шукає функція **findPair**. Вона повертає ознаку того, що пару знайдено. Праві межі ділянок зберігаються в її параметрах **Middle** і **Right**. Якщо після її виклику справджується умова (**Left==0 && Right==N–1**), то пару не знайдено, тобто масив відсортовано.

Для злиття використовуємо функцію **merge** (див. вище). Допоміжна функція копіювання ділянки масиву в інший масив **copyAr** є очевидною.

```
bool findPair(int m[],int N, int Left, int & Middle, int &Right)
{
  //Функція повертає ознаку того, що пару суміжних
  //упорядкованих ділянок знайдено
```

```
  if (Left>=N) return false; //Масив закінчився
  //Пошук початку другої упорядкованої ділянки
  Middle=Left;
  while(Middle<N-1 && m[Middle]<=m[Middle+1]) Middle++;
  if (Middle==N-1)
  {
    //Другої ділянки немає
    Right=N-1;
    return false;
  }
  //Пошук закінчення другої упорядкованої ділянки
  Right=Middle+1;
  while (Right<N-1 && m[Right]<=m[Right+1]) Right++;
  return true;
}
void copyAr(int m[], int temp[], int Left, int Right)
{
  for (int i=Left; i<=Right; i++)
    temp[i]=m[i];
}
bool mergeStep(int m[], int temp[], int N)
{
  //функція повертає ознаку того, що злиття
  //масиву m у масив temp відбулося
  int Left=0, Middle, Right;
  while (findPair(m,N,Left,Middle,Right))
  {
    Merge(m,temp,Left,Middle,Right);
    Left=Right+1;
  }
  //Останній виклик findPair повернув false
  if (Left==0 && Right==N-1) return false;//Масив відсортовано
  if (Left<=N-1)copyAr(m,temp,Left,N-1); //Копіювання «хвоста»
  return true;
}
void SortByMrg(int m[], int N)
{
  int temp[100000]; //Допоміжний масив
  int step=0;    //Номер кроку
  bool notSorted=true; //Ознака невідсортованості
  while (notSorted)
  {
    step++;
    if (step%2!=0)
      //Ділянки масиву m «зливаються» у масив temp
      notSorted=mergeStep(m,temp,N);
```

```
      else
        //Ділянки масиву temp «зливаються» у масив m
        notSorted=mergeStep(temp,m,N);
    }
    //Після злиття один з масивів відсортовано
    if (step%2==0) copyAr(temp,m,0,N-1);
  }
```

Алгоритм сортування злиттям сортує масиви з великою довжиною значно швидше, ніж базові методи. Проте його головний недолік – йому потрібен ***додатковий масив розміром*** N. Крім того, сортування злиттям зберігає взаємне розташування однакових значень, тобто ***є стійким***.

В алгоритмах, заснованих на злитті, елементи послідовності обробляються у порядку їх розташування, тобто, по суті, без прямого доступу. Тому саме ці алгоритми використовуються для зовнішнього сортування.

Сергій Жуковський. Бітові операції

Основи порозрядної арифметики

Для забезпечення можливості роботи з двійковими розрядами у мові програмування C++ існують бітові операції, які є основою бітової або порозрядної арифметики. Розглянемо основні бітові операції:

Назва операції	Запис операції C++
Бітове заперечення, або інверсія	~
Бітове додавання	|
Бітове множення	&
Порозрядний ксор	^
Зсув вліво	<<
Зсув вправо	>>

Бітові операції застосовуються тільки до цілих чисел (у мові C++ також для типу char). При цьому потрібно враховувати відмінність у поданні цілих та беззнакових (у C++ unsigned) цілих чисел. Розглянемо приклади застосування цих операцій:

Запишемо двохбайтове (16-бітове) ціле знакове число у 2-вій системі числення.

short

1 – 0000 0000 0000 0001
32767 – 0111 1111 1111 1111
–32768 – 1000 0000 0000 0000 (32767+1)
–1 – 1111 1111 1111 1111

Запишемо двохбайтове (16-бітове) ціле беззнакове число у 2-вій системі числення.

unsigned short

1 – 0000 0000 0000 0001
32767 – 0111 1111 1111 1111
32768 – 1000 0000 0000 0000 (32767+1)
65535 – 1111 1111 1111 1111

Функція переведення числа з 10-вої в 2-ву систему числення

C++

```
int bin(int n)
{
  if(n>1) bin(n / 2);
  cout<<(n % 2);
}
```

```
int bin(int n)
{
  if(n>1) bin(n>>1);
  cout<<(n&1);
}
```

Приклад запису основних бітових операцій мовою C++.

```
#include <iostream>
using namespace std;
int main()
{
  int k=3,r,x=122,a=45,b=23,c;
  x = ~z;
  r1=a^(1<<k);
  cout<<r<<endl;
  cout<<x<<endl;
  cout<<r1<<endl;
  c=a | b;
  cout<<c<<endl;
  c=a&b;
  cout<<c<<endl;
  c=a^b;
  cout<<c<<endl;
  return 0;
}
```

Наведемо перелік основних задач на використання бітових операцій:

1. **Формула $x \cdot 2^k$ може бути виражена в C++ як** $x<<k$.
2. **Знайти цілу частину від ділення на k-у степінь двійки.**
   x1=x>>2;
3. **Визначити значення молодшого біта числа.**
   bit=x&1;
4. **Установити крайній справа нульовий біт (необов'язково останній) рівним одиниці.**
   x1=x|(x+1);

Наприклад: Нехай $x=103_{10}=1100111_2$. Отримаємо результат: $x=1100111_2$ or $(1100111_2+1)=1101111_2=111_{10}$

5. **Перевірити чи число x є числом виду:** 2^n-1.
   x1=x&(x+1) – значення вказаного виразу буде рівно 0, якщо $x=2^n-1$.

6. Виділити крайній справа нульовий біт числа.

x1=(~x)&(x+1);

Наприклад: Нехай $x=103_{10}=1100111_2$. Отримаємо результат: x = not (1100111_2) and (1100111_2+1) = (0011000_2) and (1101000_2) = $0001000_2=8_{10}$

7. Виділити крайній справа одиничний біт числа.

x1=x&((~x)+1);

8. Установити k-й біт числа рівний одиниці.

Приклад на C++: x1=x|(1>>(k–1));

Пояснення: для числа $69_{10}=0100\mathbf{0}101_2$ при k=4 результат матиме вигляд $0100\mathbf{1}101_2=77_{10}$.

9. Обнулити k-й біт числа.

x1=x&(~(1<<(k–1)));

Пояснення: для числа $77_{10}=0100\mathbf{1}101_2$ при k=4 результат матиме вигляд $0100\mathbf{0}101_2=69_{10}$.

10. Виконати інверсію k-го біта числа.

x1=x^(1<<(k–1));

Пояснення: для числа $77_{10}=0100\mathbf{1}101_2$ при k=4 результат матиме вигляд $0100\mathbf{0}101_2=69_{10}$.

11. Виділити k-й біт числа.

Формула: bit=(x>>(k–1))&1;

Пояснення: для числа $79_{10}=01001111_2$ при k=3 результат рівний 1.

12. Перетворити на одиничний крайній справа неперервний ланцюжок нульових бітів.

x1=x|(x–1);

Пояснення: число $168_{10}=10101000_2$ перетворюється у число $10101111_2=175_{10}$.

13. Продублювати правий крайній одиничний біт вправо, при цьому обнулити всі інші біти.

x1=x^(x–1);

Пояснення: число $168_{10}=10101000_2$ перетворюється у число $00001111_2=15_{10}$.

14. Встановити кінцеві нульові біти числа рівними одиниці, при цьому всі інші перетворити на нульові. – функція Фенвіка

x1=(~x)&(x-1);

Пояснення: число $168_{10}=10101000_2$ перетворюється у число $00000111_2=7_{10}$.

15. Перетворити на нульові k молодших бітів цілого додатного числа.

x1=(x)&((–1)<<k);

Пояснення: при k=3 число $76_{10}=01001100_2$ перетворюється у число $01001000_2=72_{10}$.

16. Обнулити крайній справа неперервний ланцюжок одиничних бітів.

x1=((x|(x–1))+1)&x;

Пояснення: для числа $76_{10}=01001100_2$ результат матиме вигляд: $64_{10}=01000000_2$.

Задачі

$2^k + 2^n$

https://www.e-olymp.com/uk/problems/5314

Задано два різних числа k та n. Виведіть значення $2^k + 2^n$, використовуючи лише бітові операції.

Вхідні дані: Два різних числа k та n ($0 \le k, n \le 30$).

Вихідні дані: Виведіть число $2^k + 2^n$.

Input 0 1 Output 3

Скористаємся побітовими операціями. Зсунемо 1 на k біт ліворуч і іншу одиницю на n біт ліворуч і два результати додамо (1 << k+1 << n).

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  int k,n,r;
  cin>>k>>n;
  r=(1<<k)+(1<<n);
  cout<<r<<endl;
  return 0;
}
```

Встановити біт

Задано цілі числа a та k. Виведіть число, яке отримується з a встановленням значення k-го біта в 1.

Вхідні дані: В одному рядку задано два числа a та k ($0 \le a \le 10^9$).

Вихідні дані: Виведіть число a зі встановленим k-им бітом.

Input 12 1 Output 14

Потрібно поставити 1 на k-й біт зліва.

Це можна зробити за допомогою такого запису a|(1<<k)

```
#include <iostream>
using namespace std;
int main()
{
  int k,a,r;
  cin>>a>>k;
  r=a|(1<<k);
  cout<<r<<endl;
  return 0;
}
```

Інвертувати біт

Задано цілі числа a та k. Виведіть число, яке отримується з a інвертуванням k-го біта.

Вхідні дані: В одному рядку задано два числа a та k ($0 \le a \le 10^9$).

Вихідні дані: Виведіть число a з інвертованим k-им бітом.

Input 15 2 Output 11

Щоб інвертувати біт, потрібно виконати операцію виключне або "^" даного біта з 1. Отже, результат a^(1<<k).

```
#include <iostream>
using namespace std;
int main()
{
  int a,k,r;
  cin>>a>>k;
  r=a^(1<<k);
  cout<<r<<endl;
  return 0;
}
```

Значення біта

https://www.e-olymp.com/uk/problems/5317

Дано цілі числа a та k. Виведіть значення k-го біта числа a, що дорівнює 0 або 1.

Вхідні дані: В одному рядку задано два числа a та k ($0 \le a \le 10^9$).

Вихідні дані: Виведіть значення k-ого біта числа a.

Input 179 0 Output 0

Зсунемо біти числа на k бітів праворуч і визначимо, яким буде крайній біт в утвореному числі, за допомогою операції & 1. (a>>k) & 1.

```
#include<iostream>
using namespace std;
int main()
{
  int a,k,r;
  cin>>a>>k;
  r=(a>>k)&1;
  cout<<r<<endl;
  return 0;
}
```

Обрізати старші біти

https://www.e-olymp.com/uk/problems/5318

Задано ціле число a та натуральне число k. Виведіть число, яке складається лише з k останніх біт числа a (тобто обнуліть усі біти числа a, крім останніх k).

Вхідні дані: В одному рядку знаходиться два числа a та k ($0 \le a \le 10^9$).

Вихідні дані: Виведіть число a з обнуленими бітами, крім останніх k.

Input 126 3 Output 6

Щоб обнулити біти, залишивши k останніх біт числа, потрібно виконати побітову операцію & (і) з числом, що складається з k одиниць у двійковому записі. Це число можна утворити за допомогою виразу (1<<k)–1.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  int n=0;
  int k=0;
  cin>>a>>k;
  int r = a & ((1<<k)-1);
  cout<< r <<endl;
  return 0;
}
```

Обнулити біт

https://www.e-olymp.com/uk/problems/5319

Задано цілі числа a та k. Виведіть число, яке отримується з a скиданням значення k-го біта в 0. Молодший біт має номер 0.

Вхідні дані: Два числа a та k ($0 \le a \le 10^9$).

Вихідні дані: Виведіть число з обнуленим k-им бітом.

Input 14 1 Output 12

Щоб обнулити k-й біт числа a, потрібно виконати операцію &(і) з числом яке складається з одиниць,і лише на k-й позиції стоїть 0. Щоб створити таке число, виконаємо операцію 1<<k інвертуємо "~" число (замінимо всі 0 на 1 і 1 на 0).

```
#include <iostream>
using namespace std;
int main()
{
  int a,k,r;
  cin>>a>>k;
  r=a&~(1<<k);
  cout<<r<<endl;
  return 0;
}
```

Функція Фенвіка

https://www.e-olymp.com/uk/problems/1648

Значенням функції Фенвіка для числа n називається максимальна степінь двійки, на яку націло ділиться число n. За заданим числом nвизначити для нього значення функції Фенвіка.

Рекомендується зробити цю задачу без використання циклів.

Вхідні дані: Одне число n ($0 < n \le 2^{31}–1$).

Вихідні дані: Виведіть значення функції Фенвіка для числа n.

Input 12 Output 4

Розглянемо приклад з умови. 12 у двійковій системі числення буде дорівнювати 1100. Нам потрібно отримати число $8_{10}=100_2$. Віднімемо від даного числа 1. В результаті всі остання нулі перетворяться на 1, а остання одиниця на нуль. 12–1=11, (1011_2). Інвертуємо "~" це число (0100) і виконаємо операцію &(і) з початковим числом n&(~(n–1)).

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  int n,k=0,r;
  cin>>n;
  r=n & (~(n-1));
  cout<<r<<endl;
  return 0;
}
```

Циклічні зсуви

https://www.e-olymp.com/uk/problems/27

Запишемо ціле десяткове число n у двійковій системі і утворимо всі ліві циклічні зсуви числа n, при яких перша цифра числа переноситься в кінець числа.

Наприклад, якщо n = 11, в двійковій системі буде 1011, його циклічні зсуви: 0111, 1110, 1101, 1011. Максимальне значення m з усіх отриманих у такий спосіб чисел буде мати число $1110_2 = 14_{10}$.

Для заданого числа n визначити максимальне значення m.

Вхідні дані: Єдине число n ($1 \le n \le 2*10^9$).

Вихідні дані: Шукане число m.

Input 11 Output 14

Будемо виконувати циклічний зсув не з початку числа на кінець, а навпаки. Від цього результат не зміниться. Для цього визначимо найбільший біт. Для числа $11_{10}=1011_2$ це число $8_{10}=1000_2$. Це можна зробити так. За допомогою циклу порахувати кількість k значущих біт в числі, а потім зсунути 1 на (k–1) біт праворуч. А далі виконаємо наступну операцію стільки разів, скільки значущих бітів у числі. Якщо останній біт дорівнює 1, то зсунемо число на 1 біт праворуч і додаємо максимальний біт. Інакше зсунемо число праворуч на 1. Серед утворених чисел знайдемо найбільше.

```
#include <iostream>
using namespace std;
int main()
{
  int m,a,k=0,b,c,t,mm,i;
  cin>>a;
  m=a;
  b=a;
  while(b>0) // (*1)
  {
    k++;
    b=b>>1;
  }
  t=1<<(k-1);
  b=a;
  mm=a;
  for (i=1;i<=k;i++)
  {
    if(b&1)
      b=(b>>1)|t;
```

```
      else
        b=b>>1;
      if (b>mm)mm=b;
    }
    cout<<mm<<endl;
    return 0;
}
```

Прямокутник

https://www.e-olymp.com/uk/problems/769

Петрику потрібно вибрати на площині 4 точки так, щоб вони утворювали прямокутник зі сторонами, паралельними осям координат. Петрик вже вибрав три точки і впевнений, що він вибрав їх вірно. Допоможіть Петрику знайти координати четвертої точки.

Вхідні дані

Містить три рядки. Кожен рядок містить два натуральних числа, відокремлених пропуском – координати однієї з вершин прямокутника. Всі координати лежать у діапазоні від 1 до 1000.

Вихідні дані

Вивести два цілих числа-координати четвертої вершини прямокутника.

Input 5 5 Output 7 7
5 7
7 5

Проаналізуємо вхідні дані x1, y1, x2, y2, x3, y3. Серед 3-х координат x1, x2, x3 – дві координати однакові а третя відрізняється. Результат x4 буде дорівнювати тому з x, який немає пари. Аналогічно і координати y1,y2,y3. За властивістю операції ^(виключне або) a^a=0.

Отже:

x4=x1^x2^x3;

y4=y1^y2^y3;

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  int x1,y1,x2,y2,x3,y3,x4,y4;
  cin>>x1>>y1>>x2>>y2>>x3>>y3;
  x4=x1^x2^x3;
  y4=y1^y2^y3;
```

```
  cout<<x4<<' '<<y4<<endl;
  return 0;
}
```

Втрачена картка

https://www.e-olymp.com/uk/problems/2616

Для настільної гри використовуються картки з номерами від 1 до n (натуральне число n не перевищує 10^6). Одна картка загубилась. Знайдіть її.

Вхідні дані: Першим задано число n. Далі йдуть n–1 номерів карток, що залишились.

Вихідні дані: Вивести номер втраченої картки.

Input 5 1 2 3 4 Output 5

За властивістю операції ^(виключне або) a^a=0. Тому виконаємо операцію ^ для всіх чисел від 1 до n, а далі виконаємо операцію для всіх вхідних чисел. Результат і буде шукане число.

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
  int res=0,n,t;
  cin>>n;
  for (int i=1;i<=n;i++)
    res = res^i;
  for (int i=1;i<=n-1;i++)
  {
    cin>>t;
    res^=t;
  }
  cout<<res<<endl;
  return 0;
}
```

Велика різниця

https://www.e-olymp.com/uk/problems/5718

Задано натуральне число n. З цим числом дозволено нескінченну кількість разів проводити перестановку значущих бітів заданого числа, отримуючи таким чином нові числа.

Яку найбільшу різницю отриманих двох чисел можна отримати в результаті виконання цих операцій?

Вхідні дані

Одне натуральне число n ($1 \le n \le 2*10^9$).

Вихідні дані

Одне число – шукана «Велика різниця».

Input 19 Output 21

Порахуємо кількість значущих одиниць k1 і нулів k0 в двійковому записі числа.

Найменше число буде дорівнювати min_t=(1<<k1)–1. Максимальне число max_t =min_t<<k0.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  int n,k0=0,k1=0,min_t,max_t,res;
  cin>>n;
  while(n>0)
  {
    if( n&1 )
      k1++;
    else
      k0++;
    n>>=1;
  }
  min_t=(1<<k1)-1;
  max_t = (min_t<<k0);
  res=max_t-min_t;
  cout<<res;
  return 0;
}
```

Одиниці та сімки

https://www.e-olymp.com/uk/problems/7509

Знайти n-й член зростаючої послідовності натуральних чисел:

1 7 11 17 71 77 111 117 171 177 711 717...

Вхідні дані: Натуральне число n ($1 \le n \le 10^7$).

Вихідні дані: Вивести n-й член заданої послідовності.

Input 11 Output 7

Очевидно, що така послідовність схожа на двійкові числа, у яких 0 замінили на 1, а 1 на 7. Але є ще одна відмінність. Розглянемо такі послідовності:

n	n+1	bin(n+1)	bin(n+1)[2:]	u
1	2	0b10	0	1
2	3	0b11	1	7
3	4	0b100	00	11
4	5	0b101	01	17
5	6	0b110	10	71
6	7	0b111	11	77
7	8	0b1000	000	111
8	9	0b1001	001	117
9	10	0b1010	010	171
10	11	0b1011	011	177

Видаливши першу одиницю з двійкового зображення числа n+1, маємо послідовність 0 1 00 01 10 11 000 001 010 011 100 101, якщо ж тепер виконати заміну цифр, то отримаємо потрібний результат.

```
#include <bits/stdc++.h>
using namespace std;
void bin(int n)
{
  if(n>3) bin(n>>1);
  cout<<((n&1)? 7:1 );
}

int main()
{
  int n;
  cin>>n;
  n++;
  bin(n);
  cout<<endl;
  return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  bitset<32> a;
  int t;
  cin>>t;
  a=t+1;
  string s=a.to_string();
  while(s[0]=='0') s.erase(0,1);
  s.erase(0,1);
  for(int i=0; i<s.length();i++)
    if (s[i]=='0')
      s[i]='1';
    else
      s[i]='7';
  cout<<s<<endl;
  return 0;
}
```

1 Теоретичний матеріал

> Динамічне программування — це коли маємо одну велику задачу, яку неясно як розв'язувати, і розбиваємо її на менші задачі, які теж неясно як розв'язувати.
>
> Акім Кумок

Жартівливе «означення» з епіграфа стає майже правильним, якщо замінити слова́ *«які теж неясно як розв'язувати»* на *«які можна розв'язати так са́мо»*. Тому що найперший крок динамічного програмування — виділити *серію підзадач* однакового формулювання з різними параметрами (вони ж *«підзадачі різних розмірів»*).

Динпрогом («динпрог» та «ДП» — стандартні скорочення від «динамічне програмування») можна розв'язати далеко не всі задачі, й, дивлячись на задачу, не завжди легко зрозуміти, чи можна й чи варто розв'язувати її динпрогом. Більш того: навіть якщо відомо, що задача «на динпрог», це задає лише загальний напрям розробки алгоритму, а більшість деталей треба придумувати для кожної задачі по-своєму. Тому́ для вивчення ДП потрібно знайомитися як із теорією, так і з прикладами застосування. За все це ДП дуже люблять на олімпіадах. У практичному програмуванні, з тих самих причин, ДП швидше не люблять, але час від часу все ж використовують — хоча б тому, що бувають ситуації, коли такий розв'язок задачі значно ефективніший за будь-який інший правильний.

До детальнішого розгляду теорії перейдемо після того, як розглянемо класичну просту задачу, яку зручно розв'язувати са́ме динпрогом.

1.1 Задача «Платформи»

1.1.1 Базовий варіант задачі

Герой стрибає по платформам, що висять у повітрі. Він повинен перебратися від одного краю екрана до іншого. При стрибку з платформи на сусідню, герой витрачає $|y(2) - y(1)|$ енергії, де $y(1)$ і $y(2)$ — висо́ти цих платформ. Суперприйом дозволяє перескочити через платформу, але на це затрачається $3 \cdot |y(3) - y(1)|$ енергії. Суперприйом можна використовувати скільки завгодно (в т. ч. й 0) разів. Відомі висо́ти платформ у порядку зліва направо. Знайдіть мінімальну кількість енергії, достатню, щоб дістатися з 1-ї платформи до N-ї (останньої).

Щоб застосувати ДП, введемо серію підзадач *«Скільки енергії $E(i)$ необхідно, щоб дістатися з 1-ї платформи до i-ї?»* ($1 \leqslant i \leqslant N$).

Розв'язки 1-ї і 2-ї підзадач *тривіальні*:

- $E(1) = 0$ (рухатися не треба);
- $E(2) = |y(2) - y(1)|$ (можливий *тільки* стрибок з 1-ї).

Для подальших платформ, справедливо

$$E(i) = \min \left\{ \begin{array}{c} E(i-1) + |y(i) - y(i-1)|, \\ E(i-2) + 3 \cdot |y(i) - y(i-2)| \end{array} \right\}. \tag{1}$$

Такі формули (котрі виражають залежність розв'язків більших підзадач серії від менших) називають *рівнянням ДП*.

Розберемо докладніше, звідки взялося це рівняння:

$$E(i) = \underbrace{\min}_{\text{кращий зі способів}} \left\{ \begin{array}{lcl} \overbrace{E(i-1)}^{\text{досягти } (i-1)\text{-ї платформи}} & + & \overbrace{|y(i) - y(i-1)|}^{\text{і перестрибнути з неї на } i\text{-у}}, \\ \underbrace{E(i-2)}_{\text{досягти } (i-2)\text{-ї платформи}} & + & \underbrace{3 \cdot |y(i) - y(i-2)|}_{\text{і перестрибнути з неї на } i\text{-у}} \end{array} \right\} \tag{2}$$

Тепер ці формули можна перетворити у програму, приблизно так:

```
{ прочитали вхідні дані в масив y }
E[1]: = 0;
E[2]: = abs(y[2]-y[1]);
for i: = 3 to N do
  E[i]: = min(E[i-1] + abs(y[i]-y[i-1]),
              E[i-2] + 3*abs(y[i]-y[i-2]));
writeln(E[N]); { результат }
```

(Якщо нема готової функції `min`, можна реалізувати її самостійно, або переписати вибір меншого через `if`. Код написаний у припущенні, що і платформи, й індекси нумеруються з 1 (Паскаль так вміє). У більшості сучасніших мов програмування нумерація масивів починається з 0, тож це може призводити до не складни́х, але нудних технічних ускладень: треба або виділяти на один елемент більше (втрачаючи деякі бібліотечні засоби роботи зразу з усім масивом), або переписувати формули, зміщуючи індексацію. Але в нашій культурі все-таки не прийнято нумерувати з 0, тому в умовах та розборах задач буде переважно «людська» нумерація з 1, а питання, як це реалізувати мовою програмування, як правило, віддаватиметься на розсуд читачів.)

1.1.2 Ітеративна та рекурсивна реалізації ДП

Розглянутий підхід (спочатку заповнити розв'язки тривіальних підзадач, потім у циклі застосовувати рівняння ДП, спираючись на вже готові відповіді й отримуючи нові, доки не будуть розв'язані всі) називають *ітеративною* (або *ітераційною*, що те са́ме) реалізацією. Можливий і альтернативний підхід — реалізувати рівняння ДП рекурсивною функцією. Він нічим не кращий для конкретно цієї задачі про платформи, але може бути зручнішим для деяких інших задач.

На перший погляд, мало б вийти щось приблизно як на верхньому з фрагментів коду наступної сторінки. Але це *дуже погана реалізація*, яка працюватиме *значно* повільніше за ітеративну. Безпосередньо під цим фрагментом наведено процес виклику `calc_E(7)` у вигляді де́рева рекурсії. При $i \geqslant 3$, щоразу відбувається по два рекурсивних виклики. Наприклад, із $E(7)$ виходять лінія ліворуч-униз у $E(6)$ (`calc_E(i-1)`) та лінія праворуч-униз у $E(5)$ (`calc_E(i-2)`). Бачимо багатократне знаходження одних і тих самих відповідей: двічі виконується пошук $E(5)$ (який потребує сумарно 9 викликів рекурсивної функції), тричі — $E(4)$ (5 викликів), тощо. Через це, при зростанні N сумарна кількість викликів зростає катастрофічно швидко:

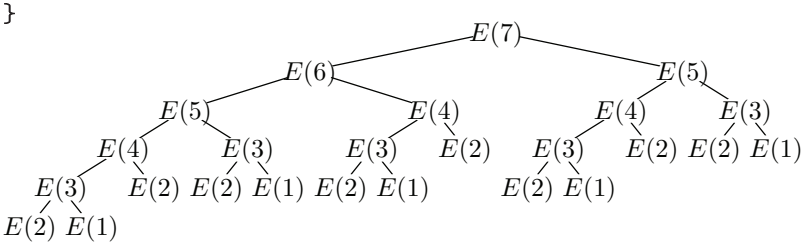
N	1	2	3	4	5	6	7	8	9	10	$\cdots$	20	$\cdots$	30	$\cdots$	40	$\cdots$
кіль-ть	1	1	3	5	9	15	25	41	67	109	$\cdots$	13529	$\cdots$	1664079	$\cdots$	204668309	$\cdots$

Тим не менш, релізація ДП через рекурсію можлива — якщо це рекурсія *із запам'ятовуваннями* (*memoized* recursion; не "memorized", а са́ме "memoized", так у англомовних першоджерелах). Її суть — перед запуском рекурсії перевірити, чи нема для цієї підзадачі готової (отриманої й запам'ятованої раніше) відповіді. Конкретний код рекурсії з запам'ятовуваннями для цієї ж задачі наведено на тій самій наступній сторінці, під щойно згаданим деревом рекурсії.

У наведеному прикладі виклику, `calc_E(N)` — сам виклик, він очевидний; `fill_n` — функція (з бібліотеки `algorithm` мови C++), яка заповнює вказаний масив `E` вказаними значеннями `-1`. Тобто, `E[i]==-1` виражає, що відповідне значення $E(i)$ ще не знайдене, і його треба знайти (для $i \geqslant 3$ — рекурсивно), будь-яке інше значення `E[i]` — що підзадача $E(i)$ вже розв'язувалася, її відповідь можна брати готову.

Тут є багато моментів, які можна реалізувати інакше. Можна дотриматися традиції сучасних мов програмування, змістивши нумерацію на `0..N-1`; можна інакше ініціалізувати масив значеннями `-1` та/або позначати «ще не вирішено» якось інакше (але дуже бажано, щоб ця

```
int calc_E(int i) {     // "Звичайна" рекурсія
  if (i == 1) return 0; // (без запам'ятовувань)
  if (i == 2) return abs(y[2]-y[1]);
  return min(calc_E(i-1) + abs(y[i]-y[i-2]),
             calc_E(i-2) + 3*abs(y[i]-y[i-1]));
}
```

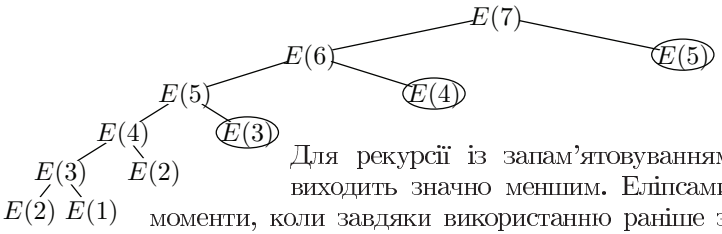


«Звичайна» рекурсія дуже багато разів розглядає ті самі підзадачі.

```
int calc_E(int i) { // Рекурсія із запам'ятовуваннями
  if(E[i] != -1) // якщо розв'язок вже відомий --
    return E[i]; // використати його
  if (i == 1)
    return E[1] = 0;
  if (i == 2)
    return E[2] = abs(y[2]-y[1]);
  E[i] = min(calc_E(i-1) + abs(y[i]-y[i-2]), // спочатку
             calc_E(i-2) + 3*abs(y[i]-y[i-1])); //запам'ятати
  return E[i]; // розв'язок, а вже потім повернути
}
```

Відповідно, виклик цієї функції має бути приблизно таким:

```
  fill_n(E, N+1, -1);
  out << calc_E(N) << endl;
```



Для рекурсії із запам'ятовуваннями дерево виходить значно меншим. Еліпсами виділені моменти, коли завдяки використанню раніше запам'ятованих розв'язків вдається уникнути рекурсивних викликів.

позначка не могла бути відповіддю вже розв'язаної підзадачі); тощо. То деталі. Важливо, щоб запам'ятовування було, і ним користувалися.

Швидкість виконання рекурсії з запам'ятовуваннями сильно залежить і від особливостей задачі, і від особливостей архітектури комп'ютера, на якому виконується програма, але, як правило, вона трохи повільніша за ітеративну реалізацію (бо накладні витрати на рекурсію більші, ніж на цикл). Дуже приблизно в середньому в 1,1–2 рази.

Навіщо тоді взагалі рекурсія із запам'ятовуваннями? У цій задачі вона і не потрібна. А в інших... По-перше, бувають заплутані серії підзадач, де важко розібратись, у якому порядку їх розв'язувати, щоб завжди спиратися на вже розв'язані. Рекурсія із запам'ятовуваннями дозволяє не думати над цим, само вийде, що для вже розв'язаних підзадач візьмуть готові відповіді, а ще не розв'язані розв'яжуть рекурсивно. По-друге, бувають задачі, при рекурсивному розв'язуванні яких можна бачити, що деякі підзадачі не потрібні (явно гірші за інші), і взагалі не розв'язувати їх, а при ітеративній реалізації цього не видно. В цьому випадку рекурсія з запам'ятовуваннями може бути швидшою.

1.1.3 Зворотній хід (формування розгорнутої відповіді)

Нехай у задачі питають не лише мінімальну кількість енергії, а ще й маршрут (послідовність платформ), який забезпечує досягнення фінішу за цю мінімальну кількість енергії. Наприклад:

Вхідні дані	Результати
6 *1 100 1 100 1 100*	*99* *4* *1 2 4 6*

Виведіть у 1-му рядку мінімальну кількість енергії, далі відповідний маршрут: у 2-му рядку — кількість пройдених платформ, у 3-му — послідовність номерів цих платформ. Якщо є різні маршрути з однаковими мінімальними витратами, виведіть будь-який один.

Є різні точки зору щодо того, чи слід включати таку складову в олімпіадні задачі. Дехто вважає, що формування розгорнутої відповіді більш технічна, ніж інтелектуальна робо́та, і їй не дуже-то місце на олімпіадах. Але на олімпіадах такі завдання трапляються, а на практиці розгорнута відповідь часто важливіша за числову: щоб доїхати у реальній мережі доріг, знати, де і куди повертати, важливіше, ніж знати абсолютно точну відстань.

Щоб забезпечити формування розгорнутої відповіді, можна при розв'язуванні (кожної) підзадачі запам'ятовувати не лише знайдену найкращу відповідь, а *ще* й *вибір, що призвів* до цього оптимума.

Зокрема, у цій задачі, крім масива `y` з висотами платформ і масива `E` з відповідями на питання *«Скільки енергії $E(i)$ необхідно, щоб дістатися з 1-ї платформи до i-ї?»*, ввести ще масив `prev`, де смисл `prev[i]` — *«З якої ($(i-1)$-ої чи $(i-2)$-ої) платформи слід перескакувати на i-ту, щоб досягти цих мінімальних затрат енергії $E(i)$?»*.

На етапі розробки рівняння ДП все лишається, як було. У коді доводиться відмовитися від зручного запису через виклик функції `min` і перейти до явного розгалуження (щоб знати, *який варіант* дав мінімум).

```
E1=E[i-1] + abs(y[i]-y[i-1]);
E2=E[i-2]+3*abs(y[i]-y[i-2]);
if (E1 < E2) {
  E[i] = E1;  prev[i] = i-1;
} else {
  E[i] = E2;  prev[i] = i-2;
}
```

Аналогічно, для більшої з тривіальних підзадач слід вказати не лише `E[2] = abs(y[2]-y[1])`, а ще й `prev[2] = 1`.

Коли масиви вже заповнені, робиться *зворотній хід*. Він починається з останньої підзадачі і «задкує» згідно зі значеннями масиву виборів: завдяки `prev[N]` стає відомо, звідки треба прибути на останню платформу, потім береться `prev` тієї платформи, і т. д. Так у результаті цих «задкувань» і отримується (у зворотньому порядку) весь маршрут.

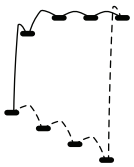
	1	2	3	4	5	6	7	8
y	3	1	4	1	5	9	2	6
E	0	2	3	2	6	10	15	19
prev	??	1	1	2	4	5	5	7

початок зворотнього ходу

Звідки можуть братися «різні шляхи з однаковими мінімальними витратами», якщо зворотній хід однозначно вказує, що робити? Неоднозначність з'являється не у зворотньому ході, а при виборі мінімума. Якщо змінити "`if(E1<E2)`" на "`if(E1<=E2)`", жоден елемент масива `E` не зміниться, а у масиві `prev` буде кілька змін; провівши той самий зворотній хід по зміненому масиву `prev`, отримаємо іншу послідовність.

1.1.4 А можна якось простіше?

Може якось і можна. Але ще ніхто не запропонував, як (щоб і простіше, і правильно). А проти найбільш «природних» (для тих, хто не знає ДП) «ідей» є контрприклади, які показують їхню неправильність.

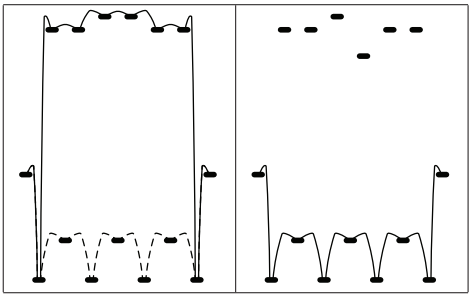


Хибна «ідея» № 1: *«щоразу вибирати, який з переходів (на наступну платформу чи через одну) потребу́є менше енергії, і стрибати туди»*. Розглянемо вхідні дані «8 платформ, висо́ти 10 15 9 16 8 16 7 16» (див. рис.). Суцільною лінією виділено оптимальний маршрут, з затратами енергії $1 \cdot 5 + 3 \cdot 1 + 3 \cdot 0 + 3 \cdot 0 = 8$. А ця «ідея» вибере маршрут, позначений штриховою лінією: з $(y_1 = 10)$ на $(y_3 = 9)$ за $3 \cdot |9 - 10| = 3$, бо це ніби «вигідніше», ніж на $(y_2 = 15)$ за $1 \cdot |15 - 10| = 5$; потім на $(y_5 = 8)$ за $3 \cdot |8 - 9| = 3$ ніби «вигідніше», ніж на $(y_4 = 16)$ за $1 \cdot |16 - 9| = 7$, потім на $(y_7 = 7)$ за $3 \cdot |7 - 8| = 3$ ніби «вигідніше», ніж на $(y_6 = 16)$ за $1 \cdot |16 - 8| = 8\ldots$ і останній стрибок на $(y_8 = 16)$, який потребу́є більше енергії $1 \cdot |16 - 7| = 9$, ніж увесь оптимальний маршрут.

Нездола́нна (така, що її неможливо «пофіксити» дрібними правками) проблема цієї «ідеї» — при остато́чних виборах стрибків не враховується, де розміщена платформа-фініш; отже, не може бути гарантій, що «дешевший» маршрут невідомо куди буде кращим са́ме для фініша.

Цю «ідею» не рятують такі модифікації, як «дивитись у зворотньому напрямку, від фінішу до старту», і навіть «подивитися двічі (від старту до фінішу й від фінішу до старту) й вибрати мінімум». Наприклад, можна розглянути вхідні дані (див. рис.; зображено, у дрібнішому масштабі, лише платформи), де такі фрагменти повторюються багатократно, причому іноді розвернуті, іноді ні.

Хибна «ідея» № 2: *«щоразу вибирати, яка платформа (наступна чи через одну) ближча (по вертикалі) до фініша, і стрибати туди»*.



Розглянемо вхідні дані «15 платформ, висо́ти 20 12 31 15 31 12 32 15 32 12 31 15 31 12 20» (див. рис. (лівий), смисл суцільної та пунктирної ліній той са́мий, де пунктирної не видно, там вона проходить по суцільній). За цією «ідеєю» з $(y_2 = 12)$ ніби вигідно стрибати на $(y_4 = 15)$, але маршрут через верхні платформи $(y_3 = 31) \to (y_5 = 31) \to (y_7 = 32) \to (y_9 = 32) \to$ $\to (y_{11} = 31) \to (y_{13} = 31)$ потребу́є значно менше затрат, ніж через нижні $(y_4 = 15) \to (y_6 = 12) \to (y_8 = 15) \to (y_{10} = 12) \to (y_{12} = 15)$ (а са́ме,

$0+3+0+3+0=6$ проти $9+9+9+9=36$), так що «локально неоптимальний» стрибок $(y_2=12)\to(y_3=31)$ окупається на інших етапах.

Звісно, окупається са́ме при тих вхідних даних; на правому рис. наведено приклад (відрізняється лише тим, що $y_9=29$ замість 32), коли не окупається («верхня послідовність» не настільки вигідна).

Дуже важливий висновок з порівняння цих рисунків — навіть початкові стрибки оптимального маршруту до фінішу *можуть* залежати від невеликих змін вхідних даних десь посередині чи майже наприкінці. Прості жадібні підходи (які намагаються робити остато́чний безповоротний вибір, проаналізувавши лише кілька платформ) правильно працювати з таким не вміють. А динамічне програмування вміє. Головним чином, тому, що «знайти $E(i)$» ще не означає, ніби ця платформа № i буде використана в оптимальному маршруті; підзадача просто розв'язана, щоб бути розглянутою при аналогічних розв'язуваннях для подальших платформ, а які з підзадач справді вибрані при формуванні оптимума цільової підзадачі, стане відомо аж наприкінці.

Попередній абзац пояснює зразу кілька особливостей ДП. «Ідею № 1» критикували, що при виборі стрибка не враховується розміщення фініша $y(N)$, рівняння ДП (1) теж його не враховує, але це не критикуємо? Нормально, бо (1) не робить остато́чного вибору, $y(N)$ не впливає на $E(i)$ (при $i<N$), але потім вплине на $E(N)$. Незручно формувати розгорнуту відповідь, «задкуючи» зворотнім ходом? А інакше не можна, бо, поки не дійшли до кінця, не знаємо, які з підзадач реально використані в «цільовій» підзадачі $E(N)$.

(Якщо дуже треба, щоб зворотній хід видав результати відразу в потрібному порядку, можна «розвернути» серію підзадач *«Скільки енергії $E(i)$ необхідно, щоб дістатися з поточної i-ї платформи до останньої N-ї?»* ($1\leqslant i\leqslant N$, підзадачі $E(N)$ та $E(N-1)$ тривіальні, $E(1)$ цільова́). Тоді основний етап ДП відбуватиметься справа наліво, зворотній хід зліва направо, але все одно назустріч основному етапу.)

1.1.5 Платформи з квадратичними вартостями стрибків — циклічні залежності між підзадачами

Внесемо дрібну, на перший погляд, зміну в умову задачі:

Витрати енергії	*було*	*стало*
на звичайний стрибок	$\lvert y(2)-y(1)\rvert$	$(y(2)-y(1))^2$
на Суперприйом	$3\cdot\lvert y(3)-y(1)\rvert$	$3\cdot(y(3)-y(1))^2$

Несподівано, така зміна виявляється дуже істотною!

Герою *не заборонено стрибати назад*. Наприклад, при $y_1 = 10$, $y_2 = 32$, $y_3 = 18$, $y_4 = 40$ сумарні витрати енергії для маршруту $1 \to 3 \to 2 \to 4$ становлять $3 \cdot 64 + 196 + 3 \cdot 64 = 580$, що значно менше, ніж $484 + 196 + 484 = 1164$ (маршрут $1 \to 2 \to 3 \to 4$) і дещо менше, ніж $484 + 3 \cdot 64 = 3 \cdot 64 + 484 = 676$ (маршрути $1 \to 2 \to 4$ та $1 \to 3 \to 4$).

Наївна спроба змінити рівняння ДП на

$$E(i) = \min \left\{ \begin{array}{l} E(i-1) + (y(i) - y(i-1))^2, \\ E(i-2) + 3 \cdot (y(i) - y(i-2))^2, \\ E(i+1) + (y(i) - y(i+1))^2, \\ E(i+2) + 3 \cdot (y(i) - y(i+2))^2 \end{array} \right\} \tag{3}$$

не призводить до негайного успіху. Це в деякому смислі правильно, але тут є циклічні залежності (наприклад, $E(4)$ залежить від $E(3)$, а $E(3)$ — від $E(4)$). І як таке програмувати? Коли можна припинити спиратися на іншу підзадачу й оголосити значення остаточно знайденими? Хоч циклами, хоч рекурсією — все одно неясно.

То що робити?

Варіант А: вирішувати задачу, взагалі не користуючись ДП. Наприклад, алгоритмом Дейкстри: вершини графа — платформи, ре́бра — можливості потрапити з платформи на платформу одним стрибком, довжина ребра́ — витрати енергії на відповідний стрибок.

Варіант Б: помітити особливі властивості задачі, які компенсують циклічність залежностей, і все ж застосувати ДП.

Варіант А цілком вартий уваги. Якщо ре́бра мають невід'ємні довжи́ни й існують циклічні залежності — це ситуація, де алгоритм Дейкстри часто доречний, а динпрог — невідомо, чи застосовний.

Плануючи застовувати алгоритм Дейкстри до цієї задачі, варто помітити, що граф тут розріджений (при N вершинах всього $2N - 3$ неорієнтованих ребер), тому на час виконання сильно впливатиме, чи робити вибір чергової поточної вершини шляхом перебору вершин (як спочатку і пропонував сам Дейкстра), чи застосувати пізніше розроблені оптимізації, головним чином `priority_queue` (вона ж піраміда) або іншу структуру даних, що дозволяє швидко знаходити мінімум.

Але алгоритм Дейкстри не є зараз предметом розгляду, так що перейдемо до варіанту Б і подивимося, що ж там за особливі властивості. Конкретно у цій задачі це такі два спостереження:

1. Безглуздо розглядати маршрути, які повторно відвідують ту са́му платформу — якщо фрагменти між цими повторами «вирізати», витрата енергії *не збільшиться*.
2. Послідовність $(+2)$, (-1), $(+2)$ — *єдина*, яка використовує стрибки назад, але не створює циклів.

(В інших задачах, де є аналогічні властивості, вони можуть бути зовсім іншими. А у багатьох задачах їх просто нема (і циклічність залежностей таки унеможливлює ДП). Тобто, пошук таких властивостей — зовсім не типовий, а чи то творчий, чи то погано алгоритмізований.)

Завдяки цим спостереженням, можна вважати, ніби рухатися дозволено тільки вперед, але є три види стрибків:

1. $(i-1)\to i$, витрати енергії $(y(i)-y(i-1))^2$;
2. $(i-2)\to i$, витрати енергії $3\cdot(y(i)-y(i-1))^2$;
3. $(i-3)\to i$, витрати такі ж, як для $(i-3)\to(i-1)\to(i-2)\to i$, тобто $3\cdot\big(y(i-1)-y(i-3)\big)^2+\big(y(i-2)-y(i-1)\big)^2+3\cdot\big(y(i)-y(i-2)\big)^2$.

Тепер можна написати програму, що відрізняється від базового варіанту з розд. 1.1.1 лише громіздкішим рівнянням ДП (мінімум не з двох, а з трьох варіантів), і підзадачею $E(3)$, де відбувається вибір кращого з двох варіантів. Така програма працюватиме швидше, ніж алгоритм Дейкстри ($\Theta(N)$ проти $O(N\log N)$ чи $O(N^2)$).

Повертатись не заборонено і в початковій версії задачі (з модулями). Може, багато років усі розв'язують її неправильно, не враховуючи варіант $(i-3)\to(i-1)\to(i-2)\to i$? Ні, там усе гаразд: з *тими* витратами енергії, стрибати назад якщо і можна, все одно невигідно.

$$\underbrace{3}_{=2+1}\cdot|y(i-2)-y(i)|+|y(i-1)-y(i-2)|+3\cdot|y(i-3)-y(i-1)|=$$

$$=\underbrace{2\cdot|y(i-2)-y(i)|}_{\geqslant 0}+\underbrace{|y(i)-y(i-2)|+|y(i-2)-y(i-1)|}_{\geqslant|y(i)-y(i-1)|}+$$
$$+3\cdot|y(i-3)-y(i-1)|\geqslant$$

$$\geqslant|y(i)-y(i-1)|+3\cdot|y(i-3)-y(i-1)|.$$

Тобто, розглядати $(i-3)\to(i-1)\to(i-2)\to i$ нема сенсу, бо $(i-3)\to(i-1)\to i$ (при формулах з модулями) гарантовано *не гірше*.

1.2 Загальні умови застосовності і доцільності ДП

Далеко не кожну задачу можливо розв'язати використанням ДП. Ці 6 пунктів описують, яким умовам повинна задовольняти задача, щоб її можна і варто було розв'язувати са́ме динпрогом.

1. У задачі можна виділити *однотипні підзадачі* різних розмірів.
2. Серед виділених підзадач є *тривіальні*, що мають малий розмір і очевидне рішення (або очевидну готову відповідь).
3. Решта підзадач залежать від інших однотипних підзадач, причому ці залежності або не містять циклів, або мають особливі властивості, які компенсують цикли.
4. Оптимальне рішення підзадачі більшого розміру можна побудувати *з оптимальних рішень* менших підзадач (а не оптимальні рішення не знадобляться).
5. Одні й ті ж менші підзадачі використовують при вирішенні різних більших підзадач (*підзадачі перекриваються*).
6. Кількість різних підзадач помірна, і для запам'ятовування результатів потрібно *не надто багато пам'яті*.

Пункти 1–2 очевидні. Але наголосимо, що мова йде про *однотипні* підзадачі, де словесне формулювання однакове, лише підставляється деякий параметр (чи кілька параметрів).

Про пункт 3 йшлося у розд. 1.1.5.

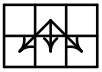
Пункт 4 називають *принципом оптимальності* або *принципом Беллмана*. Часто са́ме він є найскладнішим з усіх пунктів. Простим прикладом його застосування є (2) з розд. 1.1.1 як обґрунтування (1), а для глибшого розуміння варто ознайомитися з обґрунтуванням (5) у розд. 1.3.1, а також міркуваннями з початку розд. 1.3.2 та в деяких задачах другого дня, де вимоги цього пункту порушені.

Перекривання з пункту 5 згадувались у розд. 1.1.2 як причина запам'ятовувань у разі рекурсивної реалізації. Цей пункт 5 — єдиний, порушення якого не забороняє застосувати ДП. Але дає підстави сумніватися, *чи варто* це робити. Якщо багатократного використання не буває — навіщо запам'ятовувати? Чи не краще реалізувати просту рекурсію (без запам'ятовувань), або однопрохідний жадібний алгоритм, чи ще якусь (простішу за ДП) альтернативу?

Пункт 6 зрозумілий і без пояснень.

1.3 Задача MaxSum

1.3.1 Базовий варіант задачі



Вхідні дані	Результати
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	42

Є прямокутна таблиця розміром N рядків на M стовпчиків. У кожній клітинці записано ціле число. Через неї потрібно пройти 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. «Нижньо-сусідня» означає, що з клітинки (i,j) можна перейти у $(i+1,j-1)$, або у $(i+1,j)$, або у $(i+1,j+1)$ (див. також рис.), але не виходячи за межі таблиці (при $j=1$ перший з наведених варіантів стає неможливим, а при $j=M$ — останній).

Напишіть програму, яка знаходитиме максимально можливу суму значень пройдених клітинок серед усіх допустимих шляхів.

Серія підзадач — *«Яку максимальну суму $S(i,j)$ можна набрати, пройшовши найкращий допустимий шлях до клітинки (i,j)?»*. (З якої клітинки верхнього ряду починати — у серії підзадач *не* фіксується, і це добре: отримаємо відразу найкращий серед усіх варіантів.)

Цільова задача *не* є однією з підзадач серії, але легко отримується з розв'язків серії, як $\max_{1\leqslant j\leqslant M} S(N,j)$. Адже, якщо $S(N,1)$ — максимальна сума серед тих шляхів, що закінчуються 1-ю клітинкою останнього рядка, $S(N,2)$ — серед тих, що закінчуються 2-ю, і т. д., а шуканий шлях закінчується в одній із цих клітинок, то максимум серед $S(N,j)$ $(1\leqslant j\leqslant M)$ якраз і буде максимумом серед взагалі всіх шляхів.

Тривіальні підзадачі мають вигляд

$$S(1,j)=a(1,j) \qquad \text{для всіх } 1\leqslant j\leqslant M, \tag{4}$$

тобто 1-й рядок переписується зі вхідних даних. Це правильно, бо шлях, що задовольняє правилам і закінчується у клітинці верхнього рядка, завжди існує, єдиний і складається з самóї лише цієї клітинки.

Рівняння ДП:
$$S(i,j)=a(i,j)+\max\begin{pmatrix} S(i-1,j-1), \\ S(i-1,j), \\ S(i-1,j+1) \end{pmatrix} \quad \begin{array}{l}\text{(пропускаючи варіанти, які виводять}\\ \text{за межі таблиці)}\end{array} \tag{5}$$

Чому це правильно? По-перше, сума уздовж будь-якого шляху, що закінчується клітинкою (i,j) (при $i>1$), дорівнює сумі всього шляху,

крім останньої клітинки, плюс значення самóї клітинки. А значення самóї клітинки не залежить від того, звідки в неї прийти. Отже, для всіх можливих шляхів, значення самóї клітинки є однаковим доданком, і для максимізації суми всього шляху ніколи не вигідно приходити у передостанню клітинку не оптимальним шляхом — краще прийти оптимальним, загальна сума від того покращиться. Іншими словами, достатньо розглядати лише оптимальні рóзв'язки підзадач минулого рядка. Що якраз і є принципом Беллмана (пункт 4 розд. 1.2). По-друге, будь-який допустимий шлях обов'язково приходить до клітинки (i, j) або з клітинки $(i-1, j-1)$, або з $(i-1, j)$, або з $(i-1, j+1)$ (пропускаючи варіанти за межами таблиці); тому достатньо вибрати максимум лише з цих трьох (чи менше) величин.

Розглянемо приклад. У лівій табличці записані вхідні дані (значення клітинок, вони ж $a(\cdot,\cdot)$). У правій — побудовані для них відповіді на підзадачі (вони ж $S(\cdot,\cdot)$).

0	12	10	0	5
0	20	10	5	2
7	5	2	3	0
9	10	10	2	0

0	*12*	10	0	5
12	*32*	22	15	7
39	37	34	25	15
48	*49*	47	36	25

Якщо постановка задачі не передбачає відновлення шляху, розміри таблиці великі, а обмеження пам'яті жорсткі — в принципі можна взагалі не зберігати вхідні дані (прочитав — обробив — забув кожен окремий елемент), а з масива S пам'ятати лише два рядки (попередній і поточний), чи навіть ще менше (лише праву частину попереднього й лише ліву3 частину поточного).

Цю задачу теж неправильно намагатися розв'язати простим жадібним алгоритмом. Зокрема, «ідея» *«вибрати максимальне число 1-го рядка, далі щоразу переходити до максимального з нижньо-сусідніх»* нездоланно помилкова. Не зважаючи на те, що вона знаходить правильні відповіді для досі розглянутих прикладів. Це (майже) випадково.

Ось приклад, що відрізняється від попереднього 3-ма числами: $a(2,4) = 15$ замість 5, $a(3,5) = 16$ замість 0, $a(4,5) = 9$ замість 0. Максимуми всієї таблиці, верхнього та нижнього рядків не змінювались, але відповідь змінилася, причому шлях став зовсім інший. Не можна хапатися за ідею лише тому, що вона дає правильну відповідь на приклад з умови. Доведення бувають важливі навіть «для себе», коли їх не питають.

0	12	10	0	5
0	20	10	15	2
7	5	2	3	16
9	10	10	2	9

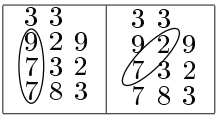
0	12	*10*	0	5
12	32	22	*25*	7
39	37	34	28	*41*
48	49	47	43	*50*

1.3.2 Найбільша серед непарних сум

Вхідні дані	Результати
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	39

Треба пройти по таблиці, дотримуючись таких самих правил. Тільки тепер питають максимальну ***серед непарних*** *сум. Непарною має бути* ***сума, а не окремі доданки.*** *Якщо абсолютно всі суми парні — вивести "*`impossible`*". Для наведеного прикладу, відповіддю є максимальна серед непарних сума* $39 = 15 + 9 + 9 + 6$.

Наївна спроба поставити серію підзадач *«Яку максимальну непарну суму $S_{odd}(i, j)$ можна набрати, пройшовши найкращий допустимий шлях до клітинки?»* ні до чого доброго не приводить. І важливо розібратися, *чому* ця спроба приречена на невдачу.



Знайдемо (не через ДП, а через здоровий глузд) спочатку (лівий рис.) шлях, яким досягається найкраща непарна сума до клітинки $(3, 1)$, потім (правий) шлях з найкращою непарною сумою до клітинки $(2, 1)$ тієї ж таблиці. Бачимо, що найкращий шлях до $(3, 1)$ проходить через $(2, 1)$, *не* будучи продовженням найкращого шляху до $(2, 1)$. Тобто, для підзадач *порушується принцип Беллмана* (пункт 4 розд. 1.2).

То що — задачу треба розв'язувати геть інакше, взагалі не динпрогом? Не факт. Поки що з'ясовано, що ДП не застосовне *до розглянутої серії підзадач*. Можливо (але поки що не гарантовано), що якась інша серія все ж дасть можливість застосувати ДП. У таких ситуаціях може виявитися (а може і не виявитися) доцільним *розширити* серію підзадач, тобто ввести додатковий параметр (або кілька).

За́раз, серію варто переформулювати так: *«Для кожної (i, j)-ої клітинки знайти максимальну непарну і максимальну парну суми, які можна набрати, прийшовши до неї.»*. Що технічно може бути реалізовано або як двовимірний масив, елементами якого є па́ри чисел, або як підзадача з трьома параметрами $S(i, j, p)$ (і, відповідно, *три*вимірний масив), у якому за двома вимірами лишається те, що й було (i — номер рядка, j — номер стовпчика), а новостворений третій вимір (параметр p) має діапазон `0..1`, де `0` означає парність суми, `1` — непарність.

Формули виходять громіздкими на вигляд, але (при розумінні вищесказаного) простими за смислом:

$$\text{Тривіальні підзадачі:} \begin{cases} \text{якщо } a(1,j) \text{ парне} \begin{cases} S(1,j,0) = a(1,j), \\ S(1,j,1) = -\infty; \end{cases} \\ \text{якщо } a(1,j) \text{ непарне} \begin{cases} S(1,j,0) = -\infty, \\ S(1,j,1) = a(1,j). \end{cases} \end{cases} \tag{6}$$

де "$-\infty$" виражає неможливість. Як і у тривіальних підзадачах (4) базової задачі, в 1-му рядку єдиний шлях складається з одного елемента. Якщо елемент єдиний і парний, ним не можна набрати непарну суму, так само як і непарним парну. Оскільки суму треба *максимізувати*, зручно виражати неможливість *мінус* нескінче́нністю: при порівнянні "$-\infty$" з реальним значенням, більшим буде реальне значення.

Основне рівняння ДП теж громіздке, але його подробиці неважко відновити самостійно, спираючись на все вищесказане. Відзначимо лише, що воно виражає $S(i,j,p)$ через $a(i,j)$ та:

у випадку парного $a(i,j)$**,** верхньо-сусідні тієї ж парності, тобто $S(i-1,j-1,p)$, $S(i-1,j,p)$, $S(i-1,j+1,p)$;

у випадку непарного $a(i,j)$**,** верхньо-сусідні протилежної парності, тобто $S(i-1,j-1,1-p)$, $S(i-1,j,1-p)$, $S(i-1,j+1,1-p)$

(скрізь пропускаючи варіанти, які виводять за межі таблиці).

Аналогічно базовій задачі, остато́чну відповідь треба сформувати, вибравши максимум серед $S(N,j,1)$ (при $1\leqslant j\leqslant M$), але треба ще врахувати ситуацію "`impossible`", коли цей максимум рівний "$-\infty$".

1.4 Розмін мінімальною кількістю банкнот

В обігу перебувають банкноти номіналами x_1, x_2, …, x_N. Як видати суму S мінімальною кількістю банкнот?

Багато хто щиро переконаний, ніби слід жадібно «щоразу брати банкноту найбільшого можливого номінала». Наприклад, якщо номінали 1, 2, 5, 10, 20, 50, 100, 200, 500 і треба видати 2018 — значить, 4 банкноти по 500 (лишається 18), банкнота 10 (лишається 8), банкнота 5 (лишається 3), банкнота 2 (лишається 1), банкнота 1 (видано).

Відзначимо без доведення, що *за умови* номіналів «1, 2, 5, 10, 20, 50, 100, 200, 500» цей підхід дає мінімальну кількість банкнот для будь-якої суми. Але це може бути не так для інших номіналів! Наприклад, якщо є три номінали 1, 17, 42 і треба видати 51, то вийде, ніби треба дати 42 і ще 9 по одиничці (всього 10 банкнот); це не оптимально, бо можна обійтися всього трьома банкнотами по 17.

Спроба заявити «ніяка нормальна країна не вводила в обіг номінали 1, 17, 42» не вирішує проблему. І не лише тому́, що олімпіадні задачі частенько мають справу з не зовсім «нормальними» вхідними даними. В реальному світі спостерігалося (правда, досить давно; схоже, потім цю помилку виправили), як банкомат одного з великих українських

банків, у якому були лише банкноти по 20 грн і 50 грн, на запит видати 60 грн відповідав «суму видати неможливо». При тому, що міг видати окремо 40 грн і зразу після цього 20 грн. Найімовірніше, причина якраз і була в тому, що старіша версія програмного забезпечення банкомата діяла за жадібним алгоритмом «щоб видати 60, почнемо з 50, лишається 10, які ніяк не видати по 20 — отже, неможливо».

Тож як розв'язати задачу правильно для будь-яких номіналів? Якщо сума S, яку треба видати, не дуже велика, можна поставити серію підзадач *«Якою мінімальною кількістю банкнот $Q(i,s)$ можна видати суму s, користуючись лише банкнотами 1-го, 2-го, …, i-го номіналів?»*. Номінали банкнот попередньо мають бути виписані в якомусь порядку (*не* обов'язково осмисленому; важливо лише, щоб було ясно, який номінал 1-й, який 2-й, і т. д.); «користуючись лише банкнотами 1-го, 2-го, …, i-го номіналів» не вимагає задіювати їх усі, важливо лише не використовувати інші.

Тривіальними будуть усі підзадачі при i=1:

$$Q(1,s) = \begin{cases} s/x_1, & \text{при } s \vdots x_1; \\ \infty, & \text{інакше} \end{cases} \tag{7}$$

(дозволені лише банкноти 1-го виду; отже, якщо сума кратна номіналу — видати можна, причому кількість дорівнює сумі, поділеній на номінал; якщо не кратна — видати цим номіналом неможливо).

Основне рівняння ДП має вигляд

$$Q(i,s) = \begin{cases} Q(i-1,s), & \text{при } s < x_i, \\ \min \begin{pmatrix} Q(i-1,s), \\ Q(i,s-x_i)+1 \end{pmatrix}, & \text{при } s \geqslant x_i. \end{cases} \tag{8}$$

Ця формула правильна, бо: завжди можна просто не користуватися новим номіналом (варіант "$Q(i-1,s)$"); при $s \geqslant x_i$ (сума не менша за номінал банкноти) можна таки використати банкноту цього номіналу. Якщо таку банкноту використали, то, крім неї однієї, треба ще суму $s-x_i$. Можливість узяти кілька банкнот поточного номіналу розглядається завдяки тому, що використовується результат підзадачі $Q(i,s-x_i)$, а не $Q(i-1,s-x_i)$, тобто для суми $s-x_i$ знов можна або використати, або не використати поточний номінал.

Приклад заповненої таблички для $S = 22$, $x_1 = 4$, $x_2 = 5$, $x_3 = 7$:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
x_1=4	0	∞	∞	∞	1	∞	∞	∞	2	∞	∞	∞	3	∞	∞	∞	4	∞	∞	∞	5	∞	∞
x_2=5	0	∞	∞	∞	1	1	∞	∞	2	2	2	∞	3	3	3	3	4	4	4	4	4	5	5
x_3=7	0	∞	∞	∞	1	1	∞	1	2	2	2	2	2	3	2	3	3	3	3	3	4	3	4

При постановці серії підзадач навіщось особливо наголосили «*якщо* сума S ... не дуже велика». Це ж завжди так — чим більші вхідні дані, тим більші витрати на виконання програми... Навіщо наголошувати? Справа в тім, що тут розмір таблички ДП пропорційний *кількості* номіналів (що типово) і *значенню* суми S (чого у цьому наборі задач ще не було). Це називається *псевдополіноміальною* асимптотичною оцінкою. З одного боку, розмір таблички всього лиш пропорційний S, з іншого — якщо задачу розв'язують (для однакових номіналів) один раз для суми 123, інший — для суми 1234567, збільшення розміру вхідних даних всього на чотири символи призводить до збільшення таблички у $\approx 10^4$ разів.

2 Література

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. «Алгоритмы: построение и анализ» (второе издание) М.–СПБ–К., Вильямс, 2005 — глава 15 «Динамическое программирование»
2. `http://informatics.mccme.ru/course/view.php?id=9`
3. `http://e-maxx.ru/algo`
4. Порублёв И. Н., Ставровский А. Б., «Алгоритмы и программы. Решение олимпиадных задач», М.–СПБ–К., Диалектика, 2007 — глава 13 «Динамическое программирование».

3 Задачі першого дня (ДП)

Цей комплект задач доступний для on-line перевірки як змагання № 64 сайту `ejudge.ckipo.edu.ua`. Там можна побачити також повні формулювання умов (у збірнику, задля економії місця, вони скорочені).

Значна частина задач цього комплекту — класичні, авторство яких встановити вже важко. Такими є, зокрема, «базові» задачі 1A, 1D, 1H.

Деякі з модифікацій цих задач розроблені укладачем комплекту І. Порубльовим — зокрема, задача 1C (де зміна вартості стрибків змінює задачу 1A). Ідея розширень серії підзадач загальновідома, але задачі 1F, 1G, де така потреба виникає природньо, розроблялися укладачем (у 1G використано також ідею Є. Поліщука).

Задача 1A «Комп'ютерна гра (платформи)»

Загальне формулювання умови цілком відповідає розд. 1.1.1.

Вхідні дані. У першому рядку записано кількість платформ n ($1 \leqslant n \leqslant 30000$). Другий рядок містить n натуральних чисел, що не перевищують 30000 — висóти, де розташовані платформи.

Вхідні дані	Результати
3 1 5 10	9
3 1 5 2	3

Результати. Виведіть єдине число — мінімальну кількість енергії, яку має витратити гравець.

Задача повністю розібрана у розд. 1.1.1.

Задача 1B «Комп'ютерна гра (платформи) з відновленням шляху»

Загальне формулювання умови та формат виведення результатів відповідають розд. 1.1.3. Формат вхідних даних відповідає попередній задачі 1A, але має інші обмеження: $2 \leqslant N \leqslant 10^5$, значення висот платформ довільного знаку, не перевищують за модулем 4000.

Вхідні дані	Результати
4 1 2 3 30	29 4 1 2 3 4
5 1 1 1 1 1	0 3 1 3 5
10 1 100 1 100 1 100 1 100 1 100	99 6 1 2 4 6 8 10

Задача повністю розібрана у розд. 1.1.1 та 1.1.3.

Задача 1C «Комп'ютерна гра (платформи) — квадратичні стрибки»

Загальне формулювання умови цілком відповідає розд. 1.1.5.

Вхідні дані. 1-й рядок містить кількість платформ N ($2 \leqslant N \leqslant 100000$), 2-й — N цілих чисел, значення яких не перевищують за модулем 4000 — їхні висóти.

Вхідні дані	Результати
4 1 2 3 30	731
5 1 1 1 1 1	0
10 1 100 1 100 1 100 1 100 1 100	9801

Результати. У єдиному рядку виведіть єдине число — мінімальну кількість енергії.

Задача повністю розібрана у розд. 1.1.1 та 1.1.5.

Задача 1D «MaxSum (базова)»

Загальне формулювання умови цілком відповідає розд. 1.3.1.

Вхідні дані. У першому рядку записані N і M — кількість рядків і кількість стовпчиків ($1 \leqslant N, M \leqslant 200$); далі у кожному з наступних N рядків записано рівно по M розділених пробілами цілих чисел (кожне не перевищує за модулем 10^6) — значення клітинок таблиці.

Вхідні дані	Результати
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	42

Результати. Вивести єдине ціле число — максимально можливу суму за маршрутами зазначеного вигляду.

Розбір задачі.

Суть розв'язку розібрана у розд. 1.3.1. А тут розглянемо дві типові для розв'язків цієї задачі помилки.

Помилка перша. У рівнянні (5) розд. 1.3.1 сказано «$S(i,j) = a(i,j) + \max\big(S(i-1,j-1),\, S(i-1,j),\, S(i-1,j+1)\big)$ (пропускаючи варіанти, які виводять за межі таблиці)». А в умові написано «при $j=1$ перший з наведених варіантів стає неможливим, а при $j=M$ — останній». І може здатися логічним реалізувати це якось так:

```
for i:=2 to N do begin
  S[i,1] := a[i,1] + max2(S[i-1, 1], S[i-1, 2]);
  for j:=2 to M-1 do
    S[i,j] := a[i,j] + max3(S[i-1,j-1], S[i-1,j], S[i-1,j+1]);
  S[i,N] := a[i,N] + max2(S[i-1, N-1], S[i-1, N]);
end;
```

(де `max2` та `max3` — власні функції, які знаходить максимальне з двох та з трьох відповідно). Начебто, окремо розібралися з 1-м елементом (нема лівого сусіда), окремо з останнім (нема правого), окремо з рештою, в яких є обидва... Але біда приходить від (дозволених!) вхідних даних з єдиним стовпчиком: для них відбуватиметься вихід за межі масиву.

Є щонайменше три способи вирішити цю проблему:

1. Написати `if`, який розгляне випадок $M = 1$ окремо, а згаданий код буде лише для $M \geqslant 2$, де він працює правильно.
2. Зробити додаткові 0-й та $(M+1)$-й стовпчики, заповнивши їх "$-\infty$" (див. розд. 1.3.2 та розбір задачі 1F). Тоді `S[i,j] := a[i,j] + max3(S[i-1,j-1], S[i-1,j], S[i-1,j+1])` можна запускати однаково в усьому проміжку j від 1 до M.
3. ```
   max := S[i-1, j];
   if (j>1) and (S[i-1, j-1] > max) then
     max := S[i-1, j-1];
   if (j<M) and (S[i-1, j+1] > max) then
     max := S[i-1, j+1];
   S[i,j] := a[i,j] + max;
   ```

Помилка друга. Максимум з трьох (a, b, c) дехто шукає так:

```
if (a>b) and (a>c) then
  max := a
else if (b>a) and (b>c) then
  max := b
else
  max := c
```

Але при $a = b = 7$, $c = 5$ такий код оголосить максимумом $c = 5$.
Це можна справити заміною усіх ">" на ">=". Але можливі й інші варіації на тему цієї помилки.

Задача 1E «MaxSum (з кількістю шляхів)»

Формулювання умови в цілому відповідає попередній задачі 1D, але, додатково до макс. суми, просять знайти *також кількість різних шляхів, на яких ця сума досягається*. Формат вхідних даних в точності повторює формат попередньої задачі 1D, але з уточненням: при
```

перевірці будуть використані лише такі вхідні дані, для яких шукана кількість шляхів з макс. сумою не перевищує 10^9.

Результати. Вивести в одному рядку два цілі числа, розділені пробілом: максимально можливу суму за маршрутами зазначеного вигляду та кількість різних маршрутів, уздовж яких вона досягається.

Вхідні дані	Результати
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	42 1
3 3 1 1 100 1 1 10 10 1 1	111 3

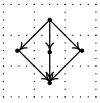
Примітка. У першому тесті, максимальне значення 42 можна набрати уздовж лише одного шляху $(15+9+9+9)$. А у другому, максимальне значення 111 можна набрати трьома способами: або `a[1][3]=100`, `a[2][2]=1`, `a[3][1]=10`, або `a[1][3]=100`, `a[2][3]=10`, `a[3][2]=1`, або `a[1][3]=100`, `a[2][3]=10`, `a[3][3]=1`.

Розбір задачі.

Може здатися, ніби достатньо, розв'язавши попередню задачу, порахувати кількість тих клітинок останнього рядка, для яких $S(N,j)$ виявилося рівним остато́чному максимуму.

Але це не так. На лівому рисунку наведено вхідні дані. На правому виділені всі шляхи з однаковою макс. сумою 7. Їх три, і вони сходяться. Зрозуміло, при більшій кількості рядків аналогічні конструкції можуть траплятися десь угорі, потім іти далі єдиним продовженням, але за рахунок різного початку вважатися різними шляхами. Отже, для знаходження кількості шляхів треба щось робити для всієї таблиці, а не само́го лише останнього рядка (чи кількох).

2	3	2
1	1	1
2	3	2



Введімо (додатково до таких са́мих, як у базовому варіанті задачі, масивів `a` та `S`) масив `K` таких са́мих розмірів, де $K(i,j)$ виражатиме *кількість шляхів до клітинки* (i,j), *які дають суму* $S(i,j)$. Аналогічно S, 1-й рядок K заповнюється тривіально:

$$K(1,j)=1 \qquad \text{для всіх } 1 \leqslant j \leqslant M, \tag{9}$$

бо шлях складається з самої лише цієї клітинки, отже рівно один, а його сума максимальна серед цього одного.

Для подальших рядків, при знаходженні `S[i,j]` (згідно того са́мого рівняння (5)) слід класти у `K[i,j]` суму тих із `K[i-1,j-1]`, `K[i-1,j]`, `K[i-1,j+1]`, які у межах таблиці *та* відповідні яким зна-

чення $S(i-1,\cdot)$ максимальні (чи то єдиний максимум, чи один з однакових). Інакше кажучи, якщо максимум S досягався лише на одній з верхньо-сусідніх клітинок, значення K переноситься з відповідної клітинки; якщо відразу кілька верхньо-сусідніх клітинок мають однакове максимальне S — слід додати значення усіх відповідних K.

Наводити код не будемо, реалізуйте все це самостійно.

В умові недаремно гарантовано, що шукана кількість шляхів з макс. сумою не перевищує 10^9. Взагалі їх може бути *значно* більше.

1	1	1	1	1	1	1	1	1	1
2	3	3	3	3	3	3	3	3	2
5	8	9	9	9	9	9	9	8	5
13	22	26	27	27	27	27	26	22	13
35	61	75	80	81	81	80	75	61	35
96	171	216	236	242	242	236	216	171	96
267	483	623	694	720	720	694	623	483	267
750	1373	1800	2037	2134	2134	2037	1800	1373	750
2123	3923	5210	5971	6305	6305	5971	5210	3923	2123
6046	11256	15104	17486	18581	18581	17486	15104	11256	6046

Наприклад, на рис. зображено кількості шляхів (без сум) виродженого випадку «$N=M=10$, значення усіх $a(i,j)$ однакові». Засобами комбінаторики можна оцінити, що при $M>1$ та всіх однакових $a(i,j)$ сумарна кількість шляхів десь у проміжку між $M\cdot 2^{N-1}$ та $M\cdot 3^{N-1}$. Так що без гарантії «до 10^9» потрібна була б «довга арифметика».

До речі, гарантія стосується лише відповіді, а `K[i,j]` доводиться шукати в т. ч. й для клітинок, які не належать шляхам з макс. сумою. Так що переповнення все одно можливі. При більшості налаштувань більшості компіляторів це не вплине на відповідь (бо ті чи́сла не вибираються). Але якщо увімкнений контроль переповнень — це може призводити до аварійних завершень, тож вмикати його не варто.

А ще така велика кількість шляхів показує, що задачу (як і більшість задач на ДП) не варто намагатися робити перебором усіх шляхів. Сонце погасне раніше, ніж він завершиться при $N\approx M\approx 200$.

Задача 1F «MaxSum (непарна сума)»

Загальне формулювання умови, формат виведення результату, а також приклад та коментар до нього відповідають розд. 1.3.2. Формат вхідних даних, включно з обмеженнями, відповідає задачі 1D.

Розбір задачі.

Суть розв'язку розібрана у розд. 1.3.2. А тут розглянемо способи, якими можна реалізовувати "$-\infty$".

Один з них (у багатьох старих текстах, єдиний) — взяти велике за модулем від'ємне число. Яке — треба вибирати з таких міркувань. Якщо взяти його сильно малим за модулем, така "$-\infty$" плюс багато великих додатних чисел можуть дати суму, більшу за суму потрібних малих «справжніх» чисел, тобто "$-\infty$" перестає виражати неможливість. Якщо сильно великим за модулем, така "$-\infty$" плюс відразу багато від'ємних чисел можуть у результаті переповнення стати більшими (замість меншими) за всі «справжні» чи́сла. Акуратно треба і з перевіркою «$\max_{1\leqslant j\leqslant M} S(N,j,1) = -\infty$» — на відміну від математики, така "$-\infty$" плюс ненульове число вже «не зовсім дорівнює» "$-\infty$". Враховуючи все це, діапазон 32-бітового `int` та обмеження «до 200 рядків, значення $a(i,j)$ від -10^6 до 10^6», можна, наприклад, узяти в якості "$-\infty$" число -10^9 і проводити перевірку "$S_{\max} = -\infty$" як "`Smax < -5e8`". Але не варто брати за "$-\infty$" ні -10^8, ні $-2\cdot10^9$.

Ще один спосіб (*якщо* працює, то простий і зручний) — типи з плаваючою точкою (наприклад, `double`) можуть мати вбудовану підтримку робо́ти з нескінченностями. Можна або знайти, як це позначається у потрібній версії потрібної мови програмування, або написати щось у стилі `const double MINUS_INFTY = -exp(1e6)` (щоб гарантовано виходило за межі типу) й потім скрізь використовувати `MINUS_INFTY`. Але він поганий тим, що не досить кросплатформенний, може працювати з одними компіляторами (тим паче, мовами) і не працювати з іншими. (Тут слід розуміти, що у практичному програмуванні важливо, чи працюватиме програма на комп'ютерах замовників, а на олімпіаді важливо, чи працюватиме програма на сервері перевірки. А чи працює програма у самого програміста — мало кого цікавить...)

Типи з плаваючою точкою можуть мати також проблему похибок. Наприклад, $(10^{18}+1) - 10^{18}$ для деяких поєднань типу даних, компілятора, його налаштувань, особливостей OS та «заліза» дає 1, але для багатьох 0. У більшості ситуацій, розрядна сітка не гірша, ніж у типі `double` за стандартом IEEE 754, де похибки цілих чисел можуть з'являтися лише для значень, більших (за модулем) $2^{53} \approx 8\cdot10^{15}$. Але якщо економити по-дурному, як-то вибираючи з усіх наявних у C/C++ типів з плаваючою точкою найвужчий тип `float`, похибки можливі вже при значеннях, ледь більших $2^{23} \approx 8\cdot10^6$, тобто також і в цій задачі.

Задача 1G «MaxSum (усі стовпчики)»

Вхідні дані	Результати
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	28

Опис таблиці та дозволених шляхів відповідають розд. 1.3.1 та задачі 1D, але тепер треба знайти максимальну суму лише серед тих шляхів, які i задовольняють вимоги з тієї задачі, *і проходять хоча б по одному разу через кожен зі стовпчиків*. Формат вхідних даних в цілому відповідає задачі 1D, але тепер можливі більші розміри: $1 \leqslant N \leqslant 1024$, $1 \leqslant M \leqslant N$. Формат виведення результатів цілком відповідає задачі 1D; оскільки гарантовано, що $M \leqslant N$, відповідь існує завжди.

Примітки. Відповідь дорівнює $28 = 15 + 5 + 2 + 6$, бо всі шляхи з більшою сумою проходять не через усі стовпчики.

Розбір задачі.

Зрозуміло, що слід розширити серію підзадач базової задачі. Виявляється, це можна зробити, збільшивши кількість підзадач, обсяг пам'яті та час роботи всього лиш учетверо. Замість серії (i, j)-підзадач, поставимо серію (i, j, wl, wr)-підзадач, де цілочисельні i та j так само номери рядка та стовпчика, загальне формулювання *«яку максимальну суму $S(i, j, wl, wr)$ можна назбирати, дійшовши від верхнього рядка до ...»* теж аналогічне, а смисл *булевих* wl та wr — *«чи був хоча б раз відвіданий крайній (лівий для wl, правий для wr) стовпчик?»*. Якщо побували геть в усіх, то в тому числі й у першому та останньому; а завдяки тому, що на кожному кроці можна переходити лише або в той самий стовпчик, або в один із сусідніх, якщо побували і в першому, і в останньому, то й у всіх проміжних (отже, геть у всіх). Так що остато́чною відповіддю задачі буде $\max_{1 \leqslant j \leqslant M} S(N, j, \texttt{true}, \texttt{true})$.

Значна частина підзадач неможливі — зокрема, але не тільки, $(1, j, \texttt{false}, wr)$ при довільних j та wr (неможливо бути в 1-му стовпчику прямо зараз і при цьому не побувати в ньому). Відповідями всіх неможливих підзадач вважатимемо "$-\infty$" (див. також розд. 1.3.2 та розбір попередньої задачі 1F).

Параметри wl та wr створюють більше, ніж у «базовій» задачі, відмінностей між випадком $M = 1$ та $M \geqslant 2$. Тому випадок $M = 1$ пропонується розглянути окремо, а в усіх подальших міркуваннях вважаємо $M \geqslant 2$. З урахуванням цього, тривіальні підзадачі та рівняння ДП можна сформулювати, наприклад, так:

1. $S(1,1,\texttt{true},\texttt{false}) = a(1,1)$, бо єдиний шлях складається з єдиної цієї клітинки, причому 1-го стовпчика, отже в ньому побували (а в останньому ні, бо $M \geqslant 2$);
2. $S(1,M,\texttt{false},\texttt{true}) = a(1,M)$, аналогічно щодо останнього;
3. $S(1,j,\texttt{false},\texttt{false}) = a(1,j)$ (для $2 \leqslant j \leqslant M-1$), бо єдиний шлях складається з єдиної цієї клітинки, причому не 1-го й не останнього стовпчика;
4. $S(i,1,\texttt{false},wr) = -\infty$ (для всіх $1 \leqslant i \leqslant N$, обох значень wr), бо неможливо бути в 1-му стовпчику зараз і не побувати в ньому;
5. $S(i,M,wl,\texttt{false}) = -\infty$ (для всіх $1 \leqslant i \leqslant N$, обох значень wl), аналогічно щодо останнього;
6. $S(1,j,\texttt{true},\texttt{true}) = -\infty$ (для всіх $1 \leqslant j \leqslant M$), бо (при $M \geqslant 2$) неможливо за одну клітинку побувати зразу на обох краях;
7. $S(1,j,\texttt{true},\texttt{false}) = S(1,j,\texttt{false},\texttt{true}) = -\infty$ (для $2 \leqslant j \leqslant M-1$), бо коли єдина клітинка не з 1-го й не з останнього стовпчика, то неможливо побувати не лише відразу в обох крайніх стовпчиках, а й у якомусь одному з них;
8. $S(i,1,\texttt{true},wr) = a(i,1) + \max\bigl(S(i-1,1,\texttt{true},wr),\ S(i-1,2,\texttt{true},wr),\ S(i-1,2,\texttt{false},wr)\bigr)$ (для всіх $2 \leqslant i \leqslant N$, обох значень wr), бо: раз прямо зараз у 1-му стовпчику, то, хоч там вже бували раніше, хоч ні, в підсумку побували; варіант $S(i-1,\ 1,\ \texttt{false},wr)$ все одно неможливий[1]; wr слід лишити, як було, бо від такого переходу відвідування останнього стовпчика не щезає (якщо було) і не з'являється (якщо ні);
9. $S(i,M,wl,\texttt{true}) = a(i,M) + \max\bigl(S(i-1,M,wl,\texttt{true}),\ S(i-1,M-1,\ wl,\texttt{true}),\ S(i-1,M-1,wl,\texttt{false})\bigr)$ (для всіх $2 \leqslant i \leqslant N$, обох значень wl), з аналогічних (симетричних) міркувань;
10. $S(i,j,wl,wr) = a(i,j) + \max\bigl(S(i-1,j-1,wl,wr),\ S(i-1,\ j,\ wl,\ wr),\ S(i-1,j+1,wl,wr)\bigr)$ (для всіх $2 \leqslant i \leqslant N$, всіх $2 \leqslant j \leqslant M-1$, усіх поєднань wl та wr) — основний, не пов'язаний з «крайовими ефектами», випадок, коли всі переходи з верхніх-сусідніх можливі й не змінюють wl чи wr.

Цей перелік, звісно, громіздкий. Але якщо його не зазубрювати, а відтворювати по смислу, розуміючи, то не так усе і складно. Варто також переконатися, що випадок $M = 2$, хоч і робить неможливою

[1]Хоча, якщо так зручніше, його можна теж розглянути, там все одно "$-\infty$".

ситуацію $2 \leqslant j \leqslant M-1$, цілком описується частиною цього ж переліку, не потребу́ючи ніяких особливих формул.

Можуть бути правильними й деякі інші варіації на тему цього переліку випадків. Наприклад, пункт 6 при бажанні можна розширити на всі $1 \leqslant i \leqslant M-1$, бо за один рядок відвідується щонайбільше один новий стовпчик, тож неможливо відвідати всі M стовпчиків, не пройшовши хоча б M рядків; таким чином, пункт 6 міг би забрати на себе частину того, що зараз віднесено до пункту 10. І є ще кілька аналогічних ситуацій.

Чи можна лишити з цих пунктів тільки перші три й останні три, а замість решти сказати «перед застосуванням усіх цих пунктів, ініціалізуємо весь масив значеннями "$-\infty$"»? І так, і ні. Залежить від мети. Якщо хотіти твердо переконатися, що в масиві геть усі елементи заповнені правильно, то не варто. Можна навіть, навпаки, спочатку ініціалізувати все значеннями "$+\infty$", потім застосувати цей перелік, потім подивитися, чи точно ніде нема "$+\infty$". Якщо ж мати протилежну мету «якщо десь є помилки, будемо намагатися приховати й компенсувати їх», то це дуже навіть добра ідея. Що кінець кінцем краще — важко сказати. З одного боку, такі «компенсації» частенько дають програмі можливість запрацювати, навіть коли програміст не повністю уявляє всі деталі. З іншого, якщо виявляється, що програма працює неправильно, такі «компенсації» сильно ускладнюють пошук помилок.

Задача 1H «Банкомат–1»

Загальне формулювання умови цілком відповідає розд. 1.4.

Вхідні дані	Результати
7 1 2 5 10 20 50 100 72	3
2 20 50 60	3

Вхідні дані. Перший рядок містить натуральне число N, що не перевищує 50 — кількість номіналів банкнот у обігу. Другий рядок вхідних даних містить N різних натуральних чисел x_1, x_2, …, x_N, що не перевищують 10^5 — номінали банкнот. Третій рядок містить натуральне число S, що не перевищує 10^5 — суму, яку необхідно видати.

Результати. Програма повинна вивести єдине число — знайдену мінімальну кількість банкнот. Якщо видати вказану суму вка-

заними банкнотами неможливо, програма повинна вивести рядок "`No solution`" (без лапок, перша літера велика, решта маленькі).

Примітка. У 1-му тесті, 72 можна видати 3-ма банкнотами 50, 20 і 2. У 2-му, 60 можна видати 3-ма банкнотами як 20, 20 і 20.

Задачу можна розв'язати, реалізувавши алгоритм з розд. 1.4. Можна відштовхуватись і від його модифікації з наступної задачі 1I.

Задача 1I «Банкомат–2 (з відновленням)»

Загальне формулювання умови відповідає розд. 1.4 та попередній задачі 1H. Формат вхідних даних в цілому відповідає попередній задачі 1H, але обмеження більші: кількість номіналів $N \leqslant 100$; сума S, яку треба видати, та номінали банкнот $x_1, x_2, \ldots, x_N$ не перевищують 10^6.

Результати. Програма повинна знайти подання числа S у вигляді суми доданків з множини x_i, що містить мінімальне число доданків, і вивести це подання у вигляді послідовності чисел, розділених пробілами.

Вхідні дані	Результати
7 1 2 5 10 20 50 100 72	50 20 2
2 20 50 60	20 20 20

Якщо таких подань існує декілька, то програма повинна вивести будь-яке (одне) з них. Якщо такого подання не існує, то програма повинна вивести рядок "`No solution`" (без лапок, перша літера велика, решта маленькі).

Розбір задачі.

Виявляється, у розд. 1.4 розглянуто не найкращий алгоритм! Конкретніше, він неекономно використовує пам'ять. При бажанні, можна увіпхнути в одне число те, що в алгоритмі з розд. 1.4 займало цілий стовпчик таблиці. Тобто, можна поставити і розв'язати серію підзадач *«Якою мінімальною кількістю банкнот $Q(s)$ можна видати суму s?»*. Алгоритм лишається псевдополіноміальним, але об'єм пам'яті зменшується у N разів. Рівняння ДП —

$$Q(s) = \min_{i\,:\,x_i \leqslant s} \{Q(s - x_i) + 1\}, \tag{10}$$

тобто $Q(s)$ треба шукати циклом по i, беручи до уваги лише номінали $x_i \leqslant s$, щоб $s - x_i$ не ставало від'ємним. (Ситуація, коли для кожної

підзадачі треба запускати вкладений цикл, нормальна для ДП; ненормально, що вона не виникала ні в одній з попередніх задач.) (Єдиною) тривіальною підзадачею можна вважати $Q(0)=0$. Оскільки (в разі відсутності номіналу 1) можлива ситуація, коли якісь суми неможливо видати вказаними номіналами, для всіх $s>0$ зручно ініціалізувати $Q(s)$ як "$+\infty$", а потім намагатися зменшити згідно (10).

Для зручності зворотнього ходу варто записувати у допоміжний масив, наприклад, значення того номінала, при якому вдалося отримати оптимальну відповідь відповідної підзадачі. На те, щоб тримати такий допоміжний одновимірний масив, пам'яті цілком достатньо: два масиви по мільйону 4-байтових `int`-ів — менше 8 Мб. Пам'яті не вистачало, щоб розмістити N ($N\leqslant 100$) рядків.

Чому в розд. 1.4 не був розглянутий відразу цей варіант алгоритму? Відповідь на це питання — у наступній задачі.

Задача 1J «Банкомат–3 (з обмеженнями кількостей, з відновленням)»

Правда, вам набридли абсолютно неприродні олімпіадні задачі? Ну навіщо казати: «Якби банкомат заправили банкнотами по 10, 50, 60 і 100, то суму 120 варто було б видавати не як $100+10+10$, а як $60+60\ldots$»? Ніхто ж не стане вводити в обіг банкноти номіналом 60... Тому зараз пропонуємо розв'язати абсолютно практичну задачу.

В обігу перебувають банкноти номіналами 1, 2, 5, 10, 20, 50, 100, 200 та 500 гривень. Причому, банкноти номіналами 1 грн та 2 грн в банкомати ніколи не кладуть. Так що в банкоматі є N_5 штук банкнот по 5 грн, N_{10} штук банкнот по 10 грн, N_{20} штук банкнот по 20 грн, N_{50} штук банкнот по 50 грн, N_{100} штук банкнот по 100 грн, N_{200} штук банкнот по 200 грн та N_{500} штук банкнот по 500 грн.

Для банкомата діють адміністративне обмеження «видавати не більш як 2000 грн за один раз» та технічне обмеження «видавати не більш як 40 банкнот за один раз». В останньому обмеженні мова йде про сумарну кількість банкнот (можливо, різних номіналів).

Напишіть програму, яка визначатиме, як видати потрібну суму мінімальною кількістю банкнот (з урахуванням указаних обмежень).

Вхідні дані. Програма читає спочатку кількості банкнот N_5, N_{10}, N_{20}, N_{50}, N_{100}, N_{200} та N_{500}, потім суму S, яку треба видати. Усі чи́сла вхідних даних цілі, у межах від 0 включно до 5000 включно.

Результати. Програма повинна вивести сім чисел — скільки треба видати банкнот 5 грн, 10 грн, 20 грн, 50 грн, 100 грн, 200 грн та 500 грн. Ці сім чисел треба вивести в один рядок, розділяючи пропусками. Сума цих чисел (загальна кількість банкнот до видачі) повинна бути мінімальною.

Якщо видати суму, дотримуючись обмежень, неможливо, програма повинна замість відповіді вивести (єдине) число -1.

Вхідні дані	Результати
0 100 1 100 0 0 0 190	0 2 1 3 0 0 0
5000 2000 5000 2000 5000 2000 500 17	-1

Розбір задачі.

У розд. 1.4 розглянуто, як, при наявності банкнот по 50 грн і 20 грн та відсутності банкнот по 10 грн, жадібний алгоритм може дати неправильну відповідь (зокрема, на запит видати 60 грн). Так що задачу варто розв'язувати аналогічно попереднім, користуючись ДП.

Значить, заради однотипності з попередніми задачами, можна вважати, що у нас все-таки заданий масив номіналів x_1, x_2, …, x_N. Вони не читаються зі вхідних даних, а задаються константами $N{=}7$, $x_1{=}5$, $x_2{=}10$, $x_3{=}20$, $x_4{=}50$, $x_5{=}100$, $x_6{=}200$, $x_7{=}500$, але на алгоритм це майже не впливає. Аналогічно, для зручності, перейменуємо кількості банкнот, що є в наявності N_5, N_{10}, …, N_{500} у, наприклад, $m_1{=}N_5$, $m_2{=}N_{10}$, $m_3{=}N_{20}$, $m_4{=}N_{50}$, $m_5{=}N_{100}$, $m_6{=}N_{200}$, $m_7{=}N_{500}$.

Задача відрізняється від обох попередніх обмеженнями кількостей банкнот конкретних номіналів. Природньо, що задача 1Н (серія підзадач якої є розширенням серії підзадач задачі 1І) набагато легше піддається модифікації по введенню таких додаткових обмежень — са́ме тому, що містить більш явні залежності підзадач від номіналу.

Серію підзадач поставимо в точності як у задачі 1Н: *«Якою мінімальною кількістю банкнот $Q(i,s)$ можна видати суму s, користуючись лише банкнотами 1-го, 2-го, …, i-го номіналів?»*.

$$\text{Тривіальні:}\quad Q(1,s)=\begin{cases} s/x_1, & \text{при } (s\vdots x_1)\text{ and }((s/x_1)\leqslant m_1);\\ \infty, & \text{інакше}\end{cases} \tag{11}$$

тобто тут враховується як подільність суми s на номінал x_1, так і обмеження кількості банкнот.

У поясненні до (8) (розд. 1.4) є фрагмент «… використовується … $Q(i,s-x_i)$, а не $Q(i-1,s-x_i)$, тобто для суми $s-x_i$ знов можна або

використати, або не використати поточний номінал». Са́ме це і треба змінити — наприклад, як

$$Q(i,s) = \min_{\begin{cases} 0 \leqslant k \leqslant m_i \\ k \cdot x_i \leqslant s \end{cases}} \big(Q(i-1, s - k \cdot x_i) + k\big), \tag{12}$$

де $Q(i-1, s-k\cdot x_i)+k$ означає: взяти k банкнот поточного i-го номіналу, решту суми $s-k\cdot x_i$ набрати банкнотами попередніх номіналів. Права частина цього рівняння ДП містить лише $Q(i-1,\dots)$, тобто гарантований перехід до попередніх номіналів. Разом з тим, за рахунок оцього k, використати поточний номінал кілька разів все-таки можна, причому на кількість цих використань накладено в т. ч. й обмеження $k \leqslant m_i$. Варіант «не бра́ти цей номінал» теж розглянутий (при $k{=}0$).

Само собою, треба врахувати додаткові обмеження — «не більше 2000 грн» (до запуску ДП); «не більше 40 банкнот» (після).

Чи можна так само виражати $Q(i,s)$ через $Q(i-1,\ s-k\cdot x_i)$ (при $0 \leqslant k\cdot x_i \leqslant s$), а не через $Q(i,\ s-x_i)$ і в задачі 1H (першій у серії)? З точки зору теоретичної правильності — можна, відповіді будуть такі самі. Але час робо́ти буде набагато гіршим, бо запускати зайвий цикл перебору можливих k, особливо при малих x_i — збільшувати найгірший можливий час роботи з $O(S\cdot N)$ до $O(S^2\cdot N)$, що допустимо для обмежень поточної задачі 1J, але не задачі 1H. Так що звідси доцільніше винести інше: якщо програма працює надто довго, рівняння ДП нагадує (12) і нема обмежень на кількості використань однакових значень — варто спробувати перетворити рівняння до вигляду, більш схожого на (8).

4 Задачі другого дня (коли ДП недоречне)

Цей комплект задач доступний для on-line перевірки як змагання № 65 сайту `ejudge.ckipo.edu.ua`. Для цього комплекту, збірник теж містить скорочені версії умов, але таких скорочень дещо менше.

Частина задач комплекту (2A, 2B, 2C) спеціально розроблені так, щоб на перший погляд сильно нагадувати деякі задачі комплекту попереднього дня і тим провокувати застосування ДП. Решта задач теж мають *частину* властивостей, потрібних динпрогу, але кінець кінцем

ДП виявляється або взагалі незастосовним, або недоречним. Авторами задачі 2E (розподіл станцій по зонам) є В. Челноков та Д. Поліщук.

Задача 2A «MaxSum (стрибки у будь-який стовпчик)»

Відрізняється від «базової версії» (задачі 1D попереднього дня) лише тим, що на кожному кроці можна переходити в *будь-яку* клітинку наступного рядка (не лише «нижньо-сусідню»).

Формат вхідних даних, включно з обмеженнями, відповідає задачі 1D. Формат результатів теж, тільки максимум слід шукати серед інших шляхів.

Вхідні дані	Результати
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	42
3 3 1 1 100 1 1 10 100 1 1	210

Примітки. У першому тесті, хоч і можна переходити в *будь-яку* клітинку наступного рядка, це не дає можливості збільшити цільове значення у порівнянні з «базовим» варіантом задачі. У другóму ж тесті цю можливість варто використати й отримати 210 як $100 + 10 + 100$.

Розбір задачі.

Досить рідкісний випадок, коли застосувати ДП *можна, але не варто*. Раз дозволені стрибки з будь-якого стовпчика у будь-який стовпчик, ніщо не заважає просто вибирати з кожного рядка максимум, і остаточна відповідь задачі буде просто сумою таких максимумів. Спроби ж застосовувати ДП призводили б лише до зайвих витрат пам'яті та зайвих обчислень (можливо, навіть асимптотично зайвих).

Задача 2B «Комп'ютерна гра (платформи) — необмежена довжина стрибку»

У старих іграх можна зіткнутися з такою ситуацією. Герой стрибає по платформах, які висять у повітрі. Він повинен перебратися від одного краю екрана до іншого. Гравець може стрибнути з будь-якої платформи i на будь-яку платформу k, затративши при цьому $(i-k)^2 \cdot (y_i - y_k)^2$ енергії, де y_i та y_k — висóти, на яких розташовані ці платформи.

Відомі висóти платформ у порядку від лівого краю до правого. Знайдіть мінімальну кількість енергії, достатню, щоб дістатися з 1-ої платформи до N-ої (останньої).

Вхідні дані. Перший рядок містить кількість платформ ($1 \leqslant N \leqslant 4000$), дру́гий — N цілих чисел, значення яких не перевищують за модулем 200000 — висóти платформ.

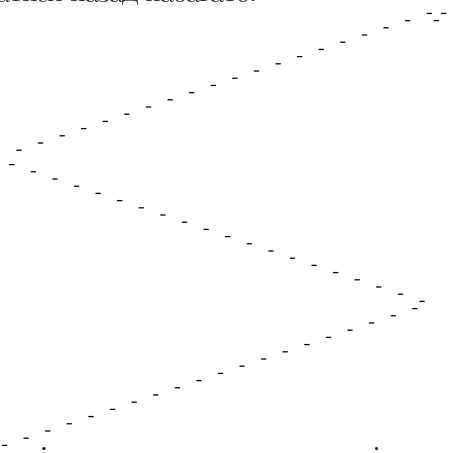
Вхідні дані	Результати
4 1 2 3 30	731

Результати. Виведіть єдине число — мінімальну кількість енергії, яку має витратити гравець на подолання платформ.

Примітка. $731 = 1^2 \cdot 1^2 + 1^2 \cdot 1^2 + 1^2 \cdot 27^2$.

Розбір задачі.

Як і у розд. 1.1.5, тут буває вигідно стрибати назад. Але, на відміну від розд. 1.1.5, іноді вигідно вертатися назад набагато.



Наприклад, щось у стилі зображеного на рисунку, де вигідно дійти майже до правого краю нижнім ланцюгом платформ, потім назад майже до лівого середнім, і знов, тепер уже остаточно, до правого верхнім ланцюгом. Якщо ж розглядати більшу кількість платформ (не десятки, а тисячі), то можна побудувати й такі приклади, де вигідно змінювати напрям руху десятки разів і при цьому щоразу зміщуватися на сотні платформ. Так що нічого схожого на компенсацію циклічності, описану в розд. 1.1.5, не вийде.

Зате в тому ж розд. 1.1.5 є рекомендація, що в таких випадках варто проаналізувати, чи застосовний алгоритм Дейкстри. І тут це так і є. Причому, оскільки тепер граф щільний (навіть повний: з кожної платформи можна стрибати на будь-яку, питання лише в затратах), доречнішою є проста версія алгоритму Дейкстри, що працює за $O(N^2)$. Легко бачити, що при $N \leqslant 4000$ це цілком прийнятно.

Детальніше про алгоритм Дейкстри прочитайте деінде.

Задача 2C «MaxSum (щаслива)»

Загальний опис таблиці та дозволених шляхів, включно з поняттям «нижньо-сусідньої клітинки», такий самий, як у задачах 1D, 1E, 1F та 1G попереднього дня.

Напишіть програму, яка знаходитиме максимально можливу *щасливу* суму значень пройдених клітинок серед усіх допустимих шляхів. Як широко відомо у вузьких колах, щасливими є ті й тільки ті чи́сла, десятковий запис яких містить лише цифри 4 та/або 7 (можна обидві, можна лише якусь одну; ніяких інших цифр використовувати не можна). Зверніть увагу, що щасливою повинна бути са́ме сума, а обмежень щодо окремих доданків нема.

Вхідні дані. У першому рядку записані N та M — кількість рядків і кількість стовпчиків ($1 \leqslant N, M \leqslant 12$); далі у кожному з наступних N рядків записано по M розділених пробілами невід'ємних цілих чисел, кожне не більш ніж з 12 десяткових цифр — значення клітинок.

Вхідні дані	Результати
3 4 3 0 10 10 5 0 7 4 4 10 5 4	7

Результати. Вивести або єдине ціле число (знайдену максимальну серед щасливих сум за маршрутами зазначеного вигляду), або рядок «impossible» (без лапок, маленькими латинськими буквами). Рядок «impossible» має виводитися тільки у разі, коли жоден з допустимих маршрутів не має щасливої суми.

Примітки. Взагалі максимально можливою сумою є $27 = 10 + 7 + 10$, але і саме́ число 27 не є щасливим, і в діапазоні від 8 до 26 нема жодного щасливого числа́. Тому відповіддю буде максимальна серед щасливих сума $7 = 3 + 0 + 4$, яка досягається уздовж маршруту $\texttt{a[1][1]} \to \texttt{a[2][2]} \to \texttt{a[3][1]}$.

Наскільки відомо автору задачі, автором са́ме такого трактування «щасливого числа» є Василь Білецький, випускник Львівського національного університету імені Івана Франка, котрий був капітаном першої з українських команд, що вибороли золоту медаль на фіналі першості світу ACM ICPC, і тривалий час входив у десятку найсильніших спортивних програмістів світу за рейтингом TopCoder.

Розбір задачі.

Для такої постановки задачі неясно, як можна було б розширювати серію, щоб добитися відрізнення щасливих сум від не щасливих. Але, якщо перечитати обмеження ($1 \leqslant N, M \leqslant 12$) та співвіднести його

з оцінками кількості шляхів з задачі 1E попереднього дня, стає видно, що *тут можна перебрати всі шляхи* (їх менше $12 \cdot 3^{11} \approx 2$ млн, що для комп'ютера відносно небагато), для кожного порахувати суму, й вибрати максимальну звичайним («шкільним») алгоритмом вибору максимуму, з модифікованою умовою «замінювати поточний максимум, лише якщо нове число одночасно і більше, і щасливе».

Формат збірника не дозволяє пояснити все це справді детально (тим паче, що це *не* ДП), тому наведемо код (C++, масив нумерується з 0) самої «серцевини» перебору без пояснень, а деталі додумайте, використовуючи інші джере́ла. Цей спосіб перегляду всіх можливих шляхів називають *рекурсивним перебором*. Корисно почитати також про *пошук з поверненнями* та *бектрекінг* (*back-tracking*), але ці назви вже стосуються не так само́го перебору, як засобів його скорочення.

```
void rec_bf(int i, int j, long long sum_before)
{
  long long sum_with_current = sum_before + data[i][j];
  // до набраного раніше додали значення поточної клітинки
  if(i==N-1) { // останній рядок
    if(sum_with_current > ans && is_lucky(sum_with_current))
      ans = sum_with_current;
  } else {
    if(j>0) { // пробуємо піти вниз-ліворуч; для глибшого
      rec_bf(i+1, j-1, sum_with_current); //рівня вкладеності,
    } // поточна клітинка вже входитиме у <<набране раніше>>
    rec_bf(i+1, j, sum_with_current); // пробуємо просто вниз
    if(j+1<M) { // пробуємо вниз-праворуч
      rec_bf(i+1, j+1, sum_with_current);
    }
  }
}
```

Викликáти цю функцію треба, наприклад, так: “`for(int j=0; j<M; j++) rec_bf(0, j, 0LL);`”.

Задача 2D «Єгипетські дроби»

Математики стародавнього Єгипту не знали дробів у сучасному розумінні, але вміли подавати нецілі числа як суму дробів вигляду $1/k$, причому всі k в цій сумі мали бути різними. Наприклад, сучасне поняття $2/5$ виражалося як «одна третя та одна п'ятнадцята» (справді, $1/3+1/15=\frac{5}{15}+\frac{1}{15}=\frac{6}{15}=2/5$). Математики Нового часу довели, що будь-який правильний звичайний дріб можна подати у єгипетському поданні (як суму дробів вигляду $1/k$), причому це подання не єдине.

Напишіть програму, яка перетворюватиме правильний звичайний дріб до єгипетського подання із мінімальною кількістю доданків.

Вхідні дані. Єдиний рядок містить два натуральні числа n та m ($1\leqslant n<m\leqslant 1000$) — чисельник та знаменник правильного звичайного дробу.

Вхідні дані	Результати
2 5	3 15
732 733	2 5 8 9 16 65970 105552

Результати. Виведіть в один рядок, розділяючи пропусками, сукупність знаменників у єгипетському поданні цього дробу. Всі ці знаменники повинні бути різними, а їхня кількість — мінімальною.

Примітки. У цій задачі, для деяких вхідних даних, можуть бути різні правильні відповіді (з однаковою мінімальною кількістю доданків). Ваша програма повинна знайти будь-яку одну.

Як видно з прикладів, значення знаменників відповіді можуть бути значно більшими за значення чисел у вхідних даних. Автор задачі гарантує, що для всіх дозволених умовою вхідних даних існують такі правильні відповіді, що при їх знаходженні «довга» арифметика не потрібна (тобто, існують правильні відповіді, для яких і кінцеві значення знаменників, і всі проміжні значення правильно організованих обчислень не виходять за межі стандартних цілих типів).

При перевірці вимагатиметься, щоб значення кожного окремо зі знайдених знаменників не перевищувало $10^{18}-1$; добуток знайдених знаменників, якщо учаснику так зручно, може й перевищувати.

Розбір задачі.

Дру́гий приклад $\frac{732}{733}=\frac{1}{2}+\frac{1}{5}+\frac{1}{8}+\frac{1}{9}+\frac{1}{16}+\frac{1}{65970}+\frac{1}{105552}$, при всій громіздкості, доносить важливу інформацію, що мінімальна кількість дробів може бути відносно великою. Прості приклади, які легко перевіряти, можуть навести на хибну думку, ніби ця кількість завжди 1–3, і без цього прикладу зрозуміти хибність такої гіпотези значно важче.

Спробуємо міркувати так. Скоротимо заданий дріб $\frac{n}{m}$, тобто знайдемо НСД(n,m) і поділимо на нього n та m, поклавши результати ділень у ті самі `n` та `m`. (Тут і далі, «шрифт друкарської машинки» ("`n`", а не "n") вказує, що йдеться про змінні (у смислі програмування).) Потім перевіримо, чи `n`$=1$. Якщо так, то доданок єдиний, відповідь готова. Інакше, слід подати $\frac{\texttt{n}}{\texttt{m}}$ як суму деякого $\frac{1}{\texttt{a}}$ і «решти» (котра дорівнює $\frac{\texttt{n}}{\texttt{m}} - \frac{1}{\texttt{a}} = \frac{\texttt{n·a−m}}{\texttt{m·a}}$, тобто знаходимо чисельник `n*a-m`, знаменник `m*a`, скорочуємо). Цю «решту» теж слід перетворити в єгипетське подання, тож покладемо нові чисельник і знаменник у ті самі `n` та `m`. І повторюватимемо все це, доки не знайдемо розкладення.

Тільки до цього є чимало питань: *«чи гарантовано, що такий процес завершиться?»* (при тому, що треба ще й мінімальність!); *«як уникати повторного використання тих самих знаменників?»*; *«чи можна щоразу легко визначати "правильне" значення* `a`*, чи треба перебирати різних претендентів і вибирати найкращий варіант?»*; тощо.

Щодо останнього — здається природним (і прості приклади це «підтверджують»), ніби можна бра́ти за «правильне» `a` `(m div n) +1` (найменший знаменник $\Leftrightarrow$ найбільший дріб $\frac{1}{\texttt{a}}$ серед строго менших $\frac{\texttt{n}}{\texttt{m}}$). Але, наприклад, $\frac{9}{20}$ цей жадібний підхід подасть як $\frac{1}{3}+\frac{1}{9}+\frac{1}{180}$, хоча можна як $\frac{1}{4}+\frac{1}{5}$, тобто 3 доданки проти 2-х, що не мінімально.

(Де бра́ти такі приклади самостійно під час туру? Універсальної відповіді нема й не може бути, а часткові поради залежать від знань та вмінь учасника. Комусь легше шукати олівцем на папері, комусь — реалізовувати різні не доведені ідеї і порівнювати результати, тощо.)

Раз не працює жадібний підхід, природньо спробувати ДП, щоб перебрати різні `a` й вибрати найкращий варіант: виділити підзадачі *«Якою мінімальною кількістю доданків $Q(i,j)$ можна розкласти в єгипетське подання дріб $\frac{i}{j}$?»*, зменшення кількості дробів-доданків у розкладенні дробу-«решти» призводить до зменшення кількості дробів-доданків того більшого дробу, з якого виділили «решту»...

Але є щонайменше три проблеми. (Найгірше, що істотні, заважають застосувати ДП просто і стандартно, але не нездола́нні, можна придумувати засоби, щоб *пробувати* їх нівелювати.) Одна — як уникати повторів знаменників. Це можна робити розширенням серії підзадач, включивши до параметрів якусь інформацію про заборонені знаменники. Але яку? Ще одна — менші дроби часто мають більші знаменники, тож обмеження $n < m \leqslant 1000$ нічогісінько не каже про макси-

мальний знаменник проміжних підзадач.[2] Третя проблема — складни́м виявляється (зазвичай елементарне) питання *«в якому діапазоні перебирати* a*?»* при виборі мінімума у рівнянні ДП. Вже пояснено, чому (для дробу $\frac{i}{j}$, при $i>1$) нижня межа $(j \operatorname{div} i)+1$. Але яка верхня?

Спробуємо не забути, але відкласти ці питання, й зайти з іншого боку. Як варто було б писати програму, якби зосередилися на правильності пошуку тих оптимальних розкладень, що містять лише або 1 доданок, або 2? (Навіщо, якщо це не дасть повного розв'язку? Частково, щоб можна було здати хоч такий розв'язок, щоб набрати бали хоча б за такі тести. Якщо це поєднати з «якщо так не знайшли, то повторювати в циклі жадібний вибір "`a := (m div n) +1`"» — є надія на якісь бали навіть при поблоковій системі оцінювання, а при потестовій навіть на відносно високі. Але це другорядна мета. Важливіше, що в таких спрощених випадках часом вдається побачити особливі властивості задачі, й подумати, як їх використати у складніших ситуаціях.)

Скоротимо дріб; якщо $\mathtt{n}=1$, виведемо відповідь m, інакше запустимо перебір, що має знайти такі a, b, що $\frac{1}{a}+\frac{1}{b}=\frac{n}{m}$. Писати вкладені цикли (один перебирає a, інший b) не варто, бо навіщо *перебирати* b, якщо його можна *визначити*: $\frac{1}{\mathtt{a}}+\frac{1}{\mathtt{b}}=\frac{\mathtt{n}}{\mathtt{m}} \Rightarrow \frac{1}{\mathtt{b}}=\frac{\mathtt{n}}{\mathtt{m}}-\frac{1}{\mathtt{a}}=\frac{\mathtt{n}\cdot\mathtt{a}-\mathtt{m}}{\mathtt{m}\cdot\mathtt{a}} \Rightarrow \mathtt{b}=\frac{\mathtt{n}\cdot\mathtt{a}-\mathtt{m}}{\mathtt{m}\cdot\mathtt{a}}$. Тобто, якщо $(\mathtt{n}\cdot\mathtt{a}-\mathtt{m})$ кратне $(\mathtt{m}\cdot\mathtt{a})$, говоримо, що знайшли $\mathtt{b}=\frac{\mathtt{n}\cdot\mathtt{a}-\mathtt{m}}{\mathtt{m}\cdot\mathtt{a}}$ і припиняємо (єдиний) цикл по a; інакше, продовжуємо. (Доки? Ще не ясно.)

Для суми діє комутативність (переставний закон), що набуває вигляду $\frac{1}{\mathtt{a}}+\frac{1}{\mathtt{b}}=\frac{1}{\mathtt{b}}+\frac{1}{\mathtt{a}}$; великі a означають, що $\frac{1}{\mathtt{a}}$ малі; отже, при великих a, $\frac{1}{\mathtt{b}}$ близькі до $\frac{\mathtt{n}}{\mathtt{m}}$, тобто, b близькі до $\frac{\mathtt{m}}{\mathtt{n}}$, що близько до *нижньої* межі перебору a; отже, якби таке ціле значення b існувало, то воно вже було б знайдене як значення a. Значить, нема смисла дозволяти $\mathtt{b}<\mathtt{a}$ (з урахуванням заборони повтору знаменників, $\mathtt{b}\leqslant\mathtt{a}$), тобто можна вимагати $1<\mathtt{a}<\mathtt{b}$, звідки випливає $\frac{1}{\mathtt{a}}>\frac{1}{\mathtt{b}}$. Значить, раз $\frac{1}{\mathtt{a}}$ є більшим з двох доданків, сума яких рівна $\frac{\mathtt{n}}{\mathtt{m}}$, повинно бути $\frac{1}{\mathtt{a}}>\frac{\mathtt{n}}{\mathtt{m}}/2=\frac{\mathtt{n}}{2\cdot\mathtt{m}}$, що перетворюється у так давно очікувану верхню межу перебору $\mathtt{a}<\frac{2\cdot\mathtt{m}}{\mathtt{n}}$.

Не таку межу, якої хотілося, бо двійка взята з припущення, ніби доданків рівно два. Але краще така межа, яка вийшла, ніж ніякої. Тим па-

[2] Можна подумати, ніби саме́ це робить ДП незастосовним, бо ніяк не ввіпхнути у пам'ять такі масиви. Але не виключено, хоч і не гарантовано, що вийде зберігати рóзв'язки підзадач у словниковій структурі даних, вона ж «асоціативний масив»; це може називатися `map` (`TreeMap`, `HashMap`, ...), `dict`, `Dictionary`, ... Так що питання відкрите. А про що структуру даних почитайте деінде, дуже корисний засіб.

че, що вона піддається хоча б мінімальному узагальненню «якщо кількість доданків $\leqslant k$, то досить розглянути $\mathtt{a} < \frac{k\cdot\mathtt{m}}{\mathtt{n}}$». Адже, якою б не була кількість доданків, комутативність та асоціативність (переставний та сполучний закони) дозволяють[3] розглядати лише випадок, коли першим іде найбільший доданок (найменший знаменник).

Обмеження «якщо кількість доданків $\leqslant k$, то $\mathtt{a} < \frac{k\cdot\mathtt{m}}{\mathtt{n}}$» пропонує *користуватися* кількістю доданків, хоча якраз її-то й треба *знайти*. Але це не щось нечуване. Зокрема, це має добре поєднатися з бінарним пошуком за відповіддю (тема розглянута в багатьох джерелах, зокрема, на «Школі Бобра» 2016 р.). Приблизно так: «реалізуємо функцію розв'язування оберненої задачі *"Розкласти дріб $\frac{n}{m}$ у єгипетське подання з $\leqslant k$ доданками, або з'ясувати, що це неможливо"* та зробимо бінпошук за k: якщо для пробного k результат "неможливо", то вибираємо половину з більшими k (щоб знайти розкладення у більшу кількість доданків, раз у поточну неможливо), а якщо можливо, то запам'ятовуємо самé розкладення і вибираємо половину з меншими k (щоб спробувати знайти краще розкладення, у меншу кількість доданків); так і буде знайдене розкладення у мінімальну кількість».

Щоб чіткіше сформулювати, як ставити та розв'язувати таку обернену задачу, частково використаємо те, як раніше планували ставити серію підзадач ДП, але змінимо згідно з ідеями про бінпошук за відповіддю та про зростання знаменників. Зростання знаменників не лише потрібне для обмеження $\mathtt{a} < \frac{k\cdot\mathtt{m}}{\mathtt{n}}$, а ще й дає простий спосіб уникати повторень знаменників (досить вимагати сáме зростання, а не неспадання), і сильно зменшує перекривання підзадач.[4] Тож поставимо (i,j,k,q)-підзадачі *«або знайти спосіб розкласти дріб $\frac{i}{j}$ у єгипетське подання, де кількість доданків мусить бути $\leqslant k$, а значення знаменників $\geqslant q$, або з'ясувати, що це неможливо»*. Як вже згадувалося, ДП тут якщо й можливе, то лише у вигляді рекурсії з запам'ятовуванням у

[3] Не вимагають, а дозволяють. Тож можна подумати «і навіщо це штучне не обов'язкове обмеження?». Але така думка жахливо помилкова. Властивості вигляду «не обов'язково розглядати багато претендентів, можна обмежитися такою-то їх частиною», зазвичай *дуже(!) корисні*. Можна розглянути й інші випадки — воно-то можна, але навіщо? Якщо варіанти поза межами виділеної частини не дадуть кращого розв'язку? (Дещо спільне з цими міркуваннями є у багатьох доведеннях правильності алгоритмів, причому *і* перебірних, *і* жадібних, *і* ДП…)

[4] Бо не розглядатимуться нáрізно способи, коли відняли ті самі дроби в різному порядку, як-то $\frac{n}{m} - \frac{1}{3} - \frac{1}{5}$ та $\frac{n}{m} - \frac{1}{5} - \frac{1}{3}$. Але стверджувати, ніби підзадачі тепер геть не перекриваються, не можна: наприклад, $\frac{n}{m} - \frac{1}{2} - \frac{1}{12} - \frac{1}{13}$ та $\frac{n}{m} - \frac{1}{3} - \frac{1}{4} - \frac{1}{13}$ дадуть повністю, за всіма параметрами, однакові підзадачі.

словникову структуру (`map`, ...); тепер, коли параметрів стало ще більше, твердження лише посилилося. Так що пишемо рекурсію, а чи включати в неї запам'ятовування, вирішимо пізніше. Наявність чи відсутність запам'ятовувань формально відносить алгоритм до різних класів (ДП і перебір), але тексти програм відрізнятимуться несильно, причому виключно в парі-трійці місць, а не розпорошено по всьому коду, тож можна просто спробувати обидва варіанти й вибрати кращий.

Чому в постановці підзадач сказано «знайти спосіб», а не «спосіб з мінімальною кількістю доданків» (при тих самих параметрах)? Головним чином тому, що загальний досвід і здоровий глузд підказують, що знайти рекурсією який-небудь (перший, який трапиться) спосіб часто значно швидше й легше, ніж шукати рекурсією са́ме мінімальний. Мінімальність буде забезпечена підбиранням k.

Отже, пишемо рекурсивну функцію, що має параметри `nom:QWord` (чисельник, відповідає параметру i з позаминулого абзацу), `denom:QWord` (знаменник, j), `maxDenomAmount:byte` (k) та `minNextDenom:QWord` (q), які збігаються з параметрами підзадачі, і, можливо, ще якісь. Нехай ця функція вертає `true`, якщо їй вдалося знайти розкладення в суму (і `false`, якщо не вдалося), причому якщо вдалося, то послідовність знаменників нехай формується у глобальному масиві `allUsedDenoms`.

Скоротимо дріб $\frac{\texttt{nom}}{\texttt{denom}}$. Якщо $\texttt{nom} = 1$, повернемо `true` (попередньо скопіювавши `denom` у елемент масива `allUsedDenoms`), бо якраз знайшли розкладення. Інакше, якщо $\texttt{maxDenomAmount} > 1$ (ще можна розкладати в кілька дробів), запустимо цикл перебору можливих дробів $\frac{1}{\texttt{a}}$.

Вибравши потрібні частини з усіх раніших пояснень, легко бачити, що `a` слід перебирати в межах від $\max\big((\texttt{denom} \operatorname{div} \texttt{nom}) + 1,$ `minNextDenom`$\big)$ до $(\texttt{maxDenomAmount} \cdot \texttt{denom} - 1) \operatorname{div} \texttt{nom}$ включно. (Щодо верхньої межі́ природніми є питання «чому включно?» і «звідки "$\ldots - 1$"?», але насправді са́ме це і виражає $\texttt{a} < \frac{k \cdot j}{i}$: у випадку, коли $(k \cdot j)$ не кратне i, саме́ цілочисельне ділення (div) дає менший результат, ніж дробове, причому "$\ldots - 1$" не впливає на результат div; якщо ж кратне, то "$\ldots - 1$" забезпечує перехід до меншої цілої частки.)

Діапазон `a`, заданий цими формулами, може виявлятися порожнім. Але це непогано: замість запускати рекурсію, швиденько закінчимо, а ніякі ро́зв'язки при цьому не будуть втрачені, бо всі ме́жі тим і обґрунтовані, що за ними або взагалі нема розв'язків, або такі самі ро́зв'язки з іншим порядком доданків повинні бути знайдені в інших гілках

рекурсії. Це не повинно бути проблемою, слід лише переконатися, що код написаний так, що цикл з 0 ітераціями працює нормально.

Для кожного `a` з діапазона, слід зробити рекурсивний виклик. Ще на початку розбору пояснено, чому чисельник та знаменник нового дробу рівні (до скорочення) $\texttt{nom} \cdot \texttt{a} - \texttt{denom}$ та $\texttt{denom} \cdot \texttt{a}$ відповідно. Значення `maxDenomAmount` глибшого рівня на 1 менше, ніж поточного. Значенням `minNextDenom` глибшого рівня є $\texttt{a} + 1$, бо са́ме це забезпечує, що знаменники розглядаються у порядку зростання. (Можлива, але навряд чи краща, альтернатива — перенести вибір $\max((\texttt{newDenom} \operatorname{div} \texttt{newNom}) + 1, \texttt{a} + 1)$ сюди, спростивши той код, де цей вибір робився раніше.)

Також слід врахувати смисл тих `true` чи `false`, які повертають глибші рівні рекурсії, та з'ясувати, що коли повертати назовні з поточного рівня. Якщо з глибшого рівня повернулося `true`, слід негайно (обірвавши цикл перебору `a`) теж повернути `true` (не забувши додати поточне значення `a` до масиву `allUsedDenoms`, бо са́ме при ньому вдалося сформувати єгипетське подання). Якщо ж глибший рівень рекурсії повернув `false`, треба просто продовжити цикл: при цьому значенні `a` не вдалося, але, можливо, вдасться при одному з подальших...

Після кінця циклу слід повертати `false`, адже до того місця виконання дійде, лише якщо не було дострокового повернення `true`, тобто всі варіанти `a` розглянуті, а сформувати єгипетське подання не вдалося.

З якими параметрами слід викликати цю функцію? Як вже сказано, `maxDenomAmount` (він же k) слід підбирати, щоб узнати; за `nom` та `denom` слід бра́ти n та m зі вхідних даних; за `minNextDenom` можна бра́ти 2 (або, наприклад, $(m \operatorname{div} n) + 1$).

Який асимптотичний час роботи цієї функції? Як для багатьох рекурсій, це визначити важко. Окремої складності додає, що досі не ясно, чи будуть запам'ятовування результатів підзадач; припустимо, ні. Для цього припущення очевидно, що, при однакових `nom` (i), `denom` (j) та `minNextDenom` (q), збільшення `maxDenomAmount` (k) має *сильно* збільшувати час. Якби була, наприклад, більш-менш класична ситуація, коли на кожному рівні рекурсії (крім найглибшого) робиться рівно по 5 рекурсивних викликів, цей час можна було б оцінити як $\left(5^1 + 5^2 + \ldots + 5^{k-1}\right) + 5^{k-1} \in \Theta(5^k)$. Фактично, кількість рекурсивних викликів різна для різних проміжних підзадач, тож оцінити важче, але збільшення k завжди збільшує верхню межу ($\texttt{a} < \frac{k \cdot j}{i}$), частенько не змінюючи нижню. Тому, слід очікувати високої ймовірності ситуацій, коли від збільшення k час зростає ще швидше, ніж $(\text{const})^k$.

Через це, краще спертися в точності на описану обернену задачу, розв'язувати її в точності описаною рекурсією, але зовнішній відносно неї пошук k зробити не бінарним, а послідовним: спробувати подати потрібний дріб $\frac{n}{m}$ одним доданком; якщо не вийшло — двома; якщо не вийшло — трьома; і так, доки не вийде. Таку послідовність дій називають (якщо вкладена функція, як зараз, рекурсивна) «*iterative deepening*» (дослівно, «*послідовне заглиблення*»). Розглянемо приклад (дуже приблизний, обґрунтування цих чисел не існує): нехай час роботи оберненої задачі дорівнює 7^k, мінімальне k, при якому єгипетське подання існує, рівне 7 (але це взнаємо лише після завершення або бінпошуку, або послідовних заглиблень), а верхню межу бінпошука з якихось міркувань оцінили у 12. Тоді сумарний час роботи всіх рекурсій при послідовному заглибленні буде $7 + 7^2 + 7^3 + 7^4 + 7^5 + 7^6 + 7^7 \approx 137$ тис., а при бінпошуці за відповіддю $7^6 + 7^9 + 7^7 \approx 5899$ тис., що, при меншій кількості доданків, значно більше. Звісно, ці приблизні міркування не є доведенням. Але вони дають підстави перевірити це експериментально, й експеримент це в середньому підтверджує. Ще одна перевага послідовного заглиблення над бінпошуком за відповіддю — позбуваємося питання «як визначати верхню межу бінпошука?».

То кінець кінцем вийшов динпрог чи перебір? Окупають себе запам'ятовування чи ні? Експеримент показав, що якщо вважати, що підзадача повторюється лише при повторенні всіх чотирьох (i, j, k, q), то ні. Гіпотетично можна аналізувати підзадачі якось розумніше (включаючи міркування «при цих (i, j, k, q) не буде розкладення, бо вже відомо, що його нема при тих самих i, j, q і більшому k», але не обмежуючись ними); тоді можна очікувати більших відтинань при меншій кількості запам'ятованих підзадач. Але для розглянутої задачі при $1 \leqslant n < m \leqslant 1000$ досить рекурсії без запам'ятовувань.

Ще одна не очевидна помічена експериментально властивість — у середньому, перебір працює швидше, коли перебирати `a` в порядку від максимального $(\texttt{maxDenomAmount} \cdot \texttt{denom} - 1) \operatorname{div} \texttt{nom}$ до мінімального $\max\big((\texttt{denom} \operatorname{div} \texttt{nom}) + 1,\ \texttt{minNextDenom}\big)$.

Задача 2E «Розподіл станцій по зонам»

Керівництво Дуже Довгої Залізниці (ДДЗ) вирішило встановити нову систему оплати за проїзд. ДДЗ являє собою відрізок прямої, на якій послідовно розміщені N станцій. Планується розбити їх на M неперервних зон, що слідують підряд, таким чином, щоб кожна зона мі-

стила хоча б одну станцію. Оплату проїзду від станції j до станції k необхідно встановити рівною $1 + |z_j - z_k|$, де z_j і z_k — номери зон, яким належать станції j та k відповідно. Відома кількість пасажирів, які відправляються за день з кожної станції на кожну іншу.

Напишіть програму, що визначатиме максимальну денну виручку за новою системою при оптимальному розбитті на зони.

Вхідні дані. Перший рядок містить два цілих числá N та M ($1 \leqslant \leqslant M \leqslant N \leqslant 1000$). Другий — одне число, що означає кількість пасажирів, які їдуть між станціями 1 та 2. Третій — два числá, що означають кількість пасажирів, які їдуть між станціями 1 та 3 та між станціями 2 та 3, відповідно. І так далі. В N-ому рядку міститься $N-1$ число, i-е з них визначає кількість пасажирів між станціями i та N. Кількість пасажирів для кожної пари станцій дано з урахуванням руху в обидві сторони. Всі чи́сла цілі, невід'ємні та не перевищують 10000.

Результати. Програма повинна вивести єдине ціле число — шукану максимальну денну виручку.

Вхідні дані	Результати
3 2 200 10 20	440

Примітки. Іншими словами, рядки вхідних даних з 2-го по N-й являють собою нижню-ліву половину симетричної матриці пасажиропотоку з нулями по головній діагоналі (де симетрично розміщені елементи не треба додавати, бо кожен з них — вже сума потоків туди й назад).

Денну виручку 440 можна отримати, якщо розбити станції на зони як «1-а станція у 1-ій зоні, 2-а та 3-я станції у 2-ій зоні». Тоді ціну $1 + 1 = 2$ платитимуть $200 + 10$ пасажирів (які їздять між 1-ю та 2-ю та між 1-ю та 3-ю станціями відповідно), а ціну $1 + 0 = 1$ платитимуть 20 пасажирів (які їздять між 2-ю та 3-ю станціями); $(200 + 10) \times 2 + + 20 \times 1 = 440$. Денної виручки, більшої за 440, досягти неможливо.

Розбір задачі.

Може раптом ефективний розв'язок цієї задачі динпрогом і існує, але ні автору текста, ні авторам задачі він невідомий, а для найприродніших серій підзадач були знайдені контрприклади.

Зокрема, природною є серія підзадач *«Яку максимальну виручку $C(a, b)$ можна отримати, розбивши на b зон станції з 1-ї по a-ту?»*. Але такий підхід, працюючи у схожих випадках, тут неправильний.

1-й (лівіший) з наведених на рис. прикладів є складовою 2-го, бо у 2-му пасажиропотоки між станціями 1–4 такі самі, лише з'являється нова 5-та станція. При цьому в 1-му прикладі пасажиропотік між стан-

ціями 1 та 2 перевищує всі інші, тож межу зон варто провести між ними; у 2-му прикладі потоки між 3 та 5 і 4 та 5 перевищують всі інші, тож обидві межі варто провести там (одну між 3 та 4, іншу між 4 та 5).

```
4 2          5 3
7            7
1 1          1 1
1 1 1        1 1  1
             1 1 99 30
```

Тобто, 5-а станція *вплинула* на те, як оптимально розбити на 2 зони проміжок з 1-ої по 4-ту станції — навіть при тому, що в оптимальному розв'язку $C(5,3)$ одна з меж якраз після 4-ої станції. Отже, для цієї серії підзадач принцип Беллмана не виконується.

Розширювати цю серію підзадач — і не ясно, *як*, і навіть для неї вже є сумніви, чи поміщатиметься виконання програми при $N = 1000$, $M \approx 500$ у обмеження часу; а якщо ще й розширити?..

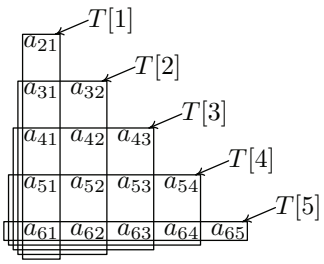
Так що перейдемо до правильного розв'язку, основаного *не* на динамічному програмуванні. Він є уточненням раніше вжитого приблизного міркування *«Пасажиропотік між такими-то станціями настільки великий, що межу між зонами варто провести між цими станціями»*.

Абстрагувавшись від механізму продажу квитків, можна помітити, що сумарна виручка згідно правила «проїзд між зонами z_j та z_k коштує $1 + |z_j - z_k|$» рівно така ж, яка була б, якби: (1) на кожній станції збирали по одиничці грошей з усіх, хто заходить у поїзд; (2) при кожному перетині межі́ між зонами (не на станції, а на перего́ні між ними) збирали ще по одиничці грошей з усіх пасажирів, які перебувають у поїзді. Хто перетинає кілька меж — платить за кожен перетин у момент цього перетину. (Тут *не* пропонується міняти систему оплати (цього не можна, задачу треба розв'язати для вказаної системи), а аналізуються особливості формули $1 + |z_j - z_k|$ і встановлюється її *теоретична рівносильність* описаній системі. А потім усе це використовується, щоб побудувати й обґрунтувати ефективний розв'язок.)

Тобто, скільки пасажирів слідує через перегон між i-ю та $(i+1)$-ю станціями (позначимо цю величину $T[i]$), на стільки вдасться збільшити виручку, якщо са́ме між цими станціями і провести межу між зонами. Оскільки повинно бути M різних непорожніх неперервних зон, то для максимізації виручки ме́жі варто встановити на перегоні з глобально максимальним пасажиропотоком $T[i_{\max}]$, з максимальним серед решти $T[i_{\max_2}]$, і т. д. — всього $M-1$ раз («мінус один», бо меж на 1 менше, ніж зон; наприклад, для двох зон є лише одна межа).

Отже, для розв'язку задачі достатньо спочатку додати всі задані у вхідних даних пасажиропотоки (для всіх пасажирів є складова

ціни «1+...»), потім додати $M-1$ максимальних серед значень $T[1]$, $T[2]$, ..., $T[N-1]$ (наприклад, для цього можна відсортувати масив, що містить значення T, але можна і якось інакше).



Легко бачити, що $T[i]$ можна знаходити як суми виділених на рисунку прямокутних підтабличок вхідних даних. Причому, є «лобова» реалізація цієї ідеї (для кожного прямокутника окремо порахувати суму очевидними вкладеними циклами), а є хитріша, яка використовує, що прямокутники для $T[i]$ і для $T[i+1]$ мають спільну частину. Наприклад, щоб отримати $T[3]$, можна взяти вже знайдене $T[2]$, відняти a_{31} та a_{32}, додати a_{43}, a_{53}, ..., a_{N3}. І виявляється, що асимптотична оцінка часу обчислення всієї послідовності $T[1]$, $T[2]$, ..., $T[N-1]$ для «лобової» реалізації становить $\Theta(N^3)$, а 2-го способу — $\Theta(N^2)$. Оцінку $\Theta(N^2)$ легко довести: при знаходженні $T[1]$, $T[2]$, ..., $T[N-1]$ кожен елемент a_{ij} лише один раз додається і не більше одного разу віднімається.

При бажанні можна сказати, що 2-й спосіб має дещо спільне з ДП, а отже, ДП у задачі є. Але одне, що автор тексту дотримується (не загальноприйнятої, але поширеної) точки зору, що справжній динпрог має містити у своєму рівнянні вибір оптимума, а те, що тут, слід називати рекурентними співвідношеннями чи табличною технікою; інше — навіть якщо це ДП, воно відіграє допоміжну роль і не схоже на природну спробу, згадану на початку.

Наведений розв'язок істотно спирається на *лінійність* залежності вартості від кількості зон $1+|z_j-z_k|$. Насправді більшість перевізників використовують *регресивні* тарифи, коли зі збільшенням відстані тарифи зростають, але повільно, й чим далі їхати, тим менша ціна кожного додаткового кілометра чи кожної додаткової зони. Для регресивних тарифів розглянутий підхід незастосовний, бо збільшення виручки від введення на перегоні межі між зонами не дорівнювало б $T[i]$, а залежало б ще й від того, для скількох пасажирів кількість зон збільшилася з однієї до двох, для скількох з двох до трьох, тощо.

Автори задачі стверджують, що її ідея придумалася якраз під впливом того, як на одній реальній (не українській) залізниці вони спостерігали не регресивний, а приблизно лінійний тариф.

Задача 2F «Ендшпіль»

У грі «*шахи*» грають два гравці; один грає білими, інший чорними. Гра відбувається на шахівниці, тобто дошці 8×8, стовпчики позначаються буквами від "a" до "h" зліва направо, рядки — цифрами від 1 до 8 знизу догори. Кожна клітинка дошки або порожня, або містить одну фігуру. Якщо фігура A (не пішак) може походити згідно з правилами у клітинку, зайняту чужою фігурою B, то внаслідок такого ходу фігуру B б'ють, тобто знімають з дошки. Тому про всі клітинки, куди деяка фігура може походити, кажуть, що вони «під боєм» цієї фігури.

Королю заборонено ходити у клітинки, які перебувають під боєм будь-якої чужої фігури. Якщо один з гравців зробив такий хід, що король суперника опинився під боєм (це називають «шах»), суперник зобов'язаний відповісти таким ходом, щоб його король вже не був під боєм чужої фігури. Якщо виконати цю вимогу неможливо, це називають «*мат*»; гравець, якому поставили мат, програє.

Король може ходити на одну клітинку в будь-якому з 8-ми напрямків (ліворуч, праворуч, вперед, назад, в будь-якому напрямку за будь-якою діагоналлю). Ферзь може ходити в будь-якому з цих самих 8-ми напрямків на будь-яку кількість клітинок, але не перестрибуючи через клітинки, зайняті фігурами (байдуже, своїми чи чужими).

Нехай на шахівниці розміщено три фігури: білий король, білий ферзь і чорний король. Зараз хід білих. За яку мінімальну кількість ходів вони гарантовано зможуть поставити мат? Чорні робитимуть усе, дозволене правилами гри, щоб уникнути мату або відтермінувати його.

Вхідні дані. Програма повинна прочитати спочатку кількість тестових блоків T ($1 \leqslant T \leqslant 70000$), потім самі́ блоки. Кожен блок є окремим рядком, у якому записані позначення трьох клітинок, де розміщені білий король, білий ферзь і чорний король. Позначення клітинки складається із записаних разом букви вертикалі і номера горизонталі, позначення клітинок всередині рядка розділені одиничними пробілами.

Усі задані позиції допустимі з точки зору шахових правил (зокрема, чорний король не під боєм).

Вхідні дані	Результати
2 a3 b3 a1 a3 e3 b1	1 2

Результати. Ваша програма повинна вивести для кожного тесту єдине число — мінімальну кількість ходів. Рахується лише кількість ходів білих (кількість ходів-відповідей чорних не додається).

Розбір задачі.

До задачі навряд чи застосовне ДП. Якщо вважати параметром ДП позицію гри, виникає циклічність залежностей (див. пункт 3 розд. 1.2 та розд. 1.1.5), бо можливо і навіть типово, коли з позиції А можна перейти за два півходи в позицію Б, а з Б за два півходи — в А, й не схоже, щоб щось компенсовувало таку циклічність. (Тут і далі, *півхід* — хід однієї зі сторін (білих чи чорних); це загальноприйнятий термін.)

Закодуємо позиції 19-бітовими числами: один біт — чий хід (чорних чи білих), решта $3 \cdot 2 \cdot 3 = 18$ — координати на дошці кожної з трьох фігур (бо три фігури, а розмір шахівниці становить два виміри по 8 клітинок, де якраз $\log_2 8 = 3$). Таких кодів, включаючи і коди неможливих позицій, 2^{19} (приблизно півмільйона), що для комп'ютера відносно небагато. Розсортуємо їх за трьома групами: мати (хід чорних, король чорних під боєм, ходити нікуди), неможливі (більше однієї фігури в одній клітинці; королі на сусідніх клітинках; хід білих, а чорний король під боєм білого ферзя) і «решта позицій» (їхні статуси поки що невідомі, для більшості пізніше будуть уточнені).

Переглянувши перелік усіх позицій-матів, знайдемо всі *1-півходові* позиції, тобто такі, з яких за один півхід (білих) *можна* прийти у мат. Потім, знаючи про кожну позицію, чи є вона 1-півходовою, переглянемо «решту» позицій і виберемо з них *2-півходові*, тобто позиції, з яких *будь-який* дозволений правилами хід (чорних) веде в 1-півходову. Таким чином, якщо позиція 2-півходова, то після двох півходів (спочатку чорних, потім білих) чорним поставлять мат (при правильній грі білих, для будь-якої допустимої гри чорних).

Далі повторимо аналогічні дії з невеликою відмінністю. Переглянувши перелік усіх 2-півходових позицій, знайдемо всі *3-півходові*, тобто такі, з яких за один півхід (білих) можна прийти у 2-півходову, але при цьому 1-півходові позиції слід пропускати — якщо білі можуть поставити мат за один півхід, нема чого розтягувати гру на три півходи. Потім, переглянемо «решту» позицій і виберемо з них *4-півходові*, тобто такі, будь-який дозволений правилами хід (чорних) з яких веде або в 1-, або у 3-півходову, причому обов'язково існує хоча б один хід (чорних), який приводить у 3-півходову позицію. Таким чином, якщо позиція 4-півходова, то при правильній грі обох сторін після чотирьох півходів (чорних, білих, чорних, білих) чорним поставлять мат. Якщо білі гратимуть неправильно, мат може настати пізніше чи не настати взагалі; якщо чорні — мат може настати вже через два півходи.

Подальші дії (знаючи $2k$-півходові позиції, будуємо $(2k+1)$- та $(2k+2)$-півходові) цілком аналогічні. Процес обривається, коли чергових позицій (експериментально, 21-півходових) вже не з'являється.

Умовою кінця циклу не можна ставити «перелік "решти" позицій став порожнім», бо порожнім він не стає. Зокрема, за рахунок *патових* позицій, де «*пат*» — позиція, в якій повинен ходити гравець, жодна фігура якого не може нікуди походити (не порушуючи правил гри), але, на відміну від мату, король цього гравця не під боєм. За сучасними шаховими правилами, пат є не програшем, а нічиєю. В умові задачі це не описано, але, трактуючи її суто формально, отримуємо цілком узгоджений висновок, що такі позиції не є метою білих. (Чорні ж, у рамках цієї задачі, не можуть приводити гру в такі позиції.)

Ще один момент, який не згаданий в умові чітко, але випливає хоч з неї, хоч з повних шахових правил — при неправильній грі, білі можуть втратити ферзя (підставивши його під бій чорного короля у клітинці, не сусідній зі своїм королем), після чого вже не зможуть поставити мат. Це треба врахувати при відборі позицій-матів, а також або при визначенні переліків можливих ходів з кожної позиції, або при визначенні, які позиції неможливі.

Цей алгоритм виконує досить багато дій (приблизно $P \cdot D \cdot M$, де P — кількість позицій ($\approx 2^{19}$), D — максимальна кількість півходів (20), M — середня кількість ходів з позиції (≈ 10)). Але ці дії майже не залежать від кількості позицій у вхідних даних (задача розв'язується для всіх позицій, і лишається тільки бра́ти готові відповіді).

Розглянутий підхід називається «*ретроаналіз*» і може бути застосований до багатьох ігор двох гравців (якщо, звісно, прийнятні оцінки $\Theta(P)$ пам'яті, $O(P \cdot D \cdot M)$ часу). Зокрема, приблизно так і будують так звані «*таблиці ендшпілів*», де саме́ слово «*е́ндшпіль*» означає «завершальний етап гри (зазвичай, шахів)» і передбачає, що на шахівниці лишилось відносно мало фігур (але не обов'язково аж так мало, як у цій задачі). Зрозуміло, що дослідити таким чином всю гру в шахи абсолютно нереально, бо при більшій кількості фігур кількість позицій стрімко зростає й стає принципово недоступною будь-яким комп'ютерам (принаймні, тим, які існують нині чи будуть створені ближчим часом).

Як бути з тим, що раніше циклічність залежностей зводили до алгоритму Дейкстри, а тепер ні? По-перше, алгоритм Дейкстри не панацея. Десь доречний, десь ні. По-друге, раз тут питають *кількість* ходів, алгоритм Дейкстри природно замінювати пошуком ушир, він же BFS, а щось схоже на BFS тут якраз є: шукаємо спочатку 1-півходові позиції, потім 2-півходові, і т. д. По-третє, в цій задачі (єдиній з переліку) маємо справу з двома гравцями, які мають прямо протилежні цілі; саме це пояснює, чому правила знаходження $(2k+1)$- та $(2k+2)$-півходових позицій різні (а отже, задача не зводиться до стандартного для теорії графів поняття маршруту мінімальної довжини й тому не може бути розв'язана звичайним графовим алгоритмом).

Конкурс «Бебрас-2018» у світі та Україні

11–13 листопада 2018 року в Україні проведено Міжнародний конкурс з інформатики та комп'ютерного мислення «Бобер-2018» для учнів 2–11-х класів. Цього року у конкурсі взяли участь понад 2 мільйони 780 тисяч учнів з 57-ми країн світу.

За кількістю учасників конкурсу Україна посіла сьоме місце у світі:

№ з/п	Країна	К-ть учасників
1	Франція	682053
2	Німеччина	373406
3	Великобританія	201911
4	Білорусь	150237
5	Індія	136607
6	Тайвань	118332
7	**Україна**	**117885**
8	Китай	104573
9	Чехія	79988
10	Словаччина	77928
11	Туреччина	68484
12	Італія	51297
13	Сербія	50168
14	США	46699
15	Литва	44136
16	Австралія	43163
17	Словенія	33590
18	Австрія	32675
19	Угорщина	25464
20	Південна Корея	25455
21	Північна Македонія	25372
22	Хорватія	22887
23	Польща	22540
24	Швейцарія	21313
25	Південна Африка	21035
26	Канада	18874
27	Нідерланди	18852
28	Латвія	17574
29	Румунія	14976
30	Греція	13905
31	В'єтнам	12957
32	Росія	12909
33	Боснія і Герцеговина	9732
34	Пакистан	8754
35	Ірландія	6851
36	Малайзія	6815
37	Туніс	5708
38	Фінляндія	5613
39	Швеція	5499
40	Японія	5128
41	Індонезія	5065

42	Казахстан	4671	50	Філіппіни	2236
43	Гонконг	4659	51	Єгипет	1416
44	Бельгія	4400	52	Ісландія	1357
45	Іран	4161	53	Сингапур	1352
46	Таїланд	4132	54	Іспанія	965
47	Естонія	3458	55	Алжир	796
48	Нова Зеландія	2745	56	Кіпр	526
49	Нігерія	2500	57	Болгарія	514

За кількістю учасників серед учнів 2–3 класів Україна впевнено посідає перше місце:

№ з/п	**Країна**	**К-ть учасників**			
1	**Україна**	**34419**	16	В'єтнам	3451
2	Білорусь	20637	17	Південна Африка	3447
3	Чехія	18494	18	Греція	3364
4	Словаччина	18460	19	Швейцарія	3218
5	Італія	12615	20	Хорватія	2812
6	Франція	10587	21	Нідерланди	2811
7	Великобританія	10021	22	Польща	2677
8	Словенія	9582	23	Північна Македонія	2456
9	Німеччина	8849	24	Пакистан	2420
10	США	8020	25	Південна Корея	1285
11	Китай	6056	26	Ірландія	1174
12	Індія	5782	27	Угорщина	1145
13	Сербія	5023	28	Австрія	1141
14	Литва	3742	29	Індонезія	1092
15	Росія	3582	30	Малайзія	851

31	Філіппіни	564	37	Казахстан	261
32	Швеція	477	38	Нова Зеландія	258
33	Фінляндія	458	39	Ісландія	253
34	Єгипет	451	40	Алжир	106
35	Іран	407	41	Сингапур	28
36	Боснія і Герцеговина	352	42	Гонконг	25

Ще за одним показником Україна значно випереджає інші країни світу:

Кількість шкіл, які взяли участь у конкурсі:

№ з/п	**Країна**	**К-ть учасників**			
1	**Україна**	**4439**	16	Румунія	413
2	Франція	3785	17	Австралія	397
3	Німеччина	2100	18	Індія	380
4	Білорусь	1835	19	Нідерланди	356
5	Італія	1076	20	Пакистан	333
6	Словаччина	979	21	Швейцарія	310
7	Китай	939	22	Австрія	310
8	Росія	711	23	Південна Африка	272
9	Іран	662	24	Північна Македонія	219
10	Чехія	648	25	Угорщина	194
11	Південна Корея	606	26	Боснія і Герцеговина	160
12	Канада	493	27	Болгарія	106
13	Латвія	481	28	Туніс	93
14	Малайзія	431	29	Фінляндія	88
15	Греція	416	30	Кіпр	71

31	Естонія	67	35	Таїланд	31
32	Бельгія	61	36	Ісландія	24
33	Алжир	60	37	Єгипет	16
34	Гонконг	39	38	Нова Зеландія	15

У банк задач, рекомендованих для конкурсу Міжнародним оргкомітетом, були включені задачі українських авторів:

Світлана Васильченко (Запорізька єврейська гімназія «ОРТ-Алєф»),

Ростислав Шпакович (Львівський фізико-математичний ліцей).

Крім того, для українського конкурсу були підготовлені задачі наступними авторами:

Олександр Дробот (Олександрійський НВК Кіровогадської області),

Андрій Мірошниченко (Дніпровська академія неперервної освіти),

Олеся Свіргун (Южноукраїнська ЗОШ № 2 Миколаївської області),

Тетяна Фролова (Великописарівська спеціалізована школа Миколаївської області).

В Україні конкурс проходив у 4439 координаційних центрах, школах чи об'єднаннях шкіл. У ньому взяло участь 117 885 учнів

Загальна кількість учасників за віковими групами:

Клас	2	3	4	5	6
Кількість	15042	19377	18920	13813	11904
	7	8	9	10	11
	9960	9820	8909	5927	4213

Кількість учасників по регіонах України:

Область	**Кількість**		
Вінницька	2419	Львівська	12208
Волинська	2980	Миколаївська	2315
Дніпропетровська	11907	Одеська	6172
Донецька	5914	Полтавська	6331
		Рівненська	3679

Житомирська	2805	Сумська	6350
Закарпатська	2858	Тернопільська	2642
Запорізька	9828	Харківська	8858
Івано-Франківська	2293	Херсонська	5814
Київ	3501	Хмельницька	2517
Київська	3197	Черкаська	2913
Кіровоградська	4843	Чернівецька	2767
Луганська	1393	Чернігівська	2844

Висловлюємо вдячність обласним координаторам за організацію та забезпечення успішного проведення конкурсу у своїх регіонах:

Слушний Олег Миколайович	Вінницька область
Семенюк Ірина Василівна	Волинська область
Мірошниченко Андрій Анатолійович	Дніпропетровська область
Пилипчук Олена Анатоліївна	Донецька область
Жуковський Сергій Станіславович	Житомирська область
Шаркадій Інна Володимирівна	Закарпатська область
Васильченко Світлана Володимирівна	Запорізька область
Микицей Сергій Михайлович	Івано-Франківська область
Мірошниченко Наталія Михайлівна	м. Київ
Федорчук Валерій Анатолійович	Київська область
Чала Марина Станіславівна	Кіровоградська область
Лобода Володимир Вікторович	Луганська область
Зелез Мирон Михайлович	Львівська область
Гапиченко Галина Євгенівна	Миколаївська область
Мітельман Ігор Михайлович	Одеська область
Шостя Світлана Петрівна	Полтавська область
Буняк Володимир Олександрович	Рівненська область

Павленко Ірина Миколаївна	Сумська область
Кривокульський Любомир Євстахович	Тернопільська область
Старченко Людмила Миколаївна	Харківська область
Сисоєнко Наталя Анатоліївна	Херсонська область
Дрижал Олександр Михайлович	Хмельницька область
Шемшур Вадим Михайлович	Черкаська область
Гуменюк Марина Володимирівна	Чернівецька область
Євтушенко Наталія Василівна	Чернігівська область

Наступний конкурс відбудеться 10–12 листопада 2019 року (інформаційний лист Інституту Модернізації Змісту Освіти МОН України № 22.1/10-1002 від 22.03.2019 р.

Детальніше про конкурс на сайті **http://bober.net.ua**

Запрошуємо вчителів та учнів приєднуватись до цікавого масового міжнародного заходу з інформатики.

З М І С Т

Передмова 3

Ростислав Шпакович. Вибрані задачі конкурсу
«Бебрас-2018» та вказівки до їх розв'язування 5

Ірина Скляр. Сортування лінійних масивів 10

Сергій Жуковський. Бітові операції
Основи порозрядної арифметики 38
Задачі 41

Ілля Порубльов. Динамічне програмування
Теоретичний матеріал. 50
Задачі першого дня (динамічне програмування) 66
Задачі другого дня (коли ДП недоречне) 79

Конкурс «Бебрас–2018» у світі та Україні 98

Навчальне видання

Міжнародний конкурс з інформатики та комп'ютерного мислення «Бебрас-2018»

Матеріали школи програмування (Львів-2018)

Редактор *В. Г. Паньков*
Комп'ютерне верстання *У. М. Зарицька*

Формат 60x84/16. Ум. друк. арк. 6,05.
Тираж 6200 пр. Зам. № 940.

Видавництво «Аксіома».
вул. Симона Петлюри, 30а, м. Кам'янець-Подільський, 32300.
Тел./факс: (03849) 3 90 06, (067) 381 29 43.
E-mail: aksiomaprint@ukr.net, sales@aksioma.org.ua
Свідоцтво суб'єкта видавничої справи ДК № 1808 від 26.05.2004 р.