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-2019»

Матеріали школи програмування (Львів-2019)

Кам'янець-Подільський
«Аксіома»
2020

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 10.07.2020 № 22.1/12-Г-562)

М58 Міжнародний конкурс з інформатики та комп'ютерного мислення «Бебрас–2019». Матеріали школи програмування : навчально-методичний посібник / С. С. Жуковський, І. М. Порубльов, І. В. Скляр, Р. С. Шпакович – Кам'янець-Подільський : Аксіома, 2020. – 120 с.

ISBN 978-966-496-513-9

У посібнику подано теоретичний матеріал та задачі, які опрацьовувались слухачами Школи програмування переможців конкурсу Бебрас, яка проходила у жовтні 2019 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 2019 року.

Видання здійснено на благодійницьких засадах
і розповсюджується безкоштовно

УДК 519.683

ISBN 978-966-496-513-9

Передмова

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

Всього у конкурсі «Бебрас -2019» взяли участь понад 2 мільйони 970 тисяч учнів з 54-х країн світу. В Україні конкурс пройшов дванадцятий раз. Детальніше - у статті «Конкурс «Бебрас-2019» 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 шість років у м. Львові проводиться Осіння школа програмування для переможців конкурсу «Бебрас». Остання школа відбулась 13 - 19 жовтня 2019 року.

Цього разу у ній взяли участь 28 учнів з 9 областей України (Дніпропетровської, Запорізької, м. Києва, Кіровоградської, Львівської, Одеської, Полтавської, Рівненської, Сумської).

Протягом 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.

Дуже цікаве заняття провів бронзовий призер Міжнародної студентської олімпіади 2016 року Роман Білий.

Заняття проводились паралельно у двох групах:

1 група – переможці обласних та Всеукраїнських олімпіад.

Теми занять:

1 день – Ілля Порубльов. Рекурсія.

2 день – Роман Білий. Дерева відрізків.

3 день – Ілля Порубльов. Теорія чисел.

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

Теми занять:

1 день – Ірина Скляр. Задачі на ідеї.
2 день – Сергій Жуковський. Моделювання.
3 день – Ірина Скляр. Рядки.

У останній день була проведена олімпіада.

ЇЇ переможцями стали:

11 клас

1. Віктор Рудь (Запорізький ліцей № 105)
2. Тарас Козодой (Володимирецька ЗОШ № 1 Рівненської обл.)

10 клас

1. Іван Боровий ПНЛ № 145, м. Київ)
2. Денис Харьков (Сумська ЗОШ № 15)
3. Максим Дейнега (Полтавський міський багатопрофільний ліцей №1)
4. Михайло Шевченко (Знам'янська ЗШ I–III ст. № 1 Кіровоградської обл.)

9 клас

1. Іван Іщенко (ПНЛ № 145, м. Київ)
2. Тимофій Рейзін (ПНЛ № 145, м. Київ)
3. Дмитро Різник (Запорізька гімназія № 31)

Денис Харьков та Михайло Шевченко відібрались на 4-й го тур Всеукраїнської олімпіади з інформатики 2020 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-2019» 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.

1. Задача «Злиття», автор – Володимир Ксьондзик, м. Львів

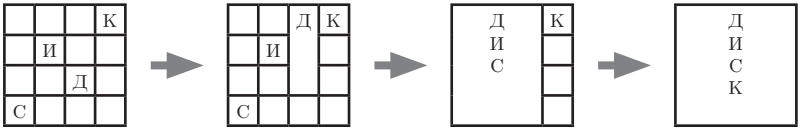
Бобренята грають у таку гру в текстовому редакторі:

Літери слова вписані у клітинки таблиці у довільному порядку.

Можна об'єднувати клітинки довільного прямокутного діапазону.

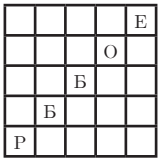
Всі літери у виділеному діапазоні зливаються у один текст зліва направо, зверху вниз.

Наприклад. На малюнку зліва задано початкове розташування літер. Слово "ДИСК" можна утворити за три злиття:



А) Задача для 2–3 класів

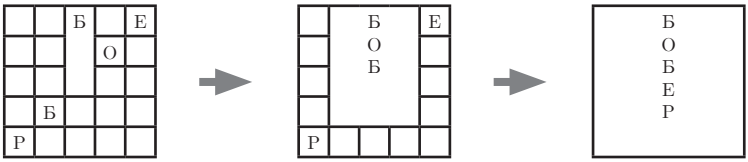
Отримайте слово "БОБЕР" за найменшу кількість злиттів:



Розв'язування:

Задача розв'язується за три операції злиття:

1) Літеру Б підтягуємо у верхню стрічку;
2) Виділяємо і впорядковуємо три перші літери слова;
3) Виділяємо всю таблицю і отримуємо бажаний результат.

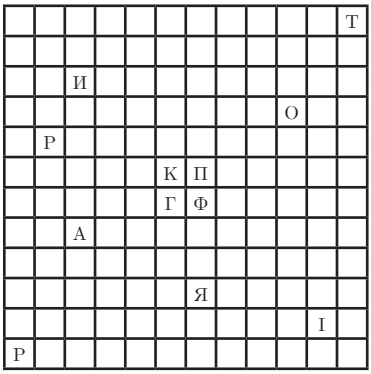


За три злиття задачу розв'язали 14% учнів, за чотири злиття – 69%.

Зауважимо, що саме таким чином виконується злиття тексту у таблицях текстового редактора Open Office.

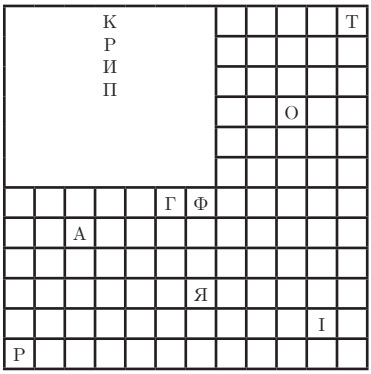
Б) Задача для 10-11 класів

Отримайте слово “КРИПТОГРАФІЯ” за найменшу кількість злиттів.



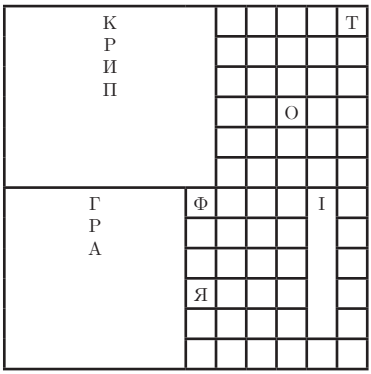
Розв’язування:

Задача розв’язується за сім операції злиття:



1–3 злиття:

4–6 злиття:

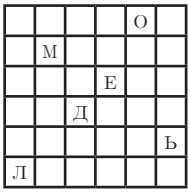


За сім злиттів задачу розв’язали 4% учнів, за вісім злиттів – 55%.

Наступні задачі попробуйте розв'язати самостійно:

В) Задача для 4–5 класів

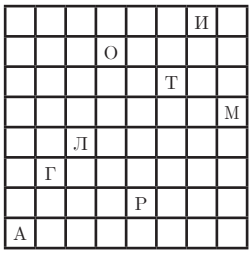
Отримайте слово "МОДЕЛЬ" за найменшу кількість злиттів.



За чотири злиття задачу розв'язали 11% учнів, за п'ять злиттів – 77%.

Г) Задача для 6–7 класів

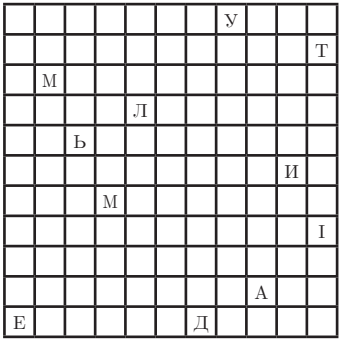
Отримайте слово "АЛГОРИТМ" за найменшу кількість злиттів.



За чотири злиття задачу розв'язали 15% учнів, за п'ять злиттів – 74%.

Д) Задача для 8–9 класів

Отримайте слово "МУЛЬТИМЕДІА" за найменшу кількість злиттів:



За чотири злиття задачу розв'язали 5% учнів, за п'ять злиттів – 66%.

2. Задача «Сірники», автор – Марина Чала, м. Кропивницький

А) Задача для 2–3 класів

Утворіть якнайбільше число. Дозволяється переставити лише два сірники.

Розв'язування:

Найбільше число можна отримати, якщо утворити цифру «9» у старшому розряді. Для цього потрібно переставити два сірники з останньої цифри.

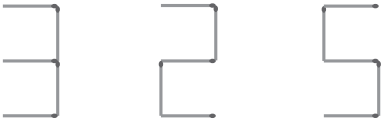
Відповідь: 941. Задачу розв'язали 61% учнів.

Б) Задача для 6–7 класів

Для цього ж розташування сірників утворіть якнайменше трьохцифрове число. Дозволяється переставити лише три сірники. Число з нуля починатись не може. Розв'яжіть цю задачу самостійно.

В) Задача для 8–9 класів

Утворіть якнайбільше число. Дозволяється переставити лише чотири сірники.



Розв'язування:

Найбільше число можна отримати, якщо утворити цифри «9» у двох старших розрядах. Тоді з трьох сірників, що залишились, можна утворити лише цифру «7». Такий спосіб швидкого отримання розв'язку, уникаючи перебору багатьох можливих варіантів, називається жадібним алгоритмом.

Відповідь: 997. Задачу розв'язали 62% учнів.

Наступні дві задачі розв'яжіть самостійно:

Г) Задача для 10–11 класів

Для цього ж розташування сірників утворіть якнайменше трьохцифрове число. Дозволяється переставити лише чотири сірники. Число з нуля починатись не може.

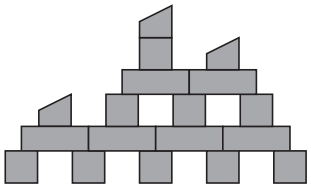
Д) Задача для 4–5 класів

Утворіть якнайменше число. Дозволяється переставити лише два сірники.

Число з нуля починатись не може.

3. Задача «Палац», автор – Світлана Васильченко, м. Запоріжжя (2–11 класи)

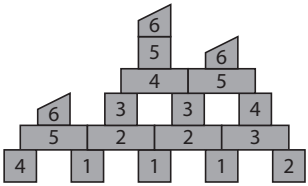
Три бригади бобрів-будівельників повинні побудувати палац з 18 блоків за таким кресленням:



Кожна бригада за один день може збудувати лише по одному блоку. Блок починають будувати тоді, коли нижні блоки, на які він опирається, вже збудовані.

Допоможіть бобрам збудувати палац якнайшвидше. Для цього клікайте по блоках у порядку їх будівництва.

Розв'язування:



Оскільки щодня можна будувати лише по три блоки, роботу можна завершити не швидше, ніж за 6 днів. Знову використаємо жадібний алгоритим. Існує лише три блоки, які можна збудувати у останній день. Позначимо їх цифрою 6. Після цього визначаємо три блоки, які можна збудувати на п'ятий день, і т. д. Таким чином отримаємо єдино можливий план завершення будівництва за 6 днів. Такий ров'язок отримали 7% учнів 2–7 класів, 11% учнів 8–9 класів, та 18% учнів 10–11 класів.

4. Задача «Рибаки» (4–11 класи), автор – Галина Гапиченко, м. Миколаїв

Бобренята Іванко, Петрик, Гануся та Оксанка повернулись з рибалки.

А) Оксанка зловила більше рибин, ніж Гануся.

Б) Іванко та Петрик разом піймали рівно половину сумарного улову.

В) Іванко та Оксанка разом зловили менше, ніж Гануся та Петрик.

Розставте бобренят зліва направо згідно їхніх уловів (від меншого до більшого).

Розв'язування:

А) => Гануся повинна стояти лівіше від Оксанки.

В) => Іванко повинен стояти лівіше від Петрика

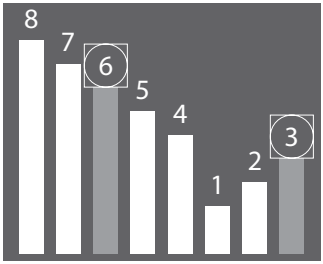
Б+В) => Іванко повинен стояти найлівіше, а Петрик – найправіше.

Відповідь: Іванко, Гануся, Оксанка, Петрик.

Задача на аналіз трьох логічних тверджень виявилась досить важкою. Її розв'язали 9% учнів 4–9 класів, та 15% учнів 10–11 класів.

5. Задача «Впорядкування», автор – Матіас Хірон, Франція

Задача для 6–9 класів



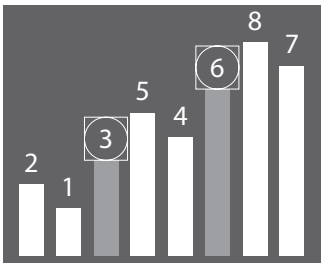
Для стовпчиків висотою 6 і 3 повинно виконуватись правило:

Всі вищі стовпчики повинні стояти справа від нього, а всі нижчі – зліва.

Це правило не повинно виконуватись для жодного іншого стовпчика.

При перетягуванні сусідні стовпчики міняються місцями. Виконайте впорядкування за найменшу кількість обмінів.

Відповідь: 24 обміни.



Основна помилка, яку допускали учні – вони змінювали взаємне розташування стовпчиків висотою 4 і 5, або стовпчиків висотою 7 і 8. За 24 обміни задачу розв'язали 26% учнів.

Відповідь у задачі для 2–3 класів – 7 обмінів, для 10-11 класів – 60 обмінів. Ця задача виявилась найважчою для старшокласників. Лише 2% учнів отримали розв'язок за 60 обмінів. Ще 66% вклались у 70 обмінів. Зазначимо, що лише в українському та французькому конкурсах використовуються такі інтерактивні задачі з зворотнім зв'язком.

У процесі розв'язування учень бачить кількість виконаних обмінів, після завершення впорядкування підсвічуються неправильно розташовані стовпчики. Учень може побачити свої помилки і покращити розв'язок.

6. Задача «Бокс», автор – Андрій Мірошниченко, м. Дніпро

Задача для 6–9 класів

У чемпіонаті світу з боксу серед бобрів приймає участь 65 учасників. У кожному поєдинку турніру визначається переможець, а переможений вибуває з змагань. Організатори укладають пари суперників таким чином, щоб турнір завершився якнайшвидше. Жоден боксер не може проводити більше одного поєдинку на день. За яку мінімальну кількість днів можна провести турнір?

Розв'язування:

У цій задачі теж можна скористатись жадібним алгоритмом:

Щодня потрібно утворювати максимальну кількість пар. У перший день можна утворити 32 пари. Тому в кінці дня залишаться 33 учасники. Відповідно, після кожного наступного дня поєдинків залишатиметься 17, 9, 5, 3, 2 і 1 боксер відповідно.

Відповідь: 7 днів. Таку відповідь отримали 32% учнів.

7. Задача «Подарунки», автор – Володимир Буняк, м. Рівне

А) Задача для 2–9 класів

На день Святого Миколая бобер Клаус готує подарунки для своїх шести бобренят. У нього є великі, середні та малі шоколадки ціною 4 бебро, 2 бебро та 1 бебро відповідно. Кожне бобреня отримає подарунок з 16 шоколадок.

Ціна кожного подарунка 32 бебро.

Клаус вже розклав великі шоколадки по подарунках:

Бобреня	Кількість отриманих шоколадок		
	За 4 бебро	За 2 бебро	За 1 бебро
Матвій	0		
Ніка	1		
Петрик	2		
Ксеня	3		
Оля	4		
Боб	5		

Допоможіть йому розкласти середні та малі шоколадки – заповніть всі порожні клітинки таблиці.

Розв'язування:

Зрозуміло, що Матвій отримав 16 шоколадок ціною по 2 бебро. У кожній наступній стрічці кількість шоколадок по 4 бебро зростає на 1. Щоб загальна вартість та кількість шоколадок у кожній стрічці не змінювалась, кількість шоколадок по 1 бебро повинна зростати на 2, а кількість шоколадок по 3 бебро повинна зменшуватись на 3:

Бобреня	Кількість отриманих шоколадок		
	За 4 бебро	За 2 бебро	За 1 бебро
Матвій	0	16	0
Ніка	1	13	2
Петрик	2	10	4
Ксеня	3	7	6
Оля	4	4	8
Боб	5	1	10

Задачі на знаходження наборів цілочисельних значень у теорії алгоритмів входять у розділ «Діофантові рівняння».

У другому класі задачу розв'язали 35% учнів, у дев'ятому класі – 40%.

Б) Задача для 10–11 класів

Є великі, середні та малі шоколадки ціною 4 бебро, 2 бебро та 1 бебро відповідно. Кожний подарунок складається з 32 шоколадок. Ціна кожного подарунка 64 бебро.

Яку максимальну кількість великих шоколадок можна покласти в один подарунок, щоб виконувались вказані умови?

Знайдіть розв'язок самостійно, використовуючи цю ж ідею.

8. Задача «Рецепти», автор – Юлія Троян, м. Кременчук

А) Задача для 8–9 класів.

На день народження мами Оленка вирішила приготувати оригінальний торт. Для цього вона зробила три пошукові запити у електронній базі рецептів приготування тортів. Їх результати Оленка записала у електронну таблицю. У запит "торт з шоколадом або горіхами" включаються всі торти, що містять або шоколад, або горіхи, або шоколад і горіхи одночасно.

B6 f_x Σ =

	A	B
1	**Назва запиту**	**Кількість рецептів**
2	торт з шоколадом або горіхами	12000
3	торт з шоколадом і горіхами	6500
4	торт з горіхами	7700
5		
6	торт з шоколадом	?

За цими даними вона вирішила підрахувати, скільки є рецептів для тортів з шоколадом. Яку формулу для цього потрібно внести в комірку B6?

Розв'язування:

Кількість тортів без шоколаду: **B4**–**B3**=1200.

Всі інші торти містять шоколад.

Відповідь: **B2**+**B3**–**B4**.

Її отримали 40% учнів.

Б) Задача для 10–11 класів.

У запит "торт (з шоколадом і горіхами) або (з шоколадом і фруктами)" включаються всі торти, що містять або шоколад і горіхи; або шоколад і фрукти; або шоколад, горіхи і фрукти одночасно.

B6 f_x Σ =

	A	B
1	**Назва запиту**	**Кількість рецептів**
2	торт (з шоколадом і горіхами) або (з шоколадом і фруктами)	1100
3	торт з шоколадом і фруктами	600
4	торт з горіхами, і фруктами, і шоколадом	50
5		
6	торт з шоколадом і горіхами	?

За цими даними вона вирішила підрахувати, скільки є рецептів для тортів з шоколадом і горіхами. Яку формулу потрібно внести в комірку B6.

Цю і наступну задачі розв'яжіть самостійно.

9. Задача «Пилорама», автор – Ахто Труу, Естонія, (10–11 класи)

Андрій має невелику пилораму, на якій виробляє дерев'яні бруси на замовлення. Він хоче розподілити своїх замовників на дві категорії:

гуртовики – клієнти, що замовляють не менше 50 виробів;

роздрібні замовники – клієнти, що замовляють менше 50 виробів.

Яку формулу потрібно ввести в комірку C2 та скопіювати на діпазон C3:C11 електронної таблиці:

=IF(B2>50;»гурт»;»роздріб»)
=IF(B2<50;»гурт»;»роздріб»)
=IF(B2>50;»роздріб»;»гурт»)
=IF(B2<50;»роздріб»;»гурт»)

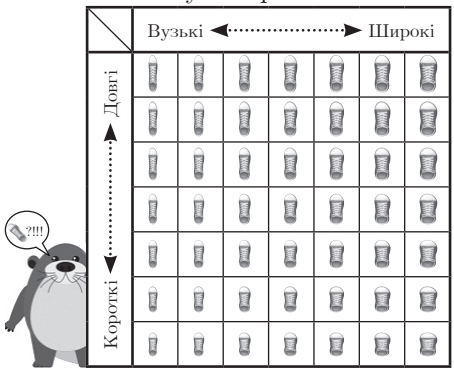
Лише 22% учнів вибрали правильну формулу.

C2 f_x Σ =

	A	B	C
1	**Замовник**	**Кількість виробів**	**Тип замовлення**
2	Тіна		
3	Ганна		
4	Антон		
5	Ігор		
6	Марія		
7	Карл		
8	Микола		
9	Гриць		
10	Марко		
11	Семен		
12	Оля		

10. Задача «Кеди», автор – Джіх'є Кім, Корея, (8–11 класи)

Бобрик купує кеди у магазині спорттоварів. Кеди впорядковані на стелажі по розмірах, як вказано на малюнку. Бобрик не пам'ятає свого розміру.



Тому вибирає і примірює довільну пару зі стелажу. Під час кожного примірювання він отримує два наступні види інформації:

а) по ширині: розмір – завузький, нормальний або зашироκий;

б) по довжині: розмір – закороткий, нормальний або задовгий.

Допоможіть бобрику знайти його розмір за найменшу кількість примірювань.

Розв'язування:

Кеди можна підібрати за три примірювання, використовуючи бінарний пошук. Спочатку потрібно приміряти центральну пару (четвертий стовпчик, четвертий ряд). Після цього зона пошуку зменшується до квадрата розмірами 3x3. У цьому квадраті знову вибираємо центральну пару кедів.

Після другого примірювання розташування потрібного розміру буде визначено однозначно. За три примірювання задачу розв'язали 30% учнів.

Сергій Жуковський. Функції та рекурсія.

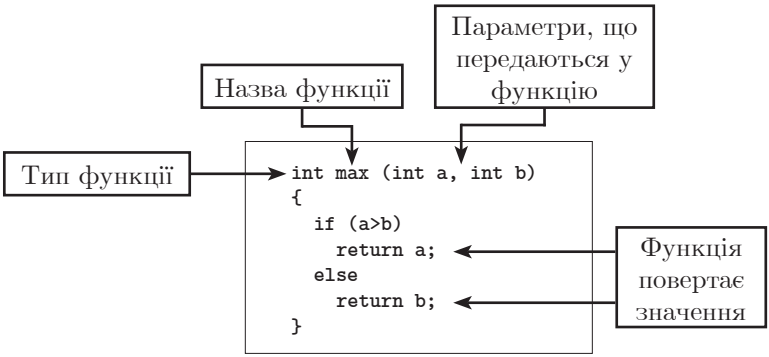
Функції мовою C++

Програма мовою C/C++ складається з функцій, одна з яких повинна мати ім'я main.

Структура кожної функції співпадає зі структурою головної функції.

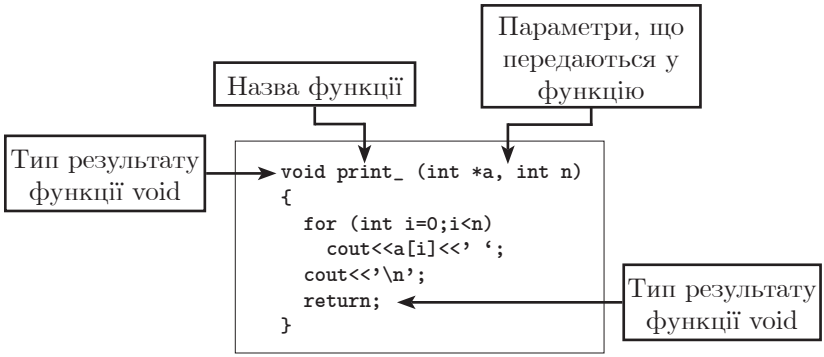
```
Тип_результату_функції  Назва_функції(список параметрів)
{
  Оператори тіла функції;
}
```

Функції мови C/C++ бувають двох типів: функція, що повертає значення, та функція, яка не повертає значення. Функція може повертати значення як стандартного типу (int, long, float, double...), так і користувацького типу (наприклад, структура).



Функція повертає значення за допомогою команди **return <значення>**, після чого функція закінчує своє виконання. Якщо функція, яка повертає значення, не буде містити команди **return**, при деяких налаштуваннях компілятор повідомить про помилку, але при більшості налаштувань лише видасть попередження (warning) і буде повертати випадкове значення.

Функції, які не повертають значення, мають тип **void**. Закінчення виконання функції відбувається за допомогою команди **return**, без параметрів або після виконання останнього рядка функції. Функція типу **void** може не містити команди **return**.



Функції в програмі можна записувати перед головною функцією, після головної, чи в іншому файлі.

Якщо функція записана після головної функції, то перед головною функцією її необхідно описати.

Якщо функція записана в окремому файлі, то необхідно підключити цей файл за допомогою директиви #include.

Програма знаходження найбільшого спільного дільника для двох натуральних чисел

```
/* функція NSD записана перед
головною функцією */
#include<iostream>
using namespace std;

int NSD (int x, int y)
{
  while (x != y)
    if (x>y) x -= y;
    else y -= x;
  return x;
}

int main()
{
  int a, b, rez;
  cout << "Введіть a, b";
  cin >> a >> b;
  rez = NSD(a, b);
  cout << "NSD=" << rez;
  return 0;
}
```

```
/* функція NSD записана після
головної функції */
#include<iostream>
using namespace std;
int NSD (int x, int y);

int main()
{
  int a, b, rez;
  cout << "Enter a, b";
  cin >> a >> b;
  rez = NSD(a, b);
  cout << "NSD=" << rez;
}

int NSD(int x, int y)
{
  while (x != y)
    if (x>y) x -= y;
    else y -= x;
  return x;
}
```

```
// функція NSD записана у іншому файлі

// файл NSD.cpp

#include <iostream>
#include «algoritm.cpp»
using namespace std;

int main()
{
  int a, b, rez;
  cout << "Enter a, b";
  cin >> a >> b;
  rez = NSD(a, b);
  cout << "NSD=" << rez;
  return 0;
}

// файл algoritm.cpp

int NSD(int x, int y)
{
  while (x != y)
    if (x>y) x -= y;
    else y -= x;
  return x;
}
```

Передача параметрів у функцію

Змінні, які оголошені перед функціями або перед їх оголошенням, називаються глобальними.

Глобальні змінні можна використовувати в будь-якій функції.

Змінні, які оголошені у функції, називається локальними. Вони видимі тільки у даній функції.

Передача змінних у функцію відбувається через параметри функції. Щоб повернути значення функції через параметри, використовують знак посилання &. Є різні способи це робити, один з них годиться для всіх C/C++, інший лише для C++. Наведений далі приклад відповідає способу, можливому лише для C++.

Правильно

```
#include <iostream>
using namespace std;

void SWAP(int &x, int &y)
{
  int z = x; x = y; y = z;
}

void main()
{
  int a, b, rez;
  cout << "Введіть a, b";
  cin >> a >> b;
  SWAP(a, b);
  cout << "a=" << a \
    << "b=" << b;
}
```

Неправильно

```
#include <iostream>
using namespace std;

void SWAP(int x, int y)
{
  int z = x; x = y; y = z;
}

void main()
{
  int a, b, rez;
  cout << "Введіть a, b";
  cin >> a >> b;
  SWAP(a, b);
  cout << "a=" << a \
    << "b=" << b;
}
```

Під час використання масива в якості параметра передається вказівник на його перший елемент.

```
// masiv.cpp

#include <iostream>
#include "algoritm.cpp"
using namespace std;

int main()
{
  int masiv[20], max, n;
  cout << "vvedit n ";
  cin >> n;
  for(int i = 0; i < n; i++)
    cin >> masiv[i];
  max = max_mas(masiv, n);
  cout << "max=" << max << endl;
  sort_mas(masiv, n);
  out_masiv(masiv, n);
  return 0;
}
```

```
// algoritm.cpp

#include <iostream>
using namespace std;

void swap(int &x, int &y)
{
  int c = x; x = y; y = c;
}

int max_mas(int *a, int n)
{
  int max = a[0];
  for(int i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}

void sort_mas(int *a,  int n)
{
  int f = 1;
  while(f==1)
  {
    f = 0;
    for(int i = 0; i < n - 1; i++)
      if (a[i] > a[i+1])
      {
        swap(a[i], a[i+1]); f = 1;
      }
    n--;
  }
}

void out_masiv(int *a, int n)
{
  cout << "\n";
  for(int i = 0; i < n; i++)
    cout << a[i] << ' ';
  cout << "\n";
}
```

Рекурсія

Рекурсія – це процес звернення функції до самої себе.

Приклади рекурсії:

- зображення журналу в цьому самому журналі;
- відображення в двох дзеркалах, які висять одне навпроти одного;

- в телевізорі видно зображення цього телевізора, в якому видно зображення цього телевізора, і т. д.;
- свист мікрофона.

Рекурсія використовується для спрощення запису алгоритму.

Під час написання рекурсивних програм необхідно дотримуватись таких правил:

- рекурсивна функція повинна містити перевірку останнього входження;
- рекурсивна функція повинна звертатись до самої себе безпосередньо, або через іншу функцію.

Рекурсивні задачі можна розбити на 2 види:

1. Задачі з рекурсивним формулюванням;
2. Задачі, які можна звести до рекурсивних.

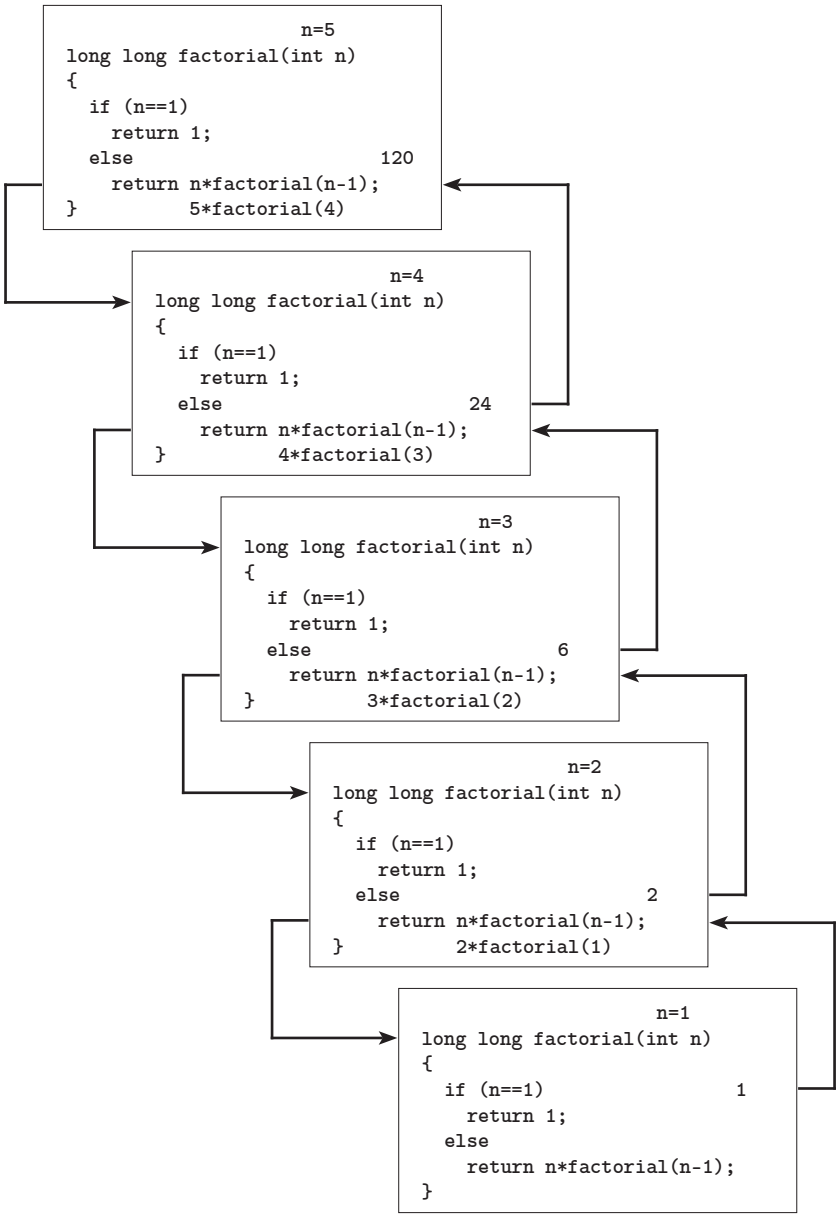
1. Задачі з рекурсивним формулюванням;

Приклад.

Обчислити факторіал натурального числа.

n! = 1*2*3*4*...*n
n! = n * (n–1)!
(n–1)! = (n–1) * (n–2)!
(n–2)! = (n–2) * (n–3)!
.....
3! = 3 * 2!
2! = 2 * 1!
1! = 1

```
#include <iostream>
using namespace std;
long long factorial(int n)
{
  if (n==1)
    return 1;
  else
    return n*factorial(n - 1);
}
int main()
{
  int n;   long long f;
  cout << "Vvedit n";
  cin>>n;
  f = factorial(n);
  cout << "factorial(" << n << ")=" << f;
  return 0;
}
```


n=5
long long factorial(int n)
{
if (n==1)
return 1;
else 120
return n*factorial(n-1);
} 5*factorial(4)
n=4
long long factorial(int n)
{
if (n==1)
return 1;
else 24
return n*factorial(n-1);
} 4*factorial(3)
n=3
long long factorial(int n)
{
if (n==1)
return 1;
else 6
return n*factorial(n-1);
} 3*factorial(2)
n=2
long long factorial(int n)
{
if (n==1)
return 1;
else 2
return n*factorial(n-1);
} 2*factorial(1)
n=1
long long factorial(int n)
{
if (n==1) 1
return 1;
else
return n*factorial(n-1);
}

2. Задачі, які можна звести до рекурсивних;

У деяких задачах рекурсія не присутня у прямому вигляді, але їх можна звести до рекурсивних.

Приклад. Знайти найбільший спільний дільник двох натуральних чисел.

НСД(24, 42) 42–24

НСД(24, 18) 24–18

НСД(6, 18) 18 6

НСД(6, 12) 12–6

НСД(6, 6)

```
#include<iostream>
using namespace std;
int NSD (int a, int b)
{
  if (a==b)
    return a;
  else if (a > b)
    return NSD (a-b, b);
  else
    return NSD(a, b-a);
}
int main()
{
    int x, y, res;
    cout << "Введіть x, y";
    cin >> x >> y;
    res = NSD(x, y);
    cout << "NSD(" << x << ',' << y << ")=" << res;
    return 0;
}
```

Перевантаження функцій

Часто буває, що доводиться виконувати одні і ті ж операції для різних типів даних, або різної кількості даних. В мові C++ (але не C) можна використовувати декілька функцій з різними типами параметрів, але з одним іменем.

Наприклад:

```
int max(int x, int y);
long max(long x, long y);
```

```
#include<iostream>

using namespace std;

#include <algoritm.cpp>
#include <conio.h>

void main()
{
  int a, b, c;
  float x, y;
  cout << "\nVvedit a, b";
  cin >> a >> b;
  cout << max(a, b);
  cout << "\nVvedit x, y";
  cin >> x >> y;
  cout << max(x, y);
  cout << "\nVvedit a, b, c";
  cin >> a >> b >> c;
  cout << max(a, b, c);
  return 0;
}
```

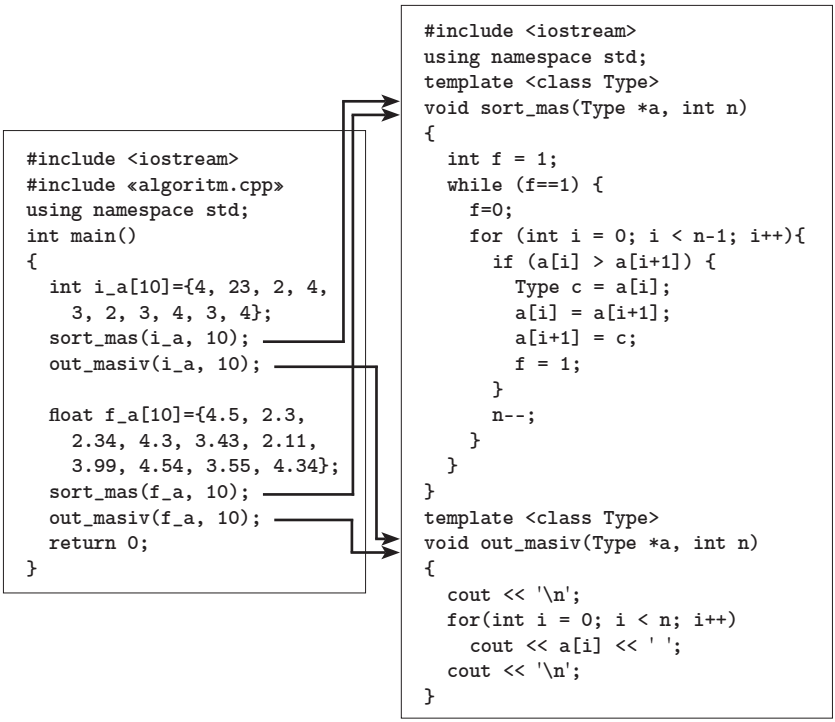
```
// algoritm.cpp

#include<iostream>
using namespace std;

int max(int a, int b)
{
  if (a>b) return a;
  else return b;
}

float max(float a, float b)
{
  if (a>b) return a;
  else return b;
}

int max(int a, int b, int c)
{
  if (a > b)
    if (a > c) return a;
    else return c;
  else
    if (b > c) return b;
    else return c;
}
```

Шаблони функцій

Деякі алгоритми на залежать від типів даних, з якими вони працюють (сортування масиву). Для таких алгоритмів в мові C++ (але не C) не потрібно переписувати алгоритми. Для таких алгоритмів можна передати в параметр тип даних. Для цього використовують шаблони функцій.

Формат функції шаблона:

```
template <classType> назва_функції ( параметри )
{
  // тіло функції;
}
```



```
#include <iostream>
#include «algoritm.cpp»
using namespace std;
int main()
{
  int i_a[10]={4, 23, 2, 4,
    3, 2, 3, 4, 3, 4};
  sort_mas(i_a, 10);
  out_masiv(i_a, 10);

  float f_a[10]={4.5, 2.3,
    2.34, 4.3, 3.43, 2.11,
    3.99, 4.54, 3.55, 4.34};
  sort_mas(f_a, 10);
  out_masiv(f_a, 10);
  return 0;
}
```

```
#include <iostream>
using namespace std;
template <class Type>
void sort_mas(Type *a, int n)
{
  int f = 1;
  while (f==1) {
    f=0;
    for (int i = 0; i < n-1; i++){
      if (a[i] > a[i+1]) {
        Type c = a[i];
        a[i] = a[i+1];
        a[i+1] = c;
        f = 1;
      }
      n--;
    }
  }
}
template <class Type>
void out_masiv(Type *a, int n)
{
  cout << '\n';
  for(int i = 0; i < n; i++)
    cout << a[i] << ' ';
  cout << '\n';
}
```

Задачі, які були запропоновані на школі Бобра

Всі задачі з сайту e-olymp.com

8762. Сума (функції)

Реалізуйте функцію:

```
integer solve(integer a, integer b)
```

a – перше число;

b – друге число;

функція має повертати одне ціле число – суму двох чисел.

Вхідні дані: Два цілих числа a та b ($1018 \le a, b \le 1018$) – числа, які потрібно додати.

Вихідні дані: Буде виведено одне число – суму двох чисел.

Розв'язок.

Дана задача потребує здати тільки функцію, яка отримує два параметри, та повертає суму цих чисел:

```
long long solve(long long a=0, long long b=0)
{
  long long c = a + b;
  return c;
}
```

Примітка 1. В цій задачі (та ще деяких) потрібно здати розв'язок, що містить лише функцію, а не всю програму. При здачі, сайт e-olymp сам пропонує заголовок цієї функції.

Примітка 2. Записи «=0» після параметрів означають, що якщо не передати ніяке значення, за промовчанням буде взято 0. Задачу можна розв'язати і без цього.

9394. Сума цифр числа (функція)

Написати функцію

```
int suma(int n)
```

яка повертає суму цифр числа n.

Наприклад: для числа 123 сума цифр числа дорівнює 6.

Розв'язок:

Дана функція отримує одне ціле число, і повертає суму цифр числа.

```
int suma(int a)
{
  int s = 0,  c;
  a = abs(a);
               // знаходимо модуль числа на випадок від'ємного числа
  while (a>0)  // поки число більше 0
  {
    c = a%10;  // знаходимо останню цифру числа
    s += c;    // до суми додаємо цифру
    a /= 10;   // відкидаємо останню цифру
  }
  return s;    // функція повертає суму цифр
}
```

Цю саму функцію можна реалізувати рекурсивно:

```
int suma(int n)
{
  if (n==0) return 0;
  return abs(n % 10) + suma(n / 10);
}
```

20 Скільки можна?

Дано натуральне число n. Від даного числа віднімемо суму цифр цього числа, від утвореного числа знову віднімемо суму цифр утвореного числа, і т. д. Дану операцію над числом будемо виконувати, поки утворене число додатне. Скільки разів будемо виконувати дану операцію?

Вхідні дані: Одне натуральне число n, що не перевищує $2*10^9$.

Вихідні дані: Кількість виконаних операцій.

Приклад введення	Приклад виведення
21	3

Розв'язок:

Застосуємо функцію суму цифр числа. Змоделюємо процес, вказаний в умові задачі. Поки число більше нуля, від числа віднімемо суму цифр числа. Кожного разу кількість операцій збільшимо на 1.

```
#include <iostream>
using namespace std;

int suma(int n)
{
  if (n==0) return 0;
  return abs(n % 10) + suma(n / 10);
}

int main()
{
  int n, k = 0;
  cin >> n;
  while (n > 0)        // поки число більше 0
  {
    n -= suma(n);      // від числа віднімемо суму цифр
    k++;               // збільшимо кількість на 1
  }
  cout << k << endl;
  return 0;
}
```

2999 Функція-10

Задано функцію, аргументи якої – невід'ємні цілі числа m та n (m ≤ n):

$$f(m,n)=\begin{cases}1, if\ m=0\\1, if\ m=n\\f(m-1,n-1)+f(m,n-1), if\ 0<m<n\end{cases}$$

Обчисліть значення функції.

Вхідні дані: Два цілих невід'ємних числа n та m ($0 \le n,\ m \le 20$).

Вихідні дані: Виведіть шукане значення заданої функції $f(m,\ n)$.

Розв'язок:

Реалізуємо рекурсивну функцію, записану в умові задачі.

```
#include <bits/stdc++.h>
using namespace std;

long long f(long long m, long long n)
{
  if (m==0) return 1;
  if (m==n) return 1;
  return f(m-1, n-1) + f(m, n-1);
}

int main()
{
  long long n, m;
  cin >> n >> m;
  cout << f(m, n) << endl;
  return 0;
}
```

8304 Весела функція

Обчисліть значення функції

$$f(x,y)=\begin{cases}0, x\le 0\ or\ y\le 0\\f(x-1,y-2)+f(x-2,y-1)+2, x\le y\\f(x-2,y-2)+1, x>y\end{cases}$$

Вхідні дані: Два цілих числа $x,\ y$ ($0 \le x,\ y \le 50$).

Вихідні дані: Вивести значення функції $f(x,\ y)$.

Приклад введення	Приклад виведення
2 3	4
34 12	6

Розв'язок:

Задача подібна до попередньої, проте, якщо реалізувати те, що вказано в умові задачі, перевірка повідомить про перевищення ліміту часу на декількох тестах:

```
#include <iostream>
using namespace std;

long long f(long long a, long long b)
{
  long long d;
  if(a<=0 || b<=0)  d = 0;
  else
    if(a<=b)
      d = f(a - 1, b - 2) + f(a - 2, b - 1) + 2;
    else
      d = f(a - 2, b - 2) + 1;
  return d;
}

int main()
{
  long long x, y, k;
  cin >> x >> y;
  k = f(x, y);
  cout << k << endl;
}
```

На даному прикладі можна показати недоліки рекурсії. В реалізації рекурсії функція звертається багато разів до рекурсивної функції з одними і тими ж значеннями. Щоб цього запобігти, створимо двовимірний масив, розміром `a[51][51]`, в якому будемо зберігати всі значення функції, з відповідними індексами. Якщо у відповідній комірці значення знайдено (в даній комірці не 0), функція буде повертати значення з масиву. Якщо значення 0, тоді будемо рекурсивно викликати функцію.

```
#include <bits/stdc++.h>
using namespace std;

long long a[51][51] = { 0 };
long long f(long long x, long long y)
{
  if (x <= 0 || y <= 0) return 0;
  if (a[x][y] > 0) return a[x][y];
  if (x <= y) {
    a[x][y] = f(x - 1, y - 2) + f(x - 2, y - 1) + 2;
    return a[x][y];
  }
  else // x > y
  {
    a[x][y] = f(x - 2, y - 2) + 1;
    return a[x][y];
  }
}

int main()
{
  long long x, y;
  cin >> x >> y;
  cout << f(x, y) << endl;
  return 0;
}
```

9026 Проста задача (функції)

Вам потрібно реалізувати одну підпрограму:

```
void solve( integer n)
```

Підпрограма отримує двоцифрове число і виводить через пропуск кожну цифру окремо.

Наприклад, якщо дано число 23, то процедура повинна вивести 2 3.

Приклад введення	Приклад виведення
23	2 3

```
void solve(int a)
{
    cout << a / 10 << « « << a % 10;
}
```

3936 Ханойські вежі

Задано три стержня. На першому стержні знаходиться декілька дисків зверху донизу за зростанням розміру диска. Два інші порожні. Потрібно перенести усі диски з першого стержня на другий. Переносити диски дозволяється лише по одному. Не дозволяється класти більший диск на менший. У програмі заборонено користуватись циклами.

Вхідні дані: Кількість дисків n ($1 \le n \le 19$) на першому стержні.

Вихідні дані: Виведіть по два числа у рядку – номери стержнів, звідки і куди переноситься диск. Розв'язок повинен бути найкоротшим.

Приклад введення	Приклад виведення
3	1 2 1 3 2 3 1 2 3 1 3 2 1 2

Розв'язок:

Рекурсивна функція описує процес переміщення одного диску. Для того, щоб перемістити n дисків зі стержня 1 на стержень 2 за допомогою стержня 3, потрібно спочатку n–1 диск перемістити з стержня 1 на стержень 3, потім перемістити один диск зі стержня 1 на стержень 2, потім перемістити n–1 диск зі стержня 3 на стержень 2.

```
#include <iostream>
#include <cmath>

using namespace std;
int k = 0;
void hanoi(int n, int a, int b, int c)
// перенесення n дисків з a на b, використовуючи c як допоміжний
{
  if(n==0)
    return;  // якщо диски закінчилися, нічого не робимо
  hanoi(n - 1, a, c, b);
  // переміщаємо n-1 диск зі стержня a на стержень c
  cout << a << " " << b << endl;
  // переміщаємо один диск зі стержня a на стержень b
```

```
  hanoi(n - 1, c, b, a);
  // переміщаємо n-1 диск зі стержня c на стержень b
}

int main()
{
  int n;
  cin >> n;
  hanoi(n, 1, 2, 3);
  return 0;
}
```

9395 Кількість дільників (Функція)

Задано ціле число n. Потрібно знайти кількість його дільників, не рахуючи 1 та самого числа n.

Написати функцію:

```
int CountDivisors(int n) // C++
```

яка повертає кількість дільників числа n.

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

Вихідні дані: Вивести кількість дільників числа n.

Приклад введення	Приклад виведення
2	0

Розв'язок.

Для того, щоб знайти кількість дільників потрібно перебрати всі числа i від 2 до n–1, і перевірити, якщо число n ділиться на i, то збільшуємо кількість дільників на 1. Проте число $n < 2^{31}$ ($\approx 2*10^9$), і даний алгоритм буде працювати більше 1 сек. Щоб скоротити кількість операцій, будемо перевіряти на подільність від 2 до кореня квадратного з n, і кожного разу додавати 2 дільники (з міркувань, що є дільник i та дільник n / i). В кінці перевірити, чи число n є квадратом (якщо квадрат, то потрібно відняти 1, бо цілий корінь додали двічі, а треба один раз.)

```
int CountDivisors(int n)
{
    int t = sqrt(n), i, k = 0;
    for (i = 2; i<=t; i++)
        if (n%i==0) k += 2;
    if(t*t == n)  k--;
    return k;
}
```

2712 Задача про ферзів

Вам, напевне, добре відома класична задача про розстановку ферзів: на шаховій дошці $n^{\times}n$ потрібно розставити n ферзів таким чином, щоб ніякі два ферзі не били один одного. Така розстановка ферзів називається мирною. Проте у даній задачі нас буде цікавити не якась одна мирна розстановка ферзів, а усі різні мирні розстановки. Точніше, їх загальна кількість. Наприклад, для дошки $8^{\times}8$ існує 92 різних мирних розстановок ферзів.

Вхідні дані: У вхідному файлі записано єдине натуральне число n (n ≤ 12).

Вихідні дані: У вихідний файл виведіть шукану кількість мирних розстановок ферзів.

Приклад введення	Приклад виведення
8	92

Розв'язок:

Класична перебірна задача пошуку кількості розстановок ферзів на шаховій дошці $n^{\times}n$.

Створимо масив `mas[]`, в якому будемо зберігати розміщення ферзів на дошці. В k-й комірці записане число `mas[k]`, що означає, що на дошці в клітинці з координатами (`k`, `mas[k]`) розміщений ферзь.

Рекурсивна функція описує розміщення одного ферзя в рядку k.

Якщо n ферзів розставлено (ввійшли в рекурсію з номером n+1) кількість розстановок збільшуємо на 1.

Проходимо по рядку, якщо на i-й позиції ферзь не б'є всі попередні розставлені ферзі, його ставлять (в комірку `mas[k]` присвоюють значення i), і викликається функція із значенням k+1 (заходимо в наступний рядок).

Функція `boy` перевіряє, чи б'є ферзь з координатами (`x`, `y`) хоч одного ферзя, що розміщені на y–1 попередніх рядках, і повертає true, коли не б'є жодного ферзя, і false, коли б'є хоча б одного ферзя.

Один ферзь б'є іншого, коли вони знаходяться в одному стовпці (координата `x==mas[i]`), знаходяться в одному рядку (це виключено в алгоритмі, бо для кожного рядка вибираємо лиш одного ферзя) або знаходиться на діагоналі (`mas[i] + i == x + y` або `mas[i] - i == x - y`).

```
#include <bits/stdc++.h>
using namespace std;
int mas[15] = {0}, n, res = 0;

bool boy(int x, int y)
{
  for(int i = y - 1; i > 0; i--)
  {
    if(mas[i] + i == x + y || mas[i] - i == x - y || x == mas[i])
      return false;
  }
  return true;
}

void rec(int k)
{
  if (k == n + 1)
  {
    res++;
    return;
  }
  for (int i = 1; i <= n; i++)
  {
    if (boy(i, k)) {  mas[k] = i;  rec(k+1);  }
  }
}

int main()
{
  cin >> n;
  rec(1);
  cout << res << endl;
  return 0;
}
```

7339 Послідовність

Знайти n-й член послідовності 1 10 11 100 101 110 111 1000 ...
Вхідні дані: Одне натуральне число n (n ≤ 10000).
Вихідні дані: n-й член послідовності.

Приклад введення	Приклад виведення
24	1100

Послідовність, що задана в задачі – натуральні числа, записані у двійковій системі числення. Нам потрібно перевести число n у двійкову систему числення. Для переведення числа у двійкову

систему числення нам потрібно знаходити остачу від ділення на 2, потім ділити число на 2. Всі остачі, записані у зворотному порядку, утворять число у двійковій системі числення.

$\mathbf{24}_{10} = \mathbf{1100}_{2}$

Є два способи реалізації цієї програми.

Перший – за допомогою циклу, поки число більше 0, будемо додавати до результату на початок рядка остачу від ділення на 2. Далі ділити число на 2. Запис `n & 1` (важливо, що амперсанд один!) повертає 1, якщо число непарне, і 0, коли парне. Коли число використовується як умова, 1 трактується як істина, 0 – як хиба. `n >>= 1` – побітовий зсув числа n на 1 праворуч, що означає ділення числа на 2.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int n; string s = «»;
  cin >> n;
  while (n > 0)
  {
    if (n & 1) s = «1» + s;
    else       s = «0» + s;
    n >>= 1;
  }
  cout << s << endl;
  return 0;
}
```

Другий спосіб реалізований за допомогою рекурсивної функції. Виведення остач від ділення 2 відбувається під час виходу з рекурсивної функції.

```
#include <bits/stdc++.h>
using namespace std;

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

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

7371 ЗНО (Testing)

В цьому році Андрію доведеться складати випробування із ЗНО, у тому числі з математики. Він наполегливо готується – вже розв'язує складні тригонометричні рівняння, легко вправляється із стереометрією, а ось з раціональними числами (дробами) справи кепські, особливо з їх додаванням. Він навіть боїться припуститися помилки у написанні перевірочної програми. Зробіть це замість нього. Нагадаємо, що раціональне число визначається, як нескоротний дріб, у якого чисельник – ціле число, а знаменник – натуральне число.

Вхідні дані: Чотири цілих числа, які відповідають умові задачі – чисельник та знаменник першого числа і чисельник та знаменник другого числа. Вхідні дані за модулем не перевищують 10^5.

Вихідні дані: Вивести чисельник та знаменник суми дробів.

Приклад введення	Приклад виведення
4 9 5 9	73 63
2 3 -1 6	1 2

Змоделюємо процес додавання звичайних дробів за формулою $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$, а потім скоротимо результат.

Для скорочення дробів використаємо функцію **nsd**. Так як в чисельнику можуть бути від'ємні числа, у функції **nsd** візьмемо модуль.

```
#include <bits/stdc++.h>
using namespace std;

long long nsd(long long a, long long b)
{
  a = abs(a); b = abs(b);
  while (a != 0 && b != 0)
    if(a > b)
      a %= b;
    else
      b %= a;
  return a + b;
}
```

```
int main()
{
  long long a, b, c, d, zn, ch, nSd;
  cin >> a >> b >> c >> d;
  ch = a*d + b*c;
  zn = b*d;
  nSd = nsd(ch, zn);
  ch = ch / nSd;
  zn = zn / nSd;
  cout << ch << ' ' << zn << endl;
  return 0;
}
```

Ірина Скляр. Обробка текстових даних

Дуже велика кількість задач, що розв'язуються, вимагають обробки не тільки числових, а й текстових даних. Частина з них використовують тексти тільки для організації інтерфейсу з користувачем, але є й такі, що повинні обробляти тексти за деякими алгоритмами, редагувати їх та зберігати. Отже, наша задача розібрати, що являють собою тексти (рядки) у мові програмування C++.

Традиційно *рядком* називають послідовність символів. Найпростішим прикладом рядка є текстова константа, укладена у подвійні лапки, наприклад, **"Привіт!"**. Лапки використовуються для визначення початку та кінця рядка, але до його складу не відносяться. Символьні константи досить часто використовуються при виведенні відповіді розв'язаної задачі. Однак вони не дозволяють роботу зі змінними рядками.

У мові C++ доступні два способи роботи з рядками-змінними. Перший, успадкований від мови C, найчастіше називається *рядками у стилі C* і фактично є послідовністю символів, збереженою у масиві типу **char**. Другий – оснований на використанні вбудованого класу **string**. Розглянемо спочатку перший з них.

Рядки у стилі C

Рядки у стилі C мають спеціальну характеристику:останнім символом кожного рядка є так званий «нуль-термінатор» (позначається **«\0»**) – символ ASCII-таблиці з кодом 0. Розглянемо такі два оголошення:

```
charfirst[6] = {'H','e','l','l','o','!'};
charsecond[6] = {'H','e','l','l','o','\0'};
```

Перший з наведених прикладів рядком не являється, а є звичайним масивом символів і тому оброблятися буде тільки поелементно. Другий є рядком, оскільки містить наприкінці нуль-термінатор. Цей символ не тільки є ознакою рядка, а й відіграє значну роль у його обробці. Мова C++ має велику кількість стандартних функцій для роботи з рядками, які використовують нульовий символ як ознаку завершення рядка. У випадку, якщо ці функції не знаходять у масиві нуль-символу, може трапитися ситуація виходу за межі оголошеного символьного масиву з усіма негативними наслідками.

При оголошенні рядкового масиву потрібно завжди враховувати наявність термінатора у кінці рядка і відводити для цього додатковий байт. Наприклад, при описі `charstr[10];` у рядку може міститися тільки 9 символів.Не обов'язково при оголошенні ініціалізовуватирядок значеннями поелементно. Набагато зручніше використовувати для цього рядкові константи або рядкові літерали – послідовності символів, укладені у подвійні лапки, наприклад:

```
charstr[14]="Hello, world!";
```

Тоді у кожному елементісимвольного масиву буде по одному символу (в цьому прикладі їх усього 13) і на останній позиції буде нуль-термінатор, який компілятор добавить у кінець рядка самостійно.

При оголошенні рядка не обов'язково вказувати його розмір, але при цьому обов'язково потрібно ініціалізуватирядок початковим значенням. Тоді розмір рядка визначиться автоматично і в його кінець буде додано нуль-термінатор. Наприклад:

```
charstr[]='Hello, world!';
```

Рядок може містити будь-які символи ASCII таблиці, укладенів подвійні лапки. Ім'я рядка є константним вказівником на його перший символ. Можна при оголошенні вказати розмір рядка, не ініціалізуючи його початковим значенням. Але у такому випадку потрібно резервувати розмір масиву з запасом: елементи після нуль-термінатора все одно не враховуються. Обмеження на розмір рядка у мові C++ співпадають з обмеженнями на розмір масиву.

Розглянемо тепер основні принципи обробки рядків у стилі C. Оскільки єдиний спосіб отримати різні рядки на початку програми це увести їх зі вхідного потоку, почнемо саме з уведення. Для цього можна використовувати звичний для нас спосіб:

```
cin>>str;
```

Однак при цьому потрібно враховувати, що якщо вхідний потік буде містити пробіли, зчитування відбудеться до першого пробілу. Справа в тому, що оскільки нуль-символ увести з вхідного потоку неможливо, розділювачами між елементами даних будуть вважати-

ся стандартні: пробіл, символ табуляції або символ завершення рядка. Таким чином, **cin** автоматично зчитує тільки одне слово і додає до нього в кінці обмежувальний нуль-символ. Після цього отримана послідовність символів записується у масив.

Читання рядка по одному слову часто є небажаним. Так, навіть фразу **Hello, world!** не можна прочитати одномоментно. Щоб читати речення цілком, потрібен інший метод. Для цього в класі **istream**, екземпляром якого є **cin**, існують функції-члени, призначені для рядково-орієнтованого введення: **getline()** та **get()**. Обидві читають повний рядок уведення аж до символу нового рядка. Однак **getline()** потім відкидає символ нового рядка, а **get()** залишає його у вхідній черзі. Розглянемо це детально.

Нехай ми маємо такий опис рядка:

```
char Name[20];
```

Як було зазначено вище, функція **getline()** читає весь рядок, використовуючи символ нового рядка, який передається клавішею <*Enter*>, для визначення завершення уведення. Цей метод ініціюється викликом функції **cin.getline()** з двома аргументами: перший аргумент – ім'я масиву символів, куди відбувається зчитування, і другий – максимальна кількість символів, що може бути прочитана. Наприклад, **cin.getline(Name,20);**

За такого виклику з вхідного потоку буде прочитаний рядок, що містить не більше 19 символів (плюс один останній символ автоматично буде встановлений у нуль-термінатор). Якщо вхідний потік буде містити менше символів, завершення зчитування виконається раніше, Якщо ж вхідний потік містить більше 19 символів, уведення не буде завершене. Фактично функція-член **getline()** завершує читання, коли досягає вказаної межі кількості символів або коли читає символ нового рядка – дивлячись, що трапиться раніше.

Функція-член **get()** може бути викликана в такий самий спосіб, що й попередня, тобто **cin.get(Name,20);** У такому випадку її аргументи інтерпретуються таким самим чином, як і при виклику функції **getline()**. Однак символ нового рядку прочитано не буде і він залишиться у вхідному потоці. Повторний виклик цього самого методу не дозволить зрушитися у вхідному потоці, і програма фактично «зависне» наприкінці прочитаного рядка. Вирішенням проблеми у такому випадку є використання функції **get()** без аргу-

ментів. Такий виклик читає один будь-який символ, навіть якщо це символ нового рядка.

Виникає питання, навіщо взагалі користуватися функцією `get()`, якщо більш зручною є функція `getline()`? Справа в тому, що `get()` дозволяє бути більш обережним при зчитуванні. Наприклад, при зчитуванні функцією getline() неможливо перевірити, зчитано весь рядок чи просто читання перервалось у зв'язку із завершенням масиву. Для цього потрібно перевірити наступний символ потоку. Якщо це символ нового рядку, прочитано все, у протилежному випадку ще не все. Таким чином, при зчитуванні з вхідного потоку рядку, що перевищує розмір виділеного рядку, треба бути обережним, аби не отримати неочікувані результати.

Виникає також питання, що відбудеться при зчитуванні пустого рядка? Зверніть увагу, що дія вказаних функцій суттєво відрізняється. Після того, як функція `get()` прочитає пустий рядок, вона встановлює прапорець, який називається `failbit`. Вплив цього прапорця полягає в тому, що наступне уведення блокується і для його відновлення потрібно скористатися командою `cin.clear()`.

При використанні функції `getline()` для зчитування пустого рядка такого не відбувається і тому уведення продовжується без змін. Однак прапорець `failbit` встановлюється, якщо цією функцією намагаються зчитати рядок, що має довжину, більшу ніж масив, у який відбувається зчитування. Функція `get()` у цьому випадку продовжує зчитування.

Проблеми виникають також, якщо змішувати уведення чисел та рядків. Справа у тому, що при зчитуванні числа, що розташовано у окремому рядку вхідного потоку, символ нового рядка, який було згенеровано натисканням клавіші <*Enter*>, залишається у вхідній черзі. Щоб потім прочитати рядок, потрібно спочатку прочитати та відкинути символ нового рядку як пустий рядок. Це можна зробити викликом `cin.get()`, наприклад, так:

```
int N;
cin>>N;
cin.get();
```

У мові C++ також можна використати так зване зчеплення, скориставшись тим фактом, що вираз `cin>>N` повертає об'єкт `cin`:

```
(cin >> N).get();
```

Якщо потрібно обробити багаторядковий текст із вхідного потоку, також можна скористатися фактом, що зчитування з **cin** повертає об'єкт **cin**. Це приводить до того, що коли вхідний потік завершиться, **cin** стане пустим об'єктом і цикл **while** отримає хибну умову (відсутність об'єкту рівносильна хибній умові):

```
// Визначення максимально можливої довжини рядку
const int len=1001;
char str[len]; // Оголошення рядку
while(cin>>str) // Зчитування одного слова рядку
{
  // Обробка рядка
}
```

Якщо зчитування відбувається з файлу, кінець файлу сприймається як закінчення вхідного потоку. Якщо ж зчитується рядок з консолі, ознакою завершення вхідного потоку є одночасне натискання клавіш *<Ctrl+z>*.

Аналогічно можна читати і цілими рядками:

```
// Зчитування цілими рядками
while (cin.getline(str,len))
{
  // Обробка рядка
}
```

Мова C++ має досить велику кількість стандартних функцій для обробки рядків. Нижче наведений описдеяких з них.

Назва функції	*Пояснення*
`strlen(ім'я_рядку)`	Визначає довжину вказаного рядка без урахування нуль-символу.
`strcpy(рядок_1, рядок_2)`	Виконується побайтове копіювання символів з **рядка_2** у **рядок_1**.
`strncpy(рядок_1, рядок_2, кількість_символів)`	Побайтово копіює кількість_символів зі змінної **рядок_2** у змінну **рядок_1**.
`strcmp(рядок_1, рядок_2)`	Порівнює вказані рядки з урахуванням регістру. Повертає ціле число: 0, якщо рядки еквівалентні; >0, якщо **рядок_1>рядок_2**; <0, якщо **рядок_1>рядок_2**.

`strncmp(рядок_1, рядок_2, кількість_символів)`	Аналогічно попередньому випадку порівнює задану **`кількість_символів`** двох рядків. Значення, що повертаються, аналогічні.
`strcat(рядок_1, рядок_2)`	Конкатенація (об'єднання) першого та другого рядків. Результат вміщується в **`рядок_1`**. В кінці модифікованого першого рядка встановлюється нуль-символ, а нуль-символ, що раніше завершував перший рядок, заміщується першим символом **`рядка_2`**, вміст якого не змінюється. Зверніть увагу, що програміст повинен самостійно відслідковувати, щоб перший рядок мав достатню довжину для вмісту результату конкатенації.
`strncat(рядок_1, рядок_2, кількість_символів)`	Функція працює аналогічно функції **`strcat`**, але з другого рядку береться не більше символів, ніж задано аргументом кількість_символів.
`strupr(рядок)`	Функція перетворює усі символи рядка у символи верхнього регістру. Символи, що не можна перетворити у символи верхнього регістру, залишаються без змін.
`strchr(рядок, символ)`	Пошук першого входження заданого **`символу`** в рядок. У випадку вдалого пошуку функція повертає вказівник на місце першого входження. Якщо символ не знайдено, функція повертає нуль.
`strcspn(рядок_1, рядок_2)`	Визначає довжину початкового сегменту **`рядку_1`**, що містить символи, яких немає у **`рядку_2`**.
`strspn(рядок_1, рядок_2)`	Визначає довжину початкового сегменту **`рядку_1`**, що містить тільки ті символи, які вказані у **`рядку_2`**.

`strprbk(рядок_1, рядок_2)`	Повертає вказівник першого входження будь-якого символу з **`рядка_2`** у **`рядок_1`**.
`atof(рядок)`	Перетворює рядок-аргумент у дійсне число типу **`double`**, яке повертається як значення функції. Якщо перетворення неможливе, повертається значення 0.
`atoi(рядок)`	Перетворює рядок-аргумент у ціле число типу **`int`**, яке повертається як значення функції. Якщо перетворення неможливе, повертається значення 0.
`atoll(рядок)`	Перетворює рядок-аргумент у ціле число типу **`long long`**, яке повертається як значення функції. Якщо перетворення неможливе, повертається значення 0.
`gets(рядок)`	Зчитує потік символів зі стандартного пристрою уведення у **`рядок`**, доки не буде натиснута клавіша *<Enter>*.

У процесі обробки рядків можуть знадобитися функції для роботи з окремими символами.

Назва функції	*Пояснення*
`isalnum(символ)`	Повертає значення **`true`**, якщо аргумент є буквою або цифрою, та **`false`** у протилежному випадку.
`isalpha(символ)`	Повертає значення **`true`**, якщо аргумент є буквою, та **`false`** у протилежному випадку.
`isdigit(символ)`	Повертає значення **`true`**, якщо аргумент є цифрою, та **`false`** у протилежному випадку.
`isxdigit(символ)`	Повертає значення **`true`**, якщо аргумент є шістнадцятковою цифрою та **`false`** у протилежному випадку.
`islower(символ)`	Повертає значення **`true`**, якщо аргумент є буквою нижнього регістру та **`false`** у протилежному випадку.

`isupper(символ)`	Повертає значення **true**, якщо аргумент є буквою верхнього регістру, та **false** у протилежному випадку.
`isspace(символ)`	Повертає значення **true**, якщо аргумент є пропуском (пробілом, табуляцією чи переведенням рядка) та **false** у протилежному випадку.
`ispunct(символ)`	Повертає значення **true**, якщо аргумент є символом пунктуації, та **false** у протилежному випадку.
`toupper(символ)`	Якщо аргумент є маленькою буквою, функція повертає його, як велику букву, і залишає без змін у протилежному випадку.
`tolower(символ)`	Якщо аргумент є великою буквою, функція повертає його, як маленьку букву,і залишає без змін у протилежному випадку.

Стандартно, функції **isalnum**, **isalpha**, **islower**, **isupper**, **toupper** та **tolower** вважають буквами лише англійські букви. В деяких версіях C++ є бібліотеки, що надають можливість робити аналогічні дії з буквами інших алфавітів, але ми зараз це не розглядатимемо.

Розглянемо тепер кілька задач по обробці рядків у стилі C. Спочатку розглянемо приклад задачі, яка використовує особливості зчитування рядків за допомогою вхідного потоку **cin**. Нагадуємо, що без використання метода **getline** зчитування рядків з потоку виконується до розділювача, тобто текст буде зчитуватися стандартними словами (словом вважається послідовність будь-яких символів між пробілами, табуляціями або символами завершення рядка). Звідси випливає розв'язок такої задачі.

Приклад 1. Дано рядок. Вивести у вихідний потік новий рядок, утворений вилученням з початкового рядкаслів, що мають довжину менше трьох символів.

Ідея розв'язання. Якщо зчитувати текст по одному слову, розв'язання зведеться до виведення тільки тих слів, що мають довжину більше двох символів. Тобто програма буде мати вигляд:

```
int main()
{
  const int Len=1000;
  char str[Len];
  while (cin>>str)
  {
    if (strlen(str)>2) cout<<str<<' ';
  }
  return 0;
}
```

Аналогічно можна знайти довжину найдовшого слова. У цьому випадку потрібно запам'ятовувати довжину того слова, яке має більше символів, ніж усі попередні. Для цього уводиться поняття еталону – змінної, що зберігає довжину слова, довшого, ніж попередній еталон.

```
int main()
{
  const int Len=1000;
  char str[Len];
  int etalon=0; // Слів довжини менше 0 не буває
  while (cin>>str)
  {
    // Якщо знайдено слово більшої довжини,
    // etalon переприсвоюється
    if (strlen(str)>etalon)
      etalon=strlen(str);
  }
  cout<<etalon<<endl;
  return 0;
}
```

Дещо схоже виконується пошук самого найдовшого слова: просто у цьому випадку еталоном є саме слово, а не його довжина.

```
int main()
{
  const int Len=1000;
  char str[Len];
  char etalon[Len]; // Тепер еталоном є саме слово
  cin>>etalon;
  // Перше слово зчитується в рядок etalon.
```

```
  while (cin>>str)
    if (strlen(str)>strlen(etalon))
      strcpy(etalon,str);
  cout<<etalon<<endl;
  return 0;
}
```

Тепер розглянемо більш складні задачі, які для свого розв'язання вимагають використання функцій, як стандартних, так і користувацьких.

Оскільки методи обробки багаторядкового тексту не відрізняються від обробки одного рядка, будемо пропонувати розв'язки для одного рядка. Основна програма, яка забезпечує безпосереднє уведення багаторядкового тексту буде мати приблизно такий вигляд:

```
int main()
{
  const int len=1000;
  char str[len];
  while (cin.getline(str,len))
  {
    // Обробка уведеного рядка
  }
  // Виведення результату
  return 0;
}
```

Приклад 2. Знайти в рядку кількість заданих символів. Символ уводиться у першому рядку вхідного потоку.

Ідея розв'язання. Щоб прочитати один символ з вхідного потоку, використаємо метод `get()` у такий спосіб:

```
char ch;
cin.get(ch).get();
```

Оформимо розв'язок задачі функцією, аргументами якої буде заданий символ та даний рядок:

```
int count_char(char ch, char str[])
{
  int cnt=0;
  for (int i=0; i<strlen(str); i++)
    if (toupper(str[i])==toupper(ch))
      cnt++;
  return cnt;
}
```

Зверніть увагу, що в умові не вказано, які – великі чи маленькі – букви потрібно шукати. Як правило, в нашій свідомості великі та маленькі букви є однаковими, а тому функція, передбачаючи це, перетворює і шуканий символ, і символи рядка у великі, а тільки потім порівнює.

Тепер ускладнимо задачу і спробуємо знайти не один символ, а підрядок. Підхід до розв'язання цієї задачі схожий: потрібно порівнювати символи рядків з урахування регістру. Але тепер одного порівняння недостатньо – доведеться посимвольно порівнювати два рядки. Для цього потрібно запустити вкладений цикл, що завершить свою роботу у випадку закінчення еталонного рядку або знаходження відмінного символу.

```
int count_fragment(char etalon[], char S[])
{
  intcnt=0;
  int LenS=strlen(S), LenEtalon=strlen(etalon);
  for (int i=0; i<=LenS-LenEtalon; i++)
  {
    int j=0;
    while (j<LenEtalon && toupper(S[i+j])==toupper(etalon[j]))
      j++;
    if (j==LenEtalon) cnt++;
  }
  return cnt;
}
```

Зверніть увагу на умову роботи першого циклу: вона передбачає аналіз виходу за межі рядку, що перевіряється, а крім того ще й уникає багатократних вимірювань довжин рядків.

Взагалі кажучи, функція **strlen** працює дещо повільно, справді шукаючи кінець рядка шляхом його перегляду. Коли треба працювати зі справді довгими рядками (як-то сотні тисяч чи мільйони символів), вимірювання довжини через **strlen(str)** надто повільне, щоб робити його на кожній ітерації. Якщо зберегти довжину рядка в окрему змінну, цих затримок можна уникнути.

Ще один спосіб – переписати цикл проходження по рядку так:

```
for (int i = 0; str[i]; i++) …
```

(тобто, замість «**i < strlen(str)**» вжито «**str[i]**»). Це працює завдяки нуль-термінатору наприкінці рядка у стилі C та ставленню

до 0 як до `false`, а до ненулів як до `true`. Так і виходить: поки є символи (ASCII-код ненульовий), цикл продовжується; коли настає кінець (нуль-термінатор), цикл припиняється.

Приклад 3. Дано текст. Потрібно «стиснути» його, вилучивши усі подвоєні символи. Наприклад, з тексту `"Classaasskk"` потрібно отримати текст `"Clasask"`.

Ідея розв'язання. Вилучення елементів з символьного масиву (як, між іншим, і з будь-якого іншого)неможливо робити безпосередньо (можна лише зі «зсувами хвоста»). Тому пропонуємо зарезервувати допоміжний рядок, у який переписувати тільки ті символи, які не мають подвоєння (тобто два сусідніх символи не однакові). Функція, що виконує описаний алгоритм, буде мати вигляд:

```
void rewrite(char S[],char answer[])
{
  // Перший символ завжди потрапляє
  // у результуючий рядок
  answer[0]=S[0];
  // Початок індексації результуючого рядка
  int j=1;
  // Переписування у результуючий рядок без подвоєння
  for (int i=1; i<strlen(S);i++)
    if (S[i]!=S[i-1]){
      answer[j]=S[i];
      j++;
    }
    //Внесення завершального символу рядка
    answer[j]='\0';
}
```

Приклад 4. Виконати заміну заданого символу на інший, причому якщо знайдений символ – велика буква, замінити його потрібно на велику (незалежно від регістру введених символів), якщо ж маленька – на маленьку (знову ж таки незалежно від регістру уведених символів). Наприклад, якщо заданий символ «`A`», а шуканий «`e`», то у тексті `Amma` перший символ «`A`» потрібно замінити на велику букву «`E`», а другий – на маленьку букву «`e`». Тобто, результатом заміни буде слово `Emme`.

Ідея розв'язання. При розв'язанні цієї задачі має сенс використати не тільки функції визначення регістру, а й переведення у зада-

ний регістр окремих символів. Функція, що виконує задані заміни, має такий вигляд:

```
void change(char a, char b, char str[])
{
  for (int i=0;i<strlen(str); i++)
    // Перевірка на збіг символів незалежно від регістру
    if (toupper(str[i])==toupper(a)) {
      if (isupper(str[i]))
        str[i]=toupper(b);
        // Заміна на велику букву
      else
        str[i]=tolower(b); // на малу
    }
}
```

Аналогічно виконується заміна одного підрядка іншим. Домовимося тільки у цьому випадку при пошуку потрібного слова відслідковувати регістр усіх букв, а вставляти вже у тому регістрі, який задано. Наступна функція реалізує описаний підхід:

```
void change_fragment(char first[],char second[],char S[])
{
  int index=0, i=0, LenS=strlen(S);
  int LenFirst=strlen(first);
  char answer[LenS];
  // Перший рядок переводимо у верхній регістр
  // для підвищенняшвидкодії
  strupr(first);
  // Обробка заданого рядка в межах, що може
  // містити рядок first
  while (i <=LenS-LenFirst)
  {
    // Пошук першого рядку у другому
    int j=0;
    while(j<LenFirst && toupper(S[i+j])==first[j])
      j++;
    if (j==LenFirst)
    {
      // Якщо перший рядок first знайдено,
      // переписуємодругий рядок
      // у результуючий масив
      for (int j=0; second[j]; j++,index++)
        answer[index]=second[j];
      // Пропускаємо символи першого рядка
      i+=LenFirst;
    }
```

```
      else
      {
        // Символи, що не утворюють перший рядок,
        // просто переписуємо у результуючий рядок
        answer[index]=S[i];
        index++;i++;
      }
    }
    // Переписування необробленого «хвоста»
    // у результуючий рядок
    while (i<LenS)
    {
      answer[index]=S[i];
      index++; i++;
    }
    // Копіювання результуючого рядку в початковий
    strcpy(S,answer);
  }
```

Як бачите, розв'язок вийшов досить громіздкий. Знову-таки це пов'язано з тим, що у символьному масиві заборонені не тільки операції видалення його частини, а й вставлення іншого символьного масиву у даний. Тому у запропонованому розв'язку знову уведено поняття результуючого рядка, який заповнюється таким чином:

- Якщо шуканий рядок **first** знайдено (в будь-якому регістрі), у результуючий рядок переписується посимвольно другий рядок **second**, після чого у заданому рядку пропускається відповідна кількість символів.
- Інакше (символ заданого рядка відрізняється від першого символу пошукового рядка **first**, чи наступний символ заданого рядка відрізняється від другого символу пошукового рядка **first**, і т. д.), один поточний символ заданого рядка переписується у результуючий символьний масив **answer** без змін.

Після завершення перегляду даного рядка в межах, де може бути присутнім шуканий рядок, здійснюється посимвольне переписування необробленого «хвоста» з першого у результуючий рядок. Крім того, у даній функції реалізується ідеологія зміни початкового рядка, а не створення нового. Тому наприкінці функції утворений результуючий рядок **answer** копіюється стандартною функцією **strcpy** у початковий рядок **S**.

Приклад 5. Перевірити, чи є заданий рядок ідентифікатором, тобто містить тільки англійські букви будь-якого регістру, символ

підкреслення або цифри, та починається не з цифри. Наприклад, слова `class`, `x1`, `X_2` та `_9clas` є ідентифікаторами, а слова `11_A`, `13_septembers` та `15-` ні.

Ідея розв'язання. Цю задачу можна розв'язати двома методами:

- посимвольно перевірити усі символи на «правильність» (тільки дозволені для використання символи);
- скористатися стандартними функціями для роботи з рядками.

Перший символ в обох методах перевіряється одним виразом за допомогою стандартної функції `isdigit(S[0])`, яка повертає значення `true`, якщо символ є цифрою.

Напишемо функцію, яка у випадку входження символу, що не є дозволеним (англійською буквою, цифрою або знаком підкреслення), буде повертати значення `false`. Крім того, `false` повертатиметься у випадку, коли перший символ – цифра або довжина рядка дорівнює 0 (рядок пустий). Функція має такий вигляд:

```
bool isIdentifier(char S[])
{
  // Перевірка довжини рядка
  if (strlen(S)==0) return false;
  // Перевірка першого символу
  if (isdigit(S[0])) return false;
  int i=0;
  while (i<strlen(S) && (S[i]=='_' || isalnum(S[i])))
    i++;
  // Якщо не знайдено жодного небажаного символу
  // до завершення рядка, повертається значення true
  return (i==strlen(S));
}
```

Приклад 6. Задано текст, що може містити натуральні десяткові числа. Знайти суму всіх чисел, що є у тексті. Наприклад, якщо текст містить послідовність символів `"13jhffgefje 5 23 hjdhfs 123"`, результат повинен бути рівним 164.

Ідея розв'язання. Пропонуємо такий алгоритм: рухаючись по рядку, вибираємо з нього в інший рядок всі цифри, розташовані підряд. Перетворюємо отриманий рядок (фактично це багатоцифрове число) у ціле число стандартною функцією `atoll`. На наш погляд, зручно оформити функцію, яка спочатку буде пропускати усі сим-

воли, що не є цифрами, а вже потім вибирати всі знайдені цифри у інший рядок. Враховуючи, що тип `long long` не може мати більше 19 цифр, допоміжний рядок зарезервуємо на 20 символів. Функція, що реалізовує знаходження чергового числа, має такий вигляд:

```
long long str_to_int(char S[], int &pos)
{
  // Пропускаємо нецифрові символи рядку
  while (pos<strlen(S) && !isdigit(S[pos])) pos++;
  char temp[20]; int i=0;
  // Вибираємо цифрові символи у допоміжний рядок
  while (pos<strlen(S) && isdigit(S[pos]))
  {
    temp[i]=S[pos];
    pos++; i++;
  }
  // Вносимов рядок завершальний символ
  temp[i]='\0';
  // Конвертуємо рядок у ціле число (longlong)
  return atoll(temp);
}
```

Зверніть увагу, що аргументами цієї функції є заданий рядок та поточний індекс (позиція розглядуваного символу в рядку). Ця функція приймає другий аргумент за посиланням, щоб після знаходження чергового числа його значення було рівним індексу наступного після знайденого числа символу рядка (й це було доступно в тому місці коду, звідки викликали цю функцію).

Наведемо також основну програму для даного прикладу.

```
int main()
{
  const int len=1000;
  char str[len];
  long long res=0;
  while(cin.getline(str,len))
  {
    int i=0;
    while(i<strlen(str))
    {
      // Знаходження чергового числа та
      // додавання його до результату
      res+=str_to_int(str,i);
    }
  }
```

```
  cout<<res<<endl;
  return 0;
}
```

Взагалі для інших прикладів основна програма буде мати схожий вигляд. Але пропонуємо вам самостійно її модифікувати під кожну задачу.

Приклад 7. Дано рядок. Перевернути його. Наприклад, з рядку abcdef потрібно отримати fedcba.

Ідея розв'язання. Очевидно, що для отримання бажаного результату потрібно поміняти місцями перший символ рядку з останнім, другий – з передостаннім, третій – з передпередостаннім і так далі. Всього буде зроблено $\frac{Len}{2}$ замін, де Len – довжина рядка. Цикл, що виконує описаний алгоритм, має такий вигляд:

```
int Len=strlen(S);
for (int i=0; i<Len/2; i++)
  swap(S[i],S[Len-i-1]);
```

Однак, оскільки дана задача може бути підзадачею інших задач, узагальнимо її розв'язок і напишемо функцію, що перевертає не весь рядок, а його частину від елементу з індексом **Left** до елементу з індексом **Right**:

```
void reverse(char S[], int Left, int Right)
{
  //Кількість потрібних перемін
  int Len=(Right-Left)/2;
  for (int i= 0; i<=Len; i++)
    swap(S[Left+i],S[Right-i]);
}
```

Тепер за допомогою цієї функції можна перевернути весь рядок, виконавши виклик **reverse(str,0,strlen(str)-1)**, або його частину. Перевернутий рядок можна використовувати для перевірки, чи є заданий рядок паліндромом (читається в обох напрямках однаково, наприклад, **qwerewq**). Порівняння початкового та перевернутого рядків можна виконати за допомогою ще однієї стандартної функції **strcmp**.

Ще одне дуже цікаве застосування цієї функції можна запропонувати для розв'язання такої задачі. Дано рядок. Потрібно у

ньому поміняти місцями два задані фрагменти (можливо, різної довжини). Фрагменти задаються своїми лівими та правими індексами. Наприклад, якщо для рядку **absdefghijklmnop** потрібно поміняти місцями фрагменти з 2-го символу по 4-й та з 7-го по 11-й (нумерація скрізь з 0), отримаємо такий результат. Початковий рядок має вигляд:

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Нижній рядок таблиці являє собою номери символів рядку.

Після вказаної переміни місцями ми повинні отримати такий рядок:

a	**b**	**h**	**i**	**j**	**k**	**l**	**f**	**g**	**c**	**d**	**e**	**m**	**n**	**o**	**p**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Зверніть увагу, що з лівої та правої сторін рядку символи свого місцеположення не змінили.

Перша ідея, яка спадає на думку, це використання допоміжного рядка. Але це потребує додаткових витрат пам'яті, що при великих розмірах рядків нераціонально. Пропонуємо інший варіант: кілька разів перевернути частини масиву:

1. Перевернути кожен із заданих фрагментів та фрагмент між ними (кожен окремо). В результаті для наведеного прикладу отримаємо такий рядок **abedcgflkjihmnop** (перший та другий перевернуті фрагменти підкреслені).

2. Тепер перевернути всючастину рядка цілком від початку першого фрагмента до кінця другого. Отримаємо **abhijklfgcdemnop**, тобто задача розв'язана.

Фрагмент програми, що розв'язує вказану задачу цим способом:

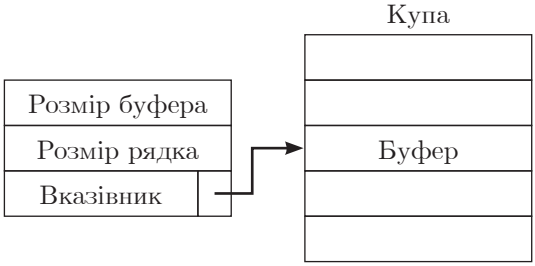
```
reverse(S,left1,right1);
reverse(S,right1+1,left2-1);
reverse(S,left2,right2);
reverse(S,left1,right2);
```

Тут **left1**, **right1** – індекси першого фрагмента, а **left2**, **right2** – індекси другого фрагмента рядка (**left1<right1<left2<right2**).

Клас string

Другим методом роботи з рядками є використання так званих рядків `basic_string<...>`. Фактично, `basic_string<...>` – це сховище для фізичного подання символів, що можуть зберігатися у різних кодуваннях. Інваріантом `basic_string<...>` є клас `string`, який дозволяє зберігати символи ASCII-кодування (розміром в 1 байт), тобто послідовність символів типу `char`.

Реалізацію класу у стандартній бібліотеці можна подати так:

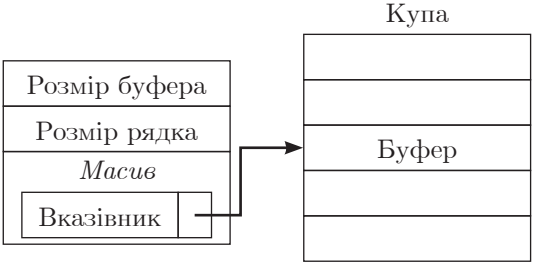


Як видно, сам рядок розміщений у вільній оперативній пам'яті комп'ютера (купі), а клас зберігає, по-перше, вказівник на область пам'яті у купі, та два поля, що відповідають за розмір. Виникає питання: навіщо два поля розміру? Справа в тому, що розмір буфера, який виділений для збереження рядка, та безпосередньо довжина рядка можуть відрізнятися. Розмір рядка (кількість символів, яку він фактично містить) зберігати потрібно для того, щоб під час обробки не витрачати час на виконання достатньо розповсюдженого запиту на довжину рядка. А ось буфер, більший самого рядка, потрібен для оптимізації роботи програми. Пояснимо, навіщо робити буфер більшим рядка, що оброблюється.

Справа в тому, що однією з часто використовуваних операцій при обробці рядків є операції додавання до нього нових символів або «зчеплення» одного рядказ іншим. Замість того, аби кожен раз виділяти пам'ять під нові символи та переміщувати символи при кожній зміні розміру рядка, реалізація класу `string` має «розумну» систему виділення пам'яті більшого розміру, ніж рядок, який потрібно зберігати. У випадку, коли залишки пам'яті підходять до кінця і потрібне нове виділення, новий буфер робиться знову більше, ніж потрібно для приєднання іншого рядка. При такому підході, хоч і збільшується загальний розмір рядка, зате значно зменшу-

ються кількості запитів на виділення пам'яті та переносів символів. Врешті-решт це досить суттєво підвищує швидкість обробки рядків.

Аналіз програм з рядками показує, що більшість оброблених рядків мають доволі малий розмір. А тому розробники бібліотеки використали такий прийом: якщо рядок повністю вміщується у масив, додатковий буфер у купі не виділяється і рядок зберігається у самому об'єкті **string**. Якщо ж довжина рядка перевищує деяке задане число, то у купі виділяється буфер і вказівник стає зв'язуючою ланкою між об'єктом **string** та буфером. Відповідно, сучасна реалізація класу **string** ще більш оптимізована за часом обробки. Він містить, крім вже вказаних частин, ще й статичний масив символів, тобто його реалізація така:



Об'єкт класу **string** можна створити, використовуючи різні види перевантаженого конструктора:

- **string one("Hello, world!");** – створюється об'єкт, ініціалізований значенням **"Hello, world!"**;
- **string two(20, '$');** – створюється об'єкт, ініціалізований значенням, що містить 20 однакових символів **'$'**;
- **char alls[] = "All's well that ends well";**
stringt hree(alls, 20); – створюється об'єкт, ініціалізований першими двадцятьма символами символьного рядку типу C;
- **string four(three);** – створюється об'єкт, ініціалізований об'єктом **three** такого самого класу **string**;
- **string five(three, 7, 16);** – створюється об'єкт, ініціалізований шістнадцятьма символами об'єкту **three** класу **string**, починаючи з сьомого.

А можна описати об'єкт класу **string**, використовуючи звичний метод:

```
string first = "All's well that ends well";
```

Можна також при описі рядку не ініціалізувати його початковим значенням взагалі:

```
string first; // Опис без ініціалізації
```

При описі об'єкта класу **string** без ініціалізації гарантовано, що він набуде значення порожнього рядка, тобто жодної літери. (Всупереч тому, що для примітивних типів, а також для рядків у стилі C, опис без ініціалізації нічого не гарантує. Зокрема, не ініціалізоване число не зобов'язане дорівнювати 0, його значення може залежати від багатьох несподіваних причин. А для класу **string** гарантія є.)

В усіх випадках при описі розмір об'єкту **string** не вказується, оскільки цей рядок є динамічним (його розмір змінюється під час роботи), а об'єкт містить посилання на область пам'яті у купі.

Для об'єктів класу **string** перевантажено багато звичних для нас операцій. Так, рядки можна присвоювати один одному звичайною командою присвоєння:

```
string first, second;
second = first;
```

Можна також об'єкту класу string присвоїти рядок у стилі C:

```
first = "Hello, world!";
```

Над рядками типу string набагато простіше виконувати більш складні операції. Так, рядки можна «зчеплювати» за допомогою звичайної операції «+», наприклад:

```
first = first + second;
```

У результаті рядок **first** буде містити символи обох рядків, причому, не потрібно слідкувати за довжиною результуючого рядка: оскільки об'єкти **string** є динамічними, система автоматично збільшує їх розмір до потрібного.

Для об'єктів класу **string** перевантажена також операція «**+=**». Так, попередня команда присвоєння може бути записана в такий спосіб:

```
first += second;
```

Результатом її роботи також буде рядок, що містить символи обох вказаних рядків.

Для введення та виведення рядків можна використовувати об'єкти **cin** та **cout**, але, як і раніше, зчитуватися буде послідовність символів до стандартного розділювача – пробілу, табуляції або кінця рядка, наприклад:

```
string first;
cin >> first;
cout << first;
```

Якщо потрібно прочитати весь рядок з пропусками, то, як і для рядків у стилі С, використовується функція **getline()**:

```
getline(cin, first);
```

Цю функцію можна також використовувати з третім необов'язковим аргументом – символом, який буде використовуватись для завершення уведення. Наприклад, при виклику

```
getline(cin, first, '!');
```

зчитування буде тривати до першого входження символу ! (знак оклику). У разі відсутності цього символу уведення завершиться тільки в кінці файлу або при введенні з клавіатури *<Ctrl+z>* (кінець даних).

У всіх випадках виконується автоматична зміна розміру об'єкта класу **string** у відповідності до кількості уведених символів. Саме тому, кількість уведених символів не може бути аргументом функції **getline()** на відміну від зчитування рядків у масив символів (рядки в стилі С).

Функція **getline()**, застосована до об'єкту класу **string**, буде зчитувати дані з вхідного потоку, доки не відбудеться одна з наступних подій:

- Досягнуто кінець файлу.
- Досягнуто роздільний символ (за замовчанням **\n**). Цей символ вилучається з вхідного потоку, але не зберігається.
- Прочитано максимальну можливу кількість символів.

На максимально можливу кількість символів рядку існує кілька обмежень. Перше з них – максимально дозволена довжина рядка,

що задається системою. Як правило, вона є максимально можливим значенням типу **int**. Другий обмежуючий фактор – розмір пам'яті, що доступна програмі.

Враховуючи, що тип **string** є класом, у нього є методи, які викликаються стандартним шляхом:

```
ім'я_об'єкту.ім'я_методу(...);
```

Існує метод **getline** для об'єкту **cin**, який теж дозволяє увести рядок разом з пробілами. У цьому випадку функція-член потребує два аргументи: перший – об'єкт типу **string**, а другий – кількість символів, які потрібно прочитати. Наприклад, **cin.getline(S, 20)**. Очевидно, що цей метод менш зручний, ніж попередній, оскільки розмір рядка, що уводиться, як правило невідомий. Правда, якщо розмір вказано більше реального, зчитування буде відбуватися до символу кінця рядку, але все одно це не завжди зручно, особливо для зовсім невідомого тексту.

Для розв'язання багатьох задач з обробки рядків, потрібно знати, скільки символів зберігає в поточний момент об'єкт string. Існують дві функції-члени класу string для визначення довжини рядка: **size()** і **length()**, які повертають кількість символів у рядку:

```
string S;
int LenS = S.size();
```

або

```
int LenS = S.length();
```

Вони абсолютно ідентичні і їх вибір забезпечується тільки уподобаннями самого програміста. Як вже відзначено, ця кількість символів зберігається в об'єкті класу **string** готова, тож використання в циклі, наприклад, умови продовження **i < S.size()** не призводить до затримок, аналогічних strlen.Тому, згадана там перевірка «**S[i]** не є нуль-термінатором» для рядків класу **string** абсолютно не потрібна (і при цьому ризикована, бо за деяких налаштувань компілятора працює, а за деяких інших вважається виходом за межі діапазону).

Якщо з певних причин потрібно знати максимально можливу довжину рядка, що забезпечується системою (дивись вище), можна використати метод **max_size()**.

Крім того, як і раніше, об'єкт класу **string** є послідовністю символів, а тому дістатися до них можна за допомогою індексів, причому перший символ має індекс 0, а останній – **S.size()-1**. Врешті-решт, при суттєвій зміні методу збереження рядка, все одно клас **string** подає рядок як деяку послідовність символів, а тому основні принципи його обробки не відрізняються від принципів обробки рядків типу C.

Порівняння об'єктів класу **string** здійснюється з використанням перевантажених операцій:

- **==**, **!=** – посимвольне порівняння (результатом операції буде значення **true** при повному збігу рядків, або **false** при розбіжності хоча б в одному символі відповідно);
- **<**, **>**, **<=**, **>=** – лексикографічне порівняння рядків, тобто порівняння до першого відмінного символу. Більшим буде вважатися рядок, у якого перший відмінний символ буде більшим (мати більший код в ASCII-таблиці), а якщо одне зі слів є початком іншого, то більшим буде довше слово.

Мова C++ має досить велику кількість методів для обробки об'єктів класу **string**. Нижче наведений опис деяких з них:

Назва функції	*Пояснення*
`clear()`	Вилучає всі символи рядка.
`empty()`	Повертає значення **true**, якщо рядок пустий, та **false** у протилежному випадку.

`erase(<позиція>)`
`erase(<позиція>,`
`<кількість символів,`
`що вилучаються>)`
	Вилучає символи з рядка, починаючи із заданої позиції до кінця рядка, або задану кількість символів, починаючи із заданої позиції.

`insert(<позиція>,`
`<кількість символів>,`
`<символ>)`
`insert(<позиція>,`
`<підрядок, що`
`вставляється>)`
	Починаючиіз заданої позиції, вставляється задана кількість заданих символів або, починаючи із заданої позиції, вставляється заданий підрядок повністю. Довжина рядка збільшується на кількість символів, що вставляються.

`substr(<позиція>)`
`substr(<позиція>,`
`<кількість символів>)`
	Повертає підрядок даного рядка, починаючи із символу, що міститься у заданій позиції, і до кінця рядка, або підрядок, що починається із заданої позиції та містить задану кількість символів (якщо символів замало, то всі до кінця рядка).

`find(<підрядок>,` `<позиція>)`	Шукає перше входження заданого підрядка у рядку, починаючи із заданої позиції. Якщо позиція не задана, пошук здійснюється з початку рядка. Якщо підрядок знайдено, повертається індекс першого символу, з якого починається підрядок (при заданій позиції це значення буде не менше цієї позиції). Якщо підрядок не знайдено, повертається значення «–1», однак це значення є беззнаковим, тобто насправді є максимальним числом, що вміщується в тип **`unsigned int`**.
`find_first_of(<підрядок>,` `<позиція>)`	Шукає у заданому рядку перше входження будь-якого символу із заданого підрядка, починаючи із заданої позиції. Якщо позиція не задана, пошук здійснюється з початку рядка. Значення, що повертається функцією, визначається аналогічно попередньому методу.

`find_first_not_of`
`(<підрядок>,`
`<позиція>)`
	Шукає в заданому рядку перше входження символу, що відрізняється від символів підрядка. Виклик та результат значення, що повертається, аналогічні попередній функції.

`c_str()`	Повертає вказівник на масив символів, який зберігається у рядку.

Це далеко не всі методи, що має клас **`string`**, але наша мета показати загальні принципи обробки рядків. В подальшому ми пропонуємо Вам шукати інші стандартні методи або створювати власні залежно від поставлених задач.

Розглянемо тепер кілька задач, що вимагають обробки текстів.

Задача 1. Дано довільний англійський текст. Провести частотний аналіз тексту, тобто знайти, скільки разів зустрічається в ньому кожна англійська буква (маленькі та великі букви вважаються однаковими).

Ідея розв'язання. По-перше, оскільки за умовою дано довільний текст, потрібно передбачити, що у вхідному потоці він може бути багаторядковим. Саме тому зчитування потрібно організувати циклом **while**, який працюватиме, доки вхідний потік буде мати хоч якісь символи. При відсутності у вхідному потоці даних (або досягненні символу «**^Z**», він же *<Ctrl+z>*, що свідчить про завершення введення) умова роботи циклу стає хибною і зчитування завершується.

По-друге, пропонуємо для розв'язання задачі використати ідею індексного сортування, яка полягає в тому, що кожна латинська буква тексту буде індексом масиву лічильників. Однак, оскільки індексація в масиві у мові програмування C++ починається з нуля, кожен символ «коригується» відніманням коду великої латинської букви «A». Це приводить до того, що нульовий елемент масиву буде містити кількість латинських букв «A», перший елемент – кількість латинських букв «B», другий елемент – кількість латиських букв «C» і так далі.

Програма мовою C++ буде мати такий вигляд:

```
// Функція виведення результатів
// проведеного частотного аналізу
// Оскільки у латинському алфавіті тільки 26 букв,
// масив містить саме стільки елементів
void print(int cnt[])
{
  for (int i=0; i<26; i++)
    cout << char(i+'A') << ' ' << cnt[i] << endl;
}

int main()
{
  string S;
  // Масив лічильників, ініціалізований нулями
  int answer[26]= {0};
  while(getline(cin,S))
  {
    for (int i=0; i<S.size(); i++)
```

```
        if (toupper(S[i])>='A' && toupper(S[i])<='Z')
          answer[toupper(S[i])-'A']++;
    }
    print(answer);
    return 0;
}
```

Зверніть увагу, що програма обробляє будь-який текст, який може містити не тільки латинські букви, а тому кожен символ рядка перевіряється на приналежність англійському алфавіту. Чи є символ рядка буквою латинського алфавіту можна перевірити й використанням стандартної функції `isalpha`. Тоді команда розгалуження в тілі циклу буде мати такий вигляд:

```
if (isalpha(S[i]))
  answer[toupper(S[i])-'A']++;
```

Задача 2. Дано римське число, записане великими латинськими літерами. Гарантується, що це число не перевищує значення 2000 десяткової системи числення. Знайдіть десятковий еквівалент цього числа.

Ідея розв'язання: Як відомо, римська система числення має всього сім цифр:

I – 1 (одиниця);	C – 100
V – 5	D – 500
X – 10	M – 1000
L – 50	

Далі з цих цифр утворюються числа, причому для отримання сучасного десяткового еквівалента числа потрібно між цифрами виконувати арифметичні дії додавання або віднімання. Якщо менша цифра стоїть лівіше більшої, її значення віднімається від результату, інакше додається. Додаються також рівні цифри. Простіше всього дивитися на число справа наліво, і тоді перша праворуч цифра завжди додається, а всі наступні або додаються (якщо вони більші або рівні попередньої), або віднімаються (якщо вони менші попередньої). З урахуванням описаного алгоритму, розв'язок мовою C++ має такий вигляд.

```
// Функція, що повертає десятковий еквівалент
// однієї римської цифри
int roman_to_dec(char c)
{
```

```
  if (c=='I') return 1;
  if (c=='V') return 5;
  if (c=='X') return 10;
  if (c=='L') return 50;
  if (c=='C') return 100;
  if (c=='D') return 500;
  if (c=='M') return 1000;
}

int main()
{
  string s;
  cin >> s;
  int answer = roman_to_dec(s[s.size()-1]);
  for (int i=s.size()-2; i>=0; i--)
    if (roman_to_dec(s[i])>=roman_to_dec(s[i+1]))
      answer+=roman_to_dec(s[i]);
    else
      answer -= roman_to_dec(s[i]);
  cout << answer << '\n';
  return 0;
}
```

Задача 3. Дано десяткове число, що не перевищує за значенням 3000. Знайти еквівалент цього числа у римській системі числення.

Ідея розв'язання. Римська система числення є непозиційною і має 7 цифр (дивись задачу 2). Причому, якщо уважно подивитися на цифри, то виявиться, що три цифри (`I`, `V` та `X`) беруть участь в утворенні цифри одиниць десяткового числа, три римські цифри (`X`, `L` та `C`) утворюють цифру десятків десяткового числа,та три цифри (`C`, `D` та `M`)утворюють цифру сотень. Тисячі в десятковому числі можна отримати тільки за допомогою цифри `M`. Між цифрами виконуються арифметичні дії додавання або віднімання в залежності від їх взаємного розташування вчислі. Відповідно, якщо потрібно отримати римський записчисла 3, використовується єдина римська цифра `I`, але тричі – `III`, а якщо числа 6, то дві римських цифри `V` та `I` – `VI`. Все було б нескладно, якби не існувало обмеження: більше трьох однакових цифр стояти поруч не може!!! Відповідно, вісім можна отримати, як `VIII`, а дев'ять, як `VIIII`– ні! Для цифр, що пограничні з п'ятіркою, десяткою, п'ятидесяткою, соткою, п'ятисоткою та тисячею, існує інше подання: четвірка – це `IV`, дев'ять – `IX`, сорок – `XL`, дев'яносто – `XC`, чотириста – `CD` та дев'ятсот – `CM`.

Оскільки для цифр різних розрядів десяткового числа використовуються різні трійки чисел, пропонуємо такий підхід: для кожного розряду десяткового числа використаємо свою трійку римських цифр (дивись вище), яку будемо передавати як аргумент у функцію конвертації однієї десяткової цифри у римський еквівалент. Остаточне римське число будемо отримувати склеюванням окремих частин римського числа, що конвертуються в залежності від розряду десяткового представлення числа. Розв'язок задачі мовою C++:

```
//Функція, що повертає римський еквівалент
//однієї десяткової цифри
string dec_to_roman(int numeral,string dig_roman)
{
  string s; //Гарантовано порожній рядок
  if(numeral==1) s+=dig_roman[0];
  if(numeral==2) s+=dig_roman[0]+dig_roman[0];
  if(numeral==3) s+=dig_roman[0]+dig_roman[0]+dig_roman[0];
  if(numeral==4) s+=dig_roman[0]+dig_roman[1];
  if(numeral==5) s+=dig_roman[1];
  if(numeral==6) s+=dig_roman[1]+dig_roman[0];
  if(numeral==7) s+=dig_roman[1]+dig_roman[0]+dig_roman[0];
  if(numeral==8) s+=dig_roman[1]+dig_roman[0]
                   +dig_roman[0]+dig_roman[0];
  if(numeral==9) s+=dig_roman[0]+dig_roman[2];
  return s;
}

int main()
{
  int N;
  cin>>N;
  // Отримання цифри тисяч: N/1000 букв 'M'
  string answer(N/1000, 'M'); // 'M' типу char
  // Отримання цифри сотень
  answer += dec_to_roman(N/100%10,"CDM");
  // Отримання цифри десятків
  answer += dec_to_roman(N/10%10,"XLC");
  // Отримання цифри одиниць
  answer += dec_to_roman(N%10,"IVX");
  cout << answer << '\n';
  return 0;
}
```

Задача 4. Дано рядок, що містить тільки круглі (відкриваючі та закриваючі) дужки. Напишіть програму, яка перевіряє правиль-

ність даної дужкової послідовності, тобто кожній закриваючій дужці обов'язково передує відкриваюча і кількість дужок обох типів однакова. Наприклад, послідовність `(()())(())` є правильною, а послідовності `())))(` або `((())` – ні.

Ідея розв'язання. Уведемо поняття так званого балансу дужок, який буде збільшуватися на 1, якщо в рядку зустрічається відкриваюча дужка, та зменшується на 1 при зустрічанні закриваючої. Тоді для правильної послідовності `(()())(())` стан балансу буде таким:

```
0 1 2 1 2 1 0 1 2 1 0
```

Перший 0 – початковий стан балансу. Далі кожне число відповідає за баланс на кожному кроці перевірки символів рядку.

Для помилкового рядку, наприклад, `())))(` стан буде таким:

```
0 1 2 1 0 -1 0
```

Поява числа «–1» серед результатів свідчить про помилку: зустрілася закриваюча дужка раніше відкриваючої. Крім того, очевидно, що наприкінці роботи значення балансу повинно бути 0 – кількість дужок обох типів однакова. Функція, яка перевіряє баланс однієї дужкової послідовності буде мати вигляд:

```
bool balance(string S)
{
  int cnt=0;
  for (int i=0; i<S.size(); i++)
  {
    if (S[i]==’(‘) cnt++;
    else cnt--;
    if (cnt<0) return false;
  }
  return (cnt==0);
}
```

Сподіваємося, що основну програму Ви оформите самостійно.

Задача 5. Нещодавно експедиція у сусідню галактику знайшла на одній з тамтешніх планет цивілізацію маленьких жвавих гуманоїдів, яких дослідники умовно стали називати «атугалапами». Їх самобутню мову науковці зрозуміли далеко не одразу. Але тепер вони знають, за якими правилами граматики атугалапи утворюють слова. Вони дуже прості:

1. Алфавіт атугалапської мови містить всього шість літер: `~ @ # $ % &`

2. Якщо закодувати пари літер:

- `~ - (`
- `@ - )`
- `# - [`
- `$ - ]`
- `% - {`
- `& - }`

і записати закодоване слово атугалапської мови, то одержимо правильну дужкову послідовність з трьох типів дужок.

Напишіть програму, яка визначає, чи є запропоноване слово правильним з точки зору атугалапів.

Ідея розв'язання. Як відомо, задача про дужкові послідовності розв'язується шляхом підрахунку так званого балансу дужок. Однак, запропонована задача відрізняється від стандартної тим, що рядок містить три пари різновидів дужок. Підрахунок балансу кожної з пар може не виявити помилки, оскільки баланс кожного різновиду може бути правильним, а розташування дужок – помилковим. Наприклад, послідовність `({}[])` з точки зору балансу кожного з різновидів дужок є правильною, але дві останні дужки стоять неправильно – квадратна закриваюча дужка, очевидно, повинна передувати круглій, оскільки відкриваюча кругла стояла раніше відкриваючої квадратної. Тобто, правильною буде така послідовність `({}[])`.

Виходячи з цього, нам потрібно слідкувати за правильним розташуванням відповідних пар дужок: якщо остання відкрита дужка була круглою, перша закрита повинна бути також круглою, а якщо квадратною – то квадратною. Допомогти у розв'язанні цієї задачі може такий підхід. Домовимося при перегляді символів заданого рядку всі відкриваючі дужки зберігати у допоміжному рядку, а при попаданні на закриваючу дужку перевіряти відповідність її останній збереженій дужці. Якщо вони не відповідають одна одній (наприклад, закриваюча є круглою, а відкриваюча ні), рядок є помилковим, якщо все правильно, з допоміжного рядку можна вилучити останній символ, оскільки знайдена пара дужок є правильною і далі потрібно перевіряти відповідність іншої пари. Наприкінці роботи допоміжний рядок повинен стати пустим, оскільки кількість відкритих дужок повинна дорівнювати кількості закритих.

Оскільки нові дужки потрапляють завжди тільки в кінець допоміжного рядка і вилучаються у випадку правильної пари теж з кінця, описана структура працює за законом стеку: останній прийшов – перший пішов. Саме тому у запропонованому розв'язку цей рядок має назву `Stack`.

Розв'язок мовою C++ буде мати вигляд:

```
bool check(string S)
{
  string Stack;
  for (int i=0; i<S.size(); i++)
    if (S[i]=='~' || S[i]=='#' || S[i]=='%')
      Stack+=S[i];
    else
    {
      if (Stack.empty())
        return false; // закриваюча дужка
        // взагалі без відкриваючої
      if (S[i]=='@' && Stack.back() !='~')
        return false;
      if (S[i]=='$' && Stack.back() !='#')
        return false;
      if (S[i]=='&' && Stack.back() !='%')
        return false;
      Stack.erase(Stack.size()-1,1);
    }
  return Stack.empty();
}

int main()
{
  string S;
  while(cin>>S)
  {
    if (check(S)) cout<<"OK\n";
    else cout<<"WRONG\n";
  }
  return 0;
}
```

Використаний метод **back()** повертає останній символ рядка. Того ж ефекту можна добитися, написавши **Stack[Stack.size()-1]**. Клас string взагалі часто надає різні способи зробити одне й те саме. Зокрема, вилучення останнього символу, замість наведеного у програмі поєднання erase та size, можна зробити через **Stack.pop_back()**.

Задача 6. Дано рядок, що складається зі слів та чисел. Слова складаються з латинських букв та цифр, числа – тільки з цифр та гарантовано не починаються з 0. Розділювачами між словами та

числами є пробіли та переходи на новий рядок (не обов'язково по одному). Знайти у цьому рядку всі числа-паліндроми.

Ідея розв'язання: Нагадуємо, що паліндромами називаються будь-які слова, які читаються в обох напрямках однаково, наприклад, `Alla`, `потоп`, `aasdDfddsaa` тощо. Очевидно, що перевірку таких слів потрібно здійснювати шляхом порівняння першого та останнього, другого та передостаннього і т. д. символів рядка. Загальна кількість порівнянь буде дорівнювати половині довжини рядка.

У запропонованій задачі це ще повинні бути числа: `1221`, `1256521` тощо. А тому у функцію перевірки потрібно ще вставити перевірку кожного символу рядка на те, чи є він цифрою. У випадку відмінності символів або наявності в слові символів, відмінних від цифр, функція повинна повертати значення `false`. Якщо ж всі перевірки будуть хибними до завершення циклу, функція повинна повернути значення `true`.

Оскільки за умовою всі слова в тексті розділені пробілами, їх можна читати, використовуючи особливості роботи стандартного уведення за допомогою об'єкта `cin`, тобто `cin>>S`.

Програма, що розв'язує дану задачу, мовою C++ має такий вигляд:

```
bool is_palindrome(string s)
{
  for(int i=0;i<=s.size()/2;i++)
    if(s[i]!=s[s.size()-i-1]) return false;
  return true;
}

bool str_is_number(string s)
{
  for (int i=0; i<s.size(); i++)
    if (!isdigit(s[i])) return false;
  return true;
}

int main()
{
  string s;
  bool flag=false;
  while (cin>>s)
  {
```

```
      if (str_is_number(s) && is_palindrome(s))
      {
        flag=true;
        cout<<s<<'\n';
      }
    }
    if (!flag) cout<<"No numericalpalindromes.\n";
    return 0;
}
```

Зверніть увагу, що в даному розв'язку передбачена можливість відсутності шуканих чисел. Для цього застосовується логічна змінна **flag**, яка на початку роботи має значення **false** (числа не знайдені) і змінює своє значення на **true**, якщо число з шуканими властивостями знайдено. Такий підхід можна застосовувати і в інших задачах, де потрібні елементи можуть бути не знайдені.

Розв'язок задачі можна скоротити, якщо дві функції об'єднати в одну в такий спосіб:

```
bool is_num_palindrome(string s)
{
  for(int i=0; i<=s.size()/2; i++)
    if(s[i]!=s[s.size()-i-1] || !isdigit(s[i]))
      return false;
  return true;
}
```

Однак, ми використали функцію для того, щоб показати, як перевіряти будь-які паліндроми (не тільки числові). Правда, у даній функції не враховується, що великі та маленькі букви вважаються однаковими. Врахуємо це у наступній версії функції, яку, крім того, ускладнимо тим, що будемо перевіряти не тільки слово-паліндром, а й цілу фразу-паліндром. Із найвідоміших фраз-паліндромів є речення «А роза упала на лапу Азора». Якщо його читати, не звертаючи увагу на пробіли, то воно є паліндромом.

Поміркувавши над різницею цих двох завдань, нескладно прийти до висновку, що при порівнянні потрібно ігнорувати пробіли, а порівнювати тільки непробільні символи. Крім того, кількість перевірок заздалегідь визначити не можна, а тому пропонуємо організувати прохід по рядку двома індексами назустріч один одному до точки перетину індексів. Функція, яка виконує описаний алгоритм перевірки, має такий вигляд:

```
bool is_palindrome(string s)
{
  int i=0, j=s.size()-1;
  while(i<j)
  {
    if(s[i]==' ') i++; // Пропуск пробілів зліва
    else if(s[j]==' ') j--; // Пропускпробілівсправа
    else if(toupper(s[i])!=toupper(s[j]))
      return false; // Символи відрізняються
    else { // Букви співпали, йдемо далі,
      i++; j--; // пропускаючи ці однакові букви
    }
  }
  return true;
}
```

Задача 7. Дано багаторядковий текст, кожен з рядків якого містить синтаксично правильний арифметичний вираз. Вираз складається з двох десяткових чисел (цілих або дійсних), між якими знаходиться одна з арифметичних операцій – додавання, віднімання, множення або ділення. Обчислити значення кожного з виразів.

Ідея розв'язання. Виходячи з умови, кожен рядок тексту має такий загальний вигляд:

`число⊕число`

Наприклад, `12+4.5` або `3.298/7.1`.

Відповідно розв'язок задачі полягає в тому, щоб розбити кожен рядок на дві частини: до знаку арифметичної дії та після. Далі виконати конвертацію кожного з виділених чисел у дійсне число, і, нарешті, виконати арифметичну дію з урахуванням того, що на 0 ділити не можна.

```
int main()
{
  string expression,first,second;
  double answer;
  while (getline(cin,expression))
  {
// Визначення позиції знака арифметичної операції
    intpos_operation=
        expression.find_first_of(«+-*/»);
// Виділення першого числа з виразу
    first=expression.substr(0,pos_operation);
// Конвертація першого числа
    double Num1=atof(first.c_str());
```

```
// Виділення другого числа з виразу
    second=expression.substr(pos_operation+1,
        expression.size()-1);
// Конвертація другого числа
    double Num2=atof(second.c_str());
    char operation=expression[pos_operation];
// Перевірка ділення на 0
    if (operation=='/' && Num2==0)
      cout<<»Error\n»;
    else
    {
      if (operation=='+') answer=Num1+Num2;
      if (operation=='-') answer=Num1-Num2;
      if (operation=='*') answer=Num1*Num2;
      if (operation=='/') answer=Num1/Num2;
// Виведення результату з точністю до тисячних
      cout<<fixed<<setprecision(3)<<answer<<'\n';
    }
  }
  return 0;
}
```

У наведеній програмі використані такі стандартні методи класу `string`:

- `expression.find_first_of("+-*/")` – пошук позиції першого входження заданого символу (у нашому випадку це пошук позиції арифметичної операції);
- `expression.substr(0,pos_operation)` – виділення частини рядка між двома заданими позиціями (у наведеному прикладі від початку рядка до позиції арифметичної операції);
- `first.c_str()` – вказівник на масив символів, в якому міститься рядок `first`.

Останній метод потрібен для того, щоб можна було скористатися стандартною функцією конвертації рядка в дійсне число `atof`, яка може виконати конвертацію тільки масиву символів.

Описаний прийом можна застосувати для обчислення довшого виразу, що містить кілька арифметичних дій з однаковим пріоритетом виконання операцій, наприклад, `2+34-15` або `13*45/23/2*12.5`.

Припустимо, що заданий вираз може містити тільки цілі числа та операції додавання або віднімання. Перше число виділяємо окремо і відразу записуємо в результат. Перед запуском циклу нам буде відомий індекс першої операції у виразі, яку ми запам'ятаємо

для подальшої роботи. Далі, оскільки всі інші операції будуть виконуватися в порядку слідування, будемо виділяти числа по черзі і відповідно до запам'ятованої операції додавати або віднімати їх від результату. Програма мовою C++ матиме такий вигляд:

```
int main()
{
  string expression;
  cin>>expression;
  int i;
  char operation;
// Пошук першого символу додавання або віднімання,
// що знаходиться далі першої позиції
  int index=expression.find_first_of(«+-»,1);
// Виділення першого числа з виразу
  string addendum=expression.substr(0,index);
// Конвертація першого числа у відповідь
  long long answer=atoll(addendum.c_str());
  long long Num;
  while (index>=0)
  {
    operation=expression[index];
//Наступне число після арифметичної операції
    i=index+1;
    index=expression.find_first_of(«+-»,i);
    addendum=expression.substr(i,index-i);
    Num=atoll(addendum.c_str());
    if(operation=='+') answer+=Num;
    elseanswer-=Num;
  }
  cout<<answer<<'\n';
  return 0;
}
```

Задача 8. Дано багаторядковий текст. Будемо вважати у тексті словом послідовність довільних символів між символами-розділювачами, якими є: пробіл, «`,`», «`.`», «`:`», «`;`», «`!`», «`?`», «`"`», «`[`», «`]`» та їх послідовності. Знайдіть у тексті слово, що містить найбільшу кількість різних англійських букв (маленькі та великі літери вважаються однаковими). Якщо слів з найбільшою кількістю різних букв кілька, потрібно вивести будь-яке з них.

Ідея розв'язання. Зверніть увагу на те, що дана задача відрізняється від усіх попередніх тим, що обробка рядку ведеться не посимвольно, а словами, причому роздільниками між словами є нестан-

дартний набір символів. Виходячи з цього, використати особливості уведення в мові C++ до пробілів або кінця рядка неможливо (а можливість задати роздільник у **getline** не допоможе, бо роздільників багато). Доведеться зчитувати цілими рядками, а потім розбивати рядки на окремі слова. Пропонуємо один з методів виділення поточного слова із заданого рядка, який оформимо у вигляді функції. Ця функція має три аргументи:

- рядок **S**, який передається у функцію за посиланням, щоб не витрачати зайві ресурси на створення копії;
- поточний індекс **ind**, правіше якого буде знаходитися найближче слово, передається також за посиланням;
- рядок **separator**, що містить всі символи, які за умовою вважаються розділювачами.

Сама функція має такий вигляд:

```
string word(string &S, int ind, string separator)
{
  int len=S.find_first_of(separator,ind)-ind;
  return S.substr(ind,len);
}
```

Крім того, нам потрібна функція, яка підраховує кількість унікальних латинських символів у заданому рядку. Для цього пропонується такий підхід. По-перше, методом, який було запропоновано у *Задачі №1*, підрахуємо кількість кожної латинської букви без урахування регістру в масиві лічильників. Потім потрібно пройти по цьому масиву і підрахувати кількість його ненульових елементів. Оскільки у мові програмування C++ значення **true** трактується як «1», кількість ненульових елементів можна порахувати без використання команди розгалуження. Функція підрахунку різних латинських букв у слові має вигляд:

```
int count_lat(string &S)
{
  int cnt[26]={0};
// Підрахунок кількості кожної латинської букви
  for (int i=0; i<S.size(); i++)
    if (isalpha(S[i]))
      cnt[toupper(S[i])-'A']++;
// Підрахунок кількості ненульовихелементів масиву
  int ans=0;
```

```
  for(int i=0; i<26; i++)
    ans+=cnt[i]>0;
  return ans;
}
```

Використовуючи дві наведені функції, основна програма, що знаходить будь-яке слово, що містить максимальну кількість латинських букв (без урахування регістру) в багаторядковому довільному тексті, має такий вигляд.

```
int main()
{
  string S,temp, answer;
  while(getline(cin,S))
  {
    int i=0;
    while(i<S.size())
    {
// Виділеннячергового слова міжрозділювачами
      temp=word(S,i,».,\»:;[]!? «);
// Перехід через знайдене слово
      i +=temp.size()+1;
      if (count_lat(temp)>count_lat(answer))
        answer=temp;
    }
  }
  cout<<answer<<'\n';
  return 0;
}
```

Очевидно, що запропоновані розв'язки не є єдино можливими, однак вони є прикладом посимвольної обробки рядків, обробки підрядками та обробки окремими словами – послідовностями символів між розділювачами. Розвиваючи ідеологію розділювачів, можна текст ділити на інші структурні елементи: числа (в задачі №7 арифметичні операції), речення, лексеми у виразах, тощо. Наприклад, у задачі «Фрази-паліндроми» можна розділювачами вважати розділові знаки «.», «?», «!» або їх послідовності, і шукати у тексті всі фрази-паліндроми, а не одну.

1 Теоретичний матеріал

Розглянемо деякі базові поняття та засоби теорії чисел — комбінаторну формулу знаходження кількості дільників числá (за відомим розкладенням на прості множники), побудову такого розкладення, перевірку простоти́ (перебором та «решетом Ератосфена»), а також НСД, НСК та алгоритм Евкліда. Крім того, будуть згадані ще деякі питання.

У рамках цієї теми скрізь, де не сказано іншого, розглядаються натуральні (цілі додатні) чи́сла.

У рамках цієї теми масово використовуватимемо цілочисельні ділення (Pascal — "`div`"; Python 3 — "`//`"; C/C++/Java та Python 2 — "`/`", за умови, що обидва аргументи цілочисельні) та взяття залишку (він же «рештка», рос. «остаток», англ. «remainder»; Pascal — "`mod`"; C/C++/Java/Python — "`%`"). У математичних записах будемо позначати ці дії "div" та "mod" відповідно. Один зі смислів цих дій — якщо мати 17 грн і хотіти витратити їх усі на товари вартістю 5 грн/шт, вдасться купити $17 \operatorname{div} 5 = 3$ (шт) цього товару, і $17 \bmod 5 = 2$ (грн) залишиться. Дробове ділення (за яким, наприклад, $17/5 = 3{,}4$) теж використовуватиметься, але рідше.

Твердження та позначки про два числá a, b

- a ділиться на b націло;
- a *кра́тне* b;
- b — *дільник* a;
- $a \vdots b$ (математичний запис);
- `a mod b = 0` (Pascal);
- `a%b==0` (C/C++/Java/Python)

є рівносильними (позначають одне й те са́ме).

Число *просте*, якщо воно має рівно два дільники — число 1 та самого себе. (Отже, *число 1 не є простим.*) Перші 10 простих чисел: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29. Сукупність простих чисел нескіче́нна. (Це класична теорема, охочі можуть знайти доведення чи навіть виконати його самостійно, воно досить легке.)

Чи́сла, що мають більше двох дільників, називаються *скла́дені* (не склад**ні́**, а скла́д**е**ні). Російською — *просто́е/составно́е* (не «сложное»!). Англійською — *prime/composite* number (не «easy» чи «simple»!). Польською — liczba *pierwsza/złożona*.

Розкла́дення (або *ро́зклад*) *на прості множники* — подання у вигляді добутку, де множниками є *лише різні прості* чи́сла (можливо, у степенях). Наприклад: $17 = 17$; $720 = 2^4 \times 3^2 \times 5$; $2015 = 5 \times 13 \times 31$; $2017 = 2017$; $2019 = 3 \times 673$ (де 2017 та 673 є простими). Англійською — *factorization*; багато хто каже «*факторизація*» і українською.

Основна теорема арифметики стверджує: «*для кожного числа́ існує єдине (з точністю до порядку) розкладення на прості множники*». Це твердження (не формулюючи як теорему і не доводячи) використовував ще Евклід у IV ст. до н. е.; строге доведення дав К. Ф. Ґаусс у 1801 р. «З точністю до порядку» означає, що множники можна переставляти (як-то «3×2^2» замість «$2^2\times3$»), але не можна отримати інше розкладення того самого числа́, що відрізнялося б істотніше (набором та/або степенями простих чисел). *Якби* одиницю вважали простим числом, *були б* виключення (наприклад, $2^2\times3=1\times2^2\times3=1^{100}\times2^2\times3$); власне, це одна з причин не вважати одиницю простим числом.

Спільний дільник чисел a та b — число, на яке діляться одночасно і a, і b. Аналогічно, *спільне кратне* чисел a та b — число, яке ділиться одночасно і на a, і на b. Терміни *найбільший спільний дільник* a та b, скорочено *НСД*(a,b) та *найменше спільне кратне* a та b, скорочено *НСК*(a,b) вводяться очевидним чином, як найбільший серед спільних дільників та найменше серед (натуральних) спільних кратних.

Приклади: НСД$(7,9)$=1 НСД$(8,10)$=2 НСД$(4,8)$=4
НСК$(7,9)$=63 НСК$(8,10)$=40 НСК$(4,8)$=8

Переклады:

	greatest common divisor	*gcd*
НСД	*наибóльший общий делитель*	*НОД*
НСК	*least common multiple*	*lcm*
	наибóльшее общее кратное	*НОК*

Інваріант циклу (у програмуванні) — умова, істинна перед початком виконання циклу та наприкінці кожної ітерації циклу. Інваріант не є складовою програми і не потрібен для її запуску; його типове призначення — задати деякий зв'язок (співвідношення) між різними значеннями одних і тих самих змінних у різні моменти виконання алгоритму, з метою теоретичного доведення правильності неочевидного алгоритму. Далі по тексту інваріанти будуть використані у поясненнях розд. 1.4 (алгоритм Евкліда) та розборах деяких задач.

1.1 Комбінаторна формула знаходження кількості дільників за розкладенням на прості множники

Нехай для деякого числа́ відоме розкладення

$$N = p_1^{m_1} \times p_2^{m_2} \times \cdots \times p_k^{m_k}$$

(усі p_j прості та різні, усі m_j натуральні). Виявляється, тоді можна дуже легко знайти *загальну кількість усіх дільників* такого N.

Розглянемо число $12 = 2^2 \cdot 3^1$; усі його дільники: 1, 2, 3, 4, 6, 12. Запишемо їх, у вигляді розкладень, у табличці праворуч. Бачимо, що у розкладеннях цих дільників просте число 2 може бути у степені 0, або 1, або 2; просте число 3 — у степені 0, або 1. При цьому, кожен можливий степінь 2 (рядок таблички) може поєднуватися з кожним можливим степенем 3 (стовпчиком таблички).

$1 = 2^0 \cdot 3^0$	$3 = 2^0 \cdot 3^1$
$2 = 2^1 \cdot 3^0$	$6 = 2^1 \cdot 3^1$
$4 = 2^2 \cdot 3^0$	$12 = 2^2 \cdot 3^1$

Звідси очевидне таке узагальнення: дільники $p_1^{m_1} \times p_2^{m_2} \times \ldots \times p_k^{m_k}$ можуть містити p_1 у будь-якому степені від 0 включно до m_1 включно ($m_1 + 1$ способів), p_2 у степені від 0 до m_2 ($m_2 + 1$ способів), і так далі, причому кожен можливий степінь може поєднуватися з усіма степенями інших простих. Що дає загальну кількість дільників

$$(m_1 + 1) \times (m_2 + 1) \times \cdots \times (m_k + 1).$$

1.2 Кілька застосувань перебору до $\sqrt{n}$

1.2.1 Перелік дільників

Розглянемо задачу: *«Для вказаного числá, вивести у порядку зростання всі його дільники»*. Здавалося б, це може бути щось дуже просте, як-то `for i:=1 to n do if n mod i = 0 then write(i, ' ')`. Але це потребу́є n ітерацій циклу, тоді як можна значно менше.

1 2 3 4 6 12

Якщо знов подивитися на дільники числá 12, можна побачити, що $1 \times 12 = 2 \times 6 = 3 \times 4 = 12$, тобто всі дільники розбилися на пари, добутки яких дорівнюють 12. Покажемо, що це не особливість числá 12, а властивість, яку легко узагальнити. Нехай i — дільник n. Тоді n/i — ціле, й також дільник n. Чи́сла i та n/i не можуть одночасно бути більшими $\sqrt{n}$, тому́ досить *перебрати* (з перевіркою, чи ділиться націло) $1 \leqslant i \leqslant \sqrt{n}$, і на підставі цього можна *обчислити* решту дільників.

Наприклад, можна один раз прокрутити цикл 1 `to` $\sqrt{n}$ і вивести всі знайдені дільники, а потім наступний (не вкладений, а наступний) цикл $\sqrt{n}$ `downto` 1 (у порядку спадання) і вивести `n div i`. Або, можна у циклі 1 `to` $\sqrt{n}$ не лише виводити дільники, а ще й запам'ятовувати їх у масив, щоб дру́гий цикл перебирав лише дільники, а не всі чи́сла проміжку. Відмінність між цими варіантами *не* істотна.

А «n чи $\sqrt{n}$» цілком може бути істотною відмінністю. Наведене далі теоретичне оцінювання, скільки часу працюватиме який алгоритм, вельми не точне; багато залежить від «заліза», компілятора та інших

подібних чинників, тож «прикинутий» результат легко може відрізнятися від правильного у десятки разів. Але відмінність у тривалості роботи алгоритмів ще більша.

Тактова частота сучасного комп'ютера — кілька гігагерців (мільярдів тактів за секунду); навіть у примітивному алгоритмі «до n» кожна ітерація циклу займає кілька тактів, тож 10^9 (один мільярд) ітерацій за секунду — станом на 2019 р., можливо, але лише враховуючи, що там досить простий цикл, і лише при поєднанні швидкого «заліза» і ефективного компілятора. Так що слід очікувати, що алгоритм «до n» при $n \approx 10^6$ працюватиме орієнтовно $\frac{10^6 \text{дій}}{10^9 \text{дій/с}} \approx 0{,}001$ с, а при $n \approx 10^{12}$ — орієнтовно $\frac{10^{12} \text{дій}}{10^9 \text{дій/с}} \approx 10^3$ с $\approx$ десятки хвилин. Для алгоритму «до $\sqrt{n}$» дії, яких треба зробити $\sqrt{n}$ штук, дещо складніші (з'являється обчислення $n \operatorname{div} i$; треба чи то запам'ятовувати ці частки $n \operatorname{div} i$, чи то крутити два послідовні цикли з $\sqrt{n}$ ітераціями кожен; тощо), тому сподіватися на 10^9 ітер/с вже нереально, але й нема причин, щоб витрати на одну дію виросли більш, чим у кілька разів. Тобто, 10^8 ітер/с чи навіть трохи більше — цілком реально. Так що при $n \approx 10^{12}$ слід очікувати тривалості порядка $\frac{\sqrt{10^{12}} \text{дій}}{10^8 \text{дій/с}} \approx 0{,}01$ с (проти десятків хвилин).

Вираження тривалості роботи алгоритму функціями від розміру вхідних даних є стандартним розділом аналізу алгоритмів і називається *асимптотичними оцінками*; хто цього не знає, знайдіть за іншими джерелами, надалі цей засіб вважатиметься відомим.

В цьому описі алгоритму «до $\sqrt{n}$» замовчано одну з проблем: якщо n є точним квадратом ($\sqrt{n}$ цілий), то цей $\sqrt{n}$ буде дільником, але без пари ($n \operatorname{div} \sqrt{n}$ дасть $\sqrt{n}$, тобто знову його ж); такий дільник треба вивести, але лише один раз. За цим просто треба прослідкувати, акуратно визначивши межу цикла та, можливо, написавши відповідний `if`.

1.2.2 Перевірка простоти́

Якщо для числа́ $n > 1$ потрібно лише сказати, просте воно чи скла́дене, ідея «крутити цикл до $\sqrt{n}$» теж правильна. Дільник у проміжку $\sqrt{n} \leqslant d < n$ можливий лише коли $1 < n \operatorname{div} d \leqslant \sqrt{n}$ теж є дільником, тож якщо у проміжку $1 < p \leqslant \sqrt{n}$ дільників нема, то їх нема взагалі.

Тут можлива додаткова оптимізація: перебираючи потенційні дільники від 2 до $\sqrt{n}$, як тільки поділити націло вдалося, негайно обривати цикл і стверджувати «число скла́дене». Наприклад, як праворуч (варто

попереднью переконатися, що $n>1$, бо на $n=1$ чи $n=0$ цей код помилково скаже, що вони прості, а при $n<0$ станеться помилка).

Чи важливо мати для заокругленого кореня окрему змінну `sqrtN`, чи можна писати й `round(sqrt(n))`? Теоретично, з точки зору лише правильності та асимптотичних оцінок, однаково. Практично — можлива різниця у швидкодії до 1,5–3 разів, причому деталі сильно залежать від мови програмування. Мовою Pascal запис `for p:=2 to round(sqrt(n))...` означає, що корінь обчислюється один раз. Як сприйме запис `for(int p=2; p<=sqrt((double)n); p++)...` C/C++, залежить від версії, опцій компілятора, тощо. Може обчислити корінь один раз, а може і переобчислювати його на кожній ітерації, що досить довго. А запис через додаткову змінну дозволяє гарантувати, що це буде один раз. Заодно відзначимо, що у C/C++/Java/C♯ можливий ще спосіб `for(long long p=2; p*p<=n; p++)...`; такий код працює швидше, ніж переобчислення `sqrt` на кожній ітерації, але довше, ніж порівняння зі збереженим `sqrtN`.

```
sqrtN := round(sqrt(n));
prime := true;
for p:=2 to sqrtN do
  if n mod p = 0 then begin
    prime := false;
    break; // обриває цикл
  end;
// prime дорівнює true <=>
// <=> число n просте
```

Наскільки важливе обривання циклу (`break`)? Для деяких чисел (простих; квадратів простих; тих, що мають лише два прості дільники, близькі до $\sqrt{n}$) це не дає виграшу. Але якщо взяти іншу крайність — великі па́рні чи́сла — там кількість ітерацій зменшується з $\sqrt{n}$ до 1. А па́рні чи́сла — зовсім не рідкісна ситуація, їх аж половина від усіх. Зокрема, при перевірці усіх підряд чисел деякого проміжку a, $a+1$, $a+2$, ..., $b-1$, b такий `break` дає виграш приблизно у 10 разів.

1.2.3 Побудова розкладення на прості множники

Щоб знайти розкладення (як-то $720=2^4\times3^2\times5$), теж переберемо потенційні дільники. При знаходженні дільника p (на який n справді поділилося), *зменшимо* n (`n:=n div p`, воно ж `n/=p`). Дільник може входити кілька разів (як степінь), тому це варто взяти у вкладений цикл.

Тут не можна обривати цикл, знайшовши перший дільник (подальші теж потрібні!). Зате оптимізацію «цикл не до n, а до $\sqrt{n}$» можна не лише зберегти, а й посилити, перевстановлюючи при зменшенні n межу перебору на корінь зі зменшеного n. (Використання змінної `sqrtN`

стає важливим, бо дозволяє і уникнути переобчислень кореня, коли n не змінювалось, і зменшувати діапазон перебору при зменшенні n.)

Розглянемо приклад цього процесу для 111125. Спроби поділити на 2, на 3 та на 4 невдалі. А на 5 ділиться націло зразу кілька разів: $111125 \operatorname{div} 5 = 22225$; $22225 \operatorname{div} 5 = 4445$; $4445 \operatorname{div} 5 = 889$; більше на 5 не ділиться. Значить, у розкладення йде пара $\langle 5; 3\rangle$ (5^3), і продовжуємо з числом 889 і межею перебору дільників $\sqrt{889} \approx 30$. Спроба поділити 889 на 6 невдала. $889 \operatorname{div} 7 = 127$, більше на 7 не ділиться. У розкладення йде пара $\langle 7; 1\rangle$ (7^1), продовжуємо з $n = 127$, $\sqrt{n} = \sqrt{127} \approx 11$. Спроби поділити 127 на 8, на 9, на 10 та на 11 невдалі, $12 > \sqrt{127}$ — отже, 127 просте число, і розкладення має вигляд $111125 = 5^3 \times 7^1 \times 127^1$.

Реалізація може бути приблизно такою.

Pascal

```
numFacts:=0;
p:=2;
sqrtN:=round(sqrt(n));
while p<=sqrtN do begin
  if n mod p = 0 then begin
    inc(numFacts);
    factRes[numFacts].prime := p;
    factRes[numFacts].degree := 0;
    repeat
      inc(factRes[numFacts].degree);
      n := n div p;
    until n mod p <> 0;
    sqrtN:=round(sqrt(n));
  end;
  inc(p);
end;
if n>1 then begin
  inc(numFacts);
  factRes[numFacts].prime := n;
  factRes[numFacts].degree := 1;
end;
```

C++ (з STL)

```
int p=2;
int sqrtN = (int)sqrt(n+0.5);
vector<pair<long long, short>> factRes;
while(p<=sqrtN) {
  if(n%p==0) {
    factRes.push_back(make_pair(p,0));
    do {
      factRes.back().second++;
      n /= p;
    } while(n%p==0);
    sqrtN=(int)sqrt(n+0.5);
  }
  p++;
}


if(n>1)
  factRes.push_back(make_pair(n,1));
```

Розгалуження "`if(n>1)...`" наприкінці потрібне, бо, з одного боку, треба долучити до розкладення-результату останній простий множник (як 127^1 у наведеному прикладі); водночас, для деяких чисел (як-то $2500 = 2^2 \times 5^4$), за рахунок багатьох ділень на одне й те ж p у вкладеному циклі, наприкінці зовнішнього циклу `while(p<=sqrtN)...` виходить $n = 1$, і дописувати його у розкладення не треба й не можна.

У наведеному Pascal-коді, `factRes` — масив записів (структур) з полями `prime` (яке просте) та `degree` (у якій степені); `numFacts` означає реально використану кількість елементів цього масиву (різних простих

у розкладенні). У наведеному C++-коді, в якості масиву використовується `vector` стандартних пар, в яких `first` — яке просте, `second` — у якому степені. (Не варто сприймати це за рекомендацію завжди користуватися `pair`; якщо всяких різних пар багато, і `first` та `second` у різних місцях позначають різне — це погано, краще використати свою структуру/клас зі зрозумілими назвами полів. Але якщо плутанини нема, то можна й пари: писати мало, виходить коротко і зручно.)

Чи можна якось простіше, без масиву `factRes`? Залежить від обставин. Наприклад, якщо розкладення *лише* підставлятиметься у формулу з розд. 1.1, то можна спростити код до такого, як праворуч (і не потребува́ти окремого перетворення масиву в кількість дільників). Але якщо знадобиться використати розкладення якось інакше, цей код доведеться викинути й писати все заново, тоді як попередній не потребува́в би істотних змін.

```
totDivs:=1;
p:=2;
sqrtN:=round(sqrt(n));
while p<=sqrtN do begin
 if n mod p = 0 then begin
  currDeg:=0;
  repeat
   inc(currDeg);
   n := n div p;
  until n mod p <> 0;
  sqrtN:=round(sqrt(n));
  totDivs:=totDivs*(currDeg+1);
 end;
 inc(p);
end;
if n>1 then
 totDivs := totDivs*2;
```

Доводячи до абсурду, що краще: стандартні функції `sqrt` та `sin` у тому вигляді, як вони є, чи якби замість них були процедури `sqrt` та `sin`, які лише виводять обчислене значення на екран, не даючи можливості використати у виразах? От і побудова масиву `factRes` (а краще, повернення такого масиву як результату функції) дозволяє іншим частинам програми використовувати розкла́дення (результат) так, як їм треба, не потребу́ючи переписування функції розкладання (процесу).

Може здатися, ніби пробувати ділити на всі підряд чи́сла — погано, бо раз розклаємо на прості множники, то слід ділити лише на прості. Але перш, ніж цикл дійде до скла́деного дільника $c = q_1 \times q_2 \times \ldots \times q_k$, спочатку з n будуть прибрані менші $q_1, \ldots, q_k$ і зменшене n вже не буде кратним c. Тож з правильністю все гаразд. Що ж стосується швидкодії, то: *якщо* треба розкладати *багато* чисел n_i, можна знайти прості до $\sqrt{\max n_i}$ решетом Ератосфена (розд. 1.3) і отримати прискорення за рахунок уникнення пробних ділень на скла́дені. Але це виправдано, коли витрати на знаходження переліку простих відбуваються один раз, а прискорення розкладання відбувається багатократно. В інших випадках, і легше, і швидше пробувати ділити на всі підряд $p \leqslant \sqrt{n}$.

Насамкінець, тут розглянуті лише три застосування стандартної ідеї «перебирати до $\sqrt{n}$», які, хоч і скорочують перебір відносно найпримітивніших підходів, але в серйозній літературі вважаються «наївними» та повільними. Для справді великих чисел, $\sqrt{n}$ — теж забагато. В літературі чи Інтернеті можна знайти інші, ефективніші, методи перевірки простоти́ та розкладання на прості множники. Але всі вони значно складніші, і при цьому більшість таких тестів простоти́ та всі такі алгоритми розкладання ймовірнісні: з великою ймовірністю дають правильну відповідь, але можуть і помилятись (як правило, кажучи на скла́дене число, ніби воно просте). Тож повернемось до не надто ефективних, зате зрозумілих і гарантовано правильних методів.

1.3 Решето Ератосфена

1.3.1 Стандартна версія

Цей алгоритм знаходить усі прості чи́сла проміжку від 2 до деякого $maxN$, і робить це значно швидше, ніж перевірка кожного числа́ окремо алгоритмом з розд. 1.2.2.

Беремо масив з діапазоном індексів `[2..maxN]` (навіть якщо писати С-подібною мовою, де є традиція, щоб останній елемент мав індекс `[N-1]`, *у цьому* алгоритмі варто зробити масив розміром `maxN+1` і працювати з тими ж індексами від 2 до `maxN` включно). Спочатку заповнимо його значеннями `true`. Потім беремо значення 2, і всі елементи з індексами, кратними 2 (*крім само́го індексу 2*) встановимо у `false`:

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
було	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
стало	+	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−

Потім аналогічно беремо значення 3, і елементи з індексами, кратними 3 (але *не сам індекс 3*) встановимо у `false`:

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
було	+	+	−	+	−	+	−	−	+	−	−	+	−	−	+	−	−	+	−	−	+	−	−	+	−	−	+	−	−
стало	+	+	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−

Як бачимо, чи́сла проміжку, кратні 2 та/або 3 (крім самих 2 та 3), вже змінили позначки з `true` на `false`. Це і є інваріантом (зовнішнього) циклу: після завершення проходів по масиву при усіх p проміжку $2 \leqslant p \leqslant k$, `false`-ом позначені ті й тільки ті чи́сла, які, не будучи простими від 2 до k, мають серед своїх дільників чи́сла від 2 до k. Тому, коли алгоритм буде виконано до кінця, позначки `true` залишаться *лише на простих* числах (це стосується проміжку від елемента №2 до кінця масива; 0-му та 1-му елементам, якщо такі є, краще присвоїти `false`

десь окремо). Цього ще не сталося, бо бувають скла́дені чи́сла, кратні лише дільникам, більшим 3 (як-то 25), треба продовжувати.

Значення, кратні 4, перепозначати не треба, бо вони вже позначені як кратні 2. Якщо говорити про загальне правило, то запускати вкладений цикл позначення кратних чисел достатньо лише для простих чисел, тобто тих, які все ще не були перепозначені з `true` у `false`. (Якщо число вже перепозначене раніше, то в одному циклі з ним перепозначені й усі кратні йому, тож такий цикл не дасть нічого нового.)

У цьому прикладі, досить перепозначити чи́сла, кратні 5, і, коли 25 змінить позначку, позначеними `true` лишаться тільки прості чи́сла:

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
було	+	+	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−
стало	+	+	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−	−	−	+	−	−	−	−	−	+	−

А коли завершувати, якщо говорити не про приклад, а про загальне правило? Помітимо, що коли скла́дене число має різні прості дільники, то його позначка міняється на `false` при перегляді чисел, кратних *мінімальному* з цих дільників. Тобто, при великому p нема сенсу позначати $2\cdot p$, $3\cdot p$, …, $(p-1)\cdot p$ — усі вони мають дільники, менші p, отже, вже позначені; можна починати з p^2. А це звужує діапазони перебору обох циклів: зовнішній варто завершувати при перевищенні $\sqrt{maxN}$ (а не $maxN$), внутрішній починати з p^2 (а не $2\cdot p$).

Утримаємося від наведення повного коду, але поговоримо окремо про організацію внутрішнього циклу й окремо зовнішнього.

```
q:=p*p;
while q<= maxN do begin
  er[q]:=false;
  q:=q+p;
end;
```

```
for(int q=p*p; q<=maxN; q+=p)
  er[q]=false;
```

```
for i:=p to maxN div p do
  er[i*p]:=false;
```

Бажано, щоб внутрішній цикл, який позначає всі чи́сла, кратні поточному p, був організований так, як наведено угорі; трошечки гірше (зайве множення), але теж можна так, як наведено ліворуч трохи нижче. Але украй небажано всере́дині цього цикла перебирати кожнісіньке число й щоразу перевіряти, чи кратне воно p. Це *істотно* впливає на швидкодію.

Зовнішній же цикл треба організовувати приблизно так, як наведено праворуч.

```
for p:=2 to maxN do er[p]:=true;
sqrtN:=round(sqrt(maxN));
for p:=2 to sqrtN do
  if er[p] then // позначку не знято
    ...(внутр. цикл)...
```

Асимптотична оцінка кількості дій усього решета Ератосфена $\Theta(n\log\log n)$ (де $n=maxN$). Доведення са́ме цієї оцінки пропустимо (охочі можуть знайти у літературі чи Інтернеті), а наближена

оцінка $O(n \log n)$ (тобто, чи то $\Theta(n \log n)$, чи то менше) доводиться так. Завдяки тому, що внутрішній цикл не перебирає всі чи́сла до n, а відразу позначає лише кратні p, рухаючись з кроком p, внутрішній цикл має $\approx n/p$ ітерацій. Значить, сумарну (по всім ітераціям зовнішнього циклу) кількість ітерацій внутрішнього можна оцінити як приблизно

$$n/2 + n/3 + \cdots + n/n = n \times \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) \approx$$
$$\approx n \times \int_2^n \frac{1}{x}\,dx = n \times \left(\ln x\Big|_2^n\right) \approx n \ln n.$$

Тобто, оцінка $\Theta(n \log n)$ справедлива для менш ефективної версії, де і внутрішній цикл починається з $2p$, а не p^2, і запускається той внутрішній цикл для всіх, а не лише простих, p, і зовнішній цикл завершується аж при $p \approx n$, а не $p \approx \sqrt{n}$. Всі ці оптимізації (особливо, разом узяті) дають досить значне пришвидшення. Але загальна кількість дій усе ще більша n.

(Залежно від обставин, «вузьким місцем» алгоритму «решето Ератосфена» може бути час роботи, а може об'єм пам'яті.

Тим, хто пише мовою C++, варто знати, що можливі помітно різні витрати часу та пам'яті при використанні `vector<bool>` та інших способів подання масиву. У більшості версій STL, `vector<bool>` влаштований так, що витрачає на елемент всього один біт (а не байт), що дає можливість без зусиль економити пам'ять (можливо, не рівно у 8 разів, бо `vector` має свої накладні витрати). Але й доступ до таких «стиснутих» елементів виявляється повільнішим, ніж до елементів `vector<short>`, чи `vector<char>`, чи глобальних статичних масивів `bool`-ів. З іншого боку, менший розмір — більше шансів поміститися в оперативній пам'яті, а не swap-і (файлі підкачки), або у кеші, а не основній оперативній пам'яті. Тобто, тут неможливо дати універсальну пораду, і для незнайомих комбінацій компілятора, OS та «заліза» варто просто спробувати обидва варіанти.)

Що є остаточним результатом решета Ератосфена? Часто зручно вважати результатом сам утворений масив логічних значень `er`. Якщо потрібен все-таки перелік сами́х лише простих чисел — треба пройти по масиву `er` і вибрати (вивести як результат, чи занести в інший масив, тощо) всі ті чи́сла p, для яких `er[p]` усе ще `true`.

Назва «*решето* Ератосфена» зумовлена тим, що дію «позначити число як скла́дене» (у наведених фрагментах коду, `er[q]:=false`) сам Ератосфен (який жив у III–II ст. до н. е.) виконував проколюванням того місця, де до того було написане число q; тож, після виконання алгоритму дощечка з числами справді ставала решетом.

1.3.2 Варіант решета для проміжку з довільним початком

Розглянута у розд. 1.3.1 класична версія решета Ератосфена може працювати *лише для проміжків, що починаються з двійки*. А що робити, коли треба знайти всі прості на якомусь іншому проміжку «від A до B»? (Наприклад, «від 2000 до 3000».) Часто заявляють, ніби є лише можливість вибрати один з двох варіантів — або класичне решето Ератосфена від 2 до B (непотрібні прості, менші A, все одно знайти, просто потім відкинути), або перевіряти кожне число від A до B окремо способом з розд. 1.2.2. Що ж, немало ситуацій, де справді варто вибирати з цих варіантів. Але при досить великому B кількість елементів проміжку $B-A+1$ може бути водночас і надто малою, щоб легко змиритися з тим, щоб зберігати в пам'яті також і непотрібну частину до $A-1$, і надто великою, щоб перевіряти кожен елемент окремо. Наприклад, $A=10^{10}$, $B=10^{10}+10^7$. Са́ме орієнтуючись на такі випадки і пропонується модифікувати (змінити) решето Ератосфена.

Будемо зберігати елементи з індексами від 0 до $(B-A)$ (обидві межі включно), маючи на увазі, що 0-й елемент відповідає числу A, 1-й — числу $A+1$, і так далі, по $(B-A)$-й елемент, що відповідає B.

Будемо працювати з двома вкладеними циклами, подібними до стандартного алгоритму решета Ератосфена, але:

1. зовнішній цикл перебирає p, як і класичне решето, на проміжку $2\leqslant p\leqslant\sqrt{B}$, але *не* намагаючись пропускати скла́дені (для $p<A$ таку інформацію і бра́ти-то нема звідки);
2. внутрішній перебирає чи́сла, кратні p, з урахуванням і обмежень $p^2\leqslant q\leqslant B$ з класичного решета Ератосфена, і обмеження $A\leqslant q$:

```
q:=(((A-1) div p)+1)*p;
// мінімальне число, одночасно кратне p та більше-рівне A
if q < p*p then q:=p*p;
while q<=B do begin // q перебирає фактичне значення числA,
  er[q-A]:=false; // а відповідає цьому числу елемент er[q-A]
  q:=q+p;
end
```

Інваріант цикла той самий, що у класичній версії; ме́жі перебору p теж; запуск внутрішнього циклу для всіх p чи лише для простих впливає лише на швидкодію, не на правильність. Отже, все правильно.

Запуски внутрішнього цикла при всіх підряд (а не лише простих) p, а також громіздкувате визначення початкового значення q та наявність зайвого віднімання "`q-A`" роблять цю модифікацію дещо повільнішою

за класичне решето при невеликих A. (Так що замінювати класичне решето Ератосфена цією модифікацією при невеликих A не варто.) Асимптотична ж оцінка непогана: $O\big((B-A)\times\log B+\sqrt{B}\big)$ (доведення аналогічне доведенню для класичної версії решета, деталі додумайте самостійно; «$+\sqrt{B}$» виражає, що кожна ітерація зовнішнього цикла займає час, навіть якщо внутрішній цикл фактично не виконується; цей доданок істотний для малих $B-A$ при великих B).

Ще можна окремо побудувати класичне решето Ератосфена до $\sqrt{B}$. Це займе $\Theta(\sqrt{B})$ додаткової пам'яті та $O(\sqrt{B}\cdot\log\log\sqrt{B})$ додаткового часу, зате відновить можливість запускати вкладений цикл для проміжку від A до B не для всіх p, а лише для простих, і тим зменшить множник при $(B-A)$. Але чи це справді даватиме прискорення, чи навпаки — досить заплутано й майже непередбачувано, залежить і від співвідношення між $\sqrt{B}$ та $B-A$, і від того, як співвідносяться об'єм пам'яті, потрібний лише на масив від A до B, об'єм пам'яті, потрібний на обидва масиви, та скільки доступно пам'яті у кеші, в основній оперативній пам'яті та в файлі підкачки (swap).

Далі наведено приклад для діапазону від 2015 до 2029 (багатократно, у різних рядках, зображено стани одновимірного масиву в різні моменти часу). Зверніть увагу, що позначка числá 2021 змінилася аж при $p=43\approx45\approx\sqrt{2029}$, хоч до того й була велика кількість ітерацій, на яких нічого не змінювалося. Це тому, що $2021=43^1\times47^1$ (43 та 47 — прості), тож 43 є мінімальним дільником. Якби було $B=p^2$ (p просте), то позначка могла б змінитися й аж при $p=\sqrt{B}$; отже, припиняти зовнішній цикл раніше $\sqrt{B}$ неправильно (навіть якщо p вже здається надто великим у порівнянні з довжиною проміжку $B-A+1$).

індекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
число	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2029
спочатку	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
після $p=2$	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
після $p=3$	+	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−
після $p=4$	+	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−
після $p=5$	−	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−
після $p=6$	−	−	+	−	−	−	+	−	+	−	−	−	+	−	+	−
після $p=7$	−	−	+	−	−	−	+	−	−	−	−	−	+	−	+	−
(жоден з проходів при $8\leqslant p\leqslant42$ не змінює вміст масиву)																
після $p=43$	−	−	+	−	−	−	−	−	−	−	−	−	+	−	+	−
після $p=44$	−	−	+	−	−	−	−	−	−	−	−	−	+	−	+	−
після $p=45$	−	−	+	−	−	−	−	−	−	−	−	−	+	−	+	−

1.4 НСД, НСК та алгоритм Евкліда

Означення НСД та НСК дані на початку лекції.

Для НСД та НСК виконується властивість

$$\text{НСД}(a,b)\times\text{НСК}(a,b)=a\times b.$$

Обидва поняття НСД та НСК можна узагальнити на сукупності кількох чисел. Зокрема, $\text{НСД}(a_1,a_2,\ldots,a_k)$ — найбільше серед чисел, на які діляться одночасно всі a_1, a_2, ..., a_k. Виконується властивість

$$\text{НСД}(a_1,a_2,\ldots,a_k)=\text{НСД}\Big(\text{НСД}(\ldots\text{НСД}(a_1,a_2),\ldots),a_k\Big),$$

причому a_1, a_2, ..., a_k можна переставляти у будь-якому порядку. Означення та властивості НСК кількох чисел повністю аналогічні.

НСД та НСК можна знайти через розкладення на прості множники: у НСД вибирають мінімум показника степені кожного простого, у НСК — максимум. Наприклад, для $8=2^3$ та $10=2^1\times5^1$: у розкладенні 10 наявне 5, тож запишемо $8=2^3\times5^0$; $\text{НСД}(8,10)=2^{\min(3,1)}\times\times5^{\min(0,1)}=2^1\times5^0=2$; $\text{НСК}(8,10)=2^{\max(3,1)}\times5^{\max(0,1)}=2^3\times5^1=40$. Але це зручно здебільшого для доведень (включно з доведеннями властивостей, згаданих у попередніх абзацах), або коли чи́сла вже розкладені на множники з якоюсь іншою метою. А коли треба шукати НСД чи НСК чисел, поданих звичайним чином, краще користуватися алгоритмом Евкліда, який і легше писати, і швидше виконується.

Класична версія алгоритма Евкліда така: *«Поки чи́сла не рівні між собою, віднімати з більшого менше»*. Наприклад, для тих самих 8 і 10: $(8,10)\to(8,2)\to(6,2)\to(4,2)\to(2,2)$. У цієї класичної версії є недолік: якщо одне з чисел *значно* більше іншого, можна дуже багато разів віднімати з великого числа́ одне й те са́ме мале́. Тому́ *сучасна версія алгоритму Евкліда* замінює віднімання на взяття залишку; внаслідок цього, умову «чи́сла стали рівними» доводиться змінити на «одне з чисел стало нулем». Що для тих самих 8 і 10 буде $(8,10)\to(8,2)\to\to(0,2)$, а в загальному вигляді — наприклад, так:

```
function gcd(a,b:QWord):QWord;
Begin // gcd -- greatest common divisor, тобто НСД
  while (a>0) and (b>0) do
    if a>b then a:=a mod b // у термінах C-подібних мов, a%=b
           else b:=b mod a; // та b%=a відповідно
  gcd:=a+b // одне (невідомо яке) 0, тож вертається значення іншого
  // (у термінах C-подібних мов, return a+b)
End;
```

Кількість ітерацій цього алгоритма має порядок $O(\log(a+b))$, бо за *дві підряд* ітерації кожна зі змінних a та b зменшується щонайменше вдвічі (як правило, ще сильніше). Деталі придумайте самостійно або знайдіть в Інтернеті чи літературі.

Якщо змінити тип з `QWord` на знаковий і передати сюди від'ємне число (одне з двох чи обидва), результат `a+b` може й не бути НСД. Тож якщо (всупереч сказаному раніше «розглядаємо лише натуральні чи́сла») все ж треба працювати з числами довільних знаків — краще сюди передавати модулі чисел, а знаки враховувати деінде. Аналогічно, якщо не влаштовує, що для `gcd(0,0)` отримується результат 0.

Тепер щодо НСК. З урахуванням $\text{НСК}(a,b) = \frac{a\times b}{\text{НСД}(a,b)}$, НСК можна виразити як `(a*b) div gcd(a,b)`. Але тут можлива прикра ситуація, коли $\text{НСК}(a,b)$ поміщається у типі, а при обчисленні виникає переповнення, бо `a*b` не поміщається. Що ж, НСД є спільним дільником; отже, $\frac{b}{\text{НСД}(a,b)}$ є цілим числом; отже, можна переставити дужки як `a * (b div gcd(a,b))`: і результат не зміниться, і переповнення тепер буде тільки якщо саме́ $\text{НСК}(a,b)$ не поміщається у типі. Остато́чно:

```
function lcm(a,b:QWord):QWord;
Begin // lcm -- least common multiple, тобто НСК
  lcm := a * (b div gcd(a,b));  // return a * (b / gcd(a,b));
End;
```

Алгоритм Евкліда не те, щоб складни́й, але не очевидний: *чому* виконання са́ме таких дій призводить до знаходження са́ме НСД? Доведемо це (для цілих невід'ємних a, b, хоча б одне з яких не 0). Цьому доведенню присвячений увесь текст звідси до кінця цього розд. 1.4.

Якщо одне зі значень a чи b дорівнює 0, інше $\neq 0$, функція вертає значення ненульового іншого; це НСД: на нього діляться і 0, і те інше (саме́ на себе), а більшого за себе дільника додатне число мати не може.

Надалі вважаємо гарантованим, що $a>0$, $b>0$.

Розглянемо випадок $a>b>0$. Позначимо $a \bmod b$ як c. (Тут і далі мається на увазі, що позначки c, d, p, q та інші вводяться лише у доведенні; переписувати програму, вводячи такі змінні туди, не варто.) Тоді a можна подати як $a=b\times d+c$, де $d=a \operatorname{div} b$ є натуральним. Нехай деяке p є спільним дільником a та b (не обов'язково найбільшим). Тобто, $a_1=a/p$ та $b_1=b/p$ натуральні. Тоді, $c=a-b\times d=a_1\times p-b_1\times p\times d=$ $=(a_1-b_1\times d)\times p$, тобто $c \vdots p$. Аналогічно, нехай деяке q (можливо як $q=p$, так і $q\neq p$) є спільним дільником b та c, тобто $b_2=b/q$ та $c_2=c/q$

є цілими. Тоді $a = b \times d + c = b_2 \times q \times d + c_2 \times q = (b_2 \times d + c_2) \times q$, тобто $a \vdots q$. Висновок: дія `a:=a mod b`, змінюючи `a`, не змінює сукупності спільних дільників (якби спільні дільники могли втрачатися, не було б твердження «$c \vdots p$», з'являтися — «$a \vdots q$»).

При $0 < a \leqslant b$, можна провести такі сáмі міркування для b, a та $c = b \bmod a$. Тому висновок попереднього абзацу можна посилити: виконання усього циклу `while (a>0) and (b>0) do...` не змінює сукупності спільних дільників. А раз не змінюється сукупність, то не змінюється й найбільший з них (НСД). Тобто, є інваріант: на всіх ітераціях циклу, НСД поточних `a` та `b` дорівнює НСД початкових `a` та `b`.

Поєднуючи цю незмінність НСД з раніше показаним «коли наприкінці робóти алгоритму одне з чисел стає 0, НСД визначається правильно», отримуємо: результат алгоритма Евкліда дорівнює НСД початкових значень `a` та `b`. Що й потрібно було довести.

Але досі доведено *лише* те, що *якщо* алгоритм завершує роботу, то результат правильний. Те, що він при будь-яких $a > 0$, $b > 0$ справді завершить роботу, а не зациклиться, треба доводити окремо. Це легко: при $a > b > 0$, дія `a:=a mod b` замінює значення `a` на ціле невід'ємне, строго менше старого `a`; інакше (при $b \geqslant a > 0$), дія `b:=b mod a` замінює значення `b` на ціле невід'ємне, строго менше старого `b`; тобто, на кожній ітерації циклу, сума `a+b`, лишаючись натуральною, стає строго меншою. А це не може тривати вічно.

1.5 Основи модульної арифметики

І в олімпіадних, і в деяких практичних задачах (наприклад, у шифруваннях) часом виникає потреба «знайти таку-то величину *за модулем p*». У стандартному (єдиному розглянутому тут) варіанті, це означає, що дано (або треба придумати самостійно) деяку формулу з цілочисельним результатом, і треба отримати залишок від ділення того цілочисельного результату формули на деяке натуральне p.

Математичні формули (вирази), зазвичай, задають, в якому порядку виконувати деякі відносно прості дії (як-то додавання, множення, ...). От і почнемо з аналізу дії додавання $(a + b) \bmod p$.

Якщо позначити $a \operatorname{div} p$ як a_1, $a \bmod p$ як a_2, $b \operatorname{div} p$ як b_1 та $b \bmod p$ як b_2, то можна помітити, що $(a + b) \bmod p = (a_1 \cdot p + a_2 + b_1 \cdot p + b_2) \bmod p = \big((a_1 + b_1) \cdot p + (a_2 + b_2)\big) \bmod p = (a_2 + b_2) \bmod p$ (останнє перетворення спирається на те, що доданок, кратний p,

не впливає на залишок). Тобто, для знаходження залишку суми можна взяти залишок кожного доданка окремо, додати ці залишки й ще раз взяти залишок, і результат дорівнюватиме залишку від суми.

Яка користь робити три операції mod замість однієї? Головним чином та, що значення стають менші. Що само по собі не завжди корисне: поки дії без проблем відбуваються в межах стандартних типів, комп'ютеру однаково, які чи́сла додавати. Але в тім-то й справа, що «поки ...». Якщо підхід «рахувати по-простому й наприкінці один раз взяти mod» виводить за межі стандартних типів даних, а заміна $(a+b) \bmod p$ на $(a \bmod p + b \bmod p) \bmod p$ дозволяє лишитися в цих межах, то користь може бути значною.

(Мов програмування, де є готова довга арифметика (Python, Java), це стосується у дещо меншій мірі. Буває (особливо, мовою Python, де перехід до довгої арифметики автоматичний без зусиль програміста, а операція "%" повільна), що краще дозволити проміжні великі значення, з якими працюватиме вбудована довга арифметика. Але буває й так, що відмова від проміжного взяття залишків дає настільки довгі чи́сла, що це істотно впливає на об'єм пам'яті та час робо́ти, і краще все-таки частіше робити mod. Або, у Java може бути легше акуратно вчасно бра́ти mod, аби лиш не зв'язуватися з незручним класом `BigInteger`. А ще бувають ситуації, де важливо, що Python та Java — повільні мови, і слід вибрати швидшу мову, в якій довгої арифметики нема.)

З множенням усе аналогічно: ввівши аналогічні позначення a_1, a_2, b_1, b_2, можна отримати аналогічне перетворення $(a \cdot b) \bmod p = \big((a_1 \cdot p + a_2) \times (b_1 \cdot p + b_2)\big) \bmod p = \big(a_1 \cdot b_1 \cdot p^2 + a_1 \cdot b_2 \cdot p + a_2 \cdot b_1 \cdot p + a_2 \cdot b_2\big) \bmod p = (a_2 \cdot b_2) \bmod p$, тобто $((a \bmod p) \cdot (b \bmod p)) \bmod p$.

З відніманням усе здається аналогічним: ніби, $(a-b) \bmod p = \big((a_1 \cdot p + a_2) - (b_1 \cdot p + b_2)\big) \bmod p = \big((a_1 - b_1) \cdot p + (a_2 - b_2)\big) \bmod p = (a_2 - b_2) \bmod p$... але правильність останнього перетворення насправді залежить від того, як рахувати mod для від'ємних чисел. (Те, що на початку статті сказано, що працюємо лише з натуральними, не вирішує цю проблему; наприклад, $(14545345351616553 - 4531351504654145) \bmod 10 = 10013993846962408 \bmod 10 = 8$ цілком собі рахувалося в межах натуральних, поки ми за́раз не влізли туди додатковими mod-ами й не почали рахувати те са́ме як $\big((14545345351616553 \bmod 10) - (4531351504654145 \bmod 10)\big) \bmod 10 = (3-5) \bmod 10 = (-2) \bmod 10 = ???$.)

У відповідних розділах математики прийнято, що $a \bmod b$ при додатних b завжди, незалежно від знаку a, невід'ємне (наприклад, $(-2) \bmod 10 = +8$), а в мовах програмування має місце певна плутанина. Мовою Python гарантований щойно згаданий підхід (`-2%10` дає `8`),

але більшістю з решти мов буде **-2**, причому багатьма з них це ще й не гарантовано (чи то -2, чи то 8 залежно від версії компілятора, його налаштувань, тощо). Тому́ пропонується не розраховувати на компілятор і переписати формулу віднімання у модульній арифметиці як

$$(a - b) \bmod p = (p + (a \bmod p) - (b \bmod p)) \bmod p$$

(наявність "$p+$" робить лівий аргумент останнього mod гарантовано додатним, і ця проблема нівелюється).

З діленням ситуація ще складніша, й розписувати її *детально* — далеко за межами цієї статті. Коротко відзначимо, зокрема, таке:

1. «Поділити a на b» (в арифметиці за модулем p, при $0 \leqslant a < p$, $0 < b < p$) зазвичай розуміють як «підібрати таке c, щоб виконувалося $(b \cdot c) \bmod p = a$». Наприклад, за модулем 7, можна вважати $1/5 = 3$, бо виконується $(5 \cdot 3) \bmod 7 = 15 \bmod 7 = 1$. Наскільки часто такий результат збігається з тим, щоб спочатку поділити й потім уперше взяти залишок — це вже інше питання.
2. Коли p скла́дене, ця дія перестає бути однозначною. Наприклад, при $p = 10$, можна взяти $6/2 = 3$, і все виходить $((3 \cdot 2) \bmod 10 = 6 \bmod 10 = 6)$, але можна узяти й $6/2 = 8$, і все теж виходить $((8 \cdot 2) \bmod 10 = 16 \bmod 10 = 6)$.
3. Коли p просте, $a/b = (a \cdot b^{p-2}) \bmod p$ (доведення знайдіть деінде).

Замість усього, досі згаданого про ділення, часто буває кориснішою (й більш схожою на «формулу знаходження залишку від частки») зовсім інша властивість: *«якщо з якихось міркувань відомо, що, при цілому $b \geqslant 1$, частка $\frac{a}{b}$ є цілою, то* $\dfrac{a}{b} \bmod p = \dfrac{a \bmod (p \cdot b)}{b}$*»*.

Позначимо ціле $\frac{a}{b}$ як k і виділимо в ньому $k_1 = k \operatorname{div} p$ та $k_2 = k \bmod p$. Тоді $a = k \cdot b = \big(k_1 \cdot p + k_2\big) \cdot b = (k_1 \cdot p \cdot b) + (k_2 \cdot b)$. Оскільки $(k_1 \cdot p \cdot b) \vdots (p \cdot b)$, цей доданок не впливає на залишок, і можна стверджувати, що $a \bmod (p \cdot b) = (k_2 \cdot b) \bmod (p \cdot b)$; оскільки, за побудовою, $(0 \leqslant k_2 < p)$, то домноження на додатне b дає $(0 \leqslant k_2 \cdot b < p \cdot b)$. Звідси, $(k_2 \cdot b) \bmod (p \cdot b) = (k_2 \cdot b)$ (коли лівий аргумент mod у проміжку від 0 до правого аргумента не включно, результат mod дорівнює лівому аргументу). Тобто, $a \bmod (p \cdot b) = k_2 \cdot b$, й лишається тільки поділити обидві частини на b, щоб отримати $k_2 = k \bmod p = \frac{a}{b} \bmod p$. Кінець доведення.

1.6 Кількість не кратних на проміжку

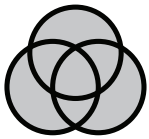
Розглянемо ще одне (дещо менш стандартне й менш важливе) питання теорії чисел: *«Скільки чисел у проміжку від A до B (обидві межі включно) не діляться націло ні на C, ні на D, ні на E?»*

Спочатку розглягнемо простішу задачу *«Скільки чисел від 1 до B кратні C?»*. Її розв'язати неважко: кратними C є чи́сла C, $2C$, $3C$, …, тож на проміжку від 1 до B таких чисел рівно $B \operatorname{div} C$.

У розв'язаній задачі чимало відмінностей від потрібної: від 1, а не від A; «кратні», а не «не кратні»; само́му лише C, а не трьом різним числам C, D, E. Але всі ці відмінності вдасться врахувати.

Кількість чисел від 1 до B, *не* кратних C, дорівнює $B - (B \operatorname{div} C)$, бо кількість не кратних дорівнює кількості усіх мінус кількість кратних. Використаємо це (не конкретну формулу, а спостереження, що кількість не кратних = кількість усіх мінус кількість кратних).

Наступна задача — *«Скільки чисел від 1 до B кратні хоча б одному з чисел C, D або E?»* (Адже «не діляться націло ні на C, ні на D, ні на E» є точною протилежністю (запереченням) са́ме такої умови. Кому це не очевидно — може, наприклад, порівняти цю ситуацію із *законом де Моргана*, який стверджує, що `not(p or q)` дорівнює `(not a) and (not b)`, де `p` та `q` — значення чи вирази типу `boolean`.)



«Кратні хоча б одному з чисел C, D або E» має багато спільного з такою задачею. Нехай є три круги X, Y, Z, які можуть перетинатися. Все, що потрапило до цих кругів, замальовується (причому щільність однакова і для просто кругів, і для частин, де круги накладаються).

Тоді площу замальованої частини можна порахувати так: спочатку порахуємо суму площ кругів; перетини (спільні частини) кругів були враховані двічі, тому їх треба відняти; перетин усіх трьох кругів спочатку тричі додали, а потім тричі відняли — отже, його треба додати. Або, все разом: $S(X \cup Y \cup Z) = S(X) + S(Y) + S(Z) - S(X \cap Y) - S(X \cap Z) - S(Y \cap Z) + S(X \cap Y \cap Z)$ (де $\cap$ — перетин, тобто спільна частина; $\cup$ — об'єднання, тобто все, що належить хоча б одному). Цю рівність називають *принципом включень та виключень*. Цей принцип може бути узагальненим на більш, ніж три сукупності, охочі можуть знайти деталі в літературі чи Інтернеті. Але чисел C, D, E рівно три, тож потрібна сама ця формула, без узагальнень.

Очевидно, «$X \cap Y$, тобто все, що належить X і Y одночасно» перетворюється у «кількість чисел, кратних водночас і C, і D». А які чи́сла

кратні і C, і D? Очевидно, ті, які кратні НСК(C, D). Лишається узяти функцію `lcm` з розд. 1.4 та пристосувати формулу включень та виключень до цієї задачі як "`(B div C) + (B div D) + (B div E) - (B div lcm(C,D)) - (B div lcm(C,E)) - (B div lcm(D,E)) + (B div lcm(lcm(C,D),E))`" — і задача *«Скільки чисел від 1 до B кратні хоча б одному з чисел C, D або E?»* розв'язана.

Якщо відняти кількість, знайдену за цією формулою, із загальної кількості чисел B, отримаємо кількість чисел від 1 до B, не кратних ні C, ні D, ні E. Так що тепер задача відрізняється від потрібної *лише* тим, що вміємо знаходити на проміжку від 1 до B, а треба від A до B. А для таких ситуацій є стандартний прийом: кількість на проміжку від A до B дорівнює кількості на проміжку від 1 до B мінус кількість на проміжку від 1 до $(A-1)$. (Правильний стиль програмування у такій ситуації — написати знаходження на проміжку від 1 до N один раз, оформити як функцію, і двічі ви́кликати з різними аргументами.)

Програма зводиться до кількох формул, кількість застосувань яких невелика й не залежить від значень вхідних даних. Єдине місце, де залежність часу роботи від значень все-таки є — алгоритм Евкліда. Отже, оцінка складності алгоритму становить $O(\log C + \log D + \log E)$.

2 Задачі основного дня (16.10.2019)

Цей комплект задач доступний для on-line перевірки як змагання №69 сайту `ejudge.ckipo.edu.ua`. Там можна побачити також повні формулювання умов (у збірнику, задля економії місця, вони скорочені).

Задача A «Перелік дільників»

Для натурального числа́ N, виведіть у порядку зростання всі його різні натуральні дільники.

Вхідні дані. Єдине натуральне число N, $1 \leqslant N \leqslant 1234567891011$.

Результати. Послідовність усіх різних натуральних дільників, у порядку зростання. Виводити в один рядок, розділяючи пропусками.

Приклади.

Вхід	Рез-ти
9	1 3 9
120	1 2 3 4 5 6 8 10 12 15 20 24 30 40 60 120

Розбір задачі.

Суть розібрана у розд. 1.2.1. Приклад конкретної реалізації:

```
{$mode delphi}
var dividers : array[1..1000000] of longint;
    N : int64;
    sqrt_N, i, k : longint;
BEGIN
  readln(N);
  sqrt_N := round(sqrt(N*1.0));
  k:=0;
  for i:=1 to sqrt_N do begin
    if N mod i = 0 then begin
      k:=k+1;
      dividers[k]:=i;
      write(i,' ');
    end;
  end;
  if int64(dividers[k])*int64(dividers[k]) = N then
    k:=k-1;
  for i:=k downto 1 do
    write(N div dividers[i], ' ');
  writeln
END.
```

Варто відзначити такі моменти цього коду:

1. `N` мусить бути 64-бітовим, але решту величин зручніше лишити 32-бітовими.
2. При взятті кореня перестраховочно написано «`*1.0`» бо *деякі* компілятори не дають бра́ти `sqrt` з типу `int64` (насправді можна й не писати, бо з `fpc` на `ejudge.ckipo.edu.ua` цієї проблеми нема). Аналогічна проблема має місце і в більшості версій C/C++.
3. Для масиву `dividers`, розмір мільйон узятий з величезним запасом, насправді досить 7000; але правильно оцінити цю кількість дуже складно, тож якщо не знати, то краще взяти з запасом (але враховуючи обмеження пам'яті).
4. Відлагоджуючи розв'язок цієї задачі, варто перевірити, чи правильно програма враховує такі випадки:
   (а) $\sqrt{N}$ цілий і є одним (а не двома) з дільників, як 6 для 36;
   (б) $\sqrt{N}$ не цілий, але є два дільники близько до $\sqrt{N}$, як 6 і 7 для 42;
   (в) дільників, близьких до $\sqrt{N}$, нема.

Задача B «Перевірка на простоту́–1»

Напишіть програму, яка знайде усі підряд, у порядку зростання, прості чи́сла у проміжку від A до B (обидві межі включно).

Вхідні дані. У єдиному рядку через пробіл задані два натуральні числá A та B, які є межами проміжку. Обмеження:

- $1 \leqslant A$;
- $B \leqslant 10^{12}$;
- $A \leqslant B \leqslant A+100$.

Результати. Виведіть усі прості чи́сла проміжку, кожне у окремому рядку. Якщо буде введений проміжок, що не містить жодного простого числá, слід нічого не виводити (навіть символа завершення рядка).

Вхід	Рез-ти
2 5	2 3 5
4 4	

Розбір задачі.

Для всіх чисел від A до B (їх не більше 101) можна застосувати перевірку з розд. 1.2.2. Для чисел такого розміру потрібен 64-бітовий тип (`int64` або `QWord` у Pascal, `long long` у C/C++); мовою Pascal цикл `for n:=a to b do...` технічно несумісний з цими типами, тому доводиться виражати це саме через `while` (іншими мовами такої проблеми нема). Ще слід не забути, що число 1 не є простим.

Як альтернативний спосіб, задачу можна розв'язати також модифікацією решета Ератосфена з розд. 1.3.2; оскільки за один запуск слід перевіряти хоч відносно невеликий, але проміжок, той спосіб теж фактично проходить усі тести. (Що, втім, не варто сприймати, ніби спосіб з розд. 1.3.2 в усіх смислах кращий; спосіб з розд. 1.2.2 і простіший для написання, і, якби на простоту́ перевіряли окремі чи́сла, а не проміжки, був би швидшим.) Само собою, класичне решето Ератосфена тут ніяких шансів не має, бо B може бути дуже великим.

Задача C «Перевірка на простоту́–2»

Задача відрізняється від попередньої *лише* обмеженнями:

- $1 \leqslant A$;
- $B \leqslant 10^{7}$;
- $A \leqslant B \leqslant A+10^{6}$.

Розбір задачі.

Цю задачу доцільно вирішувати решетом Ератосфена. Наприклад, класичною його версією з розд. 1.3.1 знайти прості фактично на проміжку від 2 (а не від A) до B, а умову «від A» врахувати вже при виведенні результату. Тут теж слід не забути, що число 1 не є простим.

Модифікація решета Ератосфена з розд. 1.3.2 при малих A повільніша за класичний варіант, і тут може пройти чи не пройти залежно від того, наскільки добре оптимізувати розв'язок у деталях та якою мовою програмування його писати.

Спосіб з розд. 1.2.2 тут категорично непридатний (за часом).

Задача D «Перевірка на простоту́–3»

Задача відрізняється від двох попередніх *лише* обмеженнями:

- $1 \leqslant A$;
- $B \leqslant 10^{10}$;
- $A \leqslant B \leqslant A+10^5$.

Розбір задачі.

Цю задачу доцільно вирішувати модифікованим варіантом решета Ератосфена з розд. 1.3.2. Класичне решето з розд. 1.3.1 тут категорично непридатне за обсягом пам'яті, а спосіб з розд. 1.2.2 — за часом.

Задача E «Кількість дільників факторіала»

Напишіть програму, що за заданим натуральним числом N обчислюватиме кількість дільників $N!$ (факторіалу числа́ N).

Вхідні дані. Єдиний рядок містить одне ціле число N ($1 \leqslant N \leqslant 45$).

Результати. Виведіть єдине невід'ємне ціле число — знайдену кількість дільників числа́ $N!$

Вхід	Рез-ти
4	8

Примітка. При $N=4$, $N! = 4 \times 3 \times 2 \times 1 = 24$. Його дільники: 1, 2, 3, 4, 6, 8, 12, 24. Їх 8 штук.

Розбір задачі.

Основною складовою тут є формула вираження кількості дільників через відоме розкла́дення (розд. 1.1). Але рахувати $N!$, а потім розкладати його згідно розд. 1.2.3 — не найкраща ідея. Зокрема, тому́, що $45! \approx 1{,}2 \cdot 10^{56}$, тобто не помішається у стандартні цілочисельні типи.

Тобто, якщо є готова довга арифметика, то «спочатку порахувати $N!$, потім розкласти на прості множники» — прийнятний спосіб для обмежень «$N \leqslant 45$, час виконання до 1 сек». Цілком *можна* рівно це й реалізувати мовою Python чи Java; в принципі, найкорисніше було б зробити цю задачу і так, як щойно сказано, і так, як пояснено далі. Інше питання, що ця задача са́ме з такими обмеженнями взята з IV (фінального) етапу Всеукраїнської олімпіади з інформатики 2006 р., де жодна з мов із вбудованою довгою арифметикою не була доступна учасникам.

Розглянемо два способи, що не потребу́ють довгої арифметики. Вони обидва істотно спираються на те, що розкласти треба не довільне число, а $N!$, тож гарантовано, що усі прості множики будуть $\leqslant N$.

Спосіб перший. Заведемо масив з діапазоном індексів до N включно і будемо утворювати в кожному `i`-му елементі степінь відповідного числа́, *накопичуючи* сумарні кількості входжень цих множників у чи́сла 2, 3, …, N: $2 = 2^1$ додає 1 в елемент №2; $3 = 3^1$ додає 1 в елемент №3; $4 = 2^2$ додає 2 в елемент №2; $5 = 5^1$ додає 1 в елемент №5;

```
for p:=2 to n do divcnt[p]:=0;
for i:=2 to n do begin
  nn:=i;
  for p:=2 to i do
    while nn mod p = 0 do begin
      Inc(divcnt[p]);
      nn:=nn div p;
    end;
end;
res:=1;
for p:=2 to n do
  res:=res*(divcnt[p]+1);
```

```
res:=1;
for p:=2 to n do begin
  <перевірили, чи p просте>
  if <просте> then begin
    divcnt:=0;
    for i:=p to n do begin
      iii:=i;
      while iii mod p = 0 do begin
        iii:=iii div p;
        inc(divcnt)
      end
    end;
    res:=res*(divcnt+1)
  end;
end;
```

$6 = 2^1 \cdot 3^1$ додає по 1 в кожен з елементів №2 та №3; тощо. Праворуч наведено приклад станів такого масиву (при $N = 10$) в різні моменти часу (після чергового i).

	2	3	4	5	6	7	8	9	10
спочатку	0	0	0	0	0	0	0	0	0
після $i=2$	1	0	0	0	0	0	0	0	0
після $i=3$	1	1	0	0	0	0	0	0	0
після $i=4$	3	1	0	0	0	0	0	0	0
після $i=5$	3	1	0	1	0	0	0	0	0
після $i=6$	4	2	0	1	0	0	0	0	0
після $i=7$	4	2	0	1	0	1	0	0	0
після $i=8$	7	2	0	1	0	1	0	0	0
після $i=9$	7	4	0	1	0	1	0	0	0
після i=10	8	4	0	2	0	1	0	0	0

В елементах, індексами яких є скла́дені чи́сла, лишаються нулі, але це не заважає використати формулу $(m_1+1) \times (m_2+1) \times \ldots \times (m_k+1)$ з розд. 1.1, навіть не розрізняючи скла́дені від простих (зайві домноження на $0+1=1$ на результат не вплинуть). Код наведено нагорі цієї сторінки ліворуч.

Спосіб другий. Можна спочатку порахувати серед усіх множників факторіала від 2 до N загальну кількість простих дільників 2, потім загальну кількість множників 3, і т. д., підставляючи все це негайно у формулу $(m_1+1) \times (m_2+1) \times \ldots \times (m_k+1)$. Наприклад, при $N=10$ буде сказано, що 2 має 1 двійку, 4 — 2 двійки, 6 — 1 двійку, 8 — 3 двійки, 10 — 1 двійку, всього $1+2+1+3+1=8$ штук двійок; 3 має 1 трійку, 6 — 1 трійку, 9 — 2 трійки, всього $1+1+2=4$ штуки трійок; і так далі.

Якщо перевіряти, які чи́сла від 2 до N прості й запускати наведені далі дії лише для простих, в принципі можна обійтися без масивів (втім, це сумнівне досягнення). Реалізацію такого підходу можна бачити нагорі цієї сторінки праворуч (туди слід додати перевірку простоти, бажано способом з розд. 1.2.1; ну, щоб без масивів...).

Якщо не знати формули з розд. 1.1, задачу важко розв'язати повністю, хіба що придумати ту формулу самостійно. Але, якщо оцінювання передбачає часткові бали за проходження частини тестів, можна писати явне обчислення факторіала (хоча б для $N \leqslant 20$, де $N!$ поміщається у тип `int64`) і знаходити кількість дільників аналогічно задачі A.

(Для $19! \approx 1{,}2 \cdot 10^{17}$ та $20! \approx 2{,}4 \cdot 10^{18}$, $\sqrt{N!}$ теж багато; але код, який це обчислює, можна виконати на своєму комп'ютері (де можна й почекати кілька хвилин), а на сервер надіслати вже програму, що містить масив з готовими результатами. Цей прийом називають *precalc*; на більшості олімпіад він дозволений. Інша справа, що мало в яких задачах аж настільки мала́ сукупність можливих вхідних даних, а проти великих таблиць precalc-а успішно борються, обмежуючи розмір вихідного файлу розв'язку, наприклад до 32 Кб.)

Ручний аналіз результатів цього неповного розв'язку може бути корисним і для самостійного придумування формули. Намагання аналізувати все суто вручну легко може схилити до помилкової думки «$1! = 1$ має 1 дільник, $2! = 2$ має 2 дільники, $3! = 6$ має 4 дільники, $4! = 24$ має 8 дільників, тощо — отже, відповіддю є 2^{N-1}». Це не правда, а збіг обставин са́ме для цих чисел, Неповний же розв'язок, принаймні, покаже «$6! = 720$ має 30 дільників (а не 32)», що отримати вручну вже важко.

Задача F «Дільники на проміжку–1»

Напишіть програму, яка знайде кількості дільників усіх підряд чисел проміжку від A до B (обидві межі включно), і виведе:

- суму цих кількостей;
- суму квадратів цих кількостей;
- суму чисел, утворених з окремо взятих цифр цих кількостей.

Вхідні дані. У єдиному рядку через пробіл задані два натуральні числа A та B: ме́жі проміжку. Виконуються обмеження:

- $1 \leqslant A$;
- $B \leqslant 10^{12}$;
- $A \leqslant B \leqslant A + 100$.

Результати. Виведіть у одному рядку через пробіли три числа́: суму кількостей дільників чисел проміжку; суму квадратів кількостей дільників; суму чисел, утворених з окремо взятих цифр цих кількостей.

Вхід	Рез-ти
119 122	27 297 18

Примітка. Число 119 має 4 дільники (1, 7, 17, 119); число 120 має 16 дільників (1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120); число 121 має 3 дільника (1, 11, 121); число 122 має 4 дільники (1, 2, 61, 122). Звідки, $4 + 16 + 3 + 4 = 27$ дає першу відповідь, $4^2 + 16^2 + 3^2 + 4^2 = 16 + 256 + 9 + 16 = 297$ дає другу відповідь, $4 + (1 + 6) + 3 + 4 = 18$ дає третю відповідь.

Розбір задачі.

У цій задачі, виведення в якості результата суми кількостей, суми квадратів кількостей та суми цифр кількостей зроблено з єдиною метою: одночасно і зробити розмір результату досить малим, і ускладнити намагання підібрати відповідь, не розв'язавши задачу по суті.

Помітну частину балів можна набрати, застосувавши до кожного числа́ від A до B спрощену версію розв'язку задачі A, а са́ме: пробуємо ділити на всі чи́сла від 1 до кореня і рахуємо кількість тих, на які поділилося; отриманий результат множимо на 2 (для кожного дільника, меншого кореня, є відповідний йому більший кореня); додатково перевіряємо, чи є корінь цілим числом (якщо є, то він є дільником, для якого нема відповідного йому іншого дільника; отже, треба відняти 1).

Повні бали можна взяти, якщо розкладати кожне число від A до B на прості множники (розд. 1.2.3) та знаходити кількість дільників згідно розд. 1.1. Відмінність часу роботи цього алгоритму від попереднього не зовсім асимптотична: $O((B-A)\times\sqrt{B})$ проти $\Theta((B-A)\times\sqrt{B})$. Але сама́ лише можливість не крутити цикл до початкового $\sqrt{n}$, а враховувати зменшення n, пришвидшує виконання у кілька разів; згаданий наприкінці розд. 1.2.3 прийом «один раз побудувати прості до $\sqrt{B}$» — ще у кілька разів; разом — у десятки разів, а десятки разів — чимало, навіть при однаковій асимптотиці.

Задача G «Дільники на проміжку–2»

Задача відрізняється від попередньої *лише* обмеженнями:

- $1 \leqslant A$;
- $B \leqslant 10^7$;
- $A \leqslant B \leqslant A+10^6$.

Розбір задачі.

Відмінність цієї задачі від попередньої вельми подібна до відмінності між задачами B та C, тож варто пошукати аналогію й у відмінностях у способі розв'язання. Тобто, подумати, як поєднати знаходження кількості дільників із якоюсь модифікацією решета Ератосфена. Таких способів можна запропонувати, щонайменше, два.

Спосіб перший. Поєднаємо ідею решета Ератосфена з формулою з розд. 1.1 та з ідеєю табличної техніки чи то динамічного програмування, чи то рекурентних комбінаторних співвідношень. Замість масиву `er:array[2..maxB] of boolean`, що у решеті Ератосфена містив позначки, чи число просте, будемо підтримувати масив

p	ppp	degP1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
спочатку			1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	1	2	2	4	2	4	2	8	2	4	2	8	2	4	2	16	2	4	2	8	2	4	2	16	2	4	2	8	2
2	4	3	1	2	2	3	2	4	2	6	2	4	2	6	2	4	2	9	2	4	2	6	2	4	2	12	2	4	2	6	2
2	8	4	1	2	2	3	2	4	2	4	2	4	2	6	2	4	2	8	2	4	2	6	2	4	2	8	2	4	2	6	2
2	16	5	1	2	2	3	2	4	2	4	2	4	2	6	2	4	2	5	2	4	2	6	2	4	2	8	2	4	2	6	2
2	32	6	1	2	2	3	2	4	2	4	2	4	2	6	2	4	2	5	2	4	2	6	2	4	2	8	2	4	2	6	2
3	3	2	1	2	2	3	2	4	2	4	4	4	2	6	2	4	4	5	2	8	2	6	4	4	2	8	2	4	8	6	2
3	9	3	1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2	6	2	6	4	4	2	8	2	4	6	6	2
3	27	4	1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2	6	2	6	4	4	2	8	2	4	4	6	2
5	5	2	1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2	6	2	6	4	4	2	8	4	4	4	6	2
5	25	3	1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2	6	2	6	4	4	2	8	3	4	4	6	2
7	7	2	1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2	6	2	6	4	4	2	8	3	4	4	6	2
7	49	3	1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2	6	2	6	4	4	2	8	3	4	4	6	2

`qd:array[1..maxB] of longint`, де `qd` — quantity of divisors, тобто кількість дільників. Ініціалізуємо його елементи двійками, за єдиним виключенням `qd[1]:=1`. Для `qd[1]` значення так назавжди і залишиться 1; для простих чисел `p`, значення `qd[p]` так назавжди і залишиться 2 — і те, і те вже правильно. А для всіх елементів зі скла́деними індексами `j`, значення `qd[j]` буде перепризначене такими циклами:

1. зовнішній цикл перебирає числа проміжку $2 \leqslant p \leqslant \sqrt{B}$, вибираючи лише прості (для яких усе ще `qd[p]=2`);
2. наступний за вкладеністю, перебирає $ppp = p$, $ppp = p^2$, $ppp = p^3$, ... у межах $ppp \leqslant B$; при цьому в ще одній змінній `degP1` підтримується потрібне для формули з розд. 1.1 $m_j + 1$, тобто 2 для $ppp = p$, 3 для $ppp = p^2$, 4 для $ppp = p^3$, тощо;
3. для всіх чисел вигляду $k \times ppp$, тобто для всіх кратних поточному ppp, замінюємо `qd[`$k\times ppp$`]` на `qd[`k`]` $\times$ `degP1`. При цьому k слід перебирати, починаючи з 1 і поки $k \times ppp \leqslant B$.

Приклад застосування цього алгоритму див. нагорі цього розвороту; далі йде вже не опис алгоритму в цілому, а пояснення прикладу.

Після вже поясненої ініціалізації, виконавши перетворення для `ppp`, рівних $2 = 2^1$, $4 = 2^2$, $8 = 2^3$, $16 = 2^4$ та $32 = 2^5$, отримуємо правильні кількості дільників не лише у комірках 1, простих числах та самúх 2, 4, 8, 16, 32, а й узагалі в усіх комірках з номерами вигляду $2^a \times p^1$ (a ціле, від 0 доки поміщається у масив; p просте). Наприклад, у ко-

30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	p	ppp	degP1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	спочатку		
4	2	32	2	4	2	8	2	4	2	16	2	4	2	8	2	4	2	32	2	4	2	8	2	4	2	16	2	2	2
4	2	18	2	4	2	6	2	4	2	12	2	4	2	6	2	4	2	18	2	4	2	6	2	4	2	12	2	4	3
4	2	12	2	4	2	6	2	4	2	8	2	4	2	6	2	4	2	16	2	4	2	6	2	4	2	8	2	8	4
4	2	10	2	4	2	6	2	4	2	8	2	4	2	6	2	4	2	10	2	4	2	6	2	4	2	8	2	16	5
4	2	6	2	4	2	6	2	4	2	8	2	4	2	6	2	4	2	10	2	4	2	6	2	4	2	8	2	32	6
8	2	6	4	4	2	12	2	4	4	8	2	8	2	6	8	4	2	10	2	4	4	6	2	16	2	8	3	3	2
8	2	6	4	4	2	9	2	4	4	8	2	8	2	6	6	4	2	10	2	4	4	6	2	12	2	8	3	9	3
8	2	6	4	4	2	9	2	4	4	8	2	8	2	6	6	4	2	10	2	4	4	6	2	8	2	8	3	27	4
8	2	6	4	4	4	9	2	4	4	8	2	8	2	6	6	4	2	10	2	8	4	6	2	8	4	8	5	5	2
8	2	6	4	4	4	9	2	4	4	8	2	8	2	6	6	4	2	10	2	6	4	6	2	8	4	8	5	25	3
8	2	6	4	4	4	9	2	4	4	8	2	8	2	6	6	4	2	10	4	6	4	6	2	8	4	8	7	7	2
8	2	6	4	4	4	9	2	4	4	8	2	8	2	6	6	4	2	10	3	6	4	6	2	8	4	8	7	49	3

мірці номер $11 = 2^0 \times 11^1$ записано 2, бо так проведена ініціалізація, 11 не кратне 2, і значення не змінювалося; це правильно: у простого числá 11 рівно два дільники. Наприклад, у комірці номер $14 = 2^1 \times 7^1$ тепер записано $4 = 2 \times 2 = (1+1) \times (1+1)$, бо комірка 14 мінялася єдиний раз: при $ppp = 2 = 2^1$, $\texttt{degP1} = 2$; це правильно, відповідає формулі з розд. 1.1. Наприклад, у комірці номер $20 = 2^2 \times 5^1$ тепер записано $6 = 3 \times 2 = (2+1) \times (1+1)$, бо комірка 20 останній з двох разів змінювалася при $ppp = 4 = 2^2$, $\texttt{degP1} = 3$; це правильно, відповідає формулі з розд. 1.1. Наприклад, у комірці номер $48 = 2^4 \times 3^1$ тепер записано $10 = 5 \times 2 = (4+1) \times (1+1)$, бо комірка 48 останній з чотирьох разів змінювалася при $ppp = 16 = 2^4$, $\texttt{degP1} = 5$; це правильно, відповідає формулі з розд. 1.1. Ще раз повторимо: правильно тепер у всіх комірках з номерами вигляду $2^a \times p^1$ ($a \geqslant 0$ ціле, p просте).

У решті комірок значення поки що не обов'язково правильні. Наприклад, у комірці номер $30 = 2^1 \times 3^1 \times 5^1$ записано 4, хоча треба $(1+1) \times (1+1) \times (1+1) = 8$; у комірці номер $35 = 5^1 \times 7^1$ записано 2, хоча треба $(1+1) \times (1+1) = 4$; у комірці номер $36 = 2^2 \times 3^2$ записано 6, хоча треба $(2+1) \times (2+1) = 9$; тощо. Але це означає лише те, що треба продовжувати, розглядаючи подальші прості чúсла та їхні степені аналогічно тому, як розглянули двійку та її степені.

Аналогічно, після перетворень для ppp, рівних $3 = 3^1$, $9 = 3^2$ та $27 = 3^3$, кількість дільників правильна для номерів вигляду $2^a \times 3^b \times p^1$

(a, b невід'ємні цілі, p просте). Наприклад, у раніше розкритикованій комірці номер $30 = 2^1 \times 3^1 \times 5^1$ тепер записане правильне $8 = (1+1) \times (1+1) \times (1+1)$, номер $36 = 2^2 \times 3^2$ — правильне $9 = (2+1) \times (2+1)$.

Чи́сла *не* такого вигляду ще є, але їх вже мало: $25 = 5^2$, $35 = 5^1 \times 7^1$, $49 = 7^2$, $55 = 5^1 \times 11^1$. Очевидно, що після розгляду `ppp`, рівних $5 = 5^1$, $25 = 5^2$, $7 = 7^1$ та $49 = 7^2$, чисел з неправильною кількістю дільників у масиві не залишиться. (Якщо говорити про загальне правило, а не приклад — коли p переберe усі прості $\leqslant \sqrt{B}$.)

Наведені пояснення щодо «$2^a \times p^1$» та «$2^a \times 3^b \times p^1$» при бажанні неважко узагальнити до чіткого інваріанту, який охочі можуть строго довести, а на основі цього довести коректність алгоритму в цілóму.

Незважаючи на *три* вкладені цикли, для цього алгоритму теж має місце асимптотична оцінка часу робóти $\Theta(B \log\log B)$, як і в розд. 1.3.1. (Наприклад, всі проходи при `ppp`, рівних $4 = 2^2$, $8 = 2^3$, $16 = 2^4$, … займають час, асимтотично такий же, як один лише прохід при $\texttt{ppp} = 2$, бо $\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots < \frac{1}{2}$.) Фактичний час виконання, звісно, дещо більший за розд. 1.3.1: треба і множити замість того, щоб просто перекидати елемент на `false`, і організація трьох циклів усе-таки займає час…

Для цього алгоритма значно важливіше, ніж для класичного решета з розд. 1.3.1, розрізняти у зовнішньому циклі прості чи́сла від скла́дених (тут це впливає не лише на швидкодію, а й на правильність). Для цього алгоритма неможлива модифікація, аналогічна розд. 1.3.2, бо тут кількості дільників багатьох із менших чисел необхідні для знаходження кількостей дільників більших (конкретніше, коли `qd[`$k \times ppp$`]` обчислюється як `qd[`k`]` $\times$ `degP1`, потрібно знати `qd[`k`]`).

Спосіб другий. Інша можлива модифікація решета Ератосфена — теж будувати такий самий масив кількостей дільників `qd`, але зовсім іншим чином. (Са́ме так: теж базуючись на решеті Ератосфена, але зовсім іншим чином.) Ініціалізуємо всі елементи значенням 1, бо в кожного числа́ є дільник одиниця. Далі переберемо всі підряд (прості та скла́дені) p з проміжку $2 \leqslant p \leqslant B$ (зверніть увагу, що не $\sqrt{B}$, а B), і для кожного такого p збільшимо на 1 усі елементи `qd[p]`, `qd[2*p]`, `qd[3*p]`, і так далі (поки $k \cdot p \leqslant B$). Таким чином, кожен дільник кожного числа́ буде врахований, бо якщо p є дільником деякого n ($n \vdots p$, при $n \leqslant B$), то це n буде перебране щойно згаданим циклом «p, $2 \cdot p$, …, поки $k \cdot p \leqslant B$». (До речі, це також означає, що са́ме цьому́ розв'язку взагалі не потрібна формула з розд. 1.1. Водночас, са́ме це спричинює необхідність крутити зовнішній цикл до B, а не $\sqrt{B}$.)

На перший погляд може здатися, ніби мало б бути ду-уже довго додавати по одиничці замість отримувати результат комбінаторною формулою з 1.1. Та ще й зовнішній цикл до B, а не $\sqrt{B}$. Але через те, що середня (а отже, й сумарна) кількість дільників насправді досить мала́, це не настільки довго, як може здатися. Крім того, в цьому способі якраз *можлива* оптимізація, аналогічна розд. 1.3.2, тобто підбирання меж внутрішнього циклу так, щоб охоплювати лише проміжок від A до B. І асимптотичний час робо́ти $\Theta((B-A)\times\log B+B)$ виявляється достатнім, щоб конкретно для вказаних обмежень отримати повний бал. Обмежень, де цей дру́гий спосіб був би прямо кращим за попередній перший, небагато, але все ж можна вважати, що цей дру́гий спосіб *теж* розв'язує задачу.

Задача H «Кількість не кратних»

Скільки чисел у проміжку від A до B (обидві межі включно) не діляться націло ні на C, ні на D, ні на E?

Вхідні дані. У єдиному рядку через одинарні пробіли задані п'ять натуральних чисел A, B, C, D, E. Обмеження: $1\leqslant A\leqslant B\leqslant 10^{18}$; чи́сла C, D, E різні і перебувають у проміжку від 2 до 10^6.

Результати. Виведіть єдине невід'ємне ціле число — кількість чисел у проміжку від A до B (межі включно), які не діляться націло ні на C, ні на D, ні на E.

Вхід	Рез-ти
17 42 2 3 5	7

Розбір задачі.

Задача розібрана у розд. 1.6. Додати варто хіба те, що при складності $O(\log C+\log D+\log E)$ обмеження можна було б робити ще більшими, але взяті такі, щоб не треба було морочитися ні з довгою арифметикою, ні з перевірками, чи можна множити (чи нема переповнень).

Задача I «Піраміда»

Піраміда має висоту n Стандартних Будівельних Блоків (СББ), і кожен її рівень — квадрат $k\times k$ блоків, де k — номер рівня, рахуючи згори. Фірма, що виготовляє СББ, продає їх лише партіями по m штук.

Напишіть програму, яка читає в один рядок через пробіл спочатку кількість бажаних рівнів піраміди n ($1\leqslant n\leqslant 10^9$), потім розмір партії СББ m ($1\leqslant m\leqslant 10^6$), і виводить єдине ціле число — кількість Блоків,

що залишаться не використаними після побудови піраміди, якщо купити найменшу можливу кількість цілих партій.

Примітка. Піраміда з 7 рівнів має $1^2+2^2+3^2+4^2+5^2+6^2+7^2=140$ блоків; якщо купити 8 партій по 16 блоків, цих 128 блоків не вистачить; тому слід купити 9 партій по 16 блоків, і з цих 144 блоків 4 залишаться зайвими.

Вхід	Рез-ти
7 16	4

Розбір задачі.

Ця задача проста, щоб узяти *частину* балів, але розв'язати її повністю не так просто. «Очевидний» розв'язок (праворуч) проходить $\approx$половину тестів, отримуючи частину вердиктів «Неправильна відповідь» і «Перевищено час роботи». І тут є де помили́тися й отримати ще менше (важливо правильно врахувати смисл m і взяти тип `int64`).

```
res:=0;
for i:=1 to n do
  res:=res +sqr(int64(i));
res:=res mod m;
if res<>0 then
  res := m - res;
```

При $n\geqslant 3\cdot 10^6$ (приблизно), $1^2+2^2+\ldots+n^2$ виходить за ме́жі 64-бітового типу. Згідно розд. 1.5, з цим можна боротися, додаючи i^2 як `s:=(s+sqr(int64(i)))mod m`. Реалізація з вчасними «`... mod m`», набирає більше балів, але ненабагато. Адже $\approx 10^9$ ітерацій (ще й з повільною операцією `mod`) не поміщаються в ліміт часу.

100%-ий спосіб №1. Якщо знати (або вивести) формулу $1^2+2^2+\ldots+n^2=\frac{n(n+1)(2n+1)}{6}$, можна, поєднавши її з засобами розд. 1.5, отримати зовсім інший розв'язок. Він взагалі не містить циклів (складність $\Theta(1)$), тож працює миттєво. Щодо того, як вивести цю формулу самому — див., наприклад, `dxdy.ru/topic22151.html`. Звісно, доцільність витрачати час туру на такі виведення залежить від умінь конкретного учасника.

```
res:=(n*(n+1)) mod (6*m);
res:=(res*(2*n+1)) mod (6*m);
res:=res div 6;
if res<>0 then
  res := m - res;
```

100%-ий спосіб №2. Навіть не знаючи формули зі способу №1, можна побудувати інший повнобальний розв'язок, користуючись лише базовими знаннями математики, спостережливістю та кмітливістю.

Якщо $n<2\cdot 10^6$, можна порахувати відповідь «у лоб», як на початку розбору. Інакше (враховуючи $m\leqslant 10^6$ та $n\geqslant 2\cdot 10^6$) проміжок від 1 до n містить кілька (як мінімум, два, як максимум — сотні мільйонів) проміжків від 1 до m, від $m+1$ до $2m$, від $2m+1$ до $3m$, тощо.

Розглянемо (праворуч) очевидні тотожності на проміжках від 1 до m та від $m+1$ до $2m$. Помітимо, що кожна з сум m^2+2m, m^2+4m, m^2+6m, ... кратна m і тому не впливає на остато́чну відповідь задачі. Тобто, $(m+1)^2 \bmod m = 1^2 \bmod m$, $(m+2)^2 \bmod m = 2^2 \bmod m$,

$(m+3)^2 \bmod m = 3^2 \bmod m$, **...**, а звідси — сума усього проміжку $(m+1)^2+(m+2)^2+\ldots+(m+m)^2$ має той самий залишок від ділення на m, що й сума усього $1^2+2^2+\ldots+m^2$.

З аналогічних причин, такий са́мий залишок мають і сума усього третього проміжку $(2m+1)^2+(2m+2)^2+\ldots+(2m+m)^2$, і сума будь-якого подальшого. Тому досить цей однаковий для всіх проміжків залишок домножити на $n \operatorname{div} m$, а потім, якщо n не кратне m, окремо порахувати й додати шматочок від $(n \operatorname{div} m)\cdot m+1$ до n (або від 1 до $n \bmod m$).

$1^2=1$	$(m+1)^2=m^2+2m+1$
$2^2=4$	$(m+2)^2=m^2+4m+4$
$3^2=9$	$(m+3)^2=m^2+6m+9$
$\vdots$	$\vdots$
$m^2=m^2$	$(m+m)^2=m^2+2m^2+m^2$

Отже, сумарна кількість ітерацій усіх циклів рівна $m+(n \bmod m) < 2m \leqslant 2\cdot 10^6$. Це довше, ніж $\Theta(1)$ зі «способу №1», але теж вкладається у 1 сек. Щоправда, якби обмеження було не $m \leqslant 10^6$, а $m \leqslant 10^9$, цей розв'язок став би неповним (неефективним), а «спосіб № 1» лишився б повним і ефективним.

Задача J «Обернений факторіал»

Напишіть програму, яка вводить одне натуральне число k $(2 \leqslant k \leqslant 10^9)$ і знаходить, факторіал якого найменшого числа кратний цьому k.

Вхід	Рез-ти
9	6

Примітка. $6! = 720$ ділиться на 9 націло, але жоден з менших факторіалів $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$ чи $5! = 120$ не ділиться на 9 націло.

Розбір задачі.

Зрозуміло, що розв'язок «в лоб» (справді рахувати послідовні факторіали) може отримати лише якісь малі бали. У 64-бітовому типі можна порахувати лише 20!, а відповідь може бути значно більшою за 20. Поєднання «лобового» методу з довгою арифметикою розширює межі, але не дуже: факторіали швидко доростають до тих розмірів, коли починаються проблеми з часом (триваліcтю) їх обробки.

Найбільш природній і очікуваний 100%-ий алгоритм вирішення цієї задачі — розкласти (розд. 1.2.3) k на прості множники, потім за кожним (з присутніх у розкладенні) простим числом окремо підібрати, факторіал якого мінімального числа́ забезпечить таку степінь цього простого, потім узяти з усіх цих результатів максимум.

Отже, треба вміти розв'язувати таку підзадачу: *«Даний степінь вигляду p^m, де p просте, m ціле додатне. Для якого мінімального числá z виконуватиметься умова $z! \vdots p^m$ (зет факторіал кратне p^m)?»*.

Завдяки тому, що p просте, тепер гарантовано, що множники p з'являтимуться у факторіалі лише при досягненні $p!$, $(2\cdot p)!$, $(3\cdot p)!$, ..., і ніколи більше. Але твердження «дільник p^m завжди вперше з'являється у $(m{\cdot}p)!$, бо треба набрати у факторіалі m чисел, кратних p» не завжди правильне: при досить великому m, серед чисел p, $2\cdot p$, $3\cdot p$, ..., $m\cdot p$ можуть бути кратні більшим степеням p. Наприклад, 3, 6, 12, 15, 21, 24 привносять у факторіал по одному простому множнику 3; але 9, 18 — по два; 27 — три множники, тощо (див. також приклад до розв'язку задачі E).

Так що код, який знаходить `z` на основі `m` та `p`, може бути приблизно таким, як праворуч.

```
z:=0;
while m > 0 do begin
  z := z + p;
  zCopy := z;
  while zCopy mod p = 0 do begin
    dec(m);
    zCopy := zCopy div p;
  end;
end; // у змінній z - результат
```

Чи можна це переписати якось ефективніше? При бажанні, можна. Але не варто. Тому що при $k \leqslant 10^9$ показник степені ніяк не може бути більшим $\log_p(10^9) \leqslant \log_2(10^9) < 30$.

Повторимо: спочатку треба розкласти k (зазвичай, найдовшим буде сáме цей етап, і складність алгоритму в цілóму становить $O(\sqrt{k})$), потім виконати наведений код для кожного p^m окремо, потім вибрати $\max z$.

Альтернативний 100%-ий спосіб. Він і менш «очікуваний», і трохи довше працює; але при «$k \leqslant 10^9$, час до 1 сек» він теж придатний, а писати його швидше (сáме реалізовувати, а не придумувати). Перебираючи всі підряд $j = 2, 3, 4, \ldots$, будемо постійно шукати НСД(j, k) і зменшувати k, ділячи на цей НСД (НСД може бути й одиницею, тоді k фактично не змінюється). При таких діях підтримується інваріант «зменшене k містить в собі ті й тільки ті множники початкового k, які все ще відсутні у поточному j!» (самé j!, ясна річ, не обчислюється, але це не заважає виконуватися математичному співвідношенню). Очевидно, що відповіддю буде те j, при якому k вперше зменшиться до 1.

Тобто, реалізувати треба всього-навсього алгоритм Евкліда (сучасну версію) і ще один дуже простий цикл. Не треба ні розкладання на множники, ні ще якихось складних для написання конструкцій.

Тільки це ще не все. Бо якщо k — велике просте число, то при буквальному дотриманні описаного довелося б перебирати j аж до k.

А $\approx 10^9$ ітерацій (ще й з алгоритмом Евкліда у кожній!) — забагато.

Начебто, стандартна проблема, що має стандартне вирішення: цикл не до k, а до $\sqrt{k}$, якщо ця межа перевищена — k просте... Але са́ме в цій задачі це просто неправда! Навіть на прикладі з умови: перебравши $j=2$ та $j=3$, значення $\sqrt{9}=3$ вже перевищили, k при цьому один раз зменшилося з 9 до 3, але все ще не стало рівним 1; правильна відповідь 6 не дорівнює ні поточному $j=3$, ні поточному $k=3$, ні початковому $k=9$. Аналогічна проблема можлива і з іншими вхідними даними: наприклад, для $54=2\times 3^3$ потрібні три штуки простого множника 3 наберуться лише у 9!, хоча $\sqrt{54}\approx 7{,}35$; для $8136578=2\times 2017^2$ (2017 — просте число) потрібні дві штуки 2017, які з'являться лише у факторіалі числа́ $4034=2017\times 2$; і так далі.

З усім цим можна розібратися так, що спосіб не втратить своєї (єдиної) позитивної риси «легко реалізувати». Але доведення правильності прийому, яким будемо рятувати ситуацію, досить складне́. Раз це лише один із способів, дозволимо собі таку некрасивість дати йому лише наближену нестрогу аргументацію.

Усі проблемні приклади містили у факторизації числа́ зі вхідних даних квадрат або куб простого. Виявляється, цим і пояснюється проблема: на проміжку до кореня з початкового k те просте не встигало з'явитися другий (для квадрату) чи третій (для кубу) раз. А якщо збільшити величину проміжку, то встигатиме.

Тож алгоритм може бути таким. Спочатку встановимо верхню межу перебору як `j_max:=round(2*sqrt(k))+100`, тобто так, щоб вона була і значно меншою k (для великих k), і, водночас, точно більшою $\max\{2\cdot\sqrt{k},\ 3\cdot\sqrt[3]{k},\ 4\cdot\sqrt[4]{k},\dots\}$; переберемо $j=2,3,4,\dots$, щоразу ділячи k на НСД(j,k) (не змінюючи `j_max`), доки не настане перша з двох подій: або k зменшиться до 1 (тоді, як і раніше, відповіддю є те j, при якому це сталося), або j перевищить `j_max`, а k усе ще не 1 (тоді відповіддю є поточне значення k). Загальна складність: $O(\sqrt{k}\cdot\log k)$.

Задача K «Кількість дільників на проміжку»

Напишіть програму, яка знайде суму кількостей дільників усіх чисел у проміжку від A до B (обидві межі включно).

Вхідні дані. У єдиному рядку через пробіл задані два натуральні числа A та B ($1\leqslant A\leqslant B\leqslant 10^{12}$), які є межами проміжку.

Результати. Виведіть єдине число — суму кількостей дільників усіх чисел проміжку.

Примітка. Число 119 має 4 дільники (1, 7, 17, 119); число 120 має 16 дільників (1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120); число 121 має 3 дільника (1, 11, 121); число 122 має 4 дільники (1, 2, 61, 122). Звідси відповідь $4+16+3+4=27$.

Розбір задачі.

Вхід	Рез-ти
119 122	27

Враховуючи задачі A, F, G, відомо чимало різних способів отримати *частину* балів. Але жоден з них не може бути повним розв'язком задачі з обмеженнями $1 \leqslant A \leqslant B \leqslant 10^{12}$.

Не варто дотримуватися (нав'язуваної приміткою) тактики «знайдемо кількості дільників кожного числá й пододаємо». Кількості дільників окремих чисел ніхто не питає, тож можна порахувати ті сáмі дільники якось інакше. Наприклад, у іншому порядку. (В задачах F, G окремі кількості теж не питали, але обчислити і суму, і суму квадратів, і суму цифр, не будуючи самí кількості, значно важче, ніж лише суму.)

Як і в задачі H, кількості легше не шукати на проміжку від A до B, а виразити як кількість від 1 до B мінус кількість від 1 до $(A-1)$. Ще з задачі H відомо, що серед усіх чисел від 1 до N є рівно $N \operatorname{div} k$ чисел, кратних k.

На перший погляд, це дає хіба що можливість знайти шукану суму як $(N \operatorname{div} 1)+(N \operatorname{div} 2)+\ldots+(N \operatorname{div} N)$, тобто перебираючи всі дільники аж до N (яке один раз дорівнює $A-1$, інший раз B), і такий алгоритм теж не досить ефективний. Але це можна оптимізувати, якщо придумати, як ще й тут використати перебір до $\sqrt{N}$, а не N.

Приблизно так: перебравши дільники лише до $\sqrt{N}$, подвоїти результат, щоб урахувати дільники, більші $\sqrt{N}$. Це не зовсім правда, бо враховує деякі дільники двічі; але цю похибку можна компенсувати, й отримати правильний алгоритм підзадачі складності $\Theta(\sqrt{N})$ (отже, складність усієї задачі буде $\Theta(\sqrt{A}+\sqrt{B})=\Theta(\sqrt{B})$).

Розглянемо (див. рис.) всі дільники всіх чисел до 32. Кружéчки позначають, що число «№ рядка» є дільником числá «№ стовпчика». Чорний (•) кружечок — дільник менший кореня відповідного числа, білий (○) — більший, напівзаповнений — рівний. Вертикальні лінії виражають попарний зв'язок між p та N/p. Штрихова горизонтальна лінія між 5 та 6 виражає перебір дільників до $\sqrt{N}$ (точніше кажучи, $\lfloor\sqrt{N}\rfloor$, воно ж `trunc(sqrt(N))` чи `floor(sqrt(1.0*N))`). Очевидно, сумарну кількість дільників можна виразити як кількість чорних кружечків, помножену на 2, плюс кількість напівзаповнених.

А для пошуку цих кількостей достатньо перебрати рядки з 1-го по $\lfloor\sqrt{N}\rfloor$-й, відзначаючи, що кількість усіх кружечків у рядку №`i` (при $1 \leqslant$ `i` $\leqslant \lfloor\sqrt{N}\rfloor$) становить `N div i`, `i` серед них є рівно 1 напівзаповнений та рівно (`i`-1) білий (що дає кількість чорних `(N div i) - i`). Завершіть ці міркування та перетворіть їх до вигляду програми самостійно.

Насамкінець, можлива ситуація, коли учасник олімпіади не придумав останню з ідей, але вміє реалізувати як алгоритм «шукати кількість кожного окремо», так і алгоритм «$(N \operatorname{div} 1) + (N \operatorname{div} 2) + \ldots + (N \operatorname{div} N)$»; жоден з цих алгоритмів не проходить усі тести, але тести, які вони не проходять, частково різні. Чи може такий учасник отримати бали як за тести, які проходить один алгоритм, так і за тести, які проходить інший? Запросто, якщо зробить так: прочитавши зі вхідних даних A та B, знайде, яке зі значень $(A + B)$ чи $((B - A) \times \sqrt{B})$ виявляється меншим, і залежно від цього викличе чи то одну, чи то іншу підпрограму, що реалізує відповідний алгоритм.

Конкурс «Бебрас-2019» у світі та Україні

10–12 листопада 2019 року в Україні проведено Міжнародний конкурс з інформатики та комп'ютерного мислення «Бобер-2019» для учнів 2–11-х класів. Цього року у конкурсі взяли участь понад 2 мільйони 970 тисяч учнів з 54-х країн світу.

За кількістю учасників конкурсу Україна посіла сьоме місце у світі:

№ з/п	Країна	К-ть учасників
1	Франція	702 060
2	Німеччина	401 737
3	Великобританія	260 971
4	Індія	178 239
5	Білорусь	176 492
6	Тайвань	143 076
7	**Україна**	**110 978**
8	Туреччина	109 563
9	Чехія	90 976
10	Словаччина	89 768
11	США	56 534
12	Сербія	53 099
13	Австралія	46 738
14	Італія	46 052
15	Південна Корея	44 332
16	Литва	43 490
17	Австрія	37 621
18	Словенія	28 803
19	Угорщина	27 702
20	Нідерланди	25 482
21	Швейцарія	25 345
22	Північна Македонія	25 166
23	Хорватія	24 819
24	Греція	21 717
25	Польща	20 622
26	Латвія	19 550
27	Канада	19 546
28	Румунія	15 695
29	В'єтнам	15 130
30	Китай	13 475
31	Росія	11 937
32	Боснія і Герцеговина	9 765
33	Пакистан	9 240
34	Ірландія	6 882
35	Індонезія	6 773
36	Казахстан	6 698
37	Саудівська Аравія	6 008
38	Фінляндія	5 395
39	Португалія	5 137
40	Бельгія	4 050

41	Таїланд	3 964	48	Єгипет	2 410
42	Швеція	3 609	49	Нова Зеландія	2 312
43	Естонія	3 475	50	Алжир	894
44	Японія	3 420	51	Іспанія	694
45	Узбекистан	3 347	52	Кіпр	521
46	Іран	2 768	53	Болгарія	502
47	Ісландія	2 451	54	Сингапур	288

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

№ з/п	**Країна**	**К-ть учасників**			
1	**Україна**	**32 127**	16	Нідерланди	5 254
2	Туреччина	28 117	17	Швейцарія	3 888
3	Білорусь	23 328	18	Хорватія	3 563
4	Словаччина	23 023	19	Литва	3 528
5	Чехія	21 162	20	Польща	3 443
6	Великобританія	13 980	21	Вєтнам	3 120
7	Австралія	13 267	22	Північна Македонія	3 098
8	Німеччина	11 838	23	Південна Корея	3 068
9	Франція	11 071	24	Пакистан	2 548
10	США	10 278	25	Росія	1 664
11	Італія	9 879	26	Китай	1 480
12	Словенія	6 943	27	Австрія	1 478
13	Індія	6 216	28	Індонезія	1 384
14	Греція	5 510	29	Португалія	1 342
15	Сербія	5 467	30	Угорщина	1 293

31	Ірландія	1 247	37	Швеція	409
32	Саудівська Аравія	1 029	38	Фінляндія	263
33	Узбекистан	678	39	Ісландія	247
34	Казахстан	525	40	Єгипет	234
35	Боснія і Герцеговина	484	41	Алжир	203
36	Нова Зеландія	343	42	Сингапур	59

Ще за одним показником Україна значно випереджає інші країни світу:

Кількість шкіл, які взяли участь у конкурсі:

№ з/п	**Країна**	**К-ть шкіл**			
1	**Україна**	**4081**	16	Іран	536
2	Франція	3 820	17	Нідерланди	510
3	Туреччина	3167	18	Канада	503
4	Китай	2637	19	Литва	482
5	Німеччина	2 308	20	Румунія	438
6	Польща	2163	21	США	413
7	Словаччина	1116	22	Узбекистан	402
8	Великобританія	1014	23	Швейцарія	361
9	Італія	875	24	Латвія	331
10	Південна Корея	808	25	Пакистан	320
11	Чехія	745	26	Північна Македонія	209
12	Тайвань	639	27	Угорщина	202
13	Сербія	600	28	Ірландія	196
14	Греція	596	29	Болгарія	96
15	Індія	580	30	Саудівська Аравія	77

31	Фінляндія	64	36	Естонія	44
32	Бельгія	62	37	Таїланд	40
33	Швеція	52	38	Сінгапур	38
34	Ісландія	48	39	Японія	30
35	Португалія	45	40	Нова Зеландія	19

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:

Володимир Буняк (Рівненський ОІППО),

Галина Гапиченко (Міська станція юних техніків, м. Миколаїв),

Володимир Ксьондзик (СЗОШ №9, м. Львів),

Андрій Мірошниченко (Дніпровська академія неперервної освіти),

Юлія Троян (Кременчуцька ЗОШ №23 Полтавської області),

Марина Чала (Кіровоградський ОІППО)

В Україні конкурс проходив у 4081 координаційних центрах, школах чи об'єднаннях шкіл. У ньому взяло участь 110 978 учнів.

Загальна кількість учасників за віковими групами:

Клас	2	3	4	5	6
Кількість	13332	18795	18503	13528	11358
	7	8	9	10	11
	9135	9820	7775	5110	4426

Кількість учасників по регіонах України:

Область	**Кількість**			
Вінницька	2162		Львівська	11787
Волинська	2684		Миколаївська	2587
Дніпропетровська	10907		Одеська	6445
Донецька	5176		Полтавська	5273
			Рівненська	3293

Житомирська	2561
Закарпатська	2940
Запорізька	9723
Івано-Франківська	2234
Київ	4289
Київська	3028
Кіровоградська	4840
Луганська	1098

Сумська	6082
Тернопільська	2221
Харківська	7964
Херсонська	4804
Хмельницька	2069
Черкаська	2551
Чернівецька	2351
Чернігівська	2813

Честь і слава

Учні, які показали відмінний результат у конкурсі «Бебрас-2019» та відібралися на четвертий етап Всеукраїнської олімпіади з інформатики УОІ-2020:

Прізвище, ім'я	**Навчальний заклад**	**Результат**
11 Клас		
Нємкевич Даша	м. Чернігів, ЗСШФМП № 12	108
Рясний Данило	м. Запоріжжя, Гімназія № 28	105
Семенистий Андрій	Сумська обл., Шосткинський НВК	97
Селеман Денис	м. Херсон, ЗОШ № 30	95
Сачко Дмитро	Донецька обл., Курахівський ліцей «Престиж»	86
10 Клас		
Брозинський Олег	м. Чернівці, ліцей № 1	110
Харьков Денис	м. Суми, ЗОШ № 15	110
Казаков Дмитро	м. Харків, гімназія № 83	106
Манвелян Михайло	м. Київ, ліцей "Лідер"	106
Шевченко Михайло	Кіровоградська обл., Знамянська ЗШ №1	101
Троценко Антон	м. Одеса, Рішельєвський ліцей	100
Стрельбицький Кирило	Донецька обл., Маріупольський технічний ліцей	99

Хоменко Катерина	Херсонська обл., ЗОШ №30	97
Савчук Костянтин	Житомирська обл., Коростишівський НВК	96
Бідзіля Святослав	м. Київ, ліцей «Лідер»	90
Кліщ Дмитро	Львівський фізико-математичний ліцей	88
Ігумнов Олександр	м. Житомир, ліцей № 25	87
Деркач Андрій	Волинська обл., с. Боратин	79
9 Клас		
Ферендович Юрій	Львівський фізико-математичний ліцей	100
Тихонюк Едуард	м. Вінниця, ЗШ № 17	97
Филипюк Андрій	Івано-Франківська обл., Коломийський ліцей ім. Грушевського	97
Лащенков Руслан	Сумська обл., Лебединська ЗОШ № 5	95
Кравченко Олег	м. Київ, ліцей «Лідер»	93
Мокін Григорій	м. Київ, ліцей «Лідер»	88
Левицький Станіслав	Херсонська обл., ЗОШ № 30	87
Самченко Сергій	м. Запоріжжя, Гімназія № 28	86
Тарасюк Назарій	Тернопільська обл., м. Кременець, ліцей ім. Самчука	76
8 Клас		
Просянніков Дмитро	м. Вінниця, ЗШ № 17	92
Столітній Андрій	Дніпропетровська обл., м. Кривий Ріг, КНВК № 129	87

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	Кіровоград 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 8–10 листопада 2020 року (інформаційний лист Інституту Модернізації Змісту Освіти МОН України № 22.1/10-795 від 10.04.2020 р.).

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

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

Для нотаток

Для нотаток

З М І С Т

Навчальне видання

Міжнародний конкурс з інформатики та комп'ютерного мислення «Бебрас-2019»

Матеріали школи програмування (Львів-2019)

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

Формат 60x84/16. Ум. друк. арк. 6,98.
Тираж 6200 пр. Зам. № 986.

Видавництво «Аксіома».
вул. Симона Петлюри, 30б, 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.