Данило Петрович Мисак,

керівник гуртка СШ № 52 м. Києва

Олександр Борисович Рудик,

доцент Київського університету імені Бориса Грінченка

Олімпіада з інформатики у місті Києві у 2015–2016 навчальному році

Передмова

Стаття містить умови завдань II (районного) і завдань III (міського) етапу олімпіади з основ інформатики й обчислювальної техніки у місті Києві у 2015–2016 навчальному році та авторські розв'язання цих завдань. Публікацію адресовано учням класів з поглибленим вивченням математики, учасникам олімпіад з інформатики, студентам математичних спеціальностей, учителям і викладачам вищих навчальних закладів.

Завдання II етапу і задачі 1–3 III етапу упорядкував Данило Мисак, задачі 4–6 III етапу — Олександр Рудик.

Серед робіт учасників є повні розв'язання задач 1, 2, 3, 6. Найкращий результат щодо задачі 4 — 90% балів, щодо задачі 5 — 92 бали.

1. Умови завдань II етапу

Максимальна оцінка за кожну з чотирьох задач — 100 балів. Для всіх задач обмеження на час — 1 секунда / тест; обмеження на пам'ять — 256 МБ.

1. Митько та подібні трикутники (назва програми: similar.*)

Якось на уроці геометрії Митько дізнався, що два трикутники є подібними тоді й лише тоді, коли три сторони одного з них є пропорційними трьом

сторонам іншого. Допоможіть Митькові з домашнім завданням: визначте, чи є два заданих трикутники подібними.

Вхідні дані

У вхідному файлі вказано шість натуральних чисел: перші три — довжини сторін першого трикутника, наступні три — довжини сторін другого трикутника. Усі числа менші за тисячу. Відомо, що з відрізків заданих довжин дійсно можна скласти трикутники.

Вихідні дані

У вихідний файл виведіть число `1`, якщо задані трикутники подібні; в іншому разі виведіть `0`.

Приклади

Вхідний файл similar.in	**Вихідний файл similar.out**
2 3 4 4 6 8	1
3 5 3 50 30 30	1
11 3 9 4 3 5	0

Пояснення до прикладів

У першому прикладі трикутники подібні, бо їхні сторони пропорційні: 2 : 4 = 3 : 6 = 4 : 8.

У другому прикладі сторони також пропорційні, хоч і не в такому порядку, в якому задані у вхідному файлі: 3 : 30 = 5 : 50 = 3 : 30.

У третьому прикладі трикутники не подібні, адже, незалежно від порядку, їхні сторони не є пропорційними.

2. Митько та дивовижний острів (назва програми: island.*)

Якось на уроці географії Митько почув про незвичайний острів, що має форму круга: посередині острова височіє скеля, а населення живе у хижах уздовж периметра острова і через прямовисність скелі може пересуватися від хижі до хижі також виключно по периметрі. Для зручності вважатимемо, що периметр острова розбито на кілька однакових

частин, які умовно назвемо секторами, і з однієї такої частини в сусідню можна перейти рівно за хвилину. У деяких секторах розташовано по хижі (але не більш ніж одна хижа в секторі). Визначте, за який час можна подолати відстань між парою найвіддаленіших хиж на острові.

Вхідні дані

У першому рядку вхідного файлу вказано два натуральних числа n та h — кількість секторів та хиж на острові відповідно. Відомо, що $2 \leqslant h \leqslant n \leqslant 500\,000$. Сектори занумеровано числами від 1 до n у тому порядку, в якому вони йдуть на острові (при цьому сектори з номерами 1 та n замикають коло і також є сусідніми). У другому рядку в порядку зростання вказано номери секторів, у яких є хижі.

Вихідні дані

У вихідний файл виведіть єдине число — відстань між двома найвіддаленішими хижами острова, тобто час у хвилинах, за який можна дійти від однієї з цих хиж до іншої.

Приклади

Вхідний файл island.in	**Вихідний файл island.out**
100 4 3 7 19 20	17
22 4 3 7 19 20	10

Пояснення до прикладів

У першому прикладі найвіддаленішими є перша та остання хижі, тож відповідь дорівнює 20 – 3 = 17.

У другому прикладі перша та остання хижі вже не є найвіддаленішими, адже між ними можна пройти за 5 хвилин (таким чином: сектор 3 — сектор 2 — сектор 1 — сектор 22 — сектор 21 — сектор 20). Найвіддаленішими натомість є хижі в секторах 7 і 19: вибравши

оптимальний напрямок руху, дійти від однієї з них до іншої можна лише за 10 хвилин.

3. Митько та арифметичні прогресії (назва програми: progress.*)

Якось на уроці алгебри Митько довідався, що арифметичною прогресією називають послідовність чисел, у якій різниця між кожними двома сусідніми членами однакова. Щоб учні краще засвоїли матеріал, учитель взяв деякі дві арифметичні прогресії, кожна з яких складається з n натуральних чисел, перемішав між собою всі $2n$ чисел (вони виявилися попарно різними) і виписав утворену послідовність на дошці. Допоможіть Митьку виконати вчителеве завдання: відновити з заданого набору чисел дві початкові арифметичні прогресії. Вхідні дані гарантують, що зробити це є рівно один спосіб.

Вхідні дані

У першому рядку вхідного файлу вказано натуральне число n — кількість членів кожної з двох арифметичних прогресій, $3 \leqslant n \leqslant 100\,000$. У другому рядку записано $2n$ різних натуральних чисел, менших за 10^9, — перемішані елементи обох прогресій.

Вихідні дані

У перший рядок вихідного файлу виведіть усі члени першої арифметичної прогресії в порядку зростання, а в другий рядок — усі члени другої арифметичної прогресії в порядку зростання. Прогресії виведіть у такому порядку, щоб перше число в першому рядку було меншим за перше число у другому рядку.

Приклад

Вхідний файл progress.in	**Вихідний файл progress.out**
4 7 9 23 3 16 15 11 2	2 9 16 23 3 7 11 15

Пояснення до прикладу

Виведені у вихідний файл послідовності є арифметичними прогресіями, бо 9 – 2 = 16 – 9 = 23 – 16 і 7 – 3 = 11 – 7 = 15 – 11.

4. Митько та міжпланетна подорож (назва програми: journey.*)

Якось після важкого дня у школі з уроками астрономії, фізики та економіки у голові Митька все перемішалося, і хлопцю наснився дивний сон. У віддаленому майбутньому люди заселяють n планет, між якими пересуваються за допомогою телепортації. Для зручності планети занумеровано числами від 1 до n. Процес телепортації обслуговують m різних компаній, і вони конкурують між собою. Тому телепортуватися можна не між будь-якою парою планет, а лише між тими, які обслуговує одна й та сама компанія. На щастя, одну й ту саму планету може обслуговувати відразу кілька різних компаній. До того ж відомо, що з кожної планети можна перемістися на будь-яку іншу якщо й не за одну, то принаймні за декілька послідовних телепортацій. З'ясуйте, за яку найменшу кількість послідовних телепортацій можна перемістися з планети 1 на планету n.

Вхідні дані

У першому рядку вхідного файлу записано два натуральних числа n та m — кількість планет та компаній відповідно; $n \geqslant 3$, $m \geqslant 2$, а добуток цих двох чисел не перевищує мільйона. Кожен з наступних n рядків містить по m цифр, *не розділених пробілом*, та задає інформацію про відповідну планету (у першому з цих рядків — інформація про планету 1, в останньому — про планету n): якщо цифра на позиції k в рядку є одиницею, то компанія під номером k обслуговує дану планету; якщо ж ця цифра нуль, то не обслуговує. Кожна компанія обслуговує хоча б дві планети.

Вихідні дані

У вихідний файл виведіть єдине число — найменшу кількість послідовних телепортацій, необхідних, щоб з планети 1 дістатися на планету n.

Приклад

Вхідний файл journey.in	Вихідний файл
4 2 01 01 11 10	2

Пояснення до прикладу

Першу планету обслуговує тільки друга компанія, тому з неї можна потрапити на другу і третю планети, але не на четверту. Зате за дві телепортації — з транзитом через третю планету — з першої на четверту потрапити вже можна.

2. Ідеї розв'язання завдань II етапу

1. Митько та подібні трикутники

Щоб зрозуміти, чи є трикутники подібними, можна перепробувати всі можливі варіанти порядку, в якому сторони можуть виявитися пропорційними (якщо зафіксувати порядок сторін одного з трикутників, для іншого є 6 варіантів розстановки). А можна зробити інакше: просто відсортувати сторони кожного з трикутників. Тоді найменшій стороні першого трикутника відповідатиме найменша сторона другого трикутника; середній стороні відповідатиме середня; найбільшій — найбільша.

Перевірку пропорційності ні в якому разі не можна здійснювати за допомогою оператора цілочисельного ділення (як-от `div` у Pascal'і), адже цей оператор бере лише цілу частину від ділення одного числа на інше та ігнорує остачу. Не варто користуватися і звичайним діленням у дійсних числах: при обчисленні та поданні таких чисел комп'ютер може втратити точність, а відмінність хоч би й в одну мільйонну означатиме, що два

числа, які насправді є рівними , комп'ютер рівними не ви́знає . Натомість можна скористатися рівноцінним порівнянням добутків, адже $a_1 : b_1 = a_2 : b_2$ — це те саме, що $a_1 \times b_2 = b_1 \times a_2$. Щоправда, такі добутки можуть вийти за межі двобайтової змінної (як `integer` у Pascal'і). Це треба врахувати й скористатися відповідним типом даних.

Отже, один з можливих способів розв'язати задачу на повний бал такий: вивести одиницю, якщо пара з найменшої та середньої сторони першого трикутника пропорційна парі з найменшої та середньої сторони другого трикутника, а пара з середньої та найбільшої сторони першого трикутника пропорційна парі з середньої та найбільшої сторони другого трикутника; інакше вивести нуль.

2. Митько та дивовижний острів

Відстань між хижами в секторах k та l (де $k \leqslant l$) обчислюється за формулою $\min\{l - k, n + k - l\}$, тобто дорівнює меншій з двох відстаней: у випадку, якщо йти в порядку збільшення номера сектора, та у випадку, якщо йти в порядку зменшення номера сектора. Ідейно найпростіший спосіб розв'язати задачу — перебрати всі пари хиж, порахувати відстань між кожною та вибрати з усіх підрахованих величин найбільшу. Час виконання програми в такому випадку буде квадратичним від кількості хиж, що дозволить набрати лише половину балів за задачу. А от більш оптимальний алгоритм, що заробить повний бал, можна побудувати, спираючись на спостереження, яке наводимо нижче.

Нехай на колі зафіксовано деякий набір точок (хиж). Найвіддаленішою від точки A буде та з точок набору, що лежить найближче до точки кола, діаметрально протилежної до A. Якщо ми пересуватимемо точку A, скажімо, за годинниковою стрілкою, то й діаметрально протилежна до неї точка рухатиметься за годинниковою стрілкою, а значить, у порядку руху за годинниковою стрілкою змінюватиметься і найвіддаленіша від A хижа.

Отже, алгоритм буде таким: зчитуючи розташування кожної

наступної хижі, визначаємо найвіддаленішу від неї (вже зчитану раніше) хижу. Для цього беремо хижу, яка виявилася найвіддаленішою від попередньої зчитаної хижі, та рухаємось у порядку збільшення номера хижі, допоки відстань до поточної хижі збільшується. Зокрема, якщо відстань не збільшилася вже при першому порівнянні, то найвіддаленішою від даної хижі є та сама хижа, яка була найвіддаленішою від попередньої; якщо, навпаки, відстань збільшувалася увесь час, аж поки ми не натрапили на ту саму хижу, яку зчитували (а від неї до неї самої відстань, очевидно, нульова), то найвіддаленішою від даної є попередня зчитана хижа. Таким чином, здійснивши в процесі зчитування даних загалом не більше ніж один повний прохід по колу у пошуках найвіддаленіших хиж, ми визначимо відповідь — це максимальна зі знайдених для кожної хижі найбільших відстаней.

Є й інші алгоритми, що працюють, як і даний, лінійний від кількості хиж час, проте наведений алгоритм є одним із найпростіших у реалізації.

Насамкінець додамо, що алгоритм, який спирається на ідею двійкового пошуку найвіддаленішої хижі, хоч і має час виконання $O(h \log h)$, що гірше за лінійний, тим не менше набирає повний бал. Стільки ж потенційно дозволяють заробити і решта алгоритмів із часом виконання $O(h \log h)$.

3. Митько та арифметичні прогресії

Один з можливих способів розв’язати задачу такий. Відсортуємо всі $2n$ чисел у порядку від найменшого до найбільшого. Серед трьох найменших елементів відсортованої послідовності принаймні два належатимуть до однієї й тієї ж прогресії, причому будуть двома найменшими її членами (а якщо всі три найменші числа належать одній і тій самій прогресії, то найменшими двома її членами є, очевидно, два перших числа). Тепер, послідовно розглядаючи гіпотези про те, що двома найменшими членами однієї з прогресій є перший і другий; перший і третій; другий і третій елементи відсортованої послідовності, встановимо,

котра з цих гіпотез є правильною. Для перевірки можемо скористатися таким підходом: перебиратимемо в порядку збільшення всі $2n$ чисел; якщо чергове число, яке ми розглядаємо, є таким, що підходить до першої прогресії (а, беручи припущення гіпотези, ми вже знаємо і перший член цієї прогресії, і її різницю, і кількість елементів), то долучаємо це число до першої прогресії, інакше — до другої. Якщо обидві побудовані послідовності дійсно є арифметичними прогресіями (з n елементами в кожній), то маємо відповідь; інакше переходимо до наступної гіпотези. Описаний процес перевірки можна втілити за один лінійний прохід послідовності.

Оскільки алгоритм складається з сортування і кількох лінійних проходів масиву на $2n$ елементів, його складність можна оцінити як $O(2n \log 2n) = O(n \log n)$. Це дозволяє заробити повний бал.

Утім, алгоритм можна оптимізувати і до лінійного. Для пошуку трьох найменших елементів замість сортування використаємо, наприклад, три послідовних лінійних проходи. А для перевірки гіпотези перебиратимемо числа не в порядку збільшення, а в довільному. Це не завадить відібрати з набору ті й лише ті числа, що належать до першої прогресії. Щоб установити, чи решта чисел утворюють другу прогресію, знайдемо шляхом двох лінійних проходжень два найменших числа, що не потрапили до першої прогресії: вони якраз і мають становити два перших члени другої прогресії. Залишається ще раз пройтися по масиву і перевірити, чи решта чисел у ньому «узгоджуються» зі знайденими першими членами потенційної прогресії. При цьому, звичайно, не слід забувати, що в обох прогресіях повинно бути рівно по n членів. Нарешті, щоб уникнути сортування при виведенні відповіді, можна конструювати прогресії безпосередньо з їхніх арифметичних властивостей (а не виводити у вихідний файл упорядковані елементи масиву).

4. Митько та міжпланетна подорож

Це — одна з типових задач, які можна розв’язати за допомогою

пошуку в ширину: між двома планетами існує перехід тоді й лише тоді, коли їх обслуговує хоча б одна спільна компанія. Однак у даному випадку планет може бути кілька сотень тисяч, і кожну або майже кожну їх пару може бути сполучено. Тому при реалізації звичайного пошуку в ширину доведеться зіткнутися якщо й не з проблемою виділення пам'яті, то принаймні з тим, що програма не вкладатиметься в обмеження на час на великих вхідних даних.

Ключовим спостереженням, що дозволить вирішити дану проблему, буде таке: на довільному кроці виконання алгоритму немає сенсу розглядати сполучення по тих компаніях (по тих стовпцях вхідних даних), сполучення по яких ми вже один раз розглядали раніше. При цьому, розглядаючи планету з черги пошуку, суміжні до неї планети знаходимо за допомогою дворівневого циклу, де в зовнішньому циклі перебираємо компанії, що обслуговують дану планету, а для кожної компанії, яку ще є сенс розглядати, у вкладеному циклі перебираємо інші планети, які ця компанія обслуговує. Складність виконання алгоритму — $O(nm)$.

Для кращого розуміння радимо переглянути прокоментований код авторського розв'язання задачі.

Окремо пояснимо, як можна ефективно зберігати вхідні дані, враховуючи, що конкретні обмеження на кількість планет і компаній нам не відомі, а знаємо лише добуток цих кількостей. Нам достатньо завести один одновимірний масив, розмірність якого відповідає максимальному значенню добутку кількостей планет і компаній. Тоді значення, що стоїть на перетині i-го рядка та j-го стовпця у вхідному файлі, якщо домовитись, що нумерацію рядків і стовпців ведемо з нуля, а не з одиниці, слід зберігати в комірці з індексом $i \times m + j$ (нумерацію комірок теж ведемо з нуля — стандартно для C++). Це дозволить заповнити вхідними даними без жодного перекриття всі комірки масиву з нульової до $(nm - 1)$-ї включно.

3. Авторські розв'язання завдань II етапу

1.Митько та подібні трикутники (similar.cpp)

```
/* GCC */

#include <stdio.h>
#include <algorithm>

using namespace std;

// Функція приймає на вхід дві пари сторін.
// Визначає, чи є вони пропорційними (в
// заданому порядку).
bool proportional(int a1, int a2,
                  int b1, int b2)
{
    // Оскільки для змінних типу int ділен-
    // ня є цілочисельним, а виходити у
    // дробові числа не хочеться (втрачає-
    // ться точність), зводимо порівняння
    // до добутків:
    return a1 * b2 == b1 * a2;
}

int main()
{
    freopen("similar.in", "r", stdin);
    freopen("similar.out", "w", stdout);

    int s[2][3]; // Сторони першого і дру-
                 // гого трикутників

    for (int t = 0; t < 2; t++) // t — но-
                          // мер трикутника
    {
        for (int i = 0; i < 3; i++) // i —
                           // номер сторони
            scanf("%d", &s[t][i]);

        // Упорядковуємо сторони трикутни-
        // ка, який зараз розглядаємо, за
        // неспаданням:
        sort(s[t], s[t] + 3);
    }

    // Перевіряємо на пропорційність дві
    // пари відповідних сторін трикутників:
    bool answer = proportional(
                        s[0][0], s[0][1],
                        s[1][0], s[1][1])
              && proportional(
                        s[0][1], s[0][2],
```

```
                            s[1][1], s[1][2]);

      printf("%d\n", answer ? 1 : 0);
}
```

2. Митько та дивовижний острів (island.cpp)

```
/* GCC */

#define maxN 500000

#include <stdio.h>
#include <algorithm>

using namespace std;

int a[maxN]; // Розташування хиж

// Функція приймає на вхід розташування
// двох хиж (from <= to) і загальну кіль-
// кість секторів.
// Повертає коротшу з двох відстаней між
// даними хижами.
int distance(int from, int to, int n)
{
    return min(to - from, n + from - to);
}

int main()
{
    freopen("island.in", "r", stdin);
    freopen("island.out", "w", stdout);

    int n, h;
    scanf("%d %d\n%d", &n, &h, &a[0]);

    int c = 0; // Індекс хижі, найвіддале-
     // нішої від поточної/попередньої зчи-
     // таної хижі
    int maxDistance = 0; // Найбільша від-
     // стань між хижами на даний момент
    for (int i = 1; i < h; i++) // Зчитуємо
     // всі хижі, крім першої (її вже зчи-
     // тано раніше)
    {
        scanf("%d", &a[i]);
        int curDistance = distance(
                            a[c], a[i], n);
        int nextDistance = distance(
                        a[c + 1], a[i], n);
        while (nextDistance > curDistance)
         // Поки можна йти вперед, збільшу-
         // ючи відстань до поточної зчита-
```

```
         // ної хижі, робимо це
        {
            c++;
            curDistance = nextDistance;
            nextDistance = distance(
                        a[c + 1], a[i], n);
        }
        // Зараз у curDistance записано
        // відстань від поточної зчитаної
        // хижі до хижі, найвіддаленішої
        // від неї. Відповідним чином онов-
        // люємо загальний максимум:
        maxDistance = max(maxDistance,
                          curDistance);
    }

    printf("%d\n", maxDistance);

    return 0;
}
```

3. Митько та арифметичні прогресії (progress.cpp)

```
/* GCC */

#define maxN 100000

#include <stdio.h>
#include <algorithm>
#include <vector>

using namespace std;

int all[maxN * 2], n; // Усі числа із вхід-
 // ного файлу та їхня кількість
vector<int> p[2]; // Дві прогресії, які бу-
 // дуватиме програма

// Функція test пробує розбити послідов-
// ність на дві прогресії за припущення, що
// в однієї з прогресій (яку умовно назвемо
// першою) перший елемент дорівнює first, а
// другий — second. Якщо розбити послідов-
// ність на прогресії вдалося, повертає
// true, інакше — false. У разі успіху за-
// повнює вектори p[0] та p[1] (глобальні
// змінні) членами відповідних прогресій.
bool test(int first, int second)
{
    p[0].clear(); // Очищуємо вектори перед
                  // новим використанням
    p[1].clear();
```

```
    int a[2], d[2]; // Перші члени та різ-
                // ниці прогресій відповідно
    a[0] = first;
    d[0] = second - first;
    a[1] = d[1] = 0; // Нулі означають, що
     // поки що інформації про другу
     // прогресію нема
    for (int i = 0; i < 2 * n; i++) // На-
     // магаємось припасувати кожне число
     // до однієї з прогресій
        if (p[0].size() < n && all[i] ==
                 a[0] + d[0] * p[0].size())
            // Якщо число підходить до пер-
            // шої прогресії, його необхід-
            // но додати саме в неї
            p[0].push_back(all[i]);
        else // Інакше число має належати
             // другій прогресії:
        {
            if (a[1] == 0) // Якщо це пер-
            // ший елемент другої прогре-
            // сії, на який ми натрапили,
            // запам'ятовуємо його
                a[1] = all[i];
            else if (d[1] == 0) // Якщо це
            // другий елемент прогресії, на
            // який ми натрапили, то обра-
            // ховуємо і запам'ятовуємо
            // різницю прогресії
                d[1] = all[i] - a[1];
            else if (p[1].size() >= n ||
       all[i] != a[1] + d[1] * p[1].size())
              // Якщо ж інформація про прогресію
              // вже відома, але число не підхо-
              // дить, виходимо
                  return false;
            p[1].push_back(all[i]); // Якщо
            // ми не вийшли, то елемент
            // слід додати до прогресії
        }
    return true; // Якщо ми досі не вийшли
    // з функції, то побудувати прогресії
    // вдалося
}

int main()
{
    freopen("progress.in", "r", stdin);
    freopen("progress.out", "w", stdout);

    // Зчитуємо дані:
    scanf("%d", &n);
    for (int i = 0; i < 2 * n; i++)
```

```
        scanf("%d", &all[i]);

    sort(all, all + 2 * n); // Сортуємо
                    // числа обох прогресій

    // Послідовно перевіряємо гіпотези про
    // те, що найменшими елементами однієї
    // з прогресій є деякі два з найменших
    // трьох чисел відсортованої послідов-
    // ності (оператори || потрібні, щоб
    // після виявлення істинної гіпотези не
    // продовжувати розгляд інших):
    test(all[0], all[1]) ||
    test(all[0], all[2]) ||
    test(all[1], all[2]);

    if (p[0][0] > p[1][0]) // Забезпечуємо
           // правильний порядок виведення
        swap(p[0], p[1]);

    // Виводимо прогресії:
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < n; j++)
            printf(
      j < n - 1 ? "%d " : "%d\n", p[i][j]);

    return 0;
}
```

Лінійний алгоритм

```
/* GCC */

#include <stdio.h>
#include <algorithm>
#include <vector>

using namespace std;

int n;
vector<int> all; // Усі числа із вхідного
                 // файлу
pair<int, int> p[2]; // p[i] — пара з пер-
 // шого та другого члена i-ї прогресії

// Переставляє howMany найменших елементів
// із проміжку чисел з індексами від from
// до to на перші місця цього проміжку
// (тобто на місці з індексом from стоятиме
// найменший елемент, на місці from+1 —
// другий за величиною і т. д.).
```

```
void swapSmallest(
               vector<int>::iterator from,
               vector<int>::iterator to,
               int howMany)
{
    for (int i = 0; i < howMany; i++)
        iter_swap(from + i,
               min_element(from + i, to));
}

// Визначає, чи належить число number
// n-елементній арифметичній прогресії з
// першим членом first та другим членом
// second.
inline bool belongsToProgression(
  int number, int first, int second, int n)
{
    int d = second - first,
        cur = number - first;
    return cur >= 0 && cur % d == 0
                    && cur / d < n;
}

// Функція test пробує розбити послідов-
// ність на дві прогресії за припущення, що
// в однієї з прогресій (яку умовно назвемо
// першою) перший елемент дорівнює first, а
// другий — second. Якщо розбити послідов-
// ність на прогресії вдалося, повертає
// true, інакше — false. У разі успіху за-
// повнює глобальний масив p парами з пер-
// ших і других членів прогресій.
bool test(int first, int second)
{
    vector<int> leftovers; // Числа, що за-
     // лишаться після вибирання членів
     // першої прогресії
    for (int i = 0; i < all.size(); i++)
        if (! belongsToProgression(all[i],
                        first, second, n))
        { // Якщо число не належить до пер-
          // шої прогресії, додаємо його до
          // потенційної другої:
            leftovers.push_back(all[i]);
            if (leftovers.size() > n)
                // Гіпотеза не справдилася:
                // у другій прогресії може
                // бути лише n елементів
                return false;
        }
    // Якщо ми дійшли до цього місця, то з
    // першою прогресією все добре, а в ма-
    // сиві leftovers рівно n елементів.
```

```
    // Залишилося з'ясувати, чи утворюють
    // вони прогресію.
    swapSmallest(leftovers.begin(),
               leftovers.end(), 2); // Зна-
           // ходимо два найменших елементи
    for (int i = 2; i < n; i++)
        if (! belongsToProgression(
                 leftovers[i], leftovers[0],
                 leftovers[1], n))
             return false; // Якщо хоча б
     // один з решти елементів не «узгоджу-
     // ється» з двома першими, то гіпотеза
     // не справдилася
    // Якщо ми дійшли аж сюди, то гіпотеза
    // справдилася. Заповнюємо інформацію
    // про прогресії:
    p[0] = make_pair(first, second);
    p[1] = make_pair(leftovers[0],
                     leftovers[1]);
    return true;
}

int main()
{
    freopen("progress.in", "r", stdin);
    freopen("progress.out", "w", stdout);

    // Зчитуємо дані:
    scanf("%d", &n);
    for (int i = 0; i < 2 * n; i++)
    {
        int t;
        scanf("%d", &t);
        all.push_back(t);
    }

    // Переставляємо три найменших числа на
    // перші місця:
    swapSmallest(all.begin(),
                 all.end(), 3);

    // Послідовно перевіряємо гіпотези про
    // те, що найменшими елементами однієї
    // з прогресій є деякі два з найменших
    // трьох чисел відсортованої послідов-
    // ності (оператори || потрібні, щоб
    // після виявлення істинної гіпотези не
    // продовжувати розгляд інших):
    test(all[0], all[1]) ||
    test(all[0], all[2]) ||
    test(all[1], all[2]);

    if (p[0].first > p[1].first) // Забез-
```

```
            // печуємо правильний порядок
            // виведення
        swap(p[0], p[1]);

    // Виводимо прогресії:
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < n; j++)
            printf(j < n - 1 ? "%d " :
                               "%d\n",
p[i].first + j * (p[i].second - p[i].first)
                  );

    return 0;
}
```

4. Митько та міжпланетна подорож (journey.cpp)

```
/* GCC */

#define maxProduct 1000000
// Запроваджуємо ще дві константи, які до-
// рівнюють тому ж числу, для кращого розу-
// міння коду:
#define maxPlanets maxProduct // Найбільша
 // можлива кількість планет
#define maxCompanies maxProduct // Найбіль-
 // ша можлива кількість компаній

#include <stdio.h>
#include <queue>

using namespace std;

int n, m;
// У масиві data зберігатимуться дані із
// вхідного файлу: які компанії обслугову-
// ють які планети. Оскільки у нас є обме-
// ження тільки на добуток кількостей пла-
// нет і компаній, а не на самі ці кілько-
// сті, ми змушені зробити масив одновимір-
// ним і «симулювати» двовимірність таким
// чином: якщо i — номер планети, а j — но-
// мер компанії (нумерація з нуля), комір-
// кою data[i][j] вважатимемо комірку
// data[i * m + j] (де m — кількість компа-
// ній). Це дозволить щільно заповнити од-
// новимірний масив, але водночас жодних
// «накладок» в індексах комірок не виник-
// не. Масив великий, тому оголошуємо його
// глобальною змінною: локальні змінні, на
// відміну від глобальних, розміщуються у
// стеку, а ми хочемо уникнути його пере-
// повнення.
```

```
bool data[maxProduct];
// planetVisited[k] міститиме найменшу кі-
// лькість переміщень, за яку з планети 0
// можна дістатися до планети k; або -1,
// якщо ця кількість ще не відома (тобто до
// планети k ми ще не дійшли).
int planetVisited[maxPlanets];
// companyVisited[k] міститиме інформацію
// про те, чи дійшли ми вже хоча б до
// однієї планети, яку обслуговує компанія
// під номером k.
bool companyVisited[maxCompanies];

// У функціях, що йдуть далі, ключове слово
// inline використано для пришвидшення їх
// виклику (див. документацію до C++ із
// цього приводу).

// За номерами планети та компанії (в обох
// нумерація з нуля) функція відновлює ін-
// декс відповідної комірки в одновимірному
// масиві (див. коментар про «симуляцію»
// двовимірності вище):
inline int getIndex(int planet,
                    int company)
{
    return planet * m + company;
}

// Додає інформацію про те, чи обслуговує
// планету з номером planet компанія з но-
// мером company:
inline void setInfo(int planet,
                    int company,
                    bool value)
{
    data[getIndex(planet, company)] =
                                      value;
}

// Повертає інформацію про те, чи обслуго-
// вує планету planet компанія company:
inline bool getInfo(int planet,
                    int company)
{
    return data[getIndex(planet, company)];
}

int main()
{
    freopen("journey.in", "r", stdin);
    freopen("journey.out", "w", stdout);
```

```
scanf("%d %d\n", &n, &m);
// Посимвольно зчитуємо табличку із
// вхідного файлу:
for (int planet = 0; planet < n;
                              planet++)
{
    for (int company = 0; company < m;
                             company++)
    {
        char c;
        scanf("%c", &c);
        setInfo(planet, company,
                             c == '1');
    }
    scanf("\n");
}

// Ініціалізуємо масиви — спочатку жод-
// ної планети та компанії ще не від-
// відано:
for (int planet = 0; planet < n;
                              planet++)
    planetVisited[planet] = -1;
for (int company = 0; company < m;
                             company++)
    companyVisited[company] = false;

// Створюємо чергу модифікованого пошу-
// ку в ширину, додаємо туди початковий
// елемент (першу планету — тобто пла-
// нету з номером 0, оскільки нумерація
// у програмі — з нуля), а також декла-
// руємо, що з планети 0 у неї ж саму
// можна потрапити за 0 телепортацій:
queue<int> q;
q.push(0);
planetVisited[0] = 0;

while (planetVisited[n - 1] == -1)
{ // Поки найкоротший шлях до останньої
  // планети не знайдено, продовжуємо
  // пошук
    int currentPlanet = q.front();
    // Запам'ятовуємо перший елемент
    // з черги пошуку (його ми зараз
    // будемо розглядати) та видаляємо
    // його
    q.pop();
    for (int company = 0; company < m;
                             company++)
        if (! companyVisited[company]
            && getInfo(currentPlanet,
                       company))
```

```
            { // Для кожної компанії, якої
              // не було раніше і яка об-
              // слуговує дану планету:
                companyVisited[company] =
                                      true;
                // Запам'ятовуємо, що ком-
                // панія вже трапилась
                for (int planet = 0;
                         planet < n;
                         planet++)
                    if (planetVisited[
                              planet] == -1
                 && getInfo(planet, company))
                    { // Для кожної плане-
                      // ти, яку не прохо-
                      // дили раніше і яку
                      // обслуговує компа-
                      // нія:
                        q.push(planet);
                        // Додаємо планету
                        // до черги пошуку
                        // Найкоротший шлях
                        // у дану планету —
                        // дійти до поточ-
                        // ної планети з
                        // черги, яку ми
                        // розглядаємо, та
                        // ще за один крок
                        // перейти з неї
                        // в дану:
                        planetVisited[
                                  planet] =
          planetVisited[currentPlanet] + 1;
                    }
            }
    }

    // Відповідь уже готова і зберігається
    // у відповідний комірці масиву:
    printf("%d\n", planetVisited[n - 1]);

    return 0;
}
```

4. Умови завдань ІІІ етапу

1. Числа

Максимальна оцінка: 200 балів

Обмеження на час: 0,15 сек.

Обмеження на пам’ять: 256 МБ

Вхідний файл: numbers.in

Вихідний файл: numbers.out

Програма: numbers.*

Три натуральних числа назвемо *дружніми*, якщо вони попарно різні і добуток будь-яких двох із них націло ділиться на третє.

Завдання

За двома заданими натуральними числами встановіть кількість чисел, що утворюють з ними дружню трійку.

Вхідні дані

У єдиному рядку вхідного файлу вказано два різних натуральних числа, жодне з яких не перевищує 40 000.

Вихідні дані

Вихідний файл повинен містити єдине число: кількість натуральних чисел таких, що в сукупності з двома заданими вони утворюють трійку дружніх чисел.

Приклади

№	numbers.in	numbers.out
1	5 15	2
2	18 12	7

Пояснення до першого прикладу

Є рівно два числа, що в сукупності з числами 5 і 15 утворюють дружню трійку: це число 3 (бо 3 × 5 ділиться на 15, 3 × 15 ділиться на 5, а 5 × 15 ділиться на 3), а також число 75 (бо 5 × 15 ділиться на 75, 5 × 75 ділиться на 15, а 15 × 75 ділиться на 5).

2. Трикутник

Максимальна оцінка: 200 балів

Обмеження на час: 2,5 сек.

Обмеження на пам'ять: 256 МБ

Вхідний файл: triangle.in

Вихідний файл: triangle.out

Програма: triangle.*

Задано деякий набір відрізків цілих довжин.

Завдання

З відрізків заданого набору побудуйте трикутник із найбільшим або найменшим можливим периметром.

Вхідні дані

У першому рядку вхідного файлу вказано число n — кількість відрізків; ця кількість не менша за 4 і не перевищує 10^6. У другому рядку міститься n натуральних чисел, менших за 10^9, — довжини відрізків. У третьому рядку записано три латинських літери, що утворюють або слово `max`, або слово `min`: вони відповідають задачам пошуку найбільшого та найменшого периметра трикутника відповідно.

Вихідні дані

Вихідний файл повинен містити єдине число: залежно від поставленої у вхідних даних задачі або найбільший, або найменший можливий периметр трикутника, побудованого на деяких трьох відрізках із заданого набору. Якщо з жодних трьох відрізків побудувати трикутник неможливо, виведіть слово `none`.

Приклади

№	triangle.in	triangle.out
1	5 2 3 9 4 4 Max	11

№	triangle.in	triangle.out
2	5 2 3 9 4 4 min	9
3	4 1 2 3 6 max	none

3. Граф

Максимальна оцінка: 200 балів

Обмеження на час: 1,5 сек.

Обмеження на пам’ять: 256 МБ

Вхідний файл: graph.in

Вихідний файл: graph.out

Програма: graph.*

У цій задачі ми говоримо про неорієнтовані графи.

Неорієнтований граф — це множина, що містить елементи двох типів:

- *вершини* графа — елементи довільної природи;
- *ребра* графа — неупорядковані пари вершин графа.

Традиційно вершини графа зображають точками на площині, а ребра — відрізками або кривими, що сполучають зображення відповідних вершин графа.

Інформацію про графи можна зберігати в пам’яті комп’ютера багатьма різними способами. Наприклад — простим переліком ребер, тобто пар номерів сполучених між собою вершин. Або з допомогою т. зв. списків суміжності: для кожної вершини зберігаємо в окремому масиві номери вершин, сполучених із даною вершиною ребром (тобто *суміжних* вершин).

Одержати з першого варіанта подання графа (переліку ребер) другий (списки суміжності) нескладно:

1. Запровадимо для кожної вершини окремий масив суміжних вершин, який спершу є порожнім.
2. Перебираючи ребра від першого до останнього, робимо таке: якщо на даному кроці ми розглядаємо ребро між вершинами *A* та *B*, то до списку суміжності вершини *A* додаємо вершину *B*, а до списку суміжності вершини *B* додаємо вершину *A*.
3. Закінчивши перебір ребер, матимемо подання графа списками суміжності.

Завдання

За списками суміжності, утвореними за допомогою вищенаведеного алгоритму, відновіть порядок, у якому було перераховано ребра графа в початковому переліку.

Вхідні дані

У першому рядку вхідного файлу вказано натуральне число *n*, що не перевищує 1000, — кількість вершин графа. Вершини занумеровано натуральними числами від 1 до *n*. Далі йдуть *n* рядків: у *k*-му з них перераховано без повторів номери вершин, суміжних з *k*-ю. Усі такі номери відмінні від самого числа *k*. Відомо також, що кожна вершина сполучена принаймні з однією іншою і якщо в списку суміжності вершини *k* є вершина *l*, то і в списку суміжності вершини *l* є вершина *k*.

Вихідні дані

У вихідний файл виведіть ребра графа в тому порядку, в якому вони йшли у початковому наборі. Одному ребру має відповідати один рядок: першим у рядку слід зазначити менший з двох номерів сполучених між собою вершин, другим — більший. Якщо є декілька варіантів порядку, в якому могло бути перераховано ребра, виведіть будь-який із них. Вхідні дані гарантують, що принаймні один такий порядок існує.

Приклад

graph.in	graph.out
4 2 4 4 1 4 2 3 1	2 4 1 2 3 4 1 4

4. Табори

Максимальна оцінка: 200 балів

Обмеження на час: 1 сек.

Обмеження на пам’ять: 32 МБ

Вхідний файл: camps.in

Вихідний файл: camps.out

Програма: camps.*

Група альпіністів планує піднятися на вершину гори стежкою, з якої неможливо звернути. Розташування на маршруті однозначно визначено відстанню від старту. Телевізійна компанія, яка профінансувала підйом, обумовила, що протягом підйому група розіб’є *n різних* таборів для ночівлі й відпочинку, де пройдуть відеозйомки з життя альпіністів.

Завдання

Визначити, скількома способами можна встановити *n* таборів для ночівлі й відпочинку за умови, що і s — довжина стежки, і s_1, s_2, …, s_n — відстані від старту до точок розташування таборів (у певних одиницях вимірювання довжини) можна подати різними натуральними числами, $0 < s_1 < s_2 < \ldots < s_n < s$.

Вхідні дані

Вхідний файл містить у вказаному порядку два *натуральні* числа:

s — довжина стежки;

n — кількість таборів, $n < s < 200\,000$.

Вихідні дані

Вихідний файл має містити одне натуральне число — шукану кількість способів розташування таборів. Відомо, що кількість цифр десяткового запису цієї кількості не перевищує 45 000.

Приклади

camps.in	camps.out
3 2	1
3 1	2
99 49	254776122589808569027304286 00

5. Дороги

Максимальна оцінка: 200 балів

Обмеження на час: 9 сек.

Обмеження на пам’ять: 256 МБ

Вхідний файл: roads.in

Вихідний файл: roads.out

Програма: roads.*

Керівництво будь-якої держави має постійно піклуватися про дороги, які сполучають усі населені пункти держави у єдине ціле та слугують як військовій, так і економічній могутності держави.

Завдання

Визначити набір доріг, за допомогою яких можна потрапити з будь-якого населеного пункту у будь-який інший населений пункт і утримання яких обходиться якомога дешевше для державної скарбниці. Рух кожною дорогою — двосторонній.

Вхідні дані

Перший рядок вхідного файлу містить у вказаному порядку два *натуральні* числа:

- n — кількість населених пунктів, занумерованих натуральними числами від 1 до n включно, $3 \le n \le 4096$;
- m — кількість усіх доріг, $3 \le m \le 8\,386\,560$.

Кожний з наступних m рядків містить опис певного шляху:

- j, k — номери населених пунктів, сполучених дорогою, $1 \le j, k \le n, j \ne k$;
- v — натуральне число — вартість утримання дороги в солідусах (римських золотих монетах), $1 \le v \le 654321$.

Вихідні дані

Єдиний рядок вихідного файлу містить номери доріг *у порядку зчитування*, що утворюють потрібний набір. Порядок цих номерів — довільний. Вхідні дані гарантують існування розв'язку. Якщо розв'язків кілька, потрібно записати будь-який, але лише один.

Приклад

roads.in	roads.out
3 3 1 2 3 2 3 1 3 1 2	2 3

6. Парасолька

Максимальна оцінка: 200 балів

Обмеження на час: 0,15 сек.

Обмеження на пам'ять: 32 МБ

Вхідний файл: umbrella.in

Вихідний файл: umbrella.out

Програма: umbrella.*

Фірма-виробник парасольок з рекламною метою обіцяє покупцям унікальність кожної виробленої нею парасольки. Цього планують

досягнути за рахунок різного розфарбування секторів купола, які неможливо сумістити обертанням навколо стержня парасолі. Всі сектори купола однакової форми.

Завдання

Визначити кількість різних видів розфарбувань купола парасольки.

Вхідні дані

Вхідний файл містить у вказаному порядку *натуральні* числа:

n — кількість секторів купола парасолі, $2 \le n \le 19$;

m — кількість можливих кольорів, $2 \le m \le 19$.

У 55% тестів $n \le 6$. У 70% тестів $m \le 6$.

Вихідні дані

Вихідний файл має містити одне натуральне число — шукану кількість різних способів розфарбування куполу парасолі. Вхідні дані гарантують, що добуток n і цієї шуканої кількості не перевищує 10^{19}.

Приклади

umbrella.in	umbrella.out
3 2	4
3 4	24

5. Ідеї розв'язання завдань III етапу

1. Числа

Позначимо задані у вхідному файлі два числа через a та b.

Ідейно найпростіший шлях розв'язати задачу — скориставшись тим, що жодне з чисел у дружній трійці не може перевищувати добутку двох інших, перебрати всі числа від 1 до ab і безпосередньо порахувати, скільки з них дають разом із числами a та b дружню трійку. Утім, такий розв'язок набере неповний бал, оскільки не витримає обмеження на час.

Щоб заробити повний бал, необхідно розкласти обидва числа на прості множники. Якщо деяке просте число p входить у степені p_a у розклад числа a і в степені p_b у розклад числа b, то в розклад третього числа дружньої трійки число p не може входити у степені, більшому за $p_a + p_b$ (інакше добуток ab не поділиться на третє число). Водночас цей степінь не може бути меншим за $|p_a - p_b|$ (інакше добуток одного з чисел a та b і третього числа не поділиться на інше з чисел a та b). При цьому одне з чисел p_a та p_b може бути й нулем (таке станеться, коли одне з чисел a та b ділиться на просте число p, а інше — ні). Якщо взяти всі прості числа, на які ділиться бодай одне з чисел a та b, то ці обмеження на степінь їх входження будуть водночас і достатніми для того, щоб третє число дійсно становило дружню трійку з двома заданими. На жодне ж інше просте число воно ділитися не може.

Отже, відповідь — добуток чисел вигляду $(p_a + p_b) - |p_a - p_b| + 1$ по всіх простих числах p, що входять у розклад хоча б одного з чисел a та b. Щоб порахувати цей добуток, достатньо, наприклад, перебрати всі числа від 2 до одного з чисел a та b у порядку збільшення: якщо хоча б одне з чисел a та b поділилося на чергове число, яке ми розглядаємо, то ділимо їх, поки діляться, обчислюючи таким чином p_a та p_b, після чого домножуємо відповідь на відповідну величину.

Лишається лише врахувати те, що в отриману відповідь ми могли включити одне чи обидва числа a та b, якщо вони задовольняють умови дружньої трійки, окрім обмеження на те, що числа мають бути різними. Тому після відповідної перевірки кожного з цих чисел, можливо, повинні будемо відняти від результату одну чи дві одиниці.

2. Трикутник

Простий перебір усіх трійок заданих довжин, звичайно, не пройде великі тести за обмеженням на час. Тому використаємо інший підхід. Спершу ефективно відсортуємо заданий набір чисел. А тоді залежно від підзадачі, яку розв’язуємо, зауважимо:

- Якщо ми шукаємо трикутник найбільшого периметра, то його сторони будуть трьома послідовними елементами у відсортованому масиві. Якби це було не так, ми могли б узяти дві менших сторони і збільшити їх до тих двох, які передують у відсортованому масиві більшій стороні. При цьому трикутник не перестав би існувати (оскільки сума двох менших сторін лише збільшилась, а більша сторона залишилась тією самою), а от периметр тільки збільшився б. Отже, залишається за лінійний час знайти у відсортованому масиві найбільшу (найближчу до кінця масиву) трійку послідовних елементів, які задовольняють нерівність трикутника.
- Якщо ми шукаємо трикутник найменшого периметра, то дві його більші сторони будуть послідовними елементами у відсортованому масиві. Якби це було не так, ми могли б узяти більшу сторону і зменшити її до тієї довжини, що йде в масиві одразу після середньої сторони трикутника. При цьому трикутник не перестав би існувати (оскільки сума двох менших сторін не змінилася, а більша сторона стала коротшою), а от периметр тільки зменшився б. Отже, залишається перебрати всі пари послідовних елементів масиву і для кожної за допомогою двійкового пошуку у лівій частині масиву знайти найменше число, яке б у сумі з середньою стороною перевищувало довжину найбільшої. Далі порівняти периметри усіх знайдених трикутників і вивести найменший.

Враховуючи, що для обох підзадач потрібно здійснити сортування, складність алгоритму завжди становить $O(n \log n)$.

Наостанок додамо, що периметр трикутника може вийти за межі стандартної знакової 4-байтової змінної, тож для зберігання відповіді потрібно використовувати, скажімо, або змінну беззнакового типу, або 8-байтову.

3. Граф

Опишемо спочатку *структуру даних* для списків суміжності вершин.

Черга — структура даних, що працює за принципом «хто раніше прийшов, той раніше пішов». У черги є голова та хвіст. Основні операції з чергою такі:

- «Поставити в чергу» — додати елемент у «хвіст» черги. Довжину черги при цьому буде збільшено на одиницю.
- «Отримати з черги» — повернути значення елемента з голови та видалити його з черги, встановлюючи голову черги на наступний за видаленим елемент. Довжину черги при цьому буде зменшено на одиницю.

Можливо реалізувати чергу за допомогою масиву a[1...n], в якому зберігають дані, та двох додаткових змінних head і tail, у яких зберігають індекси відповідно «голови» та «хвоста» черги. Основні операції можна записати кількома рядками:

«поставити в чергу»	**«отримати з черги»**

`if tail=n then tail:=1`
`else`
`tail:=tail+1;`
`a[tail]:=x;`
	`x:=a[head];`
	`if head=n then head:=1`
	`else`
	`head:=head+1;`

Але у випадку багатьох черг різної довжини чергу доречно реалізувати з використанням динамічного розподілу пам’яті (або скористатися вже готовою структурою даних queue у C++). Крім черг для списків суміжності вершин, одну додаткову чергу edges потрібно використати для незаблокованих ребер (означення див. далі).

Опишемо *жадібний алгоритм* розв’язання задачі: послідовно виписувати ті ребра, які в таблиці суміжності не заблоковано жодним іншим (ще не виписаним) ребром, і видаляти їх із таблиці.

Незаблокованими вважати ребра (*A*, *B*) з такими властивостями:

- першою у списку суміжності вершини *A* іде на даний момент

вершина *B;*

- першою у списку суміжності вершини *B* іде на даний момент вершина *A*.

Початковим набором незаблокованих ребер чергу edges можна заповнити просто під час зчитування вхідних даних або за допомогою додаткового лінійного проходу. Після цього чергу потрібно пройти «від голови до хвоста» і на кожному кроці:

- вивести й видалити з черги поточне ребро (*A*, *B*);
- видалити це саме ребро з таблиці суміжності — видалити перші вершини з черг вершин *A* і *B*;
- проаналізувати нові перші вершини в чергах *A* та *B* і в разі необхідності додати одне або два нових ребра до черги незаблокованих ребер.

Процес завершується тоді, коли в черзі більше немає ребер.

Як можна зрозуміти, час виконання алгоритму є пропорційним до кількості чисел у вхідному файлі.

4. Табори

Шукана кількість дорівнює кількості способів вибору *n* різних натуральних чисел з множини {1, 2, …, *s* – 1} — кількості *n*-елементних підмножин (*s* – 1)-елементної множини:

$$C_{s-1}^{n} = \binom{s-1}{n} = \frac{(s-1)!}{n!(s-1-n)!}$$

Можна вибрати один з таких алгоритмів.

1. Скоротити на більший з факторіалів-добутків у знаменнику і, використавши алгоритм Евкліда, скоротити на найбільші спільні дільники множники чисельника та знаменника, щоб подати знаменник добутком одиниць. Це було прийнятно для розв’язання задачі «Каса» III (міського) етапу Всеукраїнської учнівської олімпіади з інформатики у місті Києві у 2015 році, але у даному випадку це не гарантує дотримання обмежень на час.

2. Попередньо обчисливши усі прості числа від 1 до s (решето Ератосфена), знайти степінь простого числа p у канонічному розкладі $m!$ на прості множники: $[m/p] + [m/p^2] + [m/p^3] + \ldots$ і використати отриману інформацію для знаходження добутку. Саме цей підхід реалізовано в авторському розв'язанні цієї задачі.

На останньому етапі — множенні чисел базового типу — для повного розв'язання потрібно використати подання числа масивом його цифр — «довгу арифметику». Інакше останні 3 тести пройти неможливо.

5. Дороги

У термінах теорії графів дану задачу називають пошуком мінімального кістякового дерева — зв'язного графа без циклів, що містить усі вершини початкового графа і має найменшу суму ваг ребер серед усіх таких графів. Одним із способів розв'язання цієї задачі є жадібний *алгоритм Прима* (Prim)*:*

1. Утворити дерево T_1, що містить:
   - одне ребро, що має *найменшу вагу* серед ребер початкового графа;
   - дві вершини — кінці цього ребра найменшої ваги

   та надати значення $k = 1$.
2. Якщо існують вершини початкового графа G зовні останнього побудованого дерева T_k з ребрами $e_1, e_2, e_3, \ldots, e_k$, то зробити таке:
   - вибрати ребро e_{k+1} з найменшою вагою серед тих, у яких одна вершина належить до T_k, а інша вершина не належить;
   - утворити дерево T_{k+1} долученням до T_k вибраного ребра e_{k+1} і його вершини, яка не належить до T_k;
   - збільшити значення k на 1;
   - перейти на початок виконання пункту 2.
3. Якщо всі вершини початкового графа G належать до дерева T_k, то припинити побудову мінімального кістякового дерева.

Теорема. *Алгоритм Прима породжує мінімальне кістякове дерево.*

Доведення (від супротивного). Позначимо через e_1, e_2, e_3, …, e_n ребра дерева T_n у тому порядку, як їх вибрано згідно з алгоритмом Прима. Нехай k — найбільше таке ціле невід'ємне число, при якому e_1, e_2, e_3, …, e_k — ребра деякого мінімального кістякового дерева T'. Припустимо, що $k < n$. Приєднаємо ребро e_{k+1} до мінімального кістякового дерева T'. В утвореному таким чином графі є цикл. Цикл містить ребро e, відмінне від e_{k+1}, що також має лише одну вершину в дереві T_k. Утворимо дерево T'', вилучивши з T' ребро e і долучивши ребро e_{k+1}. Згідно з алгоритмом Прима, вага e_{k+1} не перевищує вагу e, тому T'' є мінімальним каркасним деревом, що суперечить означенню k. Отже, $k = n$, T_n — мінімальне каркасне дерево.

Алгоритм Прима використано в авторському розв'язанні для калібрування системи тестування. На думку автора завдання, найімовірніше очікувати використання саме цього алгоритму учасниками олімпіади. Приклад вихідного файлу може наштовхнути на думку використати саме цей жадібний алгоритм.

Не менше відомий *алгоритм Крускала* (Kruskal):

1. Надати множині E значення величини $\emptyset$ — порожньої множини.
2. Визначити, які з ребер початкового графа, що не належать до E, при долученні до E не утворюють циклів.
3. Якщо такі ребра є, то:
    - серед цих ребер визначити «найлегше» — з найменшою вагою;
    - долучити це ребро до множини E;
    - перейти до виконання пункту 2.
4. Якщо таких ребер немає, то припинити побудову мінімального кістякового дерева E.

Учасникам відбірково-тренувальних зборів радимо звернути увагу на *алгоритм Борувки,* що він гарантує максимальну кількість балів. Він полягає у послідовному долученні ребер до лісу, що містить усі вершини, а

ребра утворюють окремі дерева, доки ліс не перетвориться на одне дерево:

1. Нехай спочатку T — порожня множина ребер. Інакше кажучи, починаємо з лісу, до якого кожна вершина входить як окреме дерево (без ребер).
2. Поки T не є деревом (кількість ребер у T менше ніж $V - 1$, де V — кількість вершин у графі), робити таке:
   - для кожної компоненти зв'язності, що є деревом поточного лісу, знайти одне найлегше ребро, що пов'язує цю складову з будь-якою іншою складовою зв'язності. Якщо таких ребер кілька, вибрати довільне, але лише одне;
   - долучити усі знайдені ребра до множини T.

Отримана у результаті ребер T є мінімальним кістяковим деревом початкового графа. Уперше цей алгоритм опубліковано 1926 року Отакаром Борувкою як метод пошуку оптимальної електричної мережі у Моравії.

У програмі найголовніше організувати облік належності вершин до складових (дерев). Для цього в авторському розв’язанні використано лінійні масиви:

- c — утворений записами з такими полями:
  - p — вказівник на попередній елемент опису складової;
  - f — вказівник на перший елемент опису складової;
  - k — номер ребра, що належить до складової або яке потрібно долучити;
  - m — вага ребра з номером k;
- p — вказує на елемент c, що завершує опис складової, що містить дану вершину;
- v — вказує на елемент c, що завершує опис даної складової.

При цьому для опису динамічної структури не потрібно використовувати динамічний розподіл пам’яті.

6. Парасолька

Можливі шляхи розв’язання такі:

1. *Перебір* можливих розфарбувань, які неможливо сумістити поворотами навколо стержня парасолі.
2. *Подання шуканої кількості многочленом змінної m:*
   a) на основі аналізу *учасником* різних способів розфарбування у термінах базових понять комбінаторики;
   b) використовуючи *обчислення за допомогою ПК* на основі наслідків теореми Редфілда — Пойа.

Спосіб 1 найпрозоріший, але найменш результативний.

Спосіб 2а проілюструємо на прикладі $n = 4$, використавши такі позначення:

$A_m^k = \dfrac{m!}{(m-k)!}$ — кількість розташувань без повторення по k елементів з m;

$C_m^k = \dfrac{m!}{k!(m-k)!}$ — кількість k-елементних підмножин m-елементної множини.

У наступній таблиці у стовпчику ліворуч подано різні схеми розфарбування секторів (цифрами 1, 2, 3, 4 позначено різні кольори розфарбування), у стовпчику праворуч — кількості відповідних розфарбувань.

11 11	$C_m^1 = m$
12 21	$C_m^2 = m(m-1)/2$
11 22	$C_m^2 = m(m-1)/2$
12 22	$A_m^2 = m(m-1)$
11	$A_m^3 = m(m-1)(m-2)$

23	
12 31	$C_m^1 \cdot C_{m-1}^2 = m(m-1)(m-2)/2$
12 43	$A_m^4 / 4 = m(m-1)(m-2)(m-3)/4$

Використання результатів цього аналізу та схожого аналізу для n = 2, 3, 5, 6 гарантує нарахування більше половини балів за задачу.

Теоретичні основи способу 2b детально з доведенням викладено за адресою http://www.kievoi.ippo.kubg.edu.ua/kievoi/lectures/polya.html. У розділі 6 цієї публікації згадано про *еквівалентну* популярну у навчальній літературі задачу про знаходження кількості різних намист, що складаються з намистинок x кольорів і містять по n намистинок. Два намиста вважають різними, якщо їх не можна сумістити рухами *у площині* (не плутати з рухами площини!). Подано і формулу для обчислення кількості намист:

$$\frac{1}{n}\sum_{d|n}\varphi(d)x^{n/d} .$$

У цій формулі:

- додавання $\sum$ здійснюють за всіма d — натуральними дільниками числа n;
- $\varphi(d)$ — функція Ейлера — кількість натуральних чисел, які менші від числа d і взаємно прості з ним, — дорівнює добутку d та різниць вигляду $(1 - 1/p)$, де p — простий дільник d.

Саме цей спосіб (m підставити у формулу вище замість x) гарантує успішне проходження усіх тестів. Його реалізовано у авторському розв'язанні. Він прийнятний для істотно більших значень n та m, якщо використати подання числа масивом його цифр («довга арифметика») або знаходити не саму кількість, а лишок від ділення на число базового типу мови програмування.

Для 60 тестів використано такі множини значень:

- n — {2,3,4,5,6,11,12,18,19};
- m — {2,3,4,5,6,11,19}.

Не використано лише пари (18, 19), (19, 11) і (19,19), при яких сума $\sum$ — поза діапазоном значень базового типу qword мови Pascal.

6. Авторські розв'язання завдань III етапу

1. Числа

```
/* GCC */

#include <stdio.h>
#include <algorithm>

using namespace std;

// Повертає показник найбільшого степеня
// числа divisor, на який ділиться число n.
// Паралельно ділить число n на цей
// степінь.
inline int getPowerAndDivide(int &n,
                             int divisor)
{
    int power = 0;
    while (n % divisor == 0)
    {
        power++;
        n /= divisor;
    }
    return power;
}

// Перевіряє, чи три заданих числа утворю-
// ють дружню трійку, ігноруючи те, чи вони
// різні.
inline bool check(int a, int b, int c)
{
    return a * b % c == 0 && a * c % b == 0
                         && b * c % a == 0;
}

int main()
{
    freopen("numbers.in", "r", stdin);
    freopen("numbers.out", "w", stdout);

    int a, b;
    scanf("%d %d", &a, &b);
```

```
    int ans = 1;

    int m = min(a, b), curA = a, curB = b;
    for (int divisor = 2; divisor <= m;
                          divisor++)
    {
        int powerA = getPowerAndDivide(
                             curA, divisor);
        int powerB = getPowerAndDivide(
                             curB, divisor);
        // Якщо принаймні одне з чисел
        // powerA та powerB виявилося нену-
        // льовим, то поточний дільник
        // (divisor) є за побудовою алго-
        // ритму простим числом.
        ans *= (powerA + powerB) -
                  abs(powerA - powerB) + 1;
    }

    // Поправка на те, що жодні два числа
    // не повинні збігатися:
    if (check(a, a, b))
        ans--;
    if (check(a, b, b))
        ans--;

    printf("%d\n", ans);

    return 0;
}
```

2. Трикутник

```
/* GCC */

#define maxN 1000000 // Найбільша можлива
                     // кількість відрізків

#include <stdio.h>
#include <algorithm>

using namespace std;

unsigned int a[maxN]; // Довжини відрізків
int n;

int main()
{
    freopen("triangle.in", "r", stdin);
    freopen("triangle.out", "w", stdout);

    scanf("%d\n", &n);
    for (int i = 0; i < n; i++)
```

```
        scanf("%u", &a[i]);
    scanf("\n");

    char c[4];
    scanf("%3s", c);

    bool findMax = c[1] == 'a';

    sort(a, a + n);

    if (findMax) // Шукаємо найбільший
    {            // периметр
        for (int i = n - 3; i >= 0; i--)
            if (a[i] + a[i + 1] > a[i + 2])
            {
                printf("%u\n", a[i] +
                      a[i + 1] + a[i + 2]);
                return 0;
            }
    } else // Шукаємо найменший периметр
    {
        unsigned int curMin = 0;
        for (int i = 2; i < n; i++)
        {
            // Шукаємо найменшу сторону
            // трикутника, яка б у сумі з
            // (i-1)-ю перевищила б i-ту:
            int index = lower_bound(
    a, a + i - 1, a[i] - a[i - 1] + 1) - a;
            if (index < i - 1) // Якщо це
            // вдалося, порівнюємо з
            // поточним мінімумом:
            {
                unsigned int curValue =
                a[index] + a[i - 1] + a[i];
                if (curMin == 0 ||
                         curValue < curMin)
                    curMin = curValue;
            }
        }

        if (curMin > 0)
        {
            printf("%u\n", curMin);
            return 0;
        }
    }

    printf("none\n");
    return 0;
}
```

3. Граф

```
/* GCC */

#include <stdio.h>
#include <vector>
#include <queue>

using namespace std;

int n; // Кількість вершин графа
vector<queue<int> > all; // all[v - 1] —
// список суміжності вершини v
queue<pair<int, int> > edges; // Черга з
// ребер, які виводимо у вихідний файл

// Видаляє передній елемент зі списку сумі-
// жності вершини v та оновлює чергу ребер,
// які ми виводимо у вихідний файл, додавши
// туди нове доступне ребро, якщо таке
// з'явилося.
inline void pop(int v)
{
    all[v - 1].pop();
    if (!all[v - 1].empty())
    {
        int adjacent = all[v - 1].front();
        if (!all[adjacent - 1].empty() &&
            all[adjacent - 1].front() == v)
            edges.push(make_pair(
      min(v, adjacent), max(v, adjacent)));
    }
}

int main()
{
    freopen("graph.in", "r", stdin);
    freopen("graph.out", "w", stdout);

    scanf("%d\n", &n);

    // Зчитуємо інформацію та заносимо
    // початкові ребра в чергу edges:
    for (int v = 1; v <= n; v++)
    {
        queue<int> adj; // Список вершин,
                        // суміжних з поточною
        int num = 0; // Поточне зчитуване
                     // число
        while (true)
        {
            char c;
            scanf("%c", &c); // Оскільки
```

```
            // кількість чисел у рядку не-
            // відома, зчитуємо посимвольно
            if (c >= '0' && c <= '9')
            // Якщо черговий зчитаний
            // символ — цифра
                num = num * 10 + (c - '0');
            else // Якщо черговий зчитаний
                 // символ — пробіл або
                 // перенесення рядка
            {
                adj.push(num);
                num = 0;
            }
            if (c == '\n') // Якщо ми дійш-
            // ли до кінця рядка, виходимо
            // з циклу
                break;
        }
        all.push_back(adj);
        // Якщо передня вершина списку су-
        // міжності уже опрацьована, і в її
        // списку суміжності поточна роз-
        // глядувана вершина теж є перед-
        // ньою, додаємо це ребро до черги:
        if (adj.front() < v &&
         all[adj.front() - 1].front() == v)
            edges.push(
                make_pair(adj.front(), v));
    }

    while (!edges.empty())
    {
        pair<int, int> edge = edges.front();
        edges.pop();
        printf("%d %d\n", edge.first,
                          edge.second);
        pop(edge.first);
        pop(edge.second);
    }

    return 0;
}
```

4. Табори

```
{ Free Pascal }
const
   m=200000;              {Верхня межа n}
  mc=5000;
base=1000000000; {Основа системи числення}

var j,k,l,  {Лічильники}
    h,n,    {Вхідні дані}
    hn,     { = h-n-1}
```

```
    nc, {Кількість цифр відповіді}
    np, {Кількість знайдений простих чисел}
    z:   longint;
  y,r: qword;
    o: text;
 s: array[1..m]  of boolean; {Число просте?}
 c: array[1..mc] of qword;   {Цифри відповіді}
 p: array[1..m]  of longint; {Прості числа}
pp: array[1..m]  of longint; {Степені простих
чисел у розкладі відповіді на прості множники}
                                          BEGIN
assign(o,'camps.in');     reset(o);
  read(o,h,n);          close(o);
assign(o,'camps.out');  rewrite(o);
dec(h);
hn:=h-n;
               {Пошук простих чисел}
for j:=1 to h do s[j]:=true;
for l:=2 to h div 2 do if s[l] then
for k:=2 to h div l do s[k*l]:=false;
np:=0;
for j:=2 to h do
if s[j] then begin
    inc(np);
      p[np]:=j;
     pp[np]:=0 end;

      {Пошук степенів простих чисел}
for j:=1 to np do begin
  y:=1;
  repeat y:=y*p[j];
         z:=h div y;
         inc(pp[j],z);

   until z=0;
  y:=1;
  repeat y:=y*p[j];
         z:=n div y;
         dec(pp[j],z);
   until z=0;
  y:=1;
  repeat y:=y*p[j];
         z:=hn div y;
         dec(pp[j],z);
   until z=0      end;

       {Пошук цифр відповіді}
nc:=1;
c[1]:=1;
for j:=1 to np    do
for k:=1 to pp[j] do  begin

r:=0;
```

```
for l:=1 to nc do begin
     r:=r+p[j]*c[l];
  c[l]:=r mod base;
     r:=r div base  end;
while r>0 do      begin
  inc(nc);
  c[nc]:=r mod base;
      r:=r div base end end;
       {Запис відповіді}
write(o,c[nc]);
for j:=nc-1 downto 1 do
if c[j]<10 then write(o,'00000000',c[j])else
if c[j]<100 then write(o,'0000000',c[j])else
if c[j]<1000 then write(o,'000000',c[j])else
if c[j]<10000 then write(o,'00000',c[j])else
if c[j]<100000 then write(o,'0000',c[j])else
if c[j]<1000000 then write(o,'000',c[j])else
if c[j]<10000000 then write(o,'00',c[j])else
if c[j]<100000000 then write(o,'0',c[j])
                   else write(o,   c[j]);
writeln(o);
close(o)
END.
```

5. Дороги

Алгоритм Прима

```
{ Free Pascal }
{$I+} {Верхні межі кількості:}
const nv_max=4096;    {вершин}
      ne_max=8386560;  {ребер}
type edge=record      {Ребро:}
  w: longint;           {вага}
a,b: word;           {вершини}
  n: longint           {номер}
             end;
var e: array[0..ne_max] of edge;
    u: array[1..nv_max] of boolean;
               {Кількості:}
   nv,         {вершин}
   nt: word;   {вершин дерева}
   ne,         {ребер}
    k: longint;
    o: text;
                 {Обмін значень}
procedure swap(j,k: longint); BEGIN
      e[0]:=e[j];
            e[j]:=e[k];
                  e[k]:=e[0]    END;

                 {Упорядкування}
```

```
procedure QSort(l,r: longint);
  var x: edge;  i,j: longint;
                        BEGIN
x:=e[l+random(r-l+1)];
i:=l;  j:=r;
while i<=j do  begin
  while e[i].w<x.w do inc(i);
  while e[j].w>x.w do dec(j);
  if i<=j then begin
    swap(i,j);
    inc(i);
    dec(j)   end end;
if l<j then qsort(l,j);
if i<r then qsort(i,r)   END;
                         BEGIN
assign(o,'roads.in');
 reset(o);
readln(o,nv,ne);
for k:=1 to ne do          begin
  readln(o,e[k].a,e[k].b,e[k].w);
  e[k].n:=k                  end;
close(o);

randomize;
qsort(1,ne);
for k:=1 to nv do u[k]:=false;
 assign(o,'roads.out');
rewrite(o);

write(o,e[1].n);
u[e[1].a]:=true;
u[e[1].b]:=true;

nt:=2;
repeat  k:=1;
  repeat inc(k)
   until (u[e[k].a] and (not u[e[k].b]))
      or (u[e[k].b] and (not u[e[k].a]));
  inc(nt);
  write(o,' ',e[k].n);
  u[e[k].a]:=true;
  u[e[k].b]:=true;
until nt=nv;
writeln(o);
close(o)  END.
```

Алгоритм Борувки

```
{ Free Pascal }
      {Верхні межі кількості:}
const nv_max=4096;    {вершин}
      ne_max=8386560;  {ребер}
```

```
         max=654322;
type edge=record      {Ребро:}
          k,w: longint; {вага}
          a,b: word; {вершини}
                          end;
var a,b,
               {Кількості:}
   nv,         {вершин}
   np,         {складових}
  nte: word;   {використаних ребер}
   ne,         {ребер}

i,j,k,l,m: longint;
        o: text;
e: array[0..ne_max] of edge;   {Ребра}

                    {Частини складової}
c: array[1..nv_max] of record
            p,             {попередник}
            f: word;          {початок}
            k,             {нове ребро}
            m:     {вага нового ребра}
                         longint; end;
{Вказівники на початок опису складової}
p,                          {для вершин}
v:                       {для складових}
   array[1..nv_max] of word;
                          BEGIN
assign(o,'roads.in');
 reset(o);
readln(o,nv,ne);
for k:=1 to ne do
readln(o,e[k].a,e[k].b,e[k].w);
close (o);
assign(o,'roads.out');
rewrite(o);
for k:=1 to nv do begin
  p[k]:=k;
  v[k]:=k;
  c[k].p:=0;
  c[k].f:=k;
  c[k].m:=max       end;
nte:=0;
 np:=nv;
  i:=0;  {Кількість складових з
          відомими наступними ребрами}
REPEAT
{Пошук наступних ребер найменшої ваги}
  k:=0;  {Лічильник розглянутих ребер}
  repeat inc(k);
    if  p[e[k].a]<>p[e[k].b]  then begin
    if c[p[e[k].a]].m >e[k].w then begin
       if c[p[e[k].a]].m=max then inc(i);
```

```
      c[p[e[k].a]].m:=e[k].w;
      c[p[e[k].a]].k:=k             end;
    if c[p[e[k].b]].m >e[k].w then begin
      if c[p[e[k].b]].m=max then inc(i);
      c[p[e[k].b]].m:=e[k].w;
      c[p[e[k].b]].k:=k         end end;
  until {i=np} k=ne;

  l:=0; {Кількість складових з
         невідомими наступними ребрами}

  repeat
    inc(l);
    if c[v[l]].m<>max then BEGIN
      dec(i);
      inc(nte);
      j:=c[v[l]].k;
      if v[l]=p[e[j].b]
      then begin a:=e[j].a; b:=e[j].b end
      else begin a:=e[j].b; b:=e[j].a end;

      if((p[e[c[p[a]].k].a]=p[e[c[p[b]].k].a])
      and(p[e[c[p[a]].k].b]=p[e[c[p[b]].k].b]))
      or((p[e[c[p[a]].k].a]=p[e[c[p[b]].k].b])
      and(p[e[c[p[a]].k].b]=p[e[c[p[b]].k].a]))
      then       begin
        c[p[a]].m:=max;
        dec(i)     end;

      c[c[p[a]].f].p:=p[b];
        c[p[a]].f :=c[p[b]].f;
      j:=p[b];
      repeat p[e[c[j].k].a]:=p[a];
             p[e[c[j].k].b]:=p[a];
             j:= c[j].p
      until  j=0;
      v[l]:=v[np];
      dec(l);
      dec(np);              END;
  until (nte=nv-1) or (l=np);
until nte=nv-1;
k:=c[p[1]].p;
write(o,c[k].k);
repeat k:=c[k].p;
       write(o,' ',c[k].k);
 until c[k].p=0;
writeln(o);
close(o)  END.
```

6. Парасолька

```
{ Free Pascal }
```

```
const
   m=100000;              {Верхня межа n}

var j,k,l,  {Лічильники}
    n,c,    {Вхідні дані}
    np: word; {Кількість простих дільників n}
answer: qword;
     o: text;
 s: array[1..m]  of boolean; {Число просте?}
 p: array[1..m,0..64]  of longint; {Прості числа}
pn,pd: array[1..m] of byte;     {Степені
     простих чисел у розкладі n і відповіді
                          на прості множники}
procedure step(i: word);
var j,k,l,d: word;  f: qword;
                                BEGIN
for j:=0 to pn[i] do            begin
  pd[i]:=j;
  if i<np then step(i+1) else begin
  d:=1;
  for k:=1 to np do d:=d*p[k,pd[k]];
  f:=d;
  for k:=1 to np do if pd[k]>0 then
  f:=(f div p[k,1])*(p[k,1]-1);
  for k:=1 to n div d do f:=f*c;
  answer:=answer+f      end end END;
                                BEGIN
assign(o,'umbrella.in');      reset(o);
  read(o,n,c);                close(o);
assign(o,'umbrella.out');   rewrite(o);
                {Пошук простих чисел}
for j:=1 to n do s[j]:=true;
for l:=2 to n div 2 do if s[l] then
for k:=2 to n div l do s[k*l]:=false;
   {Розкладання n на прості множники}
np:=0;
for j:=2 to n do
if s[j] and (n mod j = 0) then begin
    inc(np);
     p[np,0]:=1;
    pn[np]:=0;
    l:=n;
    while l mod j = 0 do begin
      l:=l div j;
      inc(pn[np]);
      p[np,pn[np]]:=p[np,pn[np]-1]*j
      end   end;
answer:=0;
step(1);
writeln(o,answer div n);
close(o)
END.
```