ОЛІМПІАДА З ІНФОРМАТИКИ У МІСТІ КИЄВІ У 2015–2016 НАВЧАЛЬНОМУ РОЦІ

Мисак Данило Петрович,
керівник гуртка СШ №52 м. Києва.
Рудик Олександр Борисович,
доцент Київського університету імені Бориса Грінченка.

Стаття містить умови завдань II (районного) і завдань III (міського) етапів олімпіади з основ інформатики й обчислювальної техніки у місті Києві у 2015–2016 навчальному році й 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.

I. 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, у 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 периметру. 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 і хиж на острові відповідно. Відомо, що $2 \le h \le n \le 500\,000$. Сектори занумеровано числами від 1 до n у тому порядку, у якому вони йдуть на острові (при цьому сектори з номерами 1 і 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 й остання хижі, тож відповідь дорівнює 20–3=17.

У другому прикладі перша й 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 чле-

нами 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 із 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, $3 \le n \le 100\,000$. 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. Наразі 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 і компаній відповідно; $n \ge 3$, $m \ge 2$, а добуток цих двох чисел не перевищує мільйона. Кожен з наступних n рядків містить по m цифр, *не розділених пробілом*, і 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 journey.out
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.

II. ІДЕЇ РОЗВ'ЯЗАННЯ ЗАВДАНЬ II ЕТАПУ

1. Митько та подібні трикутники

Щоб зрозуміти, чи є трикутники подібними, можна перепробувати всі можливі варіанти порядку, у 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 й ігнорує остачу. Не варто користуватися і звичайним діленням у дійсних числах: при обчисленні й 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 визнає. 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 із можливих способів розв'язати задачу на повний бал такий: вивести одиницю, якщо пара з найменшої і середньої сторін першого трикутника пропорційна парі з найменшої і середньої сторін другого трикутника, а пара із середньої і найбільшої сторін першого трикутника пропорційна парі із середньої і найбільшої сторін другого трикутника; інакше вивести нуль.

2. Митько та дивовижний острів

Відстань між хижаками в секторах k та l (де $k \le l$) обчислюється за формулою $\min\{l-k,\ n+k-l\}$, тобто дорівнює меншій з двох відстаней: у випадку, якщо йти в порядку збільшення номера сектора, й у випадку, якщо

йти в порядку зменшення номера сектора. Ідейно найпростіший спосіб розв'язати задачу — перебрати всі пари хиж, порахувати відстань між кожною і 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 (уже 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. Отже, 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, утім 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 із 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. Наразі, звичайно, не слід забувати, що в обох прогресіях повинно бути рівно по n членів. Нарешті, щоб уникнути сортування під час виведення 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 розв'язати 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.

III. 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 ділення
// є цілочисельним, а виходити у
// дробові числа не хочеться (втрачається
// 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 другого
             // трикутників
  for (int t = 0; t < 2; t++) // t — номер
// трикутника
  {
for (int i = 0; i < 3; i++) // i —
// номер сторони
      scanf(«%d», &s[t][i]);
    // Упорядковуємо сторони трикутника,
    // 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 кількість
// 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, найвіддаленішої
// від поточної/попередньої зчитаної хижі
intmaxDistance = 0; // Найбільша відстань
   // між хижами на даний момент
for (int i = 1; i < h; i++) // Зчитуємо
   // всі хижі, крім першої (її вже зчитано
   // 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, збільшуючи
// відстань до поточної зчитаної 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 оновлюємо
// 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
// будуватиме програма
// Функція test пробує розбити послідовність
// на дві прогресії за припущення, що
// в однієї з прогресій (яку умовно назвемо
// першою) перший елемент дорівнює first, а
// другий — second. Якщо розбити послідовність
// на прогресії вдалося, повертає
// true, інакше — false. У разі успіху
// заповнює 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 різниці
              // 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++) // Намагаємось
// 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 першої
        // прогресії, його необхідно
        // додати саме в неї
p[0].push_back(all[i]);
else // Інакше число має належати
        // другій прогресії:
{
if (a[1] == 0) // Якщо це перший
         // елемент другої прогресії,
        // 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 обраховуємо
       // 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 підходить,
     // 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 першого
 // 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 перших
// і других членів прогресій.
bool test(int first, int second)
{
  vector<int> leftovers; // Числа, що
   // залишаться 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) // 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].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
 // Найбільша можлива кількість компаній
#include <stdio.h>
#include <queue>
using namespace std;
int n, m;
// Умасиві 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 номером company:
inline void setInfo(int planet,
            int company,
            bool value)
{
  data[getIndex(planet, company)] = value;
}
// Повертає інформацію про те, чи обслуговує
// 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 жодної
  // планети та компанії ще не відвідано:
  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 обслуговує
// дану планету:
        companyVisited[company] = true;
        // Запам'ятовуємо, що компанія
        // вже трапилась
for (int planet = 0; planet < n; planet++)
          if (planetVisited[planet] == −1
&& getInfo(planet, company))
{ // Для кожної планети, яку не проходили
 // раніше і яку обслуговує компанія:
            q.push(planet);
            // Додаємо планету до черги пошуку
            // Найкоротший шлях у дану
            // планету — дійти до поточної
            // 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;
}
```

Далі буде