1 Теоретичний матеріал

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

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

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

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

1.1 Задача «Платформи»

1.1.1 Базовий варіант задачі

Герой стрибає по платформах, що висять у повітрі. Він повинен перебратися від одного краю екрана до іншого. При стрибку з платформи на сусідню, герой витрачає $|y(2) - y(1)|$ енергії, де $y(1)$ і $y(2)$ — висо́ти цих платформ. Суперприйом дозволяє перескочити через платформу,

але на це затрачається $3 \cdot |y(3) - y(1)|$ *енергії. Суперприйом можна використовувати скільки завгодно (в т. ч. й 0) разів. Відомі висóти платформ у порядку зліва направо. Знайдіть мінімальну кількість енергії, достатню, щоб дістатися з 1-ї платформи до* N*-ї (останньої).*

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

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

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

Для подальших платформ, справедливо

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

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

Розберемо докладніше, звідки взялося це рівняння:

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

Тепер усі ці формули можна перетворити у програму, приблизно так:

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

(Якщо нема готової функції `min` — можна реалізувати її самостійно, або переписати вибір меншого через `if`. Код написаний у припущенні, що і платформи, й індекси нумеруються з 1 (Паскаль так вміє). У більшості

сучасніших мов програмування нумерація масивів починається з 0, тож це може призводити до не складни́х, але нудних технічних ускладень: треба або виділяти на один елемент більше (втрачаючи деякі бібліотечні засоби роботи зразу з усім масивом), або переписувати формули, зміщуючи індексацію. Але в нашій культурі все-таки не прийнято нумерувати з 0, тому в умовах та розборах задач буде переважно «людська» нумерація з 1, а питання, як це реалізовувати мовою програмування, як правило, віддаватиметься на розсуд читачів.)

1.1.2 Ітеративна та рекурсивна реалізації ДП

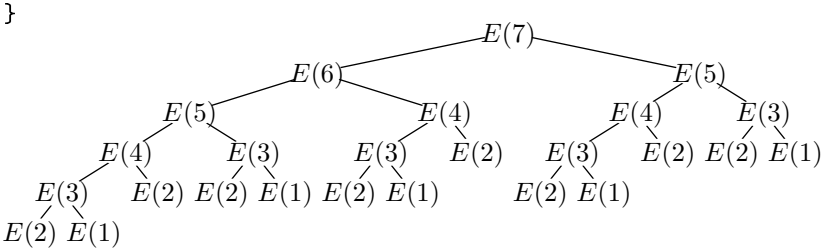
Розглянутий підхід (спочатку заповнити розв'язки тривіальних підзадач, потім у циклі застосовувати рівняння ДП, спираючись на вже готові відповіді й отримуючи нові, доки не будуть розв'язані всі) називають *ітеративною* (або *ітераційною*, що те са́ме) реалізацією. Можливий і альтернативний підхід —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

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

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

Тим не менш, релізація ДП через рекурсію можлива — якщо це рекурсія *із запам'ятовуваннями* (*memoized* recursion; не "memorized", а са́ме "memoized", так у англомовних першоджерелах). Її суть — перед запуском рекурсії перевірити, чи нема для цієї підзадачі готової (отриманої й запам'ятованої раніше) відповіді. Конкретний код рекурсії з за-

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

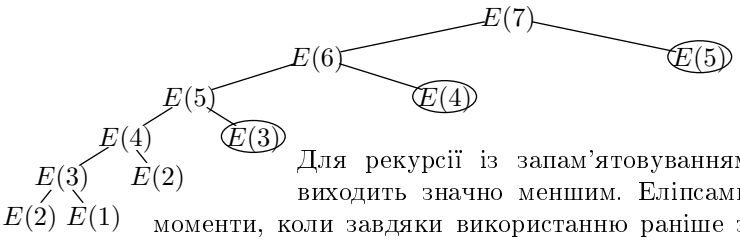


«Звичайна» рекурсія дуже багато разів розглядає ті самі підзадачі.

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

Відповідно, виклик цієї функції має бути приблизно таким:

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



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

пам'ятовуваннями для цієї ж задачі наведено на тій самій (тепер уже попередній) сторінці, під щойно згаданим деревом рекурсії.

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

Тут є багато моментів, які можна реалізувати інакше. Можна дотриматися традиції сучасних мов програмування, змістивши нумерацію на `0..N-1`; можна інакше ініціалізувати масив значеннями `-1` та/або позначати «ще не вирішено» якось інакше (але дуже бажано, щоб ця позначка не могла бути відповіддю вже розв'язаної підзадачі); тощо. То деталі. Важливо, щоб запам'ятовування було, і ним користувалися.

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

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

1.1.3 Зворотній хід (формування розгорнутої відповіді)

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

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

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

Дехто вважає, що формування розгорнутої відповіді більш технічна, ніж інтелектуальна робо́та, і їй не місце на олімпіадах. Але на олімпіадах такі завдання трапляються, а на практиці розгорнута відповідь часто важливіша за числову: щоб доїхати у реальній мережі доріг, знати, де і куди повертати, важливіше, ніж знати абсолютно точну відстань.

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

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

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

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

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

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

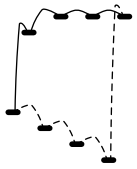
	1	2	3	4	5	6	7	8
y	3	1	4	1	5	9	2	6
E	0	2	3	2	6	10	15	19
prev	??	1	1	2	4	5	5	7

початок зворотнього ходу

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

1.1.4 А можна якось простіше?

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

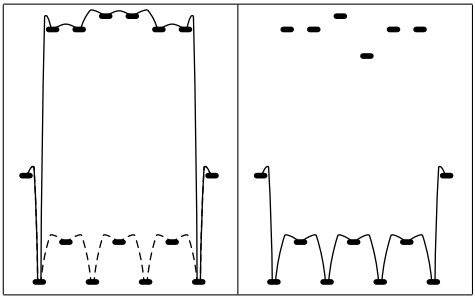


Неправильна «ідея» № 1: *«щоразу вибирати, який з переходів (на наступну платформу чи через одну) потребу́є менше енергії, і стрибати туди»*. Розглянемо вхідні дані «8 платформ, висо́ти 10 15 9 16 8 16 7 16» (див. рис.). Суцільною лінією виділено оптимальний маршрут, з затратами енергії $1 \cdot 5 +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

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

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

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



Розглянемо вхідні дані «15 платформ, висóти 20 12 31 15 31 12 32 15 32 12 31 15 31 12 20» (див. рис. (лівий), смисл суцільної та пунктирної ліній той сáмий, де пунктирної не видно, там вона проходить по суцільній). За цією «ідеєю» з $(y_2 = 12)$ ніби вигідно стрибати на $(y_4 = 15)$, але маршрут через верхні платформи $(y_3 = 31) \to (y_5 = 31) \to (y_7 = 32) \to (y_9 = 32) \to (y_{11} = 31) \to (y_{13} = 31)$ потребýє значно менше затрат, ніж через нижні $(y_4 = 15) \to (y_6 = 12) \to (y_8 = 15) \to (y_{10} = 12) \to (y_{12} = 15)$ (а сáме, $0 + 3 + 0 + 3 + 0 = 6$ проти $9 + 9 + 9 + 9 = 36$), так що «локально неоптимальний» стрибок $(y_2 = 12) \to (y_3 = 31)$ окупається на інших етапах.

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

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

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

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

1.1.5 Платформи з квадратичними вартостями стрибків — циклічні залежності між підзадачами

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

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

Несподівано, така зміна виявляється дуже істотною!

Герою *не заборонено стрибати назад*. Наприклад, при $y_1=10$, $y_2=32$, $y_3=18$, $y_4=40$ сумарні витрати енергії для маршруту $1\to3\to2\to4$ становлять $3\cdot64+196+3\cdot64=580$, що значно менше, ніж $484+196+484=1164$ (маршрут $1\to2\to3\to4$) і дещо менше, ніж $484+3\cdot64=3\cdot64+484=676$ (маршрути $1\to2\to4$ та $1\to3\to4$).

Наївна спроба змінити рівняння ДП на

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

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

То що робити?

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

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

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

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

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

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

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

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

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

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

енергії, стрибати назад якщо і можна, все одно невигідно.

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

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

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

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

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

1.2 Загальні умови застосовності і доцільності ДП

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

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

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

Про пункт 3 йшлося у розд. 1.1.5.

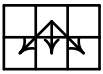
Пункт 4 називають *принципом оптимальності* або *принципом Белл-мана*. Часто са́ме він є найскладнішим з усіх пунктів. Простим прикладом його застосування є (2) з розд. 1.1.1 як обґрунтування (1), а для глибшого розуміння варто ознайомитися з обґрунтуванням (5) у розд. 1.3.1, а також міркуваннями з початку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

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

Пункт 6 зрозумілий і без пояснень.

1.3 Задача MaxSum

1.3.1 Базовий варіант задачі



Є прямокутна таблиця розміром N рядків на M стовпчиків. У кожній клітинці записано ціле число. Через неї потрібно пройти згори донизу, починаючи з будь-якої клітинки верхнього рядка, далі переходячи щоразу в одну з «нижньо-сусідніх» і закінчити маршрут у якій-небудь клітинці нижнього рядка. «Нижньо-сусідня» означає, що з клітинки (i, j) можна перейти у $(i+1, j-1)$, або у $(i+1, j)$, або у $(i+1, j+1)$ (див. також рис.), але не виходячи за межі таблиці (при $j=1$ перший з наведених варіантів стає неможливим, а при $j=M$ — останній).

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

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

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

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

Тривіальні підзадачі мають вигляд

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

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

Рівняння ДП:

$$S(i,j)=a(i,j)+\max\left(\begin{array}{l}S(i-1,j-1),\\ S(i-1,j),\\ S(i-1,j+1)\end{array}\right)\quad\begin{array}{l}\text{(пропускаючи варі-}\\ \text{анти, які виводять}\\ \text{за межі таблиці)}\end{array} \tag{5}$$

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

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

0	12	10	0	5	0	*12*	10	0	5
0	20	10	5	2	12	*32*	22	15	7
7	5	2	3	0	*39*	37	34	25	15
9	10	10	2	0	48	*49*	47	36	25

Цю задачу теж неправильно намагатися розв'язати простим жадібним алгоритмом. Зокрема, «ідея» *«вибрати максимальне число 1-го ряд-*

ка, далі щоразу переходити до максимального з нижньо-сусідніх» нездоланно помилкова. Не зважаючи на те, що вона знаходить правильні відповіді для досі розглянутих прикладів. Це (майже) випадково.

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

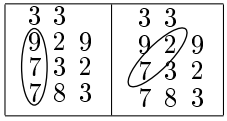
0	12	10	0	5	0	12	*10*	0	5
0	20	10	15	2	12	32	22	*25*	7
7	5	2	3	16	39	37	34	28	*41*
9	10	10	2	9	48	49	47	43	*50*

1.3.2 Найбільша серед непарних сум

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

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

Наївна спроба поставити серію підзадач *«Яку максимальну непарну суму $S_{odd}(i,j)$ можна набрати, пройшовши найкращий допустимий шлях до клітинки?»* ні до чого доброго не приводить. І важливо розібратися, *чому* ця спроба приречена на невдачу. Знайдемо (не через ДП, а через здоровий глузд) спочатку (лівий рисунок) шлях, яким досягається найкраща непарна сума до клітинки $(3,1)$, потім (правий) шлях з найкращою непарною сумою до клітинки $(2,1)$ тієї ж таблиці.



Бачимо, що найкращий шлях до $(3,1)$ проходить через $(2,1)$, *не* будучи продовженням найкращого шляху до $(2,1)$. Тобто, для підзадач *порушується принцип Беллмана* (пункт 4 розд. 1.2).

То що — задачу треба розв'язувати геть інакше, взагалі не динпрогом? Не факт. Поки що з'ясовано, що ДП не застосовне *до розглянутої серії підзадач*. Можливо (але поки що не гарантовано), що якась інша серія все ж дасть можливість застосувати ДП. У таких ситуаціях може ви-

явитися (а може і не виявитися) доцільним *розширити* серію підзадач, тобто ввести додатковий параметр (або кілька).

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

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

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

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

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

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

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

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

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

1.4 Розмін мінімальною кількістю банкнот

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

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

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

Спроба заявити «ніяка нормальна країна не вводила в обіг номінали 1, 17, 42» не вирішує проблему. І не лише тому́, що олімпіадні задачі частенько мають справу з не зовсім «нормальними» вхідними даними. В реальному світі спостерігалося (правда, досить давно; схоже, потім цю помилку виправили), як банкомат одного з великих українських банків, у якому були лише банкноти по 20 грн і 50 грн, на запит видати 60 грн відповідав «суму видати неможливо». При тому, що міг видати окремо 40 грн і зразу після цього 20 грн. Найімовірніше, причина якраз і була в тому, що старіша версія програмного забезпечення банкомата діяла за жадібним алгоритмом «щоб видати 60, почнемо з 50, лишається 10, які ніяк не видати по 20 — отже, неможливо».

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

Тривіальними будуть усі підзадачі при i=1:

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

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

Основне рівняння ДП має вигляд

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

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

Приклад заповненої таблички $Q(i,s)$ для S=20, x_1=4, x_2=5, x_3=7:

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

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

2 Література

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

3 Задачі першого дня (ДП)

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

Значна частина задач цього комплекту — класичні, авторство яких встановити вже важко. Такими є, зокрема, «базові» задачі 1A, 1D, 1H. Деякі з модифікацій цих задач розроблені укладачем комплекту І. Порубльовим — зокрема, задача 1C (де зміна вартості стрибків змінює задачу 1A). Ідея розширень серії підзадач загальновідома, але задачі 1F, 1G, де така потреба виникає природньо, розроблялися укладачем (у 1G використано також ідею Є. Поліщука).

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

Загальне формулювання умови цілком відповідає розд. 1.1.1.

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

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

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

Задача повністю розібрана у розд. 1.1.1.

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

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

Загальне формулювання умови та вимоги до формату виведення результатів цілком відповідають розд. 1.1.3. Формат вхідних даних в цілому відповідає попередній задачі 1A, але має інші обмеження: $2 \leqslant N \leqslant 100000$, значення висот платформ не перевищують за модулем 4000.

Задача повністю розібрана у розд. 1.1.1 та 1.1.3.

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

Загальне формулювання умови цілком відповідає розд. 1.1.5.

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

Вхідні дані. Перший рядок містить кількість платформ N ($2 \leqslant N \leqslant 100000$), дру́гий — N цілих чисел, значення яких не перевищують за модулем 4000 — висо́ти платформ.

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

Задача повністю розібрана у розд. 1.1.1 та 1.1.5.

Задача 1D «MaxSum (базова)»

Загальне формулювання умови цілком відповідає розд. 1.3.1.

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

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

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

Розбір задачі.

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

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

І може здатися логічним реалізувати це якось так:

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

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

Є щонайменше три способи вирішити цю проблему:

1. Написати `if`, який розгляне випадок $M=1$ окремо, а згаданий код буде лише для $M \geqslant 2$, де він працює правильно.

2. Зробити додаткові 0-й та $(M+1)$-й стовпчики, заповнивши їх "$-\infty$" (див. розд. 1.3.2 та розбір задачі 1F). Тоді `S[i,j] := a[i,j] + max3(S[i-1,j-1], S[i-1,j], S[i-1,j+1])` можна запускати однаково в усьому проміжку j від 1 до M.
3.
```
max := S[i-1, j];
if (j>1) and (S[i-1, j-1] > max) then
  max := S[i-1, j-1];
if (j<M) and (S[i-1, j+1] > max) then
  max := S[i-1, j+1];
S[i,j] := a[i,j] + max;
```

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

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

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

Задача 1E «MaxSum (з кількістю шляхів)»

Формулювання умови в цілому відповідає попередній задачі 1D, але, додатково до макс. суми, просять знайти *також кількість різних шляхів, на яких ця сума досягається*. Формат вхідних даних в точності повторює формат попередньої задачі 1D, але з уточненням: при перевірці будуть використані лише такі вхідні дані, для яких шукана кількість шляхів з макс. сумою не перевищує 10^9.

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

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

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

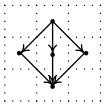
`a[2][2]=1`, `a[3][1]=10`, або `a[1][3]=100`, `a[2][3]=10`, `a[3][2]=1`, або `a[1][3]=100`, `a[2][3]=10`, `a[3][3]=1`.

Розбір задачі.

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

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

2	3	2
1	1	1
2	3	2



Введімо (додатково до таких сáмих, як у базовому варіанті задачі, масивів `a` та `S`) масив `K` таких сáмих розмірів, де $K(i, j)$ виражатиме *кількість шляхів до клітинки* (i, j)*, які дають суму* $S(i, j)$.

Аналогічно S, 1-й рядок K заповнюється тривіально:

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

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

Для подальших рядків, при знаходженні `S[i,j]` (згідно того сáмого рівняння (5)) слід класти у `K[i,j]` суму тих із `K[i-1,j-1]`, `K[i-1,j]`, `K[i-1,j+1]`, які у межах таблиці *та* відповідні яким значення $S(i-1, \cdot)$ максимальні (чи то єдиний максимум, чи один з однакових). Інакше кажучи, якщо максимум S досягався лише на одній з верхньо-сусідніх клітинок, значення K переноситься з відповідної клітинки; якщо відразу кілька верхньо-сусідніх клітинок мають однакове максимальне S — слід додати значення усіх відповідних K.

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

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

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

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

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

Задача 1F «MaxSum (непарна сума)»

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

Розбір задачі.

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

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

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

Задача 1G «MaxSum (усі стовпчики)»

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

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

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

Розбір задачі.

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

пчик, або в один із сусідніх, якщо побували і в першому, і в останньому, то й у всіх проміжних (отже, геть у всіх). Так що остáточною відповіддю задачі буде $\max_{1\leqslant j\leqslant M} S(N, j, \texttt{true}, \texttt{true})$.

Значна частина підзадач неможливі — зокрема, $(1, j, \texttt{false}, wr)$ при довільних j та wr (неможливо бути в 1-му стовпчику прямо зараз і при цьому не побувати в ньому) чи $(i, j, \texttt{true}, \texttt{true})$ при всіх j за умови $i < M$ (на кожному кроці відвідується лиш один стовпчик, і щоб відвідати всі M, треба хоча б M рядків). Тож відповіддю всіх таких підзадач вважатимемо "$-\infty$" (див. також розд. 1.3.2 та розбір задачі 1F).

Тривіальні підзадачі та рівняння ДП можна сформулювати, наприклад, так:[1]

- $S(i, 1, \texttt{false}, wr) = -\infty$ (для всіх $1 \leqslant i \leqslant N$, обох значень wr), бо неможливо бути в 1-му стовпчику зараз і не побувати в ньому;
- $S(i, M, wl, \texttt{false}) = -\infty$ (для всіх $1 \leqslant i \leqslant N$, обох значень wl), аналогічно щодо останнього;
- $S(1, 1, \texttt{true}, \texttt{false}) = a(1, 1)$, бо єдиний шлях складається з єдиної цієї клітинки, причому 1-го стовпчика, отже в ньому побували (а в останньому ні, бо $M \geqslant 2$);
- $S(1, M, \texttt{false}, \texttt{true}) = a(1, M)$, аналогічно щодо останнього;
- $S(1, j, \texttt{true}, \texttt{true}) = S(1, j, \texttt{true}, \texttt{false}) = S(1, j, \texttt{false}, \texttt{true}) = -\infty$ (для $2 \leqslant j \leqslant M-1$), бо коли шлях складається з єдиної клітинки, і вона з не 1-го й не останнього стовпчика, то в жодному з цих крайніх стовпчиків побувати неможливо;[2]
- $S(1, j, \texttt{false}, \texttt{false}) = a(1, j)$ (для $2 \leqslant j \leqslant M-1$), бо єдиний шлях складається з єдиної цієї клітинки, причому не 1-го й не останнього стовпчика;
- $S(i, 1, \texttt{true}, wr) = a(i, 1) + \max\big(S(i-1, 1, \texttt{true}, wr),\ S(i-1, 2, \texttt{true}, wr),\ S(i-1, 2, \texttt{false}, wr)\big)$ (для всіх $2 \leqslant i \leqslant N$, обох значень wr), бо: раз прямо зараз у 1-му стовпчику, то годяться і варі-

[1] У припущенні $M \geqslant 2$; відмінностей випадку $M \geqslant 2$ від випадку $M = 1$ більше, ніж у базовій задачі, тож випадок $M = 1$ пропонується розглянути окремо. Випадок же M=2, хоч і робить неможливою ситуацію $2 \leqslant j \leqslant M-1$, не потребу́є особливих формул (принаймні, поки користуємось "$-\infty$").

[2] Якщо порівняти цей випадок з останнім з наведених перед цим списком прикладів неможливих підзадач, можна побачити, що частина цього випадку частково охоплює той приклад, але не повністю. За рахунок такого роду моментів можливі дещо різні правильні формули, які кінець кінцем дають однакові результати.

ант, що там вже бували раніше, і варіант, що ні, в підсумку побували; варіант $S(i-1,1,\texttt{false},wr)$ все одно неможливий[3]; wr слід лишити, як було, бо від такого переходу відвідування останнього стовпчика не щезає (якщо було) і не з'являється (якщо ні);

- $S(i,M,wl,\texttt{true}) = a(i,M) + \max\bigl(S(i-1,M,wl,\texttt{true}), S(i-1, M-1, wl, \texttt{true}), S(i-1,M-1,wl,\texttt{false})\bigr)$ (для всіх $2\leqslant i\leqslant N$, обох значень wl), з аналогічних (симетричних) міркувань;
- $S(i,j,wl,wr) = a(i,j) + \max\bigl(S(i-1,j-1,wl,wr),\ S(i-1,\ j,\ wl,\ wr),\ S(i-1,j+1,wl,wr)\bigr)$ (для всіх $2\leqslant i\leqslant N$, всіх $2\leqslant j\leqslant M-1$, всіх поєднань всіх значень wl та wr) — основний, не пов'язаний з «крайовими ефектами», випадок, коли всі переходи з верхніх-сусідніх можливі й не змінюють wl чи wr.

Цей перелік, звісно, громіздкий. Але якщо його не зазубрювати, а відтворювати по смислу, розуміючи, то не так усе і складно.

Задача 1H «Банкомат–1»

Загальне формулювання умови цілком відповідає розд. 1.4.

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

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

Результати. Програма повинна вивести єдине число — знайдену мінімальну кількість банкнот. Якщо видати вказану суму вказаними банкнотами неможливо, програма повинна вивести рядок "`No solution`" (без лапок, перша літера велика, решта маленькі).

Примітка. У першому тесті, 72 можна видати трьома банкнотами 50, 20 і 2. У дру́гому, 60 можна видати трьома банкнотами як 20, 20 і 20.

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

[3] Хоча, якщо так зручніше, його можна теж розглянути, там все одно $(-\infty)$.

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

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

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

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

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

Розбір задачі.

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

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

тобто $Q(s)$ треба шукати циклом по i, беручи до уваги лише номінали $x_i \leqslant s$, щоб $s - x_i$ не ставало від'ємним. (Ситуація, коли для кожної підзадачі треба запускати вкладений цикл, нормальна для ДП; ненормально, що вона не виникала ні в одній з попередніх задач.) (Єдиною) тривіальною підзадачею можна вважати $Q(0) = 0$. Оскільки (в разі відсутності номіналу 1) можлива ситуація, коли якісь суми неможливо видати вказаними номіналами, для всіх $s > 0$ зручно ініціалізувати $Q(s)$ як "$+\infty$", а потім намагатися зменшити згідно (10).

Для зручності зворотнього ходу варто записувати у допоміжний масив, наприклад, значення того номінала, при якому вдалося отримати

оптимальну відповідь відповідної підзадачі. На те, щоб тримати такий допоміжний одновимірний масив, пам'яті цілком достатньо: два масиви по мільйону 4-байтових `int`-ів — менше 8 Мб. Пам'яті не вистачало, щоб розмістити N ($N \leqslant 100$) рядків.

Чому в розд. 1.4 не була розглянута відразу ця модифікація алгоритму? Відповідь на це питання — у наступній задачі.

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

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

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

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

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

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

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

Результати. Програма повинна вивести сім чисел — скільки треба видати банкнот 5 грн, 10 грн, 20 грн, 50 грн, 100 грн, 200 грн та 500 грн. Ці сім чисел треба вивести в один рядок, розділяючи пропусками. Сума цих чисел (загальна кількість банкнот до видачі) повинна бути мінімальною. Якщо видати суму, дотримуючись обмежень, неможливо, програма повинна замість відповіді вивести (єдине) число -1.

Розбір задачі.

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

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

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

Серію підзадач поставимо в точності як у задачі 1H: *«Якою мінімальною кількістю банкнот $Q(i,s)$ можна видати суму s, користуючись лише банкнотами 1-го, 2-го, …, i-го номіналів?»*. Кількості наявних банкнот m_1, m_2, …, m_7 використовуватимуться у рівнянні ДП та визначенні тривіальних підзадач, але вони не є параметрами підзадачі, бо (протягом обробки одних і тих самих вхідних даних) підзадачі ніколи не відрізняються одна́ від о́дної цими кількостями.

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

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

У поясненні до (8) (основного рівняння ДП задачі 1H) є фрагмент «... використовується ... $Q(i, s - x_i)$, а не $Q(i-1, s-x_i)$, тобто для суми $s - x_i$ знов можна або використати, або не використати поточний номінал». Са́ме це і треба змінити — наприклад, так:

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

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

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

Чи можна так само виражати $Q(i,s)$ через $Q(i-1,\ s-k\cdot x_i)$ (при $0 \leqslant$ $\leqslant k \cdot x_i \leqslant s$), а не через $Q(i,\ s-x_i)$ і в задачі 1H (першій у серії)? З точки зору формальної теоретичної правильності — можна, відповіді будуть такі самі. Але час робо́ти буде набагато гіршим, бо запускати зайвий цикл перебору можливих k, особливо при малих x_i — збільшувати найгірший можливий час роботи з $O(S \cdot N)$ до $O(S^2 \cdot N)$, що допустимо для обмежень поточної задачі 1J, але ніяк не задачі 1H.

Так що звідси доцільніше винести зовсім інше: якщо програма працює занадто довго, рівняння ДП чимось нагадує (12) і нема обмежень на кількості використань однакових значень — варто спробувати перетворити рівняння до вигляду, більш схожого на (8).

4 Задачі другого дня (коли ДП недоречне)

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

Частина задач комплекту (2A, 2B, 2C) спеціально розроблені так, щоб на перший погляд сильно нагадувати деякі задачі комплекту попереднього дня і тим провокувати застосування ДП. Решта задач теж мають *частину* властивостей, потрібних динпрогу, але кінець кінцем ДП виявляється або взагалі незастосовним, або недоречним. Авторами задачі 2E (розподіл станцій по зонам) є В. Челноков та Д. Поліщук.

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

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

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

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

Примітки. У першому тесті трапився такий збіг обставин, що саме для цього тесту, хоч і можна переходити в *будь-яку* клітинку наступного рядка (дозволені стрибки між 1-м і 3-м стовпчиками), це не дає можливості збільшити цільове значення у порівнянні з «базовим» варіантом задачі. У дру́гому ж тесті цю можливість варто використати й отримати 210 як $100 + 10 + 100$.

Розбір задачі.

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

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

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

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

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

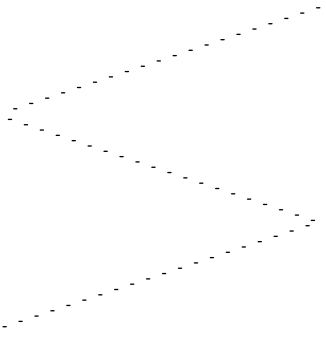
Результати. Виведіть єдине число — мінімальну кількість енергії, яку має витратити гравець на подолання платформ.

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

Вхідні дані	Результати
4 1 2 3 30	731

Розбір задачі.

Як і у розд. 1.1.5, тут можливі вхідні дані, для яких вигідно стрибати назад. Але, на відміну від розд. 1.1.5, іноді вигідно вертатися назад набагато.



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

Але в тому ж розд. 1.1.5 є рекомендація, що в таких випадках варто проаналізувати, чи застосовний алгоритм Дейкстри. І тут це так і є.

Причому, оскільки тепер граф щільний (навіть повний: з кожної платформи можна стрибати на будь-яку, питання лише в затратах), доречнішою є проста версія алгоритму Дейкстри, що працює за $O(N^2)$. Легко бачити, що при $N \leqslant 4000$ це цілком прийнятно.

Детальніше про алгоритм Дейкстри прочитайте деінде.

Задача 2C «MaxSum (щаслива)»

Є прямокутна таблиця розміром N рядків на M стовпчиків. У кожній клітинці записане невід'ємне ціле число. По ній потрібно пройти згори донизу, починаючи з будь-якої клітинки верхнього рядка, далі переходячи щоразу в одну з «нижньо-сусідніх» і закінчити маршрут у якій-небудь клітинці нижнього рядка. (Поняття «нижньо-сусідня» таке ж, як у задачах 1D, 1E, 1F та 1G попереднього дня.)

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

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

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

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

Примітки. Взагалі-то максимально можливою сумою є $27 = 10 + 7 + 10$, але число 27 не є щасливим. Тому відповіддю буде максимальна серед щасливих сума $7 = 3 + 0 + 4$, яка досягається уздовж маршруту `a[1][1]` → `a[2][2]` → `a[3][1]`.

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

Розбір задачі.

Для такої постановки задачі неясно, як можна було б розширювати серію, щоб добитися відрізнення щасливих сум від не щасливих. Але, якщо перечитати обмеження ($1 \leqslant N, M \leqslant 12$) та співвіднести його з оцінками кількості шляхів з задачі 1E попереднього дня, стає видно, що *тут можна перебрати всі шляхи* (їх менше $12 \cdot 3^{11} \approx 2$ млн, що для комп'ютера відносно небагато), для кожного порахувати суму, й вибрати максимальну звичайним («шкільним») алгоритмом вибору максимуму, з модифікованою умовою «замінювати поточний максимум, лише якщо нове число одночасно і більше, і щасливе».

Формат збірника не дозволяє пояснити все це справді детально (тим паче, що це не ДП, а одна з дуже багатьох згаданих тут альтернатив), тому наведемо код (C++, масив нумерується з 0) самої «серцевини» перебору без пояснень, а деталі додумайте, використовуючи інші джере́ла.

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

Задача 2D «Єгипетські дроби»

Математики стародавнього Єгипту не знали дробів у сучасному розумінні, але вміли подавати нецілі числа як суму дробів вигляду $1/k$, причому всі k в цій сумі мали бути різними. Наприклад, сучасне поняття $2/5$ виражалося як «одна третя та одна п'ятнадцята» (справді, $1/3+1/15=\frac{5}{15}+\frac{1}{15}=\frac{6}{15}=2/5$).

Математики Нового часу довели, що будь-який правильний звичайний дріб можна подати у єгипетському поданні (як суму дробів вигляду $1/k$), причому це подання не єдине. Напишіть програму, яка перетворюватиме правильний звичайний дріб до єгипетського подання із мінімальною кількістю доданків.

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

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

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

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

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

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

Розбір задачі.

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

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

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

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

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

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

того більшого дробу, з якого виділили «решту»...

Але є щонайменше три проблеми. (Найгірше, що істотні, заважають застосувати ДП просто і стандартно, але не нездоля́нні, можна придумувати засоби, щоб *пробувати* їх нівелювати.) Одна — як уникати повторів знаменників. Це можна робити розширенням серії підзадач, включивши до параметрів якусь інформацію про заборонені знаменники. Але яку? Ще одна — менші дроби часто мають більші знаменники, тож обмеження $n < m \leqslant 1000$ нічогісінько не каже про максимальний знаменник проміжних підзадач.[4] Третя проблема — складни́м виявляється (зазвичай елементарне) питання *«в якому діапазоні перебирати* `a`*?»* при виборі мінімума у рівнянні ДП. Вже пояснено, чому (для дробу $\frac{i}{j}$, при $i > 1$) нижня межа $(j \operatorname{div} i) + 1$. Але яка верхня?

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

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

Для суми діє комутативність (переставний закон), що набуває вигляду $\frac{1}{\mathtt{a}} + \frac{1}{\mathtt{b}} = \frac{1}{\mathtt{b}} + \frac{1}{\mathtt{a}}$; великі `a` означають, що $\frac{1}{\mathtt{a}}$ малі; отже, при великих `a`,

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

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

Не таку межу, якої хотілося, бо двійка взята з припущення, ніби доданків рівно два. Але краще така межа, яка вийшла, ніж ніякої. Тим паче, що вона піддається хоча б мінімальному узагальненню «якщо кількість доданків $\leqslant k$, то досить розглянути $\mathtt{a} < \frac{k\cdot\mathtt{m}}{\mathtt{n}}$». Адже, якою б не була кількість доданків, комутативність та асоціативність (переставний та сполучний закони) дозволяють[5] розглядати лише випадок, коли першим іде найбільший доданок (найменший знаменник).

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

Щоб чіткіше сформулювати, як ставити та розв'язувати таку обернену задачу, частково використаємо те, як раніше планували ставити серію підзадач ДП, але змінимо згідно з ідеями про бінпошук за відповіддю та про зростання знаменників. Зростання знаменників не лише потрі-

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

бне для обмеження $\mathtt{a} < \frac{k \cdot \mathtt{m}}{\mathtt{n}}$, а ще й дає простий спосіб уникати повторень знаменників (досить вимагати са́ме зростання, а не неспадання), і сильно зменшує перекривання підзадач.[6] Тож поставимо (i, j, k, q)-підзадачі *«або знайти спосіб розкласти дріб $\frac{i}{j}$ у єгипетське подання, де кількість доданків мусить бути $\leqslant k$, а значення знаменників $\geqslant q$, або з'ясувати, що це неможливо»*. Як вже згадувалося, ДП тут якщо й можливе, то лише у вигляді рекурсії з запам'ятовуванням у словникову структуру (`map`, ...); тепер, коли параметрів стало ще більше, твердження лише посилилося. Так що пишемо рекурсію, а чи включати в неї запам'ятовування, вирішимо пізніше. Наявність чи відсутність запам'ятовувань формально відносить алгоритм до різних класів (ДП і перебір), але тексти програм відрізнятимуться несильно, причому виключно в парі-трійці місць, а не розпорошено по всьому коду, тож можна просто спробувати обидва варіанти й вибрати кращий.

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

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

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

Вибравши потрібні частини з усіх раніших пояснень, легко бачити, що `a` слід перебирати в межах від $\max\big((\mathtt{denom} \operatorname{div} \mathtt{nom}) + 1,\ \mathtt{minNextDenom}\big)$

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

до $(\texttt{maxDenomAmount} \cdot \texttt{denom} - 1) \operatorname{div} \texttt{nom}$ включно. (Щодо верхньої межі́ природніми є питання «чому включно?» і «звідки "$\ldots - 1$"?», але насправді са́ме це і виражає $\texttt{a} < \frac{k \cdot j}{i}$: у випадку, коли $(k \cdot j)$ не кратне i, само́ цілочисельне ділення (div) дає менший результат, ніж дробове, причому "$\ldots - 1$" не впливає на результат div; якщо ж кратне, то "$\ldots - 1$" забезпечує перехід до меншої цілої частки.)

Діапазон `a`, заданий цими формулами, може виявлятися порожнім. Але це непогано: замість запускати рекурсію, швиденько закінчимо, а ніякі ро́зв'язки при цьому не будуть втрачені, бо всі ме́жі тим і обґрунтовані, що за ними або взагалі нема розв'язків, або такі самі ро́зв'язки з іншим порядком доданків повинні бути знайдені в інших гілках рекурсії. Це не повинно бути проблемою, слід лише переконатися, що код написаний так, що цикл з 0 ітераціями працює нормально.

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

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

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

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

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

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

То кінець кінцем вийшов динпрог чи перебір? Окупають себе запам'ятовування чи ні? Експеримент показав, що якщо вважати, що підзадача повторюється лише при повторенні всіх чотирьох (i, j, k, q), то ні. Гіпотетично можна аналізувати підзадачі якось розумніше (включаючи міркування «при цих (i, j, k, q) не буде розкладення, бо вже

відомо, що його нема при тих самих i, j, q і більшому k», але не обмежуючись ними); тоді можна очікувати більших відтинань при меншій кількості запам'ятованих підзадач. Але для розглянутої задачі при $1 \leqslant \leqslant n < m \leqslant 1000$ досить рекурсії без запам'ятовувань.

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

Задача 2E «Розподіл станцій по зонам»

Керівництво Дуже Великої Залізниці (ДВЗ) вирішило встановити нову систему оплати за проїзд. ДВЗ являє собою відрізок прямої, на якій послідовно розміщені N станцій. Планується розбити їх на M неперервних зон, що слідують підряд, таким чином, щоб кожна зона містила хоча б одну станцію. Оплату проїзду від станції j до станції k необхідно встановити рівною $1 + |z_j - z_k|$, де z_j і z_k — номери зон, яким належать станції j та k відповідно. Відома кількість пасажирів, які відправляються за день з кожної станції на кожну іншу.

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

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

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

Вхідні дані	Результати
3 2 200 10 20	440

Примітки. Іншими словами, рядки вхідних даних з 2-го по N-й являють собою нижню-ліву половину симетричної матриці пасажиропотоку з нулями по головній діагоналі (де симетрично розміщені елементи не треба до-

давати, бо кожен з них — вже сума потоків туди й назад).

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

Розбір задачі.

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

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

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

1-й (лівіший) з прикладів *є* складовою 2-го, бо у 2-му пасажиропотоки між станціями 1–4 такі самі, лише з'являється нова 5-та станція. При цьому в 1-му прикладі пасажиропотік між станціями 1 та 2 перевищує всі інші, тож межу зон варто провести між ними; у 2-му прикладі потоки між 3 та 5 і 4 та 5 перевищують всі інші, тож обидві межі варто провести там (одну між 3 та 4, іншу між 4 та 5). Тобто, 5-а станція *вплинула* на те, як оптимально розбити на 2 зони проміжок з 1-ої по 4-ту станції — навіть при тому, що в оптимальному розв'язку $C(5,3)$ одна з меж якраз після 4-ої станції. Отже, для цієї серії підзадач принцип Беллмана не виконується.

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

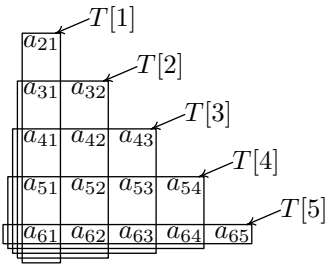
Так що перейдемо до правильного розв'язку, основаного *не* на динамічному програмуванні. Він є уточненням вжитого раніше приблизного міркування «Пасажиропотік між такими-то станціями настільки великий, що межу між зонами явно варто провести між цими станціями».

Абстрагувавшись від механізму продажу квитків, можна помітити, що сумарна виручка згідно правила «проїзд між зонами z_j та z_k коштує $1+|z_j-z_k|$» рівно така ж, яка була б, якби: (1) на кожній станції зби-

рали по одиничці грошей з усіх, хто заходить у поїзд; (2) при кожному перетині межí між зонами (не на станції, а між ними) збирали ще по одиничці грошей з усіх пасажирів, які перебувають у поїзді. Хто перетинає кілька меж — платить за кожен перетин у момент цього перетину. (Тут *не* пропонується міняти систему оплати (цього не можна, задачу треба розв'язати для вказаної системи), а аналізуються особливості формули $1+|z_j-z_k|$ і встановлюється її *теоретична рівносильність* описаній системі. А потім усе це використовується, щоб побудувати й обґрунтувати ефективний розв'язок.)

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

Отже, для розв'язку задачі достатньо спочатку додати всі задані у вхідних даних пасажиропотоки (для всіх пасажирів є складова ціни «1+...»), потім додати $M-1$ максимальних серед значень $T[1]$, $T[2]$, ..., $T[N{-}1]$ (наприклад, для цього можна відсортувати масив, що містить значення T, але можна і якось інакше).



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

При бажанні можна сказати, що цей 2-й спосіб має дещо спільне з ДП, а отже, ДП у задачі все ж є. Але одне, що автор тексту дотримується (не загальноприйнятої, але поширеної) точки зору, що це слід називати не динпрогом, а рекурентними комбінаторними співвідношеннями; інше — навіть якщо це динпрог, він відіграє допоміжну роль і геть не схожий на природну спробу, згадану на початку.

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

Задача 2F «Ендшпіль»

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

Королю заборонено ходити у клітинки, які перебувають під боєм будь-якої чужої фігури. Якщо один з гравців зробив такий хід, що король суперника опинився під боєм (це називають «шах»), суперник зобов'язаний відповісти таким ходом, щоб його король вже не був під боєм чужої фігури. Якщо такого ходу не існує, то це називають «мат».

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

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

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

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

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

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

Розбір задачі.

Можливо, існують (а може, й ні) ефективніші розв'язки, які спираються на знання теорії шахів. Але ми розглянемо підхід, що не потребує вузькоспеціалізованих знань. До того ж, методи, традиційні для шахових програм, і складні́, й використовують спільно з деревом ходів евристичні (отже, не завжди точні) оцінки позицій, тож нема впевненості, чи знайдуть вони са́ме мінімальну кількість ходів.

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

Закодуємо позиції 19-бітовими числами: один біт — чий хід (чорних чи білих), решта $3 \cdot 2 \cdot 3 = 18$ — координати на дошці кожної з трьох фігур (бо три фігури, а розмір шахівниці становить два виміри по 8 клітинок, де якраз $\log_2 8 = 3$). Таких кодів, включаючи і коди неможливих позицій, 2^{19} (приблизно півмільйона), що для комп'ютера відносно небагато.

Розсортуємо їх за трьома групами: мати (хід чорних, король чорних під боєм, ходити нікуди), неможливі (більше однієї фігури в одній клітинці; королі на сусідніх клітинках; хід білих, а чорний король під боєм білого ферзя) і «решта позицій» (їхні статуси поки що невідомі, для більшості пізніше будуть уточнені).

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

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

Подальші дії (знаючи $2k$-півходові позиції, будуємо $(2k+1)$- та $(2k+2)$-півходові) цілком аналогічні. Процес обривається, коли чергових ...-півходових позицій вже не з'являється (експериментально, 20-півходові ще є, а 21-півходових вже нема).

Умовою кінця циклу не можна ставити «перелік "решти" позицій став порожнім», бо порожнім він так і не стає. Зокрема, за рахунок патових позицій, де *пат* — позиція, в якій повинен ходити гравець, жодна фігура якого не може нікуди походити (не порушуючи правил гри), але, на відміну від мату, король цього гравця не під боєм. За сучасними шаховими правилами, пат є не програшем, а нічиєю. В умові задачі

це не описано, але, трактуючи її суто формально, отримуємо цілком узгоджений висновок, що такі позиції не є метою білих. Чорні ж, у рамках цієї задачі, не мають можливості приводити гру в такі позиції.

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

Щоб швидко визначати, якою є позиція (неможливою, матом, 1-півходовою, тощо) варто підтримувати масив з діапазоном індексів від 0 до $2^{19}-1$, значення якого якось кодуватимуть ситуації «неможлива», «мат», «1-півходова», «2-півходова», ..., «ще не відомо».

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

Варто також зазначити, що розглянутий підхід має більш-менш стандартну назву «*ретроаналіз*», і може бути застосований до значної частини ігор двох гравців (якщо, звісно, прийнятні оцінки $\Theta(P)$ пам'яті, $\Theta(P \cdot D \cdot M)$ часу). Водночас, термін «...-півходова позиція» не є загальноприйнятим.

Зміст