Зміст

1 Теоретичний матеріал

> Динамическое программирование — это когда у нас есть одна большая задача, которую непонятно как решать, и мы разбиваем её на меньшие задачи, которые тоже непонятно как решать.
>
> Аким Кумок

Жартівливе «означення» з епіграфа стає майже правильним, якщо замінити слова́ *«которые тоже непонятно как решать»* на *«які можна вирішити 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 словосполучення «динамічне програмування») можна розв'язати далеко не всі задачі, й, дивлячись на задачу, не завжди легко зрозуміти, чи можливо і чи варто вирішувати її са́ме динпрогом. Більш того: навіть якщо звідкись відомо, що задача «на динпрог», все одно це задає лише загальний напрям розробки алгоритму, а більшість подробиць треба придумувати для кожної задачі по-своєму. Тому́ для вивчення ДП особливо потрібно знайомитися як із теорією, так і з різними прикладами застосування.

За все це ДП дуже люблять на олімпіадах (на змаганнях високого рівня майже завжди є задачі на динпрог). 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 нуль) разів.

Відомі висо́ти платформ у порядку зліва направо. Знайдіть мінімальну кількість енергії, достатню, щоб дістатися з 1-ї платформи до N-ї (останньої).

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

Рішення 1-ї і 2-ї підзадач *тривіальні*:

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

Для подальших платформ, справедливо

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

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

Розберемо докладніше, звідки взялося це рівняння:

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

Тепер 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 або реалізувати її самостійно, або переписати вибір меншого через `if`.)

1.1.2 Ітеративна та рекурсивна реалізації ДП

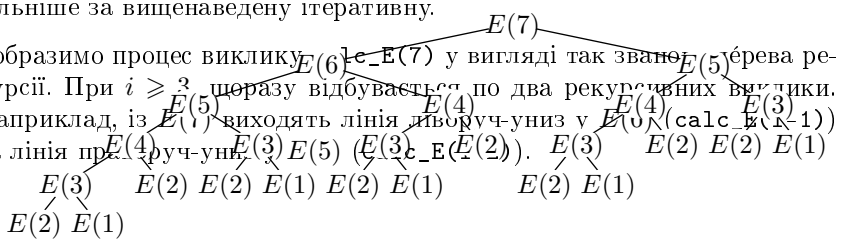
Вищенаведений підхід (спочатку заповнити розв'язки тривіальних підзадач, потім у циклі застосовувати рівняння ДП, постійно спираючись на вже готові відповіді й отримуючи нові, доки не дійде до відповіді остаточної задачі) називають *ітеративною* (або *ітераційною*, що те са́ме) реалізацією. Можливий і альтернативний підхід — реалізувати рівняння ДП рекурсивною функцією. Він точно нічим не кращий для даної конкретної задачі про платформи, але може виявлятися зручнішим для деяких інших задач.

На перший погляд, мало б вийти щось приблизно таке:

```
int calc_E(int i) {
  if (i == 1) return 0;
  if (i == 2) return E[2] = 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]));
}
```

Але насправді *це дуже погана реалізація*, яка працюватиме *значно* повільніше за вищенаведену ітеративну.

Зобразимо процес виклику `calc_E(7)` у вигляді так званого дерева 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 значення $E(5)$ (який потребує сумарно дев'ять викликів рекурсивної функції), тричі — $E(4)$ (п'ять викликів), і т. д. Внаслідок цього, при зростанні кількості платформ 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	$\cdots$

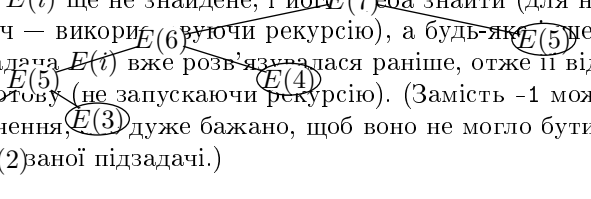
Тим не менш, релізація ДП через рекурсію можлива — якщо це рекурсія *із запам'ятовуваннями* (*memoized* recursion):

```
int calc_E(int i) {
  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]),
             calc_E(i-2) + 3*abs(y[i]-y[i-1]));
  return E[i];
}
```

Відповідно, виклик цієї функції має бути приблизно таким:

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

Тут `calc_E(N)` — очевидний виклик рекурсії; `fill_n` — функція (з бібліотеки `algorithm`), яка заповнює вказаний масив `E` вказаними значеннями `-1`. Таким чином, `-1` у i-му елементі масива виражає, що відповідне значення $E(i)$ ще не знайдене, і його треба знайти (для нетривіальних підзадач — використовуючи рекурсію), а будь-яке інше значення — що підзадача $E(i)$ вже розв'язувалася раніше, отже її відповідь можна брати готову (не запускаючи рекурсію). (Замість `-1` може бути якесь інше значення, але дуже бажано, щоб воно не могло бути відповіддю в розв'язаної підзадачі.)



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

Звичайно, тут є багато моментів, які можна реалізувати інакше. Можна дотриматися традицій C-подібних мов програмування, змістивши нумерацію з `1..N` на `0..N-1`; можна інакше реалізувати ініціалізацію масиву позначками «ще не вирішено», і т. д. То все деталі. Важливо лише, щоб запам'ятовування було, і ним користувалися.

Чи буде рекурсія із запам'ятовуваннями швидша за ітеративну реалізацію? Як правило — ні, вона все одно трохи повільніша. Наскільки? Чіткої відповіді не існує, бо це сильно залежить і від особливостей конкретної задачі, і від особливостей архітектури комп'ютера, на якому виконується програма. Дуже приблизно і дуже в середньому можна говорити про сповільнення у 1,2–2 рази.

Навіщо тоді взагалі рекурсія із запам'ятовуваннями? У даній задачі вона і не потрібна. А в інших задачах... По-перше, бувають настільки заплутані серії підзадач, що важко розібратись, у якому порядку їх ви-

рішувати, щоб завжди спиратися виключно на вже розв'язані. Рекурсія із запам'ятовуваннями дозволяє не думати над цим, під час виконання само собою вийде, що для вже розв'язаних підзадач будуть взяті готові відповіді, а ще не розв'язані будуть розв'язані рекурсивно. По-друге, іноді бувають задачі, при рекурсивному вирішенні яких можна бачити, що деякі з підзадач насправді не потрібні, і взагалі не розв'язувати їх, а при ітеративній реалізації цього не видно.

1.1.3 Зворотній хід (відновлення розгорнутої відповіді)

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

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

Виведіть спочатку (у 1-му рядку) мінімальну кількість енергії, а потім шлях, що забезпечує цей мінімум: у 2-му рядку — кількість платформ, у 3-му — послідовність номерів платформ. Якщо можливі різні шляхи з однаковими мінімальними витратами — виведіть будь-який один.

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

Щоб забезпечити відновлення 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.

У даній задачі про платформи це означає: крім масива `y` зі вхідними даними (висотами платформ) і масива `E` зі знайденими 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 оцих мінімальних затрат енергії $E(i)$?»*.

На етапі розробки рівняння ДП на папері / в голові все лишається як було, а от під час остато́чної програмної реалізації доводиться відмовитися від зручного запису через виклик функції `min` і перейти до явного розгалуження.

```
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 додатково ще й `prev[2] = 1`.

Коли масиви вже заповнені, робиться сам *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 шлях.

	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 на етапі зворотнього ходу, а на етапі вибору мінімума. Якщо поміняти "`if E1 < E2`" на "`if E1 <= E2`", жодне значення масива `E` не зміниться, а у масиві `prev` відбудеться зразу кілька змін; провівши той самий зворотній хід по зміненому масиву `prev`, отримаємо іншу конкретну послідовність.

1.1.4 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* (у зворотньому напрямку). Наприклад, при y_1=10, y_2=32, y_3=18, y_4=40 сумарні витрати енергії для маршруту $1 \to 3 \to 2 \to 4$ становлять $3{\times}64 + 196 + 3{\times}64 = 580$, що значно менше, ніж $484 + 196 + 484 = 1164$ (маршрут $1 \to 2 \to 3 \to 4$) і дещо менше, чим $3 \times 64 + 484 = 676$ (маршрут $1 \to 3 \to 4$).

Наївна спроба змінити рівняння ДП на

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

не призводить до негайного успіху. Це в якомусь смислі правильно, але тут є циклічні залежності (наприклад, $E(2)$ залежить від $E(3)$, а $E(3)$ — від $E(2)$). А таке не запрограмуєш ні циклом, ні рекурсією (байдуже, з запам'ятовуваннями чи без).

Що робити?

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

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

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

Плануючи застовувати алгоритм Дейкстри до цієї задачі, слід помітити, що граф тут дуже розріджений (при N вершинах всього $2N{-}3$ неорієнтованих ребер), тому на час виконання дуже сильно впливатиме, чи робити вибір чергової поточної вершини шляхом перебору вершин (як спочатку і пропонував сам Дейкстра), чи застосувати пізніше розроблені оптимізації, головним чином `priority_queue` (вона ж піраміда) або іншу структуру даних, що дозволяє швидко знаходити мінімум. Детальніше про це можна прочитати, зокрема, за адресою `e-maxx.ru/algo/dijkstra_sparse`.

Втім, алгоритм Дейкстри не є зараз предметом розгляду, так що перейдемо до варіанту Б і подивимося, що ж там за такі особливі властивості. У даній конкретній 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 подібні 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 наведеним спостереженням щодо даної 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 = витратам для $(i-3) \to (i-1) \to (i-2) \to i$, тобто $3 \cdot \big(y(i-1) - y(i-3)\big)^2 + \big(y(i-2) - y(i-1)\big)^2 + 3 \cdot \big(y(i) - y(i-2)\big)^2$.

Тепер уже можна написати програму, в якій просто громіздкіше рівняння ДП (мінімум вибирається не з двох, а з трьох варіантів), і є «напівтривіальна» підзадача $E(3)$, де відбувається вибір кращого з двох варіантів. До речі, така програма ще й працюватиме трохи швидше, ніж оптимізований алгоритм Дейкстри ($\theta(N)$ проти $O(N \log N)$).

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

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

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

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

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

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 якраз і 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, про них уже йшлося на самому початку розд. 1.1 і ще йтиметься в абсолютно всіх подальших задачах.

Про пункт 3 йшлося у розд. 1.1.4.

Пункт 4 називають *принципом оптимальності* або *принципом Белл-мана*. Часто са́ме його аналіз і є найскладнішим з усіх пунктів. Для кращого розуміння його суті варто ознайомитися з обґрунтуванням формули (5) наприкінці розд. 1.3.1, а також міркуваннями з початку розд. 1.3.2 та початку аналізу задачі H, де розглядаються ситуації, коли вимоги даного пункту порушені.

Пункт 5 слід розуміти так, що для ДП типова ситуація, подібна до описаної у розд. 1.1.2 потреби проводити запам'ятовування у разі ре-

курсивної реалізації. Цей пункт 5 — єдиний, порушення якого в принципі не забороняє застосувати ДП. Але його порушення дає підстави сумніватися, *чи варто* його використовувати. Якщо багатократного використання одних і тих самих результатів не буває — навіщо, власне, щось запам'ятовувати? Чи не краще реалізувати чи то просту рекурсію (без запам'ятовувань), чи то якийсь однопрохідний жадібний алгоритм, чи ще якусь альтернативу?

Пункт 6 зрозумілий і без пояснень; як приклад ситуації, коли він насилу дотриманий, на грані порушення, можна навести задачу з розд. 1.4 (та її модифікацію — задачу J), а також задачу G.

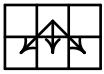
1.3 Задача MaxSum

1.3.1 Базовий варіант задачі

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

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

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



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

Цільова задача в цьому випадку *не* є однією із підзадач серії, але легко отримується з розв'язків серії, як $\max_{1 \leqslant j \leqslant M} S(N, j)$. Це правильно, бо якщо $S(N, 1)$ — максимально можлива сума серед усіх шляхів, що закінчуються 1-ою клітинкою останнього рядка, $S(N, 2)$ — серед усіх, що закінчуються 2-ою, і т. д., а шуканий шлях закінчується в одній із цих клітинок, то вибрати максимум серед усіх $S(N, j)$ $(1 \leqslant j \leqslant M)$ якраз і буде максимумом серед взагалі усіх 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 із вхідних даних. Це правильно, бо шлях, що задовольняє правилам і закінчується у клітинці верхнього ряда, завжди існує і єдиний, і складається з самої лише цієї клітинки.

Рівняння ДП:

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

Це правильно, бо:

1. Сума уздовж будь-якого шляху, що закінчується клітинкою (i, j) (при $i{>}1$), дорівнює «сумі усього шляху крім останньої клітинки» плюс «сама ця клітинка»; отже, для отримання максимуму по всьому шляху (включаючи клітинку (i, j)) не може бути вигідно приходити у передостанню клітинку не оптимальним шляхом — краще прийти оптимальним, загальна сума від того покращиться. Іншими словами, достатньо розглядати лише оптимальні ро́зв'язки підзадач минулого рядка. А це якраз і є суттєвою складовою принципу Беллмана (пункт 4 з розд. 1.2).
2. Будь-який шлях до клітинки (i, j) обов'язково приходить або з клітинки $(i{-}1, j{-}1)$, або з $(i{-}1, j)$, або з $(i{-}1, j{+}1)$ (пропускаючи варіанти, які виводять за межі таблиці); тому достатньо вибрати максимум лише з цих трьох (або менше) величин.

Наведемо також приклад. У лівій табличці записані вхідні дані (вони ж $a(\cdot, \cdot)$, вони

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

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

ж значення самих клітинок). У правій — побудовані для цих вхідних даних відповіді на підзадачі (вони ж $S(\cdot,\cdot)$).

Цю задачу *неправильно* намагатися вирішити за допомогою простого жадібного алгоритму. Зокрема, «алгоритм» вигляду *«вибрати максимальне число 1-го рядка, далі щоразу переходити до максимального з нижньо-сусідніх»* — *неправильний*. Не зважаючи навіть на те, що він знаходить правильні відповіді для обох досі розглянутих прикладів. Це (майже) випадковість — такий «алгоритм» дає правильні відповіді для деяких вхідних даних, але далеко не всіх можливих.

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*

Цей приклад відрізняється від попереднього трьома числами: $a(2,4)$ стало 15 замість 5, $a(3,5)$ стало 16 замість 0, $a(4,5)$ стало 9 замість 0. Максимальні значення по всій таблиці, по верхньому та по нижньому рядкам ніяк не мінялися, але відповідь змінилася, причому шлях став зовсім інший.

Так що у подібних задачах не можна хапатися за неперевірену ідею лише тому, що вона дає правильну відповідь на приклад з умови. І доведення алгоритму, аналогічні викладеним вище, досить важливі, хоч на олімпіадах з інформатики їх і не питають.

1.3.2 Найбільша серед непарних сум

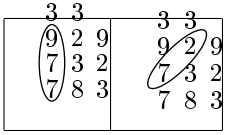
Є така сама прямокутна таблиця, по ній треба так само перейти згори донизу, так само переходячи щоразу у нижньо-сусідню.

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

Тільки тепер питають максимальну ***серед непарних*** *сум. Непарною має бути са́ме* ***сума, а не окремі доданки***. *Якщо абсолютно всі суми парні — вивести "*`impossible`*". «Максимальною взагалі» сумою для прикладу з рисунку є $42 = 15 + 9 + 9 + 9$, але число 42 парне. Тому відповіддю буде максимальна серед непарних сума $39 = 15 + 9 + 9 + 6$.*

Наївна спроба поставити серію підзадач *«Яку максимальну непарну суму $S_{odd}(i,j)$ можна набрати, пройшовши найкращий допустимий шлях*

до даної клітинки?» ні до чого доброго не приводить.



Важливо розібратися, *чому* ця спроба приречена на невдачу. Знайдемо (не через ДП, а через здоровий глузд і ручний перебір) спочатку (лівий рисунок) шлях, яким досягається найкраща непарна сума до клітинки $(3,1)$, потім (правий рисунок) — шлях з найкращою непарною сумою до клітинки $(2,1)$ такої ж таблиці.

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

То що — задачу треба розв'язувати геть інакше і взагалі не динпрогом? Не факт. Поки що з'ясовано лише, 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 (не виконується принцип Беллмана) може виявитися (а може і не виявитися) доцільним *розширити* серію підзадач, тобто ввести додатковий параметр (або кілька).

Конкретно зáраз серію підзадач варто переформулювати так: *«Для кожної (i,j)-ої клітинки знайти дві величини: максимальну непарну суму і максимальну парну суму, які можна набрати, прийшовши до неї.»*. Що технічно може бути реалізовано або як двовимірний масив, елементами якого є не числа, а пари чисел, або ж як підзадача з трьома параметрами $S(i,j,p)$ (і, відповідно, *три*вимірний масив), у якому по двом вимірам 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 версії задачі, у 1-му рядку єдиний шлях складається лише з одного елемента. А якщо елемент єдиний і парний, ним ніяк не можна набрати непарну суму, так само як і непарним парну. Оскільки

суму треба *максимізувати*, зручно виражати неможливість са́ме *мінус* нескінче́нністю: при порівнянні $-\infty$ з будь-яким реальним значенням, буде вибране са́ме реальне значення.

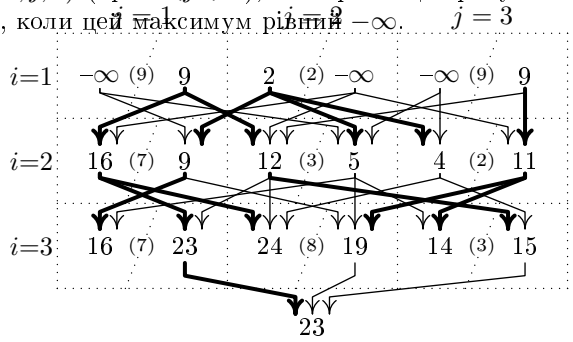
Основне рівняння ДП теж громіздке, але його подробиці неважко відновити самостійно, спираючись на все вищесказане. Відзначимо лише, що воно виражає $S(i,j,p)$ через $a(i,j)$ та:

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 $S(i{-}1,j{-}1,1{-}p)$, $S(i{-}1,j,1{-}p)$, $S(i{-}1,j{+}1,1{-}p)$.

(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 $-\infty$.



Приклад процесу вирішення задачі. У кожній клітинці посере́дині в дужках записане значення самої клітинки, ліворуч — максимальна парна сума, праворуч — максимальна непарна. Стрілки зображають, від яких попередніх підзадач залежить дана, а жирним виділено, яка з них дає максимум.

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 цю задачу правильно розв'язувати жадібно, по принципу «щоразу брати банкноту якнайбільшого можливого номінала». Наприклад, якщо номінали 1, 2, 5, 10, 20, 50, 100, 200, 500 і видати треба суму 2014 — значить, банкнота 500 (ли-

шається 1514), банкнота 500 (лишається 1014), банкнота 500 (лишається 514), банкнота 500 (лишається 14), банкнота 10 (лишається 4), банкнота 2 (лишається 2), банкнота 2 (видано).

Відзначимо без доведення, що *за умови* набору номіналів «1, 2, 5, 10, 20, 50, 100, 200, 500» цей жадібний алгоритм таки є правильним (дає гарантовано мінімальну кількість банкнот для будь-якої суми). Але це може бути й не так для інших систем номіналів!

Наприклад, якщо є три номінали 1, 17, 42 і треба видати суму 51, то за жадібним алгоритмом вийде, що треба узяти 42 і потім донабирати решту 9 по одиничці (всього 10 банкнот) — що неоптимально, бо можна взяти три банкноти по 17 (всього 3 банкноти).

Спроба заявити «ніяка нормальна країна не вводила в обіг банкноти номіналами 1, 17, 42» не вирішує проблему. Наприклад, неоднократно спостерігалося (правда, лише у минулі роки; схоже, потім цю помилку нарешті виправили), як банкомат одного з великих українських банків, в якому були в наявності 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 окремо 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 діяти, щоб 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?»*. Мається на увазі, що номінали банкнот попередньо виписані у якомусь порядку (*не* обов'язково порядок зростання, спадання чи ще якийсь осмислений; важливо лише, щоб було зрозуміло, який номінал 1-й, який 2-й і т. д.). При цьому, «користуючись лише банкнотами 1-го, 2-го, ..., i-го номіналів» не вимагає задіювати усі ці номінали, важливо лише не використовувати інші.

Тривіальними будуть усі підзадачі при i=1:

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

(можна користуватися лише банкнотами 1-го виду; отже, якщо сума кратна номіналу — її видати можна, причому кількість банкнот дорів-

нює сумі, поділеній на номінал; якщо не кратна — видати, користуючись лише 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 набрати ще суму $s-x_i$. Можливість узяти кілька (більше однієї) 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 особливо наголосили «*якщо* сума S, яку треба видати, не дуже велика». Це ж типова (а зовсім не унікальна) ситуація, коли чим більші вхідні дані, тим більші витрати на виконання програми... Навіщо наголошувати?

Справа у тім, що тут розмір таблички пропорційний *кількості* номіналів (що типово) і *значенню* суми S (чого у даному наборі задач ще не було). Хто знає, що таке поліноміальні та експоненційні асимптотичні оцінки — зверніть увагу, що це називається *псевдополіноміальною* асимптотичною оцінкою. А по-простому суть цієї особливості можна виразити так: з одного боку, розмір таблички всього лишень пропорційний S, з іншого — якщо, наприклад, задачу розв'язують (для однакових номіналів) один раз для суми 123, а інший — для суми 1234567, збільшення розміру вхідних даних всього на чотири символи призводить до збільшення таблички у приблизно 10 тисяч 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. `https://www.youtube.com/watch?v=iKj-xI4enLw`
4. `http://e-maxx.ru/algo`
5. Порублёв И. Н., Ставровский А. Б., «Алгоритмы и программы. Решение олимпиадных задач», М.–СПБ–К., Диалектика, 2007 — глава 13 «Динамическое программирование».

3 Задачі

Даний комплект задач доступний для on-line перевірки на сайті `http://ejudge.ckipo.edu.ua/`, змагання № 44.

Значна частина задач даного комплекту — класичні, справжнє авторство яких встановити вже важко. Такими є і задача A (платфомибазова), і задача D (MaxSum-базова), і задача I (банкомат-базова). Деякі з істотних модифікацій цих задач розроблені укладачем даного комплекту І. Порубльовим — зокрема, задача C (аналіз того, як зміна вартості стрибків може змінити задачу A). Ідея розширень серії підзадач давно відома, але послідовність задач D, F, G, де така потреба виникає просто і природньо, розроблялася укладачем. Авторами задачі H (розподіл станцій по зонам) є В. Челноков та Д. Поліщук.

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

Ви можете згадати хоч одного свого знайомого до двадцятирічного віку, який у дитинстві не грав у комп'ютерні ігри? Якщо так, то може ви й самі не знайомі з цією розвагою? Втім, це не повинно створити труднощів при вирішенні цієї задачі.

У багатьох старих іграх з двовимірною графікою можна зіткнутися

з такою ситуацією. Який-небудь герой стрибає по платформах (або острівцям), які висять у повітрі. Він повинен перебратися від одного краю екрану до іншого. При цьому при стрибку з платформи на сусідню, герой витрачає $|y_2 - y_1|$ одиниць енергії, де y_1 і y_2 — висóти, на яких розташовані ці платформи. Крім того, у героя є суперприйом, який дозволяє перескочити через платформу, але на це витрачається $3 \cdot |y_3 - y_1|$ одиниць енергії. Звичайно ж, енергію слід витрачати максимально економно.

Нехай вам відомі координати всіх платформ у порядку від лівого краю до правого. Чи зможете ви знайти, яка мінімальна кількість енергії необхідна герою, щоб дістатися з першої платформи до останньої?

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.

Результати

Виведіть єдине число — мінімальну кількість енергії, яку має витратити гравець на подолання платформ (звісно ж у припущенні, що cheat-коди використовувати не можна).

Приклади

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
3 1 5 10	9
3 1 5 2	3

Задача повністю розібрана у розд. 1.1.1.

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

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. При стрибку з платформи на сусідню, герой витрачає $|y_2 - y_1|$ енергії, де y_1 і y_2 — висóти, на яких розташовані ці платформи. Крім того, є суперприйом, що дозволяє перескочити через платформу, але на це витрачається $3 \cdot |y_3 - y_1|$ енергії. (Суперприйом *можна* застосовувати багатократно.)

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) і список (послідовність) платформ, через які потрібно пройти.

Вхідні дані

Перший рядок містить кількість платформ N ($2 \leqslant N \leqslant 100000$), дрýгий — N цілих чисел, значення яких не перевищують по модулю 4000 — висóти платформ.

Результати

У першому рядку виведіть мінімальну кількість енергії. У другому — кількість платформ, через які потрібно пройти, а у третьому виведіть послідовність цих платформ.

Приклади

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
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

Примітка

Зрозуміло, в цій задачі для деяких вхідних даних можливі різні правильні послідовності платформ. Ваша програма повинна виводити будь-яку одну з них.

Задача повністю розібрана у розд. 1.1.1 та 1.1.3.

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

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. При стрибку з платформи на сусідню, герой витрачає $(y_2 - y_1)^2$ енергії, де y_1 і y_2 — висóти, на яких розташовані ці платформи. Крім того, є суперприйом, що дозволяє перескочити через платформу, але на це витрачається $3 \cdot (y_3 - y_1)^2$ енергії. (Суперприйом *можна* застосовувати багатократно.)

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 N ($2 \leqslant N \leqslant 100000$), дрýгий — N цілих чисел, значення яких не перевищують по модулю 4000 — висóти платформ.

Результати

У єдиному рядку виведіть єдине число — мінімальну кількість енергії.

Приклади

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
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

Задача повністю розібрана у розд. 1.1.1 та 1.1.4.

Задача D. «MaxSum (базова)»

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

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

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 по модулю 10^6) — значення клітинок таблиці.

Результати

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

Приклад

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	42

Розбір

Суть розв'язку розібрана у розд. 1.3.1.

А тут розглянемо дві типові для розв'язків даної задачі помилки.

Помилка перша В умові написано «$(i{+}1, j{-}1)$, або у $(i{+}1, j)$, або у $(i{+}1, j{+}1)$ ... (при $j{=}1$ перший з наведених варіантів стає неможливим, а при $j{=}M$ — останній)». А у розборі 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)».

І може здатися логічним реалізовувати 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 N-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` — власна функція, яка знаходить максимальне з двох, `max3` — власна функція, яка знаходить максимальне з трьох). Начебто все чудово — окремо розібралися з 1-м елементом (для якого нема лівого сусіда), окремо з останнім (у якого нема правого), окремо з рештою у котрих є обидва сусіди... Але тут приходить біда не від великих вхідних даних, а від малих. Точніше — від (дозволених згідно з умовою задачі!) вхідних даних з багатьма рядками, але 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 наступної задачі). Тоді `S[i,j] := a[i,j] + max3(S[i-1,j-1], S[i-1,j], S[i-1,j+1])` можна запускати абсолютно однотипно 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.
Цей код можна справити заміною усіх “>” на “>=”.

Але можливі й інші варіації на тему цієї помилки. Так що є великий сумнів, чи варто взагалі шукати максимум з трьох подібним чином.

Задача E. «MaxSum (з кількістю шляхів)»

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

Напишіть програму, яка знаходитиме максимально можливу суму значень пройдених клітинок серед усіх допустимих шляхів, *а також кількість різних шляхів, на яких ця сума досягається.*

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 по модулю 10^6) — значення клітинок таблиці.

Гарантовано, що при перевірці будуть використані тільки такі вхідні дані, для яких шукана кількість шляхів з максимальною сумою не перевищує 10^9 (мільярд).

Результати

Вивести в одному рядку два цілі числа, розділені пробілом: максималь-

но можливу суму за маршрутами зазначеного вигляду та кількість різних маршрутів, уздовж яких досягається дана максимальна сума.

Приклади

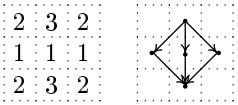
Клавіатура (стандартний вхід)	Екран (стандартний вихід)
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	42 1
3 3 1 1 100 1 1 10 10 1 1	111 3

Примітка

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

Розбір

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 (приклади з умови знов підібрані підступно). На лівому рисунку наведено вхідні дані. На правому виділені всі шляхи з однаковою максимальною сумою 7. Їх три, і вони сходяться в одну клітинку. 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 для самóго лише останнього рядка (чи кількох останніх).

Введемо (додатково до таких сáмих як у базовій версії задачі масивів `a`

та `S`) масив `K` таких са́мих розмірів, де $K(i,j)$ означатиме *кількість шляхів до клітинки* (i,j)*, які дають суму в точності* $S(i,j)$.

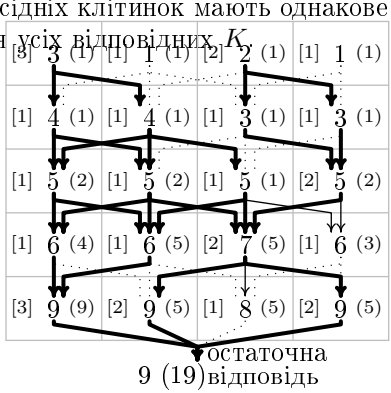
Аналогічно S, 1-й рядок K заповнюється тривіально, а подальші — на основі попередніх. Конкретніше,

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

бо там шлях до кожної клітинки складається з самої лише цієї клітинки, отже завжди рівно один, а його сума неминуче максимальна серед оцього одного, що веде до даної клітинки.

Для подальших рядків, негайно після знаходження `S[i,j]` (згідно того са́мого рівняння ДП (5)) слід покласти у `K[i,j]` суму тих із `K[i-1,j-1]`, `K[i-1,j]`, `K[i-1,j+1]`, які у межах $1 \leqslant j \leqslant M$ *та* відповідні яким значення $S(i-1,\cdot)$ виявилися максимальними (чи то єдиним максимумом, чи то одним із однакових).

Інакше кажучи, якщо максимум S досягався лише на одній з верхньо-сусідніх клітинок, значення K переноситься з цієї клітинки; якщо відразу 2 з 2, чи 2 з 3, чи 3 з 3 верхньо-сусідніх клітинок мають однакове максимальне S — слід додати значення усіх відповідних K.



Приклад. У кожній клітинці, ліворуч у квадратних дужках — число цієї клітинки зі вхідних даних $a(i,j)$; по центру — максимальна сума $S(i,j)$; праворуч у круглих дужках — кількість $T(i,j)$ шляхів з макссумою.

Пунктиром зображено ситуації, коли верхньо-сусідня клітинка не дає максимума S (у інших верхньо-сусідніх S більша), стрі́лками — коли дає; тоненькі стрі́лки означають, що цей шлях десь потім обривається, не даючи остато́чно максимальної суми в останньому рядку.

Тобто, коли у клітинку входить одна стрілка — лише одна з верхньо-сусідніх клітинок ма́ла максимальну S, тож кількість шляхів перенесена звідти; коли кілька стрілок — однаковий максимум у кількох клітинках, кількості шляхів додаються.

Наводити код не будемо, реалізуйте все це самостійно.

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

В умові недаремно гарантовано, що шукана кількість шляхів з максимальною сумою не перевищує 10^9. Взагалі-то їх може бути *набагато* більше. Наприклад, на рисунку зображено кількості шляхів (без сум), для виродженого випадку вхідних даних «$N{=}M{=}10$, усі клітинки мають однакові значення». Засобами комбінаторики можна оцінити, що при $M{>}1$ та всіх однакових числах сумарна кількість шляхів десь у проміжку між $M \cdot 2^N$ та $M \cdot 3^N$. Так що без додаткової гарантії «до 10^9» треба було б писати «довгу арифметику».

До речі, гарантія стосується лише відповіді, а значення `K[i,j]` доводиться шукати в т. ч. й для тих клітинок, які не належать жодному шляху зі справді максимальною сумою. Так що переповнення типу (будь-якого зі стандартних цілочисельних) все одно можливі. При більшості можливих налаштувань більшості компіляторів це ніяк не вплине на відповідь (бо ті чи́сла не вибираються). Але якщо увімкнений контроль переповнень — це може призводити до аварійних завершень програми.

А ще така велика кількість можливих шляхів зайвий раз показує, що задачу не варто намагатися робити повним перебором всіх можливих шляхів. Сонце погасне раніше, ніж він завершиться.

Задача F. «MaxSum (непарна сума)»

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

Напишіть програму, яка знаходитиме максимально можливу *непарну* суму значень пройдених клітинок серед усіх допустимих шляхів. Зверніть увагу, що непарною повинна бути са́ме сума, а обмежень щодо парності чи непарності окремих доданків нема.

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 по модулю 10^6) — значення клітинок таблиці.

Результати

Вивести або єдине ціле число (знайдену максимальну серед непарних сум за маршрутами зазначеного вигляду), або рядок "`impossible`" (без лапок, маленькими латинськими буквами). Рядок "`impossible`" має виводитися тільки у разі, коли абсолютно всі допустимі маршрути мають парні суми.

Приклад

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	39

Примітка

Взагалі-то максимально можлива сума — $42 = 15 + 9 + 9 + 9$, але число 42 парне. Тому відповіддю буде максимальна серед непарних сума $39 = 15 + 9 + 9 + 6$, яка досягається уздовж маршруту `a[1][2]` $\to$ $\to$ `a[2][1]` $\to$ `a[3][1]` $\to$ `a[4][1]`.

Розбір

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

Один з них — взяти досить велике за модулем від'ємне число. Але яке са́ме — треба вибирати більш-менш акуратно. Взяти його сильно малим за модулем — може виявитися, що така "$-\infty$" плюс відразу багато додатних чисел вийде сума, яка вже трактуватиметься як нормальне число, а не "$-\infty$". Взяти сильно великим за модулем — така "$-\infty$" плюс відразу багато від'ємних чисел може вийти за межі типу. Акуратно треба і з

перевіркою «максимум серед $S(N, j, 1)$ (при $1 \leqslant j \leqslant M$) рівний $-\infty$» — на відміну від чистої математики, така "$-\infty$" плюс конкретне скінченне ненульове число вже не "$-\infty$".

Враховуючи все вищесказане, 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 "$S \neq -\infty$" як "`S>-5e8`". Але не можна брати в якості "$-\infty$" ні -10^8, ні $-2 \cdot 10^9$, бо можливі вищезгадані проблеми.

Ще один спосіб — поганий тим, що не кросплатформенний, тобто може працювати в одних компіляторах і не працювати в інших. (Тут слід чітко розуміти, що у практичному програмуванні важливо, чи працюватиме програма на комп'ютерах замовника, а на олімпіаді важливо, чи працюватиме програма на сервері перевірки задач. А працює чи не працює програма у самого програміста — мало кого цікавить...) Але, *якщо* цей спосіб працює, то дуже легкий: 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 де треба використовувати `MINUS_INFTY`.

Задача G. «MaxSum (відвідати усі стовпчики ходами коня)»

Є прямокутна таблиця розміром N рядків на M стовпчиків. У кожній клітинці записано ціле число. По ній потрібно пройти згори донизу, починаючи з будь-якої клітинки верхнього рядка, далі рухаючись вниз ходами коня і закінчити маршрут у якій-небудь клітинці нижнього рядка. Тобто, з клітинки (i, j) можна перейти у $(i+1, j-2)$, або у $(i+2, j-1)$, або у $(i+2, j+1)$, або у $(i+1, j+2)$, виключаючи варіанти, що виходять за межі таблиці.

Напишіть програму, яка знаходитиме максимально можливу суму значень пройдених клітинок *серед усіх допустимих шляхів ходами коня, що проходять хоча б по одному разу через кожен зі стовпчиків.*

Вхідні дані

У першому рядку записані N і M — кількість рядків і кількість стовпчиків ($1 \leqslant N \leqslant 42$, $1 \leqslant M \leqslant 17$); далі у кожному з наступних N рядків записано рівно по M розділених пробілами цілих чисел (кожне не перевищує по модулю 10^6) — значення клітинок таблиці.

Результати

Вивести або єдине ціле число (знайдену максимальну серед сум за маршрутами зазначеного вигляду), або рядок "`impossible`" (без лапок, маленькими латинськими буквами). Рядок "`impossible`" має виводитися тільки у разі, коли не існує жодного маршруту ходами коня, що проходить через всі стовпчики хоча б по одному разу.

Приклади

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	25
3 3 1 15 2 9 7 5 9 2 4	impossible

Примітка

Для поля 4×3 є рівно чотири способи спуститися ходами коня, відвідавши кожен стовпчик:

перший	a[1][1]→a[2][3]→a[4][2];
другий	a[1][2]→a[3][1]→a[4][3];
третій	a[1][2]→a[3][3]→a[4][1];
четвертий	a[1][3]→a[2][1]→a[4][2]

Максимальна можлива сума $25 = 15 + 4 + 6$ досягається на 3-му з них.

Для поля 3×3 таких способів взагалі немає.

Розбір

Якби задача відрізнялася від попередніх *лише* вимогою рухатися ходами коня — відмінність від базової версії задачі була б не принципова. Перетворити 2–3 `if`-и на 4 трохи громіздкіші `if`-и потребує акуратності, але не так уже й складно.

Набагато суттєвішою є вимога відвідати усі стовпчики. Ставити серію підзадач так, щоб у абсолютно кожній підзадачі була присутня вимога відвідати усі стовпчики, не можна — при такій серії і порушуватиметься принцип Беллмана, і навіть не вийде виділити тривіальні підзадачі.

Знаючи розд. 1.3.2, неважко здогадатися, що тут мабуть теж потрібно розширити серію підзадач. Але той додатковий параметр тепер буде значно складнішої природи: «*Яку максимальну суму $S(i, j, X)$ можна назбирати, дійшовши дозволеними ходами від верха таблиці до клітинки (i, j) і відвідавши в точності стовпчики з множини X?*».

Що взагалі таке множина́? Це математичне поняття, яке розглядає набір (сукупність, зібрання) елементів як єдине ціле. Взагалі кажучи, множи́ни можуть мати елементи абсолютно будь-якої природи, але *у даній задачі* елементами будуть лише чи́сла (номери стовпчиків).

Елемент або належить множи́ні, або ні (не може належати наполовину, двічі належати, і т. д.). Множина́ не розрізняє порядок елементів (чи спочатку 1, потім 3, потім 2, чи спочатку 3, потім 2, потім 1 — вийде одна й та ж множина $\{1, 2, 3\}$). Множина не зобов'язана містити усі підряд чи́сла проміжку; наприклад, може бути $\{1, 2, 4, 6, 7, 9\}$.

Вертаючись до поставленої серії підзадач, тривіальними будуть $S(1, 1, \{1\}) = a(1, 1)$, $S(1, 2, \{2\}) = a(1, 2)$, …, $S(1, M, \{M\}) = a(1, M)$ (де $\{1\}$, $\{2\}$, … — множи́ни, кожна з яких містить лише один елемент). Остато́чна відповідь шукатиметься як $\max_{1 \leqslant j \leqslant M} S(N, j, \{1, 2, \ldots, M\})$ (де множина містить усі чи́сла від 1 до M).

Деталі рівняння ДП пропонуємо розробити самостійно. А на реалізації оцих множин трохи зупинимося. Якщо наведене пояснення виявиться не досить детальне — шукайте інші джерела, за назвою «динамічне програмування по підмножинам» (рос. «по подмножествам»).

Перш за все — *ці* множи́ни *не* варто подавати у типі `set` (C++ STL) чи якомусь подібному. Стандартний для таких задач спосіб — самому ко-

дувати множи́ни у так звані *бітові маски* (bitmasks), а потім ставитися до них (бітмасок) як до чисел і використовувати як індекси масиву.

Сама назва «бітмаска» виражає, що множину можна подавати, ставлячи одинички у тих бітах, для яких відповідні елементи належать множи́ні, і нулі у решті. Нехай $M{=}10$. Тоді, наприклад, множина $\{2,3,5\}$ буде подана як 0000010110 (одинички у 2-му, 3-му і 5-му з кінця бітах). А, наприклад, $\{1,2,4,6,7,9\}$ — як 0101101011; $\{10\}$ — як 1000000000. Враховуючи $0000010110_2 = 22_{Dec}$, $0101101011_2 = 363_{Dec}$ та $1000000000_2 = 512_{Dec}$, це означає, що множина $\{2,3,5\}$ відповідатиме 22-му, $\{1,2,4,6,7,9\}$ — 363-му, і $\{10\}$ — 512-му елементу.

Причому, відповідності між двійковими на десятковими числами наведені виключно щоб показати, як це бітмаска може бути індексом; технічно ніякого «переведення між двійковою та десятковою системами числення» не треба, а треба робити дії безпосередньо у двійковій ситемі (всере́дині комп'ютера все одно вона!).

Операція, яка у Паскалі позначається як "`n shl k`", а у мовах C, C++, C♯, Java, Python — як "`n << k`", зміщує усі біти двійкового запису числа́ `n` ліворуч, дописуючи з правого боку `k` нулів.

Значить, має місце зображена праворуч тотожність (зліва від "<<" одиниця; справа — ціле невід'ємне значення, менше за кількість бітів у типі `int`). Тобто, бітмаска будь-якої одноелементної множини́ $\{j\}$ може бути записана виразом `1<<(j-1)` (якщо нумерація чисел-елементів починається з 1) або `1<<j` (якщо нумерація з 0). Значить, у цій задачі особливо варто нумерувати стовпчики не «1, 2, ..., M», а «0, 1, ..., $M{-}1$».

`1<<k` $= 00\ldots01\underbrace{00\ldots0}_{k}{}_2$

Розглянемо кілька стандартних дій при роботі з бітмасками множин.

Побітові операції Загальновідомо, що `and` (`&&`) та `or` (`||`) — логічні операції, причому `and` набуває значення `true` лише коли обидва аргументи `true`, а `or` — коли хоча б один з аргументів.

У цих операцій є побітові варіанти, які застосовуються не до логічних значень, а до чисел (лише цілих). Смисл цих операцій — застосувати ту саму дію `and` або `or` *до кожного біта окремо.*

$00101010{=}42_{Dec}$	$00101010{=}42_{Dec}$
$\&\,00010110{=}22_{Dec}$	$\mid\,00010110{=}22_{Dec}$
$\overline{00000010{=}\ 2_{Dec}}$	$\overline{00111110{=}62_{Dec}}$

Мовою Pascal побітові `and` та `or` пишуться так само, як логічні, С-подібними мовами пишуть одинарні значки `&` та `|`, а не подвійні.

Існує також і побітова версія унарної (одноелементної) операції `not`. Для неї теж у Паскалі позначення таке саме, як для логічної (`not`), а у С-подібних мовах побітова позначається `~` (логічна — `!`).

Перевірка належності елемента множині Є множина, задана бітмаскою `SET_CODE` і елемент `k` (нумерація з 0). Потрібно перевірити, чи належить цей елемент цій множині.

Pascal	`if (SET_CODE and (1 shl k)) <> 0 then`
C, C++, C♯, Java	`if((SET_CODE & (1<<k)) != 0)`

`1<<k` створює число, в якому одиниця *лише* у біті, відповідному елементу `k`. Якщо `SET_CODE` теж містить одиницю у цьому біті, результат виходить ненульовий (конкретніше, рівний `1<<k`, але у наведеному фрагменті це не використовується), інакше виходить 0.

Мовами C і C++ (але не C♯ і не Java) часто пропускають частину "`!=0`", тобто пишуть `if(SET_CODE&(1<<k))`. У C/C++ і так будь-яке ненульове значення означає `true`, а 0 означає `false`. Чи варто так скорочувати — питання філософське, бо з одного боку воно коротше, з іншого — використання в `if`-і чисел замість логічних значень може заплутати.

Вставка (додавання, приєднання) елемента у множину Є множина, задана бітмаскою `SET_CODE` і елемент `k` (нумерація з 0). Потрібно створити бітмаску нової множини, якій належать і всі ті елементи, що належали `SET_CODE`, і елемент `k`.

Pascal	`SET_CODE or (1 shl k)`
C, C++, C♯, Java	`SET_CODE | (1<<k)`

Цей вираз працює як у випадку, коли множина `SET_CODE` не містить елемента `k`, так і коли містить (вставка вже існуючого елементу не міняє множину).

Вилучення елемента з множини Є множина, задана бітмаскою `SET_CODE` і елемент `k` (нумерація з 0). Потрібно створити бітмаску нової множини, де із множини `SET_CODE` вилучений елемент `k`.

	Працює завжди	Вимагає k∈SET_CODE
Pascal	`SET_CODE and not(1 shl k)`	`SET_CODE xor (1 shl k)`
C,C++,C♯,Java	`SET_CODE & ~(1<<k)`	`SET_CODE ^ (1<<k)`

Створення множини $\{0,1,\ldots,M{-}1\}$ Згадаємо, що остáточна відповідь шукається сáме як $\max_{0\leqslant j<M} S(N,j,\{0,1,\ldots,M{-}1\})$ (при нумерації з 0), бо це і є «шляхи, які проходять хоча б по одному разу через усі стовпчики». Щось аналогічне є й у багатьох інших задачах, які варто розв'язувати динпрогом по підмножинам.

Pascal	`(1 shl M)-1`	$00\ldots01\underbrace{00\ldots0}_{k}{}_2-1 = 00\ldots00\underbrace{11\ldots1}_{k}{}_2$
C, C++, C♯, Java	`(1<<M)-1`	

У цій задачі недаремно досить малі обмеження на розміри ($1\leqslant N\leqslant 42$, $1\leqslant M\leqslant 17$ проти $1\leqslant N,M\leqslant 200$ у простіших версіях задачі MaxSum).

Алгоритми, рекомендовані до усіх попередніх задач, потребували $\theta(N{\cdot}M)$ пам'яті і часу, так що $200{\times}200 = 4{\cdot}10^4$ було досить мало (навіть якщо треба кілька масивів, навіть якщо робити занадто широкі типи `int64` або `double` там, де можна обійтися `int`-ом — загальний об'єм пам'яті не перевищував 1 мегабайт; час поміщався у соті секунди, і при цьому витрачався в основному на читання вхідних даних).

У цій задачі все це не так. Раз множина $\{0\}$ кодується числом 1, а $\{0,1,\ldots,M{-}1\}$ — числом $2^M{-}1$, то при $M{=}17$ результати підзадач для усіх можливих підмножин навіть для однієї клітинки (конкретного рядка і конкретного стовпчика) займатимуть 4×2^{17} байтів (де 4 — розмір у байтах одного `int`-а), що являє собою півмегабайта.

Так що доводиться ще й викручуватися, щоб вміститися у обмеження пам'яті: при вирішенні підзадач i-го рядка потрібні лише результати $(i{-}1)$-го та $(i{-}2)$-го рядків, а ще старіші не потрібні; можна, наприклад, мати не 42 рядки, а три, з 0-го по 2-й, і скрізь замість `i`-го рядка звертатися до `(i%3)`-го (де `%`, він же `mod` — залишок від ділення). Або, наприклад, динамічно виділяти пам'ять під дані того рядка, до якого щойно дістались, звільняючи пам'ять від тих рядків, які стали гарантовано не потрібні. Само собою, будь-який варіант цього прийому можливий лише завдяки тому, що питають саму лише числову відповідь (не треба відновлювати шлях зворотнім ходом).

Час роботи (при N=42, M=17) теж виявляється порядка секунд, а зовсім не сотих секунди.

Все ж навіть такий громіздкий динпрог ефективніший (за часом), ніж повний перебір. Побудувавши і застосувавши рекурентне комбінаторне співвідношення (з урахуванням задачі «MaxSum (з кількістю шляхів)» це під силу і вам), можна побачити, що загальна кількість шляхів згори донизу ходами коня при N=42, M=17 перевищує 10^{18}. А загальна кількість дій у ДП по підмножинам менша 10^9. Щоправда, 10^{18} — кількість усіх шляхів (не лише тих, що проходять через усі стовпчики), а перебір може намагатися відкидати не потрібні; але справді ефективно відкидати зайве у переборі — важче, ніж реалізувати ДП по підмножинам.

Задача H. «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, яку максимальну денну виручку можна отримати за новою системою при оптимальному розбитті на зони.

Вхідні дані

Перший рядок містить два цілих числá N та M ($1\leqslant M\leqslant N\leqslant 1000$).

Другий — одне число, що означає кількість пасажирів, які їдуть між станціями 1 та 2.
Третій — два числá, що означають кількість пасажирів, які їдуть між станціями 1 та 3 та між станціями 2 та 3, відповідно.
І так далі. В N-ому рядку міститься $N-1$ число, i-е з них визначає кількість пасажирів 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.

Приклад

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
3 2 200 10 20	440

Примітка

Іншими словами, рядки вхідних даних з 2-го по N-й являють собою нижню-ліву половину симетричної матриці пасажиропотоку з нулями по головній діагоналі. Зокрема, у прикладі по суті задано матрицю

$$\begin{pmatrix} 0 & 200 & 10 \\ 200 & 0 & 20 \\ 10 & 20 & 0 \end{pmatrix}$$

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

Дана задача включена у цей набір головним чином як приклад ситуації, коли за багатьма ознаками здається, ніби задачу доцільно вирішувати динпрогом, а насправді її варто вирішити зовсім інакше.

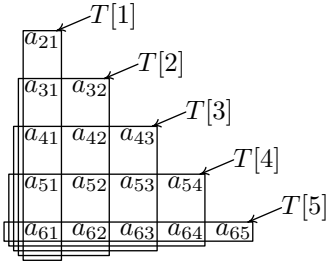
Абстрагувавшись від механізму продажу квитків, можна помітити, що сумарна виручка згідно правила «проїзд між зонами j та k коштує $1 + |z_j - z_k|$» рівно така ж, яка була б, якби гроші збирали так: (1) на кожній станції збирають по одиничці грошей з усіх, хто заходить у по-

їзд; (2) при кожному перетині межі́ між зонами (причому не на станції, а між ними) збирають ще по одиничці грошей з усіх пасажирів, котрі перебувають у поїзді. Хто перетинає кілька меж — платить за кожен перетин окремо і са́ме у момент цього перетину.

(Тут *не* пропонується справді міняти систему оплати, а лише аналізуються особливості формули $1+|z_j-z_k|$, встановлюється її *теоретична* еквівалентність описаній системі, й потім усе це використовується, щоб побудувати і 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 і вдасться збільшити виручку, якщо са́ме між i-ю та $(i+1)$-ю станціями і провести межу́ 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 «$1+\ldots$»), 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. Експеримент теж показує, що (при великих N) 2-й спосіб працює набагато швидше.

Задача I. «Банкомат–1»

У деякій державі в обігу перебувають банкноти певних номіналів. Національний банк хоче, щоб банкомат видавав будь-яку запитану суму за допомогою мінімального числа банкнот, вважаючи, що запас банкнот кожного номіналу необмежений. Допоможіть Національному банку вирішити цю задачу.

Вхідні дані

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

Результати

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

Приклади

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
7 1 2 5 10 20 50 100 72	3
2 20 50 60	3

Примітка

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

Задачу можна розв'язати, реалізувавши алгоритм з розд. 1.4. Можна відштовхуватися і від його модифікації, потрібної для наступної задачі.

Задача J. «Банкомат–2 (з відновленням)»

У деякій державі в обігу перебувають банкноти певних номіналів. Національний банк хоче, щоб банкомат видавав будь-яку запитану суму за допомогою мінімального числа банкнот, вважаючи, що запас банкнот кожного номіналу необмежений. Допоможіть Національному банку вирішити цю задачу.

Вхідні дані

Перший рядок містить натуральне число N, що не перевищує 100 — кількість номіналів банкнот у обігу. Другий рядок вхідних даних містить N різних натуральних чисел x_1, x_2,..., x_N, що не перевищують 10^6 — номінали банкнот. Третій рядок містить натуральне число S, що не перевищує 10^6 — суму, яку необхідно видати.

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.

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).

Приклади

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
7 1 2 5 10 20 50 100 72	50 20 2
2 20 50 60	20 20 20

Розбір

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 вирішити 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. Власне 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 виходило у від'ємні чи́сла. (Ситуація, коли для вирішення кожної підзадачі треба запускати цикл, нормальна для ДП; ненормально, що вона не виникала ні в одній з попередніх задач.)

(Єдиною) тривіальною підзадачею можна вважати $Q(0) = 0$. Оскільки тут теж *можлива* ситуація, коли якісь суми неможливо видати вказаними номіналами, для всіх $s > 0$ зручно ініціалізувати $Q(s)$ як "$+\infty$", а потім намагатися замінити на менші значення згідно (10).

Для зручності відновлення (зворотнього ходу) варто записувати у допоміжний масив, наприклад, величину 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 K. «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 мінімально можливою 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 числа вхідних даних є цілими, перебувають у межах від 0 включно до 5000 включно.

Результати

Програма повинна вивести на пристрій стандартного виведення (екран) сім чисел — скільки треба видати банкнот по 5 грн, по 10 грн, по 20 грн, по 50 грн, по 100 грн, по 200 грн та по 500 грн. Ці сім чисел треба вивести в один рядок, розділяючи пропусками. Сума цих чисел (загальна кількість банкнот до видачі) повинна бути мінімально можливою.

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

Приклади

Клавіатура (стандартний вхід)	Екран (стандартний вихід)
0 100 1 100 0 0 0 190	0 2 1 3 0 0 0
5000 2000 5000 2000 5000 2000 500 17	-1

Розбір

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 I (серія підзадач якої є, по суті, розширенням серії підзадач задачі J) 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 I: *«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, тобто (протягом обробки одних і тих самих вхідних даних) підзадачі ніколи не відрізняються одна́ від о́дної цими кількостями.

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

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

У поясненні до основного рівняння ДП (8) 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:

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

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

Само собою, треба 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 I (першій у серії)? З точки зору формальної теоретичної правильності — можна, відповіді будуть такі самі. Але час робо́ти буде набагато гіршим, бо запускати оцей зайвий цикл перебору можливих k, особливо при малих x_i — збільшувати найгірший можливий час роботи з $O(S \cdot N)$ до, по суті, $O(S^2 \cdot N)$, що для обмежень вхідних даних задачі I недопустимо.

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).

Разом з тим, при обмеженнях поточної задачі K, оцінка $O(S^2{\cdot}N)$ цілком прийнятна.