ЗАДАЧІ XXIV ВСЕУКРАЇНСЬКОЇ ОЛІМПІАДИ З ІНФОРМАТИКИ ТА РЕКОМЕНДАЦІЇ ЩОДО ЇХ РОЗВ'ЯЗУВАННЯ

Бондаренко В.В., Коротков А.С., Нейтер Д.Ю., Ягіяєв Ш.І.

ЗАВДАННЯ ПЕРШОГО ТУРУ

1. Plus та Хог (Д. Нейтер)

Побітове виключне АБО (або побітове додавання за модулем два) — це бінарна операція, дія якої еквівалентна застосуванню логічного виключного АБО до кожної пари бітів, які стоять на однакових позиціях у двійковому записі операндів. Іншими словами, якщо відповідні біти операндів різні, то відповідний двійковий розряд результату дорівнює 1; якщо ж біти збігаються, то двійковий розряд результату дорівнює 0.

Наприклад, якщо $X=101=01100101_2$,

$Y=41=00101001_2$;

то X xor $Y=76=01001100_2$.

У мові програмування Fascal побітове виключне або позначається «xor», а в C/C++ — символом «^».

Завдання. Напишіть програму plusxor, яка за двома цілими невід'ємними числами A та B знайде такі невід'ємні цілі числа X і Y, для яких виконуються умови:

1. $A=X+Y$;
2. $B=X$xor$Y$, де xor — побітове виключне або;
3. X — найменше серед чисел, для яких виконуються умови 1 і 2.

Вхідні дані. Перші два рядки вхідного файла plusxor. dat містять відповідно цілі числа A та B ($0\le A, B\le 2^{64}-1$).

Вихідні дані. Єдиний рядок вихідного файла plusxor.sol має містити два цілих невід'ємних числа X і Y, або єдине число — 1 , якщо таких пар не існує.

Оцінювання. Щонайменше у 50% тестів виконуються додаткові обмеження $0\le A, B\le 2^{20}-1$. Щонайменше у 70% тестів виконуються додаткові обмеження $0\le A, B=2^{31}-1$.

Приклад вхідних і вихідних даних

plusxor.dat
142 76

plusxor.sol
33 109

Рекомендації щодо розв'язування

Спочатку розв'яжемо задачу в припущенні, що шукана пара чисел X, Y існує. Будемо позначати двійкові цифри в записі чисел так: $X=(x_{63}x_{62}\ \dots\ x_1x_0)_2$, $Y=(y_{63}y_{62}\ \dots\ y_1y_0)_2$, $A=(a_{63}a_{62}\ \dots\ a_1a_0)_2$, $B=(b_{63}b_{62}\dots b_1b_0)_2$.

З означення побітового «виключного або» випливає, що $b_i=x_i+y_i \pmod 2$ (1). З іншого боку, $a_i=x_i+y_i+c_i-1 \pmod 2$ (2), де c_i позначає перенос з i-го двійкового розряду, який обчислюється за такою формулою: $c_i=[(x_i+y_i+c_i-1)/2]$, де через [ ] позначається операція взяття цілої частини. Для зручності позначимо $c_{-1}=0$.

З (1) і (2) випливає $(a_i+b_i) \bmod 2=[(x_i+y_i) \bmod 2+(x_i+y_i+c_{i-1}) \bmod 2] \bmod 2 = (2x_i+2y_i+c_{i-1}) \bmod 2 = [(2x_i) \bmod 2+(2y_i) \bmod 2+c_{i-1} \bmod 2] \bmod 2=0+0+c_{i-1}$. Отже, $c_i=a_{i+1}+b_{i+1} \pmod 2$. Звідси маємо $C=(A$ xor $B)$ shr 1, де через shr позначений бітовий зсув вправо.

Враховуючи, що $b_i=0$ тоді і тільки тоді, коли $x_i=y_i$, а $c_i=0$ тільки тоді, коли $x_i+y_i<2$, складемо таблицю істинності для визначення x_i і y_i через b_i і c_i:

b_i	c_i	x_i	y_i	Примітки
0	0	0	0	
1	0	0	1	Необхідно, щоб $c_{i-1}=0$
0	1	1	1	
1	1	0	1	Необхідно, щоб $c_{i-1}=1$

У випадку, коли $b_i=1$, вибираємо саме $x_i=0$, $y_i=1$ для забезпечення умови мінімальності X.

Виходячи з таблиці, можна записати бітові вирази для знаходження X та Y через нормальну диз'юнктну форму $X=C$ and not B, $Y=B$ or C, де and, or , not позначають побітове «І», побітове «АБО» й побітове «НІ» відповідно.

Отже, якщо розв'язок задачі існує, його можна знайти за наведеними вище формулами. Для перевірки існування розв'язку знайдемо X і Y за цими формулами й перевіримо виконання умов $A=X+Y$, $B=X$ xor Y. Якщо вони виконуються — розв'язок знайдено, якщо ні — розв'язку для заданої пари чисел A, B не існує.

Отриманий алгоритм дозволяє розв'язати задачу за константний час, тобто його часова складність $O(1)$.

Альтернативний підхід. Можна помітити, що біти, встановлені в X, встановлені і в Y. Тоді $Y=X+B$. Враховуючи, що $X+Y=X+X+B=A$, отримуємо такі формули для обчислення X і Y: $X=(A-B)/2$, $Y=X+B$.

Наївний підхід. Перебирати всі числа X за зростанням від 0 до $A/2$. Якщо X xor $(A-X)=B$, то пара X, $A-X$ є розв'язком задачі.

Часова складність алгоритму $O(A)$. Такий розв'язок набирає 50 балів.

Особливості реалізації. Обмеження задачі потребує використання 64-бітних беззнакових типів даних (у C/C++ це unsigned long long, в Pascal — uint64).

2. Подарунок (Роман Єдемський)

У королівстві Олімпія знаходиться N міст і M двосторонніх доріг, кожна з яких з'єднує рівно два міста. Між двома містами може бути більше однієї дороги. Усі дороги постійно грабуються розбійниками. Останнім часом розбійникам набридло витрачати сили на пограбування і вони звернулися до могутнього і справедливого короля Олімпії з комерцій-

ною пропозицією. За цією пропозицією король повинен надіслати розбійникам подарунок, що складається із золотих і срібних монет. У відповідь на люб'язність короля розбійники припинять грабувати певні дороги. Для кожної з доріг встановлена мінімальна кількість золотих і мінімальна кількість срібних монет у подарунку, щоб її пограбування закінчились. Тобто, якщо у подарунку K золотих і L срібних монет, то припиняється пограбування усіх доріг, для яких необхідна кількість золотих монет менша або рівна K, а кількість срібних монет менша або рівна L. У скарбниці короля немає жодної золотої або срібної монети, але є Олімпійські Тугрики. Ціна однієї золотої монети в тугриках — G, а срібної — S. Король дуже хоче, щоб після відправлення подарунку розбійникам між кожною парою міст існував хоча б один безпечний шлях, який, можливо, проходить через інші міста.

Завдання. Напишіть програму gift, яка за даними про міста і дороги у королівстві й цінами монет, знайде мінімальну кількість тугриків, яку потрібно витратити королю для того, щоб отримати безпечні шляхи між кожною парою міст у королівстві.

Вхідні дані. Перший рядок вхідного файла gift. dat містить два цілих числа N і M ($2 \le N \le 200$, $1 \le M \le 50000$) — кількість міст і кількість доріг у королівстві Олімпія. Другий рядок містить числа G і S ($1 \le G, S \le 10^9$). Наступні M рядків містять інформацію про дороги й опис пропозиції розбійників. У i+2-му рядку вхідного файла розташовано 4 натуральних числа, перші два з яких — номери міст, що з'єднані i-ою дорогою (міста нумеруються від 1 до N), наступні два — мінімальна кількість золотих і мінімальна кількість срібних монет, що треба вислати розбійникам в подарунку, щоб i-ту дорогу припинили грабувати. Обидва числа не перевищують 10^9.

Вихідні дані. Єдиний рядок вихідного файла gift.sol має містити одне ціле число — мінімальну кількість тугриків, що потрібно витратити королю на купівлю золотих і срібних монет, щоб досягнути своєї мети, або — 1, у випадку, якщо жодна кількість тугриків не допоможе.

Оцінювання. Щонайменше у 30% тестів N не буде перевищувати 30 і M не буде перевищувати 150. Щонайменше у 70% тестів M не буде перевищувати 4000.

Приклад вхідних і вихідних даних

gift.dat
3 3 2 1 1 2 10 15 1 2 4 20 1 3 5 1

gift.sol
30

Рекомендації щодо розв'язування

Формалізація умови задачі. Перекладемо умову задачі мовою теорії графів. Дано граф, який складається з N вершин і M ребер, у якому між двома вершинами може існувати більше одного ребра. Для кожного ребра відомі деякі параметри $Silver_i$ і $Gold_i$ (в умові — мінімальні обмеження на кількість срібних і золотих монет).

Означення. Нехай задано деяку пару невід'ємних чисел (A, B). Граф, який складається з вихідних вершин, а також набору ребер, для яких виконується $Silver_i \le A \wedge Gold_i \le B$, будемо позначати $R(A, B)$.

Означення. Розв'язком задачі будемо називати пару чисел (A, B), для якої $R(A, B)$ — зв'язний.

Означення. Будемо вважати розв'язок (A, B) оптимальним, якщо величина $Cost(A, B)=A \cdot S + B \cdot G$ є мінімально можливою.

Означення. Позначимо $Silver=\{Silver_i | i \in (1,M)\}$.

Означення. Позначимо $Gold=\{Gold_i | i \in (1,M)\}$.

Очевидний алгоритм. Нехай пара чисел (A, B) є розв'язком задачі. Якщо $A \notin Silver$. то ми можемо зменшити A, не змінивши зв'язність графу. Аналогічно можна зробити, якщо $B \notin Gold$. Отже, оптимальний розв'язок слід шукати лише серед пар чисел (A, B), у яких $A \in Silver \wedge B \in Gold$.

Звідси можна отримати алгоритм із часовою складністю $O((N+M)M^2)$: переберемо усі такі пари (A, B) і для кожної перевіримо, чи є ця пара розв'язком, тобто чи є граф $R(A, B)$ зв'язним. Якщо є — порівняємо $Cost(A, B)$ з поточною відповіддю і змінимо її у разі необхідності.

Покращення очевидного алгоритму за допомогою бінарного пошуку. Нехай пара чисел (A, B) є розв'язком задачі. Тоді пара чисел $(A, B+1)$ також буде розв'язком задачі, оскільки ребра у граф $R(A, B)$ можуть тільки додатися.

Аналогічно, якщо пара (A, B) не є розв'язком задачі, то і пара $(A, B-1)$ також не є розв'язком задачі.

Розглянемо такий алгоритм.

- Переберемо усі можливі значення A з множини $Silver$.
- Для кожного фіксованого A зробимо бінарний пошук по B.
- Якщо для даного B граф зв'язний, то і для всіх більших B він також буде зв'язним, отже, ми можемо зменшити праву границю пошуку.
- Якщо для даного B граф незв'язний, то і для всіх менших B він також буде незв'язним, отже, ми можемо збільшити ліву границю пошуку.

Ми отримали алгоритм з часовою складністю $O((N+M)M+\log M)$.

Ефективний алгоритм. Зафіксуємо якесь $A \in Silver$. Розглянемо усі можливі ребра i, для яких $Silver_i \le A$. Побудуємо на усіх таких ребрах граф $H(A)$ з вагами ребер $Gold_i$. Нам треба знайти таке найменше B, для якого, прибравши з цього графа усі ребра з вагами, більшими за B, граф усе ще залишиться зв'язним. Якщо граф зв'язний, то в нього існує каркасне дерево, причому всі його ребра мають вагу не більшу за B.

Не важко переконатись, що зворотнє твердження також правильне: якщо у графі $H(A)$ існує каркасне дерево, усі ребра якого не перевищують якогось B', то $B \le B'$. Отже, нам треба знайти каркасне дерево, у якого максимальне ребро є найменшим можливим.

Розглянемо такий алгоритм.

- Відсортуємо ребра за зростанням $Silver_i$.
- Будемо йти по цьому масиву зліва направо й на i-му кроці додавати i-те ребро до графа. Вага ребра рівна $Gold_i$.

- Якщо при цьому утворився цикл, то знайдемо максимальне ребро у ньому і видалимо його.
- Після додавання кожного ребра перевіримо, чи зв'язний граф і якщо так — знайдемо у ньому максимальне ребро. Порівняємо вагу цього ребра з відповіддю і змінимо її, якщо треба.

Доведення коректності алгоритму

Перед початком роботи алгоритму ребер у графі немає і у ньому N компонент зв'язності. При додаванні чергового ребра у нас або зливаються 2 компоненти між собою, або ребро додається у якусь одну компоненту і з неї ж видаляється одне ребро. Неважко переконатись, що після кожного кроку алгоритму граф буде являти собою ліс.

Твердження 1. Якщо для якогось ребра (a, b) існує шлях з вершини a до вершини b, який не містить це ребро, причому у цьому шляху всі ребра мають не більшу вагу, ніж ребро (a, b), то існує мінімальне каркасне дерево, яке не містить цього ребра.

Доведення. Розглянемо будь-яке мінімальне каркасне дерево. Якщо йому не належить ребро (a, b) — твердження доведено. Припустимо, що воно містить це ребро. Після його видалення граф розпадеться на 2 дерева. Оскільки з a у b існує шлях, який оминає це ребро, то існує ребро крім (a, b), яке сполучає ці 2 компоненти, причому це ребро має вагу не більшу, ніж у (a, b). Отже, при додаванні цього ребра вага мінімального каркасного дерева не збільшується, отже, нове дерево також буде мінімальним каркасним.

Твердження 2. Кожна компонента графа є мінімальним каркасним деревом для графа з вершин тієї компоненти.

Доведення. Перед початком роботи алгоритму кожний компонент складається з однієї вершини, і, очевидно, є мінімальним каркасним деревом графа з однією вершиною. Нехай після кроку K твердження 2 справедливе. Доведемо, що після додавання одного ребра, воно також залишиться справедливим. Можливі 2 випадки.

1. Ребро сполучає 2 різних компоненти зв'язності. Оскільки до цього вони не були сполучені, то це ребро обов'язково належить мінімальному каркасному дереву з цих двох компонент. Видаливши його, ми отримуємо 2 компоненти, у яких вже побудовані мінімальні каркасні дерева. Очевидно, що при цьому отримане дерево також буде мінімальним каркасним.

2. Ребро додається до одного компонента. Розглянемо граф G, який складається з вершин і ребер цього компонента до додавання нового ребра. Усі ребра, які не належать цьому графу, можна викинути з розгляду, оскільки для кожного з них існує мінімальне каркасне дерево, якому вони не належать, що буде правильним і після додавання ребра e (за твердженням 1 для кожного з цих ребер існує шлях по дереву з одного кінця до іншого). При додаванні нового ребра в графі утвориться цикл, і щоб мінімізувати вагу дерева, потрібно з цього циклу видалити максимальне ребро. Неважко переконатись, що отримане дерево є мінімальним каркасним.

Отримано алгоритм, часова складність якого $O(NM+M\log M)$.

3. Турист (Ілля Порубльов)

Турист подорожує пішки уздовж координатної осі Ox. Йти можна в будь-якому з двох можливих напрямків і з будь-якою швидкістю, що не перевищує V, у тому числі знаходитись на місці. З газетних анонсів він знає, що у момент t_1 у точці з координатою x_1 відбудеться одна цікава подія, у момент t_2 у точці з координатою x_2 — ще одна, і т. д., до (x_N, t_N). Цікаві події достатньо короткотривалі, їх можна вважати миттєвими. Вважається. що турист відвідав подію i, якщо у момент t_i він знаходився у точці з координатою x_i.

Завдання. Напишіть програму tourist, яка знайде максимальну кількість подій, які зможе відвідати турист, для таких двох припущень: А) з початку руху (у момент часу 0) турист знаходиться у точці 0; Б) турист може обрати початкову точку, з якої він вирушить.

Вхідні дані. Перший рядок вхідного файла tourist .dat містить єдине натуральне число N ($1 \le N \le 100000$) — кількість цікавих подій. Наступні N рядків містять по два цілих числа x_i та t_i — координату і момент часу події з номером i. Останній (N+2)-ий рядок файла містить єдине ціле число V — значення максимальної швидкості руху туриста. Усі значення x_i належать діапазону $-10^8 \le x_i \le 10^8$, усі значення t_i належать діапазону $1 \le t_i \le 10^6$, значення V належить діапазону $1 \le V \le 1000$. У вхідних даних можливі різні події, що мають однакову координату x або однаковий час t, але неможливі різні події, що мають однакові x і t одночасно.

Вихідні дані. Єдиний рядок вихідного файла tourist.sol має містити два цілих числа — максимально можливу кількість подій, які турист може відвідати, якщо він розпочне рух у момент 0 з точки 0, потім максимально можливу кількість подій, які турист може відвідати, самостійно обравши точку старту.

Оцінювання. Щонайменше у 20% тестів число N не перевищуватиме 25. Щонайменше у 50% тестів число N не перевищуватиме 1000. Щонайменше у 40% тестів відповіді будуть однакові (вибір іншої точки старту не збільшуватиме максимальну кількість подій, які турист може відвідати).

Приклад вхідних і вихідних даних

tourist.dat	tourist.sol
3 –1 1 42 7 40 8 2	1 2

Пояснення. Вийшовши у момент 0 з точки 0, турист може встигнути лише на першу подію. Вийшовши з точки 42, турист може відвідати другу і третю події.

Рекомендації щодо розв'язування

З двох відповідей, які просять знайти, легше шукати другу (коли турист може прибути до моменту 0 в точку з будь-якою координатою). Першу відповідь можна знайти повторним застосуванням того ж алгоритму, відкинувши з вхідних даних події, на які

неможливо встигнути пішки з (x=0, t=0). (Отже, алгоритм доцільно організувати як підпрограму.) Зосередимося на пошуку другої відповіді.

Проста динаміка. Відсортуємо події за часом. Поставимо серію підзадач «Яку максимальну кількість подій $R(i)$ можна відвідати так, щоб останньою була подія i ?» (де i — номер події після сортування за неспаданням t). Легко отримати рівняння ДП (динамічного програмування)

$$R(i) = \max_{k\,:\,\begin{cases} 1 \leqslant k < i \\ |x_i - x_k| \leqslant (t_i - t_k)\cdot V \end{cases}} \{R(k)\} + 1.$$

Перебираємо у зовнішньому циклі значення i, для кожного i перебираємо всі менші значення k, пропускаючи ті, з яких неможливо встигнути на подію i, а серед тих, з яких можна встигнути, шукаємо максимальне $R(k)$. Знайшовши максимум, додаємо 1 — відвідання самої події i. Для 1-ої події, а також для всіх тих, на які не можна встигнути з жодної попередньої $R(i)$=1. Остаточна відповідь — max $R(i)$ (по всіх i). Цей алгоритм має складність $O(N^2)$, що при $N \approx 100\,000$ надто багато.

Евристичні оптимізації даного ДП. У щойно розглянутому ДП перебираються всі $k<i$. Це можна оптимізувати, зберігаючи результати розв'язаних підзадач у масиві пар «номер події, значення R для цієї події», впорядкованому за R. Ідучи від більших R до менших, можна гарантувати, що перша ж знайдена подія (така, що з неї можна встигнути на дану) дасть шуканий максимум.

Застосування цієї ідеї для більшості вхідних даних дає оптимізацію і тому збільшує бали за реалізацію вищенаведеного ДП. Але як розвинути її у повноцінний (завжди правильний і завжди ефективний) розв'язок — незрозуміло.

«Чесний» варіант (пам'ятати розв'язки всіх раніше розглянутих підзадач) працюватиме надто довго на вхідних даних, подібних до таких (рис. 1) (багато подій мають координати $x \approx x^\star$ у різні моменти часу, і багато подій відбуваються у час $t \approx t^\star$ у різних точках; $t^\star$ пізніший за моменти більшості подій у околі $x^\star$; швидкість V мала, і на більшість подій часу $t \approx t^\star$ або не можна встигнути взагалі ні з якої ранішої події, або можна встигнути лише з деяких подій, для яких R зовсім мале). Тоді при аналізі більшості подій вигляду $t \approx t^\star$ доведеться перебирати практично всі події у околі координати $x^\star$, нічого не знаходячи. Тобто, отримуємо ту саму складність $O(N^2)$.

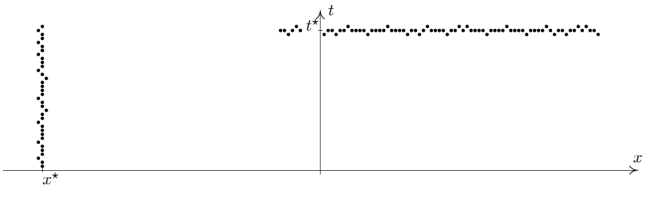

Рис. 1

«Нечесний» варіант — підтримувати перелік лише кращих результатів розв'язаних підзадач: наприклад, 500 кращих на даний момент результатів зберігаються, а ті, які не ввійшли до п'ятисот кращих — забуваються. Ця евристика прискорює програму від $O(N^2)$ до $O(500N) \approx O(N)$. Але вона не є правильним алгоритмом. Нехай зберігають не 500 кращих оцінок, а лише 5; розглянемо вхідні дані (рис. 2).

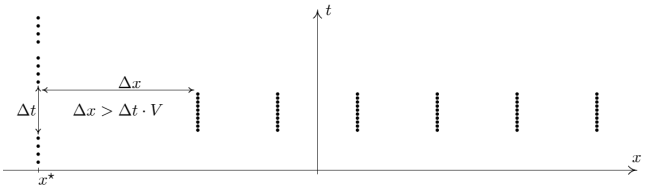

Рис. 2

Оскільки є 6 вертикальних «стовпчиків» з 10 подій кожен, інформація про найраніші 4 події при $x \approx x^\star$ буде забута. Але потім (при більших t) старі 4 плюс нові 8 виявляються більшими за 10. А дізнатись про це неможливо, бо інформація про старі 4 вже забута.

Зрозуміло, що коли запам'ятовувати 500 кращих результатів, конкретно цей приклад опрацьовується правильно. Але можна зробити вхідні дані з кількома тисячами аналогічних «стовпчиків». А збільшувати кількість кращих задач (які пам'ятаємо) суттєво понад 500 ризиковано, бо тоді втратиться оптимізація за часом роботи.

Ми детально розглянули дані евристичні модифікації ДП, бо, з одного боку, вони не є правильним ефективним алгоритмом; з іншого — *якби* тести були випадкові (незалежно рівномірно розподілені), реалізація будь-якої з цих евристик набирала б 80–95% балів.

Ефективний розв'язок (пошук монотонної підпослідовності)

Введемо косокутну систему координат (p, q) так (рис. 3), щоб подія у момент t_A в точці з координатою x_A мала координати $p_A = -x_A + t_A \cdot V$, $q_A = x_A + t_A \cdot V$. Очевидна умова $|x_k - x_j| \leq |t_k - t_j| \cdot V$ (після події j можна встигнути на подію k) набуває вигляду

$$\begin{cases} x_k - x_j \leqslant (t_k - t_j)\cdot V, \\ x_j - x_k \leqslant (t_k - t_j)\cdot V \end{cases} \Leftrightarrow$$

$$\Leftrightarrow \begin{cases} -x_j + t_j \cdot V \leqslant -x_k + t_k \cdot V, \\ x_j + t_j \cdot V \leqslant x_k + t_k \cdot V \end{cases} \Leftrightarrow \begin{cases} p_j \leqslant p_k, \\ q_j \leqslant q_k, \end{cases}$$

тобто ($p_j \leq p_k$) and ($q_j \leq q_k$). Отже, після однієї події можна встигнути на іншу тоді й тільки тоді, коли обидві координати p і q наступної події не менші, ніж у попередньої.

Відсортуємо всі події, як пари (p, q), *лексикографічно (пара* (p_1, q_1) *має йти раніше по порядку, ніж па-*

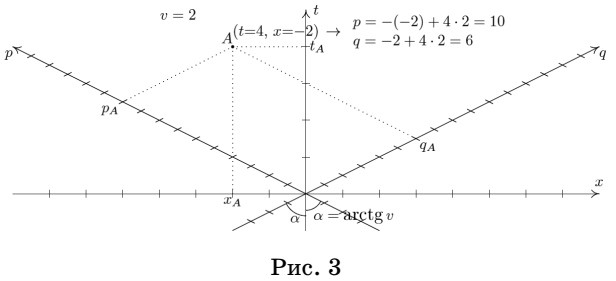

Рис. 3

ра (p_2, q_2), тоді й тільки тоді, коли ($p_1<p_2$) or (($p_1=p_2$) and ($q_1<q_2$))). Легко бачити, що умова «($p_A \le p_B$) and ($q_A \le q_B$)» рівносильна одночасному виконанню двох умов «подію A при сортуванні розмістили раніше за подію B» та «$q_A \le q_B$».

Отже, послідовність подій, таких, що після A_1 можна встигнути на A_2, після A_2 можна встигнути на A_3, ..., після A_{k-1} мож;на встигнути на A_k, характеризується тим, що усі події $A_1, A_2, ..., A_k$ розміщені після лексикографічного сортування саме в такому порядку між собою (не обов'язково підряд, між ними можуть бути якісь інші події), і $q_{A_1} \le q_{A_2} \le . . . \le q_{A_k}$. Тобто, *після вищезгаданого лексикографічного сортування достатньо знайти довжину найдовшої монотонно неопадної (за q) підпослідовності.*

Алгоритм, що дозволяє знайти найдовшу монотонно неспадну підпослідовность за час $O(n\log n)$, описаний у багатьох джерелах, зокрема — А. Шень, «Программирование: теоремы и задачи», задача 1.3.4; ru.wikipedia.org, стаття «Задача поиска наибольшей увеличивающейся подпоследовательности»; Порублев—Ставровский, «Алгоритмы и программы. Решение олимпиадных задач», розд. 13.2.2. Додамо лише, що при використанні мови C++ можна не писати свій бінарний пошук, а застосувати бібліотечну функцію upper_bound (мова йде про функцію, що застосовується, наприклад, до відсортованого vector-а, а не про метод контейнера set або подібного).

Складність даного розв'язку — $O(n\log n)$, причому є два етапи, кожен з яких має таку складність: спочатку сортування згідно ($p_1<p_2$) or (($p_1=p_2$) and ($q_1<q_2$)), потім пошук найдовшої монотонно неспадної підпослідовності.

На думку автора задачі, геометричне її трактування провокує ідею розв'язувати через замітання (воно ж «вимітання», «метод скануючої прямої», «sweeping»), але навряд чи цей метод може дати оптимальний розв'язок. Хоча б тому, що і при горизонтальному руху вертикальної прямої, і при вертикальному руху горизонтальної прямої незрозуміло, як добитися, щоб кількість точок подій ні при яких вхідних даних не сягала порядку N^2. Замітання після введення косокутньої системи координат може дати ефективний розв'язок, але він складніший за наведений.

ЗАВДАННЯ ДРУГОГО ТУРУ

1. «Точки» (Даніїл Нейтер)

На площині задано N точок.

Завдання. Напишіть програму points, яка знайде суму квадратів відстаней між всіма парами точок.

Вхідні дані. Перший рядок вхідного файла points.dat містить єдине натуральне число N ($1 \le N \le 100000$) — кількість точок. Наступні N рядків містять по два цілих числа X і Y ($-10\,000 \le X, Y \le 10\,000$) — координати точок.

Вихідні дані. Єдиний рядок вихідного файла points.sol має містити єдине ціле число — суму квадратів відстаней між усіма парами точок.

Оцінювання. Щонайменше у 50% тестів будуть виконуватись додаткові обмеження $1 \le N \le 1000$, $-1000 \le X, Y \le 1000$. Щонайменше у 70% тестів будуть виконуватись додаткові обмеження $1 \le N \le 20\,000$.

Приклад вхідних і вихідних даних

points.dat	points.sol
4 1 1 –1 –1 1 –1 –1 1	32

Рекомендації щодо розв'язування

Наївний розв'язок. Підрахувати шукану суму за означенням: перебрати всі пари індексів i і j і просумувати доданки вигляду $(x_i-x_j)^2+(y_i-y_j)^2$. Часова складність алгоритму $O(N^2)$. Такий розв'язок набирає 50 балів.

Ефективний розв'язок. Якщо перегрупувати доданки, можна розбити шукану суму на дві:

$$\sum_{i=1}^{N}\sum_{j=1}^{i-1}((x_i-x_j)^2+(y_i-y_j)^2) = \\ = \sum_{i=1}^{N}\sum_{j=1}^{i-1}(x_i-x_j)^2 + \sum_{i=1}^{N}\sum_{j=1}^{i-1}(y_i-y_j)^2.$$

Таким чином, ми можемо окремо розв'язувати задачу для іксів і для ігреків, а потім просто просумувати результат.

$$\sum_{i=1}^{N}\sum_{j=1}^{i-1}(x_i-x_j)^2 = \sum_{i=1}^{N}\sum_{j=1}^{i-1}(x_i^2-2x_ix_j+x_j^2) = \\ = \sum_{i=1}^{N}((i-1)x_i^2 - 2x_i\sum_{j=1}^{i-1}x_j + \sum_{j=1}^{i-1}x_j^2).$$

Розглянемо підрахунок суми по іксах:

$$sumx = \sum_{j=1}^{i-1}x_j \text{ та } sumx2 = \sum_{j=1}^{i-1}x_j^2$$

Отримані формули дозволяють обчислити суму реккурентно за один прохід: у міру підрахунку суми по i значення підраховуються в окремих змінних. Використовуючи підраховані значення цих сум, черговий доданок суми обчислюється за константний час. Після цього залишається додати x_i і x_i^2 до значень змінних *sumx* і *sumx2* відповідно і перейти до обчислення наступного доданку. Часова складність алгоритму $O(N)$.

Особливості реалізації. Обмеження задачі вимагають для представлення *sumx*, *sumx*2 і сумарної відповіді використовувати 64-бітні типи даних (у C/C++ це long long, в Pascal — int64).

2. Миші та сир (Роман Різванов)

Сучасні дослідження виявили, що зграя голодних мишей в пошуках сиру діє так: якщо поблизу є декілька шматків сиру, то кожна миша обирає собі найближчий, після чого всі миші одночасно починають рухатись в напрямку обраного шматка сиру. Як тільки миша або декілька мишей досягли точки призначення і там є сир, вони його з'їдають, а всі

миші, які прибігли пізніше, залишаються голодними. Швидкість пересування всіх мишей однакова.

Якщо існує декілька способів обрати найближчі шматки сиру, то миші оберуть такий спосіб, за яким мінімальна кількість мишей зграї залишиться голодною. Щоб перевірити цю теорію, учені вирішили провести експеримент. Вони розташували N мишей і M шматків сиру в прямокутній системі координат так, що всі миші знаходяться на деякій прямій $y=Y_0$, а всі шматки сиру — на інший прямій $y=Y_1$. Але, щоб перевірити результати експерименту, ученим потрібна програма, яка відтворює поведінку зграї голодних мишей.

Завдання. Напишіть програму mice, яка знаходить кількість мишей, які залишаться без сиру.

Вхідні дані. Перший рядок вхідного файла mice.dat містить чотири цілих числа N ($1\le N\le 10^5$), M ($0\le M\le 10^5$), Y_0 ($0\le Y_0\le 10^7$), Y_1 ($0\le Y_1\le 10^7$). Другий рядок містить послідовність із N строго зростаючих чисел — абсциси мишей. Третій рядок містить M строго зростаючих чисел — абсциси шматків сиру. Усі абсциси цілі і не перевищують за модулем 10^7.

Вихідні дані. Єдиний рядок вихідного файла mice.sol має містити єдине число — мінімальну кількість мишей, які залишаться без сиру.

Оцінювання. Щонайменше у 20% тестів виконуються додаткові обмеження $0\le N\le 20$. Щонайменше у 50% тестів виконуються додаткові обмеження $0\le N, M\le 1000$.

Приклад вхідних і вихідних даних

mice.dat	mice.sol
3 2 0 2 0 1 3 2 5	1

Пояснення. Усі три миші оберуть перший шматок сиру. Сир з'їдять друга і третя, які прибіжать до нього одночасно. Перша залишиться голодною, бо бігла в тому ж напрямку і запізнилася. Другий шматок сиру залишиться нез'їденим.

Рекомендації щодо розв'язування

Позначення. $MouseX_i$ — абсциса миші з номером i. $CheeseX_j$ — абсциса шматка сиру з номером j. $D(i, j)$ — відстань від i-ої миші до j-ого шматка сиру.

Припущення. Будемо вважати, що $N>0$ і $M>0$, бо якщо N або M дорівнює нулю, то відповідь до задачі — це N. Також припустимо що $Y_0=Y_1=0$, тобто всі миші і шматки сиру знаходяться на одній горизонтальній прямій.

Ефективний розв'язок

Лема 1. Кількість найближчих шматків сиру для кожної миші не більше ніж 2.

Доведення. Існує всього 2 напрямки, куди миша може побігти: вліво або вправо. Як тільки вона натрапить на сир, то цей сир буде найближчим у відповідному напрямку.

Об'єднаємо мишей і шматки сиру в один відсортований за абсцисою список. Так як в умові сказано, що списки мишей і шматків сиру відсортовані, то це можна зробити зі складністю $O(N)$. Тоді за один прохід зліва направо можна для кожної миші знайти найближчий шматок сиру зліва. Для цього будемо запам'ятовувати останній розглянутий шматок сиру, він і є найближчий для миші, яку ми розглядаємо. Аналогічно шукаються найближчі справа. Для кожної миші виберемо з двох напрямків той, який дає коротший шлях, або обидва, якщо вони однакової довжини. Якщо на обраному напрямку між сиром і мишею є інша миша, то ми цей напрямок вилучаємо, так як інша миша швидше з'їсть той сир. Тепер усі напрямки мишей безпосередньо ведуть до сиру, і до кожного шматка сиру веде не більше одного напрямка з кожної сторони.

Будемо опрацьовувати мишей зліва направо. Якщо чергова миша може рухатись вліво, і сир зліва не обрала жодна з попередніх мишей, або цей сир обрала миша з такою самою відстанню до сиру, то ця миша, рухаючись вліво, встигне поїсти сир, не заважаючи іншим мишам. Так як до сиру веде не більше одного напрямку, то цей вибір не вплине на наступних мишей, а тому і не може в подальшому погіршити відповідь. В інших випадках, рухаючись вліво, ми не зможемо покращити відповідь, тому, якщо миша має можливість рухатись вправо, то це єдина можливість для цієї миші відвідати сир, тому вона обирає правий напрямок. Якщо миша не може рухатись вправо, то ніяк покращити відповідь вона не зможе.

Оскільки кожен вибір рухатись вліво ніяк не впливає на наступні і кожен такий вибір не збільшує відповіді, то цей вибір є оптимальним для відповідної миші. Правий напрямок обирається тільки, якщо, рухаючись вліво, ми погіршимо відповідь, цей вибір може в подальшому збільшити відповідь на одиницю, але так як ми заощадили одиницю, не рухаючись вліво, то цей вибір не гірше, чим вибір рухатись вліво (але рухатись вліво може бути гірше, чим рухатись вправо). Тому наведена стратегія дає оптимальну відповідь. Наведений алгоритм має складність $O(N)$.

Зведення до одновимірного випадку

Лема 2. Y_0 і Y_1 не впливають на результат.

Доведення. Єдине, на що впливають Y_0 і Y_1, це відстані $D(i, j)$, але згідно до розв'язку нам не потрібно їх обчислювати, а потрібно вміти перевіряти чи $D(i, j)<D(i, k)$ для деяких j і k, так як відстані невід'ємні, то цю перевірку можна переписати як

$$D^2(i,j)<D^2(i,k) \Leftrightarrow (MouseX_i-CheeseX_j)^2+(Y_0-Y_1)^2<$$
$$<(MouseX_i-CheeseX_k)^2+(Y_0-Y_1)^2 \Leftrightarrow (MouseX_i-$$
$$-CheeseX_j)^2<(MouseX_i-CheeseX_k)^2 \Leftrightarrow |MouseX_i-$$
$$-CheeseX_j|<|Mouse_i-CheeseX_k|.$$

Таким чином, не втрачаючи загальності, можна перенести (зберігаючи абсциси) всіх мишей і шматки сиру на пряму $Y=0$.

3. Мутація (Ярослав Твердохліб)

Учені планети Олімпія проводять черговий експеримент з мутації примітивних організмів. Геном організму з цієї планети може бути представлений у вигляді рядка з перших K великих літер англійського алфавіту. Для кожної пари типів генів було встановлено ризик виникнення захворювання, за умови, що гени цих типів стоять підряд у геномі. Ризик виникнення захворювання в організму рів-

ний сумі ризиків для кожної пари сусідніх генів у геномі і вимірюється він у ризикограмах.

Учені вже отримали базовий організм, з якого шляхом мутації можуть бути отримані інші організми. Механізм мутації включає видалення всіх генів певних типів. Таке видалення збільшує ризик виникнення захворювання, причому для кожного типу генів було визначено, на скільки збільшиться ризик захворювання організму, якщо цей тип буде видалено. Мета вчених — порахувати кількість різних організмів, які можна отримати з базового описаним вище шляхом, причому ризик виникнення захворювання у яких не перевищує T ризикограм. Два організми вважаються різними, якщо рядки, що задають їхні геноми, відрізняються. Геном отриманого організму має складатись хоча б з одного гена.

Завдання. Напишіть програму mutation, яка за інформацією про геном базового організму і параметрах виникнення ризику захворювання, знайде кількість різних організмів, які можна отримати з базового, таких, що ризик виникнення захворювання не перевищує T ризикограм.

Вхідні дані. Перший рядок вхідного файла mutation.dat містить 3 цілих числа: N ($1 \le N \le 200\,000$) — довжину генома початкового організму, K ($1 \le K \le 21$) — кількість різних літер, які можуть зустрічатись у геномі й T ($1 \le T \le 10^9$) — максимальний допустимий ризик виникнення захворювання. Другий рядок містить геном базового організму — рядок довжини N, що складається лише з перших K великих літер англійського алфавіту. Третій рядок містить K чисел, що задають ризик виникнення захворювання при видаленні всіх генів певного типу, причому i-те число відповідає i-ій букві англійського алфавіту. Наступні K рядків містять по K чисел, причому j-те число в рядку з номером i з них задає ризик виникнення захворювання для пари генів, перший з яких відповідає i-ій букві, а другий — j-ій букві. Усі числа у вхідному файлі цілі, невід'ємні й не перевищують 10^9. Усі ризики задано в ризикограмах. Гарантується, що найбільший можливий ризик захворювання організму, який може бути отриманий з початкового, строго менший за 2^{31}.

Вихідні дані. Єдиний рядок вихідного файла mutation. sol має містити одне ціле число — шукану кількість різних організмів з ризиком виникнення захворювання не більше T ризикограмів, які можуть бути отримані з базового шляхом, описаним в умові.

Оцінювання. Набір тестів складається з 5 окремих блоків, з такими додатковими обмеженнями:

1. $K \le 10$, $N \le 30000$ — 15% тестів.
2. $K \le 14$, $N \le 200000$ — 35% тестів.
3. $K \le 17$, $N \le 200000$ — 25% тестів.
4. $K \le 19$, $N \le 200000$ — 10% тестів.
5. $K \le 21$, $N \le 200000$ — 15% тестів.

Приклад вхідних і вихідних даних

mutation.dat	mutation.sol
5 3 13 BACAC 4 1 2 1 2 3 2 3 4 3 4 10	5

Пояснення. Можливо отримати такі організми (у дужках вказано ризики): BACAC(П), ACAC(IO), BAA(5), B(6), AA(4).

Рекомендації щодо розв'язування

1. Очевидний алгоритм. Переберемо всі можливі множини символів, які будуть викидатись. Для кожної такої множини знайдемо рядок (геном), який залишиться після викидання цієї множини. Для цього рядка знайдемо його вартість (ризик захворювання) і додамо до неї вартість викидання набору. Складність такого алгоритму $O(2^K N)$.

2. Попередній підрахунок. Умова задачі наштовхує на перебір усіх множин символів, що видаляються. Тому множник 2^K з оцінки складності прибрати не вийде. Отже, залишається лише зменшувати коефіцієнт при ньому. Для цього треба навчитись для кожної множини символів швидко шукати вартість рядка, що залишиться після її видалення.

Нехай M — набір символів, що викидаються, а S — початковий рядок.

Розглянемо символи початкового рядка, що стоять на позиціях l і r ($l<r$). Який вигляд повинен мати M, щоб l і r стали поруч після викидання? Виходячи з природніх міркувань, можемо поставити 2 умови:

$$\forall i \in (l, r) : S_i \in M;$$
$$S_l \notin M \wedge S_r \notin M.$$

З другої умови маємо, що для фіксованого l кандидатів на r не може бути більше, ніж K, оскільки між l і r не може бути символів S_r. Таким чином, кількість пар потенційних сусідів не перевищує NK. Ми можемо перебрати їх усіх за $O(NK)$, йдучи вказівником на l справа наліво і підтримуючи для кожного символу номер його останньої зустрічі.

Зафіксуємо деяку пару потенційних сусідів, які стоять на позиціях l і r. Насправді, для нас не важливо, на яких саме позиціях стоять ці символи у початковому рядку, якщо ми знаємо, що це за символи і які символи стоять між ними, оскільки нам потрібно лише вміти для кожного M знаходити вартість рядка, що залишився, а не сам рядок. Нехай $M' = \{S_i \mid i \in (l, r)\}$. Тоді трійка (S_l, S_r, M') дозволяє однозначно визначити, чи стануть символи на позиціях l і r сусідами, що, у свою чергу, дає нам знати, чи треба додавати вартість їх сусідства до сумарної вартості рядка при її підрахунку.

Звідси отримуємо алгоритм.

Переберемо усі пари кандидатів на l і r, яких не більше за NK. Для кожної пари знайдемо M'. Заведемо масив, у якому для усіх трійок (a, b, P) будемо зберігати сумарну вартість сусідства для усіх пар l і r, таких що:

- $l<r$;
- $S_l = a \wedge S_r = b$;
- $M' = P$.

Усе це можна зробити за $O(NK)$ часу. Далі переберемо всі набори символів, що викидаються. Для кожного фіксованого набору M переберемо всі трійки (a, b, P), і перевіримо, чи виконуються для них такі умови:

- $a \notin M \wedge b \notin M$;
- $P \subset M$.

Для трійок, для яких вони виконуються, знайдемо сумарну вартість, додамо до неї вартість викидання набору M та якщо усе це не перевищує T —

збільшимо відповідь на 1. Складність такого алгоритму $O(4^K K^2+NK)$.

3. Швидкий перебір підмасок. Викладений вище алгоритм за даних обмежень працює не набагато швидше, ніж очевидний перебір, але він має одну суттєву перевагу — при експотенційному множнику в оцінці складності не присутня довжина рядка. Не зважаючи на це, експонента зросла від 2 до 4 , що без сумніву є дуже повільним. Якщо подивитись, звідки береться це 4, ми побачимо, що для кожного набору M ми перебираємо всі можливі P і вже потім перевіряємо, чи є вони його підмножинами. Таким чином, ми переглядаємо багато зайвих кандидатів на P, що дуже сильно впливає на час роботи програми.

Розглянемо всі пари (M, P). Не важко переконатись, що для кожного символа існують лише 3 варіанти:

- символ належить і M, і P;
- символ належить M, але не належить P;
- символ не належить ні M, ні P.

Отже, усього таких пар 3^K. Якщо ми зможемо для фіксованого M перебрати лише такі P, які є його підмножинами, то сумарна складність алгоритму зменшиться до $O(3^K K^2+NK)$.

Зазвичай, під час перебору множини задаються бітовими масками, які зберігаються у вигляді цілих чисел. Останні K бітів числа однозначно визначають множину: якщо біт з номером i рівний одиниці, то i-й символ присутній у множині, інакше він відсутній. Зараз буде описано алгоритм, який за даною бітовою маскою перебирає усі її підмаски у спадному порядку (під час порівняння бітових масок порівнюються числа, що їх задають).

Нехай нам дано бітову маску $mask$ і ми вже перебрали всі її підмаски, які більші або рівні за $mask'$, причому $mask' \neq 0$. Знайдемо наступну в порядку спадання підмаску. Поглянемо на останню групу нульових бітів у числі $mask'$, перед якими стоїть одиничний біт. Щоб отримати наступну у порядку спадання підмаску, нам потрібно обнулити цей одиничний біт, а всі біти після нього, які є одиничними у $mask$, зробити одиничними і у $mask'$. Неважко переконатись, що операція $mask'=(mask'-1)\&mask$ саме це і зробить. Тут під операцією & розуміється побітове «І».

Як уже було написано вище, це дає нам змогу покращити складність алгоритму до $O(3^K K^2+NK)$.

4. Формула включення-виключення. Нам вдалося зменшити експоненту в оцінці складності, тепер спробуємо зменшити множник при 3^K. Цей множник з'являється через те, що окрім множини M та її підмножини P ми перебираємо ще 2 символи, які не повинні бути присутніми в M.

Розглянемо такий (неправильний) алгоритм.

Для кожної пари кандидатів на сусідство l і r будемо зберігати лише множину M' символів, що знаходяться між ними у S. Заведемо масив v, у якому для кожної множини P будемо зберігати сумарну вартість усіх пар l і r, для яких M' співпадає з P. При фіксованому M знайдемо суму по всіх його підмножинах у v і вважатимемо, що це і є вартістю отриманого рядка.

Як уже було сказано, цей алгоритм не є правильним. Кожну пару (l, r) ми враховуємо у деяких M, у яких за попереднім алгоритмом ми не повинні були їх враховувати, а саме:

- Ми додали вартість (l, r) до всіх множин, які містять l, отже, нам потрібно відняти цю вартість від $v[M\cup\{S_l\}]$.
- Аналогічно, віднімемо цю вартість від $v[M\cup\{S_r\}]$.
- Оскільки тепер ми відняли цю вартість двічі від усіх множин, що містять обидва кінці, віднімемо її від $v[M\cup\{S_l, S_r\}]$.

Тепер після виконання описаного вище алгоритму сумарна вартість отриманого рядка буде рахуватись правильно, отже, і алгоритм буде працювати правильно. Складність становить $O(3^K+NK)$.

5. Підрахунок суми по всіх підмасках усіх масок у режимі off-line. Описаний вище алгоритм при кожному новому M перебирає усі його підмножини, хоча масив v не змінюється з моменту заповнення. Якщо нам вдасться шляхом деяких маніпуляцій перетворити масив v так, щоб для кожної множини M у ньому одразу зберігалась сума значень зі старого v по всіх її підмасках, то знаходити вартість рядка ми зможемо за $O(1)$.

Позначимо через $mask_k$ бітову маску, яка утворена з останніх (молодших) k біт маски $mask$.

Розглянемо такий ітеративний алгоритм, який складається з K ітерацій. Перед початком (після нульової ітерації) масив v побудований за попереднім пунктом розв'язку. Після ітерації з номером k для кожної бітової маски $mask$ у комірці $v[mask]$ буде зберігатись сума всіх значень з початкового v по усіх масках $mask'$, для яких виконується

$$mask_k' \subset mask_k;$$

перші (старші) $K-k$ біт маски $mask'$ співпадають з першими $K-k$ бітами маски $mask$.

На ітерації з номером k будемо послідовно у порядку зростання для усіх масок знаходити $v[mask]$. Неважко переконатись, що якщо біт з номером k (якщо вважати, що біти нумеруються з одиниці) рівний 0, то $v[mask]$ не зміниться з попередньої ітерації. Якщо ж цей біт рівний 1, то до $v[mask]$ потрібно додати $v[mask\setminus\{k\}]$.

Після виконання K ітерацій цього алгоритму у кожній комарці нового масиву v буде сума по всіх підмасках маски, що задається номером цієї комірки, з початкового v. Розв'язок набуває складності $O(2^K K+NK)$.

6. Остаточний алгоритм. Підсумуємо викладений вище алгоритм.

- Переберемо всі пари потенційних сусідів з рядка S (параграф 2).
- За цими парами, використовуючи формулу включення-виключення, побудуємо масив v (параграф 4).
- Застосуємо ітеративний алгоритм, після виконання якого у масиві v будуть суми з початкового v по всіх підмасках кожної маски (параграф 5).
- Переберемо множину символів M, які будуть викинуті, і за допомогою масиву v знайдемо вартість рядка, який залишиться після її викидання, а також вартість самого викидання множини M.
- Якщо ця вартість не перевищує T — збільшимо відповідь на 1.

7. Зауваження. Під час виконання пунктів 2–3 алгоритму з попереднього параграфа в масиві v може статись переповнення типу int у C++ або longint у Pascal, але оскільки в умові гарантується, що вартість кожного з отриманих рядків є меншою за 2^{31} — на загальний результат це не вплине.