Завдання XXVI Всеукраїнської учнівської олімпіади з інформатики

2012/13 н. р.

Віталій Бондаренко, Роман Єдемський, Данило Мисак,
Сергій Нагін, Ярослав Твердохліб, Шаміль Ягіяєв

Зміст

Завдання першого туру

1.1. Календар (Данило Мисак)

Умова

Учені-археологи планети Олімпія знайшли дві печери з ознаками перебування доісторичних племен. Їхню увагу привернули N різних слів, накреслених на стіні у кожній з печер. Цікаво, що ці слова в обох печерах виявилися однаковими, щоправда виписані у різній послідовності. Вчені з'ясували:

1. Накреслені слова — це назви місяців року, що перераховані в порядку настання у відповідного племені.
2. Рік у племен був розбитий на N рівних за тривалістю місяців, а дні початку місяців збігалися.

Однак, учені так і не визначили, в який день починався рік у кожного з племен.

Завдання. Напишіть програму `calendar`, що за даними про послідовності назв місяців в обох печерах знайде найбільшу кількість місяців, які могли б мати однакові назви в обох племен, враховуючи, що рік у племен міг починатися в різні моменти часу. Для спрощення аналізу вчені встановили для кожної з назв місяців свій номер — натуральне число від 1 до N.

Вхідні дані. Вхідний файл `calendar.dat` складається з трьох рядків. У першому рядку міститься натуральне число N ($2 \leqslant N \leqslant 10^5$) — кількість назв місяців, накреслених на стіні кожної з печер. Другий рядок містить N різних натуральних чисел, кожне з яких не перевищує N, — номери слів у порядку, в якому вони накреслені у *першій* печері. Третій рядок також містить N різних натуральних чисел, кожне з яких не перевищує N, — номери слів у порядку, в якому вони накреслені у *другій* печері.

Вихідні дані. Вихідний файл `calendar.sol` повинен містити єдине число — найбільшу кількість місяців, які могли б називатися однаково в обох племен.

Оцінювання. Набір тестів складається з 4 блоків, для яких додатково виконуються такі умови:

- 20 % балів: $2 \leqslant N \leqslant 5$.
- 20 % балів: $5 < n \leqslant 150$.
- 20 % балів: $150 < N \leqslant 3000$.
- 40 % балів: $3000 < N \leqslant 10^5$.

Приклади вхідних та вихідних даних.

calendar.dat	calendar.sol
4 2 4 3 1 4 2 1 3	2
3 3 2 1 1 2 3	1

Пояснення до першого прикладу. Якщо рік у другого племені починається, наприклад, на місяць пізніше, ніж у першого, то два місяці мають у племен однакові назви (номер 1 і 4):

Плем'я 1:	3	1	2	4	3	1	2	4	3	1	2	4
Плем'я 2:	2	1	3	4	2	1	3	4	2	1	3	4

Жодна інша комбінація початків року не приводить до збігу більшої кількості назв місяців.

Пояснення до другого прикладу. Незалежно від того, коли саме у племен починається рік, однакову назву завжди матиме рівно один місяць.

Розв'язання

Задачу можна розв'язувати простим способом, перебравши всі можливі відповідності між місяцями першого і другого племені та для кожної відповідності підрахувавши кількість місяців, які мають однакові назви. Щоб установити відповідність між місяцями, достатньо визначити одну пару місяців, які збігаються в часі. Наприклад, у таблиці з умови задачі збігаються у часі місяць під назвою 3 у першого племені і місяць під назвою 2 у другого племені (перший стовпчик таблиці):

Плем'я 1:	3	1	2	4	3	1	2	4	3	1	2	4
Плем'я 2:	2	1	3	4	2	1	3	4	2	1	3	4

Відповідність пари місяців однозначно визначає решту таблиці. Отже, достатньо перебрати всі N^2 пар місяців, які можуть збігатися у часі, і для кожної пройти послідовні N стовпчиків таблиці, щоб з'ясувати, скільки з них містять однакові числа. Ефективність часу виконання такого алгоритму становить $O(N^3)$, він набирає 40 % від загальної кількості балів.

Пришвидшити алгоритм можна, якщо для знаходження відповідності календарів племен не перебирати всі можливі пари місяців. Достатньо зафіксувати довільний місяць першого племені і перебрати всього N варіантів — місяці у календарі другого племені, які можуть йому відповідати. Тим не менше для кожного варіанта все одно доведеться рахувати кількість однакових чисел у N стовпчиках таблиці, тому складність удосконаленого алгоритму становить $O(N^2)$ і він набирає тільки 60 % від загальної кількості балів. Повний же бал дозволяє набрати принципово інший підхід.

Позначимо через a_i, $1 \leqslant i \leqslant N$, місце, на якому стоїть число i у календарі першого племені, а через b_i, $1 \leqslant i \leqslant N$, місце, на якому стоїть число i у календарі другого племені. При цьому для зручності вважатимемо, що нумерація місць починається з нуля, тобто $0 \leqslant a_i \leqslant N-1$ і $0 \leqslant b_i \leqslant N-1$. Якщо у другого племені рік починається на d місяців пізніше, ніж у першого ($0 \leqslant d \leqslant N-1$) і таблиця починається з першого місяця року першого племені, то число i стоїть у стовпчиках $a_i \pmod N$ (тобто a_i, $a_i + N$, $a_i + 2N$ і т. д.) у першому рядку таблиці та у стовпчиках $b_i + d \pmod N$ у другому рядку. Ці стовпчики збігаються тоді й лише тоді, коли $a_i \equiv b_i + d \pmod N$, тобто $a_i - b_i \equiv d \pmod N$.

Таким чином, за умови, що у другого племені рік починається на d місяців пізніше, ніж у першого, збігатися будуть місяці з тими й тільки тими назвами i, $1 \leqslant i \leqslant N$, для яких різниця $a_i - b_i \equiv d \pmod N$. Тепер залишається утворити масиви чисел a_i та b_i, підрахувати всі різниці $a_i - b_i$ за модулем N та сформувати масив чисел c_k, $0 \leqslant k \leqslant N-1$, де c_k дорівнює кількості різниць $a_i - b_i$, що за модулем N дорівнюють числу k. Найбільше з чисел c_k і є шуканою величиною (а відповідне значення k визначає, на скільки місяців пізніше повинен починатися рік у другого племені, ніж у першого, щоб однакові назви у племен мали c_k місяців).

Як видно, алгоритм розв'язує задачу за лінійний час $O(N)$.

1.2. Мутація (Ярослав Твердохліб)

Умова

Учені-генетики планети Олімпія знову проводять експерименти над ДНК примітивних організмів. Геном організму — це послідовність генів, кожний з яких можна закодувати одним натуральним числом. Гени, що кодуються одними і тими самими числами, вважаються однаковими, і навпаки, гени, що кодуються різними числами, вважаються різними.

Учені вже вивели деякий примітивний організм, якому вони хочуть модифікувати геном таким чином, щоб отримати ідеальний організм. Вони вважають, що у подальшому це допоможе знайти ліки від багатьох хвороб.

Організм вважається ідеальним, якщо будь-які два однакових гени або стоять на сусідніх позиціях у геномі, або між ними є хоча б один ген такий самий, як вони.

За одну операцію вчені можуть вибрати й видалити один або кілька *однакових* генів з генома організму, після чого вставити всі ці гени назад у геном, але, можливо, в інші позиції. Оскільки кожна така операція послаблює організм, учені хочуть досягнути своєї мети, виконавши якнайменшу кількість операцій.

Завдання. Напишіть програму `mutation`, яка за поданням генома примітивного організму визначить найменшу кількість операцій, яку необхідно виконати, щоб отримати ідеальний організм.

Вхідні дані. Перший рядок вхідного файлу `mutation.dat` містить ціле число N ($1 \leqslant N \leqslant 10^5$) — кількість генів у геномі примітивного організму. У наступному рядку записано N натуральних чисел, кожне з яких не перевищує N, — послідовність генів у геномі.

Вихідні дані. Вихідний файл `mutation.sol` має містити одне ціле число — найменшу кількість операцій, за яку вчені зможуть отримати ідеальний організм.

Оцінювання. Набір тестів складається з 2 блоків, для яких додатково виконуються такі умови:

- 50 % балів: N не перевищує 16.
- 50 % балів: немає додаткових обмежень.

Приклад вхідних та вихідних даних.

mutation.dat	mutation.sol
9 1 2 1 3 1 3 2 4 5	2

Пояснення. Нижче подано одну з можливих послідовностей виконання операцій. Жирним виділено гени, які будуть переміщені після виконання чергової операції:

1 **2** 1 3 1 3 **2** 4 5 → **1 1** 3 **1** 3 2 2 4 5 → 1 1 1 3 3 2 2 4 5

Розв'язання

Розглянемо геном, який було отримано після виконання деякої оптимальної послідовності операцій. Очевидно, що під час кожної операції нам є сенс ставити всі гени одразу на свої місця у кінцевому геномі. Це означає, що кожен тип генів було видалено не більше ніж один раз. Отже, наша задача стає еквівалентна такій: вибрати максимальну кількість типів генів, яка при видаленні генів усіх інших типів перетворює початковий геном в ідеальний.

Кожному типу генів поставимо у відповідність відрізок прямої, який починається у крайньому зліва гені такого типу й закінчується у крайньому справа. Тоді нам потрібно вибрати максимальну за розміром підмножину відрізків, у якій жодні два відрізки не перетинаються.

Останню задачу можна розв'язувати таким жадібним алгоритмом. Відсортуємо всі відрізки за неспаданням правого кінця. Будемо йти по відсортованому списку відрізків зліва направо. Якщо черговий відрізок перетинається з тим, який ми останнім включили до нашої множини, то пропустимо його, інакше — додамо у відповідь.

Доведемо коректність описаного алгоритму методом математичної індукції за розміром початкової множини відрізків. Для множини розміру 1 алгоритм, очевидно, є правильним. Припустимо тепер, що алгоритм працює коректно для всіх множин відрізків розміру, меншого за K. Нехай у нас є множина розміру K та ми хочемо знайти для неї оптимальну підмножину відрізків. Розглянемо будь-яку підмножину, що є оптимальною. Якщо її крайній лівий відрізок не є першим у порядку сортування за правим кінцем, то видалимо його з відповіді та замість нього додамо відрізок, у якого правий кінець найлівіший. При цьому новий відрізок, очевидно, не буде перетинатись з жодним іншим відрізком з оптимальної підмножини. Тож можемо вважати, що оптимальна підмножина містить даний відрізок, та видалити його й усі відрізки, які з ним перетинаються, з нашої множини розміру K. Отримаємо множину меншого розміру, для якої за припущенням алгоритм працює коректно.

Лишається зауважити, що всі відрізки вже є відсортованими від самого початку: достатньо йти по масиву зліва направо і розглядати черговий відрізок, лише коли поточний ген є правим кінцем якогось відрізка.

Складність запропонованого алгоритму становить $O(N)$.

1.3. Космічні камінці (Данило Мисак)

Умова

Основною коштовністю на планеті Олімпія є камінці, які час від часу падають на поверхню планети з космосу. Що важчий камінець, то він цінніший. Щоб забезпечити функціонування планетарних установ, уряд час від часу збирає з містечок Олімпії податок. Із кожного з M містечок у столицю привозять по одному камінцю. Міністр вибирає з усіх камінців найважчий і бере його як податок. Решту $M-1$ камінців відвозять назад у містечка, звідки їх привезли. Бажаючи заощадити, кожне містечко завжди везе у столицю найлегший з усіх коштовних камінців, які на даний момент має у своїй скарбниці.

Завдання. Напишіть програму `stones`, яка, знаючи, в якому порядку в містечка падали камінці з космосу й маси цих камінців, для кожної події оподаткування визначить, камінець якої ваги уряд забрав як податок.

Вхідні дані. У першому рядку вхідного файлу `stones.dat` записано два числа: кількість подій N і кількість містечок M, $2 \leqslant M < N \leqslant 2 \cdot 10^5$. Кожна подія одного з двох типів: або в деяке містечко падає камінець (тип 1), або уряд збирає податок (тип 2). Наступні N рядків файлу містять опис відповідних подій у тому порядку, в якому вони відбувалися. Перше число рядка, який задає подію, — це тип події (1 або 2).

1. Якщо перше число рядка 1, то після нього рядок містить іще два натуральних числа T і W, де T — номер містечка, куди впав камінець, $1 \leqslant T \leqslant M$, а W — вага камінця, $1 \leqslant W < 10^9$.
2. Якщо перше число рядка 2, то воно єдине у рядку.

Вважаємо, що перед першою подією в жодному містечку не було жодного коштовного каміння. Вхідні дані гарантують виконання таких умов:

1. Маси всіх камінців, що падали на планету, попарно різні.
2. На момент, коли уряд збирає податок, у кожному з M містечок є хоча б по одному камінцю.
3. Уряд зібрав податок хоча б один раз.

Вихідні дані. Вихідний файл `stones.sol` повинен містити стільки рядків, скільки подій типу 2 задано у вхідному файлі: для i-ї по порядку події типу 2 в i-му рядку вихідного файлу має бути записане єдине число — маса камінця, взятого урядом як податок під час даної події.

Оцінювання. Набір тестів складається з 3 блоків, для яких додатково виконуються такі умови:

- 30 % балів: $2 \leqslant M < N \leqslant 5000$.
- 30 % балів: $5000 < N \leqslant 2 \cdot 10^5$, $2 \leqslant M \leqslant 5$.
- 40 % балів: $5000 < N \leqslant 2 \cdot 10^5$, $5 < M < N$.

Приклад вхідних та вихідних даних.

stones.dat	stones.sol
9 2 1 1 9 1 2 3 1 1 4 2 1 1 2 1 2 5 2 2 1 2 1	4 3 5

Розв'язання

Для зручності у розв'язку називатимемо події операціями.

Найпростіший спосіб розв'язання такий: для кожного містечка ми заводимо масив, скажімо, на N елементів, куди будемо записувати маси камінців, які впали у дане містечко (оскільки операцій усього N, у жодне місто не впаде більше ніж N камінців). Під час кожної операції типу 1 потрібно лише дописати в кінець відповідного масиву вагу нового камінця, а під час операції типу 2 — пройти по всіх містечках, у кожному визначити вагу найлегшого камінця та з m отриманих чисел вивести найбільше. Після цього слід видалити виведене число з масиву відповідного містечка: можна просто позначити число як вилучене, можна зсунути наступні елементи масиву на одне місце вліво або поміняти місцями останній елемент масиву з даним і зменшити лічильник кількості елементів у масиві на 1.

Підрахуємо ефективність опрацювання операцій обох типів. Операції типу 1 алгоритм, очевидно, опрацьовує за $O(1)$. Утім, для опрацювання операцій типу 2 алгоритм має пройтися по всіх містах і по всіх камінцях у них, тобто витратити $O(N+M)=O(N)$ часу. Разом із тим кількість операцій типу 2 може досягати $[(N-M+1)/2]=O(N-M)$. Таким чином, алгоритм працює $O(N(N-M))$ часу й набирає 30 % від загальної кількості балів.

Добрати ще 30 % балів можна, якщо зберігати камінці в кожному містечку не як звичайний масив, а у вигляді черги з пріоритетами (легші камінці пріоритетніші), наприклад бінарною купою. Операції типу 1 вимагатимуть додавання в одну з куп нового елемента. Оскільки в містечку може бути до $N-M+1=O(N-M)$ камінців водночас, одна операція займатиме $O(\log(N-M))$ часу. Операції типу 2 вимагатимуть розгляду верхніх елементів усіх M куп та видалення елемента однієї з них — це потребуватиме $O(M+\log(N-M))$ часу. Сумарно алгоритм працюватиме $O(M(N-M)+N\log(N-M))$ часу.

Кожен із двох вищенаведених алгоритмів цілком може використовувати масиви однакової розмірності — $O(N)$ або $O(N-M)$ — для кожного містечка, адже загальний обсяг використовуваної пам'яті $O(NM)$ чи $O(M(N-M))$ виходить відносно невеликим для тих значень змінних, за яких алгоритм не перевищує обмеження на час. Але, якщо ми хочемо створити ефективніший алгоритм, який працюватиме для більших значень змінних, заводити окремі масиви сталої довжини не вийде. Щоб обійти цю проблему, можна використати динамічні масиви.

Втім, використання великої кількості динамічних масивів (у даному випадку до двохсот тисяч) може призвести до суттєвого сповільнення програми або навіть до неможливості на деякому кроці перерозподілити пам'ять. Забезпечити стабільне виконання програми можна в такий спосіб: перед запуском основного алгоритму зчитуємо вхідний файл до кінця і для кожного міста i, $1 \leqslant i \leqslant M$, визначаємо кількість камінців c_i, які там упадуть. Далі статичний масив довжини n розділяємо на $M + 1$ частину довжин $c_1, c_2, \ldots, c_m, n - \sum_{i=1}^{m} c_i$. Кожну частину, крім останньої, будемо використовувати як окремий масив для зберігання мас камінців, що впали у відповідне містечко. Вхідний же файл потім зчитуємо повторно.

Слабке місце алгоритму, який набирає 60 % балів, полягає в тому, що для знаходження найбільшої з M мас він витрачає $O(M)$ часу. Чому б не реалізувати бінарну купу також і на цих M числах (масах, які є найменшими на даний момент у M містечках)? При цьому в новій купі пріоритетнішими будуть уже важчі камінці, а не легші.

Так і зробимо. Назвімо бінарні купи, що відповідають M містечкам, купами першого рівня, а нову купу — купою другого рівня. Домовимось, що купа другого рівня, як і всі купи першого рівня, перед виконанням першої операції порожня. Оцінимо ефективність алгоритму. Операцію типу 1 він виконує у два кроки:

1. Спершу, як і раніше, додаємо камінець (а точніше, його вагу) до бінарної купи першого рівня, яка відповідає містечку T, куди впав камінець. На це потрібно $O(\log(N - M))$ часу.
2. Якщо камінець, що впав, є на даний момент єдиним камінцем у містечку T, просто додаємо його в купу другого рівня. Якщо ні і якщо в місті T, куди впав новий камінець, змінився камінець найменшої ваги (очевидно, на цей новий камінець), потрібно оновити відповідний елемент купи другого рівня. Для цього за спеціальним індексом, який будемо зберігати й оновлювати, знаходимо місце, на якому стоїть у купі другого рівня мінімальний елемент купи T першого рівня. Оновлюємо (тобто зменшуємо) цей елемент. Далі діємо подібно до того, як проводять у купі спуск елемента після видалення кореня: порівнюємо оновлений елемент із двома дочірніми, у разі потреби обмінюємо з більшим із них і повторюємо операцію доти, доки оновлений елемент не опиниться на «своєму» місці. Незалежно від того, додаємо ми чи оновлюємо елемент, на другий крок операції нам потрібно щонайбільше $O(\log M)$ часу.

Таким чином, складність виконання однієї операції типу 1 дорівнює $O(\log(N - M) + \log M) = O(\log N)$.

Операцію типу 2 алгоритм також виконує у кілька етапів:

1. Виводимо верхній елемент купи другого рівня й видаляємо його. На це потрібно $O(\log M)$ часу.
2. Видаляємо верхній (тобто той самий) елемент із купи першого рівня, якій належав видалений елемент купи другого рівня. Щоб не шукати його по всіх M купах першого рівня, разом з елементом купи другого рівня слід зберігати інформацію про місто, з якого він походить. На даний крок витрачаємо $O(\log(N - M))$ часу.

3. Якщо купа першого рівня, з якої ми видалили елемент, не стала порожньою, додаємо новий її верхній елемент до купи другого рівня. На це треба ще $O(\log M)$ часу.

Отже, складність виконання однієї операції типу 2 дорівнює $O(\log(N-M)+2\log M)=O(\log N)$.

Оскільки кожну операцію алгоритм опрацьовує за $O(\log N)$, загальну ефективність алгоритму можна оцінити як $O(N\log N)$. Такий час виконання дозволяє заробити повний бал.

Насамкінець наведемо опис асимптотично менш ефективного алгоритму, який, проте, також набирає повний бал. У його основі лежить інша відома структура даних, яку, на відміну від бінарної купи, учасник цілком міг би вигадати самостійно під час туру. Структура даних базується на розбитті масиву з l елементів на (приблизно) $\sqrt{l}$ частин по (приблизно) $\sqrt{l}$ елементів у кожній і визначенні найменшого/найбільшого числа окремо в кожній з частин. Сам алгоритм дуже подібний до наведеного вище.

Далі через $\lceil x\rceil$ позначатимемо найменше ціле число, не менше за x. Якщо в місті i, $1\leqslant i\leqslant M$, упало c_i камінців, розіб'ємо масив цього міста на частини по $\lceil\sqrt{c_i}\rceil$ елементів у кожній (в останній частині елементів може бути менше). Усього частин (відрізків), на які ми розбили масив, вийде не більше ніж $\left\lceil\frac{c_i}{\lceil\sqrt{c_i}\rceil}\right\rceil\leqslant\left\lceil\frac{c_i}{\sqrt{c_i}}\right\rceil=\lceil\sqrt{c_i}\rceil$. Оскільки $\lceil\sqrt{c_i}\rceil<\sqrt{c_i}+1$, то і кількість частин, і кількість елементів в одній частині масиву мають порядок $O(\sqrt{c_i})$.

Масиви, що відповідають M містам, назвімо масивами першого рівня. Уведемо також масив другого рівня, який у кожен момент часу містить M (або менше) мас найлегших камінців з M (або меншої кількості) міст, у яких є хоча б один камінець. Аналогічне розбиття запровадимо і на цьому масиві: кількість частин та елементів в одній частині масиву складатиме $O(\sqrt{M})$.

Спочатку всі масиви порожні. Кожну операцію типу 1 алгоритм виконуватиме за два кроки:

1. Додаємо камінець (точніше, його вагу) до масиву першого рівня, який відповідає містечку T, куди впав камінець. Перераховуємо мінімум на тому відрізку довжини $\lceil\sqrt{c_T}\rceil$ (або меншої), куди потрапив новий елемент. Також перераховуємо, якщо потрібно, найменше число серед мінімумів на кожному з $\lceil\sqrt{c_T}\rceil$ (або меншої кількості) відрізків. Оскільки на кожному з цих двох кроків додану вагу достатньо порівняти з одним числом — попереднім мінімумом (спочатку на відрізку, а потім із загальним), на таку операцію потрібно $O(1)$ часу.
2. Якщо камінець, що впав, є на даний момент єдиним камінцем у містечку T, додаємо його до масиву другого рівня. Якщо ні і якщо в містечку T, куди впав новий камінець, змінився камінець найменшої ваги, потрібно оновити відповідний елемент масиву другого рівня. Для цього за спеціальним індексом, який будемо зберігати й оновлювати, знаходимо місце, на якому стоїть у масиві другого рівня мінімальний елемент масиву T першого рівня. Оновлюємо (тобто зменшуємо) цей елемент. Незалежно від того, додавали ми чи оновлювали елемент у масиві другого рівня, перераховуємо максимум на відрізку довжини $\lceil\sqrt{M}\rceil$ (або меншої), в якому міститься цей елемент. Потім, якщо пот-

рібно, перераховуємо найбільше число серед максимумів на кожному з $\lceil\sqrt{M}\rceil$ (або меншої кількості) відрізків. На це все в гіршому випадку потрібно $O(\sqrt{M})$ часу.

Таким чином, ефективність виконання однієї операції типу 1 складає $O(\sqrt{M})$.

Операцію типу 2 алгоритм також виконує у кілька етапів:

1. Виводимо (вже підрахований) найбільший елемент масиву другого рівня та видаляємо звідти цей елемент: переставляємо місцями з останнім елементом, зменшуємо лічильник кількості елементів на 1, перераховуємо максимум щонайбільше на двох змінених відрізках, а також загальний максимум. На все потрібно $O(\sqrt{M})$ часу.
2. Видаляємо той самий елемент із масиву першого рівня, якому він належав. Щоб не шукати елемент по всіх M масивах першого рівня, разом з елементом масиву другого рівня слід зберігати інформацію про місто T, з якого він походить. Видалення відбувається в уже знайомий спосіб: переставляємо елемент, що видаляється, з останнім елементом у місті, зменшуємо лічильник кількості елементів на 1, перераховуємо мінімум щонайбільше на двох змінених відрізках, а також загальний мінімум. На даний крок витрачаємо $O(\sqrt{c_T})$, тобто в гіршому випадку $O(\sqrt{N-M})$, часу.
3. Якщо масив першого рівня, з якого ми видалили елемент, не став порожнім, додаємо новий його найменший елемент до масиву другого рівня й перераховуємо максимуми. На це треба $O(1)$ часу.

Отже, складність виконання однієї операції типу 2 дорівнює $O(\sqrt{N-M}+\sqrt{M}) = O(\sqrt{N})$.

Загальну ефективність алгоритму можна оцінити як $O((N-M)\sqrt{N}+M\sqrt{M}) = O(N\sqrt{N})$.

1.4. Олімпійські Авіалінії (Ярослав Твердохліб)

Умова

Імператор Олімпії вирішив сполучити N міст імперії повітряним транспортом. Придворному програмісту було наказано написати програму під назвою «ОлімпБуд», яка спрощує процес побудови карти авіарейсів. Ця програма має виконувати операції двох типів, кожна з яких залежить від трьох параметрів A, B та C:

1. Створити новий рейс з міста A до міста B, переліт по якому триває C хвилин.
2. Розглянути всі авіарейси, які починаються в місті A. Нехай рейс номер i з них закінчується у місті v_i та триває t_i хвилин. Для кожного такого i створити рейс з міста B до міста v_i, який триває $t_i + C$ хвилин.

Після виконання всіх операцій програма має для кожного міста знайти найменший час, потрібний для перельоту до нього зі столиці, можливо з пересадками в інших містах. Вважається, що час посадки, висадки та очікування в аеропорту рівний нулю.

Між двома містами може бути більше від одного рейсу, причому перельоти по них можуть займати різну кількість часу. Можливе існування рейсів, які починаються й закінчуються в одному й тому самому місті. Всі рейси односторонні, тобто наявність прямого рейсу з міста A до міста B ще не гарантує, що існує прямий рейс з міста B до міста A.

Завдання. Напишіть програму `olympair`, яка за послідовністю з M операцій описаних вище типів, виконаних програмою «ОлімпБуд», для кожного міста, окрім столиці, знайде найменший час, потрібний для перельоту зі столиці до цього міста.

Вхідні дані. Перший рядок вхідного файлу `olympair.dat` містить два цілих числа N і M ($2 \leqslant N \leqslant 10^5$, $1 \leqslant M \leqslant 10^5$) — кількість міст в Олімпії та кількість операцій, виконаних програмою «ОлімпБуд». У кожному з наступних M рядків міститься інформація про чергову операцію. Перше число рядка дорівнює `1`, якщо виконувана операція має перший тип, і `2`, якщо другий. Далі записані три цілих числа A, B та C ($1 \leqslant A \leqslant N$, $1 \leqslant B \leqslant N$, $-10^8 \leqslant C \leqslant 10^8$), значення яких описано вище. Міста нумеруються числами від 1 до N; столиця має номер 1. Гарантується, що час перельоту по кожному з рейсів буде *строго додатним*.

Вихідні дані. Вихідний файл `olympair.sol` має містити $N - 1$ рядок. В i-му з них має міститись найменший час у хвилинах, за який можна долетіти зі столиці в місто з номером $i + 1$. Якщо до якогось міста дістатись неможливо, то виведіть у відповідному рядку число `-1`.

Оцінювання. Набір тестів складається з 6 блоків, для яких додатково виконуються такі умови:

- 15 % балів: $N \leqslant 2000$, $M \leqslant 4000$.
- 5 % балів: виконуються операції лише першого типу.
- 15 % балів: $C \geqslant 0$ та якщо деяке місто було містом A в операції другого типу, то далі у вхідному файлі воно більше не бере участь у жодній операції ні як A, ні як B.
- 15 % балів: те саме, що в попередньому блоці, але на C не накладено умову невід'ємності.
- 20 % балів: $C \geqslant 0$.
- 30 % балів: немає додаткових обмежень.

Приклад вхідних та вихідних даних.

olympair.dat	olympair.sol
4 3 1 1 2 10 2 1 3 -9 1 1 3 8	9 8 −1

Розв'язання

Обмеження задачі виключають можливість явно побудувати граф та запустити алгоритм пошуку найкоротшого шляху у ньому, оскільки кількість ребер може рости експоненційно зі збільшенням кількості операцій. Щоб розв'язати задачу, ми побудуємо новий граф, у якому (крім початкових N вершин) будуть $O(M)$ додаткових вершин та $O(N + M)$ ребер. При цьому довжина найкоротшого шляху від вершини 1 до початкових N вершин буде тією ж, що й у графі, побудованому після виконання всіх операцій з умови (назвімо цей граф початковим).

Кожній вершині початкового графа поставимо у відповідність деяку множину вершин нового графа. Перед початком виконання програми для кожної вершини початкового графа v створимо у новому графі дві вершини in_v та $active_v$. Кожну операцію будемо виконувати таким чином:

1. При виконанні операції першого типу з параметрами A, B та C додамо ребро з вершини $active_A$ у вершину in_B довжини C.
2. При виконанні операції другого типу з параметрами A, B та C додамо ребро з вершини $active_B$ у вершину $active_A$ довжини C, після чого створимо нову вершину $active'_A$. Додамо ребро довжини 0 з цієї нової вершини в $active_A$, після чого оголосимо вершину $active'_A$ новою $active_A$ (стара вершина залишиться у графі «нейтральною»). Цими діями ми імітуємо копіювання списку суміжності вершини A до списку вершини B з додаванням величини C до всіх ребер. До того ж при додаванні нових ребер з початком у вершині A вони не додаватимуться до списку вершини B, оскільки зі старої вершини $active_A$ немає шляху до нової $active'_A$, який не пройшов би через вершину типу in.

Після виконання всіх операцій для кожної вершини v початкового графа додамо у новому графі ребро з in_v в $active_v$ довжини 0. Після цього треба виконати алгоритм пошуку найкоротшого шляху з вершини in_1 до всіх інших вершин типу in. Якщо для цих цілей використовувати алгоритм Левіта або алгоритм Беллмана — Форда, то розв'язок набере 15 балів. Алгоритм Дейкстри при цьому набирає 55 балів, оскільки у графі можуть бути наявні від'ємні ребра. Щоб позбутися від'ємних ребер, потрібно скористатися тим, що у початковому графі, отриманому після виконання операцій з умови, всі ребра є додатними.

Розглянемо будь-яку вершину нового графа, що не належить до вершин типу in. Нехай найменше ребро, яке входить до неї, має довжину d. Тоді ми можемо відняти від довжин усіх ребер, що входять у неї, величину d, після чого додати цю ж величину до всіх ребер, що виходять з даної вершини. Очевидно, що будь-який шлях, який проходить через дану вершину і не закінчується у ній, не змінить своєї довжини. При цьому в дану вершину вже не входить жодне ребро від'ємної довжини. Отже, за допомогою таких операцій ми можемо видаляти з графа одні від'ємні ребра та, можливо, додавати до нього інші від'ємні ребра.

Оскільки за умовою всі ребра, які входять до вершин типу in, є додатними, а також у графі не існує циклів, які не містять жодної вершини типу in, то через скінченний час у графі залишаться лише невід'ємні ребра. Твердження про відсутність циклів без вершин типу in випливає з того, що при проходженні по будь-якому ребру, яке не входить у вершину in, ми зможемо потрапити лише у ті ребра, які були додані до графа раніше. Оскільки на початку у графі не було ребер, то по циклу ми не пройдемо.

Отже, видалимо всі вершини типу in із графа і топологічно відсортуємо отриманий ациклічний граф. Після цього повернемо усі вершини назад у граф та виконаємо описані вище операції зміни довжин ребер. Оскільки граф без вершин типу in є ациклічним, після виконання всіх операцій у ньому не буде від'ємних ребер, а тому можна використати алгоритм Дейкстри.

Складність запропонованого алгоритму — $O((N + M)\log(N + M))$.

Завдання другого туру

2.1. Конструктор (Данило Мисак)

Умова

На свій перший день народження Меґґі Сімпсон, персонаж мультсеріалу «Сімпсони», отримала в подарунок конструктор: ігровий набір, що складається з паличок різної довжини. Кінці паличок можна скріплювати, причому з'єднані таким чином палички можуть утворювали довільний ненульовий кут, крім розгорнутого $(180°)$. Меґґі хоче скласти опуклий многокутник, використавши якомога більшу кількість паличок із конструктора як сторони цього багатокутника.

Завдання. Напишіть програму `set`, що за розмірами паличок у конструкторі визначить, чи вдасться Меґґі скласти із паличок опуклий многокутник, і якщо вдасться, то визначить, яку найбільшу кількість паличок вона зможе для цього використати.

Вхідні дані. У першому рядку вхідного файлу `set.dat` указано кількість N паличок у наборі, $2 \leqslant N \leqslant 10^5$. У другому рядку записано N натуральних чисел, менших за 10^9 (не обов'язково попарно різних) — довжини паличок.

Вихідні дані. Вихідний файл `set.sol` повинен містити єдине число — найбільшу кількість паличок з набору, із яких можна скласти опуклий многокутник, або число `0`, якщо скласти опуклий многокутник не вдасться.

Оцінювання. Набір тестів складається з 3 блоків, для яких додатково виконуються такі умови:

- 40 % балів: $2 \leqslant N \leqslant 15$.
- 30 % балів: $15 < N \leqslant 3000$.
- 30 % балів: $3000 < N \leqslant 10^5$.

Крім того, у тестах на 25 % балів правильна відповідь не перевищує 4.

Приклади вхідних та вихідних даних.

set.dat	set.sol
4 5 1000 5 5	3
3 1 2 3	0

Розв'язання

Під опуклим многокутником будемо розуміти строго опуклий многокутник, тобто такий, усі кути якого строго менші за $180°$. Доведемо спершу допоміжне твердження, що є узагальненням нерівності трикутника для випадку опуклого N-кутника.

Лема. Нехай задано $N \geqslant 1$ відрізків додатних довжин $d_1, d_2, \ldots, d_N$. Із них можна скласти опуклий многокутник тоді й лише тоді, коли довжина найбільшого відрізка менша за суму довжин решти $N-1$ відрізків, тобто коли $2\max\{d_1, d_2, \ldots, d_N\} < d_1 + d_2 + \ldots + d_N$.

Доведення леми. Нехай із заданих відрізків вдалося скласти опуклий N-кутник $A_1A_2\ldots A_N$. Хай, без утрати загальності, A_1A_N — найдовший з усіх відрізків. Послідовно використовуючи $N-2$ рази нерівність трикутника, матимемо

$$A_1A_2 + A_2A_3 + A_3A_4 + \ldots + A_{N-1}A_N > A_1A_3 + A_3A_4 + \ldots + A_{N-1}A_N > \\ > A_1A_4 + \ldots + A_{N-1}A_N > \ldots > A_1A_N.$$

Таким чином, нерівність з умови леми справджується.

Доведімо тепер зворотне твердження: якщо нерівність для відрізків довжин $d_1, d_2, \ldots, d_N$ справджується, то з них можна скласти опуклий багатокутник. Легко бачити, що при $N = 1$ та $N = 2$ нерівність виконуватися не може. Отже, $N \geqslant 3$. Не втрачаючи загальності, припустимо, що найдовшим із відрізків є d_N, тобто $d_i \leqslant d_N$, $1 \leqslant i \leqslant N$.

Нехай на площині зафіксували коло радіуса $R \geqslant d_N/2$ із центром у точці O, а також деяку декартову систему координат, причому коло дотикається до прямої $x = 0$ (вісь ординат) у точці $(0, 0)$ (початок координат). Визначимо на колі точки $A_1, A_2, \ldots, A_N$ у такий спосіб:

- A_1 — точка $(0, 0)$;
- A_2 — перша за рухом годинникової стрілки точка після A_1 на колі така, що $A_1A_2 = d_1$;
- A_3 — перша за рухом годинникової стрілки точка після A_2 на колі така, що $A_2A_3 = d_2$;
  …
- A_N — перша за рухом годинникової стрілки точка після A_{N-1} на колі така, що $A_{N-1}A_N = d_{N-1}$.

Оскільки за побудовою діаметр кола не менший за кожен із відрізків $d_1, d_2, \ldots, d_{N-1}$, конструкцію визначено коректно. Приклад розташування точок зображено на рис. 1.

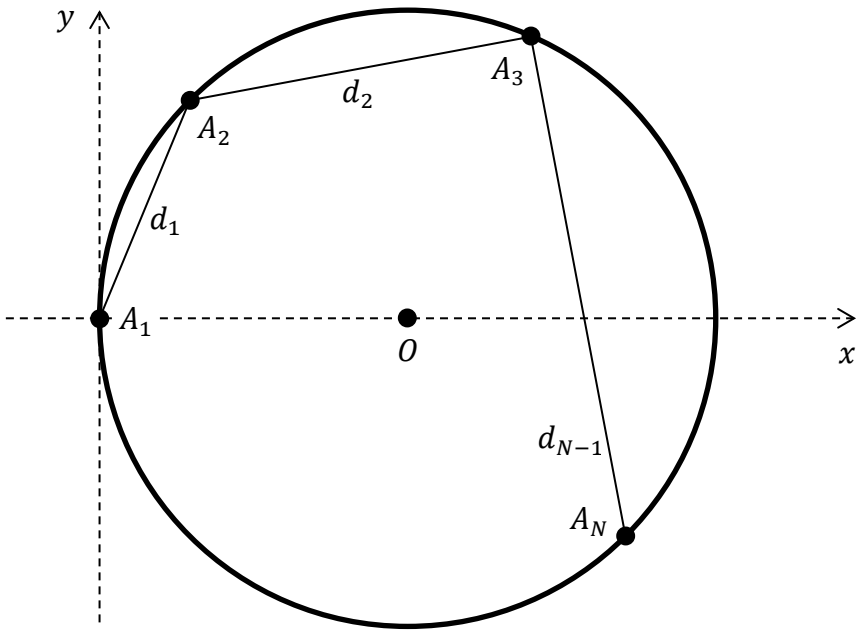


Рис. 1

Доведемо, що, взявши коло достатньо великого радіуса, ми зможемо забезпечити одночасне виконання таких двох умов:

1) $\angle A_1OA_2 + \angle A_2OA_3 + \ldots + \angle A_{N-1}OA_N < 180°$ (звідси, зокрема, випливатиме, що ламана $A_1A_2\ldots A_N$ не має самоперетинів);
2) $A_1A_N > d_N$.

Щоб довести цей факт, розглянемо довільні дві послідовні хорди $A_{k-1}A_k$ та A_kA_{k+1}, $2 \leqslant k \leqslant N-1$ (рис. 2). Нехай $\angle A_{k-1}OA_k = 2\alpha$, а $\angle A_kOA_{k+1} = 2\beta$.

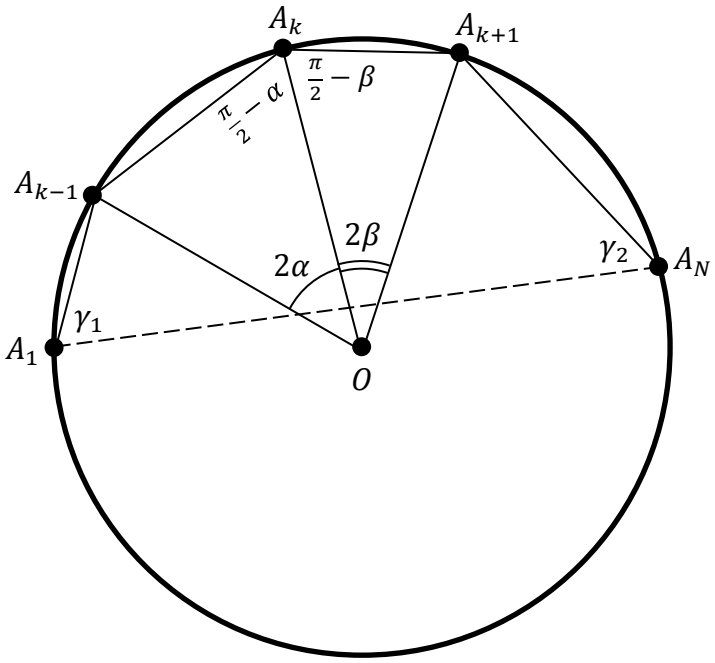


Рис. 2

Оскільки $OA_{k-1} = OA_k = OA_{k+1} = R$, трикутники $A_{k-1}OA_k$ та A_kOA_{k+1} рівнобедрені, а тому

$$\angle A_{k-1}A_kO = \frac{\pi}{2} - \alpha, \quad \angle A_{k+1}A_kO = \frac{\pi}{2} - \beta,$$
$$\angle A_{k-1}A_kA_{k+1} = \angle A_{k-1}A_kO + \angle A_{k+1}A_kO = \pi - \alpha - \beta.$$

Як відомо, $A_{k-1}A_k = 2R\sin\alpha$, звідки

$$\alpha = \arcsin\frac{A_{k-1}A_k}{2R} = \arcsin\frac{d_{k-1}}{2R} \to 0,\ R \to \infty.$$

Аналогічно $\beta \to 0,\ R \to \infty$. Тому

$$\angle A_1OA_2 + \angle A_2OA_3 + \ldots + \angle A_{N-1}OA_N \to 0,\ R \to \infty;$$
$$\angle A_{k-1}A_kA_{k+1} = \pi - \alpha - \beta \to \pi,\ R \to \infty.$$

Таким чином, виконання умови 1) при достатньо великих значеннях R доведено. Щоб довести виконання умови 2), розглянемо многокутник $A_1A_2\ldots A_N$ при значеннях R, що задовольняють умову 1). Сума кутів N-кутника складає, як відомо, $(N-2)\pi$. Якщо позначити кути $A_NA_1A_2$ і $A_1A_NA_{N-1}$ через γ_1 і γ_2 відповідно, матимемо

$$\gamma_1 + \gamma_2 = (N-2)\pi - \sum_{k=2}^{N-1} \angle A_{k-1}A_kA_{k+1} \to 0,\ R \to \infty \Rightarrow$$
$$\Rightarrow \max\{\gamma_1, \gamma_2\} \to 0,\ R \to \infty.$$

Позначимо через φ_i, $1 \leqslant i \leqslant N-1$, кут, який утворюють прямі A_iA_{i+1} та A_1A_N. Якщо прямі паралельні, цей кут дорівнює 0. Якщо точка перетину прямих лежить на промені A_NA_1, то кут φ_i не перевищує γ_1 (дорівнює γ_1, коли $i = 1$, та менший за γ_1, коли $i > 1$). А якщо точка перетину прямих лежить на промені A_1A_N, то кут φ_i не перевищує γ_2 (дорівнює γ_2, коли $i = N-1$, та менший за γ_2, коли $i < N-1$). Таким чином, $\varphi_i \leqslant \max\{\gamma_1, \gamma_2\}$, $1 \leqslant i \leqslant N-1$. А тому

$$A_1A_N = \sum_{i=1}^{N-1} A_iA_{i+1}\cos\varphi_i \geqslant \sum_{i=1}^{N-1} A_iA_{i+1}\cos\max\{\gamma_1, \gamma_2\} \to \sum_{i=1}^{N-1} A_iA_{i+1} = \sum_{i=1}^{N-1} d_i > d_N,\ R \to \infty.$$

Отже, виконання умови 2) при достатньо великих R також доведено.

Візьмемо деяке достатньо велике число $R_1 > d_N/2$, для якого умови 1) і 2) виконані, та почнемо неперервно зменшувати радіус кола. При цьому точки $A_2, A_3, \dots, A_N$ «ковзатимуть» по колу в напрямку годинникової стрілки. Припинимо процес «стискання» кола, щойно справдиться хоча б одне з двох тверджень:

- радіус кола стане рівним $d_N/2$ або
- точка A_N збігатиметься з точкою A_1.

Ураховуючи природу першої умови зупинки, рано чи пізно ми обов'язково припинимо процес. Позначимо радіус кола, на якому відбулася зупинка, через R_0. Нехай функція $f\colon [R_0, R_1] \to [0, +\infty)$ визначає відстань між точками A_1 та A_N при заданому радіусі $R \in [R_0, R_1]$. Незалежно від того, яка з умов послужила причиною зупинки, виконується нерівність $f(R_0) \leqslant d_N$. Справді, якщо $R_0 = d_N/2$, то відстань між будь-якими двома точками на колі не перевищує $2R_0 = d_N$. А якщо $A_N = A_1$, то взагалі $f(R_0) = 0$. У той же час із визначення числа R_1 маємо $f(R_1) > d_N$. Оскільки функція f неперервна, з цих співвідношень випливає, що існує число $R \in [R_0, R_1]$, для якого $f(R) = d_N$. Зважаючи на те, що це значення R ми «пройшли» до того, як точка A_N уперше збіглася з A_1, можемо стверджувати, що при відповідному радіусі кола замкнена ламана $A_1A_2\dots A_N$ — уписаний (а отже, опуклий) N-кутник без самоперетинів із довжинами сторін, що дорівнюють $d_1, d_2, \dots, d_N$. Лему доведено. ■

Повернімося тепер до самої задачі — нехай N знову позначає кількість заданих у вхідному файлі довжин, а через M позначимо найбільше значення, якого може набувати довжина одного відрізка (в даному випадку $M = 10^9 - 1$).

Ідейно найпростіший спосіб розв'язати задачу — перебрати всі підмножини заданої множини відрізків і, скориставшись твердженням леми, для кожної підмножини визначити, чи задає вона сторони опуклого багатокутника. Час виконання такого алгоритму дорівнює $O(N \cdot 2^N)$; або $O(N^2 \cdot 2^N)$, якщо перевіряти виконання нерівності многокутника не лише для найбільшої, а для всіх його сторін. Перебірні алгоритми набирають 40 % від загальної кількості балів.

Задачу можна розв'язувати так. Спершу відсортуємо заданий масив чисел. Позначимо послідовність чисел, утворену в результаті, як $d_1 \leqslant d_2 \leqslant \ldots \leqslant d_N$. Підрахуємо суму $S_N = d_1 + d_2 + \ldots + d_N$. Якщо подвоєне найбільше число в масиві d_N є меншим за цю суму (тобто задовольняє нерівність многокутника), то відповідь — N. Інакше незалежно від того, які відрізки стануть у підсумку сторонами многокутника, відрізка довжини d_N бути серед них не може, адже нерівність многокутника це число задовольнити ніяк не зможе. Тому відкинемо його та, перерахувавши суму $S_{N-1} = d_1 + d_2 + \ldots + d_{N-1} = S_N - d_N$, повторимо ті самі дії для числа d_{N-1}: якщо $2d_{N-1} < S_{N-1}$, то з відрізків довжин $d_1, d_2, \ldots, d_{N-1}$ можна утворити опуклий многокутник, а тому відповідь — число $N - 1$. А якщо $2d_{N-1} \geqslant S_{N-1}$, то відрізок довжини d_{N-1} не може бути стороною многокутника, тож слід відкинути число d_{N-1} і перерахувати суму $S_{N-2} = S_{N-1} - d_{N-1}$. Такі операції повторюємо доти, доки не знайдемо відповідь або не відкинемо всі відрізки. Останнє означатиме, що із заданих відрізків скласти опуклий многокутник неможливо.

Під час підрахунку суми чисел треба зважити на те, що вона може виявитися досить великою і, на відміну від самих чисел, не вміститися у чотирибайтову змінну. Тому слід або використати відповідний тип даних, або припиняти підрахунок суми, коли стає зрозуміло, що вона більша за $2M$, а отже, перевищує подвоєну довжину будь-якого з заданих відрізків.

Складність наведеного алгоритму лінійна, якщо знехтувати сортуванням масиву на початку виконання. Таким чином, залежно від реалізації сортування можна досягти ефективності $O(N^2)$ і набрати 70 % балів або вкластися в $O(N \log N)$ і заробити повний бал.

Інший ефективний спосіб розв'язати задачу ґрунтується на твердженні такої леми.

Лема 2. Перед тим як вищенаведений алгоритм завершить свою роботу, він відкине щонайбільше $[\log_2 M] + 2$ відрізки (де $[\log_2 M]$ позначає найбільше ціле число, що не перевищує $\log_2 M$).

Доведення леми 2. Нехай алгоритм відкинув l відрізків завдовжки $d_N, d_{N-1}, \ldots, d_{N-l+1}$, $l \leqslant N$.

Якщо $l < N$, то маємо $d_1 \geqslant 1, d_2 \geqslant 1, \ldots, d_{N-l} \geqslant 1$, а далі, враховуючи, що наступні числа алгоритм відкинув,

$$d_{N-l+1} \geqslant d_1 + d_2 + \ldots + d_{N-l} \geqslant N - l,$$
$$d_{N-l+2} \geqslant d_1 + d_2 + \ldots + d_{N-l} + d_{N-l+1} \geqslant (N-l) + (N-l) = 2(N-l),$$
$$d_{N-l+3} \geqslant d_1 + \ldots + d_{N-l} + d_{N-l+1} + d_{N-l+2} \geqslant (N-l) + (N-l) + 2(N-l) = 2^2(N-l),$$
$$\ldots$$
$$d_N \geqslant (1 + 1 + 2 + 2^2 + \ldots + 2^{l-2})(N-l) = 2^{l-1}(N-l).$$

Тому $M \geqslant d_N \geqslant 2^{l-1}(N-l) \geqslant 2^{l-1}$, звідки $l \leqslant [\log_2 M] + 1$.

Якщо $l = N$, то маємо $d_1 \geqslant 1, d_2 \geqslant 1$, а далі

$$d_3 \geqslant d_1 + d_2 \geqslant 2,$$
$$d_4 \geqslant d_1 + d_2 + d_3 \geqslant 1 + 1 + 2 = 2^2,$$
$$d_5 \geqslant d_1 + d_2 + d_3 + d_4 \geqslant 1 + 1 + 2 + 2^2 = 2^3,$$
$$\ldots$$

$$d_N \geqslant 1 + 1 + 2 + 2^2 + \ldots + 2^{N-3} = 2^{N-2}.$$

Тому $M \geqslant d_N \geqslant 2^{N-2} = 2^{l-2}$, звідки $l \leqslant [\log_2 M] + 2$.

Лему доведено. ■

Таким чином, алгоритм, який не сортуватиме масив, а просто щоразу після відкидання елемента заново шукатиме найбільше число, буде мати час виконання порядку $O(N \log M)$. Така ефективність теж оцінюється повним балом.

Однак твердження леми 2 дозволяє побудувати навіть швидший метод розв'язання задачі, який учасникам зовсім не обов'язково було втілювати, щоб отримати повний бал, але який ми наведемо із теоретичного інтересу.

Не будемо сортувати масив, а натомість за лінійний час упорядкуємо елементи масиву таким чином, щоб на останніх $m = [\log_2 M] + 3$ місцях стояли (у довільному порядку) саме ті числа, які б стояли на цих m місцях, якби масив було відсортовано в порядку неспадання. За лінійний час цю операцію дозволяють виконати так звані алгоритми вибору порядкових статистик (selection algorithms), деталі реалізації яких можна знайти в інтернеті. Після цього елементи на останніх m місцях слід відсортувати (за $O(m \log m)$ часу), а далі діяти так, начебто відсортовано увесь масив. Оскільки згідно з лемою 2 алгоритм відкине щонайбільше $m - 1$ елемент, то до невідсортованої частини масиву він просто не дійде. Сумарний час виконання програми складатиме $O(N + \log M \log \log M)$, що на практиці означає лінійну (за кількістю чисел у вхідному файлі) ефективність.

Насамкінець кілька слів стосовно того, наскільки реально було написати розв'язок задачі учаснику, не знайомому до олімпіади із твердженням леми про нерівність многокутника. Думка автора така: виходячи з того, наскільки широко відомою є нерівність трикутника і наскільки просто обґрунтовується твердження про нерівність многокутника в один бік, учасник, пробуючи розв'язати задачу, цілком міг щонайменше висунути гіпотезу про нерівність многокутника. З огляду на засади проведення олімпіади, перевірка гіпотези не вимагала від учасника побудови доведення в інший бік: достатньо було написати програму, що ґрунтується на гіпотезі, здати її й подивитися на результат.

2.2. Ковпачок (Данило Мисак)

Умова

Нудьгуючи на одному з уроків, відмінник Петро П'яточкін придумав собі розвагу. Він замалював ручкою деякі клітинки прямокутного аркуша, вирваного з зошита, зняв із ручки ковпачок та поставив його на одну з зафарбованих клітин. Далі Петрик послідовно переставляє ковпачок з одної замальованої клітинки на іншу замальовану клітинку, яка міститься в тому ж рядку або в тому ж стовпчику, що і попередня. Петрик вибрав деяку зафарбовану клітину й хоче перемістити туди ковпачок із початкової клітини за якомога меншу кількість ходів.

Завдання. Напишіть програму `cap`, що за даними про розміри аркуша паперу, конфігурацію зафарбованих клітин, розміщення ковпачка та цільової клітини знайде найменшу можливу кількість переставлянь, за які Петрик зможе перемістити ковпачок з початкової клітини до цільової, керуючись придуманими ним правилами.

Вхідні дані. У першому рядку вхідного файлу `cap.dat` записано два цілих числа: кількість рядків N і кількість стовпчиків M клітинок, із яких складається аркуш, $2 \leqslant M \leqslant N \leqslant 1000$. Кожен із наступних N рядків містить по M символів:

- x (маленька літера латинського алфавіту) — зафарбована клітина.
- . (крапка) — порожня клітина.
- o (маленька літера латинського алфавіту) — початкова клітина.
- + (плюс) — цільова клітина.

У вхідних даних задано рівно одну початкову та рівно одну цільову клітину.

Вихідні дані. Вихідний файл `cap.sol` повинен містити єдине число — найменшу кількість переміщень ковпачка, які потрібно зробити Петрику задля досягнення мети. Якщо ж відповідно до заданих правил не можна досягти цільової клітини, то слід вивести `-1`.

Оцінювання. Набір тестів складається з 3 блоків, для яких додатково виконуються такі умови:

- 30 % балів: $2 \leqslant M \leqslant N \leqslant 10$.
- 30 % балів: $10 < M \leqslant N \leqslant 100$.
- 40 % балів: $100 < M \leqslant N \leqslant 1000$.

Приклади вхідних та вихідних даних.

cap.dat	cap.sol
3 2 x+ xx o.	2
4 4 .o.x x.x. .x.x x.+.	-1

Розв'язання

Аркуш паперу можна подати як граф, вершинами якого є зафарбовані клітинки, а ребро між двома вершинами проведене тоді й лише тоді, коли відповідні їм клітинки містяться в одному рядку або в одному стовпчику. Природний підхід до розв'язання задачі — здійснити пошук у ширину на такому графі, починаючи з вершини, що відповідає початковій клітинці A, та знайти довжину найкоротшого шляху з неї до цільової клітини B. Однак час виконання такого алгоритму буде занадто великим, адже пошук у ширину працює $O(V + E)$ часу, де V — кількість вершин, а E — кількість ребер у графі. У нашому випадку кількість вершин не перевищує NM, але кількість ребер може досягати $NM(N + M - 2)/2$ (і дорівнює цьому числу, якщо зафарбо-

вано всі NM клітин). Тож час виконання програми складе $O(NM(N+M))$. Це дасть 60 % від загальної кількості балів.

Суттєво пришвидшити алгоритм можна, зауваживши таке: якщо на деякому кроці пошуку в ширину ми розглядаємо клітинку C, а раніше вже було розглянуто хоча б одну клітинку D, що стоїть у тому ж рядку, що й клітинка C, то немає сенсу розглядати ребра графа, які сполучають клітину C з іншими клітинками в її рядку. Справді, припустимо, що D — перша з клітинок у даному рядку, пройдена алгоритмом. Якщо деяке ребро (C, E) сполучає C з клітинкою E, яка міститься в тому ж рядку, що й C, то або $E = D$, або у графі наявне ребро (D, E) (бо D та E містяться в одному рядку). А отже, вершину E вже було додано в чергу або навіть пройдено раніше — і розгляд ребра (C, E) не дасть жодного результату. Таким чином, можна не розглядати багато «зайвих» ребер — достатньо пам'ятати, клітини в яких рядках алгоритм уже встиг пройти.

Звичайно, ті самі міркування справедливі не тільки для рядків, але й для стовпчиків таблиці. Тому в кожному рядку буде розглянуто щонайбільше $M-1$ ребро, а в кожному стовпчику — не більше ніж $N-1$ (ребра, що виходять із деякої однієї клітини даного рядка або стовпчика). Усього алгоритм розгляне $O(NM)$ ребер, а тому й час виконання дорівнюватиме $O(NM)$. Це дасть повний бал.

Задачу можна розв'язувати дещо інакше. Знов побудуємо граф, що відповідає таблиці у вхідному файлі, але тепер зробимо це в інший спосіб. Новий граф матиме $N+M$ вершин і не більше ніж NM ребер. Нехай N вершин $a_1, a_2, \dots, a_N$ графа відповідають рядкам $1, 2, \dots, N$, а інші M вершин $b_1, b_2, \dots, b_M$ відповідають стовпчикам $1, 2, \dots, M$ таблиці. Ребро між вершинами a_i та b_j проведене тоді й лише тоді, коли на перетині i-го рядка та j-го стовпчика стоїть зафарбована клітинка. Інших ребер у графі немає.

Поставимо у відповідність ходу з клітинки (i, j_1) у клітинку (i, j_2) таблиці переміщення з вершини a_i у вершину b_{j_2} графа, а ходу з клітинки (i_1, j) у клітинку (i_2, j) — переміщення з вершини b_j у вершину a_{i_2}. Оскільки в оптимальному (найкоротшому) маршруті між клітинками A і B не може бути двох послідовних ходів у межах одного рядка або двох послідовних ходів у межах одного стовпчика (інакше їх можна було б замінити на один еквівалентний хід), то оптимальному маршруту між A і B відповідає деякий шлях по ребрах графа. При цьому якщо $A = (i_1, j_1)$, а $B = (i_2, j_2)$, то шлях починається або у вершині a_{i_1}, або у вершині b_{j_1}, а закінчується ребром, що сполучає вершини a_{i_2} та b_{j_2} (або в напрямку від вершини a_{i_2} до b_{j_2}, або в протилежному). І навпаки: кожному такому шляху по ребрах графа відповідає маршрут тієї ж довжини між клітинками A і B.

Таким чином, задача зводиться до пошуку довжини найкоротшого шляху, що починається в одній з вершин a_{i_1} або b_{j_1}, а закінчується ребром, що сполучає вершини a_{i_2} та b_{j_2} графа. Якщо позначити через $d(x, y)$ довжину найкоротшого шляху між вершинами x та y, шукана величина дорівнює

$$\min\{d(a_{i_1}, a_{i_2}), d(a_{i_1}, b_{j_2}), d(b_{j_1}, a_{i_2}), d(b_{j_1}, b_{j_2})\} + 1.$$

А якщо між відповідними парами вершин шляхів не існує, то нема й маршруту між клітинками A і B.

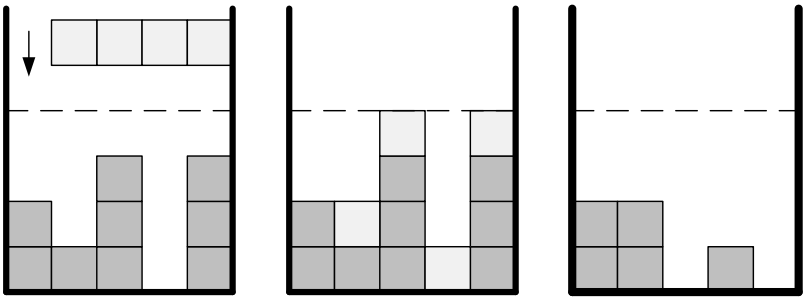
Зауважимо, що відстані між парами вершин можна підрахувати з допомогою двох пошуків у ширину (один з вершини a_{i_1}, а інший — із b_{j_1}). Утім, для знаходження відповіді можна обійтися й одним пошуком у ширину. Для цього додамо до графа фіктивну вершину c й проведемо з неї два ребра: у вершини a_{i_1} та b_{j_1}. Тоді шукане значення — це довжина найкоротшого шляху з вершини c в одну з вершин a_{i_2} або b_{j_2}, тобто $\min\{d(c, a_{i_2}), d(c, b_{j_2})\}$.

Отже, маємо альтернативний алгоритм розв'язання задачі із таким же порядком часу виконання $O(NM)$.

2.3. Тетрис (Сергій Нагін)

Умова

Тінкі-Вінкі грає у модулярний тетрис. Поле складається з N стовпчиків, у кожному з яких може міститися від нуля до трьох кубиків. Після того, як у стовпчику опиняється четвертий кубик, усі чотири кубики зникають. За один хід гравець може вибрати довільну кількість від 1 до N послідовних стовпчиків, на які впаде по одному кубику, як зображено на рисунку. Тінкі-Вінкі хоче, починаючи з наявної конфігурації кубиків на полі, якомога скоріше досягти певної цільової конфігурації.



Завдання. Напишіть програму `tetris`, яка за інформацією про кількість стовпчиків на полі, початкову та цільову конфігурації кубиків визначить найменшу кількість ходів, які має зробити Тінкі-Вінкі.

Вхідні дані. У першому рядку вхідного файлу `tetris.dat` міститься ціле число N ($1 \leqslant N \leqslant 1000$) — кількість стовпчиків на полі тетриса. У другому рядку записано N цілих чисел від 0 до 3, які задають початкову конфігурацію кубиків на полі. У третьому рядку записано N цілих чисел від 0 до 3, які задають кінцеву конфігурацію кубиків. Початкова та кінцева конфігурації не збігаються.

Вихідні дані. Єдиний рядок вихідного файлу `tetris.sol` повинен містити єдине ціле число — мінімальну можливу кількість ходів Тінкі-Вінкі для досягнення цільової конфігурації.

Оцінювання. Набір тестів складається з 4 блоків, для яких додатково виконуються такі умови:

- 30 % балів: $N \leqslant 8$.

- 20 % балів: $N \leqslant 1000$ та правильна відповідь не перевищує 10.
- 25 % балів: $N \leqslant 100$.
- 25 % балів: немає додаткових обмежень.

Приклади вхідних та вихідних даних.

tetris.dat	tetris.sol
5 2 1 3 0 3 2 2 0 1 0	1
4 0 1 2 3 3 2 1 0	5

Розв'язання

Позначимо початкову конфігурацію як A, а кінцеву — як B.

Розв'язок за $O(4^N \cdot N^3)$

Нехай кожній конфігурації кубиків відповідає вершина графа, а між конфігураціями, між якими можна зробити перехід за один хід, є орієнтоване ребро. Запустимо стандартний пошук у ширину від вершини, що відповідає конфігурації A, до вершини, що відповідає конфігурації B. Отримана найкоротша відстань і буде відповіддю.

Розв'язок за $O(N^3)$

Зрозуміло, що зміна порядку операцій не змінює результуючої конфігурації. Впорядкуємо ходи гравця — відрізки кубиків — за лівим краєм.

Будемо використовувати динамічне програмування: нехай `dp(pref, opened)` — мінімальна кількість операцій, яка потрібна, щоб прирівняти конфігурації в перших `pref` символах за умови, що на теперішньому кроці буде відкрито `opened` відрізків від деяких попередніх операцій. Перехід можна робити таким чином:

- Закінчимо деяку кількість попередніх операцій.
- Почнемо деякі інші операції, ліві кінці яких будуть збігатися з теперішньою позицією.
- Після перших двох пунктів чи́сла в обох конфігураціях на теперішній позиції повинні збігатися.

Більш формально динаміку можна записати так (`deleted` — кількість закінчених операцій; `new` — кількість розпочатих операцій):

- `dp(pref, opened)+new –> dp(pref+1, opened–deleted+new)`:
  - `deleted` $\leqslant$ `opened`;
  - `new` $\leqslant 3$;
  - `(A[pref+1]+opened–deleted+new) mod 4 = B[pref+1]`.
- `dp(0, 0)=0`.

Розв'язок за $O(N^2)$

Додамо оптимізацію до попереднього розв'язку — обмеження `deleted` $\leqslant 3$. Зрозуміло, що на жодному кроці не треба скасовувати більше ніж три операції, інакше їх не треба було б починати.

Неасимптотичні оптимізації

Замість операцій `mod 4` (Pascal) та `%4` (C++) можна використовувати операції `and 3` (Pascal) та `&3` (C++). Дана оптимізація може скоротити час виконання в 10 разів.

Також можна помітити, що ми або скасовуємо деякі операції, або створюємо нові, але не робимо цього одночасно. Скориставшись цим, можна зменшити час виконання в 4 рази.

Також можна перебирати тільки досяжні стани динаміки, це теж може скоротити час виконання в 4 рази.

2.4. Камелот (Роман Єдемський)

Умова

Король Артур вирішив зібрати лицарів задля термінової військової наради. На землях, якими правив Артур, розташовувалися оборонні фортеці, побудовані у формі кола, — такі вважалися найбільш неприступними. Певні фортеці були розташовані всередині інших, що забезпечувало їм іще більшу захищеність. Як тільки лицарі отримують наказ від Артура, вони вирушають у дорогу зі свого маєтку в супроводі охорони. Якщо шлях лицаря проходить ззовні фортеці всередину чи навпаки, лицар повинен заплатити данину за перетин брами. Кожна фортеця встановлює розмір данини за прохід однієї людини, тобто лицарю потрібно заплатити за себе та своїх охоронців.

Артур бажає вибрати таке місце проведення наради, щоб мінімізувати сумарні витрати лицарів, адже вони будуть відшкодовані з державної скарбниці. Місце проведення наради може розташовуватися будь-де на землях Артура, крім границь фортець. Крім того, Артур може зменшити свої витрати, скасувавши данину не більш ніж у K вибраних ним фортецях. На свій шлях з Камелота до місця наради Король нічого не витрачає, а всі лицарі завжди вибирають найдешевший маршрут.

Завдання. Напишіть програму `camelot`, яка за інформацією про карту земель Артура, розміри данини кожної з фортець, кількість охоронців у лицарів та кількість фортець, де данина може бути скасована за наказом Короля, знайде мінімальну кількість грошей, яку він має витратити, щоби провести нараду. Карту земель може бути подано як площину із колами, що задають фортеці, і точками, що задають маєтки лицарів.

Вхідні дані. Перший рядок вхідного файлу `camelot.dat` містить три цілих числа N, M і K ($2 \leqslant N \leqslant 35\,000,\ 1 \leqslant M \leqslant 35\,000,\ 0 \leqslant K \leqslant N$), де N — кількість фортець на землях Артура, M — кількість лицарів, викликаних на нараду, а K — кількість фортець, у яких Артур може ска-

сувати данину. Наступні N рядків задають фортеці та містять по чотири цілих числа x, y, R, C ($-10^6 \leqslant x \leqslant 10^6$, $-10^6 \leqslant y \leqslant 10^6$, $1 \leqslant R \leqslant 2 \cdot 10^6$, $1 \leqslant C \leqslant 10^5$), де (x, y) — координати центра фортеці на мапі, R — радіус кола, що задає фортецю, а C — розмір данини з людини. Наступні M рядків задають інформацію про лицарів та містять по три цілих числа x, y, L ($-10^6 \leqslant x \leqslant 10^6$, $-10^6 \leqslant y \leqslant 10^6$, $1 \leqslant L \leqslant 10^5$), де (x, y) — координати маєтку лицаря на мапі, а L — кількість охоронців, що подорожують разом із лицарем, включаючи самого лицаря. Вхідні дані гарантують:

1. Жодні два кола, що задають фортеці, не мають спільних точок.
2. Жодні дві точки, що задають маєтки лицарів, не збігаються та не лежать на колах.

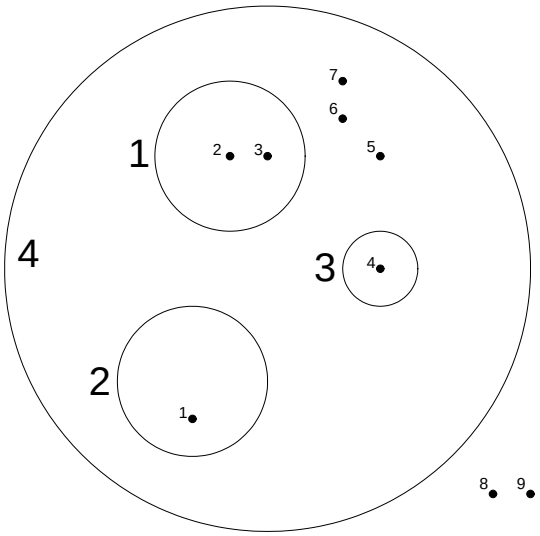
Вихідні дані. Єдиний рядок вихідного файлу `camelot.sol` повинен містити ціле число — мінімальну суму грошей, які Король Артур має витратити, щоб зібрати лицарів.

Оцінювання. Набір тестів складається з 6 блоків, для яких додатково виконуються такі умови:

- 10 % балів: $N \leqslant 1000$, $M \leqslant 1000$ і немає фортець, що розташовані на території інших фортець.
- 10 % балів: $N \leqslant 35\,000$, $M \leqslant 35\,000$ і немає фортець, що розташовані на території інших фортець.
- 10 % балів: $N \leqslant 1000$, $M \leqslant 1000$, $K = 0$.
- 20 % балів: $N \leqslant 35\,000$, $M \leqslant 35\,000$, $K = 0$.
- 15 % балів: $N \leqslant 1000$, $M \leqslant 1000$.
- 35 % балів: немає додаткових обмежень.

Приклад вхідних та вихідних даних.

camelot.dat	camelot.sol
4 9 1 6 10 2 1 5 4 2 1 10 7 1 200 7 7 7 1 5 3 10 6 10 1 7 10 1 10 7 1 10 10 1 9 11 1 9 12 1 13 1 1 14 1 1	12



Пояснення. Щоб досягти оптимальності витрат, Королю Артуру необхідно скасувати данину у фортеці 3 і зібрати нараду у фортеці 2.

Розв'язання

Зазначимо, що кола, які не перетинаються, утворюють дерево, причому кола відповідають ребрам, а області зв'язності — вершинам. Тому задачу можна переформулювати так: у кожній

вершині дерева перебуває деяка кількість людей, що можуть переходити з вершини до вершини по ребрах, сплачуючи деяку суму грошей за цю операцію. Дозволено зробити ціну переходу нульовою не більш ніж для K ребер. Треба зібрати всіх людей в одній вершині, витративши на це якомога менше грошей.

Таким чином, задачу можна розбити на дві підзадачі: геометричну та оптимізаційну. *Геометрична підзадача* полягає у побудові дерева та визначенні початкової кількості людей у кожній вершині.

Геометрична підзадача: наївний розв'язок за $O(N^2 + NM)$

Для кожного кола знайдемо коло мінімального радіуса, що містить дане, та позначимо область зв'язності, що лежить між цими двома колами, як предка у дереві до вершини, що відповідає області зв'язності внутрішнього кола. У випадку, якщо немає кола, що містить задане, вважатимемо його предком деякий глобальний корінь нашого дерева — вершину, що відповідає області зв'язності, яка лежить поза межами всіх кіл.

Геометрична підзадача: метод скануючої прямої (sweep line) за $O\big((N + M)\log(N + M)\big)$

Для полегшення подальшого викладу будемо вважати, що на додаток до всіх кіл, що задані у вхідному файлі, ми маємо деяке коло із центром у точці $(0, 0)$ і радіусом таким, що всі інші точки та кола лежать усередині нього. Під час «сканування» ми не будемо розглядати події початку та закінчення цього кола, натомість дуги цього кола будуть постійно міститися у допоміжних структурах даних.

Будемо «сканувати» прямою площину зліва направо, підтримуючи у деякому бінарному збалансованому дереві верхні та нижні дуги кіл, що перетинають пряму в її поточному розташуванні. Ці дуги будемо впорядковувати у дереві за ординатою точки перетину з прямою. Кожну з можливих подій будемо опрацьовувати так:

Якщо скануюча пряма **натрапила на нове коло**, ми знаходимо у дереві дуг дугу, найближчу (або зверху, або знизу — неважливо) до точки дотику прямої та кола, і перевіряємо, чи коло, якому належить ця дуга, є таким, що містить нове коло. У випадку, якщо це так, позначаємо його «батьком» (предком) нового кола; якщо ні, то, як нескладно зрозуміти, батьком нового кола є батько кола, якому належала знайдена дуга. Після того як вершина — батько нового кола визначена, додамо дві його дуги до дерева дуг.

Якщо скануюча пряма **виходить за межі кола**, видалимо з дерева дуг вершини, що відповідають дугам цього кола.

Якщо скануюча пряма **натрапила на точку**, знаходимо коло мінімального радіуса, яке містить точку, у спосіб, аналогічний до наведеного вище методу знаходження батька нового кола: точку можна вважати колом радіуса 0.

Зауваження щодо реалізації дерева дуг:

- У мові C++ можна використовувати стандартний контейнер `set` із динамічним компаратором, що порівнює дуги за значенням y-координат точок перетину дуг зі скануючою

прямою в її поточному розташуванні. При цьому є нюанс: дуги, що належать одному колу, треба порівнювати безпосередньо (нижня під верхньою), не порівнюючи y-координати цих дуг. Це пов'язано з тим, що в перший та в останній моменти, коли пряма перетинатиме коло, вона буде перетинати обидві дуги в одній і тій самій точці.

- Існує також розв'язок, що не використовує динамічний компаратор: він порівнює дуги за відносним розташуванням кіл, до яких ці дуги належать. Такий розв'язок є більш ніж удвічі швидшим за розв'язок із динамічним компаратором.

До *оптимізаційної підзадачі* також є кілька підходів.

Оптимізаційна підзадача: випадок $K = 0$

У цьому випадку в Артура немає можливості скасовувати плату за перехід по ребрах. Тоді єдине, що залежить від Артура, — вибір вершини збору. Маємо наївний розв'язок з асимптотикою $O(N^2)$: перебрати варіанти вершини збору та для кожного варіанта за допомогою пошуку в глибину знайти ціну збору в даній вершині. Зробити це можна, рахуючи дві допоміжні величини: кількість людей у піддереві та сумарну ціну збору всіх людей з піддерева у корені цього піддерева. Такий розв'язок оптимізується до лінійного за допомогою підтримання під час пошуку в глибину описаних величин не тільки у піддеревах, а також і в наддереві (тобто в дереві без піддерева).

Оптимізаційна підзадача: розв'язання з допомогою структур даних

Нехай зафіксовано деяку вершину збору. Зорієнтуємо всі ребра так, як по них будуть іти люди до вершини збору. З цих міркувань для кожного ребра можна оцінити те, яку знижку отримає Артур, якщо зробить ціну проходу по даному ребру нульовою. Ця величина дорівнює кількості людей у відповідному піддереві, помноженій на ціну переходу. Таким чином отримаємо розв'язок за $O(N^2 \log N)$: перебираємо можливі варіанти вершини збору та за допомогою пошуку в глибину вираховуємо знижки для кожного ребра; далі вибираємо K ребер, для яких знижки найбільші.

Для оптимізації цього розв'язку до $O(N \log N)$ подивимось, яким чином змінюється множина знижок при перенесенні вершини збору до сусідньої вершини: змінюється лише знижка на ребро, через яке це перенесення здійснювалося. Отже, необхідно запровадити деяку структуру даних, що підтримує набір чисел (разом із дублікатами) та може швидко опрацьовувати запити двох типів:

- замінити деякий елемент x на y;
- знайти K максимальних елементів.

Зауважимо, що перша операція еквівалентна послідовності операцій: додати елемент y до набору, видалити елемент x із набору. Цю структуру даних можна промоделювати за допомогою контейнера `map` зі стандартних бібліотек C++, підтримуючи поточне значення K-го найбільшого елемента та кількість елементів, строго менших за K-й найбільший елемент. Альтернативою до цієї структури даних є дерево відрізків зі стисканням координат або розріджена його модифікація.

Оптимізаційна підзадача: розв'язання з допомогою жадібного алгоритму

Розглянемо деяке ребро (U, V) дерева. Нехай його ціна — $cost$. Позначимо кількість людей у піддереві вершини U через L, а у піддереві вершини V — через R. Доведемо, що до ціни збору це ребро додає $\min\{L, R\} \cdot cost$ грошей при оптимальному виборі вершини збору. Справді, нехай, наприклад, $L < R$, а вибрана вершина збору X міститься у піддереві L. Хай зібрати всіх людей з L у X коштує A, а зібрати всіх з R у V коштує B. Позначимо ціну проходу однієї людини по шляху з V у X через W. Тоді ціна збору наради у вершині X складає $A + B + W \cdot R$, а у вершині V — $A + B + W \cdot L$, що, очевидно, менше. Отже, X не є оптимальним вибором вершини збору.

Зауважимо, що оцінка $\min\{L, R\} \cdot cost$ не залежить від того, чи ми скасовували плату за перехід через інші ребра. Тому достатньо вирахувати дану величину для всіх ребер та вилучити K найбільших значень — сума решти і є відповіддю.