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

Бондаренко Віталій Вікторович,
асистент факультету кібернетики Київського Національного університету ім. Тараса Шевченка.
Ягіяєв Шаміль Ігорович
менеджер проектів компанії «Арісент Україна».

ЗАВДАННЯ ПЕРШОГО ТУРУ

1. Календар (запропонував Данило Мисак)

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. Учені з'ясували таке.

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, у 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, у якому вони накреслені у ***першій*** печері. Третій рядок також містить N різних натуральних чисел, кожне з яких не перевищує N, — номери слів у порядку, у якому вони накреслені у ***другій*** печері.

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

Оцінювання

1. 20 % балів: $2 \leqslant N \leqslant 5$.
2. 20 % балів: $5 < N \leqslant 150$.
3. 20 % балів: $150 < N \leqslant 3000$.
4. 40 % балів: $3000 < N \leqslant 10^5$.

Приклади вхідних і 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 на рис. 1).

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

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

Рис. 1

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

Рекомендації щодо розв'язання

Задачу можна розв'язувати простим способом, перебравши всі можливі відповідності між місяцями першого і другого племені і для кожної відповідності, підрахувавши кількість місяців, які мають однакові назви. Щоби встановити 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 $O(N^2)$, 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. Наразі 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 ($0 \leqslant d_i \leqslant N-1$) 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 й 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 $a_i-b_i \equiv d \pmod N$.

Отже, 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

$a_i - b_i \equiv d \pmod N$. Тепер залишається утворити масиви чисел a_i та b_i, підрахувати всі різниці $a_i - b_i$ за модулем N і 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 $O(N)$.

2. Мутація (Ярослав Твердохліб)

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 так, 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 із 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 як найменшу 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

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

Приклад вхідних і 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.

Рекомендації щодо розв'язання

Розглянемо геном, який був отриманий 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 одного разу. Отже, наша задача стає еквівалентною такій: вибрати максимальну кількість типів генів, яка у разі видалення генів всіх інших типів перетворює початковий геном у ідеальний.

Кожному типу генів поставимо у відповідність відрізок прямої, який починається у найлівішому гені такого типу і закінчується у найправішому. Тоді нам потрібно вибрати максимальну за розміром підмножину відрізків, у якій жодні 2 відрізки не перетинаються.

Останню задачу можна розв'язувати наступним 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 описаний алгоритм методом математичної індукції по розміру підмножини початкової множини відрізків. Для множини розміру 1 алгоритм, очевидно, є правильним. Припустимо тепер, що алгоритм працює коректно для всіх підмножин відрізків розміру строго меншого за K. Нехай ми маємо підмножину розміру K і хочемо знайти оптимальну підмножину відрізків для неї. Розглянемо будь-яку таку оптимальну підмножину. Якщо її найлівіший відрізок не є першим у порядку сортування за правим кінцем, то видалимо його з відповіді і замість нього додамо відрізок, у якого правий кінець найлівіший. При цьому новий відрізок, очевидно, не буде перетинатись із жодним іншим відрізком з оптимальної множини. Отже, тепер видалимо найлівіший відрізок і всі ті, які з ним перетинаються, з нашої підмножини розміру K. Отримаємо підмножину меншого розміру, для якої за припущенням алгоритм працює коректно.

Залишилось лише помітити, що всі відрізки вже є відсортованими, якщо йти по масиву зліва направо і розглядати черговий відрізок лише коли поточний ген є правим кінцем якогось відрізка.

Складність запропонованого алгоритму $O(N)$.

3. Космічні камінці (Данило Мисак)

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, у 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 — но-

мер 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 камінця. Вхідні дані гарантують виконання таких умов:

- маси всіх камінців, що падали на планету, попарно різні;
- на момент, коли уряд збирає податок, у кожному з M містечок є хоча б по одному камінцю;
- 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

1. 30 % балів: $2 \leqslant M < N \leqslant 5000$.
2. 30 % балів: $5000 < N \leqslant 2 \cdot 10^5$, $2 \leqslant M \leqslant 5$.
3. 40 % балів: $5000 < N \leqslant 2 \cdot 10^5$, $5 < M < N$.

Приклад вхідних і 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

Рекомендації щодо 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 і 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 $[(N-M+1)/2]=O(N-M)$. Отже, 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 більш пріоритетні), 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 і 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. Утім, 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 таким способом: 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)? Наразі в новій купі більш пріоритетними 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.

Отже, 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)+2log$M$)=O(log$N$).

Оскільки кожну операцію алгоритм опрацьовує за O(log$N$), загальну ефективність алгоритму можна оцінити як O(Nlog$N$). 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

$$\left\lceil \frac{c_i}{\lceil \sqrt{c_i} \rceil} \right\rceil \leq \left\lceil \frac{c_i}{\sqrt{c_i}} \right\rceil = \lceil \sqrt{c_i} \rceil.$$

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, назвемо 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 й 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), у якому міститься цей елемент. Потім, якщо потрібно, 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.

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

Операцію типу 2 алгоритм також виконує у кілька етапів.

1. Виводимо (уже підрахований) найбільший елемент масиву другого рівня і 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

$$O((N-M)\sqrt{N}+M\sqrt{M}) = O(N\sqrt{N}).$$

4. Олімпійські Авіалінії (Ярослав Твердохліб)

Імператор Олімпії вирішив з'єднати 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 із 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 і 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 операційпрограма 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 одного рейсу, причому перельоти по них можуть займати різну кількість часу. Можливе існування рейсів, які починаються і закінчуються в одному і тому самому місті. Усі 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, що потрібен 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 імперії Олімпія і кількість операцій, виконаних програмою **ОлімпБуд**. У кожному з наступних M рядків міститься інформація про чергову операцію. Перше число в рядку рівне 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. У i-му з них має міститись найменший час у хвилинах, за який можна долетіти зі столиці у 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: $N \leqslant 2000$, $M \leqslant 4000$;
- 10% балів: виконуються операції лише першого типу;
- 15% балів: $C \geqslant 0$ та якщо місто зустрілось у якості A в операції другого типу, то далі у вхідному файлі воно більше не зустрічається в жодній операції ні в якості A, ні в якості B;
- 15% балів: те саме, що у блоці 3, але на C не накладається умова невід'ємності;
- 15% 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

Рекомендації щодо розв'язання

Обмеження задачі виключають можливість явно побудувати граф і запустити алгоритм пошуку найкоротшого шляху у ньому, оскільки кількість ребер може зростати експоненційно зі збільшенням кількості операцій. Для розв'язку задачі ми побудуємо новий граф, у якому окрім початкових N вершин будуть ще $O(M)$ додаткових вершин і $O(N+M)$ ребер. Наразі довжина найкоротшого шляху від вершини 1 до початкових N вершин буде такою самою, як і у графі, побудованому після виконання всіх операцій з умови (назвемо 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 2 вершини in_v і $active_v$. Кожну операцію будемо виконувати так.

1. При виконанні операції першого типу з параметрами A, B та C додамо ребро з вершини $active_A$ до вершини in_B довжини C.

2. При виконанні операції другого типу з параметрами A, B та C додамо ребро з вершини $active_B$ до вершини $active_A$ довжини C, після чого створимо нову вершину $active_A'$. Додамо ребро довжини 0 з цієї нової вершини до $active_A$, після чого оголосимо вершину $active_A'$ новою $active_A$ (стара вершина просто залишиться в графі). Цими діями ми імітуємо копіювання списка суміжності вершини A до списку вершини B з додаванням величини C до всіх ребер. До того ж у разі додавання нових ребер з початком у вершині A вони не будуть додаватись до списку вершини B, оскільки зі старої вершини $active_A$ нема шляху до нової $active_A'$, який би не проходив по жодній вершині типу in.

Після виконання всіх операцій для кожної вершини v початкового графа додамо у новому графі ребро з in_v до $active_v$ довжини 0. Після цього треба виконати алгоритм пошуку найкоротшого шляху з вершини in_1 до всіх інших вершин типу in.

Якщо для зазначених цілей використовувати алгоритм Левіта або алгоритм Беллмана-Форда, то розв'язок набере 15 балів. Алгоритм Дейкстри наразі набирає 55 балів, оскільки у графі можуть бути наявні від'ємні ребра.

Для того, щоб позбутися від'ємних ребер, потрібно скористатися умовою того, що у початковому графі, отриманому після виконання всіх операцій з умови, усі ребра є додатними.

Розглянемо будь-яку вершину нового графа, яка не належить до типу in. Нехай найменше ребро, яке входить до неї, має довжину d. Тоді ми можемо відняти від довжин усіх ребер, що входять до неї величину d, після чого додати цю саму величину до всіх ребер, що виходять з цієї вершини. Очевидно, що будь-який шлях, який проходить через дану вершину і не закінчується у ній, не змінить свою довжину. Наразі в дану вершину вже не входить жодного ребра від'ємної довжини. Отже, за допомогою таких операцій ми можемо видаляти з графа одні від'ємні ребра і, 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. Факт про відсутність циклів без вершин типу in випливає з того, що у разі проходження 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 пройти не зможемо.

Отже, ми можемо видалити всі вершини типу in із графа, та топологічно відсортувати отриманий ациклічний граф. Після цього повернемо назад всі вершини у граф і виконаємо операції зі зміною довжин ребер, описані вище. Оскільки граф без вершин типу in є ациклічним, після виконання всіх операцій в ньому не буде від'ємних ребер, тому можна використовувати алгоритм Дейкстри.

Складність запропонованого алгоритму:

$$O((N+M)\cdot\log(N+M)).$$

(Далі буде)