Козак Вус та граф

Автор та розробник: Антон Ципко

Блок 1. Якщо $k_i = 1$, то потрібно вивести цю вершину. Інакше — -1.

Блок 2. В операціях другого типу можемо використовувати СНМ (систему неперетинних множин). В операціях першого типу можемо проходитися по всім вершинах і перевіряти, чи ця вершина належить множині. Для того, щоб перевірити, чи вершина знаходиться у множині, потрібно перевірити чи корінь цієї вершини у СНМ збігається з коренем запитаної вершини.

Блок 3. Можемо для кожного стану зберігати масив СНМ. Після кожної операції нам потрібно запам'ятати масив СНМ. Тоді, коли є операція другого типу, то нам потрібно скопіювати повністю масив, а потім виконати на ньому операцію. А коли є операція третього типу, то потрібно лише скопіювати правильний масив.

Блок 4. Обмеження означає, що компонента буде відрізком на масиві. Коли ми отримуємо запит другого типу, то нам потрібно знайти крайню вершину зліва, це можна зробити бінарним пошуком, а потім перевірити чи k-та вершина справа також знаходиться в цій же компоненті. Також можна зберігати відрізки у `set< pair<int, int> >`, а потім використовувати lower_bound для знаходження відповіді.

Блок 5. Можемо для кожної компоненти зберігати множину вершин в ній. Коли у нас є операція другого типу, нам потрібно додати з меншої за розміром множини у більшу. Цей метод називається «від меншого до більшого». Він дасть можливість виконати всі ці операції сумарно за $O(n \log^2 n)$. На операції першого типу потрібно знаходити k-ий елемент у множині.

Множину можна підтримувати за допомогою декартового дерева, або за допомогою спеціальної структури даних в C++, яка дуже схожа на set, але крім операцій вставки, також вміє шукати k-ий елемент.

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
// s.find_by_order(num) - returns iterator to the num-th element (0 <= num < s.size())
// s.order_of_key(key) - returns number of elements smaller than key
```

Блок 6. Можемо підтримувати персистентне дерево відрізків. Для операцій другого типу нам потрібно вміти присвоювати на відрізку. А для операцій першого типу потрібно вміти знаходити перший елемент, який має таке ж значення (початок компоненти). Дуже схожий метод до четвертого блоку.

Блоки 7-9. Ми можемо відповідати на запити оффлайн, тому ми можемо побудувати dfs на запитах. Тобто, у нас будуть вершини - запити. У нас в кожну вершину типу 1 чи 2 можна потрапити через попередню за номером вершину, а у i-ту вершину третього типу, можна потрапити лише з вершини x_i. Тобто, якщо $t_i = 3$, то у нас буде ребро (x_i, i), якщо ж $t_i \neq 3$, то буде ребро $(i-1, i)$. Такий dfs дозволить підтримувати певний стан, проте такий стан потрібно вміти оновлювати та відкочувати. Основна проблема саме у відкочуванні.

Давайте для початку розберемося, як це робити за $O(nq \log n)$. Будемо підтримувати СНМ, проте ми не будемо використовувати оптимізацію, у якій усі вершини будуть вести у корінь. Кожна вершина буде вести у свого прямого батька, а не в корінь. Коли у нас є операція другого типу, то, щоб об'єднати компоненти, нам потрібно під'єднати меншу компоненту до більшої. Тобто, нехай ми об'єднуємо компоненти вершин v_i та u_i. Хай v'_i — корінь компоненти v_i в СНМ. u'_i — аналогічно для u_i. Також нехай компонента v'_i менша за компоненту u'_i. Тоді нам потрібно сказати, що батьком вершини v'_i є вершина u'_i. Потім, щоб скасувати це об'єднання, нам потрібно сказати, що батьком вершини v'_i є сама вершина v'_i (тобто, вона сама є коренем). Для операцій першого типу, ми можемо пройтися по всім вершинам і перевірити, у якій компоненті вона знаходиться. Потрібно буде вивести k-ту таку вершину. Для кожної вершини ми можемо знайти корінь компоненти за $O(\log n)$ (згадуємо метод «від меншого до більшого» з п'ятого блоку). Отже, ми вміємо обробляти запит першого типу за $O(n \log n)$, а другого за $O(\log n)$.

Розіб'ємо усі вершини на $\sqrt{n}$ блоків довжини $\sqrt{n}$. Для i-го блоку будемо зберігати кількість вершин у цьому блоці, які знаходяться в компоненті вершини j. Нехай це буде зміна c_{ij}. Якщо у

нас є операція другого типу, то ми можемо кожен блок оновити за $O(1)$, додавши одне значення до іншого. Отже, за одну операцію другого типу нам потрібно $O(\sqrt{n})$ часу, щоб оновити масив c. Стільки ж часу потрібно, щоб роз'єднати компоненти. Для виконання операції першого типу, будемо спочатку проходитися по всім блокам і дивитися, у якому блоці знаходиться наша шукана відповідь. Тут ми витратимо $O(\sqrt{n})$ часу. Потім ми будемо перебирати усі вершини, так само як у попередньому абзаці, проте будемо перебирати не абсолютно всі вершини, а лише ті, що в цьому блоці. Отже, для такого нам потрібно $O(\sqrt{n}\log n)$ часу. Сумарна асимптотика виходить $O(q\sqrt{n}\log n)$.

Зверніть увагу, що необов'язково використовувати саме $\sqrt{n}$ як розмір блока. Нехай s — розмір блока. Тоді асимптотика вийде $O(q\max(\frac{n}{s}, s\log n))$.