Лекция 2.
SQRT-декомпозиция

179 Школа, Овчинников Андрей

21 сентября 2022 г.

Аннотация

Перед тем, как вы прочитаете то, что написано ниже обращу внимание на то, что в данном конспекте могут быть допущены ошибки и опечатки, если вы находите подобное, то пишите мне в личку или группу в телеграмм с упоминанием (*то есть через* @). Буду рад, если вы поможете довести конспект до хорошего безошибочного состояния.

О применении

Постановка задачи

Пусть дана задача такая, что мы можем 'в тупую' считать ответ на подотрезке чего-либо (массива, запросов и т.д.) и есть операция обновления/добавления *элемента*. И при этом мы умеем объединять уже посчитанный результат и получать ответ для всего (массива, запросов и т.д). Тогда заметим, что если мы будем работать с *блоками*, то есть при обновлении/добавлении *элемента* в *блок*, мы будем пересчитывать только локальный ответ. А при запросе на ответ для некоторого множества, будем просто мерджить ответы для необхоимых блоков.

Оценка сложности

Оценим сложность операций.

- *Операция обновления.* Оценим её как $O(size_{block} * \alpha(size_{block}))$, где α - некоторая функция количества операций необходимых для обновления ответа для $size_{block}$ *элементов.*

- *Операция получения ответа.* Оценим её как $O(cnt_{block} * \beta(cnt_{block}))$, где β - некоторая функция количества операций необходимых для обновления ответа для cnt_{block} *блоков.*

Достаточно *размытая* асимптотика. Но как говориться, всё познаётся в сравнении. Пусть мы решаем нашу задачу, но с одним блоком, содержащим все *элементы.*

Тогда *операция обновления* будет производиться за $O(cnt_{block} * size_{block} * \alpha(cnt_{block} * size_{block}))$, а получения ответа за $O(1 * \beta(1))$. Казалось бы мы потеряли немного в обновлении ответа, но зато ускорили получение ответа. Однако следует заметить, что вам могут дать тест, где будет, как большая часть операций - обновления, так и наоборот. Поэтому сложность *операции* нужно оценивать, как среднее сложности обеих операций.

- Для одного блока: $O(cnt_{block} * size_{block} * \alpha(cnt_{block} * size_{block}) + \beta(1))$

- Для нескольких блоков: $O(size_{block} * \alpha(size_{block}) + cnt_{block} * \beta(cnt_{block}))$

Соответственно, имеем только одну неизвестную во втором случае, тожем найти минимум и при возможности уменьшить асимптотику.

Обычно $\beta(x) = \alpha(x)$, и тогда имеем право сравнивать:

- $O(cnt_{block} * size_{block} * \alpha(cnt_{block} * size_{block}))$

- $O(size_{block} * \alpha(size_{block}) + cnt_{block} * \alpha(cnt_{block}))$

Откуда и вытекают оптимальные значения для $size_{block} = cnt_{block} = \sqrt{cnt_{items}}$ Сравним итоговую сложность.

- $O(cnt_{items} * \alpha(cnt_{items}))$

- $O(\sqrt{cnt_{items}} * \alpha(\sqrt{cnt_{items}}))$

Так как $\alpha(x)$, обычно, *неубывающая* функция, то получаем минимальное ускорение в $\sqrt{cnt_{items}}$ раз!

Пример задачи

Постановка задачи

Пусть дана задача поиска минимума на отрезке массива и дана операция изменения значения на Δ на отрезке.

Теоретическое решение

Заметим, что мы умеем пересчитывать минимум для k элементов за $O(k)$.

Просто проходим за линию по всем элементам сохраняя минимум.

Аналогично для блоков, зная ответ для каждого блока, можем найти среди всех ответов для блоков минимум за число блоков.

Как обновлять?

Очень просто, давайте для каждго блока хранить дополнительный элемент, который отслеживает изменение для всего блока. А если запрос попадает на блок частично, то будем 'в тупую' проходить и обновлять значения массива.

Код

```
#include <bits/stdc++.h>

using namespace std;


const int SIZE = 500;
const int INF = 2e9;

int
main()
{
    int n, m; cin >> n >> m;

    vector<int> array(n, INF);
    vector<int> block_ans(SIZE, INF);
    vector<int> block_del(SIZE, 0);

    for (int i = 0; i < n; ++i) {
        cin >> array[i];
        block_ans[i / SIZE] = min(block_ans[i / SIZE], array[i]);
    }

    for (int i = 0; i < m; ++i) {
        char c; cin >> c;
        int l, r; cin >> l >> r;
        l--; r--;
        if (c == '+') {
            int delta; cin >> delta;
            int start_block = (l + SIZE - 1) / SIZE;
            int finish_block = r / SIZE;

            for (int j = l; j < min(start_block * SIZE, r + 1); ++j) {
                array[j] += delta;
                block_ans[j / SIZE] = min(block_ans[j / SIZE], array[j]);
            }
```

```
            for (int j = start_block; j <= finish_block; ++j) {
                block_del[j] += delta;
            }
            if (start_block <= finish_block) {
                for (int j = finish_block * SIZE; j <= r; ++j) {
                    array[j] += delta;
                    block_ans[j / SIZE] = min(bloack_ans[j / SIZE], array[j]);
                }
            }
        } else {
            int start_block = (l + SIZE - 1) / SIZE;
            int finish_block = r / SIZE;
            int res = INF;

            for (int j = l; j < min(start_block * SIZE, r + 1); ++j) {
                res = min(res, array[j] + block_del[j / SIZE]);
            }
            for (int j = start_block; j <= finish_block; ++j) {
                res = min(res, block_del[j] + block_ans[j]);
            }
            if (start_block <= finish_block) {
                for (int j = finish_block * SIZE; j <= r; ++j) {
                    res = min(res, array[j] + block_del[j / SIZE]);
                }
            }

            cout << res << endl;

        }
    }
    return 0;
}
```