Разбор ДО часть 1

Сергей Панин

March 2024

- Для нахождения ближайшего числа, являющегося степенью двойки, котрое больше либо равно заданному, можно использовать std::bit_ceil ссылка
  Принимает только unsigned типы и есть в стандарте C++20 и выше
  Стандарт C++ нужно изменить руками в среде разработки где вы пишете. Либо добавить флаг -std=c++20 в параметры компиляции
- ОЧЕНЬ ЧАСТОЙ ОШИБКОЙ ЯВЛЯЕТСЯ ПЕРЕПОЛНЕНИЕ int ИСПОЛЬЗУЙТЕ long long

1 Задача A (Сумма на отрезке)

Основной код для задачи был написан на лекции. Давайте научимся им пользоваться.

После того как ввели массив надо вызвать функцию build, передва в неё только что введённый вектор.

Чтобы отвечать на запросы:
sum(1, 0, n-1, l-1, r-1) .
upd(i-1, x)

2 Задача B (Прибавление на отрезке)

В этой задаче не нужно писать дерево отрезков на массовые операции. Здесь можно использовать трюк с префиксными суммами. Давайте рассмотрим другую задачу и вначале решим её.

Дан массив ar длины n, заполненый нулями, приходят запросы прибавить значение x на отрезке [l,r] После всех измнений вывести получившийся массив.

Заведём отдельный массив pref длины n+1 изначально тоже заполненый нулями. Теперь давайте обрабатывать запросы таким образом:

$pref[l] += x$

$pref[r+1] -= x$

Таким образом, если просуммировать все значения от 0 до i в этом массиве то можно понять на сколько изменился i элемент. Фактически ar[i] = $\sum_{j=1}^{i} pref_j$

Теперь применим эту же технику, чтобы решить нашу задачу, только вместо того чтобы в конце брать операцию префиксной суммы от массива, будем хранить дерево отрезков и использовать его для ответа на запросы.

Когда нас просят прибавить на отрезке, то в l индексе прибавляем x, а в r+1 вычитаем x. При запросе на вывод элемента будем выводить sum(1, 0, n-1, 0, i)

3 Задача C (Разница между максимумом и минимумом на отрезке)

Идея решения задачи идентична первой. Только вместо суммы у нас операции минимума и максимума и два массива под дерево отрезков - минимум и максимум. Для минимум и максимума можно использовать одну функцию upd, build и функцию запроса (get). Вместо того чтобы писать отдельные build, upd, get под min и max, в get можно возвращать пару значений, а в функции upd изменять сразу 2 массива.

В коде изменится не так много, лишь те места где мы объединяем значения двух вершин. В upd строка $tree[i] = tree[2*i]+tree[2*i+1]$ изменится на строку $t[i] = max(t[2\cdot i], t[2\cdot i+1])$ для минимума аналогично. В коде запроса нужно изменить возвращаемый нейтральный элемент когда отрезок в котором мы находимся не включён в ответ. И нужно реализовать правильную конкатенацию ответов полученных из левого и правого сына.

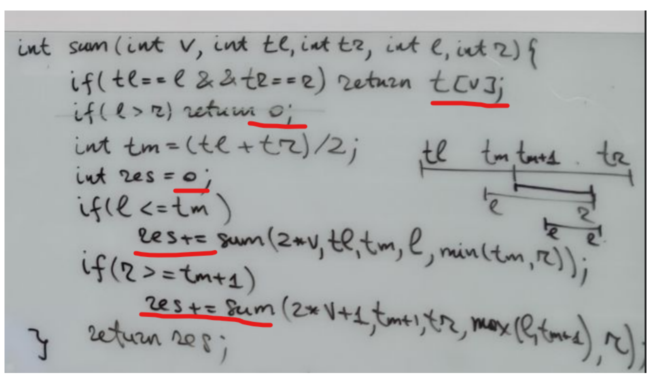

Рис. 1: Что изменится в коде запроса

На фото отмечено что надо поменять. Вместо res += sum($2 \cdot v$...) надо написать свою функцию объединения. Хорошим тоном считается написать отдельную функцию которая умеет объединять 2 значения хранящихся в ДО.

4 Задача D (Игра с волчком)

В любой задаче при появлении круга надо приписывать к основному массиву его копию, тогда работать с ним будет значительно легче. Построим ДО(сумма на отрезке и изменение в точке) на двух объединённых массивах и будем делать ровно то что просят в задаче, считать сумму на отрезке, делить на его длину и изменять элемент. Изменяем 2 индекса потому что справа приписана копия массива.

В этой задаче удобна функция std::iota(begin, end, start) для заполнения изначального массива. Также в этой задаче надо быть аккуратным с памятью, например здесь не заходит ДО на указателях. И если использовать 4N памяти для дерева отрезков, то тоже будет ML - лучше всего подсчитать сколько нужно памяти с помощью std::bit_ceil и использовать это как константу.

Возможная реализация при использовании до на полуинтервалах и корнем в 0:

```cpp
    int n, k;
    cin >> n >> k;

    vector<ll> ar(2*n);
    iota(ar.begin(), ar.begin()+n, 1);
    iota(ar.begin()+n, ar.begin()+2*n, 1);

    build(ar, 0,  0, 2*n);

    for (int i = 0; i < k; i++) {
        ll ni;
        cin >> ni;

        ll len = Sum(0, ni-1, ni, 0, 2*n);
        ll sum = Sum(0, ni, ni + len, 0, 2*n);

        sum /= len;
        Set(0, ni-1, sum, 0, 2*n);
        Set(0, ni-1+n, sum, 0, 2*n);
    }

    cout << Sum(0, 0, n, 0, 2*n) << '\n';
    return 0;
```

Рис. 2: Основная часть решения

5 Задача E (Армия)

В этой задаче используется часто встречающаяся идея. Её нужно запомнить и использовать в других задачах.

Заведём массив массив in_row в котором единичка в i позиции будет обозначать что солдат с ростом i есть в ряду. На массиве построим дерево отрезков с суммой на отрезке и изменением в точке. Будем идти слева направо и выставлять солдат по очереди, для этого будем устанавливать in_row[h[i]] = 1 где h[i] - рост i солдата. Чтобы узнать сколько солдат стоит левее него и выше надо просуммировать in_row от h[i]+1 до n индекса.

Смысловая часть решения

```
    int n, k;
    cin >> n >> k;
    ll ans = 0;

    for (int i = 0; i < k; i++) {
        tree.assign(maxn, 0);
        for (int i = 0; i < n; i++) {
            int x;
            cin >> x;
            ans += Sum(0, x, n+1, 0, n);
            Set(0, x-1, 1, 0, n);
        }
    }

    cout << ans << '\n';
```

Рис. 3: Армия

6 Задача F (Военные учения 2)

В задаче появился круг - значит дописываем в конец его копию, для удобства. Эту задачу можно решать спуском по дереву, а можно бинарным поиском. Спуск по дереву является более сложной техникой, поэтому рассмотрим бинарный поиск.

Давайте заметим следующий факт - если осталось last человек, а нам надо пройти k людей, то можно не проходить круг несколько раз, а взять остаток деления k на last и столько людей надо набрать проходясь по часовой стрелке. Исключением является остаток 0, тогда надо обойти last людей.

Заведём массив из единичек длины n. Единичка означает что человек с индексом где она находится всё ещё в круге. Как только он будет выбывать мы будем ставить на его индексе 0 и выводить. На этом массиве построим ДО на сумму и изменение в индексе

Теперь поймём как с помощью бин поиска решать задачу. Индексация с 0. Изначально находимся на 0 позиции и хотим отсчитать k людей и получить k-го. Для этого сделаем бинарный поиск по сумме единичек справа, если сумма будет больше нужной, то перепрыгнули и нужный человек левее, иначе он правее. Отдельно заботиться о "хвосте" не приходится из-за трюка с дописыванием в конец. Когда бинарный поиск сходится, то получаем нужный индекс и выводим его, ставим нолики в 2 нужные позиции. И обновляем стартовую позицию с 0 на только что найденную и запускаем бинарный поиск заново

```
int n, k;
cin >> n >> k;
vector<ll> ar(2*n, 1);
build(ar, 0,  0, 2*n);
int pos = 0;
for (int i = 0; i < n; i++) {
    int need = k % (n-i);
    if (need == 0) {
        need = n-i;
    }
    int l = pos;
    int r = 2*n;
    while (r - l > 1) {
        int m = (l + r) / 2;
        int sum = Sum(0, pos, m, 0, 2*n);
        if (sum >= need) {
            r = m;
        } else {
            l = m;
        }
    }
    pos = (l % n);
    cout << pos+1 << " ";
    Set(0, pos, 0, 0, 2*n);
    Set(0, pos+n, 0, 0, 2*n);
}
```

Рис. 4: Смысловая часть решения

7 Задача E (И снова запросы на отрезке)

Одним из решений является MergeSort tree. Это дерево отрезков в вершине которого мы храним отсортированный вектор чисел. Чтобы построение работало быстро необходимо использовать слияние двух отсортированных отрезков за линейное время. Проще всего это сделать с помощью встроенной функции std::merge() Она принимает на вход 5 параметров, итераторы на начало и конец первого отрезка, 3 и 4 параметры по аналогии для 2 отрезка. И 5 параметром итератор на начало массива куда нужно записывать готовый отсторированный массив.

Запрос работает аналогично с обычным деревом отрезков, только когда нужно возвращать значение из текущей вершины, то пользуемся std::lower_bound чтобы найти нужное значение. Но если пользоваться им напрямую, то придётся разбирать сложные случаи. Чтобы упростить себе жизнь можно изменить знак всех чисел на противоположный. А при запросе отправлять -x. Нейтральным элементом в таком случает будет выступать 1. Пример реализации функции подсчёта в вершине:

```
int calc_in_node(int v, int x) {
    auto it = lower_bound(all(tree[v]), x);
    if (it == tree[v].end()) {
        return 1;
    }
    return *it;
}
```

Рис. 5: Подсчёт функции в вершине

И код функции запроса в случае использования ДО на полуинтервалах и нумерации корня с 0. [l,r) - полинтервал запроса, [lx, rx) - полуинтервал за который отвечает вершина в которой мы сейчас находимся

```
int Querry(int v, int l, int r, int lx, int rx, int x) {
    if (l <= lx && rx <= r) {
        return calc_in_node(v, x);
    }
    if (rx <= l || r <= lx) {
        return 1;
    }
    int m = (lx + rx) / 2;
    int ql = Querry(2*v + 1, l, r, lx, m, x);
    int qr = Querry(2*v + 2, l, r, m, rx, x);
    return min(ql, qr);
}
```

Рис. 6: Функция запроса

8 Задача E (Баш и сложная математическая головоломка)

Заметим, что если на отрезке существует более одного числа, которое не делится на gcd, то тогда нельзя путём изменения одного элемента получить необходимое значение. Чтобы это проверить давайте спускаться по дереву отрезков и при необходимости запускать спуск в самый низ. Для реализации ДО на g++ (GCC) компиляторе поможет функция __gcd. Построим дерево отрезков которое в вершине будет хранить gcd чисел на отрезке за который отвечает эта вершина. Строится по аналогии с деревом отрезков на min, max с заменой соответствующих частей кода. Нейтральный элемент в этом случае будет 0, gcd(0, x) = x.

Теперь надо научиться как-то эффективно проверять что на отрезке максимум один элемент не делится на число из запроса. Для этого напишем 2 функции - одна (QuerrySeg) будет работать как обычная сумма - находить нужные отрезки которые полностью лежат в отрезке запроса. Вторая (QuerryPos) будет спускаться до самого нижнего слоя где и лежит число не делящееся на число из запроса. Эту функцию запускаем если отрезок полностью лежит в запросе. Эта функция постоянно должна идти в сына значение gcd которого не делится. Чтобы не делать лишних операций надо будет завести глобальный счётчик количества найденных чисел не делящихся на число y из запроса. Если при очередном рекурсивном заходе в QuerrySeg окажется что этот счётчик 2 или больше, то сразу делаем return иначе один запрос может начать работать за линию, а не за log. Также нужно аккуратно обрабатывать случай когда функцию поиска элемента в позиции обнаружила что и в правом и в левом сыне находятся числа не делящиеся на y. Код реализации ниже. Как обычно на полинтервалах, массиве и нумерации корня с 0

```
int cnt_not_devide = 0;
void QuerryPos(int v, int g, int lx, int rx) {
    if (rx - lx == 1) {
        cnt_not_devide += 1;
        return;
    }
    int m = (lx + rx) / 2;
    if ((tree[2*v + 1] % g) != 0 && (tree[2*v + 2] % g) != 0) {
        cnt_not_devide += 2;
        return;
    }
    if ((tree[2*v + 1] % g) != 0) {
        QuerryPos(2*v + 1, g, lx, m);
    } else {
        QuerryPos(2*v + 2, g, m, rx);
    }
}


void QuerrySeg(int v, int g, int lq, int rq, int lx, int rx) {
    if (cnt_not_devide > 1) return;
    if (lq <= lx && rx <= rq) {
        if ((tree[v] % g) != 0) {
            QuerryPos(v, g, lx, rx);
        }
        return;
    }
    if (rx <= lq || rq <= lx) {
        return;
    }
    int m = (lx + rx) / 2;
    QuerrySeg(2*v + 1, g, lq, rq, lx, m);
    QuerrySeg(2*v + 2, g, lq, rq, m, rx);
}
```

Рис. 7: Функции QuerrySeg и QuerryPos

9 Задача I (Валера и запросы)

Эта задача уже очень сложная, её можно решать разными способами, но глобально она упирается в подсчитать количество элементов меньше K на отрезке.

В задаче нужно считать обратную величину - количество отрезков внутри которых нет ни одной точки. Теперь давайте поймём что это за отрезки. Пусть у нас изначально есть отрезки:

$[l_1, r_1]$
$[l_2, r_2]$
...
$[l_n, r_n]$

Причём $l_1 \leq l_2 \leq ... \leq l_n$ Приходит запрос с точками $p_1 \leq l_2 \leq ... \leq l_{cnt}$ и хотим узнать сколько отрезков лежит между точками p_i и p_{i+1} То есть узнать количество l_i таких что $p_i < l_i < p_{i+1}$ и $r_i < p_{i+1}$ Это можно воспринимать так - взять все отрезки у которых начало лежит между двумя точками, и среди всех таких отрезков подсчитать количество правых границ что они меньше координаты правой точки. Построим массив ar длины $10^6 + eps$ что $ar[l_i] = r_i$ а в пустых ячейках положим минус бесконечность. И на этом массиве построим MergeSortTree - как в задаче G. А теперь когда приходит набор точек, то на каждый отрезок между соседними точками делаем запрос к дереву отрезков чтобы подсчитать количество нужных отрезков. Асимптотика получится $O(N \cdot log(N)^2)$

Но можно пойти и другим путём чтобы считать количество чисел на отрезке меньше заданного. Можем воспользоваться идеей из задач про круги, где ставили единички на определённые индексы. Только идексами у нас здесь будут выступать правые границы отрезков, и будем прибавлять единички и вычитать. Для этого надо будет отсортировать все отрезки $[l_i, r_i]$ и прибавить единичку к каждому r_i индексу. Позже мы будем какие-то из них удалять. Собираем все запросы которые надо делать между точками (надо будет сохранить левую, правую границы и номер запроса) Их надо отсортировать по левой границе. И по порядку проходиться по ним. Когда делаем запрос на отрезке [l, r] все отрезки у которых левая граница находится леве должны быть удалены из дерева отрезков. То есть их единички надо убрать, то есть повычитать 1 из их r-х границ. В реализации для этого можно поддерживать индекс самого левого отрезка который лежит правее l (или равен)

Примерно так это может выглядеть в коде

```
for (auto [l, r, ind] : querries) {
    while (sl+1 < n && segs[sl+1].first <= l) {
        Set(root, segs[sl+1].second, -1, 0, maxn);
        sl += 1;
    }
    if (l+1 < r) {
        answers[ind] -= Sum(root, l+1, r, 0, maxn);
    }
}
```

Рис. 8: Обработка запросов