Структуры данных, часть 2

В прошлый раз мы познакомились с sqrt-декомпозицией — структурой данных, которая позволяет хранить набор элементов и делать разные манипуляции с ними: изменять отдельные элементы, изменять отрезки, вычислять какие-то функции на отрезках и так далее. Сегодня же мы изучим другие структуры данных, которые позволяют делать такие же запросы, но асимптотически быстрее (правда, часто от этого расход памяти будет выше).

Дерево отрезков

В первый раз, когда слышится фраза *«дерево отрезков»*, можно подумать, что это про геометрию. Но на самом деле это не так, потому что дерево отрезков (ДО) — это структура данных, которые умеет работать с отрезками в наборах элементов (как и было анонсировано выше). Иногда можно встретить и другие названия этой структуры данных: *«SegmentTree»* и *«дерево сегментов»* — но это всё об одном и том же.

Итак, давайте перейдём к описанию самой структуры. Она будет древовидной и немного напоминать вложенную друг в друга sqrt-декомпозицию, в которой каждый блок состоит из двух элементов. На первом слое сохраним наши элементы набора, на втором сделаем так: первый элемент будет отвечать за первый и второй элементы набора, второй — за третий и четвёртый, третий — за пятый и шестой и так далее. После этого сделаем третий слой, который будет похож на предыдущий: первый элемент отвечает за первую пару элементов второго слоя, второй — за вторую пару и так далее. После этого добавим сверху ещё сколько-то таких же слоёв, пока у нас не станет слой из одного элемента, который называется корнем.

Теперь попробуем узнать полезную информацию про слои такой структуры. На первом слое у нас n элементов набора и каждый из них отвечает за отрезок длины 1. На втором слое элементов уже $\frac{n}{2}$ и каждый элемент отвечает за отрезок длины 2. На третьем слое $\frac{n}{4}$ элементов, каждый отвечает за 4 элемента исходного набора. И так далее, на каждом следующем слое отрезки становятся в два раза больше, а количество элементов в два раза меньше. А следовательно, всего слоёв будет $O(\log n)$, что очень даже приятно, если мы сможем этим воспользоваться.

Теперь давайте оценим количество элементов в дереве отрезков. В самом последнем слое всего 1 элемент, в предпоследнем — 2, потому что корень отвечает ровно за 2 элемента предыдущего слоя. И так далее все слои будут иметь размеры: $1,\ 2,\ 4,\ \ldots,\ 2^i,\ \ldots$. Но согласитесь, что если у нас каждый слой может вмещать 2^k элементов, то хочется, чтобы и в самом нижнем, первом слое (где хранится исходный набор) количество элементов было тоже степенью двойки. Поэтому мы в исходный набор добавим какое-то количество нейтральных элементов так, чтобы размер набора стал степенью двойки. А нейтральный элемент нужно выбирать в зависимости от цели ДО: если мы считаем суммы, то нейтральным будет 0, если минимум — то ∞, и так далее. В конце концов можно просто заполнить все лишние элементы каким-то специальным значением, которые потом можно отдельно обрабатывать.

В прошлом абзаце мы выяснили, что количество элементов в слоях ДО будут степенями двойки: $1,\ 2,\ 4,\ \ldots,\ 2^i,\ \ldots,\ 2^{sz-1},\ 2^{sz}$, где $2^{sz-1} < n \leqslant 2^{sz}$ и исходные элементы хранятся в слое размера 2^{sz}. Теперь хочется оценить, сколько это занимает памяти, для это посчитаем общее количество элементов: $1 + 2 + 4 + \ldots + 2^i + \ldots + 2^{sz} = 2^{sz+1} - 1$ (сумма геометрической прогрессии, или же просто сложение этих чисел в двоичной системе счисления). А если учесть наше неравенство на количество элементов ($2^{sz-1} < n \leqslant 2^{sz}$), то мы можем прийти к выводу, что при $n \approx 2^{sz-1}$ нам требуется $O(2^{sz+1} - 1) \approx O(2^{sz+1}) = O(4 \cdot 2^{sz-1}) = O(4n)$ памяти на хранение ДО, а в другом крайнем случае $n \approx 2^{sz}$ получаем $O(2^{sz+1} - 1) \approx O(2^{sz+1}) = O(2 \cdot 2^{sz}) = O(2n)$. То есть если мы дополняем набор нейтральными элементами до размера равного степени двойки, то мы можем считать, что

$n = 2^{sz}$ и всего элементов в ДО $2n$.

Если в sqrt-декомпозиции было понятно, к какие элементы попадают в один блок, то в ДО переход от родительского элемента к двум дочерним, за которые он отвечает (и обратный переход), не столь очевиден. Но оказывается, что и навигация по ДО достаточно простая, если все элементы хранить в одном большом массиве, слой за слоем, от корня и исходном набору, причём корень будет в ячейке с индексом 1. Тогда слои занимают соответственно ячейки: $[1]$, $[2, 3]$, $[4, 5, 6, 7]$, $\ldots$, $[2^i, 2^i + 1, \ldots, 2^{i+1} - 2, 2^{i+1} - 1]$, $\ldots$, $[2^{sz}, \ldots, 2^{sz+1} - 1]$. И уже даже можно предположить (по первым элементам), что для вершины j дочерними будут $2j$ и $2j + 1$, а родительской — $\left\lfloor \frac{j}{2} \right\rfloor$ (такие скобки — это округление вниз).

Теперь давайте докажем эти простые формулы. Рассмотрим слой, в котором лежит вершина j: $2^i \leqslant j \leqslant 2^{i+1} - 1$, тогда можно сказать, что $j = 2^i + d$, где $0 \leqslant d \leqslant 2^i - 1$ — количество элементов в слое до j. Но понятно, что первые d элементов текущего слоя будут отвечать за первый $2d$ элементов следующего слоя (который больше), а значит элементы $[2^{i+1}; 2^{i+1} + 2d - 1]$ уже имеет родительский, а вот для j дочерними будут $2^{i+1} + 2d$ и $2^{i+1} + 2d + 1$. Остаётся лишь подставить все числа: $2^{i+1} + 2d = 2^{i+1} + 2(j - 2^i) = 2^{i+1} + 2j - 2^{i+1} = 2j$ и аналогично $2^{i+1} + 2d + 1 = 2^{i+1} + 2(j - 2^i) + 1 = 2j + 1$. То есть мы теперь знаем, что для j дочерними элементами являются $2j$ и $2j + 1$ и тогда формула перехода к родительскому элементу понятна, ведь $\left\lfloor \frac{2j}{2} \right\rfloor = j$ и $\left\lfloor \frac{2j+1}{2} \right\rfloor = j$, что и требовалась доказать.

Теперь узнав много фактов о структуре ДО, мы можем уже и перейти к работе с ним.

Функция на отрезке. Пусть нам дан какой-то набор данных и требуется вычислять на его отрезках минимум (или другую функцию, в которой не важен порядок вычисления). И оказывается, что в ДО это реализовывается очень удобно: для каждой вершины дерева предподсчитаем функцию на отрезке, за который вершина отвечает, а после этого из таких частичных ответов будем получать ответ для нужного отрезка.

При этом понятно, что предподсчёт занимает $O(n)$ времени, ведь половина значений уже вычислена (если у отрезка длина 1, то этот элемент и будет минимальным), а вторую половину можно вычислить, взяв минимум из значений, которые получились у дочерних вершин. Правда при этом для вычисления значения в вершине необходимо, чтобы мы уже знали значения для её детей, но это тоже не сложно, ведь для этого достаточно вычислять значения для вершин в обратном порядке.

Теперь пусть мы хотим обработать запрос вычисления минимума на отрезке $[l; r]$. Тогда заведём два указателя `i = l, j = r`, которые мы будем двигать, постепенно вычисляя минимум на всём отрезке (изначально берём его за ∞). Двигать же указатели мы будем по простому правилу: если можем перейти к родительской вершине, то делаем это, а иначе обработаем значение, в которое указывает указатель и сдвинем границу, после чего уже перейдём к родительской вершине. Если вспомнить структуры ДО, то становится понятно, что если `i % 2 = 1`, то это правый ребёнок своего родителя, а значит нужно обработать элемент `i` и сузить отрезок: `i = i + 1`. Аналогично делаем с правой границей, если вдруг она является левый ребёнком: при `j % 2 = 0` обрабатываем `j` и делаем `j = j - 1`. После таких операций `i` будет точно указывать на левого сына, а `j` — на правого, а значит мы можем перейти к родительским вершинам `i /= 2, j /= 2`, а они будут покрывать тот же отрезок, что и до перехода.

Остаётся лишь только понять, когда такие переходы к родительским вершинам нужно остановить. Понятно, что в нашем дереве указатели двигаются так, что между ними всё время уменьшается расстояние, при этом не слишком резко (указатели не перепрыгивают друг через друга. если до этого не были рядом). Поэтому будем рассматривать только когда отрезки стали маленькими. Пусть у нас на каком-то шаге $i + 2 = j$, тогда i и j одной чётности, а значит из них сдвинется ровно один и мы получим отрезок из двух элементов. Теперь пусть $i + 1 = j$, при этом могут быть два варианта: если $i \,\%\, 2 = 1$, то наш алгоритм сдвинет границы $i = i + 1$ и $j = j - 1$, перейдёт к разным родительским вершинам (то есть получит $j < i$), и можно завершить работу алгоритма, ведь весь отрезок уже обработался. Если же $i \,\%\, 2 = 0$, то i и j указывают правильно, поэтому мы перейдём к их общему родителю, который обработается только один раз согласно алгоритму выше и после этого мы поднимемся ещё на один уровень вверх и получим $j < i$, то есть алгоритм можно завершать.

А теперь, по длинному описанию выше, можно легко написать ДО:

```
1  const int N = 1 << 17;  // 2^17 = 131072, это точно больше, чем обрабатываемых элементов
```

```
struct SegmentTree{
    int st[2 * N];
    SegmentTree(vector<int> v){
        for(int i = 0; i < v.size(); ++i) st[N + i] = v[i];              // сохранение данных
        for(int i = v.size(); i < N; ++i) st[N + i] = INF;               // нейтральные элементы
        for(int i = N - 1; i > 0; --i) st[i] = min(st[2 * i], st[2 * i + 1]);   // предподсчёт
    }

    int get_min(int l, int r){
        l += N, r += N;                                  // сдвинулись в самый нижний слой
        int ans = INF;                                   // текущей ответ
        while(l <= r){
            if (l & 1) ans = min(ans, st[l++]);          // обработали и сдвинули l
            if (!(r & 1)) ans = min(ans, st[r--]);       // обработали и сдвинули r
            l /= 2, r /= 2;                              // перешли к родителям
        }
        return ans;
    }
};
```

Легко можно видеть, что строится ДО за линейное время, а на один запрос уходит $O(\log n)$, потому что всего столько уровней, а на каждом нам нужно выполнить константное число операций.

Обход вниз. Предположим, что нам также дан набор a из 0 и 1 и на каждый запрос требуется находить позицию k-ой единицы. Понятно, что в этой задаче у нас не получится подниматься вверх, начав с какого-то отрезка, ведь у нас нет этого отрезка :) Поэтому будем двигаться вниз, начиная с корня.

Но это движение, в целом, тоже довольно естественно: пусть мы сейчас находимся в какой-то вершине, тогда если в левом подотрезке достаточное количество единиц, то мы перейдём в него, а иначе — перейдём в правый, учтя количество единиц левого подотрезка. Такими шагами, если искомая единица была, то мы когда-то спустимся к ней, а если её не было, то такой случай можно обработать в самом начале, ведь мы можем посчитать общее число единиц. Теперь заметим, что поскольку все числа либо 0, либо 1, то количество единиц на отрезке как раз равно сумме цифр отрезка. То есть в каждой вершине достаточно хранить сумму подотрезка, поэтому приведём лишь код обхода ДО:

```
int get_k(int k){                                                // внутри класса
    int p = 1;                                                   // корень
    while(p < N){                                                // пока не нижний уровень
        if (st[2 * p] < k) { k -= st[2 * p]; p = 2 * p + 1; }    // переход вправо
        else p *= 2;                                             // переход влево
    }
    return p - N;                                                // вернуть обычный индекс
}
```

Понятно, что и такой запрос выполняется за $O(\log n)$, потому что на каждом слое константное число операций.

Обновление элементов и отрезков. Кроме получающих запросов, ДО также может поддерживать и обновляющие запросы. Если требуется обновить всего один элемент, то здесь всё просто: мы его обновим и после этого будем идти по всем его родителям снизу вверх и пересчитывать значения в них. Поэтому интереснее рассмотреть запросы, обновляющие отрезок (такие запросы поддерживает и sqrt-декомпозиция, что читателю требуется придумать самому).

Пусть нам нужно получать значение элемента и при этом мы можем прибавлять ко всему отрезку одно значение. Тогда построим ДО, в котором содержатся прибавления: на каждый запрос обновления мы будем также обходить дерево снизу вверх, при этом перед операцией `l = l + 1` и `r = r - 1` будем помечать, сколько добавилось к подотрезку вершины l (или r). Когда же нам понадобится определённый элемент, то мы должны будем просуммировать значение этого элемента и всех добавлений в родительских вершинах.

Так мы научились и обновлять ДО за $O(\log n)$.

Разреженная таблица

Следующая структура данных — это *«sparse table»* или *«разреженная таблица»*. Нужна она для быстрого поиска минимального значения на отрезке, при этом обновлять значения нельзя. Казалось бы, зачем вообще нужна такая структура, если есть ДО? А вот и нет, во-первых, оказывается, разреженная таблица очень даже хороша. если запросов будет достаточно много, а во-вторых — иногда все возможности ДО не нужны, а разреженной таблицы хватает.

Эта структура данных основывается на таком свойстве, что для любых длин отрезка x найдётся нужная степень двойки 2^t, где $t \in \mathbb{Z}$, такая, что $2^t \leqslant x \leqslant 2^t + 2^t$. То есть для нахождения минимума на отрезке нам достаточно разбить его на два отрезка, с размерами 2^t, а их пересечение даст нам весь искомый отрезок.

Теперь, когда мы знаем основную идею, можно и перейти к более подробному описанию. Пусть имеется n элементов, тогда для всех $2^i \leqslant n$ почитаем ответы на всех запросах длины 2^i. Пусть $a_{i,j}$ — ответ на запрос минимума на отрезке $[j; j+2^i-1]$. Тогда понятно, что при $i = 0$ у нас отрезки длины 1 и ответы для них мы знаем из входных данных, а иначе $a_{i+1,j} = \min(a_{i,j}, a_{i,j+2^i})$. Так мы сможем предподсчитать все нужные ответы, на что уйдёт $O(n \log n)$ времени, ведь длина отрезка каждый раз увеличивается в два раза, а всего отрезков одной длины не больше, чем элементов в наборе.

Когда же нам требуется найти минимум на $[l; r]$, то длина отрезка будет $x = r - l + 1$, после чего мы найдём $2^t \leqslant x \leqslant 2^t + 2^t$. И ответом на запрос станет $ans = \min(a_{t,l}, a_{t,r-2^t+1})$, то есть каждый запрос обрабатывается за $O(1)$.

Теперь можем реализовать это в коде:

```
struct SparseTable{
    vector<vector<int>> v;
    SparseTable(vector<int> a){
        v.push_back({});
        for(int x : a) v[0].push_back(x);                       // i = 0
        for(int i = 1; (1 << i) < a.size(); ++i){               // i > 0
            v.push_back({});
            int ln = (1 << i - 1), ln2 = (1 << i);              // ln2 - длина отрезка
            for(int j = 0; j + ln2 <= a.size(); ++j){           // все отрезки длины ln2
                v.back().push_back(min(v[v.size() - 2][j], v[v.size() - 2][j + ln]));
            }
        }
    }

    int get_min(int l, int r){
        int t = log2(r - l + 1);                                // подходящее t
        return min(v[t][l], v[t][r - (1 << t) + 1]);            // получение ответа
    }
};
```

Интересно, что разреженная таблица - настолько специфичная структура данных, что встречается не во всех книгах, а также не имеет (известных автору) усложнений :)

Дерево Фенвика

А теперь изучим ещё одну структуру данных (последнюю в этом блоке) — *«дерево Фенвика»* или *«FenwickTree»* — которая умеет быстро вычислять обратимую функцию на динамических данных. Если говорить менее формальным языком, то обратимые функции — это сумма и произведение, а динамичность данных означает, что структура поддерживает поэлементное изменение. Для определённости будем считать, что мы хотим вычислять сумму на отрезках.

Идея у дерева Фенвика похожа на предыдущие структуры: предподсчитаем какие-то значения, с помощью которых нам будет легко отвечать на запросы, а при обновлении часть значений будем пересчитывать. Пусть нам дан набор $a_0,\ a_1,\ \ldots,\ a_{n-1}$, тогда мы предподсчитаем значения по формуле $t_i = \sum\limits_{k=F(i)}^{i} a_k$, где самым важным элементом является функция $F(i)$, из-за которой и получается быстрая структура данных. Пока мы не будем выписывать саму функцию, а лишь заметим, что $F(i) \leqslant i$.

Теперь научимся отвечать на запрос суммы на отрезке $[l; r]$. Но понятно, что для этого нам достаточно вычислить $sum(r) - sum(l-1)$, где $sum(x)$ — сумма первых элементов до x включительно. Так что остаётся лишь найти сумму на отрезке $[0; r]$ и понятно, как это делать: сначала возьмём сумму элементов в $[F(r); r]$, она посчитана в t_r, после этого сделаем $r' = F(r)-1$ и к ответу добавится отрезок $[F(r'); F(r) - 1]$, так мы будем продолжать переходить к предыдущему отрезку, пока не выйдем за границы набора (при $r < 0$ нужно остановиться).

Теперь остаётся лишь научиться обновлять значение a_k, а для этого требуется знать, на какие t_i влияет a_k, то есть нужно решить неравенство $F(i) \leqslant k \leqslant i$. Одно из решений этого неравенства при $i = k$ очевидно, и если читателю очень интересно самостоятельно получить все решение этого неравенства, то он может ознакомиться с информацией в интернете. А мы лишь будем использовать готовую формулу: нужно взять $F(i) = i \ \&\ (i + 1)$, тогда индексы, использующие a_k будут $i_0 = k$, а после этого $i_{s+1} = i_s \mid (i_s + 1)$. При этом остановиться нужно будет, если очередное i стало слишком большим, то есть при $i \geqslant n$.

Теперь кажется, что если поверить в верность формул $F(i) = i \ \&\ (i + 1)$ и $i_{s+1} = i_s \mid (i_s + 1)$, то мы уже сможем написать дерево Фенвика:

```
struct FenwickTree{
    vector<int> t;
    int n;

    FenwickTree(vector<int> a){
        n = a.size();
        t.assign(n, 0);                                  // конструирование -
        for(int i = 0; i < a.size(); ++i) add(i, a[i]);  // это много обновлений
    }

    void add(int p, int d){
        for (; p < n; p = (p | (p + 1))) t[p] += d;      // запомнить p | (p + 1)
    }

    int sum(int r){
        int result = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1) result += t[r];  // запомнить r & (r + 1),
        return result;                                         // понять -1 из формулы t[i]
    }

    int sum(int l, int r){
        return sum(r) - sum(l - 1);
    }
};
```

Остаётся лишь сказать, что из-за битовых операций сложность операций с деревом Фенвика это $O(\log n)$, потому что в битовом представлении у нас в конце скапливаются одинаковые знаки, которых с каждым шагом становится хотя бы на 1 больше (при операции & растёт число нулей, а при | — количество единиц). Кроме того, дерево Фенвика занимает $O(n)$ памяти, что асимптотически также, как и у дерева отрезков, но всё же ДФ хранит только n элементов, а в ДО их от $2n$ до $4n$ — что больше. И последнее преимущество ДФ — краткость кода.

Выше была представлена стандартная реализация ДФ, а ниже вы можете увидеть вариант, в котором формулы немного другие, из-за чего такую реализацию будет проще запомнить:

```
struct FenwickTree{
    vector<int> t;
    int n;

    FenwickTree(vector<int> a){
        n = a.size() + 1;                                   // t[0] - не используем
        t.assign(n, 0);                                     // конструирование -
        for(int i = 0; i < a.size(); ++i) add(i + 1, a[i]);     // это много обновлений
    }

    void add(int p, int d){
        for (; p < n; p += p & -p) t[p] += d;               // запомнить p & -p
    }

    int sum(int r){
        int result = 0;
        for (; r > 0; r -= r & -r) result += t[r];          // запомнить r & -r,
        return result;
    }

    int sum(int l, int r){
        return sum(r) - sum(l - 1);
    }
};
```

Интересно, что ДФ достаточно просто расширяется на многомерные случаи: для этого достаточно сделать несколько вложенных циклов. И тогда для двумерного случая функция нахождения суммы будет возвращать сумму в прямоугольнике, начинающемся в углу и заканчивающемся в заданной клетке. А чтобы находить сумму в произвольном прямоугольнике, придётся использовать формулу из главы про двумерную префиксную сумму.

Сегодня мы изучили много структур данных, каждая из которых хорошо применима в конкретных случаях. Но если не хочется запоминать все структуры, то можно всегда использовать ДО, так как оно имеет меньше всего ограничений.