Алгоритмы и структуры данных
Модификации дерева Фенвика

Выполнил:

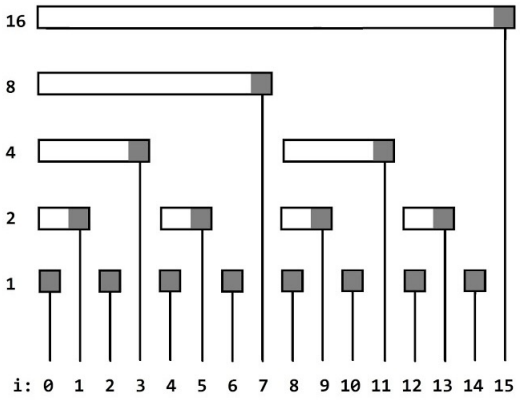
Шульпин Егор М3232

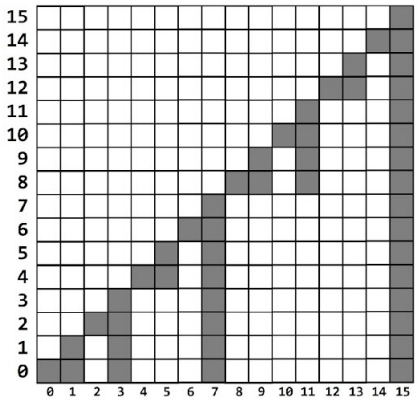
Предисловие

Дерево Фенвика (или Binary Indexed Tree) - структура данных, позволяющая за $O(\log n)$ (и лучшей константой, чем в дерево отрезков) выполнять следующие операции:

- Изменить определенный элемент массива.
- Получить значение ассоциативной, коммутативной и обратимой операции на отрезке $[l, r]$.

Хотя дерево Фенвика и имеет такую же асимптотику по памяти, что и дерево отрезков ($O(n)$), но все же имеет гораздо меньшую константу, тем более позволяя не тратить память на исходный массив.





Реализация:

```
class fenwick {
public:
    explicit fenwick(const int n) {
        this->n = n;
        this->tree.resize(n);
    }

    void add(int i, int value) {
        for ( ; i < n; i = (i | (i + 1))) {
            tree[i] += value;
        }
    }

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

private:
    int n;
    vector<int> tree;

    int sum(int i) {
        int result = 0;
        for ( ; i >= 0; i = (i & (i + 1)) - 1) {
            result += tree[i];
        }
        return result;
    }
};
```

Некоммутативные операции

Коммутативная операция - такая операция $\circ$, для которой выполнено условие перестановки аргументов:

$$a \circ b = b \circ a.$$

Таким образом, для некоммутативной операции важен порядок, например, для функции степени числа или умножения матриц. По стандартной реализации видно, что данное условие не будет выполено.

Обновление элемента в случае некоммутативной операции:

Давайте рассмотрим возможность изменения элемента в дереве Фенвика в случае, когда некая операция $\circ$ - некоммутативна.

Пусть в некоторый момент времени в дереве по индексу i расположен элемент $t_i = a_i \circ \cdots \circ a_j \circ \dots$. Мы хотим воспользоваться операцией изменения элемента a_j. Рассмотрим две ситуации:

- $t_i = a_i \circ \cdots \circ a_j \circ \cdots \circ x$ - неверно, когда функция некоммутативна;
- $t_i = a_i \circ \cdots \circ a_j \circ x \circ \dots$ - верно, так как порядок операций соблюден.

Отсюда следует вывод, что для получения корректного ответа, нам требуется убрать отрезок до изменяемого элемента, посчитать операцию, а затем снова учесть вырезанный отрезок.

Получаем, что результат операции изменяется таким образом:

- Изначально $s_i = a_1 \circ \cdots \circ a_i$ - результат операций на префиксе i.
- $s_{ij} = s_i^{-1} \circ s_j$ - результат выполнения операции на отрезке $[i, j]$.
- Тогда $t_j = t_j \circ s_{ij}^{-1} \circ x \circ s_{ij}$ - формула изменения элемента в случае некоммутативности.

Давайте рассмотрим короткое доказательство:

Так как верно $(a \circ b)^{-1} = b^{-1} \circ a^{-1}$, то:

- Новое значение $s_{ij} = (s_i \circ x)^{-1} \circ s_j = x^{-1} \circ s_i^{-1} \circ s_j = x^{-1} \circ s_{ij}$.
- Получаем верные значения: $s_{ij}^{-1} \circ x \circ s_{ij} = (x^{-1} \circ s_{ij})^{-1} \circ x \circ x^{-1} \circ s_{ij} = s_{ij}^{-1} \circ x \circ x \circ x^{-1} \circ s_{ij} = s_{ij}^{-1} \circ x \circ s_{ij}$.

Данный подход требует $O(n)$ памяти с возможностью in-place изменений, однако время работы каждой операции стало $O((\log n)^2)$, так как для каждого из $\log n$ элементов мы делаем $\log n$ вычислений.

Пример:

Задан исходный массив матриц: $a_0 = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$, $a_1 = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$, $a_2 = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}$, $a_3 = \begin{bmatrix} 0 & 2 \\ 1 & 2 \end{bmatrix}$.

Нас интересует некоммутативная операция умножения матриц на отрезке. Тогда элементы дерева Фенвика имеют вид: $t_0 = a_0$, $t_1 = a_0 \circ a_1$, $t_2 = a_2$, $t_3 = a_0 \circ a_1 \circ a_2 \circ a_3$.

Пусть нужно изменить элемент a_2 на $a_2' = a_2 \circ x$, где $x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. Тогда, воспользовавшись полученной формулой, получаем:

- $t_2' = t_2 \circ s_{2,2}^{-1} \circ x \circ s_{2,2} = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix} \circ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 2 & 4 \end{bmatrix}$.
- $t_3' = t_3 \circ s_{2,3}^{-1} \circ x \circ s_{2,3} = t_3 \circ (s_2^{-1} \circ s_3)^{-1} \circ x \circ (s_2^{-1} \circ s_3)$, тогда, получив значения на префиксах и посчитав обратные значения, получает ответ $t_3' = \begin{bmatrix} 12 & 38 \\ 8 & 24 \end{bmatrix}$.

Необратимые операции

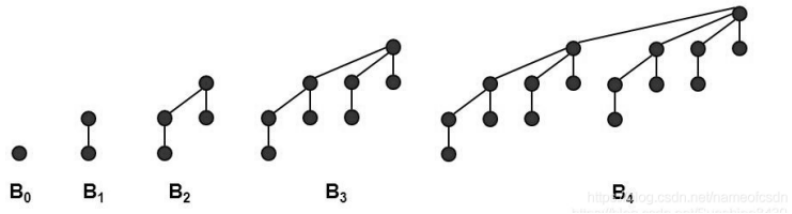
Необходимым условием для функции, используемой в дереве Фенвика, является обратимость, то есть для каждого элемента a существует такой элемент a^{-1}, что $a \circ a^{-1} = a^{-1} \circ a = e$ - нейтральный элемент.

Рассмотрим операцию минимума на отрезке $[i, j]$: стандартная реализация дерева Фенвика не позволяет вычислять операции такого рода. Такая же проблема, например, есть и у префиксных сумм.

Основная идея реализации:

Основным изменением является использование биномиальных деревьев. В данной модификации именно они называются BIT.

Биномиальное дерево - дерево, элемент B_{i+1} которого получается путем присоединения B_i к такому же B_i. Это можно увидеть на изображении:



Можно увидеть, что название биномиальное оно получило именно из-за того, что дерево порядка k имеет $\binom{k}{d}$ вершин на глубине d. А также стоит отметить, что дерево порядка k состоит из 2^k вершин, а значит подъем с самого низа до корня будет занимать $O(\log n)$ времени.

Все это к тому, что дерево Фенвика является биномиальным. Это не очевидный факт, однако если представлять для вершин родительско-дочерние отношения, то получится такой результат. Это показывает левое дерево на изображение ниже. Стоит отметить, что правое дерево отображает отношение $B_0, B_1, \ldots$.

Пусть есть запрос на отрезке $[l, r]$. Теперь мы не будем делать два запроса от $l - 1$ и r. Основная идея состоит в том, чтобы начать с узла l в BIT1, подняться вверх, пока родительский узел меньше r, использовать значение BIT2, начать с узла r в BIT2, подняться вверх, пока родительский узел больше l, и использовать значение BIT1.

Пример:

Пусть на дереве, изображенном выше, нужно посчитать операцию на отрезке $[5, 13]$. Начинаем подниматься с вершины 5 в BIT1, пройдя вершины 5 и 6. Мы остановимся на 8, так как его родитель больше 13. То есть в нашем ответе будут учавствовать значения BIT2[5], BIT2[6]. Теперь начинаем подниматься из вершины 13 в BIT2. Здесь мы используем вершины 13 и 12, останавливаясь на вершине 8 по той же причине. Берем в наш ответ BIT1[13], BIT[12]. Итого наш ответ будет результат операции с использованием значений BIT1[12], BIT1[13], BIT2[5] и BIT[6].

Реализация:

```
class fenwick {
public:
    explicit fenwick(const int n) {
        this->n = n;
        this->v.resize(n + 1, INT_MAX);
        this->BIT1.resize(n + 1, INT_MAX);
        this->BIT2.resize(n + 1, INT_MAX);
    }

    void update(int i, int val) {
        int t = v[i] = val;
        for (int j = i, l = i - 1, r = i + 1; j <= n; j += f(j)) {
            while (l > j - f(j)) {
                t = min(BIT1[l], t);
                l -= f(l);
            }
            while (r < j) {
                t = min(t, BIT2[r]);
                r += f(r);
            }
            BIT1[j] = i ^ j ? min(t, v[j]) : t;
        }
        t = val;
        for (int j = i, l = i - 1, r = i + 1; j; j -= f(j)) {
            while (l > j) {
                t = min(BIT1[l], t);
                l -= f(l);
            }
            while (r <= n && r < j + f(j)) {
                t = min(t, BIT2[r]);
                r += f(r);
            }
            BIT2[j] = i ^ j ? min(v[j], t) : t;
        }
    }

    int query(int l, int r) {
        int i;
        int L = INT_MAX;
        int R = INT_MAX;
        for (i = l; i + f(i) <= r; i += f(i)) {
            L = min(L, BIT2[i]);
        }
        for (i = r; i - f(i) >= l; i -= f(i)) {
            R = min(BIT1[i], R);
        }
        return min(min(L, v[i]), R);
    }

private:
    int n;
    vector<int> v, BIT1, BIT2;

    static int f(int i) {
        return i & -i;
    }
};
```

Итоги

Стандартное дерево Фенвика

Как я упоминал выше, дерево Фенвика имеет большое преимущество над деревом отрезков, когда мы рассматриваем операции, подходящие под все условия. Например, для операции суммы на отрезке, дерево Фенвика, за счет лучшей константы, очень быстрых битовых операций и гораздо меньшей требуемой памяти. В интернете можно найти много сранений, где приводятся конкретные значения времени исполнения на больших наборах данных. Стоит упомянуть, что в основном здесь идет речь о рекурсивном дереве отрезков. Существует итеративная реализация, которая работает гораздо быстрее, однако и она проигрывает дереву Фенвика. Это лишь напоминание о преимуществе дерева Фенвика. Давайте теперь перейдем к рассмотренным модификациям.

Дерево Фенвика для некоммутативных операций

Данная модификация хорошо подходит, как пример того, что дерево Фенвика можно менять под нужные нам операции. Здесь мы сталкнулись с ухудшением асимптотики до $O((\log n)^2)$, однако этого не избежать. Например, для операции умножения матриц на отрезке дерево дерево отрезков тоже работает только в случае, когда нет изменения нужного нам элемента. Таким образом, можно сказать, что это интересная модификация, построенная на одном факте о некоммутативной функции. Она все еще довольно проста в реализации и имеет хорошую асимптотику, позволяя делать in-place изменения.

Дерево Фенвика для необратимых операций

На мой взгляд, это самая интересная модификация для дерева Фенвика. В интернете можно найти реализации с ухудшением асимптотики до $O((\log n)^2)$. При этом код будет выглядеть, примерно, как в предыдущей модификации. Модификация с использованием биномиальных деревьев, наоборот, позволяет выполнять операции за $O(\log n)$, при этом имея очень трудную реализацию, представленную выше. В статье, откуда была взята основная идея, упоминалось о сравнении времени работы и затраченной памяти с разными реализациями дерева отрезков. Если требуемая память, практически совпадает в обоих реализациях (так как мы добавили дополнительные массивы для биномиальных деревьев), то время исполнения, наоборот, сильно отличается в лучшую сторону. Ниже приведен пример из статьи:

Operation Type	Segment(Range) Tree time	Binary Indexed Tree time
Build 100K array	0.0009 s	0.0006 s
10M Updates	1.274 s	0.672 s
10M Queries	2.397 s	0.551 s

Я решил сам проверить разницу в скорости этих структур. Для этого я нашел тестовую задачу, на которой решил проверить время исполнения модификации дерева Фенвика и рекурсивного дерева отрезков (стандартная реализация). Полученный результат можно увидеть ниже:

DL4x	B - Дерево отрезков на минимум	C++20 (GCC 13-64)	Полное решение	171 мс	3900 КБ
DL4x	B - Дерево отрезков на минимум	C++20 (GCC 13-64)	Превышено ограничение времени на тесте 56	1000 мс	100 КБ

Итоговые результаты говорят сами за себя: модификация дерева Фенвика действительно работает быстрее дерева отрезков. Однако, по-моему мнению, данная модификация сложна в реализации и намного труднее в понимании, чем любая реализация дерева отрезков.

Источники:

https://neerc.ifmo.ru/wiki/index.php?title=Дерево_Фенвика_для_некоммутативных_операций
https://blog.naver.com/jinhan814/222627110424
https://ioinformatics.org/journal/v9_2015_39_44.pdf