Теперь можно начинать.

Всё о дереве Фенвика и даже больше

Сапожников Денис

Содержание

1 Дерево Фенвика

1.1 Задача

Пусть у есть есть ассоциативная коммутативная обратимая операция f, которую здесь и далее будем обозначать за $+$. Если написать это по-русски, то:

1. $a + b = b + a$ — коммутативность
2. $(a + b) + c = a + (b + c)$ — ассоциативность
3. Для любого элемента a существует обратный $-a$, такой что $a + (-a) = 0$ (0 здесь — это так называемый нейтральный элемент: $a + 0 = 0 + a = a$; для умножение это будет 1, для сложения 0 и т.д.).

Мы хотим уметь в 2 типа запросов:

- $add\ i\ x$ — увеличить значение в позиции i на x
- $sum\ l\ r$ — узнать сумму на отрезке $[l; r]$

Второй запрос за счёт обратимости сводится к сумме на префиксе: $sum(l, r) = sum(1, r) - sum(1, l)$.

1.2 Дерево Фенвика

1.2.1 Идея алгоритма

Пусть существует некоторая функция $F : \mathbb{Z}_+ \to \mathbb{Z}_+$ и $\forall x \in \mathbb{Z}_+ : F(x) \leq x$. Введём хитрую сумму:

$$t_k = \sum_{i=F(k)}^{k} a_i$$

где a — это наш массив, с которым мы хотим делать операции sum и add.

На самом деле мы уже можем написать псевдокод, который будет по массиву t отвечать на оба запроса:

```
int sum(int i) {
    int res = 0;
    for (; i >= 0; i = F(i) - 1)
        res += t[i];
    return res;
}
```

```
void add(int i, int x) {
    for (; i < n; i лежит в каком-то отрезке [F(j)...j])
        t[i] += x;
}

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

Чтобы это работало быстро от функции F требуется 2 условия:

- Каждый префикс раскладывается в $O(\log n)$ отрезков $[F(j); j]$.
- Каждый элемент лежит в $O(\log n)$ отрезков $[F(j); j]$.

1.2.2 Убираем последний бит

Рассмотрим функцию F, которая убирает последний бит числа. Тогда заметим, что каждый отрезок раскладывается в $O(\text{количество единиц в числе})$ отрезков, что равно $O(\log n)$.

Давайте разберёмся, как вычислять функцию F:

1. $F(x) = x - (x\& - x) + 1$. Это работает за счёт того, как отрицательные числа хранятся в компьютере. потребует знания, как в компьютерах хранятся целые числа. Чтобы процессор не сжигал лишние такты, проверяя знак числа при арифметических операциях, их хранят как бы по модулю 2^k, а первый бит отвечает за знак (0 для положительных и 1 для отрицательных). Поэтому когда мы хотим узнать, как выглядит отрицательное число, нужно его вычесть из нуля: $-x = 0 - x = 2^k - x$.

   Как будет выглядеть $-x$ в битовой записи? Ответ можно мысленно разделить на три блока:

   - Первые сколько-то (возможно, нисколько) нулей с конца числа x ими же в ответе и останутся.
   - Потом, ровно на самом младшем единичном бите x, мы «займём» много единиц, так что весь префикс станет единицами. В ответе на этом месте точно будет единица.
   - Потом отменятся ровно те биты из этого префикса, которые были единицами в исходном числе.

   Пример:
   $$\begin{aligned} +90 &= 2 + 8 + 16 + 64 = 0\,10110_2 \\ -90 &= 00000_2 - 10110_2 = 1\,01010_2 \\ (+90)\&(-90) &= 0\,00010_2 \end{aligned}$$

2. $F(x) = x - (((x \oplus (x-1)) + 1) >> 1) + 1$. Только не бейте. На самом деле это проще понять, чем прошлую формулу, но выглядит она стрёмно. $x \oplus (x-1)$ поставит в конце все единицы, начиная с последнего бита включительно, поэтому если мы прибавим 1, то получим бит, который будет старше на один, чем младший бит в x, собственно поэтому нам нужно ещё сдвинуть результат битово вправо.

3. $F(x) = x\&(x-1)+1$ — самая известная формула, чтобы убрать последний бит в числе. Почему мы будем пользоваться не ей, а первой? Потому что далее у нас появится симметрия в формулах.

Осталось разобраться, а в скольких отрезках лежит число i, а ещё лучше определить, в каких именно. То есть, нам нужно найти все такие $j : F(j) \le i < j$. Запишем это в двоичной системе счисления:

$$
\begin{array}{rcccccccc}
F(j) & = & a & b & c & 0 & 0 & 0 & 0 & 0 \\
i & = & a & b & c & 0 & * & * & * & * \\
j & = & a & b & c & 1 & 0 & 0 & 0 & 0
\end{array}
$$

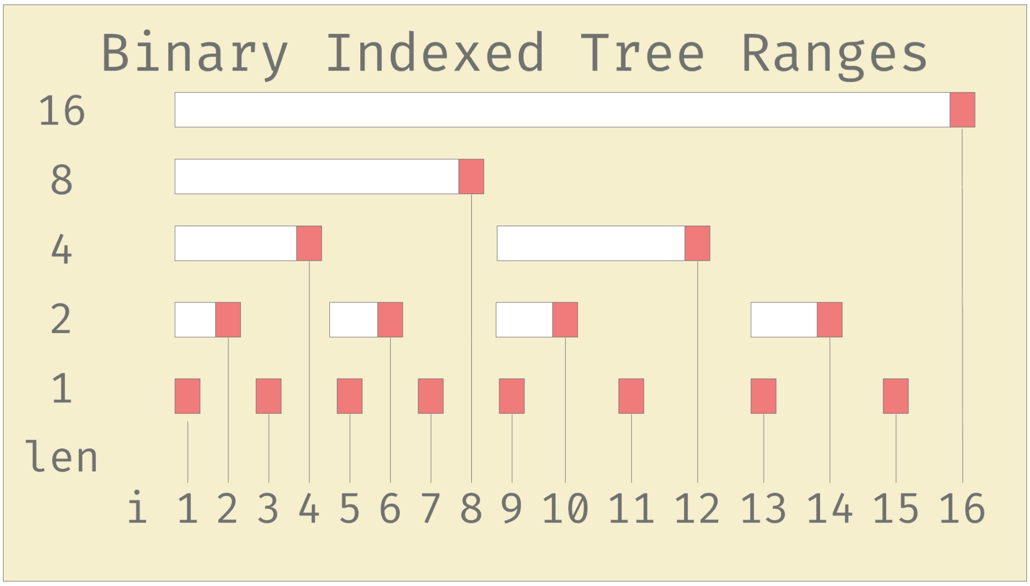
А теперь осознаём, что j — это все такие числа, что в k-м бите в числе i стоит 0, в k-м бите в j стоит 1, после k-го стоят нули, а префиксы до k-го совпадают. То есть таких j ровно O(количество нулей), что равно $O(\log n)$.

Более того, пусть у нас есть число i И мы хоти получить минимальный отрезок, который покрывает i. Тогда мы заменим последний 0 на 1, а всё что было после — заменим на 0. Эта операция эквивалентна прибавлению самого младшего бита к числу i:

$$
\begin{array}{rccccccccc}
F(j) & = & a & b & c & 0 & 0 & 0 & 0 & 0 \\
i & = & a & b & c & 0 & 1 & 1 & 0 & 0 \\
i\&-i & = & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
j & = & a & b & c & 1 & 0 & 0 & 0 & 0
\end{array}
$$

Тогда введём функцию $G(i) = i + (i\& - i)$, которая в каком-то смысле будет обратной к F: в дереве Фенвика с помощью такой операции мы идём в предка вершины i.

Вот мы говорим дерево-дерево, а какое оно, это дерево? А вот такое!



1.2.3 Реализация

Так как $[F(0); 0]$ — некорректный отрезок, то будем существовать в 1-индексации.

```
int sum(int i) {
  int res = 0;
  for (; i > 0; i -= x & -x)
    res += t[i];
  return res;
}

int add(int i, int x) {
  for (; i <= n; i += g(i))
    t[i] += x;
}
```

1.2.4 Спуск по Фенвику

Часто бывает, что нужно найти минимальное k: $sum(1, k) \geq x$. Мы могли бы сделать это бинарным поиском за $O(\log^2)$ на запрос, но можно обойтись $O(\log n)$. Теперь то у нас есть картинка и по ней легко понять, что мы можем либо взять отрезок длиной 2^{k-1}, либо не брать и собственно это весь спуск по дереву.

```
int upper_bound (int s) {
  int k = 0;
  for (int l = LOGN - 1; l >= 0; l--)
    if (k + (1<<l) <= n && t[k + (1<<l)] < s) {
      k += (1<<l);
      s -= t[k];
    }
  return k;
}
```

1.2.5 Убираем сразу несколько последних бит

Только что мы разобрали одну функцию F, оказывается есть вторая! Пусть теперь F убирает все последние подряд идущие единичные биты. Функция, которая это делает — это функция $F(x) = x\&(x+1)$:

$$\begin{array}{lcccccccc} x & = & a & b & c & 0 & 1 & 1 & 1 & 1 \\ x+1 & = & a & b & c & 1 & 0 & 0 & 0 & 0 \\ x\&(x+1) & = & a & b & c & 0 & 0 & 0 & 0 & 0 \end{array}$$

Тогда количество скачков по $i \to F(i) - 1 \to \ldots$ будет равно $O(\text{количество групп единц в числе}) = O(\log n)$.

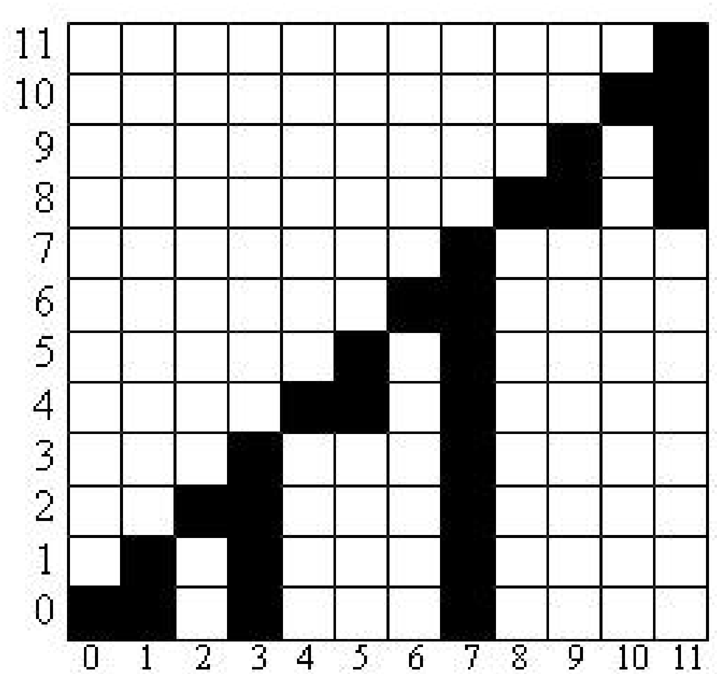
Теперь мы хотим найти все $j : F(j) \le i < j$, для этого опять посмотрим на двоичную запись:

$$\begin{array}{lcccccccc} F(j) & = & a & b & c & 0 & 0 & 0 & 0 & 0 \\ i & = & a & b & c & 0 & * & * & * & * \\ j & = & a & b & c & 0 & 1 & 1 & 1 & 1 \end{array}$$

Из такой картинки следует, что j — это некоторый префикс i, заканчивающийся нулём и суффикс из единиц, таких чисел $O(\text{количество нулевых бит}) = O(\log n)$.

Заметим, что функция $G(i) = i|(i+1)$ ставит последний нулевой бит в значение 1, а значит мы можем применить G последовательно несколько раз и каждый раз будем получать подходящее j.

И картинка Фенвика для счастья:



1.2.6 Реализация 2.0

В данном случае мы можем использовать t_0, поэтому тут удобно жить в 0-индексации, мы же с вами программисты.

```
int f(int x) { return x & (x + 1); }
int g(int x) { return x | (x + 1); }

int sum(int i) {
  int res = 0;
  for (; i >= 0; i = f(i) - 1)
    res += t[i];
  return res;
}

int add(int i, int x) {
  for (; i < n; i = g(i))
    t[i] += x;
}
```

Однако, с такой функцией уже не получится спускаться по Фенвику.

1.3 Многомерный Фенвик

Эту структуру все любят за то, что чтобы написать двумерное дерево Фенвика, нужно дописать лишнюю строку, а именно:

```
int f(int x) { return x & (x + 1); }
int g(int x) { return x | (x + 1); }

int sum(int ii, int jj) {
  int res = 0;
  for (int i = ii; i >= 0; i = f(i) - 1)
    for (int j = jj; j >= 0; j = f(j) - 1)
      res += t[i];
  return res;
}

int add(int ii, int jj, int x) {
  for (int i = ii; i < n; i = g(i))
    for (int j = jj; j < m; j = g(j))
      t[i] += x;
}
```

Почему всё так просто? Двумерное дерево Фенвика можно представлять будто внутри одного Фенвика лежит второе, собственно, это тут и написано.

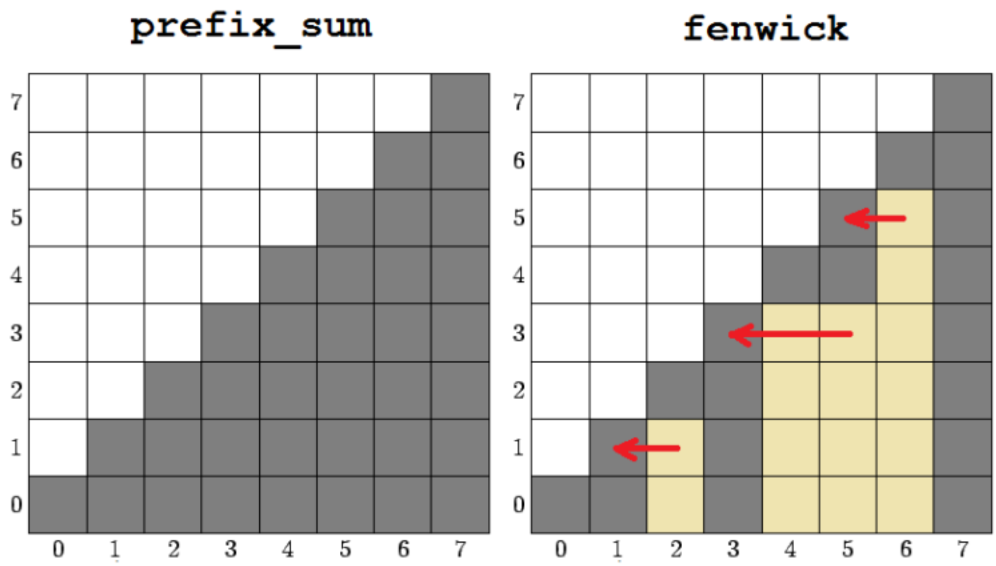
1.4 Инициализация

Мы могли бы вызвать *upd* для каждой клетки и проинициализировать Фенвика за $O(n \log n)$, но зачем, если мы знаем, что по определению Фенвик хранит сумму на непрерывном отрезке, такую сумму можно легко вычислить по префиксным суммам и получить инициализацию за $O(n)$.

```
int a[N], pref[N], t[N];

void init() {
  for (int i = 0; i < n; i++) {
    pref[i] = a[i] + (i == 0 ? 0 : pref[i - 1]);
    t[i] = pref[i] - pref[f(i) - 1];
  }
}
```

Очевидно, что мы можем избавиться от лишнего массива $pref$ и считать префиксные суммы внутри a, но на самом деле нам достаточно всего одного массива. Заметим, что для инициализации $t[i]$ нам могут потребоваться только такие элементы $pref[j]$, где $j \leq i$. Это даёт нам возможность использовать только один массив и, начиная с конца, постепенно переводить его из префиксных сумм в дерево Фенвика.




```
void init() {
  for (int i = 1; i < n; i++)
    a[i] = a[i] + a[i - 1];
  for (int i = n - 1; i >= 0; i--)
    if (f(i))
      a[i] -= a[f(i) - 1]
}
```

1.5 Групповые операции

Наверняка каждый понимает, как реализовать следующую структуру, используя примитив Фенвика:

- $+= \ (l, r, x)$

- ? (pos)

Для этого превратим запрос $+=$ на отрезке в запрос $+=$ в точке: $add(l, r, x) = add(r, x) + add(l - 1, -x)$, тогда значение в точке превратится в сумму на суффиксе, начиная с pos.

Но что если мы хотим узнавать сумму на отрезке и делать групповое $+=$? Писать дерево отрезков? Нет! Это путь для слабых духом!

В дереве Фенвика можно, как и в ДО хранить функции на отрезке, в данном случае будем хранить линейные функции вида $pref(i) = pref\_mul \cdot i + pref\_add$, где $pref\_add$ — массив, который вычитает "лишнее" из $pref\_mul \cdot i$.

```
vector<int> t_mul, t_add;
int n;

void internal_update(int i, int mul, int add) {
  for (; i < n; i |= i + 1) {
    t_mul[i] += mul;
    t_add[i] += add;
  }
}

void update(int l, int r, int x) {
  internal_update(l, x, -x * (l - 1));
  internal_update(r, -x, x * r);
}

int query(int i) {
  int mul = 0, add = 0, start = i;
  for (; i >= 0; i = (i & (i + 1)) - 1) {
    mul += t_mul[i];
    add += t_add[i];
  }
  return mul * start + add;
}
```

2 Встречное дерево Фенвика

Встречное дерево Фенвика хорошо тем,
что его никогда не придётся писать.
Джейсон Стетхем ©

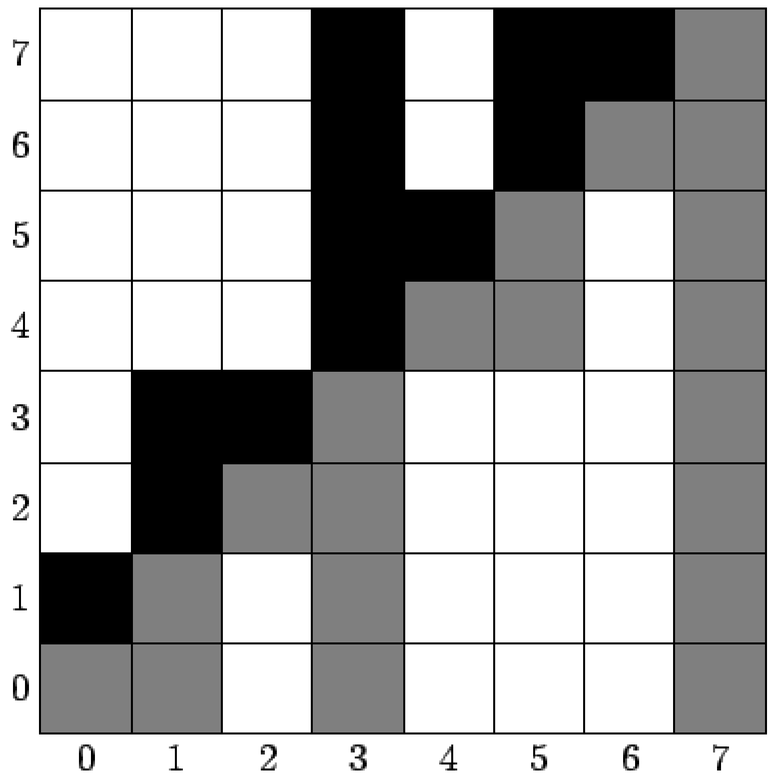
Мы только что говорили об ассоциативных коммутативных обратимых функциях, но обратимость вообще говоря не частое явление. Например, операция min не имеет обратной.

Встречное дерево Фенвика, в отличии от обычного дерева Фенвика, не требует существования обратной операции, в замен оно сложнее в реализации и занимает $2n$ памяти вместо n.

Как вы могли догадаться по названию, встречное дерево Фенвика – это на самом деле 2 дерева Фенвика, которые идут «на встречу друг другу». А именно, у нас вводится уже известная нам функция $F(x) = x - x\&(x+1) + 1$ и, помимо неё, функция $F'(x) = x + x\&(x+1) + 1$.

$$t_k = \sum_{i=F(k)}^{k} a_i, t'_k = \sum_{i=k+1}^{F'(k)} a_i$$

Тут лучше сразу перед глазами иметь картинку:



То есть наши два Фенвика встречаются на диагонали.

Более того, можно заметить, что $F'(x) = x|(x+1) = G(x)$. Это верно по той простой причине, что мы к x добавляем все последние под ряд идущие единичные биты $+\ 1$, то есть мы просто заменим последний 0 на 1.

Теперь хочется исследовать функцию F': как и для функции F очень хотелось бы, чтобы каждый элемент входил в $O(\log n)$ отрезков и каждый отрезок можно было бы разбить на $O(\log n)$ отрезков из встречного и прямого деревьев Фенвика. На самом деле, ответ на второй вопрос лежит на картинке: пусть у нас есть отрезок $[l; r]$. Он может либо пересекать клетку 2^k, либо лежать по одну сторону от неё. Если он её пересекает, то мы сможем разбить отрезок на 2 части: $[l; 2^k]$ и $[2^k; l]$ и повторить рассуждения для этих частей. Если же отрезок не пересекает 2^k, то он полностью лежит по одну сторону, опять переходим к меньшей задаче. Так по индукции мы сможем доказать, что нам достаточно $2 \log n$ отрезков.

Для функции F' нам опять нужно научиться понимать, какие отрезки нужно менять при изменении, то есть нужно найти все $j : F'(j) < i \leq j$.

$$\begin{array}{lcccccccc} j & = & a & b & c & 0 & 1 & 1 & 1 & 1 \\ i & = & a & b & c & 0 & * & * & * & * \\ F'(j) & = & a & b & c & 1 & 1 & 1 & 1 & 1 \end{array}$$

И тут мы замечаем, что $G'(x) = F(x)$, опять повторив рассуждения, которые уже 3 раза проговаривали.

Окей, на самом деле это всё, можно писать встречного Фенвика:

```cpp
vector <int> mx, mx1, a;

const int INF = 1e9;

int f(int i) { return i & (i + 1); }
int g(int i) { return i | (i + 1); }

void upd(int x, int up) {
  a[x] += up;
  for (int i = x; i < mx.size(); i = g(i))
    mx[i] = max(mx[i], a[x]);
  for (int i = x - 1; i >= 0; i = f(i) - 1)
    mx1[i] = max(mx1[i], a[x]);
}

int get(int l, int r) {
  --l;
  int ans = -INF;
  while (l < r) {
    if (f(r) - 1 >= l) {
      ans = max(ans, mx[r]);
      r = f(r) - 1;
    } else {
      ans = max(ans, mx1[l]);
      l = g(l);
    }
  }
  return ans;
}
```