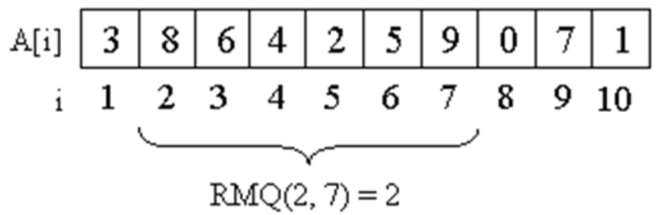
Задача RMQ

Аббревиатура RMQ расшифровывается как Range Minimum (Maximum) Query – запрос минимума (максимума) на отрезке в массиве. Для определённости мы будем рассматривать операцию взятия минимума.

Пусть дан массив A[1..n]. Нам необходимо уметь отвечать на запрос вида «найти минимум на отрезке с i-ого элемента по j-ый».



Рассмотрим в качестве примера массив A = {3, 8, 6, 4, 2, 5, 9, 0, 7, 1}. Например, минимум на отрезке со второго элемента по седьмой равен двум, то есть RMQ(2, 7) = 2.

В голову приходит очевидное решение: ответ на каждый запрос будем находить, просто пробегаясь по всем элементам массива, лежащим на нужном нам отрезке. Такое решение, однако, не является самым эффективным. Ведь в худшем случае нам придётся пробежаться по O(n) элементам, т.е. временная сложность этого алгоритма – O(n) на один запрос. Однако, задачу можно решить эффективнее.

Любой запрос о сумме можно эффективно обработать, если сначала построить массив префиксных сумм. Каждый элемент этого массива равен сумме первых элементов исходного массива:

Массив префиксных сумм:

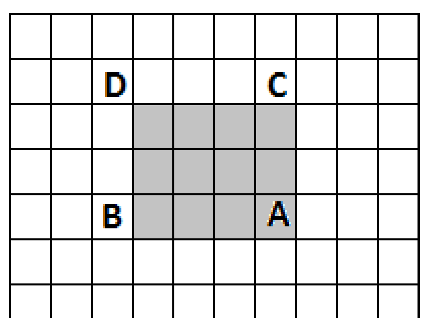
	3	11	17	21	23	28	37	37	44	45
i	1	2	3	4	5	6	7	8	9	10

Тогда любой запрос RMQ(a,b) можно вычислить за время O(1) :

RMQ(a,b) = RMQ(0,b) - RMQ(0,a - 1).

Эта идея обобщается и на многомерный случай. На рисунке показан двумерный массив префиксных сумм, который можно использовать для вычисления суммы по любому прямоугольному подмассиву за время O(1) по формуле

S(A) – S(B) – S(C) + S(D).



В этом разделе мы представим две древовидные структуры, позволяющие обрабатывать запросы по диапазону и изменять значения элементов массива за логарифмическое время.

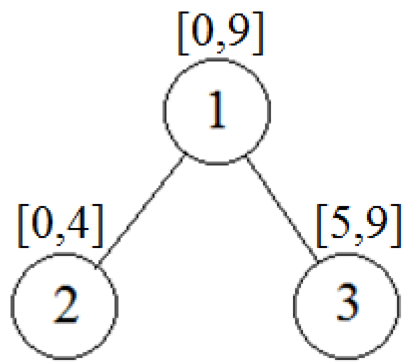
Дерево отрезков

Дерево отрезков - структура данных, которая позволяет реализовать за O(logN) операции следующего типа: нахождение суммы/минимума элементов массива в заданном отрезке (A[L..R], где L и R - это параметры запроса), изменение одного элемента массива, изменение/прибавление элементов на отрезке (A[L..R]). При этом объём дополнительно используемой памяти составляет O (N), или, если быть точным, не более 4 N.

Описание

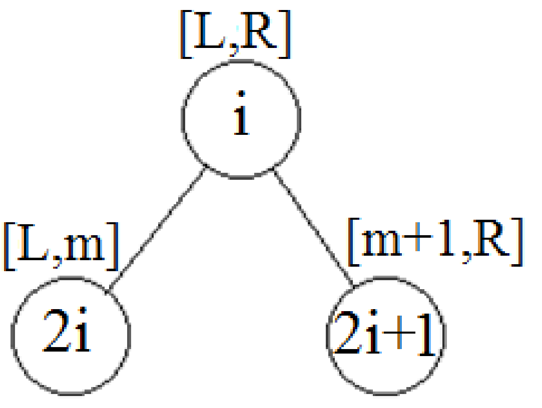
Для простоты описания будем считать, что мы строим дерево отрезков для суммы.

Построим бинарное дерево **T** следующим образом. Корень дерева будет храниться в элементе **T[1]**. Он будет содержать сумму элементов **A** от **0** элемента до **n-1** элемента, т.е. всего массива. Это мы обозначим **[0,n-1]**. Левый сын корня будет храниться в элементе **T[2]** и содержать сумму первой половины массива **A** от **0** элемента до **m** ([0,m]), а правый сын - в элементе **T[3]** и содержать сумму элементов второй половины от **m+1** до **n-1** ([m+1,n-1]) элемента, где **m=(n-1)/2**:



В общем случае, если **T[i]-**ый элемент содержит сумму элементов с **L**-го по **R-1** -ый, то его левым сыном будет элемент **T[i*2]** и содержать сумму **[L,m)**, а его правым сыном будет **T[i*2+1]** и содержать сумму **[m, R)**, где **m=(L+R)/2**.
Исключение, разумеется, составляют листья дерева - вершины, в которых **L = R**.

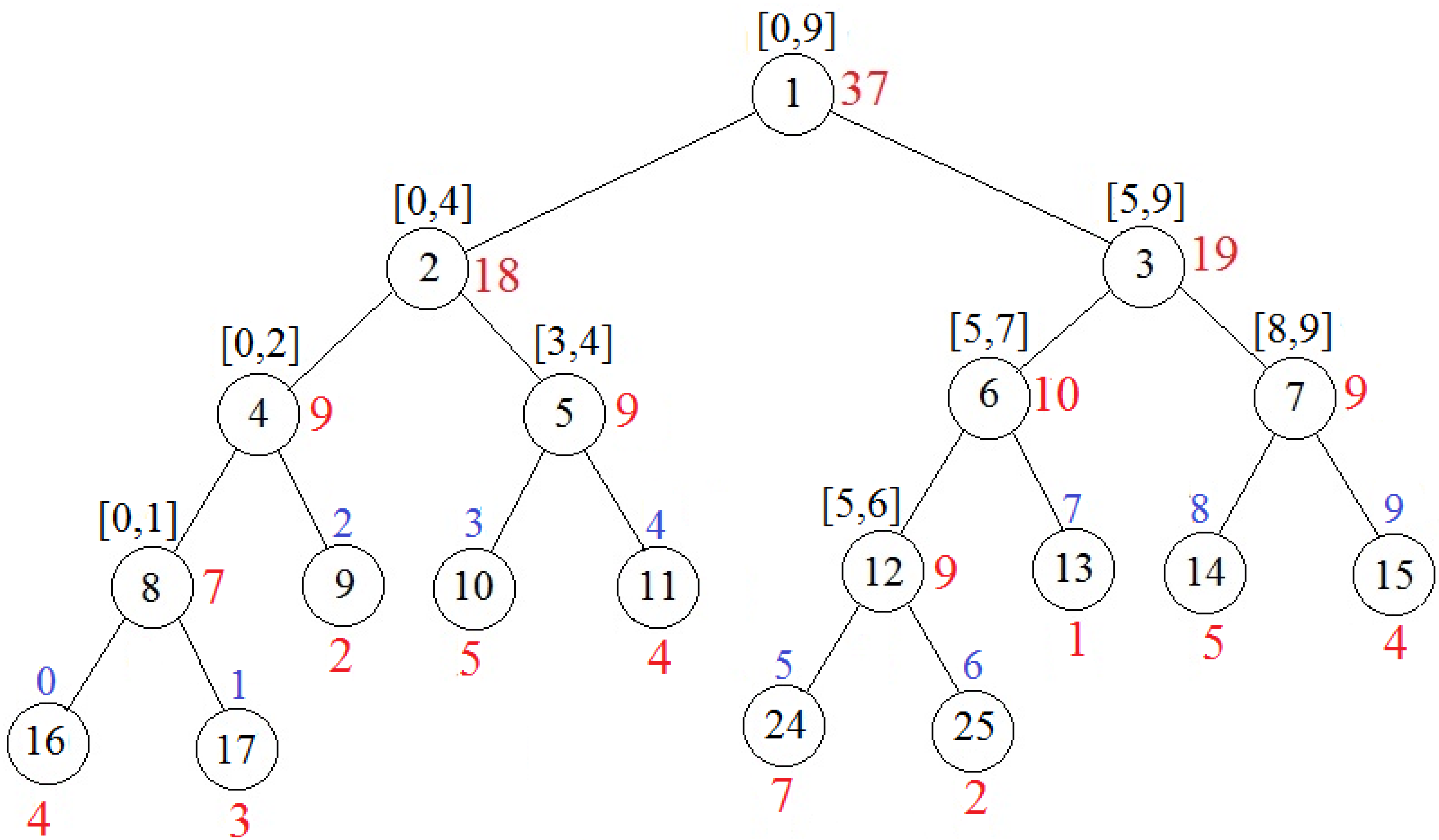
В общем случае, если **T[i]**-ый элемент содержит сумму элементов с **L**-го по **R**-ый, то его левым сыном будет элемент **T[i*2]** и содержать сумму **A[L..m]**, а его правым сыном будет **T[i*2+1]** и содержать сумму **A[m+1..R]**, где **m=(L+R)/2**. Исключение, разумеется, составляют листья дерева - вершины, в которых **L = R**.



Далее, нетрудно заметить, что это дерево будет содержать 4N элементов (а высота дерева будет порядка O (log N)). Поскольку значение в каждом элементе дерева однозначно определяется значениями в его сыновьях, то каждый элемент вычисляется за O (1), а всё дерево **строится** за O (N).

Пример. Пусть имеем массив A:

i:	0	1	2	3	4	5	6	7	8	9	**S**
A:	**4**	**3**	**2**	**5**	**4**	**7**	**2**	**1**	**5**	**4**	**37**



Рассмотрим теперь **операцию суммы** на некотором отрезке [L; R]. Мы встаём в корень дерева (i=1), и рекурсивно движемся вниз по этому дереву. Если в какой-то момент оказывается, что L и R совпадают с границами отрезка текущего элемента, то мы просто возвращаем значение текущего элемента T. Иначе, если отрезок [L; R] целиком попадает в отрезок левого или правого сына текущего элемента, то мы рекурсивно вызываем себя из этого сына и найденное значение возвращаем. Наконец, если отрезок [L; R] частично принадлежит и отрезку левого сына, и отрезку правого сына, то делим отрезок [L; R] на два отрезка [L; M] и [M+1; R] так, чтобы первый отрезок целиком принадлежал отрезку левого сына, а второй отрезок - отрезку правого сына, и рекурсивно вызываем себя и от первого, и от второго отрезков, возвращая сумму найденных сумм. В итоге вся операция суммирования работает за O (log N).

Теперь рассмотрим **операцию изменения** значения некоторого элемента с индексом K. Будем спускаться по дереву от корня, ища тот лист, который содержит значение элемента A[K]. Когда мы найдём этот элемент, просто изменим соответствующее значение в массиве T и будем подниматься от текущего элемента обратно к корню, пересчитывая текущие значения T. Понятно, что таким образом мы изменим все значения в дереве, которые нужно изменить. Итого асимптотика O (log N).

Наконец, рассмотрим **операцию изменения на отрезке**. Для реализации этой операции нам понадобится немного модифицировать дерево. Пусть каждый элемент дерева, помимо суммы, будет содержать значение Val[i]: если все элементы массива A в текущем отрезке равны друг другу, то Val[i] будет содержать это значение, а иначе он будет содержать некое значение "неопределённость". Изначально его можно просто заполнить значениями "неопределённость". А при выполнении операции изменения на отрезке мы будем спускаться по дереву, как в вышеописанном алгоритме суммирования, и если в какой-то момент L и R совпали с границами текущего отрезка, то мы присвоим Val[i] новое значение, которое мы хотим записать. Понятно, что теперь надо будет модифицировать операцию суммирования - если она в какой-то момент встречает Val[i], отличное от "неопределённости", то она прекращает спуск по дереву и сразу возвращает нужное значение - действительно, результат уже определён значением Val[i], а вот если мы продолжим спуск, то уже будем считывать неправильные, старые значения.

Операция прибавления на отрезке реализуется подобным образом, но несколько проще. В каждом элементе мы храним Add[i] - значение, которое нужно прибавить ко всем элементам этого отрезка. Операция прибавления на отрезке будет модифицировать эти значения, а операция суммирования - просто прибавлять к ответу все встретившиеся значения Add.

```
#include <iostream>
#include <vector>
using namespace std;
vector<long long> t;
vector<int> a;
int n, pos, newval;
void build (const vector<int> & a, int i = 1, int l = 0, int r = n-1) {
  if (l == r)  t[i] = a[l];
  else {
    int m = (l + r) / 2;
    build (a, i*2, l, m);   build (a, i*2+1, m+1, r);
    t[i] = t[i*2] + t[i*2+1];
  }
}
long long sum (int l, int r, int i = 1, int tl = 0, int tr = n-1) {
  if (l == tl && r == tr)  return t[i];
  int m = (tl + tr) / 2;
  if (r <= m)  return sum (l, r, i*2, tl, m);
  if (l > m)  return sum (l, r, i*2+1, m+1, tr);
  return sum (l, m, i*2, tl, m) + sum (m+1, r, i*2+1, m+1, tr);
}
```

```
void update (int pos, int newval, int i = 1, int l = 0, int r = n-1) {
   if (l == r)
   t[i] = newval;
   else {
       int m = (l + r) / 2;
       if (pos <= m)  update (pos, newval, i*2, l, m);
                      else   update (pos, newval, i*2+1, m+1, r);
       t[i] = t[i*2] + t[i*2+1];
   }
}
void main(){
cin>>n;
a.resize(n); t.resize (n*4 + 1);
for(int i=0; i<n; i++) cin >> a[i];
build(a);
cout<< "4  9 :“ << sum(4,9) << endl;
pos=5; newval=100;
update(pos,newval);
Cout << "4  9 :“ << sum(4,9) << endl;
}
```

Sparse Table, или разреженная таблица

Sparse Table – это таблица ST[][] такая, что ST[k][i] есть минимум на полуинтервале [A[i], A[i+2^k]). Иными словами, она содержит минимумы на всех отрезках, длина которых есть степень двойки.

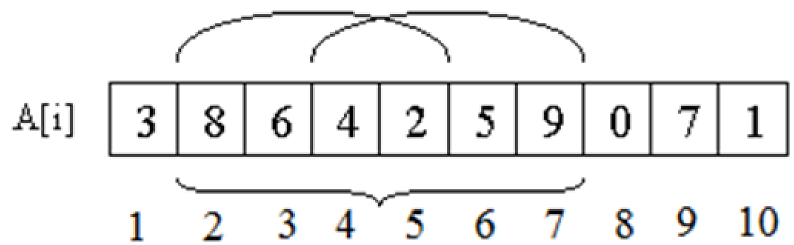
[3]	0	0	0							
[2]	3	2	2	2	0	0	0			
[1]	3	6	4	2	2	5	0	0	1	
ST [0]	3	8	6	4	2	5	9	0	7	1
	1	2	3	4	5	6	7	8	9	10
A[i]	3	8	6	4	2	5	9	0	7	1

Насчитаем массив ST[k][i] следующим образом. Понятно, что ST[0] просто и есть наш массив A. Далее воспользуемся понятным свойством:
ST[k][i] = min(ST[k-1][i], ST[k-1][i + 2^{k-1}]). Благодаря нему мы можем сначала посчитать ST[1], потом ST[2] и т. д.

Заметим, что в нашей таблице $O(n \cdot \log n)$ элементов, потому что номера уровней должны быть не больше $\log n$, т. к. при больших значениях k длина полуинтервала становится больше длины всего массива и хранить соответствующие значения бессмысленно. И на каждом уровне $O(n)$ элементов.

Снова лирическое отступление: Легко заметить, что у нас остаётся много неиспользованных элементов массива. Не стоит ли по-другому хранить таблицу дабы не тратить память впустую? Оценим количество незадействованной памяти в нашей реализации. На i-ом ряду неиспользованных элементов – $2^i - 1$. Значит, всего неиспользованными остаётся $\Sigma(2^i - 1) = O(2^{\log n}) = O(n)$, т. е. в любом случае понадобится порядка $O(n \cdot \log n) - O(n) = O(n \cdot \log n)$ места для хранения Sparse Table. Значит, можно не беспокоиться о неиспользуемых элементах.

А теперь главный вопрос: зачем мы всё это считали? Заметим, что любой отрезок массива разбивается на два перекрывающихся подотрезка длиною в степень двойки.



Получаем простую формулу для вычисления RMQ(i, j). Если $k = \log_2(j - i + 1)$, то $RMQ(i, j) = \min(ST[k][i], ST[k][j - 2^k + 1])$. Тем самым, получаем алгоритм за (O(nlogn), O(1)).
Здесь надо отметить, что $k_i = \log_2(i)$ $(i = 1,2,\ldots,n)$ надо вычислить заранее и логарифм округляется вниз.

Пример.

Вычислим RMQ(2, 7). $k = \log_2(7 - 2 + 1) = 2{,}585 = 2$.
RMQ(2, 7) = min(ST[2, 2], ST[2, 6]) = min(2, 2) = 2.

Упражнение.

Вычислим RMQ(3, 9).

Структура данных на основе sqrt-декомпозиции

Поставим задачу. Дан массив a[0 .. n-1]. Требуется реализовать такую структуру данных, которая сможет находить сумму элементов a[l .. r] для произвольных l и r за $O(\sqrt{n})$ операций.

Описание

Основная идея sqrt-декомпозиции заключается в том, что сделаем следующий **предпосчёт**: разделим массив a на блоки длины примерно $\sqrt{n}$, и в каждом блоке i заранее предпосчитаем сумму ***b[i]*** элементов в нём.

Можно считать, что длина одного блока и количество блоков равны одному и тому же числу — корню из n, ***округлённому вверх***:

$s = \lceil \sqrt{n} \rceil$,

тогда массив $a[]$ разбивается на блоки примерно таким образом:

$$\underbrace{a[0]\ a[1]\ \ldots\ a[s-1]}_{b[0]}\ \ \underbrace{a[s]\ a[s+1]\ \ldots\ a[2\cdot s-1]}_{b[1]}\ \ \ldots\ \ \underbrace{a[(s-1)\cdot s]\ \ldots\ a[n]}_{b[s-1]}.$$

Хотя последний блок может содержать меньше, чем s, элементов (если n не делится на s), — это не принципиально.

Таким образом, для каждого блока k мы знаем сумму на нём ***b[k]***:

$$b[k] = \sum_{i=k\cdot s}^{\min(n-1,(k+1)\cdot s-1)} a[i].$$

Итак, пусть эти значения $\boldsymbol{b_k}$ предварительно подсчитаны (для этого надо, очевидно, ***O(n)*** операций). Что они могут дать при вычислении ответа на очередной запрос ***(l, r)***? Заметим, что если отрезок **[*l; r*]** длинный, то в нём будут содержаться несколько блоков целиком, и на такие блоки мы можем узнать сумму на них за одну операцию. В итоге от всего отрезка **[*l; r*]** останется лишь два блока, попадающие в него лишь частично, и на этих кусках нам придётся произвести суммирование тривиальным алгоритмом.

Иллюстрация (здесь через ***k*** обозначен номер блока, в котором лежит ***l*** , а через p – номер блока, в котором лежит ***r***):

$$\dots\ \overbrace{a[l]\ \dots\ a[(k+1)\cdot s-1]\ \underbrace{a[(k+1)\cdot s]\ \dots\ a[(k+2)\cdot s-1]}_{b[k+1]}\ \dots\ \underbrace{a[(p-1)\cdot s]\ \dots\ a[p\cdot s-1]}_{b[p]}\ a[p\cdot s]\ \dots a_r}^{sum=?}\ \dots$$

На этом рисунке видно, что для того чтобы посчитать сумму в отрезке**[*l* . . *r*]**, надо просуммировать элементы только в двух "хвостах": **[*l* .. *(k+1)*s – 1*]** и [p*s .. r], и просуммировать значения ***b[i]*** во всех блоках, начиная с ***k + 1*** и заканчивая ***p - 1***:

$$\sum_{i=l}^{r} a[i] = \sum_{i=l}^{(k+1)\cdot s-1} a[i] + \sum_{i=k+1}^{p-1} b[i] + \sum_{i=p\cdot s}^{r} a[i]$$

(примечание: эта формула неверна, когда ***k = p*** : в таком случае некоторые элементы будут просуммированы дважды; в этом случае надо просто просуммировать элементы с l по r)

Тем самым мы экононим значительное количество операций. Действительно, размер каждого из "хвостов", очевидно, не превосходит длины блока s, и количество блоков также не превосходит s. Поскольку s мы выбирали $\approx \sqrt{n}$, то всего для вычисления суммы на отрезке $[l \dots r]$ нам понадобится лишь $O(\sqrt{n})$ операций.

Данный метод можно использовать для нахождения максимума/минимума на отрезке.

Возможен запрос на изменение элемента.

Рассмотрим задачу подсчета числа инверсий в перестановке.

Число инверсий (беспорядка) в перестановке – это количество пар элементов (не обязательно соседних), в которых следующий элемент имеет меньший номер, чем предыдущий.

Например, для перестановки `1 5 8 4 3 6 7 2`, имеем 13 инверсий.

Здесь в качестве массива ***b[i]*** возьмём количество введённых элементов принадлежащих этому отрезку. Количество блоков будет равно √n = 3. Данные будем вводить и сразу заполнять массив ***b[i]***:

<table>
<tr><td colspan="3">b[0]</td><td></td><td colspan="3">b[1]</td><td></td><td colspan="2">b[2]</td></tr>
<tr><td colspan="3">1</td><td></td><td colspan="3">0</td><td></td><td colspan="2">0</td></tr>
<tr><td>a[1]</td><td>a[2]</td><td>a[3]</td><td></td><td>a[4]</td><td>a[5]</td><td>a[6]</td><td></td><td>a[7]</td><td>a[8]</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td></td><td>0</td><td>0</td><td>0</td><td></td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><td colspan="3">b[0]</td><td></td><td colspan="3">b[1]</td><td></td><td colspan="2">b[2]</td></tr>
<tr><td colspan="3">1</td><td></td><td colspan="3">1</td><td></td><td colspan="2">0</td></tr>
<tr><td>a[1]</td><td>a[2]</td><td>a[3]</td><td></td><td>a[4]</td><td>a[5]</td><td>a[6]</td><td></td><td>a[7]</td><td>a[8]</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td></td><td>0</td><td>1</td><td>0</td><td></td><td>0</td><td>0</td></tr>
</table>

b[0]				b[1]				b[2]	
1				1				**1**	
a[1]	a[2]	a[3]		a[4]	a[5]	a[6]		a[7]	a[8]
1	0	0		0	1	0		0	**1**

b[0]				b[1]				b[2]	
1				2				1	
a[1]	a[2]	a[3]		a[4]	a[5]	a[6]		a[7]	a[8]
1	0	0		**1**	1			0	1

b[0]				b[1]				b[2]	
2				2				1	
a[1]	a[2]	a[3]		a[4]	a[5]	a[6]		a[7]	a[8]
1	0	**1**		1	1	0		0	1

b[0]				b[1]				b[2]	
2				**3**				1	
a[1]	a[2]	a[3]		a[4]	a[5]	a[6]		a[7]	a[8]
1	0	1		1	1	**1**		0	1

b[0]				b[1]				b[2]	
2				3				**2**	
a[1]	a[2]	a[3]		a[4]	a[5]	a[6]		a[7]	a[8]
1	0	1		1	1	1		**1**	1

b[0]				b[1]				b[2]	
3				3				2	
a[1]	a[2]	a[3]		a[4]	a[5]	a[6]		a[7]	a[8]
1	**1**	1		1	1	1		1	1

Древовидные структуры

В этом разделе мы представим две древовидные структуры, позволяющие обрабатывать запросы по диапазону и изменять значения элементов массива за логарифмическое время. Сначала обсудим двоичные индексные деревья, поддерживающие запросы о сумме, а затем - деревья отрезков, поддерживающие также запросы других видов.

Двоичные индексные деревья

Двоичное индексное дерево (или дерево Фенвика) можно рассматривать как динамический вариант массива префиксных сумм. Оно предоставляет две операции с временной сложностью O(log n): обработка запроса о сумме по диапазону и изменение значения. Хотя в названии этой структуры данных фигурирует слово «дерево», обычно она представляется в виде массива. При обсуждении двоичных индексных деревьев мы будем предполагать, что все массивы индексируются, начиная с 1, потому что это упрощает реализацию структуры.

Обозначим p(k) наибольшую степень двойки, делящую k. Двоичное индексное дерево хранится в массиве tree таким образом, что

tree[k] = sumq(k - p(k) + 1, k),

т. е. элемент в позиции k содержит сумму по заканчивающемуся в этой позиции диапазону длины p(k). Например, поскольку p(6) = 2, tree[6] содержит значение sumq(5,6). На рис. 1 показаны массив и соответствующее ему двоичное индексное дерево. На рис. 2 показано соответствие между значениями двоичного индексного дерева и диапазонами исходного массива.

	1	2	3	4	5	6	7	8
Исходный массив	1	3	4	8	6	1	4	2

	1	2	3	4	5	6	7	8
Двоичное индексное дерево	1	4	4	16	6	7	4	29

Рис. 1. Массив и его двоичное индексное дерево

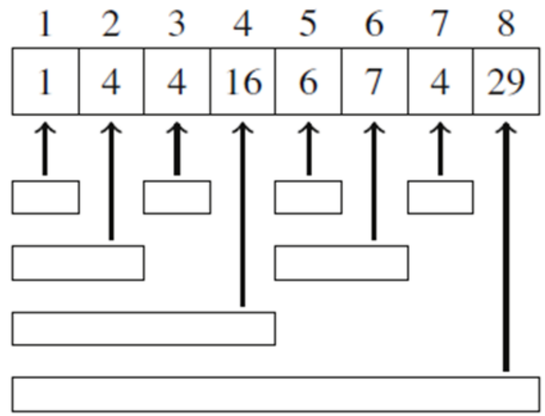


Рис. 2. Соответствие между двоичным индексным деревом и диапазонами

Зная двоичное индексное дерево, мы можем вычислить любое значение вида sumq(1, k) за время O(log n), поскольку диапазон [1, k] всегда можно разбить на O(log n) поддиапазонов, суммы по которым хранятся в дереве. Например, чтобы вычиспить sumq(l, 7), мы разобьем диапазон [1, 7] на три поддиапазона: [1,4], [5,6] и [7,7] (рис. 3). Поскольку суммы по этим поддиапазонам уже имеются в дереве, то мы можем вычислить сумму по всему диапазону по формуле:

sumq(1,7) = sumq (1, 4) + sumq(5,6) + sumq(7,7) = 16 + 7 + 4 = 27.

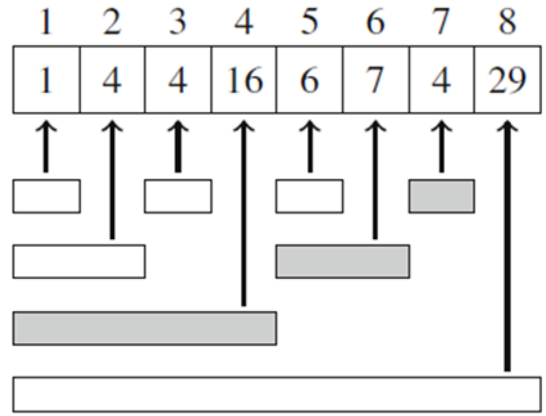


Рис. 3. Обработка запроса о сумме по диапазону с помощью двоичного индексного дерева

А чтобы вычислить значение sumq(a, b), где a > 1, мы применим тот же прием, что в случае массивов префиксных сумм:

$$\text{sumq}(1, b) = \text{sumq}(1, b) - \text{sumq}(1, a - 1).$$

И sumq(1, b), и sumq(1, a - 1) можно вычислить за время O(log n), поэтому полная временная сложность равна O(log n).

После изменения любого элемента массива необходимо обновить двоичное индексное дерево. Например, если изменяется элемент 3, то следует обновить суммы по диапазонам [3, 3], [1,4] и [1, 8] (рис. 4). Поскольку каждый элемент массива принадлежит O(log n) таким диапазонам, то достаточно обновить O(log n) элементов дерева.

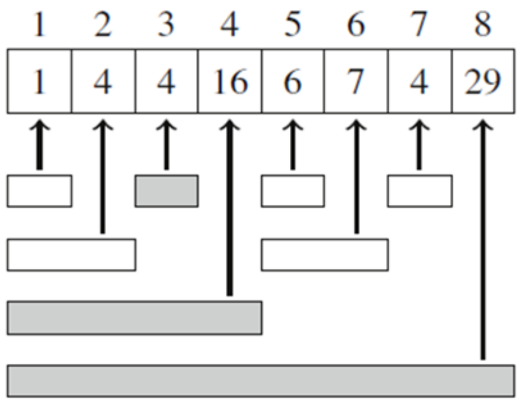


Рис. 4. Обновление значения в двоичном индексном дереве

Реализация. Операции двоичного индексного дерева эффективно реализуются с помощью поразрядных операций. Нам понадобится следующий факт: значение p(к) можно вычислить по формуле:

$$p(k) = k \,\&\, -k,$$

которая выделяет самый младший единичный бит k.

Следующая функция вычисляет sumq(1, k):

```
int sum(int k) {
int s = 0;
   while (k >= 1) {
      s += tree[k];
      k -= k & -k;
   }
return s;
}
```

А эта функция увеличивает значение k-го элемента массива на x (x может иметь любой знак):

```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x; k += k& -k;
    }
}
```

Временная сложность обеих функций равна O(log n), потому что они обращаются к O(log n) элементам двоичного индексного дерева, а переход к следующей позиции занимает время O(1).