Дерево отрезков

Данная структура данных используется, когда на массиве данных необходимо большое количество запросов вида:

1. найти значение максимального элемента на отрезке массива [a..b].
2. заменить i-й элемент массива на x
3. добавить x ко всем элементам на отрезке [a..b]

Также в качестве запросов вида 1 может быть любая ассоциативная функция:

(A ~ B) ~ C = A ~ (B ~ C)

Строим дерево отрезков

Пусть у нас есть массив b из n элементов. Для начала нам нужно найти nMax (наименьшая степень двойки, которая не превосходит n). Это можно реализовать как через формулу:

```
nMax = 2*  ((int)(log2(1.0*(n-1)) + 1);
```

Или вот так

```
nMax= 1;
while (nMax<n) nMax *= 2;
```

Далее нужно заполнить массив a нулями (соответствующего типа) и заполнить листы ДО значениями из массива b (мы помним, что в ДО индексы листьев от nMax до 2*nMax-1):

```
for (int i=0; i<n; i++)
   a[nMax+i]= b[i];
```

Теперь осталось только заполнить значения во всех родителях. Это можно сделать за один линейный проход (помним, что у i-ой вершины сыновья с индексами 2*i и 2*i+1, а в вершине мы храним значение функции от двух сыновей):

```
for (int i=nMax-1; i>0; i--)
      a[i]= f(a[2*i],a[2*i+1]);
```

Таким образом мы построили ДО с асимптотикой O(nMax) = O(n).

В нашем случае мы будем считать, что a[0] не участвует в дереве отрезков.
А корень располагается в a[1].

Узнать значение i-го элемента.

Как уже писалось ранее у нашего ДО листья имеют индексы от nMax до 2*nMax-1, поэтому значение i-го элемента находиться в ячейке с индексом nMax+i:

```
return a[nMax+i]
```

Очевидно, что данный запрос выполняется за константу.

Изменить значение i-го элемента.

Если мы изменим значение в листе дерева, то все значения на пути к корню от данного листа перестанут соответствовать действительности, поэтому их нужно пересчитать, в остальных же останутся корректные значения.

Как известно, глубина полного бинарного дерева из m вершин равна $\log_2 m$, поэтому мы должны выполнить данную операцию за логарифмическое время. Например, изменим a2 на $a2^l$:

Что бы «обновить» ДО нам нужно записать в лист новое значение, а затем подняться до корня, каждый раз пересчитывая значение функции в вершине. Изменить значение в листе очень просто (вспомним, что индексы листьев от nMax до 2*nMax-1). Значение i-го листа имеет индекс nMax+i :

```
a[nMax+i]= newValue;
```

Теперь осталось подняться до корня, это можно сделать с помощью цикла:

```
while (i>1)
{
    i/= 2;
    a[i]= f(a[2*i],a[2*i+1]);
}
```

Найти значение функции на отрезке от l до r.

Наконец-то, мы добрались до самого интересного запроса. Стоит отметить, что частный случай, когда l=r разобран в пункте 2 и выполняется за константу, в общем же случае асимптотика логарифмическая.

Введем определения.

Фундаментальный отрезок – такой отрезок, для которого существует вершина в дереве, хранящая значение функции на данном отрезке.

Уровень.

Уровень корня – 1, а для каждого сына уровень на единицу больше, чем у родителя.

Для того, что бы вычислить значение функции на отрезке, нам необходимо разбить его на МИНИМАЛЬНОЕ количество фундаментальных отрезков. Что бы найти значение для любой вершины (кроме листа), нам нужно знать значения для сыновей.
Мы будем спускаться по ДО. Изначально встаем в корень. Пусть мы находимся в какой-то вершине. Рассмотрим 3 возможных случая: отрезок [l..r] совпадает с отрезком, соответствующим текущей вершине, отрезок [l..r] полностью принадлежит одному из сыновей, отрезок принадлежит обоим сыновьям. В первом случае просто возвращаем посчитанное значение из ДО, во-втором – спускаемся в данного сына, в-третьем же случае разобьем данный отрезок на два: [l..правый конец левого сына] и [левый конец правого сына..r].

Рекурсивно вычислим значения для каждого из них.

Создаем вспомогательную функцию.

```
int query(int l, int r, int leftPosition, int rightPosition, int v)
{
// return value function f on the intersection segments [l;r] and
[leftPosition;rightPosition]
// l - левая граница запроса
// r - правая граница запроса
// v - текущая вершина дерева отрезков
// [leftPosition; rightPosition] - отрезок соответствующий v
if (r<l) return zero; //если отрезок не существует, то возвращаем ноль.
if (l==leftPosition && r==rightPosition) return a[v];
// если отрезок фундаментальный,то возвращаем значение из вершины
// раз мы дошли сюда, то отрезок принадлежит сыновьям
int middle= (leftPosition+rightPosition)/2;
// вычисляем правую границу левого сына
return
f(query(l,min(middle,r),leftPosition,middle,v*2),query(max(l,middle+1),r,middle+1,righ
tPosition,v*2+1));
// рекурсивно вычисляем запросы для сыновей

}
```

```
const int N = 1e5; // limit for array size
int n; // array size
int t[2 * N];

void build() {
 // build the tree
for (int i = n - 1; i > 0; --i) t[i] = t[i*2] + t[i*2-1]; }

void modify(int p, int value) {
// set value at position p
for (t[p += n] = value; p > 1; p /=2 ) t[p/2] = t[p] + t[p^1];
}

int query(int l, int r)
{ // sum on interval [l, r)
int res = 0;
for (l += n, r += n; l < r; l /=2, r /= 2)
{
   if (l%2==1) res += t[l++];
   if (r%2==1) res += t[--r];
}
return res;
}
```

```
int main()
{
Int l,r;
scanf("%d", &n);
for (int i = 0; i < n; ++i)
    scanf("%d", &t[n + i]);
    build();
    modify(0, 1);
for (int i = 0; i < 10; ++i)
{
    scanf(“%d %d”, &l, &r);
    printf("%d\n", query(l, r));
}
    return 0;
}
```