

$l1 = \text{RSQ1}(ST, A, i, j, s, m, 2p+1, n)$   
 $l2 = \text{RSQ2}(ST, A, i, j, m+1, e, 2p+2, n)$

} return  $(l1+l2)$

—x—

If we want to <sup>ans</sup>implement RSQ, we don't need to implement a segment tree, there is an even simpler data structure called binary index tree (BIT).

# BIT

There are 2 ~~opre~~ operations that we're going to do:-

- i)  $\text{RSQ}(A, i, j) \longrightarrow \sum_{k=i}^j A[k]$
- ii)  $\text{update}(A, i, x) \longrightarrow A[i] = x$

—x—

So to know RSQ we'll not try to find RSQ, we'll try to find the prefix sum. (PS(i))

$$PS(i) = A[1] + A[2] + \dots + A[i]$$

↓

\* { the indexing here is going to be from 1.

If we know how to find the prefix sum, we can answer the range sum query as follows

$$RSQ(A, i, j) \rightarrow \sum_{k=i}^j A[k] \rightarrow PS[j] - PS[i-1]$$

In case of update fn, we're not going to do update. We're going to write the fn

$$\text{increase} \rightarrow IN(i, x) \rightarrow A[i] = A[i] + x$$

We can use this IN to update.

for ex if we need to update  $A[i]$  to  $x$

$$\text{then use } IN(i, x - A[i])$$

↓

$$A[i] = A[i] + x - A[i] \\ = x$$

°° If we know how to do prefix sum & the increase key, we can do RSQ & update.



#

BIT

indices    0    1    2    3    4    5    6    7    8    9

A    0    4    3    2    1    -2    6    -3    7    4

\*

↓

not in given  
array. Bit here  
we start indexing ~~from~~  
from 1

BIT    0    4    7    2    8    -2    4    -3    16    4

↓

just an  
array,  
not a  
tree

⇒ At each index of BIT, we'll store  
sum of previous few no. & the no.  
at that index itself. The no. of previous  
num will depend on the index of BIT.

↓  
(i)

Let  $j$  = no. of numbers to be summed (including  
the no. at current  
index)

look at the binary rep. of  $i$

1 → 1 →  $j=1$

2 → 10 →  $j=2$

↓

first one from right to left

3 → 11 →  $j=1$

↓

first one  
from right to left



$$4 \rightarrow 100 \Rightarrow j = 4$$

1st one from right  
to left

$$6 \rightarrow 110 \Rightarrow j = 2$$

1st one from  
right to left

$$5 \rightarrow 101 \Rightarrow j = 1$$

1st one from  
right to left

$$j = i \& (-i) \quad (\text{single } \&)$$

From now onwards there is no BIT,  
our array A itself is the following

A 0 4 7 2 8 -2 4 -3 16 4

i.e. we're not going to store the actual no.,  
what we're going to store is the BIT rep.

This BIT  
rep. is  
on paper  
now. To  
form this  
through  
code, we'll  
use ~~BIT~~  
Increase  
key

∴ amount of space required is only n.

This is the advantage of BIT vs segment  
tree

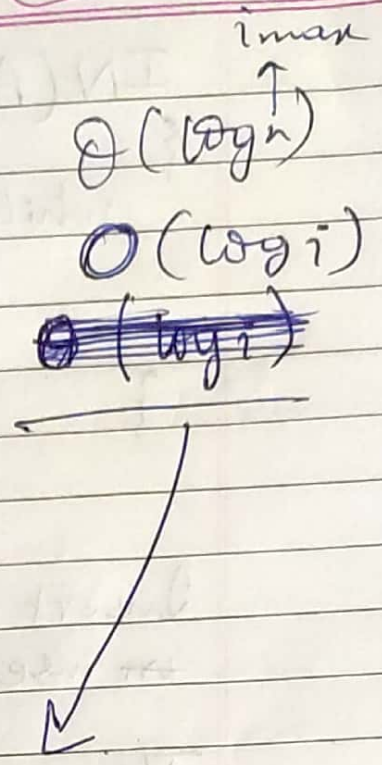
i.e. there you really needed the tree. Here  
there is no tree.

only when we read from file



100 100  
 0 1 2 3 4  
 4. 6 12 1  
 S = 1  
 4 - 9.2

(\*) PS(i)  
 {  
   s = 0  
   while (i > 0)  
   {  
     s = s + A[i]  
     i = i - (i & (-i))  
   }  
 }



Let's i be 111111000  
 when we 1st do  $i - (i \& (-i))$   
 i becomes 111110000  
 on 2nd time i becomes 111100000  
 i.e. each time a ~~dec~~ no. of 1's ↓ by one.  
 & the no. of 1's in i can atmost be  $\log i \rightarrow \Theta(\log i)$  & i can atmost be  $n \rightarrow \Theta(\log n)$

Let's look at IN, so what happens in IN is that it won't affect the no. before i if  $IN(A, i, x)$

b/c we've to certainly update our A[i] & several location after i but value at locations before i won't change.

```

IN(A, i, x)
{
    while(i < n)
        A[i] += x
        i = i + (i & (-i))
}

```

 $O(\log n)$ 

Insert is, increase is  $\log n$ . So, if we do  $n$  insertions, it will be of order  $n \log n$ .

This is the only disadvantage of BIT i.e. if we are given a certain array. The pre-processing to ans RQ & update, takes order  $n \log n$ .

Once we do this pre computation the Te & prefix sum can be found in order  $\log n$ .