Fenwick Tree → ADS

arr → | a[0] | a[1] | . . . | a[n-1] | a[n] |

So, to compute sum(l, r) = pre[r] − pre[l-1].
But what if we are also asked to update a range.

∴ sum → O(1)
update → O(N).

Fenwick tree : sum → O(logN)
update → O(logN)

arr :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	2	1	1	3	0	4	2	5	2	2	3	1	0	2

We create a Binary index tree, so choose an index 'i', write its binary representation, turn off the right most set bit & add 1 to it & let it is 'j',
So, this ~~holds~~ index holds value from ~~(i,j) or~~ (j, i)

i=1 : 001 $\xrightarrow{\text{turn off 1}}$ 000 $\xrightarrow{\text{decimal}}$ 0 $\xrightarrow{+1}$ 1 ∴ (1, 1)
i=2 : 010 $\xrightarrow{\text{turn off 1}}$ 000 → 0 $\xrightarrow{+1}$ 1 ∴ (1, 2)
i=3 : 011 $\xrightarrow{\text{turn off 1}}$ 010 → 2 $\xrightarrow{+1}$ 3 ∴ (3, 3)
i=4 : 100 $\xrightarrow{\text{turn off 1}}$ 000 → 0 $\xrightarrow{+1}$ 1 ∴ (1, 4)
⋮

initial :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(1,1)	(1,2)	(3,3)	(1,4)	(5,5)	(5,6)	(7,7)	(1,8)	(9,9)	(9,10)	(11,11)	. . .				(1,16)

Now, array is given i.e arr[1], arr[2], ... arr[n] are given as input by use.
So, arr[1] affects index in BIT : 1, 2, 4, 8, 16, . . .
arr[2] affects index " " : 2, 4, 8, 16
arr[3] affects index " " : 3, 4, 8, 16

So; how to get the series of point indexes when arr[i] is given as input?

Steps : (i) Takes 2' complement of i
(ii) & with original number
(iii) add to original number.

Continue until, the index hits value $>n$.

$$\boxed{i = i + (i \& (-i))};$$

① Update : Let say arr[i] = k_1 & we update it to arr[i] = k_2

$\therefore$ arr[i] = arr[i] + $(k_2 - k_1)$

So; its basically same as adding $k_2 - k_1$ to the original index.

Now; in BIT we go to index i, change it to BIT[i] += $(k_2 - k_1)$.

Update i += (i&(-i)) & BIT[i] += $(k_2 - k_1)$.

Keep on doing this.

② Range sum : Lets ~~to~~ say we want s(l,r)

$\therefore$ s(l, r) = s(1, r) − s(1, l−1)

So; lets see how to compute $\underline{\underline{s(1,i)}}$.

sum = 0;

sum += BIT[i];
i = i − (i&(−i));
Continue till $\underline{\underline{i < N}}$.

Pseudo code :

```
int fen[N];
void update (int i, int ~~arr[i]~~ val){
        while (i<N){
            ~~fen~~ BIT[i] += val;
                i += (i&(-i));
        }
}

~~void~~ int sum (int i){
    int s=0;
    while (i>0){
      s += BIT[i];
      i = i - (i&(-i));
    }
  return s;
}

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

2

Binary Lifting on Fenwick tree / Binary Indexed tree

→ Used for lower-bound or upper-bound of prefix sums.

```
int find (int k){
    int left =0, ~~ans~~ curr_sum=0;
    for (i = log2(n); i>=0; i--){
          if (BIT[left + (1<<i)] + curr_sum < k){
            curr_sum += BIT[left +(1<<i)];
            left += (1<<i);
          }
    }
  return (left+1);
}
```

Heuristics

- **BIT + Binary-search**: $O(n\log^2 n)$
- **Binary lifting in BIT**: $O(n\log n)$

(**Problem** D. Multiset → Codeforces)

Gist: **Fenwick trees** → updates index value $\boxed{O(\log n)}$
→ gets prefix sum $\boxed{O(\log n)}$

In the problem, $1 \le a_i \le n$.
So; we create an array for all possible "n+1" numbers.

Say: ~~x~~ $n=5$; arr = $\{1,2,3,4,5\}$

freq = $[0\ 1\ 1\ 1\ 1\ 1]$ → similar to hashing stores freq.
arr = $\{0\ 1\ 2\ 3\ 4\ 5\}$

If we want to get count of numbers $\le$ a certain number say "k", then store prefix sum for freq & return pre[k].

We can also perform binary-search in the prefix sums array.

So; we make **BIT for freq. array**.

Code (BIT + Binary_Search)

```
int n,q; cin>>n>>q;
~~for~~ loop(i, n){
    int num; cin>>num;
    update(num, 1); // Increment "num" position value
                       by 1
}

while (q--) {
    int x; cin>>x;
    if(x>0) update(x, 1);
    else {
     x = -1*x;
     int low = 0, high = N;
      while (low < high){
       int mid = (low+high)>>1;
         int val = sum(mid); // prefix sum in BIT

         if (x <= val)
             high = mid;
         else
             low = mid+1;
      }
    update(low, -1);
}  }
```

Code (Binary Lifting)

```
int findKth(int k){
    int left = 0, sum = 0;
    for (i = log2(n); i>= 0, i--){
        if (BIT[left + (1<<i)] + sum < k){
            sum += BIT[left + (1<<i)];
        }   left += (1<<i);
    }
    return (left + 1);
}
```

Range Updates & Range Query

We use $B_1[\,]$ & $B_2[\,]$ initialised to zero.
Lets say we want to update $[l, r]$ to x.

```
def range_add(l, r, x):
    add(B1, l, x);
    add(B1, r+1, -x);
    add(B2, l-1, x*(l-1));
    add(B2, r, -x*r);
```

After the range ~~sum (f,~~ update (l, r, x), the range sum query should return this:

$$sum[0,i] = \begin{cases} 0; & i<l \\ x*(i-(l-1)); & l\le i\le r \\ x*(r-l+1); & i>r \end{cases}$$

$$\therefore \; sum[0,i] = i * sum(B_1, i) - sum(B_2, i)$$

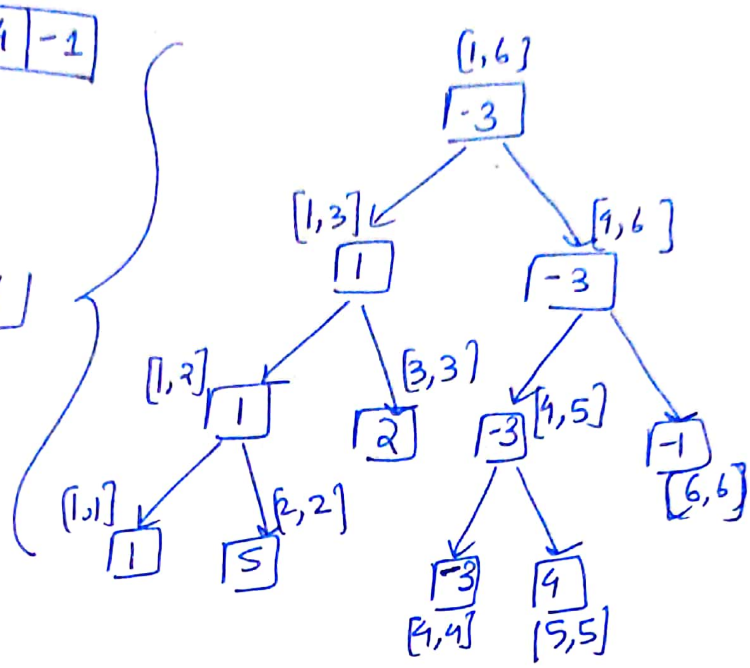
$$= \begin{cases} i*0 - 0 = 0; & i<l \\ i*x - x*(l-1) = x(i-(l-1)); & l\le i\le r \\ i*0 - \{x(l-1) - xr\} = x(r-l+1); & i>r \end{cases}$$

Scanned with CamScanner

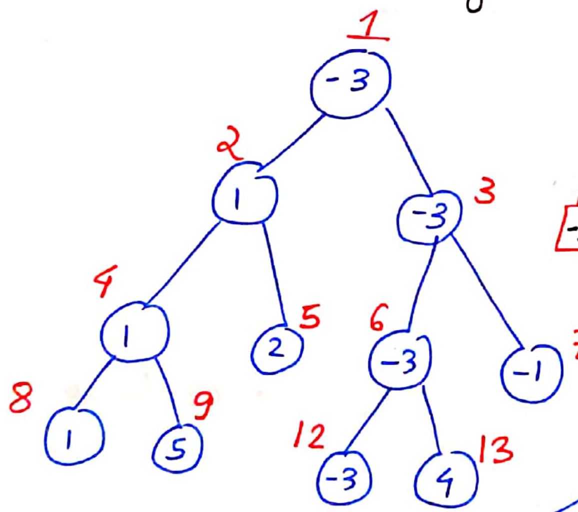
SEGMENT TREES

1	5	2	-3	4	-1

$h = \lfloor \log N \rfloor$



We can use an array to store the "values".



1	2	3	4	5	6	7	8	9	10	11	12	13
-3	1	-3	1	2	-3	-1	1	5	x	x	-3	4

```
void buildtree (int i, int l, int r)
{
    if (l==r)
    {SEGTREE[i] = arr[l]; return;}
    int mid = (l+r)/2;
    buildtree (2*i, l, mid);
    buildtree (2*i+1, mid+1, r);
    SEGTREE[i] = min(SEGTREE[2*i], SEGTREE[2*i+1]);
}
```

segment tree index

original array start & end index.

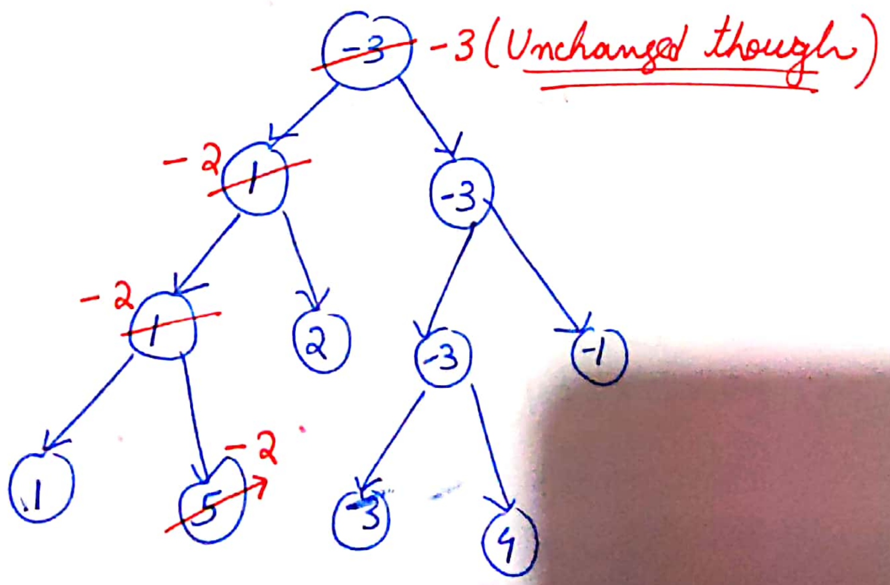
```
int query (int i, int l, int r, int ql, int qr) {
    if (ql > r || qr < l) // Completely outside.
        return INF;
    if (ql <= l && qr >= r) // Completely inside
        return SEGTREE[i];
    int mid = (l+r)/2;
    int left_min = query (2i, l, mid, ql, qr);
    int right_min = query (2i+1, mid+1, ql, qr);
    return min (l, r);
}
```

POINT UPDATE (Update arr[j] to "val")

arr[] =

1	5	2	-3	4	-1	[illegible]

Update : arr[2] = -2



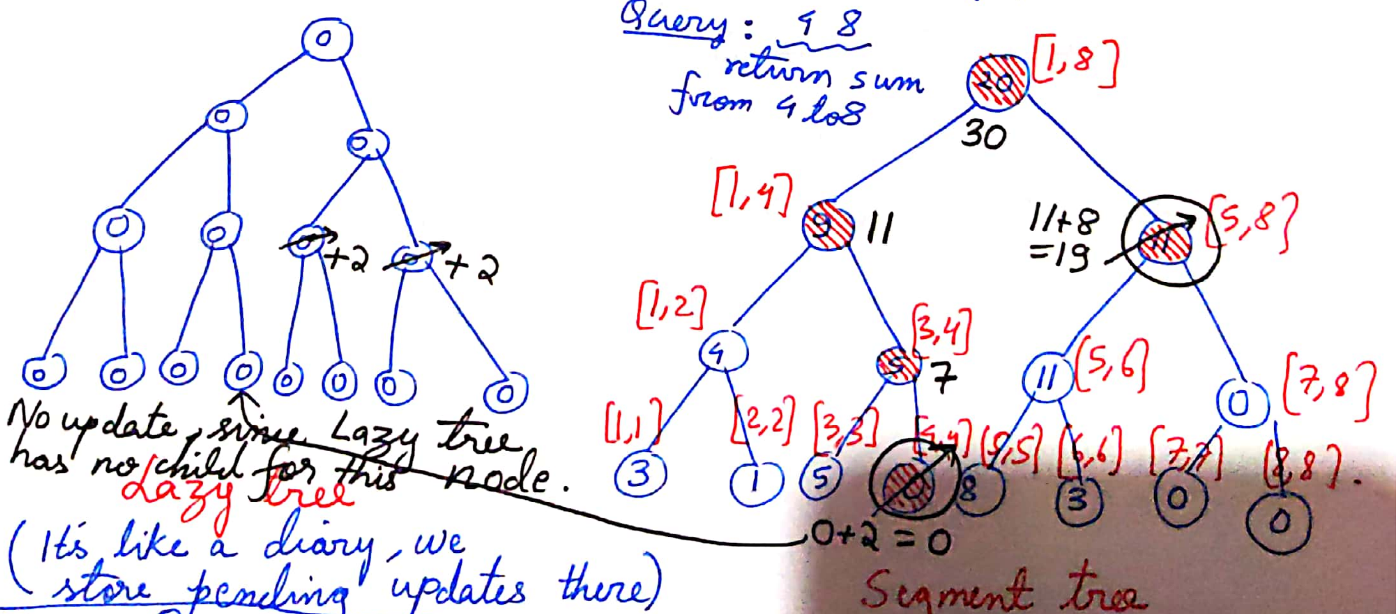
We change arr[qi] to new value, then update SEGTREE.

```
void update (int i, int l, int r, int qi) {
    if (l==r){
        SEGTREE[i] = arr[l];
        return;
    }

    int mid = (l+r)/2;
    if (qi<=mid)  update (2i, l, mid, qi);
    else if(qi>mid)  update (2i+1, mid+1, r, qi);

    SEGTREE[i] = min(SEGTREE[2i], SEGTREE[2i+1]);
    return;
}
```

RANGE UPDATE (LAZY PROPAGATION)

"Update only when needed"

If we do point update, for range → O(Nlogn)
But lazy propagation → O(logn)



(It's like a diary, we store pending updates there)

Query : 2 4 8

[add l r]

Explanation : In a query we are given [add, l, r]. We don't worry about add for now. We traverse the SEGTREE for range [l, r] and find the point nodes that contain this range. Say [4,8] is contained in nodes [4,4] & [5,8].

Now; we update these nodes to new values i.e,

SEGTREE[(4,4)] + = (4-4+1) * add;

SEGTREE[(5,8)] + = (8-5+1) * add;

& update the tree i.e nodes above it as we do for point updates.

Also; in the Lazy tree for these point nodes, we update their child nodes by "+add" value.

Now; if are asked to return sum in [5,6] range.

We traverse the SEGTREE & reach the point nodes that contain this whole range.

For every node, we add value in SEGTREE node & also check "if theres a pending update in "Lazy tree" & again change the Lazy tree's children.