Segment Tree Basics

Problem:
- array with 10^5 items
- 10^5 queries for a program

(i) Brute force approach

- run a for loop for all elements (10^5) in array
- run a for loop for all queries (10^5)

Benefits:

- short and simple code for small dataset

Limitations:

- high time complexity $O(n^2)$. For large dataset it is ($10^5 \times 10^5 = 10^{10}$) very inefficient.

(ii) Cumulative sum approach:

- calculates index-based cumalative sum
- answers a query using difference of cumulative sum.

Benefits:

- Time complexity is reduced drastically from brute force approach and is a lot more efficient.
- The output of a query is determined in $O(1)$ time using pre-build cumulative sum array.

Implementation:

Value	10	20	30	40	50
Index	1	2	3	4	5
Cum_sum	10	30	60	100	150

ith element

addition

cumulative sum of (i-1)th element

- cum[0]=0
- The sum ~~o~~is stored in the index of the ith element.
- Sum is calcalated using

  cum[i] = cum[i-1]+arr[i]

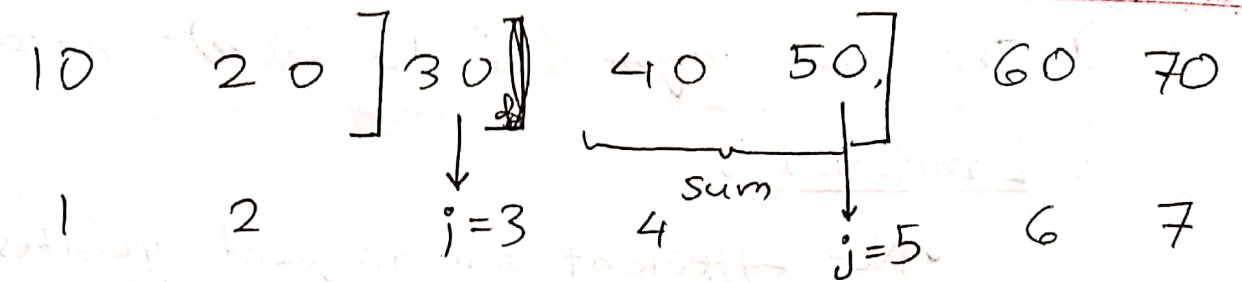
- The result of a query is ~~was~~ always asked for a particular range (i,j) Using the difference of camalative sum, result is determined.

  For result = sum(i,j),

  result = cum[j] - cum[i-1]

It is clear that the difference will result in the cumulative sum within the range.

For



if sum (3,5) is to be found, we can see that we need to subtract the sum before index, i=3 from the cumulative sum upto 5. That is why it is done upto (i-1)th element i.e. cumulative sum from ~~1~~ of (i-1)th element is subtracted.

Code: //indexing strating from 1

```
Build: arr[] = {_, 4,-9, 3,7,1,0,2}   }
    c_sum = 0                          } O(n)
    for (i=1 to n)                     }
        c_sum[i] = sum[i-1] + arr[i]   }
//result
    c_sum = {0,4,-5,-2,5,6,6,8}

Query(i,j)
    return c_sum[j]-c_sum[i-1]         } O(1)
```

The code might lead to O(n) solution.

Limitations:

- Not effecient for frequent updates since time complexity will then change to $O(n^2)$.

(iii) Data structure approach: Segment tree

- We intend to salve RMQ(range min query)
- We use a data structure where cumulative sums are stored in multiple levels in as hierarchial way.

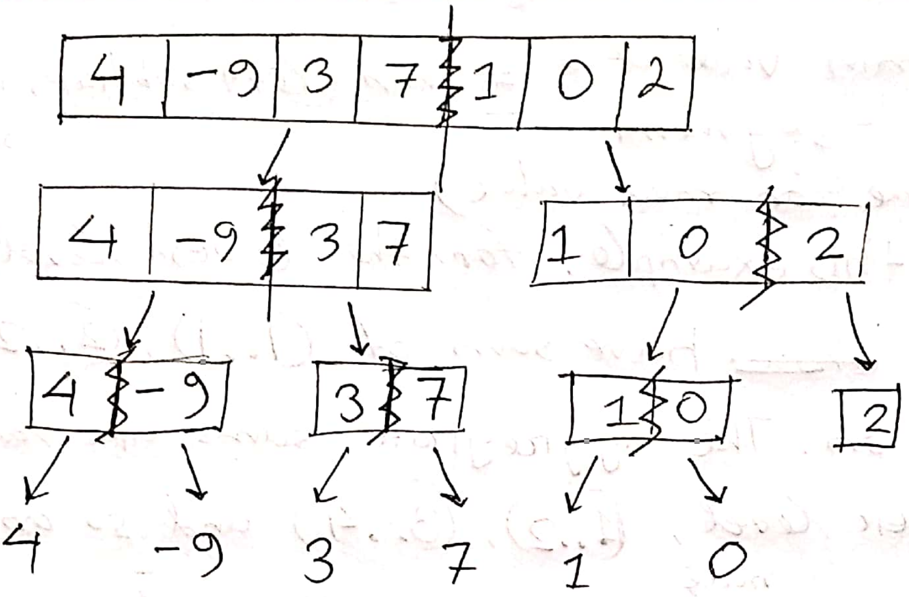
Benefits:

- max, min, sum of all values take O(logn) time complexity
- update can also be done in O(logn).

Implementation:

- visualized as conceptual tree
- leaf nodes are values from original array and intermidiate nodes are derived by adding leaf nodes. Will result RMQ of left and right subtree.

Dividing array



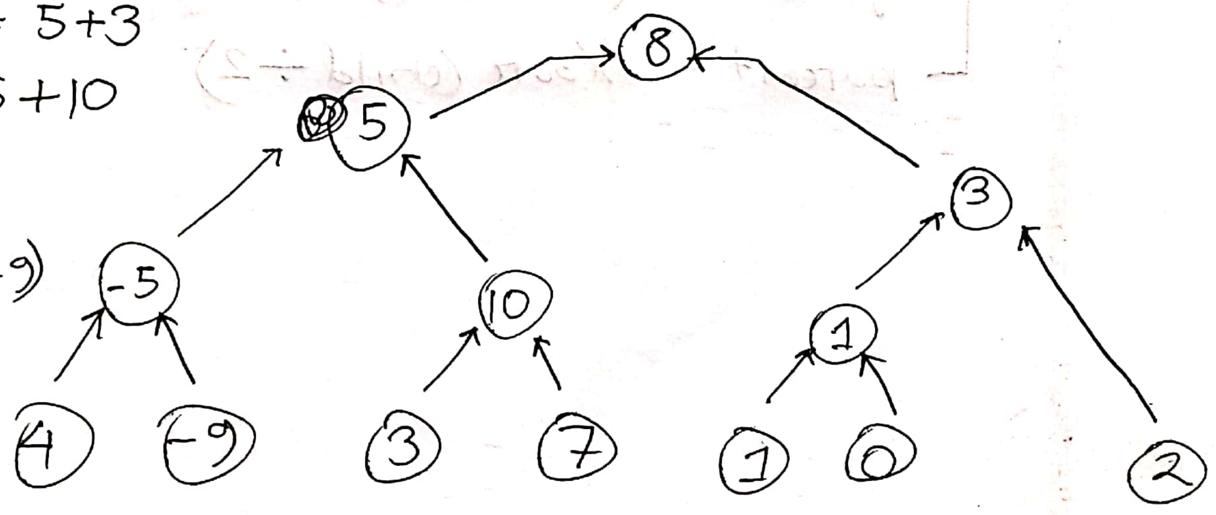
- sum of segment= sum of (left segment + right segment)

- after dividing, the elements are stored as leaf nodes.

- then intermediate rows are calculated,

⟶ sum of a segment = sum (left subtree + right subtree)

- 8= 5+3
- 5 = -5+10
- -5=4+(-9)
- 4 is base value



→ min value of a segment = min (left-subtree, right subtree)

(same for max value)

- In this example, for the lowest level, we ~~those~~ have sum of (1..1), (2..2) and so on. The aggregrate sums are found in upper level, (1..2), (3..4) and so on. The top most node ~~level~~ contains the total sum (1..7).

- Indexing is done from top to bottom.

  as it is conceptual binary tree, we store in array

  - left child = parent × 2
  - right child = (parent × 2) + 1
  - parent = floor (child ÷ 2)

Segment Tree Initialization

Code:

```
// A function is created to initialize the whole
// segment tree
void init (int node, int begin, int end){
        if(begin==end)
              //leaf node found, so, insert directly
            tree[node]=arr[begin]
            return;
 int left= node×2
 int right = node×2+1
 int mid = (begin+end)/2
 // We need to find mid to insert the values in your
 // segment tree's array
 init (left, begin, mid)
 init (right, mid+1,end)
 tree [node]=tree[left]+tree[right]
}
```

Original array

1	2	3	4	5	6	7
4	-9	3	7	1	0	2

where do we insert these values from our original array in the segment tree's array?

We use to mid to find those positions.

Segment tree array

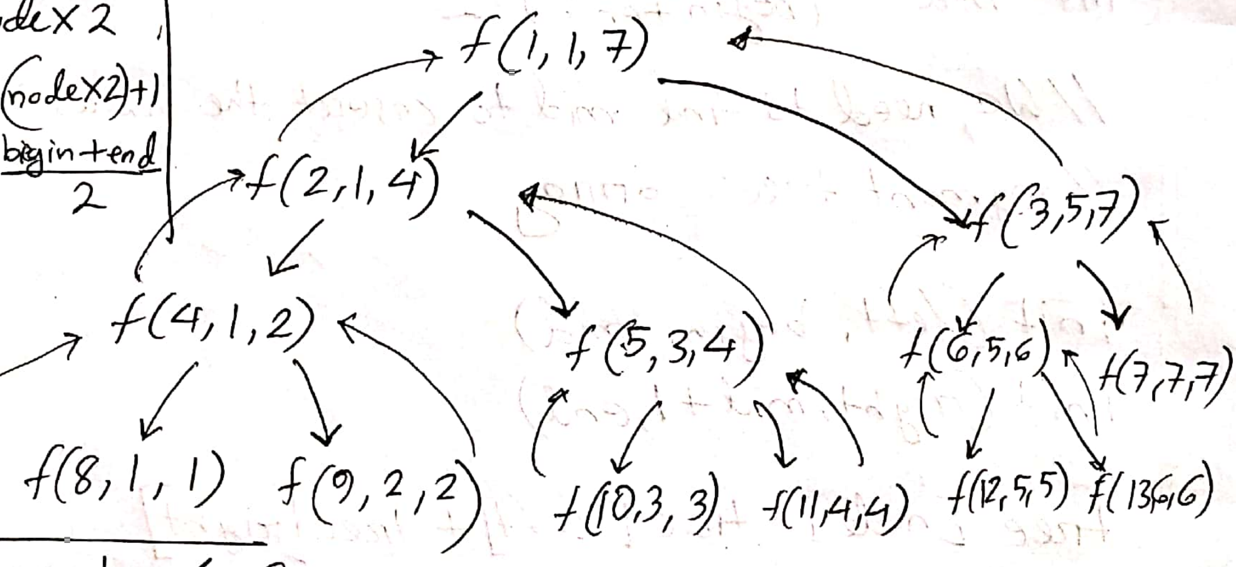
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	5	3	-5	10	1	2	4	-9	3	7	1	0	2	

Recursion - Tree

left = node × 2

right = (node × 2) + 1

mid = $\frac{\text{begin} + \text{end}}{2}$



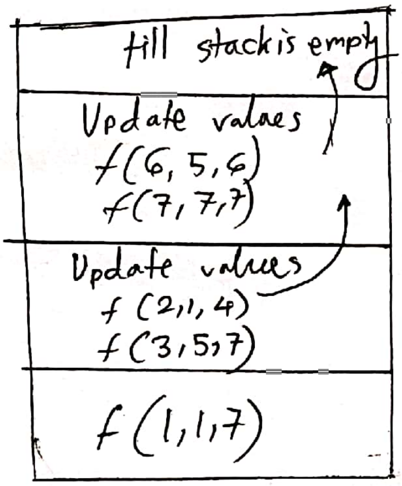
- reaches leaf
- returns recursive function call

✳ The arrows going downwards is basically going to the next recursive step inside that recursive function

(*) The arrows going upwards is basically after getting values from recursive calls.

So, from the top of the hierarchy, the functions recursively goes down to the bottom and then starts setting values from bottom upto the top.

→ The order of function call is maintained using the recursion stack.



follows standard recursion procedure and it ends when stacke gets empty and build is completed.

Complexity: Height of binary tree = logn

~~In worst case, for each element~~

So, to calculate each element, path equal to height of tree is traversed. Each element needs logn time. There are n elements in total. So, total time complexity is O(nlogn).