Segment trees

Need of segment trees

Let us take an example of returning and updating the sum of the subarray a[i....j] of an array of size n.

Example

5	3	2	4	1	8	6	10
0	1	2	3	4	5	6	7

Query: Output the sum from i=1 to i=5.

Update: Update the element at i^{th} index. Example: put a[4] = 13.

Approach 1

For query: Iterate from i=1 to i=5 and calculate the sum.

Time complexity: O(n)

For update: Update the ith index, simply put a[i] = updated_element

Query	Update
O(n)	O(1)

Approach 2 (Prefix Sum Approach)

Build the prefix sum array

Given array:

5	3	2	4	1	8	6	10
0	1	2	3	4	5	6	7

Prefix sum array:

5	8	10	14	15	23	29	39
0	1	2	3	4	5	6	7

For query: Output sum from i to j (0 <= i <= j < n) (say i=1 to i=5)

5	8	10	14	15	23	29	39
0	1	2	3	4	5	6	7

Sum[i....j] = pref[j]-pref[i-1] {if i!=0}

pref[j] {if i = 0}

Time complexity: O(1)

For update: Put a[i] = updated_value

To update in the prefix array, we need to change all pref[i_1] {i_1 >= i}

Example: Update the 4th indexed element to 13.

Original Array becomes:

5	3	2	4	**13**	8	6	10
0	1	2	3	4	5	6	7

Prefix sum array becomes:

5	8	10	14	27	35	41	51
0	1	2	3	4	5	6	7

Time complexity: O(n)

Time complexity of this approach

Query	Update
O(1)	O(n)

If we want both the operations to be in reasonable time, we use segment trees.

<u>Time Complexity comparison table</u>

Approach	Query	Update
Approach 1	O(n)	O(1)
Approach 2	O(1)	O(n)
Segment tree	O(log(n))	O(log(n))

Requirement of log(n) time complexity: Many a times, number of queries and number of updates are of the order of 10^5-10^6, we will get tle if we use Approach 1 or Approach 2.

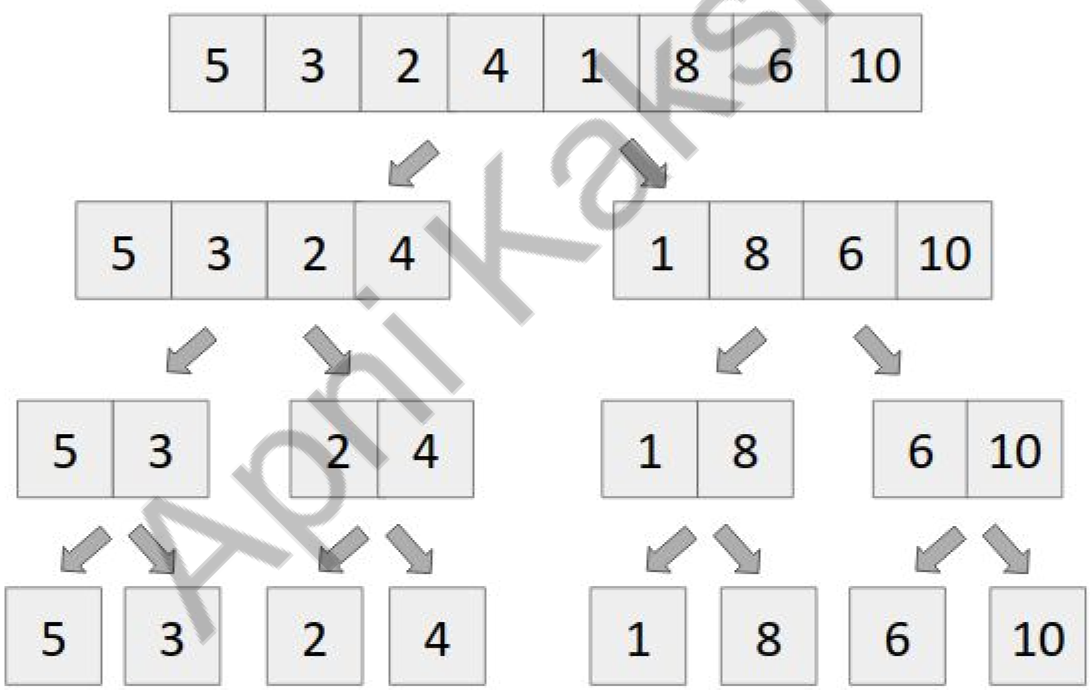
Segment tree construction

Given array:

<u>Power of number 2 in programming</u>

1. Binary Representation of numbers - All operations be it sum / subtraction/ product, all are accomplished in O(1).
2. Division of array (Divide and conquer)

We can divide the above array as



<u>Number of nodes</u> = n + n/2 + n/4 + ... + 2 + 1 , which is geometric progression

Let number of terms in the above G.P. be x, which denotes the height of the segment tree.

We know,

$$ar^{x-1} = n$$

$$putting\ a = 1,\ r = 2,\ we\ get$$

$$(2)^{x-1} = n$$

$$log_2(2^{x-1}) = log_2 n$$

$$x = 1 + log_2 n$$

Number of nodes = $1 + 2 + 4 + ... + n/4 + n/2 + n.$

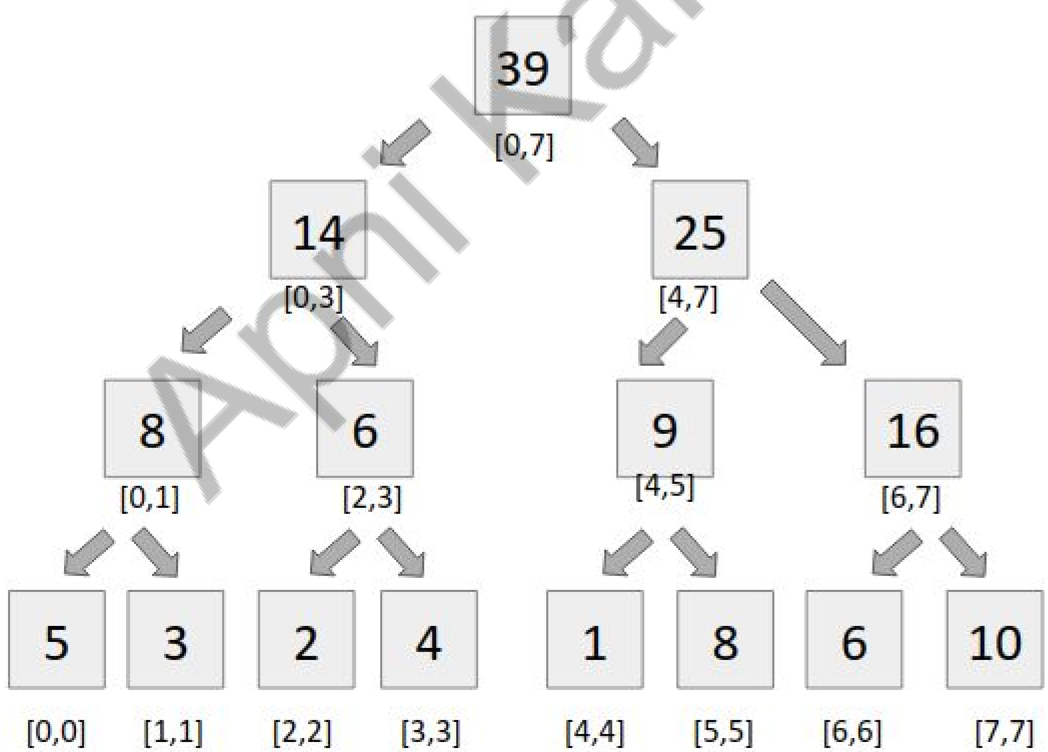
$= 1((2)^{1+log(n)} - 1) / 2-1$

$= 2.2^{log(n)}-1$

$= 2n-1$

For safety, we make segment tree of size <u>4*n</u>.

Structure of segment tree



Building a segment tree

It is very simple to build a segment tree, we use divide and conquer approach to build the segment tree.

Code:

```
int tree[4*N];
int a[N];

void build(int node, int st, int en)
{
    if(st == en)
    {
        tree[node] = a[st];
        return;
    }

    int mid = (st+en)/2;
    build(2*node, st, mid);
    build(2*node+1, mid+1, en);

    tree[node] = tree[2*node] + tree[2*node+1];
}
```

Query

For query, we see two types of segments

- Complete overlapping segments - When our st Partial overlapping segments and en lies completely in the range [l,r], it is called complete overlapping segment.
- Partial overlapping segments - When our st and en does not lie completely in the range [l,r], it is called partial overlapping segment.

Code:

```
int query(int node, int st, int en, int l, int r)
{
    if(st>r || en<l)
        return 0;      // may change according to the conditions

    if(l<=st && en<=r)
        return tree[node];

    int mid = (st+en)/2;

    int q1 = query(2*node, st, mid, l, r);
    int q2 = query(2*node+1, mid+1, en, l, r);

    return q1 + q2;
}
```

Update

Updating an element in the segment tree is very similar to binary search.

We find out mid, and compare our index with mid and two conditions arise

1. Idx <= mid, then we recursively call the left child of the tree's node.
2. Idx > mid, then we recursively call the right child of the tree's node.

Code:

```
void update(int node, int st, int en, int idx, int val)
{
    if(st == en)
    {
        a[st] = val;
        tree[node] = val;
        return;
    }

    int mid = (st+en)/2;

    if(idx<=mid)
      update(2*node, st, mid, idx, val);
    else
      update(2*node+1, mid+1, en, idx, val);

    tree[node] = tree[2*node] + tree[2*node+1];
}
```