Algorithm Engineering

Md. Atiqur Rahman

Segment Tree

Managing Electricity Usage in a Smart City

A smart city tracks the electricity usage (in kilowatt-hours) of N households along a single street. The city's control center needs to process two types of operations efficiently:

- Update Consumption:Sometimes a household's meter reading is corrected after an inspection. This means replacing the recorded consumption for that household with a new value.
- Neighborhood Consumption Report:City officials may request the total electricity usage for a continuous block of houses (for example, houses 10 through 25) to analyze neighborhood consumption patterns.

Given the number of houses and their initial recorded consumption values, you must process Q operations and provide answers for all neighborhood consumption reports.

Input Format

N Q

C[1] C[2] ... C[N]

query_1

query_2

...

query_QInput Format

Each query is in one of the following forms:

- 1 H V → Update household H's consumption to V kWh.
- 2 L R → Output the total consumption for all households from L to R (inclusive).

A test case

5 5
2 1 3 4 5
2 1 5
1 3 10
2 2 4
1 5 7
2 4 5

Skeleton Algorithm

We need to maintain the following query properties,

- update(H, V): set consumption of house H to V.
- query(L, R): sum consumption from house L to R (inclusive).
- Both update and query definition must be implemented in a way so that it requires O(log N).

The layout to save the data,

- Use an iterative segment tree stored in an array of size 2*P, where P is the next power of two ≥ N.
- Leaves (original array values) live at indices [P .. P+N-1].
- Parents store the sum of their two children.

Building the aforementioned tree,

- Copy C[i] into tree[P+i].
- For i = P-1 down to 1: tree[i] = tree[2*i] + tree[2*i+1].

Pseudo Code To Build The Tree

```
def next_pow2(n):

    p = 1

    while p < n:

    p <<= 1

    return p

P = next_pow2(N)
```

```
tree = [0] * (2 * P)


for i in range(N):
    tree[P + i] = C[i]

for i in range(P - 1, 0, -1):
    tree[i] = tree[2 * i] + tree[2 * i + 1]
```

Continued

Point update [O(logn) complexity],

- Convert house index H (1-based) to leaf position pos = P + (H-1).
- Set tree[pos] = V.
- Climb up: pos //= 2 each step, recomputing tree[pos] = tree[2*pos] + tree[2*pos+1].

Range Sum Query [O(logn) complexity],

- Convert [L, R] (1-based) to 0-based half-open in the leaf layer: l = P + (L-1), r = P + R (note: r is exclusive in this iterative style).
- While l < r:
  - If l is a right child, include tree[l] and l += 1.
  - If r is a right boundary, move left: r -= 1 and include tree[r].
  - Move both up: l //= 2, r //= 2.
- The accumulated sum is the answer.

Pseudo Code For Update

```
def update(h, v):
    pos = P + (h - 1)
    tree[pos] = v
    pos //= 2
    while pos >= 1:
        tree[pos] = tree[2 * pos] + tree[2 * pos + 1]
        pos //= 2
```

Pseudo Code For Update

```
def query(l, r):
    l = P + (l - 1)
    r = P + r
    res = 0
    while l < r:
        if l & 1:
            res += tree[l]
            l += 1
        if r & 1:
            r -= 1
            res += tree[r]
        l //= 2
        r //= 2
    return res
```

Some questions

1. Why N is the “next power of two” (P)?

- In a complete binary tree, all leaves are at the same depth.
- If N is not a power of two, the tree won’t be perfectly balanced; some leaves will be missing.
- Padding up to the next power of two (P) makes indexing predictable — the leaves occupy exactly positions P through P+P-1.
- This also allows us to use simple parent/child index math:
    - Parent of node i → i // 2
    - Left child of i → 2 * i
    - Right child of i → 2 * i + 1

Some questions

2. Why 2 * P is the total size?

- The tree has:
  - P leaves (actual data + padding)
  - P - 1 internal nodes (sums of children)
  - Total nodes = P + (P - 1) = 2*P - 1
- We allocate 2 * P slots instead of 2*P - 1 just to make indexing clean and 1-based:
  - Node 1 = root
  - Leaves start at index P
  - We ignore index 0 for simplicity.

Continued

What is the total number of nodes (including leaves), if a perfect binary tree has P leaves node?

Always 2P-1.

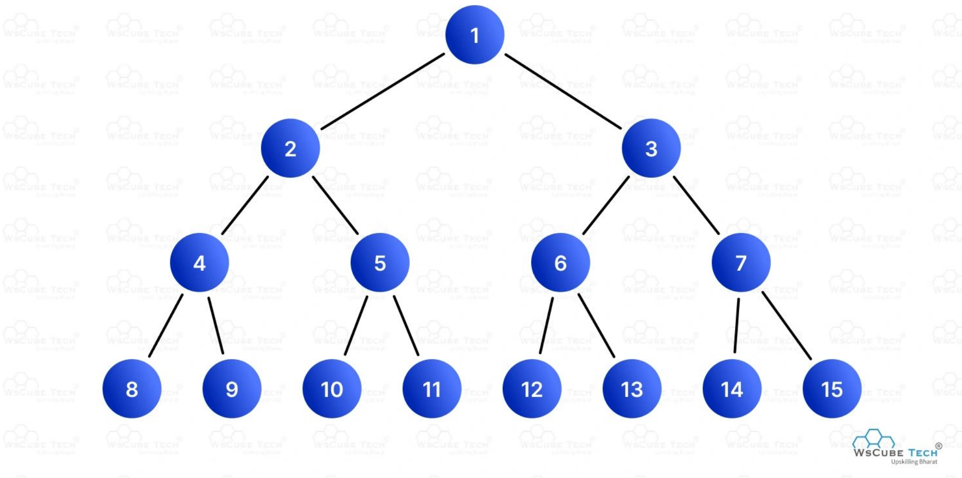
Why 2P memory allocation works?

In the extreme case scenario, the next power of 2 of n can be 2^{x}. So our P can be 2^{x} that is equal to n, and if we allocate 2P

that has P leaves. Since, $2P-1 < 2P$.

Two properties of perfect binary tree:

1. Every internal node has exactly two children.
2. All leaf nodes are at the same level (or depth).



Some questions

3. Why make r exclusive?

We want to query [L, R] inclusive from the user’s perspective,

but the iterative algorithm internally works on [l, r) — half-open intervals — because:

- It simplifies the loop: while l < r
- It lets us stop exactly when the two pointers meet without double-counting.
- It matches how array slicing works in Python ([start, end)).

Continued

4. Significance of l being a right child (l & 1)

In the tree array:

- Left child indices are even.
- Right child indices are odd.

If l is a right child, it means:

- Its parent’s segment starts before l.
- So, if we included the whole parent, we’d be including stuff before the query range.
- Therefore, we take this single node (add it to the sum), and then move l to the next segment (l += 1).

Conituned

5. Significance of r being at a right boundary (r & 1)

Remember r is exclusive — it points to the segment after the range.

If r is a right child, that means:

- The segment just before r is the last part of the query.
- So we first move back (r -= 1), then take that node’s value.

Recursive Segment Tree

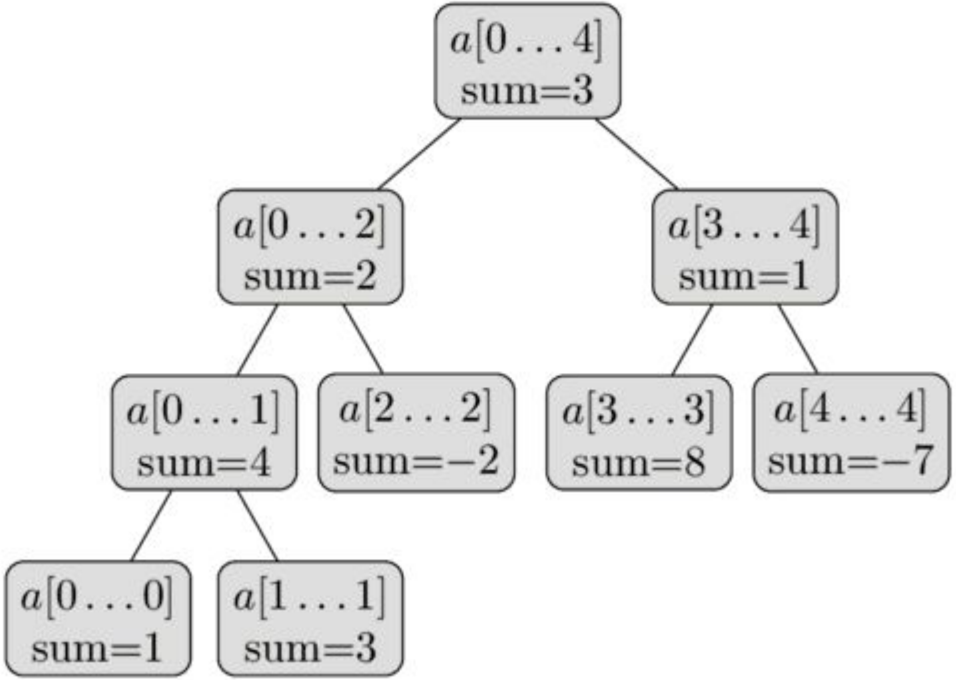
To implement the right side segment tree we need 4n allocation of memory space.

Now, tell me, is 2P > 4n?

Let, 2^{x} <= n < 2^{x+1}, now highest value of n can be 2^{x+1} - 1, in that case P will be 2^{x+1}. So, 2P will be 2^{x+2}.

Similarly, 4 which is 2^{2}, multiplied to 2^{x+1}-1, we get 2^{x+3}-2^{2},

These two terms will only be equal if x is 0 other than that the second term will always be higher. Then, why recursive approach requires higher memory?



Build Function

If tl = 0, tr = n-1, v = 1, n = 5 and a = [1,3,-2,8,-7],

Write all the elements reside inside t array after executing the right side function.

int t[4*n] = {-1};

```
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

Is 4n enough?

What is the relation between height of a perfect binary tree with the number of nodes it has?

Let, h be the height of the the perfect binary tree then, number of nodes is 2^{h}-1.

What is the length of the highest range we can find sum in a segment tree?

n.

Now, this range of size n is divided by 2 for each level of the segment tree. Thus, the maximum height of the tree can be upper_bound(log_{2}n) + 1. So even if we consider our segment tree as perfect, we have 2^{upper_bound(log_{2}n) + 1} -1 nodes.

2^{upper_bound(log_{2}n) + 1} -1 < 2^{upper_bound(log_{2}n) + 1} < 2 x 2^{upper_bound(log_{2}n) + 1}

=> 2 x 2^{upper_bound(log_{2}n)} x 2 = 4 x n

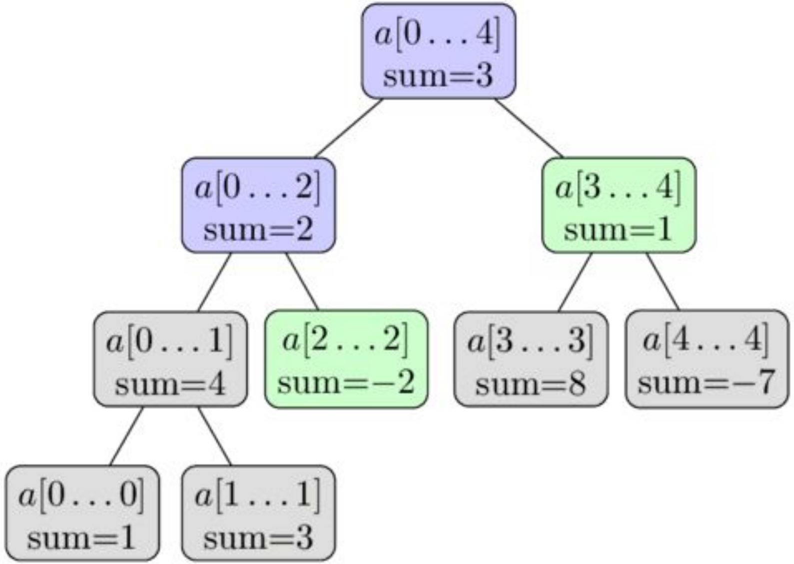
Since, 4n is higher than the maximum nodes possible, it is enough.

Sum Queries

v=1, tl=0, tr=4, l=2, r=4,

Which nodes in the segment tree, I will visit if we have the above values for the right side sum function?

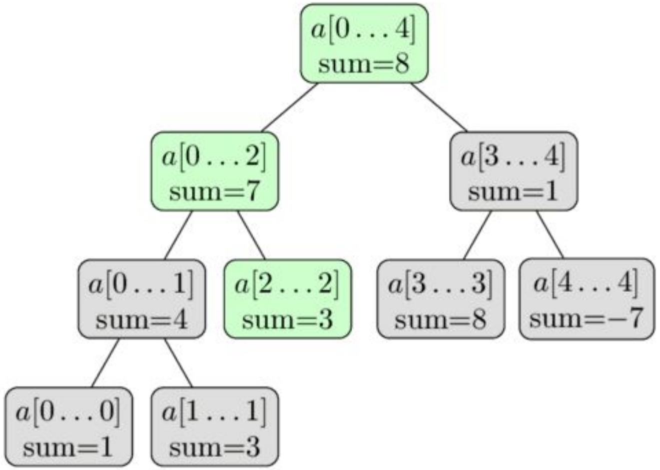
```
int sum(int v, int tl, int tr, int l, int r) {
    if (l > r)
        return 0;
    if (l == tl && r == tr) {
        return t[v];
    }
    int tm = (tl + tr) / 2;
    return sum(v*2, tl, tm, l, min(r, tm))
           + sum(v*2+1, tm+1, tr, max(l, tm+1), r);
}
```



Update Queries

Which nodes of the segment tree, I will visit if v=1, tl=0, tr=4, pos = 2, new_val = 3?

```
void update(int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) {
        t[v] = new_val;
    } else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update(v*2, tl, tm, pos, new_val);
        else
            update(v*2+1, tm+1, tr, pos, new_val);
        t[v] = t[v*2] + t[v*2+1];
    }
}
```



Memory efficient Implementation

If you look at the array t you can see that it follows the numbering of the tree nodes in the order of a BFS traversal (level-order traversal). Using this traversal the children of vertex v are $2v$ and $2v + 1$ respectively.

However if n is not a power of two, this method will skip some indices and leave some parts of the array t unused. The memory consumption is limited by $4n$, even though a Segment Tree of an array of n elements requires only $2n - 1$ vertices.

However it can be reduced. We renumber the vertices of the tree in the order of a pre-order traversal, and we write all these vertices next to each other.

If a node is responsible for [l,r], the number of vertices it needs is 2(r-l+1) -1.

Now, left child's node is v+1, but the number of vertices it needs is, 2(mid-l+1)-1, so the right child's vertex number v+2(mid-l+1).

Implement the build, query and update function using the approach explained in this slide.