

Segment Tree and Lazy Propagation

beOI Training

OLYMPIADE BELGE D'INFORMATIQUE
BELGISCHE INFORMATICA-OLYMPIADE

December 28, 2025

Table of Contents

Regular Segment Tree

Lazy Segment Tree

Other Topics

Motivating problem

You are given an integer array A of size n ($n < 10^6$).
Given two integers a and b , can you give the sum of the entries in A between indices a and b ?

$$A[a] + \dots + A[b - 1]$$

Well that's easy, just iterate over the interval and sum!

Motivating problem

You are given an integer array A of size n ($n < 10^6$).
Given two integers a and b , can you give the sum of the entries in A between indices a and b ?

$$A[a] + \dots + A[b - 1]$$

100000 times?

This is called the **range sum query** (RSQ) problem.

Naive solution

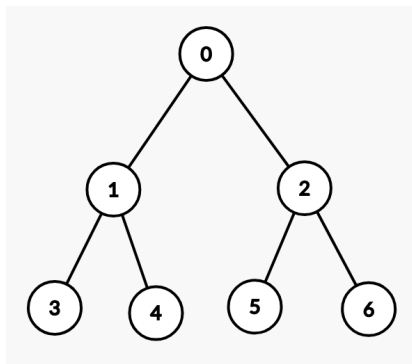
For each query, iterate over the corresponding range and sum the entries.

If k is the number of queries, time complexity is $\mathcal{O}(nk)$.

TLE

Array representation of a binary tree

- ▶ 0-based array, index 0 = root
- ▶ For each node of index p ,
 - ▶ left child has index $2p + 1$
 - ▶ right child has index $2p + 2$



Segment Tree

Each node is responsible of one segment

Root represents the whole array $[0, n[$

Given a node representing segment $[l, r[$

- ▶ left child represents the segment's first half $[l, \frac{l+r}{2}[$
- ▶ right child represents the segment's second half $[\frac{l+r}{2}, r[$

The value of a node will be the **sum of the entries in segment** $[l, r[$.

Querying

When we query the sum in an interval, we look for **big segments that are contained within the query range**, and sum their values.

Recursively,

- ▶ segment is within query range \Rightarrow return value of the node;
- ▶ segment and query range are disjoint \Rightarrow do nothing;
- ▶ otherwise, return sum of both children.

Querying implementation

```
// p is array index of current node,  
// [L,R[ is current segment,  
// [i,j[ is search interval  
// returns: position of the minimum element  
int query(int p, int L, int R, int i, int j) {  
    // inside query range  
    if (i <= L && R <= j) return st[p];  
    // outside query range  
    if (i >= R || L >= j) return 0;  
    // sum the left and right subintervals  
    return query(2*p+1, L, (L+R)/2, i, j)  
        + query(2*p+2, (L+R)/2, R, i, j);  
}  
  
// Starting a query:  
query(0, 0, n, i, j);  
// CAREFUL with i, j: 0-indexed, inclusive-exclusive!
```

Querying complexity

At each level, at most 4 nodes are visited (see coach for proof).

There are exactly $\lceil \log_2 n \rceil$ levels.

$$\mathcal{O}(4 \times \lceil \log_2 n \rceil) = \mathcal{O}(\log n)$$

Overall complexity $\mathcal{O}(k \log n)$ is now reasonable!

AC

Building

Building the Segment Tree is also done recursively.

For each node,

- ▶ if no child, store current value;
- ▶ otherwise,
 - ▶ build left child;
 - ▶ build right child;
 - ▶ store sum of children.

Building implementation

```
void build(int p, int L, int R, vector<int> const& A) {  
    if (L+1 == R) {  
        // Single element in the segment  
        st[p] = A[L];  
    } else {  
        // Build both children and then combine  
        build(2*p+1, L, (L+R)/2, A);  
        build(2*p+2, (L+R)/2, R, A);  
        st[p] = st[2*p+1] + st[2*p+2];  
    }  
}  
  
// Call with:  
build(0, 0, n, A);
```

Building complexity

We visit every node once.

In general, the number of nodes is

$N + \frac{N}{2} + \frac{N}{4} + \cdots + 2 + 1 \approx 2N$, so time complexity is

$$\mathcal{O}(2 \times N) = \mathcal{O}(N)$$

This also proves memory is $\mathcal{O}(N)$ (in practice one always takes an array of $4 \times N$ for safety).

Segment Trees are extremely powerful!

We saw how to solve the range **sum** query problem.
But we can do much more than that!

- ▶ Range minimum query
- ▶ Range maximum query
- ▶ Range *insert any function here* query

Which functions? We just need a **monoid**:

- ▶ A set of elements (numbers, strings, a custom struct, ...)
- ▶ An associative operation (+, min, gcd, string +, ...)
- ▶ An identity element (respectively 0, ∞ , 0, "", ...)

One last operation

Suppose that, between queries, the array is being **updated**.
Naive solution: re-build the Segment Tree in $\mathcal{O}(N)$.

TLE

Segment Trees allow efficient **updating**!

Updating

To update p , we only need to update the segments that contain p .

Update the leaf to root path in $\mathcal{O}(\log N)$!

Updating implementation

```
// i is the node that is to be updated  
// x is the new value  
void update(int p, int L, int R, int i, int x) {  
    if (L+1 == R) {  
        // Single element in the segment  
        st[p] = x;  
    } else {  
        // Build both children and then combine  
        if (i < (L+R)/2) {  
            update(2*p+1, L, (L+R)/2, i, x);  
        } else {  
            update(2*p+2, (L+R)/2, R, i, x);  
        }  
        st[p] = st[2*p+1] + st[2*p+2];  
    }  
}
```

```
// Call with:  
update(0, 0, n, i, x)  
// Careful: i is 0-indexed
```

Table of Contents

Regular Segment Tree

Lazy Segment Tree

Other Topics

Motivating problem

In the Range Sum Query (RSQ) problem, we add one operation: range update.

We want to update a range (e.g. increment every value in range by dx) efficiently.

Naive solution

At each range update query, re-build tree in $\mathcal{O}(N)$.

TLE

Let's be lazy!

Key idea behind lazy Segment Tree: don't update everything at once; put a flag on segments that need to be updated, and leave it for another traversal.

Propagation

Keep an array `lazy` that stores for each segment by how much each value needs to be incremented.

Every time we visit a node p (in query or update) where `lazy[p] != 0`,

- ▶ increment current segment by `lazy[p]` times size of segment;
- ▶ if node is not leaf,
 - ▶ increment `lazy[2*p+1]` by `lazy[p]`
 - ▶ increment `lazy[2*p+2]` by `lazy[p]`
- ▶ reset `lazy[p]`.

That is called **propagation**.

Obviously, complexity is $\mathcal{O}(1)$.

Propagation implementation

```
void propagate(int p, int L, int R) {  
    if (lazy[p] != 0) {  
        st[p] += (R-L)*lazy[p];  
  
        if (L+1 != R) {  
            lazy[2*p+1] += lazy[p];  
            lazy[2*p+2] += lazy[p];  
        }  
  
        lazy[p] = 0;  
    }  
}
```

Querying

We do exactly the same, but we propagate at each node!
Complexity $\mathcal{O}(\log N)$.

Querying implementation

```
int query(int p, int L, int R, int i, int j) {  
    // This line is new:  
    propagate(p, L, R);  
  
    if (i <= L && R <= j) return st[p];  
    if (i >= R || L >= j) return 0;  
  
    return query(2*p+1, L, (L+R)/2, i, j)  
        + query(2*p+2, (L+R)/2, R, i, j);  
}
```

Updating

For each node,

- ▶ propagate
- ▶ if outside of range, return
- ▶ if inside of range, set the lazy flag, and return
- ▶ otherwise
 - ▶ update left child
 - ▶ update right child
- ▶ merge both children (add them up)

Complexity $\mathcal{O}(\log N)$.

Updating implementation

```
// i, j: update range
// dx: by how much to increment
void update(int p, int L, int R, int i, int j, int dx) {
    // inside update range
    if (i <= L && R <= j) {
        lazy[p] += dx;
        propagate(p, L, R);
        return;
    }
    // outside update range
    if (i >= R || L >= j) return 0;

    update(2*p+1, L, (L+R)/2, i, j, dx);
    update(2*p+2, (L+R)/2, R, i, j, dx);

    st[p] = st[2*p+1] + st[2*p+2];
}
```

Table of Contents

Regular Segment Tree

Lazy Segment Tree

Other Topics

Segment Trees are extremely powerful! (part 2)

Many combinations of queries and updates can be solved with segment trees!

- ▶ Range minimum query with $+= dx$ updates: minimum position does not change in the interval.
- ▶ Range sum query with $\times=$ product updates: sum over interval is multiplied by product.
- ▶ And so on...

Iterative segment trees

- ▶ Shorter and more efficient than the recursive segment trees we have seen so far.
- ▶ My opinion: more difficult to remember, **not worth it** for IOI.
- ▶ See: <https://codeforces.com/blog/entry/18051>

Fenwick trees

- ▶ Faster to write, but less flexible: operation must be invertible (e.g. RSQ but not RMQ).
- ▶ Range updates are possible but complicated.
- ▶ See: 09-fenwick-trees.

"Dynamic" segment trees

- ▶ What if we cannot fit the entire range in memory?
- ▶ For example: Range Sum Query on array of size 10^9 , initialized with 0, but updated later.
 - ▶ Memory usage: 4 bytes per integer $\times 4n$ of storage.
 - ▶ $4 \times 4 \times 10^9 \approx 16$ Gb.
 - ▶ Memory Limit Exceeded
- ▶ Key idea \Rightarrow Use an unordered_map instead of a vector, build tree as needed.

2D segment trees

- ▶ Key idea \Rightarrow Inside each node of an outer segment tree, store an inner segment tree.
- ▶ Allows for queries in $\mathcal{O}(\log n \times \log n)$.
- ▶ Very complicated to implement.
- ▶ See also: Game from IOI 2013.
- ▶ (A quad tree does not work, worst case complexity is $\mathcal{O}(n)$.)