Segtrees I

Intro to the S-Tier data structure

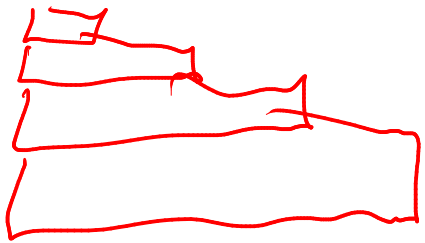
Warmup: Range sum query

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

“Given 1 <= L <= R <= N, return the sum a[L] + a[L+1] + … + a[R]”

Target complexity: O(N) precalculation, O(1) per query.

Cumsum



Range min query?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

“Given 1 <= L <= R <= N, return the minimum min(a[L],a[L+1] … a[R])”

What techniques do you know to solve this?

Naïve: O(n) query O(N) precalc

Range min query?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

“Given 1 <= L <= R <= N, return the minimum min(a[L],a[L+1] … a[R])”

What techniques do you know to solve this?

There’s quite a few (https://cp-algorithms.com/sequences/rmq.html)

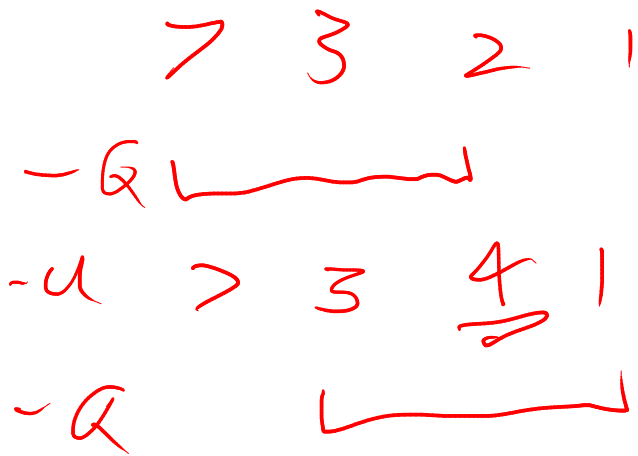
- Sqrt decomp: O(n) preprocessing, O(sqrt(n)) query
- **Segment tree: O(n) preprocessing, O(log(n)) query** – admit updates
- Sparse table: O(nlogn) preprocessing, O(1) query
- Sqrt Tree: O(nlog(logn)) preprocessing, O(1) query
- Arpa’s trick: O(n)-ish preprocessing, O(1) query (must be offline)
- Cartesian tree + Farach-Colton Bender Algorithm: O(n) preprocessing, O(1) query

Range ~~min~~ sum query + point updates

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Returns a[L] + a[L+1] + … + a[R]

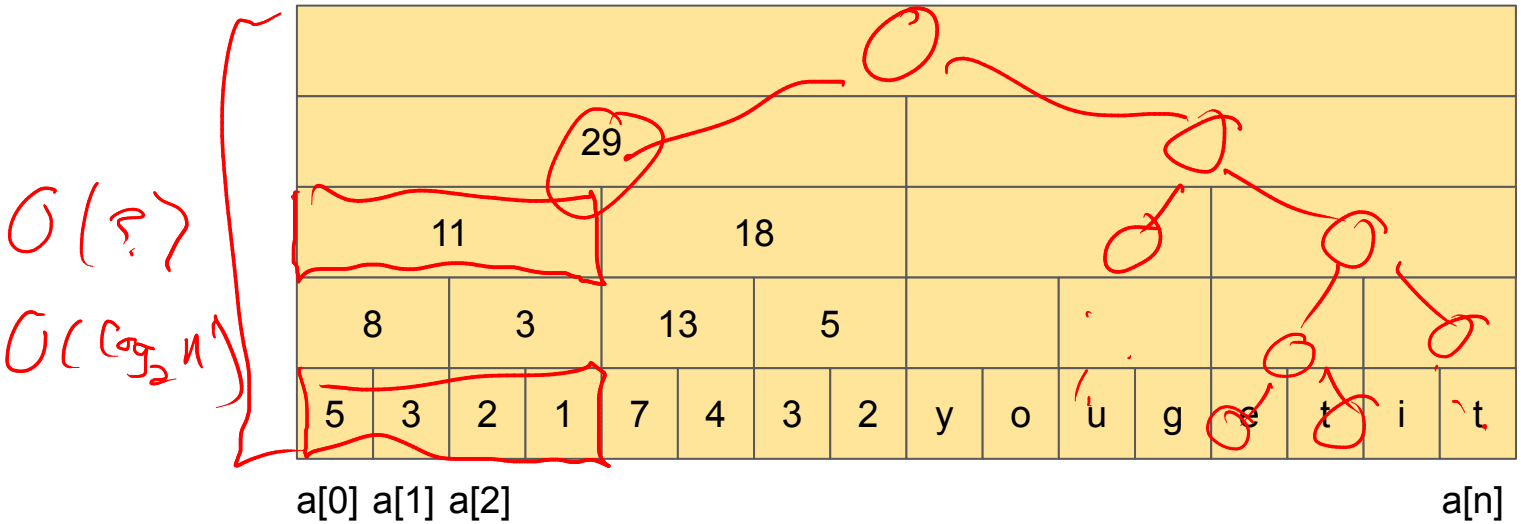
(Update) X Y: Change the value of a[X] to Y



Idea: Precalc blocks of size 2^i

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

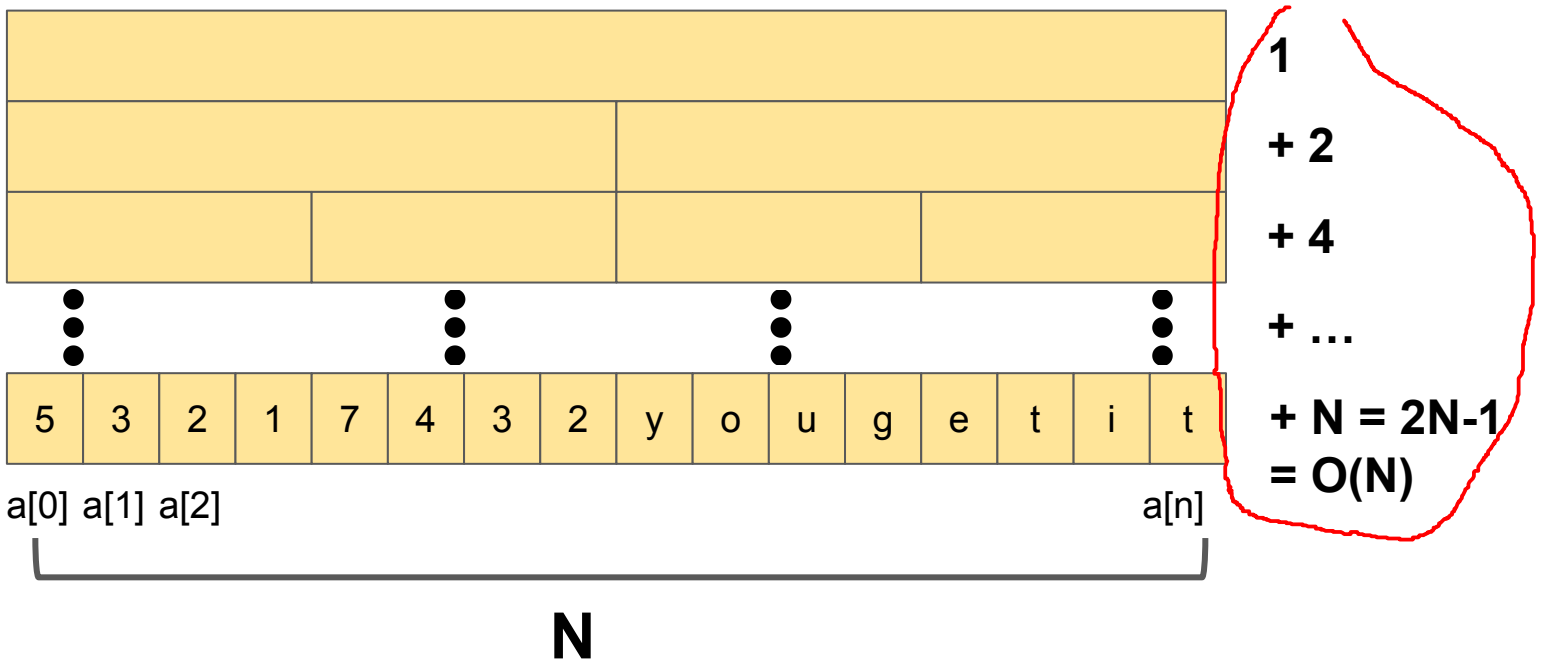
(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y



Number of things you have to precalc?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y

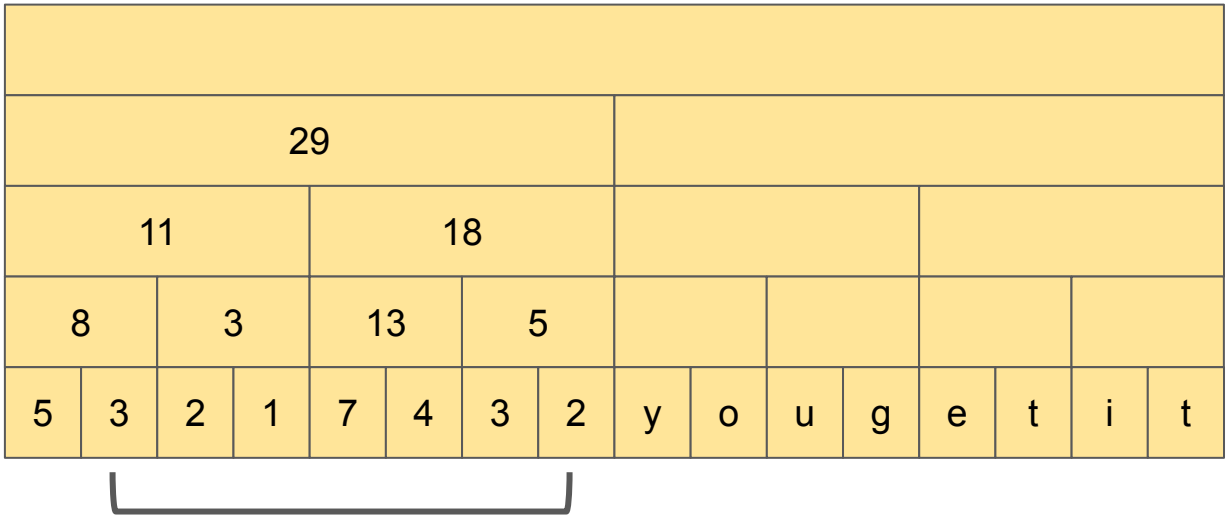


Later in implementation: how to precalc in O(N)

Query examples…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

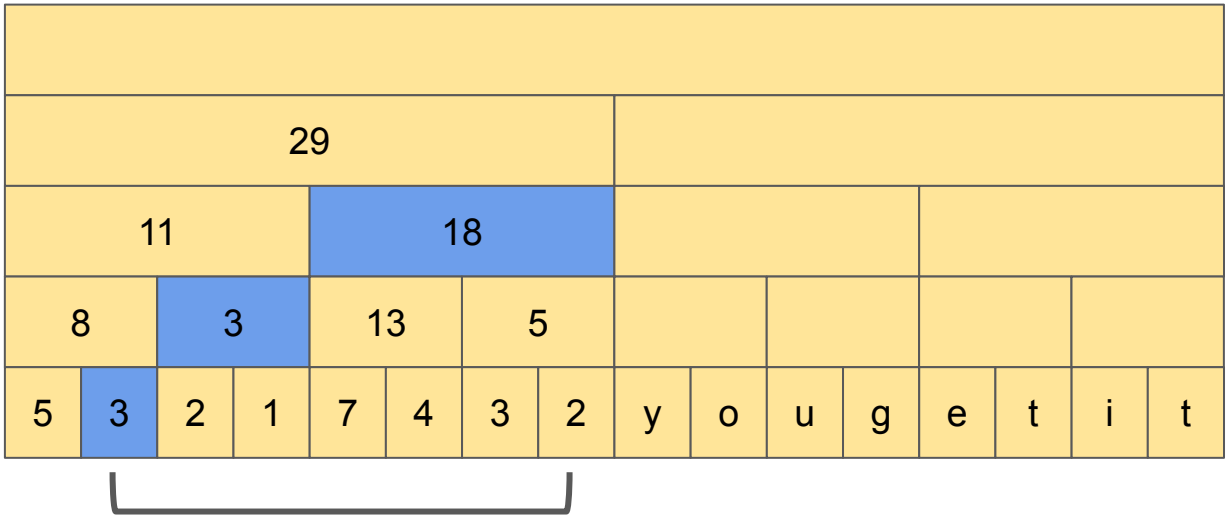
(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y



Query examples…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y

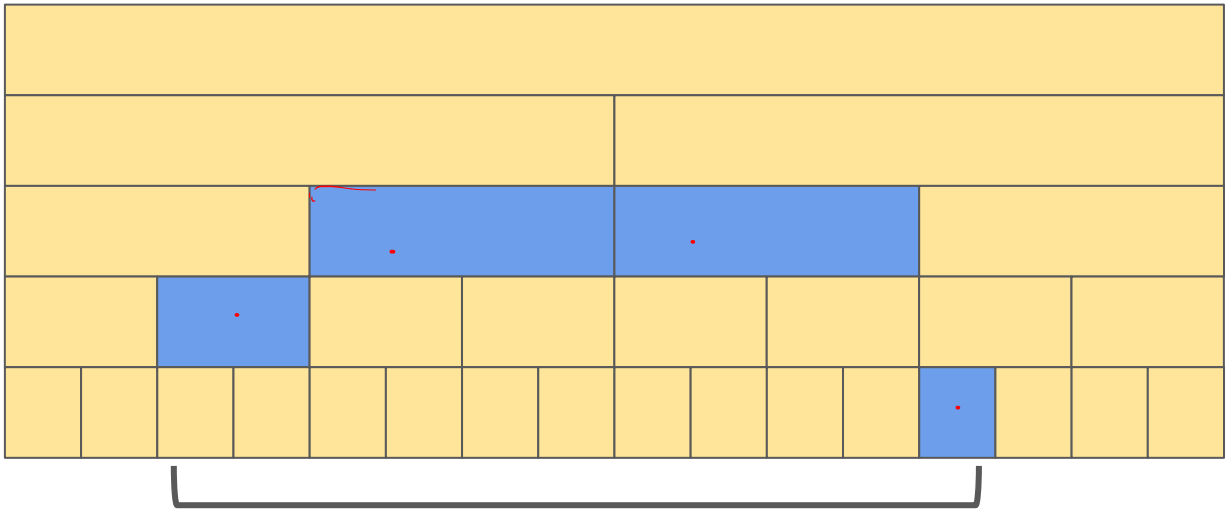


Querying sum of 7 elements, requires 3 blocks.

Query examples…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y

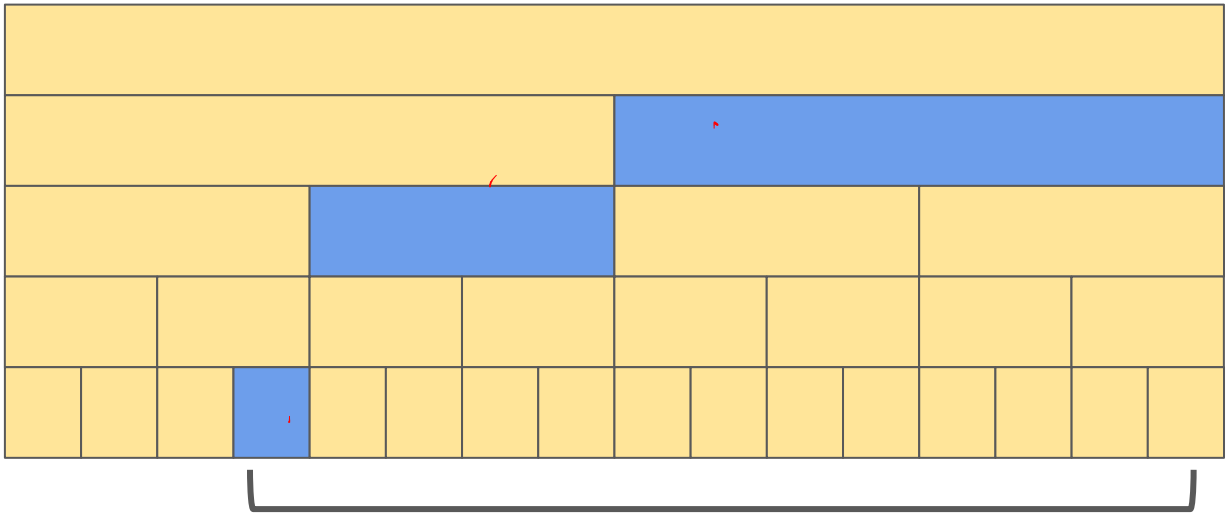


Querying sum of 11 elements, requires 4 blocks.

Query examples…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y

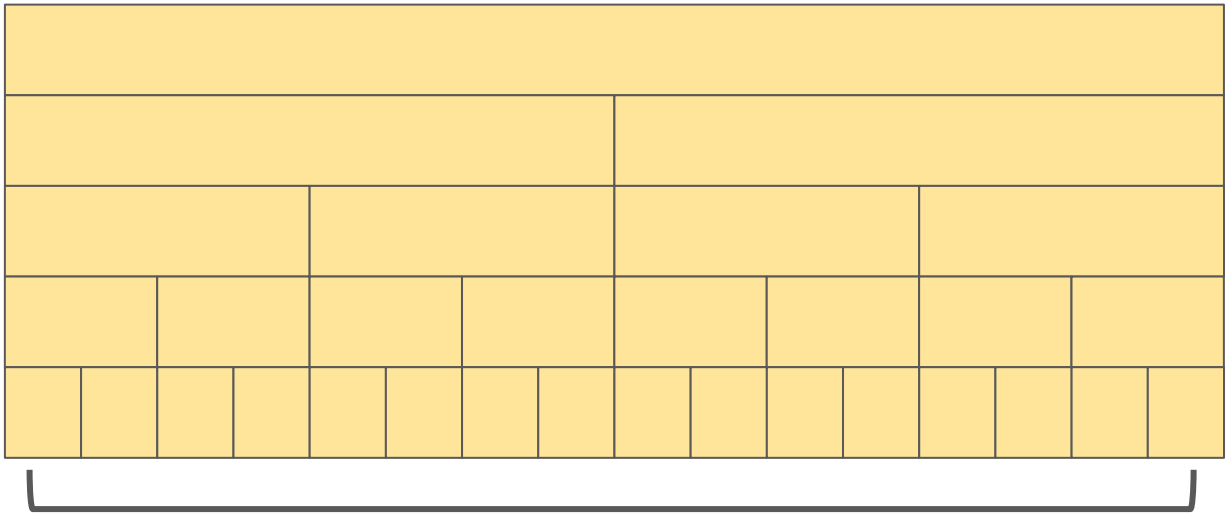


Querying sum of 13 elements, requires 3 blocks.

Query examples…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y

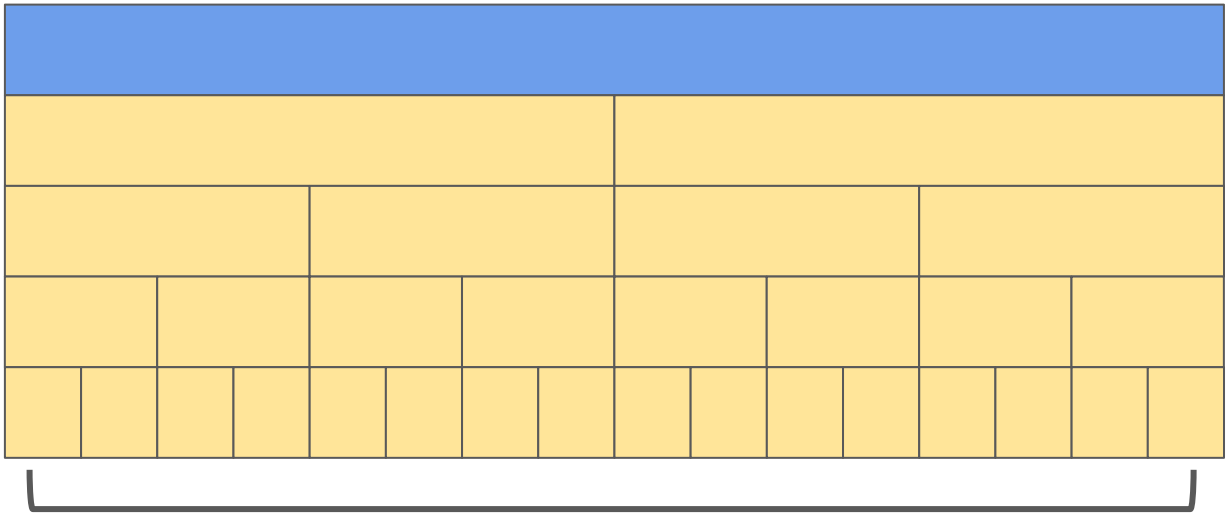


Querying sum of 16 elements, which blocks are required?

Query examples…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y

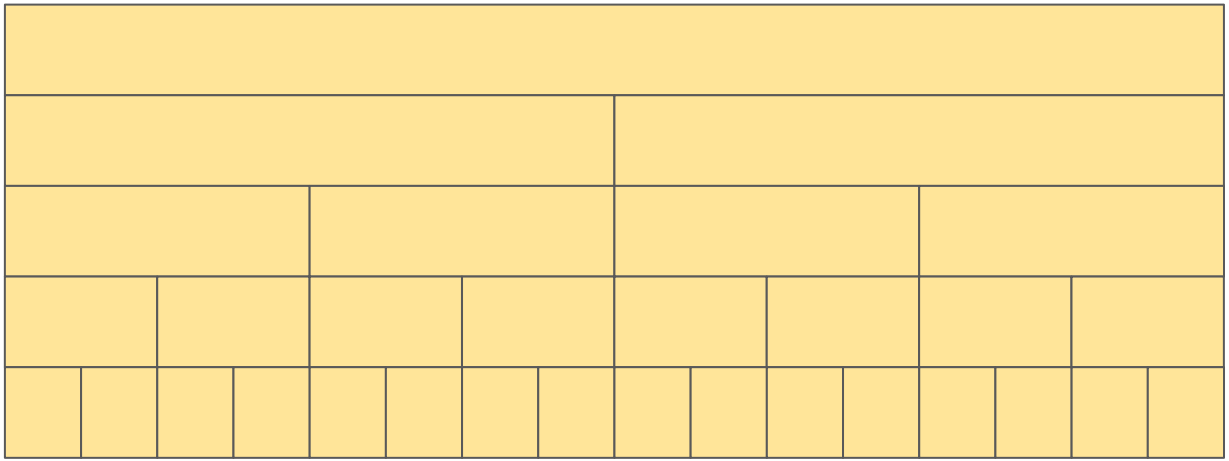


Querying sum of 16 elements, requires 1 block

Query time complexity?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y



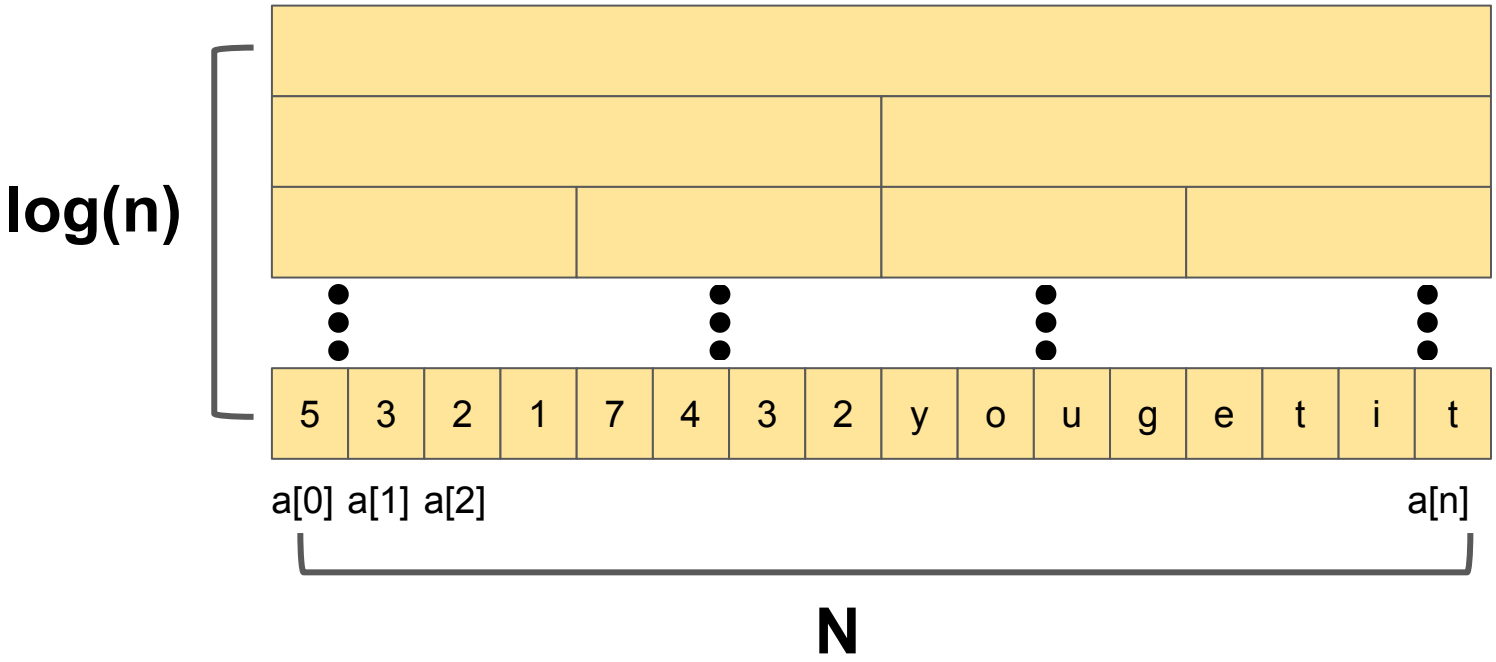
Each query uses at most 2log(n) blocks, and can be calculated in O(logn) time.

This fact will be made a bit more clear in implementation details later on.

What about updates?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

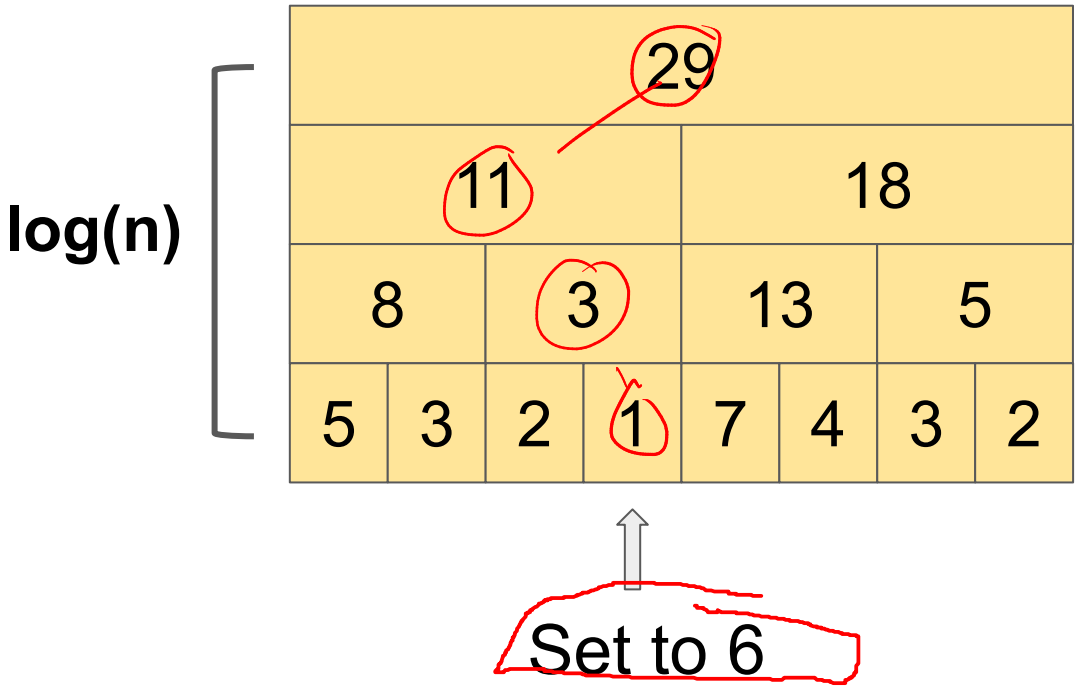
(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y



What about updates?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

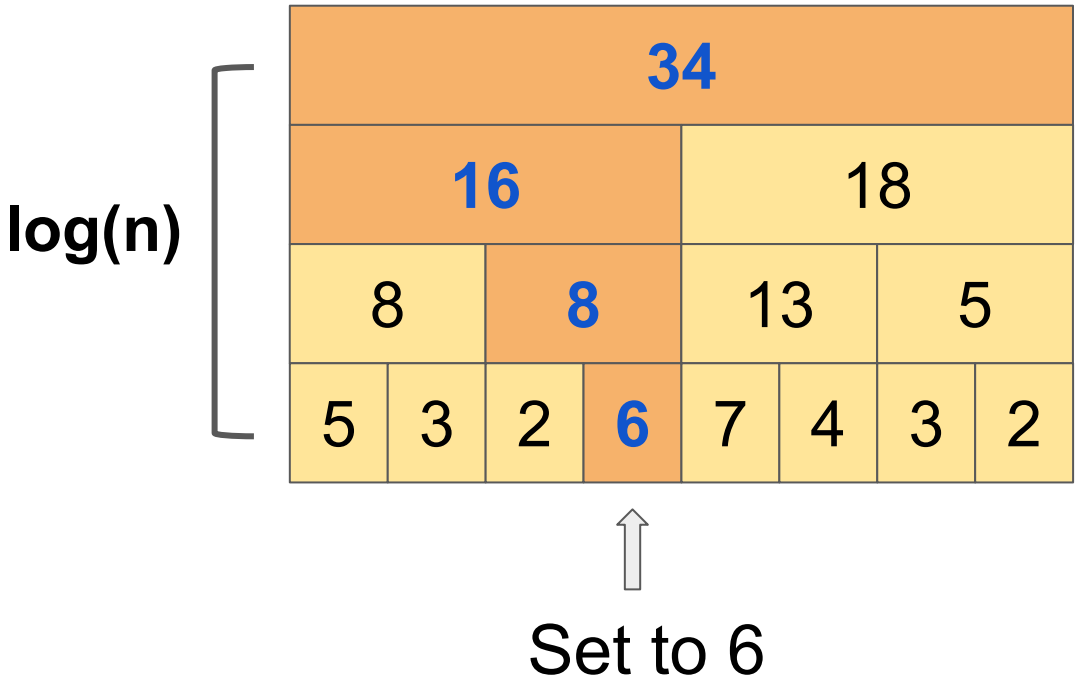
(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y



What about updates?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

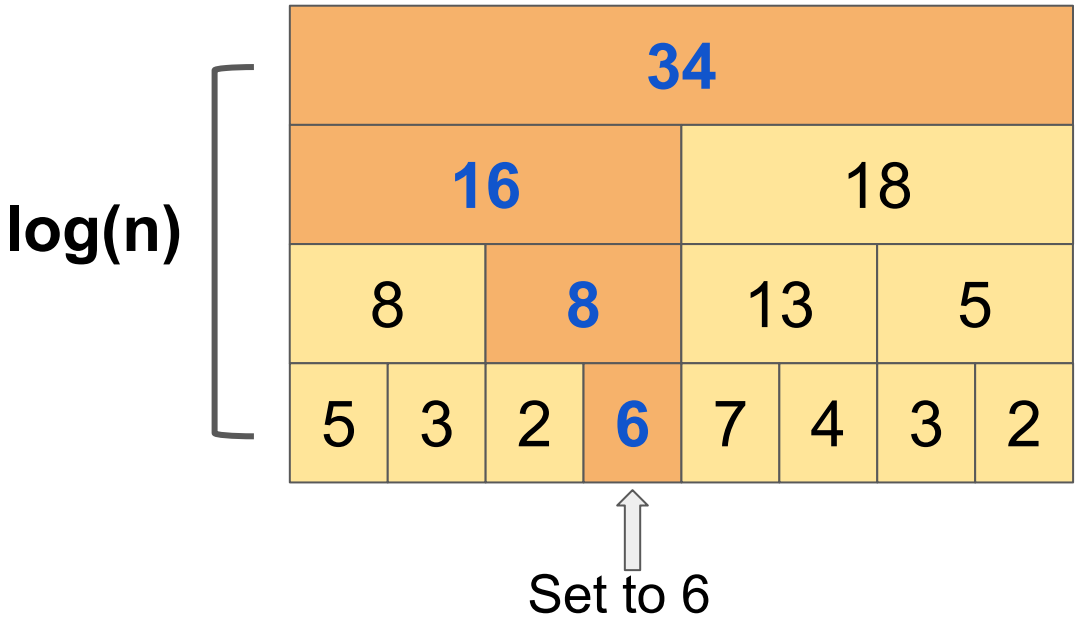
(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y



What about updates?

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

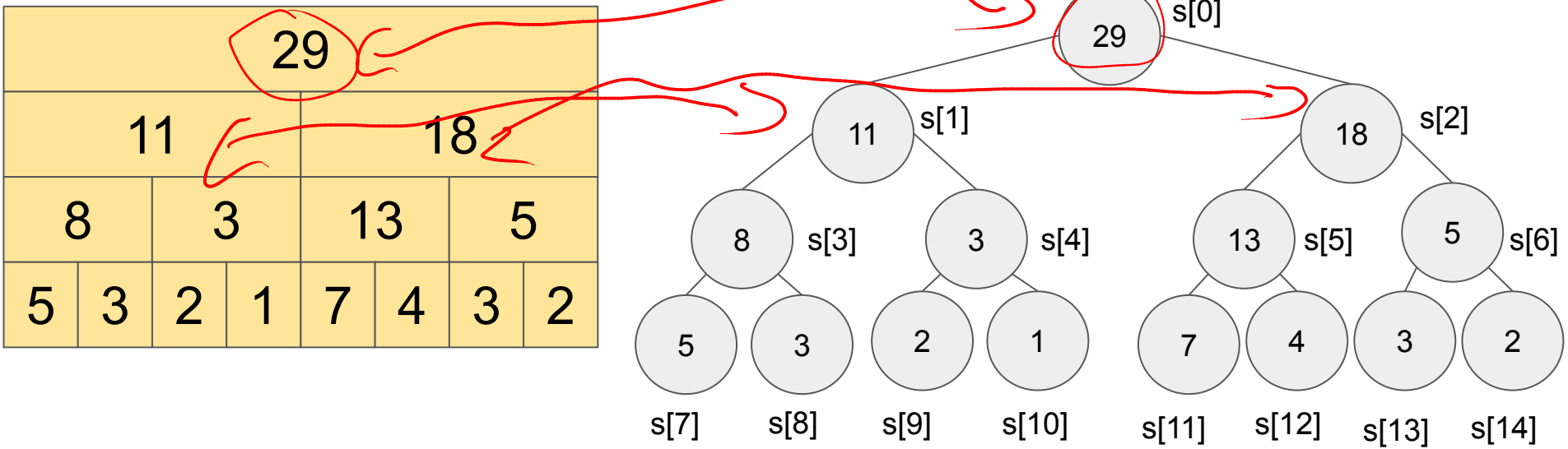
(Query) L R: Return a[L] + a[L+1] + … + a[R]
(Update) X Y: Change the value of a[X] to Y



For any update, only O(logn) nodes change!

Implementation…

Store the segment tree as a binary tree, using heap indexing:
Left child of node n is node 2n + 1
Right child is node 2n+2

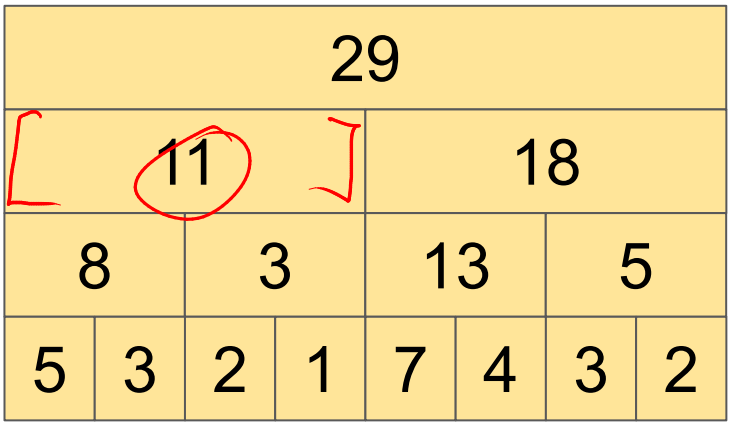


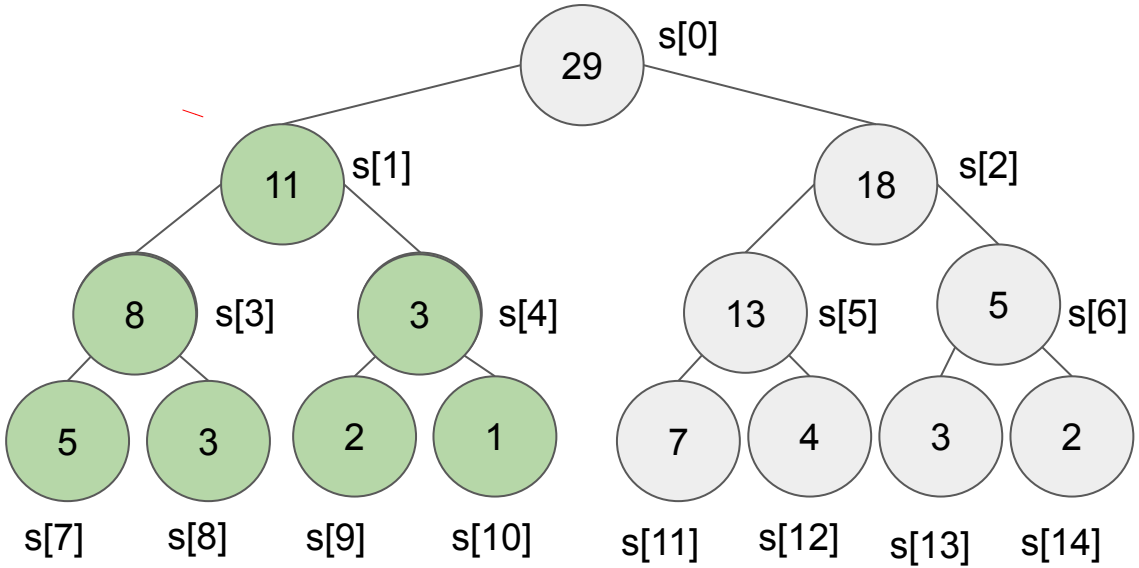
Implementation…

Each node is **“responsible”** for a range of indices that in its subtree

This information **won’t** be stored in these nodes in implementation: but rather passed through the recursive function calls.

Also don’t worry about when it’s not a perfect power of 2, the implementation later will just handle that.



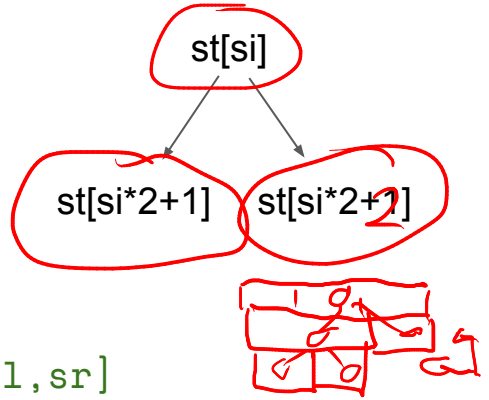


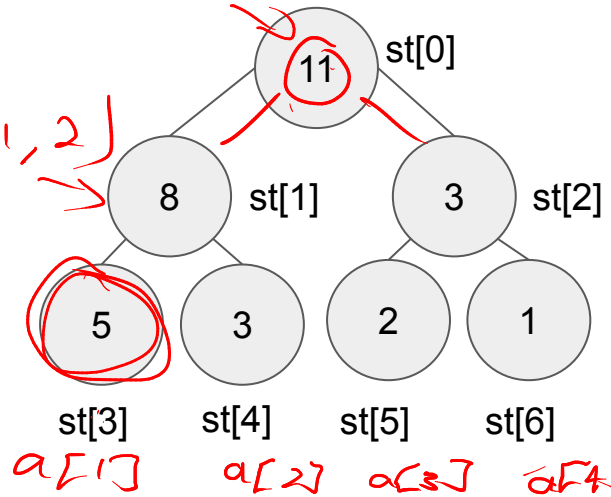
Implementation: Initialising the segtree!

```
// This function, given an array a,
// builds a range sum segtree in the array st
void build(int si, int sl, int sr){
    // si = current segtree left
    // sl = segtree left
    // sr = segtree right
    // Precondition: node si is responsible for indices [sl,sr]

    // Base case: we're at a leaf node.
    if (sl == sr){
        st[si] = a[sl];
        return;
    }
    int mid = (sl + sr)/2;
    // Initialise the children!
    build(si * 2 + 1, sl, mid);
    build(si * 2 + 2, mid+1,sr);
    st[si] = st[si*2+1] + st[si*2+2]
}
```

By the end of this function, st[si] will be the sum of a[sl:sr]

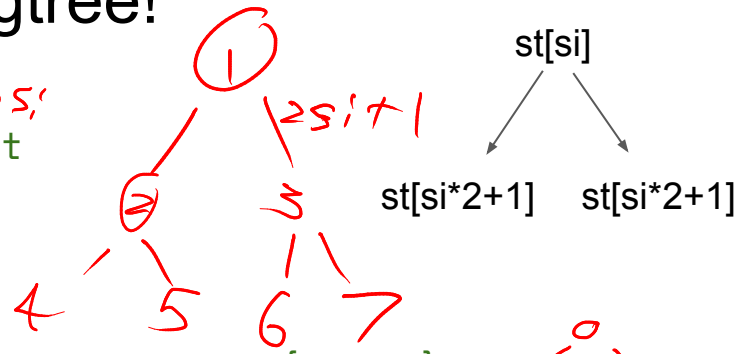


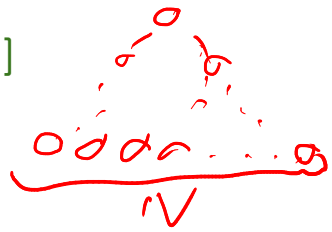


Implementation: Initialising the segtree!

```
// This function, given an array a,
// builds a range sum segtree in the array st
void build(int si, int sl, int sr){
    // si = current segtree left
    // sl = segtree left
    // sr = segtree right
    // Precondition: node si is responsible for indices [sl,sr]

    // Base case: we're at a leaf node.
    if (sl == sr){
        st[si] = a[sl];
        return;
    }
    int mid = (sl + sr)/2;
    // Initialise the children!
    build(si * 2 + 1, sl, mid);
    build(si * 2 + 2, mid+1,sr);
    st[si] = st[si*2+1] + st[si*2+2]
}
```





Question: If we're initialising a segtree over an array of size N, what arguments do we put in when we call build()?

build(0, 1, N)

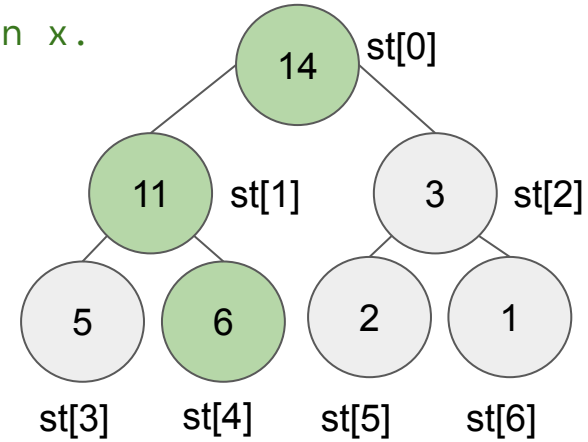
Another question: what is the time complexity?

O(N)

Implementation: Updating!

```
// This function updates a[x] to v
void update(int si, int sl, int sr, int x, int v){
    // Note si,sl,sr have the same meanings as in build
    // Base case: leaf node is simple
    if (sl == sr){
        st[si] = v;
        return;
    }
    int mid = (sl + sr)/2;
    // Update only the child that contains position x.
    if (x <= mid){
        update(si*2+1,sl,mid,x,v);
    }
    else{
        update(si*2+2,mid+1,sr,x,v);
    }
    // Update this node.
    st[si] = st[si*2+1] + st[si*2+2]
}
```

if $sl \le x \le mid$



x = 2, v = 6

Implementation: Updating!

```
// This function updates a[x] to v
void update(int si, int sl, int sr, int x, int v){
    // Note si,sl,sr have the same meanings as in build
    // Base case: leaf node is simple
    if (sl == sr){
        st[si] = v;
        return;
    }
    int mid = (sl + sr)/2;
    // Update only the child that contains position x.
    if (x <= mid){
        update(si*2+1,sl,mid,x,v);
    }
    else{
        update(si*2+2,mid+1,sr,x,v);
    }
    // Update this node.
    st[si] = st[si*2+1] + st[si*2+2]
}
```

(sr - sl) shrinks by half until it reaches 1.

Build (0, 1, N)

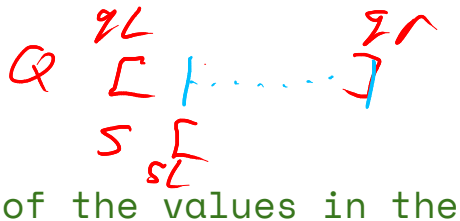
Question: What arguments do we give to update() in order for it to update position x to value y?

update(0, 1, N, x, y)

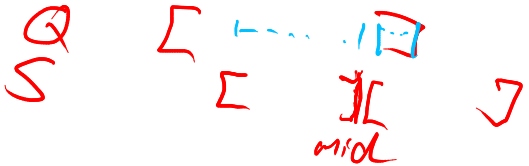
Another question: what is the time complexity of this?

O(Log N)

Implementation: Query!



```
// This function returns the sum of the values in the intersection of
// [sl, sr] and [ql,qr] (Calling query(0,1,n,ql,qr) returns the desired query)
void query(int si, int sl, int sr, int ql, int qr){
    // If [sl,sr] and [ql,qr] don't intersect, the sum is 0.
    if (sl > qr || sr < ql){
        return 0;
    }
    // If [sl,sr] is fully contained in [ql,qr], then the answer is the sum
    from sl to sr (conveniently stored in st[si])
    if (ql <= sl && sr <= qr){
        return st[si];
    }
    // Otherwise, idk, ask our children for the answer.
    int mid = (sl + sr)/2;
    return query(si*2+1,sl,mid,ql,qr) + query(si*2+2,mid+1,sr,ql,qr);
}
```

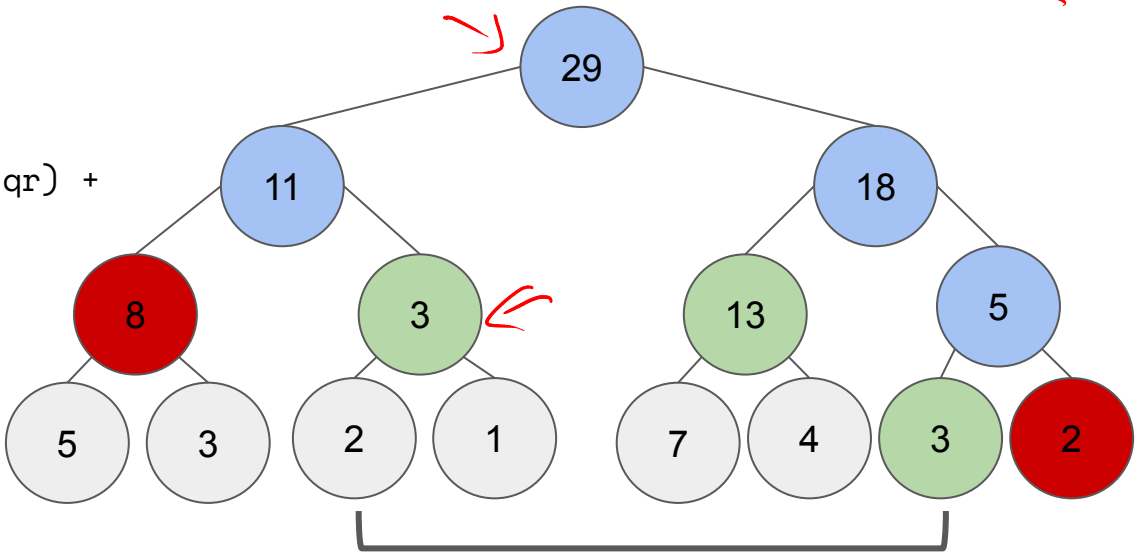


Query time complexity…

```
void query(int si, int sl, int sr, int ql, int qr){
    if (sl > qr || sr < ql){
        // Point A
        return 0;
    }
    if (ql <= sl && sr <= qr){
        // Point B
        return st[si];
    }
    // Point C
    int mid = (sl + sr)/2;
    return query(si*2+1,sl,mid,ql,qr) +
query(si*2+2,mid+1,sr,ql,qr);
}
```

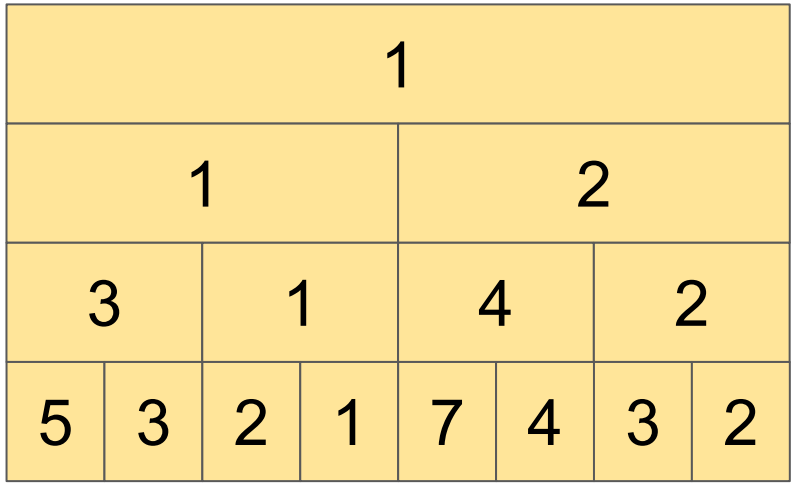
It can be shown that the query will access at **most 4 nodes** at each level in the segtree: at most two greens bookended by blue or red nodes.

Hence at most 4logn nodes are considered, so time complexity is O(logn)



Implementation: Flexibility!!!

What if we want the segtree to solve range minimum query instead? How much do we have to modify the segtree?



Implementation: Flexibility!!!

What if we want the segtree to solve range minimum query instead? How much do we have to modify the segtree?

```
void query(int si, int sl, int sr, int ql, int qr){
	if (sl > qr || sr < ql){
		return 0;
	}
	if (ql <= sl && sr <= qr){
		return st[si];
	}
	int mid = (sl + sr)/2;
	return min(query(si*2+1,sl,mid,ql,qr), query(si*2+2,mid+1,sr,ql,qr));
}
```

```
void update(int si, int sl, int sr, int x, int v){
	if (sl == sr){
		st[si] = v;
		return;
	}
	int mid = (sl + sr)/2;
	if (x <= mid){
		update(si*2+1,sl,mid,x,v);
	}
	else{
		update(si*2+2,mid+1,sr,x,v);
	}
	st[si] = min(st[si*2+1], st[si*2+2]);
}
```

```
void build(int si, int sl, int sr){

	if (sl == sr){
		st[si] = a[sl];
		return;
	}
	int mid = (sl + sr)/2;
	build(si * 2 + 1, sl, mid);
	build(si * 2 + 2, mid+1,sr);
	st[si] = min(st[si*2+1] + st[si*2+2]);
}
```

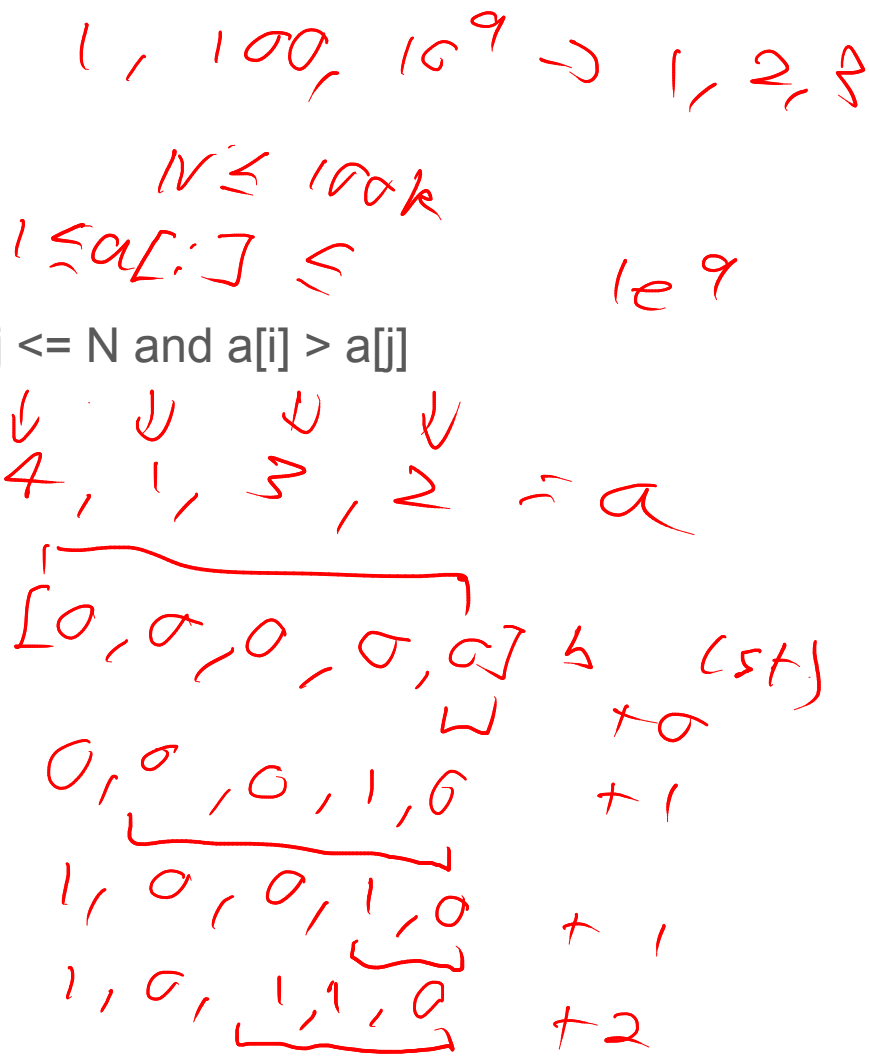
Application: Inversion counting

Given an array of N integers a[1] … a[N], count the number of (i,j) such that 1 <= i < j <= N and a[i] > a[j]

Can you find a solution in O(nlogn)?

Remember: you just learnt what a segtree is! A magical black box that can do arbitrary point updates (change any value in the array) and range queries (find the sum/min/whatever of any (you can even choose the array you build the segtree on)!

b[i] = number of times we have seen the element i.



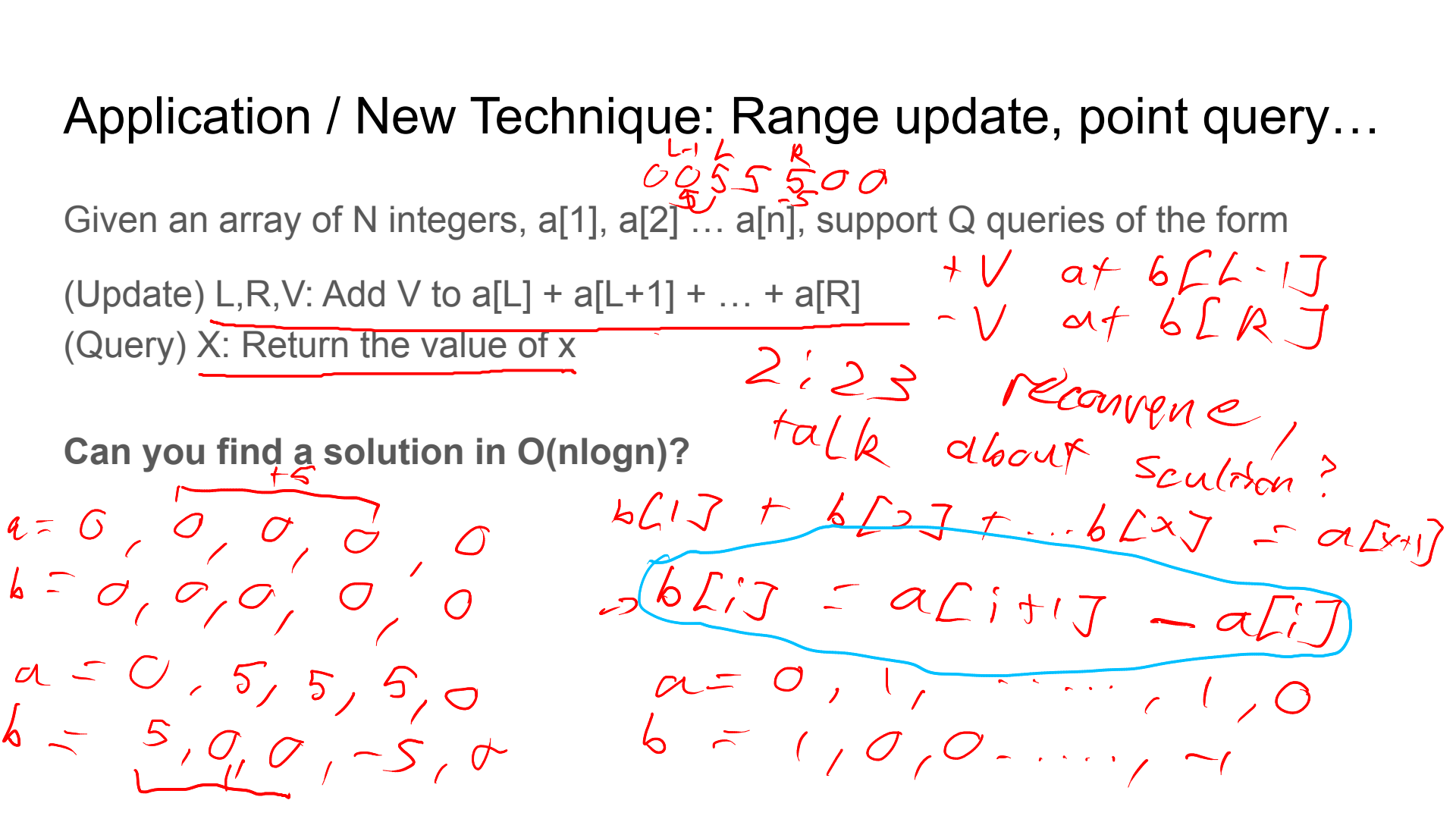
Application / New Technique: Range update, point query…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Update) L,R,V: Add V to a[L] + a[L+1] + … + a[R]

(Query) X: Return the value of x

Can you find a solution in O(nlogn)?

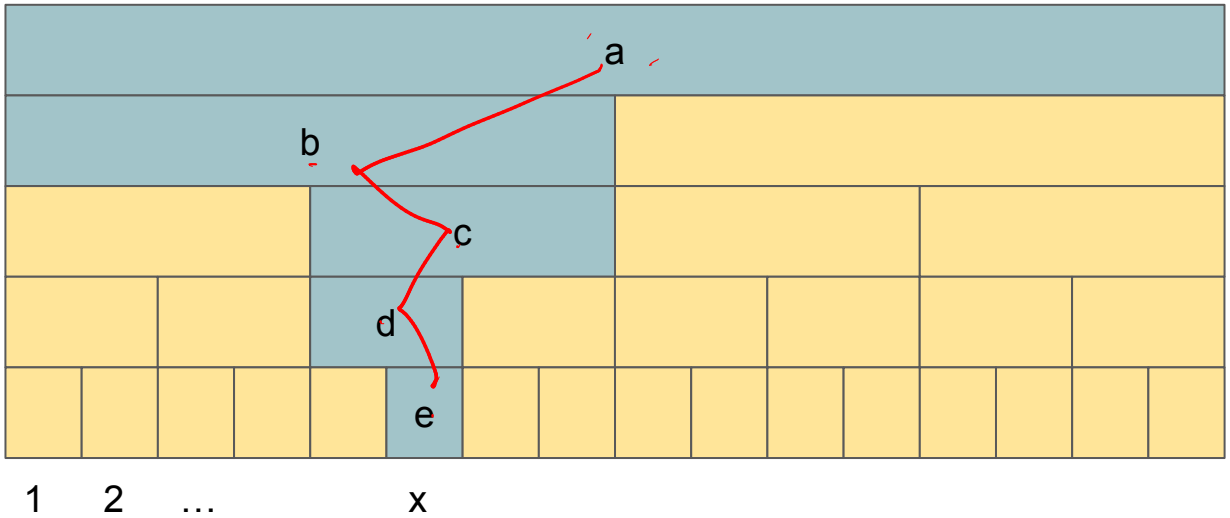


Range update, point query…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Update) L,R,V: Add V to a[L] + a[L+1] + … + a[R]
(Query) X: Return the value of x

Let's just arbitrarily decide, that our query will just walk down the segtree and return the sum of the values it comes across.
Can we design an update function such that this query is correct?



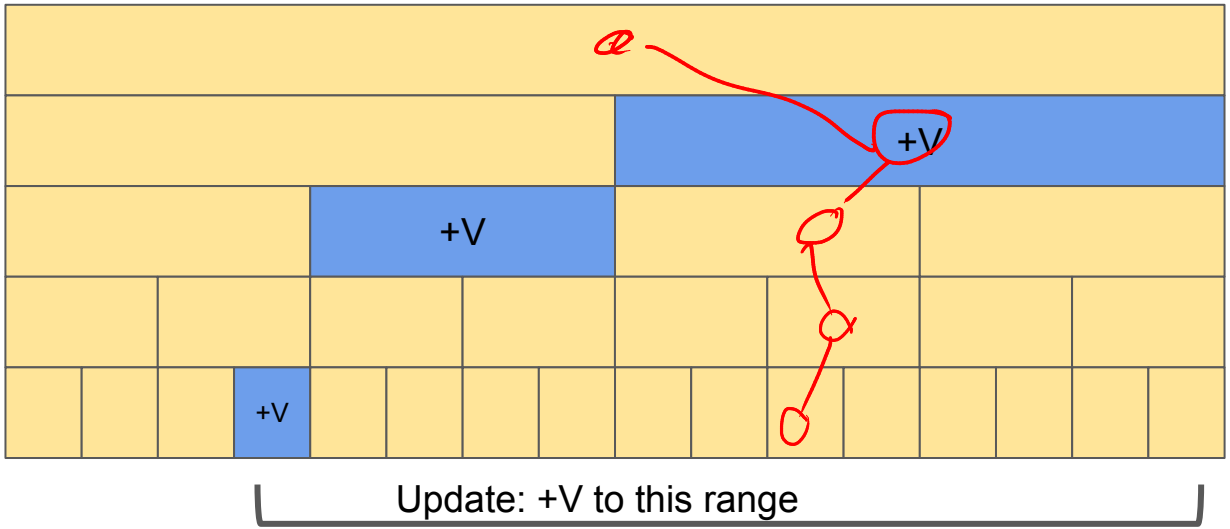
E.g. query(x) return a + b + c + d + e, whatever those values are.

Range update, point query…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Update) L,R,V: Add V to a[L] + a[L+1] + … + a[R]
(Query) X: Return the value of x

Let's just arbitrarily decide, that our query will just walk down the segtree and return the sum of the values it comes across.
Can we design an update function such that this query is correct?



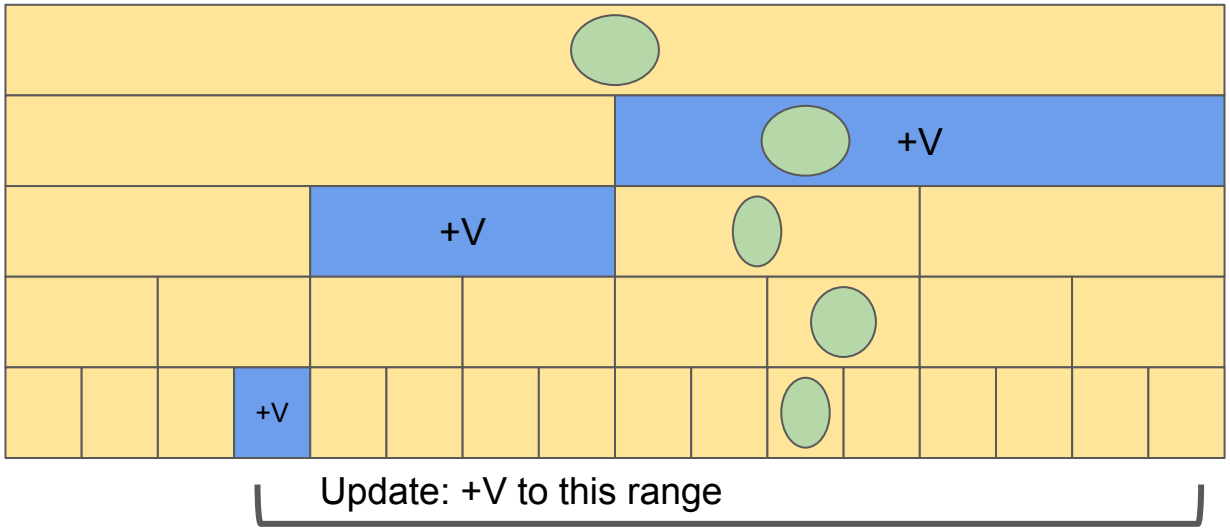
If in our update, we add V to **precisely the blue nodes**, then any point query that passes through the updated range will encounter a single blue node, effectively adding V to the query.

Range update, point query…

Given an array of N integers, a[1], a[2] … a[n], support Q queries of the form

(Update) L,R,V: Add V to a[L] + a[L+1] + … + a[R]
(Query) X: Return the value of x

Let's just arbitrarily decide, that our query will just walk down the segtree and return the sum of the values it comes across.
Can we design an update function such that this query is correct?



Green "circles": example query.

If in our update, we add V to **precisely the blue nodes**, then any point query that passes through the updated range will encounter a single blue node, effectively adding V to the query.

Bonus! Maximum subarray sum, with updates!

Given an array of N (possibly negative) integers a[1] … a[N], there are Q updates of the form:

Change a[i] to V

After each update, calculate the maximum subarray sum of the array.

The maximum subarray sum is the largest value of a[L] + a[L+1] + … + a[R] for any 1 <= L, R <= N

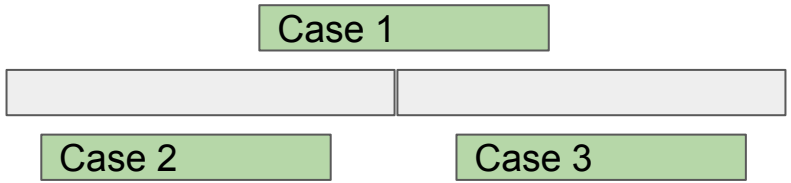
Find an O(nlogn) solution.

Absorb this problem into your head, or screenshot it now :)

Important aside: Weird solution to Maximum Subarray Sum

Suppose you had an array of length N, and you arbitrarily decide to split the array in the middle (N/2). There are 3 possibilities for the maximum subarray sum (MSS):

Case 1: The MSS lies partly on the split.
Case 2: The MSS lies to the left of the split
Case 3: The MSS lies to the right of the split

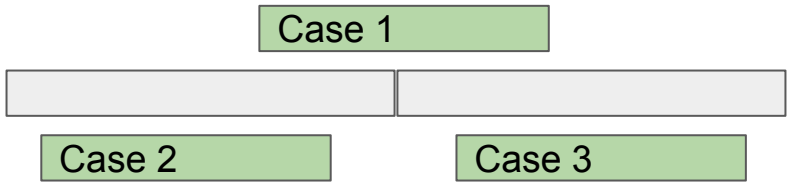


Important aside: Weird solution to Maximum Subarray Sum

Case 1: The MSS lies partly on the split.
Case 2: The MSS lies to the left of the split
Case 3: The MSS lies to the right of the split

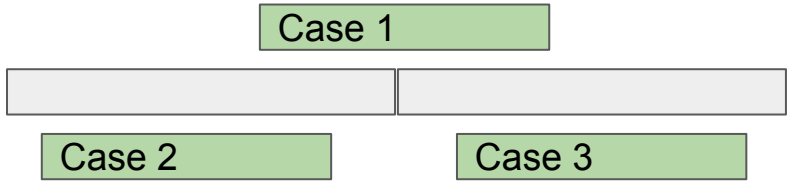
We can solve case 2 and case 3 by recursively solving for the MSS of the left half and the right half respectively.

To solve case 1: simply take the maximum prefix of the right half + the maximum suffix of the left half.



Maximum Subarray Sum continued

(Sorry we ran out of budget to keep preparing these)



Maximum Subarray Sum continued

```
void update(int si, int sl, int sr, int x, int v){
    if (sl == sr){
        st[si] = ???;
        return;
    }
    int mid = (sl + sr)/2;
    if (x <= mid){
        update(si*2+1,sl,mid,x,v);
    }
    else{
        update(si*2+2,mid+1,sr,x,v);
    }
    st[si] = ???
}
```

Maximum Subarray Sum continued

Store invariants:
st[si].total_sum = sum of values from [sl,sr]
st[si].max_prefix = largest possible prefix sum in [sl,sr]
st[si].max_suffix = largest possible suffix sum in [sl,sr]
st[si].answer = max subarray sum in [sl,sr]

// LC = left child = si*2+1
// RC = right child = si*2+2
st[si].total_sum = st[LC].total_sum + st[RC].total_sum
st[si].max_prefix = ???
st[si].max_suffix = ???
st[si].answer = ???