Prefix Sum Array, Sqrt Decomp, Segment Tree

Range Query Problems

You are given an array of integers.
You have to answer queries about consecutive ranges of elements in the array (e.g. sum of a subarray, min of a subarray, etc).
You may have to update the values of the array.
This task may be given to you directly by the problem, or it may arise naturally as part of your solution to a more complicated problem.

Example problem

You are given an array of N integers and Q queries.
Each query consists of two inputs, l and r, and requires you to output the sum of all elements in the array between positions l and r (inclusive).
$1 <= N, Q <= 2e5$
$1 <= l <= r <= N$

Brute Force Implementation (O(NQ))

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

const int MN = 2e5 + 5;
int n, q;
int arr[MN];

int main(){
    cin.sync_with_stdio(0); cin.tie(0);
    cin >> n >> q;
    for(int i = 0; i < n; i++) cin >> arr[i];
    for(int i = 0; i < q; i++){
        int l, r;
        cin >> l >> r;
        ll sum = 0;
        for(int i = l; i <= r; i++) sum += arr[i];
        cout << sum << '\n';
    }
}
```

Prefix Sum Arrays

Prefix Sum Arrays

Problem: you are given an array, and are asked queries of the from: what is the sum of elements with indices in l..r.

Prefix sum array solution:
Construct a new array where each element is equal to the sum of the corresponding prefix of elements in the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

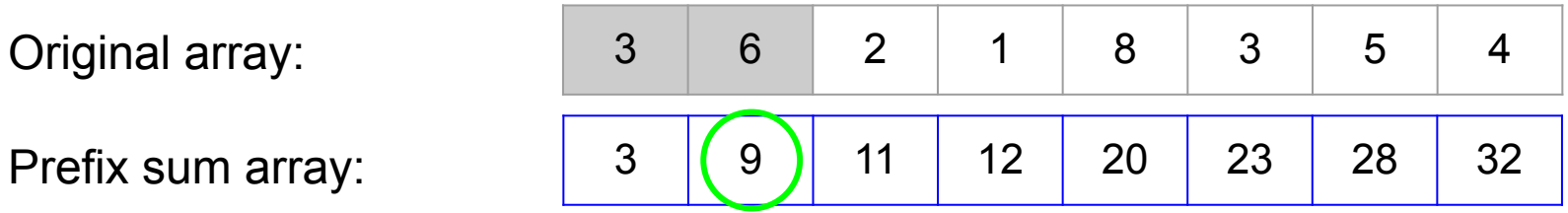
Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.



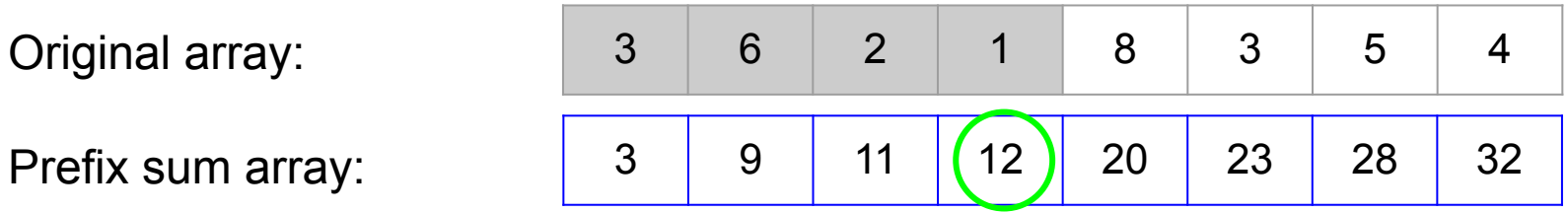
Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.



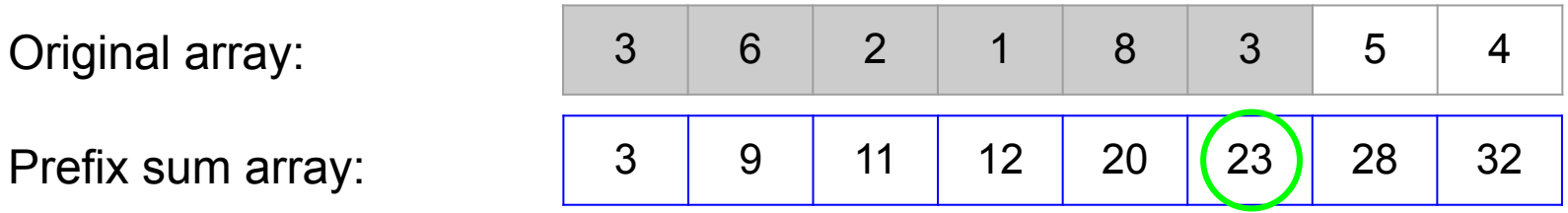
Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

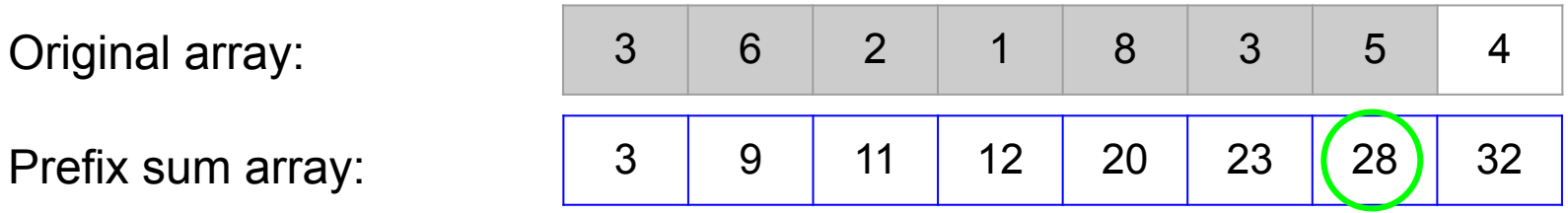
Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.



Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.



Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Queries

The nice thing about prefix sum arrays is that they perform queries really fast. In fact, they can perform them in **O(1)** time. For example, imagine that you are asked to calculate the sum of all elements of the array between index 2 and 6.

Original array:

3	6	2	1	8	3	5	4

Prefix sum array:

3	9	11	12	20	23	28	32

Queries

The nice thing about prefix sum arrays is that they perform queries really fast. In fact, they can perform them in **O(1)** time. For example, imagine that you are asked to calculate the sum of all elements of the array between index 2 and 6.

Original array:

3	6	2	1	8	3	5	4

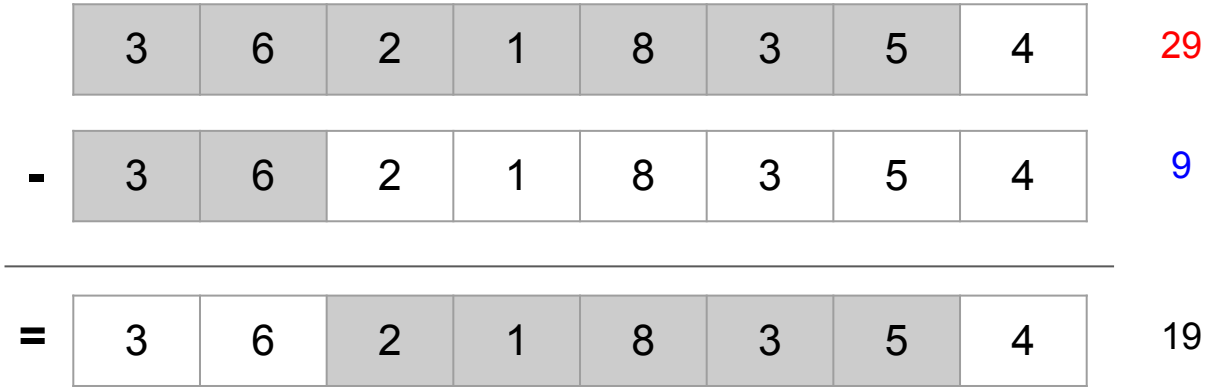
Prefix sum array:

3	9	11	12	20	23	28	32

28 - 9 = 19

2+1+8+3+5 = 19

Why does this work?



Construction

To make it more convenient to do queries, it's often a good idea to 1-index PSAs even if the original array is 0-indexed. Think about what would happen if we tried to query for example 0..5 on a 0-indexed PSA.

To construct every element of the prefix sum array, we take the prefix we calculated for the previous value and add the current element in the array to it.

Original array:

3	6	2	1	8	3	5	4

Prefix sum array:

0	3	9	11	12	20	23	28	32

Implementation

C++

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

const int MN = 2e5 + 5;
int n, q;
int arr[MN];
ll pref[MN];

int main(){
    cin.sync_with_stdio(0); cin.tie(0);
    cin >> n >> q;
    for(int i = 0; i < n; i++) cin >> arr[i];
    for(int i = 0; i < n; i++) pref[i+1] = pref[i] + arr[i];
    for(int i = 0; i < q; i++){
        int l, r;
        cin >> l >> r;
        cout << pref[r] - pref[l-1] << '\n';
    }
}
```

Implementation

Python

```python
n, q = map(int, input().split())
arr = list(map(int, input().split()))

pref = [0]*(n+1)
for i in range(n):
    pref[i+1] = pref[i] + arr[i]

for i in range(q):
    l, r = map(int, input().split())
    print(pref[r] - pref[l-1])
```

Advantages and disadvantages of prefix sum arrays

Good:

- Very fast for what it does, good constant and O(1) queries
- Very easy to implement
- Concept is very easy, some barely consider it a data structure

Bad:

- Doesn’t support updates
- For querying arbitrary intervals, it only supports operations that are invertible, such as sum or xor, but not min or max.

Example of problem PSA can’t solve

Queries require you to find the maximum value of a specified subarray.
Between queries, there are also updates of the form: set the element at position p to value x.

For reference, the brute force solution to this would be to simply maintain the array, perform each update in O(1), and each query in O(N).

Prefix sum arrays obviously do not work because we can’t subtract two max values from each other and updating a PSA would be O(N) anyways.

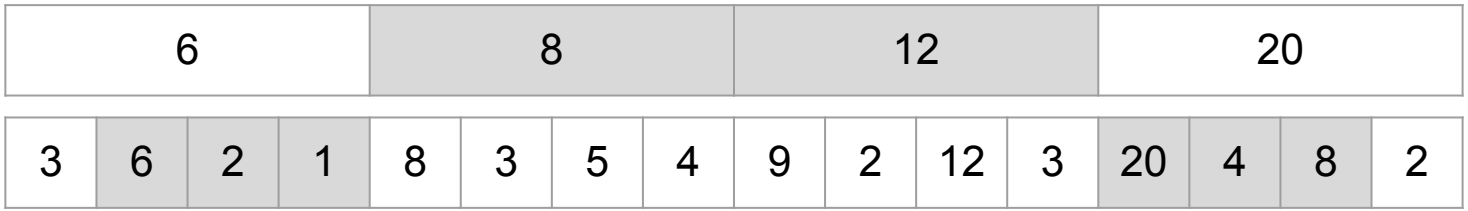
New approach

Split the array into blocks of size B.
For each block, store the maximum value in the interval the block corresponds to.

6				8				12				20			
3	6	2	1	8	3	5	4	9	2	12	3	20	4	8	2

New approach

Split the array into blocks of size B.
For each block, store the maximum value in the interval the block corresponds to.



This should make queries more efficient because for any blocks fully contained in the interval we are querying, we don’t have to iterate through every single element, but can look at the value of the block, which is already computed. We will look at values of blocks at most N/B times. We will look at values of array elements directly at most 2B times. This is N/B + 2B operations in total.

Updates will take B operations since we need to recalculate the value of the block that we updated.

If we set B = sqrt(N),

updates are N/sqrt(N) + 2sqrt(N) = 3sqrt(N) = O(sqrt(N)).

Queries are sqrt(N) = O(sqrt(N))

Sqrt decomp implementation

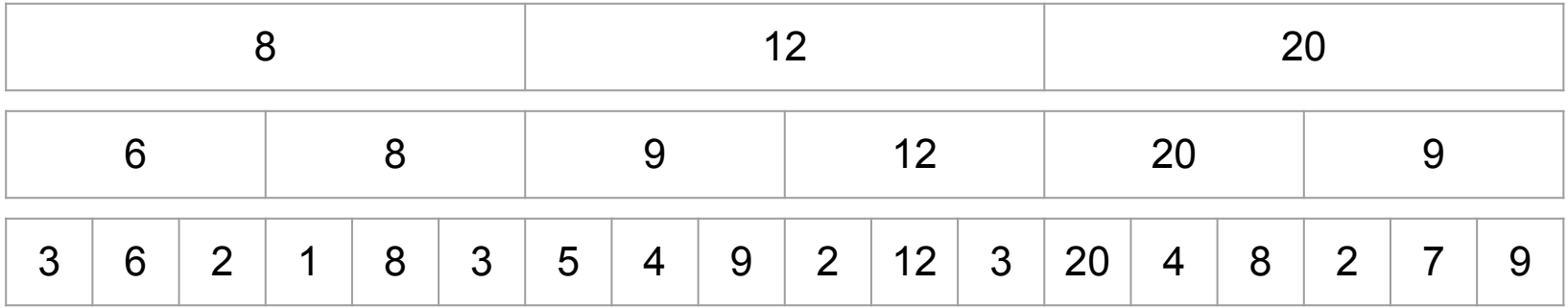
```
const int MN = 2e5 + 5;
const int BSZ = 600;
const int BC = 350;
int n, q;
int arr[MN];
int blocks[BC];

// set value at position p to v
void upd(int p, int v){
    arr[p] = v;
    int block = i/BSZ;
    int newVal = 0;
    for(int i = block*BSZ; i < (block+1)*BSZ; i++) newVal = max(newVal, arr[i]);
    blocks[block] = newVal;
}

// max of [l, r)
int query(int l, int r){
    int res = 0;
    int lBlock = (b+BSZ-1)/BSZ;
    int rBlock = r/BSZ;
    for(int i = lBlock; i < rBlock; i++) res = max(res, blocks[i]);
    for(int i = l; i < lBlock*BSZ; i++) res = max(res, arr[i]);
    for(int i = rBlock*BSZ; i < r; i++) res = max(res, arr[i]);
    return res;
}
```

What if we did this multiple times?

The subproblem of querying the blocks is equivalent to the original problem.
So why can’t we use this strategy to merge that into blocks as well?
We can.



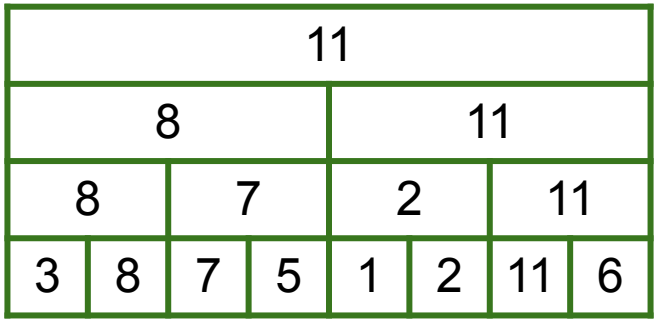
Segment Trees

The idea

Once you realize that you can apply this splitting into blocks strategy multiple times, the next logical question is to ask what rule should we use to determine the size of the blocks.

Since 2 is the smallest number greater than 1 and working with 2 is usually convenient, a natural thought would be to build the tree such that each block covers 2 blocks from the lower layer.

Data structure:



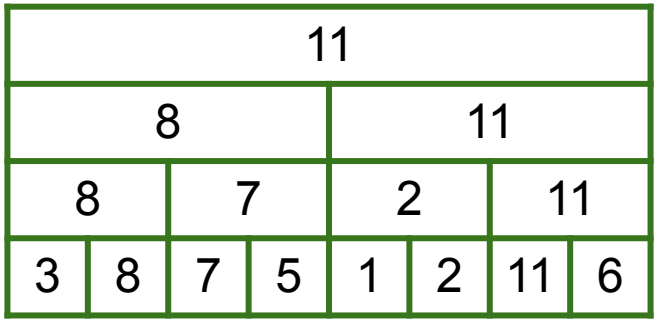
Original array:

3	8	7	5	1	2	11	6

The idea

This looks quite promising. The data structure has a total of $\log_2(N)$ layers.

Data structure:



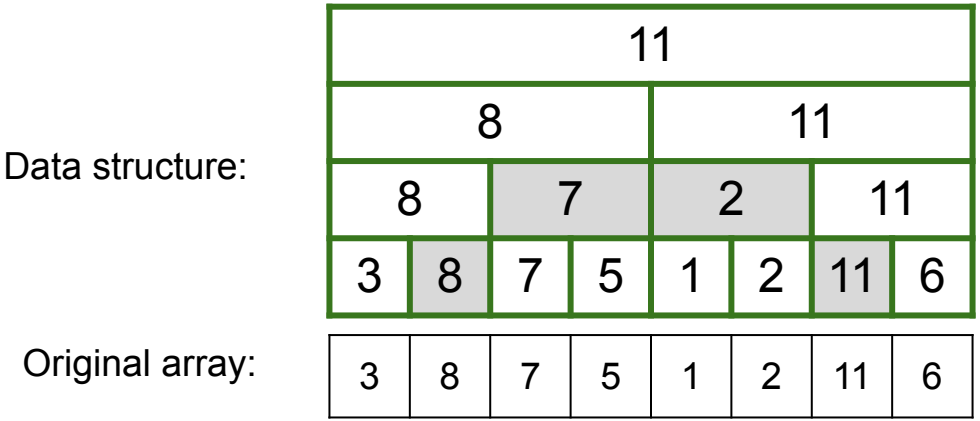
Original array:

3	8	7	5	1	2	11	6

The idea

This looks quite promising. The data structure has a total of $\log_2(N)$ layers.

Each query requires us to look at the values of at most $2\log_2(N)$ nodes (proof later).



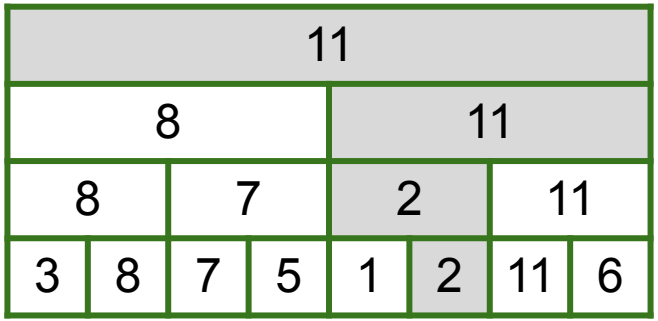
The idea

This looks quite promising. The data structure has a total of $\log_2(N)$ layers.

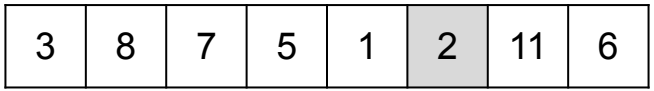
Each query requires us to look at the values of at most $2\log_2(N)$ nodes (proof later).

Each update only affects $\log_2(N)$ values in the tree.

Data structure:



Original array:



The idea

This looks quite promising. The data structure has a total of $\log_2(N)$ layers.

Each query requires us to look at the values of at most $2\log_2(N)$ nodes (proof later).

Each update only affects $\log_2(N)$ values in the tree.

So this data structure could work. We just need to figure out the details.

Data structure:

11							
8				11			
8		7		2		11	
3	8	7	5	1	2	11	6

Original array:

3	8	7	5	1	2	11	6

Summary of structure

The size of our input array is a power of 2. If it is not, we can pad it with dummy values. This makes things more convenient for now.

Each value of the data structure corresponds to the max of some subarray.

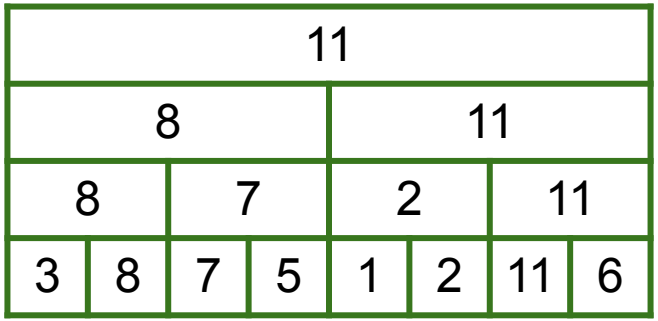
The overall structure is a binary tree; each node has 2 children.

The data structure can be split into layers, with the nodes in each layer corresponding to subarrays of the same length.

Each element in the last layer represents the max value of a subarray of length 1.

We will call this data structure a “segment tree” since it’s a binary tree where each node stores the value of some segment of the original array.

Segment tree:



Original array:

3	8	7	5	1	2	11	6

Summary of structure

The size of our input array is a power of 2. If it is not, we can pad it with dummy values. This makes things more for now.

Each value of the data structure corresponds to the max of some subarray.

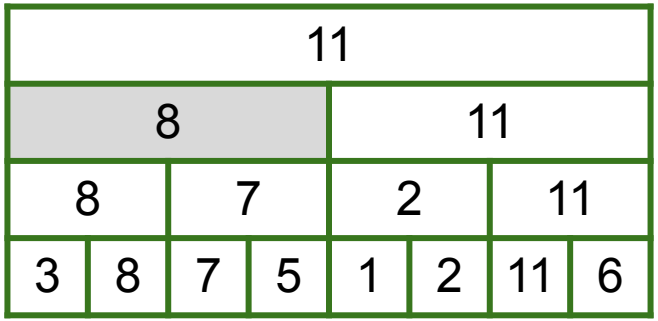
The overall structure is a binary tree; each node has 2 children.

The data structure can be split into layers, with the nodes in each layer corresponding to subarrays of the same length.

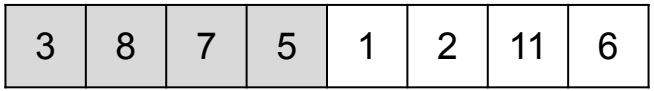
Each element in the last layer represents the max value of a subarray of length 1.

We will call this data structure a “segment tree” since it’s a binary tree where each node stores the value of some segment of the original array.

Segment tree:

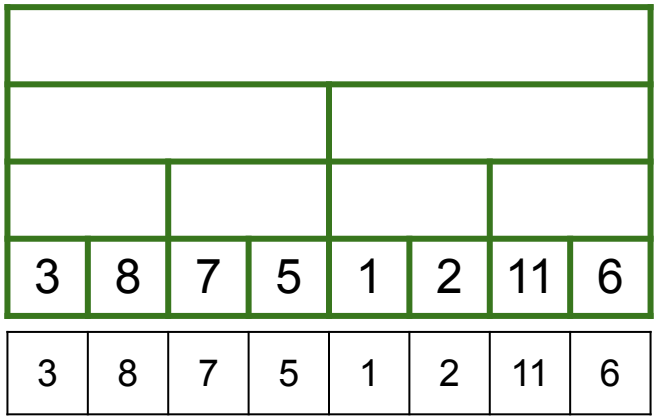


Original array:



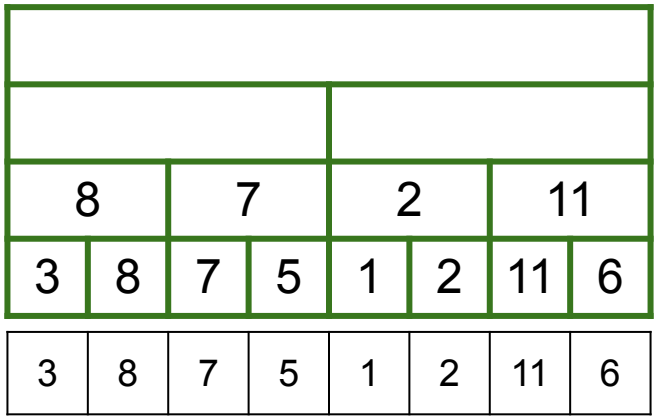
Building a segment tree from the array

The value of the last layer of nodes is equal to the values in the array.
For every other node, we only need to know the values of its 2 children to calculate its value.
We construct the tree starting from the bottom.



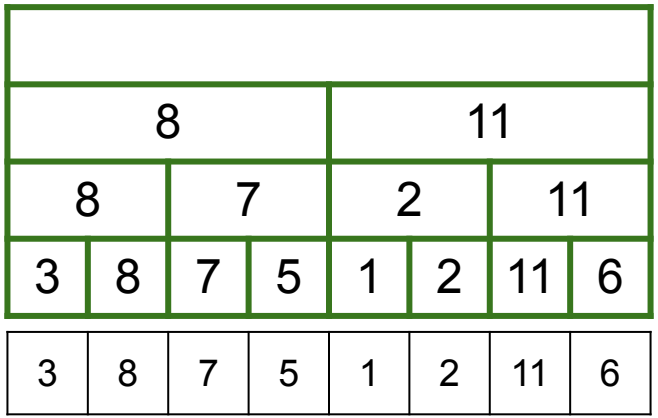
Building a segment tree from the array

The value of the last layer of nodes is equal to the values in the array.
For every other node, we only need to know the values of its 2 children to calculate its value.
We construct the tree starting from the bottom.



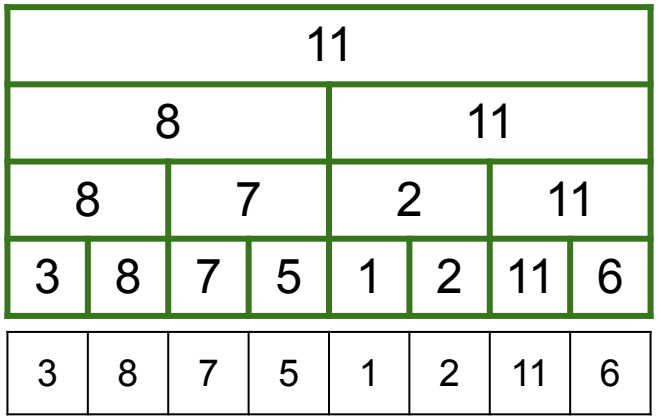
Building a segment tree from the array

The value of the last layer of nodes is equal to the values in the array.
For every other node, we only need to know the values of its 2 children to calculate its value.
We construct the tree starting from the bottom.



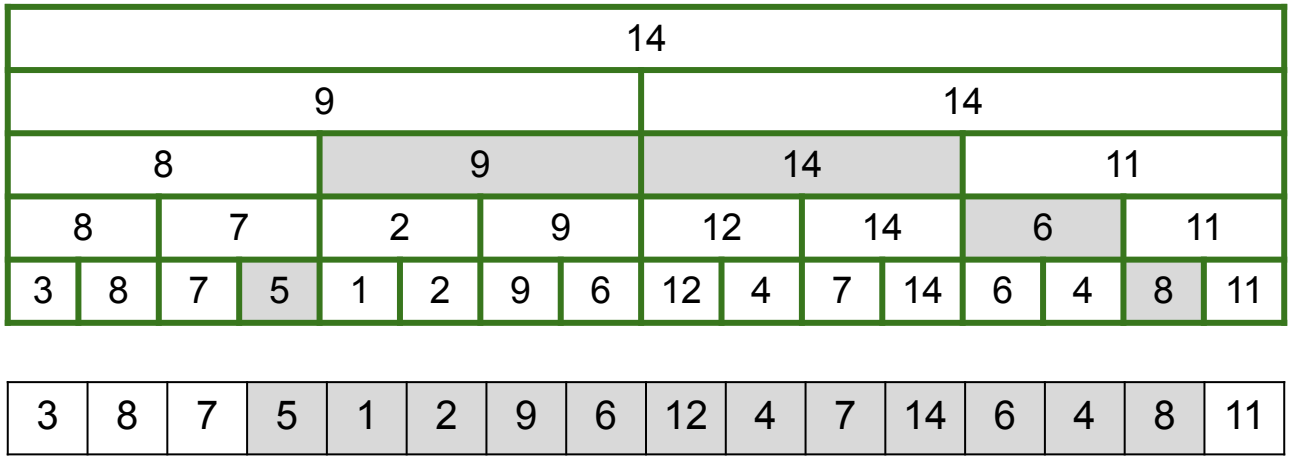
Building a segment tree from the array

The value of the last layer of nodes is equal to the values in the array.
For every other node, we only need to know the values of its 2 children to calculate its value.
We construct the tree starting from the bottom.

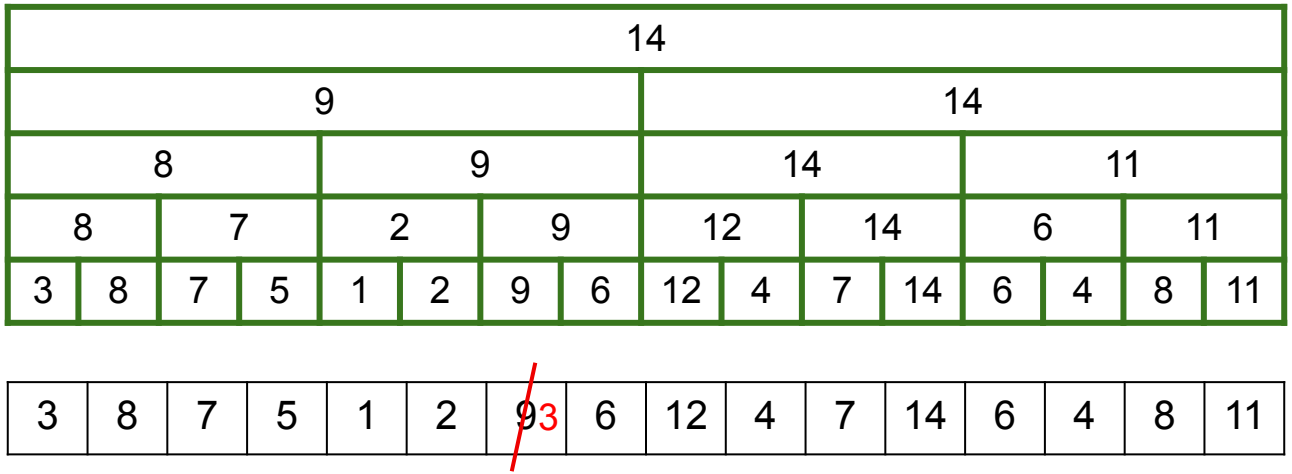


Queries

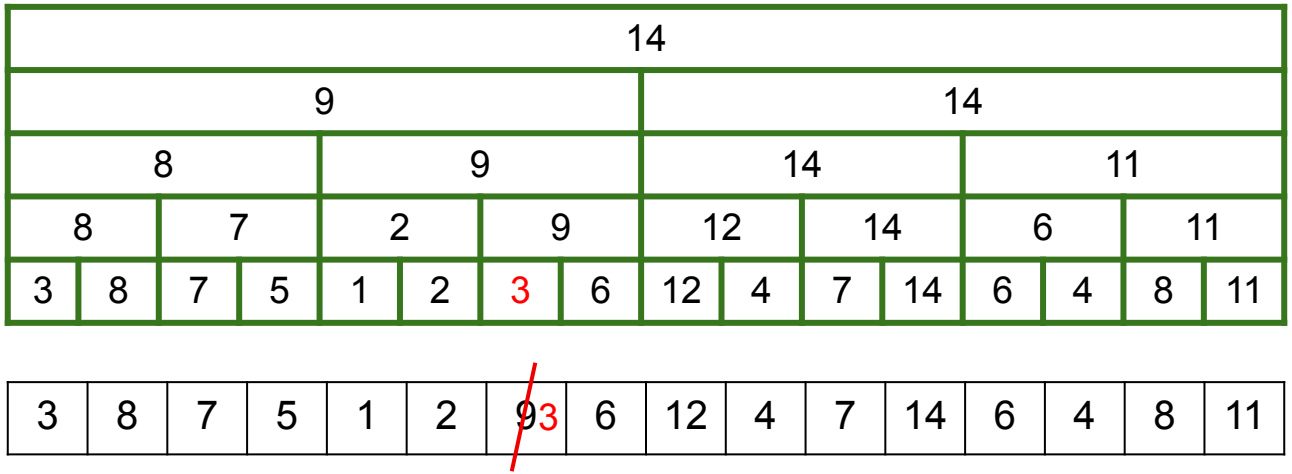
For queries, we will pick the minimum number of segments in the tree that cover the entire subarray, and take the max of those segments. This results in us taking at most 2 segments from each layer, one on the left and one on the right.



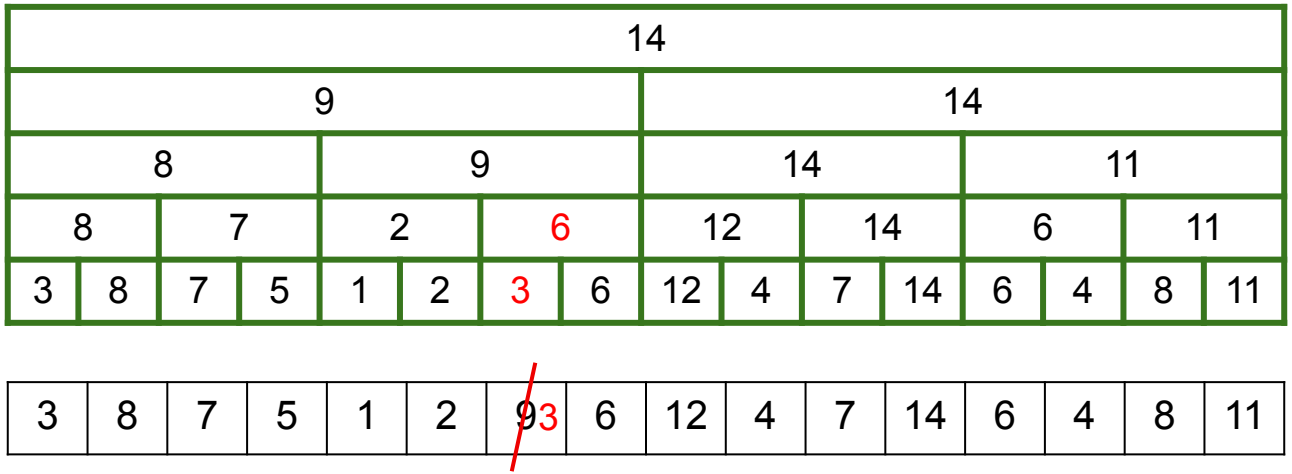
Updates


14
9
14
8
9
14
11
8
7
2
9
12
14
6
11
3
8
7
5
1
2
9
6
12
4
7
14
6
4
8
11
3
8
7
5
1
2
93
6
12
4
7
14
6
4
8
11

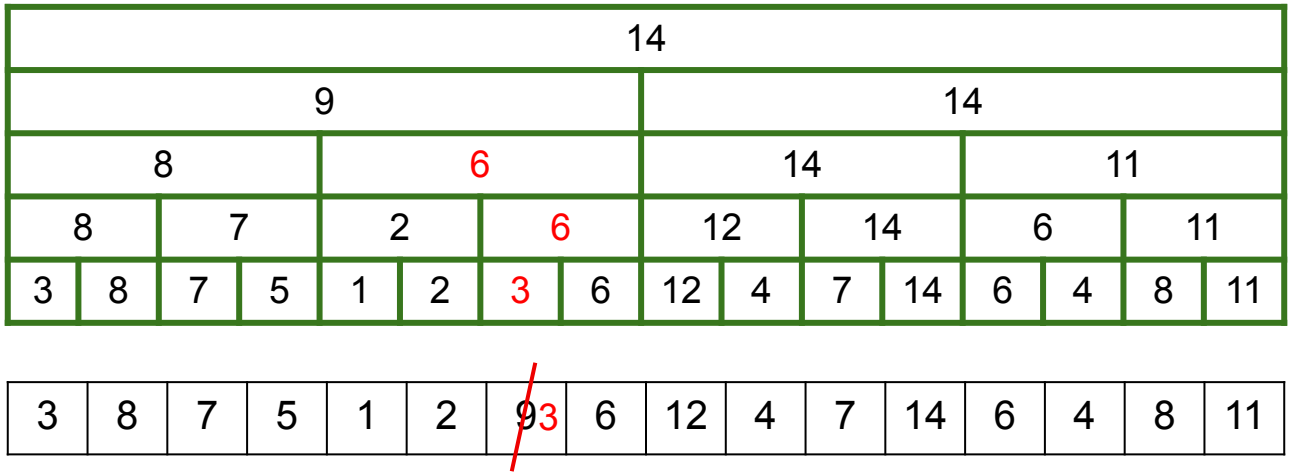
Updates


14
9
14
8
9
14
11
8
7
2
9
12
14
6
11
3
8
7
5
1
2
3
6
12
4
7
14
6
4
8
11
3
8
7
5
1
2
9
3
6
12
4
7
14
6
4
8
11

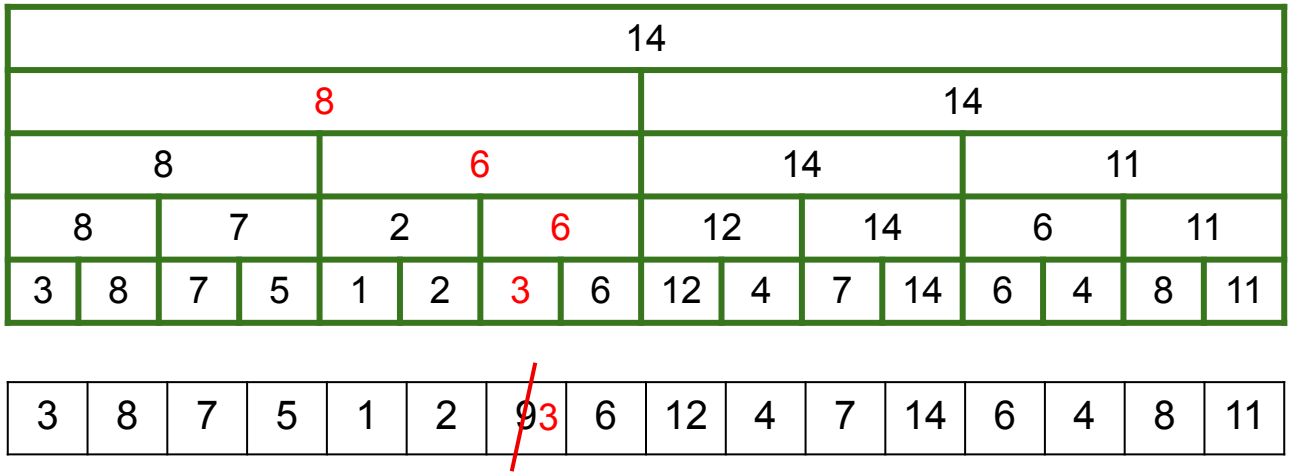
Updates


14
9
14
8
9
14
11
8
7
2
6
12
14
6
11
3
8
7
5
1
2
3
6
12
4
7
14
6
4
8
11
3
8
7
5
1
2
9
3
6
12
4
7
14
6
4
8
11

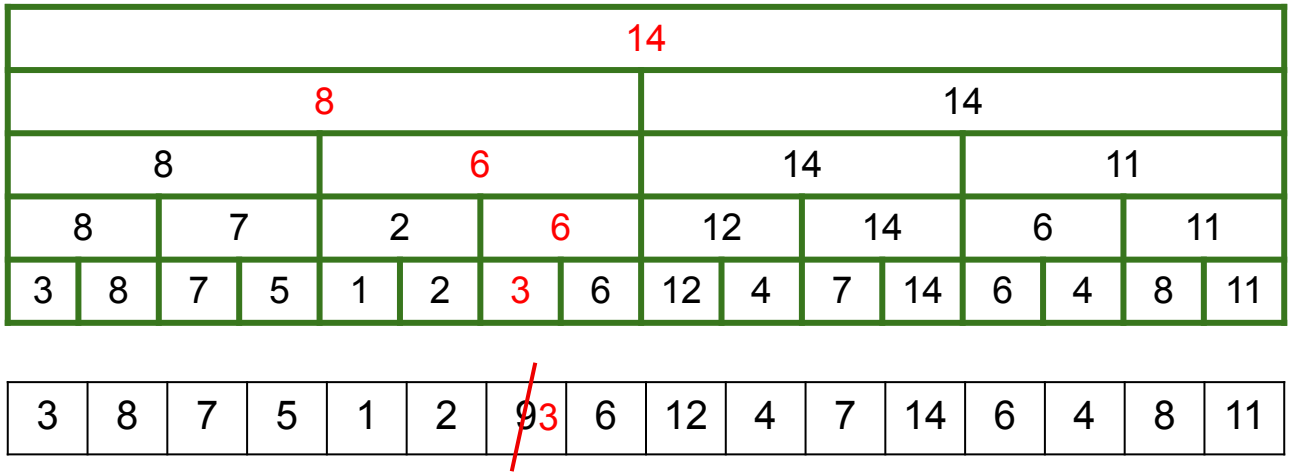
Updates


14
9
14
8
6
14
11
8
7
2
6
12
14
6
11
3
8
7
5
1
2
3
6
12
4
7
14
6
4
8
11
3
8
7
5
1
2
9
3
6
12
4
7
14
6
4
8
11

Updates

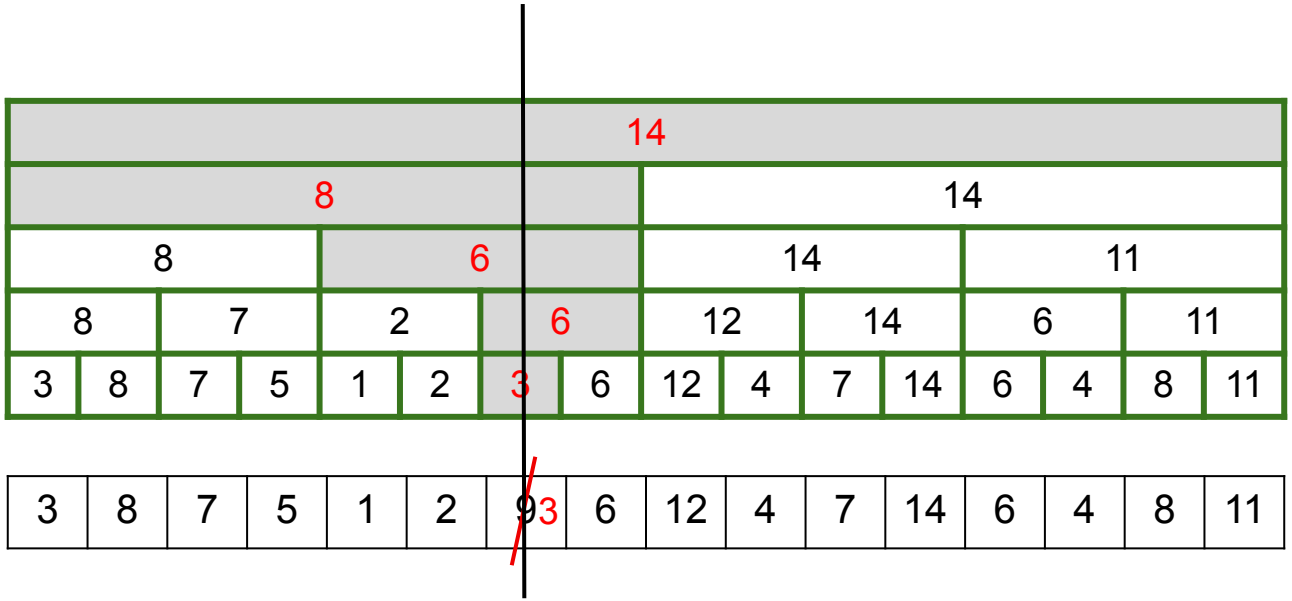

14
8
14
8
6
14
11
8
7
2
6
12
14
6
11
3
8
7
5
1
2
3
6
12
4
7
14
6
4
8
11
3
8
7
5
1
2
9
3
6
12
4
7
14
6
4
8
11

Updates


14
8
14
8
6
14
11
8
7
2
6
12
14
6
11
3
8
7
5
1
2
3
6
12
4
7
14
6
4
8
11
3
8
7
5
1
2
9
3
6
12
4
7
14
6
4
8
11

Updates

At most logN nodes were changed

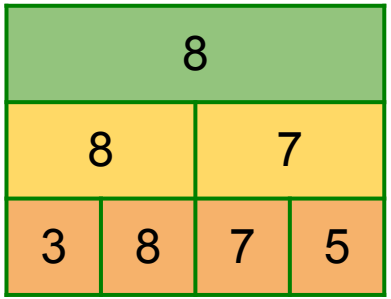


Array representation

Instead of storing this tree as an adjacency list or something, we store it as an array.
The element at position 1 is the root.
Node i has children i*2, i*2+1.
By this definition, node i has parent i/2.
This representation is very convenient and is used by almost all segment tree implementation.
One side effect of this representation is that the nodes are ordered by their layer; the element at index 1 is the root, index 2 and 3 are on second layer, indices 4, 5, 6, 7 are on third layer, indices 8, 9, 10, 11, 12, 13, 14, 15 are on fourth layer etc.

Array Representation

Segment tree



Root

Layer 1

Layer 2

Original array

3	8	7	5

Segment tree

0	8	8	7	3	8	7	5

The root is at index 1.

The children of the segment with index i are the segments with indices i*2 and i*2 + 1

Advantages of segment trees

O(logN) update
O(logN) query
Works for any associative function! (A binary operator * is associative if (a * (b * c)) = ((a * b) * c) for all a, b, c)

Implementation (recursive)

C++

```
void build(){
    for(int i = TSZ-1; i > 0; i--){
        tree[i] = max(tree[i*2], tree[i*2+1]);
    }
}

int query(int i, int segL, int segR, int l, int r){
    if(r <= segL || l >= segR) return 0;
    if(l <= segL && r >= segR) return tree[i];
    int mid = (segL+segR)/2;
    return max(query(i*2, segL, mid, l, r), query(i*2+1, mid, segR, l, r));
}

int upd(int i, int segL, int segR, int p, int v){
    if(p < segL || p >= segR) return tree[i];
    if(segR-segL == 1) return tree[i] = v;
    int mid = (segL+segR)/2;
    return tree[i] = max(upd(i*2, segL, mid, p, v), upd(i*2+1, mid, segR, p, v));
}
```

Implementation (recursive)

Python

```python
def build():
    for i in range(tsz-1, 0, -1):
        tree[i] = max(tree[i*2], tree[i*2+1])

def query(i, segL, segR, l, r):
    if r <= segL or l >= segR: return 0
    if l <= segL and r >= segR: return tree[i]
    mid = (segL+segR)//2
    return max(query(i*2, segL, mid, l, r), query(i*2+1, mid, segR, l, r))

def upd(i, segL, segR, p, v):
    if p < segL or p >= segR: return tree[i]
    if segR-segL == 1:
        tree[i] = v
        return tree[i]
    mid = (segL+segR)//2
    tree[i] = max(upd(i*2, segL, mid, p, v), upd(i*2+1, mid, segR, p, v))
    return tree[i]
```

Implementation (iterative, easy to read)

```
const int TSZ = 1 << 18;
int tree[TSZ*2];

void build(){
    for(int i = TSZ-1; i > 0; i--){
        tree[i] = max(tree[i*2], tree[i*2+1]);
    }
}

void upd(int i, int v){
    i += TSZ;
    tree[i] = v;
    while(i > 1){
        i /= 2;
        tree[i] = max(tree[i*2], tree[i*2+1]);
    }
}
```

```
// query [l, r)
int query(int l, int r){
    int res = 0;
    l += TSZ;
    r += TSZ;
    while(l < r){
        if(l % 2 == 1){
            res = max(res, tree[l]);
            l++;
        }
        if(r % 2 == 1){
            r--;
            res = max(res, tree[r]);
        }
        l /= 2;
        r /= 2;
    }
    return res;
}
```

Implementation (iterative, short)

```
const int TSZ = 1 << 18;
int tree[TSZ*2];

void build(){
    for(int i = TSZ-1; i > 0; i--){
        tree[i] = max(tree[i*2], tree[i*2+1]);
    }
}

void upd(int i, int v){
    for(tree[i += TSZ] = v; i >>= 1;){
        tree[i] = max(tree[i*2], tree[i*2+1]);
    }
}

// query [l, r)
int query(int l, int r){
    int res = 0;
    for(l += TSZ, r += TSZ; l < r; l >>= 1, r >>= 1){
        if(l & 1) res = max(res, tree[l++]);
        if(r & 1) res = max(res, tree[--r]);
    }
    return res;
}
```

Implementation (iterative, any associative func, generic template)

```cpp
template<typename T, int MSZ>
struct SegTree {
    T data[MSZ*2];
    static constexpr T unit = 0;

    inline T merge(T t1, T t2){
        return max(t1, t2);
    }

    void build(){
        for (int i = MSZ-1; i > 0; --i){
            data[i] = merge(data[i*2], data[i*2+1]);
        }
    }

    // Set value at position p to v
    void update(int p, T v){
        for (data[p += MSZ] = v; p >>= 1;){
            data[p] = merge(data[p*2], data[p*2+1]);
        }
    }

    // Query range [l, r)
    T query(int l, int r){
        T resL = unit, resR = unit;
        for (l += MSZ, r += MSZ; l < r; l >>= 1, r >>= 1) {
            if(l&1) resL = merge(resL, data[l++]);
            if(r&1) resR = merge(data[--r], resR);
        }
        return merge(resL, resR);
    }

    // Access the last layer of the segment tree
    T& operator[](int i){
        return data[i + MSZ];
    }
};
```

Implementation (lazy seg tree)

```
// queries: get max element in range
// updates: add v to each element in a range

const int TSZ = 1 << 18;
int tree[TSZ*2], lazy[TSZ*2];

void build(){
    for(int i = TSZ-1; i > 0; i--){
        tree[i] = max(tree[i*2], tree[i*2+1]);
    }
}

void pushDown(int i){
    tree[i*2] += lazy[i];
    tree[i*2+1] += lazy[i];
    lazy[i*2] += lazy[i];
    lazy[i*2+1] += lazy[i];
    lazy[i] = 0;
}
```

```
int query(int i, int segL, int segR, int l, int r){
    if(r <= segL || l >= segR) return 0;
    if(l <= segL && r >= segR) return tree[i];
    int mid = (segL+segR)/2;
    pushDown(mid);
    return max(query(i*2, segL, mid, l, r), query(i*2+1, mid, segR, l, r));
}

int upd(int i, int segL, int segR, int l, int r, int v){
    if(r <= segL || l >= segR) return tree[i];
    if(l <= segL && r >= segR){
        lazy[i] += v;
        return tree[i] += v;
    }
    int mid = (segL+segR)/2;
    pushDown(mid);
    return tree[i] = max(upd(i*2, segL, mid, l, r, v), upd(i*2+1, mid, segR, l, r, v));
}
```

Example problems

https://dmoj.ca/problem/dmopc14c2p4 (template PSA)
https://dmoj.ca/problem/bts18p2 (PSAs)

https://dmoj.ca/problem/segtree (template segtree)
https://dmoj.ca/problem/ds3 (segtree)
https://dmoj.ca/problem/dmpg17g2 (segtree)
https://dmoj.ca/problem/bsfast (binary search on seg tree)

https://dmoj.ca/problem/lazy (template lazy segtree)
https://dmoj.ca/problem/dmopc15c1p6 (basically template lazy segtree)

Most of the segtree probs can probably also be solved with sqrt.