Structs

No python.

Unfortunately, Python code will appear less frequently in this lecture. There are two main reasons for this:

1. The speed of further algorithms' execution.

2. The absence of many things in Python.

But you can always ask me for help with understanding C++ code in direct messages.

RSQ

You are given an array of n natural numbers.

1. You need to be able to find the sum in a segment.
2. You need to be able to execute a query a[i] += x, which means adding x to the element.
3. You need to be able to add in a segment, that is a[l..r] += x.

What we can?

1.pi_sum

2.prefix add

But what is next?

SQRT decomposition

Let’s divide array to blocks.

a[0..(c - 1)] - first block

a[c..(2c - 1)] - second

a[2c..(3c - 1)] - third

…

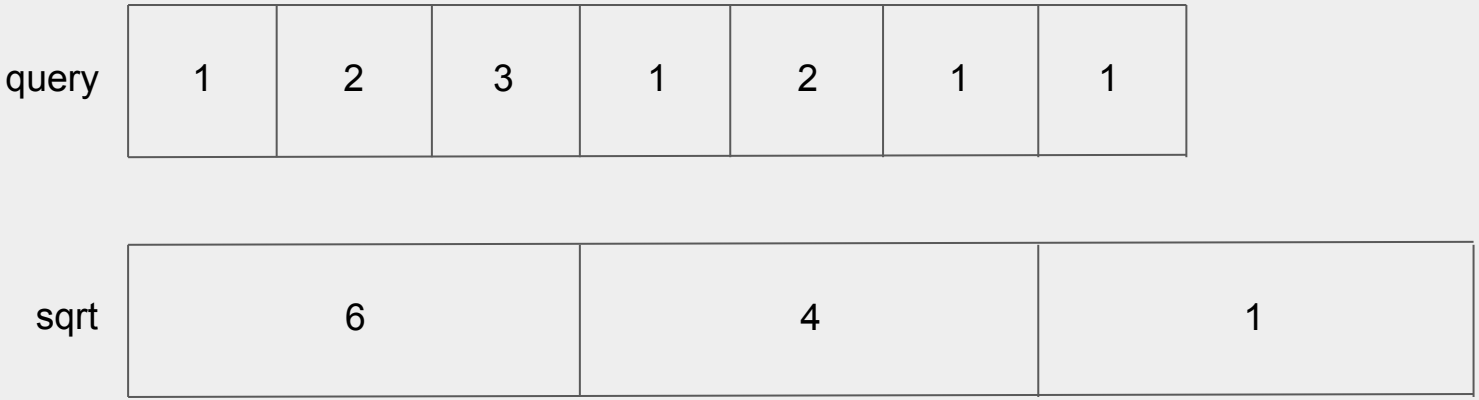
a[(n // c) * c..(n // c + 1) * c - 1] - last

SQRT decomposition

Let's calculate the sum in each of the blocks.

```
28.
29.  int main() {
30.      int n;
31.      cin >> n;
32.      vector<int> a(n);
33.      vector<Block> sqrt_dec(n / c + 1);
34.      for (int i = 0; i < n; i++) {
35.          cin >> a[i];
36.          sqrt_dec[i / c].sum += a[i];
37.      }
```

c = 3

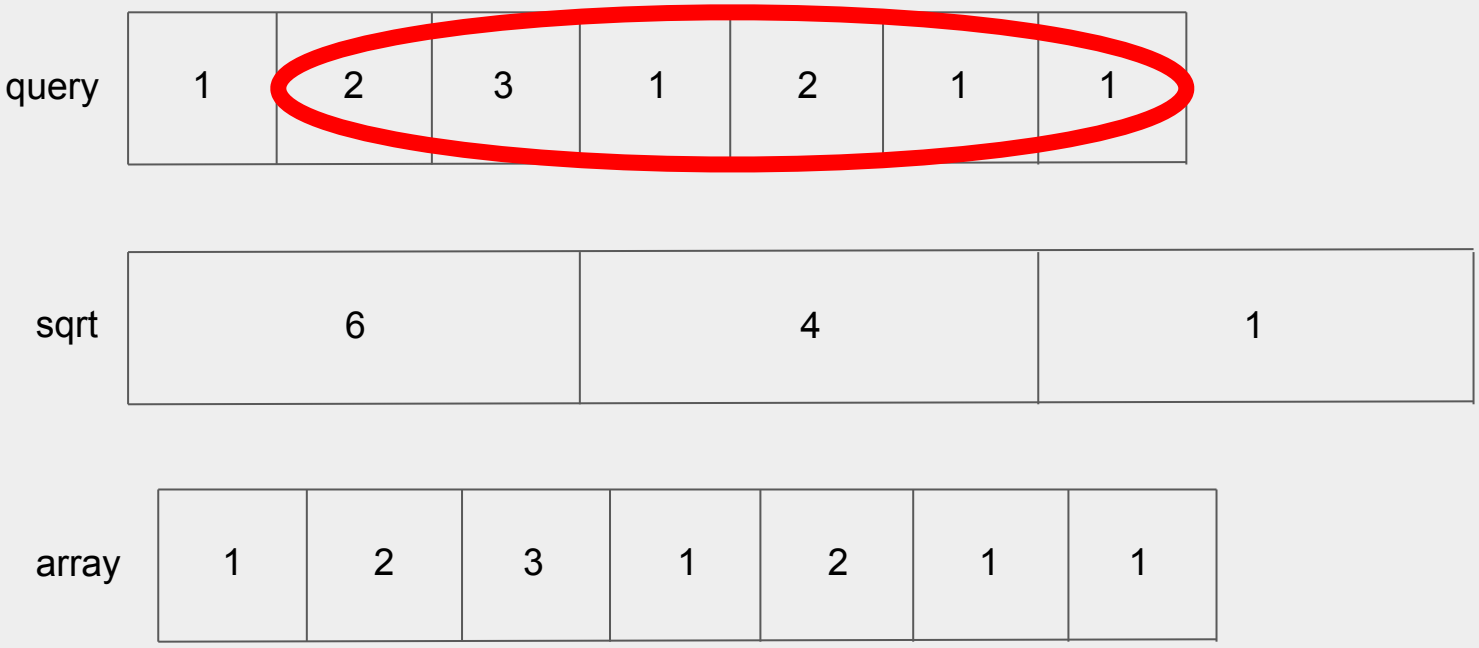


SQRT decomposition

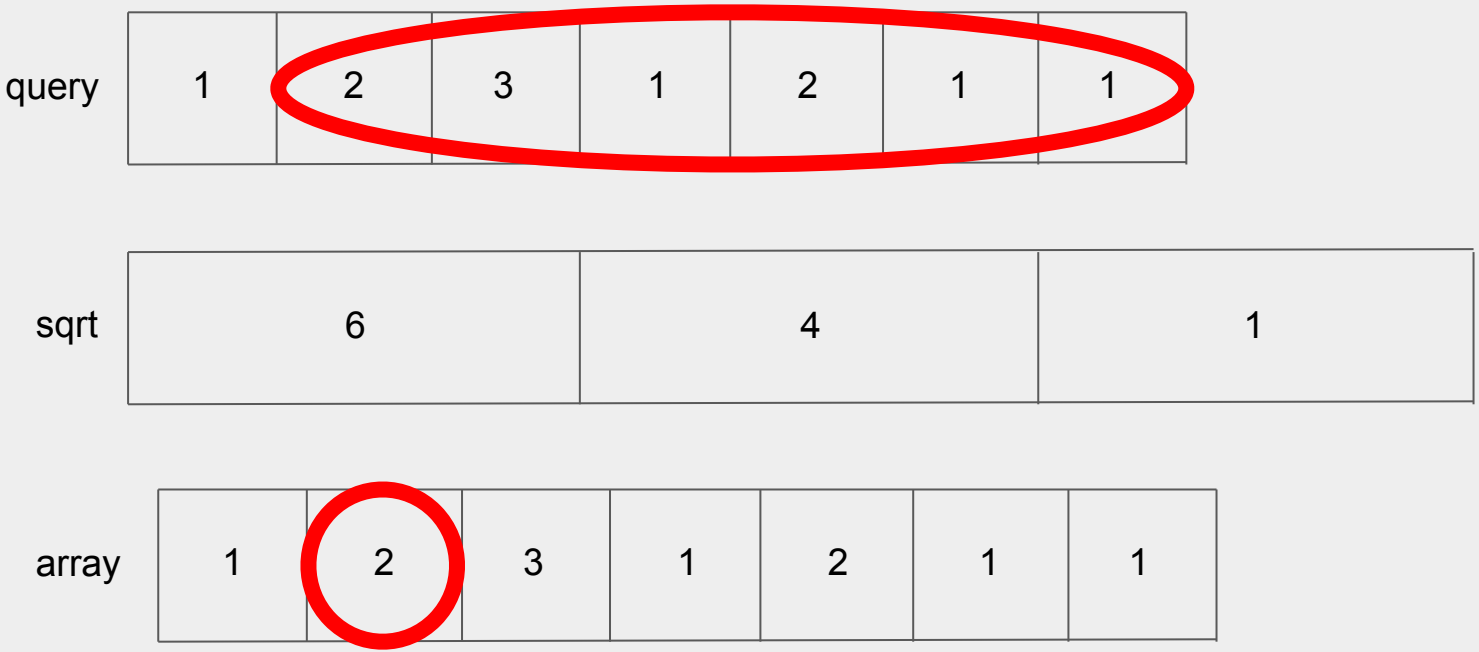
How do we respond to a query for the sum in a segment now?

Well, it's actually quite simple - let's take all the blocks that are completely in the query, and for the blocks that are not completely in it, we'll handle them element-wise.

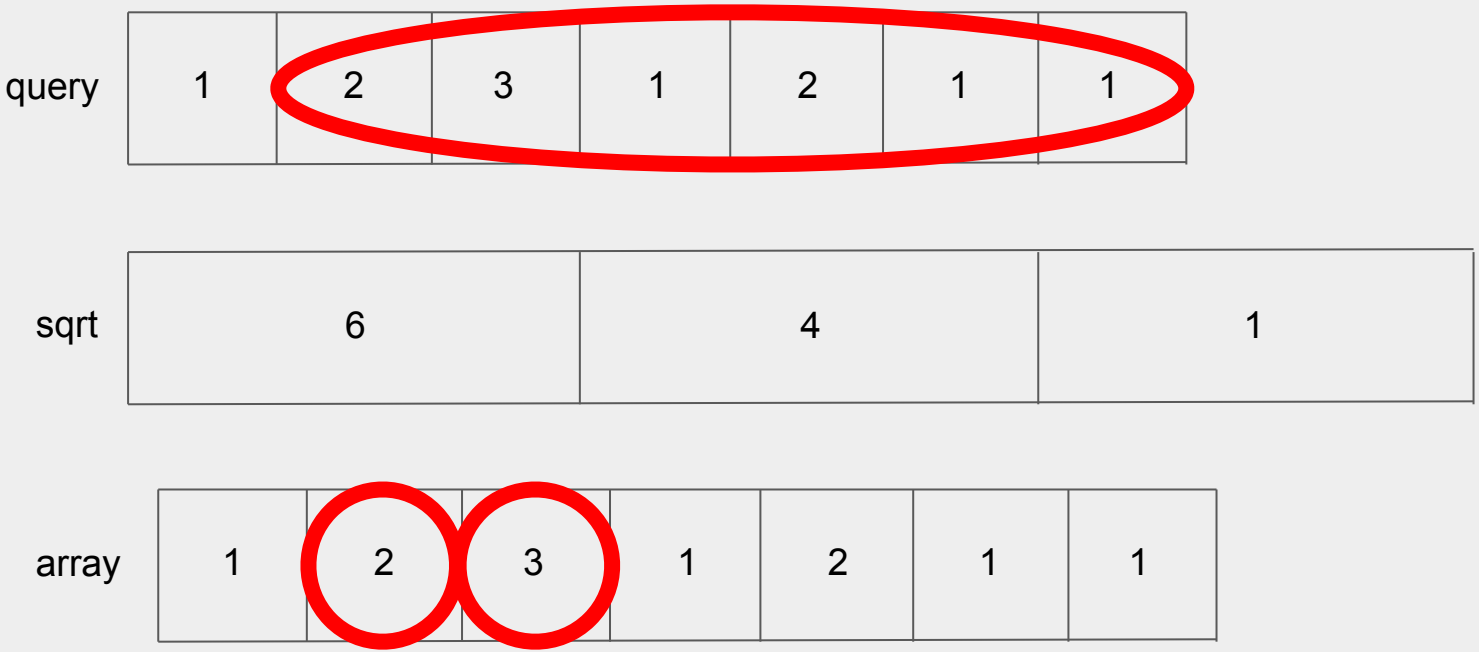
c = 3, sum = 0



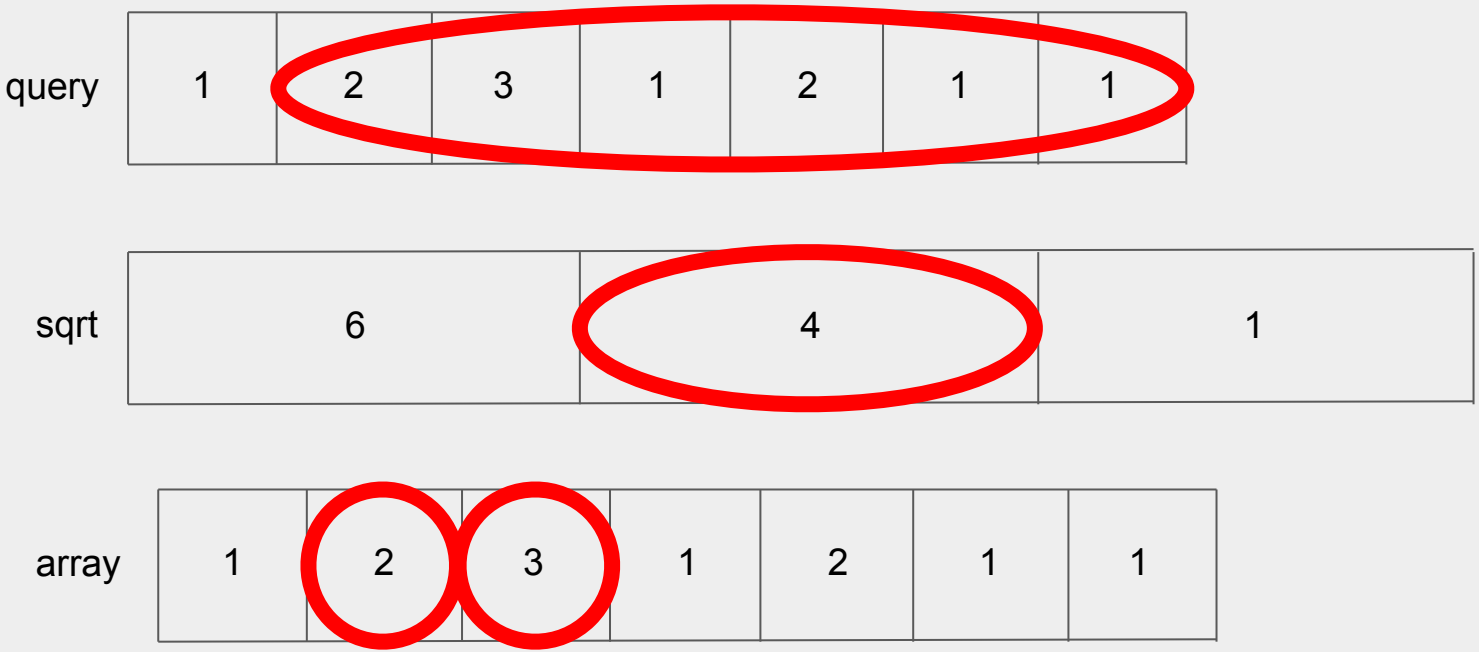
c = 3, sum = 2



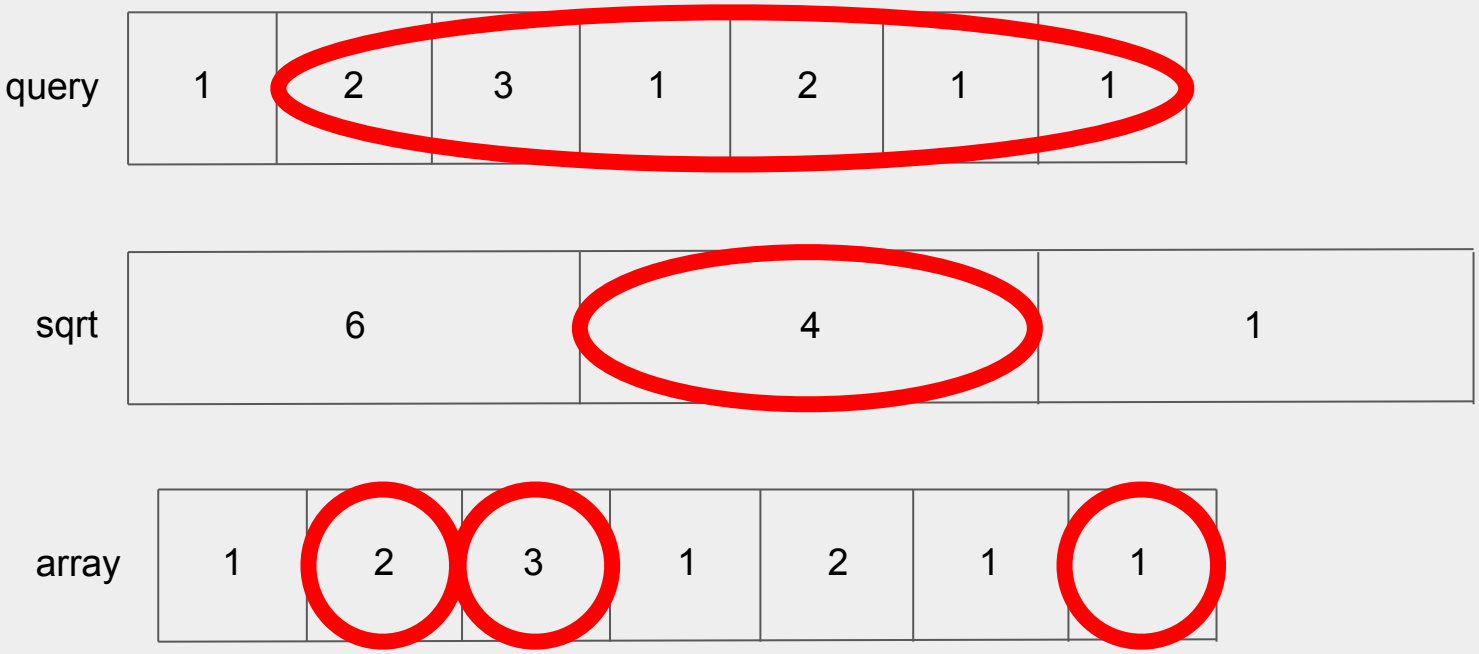
c = 3, sum = 5



c = 3, sum = 9

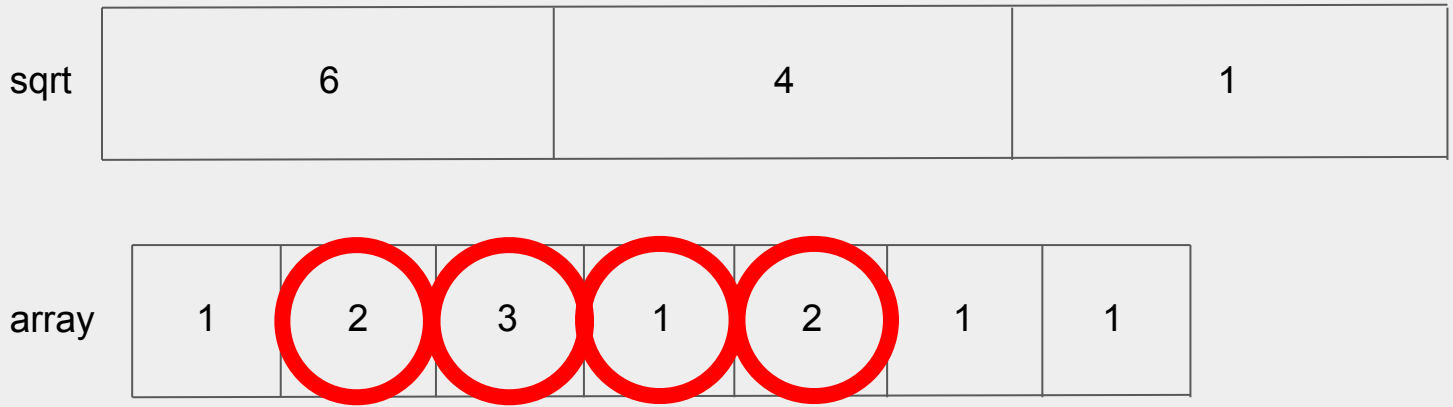


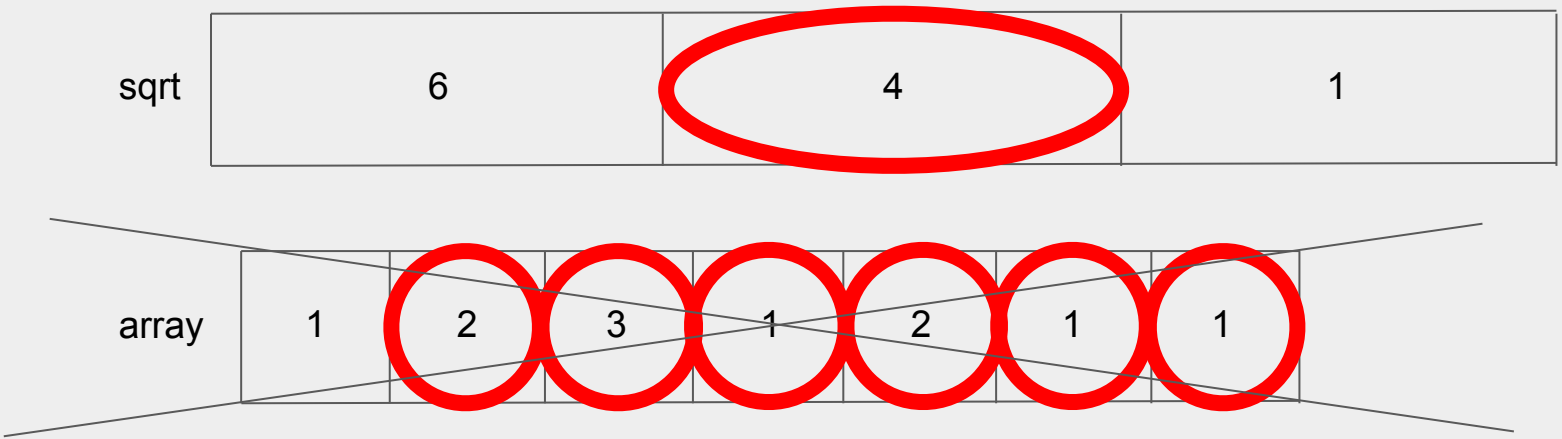
c = 3, sum = 10



Asymptotic

- Solo element - O(1), just add it to sum
- Block - O(1), just add it to sum
- Amount of blocks <= O(n / c)
- Amount of solo elements <= O(c), because we can take only 2 partial blocks(one - from left, one - from right)


sqrt
6
4
1
array
1
2
3
1
2
1
1


sqrt
6
4
1
array
1
2
3
1
2
1
1

```cpp
struct Block {
    int sum = 0;
};


int sum(int l, int r, const vector<Block> &sqrt_dec, const vector<int> &a) {
    int res = 0;
    while (l <= r) {
        // beginning of the block - l = c * x, r >= c * (x + 1) - 1
        if (l % c == 0 && l + c - 1 <= r) {
            res += sqrt_dec[l / c].sum;
            l += c; // we skip block
        }
        else {
            res += a[l];
            l += 1;
        }
    }
    return res;
}
```

```
    int q;
    cin >> q;
    for (int i = 0; i < q; i++) {
        int l, r;
        cin >> l >> r;
        l--;
        r--;
        cout << sum(l, r, sqrt_dec, a) << "\n";
    }
    return 0;
}
```

Success #stdin #stdout 0.01s 5564KB

stdin

```
7
1 2 3 1 2 1 1
5
2 7
1 6
3 4
4 5
5 7
```

stdout

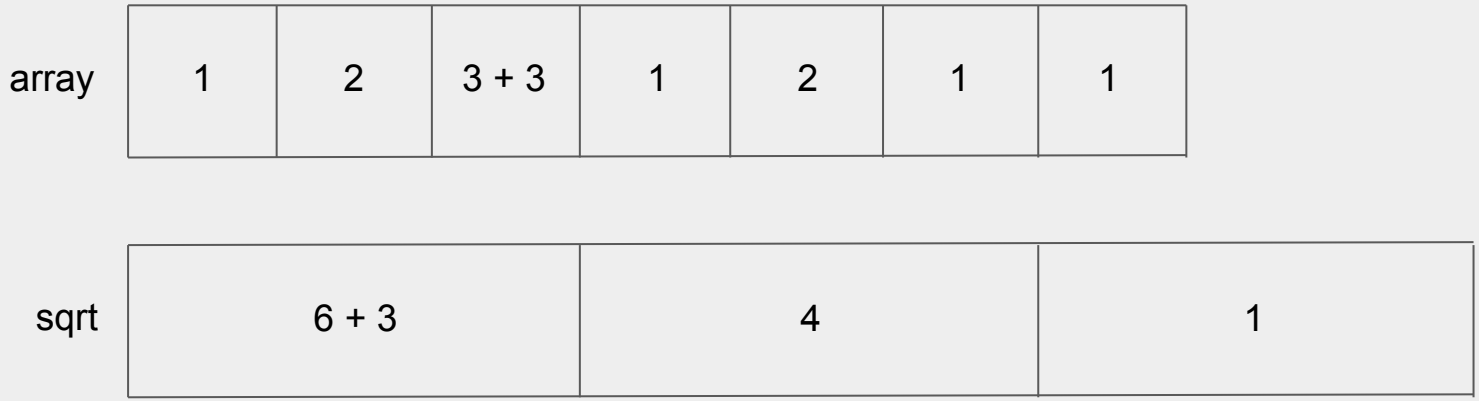
```
10
10
4
3
4
```

One element.

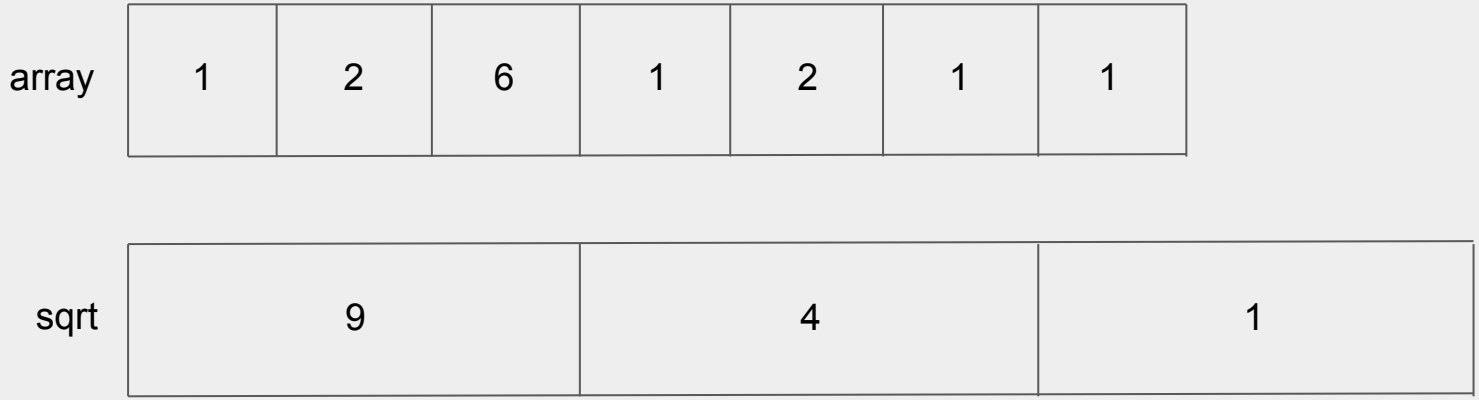
How to add to an element?

Equally simple - let's first add to the element itself, and then to the block it belongs to.

a[3] += 3



a[3] += 3



```
28.
29. void addToElem(int pos, int x, vector<Block> &sqrt_dec, vector<int> &a) {
30.     a[pos] += x;
31.     sqrt_dec[pos / c].sum += x;
32. }
33.
```

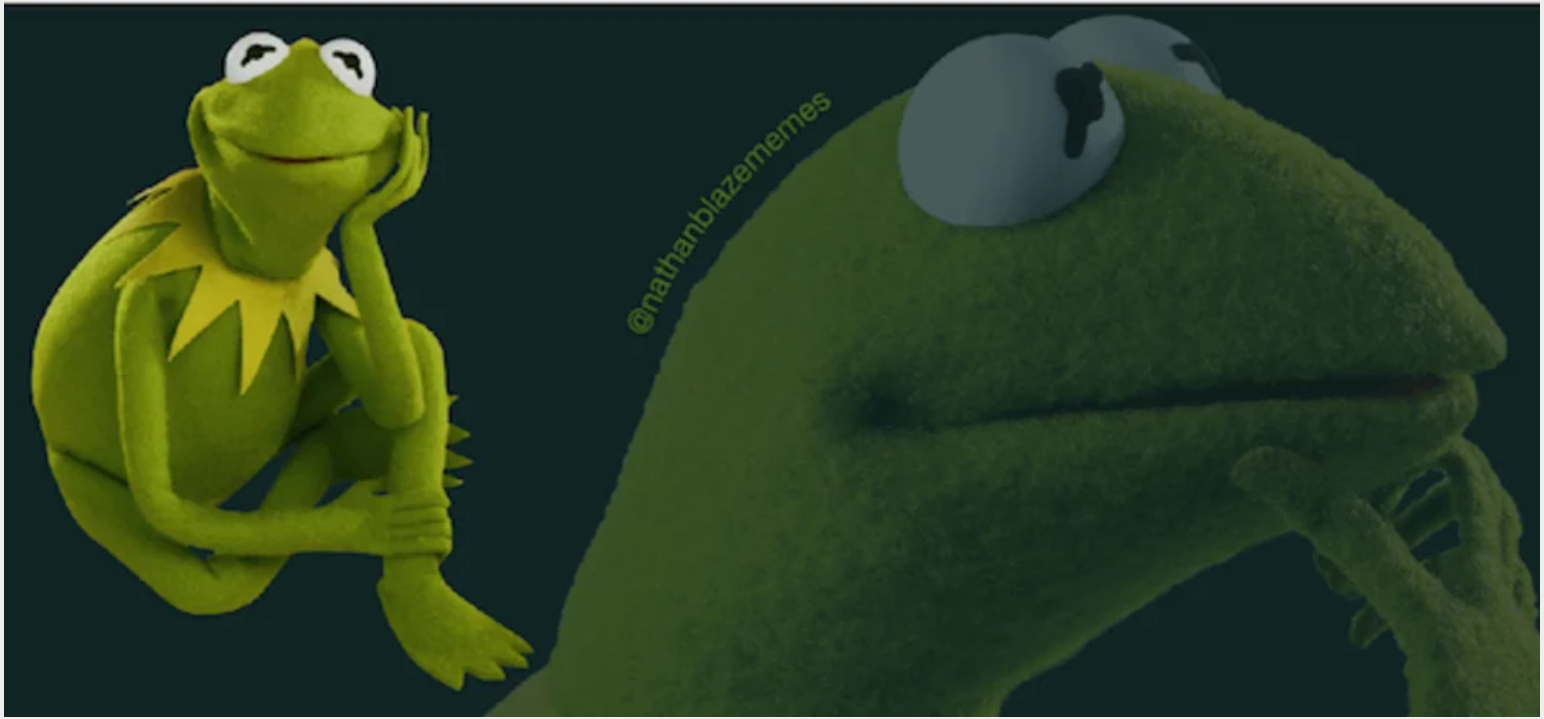
Segment

How to add to a segment?

Now, this is where it gets interesting. Let's come up with a technique called lazy propagation. In fact, it's a very important technique and we'll continue to use it further.


IF YOU EVER FEEL BAD ABOUT PROCRASTINATING
JUST REMEMBER THAT MOZART WROTE THE OVERTURE TO DON GIOVANNI THE MORNING IT PREMIERED

I don't procrastinate. I wait until the last minute to do things, because I will be older, and therefore wiser

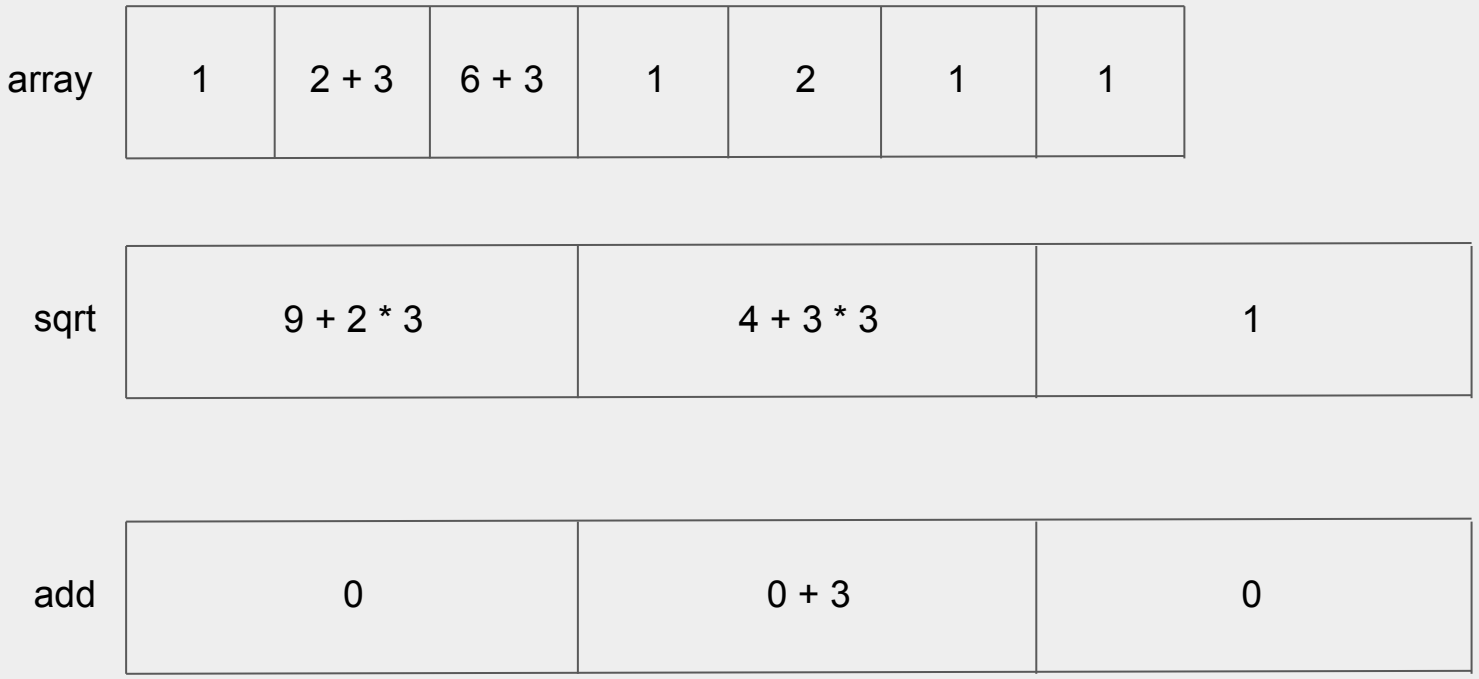


Lazy propagation

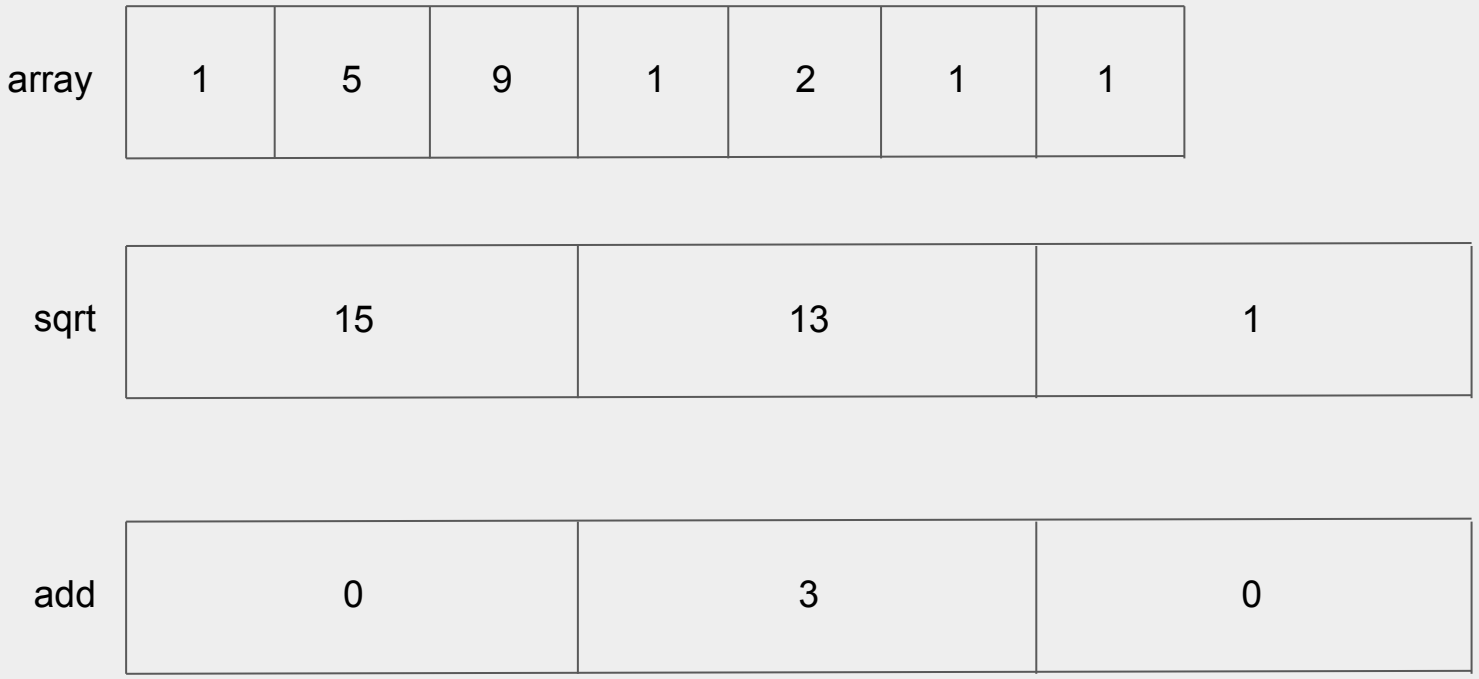
Lazy propagation - that's the correct line of thinking. It teaches us to do something only when it's absolutely necessary.

More formally speaking - let's maintain for each block the value by which it increased, while updating the sum for the block itself in an honest manner.

a[2..6] += 3



a[2..6] += 3



Lazy propagation

What should we do now if we access just one element in a certain block?

Let's push the block's addition onto all values in that block at once.

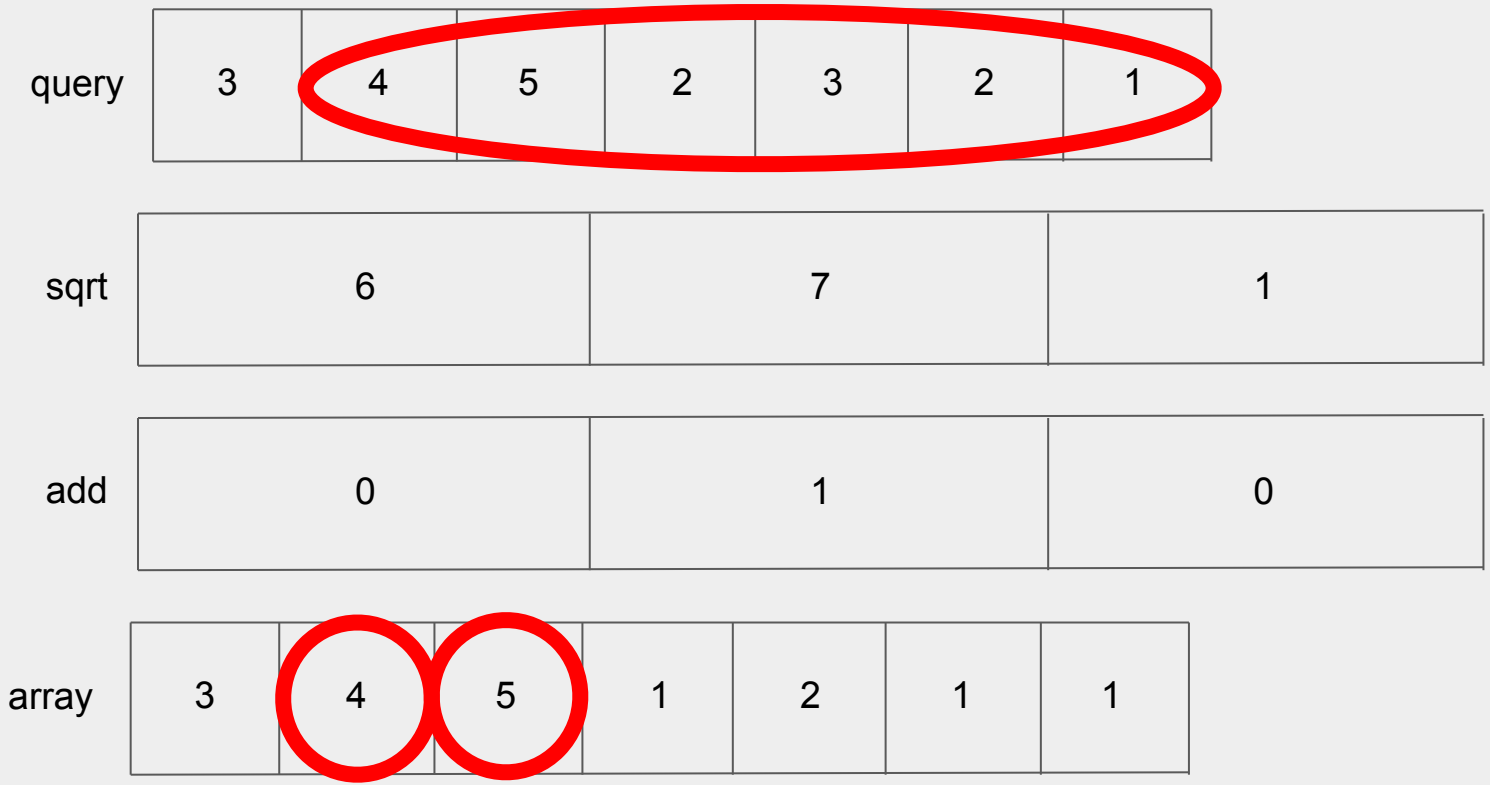
Lazy propagation

The first method involves pushing the block's addition onto all values in that block at once.

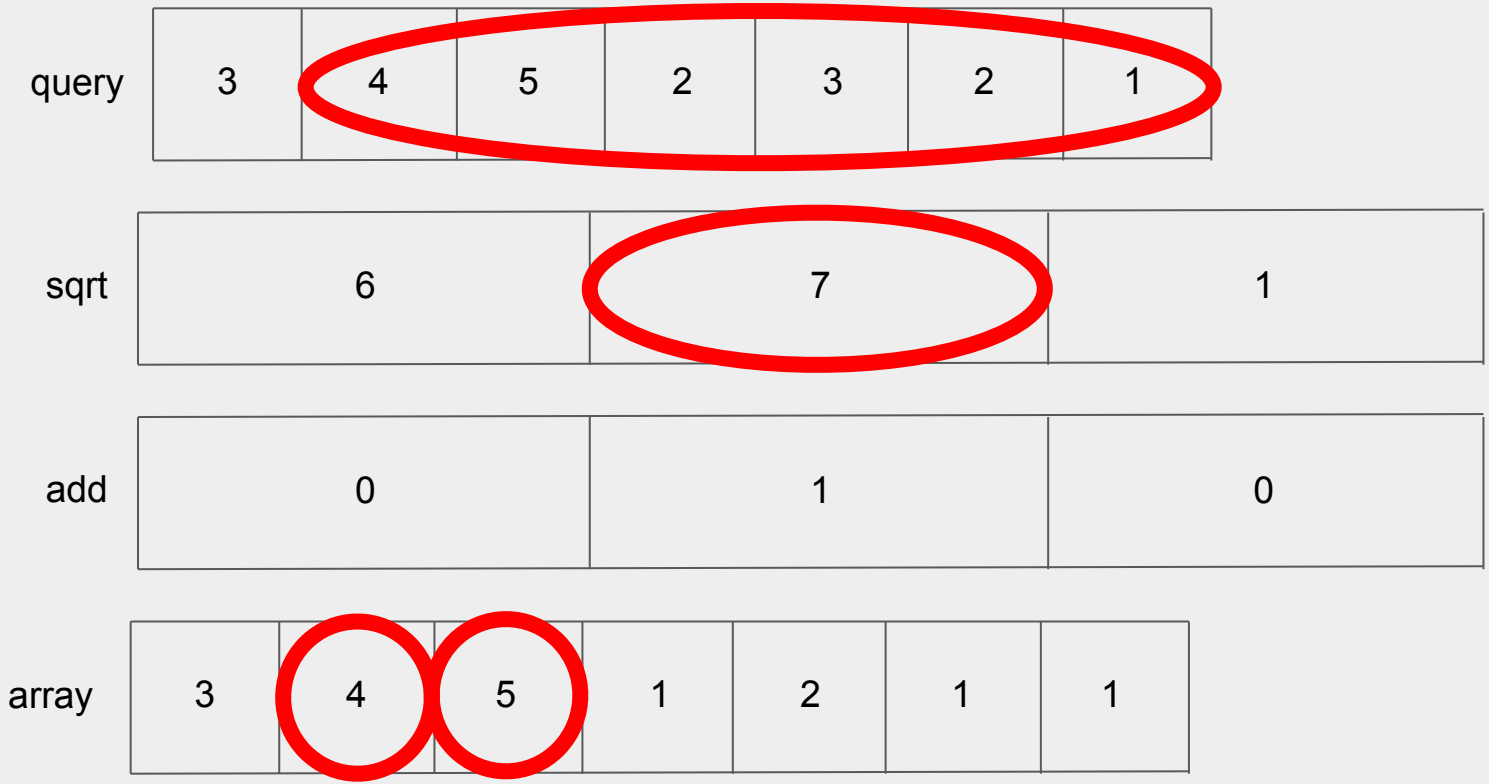
c = 3, sum = 0

c = 3, push

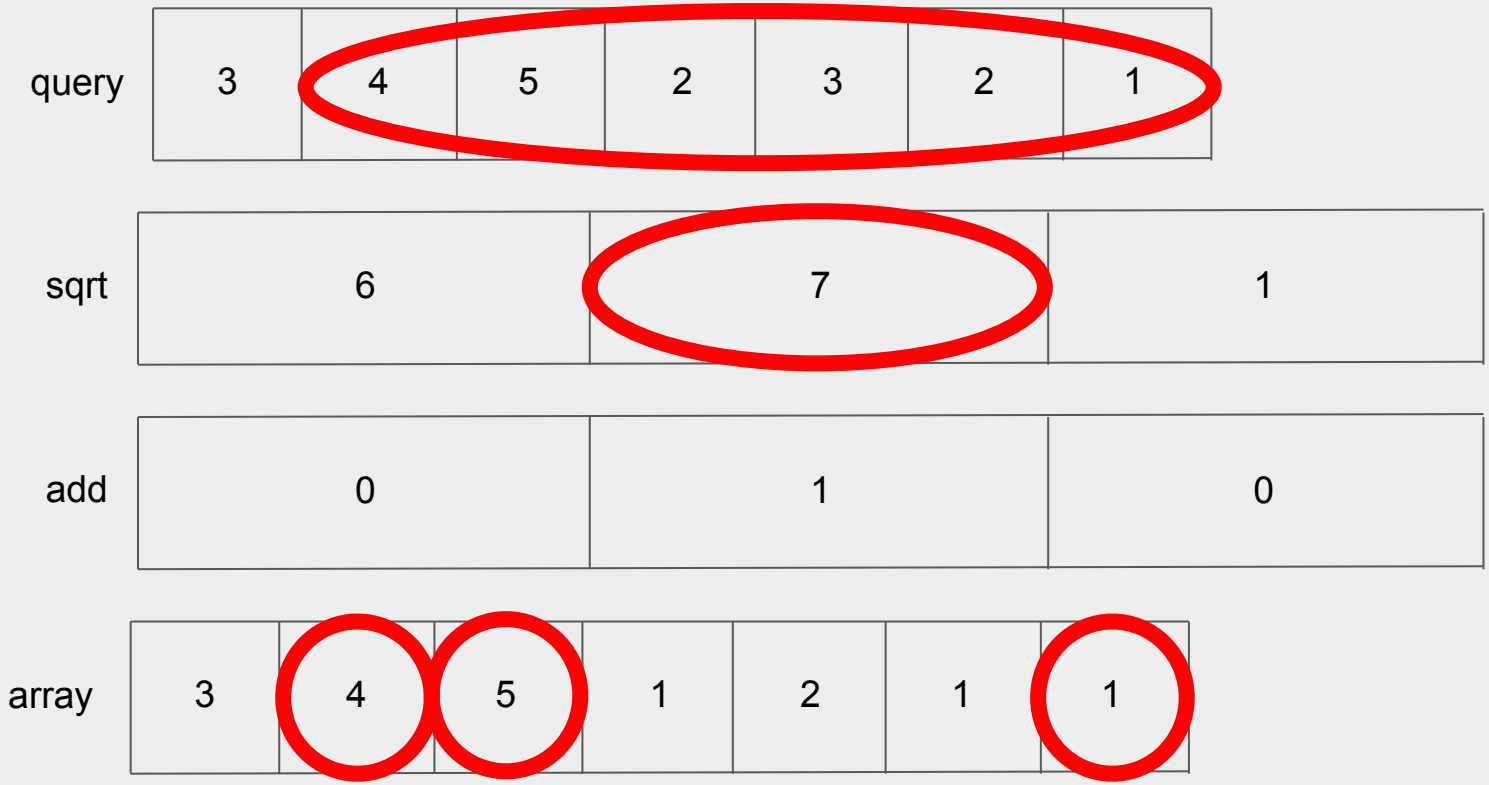
c = 3, sum = 9



c = 3, sum = 16



c = 3, sum = 17



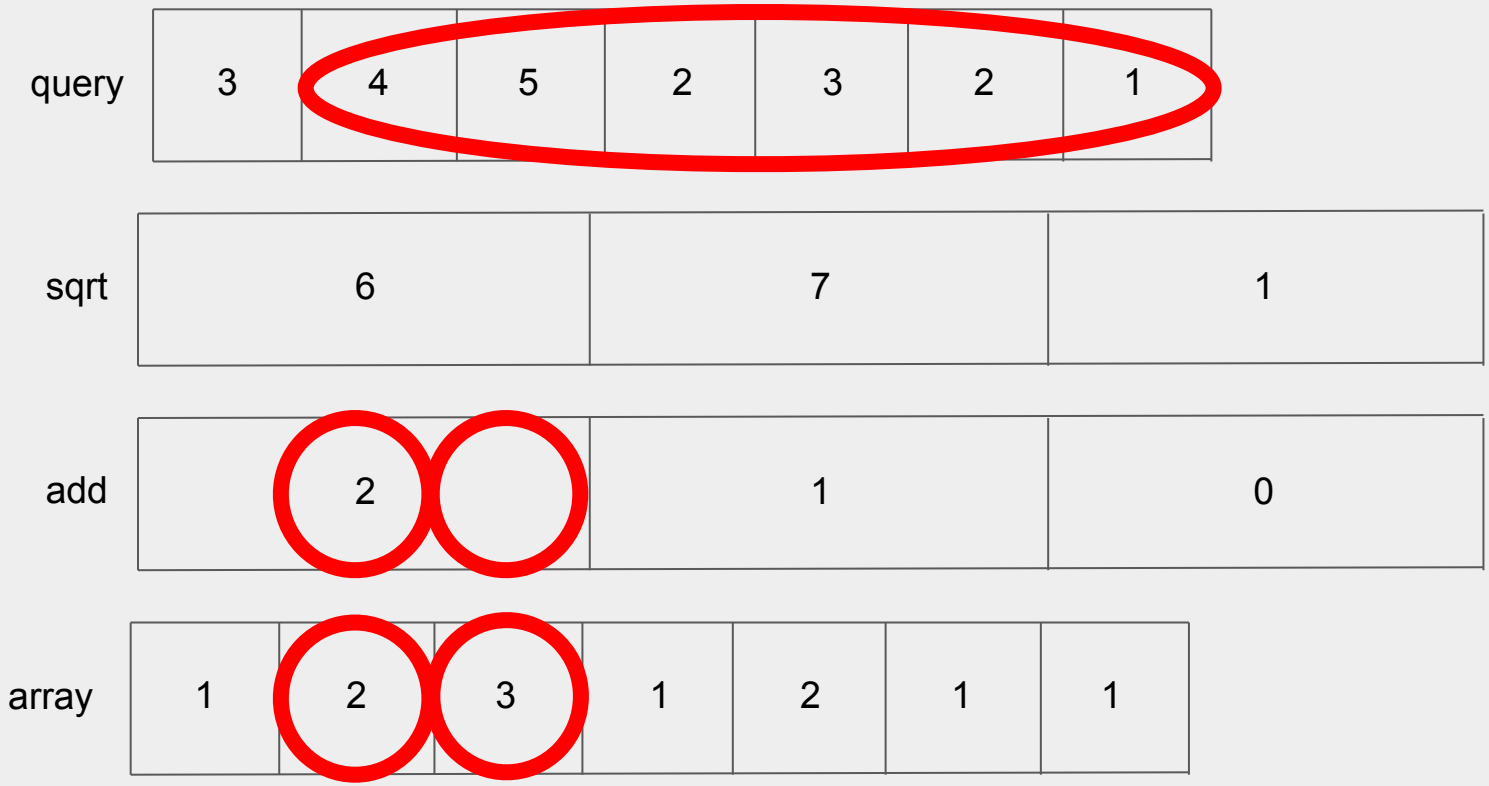
Lazy propagation

The second method involves simply adding the required value to the element that needs to be pushed.

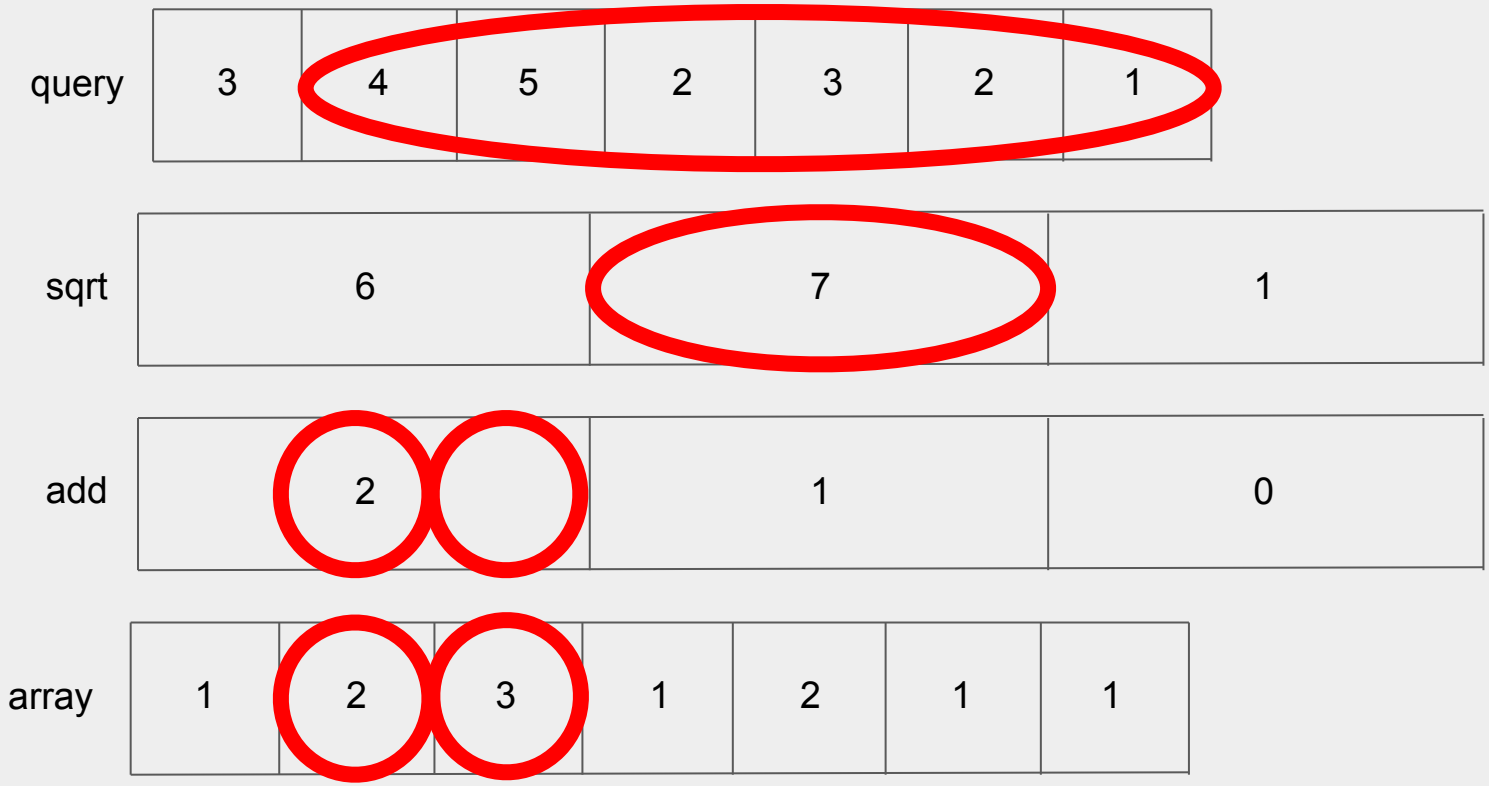
c = 3, sum = 0

c = 3, sum = 4

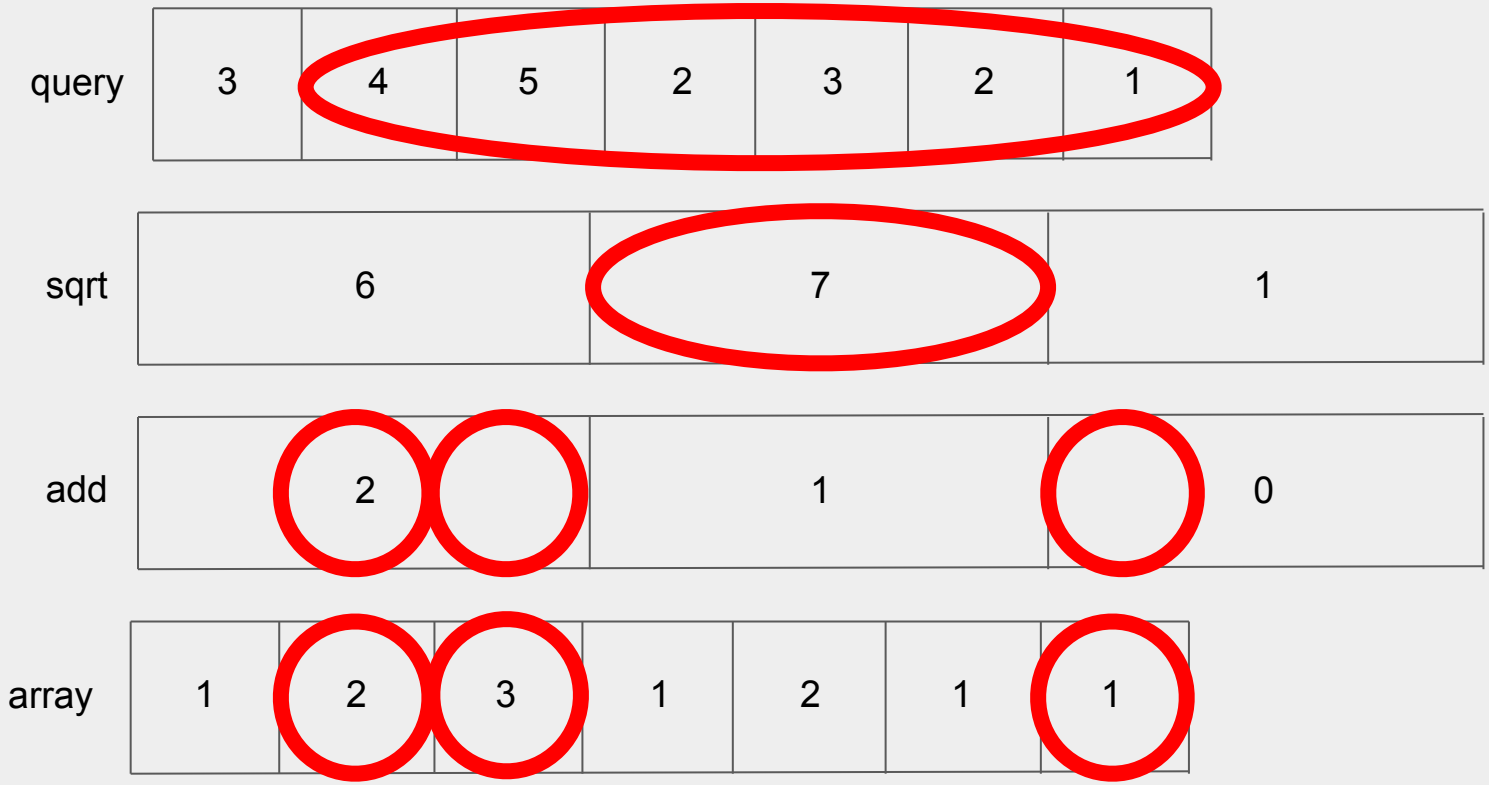
c = 3, sum = 9



c = 3, sum = 16



c = 3, sum = 16



Let’s compare

Let's compare these two methods.

1. Since we only affect elements in two blocks, it will work asymptotically fast (in O(c)). However, this method is less fast and is usually used more frequently in other structures.
2. The second method operates in O(1), but it's only applicable in the root decomposition, unfortunately.

```cpp
const int c = 3;


struct Block {
    int sum = 0;
    int add = 0;
};


int sum(int l, int r, const vector<Block> &sqrt_dec, const vector<int> &a) {
    int res = 0;
    while (l <= r) {
        // beginning of the block - l = c * x, r >= c * (x + 1) - 1
        if (l % c == 0 && l + c - 1 <= r) {
            res += sqrt_dec[l / c].sum;
            l += c; // we skip block
        }
        else {
            res += a[l] + sqrt_dec[l / c].add;
            l += 1;
        }
    }
    return res;
}

```

```
35. void add(int l, int r, int x, vector<Block> &sqrt_dec, vector<int> &a) {
36.     int res = 0;
37.     while (l <= r) {
38.         // beginning of the block - l = c * coef, r >= c * (coef + 1) - 1
39.         if (l % c == 0 && l + c - 1 <= r) {
40.             sqrt_dec[l / c].add += x;
41.             sqrt_dec[l / c].sum += x * c;
42.             l += c; // we skip block
43.         }
44.         else {
45.             a[l] += x;
46.             sqrt_dec[l / c].sum += x;
47.             l += 1;
48.         }
49.     }
50. }
```

```cpp
int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    vector<Block> sqrt_dec(n / c + 1);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
        sqrt_dec[i / c].sum += a[i];
    }
    int q;
    cin >> q;
    for (int i = 0; i < q; i++) {
        int t, l, r;
        cin >> t >> l >> r;
        l--;
        r--;
        if (t) {
            cout << sum(l, r, sqrt_dec, a) << "\n";
        }
        else {
            add(l, r, 1, sqrt_dec, a);
        }
    }
    return 0;
}
```

stdin

```
7
1 2 3 1 2 1 1
4
1 2 7
0 3 6
0 4 7
1 2 7
```

stdout

```
10
18
```

RMQ

You are given an array of n natural numbers.

1. You need to be able to find the minimum in a segment.
2. You need to be able to add to an element.
3. You need to be able to add in a segment.

RMQ

Let's expand on the concept of root decomposition.

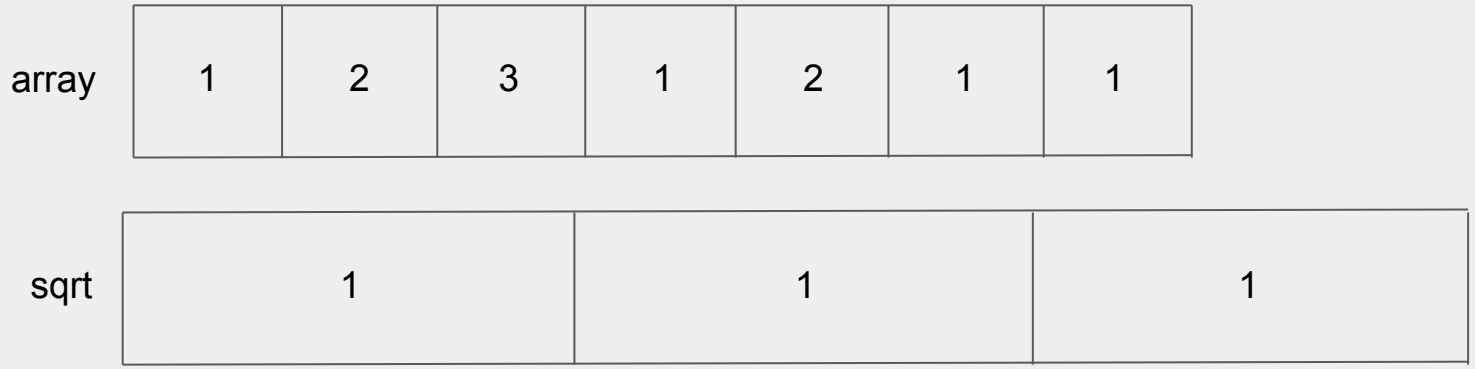
1. You need to be able to find the minimum in a segment.
2. You need to be able to SUBTRACT from an element.
3. You need to be able to SUBTRACT in a segment.

(IMPORTANT: At this point, we are only considering subtraction for finding the minimum.)

Solution

Let's store in each block the value of the minimum in that block.

c = 3



```
vector<Block> sqrt_dec(n / c + 1);
for (int i = 0; i < n; i++) {
    cin >> a[i];
    sqrt_dec[i / c].minn = min(a[i], sqrt_dec[i / c].minn);
}
```

Solution

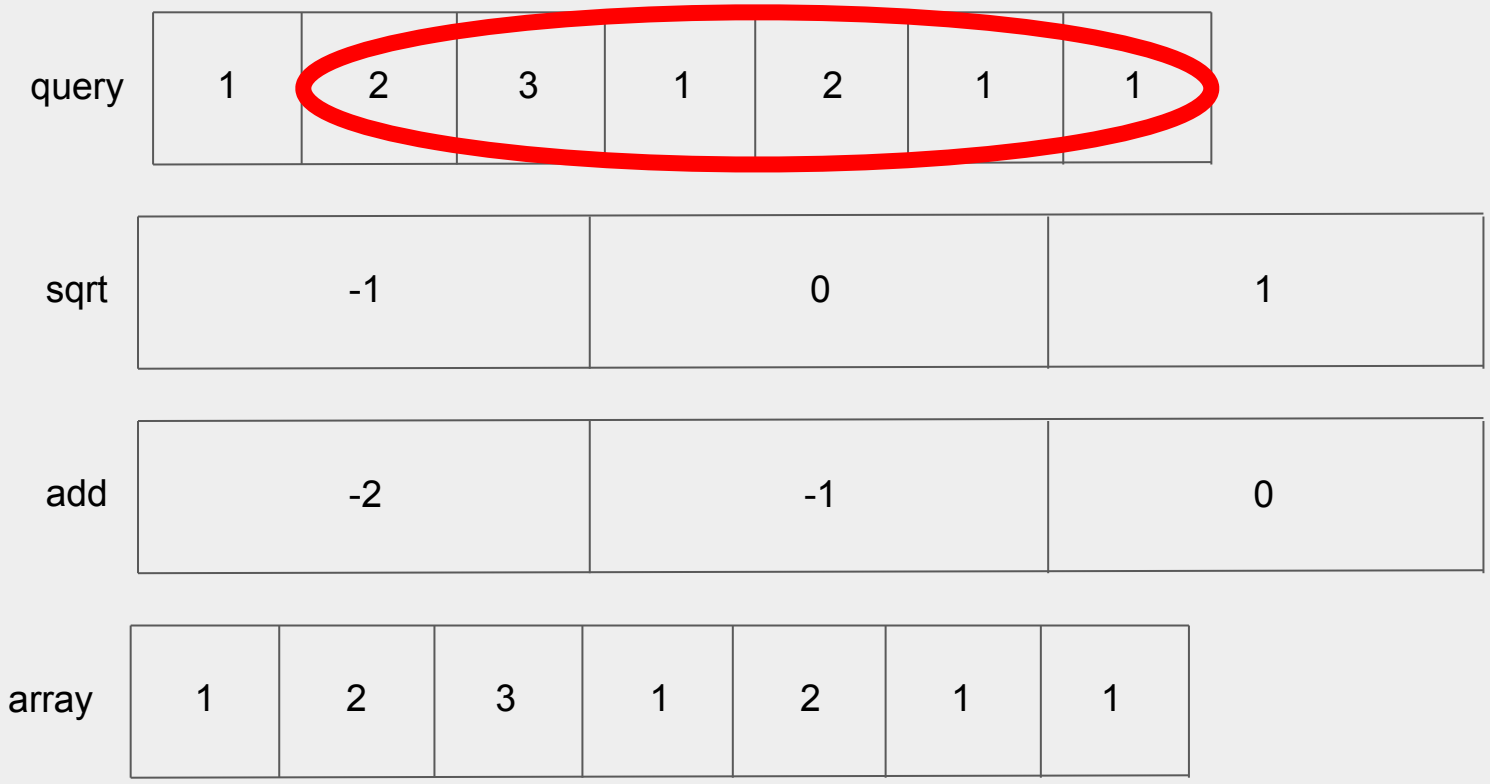
If we need to update one element, we simply update it and then update the minimum in its block.

```
30.  void addToElem(int pos, int x, vector<Block> &sqrt_dec, vector<int> &a) {
31.      a[pos] += x;
32.      sqrt_dec[pos / c].minn = min(x, sqrt_dec[pos / c].minn);
33.  }
```

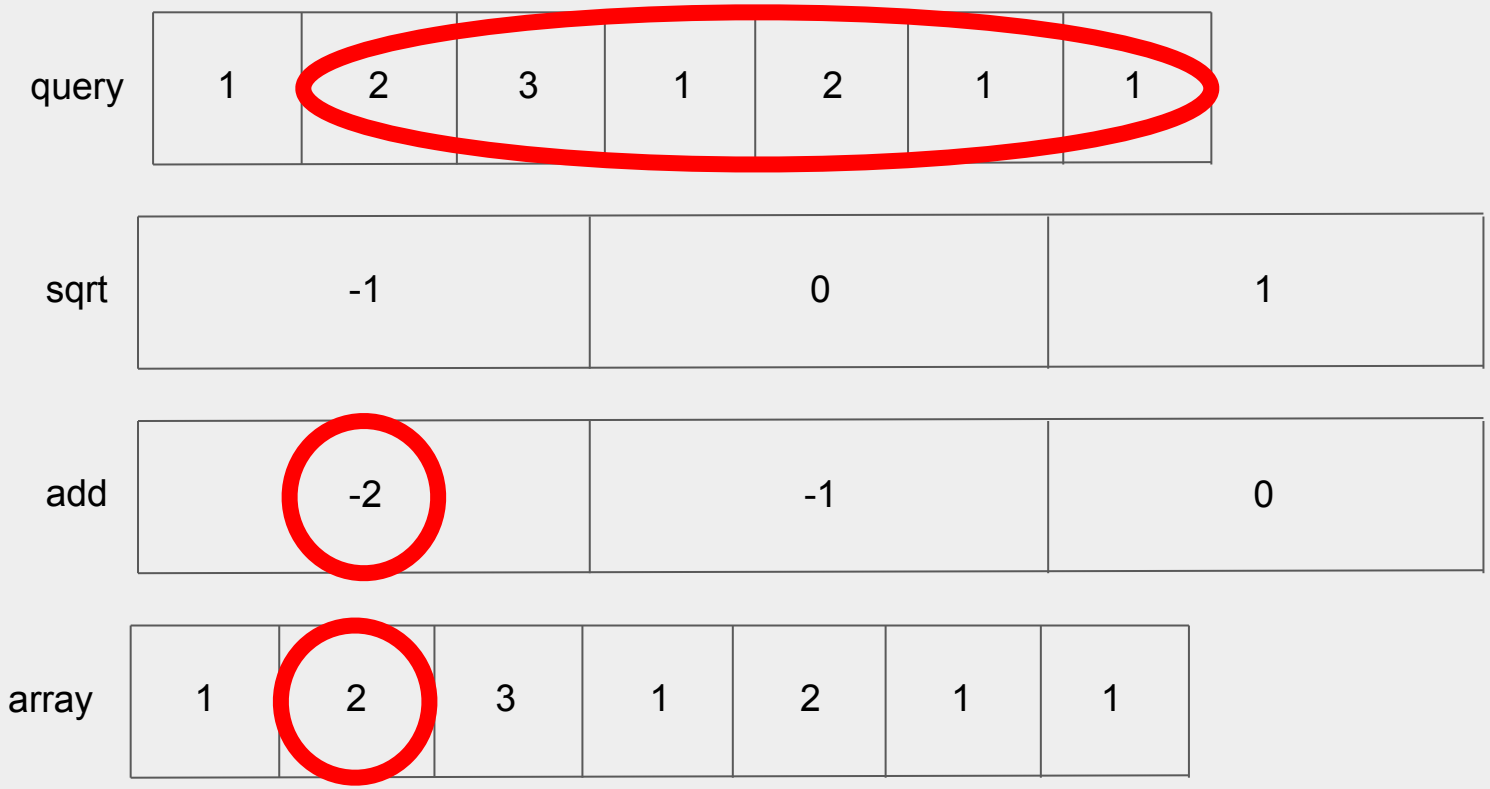
Solution

1. For a single element, we directly compare this element with the current answer.
2. For a block, we take the minimum value in the entire block and then compare it with the current answer.

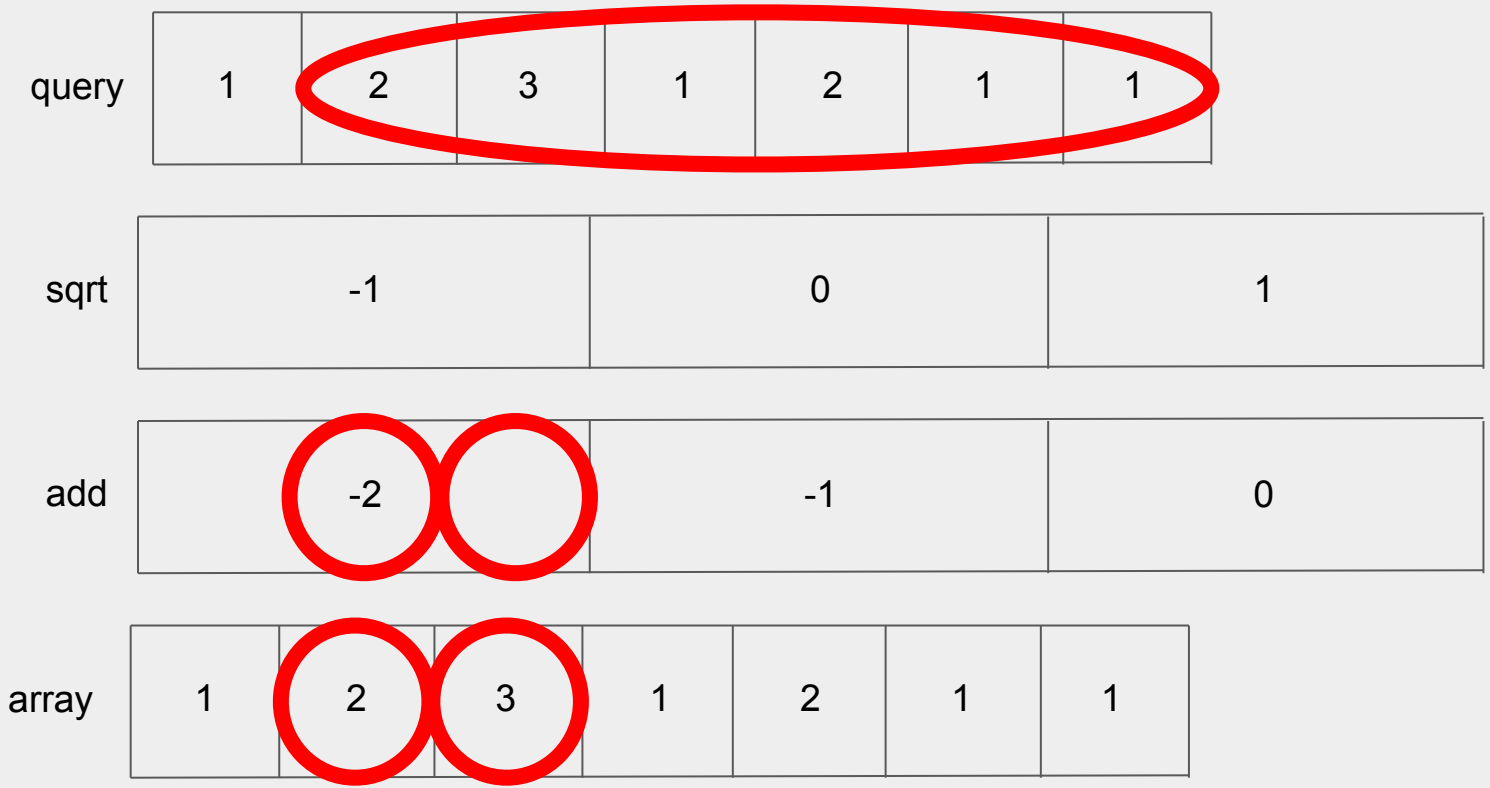
c = 3, min = inf



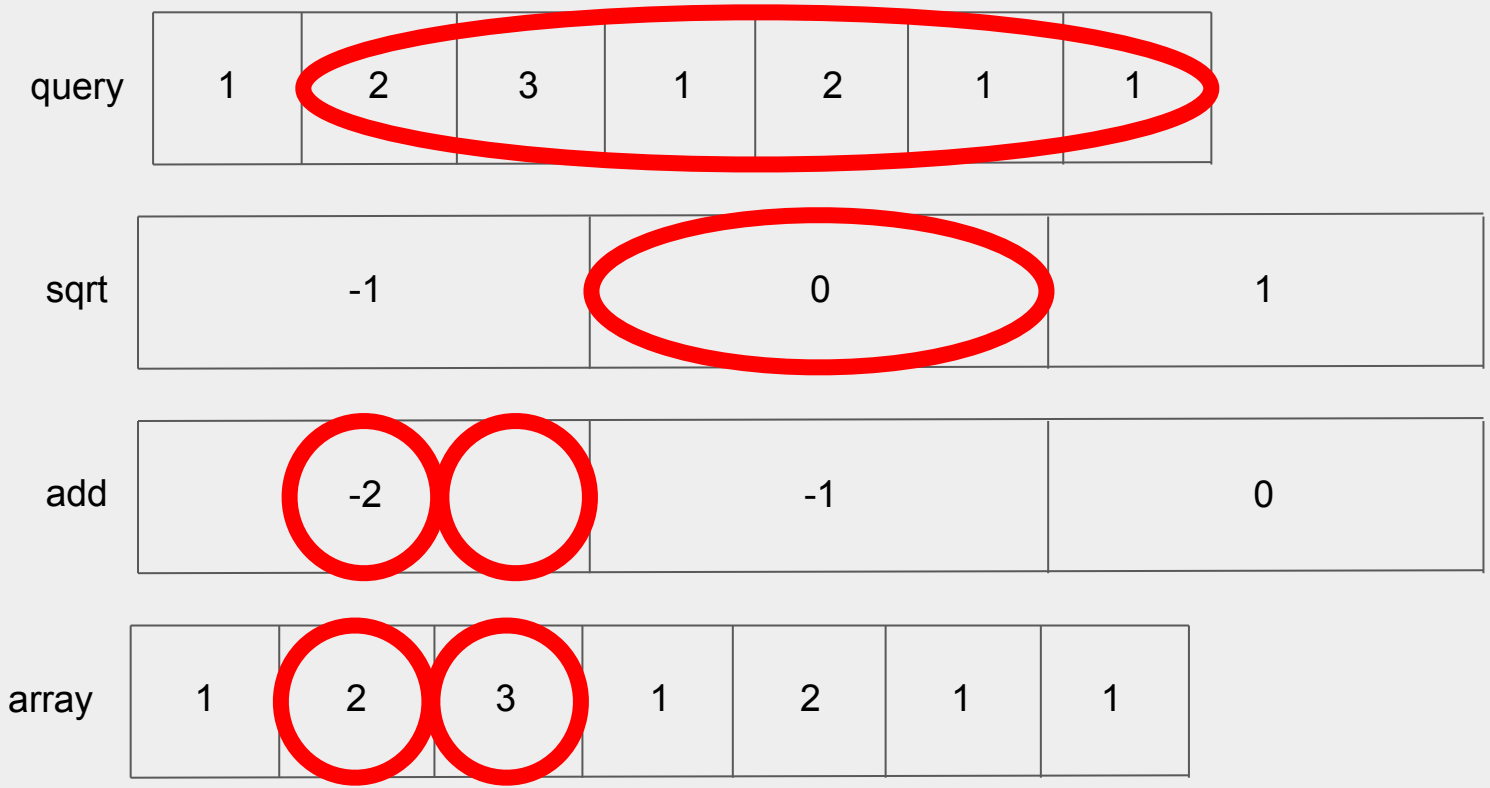
c = 3, min = min(inf, 2 - 2) = 0



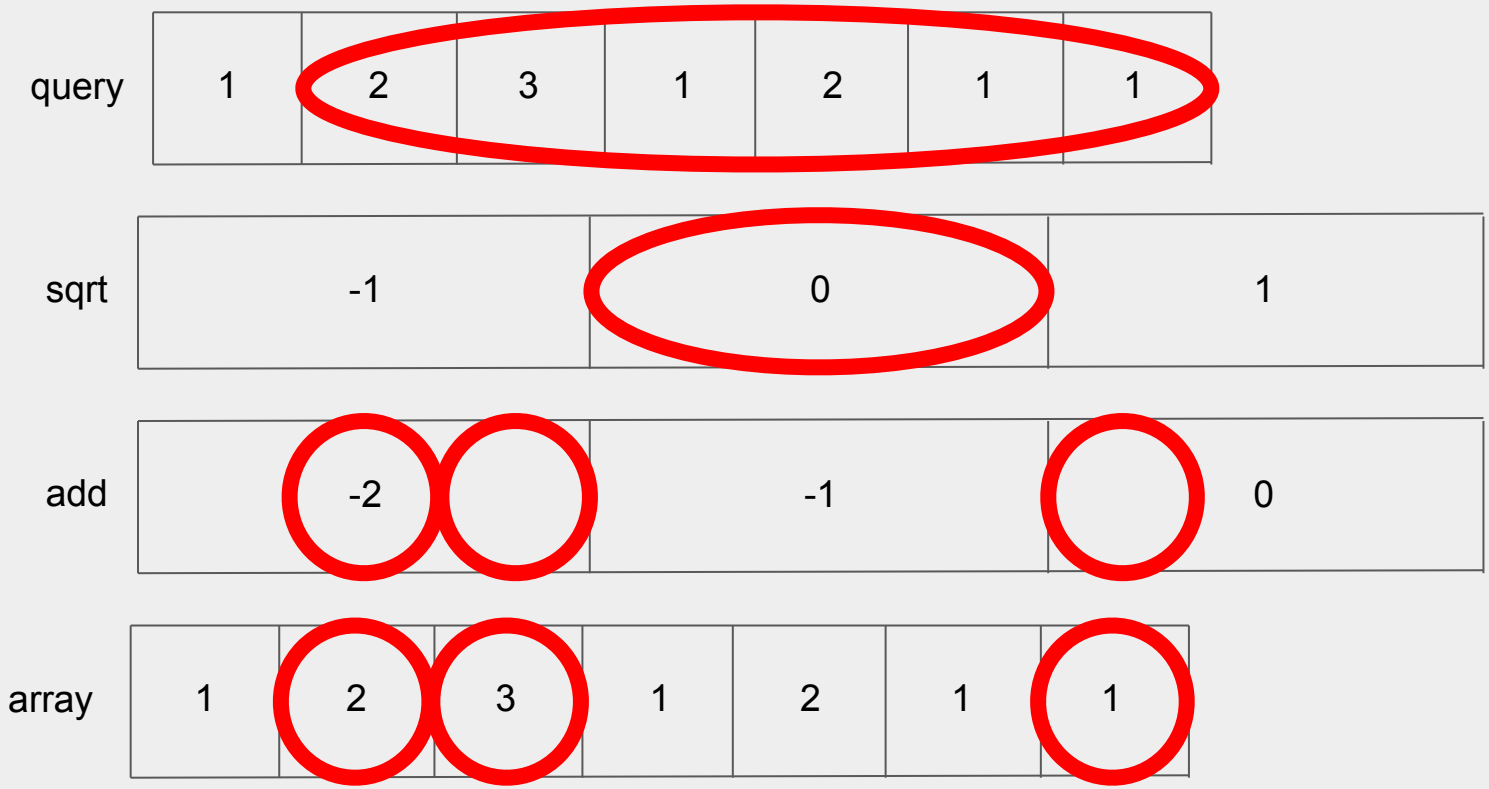
c = 3, min = min(0, 3 - 2) = 0



c = 3, min = min(0, 0) = 0



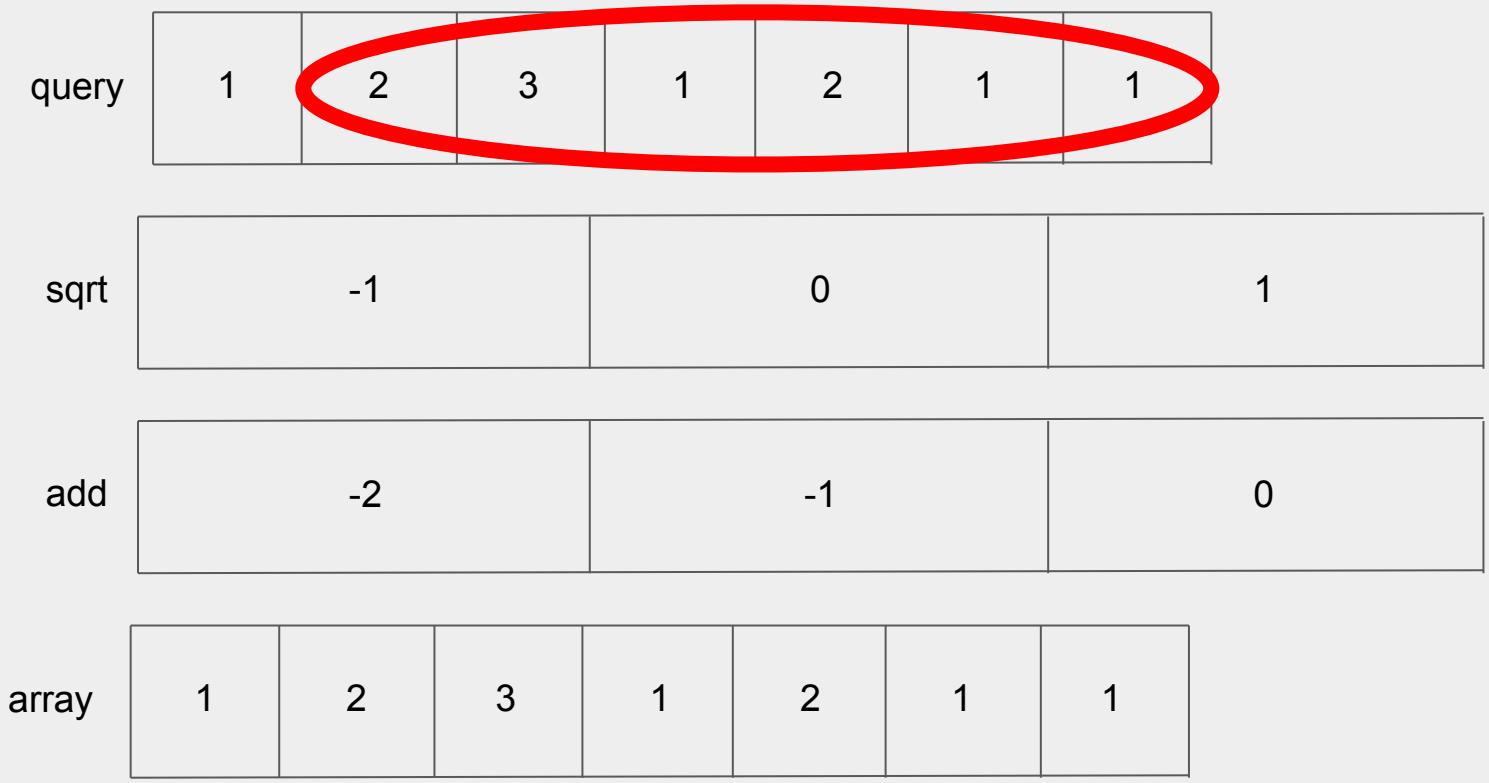
c = 3, min = min(0, 1 + 0) = 0



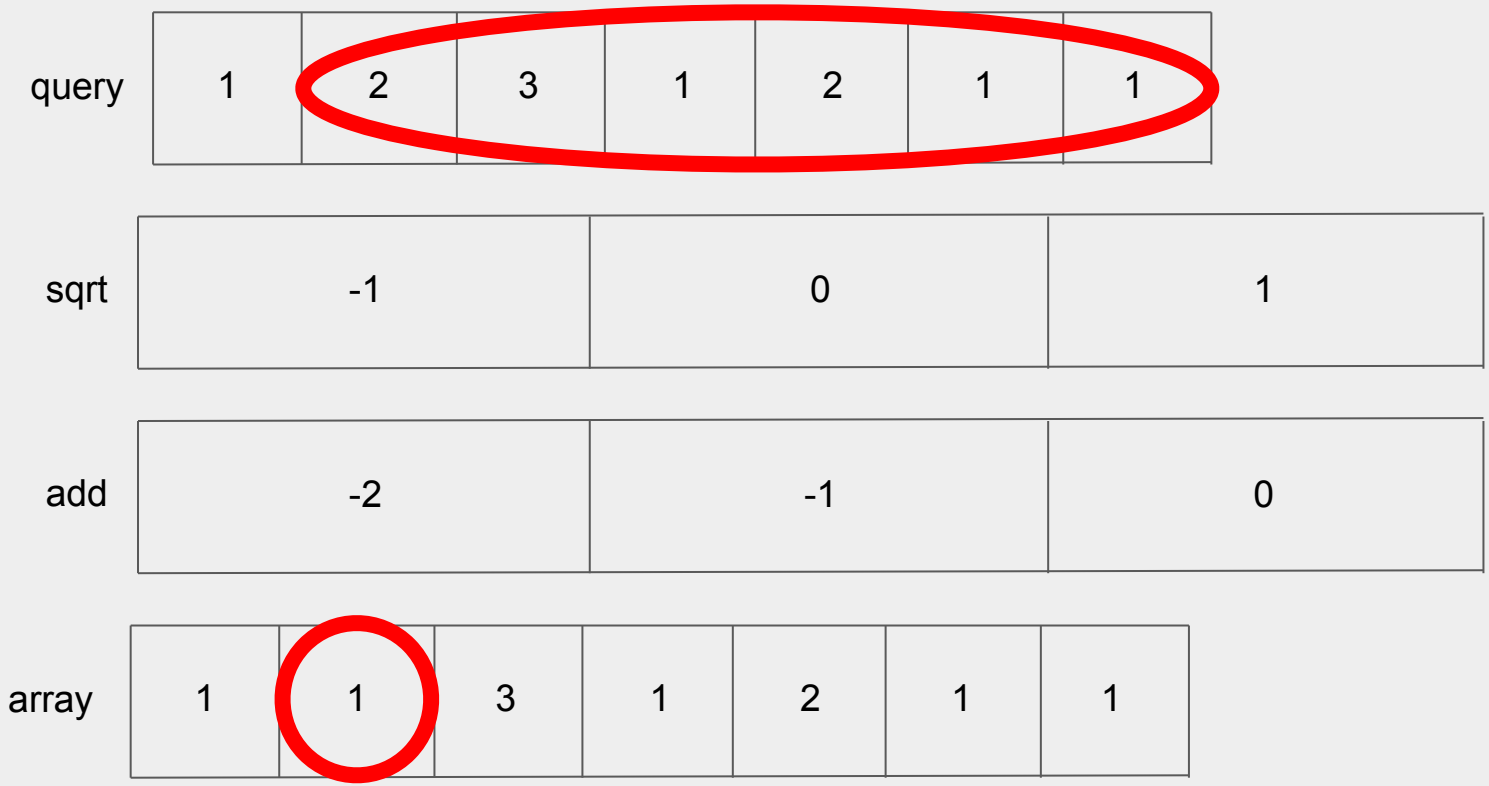
```cpp
struct Block {
    int minn = 1e9;
    int add = 0;
};


int minn(int l, int r, const vector<Block> &sqrt_dec, const vector<int> &a) {
    int res = 1e9;
    while (l <= r) {
        // beginning of the block - l = c * x, r >= c * (x + 1) - 1
        if (l % c == 0 && l + c - 1 <= r) {
            res = min(res, sqrt_dec[l / c].minn);
            l += c; // we skip block
        }
        else {
            res = min(res, a[l] + sqrt_dec[l / c].add);
            l += 1;
        }
    }
    return res;
}
```

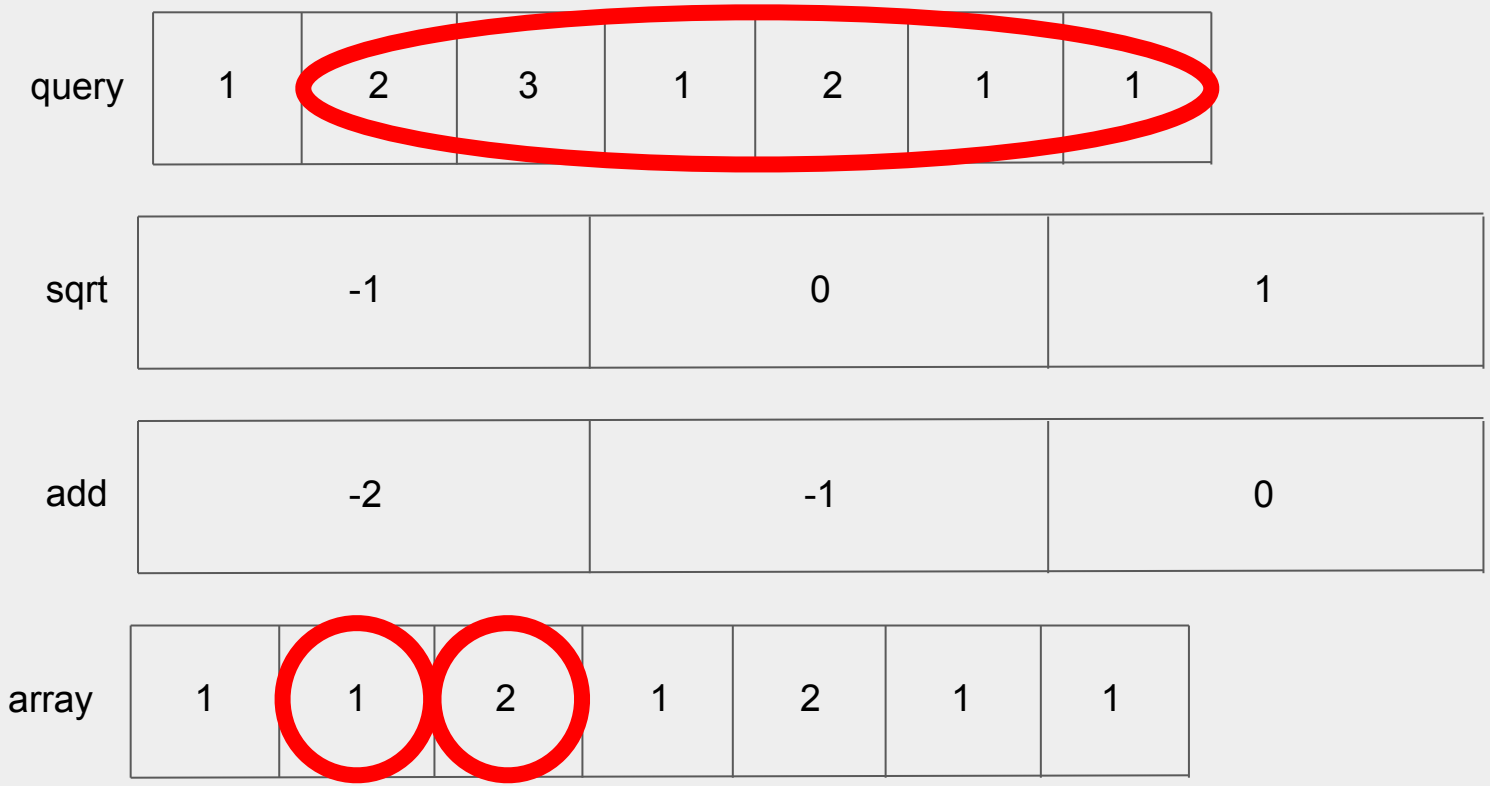
c = 3, a[2..7] -= 1



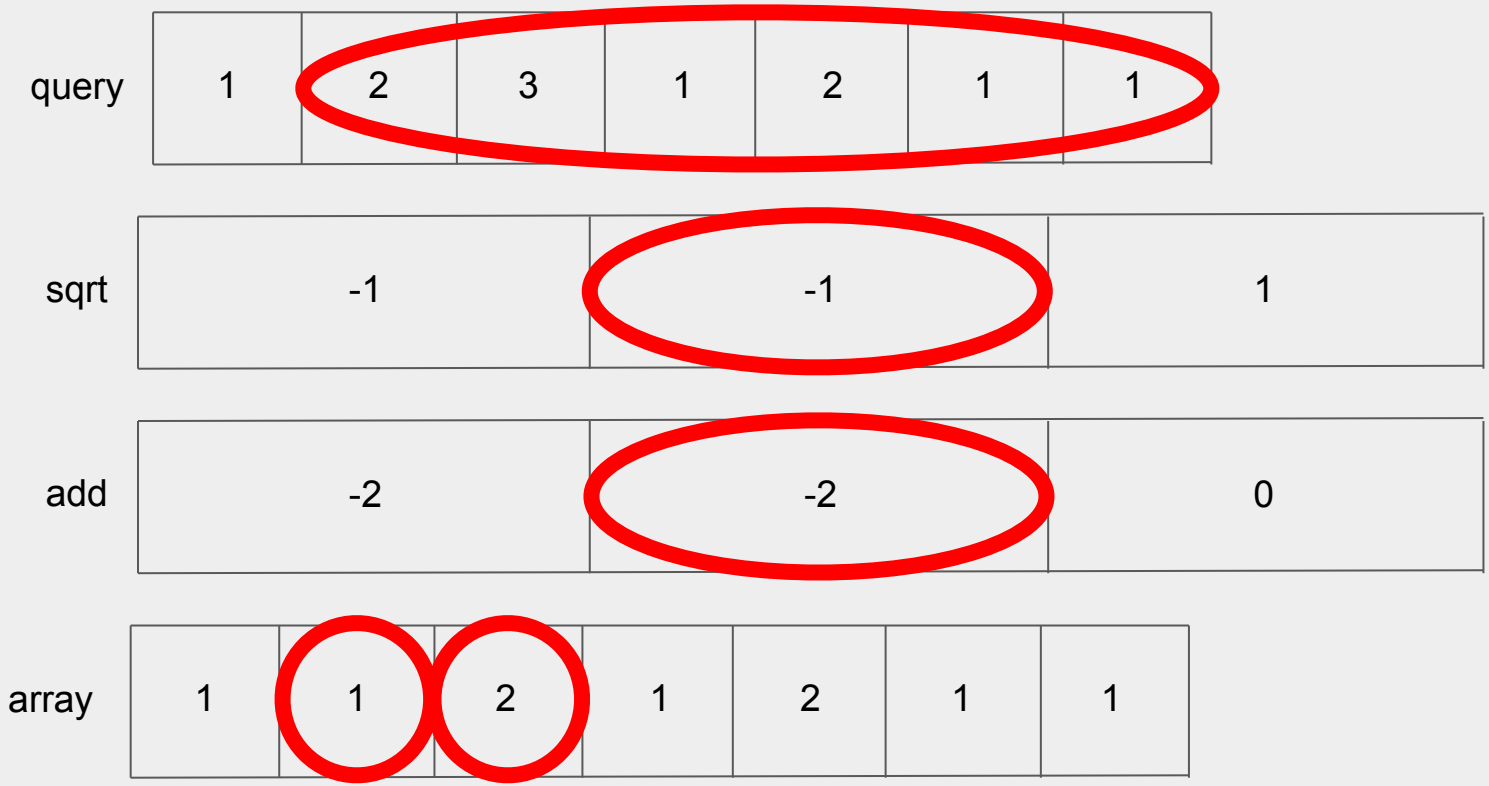
c = 3, a[2..7] -= 1



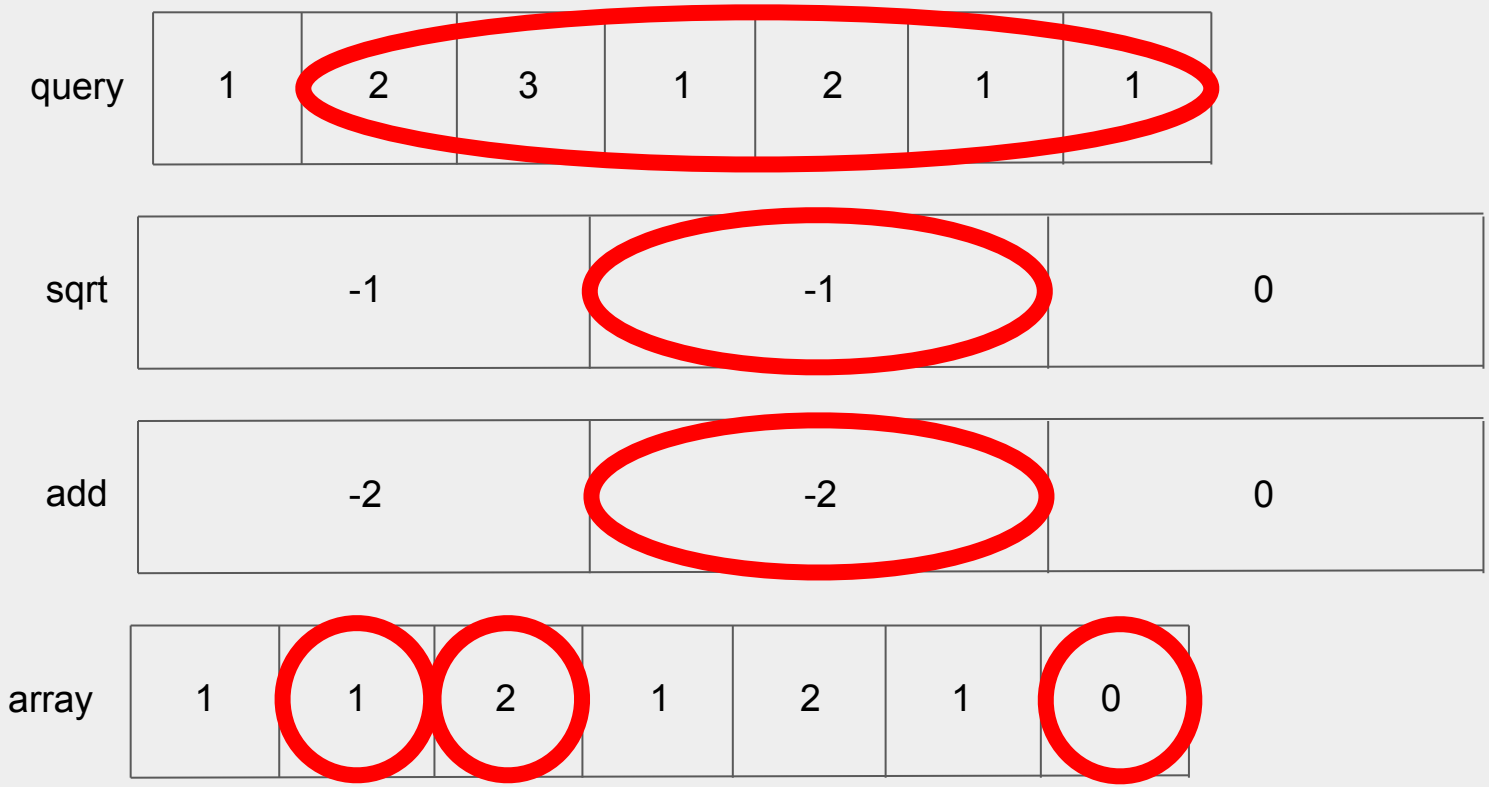
c = 3, a[2..7] -= 1



c = 3, a[2..7] -= 1



c = 3, a[2..7] -= 1



```
35. void add(int l, int r, int x, vector<Block> &sqrt_dec, vector<int> &a) {
36.     int res = 0;
37.     while (l <= r) {
38.         // beginning of the block - l = c * coef, r >= c * (coef + 1) - 1
39.         if (l % c == 0 && l + c - 1 <= r) {
40.             sqrt_dec[l / c].add += x;
41.             sqrt_dec[l / c].minn += x;
42.             l += c; // we skip block
43.         }
44.         else {
45.             a[l] += x;
46.             sqrt_dec[l / c].minn = min(sqrt_dec[l / c].minn, a[l]);
47.             l += 1;
48.         }
49.     }
50. }
```

```cpp
int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    vector<Block> sqrt_dec(n / c + 1);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
        sqrt_dec[i / c].minn = min(a[i], sqrt_dec[i / c].minn);
    }
    int q;
    cin >> q;
    for (int i = 0; i < q; i++) {
        int t, l, r;
        cin >> t >> l >> r;
        l--;
        r--;
        if (t) {
            cout << minn(l, r, sqrt_dec, a) << "\n";
        }
        else {
            add(l, r, -1, sqrt_dec, a);
        }
    }
    return 0;
}
```

Success #stdin #stdout 0.01s 5544KB

stdin

```
7
1 2 3 1 2 1 1
4
1 2 7
0 3 6
0 4 7
1 2 7
```

stdout

```
1
-1
```

Why we only subtract

If we increase we need to find honestly the minimal element in array!!!!!

(-1, 2, 3, 1, 0)

a[1..2] += 2 -> (1, 4, 3, 1, 0)

?a[1..5] -> 0

More

What else can we do:

Actually, almost anything—we'll delve into more details about root decomposition in the "hard group" session, but for now, let's focus on simpler things.

More

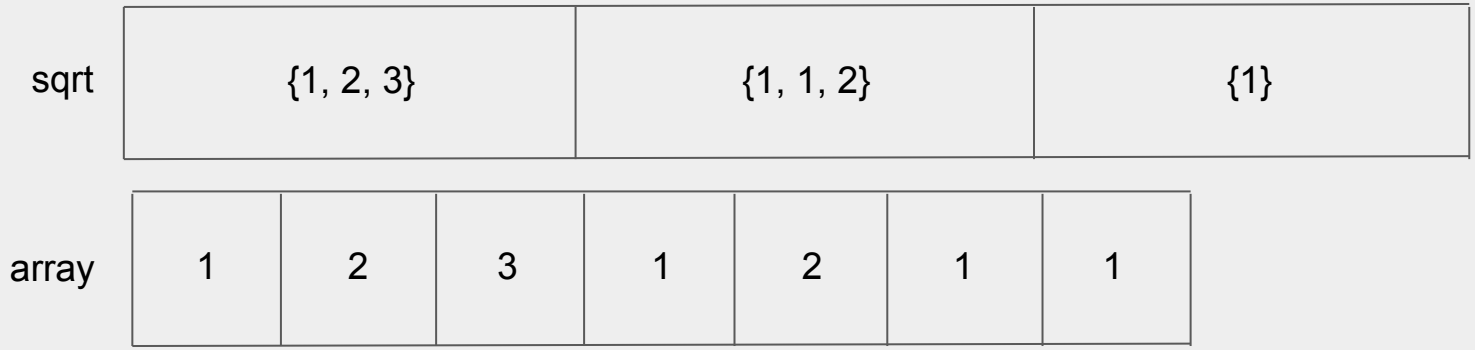
The number of elements less than X in a segment.

1. You need to be able to find amount of elements less than x in a segment.
2. Possibly, even perform updates, but this is quite complex (so we'll discuss this later).

Solution

Let's build a sorted array of numbers for the block.

c = 3



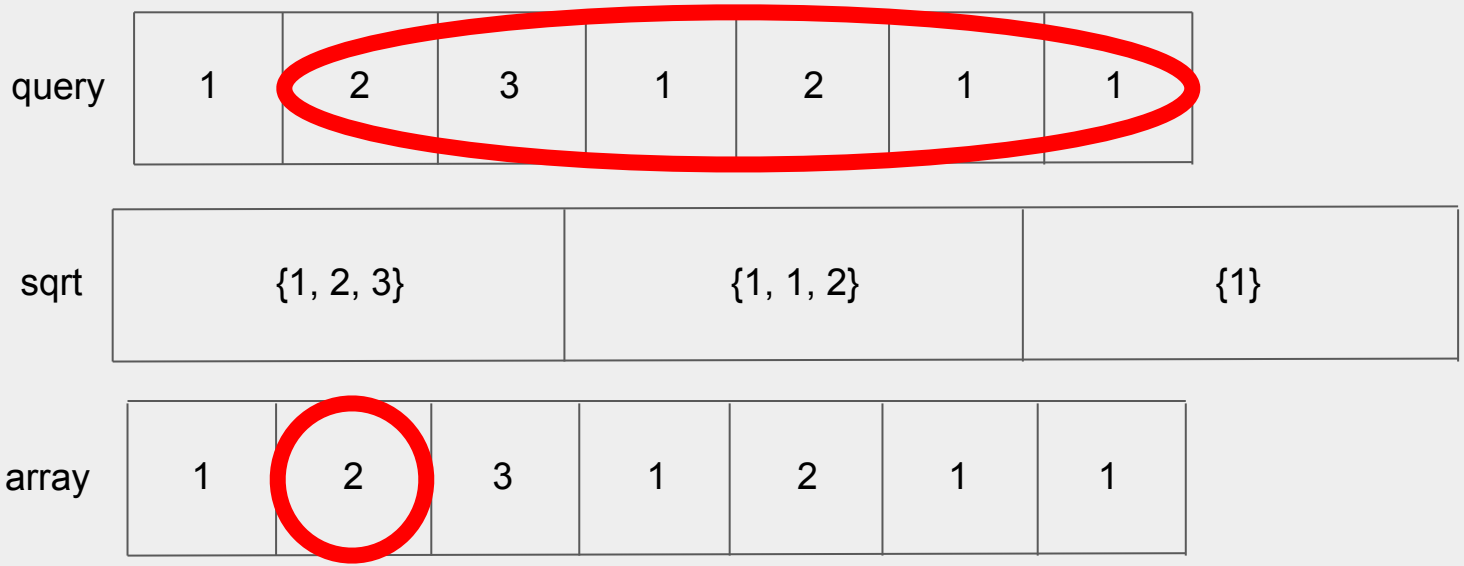
```
8.  struct Block {
9.      vector<int> elems;
10.     int add = 0;
```

```
vector<Block> sqrt_dec(n / c + 1);
for (int i = 0; i < n; i++) {
    cin >> a[i];
    sqrt_dec[i / c].elems.push_back(a[i]);
}
for (int i = 0; i < sqrt_dec.size(); i++) {
    sort(sqrt_dec[i].elems.begin(), sqrt_dec[i].elems.end());
}
```

Solution

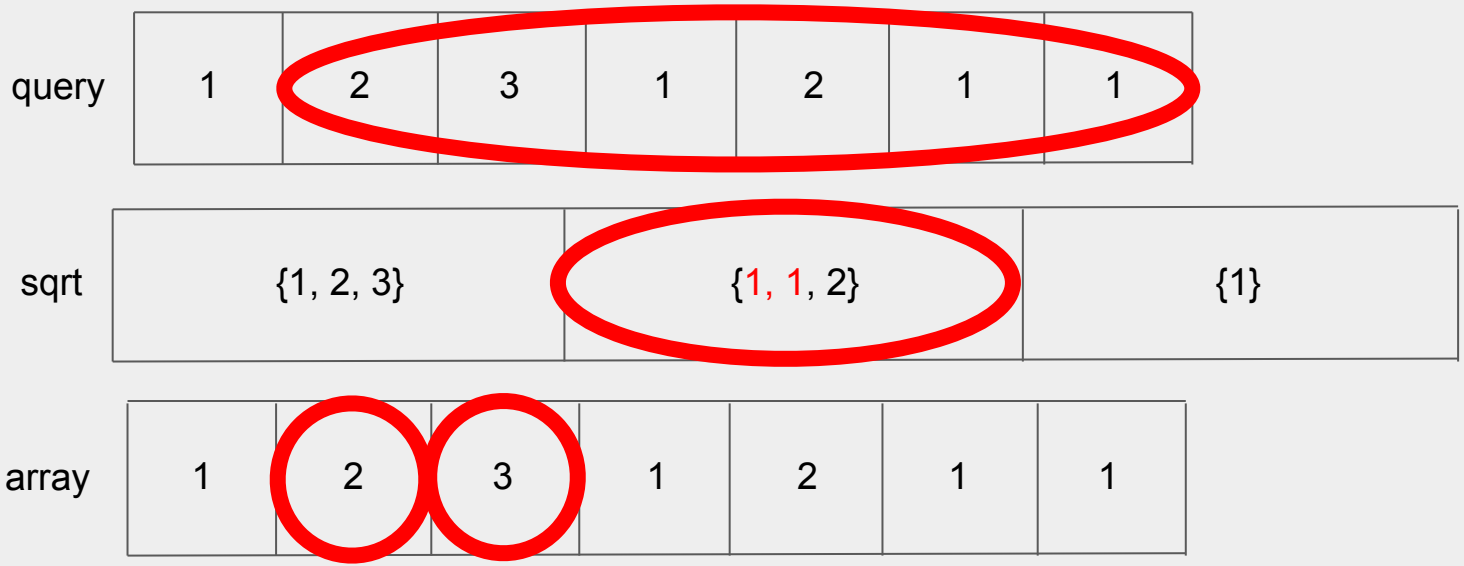
1. For a single element, we directly compare this element with X.
2. For the block, we make a binary search and use it to find the count of the required elements.

c = 3, X < 2, answer = 0

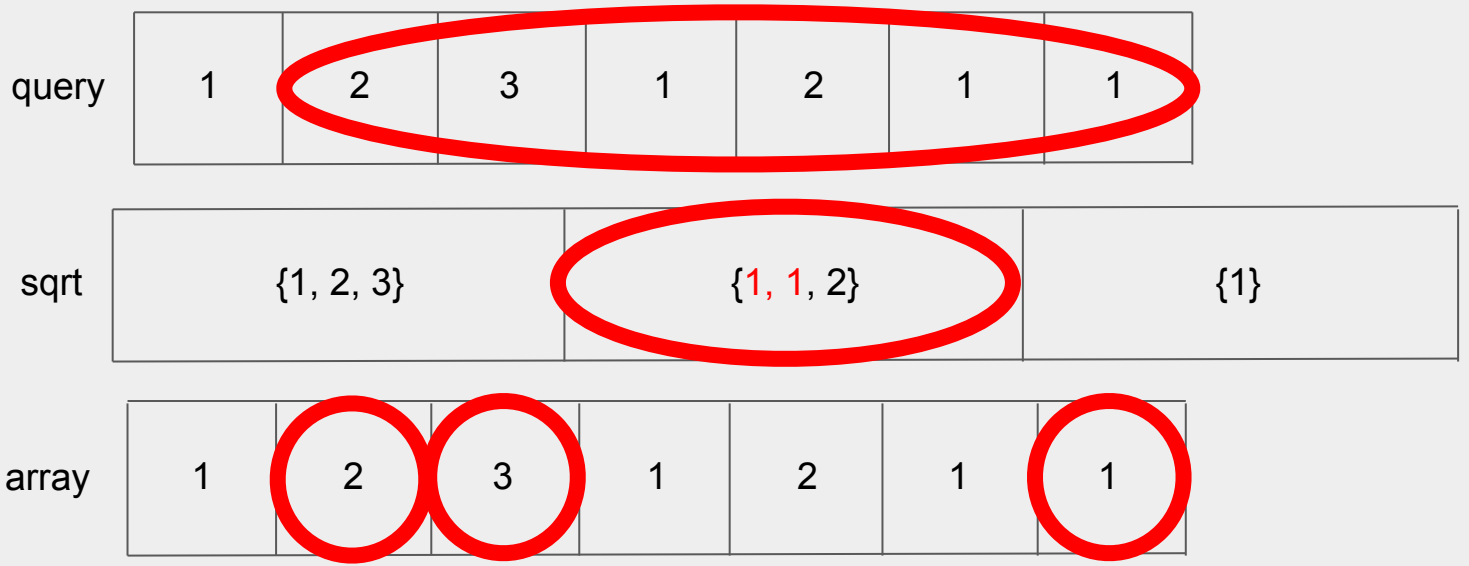


c = 3, X < 2, answer = 0

c = 3, X < 2, answer = 2



c = 3, X < 2, answer = 3



```
8.  struct Block {
9.      vector<int> elems;
10.     int add = 0;
11.
12.     int find(int x) const {
13.         auto it = std::upper_bound(elems.begin(), elems.end(), x);
14.         return std::distance(elems.begin(), it);
15.     }
16. };
```

```
19. int count(int l, int r, int x, const vector<Block> &sqrt_dec, const vector<int> &a) {
20.     int res = 1e9;
21.     while (l <= r) {
22.         // beginning of the block - l = c * coef1, r >= c * (coef1 + 1) - 1
23.         if (l % c == 0 && l + c - 1 <= r) {
24.             res = sqrt_dec[l / c].find(x);
25.             l += c; // we skip block
26.         }
27.         else {
28.             res += (a[l] >= x);
29.             l += 1;
30.         }
31.     }
32.     return res;
33. }
```

```
47.     int q;
48.     cin >> q;
49.     for (int i = 0; i < q; i++) {
50.         int l, r, x;
51.         cin >> l >> r >> x;
52.         l--;
53.         r--;
54.         cout << count(l, r, x, sqrt_dec, a) << "\n";
55.     }
56.     return 0;
57. }
```

Success #stdin #stdout 0.01s 5428KB

stdin

```
7
1 2 3 1 2 1 1
1
2 7 1
```

stdout

```
3
```

Asymptotic

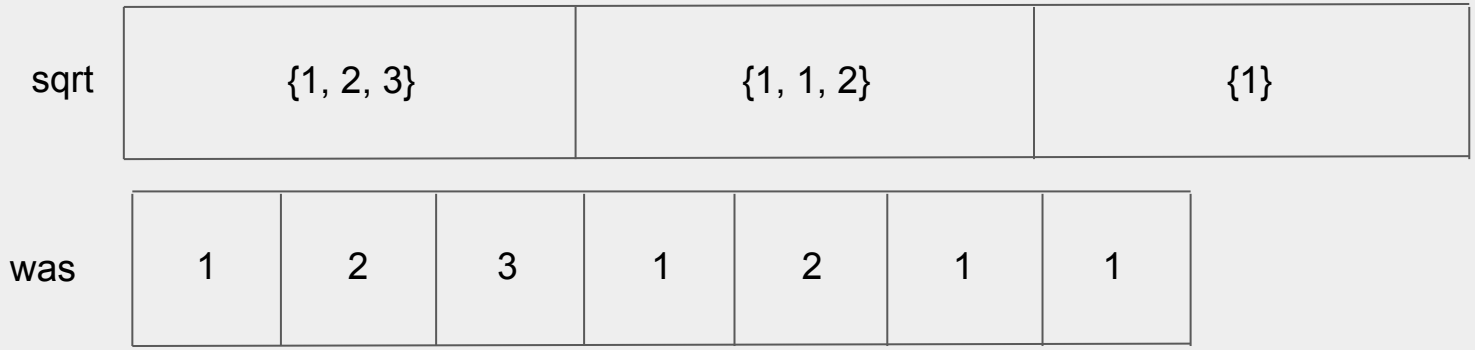
1. For a single element - O(1), thus, overall - O(c) per query.
2. For a block - O(log(c)), meaning O(n/c*log(c)) per query.

Harder

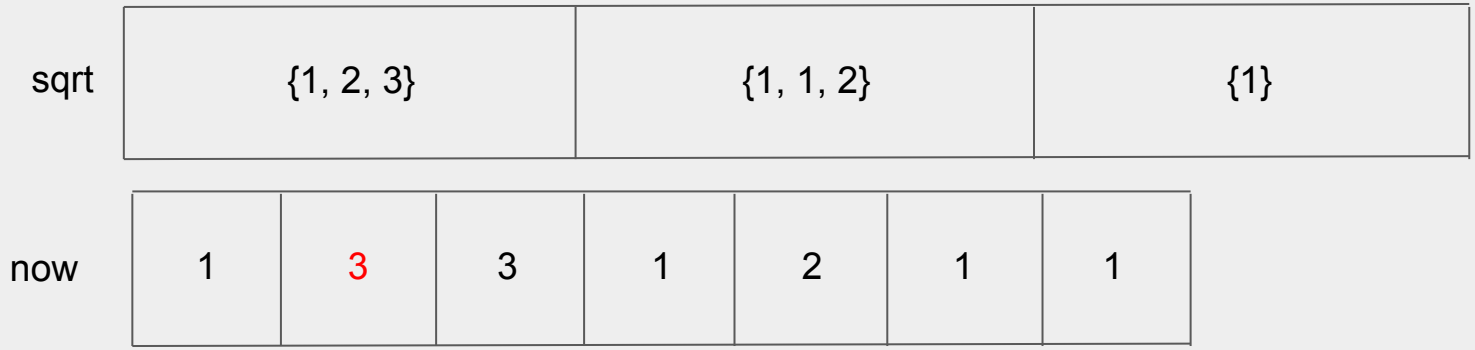
Let's add an update to the element.

To do this, first update the element itself, then re-sort the entire block to accommodate this change.

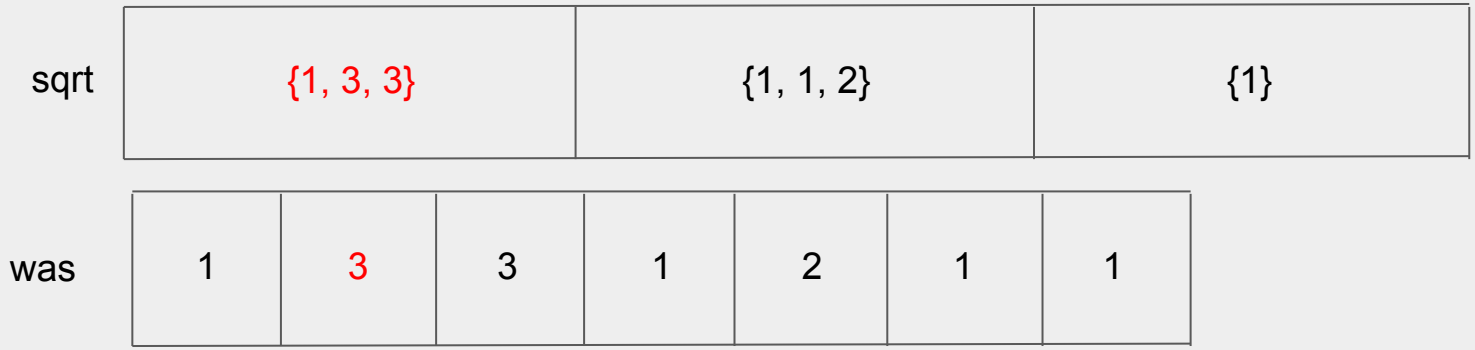
a[1] = 3



a[1] = 3



a[1] = 3



Asymptotic

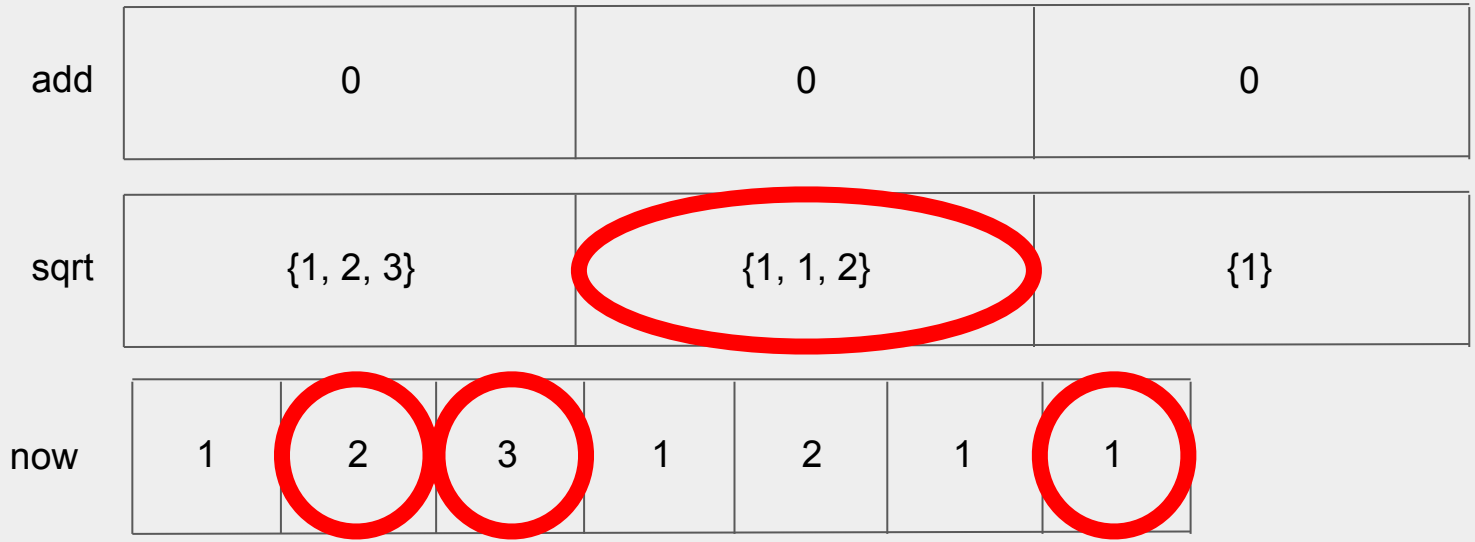
Resorting the block will operate in O(clog(c)) because its size is c.

Harder

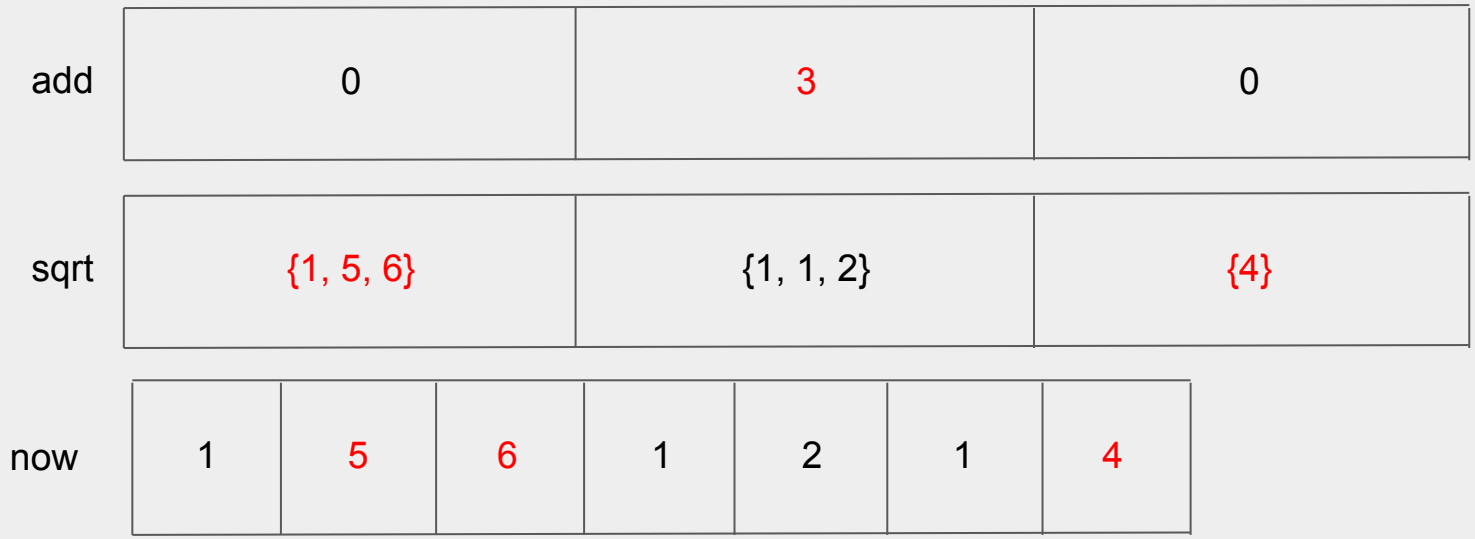
What if we want to update an entire segment?

1) Let's introduce an additional value 'add' into the root decomposition again.
2) In the case of searching for the number of elements in a block < X, perform a binary search considering X - add.

[1, 6] += 3



[1, 6] += 3

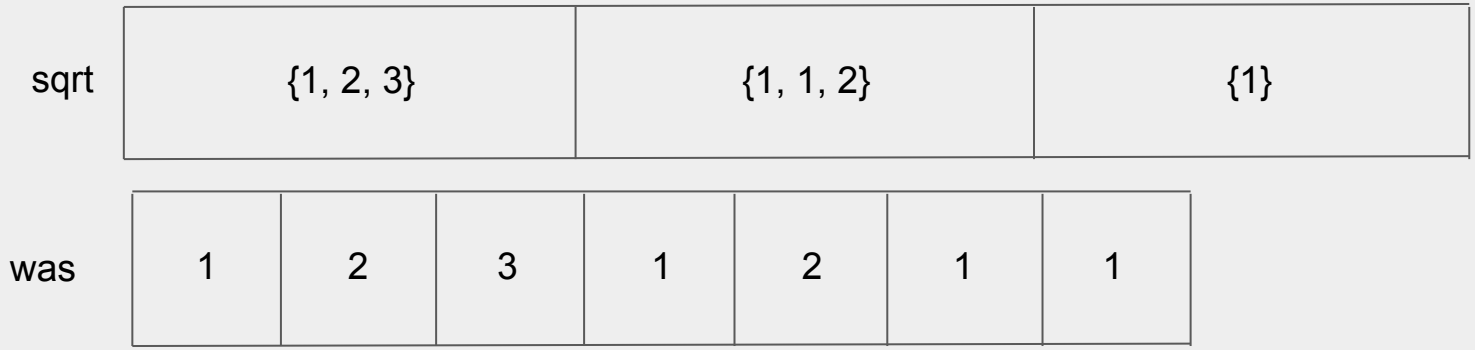


Problems

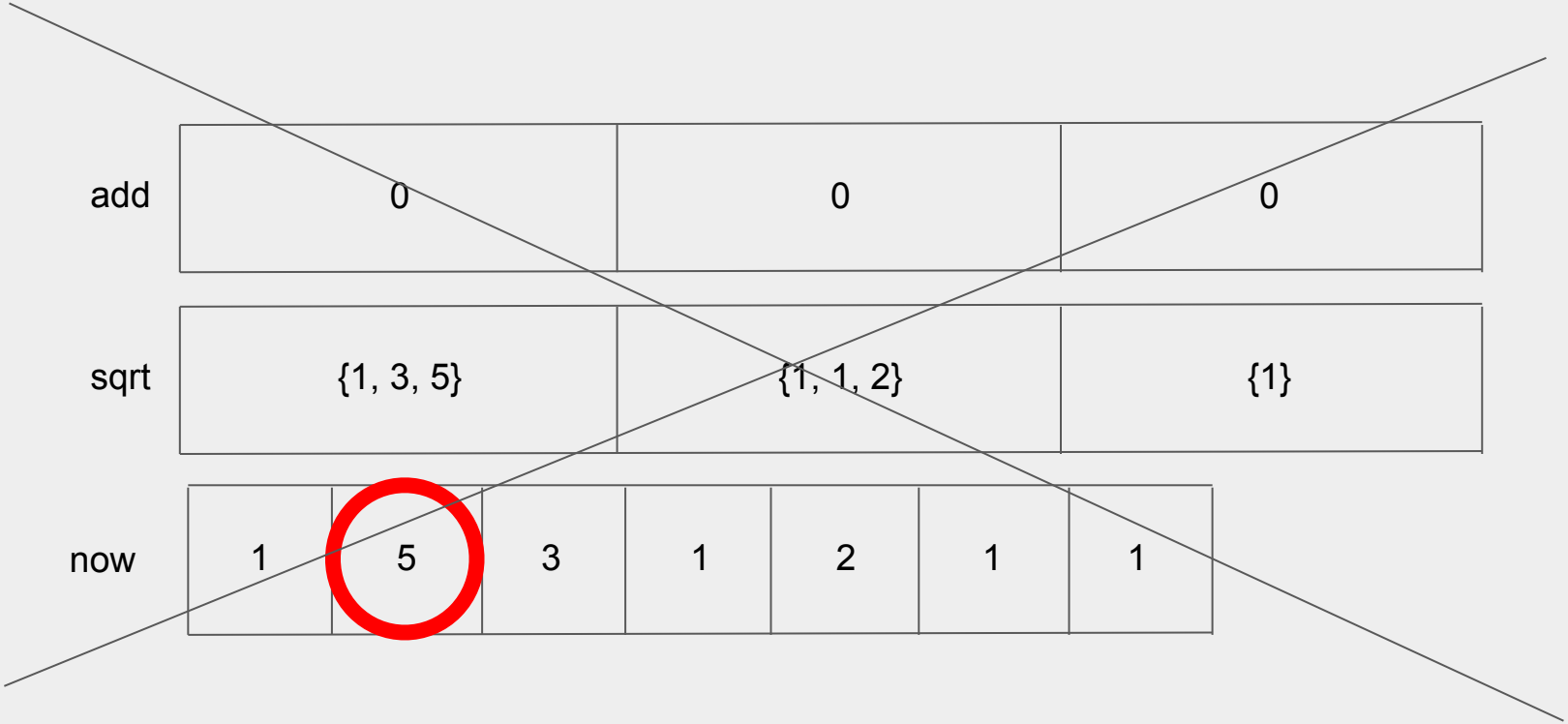
When updating elements, we only need to sort the block once, after all elements have been updated.

Therefore, we'll sort the block only in the last element.

[1, 6] += 3

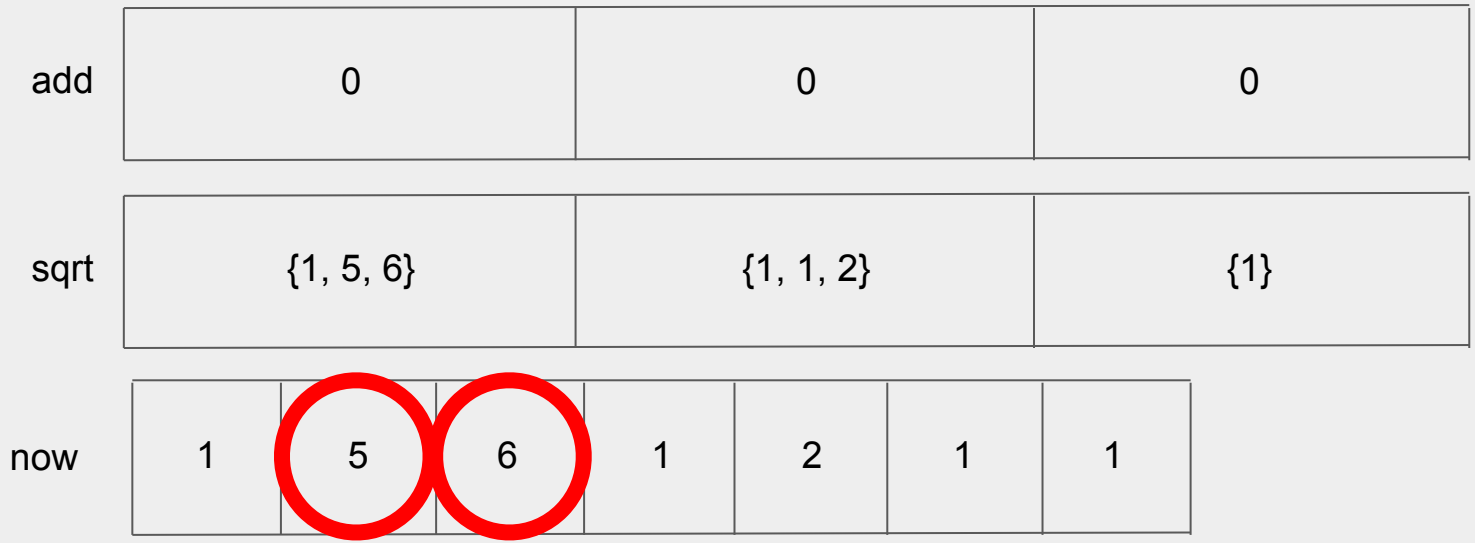


[1, 6] += 3



[1, 6] += 3

[1, 6] += 3



What is c?

Usually c = sqrt(n) (because we solve something like O(c) + O(n / c) -> the least), but

1. sqrt(n) is not precise, so better to make a const from it.
2. counting sqrt(n) is not O(1).
3. sqrt(n) is not cache-friendly
4. sometimes you have better values for c

What is c?

$O(c) + O(n/c * \log(c)) \rightarrow \min$

$O(c) = O(n/c * \log(c))$

$c = n/c * \log(c)$

$c^2 = n*\log(c)$

$c = \sqrt{n\log(c)} \approx \sqrt{n\log n}$

What is c?

O(c) + O(n/c * log(c)) - asymptotic of our solution

c = sqrt(n)

O(sqrt(n) + n / sqrt(n) * log(sqrt(n)) = O(sqrt(n) + sqrt(n) log(n)) = O(sqrt(n)log(n))

Not the best

What is c?

O(c) + O(n/c * log(c)) - asymptotic of our solution

c = sqrt(nlog(n))

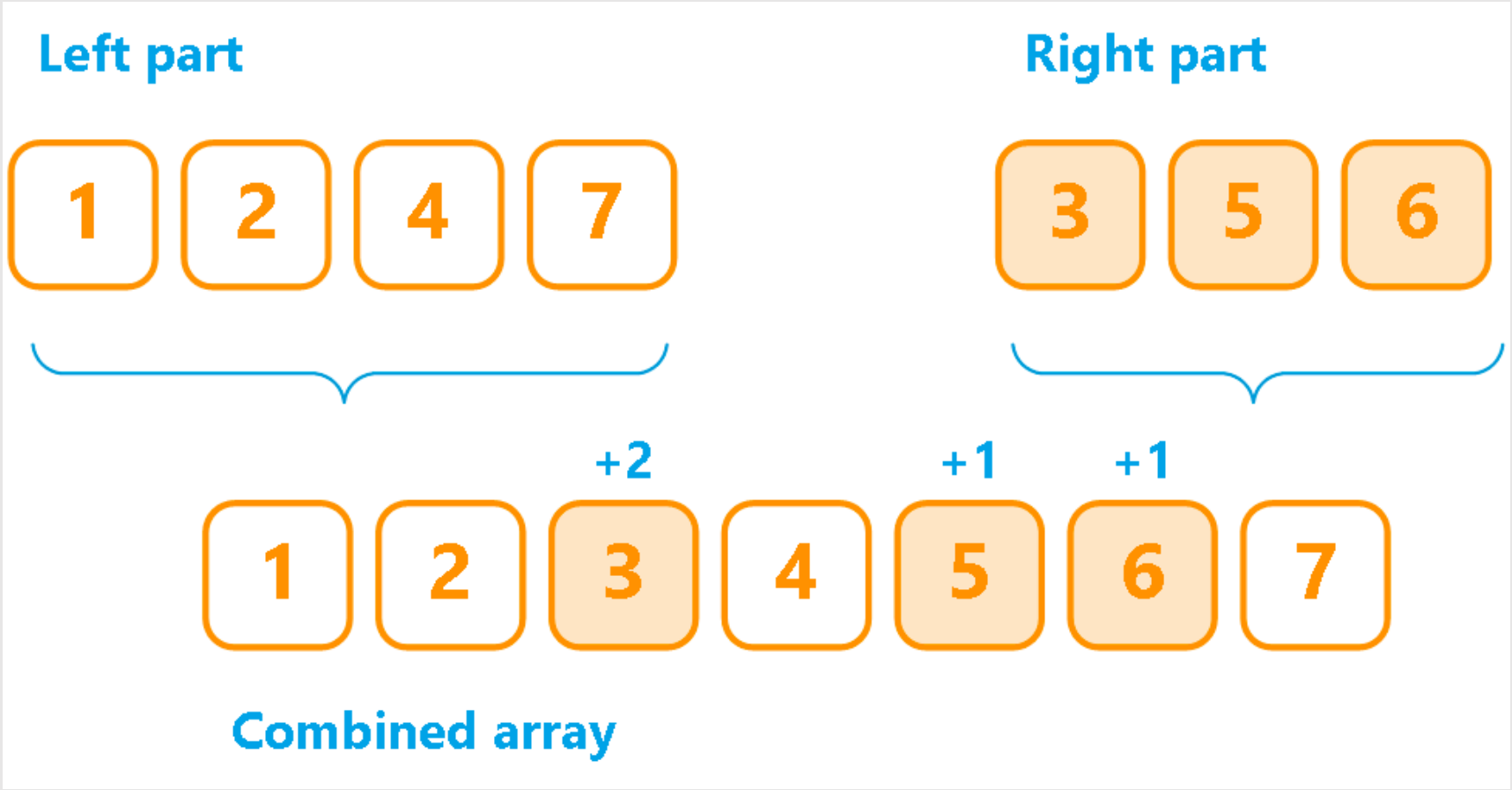
O(sqrt(nlogn) + n / sqrt(nlogn) * log(sqrt(nlogn)) =

O(sqrt(nlogn) + sqrt(n) / sqrt(log(n) * log(sqrt(nlogn))) =

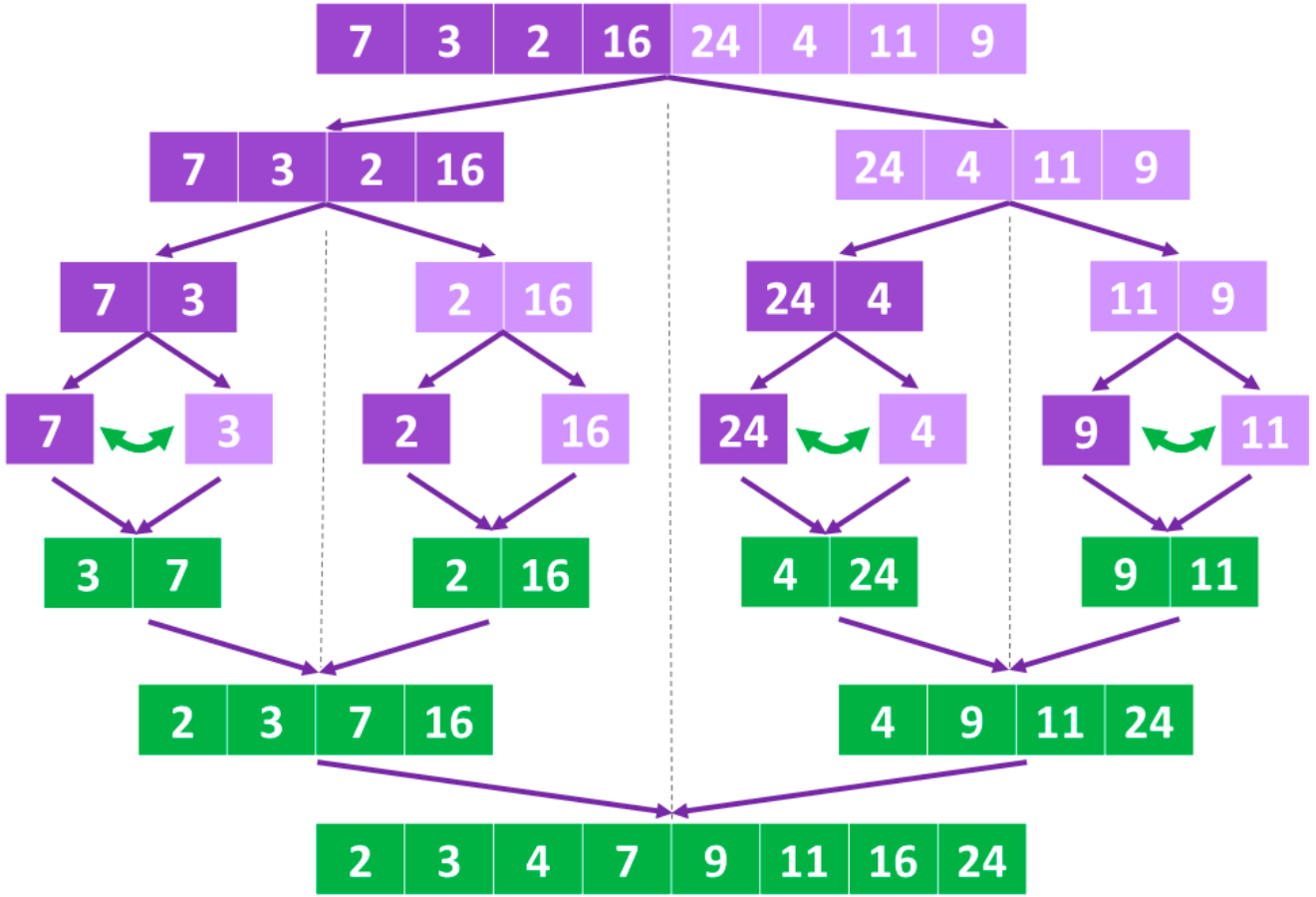
O(sqrt(nlog(n)))

Segment tree

Let's now come up with another algorithm to solve segment problems. We've already solved some strange problems using merge-sort, for instance, we used it to find the number of inversions in specific segments.


Left part
Right part
1
2
4
7
3
5
6
+2
+1
+1
1
2
3
4
5
6
7
Combined array

Merge Sort



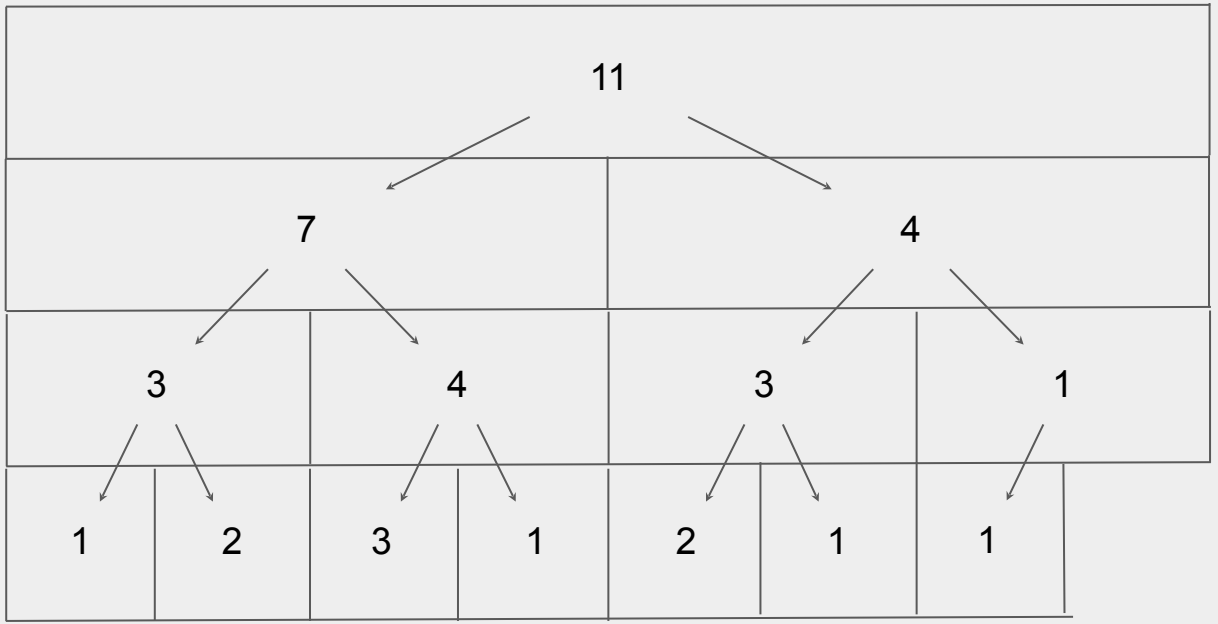
Step 1:
Split sub-lists in two until you reach pair of values.

Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Segment tree

Let's call a segment tree a structure that stores answers for each segment in the merge-sort.

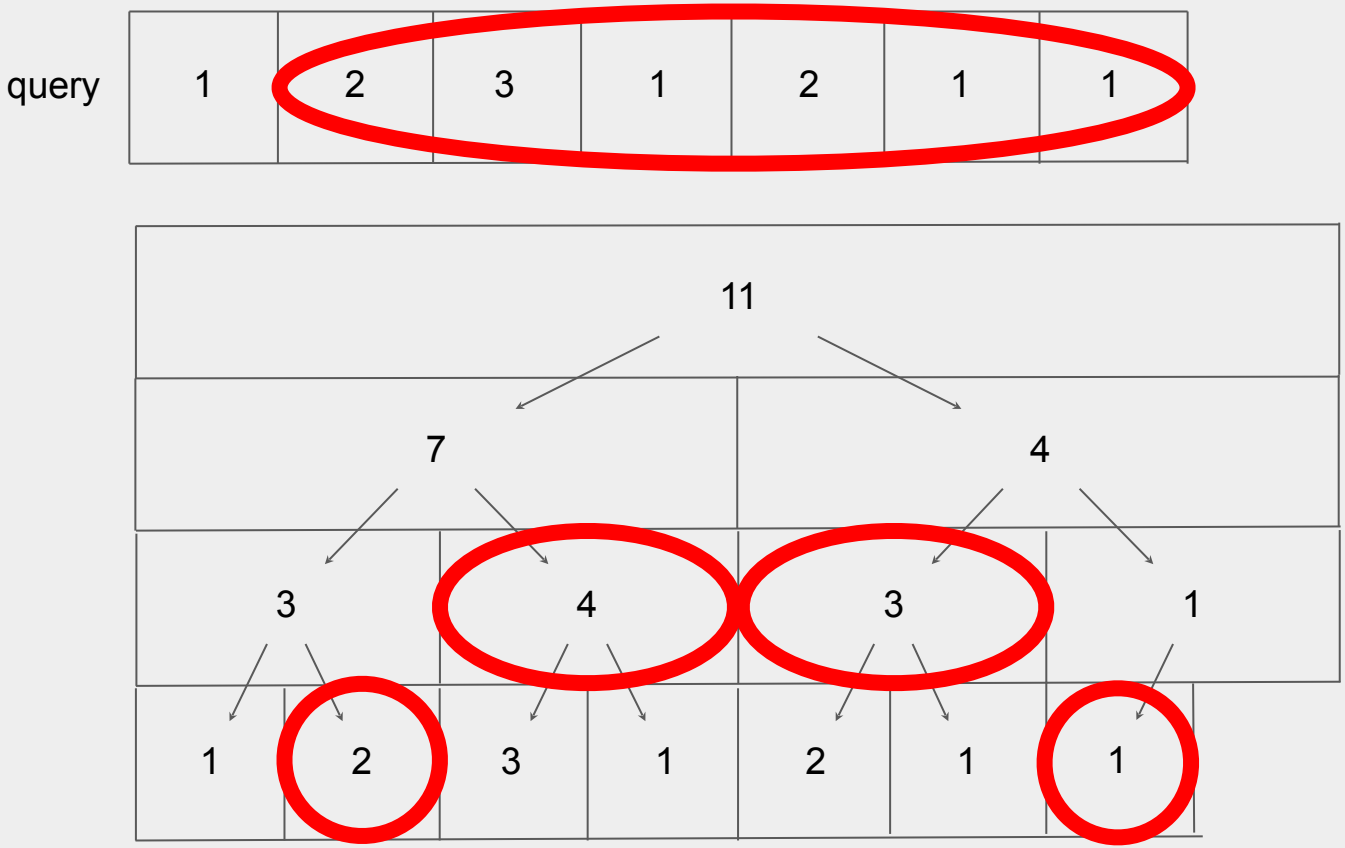

11
7
4
3
4
3
1
1
2
3
1
2
1
1

```
6.  struct Vertex {
7.      int sum = 0;
8.      int add = 0;
9.  };
```

```
24.  // build segment tree for range [begin, end)
25.  void build(int now, int begin, int end, vector<Vertex> &seg_tree, vector<int> &a) {
26.      if (begin == end - 1) {
27.          seg_tree[now].sum = a[begin];
28.      } else {
29.          int mid = (begin + end) / 2;
30.          build(now * 2, begin, mid, seg_tree, a);
31.          build(now * 2 + 1, mid, end, seg_tree, a);
32.          seg_tree[now].sum = seg_tree[now * 2].sum + seg_tree[now * 2 + 1].sum;
33.      }
34.  }
```

Segment tree

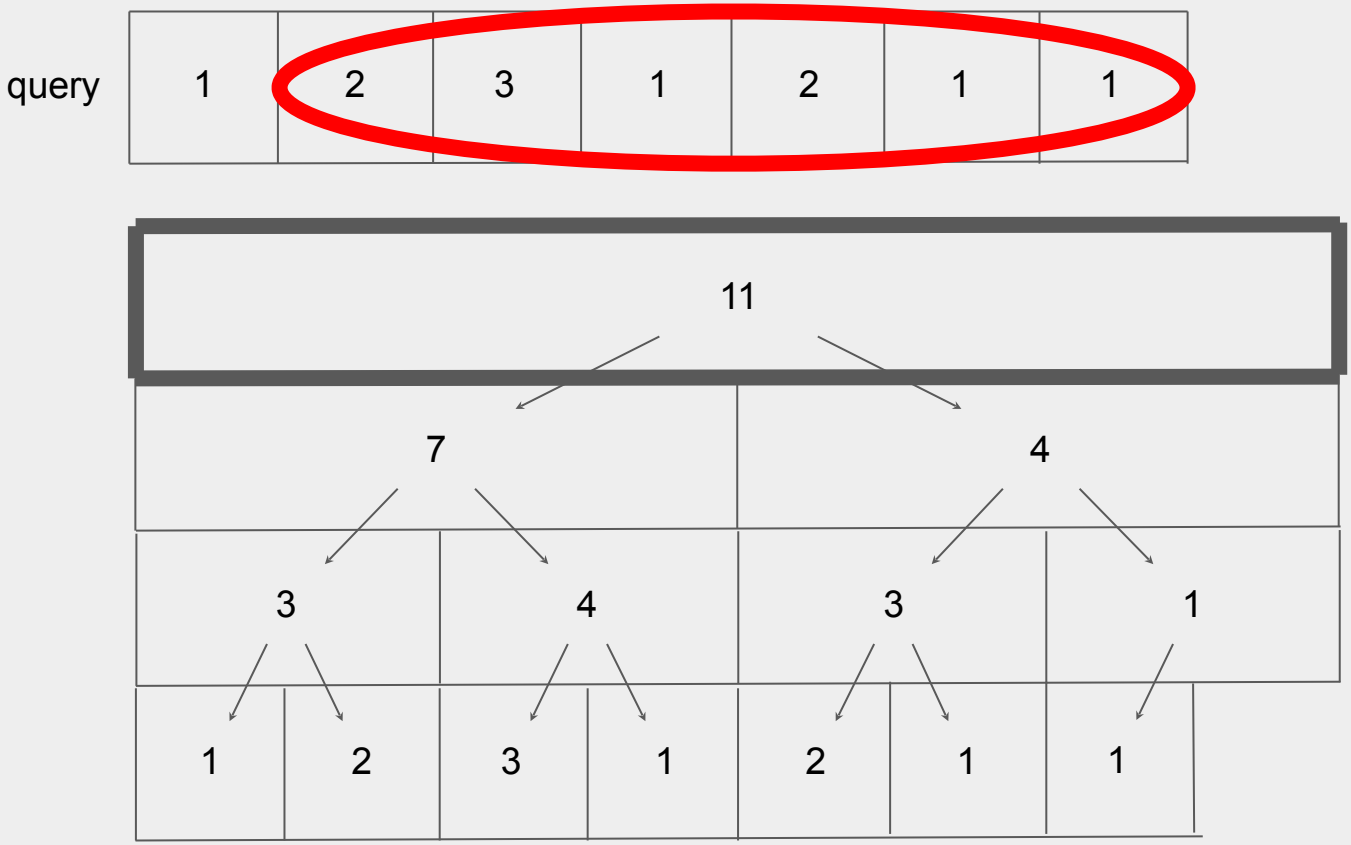
Any segment can be uniquely represented as a set of non-overlapping such segments.

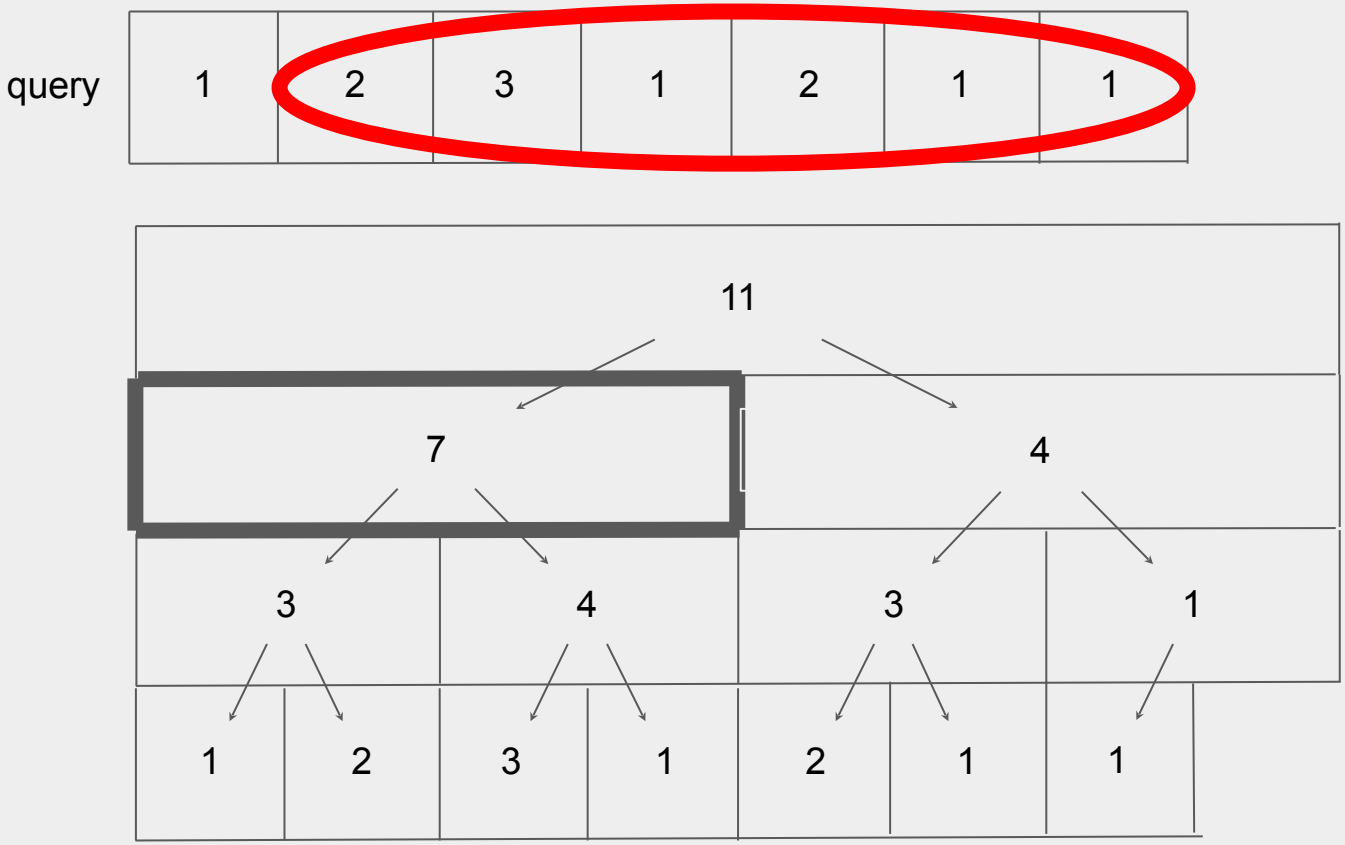

query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1

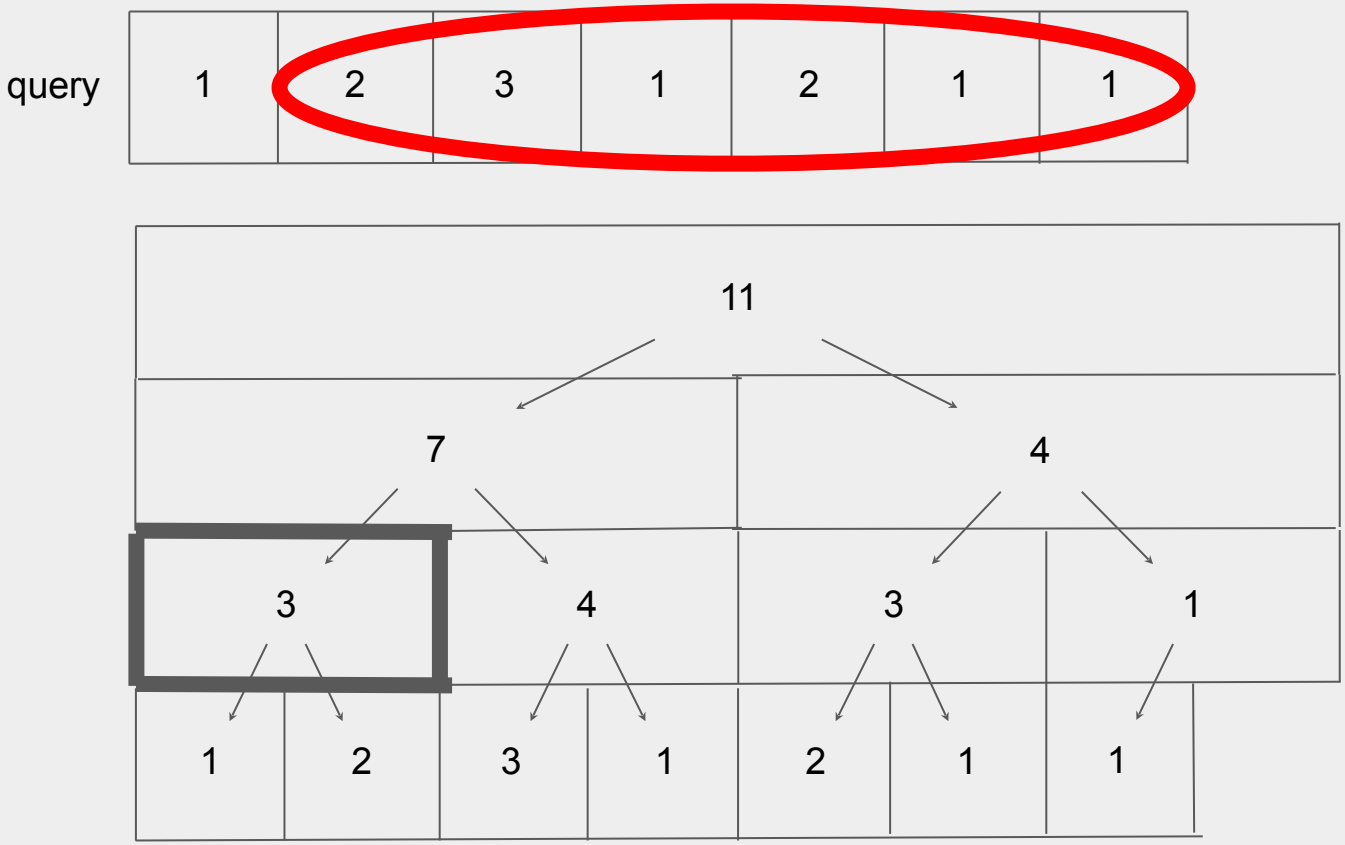
How?

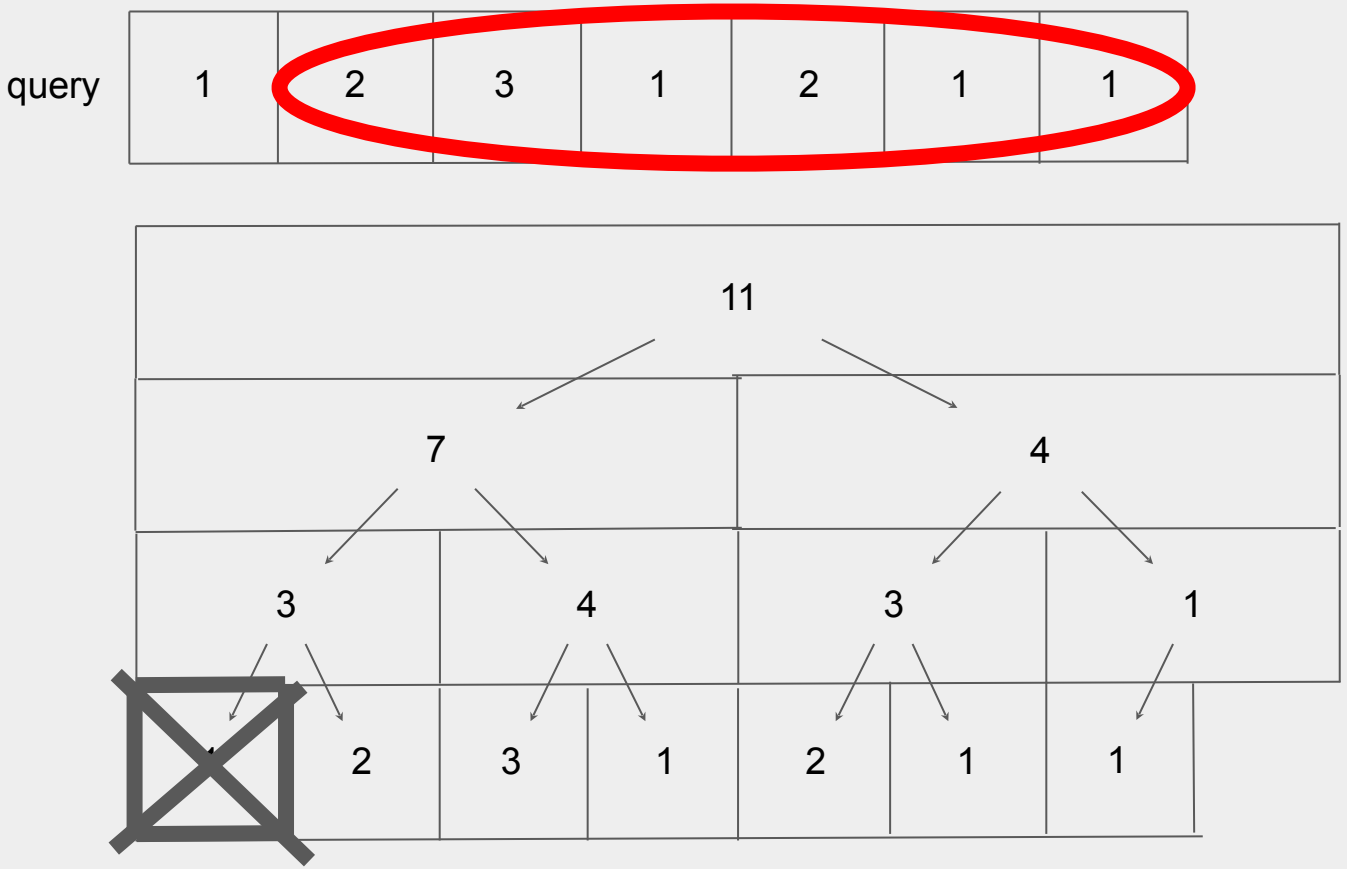
How can this be done in detail?

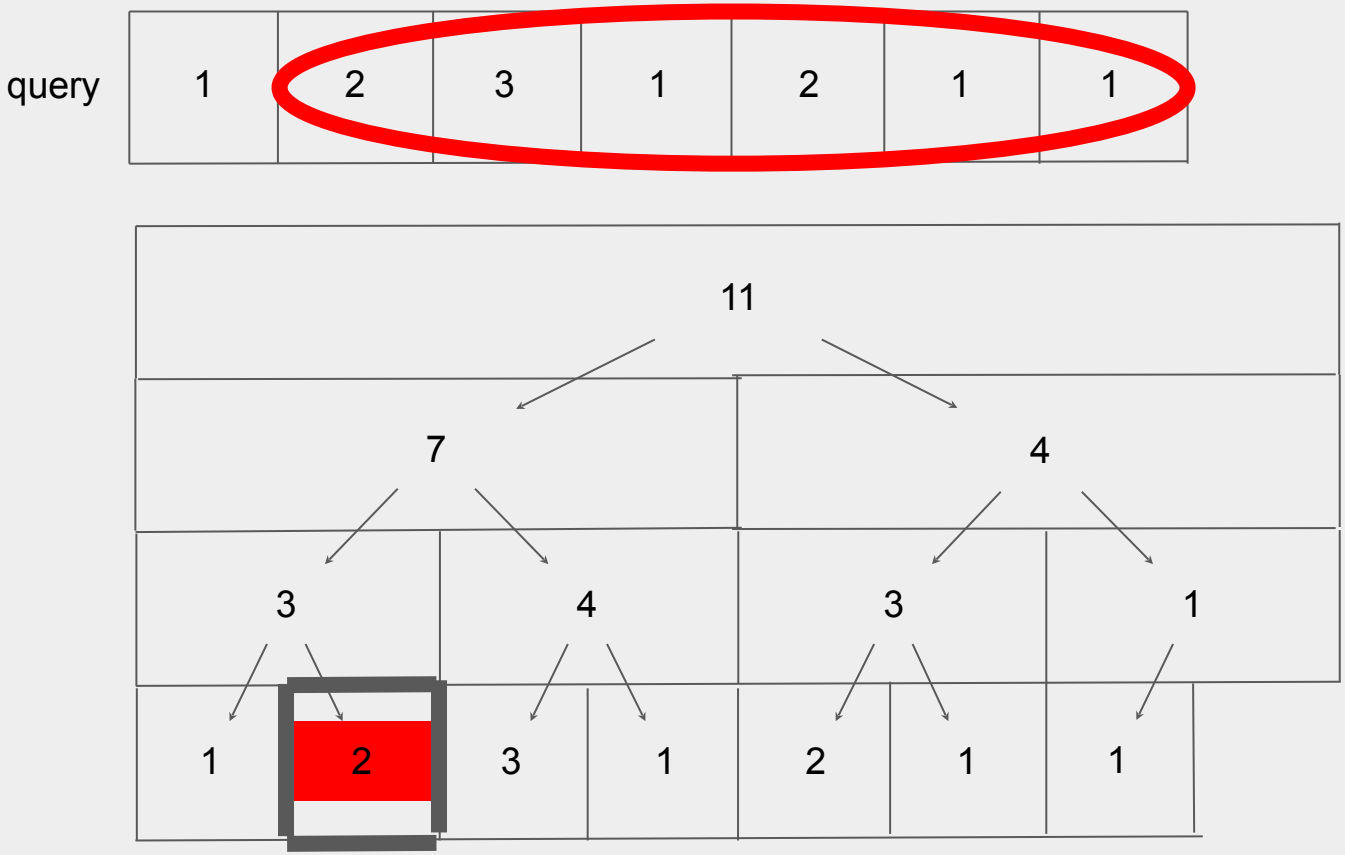
Let's perform a recursive descent in which we take a node and a segment, and descend until we find a segment that is either completely inside or completely outside.

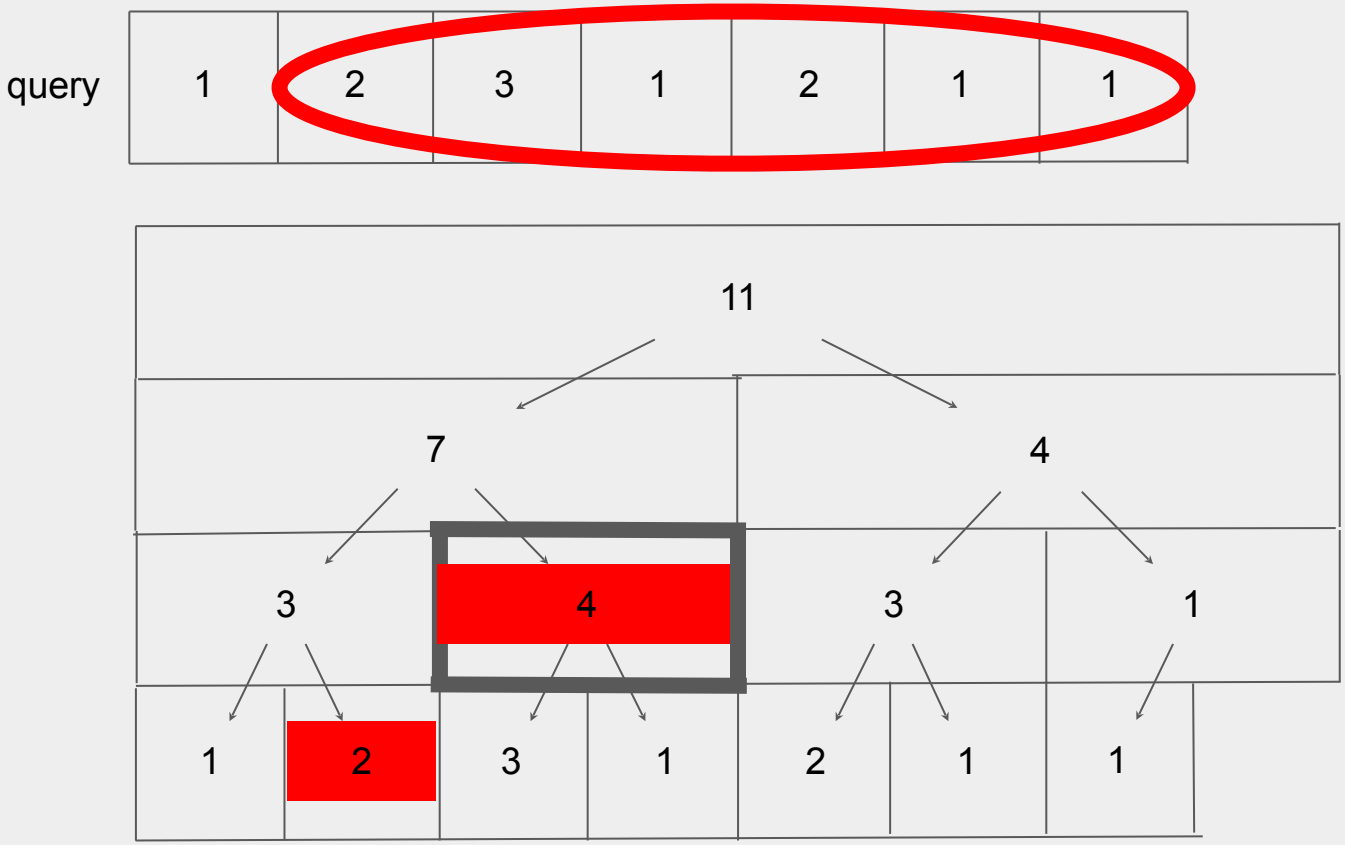

query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1

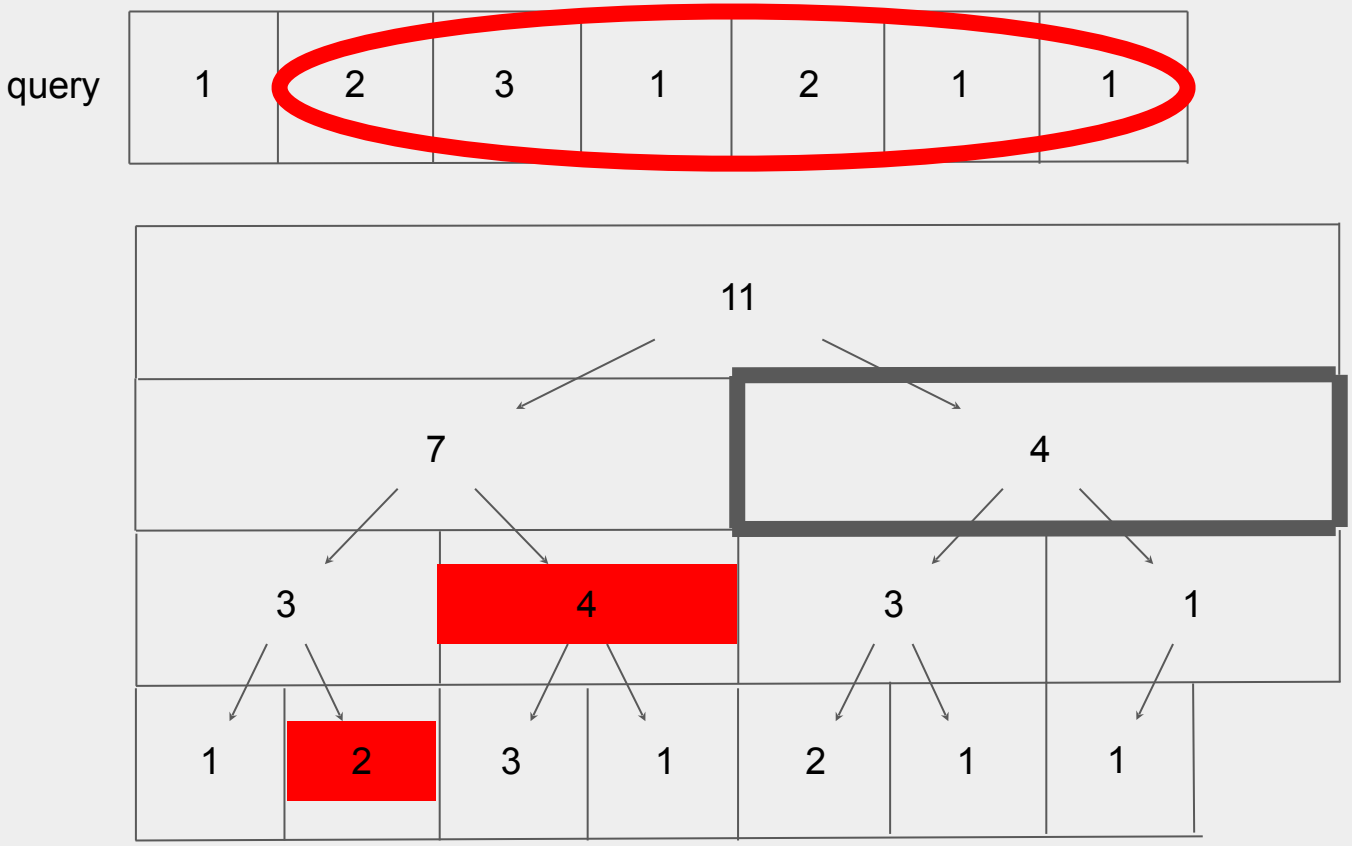

query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1

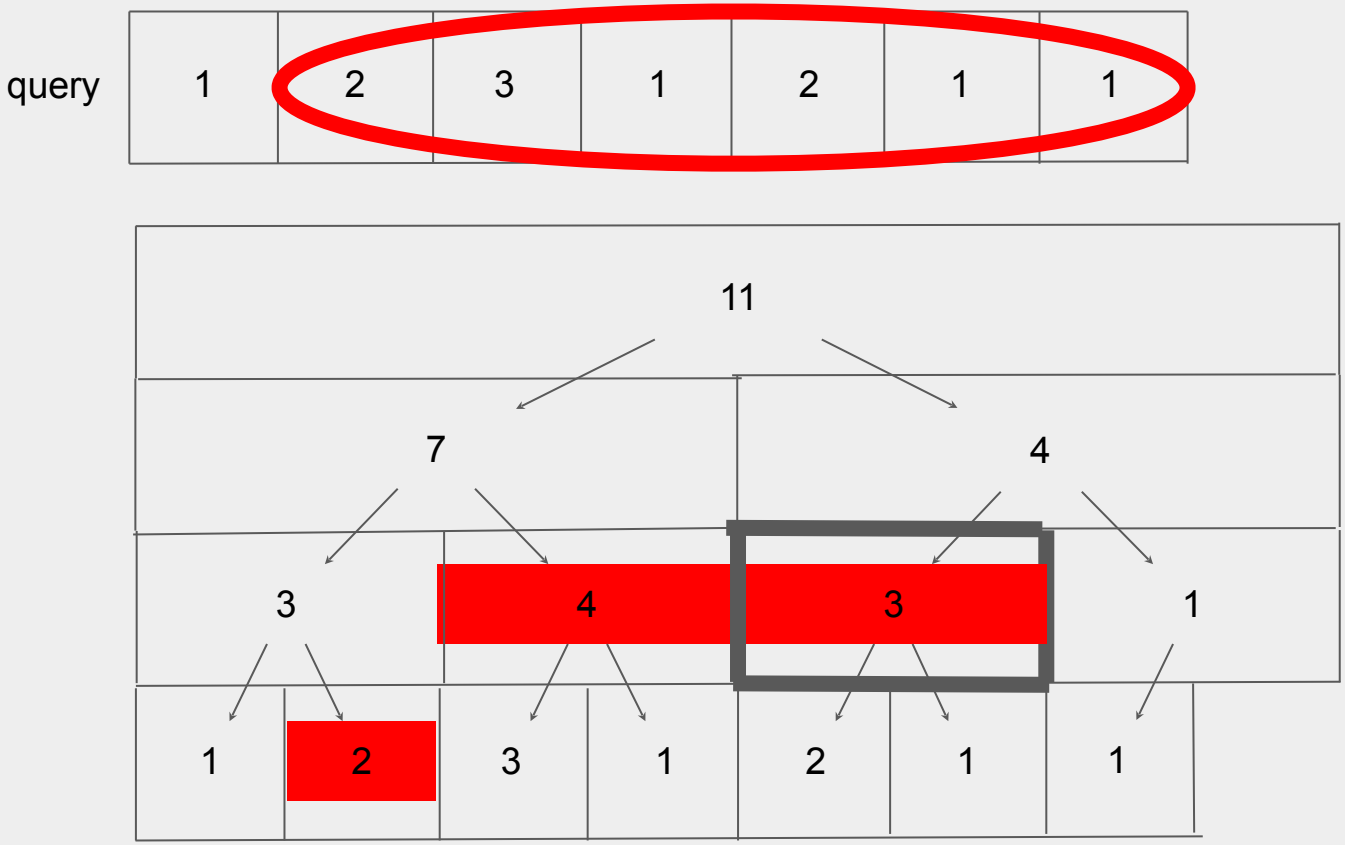

query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1

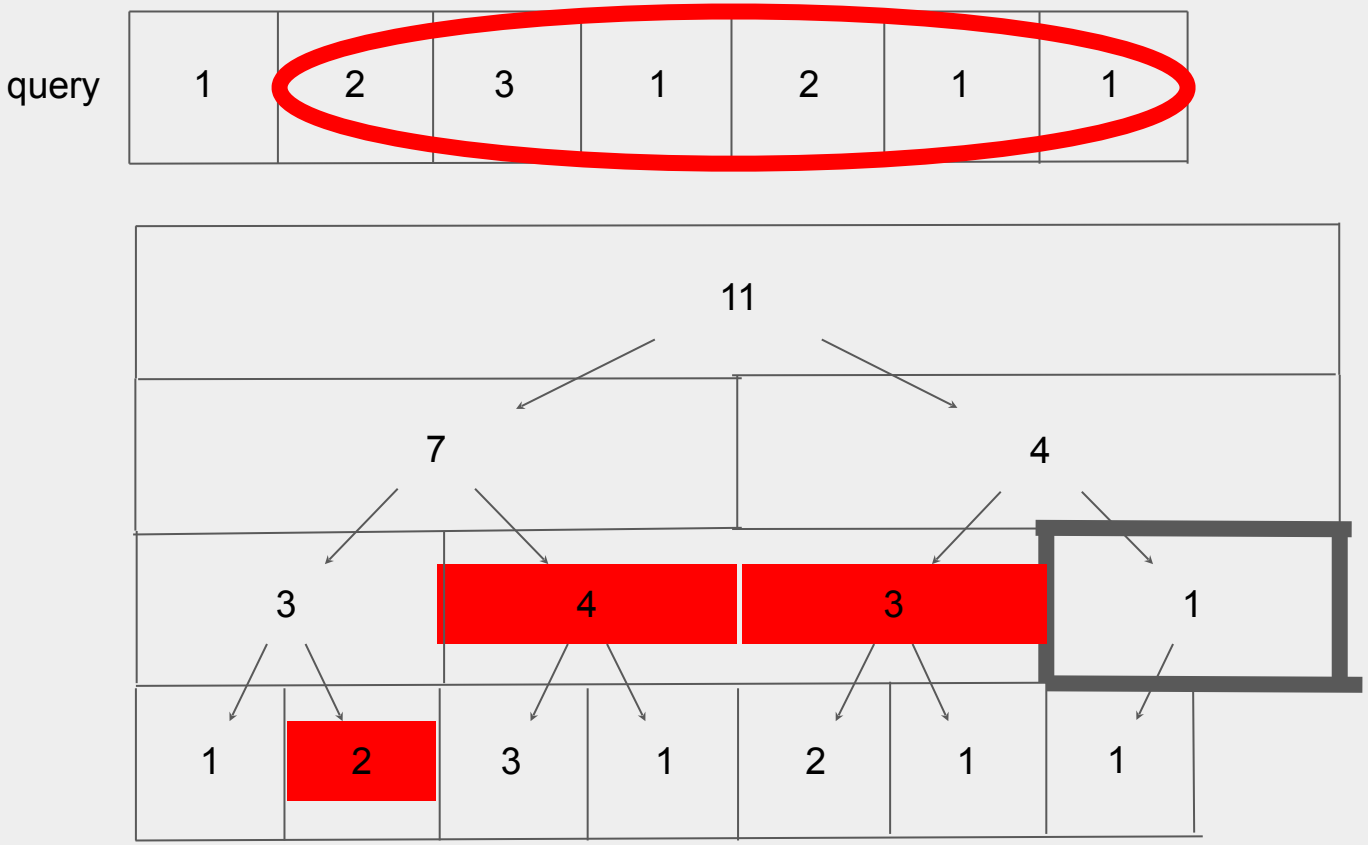

query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
2
3
1
2
1
1

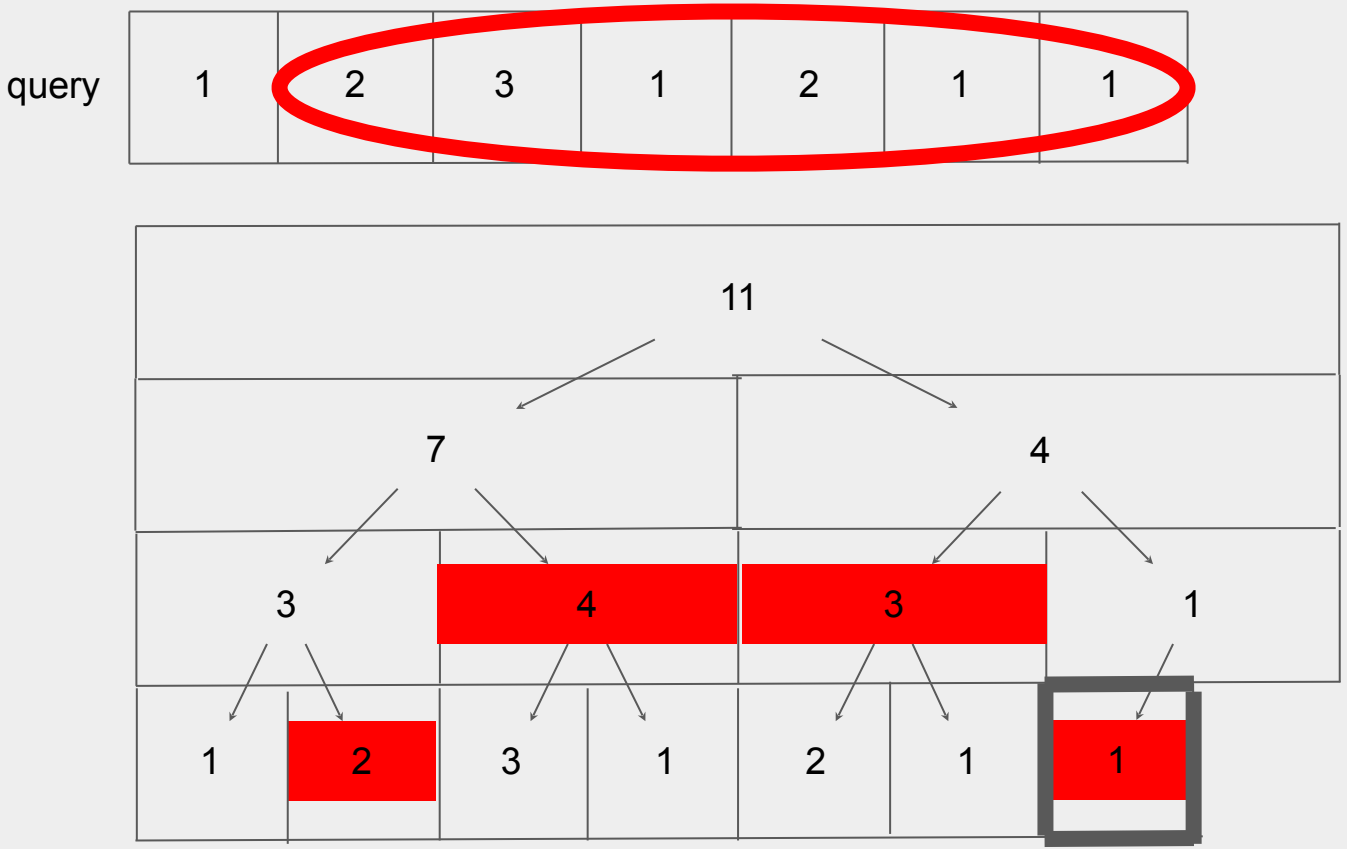

query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1


query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1


query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1


query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1


query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1


query
1
2
3
1
2
1
1
11
7
4
3
4
3
1
1
2
3
1
2
1
1

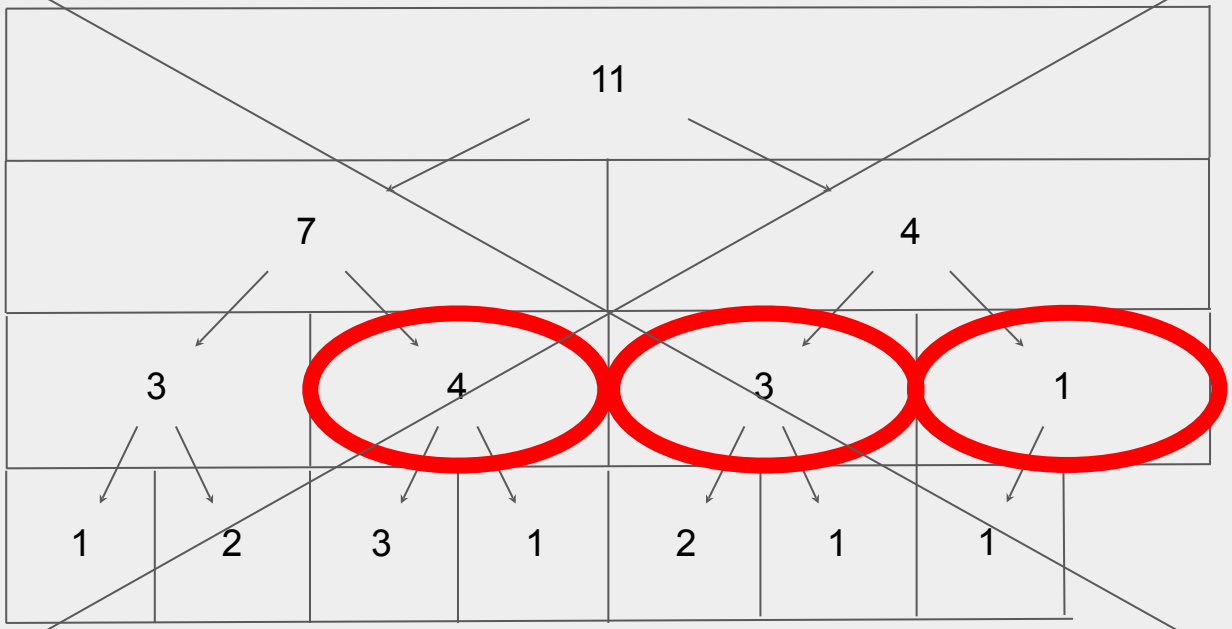
```
35.
36. int sumSeg(int now, int left, int right, int begin, int end, vector<Vertex> &seg_tree, const vecto
    r<int> &a) {
37.     if (end <= left || right <= begin) {
38.         return 0;
39.     }
40.     if (begin <= left && right <= end) {
41.         return seg_tree[now].sum;
42.     }
43.     push(now, left, right, seg_tree);
44.     int middle = (left + right) / 2;
45.     int x = sumSeg(now * 2, left, middle, begin, end, seg_tree, a);
46.     int y = sumSeg(now * 2 + 1, middle, right, begin, end, seg_tree, a);
47.     return x + y;
48. }
```

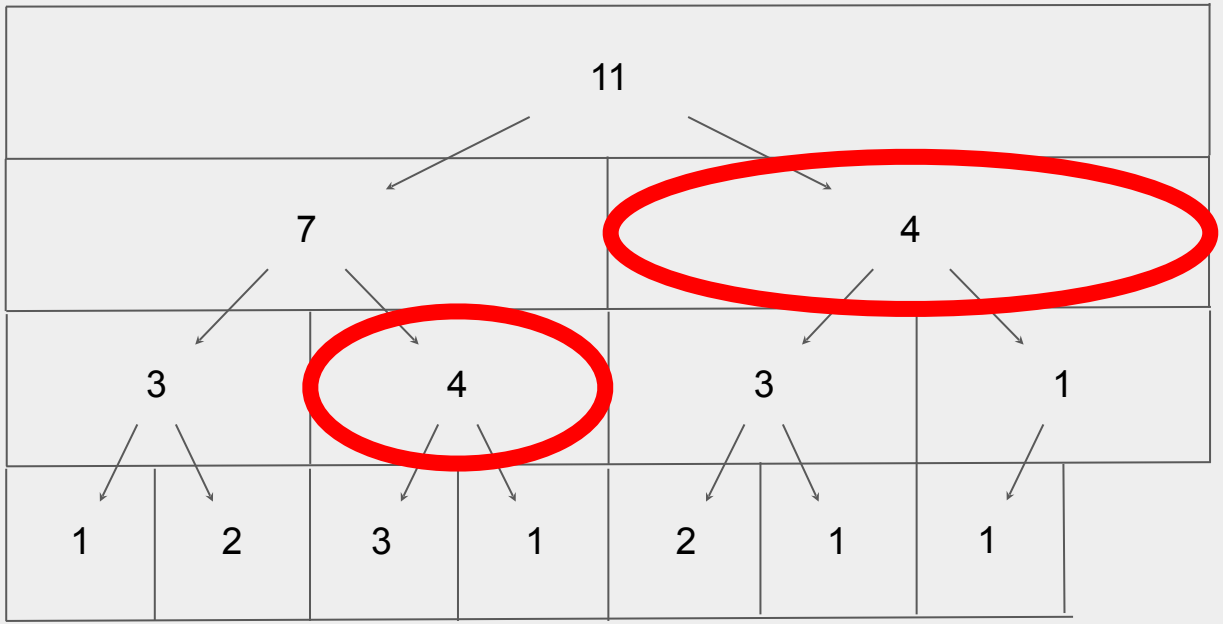
Asymptotic

The depth of the segment tree is O(log(n)).

At each level of the segment tree, we can take no more than two nodes (otherwise, some could be merged).

Amount of vertices in seg tree - 2n


11
7
4
3
4
3
1
1
2
3
1
2
1
1

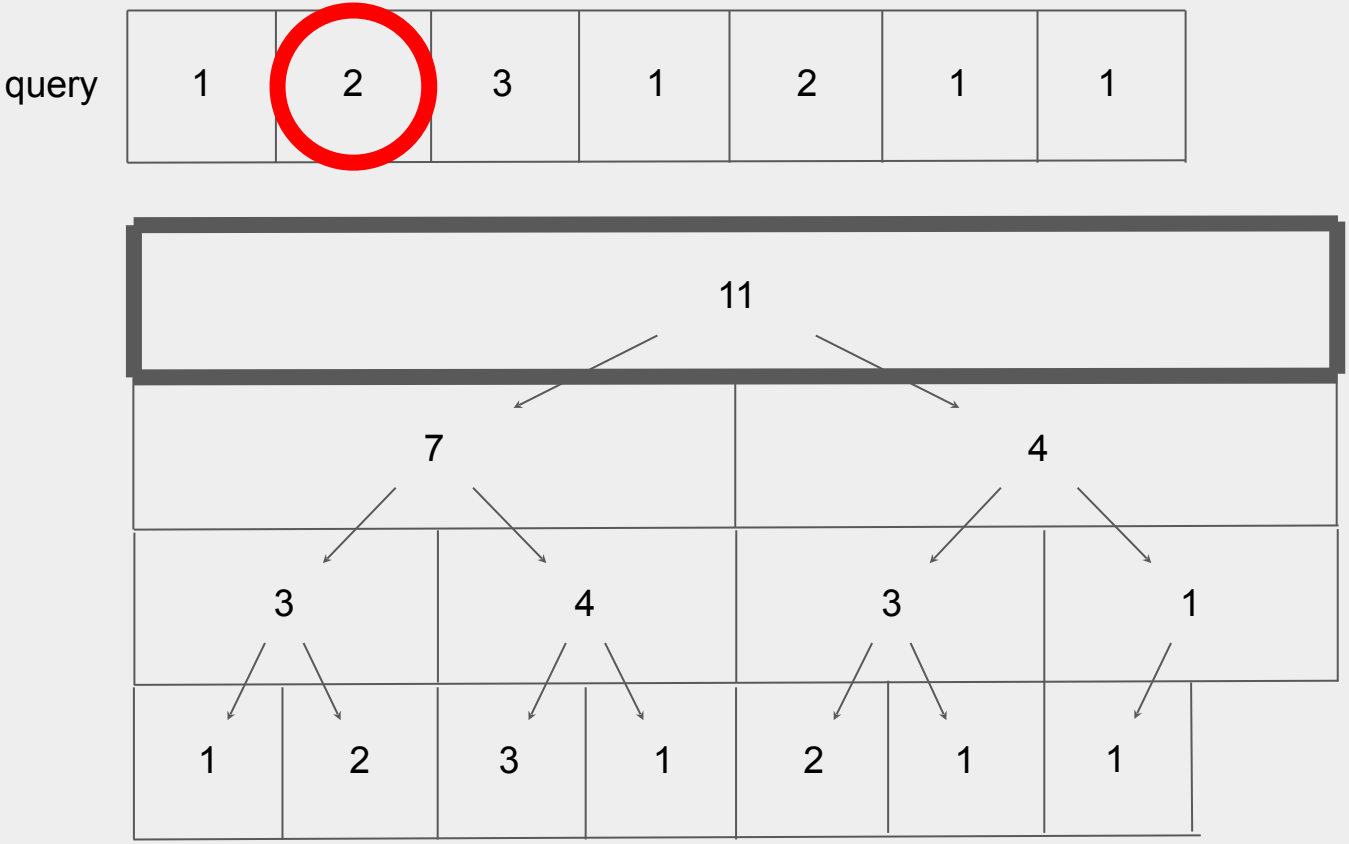

11
7
4
3
4
3
1
1
2
3
1
2
1
1

How?

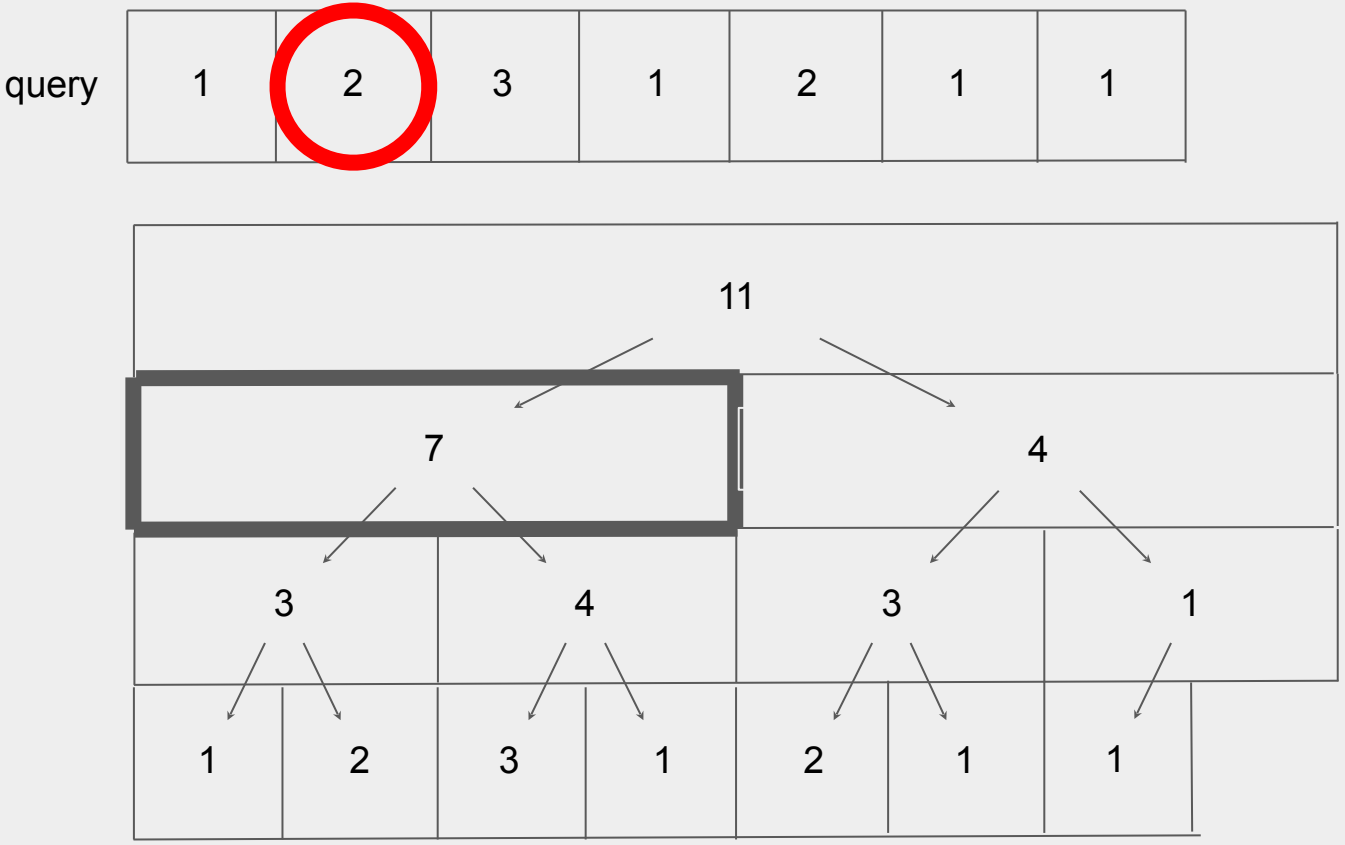
How to update an element?

Let's recursively descend to the required element and then update all its ancestors.

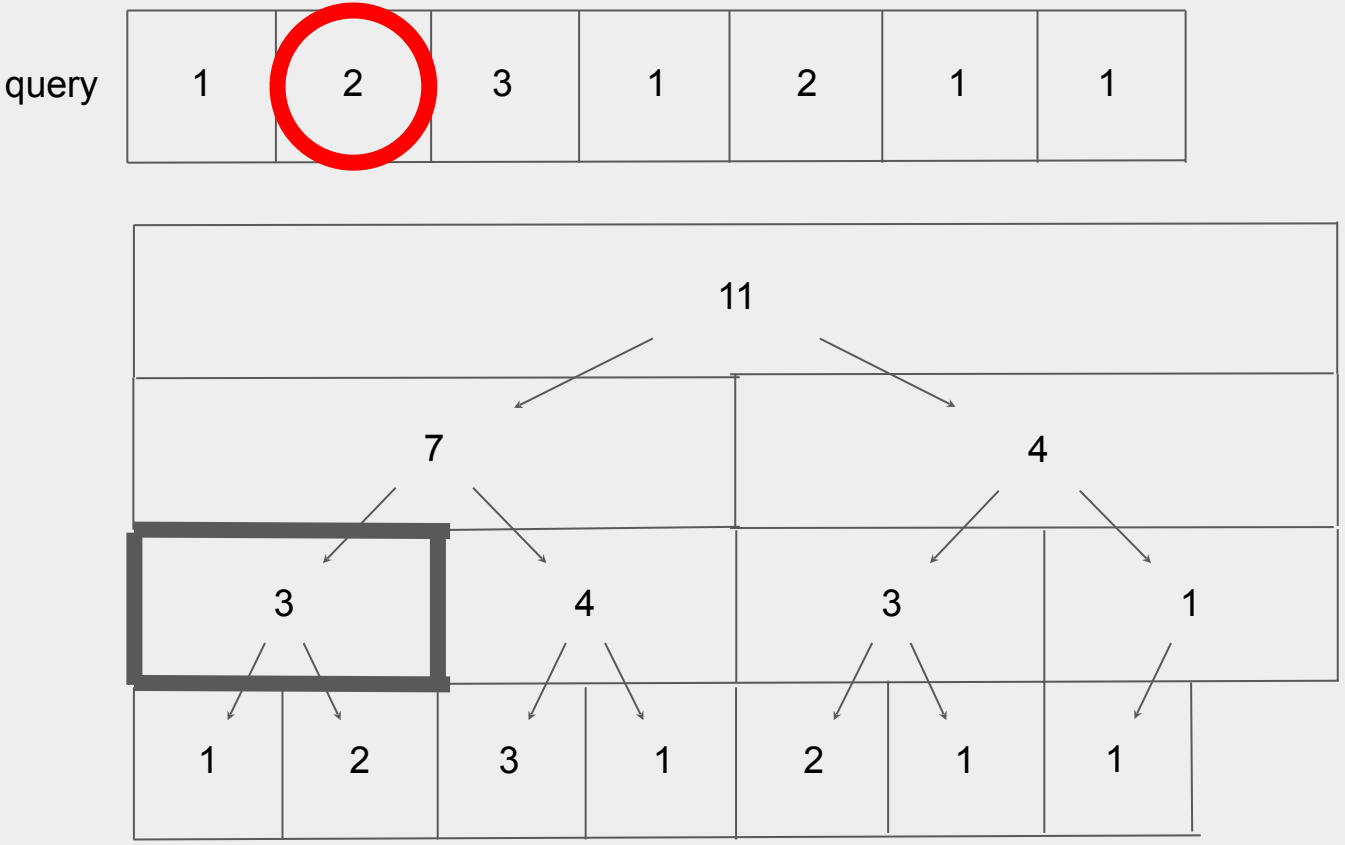
a[1] += 3



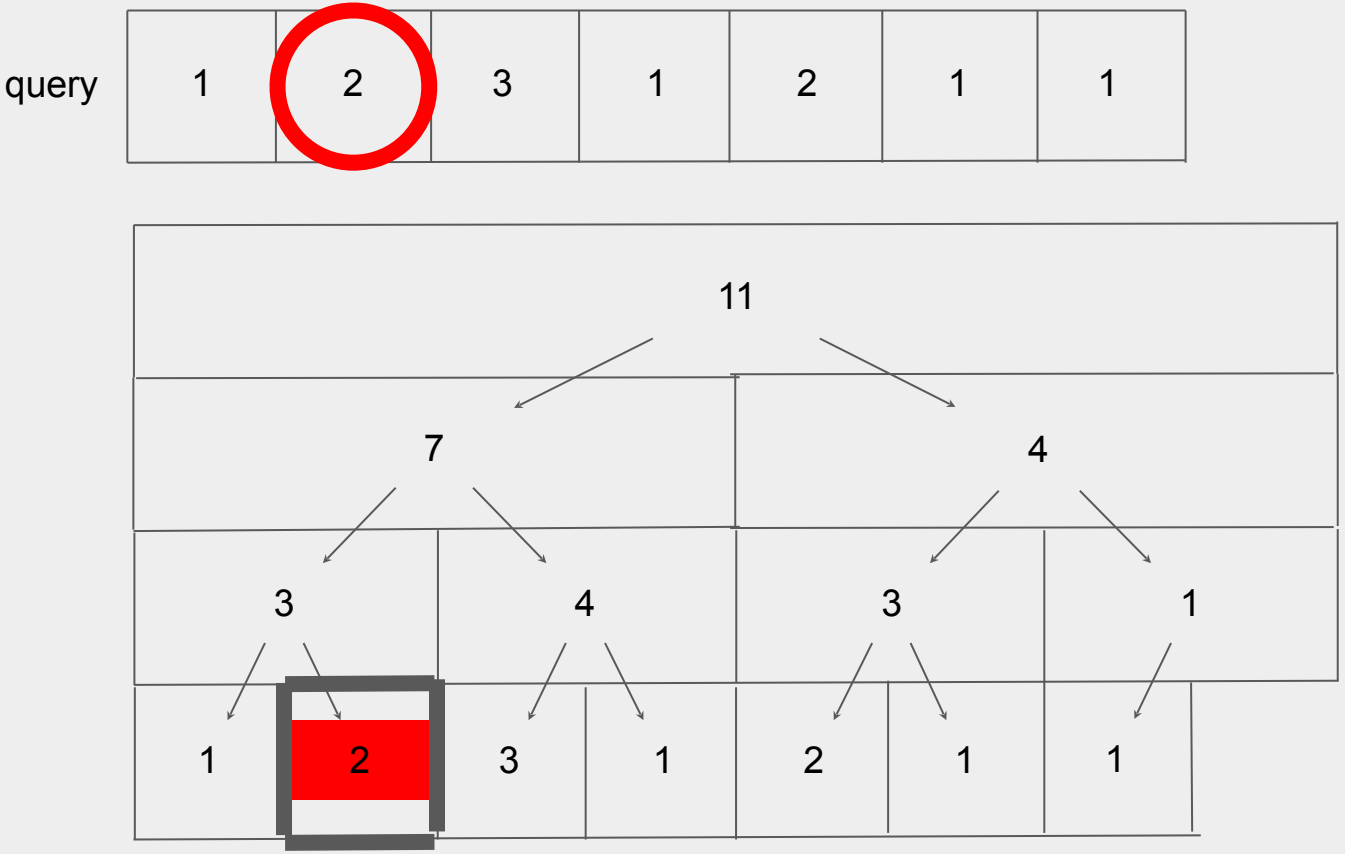
a[1] += 3



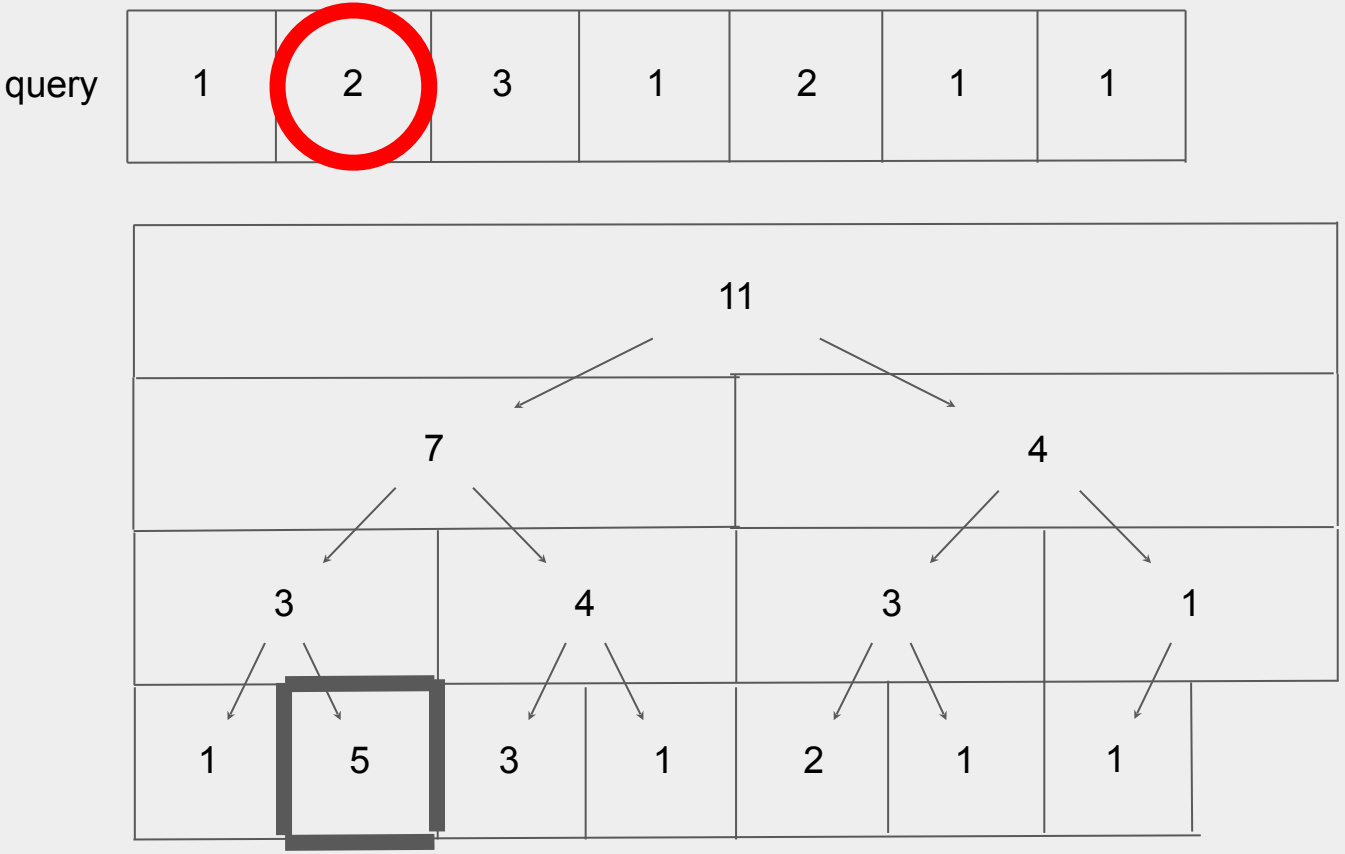
a[1] += 3



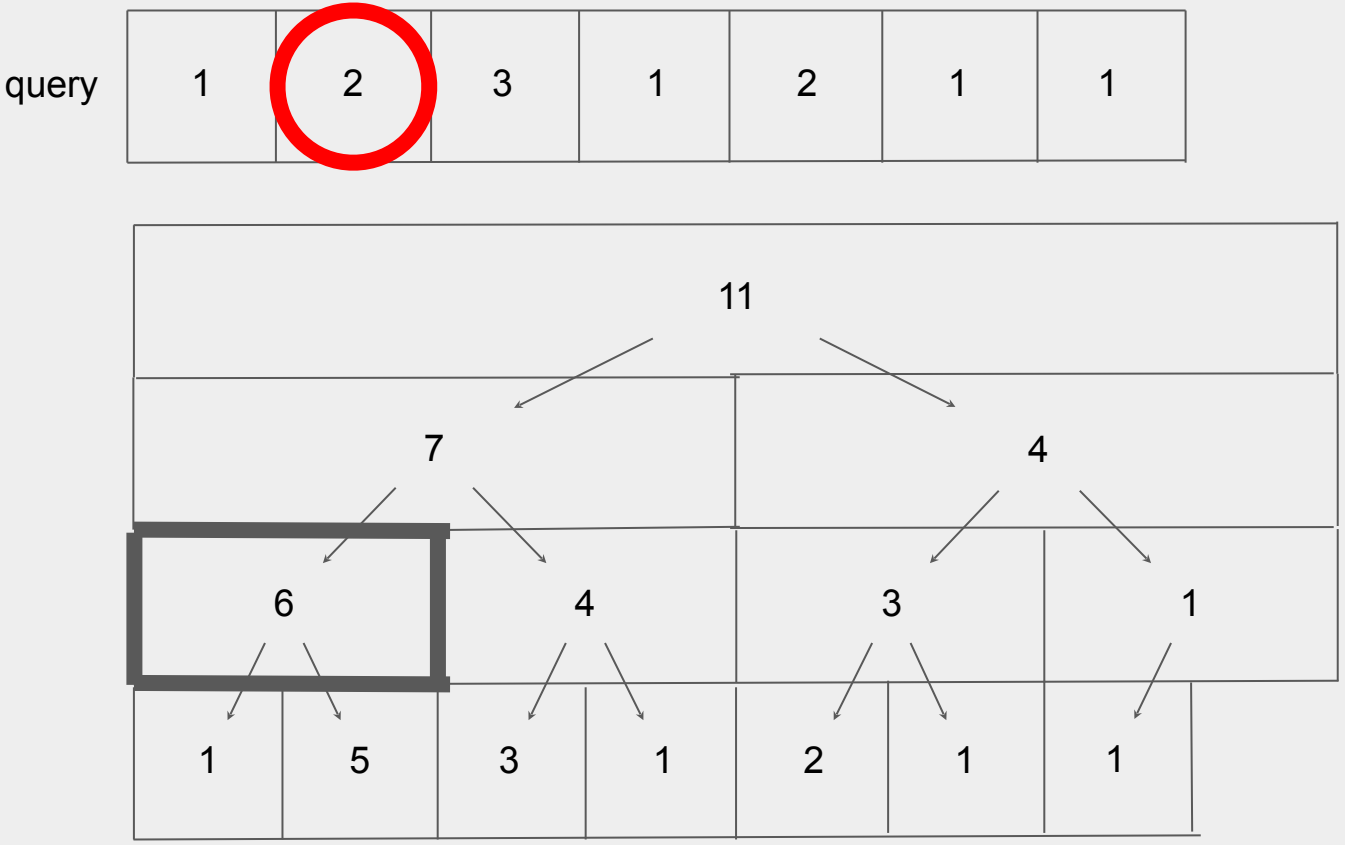
a[1] += 3



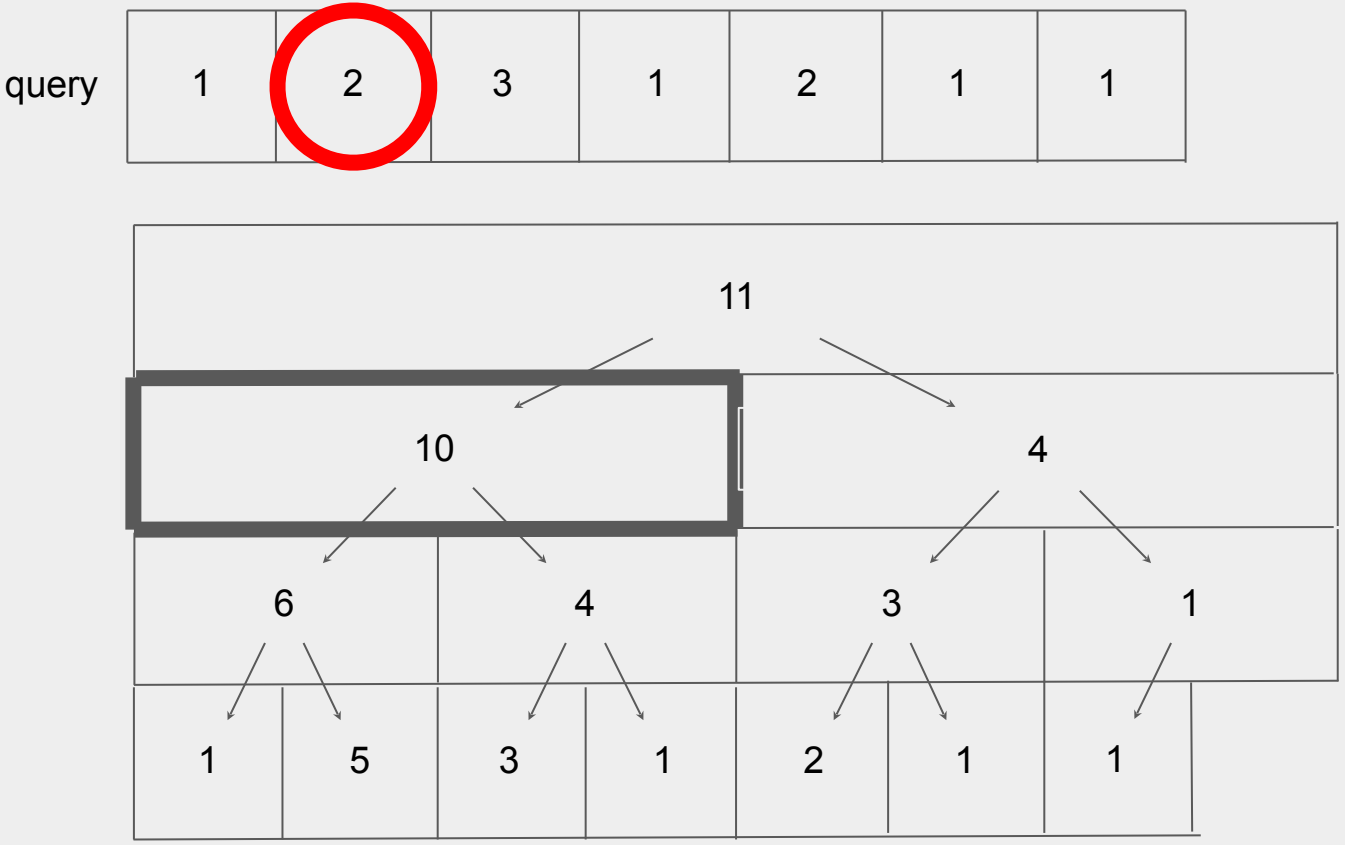
a[1] += 3



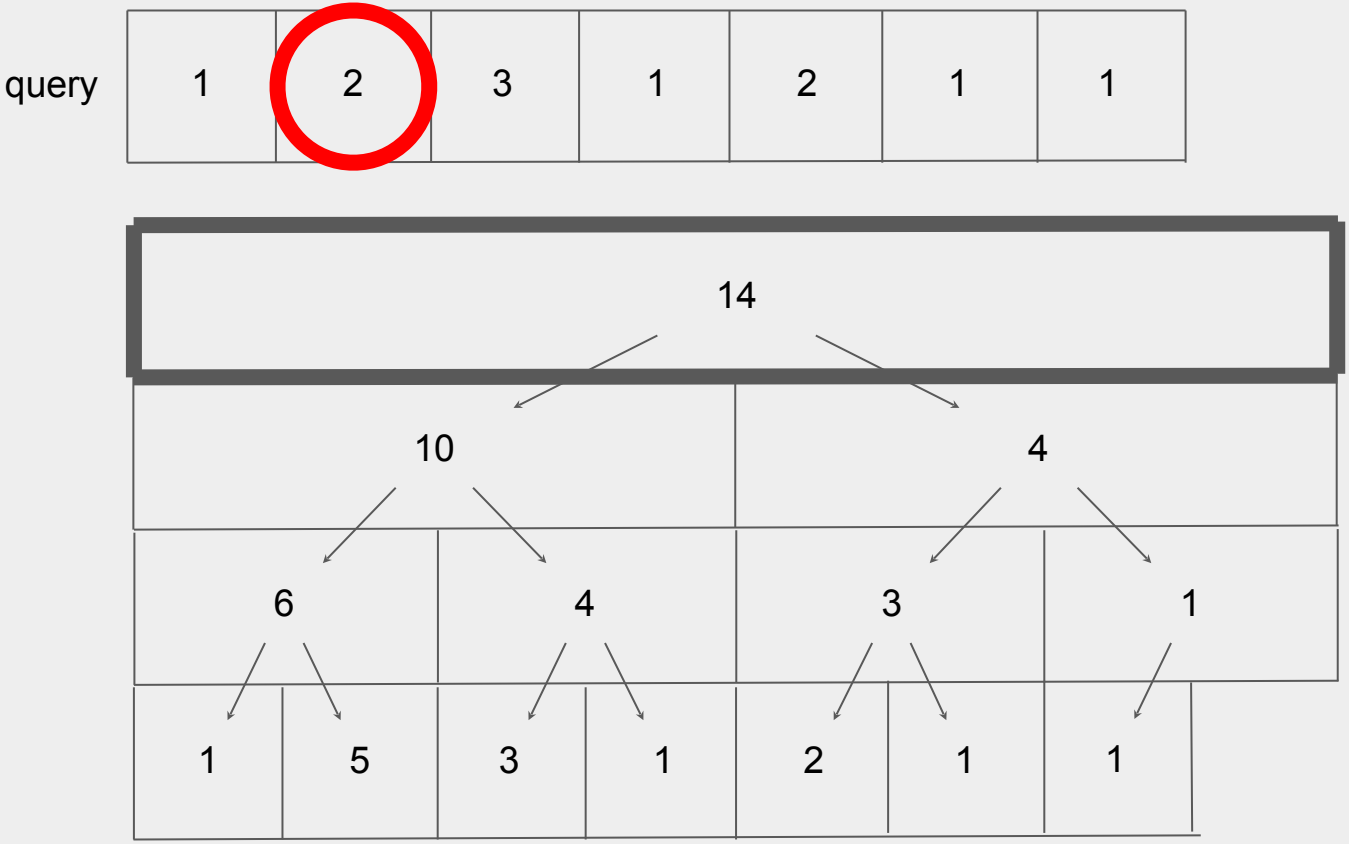
a[1] += 3



a[1] += 3



a[1] += 3

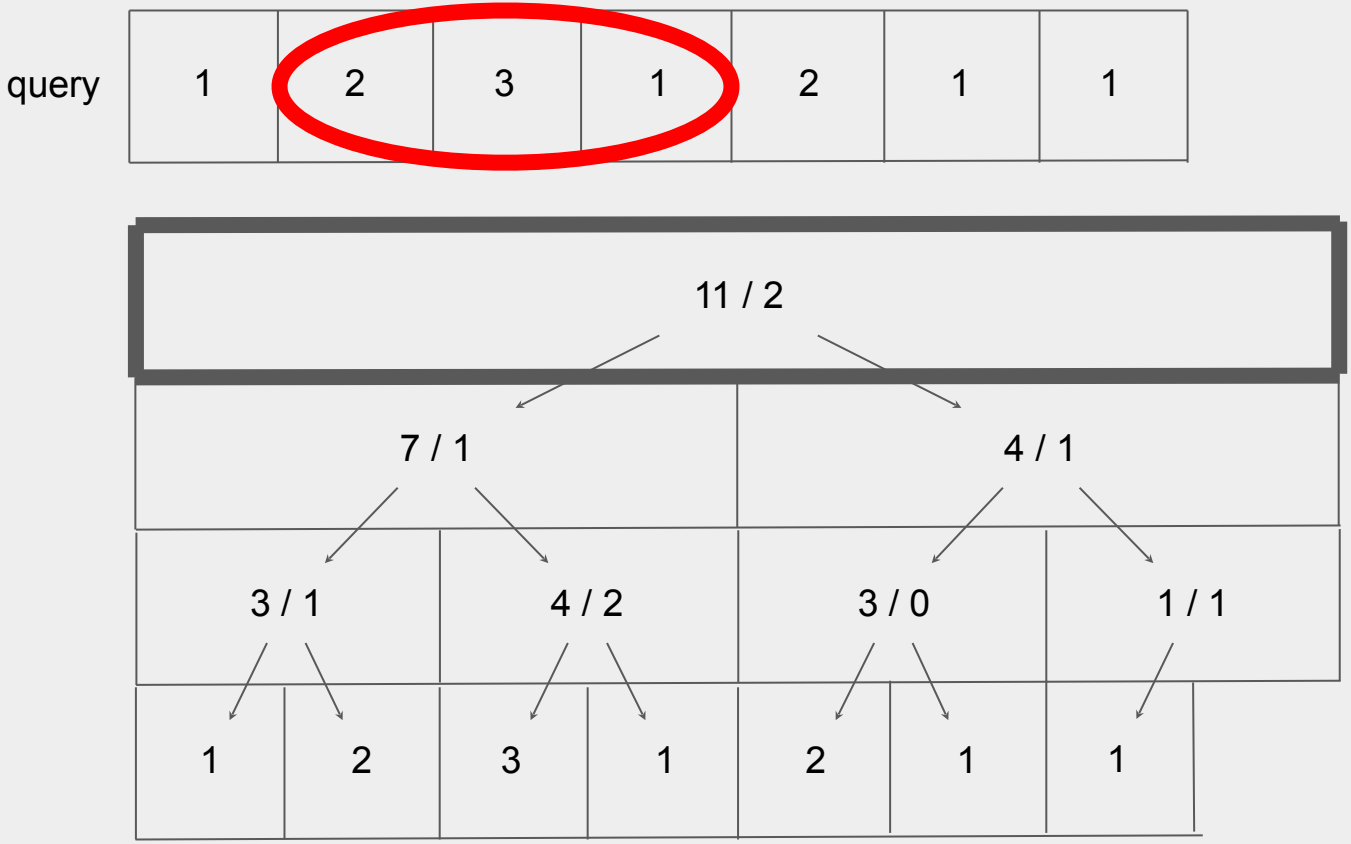


How?

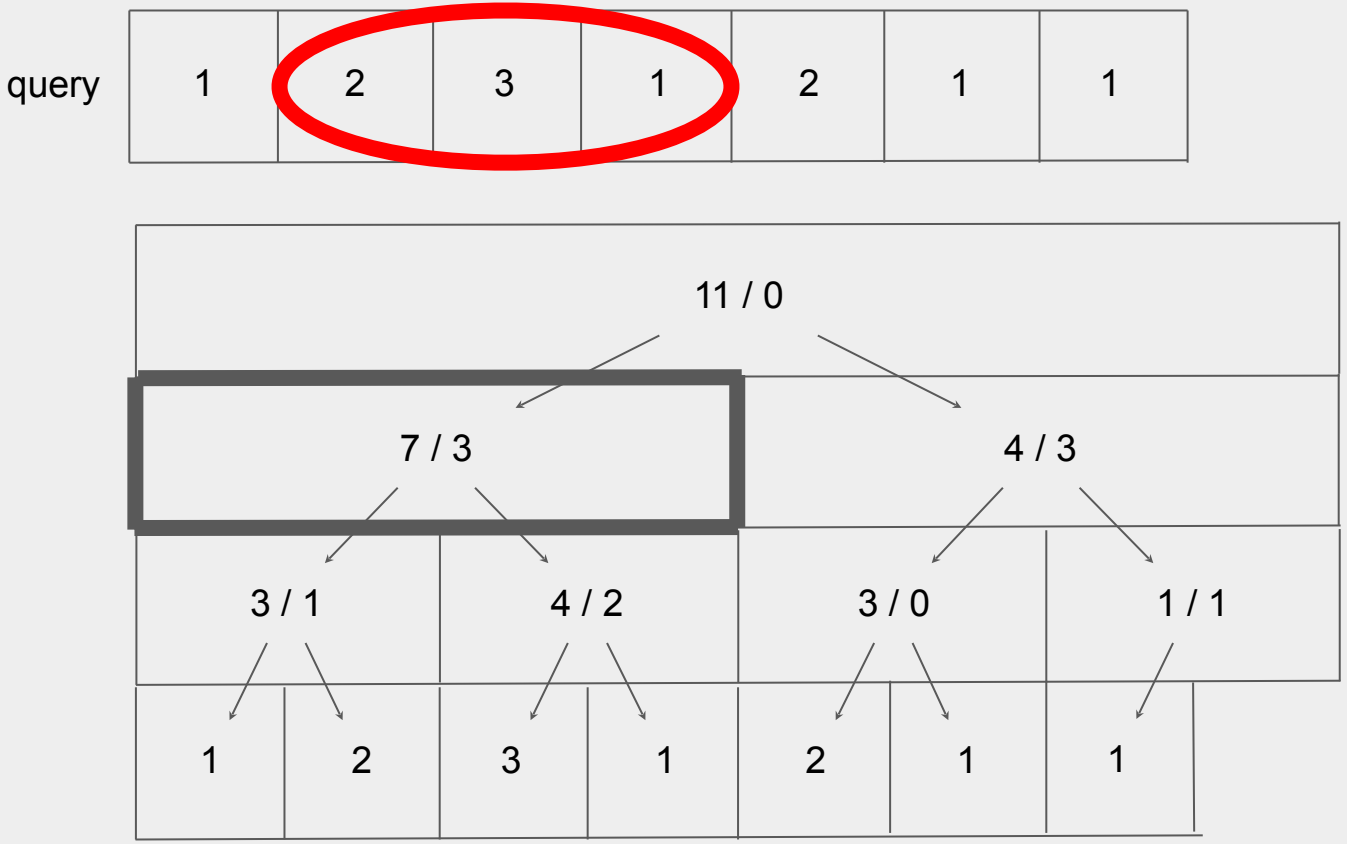
How to update a segment?

Here, we need to remember lazy propagation. Let's additionally store in the segment tree - how much we've added to each particular node. When there's a query to its children, we'll push the value to them.

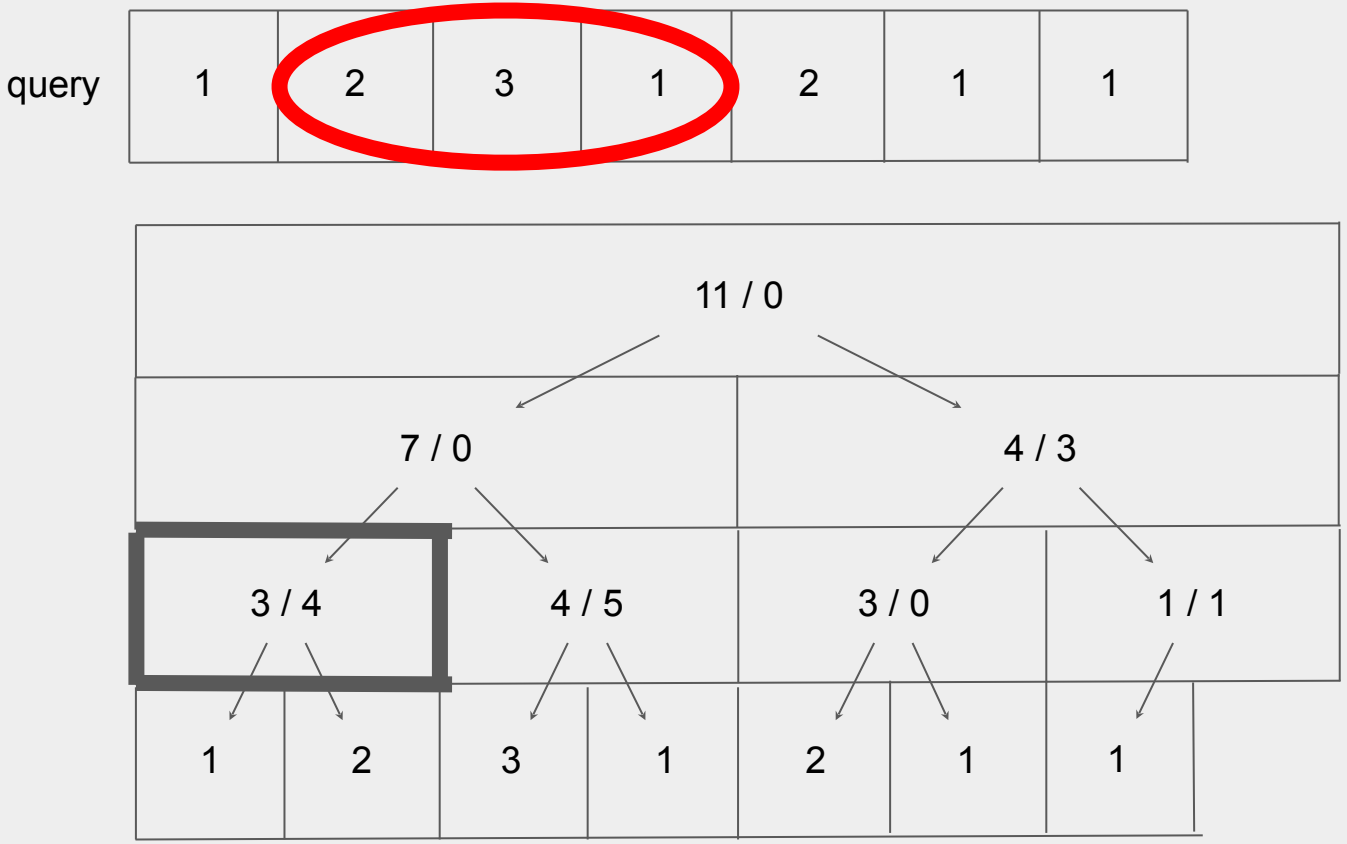
sum(1, 3)



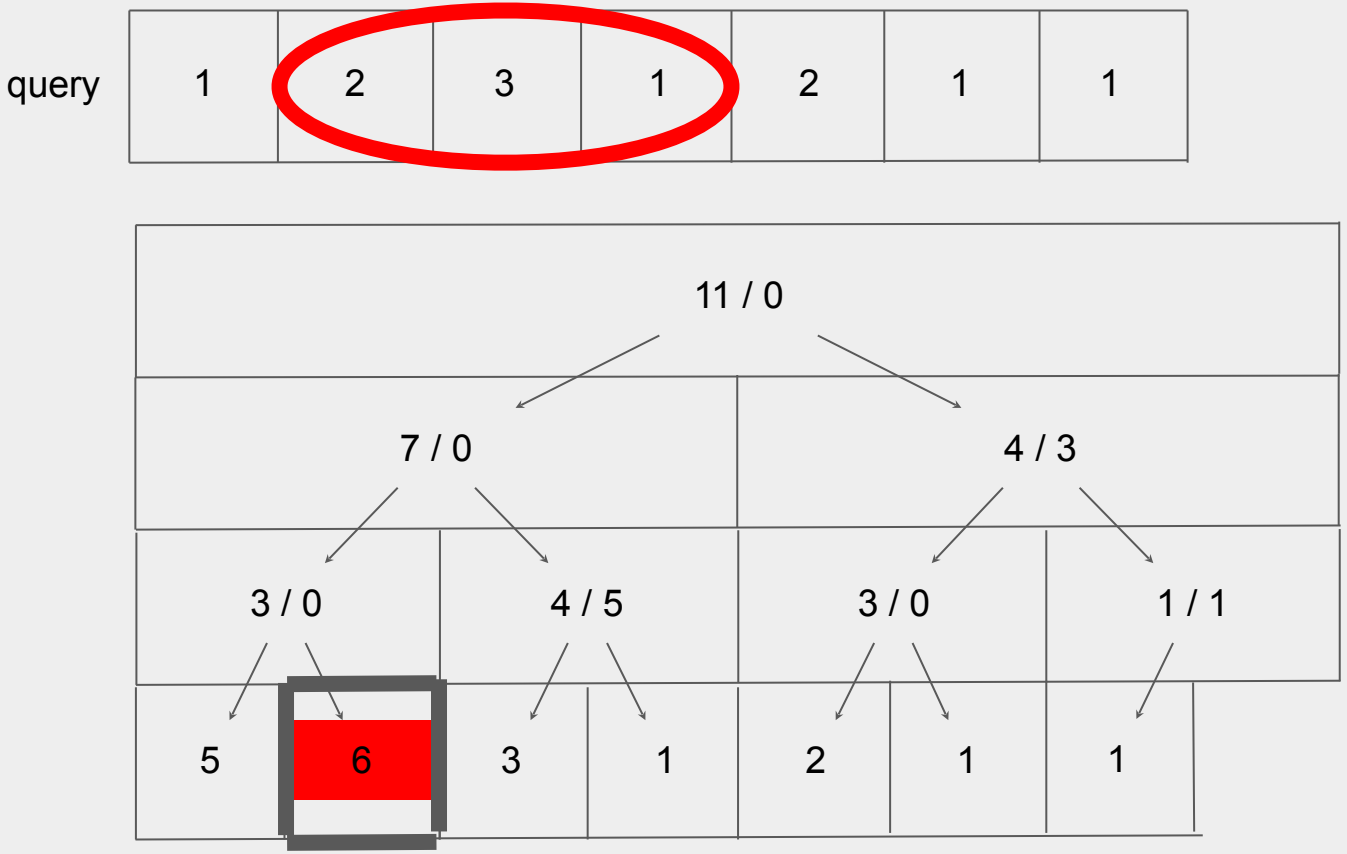
sum(1, 3)



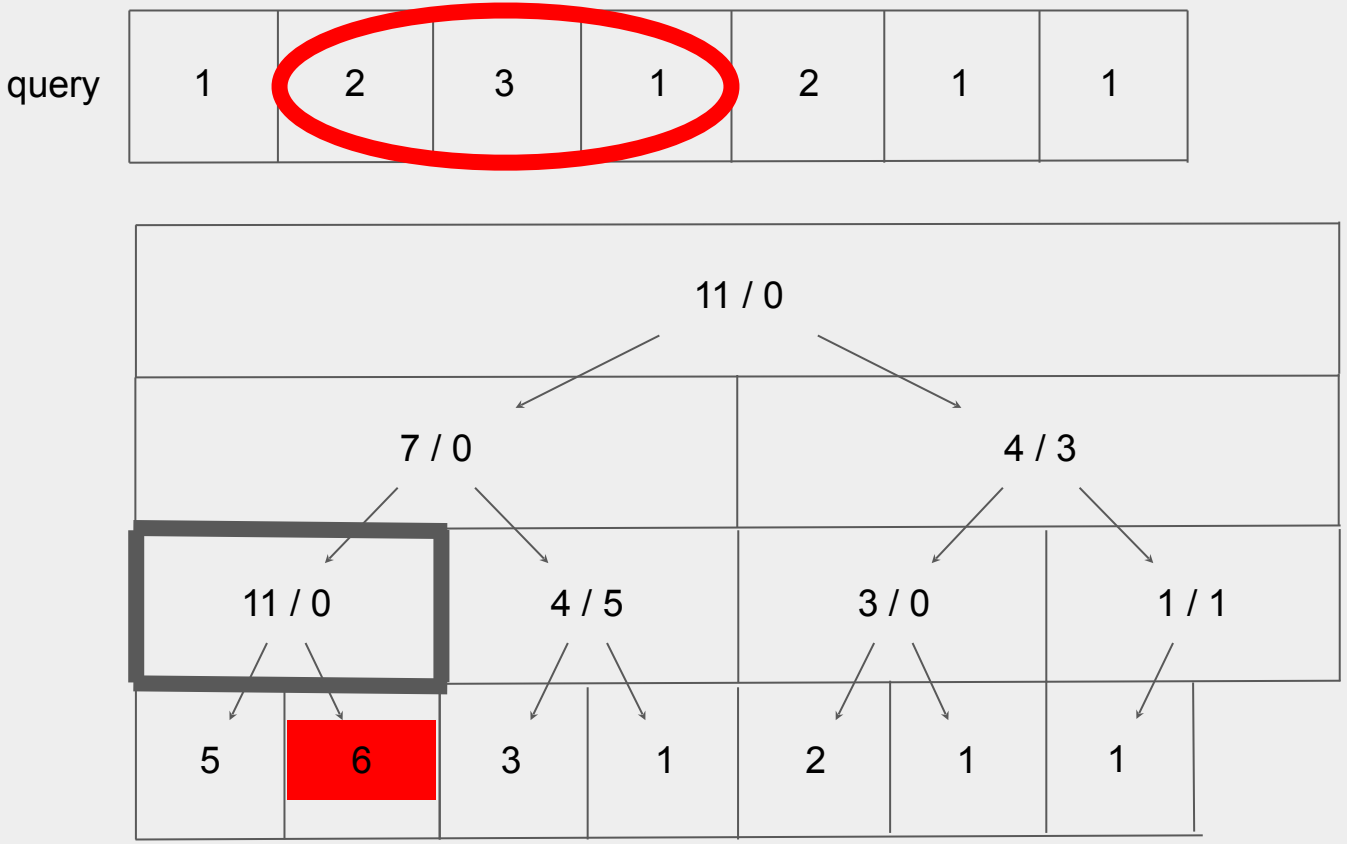
sum(1, 3)



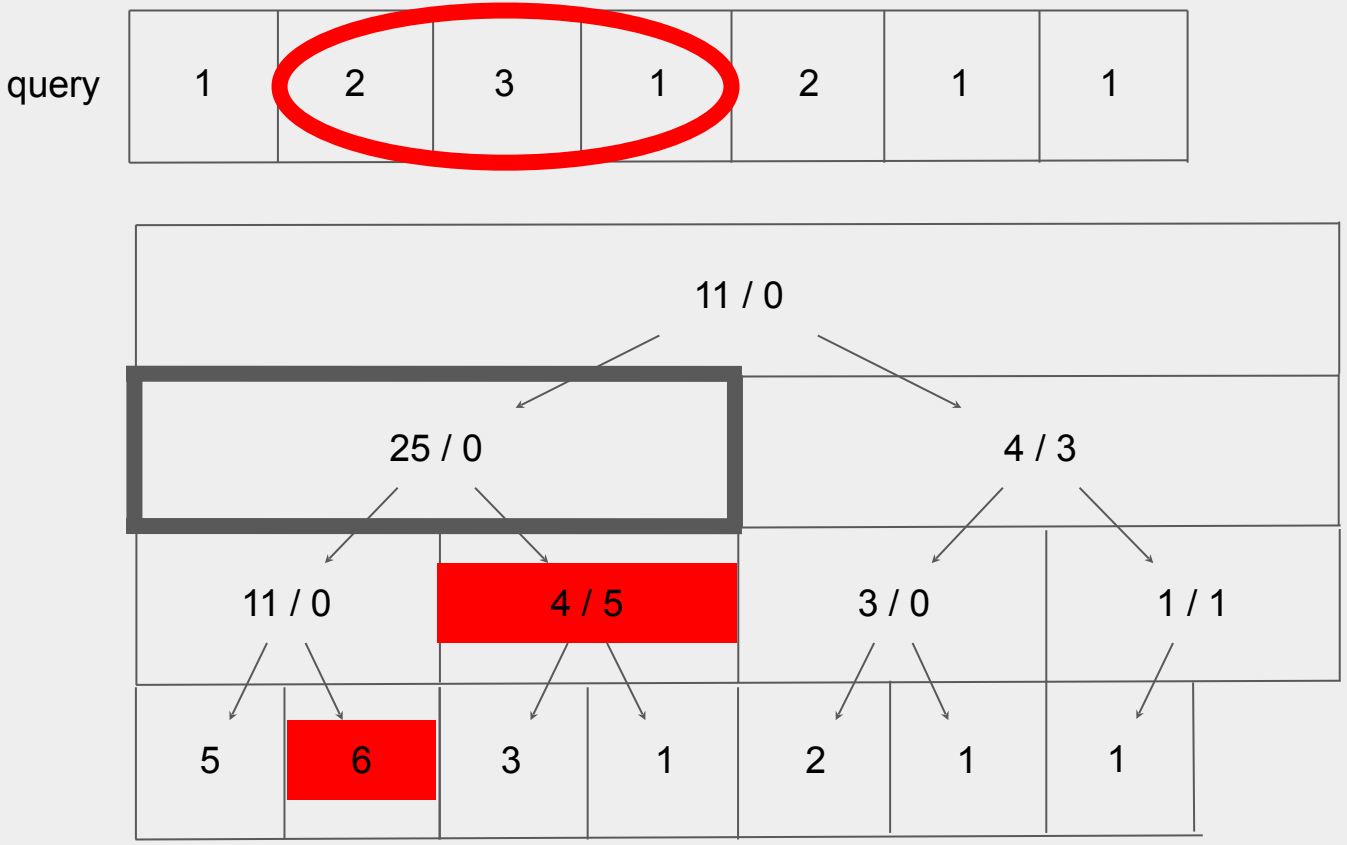
sum(1, 3)



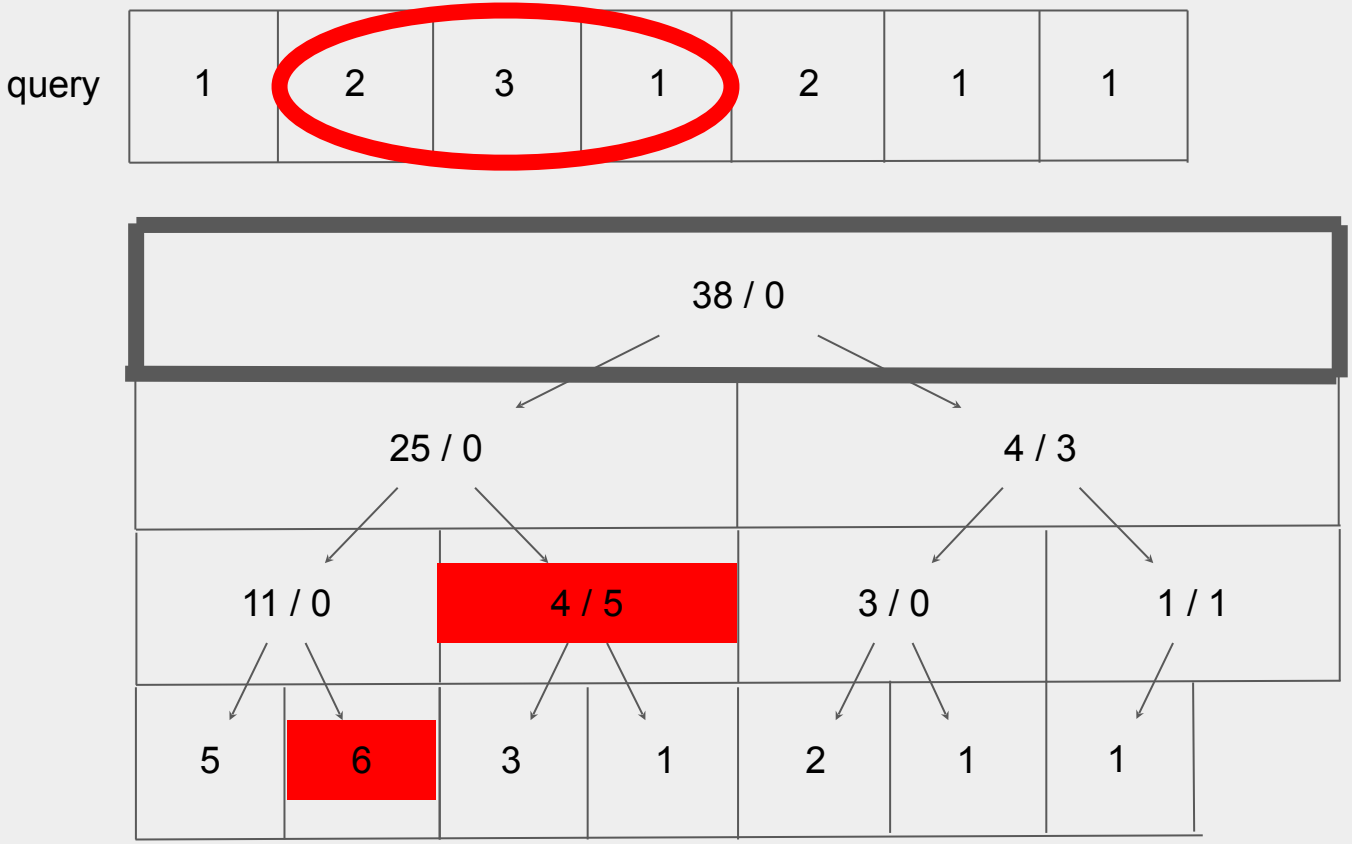
sum(1, 3)



sum(1, 3)



sum(1, 3)



```
11.  void push(int now, int l, int r, vector<Vertex> &seg_tree) {
12.      int mid = (l + r) / 2;
13.      if (now * 2 < seg_tree.size()) {
14.          seg_tree[now * 2].add += seg_tree[now].add;
15.          seg_tree[now * 2].sum += seg_tree[now].add * (mid - l);
16.      }
17.      if (now * 2 + 1 < seg_tree.size()) {
18.          seg_tree[now * 2 + 1].add += seg_tree[now].add;
19.          seg_tree[now * 2 + 1].sum += seg_tree[now].add * (r - mid);
20.      }
21.      seg_tree[now].add = 0;
22.  }
```

```
50. int addSeg(int now, int left, int right, int begin, int end, int x, vector<Vertex> &seg_tree, cons
    t vector<int> &a) {
51.     if (end <= left || right <= begin) {
52.         return 0;
53.     }
54.     if (begin <= left && right <= end) {
55.         seg_tree[now].sum += (right - left) * x;
56.         seg_tree[now].add += x;
57.         return seg_tree[now].sum;
58.     }
59.     push(now, left, right, seg_tree);
60.     int middle = (left + right) / 2;
61.     int l = addSeg(now * 2, left, middle, begin, end, x, seg_tree, a);
62.     int r = addSeg(now * 2 + 1, middle, right, begin, end, x, seg_tree, a);
63.     seg_tree[now].sum = l + r;
64. }
```

```
65. int main() {
66.     int n, k;
67.     cin >> n;
68.     vector<int> a(n);
69.     vector<Vertex> segment_tree(4 * n);
70.     for (int i = 0; i < n; i++) {
71.         cin >> a[i];
72.     }
73.     build(1, 0, n, segment_tree, a);
74.     addSeg(1, 0, n, 0, n, 1, segment_tree, a);
75.     cin >> k;
76.     while (k--) {
77.         int l, r;
78.         cin >> l >> r;
79.         l--; // Adjust to 0-based indexing
80.         int x = sumSeg(1, 0, n, l, r, segment_tree, a);
81.         cout << x << "\n";
82.     }
83.     return 0;
84. }
```

stdin

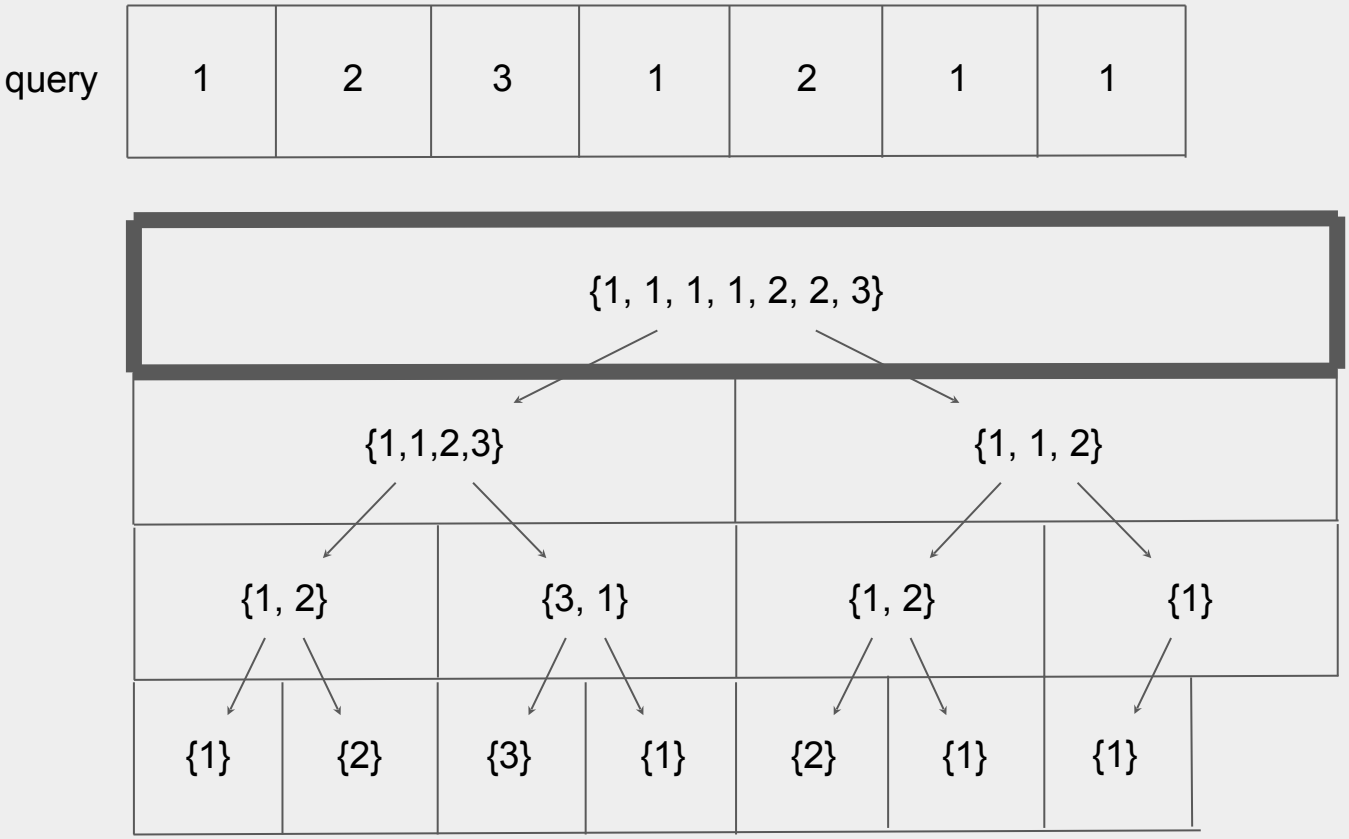
```
4
1 3 2 1
3
1 2
2 4
2 3
```

stdout

```
6
9
7
```

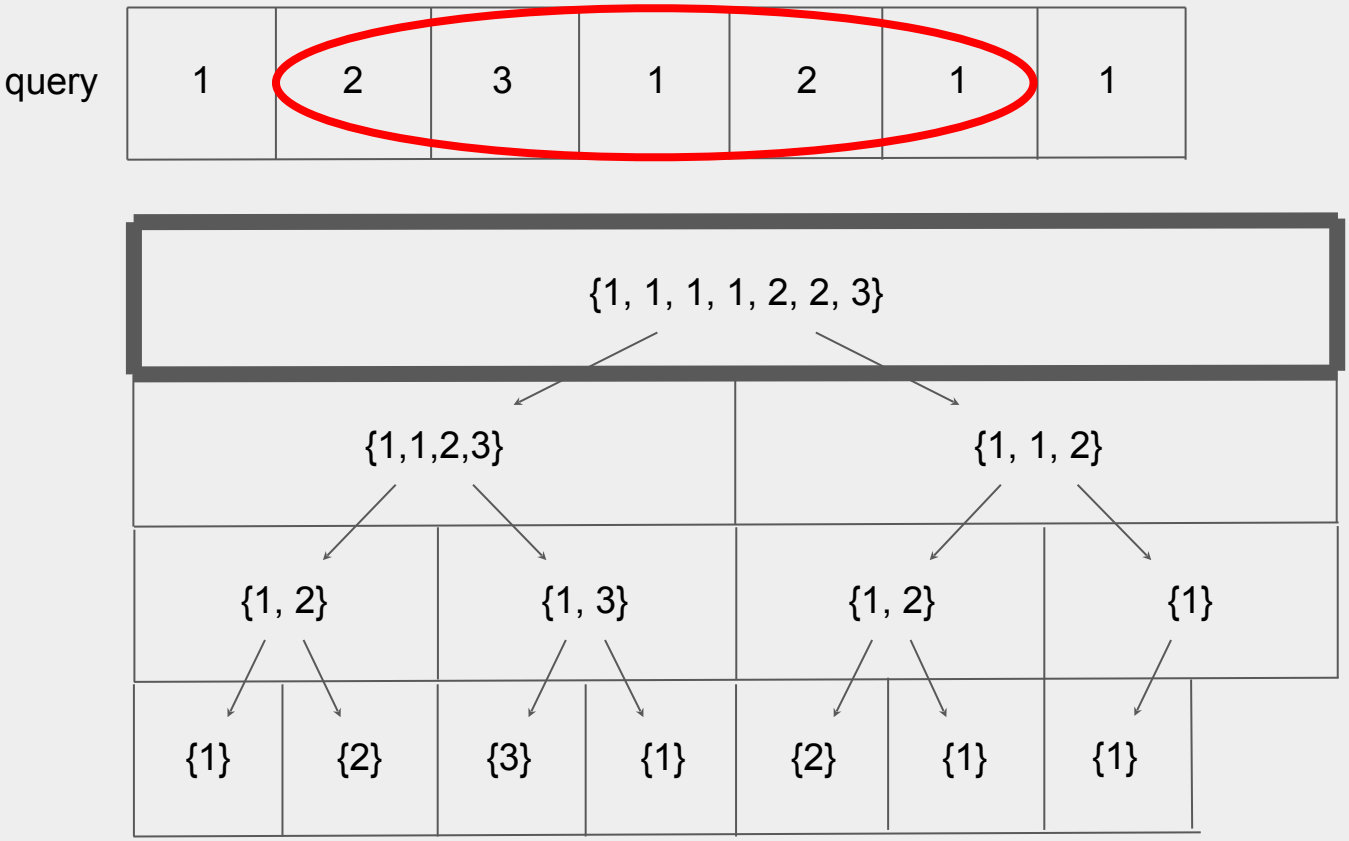
What's inside?

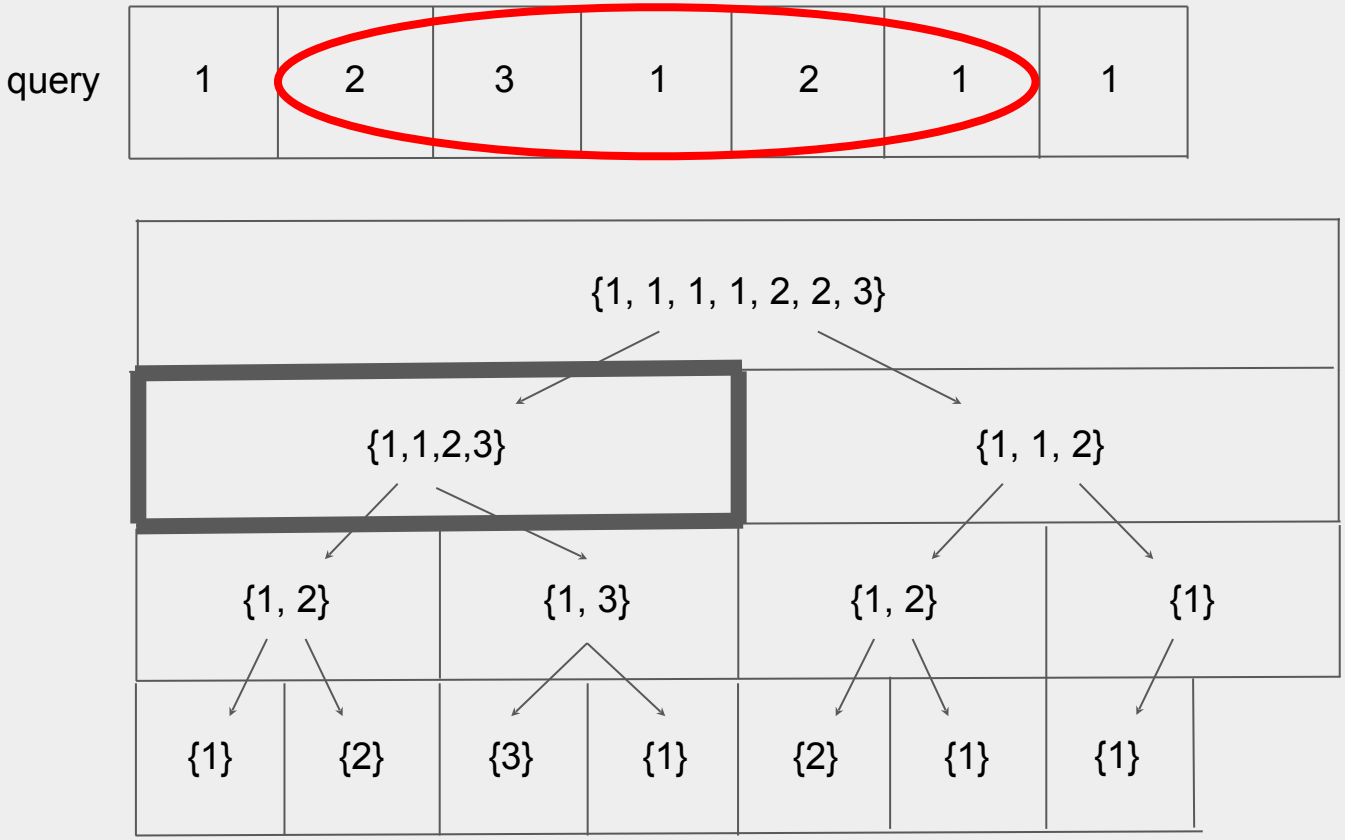
- Similar to root decomposition, we can store certain objects in it.
- Let's solve the same problem - finding the count of numbers less than X in a segment.

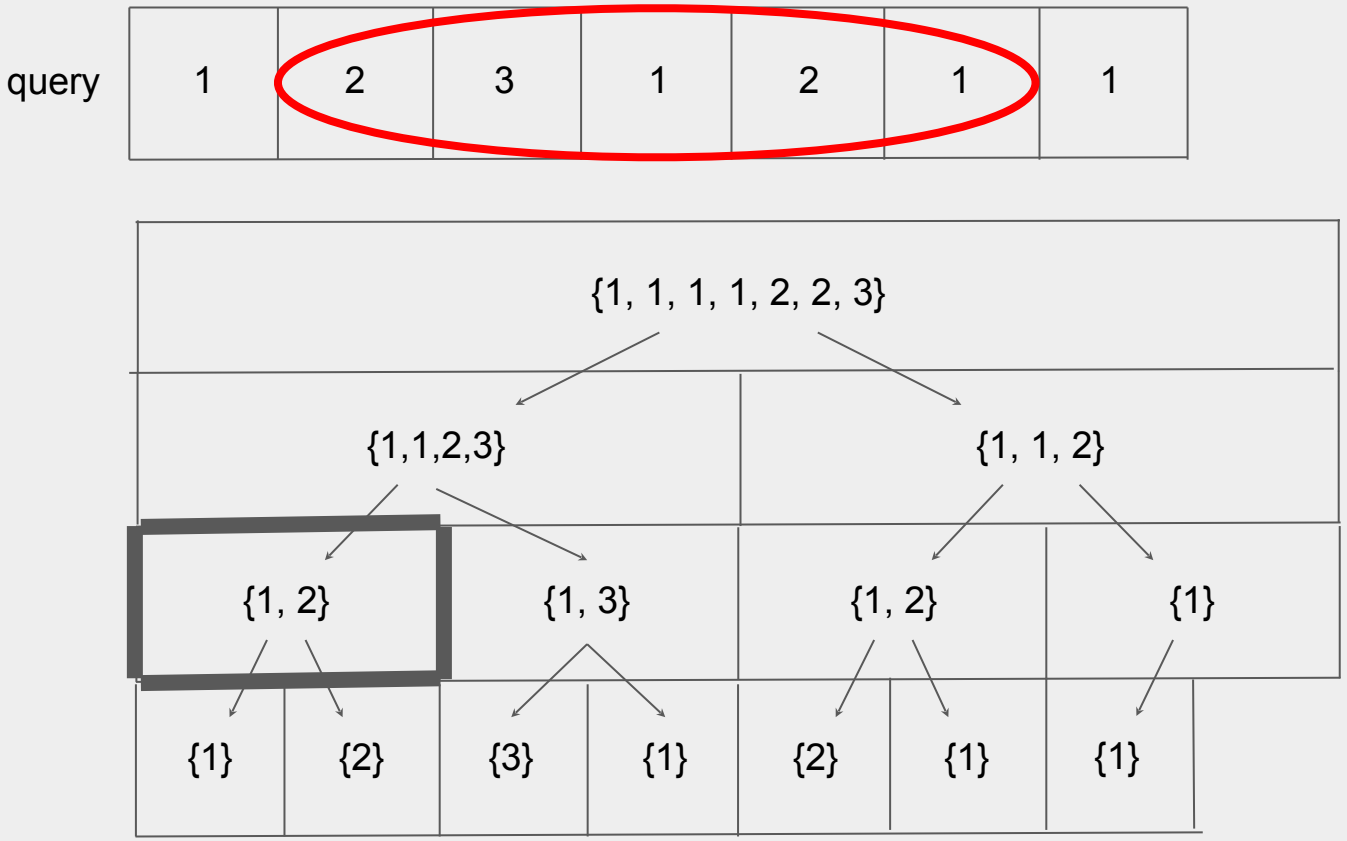

query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{3, 1}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}

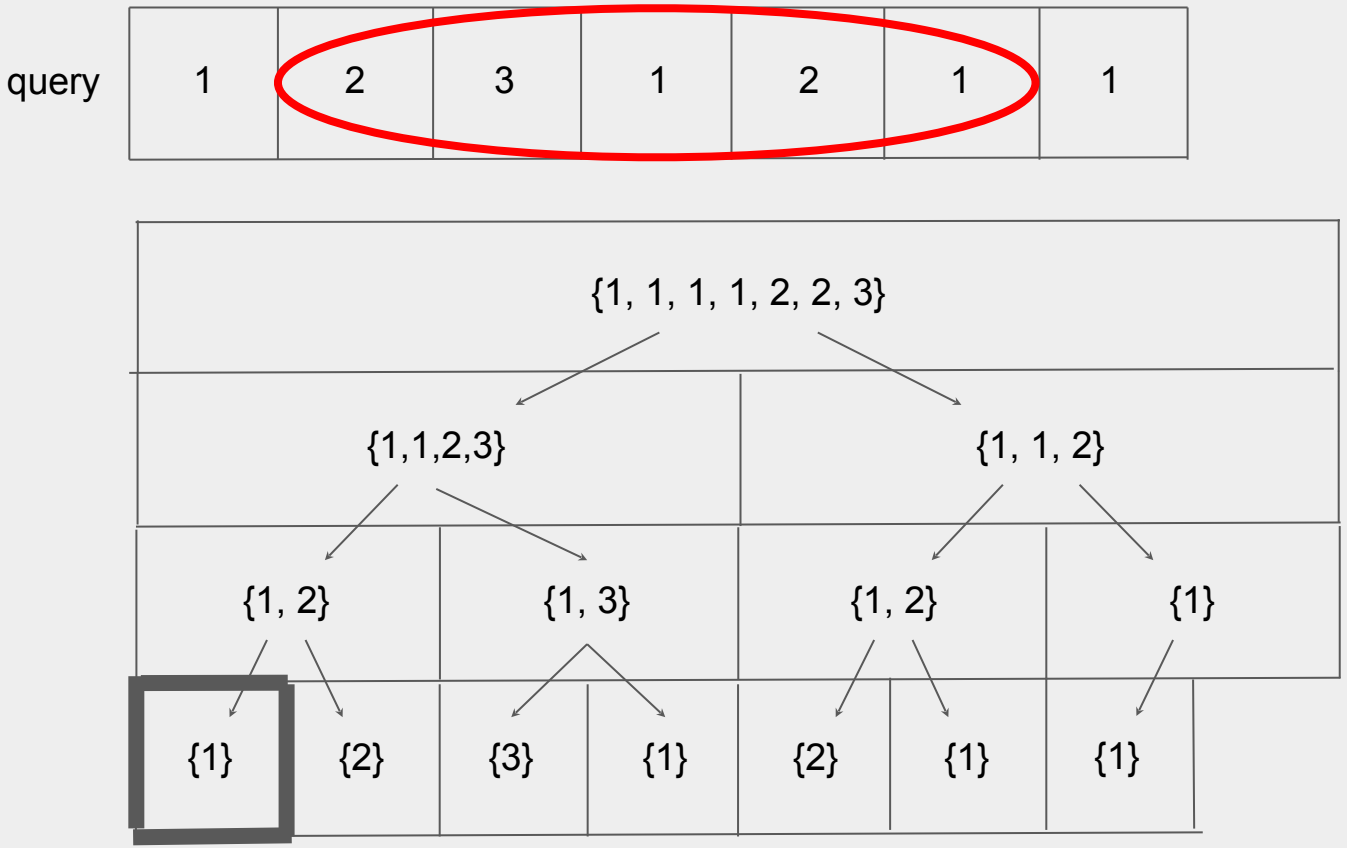
```
6.  struct Vertex {
7.      vector<int> elems = {};
8.  };
```

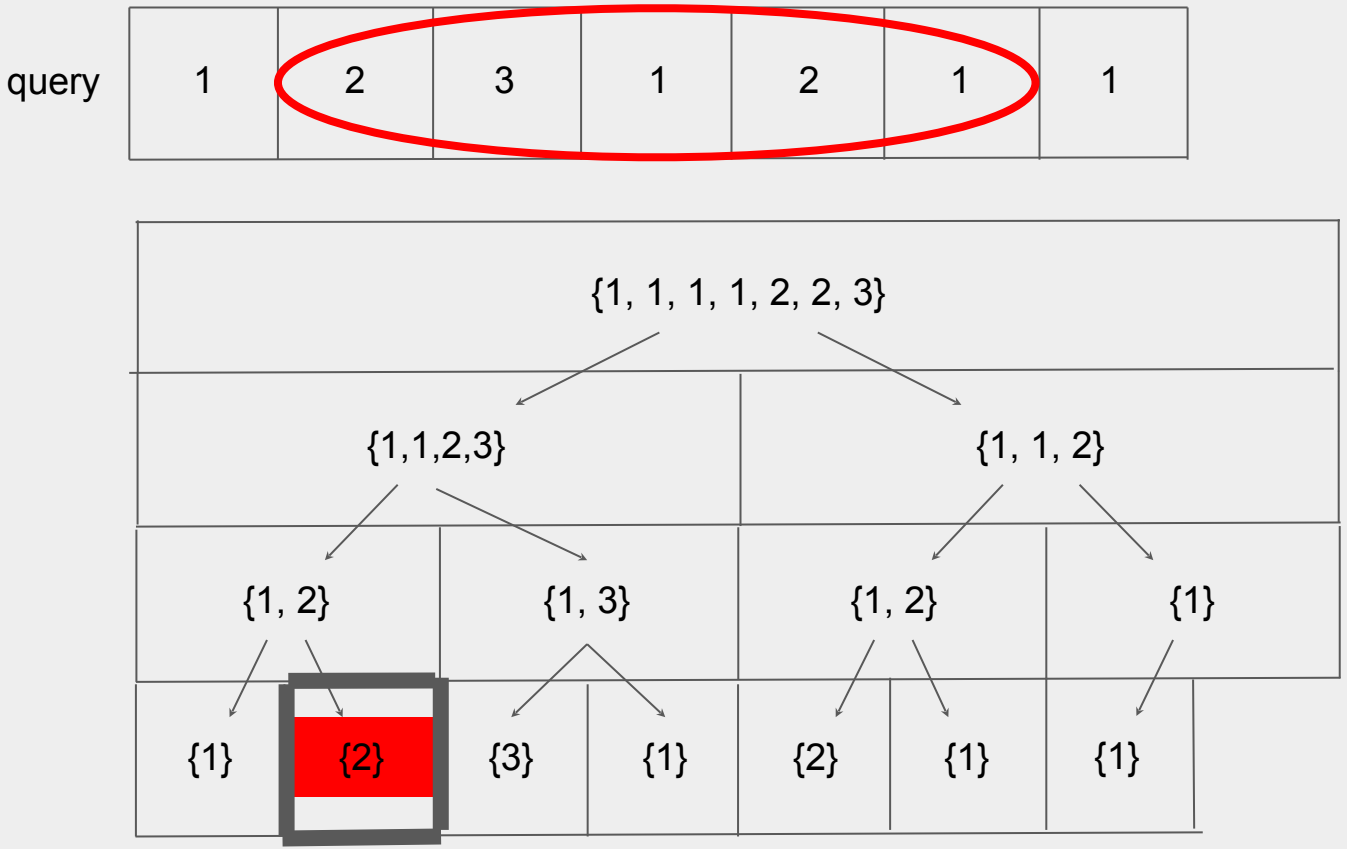
```
10. void mergeSubTrees(vector<Vertex> &seg_tree, int p, int l, int r) {
11.     merge(seg_tree[l].elems.begin(), seg_tree[l].elems.end(), seg_tree[r].elems.begin(), seg_tree
    [r].elems.end(), std::back_inserter(seg_tree[p].elems));
12. }
13.
14. // build segment tree for range [begin, end)
15. void build(int now, int begin, int end, vector<Vertex> &seg_tree, vector<int> &a) {
16.     if (begin == end - 1) {
17.         seg_tree[now].elems.push_back(a[begin]);
18.         return;
19.     }
20.     int mid = (begin + end) / 2;
21.     build(now * 2, begin, mid, seg_tree, a);
22.     build(now * 2 + 1, mid, end, seg_tree, a);
23.     mergeSubTrees(seg_tree, now, now * 2, now * 2 + 1);
24. }
```

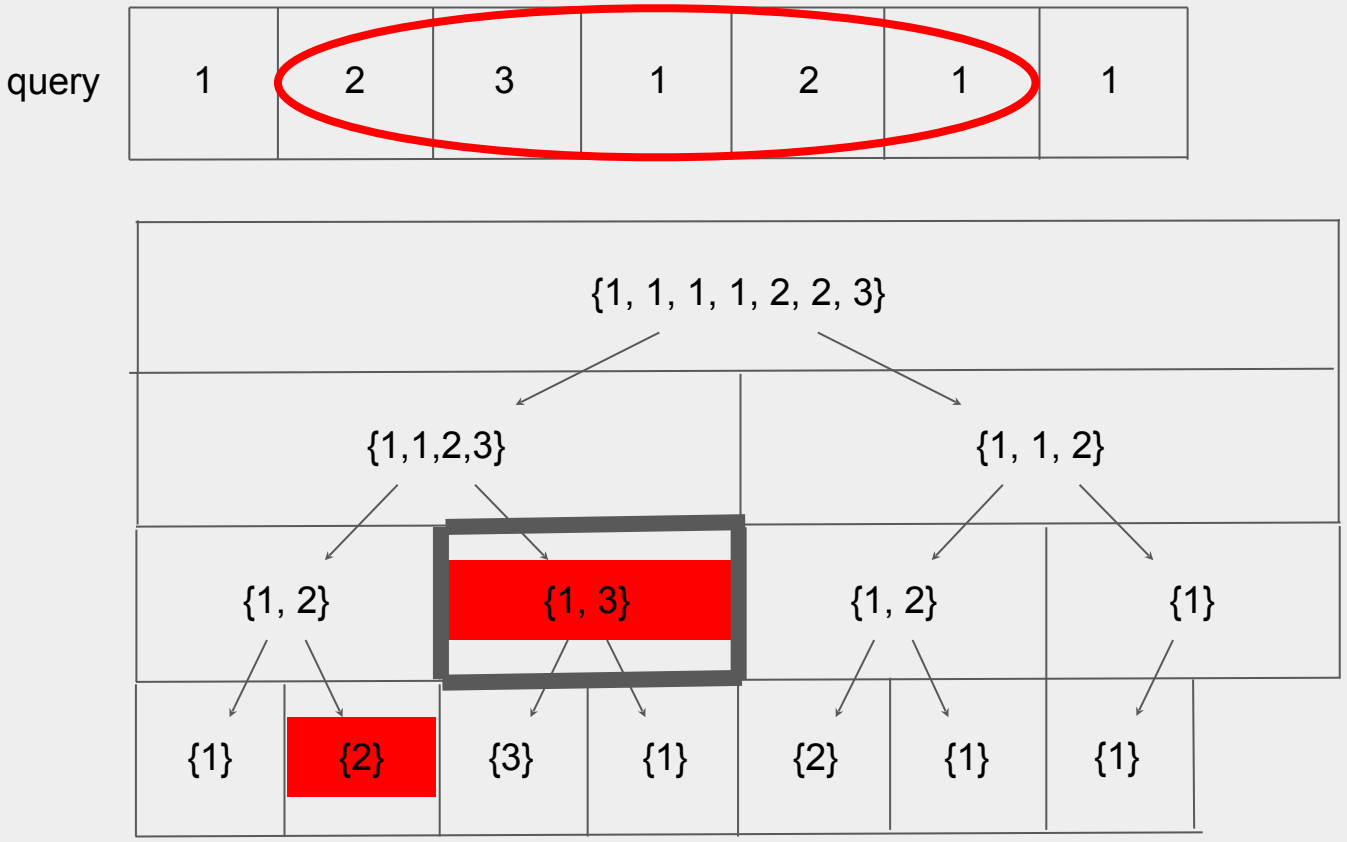

query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}

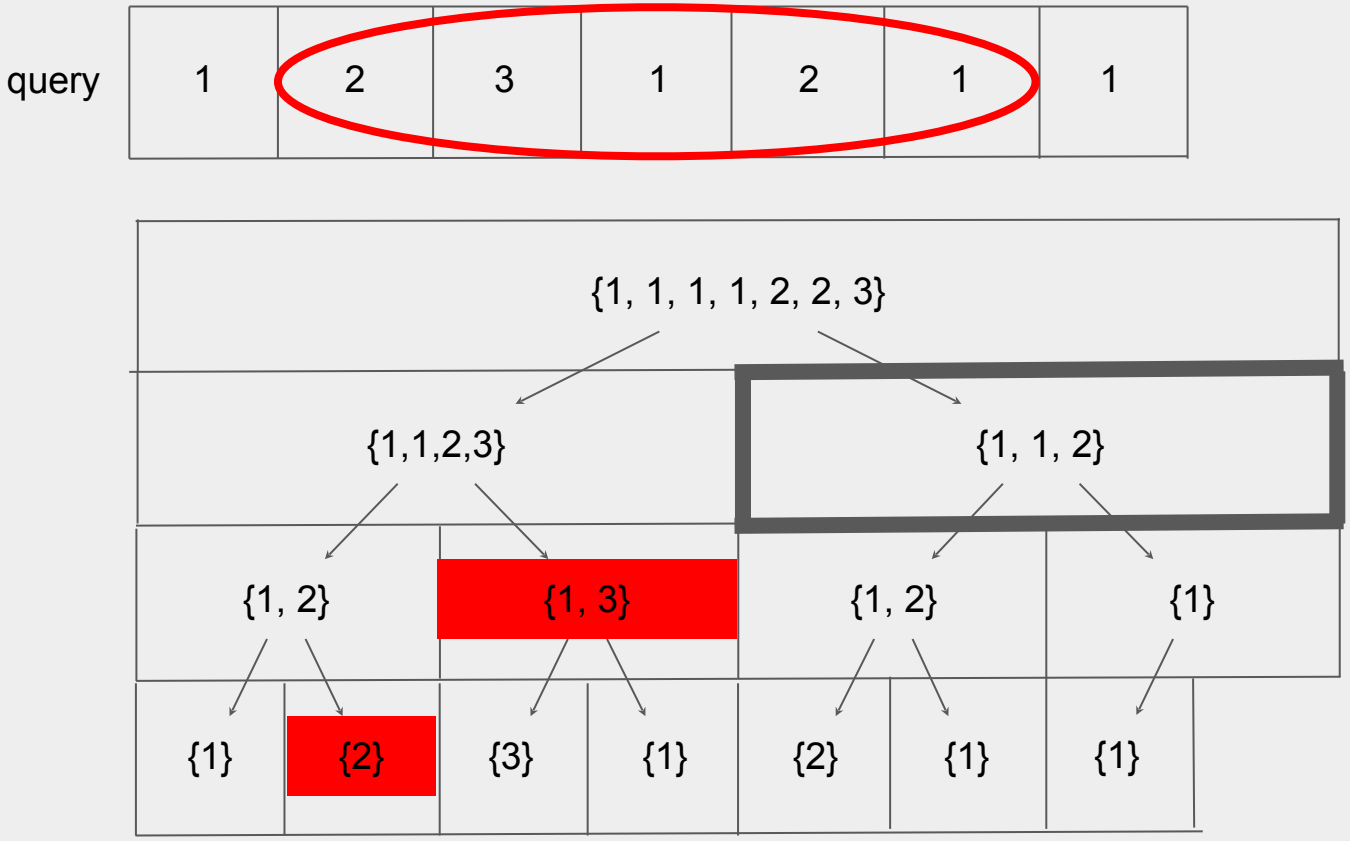

query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}

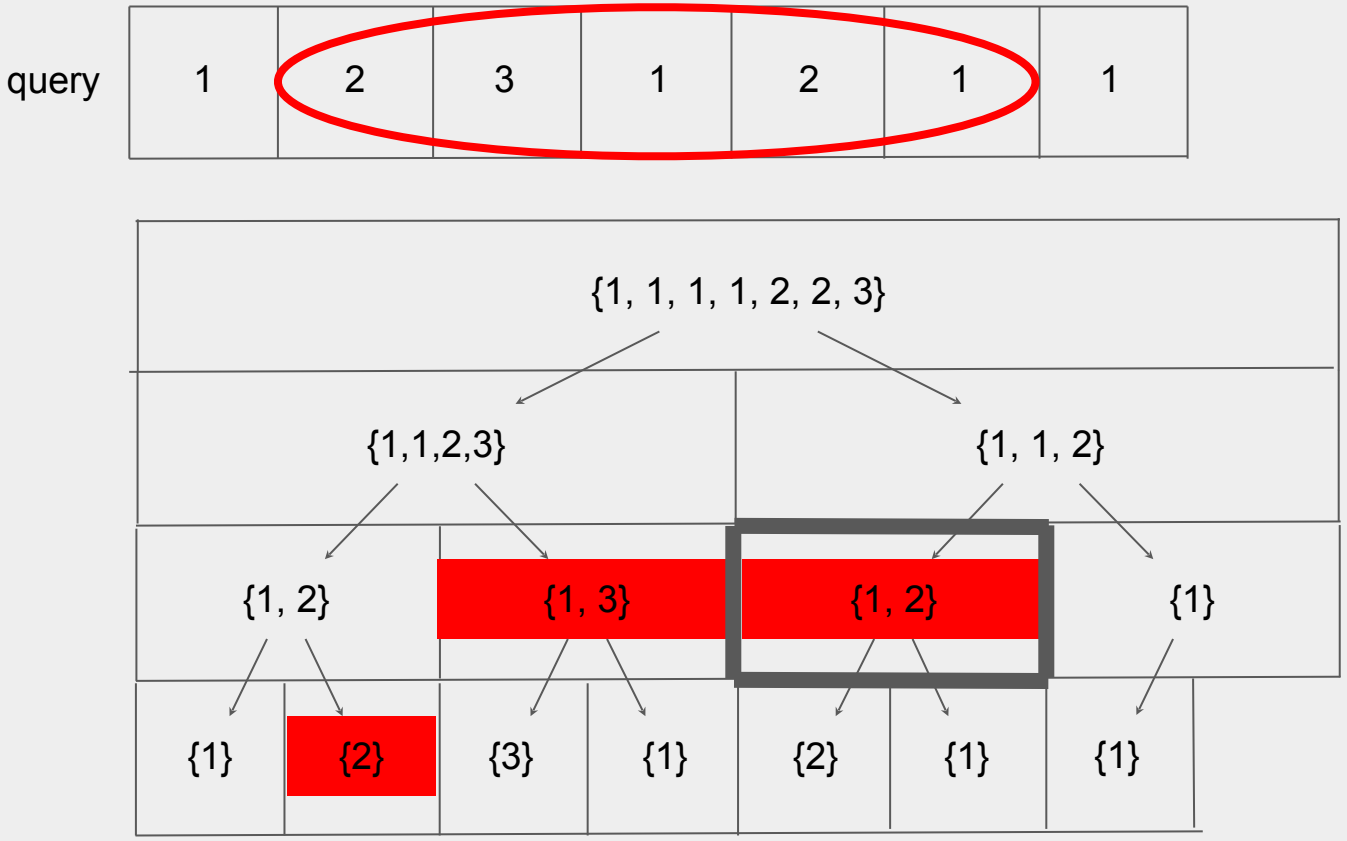

query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}

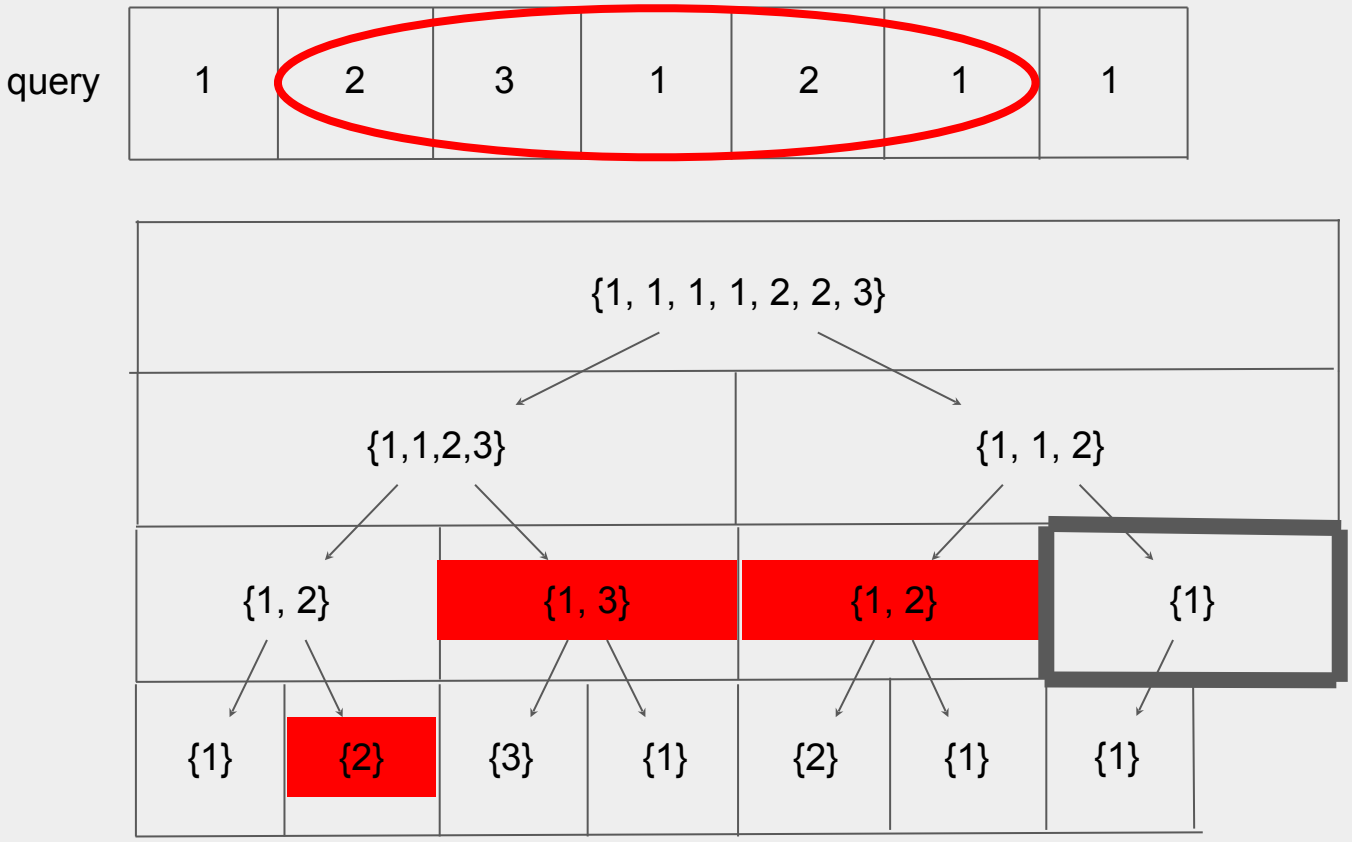

query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}


query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}


query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}


query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}


query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}


query
1
2
3
1
2
1
1
{1, 1, 1, 1, 2, 2, 3}
{1,1,2,3}
{1, 1, 2}
{1, 2}
{1, 3}
{1, 2}
{1}
{1}
{2}
{3}
{1}
{2}
{1}
{1}

```
26. int amountSeg(int now, int left, int right, int begin, int end, int x, vector<Vertex> &seg_tree) {
27.     if (end <= left || right <= begin) {
28.         return 0;
29.     }
30.     if (begin <= left && right <= end) {
31.         auto it = std::upper_bound(seg_tree[now].elems.begin(), seg_tree[now].elems.end(), x);
32.         return std::distance(seg_tree[now].elems.begin(), it);
33.     }
34.     int middle = (left + right) / 2;
35.     return amountSeg(now * 2, left, middle, begin, end, x, seg_tree) + amountSeg(now * 2 + 1, midd
    le, right, begin, end, x, seg_tree);
36. }
```

No updates(

- It's important to note that using a Segment Tree alone won't be sufficient to solve problems, especially those involving updates to elements or segments.
- For this, at the very least, we'd require an ordered data structure like Policy-Based Data Structures (pbds) or a self-implemented balanced tree.

```
38. int main() {
39.     int n, k;
40.     cin >> n;
41.     vector<int> a(n);
42.     vector<Vertex> segment_tree(4 * n);
43.     for (int i = 0; i < n; i++) {
44.         cin >> a[i];
45.     }
46.     build(1, 0, n, segment_tree, a);
47.     cin >> k;
48.     while (k--) {
49.         int l, r, x;
50.         cin >> l >> r >> x;
51.         l--; // Adjust to 0-based indexing
52.         int amount = amountSeg(1, 0, n, l, r, x, segment_tree);
53.         cout << amount << "\n";
54.     }
55.     return 0;
56. }
```

Success #stdin #stdout 0.01s 5436KB

stdin

```
4
1 3 2 1
3
1 2 1
2 4 1
2 3 1
```

stdout

```
1
1
0
```

Task 1

Need to find the index of the minimum element in a segment.

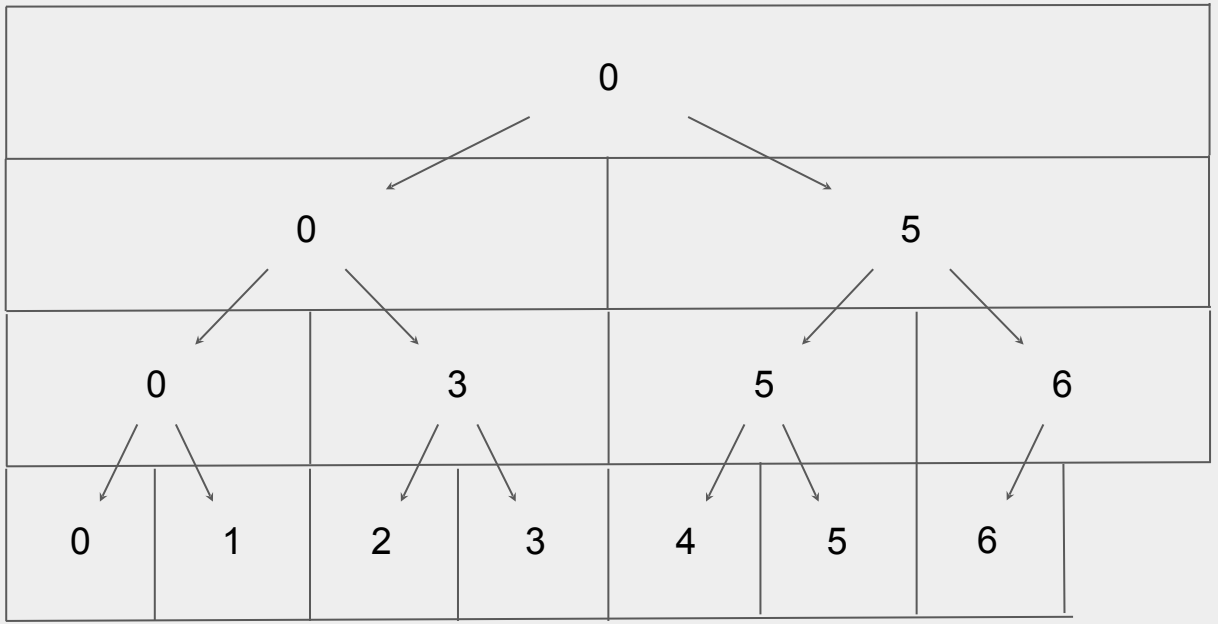
For example, for the segment '1, 2, 3, 0, 1, 2' the answer is the element at index 3, which is 0.

Idea

Let’s keep not only the minimum, but also the index of minimum.

Example: array = [1, 2, 3, 1, 2, 1, 1]

array = [1, 2, 3, 1, 2, 1, 1]



Task 2

Need to find the number of Longest Increasing Subsequences. For example, for the sequence '1, 3, 2, 4' the answer is 2 ('1, 3, 4', '1, 2, 4').

a[i] <= 10**6

DP2(GIS)

- dp[i] - {GIS(elems) for elem in elems : elem < i, elems.last = i, amount of such GIS}
- dp[i] = {0, 0} for all i, dp[a[i]] = {1, 1} for all i
- from i = 0 to n - 1 : a[i]
- dp[a[i]] = {max(dp[j] + 1) for all j < a[i], sum of such ways}
- sum(dp[i]) for all dp[i] is maximum

Example

sequence '1, 3, 2, 4'

answer - ('1, 3, 4', '1, 2, 4').

dp[4] = {3, 2}

DP2(GIS)

dp[1] = 1/1

dp[2] = 1/1

dp[3] = 2/2

1	3	-1	2	3	0	4

-1	0	1	2	3	4	5
0/0	0/0	1/1	0/0	0/0	0/0	0/0

1	3	-1	2	3	0	4

-1	0	1	2	3	4	5
0/0	0/0	1/1	0/0	2/1	0/0	0/0

1	3	-1	2	3	0	4

-1	0	1	2	3	4	5
1/1	0/0	1/1	0/0	2/1	0/0	0/0

1	3	-1	2	3	0	4

-1	0	1	2	3	4	5
1/1	0/0	1/1	2/2	2/1	0/0	0/0

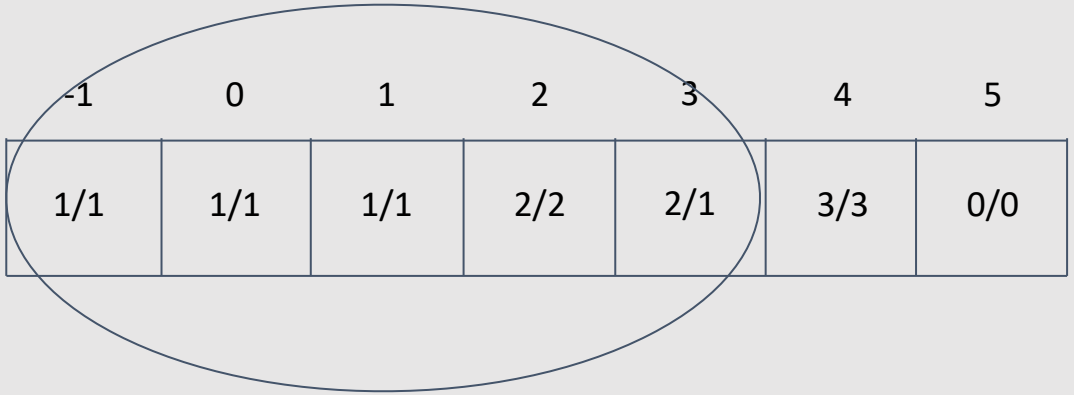
1	3	-1	2	3	0	4

-1	0	1	2	3	4	5
1/1	1/1	1/1	2/2	2/1	0/0	0/0

1	3	-1	2	3	0	4

-1	0	1	2	3	4	5
1/1	1/1	1/1	2/2	2/1	3/3	0/0

1	3	-1	2	3	0	4



Seg Tree

We have dp for n^2

To optimize it we use seg_tree, to keep sum of max elements.

We have operation increase(so it’s easy)

Task 3

Your task is to find the size of the maximum subsegment consisting of 0s in the given range. For instance, in the sequence '1000110011,' the maximum length of the subsegment consisting of 0s is 3.

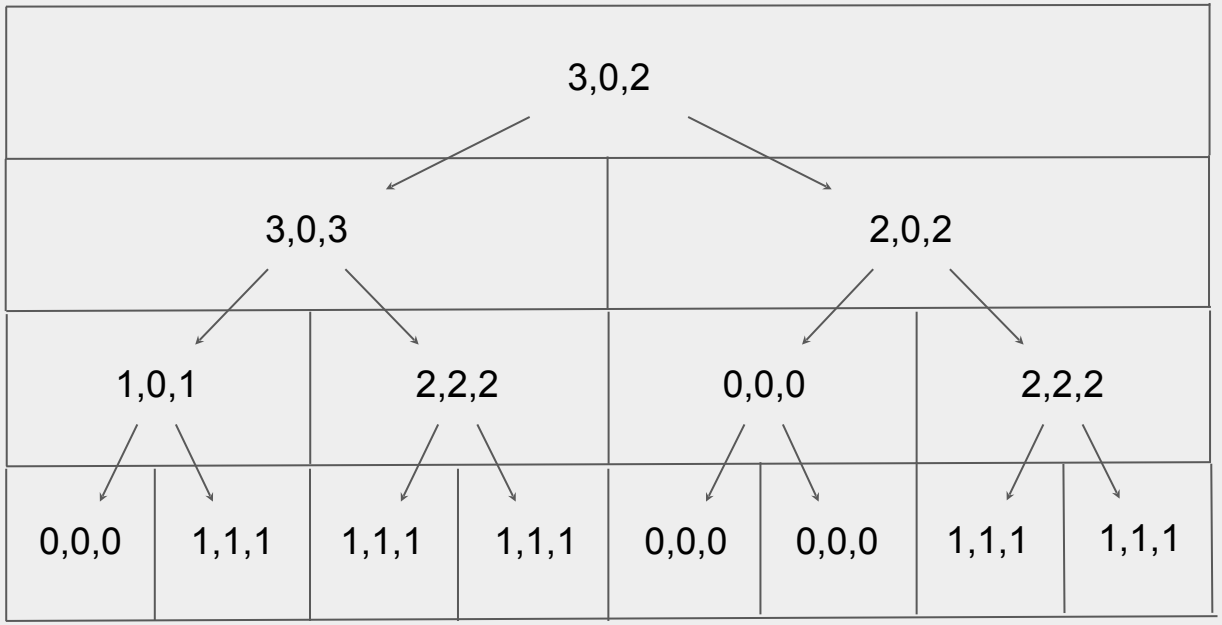
Idea

Let’s create seg_tree with three variables {answer for segment, answer for prefix, answer for suffix}

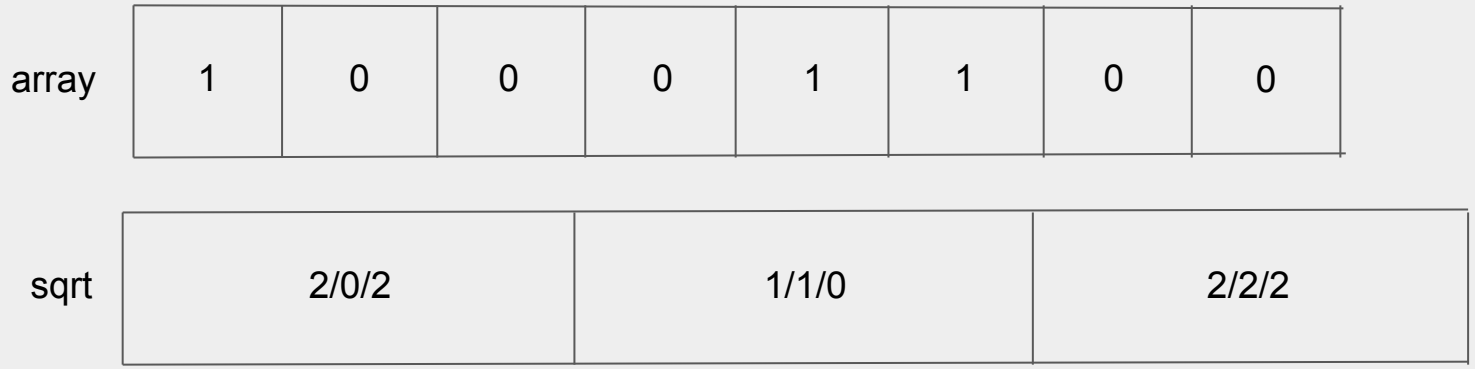
{0, 0, 1, 0, 0, 0, 1, 1, 0}

answer 3, 2, 1

array = [10001100]



array = [10001100]



Idea

son_l = {a, b, c}

son_r = {d, e, f}

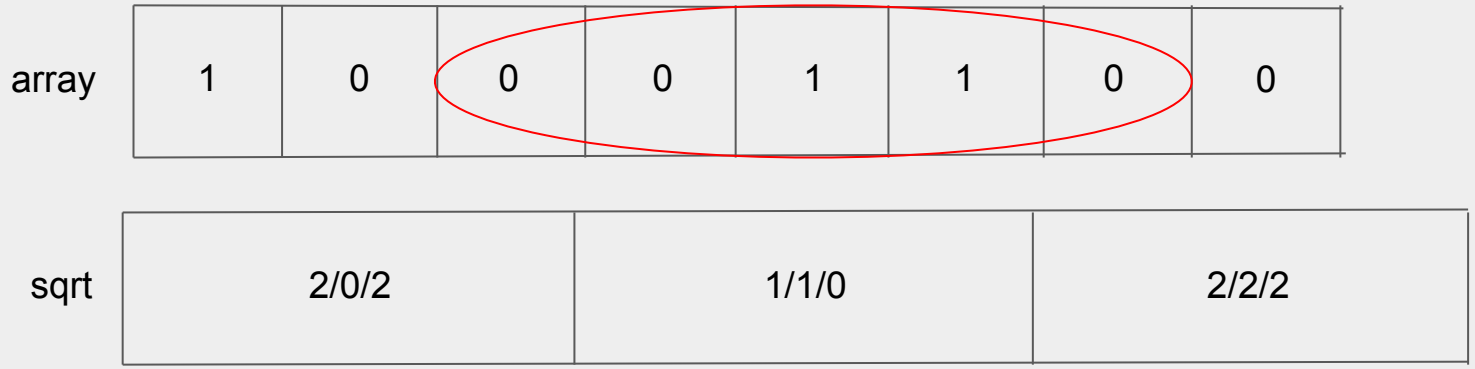
ancestor = {max(a, d, c + e},b + e * (b == n),f + c * (f == n)}

How to solve

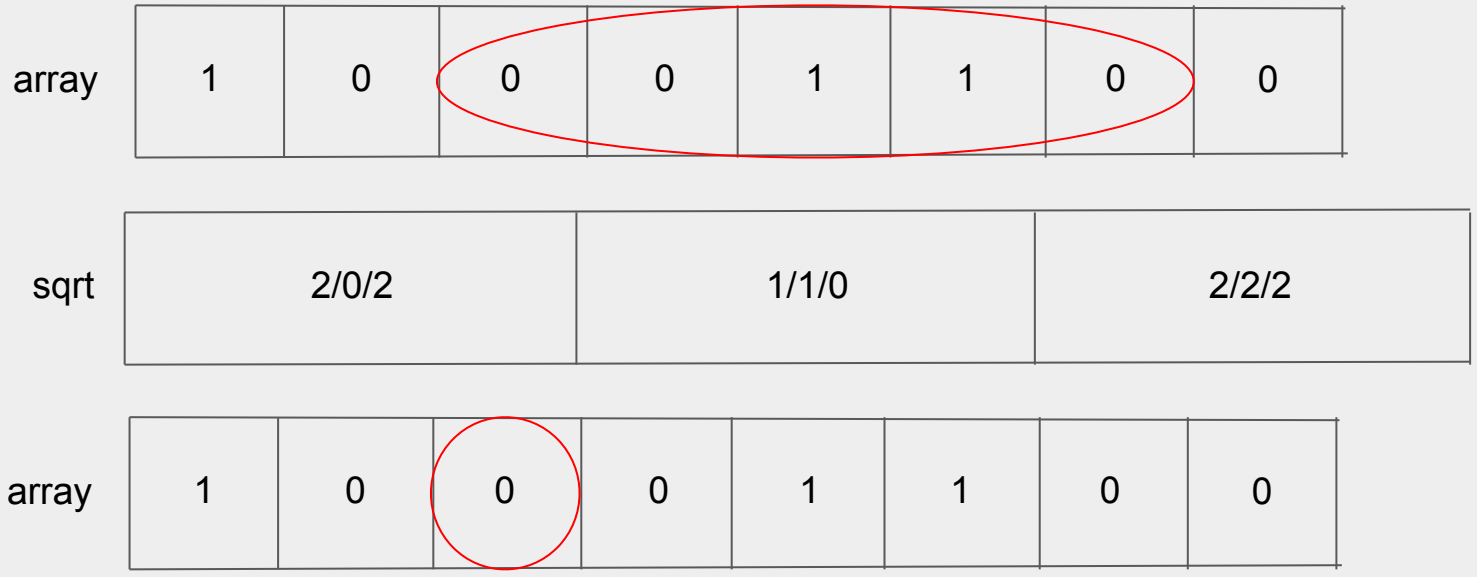
In the result you will have some answers for segments

{a, b, c}, {d, e, f}, {g, h, i} ….

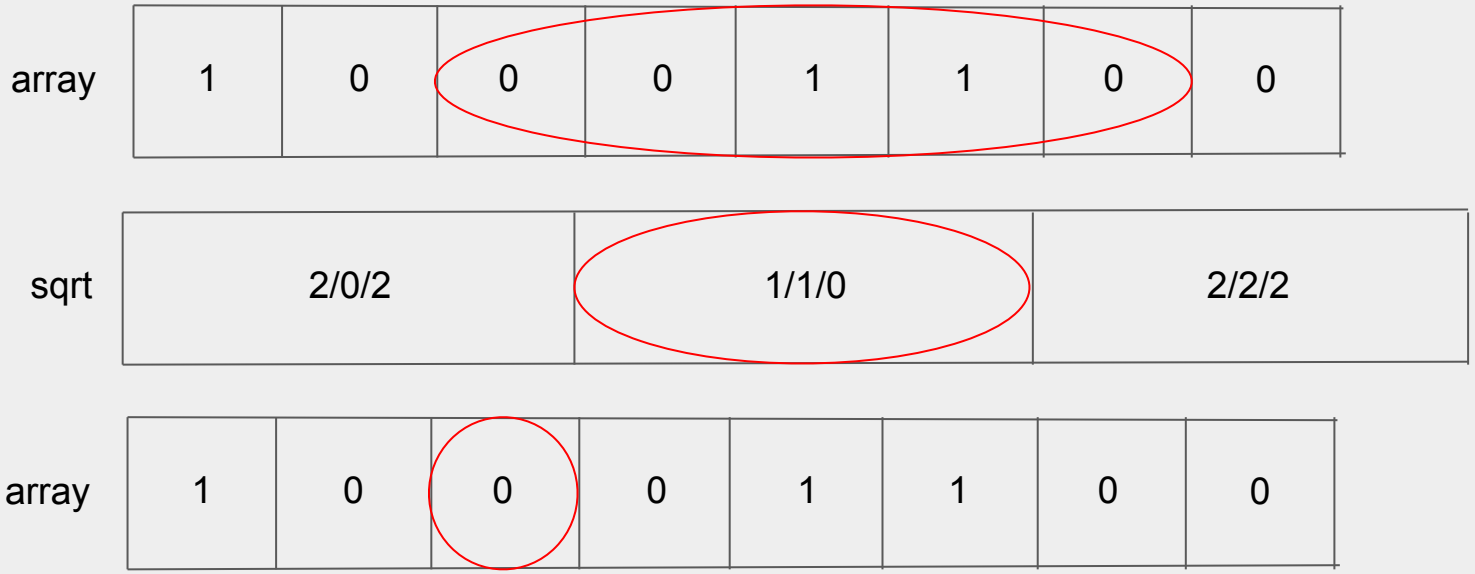
array = [10001100]



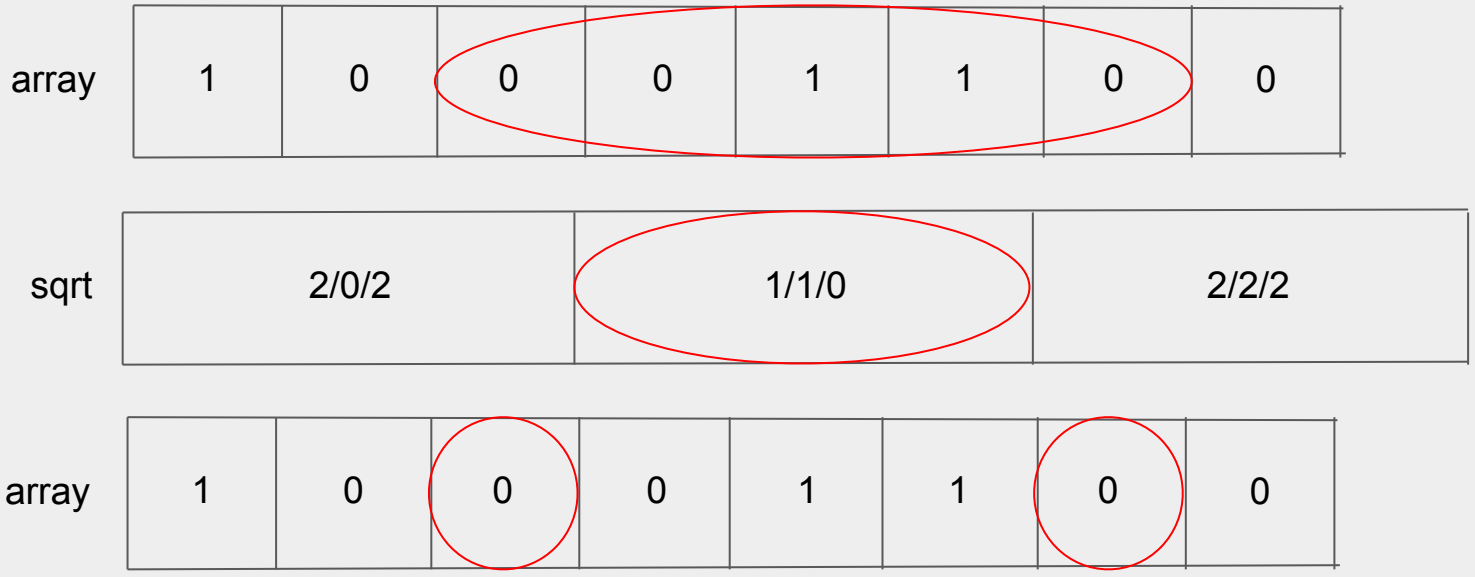
array = [10001100], answer = {1, 1, 1}



array = [10001100], answer = {2, 2, 0}

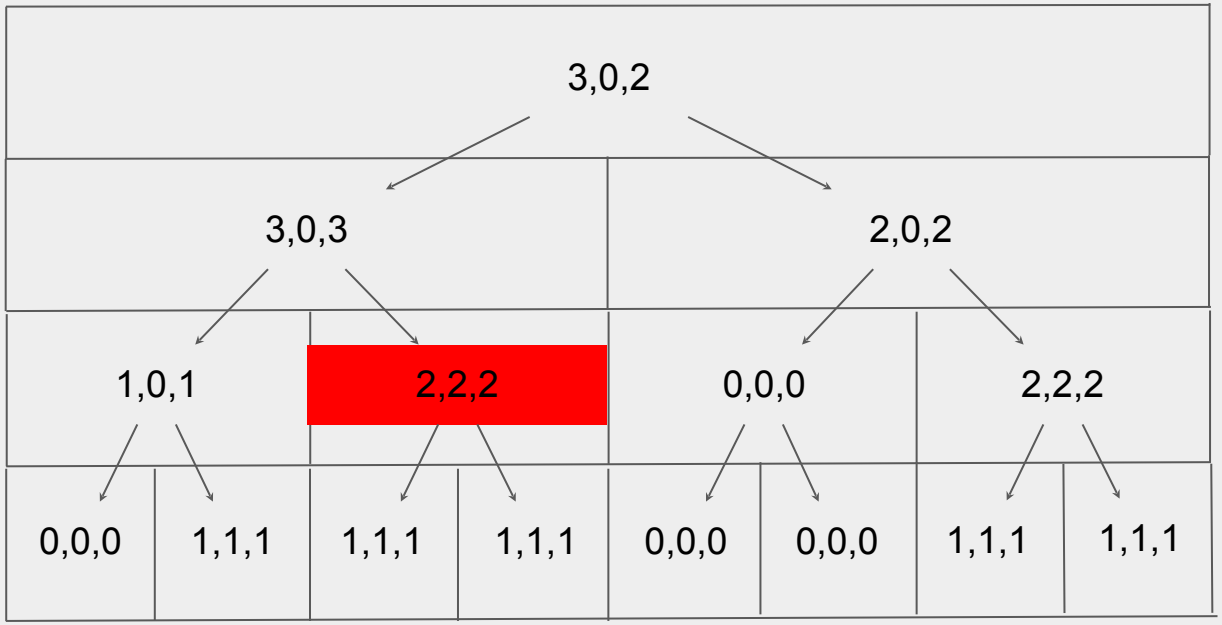


array = [10001100], answer = {2, 2, 1}



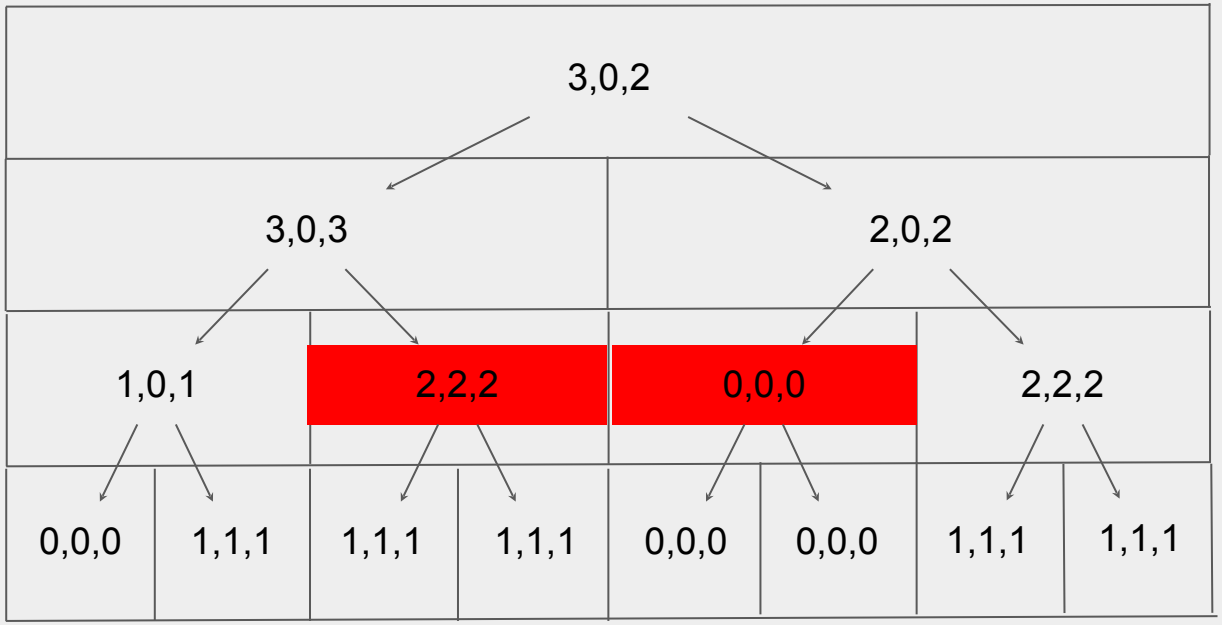
array = [10001100]

array	1	0	0	0	1	1	0	0



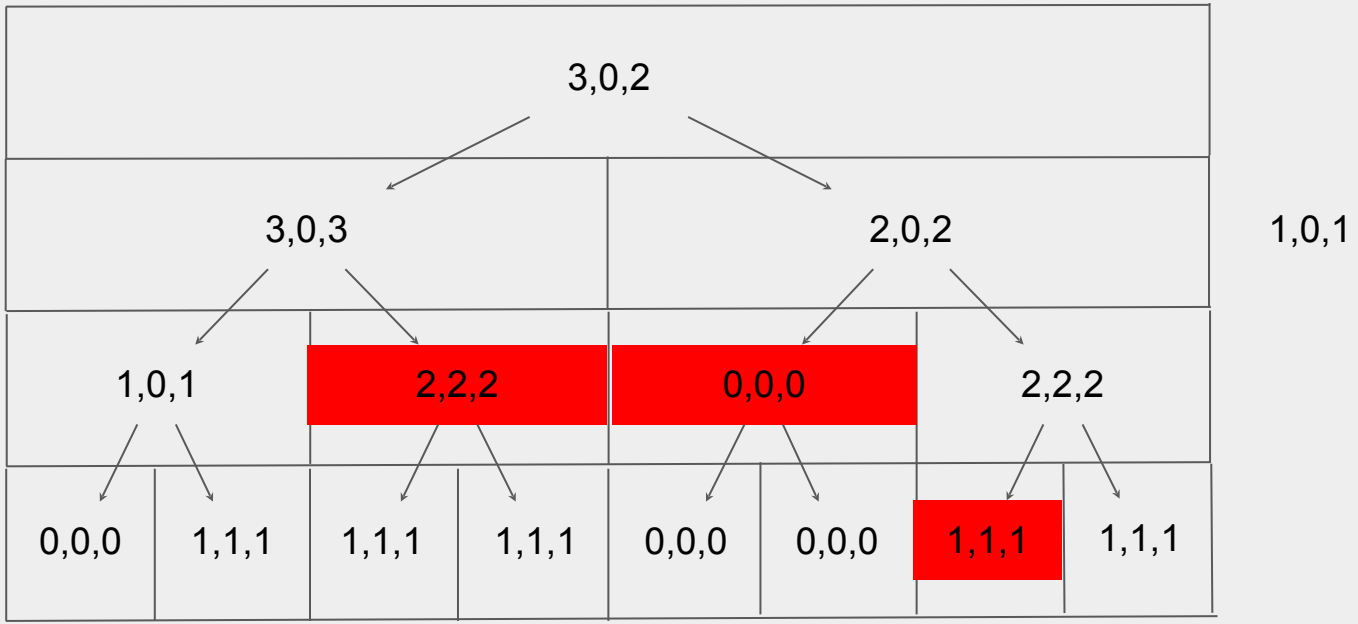
array = [10001100]

array	1	0	0	0	1	1	0	0



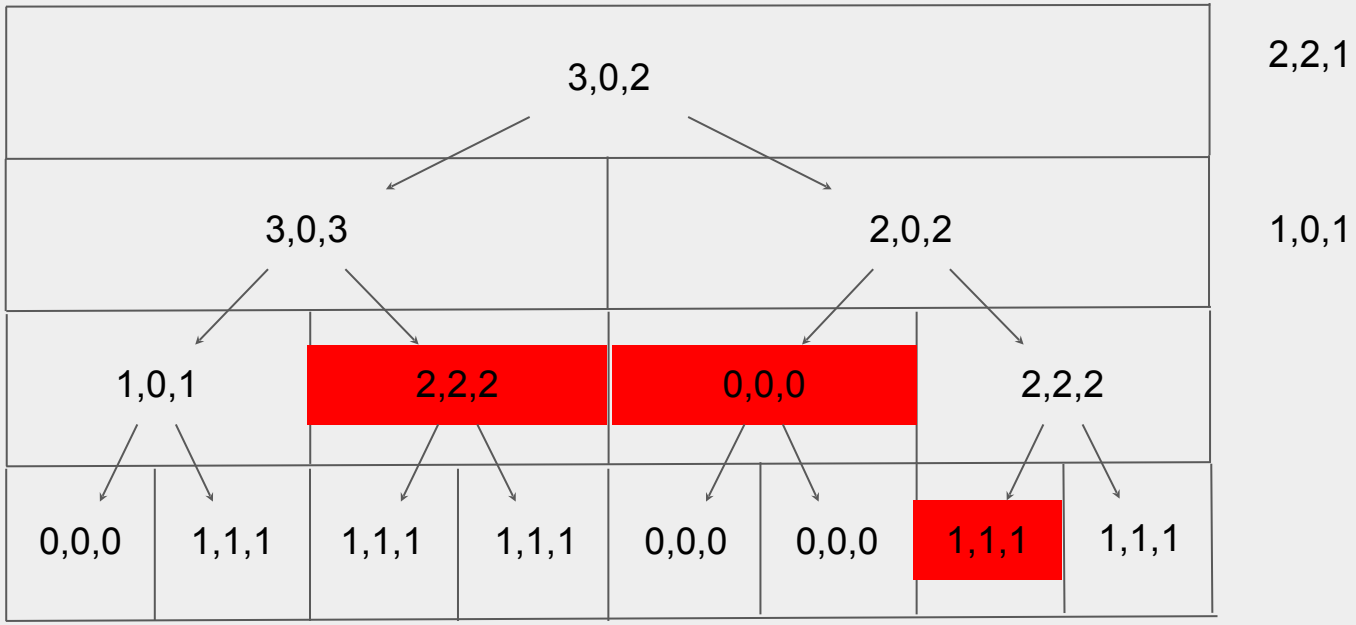
array = [10001100]

array	1	0	0	0	1	1	0	0



array = [10001100]

array	1	0	0	0	1	1	0	0



Task 4

You need to find the index of the k-th occurrence of 0 within the given range. For example, in the sequence '1, 0, 3, 0, 1, 0' the first occurrence of 0 is at the first index (0-based index), and the second one is at the third index.

Hint

original = '1, 0, 3, 0, 1, 0'

now = ‘0, 1, 0, 1, 0, 1’

Hint

To find the k-th zero, you need to find such index id, that sum(0, id) = k and id is the minimal.

original = '1, 0, 3, 0, 1, 0', 1-st zero positions is 1;

now = ‘0, 1, 0, 1, 0, 1’, the minimal id such sum(0, id) = 1 is 1.

Hint

Let’s do binary search.

$l = 0, r = n + 1$

because we have only 0 and 1, $sum(0, l) <= sum(0, r)$ if $l < r$

All codes

sqrt-sum(c++) - https://ideone.com/VoLxn2

sqrt-min(c++) - https://ideone.com/EppWNl

sqrt-count(c++) - https://ideone.com/ugAg7T

seg-tree-sum(c++) - https://ideone.com/6H5LFC

seg-tree-count(c++) - https://ideone.com/HSEKmS

task-1(c++) - https://ideone.com/QKq4kD

task-1 -sqrt(c++) - https://ideone.com/YcebMB