SQRT decomposition

Idea

You may have already encountered a structure known as sqrt decomposition, you might have even seen root decomposition on queries or even the Mo's algorithm. It could have been explained to you in various ways.

Idea

I like the approach to sqrt decomposition as dividing objects into light and heavy and then solving each type separately. Therefore, I will try to explain it from this perspective.

Prime numbers

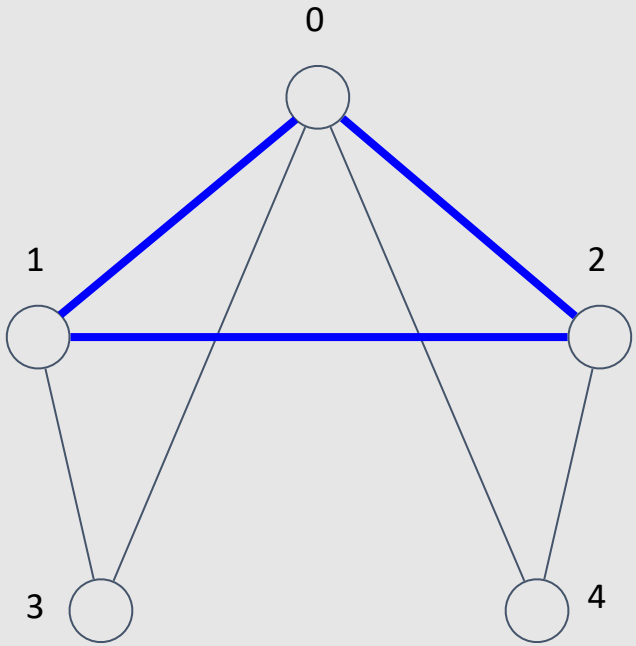
The very first example of such an assumption is prime numbers. When we check numbers, we divide them into 2 sets:

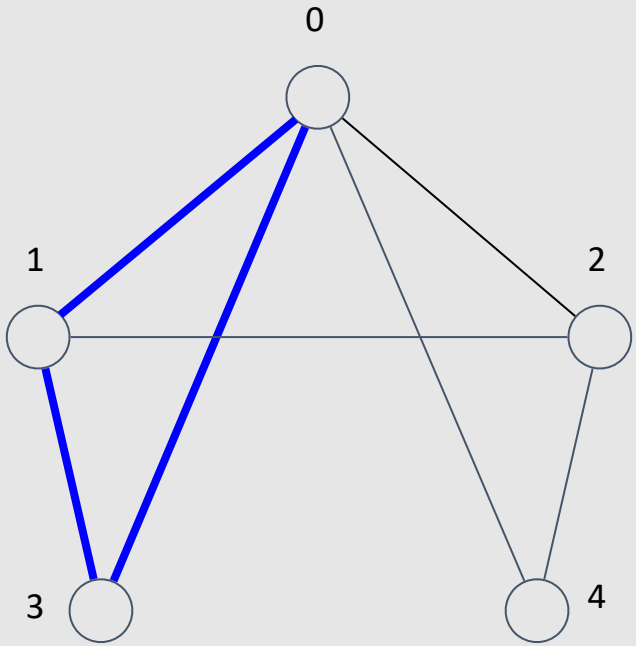
1) light (numbers $< \sqrt{n}$)
2) heavy (numbers $> \sqrt{n}$)

For the light ones, we can simply check each one. For the heavy ones, we prove that they can be ignored.

Triangles

Given a graph with n vertices and m edges. Let's agree, that the graph contains no loops and no multiple edges. The task is to find triangles - cycles of length 3.


0
1
2
3
4


0
1
2
3
4

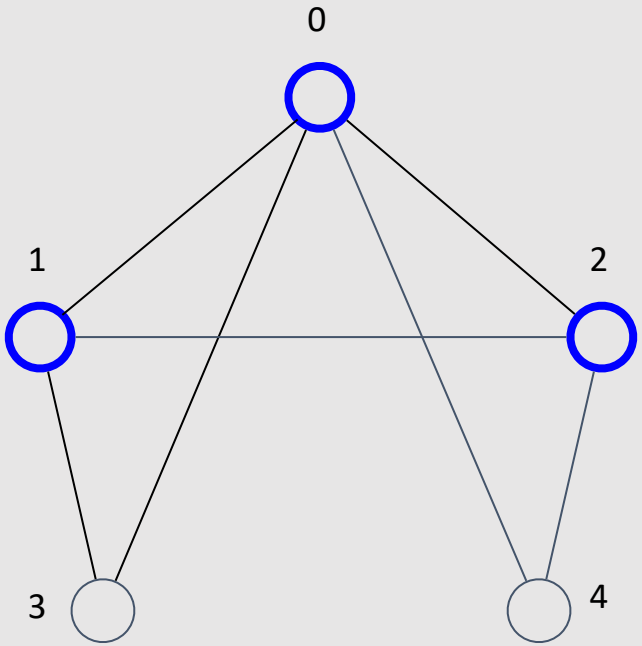
Triangles

First of all, let's divide the vertices into two types:

1) Light - number of neighbors $< \sqrt{m}$
2) Heavy - number of neighbors $> \sqrt{m}$

There can be no more than $2\sqrt{m}$ heavy vertices in the graph because there are a total of m edges, each with two ends, so there can be a total of 2m edges for the vertices.

m = 7, √m = 3, blue - heavy.



Triangles

How many triangles can there be?

Consider 4 cases:

1) LLL
2) LLH
3) LHH
4) HHH

Obviously, all other cases are the same up to isomorphism.

Triangles

LLL:

1) We iterate through the edge in O(m).
2) We iterate through the neighbor of the first vertex and check whether it is connected to the second one in O(√m).

Thus, we have proven that there are no more than O(m√m) triples (LLL) and even learned how to search for them.

Triangles

LLH and LHH:

The same logic applies:

1) We iterate through the edge.
2) We iterate through the neighbors of the light vertex.

Thus, the number of such triples is also $O(m\sqrt{m})$ and we have learned how to search for them.

Triangles

HHH:

Let's just iterate through all the heavy vertices; the number of combinations of triples of heavy vertices is $O(\sqrt{m})^3 = O(m\sqrt{m})$.

You can also iterate through the edge (H, H) and another heavy vertex; this also works in $O(m\sqrt{m})$.

Triangles

Thus, we have proven that the number of triples is $O(m\sqrt{m})$ and even learned how to search for them with such asymptotics, although quite inefficiently.

Triangles

How then to search efficiently and why this does not spoil the asymptotic behavior.

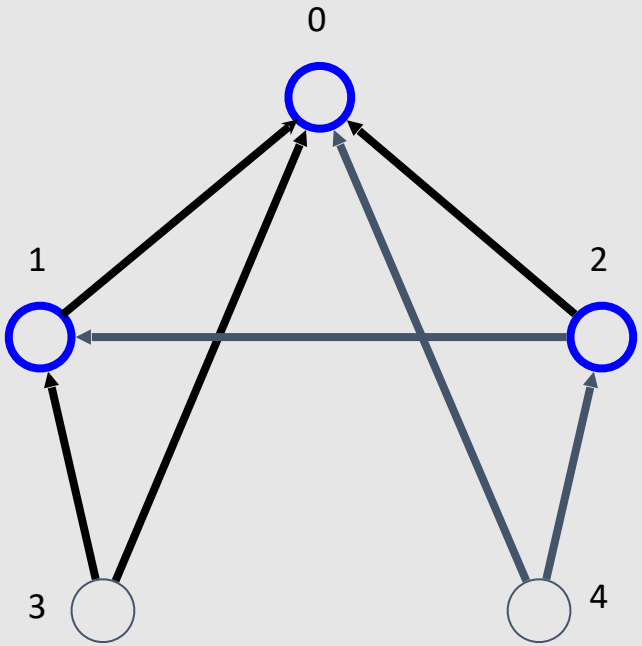
Let's sort the vertices from the vertex with the smallest degree to the vertex with the largest degree.

In this order, we will search for triangles.

Triangles

Then we will only consider triples in the order we described, that is, L->L->L, L->L->H, L->H->H, H->H->H.

As a result, we will obtain an algorithm with $O(m\sqrt{m})$ complexity.


0
1
2
3
4

```cpp
    int n, m;
    cin >> n >> m;
    vector<int> g[n], p(n), pos(n);
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    iota(p.begin(), p.end(), 0); // 0, 1, 2, 3, ...

    auto cmp_size = [&](int a, int b) {
        return g[a].size() < g[b].size() || (g[a].size() == g[b].size() && a < b);
    };

    sort(p.begin(), p.end(), cmp_size);
    for (int i = 0; i < n; i++) {
        pos[p[i]] = i;
    }

    auto cmp_p = [&](int a, int b) {
        return pos[a] > pos[b];
    };

    for (int v = 0; v < n; v++) {
        sort(g[v].begin(), g[v].end(), cmp_p);
        while (g[v].size() > 0 && pos[g[v].back()] < pos[v]) {
            g[v].pop_back();
        }
        reverse(g[v].begin(), g[v].end());
    }
```

```
    vector<bool> is_linked(n, false);
    int ans = 0;

    for (int v : p) {
        for (int w : g[v]) {
            is_linked[w] = true;
        }
        for (int u : g[v]) {
            for (int w : g[u]) {
                if (is_linked[w]) {
                    ans++;
                    cout << v << " " << u << " " << w << "\n";
                }
            }
        }
        for (int w : g[v]) {
            is_linked[w] = false;
        }
    }

    cout << ans << "\n";
    return 0;
}
```

Success #stdin #stdout 0.01s 5280KB

stdin

```
5 7
0 1
0 2
0 3
0 4
1 2
1 3
2 4
```

stdout

```
3 1 0
4 2 0
1 2 0
3
```

Strings

Let's divide string into light and heavy. Suppose the total length of the text is t. Then:

1) Light strings - strings of size $< \sqrt{t}$
2) Heavy strings - strings of size $> \sqrt{t}$

Facts

One of the useful examples of such a division:

In a text of length t, there are no more than $\sqrt{t}$ different lengths of strings.

There are two proofs - arithmetic progression and using sqrt decomposition.

We will use the second option.

For light strings, there are only $\sqrt{t}$ different lengths, for heavy ones there cannot be more than $\sqrt{t}$ different lengths, since $\sqrt{t} * \sqrt{t} = t$.

Example

I went to school yesterday with my best friend

t = 37.

6 * c length

|string| < 6 - light

|string| > 6 - heavy

at most sqrt(t) heavy string, sqrt(t) * sqrt(t) = t

at most sqrt(t) length of light strings, because only [0, .., sqrt(t)]

Knapsack

Actually, this fact is useful not only in strings but also, for example, in a knapsack.

It turns out that by the same logic, it can be proven that there can be no more than $O(\sqrt{W})$ different weights in a knapsack (W - the total weight of the items).

Example

we have W = 5 to fill and objects = [1, 2, 3, 7], so we can if we take [2, 3]

Task

You have n items with weights w_i.

The task is to check if it's possible to pack a knapsack of weight W.

Solution

We have proven that there can be no more than O(√W) different weights, therefore now we can calculate how many items there are of each weight.

Suppose we have k items of weight w_i, then we replace them with a set of items [w_i, w_i * 2, .. , w_i * (k + 1 - 2^t)), where (2^(t + 1) - 1) <= k

Solution

[w_i, w_i * 2, .. , w_i * (k + 1 - 2^t)), where (2^(t + 1) - 1) <= k

w_i * 2^0, w_i * 2^1 … w_i * 2^j = w_i * (2^0 + 2^1 .. + 2^j) =

(2^(j + 1) - 1) * w

7w - [w * 2^0, w * 2^1, w * 2^2, 0 * w]

6w - [w * 2^0, w * 2^1, w * 3]

Example

Suppose we have 7 items of weight w, then we replace them as follows:

[w, w, w, w, w, w, w] -> [w, 2w, 4w]

If there are 6 items of weight w, then we get:

[w, w, w, w, w, w] -> [w, 2w, 3w]

Asymptotic

Thus, we have replaced n items with $O(\sqrt{W}\log n)$ items (it can even be proven to be $O(\sqrt{W})$ items), and therefore obtained an algorithm with a complexity of $O(W\sqrt{W})$ instead of $O(Wn)$.

SQRT decomposition

Let’s divide array to blocks.

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

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

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

…

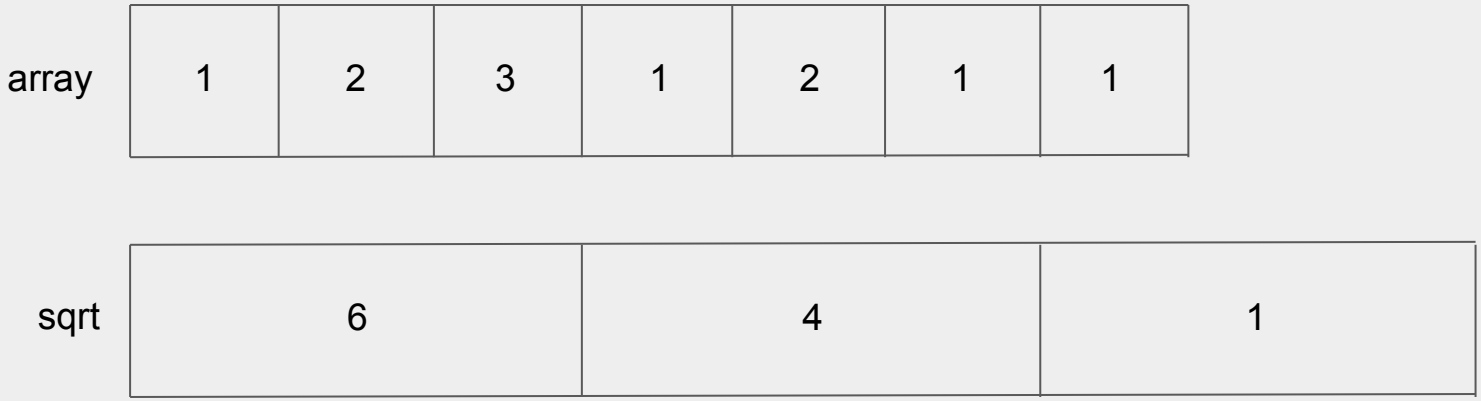
a[(n // c) * c..(n // c + 1) * c - 1] - last

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

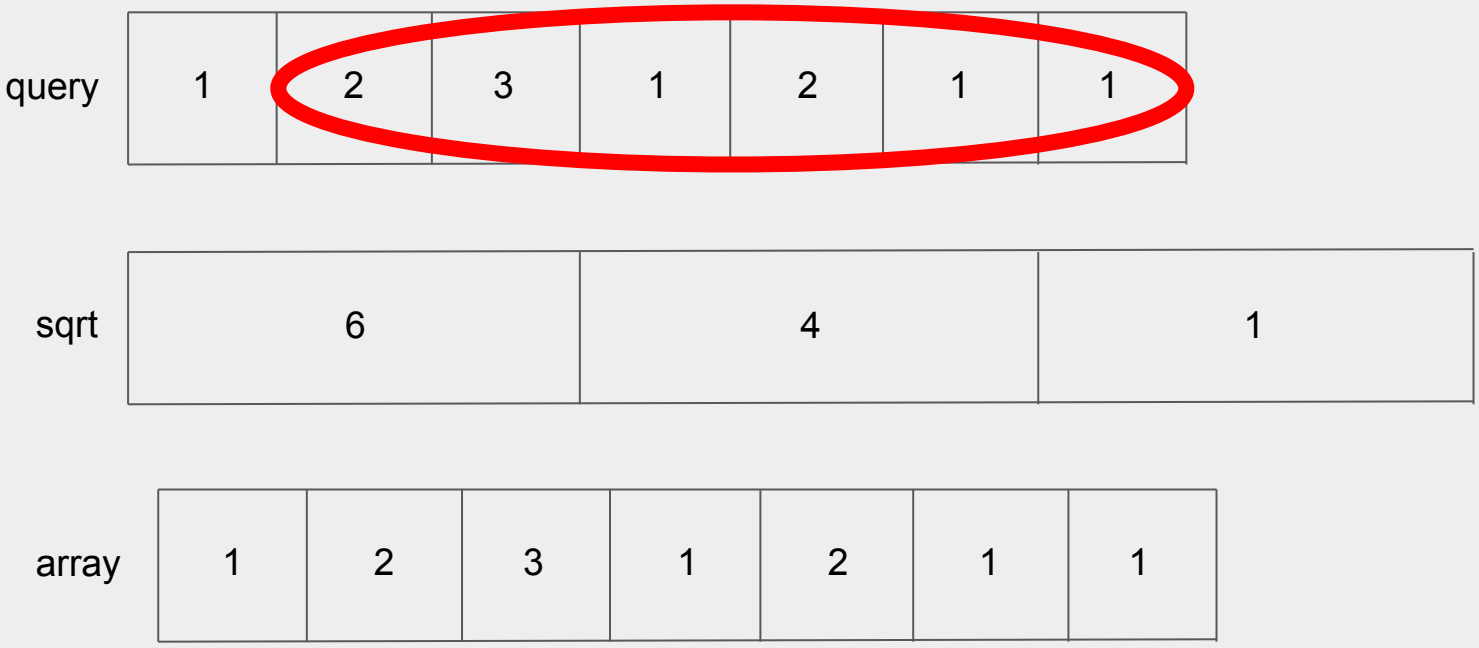


SQRT decomposition

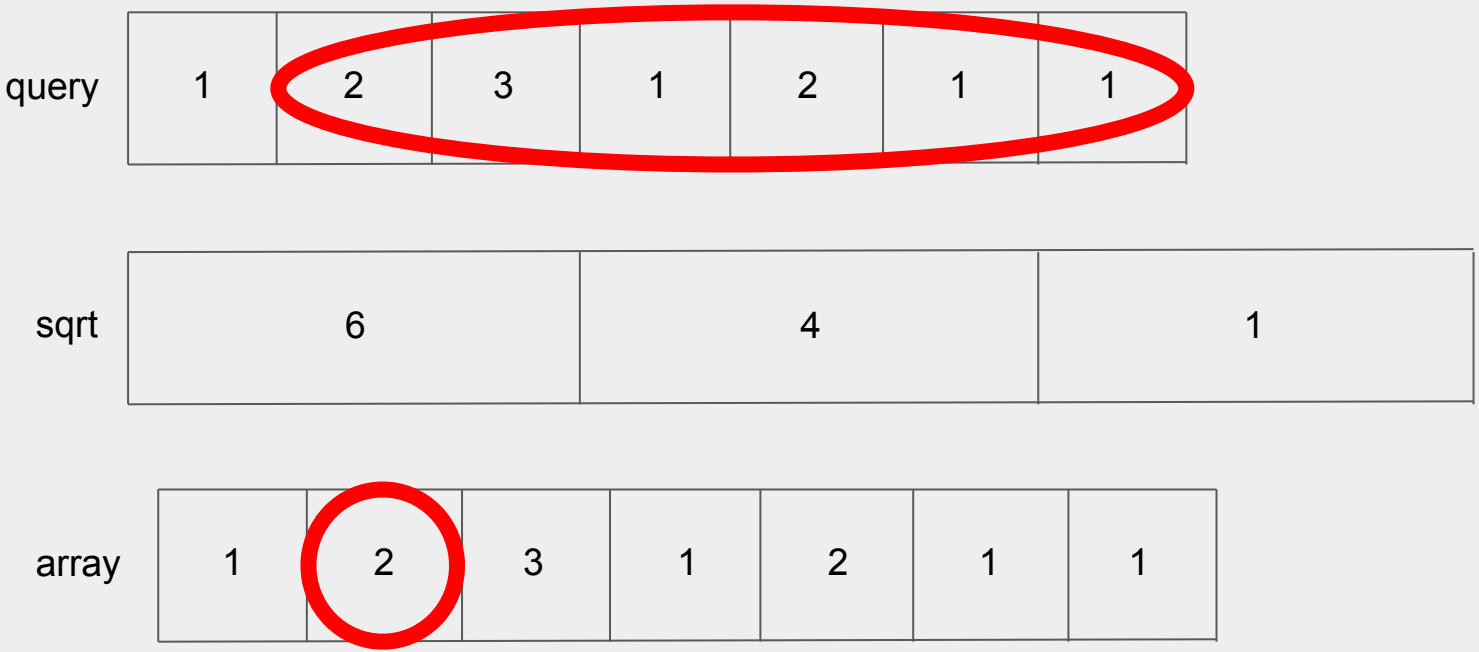
How do we respond to a query for the sum in a segment now?

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.

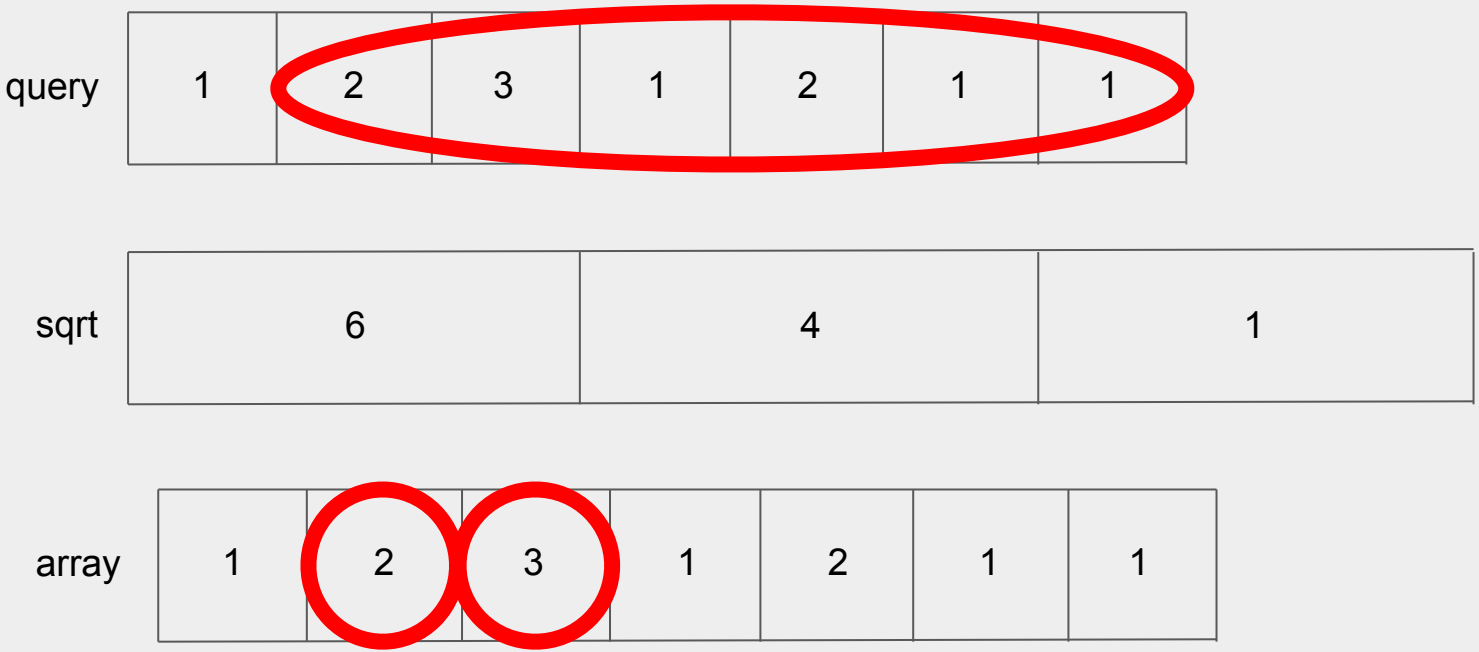
c = 3, sum = 0



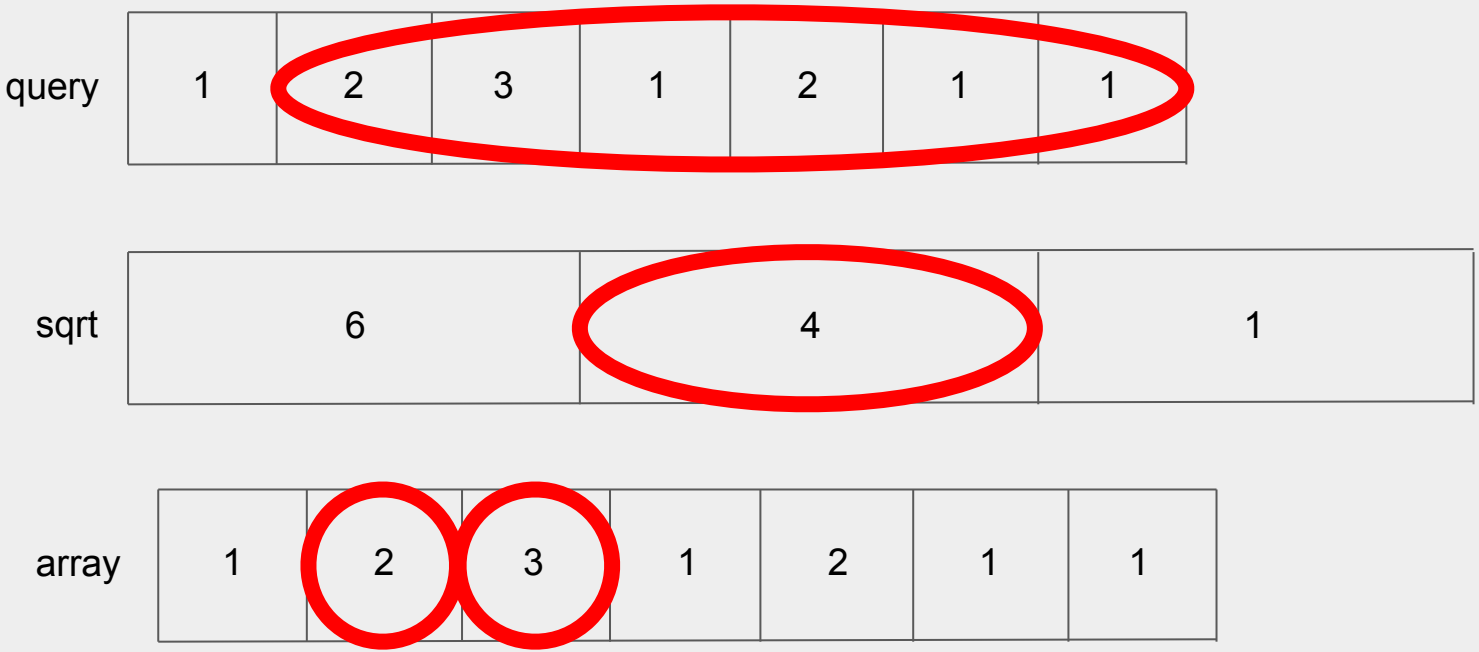
c = 3, sum = 2



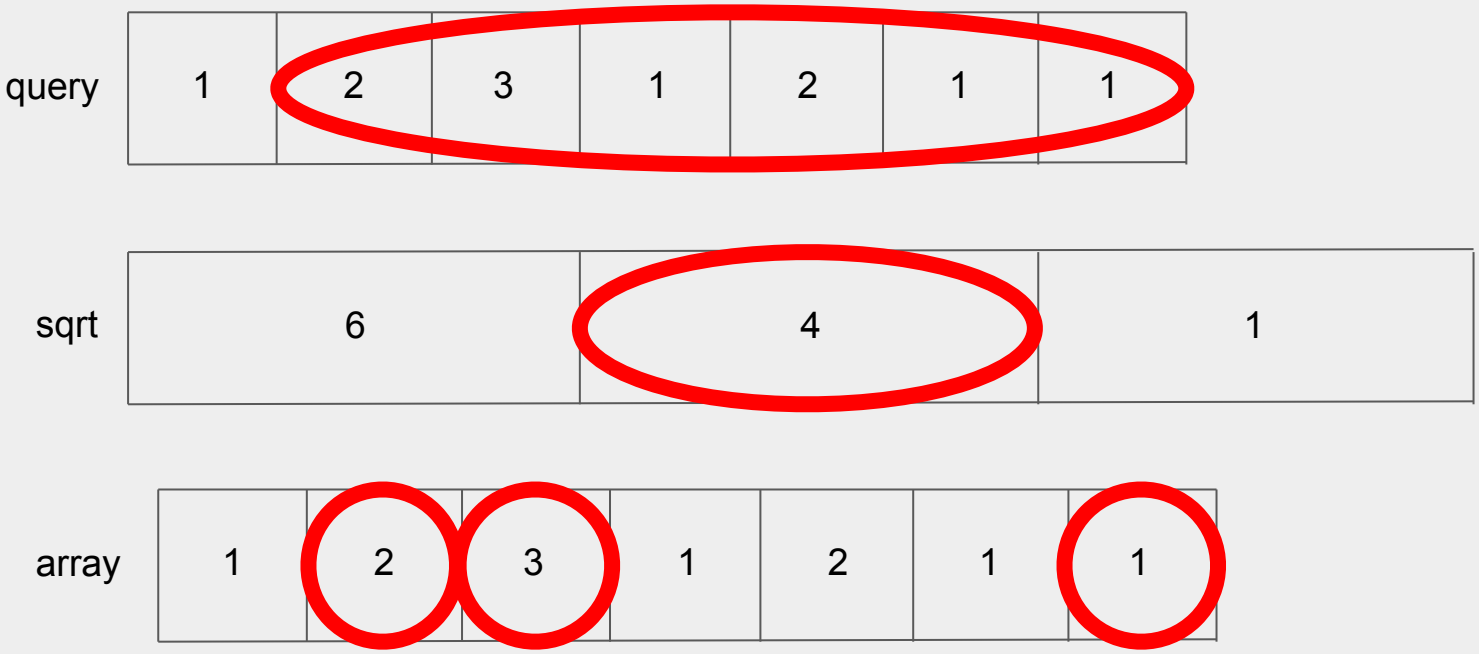
c = 3, sum = 5



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)

```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
```

Lazy propagation

Let's maintain for each block the value by which it increased, while updating the sum for the block itself in an honest manner.

Lazy propagation

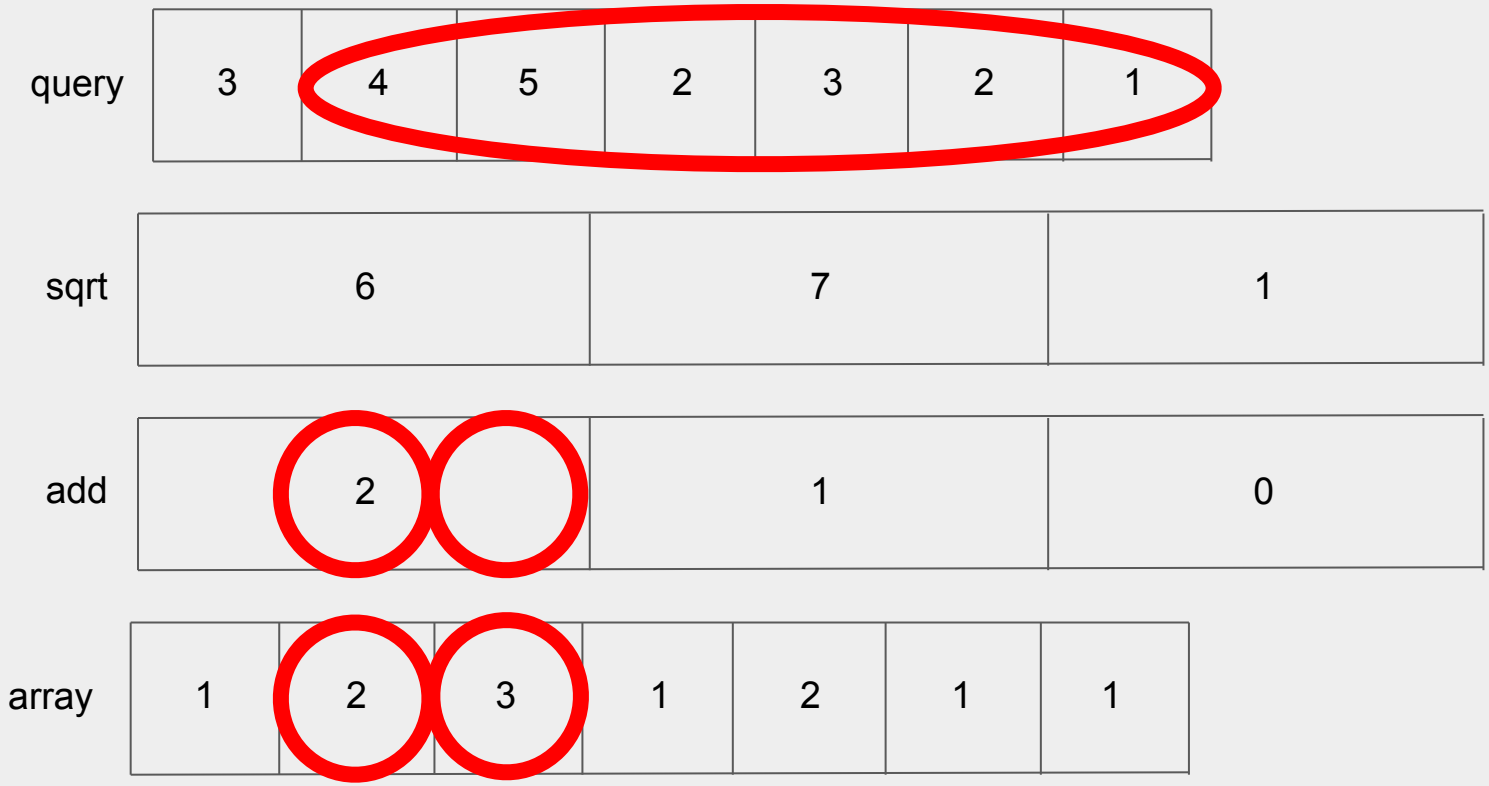
What should we do now if we access just one element in a certain block?

Let's summarize the block's addition with all values in that block.

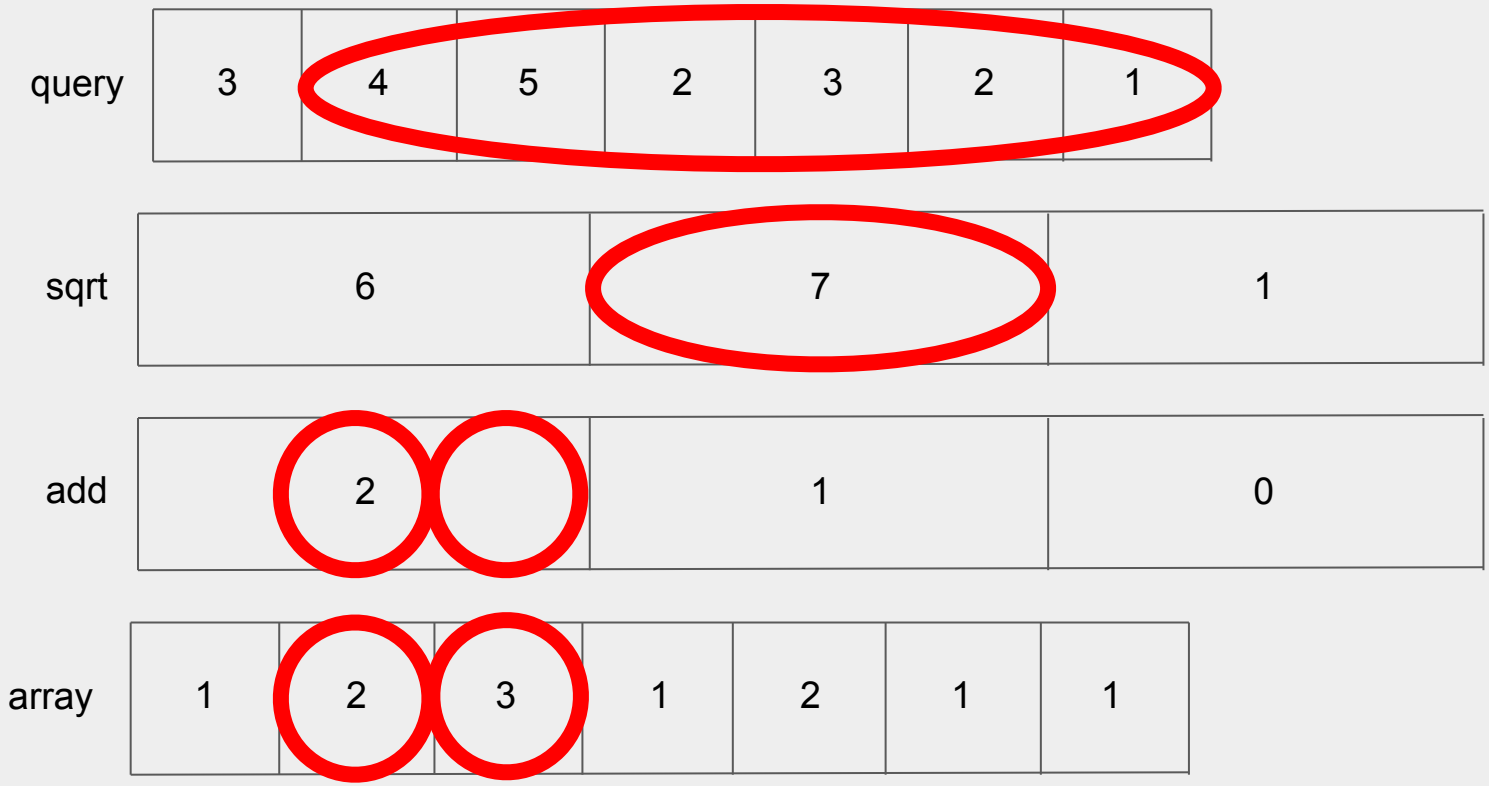
c = 3, sum = 0

c = 3, sum = 4

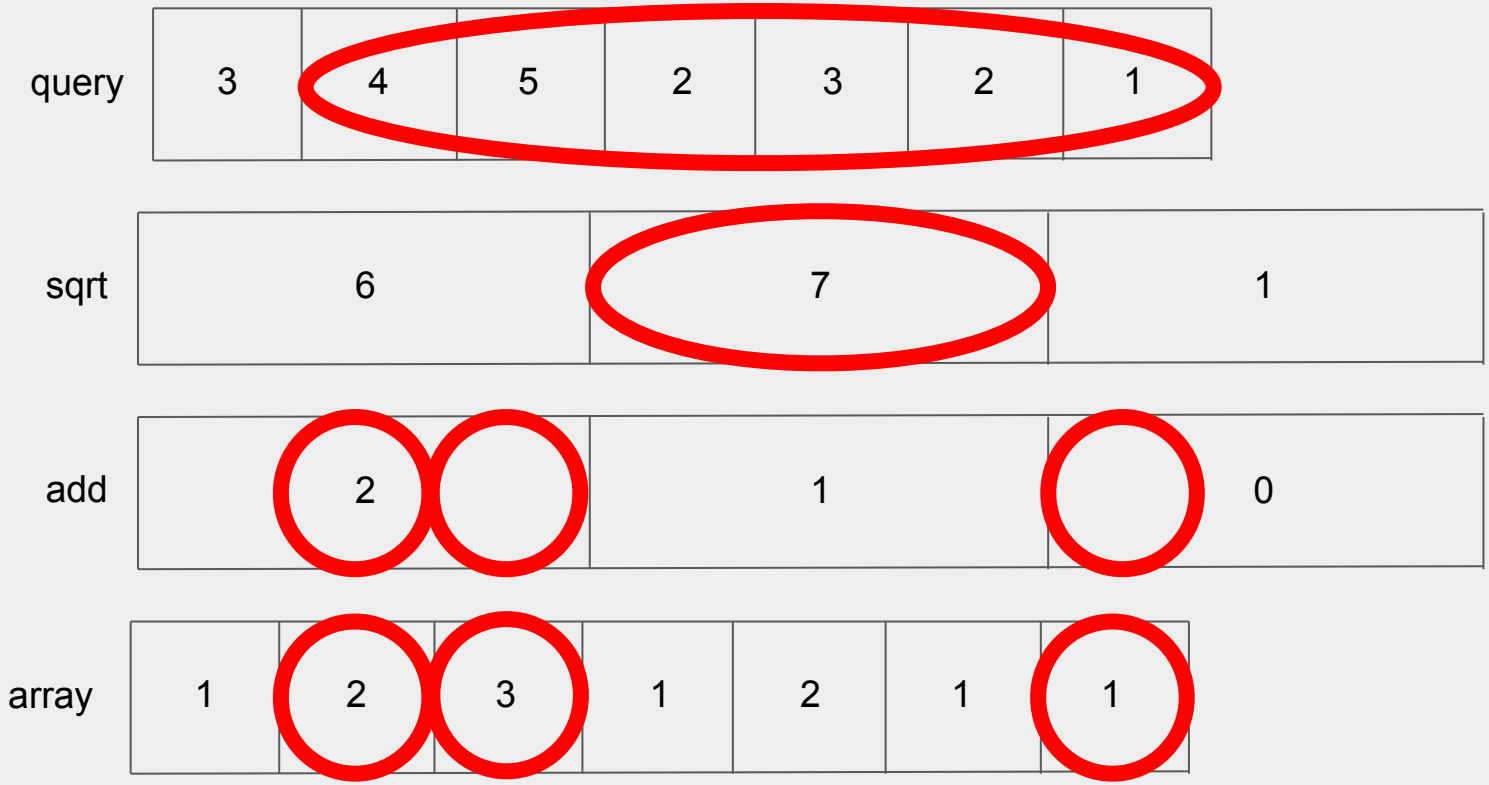
c = 3, sum = 9



c = 3, sum = 16



c = 3, sum = 16



```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
```

More

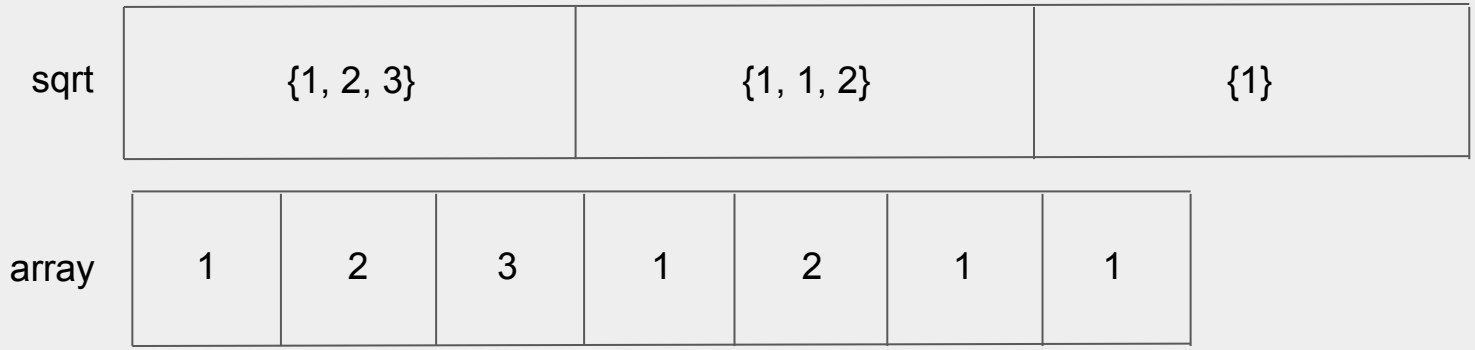
The number of elements less than X in a segment.

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



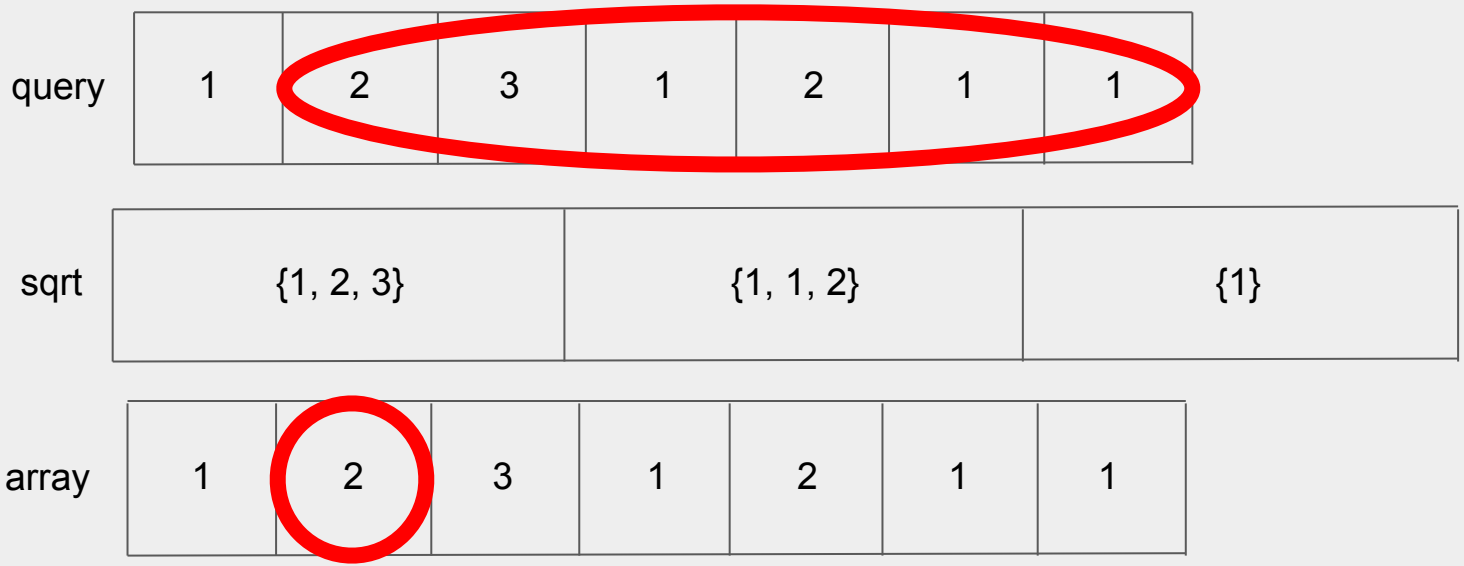
```
8.  struct Block {
9.      vector<int> elems;
10.     int add = 0;
```

```
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

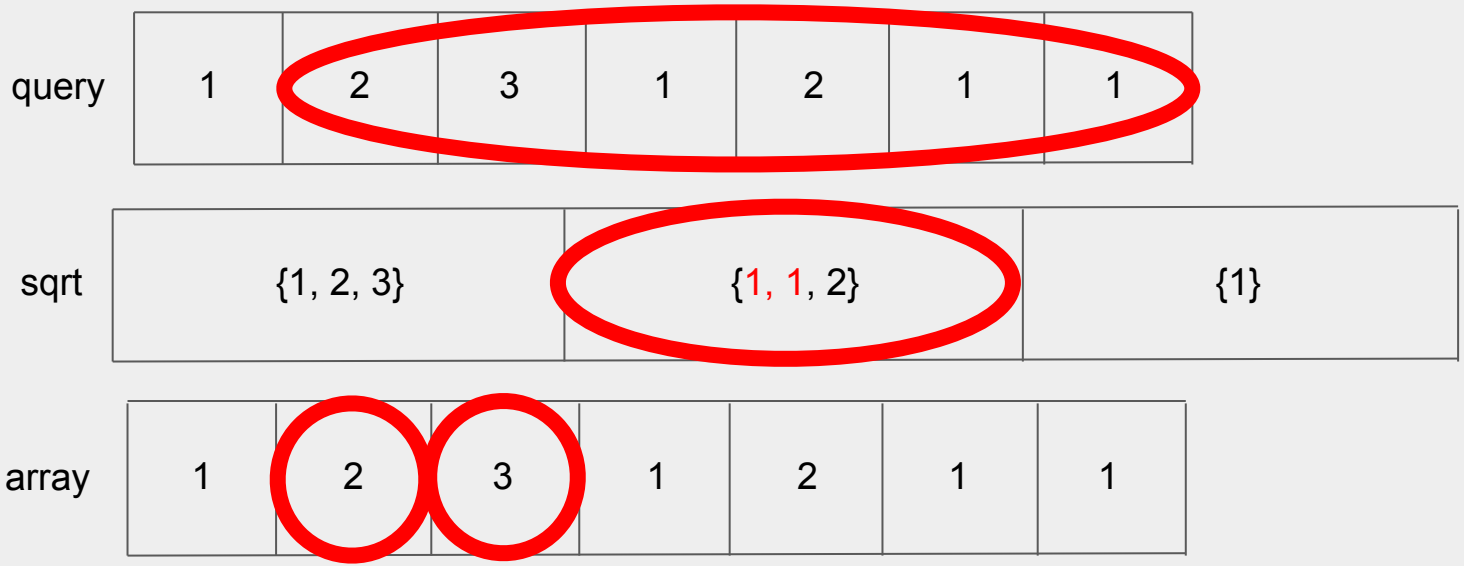
1. For a single element, we directly compare this element with X.
2. For the block, we make a binary search and use it to find the count of the required elements.

c = 3, X < 2, answer = 0

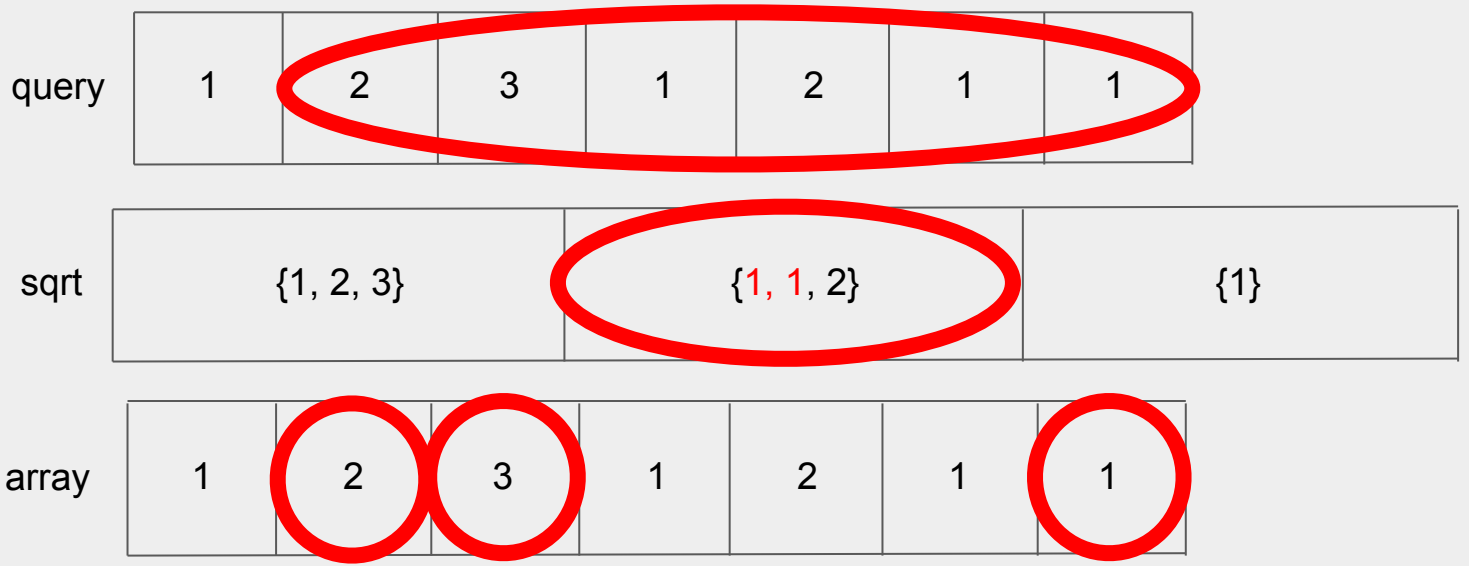


c = 3, X < 2, answer = 0

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

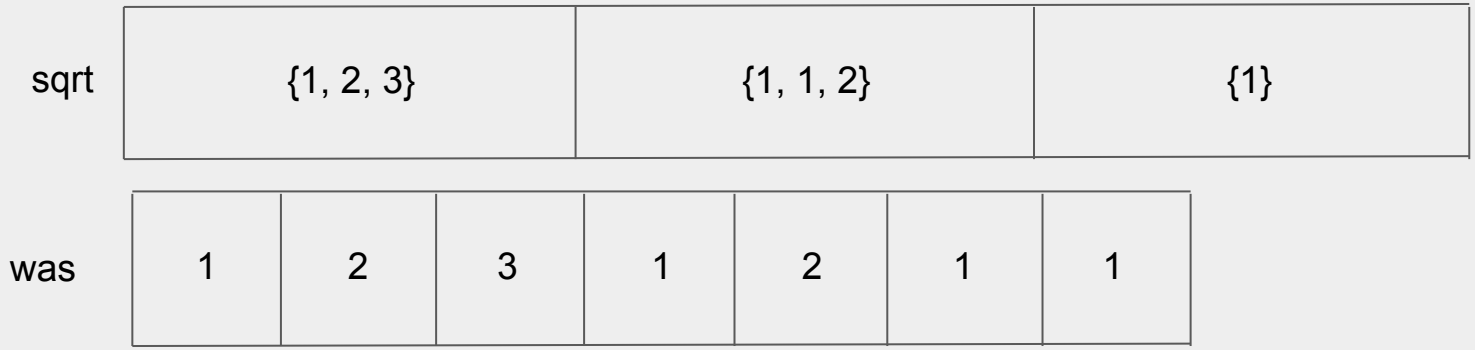
1. For a single element - O(1), thus, overall - O(c) per query.
2. For a block - O(log(c)), meaning O(n/c*log(c)) per query.

Harder

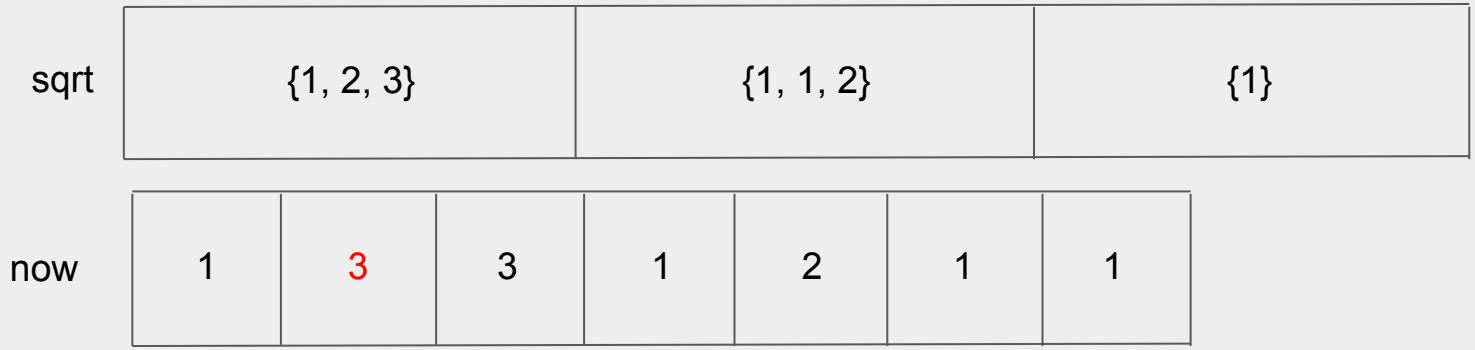
Let's add an update to the element.

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

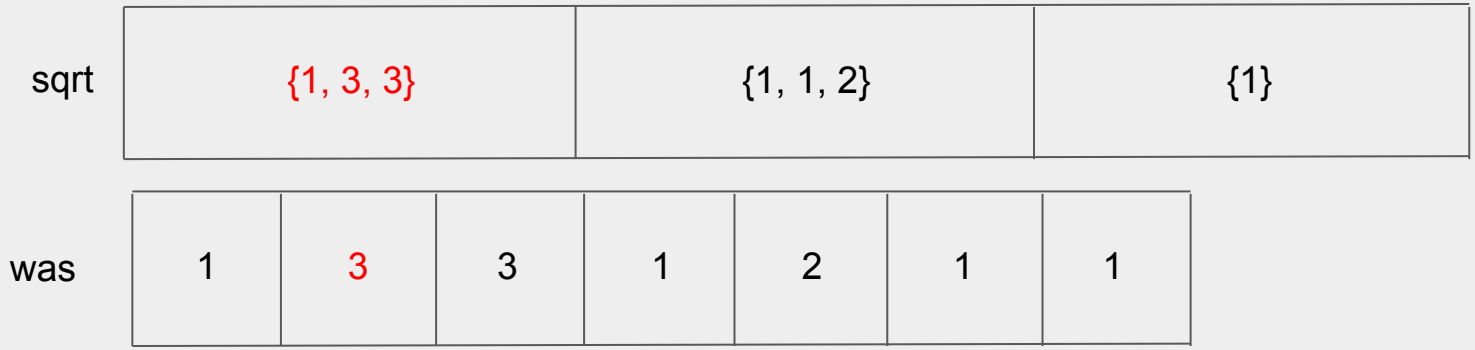
a[1] = 3



a[1] = 3



a[1] = 3



Asymptotic

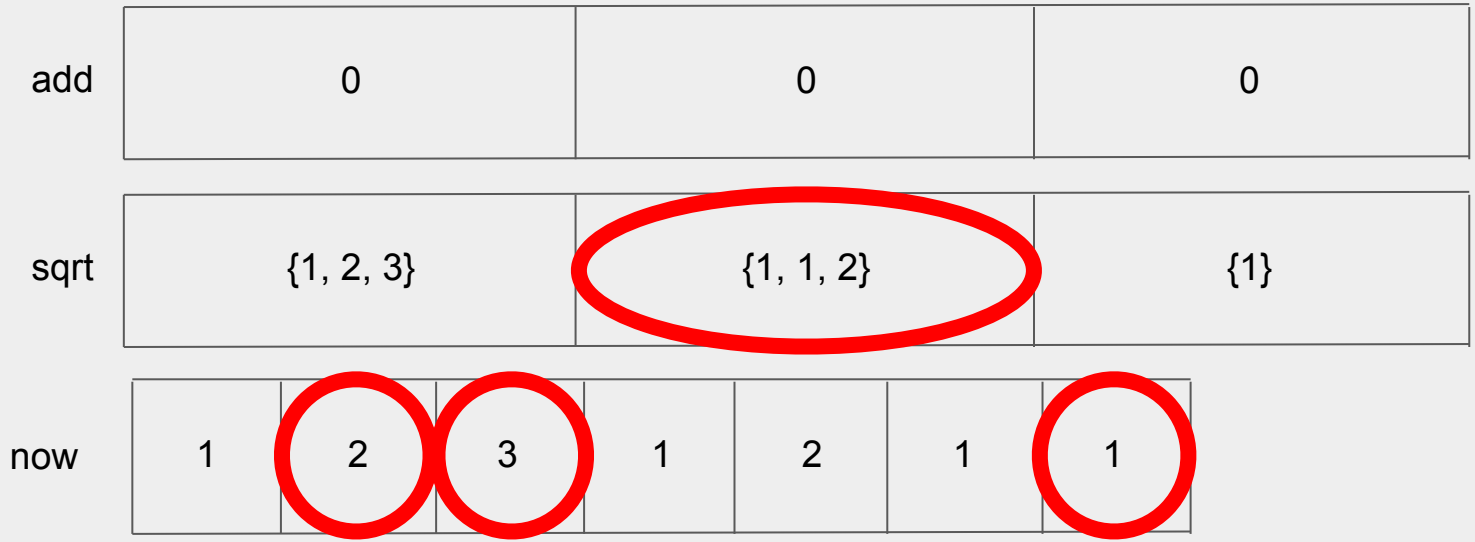
Resorting the block will operate in O(clog(c)) because its size is c.

Harder

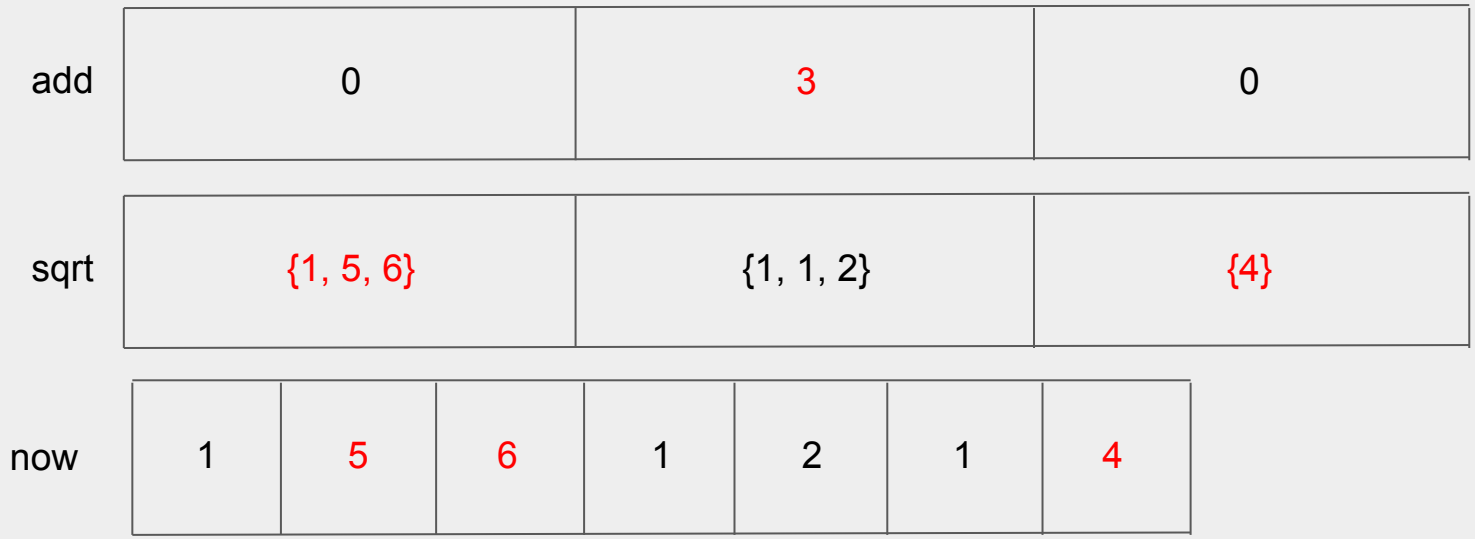
What if we want to update an entire segment?

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

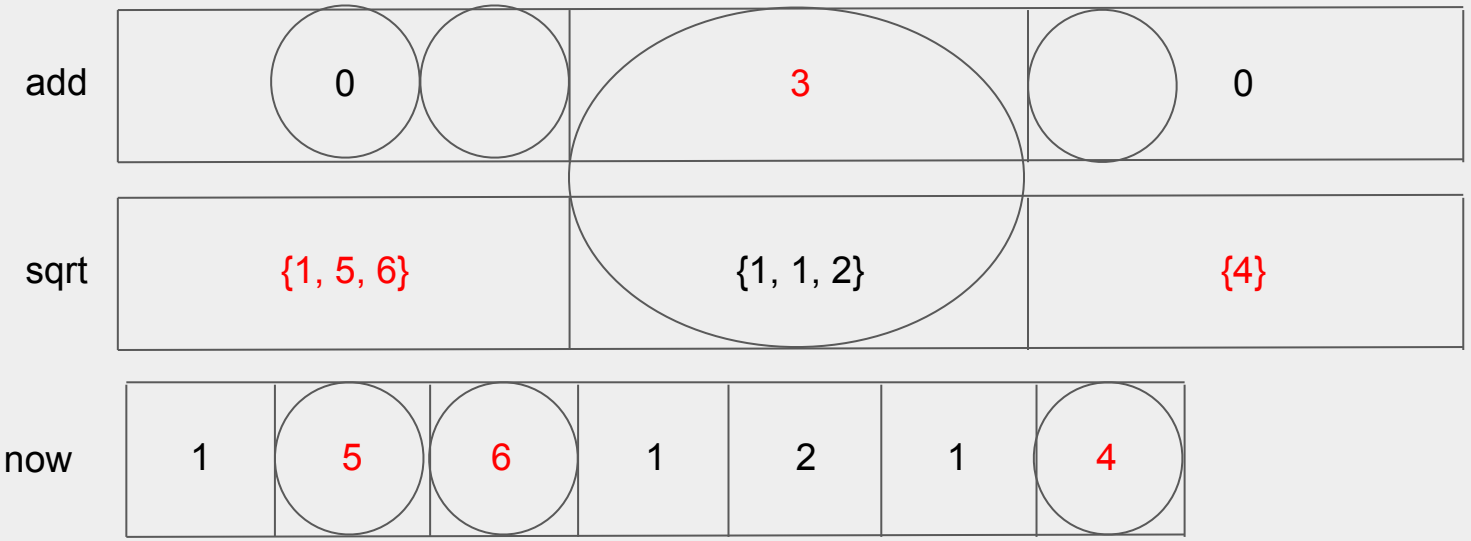
[1, 6] += 3



[1, 6] += 3



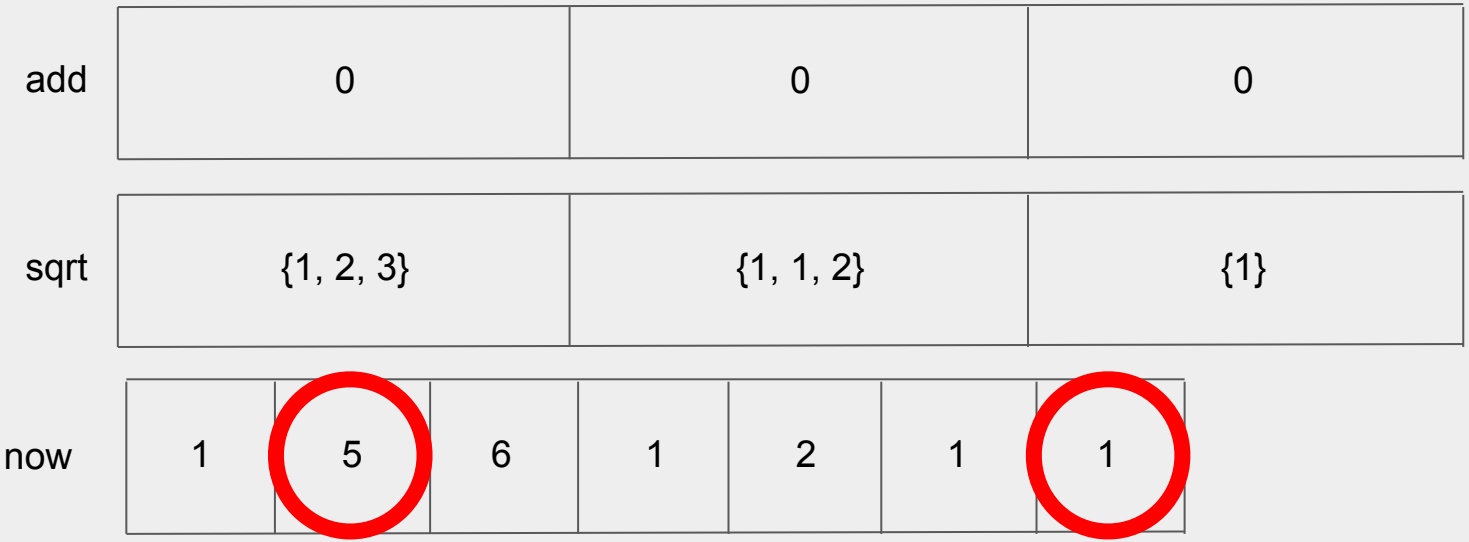
[1, 6] ?= 2

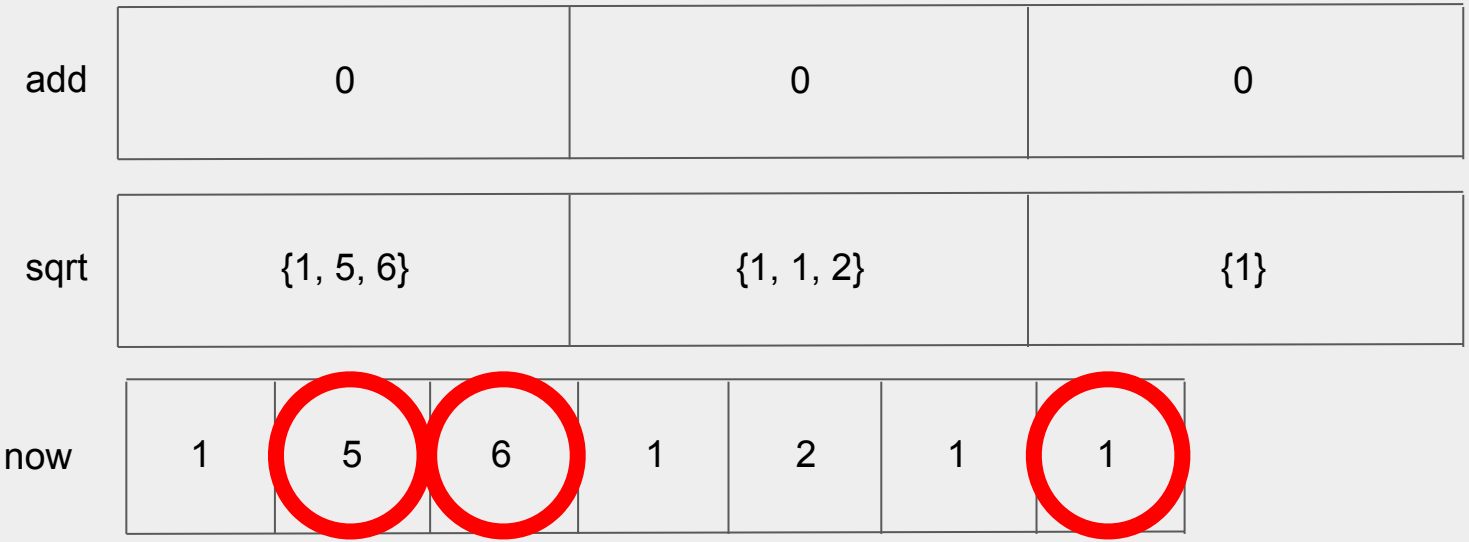


Problems

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.


add
0
0
0
sqrt
{1, 2, 3}
{1, 1, 2}
{1}
now
1
5
6
1
2
1
1


add
0
0
0
sqrt
{1, 5, 6}
{1, 1, 2}
{1}
now
1
5
6
1
2
1
1

What is c?

Usually c = √(n), but

1.√(n) is not precise, so better to make a const from it.

2.counting √(n) is not O(1).

3.√(n) is not cache-friendly

4.sometimes you have better values for c

What is c?

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

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

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

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

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

Query decomposition.

Let's now divide not the array into blocks of size $\sqrt{n}$, but the queries into blocks of size $\sqrt{q}$.

At the beginning of each block, we want to have the ability to quickly respond to all queries for finding the answer on a segment.

At the same time, we want to be able to answer for a single adding query quickly.

Query decomposition.

Suppose we have two types of queries:

1) Find the sum on a segment.

2) Make an addition on a segment.

Query decomposition.

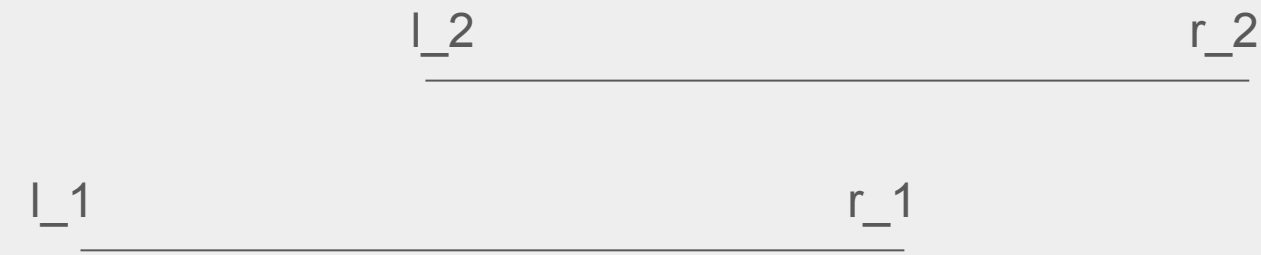
Suppose we are at the beginning of some block and we have a calculated prefix sum with which we can quickly find the sum on a segment.

Suppose also that there were COUNT update queries within the block.

How then do we respond to a query in such a case?

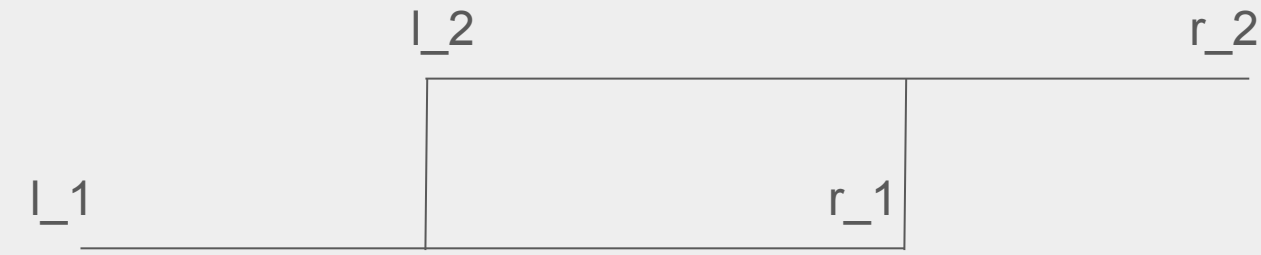
Query decomposition.

Suppose we have an update query on the segment [l_1, r_1] by x and a sum query on the segment [l_2, r_2].


l_2
r_2
l_1
r_1

Query decomposition.

Then, to the sum query, we need to add updates for the intersection of these segments, that is, for max(0, min(r1, r2) - max(l1, l2) + 1) elements, meaning we need to add max(0, (min(r1, r2) - max(l1, l2) + 1) * x).


l_2
r_2
l_1
r_1

Query decomposition.

And how do we update the prefix sums?

Using prefix updates, for example, or scanline.

ar	1	5	6	1	2	1	4
pi	1	6	12	13	15	16	20
add	0	0	0	0	0	0	0

[1, 3] += 3

ar	1	5	6	1	2	1	4
pi	1	6	12	13	15	16	20
add	0	+3	0	0	-3	0	0

[3, 5] += 4

ar	1	5	6	1	2	1	4
pi	1	6	12	13	15	16	20
add	0	3	0	4	-3	0	-4

after block

ar	1	5	6	1	2	1	4
pi	1	6	12	13	15	16	20
add	0	3	3	7	4	4	0

after block

ar	1	8	9	8	6	5	4
pi	1	6	12	13	15	16	20
add	0	0	0	0	0	0	0

after block

ar	1	8	9	8	6	5	4
pi	1	9	18	26	31	36	40
add	0	0	0	0	0	0	0

```
4.  const int C = 3;
5.
6.  struct Query {
7.      int l, r, x;
8.  };
```

```
void rebuild(vector<Query> &buffer, vector<int> &pi_sum) {
    vector<int> add(pi_sum.size(), 0);

    for (auto q : buffer) {
        add[q.l] += q.x;
        add[q.r + 1] -= q.x;
    }
    buffer.clear();

    int delta = 0, running_sum = 0;
    for (int i = 0; i < pi_sum.size(); i++) {
        delta += add[i];
        running_sum += delta;
        pi_sum[i + 1] += running_sum;
    }
}
```

```
int sum(int l, int r, const vector<Query> &buffer, const vector<int> &pi_sum) {
    int res = pi_sum[r + 1] - pi_sum[l];
    for (auto q : buffer) {
        res += q.x * max(0, min(r, q.r) - max(l, q.l) + 1);
    }
    return res;
}

void upd(int l, int r, int x, vector<Query> &buffer, vector<int> &pi_sum) {
    buffer.push_back({l, r, x});
    if (buffer.size() == C) {
        rebuild(buffer, pi_sum);
    }
}
```

```
int main() {
    int n;
    cin >> n;
    vector<Query> buffer;
    vector<int> a(n), pi_sum(n + 1);
    pi_sum[0] = 0;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
        pi_sum[i + 1] = pi_sum[i] + a[i];
    }

    int q;
    cin >> q;
    for (int i = 0; i < q; i++) {
        int t, l, r;
        cin >> t >> l >> r;
        if (t == 0) {
            cout << sum(l, r, buffer, pi_sum) << "\n";
        }
        else {
            int x;
            cin >> x;
            upd(l, r, x, buffer, pi_sum);
        }
    }
}

```

stdin

```
6
1 2 3 3 2 1
7
0 0 3
1 0 2 2
1 0 3 2
0 0 3
1 4 5 1
1 5 6 1
0 0 3
```

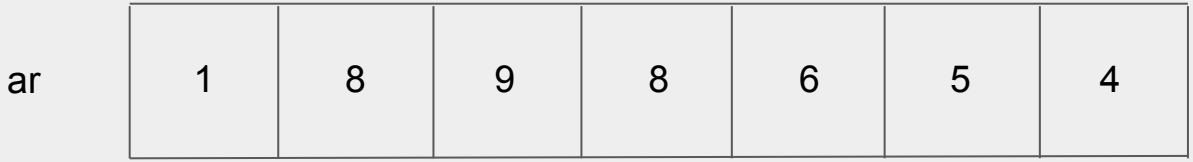
stdout

```
9
23
23
```

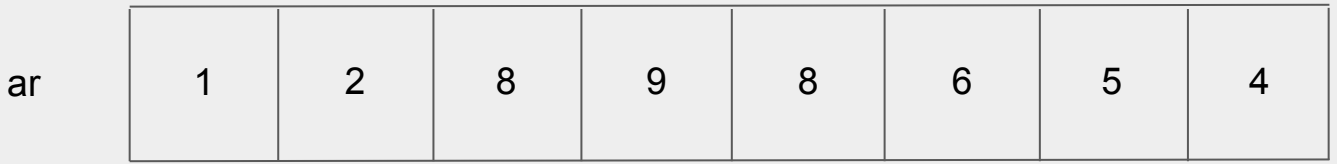
Rebuild

There are queries of two types:

1) Find the sum on a segment.
2) Insert the number x at position i.

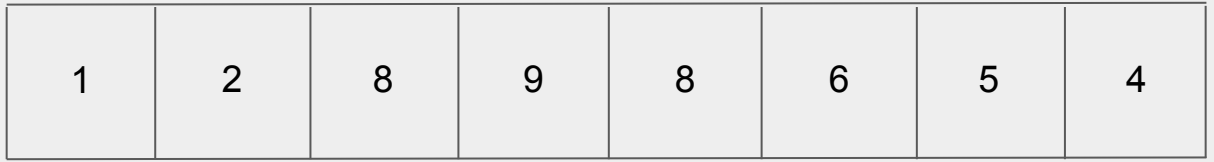

ar
1
8
9
8
6
5
4

insert (1, 2)



sum(1, 3) = 19

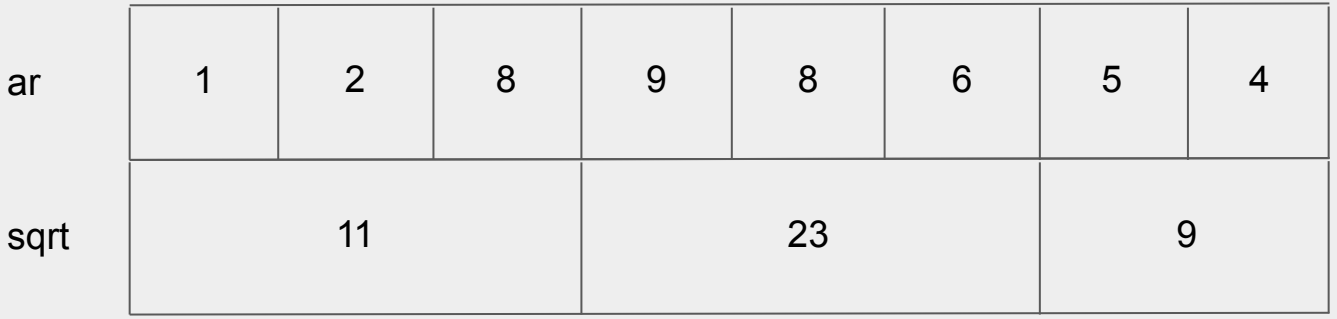
ar



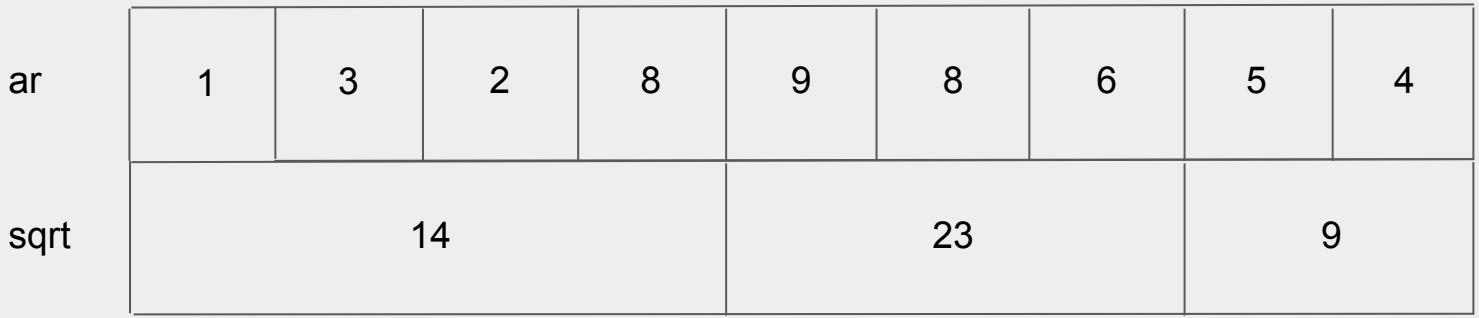
Sqrt inside sqrt

Let's divide the array into blocks, as in the usual sqrt decomposition, and calculate the sums. In the case of inserting an element, we'll simply find the necessary block and the right place within that block, insert the element there, and then recalculate the sums.

c = 3



insert 1, 3



Problem

It all seems fine, but there is a problem.

What if all the insertions are into one block?

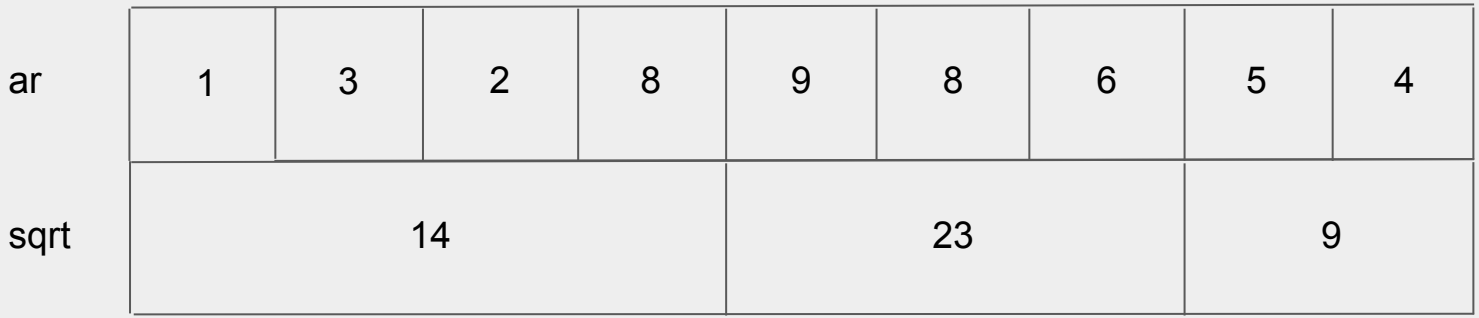
It can grow to an enormous length.


WE NEED TO GO DEEPER
meme-arsenal.ru

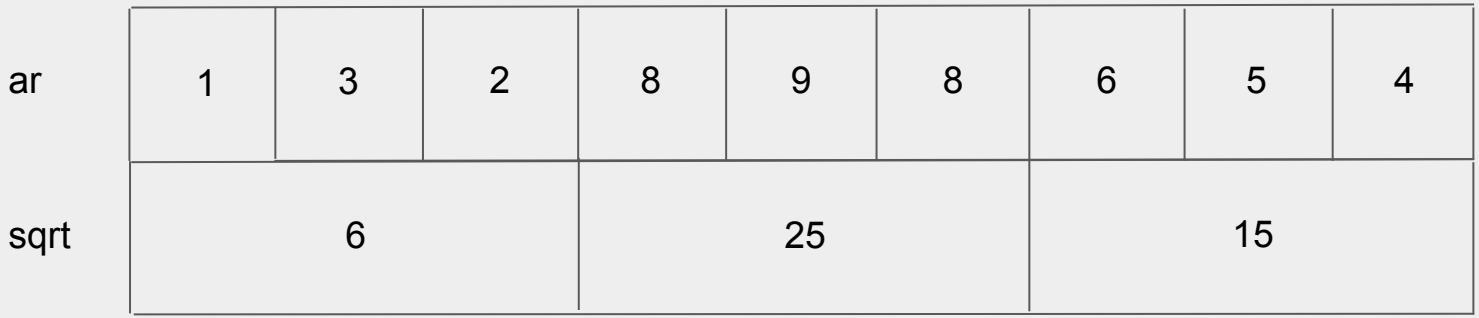
Rebuild decomposition

Let's recalculate our sqrt decomposition every $\sqrt{q}$ queries; by doing this, we will obtain a fully working solution for such a task.

insert 1, 3



rebuild

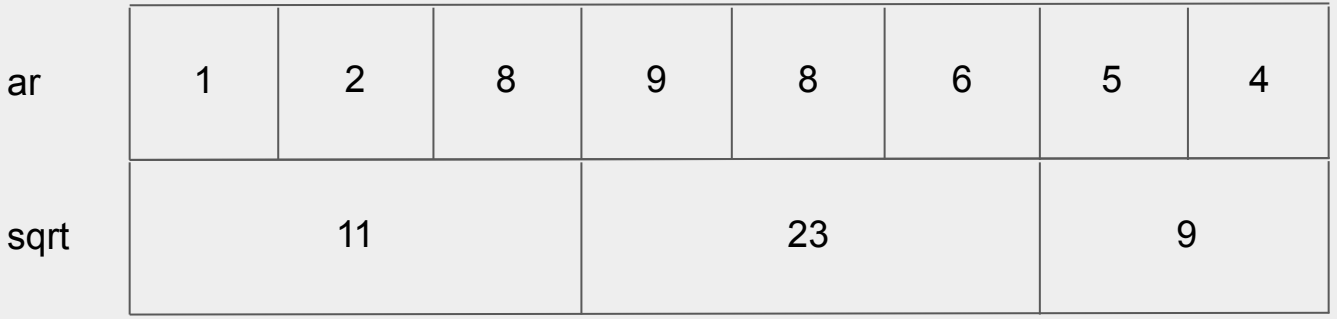


Split-rebuild

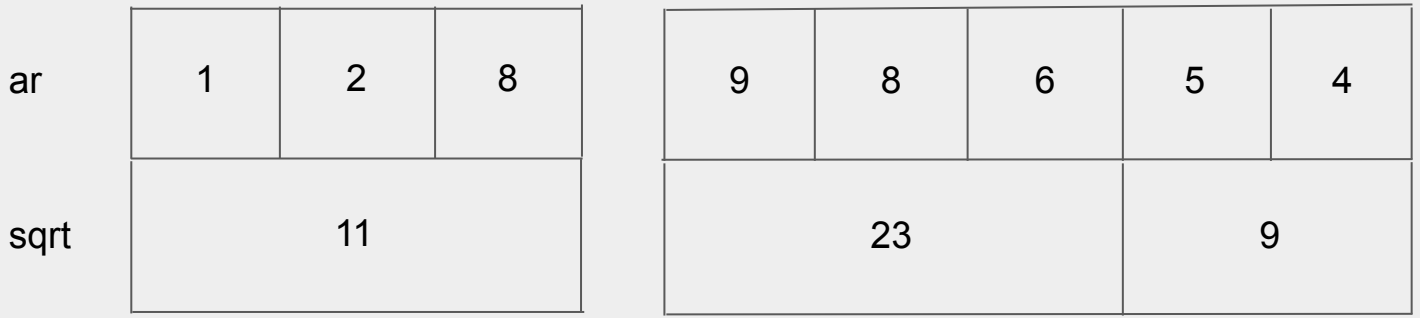
Now let's do it differently. Our main type of operation now is split.

Split will divide the array at position i and find us the index of the block that starts with the i-th element.

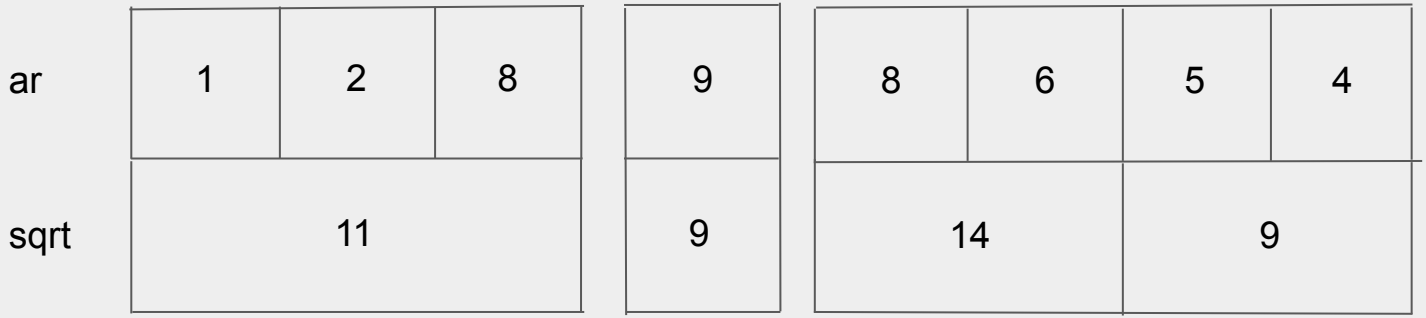
c = 3



split(3)



split(4)

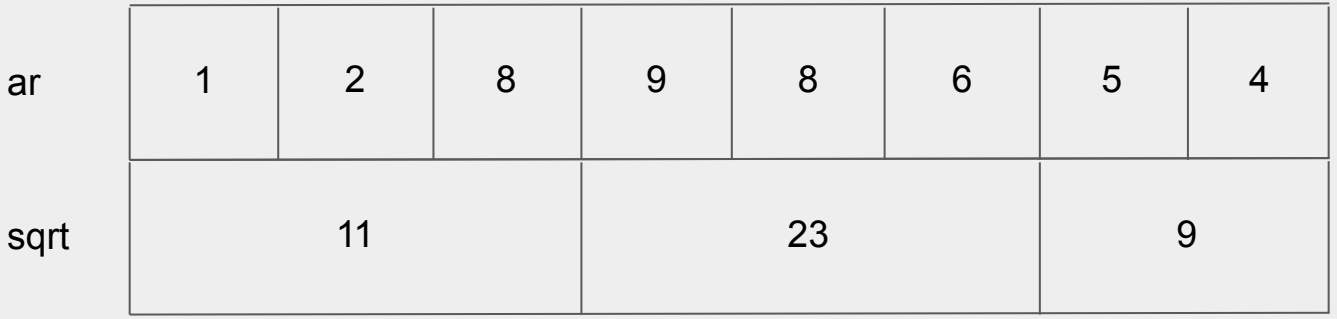


Split-rebuild

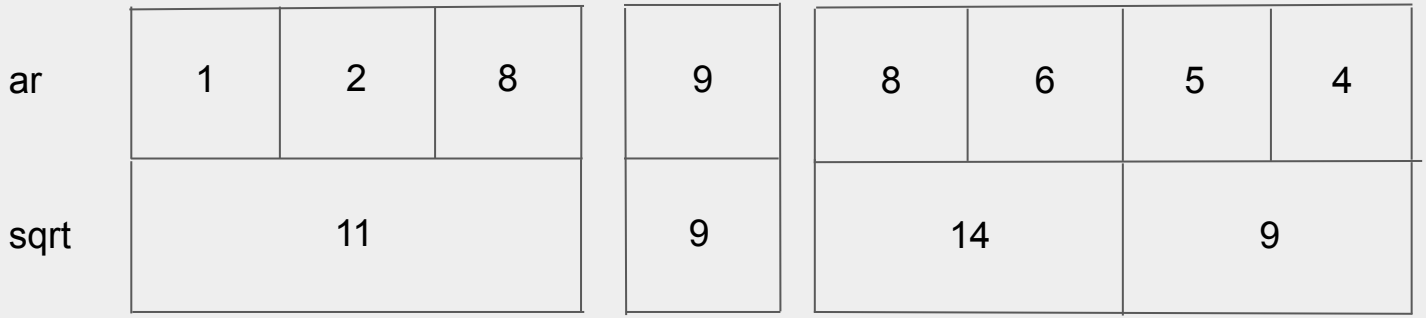
Now, insert(i, x) can be implemented quite simply.

We do split(i), and then between the resulting blocks, we insert a block consisting of the one element we need - {x}.

c = 3



split(4)



insert(4, 9)

And what?

And what else can I do with this?

Pretty much anything, in principle :)

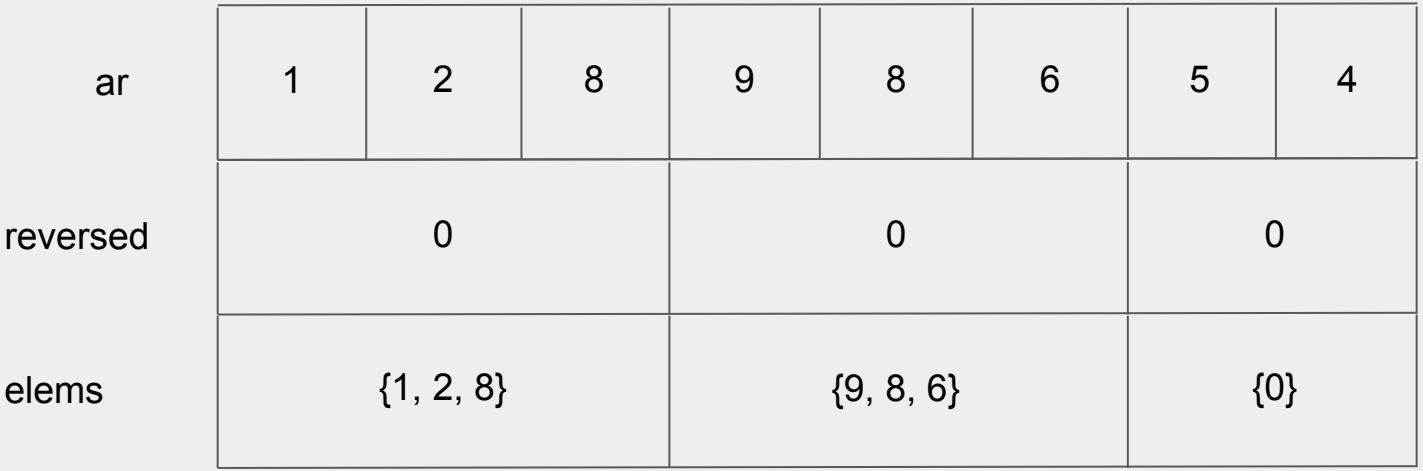
Well, let's take reversing segments as an example)

Reversing segment

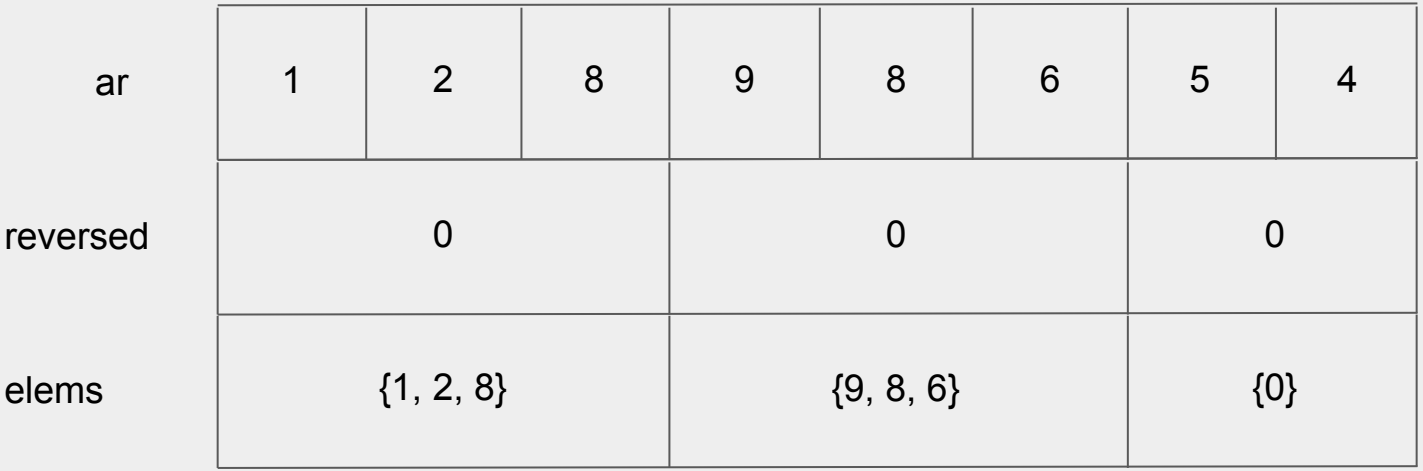
If we want to reverse a segment from l to r, let's just do the following:

1) split(l)
2) split(r + 1)
3) reverse blocks
4) make flag reversed ^= 1

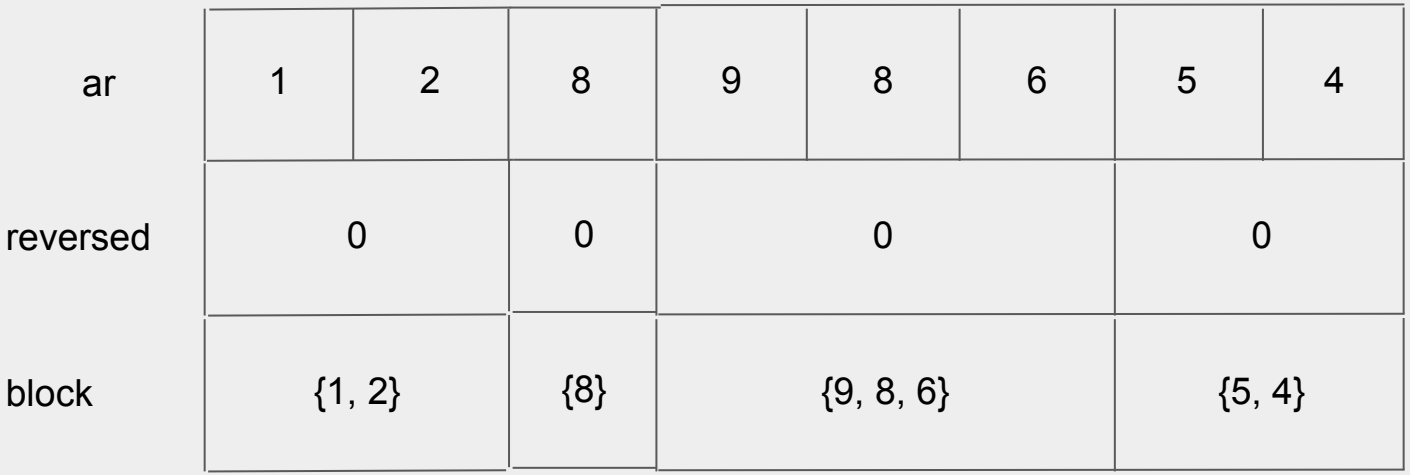
c = 3



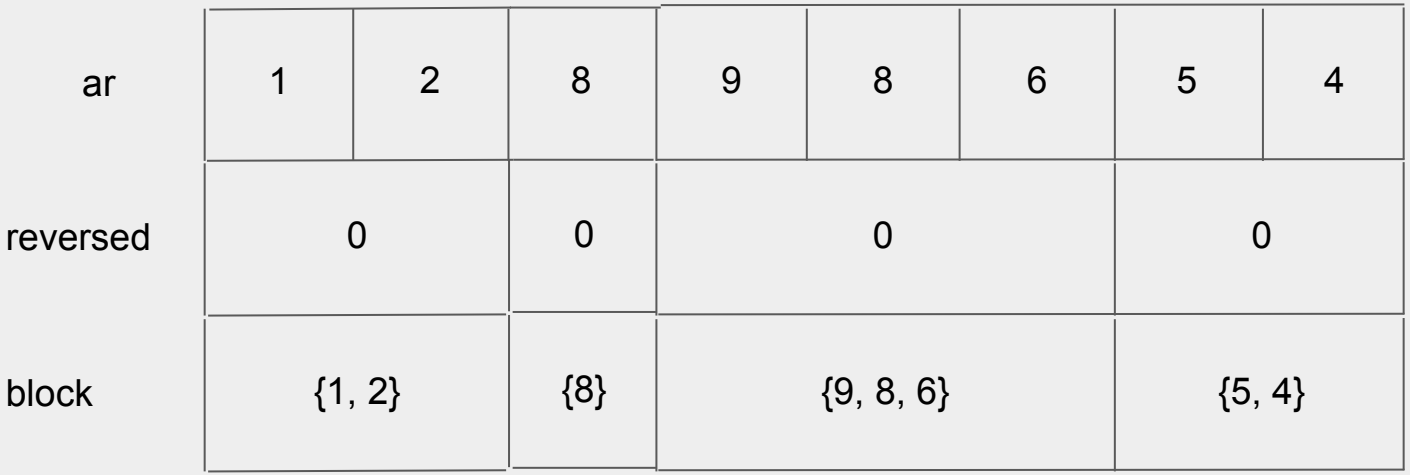
reverse(2, 7)



split(2)



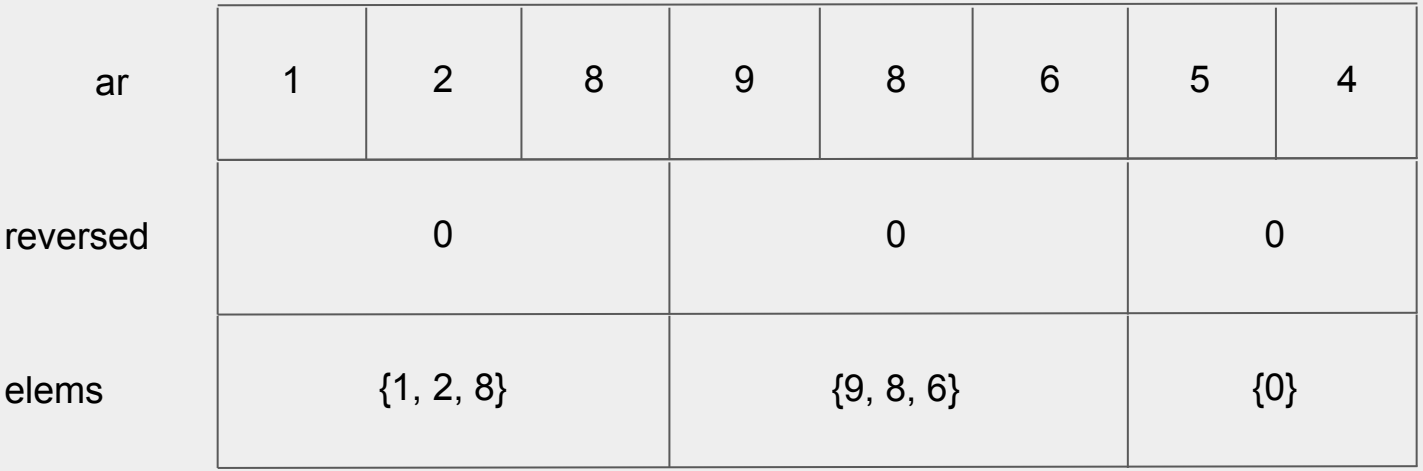
split(7)



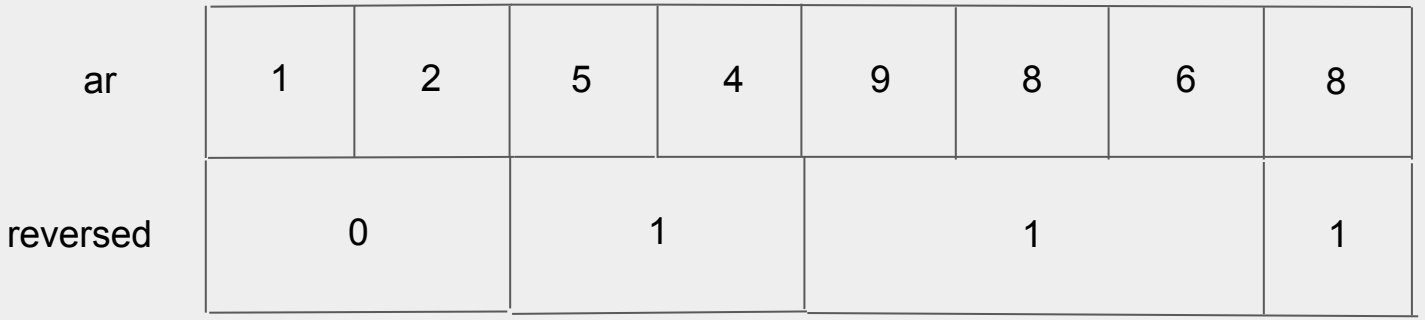
reverse blocks

reversed ^= 1

reverse(2, 7) = 1, 2, 4, 5, 6, 8, 9, 8



after traversing this sqrt decomposition - we will have [1, 2, 4, 5, 6, 8, 9, 8]



Mo

Let's start again with simple tasks - suppose we want to solve the RSQ (Range Sum Query) problem.

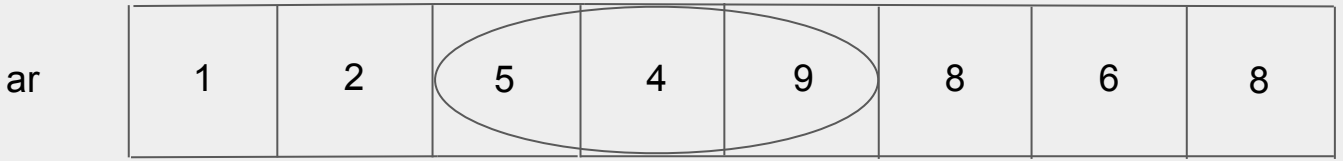
Assume that all queries are given to us in advance, and there are no update queries.

Mo

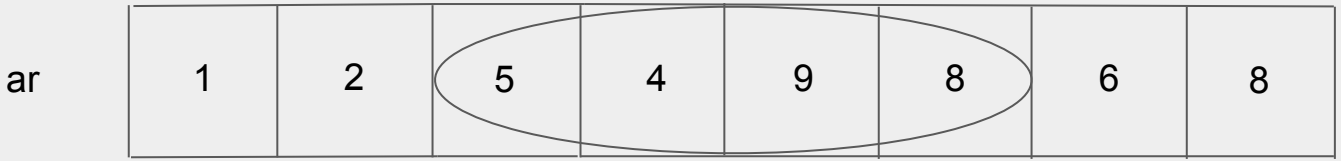
Let's solve them in the simplest way possible.

For each query, we will move the right and left boundaries element by element from the previous query to the new one.

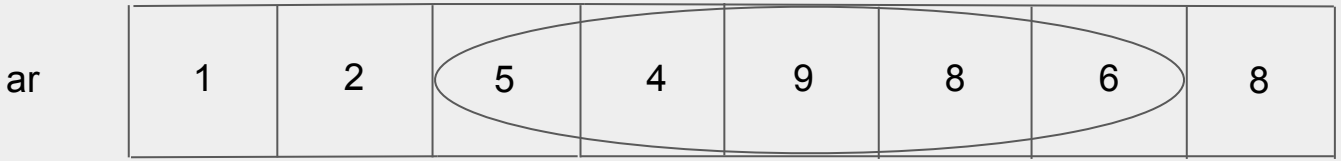
last_query = [2, 4], sum = 18(5 + 4 + 9), query = [3, 6]



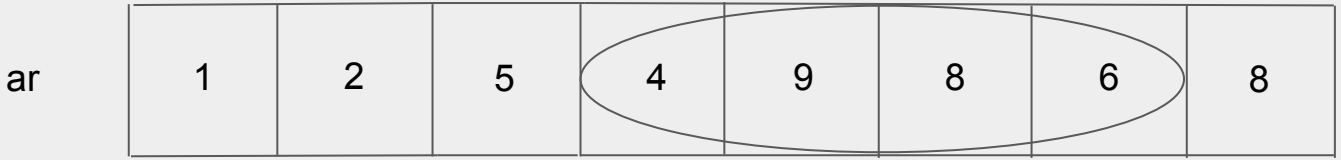
last_query = [2, 5], sum = 18 + 8 = 26, query = [3, 6]



last_query = [2, 6], sum = 26 + 6 = 32, query = [3, 6]



last_query = [3, 6], sum = 32 - 5 = 27, query = [3, 6]



Mo

If we do this with the queries from the problem statement, it will obviously operate in O(nm).

But what if we sort all the queries first and then execute such an algorithm.

Turns out, such magic will indeed work!!!

Comparator

As a comparator, we will use the following algorithm.

Let's divide the array into blocks of size c (which almost always will be $\sqrt{n}$).

1) First, we compare the left boundaries to see which one lies in the earlier block. ($a.l / \sqrt{n} < b.l / \sqrt{n}$)
2) In case of equality, we compare the right boundaries to see which one is smaller. ($a.r < b.r$)

Example

Suppose c = 3

a = [1, 2, 3, 4, 5, 6]

(3, 5)

(1, 4)

(0, 3)

After sorting

Suppose c = 3

a = [1, 2, 3, 4, 5, 6]

(0, 3)

(1, 4)

(3, 5)

```
const int C = 3;

struct Query {
    int l, r, id;
};

int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    int q;
    cin >> q;
    vector<Query> queries;
    int ans[q];
    for (int i = 0; i < q; i++) {
        int l, r;
        cin >> l >> r;
        queries.push_back({l, r, i});
    }
```

```
    auto cmp = [](const Query &a, const Query &b) {
        if (a.l / C != b.l / C) {
            return a.l / C < b.l / C;
        }
        return a.r < b.r;
    };

    sort(queries.begin(), queries.end(), cmp);

    int l = 0, r = -1; // answer to the segment [0, -1]
    int sum = 0;
    for (auto q: queries) {
        while (r < q.r) {
            sum += a[++r];
        }
        while (r > q.r) {
            sum -= a[r--];
        }
        while (l < q.l) {
            sum -= a[l++];
        }
        while (l > q.l) {
            sum += a[--l];
        }
        ans[q.id] = sum;
    }
    for (auto answer : ans) {
        cout << answer << "\n";
    }
```

Success #stdin #stdout 0s 5320KB

stdin

```
7
1 2 3 3 2 1 1
4
1 3
0 4
4 5
3 6
```

stdout

```
8
11
3
7
```

It is forbidden to use magic outside of Hogwarts

It is forbidden to use magic outside of Hogwarts.

So, we still need to prove that it works quickly.

Let's consider what situations can occur:

1) Two adjacent segments lie in different l-blocks.
2) Two adjacent segments lie in the same l-block.

First case

If two adjacent segments lie in different l-blocks, then it doesn't really matter how long they take, because there are no more than $O(n / c)$ or $O(\sqrt{n})$ such queries.

First case(example)

[1, 2, 3, 4, 5, 6]

[], [], [], []

1->2

2->3

3->4

Second case

If two adjacent segments lie in the same l-block, then let's consider the left and right boundaries separately.

Left border

The left boundary between two such queries can move no more than O(c), because (l1 / c) = (l2 / c), and hence the maximum difference is precisely O(c). Therefore, in total, the left boundaries will move O(nc) = O(n√n).

Right border

The right boundary, for one left block, only moves to the right, therefore for one block, it can move no more than O(n) times, and thus, in total, the right boundaries will move $O(n * n / c) = O(n\sqrt{n})$.

Harder task

But still, using such a complex algorithm for such an easy task is overkill.

Let's solve a more difficult problem, we need to find the count of numbers equal to x (x is different for each query) on a segment.

Solution

Now let's maintain an array for counting sort, and instead of calculating the sum, we will simply increment cnt[a] if we encounter the number a.

```
    int l = 0, r = -1; // answer to the segment [0, -1]
    int cnt[MAX_NUM];
    for (int i = 0; i < MAX_NUM; i++) {
        cnt[i] = 0;
    }
    for (auto q: queries) {
        while (r < q.r) {
            cnt[a[++r]]++;
        }
        while (r > q.r) {
            cnt[a[r--]]--;
        }
        while (l < q.l) {
            cnt[a[l++]]--;
        }
        while (l > q.l) {
            cnt[a[--l]]++;
        }
        ans[q.id] = cnt[q.x];
    }
    for (auto answer : ans) {
        cout << answer << "\n";
    }
```

stdin

```
7
1 2 3 3 2 1 1
4
1 3 1
0 4 3
4 5 2
3 6 3
```

stdout

```
0
2
1
1
```

What else?

Sometimes within Mo's algorithm, you can even store some more complex structures for the step of the algorithm (shifting either the right or the left boundary). For example, you can maintain a sqrt decomposition inside Mo's algorithm.

Task

Let's look for the mex (minimal excluding natural number) on a segment.

Example: for the segment [0, 1, 3] the answer is 2.

Example: for the segment [1, 3] the answer is 0.

Solution

Let's maintain a sqrt decomposition for counting the mex inside Mo's algorithm.

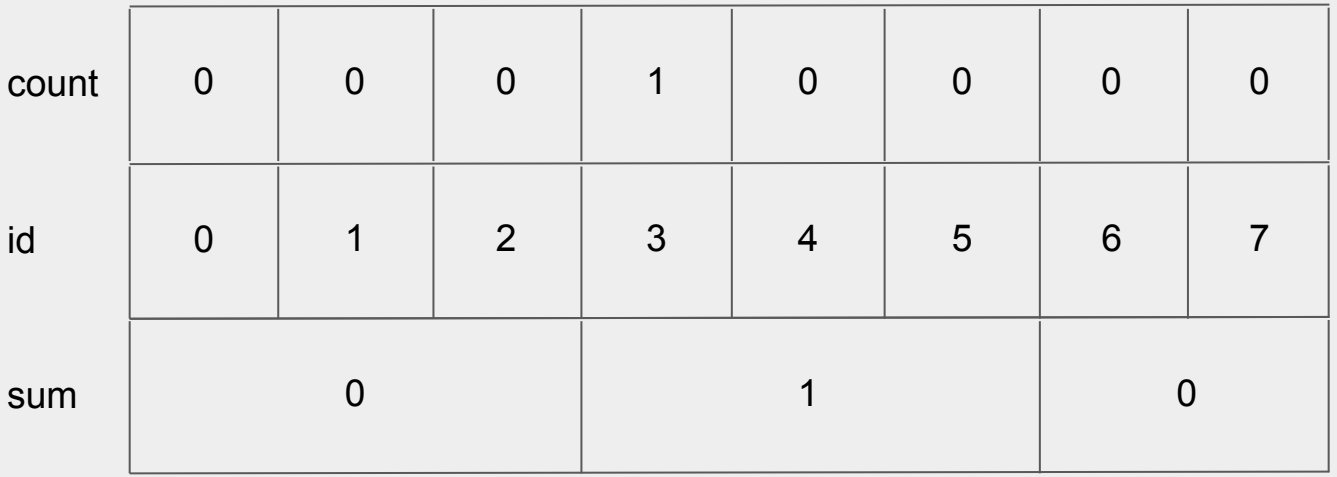
Sqrt decomposition will count the number of unique elements on a segment.

To do this, we will maintain a sqrt decomposition with sum function and a count array.

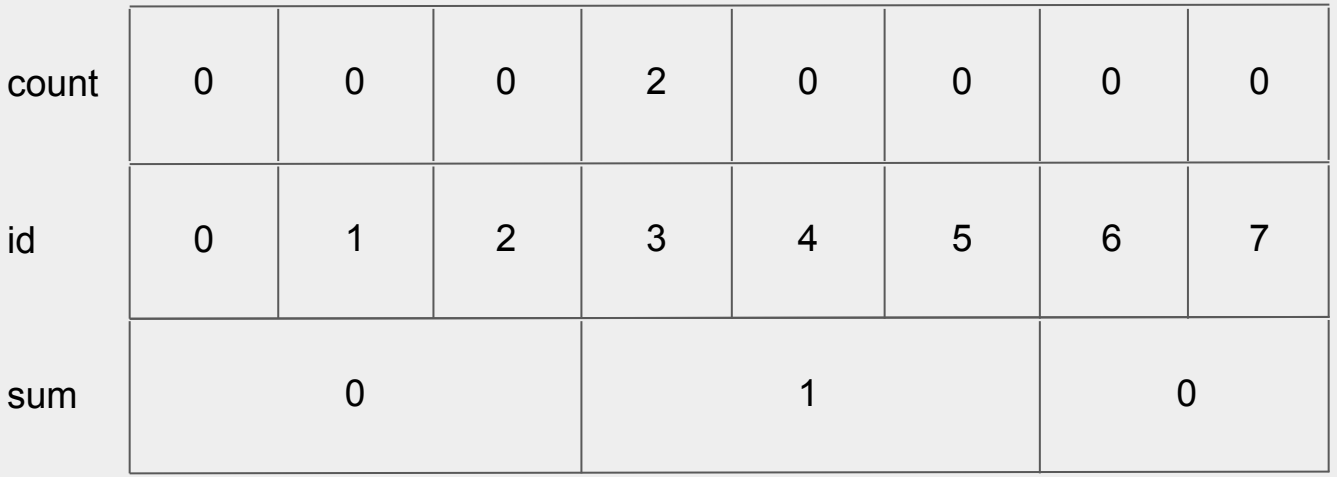
c = 3

count	0	0	0	0	0	0	0	0
id	0	1	2	3	4	5	6	7
sum	0			0			0	

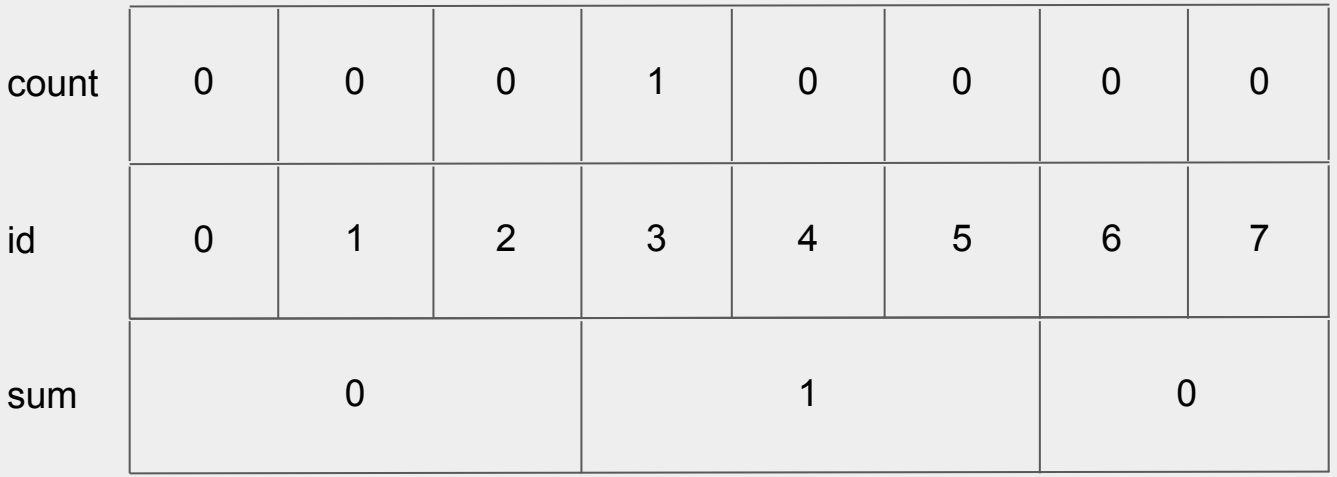
add(3)



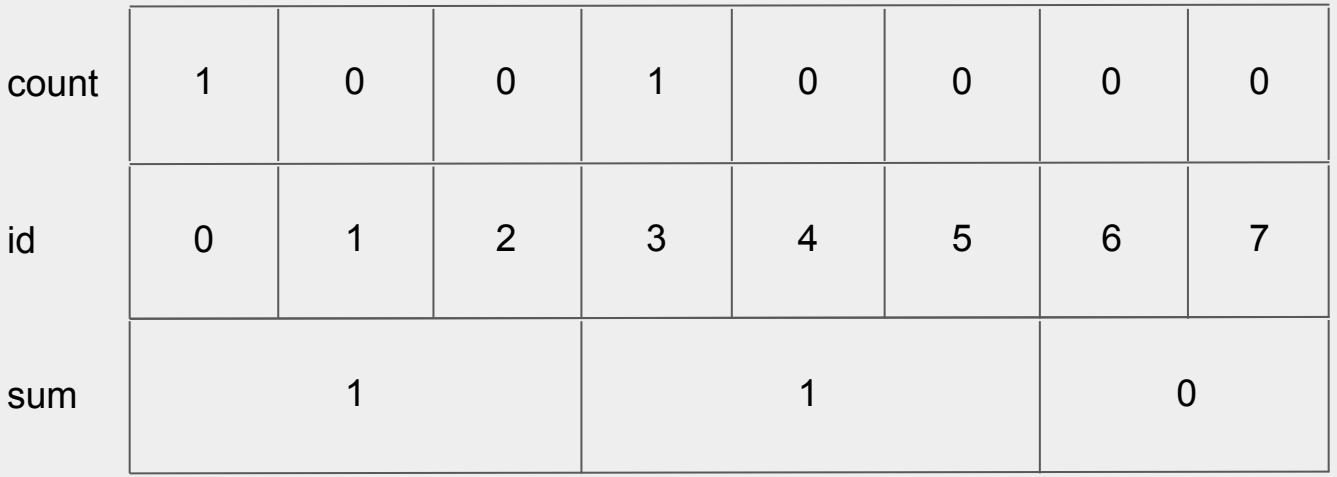
add(3)



remove(3)



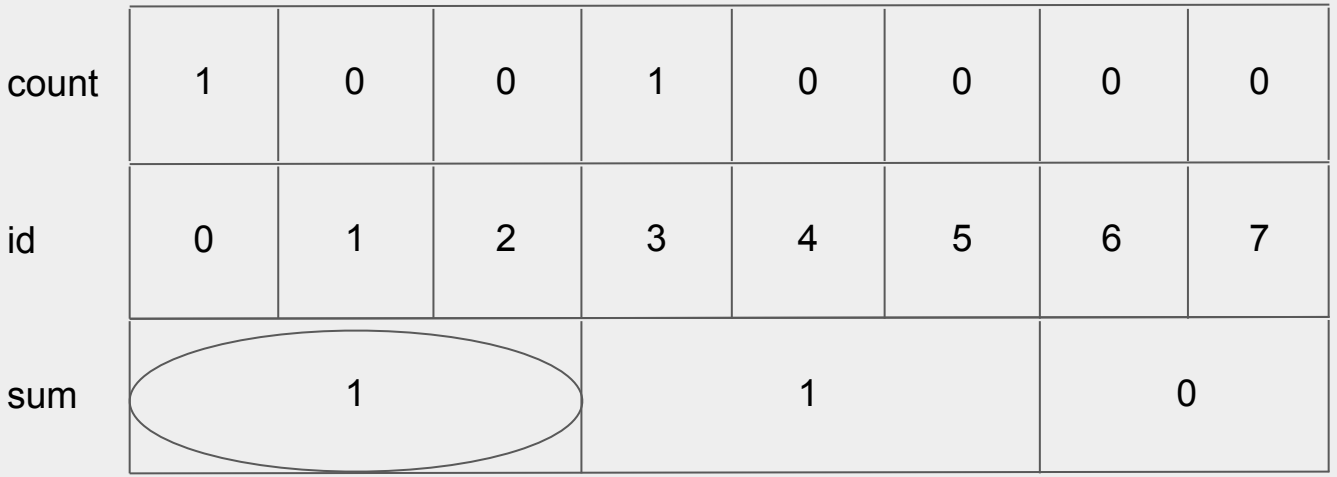
add(0)



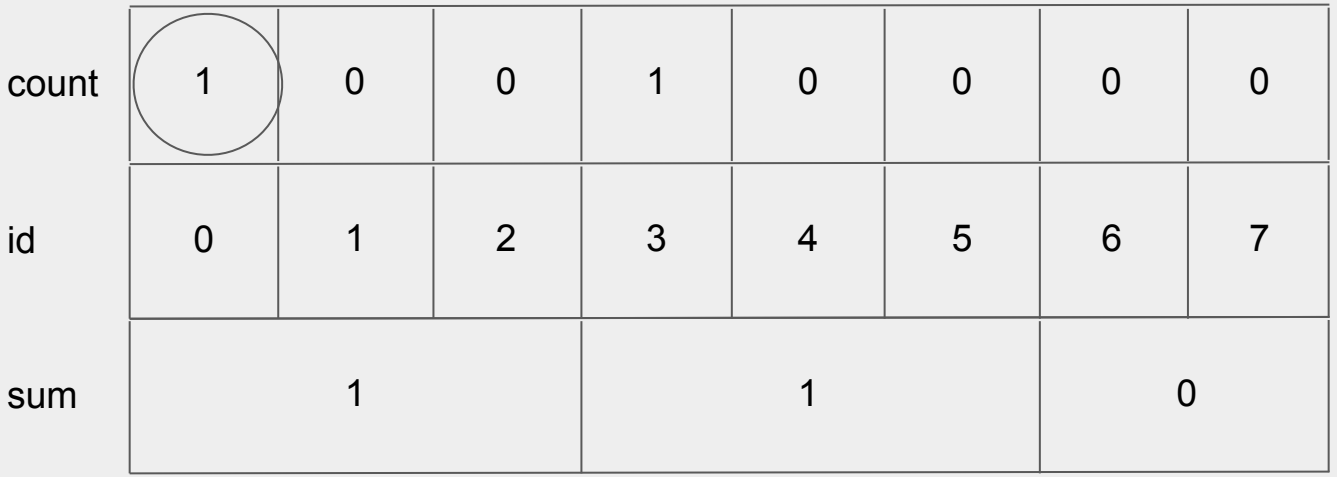
mex?

count	1	0	0	1	0	0	0	0
id	0	1	2	3	4	5	6	7
sum	1			1			0	

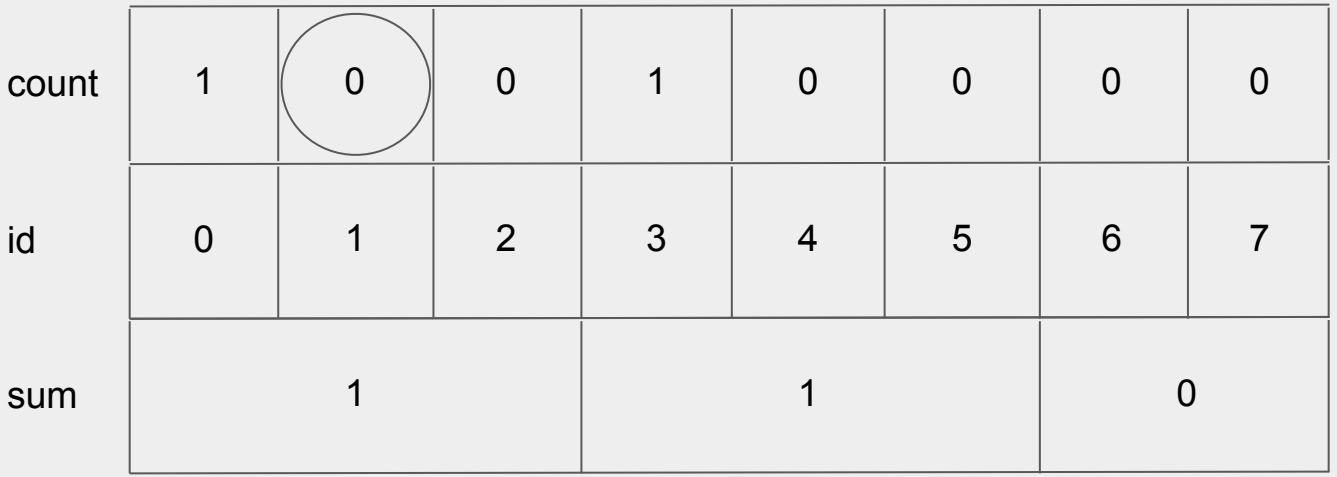
mex? (sum = 1) != c => mex in [0, c)



mex? a[0] > 0 => mex in [1, c)



mex? a[1] = 0 => mex = 1



Solution

In Mo's algorithm itself, we will only change the step, and when we find the answer, we will make a query to the root decomposition and find the answer in $O(\sqrt{n})$, so the algorithm will still work in the same amount of time.

What else?

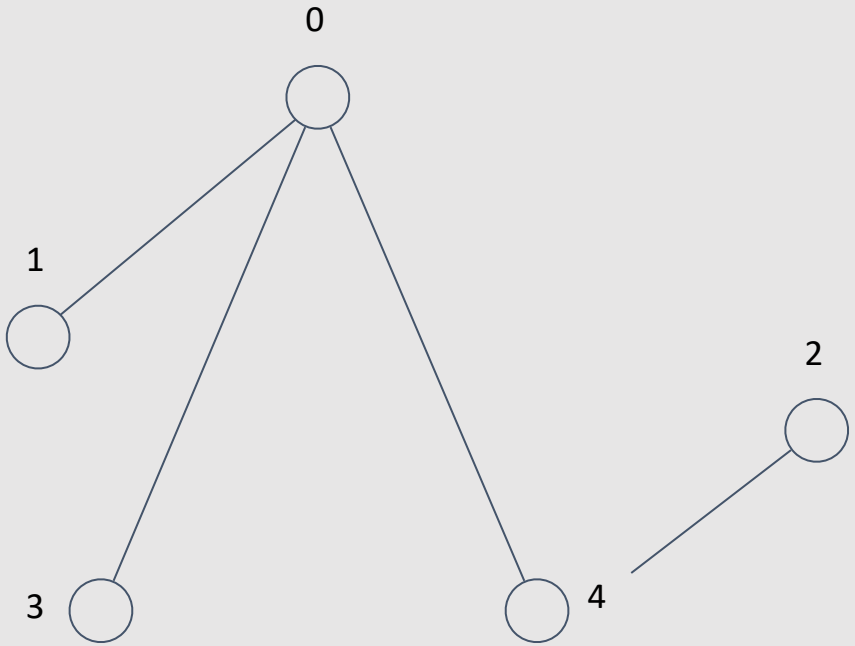
In general, we can solve any offline (and sometimes even online!!) problem in this way, in which we can quickly perform the transition step (adding or removing an element).

Mo on trees

Let's take the first Euler tour, in which we add a vertex when we enter or leave it.

Let's think about what a path between two vertices in such a tree represents.

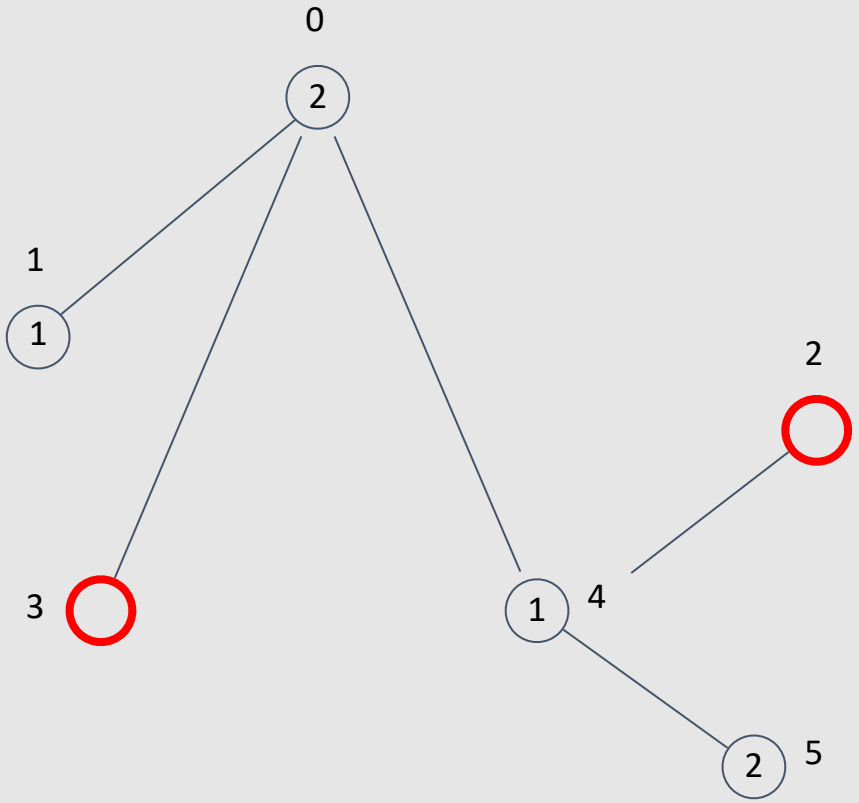
[0, 1, 1, 3, 3, 4, 2, 2, 4, 0]



Mo on trees

If we take a segment in such an Euler tour between tin[a] and tout[b] or tin[b] and tout[a], then it will either encounter vertices on the path, or extra vertices will be encountered twice.

[0, 1, 1, 3, 3, 4, 5, 5, 2, 2, 4, 0]



Mo on trees

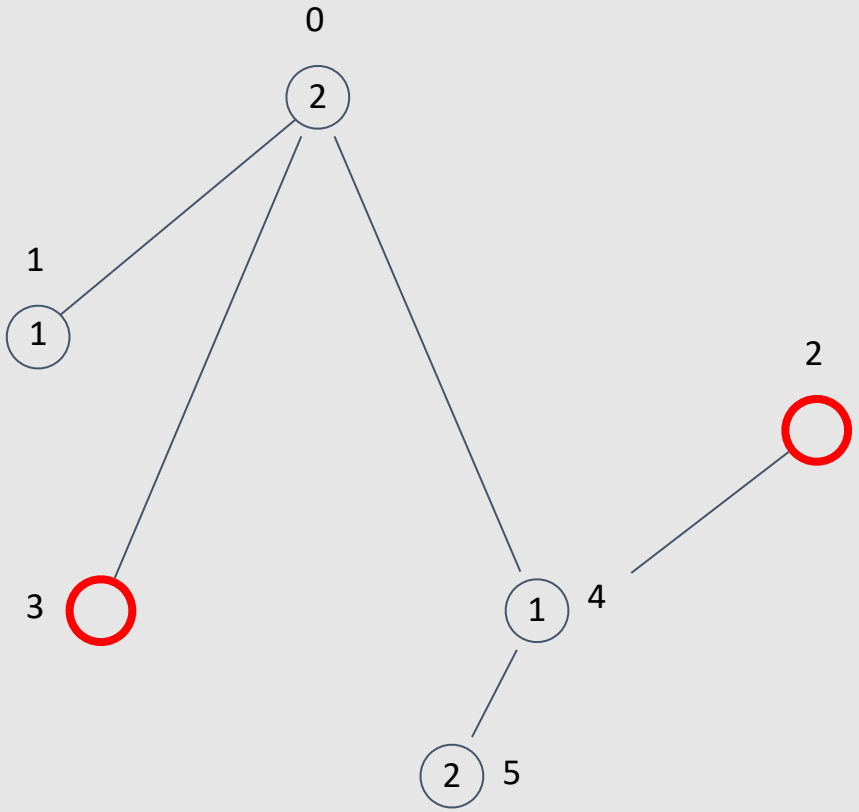
Suppose now there is some path problem.

We can reduce it to Mo's algorithm on an array, where the array will be the Euler tour, and the query on the paths from vertex a to vertex b will become a query on the segment from some combination of (tin[a], tin[b], tout[a], tout[b]).

Mo on trees

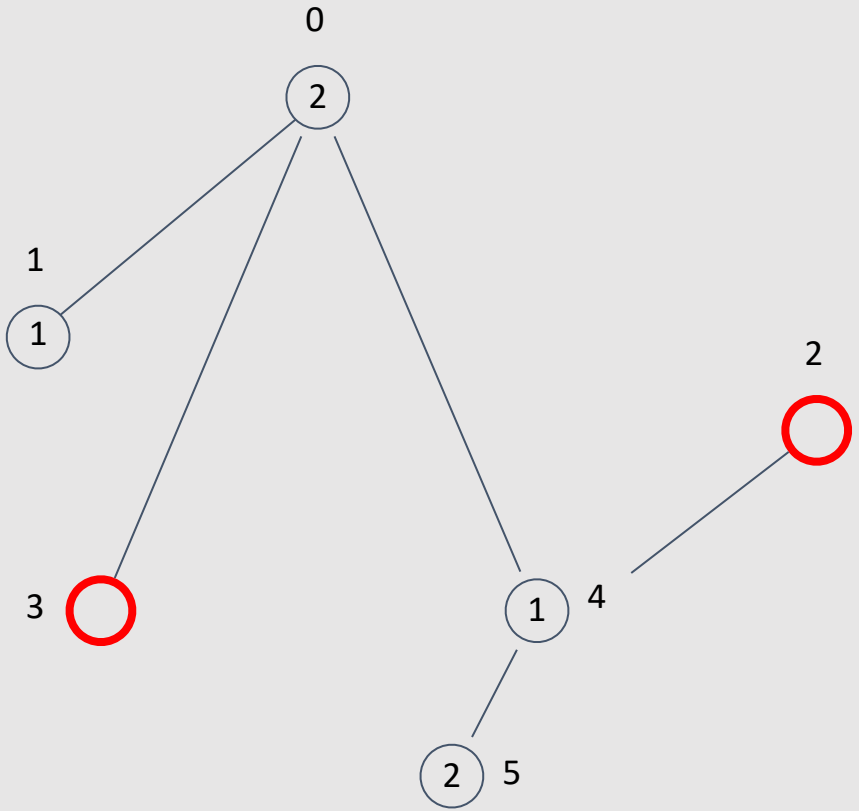
Let's solve the problem we previously paused on - find the count of numbers equal to x on the path.

[0, 1, 1, 3, 3, 4, 5, 5, 2, 2, 4, 0]



3, 4, 5, 5, 2

cnt[v[3]] += 1 cnt[1] += 1
cnt[v[4]] += 1 cnt[1] += 1
cnt[v[5]] += 1 cnt[2] += 1
cnt[v[5]] -= 1 cnt[2] -= 1
cnt[v[2]] += 1 cnt[1] += 1
cnt = [3, 0]



Mo on trees

Here an important clarification is needed - if a vertex is encountered twice on the segment, we must ignore it, so we will also keep an array for counting vertices.

But how do we choose a segment so that the elements a and b themselves are encountered exactly once?

Mo on trees

We can simply consider the boundaries separately in the implementation.

Or we can check the placement of the vertices and analyze the cases.

Cases

If tout[a] < tin[b], then we will take such a segment, because a lies strictly before b.

If tout[b] < tin[a], then we will take such a segment.

If neither condition is met, then we take the segment (min(tin[a], tin[b]), max(tin[a], tin[b])) or the same with tout, because this means that one of the vertices is an ancestor of the other.

```cpp
void dfs(int u, const vector<vector<int> > &g, vector<bool> &visited, int &timer, vector<int> &euler, vector<int> &tin, vector<int> &tout) {
    if (visited[u]) {
        return;
    }
    tin[u] = timer++;
    euler.push_back(u);
    visited[u] = true;
    for (auto v: g[u]) {
        dfs(v, g, visited, timer, euler, tin, tout);
    }
    tout[u] = timer++;
    euler.push_back(u);
}

int main() {
    int n, timer = 0;
    cin >> n;
    vector<vector<int> > graph(n);
    vector<bool> visited(n);
    vector<int> euler, tin(n), tout(n), a(n);

    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

```

```
    for (int i = 0; i < n - 1; i++) {
        int from, to;
        cin >> from >> to;
        graph[from].push_back(to);
    }

    dfs(0, graph, visited, timer, euler, tin, tout);
    int q;
    cin >> q;
    vector<Query> queries;
    int ans[q];
    for (int i = 0; i < q; i++) {
        int u, v, x;
        cin >> u >> v >> x;
        int l = min(tin[u], tin[v]), r = max(tin[v], tin[u]);
        if (tout[u] < tin[v]) {
            l = tout[u];
            r = tin[v];
        }
        if (tout[v] < tin[u]) {
            l = tout[v];
            r = tin[u];
        }
        queries.push_back({l, r, x, i});
    }
```

```
61.     auto cmp = [](const Query &a, const Query &b) {
62.         if (a.l / C != b.l / C) {
63.             return a.l / C < b.l / C;
64.         }
65.         return a.r < b.r;
66.     };
67.
68.     sort(queries.begin(), queries.end(), cmp);
69.     int l = 0, r = -1; // answer to the segment [0, -1]
```

```
    for (auto q: queries) {
        while (r < q.r) {
            auto vertex = euler[++r];
            cnt_vertex[vertex]++;
            if (cnt_vertex[vertex] == 1) {
                cnt_number[a[vertex]]++;
            }
            else {
                cnt_number[a[vertex]]--;
            }
        }
        while (r > q.r) {
            auto vertex = euler[r--];
            cnt_vertex[vertex]--;
            if (cnt_vertex[vertex] == 1) {
                cnt_number[a[vertex]]++;
            }
            else {
                cnt_number[a[vertex]]--;
            }
        }
        while (l < q.l) {
            auto vertex = euler[l++];
            cnt_vertex[vertex]--;
            if (cnt_vertex[vertex] == 1) {
                cnt_number[a[vertex]]++;
            }
            else {
                cnt_number[a[vertex]]--;
            }
        }
        while (l > q.l) {
            auto vertex = euler[--l];
            cnt_vertex[vertex]++;
            if (cnt_vertex[vertex] == 1) {
                cnt_number[a[vertex]]++;
            }
            else {
                cnt_number[a[vertex]]--;
            }
        }
        ans[q.id] = cnt_number[q.x];
```

stdin

```
6
2 1 1 1 1 2
0 1
0 3
0 4
4 5
4 2
2
3 2 1
5 4 1
```

stdout

```
3
1
```

Mo on trees

Actually, the same algorithm can be used for values on edges, not vertices, but think for yourself how to do it.

Feel free to write to me in direct messages)

3-D Mo

What if there are modification queries after all?

Actually, even that can be solved with Mo's algorithm

(although changes are only allowed at a single point).

3-D Mo

As I understand 3D Mo:

What is 1D Mo - it's a scanline, what is 2D Mo - it's a scanline on the right boundary + sqrt decomposition on the left boundary. What is 3D Mo - it's a scanline on the right boundary + sqrt decomposition on the left boundary + sqrt decomposition by time.

3-D Mo

So now we add the query number as a parameter in Mo's algorithm and now we have a time machine, because we start rolling back to the past.

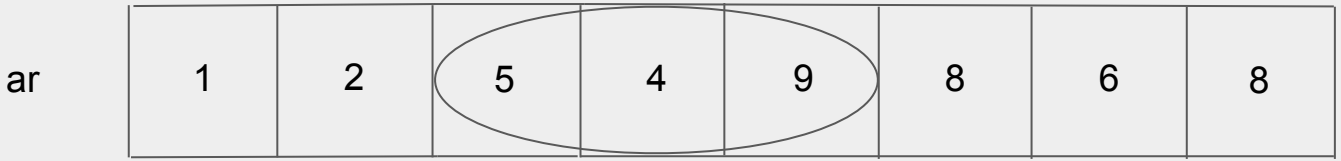
Suppose we just answered the query (t1, l1, r1), that is, a query on the segment [l1, r1], which was the t1-th query, and now we want to answer the query (t2, l2, r2).

3-D Mo

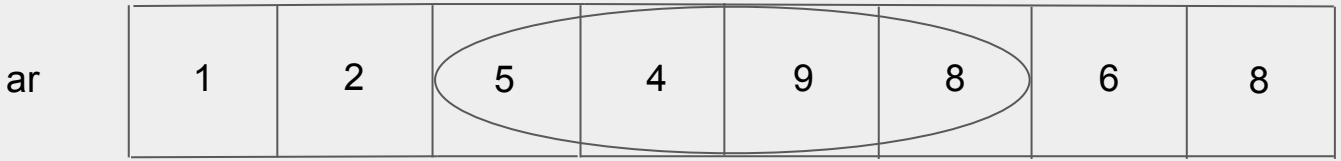
What to do with the left and right boundaries - we already know, but what to do with the time of the query?

Let's take a look at a simple example (such as the same sum). We need to apply all queries from t1 to t2, for this we will keep the current array and change the element in it, and if this element falls within the current segment, then we will also change the answer for the segment.

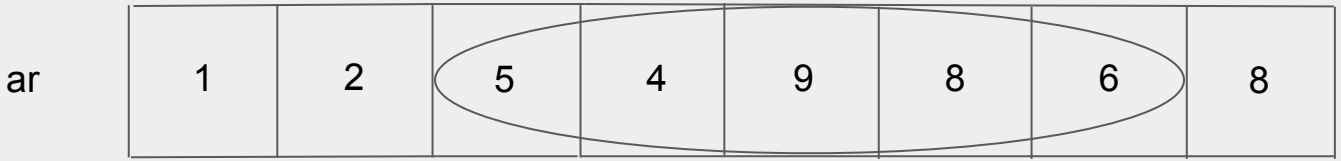
last_query = [2, 4, 2], sum = 18, query = [3, 6, 5]
q[3] = (a[1] += 3)
q[4] = (a[4] += 5)



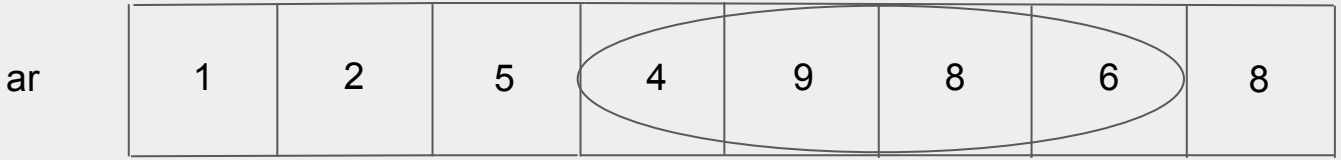
last_query = [2, 5, 2], sum = 18 + 8 = 26, query = [3, 6, 5]
q[3] = (a[1] += 3)
q[4] = (a[4] += 5)



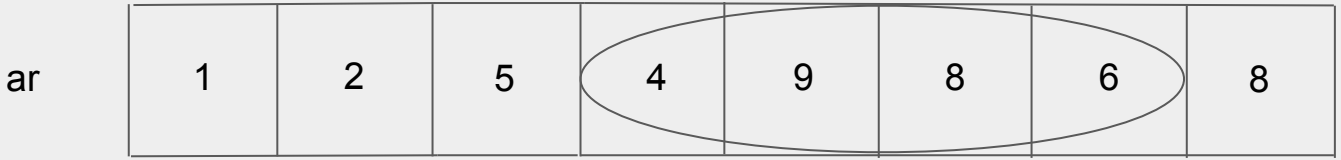
last_query = [2, 6, 2], sum = 26 + 6 = 32, query = [3, 6, 5]
q[3] = (a[1] += 3)
q[4] = (a[4] += 5)



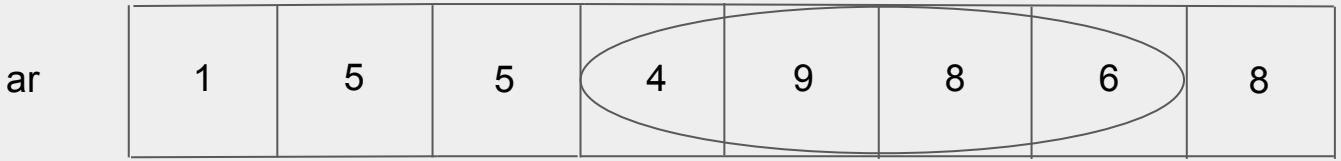
last_query = [3, 6, 2], sum = 32 - 5 = 27, query = [3, 6, 5]
q[3] = (a[1] += 3)
q[4] = (a[4] += 5)



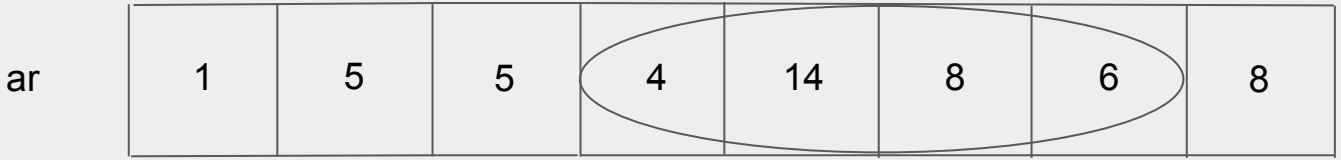
last_query = [3, 6, 2], sum = 27, query = [3, 6, 5]
q[3] = (a[1] += 3)
q[4] = (a[4] += 5)



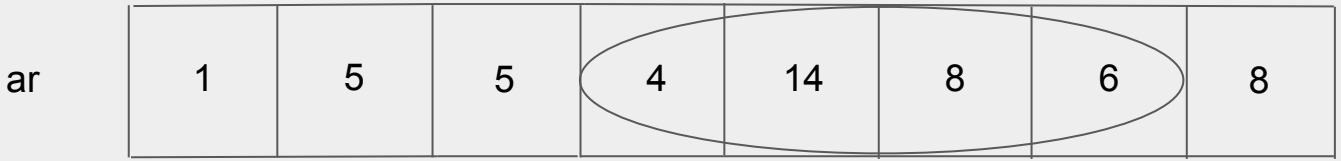
last_query = [3, 6, 3], sum = 27, query = [3, 6, 5]
q[3] = (a[1] += 3)
q[4] = (a[4] += 5)



last_query = [3, 6, 4], sum = 32, query = [3, 6, 5]
q[3] = (a[1] += 3)
q[4] = (a[4] += 5)



last_query = [3, 6, 5], sum = 32, query = [3, 6, 5]
answer = 32



3-D Mo

In the opposite direction, we will be subtracting, not adding.

3-D Mo

Can it be more complex? Of course, it can - let's go over the already solved problem of finding the count of a number on a segment. Now we also know how to change the number at a certain position.

Actually, it will be implemented just as in the sum - we will change the element, remove the old value and add the new one.

```cpp
#include <bits/stdc++.h>

using namespace std;

const int C = 2048, SIZE = 2e5 + 7;

struct Q {
    int l, r, t, x;

    bool operator< (Q& other) {
        if (l / C == other.l / C) {
            if (t / C == other.t / C) {
                return r < other.r;
            }
            return t < other.t;
        }
        return l < other.l;
    }
};

Q blocks[SIZE];
```

```
23.  int amount_of_number[SIZE]; // count array
24.  int a[SIZE], updated_position[SIZE], elem_after_update[SIZE], elem_before_update[SIZE];
25.  int array_after_operations[SIZE], ans[SIZE];
26.
27.  void add_element(int x) {
28.      amount_of_number[x]++;
29.  }
30.
31.  void delete_element(int x) {
32.      amount_of_number[x]--;
33.  }
```

```
int n, q;
cin >> n >> q;
vector<int> numbers;
for (int i = 0; i < n; i++) {
    cin >> a[i];
    array_after_operations[i] = a[i];
    numbers.push_back(a[i]);
}
int sz = 0;
int number_of_update = -1;
for (int i = 0; i < q; i++) {
    int t;
    cin >> t;
    if (t == 2) {
        int pos, elem;
        cin >> pos >> elem;
        pos--;
        number_of_update++;
        updated_position[number_of_update] = pos;
        elem_before_update[number_of_update] = array_after_operations[elem];
        array_after_operations[elem] = elem_after_update[number_of_update] = elem;
    }
    else {
        int l, r, x;
        cin >> l >> r >> x;
        l--; // to make from [l - 1, r - 1] -> [l - 1, r)
        blocks[sz++] = {l, r, number_of_update, x};
    }
    ans[i] = -1;
}

sort(blocks, blocks + sz);
```

```
67.     int L = 0, R = 0, T = -1; // answer for segment [0, 0) at the moment -1 is [0...0]
68.     for (int j = 0; j < sz; j++) {
69.         while (R < blocks[j].r) {
70.             add_element(a[R]);
71.             R++;
72.         }
73.
74.         while (L > blocks[j].l) {
75.             L--;
76.             add_element(a[L]);
77.         }
78.
79.         while (R > blocks[j].r) {
80.             R--;
81.             delete_element(a[R]);
82.         }
83.
84.         while (L < blocks[j].l) {
85.             delete_element(a[L]);
86.             L++;
87.         }
```

```
        while (T < blocks[j].t) {
            T++;
            int pos = updated_position[T];
            if (pos >= L && pos < R) {
                delete_element(a[pos]);
                add_element(elem_after_update[T]);
            }
            a[pos] = elem_after_update[T];
        }

        while (T > blocks[j].t) {
            int pos = updated_position[T];
            if (pos >= L && pos < R) {
                delete_element(a[pos]);
                add_element(elem_before_update[T]);
            }
            a[pos] = elem_before_update[T];
            T--;
        }

        ans[blocks[j].t] = amount_of_number[blocks[j].x];
    }

    for (int i = 0; i < q; i++) {
        if (ans[i] != -1) {
            cout << ans[i] << "\n";
        }
    }
}
```

Success #stdin #stdout 0.01s 9728KB

stdin

```
5 5
1 2 1 2 1
1 1 3 3
2 3 3
1 3 5 3
2 3 4
1 1 5 3
```

stdout

```
0
1
0
```

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/prZUXx
seg-tree-count(c++) - https://ideone.com/HSEKmS
triangles(c++) - https://ideone.com/d6WrrW
sqrt on queries(c++) - https://ideone.com/wlRjgv
mo(c++) - https://ideone.com/OuHQVl
cnt(c++) - https://ideone.com/8B53Zu
mo-tree(c++) - https://ideone.com/8VyjAs
3d-mo(c++) - https://ideone.com/r7uN0j