1 SQRT decomposition

1.1 Problem

Let's try solving the folllowing problem. [Problem 1] Let's have an array a of size N. The task is to answer Q queries which can be of 2 types:

1. Set element a_x to y.
2. Find min on the subsegment from l to r.

Let's pretend we don't know segment tree :-)

1.2 Idea

Let's split array into blocks of size K and **pre-calculate** minimums for each block. How do we process queries now?

1. Set element a_x to y. We can simply update the element and **recalculate** the answer for the block. Complexity: $O(K)$.
2. Find min on the subsegment from l to r. We can notice that some block are entirely in the segment $[l; r]$ so we can use **precalculated** answers for them. The number of such blocks is $O(N/K)$. Also, there would be at most $K - 1$ elements that are left in the beginning and at most $K - 1$ in the end. We can simply iterate through all of them. Complexity: $O(N/K + K)$.

Obviously, this solution works with any value of K (if properly implemented), but...
Question: What's the optimal K?
Answer: The one when $N/K = K$ which is $\sqrt{N}$.
Important note: On practice we set K as a constant and change it if needed. This simplifies the debugging process as you can use different values of K and your solution should work correct for all of them,

1.3 Pseudocode

```
// We use 0-indexation
//n, q - from the statement
//K - size of the block, a constant
//INF - something big, a constant
//b - array for SQRT decomposition
///all loops include the last number
//the ID of the block for element i is i/K
for i = 0..n/K:
    b[i] = INF
for i = 0..n-1:
    b[i / K] = min(b[i / K], a[i])

for i = 0..q-1:
    //we have queries of format (t, x, y) and (t, l, r)
    //where t is the type of the query
    //be careful with 0-indexation!
    if t[i] == 1:
        a[x] = y
        blockId = x / K
        b[blockId] = INF
        for i = blockId*k..min(blockId + k - 1, n - 1): //first and last elements of the
            b[blockId] = min(b[blockId], a[i])
    else:
        ans = INF
        i = l
        while (i <= r):
            if i % k == 0 and i + k - 1 <= r:
                //the beginning and the last element in the block are inside
                //you can add separate if for the last block but it's OK
                //if you process the elements from there one by one
                ans = min(ans, b[i / K])
                i += K
            else:
                ans = min(ans, a[i])
                i += 1
```

2 Delayed updates

We already used a block idea on the array, now let's try to use the similar one on queries. To illustrate it, let's take a look at the...

2.1 Problem

Let's have a set which is initially empty and answer Q queries of the following types:

- Add element x to the set. The set in this problem is actually a multiset so there can be repeating elements.
- Calculate the number of elements less than x in the array.

Let's pretend we never heard about treap or ordered_set in C++ :-)

2.2 Idea

Let's simplify the problem by dropping the queries of the first type. Imagine, we already have a set of numbers and a lot of type 2 queries. What could be the solution?

One of the options is to sort this set and answer the queries using binary search (or built-in C++ lower/upper-bound). Let's keep that in mind and switch to queries of the first type.

If we want to maintain the same sorted set of number and also correctly process update queries, we should update this set on each query which is $O(NlogN)$ or $O(N)$ perquery depending on implementation. *Slooooooooowwwww*

And here comes the...

Trick: Let's delay some updates, i.e. update the sorted set only after each K updates.

Now to answer the query of the second type we have 2 sources of data:

- Sorted set, where we can use binary search. Complexity: $O(logQ)$
- At most $K-1$ updates that are not yet added to the set. We can simply iterate through them. Complexity: $O(K)$.

The total complexity is $O(K + logQ)$.

Also, we shouldn't forget about the rebuilding complexity. It is $O(QlogQ)$ and we'll do it $O(\frac{Q}{K})$ times. Again, it works with any K but let's find an optimal one, i.e. where $Q(K+logQ) = \frac{Q^2logQ}{K}$. As we are not interested in the exact value but rather it's magnitude we can say that $KQ = \frac{Q^2logQ}{K}$, so $K = \sqrt{QlogQ}$.

2.3 Pseudocode

```
//a - the sorted set
//delayed - elements that we didn't add yet
function merge():
    for x in delayed:
        a.add(x)
    sort(a)
    clear(delayed)

function update(x):
    delayed.add(x)
    if delayed.size * delayed.size > q:
        merge()

function get(x):
    //1st part, elements from sorted set
    ans = //use binary search, lower_bound from C++ or anything like this

    //2nd part, delayed
    for y in delayed:
        if y < x:
            ans++
    return ans

///Now you can use update and get function to answer queries.
```

3 Mo's algorithm

In the last part we'll talk not really about splitting something but about sorting (and square roots!) As usual, let's start with a ...

3.1 Problem

Let's have an array a of N elements and Q queries. Each query contains 2 numbers l and r. We want to count, how many numbers x exist, such that number x occurs exactly x times among numbers $a_l, a_{l+1}, \dots a_r$.

3.2 Idea

We'll solve this problem **offline** which means that we firstly read all the queries and only then answer them in **some** order.

For the technique we discuss in this section I don't have any intuitive explanation how to come up with it. Usually you just memorize it and try to apply.

Let's **sort** the queries but in some weird order. Firstly, we'll sort them in increasing order by $\frac{l}{K}$ where K is some constant (*you can guess which value it is equal to...*). If two queries $[l_i; r_i]$ and $[l_j; r_j]$ have $\frac{l_i}{K} = \frac{l_j}{K}$ then we sort them by r in increasing order.

After sorting, we will answer the query in the following way. We will keep some active segment $[l_{cur}; r_{cur}]$ and all the data we need to calculate the answer for it along with the answer itself. Now, when the query $[l_i; r_i]$ arrives we need to remove some elements and add some other elements by shifting borders of the segment in order to shift $[l_{cur}; r_{cur}]$ to $[l_i; r_i]$. After doing it, the current segment becomes equal to the last query and we can move on to the next one.

How many additions and deletions of the elements we will do? Let's calculate it for a block of queries with equal $\frac{l}{K}$. As the r values are sorted in increasing order, it will have $O(N)$ shifts. On the other side, for each query the l value won't shift more than $O(K)$ times because they all have equal $\frac{l}{K}$. In total, the number of shift for block is $O(K * q_i + N)$ where q_i is the number of elements in this block.

How many blocks do we have? Well, we have $\frac{N}{K}$ of them because l can be up to N. So, the total complexity is $O(\frac{N^2}{K} + K * (q_0 + q_1 + ...))$. All q_i sum up to Q, so the answer is $O(\frac{N^2}{K} + K * Q)$.

Optimal value of K? As you may guess, it is $\sqrt{N}$.

3.3 How to solve the actual problem

We can maintain a map with counts as a data structure. While adding and deleting you can check if you're deleting element with $cnt_x = x$ or not and increment/decrement/don't change the answer.

3.4 Pseudocode

```
///a - array from the input
///cnt - counting array or map
///each query should have parameter l and r AND ALSO:
//id -- the initial number of the query so that you print answers in correct order

function add(x):
    if cnt[x] == x:
        ANS -= 1
    cnt[x]++;
    if cnt[x] == x:
        ANS += 1

function del(x):
    if cnt[x] == x:
        ANS -= 1
    cnt[x]++;
    if cnt[x] == x:
        ANS += 1
...

sort queries as described
cur_l = 1 //current l
cur_r = 0 //current r
for (l, r, id) in queries:
    while cur_r < r:
        add(a[cur_r + 1])
        r += 1
    while cur_l > l:
        del(a[cur_l - 1])
        l -= 1
    while cur_r > r:
        del(a[cur_r - 1])
        r -= 1
     while cur_l < l:
        add(a[cur_l + 1])
        l += 1
    ans[id] = ANS
```