MO’s ALGORITHM (QUERY SQRT DECOMPOSITION)

Mo’s algorithm is a generic idea. It applies to the following class of problems:

```
You are given array Arr of length N and Q queries. Each q
uery is represented by two numbers L and R, and it asks y
ou to compute some function Func with subarray Arr[L..R]
as its argument.
```

Let’s Start with a Simple Problem

Given an array of size **N**. All elements of array <= **N**. You need to answer **M** queries. Each query is of the form **L, R**. You need to answer the count of values in range **[L, R]** which are repeated at least **3** times.
Example: Let the array be **{1, 2, 3, 1, 1, 2, 1, 2, 3, 1}** (zero indexed)
Query: L = 0, R = 4. Answer = **1**. Values in the range **[L, R] = {1, 2, 3, 1, 1}** only 1 is repeated at least 3 times.
Query: L = 1, R = 8. Answer = **2**. Values in the range **[L, R] = {2, 3, 1, 1, 2, 1, 2, 3}** 1 is repeated 3 times and 2 is repeated 3 times. Number of elements repeated at least 3 times = Answer = **2**.

Now explain a simple solution which takes O(N^2)

.

```
for each query:
  answer = 0
  count[] = 0
  for i in {l..r}:
    count[array[i]]++
    if count[array[i]] == 3:
      answer++
```

Slight Modification to above algorithm. It still runs in O(N^2).

.

```
add(position):
  count[array[position]]++
  if count[array[position]] == 3:
    answer++

remove(position):
  count[array[position]]--
  if count[array[position]] == 2:
    answer--
```

```
currentL = 0
currentR = 0
answer = 0
count[] = 0
for each query:
  // currentL should go to L, currentR should go to R
  while currentL <= L:
    remove(currentL)
    currentL++
  while currentL >= L:
    add(currentL)
    currentL--
  while currentR <= R:
    add(currentR)
    currentR++
  while currentR >= R:
    remove(currentR)
    currentR--
  output answer
```

Initially we always looped from **L to R**, but now we are changing the positions from **previous query to adjust to current query.**
If previous query was **L=3, R=10**, then we will have **currentL=3** and **currentR=10** by the end of that query. Now if the next query is **L=5, R=7**, then we move the **currentL** to **5** and **currentR** to **7**.
add function means we are adding the element at position to our current set. And updating answer accordingly.

remove function means we are deleting the element at position from our current set. **And updating answer accordingly.**
MO’s algorithm is just an order in which we process the queries. We were given M queries, we will re-order the queries in a particular order and then process them. Clearly, this is an offline algorithm. Each query has L and R, we will call them opening and closing. Let us divide the given input array into Sqrt(N) blocks. Each block will be N / Sqrt(N) = Sqrt(N) size. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.

In this algorithm we will process the queries of **1st block**. Then we process the queries of **2nd block** and so on… finally **Sqrt(N)’th** block. We already have an ordering, queries are ordered in the ascending order of its block. There can be many queries that belong to the same block.

Let’s focus on how we query and answer **block 1**. We will similarly do for all blocks. All of these queries have their opening in **block 1**, but their closing can be in any block including **block 1**. Now let us reorder these queries in ascending order of their **R value**. We do this for all the blocks.

How does the final order look like?

All the queries are first ordered in ascending order of their block number (block number is the block in which its opening falls). Ties are ordered in ascending order of their R value.
For example consider following queries and assume we have 3 blocks

each of size 3.

{0, 3} {1, 7} {2, 8} {7, 8} {4, 8} {4, 4} {1, 2}

Let us re-order them based on their block number.

{0, 3} {1, 7} {2, 8} {1, 2} {4, 8} {4, 4} {7, 8}

Now let us re-order ties based on their R value.

{1, 2} {0, 3} {1, 7} {2, 8} {4, 4} {4, 8} {7, 8}

Now we use the same code stated in previous section and solve the problem. Above algorithm is correct as we did not do any changes but just reordered the queries.

Proof for complexity of above algorithm – O(Sqrt(N) * N)

We are done with MO's algorithm, it is just an ordering.
It turns out that the O(N^2) code we wrote works in O(Sqrt(N) * N) time if we follow the above order.
With just reordering the queries we reduced the complexity from O(N^2) to O(Sqrt(N) * N), and that too with out any further modification to code.

Have a look at our code above, **the complexity over all queries is determined by the 4 while loops**. First 2 while loops can be stated as "**Amount moved by left pointer in total**", second 2 while loops can be stated as "**Amount moved by right pointer**". **Sum of these two will be the over all complexity.**

Let us talk about the right pointer first.

For each block, the queries are sorted in increasing order, so clearly the right pointer **(currentR)** moves in increasing order. During the start of next block the pointer possibly be at extreme end will move to least R in next block. That means for a given block, the amount moved by right pointer is **O(N)**. We have **O(Sqrt(N))** blocks, so the total is **O(N * Sqrt(N))**.

Let us see how the left pointer moves.
For each block, the left pointer of all the queries fall in the same block, as we move from query to query the left pointer might move but as previous L and current L fall in the same block, the moment is **O(Sqrt(N)) (Size of the block)**. In each block the amount left pointer movies is **O(Q * Sqrt(N))** where **Q** is number of queries falling in that block. Total complexity is **O(M * Sqrt(N))** for all blocks.

So, total complexity is **O((N + M) * Sqrt(N)) = O(N * Sqrt(N)).**

THIS ALGORITHM IS OFFLINE

that means we cannot use it when we are forced to stick to given order of queries. That also means we cannot use this when there are update operations. Not just that, there is one important possible limitation: We should be able to write the functions add and remove. There will be many cases where add is trivial but remove is not. One such example is where we want maximum in a range. As we add elements, we can keep track of maximum. But when we remove elements it is not trivial.

Find no. of distinct elements in a range :

```
struct query {
    int l;
    int r;
    int in;
} q[200005];
```

How’ll we sort query according to above defined way:

```
bool compare(query x, query y) {
    if (x.l / BLOCK != y.l / BLOCK) {
        // different blocks, so sort by block.
        return x.l / BLOCK < y.l / BLOCK;
    }
    // same block, so sort by R value
    return x.r < y.r;
}
```

ADD AND REMOVE

```
void add(int cur) {
    cnt[arr[cur]]++;
    if (cnt[arr[cur]] == 1) {
        tans++;
    }
}

void remove(int cur) {
    cnt[arr[cur]]--;
    if (cnt[arr[cur]] == 0) {
        tans--;
    }
}
```

MAIN

```
int main() {

    fastRead_int(n);
    loop(i, 0, n) {
        fastRead_int(arr[i]);
    }

    fastRead_int(t);
    loop(i, 0, t) {
        fastRead_int(a);
        fastRead_int(b);
        a--, b--;
        q[i].l = a, q[i].r = b, q[i].in = i;
    }

    //structure sorted
    sort(q, q + t, compare);
    int curL = 0, curR = 0;
    loop(i, 0, t) {

        while (curL < q[i].l) {
            remove(curL);
            curL++;
        }
        while (curL > q[i].l) {
            add(curL - 1);
            curL--;
        }
        while (curR <= q[i].r) {
            add(curR);
            curR++;
        }
        while (curR > q[i].r + 1) {
            remove(curR - 1);
            curR--;
        }
        ans[q[i].in] = tans;

    }
    for (int i = 0; i < t; ++i) {
        printf("%d\n", ans[i]);
    }
    return 0;
}
```

POWERFUL ARRAY

An array of positive integers **a1,a2,...,an** is given. Let us consider its arbitrary subarray **al,al+1...,ar**, where **1≤l≤r≤n**. For every positive integer s denote by **Ks** the number of occurrences of s into the subarray. We call the power of the subarray the sum of products **Ks·Ks·s** for every positive integer **s**. The sum contains only finite number of nonzero summands as the number of different values in the array is indeed finite.
You should calculate the power of **t** given subarrays.

First line contains two integers n and t **(1≤n,t≤200000)** — the array length and the number of queries correspondingly.
Second line contains n positive integers ai **(1≤ai≤10^6)** — the elements of the array.
Next **t** lines contain two positive integers **l, r (1≤l≤r≤n)** each — the indices of the left and the right ends of the corresponding subarray.

CODE :

```
ll arr[200005];
ll ans[200005];
ll cnt[1000005];

ll tans;
```

```
struct query {
    int l;
    int r;
    int in;
} q[200005];

bool compare(query x, query y) {
    if (x.l / BLOCK != y.l / BLOCK) {
        // different blocks, so sort by block.
        return x.l / BLOCK < y.l / BLOCK;
    }
    // same block, so sort by R value
    return x.r < y.r;
}

void add(int cur) {
    cnt[arr[cur]]++;
    if(cnt[arr[cur]]>0)
        tans += arr[cur] * (2 * cnt[arr[cur]] - 1);
}

void remove(int cur) {
    cnt[arr[cur]]--;
    if(cnt[arr[cur]]>=0)
        tans -= arr[cur] * (2 * cnt[arr[cur]] + 1);
}

int main() {
    //ios_base::sync_with_stdio(false);
    //cin.tie(NULL);

    fastRead_int(n), fastRead_int(t);

    loop(i, 0, n) {
        fastRead_lint(arr[i]);
    }

    loop(i, 0, t) {
        fastRead_int(a), fastRead_int(b);
        a--, b--;
        q[i].l = a, q[i].r = b, q[i].in = i;
    }

    //structure sorted
    sort(q, q + t, compare);
    int curL = 0, curR = 0;
    loop(i, 0, t) {

        while (curL < q[i].l) {
            remove(curL);
            curL++;
        }
        while (curL > q[i].l) {
            add(curL-1);
            curL--;
        }
        while (curR <= q[i].r) {
            add(curR);
            curR++;
        }
        while (curR > q[i].r + 1) {
            remove(curR - 1);
            curR--;
        }
        ans[q[i].in] = tans;

    }

    for (int i = 0; i < t; ++i) {
        prl(ans[i]);
        nl;
    }

    return 0;
}
```

TRY THIS :

TREE AND QUERIES

You have a rooted tree consisting of n vertices. Each vertex of the tree has some color. We will assume that the tree vertices are numbered by integers from 1 to n. Then we represent the color of vertex v as cv. The tree root is a vertex with number 1.

In this problem you need to answer to m queries. Each query is described by two integers vj,kj. The answer to query vj,kj is the number of such colors of vertices x, that the subtree of vertex vj contains at least kj vertices of color x.

INPUT :

```
8 5
1 2 2 3 3 2 3 3
1 2
1 5
2 3
2 4
5 6
5 7
5 8
1 2
1 3
1 4
2 3
5 3
```

OUTPUT :

```
2
2
```

```
1
0
1
```