Solution worth 10 points

Update: We just set a[p] = q – O(1)

Query: We use the standard algorithm for finding the majorant. We keep the current majorant with the number of its occurrences. Let us process another number. If it is equal to the current majorant, we increase by 1 the number of occurrences, otherwise we decrease it by 1. If it happens to end up with negative result, the current number becomes a majorant and the occurrences become 1. It is possible that the subarray does not have a majorant. So we must check whether the majorant is encountered enough number of times. If we keep the occurrences of the numbers in total in std::map, we can achieve a complexity of O(n*n*log(n))

In total: O(m*n^2*log(n))

For the next subtasks, we first compress the numbers.

Solution worth 25 points

Update: Like in the last, but after update we compress again: O(n*log(n))

Query: After compression, the numbers are less or equal to n, so we can keep the occurrences in an array: O(n*n)

In total: O(m*n*n)

Solution worth 45 points

Update: Like in the last: O(n*log(n))

Query: Let us divide the numbers in two groups: "light" and "heavy". Light elements occur less or equal to C times and the heavy occur more than C times in the whole array.

Light numbers can be a majorant of the array, which size is at most 2*C, so we calculate their sum together using the algorithm of the last subtask.

For the heavy number, we first make an array b and set b[i]=1, if a[i] is equal to the current number x and set b[i] = –1, otherwise. If x is a majorant of the subarray [i, j], then $\sum b[k] > 0$, $i \le k \le j$.

Then we make the prefix sums of the array b. Then $\sum b[k] = prefix[j]–prefix[i–1]$, $i \le k \le j$.

Let us add n to every prefix[i] in order to make $0 \le prefix[i] \le 2*n$.

If we choose j, we must find how many of the previous prefixes are in the interval [0, prefix[j]–1].

We can obviously use the Fenwick tree, but this has a complexity of O(n*log(n))

But we can note that |prefix[i] – prefix[i +1]|=1, so we move the interval with exactly 1. We can keep the occurrences of every value of prefix and do *update* and *query* in O(1). In total we have O(n).

For all light numbers we have O(n*C), and O(n) for every heavy number. But there are only n/C heavy number, in total O(n*(C + n/C)). If we choose $C=\sqrt{n}$, the total complexity of a query is $O(n*\sqrt{n})$

In total: $O(m*n*\sqrt{n})$

Solution worth 65 points

Update: We can realize that we erase and insert 1 element. So it is possible to do it in O(n). We must do and update on the segment tree, described in the query. The complexity is the same as for the query.

Query: Here we use divide and conquer.

Let us be on the interval [l, r], av=(l+r)/2.

Divide: Run dc (l, av) and dc (av+1, r).

Conquer (Merge): Now only the subarray with $l \le p \le av$, $av+1 \le q \le r$ are left.

(1) If x is a majorant in [p, q], then x is a majorant in [p, i] and/or [i+1, q]. We can easily prove it by contradiction.
So if we choose i=av, then x is a majorant in [p, av] and/or [av+1, q].
(2) Let us run p from av to l. Then, in the sequence [majorant [p, av]] there are at most log(av–l+1) different numbers. By induction we can prove that if before x there are y different majorants, then when x becomes a majorant for first time, x occurs at least (2 to the y-th power) times. The statement easily follows.

Let assume that the majorant is of the described type and is placed in the left part, maybe in the right, too. Then this majorant can take at most log(av–l+1) values and after we fix it, we can use counting like those used for the heavy numbers.

The case where the majorant is in the right part is similar.

But the majorant can occur in both parts and must not be counted 2 times.

Let F(n) be complexity of the solution.

F(n)=2*F(n/2) + O(n*log(n))

Then F(n)=O(n*log(n)*log(n))

This is slow. We can use segment tree in order not to calculate again an answer, which is already known.

We can prove that on every level (except the first) there are at most 2 merges. So on every query the complexity is

n*log(n)+2*n/2*log(n/2)
+…+2*(n/2^i)*log(n/2^i)<log(n)*[n+2*(n/2+n/4+n/8+…)]<log(n)*[n+2*n]=3*n*log(n)

In total: O(m*n*log(n))

Solution worth 100 points

Here the numbers are needed only sorted.

Update: Like in the last: O(n)

Query: Let us fix the current number. We know on which positions it occurs.

We can note that frequently the subarrays, where it is a majorant are disconnected. They are in some groups.

For example the number 1 in 1 1 7 2 3 4 7 1 1. We will call this “cut line”, because there is no subarray with a majorant the current number (1) including both the left part and the right part.

Actually, in order the i-th (on position p[i]) and the j-th (on position p[j]) index of the current number to have a subarray with a majorant this number, it must be true that

$2*(j-i+1) > p[j]-p[i]+1$

$2*j-2*i+1 > p[j]-p[i] \Leftrightarrow 2*j-2*i \geq p[j]-p[i] \Leftrightarrow 2*j-p[j] \geq 2*i-p[i]$

Let $F[i]=2*i-p[i]$

If there is a “cut line” between i and i+1, it is true that $F[p]>F[q]$ for every $p<=i$ and $q>i$.

This means that MIN[F[1],F[2]…F[i]]>MAX[F[i+1],F[i+2]…F[k]], k-the number of occurrences of the current number.

We can easily find them in O(k)

Let us see an example:

1 1 2 3 2 1 2 1 2 2 1, current number is 1.

k=5, p=[1,2,6,8,11], f=[1,2,0,0,-1] and there are two2 “cut lines” - 1 2 ||| 0 0 ||| -1

When we cut, we must take and some different numbers from the both sides.

Let us assume that there are no “cut lines” in the current subarray. Let the size of the current subarray be d. Obviously, we can keep only d numbers in both sides, which are “carried” together with the current subarray. What is the maximal possible size of the subarray? It turns out it is 5*d. So we can use the algorithm for the heavy numbers.

Why? We can discard the 2 sides, which are carried (of sizes d each). So we must prove that the distance between the first and the last occurrence is at most 3*d.

Let us set $p[1]=1$ and then $F[1]=1$. So we must prove that $p[d] \leq 3*d$. To be exact, $p[d] \leq \max(2*d-1, 3*d-4)$, but $p[d]<=3*d$ is enough for our needs.

Let us divide the numbers in maximal groups, equal to the current number. For every group [l(l-th occurrence), r(r-th occurrence)] is true that $F[l]+1=F[l+1]$, $F[l+1]+1=F[l+2]$ … $F[r-1]+1=F[r]$. If [l, r] and [r+1, q] are 2 adjacent groups, then $F[r]>=F[r+1] \Leftrightarrow 2*j-p[j]>=2*(j+1)-p[j+1] \Leftrightarrow p[j+1]>= p[j]+2$, which is true because p[j+1] and p[j] are not connected, because they are maximal.

Let us make a graph with nodes for every whole number.

For every group [l,r] we connect, F[l] with F[l+1], F[l+1] with F[l+2] … F[r-1] with F[r]. If there are not “cut lines” F[1] must be connected with F[d]. But this graph has less then d edges, so $F[d]>F[1]-d$

$2*d-p[d]>1-d \Rightarrow p[d]<3*d-1$.

Why F[1] must be connected with F[d]?

Let us note that F[i]+1 = F[i+1] or F[i] ≥ F[i+1] and the edges are only of the type (m,m+1).

Let F[1] and F[d] be not connected. Let us assume that the edge (p,p+1) is missing and it is on the path between F[1] and F[d]. Up to some moment it is always true that F[i] > p and let j be the first moment with F[j] ≤ p. Then F[l] ≤ p for every j ≤ l. Thus j-1 and j make a "cut line", because F[i] > p ≥ F[l] for i < j and j ≤ l. Contradiction.

The total complexity for one query becomes ~O(n+5*n), because for 5*d operations we remove d numbers.

In total: O(m*n)

We note that the constants of the last 2 solutions are too big and they make the solutions slower and thus lead to smaller constraints.

Author: Martin Kopchev