Chef was invited to the party of **N** people numbered from **1** to **N**. Chef knows the growth of all the people, i.e. he knows the growth of the $\mathbf{i}^{\text{th}}$ person is denoted by an integer $\mathbf{A_i}$ not exceeding **M**.

Chef decided to have some fun. At first, he forms **K**groups of people. The $\mathbf{i}^{\text{th}}$ group consists of all the people numbered from $\mathbf{L_i}$ to $\mathbf{R_i}$. Groups may overlap too.

For each group, Chef wants to know the following information: the maximum difference between the numberings of two people having same growth. Formally, Chef wants to know the following:

$$\max\{|x - y| : L_i \le x, y \le R_i \text{ and } A_x = A_y\}$$

Please help Chef to have fun.

Input

There is only one test case in one test file.

The first line of input contains three space-separated integers **N**, **M** and **K**, denoting the number of people, the maximum growth and number of groups respectively. The second line contains **N** space-separated integers $\mathbf{A_1}$, $\mathbf{A_2}$, ..., $\mathbf{A_N}$ denoting the growth of people. Then the $\mathbf{i}^{\text{th}}$ line of the next **K** lines contains two space-separated integers $\mathbf{L_i}$, $\mathbf{R_i}$, denoting the $\mathbf{i}^{\text{th}}$ group.

Output

For each group, output the integer denoting the maximum difference between numbering of two people having same growth in a single line.

Constraints and Subtasks

- $1 \le A_i \le M$
- $1 \le L_i \le R_i \le N$

Subtask 1: 20 points

- $1 \le N, M, K \le 1000 = 10^3$

Subtask 2: 80 points

- $1 \le N, M, K \le 100000 = 10^5$

Example

```
Input:
7 7 5
4 5 6 6 5 7 4
```

```
6 6
5 6
3 5
3 7
1 7

Output:
0
0
1
1
6
```

Explanation

Group 1. There is only one person in the group. Thus the maximum difference of numbers should be **0**.

Group 2. There are two persons in the group. Their growth are $\mathbf{A_5 = 5}$ and $\mathbf{A_6 = 7}$. Thus there is no pair of persons who have the same growth. Thus the answer for this group will also be **0**.

Group 3. There are three persons in the group. Their growth are $\mathbf{A_3 = 6}$, $\mathbf{A_4 = 6}$ and $\mathbf{A_5 = 5}$. Here person **3** and person **4** has the same growth. Thus the answer is $\mathbf{|4 - 3| = 1}$.

Group 4. There are more persons than the group **3**. But they has different growth, other than person **3** and person **4**.Thus the answer is also $\mathbf{|4 - 3| = 1}$.

Group 5. This group contains all the people, and person **1**and person **7** has the same growth $\mathbf{A_1 = A_7 = 4}$. So the answer is $\mathbf{|7 - 1| = 6}$.

IFFICULTY:
MEDIUM

PREREQUISITES:
sqrt decomposition, preprocessing

PROBLEM:
Given a sequence of N integers $A_1, A_2, \ldots, A_N$, where each A_i is between 1 to M, you are to answer Q queries of the following kind:

- Given L and R, where $1 \le L \le R \le N$, what is the maximum $|x - y|$ such that $L \le x, y \le R$ and $A_x = A_y$?

Note that in the problem, Q is actually K.

QUICK EXPLANATION:
For each i, $1 \le i \le N$, precompute the following in $O(N)$ time:

- $\text{next}[i]$, the smallest $j > i$ such that $A_i = A_j$
- $\text{prev}[i]$, the largest $j < i$ such that $A_i = A_j$

Let $S = \lfloor \sqrt{N} \rfloor$, and $B = \lceil N/S \rceil$. Decompose the array into B blocks, each of size S (except possibly the last). For each i, $1 \le i \le N$, and $0 \le j \le B - 1$, precompute the following in $O(N\sqrt{N})$ time:

- $\text{last\_in\_blocks}[j][i]$, the largest $k \le jS + S$ such that $A_k = A_i$
- $\text{block\_ans}[j][i]$, the answer for the query $(L, R) = (jS + 1, i)$. For a fixed j, all the $\text{block\_ans}[j][i]$ can be computed in $O(N)$ time.

Now, to answer a query (L, R), first find the blocks j_L and j_R where L and R belong in ($0 \le j_L, j_R < B$). Then the answer is at least $\text{block\_ans}[j_L + 1][R]$, and the only pairs (x, y) not yet considered are those where $L \le x \le j_L S + S$. To consider those, one can simply try all x in that range, and find the highest $y \le R$ such that $A_x = A_y$. Finding that y can be done by using $\text{last\_in\_blocks}[j_R - 1][x]$ and a series of next calls. To make that last part run in $O(S)$ time, consider only the x such that $\text{prev}[x] < L$.

EXPLANATION:
We'll explain the solution for subtask 1 first, because our solution for subtask 2 will build upon it. However, we will first make the assumption that $M \le N$, otherwise we can simply replace the values $A_1, \ldots A_N$ with numbers from $1$to N, and should only take $O(N)$ time with a set. However, we don't recommended that you actually do it; this is only to make the analysis clearer.

$O(N^2)$ per query

First, a simple brute-force $O(N^2)$-time per query is very simple to implement, so getting the first subtask is not an issue at all. I'm even providing you with a pseudocode on how to do it :)

```
def answer_query(L, R):
    for d in R-L...1 by -1
        for x in L...R-d
            y = x+d
            if A[x] == A[y]
                return d

    return 0
```

We're simply checking every possible answer from $[0, R - L]$ in decreasing order. Note that the whole algorithm runs in $O(QN^2)$ time, which *could* get TLE if the test cases were stronger. But in case you can't get your solution accepted, then it's time to optimize your query time to...

$O(N)$ per query

To obtain a faster running time, we have to use the fact that we are finding the maximum $|x - y|$. What this means is that for every value v, we are only concerned with the first and last time it occurs in $[L, R]$.

We first consider the following alternative $O(N^2)$-time per query solution:

```
def answer_query(L, R):
    answer = 0
    for y in L...R
        for x in L...y
            if A[x] == A[y]
                answer = max(answer, y - x)

    return answer
```

The idea here is that for every y, we are seeking A_x, which is the first occurrence of A_y in $[L, y]$, because all the other occurrences will result in a smaller $y - x$ value. Now, to speed it up, notice that we don't have to recompute this x every time we encounter the value A_x, because we are already reading the values $A_L, \ldots, A_R$ in order, so we already have the information "when did A_y first appear" before we ever need it! Here's an implementation (in pseudocode):

```
def answer_query(L, R):
    index = new map/dictionary
    answer = 0
    for y in L...R
        if not index.has_key(A[y])
            index[A[y]] = y
        answer = max(answer, y - index[A[y]])

    return answer
```

Now, notice that this runs in $O(N)$ time if one uses a hash map for example!

We mention here that it's possible to drop the use of a hash map by using the fact that the values A_y are in $[1, M]$. This means that we can simply allocate an *array* of length M, instead of creating a hash map from scratch or clearing it. However, we must be careful when we reinitialize this array, because it is long! There are two ways of "initializing" it:

- We clear the array every time we're done using it, but we only clear those we just encountered. This required listing all the indices we accessed.
- We maintain a parallel array that contains when array was last accessed for each index. To clear the array, we simply update the *current time*.

We'll show how to do the second one:

```
class LazyMap:
    index[1..M]
    found[1..M] # all initialized to zero
    time = 0

    def clear():
        this.time++

    def has_key(i):
        return this.found[i] == this.time

    def set(i, value): # called on the statement x[i] = value for example
        this.found[i] = this.time
```

```
        this.index[i] = value

    def get(i): # called on the expression x[i] for example
        return this.index[i]

index = new LazyMap()

def answer_query(L, R):
    index.clear()
    answer = 0
    for y in L...R
        if not index.has_key(A[y])
            index[A[y]] = y
        answer = max(answer, y - index[A[y]])

    return answer
```

Using this, the algorithm still runs in $O(N)$ time (remember that we assume $M \le N$), but most likely with a lower constant.

The overall algorithm runs in $O(QN)$ time.

sqrt decomposition

When one encounters an array with queries in it, there are usually two ways to preprocess the array so that the queries can be done in sublinear time:

- **sqrt decomposition**, which splits up the array into $\lceil N/S \rceil$ blocks of size S each. S is usually taken to be $\lfloor \sqrt{N} \rfloor$(hence the term "sqrt decomposition"). Usually, one can reduce the running time to $O((N+Q)\sqrt{N})$ or $O((N+Q)\sqrt{N \log N})$. Sometimes, depending on the problem, it may also yield $O((N+Q)N^{2/3})$ time.
- **build some tree structure on top of the array**. This usually yields an $O(N + Q \log N)$ or $O((N+Q) \log N)$time algorithm.

There are other less common ways, such as **lazy updates** or combinations of the above, but first we'll try out whether the above work.

Suppose we have selected the parameter S, and we have split the array into $B = \lceil N/S \rceil$ blocks of size S, except possibly the last block which may contain fewer than S elements. Suppose we want to answer a particular query (L, R). Note that L and R will belong to some block. For simplicity, we assume that they belong to different blocks, because if they are on the same block, then $R - L \le S$, so we can use the $O(S)$ time query above.

Thus, the general picture will be:

```
|...........|...........|...........|...........|...........|...........|...........|
                     ^   ^                                    ^      ^
                     L   E_L                                  E_R    R
```

We have marked two additional points, E_L and E_R, which are the boundaries of the blocks completely inside $[L, R]$. Now, it would be nice if we have already precomputed the answer for the query pair (E_L, E_R), because then we will only have to deal with at most $2(S-1)$ remaining values: $[L, E_L)$ and $[E_R, R]$. We can indeed precompute the answers at the boundaries, but we can do even better: we can precompute the answers for all pairs (E, R), where $E$is a boundary point and R is *any* point in the array! There are only $O(BN)$ pairs, and we can compute the answers in $O(BN)$ time also:

```
class LazyMap:
    ...

S = floor(sqrt(N))
B = ceil(N/S)
index = new LazyMap()
```

```
block_ans[1..B][1..N]
def precompute():
    answer = 0
    for b in 1...B
        index.clear()
        E = b*S-S+1 # left endpoint of the b'th block
        answer = 0
        for R in E...N
            if not index.has_key(A[R])
                index[A[R]] = R
            answer = max(answer, R - index[A[R]])
            block_ans[b][R] = answer
```

(if you read the "quick explanation", note that there is a slight difference here: we're indexing the blocks from 1 to Binstead of 0 to $B - 1$)

This means that, in the query, the only remaining values we haven't considered yet are those in $[L, E_L)$. To consider those, we have to know, for each x in $[L, E_L)$, the last occurrence of A_x in $[L, R]$. To do so, we will need the following information:

- $next[i]$, the smallest $j > i$ such that $A_i = A_j$
- $prev[i]$, the largest $j < i$ such that $A_i = A_j$
- $last\_in\_blocks[j][i]$, the largest k within the first j blocks such that $A_k = A_i$

How will this help us? Well, we want to find A_x's last occurrences in $[L, R]$. So first, we find its last occurrence in the blocks up to E_R (it's just $last\_in\_blocks[floor(R/S)][x]$). However, it's possible that A_x appears in $[E_R, R]$, so we need to use its $next$ pointers, until we find the *last* one. Since there are at most $S - 1$ elements in $[E_R, R]$, this *seems* fast, but it could easily take $O(S^2)$ time for example when most of the values in $[L, E_L)$ and $[E_R, R]$ are equal. Thankfully, this is easily fixed: we only care about the *first* occurrence of A_x, so if it has been encountered before, then we don't have to process it again! This ensures that for *distinct* value in $[E_R, R]$, its set of indices is iterated only once. This therefore guarantees an $O(S)$ running time!

Checking whether an A_x has been encountered before can also be done using the `index` approach, or alternatively as $prev[x] \geq L$:

```
def answer_query(L, R):
    b_L = ((L+S-1)/S)
    b_R = R/S
    if b_L >= b_R
        # old query here
    else
        E_L = b_L*S
        answer = block_ans[b_L+1][R]
        for x in L...E_L
            if prev[x] < L # i.e. x hasn't been encountered before
                y = last_in_blocks[floor(R/S)][x]
                while next[y] <= R
                    y = next[y]
                answer = max(answer, y - x)

    return answer
```

One can now see that the query time is now $O(S)$:) Note that $b_L \leq b_R$ means that L and R are within $O(S)$ elements away, so we can do the old query instead.

Let's now see how to precompute $next$, $prev$ and $last\_in\_blocks$. First, $next[i]$ and $prev[i]$ can easily be computed in $O(N)$ time with the following code:

```
...
next[1..N]
prev[1..N]
last[1..M] # initialized to 0
```

```
...

def precompute():
    ...

    for i in 1...N
        next[i] = N+1
        prev[i] = 0

    for i in 1...N
        j = last[A[i]]
        if j != 0
            next[j] = i
            prev[i] = j
        last[A[i]] = i
```

The `last` array stores the last index encountered for every value, and is updated as we traverse the array.

And then last_in_blocks can be compute in $O(BN)$ time:

```
...
last_in_blocks[1..B][1..N] # initialized to 0
...

def precompute():
    ...
    for b in 1...B
        L = b*S-S+1
        R = min(b*S,N)
        for y in L...R
            if next[y] > R
                x = y
                while x > 0
                    last_in_blocks[b][x] = y
                    x = prev[x]

    for x in 1...N
        for b in 2...B
            if last_in_blocks[b][x] == 0
                last_in_blocks[b][x] = last_in_blocks[b-1][x]
```

The first loop finds the last value encountered at each block (with the check `next[y] > R`), and proceeds setting the `last_in_blocks` of all the indices until that position with equal value, using the `prev` pointer. The second loop fills out the remaining entries, because some values do not have representatives in some blocks.

Running time

Now, what is the total running time then? The precomputation runs in $O(BN)$ time, and each query takes $O(S)$ time, so overall it is $O(NB + QS)$. But remember that $B = \Theta(N/S)$, so the algorithm is just $O(N^2/S + QS)$. But we still have the freedom to choose the value of S. Now, most will simply choose $S = \Theta(\sqrt{N})$, so that the running time is $O((N + Q)\sqrt{N})$, but we are special, so we will be more pedantic.

Note that N^2/S is a decreasing function while QS is an increasing function. Also, remember that $O(f(x) + g(x)) = O(\max(f(x), g(x)))$ (why?). Therefore, the best choice for S is one that makes N^2/S and QSequal (at least asymptotically). Thus, we want the choice $S = \Theta(N/\sqrt{Q})$ instead, and the running time is $O(N\sqrt{Q} + Q)$:) (the +Q is there to account for when $Q > N^2$). For this problem, there's not much difference between this and $O((N + Q)\sqrt{N})$, but the running time $O(N\sqrt{Q} + Q)$ is mostly for theoretical interest, and when Q is much less than N (or much more), you'll feel the difference.

Optimization side note: there is another way to do the old query without using our `LazyMap`, or at least calling `has_key`: *traverse the array backwards*. Here is an example:

```
...
_index[1..M]
...

def answer_query(L, R):
    ...
    if b_L >= b_R
        # old query
        answer = 0
        for y in R...L by -1
            _index[A[y]] = y

        for y in L...R
            answer = max(answer, y - _index[A[y]])
    else
        ...

    return answer
```

I found that this is a teeny teeny bit faster than the original $O(S)$ old query :)

Also, when choosing S, one does not have to choose $\lfloor\sqrt{N}\rfloor$, or even $\lfloor N/\sqrt{Q}\rfloor$, because there is still a constant hidden in the Θ notation. This means that you still have the freedom to choose a multiplicative constant for S, which in practice essentially amounts to the freedom to select S however you want. To get the best value for S, try generating a large input (with varying values of M !), and finding the best choice for S via ternary search. The goal is to get the precomputation part and the query part roughly equal in running time. This technique of **tweaking the parameters** is incredibly useful in long contests where the time limit is usually tight.

Time Complexity:

$O(M + (N + Q)\sqrt{N})$ but a theoretically it is $O(N\sqrt{Q} + Q)$

Note that in the problem, Q is actually K.

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1e5+5;
const int MAXV = 1e5+5;
const int SQRN = 350;
int N, M, Q, BLOCKN, BLOCK_SIZE, a[MAXN];
int Prev[MAXN], Next[MAXN], last[MAXV], indexx[MAXV], found[MAXV];
int last_in_blocks[SQRN][MAXN], block_ans[SQRN][MAXN];

void preprocess(){

    BLOCK_SIZE = floor(sqrt(N));
    BLOCKN = ceil(N/BLOCK_SIZE);

    for(int i=1; i<=M; i++) last[i] = 0;
    for(int i=1; i<=N; i++) Prev[i] = 0;
    for(int i=1; i<=N; i++) Next[i] = N+1;

    for(int i=1; i<=N; i++){
        if(last[a[i]] != 0){
            Next[last[a[i]]] = i;
            Prev[i] = last[a[i]];
        }
        last[a[i]] = i;
    }

    memset(last_in_blocks, 0, sizeof(last_in_blocks));
    for(int b=1; b<=BLOCKN; b++){
        int L = (b*BLOCK_SIZE) - BLOCK_SIZE + 1;
        int R = min(b*BLOCK_SIZE, N);
        for(int y=L; y<=R; y++){
            if(Next[y]>R){
                int x = y;
                while(x>0){
                    last_in_blocks[b][x] = y;
                    x = Prev[x];
                }
            }
        }
    }
    for(int x=1; x<=N; x++){
        for(int b=2; b<=BLOCKN; b++){
            if(last_in_blocks[b][x]==0){
                last_in_blocks[b][x] = last_in_blocks[b-1][x];
            }
        }
    }

    int mytime = 0;
    memset(found,0,sizeof(found));
    memset(indexx,0,sizeof(indexx));
    for(int b=1; b<=BLOCKN; b++){
        mytime++;
        int ans = 0;
        int L = (b*BLOCK_SIZE) - BLOCK_SIZE + 1;
        for(int y=L; y<=N; y++){
            if(found[a[y]]!=mytime){
                indexx[a[y]] = y;
                found[a[y]] = mytime;
            }
            ans = max(ans, y-indexx[a[y]]);
            block_ans[b][y] = ans;
        }
    }
}

```

```cpp
int queryInBlock(int L,int R){
    map<int,int>MP;
    int ans = 0;
    for(int x=L; x<=R; x++){
        if(MP.find(a[x])==MP.end()){
            MP[a[x]] = x;
        }
        ans = max(ans, x-MP[a[x]]);
    }
    return ans;
}

int solve(int L, int R){
    int Lblock = (L+BLOCK_SIZE-1) / BLOCK_SIZE;
    int Rblock = (R+BLOCK_SIZE-1) / BLOCK_SIZE;
    int ans = 0;
    if(Lblock==Rblock || Lblock+1==Rblock){
        ans = queryInBlock(L,R);
    }
    else{
        int LblockEnd = Lblock*BLOCK_SIZE;
        ans = block_ans[Lblock+1][R];
        for(int x=L; x<=LblockEnd; x++){
            if(Prev[x]<L){
                int y = last_in_blocks[Rblock-1][x];
                while(Next[y]<=R){
                    y = Next[y];
                }
                ans = max(ans, y-x);
            }
        }
    }
    return ans;
}

int main(){

    scanf("%d%d%d",&N,&M,&Q);
    for(int i=1; i<=N; i++){
        scanf("%d",&a[i]);
    }

    preprocess();

    for(int i=1; i<=Q; i++){
        int L,R; scanf("%d%d",&L,&R);
        if(L>R)swap(L,R);
        int ans = solve(L,R);
        printf("%d\n",ans);
    }
    return 0;
}
```