Disjoint Sets

A disjoint set contains a set of sets such that in each set, an element is designated as a marker for the set. Here is a simple disjoint set:

{1}, {2}, {3}, {4}, {5}

clearly there can only be one marker for each of these sets. Given a disjoint sets, we can edit them using the union operation. For example:

union(1,3) would make our structure look like:

{1,3}, {2}, {4}, {5}

Here we would have to designate either 1 or 3 as the marker. Let's choose 1. Now consider doing these two operations:

union(1,4)
union(2,5) (Assume 2 is marked.)

Now we have:

{1,3,4}, {2,5}

Now, we can also do the findset operation.

findset(3) should return 1, since 1 is the marked element in the set that contains 3.

Disjoint Set Implementation

A set within disjoint sets can be represented in several ways. Consider {2, 4, 5, 8} with 5 as the marked element. Here are a few ways that could be stored:

```
   5            5          5
 / | \         /   \        |
2  4   8      2     8       8
              |            / \
              4           4   2
```

We can actually store a disjoint set in an array. For example, the sets {2,4,5,8}, {1}, {3,6,7} could be stored as follows:

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

The 5 stored in array location 2 signifies that 5 is 2's parent. The 2 in array location 8 signifies that 2 is 8's parent, etc.

Here is the visual display:

```
1              5              7
             /   \          /   \
            2     4        3     6
            |
            8
```

Based on this storage scheme, how can do implement the initial makeset algorithm and how can we implement a findset algorithm?

Union Operation

Given two values, we must first find the markers for those two values, then merge those two trees into one.

Consider union(5,1). We could do either of the following:

```
   1           5
   |         / | \
   5        2  4  1
 /   \      |
2     4     8
|
8
```

We prefer the latter since it minimizes the height of the tree. Thus, in order to implement our disjoint sets efficiently, we must also keep track of the height of each tree, so we know how to do our merges. Basically we choose which tree to merge with which based on which tree has a smaller height. If they are equal we are forced to add 1 to the height of the new tree.

Here is how our array will change for each of the options above:

First option

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

Second option

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

Notice how quickly we can implement that change in the array!

Path Compression

One last enhancement we can add to disjoint sets is path compression. Every time we are forced to do a findset operation, we can directly connect each node on the path from the original node to the root. Here's the basic idea:

```
   1             final tree is            1
   |                                    / | \
   5                                   2  8  5
 /   \                                       |
2     4                                      4
|
8
```

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

First, you find the root of this tree which is 1. Then you go through the path again, starting at 8, changing the parent of each of the nodes on that path to 1.

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

then, you take the 2 that was previously stored in index 8, and then change the value in that index to 1:

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

It has been shown through complicated analysis that the worst case running time of t operations is $O(t\alpha(t,n))$. Note that $\alpha(t,n) \leq 4$ for all $n \leq 10^{19728}$, so for all practical purposes on average, each operation takes constant time.

Application of Disjoint Sets - Kruskal's Algorithm

The most common application of a disjoint set is to detect cycles in a graph formed by a set of edges that are added. This is used in Kruskal's Algorithm. In Kruskal's algorithm, we iteratively add edges to a minimum spanning tree (MST) being built, considering edges in the order of their weights, from smallest to largest. The only exception is that edges that create a cycle when added to the current set of edges already in the MST should be skipped.

Initially, when we start Kruskal's algorithm, we start with an empty disjoint set of n separate trees. Each time we add an edge, we union the two trees storing the two vertices being connected by an edge. This amounts to connecting two separate components of the current graph.

Say we have the following disjoint set in the middle of running Kruskal's:

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

This means that vertices 1, 2, 4, 5 and 8 are one connected component of the MST we are building and 3, 6 and 7 are the other connected component.

Say the next smallest edge was connecting vertices 2 and 8. In this scenario, we run find(2) and find(8). We see that both are rooted at 1. This tells us that both vertices are in the same component of the graph and that adding this edge would create a cycle.

On the other hand, lets say that the next smallest edge after that connected vertices 3 and 4. We run find(3) to obtain 7 and find(4) to obtain 1. Since these are not equal, it means that vertices 3 and 4 are in different connected components and that it's safe to add this edge to our MST. (In this case, this would leave only one tree in the disjoint set, so our MST would be complete.)

Another Disjoint Set Application

Here is a problem from this past USACO US Open Gold Division (2016):

Farmer John and his cows are planning to leave town for a long vacation, and so FJ wants to temporarily close down his farm to save money in the meantime. The farm consists of N barns connected with M bidirectional paths between some pairs of barns ($1 \leq N, M \leq 200{,}000$). To shut the farm down, FJ plans to close one barn at a time. When a barn closes, all paths adjacent to that barn also close, and can no longer be used. FJ is interested in knowing at each point in time (initially, and after each closing) whether his farm is "fully connected" -- meaning that it is possible to travel from any open barn to any other open barn along an appropriate series of paths. Since FJ's farm is initially in somewhat in a state of disrepair, it may not even start out fully connected.

INPUT FORMAT (file closing.in):

The first line of input contains N and M. The next M lines each describe a path in terms of the pair of barns it connects (barns are conveniently numbered $1 \ldots N$). The final N lines give a permutation of $1 \ldots N$ describing the order in which the barns will be closed.

OUTPUT FORMAT (file closing.out):

The output consists of N lines, each containing "YES" or "NO". The first line indicates whether the initial farm is fully connected, and line $i+1$ indicates whether the farm is fully connected after the i^{th} closing.

SAMPLE INPUT:

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

SAMPLE OUTPUT:

```
YES
NO
YES
YES
```

The key to this problem is to imagine this process in reverse, starting with no barns open and iteratively opening them. Imagine running a disjoint set like this, in reverse. At each point in time, we can see how many separate trees are in our disjoint set. If the answer is 1, then every open barn is connected, otherwise they are not. A relatively slight modification to a regular disjoint set class will answer this query.

The crux of the idea is that we can add a variable numTrees to our Disjoint Set class. Initially there are n trees in our disjoint set of n elements. Each time we do an union between two different components (when union returns true indicating that the two input nodes were in different components), we subtract 1 from the number of trees.

Now, to solve the problem, we start with a new disjoint set and "add back" the farms in reverse order, in some sense, opening them with time going backwards. When we open a farm, we union its vertex with all of its neighbors (directly connected by an edge). When we are done with this operation, there is a certain number of trees we expect if our "main" tree is fully connected (meaning the rest of the trees are single nodes). Here is the segment of code from my solution that handles this:

```
        // Go backwards through the list of deletions.
        for (int i=n-2; i>=0; i--) {

                int item = remList[i];
                for (int j=0; j<graph[item].size(); j++) {
                        int next = (Integer)(graph[item].get(j));
                        if (inGraph[next]) {
                                dj.union(item, next);
                        }
                }

                res[i] = (dj.numTrees == i+1);
                inGraph[item] = true;
        }
```

The second to last line is storing whether or not a single connected tree exists after "adding" farm i in reverse order. For this to be true, there must be precisely i+1 trees. My full solution is here:

http://www.cs.ucf.edu/~dmarino/progcontests/mysols/highschool/usaco/2015/gold/closing.java

2016 SI@UCF Final Contest Problem: Headache

Here is the description of the problem:

Lillian's class of ***N*** children is about to go on a field trip to Seaworld to see Shamu for the last time before he "retires", as told to the kids. Since the class is so big it can be difficult to chaperone the whole group together so Lillian wishes to split the class into two groups. The problem is some children frequently argue and gets into fights with each other and if they were in the same group this could cause problems. This causes big headaches for Lillian. She wishes to minimize her headaches and needs your help to figure out the best split of children such that the largest conflict between any pair of children in the same group is minimized. (For example, if one split causes conflicts of size 5, 8 and 12 in group 1 and conflicts of size 4, and 7 in group 2, Lillian's headache is of size 12, the maximum of all the conflicts. Alternatively, if a different split causes conflicts of size 6, 7 and 9 in group 1 and conflicts of size 4 and 8 in group 2, then Lillilan's headache is of size 9 and this second arrangement is preferable to the first.)

The bounds on the number of children AND the number of pairs of children who get into fights is no more than 20,000.

Upon a first read, there doesn't seem to be any connection with a disjoint set. The students can be modeled as vertices in a graph and the amount of headache pairs of students cause can be viewed as edge weights. However, it's still not clear even though we have a graph representation of the data, how a disjoint set comes in.

What's tricky is that the edges in this graph represent students we DON'T want in the same group, but the graph interpretation of an edge is precisely the opposite. One idea to consider would be to look at the complement graph (the one with all missing edges drawn in and all current edges removed.) But this likely leads to a dead end.

In some graph questions, there's a "classic" trick called splitting the node. Instead of just assigning one vertex to a logical component of a problem, one can assign multiple vertices to a single component, in essence, "splitting that component." In this question, we assign two vertices to each student. When we want to try to place two students in different groups, we place an edge between their "counterparts" on different sides.

If we try a greedy approach, we do the following: let's say Bob and Gary cause us the most headache. We immediately put them on two different groups. Next, let's say that Sarah and Susie cause us the next biggest headache. We also want them to be on different sides…BUT…we don't know if we want Sarah on Bob's side or Gary's side for now. But…if we have two sets of vertices for each person, we don't have to decide yet!!! The idea is to place pairs of students from the most annoying pairs on down on different groups until it's impossible to do so; ie, until we get a contradiction. The key type of contradiction we can ultimately have with this sort of set up is that Bob is on a different team than Bob. The edge that causes this contradiction, then, represents the minimum headache Lillian can have over all possible ways to assign the students.

So, the basic algorithm is this:

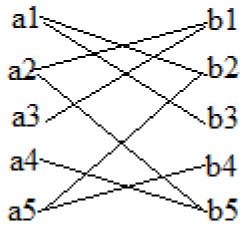
1) Sort edges (pairs who cause headaches) from greatest headache to least headache.
2) Set up an empty Disjoint Set of size 2n, where n is the number of students. Label the vertices a_1, a_2, a_3…, a_n, b_1, b_2, b_3,…,b_n.
3) Loop through the sorted list of edges.

> 3a) For an edge connecting vertices u and v, do a union in our disjoint set between the sets for a_u and b_v as well as a_v and b_u. So, basically, in either interpretation, we are preventing u and v from being on the same group.
>
> 3b) Now, check to see if either a_u and b_u are in the same set or if a_v and b_v are in the same set. If either pair is this is impossible, because it means we've placed either u or v on a different group than themselves. At this point, break out and the weight of this particular edge is the desired result.

Consider the following example of edges:

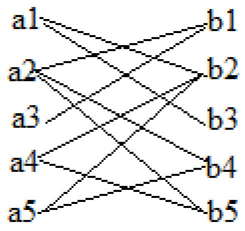
2 - 5, weight 7
1 - 3, weight 5
1 - 2, weight 4
4 - 5, weight 4
2 - 4, weight 3

After processing the first four edges, our graph looks like this:



Our Disjoint set has the following groups: {a1, a5, b2, b3, b4} and {a2, a3, a4, b1, b5}. What this means is that with the current state of affairs (splitting apart (2, 5), (1, 3) and (1,2), we could create teams {1,5} and {2,3,4} and avoid the headaches of sizes 7, 5, 4 and 4 respectively.

But, when we add the edge 2-4, you'll see that a2 and b4 are in different sets and this will merge all the nodes into a single set, and namely when we test to see if a2 and b2 are in the same set, they will be:



<u>2016 Nordic Collegiate Programming Contest Problem: Artwork</u>

In this problem, you are given a rectangular grid that starts out with all white squares. On this grid, you'll paint black vertical and horizontal strips. After each strip is painted the goal is to count the number of separate connected white regions.

A very straight-forward solution is to run a floodfill after each paint operation and count the regions. Unfortunately, r and c can be up to 1000 and the number of paint operations is up to 10^4. So, this straight-forward algorithm would take roughly $1000^2 x 10^4 = 10^{10}$ steps which is way too much for a programming contest.

One piece of information that is useful is knowing at what time a particular square is painted black. (Once a square is painted black, it stays black. But when it's first painted black, there is a potential for there to be a change in the number of connected white components.)

This information is easy to compile in a straight-forward way where you just trace each paint step. (A single paint job can't take more than 1000 steps since that's the longest possible length of a strip and there are only 10^4 such paintings done and 10^7 will run in time.)

Now, once you have this information, just like the USACO problem, the trick is to reverse the order of painting and go backwards in time, and use a disjoint set, with one vertex for each square. Two squares are connected in the disjoint set if they are neighboring squares that are white. So, after all the painting, go through the whole grid, find any white square and union it with any of its white neighbors. This is different than a floodfill, but for our purposes, it achieves what we want: The number of components in our disjoint set and the number of white squares total can be used to calculate the number of distinct white regions.

Now, we step back in time. Let's say our simulation was 10,000 paintings. Find each square that was painted at time 10,000 for the first time and "unpaint" them. To unpaint them, just union these squares with any neighboring white squares, no recursion necessary since the disjoint set naturally keeps track of which components are together. After we finish these unpaintings for a time step, we just recount the number of white components as previously mentioned.

The problem spec is included on the following page and my solution is here:

<u>http://www.cs.ucf.edu/~dmarino/progcontests/mysols/collegeregional/ncpc/2016/a.java</u>

Problem A
Artwork
Problem ID: artwork
Time limit: 4 seconds

A template for an artwork is a white grid of $n \times m$ squares. The artwork will be created by painting q horizontal and vertical black strokes. A stroke starts from square (x_1, y_1), ends at square (x_2, y_2) ($x_1 = x_2$ or $y_1 = y_2$) and changes the color of all squares (x, y) to black where $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$.

The beauty of an artwork is the number of regions in the grid. Each region consists of one or more white squares that are connected to each other using a path of white squares in the grid, walking horizontally or vertically but not diagonally. The initial beauty of the artwork is 1. Your task is to calculate the beauty after each new stroke. Figure A.1 illustrates how the beauty of the artwork varies in Sample Input 1.

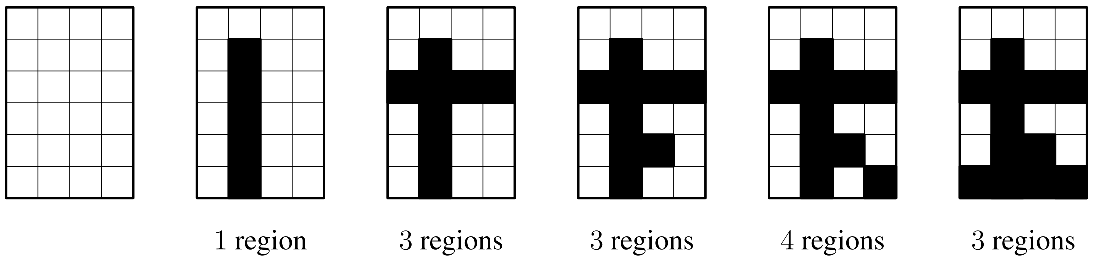


Figure A.1: Illustration of Sample Input 1.

Input

The first line of input contains three integers n, m and q ($1 \leq n, m \leq 1000$, $1 \leq q \leq 10^4$).

Then follow q lines that describe the strokes. Each line consists of four integers x_1, y_1, x_2 and y_2 ($1 \leq x_1 \leq x_2 \leq n$, $1 \leq y_1 \leq y_2 \leq m$). Either $x_1 = x_2$ or $y_1 = y_2$ (or both).

Output

For each of the q strokes, output a line containing the beauty of the artwork after the stroke.

Sample Input 1

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

Sample Output 1

```
1
3
3
4
3
```