CS 3345 Programming Project 3

Program a class for a union-find data structure (called UnionFind) that uses the smart *union by size* algorithm and *path compression* on find operations. Your program will read from `System.in`, make calls on the UnionFind's methods, and output to `System.out`

`UnionFind supports the following methods:`

```
UnionFind(int n);    // creates a union find object for integer elements 0 ... n-1

union(int x, int y); // forms the union of elements x and y using the union by size strategy.
                     // If the sizes of the trees containing x and y are the same make
                     // the tree containing y a subtree of the root of the tree
                     // containing x.

int find(int y);     //  searches for element y and returns the key in the root of the tree
                     //  containing y. Implements path compression on each find

int numberOfSets()   // returns the number of disjoint sets remaining
void printStats();   // see description below
void printSets();    // prints the array contents in the UnionFind data structure as one line.
                     // See the description below
```

The input will be as follows. The first line may contain a "d" command including an integer N. The value of N will give the number of elements in the set. The elements of the sets will be the integers $0, 1, 2, \cdots, N-1$. Then a series of commands will follow, one per line. Your program will output a summary of the results of each command, as described below. If the data file doesn't begin with a "d" command it will begin with an "m" command. See below.

```
Commands    (see the sample input and output that follows later)
d 10        // create a UnionFind data structure with elements 0,1,2...9

u 7 6       // Call union(7,6), output the root value and the size of the
            // resulting tree. See the sample output below

f 7         // Call find(7), output the root index. Keep track
            // of the total number of probes required in all find operations

p           // output the array elements in your UnionFind data structure,
            // space separated on one line

s           // output statistics as described below

m 3 30      // create a new UnionFind class with elements 0,1,..,(2^3)*(2^3)-1,
            // generate a Torus Maze as described below and print it

e           // last command of the data
```

Files that begin with an "m" command will include an "s" and an "e" command but will not include any "d", "u", or "f" commands. Files beginning with a "d" command will contain "u", "f", and "e" commands.

The "s" command

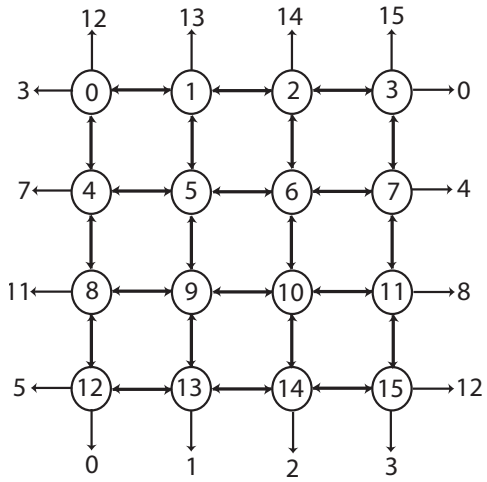
Output the following statistics:
Number of sets remaining = #### // use printf with %4d
Mean path length in find = ###.## // use printf with %6.2f

The mean path length should include all calls to `find()`, including those that are invoked by the `union()` method.
Each path length should be the number of UnionFind array cells probed on the search for the root node, i.e. those values prior to applying path compression. For any call to find the path length is at least 1.

The "m P Q" command

The "m P Q" command: Your program will generate a new P-dimensional Torus Maze. A Torus Maze is an undirected graph drawn on the surface of a torus (donut).

A P-dimensional Torus Maze has $D = 2^P \times 2^P$ nodes. P will be in the range 1 to 8 inclusive. Here are the nodes of a 2-dimensional Torus Maze, showing all legal edges. A subset of $D - 1$ of the legal edges will be included in a maze of D nodes:



Node 2 has potential neighbors 14, 6, 1 and 3. Node 0 has potential neighbors 12, 4, 3 and 1.

First create a new UnionFind data structure with elements $0, 1, 2, , (D-1)$ and create a data structure, `G`, for recording the edges of a sparse, weighted, undirected graph with vertices $0, 1, 2, , (D-1)$.

Start with the vertices all disconnected and generate random legal edges. For example, (3,6) would not be a valid edge in a $P = 2, D = 16$ maze. For each potential edge `(u,v)` generated, make calls to the `find()` method to see if `u` and `v` are members of the same set. If not, call `union(u,v)` and add the edge `(u,v)` to `G` with a random integer edge weight between 1 and Q inclusive.

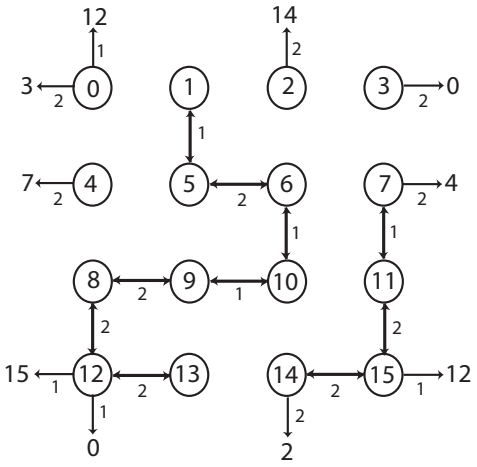
Stop the process when there is exactly one set remaining in the UnionFind data structure. At that point the graph will be connected, weighted, undirected and acyclic.

Output the Torus Maze as follows ($P = 2, Q = 2$ here):

```
2 3 12 2 1       // 0 is connected to 3 and 12 with weights 2 and 1 respectively
1 5 1            // 1 is connected to 5 with weight 1
1 14 2           // 2 is connected to 14 with weight 2
0                // 3 is not connected to any node with index > 3
1 7 2            // 4
1 6 2            // 5
1 10 1           // 6
1 11 1           // 7
2 9 12 2 2       // 8 is connected to 9 and 12 with weights 2 and 2 respectively
1 10 1           // 9
0                // 10
1 15 2           // 11
2 13 15 2 1      // 12 is connected to 13 and 15 with weights 2 and 1 respectively
0                // 13
1 15 2           // 14
0                // 15
```

There are D lines of output. Line i begins an integer Z_i giving the number of neighbors of node i that have higher ID numbers than i. Following Z_i is a list of the immediate connected neighbors of i that have IDs greater than i. This list is given in increasing order of ID number. On the same line follows a list of the Zi edge-weights for those edges. Delay generation of the edge weights until printing them.

Here is a sketch of the maze corresponding to the output above.



The "p" command:

This command will only be used for small mazes with no more than 32 nodes. Output the values in your array in the UnionFind data structure, space separated, on one line. This will be helpful for debugging and testing.

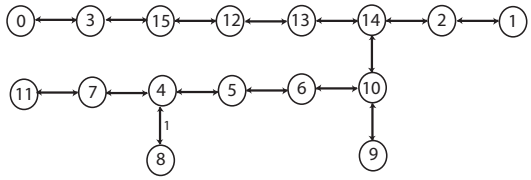
```
Sample Input 1
d 10
u 7 6
u 3 1
u 0 2
u 0 2
u 1 4
u 5 6
u 8 9
u 2 3
u 6 8
f 7
f 5
f 2
f 6
f 1
f 5
f 9
f 0
u 1 9
f 8
f 9
f 7
p
s
e
```

```
Sample Output 1
  7 2
  3 2
  0 2
  0 2
  3 3
  7 3
  8 2
  3 5
  7 5
  7
  7
  3
  7
  3
  7
  7
  3
  3 10
  3
  3
  3
  3 3 3 -10 3 7 7 3 3 3
  Number of sets remaining =    1
  Mean path length in find =   1.68
```

```
Sample Input 2
m 2 2
s
e
```

```
Sample Output 2
  1 3 1
  1 2 2
  1 14 2
  1 15 2
  3 5 7 8 1 1 2
  1 6 2
  1 10 1
  1 11 1
  0
  1 10 2
  1 14 2
  0
  2 13 15 2 2
  1 14 1
  0
  0
  Number of sets remaining =    1
  Mean path length in find =   1.67
```

Here is the maze produced above:



Consider the following data file and the results

Input Data File	What happens in find() etc.
d 10	
u 1 2	find(1), path length = 1
	find(2), path length = 1
u 3 4	find(3), path length = 1
	find(4), path length = 1
u 5 6	find(5), path length = 1
	find(6), path length = 1
u 7 8	find(7), path length = 1
	find(8), path length = 1
u 2 8	find(2), path length = 2
	find(8), path length = 2
u 6 1	find(6), path length = 2
	find(1), path length = 1
p	p command: -1 -6 1 -2 3 1 5 1 7 -1
f 3	find(3), path length = 1
f 6	find(6), path length = 3
f 8	find(8), path length = 3
f 2	find(2), path length = 2
f 1	find(1), path length = 1
f 6	find(6), path length = 2
f 8	find(8), path length = 2
p	p command: -1 -6 1 -2 3 1 1 1 1 -1
s	total calls to find = 19, total path length = 29
e	Number of disjoint sets remaining = 4
	Mean path length of all find operations = 1.53

RULES FOR PROGRAMMING AND SUBMISSION:

(1) Your submitted code must be entirely your own. If you do copy code from a text or the web, cite the copied section with a comment. Points could be lost, depending on what is copied.
(2) Write your program as one source file and remove the "package" construct from your Java source before submitting.
(3) Name your source file as $N_1N_2F_1F_2$p3.java where your given name begins with the characters N_1N_2 and your family name begins with the characters F_1F_2. For example my name is Ivor Page, so my source file will be called IVPAp3.java.
(4) **Do not include your name anywhere in your project.**
(5) Your program's output must exactly match the format of the **Sample Output** above.
(6) Do not use any Java Collection Classes except the Strings, arrays and ArrayLists.
(7) You program must read from System.in and output to System.out.
(8) Use good style and layout and comment your code well.
(9) Use the test files provided on the eLearning webpage for this class to test your program. Sample input and corresponding output files will be given.
(10) Submit your ONE source code file to the eLearning Assignment Dropbox for this project. Don't submit a compressed file. Don't submit a .class file.
(11) **There will be a 1% penalty for each minute of lateness, up to 60 minutes. After 60 minutes of lateness a grade of zero will be recorded.**