Seth Burkey, Manuel Gaddala

I. Requirements: Restate the problem specification and any detailed requirements in your own words.

Galactic Breakup: Given the dimensions of a galaxy and the group of dominions that secede each month return the number of total months that the loyal dominions in the galaxy are disconnected.

II. Design: How did you attack the problem? What choices did you make in your design, and why? Show class diagrams for more complex designs.

We created the data structure with all the disjoint set logic in the sub methods before doing any of the galactic breakup logic.

III. Security Analysis: State the potential security vulnerabilities of your design. How could these vulnerabilities be exploited by an adversary? What would be the impact if the vulnerability was exploited?

I am not aware of any potential security vulnerabilities in our design.

IV. Implementation: Outline any interesting implementation details in your solution.

We put all the disjoint sets logic in the node submethods and passed the node in the "this" parameter.

V. Testing: Explain how you tested your program, enumerating the tests if possible. Explain why your test set was sufficient to believe that the software is working properly, i.e., what were the range of errors for which you were testing.

Gradel

VI. Summary/Conclusion: Present your results. Did it work properly? Are there any limitations? NOTE: If it is an analysis-type project, this section may be significantly longer than for a simple implementation-type project.

The results were correct, and the code compiled correctly.

VII. Lessons Learned: List any lessons learned. For example, what might you have done differently if you were going to solve this problem again?

Our code compiled and ran properly, and produced the correct output.

Runtime Analysis:

We can ignore the find set because it is O(1) and connectAdjacencies because it is O(6)
It is O(nmk) because the double for loop iterates through the galaxy until all dominions have seceded. This setup is also O(nmk) and since O(nmk + nmk) = O(nmk) the runtime of our galacticbreakup problem is O(nmk).

Start of Main.java

```
package galacticbreakup;

import java.util.Scanner;
import java.util.ArrayList;

/**
* Given the dimensions of a galaxy and the group of dominions that secede
* each month return the number of total months that the loyal dominions in
* the galaxy are disconnected.
*/
public class Main {
    /**
     * @param args
     */
    public static void main(String[] args) {

        try (Scanner in = new Scanner(System.in)) {
            //initialize variables from standard in.
            int n = in.nextInt();
            int m = in.nextInt();
            int k = in.nextInt();
            int l = in.nextInt();
            in.nextLine();
            //initialize the 3d array of Nodes and
            Node[][][] galaxy = new Node[k][m][n];
            int sets = 0; //keeping track of connectedness
            int monthsDisconnected = 0;
            //Temp array to store the monarchies from the input
            ArrayList<int[]> monarchy = new ArrayList<>();

            for(int i = 0; i < l; ++i){
                //get monarchy (monarchy = set of dominions that secede in
the same month)
                String line = in.nextLine();
                //split dominions into indices in a string array
                String[] tokens = line.split("\\s+");
                int[] dominions = new int[tokens.length-1];
                //convert the string array to an integer array
```

```
                for (int x = 0; x < tokens.length - 1; x++) {
                    dominions[x] = Integer.parseInt(tokens[x+1]);
                }
                //add the monarchy to the array of monarchies.
                monarchy.add(dominions);
            }
            //for each monarchy
            for(int i = monarchy.size()-1; i >= 0; --i){
                for(int dominion : monarchy.get(i)){
                    //convert the dominion value
                    //into indices in the 3D array
                    int h = (int) Math.floor((double) dominion/(n*m));
                    int d = (int) Math.floor((double) (dominion%(n*m))/n);
                    int w = dominion%n;
                    //initialize the array of adjacent nodes
                    Node[] adjacencies = new Node[6];
                    galaxy[h][d][w] = new Node(); //makeSet
                    ++sets;//increment numOfSets
                    //fill adjacencies array if the adjacency exists
                    if(h < (k - 1)){
                        adjacencies[0] = galaxy[h+1][d][w];
                    }
                    if(h > 0){
                        adjacencies[1] = galaxy[h-1][d][w];
                    }
                    if(d < (m - 1)){
                        adjacencies[2] = galaxy[h][d+1][w];
                    }
                    if(d > 0){
                        adjacencies[3] = galaxy[h][d-1][w];
                    }
                    if(w < (n - 1)){
                        adjacencies[4] = galaxy[h][d][w+1];
                    }
                    if(w > 0){
                        adjacencies[5] = galaxy[h][d][w-1];
                    }
                    //union adjacencies
                    int numOfUnions =
galaxy[h][d][w].connectAdjacencies(adjacencies);
```

```
                    //subtract the number of unions from the number of
sets
                    sets -= numOfUnions;
                }
                //Adjudicate
                if(sets > 1){
                    ++monthsDisconnected;
                }
            }
            //print result
            System.out.println(monthsDisconnected);
        }
    }
}
```

End of Main.java

Start of Node.java

```
package galacticbreakup;

public class Node {
    Node parent;
    int rank;

    /**
     * makeSet / constructor
     * set parent to itself and set rank to 0
     */
    Node(){
        this.parent = this;
        this.rank = 0;
    }

    /**
     * @return (representative) Node
     * this method also sets the parent pointer
```

```
 * of each node between this and the
 * representative to the representative
 */
public Node findSet(){
    if(this != this.parent){
        this.rank = 0;
        this.parent = this.parent.findSet();
    }
    return this.parent;
}

/**
 * @param n
 * unconditional union by rank
 */
public void link(Node n){
    if(this.rank > n.rank){
        n.parent = this;
    }
    else{
        this.parent = n;
        if(this.rank == n.rank){
            this.rank++;
        }
    }
}

/**
 * @param n
 * union of the representatives by rank
 */
public void union(Node n){
    this.findSet().link(n.findSet());
}

/**
 * @param n
 * @return boolean
 * true if different component
 * false if same component
```

```
     */
    public boolean differentComponent(Node n){
        return this.findSet() != n.findSet();
    }

    /**
     * @param n
     * @return (numOfUnions) int
     * for each adjacency, union by rank
     * if adjacency is in a different component
     * and return the number of unions by rank.
     */
    public int connectAdjacencies(Node[] n){
        int numOfUnions = 0;
        for(Node node : n){
            if(node != null){
                if(this.differentComponent(node)){
                    this.union(node);
                    ++numOfUnions;
                }
            }
        }
        return numOfUnions;
    }
}
```

End of Node.java