CS 2420 Program 6 – 20 points  
Fall 2019 Union Find

**Part 1: Union Find**

Write the code to perform union find (using path compression) on a set of 121 (or more) elements. Use a smart union (by size). Create a series of carefully constructed tests so that you can verify union/find is working properly and that path compression works. Print out the contents of your array.

Everyone’s tests will be different. You will be graded on how well your tests demonstrate proper functioning. This ability to test your code (and dig into the details) is critical.

The Union/Find you create should work for any problem with integer items, and not be coded to only work for this assignment.

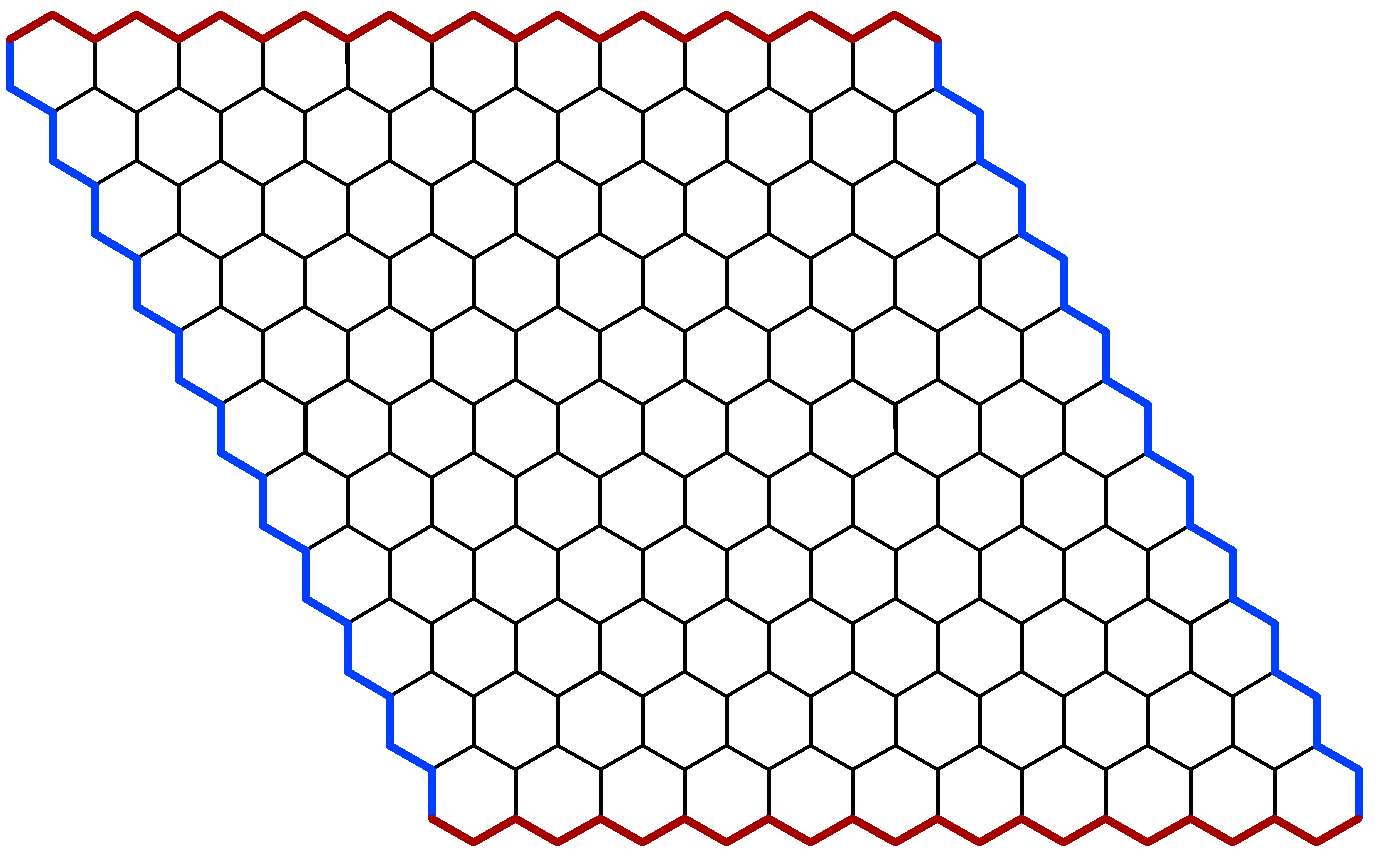
**Part 2: Hex** The game of Hex is played on a grid of hexagons. Hex is a strategy board game for two players played on a grid, theoretically of any size and several possible shapes, but traditionally as an 11×11 rhombus. Players (red and blue) alternate placing markers or stones on unoccupied spaces in an attempt to link their opposite sides of the board in an unbroken chain. In the figure below, the blue player is attempting to link the two blue edges, while the red player is attempting to link the red edges. Write the code to detect when red or blue has won the game. Use the union-find data structure. We will assume that the hexagons are numbered by rows, with the first row being 1-11, the next row being 12-22 and so on. With this numbering system, the neighbors of a non-edge node x are items x-1, x+1,x-11, x-10, x+10, x+11,

34 35 36 37 38 39 40 41 42 43 44

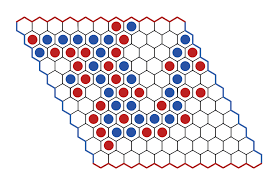
23 24 25 26 27 28 29 30 31 32 33

12 13 14 15 16 17 18 19 20 21 22

1 2 3 4 5 6 7 8 9 10 11



1. Keep track of which cells have been selected by each player.
2. Use the smart union find (with path compression) to keep track of the set of cells that are connected.
3. You will need a plan to decide how you can tell if the edges are linked. Our goal with union/find is to have an almost constant time operation. Thus, anything you add has to maintain this goal.
   1. One way to do this would be to have extra elements in the union/find associated with the TOP, BOTTOM, LEFT, and RIGHT. For the red player, when any item on the top row is selected, union it with item TOP. When any item on the bottom row is selected, union it with item BOTTOM. The game is won for the red player when find(TOP)=find(BOTTOM).



* 1. Do not check to see if the boundaries are reached by seeing if any item in row 1 is in the same group as any item in row 11 as this would be very slow.

Input:

The input is a series of moves indicated by the board location selected. Assume that the blue player always goes first. Indicate who wins and how many moves were made. If a player attempts to select a cell that has already been played, flag this as an error. Test your code with the supplied input files.

**Hint:** The awkward part is deciding who the neighbors of a node are. If a cell is in the center, the neighbors are found by adding the following offsets to item: int[] inc= {-11, -10, -1, 1, 10, 11}; These offsets correspond to neighbors in the following directions: NW,NE,W,E,SW,SE.

The tricky part is that the corners only have neighbors in two of the six direction. All other edges have neighbors in four directions, but each edge is different.

I ended up creating a getNeighbors method which returns only the legal neighbor increments.

**private int[] g**etNeighbors( **int** item)

Output from this routine may look like:

Neighbors of 83 [-11, -10, -1, 1, 10, 11]

Neighbors of 121 [-11, -1]

Neighbors of 62 [-11, -10, -1, 1, 10, 11]

Neighbors of 84 [-11, -10, -1, 1, 10, 11]

Neighbors of 112 [-11, -10, -1, 1]

I wrote another routine which told me if the player was at an edge important to them (TOP,BOTTOM,LEFT,RIGHT).

isEdge(83,RED) -> no

isEdge(121,BLUE) ->RIGHT

isEdge(62,RED) -> no

isEdge(84,BLUE) -> no

isEdge(112,RED) -> BOTTOM