**Part B Connect Coins**

In part B, you are going to use the data structure you have developed in part A to solve an interesting computational problem efficiently in a simple game. The game involves connecting gold coins in a 2-Dimensional space.

**Connect Coins Problem**

What's better than gold coins? More gold coins! In your game, a number of gold coins are placed on a 2-D space. The players can place a new gold coin by specifying a series of 2-D coordinates. We say that two coins are connected if the coins are next to each other in one of the 4 directions: left, right, up or down. Consider a particular step in that game, when a player wants to place a new coin. We are interested in finding out where to place the new coin so that the resulting connected coins are as many as possible.

**2-D Space and Coins Representation**

We represent the 2-D space as a 2-D boolean array of true (T) and false (F) values called **boolean[ ][ ] ccMatrix.**

A **T** in a coordinate indicates that there is a coin at that position in the 2-D space, while an **F** indicates an empty space.

The location of the new coin that would maximally connect the coins is specified by a 2-element integer array **int[ ]** representing the coordinates in [row, column] format.

The number of newly connected coins will be returned as an **int**.

**Part B: Connect Coins Task**

In part B, your task is to complete a skeleton code of the ConnectCoins class in order to figure out where to place a coin to maximally connect them, and how many coins can be maximally connected.

You will have to implement the following methods to complete the class:

1. **public ConnectCoins(boolean[ ][ ] ccMatrix)**. Each ConnectCoins instance is bound to a single 2-D space, which is passed in through its constructor. You may assume this space is valid, i.e., there is at least one empty coordinate to place a new coin.

2. **public int[ ] placeMaxConnCoins( )**. The method returns a 2-element integer array that represents the coordinate in [row, column] so that a coin that is placed in that coordinate will give the maximum number of newly connected coins. If there are multiple possible such placements, return the left-and-topmost coordinate.

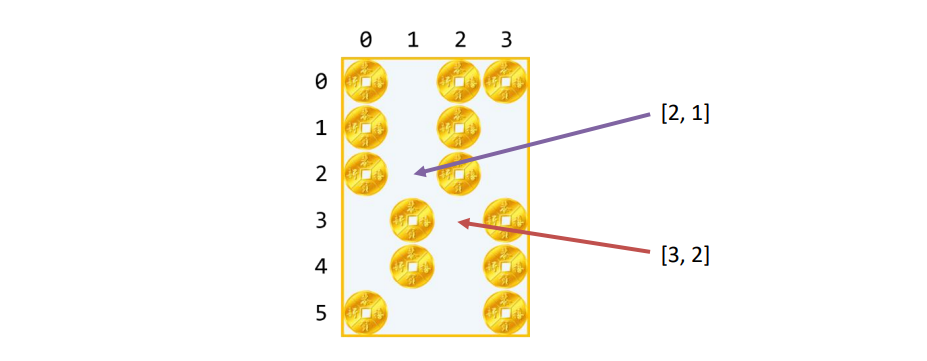
3. **public int maxConnCoins( ).** The method returns the maximum number of newly connected coins after placing a new coin.

**Test Case 1**

Input ccMatrix = [[T, F, T, T], [T, F, T, F], [T, F, T, F], [F, T, F, T], [F, T, F, T], [T, F, F, T]]

Output of placeMaxConnCoins = [2, 1]

Output of maxConnCoins = 10



**Explanation:**

Placing a coin at coordinate [2, 1] will connect 10 coins, which is the maximum number of newly connected coins in this instance of the game.

Placing a coin at coordinate [3, 2] will also connect 10 coins, but the left and topmost coordinate with the same score must be returned instead.

**Additional Notes Here are some additional notes:**

1. The correct implementation of the Union Find data structure will be provided in the automatic grader system for you readily to use.

2. You have to use the Union Find data structure in your implementation and computation. Failing to do so will result in 0 marks.

3. The number of rows and columns in the 2-D space will be in the range [1, 1000]. In particular, it means that the smallest valid array is 1-by-1.

**Advice**

The following advice may be found useful in implementing Part B:

1. Use the Automated Regression Unit Testing with your correct Weighted Union Find (without Path Compression), that is guaranteed correct, if you have not completely finished Part A.

2. Add more test cases, and create a good suite of test cases and practice the Partitioning/Boundary, Black-box/White-box, and Coverage testing.

3. Debug with the help of Java Visualizer plugin in IntelliJ IDEA.

4. You may define your own private helper methods. Include them in each of your submissions.

5. Do not define your own instance variables. They are not going to be used in the hidden test cases and may cause unpredictable errors in the grading system.