CPT204 Online 2022
Coursework 3 Task Sheet

Overview

Coursework 3 (CW3), the final assessment component this semester, consists of 3 parts: Part A and B . It contributes to 80% of your final marks.

In **Part A (100 marks)**, you will implement a data structure called Union Find with Path Compression. You will then use this data structure to solve a problem called Connect Coins in **Part B (100 marks)**.

Submit all your answers for parts A, B to Learning Mall for grading on the Submission Day.

Timeline

Week 12, Wednesday, May 11, 2022	CW3 Part A, B released (Task Sheet, Skeleton Codes, Partial Test Cases)
Week 14, Thursday, May 26, 2022	CW3: Submission Day
11.00 CST	CW3 Part A, B online submission open
11.15 CST	CW3 online submission open
12:15 CST	CW3 Part A, B online submission closed

Outline

The rest of the task sheet will describe all the two parts and the Submission Day in detail.

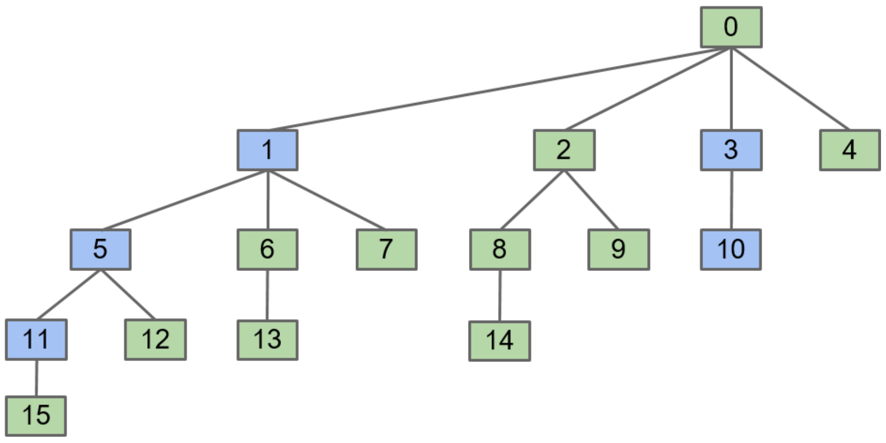
Coursework 3 – Part A
Union Find with Path Compression

Recall that in Lecture 9 and Lab 9, we have constructed a Union Find data structure called Weighted Quick Union, and achieved O(log N) time for `union`, `find` and `isSameGroup` operations. It turns out that we can do even better, to get almost constant time for both operations!

In part A, you are to improve your Lab 9: Weighted Quick Union data structure with Path Compression.

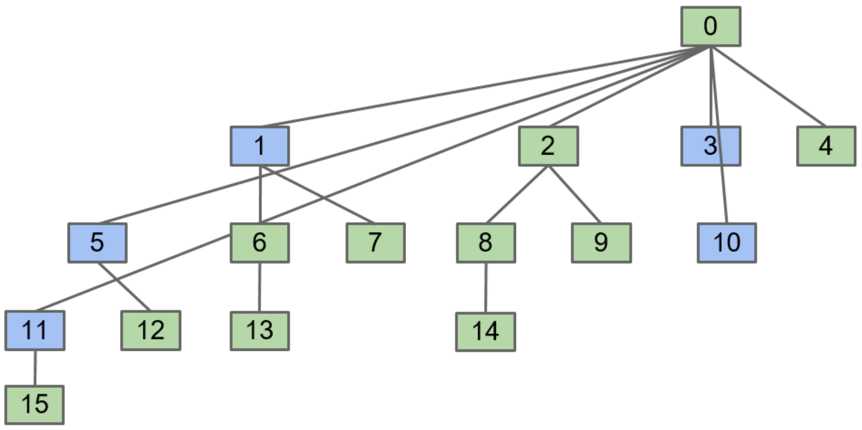
Idea of Path Compression

Consider the connectivity tree below. It is one of the possible valid trees of 16 items in a Weighted Quick Union data structure, resulting from some series of operations. Now imagine you call `isSameGroup(10, 11)`. That will involve finding the root of `10` and `11`, and will be preceded by finding the parent of the elements in blue.



The key idea is this: since we have found the root of the elements in blue while climbing the tree, whose root is 0, we want to change and set the parents of those blue elements directly to the root.

The result is the following tree:



Notice that this changes nothing about which group each element belongs to. They are still in the tree where 0 is the root.

The additional cost of this path compression operation to `isSameGroup` is still in the same order of growth, but now the future operations that require finding the root will be faster! We are going to use the same path compression idea on the other operations as well.

Note that this path compression results in amortized running time on N operations of `union/isSameGroup/size` of O(α(N)), where α is the inverse Ackermann function, of which the value, for practical purposes, is always less than 5.

Part A: Weighted Quick Union with Path Compression

In part A, your task is to complete an implementation of the Union Find data structure using Weighted Quick Union of Lab 9 **with Path Compression**.

You will have to implement the following methods to complete the data structure:

1. `public` **`UnionFind`**`(int N)` creates a Union Find data structure with `N` elements: `0` through `N-1`. Initially, each element is in its own group.

2. `public void` **`validate`**`(int p)` checks whether p is a valid element. It throws an `IllegalArgumentException` if p is not a valid index.

3. `public int` **`sizeOf`**`(int p)` returns the size of the group the element p belongs to.

4. `public int` **`find`**`(int p)` returns the group identity number which is the root of the tree element p belongs to. Assume p is a valid element. The path compression operation is applied in this method to reduce the finding root's running time.

   Note that now, the given method `public boolean` **`isSameGroup`**`(int p, int q)` is then implemented by simply calling `validate` on p and q, and then checking whether `find(p)` is the same as `find(q)`.

5. `public void` **`union`**`(int p, int q)` connects two elements p and q together, by combining the groups containing them, connecting the root of the smaller size tree to the root of the larger size tree. If the sizes of the trees are equal, ***break the tie by connecting p's root to q's root***. It throws an IllegalArgumentException if p or q is not a valid index.

The total marks of all the implementations in part A is **100 points**, for passing all the test cases.

Advice

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

1. Use the same Automated Regression Unit Testing and Integration Testing strategy that you have been using in Lab 9. Note that with the use of the Path Compression strategy, the output may be different from the result in Lab 9.
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.

Coursework 3 – 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.

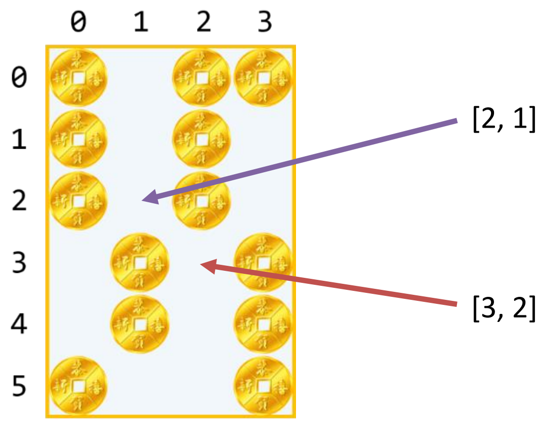
2. `public int[]` **`placeMaxConnCoins`**`()`. The method returns a 2-element integer array that represents the coordinate in [row, column] so that a coin that is placed in that coordinate will give the maximum number of newly connected coins. If there are multiple possible such placements, return the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

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

The total marks of all the implementations in part B is **100 points**, for passing all the test cases.

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 **upper-leftmost** 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) of Lab 9, 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.