Mastery III — Data Struct. & Algo. (T. III/17–18)

Name: ______________________	
ID: ________________________	

Directions:

- This examination has **two** parts, consisting of 4 problems. You have a total of 230 minutes (i.e., 3 hours and 50 minutes). **SPECIAL INSTRUCTIONS:**
  - Part 1: Problems 1 – 2 must be uploaded to Canvas within the first 2 hours while you're in the exam room. You will upload this to "Mastery 3 – Part 1" on Canvas.
  - Part 2: Problems 3 – 4 can be submitted until the end of the examination. You can be anywhere. You will upload this to "Mastery 3 – Part 2" on Canvas.
- The maximum possible points is 40, but we'll grade out of `26` points. Anything above that is extra credit.
- No collaboration of any kind whatsoever is permitted during the exam.
- **WHAT IS PERMITTED:**
  - Reading the official Java documentation
  - Accessing Canvas for submission.
- **WHAT IS *NOT* PERMITTED:**
  - Browsing (online) tutorials or reading stack overflow threads.
  - Accessing previously-written code on your own machine.
  - Communicating with other people or using any other aid.
- For each problem, the entirety of you solution must live in one file, named according to the instructions in this handout. When grading a problem, the script will only compile that one file for the problem. Do *not* modify other files.
- We're providing a starter package, which you can download at

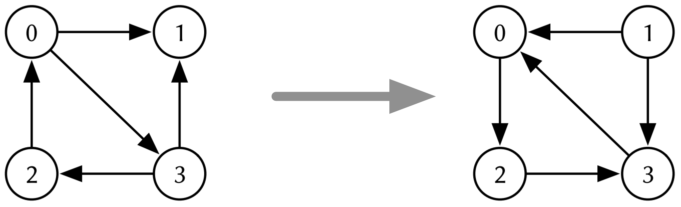
  `https://cs.muic.mahidol.ac.th/courses/ds/tetris-battle.zip`

  The password is "meeehh". When you unpack the package, you'll see one file for each problem.
- To submit your work for each part, create a single zip file containing the relevant Java files and upload that to Canvas (as described above).

1	2	3	4	Σ

Problem 1: Graph Reversal (10 points)

Given a **directed** graph $G = (V, A)$, the reverse of G, written G^{R}, is a graph on the same vertices but with the direction on each edge reversed. The figure below shows this transformation on an example graph.



This is a trivial task if the graph is represented as an edge list. For example, if the edges are `[(0, 1), (2, 0), (3, 2), (0, 3), (3, 1)]`, then the edges in the reversed graph will be `[(1, 0), (0, 2), (2, 3), (3, 0), (1, 3)]`.

In this problem, however, the graph is given as an *adjacency table*, i.e., `Map<Integer, Set<Integer>>`, where the outer map maps each vertex (an integer) to the set of its out-neighbors (vertices that it is pointing to). Inside the class `Reversal`, you are to write a static method

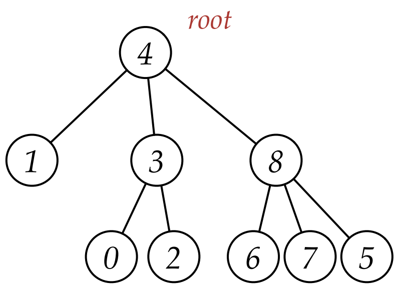
```
public static Map<Integer, Set<Integer>> reverseGraph(Map<Integer, Set<Integer>> G)
```

that takes as input a directed graph G as described and returns the reverse of G in the same format. Note: the original graph G should remain intact.

Performance Expectations: The largest test case we'll use contains up to $500,000$ edges. For every test case, your code should finish within 3 seconds to receive full credit. The `Map` here is a `HashMap` and the `Set` here is a `HashSet`. You should aim for an $O(m)$-time solution, where m is the total number of edges. Partial credit will be given to slower solutions.

Problem 2: Siblings (10 points)

A tree can naturally be organized into levels as follows: the root belongs to level 0, the children of the root belong to level 1, the children of the level-1 nodes belong to level 2, and so on. In this way, every node belongs to exactly one level. The figure below and an accompanying table show an example of a tree and the nodes in different levels.



Level	*Vertex*
0	4
1	1, 3, 8
2	0, 2, 6, 7, 5

Definition: Two nodes u and v are *siblings* if and only if they are in the same level. For example, 2 and 5 are siblings because they're both in level 2; however, 1 and 7 are not.

You'll write a class `Siblings` implementing a data structure that efficiently answers whether a pair of nodes u and v are siblings. In writing this class, you will implement two functions (write as many helpers as you need):

- The constructor for the class takes a child-to-parent map which stores the tree as a `HashMap<Integer, Integer>`. See the starter code for details. The goal of the constructor is to process the input sufficiently so that the following query can be answered quickly.

- A class method **public boolean** `isSibling(`**int** `u,` **int** `v)` that returns a boolean indicating whether `u` and `v` are siblings.

Notice that since you're implementing a class, you are free to keep data inside your class as class variables. Here's an example of this class will be used/tested:

```
// tree is a HashMap representing the input tree
Siblings sbl = new Siblings(tree);
sbl.isSibling(1, 100);
sbl.isSibling(7, 31);
// ... many other isSibling queries.
```

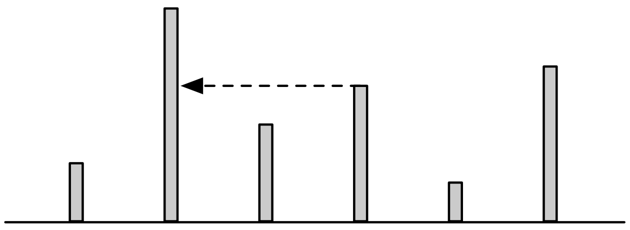
Performance Expectations: The constructor shouldn't take longer than $O(n)$ time, where n is the number of nodes in the tree. Moreover, the method `isSibling` should take at most $O(\log n)$ time—though, an $O(1)$-time implementation is much preferred.

Problem 3: Look to the Left (10 points)

There are n observation towers numbered $0, 1, \ldots, n-1$ from left to right. The towers are identically-shaped only differing in heights and are located on a straight-line road. For $i = 0, 1, \ldots, n-1$, tower i is located i meters from the left end and has height h_i.

Ply has a challenge for Gift again. He wonders, if he goes to the rooftop of tower i and looks to the left with his eye level at precisely the height of the building, which building would he see (from his eye level)? Let `lsof(i)` be the height of the tower that he sees if he looks to the left from tower i. More precisely, if a ray emanating from $(x = i, y = h_i)$ going left hits the tower j, then `lsof(i)` is h_j. If there is no tower at that level, then `lsof` should return **null**

Suppose the heights (kept as an array) are `[3, 11, 5, 7, 2, 8]`, which look as shown in the figure on right. If we ask for `lsof(3)`, that is looking left from a height of 7, then we'll encounter tower 1, which has height 11, so `lsof(3)==11`. Notice that `lsof(1)==` **null** as there is nothing to the left of tower 0 at that height.



Inside a class called `LeftSight`, you will implement a function

public static `Integer[] lookLeft(`**int**`[] hs)`

which takes in an array of heights and returns an array of `Integers` of the same length, where the i-th index contains the value of `lsof(i)`. This means for the above example, it would return `[`**null**`,` **null**`, 11, 11, 7, 11]`.

Performance Expectations: We will test your program with an input of up to 5,000,000 elements. It should return the answer within 2 seconds. To receive full credit, it should take no more than $O(n \log n)$ time—though, again, an $O(n)$-time solution is expected.

Problem 4: The Perfect Hiding Spot (10 points)

The CS student committee for recreation and well-being is renting a huge castle for a week-long party this summer. The entrance opens into Chamber 1. In this chamber, they're planning to set up several large TV monitors so they can play Puyo Puyo Tetris without seeing daylight. Not everyone is on board with this idea, though. Nice has been persuaded to join the party, but she wants to read by herself, away from the gaming crowd.

Your task is to help Nice find the perfect hiding spot. You will be presented with a map of the castle. The chambers (vertices) are numbered 1 through n. You'll also be given a list of one-directional passages from one chamber to another (directed edges). Every passage is labeled with its distance. Importantly, each chamber has exactly one passage that opens into it. More precisely, the input is a directed tree where every vertex, except for vertex 1, has precisely one incoming edge. Note: vertex 1 has no incoming edge.

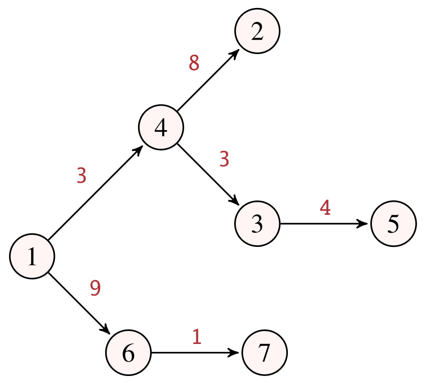
Inside the class `PerfectHiding`, you are to implement a static method

```
public static int bestSpotDistance(int n, List<WeightedEdge> passages)
```

where n is the number of chambers and `passages` is a list of directed edges. If `e` is a `WeightedEdge` in this list, then there is a passage from Chamber `e.first` to Chamber `e.second` of length `e.cost` meters. Notice that because this structure is a tree, the length of the list `passages` is always $n - 1$.

Your static method will compute the distance to the chamber (vertex) farthest away from chamber number 1. If it turns out that you cannot reach any chamber other than 1, the answer will be 0.

Example:



This graph has $n = 7$ nodes and the edges are:

```
(1, 4, 3)   (4, 2, 8)   (4, 3, 3)
(3, 5, 4)   (1, 6, 9)   (6, 7, 1)
```

Here, the notation (u, v, c) denotes a directed edge from u to v with a weight of c meters. Calling `bestSpotDistance` on this input will result in the number 11 because Chamber 2 is the farthest from 1 via $1 \to 4 \to 2$, totaling $3 + 8$ meters.

Performance Expectation: We'll test your program with n up to $100,000$. On this size of input, your program must finish within 3 seconds to receive full credit. The cost of an edge will be a number between 1 and $1,000$ (inclusive).