Mastery II — Data Structures (T. I/21–22)

Directions:

- You have 4 hours to complete the examination. Each problem has a stated performance goal. Meeting that with a correct answer will earn you full credit. Expect partial credits for correct code that doesn't run as fast.
- The maximum possible score is 40, but we'll grade it out of $T \le 30$. The exact T will be determined later. But this means fully solving 3 problems guarantees 100% or more on this exam. Anything above T is extra credit.
- **WHAT IS PERMITTED:** The exam is open- book, notes, Internet. However, you must *not* get help on Stackoverflow, Chegg, Discord, LINE etc. or read/use direct solutions to any exam problem from anywhere on the Internet
- **WHAT IS *NOT* PERMITTED:** Communication/collaboration of any kind.
- We're providing a starter package. The fact that you're reading this PDF means you have successfully downloaded the starter pack. For ease, we're putting all the starter files in the same folder: `src/main/java`.
  Also, to save you some typing, we're supplying a few tests in the starter package, some as JUnit tests and also as simple prints. These tests are decent but aren't comprehensive. Passing these tests doesn't mean you will pass our further testing. Failing them, however, means you will not pass the real one.

Handing In: To hand in, zip ***only*** the following files as `mastery2.zip` and upload your zip file to Canvas:

`WeighMyTree.java` `Xnumber.java` `CSClans.java` `SmallVil.java`

This means your code should only be in these files and nowhere else.

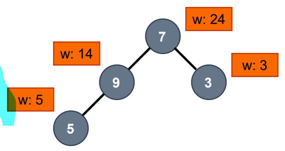
Problem 1: Weigh My Tree (10 points)

A complete binary tree (CBT) is a binary tree that is full at all levels except possibly at the bottom where the nodes are left-justified. Dr. Piti stores a CBT in an `int` array `a`, just like in a binary heap. But his CBT doesn't necessarily satisfy the heap-ordering invariant. Remember that in this representation,

- the root of the tree is at index 1; and
- for a node at index k, its left child is at index $2k$, and its right child is at index $2k + 1$.

The weight of a tree node v is the sum of the values at each of the nodes in the subtree of v, including v itself. By extension, the weight of a tree is obtained by adding up the weights of all the tree nodes. For example, the array on left corresponds to the CBT on right, where each tree node is annotated with its weight.

`a = [0, 7, 9, 3, 5]` $\Longrightarrow$



In this example, the weight of the node 9 is $9 + 5 = 14$, and the weight of the root is $7 + 9 + 3 + 5 = 24$. The weight of the tree is the sum of all the weights: $24 + 14 + 3 + 5 = 46$

Your Task: Inside the public class `WeighMyTree`, write a public method

```
public long weigh(int[] a) { ... }
```

that takes as input an array a representing a CBT and returns a `long` that is equal to the tree's weight.

Expectations: Promises, Constraints, and Grading

- $1 \le$ `a.length` $\le 50,000,000$. `a[0]` is 0, and the numbers in `a` are *not* necessarily distinct.
- For full-credit, your code must run in $O(n)$ time, where n = `a.length`. Your code must finish within 1 second on each of the test cases.
- You can write as many helper functions/inner-classes.

(*Hint:* There is a tree traversal order that is well-suited to this problem.)

Problem 2: XNumber (10 points)

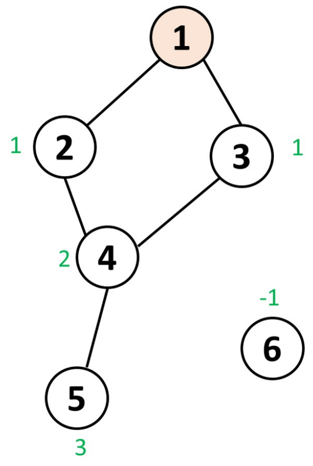
`XNumber` is a "knowing distance" between one student to another, where student A knows student B if they are registered in the same course. The base input is an array `roster` that shows student registration, like in the following example:

```
int[][] roster = {{1,2},{1,3},{1,5},
                  {2,5},{2,4},
                  {3,2},{3,3},{3,1},
                  {4,1},{4,4},{4,6},
                  {5,6},{5,7},
                  {6,9}};
```

In the `roster` array, each subarray $\{s, c\}$ represents the fact that a student id s registered for a course c, where $0 < s, c < n$. For instance, the array $\{\{1, 2\}, \{1, 3\}, \{1, 5\}\}$ means that student id 1 has registered for courses number $2, 3$, and 5.

From this, we are about to create a graph of students with an edge/link between a pair of students if they are registered in at least *ONE* same course. For example, $\{\{1, 2\}, \{3, 2\}, \{4, 4\}\}$ illustrates that the student id 1 has a link with student id 3 as they both registered to study the course number 2.

For a larger example, the input `roster` at the top yields the following graph (vertices are students):



Consider the above network: 6 vertices (n) and 5 edges (m) constructed from the given input. It is clearly seen that student id 1 registered for the same course as student id 2 and 3. The `XNumber` of student id 2 and 3 considered student id 1 as a reference is **1**. The `XNumber` of student id 4 considered student id 1 as a reference is **2**. Any students that cannot be reached from the reference student has an `XNumber` of -1. Your task is to complete the following class and methods.

Your Task: Inside the class `XNumber`, implement the following two methods:

- The method `public int getXnumber(int s, int d)`: the method take two parameter s and d, this will return `XNumber` of student d by considering student s as a reference.
- The method `public Set<Integer> getStdByCourse(int courseID)`: This method take courses id and return a set of student id for all students who registered on the course.

Expectations: Promises, Constraints, and Grading

- guaranteed that the maximum number of student will not exceed $3,000$.
- guaranteed that one course can registered up to a maximum of 30 students.
- Your code for `getXnumber(int s, int d)` must run in $O(n + m)$ time—and the other method, as well as your constructor, must not be excessively slow.
- To get a full score, your code must finish within 2 seconds on each of the test cases.

Problem 3: Cat Society Clans (10 points)

The cat society (CS) of Salaya wants you to implement a Java class to track its members. The public class will be called `CSClans`. By now, the expansive society has many members. For each member, your class will keep the member's evil aura value, which can be updated over time (see below). Additionally, the class keeps track of "who knows who," ultimately tracking groups/clans of cats. In particular, a *clan* is a group of one or more cats who can all reach one another through tailshaking (i.e., connected in the tailshaking graph).

Your class will support the following operations (a sample run appears on right):

- Initially no one belongs in this society.
- The method `public void set(String name, int evilAura)` either introduces a new cat or updates an existing cat's evil aura value. That is, if `.set` is called on a new name, it introduces a cat of that name with the specified evil aura value. However, if `.set` is called on an existing name, this method replaces the evil aura value of that cat by the new value.
- The method `public void tailShake(String catA, String catB)` informs your class of the fact that `catA` is bonding with `catB` (and vice versa). This happens through a complex tailshaking process (maybe like our handshaking) that we humans don't really (have to) understand. *Keep in mind:* it is entirely possible that tailshaking will happen between two cats who already belong to the same clan.
- The method `public Map<String, Double> report()` returns a `HashMap` containing all the clans. Each clan gives rise to an entry in the map, where
  - the key is the name of the cat in that clan that has the highest evil aura value (this is guaranteed to be unique); and
  - the value is the average evil aura value of that clan. Remember that for k numbers, the average is $\frac{1}{k}(x_1 + x_2 + x_3 + \cdots + x_k)$. The average only has to be correct up to ± 0.0001.

Example:

```java
CSClans cs = new CSClans();
cs.set("Beth", 8);
cs.set("Deb", 50);
cs.set("Jolie", 2);
cs.set("Alice", 23);
cs.tailShake("Jolie", "Deb");
cs.set("Jolie", 10);
cs.set("Vera", 4);
cs.set("Cathy", 21);
cs.tailShake("Cathy","Beth");
cs.tailShake("Beth","Vera");
cs.set("Alice", 21);
cs.report(); // => {Deb=30.0, Alice=21.0,
    Cathy=11.0}
```

Explanation: At the moment when `.report` is called, the tailshaking graph looks as follows (evil aura values in parens):

```
Vera (4)
  |                        Deb (50)
Cathy (21)  Alice (21)      |
  |                        Jolie (10)
Beth (8)
```

Hence, `Cathy` has the highest evil value in the {`Vera`, `Cathy`, `Beth`} clan—and the average is $(4+21+8)/3 = 11.0$. The entries for the other two clans are computed similarly.

Your Task: Implement a public class `CSClans` with the methods detailed above.

Expectations: Promises, Constraints, and Grading

- The total number of cats will not exceed 100,000, and the total number of `tailShake` calls will not exceed 20,000,000.
- `report()` can only be called once per class instance and is guaranteed to be the last thing in your class that is called.
- `tailShake` will only be called on names that have been introduced to the system already. However, the evil aura values may be subsequently updated.
- For full-credit, your code must run in $O((m + n) \log n)$ time or faster, where m is the total number of tailshakes, and n is the total number of cats. Concretely, your code must finish within 3 seconds on each of the test cases.
- You can write as many helper functions/inner-classes.

Problem 4: Small Village (10 points)

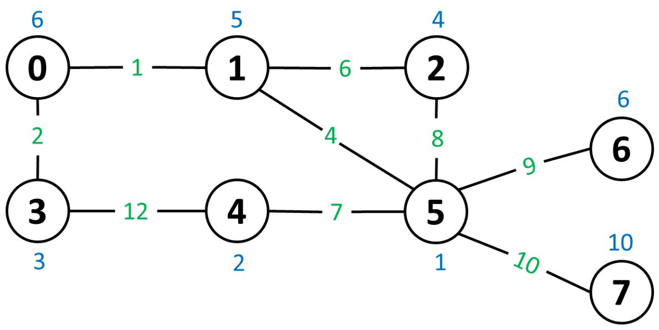
The map of a "small" village is represented as a graph. Each vertex is a house, each equipped with a popularity score. An edge between a pair of vertices (houses) comes with the distance between them. The graph will be presented to you as three arrays, as will be explained through the following example:

```
int[]    VCost = {6,5,4,3,2,1,6,10};
int[][]  edges = {{0,3},{0,1},{1,2},{1,5},{2,5},{5,4},{4,3},{5,6},{5,7}};
int[]    d     = {  2,    2,    6,    4,    8,    7,   12,    9,    10};
```

For a graph with n vertices and m edges, the vertices will be named $0, 1, 2, \ldots, n-1$.

- The array `VCost` has length n and stores the popularity scores, so `VCost[i]` is the score of vertex `i`.
- The array `edges` has length m and keeps all the edges. An entry `{x, y}` in the array indicates a bidirectional link between vertex `x` and vertex `y`. Thus, $0 \le$ `x, y` $< n$.
- The array `d` has the same length as `edges` and stores the edge distances. The edge represented by `edges[k]` has distance `d[k]`.

Hence, visually, the sample arrays above represent the graph below:



This example graph has $n = 8$ vertices and $m = 9$ edges. We are about to find the lowest total cost of edges that can connect all vertices from a given input. **There is a funky twist:** the cost of edge between vertices x and y is given by $cost(x, y) = d(x, y) + f(x, y) + f(y, x)$, where

$$f(i, j) = \min \{\texttt{VCost}[k] \mid k \in \text{neighbor of } j \text{ and } \texttt{VCost}[k] > \texttt{VCost}[i]\}$$
$$d(i, j) = \text{distance between vertex } i \text{ and } j.$$

In words, $f(i, j)$ is the smallest popularity score among the neighbors of j that have popularity scores strictly greater than the popularity score of i. For example, $f(1, 5)$ is 6 because $i = 1$ has popularity score 5; and the neighbors of $j = 5$ are $\{1, 2, 4, 6, 7\}$ but in terms of their popularity scores, only $\{6, 10\}$ are higher than 5 (the popularity of $i = 1$)—where 6 is the smallest. As an example, $cost(2, 6) = 4 + 6 + 4 = 14$. **Note that** if $f(i, j)$ is undefined, then define $f(i, j)$ to be 0.

Your Task: Inside the class `SmallVil`, write a public method

```
public static long findMinCost(int[] vcost, int[][] edges, int[] d)
```

Expectations: Promises, Constraints, and Grading

- We promise `n` $\le 30,000$ and `m` $\le 900,000$ The distance between vertex i and j is $1 \le d \le 1,000$
- The numbers in `VCost` will be between 0 and 100,000 (inclusive).
- For any pair of vertices, the input will have at most one edge between them.
- The initial graph is guaranteed to be connected.
- For full-credit, your code must run in $O(m \log n)$ time. Your code must finish within 3 seconds on each of the test cases.

(*Hint:* What kind of map/set is ordered?)