Mastery II — Data Struct. & OOP (T. I/19–20)

Name: ______________________________

ID: ________________________________

Directions:

- **WHAT IS PERMITTED:**
  - Reading the official Java documentation
  - Accessing Canvas for submission.
  - Browsing (online) tutorials or reading stack overflow threads.
- **WHAT IS *NOT* PERMITTED:**
  - Asking or attempting to get help from a person or AI either online or in person.
  - Communicating with other person or using any other aid.
- Your code must not be kept in any package—that is, all your classes must live in the default package.
- For each problem, we have specified the filename in which your solution must be. Everything pertaining to that problem must live in that file.
- To submit your work, zip the folders for both problems as one zip file called `mastery2.zip` and upload it to Canvas.

Problem 1: Nesting Cups (10 points)

You are writing a program to put together cups of different sizes. The cups have been measured using several sensors that can accurately determine the radius and color of a cup. For each cup, the measurement is reported as a string. However, there is a glitch in how the measurement is reported:

- if in the reported measurement, the color arrives after the radius, the reported radius has been mistakenly doubled (i.e., the reported value is 2x the true radius).
- otherwise, the reported values are accurate.

This means, for instance, a red cup with an actual radius of 5 units will either turn up "red 5" or "10 red".

Inside the `NestingCups` class, you'll write

```
public static String[] orderNestingCups(String[] measurements)
```

meeting the following spec: Given a list of reporting strings (per above), each describing a different cup, your function will report the colors in order of the (true) radii of the cups, from the smallest to the largest.

You should know that

- The true radius of each cup is an integer, and the radius, as reported, will always be an integer between 1 and $20,000,000$.
- A color is a single, non-empty word consisting of only lowercase letters in the English alphabet (i.e., `a-z`). Each color is a string of length at most 5.
- The true radii of the cups are all unique.
- The color component and the radius component are separated by a single space.
- The number of cups n is at most $100,000$.

Example: `orderNestingCups(["red 10", "10 blue", "green 7"])` should return `["blue", "green", "red"]`.

(*Hint:* Want to know if a character is a digit (0 - 9) or a letter? Check out `Character.is_____`. Use autocomplete in your IDE to find out.)

Performance Expectations: For full credit, your code should run in $O(n \log n)$ time. This means finishing in less than 3 seconds on the largest input.

Problem 2: Super Sweet (10 points)

Kao-too has a sweet tooth. He wants the sweetness everything he eats to be at least a certain value. To accomplish this, he repeatedly mixes two items with the least sweetness. The combined portion has a sweetness of

$$\text{sweetness} = (\text{sweetness of the least sweet item}) + 2 \times (\text{sweetness of the 2nd least sweet item}).$$

He repeats this procedure until everything in his collection has a sweetness value of at least k—or there are fewer than two items in the collection. If from the start, everything in the collection has a sweetness value of at least k, he will *not* perform this procedure at all .

You are given an initial collection of sweet items. You'll write a program to determine the average sweetness of the items in the final collection. More specifically, inside the class `Sweet`, write a method

```
public static double superSweet(long k, long[] sweetness)
```

that takes in $0 \leqslant k \leqslant 10^8$ and an array of sweetness values. This array contains at most 10^6 entries and each entry is a number between 1 and 10^6 (inclusive).

- If every item is at least as sweet as k when the process stops, the method will return the average sweetness of the items in his collection when this procedure stops. It only has to be correct up to ± 0.00001.
- Otherwise, it will return `Double.NaN`.

Example: Let's walk through `superSweet(7, [1, 12, 3, 9, 2, 10])`. At first, the least sweet item has a sweetness value 1, so he combines 1 and 2 together, resulting in a new portion with sweetness $1 + 2 \times 2 = 5$. After one application of the procedure, the collection becomes `[5, 12, 3, 9, 10]`. Still, the least sweet item is not sweet enough, so Kao-too combines 3 and 5 together, yielding $3 + 2 \times 5 = 13$. The collection then becomes `[13, 12, 9, 10]`.

At this point, every item in the collection is at least 7. We compute the average of these items, like so: $\frac{1}{4}(13 + 12 + 9 + 10) = 11.0$. Hence, the above call should return 11.0.

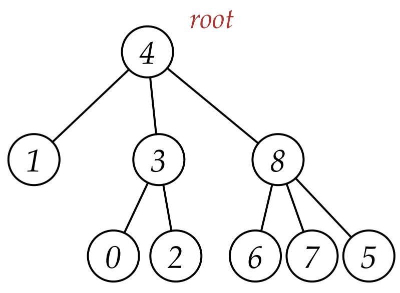
As another example, `superSweet(7, [7])` should return 7.0. Additionally, `superSweet(7, [1, 2])` will return `Double.NaN`.

Performance Expectations: On the largest test case, your method should return within 2 seconds. We expect the running time of your code to be at most $O(n \log n)$.

Hints: Be careful—if a and b are `int`s, then a/b will be an `int`. For example, `4/3` is 1 in Java.

Problem 3: 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>`. This means if `G` is such a `HashMap`, then `G.get(u)` returns the parent of `u`. The unique node whose parent is itself is the root of this tree.

  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.