Mastery II — Data Structures (T. I/22–23)

Directions:

- This mastery examination starts at 2pm on Friday December 2, 2022. You have until 5:50pm of the same day to complete the following *four* problems.
- **WHAT IS PERMITTED:** The exam is open- book, notes, Internet, Google, Stack Overflow, etc. The Internet can only be used in *read-only* mode.
- **WHAT IS *NOT* PERMITTED:** Communication/collaboration of any kind. Using answers from study-aid websites (e.g., Chegg). Obviously, asking a question online is strictly *not* allowed.
- We're providing a starter package. The fact that you're reading this PDF means you have successfully downloaded the starter pack.
- Your Java code must *not* be in any package. That is to say, your files must *not* contain the statement `package .....`

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

`HistoMap.java` `WeighMyTree.java` `MinOps.java` `CSClans.java`

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

Problem 1: HistoMap (10 points)

The *histogram* of an array `a` of **`long`**s is a `Map<Long, Integer>` that stores for each number present in `a`, how many times that number appears in `a`. More specifically, if `h` is the histogram of an array `a`, then for every number `x` in `a`, `h.get(x)` is the number of times `x` appears in `a`.

Your Task: Inside a public class `HistoMap`, write a public method

```
public Long fetchRank(TreeMap<Long, Integer> h, long k) { ... }
```

that takes as input a histogram obtained from an underlying array `a` (*not* given to us) and a number $k \geq 0$, and returns the number that is equal to `sorted(a)[k]`. If `k` is *not* in the valid range, return **`null`**

Example: Consider the `TreeMap` `{1L: 1, 4L: 2, 9L: 3}`. Among other possibilities, this could come from the underlying array `[9, 4, 1, 9, 4, 9]`. Once sorted, the array looks like this—`[1, 4, 4, 9, 9, 9]`. Hence:

```
fetchRank(h, 0) == 1
fetchRank(h, 2) == 4
fetchRank(h, 5) == 9
fetchRank(h, 11) == null
```

Expectations: Promises, Constraints, and Grading

- If the histogram contains n entries, your code must run within $O(n \log n)$ time for full credit. (*Hint:* The `TreeMap` is already sorted, so an $O(n)$ solution is possible. What do `.entrySet` and `.keySet` return?)
- The histogram will have no more than $10,000,000$ entries, but the underlying array `a` may be as long as $100,000,000,000$. It is guaranteed that every value (not key) in the Map is non-negative (i.e., ≥ 0).
- You can write as many helper functions/inner-classes.

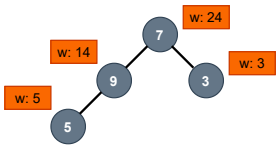
Problem 2: Weigh My Tree (10 points)

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]` $\implies$



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

- $1 \leq$ `a.length` $\leq 50,000,000$. `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 3: Minimum Operations (10 points)

Define the following two operations for a list of integers `l`:

- `rotate(l)` returns a new list after moving the back number to the front—i.e., it returns `l[-1] + l[:-1]` in Python's list notation.
- `reverse(l)` returns the reverse of the `l`—i.e., it returns `l[::-1]` in Python's list notation.

Nand and Nor are psychic pairs. When Nand comes up with a list `S`, Nor comes up with a list `T` such that `T` can be obtained from `S` by applying zero or more of `rotate` and `reverse` operations in some order. As an example, consider `S = [1, 2, 3, 4]` and `T = [2, 1, 4, 3]`, we can see that

```
T = rotate(rotate(reverse(S)))
```

But this is not the only way to transform `S` to `T`. To illustrate, here are some other sequence of operations:

```
T = reverse(rotate(rotate(S)))
T = rotate(rotate(rotate(reverse(rotate(S)))))
T = reverse(rotate(rotate(reverse(reverse(S)))))
```

Our goal in this problem is to find the smallest number of operations to achieve this transformation.

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

```
public int minimumOps(List<Integer> S, List<Integer> T)
```

that takes as input two lists `S` and `T` and returns **the smallest number of operations** to transform `S` into `T`.

Examples: Here are some examples on lists of length 5:

```
minimumOps(List.of(1,2,3,4,5), List.of(2,1,5,4,3)) == 3
minimumOps(List.of(5,4,3,2,1), List.of(1,5,4,3,2)) == 1
minimumOps(List.of(1,2,3,4,5), List.of(5,4,3,2,1)) == 1
```

Expectations: Promises, Constraints, and Grading

- We promise that `S` and `T` will always have the same length n. You can expect $1 \leq n \leq 500$.
- We also promise that it is always possible to transform `S` into `T` in this way. Moreover, the input lists will have distinct numbers (no repeats).
- For full-credit, your code must finish within 1 second on each of the test cases.
- If you want to put array lists into a hash set (i.e., `HashSet<ArrayList<Integer>>`, you should know that this combination works well and works as you expect them to. This will be useful for, for example, checking if you have seen a particular array list before.
- You can write as many helper functions/inner-classes.

(*Hint:* BFS. Each node in your graph is a sequence (e.g., `[2, 1, 4, 3]`))

Problem 4: 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:

```
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.