Problem Solving Workshop #48 Tech Interviews and Competitive Programming Meetup

February 9, 2020 https://www.meetup.com/tech-interviews-and-competitive-programming/

Instructor: Eugene Yarovoi (can be contacted through the group Meetup page above under Organizers)

More practice questions: leetcode.com, geeksforgeeks.org, hackerrank.com

Books: Cracking the Coding Interview, Elements of Programming Interviews

Have questions you want answered? Contact the instructor on Meetup, or ask on Quora. You can post questions and follow the instructor and other people who write about algorithms.

Try to find optimized solutions, and provide a time and space complexity analysis with every solution.

“Arithmetic On Entire Array”

Design a data structure that will act as an array of floating-point values, but supports some additional operations that act on the entire array. It should support the following classic array operations:

- **Constructor (int size)**: this is a fixed size that never changes after the array is created. Initially, all array positions should have value 0.
- **get(int index) -> float**: given an index, return the value of array[index] (which is a float).
- **set (int index, float value)**: set array[index] = value.

(a) **[Easy]** Additionally, support the operation **addToAll(float val)**. This adds **val** to every element of the array. Make this run in O(1) time.

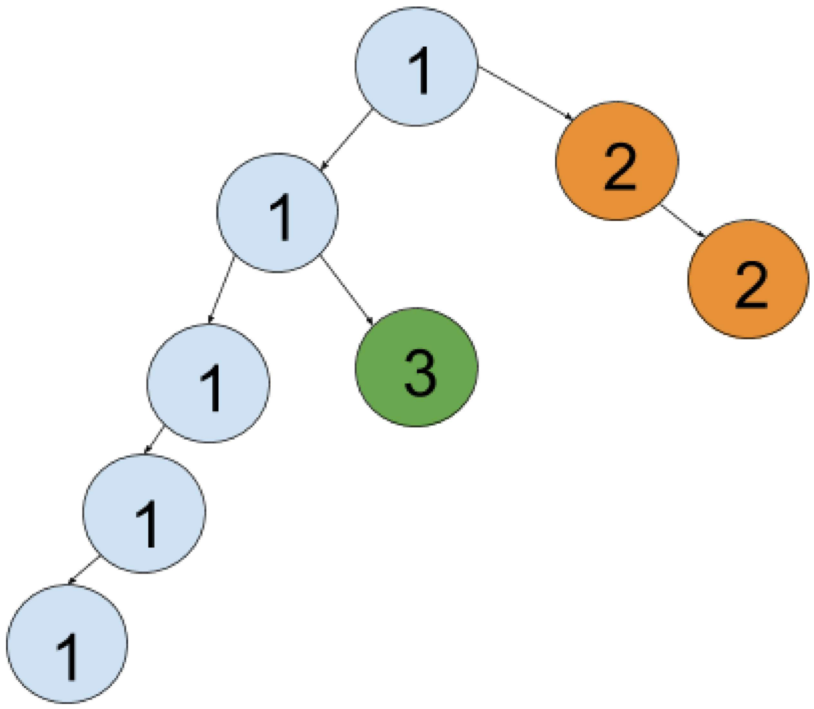
In other words, you will need an approach that ***conceptually*** does this without internally modifying each value individually (since that would not be O(1)). You may have to modify other methods such as set/get to be consistent with the approach you use for implementing addToAll. Note that calls to set/get may be used before/after calls to addToAll in any sequence. Values returned from the get method must be correct given the modifications made by set and addToAll.

(b) **[Medium]** Ignoring part (i) (you no longer need to support addToAll), support **multiplyAll(float val)** in O(1). This multiplies all values in the array by **val**. Careful, this is not perfectly analogous to the case of addition because a multiplication by 0 could happen.

(c) **[Medium]** Support both **addToAll(float val)** and **multiplyAll(float val)** in O(1) where both operations may be used in any order (and must be consistent with each other and get/set).

"Path Decomposition"

Given an N-ary tree, it can be decomposed into disjoint paths. Here, a "path" refers to a directed path that goes downward in the tree. For example, here is a decomposition of a tree into paths:



In the above diagram, the number indicates the number of the path. There are a total of 3 separate paths. Because we're considering only (downwards) directed paths, path #2 can't be absorbed into path #1, and in fact, 3 paths is the minimum number for this tree.

(a) [Easy] Given an N-ary tree, what is the minimum number of disjoint paths we need to cover all the nodes? Give an algorithm to produce such a minimal path decomposition.

Here's a more interesting question. We will define a notion we'll call the **path span** of a proposed path decomposition of a tree. Consider, for each node in the tree, how many distinct paths from the decomposition are visited on the path from the root to that node. For example, going from the root to the node marked 3, we visit paths 1 and 3, so 2 paths total. Now take the maximum of this quantity over all nodes in the tree, and that is the path span. It's essentially the worst-case number of decomposition paths that need to be visited.

For the path decomposition shown in the diagram, the path span is 2. As discussed, we can reach the node in path 3 by traveling on path 1 first and then path 3. We can reach any of the nodes in path 2 by visiting path 1 (we are forced to, since the root starts off in path 1) and then path 2. And of course, any nodes in path 1 can be reached directly from the root, without going on any other path. No node requires using more than 2 distinct paths to reach, so the path span is 2.

(b) [Medium] Given a tree and a proposed path decomposition (each node is labeled with a path number), give an algorithm to calculate its path span.
(c) [Medium-Hard] Given a tree, design an algorithm to decompose it into disjoint paths, in a way that minimizes the path span. Do something smarter than trying every possible decomposition.
(d) [Medium-Hard] If a tree has N nodes, what can you say about how large the tree span could be in a decomposition that minimizes it?

"Feasting on Grass" (Hard)

A cow is sitting at the origin of a one-dimensional number line. It's nice and peaceful there, being right at the center of things, but there are N clusters of hay positioned at integer coordinates around the cow, and she's getting hungry. As time passes, the hay gets progressively more stale. Let the staleness of a hay cluster be equal to the amount of time that passes from the start of the problem, to when the cow reaches the cluster, at which point it's consumed instantly. The cow can move 1 unit of distance per unit of time. Find the order in which the cow should consume all the clusters of hay to minimize the total sum of cluster staleness, and also report what that total staleness is.

Example Input: [-40, 7, 200]
Example output: [7, -40, 200]. Total staleness = 355
Explanation of output: The cow first visits the cluster at position 7, reaching it at t = 7. It will have staleness 7 when eaten. Then, the cow travels back to position -40, reaching it 47 units of time later (she's coming from position 7). That means t = 7 + 47 = 54 once she reaches it, and that cluster will have staleness 54. Finally, the cow travels to position 200, reaching it 240 units of time later (coming from -40), so at time t = 54 + 240 = 294. Thus, the final cluster will have staleness 294, and the total staleness of all clusters was 7 + 54 + 294 = 355. There is no way to get a smaller staleness, so this is the best strategy. (If ever there is a tie for optimal strategy, you can report any 1 such strategy.).