Problem Solving Workshop #39 Tech Interviews and Competitive Programming Meetup
September 9, 2018 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
Books: Elements of Programming Interviews, Cracking the Coding Interview
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.

Problem #1, "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-Hard]**. 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).

Problem #2, "Sliding Window Median"

(a) [Medium] Given an array of integers A, at every index i of the array, find the median of all the integers so far. In other words, output an array B where for each index i in A:
B[i] = median (A[0], A[1], ..., A[i])

(This is a classic question from Cracking the Coding Interview, but also commonly asked at companies. If you already know how to solve it, just recall the solution in detail and go to the next part; if not, solve it.)

(b) [Medium-Hard] Now, let's say we are additionally given an integer parameter K. Given A and K, find the median of every contiguous subarray of size K in A. That is, output an array B where:

B[i] = median(A[i], A[i+1], ..., A[i+k-1])

(B will be smaller than A since all the indices in the above expression need to be in-bounds.)

Problem #3, "Numbers Game" (Medium-Hard)

A turn-based two-player game is played as follows. There are three positive integers, which are given initial values at the beginning of the game. The two players alternate turns. Each player must, on their turn, choose exactly one of the integers and replace it with the average of the other two, rounded down to the nearest integer if the average is not an integer. For example, if the integers currently in play are 4, 9, and 12, the player whose turn it is may choose one of the following three actions:

1. Replace 4 with (9 + 12) / 2 = 10 (rounded down), changing the set of integers to {9, 10, 12}.
2. Replace 9 with (4 + 12) / 2 = 8, changing the set of integers to {4, 8, 12}.
3. Replace 12 with (4 + 9) / 2 = 6 (rounded down), changing the set of integers to {4, 6, 9}.

No player, however, is allowed to make a move that would not change which integers are present. In our example, suppose the first player chooses to replace 9 with 8 according to the second option above. It would now be the second player's turn to act on the integers {4, 8, 12}. The second player would not be allowed to choose to replace 8, because the replacement for 8 would be (4 + 12) / 2 = 8, which would lead to no change in the integers present. The second player therefore has only two options:

1. Replace 4 by (8 + 12) / 2 = 10, passing {8, 10, 12} back to the first player.
2. Replace 12 by (4 + 8) / 2 = 6, passing {4, 6, 8} back to the first player.

A player loses if there are no valid moves available on their turn. For example, if the set of integers on a player's turn is {3, 3, 3}, there is no available move that does not violate the constraint that a player must make a change to the integers present, since no matter which of the 3s is chosen, the replacement would be (3 + 3) / 2 = 3. Having no valid moves is the only way in which a player can lose; the opposing player then wins.

You are playing the above game against an AI. Unfortunately for you, it is a supercomputer that has already predicted all possible games on all possible inputs and knows all the optimal strategies to use against you. Your only hope is that you're allowed to choose whether you'd like to take the first turn, or let the computer play first. Can you find a way to always win the game?

Given the initial values of the three integers, determine whether you need to play first or second in order to be able to force a win against any possible opposition.

Example Input: 10 11 13
Output: Play first
Explanation: the first player can change the 10 to (11 + 13) / 2 = 12, passing {11, 12, 13} to the second player. The second player then has no choice but to pass either {12, 12, 13} or {11, 11, 12} back to the first player. The first player can then win by changing 13 to 12 in the first case, or 12 to 11 in the second case -- both {12, 12, 12} and {11, 11, 11} are losing positions for the second player. Therefore, no matter what the second player does, the first player can always force a win.

Notice, however, that if the first player had instead opened with changing 13 to (10 + 11) / 2 = 10 (rounded down), the second player would have received {10, 10, 11}, a position from where the second player could have won. The first player was able to win due only to making all the right moves throughout the game. In other words, it's not necessarily the case that the first player can win even if playing bad moves, only that the first player can always win if playing optimally.