COP 4516 Spring 2022 Week 14 Team Contest #6 Solution Sketches

Balloon Colors
This is the banger in the set. Just store their input for x and y, the colors that should not be given to the easiest and hardest problem, respectively. Then, when reading in the next n integers, store the first one as the actual color of the easiest (easy) and the last one as the actual color of the hardest (hard). Then see if x and easy are equal and also if y and hard are equal. Then, output according to the four cases given.

Bullseye
Binary search for the number of black rings to draw. A low bound of 0 is easy to calculate. We have to be a bit more careful calculating the upper bound to avoid overflow in longs. We can use the arithmetic series that we set up for solving the problem to calculate this upper bound. Once the bound is calculated, then a regular binary search can be run so long as a function is written that calculates in O(1) time the amount of paint used for some number of black rings and an initial radius.

Haircut
You can't simulate everyone getting a haircut, but you can binary search to **right before** you get one. Binary search on time. Guess a time. Given a time, it's easy to calculate how many people each barber will be able to complete hair cuts for. We want a time such that this number is strictly less than your place in line, but if we added 1 to the time, that is no longer the case. Once we get this time, we can figure out who has completed haircuts, who is sitting down for haircuts, and we can simulate the situation until we get a haircut, to figure out which barber will cut our hair!

Jumpman
Though awkwardly worded, the goal here is simply to find all the reachable squares and add up the sum of the values of the treasures at those squares. Thus, to solve the problem you can run either a DFS (floodfill) or BFS from the starting point, adding up the treasure values at all of the reachable squares. A square is adjacent to another square if the two are up, down left or right from one another and if the first square that we are jumping from is no more than j shorter than the second square you are jumping to.

Need for Speed
It's difficult to calculate c directly, but if we make a guess for c and plug it in, it should be pretty easy for us to see if our guess was too high or too low. This sounds like binary search!!! Consider the first sample case. Consider guessing c = 2. This would indicate that we drove 4 miles at 1 mph, 4 miles at 2 mph and 10 miles at 5 mph for a total drive time of 4 hours + 2 hours + 2 hours = 8 hours. Since this is above the actual trip time of 5 hours, we can infer that our guess for c was too low and we have to speed up each segment of the trip! The only "pitfall" in this question is setting low. Note that the specification indicates that c might be negative, so you can NOT set low = 0. Instead low can be set to the negative of the minimum of the speeds listed in the input, so that when you add the lowest possible value of c to the smallest speedometer reading you would get 0. (Note that in the binary search, we'll never actually plug in low, so there's no need to worry about a divide by zero error.) High just needs to be high enough and in the worst case, t = 1 and we have 1000 segments of 100 miles, so the fastest possible speed we would need is 1000000 mph. Since the worst offset is -1000, we would be safe setting high to 1001000. There is no harm in simply setting it to 2,000,000.

Tri graphs
This problem requires a dynamic programming solution. Create a table to store all answers to subproblems that is the same size as the input r x 3 grid. Then, go through the grid in the usual order. By the time you get to a square, you have already calculated the results to the subproblems you need (which are conveniently labeled with the arrows in the problem statement!). Basically, you just want to take the minimum result of all possible previous squares and add it to the value in your square, to get your result.