Airport Shuttle
If we knew how long the counselors were willing to wait, then we could run a greedy algorithm to figure out the number of different airport runs that need to be made. We would simply do this by arriving exactly when the first counselor arrives and waiting the fixed maximum wait time and then leaving. Repeat this process over and over again until all students are picked up. Since no counselor wants to go twice and we know the number of counselors we have, we know that for some fixed wait time, \( w \), if we need to make more airport runs than there are counselors, this means that \( w \) is not big enough. Alternatively, if we make the same number of fewer airport runs with a wait time of \( w \), then the minimal wait time necessary to pick up the students can not exceed \( w \). These two properties make the minimum valid value of \( w \) binary searchable. The lowest possible answer for \( w \) is 0 (there is one counselor for every kid), and the highest possible value for \( w \) is the difference in the minimal and maximal arrival times of the students. For any value of \( w \), write a function that returns a boolean, signifying whether or not the required number of counselors can pick up all the kids without waiting longer than \( w \), for each airport run, and use this function's return in the binary search to adjust low or high.

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.

Generation of Tribbles
This is just like Fibonacci and the Dinner problem from a couple weeks ago. Just do the calculation iterative, or if you do it recursively, then memoize.
**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!

**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.