# COP 4516 Spring 2021 Week 15 Final Team Contest Solution Sketches

*Problem A: AM/PM*
This was meant to be the easiest problem in the set. It involves a bit of case work. In most cases, the output hour is one more than the input hour. The exception is when the input hour is 12. In this case the output is 1. Similarly, in most cases, the string remains the same. The exception is when the input hour is 11. In this case, we always switch our output string from "AM" to "PM" or "PM" to "AM". Just use if statements to catch these two exceptions.

*Problem B: Banana Tree*
In COP 3502, a standard algorithm is taught to determine the height of the tree. Get the height of each subtree, take the maximum of these, and add 1. In this case, the only edit would be, instead of adding 1, add the number of bananas at the root. By just making this slight change, the recursion naturally returns the maximum # of bananas down any path of the tree starting at the root.

*Problem C: Cantaloupe*
Assuming that our input list was sorted (it isn't so we just sort it), if we knew which "gap" we were trying to achieve, we could use a greedy algorithm to take the fruit: namely, take the first possible fruit, and then from that point on wait the minimum required time and then take the next possible fruit and repeat. At the end of this simulation, if we have enough fruit, then the gap we just tested was achievable. If we don't, it wasn't. Thus, we can use this information to do a binary search on this minimum gap. We can set low = 0 and high = $10^{18}$ and just make sure not to overflow (this isn't too hard since not intermediate calculation should exceed 2 x $10^{18}$), and then run a typical binary search on the answer.

*Problem D: Dinner*
This problem can be solved using network flow. Set up a source vertex with edges of capacity 1 to a vertex for each person who might come to dinner. Then, add edges from each vertex representing a person to each vertex representing a meal they might eat, also with capacity 1. Finally since each meal is limited to one person, add an edge from each meal to the sink with capacity 1. The maximum flow of this graph also represents the maximum number of people who can be invited to dinner.

*Problem E: Expanded Excel*
This is the second easiest problem in the set. Scan through the input keeping note of the maximum row and maximum column. Also, for each column (there are at most 26 so just use an array of size 26), keep track of the maximum necessary width in pixels. Initialize this array to the default. For each new entry processed, update the appropriate column width, if necessary. When this is complete, then multiply the number of rows by the number of pixels tall each row is. This is the height of the spreadsheet in pixels. To get the width in pixels, just sum up the number of pixels needed for each column, up to the maximum used column. Then, multiply these two values.

*Problem F: Food Truck*

We can model each grill as an item (integer is good enough) in a priority queue. Any time we get to an item in the queue, pull the next grill that will be free from the priority queue. Then add to the time that grill is free the cooking time for that person's item (easily stored in a map of food items to cooking times). This is when this person's food will be ready. Store this, and then reinsert into the priority queue this new time, because this is when a new grill will be free. Complete the simulation and each person will have an associated time their food is complete. Finally, just sort this list by time, breaking ties by name (due to the nature of the names, both alphabetic and lexicographical comparisons will yield the correct result.)

*Problem G: Golden Kite*

This may be the most challenging problem in the set because it mixes two different skills: (a) brute force, and (b) geometry. Since there are only 5 sticks, it's easy enough to try each permutation of stick lengths to match the given locations. (Technically there are only 30 meaningfully unique orientations, but it's probably easier just to try all 5! = 120 arrangements and redundantly try 90 of them since there are so few to try to begin with. To get all the meaningful arrangements: we have 5 choices for the middle stick, c. Then, we can choose any 2 out of 4 remaining sticks in 6 ways to form one triangle. At that point, the other triangle is fixed and it doesn't matter which sides, top or bottom, the two triangles go. Thus, there are a total of 5 x 6 = 30 unique combinations worth trying.) To get the result for a permutation, just use Hero's formula to calculate the areas of triangles with sides a, b and c, and sides c, d and e. Make sure to check that the sum of the lengths of the two shortest sides exceeds the length of the longest. If it doesn't, it's important not to do the mathematical calculation and instead skip the evaluation. (In my code, I return -1 for any degenerate triangle without doing the math and then if either triangle area is less than 0, I skip that evaluation.) Then, just take the maximum area of any of the valid permutations.

*Problem H: Hopping Bunny*

This is a classical dynamic programming problem, but it's probably easier to view it going "forward" than backward. Define f(x, y) to be the maximum score that can be achieved by starting the game and row x, column y. With this definition, our goal is to maximize f(0, 0). Denote the jump from row x, column y to be jump[x][y] and the number of carrots at that location as carrots[x][y]. Then, we have:

f(x, y) = max( f(x+jump[x][y], y), f(x, y + jump[x][y]) ) + carrots[x][y]

It's probably just easiest to write this code recursively and then memoize the solution, making sure not to access any array out of bounds. (If any jump takes us out of bounds, immediately return 0 for the ensuing score.)

Alternatively, we can write a traditional DP solution by filling in the DP array "backwards" (going from the last row to the first and within a row, last column to the first).