2020 MAPS Problem Review

Problem A: Buried Treasure

There is no problem obfuscation here. This is a backtracker. Read in all of the pieces and you know what the area of the puzzle must be. Try all possible dimensions of the puzzle, based on divisibility. For each possible dimension, start placing pieces, from the top left square onward. When you place a piece, you must see if it's consistent with what is placed. This means that all adjacent squares, which we can reduce to checking square (x,y) with (x, y+1) and (x+1, y) for each (x, y). Since these values represent Manhattan distances from some fixed point, the absolute value of their difference must be 1 or 9 (for distances 9 and 10). If any of these fail, the piece can not be placed. The last annoying thing is that you have to try all four rotations of each piece. I did this by pre-storing each of the rotations when I first read in the pieces, but one can use some math instead and just store one copy.

Problem B: Divide and Conquer

The 11 divisibility rule works because $10^1 = -1 \bmod 11$. (So, at odd digits we subtract the mod and at even digits we add it.) The way they have sectioned off the number it's like

$(num4)b^{4k} - (num3)b^{3k} + (num2)b^{2k} - (num1)b^{k} + (num0) = (num4) + (num3) + (num2) + (num1) + (num0) \bmod d$

This only works if $b^{2k} = 1 \bmod d$ and $b^k = -1 \bmod d$. The latter is how we get the alternating signs.

So, given b and d, we must ask ourselves, does there exist a k such that $b^k = -1 \bmod d$.

This looks like the discrete log problem, but we have some help. We know that d is prime. This means that $b^{d-1} = 1 \bmod d$, by Fermat's Theorem. Also note that $(-1)^2 = 1$. So, if we want to find an exponent for which b raised to it yields -1, a good place to look is (d-1)/2. It turns out (via the Miller-Rabin Primality Test), that this isn't quite sufficient, but what you want to do is divide out all powers of 2 from d-1, so express $d-1 = x2^m$, where x is odd and m is the number of 2s in the factorization of d-1. The candidates you want to test to see if they are equal to -1 mod d are b^x, b^{2x}, b^{4x}, … , b^{d-1}. If any of these equals -1 then it's possible!

Problem C: Easy Multiplication

Problem D: Greedy Polygons

The formula for the area of a regular polygon is 1/2 * a * p, where a is the apothem and p is the perimeter. The perimeter is easy to find. The apothem can be found by the triangle formed by one side of the polygon and two sides drawn from the endpoints of the polygon side to the center of the polygon. Specifically, the apothem is the height of this triangle. We know that the top angle of this triangle is just 360/n, where the polygon has n sides. Cut this in half and we get 180/n. This is the angle that is part of the right triangle. Finally tan (180/n) = (L/2)/a, where L is the side of the polygon. This solves the problem for 0 expansions.

Now, let's talk about the expansions. It should be easy to see that the sides themselves each expand out as rectangles of dimension L times G times D, where G is the number of expansions and D is the length of each expansion.

This leaves the curved areas, which, if you stare at them long enough, you realize form a single circle with radius G times D. (Each one is a sliver of a circle with central angle 360/n, where n is the number of sides.)

Add up these three parts and you are good to go!

Problem E: Impossible Prices
The key is to work all the math out in longs. Convert everything so that prices are in cents and the tax is an integer in between 0 and 10000, inclusive. An untaxed price P will equal P(10000 + tax) as a taxed price in our new integer numbering system. We can't try each price in the interval. But, given a taxed interval of [A, B], we can use a binary search (or math and search the three nearest values) to find the value C which is the largest value that, when taxed is <= A, and find the value D, which is the largest value that, when taxed is <= B. There are A - B + 1 values in the input range. We know that C must either map to A or A - 1. If it maps to A, then there are D - C + 1 untaxed values that map into the range [A, B], meaning that the number of missed prices is equal to (A - B + 1) - (D - C + 1). Alternatively, if C maps to A - 1, the result is (A - B + 1) - (D - C). The idea is that each unique untaxed price maps to a unique taxed price, so the subtraction of these sets represents the number of missed values.

Problem F: Keywords
This was the banger in the set. For each input, make it lower case (or upper) and replace each " " with "-" (or the other way around). Then store the mapped item into a set of some sort. Then, output the size of the set when done.

Problem G: Litespace
This is a simulation, just do exactly what they say, making sure you don't misstype any letters. Good luck! (Debugging this is pretty annoying if you initially have an error…)

Problem H: Magical Cows
The key here is that each farm with the same number of cows "acts" the same. So, keep a frequency array such that array[i] stores the number of farms with i cows. Then, just simulate the process. For one day, we run a for loop through an array of size 1000, and we do this for at most 50 days. You can just pre-compute all the answers and then answer the queries. The key is realizing that if 2*i <= capacity, then we want to add array[i] to our frequency count of index 2*i for the following day. If this isn't the case, we add array[i] to two potentially different indexes of the new frequency array.

Problem I: Round Trips
The key observation for this problem is that if there is a strongly connected component (SCC) in the graph, no edge can be added to it. Recall that an SCC is a set of vertices in a directed graph where there exists a path from any vertex to any other vertex. Thus, let's say we add the edge u→v to an SCC, which means that previously, there was no such edge. Before adding that edge, we

know there must be a path from v to u. Adding the edge u→v to this path adds a cycle that wasn't previously there.

A second observation is that if two vertices, u and v, are in different SCC's of the original graph, in order for no cycles to be added, at most one edge of u→v and v→u can be present. (Having both present would merge the SCCs into one.) Notice that some of these edges already exist in the original graph and thus, can't be added.

Using these observations, a solution now becomes clear. It's easier to subtract out edges that aren't allowed to be added from the whole than the other way around. We start with n(n-1) possible edges. Subtract out all the edges already in the graph. Within each SCC, count how many edges are within the SCC and how many go from a vertex in the SCC to outside of the SCC. Call the former number in, the latter number out, and let c = in+out. The maximum number of edges that could be in the SCC is c(c-1). Subtract out in from this (so c(c-1) - in) and this value represents the edges that aren't in this SCC that aren't allowed to be added, so subtract this out of our main count. In addition, subtract out the edges u→v leaving the SCC to acknowledge that the edge v→u can't be added. When we are done going through this process, the only edges unaccounted for are edges of the form u→v and v→u between two vertices u and v in different SCCs where neither edge was included in the original graph. For all of these cases, at most one of these edges can be added in, not both. So, we must take our "possible" number of edges we could add in and divide it by 2 to get the final answer.

Problem J: School Spirit
The limitations are small enough on the data that one can use pure brute force to solve it. At most there are 51 lists of sorted numbers, where each list is no longer than 50 numbers, to which the given formula must be applied. You are to apply the formula to the full given list (first output), and then apply it to every list created by removing exactly one score and then take all of these school scores and output their average (second output). One can write one function for both tasks by having the function take in which score (if any) to skip in the calculation.

Problem K: Structural Integrity
Note: I haven't solved this problem yet, but know roughly how to do it, so this is what's included in the write up. If I do solve it, I'll update the write up accordingly.

This is an optimal triangularization problem, which has a similar dynamic programming structure to Matrix Chain Multiplication. Given a polygon with a starting vertex, s, and an ending vertex, e, we may either draw a line from s to a non-adjacent vertex in range, or we may clip off the "ear", connecting e to s+1. Unfortunately, not all lines are allowed to be drawn. A line isn't allowed if any portion of it is outside the polygon. Thus, for each possible line segment we may draw, we can pre-compute if it's completely inside the polygon or not. There are two things to check:

1) Does the line segment intersect any of the polygon line segments?

2) Is the line segment completely outside of the polygon?

#1 can be checked using standard line segment intersection code. Make sure to only count intersections strictly on the inside of the polygon and not at any endpoint, since some endpoints are guaranteed to meet and shouldn't count. For #2, take the midpoint of the segment and check if it's inside or outside the polygon. To facilitate #2, double the coordinates in the input so that the midpoint is guaranteed to be integral and use longs for everything. One possible way to do this is to draw a line segment from this midpoint to a point out of range (say (20100, 20101)) and count the number of intersections with the polygon sides. If this number is odd, the point is in the polygon, otherwise it's outside the polygon.

Problem L: The Wrath of Khan

Notice that in when running the algorithm, no node that is either in a strongly connected component (SCC) nor a descendant of an SCC can ever be reached. Thus, we can safely remove these nodes. The easiest way to do this is just run the algorithm and mark every node ever placed into the queue of nodes with no incoming edges.

Now, the portion of the graph that remains with only these vertices and the edges between them is a directed acyclic graph.

The problem is asking to find the maximal number of independent nodes (where one can't be reached by another) in a DAG. This can be solved applying Dilworth's Theorem which says that in any finite partially ordered set, the largest antichain has the same size as the smallest chain decomposition (Wikipedia).

To find the smallest chain decomposition, we can utilize max flow. For our partially ordered set, we must record all sets of elements such that $p \preccurlyeq q$, where p comes before q in the partial ordering. For this problem, that simply means recording all ordered pairs of distinct vertices (u, v) such that there is a path in the restricted graph from p to q. We can calculate this information running several DFSs (or Floyd's runs fast enough, at least in C++). If there are n reachable vertices in the DAG, then the flow graph will have 2n+2 vertices, a source, a sink and two nodes for each node in the DAG. To create the flow graph, place an edge with capacity 1 from the source to copy #1 for each vertex. Then place an edge with capacity 1 from copy #2 of each vertex to the sink. Finally, for each distinct pair of vertices (u, v) where there is a path from u to v, place an edge from the first copy of u to the second copy of v with capacity 1. Each unit of flow represents the removal of a vertex from the maximal anti-chain. (If the edge u->v has 1 unit of flow through it, then both u and v can't be in the maximal anti-chain, so at least 1 must be removed.) Thus, the answer to the question is the number of nodes in the original graph, minus the number of unreachable notes in the original graph, minus the max flow of this graph described, since each unit of flow represents a vertex we can't add to our maximal set.