COP 4516 Spring 2025 Final Individual Contest Solution Sketches

Problem A: Almost Magic Square

This problem was meant to be a straight-forward permutation problem. Try each permutation of the 9 input numbers given. For each permutation, calculate the 8 desired sums (3 rows, 3 columns and both diagonals). Take the maximum of these 8 sums and subtract from it the minimum of these 8 sums. If this number is less than or equal to d for the case, add 1 to a running tally.

Problem B: Carrying Power Strips

For option 1, the time it takes to carry 2 boxes to PSY-111 from the parking spot is 3a, and followed by 2b time to carry 1 box from PSY-111 to CB1-119. Thus option 1 takes 3a + 2b seconds. For option 2, the time it takes to carry one box to PSY-111 from the parking spot is 2a seconds, followed by another a seconds to walk back to the car. From there, it will take 2c seconds to carry the remaining box from the car to CB-119. This takes a total of 3a + 2c seconds. Since the 3a contribution is equal, we see that option 1 is better if b < c, option 2 is better if c < b, otherwise, if b = c, then both options take the same amount of time.

Incidentally, I've arrived at option (2) because I was spilling coffee from my mug when I tried to carry 2 boxes. It also feels faster, though I don't really know!

Problem C: Which Group?

There are several ways to solve this problem. Perhaps the most straight-forward is simply to create a map that maps each student to either 1 or 2, and then for each query, just use the map to get the mapping of that student (if one is stored, if not -1 should be outputted).

Alternatively, we can store two sets of names, one for group 1 and another for group 2. Then, for each query, look in both groups for the queried name and output accordingly. If the name is in neither group, output -1.

Problem D: Follow the Rainbow Brick Roads

There are 127 possible graphs that all of the queries could be on. This is because for each query either a color is in the query or not in it, for a total of $2^7 = 128$ subsets of colors. But, the empty subset is not allowed as a possibility for a query.

Thus, for each of these possible graphs, add the appropriate edges and run either Dikjstra's or Bellman-Ford on it. (Note: due to the rather strict restriction on the number of edges, Bellman-Ford is an option. For denser graphs this wouldn't have worked.) This will store all the shortest distances from the starting vertex 1 to all other vertices for all subsets of roads. (This full look up table would be of size 127 by ***n***, where ***n*** is the number of vertices in the graph.) Once all of these shortest distances are pre-calculated, then for each query, look up the desired shortest distance. For the purposes of implementation, the easiest way to store a subset of colors is a bitmask in between 0 and 127, where each individual bit is set to 1 if that color is included in the subset or is set to 0 if it's not. This system makes it easy to store the shortest distance arrays as the first index is the desired bitmask for the corresponding subset and the second index is the destination vertex.

Problem E: Divisor Series

This is likely the hardest problem in the set. One key idea to recognize is that since $\boldsymbol{n} \le 10^6$, for any given term of the form ***id***, ***i*** will have at most 1 prime factor greater than 1000. Run a prime sieve to generate the 168 primes less than 1000. Then, for each integer from 1 to 1,000,000, out all copies of these primes and store which ones divide into i. If the leftover number isn't 1, this one remaining number is prime. Add this to the list of primes that divides into i. Thus, for each integer upto a million, we can precompute the list of unique primes that divides into it in roughly 168 million steps. (We could improve on this bound by building off of previous cases, but the run times bounds were such that this was good enough.) This pre-computation will be used across all 25 cases.

Now, let's talk about handling each individual case. It's easy enough for each test case to obtain the prime factorization of ***d***. The issue is the computation of the number of divisors of each integer in the sequence ***d***, ***2d***, ***3d***, …, ***nd***. With the pre-computed list however, we can do the following:

Run a for loop through each value of i in the sum from 1 to n. Our goal will be to swiftly calculate the desired number of divisors. Since we already have all the unique prime divisors of both ***i*** and ***d***, precalculated, we can completely prime factorize the product ***id*** by only dividing through by these prime divisors. It's important to handle duplicates only once. (If, for example, 3 divides into both i and d, make sure to properly count the total number of times 3 divides evenly into the product ***id***. In practice, this total list will tend to be quite small, so this code actually runs quickly since each prime we try actually divides into the number evenly. The number of unique prime factors an integer formed in the way described can have is reasonably small (just try the product 2*3*5*7… for a few terms…) so this runs faster than one might imagine.

What forces us to go through each term one by one is the fact that for some values of i, i and d share some prime factors and for other values of i, perhaps no prime factors are shared. It varies. But, the calculation of number of divisors requires us to "merge" our knowledge of the number of times each prime factor divides into a number because $3^2 3^3$ doesn't have $(2 + 1)(3 + 1) = 12$ divisors, it has $(5 + 1)$ divisors since it should be written as 3^5. This is what forces us to go through each term one by one, and we need to use the pre-computation to speed up our prime-factorization calculation for each term.