COP 4516 Spring 2026 Week 3 Team #1: Brute Force (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: Covid 19

This problem was the "banger" (easiest problem) in the set. For each test case, loop through each person. For each person, if their latter weight minus their former weight equals 19, add 1 to a counter. Then, just print out the counter after processing each test case. Don't forget to set the counter to 0 at the beginning of each case!

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: Hexagram

There are 12! ways to put the numbers in the circles. Unfortunately, 12! is quite big (several hundred million) so straight brute force doesn't work. The key observation to solve the problem is to recognize that each circle gets added into exactly 2 row sums, so the sum of all six rows equals twice the sum of all the input numbers. This means that for the 6 rows to all have the same sum, let this sum of each row be R and the sum of all the 12 input numbers be S, we must have 6R = 2S. It follows that R = S/3. Thus, before we start our brute force search for solutions, we can precompute what the row sum will be. Then, when we write our brute forcer, whenever we finish any of the six rows, we can immediately check if that row adds to the proper target. If it does not, skip placing that last item in the row and move onto the next one (essentially skip trying a number in a circle that is doomed to fail.)

Problem E: Passwords
If you try to physically store each possible password, more than likely your code will receive a time limit exceeded due to the fact that code slows down when you use a lot of memory and that creating each password could potentially use quite a bit of memory. If you just create one single string and change its contents during your recursive brute force search, your code should easily run in time.

Thus, one solution is to simply write a recursive function that takes in the current password (char array), an integer k, representing the number of fixed characters in the password. The function should return the desired string. A "global" variable can be used to keep track of how may passwords have been generated, so that you can cut out of the recursion when the correct password is reached. The code would look very similar to what was shown in class where the recursive portion does a for loop through each possible character for index k of the password.

Alternatively, we can realize that if there are n_i choices for the ith letter, then the total number of possible passwords is $\prod_{i=1}^{m} n_i$. Furthermore, if the first k letters are fixed, then the total number of passwords with those letters fixed is $\prod_{i=k+1}^{m} n_i$. Now, consider the following, slightly easier problem: given the first k-1 letters fixed, and the 0-based rank, r, we desire, figure out what letter the kth letter should be. We know that except for the kth letter, there are $\prod_{i=k+1}^{m} n_i$ arrangements of the rest of the letters. Let this be X. It follows that of the possible choices for the kth letter, the $\left\lfloor \frac{r}{X} \right\rfloor$+1 of the possible choices. Once we can solve this subproblem, then we can iteratively solve for each letter. Consider the following example:

First Letter: a, g, h, m
Second Letter, b, c, d
Third Letter: e, f, n, o, p, t
Fourth Letter: r, s

Find the 98th ranked possible password.

Subtract 1 from 98 to get 97. Note that 3 x 6 x 2 = 36. Thus, there are 36 passwords that start with 'a', another 36 that start with 'g' and so forth. Calculate 97/36 = 2 via integer division, which indicates that two full sets of 36 letters compete before we get to rank 97, so the first letter of the password is 'h' (index 2 when using 0-based indexing). Next, take 97 – 2 x 36 = 25. So, now, we are looking for the 25th ranked password that starts with h. Since 6 x 2 = 12, we want the 25/12 = 2 index letter from list two, so the password starts "hd". Now, take 25 – 2 x 12 = 1, so we are looking for the 1 ranked password (0-based) starting with "hd". 1/2 = 0 so we want the 0 index of the list of letters for the third letter. Thus, the password starts "hde" and we want the 1 – 0 x 2 = 1 ranked password that starts with "hde". Since 1/1 = 1, we tack on letter in index 1 for the last list and the desired password is "hdes". Note that it's guaranteed that the product is $\prod_{i=1}^{m} n_i \leq 10^9$, so that none of our potential calculations will cause an integer overflow (long isn't necessary).

Problem F: Up-wards
Loop through each pair of consecutive letters. (So, if the word has n letters, the loop runs n-1 times.) If we ever have an indexes i and i+1 such that word[i] ≥ word[i+1], then the word is NOT an upword. If this situation never triggers, it is an upword.