COP 4516 Spring 2019 Week 1 Individual: Brute Force (Solution Sketches)

Hexagon Perplexagon

At first it seems as if you might have to try all 7! permutations of ordering the pieces and then also try all 6^7 orientations of those pieces (since each of the 7 pieces can be placed in one of 6 rotations). But, once we realize that the problem asks us to pre-rotate the middle piece to have 1 at the top, then, we realize that it ***forces*** one particular rotation on the other 6 pieces because each of these 6 pieces shares an edge with the middle piece and it's required to rotate the outside piece so that the appropriate edge matches the already fixed shared edge of the middle piece. (For example, if we look at the picture in the problem statement. Once 1 is fixed at the top for the middle piece shown, when we take piece 0 and place it up top, we know that piece 0's edge 1 must line up on the bottom since that edge is shared and matches the 1 from the middle piece shown at the top.)

So, the solution is as follows: write generic permutation code to try all 7! permutations of the pieces. When evaluating a single permutation (a permutation will evaluate to either true or false, true for a valid solution, false for an invalid one), first rotate the middle piece so that 1 is at the top. Then, rotate each of the six pieces so that their shared edge with the middle piece matches. Finally, check all the ***other*** shared edges between the outer six pieces (there are six such edges) and make sure that the corresponding edge labels that are supposed to be equal are actually equal. If any pair that is supposed to be equal isn't, return false, otherwise return true. Based on the problem statement, either one permutation will return true or none will. If one does, print it, otherwise, if none does, print no solution. It's probably best to have a function that performs a single rotation which takes in an int array of size 6 and returns a new int array of size 6 that stores the result of a single rotation. Other than that, a function that evaluates a permutation is important to have. Not changing the original input and creating copies for evaluation will probably result in fewer bugs, on average, for the initial implementation. While this slightly slows down the code, it won't slow it down to the point that it'll run slower than the time limit given.

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).

Movie Trip

Just multiply the first and third numbers and the second and fourth numbers and add the two products. Finally, express the result rounded exactly to two decimal places. printf in either Java, C, or C++ performs displaying to two decimal places most easily.

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.