Scott Calnan

Fidelity Investments

6/3/19

Quantitative Developer Coding Exam

**Part A:**

This code takes in a string (S) of letters from the English alphabet whose length is divisible by three, a second string (C) of letters from the English alphabet, and an integer (N). I started out by tackling Part 1 of this question: checking the raw inputs from the user to make sure that the above-mentioned requirements were met for the three variables. I defined two arrays, one with all the characters of the English alphabet and the second was all integers between 0 and 9. The function match() was created to return False if there were any instances of unallowed characters from either of these two arrays. S and C were both checked to make sure that they were wholly characters from the English alphabet. S also went through a length test to check if the inputted string was divisible by three. N was tested to make sure that it was an integer.

Strings were asked to be treated as arrays so the function make\_array() was developed to take in a string and return an array of characters. While designing make\_array(), it was determined that a length() function was also necessary since libraries were forbidden from use during this test. Like python’s built in len() function, this takes in an array or string and returns the number of characters or items. The function make\_string() was designed so that arrays could be outputted as strings to the user.

Part 2 was then tackled with a single function, shift(). This function takes in a string and shifts the letters by an integer N and returns the shifted string. The array of English alphabet letters, alph, was used to shift these letters. Two nested for loops are used to match each letter of the inputted string with its respected letter in the alphabet. By taking the modulo of the sum of the character’s position in the alphabet and N, the result was a shift for this character’s letter. The modulo operand took care of wrapping from a to z or vice versa. This function took in C and developed CSHIFT for Part 3.

The code for Part 3 had many more intricacies than expected but it successfully swaps the instances of C in the first third of S with instances of CSHIFT in the second third of S. I assumed that it was an equal swap for the instances of the two strings. For example, if we saw two instances of C but only one instance of CSHIFT we would only C and CSHIFT once.

My mentality was that it would be better to create several functions in the process of swapping the two sub-strings, C and CSHIFT. That way when moving onto Part C, most of the functions would be useful and some may only need to be tweaked slightly. I started by building a function that would check if a sub-string, like C, was in a larger string, like the first third of S. The function find\_sub\_string() takes in two strings. If the first letter of the sub-string was in the string, then I called a separate function, strings\_equal(). This function would take in two strings and the index where the letters were equal and check if the remaining characters of the sub-string were in the string. strings\_equal() would either return a -1 if the sub-string was not in the string or it would return the index of where the sub-string starts in the string. This index was then appended to an array of indices in find\_sub\_string(). The reason I broke it up like this was because I thought I may want to edit the strings\_equal() function or call a different function in find\_sub\_string() when it came to Part C.

Once the indices were known of the sub-strings and string, they needed to be swapped which was done with the swap\_strings() function. This was fairly straight forward since the sub-strings, in this case, were the same length. After swapping the two sub-strings, the new string was returned.

Finally, switch\_instances() pulls everything together. This function takes in S, C, and N and returns the final string S with swapped C and CSHIFT instances. S is broken down into two separate strings, one that includes the first third of S and another that includes the other remaining two third. CSHIFT is created and then instances of C are checked in the first third of S and instances CSHIFT are checked in the remaining two thirds of S. The integer *iterate* is formed from the function find\_sub\_string() to determine the total number of times were going to be swapping the two sub-strings. We then go through and swap each instance with a for loop and replace C at index i1 in S with CSHIFT at index i2 in S. After this for loop is over, the switch\_instances() returns the modified S. After some testing, I had to add in a check to make sure that the switch\_instances() function wasn’t looking for C past the one third mark of S. The resulting modified S string is then displayed to the user. Unit testing was performed along the way for each function.

**Part B:**

I took the solution in Part A and edited it to work using recursion. The console application still asks the user for a raw input of a string S that is only letters of the English alphabet and whose length is divisible by three, a string C that is only letter of the English alphabet, and an integer N. If the user does not meet these requirements an input error message pops up and the user is asked to reinput the variable.

The functions length(), make\_string(), make\_array(), shift(), strings\_equal(), swap\_strings(), and match() were all kept from Part A. The function find\_sub\_string() was changed to recursive form since the main problem is finding the indices to swap C and CSHIFT in S. The new function became recursive\_find\_sub\_string() which took inputs of a string, a substring, a starting integer for the recursion step, and a starting value for the indices array. It would output the indices array with the locations of the substring in the string. This function would loop through all characters in the inputted string and determine if the substring existed by using the strings\_equal() function. The base case for this recursive process is that the function had recursed through all the way through the string until it was the length of the substring from the end. The recursion would split and return indices which was then used in switch\_instances().

The function switch\_instances() called the recursive\_find\_sub\_string() to determine the indices of C and C\_shift in the first third and remaining two thirds of S respectively. These indices were used in the swap\_strings() function, similar to Part A. The resulting modified string is then displayed to the user. Unit testing was performed along the way for each function.

**Part C:**

As mentioned in Part A, several functions were created in hopes to make managing with Part C easier. The code program, again, made sure that the raw inputs followed the guidelines provided. A string S was all English alphabet letters and its length was divisible by three, C was a string that was all English alphabet letters, and N was an integer. I kept all the old functions but edited switch\_instances() and swap\_strings() then added find\_dispersed\_sub\_string(). The new function was built to deal with the dispersed characters of the CSHIFT string. Instead of returning an index where CSHIFT starts, the indices of all of the characters needed to be known. Therefore the find\_dispersed\_sub\_string() was built to take care of this matter. It takes in a string and a sub-string then finds the indices of all the characters of the sub-string in the string and returns these indices as an array.

The function swap\_strings() still replaces CSHIFT in the first third of S by replacing characters consecutively. Replacing C with instances of CSHIFT in the remaining two thirds now happens using dispersed indices found by the find\_dispersed\_sub\_string() function.

The switch\_instances() function which is the main function for finding and calling the swaps originally gave me troubles because after it found the indices of CSHIFT the first time and made the swap, it would sometimes reuse those old index values. Therefore, I was swapping already swapped characters. I tried several ways to deal with this using a history variable that stored all the indices of the previously swapped dispersed CSHIFT characters. Unfortunately, I had minimal luck and decided to change one of the temporary S strings that were built into the function. The variable rest\_of\_S is a string that is created at the beginning of switch\_instances(). This string is fed into find\_sub\_string() in Part A and find\_dispersed\_sub\_string() in this problem. By performing the swapping on S and finding the indices on rest\_of\_S. I was able to change each one independently. All the past swaps of CSHIFT on S, I changed to commas on rest\_of\_S so that when I searched for the next occurrence of CSHIFT the old indices wouldn’t be English letters. This is most likely not an optimal way to solve this problem but was the best way I could come to a solution. The output is then displayed to the user. Unit testing was performed along the way for each function.

**Part D:**

Part D takes in a raw input from the user for a string S that is only English letters and whose length is divisible by three, a string C that is only English letters, and an integer N. If the user does not abide by these rules an input error command is returned and the user is asked to try again for the respected input.

This solution still uses the functions length(), make\_string(), make\_array(), shift(), and match() that were created in Part A. There was no need for strings\_equal() in this problem so that function was removed. The functions swap\_strings() and switch\_instances() were edited for the recursive process.

The main problem is finding the indices of C and CSHIFT in the string S so a recursive process was built around this problem. The function recursive\_find\_dispersed\_sub\_string() was created to find each index of each character in a substring that is in a string. These indices are then used to swap C and CSHIFT in the first third or remaining two thirds of S respectively. The function recursive\_find\_dispersed\_sub\_string() takes in a string, substring, a starting value for the recursive counter, a starting value for the indices array, a starting value for the position holder to go through the substring’s characters, and a starting value for a temporary indices array. The base case is that the recursive loop has gone the whole way through the length of the inputted string. The function returns the indices of all of the characters of a substring in a string.

Since the characters of the substring do not need to be consecutive in the string, a temporary indices holder, temp, was created to check if all characters of the substring were found. If they had all been found then they get appended to the indices array. Once appended, the recursion starts back at the character following the first character found of the substring in the string. This takes care of the instance that an instance of the substring is between another dispersed instance of the substring in the string S.

The function swap\_strings() was edited so that now it takes in a string, an integer representing the number of swaps to make, and two index arrays. The function then goes through each instance and swaps the characters at their respective indices.

The switch\_instances() function was created to bring all the above mentioned functions together. We call recursive\_find\_dispersed\_sub\_string() to find all the occurrences of C in the first half of S and all the occurrences of CSHIFT in the remining two thirds of S by returning the indices of each of the substring’s characters in their respective portion of S. These two index arrays are i1 and i2. We then compare the size of the two arrays to determine how many swaps are going to be made between C and CSHIFT. We then call the swap\_strings() function with the string S and two index arrays, i1 and i2. The characters are then swapped, and the output is displayed to the user. Unit testing was performed along the way for each function.