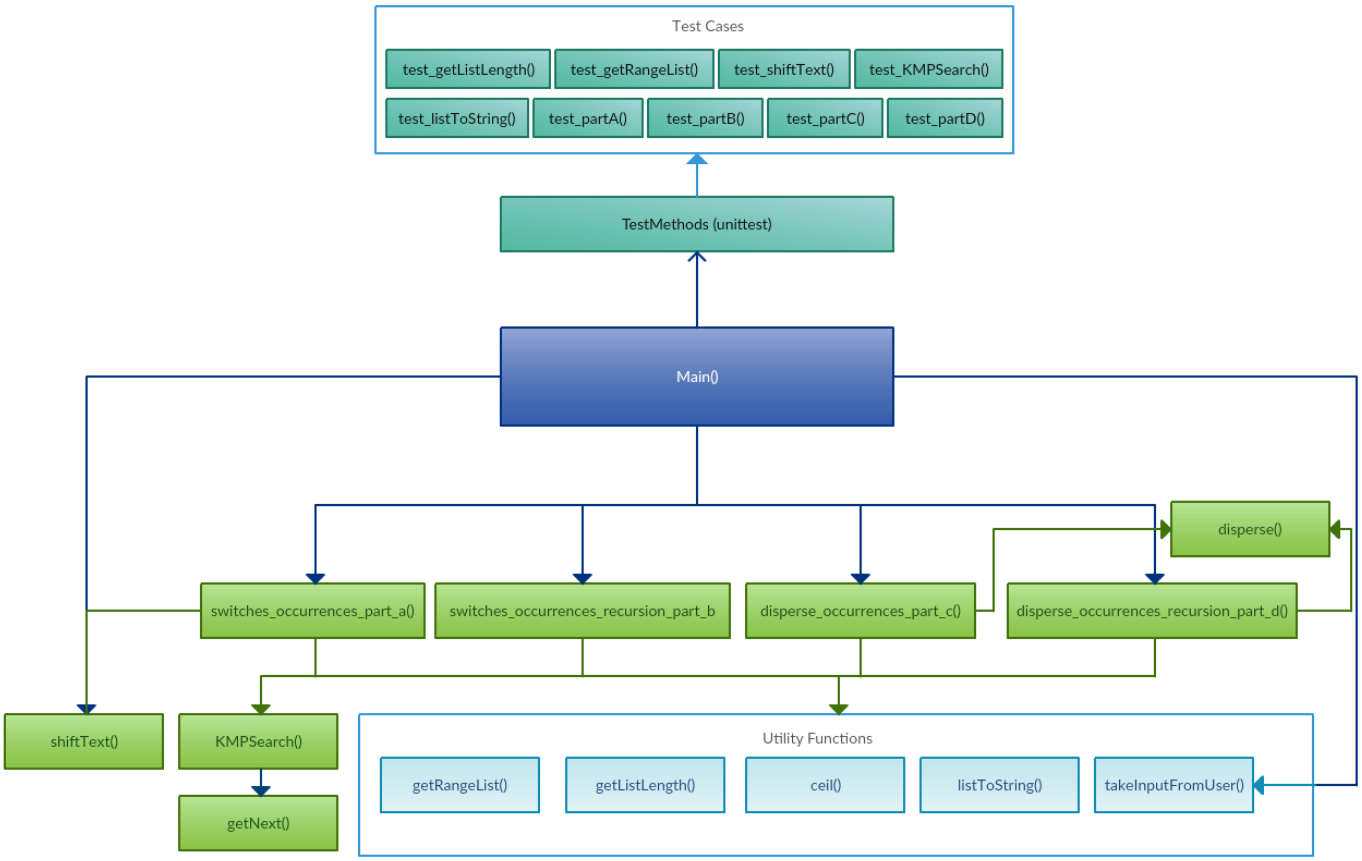
Fidelity Quant Developer Take Home Assessment

Amal Radhakrishnan

06/05/2019

High-level Program Structure



Primary Functions

Solving Part A

switches_occurrences_part_a() -

Main function to solve part A. First I find C_SHIFT by passing C (list) as an argument to the shiftText() function. I subset the first and second third of the S string. I used a while loop to search for C (list) and C_SHIFT occurrences inside first third and second third respectively using KMPSearch(). The while loop runs until there are no more C (list) occurrences inside first third or C_SHIFT inside second third. During each run C (list) occurrence is replaced using C_SHIFT in S_FIRST_THIRD and C_SHIFT occurrence is replaced using C (list) in S_SECOND_THIRD. Return is

triggered when there are no more occurrences. Each time when we replace the substring in S it may create new substring in S that matches C or C_SHIFT. So, the while loop may run multiple times.

Time Complexity to search for substring in S_FIRST_THIRD – O(S_Len*1/3)
Time Complexity to search for substring in S_SECOND_THIRD – O(S_Len*2/3)

Overall Time Complexity - O(S_Len * K)
K – Number of times we have to search through string S so that there are no further occurrences of or C_SHIFT substrings.

shiftText()
A source string and a number to shift the letters is taken as arguments. I have used ascii characters to define the ranges for upper, lower and digit characters. Using a for loop I iterate over each character and yield the character depending on whether the character is a symbol, digit, uppercase or lowercase character. No change on symbols and digits. Rotation on uppercase and lowercase characters.
Time Complexity – O(N)

KMPSearch()

KMP Pattern searching algorithm - A linear time algorithm that solves the string-matching problem by preprocessing P in Θ(m) time. Main idea is to skip some comparisons by using the previous comparison result. whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. I chose KMP because it is more efficient that naïve searching algorithms which tend to perform at O(S*C).

The function KMPSearch() takes input haystack and needle.
haystack = "AAAAABAAABA"
needle = "AAAA"
We compare first window of haystack with needle. In the next step, we compare next window of haystack with needle.

We compare first window of haystack with needle
haystack = "[AAA**A]**ABAAABA"
needle = "[AAA**A]**" [Initial position]
We find a match.

In the next step, we compare next window of haystack with needle
haystack = "AAAAABAAABA"
needle = "AAAA" [Pattern shifted one position]
This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

Time Complexity – O(N)

getNext()

In order to know how many characters to be skipped. We pre-process the pattern and prepare an integer array next[] that tells us the count of characters to be skipped.

For example, given a Pattern String "ABDYAB", we can get next = [-1, 0, 0, 0, 0, 1]
Pattern: "A B D Y A B"
First: "0 0 0 0 1 2"
The substring that begins from the beginning to 'A' has no match of prefix and suffix, so we get 0. Same as 'B', 'D', Y', till 'A'. Now we can see the first 'A' (prefix) matches this 'A'(suffix), and the matched length is 1, So we get 1. We can go all the way to the end. Then: Right shift by one bit and put '-1' on the first element. Thus, we get '-1, 0, 0, 0, 0, 1'

Solving Part B

switches_occurrences_recursion_part_b()

Main function to solve part B. Similar to part A, I find C_SHIFT by passing C (list) as an argument to the shiftText() function. I subset the first and second third of the S string. I used recursion to search for C_List and C_SHIFT occurrences inside first third and second third respectively using KMPSearch(). The exit return condition for the function is that - there is no more C (list) occurrences inside first third and C_SHIFT inside second third. During each run C (list) occurrence is replaced using C_SHIFT in S_FIRST_THIRD and C_SHIFT occurrence is replaced using C (list) in S_SECOND_THIRD. switches_occurrences_recursion_part_b() is triggered as return each time after we replace the substring in S and there are more occurrences to be switched.

shiftText(), KMPSearch() and getNext() functions are subsequently triggered similar to Part A.

Overall Time Complexity - O(S_Len * K)

Solving Part C

disperse_occurrences_part_c()

Main function to solve part B. Similar to part A, I find C_SHIFT by passing C (list) as an argument to the shiftText() function. I subset the first and second third of the S string. I used a while loop to search for C (list) occurrences inside first third using KMPSearch() and search for dispersion indexes for C_SHIFT occurrences inside second third using disperse(). The while loop runs until there are no more C (list) occurrences inside first third or C_SHIFT inside second third. I make sure the length of the return is 3 from disperse because to ensure that a disperse index was found for each character in C_SHIFT. During each run C (list) occurrence is replaced using C_SHIFT in S_FIRST_THIRD and C_SHIFT occurrence is replaced using C (list) in S_SECOND_THIRD. Return is triggered when there are no more occurrences. Each time when we replace the substring in S it may create new substring in S that matches C or C_SHIFT. This is applicable for this part of the problem as well. So, the while loop may run multiple times.

Time Complexity to search for substring in S_FIRST_THIRD – O(S_Len*1/3)
Time Complexity to search for substring in S_SECOND_THIRD – O(S_Len * C_Len * 2/3)

Time Complexity for search – min(S_Len*1/3, S_Len * C_Len * 2/3)

Overall Time Complexity - O(S_Len * K)
K – Number of times we have to search through string S so that there are no further occurrences of or C_SHIFT substrings.

disperse()
This is a custom-built function to search for indexes to disperse C_SHIFT in S_SECOND_THIRD. I initialize an empty index array. I loop through C_SHIFT and run a second loop through S_SECOND_THIRD. Each time there is a match for a C_SHIFT character in S_SECOND_THIRD and given that index is the minimum currently in the index array (to make sure that indexes are incrementally picked for dispersion), I append it to the array. Once all indexes are calculated for each C_SHIFT character, return is triggered.

Worst Case Time Complexity – O(S_Len * C_Len * 2/3)

shiftText(), KMPSearch() and getNext() functions are subsequently triggered similar to Part A.

Solving Part D

disperse_occurrences_recursion_part_d()
Main function to solve part D. Similar to part C, I find C_SHIFT by passing C (list) as an argument to the shiftText() function. I subset the first and second third of the S string. I used recursion to search for C (list) occurrences inside first third using KMPSearch() and search for dispersion indexes for C_SHIFT occurrences inside second third using disperse(). The exit return condition for the function is that - there is no more C (list) occurrences inside first third and full-length C_SHIFT capable to be dispersed inside second third. During each run C (list) occurrence is replaced using C_SHIFT in S_FIRST_THIRD and C_SHIFT occurrence is replaced using C (list) in S_SECOND_THIRD. disperse_occurrences_recursion_part_d() is triggered as return each time after we replace the substring in S and there are more occurrences to be switched.

Overall Time Complexity - O(S_Len * K)

shiftText(), KMPSearch(), disperse() and getNext() functions are subsequently triggered similar to Part C.

Utility Functions

getRangeList()
function recreates the range functionality in python. Using a while loop elements are added into a range array. The element values are incremented based on the step size. The arguments passed into the function are start(int): starting index, end(int): ending index and step(int): step size of the consecutive elements in the range. The function returns: rangeList(list): a list of values within the range and the given step value.

getListLength()
function returns the length of an array. While traversing through each element in the array using a for loop, a running counter is used to increment the size of the array. The arguments passed into the function are myList(list): array for which the length has to be calculated. The function returns: length(int) : length of array.

ceil()
function recreates the math.ceil function in python and calculates the upper ceiling of a float number. The argument passed into the function is num(float): value to find ceil for. The function returns: (int): ceil value of num.

listToString()
function converts a list of string into a single string. The argument passed into the function is myList(list): list of string. The function returns a (string): combined string.

takeInputFromUser()
function to take all inputs from the user. The function is called from main() and returns: S(string): string S, C(string): string C and N(int): number of places to shift.

Test Cases

I have implemented a total of 9 test cases for the utility and primary functions used within the program.

1. test_getListLength(): testing getListLength function
2. test_getRangeList(): testing getRangeList function
3. test_shiftText(): testing shiftText function
4. test_KMPSearch(): testing KMPSearch function
5. test_partA(): testing switches_occurrences_part_a function
6. test_partB(): testing switches_occurrences_recursion_part_b function
7. test_partC(): testing disperse_occurrences_part_c function
8. test_partD(): testing disperse_occurrences_recursion_part_d function
9. test_listToString(): testing listToString function

Output

```
amal:quant_caesar_cipher amal$ python quant_caesar_cipher.py
test_KMPSearch (__main__.TestMethods) ... ok
test_getListLength (__main__.TestMethods) ... ok
test_getRangeList (__main__.TestMethods) ... ok
test_listToString (__main__.TestMethods) ... ok
test_partA (__main__.TestMethods) ... ok
test_partB (__main__.TestMethods) ... ok
test_partC (__main__.TestMethods) ... ok
test_partD (__main__.TestMethods) ... ok
test_shiftText (__main__.TestMethods) ... ok

----------------------------------------------------------------------
Ran 9 tests in 0.001s

OK
S =  ABCXXABCXXBCDXXBCD
C =  ABC
N =  1
C_SHIFT =  BCD
S_PART_A =  BCDXXABCXXABCXXBCD
S_PART_B =  BCDXXABCXXABCXXBCD
```

```
amal:quant_caesar_cipher amal$ python quant_caesar_cipher.py
test_KMPSearch (__main__.TestMethods) ... ok
test_getListLength (__main__.TestMethods) ... ok
test_getRangeList (__main__.TestMethods) ... ok
test_listToString (__main__.TestMethods) ... ok
test_partA (__main__.TestMethods) ... ok
test_partB (__main__.TestMethods) ... ok
test_partC (__main__.TestMethods) ... ok
test_partD (__main__.TestMethods) ... ok
test_shiftText (__main__.TestMethods) ... ok

----------------------------------------------------------------------
Ran 9 tests in 0.001s

OK
S =  ABCXXABCXXBXXCXDXBCD
C =  ABC
N =  1
C_SHIFT =  BCD
S_PART_C =  BCDXXABCXXAXXBXCXBCD
S_PART_D =  BCDXXABCXXAXXBXCXBCD

amal:quant_caesar_cipher amal$ python quant_caesar_cipher.py
test_KMPSearch (__main__.TestMethods) ... ok
test_getListLength (__main__.TestMethods) ... ok
test_getRangeList (__main__.TestMethods) ... ok
test_listToString (__main__.TestMethods) ... ok
test_partA (__main__.TestMethods) ... ok
test_partB (__main__.TestMethods) ... ok
test_partC (__main__.TestMethods) ... ok
test_partD (__main__.TestMethods) ... ok
test_shiftText (__main__.TestMethods) ... ok

----------------------------------------------------------------------
Ran 9 tests in 0.001s

OK
S =  ABCXXABCXXBXXCXDXBCD
C =  ABC
N =  1
C_SHIFT =  BCD
S_PART_A =  BCDXXABCXXBXXCXDXABC
S_PART_B =  BCDXXABCXXBXXCXDXABC
S_PART_C =  BCDXXABCXXAXXBXCXBCD
S_PART_D =  BCDXXABCXXAXXBXCXBCD
```