Group members: Haolin Ye(1931141) and Xiaoming Zhang(1931049)

R. Vincent, instructor

420-LCU-05: Programming Techniques and Application

Final Project

**Manual**

This program is a solution to the third problem of the 2020 Canadian Computing Competition, Senior level. Following the instruction of this problem, “Search for Strings”, the program is able to take two input strings and return an integer that is the times of the distinct permutations of the short string appearing in the long string. If both strings are empty, the program assumes that the user wants to quit. With the Rolling Hash method that is implemented in this program, the program is capable of handling strings that have over ten-thousand lengths, and still prints the correct outcome for the user in less than one second.

For example, if the input value for the short string is “aab” and long string is “abacabaa”, the program will print:“The distinct permutations of the short string appears in the long string 2 times”. This is because the permutations of “aab” are {“aab”, “aba”, “baa”}, and two of them appears in the long string, “abacabaa”. Therefore, the program says the distinct permutations appear twice.

**Design Guide**

This program is related to the concept of string (permutation) searching. All of the coding is done in Python 3 language.

The following are the major steps we take in the program:

1. **Iteration over the long string**

Initially, we implement this using a while loop in python. It executes until the last character of our substring reaches the end of the long string. Later, we change the while loop to a for loop with python’s built-in function, **enumerate()**, because we are going to need both the letters and their indexes for implementing the Rolling Hash algorithm and update our histogram (**Step 2**). More details about the for loop in **Step 6**.

1. **Count the occurrences of each alphabetic letter in the short string and put it in the histogram**

We implement this using a python dictionary, in which we count the number of each lowercase alphabetic letter appearing in the short string and associate the letter(key) with the number(value) in the dictionary. We will also change the implementation later, using a python list called histogram, where each letter matches the index from 0 to 25, and the elements in this list are integer values of the occurrences of each letter.

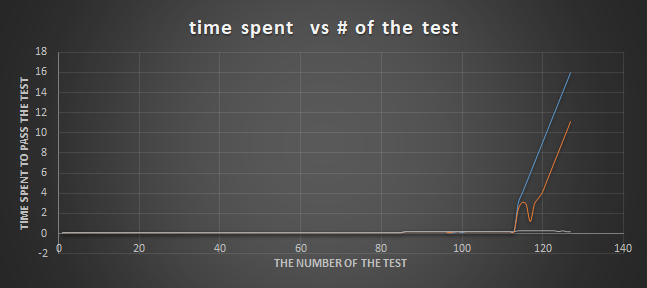
1. **Count the occurrences of each alphabetic letter in a slice of length equal to the short string of the long string and put it in the histogram and comparison**

We initially implement this using a python dictionary, in which we count the number of each lowercase alphabetic letter appearing in the slice using **count()** and associate the letter(key) with the number(value) in the dictionary. We will also change this to a python list called lis. With 2 histograms, histogram and lis, that store the occurrences of letters in the short string and the substring, we can make a comparison between the two and find a match.If we implement this by making a copy of a slice for every time we run the loop, costs O(N) and using python list method count, the time complexity of counting is O(N) where N is the length of the short string. We can improve this by changing that way we construct our histogram. By taking the slice and counting, we have to take O(N) time to iterate over the length of the short string. However, taking the next slice on the long string will only take away the first letter of the old slice and add the next character after the old slice. Thus, we only need to find the first letter of the old slice and remove one occurrence from the histogram and find the next letter after the old slice and add one occurrence to the histogram. This process will only take O(1) time so it would save time when the length of the short string becomes extremely big (ex. 50000).

1. **Find matches and add permutations/matches to a list to keep all found permutations distinct**

We compare two histograms, if they are identical, it means we find a match. Then we append the matched substring onto a list or a set which contains all the found permutations/matches. For each time we execute the loop and take a slice, we would verify if the slice has already appeared in this list. If we are using a set to store all the permutations, this will not be an issue because sets can only contain different elements, everything that is identical will not be added twice in python sets. The purpose is to avoid redundancy in the found permutations since the problem asks to find distinct permutations. However, checking if the substring is in the list takes O(MN) time where N is the length of the short string and M is the length of the list (number of distinct permutations found) because we have to iterate over the list and compare each string of length N with the substring. Later on, we will improve the way we store the permutations, they will be in the form of hash values instead of string, which would save time on string comparison.

1. **Advantage of hash function: what’s put in the answer\_set is shorter, so the comparison is faster.**

If putting all the matches in a set could be time-consuming since it has to check whether this is a distinct permutation, an easier way to store these datas is to only store their hash values. Although both the comparison between permutations and **hash()** depend on the length of the slice, the advantage of using a hash function is that hash values are shorter and easier for python to compare. This makes the comparison between permutations easier for answer\_set. In addition, hashing things is often efficient. Later, we upgrade our hash method and let it only take about constant time to return a hash value which makes it even more efficient. In summary, adding a permutation to answer\_set takes O(MN) time where N is the length of the short string, and M is the length of answer\_set. If replacing the permutations by their hash values, the efficiency would still be O(MN), but the O(N) here should be faster since hashing is efficient, and same thing for O(M) because comparisons between hash values are also faster than comparisons between long strings. Moreover, if we use Rolling Hash instead, this step will only take O(M) time. We have tested the efficiencies of these ideas on the official CCC grading website and made a graph to illustrate our observation. The “time spent” in the graph below means the time required for the program to pass the test, and “# of the test” indicates which test the program is doing. In general, the test that comes later has a larger and harder dataset than the previous ones. The gray line in the graph below represents the efficiency of our final program, O(M), using the Rolling Hash. The blue line is the one that stores all the slices and the orange line is the one that uses **hash()** which matches our prediction. We observe that with a larger dataset, **hash()** is still faster than slice comparison, and the gray one is a horizontal line all the time which proves our prediction.

1. **Advantage of Rolling Hash method: Instead of looking at a slice of the long string, we only look at one letter at a time and update this letter to the old hash value. This makes the efficiency of the Rolling Hash method O(1).**

Moreover, with the hint provided by our professor, we realized that we could upgrade this idea more by replacing the built-in hash function by the Rolling Hash Method. Using **hash()** means that the program needs to hash the entire substring every time we find a match which every letter in this string would contribute to the outcome hash value. Because the hash function takes every letter into account, the efficiency of this depends on the length of the short string. The advantage of the Rolling Hash method compared to **hash()** is that Rolling Hash only needs two arguments which are the indexes of where the substring starts and ends in the long list and after a few quick calculations, the Rolling hash method can return a hash value for this substring. As a result, when we are computing the hash value using Rolling Hash, regardless of the length of the substring, we are only looking at two indexes of it which causes the efficiency of this step to approximately approach a constant time.

1. **Explain the implementation of the Rolling Hash method. (lis, h\_tab, MASK, power, Base value which has to be a prime, x which is a letter-to-index dictionary, histogram containing the occurrence of the letters in the substring, lis which is a histogram for the short string)**

As we iterate over the long string in the main loop, we want to compute the hash value from the first character of the long string to the current character in the loop. Instead of using python’s built-in function **hash()** to compute the hash value of each match, we implement the rolling hash method, where we only add the hash value of the next character in the loop to our old hash value. h\_tab is a list that stores all hash values of the long string, the first element of the h\_tab is 0, meaning the original hash value is 0. As we iterate over the characters, we add the hash value of the next character by retrieving the value in **x**, to the old hash value with the power of 29. This is because 29 is the next prime number of 26, and 26 is the number of lowercase letters(a-z). In the end, we would ampersand everything with the variable MASK, which has the value 2^128 - 1. This step limits our hash value not to exceed MASK, also, mathematically, ampersand 2^128 - 1 is equal to modulo 2^128 (the remainder if divided by 2^128). h\_tab will be useful when we want to retrieve the hash value for a given substring. Once we find a match in two histograms, we want to store the hash value associated with the substring into our answer\_set. Since h\_tab contains the hash value from the first character to i character where i ranges from 0 to length of the long string, our hash value for the current substring will be the hash value at position i + 1 in our h\_tab, where i the index of the last character in our substring, minus the hash value multiplied by the variable **power**((P \*\* (S\_length)) & MASK), at position i + 1 - the length of the short string in our h\_tab. The reason why we implement a variable power is everytime we compute a hash value by adding the next character we would raise it to the power of P. From the first character to the last character in our substring, we raise to the power of P with a number of times that equals to length of the short string.

**Our improvement:**

First version: we implement the python list count() method, and we make a copy of every slice to store in the list containing existing permutations. This passes 96 tests in CCC 2020 Senior Problem 3.

Second version: the only change is we make a copy of every slice but we compute the hash value of them and store in that list. But the improvement of efficiency is not obvious.This passes 114 tests in CCC 2020 Senior Problem 3.

Third version: We replace the count() method with two histograms that only takes O(1) time to compute.This passes 117 tests in CCC 2020 Senior Problem 3.

Final version: Based one the updated version of histogram, we implement the rolling hash method to get the hash values of all matched substrings and store these in that list. This passes all 127 tests in CCC 2020 Senior Problem 3.

**Online sources that helped us to complete this project (basically about how to implement the Rolling Hash Method):**

[**https://courses.csail.mit.edu/6.006/spring11/rec/rec06.pdf**](https://courses.csail.mit.edu/6.006/spring11/rec/rec06.pdf)

[**https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp\_algorithm**](https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm)