CS 599: Algorithms **(Due: November 17th, 2020)**

Submission Homework 7

Instructor: Prof. Prasad Tadepalli *Name:* Rashmi Jadhav*, Student ID:* 934-069-574

1: Implementation of Edit Distance

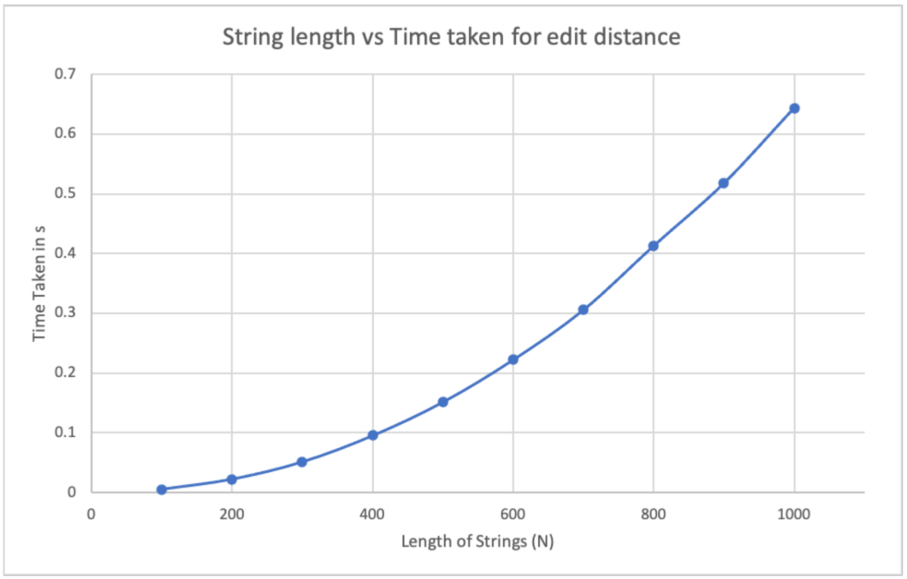
The basic dynamic program algorithm for edit distance and edit string between two input strings was implemented. The time complexity for this algorithm is $O(mn)$ where m and n are the input string lengths. Following are the results for the test cases which successfully execute with the corresponding expected outputs:

```
Edit Distance('BABBLE', 'APPLE') = 3
----------------------------------------------------------------------------------------------------
Edit Distance('ATCAT', 'ATTATC') = 2
----------------------------------------------------------------------------------------------------
Edit Distance('taacttctagtacatacccgggttgagcccccatttcttggttggatgcgaggaacattacgctagaggaacaacaaggtcagag
gcctgttactcctat',
'taacttctagtacatacccgggttgagcccccatttccgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat') = 11
----------------------------------------------------------------------------------------------------
Edit Distance('CGCAATTCTGAAGCGCTGGGGAAGACGGGT', 'TATCCCATCGAACGCCTATTCTAGGAT') = 18
----------------------------------------------------------------------------------------------------
Edit Distance('tatttacccaccacttctcccgttctcgaatcaggaatagactactgcaatcgacgtagggataggaaactccccgagtttccac
agaccgcgcgcgatattgctcgccggcatacagcccttgcgggaaatcggcaaccagttgagtagttcattggcttaagacgctttaagtacttaggatg
gtcgcgtcgtgccaa', 'atggtctccccgcaagataccctaattccttcactctctcacctagagcaccttaacgtgaaagatggctttaggatggca
tagctatgccgtggtgctatgagatcaaacaccgctttctttttagaacgggtcctaatacgacgtgccgtgcacagcattgtaataacactggacgacg
cgggctcggttagtaagtt') = 112
```

10 random pairs of edit strings of length 100, 200, ...,1000 were generated to find the edit distances between them. Following is the output for strings of various lengths for the said string lengths, their edit distances and the time taken in seconds:

```
STRING LENGTH(N)    |EDIT DISTANCE      |TIME TAKEN (in s)
100                 |97                 |0.006190061569213867
200                 |190                |0.023981094360351562
300                 |277                |0.05529904365539551
400                 |369                |0.1020359992980957
500                 |466                |0.17957019805908203
600                 |552                |0.28176307678222656
700                 |635                |0.3348569869995117
800                 |736                |0.44191598892211914
900                 |826                |0.5708110332489014
1000                |907                |0.7120888233184814
```

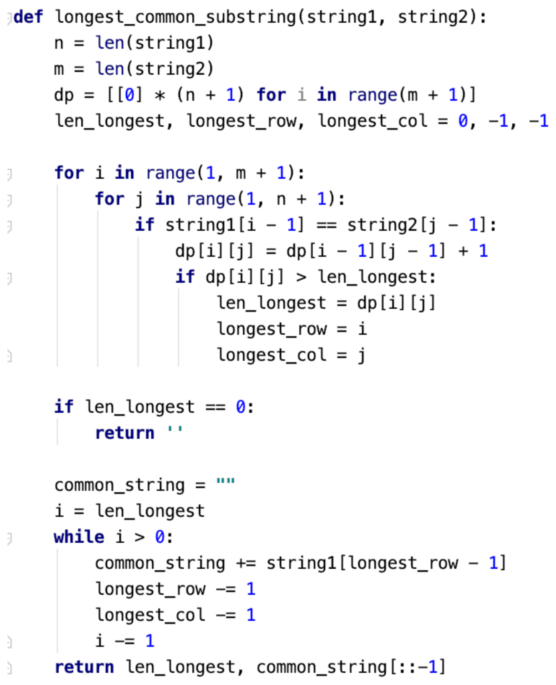
Theoretically, the time complexity for finding the edit distance between to strings of length n each is $O(n^2)$. We now analyze the practical situation. The run times for 100 runs of string lengths $n \in [100, 200, ..., 1000]$ were recorded to arrive at average time performance of the algorithm. Following is the curve that shows the time taken to compute the edit distances as a function of the length of strings. The curve mimics the same behavior as the theoretical results. It exhibits $O(n^2)$ nature.



2: Longest Common Substring

If we were to find the longest common substring between two input strings of lengths m and n in a brute force way, we would generate all possible substrings which would take time complexity of $O(m^2)$ and then for each generated substring, we'd check if this substring is contained in the other string which would take another $O(n)$ time. Thus, the brute force time complexity would be $O(m^2n)$ or $O(mn^2)$.

The brute force approach however has overlapping subproblems and hence we may use dynamic programming to optimize the algorithm further. Instead of finding each substring of one string and checking its existence in the other string, we can store the intermediate common substring lengths and refer to them as we explore substrings that have precomputed stored values. We use a dp array of dimension $m * n$ where $dp[i][j]$ represents length of longest common substring that ends in $string1[i]$ or $string2[j]$. For each $dp[i][j]$ if $i! = j$ then there's no common substring whereas if $i == j$, we can lookup from dp array if $dp[i-1][j-1]$ was already a common substring and then we increment the length of common substring by current character. Whilst filling the matrix, we would keep updating the maximum length found and the dp indexes for the same. To collect the obtained string, we trace length steps backward to form answer output.



```
def longest_common_substring(string1, string2):
    n = len(string1)
    m = len(string2)
    dp = [[0] * (n + 1) for i in range(m + 1)]
    len_longest, longest_row, longest_col = 0, -1, -1

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if string1[i - 1] == string2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                if dp[i][j] > len_longest:
                    len_longest = dp[i][j]
                    longest_row = i
                    longest_col = j

    if len_longest == 0:
        return ''

    common_string = ""
    i = len_longest
    while i > 0:
        common_string += string1[longest_row - 1]
        longest_row -= 1
        longest_col -= 1
        i -= 1
    return len_longest, common_string[::-1]
```

As an extension to this algorithm, if we want to find the longest palindromic substring in a string, we can have an $n * n$ dp array where $dp[i][j]$ represent a string that starts in index i and ends in index j. The value for $dp[i][j]$ would be $true$ if $string[i..j]$ is a palindrome and $false$ otherwise. As a base case, we would have $dp[i][i] = true$ since any string that starts and ends at an index is a string of length 1 which in turn is a palindrome. Strings of length 2 can also be marked $true$ or $false$ depending on whether both characters in string are same or not. For all other strings of length 3 and more, we could use the dp array previous solutions. Say, a string that starts in i and ends in j is a palindrome if the substring $i+1, j-1$ is a palindrome and characters at i and j are same. For example, $aabbaa$ is a palindrome if $abba$ is a palindrome and $a == a$. However, since we start by filling dp array as per strings of lengths $1, 2, ..., i$, we would have the precomputed results for string of length $i-1$. To obtain the longest palindromic substring after filling the entire dp matrix, we can do another sweep through the matrix to find the longest length substring that has $true$ corresponding to it. Wherever we found the largest length, i and j would denote the start and end indices of the longest palindromic substring. The time complexity would thus be $O(n^2)$ and same will be the space complexity.

3: Maximizing profit for BigBucks

Given: Distance array $d_1, d_2, d_3, ..., d_n$; Potential profit for store at corresponding distance in distance array $p_1, p_2, p_3, ..., p_n$; Distance between two shops must be at least k. To find: maximize profit. Consider the following example:

$distances = \{1, 2, 3, 4, 5, 6\}$

$profits = \{9, 15, 9, 6, 14, 8\}$ and $k = 2$

A greedy strategy to maximize the profit would focus on the picking maximum values in the profits array at any point. By that logic, we would pick $p_2 = 15- > p_4 = 6- > p_6 = 8$ because we have an additional constraint of at least k distance between two shops. This would give us a total maximum profit of 29. Let's look at another possible combination of shops. This time we pick $p_5 = 14- > p_3 = 9- > p_1 = 9$. This gives us a profit of 32 which is more than the greedy approach that focused on highest profits. Clearly, the previous case using greedy strategy failed due to the problem of local optima.

Dynamic programming would however help us tackle this problem. We can have a $n+1$ dimensional dp array wherein dp[i] represents maximum profit at distance i (where 0 through i shops are considered) whilst also taking care of the constraint k. At each index, we have an option to either choose that shop or to not choose that shop depending on the constraint. If we choose the shop at index i, we add it's profit value to the previous optimal answer whereas if we do not choose it, we keep the value of previous optimal answer. Whilst doing all these calculations, we make sure to keep track of the constraint. This overall algorithm would take $O(n)$ time complexity as we do one traversal through the distance array and the optimal answer would be present at $dp[n]$.

4: Modified rope cutting problem

Given: Rope of length n with m desired locations of cuts $X_1, X_2, X_3, ..., X_m$. Each cut costs us n units where n is length of the rope length we're cutting.
To find: Optimal sequence of cuts to minimize the total cost
This problem is a variation of 0/1 Knapsack in which the major importance lies in the order of cuts we make. Let X_0 and $X_{(m+1)}$ be the two ends of the rope. In order to arrive at the optimal order, we need to solve the subproblems for first optimal cut, second optimal cut and so on until all cuts have been utilized from input array. We may thus form a Bellman equation for $cost[X_i, X_j]$ which represents the minimum cost of cutting the part of the rope from location X_i to location X_j into $j-i$ pieces in between. This equation can be represented as follows:

$$cost[X_i, X_j] = (j-i) + min\ (cost[X_i, X_k] + cost[X_k, X_j]\ for\ all\ k \in [i, j])$$

We will have a dp array such that $dp[i][j]$ holds the optimal answer $cost[X_i, X_j]$. Eventually, $cost[X_0, X_{(m+1)}]$ would hold the optimal solution for the minimum cost using cuts. Also, if the input $X_1, X_2, X_3, ..., X_m$ is not in a sorted order, we'd have to sort it before proceeding with the dp approach as we need to keep track of the rope length at every instance and at every instance, our $cost[X_i, X_j]$ would depend on smaller subproblems. The time complexity of this algorithm is $O(m^3)$ where m is the length of desired locations of cuts.