Homework Assignment Week 6 and 7: Dynamic Programming

Hyuntaek Oh

ohhyun@oregonstate.edu

CS 514_400 Algorithm

Nov. 18, 2024

1. **(a) Write a Python function called 'editDistance'.**

```python
def editDistance(start, end):
    m = len(start)
    n = len(end)
    D = [[0 for _ in range(n+1)] for _ in range(m+1)]

    for i in range(m+1):
        D[i][0] = i

    for j in range(n+1):
        D[0][j] = j

    for i in range(1, m+1):
        for j in range(1, n+1):
            if start[i-1] == end[j-1]:
                D[i][j] = D[i-1][j-1]
            else:
                insertion = 1 + D[i][j-1]
                deletion = 1 + D[i-1][j]
                replacement = 1 + D[i-1][j-1]
                D[i][j] = min(insertion, deletion, replacement)

    edit_distance = D[-1][-1]

    return edit_distance
```

Fig 1. editDistance function

(b) Generate 10 random pairs of edit strings of length 100, 200, …, 1000 and find their edit distances. Plot the average time taken to compute their edit distances as a function of the length of strings. Comment on the performance of your algorithm.

To generate 10 random pairs of edit strings, string array 'char' is used to add random alphabet while looping until fulfilling the proper length of a word. Then, a copy of the generated word is converted to edit string by inserting, deleting, and replacing some letters in the copy string. The number of conversions is based on the half-length of

words. After that, they become a pair to be input for editDistance function. This processing continues at each length of words such as 100, 200, …, 1000.

The results of using different length of words are below.

Length of words:	Estimated Time O(mn):	Average Time (s):	Result:
100	10000	0.030966591835021973	[44, 46, 42, 41, 40, 42, 42, 45, 41, 37]
200	40000	0.11973328590393066	[84, 82, 81, 76, 83, 86, 82, 81, 85, 84]
300	90000	0.2920659303665161	[121, 131, 122, 123, 120, 123, 124, 125, 122, 127]
400	160000	0.5528881311416626	[170, 176, 160, 169, 171, 153, 165, 168, 163, 160]
500	250000	0.9176635980606079	[211, 207, 206, 204, 208, 201, 212, 199, 204, 197]
600	360000	1.3654386281967164	[244, 249, 255, 257, 251, 236, 249, 250, 243, 248]
700	490000	1.9286601066589355	[295, 291, 289, 290, 293, 288, 296, 291, 290, 293]
800	640000	2.5933363914489744	[327, 328, 330, 328, 339, 332, 320, 321, 345, 330]
900	810000	3.2463579416275024	[379, 361, 379, 363, 372, 376, 365, 376, 382, 373]
1000	1000000	4.109861350059509	[425, 411, 406, 409, 416, 428, 412, 428, 414, 421]

Table. 1 Estimated, average time, and edit distances by words length

Table 1. shows the estimated, average time, and edit distances of using each length of words. The more words' length increases, the longer time takes. The estimated time is based on $O(mn)$ but it actually is $O(n^2)$ since the edit string has same length of generated words. Average time is calculated by 10 for-loops for obtaining accumulative time taken. The result means that how many times the generated words are editted. To clarify the time complexity of the algorithm, we need to use a plot.

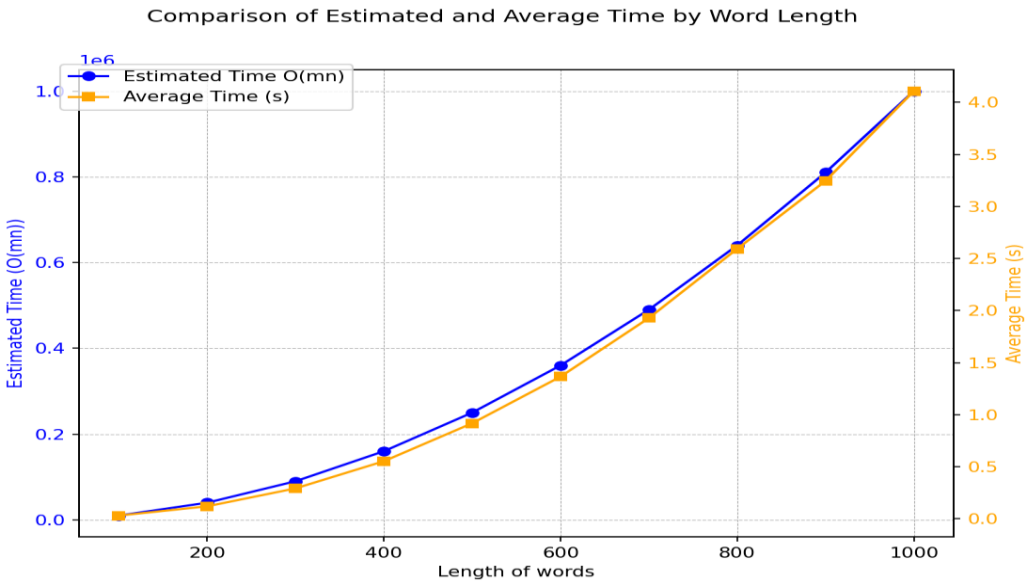

Fig. 2 Comparison of estimated and average time

As shown in Fig. 2, both estimated and average time have similar shape. This suggests that the algorithm works properly as we expected. When the words length increases, the time taken also increases.

Input Size:	Estimated Growth Ratio:	Real Growth Ratio:
200	4.0	3.8665309550958438
300	2.25	2.439304393607459
400	1.7777777777777777	1.8930250798093375
500	1.5625	1.6597636056424614
600	1.44	1.4879511741366194
700	1.3611111111111112	1.412483920427859
800	1.3061224489795917	1.3446311159209248
900	1.265625	1.2518074987617265
1000	1.2345679012345678	1.2659914353126152

Table. 2 Growth ratios of estimated and real cases

The table above shows the growth ratios of estimated and average time cases. This table indicates that the growth ratios of the algorithm has similarity to estimated growth ratios. This trend implies that the first a few growth ratios increase drasically, whereas the others' growth ratios increases smoothly.

2. Give an $O(mn)$ algorithm for finding the longest common substring of two input strings of length m and n.

```
def find_longest_common_substring(str1, str2):
    m, n = len(str1), len(str2)
    c = [[0 for _ in range(n+1)] for _ in range(m+1)]

    max_length = 0
    end_idx = 0

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                c[i][j] = c[i - 1][j - 1] + 1
                if c[i][j] > max_length:
                    max_length = c[i][j]
                    end_idx = i
            else:
                c[i][j] = 0

    longest_substring = str1[end_idx - max_length : end_idx]

    return longest_substring
```

Fig. 3 Longest common substring function

The longest common substring of two input strings should be a substring of two inputs. For example, "magazine" and "magallet" are input strings, so the longest common string of them is "maga". To achieve this, the algorithm uses 2 for-loops to count the

max_length and the index of the end letter, which takes $O(mn)$ time. During these loops, the algorithm checks whether a letter (str1[i-1]) is identical to another letter (str2[j-1]), and then it records the value of end index and max_length if max_length is needed to be updated, otherwise set to zero. After that, the algorithm calculates the length of consecutive letters by using the end index.

3. **BigBucks wants to open a set of coffee shops in the I-5 corridor. The possible locations are at miles $d_1, \ldots, d_n$ in a straight line to the south of their Headquarters. The potential profits are given by $p_1, \ldots, p_n$. The only constraint is that the distance between any two shops must be at least k (a positive integer).**

(1) A counter example of a greedy algorithm failed to choose optimal solution.

Let's assume that a set of the locations 'D' consisting of $d_1, \ldots, d_n$ is $[1, 2, 5, 6, 9]$, corresponding potential profits, $p_1, \ldots, p_n$, are $[16, 20, 15, 17, 12]$, and constraint k is 4 respectively. With greedy algorithm, it chooses $d_2(20)$ and $d_4(17)$ since the algorithm tends to select the largest potential profits. This case yields a total profit of 37. However, the optimal solution should be $d_1(16), d_3(15)$, and $d_5(12)$, which is a maximum total profit of 43. Thus, this example shows that the greedy algorithm above cannot provide an optimal solution to this problem.

(2) An efficient dynamic programming-based algorithm to maximize profit.

```
def open_coffee_shops(d, p, k):
    n = len(d)
    dp = [0] * n

    def find_prev_idx(idx):
        left, right = 0, idx-1
        while left <= right:
            mid = (left + right)//2
            if d[idx] - d[mid] >= k:
                left = mid+1
            else:
                right = mid-1
        return right

    dp[0] = p[0]
    for i in range(1, n):
        prev_idx = find_prev_idx(i)
        if prev_idx != -1:
            dp[i] = max(dp[i-1], dp[prev_idx] + p[i])
        else:
            dp[i] = max(dp[i-1], p[i])

    return dp[-1]
```

Fig. 4 dynamic programming algorithm using binary search

The parameter d is a set of possible locations, p is a set of potential profits, and k is distance constraint. Fig. 4 shows the dynamic programming-based algorithm that

maximizes profit under the given condition k. In this algorithm, binary search function is applied for finding proper index, which takes $O(logn)$. During this process, the algorithm checks the difference between the distance d_i and d_{mid} is equal to or larger than k.

At Initialization step, an array 'dp' initialized to zero and is assigned with $p[0]$. The first for-loop operates non-decreasing order and finds previous dp information at each step. If a previous index is not a value of -1, meaning that there are some accumulative profits, add them with current profit. Otherwise, the algorithm uses only current profit without adding. This process takes $O(n)$ time since it searches every index until reaching the last element.

Thus, the total time complexity of the algorithm is:

$$O(n) * O(logn) = O(nlogn)$$

4. **In a rope cutting problem, cutting a rope of length n into two pieces costs n time units, regardless of the location of the cut. You are given m desired locations of the cuts, $X_1, \dots, X_m$. Give a dynamic programming-based algorithm to find the optimal sequence of cuts to cut the rope into $m + 1$ pieces to minimize the total cost.**

We can solve a rope cutting problem by using Bellman-equation to obtain optimal solutions. The Bellman equation in the rope cutting problem is:

$$Cost[X_i, X_j] = \min_k (Cost[X_i, X_k] + Cost[X_k, X_j]) + (X_j - X_i)$$

It ensures that the immediate cost of making the current cut can be minimized at every step and the future cost of optimally cutting the resulting subsegment.

We apply this equation into the dynamic programming-based algorithm to find the optimal sequence of cuts to cut the rope into $m + 1$ pieces to minimize the total cost. The code is below.

```
def minimum_rope_cutting(n, cuts):
    cuts = [0] + sorted(cuts) + [n]
    m = len(cuts)

    dp = [[0] * m for _ in range(m)]

    for length in range(2, m):
        for i in range(m-length):
            j = i + length
            dp[i][j] = float('inf')
            for k in range(i+1, j):
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + (cuts[j] - cuts[i]))

    return dp[0][m-1]
```

Fig. 5 minimum cost of rope cutting problem with Bellman equation

At initialization step, we add ‘0’ and ‘n’ to the given ‘cuts’ array for setting, and then we use 2-D array called ‘dp’, which initialized to zero, for dynamic programming. There are three for-loops in the code to get the two divided ropes and calculate the future cost with each length of ropes. In particular, the Bellman equation is applied into the third for-loop to calculate the cost. It checks the minimum cost of current dp[i][j] and the immediate costs with k.

The time complexity of this algorithm is $O(n^3)$ since the time complexity can be affected by n, which is the size of the given array, and there are three for-loop to calculate the immediate cost and find minimum cost.