Steven Tung
CSE549: Homework 3

Note: Pseudocode was written in Python style because formatting symbols on Google Docs looks ugly and painful so I just copy-pasted it from my compiler and formatted it accordingly.

Problem 1

Claim 1 - **FALSE**

Counter Example: let X=”ab” and Y=”ba”

Calculate LCS(XX, YY) = LCS(“abab”, “baba”) = 3. The result is 3.
Calculate LCS(X, Y) = LCS(“ab”, “ba”) = 1 and multiply the result by 2. The result is 2.
$3 \neq 2$
Therefore, proved false using a counterexample.

Claim 2 - **FALSE**

Counter Example: let X=”abc” and Y=”cba”

Calculate LCS(X, Y) = LCS(“abc”, “cba”) = 1 and multiply the result by 2. The result is 2.
Calculate ED(X, Y) = ED(“abc”, “cba”) = 2. The result is 2.
Add the previous two answers to get 2+2=4.
Calculate 2|X|, which is 2 times the length of X. The result is 6.
$4 \neq 6$
Therefore, proved false using a counterexample.

Claim 3 - **TRUE**

We know that edit distance is the minimum number of single character insertions, deletions, or substitutions to transform string A into B. The key word here is minimum. So, if we can transform X into Y with ED(X, Y) edits and then transform Y into Z with ED(Y, Z) edits, adding the two transformations gives you a valid but not necessarily minimal sequence of edits to get from X to Z. ED(X, Z) must always be less than or equal to the ED(X, Y) + ED(Y, Z). Subsequently, ED(X, Y) + ED(Y, Z) can never be less than the ED(X, Z).

You can also think of this intuitively. If you go from X to Y and Y to Z and add the number of transformations, it will never be less than going from X straight to Z.

Therefore, this claim is true.

Problem 2

The easiest way to calculate the minimum number of insertions to turn a string into a palindrome string is with the following formula: $minInsertions(S) = |S| - LCS(S, S^R)$ where S^R is the reverse of S. My approach to this problem is simple:

1. Compute S^R
2. Compute $LCS(S, S^R)$
3. Return $|S| - LCS(S, S^R)$

To do this in $O(n^2)$ time and $O(n)$ space, I will utilize dynamic programming to compute the longest common subsequence portion. The following is the LCS recurrence, which I will use to implement the dp pseudocode. Let S and T each be strings of length n. T is the reverse of S. Also, let S[i] be the i-th character of S and T[j] be the j-th character of T.

$$\mathrm{LCS}(S[1..i], T[1..j]) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 1 + \mathrm{LCS}(S[1..i-1], T[1..j-1]) & \text{if } S[i] = T[j], \\ \max(\mathrm{LCS}(S[1..i-1], T[1..j]),\ \mathrm{LCS}(S[1..i], T[1..j-1])) & \text{otherwise.} \end{cases}$$

As the space complexity goal is O(n), I will utilize a one-dimensional dp update array that processes row by row (or column by column). The array dp[0..n], where, at iteration i, dp[j] will store LCS(S[1..i],T[1..j]). The variable prev stores the old value of dp[j-1], which is needed for the if comparison. When double looping is complete, dp[n] stores LCS(S, T). Once the dp part is complete, the final step is to compute |S|-LCS(S, T) and return it.

```
# compute LCS and return the length
def LCS(S, T)
    dp = [0] * (len(T) + 1) # O(n) space
    # O(n^2) because there is a loop for each string (both strings are |S| size)
    for i in range(1, len(S) + 1):
        prev = 0  # dp[j-1] from the previous row
        for j in range(1, len(T) + 1):
            temp = dp[j]  # store current dp[j] before updating
            if S[i - 1] == T[j - 1]:
                dp[j] = prev + 1
            else:
                dp[j] = max(dp[j], dp[j - 1])
            prev = temp
    return dp[len(T)]

# calculate |S|-LCS(S,S^R) which is the miniumum number of insertions we need
def minInsertionsPalindrome(S: str) -> int:
    return len(S) - LCS(S, S[::-1])

print(minInsertionsPalindrome("cabae")) # test case returns 2
```

Correctness Analysis

- LCS calculation idea: the longest palindromic subsequence is equal to $LCS(S, S^R)$ as a palindrome by definition is a string that reads the same forward and backward. Therefore, the idea to use this in the calculation makes sense.
- Minimum insertions formula: when inserting characters into S to make it a palindrome, you have to try to match up as large a palindromic subsequence as possible and fill in the gaps. The smallest number of inserted characters needed to create a palindrome is exactly the complement of the length of the longest palindromic subsequence.
- DP correctness: I use a 1D DP array instead of the typical 2D array. Each update correctly implements the LCS recurrence by storing the needed old-row values in the prev variable instead. This saves space for retrieval.

Complexity Analysis

- **Total Time Complexity**: $O(n^2)$
  - Reversing string S takes $O(n)$
  - LCS computation with 2 for loops of size n each takes $O(n^2)$
- **Total Space Complexity**: $O(n)$
  - Storing dp requires O(n) space array
  - String reversal also takes O(n) space

Problem 3

My general idea is to utilize the LCS function to calculate LIS. First, I will construct a sorted array of the distinct elements in Q called T. This will have no duplicates and will have strictly increasing subsequences. Next, apply the LCS function, which is a black box function that is assumed to be working correctly. You can see that every common subsequence between Q and T is increasing because T is sorted in increasing order. Therefore, the length of the LCS between Q and T is equal to the length of the LIS of Q. I wrote the pseudocode below since it is short and easy to follow.

```
# returns LIS using LCS black box function
# Q is list of nonnegative integers (can repeat)
def LIS(Q):
    # remove duplicates and sort Q
    T = sorted(set(Q))

    # assume LCS is a blackbox function that returns the length of the longest
common subsequence
    # every common subsequence of Q and T is an increasing sequence (check
correctness proof)
```

```
    return LCS(Q, T)

Q = [4, 1, 4, 2, 7]
print(LIS(Q))
```

Correctness Analysis

- We have that any increasing subsequence is a common subsequence. Let L be an increasing subsequence of Q. We know that the elements of L must be in strictly increasing order (definition of LIS). Now, we have T, which is the sorted set of Q. This gives us that L must be a subsequence of T.
- We also have that any common subsequence is increasing. Let L' be a common subsequence of Q and T. We know T is in increasing order, so the order in which the elements of L' appear in T guarantees that L' is in increasing order. Therefore, all common subsequences of Q and T is an increasing subsequence of Q.
- Since every increasing subsequence in Q corresponds to a common subsequence in Q and T, the LCS between Q and T must be the LIS of Q.

Complexity Analysis

- Constructing T: O(nlogn)
    - Removing duplicates is O(n) time
    - Sorting T takes O(nlogn) time in the worst case
- Computing LCS: O(nlogn)
    - LCS is given as a black box function that runs in O((n+r)logn)
    - In the worst case, r = O(n), so it takes O(nlogn)
- **Overall Complexity**: $O(n) + O(nlogn) + O(nlogn) = O(nlogn)$

Problem 4

Part 1

My approach is to store only one current row of length n+1 and a small constant number of variables. When the function call is completed, the last entry of the array holds ED(X, Y). This is very similar to the adaption from the lecture slides, which stores two rows, but my adaptation will store the current row and use variables to store specific values from the other row, which makes it slightly more efficient space-wise.

```
# compute edit distance with O(mn) time and n + O(1) words of extra space
def edit_distance(X, Y):
    m, n = len(X), len(Y)
    row = list(range(n + 1)) # n + 1 space
```

```
    # compute DP values row by row
    for i in range(1, m + 1):
        # diag holds DP[i-1][j-1] for the current j
        diag = row[0]  # DP[i-1][0] initially
        row[0] = i     # cost of deleting all i characters from X

        for j in range(1, n + 1):
            # save the current value of row[j] (DP[i-1][j])
            up = row[j]
            cost = 0 if X[i - 1] == Y[j - 1] else 1

            # compute the new value for DP[i][j]
            row[j] = min(
                up + 1,          # deletion from X (DP[i-1][j] + 1)
                row[j - 1] + 1,  # insertion into X (DP[i][j-1] + 1)
                diag + cost      # substitution (DP[i-1][j-1] + cost)
            )

            # update diag for next iteration, becomes the previous DP[i-1][j]
            diag = up

    # returns edit distance
    return row[n]

# test case returns 3
X = "abcd"
Y = "acdea"
print(edit_distance(X, Y))
```

Correctness Analysis

- Base Case: the algorithm initializes row[j] = j, which represents DP[0][j]. This is the cost of converting an empty string into the first j characters of Y
- Inductive Step:
    - Before processing row i, assume row[j] holds DP[i-1][j] for all j
    - Now for row i, the algorithm sets row[0] = i and uses diag variable to track DP[i-1][j-1]
    - DP[i][j] for each new cell is calculated $min(DP[i-1][j]+1, DP[i][j-1]+1, DP[i-1][j-1]+c)$, which uses the stored values ‘up’ for DP[i-1][j], row[j-1] for DP[i][j-1], and ‘diag’ for DP[i-1][j-1]
    - The above matches the standard recurrence. Therefore, after processing, row[j] contains DP[i][j]
    - After processing all rows, row[n] holds DP[m][n], which is the edit distance of X and Y.

Complexity Analysis

- **Total Time Complexity**: $O(mn)$
  - The outer loop (over i) runs m times
  - The inner loop (over j) runs n times for each i
  - Each iteration does a constant amount of work
  - Therefore, the total running time is O(mn)
- **Total Space Complexity**: n + O(1)
  - row variable is an array of length n + 1
  - Variables diag, up, and cost are single-value variables
  - Total extra space is (n+1) + O(1) = **n + O(1)**

Part 2

The following code is also in the attached Python file.

```
# this function is from Part A. Refer to Part A for comments
def edit_distance(X, Y):
    m, n = len(X), len(Y)
    row = list(range(n + 1))
    for i in range(1, m + 1):
        diag = row[0]
        row[0] = i
        for j in range(1, n + 1):
            up = row[j]
            cost = 0 if X[i - 1] == Y[j - 1] else 1
            row[j] = min(up + 1, row[j - 1] + 1, diag + cost)
            diag = up
    return row[n]

# read from "in.txt"
with open("in.txt", "r") as f:
    T = f.readline().strip()
    q = int(f.readline().strip())
    queries = []
    for _ in range(q):
        line = f.readline().strip()
        i, j, length = map(int, line.split())
        queries.append((i, j, length))

# get each substring and calculate the edit distance
results = []
for i, j, length in queries:
    X = T[i - 1: i - 1 + length]
    Y = T[j - 1: j - 1 + length]
    dist = edit_distance(X, Y)
```

```
    results.append(str(dist))

# write to out.txt
with open("out.txt", "w") as f:
    f.write("\n".join(results) + "\n")
```

Time Complexity: O(mn) as described in Part A
Space Complexity: n + O(1) as described in Part A

The implementation passed the test when compared using diff. All lines matched. A picture of the diff command is attached. No output means no differences.