Assignment 9

Problem 1. Longest common subsequence problem (10 pts)

Given two strings $x^1 \ldots x_n,\ y_1 \ldots y_m$ wish to find the length of their longest common subsequence, that is, the largest k for which there exist indices $i_1 < \ldots < i_k$ and $j_1 <=< j_k$ such that $x_{i1} \ldots x_{ik} \ldots = y_{j1} = y_{jk}$. Show how to do this in time *O(nm)*.

Dynamic programming can be used to find the longest common subsequence (LCS) length between two strings with a time complexity of O(nm), where n and m are the lengths of the input strings.

I can provide you with a clear, step-by-step guide to implementing this algorithm.

1. Initialize a 2D array, dp, with dimensions (n+1) x (m+1), where n and m are the lengths of the input strings. dp[i][j] represents the length of the LCS between the first i characters of the first string and the first j characters of the second string.
2. Iterate over the characters of the first string from left to right (i = 1 to n). For each character in the first string, iterate over the characters of the second string from top.
3. Compare the current characters of the two strings:
    - If the current character in both strings matches (firstString[i-1] == secondString[j-1]), increment dp[i][j] by 1 to calculate the length of the longest common subsequence without considering these characters.
    - If the characters are different, set dp[i][j] = max(dp[i-1][j], dp[i][j-1]). This means we have two options: either the current character from the first string is excluded or the current character from the second string is excluded. We choose the maximum LCS length of these two options.
4. After completing the iteration, dp[n][m] will contain the length of the LCS between the two strings.

Here's a sample implementation in Python:

```
def longest_common_subsequence(str1, str2):
    n = len(str1)
    m = len(str2)
```

```
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[n][m]
```

You can call this function by passing your two strings as arguments, like this:

```
str1 = "ABCD"
str2 = "ACDF"
result = longest_common_subsequence(str1, str2)
print(result)  # Output: 3
```

In the given example, the longest common subsequence between "ABCD" and "ACDF" is "ACD", which has a length of 3.

The time complexity of the modified algorithm to find the length of the longest common subsequence (LCS) in a bottom-up dynamic programming approach, going from right to left, is O(nm), where n and m are the lengths of the input strings. This is because we need to fill in a 2D matrix of size (n+1) x (m+1) to compute the LCS lengths.

The space complexity of the algorithm is also O(nm) because we need to allocate a 2D matrix of the same size to store the LCS lengths. Each cell of the matrix requires constant space, and there are (n+1) x (m+1) cells in total.

Overall, the time and space complexities of the algorithm are both O(nm), which means the algorithm runs in quadratic time with respect to the lengths of the input strings.

Problem 2. Longest subsequence problem with linear space complexity

Find a linear space complexity of your solution for Problem 1.

A space-optimized variation of the dynamic programming algorithm can be used to reduce the space complexity of the solution to Problem 1 to linear.

```
def lcs(x, y):
  n, m = len(x), len(y)
  prev = [0 for _ in range(m + 1)]
  cur = [0 for _ in range(m + 1)]
```

```
  for i in range(1, n + 1):
    for j in range(1, m + 1):
      if x[i - 1] == y[j - 1]:
        cur[j] = prev[j - 1] + 1
      else:
        cur[j] = max(prev[j], cur[j - 1])
    prev, cur = cur, prev
  return prev[m]
```

This algorithm works by maintaining two tables, `prev` and `cur`. The `prev` table stores the values of the table from the previous iteration of the algorithm, and the `cur` table stores the values of the table for the current iteration. The algorithm starts by initializing the `prev` table to all zeros. Then, the algorithm iterates over the rows and columns of the table, starting from the top and working down. For each row, the algorithm compares the corresponding characters in the two strings. If the characters are equal, then the algorithm adds 1 to the value in the `cur` table at that position. Otherwise, the algorithm takes the maximum of the values in the `cur` table at the previous positions.

At the end of the algorithm, the value in the `cur` table at the last column is the length of the longest common subsequence of the two strings.

The space complexity of this algorithm is linear, because it only needs to store two tables, each of which is of size O(m).

Here is an example of how to use the `lcs()` function:

```
x = "ABCD"
y = "ACBD"
print(lcs(x, y))
```

This code will print the value 3, which is the length of the longest common subsequence of the strings `x` and `y`

Problem 3. Longest increasing subsequence problem (10 pts)

Given a string $x_1 \dots x_n$ we wish to find the length of its longest increasing subsequence, that is, the largest k for which there exist indices $i_1 < \dots < i_k$ such that $x_{i_1} < x_{i_2} < \dots < x_{i_k}$. Give an efficient algorithm for this.

Finding the length of the longest increasing subsequence (LIS) in an array is the goal of the Longest Increasing Subsequence issue. Let's handle this problem utilizing dynamic programming.

We can build a dynamic programming array 'dp' of length 'n' to hold the length of the LIS terminating at each index 'i' given an input array 'nums' of length 'n'. The longest growing subsequence that has "nums[i]" as its final element is the LIS that ends at index "i."

The algorithm proceeds as follows:

1. Initialize the `dp` array with all values set to 1 since each element is a valid LIS of length 1.
2. Iterate over the array `nums` from left to right, considering each element `nums[i]`:
    - For each index `j` from 0 to `i-1`, check if `nums[i]` is greater than `nums[j]`. If it is, update `dp[i]` to the maximum between `dp[i]` and `dp[j] + 1`. This means we can extend the LIS ending at `j` by including `nums[i]`.
3. After completing the iteration, the maximum value in the `dp` array represents the length of the longest increasing subsequence in the input array. Return this maximum value as the result.

Here's an example implementation in Python:

```
def longest_increasing_subsequence(nums):
    n = len(nums)
    dp = [1] * n

    for i in range(n):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

You can call this function by passing your input array as an argument, like this:

```
nums = [10, 9, 2, 5, 3, 7, 101, 18]
result = longest_increasing_subsequence(nums)
print(result)  # Output: 4
```

In this example, the longest increasing subsequence in the input array `[10, 9, 2, 5, 3, 7, 101, 18]` is `[2, 5, 7, 101]`, which has a length of 4.

Solution2:

The most efficient way to solve the Longest Increasing Subsequence (LIS) problem is by utilizing the Patience Sorting algorithm, which has a time complexity of O(n log n). This algorithm uses the concept of piles or stacks of cards to determine the length of the LIS.

Here's how the algorithm works:

1. Initialize an empty array `piles` to represent the piles of cards. Each pile will store the top card of that pile.
2. Iterate over the input array `nums` from left to right.
3. For each element `num` in `nums`, do the following:
    - If `num` is greater than the top card of the last pile, create a new pile with `num` as the top card and append it to `piles`.
    - Otherwise, find the leftmost pile `p` in `piles` where the top card is greater than or equal to `num`. Replace the top card of `p` with `num`.

The number of piles in `piles` represents the length of the LIS.

Here's an example implementation of the Patience Sorting algorithm for the LIS problem

```
import bisect

def longest_increasing_subsequence(nums):
    piles = []  # List of top cards of each pile

    for num in nums:
        pile_idx = bisect.bisect_left(piles, num)
        if pile_idx == len(piles):
            piles.append(num)
        else:
            piles[pile_idx] = num

    return len(piles)
```

You can call this function by passing your input array as an argument, like this:

```
nums = [10, 9, 2, 5, 3, 7, 101, 18]
result = longest_increasing_subsequence(nums)
print(result)  # Output: 4
```

In this example, the longest increasing subsequence in the input array `[10, 9, 2, 5, 3, 7, 101, 18]` is `[2, 5, 7, 101]`, which has a length of 4.

The Patience Sorting algorithm is highly efficient, with a time complexity of O(n log n), where n is the length of the input array. It outperforms the dynamic programming approach, especially for larger input sizes.

The space complexity of the Patience Sorting algorithm for the Longest Increasing Subsequence (LIS) problem is O(n), where n is the length of the input array. This space is required to store the piles or stacks of cards. The number of piles at any given point will be at most n, as each pile can have at most one card from the input array.

The time complexity of the Patience Sorting algorithm is O(n log n), where n is the length of the input array. This complexity arises from the use of the `bisect_left` function, which performs a binary search to find the correct position for each element in the piles.

The overall steps of the algorithm involve iterating over the input array once and performing a binary search operation for each element. Since the binary search operation has a time complexity of O(log n), and we perform it n times, the total time complexity becomes O(n log n).

In summary, the Patience Sorting algorithm has a space complexity of O(n) and a time complexity of O(n log n) for solving the Longest Increasing Subsequence problem. It provides an efficient solution for finding the length of the LIS in an array.