Algorithm Design and Analysis

Assignment 4

Deadline: May 26, 2024

1. (25 points) Design a polynomial time algorithm to find the longest palindrome that is a subsequence of a given input string. Please refer to the last slide of Lecture 11 for the definition of palindrome.

   Solution:

   Firstly, there is an easy solotion with brute force algorithm: Taking each character in the string as the starting point, checking each character after it in sequence to determine whether the string starting and ending with these two characters is a palindrome string. The spatial complexity of this algorithm is $O(1)$, and a loose upper bound on time complexity is $O(n^3)$. The optimization method can be to record the length of the currently found longest palindrome string, and then during traversal, only consider substrings with a length greater than the current answer palindrome string.

   We can also solve this problem using dynamic programming methods. Consider the following three situations:

   A string with a length of 1 must be a palindrome string.

   A string of length 2, if these two letters are the same, is a palindrome string.

   For a string with a length greater than 2, if its first and last letters are the same and there is a palindrome string in the middle, then it is a palindrome string.

   We use $P(i, j)$ to indicate whether the string consisting of the i-th to jth letters of the string s is a palindrome string, then

   $P(i, i) = true$

   $P(i, i + 1) = (s_i == s_{i+1})$

   $P(i, j) = P(i + 1, j - 1)\&\&(s_i == s_j)$

   In this way, it is not difficult for us to find the final answer. The spatial complexity of this algorithm is $O(n^2)$, and time complexity is $O(n^2)$.

2. (25 points) In the class, we have seen a dynamic programming algorithm for computing the edit distance between strings of length m and n creates a table of size $n \times m$ and therefore needs $O(mn)$ space. Show how we can reduce it to linear space.

   Solution:

   In the dynamic programming solution of the "edit distance" problem, we can find that in the state transition equation, the value of each element is only related to its left, upper, and upper left values. Therefore, we can optimize the space using the idea of rolling arrays. In each loop, we use a new one-dimensional array to store the value of a new row, and then assign this array to the array representing the previous row. This only requires two one-dimensional arrays, and the spatial complexity is optimized to linear space.

3. (25 points) Two strings $x = x_1x_2 \cdots x_n$ and $y = y_1y_2 \cdots y_m$ are given as inputs.

   (a) Design an $O(mn)$ time algorithm that decides the length of the *longest common substring*, i.e., the largest k for which there are indices i and j with $x_ix_{i+1} \cdots x_{i+k-1} = y_jy_{j+1} \cdots y_{j+k-1}$.

   (b) Design an $O(mn)$ time algorithm that decides the length of the *longest common subsequence*, i.e., the largest k for which there are indices $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$ with $x_{i_1}x_{i_2} \cdots x_{i_k} = y_{j_1}y_{j_2} \cdots y_{j_k}$.

   Solution:

   (a) Define a 2D array dp, where $dp[i][j]$ represents the length of the common substring ending with x_i and y_j. Then we iterate through each character of x and y, if $x_i = y_j$, then $dp[i][j]$ is $dp[i-1][j-1] + 1$; otherwise, $dp[i][j] = 0$. During the iteration, we simultaneously record the maximum value of $dp[i][j]$, which is the length of the longest common substring.

   (b) Assuming the lengths of strings x and y are m and n respectively, create a two-dimensional array dp with $m + 1$ rows and $n + 1$ columns. $dp[i][j]$ is the longest common subsequence length between substrings with index from 0 to $i - 1$ in x and substrings with index from 0 to $j - 1$ in y, then

   $dp[i][j] = 0, i == 0 || j == 0.$

   $dp[i][j] = dp[i-1][j-1] + 1, x[i-1] == y[j-1].$

   $dp[i][j] = max(dp[i-1][j], dp[i][j-1], x[i-1]! = y[j-1].$

   and the answer is $dp[m][n]$.

4. (25 points) In the *subset-sum problem*, you are given a set $T = \{a_1, \ldots, a_n\}$ of n positive integers and a positive integer k as inputs, and you are to decide if there is a subset S with sum exactly k. Notice that a set in this problem may contain multiple copies of an integer.

   (a) Design an $O(kn)$ time algorithm for this problem. Note: This is not a polynomial time algorithm. In fact, as I remarked in the class, the subset-sum problem is a well-known NP-complete problem that we do not believe to be solvable in polynomial time.

   (b) Suppose now you are guaranteed that there exists a subset S with sum exactly k and you are given an extra input parameter $\varepsilon > 0$. Design an algorithm to find a subset S' such that

   $$\sum_{a_i \in S'} a_i \in [(1-\varepsilon)k, (1+\varepsilon)k] .$$

   Your algorithm's running time should be polynomial in terms of $1/\varepsilon$ and n. Prove the correctness of your algorithm, and analyze its running time.

   Solution:

   (a) This is similar to the "knapsack problem" in dynamic programming. Define a 2D array dp, where $dp[i][j]$ represents whether the first i numbers have a subset sum of j. Then, iterate through each number a_i in array T, updating the dp array. Finally, if $dp[n][k]$ is true, it indicates there exists a subset with sum k, otherwise not. The time complexity of this algorithm is $O(kn)$.

   (b) Firstly, sort the numbers in T. Then, use dynamic programming to solve the problem. Define a 2D array dp, where $dp[i][j]$ represents whether the sum of selected numbers from the first i numbers is within the range $[(1-\varepsilon)k, (1+\varepsilon)k]$. Update the dp array using the state transition equation $dp[i][j] = dp[i-1][j]$ or $dp[i-1][j-a_i]$. Finally, backtrack from the dp array to find the subset S' that s

5. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.

   I completed this assignment in two hours.

   I think the difficulty of this assignment is 3, which has decreased compared to the previous ones and is just right.

   Because I have some foundation in doing exercises on LeetCode, these dynamic programming problems are quite familiar to me. For the questions 5, 72, 1143, 416 on LeetCode, I have done these questions before and referred to the answer codes and ideas I have written before when completing this task.