CS301 HOMEWORK 3

Sabanci University

Mert Ziya

27800

QUESTION 1:

A palindrome is a sequence of characters that reads the same backward as forward. In other words, if you reverse the sequence, you get the same sequence. For example, “a”, “bb”, “aba”, “ababa”, “aabbaa” are palindromes. We also have real words which are palindromes, like “noon”, “level”, “rotator”, etc.

We also have considered the concept of a subsequence in our lectures. Given a word A, B is a subsequence of A, if B is obtained from A by deleting some symbols in A. For example, the following are some of the subsequences of the sequence “abbdcacdb”: “da”, “bcd”, “abba”, “abcd”, “bcacb”, “bbccdb”, etc.

The Longest Palindromic Subsequence (LPS) problem is finding the longest subsequences of a string that is also a palindrome. For example, for the sequence “abbdcacdb”, the longest palindromic subsequence is “bdcacdb”, which has length 7. There is no subse- quence of “abbdcacdb” of length 8 which is a palindrome.

One can find the length of LPS of a given sequence by using dynamic programming. As common in dynamic programming, the solution is based on a recurrence.

Given a sequence $A = a_1a_2...a_n, letA[i, j]$ denote the sequence $a_ia_i + 1...a_j$. Hence it is part of the sequence that starts with ai and ends with aj (including these symbols). For example, if A = abcdef , A[2, 4] = bcd, A[1, 5] = abcde, A[3, 4] = cd, etc.

For a sequence $A = a_1a_2...a_n$, let us use the function $L[i, j]$ to denote the length of the longest palindromic subsequence in $A[i, j]$.

(a) If we have a sequence $A = a_1a_2...a_n$, for which values of i and j, $L[i, j]$ would refer to the length of the longest palindromic subsequence in A?

For a sequence $A = a_1a_2...a_n$, $L[i, j]$ would refer to the length of the longest palindromic subsequence in A when i = 1 and j = n. This is because $A[1, n]$ represents the entire sequence, including all characters from a_1 to a_n. In the context of the dynamic programming table L, the value L[1][n] contains the length of the longest palindromic subsequence of the original sequence A.

(b) Write the recurrence for L[i, j].

$$L(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ L(i+1, j-1) + 2 & \text{if } i < j, a_i = a_j \\ \max(L(i+1, j), L(i, j-1)) & \text{if } i < j, a_i \neq a_j \end{cases}$$

(c) What would be the worst case time complexity of the algorithm in Θ notation? Why?

The worst-case time complexity of the LPS algorithm implemented with dynamic programming is $\Theta(n^2)$, where n is the length of the input sequence.

The reason for this complexity is that the algorithm iterates through all possible subsequences of A, which are defined by pairs of indices (i, j) such that $1 \leq i \leq j \leq n$. In other words, we are looking at all possible combinations of the start and end indices of the subsequences. There are roughly $n^2/2$ such pairs, which is proportional to n^2.

For each pair of indices (i, j), the algorithm performs a constant amount of work, either comparing characters or taking the maximum of two values. As a result, the overall time complexity is proportional to the number of pairs (i, j), which is $\Theta(n^2)$.

QUESTION 2:

Consider the 0–1 knapsack problem, where we have a set of n objects $(o_1, o_2, ..., o_n)$, each with a weight $(w_1, w_2, ..., w_n)$ and a value $(v_1, v_2, ..., v_n)$. Here, the object o_i has the weight w_i and the value v_i. Suppose that W is the capacity of our knapsack. We would like to compute the maximum value that we can pack into our backpack. Let $P[i, j]$, denote the maximum value that can be packed into a knapsack with capacity j, if we consider only the first i objects, i.e. $o_1, o_2, ..., o_i$.

(a) For which values of i and j, P[i,j] would refer to the maximum value that can be packed into our knapsack of capacity W if we consider all n items?

To find the maximum value that can be packed into the knapsack of capacity W considering all n items, you would look at the value P[n, W]. This is because i refers to the number of objects considered, and j refers to the capacity of the knapsack.

So, P[n, W] $\underline{(i = n, j = W)}$ refers to the maximum value obtained when considering all n objects (o1, o2, ..., on) and using the full capacity W of the knapsack.

(b) Write the recurrence for P [i, j].

$$P(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ P[i-1, j] & \text{if } i > 0, j < w_i \\ \max(P[i-1, j], P[i-1, j-w_i] + v_i) & \text{if } i > 0, j \geq w_i \end{cases}$$

(c) What would be the worst case time complexity of the algorithm in Θ notation? Why?

The worst-case time complexity of the dynamic programming algorithm for the 0-1 knapsack problem is $\Theta(n * W)$, where n is the number of items and W is the capacity of the knapsack. This is because we have to fill a table of size n x W in order to compute the optimal solution. And this is because;

- We have n rows in the table, one for each object (1 to n).
- We have W columns in the table, one for each capacity from 0 to W.
- For each entry P[i, j], we compute its value based on the recurrence relation, which takes constant time (O(1)).
- Therefore, we have n x W entries to compute, and each computation takes constant time.

QUESTION 3:

Consider again the 0–1 Knapsack problem. This time, instead of developing a dynamic programming solution, we would like to suggest a greedy solution.

If we have a set of n objects $(o_1, o_2, ..., o_n)$, each with a weight $(w_1, w_2, ..., w_n)$ and a value $(v_1, v_2, ..., v_n)$, and if we have a capacity W:

(a) What would be a greedy choice to pick the object to be included in our knapsack at this point?

I would choose the the object based on their value-to-weight ratio. (i would select the items with the highest value per unit weight). I would follow these steps:

1. For each object o_i, calculate its value-to-weight ratio: $r_i = v_i/w_i$.
2. Sort the objects in decreasing order of their value-to-weight ratios (r_i).
3. Initialize the knapsack's total value (V) to 0 and remaining capacity (C) to W.
4. Iterate through the sorted list of objects:
   (a) If the current object's weight (wi) is less than or equal to the remaining capacity (C):
      i. Add the object to the knapsack.
      ii. Update the total value (V = V + vi) and the remaining capacity (C = C - wi).
   (b) If the remaining capacity is not enough for the current object, move on to the next object.
5. Return the total value (V) of the knapsack.

(b) If we pick the object oi by the greedy choice in part (a), what would be the subproblem that we will be left with, after this choice of object?

If we pick the object o_i using the greedy choice in part a, the subproblem we will be left with is a reduced version of the original problem with the following changes:

1. The set of objects to consider is now reduced by one. For example, $o_1, o_2, ..., o_{i-1}, o_{i+1}, ..., o_n$ (object o_i has been removed).
2. The capacity of the knapsack has been decreased by the weight of the chosen object, For example, the new capacity is $W' = W - w_i$ (where W is the original capacity and w_i is the weight of the chosen object).

Now, we need to find the maximum value that can be packed into the knapsack with the new capacity W' and the remaining objects. The algorithm at part a would continue with these updated parameters to make subsequent choices.