CS 624 - ANALYSIS OF ALGORITHMS

Assignment 3

Aravind Haridas – 02071139

1. Problem 15.4-5 on p397, modified.

Give an O(n 2)-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Do not write out the algorithm. Instead, do the following:

(a) We will start with a related problem: Given a sequence X[1 .. n], what is the longest increasing subsequence of X[1 .. n] that ends with X[n]? I'll call this the LISE problem. State the optimal substructure property for the LISE problem. You are not required to prove it.

(b) Define a recursive function LISE(k) that returns the length of the longest monotonically increasing subsequence for X[1 .. k] that ends in X[k]. Assume that the original sequence of numbers X is fixed/global.

(c) Briefly explain how to solve the original problem by adapting the LISE-solving algorithm.

(d) Briefly explain the running time of the algorithm, assuming memoization. Note that there are other ways of solving the original problem. You should think about that, but for the homework you are required to use the structure above.

Sol : Longest Monotonically Increasing Subsequence

(a) The optimal substructure property for the LISE (Longest Increasing Subsequence Ending at *X[n])* problem states that the longest increasing subsequence ending at *X[n]* can be obtained by appending *X[n]* to the longest increasing subsequence ending at any position $i < n$ where $X[i] < X[n]$. In other words, if we know the length of the longest increasing subsequence ending at each position $i < n$, we can determine the length of the longest increasing subsequence ending at position n.

Let's assume we have the longest increasing subsequence ending at position k, denoted as *LIS(k),* and its length is denoted as *len(k).* Then, the length of the longest increasing subsequence ending at position $k + 1$, denoted as *LIS(k+1),* can be obtained as follows:

If *X[k+1]* is greater than all elements before it, then $LIS(k+1) = 1 + max\{LIS(1), LIS(2), ..., LIS(k)\}$, because we can append *X[k+1]* to any of the previous longest increasing subsequence.

If *X[k+1]* is not greater than all elements before it, then $LIS(k+1) = 1 + max\{LIS(i)\}$ where $1 \le i \le k$ *and* $X[i] < X[k+1]$, because we can append *X[k+1]* to the longest increasing subsequence ending at any position $i < k+1$ where $X[i] < X[k+1]$.

we can see that the length of the longest increasing subsequence ending at position $k+1$ depends on the lengths of longest increasing subsequence ending at positions before it, satisfying the optimal substructure property.

(b) A recursive function *LISE(k)* that returns the length of the longest monotonically increasing subsequence for *X[1 .. k]* that ends in *X[k].*

The function LISE(k) is defined as follows:

If $k = 1$, then $LISE(k) = 1$ because there is only one element, and it forms a monotonically increasing subsequence of length 1.

Otherwise, for each position $i < k$, *if* $X[i] < X[k]$, then the length of the longest monotonically increasing subsequence ending in $X[k]$ can be formed by appending $X[k]$ to the longest monotonically increasing subsequence ending in $X[i]$. Therefore, $LISE(k) = 1 + max\{LISE(i)\}$ *where* $1 \leq i < k$ *and* $X[i] < X[k]$.

If there are no such positions $i < k$ where $X[i] < X[k]$, then $LISE(k) = 1$, as $X[k]$ itself forms a monotonically increasing subsequence of length 1.

This recursive function effectively breaks down the problem into smaller subproblems and combines the solutions to these subproblems to obtain the solution for the larger problem, satisfying the optimal substructure property.

(c) The LISE-solving algorithm for finding the longest monotonically increasing subsequence of a sequence of n numbers, you can follow these steps:

Create an array lengths of size n to store the length of the longest monotonically increasing subsequence ending at each position.

Iterate through each position k in the sequence from 1 to n.

For each position k, initialize lengths$[k]$ *to* 1 (as a single element forms a subsequence of length 1).

Iterate through each position $i < k$.

If $X[i] < X[k]$, update *lengths*[k] to the maximum of *lengths*$[k]$ and *lengths*$[i] + 1$, meaning we can extend the longest increasing subsequence ending at position i by appending $X[k]$ to it.

After iterating through all positions, return the maximum value in the lengths array, which represents the length of the longest monotonically increasing subsequence in the entire sequence.

```
function LongestIncreasingSubsequence(X[1..n]):
    lengths[1..n] //// Array to store the lengths of longest increasing subsequence
    for k from 1 to n:
        lengths[k] = 1 ////  Initialize the length at position k
        for i from 1 to k - 1:
            if X[i] < X[k]:
                lengths[k] = max(lengths[k], lengths[i] + 1)
    return max(lengths) //// Return the maximum length found
```

(d) The running time of the algorithm, assuming memorization, is $\boldsymbol{O(n^2)}$. Each position k requires checking all previous positions $i < k$, resulting in a time complexity of $O(n)$ per position. With memorization, we store the results of subproblems, ensuring each subproblem is solved only once, leading to an overall time complexity of $\boldsymbol{O(n^2)}$.

CS 624 - ANALYSIS OF ALGORITHMS

2. Problem 15-2 on p405, modified.

A palindrome is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, racecar, and aibohphobia (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input character, your algorithm should return carac. What is the running time of your algorithm?

Do not write out the algorithm. Instead, do the following:

(a) State the optimal substructure property for the longest palindrome subsequence problem. You are not required to prove it.

(b) Define a recursive function LPS(??) which returns the length of the longest palindrome subsequence for a given subproblem. Assume the original string is fixed/global. You must decide what arguments identify each subproblem. Briefly explain the meaning of the arguments.

(c) Briefly explain the running time of your algorithm, assuming memorization.

(a) Optimal Substructure Property for LPS: The optimal substructure property states that if the first and last characters of a string are the same, they can be part of the longest palindromic subsequence (LPS), and the problem then reduces to finding the LPS of the substring that excludes these two characters. Conversely, if the first and last characters of the string are not the same, then the LPS of the string is the longer of the LPS of the string without the first character and the LPS of the string without the last character.

The length of the longest palindrome subsequence within a substring *S[i….j]* is the maximum of: 1 if *i = j* (single character), 2 + length of longest palindrome in *S[i+1..j-1] if S[i] = S[j],* or the maximum of longest palindrome in *S[i+1..j]* and *S[i..j-1] if S[i] != S[j].*

(b) Recursive Function LPS

The recursive function *'LPS(i, j)'* can be defined where '*i*' and '*j*' are the starting and ending indices of the substring in consideration within the original string '*S*' of length '*n*':

```
LPS(i, j):
  if i > j:
    return 0
  if i == j:
    return 1
  if S[i] == S[j]:
    return 2 + LPS(i+1, j-1)
```

```
else:
  return max(LPS(i+1, j), LPS(i, j-1))
```

'S' is the fixed/global original string.

'I' and 'j' define the boundaries of the subproblem, indicating the substring *'S[i…..j]'* under consideration.

(c) Running Time Analysis with Memorization: The running time of the LPS algorithm can be analysed as follows:

- The algorithm examines pairs of indices *'(i, j)'*, with *'i'* ranging from '*0'* to '*n-1'* and *'j'* from *'i'* to *'n-1'*.
- There are ***O(n^2)*** unique subproblems since '*i*' and '*j*' can each take '*n*' possible values.
- Each subproblem takes constant time ***'O(1)'*** because we're just doing a comparison and taking a maximum of two previously solved subproblems.
- Hence, the total running time with memorization is ***O(n^2)*** because we store the result of each subproblem and avoid redundant computations by reusing the stored results.

CS 624 - ANALYSIS OF ALGORITHMS

3. Problem 16.2-2 on p427, modified.

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in O(nW) time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack. Do not write out the algorithm. Instead, do the following:

(a) State the optimal substructure property for the 0-1 knapsack problem. You are not required to prove it.

(b) Define a recursive function KS(??) which returns the maximum total value of items in the knapsack for a given subproblem. Briefly explain what parts of the original problem fixed/global and what parts are are subproblem-specific.

(c) Briefly explain the running time of your algorithm, assuming memoization.

(a) Optimal Substructure Property:

The maximum value obtainable with n items and weight limit W is the maximum of either taking the nth item (if it fits) or not taking it, recursively applying this decision to the remaining items and adjusted weight limit.

. The maximum value obtained by excluding the current item *i*

(i.e., solving the problem for I - {i} and weight W), or

. The value of the current item *i* plus the maximum value obtained by solving the problem for I - {i} and the reduced weight W - weight(i), if including the item doesn't exceed the weight limit.

(b) Recursive Function KS(n, W)

The recursive function KS(n, W) can be defined as follows, where n is the number of items and W is the maximum weight capacity of the knapsack:

```
KS(n, W):
  if n == 0 or W == 0:
    return 0
  if weight[n] > W:
    return KS(n-1, W)
  else:
    return max(KS(n-1, W), value[n] + KS(n-1, W - weight[n]))
```

In this function:

The global/fixed parts of the problem are the arrays *value[]* and *weight[]*, which represent the value and weight of each item respectively, and the capacity *W* of the knapsack.

The subproblem-specific parts are '*n*', the current index in the arrays being considered, and the current remaining weight capacity '*W*'.

(c) Running Time with Memorization: Assuming memorization is used, the running time of the dynamic programming solution for the 0-1 Knapsack problem is *O(nW).*

This is because:

For each of the *'n'* items, the function *'KS( )'* computes the maximum value for every integral weight capacity from *'0'* to *'W'*. Thus, the subproblems form a two-dimensional table of size *n x W*.

Each entry in the table takes *O(1)* time to compute since it involves a comparison operation (max) and subtraction.

Since there are *n * W* subproblems, and each take *'O(1)'* time, the total time is *O(nW).*

Memorization ensures that each subproblem is solved only once, and the results are stored and reused, leading to significant time savings over the naive recursive approach, which would take exponential time in the absence of memorization. The dynamic programming approach fills in a table of size *'n x W'*, computing optimal solutions for progressively larger knapsacks using the optimal solutions to smaller subproblems.

CS 624 - ANALYSIS OF ALGORITHMS

4. The Fibonacci numbers are defined by the following recurrence:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_{n+2} = F_{n+1} + F_n$$

(a) Write pseudocode for a naive recursive function that computes the n th Fibonacci number.

Naive Recursive Function

The naive recursive approach directly follows the Fibonacci sequence's definition, where each term is the sum of the preceding two terms, without any optimizations for repeated calculations.

```
FUNCTION Fibonacci(n)
    IF n <= 0
        RETURN 0
    ELSE IF n == 1
        RETURN 1
    ELSE
        RETURN Fibonacci(n-1) + Fibonacci(n-2)
END FUNCTION
```

(b) Write pseudocode for a top-down memorized version of the same function. Your main function should still take one integer argument and produce one integer result; it should use a memorized helper function.

Top-Down Memorized

Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again, avoiding the need to recompute them.

```
FUNCTION FibonacciMemoized(n, memo)
    IF memo[n] >= 0
        RETURN memo[n]
    IF n <= 0
        memo[n] = 0
    ELSE IF n == 1
        memo[n] = 1
    ELSE
        memo[n] = FibonacciMemoized(n-1, memo) + FibonacciMemoized(n-2, memo)
```

```
    RETURN memo[n]
END FUNCTION
FUNCTION Fibonacci(n)
    memo = ARRAY OF n+1 ELEMENTS, INITIALIZED TO -1
    RETURN FibonacciMemoized(n, memo)
END FUNCTION
```

(c) Write pseudocode for a bottom-up memorized version of the Fibonacci function. Your main function should still take one integer argument and produce one integer result.

Tabulation is the opposite of memorization. It solves the problem "bottom-up" (i.e., by solving all related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the result, the solution to the top/original problem is then computed.

```
FUNCTION FibonacciTabulation(n)
    IF n <= 0
        RETURN 0
    memo = ARRAY OF n+1 ELEMENTS
    memo[0] = 0
    memo[1] = 1
    FOR i FROM 2 TO n
        memo[i] = memo[i-1] + memo[i-2]
    RETURN memo[n]
END FUNCTION
```