Assignment 6

- **Solve problems 15.3-4, 15.4-5, 15.4-6 in CLRS.**
  - ☐ **15.3-4**

    As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix Ak at which to split the subproduct AiAiC1 Aj (by selecting k to minimize the quantity pi1pkpj) before solving the subproblems. Find an instance of the matrix-chain multiplica-tion problem for which this greedy approach yields a suboptimal solution.

    Professor Capulet's suggested greedy approach yields a suboptimal solution for the matrix-chain multiplication problem,let's take the following matrices with the given dimensions: A1: 10x30 A2: 30x5
    A3: 5x60 A4: 60x25

    Greedy approach:

    1. Choose k=2 to split the product A1A2C1A3A4.
    Calculate p(1, 2, 4) = p(1, 2) + p(2, 4) + p(1) * p(2) * p(4) = (10x30x5) + (30x5x25) + (10x5x25) = 1500 + 3750 + 1250 = 6500 scalar multiplications.

    2. Calculate the remaining subproblem p(1, 3, 4):
    Calculate p(1, 3, 4) = p(1, 3) + p(3, 4) + p(1) * p(3) * p(4) = (10x30x60) + (30x60x25) + (10x60x25) = 18000 + 45000 + 15000 = 78000 scalar multiplications.

    The total number of scalar multiplications for this approach is 6500 (for the split at k=2) + 78000 (for the remaining subproblem) = 84500 scalar multiplications.

    However, there is a better order that reduces the total number of scalar multiplications:

    1. Choose k=1 to split the product A1C1A2A3A4.
    Calculate p(1, 1, 4) = p(1, 1) + p(1, 4) + p(1) * p(1) * p(4) = 0 + 6000 + 0 = 6000 scalar multiplications.

2. Calculate the remaining subproblem p(2, 3, 4):
Calculate p(2, 3, 4) = p(2, 3) + p(3, 4) + p(2) * p(3) * p(4) = (30x5x60) + (5x60x25) + (30x5x25) = 9000 + 7500 + 3750 = 20250 scalar multiplications.

The total number of scalar multiplications for this better order is 6000 (for the split at k=1) + 20250 (for the remaining subproblem) = 26250 scalar multiplications.

In this example, Professor Capulet's suggested greedy approach leads to a suboptimal solution, and there exists a different order that results in a significantly lower number of scalar multiplications.

☐ **15.4-5**

Give an O.n2/-time algorithm to find the longest monotonically increasing subse-quence of a sequence of n numbers.
Code:-

```
def find_lis(sequence):
    n = len(sequence)
    lis_length = [1] * n

    for i in range(1, n):
        for j in range(0, i):
            if sequence[i] > sequence[j] and lis_length[i] <= lis_length[j] + 1:lis_length[i]
                = lis_length[j] + 1

    max_length = max(lis_length)
    longest_lis = []

    for i in range(n - 1, -1, -1):
        if lis_length[i] == max_length:
            longest_lis.insert(0, sequence[i])
            max_length -= 1

    return longest_lis

sequence = [10, 22, 9, 33, 21, 50, 41, 60, 80]
result = find_lis(sequence)
print(result)
```

Output: [10, 22, 33, 50, 60, 80]

To find the longest monotonically increasing subsequence of a sequence of n numbers in O(n^2) time, we can use dynamic programming.In this above algorithm,it has a time complexity of O(n^2) because it uses a nested loop structure to compute the lengths of the longest increasing subsequences ending at each index.

☐ **15.4-6**

Give an O.n lg n/-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (Hint: Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a can-didate subsequence of length i 1. Maintain candidate subsequences by linking them through the input sequence.)

Code:-

```
def find_lis(sequence):
    n = len(sequence)
    tails = [0] * n
    indices = [0] * n
    prev_indices = [None] * n
    length = 1
    for i, num in enumerate(sequence):
        left, right = 0, length
        while left < right:
            mid = (left + right) // 2
            if tails[mid] < num:
                left = mid + 1
            else:
                right = mid

        idx = left
        tails[idx] = num
        indices[idx] = i
        if idx > 0:
            prev_indices[i] = indices[idx - 1]if
        idx == length:
            length += 1

    result = []
```

```
    idx = indices[length - 1]
    while idx is not None:
        result.insert(0, sequence[idx])
        idx = prev_indices[idx]

    return result

sequence = [10, 22, 9, 33, 21, 50, 41, 60, 80]
result = find_lis(sequence)
print(result)
```

Output: [10, 22, 33, 50, 60, 80]

To find the longest monotonically increasing subsequence of a sequence of n numbers in O(n log n) time using the hinted algorithm, we can use a variation of sorting algorithm.This algorithm has a time complexity of O(n log n) because the binary search operation inside the loop dominates the overall time complexity, and the loop itself runs in O(n).