ASSIGNMENT – 4

Introduction to Algorithms (CS 430)

1. The approach of always splitting the matrix chain at k such that p_k is the smallest among $\{p_i, ..., p_j – 1\}$ does not always yield the optimal matrix-chain multiplication cost.

Pseudo-Code:

```
Function MatrixChainOrder(p, i, j)
    if i == j
        return 0
    min_cost = ∞
    for k from i to j-1
        if p[k] is the smallest in { p[i], ..., p[j-1] }
            cost = MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k+1, j) + p[i-1] * p[k] * p[j]
            if cost < min_cost
                min_cost = cost
    return min_cost
```

Counter Example:

Consider three matrices A_1 , A_2 , A_3

- A_1 of dimension 10 × 100
- A_2 of dimension 100 × 5
- A_3 of dimension 5 × 50

Matrix Dimensions:

- $p_0 = 10$
- $p_1 = 100$
- $p_2 = 5$
- $p_3 = 50$

According to the algorithm, we should split the chain at k such that p_k is the smallest among $\{p_i, ..., p_j - 1\}$. For $A_1 \times A_2 \times A_3$

1. $p_1 = 100$
2. $p_2 = 5$

The smallest value is $p_2 = 5$, so we split the chain at k = 2.

Splitting at k = 2

1. First compute $(A_2 \times A_3)$
   - Cost = 100 × 5 × 50 = 25000
2. Then multiply the result with A_1
   - Cost = 10 × 100 × 50= 50000

Total Cost = 25000 + 50000 = 75000

Optimal Solution
To find the optimal solution we check both possible ways of multiplying the matrices.

Splitting at k = 1

1. First compute ($A_1 \times A_2$)
   - Cost = 10 × 100 × 5 = 5000
2. Then multiply the result with A3
   - Cost = 10 × 5 × 50 = 2500

Total Cost = 5000 + 2500 = 7500

Therefore, the given algorithm splits the matrices in a way that results in a total cost of 75000, while the optimal solution has a cost of 7500. This counterexample demonstrates that the algorithm of always splitting at the smallest p_k does not always yield the optimal solution for matrix-chain multiplication.

2.
a)
If N is an odd number:
When N is odd, it is impossible to find a subset B that satisfies the equation. This is because the sum of all elements is odd (N) and splitting an odd sum into two equal parts is impossible. Therefore, when N is odd, we can say that there is no solution to the problem.

b)
A recursive function *b(i, j)* that represents the Boolean value whether there is a subset in *{a1, a2, …, ai}* that add up to *j*.

1. Base Cases:
   - b(0,0) = true
   - b(0,j) = false for j > 0
2. Recursive Cases:
   - b(i,j) = b(i - 1, j) OR b(i-1, j - ai) if j >= ai
   - b(i, j) = b(i - 1, j) if j < ai

c)

a. We can use a 2D Boolean array DP[0..k][0..N/2] where DP[i][j] represents b(i,j).

b. **Pseudo-code**

```
function subsetSum(a[], k, N):
    if N is odd:
        return false
    target = N / 2
    DP = boolean array of size [k+1][target+1], initialized to false
    for i = 0 to k:
        DP[i][0] = true
```

```
for i = 1 to k:
    for j = 1 to target:
        if a[i-1] <= j:
            DP[i][j] = DP[i-1][j] OR DP[i-1][j-a[i-1]]
        else:
            DP[i][j] = DP[i-1][j]
return DP[k][target]
```

To find the actual subset, we would need to backtrack through the DP table.

d)

- The time complexity of this dynamic programming algorithm is O(k * N)

- k is number of integers
- N is sum of integers

*This is because we're filling a 2D table of size (k+1) * (N/2+1).*

- Space complexity is also O(k * N) due to the DP table.

3.

a)

A recursive function L(i, j) that represents the length of the longest subsequence which is a palindrome in substring ai ... aj.

The recursive function L(i,j) can be defined as follows:

1. Base cases:

    - If i > j: return 0 (empty substring)
    - If i == j: return 1 (single character is a palindrome of length 1)

2. Recursive cases:

    - If ai == aj: L(i,j) = L(i+1, j-1) + 2
    - If ai != aj: L(i,j) = max(L(i+1, j), L(i, j-1))

Pseudo Code:

1. If i > j, we have an empty substring, so the length of the longest palindromic subsequence is 0.
2. If *i == j*, we have a single character, which is a palindrome of length 1.
3. If *ai == aj* (if the characters at both ends match), we include these characters in our palindrome and add 2 to the length of the palindromic subsequence found in the substring *ai+1...aj-1*.
4. If *ai != aj* (the characters at the ends don't match), we take the maximum of two options:

- Exclude *ai* and find the longest palindromic subsequence in *ai+1 ... aj*
- Exclude *aj* and find the longest palindromic subsequence in *ai ... aj-1*

To find the length of the longest palindromic subsequence for the entire string, we would call L(1, n), where n is the length of the string.

b)

Pseudo Code to create an algorithm using dynamic programming:

```
function LongestPalindromicSubsequence(A):
    n = length(A)
    L = create 2D array of size n x n, initialized with 0
    P = create 2D array of size n x n, initialized with null
    for i = 0 to n-1:
        L[i][i] = 1
    for cl = 2 to n:
        for i = 0 to n-cl:
            j = i + cl - 1
            if A[i] == A[j] and cl == 2:
                L[i][j] = 2
                P[i][j] = 'MATCH'
            elif A[i] == A[j]:
                L[i][j] = L[i+1][j-1] + 2
                P[i][j] = 'MATCH'
            elif L[i+1][j] > L[i][j-1]:
                L[i][j] = L[i+1][j]
                P[i][j] = 'DOWN'
            else:
                L[i][j] = L[i][j-1]
                P[i][j] = 'RIGHT'
    return L, P

function ReconstructPalindrome(A, P):
    palindrome = empty string
    i  = 0
    j = length(A) - 1
    while i <= j:
        if P[i][j] == 'MATCH':
            palindrome = palindrome + A[i]
            i = i + 1
            j = j – 1
        elif P[i][j] == 'DOWN':
            i = i + 1
        else: // P[i][j] == 'RIGHT'
            j = j - 1

return palindrome + reverse(palindrome[:-1]) if i > j else palindrome + reverse(palindrome)
```

a. We create an *array L* of size *n* x *n*, where n is the length of the input string A. L[i][j] represents the length of the longest palindromic subsequence in the substring A[i:j+1].

b. To reconstruct the actual palindrome, we need to keep track of the decisions made at each step. We create another *array P* of the same size as L. P[i][j] stores:
   - 'MATCH' if A[i] and A[j] match and are included in the palindrome
   - 'DOWN' if we moved to the next character at i
   - 'RIGHT' if we moved to the previous character at j

c. To find the longest palindromic subsequence, we would call LongestPalindromicSubsequence(A) to get L and P, then call ReconstructPalindrome(A, P) to get the actual palindrome. The length of the longest palindromic subsequence is stored in L[0][n-1].

c)

Time complexity of this algorithm is O(n^2).

- n is the length of input string A.

4.

We were given that, in art gallery guarding problem a straight-line L that represents a long hallway in an art gallery, and we are given a set X = {x1, x2, …, xn} of real numbers that specify the positions of paintings in this hallway. Suppose that a single guard can protect all the paintings within distance at most 1 of his positions on both sides.

a)

A greedy algorithm for finding a placement of guards that uses the minimum number of guards to protect all the paintings with positions in X.

```
function PlaceGuards(X):
    Sort X in non-decreasing order
    guards = empty list
    n = length of X
    if n == 0:
        return guards
    current_guard = X[0] + 1
    guards.append(current_guard)
    for i = 1 to n-1:
        if X[i] > current_guard + 1:
            current_guard = X[i] + 1
            guards.append(current_guard)
    return guards
```

Time Complexity: O(n log n) **Space Complexity: O(n)**

Example:

Consider, X = {1.5, 2.0, 3.0, 4.5, 5.5, 6.0}

Using the algorithm:

1) Sort X: [1.5, 2.0, 3.0, 4.5, 5.5, 6.0]
2) Place the first guard at 1.5 + 1 = 2.5
3) Iterate through the paintings:
   - 2.0 and 3.0 is within the distance of 1 from 2.5, so no new guard is placed.
   - 4.5 is not within the distance of 1 from 2.5, so place a new guard at 4.5 + 1 = 5.5.
   - 5.5 and 6.0 are within the distance of 1 from 5.5, so no new guard is placed.

*Therefore, the guards should be placed at **2.5** and **5.5** to cover all the paintings with the minimum number of guards.*

b)

The algorithm satisfies the greedy choice property.

1. Optimal Substructure:

   - An optimal solution for n+1 paintings includes the optimal solution for n paintings if the $(n+1)_{th}$ painting is more than 2 units away from the nth painting.

2. Greedy Choice Property:
   - Placing a guard at xi + 1 is optimal because:
     - It covers the current painting xi.
     - It maximizes the range covered to the right (up to xi + 2).
     - Any placement to the left of xi + 1 would cover less range without additional benefit.

3. Proof by Contradiction:
   - If a better solution S' exists with fewer guards than our greedy solution S.
   - Show that this leads to a contradiction where S' either misses a painting or could be improved by adopting the greedy choice.

Therefore, the greedy choice of placing each guard at xi + 1 for the next uncovered painting ensures the minimum number of guards are used to cover all paintings.

Conclusion:
The greedy algorithm ensures that the minimum number of guards are used to cover all paintings. It achieves this by always placing guards at the optimal position (xi + 1) to maximize coverage. The proof demonstrates that this approach cannot be improved upon, guaranteeing a minimum number of guards.