ASSIGNMENT- 6

1. Design an algorithm using Dynamic Programming approach for the given function description

$$f(n) = \begin{cases} n & n=0 \text{ or } 1 \\ f(n)+f(n-1) & n>1 \text{ and even} \\ f(n)+f(n-2) & n>1 \text{ and odd} \end{cases}$$

⇒ Let us create an array 'dp' with size 'n+1' to store the results of subproblems.

ALGORITHM:

```
dp[0] = 0     //base cases
dp[1] = 1
for i in range(2, n+1):
  if i is even:
    dp[i] = dp[i] + dp[i-1]
  else:
    dp[i] = dp[i] + dp[i-2]
```

Time Complexity of the algorithm: $O(n)$

Space Complexity of the algorithm: $O(n)$

2. Let A be a NXN 2D array with all distinct elements, in which all rows and all column are sorted in descending order from larger to smaller indices. Given Key M, find out M is present in this 2D array A. Design a recursive algorithm to solve this and it must run in $O(n\log n)$ time.

⇒ One way to solve this problem is to use a modified binary search algorithm.

ALGORITHM:

```
def findKey (A, Key, i, j):
    # Base case : Key not found
      if (i >= len(A) or j<0)
        return False
    # check current element
      if (A[i][j] == Key)
        return True
    # If current element is less than key, search next row
      if (A[i][j] < Key)
        return findKey(A, Key, i+1, j)
```

If current element is greater than key, search previous column
return findKey (A, Key, i, j-1)

This algorithm runs in $O(n \log n)$ time because at each step, the size of the search space is halved, similar to a binary search.

3. Assume that multiplying a matrix M_1 of dimension $p \times q$ with another matrix M_2 of dimension $q \times r$ requires $p \times q \times r$ scalar multiplications. Consider a matrix multiplication chain $M_1 M_2 M_3 M_4 M_5$ which are of dimensions 2×25, 25×3, 3×16, 16×12, 12×4 respectively. Find the optimal parenthesized form and mention the explicty computed pairs if any.

⇒ The optimal parenthesization of the matrix multiplication chain $M_1 M_2 M_3 M_4 M_5$ is the one that minimizes the total number of scalar multiplications required to compute the product.

We can use dynamic programming, specifically we can use the matrix chain multiplication. The algorithm starts by computing the minimum no. of scalar multiplications required for sub-chains of length 2, then length 3, upto sub chains of length n (the length of the original chain).

The table is filled as follows:

(i) Initialize the diagonal of table with 0.

(ii) Fill the rest of the table by computing the minimum no. of scalar multiplications required for sub-chains of different lengths, i.e for lengths $K = 2$ to n. For each length K, consider all possible splits of the subchain has length i and the other sub-chain has length $K-i$.

(iii) At the end, the optimal parenthesization is the one corresponding to the minimum value in the table.

The optimal parenthesization would be $(((M_1 \times (M_2 \times M_3)) ((M_4 \times M_5)))$ the explicit computed pair are $M_2 M_3$ and $M_4 M_5$ and the minimal scalar multiplications required would be $(22 \times 53) + (316 \times 12) + (161 \times 24) = 2700$

Time complexity: $O(n^3)$
↳ no. of matrices in the chain

4. Since multiple events cannot run simultaneously in Vo, he has two objectives in mind:

i.) Schedule as many (non-overlapping) events as possible

ii) Schedule (non-overlapping) events in such a way that Vo is utilized for the maximum duration.

Your dynamic programming implementation must run in O(NlogN) time.

⇒ Create an array 'dp' with size 'n+1' to store the results of subproblems.

ALGORITHM: dp[0] = 0

for i in range (1, n+1):

dp[i] = max(dp[i], dp[j] + util[i]

where j is the first compatible interval after i

if it exists,

else j = 0.

The bottom of the problem is to solve small subproblems, and then gradually solve larger problems using the solutions already computed.

The dynamic programming implementation should run in O(N logN) time by storing the results in the array and using a divide and conquer approach.

5. Identify the overlapping subproblems for this problem. Detect how many subproblems are to be solved to get to the answer when (w1, w2, w3, w4) = (117, 113, 114, 115) and (p1, p2, p3, p4) = ($4200, $1200, $4000, $2500) with bag capacity w = 100.

⇒ The problem can be solved by Knapsack Algorithm.

In this case, it would be to select a subset of the 4 items with weight (w1, w2, w3, w4) = (117, 113, 114, 115) and value (p1, p2, p3, p4) = ($4200, $1200, $4000, $2500) that maximizes value while keeping the total weight less than or equal to w = 100.

One can define a function F(i, j) as the maximum value that can be obtained using items 1 through i and total weight of j. The function can then be defined recursively based on the decision of whether to include item i or not.

In this case, the overlapping subproblems are the subproblems of selecting subsets of the items with smaller capacities and the of items with smaller capacities and the subproblems of selecting subsets of the items with smaller values.

Since the capacity is 100 and the weight of items are $(w1, w2, w3, w4) = (117, 113, 114, 115)$ and the number of items is 4 the solution will be in $4 \times 100 = 400$ subproblems.

6. Suppose we are given a directed graph $G = (V, E)$ with costs on the edges; the costs may be positive or negative, but every cycle in the graph has strictly positive cost. Give an efficient algorithm that computes the no. of shortest v-w paths in G.

⇒ One way to solve this problem is to use Bellmann-Ford algorithm with a slight modification:

(i) Initialize an array dist[] of size V, where V is the no. of vertices in the graph, to store the shortest distance from the starting node v to each vertex.
set dist[v] = 0 and dist[i] = ∞ for all other vertices ~~of~~ i.

(ii) Create an array count [] of size V to store V to store the number of shortest paths from v to each vertex. Set count [v] = 1 and count [i] = 0 for all other vertices i.

(iii) Run the Bellmann-Ford algorithm for |V|-1 iterations.
In each iteration, for each edge (u, v) with weight w, if dist [u] + w < dist [v], set dist [v] = dist [u] + w and count [v] = count [u]. ~~If~~
If dist [u] + w is equal to dist [v], add count [u] to count [v].

(iv) After the |V|-1 iterations, check for negative cycles. If there is a negative cycle, the problem has no solution.

(v) Return count [w] as the no. of shortest paths from v to w.

Time Complexity: $O(V \times E)$ V → no. of vertices
Space Complexity: $O(V)$ E → no. of edges

No. of shortest paths from u to v = no. of shortest paths from the predecessor of u on this path to v.