NAME :- SHISHU
REG :- 2020CA089

Assignment 5

Question 1 : Solution

Principle of optimality :- A problem is said to satisfy the principle of optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.

e.g. The shortest path problem satisfies the principle of optimality.

Question 2 : solution

Coin change :- We have to find the lowest no. of coins that makeup the given amount if not possible than -1

Input :- Coins in the array e.g. [1, 2, 5]
Amount eg. 11

output : 3 { 5 + 5 + 1 = 11 }

```
int coinchange( coins[] , amount) {
      dp (amount +1, ∞);
       dp [0] = 0;
       for (i=1 to i<= amount; i++) {
           for (auto c : coins) {
               if (c<=i and dp[i-c] !=∞)
                       dp[i] = min (dp[i] , dp[i-c] +1)
           }
    }
    if (dp[amount] == ∞)
            return -1
      return dp[amount];
}
```

Complexity analysis:-

Time Complexity :- no. of coins x amount

$\Rightarrow m \times n$ { n = no. of coins

$\Rightarrow O(mn)$ (m = amount.

Space Complexity: $O(m)$ { to store dp

Question 3: Solution

Matric Chain Multiplication :- The mcm problem we are not actully multiplying matrixes. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Algo // matrix A[i] has dimension dims[i-1] x dims[i]
for i = 1 ... n

```
mcm (int dims[]) {
      n = dims.length-1;
      for (i=1; i<m; i++)
          m[i,i] = 0;
      for (len=2 ; len<=n; i++) {
          for (i=1 ; i<= n-len+1; i++) (
              j = i + len - 1;
```

```
        m[i,j] = INF;
        for(k=i ; k <= j-1 ; k++ ){
            cost = m[i,k] + m[k+1,j]
                    + dims[i-1] * dims[k] * dims[j];
        if (cost < m[i,j]) {
                m[i,j] = cost ;
            }
          }
    }
```

Time Complexity: $O(n^3)$
Space Complexity: $O(n^2)$

Question: 4 Solution

0-1 Knapsack problem :- In 0-1 we can not break the item, either pick the complex item or don't pick it. Therefore we have 2 choice for each item.
This can be solved recursively in 2^n time complexity. Now, we minimize this using dynamic programing:-

Algo:-

```
    int knapsack(w , wt[], val[], i, dp) {
        if (i<0)
            return 0;

        if (dp[i][w] != -1)
            return dp[i][w];

        if (wt[i] > w) {
            dp[i][w] = knapsack(w, wt, val, i-1, dp);
            return dp[i][w];
        }
```

```
else{
    dp[i][w] = max(val[i] + knapsack(w - wt[i], wt,
                                val, i-1, dp),

        knapsack(w , wt , val , i-1 , dp));

    return dp[i][w];

}
}
```

Time Complexity : O(n*w)

Space complexity : O(n*w) { for dp