Experiment 2.3

Name: UTKARSH JOSHI **UID: 21BCS9158**
Branch: BE-CSE **Section: ST_802-A**
Semester: 5th **Date of Performance: 28/9/2023**
Subject Code: 21CSH-311 **Subject Name: DAA Lab**

1. **Aim:** Develop a program and analyze complexity to implement 0-1 Knapsack using Dynamic Programming.

2. **Objectives:** To implement 0-1 Knapsack using Dynamic Programming.

3. **Tools/Apparatus Used:** Visual Studio Code

4. **Procedure/Algorithm:**

Step 1: This step checks if the base cases are met. If the number of items is 0 or the maximum capacity of the knapsack is 0, then the maximum value is 0.

Step 2: This step checks if the weight of the last item is greater than the current capacity. If it is, then the last item cannot be included in the knapsack, so the function recursively considers the case where the last item is excluded from the knapsack.

Step 3: This step recursively considers both cases: where the last item is included in the knapsack and where the last item is excluded from the knapsack.

Step 4: This step returns the maximum of the two values.

5. Sample Code:

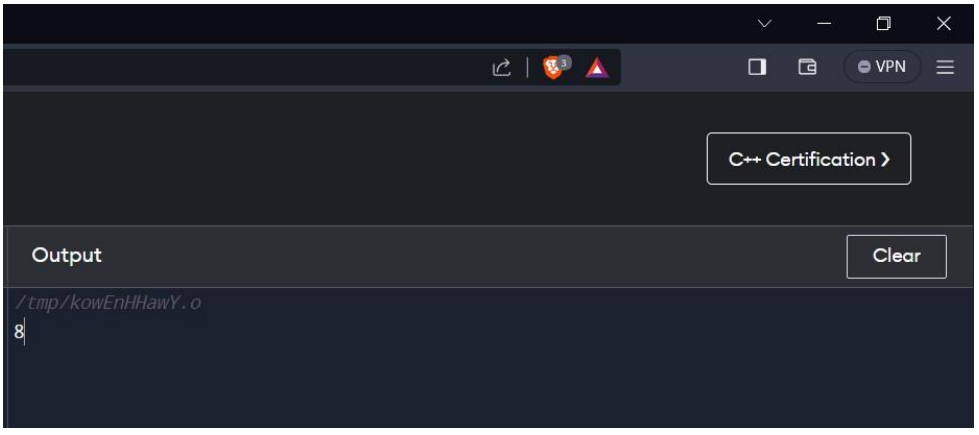
```
#include <bits/stdc++.h> using
namespace std;

int max(int a, int b){
return (a > b) ? a : b;
}

int knapSack(int W, int wt[], int val[], int n)
{
        if (n == 0 || W == 0)
        return 0;
        if (wt[n - 1] > W)
                return knapSack(W, wt, val, n - 1);
        else
                  return max( val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1), knapSack(W, wt, val, n - 1)); }

int main()
{       int profit[] = { 1 , 2 , 5 , 6 };
        int weight[] = { 2 , 3 , 4 , 5 };
        int W = 8;
        int n = sizeof(profit) / sizeof(profit[0]);
        cout << knapSack(W, weight, profit, n);
        return 0;
}
```

6. Observations/Outcome :



7. Time Complexity:

O(nW) where n is the number of items and W is the capacity of knapsack.