Name : Pratik Kanhaiya Rathod
Roll : 321056
PRN : 22010885
Batch : A2

Assignment No. 2

Aim : Implement 0/1 knapsack using Dynamic Programming.

Objective :

- Efficiency : Dynamic Programming provides an efficient solution for 0/1 knapsack problem as it avoids repetitive computations and reduces the time complexity of the algorithm.
- Optimality : DP solution provides the optimal solution to 0/1 knapsack problem. It ensures that the maximum value is obtained while respecting the capacity constraint of the knapsacks.
- Flexibility : DP allows the problem to be solved for different knapsack capacities and variety of items with different weights and values.

Theory :

What is Knapsack Problem ?

⇨ Knapsack problem is a problem in which we have certain weights given to us along with their values/cost and the total capacity of the knapsack. Our goal is to pack the knapsack in such a way that we get the maximum total value/cost, keeping in mind that the total weight we can add is fixed.

There are 2 versions of knapsack problem, here we will be discussing the 0–1 knapsack problem.

In the 0–1 knapsack problem, the items are indivisible we can either pick the item or drop it. This problem is solved using **Dynamic Programming**.

0–1 Knapsack Problem

Given: A knapsack with maximum capacity W and n items that need to be added to the knapsack.

Each item i has some weight w_i and cost c_i.

To find: How to pack the knapsack to attain the maximum total value of packed items?

- **Solving using brute force approach:**

  As there are n items, therefore there are 2^n possible combinations of items and we need to go through all the combinations to find the best fit, so the time complexity will be $O(2^n)$.

- **Solving using Dynamic Programming:**

  The recursive formula is:

$$C[k,w] = \begin{cases} C[k-1,w] & \text{if } w_k > w \\ \max\{C[k-1,w], C[k-1,w-w_k] + ck\} & \text{else} \end{cases}$$

- **First case: $w_i > W$**

  Item i cannot be added to the knapsack as the maximum weight of the knapsack is W.

- **Second case**: **$w_i \leq W$**

  Item i can be added to the knapsack and we will choose the case with more cost/benefit value.

  0–1 Knapsack Algorithm

```
for w = 0 to W                          -----O(W)
  C[0,w] = 0
for i = 1 to n
  C[i,0] = 0
for i = 1 to n                        -----Repeat n times
 for w = 0 to W                         -----O(w)
  if wi<= w // item i can be added to knapsack
    if ci + C[i-1, w-wi] > C[i-1, w]
          C[i,w] = ci + C[i-1, w-wi]
    else
          C[i,w] = C[i-1,w]
  else
    C[i,w] = C[i-1,w]  // wi > W
```

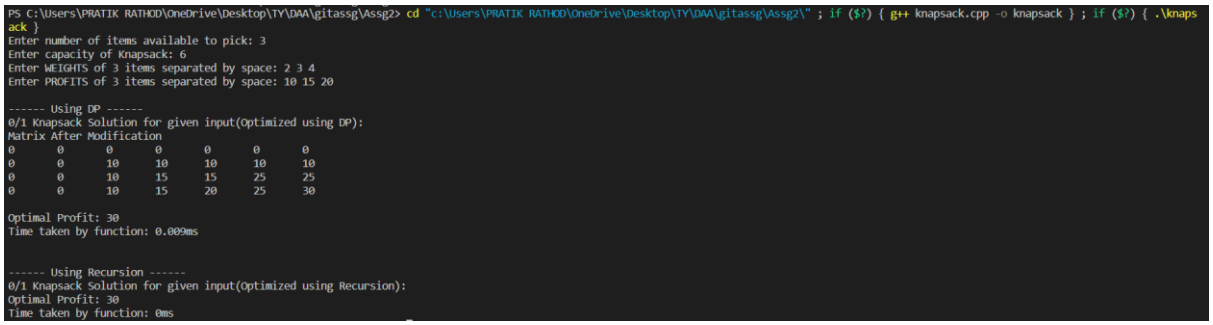
  Time complexity = O(n*W)

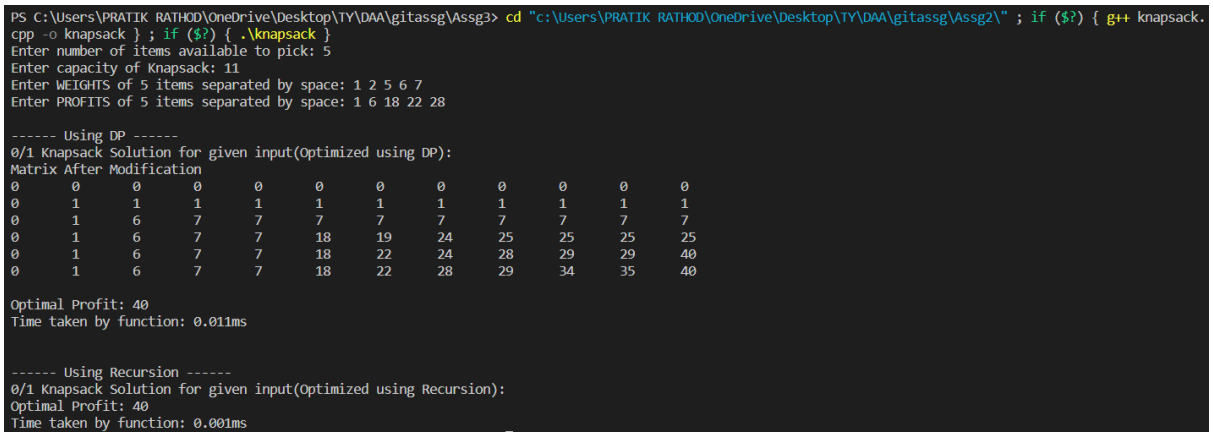
Code :

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
/*
Name : Pratik K. Rathod
Roll No : 321056
PRN No: 22010885
*/

int t[100][100];

int KnapsackRecursion(int wt[], int val[], int W, int n)
{
    if(n == 0 || W == 0) return 0;

    if(wt[n-1] <= W) return max(val[n-1] + KnapsackRecursion(wt, val, W-wt[n-
1], n-1), KnapsackRecursion(wt, val, W, n-1));

    else if(wt[n-1] > W) return KnapsackRecursion(wt, val, W, n-1);

    else return -1;

    // return (wt, val, W, n-1);
}

int KnapsackDP(int wt[], int val[], int w, int n)
{
    // int t[n+1][w+1];
    memset(t, 0, sizeof(t));
    for (int i = 1; i < n + 1; i++)
    {
        for (int j = 1; j < w + 1; j++)
        {
            if (wt[i - 1] <= j)
            {
                t[i][j] = max(val[i - 1] + t[i - 1][j - wt[i - 1]], t[i -
1][j]);
            }
            else
            {
                t[i][j] = t[i - 1][j];
            }
        }
    }
```

```
    cout << "Matrix After Modification\n";
    for (int i = 0; i < n + 1; i++)
    {
        for (int j = 0; j < w + 1; j++)
        {
            cout << t[i][j] << "\t";
        }
        cout << "\n";
    }
    return t[n][w];
}

int main()
{
    int n, w;
    cout << "Enter number of items available to pick: ";
    cin >> n;
    cout << "Enter capacity of Knapsack: ";
    cin >> w;
    int wt[n];
    int val[n];
    cout << "Enter WEIGHTS of " << n << " items separated by space: ";
    for (int i = 0; i < n; i++)
        cin >> wt[i];

    cout << "Enter PROFITS of " << n << " items separated by space: ";
    for (int i = 0; i < n; i++)
        cin >> val[i];
    clock_t start, end;
    cout << "\n------ Using DP ------" << endl;
    cout << "0/1 Knapsack Solution for given input(Optimized using DP): " <<
endl;

    start = clock();
    cout<<"\nOptimal Profit: "<< KnapsackDP(wt, val, w, n)<<endl;
    end = clock();
    cout << "Time taken by function: " << double(end - start) /
double(CLOCKS_PER_SEC) << setprecision(11) << "ms" << endl;

    cout<<"\n\n------ Using Recursion ------"<<endl;

    cout << "0/1 Knapsack Solution for given input(Optimized using
Recursion):"<< endl;
    start = clock();
    cout<<"Optimal Profit: "<< KnapsackRecursion(wt, val, w, n)<<endl;
    end = clock();
    cout << "Time taken by function: " << double(end - start) /
double(CLOCKS_PER_SEC) << setprecision(11) << "ms" << endl; }
```

Output :





Discussion :

The time complexity is O(c*n) since we have to fill each value in the table and carry out a constant time calculation at each cell.

```
public int findMaxValue(int[] w, int[] v, int capacity) {
// create the DP table
    int[][] dp = new int[w.length][capacity + 1];    // start
at n, c = 1 since the first row and column are
    // always all 0
   for (int n = 1; n < w.length; n++) {
       for (int c = 1; c < capacity + 1; c++) {
           if (w[n] > c) {
               dp[n][c] = dp[n - 1][c];
           }else {
               int remainingCapacity = c - w[n] < 0 ? 0 : c -
w[n];
               int chooseN = v[n] + dp[n -
1][remainingCapacity];
               int discardN = dp[n - 1][c];
              dp[n][c] = Math.max(chooseN, discardN);
           }
       }
```

```
    }
    return dp[w.length - 1][capacity];
}
```

This algorithm simply builds the 2d array starting at [1][1] building towards **[n-1][c]** in the manner described above. The time complexity is O(n*c) and the space complexity is also O(n*c) but since we're not using recursion, the algorithm runs with less space usage by a constant factor.

Conclusion :

When I first encountered the knapsack problem, I found it pretty tricky and also found existing online explanations frustrating since they didn't explain the underlying reasoning why you need build the DP table the way it and state transitions. What helped me out the most is to step though the naive solution, add memorization, then try to create the DP solution myself. In the end, I decided to document my procedure here. Hopefully this article helps you out on your own dynamic programming research.