**Experiment 2.3**

**Student Name: Sahul Kumar Parida UID: 20BCS4919**

**Branch: CSE Section/Group: WM-904/B**

**Semester: 5th Date of Performance: 18/10/2022**

**Subject Name: Design and Analysis Algorithm Lab**

**Subject Code: 20CSP-312**

1. **Aim/Overview of the practical:**

To implement 0-1 Knapsack using Dynamic Programming.

1. **Task to be done/which logistics used:**

Using Dynamic Programming

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler sub problems, solving each of those sub problems just once, and storing their solutions using a memory-based data structure (array, map, etc).

1. **Algorithm/Flowchart:**

 In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a DP[][] table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as the rows.   
The state DP[i][j] will denote maximum value of ‘j-weight’ considering all values from ‘1 to ith’. So if we consider ‘wi’ (weight in ‘ith’ row) we can fill it in all columns which have ‘weight values > wi’. Now two possibilities can take place:

* Fill ‘wi’ in the given column.
* Do not fill ‘wi’ in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill ‘ith’ weight in ‘jth’ column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row. So we take the maximum of these two possibilities to fill the current state.

1. **Steps for experiment/practical/Code:**

#include <bits/stdc++.h>

using namespace std;

// A utility function that returns

// maximum of two integers

int max(int a, int b)

{

return (a > b) ? a : b;

}

// Returns the maximum value that

// can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

int i, w;

vector<vector<int>> K(n + 1, vector<int>(W + 1));

// Build table K[][] in bottom up manner

for(i = 0; i <= n; i++)

{

for(w = 0; w <= W; w++)

{

if (i == 0 || w == 0)

K[i][w] = 0;

else if (wt[i - 1] <= w)

K[i][w] = max(val[i - 1] +

K[i - 1][w - wt[i - 1]],

K[i - 1][w]);

else

K[i][w] = K[i - 1][w];

}

}

return K[n][W];

}

int main()

{

cout<<"Sahul Kumar Parida"<<endl;

cout<<"20BCS4919"<<endl;

int n;

cout<<"Enter number of values = ";

cin>>n;

int \*wt=new int[n];

int \*val=new int[n];

for(int i=0;i<n;i++)

{

cout<<"Enter weight of "<<i+1<<" item ";

cin>>wt[i];

cout<<"Enter value of "<<i+1<<" item ";

cin>>val[i];

}

cout<<"Enter capacity of knapSack = ";

int W;

cin>>W;

cout << knapSack(W, wt, val, n);

return 0;

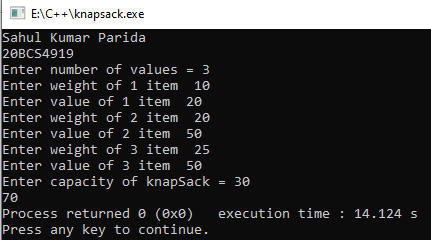
}

1. **Observations/Discussions/ Complexity Analysis:**

**Time Complexity:** O(N\*W).   
where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.

**Auxiliary Space:** O(N\*W).   
The use of 2-D array of size ‘N\*W’.

1. **Output:**



**Learning outcomes (What I have learnt):**

1. Dynamic Programming
2. To implement problems based on different algorithm design techniques.
3. To learn the importance of designing an algorithm in an effective way by considering space and time complexity.