# Experiment – 2.3

**Student Name:** Jitesh Kumar
**UID:** 20BCS2334
**Branch:** CSE
**Section/Group:** WM_903-A
**Semester:** 5
**Date of Performance:** 15/10/2022
**Subject Name:** Design & Analysis Algorithm
**Subject Code:** 20CSP-312

## 1. Aim:

Code to implement 0-1 knapsack problem using dynamic programming.

## 2. Task to be done:

Code to implement 0-1 knapsack problem using dynamic programming.

## 3. Algorithm:

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:

1. Fill 'wi' in the given column.
2. 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. This visualisation will make the concept clear.

**4. Code:**

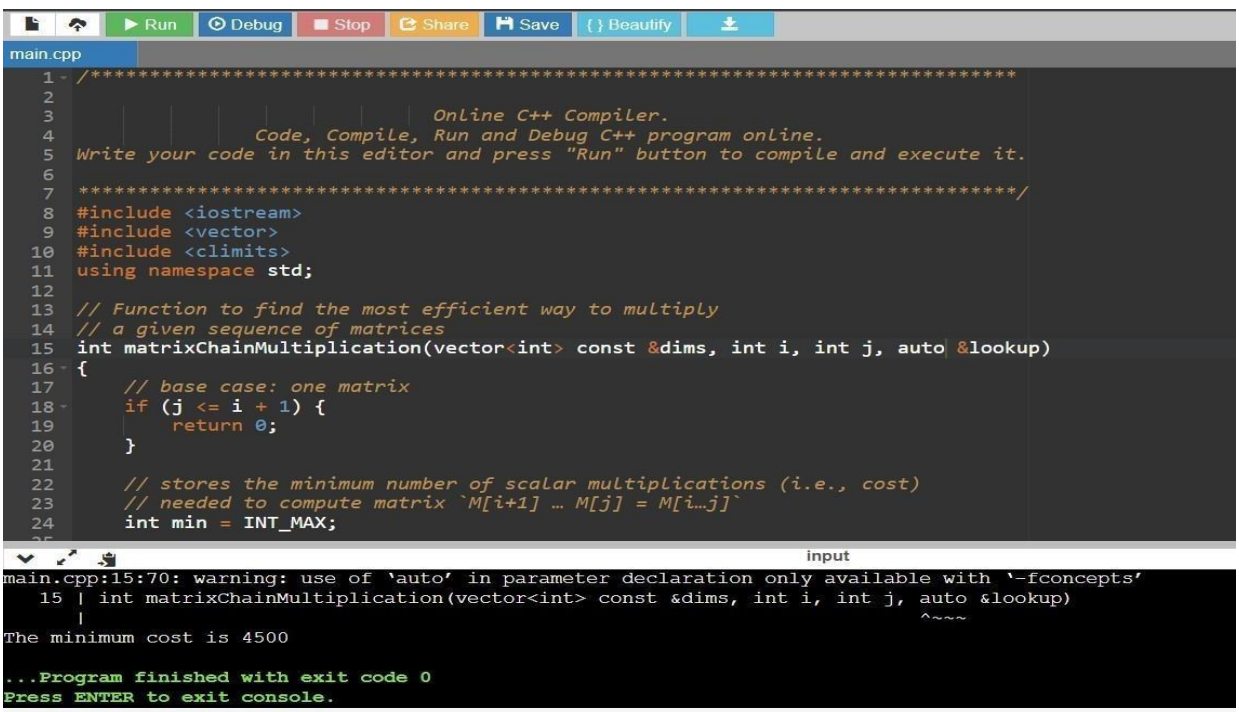
```
#include <bits/stdc++.h>
using namespace std;
max(int a, int b){ return
(a > b) ? a : b;
} knapSack(int W, int wt[], int val[], int
n){ int i, w;
     vector<vector<int>> K(n + 1, vector<int>(W + 1));
   for(i = 0; i <= n; i++){ for(w = 0;
      w <= W; w++){ if (i == 0 || w
      == 0) K[i][w] =
            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(){
```

```
    int val[] = { 60, 100, 120 }; int wt[] = { 10,
    20, 30 }; int W = 50; int n = sizeof(val) /
    sizeof(val[0]); cout << knapSack(W, wt,
    val, n); return 0;
}
```

## 5. Complexity Analysis:

- **Time Complexity:** O(N*W)
- **Auxiliary Space:** O(N*W)

## 6. Result:



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

1. Learn about dynamic programming.
2. Learn about time complexity of program.
3. Solve knapsack problem.