Practical-2.3

Student Name: Pranjal Kumar
UID: 20BCS3504
Branch: CSE
Section/Group: 607 /B
Semester: 05
Date of Performance: 14/10/2022
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;

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 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;
}
```

main.cpp

```cpp
#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 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:

main.cpp

```cpp
#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()
{
```

input

```
220

...Program finished with exit code 0
Press ENTER to exit console.
```

Learning outcomes (What I have learnt):

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