



Bansilal Ramnath Agarwal Charitable Trust's  
Vishwakarma Institute of Information Technology

Department of  
Artificial Intelligence and Data Science

Name: **Sahil Dilip Sonawane**

Class: **TY**

Division: **C**

Roll No: **373057**

Semester: **V**

Academic Year: **2023-24**

Subject Name & Code: **Design and Analysis of Algorithm**

Title of Assignment: **Implement 0/1 Knapsack problem using Following algorithmic strategies.**

**(a) Dynamic programming**

**(b) Back tracking**

**(C) Branch and bound**

## **Assignment No: 5**

### **Aim:**

To implement 0/1 Knapsack problem using different algorithmic strategies.

### **Problem Statement:**

Implement 0/1 Knapsack problem using Following algorithmic strategies.

- (a) Dynamic programming
- (b) Back tracking
- (C) Branch and bound

### **Background Information:**

Given weights and values of  $n$  items, put these items in a knapsack of capacity  $W$  to get the maximum total value in the knapsack. In other words, given two integer arrays  $val[0..n-1]$  and  $wt[0..n-1]$  which represent values and weights associated with  $n$  items respectively. Also given an integer  $W$  which represents knapsack capacity, find out the maximum value subset of  $val[]$  such that sum of the weights of this subset is smaller than or equal to  $W$ . You cannot break an item, either pick the complete item or don't pick it (0-1 property).

### **Software Requirements:**

Text Editor: VSCode, Online GDB Compiler

Environment: GCC C++

### **Using Dynamic Programming approach**

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.

### **Program Code:**

```
#include <bits/stdc++.h>

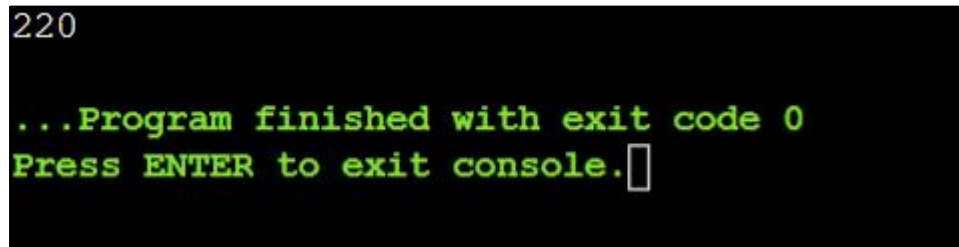
using namespace std;

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

int knapSack(int W, int wt[], int val[], int n)
{
    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;}
```

### **Output:**



```
220
...Program finished with exit code 0
Press ENTER to exit console.█
```

### **Using Backtracking approach**

In the above approach we used a 2D array to store the computed results in our DP but we can observe one thing that to compute the result for the  $n$ th element we just need the results for the  $(n-1)$ th element thus we do not need the rest of the result.

Therefore, if we just get the results for the  $(n-1)$ th element we can reach the  $n$ th result.

Thus, we can reduce the size of our DP to 1D just storing the results on different weights till the previous element.

### **Program Code:**

```
#include <bits/stdc++.h>

using namespace std;

int knapSack(int W, int wt[], int val[], int n)
{
    int dp[W + 1];
    memset(dp, 0, sizeof(dp));

    for (int i = 1; i < n + 1; i++) {
        for (int w = W; w >= 0; w--) {

            if (wt[i - 1] <= w)
                dp[w] = max(dp[w],
                    dp[w - wt[i - 1]] + val[i - 1]);
        }
    }

    return dp[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;
}

```

### **Output:**

```

220
-----
Process exited after 0.08011 seconds with return value 0
Press any key to continue . . .

```

### **Using Branch and Bound approach**

The idea is to use the fact that the Greedy approach provides the best solution for Fractional Knapsack problem.

To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using Greedy approach. If the solution computed by Greedy approach itself is more than the best so far, then we can't get a better solution through the node.

### **Algorithm:**

1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
5. Do following while Q is not empty.
  - Extract an item from Q. Let the extracted item be u.
  - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.

- Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

### **Program Code:**

```
#include <bits/stdc++.h>

using namespace std;

struct Item
{ float weight;
  int value;
};struct Node
{
  int level, profit, bound;
  float weight;
};

bool cmp(Item a, Item b)
{
  double r1 = (double)a.value / a.weight;
  double r2 = (double)b.value / b.weight;
  return r1 > r2;
}

int bound(Node u, int n, int W, Item arr[])
{
  if (u.weight >= W)
    return 0;

  int profit_bound = u.profit;
  int j = u.level + 1;
  int totweight = u.weight;
```

```

while ((j < n) && (totweight + arr[j].weight <= W))
{
    totweight += arr[j].weight;
    profit_bound += arr[j].value;
    j++;
} if (j < n)
    profit_bound += (W - totweight) * arr[j].value /
                    arr[j].weight;

return profit_bound;
}

int knapsack(int W, Item arr[], int n)
{
    sort(arr, arr + n, cmp);
    queue<Node> Q;

    Node u, v;
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    int maxProfit = 0;
    while (!Q.empty())
    {u = Q.front();
        Q.pop();
        if (u.level == -1)
            v.level = 0;
        if (u.level == n-1)
            continue;
        v.level = u.level + 1;
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;
    }
}

```

```

        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
    }
    return maxProfit;
}

int main()
{
    int W = 10;

    Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                  {5, 95}, {3, 30}};

    int n = sizeof(arr) / sizeof(arr[0]); cout << "Maximum possible profit = "
        << knapsack(W, arr, n);

    return 0;}

```

### **Output:**

```

Maximum possible profit = 235
-----
Process exited after 0.07437 seconds with return value 0
Press any key to continue . . .

```

### **Conclusion:**

Implemented 0/1 Knapsack problem using Following algorithmic strategies.  
 (a) Dynamic programming (b) Back tracking C) Branch and bound