K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: A1 **Roll No.: 16010123012**

Experiment No.: 5A

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Study, Implementation, and Analysis of 0/1 Knapsack Problem.

Objective: To learn 0/1 Knapsack Problem using Dynamic Programming Approach. (Maximize the total value of selected items while ensuring their total weight does not exceed W.)

CO to be achieved:

CO 2 Describe various algorithm design strategies to solve different problems and analyse Complexity.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf**
4. **http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/**
5. **http://www.mafy.lut.fi/study/DiscreteOpt/tspdp.pdf**
6. **https://class.coursera.org/algo2-2012-001/lecture/181**
7. **http://www.quora.com/Algorithms/How-do-I-solve-the-travelling-salesman-problem-using-Dynamic-programming**
8. **www.cse.hcmut.edu.vn/~dtanh/download/Appendix_B_2.ppt**
9. **www.ms.unimelb.edu.au/~s620261/powerpoint/chapter9_4.ppt**

Pre Lab/ Prior Concepts:

Data structures, Concepts of algorithm analysis

Historical Profile:

Dynamic Programming (DP) is used heavily in optimization problems (finding the maximum and the minimum of something). Applications range from financial models and operation research to biology and basic algorithm research. So the good news is that

understanding DP is profitable. However, the bad news is that DP is not an algorithm or a data structure that you can memorize. It is a powerful algorithmic design technique.

The 0/1 Knapsack Problem is one of the most studied problems in computer science, operations research, and combinatorial optimization. Its origins and development reflect the evolution of mathematical optimization and computational techniques.

Historical Background

Origins in Resource Allocation: The knapsack problem has its roots in resource allocation problems dating back centuries, where individuals needed to maximize their gain (value) from limited resources (capacity). The term "knapsack" derives from the idea of filling a knapsack with items to achieve the greatest benefit without exceeding its weight limit.

Early Formalization: The problem was first mathematically formalized in the early 20th century as part of broader studies in combinatorics and optimization. It gained attention as a theoretical challenge in discrete mathematics.

Greedy Algorithms and Limitations: Researchers also explored greedy approaches for simplified variants (e.g., fractional knapsack). The greedy algorithm does not work for the 0/1 knapsack problem due to its inability to handle binary choices optimally.

New Concepts to be learned:

Application of algorithmic design strategy to any problem, dynamic Programming method of problem solving Vs other methods of problem solving, optimality of the solution, Optimal Binary Search Tree Problems and their applications

Code:

```
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;

int knapsack_01(int capacity, vector<int> &values, vector<int> &weights, int
n)
{
  vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

  for (int i = 1; i <= n; i++)
  {
    for (int w = 0; w <= capacity; w++)
    {
      if (weights[i - 1] <= w)
      {
        dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i -
1]]);
      }
      else
      {
```

```
        dp[i][w] = dp[i - 1][w];
      }
    }
  }
  return dp[n][capacity];
}

int main()
{
  int n, capacity;
  cout << "Enter number of objects: ";
  cin >> n;
  cout << "Enter the capacity of knapsack: ";
  cin >> capacity;

  vector<int> values(n), weights(n);
  cout << "Enter value and weight of each object respectively: " << endl;
  for (int i = 0; i < n; i++)
  {
    cin >> values[i] >> weights[i];
  }

  int max_profit = knapsack_01(capacity, values, weights, n);
  cout << "Maximum profit: " << max_profit << endl;
}
```

Output:

```
Enter number of objects: 4
Enter the capacity of knapsack: 5
Enter value and weight of each object respectively:
4 2
8 3
9 4
11 5
Maximum profit: 12
```

Theory / Algorithm / Example / Solution for the example / Analysis of algorithm:
Time Complexity: $O(n^2)$
Space Complexity: $O(n^2)$

- Knapsack (w[], v[], W)

1. Take inputs of number of items n, array of values v & weights w, max weight capacity W
2. Create a 2D array dp[n+1][W+1]
   set all values dp[i][w] = 0 initially
3. For i = 1 to n,
   For w = 1 to W,
   if item fits in knapsack (w[i-1] ≤ W)
   dp[i][w] = max(dp[i-1][w], v[i-1] + dp[i-1][w-wt[i-1]])
   else
   dp[i][w] = dp[i-1][w]
4. Print dp[n][W]

- Example (c = 5)

item:	1	2	3	4
weight:	2	3	4	5
value:	4	8	9	11

item	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	4	4	4	4
2	0	0	4	8	8	12
3	0	0	4	8	9	12
4	0	0	4	8	9	12

dp[1][2] = max(dp[0][2], val[0] + dp[0][2 - wt[0]])
= max(dp[0][2], 4 + dp[0][0])
= max(0, 4 + 0)
= 4

similarly for others
Max value = 12

P1-10
24/03/25

CONCLUSION:

I have completed the study, implementation and analysis of the 0/1 Knapsack Problem using the Dynamic Programming approach. Through this experiment, I have learned how to maximize the total value of selected items while ensuring their total weight does not exceed the given capacity. The implementation involved constructing a DP table to store optimal solutions for subproblems, ensuring an efficient solution. The time complexity of the algorithm is O(n × W), making it feasible for moderate input sizes. This experiment enhanced my understanding of algorithm design strategies, particularly dynamic programming and its real-world applications in optimization problems like resource allocation.