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. 4

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

Signature of the Staff In-charge with date

Title: Study, Implementation, and Analysis of Fractional Knapsack Problem.

Objective: To learn fractional 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.

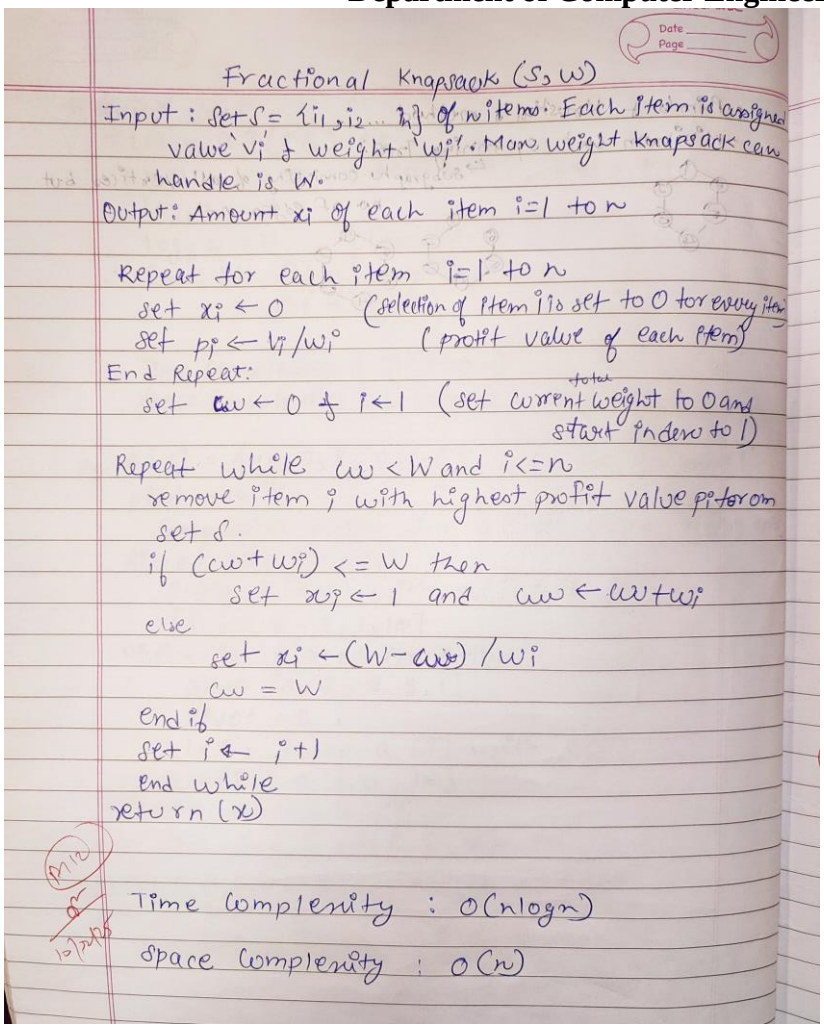
Early Formalization: The problem was first mathematically formalized in the early 20th century as part of broader studies in combinatorics and optimization. It gained attention as a theoretical challenge in discrete mathematics.

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

Algorithm:



Fractional Knapsack (S, W)

Input : Set S = {i1, i2, ... in} of n items. Each item is assigned value vi & weight wi. Max weight knapsack can handle is W.

Output: Amount xi of each item i=1 to n

Repeat for each item i=1 to n
  set xi ← 0 (selection of item i is set to 0 for every item)
  set pi ← vi/wi (profit value of each item)
End Repeat:
  set cw ← 0 & i ← 1 (set current weight to 0 and start index to 1)

Repeat while cw < W and i<=n
  remove item i with highest profit value pi from set S.
  if (cw + wi) <= W then
    set xi ← 1 and cw ← cw + wi
  else
    set xi ← (W - cw) / wi
    cw = W
  end if
  set i ← i + 1
end while
return (x)

Time complexity : O(n log n)

Space complexity : O(n)

Code:

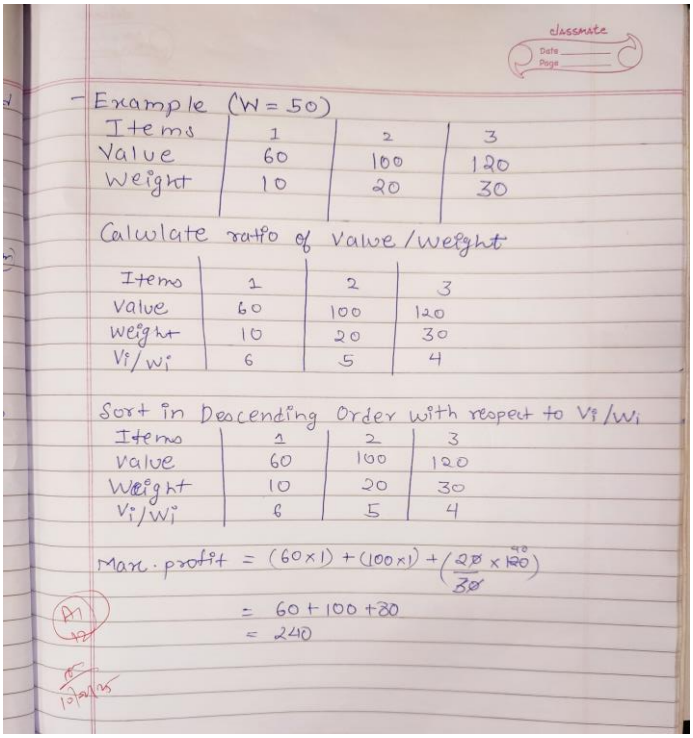
```
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;

struct Object
{
    double value, weight, value_by_weight;
};

bool compare(Object a, Object b)
{
    return a.value_by_weight > b.value_by_weight;
}

double fractional_knapsnack(int capacity, vector<Object> &Objects)
{
    int n = Objects.size();
    for (int i = 0; i < n; i++)
```

```
    {
        Objects[i].value_by_weight = Objects[i].value / Objects[i].weight;
    }

    sort(Objects.begin(), Objects.end(), compare);
    double total_value = 0.0;
    int cw = 0;

    for (int i = 0; i < n; i++)
    {
        if (cw + Objects[i].weight <= capacity)
        {
            cw += Objects[i].weight;
            total_value += Objects[i].value;
        }
        else
        {
            double fractional = (double)(capacity - cw) / Objects[i].weight;
            total_value += Objects[i].value * fractional;
            break;
        }
    }
    return total_value;
}

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

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

    double max_profit = fractional_knapsnack(capacity, Objects);
    cout << "Maximum profit: " << max_profit << endl;
}
```

Output:

```
Enter number of Objects: 3
Enter the capacity of knapsack: 50
Enter value and weight of each Object respectively:
60 10
100 20
120 30
Maximum profit: 240

Process returned 0 (0x0)   execution time : 13.291 s
Press any key to continue.
```

Example:



- Example (W = 50)

Items	1	2	3
Value	60	100	120
Weight	10	20	30

Calculate ratio of Value/Weight

Items	1	2	3
Value	60	100	120
weight	10	20	30
V_i/w_i	6	5	4

Sort in Descending Order with respect to V_i/w_i

Items	1	2	3
Value	60	100	120
Weight	10	20	30
V_i/w_i	6	5	4

Max. profit = $(60 \times 1) + (100 \times 1) + \left(\frac{20}{30} \times 120\right)$

= 60 + 100 + 80

= 240

Analysis of algorithm:
Time Complexity: O(nlogn)
Space Complexity: O(n)

CONCLUSION:
In this experiment, I implemented and analyzed the Fractional Knapsack Problem using a greedy algorithm. The approach involved sorting items based on their value-to-weight ratio and selecting items either fully or partially to maximize the total value within the given capacity constraint. The time complexity of the algorithm is O(n log n) due to the sorting step, followed by a linear scan of the items. The space complexity is O(n) as we store the item details. This experiment helped me strengthen my understanding of greedy algorithms.