# PROJECT REPORT

*Submitted by*

**K.Hemanth Reddy [RA2111030010087]**
**G.Pranay[RA2111030010115]**
**M.E.V.S.Akhil Varma[RA2111030010099]**

*Under the Guidance of*

**Dr. Lavanya.V**
Assistant Professor, Department of Networking and Communications

*In partial satisfaction of the requirements for the degree of*

**BACHELOR OF TECHNOLOGY**
in
**COMPUTER SCIENCE AND ENGINEERING**

**with specialization in CYBER SECURITY**

**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGYSRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR - 603203**

**APRIL 2023**

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR — 603 203

BONAFIDE CERTIFICATE

Certified that this project report "KNAPSACK" is the bonafide work of

K.Hemanth Reddy(RA2111030010087) & G.Pranay(RA2111030010115) & M.E.V.S.Akhil Varam(RA2111030010099) who carried out the project work under my supervision.

| | |
|---|---|
| SIGNATURE | SIGNATURE |
| Dr. V. Lavanya | Dr. Annapurani Panaiyappan K |
| Assistant Professor | Head of the Department |
| Networking and Communications | Networking and Communications |
| SRM Institute of Science and Technology | SRM Institute of Science and Technology |
| Kattankulathur | Kattankulathur |

School of Computing

SRM IST, Kattankulathur – 603 203 Course

Code: 18CSC204J

Course Name: DESIGN AND ANALYSIS OF ALGORITHMS

| | |
|---|---|
| **Problem Statement** | Knapsack algorithm for organizing a large collection of items ina bag with high profit |
| **Name of the candidate** | K Hemanth Reddy |
| **Team Members** | Pranay<br>Akhil |
| **Date of submission** | 28/04/23 |

**Mark Split Up**

| S. No | Description | Maximum Mark | Mark Obtained |
|---|---|---|---|
| 1 | Exercise | 5 | |
| 2 | Viva | 5 | |
| | **Total** | **10** | |

**Staff Signature with date**

COURSE CODE:18CSC204J

COURSE NAME: Design and analysis of Algorithms

**CONTENTS**

## 1.Problem Definition:

An  optimization problem where the objective is to determine the optimal way of packing luggage for travelling with a limited capacity so as to maximize the value of the items packed inside .

## 2.Problem Explanation example:

Imagine you are going on a trip and you have a backpack with limited space. You want to pack as many items as possible, but you cannot exceed the weight capacity of the backpack. You have a list of items with their weights and values, and you want to choose the items that will maximize the total value while not exceeding the weight capacity of the backpack.

For example, let's say your backpack can hold a maximum weight of 10 kg. You have the following items to choose from:

Item 1: Weight = 5 kg, Value = 10rs

Item 2: Weight = 3 kg, Value = 12rs

Item 3: Weight = 2 kg, Value = 5rs

Item 4: Weight = 7 kg, Value = 8rs

Item 5: Weight = 1 kg, Value = 3rs

## Greedy Approach

Given

Item 1 : weight = 5kg , value = 10

Item 2 : weight = 3kg , value = 12

Item 3 : weight = 2kg , value = 5

Item 4 : weight = 7kg , value = 8

Item 5 : weight = 1kg , value = 3.

Bag weight = 10 kgs

No of objects = 5 Items.

| Items | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| profit | 10 | 12 | 5 | 8 | 3 |
| weight | 5 | 3 | 2 | 7 | 1 |
| profit/weight | 2 | 4 | 2.5 | 1.142 | 3 |

chose the items having Maximum profit/weight

| objects | profit | weight | pro/wi | Remaining weight |
|---|---|---|---|---|
| 2 | 12 | 3 | 4 | 10-3 = 7 |
| 5 | 3 | 1 | 3 | 7-1 = 6 |
| 3 | 5 | 2 | 2.5 | 6-2 = 4 |
| 1 | $10 \times \frac{4}{5} = \frac{40}{5} = 8$ | 4 | 2 | 4-4 = 0 |

Total profit = 12+3+5+8

Total profit = 28

Total Bag weight = 3+1+2+4

Total Bag weight = 10 ( chosen items )

While this solution may be good enough in some cases, it may not always be optimal. To ensure that the solution is indeed optimal, we would need to compare it against other possible solutions using different algorithms, such as dynamic programming or branch-and-bound.

## 0/1 Knapsack :-

Given data,

| Item | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| profit | 10 | 12 | 5 | 8 | 3 |
| weight | 5 | 3 | 2 | 7 | 1 |

chosing the items Having Maximum weight to fill the Bag.

| objects | profit | weight | Remaining weight |
|---|---|---|---|
| 2 | 12 | 3 | 10 − 3 = 7 |
| 1 | 10 | 5 | 7 − 5 = 2 |
| 4 | $8 \times \frac{2}{7} = 2{\cdot}28$ | 2 | 2 − 2 = 0 |
| Total | 24·28 | 10 | |

Total weight = 9

Total profit = 12 + 10 + 2·28.

Total profit = 24·28

| p | w | items \ weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 5 | 1 | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 12 | 3 | 2 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 5 | 2 | 3 | 0 | 0 | 0 | 5 | 12 | 12 | 17 | 17 | 17 | 17 | 17 |
| 8 | 7 | 4 | 0 | 0 | 5 | 12 | 12 | 17 | 17 | 17 | 20 | 20 | 20 |
| 3 | 1 | 5 | 0 | 3 | 5 | 12 | 15 | 17 | 20 | 22 | 22 | 22 | 22 |

The optimal solution for this problem is to select items 1, 2, 5 which Have total weight of 9 and profit 24·28

## 3.Design Techniques used:

1)Greedy approach

The basic idea of the greedy approach is to calculate the ratio **profit/weight** for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).
This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

2)Dynamic programming Approach

Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0/1 Knapsack problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computation of the same subproblems can be avoided by constructing a temporary array K[][] in a bottom-up manner.

## 4. Algorithm for the problem:

**Greedy Approach:**

## Algorithm for Greedy Approach

Algorithm Greedy - knapsack (ω, n)
input : n items with profit p[1 to n].
        weight ω[1 to n]
output : optimal packing order of items
        stored in solution vector x[1 to n]

```
Begin
    // Initialize the solution vector
    for i ← 1 to n . do
        x(i) = 0 ;
    End for
    load = 0 //Initial weight in knapsack
        i = 1
    while ((load < ω) & (i <= n)) do
            if ($ω_i$ + load ≤ ω) then
            load = load + $ω_i$ ;
            else
              r = ω - load ;
              load = load + $\frac{r}{ω_i}$
              x[i] = $\frac{r}{ω_i}$
                end if
              end while
                return (x)
              end.
```

**Dynamic Programming approach:**

1.In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and the element that can be kept as rows.
2.The state DP[i][j] will denote the maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider '$w_i$' (weight in 'ith' row) we can fill it in all columns which have 'weight values > $w_i$'. Now two possibilities can take place:
3.Fill '$w_i$' in the given column.
4.Do not fill '$w_i$' in the given column.
5.Now we have to take a maximum of these two possibilities,
6.Formally if we do not fill the 'ith' weight in the 'jth' column then the DP[i][j] state will be the same as DP[i-1][j]
7.But if we fill the weight, DP[i][j] will be equal to the value of ('$w_i$'+ value of the column weighing 'j-wi') in the previous row.
8.So we take the maximum of these two possibilities to fill the current state.

## <u>**5. Explanation of algorithm :**</u>

**The greedy approach** to solving the knapsack problem involves selecting the items with the highest value-to-weight ratio first, until the knapsack is full. This algorithm is called the "greedy" approach because at each step, it chooses the item that appears to be the best choice at that moment, without considering the long-term consequences of that choice.

Here is the algorithm for the knapsack problem using the greedy approach:

1.Calculate the value-to-weight ratio for each item.
2.Sort the items in descending order based on their value-to-weight ratio.
3.Initialize the total weight and total value of the knapsack to 0.
4.Loop through each item in the sorted list:
5. If the item can fit into the knapsack without exceeding its weight capacity, add it to the knapsack and update the total weight and total value.
6. If the item cannot fit into the knapsack, move on to the next item.
When all items have been considered, return the total value of the items in the knapsack.

**The dynamic programming** approach to solving the knapsack problem involves constructing a table where each cell represents the maximum value that can be obtained with a subset of the items, given a certain weight capacity of the knapsack. The table is filled in a bottom-up manner, starting with a table of size (n+1)x(W+1), where n is the number of items and W is the weight capacity of the knapsack.

Here is the explanation of algorithm for the knapsack problem using dynamic programming:

1.Initialize a table T of size (n+1)x(W+1) to all zeros, where n is the number of items and W is the weight capacity of the knapsack.
2.For each item i from 1 to n, and each weight capacity w from 1 to W, calculate the maximum value that can be obtained by considering two cases:

a. If the weight of the current item i is greater than the current weight capacity w, then the item cannot be included and the maximum value is the same as the maximum value obtained from the previous item i-1.

b. If the weight of the current item i is less than or equal to the current weight capacity w, then we can either include or exclude the item. If we include the item, the maximum value is the sum of the value of the current item i and the maximum value obtained by considering the remaining weight capacity w-wi and the previous items i-1. If we exclude the item, the maximum value is the same as the maximum value obtained from the previous item i-1. We choose the maximum of these two options.

3.The final result is the value in the bottom-right cell of the table T[n][W].

## 6.Complexity Analysis:

### Greedy Approach:

Time Complexity: **O(N * logN)**
Auxiliary Space: **O(N)**

### Dynamic Programming:

Time Complexity: O(N * W)
Auxiliary Space: O(2 * W)

## 7.CONCLUSION:

If the knapsack problem has the property of optimal substructure and exhibits overlapping subproblems, dynamic programming would be a better choice as it can guarantee finding the optimal solution. Dynamic programming has a higher time complexity than greedy algorithms, but it can handle more complex problems.

On the other hand, if the knapsack problem does not have the optimal substructure property, a greedy algorithm may be more suitable as it can provide an approximate solution in less time.

Therefore ,knapsack using Dynamic programming is efficient than greedy approach for this problem .