Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

[Code No: Comp 314]

Lab Report 03

Submitted by

Bishal Chaudhary (Roll No. 10)

Submitted to

Dr.Rajani Chulyadyo

Department of Computer Science and Engineering

Jan 23, 2025

Purpose:
Solving Knapsack problems using different algorithm design strategies.

Implementation:
Programming Language used: Python

Knapsack Problem:
The knapsack problem is a classic example of combinatorial optimization. The objective is to determine the optimal selection of items, each with a specific weight and value, such that the total weight does not exceed a predefined limit and the total value is maximized. This problem is named after the scenario of packing a fixed-size knapsack with the most valuable combination of items. It often appears in resource allocation scenarios where decision-makers need to select from a set of indivisible projects or tasks under constraints such as a fixed budget or time limit.

In the 0-1 Knapsack Problem, items cannot be divided; we must either include the entire item in the knapsack or exclude it altogether. On the other hand, the Fractional Knapsack Problem allows items to be broken into smaller parts, enabling us to maximize the total value by including fractions of items. This approach is also referred to as the fractional knapsack problem.

There are various methods to solve the knapsack problem. In this lab, we will explore three primary approaches:

1. Brute-Force Method: Examines all possible combinations to find the optimal solution.
2. Greedy Method: Focuses on a heuristic approach to maximize value by selecting items based on specific criteria (e.g., value-to-weight ratio).
3. Dynamic Programming: Employs a systematic and efficient way of solving the problem by breaking it into overlapping subproblems.

Brute-force method:
A brute force approach is an approach that finds all the possible
solutions to find a satisfactory solution to a given problem. The brute force
algorithm tries out all the possibilities till a satisfactory solution is not found.
We will use the brute force method to solve fractional and 0/1 Knapsack
Problem in this lab. Algorithms are as follows:-

a. Fractional Knapsack-

```
Take user input of Items (weight, value)

Take user input of knapsack_capacity (weight)

Define max_value <- 0, loop_count <- 0

Check (loop_count < Items.length) if false go to 11

Define weight_sum <- 0, value_sum<-0

Check (weight_sum + weight of item in Items <knapsack_capacity) if false goto 8

weight_sum <- weight_sum + weight of item in Items,value_sum <- value_sum +
value of item in Items

weight_sum <- weight_sum + fraction of weight of item in Items to make sum =
knapsack_capacity, value sum <- value sum + fraction of value of fractionated item

Check value_sum > max_value if false goto 9

max_value <- value_sum

loop_count <- loop_count + 1

goto 4

Output value_sum
```

b. 0/1 Knapsack-

ALGORITHM BruteForce (Weights [1 ... N], Values [1 ... N],

A[1...N])
Purpose: Find the best possible combination of items for the KP

Input: Array Weights contains the weights of all items

Array Values contains the values of all items
Array A initialised with 0s is used to generate the bit strings

Output: Best possible combination of items in the knapsack
bestChoice [1 .. N]

```
for i = 1 to 2
n do

j <- n
tempWeight <- 0
tempValue <- 0
while ( A[j] != 0 and j > 0)
A[j] <- 0
j <- j – 1
A[j] <- 1
for k <- 1 to n do
if (A[k] = 1) then
tempWeight <- tempWeight + Weights[k]
tempValue <- tempValue + Values[k]
if ((tempValue > bestValue) AND (tempWeight <= Capacity))
then
bestValue <- tempValue
bestWeight <- tempWeight
bestChoice <- A
return bestChoice
```

Greedy method:

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads global solutions are best fit for Greedy. We will solve Fractional Knapsack in this lab with this method.

```
Greedy-fractional-knapsack(w, v, w)
for i = 1 to n

        do x[i] <- 0

weight <- 0

while weight < W

        do i <- best remaining item

        if weight + w[i] <= W

                then x[i] <- 1

                weight <- weight + w[i]
Else
        x[i] <- (w - weight) / w[i]

        weight <- W

return x
```

Dynamic programming:

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilising the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems. We can also say that it is mainly an optimization over plain recursion. We simply store the results of subproblems,so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

```
ALGORITHM Dynamic Programming (Weights [1 ... N], Values [1 ... N],
Table [0 ... N, 0 ... Capacity])

Input: Array Weights contains the weights of all items
Array Values contains the values of all items
Array Table is initialised with 0s; it is used to store the results from
the dynamic programming algorithm.

Output: The last value of array Table (Table [N, Capacity]) contains the
optimal solution of the problem for the given Capacity

for i = 0 to N do
    for j = 0 to Capacity
        if j < Weights[i] then
            Table[i, j] <- Table[i-1, j]
    Else
        Table[i, j] <- maximum { Table[i-1, j] AND Values[i] +
        Table[i-1, j – Weights[i]] }

return Table[N, Capacity]
```

In the implementation of the algorithm instead of using two separate arrays for the weights and the values of the items, we used one array Items of type item, where item is a structure with two fields: weight and value. To find which items are included in the optimal solution, we use the following algorithm:

```
n <- N c <- Capacity
Start at position Table[n, c]
While the remaining capacity is greater than 0 do
    If Table[n, c] = Table[n-1, c] then
        Item n has not been included in the optimal solution
Else
        Item n has been included in the optimal solution
        Process Item n
        Move one row up to n-1
        Move to column c – weight(n)
```