Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

COMP 314

Algorithm and Complexity

Lab Report 3

Submitted by

Sushan Shrestha(54)

Submitted to

Dr. Rajani Chulyado

Department of Computer Science and Engineering

Submission Date

Jan 23, 2025

Objective: Solving Knapsack problem using different algorithm design strategies.

Knapsack Problem:

The Knapsack Problem is an example of a combinatorial optimization problem, which seeks for a best solution from among many other solutions. It is concerned with a knapsack that has positive integer volume (or capacity) V. There are n distinct items that may potentially be placed in the knapsack.

1. Brute-force method:

Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. If there are n items to choose from, then there will be 2 n possible combinations of items for the knapsack. An item is either chosen or not chosen. A bit string of 0's and 1's is generated which is of length n. If the i symbol of a bit string is 0, then the i item is not chosen and if it is 1, the i item is chosen.

Pseudocode

1. Brute-force method 0/1 Knapsack

```
BRUTE-FORCE-01-KNAPSACK(p, w, m)
        n ← length(p)
        bit_strings ← GET-STRINGS(n)
        max_profit ← 0
        solution ← empty string

        for each s in bit_strings do
                weight ← 0
                profit ← 0
                for i ← 0 to n - 1 do
                if s[i] = '1' then
                        weight ← weight + w[i]
                        profit ← profit + p[i]
                end if
                end for

                if weight ≤ m and profit > max_profit then
                        max_profit ← profit
                solution ← s
                end if
        end for

        return max_profit, solution
```

2. Brute-force method Fractional Knapsack

```
BRUTE-FORCE-FRACTIONAL-KNAPSACK(p, w, m)
        n ← length(p)
        max_profit ← 0
        total_weight ← m
        best_solution ← empty list
```

```
for i ← 0 to (2^n - 1) do
    s ← binary string of i, padded to length n
    profit ← 0
    weight ← 0
    fraction_solution ← empty list

    for j ← 0 to n - 1 do
        if s[j] = '1' then
            profit ← profit + p[j]
            weight ← weight + w[j]
            append 1 to fraction_solution
        else
            append 0 to fraction_solution
        end if
    end for

    if weight > total_weight then
        continue
    end if

    remaining_capacity ← total_weight - weight
    fractional_items ← list of items where s[j] = '0'
    total_fractional_weight ← sum(w[j] for each item in fractional_items)

    if total_fractional_weight > 0 then
        fraction ← remaining_capacity / total_fractional_weight
        for each item in fractional_items do
            profit ← profit + (fraction * p[item])
            weight ← weight + (fraction * w[item])
            fraction_solution[item] ← fraction
        end for
    end if

    if profit > max_profit then
        max_profit ← profit
        best_solution ← fraction_solution
    end if
end for

return max_profit, best_solution
```

2. Greedy method (Fractional Knapsack)

The basic idea of the greedy approach is to calculate the ratio value/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 until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

Pseudocode

```
GREEDY-FRACTIONAL-KNAPSACK(p, w, m)
    n ← length(p)
    ratios ← list of (p[i] / w[i], p[i], w[i]) for i from 0 to n-1
    sort ratios in descending order by the first element (profit/weight ratio)

    total_profit ← 0
    weight_left ← m
    solution ← empty list

    for each (ratio, profit, weight) in ratios do
        if weight_left ≤ 0 then
```

```
                Break
            end if

            if weight_left ≥ weight then
                total_profit ← total_profit + profit
                weight_left ← weight_left - weight
                append (1, profit, weight) to solution
            else
                total_profit ← total_profit + (profit * (weight_left / weight))
                append (weight_left / weight, profit, weight) to solution
                weight_left ← 0
            end if
        end for

        return total_profit
```

3. Dynamic programming (0/1 Knapsack)

Dynamic Programming is a technique for solving problems whose solutions satisfy recurrence relations with overlapping subproblems. Dynamic Programming solves each of the smaller subproblems only once and records the results in a table rather than solving overlapping subproblems over and over again. The table is then used to obtain a solution to the original problem.

Pseudocode

```
    DYNAMIC-PROGRAMMING-KNAPSACK(p, w, m)
        n ← length(p)
        dp ← 2D array of size (n+1) x (m+1) initialized to 0

        for i ← 1 to n do
            for j ← 1 to m do
                if w[i - 1] ≤ j then
                    dp[i][j] ← max(dp[i - 1][j], dp[i - 1][j - w[i - 1]] + p[i - 1])
                else
                    dp[i][j] ← dp[i - 1][j]
                end if
            end for
        end for

        return dp[n][m]
```

Conclusion:
Since all the methods we used to solve the Knapsack problem, we can conclude that the Greedy approach gives the best result as it is applied as a fractional problem. We have used a memorization method for dynamic programming and the function was successfully implemented and tested along with the Brute force method. We have used the unittest library to test all the codes and were found to be correct.

Source code link:
https://github.com/sushan08/Algorithm_labs/tree/main/Lab_3