Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Estd: 1991 A.D.

LAB-3

Algorithms and Complexity

[Code No: COMP 314]

Submitted by:

Nigam Niraula (32)

Submitted to:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submission Date: 06/21/2024

PSEUDOCODES:

1) Knapsack 0/1 Bruteforce:

The pseudocode for knapsack 0/1 bruteforce algorithm is:

```
function knapsack_non_fractional(weights, profit, capacity):
    length ← size of weights
    binary_combination ← empty list
    greatestProfit ← 0
    total_iterations ← 2 ^ length

    for i from 0 to total_iterations - 1 do:
        temp_binary_combination ← empty list
        temp_profit ← 0
        temp_capacity ← capacity
        n ← i

        for j from 0 to length - 1 do:
            append (n % 2) to temp_binary_combination
            if (n % 2) is 1 then:
                temp_profit ← temp_profit + profit[j]
                temp_capacity ← temp_capacity - weights[j]
            n ← n // 2

        if temp_profit > greatestProfit and temp_capacity ≥ 0 then:
            binary_combination ← copy of temp_binary_combination
            greatestProfit ← temp_profit

    return binary_combination
```

The `knapsack_non_fractional` function employs a brute-force approach to solve the 0/1 knapsack problem by exhaustively evaluating all possible combinations of items. It initializes with variables like the length of the item lists, an empty list to store the best combination (`binary_combination`), and a variable (`greatestProfit`) to track the highest profit found. The function calculates `total_iterations` as 2 raised to the power of the number of items, representing all potential subsets of items. It iterates through each subset using a binary representation where each bit indicates whether an item is included or excluded. For each subset, it computes the total profit (`temp_profit`) and remaining capacity (`temp_capacity`) of the knapsack. If `temp_profit` exceeds `greatestProfit` and `temp_capacity` remains non-negative (indicating the subset fits within the knapsack's capacity), it updates `binary_combination` and `greatestProfit` accordingly. Ultimately, the function returns `binary_combination`, representing the optimal selection of items that maximizes profit while adhering to the knapsack's capacity constraint. This method ensures all possible combinations are considered, guaranteeing an optimal solution for the problem at hand.

The source code was then tested with different test cases which yielded following result:

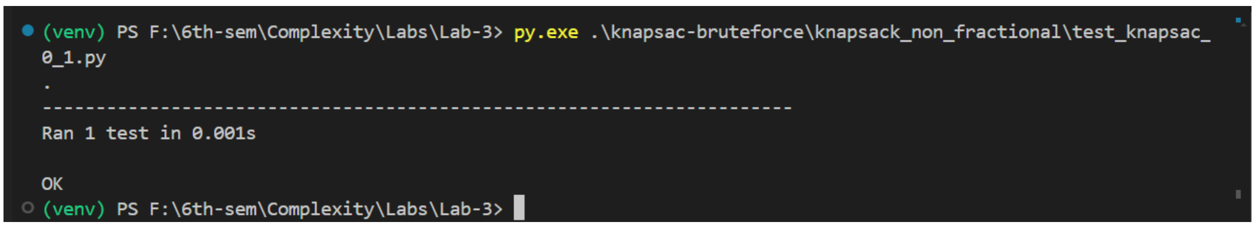

Fig: Test Cases On Bruteforce Knapsack 0/1

2) Knapsack Fractional Bruteforce:

The pseudocode for knapsack fractional bruteforce algorithm is:

```
1. function knapsack_fractional(weights, profits, capacity):
2.     length ← size of weights
3.     best_combination ← list of zeros of size length
4.     greatest_profit ← 0.0
5.     remaining_capacity ← 0.0

6.     total_iterations ← 2 ^ length

7.     for i from 0 to total_iterations - 1 do:
8.         temp_combination ← list of zeros of size length
9.         temp_profit ← 0.0
10.        temp_capacity ← capacity
11.        n ← i
```

```
    for j from 0 to length - 1 do:
        i. curr_binary ← n % 2
        ii. n ← n // 2
        iii. if curr_binary is 1 then:
            if weights[j] ≤ temp_capacity then:
                temp_combination[j] ← 1
                temp_profit ← temp_profit + profits[j]
                temp_capacity ← temp_capacity - weights[j]

    if temp_capacity > 0 then:
        frac_best_combination ← copy of temp_combination
        frac_best_profit ← temp_profit

        for k from 0 to length - 1 do:
            if temp_combination[k] is 0 then:
                i. fraction ← min(1, temp_capacity / weights[k])
                ii. frac_temp_profit ← temp_profit + fraction * profits[k]

                if frac_temp_profit > frac_best_profit then:
                    frac_best_profit ← frac_temp_profit
                    frac_best_combination ← copy of temp_combination
                    frac_best_combination[k] ← fraction

        temp_combination ← frac_best_combination
        temp_profit ← frac_best_profit
        temp_capacity ← 0

    if temp_profit > greatest_profit then:
        best_combination ← copy of temp_combination
        greatest_profit ← temp_profit
        remaining_capacity ← temp_capacity

return best_combination, remaining_capacity, greatest_profit
```

The function first initializes the necessary variables, including the length of the item list, the best combination of items, the greatest profit, and the remaining capacity. It then iterates through all possible combinations of items (2^length) using a binary representation to decide whether to include each item in the knapsack. For each combination, it calculates the total profit and capacity. If there's remaining capacity, the function tries to fill it with fractions of the items not yet included to maximize the profit further. The function keeps track of the best combination and greatest profit found during these iterations. Finally, it returns the best combination of items, the remaining capacity, and the greatest profit.
The source code was then tested with different test cases which yielded following result:

Fig: Test Cases On Bruteforce Fractional Knapsack

3) Knapsack Greedy Fractional:

The pseudocode for knapsack fractional greedy algorithm is:

1. function knapsack_fractional(weights, profits, capacity):
2. n ← length of weights

3. profit_density ← empty list
4. for i from 0 to n - 1 do:
5. append (profits[i] / weights[i], weights[i], profits[i], i) to profit_density

6. sort profit_density in descending order by the first element (profit density)

7. total_profit ← 0.0
8. knapsack ← list of zeros of size n

9. for i from 0 to n - 1 do:
10. if capacity ≤ 0 then:

```
        break

    density, weight, profit, original_index ← profit_density[i]

    if weight ≤ capacity then:
        knapsack[original_index] ← 1.0
        total_profit ← total_profit + profit
        capacity ← capacity - weight
    else:
        knapsack[original_index] ← capacity / weight
        total_profit ← total_profit + knapsack[original_index] * profit
        capacity ← 0

return knapsack, total_profit
```

The pseudocode first calculates the profit density (profit-to-weight ratio) for each item and sorts the items in descending order based on this ratio. It then iterates through the sorted items, adding each item fully to the knapsack if its weight is less than or equal to the remaining capacity. If an item's weight exceeds the remaining capacity, it adds the maximum possible fraction of that item to the knapsack. The process continues until the knapsack reaches its capacity. The function returns a list indicating the fraction of each item included in the knapsack and the total profit achieved. This approach ensures that the most profitable items per unit weight are prioritized, maximizing the total profit.
The source code was then tested with different test cases which yielded following result:

```
(venv) PS F:\6th-sem\Complexity\Labs\Lab-3> py.exe .\knapsack_greedy\test_knapsack_greedy.py
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
(venv) PS F:\6th-sem\Complexity\Labs\Lab-3>
```

Fig: Test Cases On Greedy Fractional Knapsack

4) Knapsack 0/1 Dynamic Programming:

The pseudocode for knapsack fractional bruteforce algorithm is:

```
function make_table(weights, values, capacity):
```

```
    n ← length of weights
    dp_table ← 2D list of size (n + 1) x (capacity + 1) filled with 0
    for i from 1 to n do:
        for w from 1 to capacity do:
            if weights[i - 1] ≤ w then:
                dp_table[i][w] ← max(dp_table[i - 1][w], values[i - 1] + dp_table[i - 1][w - weights[i - 1]])
            else:
                dp_table[i][w] ← dp_table[i - 1][w]
    return dp_table

function find_included_items(weights, values, capacity):
    dp_table ← make_table(weights, values, capacity)
    n ← length of weights
    max_value ← dp_table[n][capacity]
    capacity ← length of dp_table[0] - 1
    included_items ← empty list
    i, w ← n, capacity

    while i > 0 and w > 0 do:
        if dp_table[i][w] ≠ dp_table[i - 1][w] then:
            append i to included_items
            w ← w - weights[i - 1]
        i ← i - 1

    reverse included_items
    return included_items, max_value
```

Here, the `make_table` function creates a dynamic programming table (`dp_table`) to solve the 0/1 knapsack problem, where each cell `dp_table[i][w]` represents the maximum value achievable with the first `i` items and a knapsack capacity of `w`. It iterates through each item and weight, filling the table based on whether including the current item yields a higher value than excluding it. The `find_included_items` function uses this table to determine which items are included in the optimal solution. It traces back from the maximum value in the table, checking which items were included by

comparing values in the table, and constructs a list of these items, which it then returns along with the maximum value.
The source code was then tested with different test cases which yielded following result:

```
(venv) PS F:\6th-sem\Complexity\Labs\Lab-3> py.exe .\knapsac_dynamic\test_knapsac_dynamic.py
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Fig: Test Cases On Dynamic Programming Knapsack 0/1