Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Algorithm and Complexities Lab Report 03

on

‘Brute-force, Greedy and Dynamic Programming implementation in 0/1 and Fractional KnapSack’

Submitted By:

Reewaj Khanal (61)

Submitted to:

Dr. Rajani Chulyadyo

Assistant Professor

Department of Computer Science and Engineering

School of Engineering

Kathmandu University

Dhulikhel, Kavre

Submission Date: Thursday 20 June 2024

1. Purpose

Solving Knapsack problems using different algorithm design strategies.

2. Tasks

Solving the Knapsack problem using the following strategies:

Pseudocodes for each of these methods along with the source code of my program (and test cases).

2.1. Brute-force method (Both fractional and 0/1 Knapsack)

2.1.1. Brute-force Fractional Knapsack

2.1.1.1. Pseudocode:

```
FRACTIONAL-KNAPSACK-BRUTE-FORCE(profits, weights, capacity)

n = profits.length // Number of items

max_profit = 0 // Initialize maximum profit

best_fractions = new Array(n).fill(0) // Array to store best fractions

for each possible combination of fractions:

     current_profit = 0

     current_weight = 0

     for j = 0 to n - 1

          current_profit = current_profit + profits[j] * fractions[j]
```

```
            current_weight = current_weight + weights[j] * fractions[j]
        if current_weight ≤ capacity AND current_profit > max_profit
            max_profit = current_profit
            best_fractions = fractions.copy()
    return max_profit, best_fractions
```

2.1.1.2. Code:

```python
from tabulate import tabulate


def bfmKSfract(p, w, m):
    # Ensure that the profit list and weight list are of the same length
    assert len(p) == len(w), "p and w differ"


    # Get the number of items
    n = len(p)
    # Initialize the maximum profit to 0
    max_profit = 0
    # Set the total weight limit to m
    total_weight = m
    # Initialize the solution string to an empty string
    soln = ''
```

```
    # Initialize table data list
    table_data = []


    # Iterate through all possible combinations of items
    for i in range(2**n):
        # Convert the current combination number to a binary
string
        s = bin(i)[2:].rjust(n, '0')
        # Initialize profit and weight to 0
        profit = 0
        weight = 0
        # Calculate profit and weight for the current
combination
        for j in range(n):
            if s[j] == '1':
                profit += p[j]
                weight += w[j]


        # Calculate fractional weight for '0' elements in s
        if weight > total_weight:
            continue


        remaining_weight = total_weight - weight
        if remaining_weight > 0:
```

```
            total_zero_weight = sum(w[j] for j in range(n)
if s[j] == '0')

            if total_zero_weight > 0:

                fraction = remaining_weight /
total_zero_weight

                profit += fraction * sum(p[j] for j in
range(n) if s[j] == '0')

                weight += fraction * total_zero_weight



        # Append combination, profit, and weight to table
data

        table_data.append([s, round(profit, 2),
round(weight, 2)])



        # Update maximum profit and solution if the current
combination yields higher profit

        if profit > max_profit:

            max_profit = profit

            soln = s



    # Print the combinations, profits, and weights in a
tabular format

    print(tabulate(table_data, headers=['Combination',
'Profit', 'Weight'], tablefmt='grid'))



    # Print the best combination and the maximum profit

    print(f"\nBest combination: {soln}")
```

```
    print(f"Maximum profit: {int(max_profit)}")


    # Return the maximum profit found (converted to integer)

    return int(max_profit)


# Function to run test cases automatically

def run_test_cases():

    # Test case 1: Basic example

    p1 = [60, 100, 120]

    w1 = [10, 20, 30]

    m1 = 50

    print("Test Case 1:")

    bfmKSfract(p1, w1, m1)


    # Test case 2: All items can be selected

    p2 = [10, 20, 30]

    w2 = [1, 2, 3]

    m2 = 6

    print("\nTest Case 2:")

    bfmKSfract(p2, w2, m2)


    # Test case 3: Fractional selection (similar behavior to
test edge case)

    p3 = [10, 20, 30]
```

```python
w3 = [2, 3, 4]

m3 = 5

print("\nTest Case 3:")

bfmKSfract(p3, w3, m3)



# Test case 4: Edge case with zero capacity

p4 = [10, 20, 30]

w4 = [1, 2, 3]

m4 = 0

print("\nTest Case 4:")

bfmKSfract(p4, w4, m4)



# Uncomment the following test cases to use them in the
future



# Test case 5: High profit low weight items

# p5 = [100, 200, 300]

# w5 = [1, 2, 3]

# m5 = 4

# print("\nTest Case 5:")

# bfmKSfract(p5, w5, m5)



# Test case 6: Low profit high weight items

# p6 = [10, 20, 30]
```

```python
    # w6 = [10, 20, 30]

    # m6 = 15

    # print("\nTest Case 6:")

    # bfmKSfract(p6, w6, m6)



# Run the test cases

if __name__ == "__main__":

    run_test_cases()
```

2.1.1.2.1. Inputs:

```
(base) reewajkhanal.rk10@RK10 LAB03 % python bfmKSfracttestcases.py
```

2.1.1.2.2. Outputs:

```
Test Case 1:
+---------------+-----------+-----------+
|   Combination |    Profit |    Weight |
+===============+===========+===========+
|           000 |    233.33 |        50 |
+---------------+-----------+-----------+
|           001 |    226.67 |        50 |
+---------------+-----------+-----------+
|           010 |    235    |        50 |
+---------------+-----------+-----------+
|           011 |    220    |        50 |
+---------------+-----------+-----------+
|           100 |    236    |        50 |
+---------------+-----------+-----------+
|           101 |    230    |        50 |
+---------------+-----------+-----------+
|           110 |    240    |        50 |
+---------------+-----------+-----------+

Best combination: 110
Maximum profit: 240

Test Case 2:
+---------------+-----------+-----------+
|   Combination |    Profit |    Weight |
+===============+===========+===========+
|           000 |        60 |         6 |
+---------------+-----------+-----------+
|           001 |        60 |         6 |
+---------------+-----------+-----------+
|           010 |        60 |         6 |
+---------------+-----------+-----------+
|           011 |        60 |         6 |
+---------------+-----------+-----------+
|           100 |        60 |         6 |
+---------------+-----------+-----------+
|           101 |        60 |         6 |
+---------------+-----------+-----------+
|           110 |        60 |         6 |
+---------------+-----------+-----------+
|           111 |        60 |         6 |
+---------------+-----------+-----------+

Best combination: 000
Maximum profit: 60
```

```
Test Case 3:
+---------------+-----------+-----------+
|   Combination |    Profit |    Weight |
+===============+===========+===========+
|           000 |    33.33  |         5 |
+---------------+-----------+-----------+
|           001 |    36     |         5 |
+---------------+-----------+-----------+
|           010 |    33.33  |         5 |
+---------------+-----------+-----------+
|           100 |    31.43  |         5 |
+---------------+-----------+-----------+
|           110 |    30     |         5 |
+---------------+-----------+-----------+

Best combination: 001
Maximum profit: 36

Test Case 4:
+---------------+-----------+-----------+
|   Combination |    Profit |    Weight |
+===============+===========+===========+
|           000 |         0 |         0 |
+---------------+-----------+-----------+

Best combination:
Maximum profit: 0
```

2.1.2. Brute-force 0/1 Knapsack

2.1.2.1. Pseudocode:

```
BRUTE-FORCE-KNAPSACK(profits, weights, capacity)

n = profits.length // Number of items

max_profit = 0 // Initialize maximum profit

best_items = [] // List to track included items

for i = 0 to 2^n - 1 // Iterate through all combinations

    current_profit = 0

    current_weight = 0

    items_included = []

    for j = 0 to n - 1 // Check each item in the combination

        if (i >> j) & 1 == 1 // If jth bit in i is 1, item is included

            current_profit = current_profit + profits[j]

            current_weight = current_weight + weights[j]

            items_included.append(True)

        else

            items_included.append(False)
```

```
        if current_weight ≤ capacity AND current_profit > max_profit
            max_profit = current_profit
            best_items = items_included
    return max_profit, best_items
```

2.1.2.2. Code:

```
from tabulate import tabulate


def bfmKS01(p, w, m):
    # Ensure that the profit list and weight list are of the same length
    assert len(p) == len(w), "p and w differ"


    # Get the number of items
    n = len(p)
    # Initialize the maximum profit to 0
    max_profit = 0
    # Set the total weight limit to m
    total_weight = m
    # Initialize the solution string to an empty string
    soln = ''


    # Initialize table data list
    table_data = []
```

```
    # Iterate through all possible combinations of items
    for i in range(2**n):
        # Convert the current combination number to a binary string
        s = bin(i)[2:].rjust(n, '0')
        # Calculate the total profit for the current combination
        profit = sum((int(s[j])) * p[j] for j in range(n))
        # Calculate the total weight for the current combination
        weight = sum((int(s[j])) * w[j] for j in range(n))

        # Append combination, profit, and weight to table data
        table_data.append([s, profit, weight])

        # If the current combination's profit is greater than the max profit
        # and its weight is within the allowed limit, update the max profit and solution
        if profit > max_profit and weight <= total_weight:
            max_profit = profit
            soln = s

    # Print the combinations, profits, and weights in a tabular format
```

```
    print(tabulate(table_data, headers=['Combination',
'Profit', 'Weight'], tablefmt='grid'))

    # Print the best combination and the maximum profit
    print(f"\nBest combination: {soln}")
    print(f"Maximum profit: {max_profit}")

    # Return the maximum profit found
    return max_profit

# Function to run test cases automatically
def run_test_cases():
    # Test case 1: Basic example
    p1 = [60, 100, 120]
    w1 = [10, 20, 30]
    m1 = 50
    print("Test Case 1:")
    bfmKS01(p1, w1, m1)

    # Test case 2: All items can be selected
    p2 = [10, 20, 30]
    w2 = [1, 2, 3]
    m2 = 6
    print("\nTest Case 2:")
```

```
    bfmKS01(p2, w2, m2)



    # Test case 3: Fractional selection (similar behavior to
test edge case)

    p3 = [10, 20, 30]

    w3 = [2, 3, 4]

    m3 = 5

    print("\nTest Case 3:")

    bfmKS01(p3, w3, m3)



    # Test case 4: Edge case with zero capacity

    p4 = [10, 20, 30]

    w4 = [1, 2, 3]

    m4 = 0

    print("\nTest Case 4:")

    bfmKS01(p4, w4, m4)



    # Uncomment the following test cases to use them in the
future



    # Test case 5: High profit low weight items

    # p5 = [100, 200, 300]

    # w5 = [1, 2, 3]

    # m5 = 4

    # print("\nTest Case 5:")
```

```python
    # bfmKS01(p5, w5, m5)


    # Test case 6: Low profit high weight items

    # p6 = [10, 20, 30]

    # w6 = [10, 20, 30]

    # m6 = 15

    # print("\nTest Case 6:")

    # bfmKS01(p6, w6, m6)


# Run the test cases

if __name__ == "__main__":

    run_test_cases()
```

2.1.2.2.1. Inputs:

```
(base) reewajkhanal.rk10@RK10 LAB03 % python bfmKS01testcases.py
```

2.1.2.2.2. Outputs:

Test Case 1:

Combination	Profit	Weight
000	0	0
001	120	30
010	100	20
011	220	50
100	60	10
101	180	40
110	160	30
111	280	60

Best combination: 011
Maximum profit: 220

Test Case 2:

Combination	Profit	Weight
000	0	0
001	30	3
010	20	2
011	50	5
100	10	1
101	40	4
110	30	3
111	60	6

Best combination: 111
Maximum profit: 60

Test Case 3:

Combination	Profit	Weight
000	0	0
001	30	4
010	20	3
011	50	7
100	10	2
101	40	6
110	30	5
111	60	9

Best combination: 001
Maximum profit: 30

Test Case 4:

Combination	Profit	Weight
000	0	0
001	30	3
010	20	2
011	50	5
100	10	1
101	40	4
110	30	3
111	60	6

Best combination:
Maximum profit: 0

2.2. Greedy method (Fractional Knapsack)

2.2.1. Pseudocode:

```
FRACTIONAL-KNAPSACK-GREEDY(profits, weights, capacity)

n = profits.length // Number of items

ratio = new Array(n) // Array to store profit/weight ratios

for i = 0 to n - 1

    ratio[i] = profits[i] / weights[i] // Calculate profit/weight ratio

sorted_indices = argsort(ratio, reverse=True) // Sort items by ratio in descending order

max_profit = 0

current_weight = 0

for i in sorted_indices

    if current_weight + weights[i] ≤ capacity // Can take the entire item

        current_weight = current_weight + weights[i]

        max_profit = max_profit + profits[i]

    else // Can only take a fraction

        remaining = capacity - current_weight

        fraction = remaining / weights[i]

        max_profit = max_profit + profits[i] * fraction
```

```
            break // Knapsack is full
    return max_profit
```

2.2.2. Code:

```python
from tabulate import tabulate



def gmKSfract(p, w, m):

    # Ensure that the profit list and weight list are of the same length

    assert len(p) == len(w), "p and w differ"



    # Get the number of items

    n = len(p)

    items = list(range(n))

    # Sort the items in descending order of their profit/weight ratio

    items.sort(key=lambda i: p[i] / w[i], reverse=True)



    total_weight = 0

    max_profit = 0



    # Initialize table data list

    table_data = []



    for i in items:
```

```
        # Check if adding the entire item exceeds the capacity
        if total_weight + w[i] <= m:
            total_weight += w[i]
            max_profit += p[i]
            # Append selected item's data to table data with fraction 1 indicating the entire item is selected
            table_data.append([i+1, p[i], w[i], 1.0])
        else:
            # Calculate the remaining capacity
            remaining_weight = m - total_weight
            # Calculate the fraction of the item that can be added to the knapsack
            fraction = remaining_weight / w[i]
            # Update the profit and weight considering the fraction
            max_profit += p[i] * fraction
            # Append selected item's data to table data with the calculated fraction
            table_data.append([i+1, p[i], w[i], round(fraction, 2)])

            # Break the loop as the knapsack is now full
            break

    # Print the items selected and their fractions (if any) in a tabular format
    print(tabulate(table_data, headers=['Item', 'Profit', 'Weight', 'Fraction'], tablefmt='grid'))
```

```python
    # Print the maximum profit
    print(f"\nMaximum profit: {int(max_profit)}")

    # Return the maximum profit found (converted to integer)
    return int(max_profit)

# Function to run test cases automatically
def run_test_cases():
    # Test case 1: Basic example
    p1 = [60, 100, 120]
    w1 = [10, 20, 30]
    m1 = 50
    print("Test Case 1:")
    gmKSfract(p1, w1, m1)

    # Test case 2: All items can be selected
    p2 = [10, 20, 30]
    w2 = [1, 2, 3]
    m2 = 6
    print("\nTest Case 2:")
    gmKSfract(p2, w2, m2)

    # Test case 3: Fractional selection
```

```
p3 = [10, 20, 30]

w3 = [2, 3, 4]

m3 = 5

print("\nTest Case 3:")

gmKSfract(p3, w3, m3)



# Test case 4: Edge case with zero capacity

p4 = [10, 20, 30]

w4 = [1, 2, 3]

m4 = 0

print("\nTest Case 4:")

gmKSfract(p4, w4, m4)



# Uncomment the following test cases to use them in the future



# Test case 5: High profit low weight items

# p5 = [100, 200, 300]

# w5 = [1, 2, 3]

# m5 = 4

# print("\nTest Case 5:")

# gmKSfract(p5, w5, m5)



# Test case 6: Low profit high weight items

# p6 = [10, 20, 30]
```

```python
    # w6 = [10, 20, 30]

    # m6 = 15

    # print("\nTest Case 6:")

    # gmKSfract(p6, w6, m6)



# Run the test cases

if __name__ == "__main__":

    run_test_cases()
```

2.2.2.1. Inputs:

```
(base) reewajkhanal.rk10@RK10 LAB03 % python gmKSfracttestcases.py
```

2.2.2.2. Outputs:

```
Test Case 1:
+--------+----------+----------+------------+
|   Item |   Profit |   Weight |   Fraction |
+========+==========+==========+============+
|      1 |       60 |       10 |     1      |
+--------+----------+----------+------------+
|      2 |      100 |       20 |     1      |
+--------+----------+----------+------------+
|      3 |      120 |       30 |     0.67   |
+--------+----------+----------+------------+

Maximum profit: 240

Test Case 2:
+--------+----------+----------+------------+
|   Item |   Profit |   Weight |   Fraction |
+========+==========+==========+============+
|      1 |       10 |        1 |          1 |
+--------+----------+----------+------------+
|      2 |       20 |        2 |          1 |
+--------+----------+----------+------------+
|      3 |       30 |        3 |          1 |
+--------+----------+----------+------------+

Maximum profit: 60

Test Case 3:
+--------+----------+----------+------------+
|   Item |   Profit |   Weight |   Fraction |
+========+==========+==========+============+
|      3 |       30 |        4 |     1      |
+--------+----------+----------+------------+
|      2 |       20 |        3 |     0.33   |
+--------+----------+----------+------------+

Maximum profit: 36

Test Case 4:
+--------+----------+----------+------------+
|   Item |   Profit |   Weight |   Fraction |
+========+==========+==========+============+
|      1 |       10 |        1 |          0 |
+--------+----------+----------+------------+

Maximum profit: 0
```

2.3. Dynamic programming (0/1 Knapsack)

2.3.1. 0/1 Knapsack using Tabulation - Bottom Up (Iterative) Method

2.3.1.1. Pseudocode:

```
KNAPSACK_01_TABULATION(profits, weights, capacity)

n = profits.length // Number of items
dp = new table[n+1][capacity+1] // Create a table to store solutions
for i = 0 to n
    for w = 0 to capacity
        if i == 0 OR w == 0 // Base case: no items or no capacity
            dp[i][w] = 0
        else if weights[i-1] > w // Current item's weight exceeds capacity
            dp[i][w] = dp[i-1][w] // Exclude the current item
        else // Choose max value: include or exclude current item
            dp[i][w] = max(dp[i-1][w], profits[i-1] + dp[i-1][w - weights[i-1]])
max_profit = dp[n][capacity]
taken = new array[n] filled with false // Track which items are taken
```

```
w = capacity
for i = n downto 1
    if dp[i][w] != dp[i-1][w] // Item was included
        taken[i-1] = true
        w = w - weights[i-1]
return max_profit, taken // Return max profit and items taken
```

2.3.1.2. Code:

```python
import pandas as pd


# Function to solve the knapsack problem using Tabulation (Bottom-Up) approach
def knapsack_tabulation(wt, val, W, n):
    # Initialize a table to store results of subproblems
    t = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    # Build the table in bottom-up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                t[i][w] = 0
            elif wt[i - 1] <= w:
```

```
                t[i][w] = max(val[i - 1] + t[i - 1][w - wt[i
- 1]], t[i - 1][w])

            else:

                t[i][w] = t[i - 1][w]



   return t[n][W], t



# Function to print the memoization table in tabular format
using pandas

def print_tabulation_table(table, n, W):

   df = pd.DataFrame(table)

   df.index = ["Item " + str(i) for i in range(n + 1)]

   df.columns = ["W" + str(i) for i in range(W + 1)]

   print("\nMemoization Table:")

   print(df)



# Function to run test cases

def run_test_cases():

   # Test case 1: Basic example

   profit1 = [60, 100, 120]

   weight1 = [10, 20, 30]

   W1 = 50

   n1 = len(profit1)

   max_profit1, memoization_table1 =
knapsack_tabulation(weight1, profit1, W1, n1)
```

```
    print(f"\nTest Case 1: Maximum profit = {max_profit1}")
    print_tabulation_table(memoization_table1, n1, W1)


    # # Test case 2: Edge case with zero capacity
    # profit2 = [10, 20, 30]
    # weight2 = [1, 2, 3]
    # W2 = 0
    # n2 = len(profit2)
    # max_profit2, memoization_table2 =
knapsack_tabulation(weight2, profit2, W2, n2)
    # print(f"\nTest Case 2: Maximum profit =
{max_profit2}")
    # print_tabulation_table(memoization_table2, n2, W2)


    # # Test case 3: All items have zero profit
    # profit3 = [0, 0, 0]
    # weight3 = [10, 20, 30]
    # W3 = 50
    # n3 = len(profit3)
    # max_profit3, memoization_table3 =
knapsack_tabulation(weight3, profit3, W3, n3)
    # print(f"\nTest Case 3: Maximum profit =
{max_profit3}")
    # print_tabulation_table(memoization_table3, n3, W3)


# Run the test cases
```

```python
if __name__ == "__main__":

    run_test_cases()
```

2.3.1.2.1. Inputs:

```
(base) reewajkhanal.rk10@RK10 LAB03 % python dp01KStestcasesTabulationBottomUpIterative.py
```

2.3.1.2.2. Outputs:

```
Test Case 1: Maximum profit = 220

Memoization Table:
        W0  W1  W2  W3  W4  W5  W6  W7  W8  W9  W10  W11  W12  W13  W14  W15  W16  ...  W34  W35  W36  W37  W38  W39  W40  W41  W42  W43  W44  W45  W46  W47  W48  W49
W50
Item 0   0   0   0   0   0   0   0   0   0   0    0    0    0    0    0    0    0  ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
  0
Item 1   0   0   0   0   0   0   0   0   0   0   60   60   60   60   60   60   60  ...   60   60   60   60   60   60   60   60   60   60   60   60   60   60   60   60
 60
Item 2   0   0   0   0   0   0   0   0   0   0   60   60   60   60   60   60   60  ...  160  160  160  160  160  160  160  160  160  160  160  160  160  160  160  160
160
Item 3   0   0   0   0   0   0   0   0   0   0   60   60   60   60   60   60   60  ...  160  160  160  160  160  160  180  180  180  180  180  180  180  180  180  180
220

[4 rows x 51 columns]
```

2.3.2. 0/1 Knapsack using Memoization - Top Down (Recursive) Method

2.3.2.1. Pseudocode:

```
KNAPSACK_01_MEMOIZATION(profits, weights, capacity, n, memo)

if memo[n][capacity] is not null // Check if the result is already calculated

    return memo[n][capacity] // Return the cached value

if n == 0 OR capacity == 0 // Base case: no items or no capacity

    result = 0
```

```
else if weights[n - 1] > capacity // Current item's weight exceeds capacity
    result = KNAPSACK_01_MEMOIZATION(profits, weights, capacity, n - 1, memo) // Exclude
else                                  // Choose max value: include or exclude current item
    include = profits[n - 1] + KNAPSACK_01_MEMOIZATION(profits, weights, capacity - weights[n - 1], n - 1, memo)
    exclude = KNAPSACK_01_MEMOIZATION(profits, weights, capacity, n - 1, memo)
    result = max(include, exclude)
memo[n][capacity] = result // Store the result in the memoization table
return result
```

2.3.2.2. Code:

```python
#MemoizationTopDownRecursive

from tabulate import tabulate



# Class to represent a node in the decision tree

class TreeNode:

   def __init__(self, n, W, profit, include=None):

       self.n = n  # The number of items being considered
```

```
        self.W = W  # The current capacity of the knapsack

        self.profit = profit  # The current profit at this
node

        self.include = include  # Whether the current item
is included (True/False)

        self.left = None  # Left child node (include the
current item)

        self.right = None  # Right child node (exclude the
current item)



# Function to solve the knapsack problem using memoization
and build the decision tree

def knapsack(wt, val, W, n, root):

    # Base conditions: if there are no items or the capacity
of the knapsack is 0, return 0

    if n == 0 or W == 0:

        return 0



    # Check if the result is already in the memoization
table

    if t[n][W] != -1:

        return t[n][W]



    # If the weight of the nth item is less than or equal to
the knapsack capacity W,

    # we have two options: include the nth item or exclude
it

    if wt[n-1] <= W:
```

```
        # Create left child (include the item)

        root.left = TreeNode(n-1, W-wt[n-1], val[n-1], True)

        include_profit = val[n-1] + knapsack(wt, val,
W-wt[n-1], n-1, root.left)

        # Create right child (exclude the item)

        root.right = TreeNode(n-1, W, 0, False)

        exclude_profit = knapsack(wt, val, W, n-1,
root.right)

        # Store the maximum of including or excluding the
item

        t[n][W] = max(include_profit, exclude_profit)

    else:

        # If the weight of the nth item is greater than the
knapsack capacity W,

        # we cannot include the nth item in the knapsack

        root.right = TreeNode(n-1, W, 0, False)

        t[n][W] = knapsack(wt, val, W, n-1, root.right)



    # Return the value stored in the memoization table

    return t[n][W]



# Function to print the decision tree in a tree-like format

def print_tree(node, indent="", branch="Root"):

    if node is not None:

        print(f"{indent}└── {branch}: Item {node.n},
Capacity {node.W}, Profit {node.profit}, Include
{node.include}")
```

```python
        if node.left:
            print_tree(node.left, indent + "    ", "Left")
        if node.right:
            print_tree(node.right, indent + "    ", "Right")


# Function to run test cases
def run_test_cases():
    # Test case 1: Example from textbooks or online
tutorials
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    W = 50
    n = len(profit)

    global t
    t = [[-1 for i in range(W + 1)] for j in range(n + 1)]

    root = TreeNode(n, W, 0)
    max_profit = knapsack(weight, profit, W, n, root)

    print(f"\nTest Case 1: Maximum profit = {max_profit}\n")
    print("Decision Tree:")
    print_tree(root)
```

```
    # # Test case 2: Custom scenario with different values

    # profit = [70, 20, 39, 100, 50]

    # weight = [31, 10, 20, 40, 50]

    # W = 100

    # n = len(profit)



    # t = [[-1 for i in range(W + 1)] for j in range(n + 1)]



    # root = TreeNode(n, W, 0)

    # max_profit = knapsack(weight, profit, W, n, root)



    # print(f"\nTest Case 2: Maximum profit =
{max_profit}\n")

    # print("Decision Tree:")

    # print_tree(root)



    # # Test case 3: Edge case with very small capacity and
weights

    # profit = [10, 20, 30]

    # weight = [1, 2, 3]

    # W = 2

    # n = len(profit)



    # t = [[-1 for i in range(W + 1)] for j in range(n + 1)]
```

```
    # root = TreeNode(n, W, 0)

    # max_profit = knapsack(weight, profit, W, n, root)



    # print(f"\nTest Case 3: Maximum profit =
{max_profit}\n")

    # print("Decision Tree:")

    # print_tree(root)



if __name__ == '__main__':

    run_test_cases()
```

2.3.2.2.1. Inputs:

```
(base) reewajkhanal.rk10@RK10 LAB03 % python dp01KStestcasesMemoizationTopDownRecursive.py
```

2.3.2.2.2. Outputs:

```
Test Case 1: Maximum profit = 220

Decision Tree:
└── Root: Item 3, Capacity 50, Profit 0, Include None
    └── Left: Item 2, Capacity 20, Profit 120, Include True
        └── Left: Item 1, Capacity 0, Profit 100, Include True
        └── Right: Item 1, Capacity 20, Profit 0, Include False
            └── Left: Item 0, Capacity 10, Profit 60, Include True
            └── Right: Item 0, Capacity 20, Profit 0, Include False
    └── Right: Item 2, Capacity 50, Profit 0, Include False
        └── Left: Item 1, Capacity 30, Profit 100, Include True
            └── Left: Item 0, Capacity 20, Profit 60, Include True
            └── Right: Item 0, Capacity 30, Profit 0, Include False
        └── Right: Item 1, Capacity 50, Profit 0, Include False
            └── Left: Item 0, Capacity 40, Profit 60, Include True
            └── Right: Item 0, Capacity 50, Profit 0, Include False
```