

Assignment No: 4

Title of the Assignment: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Objective of the Assignment: Students should be able to understand and solve 0-1 Knapsack problem using dynamic programming

Prerequisite:

1. Basic of Python or Java Programming
 2. Concept of Dynamic Programming
 3. 0/1 Knapsack problem
-

Contents for Theory:

1. Greedy Method
 2. 0/1 Knapsack problem
 3. Example solved using 0/1 Knapsack problem
-

What is Dynamic Programming?

- Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems.
- Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.
- Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.
- For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps –

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Knapsack Problem

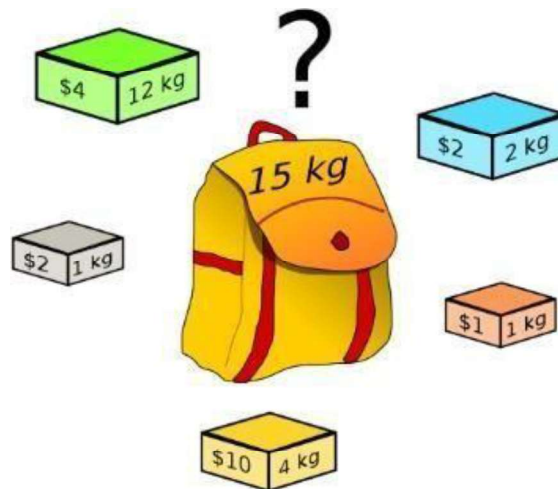
You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem

Knapsack Problem Variants

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

0/1 Knapsack Problem-

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible here.
- We can not take a fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using a dynamic programming approach.

0/1 Knapsack Problem Using Greedy Method-

Consider-

- Knapsack weight capacity = w
- Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01:

- Draw a table say „T“ with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0th row and 0th column with zeroes as shown-

	0	1	2	3	W
0	0	0	0	0	0
1	0					
2	0					
.....						
n	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$$

Here, $T(i, j)$ = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-03:

- To identify the items that must be put into the knapsack to obtain that maximum profit,
- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack

Problem-.

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of a dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

$$n = 4$$

$$w = 5 \text{ kg}$$

$$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$$

$$(b_1, b_2, b_3, b_4) = (3, 4, 5, 6)$$

Solution-

Given

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

Step-01:

- Draw a table say „T“ with $(n+1) = 4 + 1 = 5$ number of rows and $(w+1) = 5 + 1 = 6$ number of columns.
- Fill all the boxes of 0th row and 0th column with 0.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$$

Finding T(1,1)-

We have,

- $i = 1$
- $j = 1$
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,1) = \max \{ T(1-1, 1), 3 + T(1-1, 1-2) \}$$

$$T(1,1) = \max \{ T(0,1), 3 + T(0,-1) \}$$

$$T(1,1) = T(0,1) \{ \text{Ignore } T(0,-1) \}$$

$$T(1,1) = 0$$

Finding T(1,2)-

We have,

- $i = 1$
- $j = 2$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$$T(1,2) = \max \{ T(1-1, 2), 3 + T(1-1, 2-2) \}$$

$$T(1,2) = \max \{ T(0,2), 3 + T(0,0) \}$$

$$T(1,2) = \max \{ 0, 3+0 \}$$

$$T(1,2) = 3$$

Finding T(1,3)-

We have,

- $i = 1$
- $j = 3$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$$T(1,3) = \max \{ T(1-1, 3), 3 + T(1-1, 3-2) \}$$

$$T(1,3) = \max \{ T(0,3), 3 + T(0,1) \}$$

$$T(1,3) = \max \{ 0, 3+0 \}$$

$$T(1,3) = 3$$

Finding T(1,4)-

We have,

- $i = 1$
- $j = 4$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$$T(1,4) = \max \{ T(1-1, 4), 3 + T(1-1, 4-2) \}$$

$$T(1,4) = \max \{ T(0,4), 3 + T(0,2) \}$$

$$T(1,4) = \max \{ 0, 3+0 \}$$

$$T(1,4) = 3$$

Finding T(1,5)-

We have,

- $i = 1$
- $j = 5$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$$T(1,5) = \max \{ T(1-1, 5), 3 + T(1-1, 5-2) \}$$

$$T(1,5) = \max \{ T(0,5), 3 + T(0,3) \}$$

$$T(1,5) = \max \{ 0, 3+0 \}$$

$$T(1,5) = 3$$

Finding T(2,1)-

We have,

- $i = 2$
- $j = 1$
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

$$T(2,1) = \max \{ T(2-1, 1), 4 + T(2-1, 1-3) \}$$

$$T(2,1) = \max \{ T(1,1), 4 + T(1,-2) \}$$

$$T(2,1) = T(1,1) \text{ \{ Ignore } T(1,-2) \}$$

$$T(2,1) = 0$$

Finding T(2,2)-

We have,

- $i = 2$
- $j = 2$
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

$$T(2,2) = \max \{ T(2-1, 2), 4 + T(2-1, 2-3) \}$$

$$T(2,2) = \max \{ T(1,2), 4 + T(1,-1) \}$$

$$T(2,2) = T(1,2) \text{ \{ Ignore } T(1,-1) \}$$

$$T(2,2) = 3$$

Finding T(2,3)-

We have,

- $i = 2$
- $j = 3$
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

$$T(2,3) = \max \{ T(2-1, 3), 4 + T(2-1, 3-3) \}$$

$$T(2,3) = \max \{ T(1,3), 4 + T(1,0) \}$$

$$T(2,3) = \max \{ 3, 4+0 \}$$

$$T(2,3) = 4$$

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

	0	1	2	3	4	5
0	0	0	0	0	0	0
✓ 1	0	0	3	3	3	3
✓ 2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

Identifying Items To Be Put Into Knapsack

Following Step-04,

- We mark the rows labelled “1” and “2”.
- Thus, items that must be put into the knapsack to obtain the maximum value 7 are-

Item-1 and Item-2

Time Complexity-

- Each entry of the table requires constant time $\theta(1)$ for its computation.
- It takes $\theta(nw)$ time to fill $(n+1)(w+1)$ table entries.
- It takes $\theta(n)$ time for tracing the solution since tracing process traces the n rows.
- Thus, overall $\theta(nw)$ time is taken to solve 0/1 knapsack problem using dynamic programming

Conclusion-In this way we have explored Concept of 0/1 Knapsack using Dynamic approach

```
In [1]: nterms = int(input("How many terms? "))

n1, n2 = 0, 1
count = 0

if nterms <= 0:
    print("Please enter a positive integer")
elif nterms == 1:
    print("Fibonacci sequence upto",nterms,":")
    print(n1)
else:
    print("Fibonacci sequence:")
    while count < nterms:
        print(n1)
        nth = n1 + n2
        # update values
        n1 = n2
        n2 = nth
        count += 1
```

Fibonacci sequence:

0
1
1
2
3
5
8

In []:

In []:


```
In [3]: class Nodes:

    def __init__(self, probability, symbol, left = None, right = None):

        self.probability = probability

        self.symbol = symbol

        self.left = left

        self.right = right

        self.code = ''

def CalculateProbability(the_data):

    the_symbols = dict()

    for item in the_data:

        if the_symbols.get(item) == None:

            the_symbols[item] = 1

        else:

            the_symbols[item] += 1

    return the_symbols

the_codes = dict()

def CalculateCodes(node, value = ''):

    newValue = value + str(node.code)

    if(node.left):

        CalculateCodes(node.left, newValue)

    if(node.right):

        CalculateCodes(node.right, newValue)

    if(not node.left and not node.right):

        the_codes[node.symbol] = newValue

    return the_codes

def OutputEncoded(the_data, coding):

    encodingOutput = []

    for element in the_data:

        encodingOutput.append(coding[element])
```

```

the_string = ''.join([str(item) for item in encodingOutput])

return the_string

def TotalGain(the_data, coding):

    beforeCompression = len(the_data) * 8

    afterCompression = 0

    the_symbols = coding.keys()

    for symbol in the_symbols:

        the_count = the_data.count(symbol)

        afterCompression += the_count * len(coding[symbol])

    print("Space usage before compression (in bits):", beforeCompression)

    print("Space usage after compression (in bits):", afterCompression)

def HuffmanEncoding(the_data):

    symbolWithProbs = CalculateProbability(the_data)

    the_symbols = symbolWithProbs.keys()

    the_probabilities = symbolWithProbs.values()

    print("symbols: ", the_symbols)

    print("probabilities: ", the_probabilities)

    the_nodes = []

    for symbol in the_symbols:

        the_nodes.append(Nodes(symbolWithProbs.get(symbol), symbol))

    while len(the_nodes) > 1:

        the_nodes = sorted(the_nodes, key = lambda x: x.probability)

        right = the_nodes[0]

        left = the_nodes[1]

        left.code = 0

        right.code = 1

        newNode = Nodes(left.probability + right.probability, left.symbol + right.s

        the_nodes.remove(left)

```

```

        the_nodes.remove(right)

        the_nodes.append(newNode)

    huffmanEncoding = CalculateCodes(the_nodes[0])

    print("symbols with codes", huffmanEncoding)

    TotalGain(the_data, huffmanEncoding)

    encodedOutput = OutputEncoded(the_data, huffmanEncoding)

    return encodedOutput, the_nodes[0]

def HuffmanDecoding(encodedData, huffmanTree):

    treeHead = huffmanTree

    decodedOutput = []

    for x in encodedData:

        if x == '1':

            huffmanTree = huffmanTree.right

        elif x == '0':

            huffmanTree = huffmanTree.left

        try:

            if huffmanTree.left.symbol == None and huffmanTree.right.symbol == None:

                pass

            except AttributeError:

                decodedOutput.append(huffmanTree.symbol)

                huffmanTree = treeHead

    string = ''.join([str(item) for item in decodedOutput])

    return string

the_data = "AAAAAABBCCCCDDDEEEEEEE"

print(the_data)

encoding, the_tree = HuffmanEncoding(the_data)

print("Encoded output", encoding)

```

```
print("Decoded Output", HuffmanDecoding(encoding, the_tree))
```

```
AAAAAABBBCCCCCDDDEEEEEEEEE
```

```
symbols: dict_keys(['A', 'B', 'C', 'D', 'E'])
```

```
probabilities: dict_values([7, 2, 6, 3, 9])
```

```
symbols with codes {'E': '00', 'A': '01', 'C': '10', 'D': '110', 'B': '111'}
```

```
Space usage before compression (in bits): 216
```

```
Space usage after compression (in bits): 59
```

```
Encoded output 01010101010111111101010101011011100000000000000000
```

```
Decoded Output AAAAAABBBCCCCCDDDEEEEEEEEE
```

In []:

```
In [2]: class Item:
        def __init__(self, profit, weight):
            self.profit = profit
            self.weight = weight

        def fractionalKnapsack(W, arr):
            arr.sort(key=lambda x: (x.profit/x.weight), reverse=True)

            finalvalue = 0.0

            for item in arr:
                if item.weight <= W:
                    W -= item.weight
                    finalvalue += item.profit
                else:
                    finalvalue += item.profit * W / item.weight
                    break
            return finalvalue

if __name__ == "__main__":
    W = 50
    arr = [Item(60, 10), Item(100, 20), Item(120, 30)]
    max_val = fractionalKnapsack(W, arr)
    print(max_val)
```

240.0

In []:

```

In [5]: def knapsack(wt, val, W, n):
        if n == 0 or W == 0:
            return 0

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

        if wt[n - 1] <= W:
            t[n][W] = max(
                val[n - 1] + knapsack(wt, val, W - wt[n - 1], n - 1),
                knapsack(wt, val, W, n - 1)
            )
        else:
            t[n][W] = knapsack(wt, val, W, n - 1)

        return t[n][W]

if __name__ == '__main__':
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    W = 50
    n = len(profit)

    t = [[-1 for _ in range(W + 1)] for _ in range(n + 1)]

    print("Maximum Profit:", knapsack(weight, profit, W, n))

```

Maximum Profit: 220

In []: