Bharatiya Vidya Bhavan's
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department &
Information Technology Engineering Department

Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

Name	**Pratik Pujari**		
UID no.	**2020300054**	**Class:**	**Comps C Batch**
Experiment No.	4		

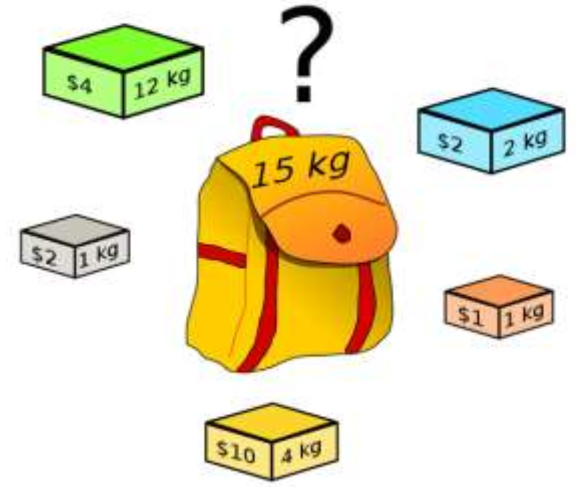
AIM:	To implement problems of Dynamic Programming
THEORY:	**What is Dynamic Programming?** Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems. **Characteristics of Dynamic Programming** Before moving on to understand different methods of solving a DP problem, let's first take a look at what are the characteristics of a problem that tells us that we can apply DP to solve it. **Top-down with Memoization** In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result. This technique of storing the results of already solved subproblems is called Memoization. **Bottom-up with Tabulation** Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem "bottom-up" (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional

table. Based on the results in the table, the solution to the top/original problem is then computed.
Tabulation is the opposite of Memoization, as in Memoization we solve the problem and maintain a map of already solved sub-problems. In other words, in memoization, we do it top-down in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

What is the Knapsack Problem?

The Knapsack Problem is a famous *Dynamic Programming* Problem that falls in the optimization category.
It derives its name from a scenario where, given a set of items with specific weights and assigned values, the goal is to maximize the value in a knapsack while remaining within the weight constraint.
Each item can only be selected once, as we don't have multiple quantities of any item.

Problem:
Given a Knapsack of a maximum capacity of W and N items each with its own value and weight, throw in items inside the Knapsack such that the final contents has the maximum value. Yikes !!



Here's the general way the problem is explained – Consider a thief gets into a home to rob and he carries a knapsack. There are fixed number of items in the home – each with its own weight and value – Jewellery, with less weight and highest value vs tables, with less value but a lot heavy. To add fuel to the fire, the thief has an old knapsack which has limited capacity. Obviously, he can't split the table into half or jewellery into 3/4ths. He either takes it or leaves it

Example:
Knapsack Max weight: W = 10 (units)

Total items: N = 4

Values of items: v[] = {10, 40, 30, 50}

Weight of items: w[] = {5, 4, 6, 3}
A cursory look at the example data tells us that the max value that we could accommodate with the limit of max weight of 10 is 50 + 40 = 90 with a weight of 7.
Approach:
The way this is optimally solved is using dynamic programming – solving for smaller sets of knapsack problems and then expanding them for the bigger problem.
Let's build an Item x Weight array called V (Value array):
V[N][W] = 4 rows * 10 columns
Each of the values in this matrix represent a smaller Knapsack problem.
Base case 1 : Let's take the case of 0th column. It just means that the knapsack has 0 capacity. What can you hold in them? Nothing. So, let's fill them up all with 0s.
Base case 2 : Let's take the case of 0 row. It just means that there are no items in the house. What do you do hold in your knapsack if there are no items. Nothing again !!! All zeroes.

Value Array - Item 0, Weight 0

	Weight 0	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6	Weight 7	Weight 8	Weight 9	Weight 10
Item 0	0	0	0	0	0	0	0	0	0	0	0
Item 1	0										
Item 2	0										
Item 3	0										
Item 4	0										

Solution:

1) Now, let's start filling in the array row-wise. What does row 1 and column 1 mean? That given the first item (row), can you accommodate it in the knapsack with capacity 1 (column). Nope. The weight of the first item is 5. So, let's fill in 0. In fact, we wouldn't be able to fill in anything until we reach the column 5 (weight 5).

2) Once we reach column 5 (which represents weight 5) on the first row, it means that we could accommodate item 1. Let's fill in 10 there (remember, this is a Value array):

Value Array - Item 1, Weight 5

	Weight 0	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6	Weight 7	Weight 8	Weight 9	Weight 10
Item 0	0	0	0	0	0	0	0	0	0	0	0
Item 1	0	0	0	0	0	10					
Item 2	0										
Item 3	0										
Item 4	0										

3) Moving on, for weight 6 (column 6), can we accommodate anything else with the remaining weight of 1 (weight – weight of this item => 6 – 5). Hey, remember, we are on the first item. So, it is kind of intuitive that the rest of the row will just be the same value too since we are unable to add in any other item for that extra weight that we have got.

Value Array - Item 1, Weight 6

	Weight 0	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6	Weight 7	Weight 8	Weight 9	Weight 10
Item 0	0	0	0	0	0	0	0	0	0	0	0
Item 1	0	0	0	0	0	10	10				
Item 2	0										
Item 3	0										
Item 4	0										

4) So, the next interesting thing happens when we reach the column 4 in third row. The current running weight is 4. We should check for the following cases.

1) Can we accommodate Item 2 – Yes, we can. Item 2's weight is 4.

2) Is the value for the current weight is higher without Item 2? – Check the previous row for the same weight. Nope. the previous row* has 0 in it, since we were not able able accommodate Item 1 in weight 4.

3) Can we accommodate two items in the same weight so that we could maximize the value? – Nope. The remaining weight after deducting the Item2's weight is 0.

	Value Array - Item 2, Weight 4

	Weight 0	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6	Weight 7	Weight 8	Weight 9	Weight 10
Item 0	0	0	0	0	0	0	0	0	0	0	0
Item 1	0	0	0	0	0	10	10	10	10	10	10
Item 2	0	0	0	0	40						
Item 3	0										
Item 4	0										

Why previous row?

Simply because the previous row at weight 4 itself is a smaller knapsack solution which gives the max value that could be accumulated for that weight until that point (traversing through the items).

Exemplifying,

1) The value of the current item = 40
2) The weight of the current item = 4
3) The weight that is left over = 4 – 4 = 0
4) Check the row above (the Item above in case of Item 1 or the cumulative Max value in case of the rest of the rows). For the remaining weight 0, are we able to accommodate Item 1? Simply put, is there any value at all in the row above for the given weight?

The calculation goes like so :

1) Take the max value for the same weight without this item:
previous row, same weight = 0
=> V[item-1][weight]
2) Take the value of the current item + value that we could accommodate with the remaining weight:
Value of current item
+ value in previous row with weight 4 (total weight until now (4) - weight of the current item (4))
=> val[item-1] + V[item-1][weight-wt[item-1]]
Max among the two is 40 (0 and 40).
3) The next and the most important event happens at column 9 and row 2. Meaning we have a weight of 9 and we have two items. Looking at the example data we could accommodate the first two items. Here, we consider few things:
    1. The value of the current item = 40
    2. The weight of the current item = 4
    3. The weight that is left over = 9 - 4 = 5

4. Check the row above. At the remaining weight 5, are we able to accommodate Item 1.

Value Array - Item 2, Weight 9 (accomodates 1 and 2)

	Weight 0	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6	Weight 7	Weight 8	Weight 9	Weight 10
Item 0	0	0	0	0	0	0	0	0	0	0	0
Item 1	0	0	0	0	0	10	10	10	10	10	10
Item 2	0	0	0	0	40	40	40	40	40	50	
Item 3	0										
Item 4	0										

So, the calculation is :
1) Take the max value for the same weight without this item:
previous row, same weight = 10
2) Take the value of the current item + value that we could accumulate with the remaining weight:
Value of current item (40)
+ value in previous row with weight 5 (total weight until now (9) - weight of the current item (4))= 10
10 vs 50 = 50.
At the end of solving all these smaller problems, we just need to return the value at V[N][W] – Item 4 at Weight 10:

Value Array - Final set

	Weight 0	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6	Weight 7	Weight 8	Weight 9	Weight 10
Item 0	0	0	0	0	0	0	0	0	0	0	0
Item 1	0	0	0	0	0	10	10	10	10	10	10
Item 2	0	0	0	0	40	40	40	40	40	50	50
Item 3	0	0	0	0	40	40	40	40	40	50	70
Item 4	0	0	0	50	50	50	50	90	90	90	90

Complexity
Analysing the complexity of the solution is pretty straight-forward. We just have a loop for W within a loop of N => O (N*W)

PSEUDOCODE:

```
array m[0..n, 0..W];
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    m[i, 0] := 0

for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

Bharatiya Vidya Bhavan's
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department &
Information Technology Engineering Department

Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

EXPERIMENT 1	
CODE:	(see code below)

```
import java.util.Scanner;
import java.io.BufferedReader;
import java.util.ArrayList;

public class KnapSack {
    public static void listtoArray(int array[], ArrayList<Integer> list) {
        for (int i = 0; i < list.size(); i++) {
            array[i] = list.get(i);
        }
    }

    public static void main(String[] args) throws Exception {

        Scanner input = new Scanner(System.in);

        // Arraylist for storing temp values
        ArrayList<Integer> _temp = new ArrayList<Integer>();

        String numbers[];
        int length = 0;
        int values[], weight[], W;

        System.out.print("\n\t\tKNAPSACK ALGORITHM");
        BufferedReader br = new BufferedReader(new
java.io.InputStreamReader(System.in));
        // Take values input
        System.out.print("\nEnter the Values of the array(with space)\n-> ");

        String inputstr = br.readLine();
        if (!inputstr.equals("")) {
            numbers = inputstr.split(" ");
            for (String number : numbers)
                _temp.add(Integer.parseInt(number));
            length = _temp.size();
        }
```

```
        values = new int[length];
        listtoArray(values, _temp);

        // weight input
        weight = new int[length];
        System.out.print("\n---- Weights of items ----\n ");
        for (int i = 0; i < length; i++) {
            System.out.print("\n-> Weight of item '" + _temp.get(i) + "' : ");
            weight[i] = input.nextInt();
        }

        // Maz weight
        System.out.print("\nEnter the Max Weight: ");
        W = input.nextInt();

        System.out.println("\nThe limit of max possible weight is " +
knapsack(values, weight, W));
        input.close();

    }

    public static int knapsack(int val[], int wt[], int W) {

        System.out.print("\nFormulating the problem \n");

        // Get the total number of items.
        // Could be wt.length or val.length. Doesn't matter

        int N = wt.length;

        // Create a matrix.
        // Items are in rows and weight at in columns +1 on each side

        int[][] values = new int[N + 1][W + 1];

        // What if the knapsack's capacity is 0 - Set
        // all columns at row 0 to be 0

        for (int col = 0; col <= W; col++) {
```

```
        values[0][col] = 0;

    }

    // What if there are no items at home.
    // Fill the first row with 0
    for (int row = 0; row <= N; row++) {
        values[row][0] = 0;
    }

    for (int item = 1; item <= N; item++) {

        // Let's fill the values row by row
        for (int weight = 1; weight <= W; weight++) {
            // Is the current items weight less
            // than or equal to running weight
            if (wt[item - 1] <= weight) {

                // Given a weight, check if the value of the current
                // item + value of the item that we could afford
                // with the remaining weight is greater than the value
                // without the current item itself

                values[item][weight] = Math.max(val[item - 1] + values[item -
1][weight - wt[item - 1]],
                        values[item - 1][weight]);
            }

            else {
                // If the current item's weight is more than the
                // running weight, just carry forward the value
                // without the current item

                values[item][weight] = values[item - 1][weight];
            }
        }

    }
```

```
        // Printing the matrix
        for (int[] rows : values) {
            for (int col : rows) {
                System.out.format("%5d", col);
            }
            System.out.println();
        }
        return values[N][W];
    }

}
```

OUTPUT:

```
                KNAPSACK ALGORITHM
Enter the Values of the array(with space)
-> 2 5 8 1

---- Weights of items ----

-> Weight of item '2' : 3

-> Weight of item '5' : 6

-> Weight of item '8' : 8

-> Weight of item '1' : 3

Enter the Max Weight: 18

Formulating the problem
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
    0    0    0    2    2    2    5    5    5    7    7    7    7    7    7    7    7    7    7
    0    0    0    2    2    2    5    5    8    8    8   10   10   10   13   13   13   15   15
    0    0    0    2    2    2    5    5    8    8    8   10   10   10   13   13   13   15   15

The limit of max possible weight is 15
```

Weights :- {2, 3, 6, 4}

Value :- {1, 5, 8, 3}

Max Weight = 5

0	0	0	0	0	0
0	2	2	2	2	2
0	2	2	2	2	2
0	2	2	2	2	2
0	2	2	2	2	3
0	2	2	4	6	6.

Max possible weight = 6

Input:
Weights: 0
Max Wgt: 1

```
                    KNAPSACK ALGORITHM
Enter the Values of the array(with space)
->

---- Weights of items ----

Enter the Max Weight: 1

Formulating the problem
    0    0

The limit of max possible weight is 0
```

No of weights: 1, Max Weight: 5
Weight: 10
Weight val1: 5

```
                    KNAPSACK ALGORITHM
Enter the Values of the array(with space)
-> 10

---- Weights of items ----

-> Weight of item '10' : 5

Enter the Max Weight: 5

Formulating the problem
    0    0    0    0    0    0
    0    0    0    0    0   10

The limit of max possible weight is 10
```

No of weights: 4 Max Weight: 10
Weights: 10 20 30 40
Values: 12 13 15 19

```
                KNAPSACK ALGORITHM
Enter the Values of the array(with space)
-> 10 20 30 40

---- Weights of items ----

-> Weight of item '10' : 12

-> Weight of item '20' : 13

-> Weight of item '30' : 15

-> Weight of item '40' : 19

Enter the Max Weight: 10

Formulating the problem
    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    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0

The limit of max possible weight is 0
```

No of weights: 4, Max Weight: 7
Weights: 80 60 40 20
Values: 2 4 6 8

```
                KNAPSACK ALGORITHM
Enter the Values of the array(with space)
-> 80 60 40 20

---- Weights of items ----

-> Weight of item '80' : 2

-> Weight of item '60' : 4

-> Weight of item '40' : 6

-> Weight of item '20' : 8

Enter the Max Weight: 7

Formulating the problem
    0    0    0    0    0    0    0    0
    0    0   80   80   80   80   80   80
    0    0   80   80   80   80  140  140
    0    0   80   80   80   80  140  140
    0    0   80   80   80   80  140  140

The limit of max possible weight is 140
```

No of weights: 5, max Weight: 15
Weights: 4 2 1 10 2
Values: 12 2 1 4 1

```
                KNAPSACK ALGORITHM
Enter the Values of the array(with space)
-> 4 2 1 10 2

---- Weights of items ----

-> Weight of item '4' : 12

-> Weight of item '2' : 2

-> Weight of item '1' : 1

-> Weight of item '10' : 4

-> Weight of item '2' : 1

Enter the Max Weight: 15

Formulating the problem
    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    4    4    4    4
    0    0    2    2    2    2    2    2    2    2    2    2    4    4    6    6
    0    1    2    3    3    3    3    3    3    3    3    3    4    5    6    7
    0    1    2    3   10   11   12   13   13   13   13   13   13   13   13   13
    0    2    3    4   10   12   13   14   15   15   15   15   15   15   15   15

The limit of max possible weight is 15
```

No of weights: 10, Max weight: 165
Weight: 92 57 49 68 60 43 67 84 87 72
Values: 23 31 29 44 53 38 63 85 89 82

```
                KNAPSACK ALGORITHM
Enter the Values of the array(with space)
-> 92 57 49 68 60 43 67 84 87 72

---- Weights of items ----

-> Weight of item '92' : 23

-> Weight of item '57' : 31

-> Weight of item '49' : 29

-> Weight of item '68' : 44

-> Weight of item '60' : 53

-> Weight of item '43' : 38

-> Weight of item '67' : 63

-> Weight of item '84' : 85

-> Weight of item '87' : 89

-> Weight of item '72' : 82

Enter the Max Weight: 165
```

Output:

```
The limit of max possible weight is 309
```

<table>
<tr><td></td><td>No of weight: 15, max Weight: 750
Weight:
135 139 149 150 156 163 173 184 192 201 210 214 221 229 240
Values: 70 73 77 80 82 87 90 94 98 106 110 113 115 118 120
The limit of max possible weight is 1458

No of weight: 24, Max Weight: 6404180
Weight: 825594 1677009 1676628 1523970 943972 97426 69666 1296457 1679693 1902996 1844992 1049289 1252836 1319836 953277 2067538 675367 853655 1826027 65731 901489 577243 466257 369261

Values: 382745 799601 909247 729069 467902 44328 34610 698150 823460 903959 853665 551830 610856 670702 488960 951111 323046 446298 931161 31385 496951 264724 224916 169684
Formulating the problem

The limit of max possible weight is 13549094</td></tr>
<tr><td>TIME COMPLEXITY:</td><td>Time complexity for knapsack problem:-
for i=0 to (n+1) - - - - n+1
for j=0 to w - - - - w+1
if w[i] > j - - - - - - w
m[i,j] = m[i-1, j] - - - w
else - - - - - w
m[i,j] = max[m[i-1,j], m[i-1, j-w[i]]+v[i]] - - - - w

$T(w) = (n+1)(w+1+w+w+w)$
$= (4w+1)(n+1)$
$T(n) = \theta(w \cdot n)$
Time complexity of Knapsack Algo $= \theta(w \cdot n)$.</td></tr>
</table>

	Time Complexity**: O (N*W).** where 'N' is the number of weight elements and 'W' is the capacity of the knapsack.

CONCLUSION: Learnt how to solve dynamic programming problems by dividing bigger problems into smaller problems.
Learnt about the time complexity of the Knapsack problem. Learnt how and why is the 2d array is used in the Knapack problem. Dynamic problems like Fibonacci series nth number uses array to store reccuring solution which is implemented here too.