CS263

ASSIGNMENT 6

NAME:

ARCHIT AGRAWAL

ROLL NO. :

202051213

SECTION:

A

1. Given weights and cost of n items, put these items in a knapsack of capacity W to get the maximum total cost of the knapsack.
   You cannot break an item, either pick the complete item or don't pick it.
   Case:1 (pick only one time)
   For example:- W = {10, 20, 30} and C = {60, 100, 120}.
   Capacity of Knapsack, W= 50; Total Cost= 220 (after picking W2 and W3 of Cost 100 and 120, respectively.)

   Case:-2 (pick unbounded time)

   For example:- W = {1, 50} and V = {1, 30}.

   Capacity of Knapsack, W= 100; Total Cost= 100 (pick W1 100 times)

   Write both the brute force and Dynamic programming algorithm with a complete analysis to solve this problem.

Algorithm (Case 1: Brute Force)

This recursive brute force solution makes an exhaustive search and finds the subset with the given or less than knapsack capacity and maximum price.

To consider all subsets of given weights, there can be two possibilities for each weight.

- The weight is included in the optimal subset
- The weight is not included in the optimal subset.

Therefore, the maximum price that can be obtained from the 'm' given weights and 'w' capacity is the maximum of the following two values.

- Maximum price obtained by (m – 1) weights and capacity 'w' i.e. excluding the m^{th} weight.
- Maximum price obtained by (m - 1) weights with a capacity of (w – m^{th} weight) plus the price of m^{th} weight i.e. including the m^{th} weight.

Suppose the m^{th} weight is greater than the capacity 'w', then the second case is not possible.

The code for the above algorithm is given below.

Code (Case 1: Brute Force)

```
public class Knapsack{

    public static int knapsackBoundedBrute(int[] weights, int[] prices,
int w, int index){
        if(index == -1 || w == 0) return 0;

        if(weights[index] > w){
            return knapsackBoundedBrute(weights, prices, w, index - 1);
        } else {
            return Math.max(knapsackBoundedBrute(weights, prices, w,
index - 1), prices[index] + knapsackBoundedBrute(weights, prices, w -
prices[index], index - 1));
        }
    }

    public static int knapsackBoundedBrute(int[] weights, int[] prices,
int w){
        return knapsackBoundedBrute(weights, prices, w, weights.length
- 1);
    }
}
```

Output (Case 1: Brute Force)

- **Output when an optimum solution exists**

```
public static void main(String[] args){

        int[] weights = {10, 20, 30};
        int[] prices = {60, 100, 120};
        int w = 50;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackBoundedBrute(weights, prices, w));
    }
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { javac Knapsack
The maximum price with a capacity of 50 is: 120
PS C:\Users\Archit\Desktop\cprog>
```

- **Output when an optimum solution does not exist**

```
public static void main(String[] args){

        int[] weights = {10, 20, 30};
        int[] prices = {60, 100, 120};
        int w = 5;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackBoundedBrute(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { javac
The maximum price with a capacity of 5 is: 0
PS C:\Users\Archit\Desktop\cprog>
```

Note-: Analysis of all the algorithms is done together at the end of the document.

Algorithm (Case 1: Dynamic Programming)

In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.

The state DP[i][j] will denote maximum value of 'j-weight' considering all values from '1 to i^{th}'. So, if we consider 'w_i' (weight in 'i^{th}' row) we can fill it in all columns which have 'weight values > w_i'. Now two possibilities can take place:

- Fill 'w_i' in the given column.
- Do not fill 'w_i' in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill 'i^{th}' weight in 'j^{th}' column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of 'w_i'+ value of the column weighing 'j-w_i' in the previous row. So, we take the maximum of these two possibilities to fill the current state.

The code for the above algorithm is given below.

Code (Case 1: Dynamic Programming)

```
public static int knapsackBoundedDP(int[] weights, int[] prices, int w){
        int[][] table = new int[weights.length + 1][w + 1];

        for(int i = 0; i <= weights.length; i++){
            for(int j = 0; j <= w; j++){
                if(i == 0 || j == 0) table[i][j] = 0;
                else if(weights[i - 1] <= j){
                    table[i][j] = Math.max(table[i - 1][j], prices[i - 1] +
table[i - 1][j - weights[i - 1]]);
                } else{
                    table[i][j] = table[i - 1][j];
                }
            }
        }

        return table[weights.length][w];
    }
```

Output (Case 1: Dynamic Programming)

- **Output when an optimum solution exists, but the full capacity is not utilised.**

```
public static void main(String[] args){

        int[] weights = {1, 3, 5};
        int[] prices = {60, 100, 120};
        int w = 5;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackBoundedDP(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { javac
The maximum price with a capacity of 5 is: 160
PS C:\Users\Archit\Desktop\cprog>
```

Since, capacity is 5, we can pick either weight 5 or (weight 3 and weight 1). Since the price is more in picking (weight 3 and weight 1), it is chosen to fill the sack.

- **Output when an optimum solution does not exist**

```
public static void main(String[] args){

        int[] weights = {2, 3, 5};
        int[] prices = {60, 100, 120};
        int w = 1;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackBoundedDP(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { javac Kna
The maximum price with a capacity of 1 is: 0
PS C:\Users\Archit\Desktop\cprog>
```

Algorithm (Case 2: Brute Force)

The maximum price can be computed using the below algorithm.

'w' is capacity, weights and prices are given arrays.

- If w == 0, return 0
- Declare an integer 'maxPrice' and initialise it with 0.
- Traverse through the weights array. In each iteration, do the following:
  - Declare currMax, and make a recursive call with w = w – weights[i], where 'i' is i^{th} iteration.
  - If currMax + prices[i] > maxPrice, update maxPrice with (currMax + prices[i]).
- Return maxPrice.

The code for above algorithm is given below.

Code (Case 2: Brute Force)

```
public static int knapsackUnboundedBrute(int[] weights, int[] prices, int w){

        if(w == 0) return 0;
        int maxPrice = 0;

        for(int i = 0; i < weights.length; i++){
            if(weights[i] <= w){
                int currMax = knapsackUnboundedBrute(weights, prices, w -
weights[i]);
                if(currMax != Integer.MIN_VALUE && currMax + prices[i] >
maxPrice){
                    maxPrice = currMax + prices[i];
                }
            }
        }

        return maxPrice;
    }
```

Output (Case 2: Brute Force)

- **When an optimum solution exists and utilises the full capacity.**

```
public static void main(String[] args){

        int[] weights = {1, 2, 5};
        int[] prices = {60, 100, 120};
        int w = 5;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackUnboundedBrute(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { java
The maximum price with a capacity of 5 is: 300
PS C:\Users\Archit\Desktop\cprog>
```

5 time the weight 1 is selected and hence, the price is 5 * 60 = 300

- **When an optimum solution exists but does not utilises the full capacity.**

```
public static void main(String[] args){

        int[] weights = {2, 5};
        int[] prices = {100, 120};
        int w = 3;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackUnboundedBrute(weights, prices, w));
 }
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { javac
The maximum price with a capacity of 3 is: 100
PS C:\Users\Archit\Desktop\cprog>
```

Weight 2 is selected only. Hence, price is 100, and capacity is not fully utilised.

- **When an optimum solution does not exist.**

```
public static void main(String[] args){

        int[] weights = {2, 5};
        int[] prices = {100, 120};
        int w = 1;
```

```
        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackUnboundedBrute(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { javac
The maximum price with a capacity of 1 is: 0
PS C:\Users\Archit\Desktop\cprog>
```

Algorithm (Case 2: Dynamic Programming)

Make a table in the form of 1-D array of size ‘w + 1’ where ‘w’ is the size of weights array.

Fill this table using the below formula:

- table[i] = 0
- table[i] = max(table[i], table[i-wt[j]] + price[j] , where j varies from 0 to m-1 such that weight[j] <= i; ‘m’ is size of weight array.

Return table[w], it is the required maxPrice.

Code (Case 2: Dynamic Programming)

```
public static int knapsackUnboundedDP(int[] weights, int[] prices, int w){
        //this table will hold the maximum prices for each weights in the
range 0 to W
        int[] table = new int[w + 1];

        for(int i = 1; i <= w; i++){
            for(int j = 0; j < weights.length; j++){
                if(weights[j] <= i){
                    table[i] = Math.max(table[i], table[i - weights[j]] +
prices[j]);
                }
            }
        }

        if(table[w] == Integer.MIN_VALUE) return -1;
        return table[w];
}
```

Output (Case 2: Dynamic Programming)

- **When an optimum solution exists and utilises the full capacity.**

```
public static void main(String[] args){

        int[] weights = {10, 20, 5};
        int[] prices = {200, 120, 50};
        int w = 25;
```

```
        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackUnboundedDP(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { javac
The maximum price with a capacity of 25 is: 450
PS C:\Users\Archit\Desktop\cprog>
```

Two weights of 10 and one weight of 5 is picked.

- **When an optimum solution exists but does not utilises the full capacity.**

```
public static void main(String[] args){

        int[] weights = {10, 20, 5};
        int[] prices = {200, 120, 50};
        int w = 22;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackUnboundedDP(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { jav
The maximum price with a capacity of 22 is: 400
PS C:\Users\Archit\Desktop\cprog>
```

Two weights of 10 are picked only, remaining capacity of 2 can't be filled.

- **When an optimum solution does not exist.**

```
public static void main(String[] args){

        int[] weights = {10, 20, 5};
        int[] prices = {200, 120, 50};
        int w = 2;

        System.out.println("The maximum price with a capacity of "+ w
+" is: " +knapsackUnboundedDP(weights, prices, w));
}
```

```
PS C:\Users\Archit\Desktop\cprog> cd "c:\Users\Archit\Desktop\cprog\" ; if ($?) { jav
The maximum price with a capacity of 2 is: 0
PS C:\Users\Archit\Desktop\cprog>
```