

### Brute force Algorithm: -

#### **Bounded Knapsack problem: -**

In the brute force approach, we first have to find out all the possible sub sets available for the given string. And then check if the sum of all the weights is less than the target weight and filter the sets so far remaining.

On the other hand, put all the weights and their corresponding costs in a map and iterate through the set to check which set have highest sum.

#### **Code: -**

```
import java.util.*;

class Main{
    public static void main(String[] args) {
        int[] weight = {10,20,30};
        int[] cost = {60,100,120};
        int target = 50;
        System.out.println(knapsack(weight,cost,target));
    }
    //Brute force
    public static int knapsack(int[] weight,int[] cost, int target){
        int maxVal = 0;
        Map<Integer,Integer> map = new HashMap<>();
        for(int i=0;i<cost.length;i++){
            map.put(weight[i], cost[i]);
        }
        List<List<Integer>> res = possibleSets(weight,target);

        for(int i=0;i<res.size();i++){
            int sum = 0;
            for(int e : res.get(i)){
                sum += map.get(e);
            }
            maxVal = Math.max(maxVal,sum);
        }
        return maxVal;
    }
    public static List<List<Integer>> possibleSets(int[] arr,int target){
        List<Integer> current = new ArrayList<>();
        List<List<Integer>> res = new ArrayList<>();
        recursiveSets(arr, 0, current,res);

        for(int i=0;i<res.size();i++){
            int sum =0;
            for(int e : res.get(i)){
                sum += e;
            }
            if(sum>target) res.remove(i);
        }
    }
}
```

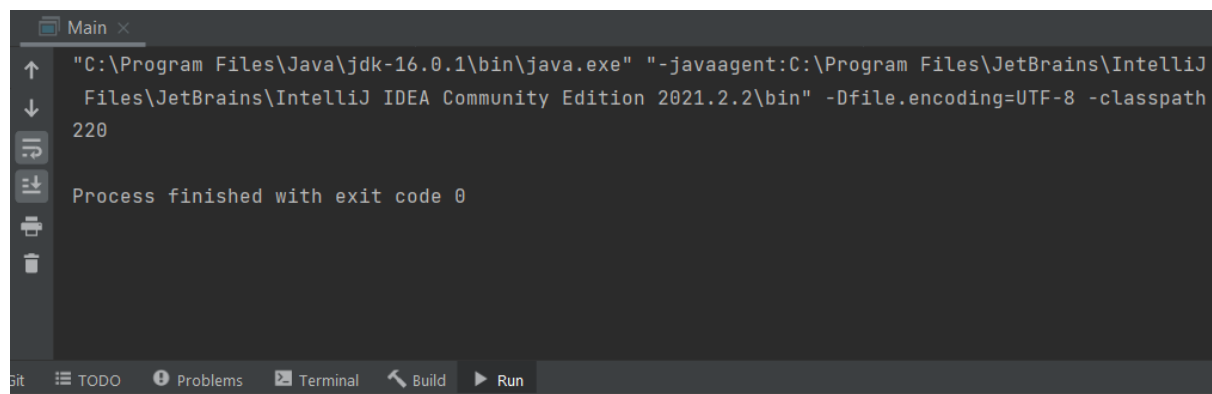
```

        return res;
    }

    private static void recursiveSets(int[] arr, int i, List<Integer>
current, List<List<Integer>> res) {
        if(i== arr.length){
            res.add(new ArrayList<>(current));
            return;
        }
        current.add(arr[i]);
        recursiveSets(arr,i+1,current,res);
        current.remove(current.size()-1);
        recursiveSets(arr,i+1,current,res);
    }
}

```

**Output: -**



```

Main x
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ
Files\JetBrains\IntelliJ IDEA Community Edition 2021.2.2\bin" -Dfile.encoding=UTF-8 -classpath
220
Process finished with exit code 0

```

**Complexity Analysis: -**

Time complexity to create all the possible sets =  $O(2^n)$

Time complexity to select all the suitable cases =  $O(2^{2n}) \Rightarrow O(2^n)$  where k = total number of sets

Time complexity to get maximum value =  $O(2^{2n}) \Rightarrow O(2^n)$

**Net Time complexity =  $O(2^n)$**

**Unbounded Knapsack: -**

In the unbounded case we just have to change the possible numbers of sets such that one number can be reaped also.

For this part we can re pick the picked number and change the target value to target value – picked number.

Rest of the operations are same.

**Code: -**

```
import java.util.*;
```

```

class Main{
    public static void main(String[] args) {
        int[] weight = {1,50};
        int[] cost = {1,30};
        int target = 100;
        System.out.println(unboundedKnapsack(weight,cost,target));

    }
    //Brute force
    public static int unboundedKnapsack(int[] weight,int[] cost, int
target){
        int maxVal = 0;
        Map<Integer,Integer> map = new HashMap<>();
        for(int i=0;i<cost.length;i++){
            map.put(weight[i], cost[i]);
        }
        List<List<Integer>> res = possibleSets(weight,target);

        for(int i=0;i<res.size();i++){
            int sum = 0;
            for(int e : res.get(i)){
                sum += map.get(e);
            }
            maxVal = Math.max(maxVal,sum);
        }
        return maxVal;
    }
    public static List<List<Integer>> possibleSets(int[] arr,int target){
        List<Integer> current = new ArrayList<>();
        List<List<Integer>> res = new ArrayList<>();
        recursiveSets(arr, 0,target, current,res);

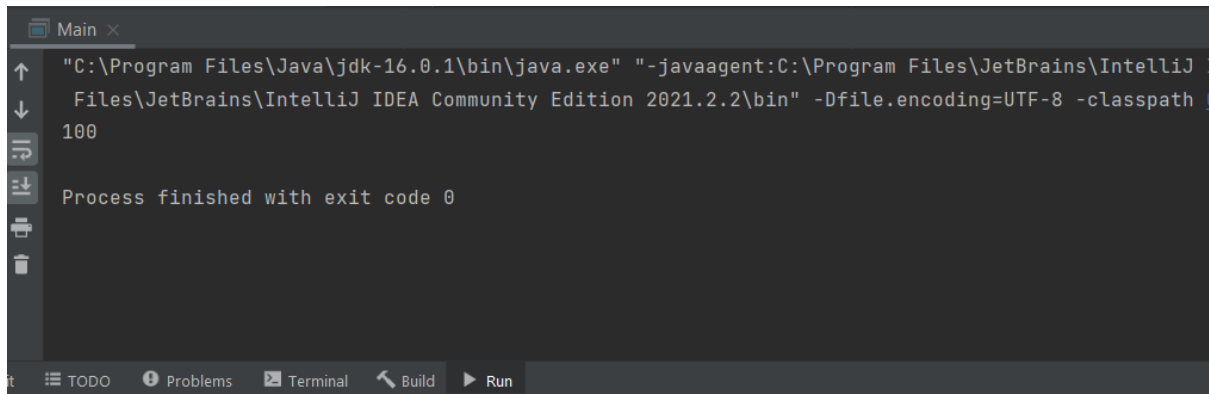
        for(int i=0;i<res.size();i++){
            int sum =0;
            for(int e : res.get(i)){
                sum += e;
            }
            if(sum>target) res.remove(i);
        }

        return res;
    }

    private static void recursiveSets(int[] arr, int i,int target,
List<Integer> current, List<List<Integer>> res) {
        if(i== arr.length){
            res.add(new ArrayList<>(current));
            return;
        }
        if(arr[i]<=target){
            current.add(arr[i]);
            recursiveSets(arr,i,target-arr[i],current,res);
            current.remove(current.size()-1);
        }
        recursiveSets(arr,i+1,target,current,res);
    }
}

```

**Output: -**



```
Main x
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021.2.2\bin" -Dfile.encoding=UTF-8 -classpath ...
100
Process finished with exit code 0
```

**Complexity Analysis: -**

Time complexity to create all the possible sets =  $O(2^{n+c})$  where  $c$  is the size of the weight array.

Time complexity to select all the suitable cases =  $O(2^{2(n+c)}) \Rightarrow O(2^{n+c})$  where  $k$  = total number of sets

Time complexity to get maximum value =  $O(2^{2(n+c)}) \Rightarrow O(2^{n+c})$

**Net Time complexity =  $O(2^{n+c})$**

**DP Algorithm: -**

**Bounded knapsack: -**

We can use a 2D array of size array size X target to store the maximum profit by storing the profit corresponding to a certain weight less than or equal to target weight and use the above row to fill the row below it.

The last element of the 2D array will be the answer.

**Code: -**

```
class Main{
    public static void main(String[] args) {
        int[] weight = {1,2,3};
        int[] cost = {6,10,12};
        int target = 5;
        System.out.println("The 2D Array formed : ");
        System.out.println("And the max value is "+
knapsack(weight,cost,target));

    }
    //Dynamic programming
    public static int knapsack(int[] weight,int[] cost, int target){
        int[][] dp = new int[weight.length+1][target+1];

        for(int i=0;i<= weight.length;i++){
            for(int w=0;w<= target;w++){
```

```

        if(i==0|| w==0) dp[i][w] = 0;
        else if (weight[i-1] <= w) dp[i][w] = Math.max( dp[i-1][w], dp[i-1][w-weight[i-1]] + cost[i-1]);
        else dp[i][w] = dp[i-1][w];
        System.out.print(dp[i][w] + "\t");
    }
    System.out.println();
}
System.out.println();
return dp[weight.length][target];
}
}

```

**Output: -**



```

Main x
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021.2.2\bin\idea-agent-2.jar" -Dfile.encoding=UTF-8 -classpath .
The 2D Array formed :
0  0  0  0  0  0
0  6  6  6  6  6
0  6  10 16 16 16
0  6  10 16 18 22

And the max value is 22

Process finished with exit code 0

```

**Complexity Analysis: -**

Time complexity = size of 2D array =  $O(n \times t)$

Space complexity = size of 2D array =  $O(n \times t)$

**Unbounded Knapsack: -**

In the case of unbounded knapsack we can re take the taken element therefore, we don't need to go to the above row to get the previous max value but we can check on the same row.

**Code: -**

```

class Main{
    public static void main(String[] args) {
        int[] weight = {1,5};
        int[] cost = {1,3};
        int target = 10;
        System.out.println("The 2D Array formed : ");
        System.out.println("And the max value is "+
unboundedKnapsack(weight,cost,target));

    }
    //Dynamic programming
    public static int unboundedKnapsack(int[] weight,int[] cost, int
target){

```

```

        int[][] dp = new int[weight.length+1][target+1];

        for(int i=0;i<= weight.length;i++){
            for(int w=0;w<= target;w++){
                if(i==0|| w==0) dp[i][w] = 0;
                else if (weight[i-1] <= w) dp[i][w] = Math.max( dp[i-1][w],dp[i][w-weight[i-1]] + cost[i-1]);
                else dp[i][w] = dp[i-1][w];
                System.out.print(dp[i][w] + "\t");
            }
            System.out.println();
        }
        System.out.println();
        return dp[weight.length][target];
    }
}

```

### Output: -

```

Main x
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ
The 2D Array formed :
0  0  0  0  0  0  0  0  0  0
0  1  2  3  4  5  6  7  8  9  10
0  1  2  3  4  5  6  7  8  9  10
And the max value is 10
Process finished with exit code 0

```

### Complexity Analysis: -

Time complexity = size of 2D array =  $O(n \times t)$

Space complexity = size of 2D array =  $O(n \times t)$

Its complexities are same as the previous case because we don't need to find any extra possibilities in the dynamic programming approach.