

CS263 : Design and Analysis of Algorithm Lab

Name: Hritik Kumar

Roll no.: 202051088

1 Problem 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 W_2 and W_3 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 W_1 100 times)

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

Brute force Algorithm: -

Bounded Knapsack problem: -

In the brute force approach, we first have to find out all the possible subsets 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 :

```
#include<bits/stdc++.h>
using namespace std;
//bounded knapsack
int knapsack(int W[],int C[],int n, int w){
    if(n==0 || w == 0){
        return 0;
    }
    if(W[n-1]>w){
        return knapsack(W,C,n-1,w);
    }
    else{
        return max(C[n-1] + knapsack(W,C,n-1,w-W[n-1]),knapsack(W,C,n-1,w));
    }
}
```

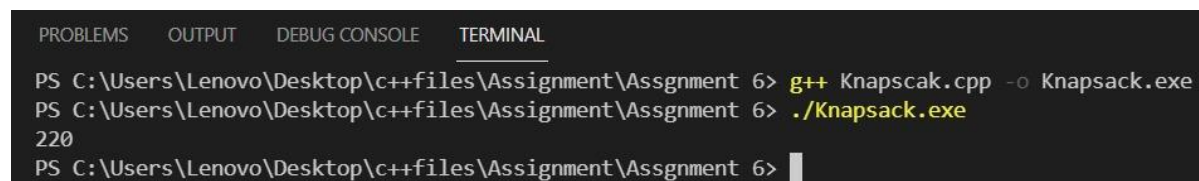
```

    }
}

int main()
{
    int n=3;
    int W[3] = {10,20,30}, C[3] = {60,100,120}, w = 50;
    cout<<knapsack(W,C,n,w);
}

```

Output :



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> g++ Knapsack.cpp -o Knapsack.exe
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> ./Knapsack.exe
220
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6>

```

Complexity Analysis: -

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

Time complexity to select all the suitable cases = $O(2^{2n}) \Rightarrow O(2n)$ 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 :

```
#include<bits/stdc++.h>
using namespace std;
int knapsack(int W[],int C[],int n,int w){
    if(n==0 || w==0){
        return 0;
    }

    if(W[n-1]>w){
        return knapsack(W,C,n-1,w);
    }
    else{
        return max(C[n-1] + knapsack(W,C,n,w-W[n-1]), knapsack(W,C,n-1,w));
    }
}

int main()
{
    int n = 2;
    int W[] = {1,50}, C[] = {1,30},w;
    w = 100;
    cout<<knapsack(W,C,n,w);
}
```

Output :

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell <https://aka.ms/pscore6>

PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> g++ Knapsack.cpp -o Knapsack.exe

PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> ./Knapsack.exe

100

PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> █

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:

```
#include<bits/stdc++.h>
using namespace std;

int dp[4][51];

int knapsack(int W[], int C[], int n, int w){
    if(dp[n][w]!=-1){
        return dp[n][w];
    }
    if(n==0 || w==0){
        return 0;
    }
    if(W[n-1]>w){
        return dp[n][w] = knapsack(W,C,n-1,w);
    }
    else{
        return dp[n][w] = max(C[n-1] + knapsack(W,C,n-1,w-W[n-1]),
knapsack(W,C,n-1,w));
    }
}
```

```

}
int main()
{
    int n=3;
    int W[3] = {10,20,30}, C[3] = {60,100,120},w = 50;
    memset(dp,-1,sizeof(dp));
    cout<<knapsack(W,C,n,w);
    return 0;
}

```

Output :

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> g++ Knapsack.cpp -o Knapsack.exe
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> ./Knapsack.exe
220
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assignment 6> █

```

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 :

```

#include<bits/stdc++.h>
using namespace std;

//dp algorithm
//unbounded knapsack

int dp[3][101];

```

```

int knapsack(int W[], int C[], int n, int w){
    if(dp[n][w]!=-1){
        return dp[n][w];
    }
    if(n==0 || w==0){
        return 0;
    }
    if(W[n-1]>w){
        return dp[n][w] = knapsack(W,C,n-1,w);
    }
    else{
        return dp[n][w] = max(C[n-1] + knapsack(W,C,n,w-W[n-1]),
knapsack(W,C,n-1,w));
    }
}

int main()
{
    int n = 2;
    int W[] = {1,50}, C[] = {1,30},w;
    w = 100;
    memset(dp,-1,sizeof(dp));
    cout<<knapsack(W,C,n,w);
}

```

Output :

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assgnment 6> g++ Knapsack.cpp -o Knapsack.exe
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assgnment 6> ./Knapsack.exe
100
PS C:\Users\Lenovo\Desktop\c++files\Assignment\Assgnment 6>

```

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.