Knapsack Problem

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Given a set of n items, each with its own value V_i and weight W_i for all 1<=i<=n and a maximum knapsack capacity C, compute the maximum value of the items that can be carried.

Two possibilities:

- Items are indivisible
  - We pick it or we leave it
  - 0/1 Knapsack (0 - Absent, 1 - Present)
  - Dynamic Programming
- Items are divisible
  - We can pick a fraction of an item
  - Fractional Knapsack
  - Greedy Approach

C = 15kg

Box 1: w_1 = 12kg, v_1 = 40K

Box 2: w_2 = 1kg, v_2 = 20K

Box 3: w_3 = 4kg, v_3 = 100K

Box 4: w_4 = 1kg, v_4 = 10K

Box 5: w_5 = 2kg, v_5 = 20K

Implement Greedy Approach !!!

Thief, C = 4kg
Item 1: w=1, v=35
Item 2: w=4, v=80
Item 3: w=3, v=50

Item	Mobile	Laptop	Tablet
Value	35	80	50
Weight	1	4	3

Weight (j) / Item (i)	1	2	3	4
1. Mobile Ph.	35	35	35	35
2. Laptop	35	35	35	80
3. Tablet	35	35	50	Option: 80 Option: 35+50=85 Selection should be 85

I = 3, J = 4, T[i-1][j] = T[2][4] = 80
V_i + T[i-1][j-w_i] = 50 + T[3-1][4-3] = 50 + T[2][1] = 50 + 35 = 85

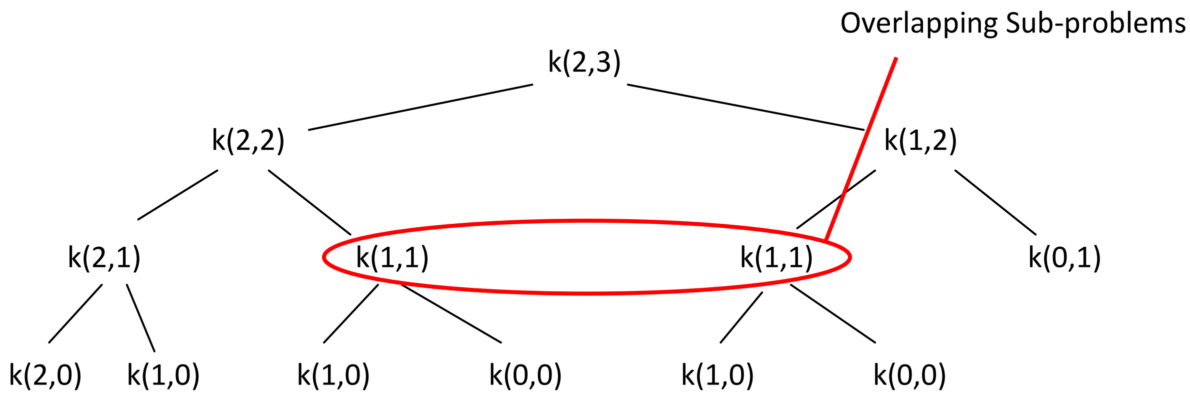
C = 20
Ball: w=5 v=10
Vase: w=10 v=40
Watch: w=3 v=50
Monitor: w=12 v=75
w[] = {5,10,3,12}
v[] = {10,40,50,75}

knapsack(C, n, w[], v[]) Capacity, No of Items, Weight, Value

knapsack(20, 4, w, v)
If monitor is selected: knapsack(8, 3, w, v)
If monitor is NOT selected: knapsack(20, 3, w, v)

k(C, n) where C is the capacity of the sack and n is the number of items available to be picked. Example: C=2, n=3.

If an item is selected, C is reduced and n is reduced.
If an item is rejected, C remains the same, n is reduced.



$$\mathrm{maxV}(i,C)=\begin{cases} 0 & \text{if } i==0 \\ 0 & \text{if } C<=0 \\ \mathrm{maxV}(i-1, C) & \text{if } W_i > C \\ \mathit{max}\left\{\begin{array}{l} \mathrm{maxV}(i-1, C) \\ Vi + \mathrm{maxV}(i-1, C-W_i) \end{array}\right\} & \text{If } W_i<=C \end{cases}$$

Complexity: Exponential

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include <bits/stdc++.h>
using namespace std;

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{

// Base Case
if (n == 0 || W == 0)
    return 0;

// If weight of the nth item is more
// than Knapsack capacity W, then
// this item cannot be included
// in the optimal solution
if (wt[n-1] > W)
    return knapSack(W, wt, val, n-1);

// Return the maximum of two cases:
// (1) nth item included
// (2) not included
else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1) );
}
```

```
// Driver code
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    cout<<knapSack(W, wt, val, n);
    return 0;
}

int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];

   // Build table K[][] in bottom up manner
   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }

   return K[n][W];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Complexity:

- Time: O(nC)
- Space: O(nC)