
We have Items, every Item has weight and profit.
Knapsack (Bag) Problems
Fixed Capacity
Problem Statement : Every Knapsack has the capacity, we should choose the items , which can give higher profit, by considering capacity in mind.
example : itemWeight = {1,2,3} profitProfit = {5,2,4} capacity : 5
Dynamic Programming Solutions
Allow only unique items.
But we can split, the item.
Greedy Algorithm
Allow only unique items.
we either take the whole item or don’t take it.
Allow duplicate items.
0/1 Knapsack
Unbound Knapsack
Fractional Knapsack
Item weight {1,2,3}
Item Profits {5,2,4}
Capacity : 5
Max Profit we can gain by choosing items {1,3} & respective profits {5,4} = 9
Capacity left in bag is 1. As we chosen
Item Weights {1,3} = 4
Item weight {1,2,3}
Item Profits {5,2,4}
Capacity : 5
Max Profit we can gain by choosing item {1} 5 times & profit is 5 * 5 = 25.
Capacity left in bag is 0. As we chosen item 1 , five times.
Item Weighs {1,2,3}
Item Profits {5,2,4}
Capacity : 5
item1 + 50% of weight of item2 + item3 = then profit is = 5 + 1 + 4 = 10
Capacity left in bag is 0.

We have Items, every Item has weight and profit.

0/1 Knapsack Problems

Fixed Capacity

Problem Statement : Given item weights and their respective profits, itemWeight = {1,2,3} ItemsProfit = {5,2,4} , Get the max profit.

Constraints :

The knapsack cacapacity : 5

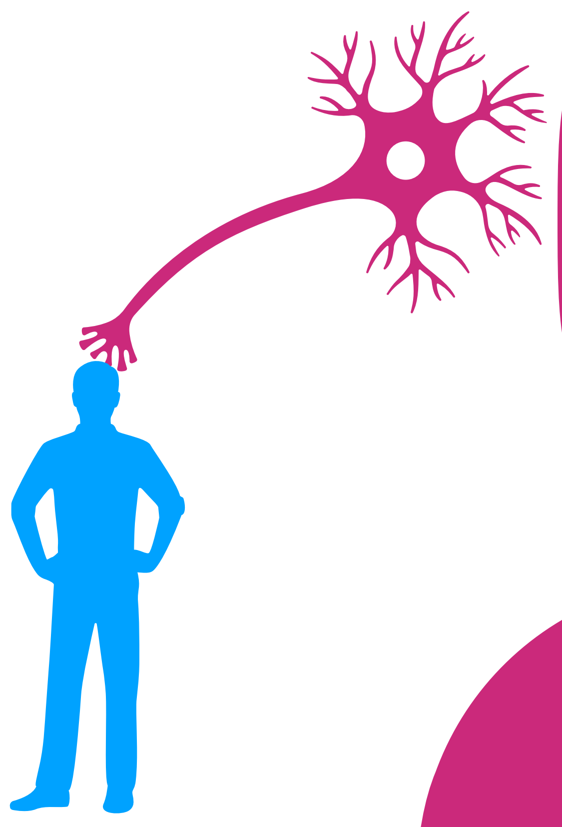
We allowed to choose the item only once.

Expected Output :

Max Profit we can gain by choosing itemWeights {1,3} & respective profits {5,4} = 9

Max Profit = 9

Selected items Weight {1,3}



Let’s figure out all possible permutations.
itemsWeight = {1,2,3}
Capacity = 5
ItemsProfit = {5,2,4}

1) Don't take any item => profit = 0, weight=0, capacityLeft = 5

2) i1 w{1} , profit {5} ,capacityLeft = 5-1 = 4

3) i2 w{2}, profit {2}, capacityLeft = 5 - 2 = 3

4) i3 w{3}, profit {4}, capacityLeft = 5 - 3 = 2

5) i1, i2 w{1+2 = 3}, profit{5+2 = 7} ,capacityLeft = 5 - 3 = 2

6) i1, i3 w{1+3 = 4}, profit{5+4 = 9} ,capacityLeft = 5 - 4 = 1

7) i2, i3 w{2+3 = 5}, profit{2+4 = 6}, capacityLeft = 5 - 5 = 0

8) i1, i2, i3 w{1+2+3 = 6} w > capacity ; 6 > 5 don't derive logic

Time Complexity : O(2^n)
Space Complexity : O(n)
weight: 0 profit: 0 capacityLeft : 5 selectedItems:[]
weights : {1,2,3} => i1, i2, i3
profits : {5,2,4}
i 1
Step1
Output : weight: 1 capacityLeft : 4 profit: 5 selectedItems:[i1]
Step2
i2
input : {2,3}
Step9
weight: 0 profit: 0 capacityLeft : 5 selectedItems:[]
i2
input : {2,3}
Output : weight: 3 capacityLeft : 2 profit: 7 selectedItems: [i1,i2]
Step3
i3
input : {3}
Output : weight: 1 capacityLeft : 4 profit: 5 selectedItems:[i1]
i3
input : {3}
Step6
weight: 2 profit: 2 capacityLeft : 3 selectedItems:[i2]
input : {3}
i3
Step10
weight: 0 profit: 0 capacityLeft : 5 selectedItems:[]
i3
Step13
input : {3}
w[i3] > capacityLeft return 0;
Step4
Step5
Step7
Step8
Step11
Step12
Step14
Step15
Output : weight: 3 capacityLeft : 2 profit: 7 selectedItems:[i1,i2]
Output : weight: 4 capacityLeft : 1 profit: 9 selectedItems:[i1,i3]
Output : weight: 1 capacityLeft : 4 profit: 5 selectedItems:[i1]
weight: 5 profit: 6 capacityLeft : 0 selectedItems:[i2,i3]
weight: 2 profit: 2 capacityLeft : 3 selectedItems:[i2]
weight: 3 profit: 4 capacityLeft : 2 selectedItems:[i3]
weight: 0 profit: 0 capacityLeft : 5 selectedItems:[]

Algorithm For Recursion

Base Condition in Recursion :
1. As we are moving from index 0 to n, the max possible valid value for currentIndex = n-1
2.As and when we add item to the knapsack, we reduce the capacity so the smallest possible value for capacity is ‘0’.