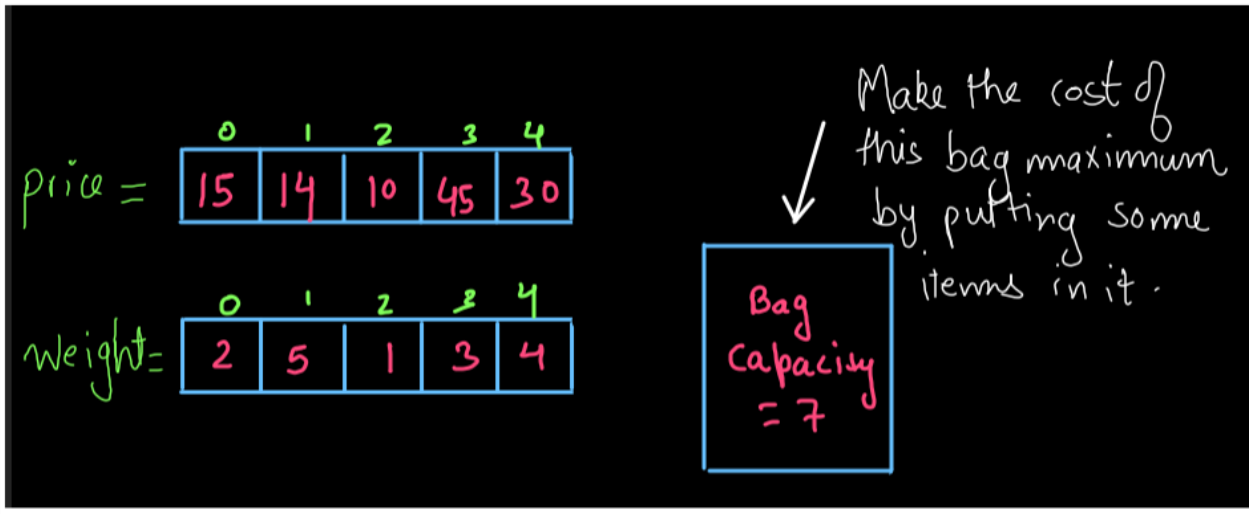
Dynamic Programming

Content

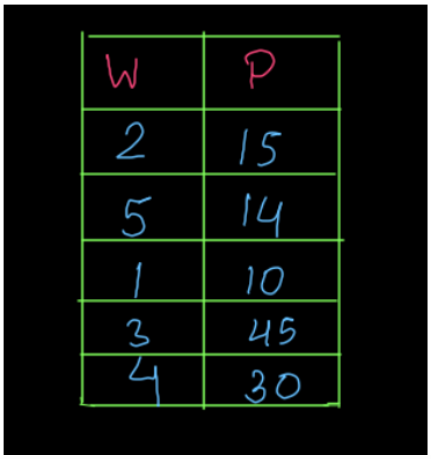
1. Knapsack Problem
    a. Problem
    b. Recursion and Backtracking
    c. Implementation
2. Largest Sum Contiguous Subarray
    a. Kadane's Algorithm
    b. Explanation
    c. Example
    d. Implementation



0-1 Knapsack Problem

Problem

We are given n items. We are given an array that tells us about the price of those items and we are also given an array that tells us about the weight of those items. This can be visualized as follows:



We can visualize this as follows. 2 units weight of the 0th item costs 15 units of money. Similarly, 5 units weight of 1st costs 14 units money and so on. We have a bag and we know the capacity of the bag (we will be given the capacity of the bag). We have to make the price of this bag maximum. For instance, in the above example, we can have only 7

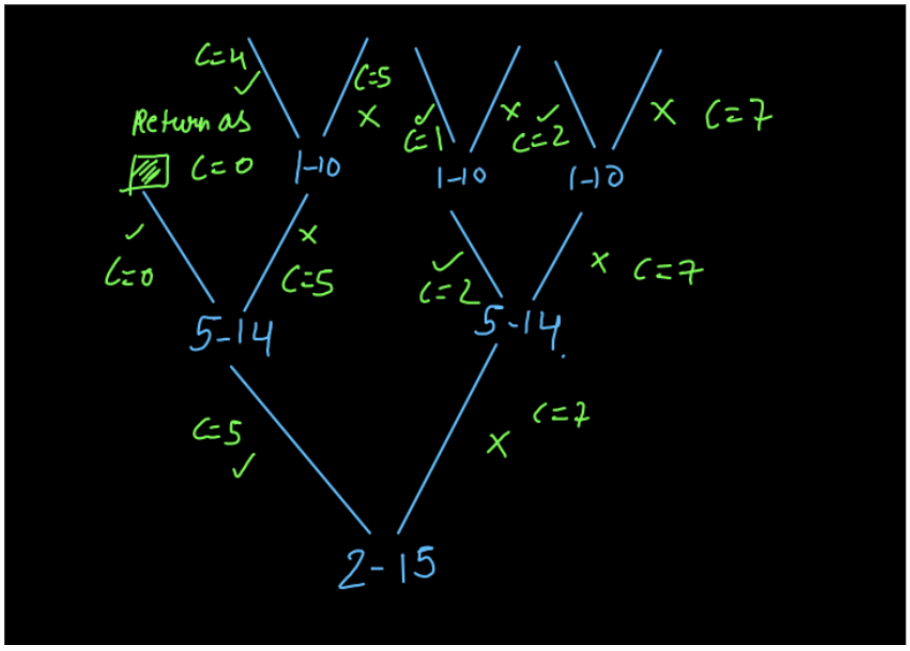
units weight of the items as the capacity of the bag is 7. The maximum price of the bag will be 75. How? Think about this!!!!!!

Note that it is the 0-1 knapsack problem. Every item can either be selected completely or can be dropped. But we cannot break the item and select it. What do we mean by this? We mean that if we have an item whose price for 2 units weight is 20, then either we can include the complete 2 units weight of the object or we can drop the object (i.e. not select it). But we cannot select 1 or 1.5 units weight of that object. Also, we cannot select a single item multiple times. We can select one item completely and only once.

Method-1: Using recursion and Backtracking

Approach :

A diagram is also shown below to give you a kickstart and help you a little bit in thinking about the recursive approach.



This is not the complete Euler tree. It is just a part of it to give you an idea of what we are doing. We either select or reject each element at every level to keep in the bag. For instance, we start with 2-15 means element 0 which has weight 2 and price is 15. If we take it in the bag the capacity of the bag now remains 5 else it remains the same i.e. 7. At

the next level if we already took element 0 and we take the next element too, then the capacity becomes 0 and we have to return. So, this is one base case that we can think of. There is another base case. Think of it and write the code yourself!!!

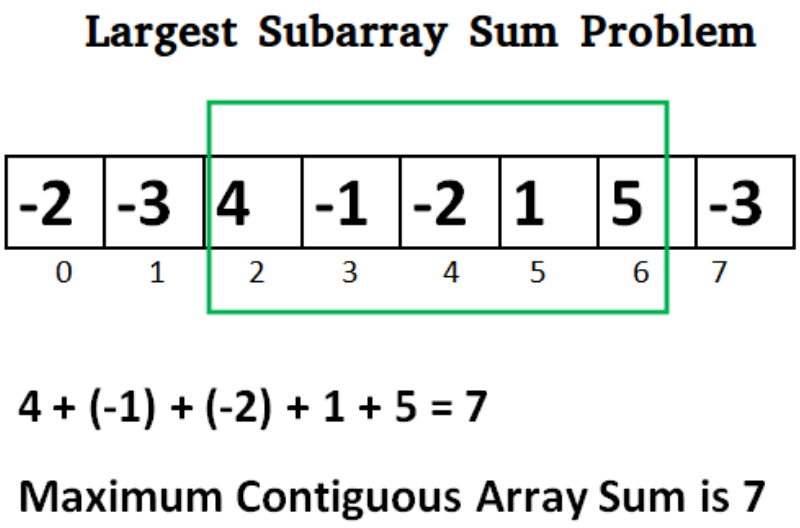
Pseudo Code for Backtracking method

Have you thought about the solution to this problem yet? We recommend you to think about this problem once. So, do you know any similar question or similar method by which we can solve this problem?

To give you an idea about that method, what we will do is that we will make a choice at each level of the recursive tree whether an item can be included or not be included taking in mind the capacity of the bag. We will have a lot of base cases reached in our tree. Then we will compare all the base cases and find out the max cost from it. That will be our answer.

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/zero_one_knapsack.cpp

Largest Sum Contiguous Subarray



Kadane's Algorithm:

Initialize:

max_so_far = INT_MIN

max_ending_here = 0

Loop for each element of the array

(a) max_ending_here = max_ending_here + a[i]

(b) if(max_so_far < max_ending_here)

max_so_far = max_ending_here

(c) if(max_ending_here < 0)

max_ending_here = 0

return max_so_far

Explanation:

The simple idea of Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum

contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

Example:

Lets take the example:

{-2, -3, 4, -1, -2, 1, 5, -3}

max_so_far = max_ending_here = 0

for i=0, a[0] = -2

max_ending_here = max_ending_here + (-2)

Set max_ending_here = 0 because max_ending_here < 0

for i=1, a[1] = -3

max_ending_here = max_ending_here + (-3)

Set max_ending_here = 0 because max_ending_here < 0

for i=2, a[2] = 4

max_ending_here = max_ending_here + (4)

max_ending_here = 4

max_so_far is updated to 4 because max_ending_here greater

than max_so_far which was 0 till now

for i=3, a[3] = -1

max_ending_here = max_ending_here + (-1)

max_ending_here = 3

for i=4, a[4] = -2

max_ending_here = max_ending_here + (-2)

max_ending_here = 1

for i=5, a[5] = 1

max_ending_here = max_ending_here + (1)

max_ending_here = 2

for i=6, a[6] = 5

max_ending_here = max_ending_here + (5)

max_ending_here = 7

max_so_far is updated to 7 because max_ending_here is

greater than max_so_far

for i=7, a[7] = -3

max_ending_here = max_ending_here + (-3)

max_ending_here = 4

Implementation:
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/kadanes_algorithm.cpp