Solution Review: The Knapsack Problem

In this lesson, we will review the solution to the knapsack problem.

We'll cover the following

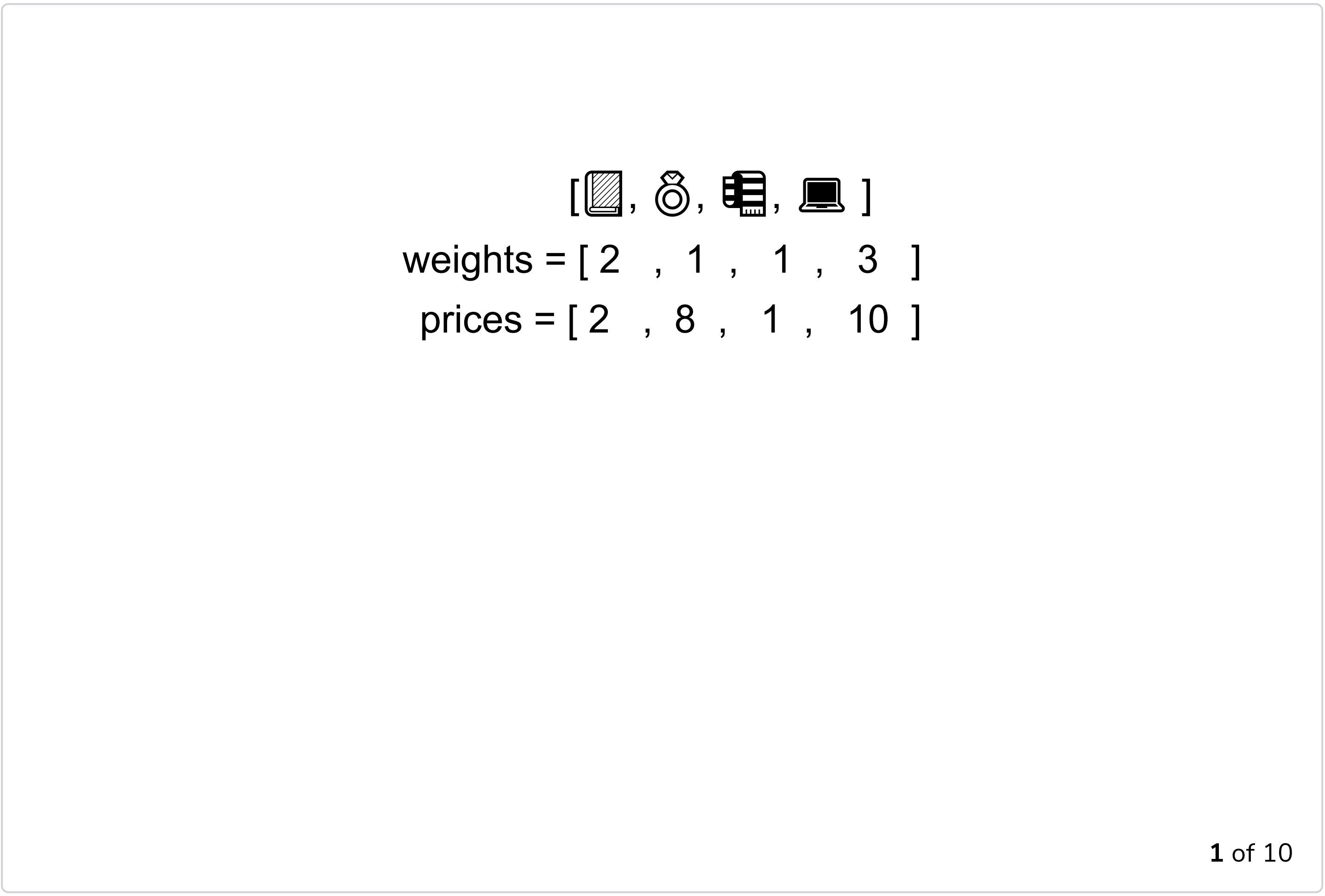
Solution 1: Simple recursion

```
def solveKnapsack(weights, prices, capacity, index):
  # base case of when we have run out of capacity or objects
  if capacity <= 0 or index >= len(weights):
    return 0
  # if weight at index-th position is greater than capacity, skip this object
  if weights[index] > capacity:
    return solveKnapsack(weights, prices, capacity, index + 1)
  # recursive call, either we can include the index-th object or we cannot, we check both possibil
  return max(prices[index]+solveKnapsack(weights, prices, capacity - weights[index], index+1),
        solveKnapsack(weights, prices, capacity, index + 1))

def knapsack(weights, prices, capacity):
  return solveKnapsack(weights, prices, capacity, 0)

print(knapsack([2,1,1,3], [2,8,1,10], 4))
```

Explanation

Let’s go over the intuition of the problem before moving on to the explanation of the code. Remember your goal was to select a subset of objects which return the maximum total profit while obeying the constraint that their weight is less than or maximum total profit while obeying the constraint that their weight is less than or equal to the total capacity of the knapsack. Now any object, let’s say a book, either

can or cannot be a part of the optimal subset. There cannot be any other possibility for this book, right? This is what we are doing in this solution. We make two recursive calls: one with an object included (*line 9*), and another without (*line 10*). Then we take the maximum of these two profit values. We continue doing this until we have either run out of capacity or objects (*line 3*). Along the way, if we reach a point where an object’s weight is greater than the `capacity`, we can skip over this object. Let’s see the following visualization for more clarification.



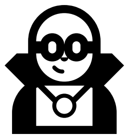
```
weights = [ 2  , 1 ,  1 ,  3  ]
 prices = [ 2  , 8 ,  1 ,  10 ]
```

1 of 10

[📔, 💍, 📒, 💻]

weights = [2 , 1 , 1 , 3]

prices = [2 , 8 , 1 , 10]



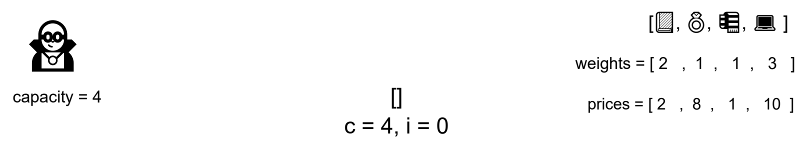
Thief has a knapsack of capacity = 4 kg

capacity = 4

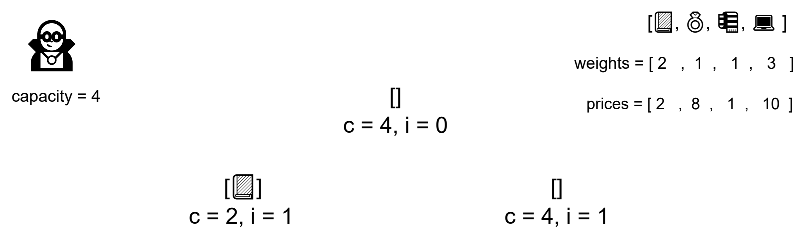
[📔, 💍, 📒, 💻]

weights = [2 , 1 , 1 , 3]

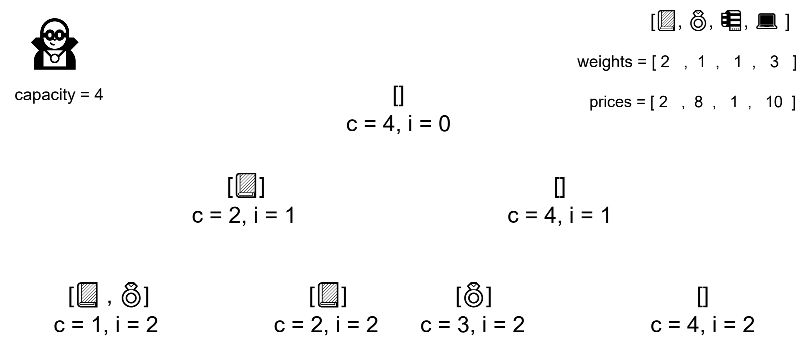
prices = [2 , 8 , 1 , 10]



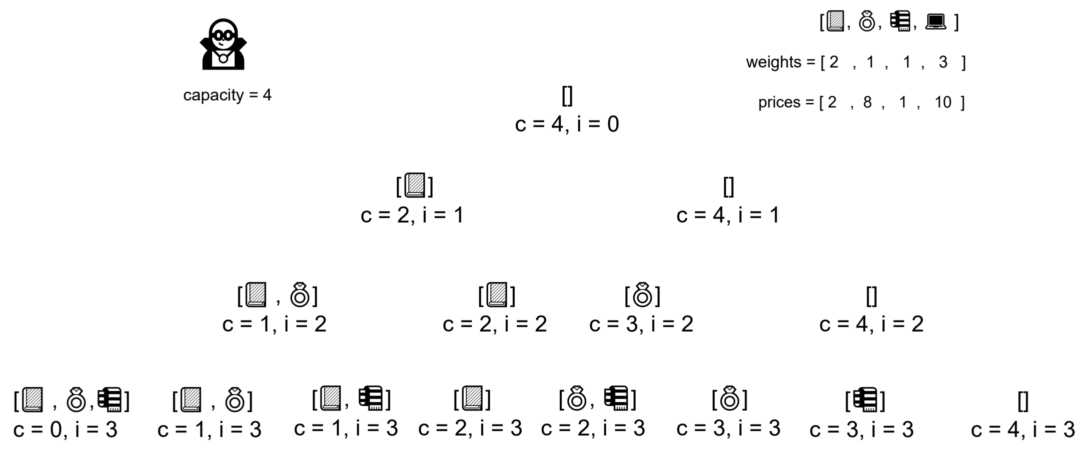
Let's run knapsack algorithm, here c denotes remaining capacity and I denotes index of the item we are on currently, e.g. 0 means book



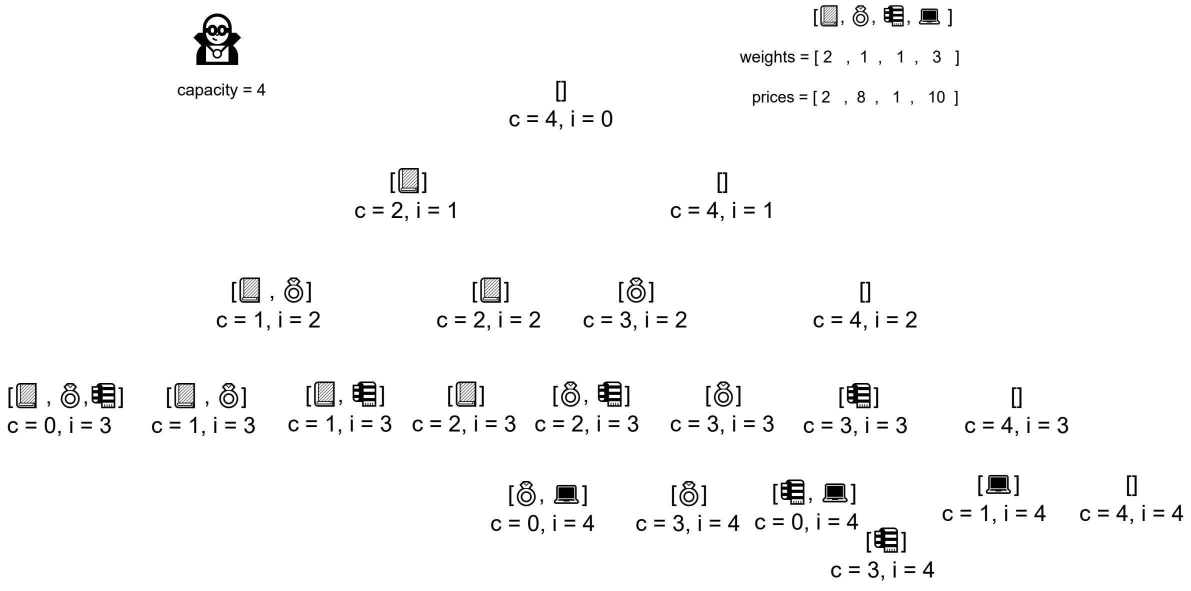
Book can either be in the sack or not



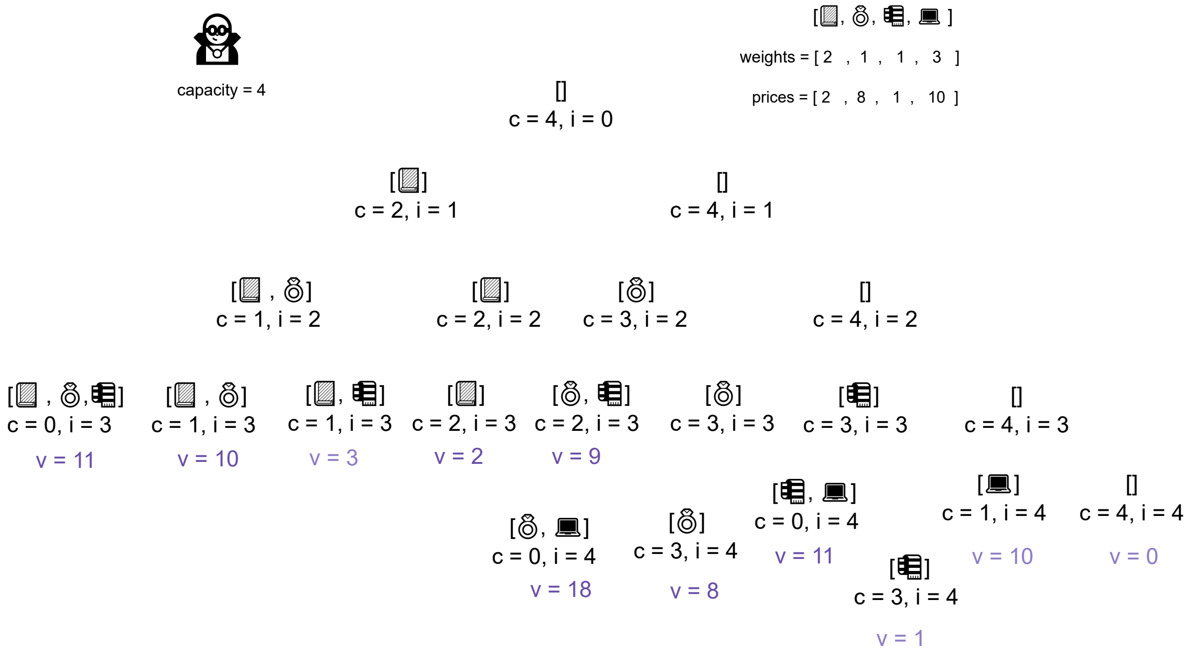
Ring can either be in the sack or not



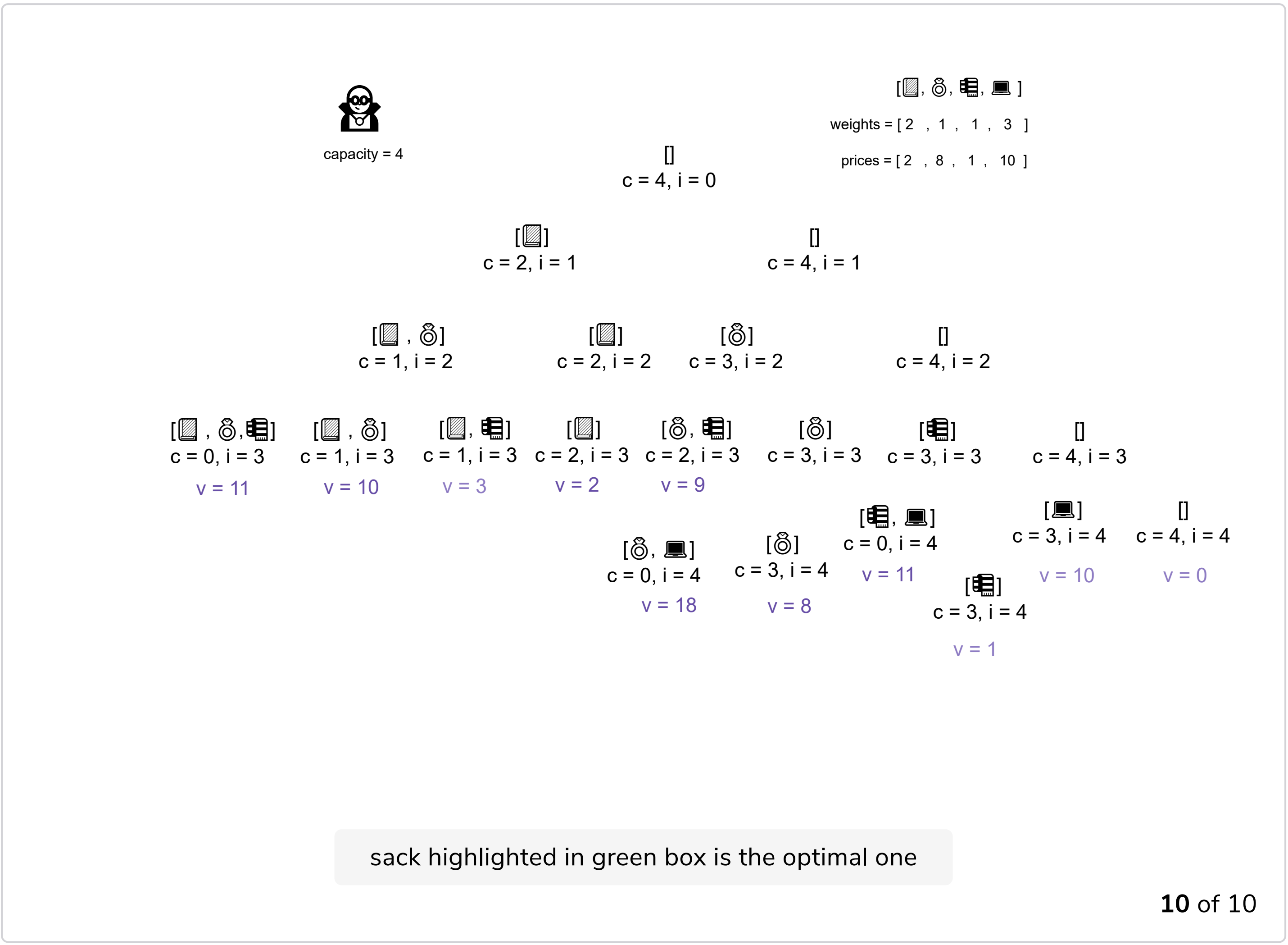
Scarf can either be in the sack or not



Laptop can either be in the sack or not; also notice in a lot of cases the weight of the laptop exceeds the remaining capacity so we don't make the recursive calls of including it in the sack



Let's see value of each sack denoted by v



sack highlighted in green box is the optimal one

Time complexity

If we have `n` objects, we find every set where any object is either in the set or it is not. This means we are looking for all the possible subsets of a set of `n` objects. How many subsets exist for a set of `n` objects? 2^n thus time complexity is $\mathbf{O(2^n)}$.

Solution 2: With memoization

Did you notice any repeating subproblems in the previous visualization? You can take a look at the last slides again before proceeding with the lesson.

Let's look at whether this problem satisfies both conditions of dynamic programming.

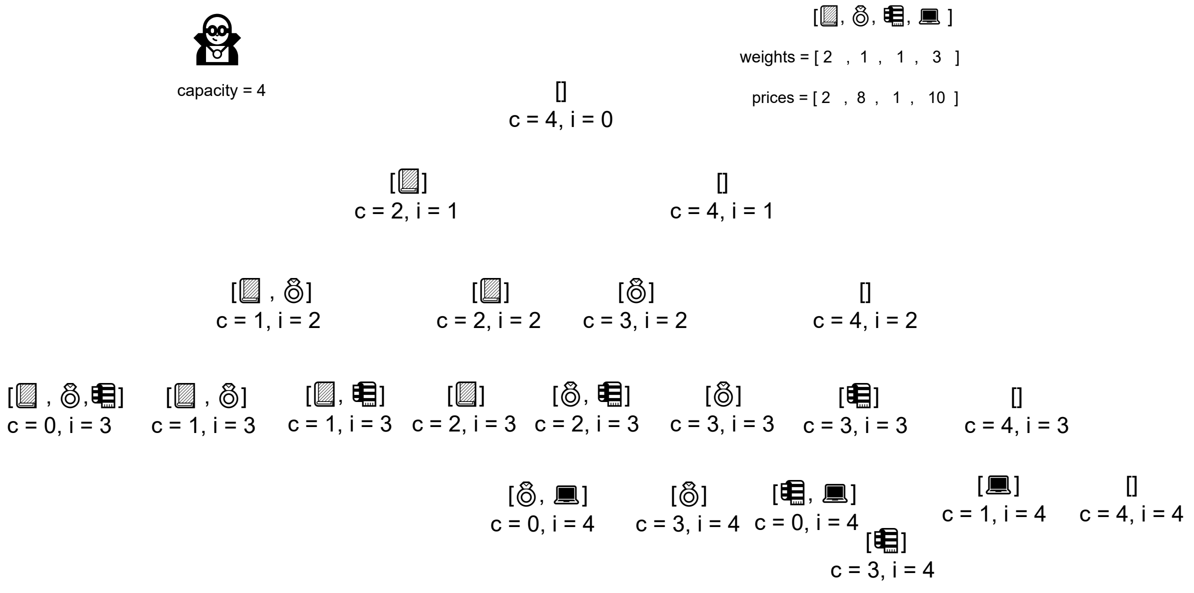
Optimal substructure

Let's say we have `n` objects and a sack with capacity `c`. The first object `o` has a weight, `w`. If I knew the optimal answer for n-1 objects excluding `o` with capacity `c` and the answer for `n-1` objects with capacity `c-w`, I could simply form the

answer by comparing these two values. Thus, this problem obeys the optimal substructure property.

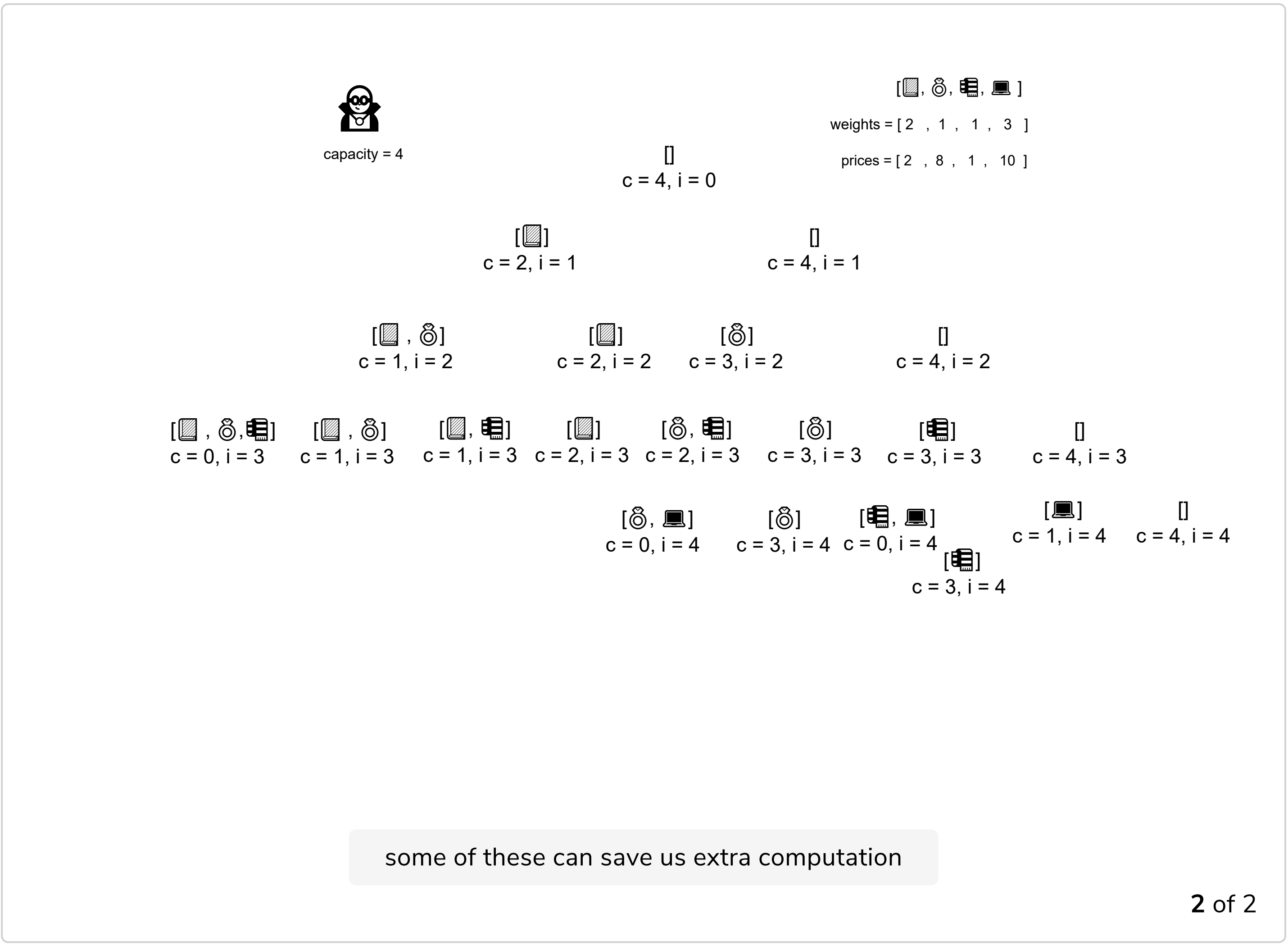
Overlapping subproblems

In this problem, it is a little hard to see overlapping problems, since they do not follow a specific pattern. But if you look closely, you will notice capacity and index tuple are oftentimes repeating. All of these subproblems have the same answers. Look at the following visualization.



look at the repeating c,i tuples

1 of 2



This essentially means that if we know the remaining capacity is `3` and index is `3` (laptop is the remaining object), we could already tell what the optimal answer for this configuration would be irrespective of what is already in the sack.

In other words, if the thief already had a sack of 3 kg filled with an optimal subset of things, he could fit this 3 kg sack in his original 4 kg sack, if he had space of 3 kg in it.

Below is the code of the knapsack problem with memoization.

```
def solveKnapsack(weights, prices, capacity, index, memo):
  # base case of when we have run out of capacity or objects
  if capacity <= 0 or index >= len(weights):
    return 0
  # check for solution in memo table
  if (capacity, index) in memo:
    return memo[(capacity, index)]
  # if weight at index-th position is greater than capacity, skip this object
  if weights[index] > capacity:
    # store result in memo table
    memo[(capacity, index)] = solveKnapsack(weights, prices, capacity, index + 1, memo)
    return memo[(capacity, index)]
  # recursive call, either we can include the index-th object or we cannot, we check both possibil

  memo[(capacity, index)] = max(prices[index]+solveKnapsack(weights, prices, capacity - weights[in
        solveKnapsack(weights, prices, capacity, index + 1, memo))
  return memo[(capacity, index)]

def knapsack(weights, prices, capacity):
  # create a memo dictionary
  memo = {}
  return solveKnapsack(weights, prices, capacity, 0, memo)

print(knapsack([2,1,1,3], [2,8,1,10], 4))
```

Explanation

As we can see from the visualization, we can memoize based on a tuple of `capacity` and `index`. That is the only add-on in this code over the previous one with simple recursion. Before the recursive step, we check if the result is already available to us in the `memo` table. Similarly, after the evaluation of the result, we store it in the `memo` table again.

Time complexity

Just think how big the size of the `memo` table will be. Since we will only be evaluating and storing as many results as we need in the `memo` table, the rest of the results will be *O(1)* lookup from it. We index our `memo` table with a tuple of `capacity` and `index`. The index can go from `1` to `n` if we have n objects, while `capacity` cannot be greater than that specified in the main call and cannot be less than zero. Moreover, `capacity` only varies discretely. Thus, it will also go from `1` to some upper bound, `C`, or the capacity of the sack. Thus, the size of the `memo` table would be `Cxn`, which will also be the time complexity of this algorithm, i.e., $\mathbf{O(C \times n)}$.

This concludes the chapter, in the next lesson you will be quizzed on your knowledge of top-down dynamic programming and memoization.