Solution Review: The Rod Cutting Problem

In this lesson, we will look at different algorithms to solve the rod cutting problem we saw in the last lesson.

We'll cover the following

Solution 1: Simple recursion

```
def rodCutting(n, prices):
  if n<0:
    return 0
  max_val = 0
  for i in range(1,n+1):
      max_val = max(max_val, prices[i-1] + rodCutting(n - i, prices))
  return max_val

print(rodCutting(3, [3,7,8]))
```

Explanation

The idea of this algorithm is to exhaustively find the most optimal cuts from every combination of cuts. We do this by making recursive calls with all possible values

of length and choosing those that give us the highest revenue (*lines 5-7*).

The crux of this algorithm is in *line 6*. We make recursive calls for each possible length of the piece (`i`); the updated value of `n` now is `i` units smaller. Once this result is evaluated, we add this length's price and find the `max`.

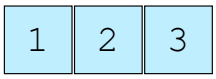
The visualization below shows a dry run of this algorithm.

rodCutting(3, [3,7,8])

rodCutting(3, [3,7,8])

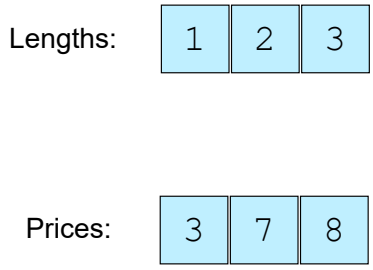
rodCutting(3, [3,7,8])

Lengths:



rodCutting(3, [3,7,8])

rodCutting(3, [3,7,8])



rodCutting(3, [3,7,8])

We can make cuts at 3 different points

Prices: 3 7 8

Lengths: 1 2 3

r = 3

One at length = 1, which will have a price of 3

Prices: 3 7 8

Lengths: 1 2 3

r = 3

r = 7

One at length = 2, which will have a price of 7

Prices: 3 7 8

Lengths: 1 2 3

r = 3

r = 7

r = 8

Last one at length = 3, which will have a price of 8

Prices: 3 7 8

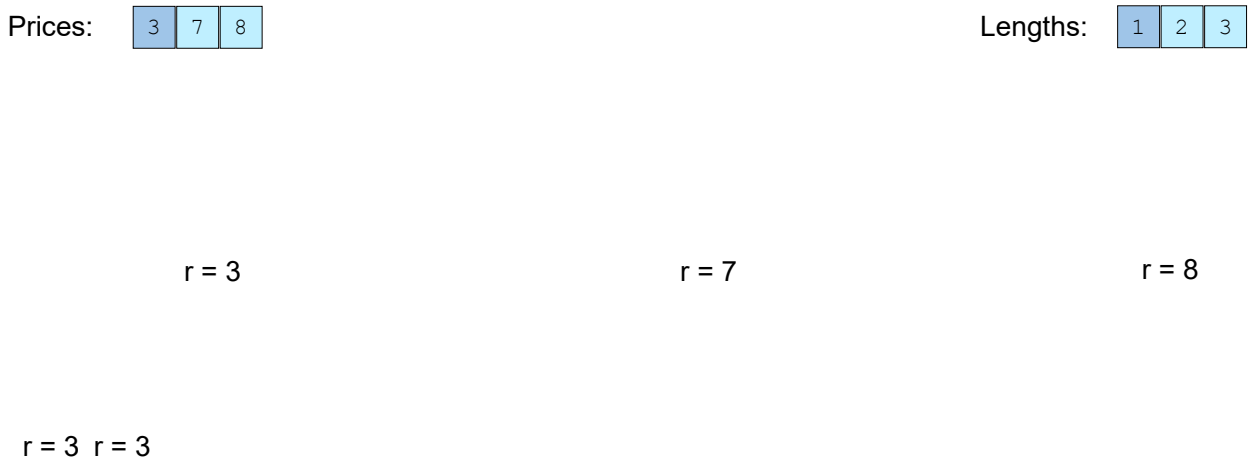
Lengths: 1 2 3

r = 3

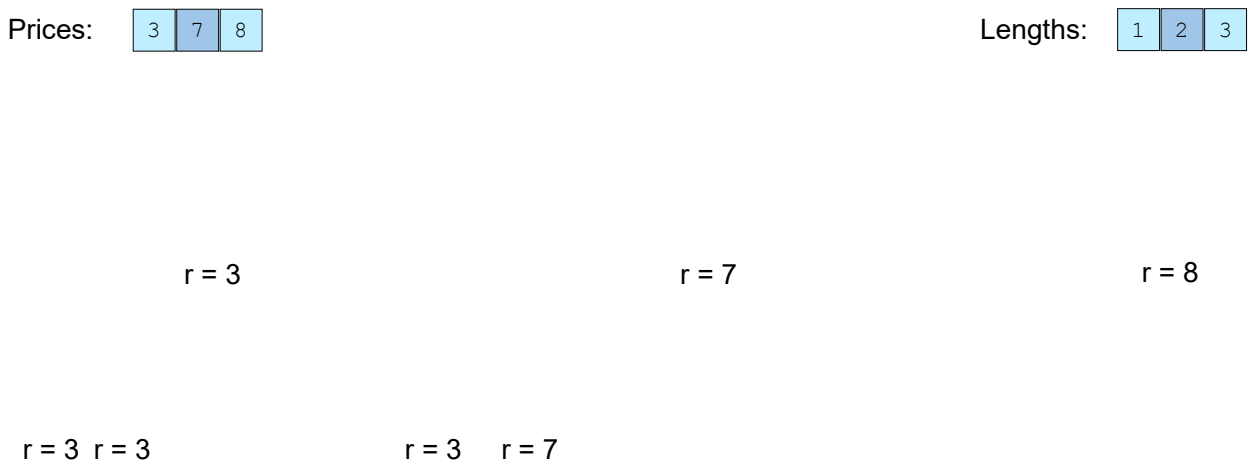
r = 7

r = 8

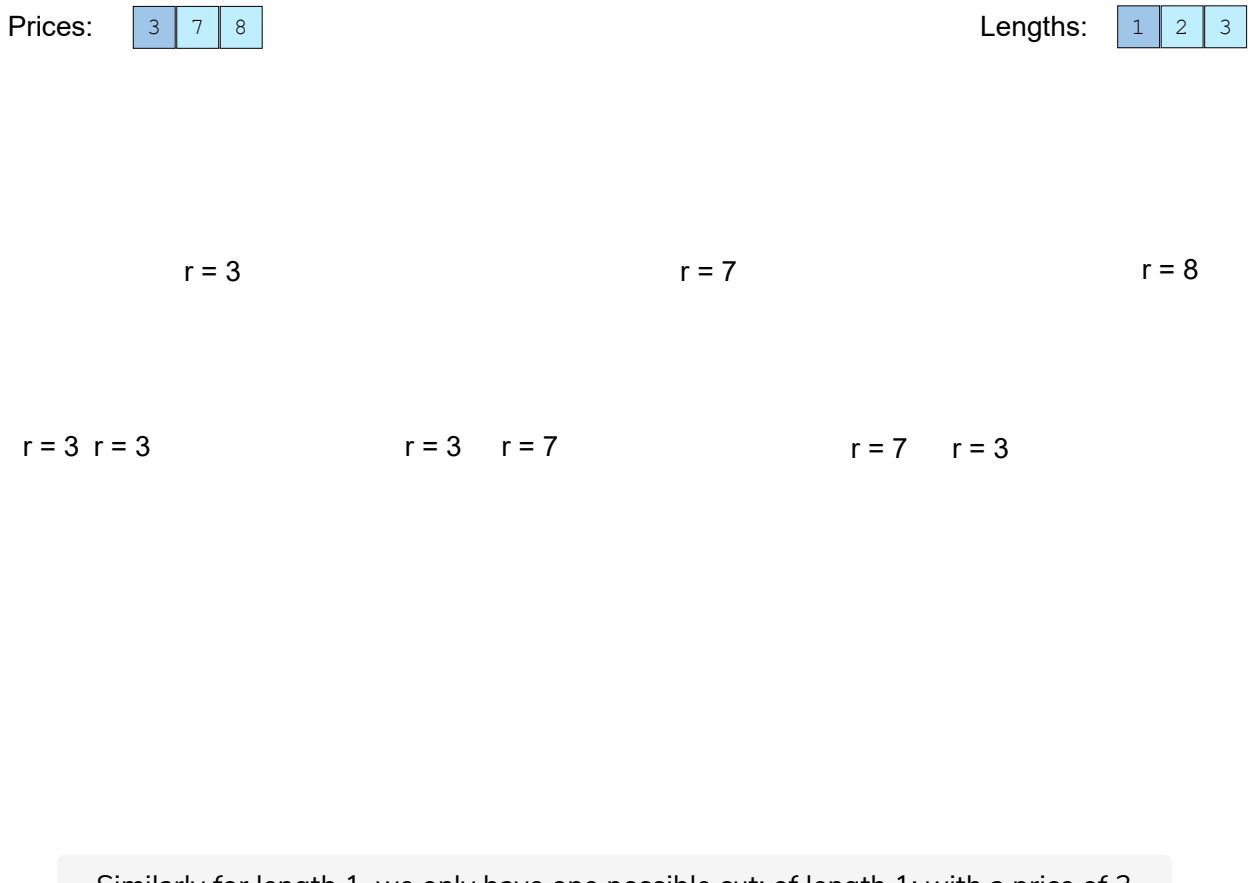
Now recursively evaluate remaining part of rods



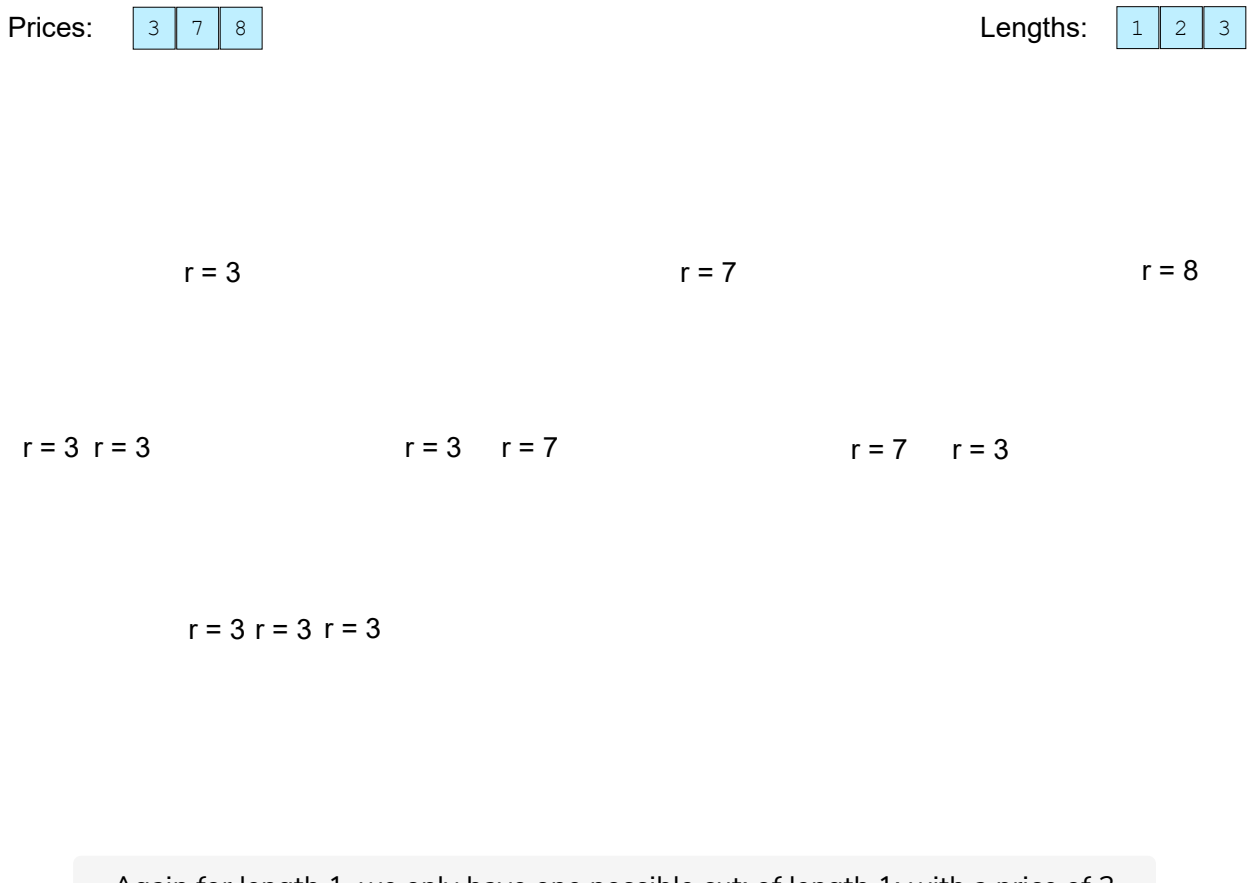
We can make two cuts for a rod of length 2: One at length = 1, which will have a price of 3



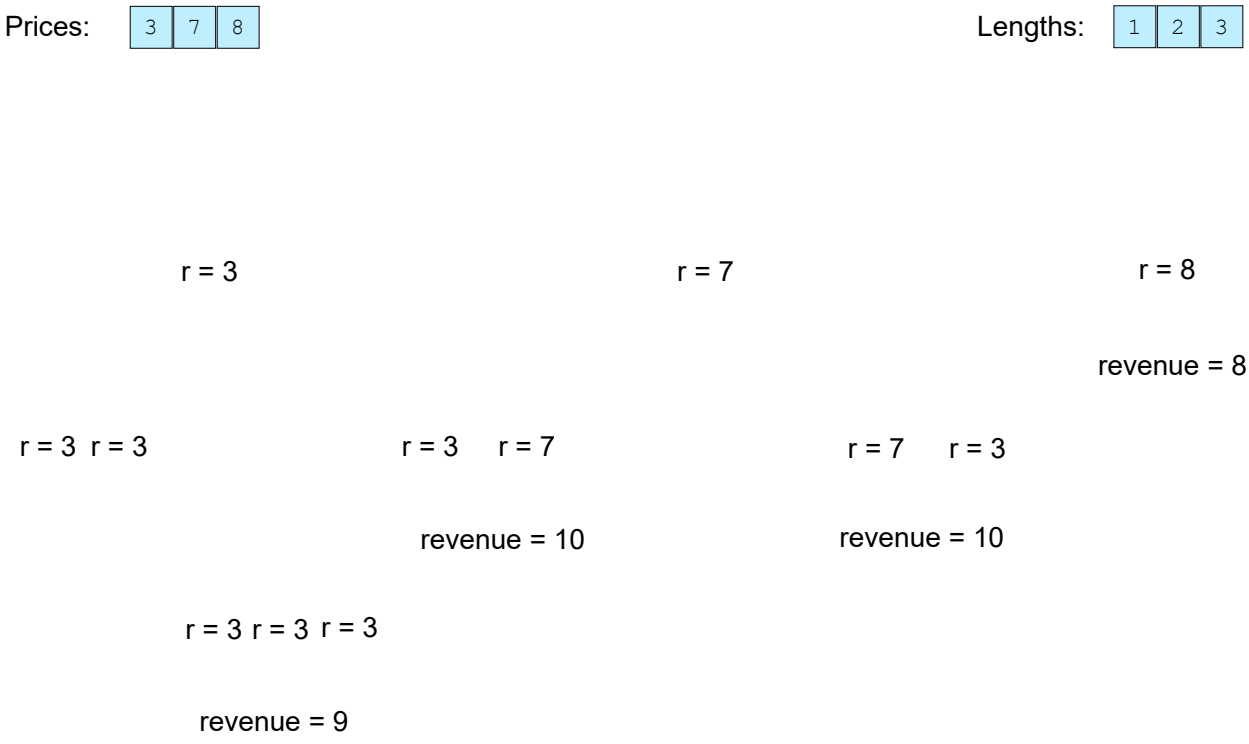
Other one at length = 2, which will have a price of 7



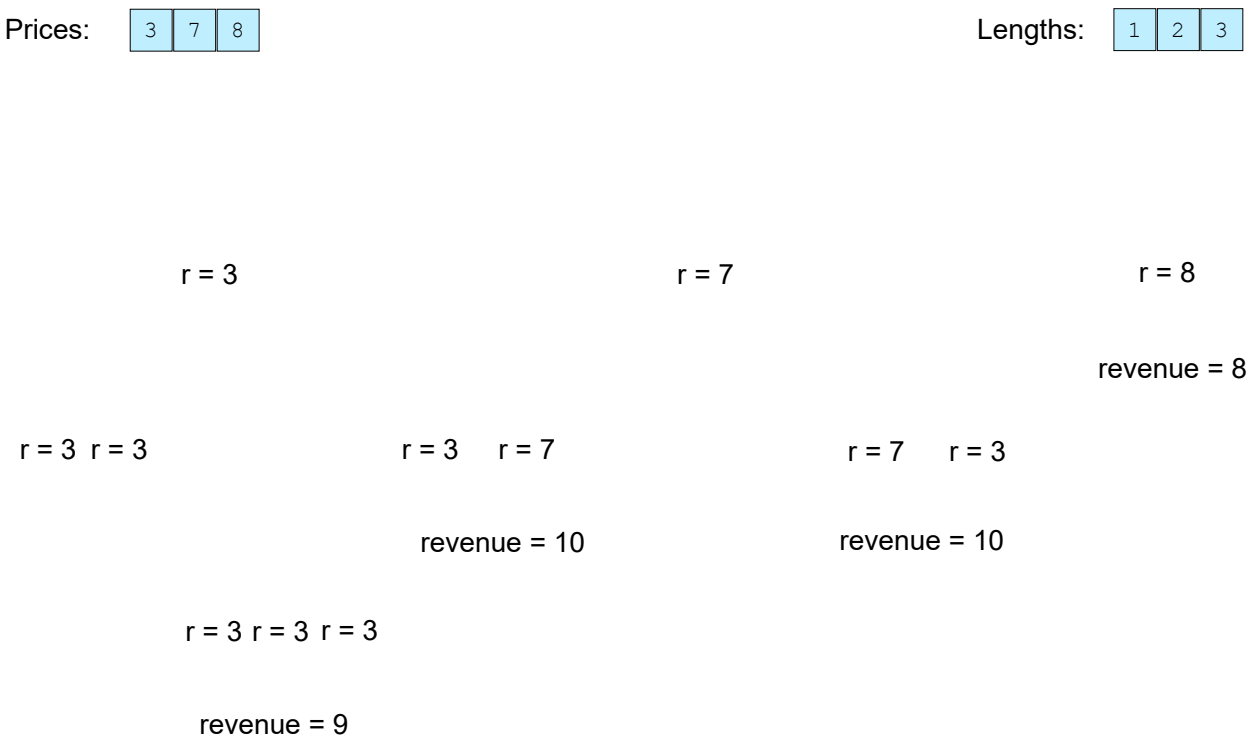
Similarly for length 1, we only have one possible cut; of length 1; with a price of 3



Again for length 1, we only have one possible cut; of length 1; with a price of 3



Now, evaluate value of every combination by summing values of r for every combination



The maximum revenue that we can get is 10

Time complexity

Can you think of the number of combinations of cuts for a rod of length n? It is 2^{n-1}! For a rod of length n, at each position, we have two choices. Either we can make a cut, or we can leave it as it is. All combinations are bounded by O(2^n) as is the time complexity of this solution.

Solution 2: Top-down dynamic programming

Let's see if this problem satisfies both the properties of dynamic programming.

Optimal substructure

We want to find the optimal answer for a rod of length n, and we have optimal answers to all the subproblems, i.e., rods of length *n-1*, *n-2*, ... , *2*, *1*. We can find the maximum of all the subproblem's results plus the price of the rod length that remains along with that subproblem. The following would be the equation for finding the optimal cut's revenue for a rod of length *n*.

$$RC(n) = max(RC(n-1) + price(1), RC(n-2) + price(2) \ \ ... \ \ RC(1) + price(n-1))$$

Thus, if we have answers to the subproblems of rods of length *n-1*, *n-2* ... *2* and *1*, we can construct the solution for the rod of length *n* by only using these results.

Overlapping subproblems

By looking at the visualization above, you can already see one overlapping subproblem. For bigger inputs, this overlap will increase substantially.

Prices: 3 7 8

Lengths: 1 2 3

r = 3

r = 7

r = 8

revenue = 8

r = 3 r = 3

r = 3 r = 7

revenue = 10

r = 7 r = 3

revenue = 10

r = 3 r = 3 r = 3

revenue = 9

Let's try to find some overlapping subproblems.

Prices: 3 7 8

Lengths: 1 2 3

r = 3

r = 7

r = 8

revenue = 8

r = 3 r = 3

r = 3 r = 7

revenue = 10

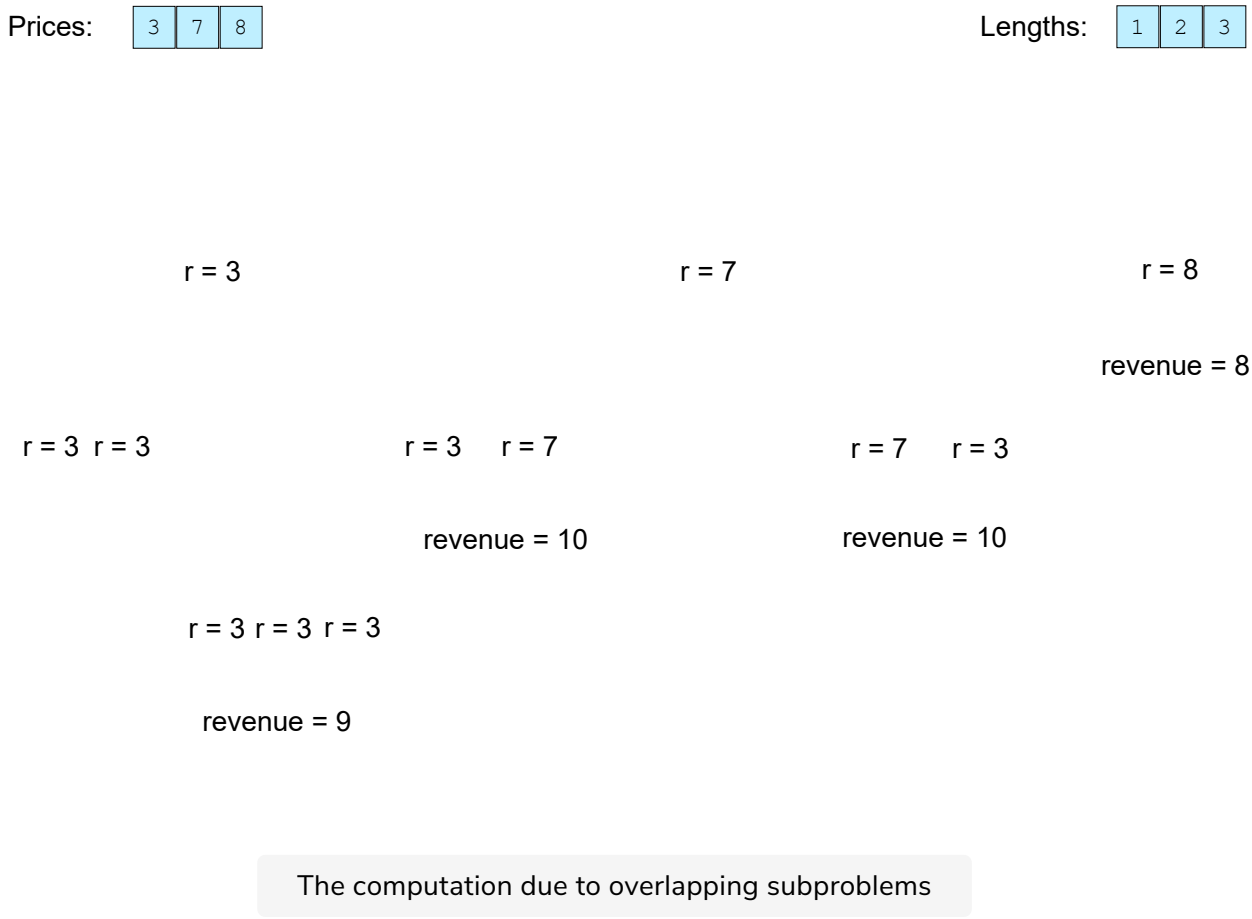
r = 7 r = 3

revenue = 10

r = 3 r = 3 r = 3

revenue = 9

a rod of same length would return the same answer



The computation due to overlapping subproblems

This calls for the memoization of results! Let's look at the top-down implementation of this algorithm.

```
def rodCutting_(n, prices, memo):
  if n<0:
    return 0
  if n in memo:
    return memo[n]
  max_val = 0
  for i in range(1,n+1):
      max_val = max(max_val, prices[i-1] + rodCutting_(n - i, prices, memo))
  memo[n] = max_val
  return memo[n]

def rodCutting(n, prices):
  memo = {}
  return rodCutting_(n, prices, memo)

print(rodCutting(3, [3,7,8]))
```

Explanation

You have seen this many times now: there's nothing fancy here! Before evaluating a result, we check if it is already computed and available in the `memo` (*lines 4-5*); in this case, we do not need to evaluate it. If we have to evaluate it, we simply store results in the `memo` for future reuse (*line 9*).

Time and space complexity

Each problem depends on the smaller subproblems; to evaluate `rodCutting(n)`, we need the answer to `rodCutting(n-1)`, `rodCutting(n-2)`, and so on until `rodCutting(1)`. Thus, the time complexity would be $\mathbf{O(n^2)}$. Moreover, we can only have n unique subproblems, making the space complexity of this algorithm $\mathbf{O(n)}$.

Solution 3: Bottom-up dynamic programming

```
def rodCutting(n, prices):
  # Create a dp array the size of (n+1)
  dp = [0 for _ in range(n + 1)]
  # starting from rod of length 1, find optimal answer to all subproblems
  for i in range(1, n + 1):
    max_val = 0
    # for a rod of length i, we can find what cuts give max answer since we have answer to all sma
    for j in range(i):
      max_val = max(max_val, prices[j]+dp[i-j-1])
    dp[i] = max_val
  # return answer to n length rod
  return dp[n]

print(rodCutting(3, [3,7,8]))
```

Explanation

The idea is to start building from the case of a rod of length `1`, and then use these results to solve the problems for bigger lengths. The solution to a problem of the rod of length `n` would not require answers to the problems of rods greater in length than `n`. Therefore, the for loop at line eight only iterates until `i`, which is any given length of the rod up to n. We make every possible cut, and for the remaining rod length, we would already have the solution in `dp`. Thus, we choose the cut that gives the highest possible answers (*line 10*).

Look at the following visualization for an understanding of this algorithm.

rodCutting(3, [3,7,8])

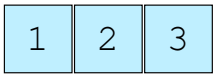
rodCutting(3, [3,7,8])

rodCutting(3, [3,7,8])

rodCutting(3, [3,7,8])

rodCutting(3, [3,7,8])

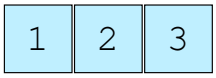
Lengths:



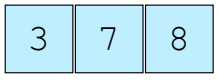
rodCutting(3, [3,7,8])

rodCutting(3, [3,7,8])

Lengths:

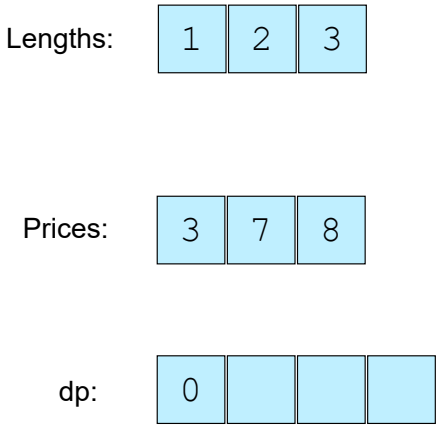


Prices:

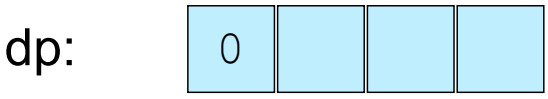


rodCutting(3, [3,7,8])

rodCutting(3, [3,7,8])



rodCutting(3, [3,7,8])



Let's start with a rod of length 1

Prices: 3 7 8

Lengths: 1 2 3

dp: 0

There is no way to cut a rod of length 1, the price you can get is 3

Prices: 3 7 8

Lengths: 1 2 3

dp: 0 3

There is no way to cut a rod of length 1, the price you can get is 3

Prices: 3 7 8

Lengths: 1 2 3

dp: 0 3

Moving on to a rod of length 2

Prices: 3 7 8

Lengths: 1 2 3

dp: 0 3

Now we can either make a cut at length 1, whose price would be 3

Prices: 3 7 8

Lengths: 1 2 3

dp: 0 3

While we already know optimal answer to the rod of length of 1

Prices: 3 7 8

Lengths: 1 2 3

value = 6

dp: 0 3

While we already know optimal answer to the rod of length of 1

Prices: 3 7 8

Lengths: 1 2 3

value = 6

dp: 0 3

Similarly, if we make no cut i.e. rod is of length 2

Prices: 3 7 8

Lengths: 1 2 3

value = 6

value = 7

dp: 0 3

its price would be 7

Prices: 3 7 8

Lengths: 1 2 3

value = 6

value = 7

dp: 0 3 7

Thus optimal answer to the rod of length 2 would be 7

Prices: 3 7 8

Lengths: 1 2 3

dp: 0 3 7

Moving on to a rod of length 3

Prices: 3 7 8

Lengths: 1 2 3

dp: 0 3 7

We can make 3 cuts, first of length 1 and price 3

Prices: 3 7 8

Lengths: 1 2 3

value = 10

dp: 0 3 7

We have already solved for the rod of length 2

Prices: 3 7 8

Lengths: 1 2 3

value = 10

dp: 0 3 7

We can also make a cut of length 2, of price 7

Prices: 3 7 8

Lengths: 1 2 3

value = 10

dp: 0 3 7

while we have already solved for a stick of length 1

Prices: 3 7 8

Lengths: 1 2 3

value = 10

value = 10

dp: 0 3 7

while we have already solved for a stick of length 1

Prices: 3 7 8

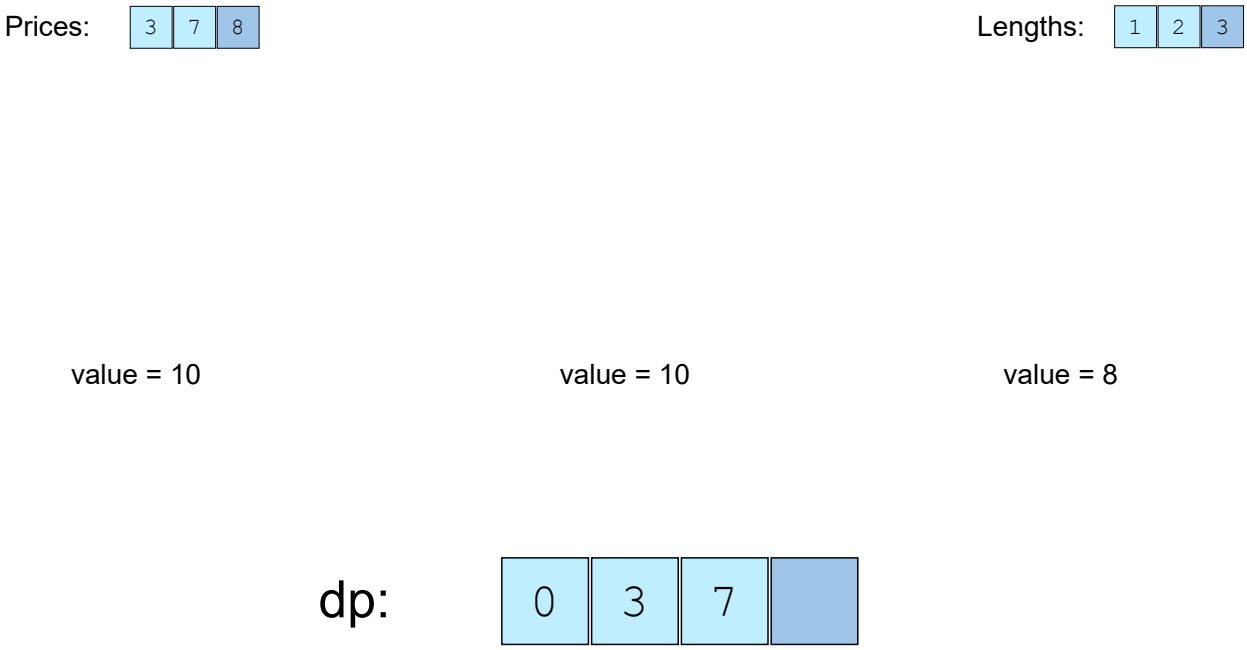
Lengths: 1 2 3

value = 10

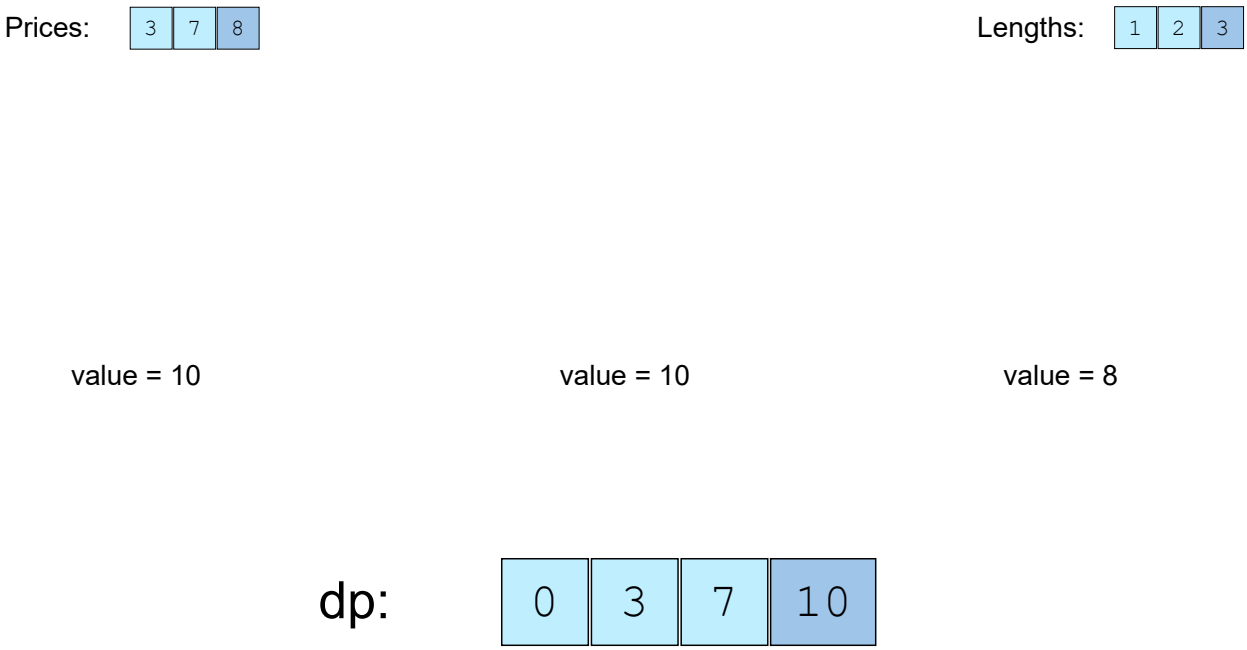
value = 10

dp: 0 3 7

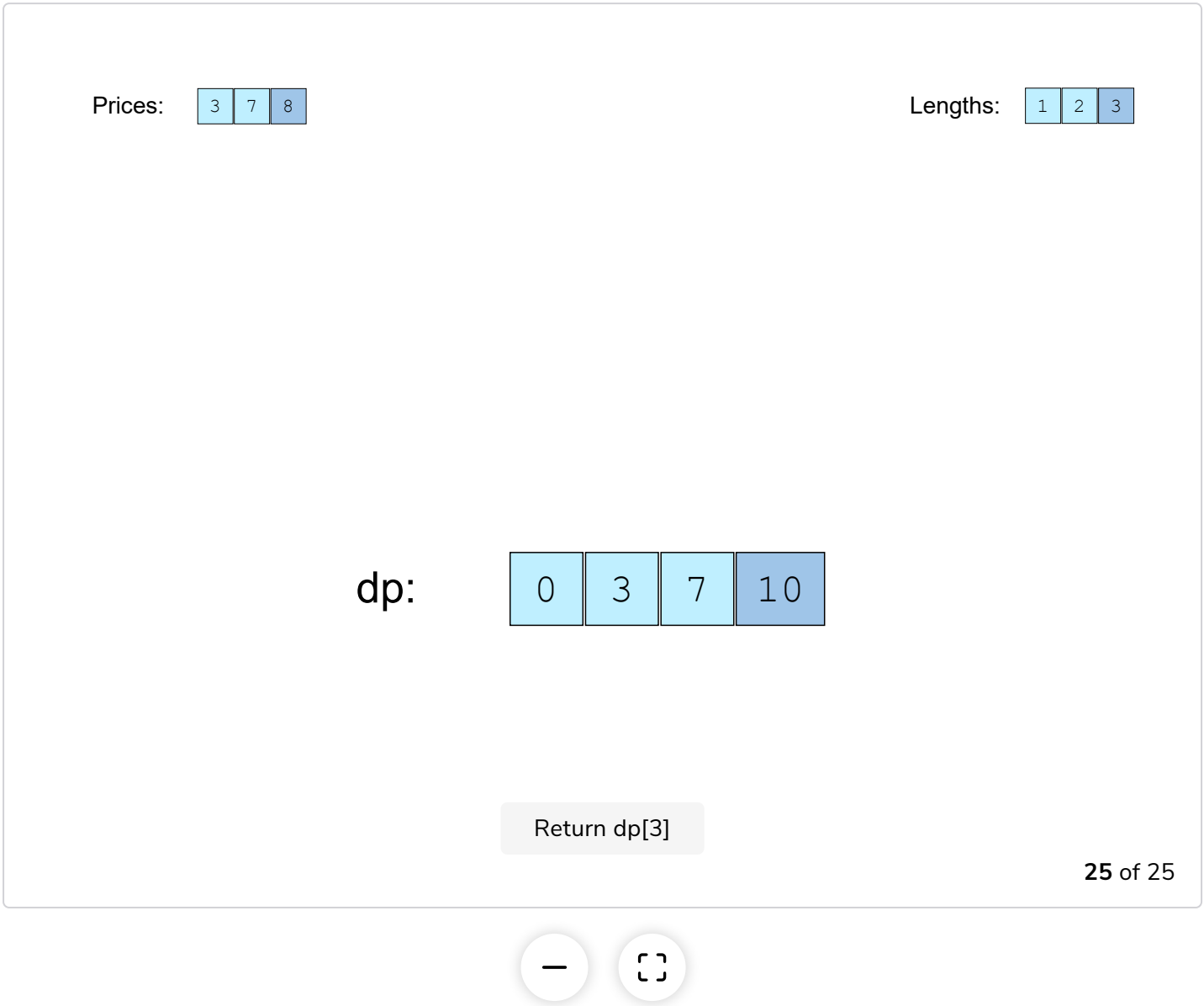
We can also have no cut i.e. length of 3 with price 8



We can also have no cut i.e. length of 3 with price 8



Thus for a rod of length 3, optimal answer is 10



Time and space complexity

Similar to the top-down solution, the time complexity here would be quadratic. You can sense this from nested for loops as well. The intuitive reasoning behind this is that to construct an optimal answer to any subproblem, you require optimal solutions to all the smaller subproblems. Thus, the time complexity becomes $\mathbf{O(n^2)}$. Moreover, since there are n possible subproblems, the size of the tabulation table, `dp`, would be n, making the space complexity $\mathbf{O(n)}$.

In the next lesson, you will solve another coding challenge.