

---

# Dynamic Programming

---

## 1 The Dynamic Programming Technique

The dynamic programming technique is a type of “divide-and-conquer” approach to solving problems by combining subproblems. Recursive algorithms are also a type of divide-and-conquer algorithm. The fundamental difference between the two types are:

- **Recursive algorithms:** typically divide the problem into **disjoint** subproblems.
- **Dynamic programming:** typically applies to situations where the subproblems **overlap**.

Furthermore, dynamic programming is usually applied to **Optimization problems**.

### Optimization Problems:

- Many possible solutions exist
- Each solution has a value
- The goal is to find the solution with the optimal (maximum or minimum) value

The key to the dynamic programming technique for an optimization problem is to solve the *smaller* subproblems first, and to *store* these values in a table. By storing the values in a table, we do not need to re-compute them when we need them later. The larger optimization problem is built by combining the smaller solutions.

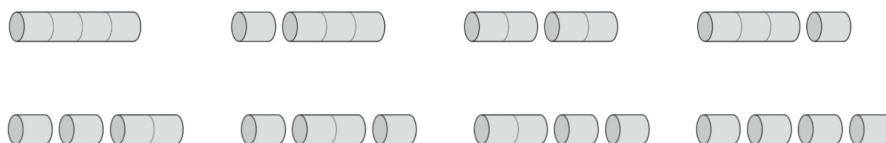
### Dynamic Programming technique:

- Identify the characteristic of the optimal solution
- Identify how to recursively define the optimum big problem in terms of the optimum sub-problems
- Compute the sub-problems, typically bottom-up, and store the values in a table
- Compute the optimal solution from the stored information

This lecture on dynamic programming presents two main examples of dynamic programming: Rod-cutting and Longest common subsequence.

## 2 Rod-Cutting

Imagine we have a rod of length  $n$  and the opportunity to cut the rod into smaller pieces and sell them for a certain price. A rod of length  $n = 4$  could be cut up into pieces of length 1, 2, 3 or 4, and each segment could be sold for the corresponding price:  $p_1, p_2, p_3$  or  $p_4$ . The total profit we make from this rod depends on how we make the cuts. A rod of length 4 for example could be cut in many different ways:



The total amount of money we make depends on the choices of our cuts and the total price we would make in that case. Suppose our prices were as below:

	$p_1$	$p_2$	$p_3$	$p_4$
<b>Price:</b>	2	5	3	5

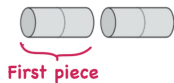
If we leave the rod as one piece of length 4 and don't cut it, then we make  $p_4 = \$5$ . It may be that a different cut-option from above results in a higher profit. Note that the order of the cuts does not matter, we are simply interested in the segment sizes and the total price associated with those sizes. The different possibilities for a rod of length 4 are shown below:

Length 4:	5
Lengths 1 and 3:	$3 + 2 = 5$
Lengths 2 and 2:	$5 + 5 = 10$
Lengths 1,1, and 2:	$2 + 2 + 5 = 9$
Lengths 1,1,1 and 1:	$2 + 2 + 2 + 2 = 8$

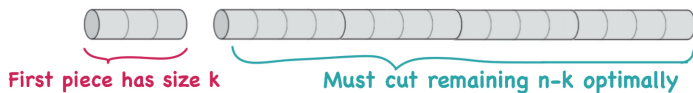
The maximum amount of money we can make is \$10 by cutting the rods into two pieces of length 2. The above solution was found by simply verifying all possible cuts. For a rod of length  $n$  there are an **exponential** number of different ways we can cut the rod. If we were to verify the profit for each of these possibilities, the runtime of the algorithm would be exponential!

## 2.1 Relating subproblems to the optimal solution:

Given a rod of length  $n$  there are many ways that this rod could be cut. The goal is determine how to make those cuts in order to maximize profit. Whatever the optimal solution is, the rod must have a segment cut off the front that can be referred to as the “first piece” of the optimal solution. In the example below, a cut was made that resulted in two pieces of length two. The “first piece” has size 2.



The first cut defines the length of the first piece, and this piece could have length either  $1, 2, 3, \dots, n$ . Whatever the optimal solution is for a rod of length  $n$ , the first piece must have some particular length, say  $k$ . Although we don't know what  $k$  is without knowing the optimal solution, we *do* know that the remaining section of length  $n - k$  must also be cut optimally:



It is this link between the optimal problem and the optimum subproblems that is fundamental to our dynamic programming solution. We now show how to use this important relationship to solve the optimal problem.

Let  $r(n)$  represent the maximum profit possible on a rod of length  $n$ . If the optimal solution resulted in the first piece having length 3, then the rest of the rod of length  $n - 3$  must also have been cut optimally. This means that the optimal profit satisfies:

$$r(n) = p_3 + r(n - 3)$$

Now of course we don't actually know where the first cut is in the optimal solution. However, we do know that the optimal solution represents the maximum profit, and so we could simply check all possibilities for the length of the first piece. Therefore:

$$r(n) = \max_{1 \leq k \leq n} (p_k + r(n - k))$$

For rods of length 0, the profit is 0 and so we define  $r(0) = 0$ .

Let's verify this formula for the example above with  $n = 4$ .

$$r(4) = \max\{p_1 + r(3), p_2 + r(2), p_3 + r(1), p_4 + r(0)\}$$

In order to determine this maximum, we need to solve for  $r(3)$ ,  $r(2)$  and  $r(1)$ . The same formula can be applied used recursively:

$$r(3) = \max\{p_1 + r(2), p_2 + r(1), p_3 + r(0)\}$$

$$r(2) = \max\{p_1 + r(1), p_2 + r(0)\}$$

$$r(1) = \max\{p_1 + r(0)\} = 2$$

Only the last equation above has an explicit value:  $r(1) = 2$ . This value represents the profit on a rod of length 1. Once this value is found, it can be replaced in the above equations:

$$r(2) = \max\{2 + 2, 5 + 0\} = 5$$

and so the maximum profit on a rod of length 2 is 5.

$$r(3) = \max\{2 + 5, 5 + 2, 3 + 0\} = 7$$

$$r(4) = \max\{2 + 7, 5 + 5, 3 + 2, 5 + 0\} = 10$$

Therefore the maximum profit is 10.

The next two subsections look at different ways of taking this recursive definition and turning it into an algorithm. The first is a *recursive* approach, and the second is our dynamic programming approach.

## 2.2 A recursive attempt:

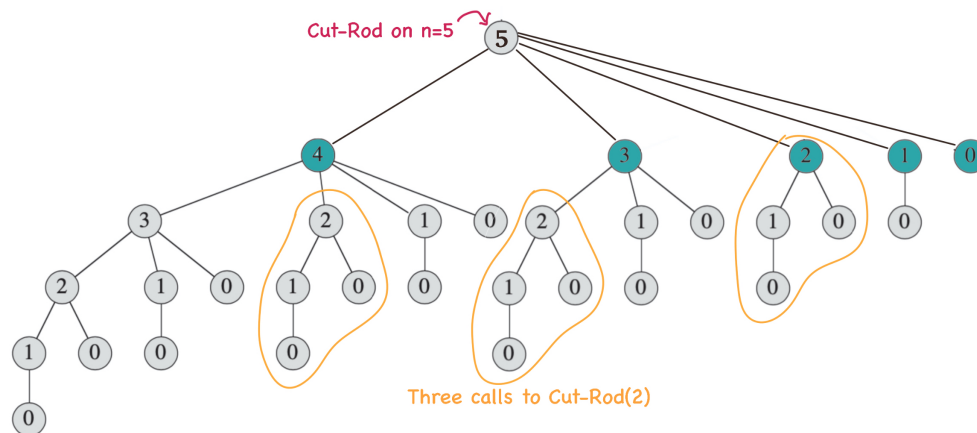
A recursive algorithm for the rod cutting program is based on the equation above. A call to  $\text{CUT-ROD}(n, p)$  with price list  $p$  and rod length  $n$  will simply compute the maximum of all possible first cut options  $k$ , and for each of those make a recursive call to  $\text{CUT-ROD}(n - k, p)$ :

```

CUT-ROD( $n, p$ )
  if  $n = 0$  return 0
   $max = 0$ 
  for  $i = 1$  to  $n$ 
     $newcut = p[i] + \text{CUT-ROD}(n - i, p)$ 
    if  $newcut > max$ 
       $max = newcut$ 
  return  $max$ 

```

This recursive algorithm is unfortunately very inefficient.  $\text{CUT-ROD}$  calls itself over and over again with the same parameters, resulting in the same subproblems being computed several times. The figure below is a recursion tree for the case  $n = 5$ .  $\text{CUT-ROD}(5)$  makes five recursive calls with input size 4, 3, 2, 1 and 0 (shown in blue). The call to  $\text{CUT-ROD}(4)$  in turn makes recursive calls with input size 3, 2, 1 and 0, and the call to  $\text{CUT-ROD}(3)$  also makes recursive calls with input sizes 2, 1 and 0. We can see below that several sections of this tree are repeated. In particular, there are three separate calls to  $\text{CUT-ROD}(2)$ . Each of these calls results in the same final value, however the algorithm does not keep track of the fact that it has *already* computed  $\text{CUT-ROD}(2)$ , and therefore it simply calls the same subproblem again.



### The Runtime:

Let  $T(n)$  represent the number of calls to CUT-ROD when the initial parameter is  $n$ . When  $n = 0$  there is only one call to the algorithm, since this represents the base case and no recursive calls are made. If  $n \geq 1$  the algorithm executes a for-loop from  $i = 1$  to  $i = n$  and each iteration results in a recursive call to CUT-ROD( $n-i$ ). Therefore:

$$T(n) = 1 + T(n-1) + T(n-2) + \dots + T(0)$$

Using induction we can show that the solution to this equation is  $2^n$ .

**Assume**  $T(k) = 2^k$  for  $k < n$

**Show**  $T(n) = 2^n$

$$\begin{aligned} T(n) &= 1 + T(n-1) + T(n-2) + \dots + T(0) \\ &= 1 + 2^{n-1} + 2^{n-2} + \dots + 1 \\ &= 1 + (2^n - 1) = 2^n \end{aligned}$$

The runtime of CUT-ROD( $n$ ) is  $2^n$ , and is therefore an exponential time algorithm. This is an extremely inefficient runtime, even for today's computers. You may be able to run it on a rod of size 40, but for every increase in 1 in your rod length, the computer will take *twice* as long to complete.

The next section demonstrates how dynamic programming drastically improves on the runtime of this recursive algorithm.

## 2.3 The Dynamic programming approach:

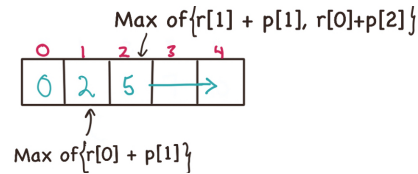
The idea behind dynamic programming is to create a table in which we store the values of CUT-ROD( $i$ ). This makes it possible for the algorithm to simply look up the values of CUT-ROD( $i$ ) for the subproblems once they have been computed. The result is that each call to CUT-ROD( $i$ ) for a specific  $i$  is only carried out once.

### The table:

Let  $r[0 \dots n]$  be an array that stores the optimal values for the CUT-ROD problem. Entry  $r[i]$  is the maximum profit possible for a rod of length  $i$ . For example, if  $r[4] = 10$ , then the optimal cuts on a rod of length 4 result in a profit of 10. Notice that in this algorithm, we assume we have an array that starts at 0, because we need to store the value  $r[0] = 0$  for rods of length 0. This array does not store the actual cuts that result in that profit, it simply stores the best profit possible for a particular length. Using the same example as above, the array  $r[]$  will contain the following values when the algorithm is completed:

	0	1	2	3	4
$r[]$	0	2	5	7	10

Because solutions to larger problems depend on the optimal values of smaller problems, this array should be filled in a **bottom-up** approach (meaning for small  $i$  towards larger  $i$  values). Once the entire array is filled, the final optimal value for the Cut-Rod problem on input of size  $n$  is stored in  $r[n]$ . Notice that each entry of the table can be filled using values that are *already filled in* because we completed the table from left to right:



The dynamic programming algorithm shown below is called DYNAMIC-CUT-ROD( $n, p$ ) where  $n$  is the rod length, and  $p$  is the price list. The procedure fills in the array  $r[]$  from left to right and returns the final optimal value in  $r[n]$ .

#### The dynamic programming solution to rod-cutting:

- Create a new array  $r[0 \dots n]$  and set  $r[0] = 0$ .
- Fill in the entries of  $r[j]$  starting with  $j = 1$  and finishing with  $j = n$ .
  - The entry  $r[j]$  is set as the maximum of

$$p_k + r[j - k]$$

over all  $k$  from  $k = 1$  to  $j$ . Note that the values  $r[j - k]$  already exist in the table when computing  $r[j]$ .

- The optimal revenue for cutting the rod of length  $n$  is found in  $r[n]$

#### DYNAMIC-CUT-ROD( $n, p$ )

```

Let  $r[0 \dots n]$  be an array
 $r[0] = 0$ 
for  $j = 1$  to  $n$ :
     $r[j] = 0$ 
    for  $k = 1$  to  $j$ 
        if  $p[k] + r[j - k] > r[j]$ 
             $r[j] = p[k] + r[j - k]$ 
return  $r[n]$ 





```

**Runtime:** The algorithm consists of an inner and outer for-loop (just like Insertion-sort), resulting in a runtime of  $\Theta(n^2)$ . The analysis is identical.

## 2.4 Reconstructing a Solution

The algorithm above returns the maximum profit possible over all possible ways of cutting a rod of length  $n$ . However, it does not actually return the segment sizes that result in that profit. This might seem strange because of course practically, we would like to know both the maximum profit and also *how* to cut the rod to achieve that profit.

A quick and simple update to the above algorithm allows us to keep track of the piece sizes. Let  $s[0 \dots n]$  be a new array where  $s[i]$  stores the size of the first piece in the optimal solution for a rod of length  $i$ . In our continued example from above, the values would be:

<b>Rod length 0:</b> $r[0] = 0$ $s[0] = 0$ No pieces to cut	
<b>Rod length 1:</b> $r[1] = 2$ $s[1] = 1$ first piece is size 1	
<b>Rod length 2:</b> $r[2] = 5$ $s[2] = 2$ first piece has size 2.	
<b>Rod length 3:</b> $r[3] = 7$ $s[3] = 1$ first piece has size 1	
<b>Rod length 4:</b> $r[4] = 10$ $s[4] = 2$ first piece has size 2	

The dynamic programming algorithm can be updated easily to also keep track of this information. Whenever  $r[j]$  is updated in the inner for-loop, the algorithm has found a preferred way to cut the rod. When this happens, we update  $s[j]$  to the size of the first piece that is the current best choice. When the inner for-loop is complete, all possible first-cut sizes on a rod of length  $j$  have been verified, and  $s[j]$  contains the size of the first cut that leads to the optimal value.

### DYNAMIC-CUT-ROD( $n, p$ )

```

Let  $r[0 \dots n]$  be an array. Let  $s[1 \dots n]$  be an array
 $r[0] = 0$ 
for  $j = 1$  to  $n$ :
     $r[j] = 0$ 
    for  $k = 1$  to  $j$ 
        if  $p[k] + r[j - k] > r[j]$ 
             $r[j] = p[k] + r[j - k]$ 
             $s[j] = k$ 
return  $r[n]$ 
```

When the algorithm completes, the optimal value is stored in  $r[n]$ . Furthermore, the particular pieces can be found by examining the size of the first cuts. In our example above, the final value  $s[4] = 2$  indicates the first piece has size 2. Thus our first segment has size 2. The length of the remaining rod is  $4 - 2 = 2$ . Next we look up  $s[2]$  which tells us the size of the first cut for a rod of length 2. From the above table, this is 2. Therefore the *second* piece has size 2. Continuing in this way, we can determine the size of all the cut segments that correspond to the optimal solution.

### PRINT-PIECES( $s, n$ )

```

j = n
while  $j > 0$ 
    print  $s[j]$ 
     $j = j - s[j]$ 
```

Therefore in  $O(n)$  time we can output the piece sizes that correspond to the optimal solution of the rod-cutting problem.

### Summary

- The recursive solution to Cut-Rod has a runtime of  $\Theta(2^n)$ . The algorithm calls itself over and over again with the same parameters, resulting in the same subproblems being computed several times.
- The dynamic programming solution for Cut-Rod runs in time  $O(n^2)$  for a rod of length  $n$ . It uses  $O(n)$  space to store the values to the subproblems.