

# LECTURE 07 – DYNAMIC PROGRAMMING (PART 1)

COMPSCI 308 – DESIGN AND ANALYSIS OF ALGORITHMS

---

Xing Shi Cai <https://newptcai.gitlab.io>

Duke Kunshan University, Kunshan, China

Jan-Mar 2026

# SUMMARY

ITA 15 Dynamic Programming

ITA 15.1 Rod Cutting

# ASSIGNMENTS<sup>1</sup>



Practice makes perfect!

Introduction to Algorithms (ITA)   
Required Readings:

- Section 15.1.

 Required Exercises:

- Exercises 15.1 — 1–5.

---

<sup>1</sup> ⚡ Assignments will not be collected; however, quiz problems will be selected from them. (This includes both Readings and Exercises.)

## ITA 15 DYNAMIC PROGRAMMING

---

# FIBONACCI'S PUZZLE

Suppose that you want to have some 🐰 as pets—

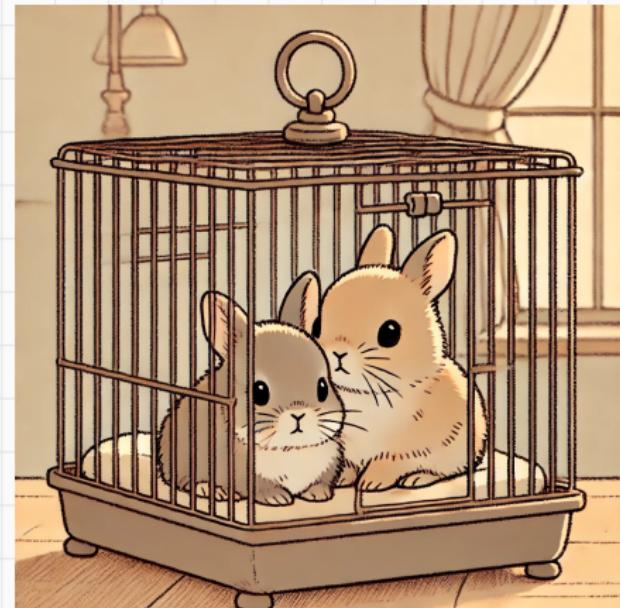
- Month 0 – You have no pets.



# FIBONACCI'S PUZZLE

Suppose that you want to have some 🐰 as pets—

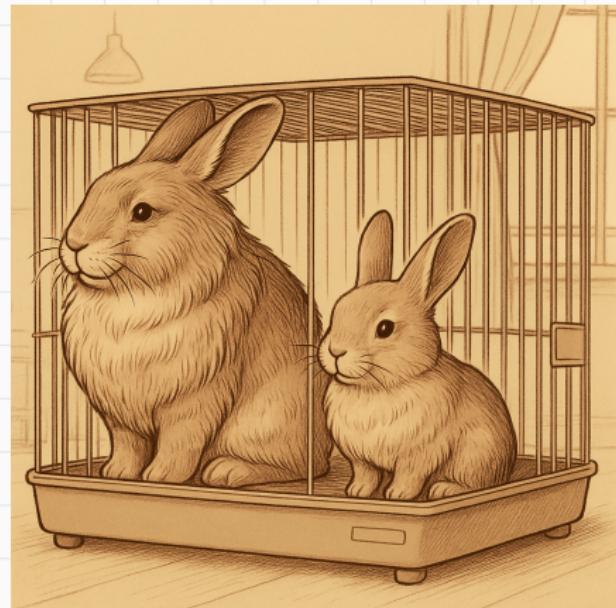
- Month 0 – You have no pets.
- Month 1—You get a pair of baby 🐰.



# FIBONACCI'S PUZZLE

Suppose that you want to have some 🐰 as pets—

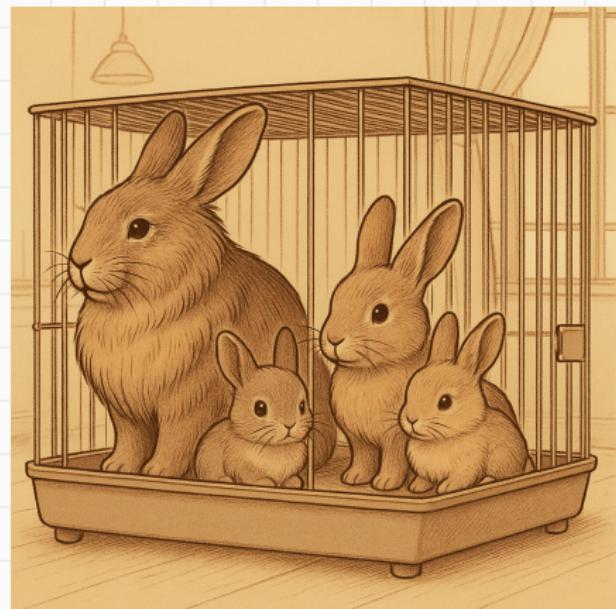
- Month 0 – You have no pets.
- Month 1—You get a pair of baby 🐰.
- Month 2—The pair of 🐰 matures.



# FIBONACCI'S PUZZLE

Suppose that you want to have some 🐰 as pets—

- Month 0 – You have no pets.
- Month 1—You get a pair of baby 🐰.
- Month 2—The pair of 🐰 matures.
- Month 4 and so on— Every subsequent month, a mature pair of 🐰 produces a pair of baby 🐰.



# FIBONACCI'S PUZZLE

Suppose that you want to have some 🐰 as pets—

- Month 0 – You have no pets.
- Month 1—You get a pair of baby 🐰.
- Month 2—The pair of 🐰 matures.
- Month 4 and so on— Every subsequent month, a mature pair of 🐰 produces a pair of baby 🐰.

If the 🐰 never 💀, how many pairs of 🐰 are there at month  $n$ ?





## THE RECURRENCE RELATION FOR THE FIBONACCI SEQUENCE

Let  $F_n$  denote the pairs of 🐰 you have at month  $n$ .

The sequence

$$\{F_n\}_{n \geq 0} = 0, 1, 1, 2, 3, 5, 8, \dots$$

is called the **Fibonacci sequence**.

💡 Can we find a recurrence relation for  $F_n$ ?



## THE RECURRENCE RELATION FOR THE FIBONACCI SEQUENCE

Let  $F_n$  denote the pairs of 🐰 you have at month  $n$ .

The sequence

$$\{F_n\}_{n \geq 0} = 0, 1, 1, 2, 3, 5, 8, \dots$$

is called the **Fibonacci sequence**.

❓ Can we find a recurrence relation for  $F_n$ ?

By observing the initial few numbers, we can see that

$$F_{n+2} = F_{n+1} + F_n, \quad \text{for all } n \geq 0$$

🤔 How to prove this?

# A VISUAL EXPLANATION OF THE FIBONACCI SEQUENCE

From the picture, we see that  $F_n$  consists of two parts

- $F_{n-1}$  🐰 who

- $F_{n-2}$  🐰 who

This leads to the recursion –

$$F_n = F_{n-1} + F_{n-2}.$$

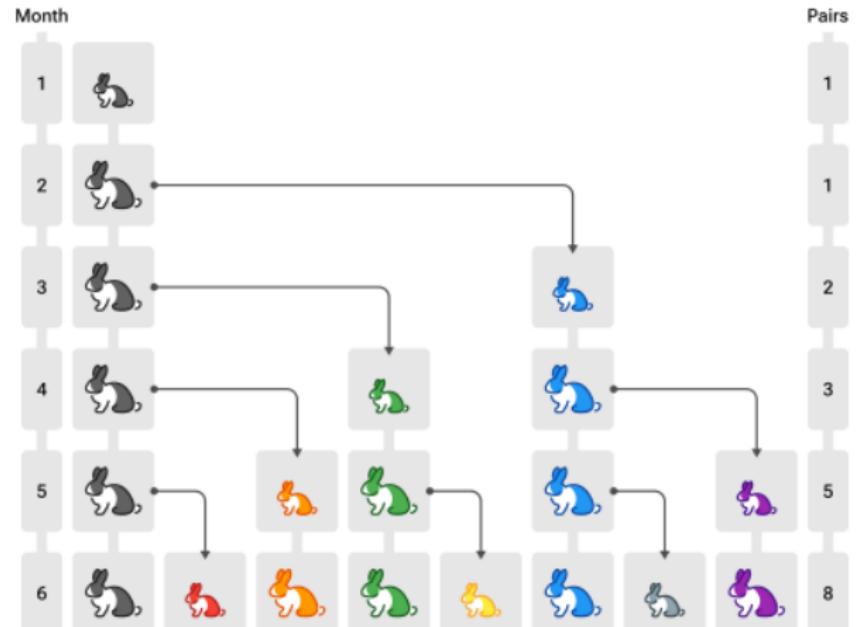


Figure 1: The recurrence relation for the Fibonacci sequence



## FIBONACCI SEQUENCE

The Fibonacci sequence is defined recursively as —

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

✳️ Can you prove the following solution?

$$F_n = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1 - \sqrt{5}}{2} \right)^n, \text{ for all } n \in \mathbb{N}.$$

😩 It's not the most efficient for large values of  $n$  due to the irrational number  $\sqrt{5}$ .

# 😊 FIBONACCI NUMBERS IN NATURE

Fibonacci numbers often appear in the spiral pattern of particular plants.

😲 There is actually quite a **elegant mathematical model** explaining it.



**Figure 2:** Yellow chamomile (洋甘菊) head showing the arrangement in 21 (blue) and 13 (cyan) spirals. Source – [Wikipedia](#)

## THE NAIVE WAY

```
1: FibonacciNaive( $n$ )
2: if  $n = 0$  then
3:     return 0
4: else if  $n = 1$  then
5:     return 1
6: else
7:     return FIBONACCINAIVE( $n - 1$ ) + FIBONACCINAIVE( $n - 2$ )
```

🍰 Let's try to compute  $F_5$  with the naive algorithm.

✳️ What is the time complexity of this algorithm?

## A BETTER WAY

---

Let's try to compute  $F_5$  with the following improved algorithm –

- 1: **Fibonacci( $n$ )**
- 2: Initialize array  $dp[0 \dots n]$  with zeros
- 3:  $dp[0] \leftarrow 0$
- 4:  $dp[1] \leftarrow 1$
- 5: **for**  $i \leftarrow 2$  to  $n$  **do**
- 6:      $dp[i] \leftarrow dp[i - 1] + dp[i - 2]$
- 7: **return**  $dp[n]$

An algorithm in this style is called **dynamic programming**.

 What is the time complexity of this algorithm?

## DYNAMIC PROGRAMMING

---

Dynamic programming is a method used primarily in optimization problems.

The **4** key elements of dynamic programming –

-  Characterize the structure of an optimal solution.

## DYNAMIC PROGRAMMING

---

Dynamic programming is a method used primarily in optimization problems.

The **4** key elements of dynamic programming –

-  Characterize the structure of an optimal solution.
-  Define the value of an optimal solution recursively.

## DYNAMIC PROGRAMMING

---

Dynamic programming is a method used primarily in optimization problems.

The **4** key elements of dynamic programming –

-  Characterize the structure of an optimal solution.
-  Define the value of an optimal solution recursively.
-  Compute the value in a bottom-up fashion.

## DYNAMIC PROGRAMMING

---

Dynamic programming is a method used primarily in optimization problems.

The **4** key elements of dynamic programming –

-  Characterize the structure of an optimal solution.
-  Define the value of an optimal solution recursively.
-  Compute the value in a bottom-up fashion.
-  Construct the optimal solution from computed information.

## DYNAMIC PROGRAMMING

Dynamic programming is a method used primarily in optimization problems.

The **4** key elements of dynamic programming –

-  Characterize the structure of an optimal solution.
-  Define the value of an optimal solution recursively.
-  Compute the value in a bottom-up fashion.
-  Construct the optimal solution from computed information.

# DYNAMIC PROGRAMMING

Dynamic programming is a method used primarily in optimization problems.

The **4** key elements of dynamic programming –

-  Characterize the structure of an optimal solution.
-  Define the value of an optimal solution recursively.
-  Compute the value in a bottom-up fashion.
-  Construct the optimal solution from computed information.

Both *dynamic programming* and *divide-and-conquer* solve problems by combining solutions to subproblems.

The differences are –

-  divide-and-conquer –
  - Disjoint subproblems
  - Recursive solution



dynamic programming –

- Overlapping subproblems
- Solve each subproblem once, store in table

In 1950, Bellman introduced **dynamic programming** to 😊 the mathematical essence of his work.

During an era when 🤑 sponsors were skeptical of mathematics, he opted for a term that would be both uncriticizable and appealing.



Figure 3: Richard E. Bellman (1920–1984)

## ITA 15.1 ROD CUTTING

---

# THE ROD CUTTING PROBLEM

Given —

- A rod of length  $n$ .
- A table containing  $p_i$ 's for the price of a piece of length  $i$ .

Determine —

- The maximum revenue  $r_n$  achievable by possibly cutting the rod into integer lengths and selling them.



Figure 4: How to cut a rod to maximize revenue?

 EXAMPLE OF ROD CUTTING

Given the following price table –

$p_1$	$p_2$	$p_3$	$p_4$	$p_5$	$p_6$	$p_7$	$p_8$	$p_9$	$p_{10}$
1	5	8	9	10	17	17	20	24	30

Let  $r_n$  be the maximum profit achievable by cutting the rod of length  $n$ .

 What are  $r_1$ ,  $r_2$  and  $r_3$ ?

## RECURRENCE FOR MAXIMUM PROFIT

For a rod of length  $n$ , let  $i$  be the length of the **first piece** we cut off. The possible values for  $i$  are –

$$\underline{\hspace{2cm}} \leq i \leq \underline{\hspace{2cm}}$$

## RECURRENCE FOR MAXIMUM PROFIT

For a rod of length  $n$ , let  $i$  be the length of the **first piece** we cut off. The possible values for  $i$  are –

$$1 \leq i \leq n$$

If we do not divide piece  $i$  further, the maximum profit for the remaining length is  $r_{n-i}$ . Thus, the profit for this cut is –

---

## RECURRENCE FOR MAXIMUM PROFIT

For a rod of length  $n$ , let  $i$  be the length of the **first piece** we cut off. The possible values for  $i$  are –

$$1 \leq i \leq n$$

If we do not divide piece  $i$  further, the maximum profit for the remaining length is  $r_{n-i}$ . Thus, the profit for this cut is –

$$p_i + r_{n-i}$$

To find the optimal profit  $r_n$ , we take the **maximum** over all possible first cuts –

$$r_n = \max_{1 \leq i \leq n} \text{_____}$$

## RECURRENCE FOR MAXIMUM PROFIT

For a rod of length  $n$ , let  $i$  be the length of the **first piece** we cut off. The possible values for  $i$  are —

$$1 \leq i \leq n$$

If we do not divide piece  $i$  further, the maximum profit for the remaining length is  $r_{n-i}$ . Thus, the profit for this cut is —

$$p_i + r_{n-i}$$

To find the optimal profit  $r_n$ , we take the **maximum** over all possible first cuts —

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$



Using this formula, what is  $r_4$  for our previous example?

## A NAIVE ALGORITHM

A naive algorithm for computing  $r_n$  is the following —

```
1: Cut-Rod( $p, n$ )
2: if  $n == 0$  then
3:   return 0
4:  $q \leftarrow -\infty$ 
5: for  $i = 1$  to  $n$  do
6:    $q \leftarrow \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
7: return  $q$ 
```

The naive algorithm is inefficient as many  $r_n$ 's must be computed more than once.

Let's consider what is involved in computing  $r_4$ .

## THE TIME COMPLEXITY

---

Let  $T(n)$  be the time complexity of the naive algorithm. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \sum_{i=0}^{n-1} T(i) & \text{if } n > 0. \end{cases}$$

## THE TIME COMPLEXITY

---

Let  $T(n)$  be the time complexity of the naive algorithm. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \sum_{i=0}^{n-1} T(i) & \text{if } n > 0. \end{cases}$$

 What are the values of  $T(0), T(1), \dots, T(3)$ ?

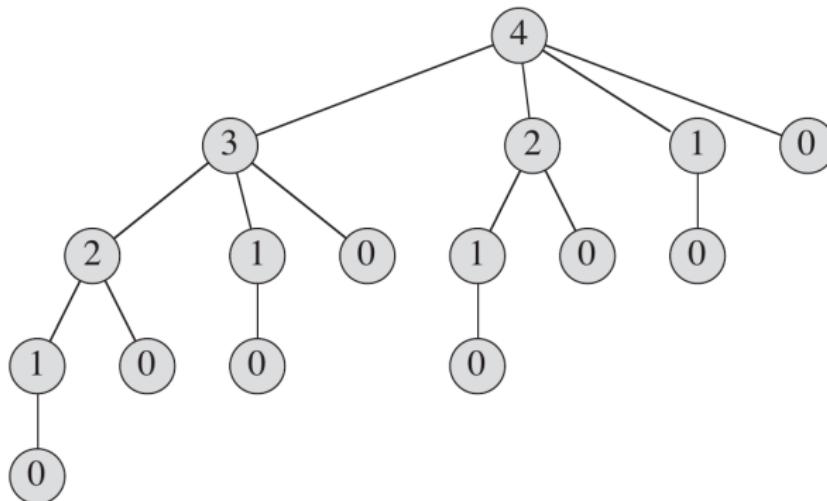
 Can you guess a closed-form expression for  $T(n)$ ?

 Prove your guess is correct.

# DYNAMIC PROGRAMMING

There are two ways in dynamic programming for improving a naive recursive algorithm —

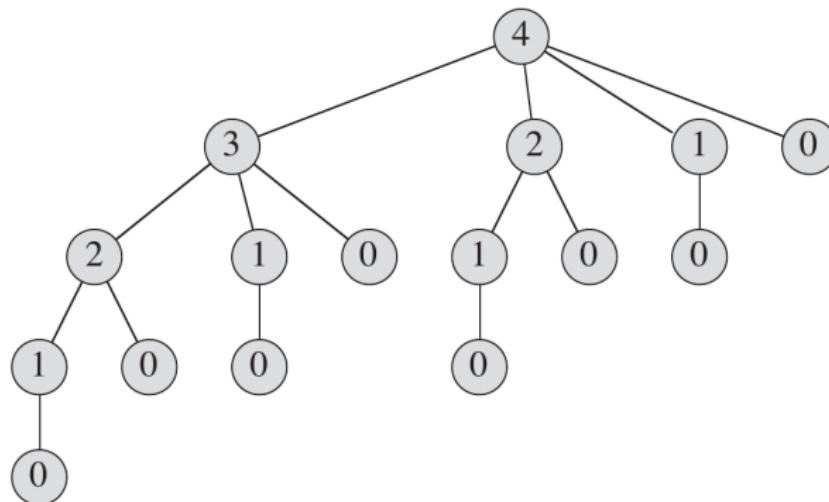
- Top-down with **memoization** — This method uses a recursive approach with a modification to save results of subproblems.



## DYNAMIC PROGRAMMING

There are two ways in dynamic programming for improving a naive recursive algorithm —

- ↑ Bottom-up — This approach sorts subproblems by their natural size. It ensures that smaller subproblems are solved first.



## ROD CUTTING WITH MEMOIZATION

```
1: Mem-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  be a new array
3: for  $i = 0$  to  $n$  do
4:    $r[i] = -\infty$ 
5: return
   MEM-CUT-ROD-AUX( $p, n, r$ )
```



This procedure just does a bit  
of initialization.

## ROD CUTTING WITH MEMOIZATION

```
1: Mem-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  be a new array
3: for  $i = 0$  to  $n$  do
4:    $r[i] = -\infty$ 
5: return
MEM-CUT-ROD-AUX( $p, n, r$ )
```

💡 This procedure just does a bit of initialization.

```
1: Mem-Cut-Rod-Aux( $p, n, r$ )
2: if  $r[n] \geq 0$  then
3:   return  $r[n]$ 
4: if  $n == 0$  then
5:    $q = 0$ 
6: else
7:    $q = -\infty$ 
8: for  $i = 1$  to  $n$  do
9:
 $q = \max(q, p[i] + \text{MEM-CUT-ROD-AUX}(p, n - i, r))$ 
10:  $r[n] = q$ 
11: return  $q$ 
```

## TIME COMPLEXITY

The most time-consuming part of the algorithm is the following loop –

```
1: Mem-Cut-Rod-Aux( $p, n, r$ )
2: ...
3:  $q = -\infty$ 
4: for  $i = 1$  to  $n$  do
5:    $q = \max(q, p[i] + \text{MEM-CUT-ROD-AUX}(p, n - i, r))$ 
6: ...
```

But for each  $j \in \{1, \dots, n\}$ , this loop is run only once.

So the running time of **Mem-Cut-Rod** is

$$\sum_{j=1}^n \Theta(j) = \underline{\hspace{2cm}}$$

## ROD CUTTING WITH BOTTOM-UP

---

```
1: Bottom-Up-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  be a new array
3:  $r[0] \leftarrow 0$ 
4: for  $j = 1$  to  $n$  do
5:    $q \leftarrow -\infty$ 
6:   for  $i = 1$  to  $j$  do
7:      $q \leftarrow \max(q, p[i] + r[j - i])$ 
8:    $r[j] \leftarrow q$ 
9: return  $r[n]$ 
```

The most time-consuming part of the algorithm is the following double loop –

```
1: Bottom-Up-Cut-Rod( $p, n$ )
2: ...
3: for  $j = 1$  to  $n$  do
4:    $q \leftarrow -\infty$ 
5:   for  $i = 1$  to  $j$  do
6:      $q \leftarrow \max(q, p[i] + r[j - i])$ 
7:    $r[j] \leftarrow q$ 
8: ...
```

## SUBPROBLEM GRAPHS

When thinking about dynamic programming, we often use graphs to represent the dependency of subproblems.

Typically, the time complexity of solving a subproblem is proportional to its out-degree.

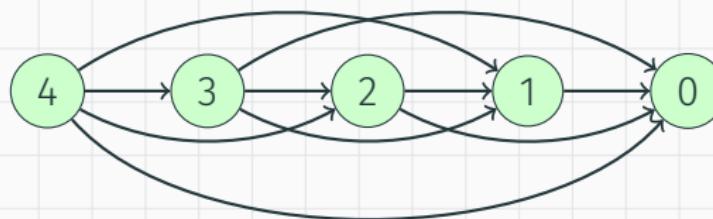


Figure 5: ❤ Example –  $r_4$  depends on  $r_1, r_2, r_3$ , etc.

So the total time complexity is proportional to the number of edges.

🎂 How many are there in the above graph?

## RECONSTRUCTING A SOLUTION

---

The cut positions can be recorded in an auxiliary array  $s[0..n]$ .

```
1: Extended-Bottom-Up-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  and  $s[0..n]$  be new arrays
3:  $r[0] \leftarrow 0$ 
4: for  $j = 1$  to  $n$  do
5:    $q \leftarrow -\infty$ 
6:   for  $i = 1$  to  $j$  do
7:     if  $q < p[i] + r[j - i]$  then
8:        $q \leftarrow p[i] + r[j - i]$ 
9:        $s[j] \leftarrow i$ 
10:     $r[j] \leftarrow q$ 
11: return ( $r, s$ )
```

## RECONSTRUCTING A SOLUTION

---

The cut positions can be recorded in an auxiliary array  $s[0..n]$ .

Then the solution can be reconstructed as follows —

```
1: Print-Cut-Rod-Solution( $p, n$ )
2:  $(r, s) \leftarrow$ 
   Extended-Bottom-Up-Cut-Rod( $p, n$ )
3: while  $n > 0$  do
4:   print  $s[n]$ 
5:    $n \leftarrow n - s[n]$ 
```

Given the following price table –

$p_1$	$p_2$	$p_3$	$p_4$
1	5	8	9

🎂 What is  $r_4$ ?

 EXERCISE

How should we modify **Bottom-Up-Cut-Rod**, if each piece must have a minimum length  $l$ ?

```
1: Bottom-Up-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  be a new array
3:  $r[0] \leftarrow 0$ 
4: for  $j = 1$  to  $n$  do
5:    $q \leftarrow -\infty$ 
6:   for  $i = 1$  to  $j$  do
7:      $q \leftarrow \max(q, p[i] + r[j - i])$ 
8:    $r[j] \leftarrow q$ 
9: return  $r[n]$ 
```

```
1: Bottom-Up-Cut-Rod-Mod( $p, n$ )
2: let  $r[0..n]$  be a new array
3: for  $j = \underline{\hspace{2cm}}$  to  $l$  do
4:    $\underline{\hspace{2cm}}$ 
5: for  $j = \underline{\hspace{2cm}}$  to  $n$  do
6:    $q \leftarrow -\infty$ 
7:   for  $i = \underline{\hspace{2cm}}$  to  $j$  do
8:      $q \leftarrow \max(q, p[i] + r[j - i])$ 
9:    $r[j] \leftarrow q$ 
10: return  $r[n]$ 
```

 EXERCISE

How should we modify **Bottom-Up-Cut-Rod**, if each cut has a cost  $c$ ?

```
1: Bottom-Up-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  be a new array
3:  $r[0] \leftarrow 0$ 
4: for  $j = 1$  to  $n$  do
5:    $q \leftarrow -\infty$ 
6:   for  $i = 1$  to  $j$  do
7:      $q \leftarrow \max(q, p[i] + r[j - i])$ 
8:    $r[j] \leftarrow q$ 
9: return  $r[n]$ 
```

```
1: Bottom-Up-Cut-Rod-Mod( $p, n$ )
2: let  $r[0..n]$  be a new array
3:  $r[0] \leftarrow 0$ 
4: for  $j = 1$  to  $n$  do
5:    $q \leftarrow -\infty$ 
6:   for  $i = 1$  to  $j$  do
7:      $q \leftarrow$  _____
8:    $r[j] \leftarrow q$ 
9: return  $r[n]$ 
```

 EXERCISE

How should we modify **Bottom-Up-Cut-Rod**, if the price of a piece of the rod depends on the total length of that rod? In other words, what if  $p[i]$  is replaced by  $p[n, i]$ ?

```
1: Bottom-Up-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  be a new array
3:  $r[0] \leftarrow 0$ 
4: for  $j = 1$  to  $n$  do
5:    $q \leftarrow -\infty$ 
6:   for  $i = 1$  to  $j$  do
7:      $q \leftarrow \max(q, p[i] + r[j - i])$ 
8:    $r[j] \leftarrow q$ 
9: return  $r[n]$ 
```

```
1: Bottom-Up-Cut-Rod-Mod( $p, n$ )
2: let  $r[0..n]$  be a new array
3: for  $j = 1$  to  $n$  do
4:    $q \leftarrow -\infty$ 
5:   for  $i = 1$  to  $j$  do
6:      $q \leftarrow$  _____
7:    $r[j] \leftarrow q$ 
8: return  $r[n]$ 
```

 EXERCISE

How should we modify **Bottom-Up-Cut-Rod**, if the number of pieces is limited to at most  $m$ ?

```
1: Bottom-Up-Cut-Rod( $p, n$ )
2: let  $r[0..n]$  be a new array
3:  $r[0] \leftarrow 0$ 
4: for  $j = 1$  to  $n$  do
5:    $q \leftarrow -\infty$ 
6:   for  $i = 1$  to  $j$  do
7:      $q \leftarrow \max(q, p[i] + r[j - i])$ 
8:    $r[j] \leftarrow q$ 
9: return  $r[n]$ 
```

 Is it enough to compute only  $r[0..n]$ ?

WHAT ARE YOUR MAIN TAKEAWAYS TODAY? ANY QUESTIONS?

