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

Assignments[1]

Practice makes perfect!

Introduction to Algorithms (ITA) 📖
Required Readings:

- Section 15.1.

✏️ Required Exercises:

- Exercises 15.1 — 1–5.

[1] 💣 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

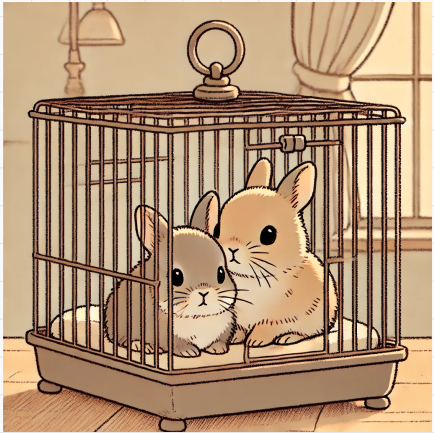
Suppose that you want to have some 🐰 as pets—

- Month 0 – You have no pets.

Fibonacci's Puzzle

Suppose that you want to have some 🐰 as pets—

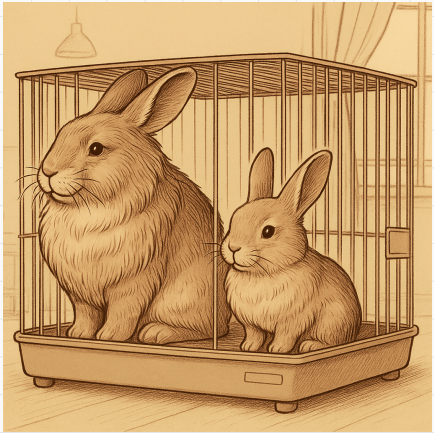
- Month 0 – You have no pets.
- Month 1—You get a pair of baby 🐰.



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.

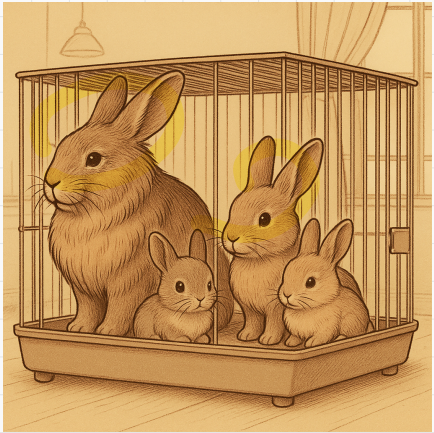


Fibonacci's Puzzle

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 🐰.

Month 5: 3 pairs



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

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

$13 = 5 + 8$, $21 = 8 + 13$,

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

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

is called the Fibonacci sequence.

🤔 Can we find a recurrence relation for F_n?

By observing the initial few numbers, we can see that

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

🤔 How to prove this?

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

- F_{n-1} who are alive in month $n-1$
- F_{n-2} who are new borns.

This leads to the recursion —

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

All bunnys alive in month $n-2$.

$F_6 = F_5 + F_4$

$8 = 5 + 3$

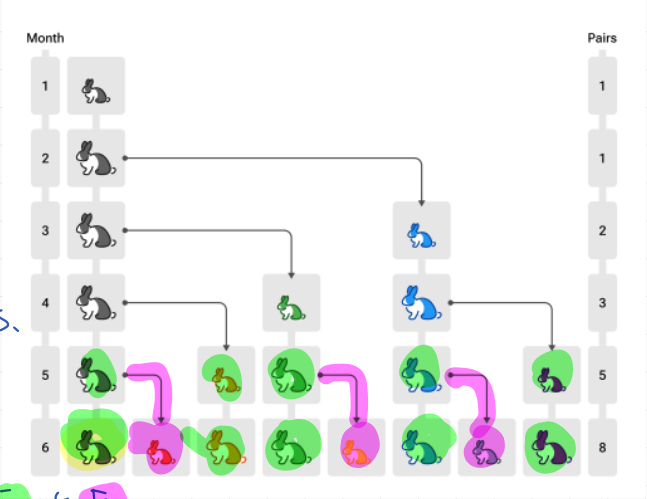


Figure 1: The recurrence relation for the Fibonacci sequence

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 often appear in the spiral pattern of particular plants.

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

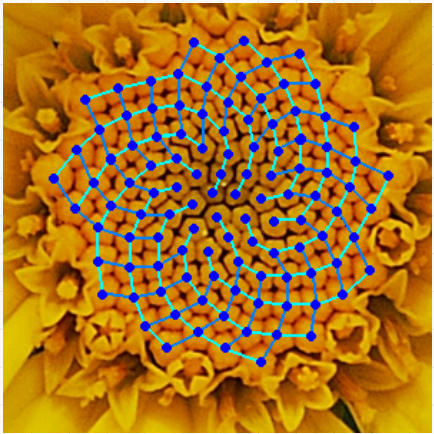


Figure 2: Yellow chamomile (洋甘菊) head showing the arrangement in 21 (blue) and 13 (cyan) spirals. Source — Wikipedia

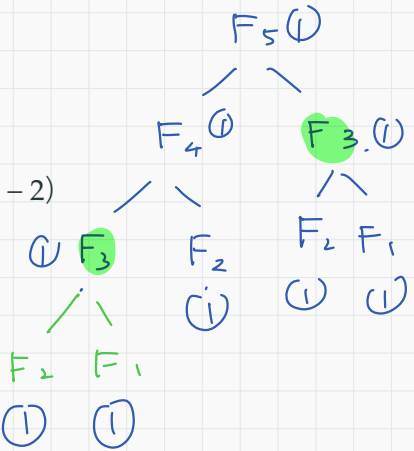
The Naive Way

```
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)
```

$F_0 = 0$

$F_1 = 1$

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



- Let's try to compute F_5 with the naive algorithm.
- What is the time complexity of this algorithm?

Exponential.

A Better Way

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

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

$F_0 = 0$
$F_1 = 1$
n times.
$F_i = F_{i-1} + F_{i-2}$.
F_n.

An algorithm in this style is called dynamic programming.

🍰 What is the time complexity of this algorithm?

$O(n)$.

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.

Find the best.

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.

↑ ↑ ↑

optimal solution to sub proble.

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. or use memoization (remember)
- 📋 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 4️⃣ 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

Richard E. Bellman

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?

$r_1 = p_1 = 1$.

$r_3 = 8$ $p_1 + p_1 + p_1 = 3$ $p_2 + p_1 = 6$ $p_3 = 8$.

$r_2 = 5$. $= 5$ p_2 $p_1 + p_1 = 2$.

Len 2. 1 + 8 5 + 5

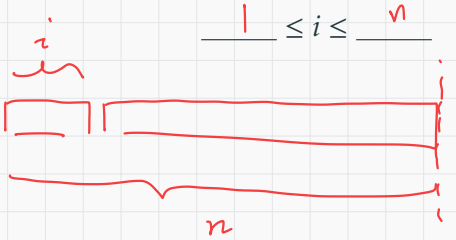
$r_4 = \max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4 + r_0)$.

$= p_2 + r_2 = 10$. $8 + 1$ $9 + 0$!!

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 —

$$1 \leq i \leq n$$



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 —

$$1 \leq i \leq n$$

of len

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 —

$p_i + r_{n-i}$

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 —

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

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 ← −∞
5: for i = 1 to n do
6:     q ← max(q, p[i] + CUT-ROD(p, n − i))
7: return q
```

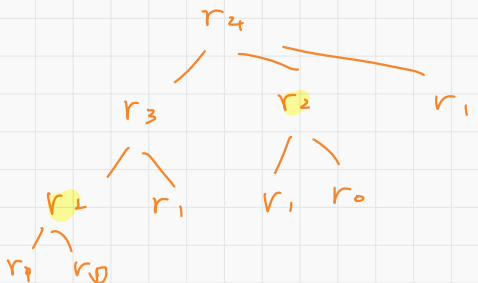
$r_0 = 0$

$r_n = \max_{1 \le i \le n} p_i + r_{n-i}$

$p_i + r_{n-i}$

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 $T(0), T(1), \ldots, T(3)$?

$T(0) = 1$, $T(1) = 1 + T(0) = 2$, $T(3) = 1 + T(0) + T(1) + T(2)$

$T(2) = 1 + T(1) + T(0) = 1 + 2 + 1 = 4$.

$= 1 + 1 + 2 + 4 = 8$.

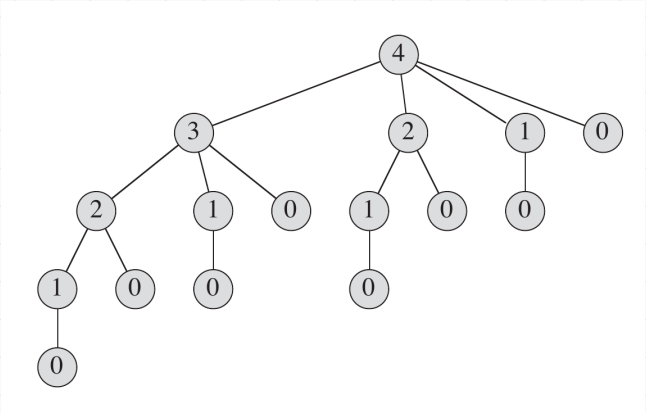
- Can you guess a closed-form expression for $T(n)$?

$T(n) = 2^n$.

- Prove your guess is correct. — Induction / substitution.

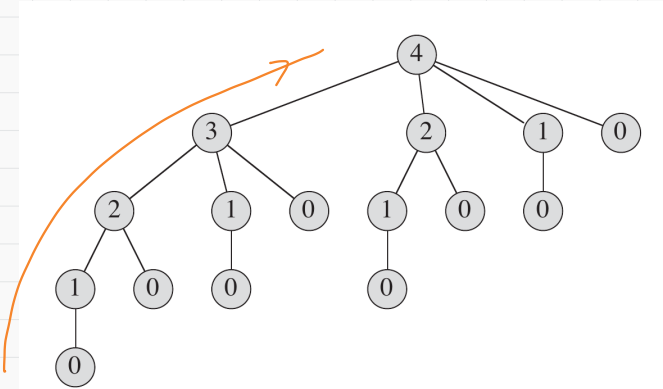
There are two ways in dynamic programming for improving a naive recursive algorithm —

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



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: $\quad r[i] = -\infty$
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] = −∞
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] ≥ 0 then
3:     return r[n]
4: if n == 0 then
5:     q = 0
6: else
7:     q = −∞
8:     for i = 1 to n do
9:
       q = max(q, p[i] + Mem-Cut-Rod-Aux(p, n − i, r))
10: r[n] = q
11: return q
```

r_n has already been computed.

$r_0 = 0$.

Best profit we have seen so far.

$r_n = \max_{1 \le i \le n} p_i + r_{n-i}$

r_n

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: $\quad q = \max(q, p[i] + \text{MEM-CUT-ROD-AUX}(p, n - i, r))$
6: ...

n times.

r_j takes $\Theta(j)$.

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

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

$$\sum_{j=1}^{n} \Theta(j) = \Theta(n^2).$$

Rod Cutting with Bottom-Up

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

$r_0 = 0$.

← compute r_j.

r_j

$r_j = \max_{1 \le i \le j} p_i + r_{j-i}$

already computed.

Time Complexity

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 ← −∞
5:     for i = 1 to j do                ⇐ j times
6:         q ← max(q, p[i] + r[j − i])
7:     r[j] ← q
8: ...
```

$\Theta(j)$

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

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.

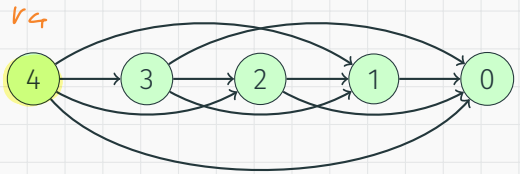


Figure 5: 📌 Example — r_4 depends on r_1, r_2, r_3, etc.

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] ← 0
4:  for j = 1 to n do
5:      q ← −∞
6:      for i = 1 to j do
7:          if q < p[i] + r[j − i] then
8:              q ← p[i] + r[j − i]
9:              s[j] ← i
10:     r[j] ← 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) ←
   Extended-Bottom-Up-Cut-Rod(p, n)
3: while n > 0 do
4:     print s[n]
5:     n ← n − s[n]
```

💬 Try It Out!

Given the following price table —

p_1	p_2	p_3	p_4
1	4	8	9

🍰 What is r_4?

$r_1 =$

$r_2 =$

$r_3 =$

$r_4 =$

FIGNAO

Exercise

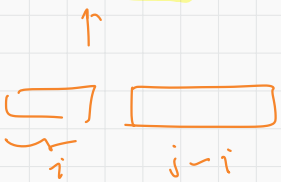
How should we modify Bottom-Up-Cut-Rod, if each piece must have a minimum length l?

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

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] ← 0
4: for j = 1 to n do
5:     q ← −∞
6:     for i = 1 to j do
7:         q ← max(q, p[i] + r[j − i])
8:     r[j] ← 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] ← 0
4: for j = 1 to n do
5:     q ← −∞
6:     for i = 1 to j do
7:         q ← max(q, p[i] + r[j − i] − c)
8:     r[j] ← 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] ← 0
4: for j = 1 to n do
5:     q ← −∞
6:     for i = 1 to j do
7:         q ← max(q, p[i] + r[j − i])
8:     r[j] ← 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 ← −∞
5:     for i = 1 to j do
6:         q ← max(q, p[j, i] + r[j − i])
7:     r[j] ← 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] ← 0
4: for j = 1 to n do
5:     q ← −∞
6:     for i = 1 to j do
7:         q ← max(q, p[i] + r[j − i])
8:     r[j] ← q
9: return r[n]
```

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

What are your main takeaways today? Any questions?