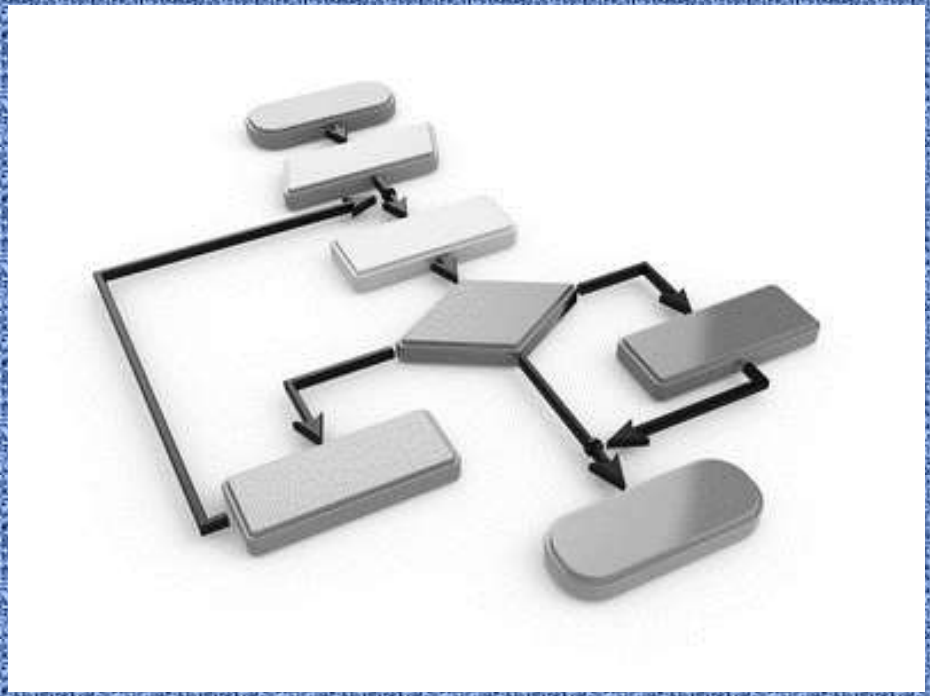


Dynamic Programming

Rod-cutting – Problem

Dynamic Programming

Dynamic Programming (DP) is an algorithm design technique for ***optimization problems***.

Like divide and conquer (D&C), DP solves problems by combining solutions to sub-problems.

Unlike D&C, sub-problems are not **independent**.

Sub-problems may share sub-sub-problems

The **solutions** to sub-problems in **DP overlap** and can **share** sub-sub-problems.

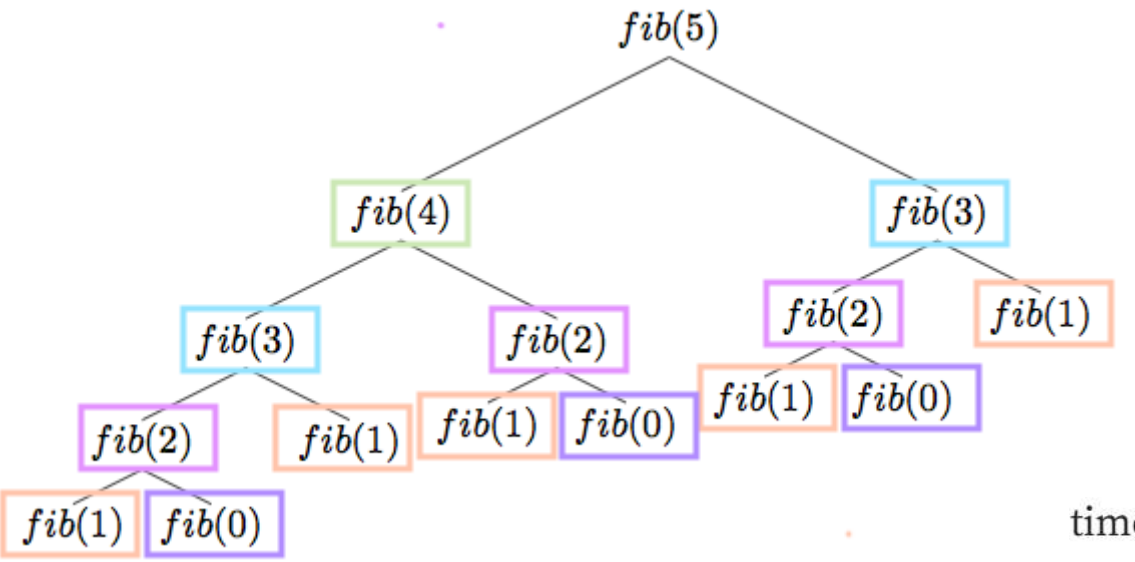
This is different from **Divide and Conquer (D&C)**, where the sub-problems are **independent** and do **not overlap**.

Fibonacci Series – Overlapping Subproblems

Subproblems are smaller versions of the original problem.

Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times.

Take the example of the Fibonacci numbers; to find the fib(5), we need to break it down into the following sub-problems:



The recursive solution is more elegant:

```
function fib(n){
  if (n < 0) return undefined;
  if (n < 2) return n;
  return fib(n-1) + fib(n-2)
}
```

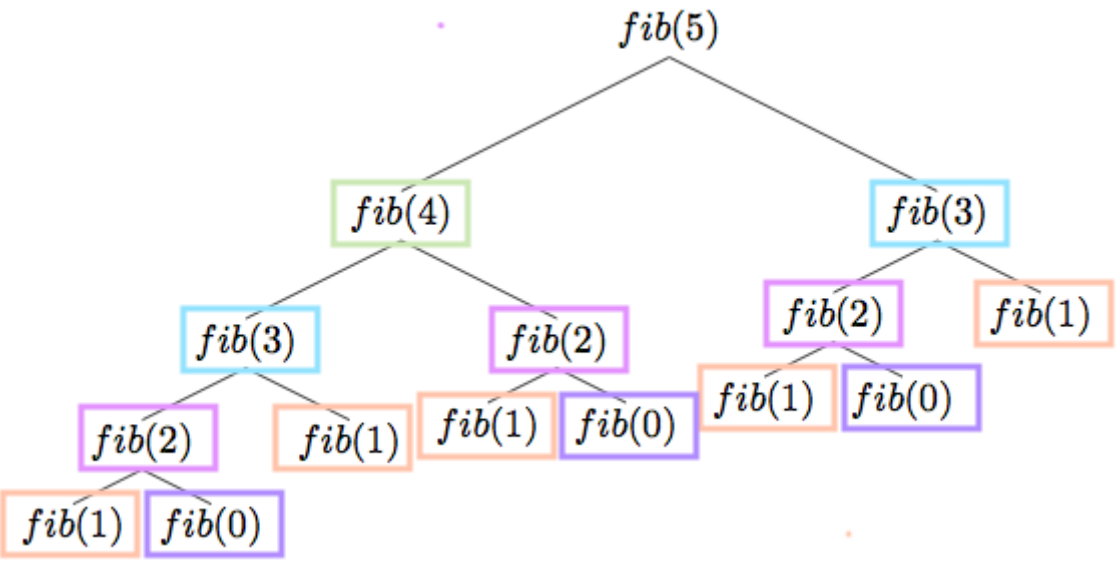
time complexity is exponential, or `O(2^n)`

Fibonacci Series – **Overlapping** Subproblems

Subproblems are smaller versions of the original problem.

Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times.

Take the example of the Fibonacci numbers; to find the fib(5), we need to break it down into the following sub-problems:



Using the **color-coded** image above, we can see just how many overlapping subproblems there are. To solve for **fib(5)**, we need to calculate **fib(3)** twice, **fib (2)** three times etc. This is **duplication** that we would like to **eliminate**.

Fibonacci Series – Optimal Substructure (Bottom-Up Approach)

Optimal substructure means an optimal solution can be constructed from optimal solutions of its subproblems.

Fibonacci of n is defined as follows:

```
fib(n) = fib(n-1) + fib(n-2)
```

The optimal solution for n depends on the optimal solution of `(n-1)` and `(n-2)`

There are two ways to solve the Fibonacci problem using dynamic programming.

Dynamic Programming – 2 ways to solve

Dynamic Programming – 2 ways to solve

1. Memoization

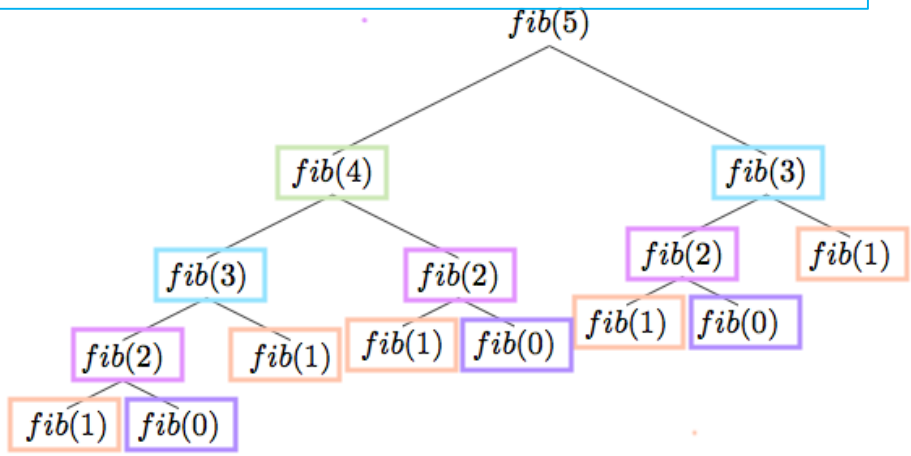
Memoization stores the result of expensive function calls (in arrays or objects) and returns the stored results whenever the same inputs occur again. In this way we can remember any values we have already calculated and access them instead of repeating the same calculation.

This is a **top-down** approach, meaning we start with what we are trying to find. We start from `fib(5)` and work our way down, filling in the gaps and adding them all together. (To to find `fib(5)` we calculate `fib(4)` and `fib(3)` etc.)

Dynamic Programming – 2 ways to solve

1. Memoization

- In the recursive approach, there is a lot of redundant calculation that are calculating again and again, So we can **store** the **results** of previously computed Fibonacci numbers in a **memo table** to avoid redundant calculations.
- This will make sure that each Fibonacci number is only computed once, this will reduce the exponential time complexity of the naive approach **O(2^n)** into a more efficient **O(n)** time complexity.



Dynamic Programming – 2 ways to solve

1. Memoization

```
# Function to calculate the nth Fibonacci number using memoization
def nth_fibonacci_util(n, memo):

    # Base case: if n is 0 or 1, return n
    if n <= 1:
        return n

    # Check if the result is already in the memo table
    if memo[n] != -1:
        return memo[n]

    # Recursive case: calculate Fibonacci number
    # and store it in memo
    memo[n] = nth_fibonacci_util(n - 1, memo) + nth_fibonacci_util(n - 2, memo)
    return memo[n]
```

Dynamic Programming – 2 ways to solve

1. Memoization

```
# Wrapper function that handles both initialization
# and Fibonacci calculation
def nth_fibonacci(n):

    # Create a memoization table and initialize with -1
    memo = [-1] * (n + 1)

    # Call the utility function
    return nth_fibonacci_util(n, memo)

if __name__ == "__main__":
    n = 5
    result = nth_fibonacci(n)
    print(result)
```

Dynamic Programming – 2 ways to solve

2. Tabulation

This approach uses dynamic programming to solve the Fibonacci problem by **storing previously calculated Fibonacci numbers**, **avoiding** the repeated calculations of the recursive approach. Instead of breaking down the problem recursively, it **iteratively** builds up the solution by calculating Fibonacci numbers from the bottom up.

This is a **bottom-up** approach. We start from the bottom, finding `fib(0)` and `fib(1)`, add them together to get `fib(2)` and so on until we reach `fib(5)`.

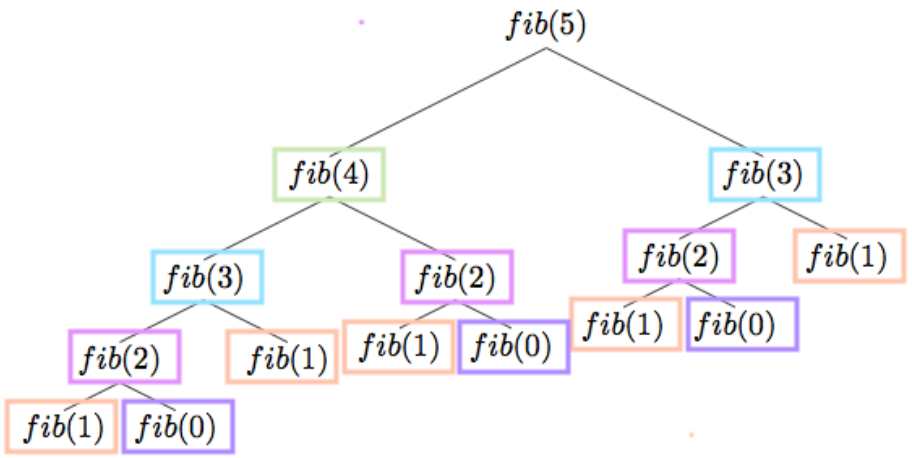
2. Tabulation

```python
def nth_fibonacci(n):

    # Handle the edge cases
    if n <= 1:
        return n

    # Create a list to store Fibonacci numbers
    dp = [0] * (n + 1)

    # Initialize the first two Fibonacci numbers
    dp[0] = 0
    dp[1] = 1

    # Fill the list iteratively
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    # Return the nth Fibonacci number
    return dp[n]

n = 5
result = nth_fibonacci(n)
print(result)
```

Fibonacci Series –
Optimal Substructure

Output

```
5
```



Problem: Fibonacci Sequence

The Fibonacci sequence is defined as:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

Our goal is to calculate $F(n)$.

Dynamic Programming Approach (with a Table)

To avoid recalculating the same values, we use a **table (array)** to store the results of sub-problems and reuse them.

Steps:

1. Start from the smallest sub-problems ($F(0)$ and $F(1)$) and calculate larger Fibonacci numbers step by step.
2. Save the result of each Fibonacci number in a table.
3. Use the table to get the results of previously solved sub-problems.

Problem: Fibonacci Sequence

The Fibonacci sequence is defined as:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

Our goal is to calculate $F(n)$.

Example:

Let's calculate $F(5)$ using DP:

n	Table (F)	How it's calculated
0	[0]	$F(0) = 0$
1	[0, 1]	$F(1) = 1$
2	[0, 1, 1]	$F(2) = F(1) + F(0) = 1 + 0$
3	[0, 1, 1, 2]	$F(3) = F(2) + F(1) = 1 + 1$
4	[0, 1, 1, 2, 3]	$F(4) = F(3) + F(2) = 2 + 1$
5	[0, 1, 1, 2, 3, 5]	$F(5) = F(4) + F(3) = 3 + 2$

Problem: Fibonacci Sequence

The Fibonacci sequence is defined as:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

Our goal is to calculate $F(n)$.

Key Points:

1. **Time is saved**: Each Fibonacci number is calculated **only once** and stored in the table. The time complexity is reduced to $O(n)$, compared to $O(2^n)$ in naive recursion.
2. **Space is used**: The table takes $O(n)$ space to store results.

Problem: Fibonacci Sequence

The Fibonacci sequence is defined as:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

Our goal is to calculate $F(n)$.

Key Points:

1. **Time is saved**: Each Fibonacci number is calculated **only once** and stored in the table. The time complexity is reduced to $O(n)$, compared to $O(2^n)$ in naive recursion.
2. **Space is used**: The table takes $O(n)$ space to store results.

General Rule for Dynamic Programming:

1. **Break the problem** into smaller sub-problems.
2. Solve the smaller problems **once** and store their solutions in a table.
3. Use the stored solutions to solve larger problems efficiently.

Dynamic Programming – 2 ways to solve

The time complexity of both the memoization and tabulation solutions are `O(n)` — time grows linearly with the size of n, because we are calculating `fib(4)`, `fib(3)`, etc each one time.

Note: While both have the linear time complexity, since memoization uses recursion, if you try to find the Fibonacci number for a large `n`, you will get a stack overflow (maximum call stack size exceeded).

Tabulation, on the other hand, doesn't take as much space, and will not cause a stack overflow.

Top-Down Approach (Memoization) vs Bottom-Up Approach (Tabulation)

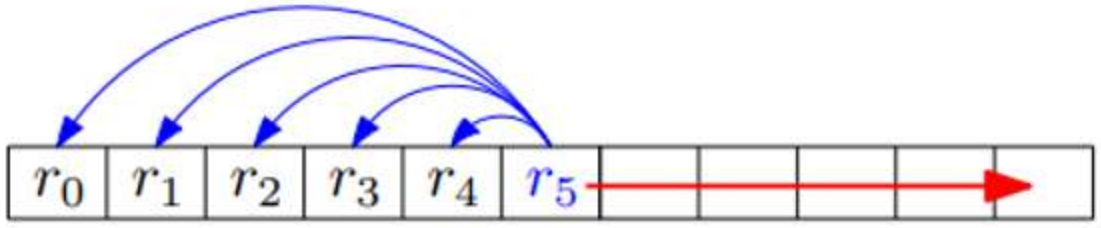
Top-Down Approach (Memoization)

- **Method**: This approach starts with the main problem and breaks it down into smaller subproblems as needed.
- **Subproblem Solving**: It solves subproblems on demand, meaning only when they are needed for solving the larger problem.
- **Memoization**: As each subproblem is solved, its result is stored (memoized) to avoid redundant calculations in future calls.
- **Ordering**: There is no need to predetermine the order of solving subproblems because they are solved recursively as needed.

Top-Down Approach (Memoization) vs Bottom-Up Approach (Tabulation)

Bottom-Up Approach (Tabulation)

- **Method**: This approach starts by solving the smallest subproblems first and uses their solutions to build up the solution to the main problem.
- **Subproblem Solving**: It requires solving all possible subproblems in a specific order, usually starting from the simplest and moving towards the more complex ones.
- **Tabulation**: It uses a table (array) to store the solutions to all subproblems in an iterative manner.
- **Ordering**: The key challenge is to determine the correct order in which to solve the subproblems so that by the time a subproblem is needed, all the subproblems it depends on have already been solved.



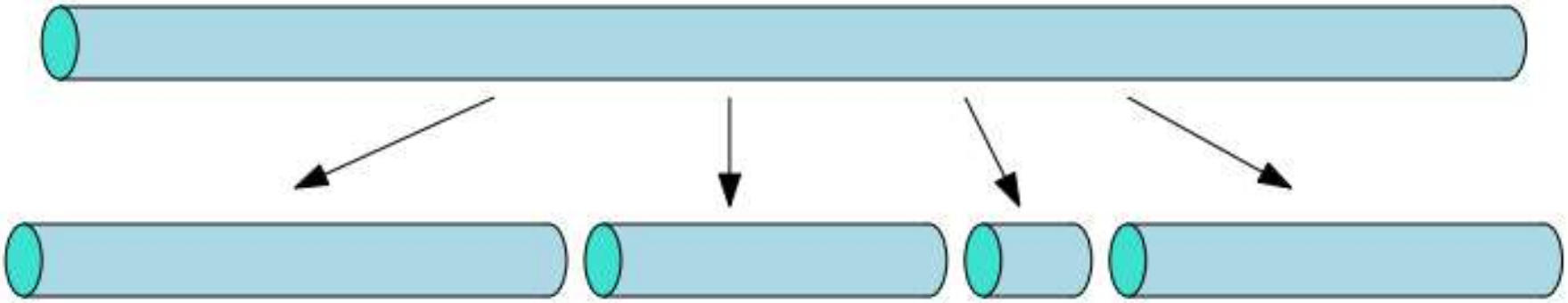
Designing a DP Algorithm

1. **Characterize** the structure of an optimal solution.
2. **Recursively** define the **value** of an optimal solution.
3. Compute the value of an optimal solution in a **bottom up** fashion.
4. Construct an optimal solution from computed information.

Rod cutting Problem

Problem Statement

- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells.



Problem Statement

- The price of a rod depends on its length, which is supposed to be an integer.

Example

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- The goal is to maximize the total price of the rods obtained from a rod of length n, given the values p_i for $i = 1, 2, \ldots n$.

Example

$n = 8$, using prices p_i from the table above.

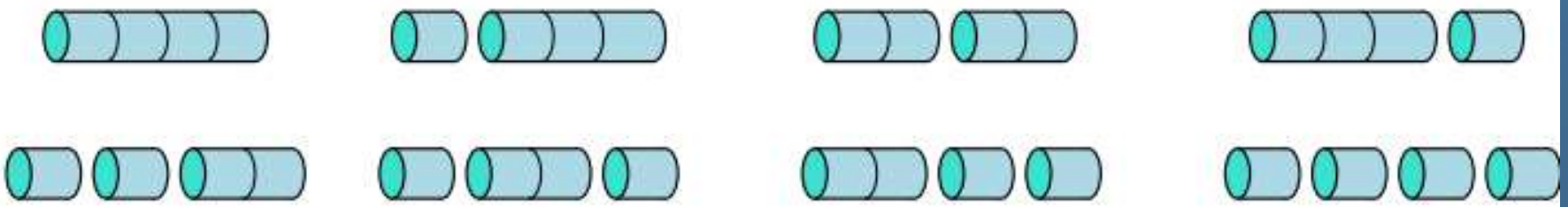
- If we cut into two rods of length 4, the total price is 9+9=18.
- If we cut into two rods of length 3 and one of length 2, the total price is 8+8+5=21.
- So the second solution is better.

Difficulty

- How many different ways are there of cutting a rod of length n?
  - 2^{n-1}, because we can cut at $n-1$ different locations.

Example

If $n = 4$, there are $2^3 = 8$ different ways:



- So brute-force search runs in $\Omega(2^n)$ time.
- This is *exponential* time. We need a faster algorithm.

Structure of the Solution

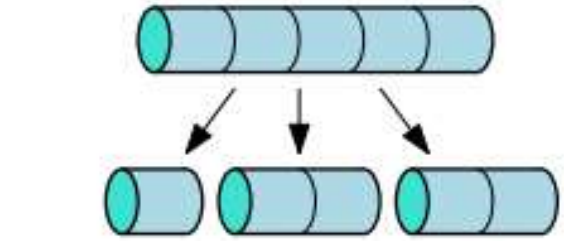
- Suppose that the prices p_i are fixed, for $i = 1, \ldots, n$. Let r_i denote the value of the optimal solution of the subproblem where we cut a rod of size $i \leqslant n$.

Example

Using the price table below,

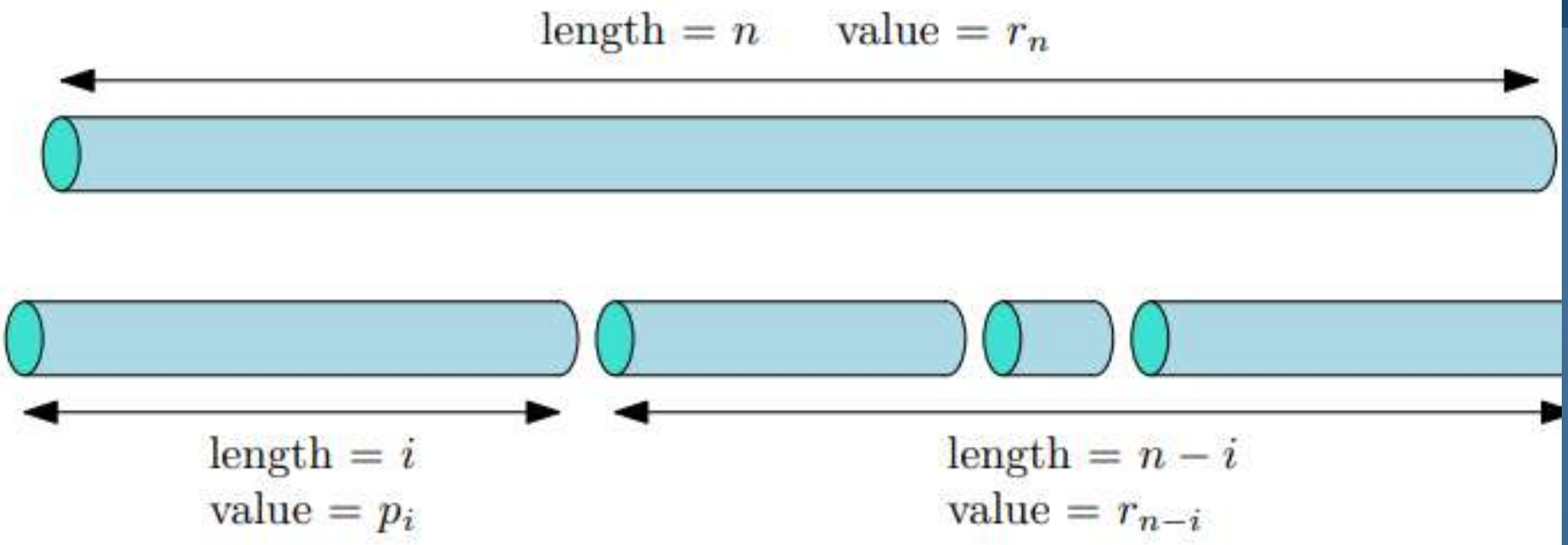
length i	1	2	3	4	5	6	7	8	9	10
price p_i	2	5	6	7	8	9	11	11	13	15

$r_5 = 2 + 5 + 5 = 12$, where we cut into 3 pieces of lengths 1,2 and 2.



$$r_5 = 2 + 5 + 5 = 12$$

Structure of the Solution



Observation

An optimal solution consists of a piece of length i, for some i, followed by a rod of length $n - i$ cut optimally.

Recurrence Relation

- So we either have $r_n = p_i + r_{n-i}$ for some i, or $r_n = p_n$.
- Then if we write $r_0 = 0$, it means:

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

$r_1 = p_1$

$r_2 = \max(p_2, p_1+r_1)$

$r_3 = \max(p_3, p_1+r_2 , p_2+r_1)$

Structure of the Solution

Proof of Equation (1).

We assume that the rod of length n is cut optimally.

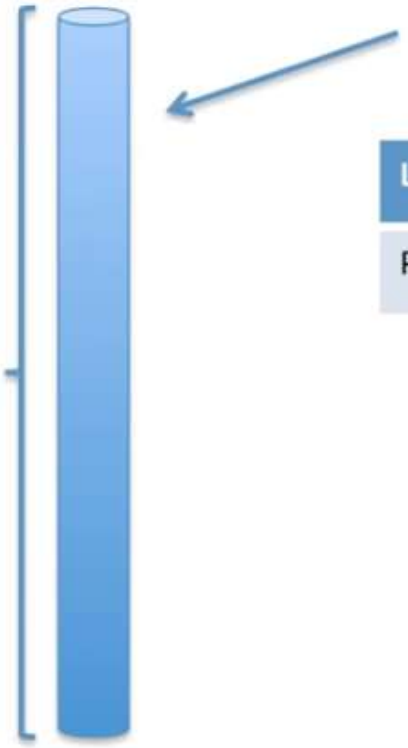
- If there is at least one cut, then the first cut occurs after a length $1 \leqslant i < n$. Then the remaining part should be cut optimally, so its value is r_{n-i}, and the total value of the pieces is $p_i + r_{n-i}$.
- Otherwise, it means that the rod is not cut, and thus the optimal value is $r_n = p_n + r_0 = p_n$.

It follows that the value of the optimal solution satisfies

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

Problem Description: Given a rod of length n units, and the price of all pieces smaller than n, find the most profitable way of cutting the rod.

Rod Cutting DP



The Rod

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Rod Cutting DP

The Rod

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Possible Combinations:

$1

$1

$1

$1

Rod Cutting DP

The Rod

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Possible Combinations:

The remaining 3 ways to cut are just permutations of the left arrangements.

Rod Cutting DP

The Rod

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Possible Combinations:

The remaining 3 ways to cut are just permutations of the left arrangements.

Hence, this problem has the optimal substructure

Rod Cutting DP

Proof of Optimal Sub Structure

Let's say we had the optimal solution for cutting the rod $C_{i...j}$ where C_i is the first piece, and C_j is the last piece.

Rod Cutting DP

Proof of Optimal Sub Structure

Let's say we had the optimal solution for cutting the rod $C_{i...j}$ where C_i is the first piece, and C_j is the last piece.

If we take one of the cuts from this solution, somewhere in the middle, say k, and split it so we have two sub problems, $C_{i..k}$, and $C_{k+1..j}$ (Assuming our optimal is not just a single piece)

Rod Cutting DP

Proof of Optimal Sub Structure

Let's say we had the optimal solution for cutting the rod $C_{i...j}$ where C_i is the first piece, and C_j is the last piece.

If we take one of the cuts from this solution, somewhere in the middle, say k, and split it so we have two sub problems, $C_{i..k}$, and $C_{k+1..j}$ (Assuming our optimal is not just a single piece)

Rod Cutting DP

Proof of Optimal Sub Structure

Let's say we had the optimal solution for cutting the rod $C_{i...j}$ where C_i is the first piece, and C_j is the last piece.

If we take one of the cuts from this solution, somewhere in the middle, say k, and split it so we have two sub problems, $C_{i..k}$, and $C_{k+1..j}$ (Assuming our optimal is not just a single piece)

Let's assume we had a more optimal way of cutting $C_{i..k}$

We would swap the old $C_{i..k}$, and replace it with the more optimal $C_{i..k}$

Overall, the entire problem would now have an even more optimal solution!

But we already had stated that we had the optimal solution! This is a contradiction!

Rod Cutting DP

Proof of Optimal Sub Structure

Let's say we had the optimal solution for cutting the rod $C_{i...j}$ where C_i is the first piece, and C_j is the last piece.

If we take one of the cuts from this solution, somewhere in the middle, say k, and split it so we have two sub problems, $C_{i..k}$, and $C_{k+1..j}$ (Assuming our optimal is not just a single piece)

Let's assume we had a more optimal way of cutting $C_{i..k}$

We would swap the old $C_{i..k}$, and replace it with the more optimal $C_{i..k}$

Overall, the entire problem would now have an even more optimal solution!

But we already had stated that we had the optimal solution! This is a contradiction!

Therefore our original optimal solution is the optimal solution, and this problem exhibits optimal substructure.

Rod Cutting DP

Let's define C(i) as the price of the optimal cut of a rod up until length i

Let V_k be the price of a cut at length k

Rod Cutting DP

Let's define C(i) as the price of the optimal cut of a rod up until length i

Let V_k be the price of a cut at length k

How to develop a solution:

We define the smallest problems first, and store their solutions.

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Rod Cutting DP

Let's define C(i) as the price of the optimal cut of a rod up until length i

Let V_k be the price of a cut at length k

How to develop a solution:

We define the smallest problems first, and store their solutions.

We increase the rod length, and try all the cuts for that size of rod, taking the most profitable one.

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Rod Cutting DP

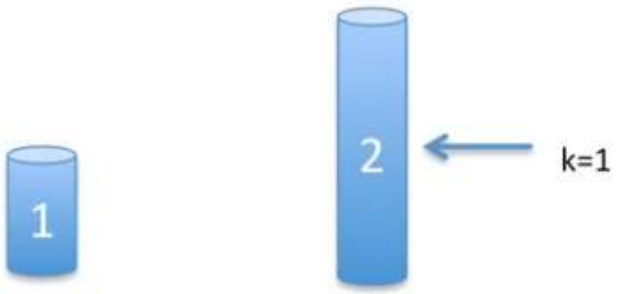
Let's define C(i) as the price of the optimal cut of a rod up until length i

Let V_k be the price of a cut at length k

How to develop a solution:

We define the smallest problems first, and store their solutions.

We increase the rod length, and try all the cuts for that size of rod, taking the most profitable one.



Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Rod Cutting DP

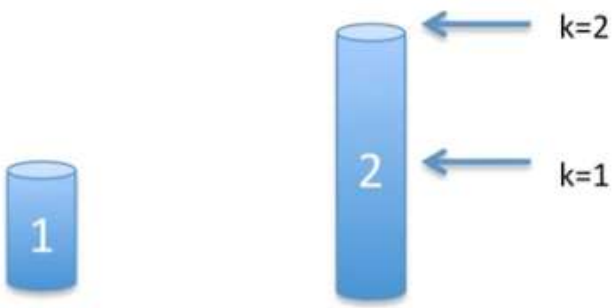
Let's define C(i) as the price of the optimal cut of a rod up until length i

Let V_k be the price of a cut at length k

How to develop a solution:

We define the smallest problems first, and store their solutions.

We increase the rod length, and try all the cuts for that size of rod, taking the most profitable one.



Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Rod Cutting DP

Let's define C(i) as the price of the optimal cut of a rod up until length i

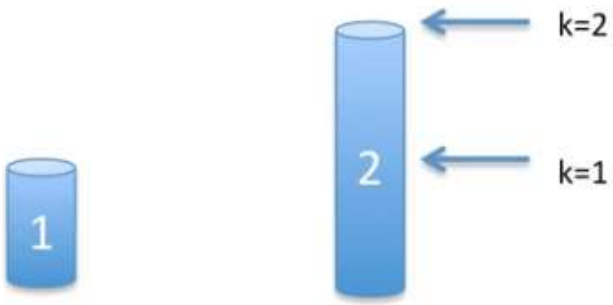
Let V_k be the price of a cut at length k

How to develop a solution:

We define the smallest problems first, and store their solutions.

We increase the rod length, and try all the cuts for that size of rod, taking the most profitable one.

We store the optimal solution for this sized piece, and build solutions to larger pieces from them in some sort of data structure.



Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Rod Cutting DP

Let's define C(i) as the price of the optimal cut of a rod up until length i

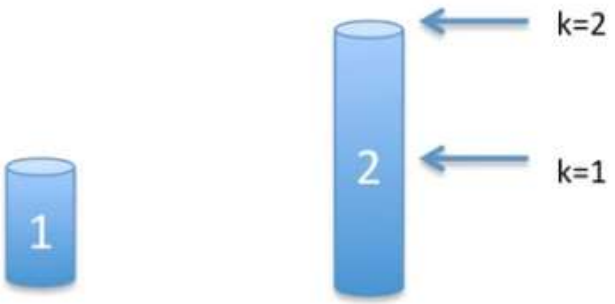
Let V_k be the price of a cut at length k

How to develop a solution:

We define the smallest problems first, and store their solutions.

We increase the rod length, and try all the cuts for that size of rod, taking the most profitable one.

We store the optimal solution for this sized piece, and build solutions to larger pieces from them in some sort of data structure.



$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Rod Cutting DP

Let's define C(i) as the price of the optimal cut of a rod up until length i

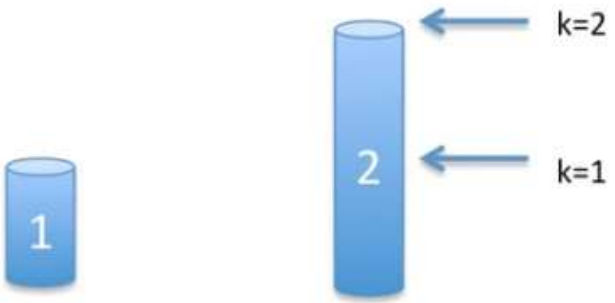
Let V_k be the price of a cut at length k

How to develop a solution:

We define the smallest problems first, and store their solutions.

We increase the rod length, and try all the cuts for that size of rod, taking the most profitable one.

We store the optimal solution for this sized piece, and build solutions to larger pieces from them in some sort of data structure.



$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Memoization

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt								

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Len (i)	1	2	3	4	5	6	7	8
Opt								

C(i)

C(1) = 1 **Base Case**

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1							

$$C(2) = \max \begin{cases} V_1 + C(1) = 1 + 1 = 2 \\ \\ \end{cases}$$

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1							

$$C(2) = \max \begin{cases} V_1 + C(1) = 1 + 1 = 2 \\ V_2 = 5 \end{cases}$$

Rod Cutting DP

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5						

$$C(2) = \max \begin{cases} V_1 + C(1) = 1 + 1 = 2 \\ V_2 = 5 \end{cases}$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8					

$$C(3) = \max \begin{cases} V_1 + C(2) = 1 + 5 = 6 \\ V_2 + C(1) = 5 + 1 = 6 \\ V_3 = 8 \end{cases}$$

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10				

$$C(4) = \max \begin{cases} V_1 + C(3) = 1 + 8 = 9 \\ V_2 + C(2) = 5 + 5 = 10 \\ V_3 + C(1) = 8 + 1 = 9 \\ V_4 = 9 \end{cases}$$

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10	13			

$$C(5) = \max \begin{cases} V_1 + C(4) = 1 + 10 = 11 \\ V_2 + C(3) = 5 + 8 = 13 \\ V_3 + C(2) = 8 + 5 = 13 \\ V_4 + C(1) = 9 + 1 = 10 \\ V_5 = 10 \end{cases}$$

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10	13	17		

$$C(6) = \max \begin{cases} V_1 + C(5) = 1 + 13 = 14 \\ V_2 + C(4) = 5 + 10 = 15 \\ V_3 + C(3) = 8 + 8 = 16 \\ V_4 + C(2) = 9 + 5 = 14 \\ V_5 + C(1) = 10 + 1 = 11 \\ V_6 = 17 \end{cases}$$

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10	13	17	18	

$$C(7) = \max \begin{cases} V_1 + C(6) = 1 + 17 = 18 \\ V_2 + C(5) = 5 + 13 = 18 \\ V_3 + C(4) = 8 + 10 = 18 \\ V_4 + C(3) = 9 + 8 = 17 \\ V_5 + C(2) = 10 + 5 = 15 \\ V_6 + C(1) = 17 + 1 = 18 \\ V_7 = 17 \end{cases}$$

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10	13	17	18	

$$C(8) = \max \begin{cases} V_1 + C(7) = 1 + 18 = 19 \\ V_2 + C(6) = 5 + 17 = 22 \\ V_3 + C(5) = 8 + 13 = 21 \\ V_4 + C(4) = 9 + 10 = 19 \\ V_5 + C(3) = 10 + 8 = 18 \\ V_6 + C(2) = 17 + 5 = 22 \\ V_7 + C(1) = 17 + 1 = 18 \\ V_8 = 20 \end{cases}$$

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

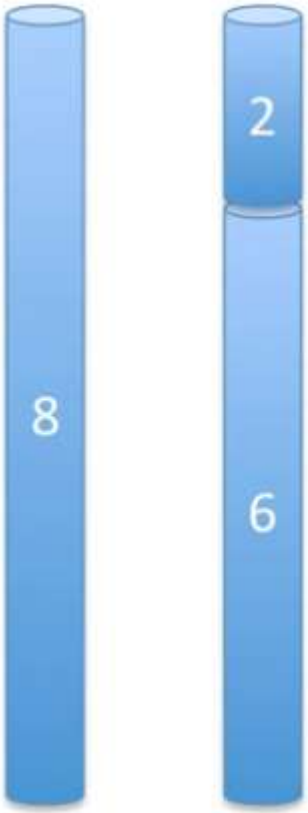
$C(8) = V_2 + C(6) = 5 + 17 = 22$

$C(6) = V_6 = 17$

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10	13	17	18	22



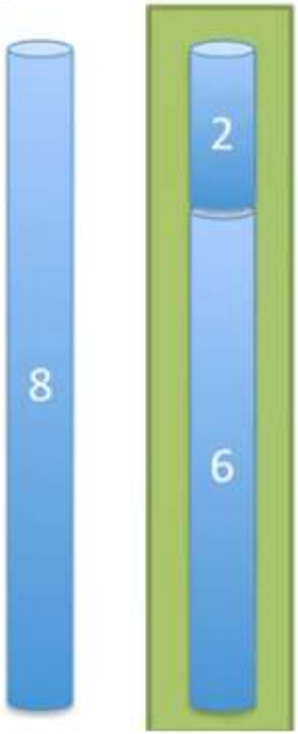
$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

$$C(i) = \max_{1 \le k \le i} \{V_k + C(i-k)\}$$

$C(8) = V_2 + C(6) = 5 + 17 = 22$

$C(6) = V_6 = 17$



Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

C(i)

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10	13	17	18	22

The optimal way to cut a rod of length 8

$$\Rightarrow c_n = max_{1 \le i \le n}(p_i + c_{n-i})$$

Rod Cutting DP

Without dynamic programming, the problem has a complexity of $O(2^n)$ **[Repeated calculations]**

For a rod of length 8, there are 128 (or 2^{n-1}) ways to cut it!

With dynamic programming, and this top down approach, the problem is reduced to $O(n^2)$

Example: Rod Cutting

In general, for any n:

$c_n = max(p_n, p_1 + c_{n-1}, p_2 + c_{n-2}, p_3 + c_{n-3}, \ldots, c_{n-1} + c_1)$

$\Rightarrow c_n = max(p_n + c_0, p_1 + c_{n-1}, p_2 + c_{n-2}, p_3 + c_{n-3}, \ldots, p_{n-1} + c_1)$:[Let, $c_0 = 0$]

$\Rightarrow \boldsymbol{c_n = max_{1 \le i \le n}(p_i + c_{n-i})}$

In other words,

• maximal revenue c_n is obtained by cutting the rod into smaller pieces of lengths: *i* and *(n-i)*, for some value of *i* (for which $\boldsymbol{p_i + c_{n-i}}$ is maximum) where $1 \le i \le n$ and

▪ $c_i = p_i$, *i.e.,* the piece of length *i* need no more cut (because cutting it into smaller pieces will not increase its revenue) whereas

▪ the piece of length *(n-i)* may need more cut; but we have already calculated the maximal revenue, c_{n-i} of a rod of length *(n-i)* before calculating c_n. We can use that value of c_{n-i} to calculate c_n

Top Down algorithm for Rod Cutting

Recursive Top-Down Implementation

- We can obtain r_n from Equation (1) using the following recursive procedure:

Naive recursive implementation

```
1: procedure CutRod(p, n)
2:     if n = 0 then
3:         return 0                                  ▷ base case
4:     q ← −∞
5:     for i ← 1, n do
6:         q ← max(q, p[i] + CutRod(p, n − i))
7:     return q
```

- This algorithm turns out to be very slow. Why?
- Similarly to the naive algorithm from the previous lecture, it repeats the same calculations over and over again. (See recursion tree on next slide.)

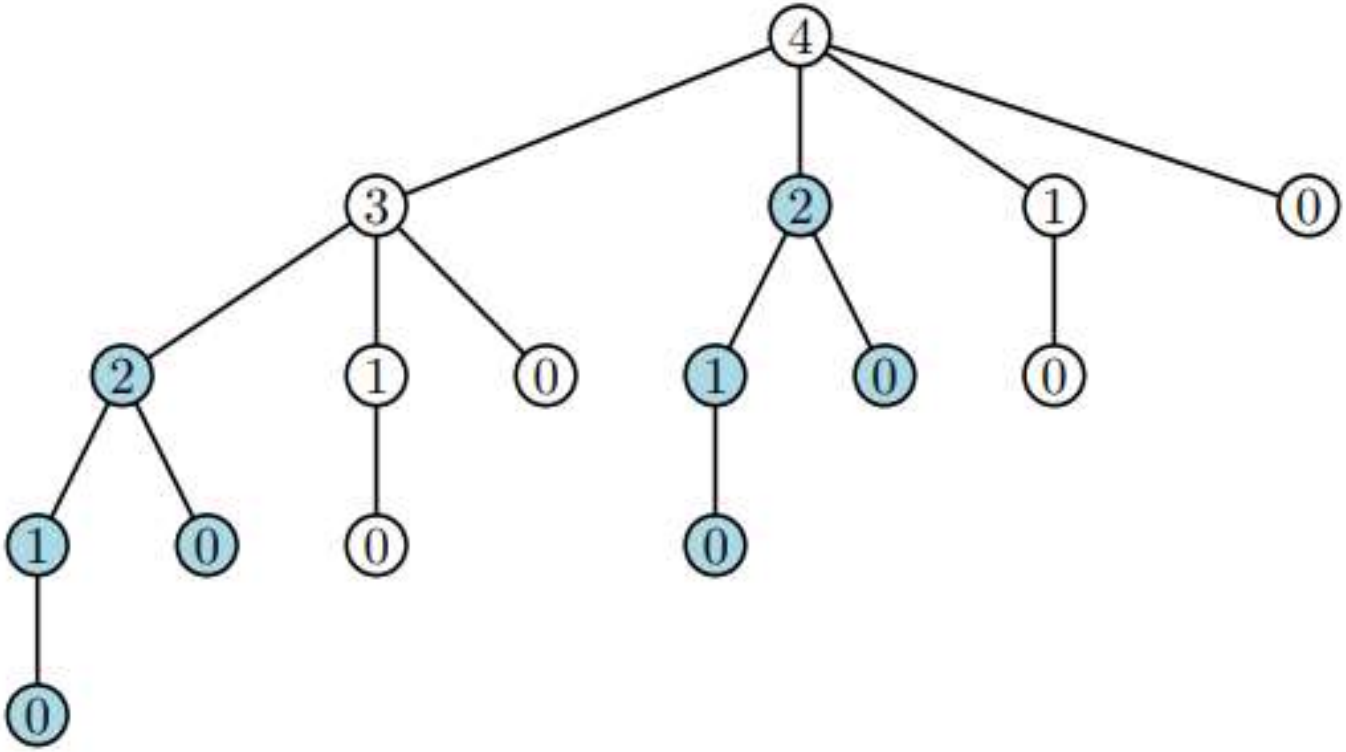
Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

```
CUT-ROD(p, 4) = max(q, p[1] + CUT-ROD(p, 3), p[2] + CUT-ROD(p, 2), p[3] + CUT-ROD(p, 1), p[4] + CUT-ROD(p, 0)) = max(-∞, 1 + 8, 5 + 5, 8 + 1, 9 + 0) = 10
|
+--CUT-ROD(p, 3) = max(q, p[1] + CUT-ROD(p, 2), p[2] + CUT-ROD(p, 1), p[3] + CUT-ROD(p, 0)) = max(-∞, 1 + 5, 5 + 1, 8 + 0) = 8
|   |
|   +--CUT-ROD(p, 2) = max(q, p[1] + CUT-ROD(p, 1), p[2] + CUT-ROD(p, 0)) = max(-∞, 1 + 1, 5 + 0) = 5
|   |   |
|   |   +--CUT-ROD(p, 1) = max(q, p[1] + CUT-ROD(p, 0)) = max(-∞, 1 + 0) = 1
|   |   |   |
|   |   |   +--CUT-ROD(p, 0) = 0
|   |   |
|   |   +--CUT-ROD(p, 0) = 0
|   |
|   +--CUT-ROD(p, 1) = max(q, p[1] + CUT-ROD(p, 0)) = max(-∞, 1 + 0) = 1
|   |   |
|   |   +--CUT-ROD(p, 0) = 0
|   |
|   +--CUT-ROD(p, 0) = 0
|
+--CUT-ROD(p, 2) = max(q, p[1] + CUT-ROD(p, 1), p[2] + CUT-ROD(p, 0)) = max(-∞, 1 + 1, 5 + 0) = 5
|   |
|   +--CUT-ROD(p, 1) = max(q, p[1] + CUT-ROD(p, 0)) = max(-∞, 1 + 0) = 1
|   |   |
|   |   +--CUT-ROD(p, 0) = 0
|   |
|   +--CUT-ROD(p, 0) = 0
|
+--CUT-ROD(p, 1) = max(q, p[1] + CUT-ROD(p, 0)) = max(-∞, 1 + 0) = 1
|   |
|   +--CUT-ROD(p, 0) = 0
|
+--CUT-ROD(p, 0) = 0
|
```

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Recursive Top-Down Implementation

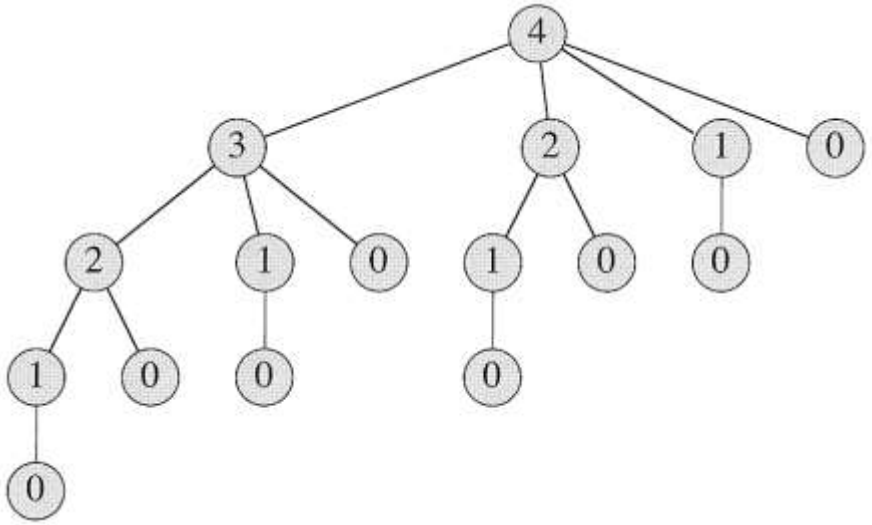


- Entire subtrees are recomputed.
- The running time is $\Omega(2^n)$. Why?
  - It tries the 2^{n-1} possible ways of partitioning the rod.

Time Complexity of Top Down algorithm

Let's call Cut-Rod(p, 4), to see the effects on a simple case:

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```



T(n) = total # of calls made to CUT-ROD for an initial call of CUT-ROD(p,n)
= The # of nodes for a recursion tree corresponding to a rod of size n = ?

Solution: CUT-ROD(p,n) calls CUT-ROD(*p, n-i*) for *i = 1,2,3, ..., n, i.e.,* CUT-ROD(p,n) calls CUT-ROD(*p, j*) for *j = 0,1,2, ..., n-1*

$$T(0) = c, T(n) = c + \sum_{j=0}^{n-1} T(j) = O(2^n), n \geq 1.$$

Why is $T(0) = 1$ in the Given Recurrence?

The given recurrence:

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

suggests an **alternative interpretation** of the problem where:

1. **Each rod configuration contributes a base value of 1**, possibly representing the count of ways to divide the rod rather than revenue.
2. The recurrence resembles the recurrence for counting **combinations of partitions**, rather than maximizing revenue.
3. If we consider the number of ways to "cut" a rod (even when it's empty), we might define **T(0) = 1** as a convention to account for the "empty" rod as a valid configuration.

Alternative Interpretation:

- If the problem is about counting **ways to partition a rod**, rather than maximizing revenue, then having $T(0) = 1$ makes sense because there is exactly **one way to have a rod of length 0**: by not using it.

1. **Given recurrence relation:** $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$

   and $T(0) = 1$.

2. **Using the substitution method, guess $T(n) = O(c^n)$ for some constant c.**

3. **Compute $T(n) - T(n-1)$:**

$$T(n) - T(n-1) = \left(1 + \sum_{j=0}^{n-1} T(j)\right) - \left(1 + \sum_{j=0}^{n-2} T(j)\right)$$

Simplify the right-hand side:

$$T(n) - T(n-1) = T(n-1)$$

Extra Explanation on simplify the expression $T(n) - T(n-1)$:

Given the expression:

$$T(n) - T(n-1) = \left(1 + \sum_{j=0}^{n-1} T(j)\right) - \left(1 + \sum_{j=0}^{n-2} T(j)\right)$$

1. **Expand both summations:**

   The summation $\sum_{j=0}^{n-1} T(j)$ includes all terms from $T(0)$ to $T(n-1)$.

   The summation $\sum_{j=0}^{n-2} T(j)$ includes all terms from $T(0)$ to $T(n-2)$.

2. **Subtract the sums:**

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

3. **Cancel out the common terms:**

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

4. **Simplify to get the remaining term:** After cancelling out the common terms, we are left with:

$$T(n) - T(n-1) = T(n-1)$$

Thus, the recurrence relation simplifies to: $T(n) = 2T(n-1)$

1. **Given recurrence relation:** $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$

   and $T(0) = 1$.

2. **Using the substitution method, guess $T(n) = O(c^n)$ for some constant c.**

3. **Compute $T(n) - T(n-1)$:**

$$T(n) - T(n-1) = \left(1 + \sum_{j=0}^{n-1} T(j)\right) - \left(1 + \sum_{j=0}^{n-2} T(j)\right)$$

   Simplify the right-hand side:

$$T(n) - T(n-1) = T(n-1)$$

4. **This simplifies to:** $T(n) = 2T(n-1)$

5. **Now, use the initial condition $T(0) = 1$ and solve the simplified recurrence:**

$$T(n) = 2T(n-1)$$

For $n = 1$: $T(1) = 2T(0) = 2 \cdot 1 = 2$

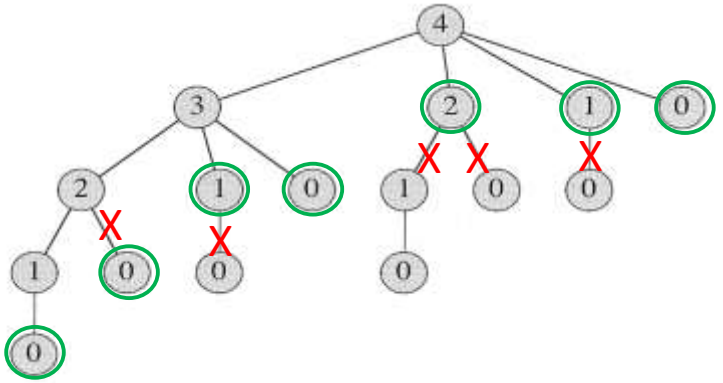
For $n = 2$: $T(2) = 2T(1) = 2 \cdot 2 = 4$

For $n = 3$: $T(3) = 2T(2) = 2 \cdot 4 = 8$

This pattern continues, showing that: $T(n) = 2^n$

The recurrence relation $T(n) = 2T(n-1)$ results in an exponential growth, specifically 2^n

Top-Down Memoization



Top-down memoization of CUTROD

```
1: procedure MEMOIZEDCUTROD(p, n)
2:     r[0...n] ← new array filled with −∞
3:     return AUXILIARY(p, n, r)
4: procedure AUXILIARY(p, n, r)
5:     if r[n] ≠ −∞ then                    ▷ checks if the result is in the table
6:         return r[n]                      ▷ if so, return it immediately
7:     if n = 0 then                        ▷ base case
8:         q ← 0
9:     else                                 ▷ recursive computation of r[n]
10:        q ← −∞
11:        for i = 1, n do
12:            q ← max(q, p[i] + AUXILIARY(p, n − i, r))
13:    r[n] ← q
14:    return q
```

Length	1	2	3	4	5	6	7	8
Price $	1	5	8	9	10	17	17	20

Analysis

Theorem

The memoized version of CUTROD *runs in* $\Theta(n^2)$ *time.*

Proof.

- Line 2–3: $O(1)$
- Line 5–6: $O(k)$ where k is the number of calls to AUXILIARY
- Line 7–14: $O(\ell n)$ where ℓ is the number of times we reach Line 7
- $\ell = n + 1$ because the value in each cell of the array $r[0 \ldots n]$ is computed only once
- $k = O(n^2)$ because for each time we reach Line 7, we execute Line 12 $O(n)$ times

This shows that the running time is $O(n^2)$. Conversely, the loop 11–12 is executed once for each value of n, as the algorithm computes all values of $r[i]$, so the running time is at least $\Omega(\sum_{i=1}^{n} i) = \Omega(n^2)$. □

Analysis

```
1:  procedure MemoizedCutRod(p, n)
2:      r[0...n] ← new array filled with −∞
3:      return Auxiliary(p, n, r)
4:  procedure Auxiliary(p, n, r)
5:      if r[n] ≠ −∞ then                    ▷ checks if the resu
6:          return r[n]                      ▷ if so, retu
7:      if n = 0 then
8:          q ← 0
9:      else                                 ▷ recursive cor
10:         q ← −∞
11:         for i = 1, n do
12:             q ← max(q, p[i] + Auxiliary(p, n − i, r))
13:     r[n] ← q
14:     return q
```

Theorem

The memoized version of CutRod *runs*

Proof.

- Line 2–3: $O(1)$
- Line 5–6: $O(k)$ where k is the number of calls to Auxiliary
- Line 7–14: $O(\ell n)$ where ℓ is the number of times we reach Line 7
- $\ell = n + 1$ because the value in each cell of the array $r[0 \ldots n]$ is computed only once
- $k = O(n^2)$ because for each time we reach Line 7, we execute Line 12 $O(n)$ times

This shows that the running time is $O(n^2)$. Conversely, the loop 11–12 is executed once for each value of n, as the algorithm computes all values of $r[i]$, so the running time is at least $\Omega(\sum_{i=1}^{n} i) = \Omega(n^2)$. □

1. $O(k)$: This represents the total number of calls to the auxiliary function Memoized-Cut-Rod-Aux.

2. $O(\ell n)$: This represents the complexity of the loop within each call to the function. Here:
   - ℓ is the total number of calls to Memoized-Cut-Rod-Aux.
   - n: This is the number of iterations of the inner loop (line 6) for each call to the function.
   - $\ell = n + 1$ because we need to compute values for all subproblems from 0 to n, inclusive.

- **Combining Factors**: Since there are $\ell = n + 1$ calls to the function, and each call involves a loop running up to n times, the total complexity is the product of the number of calls and the number of iterations per call.

$$\text{Total Complexity} = O(\ell n) = O((n+1)n) \approx O(n^2)$$

Proof of Correctness

Loop Invariant:

A loop invariant is a property that holds before and after each iteration of a loop. For the dynamic programming solution to the rod cutting problem, the loop invariant is that $r[j]$ contains the maximum revenue obtainable for a rod of length j.

Initialization

- **Condition**: Before any iteration begins.
- **Invariant**: $r[0]$ is correctly set to 0.
- **Explanation**: At the start, before any iterations, the maximum revenue for a rod of length 0 is 0, which is correctly initialized.

Proof of Correctness

Loop Invariant:

A loop invariant is a property that holds before and after each iteration of a loop. For the dynamic programming solution to the rod cutting problem, the loop invariant is that $r[j]$ contains the maximum revenue obtainable for a rod of length j.

Maintenance

- **Condition**: For each length j from 1 to n.
- **Invariant**: The algorithm correctly computes $r[j]$ using the recurrence relation.
- **Explanation**: During each iteration, the algorithm updates $r[j]$ to ensure it holds the maximum revenue for a rod of length j. This is done by checking all possible ways to cut the rod and choosing the option that gives the highest revenue. The recurrence relation ensures that $r[j]$ is the sum of the price of a piece of length i and the maximum revenue for the remaining piece of length $j - i$.

Proof of Correctness

Loop Invariant:

A loop invariant is a property that holds before and after each iteration of a loop. For the dynamic programming solution to the rod cutting problem, the loop invariant is that $r[j]$ contains the maximum revenue obtainable for a rod of length j.

Termination

- **Condition**: When the loop finishes after processing length n.
- **Invariant**: $r[n]$ contains the maximum revenue obtainable for a rod of length n.
- **Explanation**: When the loop terminates, the algorithm has processed all lengths from 0 to n. Hence, $r[n]$ will contain the maximum revenue for the full length of the rod n.

Analysis

- Similarly as for the computation of binomial coefficients, dynamic programming brings the running time down from exponential to polynomial.

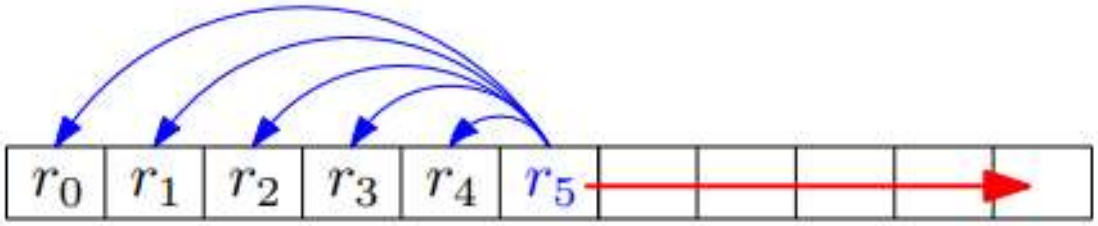
Bottom-Up Method

- Another way of doing dynamic programming is the *bottom-up method*.
- It is the method we used for computing binomial coefficients.
- The idea is to consider the subproblems in an order that guarantees that, at any step, all the intermediate results we need have already been computed.
- In the previous lecture, we achieved it by computing $\binom{n}{k}$ by increasing value of n, and decreasing value of k.
- Often it is achieved using a notion of *size* of subproblems, and solving smaller subproblems first.

• **Bottom Up Algorithm**: Compute small subproblems first, then gradually solve larger and larger subproblems by using the pre-computed solutions of smaller subproblems. For e.g., for rod-cutting problem: compute $r_0, r_1, r_2, \ldots, r_n$

Bottom-Up Method

length i	1	2	3	4	5	6	7	8	9	10
price p_i	2	5	6	7	8	9	11	11	13	15



Bottom-up version of CutRod

```
1: procedure BottomUpCutRod(p, n)
2:     r[0...n] ← new array        // r => revenue generated
3:     r[0] ← 0
4:     for j = 1, n do
5:         q ← −∞
6:         for i ← 1, j do
7:             q ← max(q, p[i] + r[j − i])
8:         r[j] ← q
9:     return r[n]
```

Simulation: Rod Cutting

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	12	17

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

We begin by constructing (by hand) the optimal solutions for i = 1, …, 10:

$r_0 = 0$

$r_1 = p_1 = 1$ (no cuts)

$r_2 = \max(\mathbf{p_2}, p_1+r_1) = \max(\mathbf{5}, 1+1) = \mathbf{5}$ (no cuts)

$r_3 = \max(\mathbf{p_3}, p_2+r_1, p_1+r_2) = \max(\mathbf{8}, 5+1, 1+5) = \mathbf{8}$ (no cuts)

$r_4 = \max(p_4, p_3+r_1, \mathbf{p_2+r_2}, p_1+r_3) = \max(9, 8+1, \mathbf{5+5}, 1+8) = \mathbf{10}$

$r_5 = ?$

$r_6 = ?$

$r_7 = ?$

Simulation: Rod Cutting

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	12	17

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

We begin by constructing (by hand) the optimal solutions for i = 1, …, 10:

$r_0 = 0$

$r_1 = \mathbf{p_1 = 1}$ (no cuts)

$r_2 = \max(\mathbf{p_2}, p_1+r_1) = \max(\mathbf{5}, 1+1) = \mathbf{5}$ (no cuts)

$r_3 = \max(\mathbf{p_3}, p_2+r_1, p_1+r_2) = \max(\mathbf{8}, 5+1, 1+5) = \mathbf{8}$ (no cuts)

$r_4 = \max(p_4, p_3+r_1, \mathbf{p_2+r_2}, p_1+r_3) = \max(9, 8+1, \mathbf{5+5}, 1+8) = \mathbf{10}$

$r_5 = \max\{p_5, p_4+r_1, p_3+r_2, \mathbf{p_2+r_3}, p_1+r_4\}$

$= \max(10, 9+1, \mathbf{8+5}, 5+8, 1+10) = \mathbf{13}$

$r_6 = \max\{p_6, p_5+r_1, p_4+r_2, \mathbf{p_3+r_3}, p_2+r_4, p_1+r_5\}$

$= \max\{12, 11, 14, \mathbf{16}, 15, 14\} = 16$

$r_7 = \mathbf{p2} + \mathbf{r5} = 5 + 13 = 18$

Bottom-Up Method

Recurrence Relation

- So we either have $r_n = p_i + r_{n-i}$ for some i, or $r_n = p_n$.
- Then if we write $r_0 = 0$, it means:

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

```
1: procedure BottomUpCutRod(p, n)
2:     r[0...n] ← new array
3:     r[0] ← 0
4:     for j = 1, n do
5:         q ← −∞
6:         for i ← 1, j do
7:             q ← max(q, p[i] + r[j − i])
8:         r[j] ← q
9:     return r[n]
```

- Correctness follows from Equation 1, and the fact that, at line 7, $r[j-i]$ has already been computed because $j - i < j$.
- It runs in time $\Theta(n^2)$ because line 7 is iterated $1 + 2 + \cdots + n = \Theta(n^2)$ times.
- So the bottom-up method is asymptotically as fast as the memoized version of CutRod, and is simpler.
- It will often be the case, so the bottom-up method is often employed.
- On the other hand, the bottom-up method requires to find a proper ordering of the subproblems, which is not needed for the memoized top-down approach.