CS-3510: Design and Analysis of Algorithms

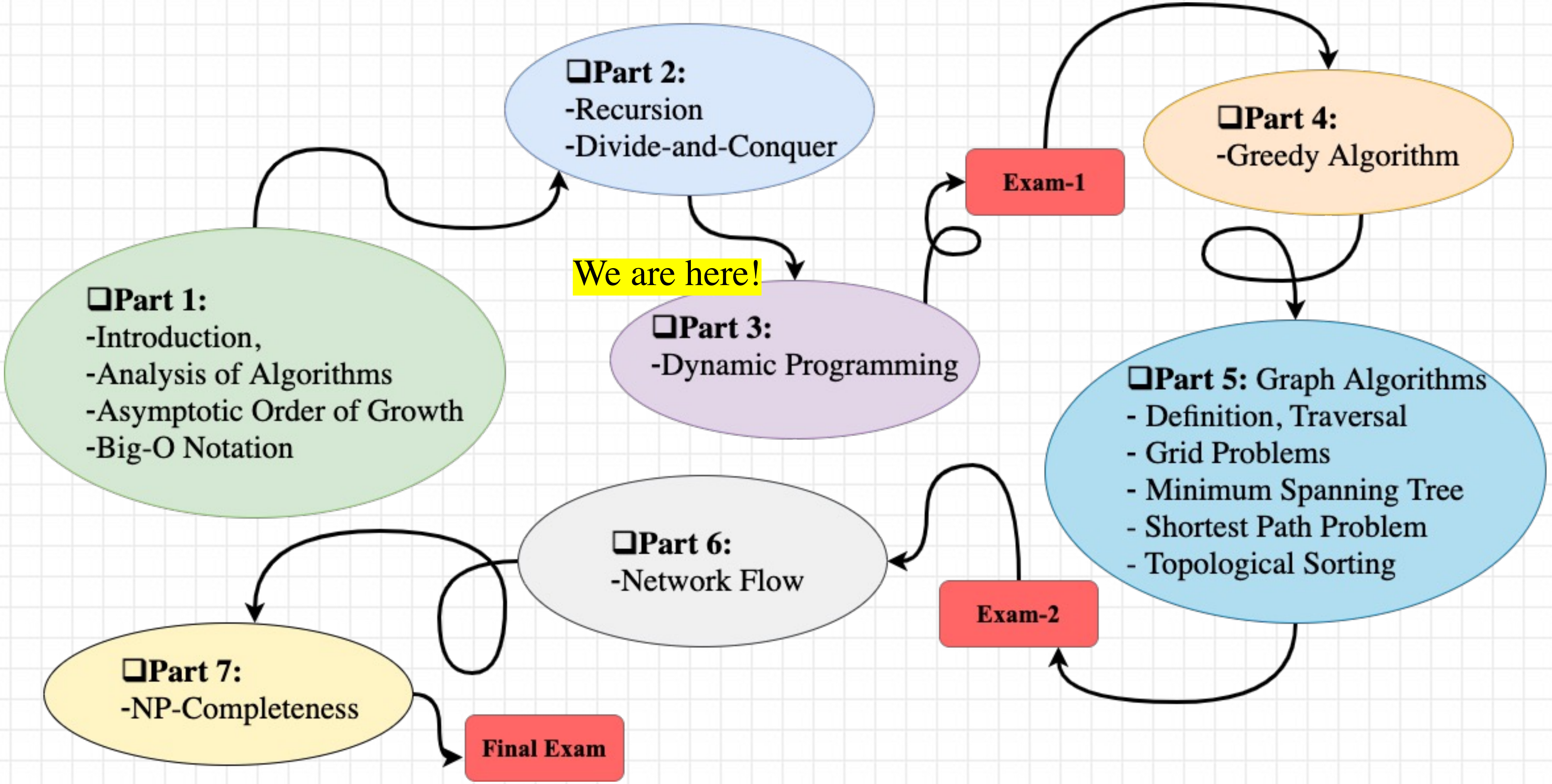
Dynamic Programming I

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology

Summer 2022

Roadmap


Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
Part 2:
-Recursion
-Divide-and-Conquer
We are here!
Part 3:
-Dynamic Programming
Exam-1
Part 4:
-Greedy Algorithm
Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
Exam-2
Part 6:
-Network Flow
Part 7:
-NP-Completeness
Final Exam

A Note about Recursive Algorithms

- In general, recursive algorithms can be used in various setups:
  - Backtracking
    - Ex. Enumerating all subsets of a given set or array
    - Usually (not always!), in these cases we can expect an exponential runtime $O(a^n)$, where a is the number of possible options to choose at each step which is equal to the number branches after each node in the recursion tree.
  - Divide-and-Conquer (D&C)
  - Dynamic programming (DP)
  - Traversing a graph or tree using the depth-first search (DFS) approach

Dynamic Programming (DP)

- Nothing to do with computer "programming"; a term defined by Richard Bellman back in the 1950's
  - "Dynamic" captures the time-varying aspect of the solution approach
  - "Programming" because "it sounded impressive"; real interest was in defining schedules and plans (same sense as linear *programming*)
- Not a particular algorithm, but rather an algorithmic paradigm for developing algorithms.

Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer

Divide-and-Conquer:

- Divide problem into subproblems
- Recursively solve the subproblems and aggregate solutions

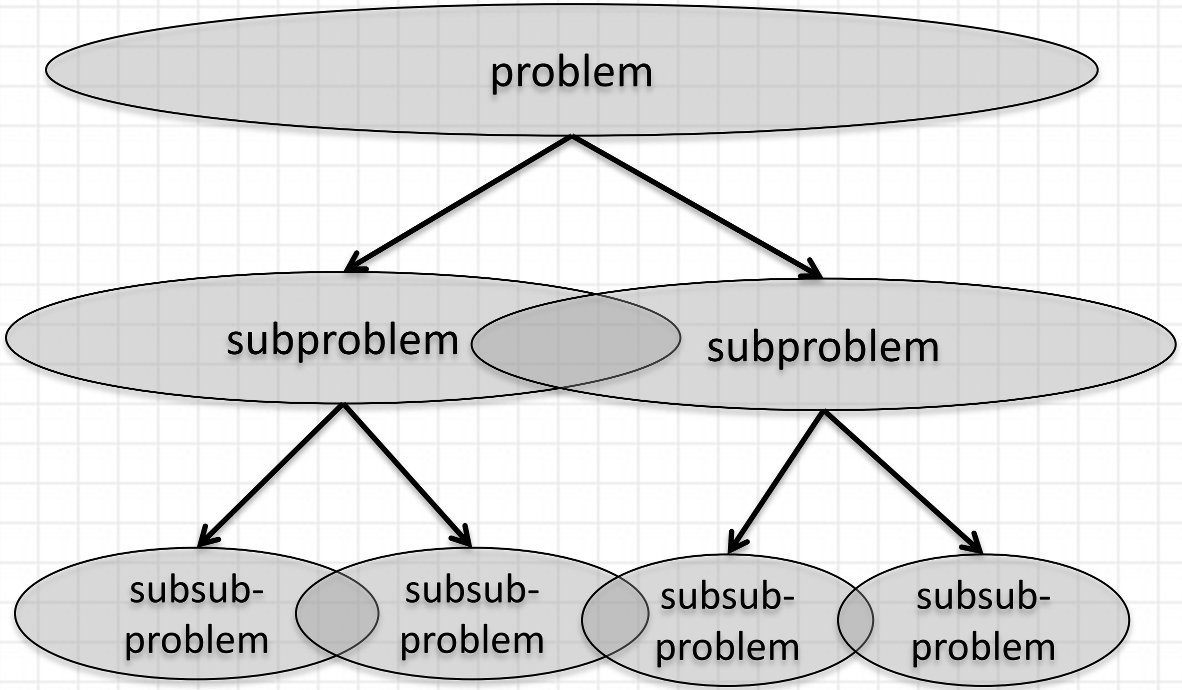
Note: The subproblems <u>do not overlap</u>

Dynamic Programming

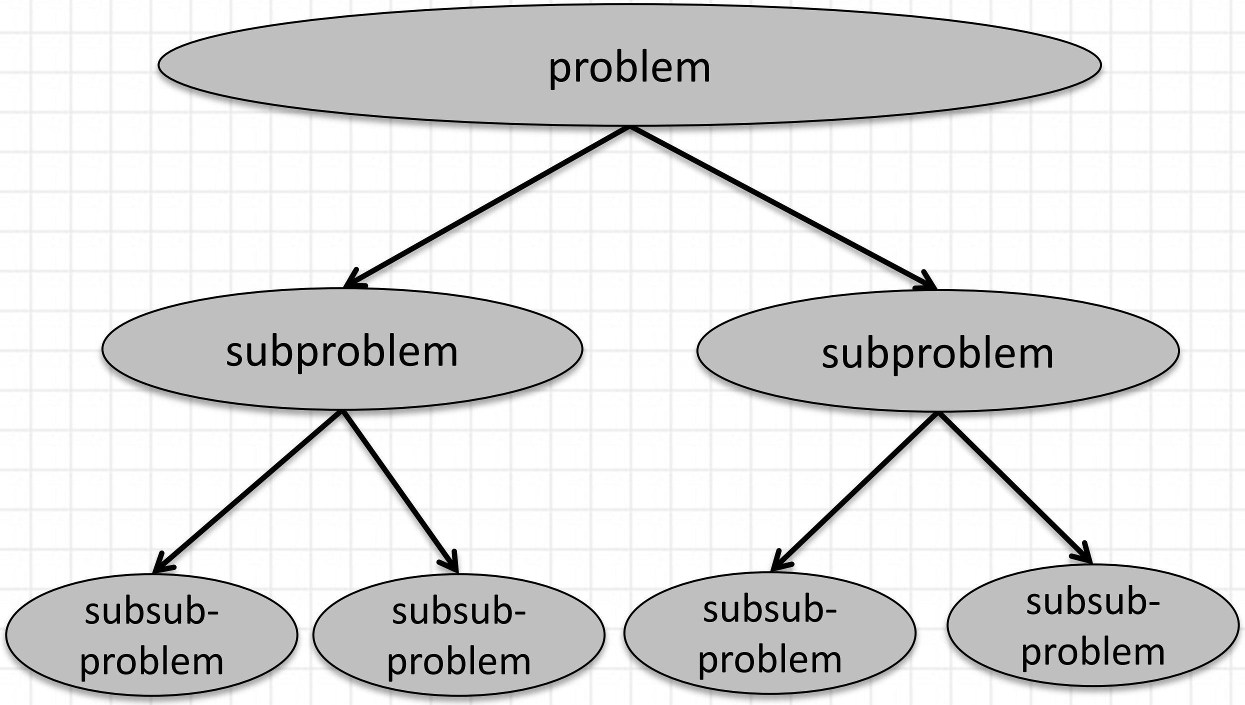
- Divide problem into subproblems, recursively solve them
- Subproblems <u>overlap</u>
- When a subproblem has been solved, remember its solution and reuse that solution rather than resolving it later (memoization)

Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer



Subproblems overlap



Subproblems do not overlap

DP Example: (1) Fibonacci

The Nth Fibonacci number F_N is defined as:

- $F_0 = 0$
- $F_1 = 1$
- for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

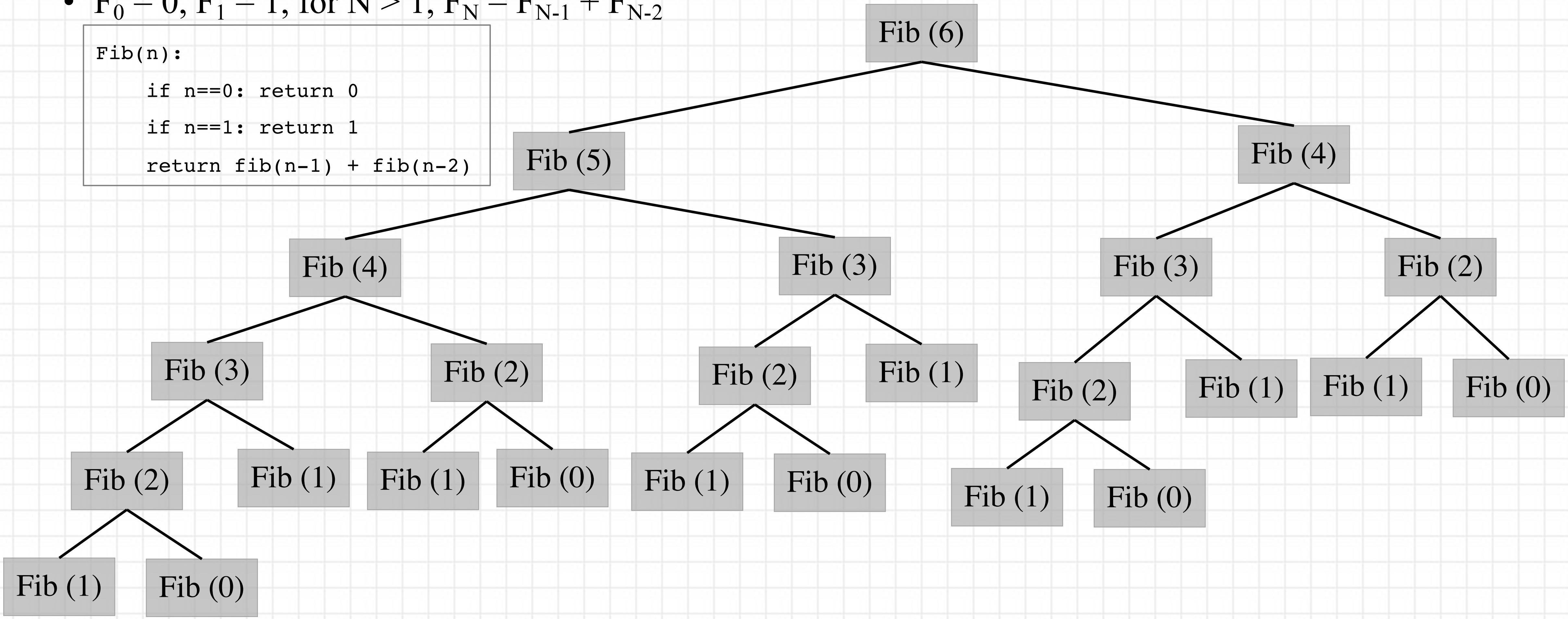
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```

- Recursive relation
- Recursion tree
- Let’s calculate Fib(6)

DP Example: (1) Fibonacci

- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

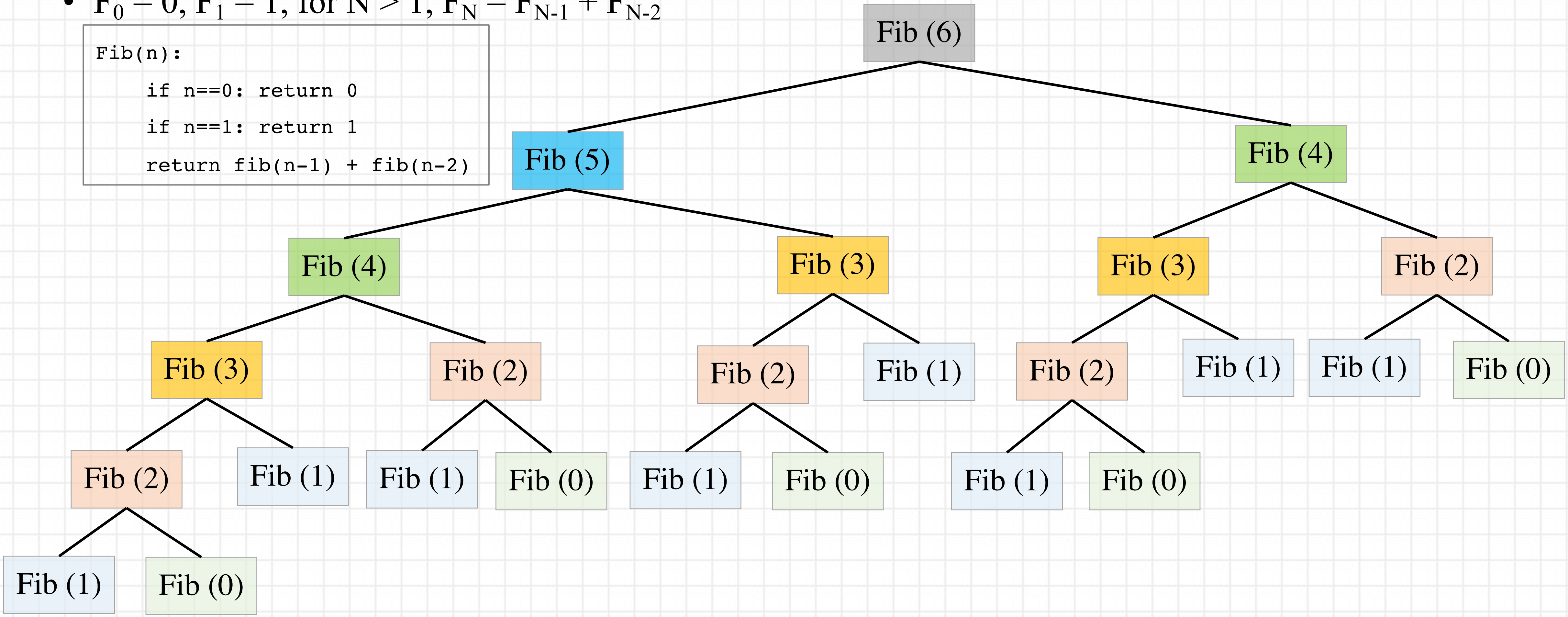
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



DP Example: (1) Fibonacci

- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

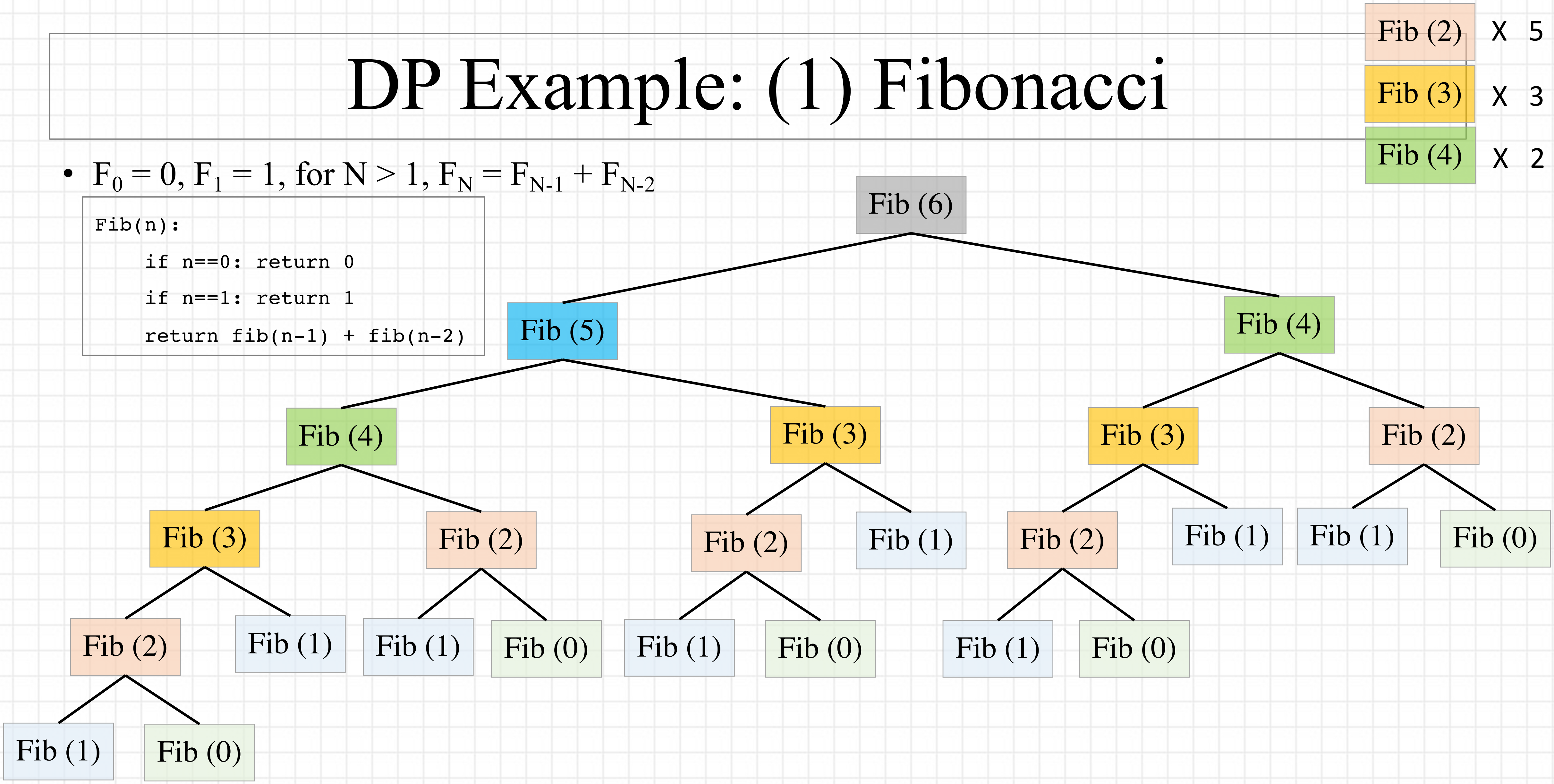
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



DP Example: (1) Fibonacci

- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

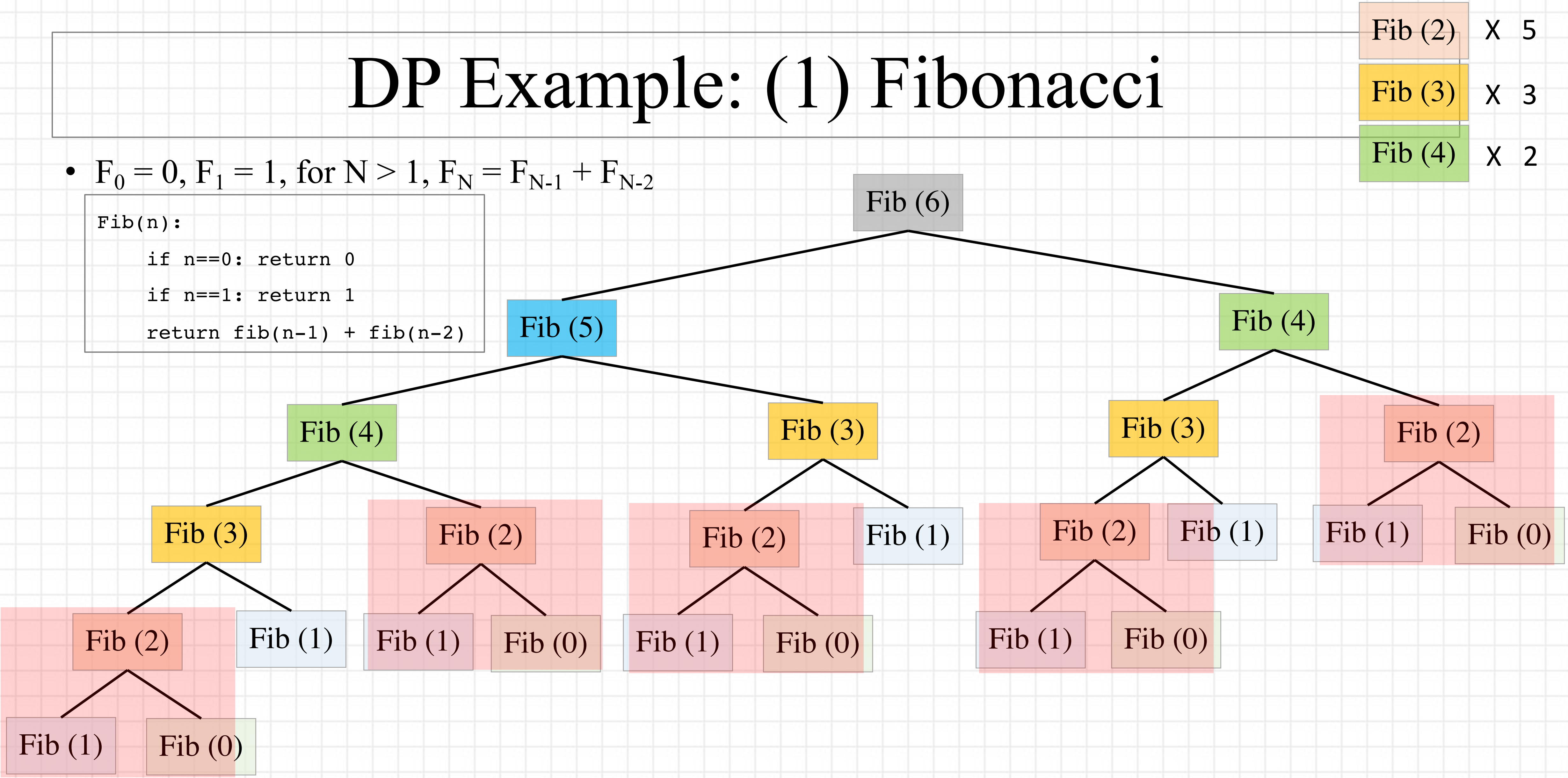
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



DP Example: (1) Fibonacci

- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

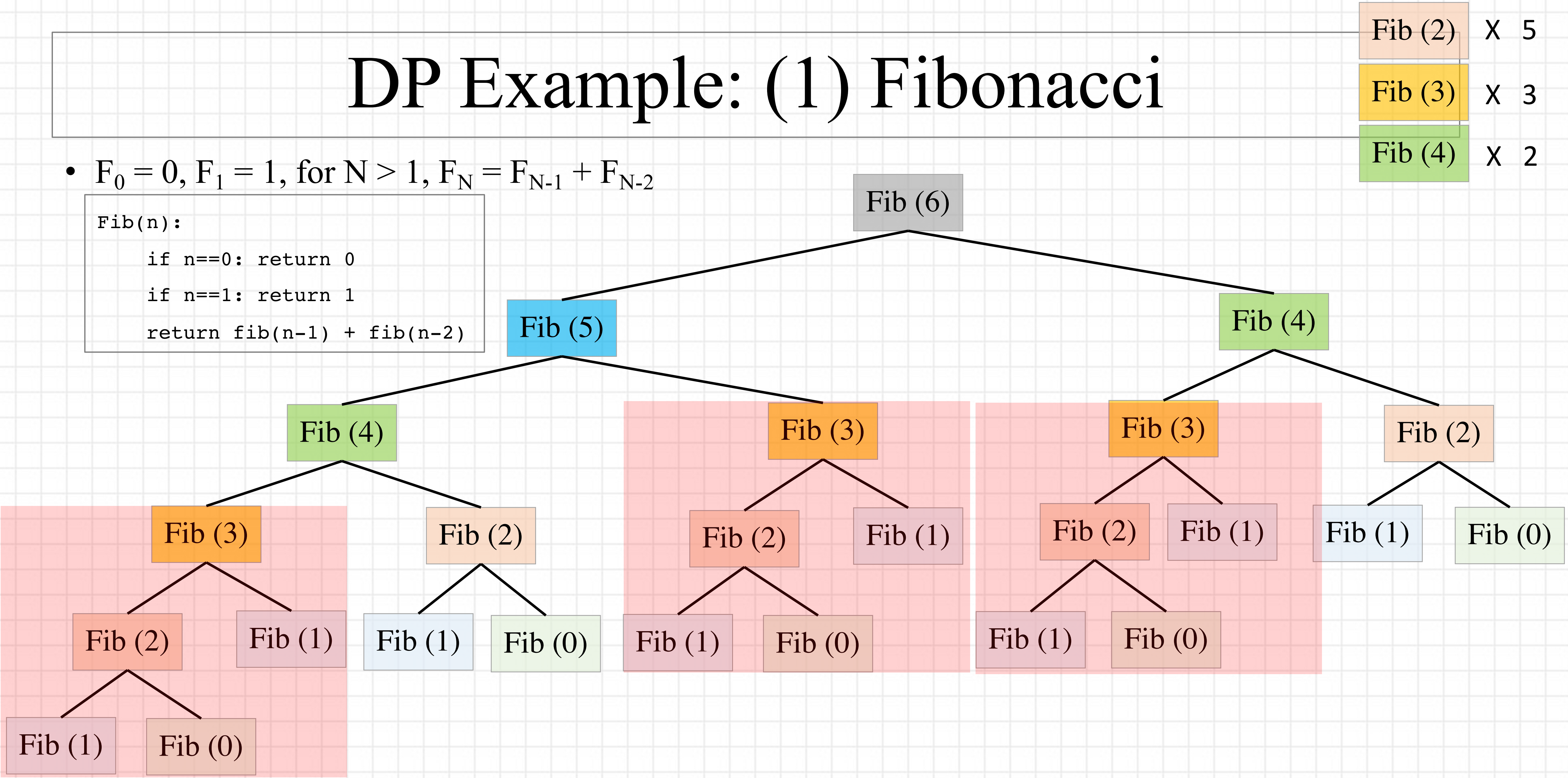
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



DP Example: (1) Fibonacci

- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

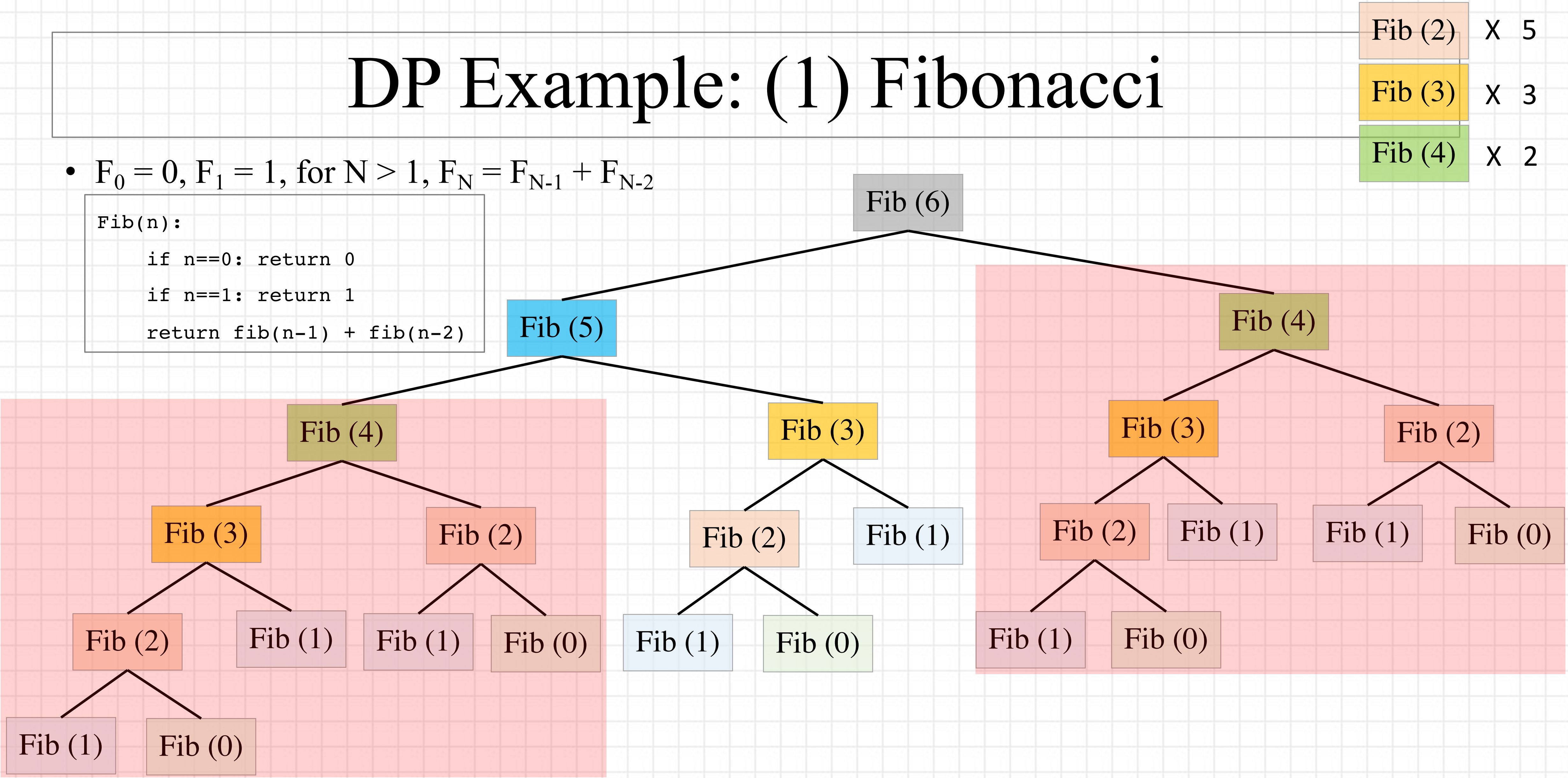
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



DP Example: (1) Fibonacci

- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

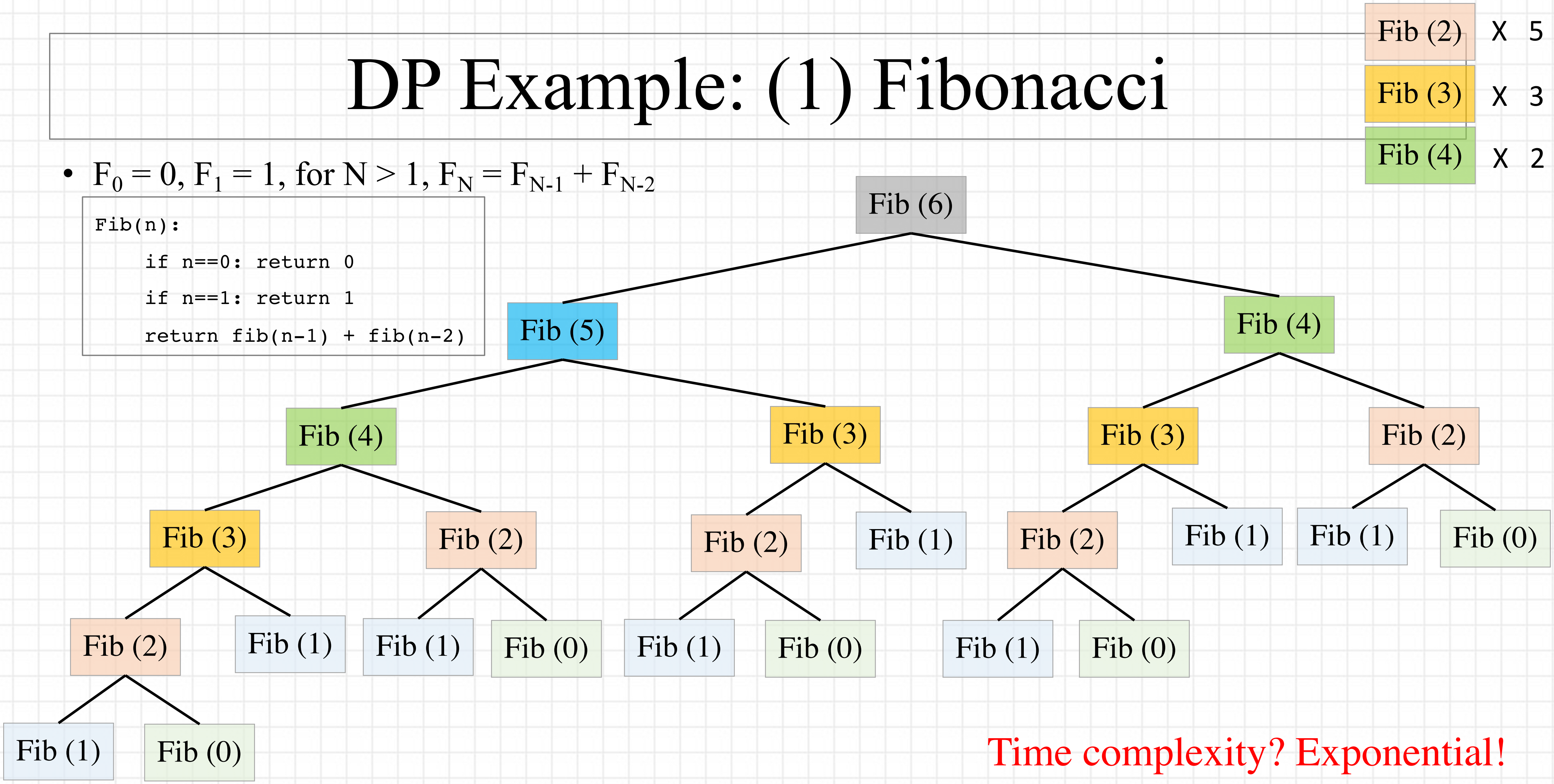
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



DP Example: (1) Fibonacci

- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



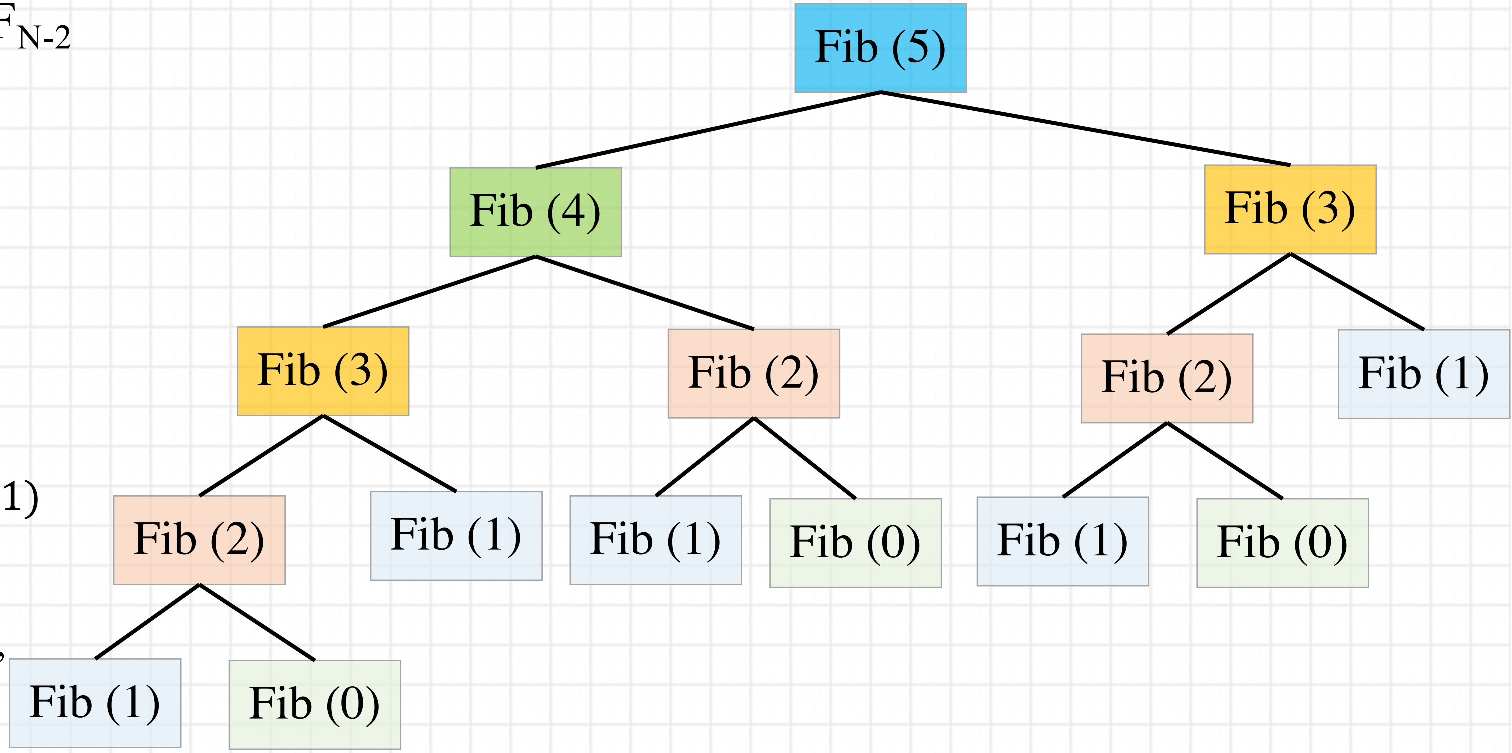
Time complexity? Exponential!

DP Example: (1) Fibonacci

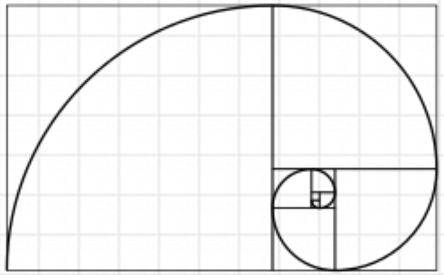
- $F_0 = 0$, $F_1 = 1$, for $N > 1$, $F_N = F_{N-1} + F_{N-2}$

Time: $O(2^n)$, Space: $O(1)$

```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



- Time complexity: (CLRS 4-4)
  - Recurrence: $T(n) = T(n-1) + T(n-2) + O(1)$
  - $T(n) \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \in O(2^n)$
  - $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the "golden ration"



Time complexity? Exponential!

DP Example: (1) Fibonacci

- Using dynamic programming paradigm:
  - Save computed value of Fib(i) in dp[i]
  - If Fib(i) has already been computed, use dp[i] rather than recomputing it

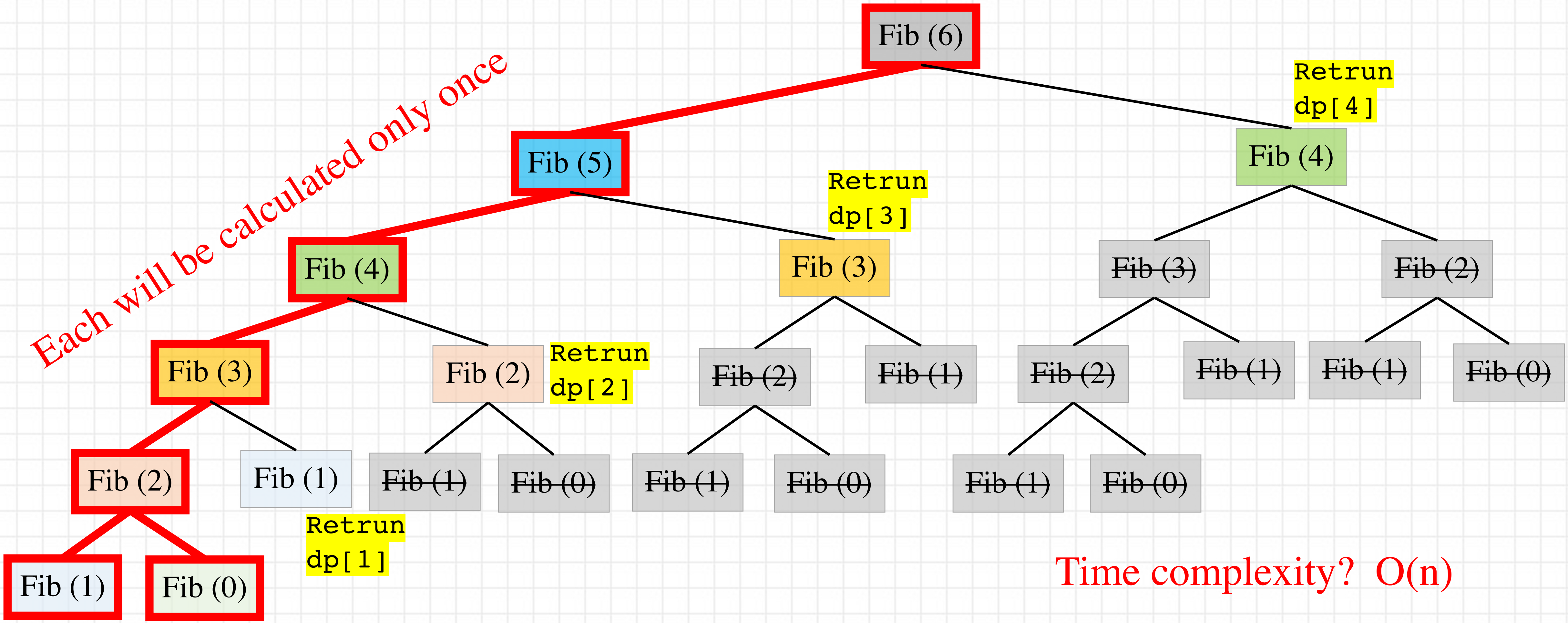
```
Fib(n):
    dp = [0]*n    # initialize dp[i]=0
    recur(i):
        if n==0: return 0
        if n==1: return 1
        if dp[i]==0:
            dp[i] = recur(i-1) + recur(i-2)
        return dp[i]

     return recur(n)
```

Time-memory trade-off

Time complexity?

DP Example: (1) Fibonacci



Time complexity? O(n)

Dynamic Programming

- Top-down vs. Bottom-up Approach
  - The development approach just described is called "top-down" dynamic programming
    - Begin with problem description
    - Recursively subdivide problem into subproblems
    - i.e., begin at root of tree and work downwards

Recursive with memoization

  - Another approach is "bottom-up" dynamic programming
    - Start at the leaf nodes of tree; solution is simple
    - Build up solution to larger problem from solutions of the simpler subproblems

Iterative

DP Example: (1) Fibonacci

- Top-down (recursive with memoization)

Time: O(n), Space: O(n)

```
Fib(n):
    dp = [0]*n    # initialize dp[i]=0
    recur(i):
        if n==0: return 0
        if n==1: return 1
        if dp[i]==0:
            dp[i] = recur(i-1) + recur(i-2)
        return dp[i]

    return recur(n)
```

Bottom-up (iterative)

Time: O(n), Space: O(n)

```
Fib(n):
    dp = [0]*n    # initialize dp[i]=0
    dp[0] = 0
    dp[1] = 1
    for i=2,…,n:
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Do we need to store all values?

DP Example: (1) Fibonacci

- Top-down (recursive with memoization)

Time: O(n), Space: O(n)

```
Fib(n):
    dp = [0]*n    # initialize dp[i]=0
    recur(i):
        if n==0: return 0
        if n==1: return 1
        if dp[i]==0:
            dp[i] = recur(i-1) + recur(i-2)
        return dp[i]

     return recur(n)
```

Bottom-up (iterative)

Time: O(n), Space: O(n)

```
Fib(n):
    dp = [0]*n    # initialize dp[i]=0
    dp[0] = 0
    dp[1] = 1
    for i=2,…,n:
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Each computation needs only the last two Fibonacci numbers!

Re-write the code with two scalars.

DP Example: (1) Fibonacci

- Top-down (recursive with memoization)

Time: O(n), Space: O(n)

```
Fib(n):
    dp = [0]*n    # initialize dp[i]=0
    recur(i):
        if n==0: return 0
        if n==1: return 1
        if dp[i]==0:
            dp[i] = recur(i-1) + recur(i-2)
        return dp[i]

    return recur(n)
```

Bottom-up (iterative)

Time: O(n), Space: O(1)

```
Fib(n):
    f1 = 0
    f2 = 1
    for i=2,…,n:
        f = f1 + f2
        f1 = f2; f2 = f
    return f
```

Each computation needs only the last two Fibonacci numbers!

Re-write the code with two scalars.

DP Example: (1) Fibonacci

- So, which one is better?

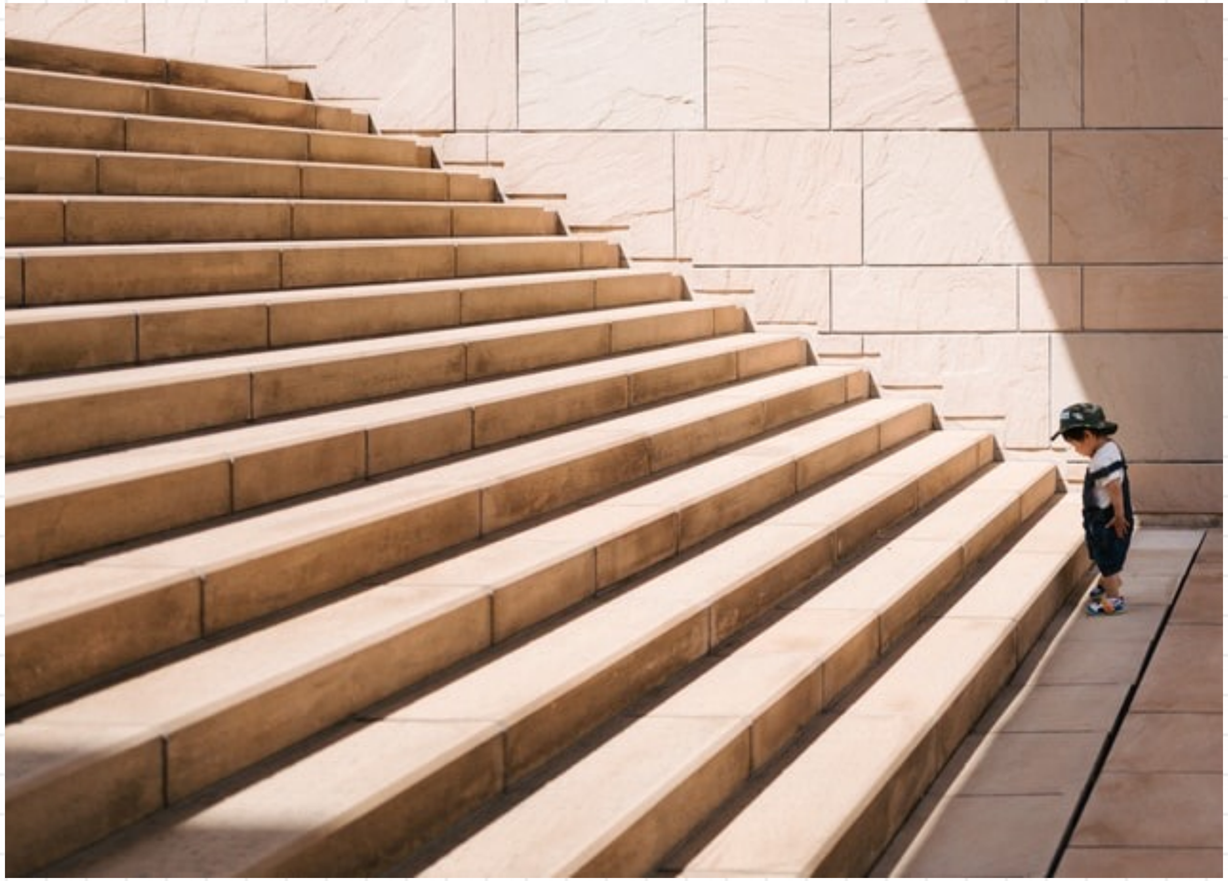
Demo: Fibonacci

Top-down (recursive with memoization)	Bottom-up (iterative) (a.k.a tabulation)

- Starts with the root of the recursion tree
- Implemented as recursive function
- [Memoization:] The result (returned values) of each recursive call will be stored in a data structure, such as array or hashmap (dictionary in Python)
- Main advantage:
    - Easier (more "intuitive") to write, as we don't need to know the ordering of the recursion calls and sub-problems
	- Starts with base cases
	- Implemented with iteration (loop)
	
	- Main advantage:
	    - Avoiding the recursion overhead (recursive calls). So, in practice, to program may run slightly faster.
	    - "Sometimes" it allows to use less memory.

DP Example: (2) Climbing Stairs

- Problem:
  - We want to climb a staircase
  - The staircase has n steps.
  - Each time we can take either 1 or 2 steps.
  - In how many distinct ways we can reach to the top?



DP Example: (2) Climbing Stairs

- Problem:

- We want to climb a staircase
- The staircase has n steps.
- Each time we can take either 1 or 2 steps.
- In how many distinct ways we can reach to the top?

DP Solution:

- Let dp[i] = number of distinct ways to reach i^{th} step.
- Recurrence relation**: dp[i] = dp[i-1] + dp[i-2]**
- Base case(s):
    - **dp[0] = 0, (when we are on the ground, no stairs)**
    - **dp[1] = 1, (only one way to reach step 1)**
    - **dp[2] = 2 (we have two ways to reach step 2)**

DP Example: (2) Climbing Stairs

- Top-down (recursive with memoization)

Time: O(n), Space: O(n)

```
StairClimbing(n):
    dp = [0]*(n+1)   # initialize dp[i]=0
    recur(i):
        if n==0: return 0
        if n==1: return 1
        if n==2: return 2
        if dp[i]==0:
            dp[i] = recur(i-1) + recur(i-2)
        return dp[i]
     return recur(n)
```

Bottom-up (iterative)

Time: O(n), Space: O(n)

```
StairClimbing(n):
    dp = [0]*(n+1)  # initialize dp[i]=0
    dp[0] = 0
    dp[1] = 1
    dp[2] = 2
    for i=3,…,n:
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Similar to Fibonacci we can re-write the code with two scalars.

DP Example: (2) Climbing Stairs

- Top-down (recursive with memoization)

Time: O(n), Space: O(n)

```
StairClimbing(n):
    dp = [0]*(n+1)    # initialize dp[i]=0
    recur(i):
        if n==0: return 0
        if n==1: return 1
        if n==2: return 2
        if dp[i]==0:
            dp[i] = recur(i-1) + recur(i-2)
        return dp[i]
    return recur(n)
```

Bottom-up (iterative)

Time: O(n), Space: O(1)

```
StairClimbing(n):
    if n < 3: return n
    f1 = 1
    f2 = 2
    for i=3,…,n:
        f = f1 + f2
        f1 = f2; f2 = f
    return f
```

Similar to Fibonacci we can re-write the code with two scalars.

DP Example: (3) Rod-cutting

- Problem:

  Given a rod of length n inches and a table of prices pi for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

DP Example: (3) Rod-cutting

- Problem:

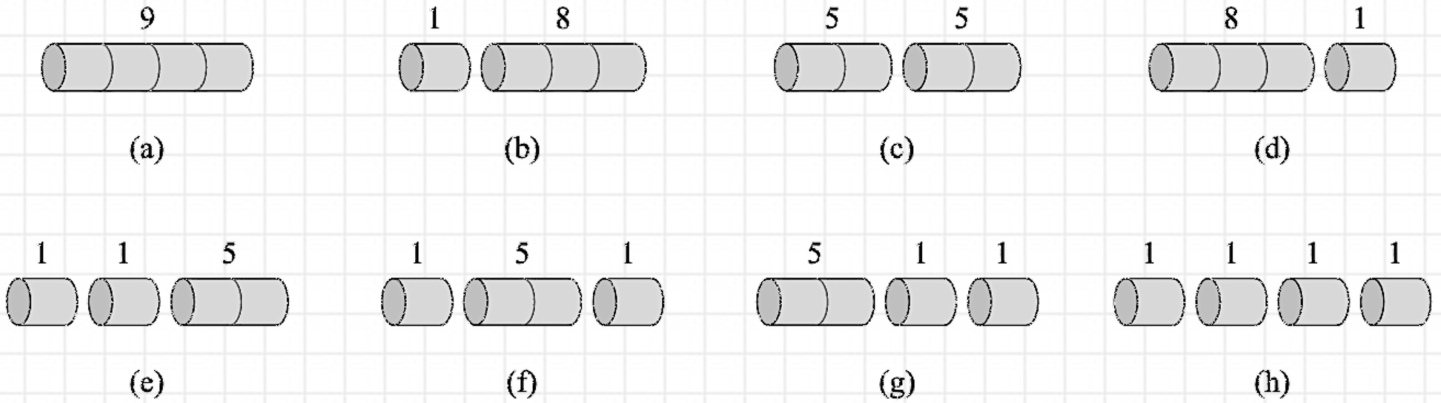
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

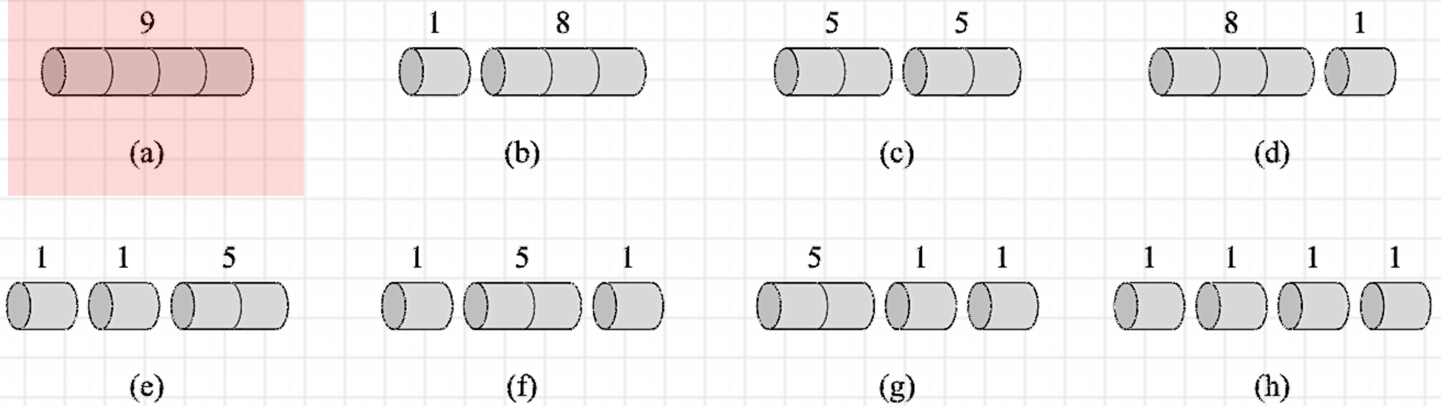
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

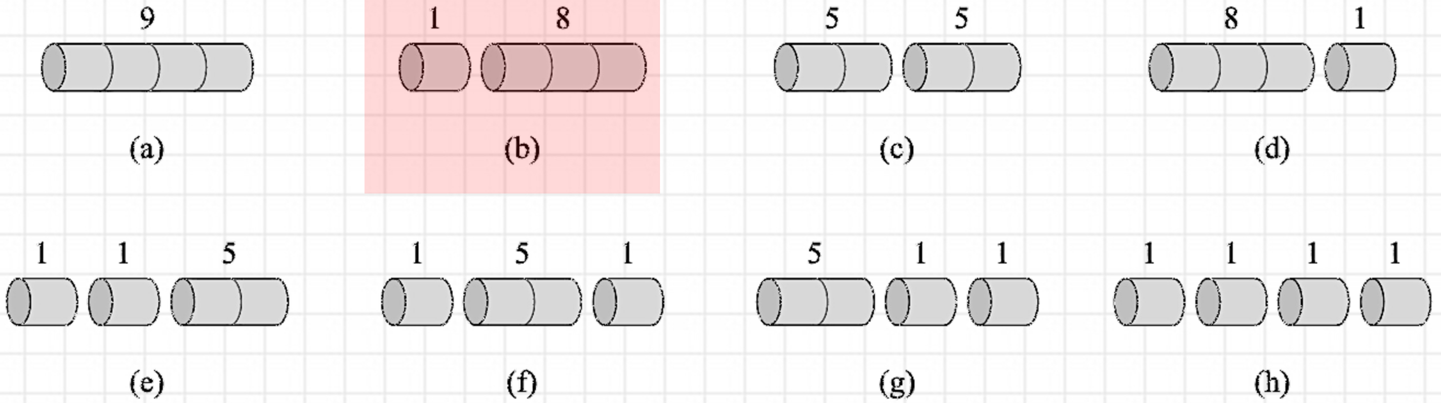
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

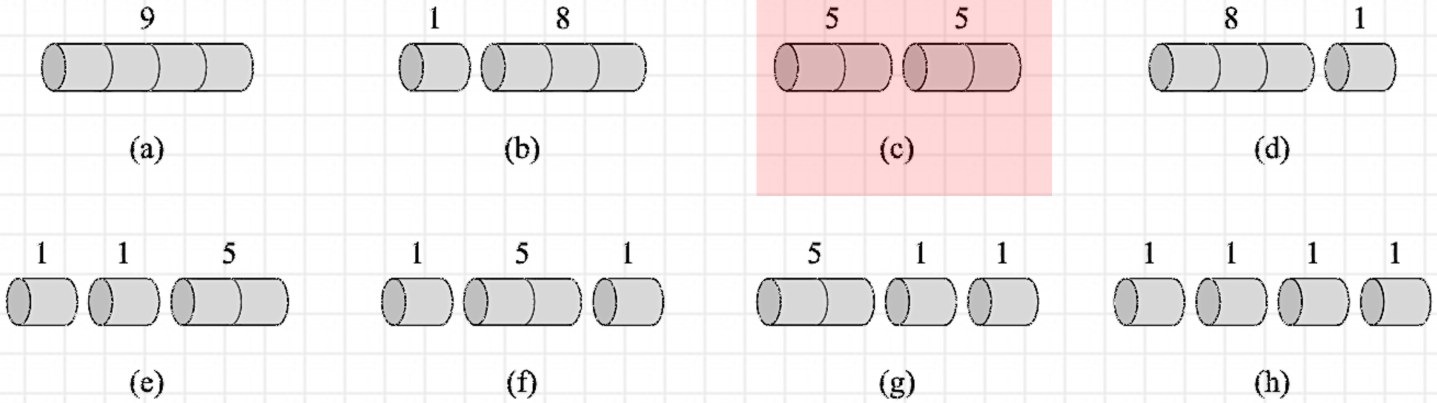
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

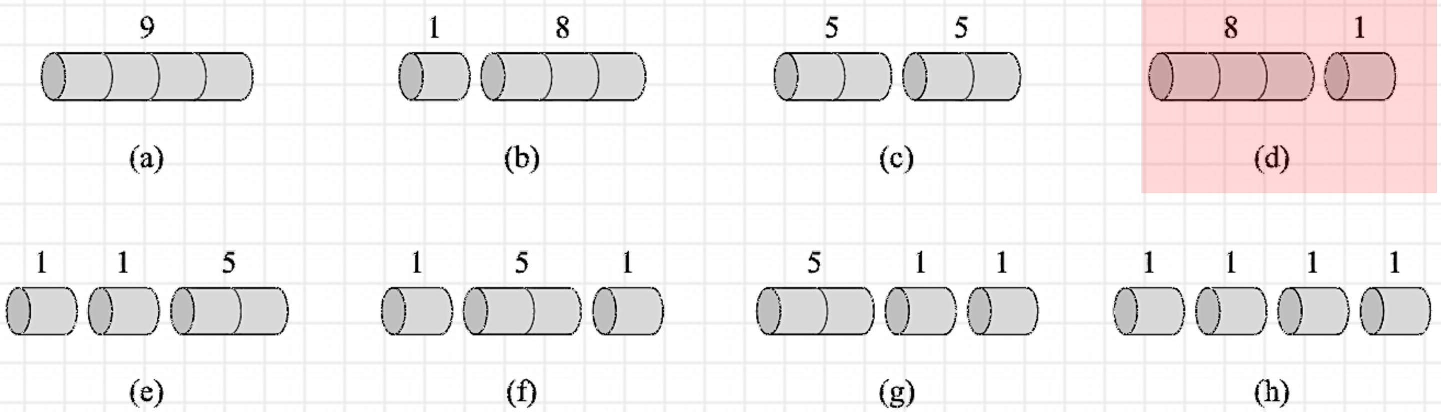
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

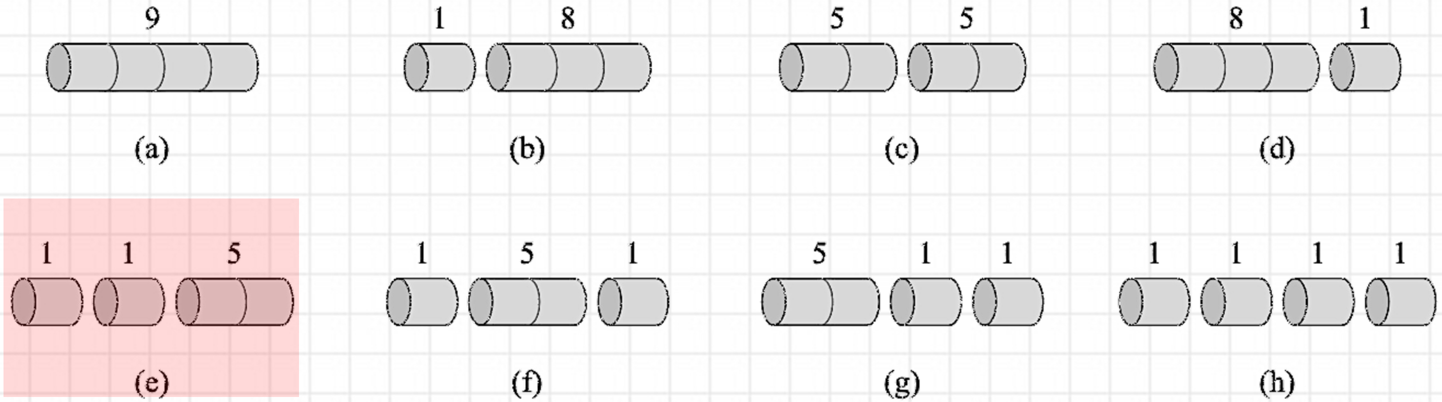
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

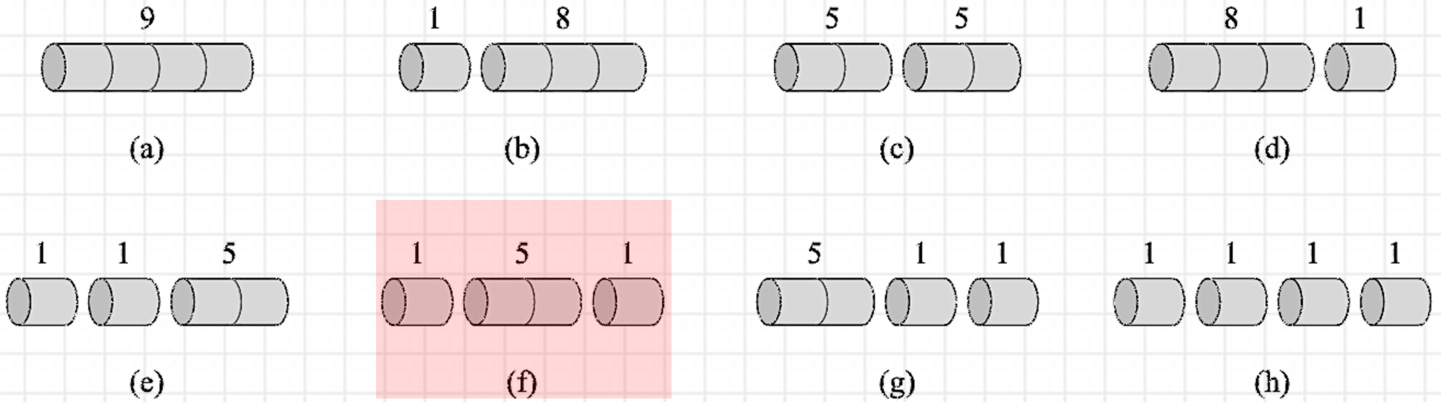
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

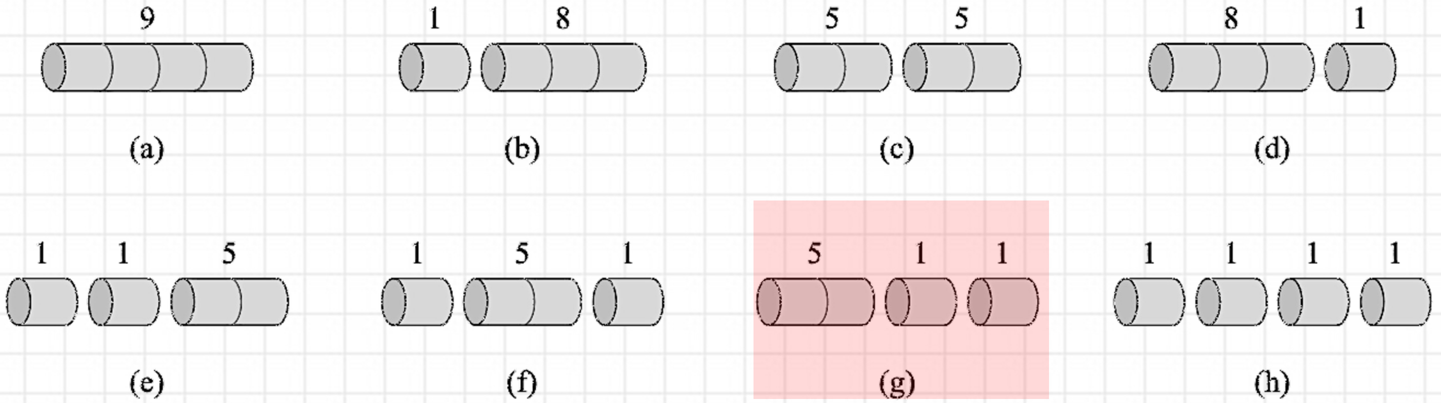
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



(a) (b) (c) (d)
(e) (f) (g) (h)

DP Example: (3) Rod-cutting

- Problem:

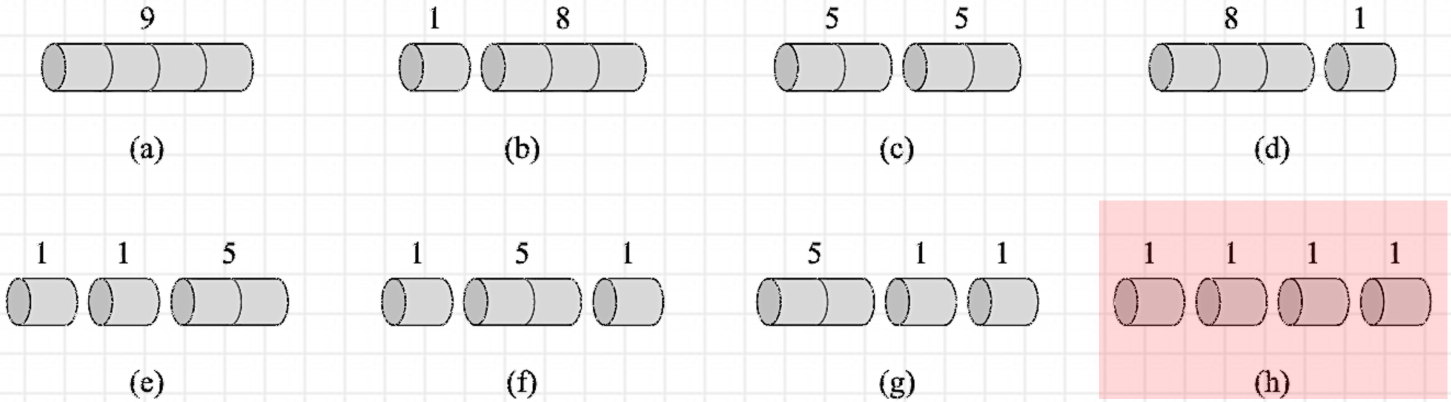
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

  Consider n=4

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



DP Example: (3) Rod-cutting

- Problem:

  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

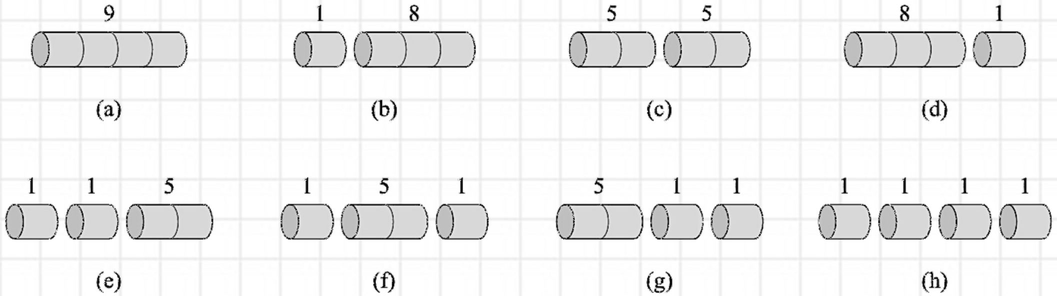
  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

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

- Example:

  Consider n=4

  How many ways to cut up a rod of length n?

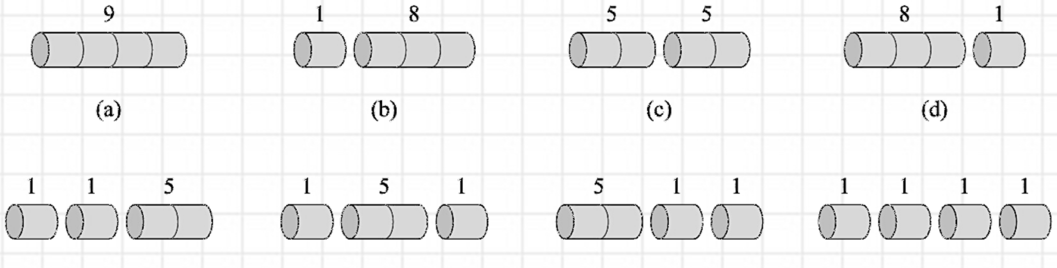


- At each integer distance i inches from the left end, we have an independent option of "cutting" or "not cutting", for i =1,…, n-1: 2^{n-1}

- Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is maximized.

DP Example: (3) Rod-cutting

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



(a) (b) (c) (d)
(e) (f) (g) (h)

- Example:
  - How many ways to cut up a rod of length n? 2^{n-1}
  - Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.

$$n = 0 \Rightarrow r_0 = 0$$

$$n = 1 \Rightarrow r_1 = \boxed{\overbrace{p_1}^{\text{no cut}}}$$

$$n = 2 \Rightarrow r_2 = \max\left(\boxed{\overbrace{p_2}^{\text{no cut}}}, \boxed{\overbrace{p_1}^{\text{cut @ i=1}} + \overbrace{r_1}^{\text{max revenue from n}-1}}\right)$$

$$n = 3 \Rightarrow r_3 = \max\left(\boxed{\overbrace{p_3}^{\text{no cut}}}, \boxed{\overbrace{p_2}^{\text{cut @ i=2}} + \overbrace{r_1}^{\text{max revenue from n}-2}}, \boxed{\overbrace{p_1}^{\text{cut @ i=1}} + \overbrace{r_2}^{\text{max revenue from n}-1}}\right)$$

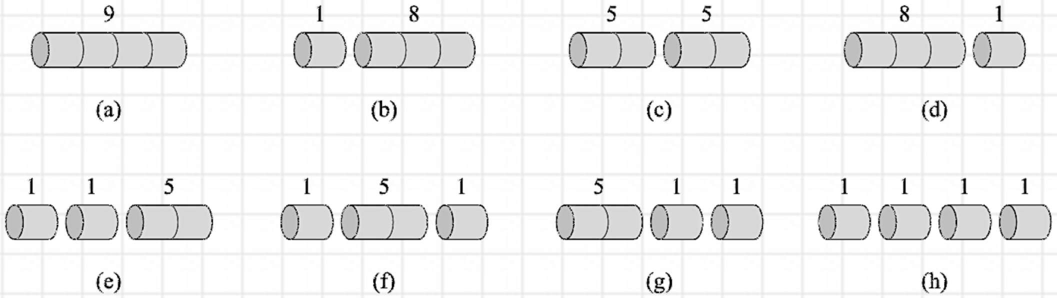
$$n = 4 \Rightarrow r_4 = \max\left(\boxed{p_4}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$$

…

DP Example: (3) Rod-cutting

- 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



- How many ways to cut up a rod of length n? 2^{n-1}
- Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.

$n = 0 \Rightarrow r_0 = 0$

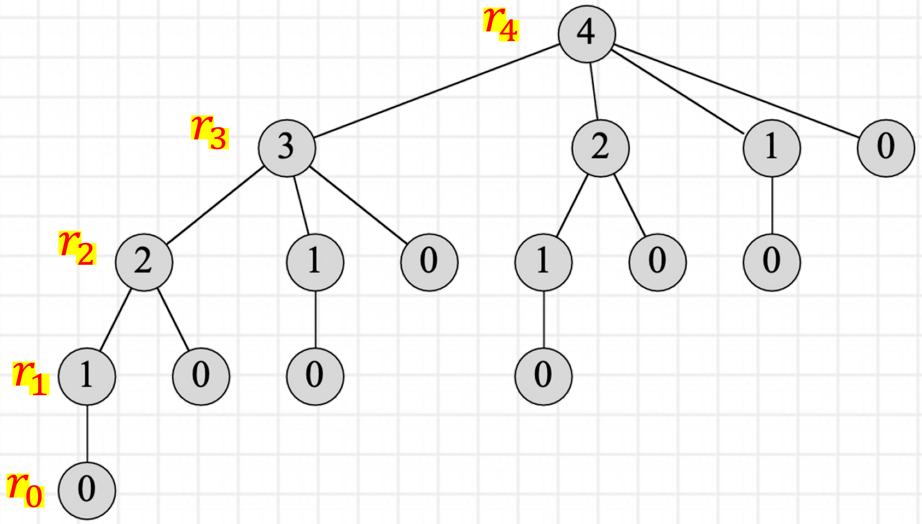
$n = 1 \Rightarrow r_1 = \boxed{p_1}$

$n = 2 \Rightarrow r_2 = \max\left(\boxed{p_2}, \boxed{p_1 + r_1}\right)$

$n = 3 \Rightarrow r_3 = \max\left(\boxed{p_3}, \boxed{p_2 + r_1}, \boxed{p_1 + r_2}\right)$

$n = 4 \Rightarrow r_4 = \max\left(\boxed{p_4}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$

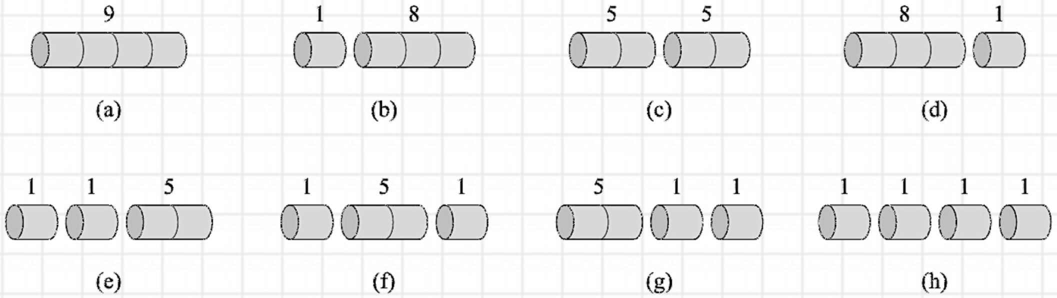
…



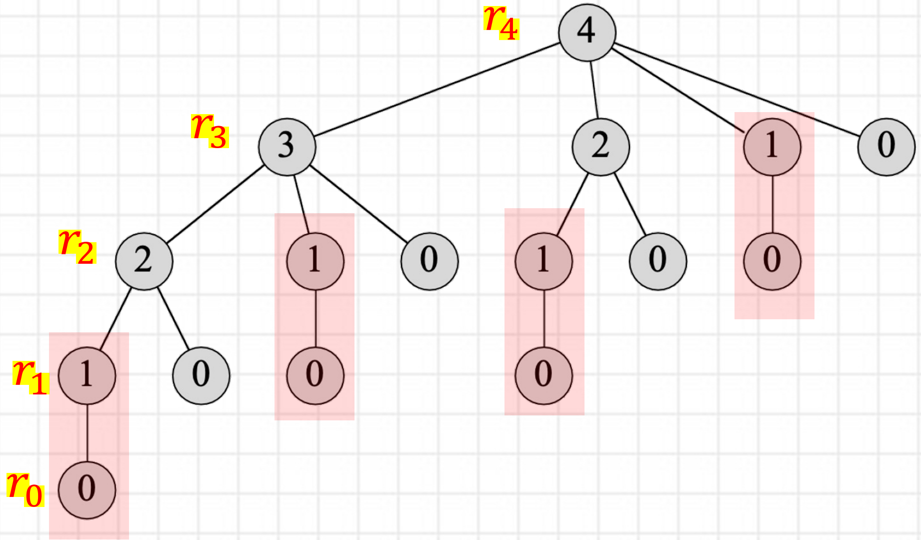
DP Example: (3) Rod-cutting

- Example:
  - How many ways to cut up a rod of length n? 2^{n-1}
  - Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.

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



$n = 0 \Rightarrow r_0 = 0$
$n = 1 \Rightarrow r_1 = \boxed{p_1}$
$n = 2 \Rightarrow r_2 = \max\left(\boxed{p_2}, \boxed{p_1 + r_1}\right)$
$n = 3 \Rightarrow r_3 = \max\left(\boxed{p_3}, \boxed{p_2 + r_1}, \boxed{p_1 + r_2}\right)$
$n = 4 \Rightarrow r_4 = \max\left(\boxed{p_4}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$
…

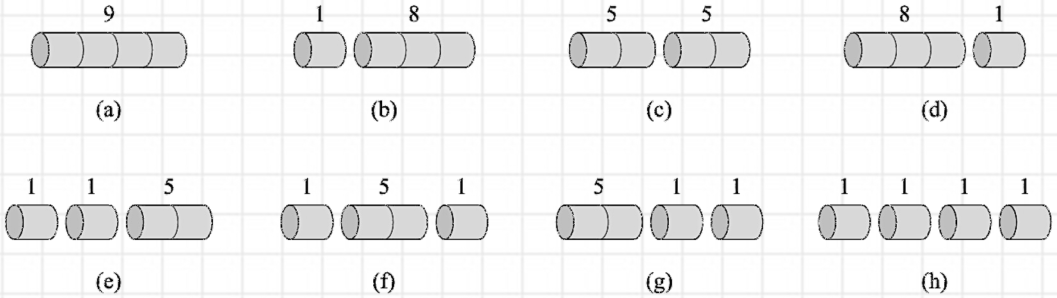


DP Example: (3) Rod-cutting

- Example:

- How many ways to cut up a rod of length n? 2^{n-1}
- Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.

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



$n = 0 \implies r_0 = 0$

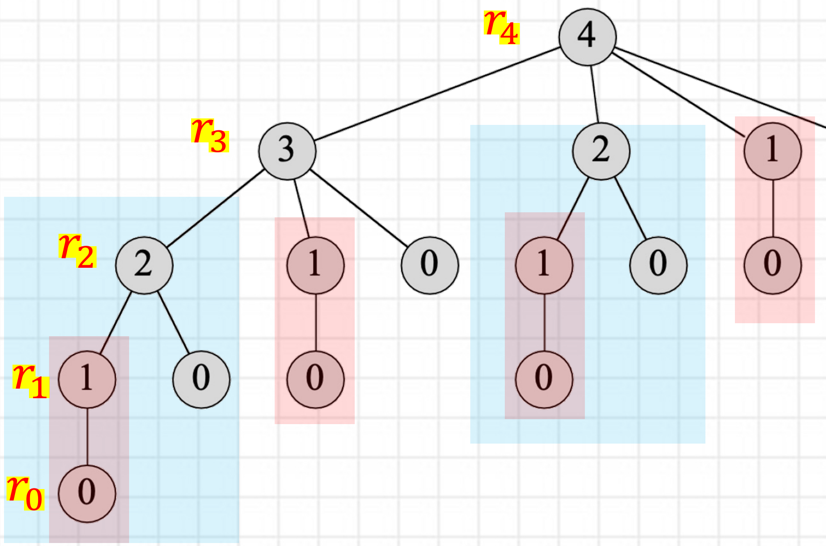
$n = 1 \implies r_1 = \boxed{p_1}$

$n = 2 \implies r_2 = \max\left(\boxed{p_2}, \boxed{p_1 + r_1}\right)$

$n = 3 \implies r_3 = \max\left(\boxed{p_3}, \boxed{p_2 + r_1}, \boxed{p_1 + r_2}\right)$

$n = 4 \implies r_4 = \max\left(\boxed{p_4}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$

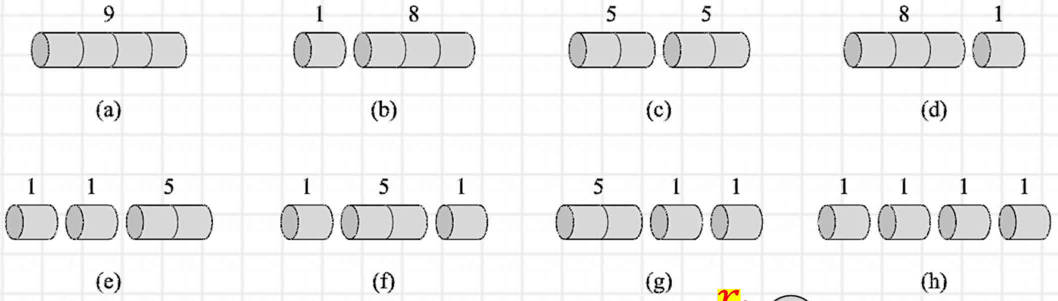
…



DP Example: (3) Rod-cutting

- 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



- How many ways to cut up a rod of length n? 2^{n-1}
- Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.

$n = 0 \Rightarrow r_0 = 0$

$n = 1 \Rightarrow r_1 = \boxed{p_1 + r_0}$

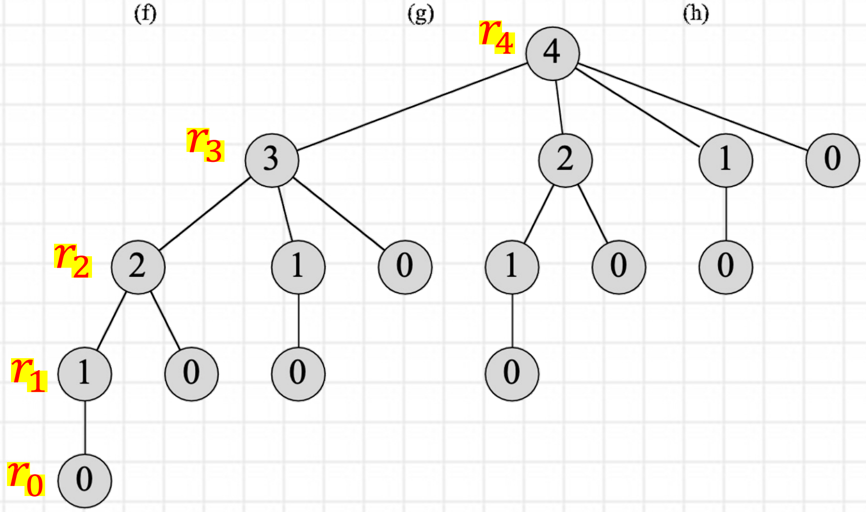
$n = 2 \Rightarrow r_2 = \max\left(\boxed{p_2 + r_0}, \boxed{p_1 + r_1}\right)$

$n = 3 \Rightarrow r_3 = \max\left(\boxed{p_3 + r_0}, \boxed{p_2 + r_1}, \boxed{p_1 + r_2}\right)$

$n = 4 \Rightarrow r_4 = \max\left(\boxed{p_4 + r_0}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$

…

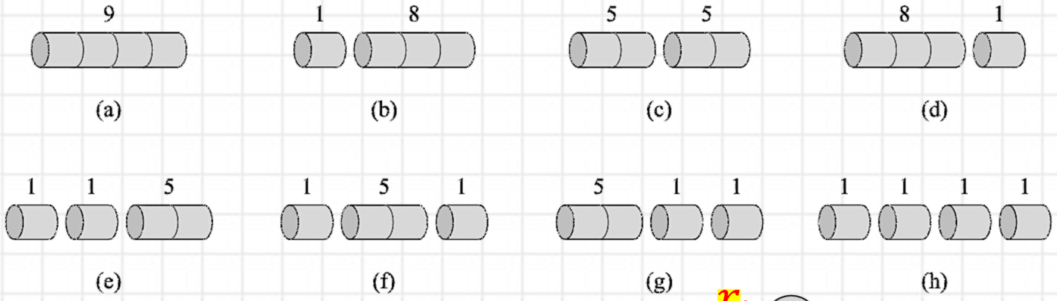
$n \quad \Rightarrow r_n = \max\left(\boxed{p_n + r_0}, \boxed{p_{n-1} + r_1}, \boxed{p_{n-2} + r_2}, \dots, \boxed{p_1 + r_{n-1}}\right) = \max_{1 \le i \le n}\left(\boxed{p_i + r_{n-i}}\right)$

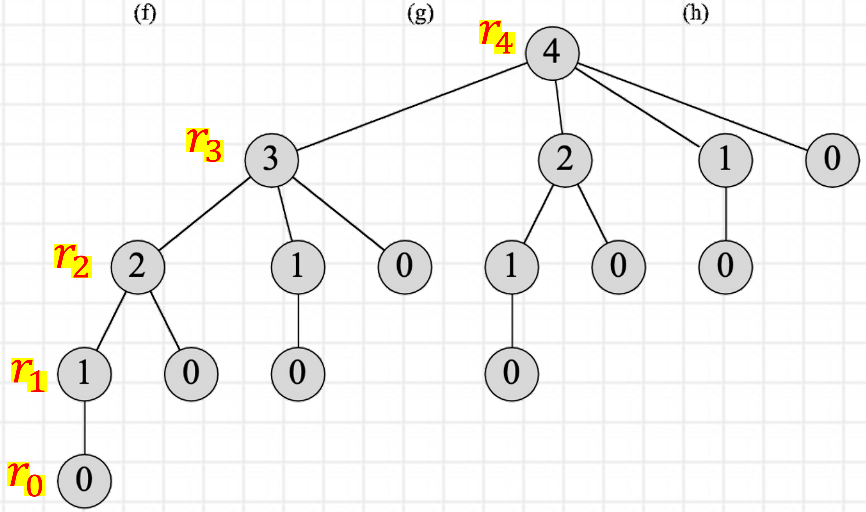


DP Example: (3) Rod-cutting

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

- Example:
  - How many ways to cut up a rod of length n? 2^{n-1}
  - Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.





$$n = 0 \implies r_0 = 0$$
$$n = 1 \implies r_1 = \boxed{p_1 + r_0}$$
$$n = 2 \implies r_2 = \max\left(\boxed{p_2 + r_0}, \boxed{p_1 + r_1}\right)$$
$$n = 3 \implies r_3 = \max\left(\boxed{p_3 + r_0}, \boxed{p_2 + r_1}, \boxed{p_1 + r_2}\right)$$
$$n = 4 \implies r_4 = \max\left(\boxed{p_4 + r_0}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$$

…

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

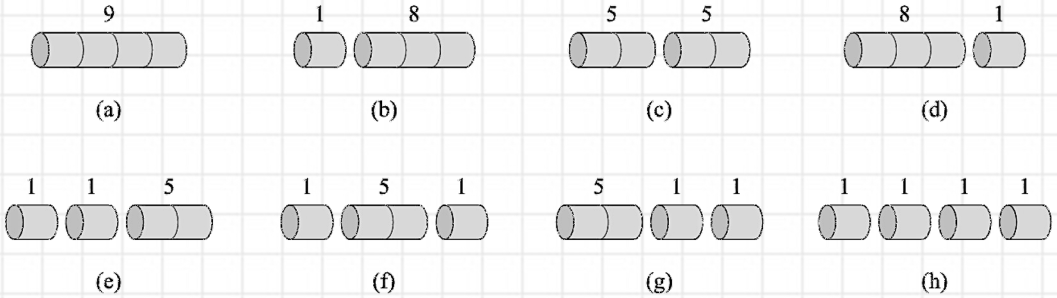
Recurrence relation $\implies$ Recursive algorithm

DP Example: (3) Rod-cutting

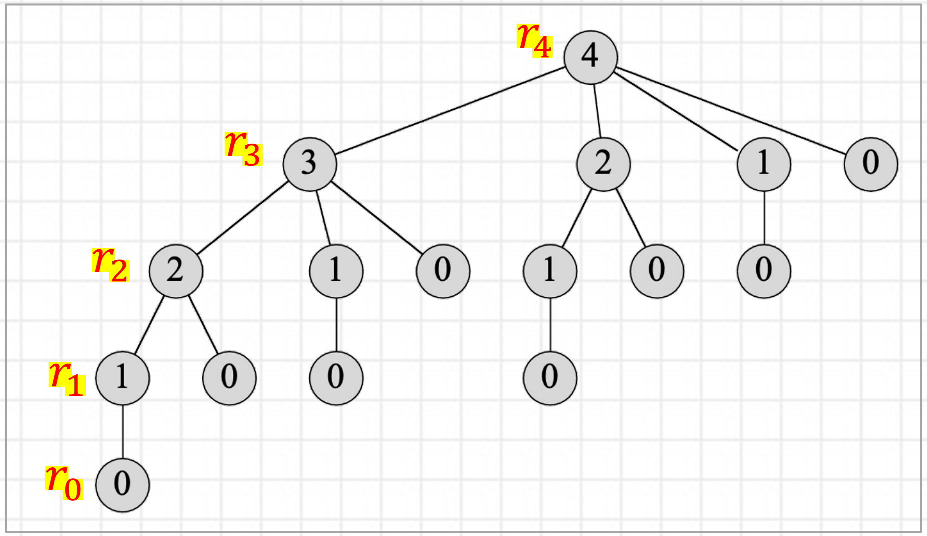
- Rod of length n
- How many ways to cut up a rod of length n? 2^{n-1}
- Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.
- Recurrence relation: $r_n = \max\limits_{1 \le i \le n} (p_i + r_{n-i})$
- Base case: $r_0 = 0$
- Recursive (brute force) algorithm

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
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

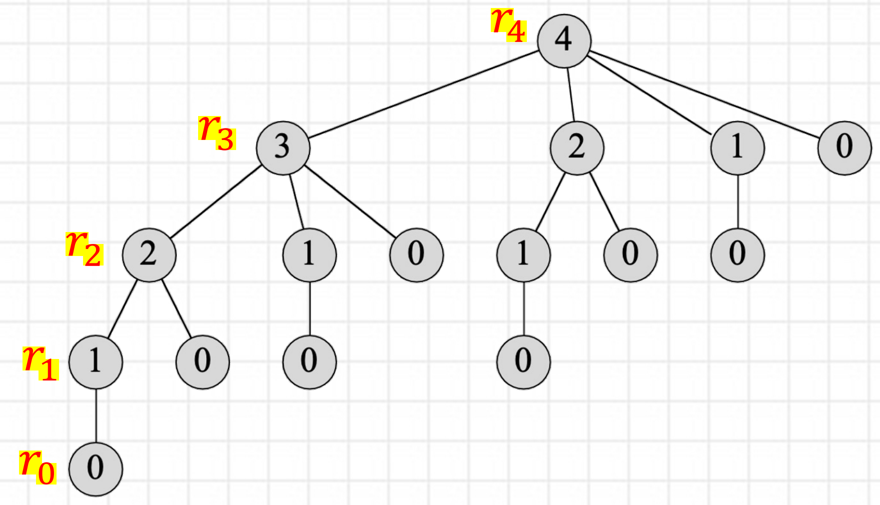


Running time?



DP Example: (3) Rod-cutting

- Rod of length n
  - How many ways to cut up a rod of length n? 2^{n-1}
  - Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.
  - Recurrence relation: $r_n = \max\limits_{1 \le i \le n} (p_i + r_{n-i}), r_0 = 0$
  - Recursive (brute force) algorithm



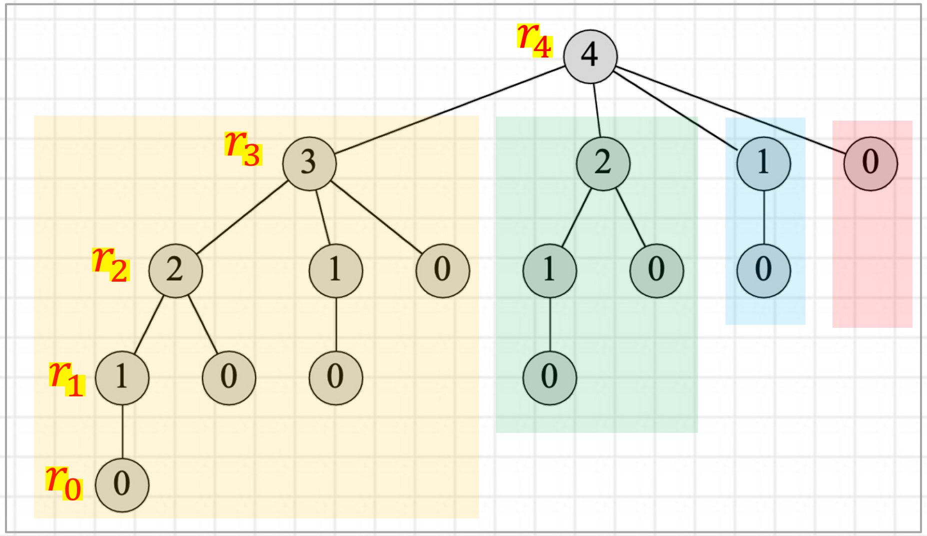

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Running time?

- $T(n)$= number of [recursive] calls to Cut-Rod function
- $T(n)$= number nodes in the subtree of r_n in the recursion tree

DP Example: (3) Rod-cutting

- Rod of length n
  - How many ways to cut up a rod of length n? 2^{n-1} = # of leaves
  - Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.
  - Recurrence relation: $r_n = \max\limits_{1 \le i \le n}(p_i + r_{n-i}), r_0 = 0$
  - Recursive (brute force) algorithm




```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

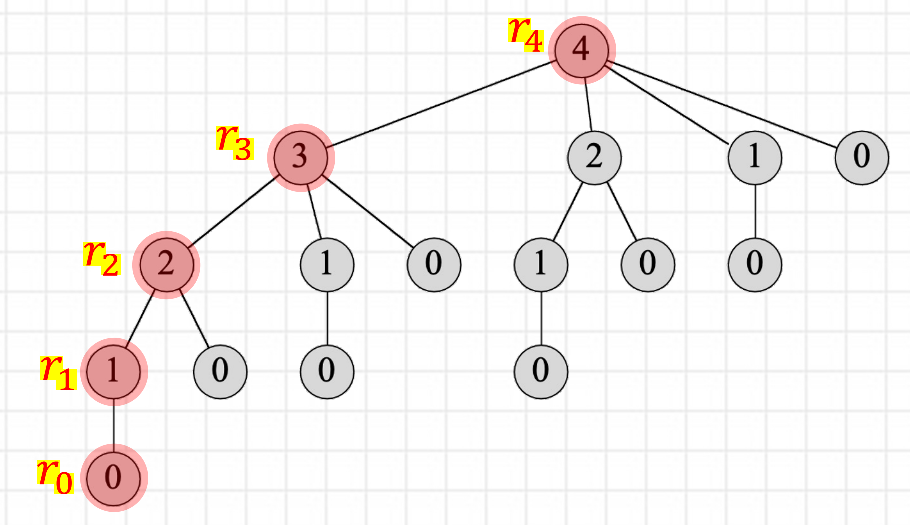
Running time?

- $T(n)$= number of [recursive] calls to Cut-Rod function
- $T(n)$= number nodes in the recursion tree
- $T(n)$= **1 + 1 + 2 + 4 + 8 + …**

  $T(n)=1 + \sum_{i=0}^{n-1} T(i)$ =**1 +** $\frac{\mathbf{2^n-1}}{\mathbf{2-1}}$ **=** $\mathbf{2^n}$
- $T(n) \in \Theta(2^n)$ Exponential (the same subproblems solved repeatedly)

DP Example: (3) Rod-cutting

- DP solution
  - Recurrence relation: $r_n = \max\limits_{1 \le i \le n} (p_i + r_{n-i})$,
  - Base case: $r_0 = 0$




```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Top-down (recursive with memoization)

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

```
BOTTOM-UP-CUT-ROD(p, n)
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]
```

Bottom-up (iterative)

Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer

Divide-and-Conquer:

- Divide problem into subproblems
- Recursively solve the subproblems and aggregate solutions

Note:
The subproblems do not overlap

Dynamic Programming

- Divide problem into subproblems, recursively solve them
- Subproblems overlap
- When a subproblem has been solved, remember its solution and reuse that solution rather than resolving it later (memoization)

Dynamic Programming (DP)

- Dynamic Programming Elements
  - DP often applicable to optimization problems
    - Large number of possible solutions
    - Must find the “best” one (maximum or minimum)
  - Problem possesses an “optimal substructure”
    - Finding the optimal solution involves finding the optimal solution to subproblems
    - The subproblems are the same as the original problem, but are “smaller” (e.g., involve smaller-sized input data) Similar to D&C
  - Subproblems overlap Key difference to D&C
    - Different subproblems operate on the same input data
    - Allows exploitation of memoization

Dynamic Programming (DP)

- Dynamic Programming Recipe
  1. Show the problem has optimal substructure, i.e., the optimal solution can be constructed from optimal solutions to subproblems (This step is concluded by writing the recurrence relation and its base case).
  2. Show subproblems are overlapping, i.e., subproblems may be encountered many times but the total number of distinct subproblems is polynomial (Recall the recursion tree for Fibonacci and Rod-cutting problems, where the total number of distinct subproblems was linear, i.e., O(n)).
  3. Construct an algorithm that computes the optimal solution to each subproblem only once and reuses the stored result all other times (This can be done by using either top-down (recursive) or bottom-up (iterative) approach).
  4. Analysis: show that time and space complexity is polynomial.

References

- The lecture slides are heavily based on the suggested textbooks and the corresponding published lecture notes:

  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
  - DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.
  - Slides by Kevin Wayne. Copyright © 2005 Pearson-Addison Wesley.
  - Slides by Elizabeth Cherry, Georgia Institute of Technology.