CS-3510: Design and Analysis of Algorithms

Dynamic Programming II

Instructor: Shahrokh Shahi

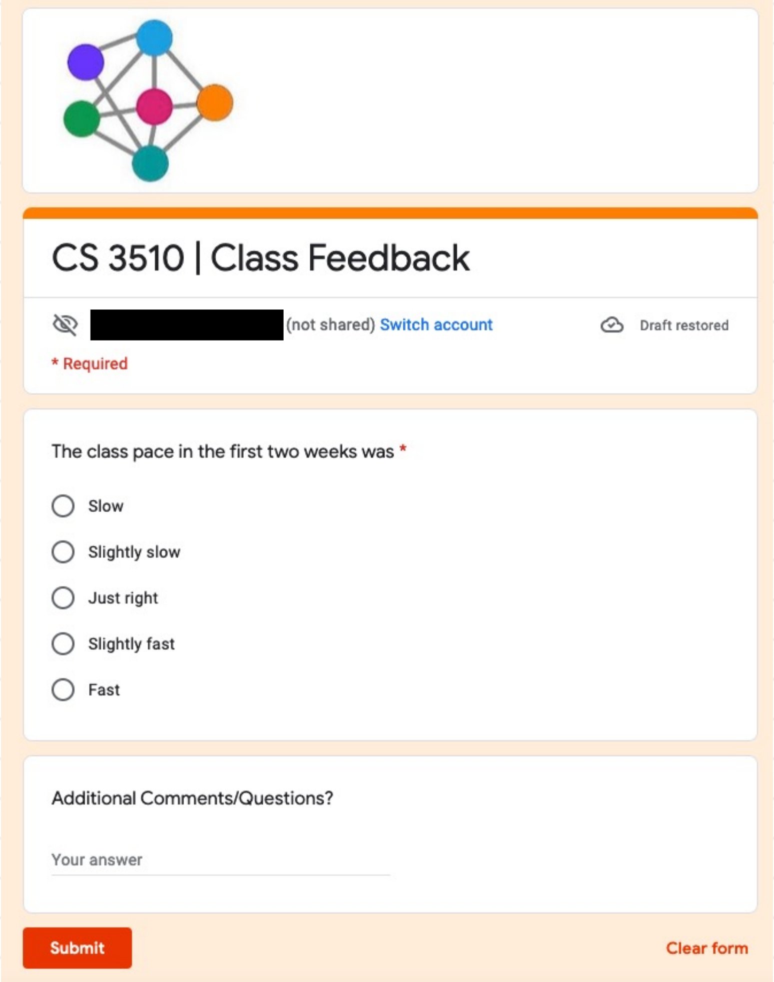
College of Computing
Georgia Institute of Technology
Summer 2022

Announcements (1/2)

- HW2 is released; due this Friday June 3, 2022.
- Exam 1 next week, Thursday June 9, 2022.
- Exam 1:
  - Asymptotic notations and complexity
  - Divide-and-Conquer
  - Dynamic Programming
- Practice problems
  - Will be published on Thursday
  - Review for Exam 1 on Thursday

Announcements (2/2)

- Lecture feedback
  - https://forms.gle/hAJVaM44Ch2uPqBPA



CS 3510 | Class Feedback

[redacted] (not shared) Switch account

Draft restored

* Required

The class pace in the first two weeks was *

- Slow
- Slightly slow
- Just right
- Slightly fast
- Fast

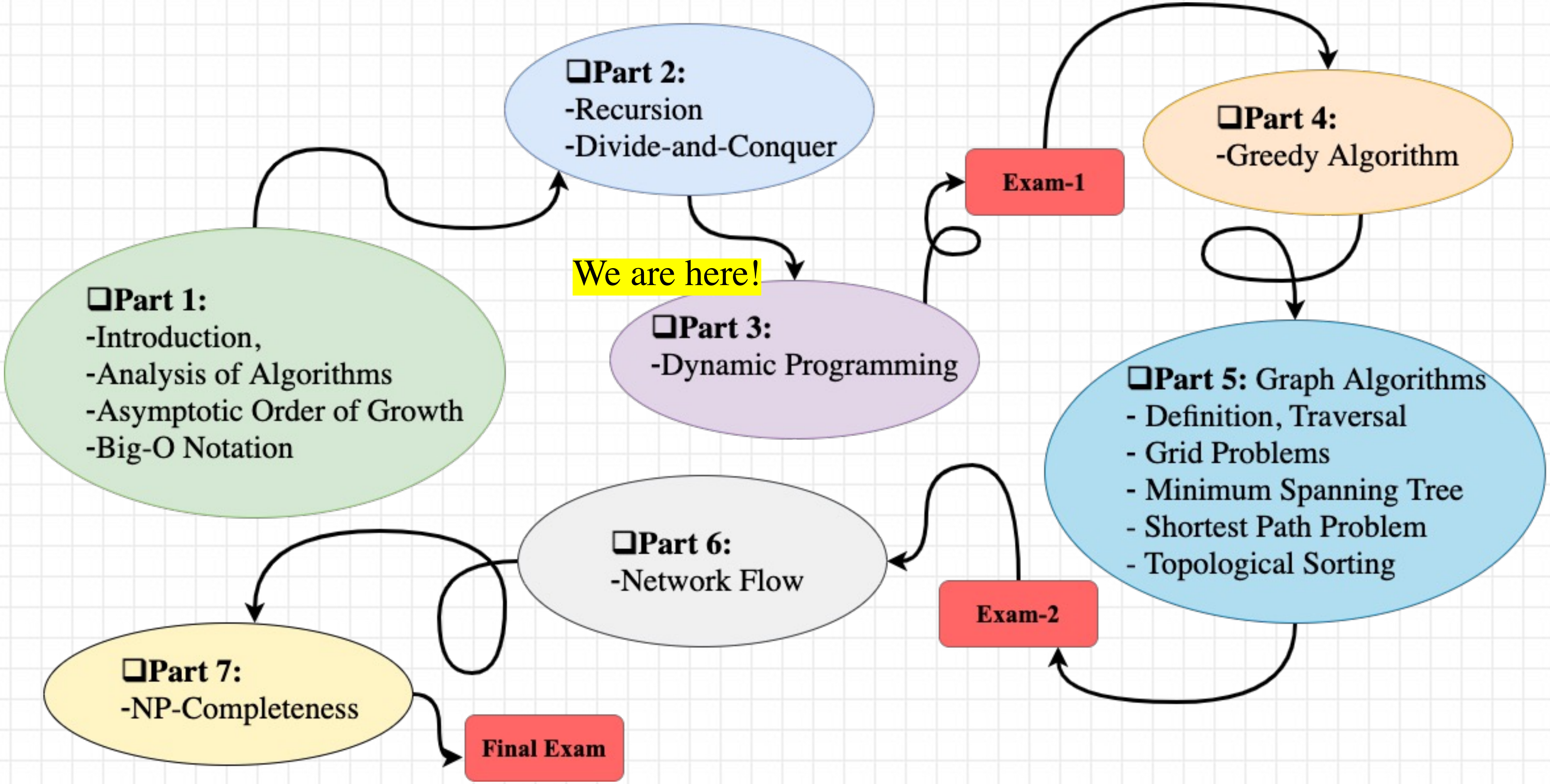
Additional Comments/Questions?

Your answer

Submit

Clear form

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)

- Dynamic Programming vs. Divide-and-Conquer

Divide-and-Conquer:

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

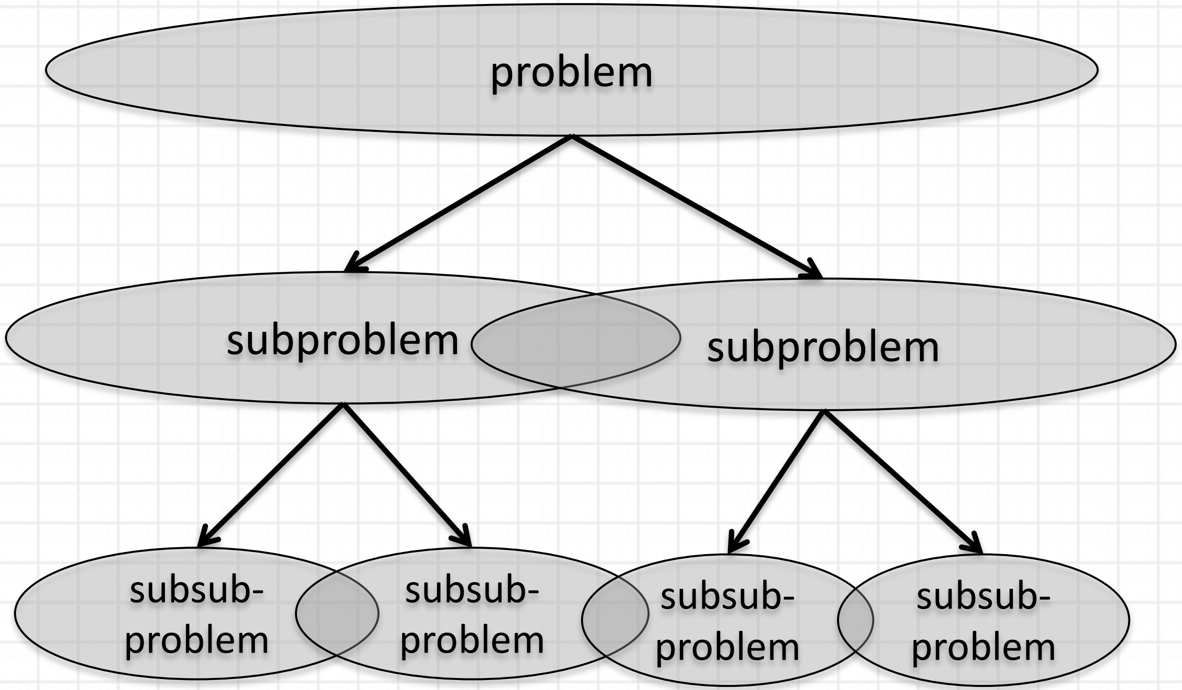
Note: The subproblems do not overlap

Dynamic Programming

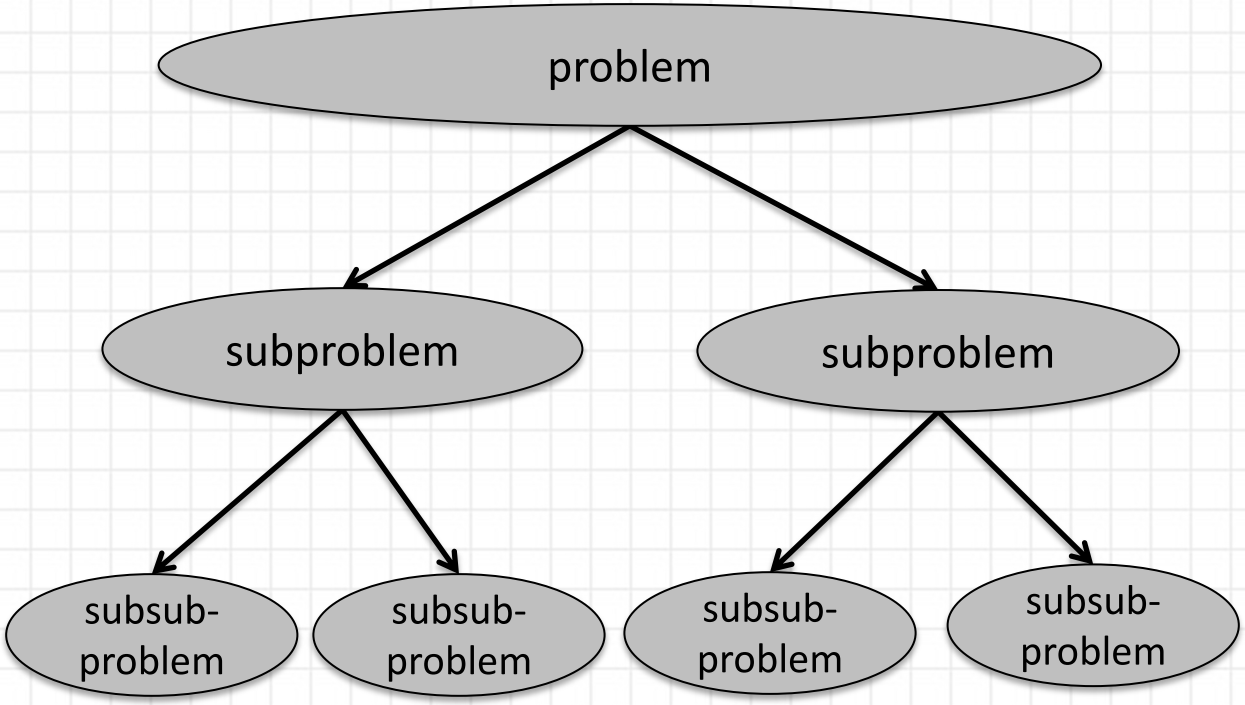
- Divide problem into subproblems, recursively solve them
- Subproblems overlap
- When a subproblem has been solved, remember its solution and reuse that solution rather than resolving it later (memoization)

Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer



Subproblems overlap

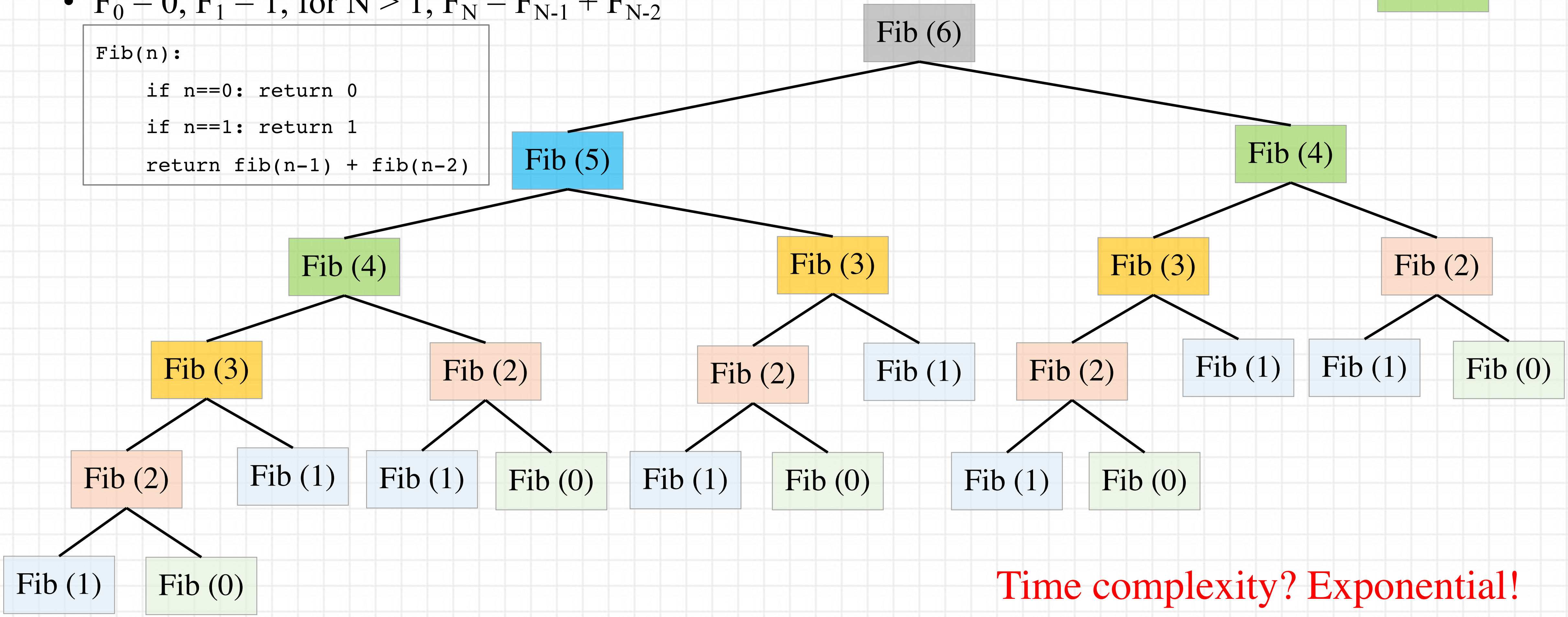


Subproblems do not overlap

DP Example: (1) Fibonacci

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

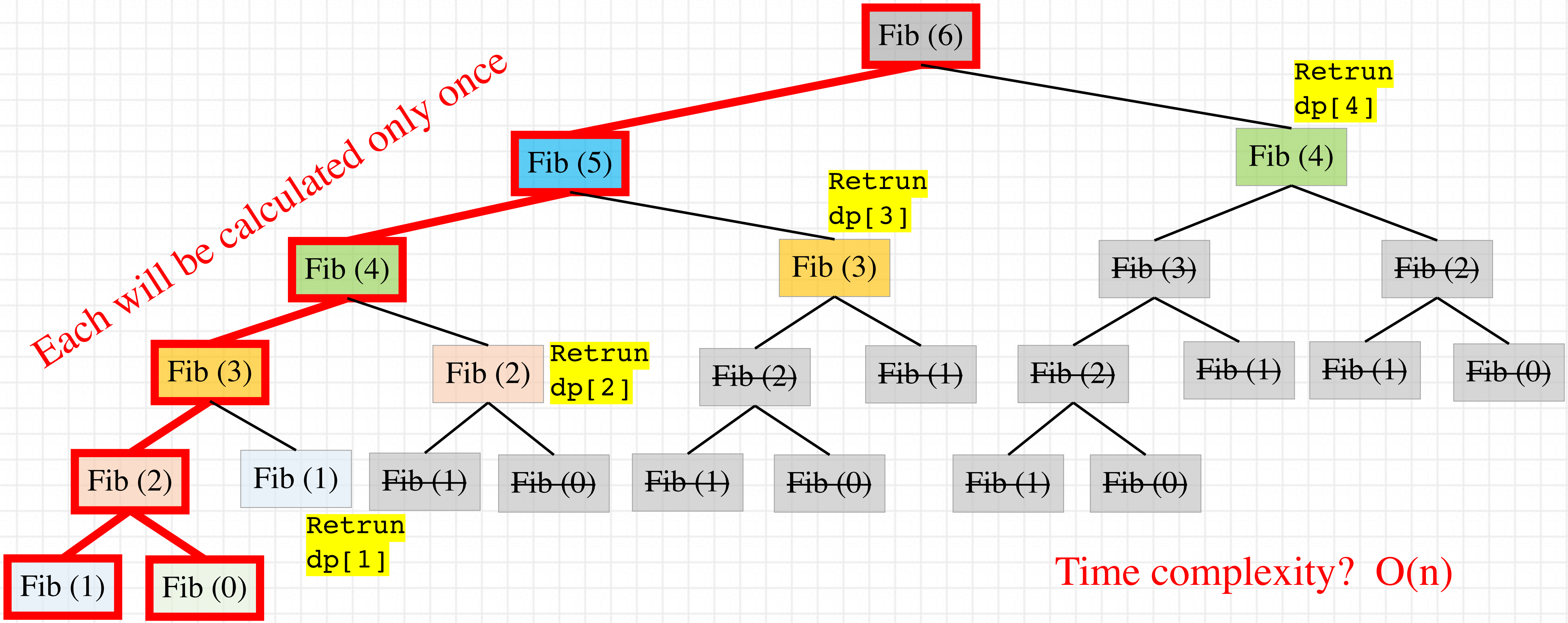
```
Fib(n):
    if n==0: return 0
    if n==1: return 1
    return fib(n-1) + fib(n-2)
```



Fib (2)	X 5
Fib (3)	X 3
Fib (4)	X 2

Time complexity? Exponential!

DP Example: (1) Fibonacci



Time complexity? O(n)

Dynamic Programming

- Top-down vs. Bottom-up Approach

  - "Top-down" dynamic programming
    - Begin with problem description
    - i.e., begin at root of tree and work downwards
    - Recursively subdivide problem into subproblems

  Recursive with memoization

  - "Bottom-up" dynamic programming
    - Start at the leaf nodes of tree, i.e., the base case(s).
    - Build up solution to larger problem from solutions of the simpler subproblems

  Iterative

DP Example: (1) Fibonacci

- So, which one is better?

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: (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)

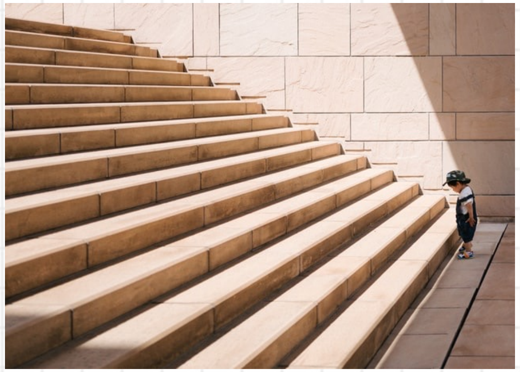
```
Fib(n):
    dp = [0]*n    # initialize dp[i]=0
    dp[0] = 0
    dp[1] = 1
    for i=2,…,n:
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Each computation needs only the last two Fibonacci numbers!

Re-write the code with two scalars.

DP Example: (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:

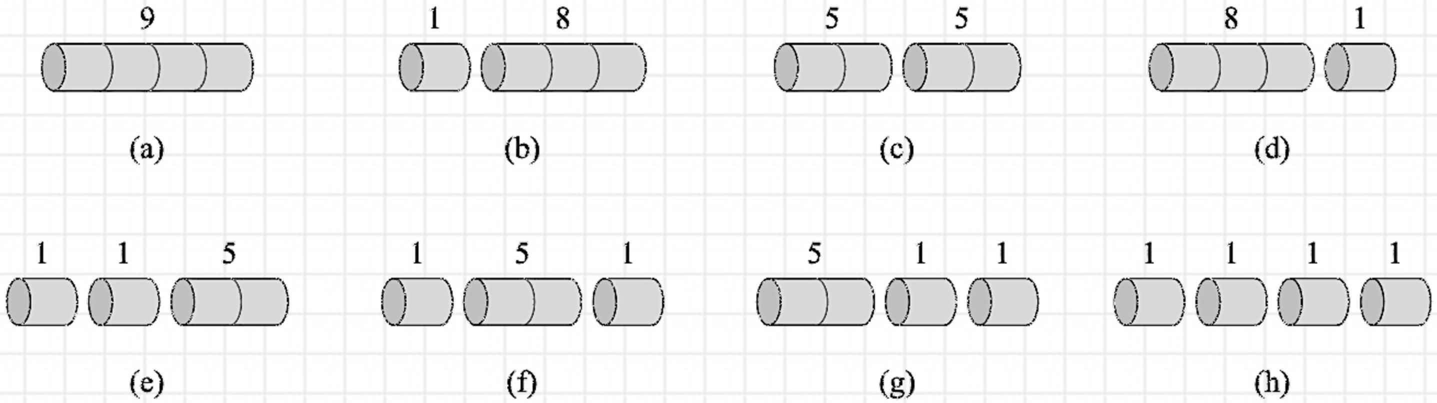
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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:

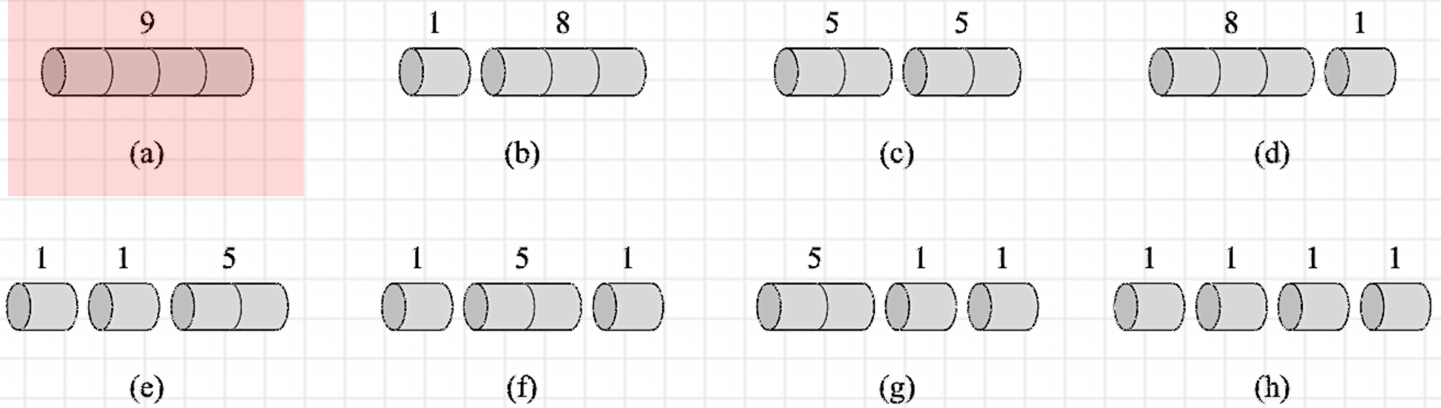
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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:

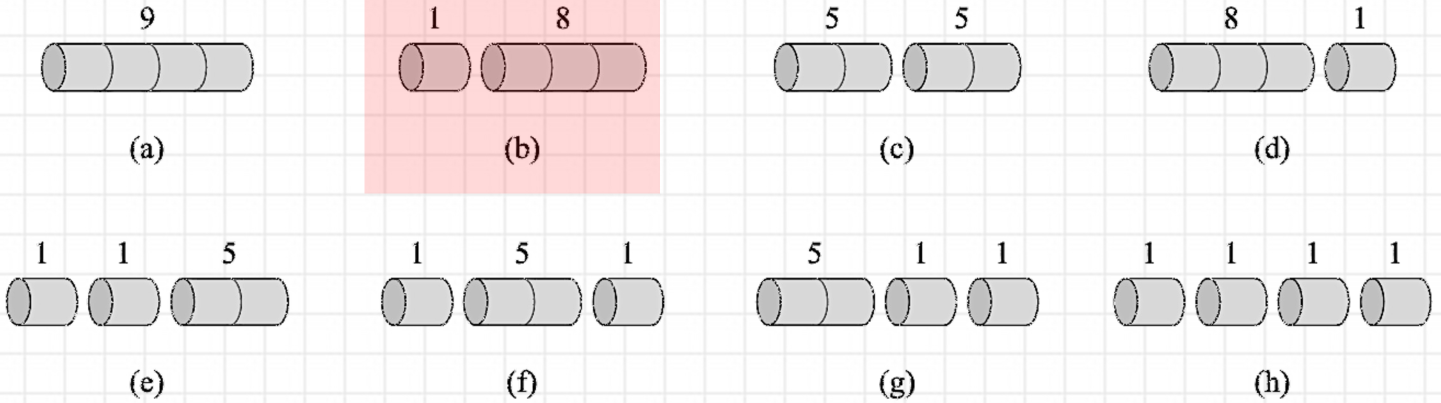
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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:

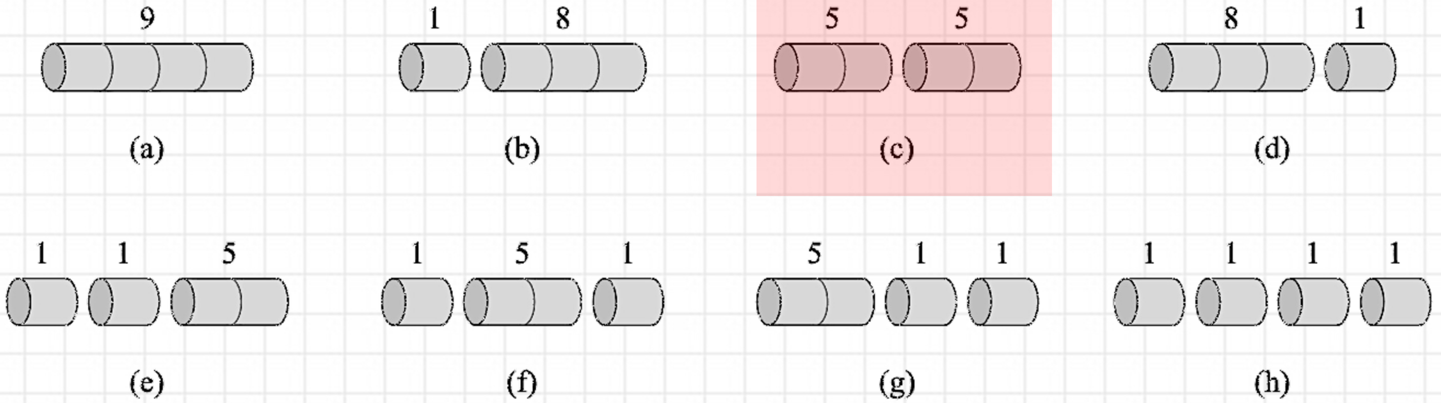
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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:

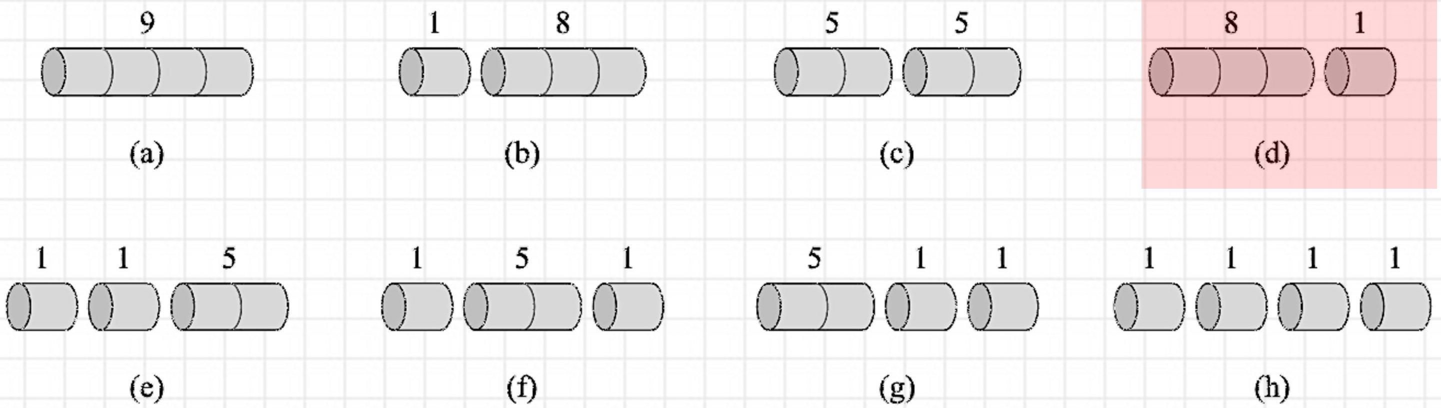
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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:

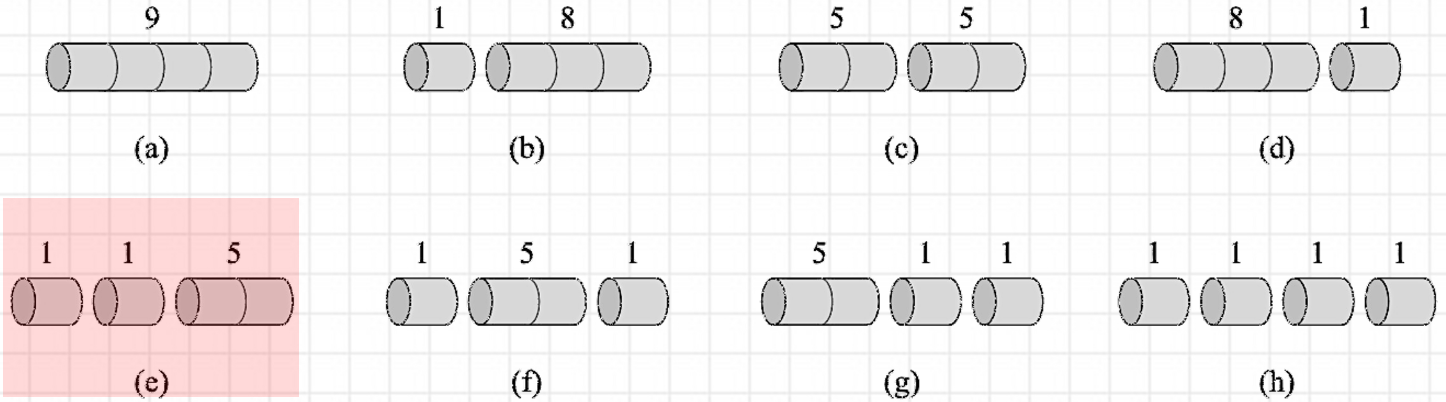
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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:

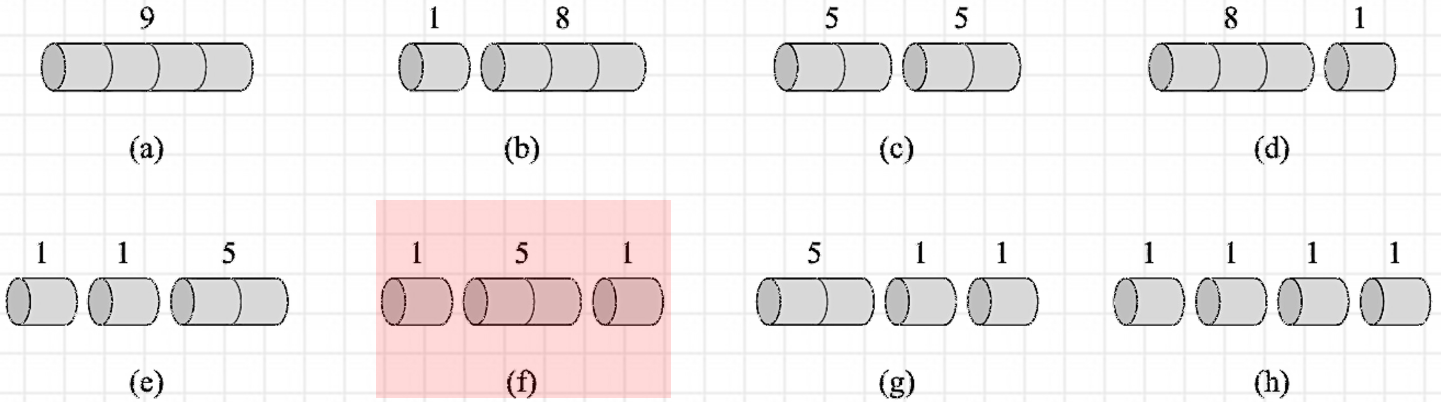
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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:

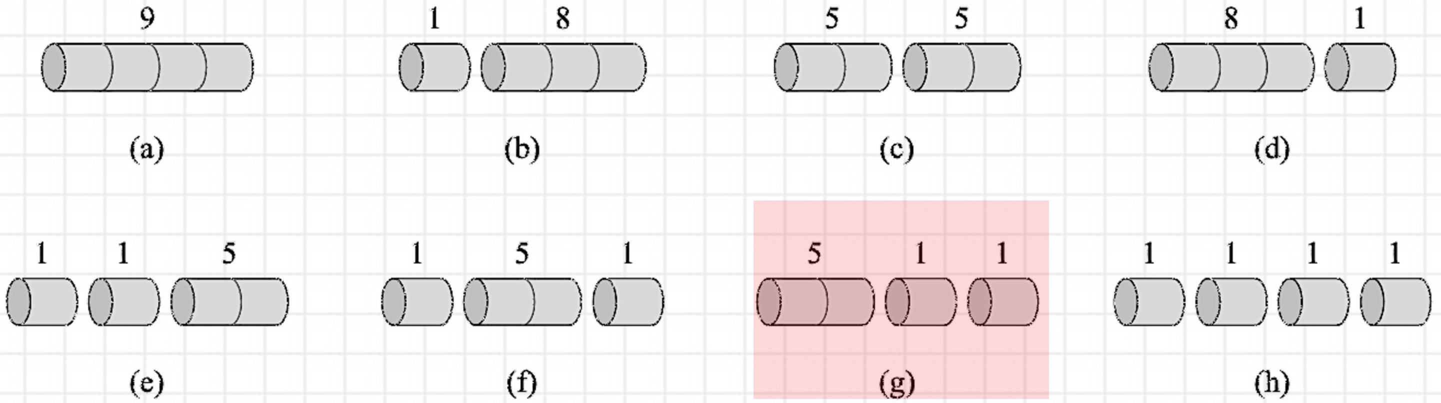
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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



DP Example: (3) Rod-cutting

- Problem:

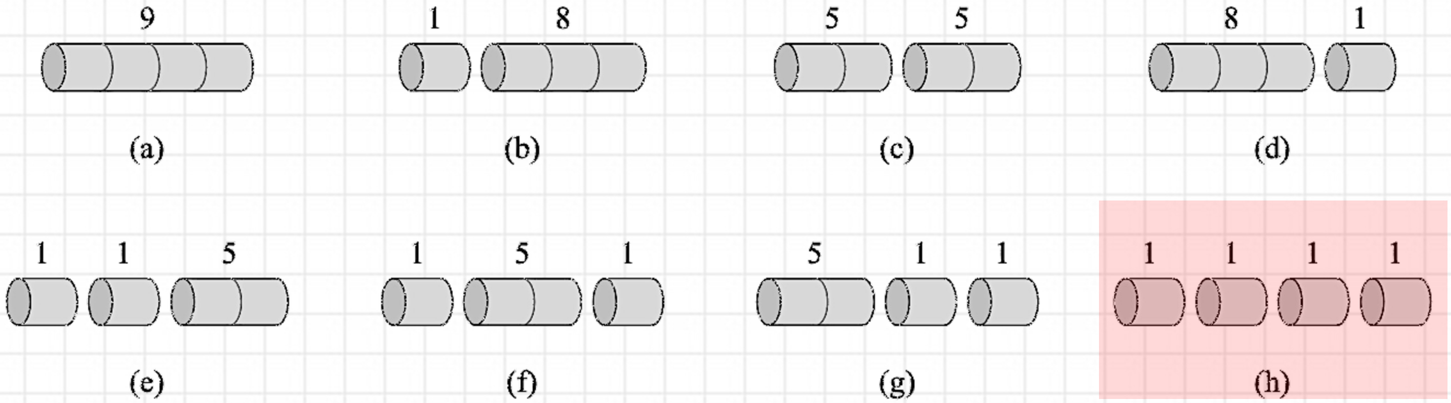
  Given a rod of length n inches and a table of prices p_i for i=1, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

  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.

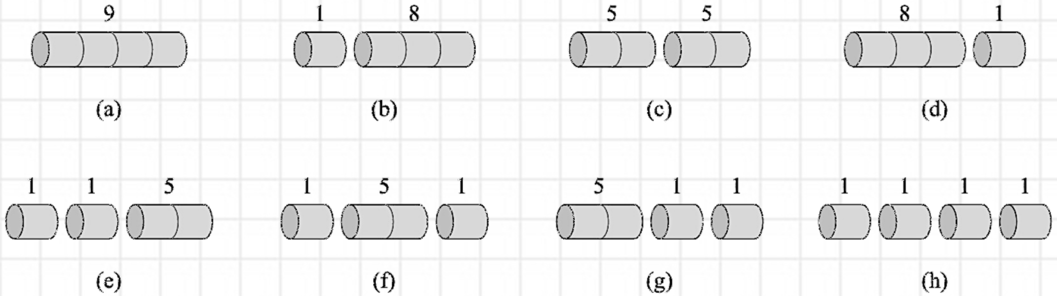
  Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- Example:

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

  Consider n=4

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



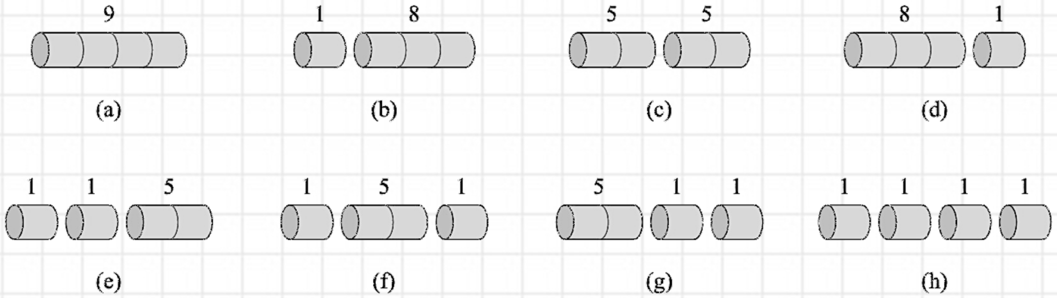
  - At each integer distance i inches from the left end, we have an independent option of "cutting" or "not cutting", for i =1,…, n-1: 2^{n-1}

  - 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

- 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 $1 \leq k \leq n$ 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



$n = 0 \implies r_0 = 0$

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

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

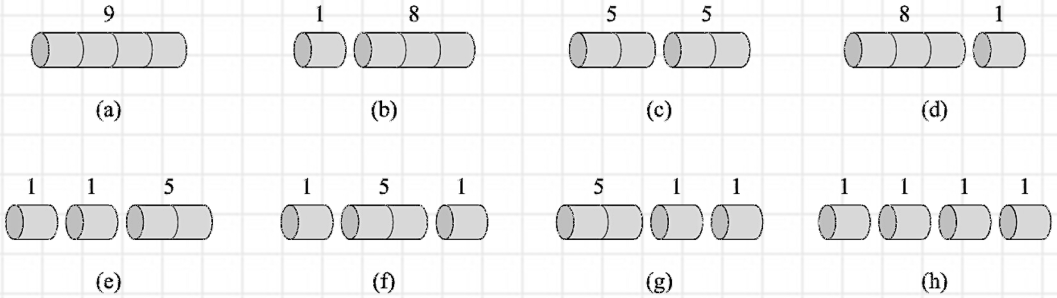
$n = 3 \implies r_3 = \max\left(\boxed{\overbrace{p_3}^{\text{no cut}}}, \boxed{\overbrace{p_2}^{\text{cut @ i=2}} + \overbrace{r_1}^{\text{max revenue from n−2}}}, \boxed{\overbrace{p_1}^{\text{cut @ i=1}} + \overbrace{r_2}^{\text{max revenue from n−1}}}\right)$

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

…

DP Example: (3) Rod-cutting

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.

$n = 0 \Rightarrow r_0 = 0$

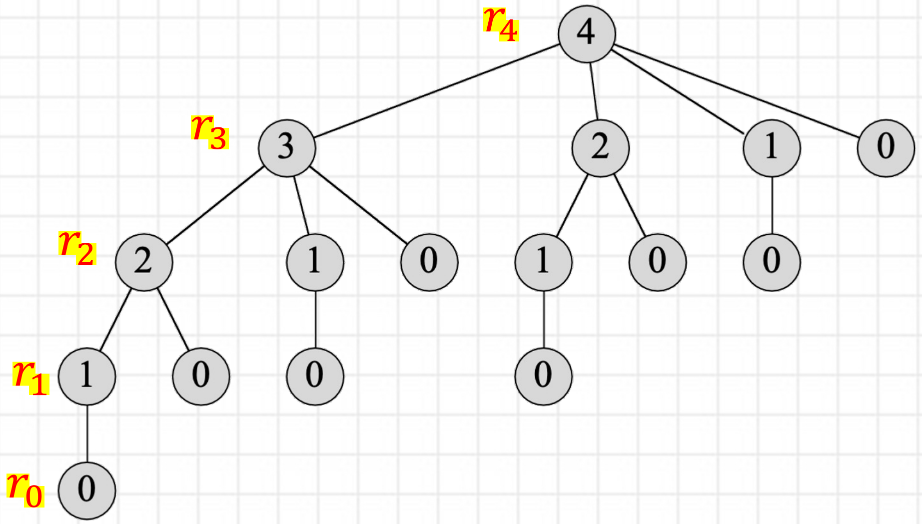
$n = 1 \Rightarrow r_1 = \boxed{p_1}$

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

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

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

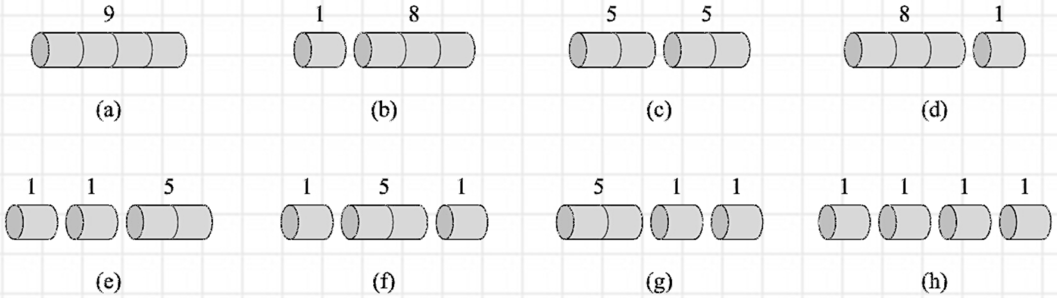
…



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 $1 \leq k \leq n$ 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



$n = 0 \Rightarrow r_0 = 0$

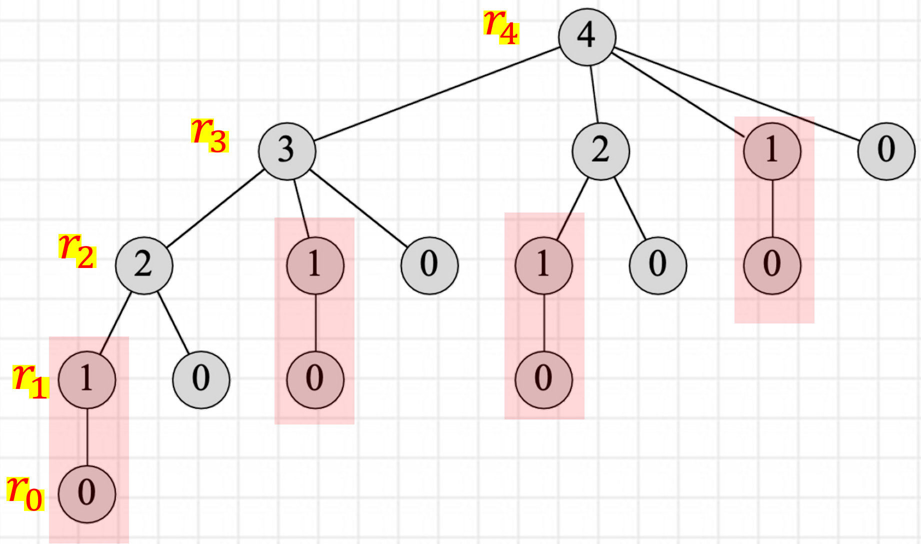
$n = 1 \Rightarrow r_1 = \boxed{p_1}$

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

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

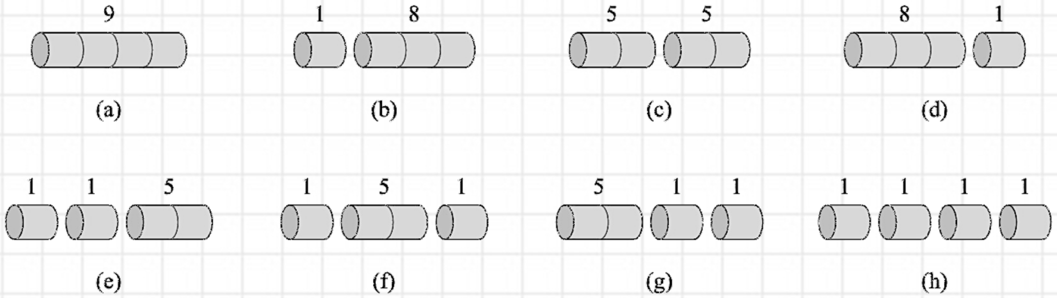
$n = 4 \Rightarrow r_4 = \max\left(\boxed{p_4}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$

…



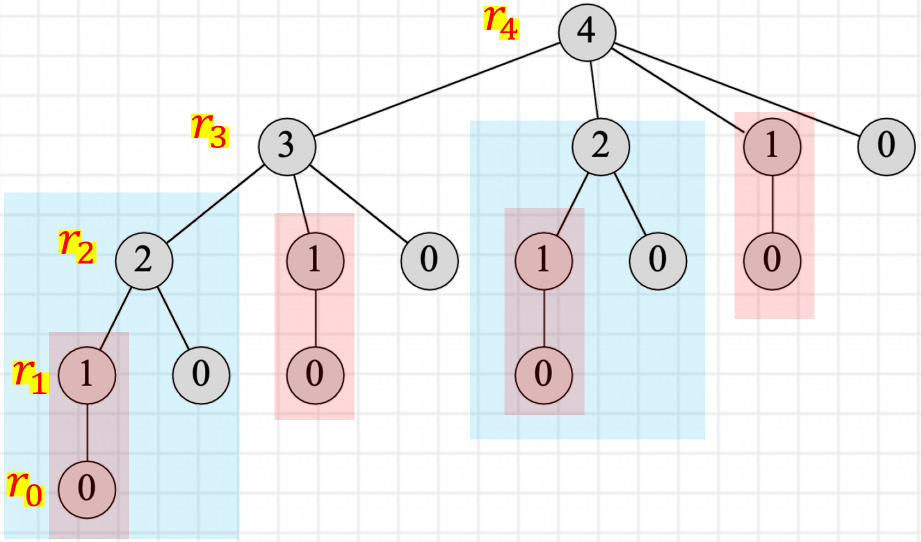
DP Example: (3) Rod-cutting

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



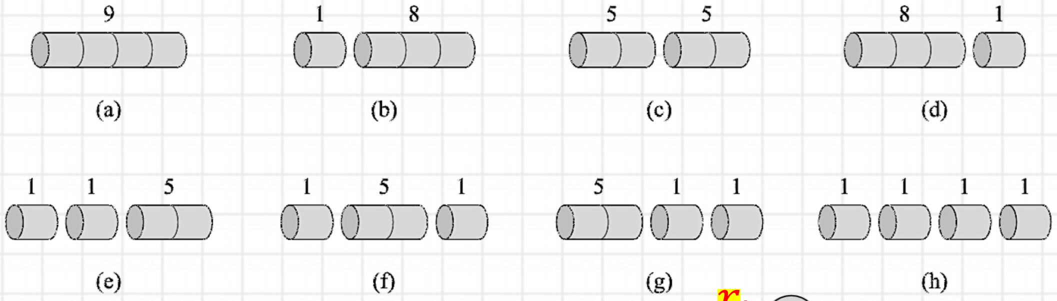
- Example:
  - How many ways to cut up a rod of length n? 2^{n-1}
  - Find an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$, for some $1 \le k \le n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.

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

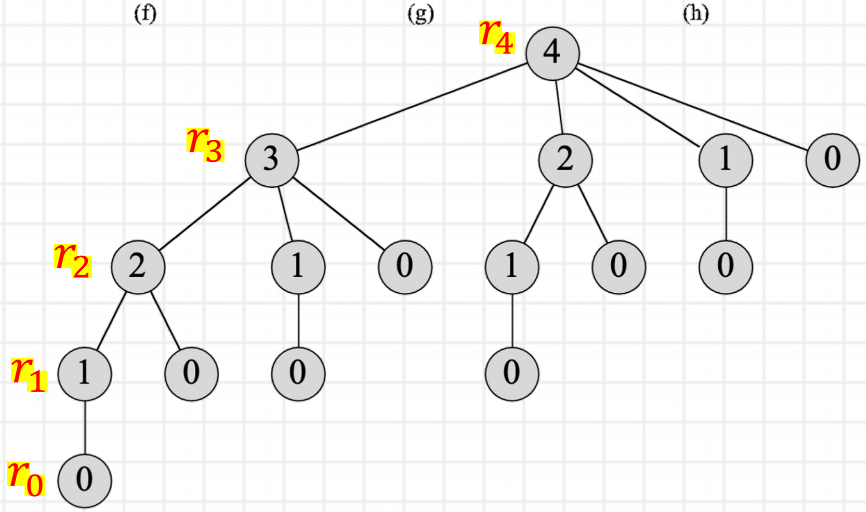


DP Example: (3) Rod-cutting

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.



$n = 0 \Rightarrow r_0 = 0$

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

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

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

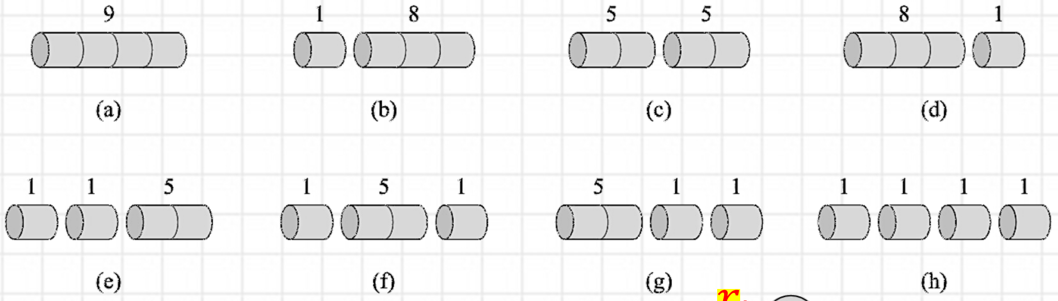
$n = 4 \Rightarrow r_4 = \max\left(\boxed{p_4 + r_0}, \boxed{p_3 + r_1}, \boxed{p_2 + r_2}, \boxed{p_1 + r_3}\right)$

…

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

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.

$n = 0 \Rightarrow r_0 = 0$

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

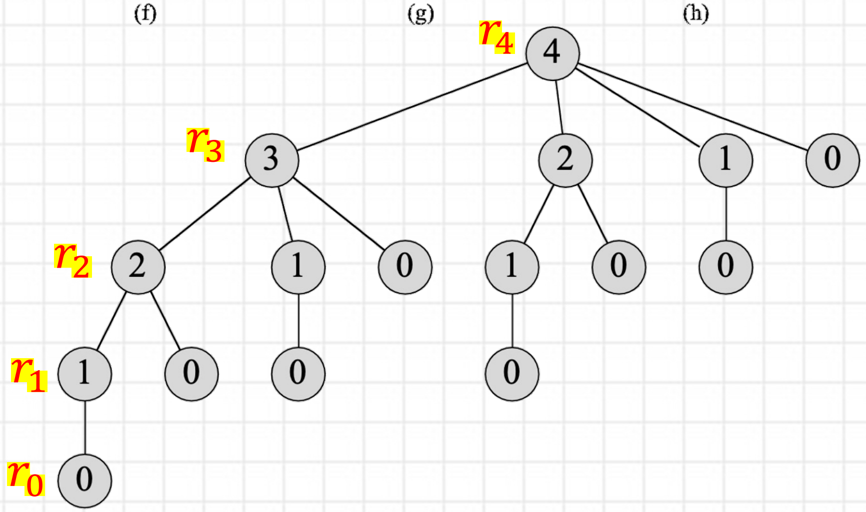
$n = 2 \Rightarrow r_2 = \max\left(\boxed{p_2 + r_0}, \boxed{p_1 + r_1}\right)$

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

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

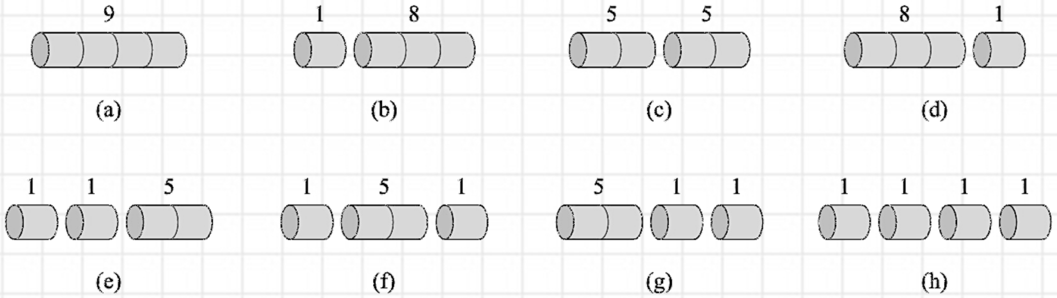
…

$n \Rightarrow r_n = \max_{1 \le i \le n}\left(\boxed{p_i + r_{n-i}}\right)$ Recurrence relation ⟹ Recursive algorithm



DP Example: (3) Rod-cutting

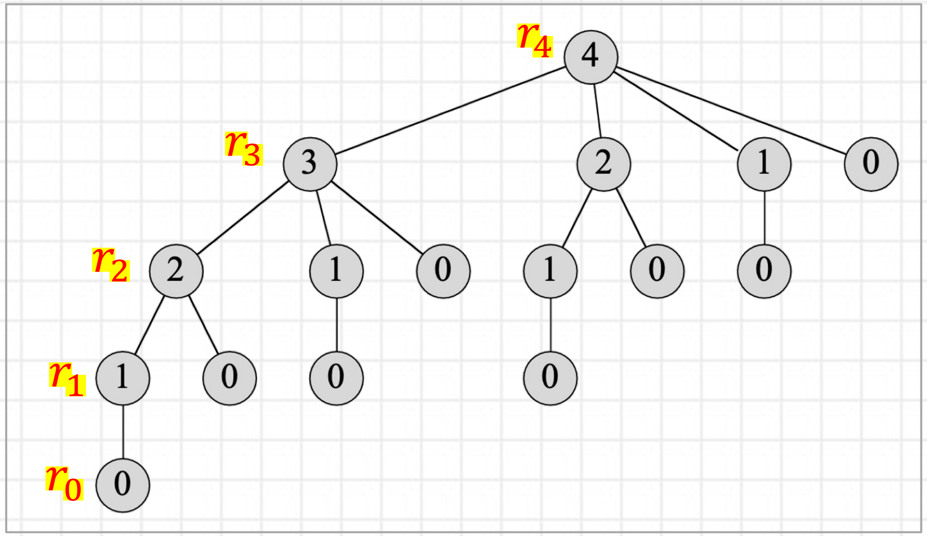
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



- 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

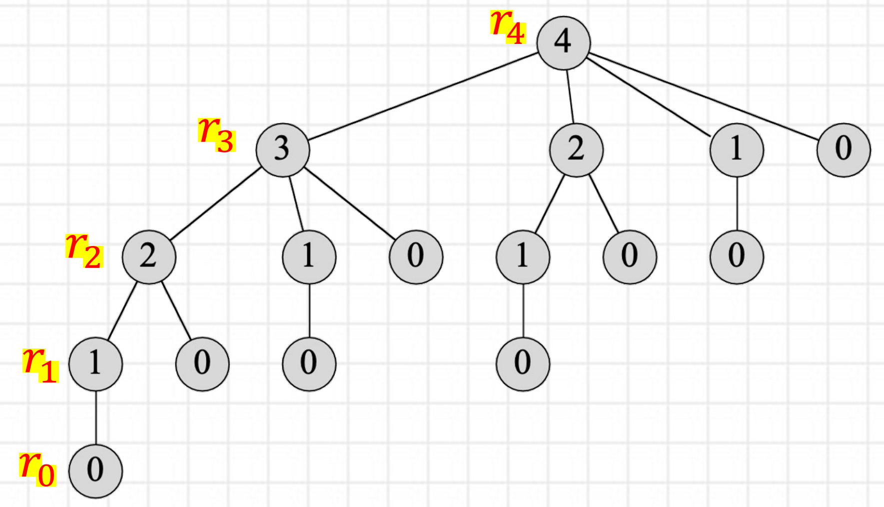
```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

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 $1 \leq k \leq n$ such that the revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is the maximum revenue.
  - Recurrence relation: $r_n = \max\limits_{1 \leq i \leq n} (p_i + r_{n-i}), r_0 = 0$
  - Recursive (brute force) algorithm



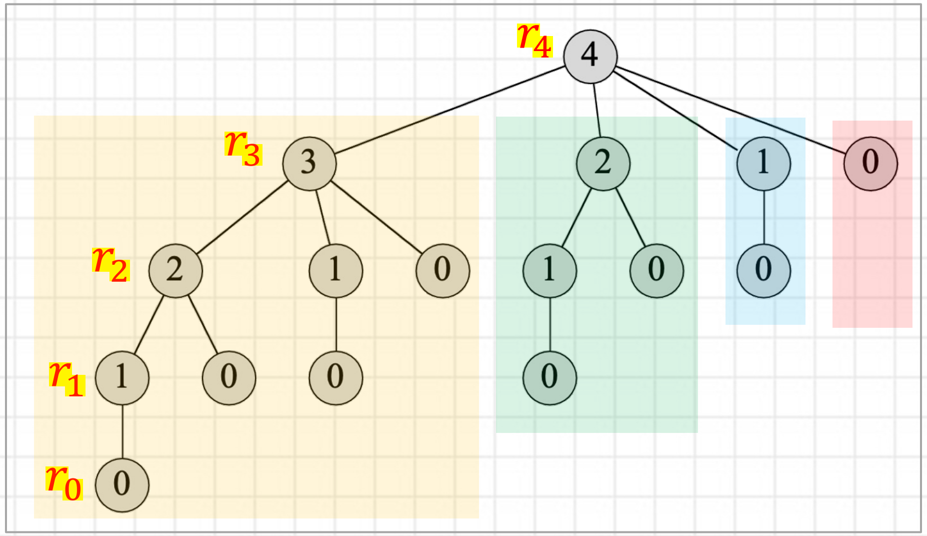

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

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: $r_n = \max_{1 \le i \le n}(p_i + r_{n-i}), r_0 = 0$
  - 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 + …**

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

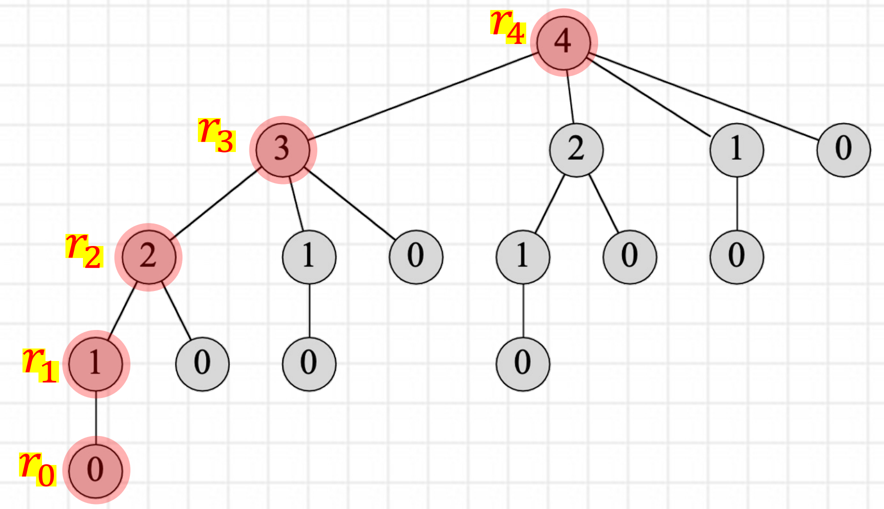
DP Example: (3) Rod-cutting

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

Top-down (recursive with memoization)

```
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)
```

```
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 (iterative)

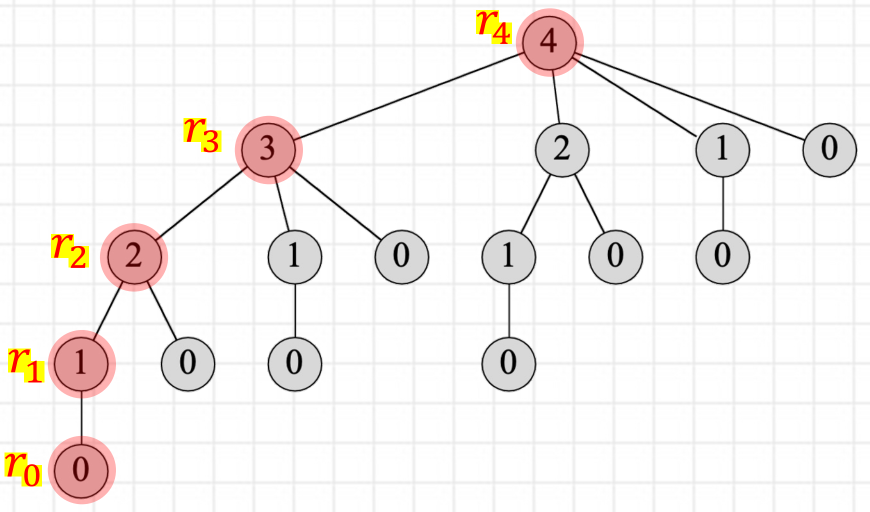
```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

DP Example: (3) Rod-cutting

- DP solution

Time complexity: $O(n^2)$
Space complexity: $O(n)$

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



Top-down (recursive with memoization)

```
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)
```

```
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 (iterative)

```
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]
```

Dynamic Programming (DP)

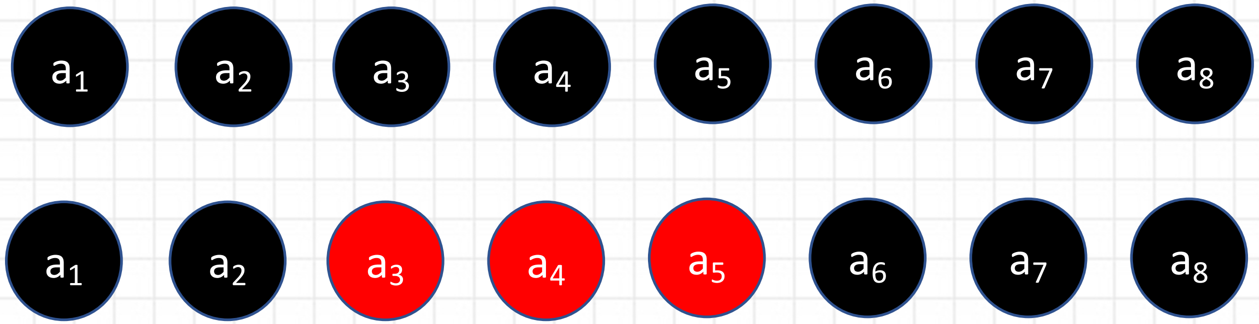
- Dynamic Programming Elements
  - DP often (not always!) applicable to optimization problems
    - Large number of possible solutions
    - Must find the "best" one (maximum or minimum)
  - "Optimal substructure"
    - Finding the optimal solution involves finding the optimal solution to subproblems
    - The subproblems are the same as the original problem, but are "smaller" (e.g., involve smaller-sized input data) Similar to D&C
  - "Overlapping subproblems" 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 note 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+memoization) or bottom-up (iterative) approach).
  4. Analysis: show that time and space complexity is polynomial.

DP Example: (4) Red-Black Game

- You are given a sequence of n positive numbers (a_1, a_2, ..., a_n). Initially, they are all colored black. At each move, you choose a black number a_k and color it and its immediate neighbors (if any) red (the immediate neighbors are the elements a_{k-1}, a_{k+1}). You get a_k points for this move. The game ends when all numbers are colored red. The goal is to get as many points as possible.



DP Example: (4) Red-Black Game

- Going for the most valuable remaining black number?
  - Counter example: A =[7, 3, 90, 100, 80, 5] → A =[7, 3, ~~90~~, 100, ~~80~~, 5]

- DP Solution:
  - Original problem is to select from n numbers s.t. maximizing the total value.
  - The optimal solution to the original problem as OPT(n)
  - Subproblem: find OPT(i), where we select from the first i numbers a_1, a_2, ..., a_i
  - The solution OPT(i) either incudes a_i or not includes a_i:
    - <u>OPT(i) includes a_i</u>. Then OPT(i) can not include a_{i-1} as a_{i-1} will be colored red. So, OPT(i) would include an optimal solution for numbers a_1, ..., a_{i-2}, that is, $OPT(i-2)$.
    - <u>OPT(i) does NOT includes a_i</u>. Then OPT(i) is an optimal solution for numbers a_1, ..., a_{i-1}.

  - Recurrence relation:
    - $OPT(i) = \max\{OPT(i\text{-}2) + a_i, OPT(i\text{-}1)\}$
    - $OPT(0) = 0$, $OPT(1) = a_1$

Multidimensional DP

- "State" variables = variables needed for defining the recurrence relation
- Dimension of a DP algorithm = number of state variables
- So far, only one state variable → one-dimensional DP
  - Fibonacci: $\text{Fib}[n] = \text{Fib}[n-1] + \text{Fib}[n-2]$
  - Rod-cutting: $\text{revenue}[n] = \max_{1 \le i \le n}(\text{prices}[i] + \text{revenue}[n-i])$
- Sometimes, we need multiple state variables (dimensions) to describe and solve the problem.
  - Two dimensional (more common).
    - Longest common subsequence (LCS), knapsack, coin-changing, etc.
  - Three dimensional:
    - All-pairs shortest path (Floyd-Warshall)

DP Example: (5) Longest Common Subsequence

Motivation

- In biology, DNA strands represented as strings of bases: adenine (A), guanine (G), cytosine (C), thymine (T)
- For example: ACCGGTCGAGTGC…
- One operation of interest is to determine the “similarity” of two different strings

Longest Common Subsequence (LCS)

- Sequence is an ordered list of elements

  $X = \langle x_1, x_2, \ldots x_m \rangle$
- Z is a subsequence of X if there is a strictly increasing sequence of indices $i_1, i_2, \ldots i_k$ such that $z_1 = x_{i1}, z_2 = x_{i2}, \ldots, z_k = x_{ik}$

Example: X = <A, B, C, B, D, A, B>

Z = <B, C, D, B> is a subsequence of X

Z = <A, C, A, D> is not a subsequence of X

- In other words, Z can be constructed by starting with X, and deleting zero or more elements

LCS

- Given two sequences:

$$X = \langle x_1, x_2, \ldots x_m \rangle$$
$$Y = \langle y_1, y_2, \ldots y_n \rangle$$

Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

Compute: LCS(X,Y) = longest common subsequence of X and Y

Example:

X = <A, B, C, B, D, A, B>   Y = <B, D, C, A, B, A>

<B, C, A> is a common subsequence of X and Y

<B, C, A, B> is an LCS of X and Y

<B, C, B, A> and <B, D, A, B> are also LCS's of X and Y

(LCS may not be unique!)

LCS

- Brute-force solution:
  - Enumerate all subsequences of X
  - For each such subsequence, is it also a subsequence of Y?
  - Pick the longest one that is a subsequence of both X and Y

- What is the runtime of the brute-force solution?
  - m elements in X
  - n elements in Y

- Hint:
  - How many subsequences in X?
  - How many comparisons needed?

LCS

- Brute-force solution:
  - Enumerate all subsequences of X
  - For each such subsequence, is it also a subsequence of Y?
  - Pick the longest one that is a subsequence of both X and Y

- What is the runtime of the brute-force solution?
  - m elements in X
  - n elements in Y

- Hint:
  - How many subsequences in X?
  - How many comparisons needed?

- There are 2^m subsequences in X (each element of X is either in the subsequence or not)
- There are n comparisons needed for each subsequence
- $n * 2^m$ comparisons
- Exponential runtime!

LCS

- Given a sequence: $X = <x_1, x_2, \ldots x_m>$
  $X_i = <x_1, x_2, \ldots x_i>$ is defined as the i^{th} prefix of X, i=0, 1, …m
  (X_i is the first i elements of X)

  - Example: X = <A, B, C, B>
  - $X_0 = <>$
  - $X_1 = <A>$
  - $X_2 = <A, B>$
  - $X_3 = <A, B, C>$
  - $X_4 = <A, B, C, B>$

LCS

- Given a sequence: $X = \langle x_1, x_2, \ldots x_m \rangle$
  $X_i = \langle x_1, x_2, \ldots x_i \rangle$ is defined as the i^{th} prefix of X, i=0, 1, …m
  (X_i is the first i elements of X)

- Example: $X = \langle A, B, C, B \rangle$
- $X_0 = \langle\rangle$
- $X_1 = \langle A \rangle$
- $X_2 = \langle A, B \rangle$
- $X_3 = \langle A, B, C \rangle$
- $X_4 = \langle A, B, C, B \rangle$

- Key Observation:
- The LCS of sequences X and Y can be found by finding the LCS of prefixes of X and Y

- This leads to development of a recursive solution to computing LCS

LCS: Optimal Substructure

- Let
  $X = <A, B, C, B, D, A, B, x_8>$ (m=8)
  $Y = <B, D, C, A, B, y_6>$ (n=6)
  $LCS(X,Y) = Z = <z_1, z_2, \ldots z_k>$
- Suppose $x_8 = y_6$:

  Then $Z = LCS(X, Y) = LCS(X_7, Y_5) + z_k$, where $z_k = x_8 (= y_6)$
- Suppose $x_8 \neq y_6$:

  if $z_k \neq x_8$ then $Z = LCS(X_7, Y)$

  if $z_k \neq y_6$ then $Z = LCS(X, Y_5)$
- In other words, LCS(X,Y) can be built of the LCS of the prefixes of X and Y
- Subproblems same as original, but with smaller input data

LCS: Optimal Substructure

- Let
  $X = <A, B, C, B, D, A, B, x_8>$ (m=8)
  $Y = <B, D, C, A, B, y_6>$ (n=6)
  $LCS(X,Y) = Z = <z_1, z_2, \ldots z_k>$
- Suppose $x_8 = y_6$:

  Then $Z = LCS(X, Y) = LCS(X_7, Y_5) + z_k$, where $z_k = x_8$ $(= y_6)$

  The last element of X and Y is the last element of the solution

- Suppose $x_8 \neq y_6$:

  if $z_k \neq x_8$ then $Z = LCS(X_7, Y)$

  if $z_k \neq y_6$ then $Z = LCS(X, Y_5)$

- In other words, LCS(X,Y) can be built of the LCS of the prefixes of X and Y
- Subproblems same as original, but with smaller input data

LCS: Optimal Substructure

- Let
  $X = \langle A, B, C, B, D, A, B, x_8 \rangle$ (m=8)
  $Y = \langle B, D, C, A, B, y_6 \rangle$ (n=6)
  $LCS(X,Y) = Z = \langle z_1, z_2, \ldots z_k \rangle$
- Suppose $x_8 = y_6$:

  Then $Z = LCS(X, Y) = LCS(X_7, Y_5) + z_k$, where $z_k = x_8$ $(= y_6)$
- Suppose $x_8 \neq y_6$:

  if $z_k \neq x_8$ then $Z = LCS(X_7, Y)$

  if $z_k \neq y_6$ then $Z = LCS(X, Y_5)$

Continue search using prefix of X

- In other words, LCS(X,Y) can be built of the LCS of the prefixes of X and Y
- Subproblems same as original, but with smaller input data

LCS: Optimal Substructure

- Let
  $X = <A, B, C, B, D, A, B, x_8> \ (m=8)$
  $Y = <B, D, C, A, B, y_6> \ (n=6)$
  $LCS(X,Y) = Z = <z_1, z_2, \ldots z_k>$
- Suppose $x_8 = y_6$:

  Then $Z = LCS\ (X, Y) = LCS\ (X_7, Y_5) + z_k$, where $z_k = x_8\ (= y_6)$
- Suppose $x_8 \neq y_6$:

  if $z_k \neq x_8$ then $Z = LCS\ (X_7, Y)$

  if $z_k \neq y_6$ then $Z = LCS\ (X, Y_5)$

  Continue search using prefix of Y
- In other words, LCS(X,Y) can be built of the LCS of the prefixes of X and Y
- Subproblems same as original, but with smaller input data

LCS: Recurrence

- Let
  $X = \langle A, B, C, B, D, A, B, x_8 \rangle$ (m=8)
  $Y = \langle B, D, C, A, B, y_6 \rangle$ (n=6)
  $LCS(X,Y) = Z = \langle z_1, z_2, \ldots z_k \rangle$

If ($x_m == y_n$):
- $z_k = x_m$;
- compute LCS (X_{m-1}, Y_{n-1})

Else:
- compute LCS (X_{m-1}, Y) and LCS (X, Y_{n-1})
- pick the longer subsequence of the two

Overlapping subproblems

- The above subproblems share many computations.
  - For example, computing LCS (X_{m-1}, Y) and LCS (X, Y_{n-1}) both involve computing LCS (X_{m-1}, Y_{n-1})

LCS: Recurrence

- Compute the length of the LCS
  - Involves computing LCS of prefixes to X and Y
- Let $c[i,j] = LCS(X_i, Y_j)$
  - Data structure used for memoization

If ($x_m == y_n$):
- $z_k = x_m$;
- compute LCS (X_{m-1}, Y_{n-1})

Else:
- compute LCS (X_{m-1}, Y) and LCS (X, Y_{n-1})
- pick the longer subsequence of the two

- $$c[i,j] = \begin{cases} 0, & \text{if } (i=0 \text{ or } j=0) \\ c[i-1,j-1] + 1, & \text{if } i>0, j>0, \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i>0, j>0, \text{ and } x_i \neq y_j \end{cases}$$

- c[m,n] is the length of LCS(X, Y)

LCS: Recurrence

- Compute the length of the LCS
  - Involves computing LCS of prefixes to X and Y
- Let $c[i,j] = LCS(X_i, Y_j)$
  - Data structure used for memoization

If $(x_m == y_n)$:
- $z_k = x_m$;
- compute $LCS(X_{m-1}, Y_{n-1})$

Else:
- compute $LCS(X_{m-1}, Y)$ and $LCS(X, Y_{n-1})$
- pick the longer subsequence of the two

- $$c[i,j] = \begin{cases} 0, & \text{if } (i=0 \text{ or } j=0) \\ c[i-1,j-1] + 1, & \text{if } i>0, j>0, \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i>0, j>0, \text{ and } x_i \neq y_j \end{cases}$$

- $c[m,n]$ is the length of $LCS(X, Y)$

LCS: Computation

- c[i,j] = 0, if (i=0 or j=0)
  = c[i-1,j-1] + 1, if i>0, j>0, and $x_i = y_j$
  = max (c[i, j-1], c[i-1, j]) if i>0, j>0, and $x_i \neq y_j$

```
// compute LCS for 0 length cases
for (i=0; i<=m; i++) c[i,0]=0;
for (j=0; j<=n; j++) c[0,j]=0;
// compute in row-major order
for (i=1; i<=m; i++)
    for (j=1; j<=n; j++)
        if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
        // c[i][j]=max(c[i-1][j],c[i][j-1])
        else if (c[i-1][j]>=c[i][j-1]): c[i][j] = c[i-1][j];
        else: c[i][j] = c[i][j-1];
```

LCS: Example

Determine longest common subsequence of X and Y

- X = ABCB
- Y = BDCAB

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS: Example

ABCB
BDCAB

n+1 columns

m+1 rows

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = 4
Y = BDCAB; n = 5

LCS: Example

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
for (i=0; i<=m; i++) c[i,0]=0;
for (j=0; j<=n; j++) c[0,j]=0;
```

LCS: Example

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	↓ **0**	↓ **0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
if (xi==yj) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

if $(x_i==y_j)$ c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])

LCS: Example

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	→ 1	→ 1	↓ 1	
3	C	0					
4	B	0					

if $(x_i == y_j)$ c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])

LCS: Example

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

if $(x_i == y_j)$ c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])

LCS: Example

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	→ **2**	↓ **2**
4	**B**		**0**					

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

i \ j		0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**				

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	↓ 1	↓ 2	↓ 2	

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

LCS: Example

ABCB
BDCAB

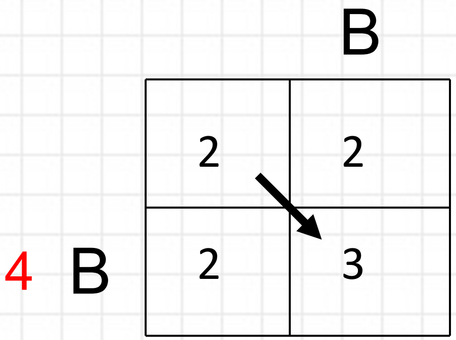
i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

Length of LCS!

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

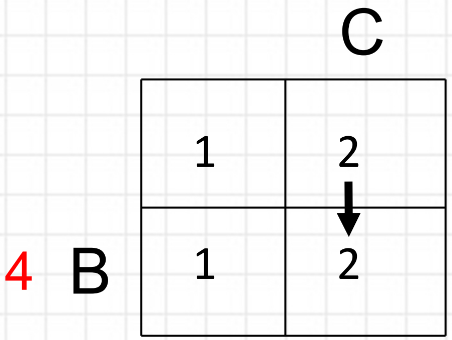
LCS: Computing the LCS

- The previous step determined the *length* of LCS, but not the LCS itself.
- Each c[i,j] depends on c[i-1,j] and c[i,j-1] or c[i-1, j-1]
- For each c[i,j] we can record how it was acquired:



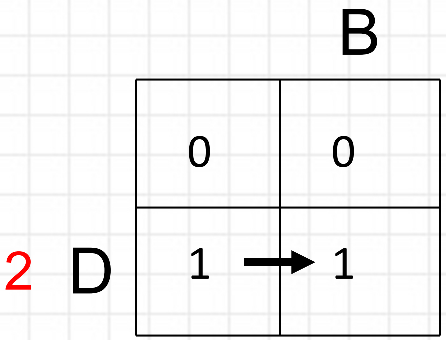

```
if (x_i==y_j)
    c[i][j]=
     c[i-1][j-1]+1;
```

"F"=found




```
else if (c[i-1][j]
       >= c[i][j-1])
c[i][j] = c[i-1][j];
```

"X"=advance X




```
else c[i][j] =
  c[i][j-1];
```

"Y"=advance Y

LCS: Computing the LCS

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0,X	0,X	0,X	1,F	1,Y
2	B	0	1,F	1,Y	1,Y	1,X	2,F
3	C	0	1,X	1,X	2,F	2,Y	2,X
4	B	0	1,F	1,X	2,X	2,X	3,F

```
// annotate: found("F"),
// advance X("X"),advance Y("Y")
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    if (x_i==y_j):
      c[i][j]=c[i-1][j-1]+1;
      b[i][j]="F";
    else if (c[i-1][j]>=c[i][j-1])
      c[i][j] = c[i-1][j];
      b[i][j]="X";
    else
      c[i][j] = c[i][j-1];
      b[i][j]="Y";
```

LCS: Computing the LCS

- Remember that

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- So, we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of the LCS computed)
- When i=0 or j=0 (i.e., we reached the beginning), output the remembered letters in reverse order

LCS: Computing the LCS

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0,X	0,X	0,X	1,F	1,Y
2	B	0	1,F	1,Y	1,Y	1,X	2,F
3	C	0	1,X	1,X	2,F	2,Y	2,X
4	B	0	1,F	1,X	2,X	2,X	3,F

```
// annotate: found("F"),
// advance X("X"),advance Y("Y")
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    if (x_i==y_j):
      c[i][j]=c[i-1][j-1]+1;
      b[i][j]="F";
    else if (c[i-1][j]>=c[i][j-1])
      c[i][j] = c[i-1][j];
      b[i][j]="X";
    else
      c[i][j] = c[i][j-1];
      b[i][j]="Y";
```

LCS: Computing the LCS

i	Xi	j=0 (Yj)	1 (B)	2 (D)	3 (C)	4 (A)	5 (B)
0		0	0	0	0	0	0
1	A	0	0,X	0,X	0,X	1,F	1,Y
2	B	0	1,F	1,Y	1,Y	1,X	2,F
3	C	0	1,X	1,X	2,F	2,Y	2,X
4	B	0	1,F	1,X	2,X	2,X	3,F

```
// annotate: found("F"),
// advance X("X"),advance Y("Y")
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    if (xi==yj):
      c[i][j]=c[i-1][j-1]+1;
      b[i][j]="F";
    else if (c[i-1][j]>=c[i][j-1])
      c[i][j] = c[i-1][j];
      b[i][j]="X";
    else
      c[i][j] = c[i][j-1];
      b[i][j]="Y";
```

LCS (reversed order): B C B → B C B (forward)

LCS: Output (Printing) the LCS

```
// annotate: found("F"),
// advance X("X"),advance Y("Y")
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    if (xi==yj):
      c[i][j]=c[i-1][j-1]+1;
      b[i][j]="F";
    else if (c[i-1][j]>=c[i][j-1])
      c[i][j] = c[i-1][j];
      b[i][j]="X";
    else
      c[i][j] = c[i][j-1];
      b[i][j]="Y";
```

```
// to print LCS, call Print_LCS:
Print_LCS(b, X, m, n);


// follow annotations to print out
Print_LCS(b, X, i, j):
  if ((i==0) || (j==0)) return;
  if (b[i][j] == "F")
    Print_LCS(b, X, i-1, j-1);
    print (x);
  else if (b[i][j] == "X")
    Print_LCS(b, X, i-1, j);
  else
    Print_LCS(b, X, i, j-1);
```

LCS: Running Time

- What is the execution time for each step of this algorithm?
  - Step 1: Computing LCS
  - Step 2: Printing

LCS: Running Time

- What is the execution time for each step of this algorithm?

  - Step 1: Computing LCS
    - O(m×n) to fill in matrix

  - Step 2: Printing
    - O(m+n)

DP: Summary

- Dynamic programming is a general algorithm approach similar to divide and conquer, but with shared/overlapped subproblems rather than disjoint ones.
- Efficiency is obtained by recording (memoization) the solution of subproblems rather than recomputing them.
- Dynamic programming applicable to many optimization problems
- Two main elements:
  - Optimal substructure
  - Overlapping subproblems

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.