

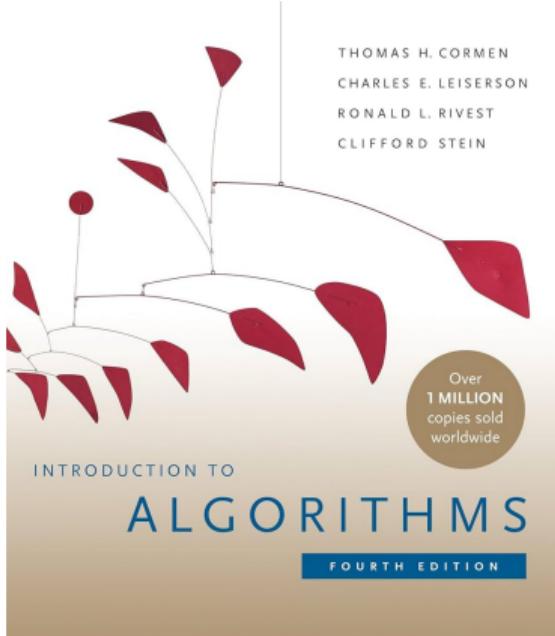
Introduction to Algorithms

Lecture 5: Dynamic Programming (DP)

Prof. Charles E. Leiserson and Prof. Erik Demaine
Massachusetts Institute of Technology

September 8, 2025

Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.

Visit <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.

Original slides from *Introduction to Algorithms* 6.046J/18.401J, Fall 2005 Class by Prof. Charles Leiserson and Prof. Erik Demaine. MIT OpenCourseWare Initiative available at
<https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/>.

Plan

Dynamic Programming

N-th Fibonacci Number

Longest Palindromic Sequence

Longest Common Subsequence

Rod Cutting Problem

All-pairs shortest paths problem

Dynamic Programming* (DP)

- ▶ Invented by Richard Bellman in 1950s.
- ▶ Design technique, like Divide & Conquer.
- ▶ Applies when the subproblems overlap (that is, when subproblems share subsubproblems).
- ▶ It solves each subsubproblem just once and then saves its answer in a table.
- ▶ DP typically applies to optimization problems:
 - ▶ have many possible solutions.
 - ▶ find a solution with the optimal (min or max) value.
 - ▶ *an* optimal solution, not *the* optimal solution.

**Programming* in this context refers to a tabular method, not to writing computer code.

DP notions

DP notions

1. Characterize the structure of an optimal solution.

DP notions

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems.

DP notions

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems.
3. Compute the value of an optimal solution in bottom-up fashion (recursion & memoization).

DP notions

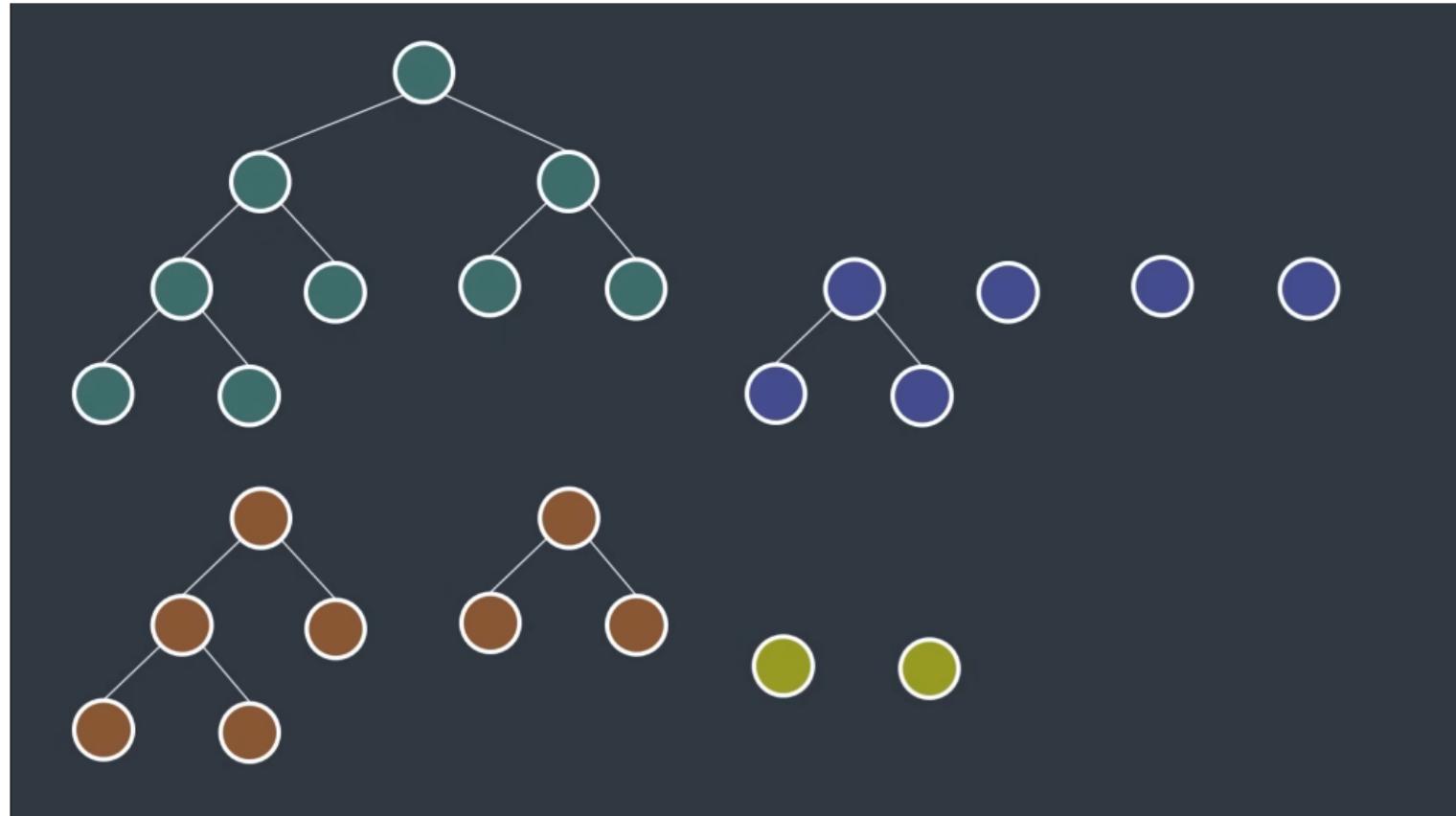
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems.
3. Compute the value of an optimal solution in bottom-up fashion (recursion & memoization).
4. Construct an optimal solution from the computed information.

DP notions

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems.
3. Compute the value of an optimal solution in bottom-up fashion (recursion & memoization).
4. Construct an optimal solution from the computed information.

DP = Recursion + Memoization

DP notions



Plan

Dynamic Programming

N-th Fibonacci Number

Longest Palindromic Sequence

Longest Common Subsequence

Rod Cutting Problem

All-pairs shortest paths problem

N-th Fibonacci Number[†]

Write a function that returns the n-th Fibonacci number.

$$F_1 = F_2 = 1$$

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

n	1	2	3	4	5	6	7
F_n	1	1	2	3	5	8	13

[†] Mastering Dynamic Programming - How to solve any interview problem (Part 1). Tech With Nikola Channel, 2024. YouTube, available at <https://youtu.be/Hdr64lKQ3e4?si=ycTe-hoyfaICRWXt>

Naive Approach

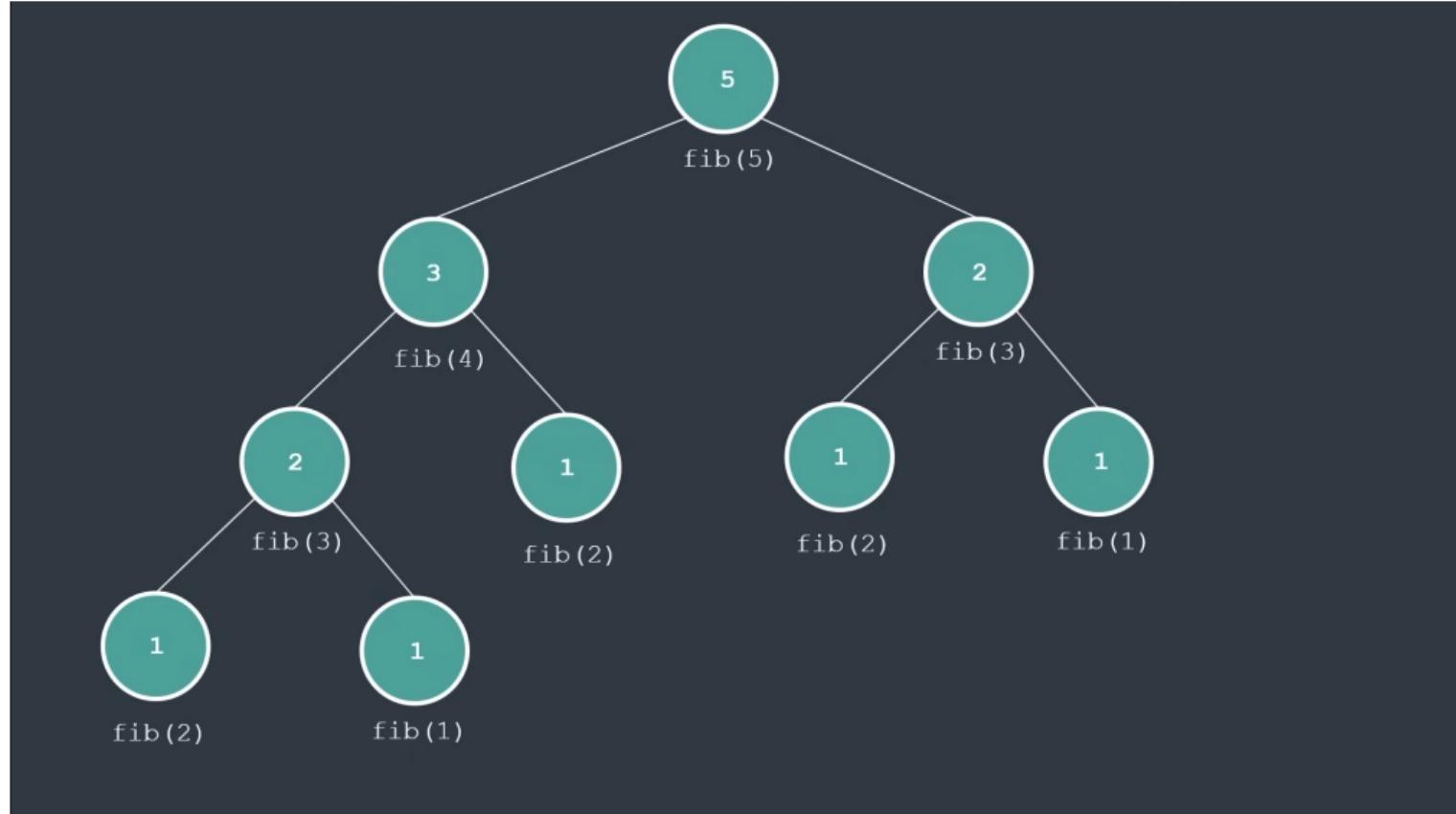
```
1  def fib(n):
2      if n <= 2:
3          result = 1
4      else:
5          result = fib(n - 1) + fib(n - 2)
6      return result
```

Naive Approach

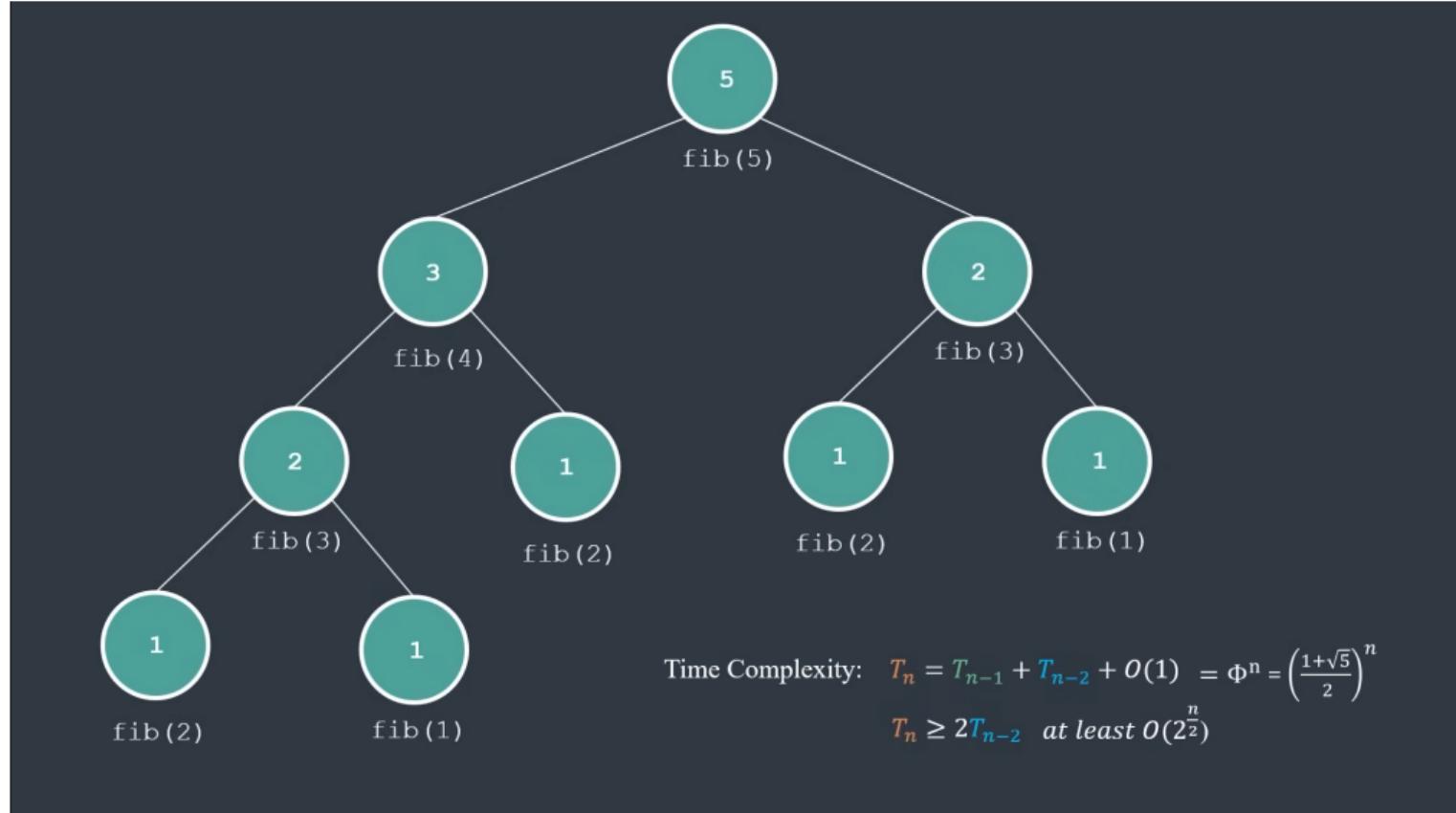
```
1  def fib(n):
2      if n <= 2:
3          result = 1
4      else:
5          result = fib(n - 1) + fib(n - 2)
6      return result
```

```
print(fib(7))
Output: 13
print(fib(50))
Output: ???
```

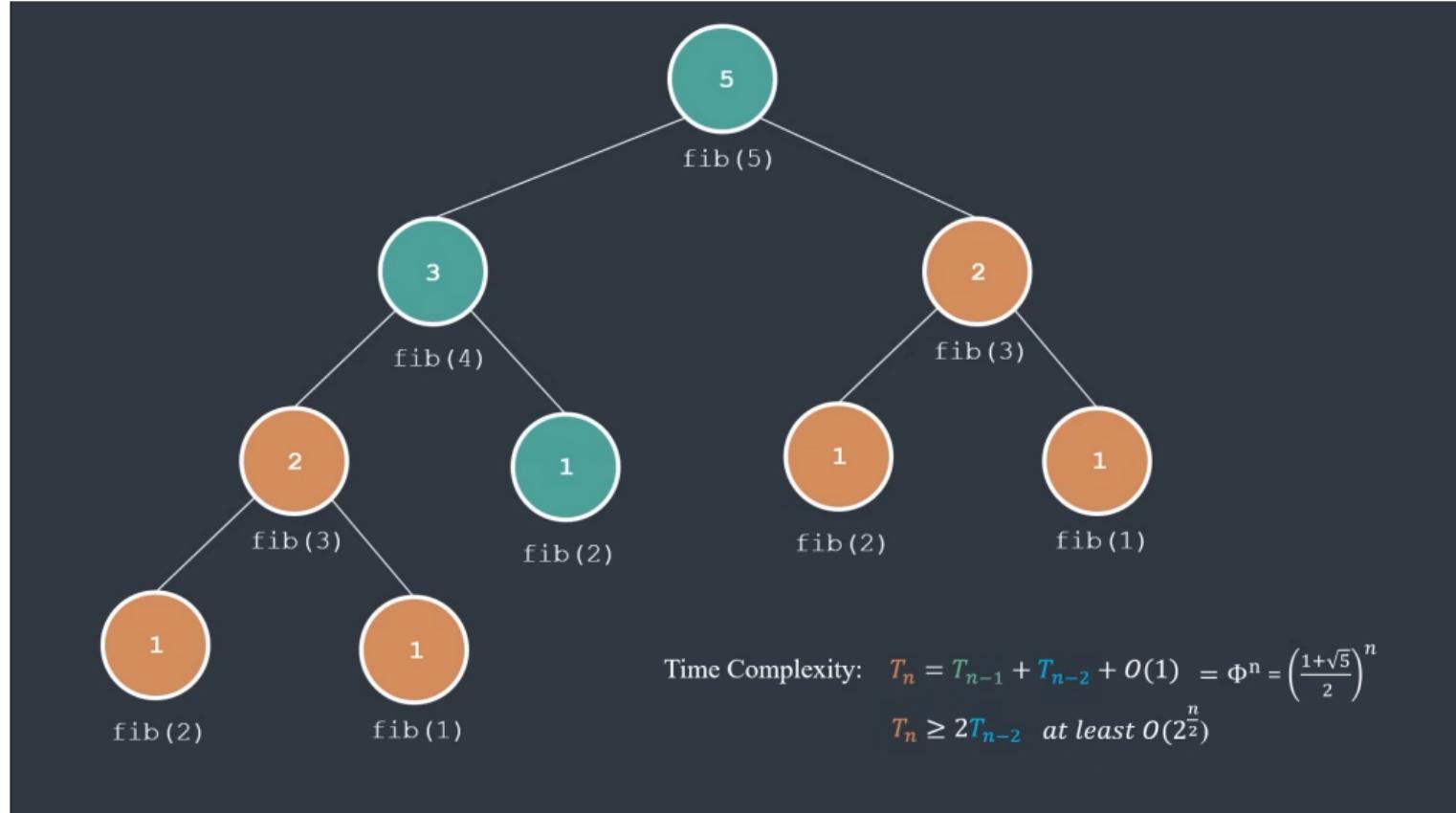
Naive Approach



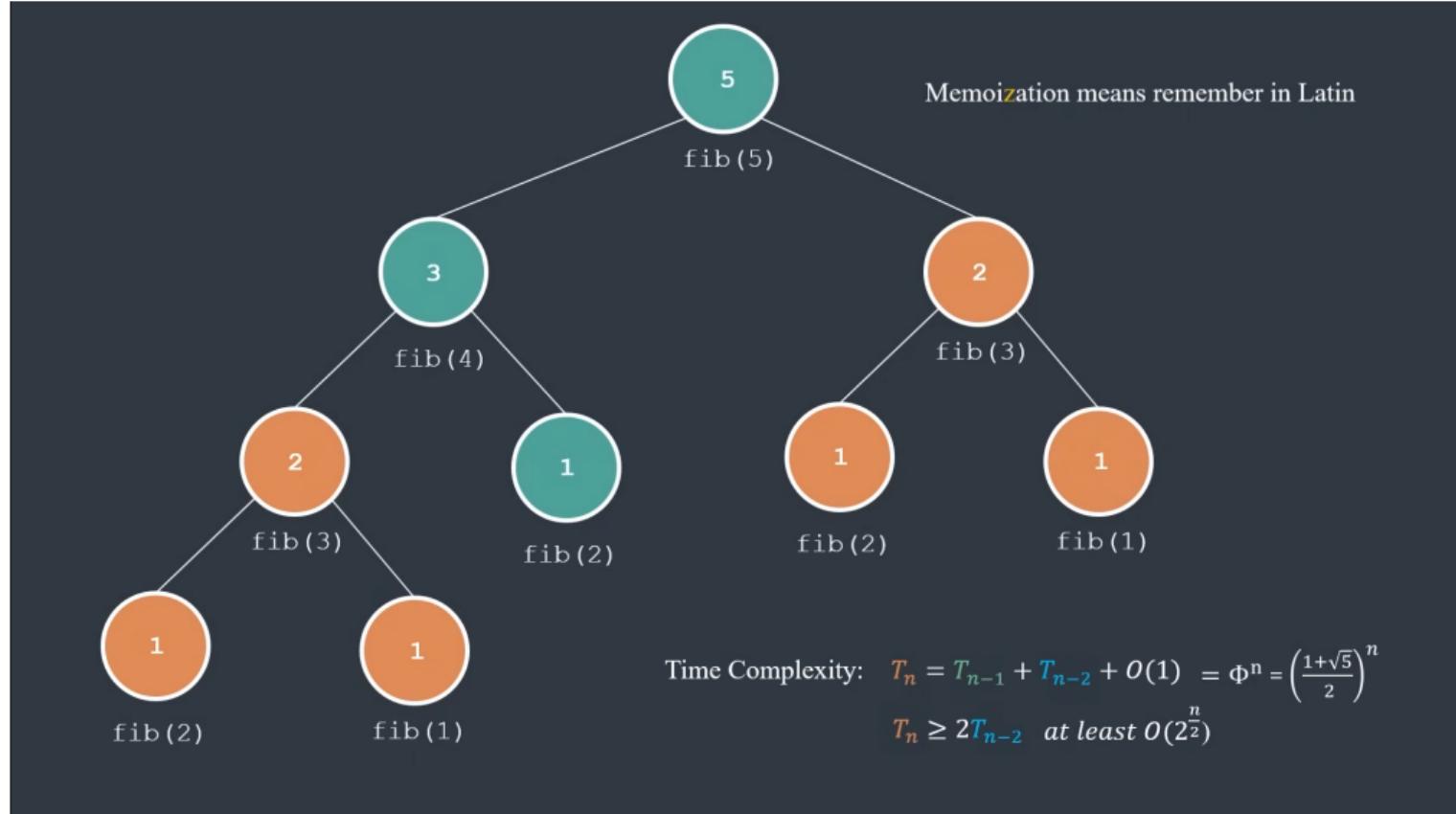
Naive Approach



Naive Approach



Naive Approach



Memoization Approach

```
1     memo = {} # adding a dictionary...
2     def fib(n):
3         if n in memo: # asking if n is in dict...
4             return memo[n] # if so, we are done!
5         if n <= 2:
6             result = 1
7         else:
8             result = fib(n - 1) + fib(n - 2)
9         return result
```

Memoization Approach

```
1     memo = {} # adding a dictionary...
2     def fib(n):
3         if n in memo: # asking if n is in dict...
4             return memo[n] # if so, we are done!
5         if n <= 2:
6             result = 1
7         else:
8             result = fib(n - 1) + fib(n - 2)
9         return result
```

```
print(fib(7))
Output: 13
print(fib(50))
Output: 12586269025
```

Memoization Approach

```
1     memo = {} # adding a dictionary...
2     def fib(n):
3         if n in memo: # asking if n is in dict...
4             return memo[n] # if so, we are done!
5         if n <= 2:
6             result = 1
7         else:
8             result = fib(n - 1) + fib(n - 2)
9         return result
```

```
print(fib(7))
Output: 13
print(fib(50))
Output: 12586269025
```

Time Complexity: $O(n)$.

DP = Recursion + Memoization

Bottom-up Approach

```
1  def fib(n):
2      memo = []
3      for i in range(1, n + 1):
4          if i <= 2:
5              result = 1
6          else:
7              result = memo[i - 1] + memo[i - 2]
8          memo[i] = result
9  return memo[n]
```

Topological sort order



Plan

Dynamic Programming

N-th Fibonacci Number

Longest Palindromic Sequence

Longest Common Subsequence

Rod Cutting Problem

All-pairs shortest paths problem

Longest Palindromic Sequence

Definition:

A palindrome is a string that is unchanged when reversed.

- ▶ Examples: **radar, civic, t, bb, redder.**
- ▶ Given: A string $X[1 \dots n]$, $n \geq 1$.
- ▶ To find: Longest palindrome that is a subsequence.
- ▶ Example: Given “c h a r a c t e r”.
- ▶ Output: “c a r a c”.
- ▶ Answer will be ≥ 1 in length.

Strategy

$L(i, j)$: length of longest palindromic subsequence of $X[i \dots j]$ for $i \leq j$.

```
1  def L(i, j):
2      if i == j:
3          return 1
4      if X[i] == X[j]:
5          if i + 1 == j:
6              return 2
7          else:
8              return 2 + L(i + 1, j - 1)
9      else:
10         return max( L(i + 1, j), L(i, j - 1) )
```

Analysis

As written, program can run in exponential time: suppose all symbols $X[i]$ are distinct.

$T(n)$ = running time on input of length n

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n - 1) & n > 1 \end{cases}$$
$$= 2^{n-1}$$

What is missing?

- ▶ Complexity is exponential... why?

What is missing?

- ▶ Complexity is exponential... why?
- ▶ We are still not completing all the DP notions...
- ▶ There is **recursion** but there is not **Memoization**...

What is missing?

- ▶ Complexity is exponential... why?
- ▶ We are still not completing all the DP notions...
- ▶ There is **recursion** but there is not **Memoization**...
- ▶ Cache is missing!
- ▶ There is a single line of code that will fix it...

Understanding the Subproblems

The problem typically involves checking whether a substring of a given string is a palindrome. If the input string has a length of n , then the natural way to define subproblems is:

- ▶ Consider all possible substrings $s[i : j]$ of the string.
- ▶ For each substring, determine whether it is a palindrome.

Counting the Subproblems

A substring is defined by two indices, i (starting index) and j (ending index), where $0 \leq i \leq j < n$. This means:

- ▶ i can take values from 0 to $n - 1$.
- ▶ For each i , j can take values from i to $n - 1$.

Total Number of Subproblems

The total number of such (i, j) pairs (i.e., total subproblems) is given by the summation:

$$\sum_{i=0}^{n-1} (n - i) = n + (n - 1) + (n - 2) + \cdots + 1$$

Using the formula for the sum of the first n natural numbers:

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Thus, the number of subproblems is:

$$n^2$$

Formula for Computing the Complexity of a DP

of subproblems \times time to solve each subproblem

Given that smaller ones are already solved[‡].

So,

- ▶ Given n^2 distinct subproblems...
- ▶ By solving each subproblem only once...
- ▶ Running time reduces to:

$$\Theta(n^2) \cdot \Theta(1) = \Theta(n^2)$$

[‡]lookup is $\Theta(1)$

New Strategy

- ▶ Memoize $L(i, j)$, hash inputs to get output value, and lookup hash table to see if the subproblem is already solved, else recurse.

```
1  def L(i, j):
2      if i == j:
3          return 1
4      if X[i] == X[j]:
5          if i + 1 == j:
6              return 2
7          else:
8              return 2 + L(i + 1, j - 1)
9      else:
10         return max( L(i + 1, j), L(i, j - 1) )
```

New Strategy

- ▶ Memoize $L(i, j)$, hash inputs to get output value, and lookup hash table to see if the subproblem is already solved, else recurse.

```
1  def L(i, j):  
2      if i == j:  
3          return 1  
4      if X[i] == X[j]:  
5          if i + 1 == j:  
6              return 2  
7          else:  
8              return 2 + L(i + 1, j - 1)  
9      else:  
10         return max( L(i + 1, j), L(i, j - 1) )
```

Look at $L[i, j]$ and don't recurse if $L[i, j]$ is already computed.

Memoizing Vs. Iterating

1. Memoizing uses a dictionary for $L(i, j)$ where value of L is looked up by using i, j as a key. Could just use a 2-D array here where null entries signify that the problem has not yet been solved.
2. Can solve subproblems in order of increasing $j - i$ so smaller ones are solved first.

Plan

Dynamic Programming

N-th Fibonacci Number

Longest Palindromic Sequence

Longest Common Subsequence

Rod Cutting Problem

All-pairs shortest paths problem

- ▶ Given two sequences $x[1 \cdots m]$ and $y[1 \cdots n]$, find a[†] longest subsequence common to them both.

[†] “a” not “the”

- ▶ Given two sequences $x[1 \cdots m]$ and $y[1 \cdots n]$, find a[†] longest subsequence common to them both.

x: A B C B D A B

y: B D C A B A

[†] “a” not “the”

- ▶ Given two sequences $x[1 \cdots m]$ and $y[1 \cdots n]$, find a[†] longest subsequence common to them both.

x: A B C B D A B

y: B D C A B A

- ▶ BDA ?

[†] “a” not “the”

- ▶ Given two sequences $x[1 \cdots m]$ and $y[1 \cdots n]$, find a[†] longest subsequence common to them both.

x: A B C B D A B

y: B D C A B A

- ▶ BDA ?
- ▶ BDAB

[†] “a” not “the”

- ▶ Given two sequences $x[1 \cdots m]$ and $y[1 \cdots n]$, find a[†] longest subsequence common to them both.

x: A B C B D A B

y: B D C A B A

- ▶ BDA ?
- ▶ BDAB
- ▶ BCAB

[†] “a” not “the”

- ▶ Given two sequences $x[1 \cdots m]$ and $y[1 \cdots n]$, find a[†] longest subsequence common to them both.

x: A B C B D A B

y: B D C A B A

- ▶ BDA ?
- ▶ BDAB
- ▶ BCAB
- ▶ BCBA

[†] “a” not “the”

- ▶ Given two sequences $x[1 \cdots m]$ and $y[1 \cdots n]$, find a[†] longest subsequence common to them both.

x: A B C B D A B

y: B D C A B A

- ▶ BDA ?
- ▶ BDAB
- ▶ BCAB
- ▶ BCBA

LCS(x, y) as notation **not** function...

[†] “a” not “the”

Brute-force LCS algorithm

Brute-force LCS algorithm

- ▶ Check every subsequence of $x[1 \cdots m]$ to see if it is also a subsequence of $y[1 \cdots n]$.

Brute-force LCS algorithm

- ▶ Check every subsequence of $x[1 \dots m]$ to see if it is also a subsequence of $y[1 \dots n]$.

Analysis

- ▶ Checking = $O(n)$ time per subsequence.
- ▶ 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time = $O(n2^m)$

Brute-force LCS algorithm

- ▶ Check every subsequence of $x[1 \cdots m]$ to see if it is also a subsequence of $y[1 \cdots n]$.

Analysis

- ▶ Checking = $O(n)$ time per subsequence.
- ▶ 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time = $O(n2^m)$ **Exponential time!**

Towards a Better Algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Towards a Better Algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
 2. Extend the algorithm to find the LCS itself.
- **Notation:** Denote the length of a sequence s as $|s|$.

Towards a Better Algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

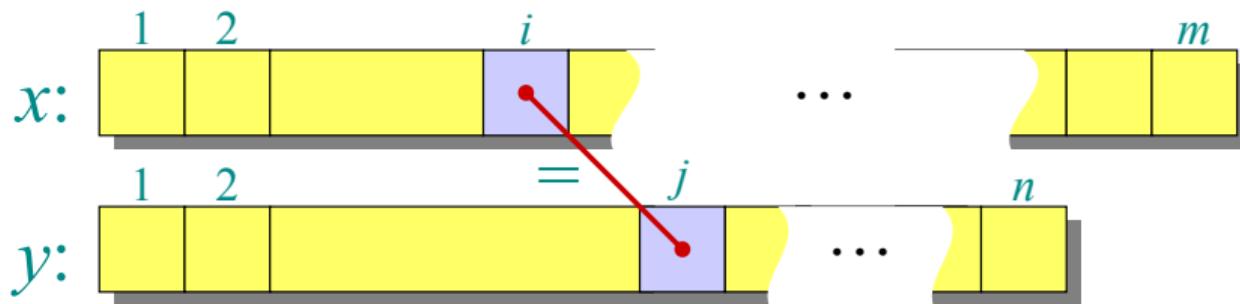
- ▶ **Notation:** Denote the length of a sequence s as $|s|$.
- ▶ **Strategy:** Consider **prefixes** of x and y :
 - ▶ Define $c[i, j] = |LCS(x[1 \cdots i], y[1 \cdots j])|$.
 - ▶ Then, $c[m, n] = |LCS(x, y)|$.

Recursive formulation

Theorem

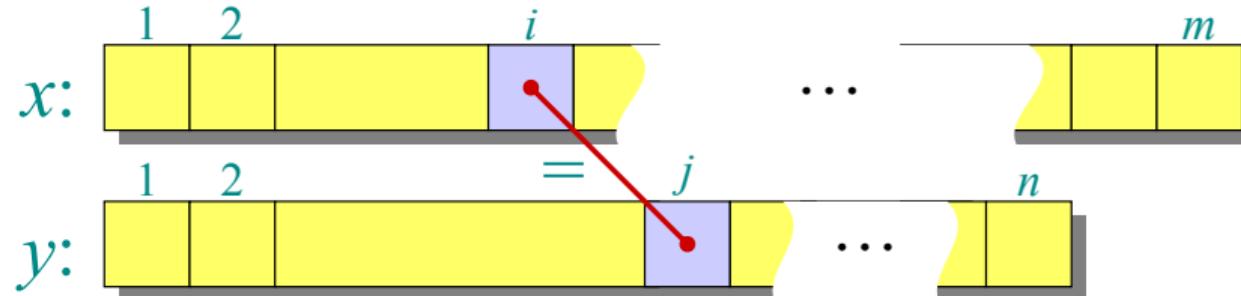
$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] & \text{if } x[i] = y[j], \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{otherwise.} \end{cases}$$

Proof: Case $x[i] = y[j] \dots$



Recursive formulation

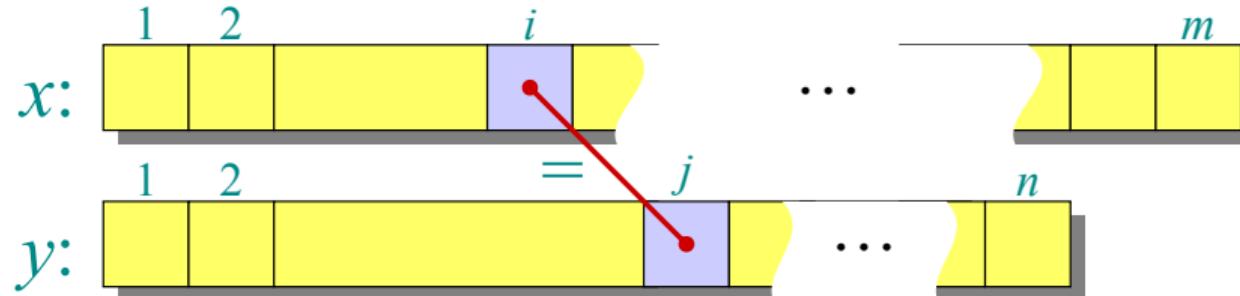
Proof: Case $x[i] = y[j] \dots$



- ▶ Let $z[1 \dots k] = LCS(x[1 \dots i], y[1 \dots j])$, where $c[i, j] = k$.

Recursive formulation

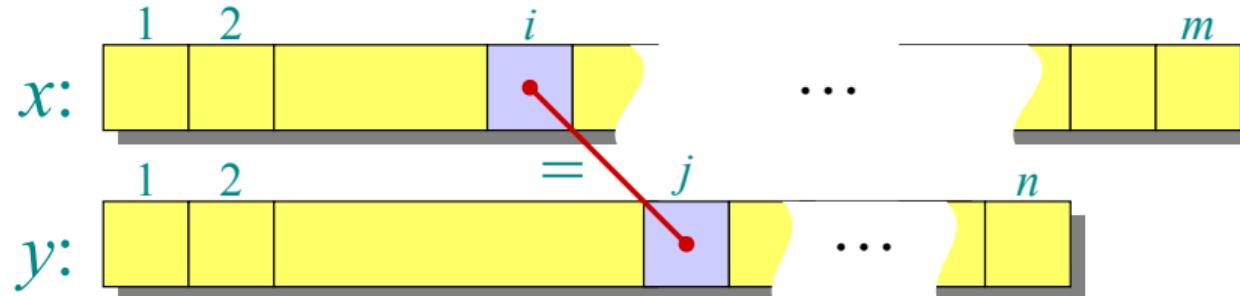
Proof: Case $x[i] = y[j] \dots$



- ▶ Let $z[1 \dots k] = LCS(x[1 \dots i], y[1 \dots j])$, where $c[i, j] = k$.
- ▶ Then, $z[k] =$

Recursive formulation

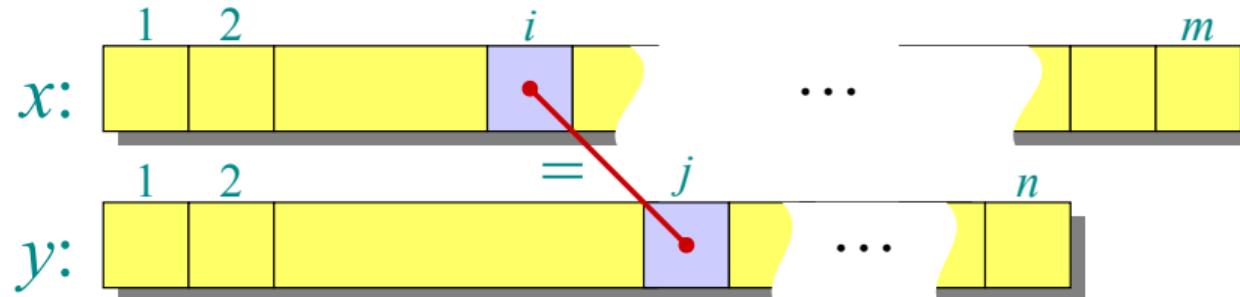
Proof: Case $x[i] = y[j] \dots$



- ▶ Let $z[1 \dots k] = LCS(x[1 \dots i], y[1 \dots j])$, where $c[i, j] = k$.
- ▶ Then, $z[k] = x[i](= y[j])$

Recursive formulation

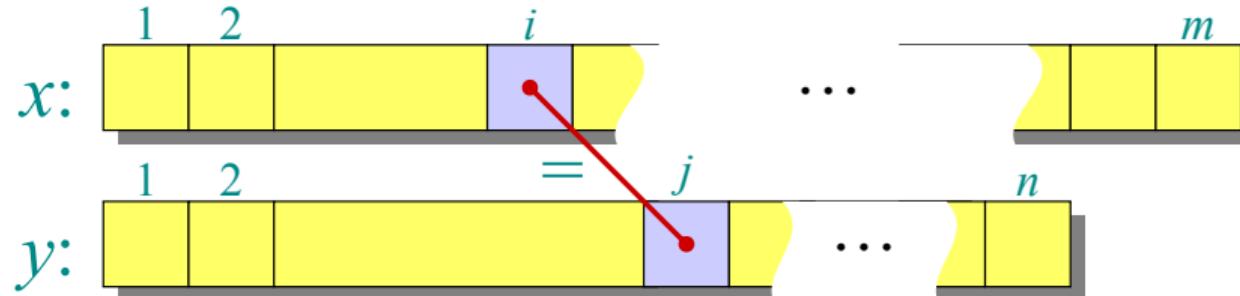
Proof: Case $x[i] = y[j] \dots$



- ▶ Let $z[1 \dots k] = LCS(x[1 \dots i], y[1 \dots j])$, where $c[i, j] = k$.
- ▶ Then, $z[k] = x[i](= y[j])$, or else z could be extended.

Recursive formulation

Proof: Case $x[i] = y[j] \dots$



- ▶ Let $z[1 \dots k] = LCS(x[1 \dots i], y[1 \dots j])$, where $c[i, j] = k$.
- ▶ Then, $z[k] = x[i](= y[j])$, or else z could be extended.
- ▶ Thus, $z[1 \dots k-1]$ is CS of $x[1 \dots i-1]$ and $y[1 \dots j-1]$.

Step-by-Step Example of LCS Algorithm

We will use the dynamic programming (DP) approach to compute the LCS for the following strings:

X = “ACDBE”

Y = “ABCDE”

Step 1: Create a DP Table

- ▶ We construct a $(m + 1) \times (n + 1)$ table, where m and n are the lengths of X and Y , respectively. The table will store the LCS length at each step.
- ▶ **Initial Table (before computation):** We initialize the first row and first column with 0s (LCS of empty strings is 0).

Step 1: Create a DP Table

	\emptyset	A	B	C	D	E
\emptyset	0	0	0	0	0	0
A	0					
C	0					
D	0					
B	0					
E	0					

Step 2: Fill the DP Table Using Recurrence

We iterate over each character of X and Y and apply the recurrence:

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

	\emptyset	A	B	C	D	E
\emptyset	0	0	0	0	0	0
A	0					
C	0					
D	0					
B	0					
E	0					

Step 2: Fill the DP Table Using Recurrence

Filling the table...

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

- ▶ Compare ‘A’ ($X[1]$) with each character in Y .
- ▶ ‘A’ matches ‘A’ $\rightarrow L[1, 1] = L[0, 0] + 1 = 1$.
- ▶ Other cells get the max of left or top.

	\emptyset	A	B	C	D	E
\emptyset	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0					
D	0					
B	0					
E	0					

Step 2: Fill the DP Table Using Recurrence

Filling the table...

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

- ▶ Compare ‘C’ ($X[2]$) with each character in Y .
- ▶ ‘C’ matches ‘C’ $\rightarrow L[2, 3] = L[1, 2] + 1 = 2$.
- ▶ Other cells get the max of left or top.

	\emptyset	A	B	C	D	E
\emptyset	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
D	0					
B	0					
E	0					

Step 2: Fill the DP Table Using Recurrence

Filling the table...

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

- ▶ Compare ‘D’ ($X[3]$) with each character in Y .
- ▶ ‘D’ matches ‘D’ $\rightarrow L[3, 4] = L[2, 3] + 1 = 3$.
- ▶ Other cells get the max of left or top.

	\emptyset	A	B	C	D	E
\emptyset	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
D	0	1	1	2	3	3
B	0					
E	0					

Step 2: Fill the DP Table Using Recurrence

Filling the table...

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

- ▶ Compare ‘B’ ($X[4]$) with each character in Y .
- ▶ ‘B’ matches ‘B’ $\rightarrow L[4, 2] = L[3, 1] + 1 = 2$.
- ▶ Other cells get the max of left or top.

	\emptyset	A	B	C	D	E
\emptyset	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
D	0	1	1	2	3	3
B	0	1	2	2	3	3
E	0					

Step 2: Fill the DP Table Using Recurrence

Filling the table...

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

- ▶ Compare ‘E’ ($X[5]$) with each character in Y .
- ▶ ‘E’ matches ‘E’ $\rightarrow L[5, 5] = L[4, 4] + 1 = 4$.
- ▶ Other cells get the max of left or top.

	\emptyset	A	B	C	D	E
\emptyset	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
D	0	1	1	2	3	3
B	0	1	2	2	3	3
E	0	1	2	2	3	4

Step 3: Extract the LCS

- ▶ The **LCS** length is in the bottom-right cell, $L[5, 5] = 4$.
- ▶ To trace back the **LCS**, we start from $L[5, 5]$ and follow:
 - ▶ If $X[i] = Y[j]$, include it in the **LCS**.
 - ▶ Otherwise, move to the maximum value from the left or top.

Tracing Back from $L[5,5]$:

1. E ($X[5] = Y[5]$) → Add ‘E’
2. D ($X[3] = Y[4]$) → Add ‘D’
3. C ($X[2] = Y[3]$) → Add ‘C’
4. A ($X[1] = Y[1]$) → Add ‘A’

Thus, the LCS = “ACDE”.

Longest Common Subsequence – Formal Steps of DP

Step 1: Characterizing a longest common subsequence

- ▶ You can solve the LCS problem with a brute-force approach:
 1. Enumerate all subsequences of X.
 2. Check each subsequence to see whether it is also subsequence of Y.
 3. Keep track of the longest subsequence.
- ▶ Because X has 2^m possible subsequence, time is exponential and, ergo, impractical.

Longest Common Subsequence – Formal Steps of DP

Step 1: Characterizing a longest common subsequence

- ▶ The LCS problem has an optimal-substructure property.
- ▶ the natural classes of subproblems correspond to pairs of “*prefixes*” of the two input sequences:
 - ▶ Given the sequence $X = \langle x_1, x_2, \dots, x_m \rangle$:
 - ▶ We can define the *i-th prefix* of X, for $i = 0, 1, \dots, m$, as $X_i = \langle x_1, x_2, \dots, x_i \rangle$.
 - ▶ For instance, if $X = \langle ABCBDAB \rangle$, then $X_4 = \langle ABCB \rangle$ and X_0 in the empty sequence.

Longest Common Subsequence – Formal Steps of DP

Step 1: Characterizing a longest common subsequence

Theorem 14.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1} .

Proof in CLRS page 395.

Longest Common Subsequence – Formal Steps of DP

Step 1: Characterizing a longest common subsequence

Theorem 14.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is a LCS of X_{m-1} and Y .

Proof in CLRS page 395.

Longest Common Subsequence – Formal Steps of DP

Step 1: Characterizing a longest common subsequence

Theorem 14.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is a LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is a LCS of X and Y_{n-1} .

Proof in CLRS page 395.

Longest Common Subsequence – Formal Steps of DP

Step 1: Characterizing a longest common subsequence

- ▶ Theorem 14.1 says that an LCS of two sequences contains within it an LCS of prefixes of the two sequences.
- ▶ Thus, the LCS problem has an optimal-substructure property.
- ▶ A recursive solution also has the overlapping-subproblems property (as we'll see in a moment).

Longest Common Subsequence – Formal Steps of DP

Step 2: A recursive solution

- ▶ To find an LCS of X and Y , you might need to find the LCSSs of X and Y_{n-1} and of X_{m-1} and Y (overlapping property).
- ▶ Each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1} .
- ▶ Many other subproblems share subsubproblems.

Longest Common Subsequence – Formal Steps of DP

Step 2: A recursive solution

The optimal substructure of the LCS problem gives the recursive formula:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x[i] = y[j], \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x[i] \neq y[j]. \end{cases}$$

Longest Common Subsequence – Formal Steps of DP

Step 2: A recursive solution

The optimal substructure of the LCS problem gives the recursive formula:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x[i] = y[j], \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x[i] \neq y[j]. \end{cases}$$

- ▶ Define $c[i, j]$ = length of LCS of X_i and Y_j . Want $c[m, n]$.

Longest Common Subsequence – Formal Steps of DP

Step 2: A recursive solution

The optimal substructure of the LCS problem gives the recursive formula:

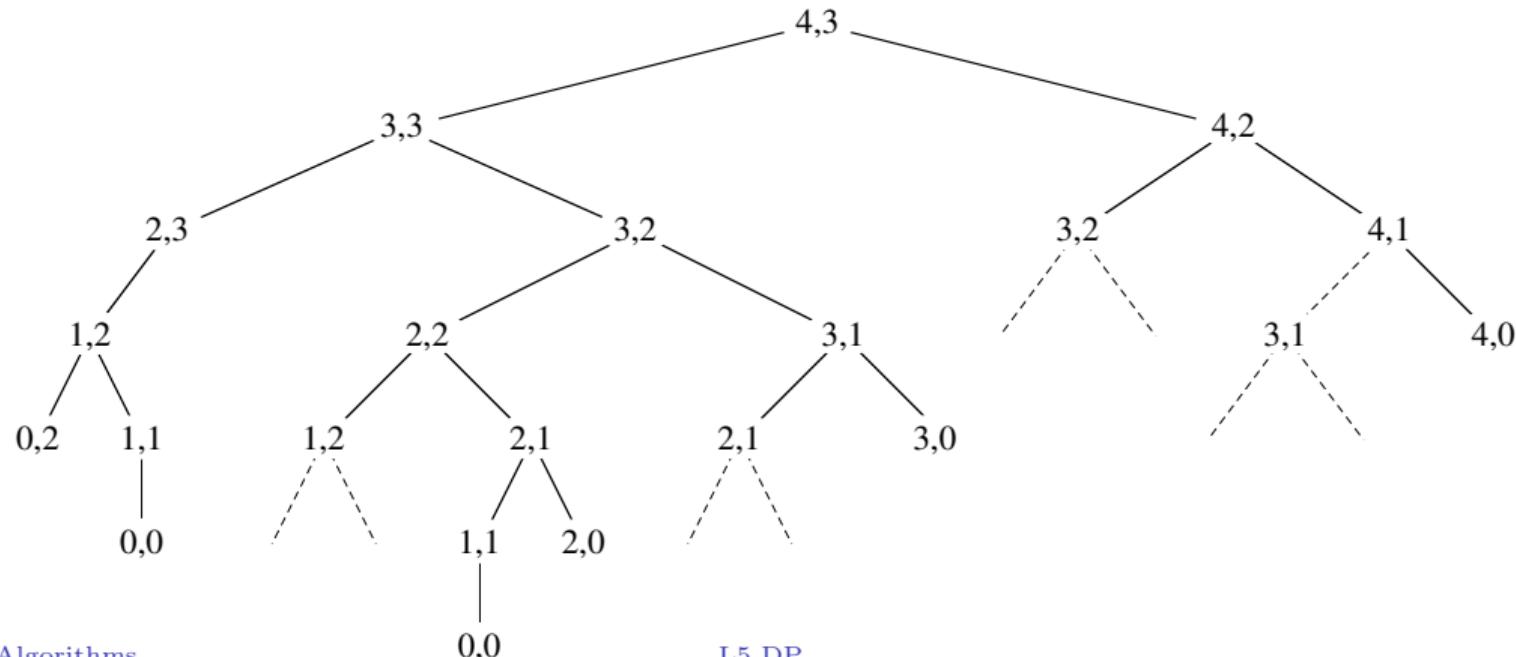
$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x[i] = y[j], \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x[i] \neq y[j]. \end{cases}$$

- ▶ Define $c[i, j]$ = length of LCS of X_i and Y_j . Want $c[m, n]$.
- ▶ Again, could write a recursive algorithm based on this formulation, but...

Longest Common Subsequence – Formal Steps of DP

Step 2: A recursive solution

- ▶ Try with $X = \langle a, t, o, m \rangle$ and $Y = \langle a, n, t \rangle$.
- ▶ Numbers in nodes are values of i, j in each recursive call.
- ▶ Dashed lines indicate subproblems already computed.



Longest Common Subsequence – Formal Steps of DP

Step 3: Computing the length of an LCS

LCS-LENGTH(X, Y, m, n)

```
1  let  $b[1 : m, 1 : n]$  and  $c[0 : m, 0 : n]$  be new tables
2  for  $i = 1$  to  $m$ 
3     $c[i, 0] = 0$ 
4  for  $j = 0$  to  $n$ 
5     $c[0, j] = 0$ 
6  for  $i = 1$  to  $m$            // compute table entries in row-major order
7    for  $j = 1$  to  $n$ 
8      if  $x_i == y_j$ 
9         $c[i, j] = c[i - 1, j - 1] + 1$ 
10        $b[i, j] = "\nwarrow"$ 
11     elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
12        $c[i, j] = c[i - 1, j]$ 
13        $b[i, j] = "\uparrow"$ 
14     else  $c[i, j] = c[i, j - 1]$ 
15        $b[i, j] = "\leftarrow"$ 
16  return  $c$  and  $b$ 
```

Longest Common Subsequence – Formal Steps of DP

Step 3: Computing the length of an LCS

```
PRINT-LCS( $b, X, i, j$ )
1  if  $i == 0$  or  $j == 0$ 
2    return          // the LCS has length 0
3  if  $b[i, j] == \nwarrow$ 
4    PRINT-LCS( $b, X, i - 1, j - 1$ )
5    print  $x_i$           // same as  $y_j$ 
6  elseif  $b[i, j] == \uparrow$ 
7    PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )
```

Longest Common Subsequence – Formal Steps of DP

Step 3: Computing the length of an LCS



Thomas H. Cormen

Emeritus Professor, Department of Computer Science

New Hampshire State Representative, Grafton 15 (Lebanon Ward 3)

CormenForNH.com

I retired on January 1, 2022. I am no longer taking any new students or interns at any level.

Ph.D., Massachusetts Institute of Technology, 1992.

- [Papers](#)
- [FG](#)
- [Other software](#)
- [Vita](#)

I taught my final course in Fall 2019. You can view a [video of my last lecture](#), which was not about computer science.

Khan Academy now carries [algorithms tutorials](#) for which [Devin Balkcom](#) and I produced content.

Are you looking for solutions to exercises and problems in *Introduction to Algorithms*?

If you are, then see the [frequently asked question and answer](#) below.

If you request solutions from me, I will not respond.

From July 2004 through June 2008, I was the director of the [Dartmouth Institute for Writing and Rhetoric](#).

I occasionally taught a graduate Computer Science course on how to write papers and how to give talks. I publish a [list of usage rules](#) that I required my students to observe.

In 2015, PRT's "The World" ran a [story on mentoring women in computer science](#) in which a couple of my students and I were interviewed.

I was interviewed for the Command Line Heroes podcast "[Learning the BASICS](#)."

You can also hear me spout off in "[Philosophical Trials #7](#)."

I talked about writing *Introduction to Algorithms* in an episode of "[Frank Stajano Explains](#)."

And an interview with a Brazilian vlogger.

Graduate Student Alumni

- [Alex Colvin](#), Ph.D. 1999
- [Peter C. Johnson](#)
- [Priya Narajan](#), Ph.D. 2011 [[Photo of Priya and me at 2012 Dartmouth graduation](#)]
- [Geeta Chaudhry Petrovic](#), Ph.D. 2004 [[Photo of Geeta and me at 2004 Dartmouth graduation](#)]
- [Elena Riccio Strange](#) (formerly Elena Riccio Davidson), Ph.D. 2006
- [Len Wisniewski](#), Ph.D. 1996
- [Melissa Hirschl Chawla](#), M.S. 1997
- [Michael Ringenburg](#), M.S. 2001
- [Georgi Vassilev](#), M.S. 1994

<https://www.cs.dartmouth.edu/~thc/>

Implementations [here](#)

Longest Common Subsequence – Formal Steps of DP

Step 3: Computing the length of an LCS

The screenshot shows a terminal window titled "Terminator" with two tabs open. The current tab displays the output of a command-line session:

```
Mar 31 21:42
and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14$ ls -laht
total 48K
drwx----- 3 and and 4.0K Mar 31 21:31 .
drwx----- 2 and and 4.0K Mar 31 17:10 __pycache__
drwx----- 31 and and 4.0K Mar 31 17:10 ..
-rw----- 1 and and 4.6K Oct 11 2021 optimal_BST.py
-rw----- 1 and and 3.4K Oct 11 2021 print_table.py
-rw----- 1 and and 6.5K Oct 11 2021 cut_rod.py
-rw----- 1 and and 4.7K Oct 11 2021 longest_common_subsequence.py
-rw----- 1 and and 7.3K Oct 11 2021 matrix_chain_multiply.py
(base) and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14$ python3 longest_common_subsequence.py
BCBA
0 0 0 0 0 0
0 0 0 0 1 1 1
0 1 1 1 1 2 2
0 1 1 2 2 2 2
0 1 1 2 2 3 3
0 1 2 2 2 3 3
0 1 2 2 3 3 4
0 1 2 2 3 4 4
↑ ↑ ↑ ↖ ↙ ↖
↖ ↙ ↘ ↗ ↘ ↗
↑ ↑ ↖ ↙ ↖ ↗
↖ ↙ ↘ ↗ ↘ ↗
↑ ↑ ↖ ↙ ↖ ↗
↖ ↙ ↘ ↗ ↘ ↗
GTCGTCTGGAAAGCCGCCGAA
(base) and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14$
```

The terminal shows the execution of the `longest_common_subsequence.py` script, which prints the sequence "BCBA". Below this, it shows the step-by-step construction of the LCS using arrows indicating matches between two sequences. The final sequence is "GTCGTCTGGAAAGCCGCCGAA". The bottom status bar indicates "Plain Text" and "Tab Width: 8".

Longest Common Subsequence – Formal Steps of DP

Step 4: Constructing an LCS

		j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A		
0	x_i	0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	-1	-1	-1
2	B	0	1	-1	-1	1	2	-2	
3	C	0	1	1	-2	-2	2	2	2
4	B	0	1	1	2	2	3	-3	
5	D	0	1	2	2	2	3	3	3
6	A	0	1	2	2	3	3	4	
7	B	0	1	2	2	3	4	4	4

Figure 14.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$ —the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y . For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i - 1, j]$, $c[i, j - 1]$, and $c[i - 1, j - 1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner, as shown by the sequence shaded blue. Each “ \nwarrow ” on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Homework

Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and
 $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

Plan

Dynamic Programming

N-th Fibonacci Number

Longest Palindromic Sequence

Longest Common Subsequence

Rod Cutting Problem

All-pairs shortest paths problem

Rod Cutting Problem

Problem Statement:

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \dots, n$, determine the maximum revenue obtainable by cutting the rod and selling the pieces.

Example Price Table:

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

Problem Statement

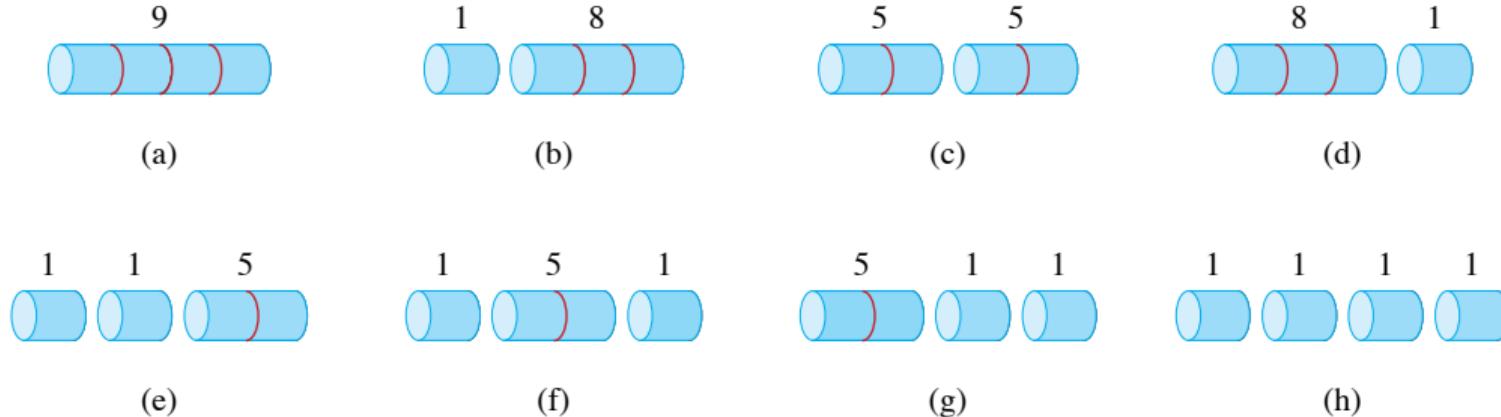


Figure 14.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 14.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Recursive Formulation

Let r_n be the maximum revenue for a rod of length n . Then:

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

Base case: $r_0 = 0$

Exponential Time Algorithm:

CUT-ROD(p, n)

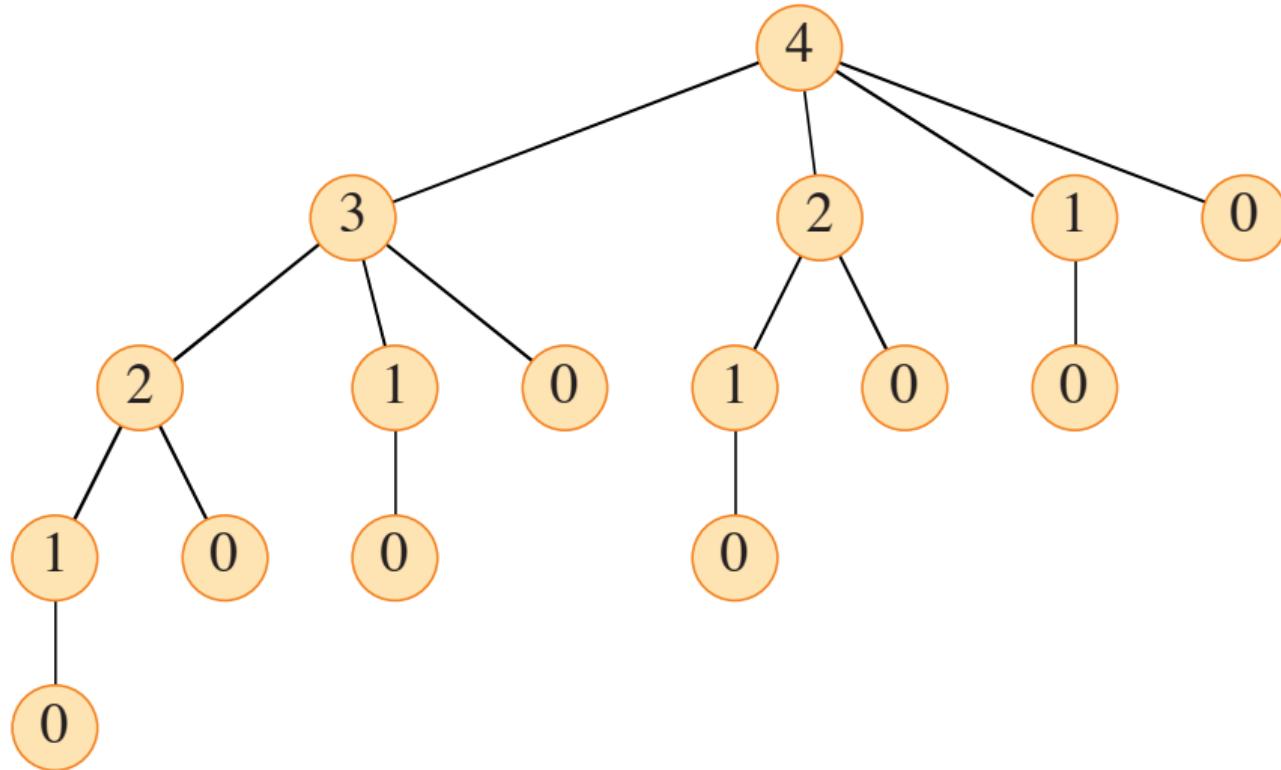
```
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max \{q, p[i] + \text{CUT-ROD}(p, n - i)\}$ 
6  return  $q$ 
```

Recursive Formulation

- ▶ Let $T(n)$ denote the total number of calls made to $\text{CUT-ROD}(p, n)$ for a particular value of n .
- ▶ The number of nodes in a subtree whose root is labeled n in the recursion tree.

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

Recursive Formulation



Memoized Version

Avoids recomputation by storing already computed values.

MEMOIZED-CUT-ROD(p, n)

```
1 let  $r[0:n]$  be a new array      // will remember solution values in  $r$ 
2 for  $i = 0$  to  $n$ 
3    $r[i] = -\infty$ 
4 return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

MEMOIZED-CUT-ROD-AUX(p, n, r)

```
1 if  $r[n] \geq 0$             // already have a solution for length  $n$ ?
2   return  $r[n]$ 
3 if  $n == 0$ 
4    $q = 0$ 
5 else  $q = -\infty$ 
6   for  $i = 1$  to  $n$     //  $i$  is the position of the first cut
7      $q = \max\{q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r)\}$ 
8    $r[n] = q$               // remember the solution value for length  $n$ 
9   return  $q$ 
```

Bottom-Up Dynamic Programming

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

```
1  let  $r[0:n]$  be a new array      // will remember solution values in  $r$ 
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$           // for increasing rod length  $j$ 
4     $q = -\infty$ 
5    for  $i = 1$  to  $j$           //  $i$  is the position of the first cut
6       $q = \max\{q, p[i] + r[j-i]\}$ 
7     $r[j] = q$                   // remember the solution value for length  $j$ 
8  return  $r[n]$ 
```

Time complexity: $\mathcal{O}(n^2)$

Subproblem graphs

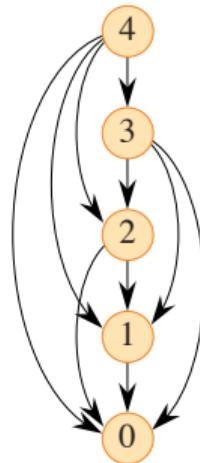


Figure 14.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that solving subproblem x requires a solution to subproblem y . This graph is a reduced version of the recursion tree of Figure 14.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

Reconstructing the Solution

Store the optimal cuts as well:

- ▶ Maintain a second array $s[1..n]$.
- ▶ $s[j]$ stores the length of the first piece to cut from a rod of length j .

Then backtrack using $s[n]$ to print the actual cuts.

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

Reconstructing a solution

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1  let  $r[0 : n]$  and  $s[1 : n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$            // for increasing rod length  $j$ 
4     $q = -\infty$ 
5    for  $i = 1$  to  $j$       //  $i$  is the position of the first cut
6      if  $q < p[i] + r[j - i]$ 
7         $q = p[i] + r[j - i]$ 
8         $s[j] = i$             // best cut location so far for length  $j$ 
9         $r[j] = q$             // remember the solution value for length  $j$ 
10   return  $r$  and  $s$ 
```

Reconstructing a solution

```
and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14
and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14 136x31
(base) and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14$ python3 cut_rod.py
10
30
10
30
10
30
2 2
10

[0, 1, 5, 8, 10, 13, 17, 18, 22, 25, 30]
[None, 1, 2, 3, 2, 2, 6, 1, 2, 3, 10]

[ 0  1   3   8   9 10 11 12 13 15 18 20 21 23 26 29 30 31 32 34 35 36 37 39]
48
25
48
25
48
25
3 3 3 3 3 3
1 3 3 3
(base) and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14$
```

Learned lessons

- ▶ Naive recursive solution has exponential time complexity.
- ▶ Dynamic programming reduces it to $\mathcal{O}(n^2)$.
- ▶ Optimal substructure and overlapping subproblems make DP suitable.

Plan

Dynamic Programming

N-th Fibonacci Number

Longest Palindromic Sequence

Longest Common Subsequence

Rod Cutting Problem

All-pairs shortest paths problem

Introduction

Different types of algorithms can be used to solve the all-pairs shortest paths problem:

- ▶ Dynamic programming
- ▶ Matrix multiplication
- ▶ Floyd-Warshall algorithm
- ▶ Johnson's algorithm
- ▶ Difference constraints

Single-Source Shortest Paths

- ▶ Given directed graph $G = (V, E)$, vertex $s \in V$ and edge weights $w : E \rightarrow \mathbb{R}$
- ▶ Find $\delta(s, v)$, equal to the shortest-path weight $s \rightarrow v, \forall v \in V$ (or $-\infty$ if negative weight cycle along the way, or ∞ if no path)

Single-Source Shortest Paths

- ▶ Given directed graph $G = (V, E)$, vertex $s \in V$ and edge weights $w : E \rightarrow \mathbb{R}$
- ▶ Find $\delta(s, v)$, equal to the shortest-path weight $s \rightarrow v, \forall v \in V$ (or $-\infty$ if negative weight cycle along the way, or ∞ if no path)

Situation	Algorithm	Time
unweighted ($w = 1$)	BFS	$O(V + E)$
non-negative edge weights	Dijkstra	$O(E + V \lg V)$
general	Bellman-Ford	$O(VE)$
acyclic graph (DAG)	Topological sort + one pass of B-F	$O(V + E)$

All of the above results are the best known. We achieve a $O(E + V \lg V)$ bound on Dijkstra's algorithm using **Fibonacci heaps**.

All-Pairs Shortest Paths (APSP)

- ▶ Given edge-weighted graph, $G = (V, E, w)$.
- ▶ Find $\delta(u, v)$ for all $u, v \in V$.
- ▶ A simple way of solving APSP problems is by running a single-source shortest path algorithm from each of the V vertices in the graph.

All-Pairs Shortest Paths (APSP)

- ▶ Given edge-weighted graph, $G = (V, E, w)$.
- ▶ Find $\delta(u, v)$ for all $u, v \in V$.
- ▶ A simple way of solving APSP problems is by running a single-source shortest path algorithm from each of the V vertices in the graph.

Situation	Algorithm	Time	$E = \Theta(V^2)$
unweighted ($w = 1$)	$ V + \text{BFS}$	$O(VE)$	$O(V^3)$
non-negative edge weights	$ V + \text{Dijkstra}$	$O(VE + V^2 \lg V)$	$O(V^3)$
general	$ V + \text{Bellman-Ford}$	$O(V^2 E)$	$O(V^4)$
general	Johnson's	$O(VE + V^2 \lg V)$	$O(V^3)$

These results (apart from the third) are also best known –don't know how to beat $|V| \times \text{Dijkstra}$.

Algorithms to solve APSP[§]

Dynamic Programming (attempt 1):

1. **Sub-problems:** $d_{uv}^{(m)}$ = weight of shortest path $u \rightarrow v$ using $\leq m$ edges.

2. **Guessing:** What's the last edge (x, v) ?

3. **Recurrence:**

$$d_{uv}^{(m)} = \min(d_{uv}^{(m-1)} + w(x, v) \quad \text{for } x \in V)$$

$$d_{uv}^{(0)} = \begin{cases} 0 & \text{if } u = v. \\ \infty & \text{otherwise.} \end{cases}$$

4. **Topological ordering:** for $m = 0, 1, 2, \dots, n - 1$: for u and v in V .

5. **Original problem:** If graph contains no negative-weight cycles (by Bellman-Ford analysis), then shortest path is simple $\Rightarrow \delta(u, v) = d_{uv}^{(n-1)} = d_{uv}^{(n)} = \dots$

§ For all the algorithms described, we assume that $w(u, v) = \infty$ if $(u, v) \notin E$

Bottom-up via Relaxation Steps ¶

```
1  for  $m \leftarrow 1$  to  $n$  by 1
2      for  $u$  in  $V$ 
3          for  $v$  in  $V$ 
4              for  $x$  in  $V$ 
5                  if  $d_{uv} > d_{ux} + d_{xv}$ 
6                       $d_{uv} = d_{ux} + d_{xv}$ 
```

¶ In the above pseudocode, we omit superscripts because more relaxation can never hurt.

Time complexity

- ▶ In this Dynamic Program, we have $O(V^3)$ total sub-problems.
- ▶ Each sub-problem takes $O(V)$ time to solve, since we need to consider V possible choices.
- ▶ This gives a total runtime complexity of $O(V^4)$.
- ▶ Note that this is no better than $|V| \times$ Bellman-Ford.

Matrix Multiplication

Recall the task of standard matrix multiplication:

- ▶ Given $n \times n$ matrices A and B , compute $C = A \cdot B$, such that $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$.

Matrix Multiplication

Recall the task of standard matrix multiplication:

- ▶ Given $n \times n$ matrices A and B , compute $C = A \cdot B$, such that
$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}.$$
 1. $O(n^3)$ using standard algorithm.

Matrix Multiplication

Recall the task of standard matrix multiplication:

- ▶ Given $n \times n$ matrices A and B , compute $C = A \cdot B$, such that
$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}.$$
 1. $O(n^3)$ using standard algorithm.
 2. $O(n^{2.807})$ using Strassen algorithm.

Matrix Multiplication

Recall the task of standard matrix multiplication:

- ▶ Given $n \times n$ matrices A and B , compute $C = A \cdot B$, such that $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$.
 1. $O(n^3)$ using standard algorithm.
 2. $O(n^{2.807})$ using Strassen algorithm.
 3. $O(n^{2.376})$ using Coppersmith-Winograd algorithm.

Matrix Multiplication

Recall the task of standard matrix multiplication:

- ▶ Given $n \times n$ matrices A and B , compute $C = A \cdot B$, such that $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$.
 1. $O(n^3)$ using standard algorithm.
 2. $O(n^{2.807})$ using Strassen algorithm.
 3. $O(n^{2.376})$ using Coppersmith-Winograd algorithm.
 4. $O(n^{2.3728})$ using Vassilevska-Williams algorithm.

Connection to Shortest Paths

- ▶ Let's define $\oplus = \min$ and $\odot = +$.
- ▶ Then, $C = A \odot B$ produces $c_{ij} = \min_k(a_{ik} + b_{kj})$.
- ▶ Define $D^{(m)} = (d_{ij}^{(m)})$, $W = (w(i, j))$, $V = \{1, 2, \dots, n\}$

With the above definitions, we see that $D^{(m)}$ can be expressed as $D^{(m-1)} \odot W$. In other words, $D^{(m)}$ can be expressed as the circle-multiplication of W with itself m times.

Matrix Multiplication Algorithm

- ▶ $n - 2$ multiplications $\Rightarrow O(n^4)$ time (still no better).

Matrix Multiplication Algorithm

- ▶ $n - 2$ multiplications $\Rightarrow O(n^4)$ time (still no better).
- ▶ Repeated squaring: $((W^2)^2)^2 \cdots$

Matrix Multiplication Algorithm

- ▶ $n - 2$ multiplications $\Rightarrow O(n^4)$ time (still no better).
- ▶ Repeated squaring: $((W^2)^2)^2 \cdots = W^{2^{\lg n}}$

Matrix Multiplication Algorithm

- ▶ $n - 2$ multiplications $\Rightarrow O(n^4)$ time (still no better).
- ▶ Repeated squaring: $((W^2)^2)^2 \cdots = W^{2^{\lg n}} = W^{n-1} = (\delta(i, j))$ if no negative-weight cycles.

Matrix Multiplication Algorithm

- ▶ $n - 2$ multiplications $\Rightarrow O(n^4)$ time (still no better).
- ▶ Repeated squaring: $((W^2)^2)^2 \cdots = W^{2^{\lg n}} = W^{n-1} = (\delta(i, j))$ if no negative-weight cycles.
- ▶ Time complexity of this algorithm is now $O(n^3 \lg n)$.

Floyd-Warshall Algorithms

Dynamic Programming (attempt 2):

1. **Sub-problems:** $c_{uv}^{(k)}$ = weight of shortest path $u \rightarrow v$ whose intermediate vertices $\in \{1, 2, \dots, k\}$

Floyd-Warshall Algorithms

Dynamic Programming (attempt 2):

1. **Sub-problems:** $c_{uv}^{(k)}$ = weight of shortest path $u \rightarrow v$ whose intermediate vertices $\in \{1, 2, \dots, k\}$
2. **Guessing:**

Floyd-Warshall Algorithms

Dynamic Programming (attempt 2):

1. **Sub-problems:** $c_{uv}^{(k)}$ = weight of shortest path $u \rightarrow v$ whose intermediate vertices $\in \{1, 2, \dots, k\}$
2. **Guessing:** Does shortest path use vertex k ?

Floyd-Warshall Algorithms

Dynamic Programming (attempt 2):

1. **Sub-problems:** $c_{uv}^{(k)}$ = weight of shortest path $u \rightarrow v$ whose intermediate vertices $\in \{1, 2, \dots, k\}$
2. **Guessing:** Does shortest path use vertex k ?
3. **Recurrence:**

$$c_{uv}^{(0)} = w(u, v)$$

$$c_{uv}^{(k)} = \min(c_{uv}^{(k-1)}, c_{ux}^{(k-1)} + c_{xv}^{(k-1)})$$

Floyd-Warshall Algorithms

Dynamic Programming (attempt 2):

1. **Sub-problems:** $c_{uv}^{(k)}$ = weight of shortest path $u \rightarrow v$ whose intermediate vertices $\in \{1, 2, \dots, k\}$
2. **Guessing:** Does shortest path use vertex k ?
3. **Recurrence:**

$$c_{uv}^{(0)} = w(u, v)$$

$$c_{uv}^{(k)} = \min(c_{uv}^{(k-1)}, c_{ux}^{(k-1)} + c_{xv}^{(k-1)})$$

4. **Topological order:** for k : for u and v in V :
5. **Original problem:** $\delta(u, v) = c_{uv}^{(n)}$. Negative weight cycle \Leftrightarrow negative $c_{uu}^{(n)}$

Time Complexity

This Dynamic Program contains $O(V^3)$ problems as well. However, in this case, it takes only $O(1)$ time to solve each sub-problem, which means that the total runtime of this algorithm is $O(V^3)$.

Bottom-up via Relaxation

```
1   $C = (w(u, v))$ 
2  for  $k \leftarrow 1$  to  $n$ 
3      for  $u$  in  $V$ 
4          for  $v$  in  $V$ 
5              if  $c_{uv} > c_{uk} + c_{kv}$ 
6                   $c_{uv} = c_{uk} + c_{kv}$ 
```

Johnson's algorithm

1. Find function $h : V \rightarrow \mathbb{R}$ such that $w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$ for all $u, v \in V$ or determine that a negative-weight cycle exists.
2. Run Dijkstra's algorithm on (V, E, w_h) from every source vertex $s \in V \Rightarrow$ get $\delta_h(u, v)$ for all $u, v \in V$.
3. Given $\delta_h(u, v)$, it is easy to compute $\delta(u, v)$.

Proof

Claim. $\delta(u, v) = \delta_h(u, v) - h(u) + h(v)$.

Proof. Look at any $u \rightarrow v$ path p in the graph G :

- ▶ Say p is $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$, where $v_0 = u$ and $v_k = v$.

$$\begin{aligned} w_h(p) &= \sum_{i=1}^k w_h(v_{i-1}, v_i) \\ &= \sum_{i=1}^k [w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)] \\ &= \sum_{i=1}^k w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\ &= w(p) + h(u) - h(v) \end{aligned}$$

- ▶ Hence all $u \rightarrow v$ paths change in weight by the same offset $h(u) - h(v)$, which implies that the shortest path is preserved. □

How to find h ?

We know that

$$w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$$

This is equivalent to,

$$h(v) - h(u) \leq w(u, v)$$

for all $(u, v) \in V$. This is called a **system of difference constraints**.

Theorem.

If (V, E, w) has a negative-weight cycle, then there exists no solution to the above system of difference constraints.

Proof

Say $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$ is a negative weight cycle.

Let us assume to the contrary that the system of difference constraints has a solution; let's call it h . This gives us the following system of equations,

$$h(v_1) - h(v_0) \leq w(v_0, v_1)$$

$$h(v_2) - h(v_1) \leq w(v_1, v_2)$$

$$h(v_3) - h(v_2) \leq w(v_2, v_3)$$

⋮

$$h(v_k) - h(v_{k-1}) \leq w(v_{k-1}, v_k)$$

$$h(v_0) - h(v_k) \leq w(v_k, v_0)$$

Summing all these equations gives us

$$0 \leq w(\text{cycle}) < 0$$

which is obviously not possible.

From this, we can conclude that no solution to the above system of difference constraints exists if the graph (V, E, w) has a negative weight cycle. □

Theorem.

If (V, E, w) has no negative-weight cycle, then we can find a solution to the difference constraints.

Proof. Add a new vertex s to G , and add edges (s, v) of weight 0 for all $v \in V$.

- ▶ Clearly, these new edges do not introduce any new negative weight cycles to the graph.
- ▶ Adding these new edges ensures that there now exists at least one path from s to v . This implies that $\delta(s, v)$ is finite for all $v \in V$
- ▶ We now claim that $h(v) = \delta(s, v)$. This is obvious from the triangle inequality:
$$\delta(s, u) + w(u, v) \geq \delta(s, v) \Leftrightarrow \delta(s, v) - \delta(s, u) \leq w(u, v) \Leftrightarrow h(v) - h(u) \leq w(u, v).$$

□

Time Complexity

1. The first step involves running Bellman-Ford from s , which takes $O(VE)$ time. We also pay a pre-processing cost to reweight all the edges ($O(E)$).

Time Complexity

1. The first step involves running Bellman-Ford from s , which takes $O(VE)$ time. We also pay a pre-processing cost to reweight all the edges ($O(E)$).
2. We then run Dijkstra's algorithm from each of the V vertices in the graph; the total time complexity of this step is $O(VE + V^2 \lg V)$.

Time Complexity

1. The first step involves running Bellman-Ford from s , which takes $O(VE)$ time. We also pay a pre-processing cost to reweight all the edges ($O(E)$).
2. We then run Dijkstra's algorithm from each of the V vertices in the graph; the total time complexity of this step is $O(VE + V^2 \lg V)$.
3. We then need to reweight the shortest paths for each pair; this takes $O(V^2)$ time.

Time Complexity

1. The first step involves running Bellman-Ford from s , which takes $O(VE)$ time. We also pay a pre-processing cost to reweight all the edges ($O(E)$).
2. We then run Dijkstra's algorithm from each of the V vertices in the graph; the total time complexity of this step is $O(VE + V^2 \lg V)$.
3. We then need to reweight the shortest paths for each pair; this takes $O(V^2)$ time.

The total running time of this algorithm is $O(VE + V^2 \lg V)$.

End of Lecture 5.

TDT5FTOTTC



Top 5 Fundamental Takeaways

Top 5 Fundamental Takeaways

- 5 Computing Fibonacci numbers or finding the longest palindromic subsequence becomes efficient when subproblem results are cached to prevent repeated computation.

Top 5 Fundamental Takeaways

- 5 Computing Fibonacci numbers or finding the longest palindromic subsequence becomes efficient when subproblem results are cached to prevent repeated computation.
- 4 The LCS problem uses a table-based approach to find the length and content of the longest common subsequence in quadratic time.

Top 5 Fundamental Takeaways

- 5 Computing Fibonacci numbers or finding the longest palindromic subsequence becomes efficient when subproblem results are cached to prevent repeated computation.
- 4 The LCS problem uses a table-based approach to find the length and content of the longest common subsequence in quadratic time.
- 3 DP Has Four Key Steps: identifying the structure, defining recurrence, computing solutions bottom-up, and reconstructing the optimal result.

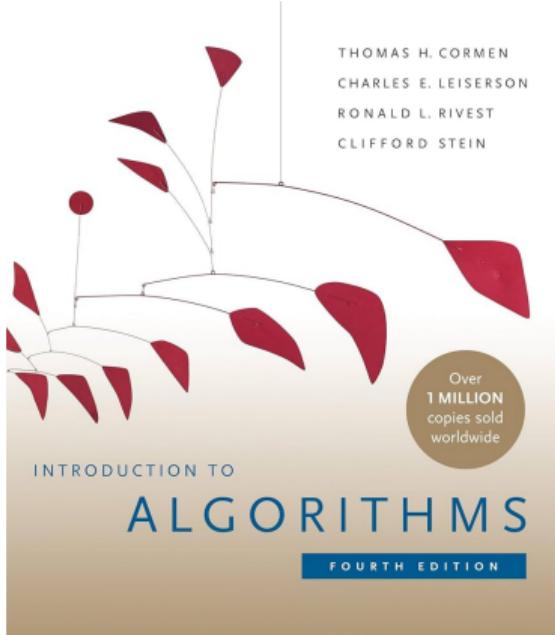
Top 5 Fundamental Takeaways

- 5 Computing Fibonacci numbers or finding the longest palindromic subsequence becomes efficient when subproblem results are cached to prevent repeated computation.
- 4 The LCS problem uses a table-based approach to find the length and content of the longest common subsequence in quadratic time.
- 3 DP Has Four Key Steps: identifying the structure, defining recurrence, computing solutions bottom-up, and reconstructing the optimal result.
- 2 DP optimizes problems that have **overlapping subproblems** and **optimal substructure**.

Top 5 Fundamental Takeaways

- 5 Computing Fibonacci numbers or finding the longest palindromic subsequence becomes efficient when subproblem results are cached to prevent repeated computation.
- 4 The LCS problem uses a table-based approach to find the length and content of the longest common subsequence in quadratic time.
- 3 DP Has Four Key Steps: identifying the structure, defining recurrence, computing solutions bottom-up, and reconstructing the optimal result.
- 2 DP optimizes problems that have **overlapping subproblems** and **optimal substructure**.
- 1 Dynamic Programming = recursion + memoization.

Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.

Visit <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.

Original slides from *Introduction to Algorithms* 6.046J/18.401J, Fall 2005 Class by Prof. Charles Leiserson and Prof. Erik Demaine. MIT OpenCourseWare Initiative available at
<https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/>.