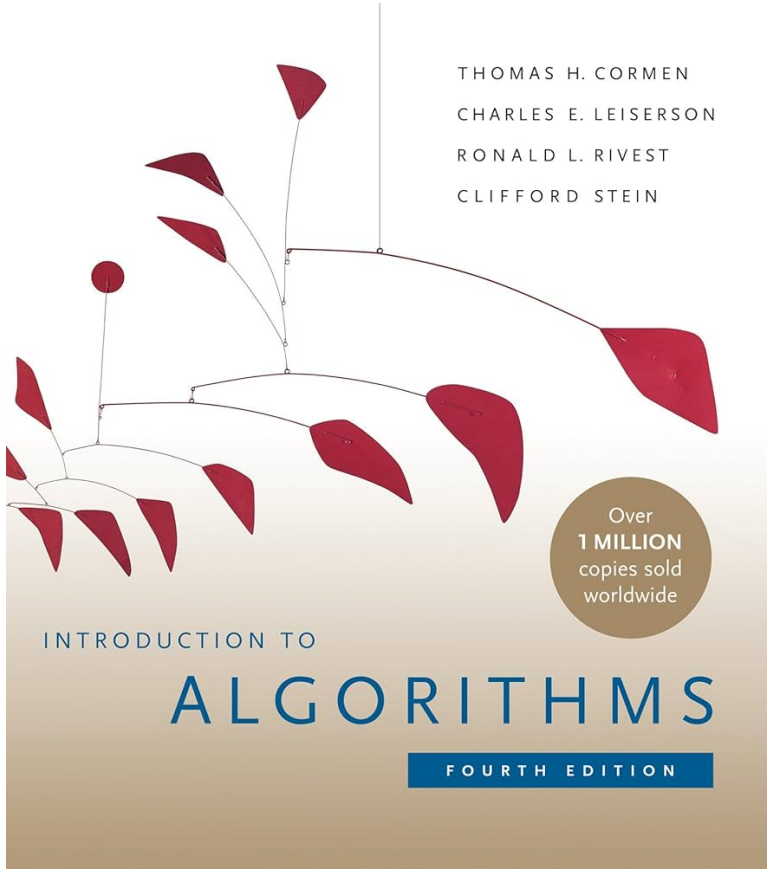
Introduction to Algorithms
Lecture 5: Dynamic Programming (DP)

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

March 18, 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

Dynamic Programming* (DP)

- Invented by Richard Bellman in 1950s.
- Desing 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.

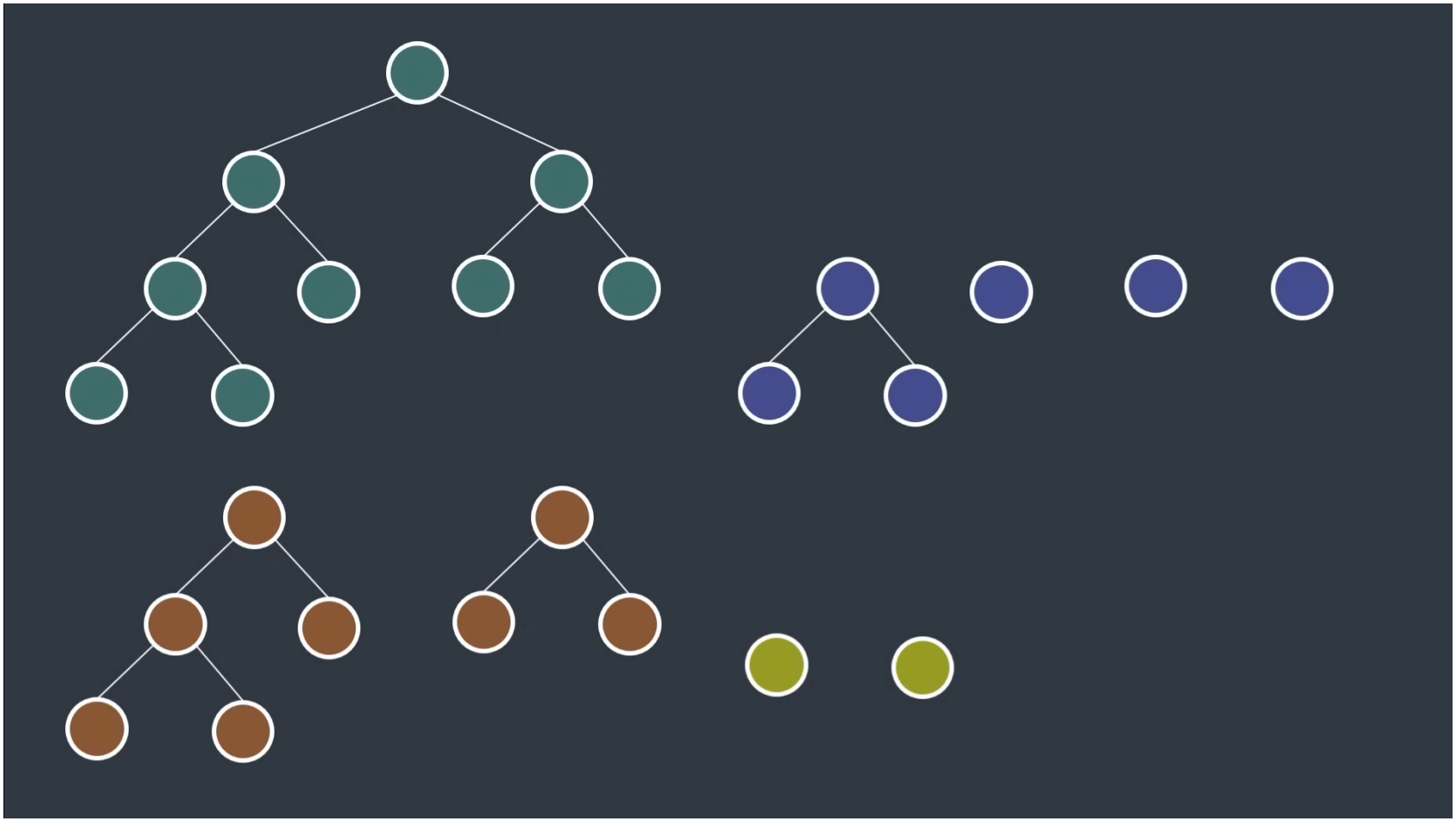
Programming in this context refers to a tabular method, not to writing computer code.

DP notions

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

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

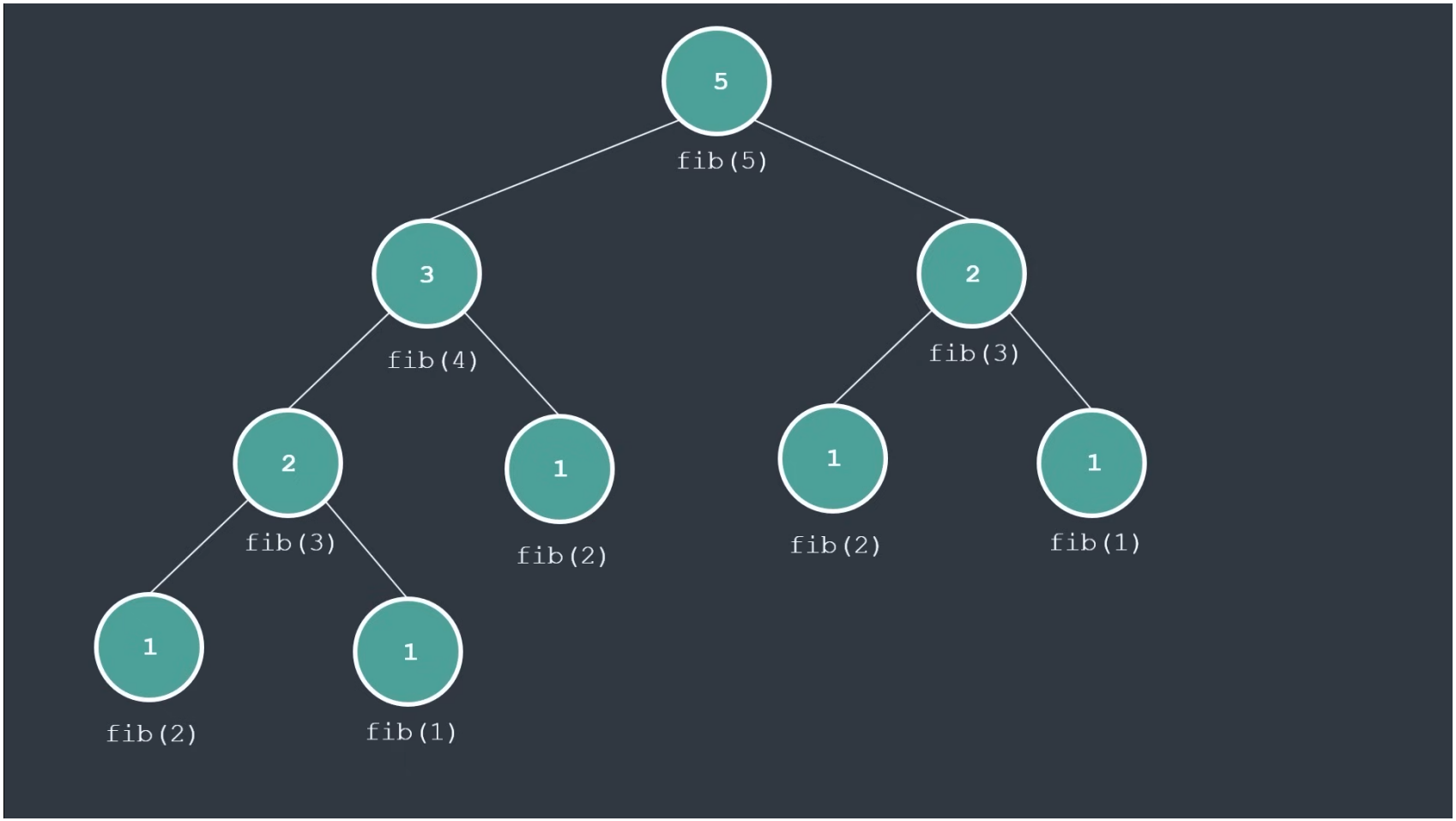
```
1    def fib(n):
2        if n <= 2:
3            result = 1
4        else:
5            result = fib(n - 1) + fib(n - 2)
6        return result
```

Naive Approach

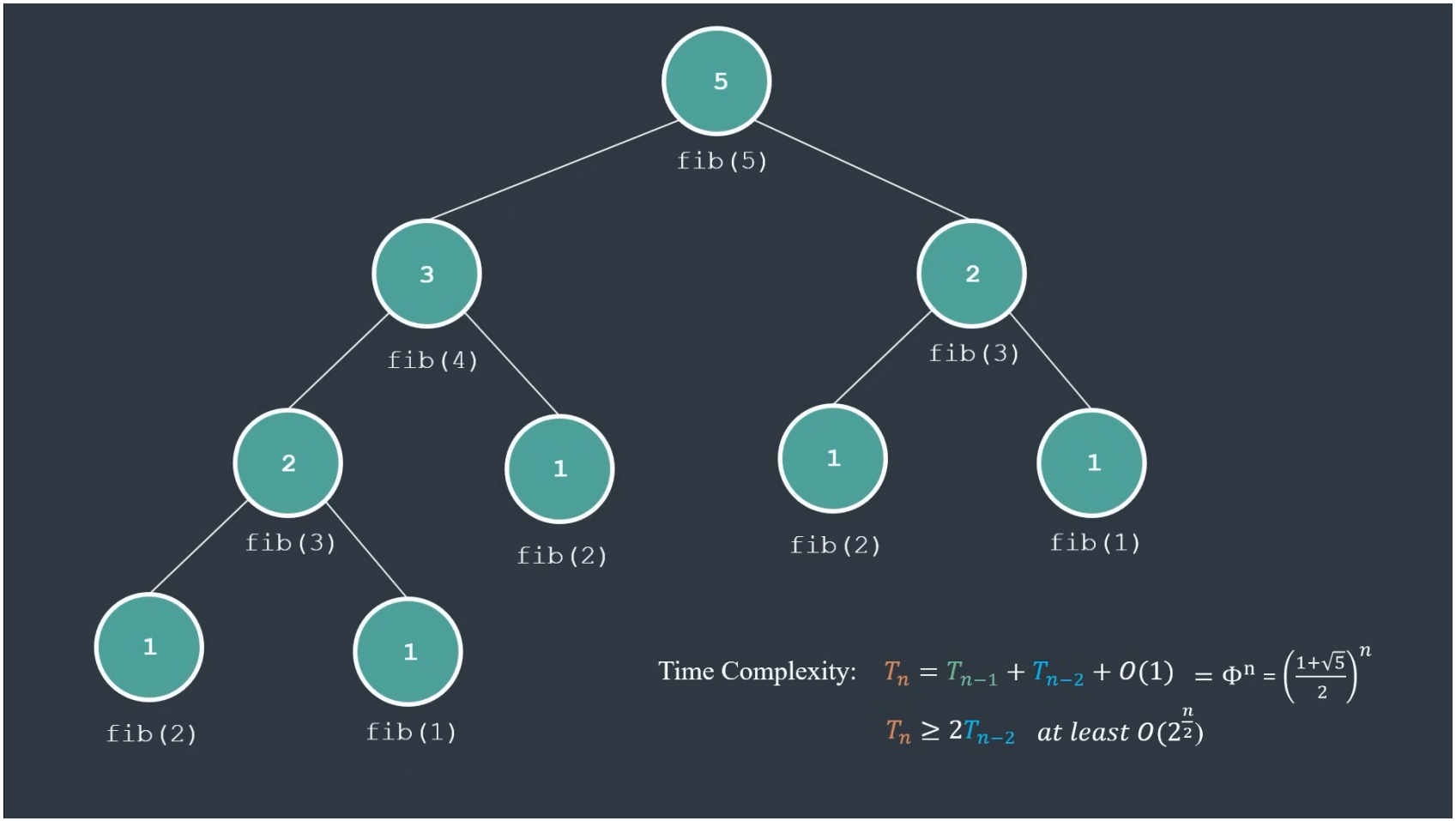
```
1    def fib(n):
2        if n <= 2:
3            result = 1
4        else:
5            result = fib(n - 1) + fib(n - 2)
6        return result
```

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

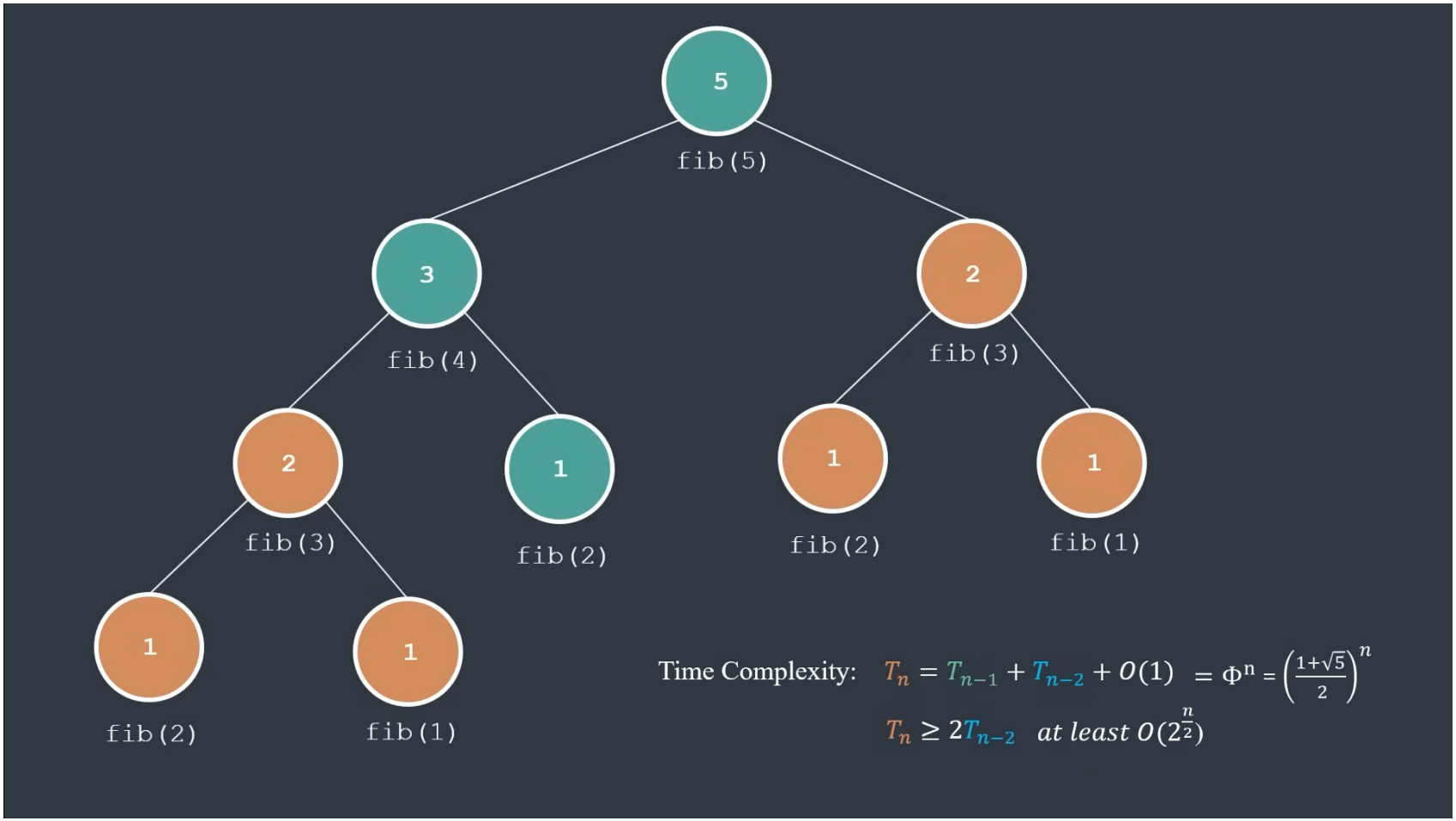
Naive Approach



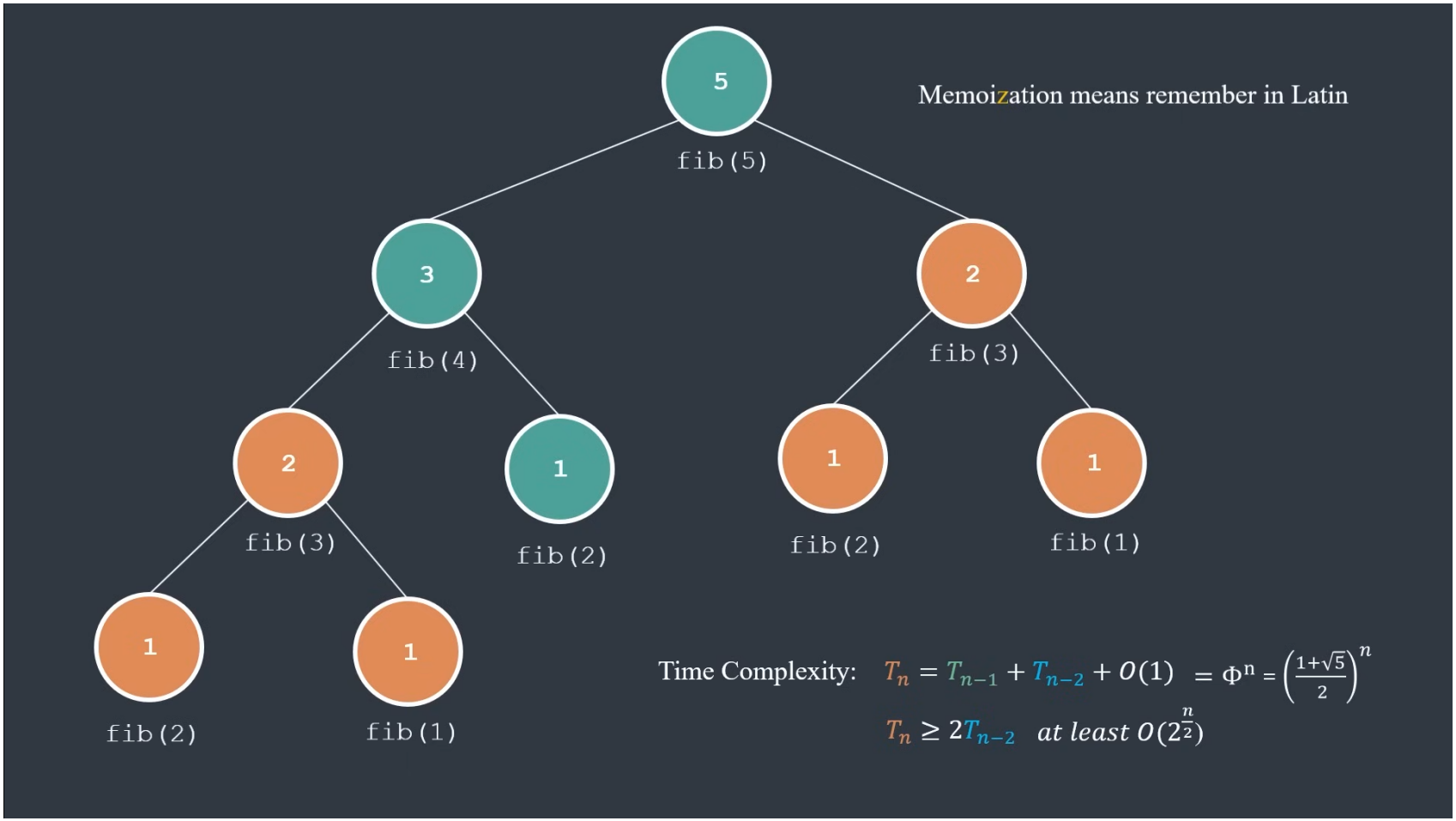
Naive Approach



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
```

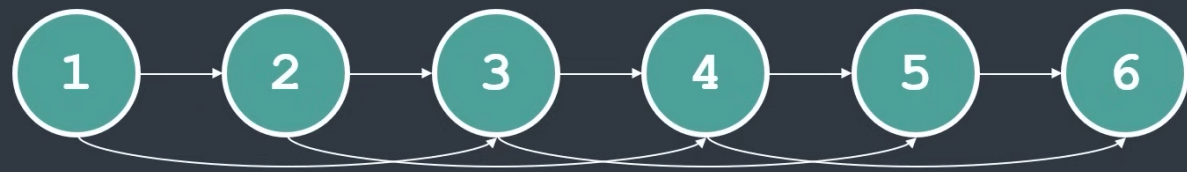
Time Complexity: $O(n)$.

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[n - 1] + memo[n - 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

Longest Palindromic Sequence

Definition:

A palindrome is a string that is unchanged when reversed.

- Examples: `radar`, `civic`, `t`, `bb`, `redder`.
- Given: A string $X[1 \ldots n]$, $n \geq 1$.
- 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 $X[i \ldots j]$ 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 + i, 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.

$$T(n) = \text{running time on input of length } n$$

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

$$= 2^{n-1}$$

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 + i, 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 + i, 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

- 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 $x[1 \cdots m]$ 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 $= O(n2^m)$

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.

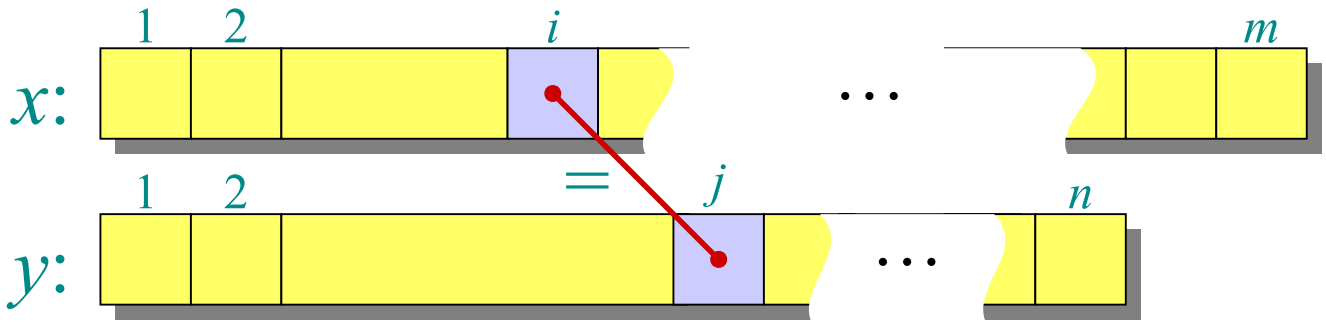
- **Notation:** Denote the length of a sequence s as $|s|$.
- **Strategy:** Consider **prefixes** of x and y:
  - Define $c[i, j] = |LCS(x[1 \cdots i], y[1 \cdots j])|$.
  - Then, $c[m, n] = |LCS(x, y)|$.

Recursive formulation

Theorem

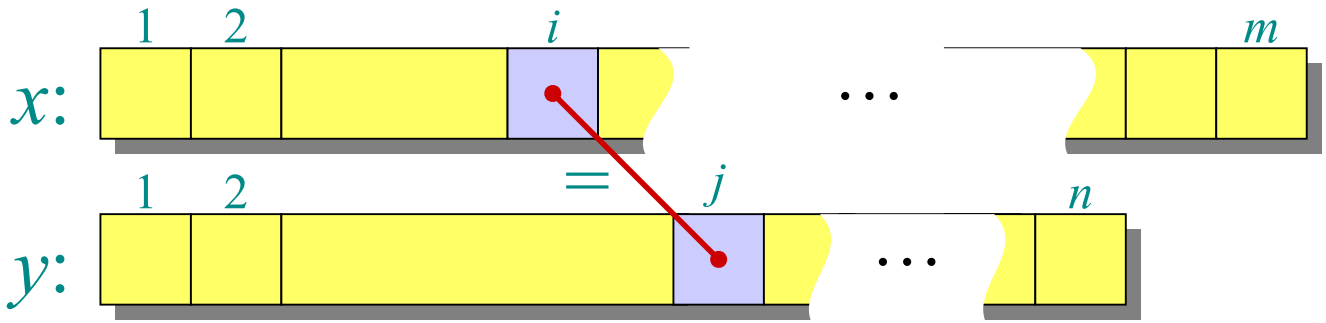
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] & \text{if } x[i] = y[j], \\ \max\{c[i, j-1], c[i-1, j]\} & \text{otherwise.} \end{cases}$$

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



Recursive formulation

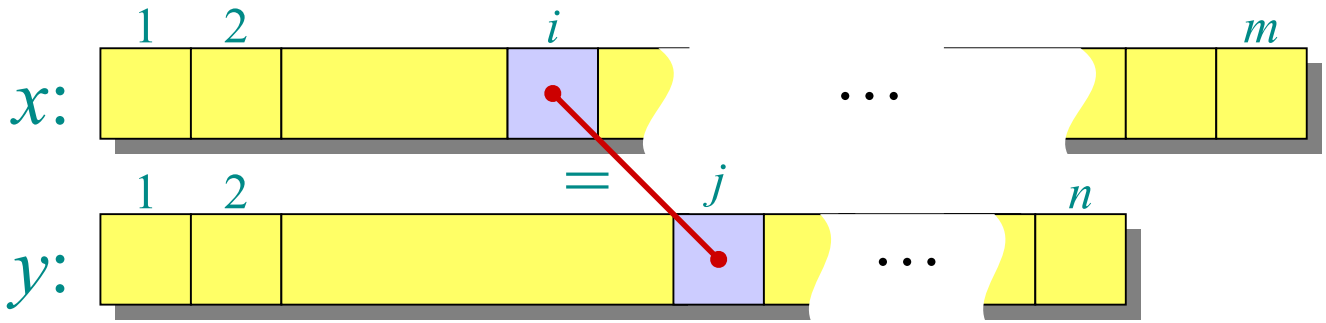
Proof: Case $x[i] = y[j] \ldots$



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

Recursive formulation

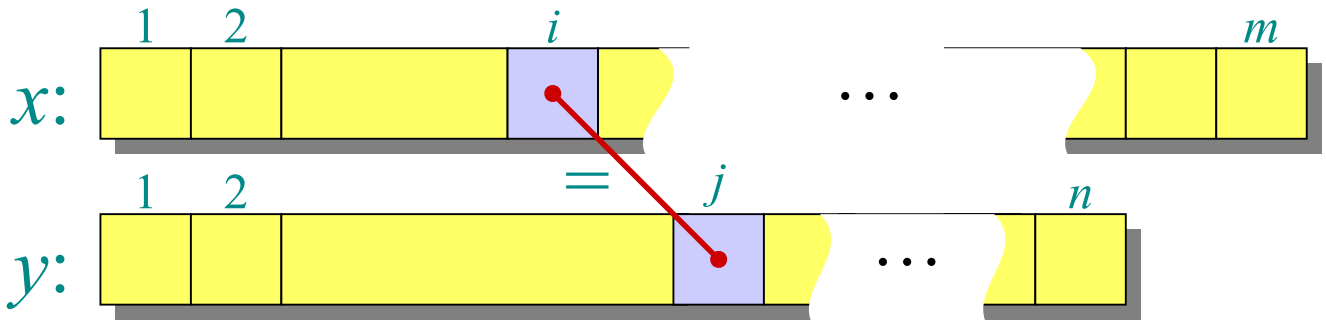
Proof: Case $x[i] = y[j] \ldots$



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

Recursive formulation

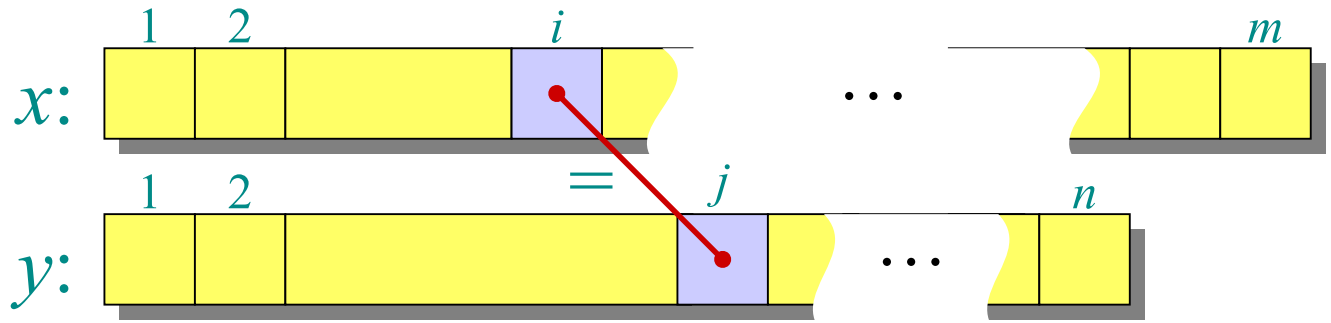
Proof: Case $x[i] = y[j] \ldots$



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

Recursive formulation

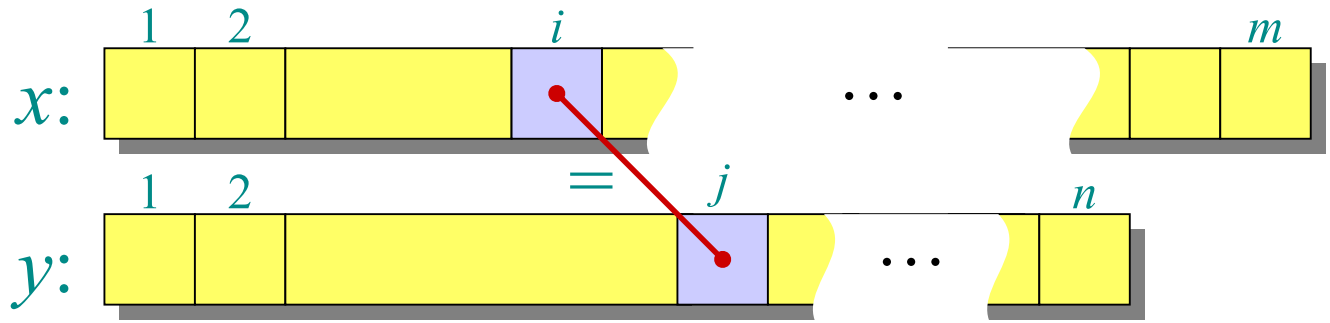
Proof: Case $x[i] = y[j] \ldots$



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

Recursive formulation

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



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

Step-by-Step Example of LCS Algorithm

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

$$X = \text{“ACDBE”}$$

$$Y = \text{“ABCDE”}$$

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

	∅	A	B	C	D	E
∅	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:

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

	$\varnothing$	A	B	C	D	E
$\varnothing$	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' $\longrightarrow L[1,1] = L[0,0] + 1 = 1$.
- Other cells get the max of left or top.

	∅	A	B	C	D	E
∅	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' $\longrightarrow L[2,3] = L[1,2]+1 = 2$.
- Other cells get the max of left or top.

	∅	A	B	C	D	E
∅	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' $\longrightarrow L[3,4] = L[2,3] + 1 = 3$.
- Other cells get the max of left or top.

	∅	A	B	C	D	E
∅	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' $\longrightarrow L[4,2] = L[3,1] + 1 = 2$.
- Other cells get the max of left or top.

	∅	A	B	C	D	E
∅	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' $\longrightarrow L[5,5] = L[4,4]+1 = 4$.
- Other cells get the max of left or top.

	∅	A	B	C	D	E
∅	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.

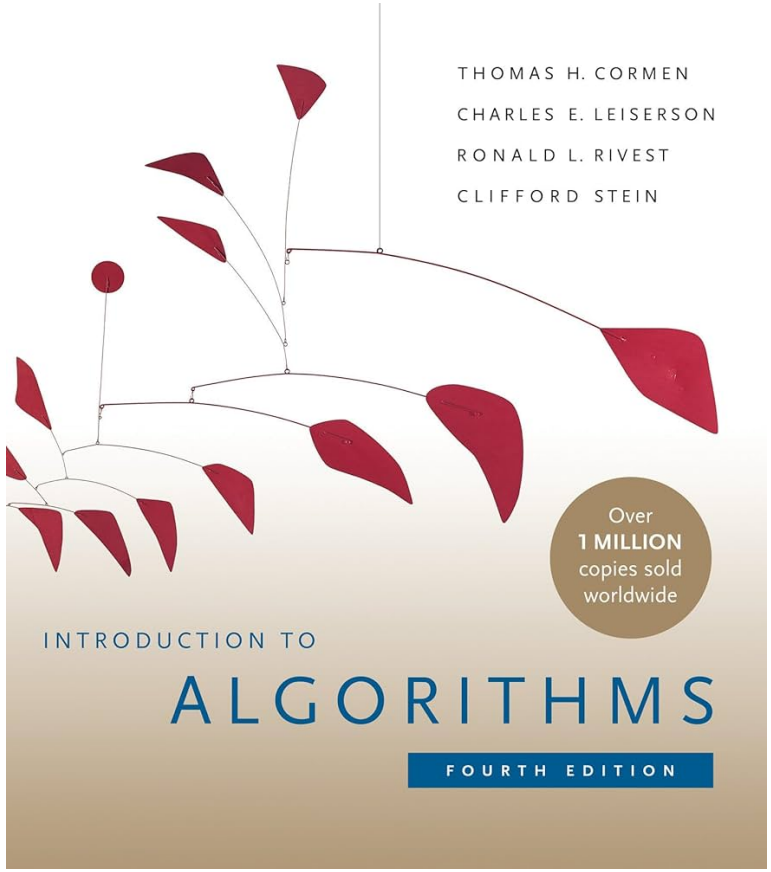
Tracing Back from L[5,5]:

1. E ($X[5] = Y[5]$) $\longrightarrow$ Add 'E'
2. D ($X[3] = Y[4]$) $\longrightarrow$ Add 'D'
3. C ($X[2] = Y[3]$) $\longrightarrow$ Add 'C'
4. A ($X[1] = Y[1]$) $\longrightarrow$ Add 'A'

Thus, the LCS = "ACDE".

End of Lecture 5.

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/`.