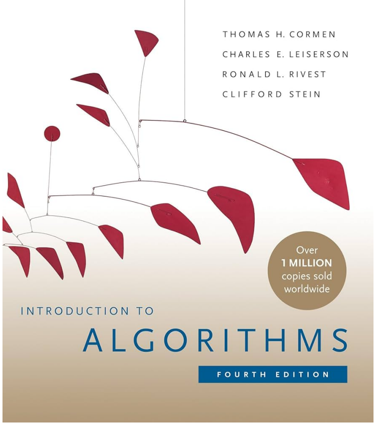
Introduction to Algorithms
Lecture 5: Dynamic Programming (DP)

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

September 1, 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

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

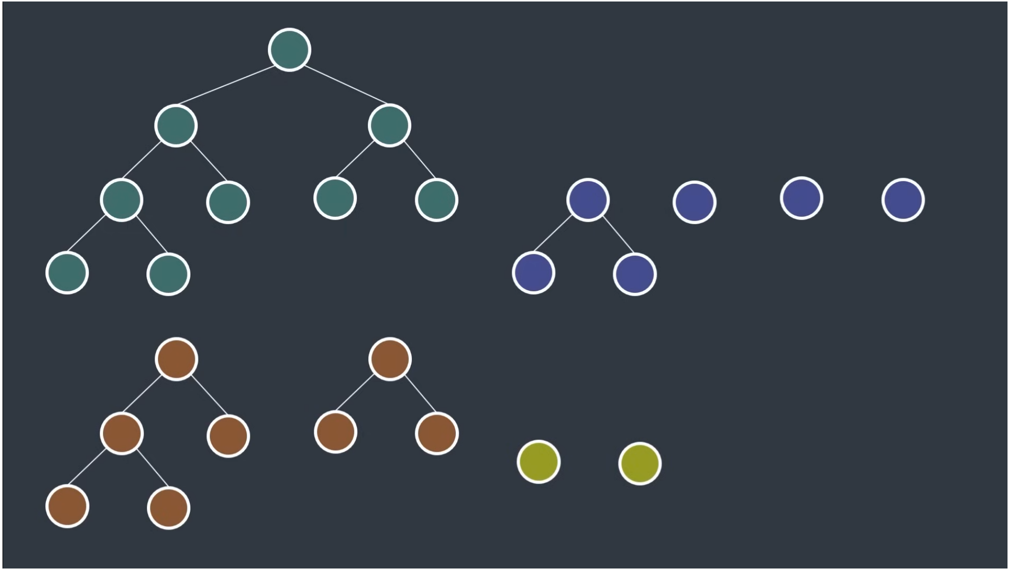
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems.
3. Compute the value of an optimal solution in bottom-up fashion (recursion & memoization).
4. Construct an optimal solution from the computed information.

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

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

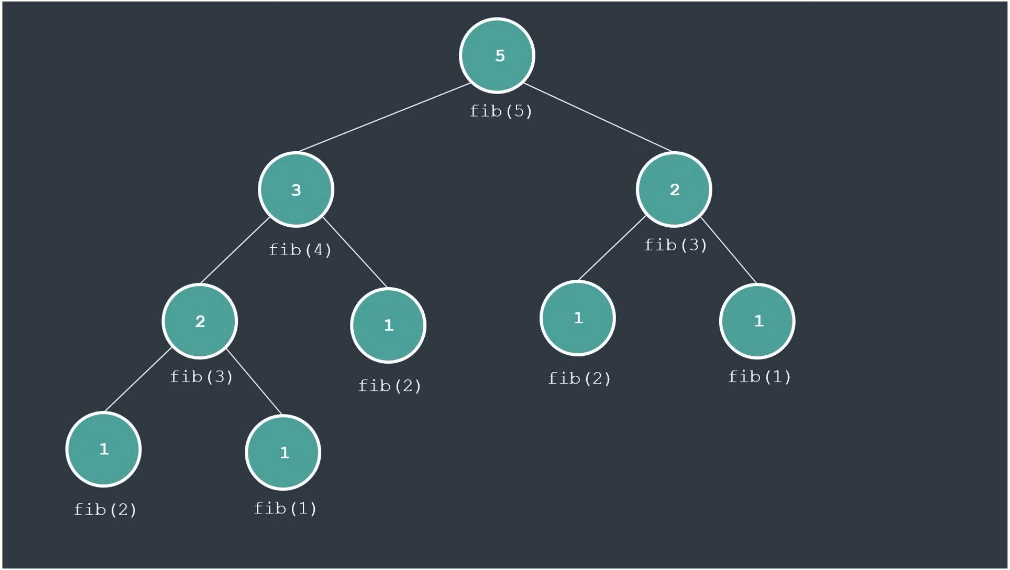
```python
def fib(n):
    if n <= 2:
        result = 1
    else:
        result = fib(n - 1) + fib(n - 2)
    return result
```

Naive Approach

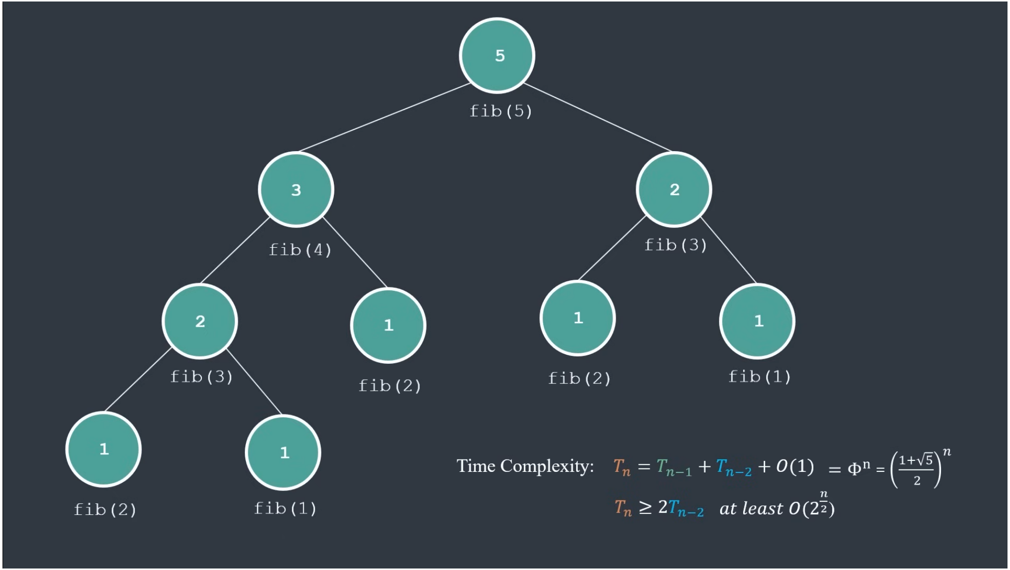
```python
def fib(n):
    if n <= 2:
        result = 1
    else:
        result = fib(n - 1) + fib(n - 2)
    return result
```

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

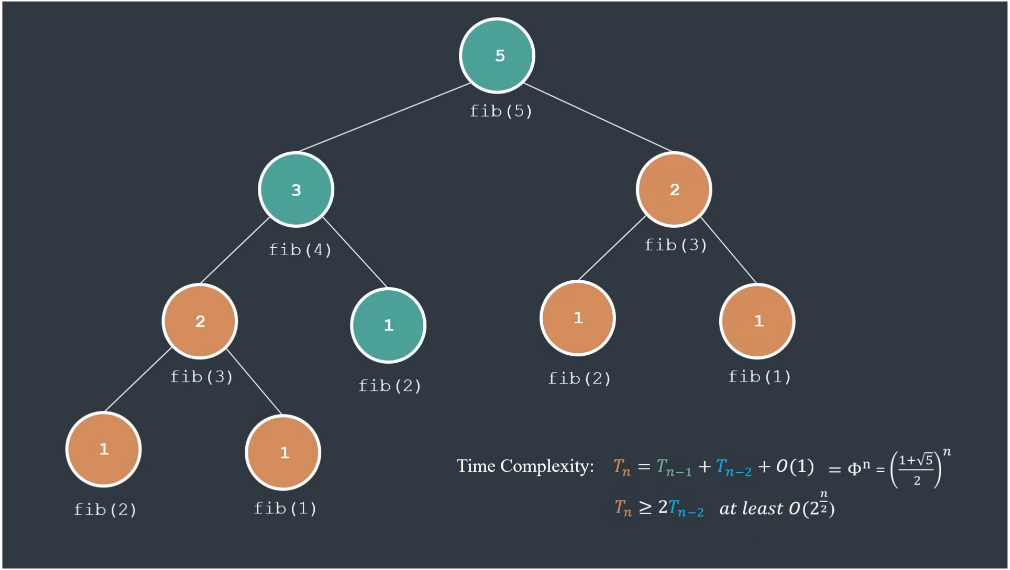
Naive Approach



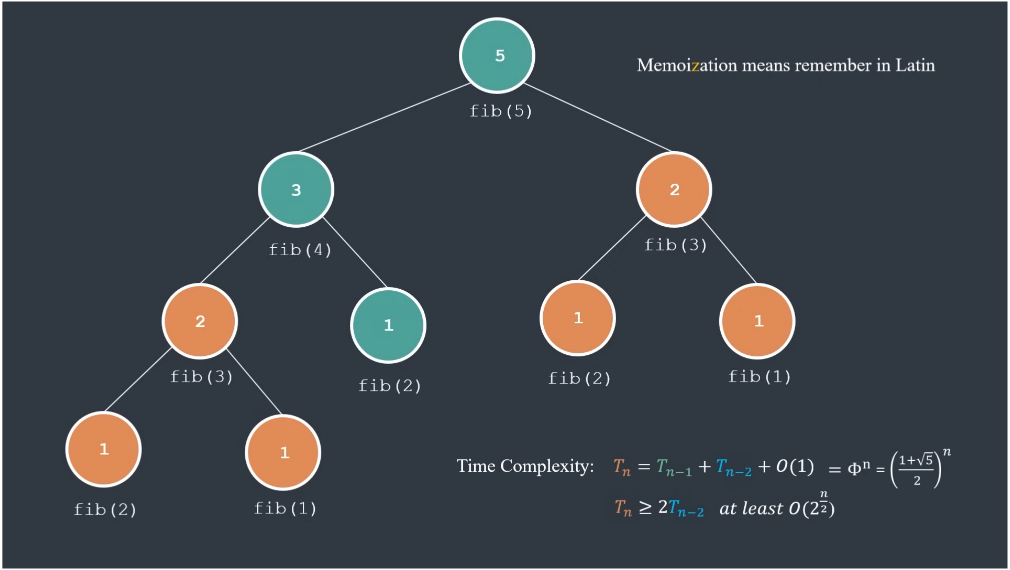
Naive Approach



Naive Approach



Naive Approach



Memoization Approach

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

Memoization Approach

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

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

Memoization Approach

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

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

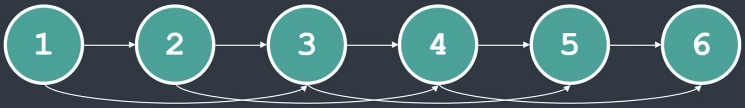
Time Complexity: $O(n)$.

DP = **Recursion** + **Memoization**

Bottom-up Approach

```python
def fib(n):
    memo = {}
    for i in range(1, n + 1):
        if i <= 2:
            result = 1
        else:
            result = memo[n - 1] + memo[n - 2]
        memo[i] = result
    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 $i \leq j$.

```python
def L(i, j):
    if i == j:
        return 1
    if X[i] == X[j]:
        if i + 1 == j:
            return 2
        else:
            return 2 + L(i + i, j - 1)
    else:
        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.

$$\begin{aligned} T(n) &= \text{running time on input of length } n \\ T(n) &= \begin{cases} 1 & n = 1 \\ 2T(n-1) & n > 1 \end{cases} \\ &= 2^{n-1} \end{aligned}$$

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.

```python
def L(i, j):
    if i == j:
        return 1
    if X[i] == X[j]:
        if i + 1 == j:
            return 2
        else:
            return 2 + L(i + i, j - 1)
    else:
        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.

```
def L(i, j):
    if i == j:
        return 1
    if X[i] == X[j]:
        if i + 1 == j:
            return 2
        else:
            return 2 + L(i + i, j - 1)
    else:
        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

- 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.

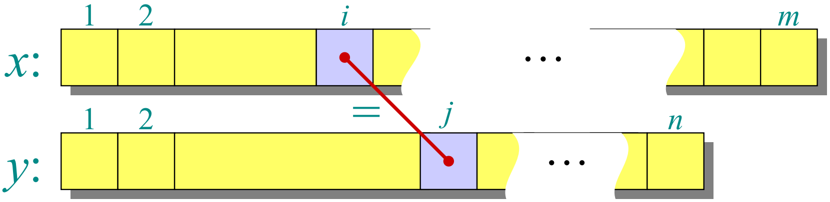
- **Notation:** Denote the length of a sequence s as $|s|$.
- **Strategy:** Consider **prefixes** of x and y:
  - Define $c[i,j] = |LCS(x[1 \cdots i], y[1 \cdots j])|$.
  - Then, $c[m,n] = |LCS(x,y)|$.

Recursive formulation

Theorem

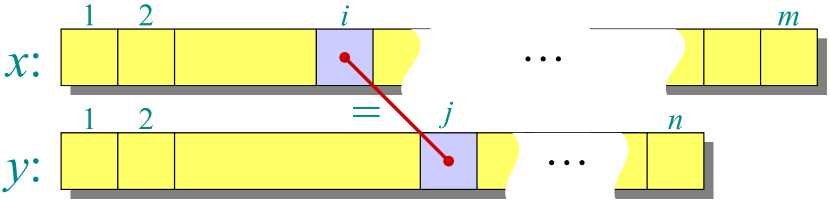
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] & \text{if } x[i] = y[j], \\ \max\{c[i, j-1], c[i-1, j]\} & \text{otherwise.} \end{cases}$$

Proof: Case $x[i] = y[j]$ ...



Recursive formulation

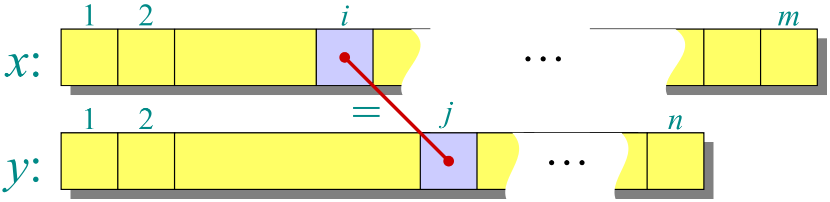
Proof: Case $x[i] = y[j] \ldots$



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

Recursive formulation

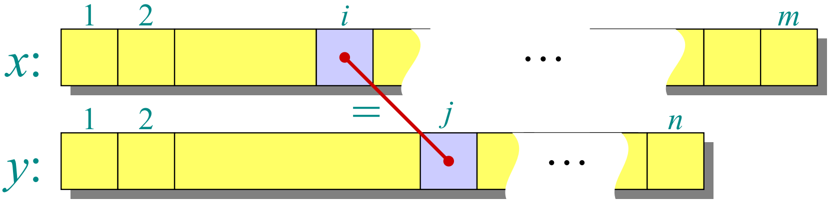
Proof: Case $x[i] = y[j] \ldots$



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

Recursive formulation

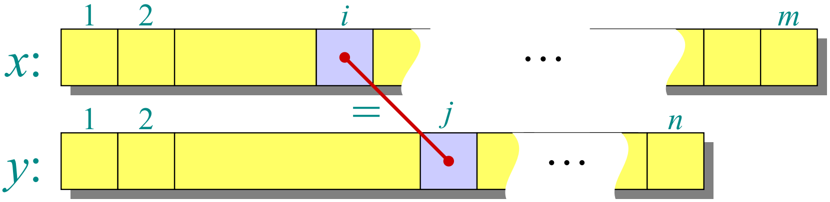
Proof: Case $x[i] = y[j] \ldots$



- 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

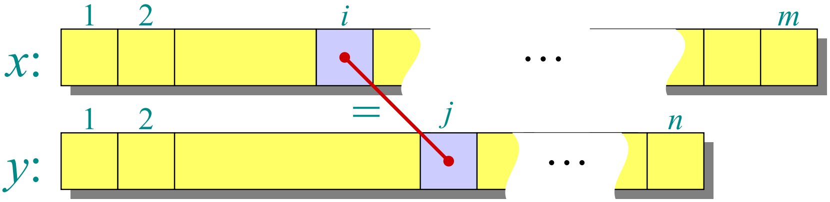
Proof: Case $x[i] = y[j] \ldots$



- 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, $z[1 \cdots k\text{–}1]$ 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

	$\varnothing$	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

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

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

	∅	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

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.

	$\varnothing$	A	B	C	D	E
$\varnothing$	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.

	$\varnothing$	A	B	C	D	E
$\varnothing$	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”.

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 weather 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 $X = \langle x_1, x_2, \ldots, x_m \rangle$:
  - We can define the *i-th* **prefix** of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.
  - 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 $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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 $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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 $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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 LCSs 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:

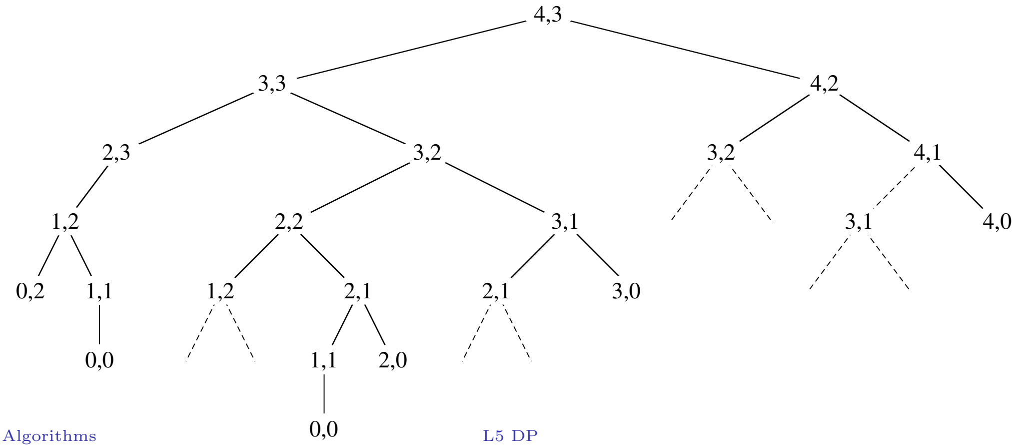
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x[i] = y[j], \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x[i] \neq y[j]. \end{cases}$$

- 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)
let b[1:m, 1:n] and c[0:m, 0:n] be new tables
for i = 1 to m
    c[i,0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m              // compute table entries in row-major order
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
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)
if i == 0 or j == 0
    return                    // the LCS has length 0
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i                 // same as y_j
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
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, PRI'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 Natarajan, 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

```
Terminator                                   Mar 31 21:42
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 146x33

(base) 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 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
↑ ↑ ↑ ↖ ← ↖
↖ ← ← ↑ ↖ ←
↑ ↑ ↖ ← ↑ ↑
↖ ↑ ↑ ↑ ↖ ←
↑ ↖ ↑ ↑ ↑ ↑
↑ ↑ ↑ ↖ ↑ ↖
↖ ↑ ↑ ↑ ↖ ↑
GTCGTCGGAAGCCGGCCGAA
(base) and@and-Inspiron-5584:~/MEGA/Work/PUJ/2025-S1/ADA/Resources/clrsPython/clrsPython/Chapter 14$
Plain Text  Tab Width: 8  Ln 1, Col 1  INS
```

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
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

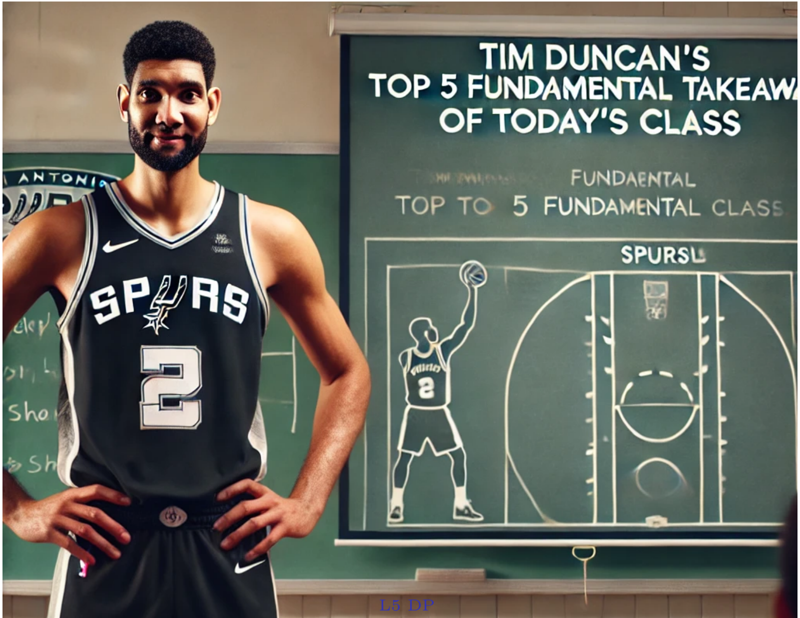
Figure 14.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the 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 "↖" on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Exercise

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

End of Lecture 5.

TDT5FTOTTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURSL
SPURS
2

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.

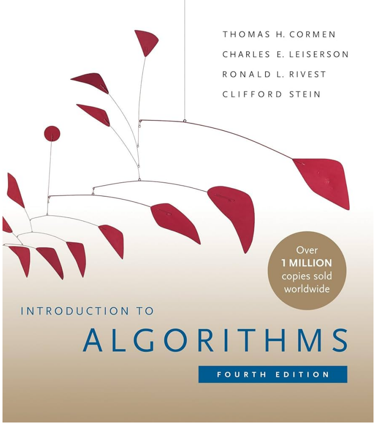
4 The LCS problem uses a table-based approach to find the length and content of the longest common subsequence in quadratic time.

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/.