# LONGEST COMMON SUBSEQUENCE

# Inspiration

- Biological applications often need to compare the DNA of two (or more) different organisms
- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine
- each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set {A, C, G, T}

# Inspiration

- For example, the DNA of one organism may be $S_1$= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organism may be $S_{2}$= GTCGTTCGGAATGCCGTTGCTCTGTAAA.
- One reason to compare two strands of DNA is to determine how “similar the two strands are, as some measure of how closely related the two organisms are.

# Inspiration

- We can define similarity in many different ways
- First way - we can say that two DNA strands are similar if one is a substring of the other
- In our example, neither $S_1$ nor $S_2$ is a substring of the other.
- Second way - two strands are similar if the number of changes needed to turn one into the other is **small**

$S_1$= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

$S_2$ = GTCGTTCGGAATGCCGTTGCTCTGTAAA

# Inspiration

- Third way measure the similarity of strands $S_1$ and $S_2$ is by finding a third strand $S_3$
- In which bases in $S_3$ appear in each of $S_1$ and $S_2$; these bases must appear in the same order, but not necessarily consecutively
- Longer the strand $S_3$ we can find, the more similar $S_1$ and $S_2$ are

$S_1$= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

$S_2$ = GTCGTTCGGAATGCCGTTGCTCTGTAAA

1. **Substring Similarity Example:**
    - S1: `AGTACGTACGGTAG`
    - S2: `TACGTA`
    - S2 is a substring of S1, so they are similar.

2. **Edit Distance Example:**
    - S1: `ACGTTAGC`
    - S2: `ACGTGAGC`
    - Only one substitution is needed, so the strands are similar.

3. **Longest Common Subsequence Example:**
    - S1: `ACGTACGGTAC`
    - S2: `GTACGCTAG`
    - The longest common subsequence is `GTAC`, with a length of 4, so the strands are considered similar.

# Inspiration

- $S_1$= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- $S_{2\ =}$ GTCGTTCGGAATGCCGTTGCTCTGTAAA
- $S_3$ is GTCGTCGGAAGCCGGCCGAA

# Problem Statement

- Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y

# Longest Common Subsequence

P = “BATD”

Q = “ABACD”

# Longest Common Subsequence

P = "BATD"

Q = "ABACD"

# Dynamic Programming

1. Recursive Solution
2. Memorize Intermediate Results
3. Bottom-Up

# Recursive Solution

$LCS(P_0, Q_0) \rightarrow 3$

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0$ = "          x"
$Q_0$ = "          x"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

**Case 2:**

$P_0$ = "          x"
$Q_0$ = "          y"

$LCS(P_0, Q_0)$
$= \max\{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Recursive Solution

"BATD"
"ABACD" $\rightarrow$ "BAD"

$LCS(P_0, Q_0)$

**Case 1:**

$P_0$ = "      x"
$Q_0$ = "      x"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

**Case 2:**

$P_0$ = "      x"
$Q_0$ = "      y"

$LCS(P_0, Q_0)$
$= \max\{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Recursive Solution

$\underbrace{LCS(P_0, Q_0)}$ → 3

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0 =$ " $\boxed{P_1}$ x"
$Q_0 =$ " $\boxed{Q_1}$ x"
" $\boxed{\ }$ x"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

**Case 2:**

$P_0 =$ "      x"
$Q_0 =$ "      y"

$LCS(P_0, Q_0)$
$= \max\{LCS(P_1, Q_0), LCS(P_0, Q_1)\}$

# Recursive Solution

$\text{LCS}(P_0, Q_0) \rightarrow 3$

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0$ = " $\boxed{P_1}$ x"
$Q_0$ = " $\boxed{Q_1}$ x"

" $\square$ x"

$\text{LCS}(P_0, Q_0)$
$= 1 + \text{LCS}(P_1, Q_1)$

**Case 2:**

$P_0$ = "         x"
$Q_0$ = "         y"

$\text{LCS}(P_0, Q_0)$
$= \max\{\text{LCS}(P_1, Q_0), \text{LCS}(P_0, Q_1)\}$

# Recursive Solution

$LCS(P_0, Q_0) \rightarrow 3$

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0$ = "$\boxed{P_1}$ x"
$Q_0$ = "$\boxed{Q_1}$ x"  "□ x"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

**Case 2:**

$P_0$ = "   x"
$Q_0$ = "   y"

$LCS(P_0, Q_0)$
$= \max \{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Recursive Solution

$LCS(P_0, Q_0) \rightarrow 3$

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0$ = " $\boxed{P_1}$ x"
$Q_0$ = " $\boxed{Q_1}$ x"
" $\square$ x"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

**Case 2:**

$P_0$ = " x"
$Q_0$ = " y"

$LCS(P_0, Q_0)$
$= \max \{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Recursive Solution

$LCS(P_0, Q_0) \rightarrow 3$

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0$ = " $\boxed{P_1}$ x"
$Q_0$ = " $\boxed{Q_1}$ x" } " $\square$ x"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

**Case 2:**

$P_0$ = " x"
$Q_0$ = " y"

$LCS(P_0, Q_0)$
$= \max \{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Recursive Solution

$LCS(P_0, Q_0) \to 3$

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0$ = "$\boxed{P_1}$ x"
$Q_0$ = "$\boxed{Q_1}$ x"

"$\boxed{\phantom{x}}$ x"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

**Case 2:**

$P_0$ = "$\boxed{P_1}$ x"
$Q_0$ = "$\boxed{Q_1}$ y"

$LCS(P_0, Q_0)$
$= \max\{LCS(P_1, Q_0), LCS(P_0, Q_1)\}$

# Recursive Solution

$\underbrace{LCS(P_0, Q_0)} \rightarrow 3$

"BATD"
"ABACD" $\rightarrow$ "BAD"

**Case 1:**

$P_0$ = "$\boxed{P_1}$ x"
$Q_0$ = "$\boxed{Q_1}$ x"

"$\square$ x"

**Case 2:**

$P_0$ = "$\boxed{P_1}$ x"
$Q_0$ = "$\boxed{Q_1}$ y"

$LCS(P_0, Q_0)$
$= 1 + LCS(P_1, Q_1)$

$LCS(P_0, Q_0)$
$= \max\{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Recursive Solution

$\mathrm{LCS}(P_0, Q_0) \to 3$

"BATD"
"ABACD" → "BAD"

**Case 1:**

$P_0$ = "$\boxed{P_1}$ x"
$Q_0$ = "$\boxed{Q_1}$ x"

"□ x"

$\mathrm{LCS}(P_0, Q_0)$
$= 1 + \mathrm{LCS}(P_1, Q_1)$

**Case 2:**

$P_0$ = "$\boxed{P_1}$ x"
$Q_0$ = "$\boxed{Q_1}$ y"

$\mathrm{LCS}(P_0, Q_0)$
$= \max\{\mathrm{LCS}(P_1, Q_0), \mathrm{LCS}(P_0, Q_1)\}$

# Recursive Solution (Code)

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

# Recursive Solution (Code)

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

$LCS(P_0 , Q_0)$
$= 1 + LCS(P_1 , Q_1)$

# Recursive Solution (Code)

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

$LCS(P_0, Q_0) = 1 + LCS(P_1, Q_1)$

$LCS(P_0, Q_0) = \max\{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Recursive Solution (Code)

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

# Recursive Solution (Code)

P= "ABc"

n=2

Q = "ABc"

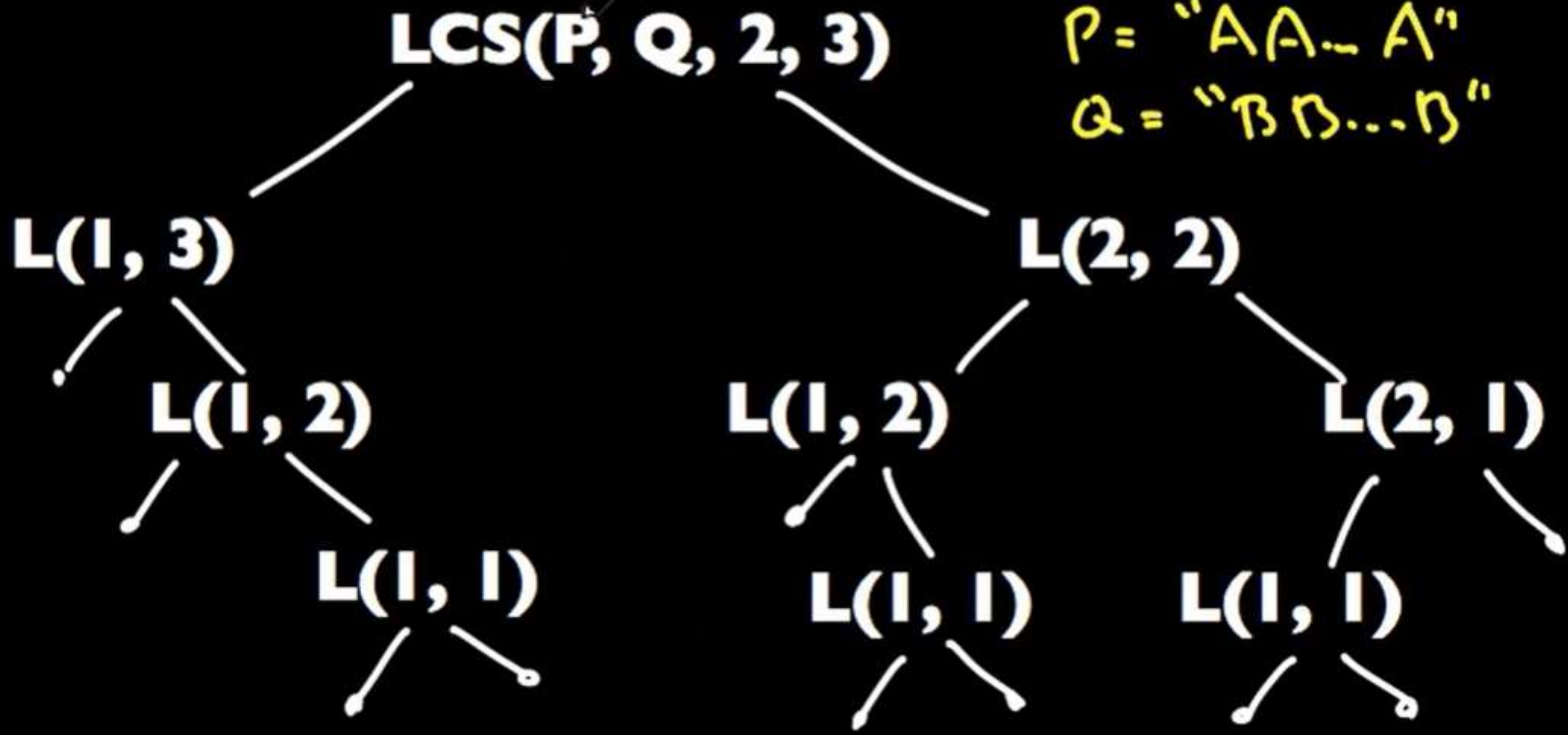
m =

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

# Analysis of Recursive Solution

P = "AA"
Q = "BBB"

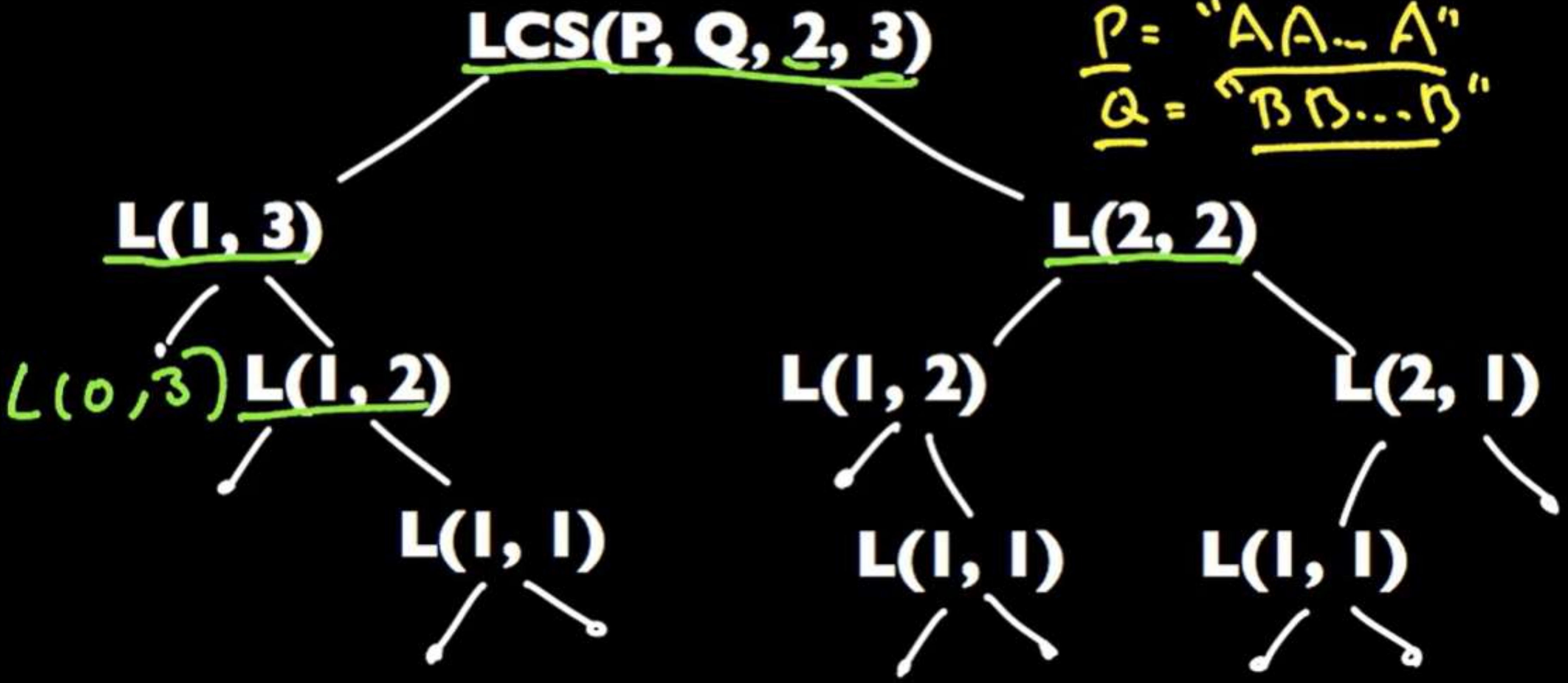
P = "AA... A"
Q = "BB... B"



# Analysis of Recursive Solution

P = "AA"
Q = "BBB"

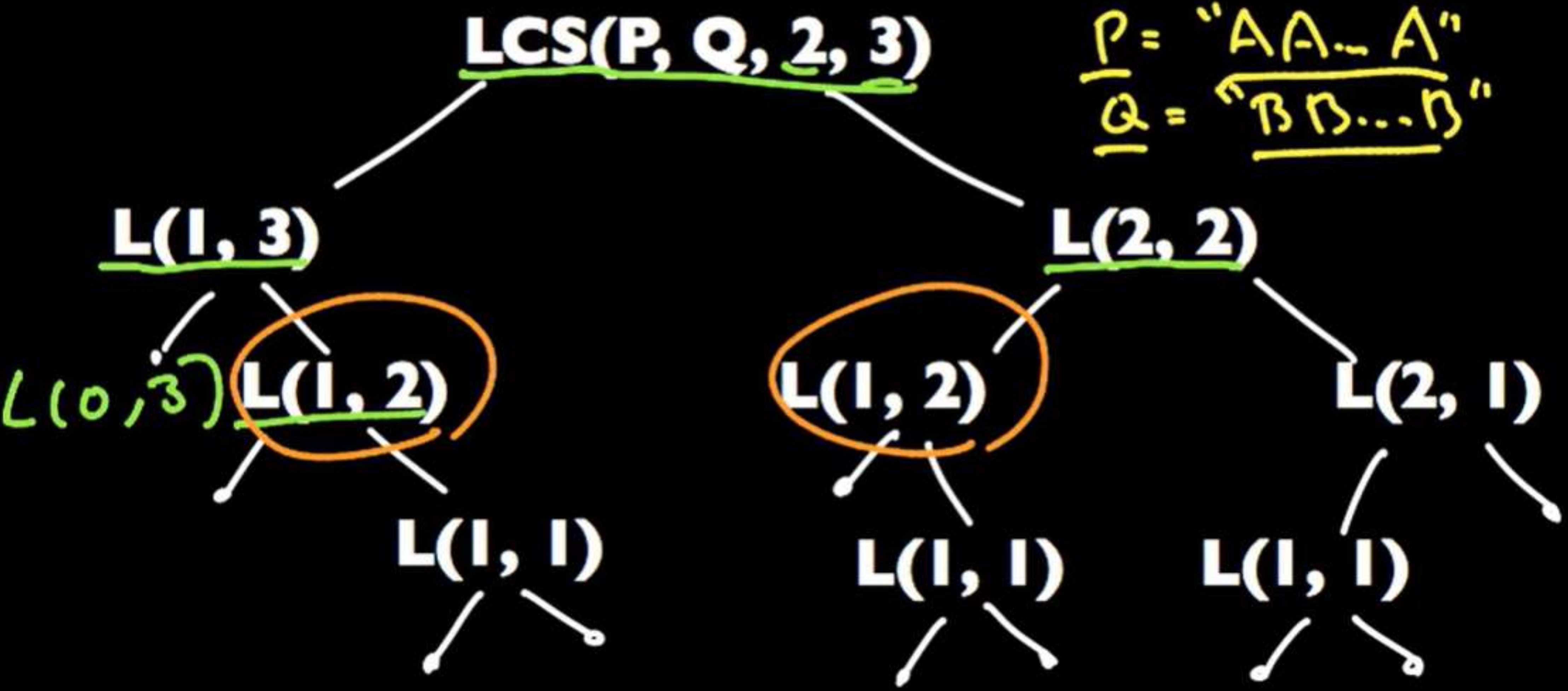
P = "AA... A"
Q = "BB...B"

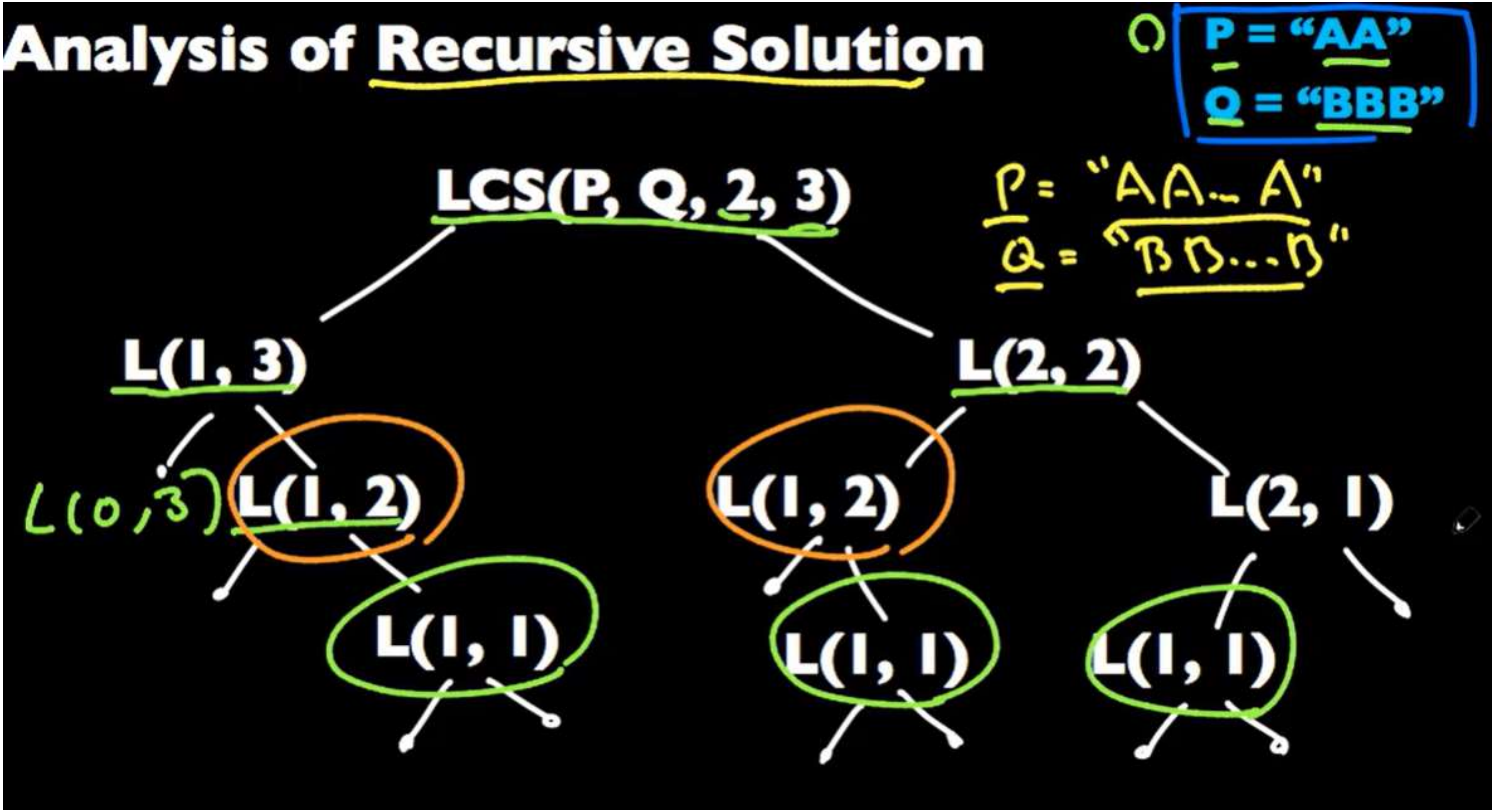


# Analysis of Recursive Solution

P = "AA"

Q = "BBB"

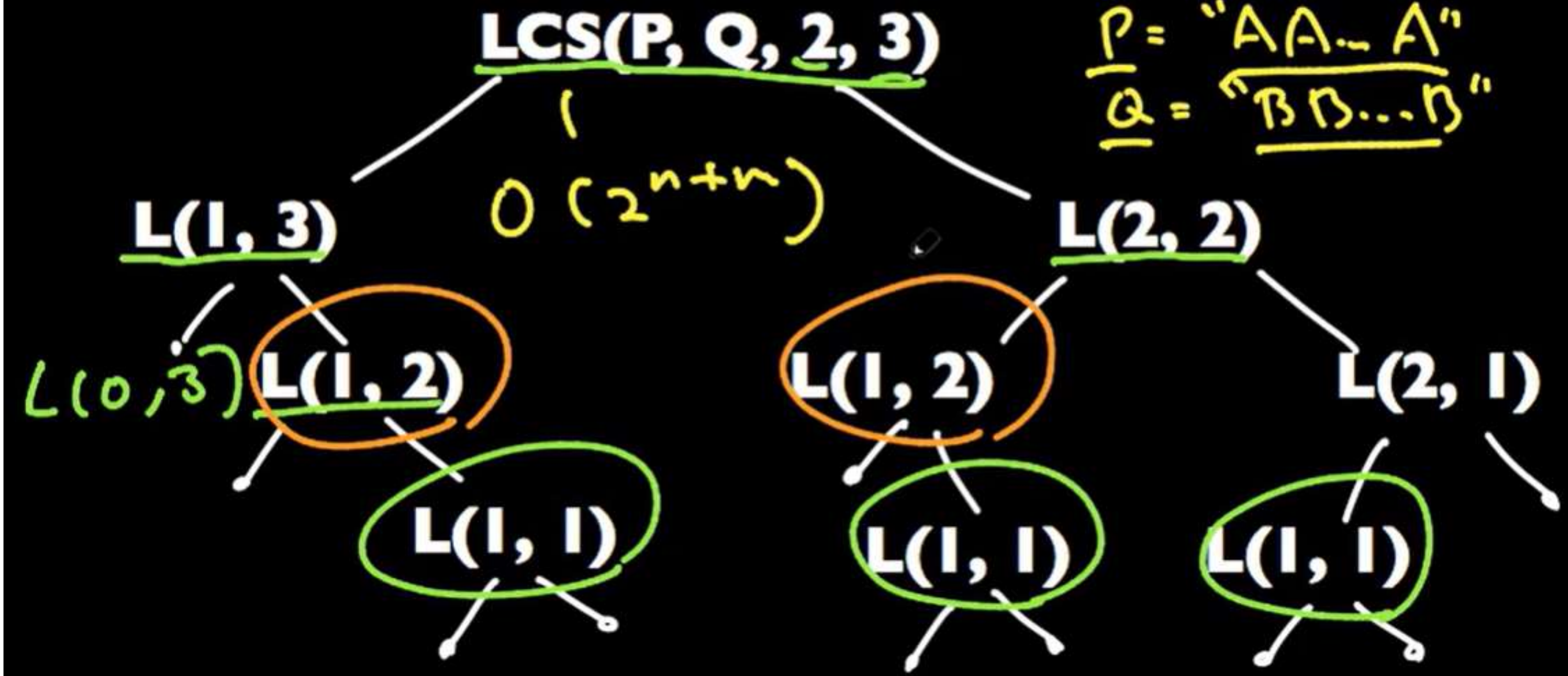


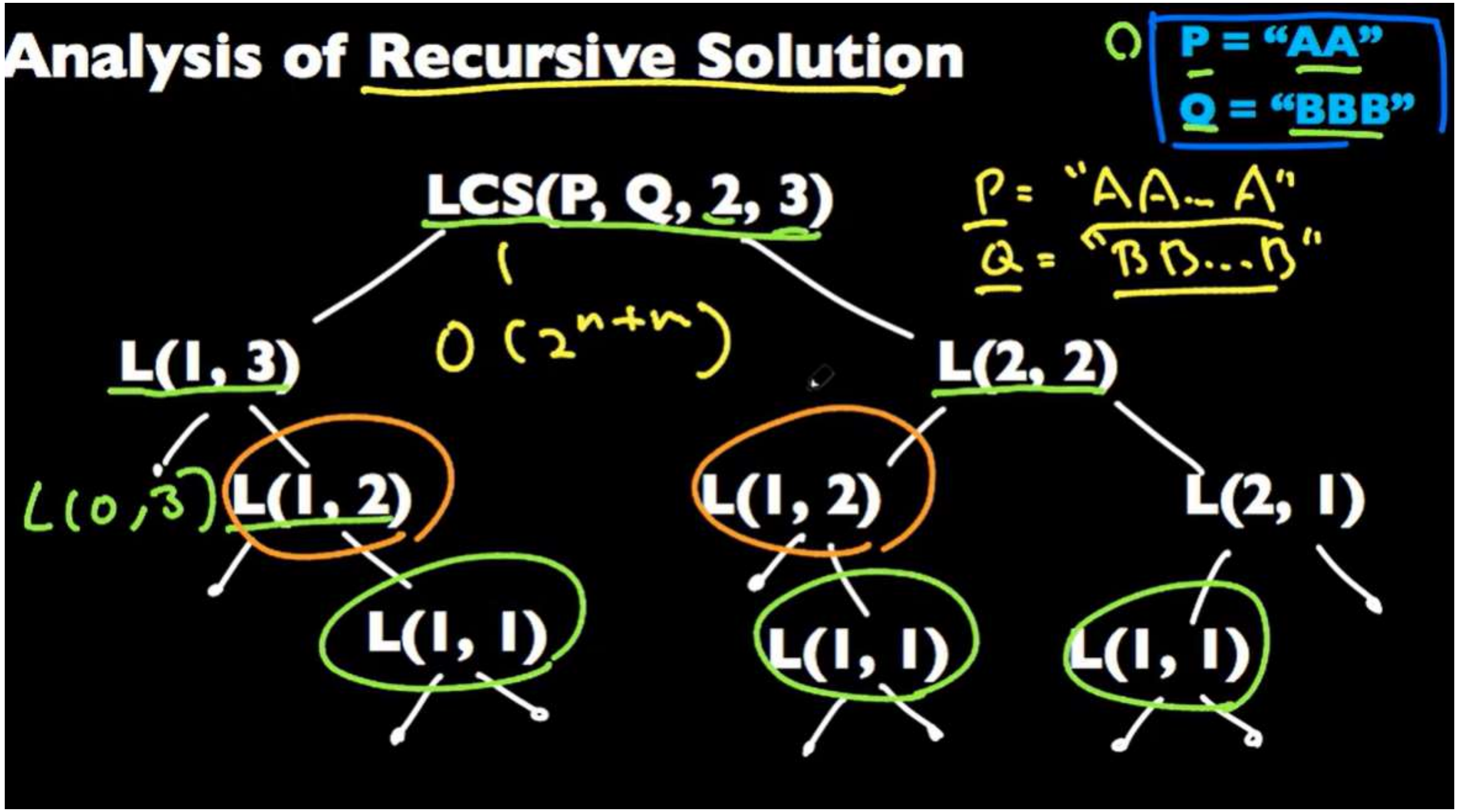


Analysis of Recursive Solution
P = "AA"
Q = "BBB"
LCS(P, Q, 2, 3)
P = "AA... A"
Q = "BB... B"
L(1, 3)
L(2, 2)
L(0, 3)
L(1, 2)
L(1, 2)
L(2, 1)
L(1, 1)
L(1, 1)
L(1, 1)

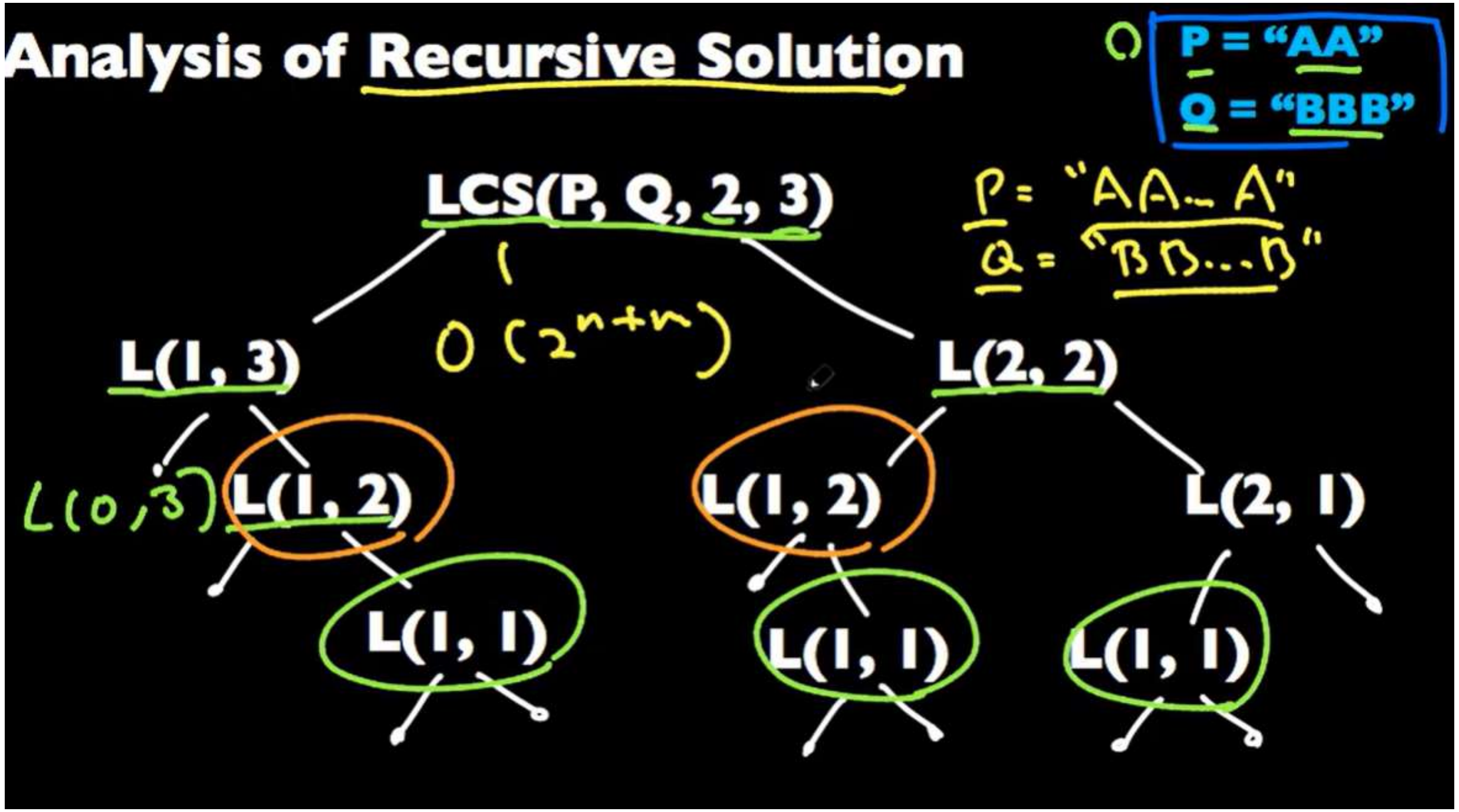
# Analysis of Recursive Solution

P = "AA"
Q = "BBB"





Analysis of Recursive Solution
P = "AA"
Q = "BBB"
LCS(P, Q, 2, 3)
P = "AA... A"
Q = "BB... B"
O(2^n+m)
L(1, 3)
L(2, 2)
L(0, 3)
L(1, 2)
L(1, 2)
L(2, 1)
L(1, 1)
L(1, 1)
L(1, 1)


Analysis of Recursive Solution
P = "AA"
Q = "BBB"
LCS(P, Q, 2, 3)
P = "AA... A"
Q = "BB... B"
O (2^n+m)
L(1, 3)
L(2, 2)
L(0, 3)
L(1, 2)
L(1, 2)
L(2, 1)
L(1, 1)
L(1, 1)
L(1, 1)

# Dynamic Programming

1. Recursive Solution
2. Memorize Intermediate Results
3. Bottom-Up

# Memorize Intermediate Results


```
// Initialize arr[n][m] to undefined
def LCS(P, Q, n, m)
    if arr[n][m] != undefined: return arr[n][m]
    if n == 0 or m == 0:
        result = 0
    else if P[n-1] == Q[m-1]:
        result =  1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    arr[n][m] = result
    return result
```

P = "AA"
Q = "BBB"

# Memorize Intermediate Results


```
0 // Initialize arr[n][m] to undefined
  def LCS(P, Q, n, m)
  0 if arr[n][m] != undefined: return arr[n][m]
    if n == 0 or m == 0:
        result = 0
    else if P[n-1] == Q[m-1]:
        result =  1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
  0 arr[n][m] = result
    return result
```

O

2nm

time/cell = O(1)

O(nm)

nm →

## Explanation:

1. The function **LCS(P, Q, n, m)** is a recursive function that computes the longest common subsequence length between two sequences **P** and **Q** of lengths **n** and **m**, respectively.
2. The key optimization used is **memoization**, where we store the results of previously computed subproblems in **arr[n][m]** to avoid redundant computations.
3. The worst-case number of function calls (i.e., the number of unique (n, m) states we compute) is **O(nm)** since we fill up a **2D table of size n × m**.
4. The computation per cell (each state) takes **O(1)** time.
5. Therefore, the total time complexity is **O(nm)**.

## Understanding the Complexity:

1. The recursive function **LCS**(P, **Q, n, m**) explores **two recursive branches** in the worst case:
    - One branch reduces **n** by 1: **LCS(P, Q, n-1, m)**
    - Another branch reduces **m** by 1: **LCS(P, Q, n, m-1)**
2. This forms a **binary recursion tree** where at each level, the number of recursive calls doubles.

## Complexity Growth:

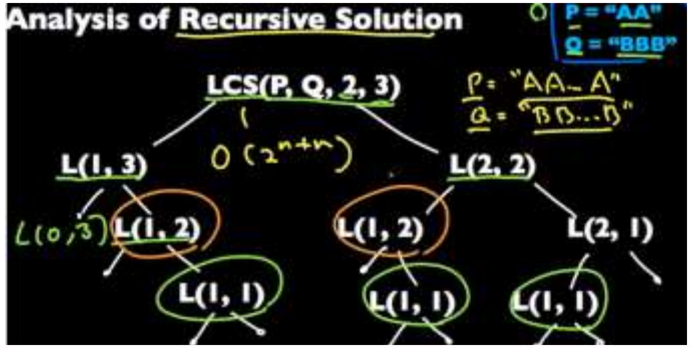
- The depth of the recursion tree is at most **m + n**, leading to **O(2^(m+n))** possible recursive calls.
- However, in standard LCS problems, where **m ≈ n**, the complexity can also be expressed as O(2^max(m, n)).

## Which One is Correct?

- **If m and n are significantly different**, then O(2^(m+n)) is the more precise complexity.
- **If m and n are roughly equal**, we often simplify it to **O(2^max(m, n))**, since max(m, n) dominates.

## Explanation:

1. The function **LCS(P, Q, n, m)** is a recursive function that computes the longest common subsequence length between two sequences **P** and **Q** of lengths **n** and **m**, respectively.
2. The key optimization used is **memoization**, where we store the results of previously computed subproblems in **arr[n][m]** to avoid redundant computations.
3. The worst-case number of function calls (i.e., the number of unique (n, m) states we compute) is **O(nm)** since we fill up a **2D table of size n × m**.
4. The computation per cell (each state) takes **O(1)** time.
5. Therefore, the total time complexity is **O(nm)**.




- Without memoization, the naive recursive approach would have an exponential complexity **O(2^(m+n))** due to redundant computations.
- With memoization, each state **(n, m)** is only computed once and stored in the **arr[n][m]** table.
- Since there are at most **nm** unique states, and each is computed in constant **O(1)** time, the final complexity remains **O(nm)**.

- **If m and n are roughly equal**, we often simplify it to **O(2^max(m, n))**, since max(m, n) dominates.

?

### 1. Starting with O(2^(m+n))

From the naive recursive approach, the worst-case time complexity is:

$$O(2^{m+n}) \quad \text{where } \mathbf{m} \text{ and } \mathbf{n} \text{ are the lengths of the two sequences.}$$

### 2. Assuming m ≈ n

- If **m and n are roughly equal**, we can express them as **m ≈ n ≈ k** for some k.
- Then, we rewrite the complexity: $O(2^{m+n}) = O(2^{k+k}) = O(2^{2k})$

### 3. Using the Property of Exponents

We use the identity: $2^{2k} = (2^k)^2$

Since **Big-O notation ignores constant factors**, we drop the squared term: $O(2^{2k}) = O(2^k)$

Since **k ≈ max(m, n)**, we get: $O(2^{\max(m,n)})$

# Longest Common Subsequence
## Bottom-Up Approach – Non-Recursive

In the context of the **Longest Common Subsequence (LCS)** using a **bottom-up dynamic programming approach**, the term **Max(Left, TOP)** refers to the recurrence relation used when the characters from the two sequences **do not match**.

## Explanation:

1. The LCS table (or DP table) is built such that:
   - Each cell **L(n, m)** represents the length of the LCS for the first **n** characters of string **P** and the first **m** characters of string **Q**.
   - The values are filled iteratively using previously computed values.

3. **When characters do not match:**
   - If **P[n] ≠ Q[m]**, then:

$$L(n, m) = \max(L(n, m-1), L(n-1, m))$$

   - **L(n, m-1)** → Represents the value from the **left** (excluding the current character from string Q).
   - **L(n-1, m)** → Represents the value from the **top** (excluding the current character from string P).
   - The **maximum** of these two values is chosen to ensure that the longest subsequence found so far is retained.

# How to predict the common Sequence

**Example from the Image:**

Given Strings:

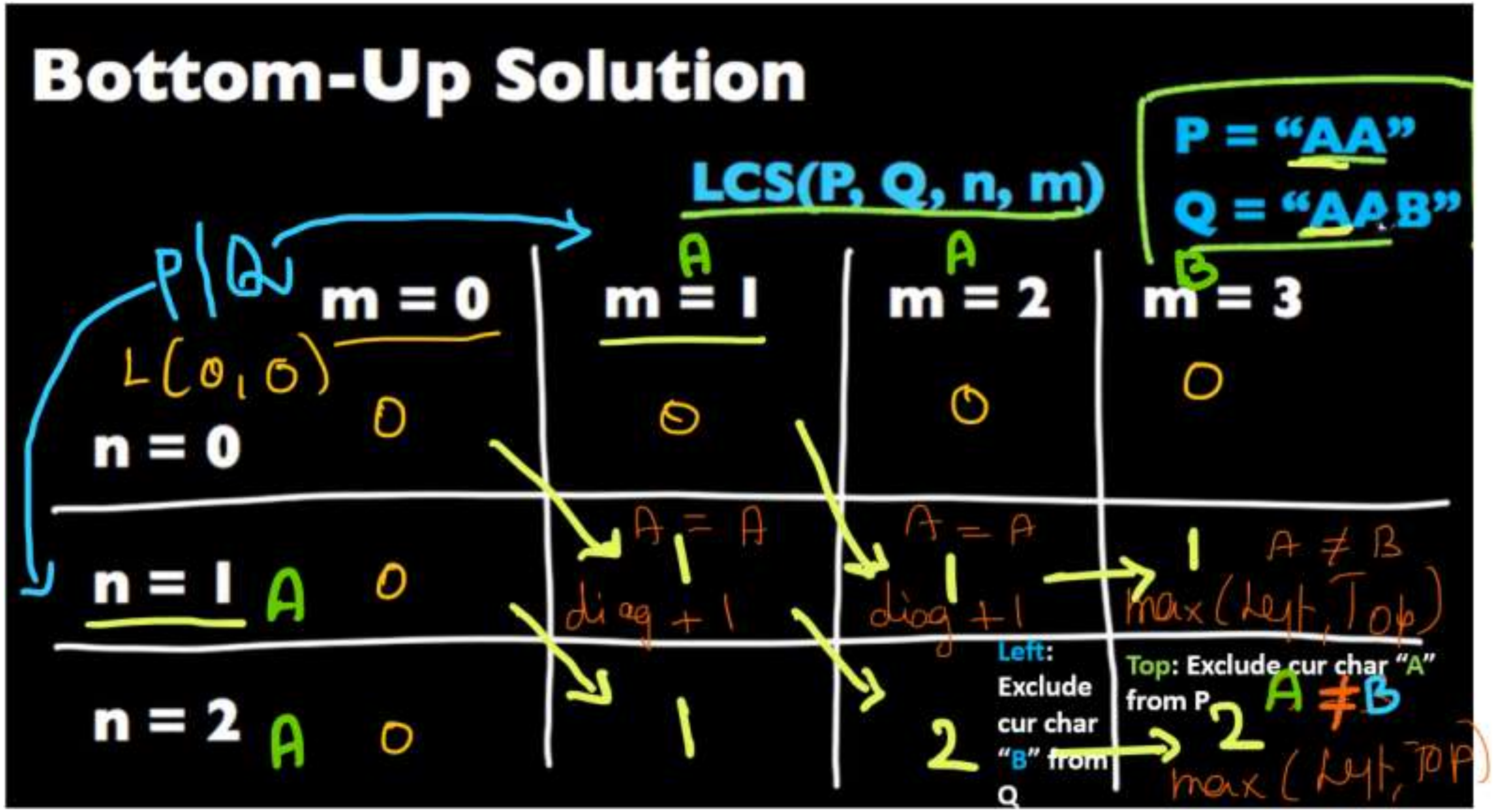
- P = "AA"
- Q = "AAB"

Final Table Value:

- L(2,3) = 2, meaning the LCS has **2 characters**.

**Backtracking Steps:**

1. L(2,3) = max(L(2,2), L(1,3)) = 2
   - **Move left to (2,2)** since L(2,2) = 2.
2. L(2,2) = 1 + L(1,1) = 2
   - Characters match: **A = A → Include 'A'**
   - Move **diagonally to** (1,1).
3. L(1,1) = 1 + L(0,0) = 1
   - Characters match: **A = A → Include 'A'**
   - Move **diagonally to** (0,0).




**Final Predicted LCS:**

- "AA" (Reading collected characters in reverse)

# Example 2

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | | | | | | |
| b | 0 | | | | | | |
| c | 0 | | | | | | |
| f | 0 | | | | | | |

abcdaf

acbcf

# Example 2

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | | | | | | |
| c | 0 | | | | | | |
| f | 0 | | | | | | |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | | | | | | |
| f | 0 | | | | | | |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | | | | | | |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf
acbcf

abcdaf

acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

diagonaly

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 Max(3,2) | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

***Max(3,2):-***
***Since either excludes cur char c from String 2(left) or excludes cur a char from string 1(top)***

3 **is** present on its **left**, **not** diagonally. So no need to include it in the answer

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

**Again**, 3 **is** present on its **left**, **not** diagonally. So no need to include it in the answer

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf

acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

From Diagonal only, Got this 3. So include c

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf

acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

abcdaf
acbcf

| | | a | b | c | d | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| f | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

# Longest Common Subsequence

# Bottom-Up Approach – Non-Recursive

if ( input**1**[i] == input**2**[j] )

T[i][j] = T[i-1][j-1] + 1

$\boxed{LCS(P_0, Q_0) = 1 + LCS(P_1, Q_1)}$

else

T[i][j] = max ( T[i-1][j], T[i][j-1] )

**Recursive Solution (Code)**

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

$\boxed{LCS(P_0, Q_0) = 1 + LCS(P_1, Q_1)}$

$\boxed{LCS(P_0, Q_0) = \max\{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}}$

# Longest Common Subsequence
# Bottom-Up Approach – Non-Recursive

if ( input**1**[i] == input**2**[j] )

T[i][j] = T[i-1][j-1] + 1

$LCS(P_0, Q_0) = 1 + LCS(P_1, Q_1)$

else

T[i][j] = max ( T[i-1][j], T[i][j-1] )

$LCS(P_0, Q_0) = \max\{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

**Recursive Solution (Code)**

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

$LCS(P_0, Q_0) = 1 + LCS(P_1, Q_1)$

$LCS(P_0, Q_0) = \max\{ LCS(P_1, Q_0), LCS(P_0, Q_1) \}$

# Longest Common Subsequence
# Bottom-Up Approach – Non-Recursive

```
int LCS(string P, string Q) {
    int n = P.length();
    int m = Q.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    // Fill the dp table bottom-up
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (P[i - 1] == Q[j - 1])
                dp[i][j] = 1 + dp[i - 1][j - 1];  // match case
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // no match
        }
    }

    return dp[n][m]; // Length of LCS
}
```

```
int main() {
    string P = "AGGTAB";
    string Q = "GXTXAYB";
    cout << "Length of LCS is " << LCS(P, Q) << endl;
    return 0;
}
```

# Longest Common Subsequence
# Bottom-Up Approach – Non-Recursive

```
int LCS(string P, string Q) {
    int n = P.length();
    int m = Q.length();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    // Fill the dp table bottom-up
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (P[i - 1] == Q[j - 1])
                dp[i][j] = 1 + dp[i - 1][j - 1];  // match case
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // no match
        }
    }

    return dp[n][m]; // Length of LCS
}
```

## DP Table Initialization:

```
vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
```

- `dp[0][*]` and `dp[*][0]` represent comparisons with an **empty string**, so they must be `0`.
- Then we compute for all other `i = 1 to n` and `j = 1 to m`.

```
for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j)
```

This is correct because we are *filling the DP table* starting after the initialized base cases in row 0 and column 0.