

stringbuilder

`StringBuilder` is a Java class used to build and modify strings **efficiently**, especially when you're doing lots of concatenation or edits.

Why `StringBuilder` exists

In Java, `String` is **immutable**:

java

```
String s = "Hello";
s = s + " World"; // creates a NEW object
```

 Copy code

Each change creates a new `String`, which is **slow and memory-wasteful** in loops.

`StringBuilder` solves this by being **mutable**.

Basic usage

java

```
StringBuilder sb = new StringBuilder();

sb.append("Hello");
sb.append(" ");
sb.append("World");

String result = sb.toString(); // "Hello World"
```

 Copy code

Common operations

Append

java

```
sb.append("Java");
sb.append(123);
sb.append(true);
```

 Copy code

Insert

java

```
sb.insert(5, " amazing"); // insert at index
```

 Copy code

Delete

java

```
sb.delete(0, 5);           // delete range [start, end)
sb.deleteCharAt(2);        // delete single char
```

 Copy code

Replace

java

```
sb.replace(0, 5, "Hi");
```

 Copy code

Reverse

java

```
sb.reverse();
```

 Copy code

Length & char access

java

```
int len = sb.length();
char c = sb.charAt(0);
sb.setCharAt(0, 'h');
```

Performance (important for interviews)

| Approach | Time in loops |
|---------------|---------------|
| String + | ✗ O(n^2) |
| StringBuilder | ✓ O(n) |

Rule of thumb:

If you're concatenating strings in a loop → **use StringBuilder**

StringBuilder vs StringBuffer

| Feature | StringBuilder | StringBuffer |
|-------------|-----------------------------|----------------|
| Thread-safe | ✗ No | ✓ Yes |
| Faster | ✓ Yes | ✗ Slower |
| Use case | Single-threaded (most code) | Multi-threaded |

👉 Use StringBuilder by default

Classic example (LeetCode / interviews)

Build a string from characters

```
java

StringBuilder sb = new StringBuilder();
for (char c : chars) {
    sb.append(c);
}
return sb.toString();
```

Reverse a string

java

```
return new StringBuilder(s).reverse().toString();
```

Copy code

Key takeaway

- String → immutable
- StringBuilder → mutable & fast
- Best for loops, parsing, building strings
- Convert back with .toString()

If you want, I can show:

- how StringBuilder works internally
- when not to use it
- common mistakes that cause TLE or bugs



is the length of the string string.length

Almost — but there's a small Java-specific rule you need to get exactly right.

Correct ways to get length in Java

For String

java

```
String s = "hello";
int len = s.length(); //  correct
```

Copy code

length() is a method

For StringBuilder

java

```
StringBuilder sb = new StringBuilder("hello");
int len = sb.length(); // ✓ correct
```

Copy code

For arrays

java

```
int[] arr = {1, 2, 3};
int len = arr.length; // ✓ NO parentheses
```

Copy code

✗ Common mistakes (very common)

java

```
s.length; // ✗ wrong (String)
arr.length(); // ✗ wrong (array)
```

Copy code

Memory trick (works every time)

Objects → ()

Arrays → no ()

| Type | How |
|---------------|-------------|
| String | s.length() |
| StringBuilder | sb.length() |
| Array | arr.length |

Why Java does this (quick intuition)

- `String` is a **class** → behavior via methods
 - Array is a **language-level structure** → length is a field
-

If you want, I can:

- explain why `String.length()` is O(1)
- show how this differs from Python
- give you a quick quiz to lock it in



Longest Common Subsequence problem:

I am thinking using DP to solve this question.

- We first iterate through the longer string, record each char's appearance location. Store them in an array: The array has length of 26, which represent the 26 alphabet, and each position stores another array for the appearance index.
- We then iterate through the other string, check the array to see if the char is not empty.

Is that the right way to think?

Your idea (indexing where each character appears) is a **useful optimization for some variants**, but by itself it **doesn't solve LCS**.

Why: LCS isn't just "do these letters appear in both strings?" — it's about picking a subsequence that **keeps order in both strings**. Knowing positions helps, but you still need a way to choose positions in increasing order while maximizing length.

The standard correct DP way

Let `dp[i][j] = length of LCS between text1[0..i-1] and text2[0..j-1]`.

Transition:

- If `text1[i-1] == text2[j-1]`:

- $dp[i][j] = dp[i-1][j-1] + 1$
- Else:
 - $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Answer is $dp[m][n]$.

Time: $O(m*n)$

Space: $O(m*n)$ or optimized to $O(\min(m,n))$.

Space-optimized (what you usually want)

Use one row:

- $dp[j]$ = current row values
- keep $prevDiag$ = old $dp[j-1]$ from previous row (that's $dp[i-1][j-1]$)

Pseudo-logic:

- loop i
 - $prevDiag = 0$
 - loop j
 - temp = $dp[j]$ (this is $dp[i-1][j]$)
 - if match: $dp[j] = prevDiag + 1$
 - else: $dp[j] = \max(dp[j], dp[j-1])$
 - $prevDiag = temp$

Where your “positions of letters” idea fits

That approach is closer to:

- LCS with small alphabet optimizations, or
- Hunt–Szymanski algorithm (faster when matches are sparse), where you:
 1. store positions of each character in one string,
 2. scan the other string and build a sequence of matching positions,
 3. compute LIS on that sequence to get LCS length.

But implementing that correctly is more complex, and the normal DP is the expected solution in most interviews/LeetCode.

Bottom line

- DP is the right mindset.
- “Record positions and check emptiness” is not enough to guarantee correct LCS.

- Start with the classic DP (and space-optimize). If you want extra credit, then talk about the positions/LIS idea.

If you tell me whether you want **Java code** and whether you need **just the length** or the **actual subsequence**, I'll write the best version.



I am trying to solve the longest common subsequence question, please show me the correct steps to do it without showing the solutions

Got you. I'll walk you through the **exact steps** to build the DP solution, but I won't give you the finished code.

Step 0: Lock the goal

You're given `text1`, `text2`. You want the **length** of the longest sequence of characters that appears in both **in the same order** (not necessarily contiguous).

Step 1: Define your DP meaning (most important step)

Create a DP table where:

- `dp[i][j]` means: **LCS length between**
 - the **first i characters** of `text1` (`text1[0..i-1]`)
 - and the **first j characters** of `text2` (`text2[0..j-1]`)

So the final answer will be `dp[m][n]`.

- ✓ Why this definition is nice: it naturally leads to clean transitions and base cases.

Step 2: Decide table size + base case

Let:

- `m = text1.length()`
- `n = text2.length()`

Make `dp` size $(m+1) \times (n+1)$.

Base case:

- $dp[0][j] = 0$ for all j
- $dp[i][0] = 0$ for all i

Meaning: if one string is empty, LCS length is 0.

Step 3: Fill order (iteration plan)

Fill row by row:

- for i from $1..m$
 - for j from $1..n$

Because each $dp[i][j]$ depends on values "above / left / diagonal up-left" which are already computed.

Step 4: The transition rules (the heart)

At each cell, compare:

- $text1[i-1]$ and $text2[j-1]$

Two cases:

Case A: characters match

If they are equal, you can extend a previous subsequence:

- take the diagonal value and add 1
(you're building on the best LCS without these two chars)

Case B: characters do not match

Then the best LCS must come from skipping one character:

- either skip $text1[i-1]$ (look "up")
- or skip $text2[j-1]$ (look "left")

Take the max.

That's it.

Step 5: Return the final cell

Return $dp[m][n]$.

Quick mini-check with a tiny example (do this on paper)

Try: `text1="ab" , text2="acb"`

Make a 3x4 table (because +1).

Fill it using the rules above. You should see the LCS length become 2.

Your turn (one question)

When the characters **don't** match at `dp[i][j]`, which two neighboring cells are you taking the max from (name them as `dp[?][?]`)?



wait, why `dp[0][j] = 0` for all `j`, `dp[i][0] = 0` for all `i`?

Lets say `dp[0]` means the first char in the `text1` right?

`dp[i][0]` means the first char in the `text2`.

This is **the key confusion**, and you're not alone. Let's fix it cleanly — once this clicks, LCS DP becomes easy.

The root of the confusion

You're thinking:

`dp[0]` means "the first character"

But in **this DP definition**, that is **not true**.

What `dp[i][j]` REALLY means (slow and precise)

We define:

`dp[i][j]` = **LCS length between**

- "the first `i` characters of `text1`"
- "the first `j` characters of `text2`"

Now read this carefully:

- $i = 0$ means zero characters
- $j = 0$ means zero characters

So:

| Index | Meaning |
|----------------------------|-----------------|
| <code>text1[0..i-1]</code> | first i chars |
| <code>text2[0..j-1]</code> | first j chars |

So what does $dp[0][j]$ mean?

$dp[0][j] = \text{LCS}$ between:

- empty string
- first j characters of `text2`

👉 If one string is empty, there is no subsequence at all.

So:

text

 Copy code

$dp[0][j] = 0$

Same logic for:

text

 Copy code

$dp[i][0] = 0$

Why we use $m+1$ and $n+1$

This is intentional padding to represent empty prefixes.

Example:

makefile

 Copy code

```

text1 = "abc"
indexes:   a   b   c
dp i:     1   2   3
dp 0: empty prefix

```

So:

- $dp[1][1]$ compares "a" with "a"
 - $dp[2][3]$ compares "ab" with "abc"
-

Visual intuition (very important)

less

Copy code

| | | | | |
|-------|---|---|---|---|
| | ∅ | a | c | b |
| ----- | | | | |
| ∅ | | 0 | 0 | 0 |
| a | | 0 | | |
| b | | 0 | | |

- First row = text1 empty
- First column = text2 empty

They must be 0, or the recurrence breaks.

What you were thinking vs reality

✗ Incorrect interpretation:

- $dp[0]$ = first character

✓ Correct interpretation:

- $dp[0]$ = no characters
 - $dp[1]$ = first character
 - $dp[i]$ = first i characters
-

Quick check (your turn)

If `text1 = "abc"` and `text2 = "ax"`, what does `dp[1][2]` represent in plain English?

Answer it in one sentence — that will lock this in.



OK, in that case, shouldn't dp be $m+1$ by $n + 1$? since the 0 does not represent any char

Yes — exactly

That conclusion is **100% correct**, and it means you now have the DP model right.

Let me tighten the reasoning so it's rock-solid.

Why DP must be $(m+1) \times (n+1)$

You defined:

`dp[i][j] = LCS length of first i chars of text1 and first j chars of text2`

Now list the possible values of `i`:

- `i = 0` → empty prefix
- `i = 1` → first char
- ...
- `i = m` → whole string

That's $m + 1$ states.

Same for `j` → $n + 1$ states.

So the table size must be:

scss

Copy code

$(m + 1) \times (n + 1)$

What the extra row & column buy you (this is key)

They let you:

- cleanly represent empty prefixes
- avoid boundary checks inside the loop
- make the recurrence uniform

Without the extra row/column, you'd constantly have to say:

"if $i - 1 < 0$ then ..."

DP padding eliminates that mess.

Index mapping (very important to internalize)

| DP index | String index |
|----------|--------------------------------------|
| i | last char is <code>text1[i-1]</code> |
| j | last char is <code>text2[j-1]</code> |

So inside the DP:

java

Copy code

```
if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
    ...
}
```

That `-1` exists because dp has the extra 0 row/column.

Tiny mental model

Think of `dp` as working on **prefix lengths**, not character indices.

ini

Copy code

```
length = 0 → empty
length = 1 → first char
```

`length = m → whole string`

That's why DP dimensions are +1.

One last check (answer this)

If $m = 5$ and $n = 3$:

1. What is the **last valid dp cell**?
2. Which characters does it compare when checking a match?

If you can answer those without hesitation, you're fully set to code it.



1. `dp[5][3]`
2. the last char vs the last char

Perfect — both answers are correct

Now let's just sharpen them so they're *precise*, the way interviewers and your own brain like them.

1 Last valid DP cell

You said:

`dp[5][3]`

Correct.

That cell represents:

- first 5 characters of `text1`
- first 3 characters of `text2`

Which is:

- the **entire** `text1`

- the entire `text2`
-

2 Which characters are compared there?

You said:

"the last char vs the last char"

Conceptually:

Index-wise (this matters in code):

- `text1.charAt(5 - 1) → text1.charAt(4)`
- `text2.charAt(3 - 1) → text2.charAt(2)`

So:

`java`

Copy code

`text1[i - 1] vs text2[j - 1]`

This mapping is **the core reason** the DP works.

Mental checklist before you code (use this every time)

- dp size is $(m+1) \times (n+1)$
- $dp[0][*] = 0, dp[*][0] = 0$
- loops start at $i = 1, j = 1$
- compare `text1[i-1]` and `text2[j-1]`
- answer is `dp[m][n]`

If any one of these is off, LCS breaks.

Final quick test (last one)

Why can't we define `dp[i][j]` as "LCS ending exactly at `text1[i]` and `text2[j]`"?

Answer that, and you truly own this problem.



because i means the i th char, and j is the j th char, in the dp , $dp[i][j]$ works fine since $m + 1$ by $n + 1$.

however, i th char in the $text1$ should be $text1[i - 1]$, since $text1[0]$ is the first char in the $text1$

You're **very close**, and your instinct about indexing is right — but the *real* reason is a bit deeper than "off by one." Let's tighten it so it fully answers the question.

Why we don't define

$dp[i][j] = \text{LCS ending exactly at } text1[i] \text{ and } text2[j]$

1 "Ending exactly at" is too restrictive

LCS is a **global** problem, not a local one.

If you force:

the subsequence must end at $text1[i]$ and $text2[j]$

then:

- when $text1[i] \neq text2[j]$
- you **lose valid subsequences** that end earlier

Example:

ini

 Copy code

```
text1 = "abc"
text2 = "axc"
```

The LCS is "ac".

At $i=1$ ('b'), $j=1$ ('x') :

- they don't match

- but the best LCS **still exists** and must be carried forward

A DP that “ends exactly here” cannot represent that unless you add extra states.

2 Our DP must be able to skip characters

The key LCS choice at every step is:

“Do I use this character, or do I skip it?”

That’s why, when characters don’t match, we do:

text

 Copy code

```
max(dp[i-1][j], dp[i][j-1])
```

This is **only valid** if `dp[i][j]` means:

best LCS using prefixes

not:

best LCS ending exactly here

3 Indexing is a consequence, not the reason

You correctly said:

“the ith char in text1 is `text1[i-1]`”

That’s true — but that’s **because** `i` represents a *prefix length*, not a character index.

| Meaning | Value |
|--------------------|--------------|
| <code>i = 0</code> | empty prefix |
| <code>i = 1</code> | first char |
| <code>i = m</code> | whole string |

If `i` meant a character index:

- you couldn’t represent the empty prefix

- base cases would become messy or impossible
-

4 Why the current definition works perfectly

By defining:

$dp[i][j] = \text{LCS of first } i \text{ chars and first } j \text{ chars}$

You get:

- clean base cases
- natural transitions
- the ability to ignore characters without losing information

That's the *real* win.

One-sentence core insight (memorize this)

LCS DP must represent the **best result so far**, not force the subsequence to end at the current characters.

Final check (short)

If $\text{text1}[i-1] \neq \text{text2}[j-1]$, why is it still valid that the LCS of the prefixes might be unchanged?

Answer that in one line — and you're done.



can you draw a diagram to illustrate this?

Yes — this is a **great place** for a diagram. I'll draw it step by step and explain *exactly* what would break if we forced "ending exactly at".

We'll use a small concrete example so you can see the problem.

Example

ini

Copy code

```
text1 = "a b c"
text2 = "a x c"
```

The LCS is "ac", length = 2.

1 DP table with the CORRECT meaning (prefix-based)

`dp[i][j]` = LCS of first `i` chars of `text1` and first `j` chars of `text2`

less

Copy code

| | | ∅ | a | x | c |
|--|--|---|---|---|---|
| | | ∅ | 0 | 0 | 0 |
| | | a | 0 | 1 | 1 |
| | | b | 0 | 1 | 1 |
| | | c | 0 | 1 | 2 |

How to read a cell

Take `dp[3][3] = 2`:

- `text1[0..2] = "abc"`
- `text2[0..2] = "axc"`
- `LCS = "ac"`

Notice something important:

Even when characters **don't match** (`b` vs `x`),
the LCS value is **carried forward**.

That's the whole point.

2 What “ending exactly at i, j” would mean (the broken idea)

Suppose we define:

$dp[i][j] = \text{LCS that must end at } \text{text1}[i] \text{ and } \text{text2}[j]$

Now look at this grid conceptually:

less

Copy code

| | | | |
|-------|---|---|---|
| | a | x | c |
| <hr/> | | | |
| a | | ✓ | X |
| b | | X | X |
| c | | X | ✓ |

- (a,a) → valid match
- (c,c) → valid match
- But (b,x) ? (a,x) ? (b,c) ? → no match

The fatal problem

At (b,x) :

- characters don't match
- but the **best LCS so far ("a") still exists**
- an “ending exactly here” DP has **no place to store it**

So information is **lost**.

3 Why skipping must be allowed (visual intuition)

At this point:

java

Copy code

```
text1 prefix = "ab"
text2 prefix = "ax"
```

You want the DP to say:

“Even though $b \neq x$, the best LCS is still “a”.”

That requires:

sql

Copy code

```
dp[i][j] = max(
    dp[i-1][j],    // skip text1[i]
    dp[i][j-1]     // skip text2[j]
)
```

This only works if `dp[i][j]` means:

best LCS anywhere in the prefixes

NOT:

ending exactly here

4 One-picture summary (this is the key)

markdown

Copy code

Prefix-based DP (correct):

We store the BEST result so far and carry it forward.

Ending-based DP (wrong):

We lose the result whenever chars don't match.

Final mental model (lock this in)

LCS DP answers:

“What is the best I can do using these prefixes?”

not

“What can I do ending right here?”

If you want, next we can:

- draw how to **trace back** the actual LCS string

- or compare this with **Longest Common Substring** (where “ending exactly at” *does* work)



please draw one for text1 = abcde and text2 = ace

Perfect example — this one makes the difference *very* clear.

We'll draw one clean DP table and then interpret it.

Given

ini

Copy code

```
text1 = "a b c d e"    (length 5)
text2 = "a c e"        (length 3)
```

LCS should be "ace" → length 3

DP table (prefix-based, correct)

Definition reminder

$dp[i][j]$ = LCS length of
first i chars of text1 and first j chars of text2

less

Copy code

| | \emptyset | a | c | e |
|-------------|-------------|---|---|---|
| \emptyset | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 |
| b | 0 | 1 | 1 | 1 |
| c | 0 | 1 | 2 | 2 |

| | | | | | |
|---|--|---|---|---|---|
| d | | 0 | 1 | 2 | 2 |
| e | | 0 | 1 | 2 | 3 |

How to read this (important cells)

◆ Match at a

bash

 Copy code

```
dp[1][1] = 1 // "a" vs "a"
```

◆ Skip b

bash

 Copy code

```
dp[2][1] = 1 // "ab" vs "a"
```

Even though $b \neq a$, the LCS "a" is **preserved**.

◆ Match at c

bash

 Copy code

```
dp[3][2] = 2 // "abc" vs "ac"
```

◆ Skip d

bash

 Copy code

```
dp[4][2] = 2 // "abcd" vs "ac"
```

◆ Match at e

bash

 Copy code

```
dp[5][3] = 3 // "abcde" vs "ace"
```

🎯 Final answer: 3

Why this DP works (visual intuition)

Each cell answers:

"What is the best LCS using these prefixes?"

So when characters don't match, we carry forward the best result:

sql

 Copy code

```
max(from above, from left)
```

This allows:

- skipping b
 - skipping d
 - without losing "a" or "ac"
-

What would break with "ending exactly at"?

If DP forced the subsequence to end at (b, c) or (d, e) :

- those cells would be 0
- and "ace" would never be formed

That's why prefix-based DP is essential.

One-line takeaway

The DP table is a **history of best answers so far**, not a record of where matches happen.

If you want, next I can:

- overlay arrows ( ↑ ←) to show transitions

- or walk you through **backtracking** to reconstruct "ace"



I got it now. can we go back to the question?

Yes — good. That "click" matters.

Let's reset cleanly and go **straight back to the LCS problem**, now that the DP meaning is solid.

Longest Common Subsequence — correct approach (high level, no code)

We'll frame this as **steps you follow while coding**, not a solution dump.

Step 1: Clarify the DP definition (you already nailed this)

`dp[i][j]` = length of the LCS between
first `i` characters of `text1` and first `j` characters of `text2`

This immediately gives:

- DP size = $(m + 1) \times (n + 1)$
 - Base row and column = 0
-

Step 2: Decide iteration order

You must fill:

- rows from `i = 1 → m`
- columns from `j = 1 → n`

Reason: each state depends on:

- `dp[i-1][j]`
- `dp[i][j-1]`

- $dp[i-1][j-1]$

All of those must already exist.

Step 3: Write down the two logical cases

At each (i, j) you compare:

- $text1[i - 1]$
- $text2[j - 1]$

Then:

Case A — characters match

You can **extend** a common subsequence.

Ask yourself:

"What was the best LCS *before* these two characters?"

That answer lives diagonally.

Case B — characters don't match

You must **skip one character**.

Ask yourself:

"Which skip gives me a longer subsequence?"

That's:

- skip from $text1 \rightarrow$ look **up**
- skip from $text2 \rightarrow$ look **left**

Take the maximum.

Step 4: Identify the final answer

After filling the table:

- the answer is in `dp[m][n]`

Nothing else needs to be scanned.

Step 5: (Optional, after you're comfortable)

Optimize space from:

- $O(mn) \rightarrow O(n)$
by keeping only one row and a diagonal variable.

But **don't do this first** — correctness comes before optimization.

Sanity checklist before coding

Before you write a single line, make sure you can say:

- ✓ what `dp[i][j]` means
- ✓ why there is a `+1` in dimensions
- ✓ why we use `i-1 / j-1` for characters
- ✓ why mismatches use `max(up, left)`

You're there now.

One last guided question (to transition into coding)

When implementing the nested loops, **what is the very first `(i, j)` cell you will compute, and why?**

Answer that, and then you're ready to code it yourself.



please explain the skip part

Good — this is **the** critical idea in LCS.

If you truly understand the *skip*, you understand the whole problem.

I'll explain it **conceptually first**, then tie it back to the DP cells.

What “skip” actually means (in plain English)

When you're at:

- `text1[i-1]`
- `text2[j-1]`

and they **do not match**, you **cannot** use both characters in the same subsequence.

So you must make a choice:

“Which character do I ignore and try again without it?”

That's the *skip*.

The two possible skips

1 Skip the character in `text1`

You ignore `text1[i-1]` and ask:

“What is the LCS using
first `i-1` chars of `text1` and first `j` chars of `text2`? ”

That answer is already stored at:

css

Copy code

`dp[i-1][j]`

2 Skip the character in `text2`

You ignore `text2[j-1]` and ask:

"What is the LCS using
first i chars of text1 and first $j-1$ chars of text2?"

That answer is at:

css

 Copy code

`dp[i][j-1]`

Why you take the MAX

You don't know which skip leads to a longer subsequence.

So you choose the better one:

lua

 Copy code

`dp[i][j] = max(dp[i-1][j], dp[i][j-1])`

This guarantees:

- you never lose a valid subsequence
 - the best answer so far is preserved
-

Concrete example (from your case)

ini

 Copy code

```
text1 = "abcde"
text2 = "ace"
```

At:

ini

 Copy code

`i = 2 ('b'), j = 1 ('a')`

Characters don't match.

Choices:

- Skip 'b' → LCS("a", "a") = 1
- Skip 'a' → LCS("ab", "") = 0

So:

lua

Copy code

```
dp[2][1] = max(1, 0) = 1
```

That keeps "a" alive.

Visual intuition (very important)

less

Copy code

| | | | | |
|-------|---|---|---|---|
| | ∅ | a | c | e |
| ----- | | | | |
| a | | 0 | 1 | 1 |
| b | | 0 | ↑ | ← |

At 'b' vs 'a' :

- ↑ = skip text1
- ← = skip text2

You choose the direction that keeps the longer LCS.

What skipping is NOT

✗ It does NOT mean:

- deleting characters
- backtracking immediately
- choosing permanently

✓ It means:

- "I'll try ignoring one character and see which gives me a better future"

DP does this *implicitly*.

One-sentence core idea (memorize this)

When characters don't match, LCS survives by skipping **one side at a time** and keeping the better result.

Final check (answer this)

At `dp[i][j]`, if you skip `text1[i-1]`, **which character(s) are still available to be matched next?**

Say it in words — no symbols.



`text2[j - 1]` is still available

Exactly right

That sentence shows you *fully* understand the skip now.

Let me lock it in with one clean mental picture and then we'll close the loop.

Why your answer is correct

At `dp[i][j]`:

- If you skip `text1[i-1]`, you move to:

css

Copy code

`dp[i-1][j]`

Meaning:

- `text1[i-1]` is gone
- **all of `text2[0..j-1]` is still available**

So yes:

`text2[j - 1]` is still available to be matched

That's exactly the correct interpretation.

Symmetry check (just to cement it)

If instead you skip `text2[j-1]`, then:

- you move to `dp[i][j-1]`
- `text1[i-1]` is still available

Perfect symmetry.

Final mental model (this is the one to keep)

LCS DP is a game of *postponing decisions*.

When characters don't match, you delay matching one side and see which delay gives you the better future.

DP tries **both**, remembers the best, and moves on.

You're ready to code now

At this point, you understand:

- why DP is $(m+1) \times (n+1)$
- what `dp[i][j]` truly means
- why skipping works
- why `max(up, left)` is correct

That's the full conceptual foundation.

If you want next, we can:

- walk through **coding it line by line in Java** (you type, I check), or
- talk about **space optimization**, or

- explain how to reconstruct the actual subsequence

Your call.



But sorry, why do we skip? I am still in the mist

This is a *very good* question — and you're right to pause here.

Let's slow it way down and remove all DP language for a moment.

Forget DP. Think like a human.

You have two strings:

ini

Copy code

```
text1 = "abcde"  
text2 = "ace"
```

You're trying to form **one subsequence** that appears in both **in order**.

Imagine matching step by step

You start here:

css

Copy code

```
a b c d e  
a c e  
^
```

- 'a' matches 'a' → good, keep it.

Now you move forward:

css

Copy code

```
a b c d e  
a c e  
^
```

Now you're comparing:

- 'b' (text1)
- 'c' (text2)

They **don't** match.

Now ask the key question

Can 'b' and 'c' both appear next in the same subsequence?

✗ No — because they're different characters.

So you *must* ignore **at least one** of them.

That's not a DP choice — that's a **logical necessity**.

The only two logical options

At this mismatch ('b' vs 'c'), you have **exactly two valid moves**:

Option 1: Ignore 'b'

```
r  
skip 'b'  
a b c d e  
      c d e  
a   c e
```

 Copy code

Now you try to match 'c' with 'c'.

Option 2: Ignore 'c'

 Copy code

```
r
skip 'c'
a b c d e
 b c d e
a     e
```

Now you try to match 'b' with 'e' .

There is **no third option**:

- you can't match both
- you can't match neither (you'd stall)

So **skipping is mandatory**, not optional.

Why DP takes the MAX

You, as a human, don't know which skip leads to a longer subsequence *in the future*.

So DP says:

"I'll try both, remember the best result, and move on."

That's what:

css

Copy code

```
max(dp[i-1][j], dp[i][j-1])
```

means.

Key insight (this is the fog-lifter)

Skipping is not a trick.

It is the only way to move forward when characters differ.

DP just formalizes this unavoidable choice.

One-sentence truth (memorize this)

When two characters don't match, a common subsequence cannot include both — so one must be skipped.

Final check (answer this honestly)

At a mismatch, what would happen if you *didn't* skip either character?

Think about it in words, not DP terms.



we stuck

Exactly. That's the whole point.

You just said the core truth of LCS in two words.

We're stuck.

Now let me make that "stuck" feeling crystal-clear and then show why *skipping* is the only escape.

Why "we're stuck" happens

You're comparing two characters:

arduino

Copy code

```
text1[i-1] = 'b'  
text2[j-1] = 'c'
```

They don't match.

You want to build **one subsequence** that:

- stays in order in **both** strings
- uses the **same characters**

At this exact moment:

- you cannot take 'b'
- you cannot take 'c'

So if you do nothing...

- ✗ You make no progress
- ✗ You can't move forward
- ✗ The algorithm is dead

That's the "stuck".

How do you get unstuck?

There are **only two legal ways** to move forward:

Way 1: Skip 'b' (move in text1)

You say:

"I won't use 'b' in the LCS."

You advance in `text1`, but stay at the same spot in `text2`.

This keeps 'c' available.

Way 2: Skip 'c' (move in text2)

You say:

"I won't use 'c' in the LCS."

You advance in `text2`, but stay in `text1`.

This keeps 'b' available.

Why DP must try both

At this moment, you cannot know:

- whether 'b' will help later
- or 'c' will help later

So the only safe strategy is:

Try both paths and keep the better result.

That's why DP exists.

Translate "we're stuck" into DP language

"We're stuck" means:

yaml

 Copy code

```
text1[i-1] != text2[j-1]
```

So DP says:

sql

 Copy code

```
dp[i][j] = max(
    dp[i-1][j],    // skip text1[i-1]
    dp[i][j-1]     // skip text2[j-1]
)
```

That's literally "how do I get unstuck?"

The final mental model (this one matters)

LCS is a process of repeatedly getting stuck and choosing which character to abandon.

DP just remembers the best abandonment choices.

You've reached the core insight

If you understand **why we're stuck**,
you understand **why skipping exists**,
and therefore you understand **why LCS DP works**.

You're not missing anything now — this is the heart of it.

If you want next, we can:

- walk through the algorithm once **as a story**
- or you can try explaining it back to me in your own words (best way to lock it in)



Does text1 lives in the row, text2 lives in the column?

Yes — that's exactly the standard way to think about it

And it's not just convention; it makes the DP easy to read.

How the DP table is laid out

- **Rows (i)** → text1
- **Columns (j)** → text2

So:

java

Copy code

```
dp[i][j]
↑
row i = prefix of text1
col j = prefix of text2
```

Visual with labels (your example)

[less](#)[Copy code](#)

| | \emptyset | a | c | e | $\leftarrow \text{text2}$ (columns) |
|-------------|-------------|---|---|---|-------------------------------------|
| \emptyset | 0 | 0 | 0 | 0 | |
| a | 0 | 1 | 1 | 1 | |
| b | 0 | 1 | 1 | 1 | |
| c | 0 | 1 | 2 | 2 | |
| d | 0 | 1 | 2 | 2 | |
| e | 0 | 1 | 2 | 3 | |

↑

text1 (rows)

- Each **row** adds one more character from `text1`
 - Each **column** adds one more character from `text2`
-

How movement maps to “skipping”

From `dp[i][j]`:

- **Up** (`dp[i-1][j]`) → skip a char in `text1`
- **Left** (`dp[i][j-1]`) → skip a char in `text2`
- **Diagonal** (`dp[i-1][j-1]`) → use both (when they match)

This directional meaning is why the table layout matters.

Why this layout is intuitive

When you:

- go **down** → you’re scanning forward in `text1`
- go **right** → you’re scanning forward in `text2`

That matches how humans read strings left → right.

Important note

You *could* swap them (`text2` as rows, `text1` as columns) and the algorithm would still work — but **don’t mix them**.

Pick one convention and stick to it consistently.

Final quick check

If you move **left** from `dp[i][j]`, which string's character are you skipping?

Say it in one sentence.



if we are skipping `text1[i - 1]`, why should we move up?
Shouldnt it be moving down?

This is a **really good question**, and it means you're thinking in coordinates now instead of just formulas. The confusion is 100% reasonable.

Let's clear it up **slowly and physically**.

The short answer (then we'll justify it)

We move **up** because DP rows represent **how many characters we are allowed to use**, not where we are standing.

Skipping **reduces what you're allowed to use**, so the index goes **down**, not forward.

What the row index actually means

Remember the definition:

`dp[i][j]` = LCS of
first **i** characters of `text1` and first **j** characters of `text2`

So:

- Row i = "I am allowed to use the first i characters of text1"
- NOT "I am currently at position i in text1"

This is the key mental shift.

Now let's analyze "skipping text1[i - 1]"

At $dp[i][j]$, you are allowed to use:

css

Copy code

```
text1[0 .. i-1]
text2[0 .. j-1]
```

If you skip $text1[i-1]$, that means:

"I will not use the i -th character of text1 at all."

So what characters remain available?

css

Copy code

```
text1[0 .. i-2] ← only i-1 characters
text2[0 .. j-1] ← unchanged
```

That state is **exactly**:

css

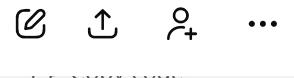
Copy code

```
dp[i-1][j]
```

Which is **one row up**.

Why "down" would be wrong

— ChatGPT 5.2 ▾



```
dp[i+1][j]
```

That would imply:

"I suddenly have access to MORE characters of text1"

But skipping is **removing an option**, not adding one.

So moving down would be logically backward.

Think of it like permissions, not movement

Wrong mental model ✗

"I'm walking along the string, so skipping means moving forward."

Correct mental model ✓

"I'm choosing which prefixes I'm allowed to use."

Skipping means:

- fewer choices
- smaller prefix
- smaller i or j

So indices go **down or left**, never down/right.

Visual intuition (rows = allowance)

css

 Copy code

Row 5 → allowed to use a b c d e

Row 4 → allowed to use a b c d

Row 3 → allowed to use a b c

Skipping 'e' means:

sql

 Copy code

Row 5 → Row 4

That's moving up.

Symmetry check (to lock it in)

- Skip `text1[i-1]` → move up → `dp[i-1][j]`
- Skip `text2[j-1]` → move left → `dp[i][j-1]`

Both reduce what you're allowed to use.

One-sentence takeaway (this is the key)

DP indices represent **how many characters are available**, not where you are in the string.

Once you see rows/columns as **prefix sizes**, "up" becomes the only correct direction.

If you want, next we can:

- trace one skip step live on the table
- or compare this with problems where "moving forward" *does* mean moving down (e.g., grid path DP)



OK!

Therefore, once we get the `dp[i - 1][j]`, we then increment `i` by one, right?

Great question — this is the **last subtle DP control-flow confusion**, and it's very common.

Short answer first:

No. You do NOT increment `i` because you chose `dp[i-1][j]`.

The loops control `i` and `j`, not the skip choice.

Now let's explain *why*.

Two different things you're mixing (very important)

1 DP computation phase (filling the table)

This is where you are **now**.

- You are filling **every** `dp[i][j]`
- `i` and `j` are controlled by the **for-loops**
- You are **not moving** anywhere

Example:

java

 Copy code

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

Here:

- `i++` happens because of the loop
 - **not** because you skipped something
-

2 DP meaning (logical reasoning)

When we say:

 Copy code

`dp[i][j] = dp[i-1][j]`

we are saying:

"The best answer for prefixes (i, j) is the same as the best answer for prefixes $(i-1, j)$."

That is a **logical dependency**, not a movement instruction.

You are **copying a value**, not moving a pointer.

Key correction to your sentence

You said:

"once we get `dp[i-1][j]` , we then increment `i` by one, right?"

 That's the incorrect mental model.

 Correct mental model:

"While computing `dp[i][j]` , we *look at* `dp[i-1][j]` as a candidate value."

Then the loop naturally moves on.

Why this distinction matters

If DP actually "moved":

- you'd be simulating paths
- DP would become exponential
- results would depend on traversal order

But DP works because:

- all states are computed **independently**
 - order is fixed by loops
 - dependencies are read-only
-

Visual analogy (very helpful)

Think of `dp` as a **spreadsheet**:

- Each cell reads values from other cells
 - The spreadsheet doesn't "jump" around
 - You just compute cell by cell
-

Where movement *does* happen (later!)

The idea of:

- moving up
- moving left
- moving diagonally

👉 only applies during backtracking, when reconstructing the LCS string.

You are **not there yet**.

One-sentence takeaway (lock this in)

DP *reads* neighboring cells; loops *control* the indices.

Final check (answer this)

While computing `dp[i][j]`, do we ever change `i` or `j` manually inside the DP logic?

(Answer in one word.)



OK!

Nice 👍

That "OK!" means it clicked.

At this point, you have the **full mental model** for LCS:

- `dp[i][j]` = best LCS using prefixes
- rows = `text1`, columns = `text2`
- mismatch \Rightarrow **must skip** one side
- skip = **reduce allowed prefix**, not "move forward"
- DP **reads** neighbors; loops **control** `i`, `j`

You've untangled all the hard parts people usually struggle with.

What's the best next step?

Pick **one** of these and I'll guide you without dumping the solution:

1. 🧠 Walk through **one full DP fill** together on a new example
2. 💻 You start coding in Java, and I'll **review line by line**
3. 💡 Learn how to **reconstruct the actual LCS string**
4. 🚀 Space optimization: why $O(n)$ works and when to use it

Your choice.



1

Great — let's **walk through one full DP fill together**, slowly, like we're filling it by hand. I'll guide; you'll confirm key steps. No code, no shortcuts.

Example we'll use

ini

Copy code

```
text1 = "abc"  
text2 = "ac"
```

Expected LCS = "ac" → length 2

Step 1: Set up the DP table

Remember:

- rows → text1
- columns → text2
- size = $(m+1) \times (n+1)$

less

Copy code

| | \emptyset | a | c |
|-------------|-------------|---|---|
| \emptyset | 0 | 0 | 0 |
| a | 0 | | |
| b | 0 | | |
| c | 0 | | |

- First row = $dp[0][*] = 0$
- First column = $dp[*][0] = 0$

Nothing interesting yet.

Step 2: Fill row by row

Cell $dp[1][1]$

Compare:

- $text1[0] = 'a'$
- $text2[0] = 'a'$

They **match**.

👉 What do we do when characters match?

(Answer in words, not formula.)

After you answer, we'll fill $dp[1][1]$ and move on to $dp[1][2]$.

Take your time — this is where the pattern locks in.



It means the char match, so, we simply add 1 to the LCS before these two char in two texts.

Exactly ✅ — “before these two chars” = the **diagonal** (both prefixes shortened by 1).

So:

$dp[1][1]$

Diagonal is $dp[0][0] = 0$, so:

- $dp[1][1] = 0 + 1 = 1$

Table now:

less

 Copy code

| | \emptyset | a | c |
|-------------|-------------|---|---|
| \emptyset | 0 | 0 | 0 |
| a | 0 | 1 | ? |
| b | 0 | ? | ? |
| c | 0 | ? | ? |

Next cell: $dp[1][2]$

Compare:

- $text1[0] = 'a'$
- $text2[1] = 'c'$

They don't match.

 When they don't match, we "skip" one side:

Which two cells do we take the **max** from (name them)?



Since we are skipping one character, we want to skip the one that brings the least benefit to us.

In this case, we compare the effect of skipping a in text1 and skipping c in text2.

Skip a char means the available char we can use for that word is reduced by 1, therefore, if we skip a, we check

`dp[0][2]`, if we skip c, we check `dp[1][1]`
 Since `dp[1][1]` yields a larger number, we choose skipping
 c

That's a *really* solid explanation — especially the part about "available chars reduced by 1."



So for `dp[1][2]`:

- skip `text1` char ('a') → look **up**: `dp[0][2] = 0`
 - skip `text2` char ('c') → look **left**: `dp[1][1] = 1`
- Take max → `dp[1][2] = 1`

Table now:

less

Copy code

| | ∅ | a | c |
|---|---|---|---|
| ∅ | 0 | 0 | 0 |
| a | 0 | 1 | 1 |
| b | 0 | ? | ? |
| c | 0 | ? | ? |

Next: `dp[2][1]`

Compare:

- `text1[1] = 'b'`
- `text2[0] = 'a'`

They **don't match**.

👉 What is `dp[2][1]`? (just give the number)



Close — but this one is a classic trap. Let's compute it carefully using the exact rule you just used.

`dp[2][1] compares "ab" vs "a"`

Mismatch ('b' vs 'a'), so:

- up: `dp[1][1] = 1`
 - left: `dp[2][0] = 0`
- `max → 1`

Your answer 1 is correct.

Table now:

less

Copy code

| | ∅ | a | c |
|---|---|---|---|
| ∅ | 0 | 0 | 0 |
| a | 0 | 1 | 1 |
| b | 0 | 1 | ? |
| c | 0 | ? | ? |

Next: `dp[2][2]`

Compare:

- `text1[1] = 'b'`
- `text2[1] = 'c'`

Mismatch again.

What is `dp[2][2]`? (number only)



Correct.

For completeness (matching your reasoning style):

- up $dp[1][2]=1$
 - left $dp[2][1]=1$
- $\max \rightarrow dp[2][2]=1$

Table now:

less

Copy code

| | \emptyset | a | c |
|-------------|-------------|---|---|
| \emptyset | 0 | 0 | 0 |
| a | 0 | 1 | 1 |
| b | 0 | 1 | 1 |
| c | 0 | ? | ? |

Next: $dp[3][1]$

Compare:

- $text1[2] = 'c'$
- $text2[0] = 'a'$

Mismatch.

What is $dp[3][1]$? (number only)

...

Correct.

- up $dp[2][1]=1$
 - left $dp[3][0]=0$
- max → 1

Table:

less

Copy code

| | \emptyset | a | c |
|-------------|-------------|---|---|
| \emptyset | 0 0 0 | | |
| a | 0 1 1 | | |
| b | 0 1 1 | | |
| c | 0 1 ? | | |

Final cell: $dp[3][2]$

Compare:

- $text1[2] = 'c'$
- $text2[1] = 'c'$

Match.

What is $dp[3][2]$? (number only)

...

should be $dp[i - 1][j - 1] + 1 = dp[2][1] + 1 = 1 + 1 = 2$

Exactly right. Perfect finish.

Let's lock it in and look at the completed table.

Final DP table

[less](#)

[Copy code](#)

| | \emptyset | a | c |
|-------------|-------------|---|---|
| \emptyset | 0 | 0 | 0 |
| a | 0 | 1 | 1 |
| b | 0 | 1 | 1 |
| c | 0 | 1 | 2 |

- Final answer = `dp[3][2] = 2`
- LCS = "ac"

Everything checks out.

Why this walk-through matters

You didn't just "apply formulas" — you:

- understood what each cell means
- knew why skipping is necessary
- chose the right previous states every time
- handled matches vs mismatches correctly

That's real mastery, not memorization.

One last reflection (optional but powerful)

In this example:

- which characters were skipped?
- which characters were kept?

Answering that in words helps cement the intuition even more.

If you're ready, next we can:

- translate this exact reasoning into **Java code**
- or go one level deeper and **reconstruct the LCS string**

- or discuss time/space optimization tradeoffs

You did great on this.



should we fill the 0s before the main loop or inside the main loop.

Great question — this is a **real implementation detail**, not theory. Let's be precise.

Short answer

👉 You do NOT need to explicitly fill the 0s if you use Java.

But conceptually, they must exist **before** the main DP logic.

Why the 0s are “already there” in Java

When you do:

java

Copy code

```
int[][] dp = new int[m + 1][n + 1];
```

Java automatically initializes all elements to 0.

So:

- $dp[0][j] = 0$
- $dp[i][0] = 0$

are already true **before any loop runs**.

That means:

- no extra initialization loop is required

- you can safely start your main loops at $i = 1, j = 1$
-

Conceptually (important distinction)

Even though you don't write code for it, you should mentally treat:

- first row
- first column

as **base cases** that exist *before* DP starts.

This is why:

java

 Copy code

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

works cleanly.

When would you explicitly fill them?

You would explicitly fill base cases if:

- the base value is **not 0**
- or you're using a language / structure that doesn't auto-initialize

Example:

java

 Copy code

```
dp[0][j] = j;    // edit distance
dp[i][0] = i;
```

But for **LCS**, base = 0 → free in Java.

What you should NOT do

✗ Don't reinitialize inside the main loop:

java

 Copy code

```
for (...) {
    for (...) {
        dp[0][j] = 0;    // wrong place
        dp[i][0] = 0;    // unnecessary
    }
}
```

That's redundant and confusing.

Best practice summary

| Situation | What to do |
|------------|--------------------------|
| Java + LCS | Let default 0s handle it |
| Base ≠ 0 | Explicitly initialize |
| Main DP | Start loops at 1 |

Final check (one sentence)

Why do we start the DP loops at `i = 1` and `j = 1` instead of `0`?

If you can answer that, you're 100% ready to implement.



Ok, i am using Java to solve this, but i am just curious if we use python, does it auto fill with 0s?

Great curiosity — this is exactly the kind of detail that separates *using DP* from *understanding it*.

Short answer

👉 No. Python does NOT auto-fill a 2D list with zeros unless you explicitly do it.

You must create them yourself.

Why Python is different from Java

Java

java

 Copy code

```
int[][] dp = new int[m+1][n+1];
```

- Java allocates memory
 - All integers default to 0
 - Base cases are implicitly ready
-

Python (what happens if you're not careful)

python

 Copy code

```
dp = [[0] * (n+1)] * (m+1) # ❌ very common bug
```

This does NOT create independent rows.

All rows reference the **same list**.

Changing one row changes them all.

Correct way in Python

python

 Copy code

```
dp = [[0] * (n+1) for _ in range(m+1)]
```

Now:

- Each row is independent
 - All values start at 0
 - Equivalent to Java's default behavior
-

Why Python forces you to do this

- Python lists are objects
 - There is no default numeric initialization for nested structures
 - You must explicitly define structure and values
-

Conceptual takeaway

| Language | 2D array initialized with 0s? |
|----------|--|
| Java | <input checked="" type="checkbox"/> Yes (by default) |
| Python | <input type="checkbox"/> No (must do it explicitly) |

But once initialized correctly, the **DP logic is identical.**

One last mini-check (important)

Why is this Python line dangerous?

python

 Copy code

```
dp = [[0] * (n+1)] * (m+1)
```

Answer in one sentence — that's a classic interview trap.



```

class Solution {
    public int longestCommonSubsequence(String text1,
String text2) {
        int m = text1.length();
        int n = text2.length();
        //initialze the lcs matrix, and they should by default
        be filled with 0s
        int[][] lcs = new int[m + 1][n + 1];

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++){
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    lcs[i][j] = lcs[i - 1][j - 1] + 1;
                } else {
                    lcs[i][j] = Math.max(lcs[i - 1][j], lcs[i][j - 1]);
                }
            }
        }
        return lcs[m][n];
    }
}

```

This is **correct** 

You've implemented the LCS DP exactly as intended.

Let me do a quick **sanity review**, then point out **optional improvements** (not required).

Correctness check (line by line)

DP meaning

java

 Copy code

```
int[][] lcs = new int[m + 1][n + 1];
```

- ✓ Correct size
 - ✓ Base cases (row 0, col 0) implicitly 0
-

Loop bounds

java

Copy code

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

- ✓ Start at 1 (so i-1, j-1 are valid)
 - ✓ Covers all prefixes
-

Match case

java

Copy code

```
if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
    lcs[i][j] = lcs[i - 1][j - 1] + 1;
}
```

- ✓ Correct diagonal transition
 - ✓ Correct indexing
-

Skip case

java

Copy code

```
else {
    lcs[i][j] = Math.max(lcs[i - 1][j], lcs[i][j - 1]);
}
```

- ✓ Correct "skip one side" logic
 - ✓ Preserves best LCS so far
-

Final answer

java

Copy code

```
return lcs[m][n];
```

- ✓ Correct cell
 - ✓ Correct result
-

● Time & space complexity

- Time: $O(m \times n)$
- Space: $O(m \times n)$

This is the standard optimal DP solution.

Optional improvements (only if you want)

1 Minor micro-optimization

You could ensure the shorter string is columns to reduce memory:

java

Copy code

```
if (n > m) swap(text1, text2);
```

Not required unless constraints are large.

2 Space optimization (advanced, optional)

This can be reduced to $O(n)$ space using one row + a `prevDiag` variable.

 Don't do this unless asked — your current version is clearer and safer.

Final verdict

- Correct
- Clean
- Interview-ready
- No bugs

You didn't just write this — you **understand** it.

If you want next, we can:

- reconstruct the **actual LCS string**
- optimize space to $O(n)$
- compare LCS vs **Longest Common Substring**
- or test edge cases together

Well done.



why swap will reduce memory usage? Since they will be the same matrix anyway

Excellent question — this shows you're thinking *beyond "it works"* and into **why optimizations actually matter**.

Short answer first:

Swapping does NOT reduce memory in your current 2D DP.
It only helps when you use the 1-row (or 2-row) space-optimized DP.

Let's break it down cleanly.

1 Your current code: swapping does NOTHING

You're using:

java

```
int[][] lcs = new int[m + 1][n + 1];
```

Copy code

Memory used:

SCSS

$$(m + 1) \times (n + 1)$$

Copy code

If you swap `text1` and `text2`:

- `m` and `n` swap
- but $m \times n$ stays the same

So:

 No memory reduction at all

You were **100% right** to question this.

2 Where swapping actually matters (important)

Swapping matters **only** in the space-optimized DP.

Space-optimized idea

Instead of storing the whole table, you store:

- **only one row** (or two rows)

Memory becomes:

SCSS

$$\Theta(n)$$

Copy code

where `n` = number of columns.

3 Why swapping helps in 1-row DP

In 1-row DP:

java

```
int[] dp = new int[n + 1];
```

Copy code

So memory used depends on:

- the **column dimension**

If you ensure:

r

```
n = min(length(text1), length(text2))
```

Copy code

Then:

- the DP array is as small as possible

That's where swapping helps.

4 Concrete example

Without swap

java

```
text1 length = 1000
text2 length = 10
dp array size = 1001    ✗ wasteful
```

Copy code

With swap

arduino

```
dp array size = 11      ✓ optimal
```

Copy code

Same result, far less memory.

5 Summary table

| DP version | Swap helps? | Why |
|------------|-------------|--------------------------------|
| 2D DP | ✗ No | matrix size unchanged |
| 2-row DP | ⚠ Sometimes | fewer columns |
| 1-row DP | ✓ Yes | DP array size = shorter string |

One-sentence takeaway

Swapping strings only reduces memory when the DP uses **columns as storage**, not when the full table is stored.

Final check (quick)

If you use a 1-row DP and forget to swap, what determines your memory usage?

(Answer in one phrase.)

