132. Palindrome Partitioning II

DSA👻	
	54
topic 🦦	Front Partition

🦧 Desc	# Remember this for palindrome
	
	`for length in range(3, n + 1): # length = 3 to n`
	`for i in range(n - length + 1): # valid start indices`
	`j = i + length - 1`

Precompute palindrome substrings

```
is_palindrome = [[False] * n for _ in range(n)]

for i in range(n):
    is_palindrome[i][i] = True
for i in range(n - 1):
    is_palindrome[i][i + 1] = (s[i] == s[i + 1])
for length in range(3, n + 1):
    for i in range(n - length + 1):
        j = i + length - 1
        is_palindrome[i][j] = (s[i] == s[j]) and is_palindrome[i + 1][j - 1]
```

✅ Goal of This Loop

We're filling in a 2D table `is_palindrome[i][j]` such that:

> is_palindrome[i][j] == True if s[i..j] is a palindrome.

To do this efficiently, we **build palindromes of increasing length**, reusing results of smaller ones.

✅ Step-by-Step Recap

🔷 Step 1: Length-1 substrings (every character)

```
for i in range(n):
    is_palindrome[i][i] = True
```

- "a", "b", "c", ... are all palindromes.

🔷 Step 2: Length-2 substrings

```
for i in range(n - 1):
    is_palindrome[i][i + 1] = (s[i] == s[i + 1])
```

- "aa", "bb", etc.

🔷 Step 3: Length-3 and above

Now for substrings of **length ≥ 3**, we check this condition:

```
s[i] == s[j] and is_palindrome[i + 1][j - 1]
```

Why?

To be a palindrome:

- The **first and last characters** `s[i]` and `s[j]` must be equal.
- And the **inner substring** `s[i+1..j-1]` must also be a palindrome (which we already computed!).

🔁 Loop Explanation

```
for length in range(3, n + 1):        # length = 3 to n
    for i in range(n - length + 1):   # valid start indices
        j = i + length - 1            # compute end index
        is_palindrome[i][j] = (s[i] == s[j]) and is_palindrome[i + 1][j - 1]
```

💡 Example: s = "ababa"

We will build palindromes like this:

Length	i	j	s[i:j+1]	is_palindrome[i+1][j-1]	s[i]==s[j]	is_palindrome[i][j]
3	0	2	"aba"	is_palindrome[1][1] = T	a == a ✔	True
3	1	3	"bab"	is_palindrome[2][2] = T	b == b ✔	True
3	2	4	“aba”	is_palindrome[3][3] = T	a == a ✔	True
4	0	3	“abab”	is_palindrome[1][2] = F	a ≠ b ❌	False
4	1	4	“baba”	is_palindrome[2][3] = F	b ≠ a ❌	False
5	0	4	"ababa"	is_palindrome[1][3] = T	a == a ✔	True

📌 Visualization of the Recursion

A substring `s[i..j]` is a palindrome **iff**:

- `s[i] == s[j]` ✅
- and `s[i+1..j-1]` is a palindrome ✅ (we already checked it earlier when length was smaller)

We build from **length 1 → length n**, which guarantees inner substrings are already computed.

🧠 Why We Do It?

Because checking each substring on the fly (`s[i:j+1] == s[i:j+1][::-1]`) is `O(n)` per check. So in a nested loop, that would be `O(n³)` total. This reduces palindrome check to **O(1)**.
