Longest Common Substring

We'll cover the following

- Problem Statement
- Basic Solution
  - Code
- Top-down Dynamic Programming with Memoization
  - Code
- Bottom-up Dynamic Programming
- Challenge

Problem Statement

Given two strings ‘s1’ and ‘s2’, find the length of the longest substring which is common in both the strings.

Example 1:

```
Input: s1 = "abdca"
       s2 = "cbda"
Output: 2
Explanation: The longest common substring is "bd".
```

Example 2:

```
Input: s1 = "passport"
       s2 = "ppsspt"
Output: 3
Explanation: The longest common substring is "ssp".
```

Basic Solution

A basic brute-force solution could be to try all substrings of ‘s1’ and ‘s2’ to find the longest common one. We can start matching both the strings one character at a time, so we have two options at any step:

1. If the strings have a matching character, we can recursively match for the remaining lengths and keep a track of the current matching length.
2. If the strings don’t match, we start two new recursive calls by skipping one character separately from each string and reset the matching length.

The length of the Longest Common Substring (LCS) will be the maximum number returned by the three recurse calls in the above two options.

Code #

Here is the code:

Java JS Python3 C++

```
def find_LCS_length(s1, s2):
  return find_LCS_length_recursive(s1, s2, 0, 0, 0)


def find_LCS_length_recursive(s1, s2, i1, i2, count):
  if i1 == len(s1) or i2 == len(s2):
    return count

  if s1[i1] == s2[i2]:
    count = find_LCS_length_recursive(s1, s2, i1 + 1, i2 + 1, count + 1)

  c1 = find_LCS_length_recursive(s1, s2, i1, i2 + 1, 0)
  c2 = find_LCS_length_recursive(s1, s2, i1 + 1, i2, 0)

  return max(count, max(c1, c2))


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))


main()
```

Because of the three recursive calls, the time complexity of the above algorithm is exponential $O(3^{m+n})$, where 'm' and 'n' are the lengths of the two input strings. The space complexity is $O(m+n)$, this space will be used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use an array to store the already solved subproblems.

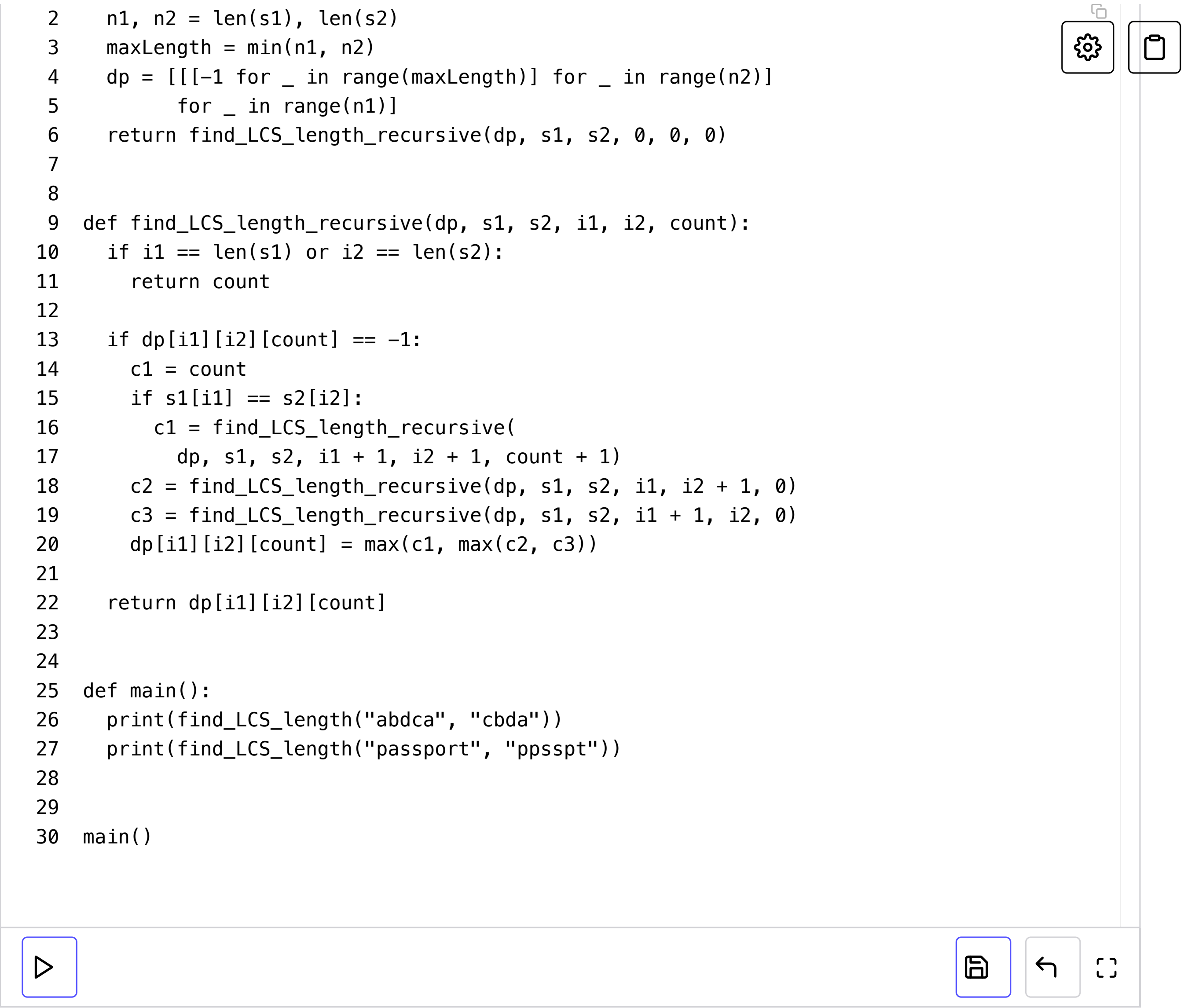
The three changing values to our recursive function are the two indexes (i1 and i2) and the 'count'. Therefore, we can store the results of all subproblems in a three-dimensional array. (Another alternative could be to use a hash-table whose key would be a string (i1 + "|" i2 + "|" + count)).

Code #

Here is the code:

Java JS Python3 C++

```
def find_LCS_length(s1, s2):
```



```
  n1, n2 = len(s1), len(s2)
  maxLength = min(n1, n2)
  dp = [[[-1 for _ in range(maxLength)] for _ in range(n2)]
        for _ in range(n1)]
  return find_LCS_length_recursive(dp, s1, s2, 0, 0, 0)


def find_LCS_length_recursive(dp, s1, s2, i1, i2, count):
  if i1 == len(s1) or i2 == len(s2):
    return count

  if dp[i1][i2][count] == -1:
    c1 = count
    if s1[i1] == s2[i2]:
      c1 = find_LCS_length_recursive(
        dp, s1, s2, i1 + 1, i2 + 1, count + 1)
    c2 = find_LCS_length_recursive(dp, s1, s2, i1, i2 + 1, 0)
    c3 = find_LCS_length_recursive(dp, s1, s2, i1 + 1, i2, 0)
    dp[i1][i2][count] = max(c1, max(c2, c3))

  return dp[i1][i2][count]


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))


main()
```

Bottom-up Dynamic Programming

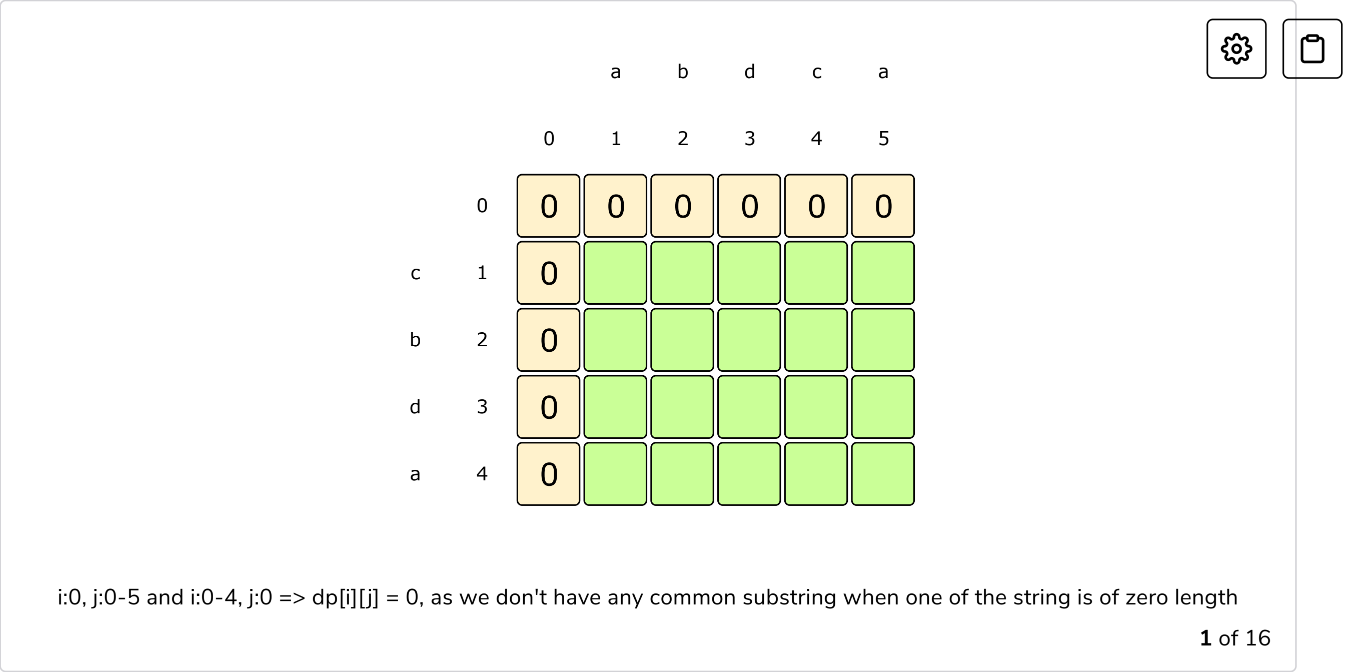
Since we want to match all the substrings of the given two strings, we can use a two-dimensional array to store our results. The lengths of the two strings will define the size of the two dimensions of the array. So for every index 'i' in string 's1' and 'j' in string 's2', we have two options:

1. If the character at `s1[i]` matches `s2[j]`, the length of the common substring would be one plus the length of the common substring till `i-1` and `j-1` indexes in the two strings.
2. If the character at the `s1[i]` does not match `s2[j]`, we don't have any common substring.

So our recursive formula would be:

```
if s1[i] == s2[j]
  dp[i][j] = 1 + dp[i-1][j-1]
else
  dp[i][j] = 0
```

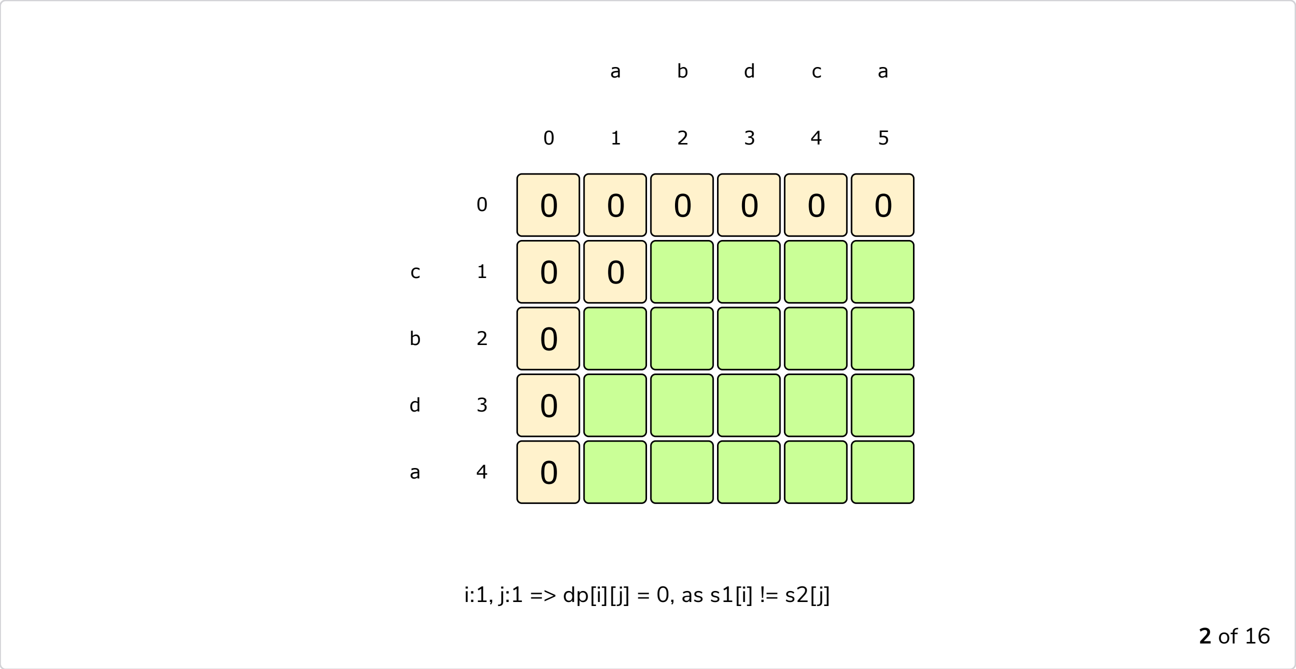
Let's draw this visually for "abcda" and "cbda". Starting with a substring of zero lengths, if any one of the string has zero length, then the common substring will be of zero length:



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0					
b	2	0					
d	3	0					
a	4	0					

i:0, j:0-5 and i:0-4, j:0 => dp[i][j] = 0, as we don't have any common substring when one of the string is of zero length

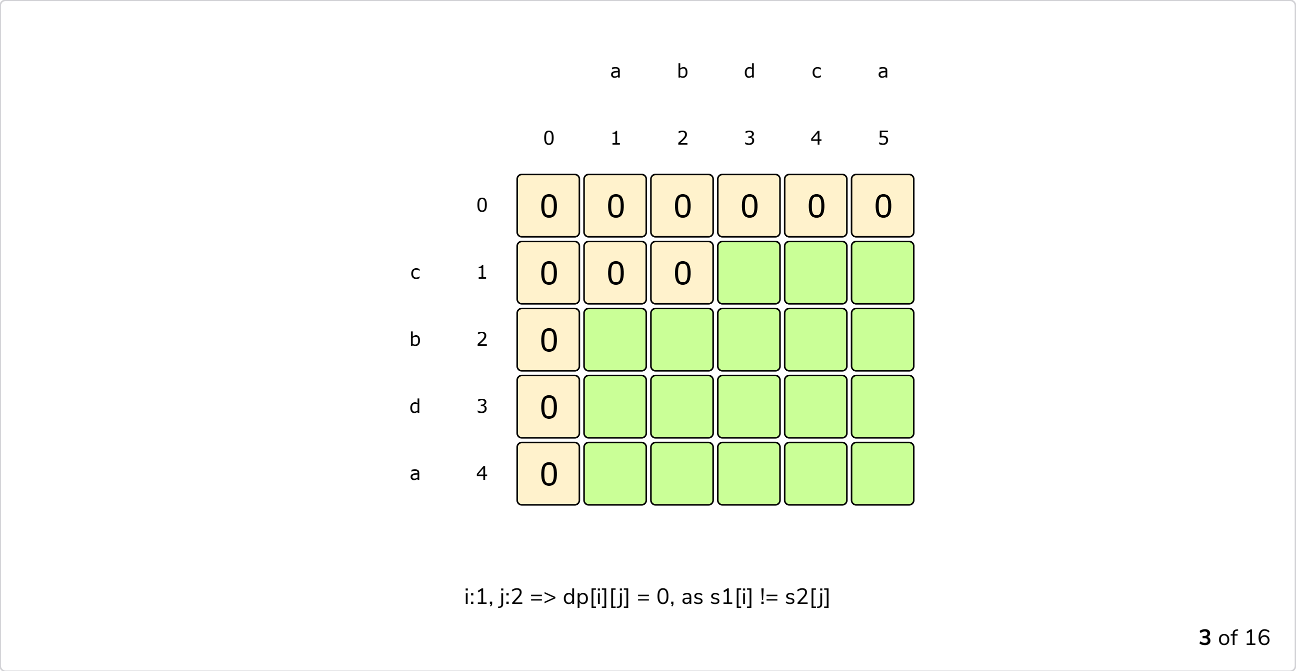
1 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0				
b	2	0					
d	3	0					
a	4	0					

i:1, j:1 => dp[i][j] = 0, as s1[i] != s2[j]

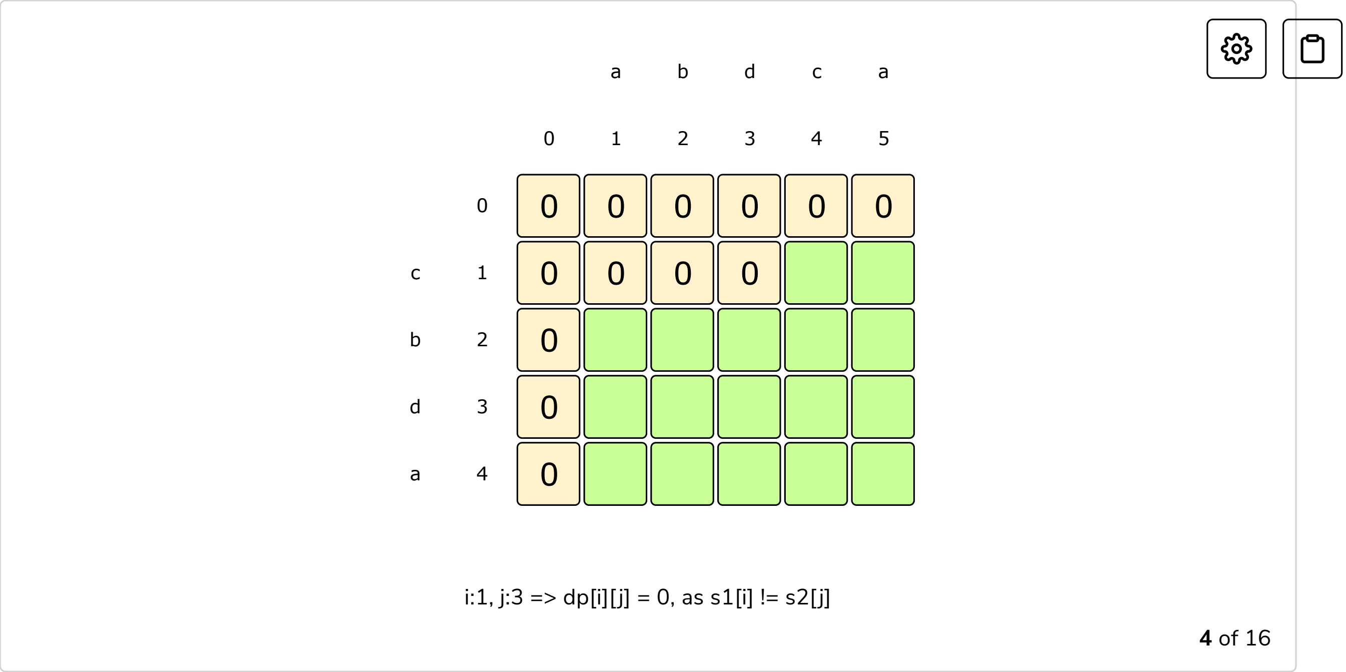
2 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0			
b	2	0					
d	3	0					
a	4	0					

i:1, j:2 => dp[i][j] = 0, as s1[i] != s2[j]

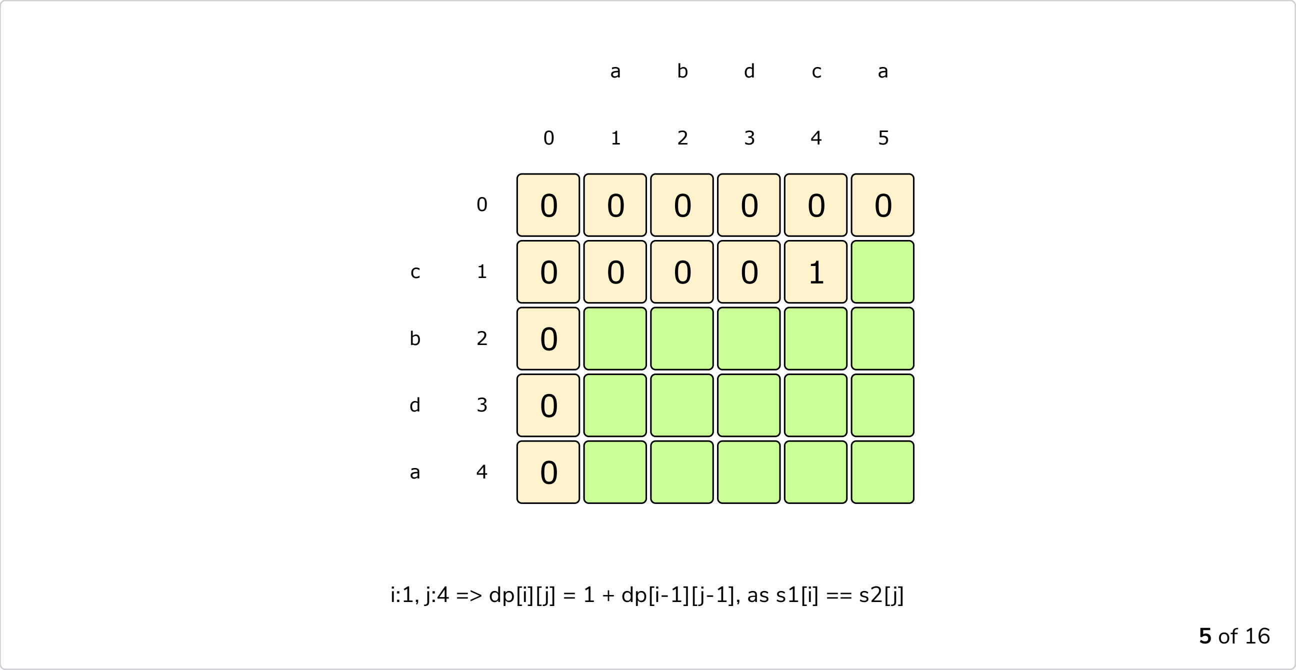
3 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0		
b	2	0					
d	3	0					
a	4	0					

i:1, j:3 => dp[i][j] = 0, as s1[i] != s2[j]

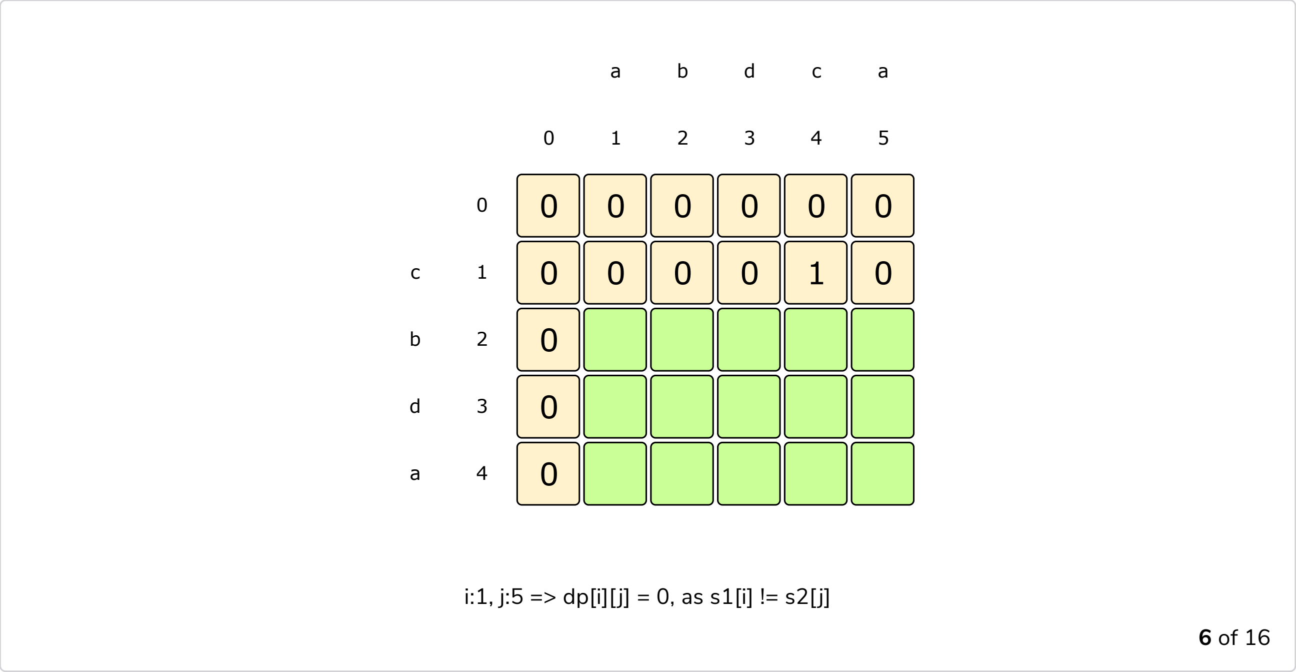
4 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	
b	2	0					
d	3	0					
a	4	0					

i:1, j:4 => dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

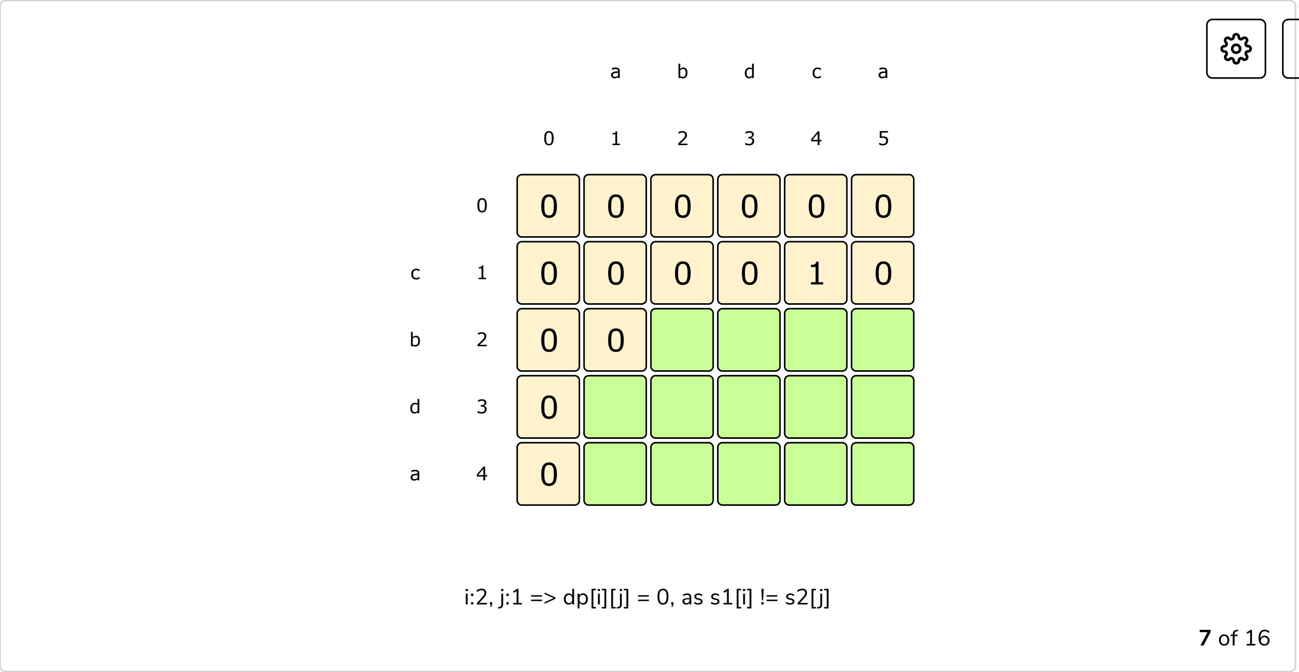
5 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0					
d	3	0					
a	4	0					

i:1, j:5 => dp[i][j] = 0, as s1[i] != s2[j]

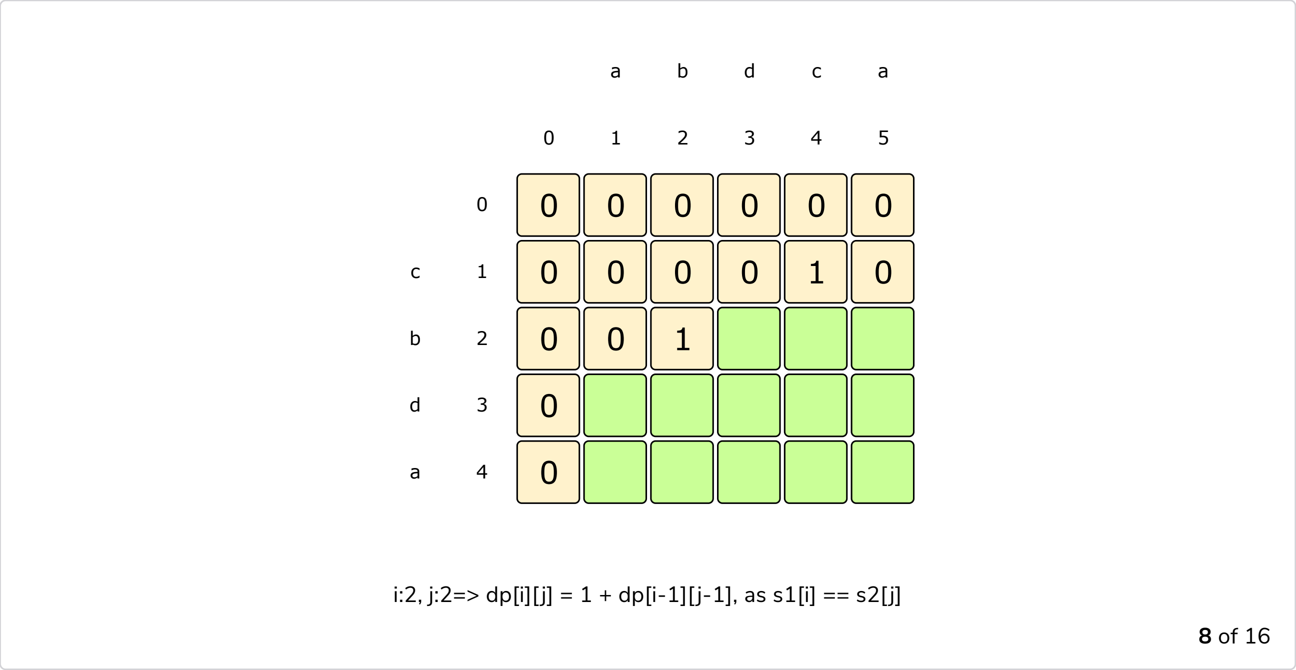
6 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	🟩	🟩	🟩	🟩
d	3	0	🟩	🟩	🟩	🟩	🟩
a	4	0	🟩	🟩	🟩	🟩	🟩

i:2, j:1 => dp[i][j] = 0, as s1[i] != s2[j]

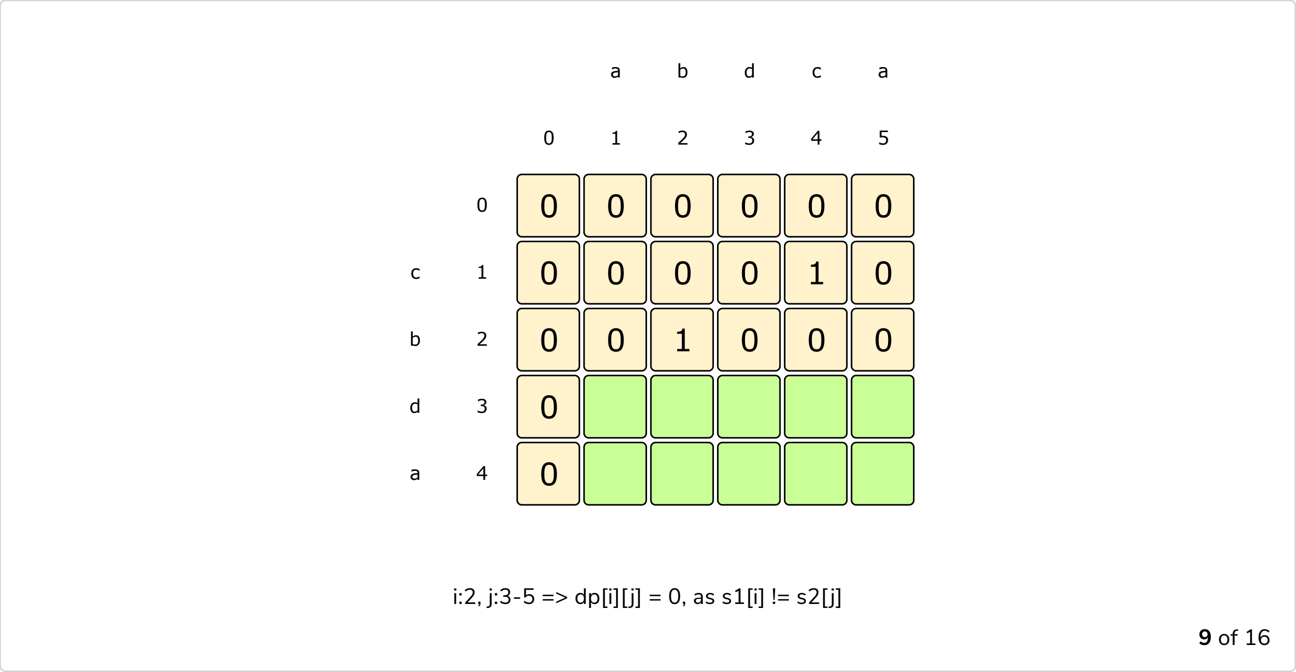
7 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	🟩	🟩	🟩
d	3	0	🟩	🟩	🟩	🟩	🟩
a	4	0	🟩	🟩	🟩	🟩	🟩

i:2, j:2=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

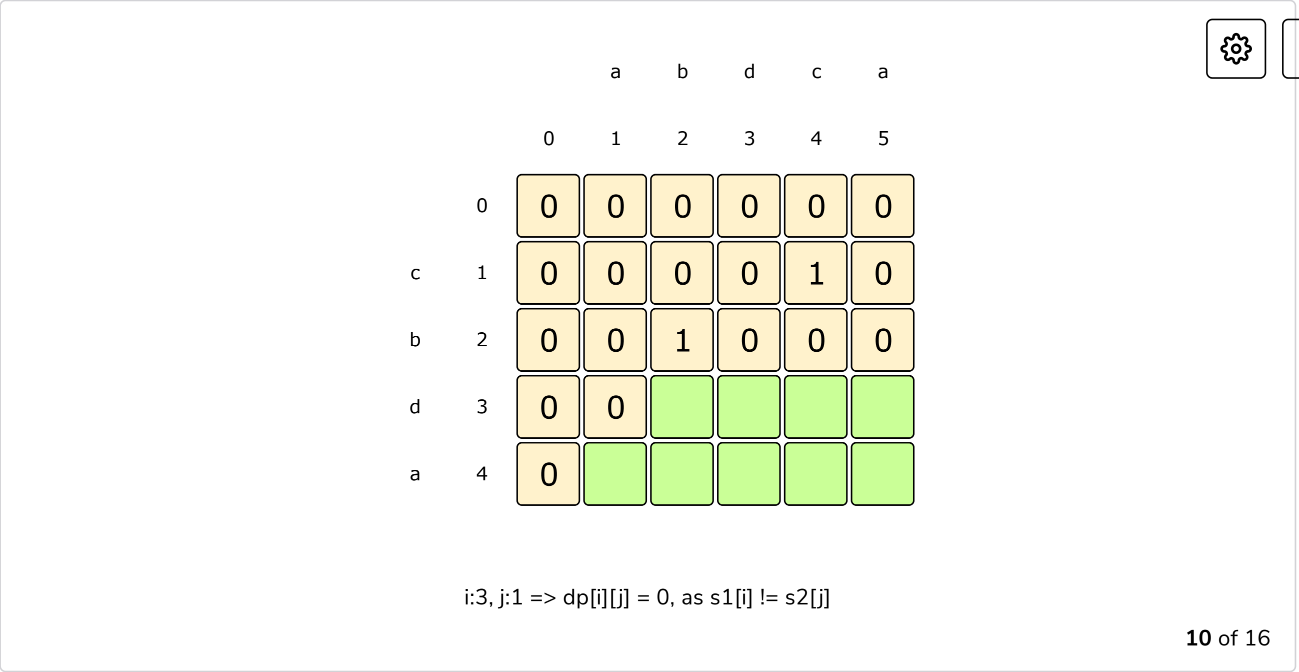
8 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	🟩	🟩	🟩	🟩	🟩
a	4	0	🟩	🟩	🟩	🟩	🟩

i:2, j:3-5 => dp[i][j] = 0, as s1[i] != s2[j]

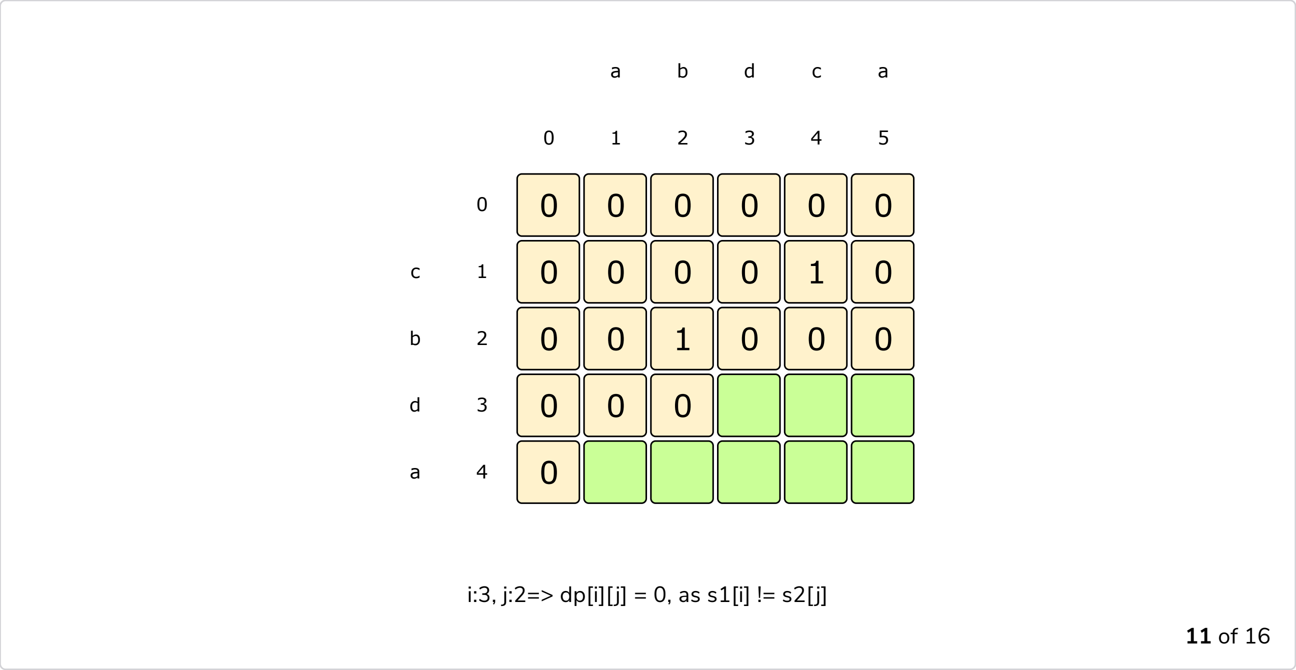
9 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	🟩	🟩	🟩	🟩
a	4	0	🟩	🟩	🟩	🟩	🟩

i:3, j:1 => dp[i][j] = 0, as s1[i] != s2[j]

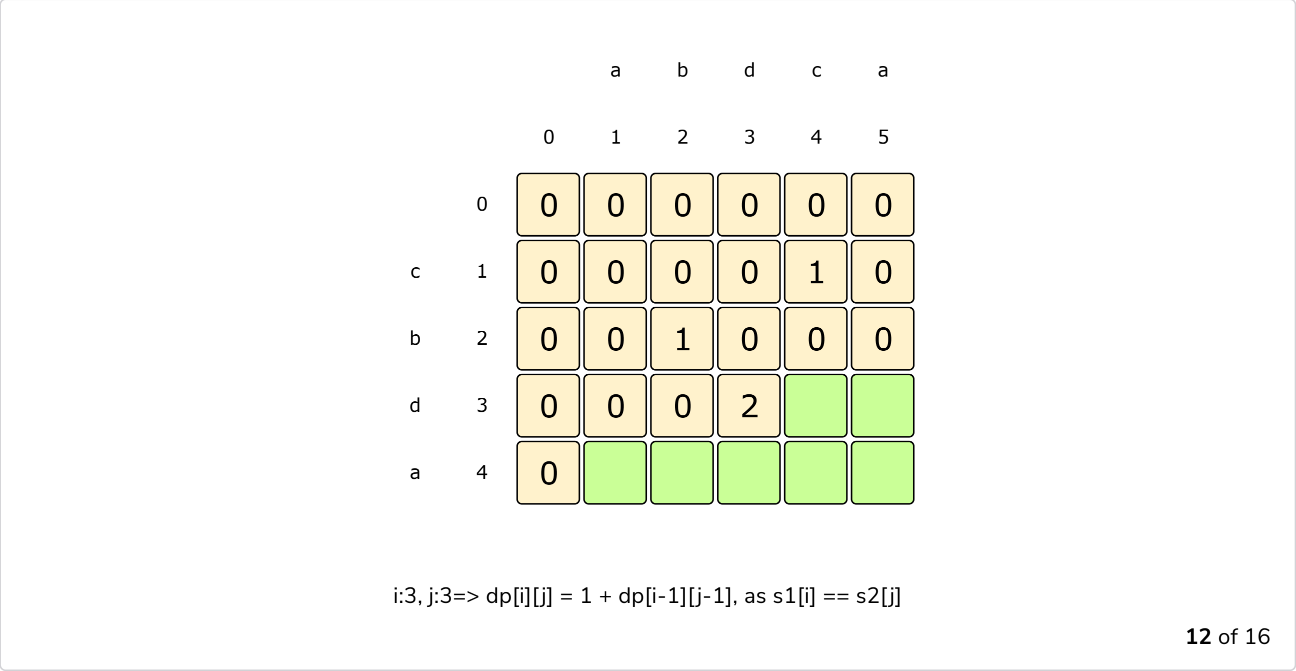
10 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	🟩	🟩	🟩
a	4	0	🟩	🟩	🟩	🟩	🟩

i:3, j:2=> dp[i][j] = 0, as s1[i] != s2[j]

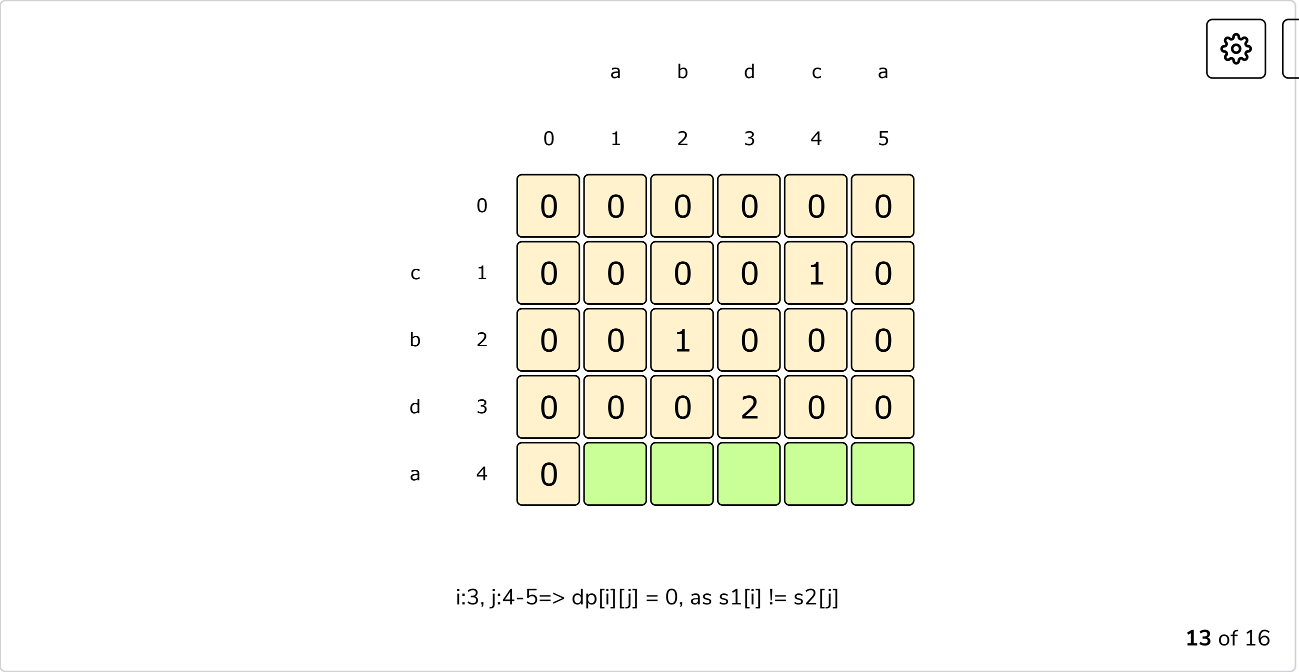
11 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	🟩	🟩
a	4	0	🟩	🟩	🟩	🟩	🟩

i:3, j:3=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

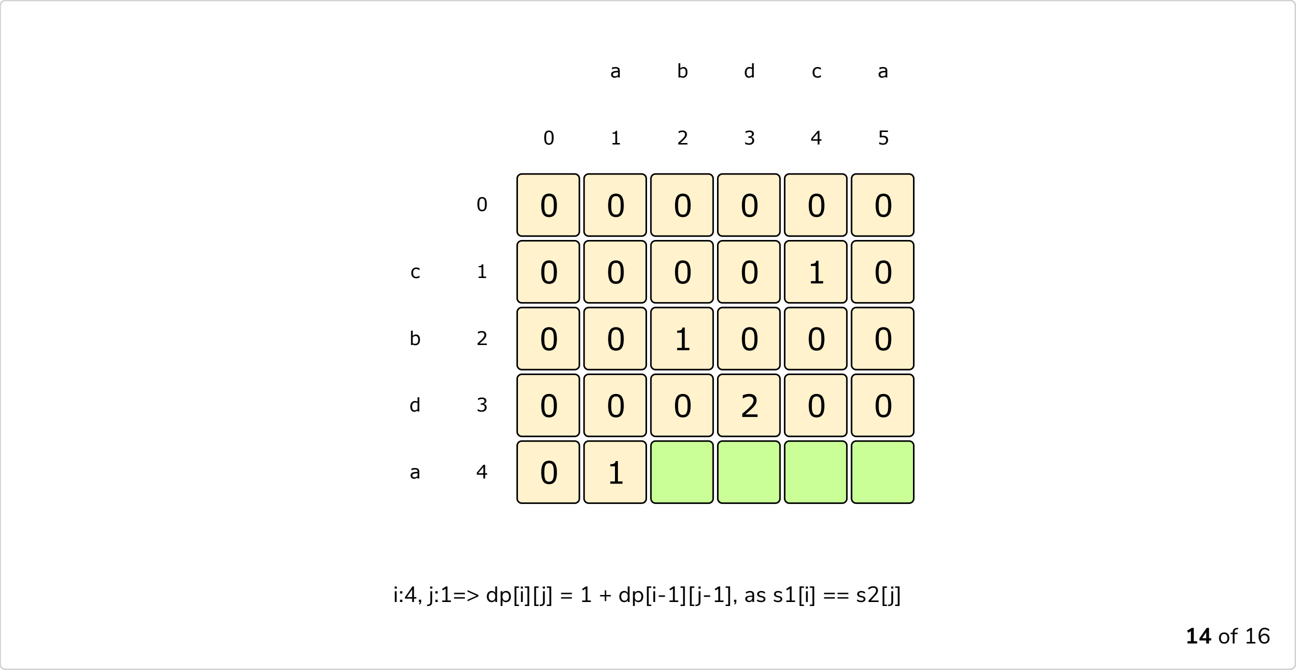
12 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	🟩	🟩	🟩	🟩	🟩

i:3, j:4-5=> dp[i][j] = 0, as s1[i] != s2[j]

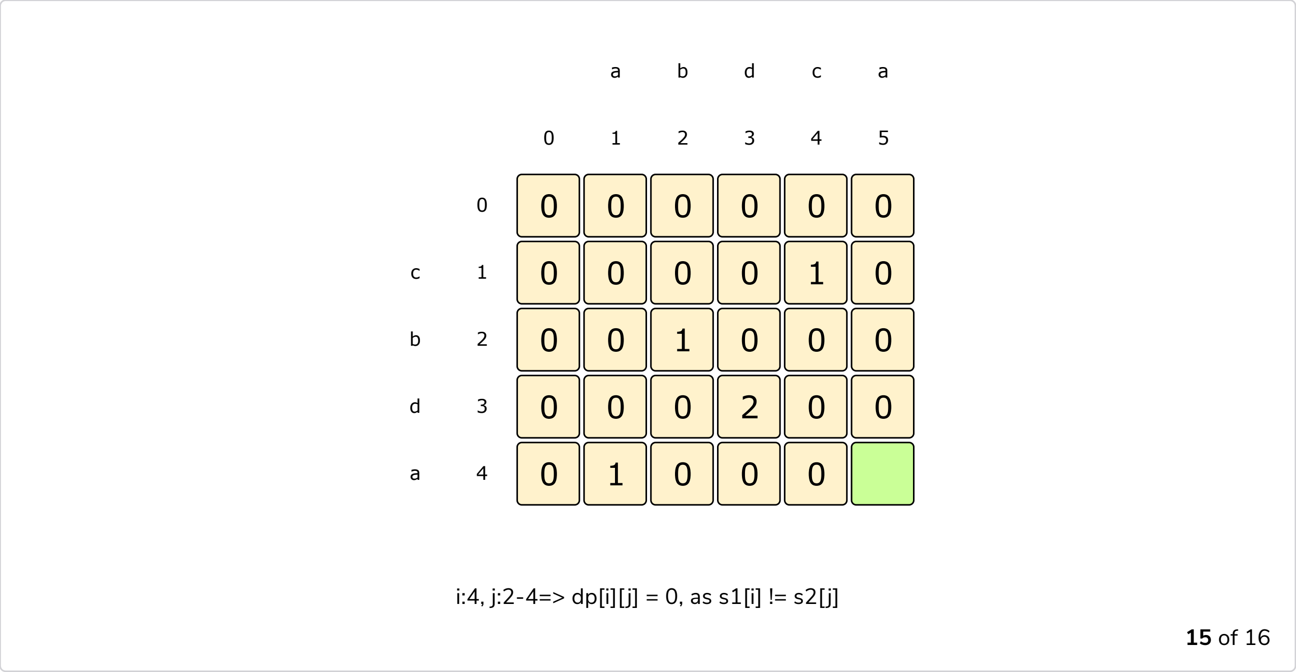
13 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	🟩	🟩	🟩	🟩

i:4, j:1=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

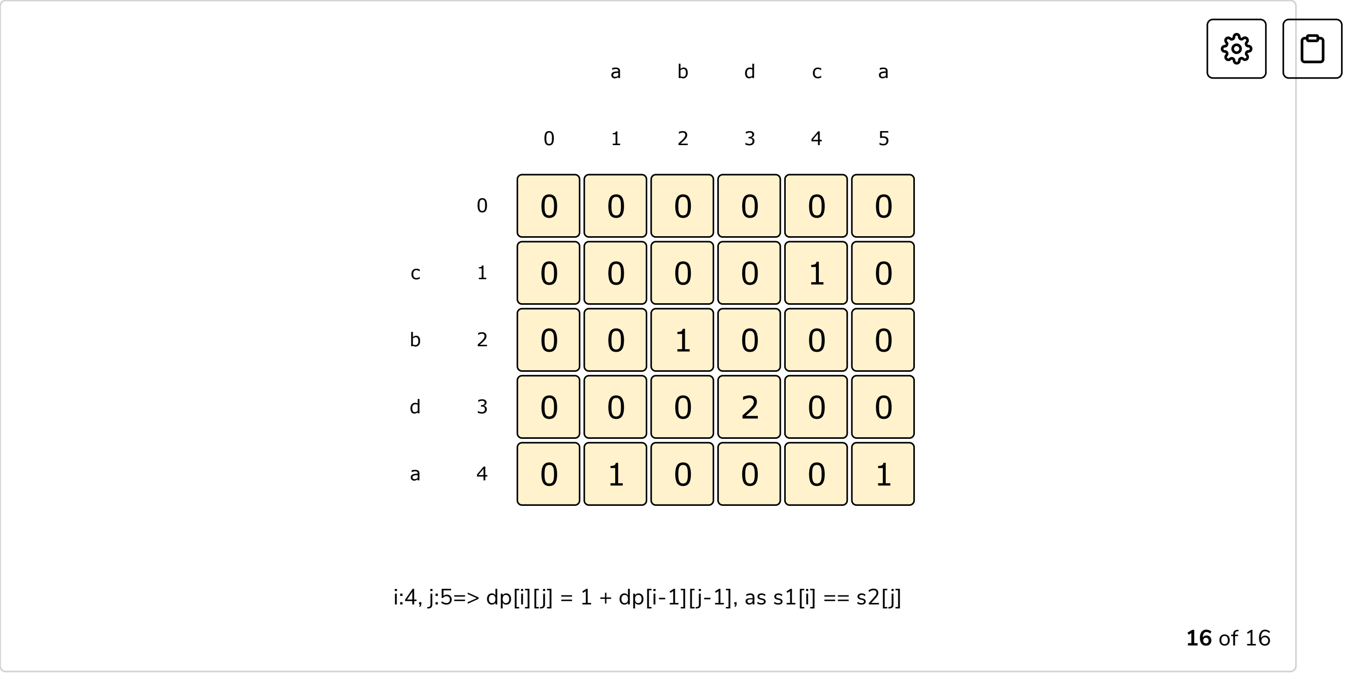
14 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	0	0	0	🟩

i:4, j:2-4=> dp[i][j] = 0, as s1[i] != s2[j]

15 of 16



			a	b	d	c	a
		0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	0	0	0	1

i:4, j:5=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

16 of 16

From the above visualization, we can clearly see that the longest common substring is of length '2'-- as shown by `dp[3][3]`. Here is the code for our bottom-up dynamic programming approach:

Java JS Python3 C++

```python
def find_LCS_length(s1, s2):
  n1, n2 = len(s1), len(s2)
  dp = [[0 for _ in range(n2+1)] for _ in range(n1+1)]
  maxLength = 0
  for i in range(1, n1+1):
    for j in range(1, n2+1):
      if s1[i - 1] == s2[j - 1]:
        dp[i][j] = 1 + dp[i - 1][j - 1]
        maxLength = max(maxLength, dp[i][j])
  return maxLength


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))


main()
```

The time and space complexity of the above algorithm is $O(m * n)$, where 'm' and 'n' are the lengths of the two input strings.

Challenge

Can we further improve our bottom-up DP solution? Can you find an algorithm that has $O(n)$ space complexity?

Hide Hint

We only need one previous row to find the optimal solution!

Java | JS | Python3 | C++

```
def find_LCS_length(s1, s2):
  # TODO: Write your code here
  return -1


```

Solution

```
def find_LCS_length(s1, s2):
  n1, n2 = len(s1), len(s2)
  dp = [[0 for _ in range(n2+1)] for _ in range(2)]
  maxLength = 0
  for i in range(1, n1+1):
    for j in range(1, n2+1):
      dp[i % 2][j] = 0
      if s1[i - 1] == s2[j - 1]:
        dp[i % 2][j] = 1 + dp[(i - 1) % 2][j - 1]
        maxLength = max(maxLength, dp[i % 2][j])

  return maxLength


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))


main()
```

← Back
Palindromic Partitioning

Next →
Longest Common Subsequence

☑ Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/longest-common-substring__pattern-5-longest-common-substring__grokking-dynamic-programming-patterns-for-coding-interviews)