**Longest Common Subsequence**

**WE'LL COVER THE FOLLOWING**

* + - [Problem Statement](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#problem-statement)
    - [Basic Solution](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#basic-solution)
    - [Top-down Dynamic Programming with Memoization](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#top-down-dynamic-programming-with-memoization)
    - [Bottom-up Dynamic Programming](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#bottom-up-dynamic-programming)
    - [Challenge](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#challenge)

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#problem-statement)

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

A [subsequence](https://en.wikipedia.org/wiki/Subsequence) is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

**Example 1:**

Input: s1 = "abdca"  
       s2 = "cbda"  
Output: 3  
Explanation: The longest common subsequence is "bda".

**Example 2:**

Input: s1 = "passport"  
       s2 = "ppsspt"  
Output: 5  
Explanation: The longest common subsequence is "psspt".

**Basic Solution**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#basic-solution)

A basic brute-force solution could be to try all subsequences of ‘s1’ and ‘s2’ to find the longest one. We can match both the strings one character at a time. So for every index ‘i’ in ‘s1’ and ‘j’ in ‘s2’ we must choose between:

1. If the character s1[i] matches s2[j], we can recursively match for the remaining lengths.
2. If the character s1[i] does not match s2[j], we will start two new recursive calls by skipping one character separately from each string.

Here is the code:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

class LCS {

  public int findLCSLength(String s1, String s2) {

      return findLCSLengthRecursive(s1, s2, 0, 0);

  }

  private int findLCSLengthRecursive(String s1, String s2, int i1, int i2) {

    if(i1 == s1.length() || i2 == s2.length())

      return 0;

    if(s1.charAt(i1) == s2.charAt(i2))

      return 1 + findLCSLengthRecursive(s1, s2, i1+1, i2+1);

    int c1 = findLCSLengthRecursive(s1, s2, i1, i2+1);

    int c2 = findLCSLengthRecursive(s1, s2, i1+1, i2);

    return Math.max(c1, c2);

  }

  public static void main(String[] args) {

    LCS lcs = new LCS();

    System.out.println(lcs.findLCSLength("abdca", "cbda"));

    System.out.println(lcs.findLCSLength("passport", "ppsspt"));

  }

}





RUN

SAVERESET

The time complexity of the above algorithm is exponential O(2^{m+n})*O*(2​*m*+*n*​​), where ‘m’ and ‘n’ are the lengths of the two input strings. The space complexity is O(n+m)*O*(*n*+*m*) which is used to store the recursion stack.

**Top-down Dynamic Programming with Memoization**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#top-down-dynamic-programming-with-memoization)

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

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

Here is the code:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

class LCS {

  public int findLCSLength(String s1, String s2) {

    Integer[][] dp = new Integer[s1.length()][s2.length()];

    return findLCSLengthRecursive(dp, s1, s2, 0, 0);

  }

  private int findLCSLengthRecursive(Integer[][] dp, String s1, String s2, int i1, int i2) {

    if (i1 == s1.length() || i2 == s2.length())

      return 0;

    if (dp[i1][i2] == null) {

      if (s1.charAt(i1) == s2.charAt(i2))

        dp[i1][i2] = 1 + findLCSLengthRecursive(dp, s1, s2, i1 + 1, i2 + 1);

      else {

        int c1 = findLCSLengthRecursive(dp, s1, s2, i1, i2 + 1);

        int c2 = findLCSLengthRecursive(dp, s1, s2, i1 + 1, i2);

        dp[i1][i2] = Math.max(c1, c2);

      }

    }

    return dp[i1][i2];

  }

  public static void main(String[] args) {

    LCS lcs = new LCS();

    System.out.println(lcs.findLCSLength("abdca", "cbda"));

    System.out.println(lcs.findLCSLength("passport", "ppsspt"));

  }

}





RUN

SAVERESET

**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#bottom-up-dynamic-programming)

Since we want to match all the subsequences 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 array’s two dimensions. So for every index ‘i’ in string ‘s1’ and ‘j’ in string ‘s2’, we will choose one of the following two options:

1. If the character s1[i] matches s2[j], the length of the common subsequence would be one plus the length of the common subsequence till the i-1 and j-1 indexes in the two respective strings.
2. If the character s1[i] does not match s2[j], we will take the longest subsequence by either skipping ith or jth character from the respective strings.

So our recursive formula would be:

1

2

3

4

if s1[i] == s2[j]

  dp[i][j] = 1 + dp[i-1][j-1]

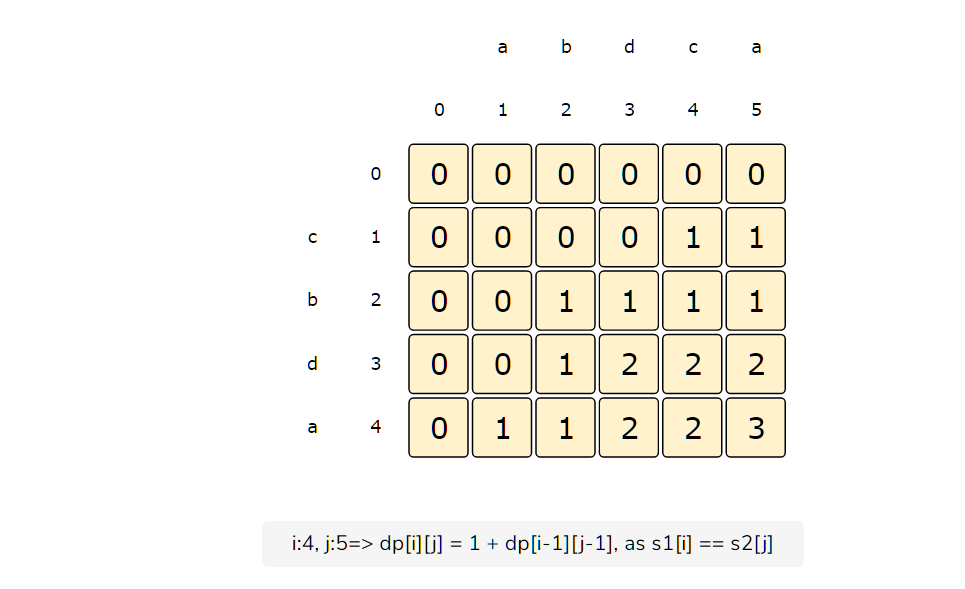
else

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





Let’s draw this visually for “abcda” and “cbda”. Starting with a subsequence of zero lengths, if any string has zero length then the common subsequence will be of zero length:



From the above visualization, we can clearly see that the longest common subsequence is of length ‘3’ – as shown by dp[4][5].

Here is the code for our bottom-up dynamic programming approach:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

class LCS {

  public int findLCSLength(String s1, String s2) {

    int[][] dp = new int[s1.length()+1][s2.length()+1];

    int maxLength = 0;

    for(int i=1; i <= s1.length(); i++) {

      for(int j=1; j <= s2.length(); j++) {

        if(s1.charAt(i-1) == s2.charAt(j-1))

          dp[i][j] = 1 + dp[i-1][j-1];

        else

          dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);

        maxLength = Math.max(maxLength, dp[i][j]);

      }

    }

    return maxLength;

  }

  public static void main(String[] args) {

    LCS lcs = new LCS();

    System.out.println(lcs.findLCSLength("abdca", "cbda"));

    System.out.println(lcs.findLCSLength("passport", "ppsspt"));

  }

}





RUN

SAVERESET

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

**Challenge**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N#challenge)

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

Show Hint

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8Pq4ZnBN0N)

1

2

3

4

5

6

class LCS {

  static int findLCSLength(String s1, String s2) {

    //TODO: Write - Your - Code

    return -1;

  }

}





TESTHIDE SOLUTION

SAVERESET

Solution

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

class LCS {

  static int findLCSLength(String s1, String s2) {

    int[][] dp = new int[2][s2.length()+1];

    int maxLength = 0;

    for(int i=1; i <= s1.length(); i++) {

      for(int j=1; j <= s2.length(); j++) {

        if(s1.charAt(i-1) == s2.charAt(j-1))

          dp[i%2][j] = 1 + dp[(i-1)%2][j-1];

        else

          dp[i%2][j] = Math.max(dp[(i-1)%2][j], dp[i%2][j-1]);

        maxLength = Math.max(maxLength, dp[i%2][j]);

      }

    }

    return maxLength;

  }

}





**MARK AS COMPLETED**

[**←    Back**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMkk7NwE44R)

Longest Common Substring

[**Next    →**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m27OkE8D08O)

Minimum Deletions & Insertions to Transform a String into another

Stuck? Get help on

[**DISCUSS**](https://discuss.educative.io/c/grokking-dynamic-programming-patterns-for-coding-interviews-design-gurus/pattern-5-longest-common-substring-longest-common-subsequence)

Send feedback

10 Recommendations

Cannot edit in read-only editor (occurred 6 times)