**Longest Repeated (Duplicate) SubSeq**

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7npz2VooPl1#problem-statement)

Given a sequence, find the length of its longest repeating subsequence (LRS). A repeating subsequence will be the one that appears at least twice in the original sequence and is not overlapping (i.e. none of the corresponding characters in the repeating subsequences have the same index).

**Example 1:**

Input: “t o m o r r o w”  
Output: 2  
Explanation: The longest repeating subsequence is “or” {tomorrow}.

**Example 2:**

Input: “a a b d b c e c”  
Output: 3  
Explanation: The longest repeating subsequence is “a b c” {a a b d b c e c}.

### Basic Solution [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7npz2VooPl1#basic-solution)

The problem is quite similar to the [Longest Common Subsequence](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5657535201673216) (LCS), with two differences:

1. In LCS, we were trying to find the longest common subsequence between the two strings, whereas in LRS we are trying to find the two longest common subsequences within one string.
2. class LRS {
3. public int findLRSLength(String str) {
4. return findLRSLengthRecursive(str, 0, 0);
5. }
6. private int findLRSLengthRecursive(String str, int i1, int i2) {
7. if(i1 == str.length() || i2 == str.length())
8. return 0;
9. if(i1 != i2 && str.charAt(i1) == str.charAt(i2))
10. return 1 + findLRSLengthRecursive(str, i1+1, i2+1);
11. int c1 = findLRSLengthRecursive(str, i1, i2+1);
12. int c2 = findLRSLengthRecursive(str, i1+1, i2);
13. return Math.max(c1, c2);
14. }
15. public static void main(String[] args) {
16. LRS lrs = new LRS();
17. System.out.println(lrs.findLRSLength("tomorrow"));
18. System.out.println(lrs.findLRSLength("aabdbcec"));
19. System.out.println(lrs.findLRSLength("fmff"));
20. }
21. }
22. The time complexity of the above algorithm is exponential O(2^n)*O*(2​*n*​​), where ‘n’ is the length of the input sequence. The space complexity is O(n)*O*(*n*) 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/7npz2VooPl1#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)).

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7npz2VooPl1#code-2)

Here is the code:

class LRS {

  public int findLRSLength(String str) {

    Integer[][] dp = new Integer[str.length()][str.length()];

    return findLRSLengthRecursive(dp, str, 0, 0);

  }

  private int findLRSLengthRecursive(Integer[][] dp, String str, int i1, int i2) {

    if(i1 == str.length() || i2 == str.length())

      return 0;

    if(dp[i1][i2] == null) {

      if(i1 != i2 && str.charAt(i1) == str.charAt(i2))

        dp[i1][i2] = 1 + findLRSLengthRecursive(dp, str, i1+1, i2+1);

      else {

        int c1 = findLRSLengthRecursive(dp, str, i1, i2+1);

        int c2 = findLRSLengthRecursive(dp, str, i1+1, i2);

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

      }

    }

    return dp[i1][i2];

  }

  public static void main(String[] args) {

    LRS lrs = new LRS();

    System.out.println(lrs.findLRSLength("tomorrow"));

    System.out.println(lrs.findLRSLength("aabdbcec"));

    System.out.println(lrs.findLRSLength("fmff"));

  }

}

### Bottom-up Dynamic Programming [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7npz2VooPl1#bottom-up-dynamic-programming)

Since we want to match all the subsequences of the given string, we can use a two-dimensional array to store our results. As mentioned above, we will be tracking two indexes to overcome the overlapping problem. So for each of the two indexes, ‘i1’ and ‘i2’, we will choose one of the following options:

1. If ‘i1’ and ‘i2’ are different and the character str[i1] matches the character str[i2], then the length of the LRS would be one plus the length of LRS up to i1-1 and i2-1 indexes.
2. If the character at str[i1] does not match str[i2], we will take the LRS by either skipping 'i1’th or 'i2’th character.

So our recursive formula would be:

1

2

3

4

if i1 != i2 && str[i1] == str[i2]

  dp[i1][i2] = 1 + dp[i1-1][i2-1]

else

  dp[i1][i2] = max(dp[i1-1][i2], dp[i1][i2-1])





#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7npz2VooPl1#code-3)

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

class LRS {

  public int findLRSLength(String str) {

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

    int maxLength = 0;

    // dp[i1][i2] will be storing the LRS up to str[0..i1-1][0..i2-1]

    // this also means that subsequences of length zero (first row and column of dp[][]),

    // will always have LRS of size zero.

    for(int i1=1; i1 <= str.length(); i1++) {

      for(int i2=1; i2 <= str.length(); i2++) {

        if(i1 != i2 && str.charAt(i1-1) == str.charAt(i2-1))

          dp[i1][i2] = 1 + dp[i1-1][i2-1];

        else

          dp[i1][i2] = Math.max(dp[i1-1][i2], dp[i1][i2-1]);

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

      }

    }

    return maxLength;

  }

  public static void main(String[] args) {

    LRS lrs = new LRS();

    System.out.println(lrs.findLRSLength("tomorrow"));

    System.out.println(lrs.findLRSLength("aabdbcec"));

    System.out.println(lrs.findLRSLength("fmff"));

  }

}

The time and space complexity of the above algorithm is O(n^2), where ‘n’ is the length of the input sequence.