**Dynamic programming**

# Longest Common Subsequence

### 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.

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"));

  }

}

The time complexity of the above algorithm is exponential O(2^{m+n}), where ‘m’ and ‘n’ are the lengths of the two input strings. The space complexity is 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:

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"));

  }

}

### 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:

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:

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"));

  }

}

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.

# Minimum Deletions & Insertions to Transform a String into another

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m27OkE8D08O#problem-statement)

Given strings s1 and s2, we need to transform s1 into s2 by deleting and inserting characters. Write a function to calculate the count of the minimum number of deletion and insertion operations.

**Example 1:**

Input: s1 = "abc"  
       s2 = "fbc"  
Output: 1 deletion and 1 insertion.  
Explanation: We need to delete {'a'} and insert {'f'} to s1 to transform it into s2.

**Example 2:**

Input: s1 = "abdca"  
       s2 = "cbda"  
Output: 2 deletions and 1 insertion.  
Explanation: We need to delete {'a', 'c'} and insert {'c'} to s1 to transform it into s2.

**Example 3:**

Input: s1 = "passport"  
       s2 = "ppsspt"  
Output: 3 deletions and 1 insertion  
Explanation: We need to delete {'a', 'o', 'r'} and insert {'p'} to s1 to transform it into s2.

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

This problem can easily be converted to the Longest Common Subsequence (LCS). If we can find the LCS of the two input strings, we can easily find how many characters we need to insert and delete from s1. Here is how we can do this:

1. Let’s assume len1 is the length of s1 and len2 is the length of s2.
2. Now let’s assume c1 is the length of LCS of the two strings s1 and s2.
3. To transform s1 into s2, we need to delete everything from s1 which is not part of LCS, so minimum deletions we need to perform from s1 => len1 - c1
4. Similarly, we need to insert everything in s1 which is present in s2 but not part of LCS, so minimum insertions we need to perform in s1 => len2 - c1

Let’s jump directly to the bottom-up dynamic programming solution:

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m27OkE8D08O#bottom-up-dynamic-programming)

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

class MDI {

  public void findMDI(String s1, String s2) {

    int c1 = findLCSLength(s1, s2);

    System.out.println("Minimum deletions needed: " + (s1.length() - c1));

    System.out.println("Minimum insertions needed: " + (s2.length() - c1));

  }

  private 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) {

    MDI mdi = new MDI();

    mdi.findMDI("abc", "fbc");

    mdi.findMDI("abdca", "cbda");

    mdi.findMDI("passport", "ppsspt");

  }

}

# Longest Increasing Subsequence

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQ0y0QOJQ69#problem-statement)

Given a number sequence, find the length of its Longest Increasing Subsequence (LIS). In an increasing subsequence, all the elements are in increasing order (from lowest to highest).

**Example 1:**

Input: {4,2,3,6,10,1,12}  
Output: 5  
Explanation: The LIS is {2,3,6,10,12}.

**Example 1:**

Input: {-4,10,3,7,15}  
Output: 4  
Explanation: The LIS is {-4,3,7,15}.

### Basic Solution [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQ0y0QOJQ69#basic-solution)

A basic brute-force solution could be to try all the subsequences of the given number sequence. We can process one number at a time, so we have two options at any step:

1. If the current number is greater than the previous number that we included, we can increment our count and make a recursive call for the remaining array.
2. We can skip the current number to make a recursive call for the remaining array.

The length of the longest increasing subsequence will be the maximum number returned by the two recurse calls from the above two options.

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQ0y0QOJQ69#code)

class LIS {

  public int findLISLength(int[] nums) {

      return findLISLengthRecursive(nums, 0, -1);

  }

  private int findLISLengthRecursive(int[] nums, int currentIndex, int previousIndex) {

    if(currentIndex == nums.length)

      return 0;

    // include nums[currentIndex] if it is larger than the last included number

    int c1 = 0;

    if(previousIndex == -1 || nums[currentIndex] > nums[previousIndex])

      c1 = 1 + findLISLengthRecursive(nums, currentIndex+1, currentIndex);

    // excluding the number at currentIndex

    int c2 = findLISLengthRecursive(nums, currentIndex+1, previousIndex);

    return Math.max(c1, c2);

  }

  public static void main(String[] args) {

    LIS lis = new LIS();

    int[] nums = {4,2,3,6,10,1,12};

    System.out.println(lis.findLISLength(nums));

    nums = new int[]{-4,10,3,7,15};

    System.out.println(lis.findLISLength(nums));

  }

}

The time complexity of the above algorithm is exponential O(2^n)*O*(2​*n*​​), where ‘n’ is the lengths of the input array. 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/YQ0y0QOJQ69#top-down-dynamic-programming-with-memoization)

To overcome the overlapping subproblems, we can use an array to store the already solved subproblems.

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

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQ0y0QOJQ69#code-2)

class LIS {

  public int findLISLength(int[] nums) {

    Integer[][] dp = new Integer[nums.length][nums.length+1];

    return findLISLengthRecursive(dp, nums, 0, -1);

  }

  private int findLISLengthRecursive(Integer[][] dp, int[] nums, int currentIndex, int previousIndex) {

    if(currentIndex == nums.length)

      return 0;

    if(dp[currentIndex][previousIndex+1] == null) {

      // include nums[currentIndex] if it is larger than the last included number

      int c1 = 0;

      if(previousIndex == -1 || nums[currentIndex] > nums[previousIndex])

        c1 = 1 + findLISLengthRecursive(dp, nums, currentIndex+1, currentIndex);

      int c2 = findLISLengthRecursive(dp, nums, currentIndex+1, previousIndex);

      dp[currentIndex][previousIndex+1] = Math.max(c1, c2);

    }

    return dp[currentIndex][previousIndex+1];

  }

  public static void main(String[] args) {

    LIS lis = new LIS();

    int[] nums = {4,2,3,6,10,1,12};

    System.out.println(lis.findLISLength(nums));

    nums = new int[]{-4,10,3,7,15};

    System.out.println(lis.findLISLength(nums));

  }

}

**What is the time and space complexity of the above solution?** Since our memoization array dp[nums.length()][nums.length()] stores the results for all the subproblems, we can conclude that we will not have more than N\*N subproblems (where ‘N’ is the length of the input sequence). This means that our time complexity will be O(N^2).

The above algorithm will be using O(N^2) space for the memoization array. Other than that we will use O(N) space for the recursion call-stack. So the total space complexity will be O(N^2 + N), which is asymptotically equivalent to O(N^2).

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQ0y0QOJQ69#bottom-up-dynamic-programming)

The above algorithm tells us two things:

1. If the number at the current index is bigger than the number at the previous index, we increment the count for LIS up to the current index.
2. But if there is a bigger LIS without including the number at the current index, we take that.

So we need to find all the increasing subsequences for the number at index ‘i’, from all the previous numbers (i.e. number till index ‘i-1’), to eventually find the longest increasing subsequence.

If ‘i’ represents the ‘currentIndex’ and ‘j’ represents the ‘previousIndex’, our recursive formula would look like:

    if num[i] > num[j] => dp[i] = dp[j] + 1 if there is no bigger LIS for 'i'

#### Code

class LIS {

  public int findLISLength(int[] nums) {

    int[] dp = new int[nums.length];

    dp[0] = 1;

    int maxLength = 1;

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

      dp[i] = 1;

      for (int j=0; j<i; j++)

        if (nums[i] > nums[j] && dp[i] <= dp[j] ) {

          dp[i] = dp[j]+1;

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

        }

    }

    return maxLength;

  }

  public static void main(String[] args) {

    LIS lis = new LIS();

    int[] nums = {4,2,3,6,10,1,12};

    System.out.println(lis.findLISLength(nums));

    nums = new int[]{-4,10,3,7,15};

    System.out.println(lis.findLISLength(nums));

  }

}

# Maximum Sum Increasing Subsequence

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8rgqKEW05N#problem-statement)

Given a number sequence, find the increasing subsequence with the highest sum. Write a method that returns the highest sum.

**Example 1:**

Input: {4,1,2,6,10,1,12}  
Output: 32  
Explanation: The increaseing sequence is {4,6,10,12}.   
Please note the difference, as the LIS is {1,2,6,10,12} which has a sum of '31'.

**Example 2:**

Input: {-4,10,3,7,15}  
Output: 25  
Explanation: The increaseing sequences are {10, 15} and {3,7,15}.

### Basic Solution [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8rgqKEW05N#basic-solution)

The problem is quite similar to the Longest Incresing subsequence. The only difference is that, instead of finding the increasing subsequence with the maximum length, we need to find an increasing sequence with the maximum sum.

A basic brute-force solution could be to try all the subsequences of the given array. We can process one number at a time, so we have two options at any step:

1. If the current number is greater than the previous number that we included, we include that number in a running sum and make a recursive call for the remaining array.
2. We can skip the current number to make a recursive call for the remaining array.

The highest sum of any increasing subsequence would be the max value returned by the two recurse calls from the above two options.

Here is the code:

class MSIS {

  public int findMSIS(int[] nums) {

      return findMSISRecursive(nums, 0, -1, 0);

  }

  private int findMSISRecursive(int[] nums, int currentIndex, int previousIndex, int sum) {

    if(currentIndex == nums.length)

      return sum;

    // include nums[currentIndex] if it is larger than the last included number

    int s1 = sum;

    if(previousIndex == -1 || nums[currentIndex] > nums[previousIndex])

      s1 = findMSISRecursive(nums, currentIndex+1, currentIndex, sum + nums[currentIndex]);

    // excluding the number at currentIndex

    int s2 = findMSISRecursive(nums, currentIndex+1, previousIndex, sum);

    return Math.max(s1, s2);

  }

  public static void main(String[] args) {

    MSIS msis = new MSIS();

    int[] nums = {4,1,2,6,10,1,12};

    System.out.println(msis.findMSIS(nums));

    nums = new int[]{-4,10,3,7,15};

    System.out.println(msis.findMSIS(nums));

  }

}

The time complexity of the above algorithm is exponential O(2^n), where ‘n’ is the lengths of the input array. The space complexity is 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/B8rgqKEW05N#top-down-dynamic-programming-with-memoization)

We can use memoization to overcome the overlapping subproblems.

The three changing values for our recursive function are the current index, the previous index, and the sum. An efficient way of storing the results of the subproblems could be a hash-table whose key would be a string (currentIndex + “|” + previousIndex + “|” + sum).

Here is the code:

import java.util.HashMap;

import java.util.Map;

class MSIS {

  public int findMSIS(int[] nums) {

    Map<String, Integer> dp = new HashMap<>();

    return findMSISRecursive(dp, nums, 0, -1, 0);

  }

  private int findMSISRecursive(Map<String, Integer> dp, int[] nums, int currentIndex, int previousIndex, int sum) {

    if(currentIndex == nums.length)

      return sum;

    String subProblemKey = currentIndex + "-" + previousIndex + "-" + sum;

    if(!dp.containsKey(subProblemKey)) {

      // include nums[currentIndex] if it is larger than the last included number

      int s1 = sum;

      if(previousIndex == -1 || nums[currentIndex] > nums[previousIndex])

        s1 = findMSISRecursive(dp, nums, currentIndex+1, currentIndex, sum + nums[currentIndex]);

      // excluding the number at currentIndex

      int s2 = findMSISRecursive(dp, nums, currentIndex+1, previousIndex, sum);

      dp.put(subProblemKey, Math.max(s1, s2));

    }

    return dp.get(subProblemKey);

  }

  public static void main(String[] args) {

    MSIS msis = new MSIS();

    int[] nums = {4,1,2,6,10,1,12};

    System.out.println(msis.findMSIS(nums));

    nums = new int[]{-4,10,3,7,15};

    System.out.println(msis.findMSIS(nums));

    nums = new int[]{1,3,8,4,14,6,14,1,9,4,13,3,11,17,29};

    System.out.println(msis.findMSIS(nums));

  }

}

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/B8rgqKEW05N#bottom-up-dynamic-programming)

The above algorithm tells us two things:

1. If the number at the current index is bigger than the number at the previous index, we include that number in the sum for an increasing sequence up to the current index.
2. But if there is a maximum sum increasing subsequence (MSIS), without including the number at the current index, we take that.

So we need to find all the increasing subsequences for a number at index i, from all the previous numbers (i.e. numbers till index i-1), to find MSIS.

If i represents the currentIndex and ‘j’ represents the previousIndex, our recursive formula would look like:

    if num[i] > num[j] => dp[i] = dp[j] + num[i] if there is no bigger MSIS for 'i'

class MSIS {

  public int findMSIS(int[] nums) {

    int[] dp = new int[nums.length];

    dp[0] = nums[0];

    int maxSum = nums[0];

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

      dp[i] = nums[i];

      for (int j=0; j<i; j++) {

        if (nums[i] > nums[j] && dp[i] < dp[j] + nums[i])

          dp[i] = dp[j] + nums[i];

      }

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

    }

    return maxSum;

  }

  public static void main(String[] args) {

    MSIS msis = new MSIS();

    int[] nums = {4,1,2,6,10,1,12};

    System.out.println(msis.findMSIS(nums));

    nums = new int[]{-4,10,3,7,15};

    System.out.println(msis.findMSIS(nums));

    nums = new int[]{1,3,8,4,14,6,14,1,9,4,13,3,11,17,29};

    System.out.println(msis.findMSIS(nums));

  }

}

# Shortest Common Super-sequence

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jjLPyLGnAR#problem-statement)

Given two sequences ‘s1’ and ‘s2’, write a method to find the length of the shortest sequence which has ‘s1’ and ‘s2’ as subsequences.

**Example 2:**

Input: s1: "abcf" s2:"bdcf"   
Output: 5  
Explanation: The shortest common super-sequence (SCS) is "abdcf".

**Example 2:**

Input: s1: "dynamic" s2:"programming"   
Output: 15  
Explanation: The SCS is "dynprogrammicng".

### Basic Solution [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jjLPyLGnAR#basic-solution)

The problem is quite similar to the Longest Common Subsequence.

A basic brute-force solution could be to try all the super-sequences of the given sequences. We can process both of the sequences one character at a time, so at any step we must choose between:

1. If the sequences have a matching character, we can skip one character from both the sequences and make a recursive call for the remaining lengths to get SCS.
2. If the strings don’t match, we start two new recursive calls by skipping one character separately from each string. The minimum of these two recursive calls will have our answer.

Here is the code:

class SCS {

  public int findSCSLength(String s1, String s2) {

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

  }

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

    // if we have reached the end of a string, return the remaining length of the other string,

    // as in this case we have to take all of the remaining other string

    if(i1 == s1.length())

      return s2.length()-i2;

    if(i2 == s2.length())

      return s1.length()-i1;

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

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

    int length1 = 1 + findSCSLengthRecursive(s1, s2, i1, i2+1);

    int length2 = 1 + findSCSLengthRecursive(s1, s2, i1+1, i2);

    return Math.min(length1, length2);

  }

  public static void main(String[] args) {

    SCS scs = new SCS();

    System.out.println(scs.findSCSLength("abcf", "bdcf"));

    System.out.println(scs.findSCSLength("dynamic", "programming"));

  }

}

class SCS {

  public int findSCSLength(String s1, String s2) {

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

  }

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

    // if we have reached the end of a string, return the remaining length of the other string,

    // as in this case we have to take all of the remaining other string

    if(i1 == s1.length())

      return s2.length()-i2;

    if(i2 == s2.length())

      return s1.length()-i1;

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

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

    int length1 = 1 + findSCSLengthRecursive(s1, s2, i1, i2+1);

    int length2 = 1 + findSCSLengthRecursive(s1, s2, i1+1, i2);

    return Math.min(length1, length2);

  }

  public static void main(String[] args) {

    SCS scs = new SCS();

    System.out.println(scs.findSCSLength("abcf", "bdcf"));

    System.out.println(scs.findSCSLength("dynamic", "programming"));

  }

}

The time complexity of the above algorithm is exponential O(2^{n+m}), where ‘n’ and ‘m’ are the lengths of the input sequences. The space complexity is 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/3jjLPyLGnAR#top-down-dynamic-programming-with-memoization)

Let’s use memoization to overcome the overlapping 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/3jjLPyLGnAR#code)

class SCS {

  public int findSCSLength(String s1, String s2) {

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

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

  }

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

    // if we have reached the end of a string, return the remaining length of the other string,

    // as in this case we have to take all of the remaining other string

    if(i1 == s1.length())

      return s2.length()-i2;

    if(i2 == s2.length())

      return s1.length()-i1;

    if(dp[i1][i2] == null) {

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

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

      else {

        int length1 = 1 + findSCSLengthRecursive(dp, s1, s2, i1, i2+1);

        int length2 = 1 + findSCSLengthRecursive(dp, s1, s2, i1+1, i2);

        dp[i1][i2] = Math.min(length1, length2);

      }

    }

    return dp[i1][i2];

  }

  public static void main(String[] args) {

    SCS scs = new SCS();

    System.out.println(scs.findSCSLength("abcf", "bdcf"));

    System.out.println(scs.findSCSLength("dynamic", "programming"));

  }

}

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jjLPyLGnAR#bottom-up-dynamic-programming)

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

1. If the character s1[i] matches s2[j], the length of the SCS would be the one plus the length of the SCS till i-1 and j-1 indexes in the two strings.
2. If the character s1[i] does not match s2[j], we will consider two SCS: one without s1[i] and one without s2[j]. Our required SCS length will be the shortest of these two super-sequences plus one.

So our recursive formula would be:

if s1[i] == s2[j]

  dp[i][j] = 1 + dp[i-1][j-1]

else

  dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1])

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jjLPyLGnAR#code-2)

class SCS {

  public int findSCSLength(String s1, String s2) {

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

    // if one of the strings is of zero length, SCS would be equal to the length of the other string

    for(int i=0; i<=s1.length(); i++)

      dp[i][0] = i;

    for(int j=0; j<=s2.length(); j++)

      dp[0][j] = j;

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

      }

    }

    return dp[s1.length()][s2.length()];

  }

  public static void main(String[] args) {

    SCS scs = new SCS();

    System.out.println(scs.findSCSLength("abcf", "bdcf"));

    System.out.println(scs.findSCSLength("dynamic", "programming"));

  }

}

# Minimum Deletions to Make a Sequence Sorted

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/qV251wjM2ZD#problem-statement)

Given a number sequence, find the minimum number of elements that should be deleted to make the remaining sequence sorted.

**Example 1:**

Input: {4,2,3,6,10,1,12}  
Output: 2  
Explanation: We need to delete {4,1} to make the remaing sequence sorted {2,3,6,10,12}.

**Example 2:**

Input: {-4,10,3,7,15}  
Output: 1  
Explanation: We need to delete {10} to make the remaing sequence sorted {-4,3,7,15}.

**Example 3:**

Input: {3,2,1,0}  
Output: 3  
Explanation: Since the elements are in reverse order, we have to delete all except one to get a   
sorted sequence. Sorted sequences are {3}, {2}, {1}, and {0}

### Basic Solution [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/qV251wjM2ZD#basic-solution)

A basic brute-force solution could be to try deleting all combinations of elements, one by one, and checking if that makes the subsequence sorted.

Alternately, we can convert this problem into a Longest Increasing Subsequence (LIS) problem. As we know that LIS will give us the length of the longest increasing subsequence (in the sorted order!), which means that the elements which are not part of the LIS should be removed to make the sequence sorted. This is exactly what we need. So we’ll get our solution by subtracting the length of LIS from the length of the input array: Length-of-input-array - LIS()

Let’s jump directly to the bottom-up dynamic programming solution.

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/qV251wjM2ZD#bottom-up-dynamic-programming)

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

class MDSS {

  public int findMinimumDeletions(int[] nums){

    // subtracting the length of LIS from the length of the input array to get minimum number of deletions

    return nums.length - findLISLength(nums);

  }

  private int findLISLength(int[] nums) {

    int[] dp = new int[nums.length];

    dp[0] = 1;

    int maxLength = 1;

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

      dp[i] = 1;

      for (int j=0; j<i; j++)

        if (nums[i] > nums[j] && dp[i] <= dp[j] ) {

          dp[i] = dp[j]+1;

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

        }

    }

    return maxLength;

  }

  public static void main(String[] args) {

    MDSS mdss = new MDSS();

    int[] nums = {4,2,3,6,10,1,12};

    System.out.println(mdss.findMinimumDeletions(nums));

    nums = new int[]{-4,10,3,7,15};

    System.out.println(mdss.findMinimumDeletions(nums));

    nums = new int[]{3,2,1,0};

    System.out.println(mdss.findMinimumDeletions(nums));

  }

}

# Longest Repeating Subsequence

### 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}.

**Example 3:**

Input: “f m f f”  
Output: 2  
Explanation: The longest repeating subsequence is “f f” {f m f f, f m f f}. Please note the second last character is shared in LRS.

### 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 (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. In LRS, every corresponding character in the subsequences should not have the same index.

A basic brute-force solution could be to try all subsequences of the given sequence to find the longest repeating one, but the problem is how to ensure that the LRS’s characters do not have the same index. For this, we can start with two indexes in the given sequence, so at any step we have two choices:

1. If the two indexes are not the same and the characters at both the indexes are same, we can recursively match for the remaining length (i.e. by incrementing both the indexes).
2. If the characters at both the indexes don’t match, we start two new recursive calls by incrementing each index separately. The LRS would be the one with the highest length from the two recursive calls.

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

class LRS {

  public int findLRSLength(String str) {

      return findLRSLengthRecursive(str, 0, 0);

  }

  private int findLRSLengthRecursive(String str, int i1, int i2) {

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

      return 0;

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

      return 1 + findLRSLengthRecursive(str, i1+1, i2+1);

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

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

    return Math.max(c1, c2);

  }

  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 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)

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:

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

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"));

  }

}

# Subsequence Pattern Matching

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#problem-statement)

Given a string and a pattern, write a method to count the number of times the pattern appears in the string as a subsequence.

**Example 1:** Input: string: “baxmx”, pattern: “ax”  
Output: 2  
Explanation: {baxmx, baxmx}.

**Example 2:**

Input: string: “tomorrow”, pattern: “tor”  
Output: 4  
Explanation: Following are the four occurences: {tomorrow, tomorrow, tomorrow, tomorrow}.

### Basic Solution [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#basic-solution)

This problem follows the Longest Common Subsequence (LCS) pattern and is quite similar to the Longest Repeating Subsequence; the difference is that we need to count the total occurrences of the subsequence.

A basic brute-force solution could be to try all the subsequences of the given string to count all that match the given pattern. We can match the pattern with the given string one character at a time, so we can do two things at any step:

1. If the pattern has a matching character with the string, we can recursively match for the remaining lengths of the pattern and the string.
2. At every step, we can always skip a character from the string to try to match the remaining string with the pattern. So we can start a recursive call by skipping one character from the string.

Our total count will be the sum of the counts returned by the above two options.

Here is the code:

class SPM {

  public int findSPMCount(String str, String pat) {

    return findSPMCountRecursive(str, pat, 0, 0);

  }

  private int findSPMCountRecursive(String str, String pat, int strIndex, int patIndex) {

    // if we have reached the end of the pattern

    if(patIndex == pat.length())

      return 1;

    // if we have reached the end of the string but pattern has still some characters left

    if(strIndex == str.length())

      return 0;

    int c1 = 0;

    if(str.charAt(strIndex) == pat.charAt(patIndex))

      c1 = findSPMCountRecursive(str, pat, strIndex+1, patIndex+1);

    int c2 = findSPMCountRecursive(str, pat, strIndex+1, patIndex);

    return c1 + c2;

  }

  public static void main(String[] args) {

    SPM spm = new SPM();

    System.out.println(spm.findSPMCount("baxmx", "ax"));

    System.out.println(spm.findSPMCount("tomorrow", "tor"));

  }

}

The time complexity of the above algorithm is exponential O(2^{m}), where ‘m’ is the length of the string, as our recursion stack will not be deeper than m. The space complexity is O(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/JY7yZ2VyJnP#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 strIndex and patIndex. 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 (strIndex + “|” + patIndex)).

Here is the code:

class SPM {

  public int findSPMCount(String str, String pat) {

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

    return findSPMCountRecursive(dp, str, pat, 0, 0);

  }

  private int findSPMCountRecursive(Integer[][] dp, String str, String pat, int strIndex, int patIndex) {

    // if we have reached the end of the pattern

    if(patIndex == pat.length())

      return 1;

    // if we have reached the end of the string but pattern has still some characters left

    if(strIndex == str.length())

      return 0;

    if(dp[strIndex][patIndex] == null) {

      int c1 = 0;

      if(str.charAt(strIndex) == pat.charAt(patIndex))

        c1 = findSPMCountRecursive(dp, str, pat, strIndex+1, patIndex+1);

      int c2 = findSPMCountRecursive(dp, str, pat, strIndex+1, patIndex);

      dp[strIndex][patIndex] = c1 + c2;

    }

    return dp[strIndex][patIndex];

  }

  public static void main(String[] args) {

    SPM spm = new SPM();

    System.out.println(spm.findSPMCount("baxmx", "ax"));

    System.out.println(spm.findSPMCount("tomorrow", "tor"));

  }

}

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#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 separate indexes for the string and the pattern, so we will be doing two things for every value of strIndex and patIndex:

1. If the character at the strIndex (in the string) matches the character at patIndex (in the pattern), the count of the SPM would be equal to the count of SPM up to strIndex-1 and patIndex-1.
2. At every step, we can always skip a character from the string to try matching the remaining string with the pattern; therefore, we can add the SPM count from the indexes strIndex-1 and patIndex.

So our recursive formula would be:

if str[strIndex] == pat[patIndex] {

  dp[strIndex][patIndex] = dp[strIndex-1][patIndex-1]

}

dp[i1][i2] += dp[strIndex-1][patIndex]

#### Code

class SPM {

  public int findSPMCount(String str, String pat) {

    // every empty pattern has one match

    if(pat.length() == 0)

      return 1;

    if(str.length() == 0 || pat.length() > str.length())

      return 0;

    // dp[strIndex][patIndex] will be storing the count of SPM up to str[0..strIndex-1][0..patIndex-1]

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

    // for the empty pattern, we have one matching

    for(int i=0; i<=str.length(); i++)

      dp[i][0] = 1;

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

      for(int patIndex=1; patIndex<=pat.length(); patIndex++) {

        if(str.charAt(strIndex-1) == pat.charAt(patIndex-1))

          dp[strIndex][patIndex] = dp[strIndex-1][patIndex-1];

        dp[strIndex][patIndex] += dp[strIndex-1][patIndex];

      }

    }

    return dp[str.length()][pat.length()];

  }

  public static void main(String[] args) {

    SPM spm = new SPM();

    System.out.println(spm.findSPMCount("baxmx", "ax"));

    System.out.println(spm.findSPMCount("tomorrow", "tor"));

  }

}

# Longest Bitonic Subsequence

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx842pQXGM9#problem-statement)

Given a number sequence, find the length of its Longest Bitonic Subsequence (LBS). A subsequence is considered bitonic if it is monotonically increasing and then monotonically decreasing.

**Example 1:**

Input: {4,2,3,6,10,1,12}  
Output: 5  
Explanation: The LBS is {2,3,6,10,1}.

**Example 2:**

Input: {4,2,5,9,7,6,10,3,1}  
Output: 7  
Explanation: The LBS is {4,5,9,7,6,3,1}.

### Basic Solution [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx842pQXGM9#basic-solution)

A basic brute-force solution could be to try finding the Longest Decreasing Subsequences (LDS), starting from every number in both directions. So for every index ‘i’ in the given array, we will do two things:

1. Find LDS starting from ‘i’ to the end of the array.
2. Find LDS starting from ‘i’ to the beginning of the array.

LBS would be the maximum sum of the above two subsequences.

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx842pQXGM9#code)

class LBS {

  private int findLBSLength(int[] nums) {

    int maxLength = 0;

    for(int i=0; i<nums.length; i++) {

      int c1 = findLDSLength(nums, i, -1);

      int c2 = findLDSLengthRev(nums, i, -1);

      maxLength = Math.max(maxLength, c1+c2-1);

    }

    return maxLength;

  }

  // find the longest decreasing subsequence from currentIndex till the end of the array

  private int findLDSLength(int[] nums, int currentIndex, int previousIndex) {

    if(currentIndex == nums.length)

      return 0;

    // include nums[currentIndex] if it is smaller than the previous number

    int c1 = 0;

    if(previousIndex == -1 || nums[currentIndex] < nums[previousIndex])

      c1 = 1 + findLDSLength(nums, currentIndex+1, currentIndex);

    // excluding the number at currentIndex

    int c2 = findLDSLength(nums, currentIndex+1, previousIndex);

    return Math.max(c1, c2);

  }

  // find the longest decreasing subsequence from currentIndex till the beginning of the array

  private int findLDSLengthRev(int[] nums, int currentIndex, int previousIndex) {

    if(currentIndex < 0)

      return 0;

    // include nums[currentIndex] if it is smaller than the previous number

    int c1 = 0;

    if(previousIndex == -1 || nums[currentIndex] < nums[previousIndex])

      c1 = 1 + findLDSLengthRev(nums, currentIndex-1, currentIndex);

    // excluding the number at currentIndex

    int c2 = findLDSLengthRev(nums, currentIndex-1, previousIndex);

    return Math.max(c1, c2);

  }

  public static void main(String[] args) {

    LBS lbs = new LBS();

    int[] nums = {4,2,3,6,10,1,12};

    System.out.println(lbs.findLBSLength(nums));

    nums = new int[]{4,2,5,9,7,6,10,3,1};

    System.out.println(lbs.findLBSLength(nums));

  }

}

The time complexity of the above algorithm is exponential O(2^n), where ‘n’ is the lengths of the input array. The space complexity is 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/gx842pQXGM9#top-down-dynamic-programming-with-memoization)

To overcome the overlapping subproblems, we can use an array to store the already solved subproblems.

We need to memoize the recursive functions that calculate the longest decreasing subsequence. The two changing values for our recursive function are the current and the previous index. Therefore, we can store the results of all subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string (currentIndex + “|” + previousIndex)).

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx842pQXGM9#code-2)

class LBS {

  private int findLBSLength(int[] nums) {

    Integer[][] lds = new Integer[nums.length][nums.length+1];

    Integer[][] ldsRev = new Integer[nums.length][nums.length+1];

    int maxLength = 0;

    for(int i=0; i<nums.length; i++) {

      int c1 = findLDSLength(lds, nums, i, -1);

      int c2 = findLDSLengthReverse(ldsRev, nums, i, -1);

      maxLength = Math.max(maxLength, c1+c2-1);

    }

    return maxLength;

  }

  // find the longest decreasing subsequence from currentIndex till the end of the array

  private int findLDSLength(Integer[][] dp, int[] nums, int currentIndex, int previousIndex) {

    if(currentIndex == nums.length)

      return 0;

    if(dp[currentIndex][previousIndex+1] == null) {

      // include nums[currentIndex] if it is smaller than the previous number

      int c1 = 0;

      if(previousIndex == -1 || nums[currentIndex] < nums[previousIndex])

        c1 = 1 + findLDSLength(dp, nums, currentIndex+1, currentIndex);

      // excluding the number at currentIndex

      int c2 = findLDSLength(dp, nums, currentIndex+1, previousIndex);

      dp[currentIndex][previousIndex+1] = Math.max(c1, c2);

    }

    return dp[currentIndex][previousIndex+1];

  }

  // find the longest decreasing subsequence from currentIndex till the beginning of the array

  private int findLDSLengthReverse(Integer[][] dp, int[] nums, int currentIndex, int previousIndex) {

    if(currentIndex < 0)

      return 0;

    if(dp[currentIndex][previousIndex+1] == null) {

      // include nums[currentIndex] if it is smaller than the previous number

      int c1 = 0;

      if(previousIndex == -1 || nums[currentIndex] < nums[previousIndex])

        c1 = 1 + findLDSLengthReverse(dp, nums, currentIndex-1, currentIndex);

      // excluding the number at currentIndex

      int c2 = findLDSLengthReverse(dp, nums, currentIndex-1, previousIndex);

      dp[currentIndex][previousIndex+1] = Math.max(c1, c2);

    }

    return dp[currentIndex][previousIndex+1];

  }

  public static void main(String[] args) {

    LBS lbs = new LBS();

    int[] nums = {4,2,3,6,10,1,12};

    System.out.println(lbs.findLBSLength(nums));

    nums = new int[]{4,2,5,9,7,6,10,3,1};

    System.out.println(lbs.findLBSLength(nums));

  }

}

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx842pQXGM9#bottom-up-dynamic-programming)

The above algorithm shows us a clear bottom-up approach. We can separately calculate LDS for every index i.e., from the beginning to the end of the array and vice versa. The required length of LBS would be the one that has the maximum sum of LDS for a given index (from both ends).

#### Code

class LBS {

  private int findLBSLength(int[] nums) {

    int[] lds = new int[nums.length];

    int[] ldsReverse = new int[nums.length];

    // find LDS for every index up to the beginning of the array

    for (int i = 0; i < nums.length; i++) {

      lds[i] = 1; // every element is an LDS of length 1

      for (int j = i - 1; j >= 0; j--)

        if (nums[j] < nums[i]) {

          lds[i] = Math.max(lds[i], lds[j] + 1);

        }

    }

    // find LDS for every index up to the end of the array

    for (int i = nums.length - 1; i >= 0; i--) {

      ldsReverse[i] = 1; // every element is an LDS of length 1

      for (int j = i + 1; j < nums.length; j++)

        if (nums[j] < nums[i]) {

          ldsReverse[i] = Math.max(ldsReverse[i], ldsReverse[j] + 1);

        }

    }

    int maxLength = 0;

    for (int i = 0; i < nums.length; i++) {

      maxLength = Math.max(maxLength, lds[i] + ldsReverse[i] - 1);

    }

    return maxLength;

  }

  public static void main(String[] args) {

    LBS lbs = new LBS();

    int[] nums = { 4, 2, 3, 6, 10, 1, 12 };

    System.out.println(lbs.findLBSLength(nums));

    nums = new int[] { 4, 2, 5, 9, 7, 6, 10, 3, 1 };

    System.out.println(lbs.findLBSLength(nums));

  }

}

# Longest Alternating Subsequence

### Problem Statement [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxp0yR37zBr#problem-statement)

Given a number sequence, find the length of its Longest Alternating Subsequence (LAS). A subsequence is considered alternating if its elements are in alternating order.

A three element sequence (a1, a2, a3) will be an alternating sequence if its elements hold one of the following conditions:

{a1 > a2 < a3 } or { a1 < a2 > a3}.

**Example 1:**

Input: {1,2,3,4}  
Output: 2  
Explanation: There are many LAS: {1,2}, {3,4}, {1,3}, {1,4}

**Example 2:**

Input: {3,2,1,4}  
Output: 3  
Explanation: The LAS are {3,2,4} and {2,1,4}.

**Example 3:**

Input: {1,3,2,4}  
Output: 4  
Explanation: The LAS is {1,3,2,4}.

### Basic Solution [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxp0yR37zBr#basic-solution)

A basic brute-force solution could be to try finding the LAS starting from every number in both ascending and descending order. So for every index ‘i’ in the given array, we will have three options:

1. If the element at ‘i’ is bigger than the last element we considered, we include the element at ‘i’ and recursively process the remaining array to find the next element in descending order.
2. If the element at ‘i’ is smaller than the last element we considered, we include the element at ‘i’ and recursively process the remaining array to find the next element in ascending order.
3. In addition to the above two cases, we can always skip the element ‘i’ to recurse for the remaining array. This will ensure that we try all subsequences.

LAS would be the maximum of the above three subsequences.

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxp0yR37zBr#code)

class LAS {

  public int findLASLength(int[] nums) {

    // we have to start with two recursive calls, one where we will consider that the first element is

    // bigger than the second element and one where the first element is smaller than the second element

    return Math.max(findLASLengthRecursive(nums, -1, 0, true),

                    findLASLengthRecursive(nums, -1, 0, false));

  }

  private int findLASLengthRecursive(int[] nums, int previousIndex, int currentIndex, boolean isAsc) {

    if(currentIndex == nums.length)

      return 0;

    int c1=0;

    // if ascending, the next element should be bigger

    if(isAsc) {

      if(previousIndex == -1 || nums[previousIndex] < nums[currentIndex])

        c1 = 1 + this.findLASLengthRecursive(nums, currentIndex, currentIndex+1, !isAsc);

    } else { // if descending, the next element should be smaller

    if(previousIndex == -1 || nums[previousIndex] > nums[currentIndex])

      c1 = 1 + this.findLASLengthRecursive(nums, currentIndex, currentIndex+1, !isAsc);

    }

    // skip the current element

    int c2 = this.findLASLengthRecursive(nums, previousIndex, currentIndex+1, isAsc);

    return Math.max(c1,c2);

  }

  public static void main(String[] args) {

    LAS las = new LAS();

    int[] nums = {1,2,3,4};

    System.out.println(las.findLASLength(nums));

    nums = new int[]{3,2,1,4};

    System.out.println(las.findLASLength(nums));

    nums = new int[]{1,3,2,4};

    System.out.println(las.findLASLength(nums));

  }

}

The time complexity of the above algorithm is exponential O(2^n), where ‘n’ is the lengths of the input array. The space complexity is 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/gxp0yR37zBr#top-down-dynamic-programming-with-memoization)

To overcome the overlapping subproblems, we can use an array to store the already solved subproblems.

The three changing values for our recursive function are the current and the previous indexes and the isAsc flag. Therefore, we can store the results of all subproblems in a three-dimensional array, where the third dimension will be of size two, to store the boolean flag isAsc. (Another alternative could be to use a hash-table whose key would be a string (currentIndex + “|” + previousIndex + “|” + isAsc)).

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxp0yR37zBr#code-2)

class LAS {

  public int findLASLength(int[] nums) {

    Integer[][][] dp = new Integer[nums.length][nums.length][2];

    return Math.max(findLASLengthRecursive(dp, nums, -1, 0, true),

                    findLASLengthRecursive(dp, nums, -1, 0, false));

  }

  private int findLASLengthRecursive(Integer[][][] dp, int[] nums,

      int previousIndex, int currentIndex, boolean isAsc) {

    if(currentIndex == nums.length)

      return 0;

    if(dp[previousIndex+1][currentIndex][isAsc ? 1 : 0] == null) {

      int c1=0;

      // if ascending, the next element should be bigger

      if(isAsc) {

        if(previousIndex == -1 || nums[previousIndex] < nums[currentIndex])

          c1 = 1 + this.findLASLengthRecursive(dp, nums, currentIndex, currentIndex+1, !isAsc);

      } else { // if descending, the next element should be smaller

        if(previousIndex == -1 || nums[previousIndex] > nums[currentIndex])

          c1 = 1 + this.findLASLengthRecursive(dp, nums, currentIndex, currentIndex+1, !isAsc);

      }

      // skip the current element

      int c2 = this.findLASLengthRecursive(dp, nums, previousIndex, currentIndex+1, isAsc);

      dp[previousIndex+1][currentIndex][isAsc ? 1 : 0] = Math.max(c1,c2);

    }

    return dp[previousIndex+1][currentIndex][isAsc ? 1 : 0];

  }

  public static void main(String[] args) {

    LAS las = new LAS();

    int[] nums = {1,2,3,4};

    System.out.println(las.findLASLength(nums));

    nums = new int[]{3,2,1,4};

    System.out.println(las.findLASLength(nums));

    nums = new int[]{1,3,2,4};

    System.out.println(las.findLASLength(nums));

  }

}

### Bottom-up Dynamic Programming [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxp0yR37zBr#bottom-up-dynamic-programming)

The above algorithm tells us three things:

1. We need to find an ascending and descending subsequence at every index.
2. While finding the next element in the ascending order, if the number at the current index is bigger than the number at the previous index, we increment the count for a LAS up to the current index. But if there is a bigger LAS without including the number at the current index, we take that.
3. Similarly for the descending order, if the number at the current index is smaller than the number at the previous index, we increment the count for a LAS up to the current index. But if there is a bigger LAS without including the number at the current index, we take that.

To find the largest LAS, we need to find all of the LAS for a number at index ‘i’ from all the previous numbers (i.e. number till index ‘i-1’).

We can use two arrays to store the length of LAS, one for ascending order and one for descending order. (Actually, we will use a two-dimensional array, where the second dimension will be of size two).

If ‘i’ represents the currentIndex and ‘j’ represents the previousIndex, our recursive formula would look like:

* If nums[i] is bigger than nums[j] then we will consider the LAS ending at ‘j’ where the last two elements were in descending order =>
* if num[i] > num[j] => dp[i][0] = 1 + dp[j][1], if there is no bigger LAS for 'i'
* If nums[i] is smaller than nums[j] then we will consider the LAS ending at ‘j’ where the last two elements were in ascending order =>
* if num[i] < num[j] => dp[i][1] = 1 + dp[j][0], if there is no bigger LAS for 'i'

# Code

class LAS {

  public int findLASLength(int[] nums) {

    if(nums.length == 0) return 0;

    //dp[i][0] = stores the LAS ending at 'i' such that the last two elements are in ascending order

    //dp[i][1] = stores the LAS ending at 'i' such that the last two elements are in descending order

    int[][] dp = new int[nums.length][2];

    int maxLength = 1;

    for(int i=0; i < nums.length; i++) {

      // every single element can be considered as LAS of length 1

      dp[i][0] = dp[i][1] = 1;

      for(int j=0; j < i; j++) {

        if(nums[i] > nums[j]) {

          // if nums[i] is BIGGER than nums[j] then we will consider the LAS ending at 'j' where the

          // last two elements were in DESCENDING order

          dp[i][0] = Math.max(dp[i][0], 1 + dp[j][1]);

          maxLength = Math.max(maxLength, dp[i][0]);

        } else if (nums[i] != nums[j]) { // if the numbers are equal don't do anything

          // if nums[i] is SMALLER than nums[j] then we will consider the LAS ending at 'j' where the

          // last two elements were in ASCENDING order

          dp[i][1] = Math.max(dp[i][1], 1 + dp[j][0]);

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

        }

      }

    }

    return maxLength;

  }

  public static void main(String[] args) {

    LAS las = new LAS();

    int[] nums = {1,2,3,4};

    System.out.println(las.findLASLength(nums));

    nums = new int[]{3,2,1,4};

    System.out.println(las.findLASLength(nums));

    nums = new int[]{1,3,2,4};

    System.out.println(las.findLASLength(nums));

  }

}