**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}.

### 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)

Here is the 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)*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/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)

Here is the code:

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:

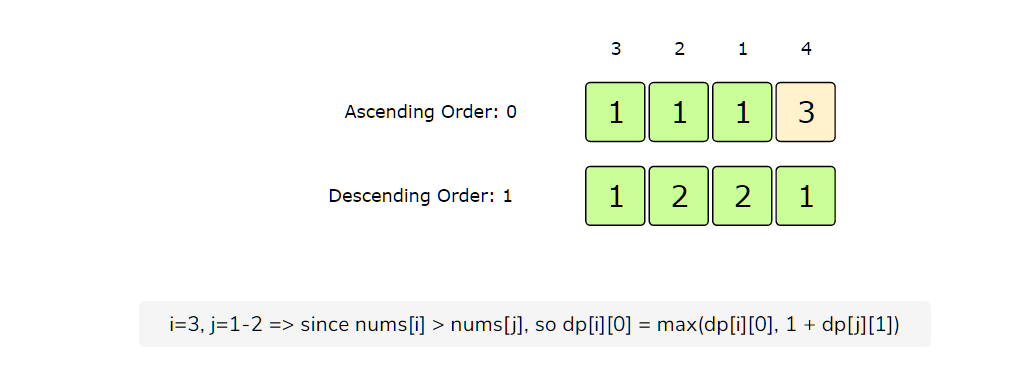
* If nums[i] is bigger than nums[j] then we will consider the LAS ending at ‘j’ where the last two elements were in descending order =>

      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'

Let’s draw this visually for {3,2,1,4}. Start with a subsequence of length ‘1’, as every number can be a LAS of length ‘1’:



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));

  }

}

The time complexity of the above algorithm is O(n^2)and the space complexity is O(n).