<https://leetcode.com/problems/number-of-longest-increasing-subsequence/description/>

Given an integer array nums, return *the number of longest increasing subsequences.*

**Notice** that the sequence has to be **strictly** increasing.

**Example 1:**

Input: nums = [1,3,5,4,7]

Output: 2

Explanation: The two longest increasing subsequences are [1, 3, 4, 7] and [1, 3, 5, 7].

**Example 2:**

Input: nums = [2,2,2,2,2]

Output: 5

Explanation: The length of the longest increasing subsequence is 1, and there are 5 increasing subsequences of length 1, so output 5.

**Constraints:**

1 <= nums.length <= 2000

-10^6 <= nums[i] <= 10^6

The answer is guaranteed to fit inside a 32-bit integer.

**Attempt 1: 2023-04-09**

**Solution 1.1: Native DFS - Divide and Conquer (360 min, TLE 34/223)**

**Style 1: 'prev' as actual value**

class Solution {

    public int findNumberOfLIS(int[] nums) {

        return helper(nums, 0, -1000001)[1];

    }

    private int[] helper(int[] nums, int index, int prev) {

        // Base case: If we’ve iterated through all elements, return {0, 1}

        // [0] -> Length of LIS

        // [1] -> Count of LIS

        if(index >= nums.length) {

            return new int[]{0, 1};

        }

        // Not take

        int[] not\_take = helper(nums, index + 1, prev);

        int not\_take\_len = not\_take[0];

        int not\_take\_count = not\_take[1];

        // Take

        int take\_len = 0;

        int take\_count = 0;

        if(nums[index] > prev) {

            int[] take = helper(nums, index + 1, nums[index]);

            take\_len = take[0] + 1;

            take\_count = take[1];

        }

        if(take\_len > not\_take\_len) {

            return new int[]{take\_len, take\_count};

        } else if(take\_len == not\_take\_len) {

            return new int[]{take\_len, take\_count + not\_take\_count};

        } else {

            return new int[]{not\_take\_len, not\_take\_count};

        }

    }

}

Time Complexity:

There are 2 choices at every element: either take it or not take it.

For each element, these two choices branch out recursively, so the number of recursive calls increases exponentially.

Thus, the time complexity is O(2^n), where n is the number of elements in the input array.

Space Complexity:

The recursion depth is proportional to the number of elements n, so the space complexity due to recursion is O(n).

**Refer to**

<https://leetcode.com/problems/number-of-longest-increasing-subsequence/solutions/4651203/python-dp-from-recursive-relation-to-memoization-to-tabulation-intuitively/>

**1. Find Recurisve Relation**

Trivially, this problem has two parts we need to solve:

Find the longest increasing subsequence

Find the count of the longest increasing subsequence

The tricky part is finding a recurrence relation that is able to do both parts concurrently.

**1.1** For part 1 of this issue, we can check for strictly increasing subsequences as follows:

def lis(start, prev):

#Base case, we've iterated through all of nums, and return 0

if start == len(nums):

return 0

taken\_length = 0

if nums[start] > prev:

taken\_length = lis(start + 1, nums[start]) + 1

not\_taken\_length = lis(start + 1, prev)

return max(taken\_length, not\_taken\_length)

return lis(0, float('-inf'))

For each num in nums, there are two cases:

Choose to include num into the subsequence, and increment the length of the subsequence.

Choose to not include num into the subsequence, and not increment the length of the subsequence.

Trivially the subsequence can only include strictly increasing numbers, so we only enter case 1 when the current num is greater than the last entry of the current subsequence.

We then want to return the greater of the two lengths.

**1.2** Adding the count of the longest increasing subsequence is actually quite facile:

def count\_lis(start, prev):

#Base case, we've iterated through all of nums, and return 0 for a subsequence of length 0 and 1 for a count of 1 for a subsequence of length 0

if start == len(nums):

return 0, 1 # Length of LIS, Count of LIS

taken\_length, taken\_count = 0, 0

if nums[start] > prev:

taken\_length, taken\_count = count\_lis(start + 1, nums[start])

taken\_length += 1

not\_taken\_length, not\_taken\_count = count\_lis(start + 1, prev)

if taken\_length > not\_taken\_length:

return taken\_length, taken\_count

elif taken\_length == not\_taken\_length:

return taken\_length, taken\_count + not\_taken\_count

else:

return not\_taken\_length, not\_taken\_count

return count\_lis(0, float('-inf'))[1]

Instead of only maintaining the length of the current subsequence, we maintain an ordered pair of (length, count) where count is the number of subsequences of length.

In 1.1, we only sought to return the greater of two lengths (since we were only considered with the length of the longest increasing subsequence) and as such simply used max(taken\_length, not\_taken\_length). Since we are now considering count, we must compare the two values taken\_length and not\_taken\_length case by case (since count must be updated to the corresponding count of the longer length):

taken\_length > not\_taken\_length: we simply return taken\_length,taken\_count

taken\_length == not\_taken\_length: since the length of the two are equal, we update count to be both of the respective counts combined

taken\_length < not\_taken\_length: we simply return not\_taken\_length,not\_taken\_count

With the simple replacement of max(taken\_length, not\_taken\_length) with the case by case checking of taken\_length and not\_taken\_length we are able to maintain a count of the longest increasing subsequence.

**2. Recursive (top-down)**

In totality:

class Solution:

def findNumberOfLIS(self, nums: List[int]) -> int:

def count\_lis(start, prev):

if start == len(nums):

return 0, 1 # Length of LIS, Count of LIS

taken\_length, taken\_count = 0, 0

if nums[start] > prev:

taken\_length, taken\_count = count\_lis(start + 1, nums[start])

taken\_length += 1

not\_taken\_length, not\_taken\_count = count\_lis(start + 1, prev)

if taken\_length > not\_taken\_length:

return taken\_length , taken\_count

elif taken\_length == not\_taken\_length:

return taken\_length , taken\_count + not\_taken\_count

else:

return not\_taken\_length, not\_taken\_count

return count\_lis(0, float('-inf'))[1]

**Style 2: 'prev' as index (inspired by DFS + Memoization as Solution 2)**

class Solution {

    public int findNumberOfLIS(int[] nums) {

        return helper(nums, 0, -1)[1];

    }

    private int[] helper(int[] nums, int index, int prev) {

        // Base case: If we’ve iterated through all elements, return {0, 1}

        // [0] -> Length of LIS

        // [1] -> Count of LIS

        if(index >= nums.length) {

            return new int[]{0, 1};

        }

        // Not take

        int[] not\_take = helper(nums, index + 1, prev);

        int not\_take\_len = not\_take[0];

        int not\_take\_count = not\_take[1];

        // Take

        int take\_len = 0;

        int take\_count = 0;

        if(prev == -1 || nums[index] > nums[prev]) {

            int[] take = helper(nums, index + 1, index);

            take\_len = take[0] + 1;

            take\_count = take[1];

        }

        if(take\_len > not\_take\_len) {

            return new int[]{take\_len, take\_count};

        } else if(take\_len == not\_take\_len) {

            return new int[]{take\_len, take\_count + not\_take\_count};

        } else {

            return new int[]{not\_take\_len, not\_take\_count};

        }

    }

}

Time Complexity:

There are 2 choices at every element: either take it or not take it.

For each element, these two choices branch out recursively, so the number of recursive calls increases exponentially.

Thus, the time complexity is O(2^n), where n is the number of elements in the input array.

Space Complexity:

The recursion depth is proportional to the number of elements n, so the space complexity due to recursion is O(n).

**Solution 1.2: Native DFS - For loop inside recursion method (30 min, TLE 34/223)**

class Solution {

    public int findNumberOfLIS(int[] nums) {

        int maxLength = 0;

        int countLIS = 0;

        // For each index, start a DFS to calculate LIS length and count

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

            // result[0] is length, result[1] is count

            int[] result = helper(nums, i);

            if(result[0] > maxLength) {

                maxLength = result[0];

                countLIS = result[1];

            } else if(result[0] == maxLength) {

                countLIS += result[1];

            }

        }

        return countLIS;

    }

    // Helper method to find the length and count of LIS ending at index i

    private int[] helper(int[] nums, int i) {

        // Minimum LIS length is 1 (element itself)

        int maxLength = 1;

        // Count for subsequence length 1 is 1

        int countLIS = 1;

        // Loop over all previous elements to form increasing subsequences

        for(int j = 0; j < i; j++) {

            if(nums[j] < nums[i]) {

                // Recursively find LIS for index j

                int[] result = helper(nums, j);

                // Update countLIS to count of the longer subsequences

                if(result[0] + 1 > maxLength) {

                    maxLength = result[0] + 1;

                    countLIS = result[1];

                // Add to count of subsequences of equal length

                } else if(result[0] + 1 == maxLength) {

                    countLIS += result[1];

                }

            }

        }

        // Return length and count of LIS

        return new int[]{maxLength, countLIS};

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Refer to chatGPT**

**Problem Understanding:**

You want to find both the length and the number of the longest increasing subsequences (LIS) in the array. You can build the solution in a similar way to how you did for Leetcode 300 but with an additional step to count the number of times the LIS occurs.

**Java DFS solution with a for loop in recursion:**

class Solution {

public int findNumberOfLIS(int[] nums) {

int maxLength = 0;

int countLIS = 0;

// For each index, start a DFS to calculate LIS length and count

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

int[] result = helper(nums, i); // result[0] is length, result[1] is count

if (result[0] > maxLength) {

maxLength = result[0];

countLIS = result[1];

} else if (result[0] == maxLength) {

countLIS += result[1];

}

}

return countLIS;

}

// Helper method to find the length and count of LIS starting from index `i`

private int[] helper(int[] nums, int i) {

int maxLength = 1; // Minimum LIS length is 1 (element itself)

int countLIS = 1; // Count for subsequence length 1 is 1

// Loop over all previous elements to form increasing subsequences

for (int j = 0; j < i; j++) {

if (nums[j] < nums[i]) {

int[] result = helper(nums, j); // Recursively find LIS for index j

if (result[0] + 1 > maxLength) {

maxLength = result[0] + 1;

countLIS = result[1]; // Update countLIS to count of the longer subsequences

} else if (result[0] + 1 == maxLength) {

countLIS += result[1]; // Add to count of subsequences of equal length

}

}

}

return new int[]{maxLength, countLIS}; // Return length and count of LIS

}

}

**Explanation:**

Main DFS Loop:

We loop over each element in nums[] and treat each as a potential starting point for a DFS.

For each element, we call the helper function to get the LIS length and count starting from that element.

We then update the overall maximum length (maxLength) and count (countLIS).

Helper Function:

For each element at index i, the helper function recursively checks all previous elements (j) to see if we can extend an increasing subsequence.

If nums[j] < nums[i], we call helper on j to find the LIS length and count for that subsequence.

If including nums[i] results in a longer subsequence, we update maxLength and reset countLIS to the count of that longer subsequence.

If the subsequence length remains the same, we simply add the counts to keep track of the number of subsequences of that length.

**Time Complexity:**

Since this is a pure DFS without memoization, the time complexity is **exponential** in the worst case: O(2^n), where n is the length of the input array.

This is inefficient for large inputs, but it demonstrates the recursive DFS approach in a clear way.

For a more efficient solution, you would need to introduce **memoization** or dynamic programming as you've seen in other solutions for Leetcode 673.

**Solution 2.1: DFS + Divide and Conquer + Memoization (10 min)**

**Style 1: Second dimension based on actual value of 'prev' (TLE, 222/223)**

**Since the return type is not primary int, but int[], we better use Map to create memo**

class Solution {

    public int findNumberOfLIS(int[] nums) {

        // Since the return type is not primary int, but int[], we better use Map to create memo

        Map<String, int[]> memo = new HashMap<>();

        return helper(nums, 0, -1000001, memo)[1];

    }

    private int[] helper(int[] nums, int index, int prev, Map<String, int[]> memo) {

        // Base case: If we’ve iterated through all elements, return {0, 1}

        // [0] -> Length of LIS

        // [1] -> Count of LIS

        if(index >= nums.length) {

            return new int[]{0, 1};

        }

        String key = index + "\_" + prev;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        // Not take

        int[] not\_take = helper(nums, index + 1, prev, memo);

        int not\_take\_len = not\_take[0];

        int not\_take\_count = not\_take[1];

        // Take

        int take\_len = 0;

        int take\_count = 0;

        if(nums[index] > prev) {

            int[] take = helper(nums, index + 1, nums[index], memo);

            take\_len = take[0] + 1;

            take\_count = take[1];

        }

        int[] result;

        if(take\_len > not\_take\_len) {

            result = new int[]{take\_len, take\_count};

        } else if(take\_len == not\_take\_len) {

            result = new int[]{take\_len, take\_count + not\_take\_count};

        } else {

            result = new int[]{not\_take\_len, not\_take\_count};

        }

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n^2)

**Refer to chatGPT**

To add memoization to the solution, we need to store results of previous computations so that we avoid recalculating the length and count of the longest increasing subsequence (LIS) for the same states. A common approach for memoizing a solution is using a Map or a 2D array.

In this case, we can use a HashMap to memoize the results, where the key is a combination of index and prev, and the value is the result of helper(nums, index, prev).

Here's how the memoized solution looks in Java:

**Memoized Java Solution:**

import java.util.HashMap;

import java.util.Map;

class Solution {

private Map<String, int[]> memo;

public int findNumberOfLIS(int[] nums) {

// Initialize memoization map

memo = new HashMap<>();

// Start the recursive helper with index 0 and initial prev value as -1000001 (or Integer.MIN\_VALUE)

return helper(nums, 0, -1000001)[1];

}

private int[] helper(int[] nums, int index, int prev) {

// Base case: If we’ve iterated through all elements, return {0, 1}

if (index >= nums.length) {

return new int[]{0, 1};

}

// Create a unique key for the current state of (index, prev)

String key = index + "," + prev;

// Check if the result for this state has already been computed

if (memo.containsKey(key)) {

return memo.get(key);

}

// Option 1: Not take the current element

int[] not\_take = helper(nums, index + 1, prev);

int not\_take\_len = not\_take[0];

int not\_take\_count = not\_take[1];

// Option 2: Take the current element if it’s greater than the previous one

int take\_len = 0, take\_count = 0;

if (nums[index] > prev) {

int[] take = helper(nums, index + 1, nums[index]);

take\_len = take[0] + 1;

take\_count = take[1];

}

// Compare the results from taking and not taking the current element

int[] result;

if (take\_len > not\_take\_len) {

result = new int[]{take\_len, take\_count};

} else if (take\_len == not\_take\_len) {

result = new int[]{take\_len, take\_count + not\_take\_count};

} else {

result = new int[]{not\_take\_len, not\_take\_count};

}

// Memoize the result for the current state

memo.put(key, result);

return result;

}

}

**Key Changes:**

Memoization Map: We introduce a HashMap<String, int[]> called memo. The key is a unique string composed of index and prev, and the value is the result of the helper function for that state (both the length and count of LIS).

Memoization Check: Before performing any recursion, we check if the result for the current state (index, prev) is already stored in the map. If so, we directly return the stored result.

Storing Results: After computing the result for a specific (index, prev), we store it in the memo map.

**Time Complexity:**

The time complexity is O(n^2), where n is the length of the array. Each (index, prev) combination can only be computed once, and there are at most n \* n states in total (since prev can take up to n different values in practice).

**Space Complexity:**

The space complexity is O(n^2), due to the memoization table storing results for up to n \* n states.

This memoized version optimizes the original DFS solution by avoiding redundant calculations, leading to significant performance improvements.

**Refer to**

<https://leetcode.com/problems/number-of-longest-increasing-subsequence/solutions/4651203/python-dp-from-recursive-relation-to-memoization-to-tabulation-intuitively/>

We simply add a memoization device without altering the code drastically:

class Solution:

def findNumberOfLIS(self, nums: List[int]) -> int:

memo = {}

def count\_lis(start, prev):

if start == len(nums):

return 0, 1

if (start, prev) in memo:

return memo[(start, prev)]

taken\_length, taken\_count = 0, 0

if nums[start] > prev:

taken\_length, taken\_count = count\_lis(start + 1, nums[start])

taken\_length += 1

not\_taken\_length, not\_taken\_count = count\_lis(start + 1, prev)

if taken\_length > not\_taken\_length:

memo[(start, prev)] = taken\_length, taken\_count

elif taken\_length == not\_taken\_length:

memo[(start, prev)] = taken\_length, taken\_count + not\_taken\_count

else:

memo[(start, prev)] = not\_taken\_length, not\_taken\_count

return memo[(start, prev)]

return count\_lis(0, float('-inf'))[1]

**Style 2: Second dimension based on index of 'prev' (TLE, 195/223)**

class Solution {

    public int findNumberOfLIS(int[] nums) {

        Map<String, int[]> memo = new HashMap<>();

        return helper(nums, 0, -1, memo)[1];

    }

    private int[] helper(int[] nums, int index, int prev, Map<String, int[]> memo) {

        // Base case: If we’ve iterated through all elements, return {0, 1}

        // [0] -> Length of LIS

        // [1] -> Count of LIS

        if(index >= nums.length) {

            return new int[]{0, 1};

        }

        String key = index + "\_" + prev;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        // Not take

        int[] not\_take = helper(nums, index + 1, prev, memo);

        int not\_take\_len = not\_take[0];

        int not\_take\_count = not\_take[1];

        // Take

        int take\_len = 0;

        int take\_count = 0;

        if(prev == -1 || nums[index] > nums[prev]) {

            int[] take = helper(nums, index + 1, index, memo);

            take\_len = take[0] + 1;

            take\_count = take[1];

        }

        int[] result;

        if(take\_len > not\_take\_len) {

            result = new int[]{take\_len, take\_count};

        } else if(take\_len == not\_take\_len) {

            result = new int[]{take\_len, take\_count + not\_take\_count};

        } else {

            result = new int[]{not\_take\_len, not\_take\_count};

        }

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n^2)

**Solution 2.2: DFS + For loop inside recursion method + Memoization (10 min)**

**Stye 1: Memo create as 2D array int[][]**

**Refer to chatGPT**

To add memoization to the given solution for Leetcode 673, we need to store the results of the recursive calls so that we don’t recompute the same results multiple times. We can use a 2D array to cache the results based on the index of the current element and the state of the computation.

class Solution {

public int findNumberOfLIS(int[] nums) {

int maxLength = 0;

int countLIS = 0;

// Create memoization table for storing results

// dp[i][0] -> length of LIS ending at index i

// dp[i][1] -> count of LIS ending at index i

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

// Initialize the memo array with -1 (to indicate uncomputed values)

for(int[] m : memo) {

// -1 indicates the value has not been computed

            Arrays.fill(m, -1);

        }

// For each index, start a DFS to calculate LIS length and count

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

int[] result = helper(nums, i, memo);

if (result[0] > maxLength) {

maxLength = result[0];

countLIS = result[1];

} else if (result[0] == maxLength) {

countLIS += result[1];

}

}

return countLIS;

}

// Helper method to find the length and count of LIS ending at index i

private int[] helper(int[] nums, int i, int[][] memo) {

// If already computed, return the memoized result

if (memo[i][0] != -1) {

return memo[i];

}

// Minimum LIS length is 1 (element itself)

int maxLength = 1;

// Count for subsequence length 1 is 1

int countLIS = 1;

// Loop over all previous elements to form increasing subsequences

for (int j = 0; j < i; j++) {

if (nums[j] < nums[i]) {

// Recursively find LIS for index j

int[] result = helper(nums, j, memo);

// Update countLIS to count of the longer subsequences

if (result[0] + 1 > maxLength) {

maxLength = result[0] + 1;

countLIS = result[1];

// Add to count of subsequences of equal length

} else if (result[0] + 1 == maxLength) {

countLIS += result[1];

}

}

}

// Store the result in the memoization table

memo[i][0] = maxLength;

memo[i][1] = countLIS;

return new int[]{maxLength, countLIS};

}

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Explanation of the Memoized Solution:**

Memoization Array: We create a 2D array memo where memo[i][0] stores the length of the longest increasing subsequence (LIS) ending at index i, and memo[i][1] stores the count of such LISs.

Initialization: Initially, all values in memo[i][0] are set to -1, which indicates that the value has not been computed.

Recursive Calls: The helper function is now modified to check if the result for a given index i is already computed. If it is, we return the cached result from memo. Otherwise, we compute the result and store it in memo before returning.

Time Complexity: Since we now cache the results for each index and reuse them, the time complexity is reduced to **O(n²)**, where n is the number of elements in nums.

This approach avoids recomputation and significantly improves performance over the plain recursive (DFS) solution.

**Stye 2: Memo create as Hash Map**

**Refer to chatGPT**

We can implement memoization using a Map in Java instead of a 2D array. The benefit of using a Map is that it allows you to be more flexible with the types of keys and values, and it can be useful when your memoization involves more complex state tracking (such as multiple variables like in this case).

Here’s how you can convert the 2D array-based memoization to use a Map:

class Solution {

    public int findNumberOfLIS(int[] nums) {

        int maxLength = 0;

        int countLIS = 0;

        // Create a Map to memoize the results for each index

        // Key: index i, Value: array {length of LIS ending at i, count of LIS ending at i}

        Map<Integer, int[]> memo = new HashMap<>();

        // For each index, start a DFS to calculate LIS length and count

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

            int[] result = helper(nums, i, memo);

            if (result[0] > maxLength) {

                maxLength = result[0];

                countLIS = result[1];

            } else if (result[0] == maxLength) {

                countLIS += result[1];

            }

        }

        return countLIS;

    }

    // Helper method to find the length and count of LIS ending at index i

    private int[] helper(int[] nums, int i, Map<Integer, int[]> memo) {

        // If already computed, return the memoized result

        if (memo.containsKey(i)) {

            return memo.get(i);

        }

        // Minimum LIS length is 1 (element itself)

        int maxLength = 1;

        // Count for subsequence length 1 is 1

        int countLIS = 1;

        // Loop over all previous elements to form increasing subsequences

        for (int j = 0; j < i; j++) {

            if (nums[j] < nums[i]) {

                // Recursively find LIS for index j

                int[] result = helper(nums, j, memo);

                // Update countLIS to count of the longer subsequences

                if (result[0] + 1 > maxLength) {

                    maxLength = result[0] + 1;

                    countLIS = result[1];

                } else if (result[0] + 1 == maxLength) {

                    countLIS += result[1];

                }

            }

        }

        // Store the result in the memoization map

        memo.put(i, new int[]{maxLength, countLIS});

        return new int[]{maxLength, countLIS};

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Key Changes:**

Map Instead of Array:

Instead of using a 2D array memo[i][0] and memo[i][1], we now use a Map<Integer, int[]>. The key is the index i, and the value is an array where:

value[0] is the length of the LIS ending at index i.

value[1] is the count of such LISs.

Memoization Using **Map:**

We check if the result for index i is already stored in the Map by using memo.containsKey(i).

If the result is found, we return it directly from the Map using memo.get(i).

Complexity:

The time complexity remains **O(n²)** because each recursive call is made once for each index, and the result is stored and reused using the Map.

Using a Map for memoization provides flexibility and makes the solution cleaner, especially when working with variable-sized problems or more complex states.

**Solution 3: DP (360 min)**

**Style 1**

class Solution {

    public int findNumberOfLIS(int[] nums) {

        int result = 0;

        int maxLen = 0;

        int n = nums.length;

        // len[i]: the length of the Longest Increasing Subsequence which ends with nums[i]

        // count[i]: the number of the Longest Increasing Subsequence which ends with nums[i]

        int[] len = new int[n];

        int[] count = new int[n];

        Arrays.fill(len, 1);

        Arrays.fill(count, 1);

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < i; j++) {

                // If combining with i makes an increasing subsequence

                if(nums[j] < nums[i]) {

                    // If combining with i makes another longest increasing subsequence

                    if(len[i] == len[j] + 1) {

                        /\*\*

                        Why is doing count[i] += count[j] and not simply increasing

                        the count[i] += 1 ?

                        Consider this example [1,3,5,4,9,8,10]

                        This is the step by step output for the last index:

                        Length [1, 2, 3, 3, 4, 4, 2]

                        Count  [1, 1, 1, 1, 2, 2, 1]

                        Length [1, 2, 3, 3, 4, 4, 3]

                        Count  [1, 1, 1, 1, 2, 2, 1]

                        Length [1, 2, 3, 3, 4, 4, 4]

                        Count  [1, 1, 1, 1, 2, 2, 1]

                        Length [1, 2, 3, 3, 4, 4, 4]

                        Count  [1, 1, 1, 1, 2, 2, 2]

                        Length [1, 2, 3, 3, 4, 4, 5]

                        Count  [1, 1, 1, 1, 2, 2, 2]

                        Length [1, 2, 3, 3, 4, 4, 5]

                        Count  [1, 1, 1, 1, 2, 2, 4]

                        Let's understand when does the count for 10 get incremented from 1 to 2.

                        When I get a sequence of 1,3,4,10. So the reason i have 2 at the last

                        index is because I got two sequences with the same length

                        e.g. 1 3 5 10 and 1 3 4 10

                        Now, when we include a 9 in the sequence, why do we still have 2

                        sequences, because 9 goes inside those existing sequences, so the

                        sequences become 1 3 5 9 10 and 1 3 4 9 10. So u still have 2 sequences.

                        Now, comes the 8

                        The reason we have 4 as the answer at the last index is because think

                        about it, there were 2 ways to get to 8 i.e 1 3 5 8 and 1 3 4 8

                        Appending 10 will still get us to those two ways. But 10 has two ways

                        already that it can be reached without even considering 8 at all which

                        are 1 3 5 9 10 and 1 3 4 9 10. So the total answer is no of ways to reach

                        10 via 8 + ways not via 8 which is 4

                        \*/

                        count[i] += count[j];

                    }

                    // If combining with i makes a longer increasing subsequence

                    if(len[i] < len[j] + 1) {

                        len[i] = len[j] + 1;

                        count[i] = count[j];

                    }

                }

            }

            // The result is the sum of each count[i] while its corresponding

            // len[i] is the maximum length

            if(maxLen == len[i]) {

                result += count[i];

            }

            if(maxLen < len[i]) {

                maxLen = len[i];

                result = count[i];

            }

        }

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/number-of-longest-increasing-subsequence/solutions/107293/java-c-simple-dp-solution-with-explanation/>

The idea is to use two arrays len[n] and cnt[n] to record the maximum length of Increasing Subsequence and the corresponding number of these sequence which ends with nums[i], respectively. That is:

len[i]: the length of the Longest Increasing Subsequence which ends with nums[i].

cnt[i]: the number of the Longest Increasing Subsequence which ends with nums[i].

Then, the result is the sum of each cnt[i] while its corresponding len[i] is the maximum length.

public int findNumberOfLIS(int[] nums) {

    int n = nums.length, res = 0, max\_len = 0;

    int[] len =  new int[n], cnt = new int[n];

    for(int i = 0; i<n; i++){

        len[i] = cnt[i] = 1;

        for(int j = 0; j < i; j++){

            if(nums[i] > nums[j]){

                if(len[i] == len[j] + 1)cnt[i] += cnt[j];

                if(len[i] < len[j] + 1){

                    len[i] = len[j] + 1;

                    cnt[i] = cnt[j];

                }

            }

        }

        if(max\_len == len[i])res += cnt[i];

        if(max\_len < len[i]){

            max\_len = len[i];

            res = cnt[i];

        }

    }

    return res;

}

**Why doing count[i] += count[j] and not simply increasing the count[i] += 1 ?**

**Refer to**

<https://leetcode.com/problems/number-of-longest-increasing-subsequence/solutions/500880/java-dp-with-explanation/comments/619873>

<https://leetcode.com/problems/number-of-longest-increasing-subsequence/solutions/500880/java-dp-with-explanation/comments/748606>

This is the step by step output for the last index:

Length [1, 2, 3, 3, 4, 4, 2]

Count [1, 1, 1, 1, 2, 2, 1]

Length [1, 2, 3, 3, 4, 4, 3]

Count [1, 1, 1, 1, 2, 2, 1]

Length [1, 2, 3, 3, 4, 4, 4]

Count [1, 1, 1, 1, 2, 2, 1]

Length [1, 2, 3, 3, 4, 4, 4]

Count [1, 1, 1, 1, 2, 2, 2]

Length [1, 2, 3, 3, 4, 4, 5]

Count [1, 1, 1, 1, 2, 2, 2]

Length [1, 2, 3, 3, 4, 4, 5]

Count [1, 1, 1, 1, 2, 2, 4]

Let's understand when does the count for 10 get incremented from 1 to 2. When I get a sequence of 1,3,4,10. So the reason i have 2 at the last index is because I got two sequences with the same length, e.g. 1 3 5 10 and 1 3 4 10.

Now, when we include a 9 in the sequence, why do we still have 2 sequences, because 9 goes inside those existing sequences, so the sequences become 1 3 5 9 10 and 1 3 4 9 10. So u still have 2 sequences

Now, comes the 8.The reason we have 4 as the answer at the last index is because think about it, there were 2 ways to get to 8 i.e. 1 3 5 8 and 1 3 4 8.

Appending 10 will still get us to those two ways. But 10 has two ways already that it can be reached without even considering 8 at all which are 1 3 5 9 10 and 1 3 4 9 10. So the total answer is no of ways to reach 10 via 8 + ways not via 8 which is 4.

**Style 2: Another format which split out the for loop focus on final result sum up**

class Solution {

    public int findNumberOfLIS(int[] nums) {

        if(nums == null || nums.length == 0) {

            return 0;

        }

        // state:

        // The idea is to use two arrays f[n] and cnt[n] to record the maximum length of

        // Increasing Subsequence and the coresponding number of these sequence which

        // ends with nums[i], respectively. That is:

        // f[i]: the length of the Longest Increasing Subsequence which ends with nums[i].

        // cnt[i]: the number of the Longest Increasing Subsequence which ends with nums[i].

        // Then, the result is the sum of each cnt[i] while its corresponding f[i] is the maximum length.

        int n = nums.length;

        // intialize:

        int result = 0;

        int max = 0;

        int[] f = new int[n];

        int[] cnt = new int[n];

        for(int i = 0; i < n; i++) {

            f[i] = 1;

            cnt[i] = 1;

        }

        // function

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < i; j++) {

                if(nums[j] < nums[i]) {

                    // Refer to

                    // https://discuss.leetcode.com/topic/103020/java-c-simple-dp-solution-with-explanation/9

                    // f[i] == f[j] + 1 means that you find another subsequence with the

                    // same length of LIS which ends with nums[i].

                    // While f[i] > f[j] + 1 means that you find a subsequence, but its

                    // length is smaller compared to LIS which ends with nums[i]. --> so

                    // for this case we will ignore

                    if(f[i] == f[j] + 1) {

                        // Important: not ++

                        cnt[i] += cnt[j];

                    }

                    if(f[i] < f[j] + 1) {

                        f[i] = f[j] + 1;

                        cnt[i] = cnt[j];

                    }

                }

            }

            // if(max\_len == f[i]) {

            //    result += cnt[i];

            // }

            // if(max\_len < f[i]) {

            //    max\_len = f[i];

            //    result = cnt[i];

            // }

            // Refer to

            // https://discuss.leetcode.com/topic/102976/java-with-explanation-easy-to-understand

            // we can change the above section in same style as previous two question

            // https://leetcode.com/problems/longest-continuous-increasing-subsequence/description/

            // https://leetcode.com/problems/longest-increasing-subsequence/description/

            max = Math.max(max, f[i]);

        }

        for(int i = 0; i < n; i++) {

            if(max == f[i]) {

                result += cnt[i];

            }

        }

        // answer

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/number-of-longest-increasing-subsequence/solutions/1230468/c-clean-dp-solution-easy-and-explained/>

class Solution {

public:

    int findNumberOfLIS(vector<int>& nums) {

        const int n = nums.size();

        vector<int> lis(n,1);  // stores length of longest sequence till i-th position

        vector<int> count(n,1);  // stores count of longest sequence of length lis[i]

        int maxLen = 1;  // maximum length of lis

// O(N^2) DP Solution

        for(int i=1;i<n;i++){

            for(int j=0;j<i;j++){

                if(nums[i]>nums[j]){

                    if(lis[j] + 1 > lis[i]){ // strictly increasing

                        lis[i] = lis[j] + 1;

                        count[i] = count[j];

                    }

    // this means there are more subsequences of same length ending at length lis[i]

    else if(lis[j]+1 == lis[i]){

                        count[i] += count[j];

                    }

                }

            }

            maxLen = max(maxLen,lis[i]);

        }

        int numOfLIS = 0;

        // count all the subseq of length maxLen

        for(int i=0;i<n;i++){

            if(lis[i]==maxLen)

                numOfLIS += count[i];

        }

        return numOfLIS;

    }

};

**Refer to**

<https://leetcode.com/problems/number-of-longest-increasing-subsequence/solutions/4651203/python-dp-from-recursive-relation-to-memoization-to-tabulation-intuitively/>

**Iterative + memo (bottom-up) (Tabulation)**

As such, to go from recursion to iterative, we simply create a nested for loop as follows:

for i in range(len(nums)):

for j in range(i, len(nums)):

1.The outer loop represents the f(start,prev) part of the equation, such that each index i in nums gets to take "its turn" being the prev value.

2.The inner loop represents the recursion of nums[start+1], such that we iterate all values in nums that comes after the prev in the current iteration.

Once we have established the iterative order, we can simply translate the solution from step 3:

class Solution:

def findNumberOfLIS(self, nums: List[int]) -> int:

if not nums:

return 0

lengths = [1] \* len(nums)

counts = [1] \* len(nums)

for i in range(len(nums)):

for start in range(i,len(nums)):

prev = nums[i]

if nums[start] > prev:

taken\_length = lengths[i] + 1

not\_taken\_length = lengths[start]

if taken\_length > not\_taken\_length:

lengths[start] = lengths[i] + 1 #return taken\_length

counts[start] = counts[i] #return taken\_count

elif taken\_length == not\_taken\_length:

lengths[start] = lengths[i] + 1 #return taken\_length

counts[start] += counts[i] #return taken\_count + not\_taken\_count

elif taken\_length < not\_taken\_length:

lengths[start] = lengths[start] #return not\_taken\_length

counts[start] = counts[start] #return not\_taken\_count

max\_length = max(lengths)

return sum(count for length, count in zip(lengths, counts) if length == max\_length)

We let lengths[i] denote the longest increasing subsequence ending at index i.

We let counts[i] denote the count of the number of increasing subsequences of length lengths[i] ending at index i.

Simplified code:

class Solution:

def findNumberOfLIS(self, nums: List[int]) -> int:

if not nums:

return 0

n = len(nums)

lengths = [1] \* n

counts = [1] \* n

for i in range(n):

for j in range(i,n):

if nums[i] < nums[j]:

if lengths[i] + 1 > lengths[j]:

lengths[j] = lengths[i] + 1

counts[j] = counts[i]

elif lengths[i] + 1 == lengths[j]:

counts[j] += counts[i]

max\_length = max(lengths)

return sum(count for length, count in zip(lengths, counts) if length == max\_length)

**Refer to chatGPT**

Leetcode 673, "Number of Longest Increasing Subsequence," asks you to find the number of longest increasing subsequences in a given array of integers.

**Problem Outline:**

Given an integer array nums, return the number of longest increasing subsequences (LIS). An increasing subsequence is a subsequence where all elements are strictly increasing, and you need to find out how many such subsequences of the longest length exist.

**Approach:**

Dynamic Programming (DP):

We'll maintain two arrays:

dp[]**:** This will store the length of the longest increasing subsequence ending at each index.

count[]**:** This will store the number of such subsequences of length dp[i] ending at index i.

Traversal:

For each element nums[i], we will look at all the previous elements nums[j] (where j < i). If nums[i] > nums[j], it means nums[i] can extend the subsequence ending at j.

If extending nums[j] gives a longer subsequence than dp[i], we update dp[i] and set count[i] = count[j].

If extending nums[j] gives another subsequence of the same length as the current dp[i], we increment count[i] by count[j].

Result Calculation:

After filling the dp[] and count[] arrays, the longest subsequence length will be the maximum value in dp[].

The total number of such subsequences will be the sum of count[i] for all indices i where dp[i] is equal to the maximum subsequence length.

**Java Code Implementation:**

class Solution {

public int findNumberOfLIS(int[] nums) {

if (nums == null || nums.length == 0) return 0;

int n = nums.length;

int[] dp = new int[n]; // dp[i] will store the length of the LIS ending at i

int[] count = new int[n]; // count[i] will store the number of LIS ending at i

Arrays.fill(dp, 1); // Every element is an LIS of length 1

Arrays.fill(count, 1); // Every element has one way to form an LIS of length 1

int maxLen = 1; // The length of the longest increasing subsequence

for (int i = 0; i < n; i++) {

for (int j = 0; j < i; j++) {

if (nums[i] > nums[j]) { // nums[i] can extend the subsequence ending at nums[j]

if (dp[j] + 1 > dp[i]) {

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

count[i] = count[j]; // Reset count because we found a longer subsequence

} else if (dp[j] + 1 == dp[i]) {

count[i] += count[j]; // Another way to achieve the same length subsequence

}

}

}

maxLen = Math.max(maxLen, dp[i]); // Update max length of LIS found so far

}

// Now calculate how many subsequences have the maximum length

int result = 0;

for (int i = 0; i < n; i++) {

if (dp[i] == maxLen) {

result += count[i];

}

}

return result;

}

}

**Explanation:**

Arrays Initialization:

dp[] starts with all elements set to 1, meaning every element can at least be a subsequence of length 1 by itself.

count[] also starts with all elements set to 1, meaning there is one way to form a subsequence of length 1 with each element.

Nested Loops:

The outer loop (i) checks each element nums[i].

The inner loop (j) checks all elements before i (nums[j]), and if nums[i] > nums[j], it considers extending the subsequence.

Updating **dp[] and count[]:**

If we find a longer subsequence ending at i (dp[j] + 1 > dp[i]), we update dp[i] to dp[j] + 1 and set count[i] to count[j].

If we find another subsequence of the same length (dp[j] + 1 == dp[i]), we increment count[i] by count[j] to account for multiple ways to reach the same subsequence length.

Final Result:

After processing all elements, we sum up all count[i] where dp[i] is equal to the maximum LIS length (maxLen).

**Time Complexity:**

Time complexity: 𝑂(n^2) because of the nested loops where we compare each pair of elements.

Space complexity: 𝑂(n) for the dp[] and count[] arrays.

This solution efficiently computes the number of longest increasing subsequences in the array with a quadratic time complexity, which is feasible for most inputs.

**Refer to**

<https://algo.monster/liteproblems/673>

**Problem Description**

The problem we're given involves an integer array called nums. Our goal is to figure out how many subsequences of this array have the greatest length and are strictly increasing. A subsequence is defined as a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Importantly, the condition here is that our subsequence must be strictly increasing, meaning that each element must be greater than the one before it.

**Intuition**

To address the problem, we consider a dynamic programming approach that allows us to build up the solution by considering the number of increasing subsequences ending at each element.

We use two arrays, dp and cnt. The dp array will keep track of the length of the longest increasing subsequence that ends with nums[i] for each element i. The cnt array, conversely, records the total number of such subsequences of maximum length ending with nums[i].

We need to iterate over each number in the nums array and, for each element, iterate over all previous elements to check if a longer increasing subsequence can be created. When we find such an element nums[j] that is less than nums[i] (ensuring an increase), we have two cases:

If the subsequence ending at nums[j] can be extended by nums[i] to yield a longer subsequence, we update dp[i] and set cnt[i] to be the same as cnt[j] since we have identified a new maximum length.

If the extended subsequence has the same length as the current maximum subsequence ending at nums[i], we increase cnt[i] by cnt[j] because we have found an additional subsequence of this maximum length.

As we move through the array, we maintain variables to keep track of the longest subsequence found so far (maxLen) and the number of such subsequences (ans). If a new maximum length is found, maxLen is updated and ans is set to the number of such subsequences up to nums[i]. If we find additional sequences of the same maximum length, we add to ans accordingly.

At the end of the iterations, ans represents the total number of longest increasing subsequences present in the whole array nums.

**Solution Approach**

The solution uses a dynamic programming approach, which is a method for solving complex problems by breaking them down into simpler subproblems. It stores the results of these subproblems to avoid redundant work.

Here is a step-by-step breakdown of the approach:

Initialize two lists, dp and cnt, both of size n, where n is the length of the input array nums. Each element of dp starts with a value of 1, because the minimum length of an increasing subsequence is 1 (the element itself). Similarly, cnt starts with 1s because there's initially at least one way to have a subsequence of length 1 (again, the element itself).

Iterate over the array with two nested loops. The outer loop goes from i = 0 to i = n - 1, iterating through the elements of nums. The inner loop goes from j = 0 to j < i, considering elements prior to i.

Within the inner loop, compare the elements at indices j and i. If nums[i] is greater than nums[j], it means nums[i] can potentially extend an increasing subsequence ending at nums[j].

**Check if the increasing subsequence length at j plus 1 is greater than the current subsequence length at i (dp[j] + 1 > dp[i]). If it is, this means we've found a longer increasing subsequence ending at i, so update dp[i] to dp[j] + 1, and set cnt[i] to cnt[j] because the number of ways to achieve this longer subsequence at i is the same as the number at j.**

**If dp[j] + 1 equals dp[i], it indicates another subsequence of the same maximum length ending at i, so increment cnt[i] by cnt[j].**

Keep track of the overall longest sequence length found (maxLen) and the number of such sequences (ans). After processing each i, compare dp[i] with maxLen. If dp[i] is greater than maxLen, update maxLen to dp[i] and set ans to cnt[i]. If dp[i] equals maxLen, add cnt[i] to ans.

Continue this process until all elements have been considered. At the end, the variable ans will contain the number of longest increasing subsequences in nums.

By using dynamic programming, this solution effectively builds up the increasing subsequences incrementally and keeps track of their counts without having to enumerate each subsequence explicitly, which would be infeasible due to the exponential number of possible subsequences.

**Example Walkthrough**

Let's illustrate the solution approach with a small example. Suppose the given integer array nums is [3, 1, 2, 3, 1, 4].

We initialize dp and cnt arrays as [1, 1, 1, 1, 1, 1] since each element by itself is a valid subsequence of length 1 and there is at least one way to make a subsequence of one element.

Start iterating with i from index 0 to the end of the array, and for each i, we iterate with j from 0 to i-1.

**When i = 2 (element is 2), we find that nums[2] > nums[0] (2 > 3). However, as 2 is not greater than 3, no updates are done. Then we compare nums[2] with nums[1] (2 > 1), this is true, and since dp[1] + 1 > dp[2], we update dp[2] to dp[1] + 1 = 2, also updating cnt[2] to cnt[1].**

**Moving to i = 3 (element is 3), we compare with all previous elements one by one. For j = 0, nums[3] (3) is not greater than nums[0] (3). For j = 1, nums[3] (3) is greater than nums[1] (1), and dp[1] + 1 > dp[3] so we update dp[3] and cnt[3] to 2. For j = 2, nums[3] (3) is greater than nums[2] (2), and dp[2] + 1 = 3 which is greater than the current dp[3] (2). So we update dp[3] to 3, and cnt[3] to cnt[2].**

Continue this process for i = 4 and i = 5. When we reach i = 5 (element is 4) and j = 3, since nums[5] (4) is greater than nums[3] (3) and dp[3] + 1 > dp[5] (4 > 1), we update dp[5] to 4, and cnt[5] to cnt[3].

We keep track of the maxLen and ans throughout the process. Initially, maxLen is 1 and ans is 6 (because we have six elements, each a subsequence of length 1). After updating all dp[i] and cnt[i], we find maxLen to be 4 (subsequences ending at index 5) and ans will be updated to the count of the longest increasing subsequences ending with nums[5], which in this case is 1.

At the end of this process, the dp array is [1, 1, 2, 3, 1, 4] and the cnt array is [1, 1, 1, 1, 1, 1]. We conclude that the length of the longest increasing subsequence is 4 and the number of such subsequences is 1 (ending with the element 4 in the original array).

This walk-through illuminates the step-by-step application of our dynamic programming solution to find the total number of the longest increasing subsequences present in the array nums.

**Solution Implementation**

class Solution {

public int findNumberOfLIS(int[] nums) {

int maxLength = 0; // Store the length of the longest increasing subsequence

int numberOfMaxLIS = 0; // Store the count of longest increasing subsequences

int n = nums.length; // The length of the input array

// Arrays to store the length of the longest subsequence up to each element

int[] lengths = new int[n]; // dp[i] will be the length for nums[i]

int[] counts = new int[n]; // cnt[i] will be the number of LIS for nums[i]

for (int i = 0; i < n; i++) {

lengths[i] = 1; // By default, the longest subsequence at each element is 1 (the element itself)

counts[i] = 1; // By default, the count of subsequences at each element is 1

// Check all elements before the i-th element

for (int j = 0; j < i; j++) {

// If the current element can extend the increasing subsequence

if (nums[i] > nums[j]) {

// If a longer subsequence is found

if (lengths[j] + 1 > lengths[i]) {

lengths[i] = lengths[j] + 1; // Update the length for nums[i]

counts[i] = counts[j]; // Update the count for nums[i]

} else if (lengths[j] + 1 == lengths[i]) {

counts[i] += counts[j]; // If same length, add the count of subsequences from nums[j]

}

}

}

// If a new maximum length of subsequence is found

if (lengths[i] > maxLength) {

maxLength = lengths[i]; // Update the maxLength with the new maximum

numberOfMaxLIS = counts[i]; // Reset the count of LIS

} else if (lengths[i] == maxLength) {

// If same length subsequences are found, add the count to the total

numberOfMaxLIS += counts[i];

}

}

// Return the total count of longest increasing subsequences

return numberOfMaxLIS;

}

}

**Time and Space Complexity**

The given Python function findNumberOfLIS calculates the number of Longest Increasing Subsequences (LIS) in an array of integers. Analyzing the time complexity involves examining the nested loop structure, where the outer loop runs n times (once for each element in the input list nums) and the inner loop runs at most i times for the ith iteration of the outer loop. Consequently, in the worst case, the inner loop executes 1 + 2 + 3 + ... + (n - 1) times, which can be expressed as the sum of the first n-1 natural numbers, resulting in a time complexity of O(n^2).

The space complexity of the function is determined by the additional memory allocated for the dynamic programming arrays dp and cnt, each of size n. Therefore, the space complexity of the function is O(n), which is linear with respect to the size of the input list.

**Refer to**

[L300.Longest Increasing Subsequence](note://20C954956F6E4AA6A3575EA3146A1FCA)