# Longest Increasing Sequence

There is a special type of sort problem which is longest increasing sequence. The typical problem asks, given an array, if you scan from left to right what is the longest increasing sequence you will get. The typical solution is that you store the numbers in increasing order, when a new number comes you will use binary search to find the position where the number is equal or great than the current number and use the current number to replace it, you may end up adding the current number at the end of the array. The index of the current number indicates the longest sequence which is less than the current number.

## 300. Longest Increasing Subsequence

Medium

Given an unsorted array of integers, find the length of longest increasing subsequence.

**Example:**

**Input:** [10,9,2,5,3,7,101,18]

**Output:** 4

**Explanation:** The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

**Note:**

* There may be more than one LIS combination, it is only necessary for you to return the length.
* Your algorithm should run in O(*n2*) complexity.

**Follow up:** Could you improve it to O(*n* log *n*) time complexity?

### Analysis:

This is the standard LIS problem. You need to keep track on the longest increasing sequence and use binary search to locate the right position for each number. Use the smaller ones to replace the bigger ones, so keep the LIS growing.

/// <summary>

/// Leet code #300. Longest Increasing Subsequence

///

/// Given an unsorted array of integers, find the length of longest

/// increasing subsequence.

///

/// Example:

///

/// Input: [10,9,2,5,3,7,101,18]

/// Output: 4

/// Explanation: The longest increasing subsequence is [2,3,7,101],

/// therefore the length is 4.

/// Note:

///

/// There may be more than one LIS combination, it is only necessary for

/// you to return the length.

/// Your algorithm should run in O(n2) complexity.

/// Follow up: Could you improve it to O(n log n) time complexity?

/// </summary>

int LeetCodeSort::lengthOfLIS(vector<int>& nums)

{

vector<int> result;

for (int num : nums)

{

vector<int>::iterator iterator =

lower\_bound(result.begin(), result.end(), num);

if (iterator == result.end())

{

result.push\_back(num);

}

else

{

\*iterator = num;

}

}

return result.size();

}

## 673. Number of Longest Increasing Subsequence

Medium

Given an unsorted array of integers, find the number of longest increasing subsequence.

**Example 1:**

**Input:** [1,3,5,4,7]

**Output:** 2

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

**Example 2:**

**Input:** [2,2,2,2,2]

**Output:** 5

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

**Note:** Length of the given array will be not exceed 2000 and the answer is guaranteed to be fit in 32-bit signed int.

### Analysis:

We keep track a longest increasing sequence, on every position, we record all the possible numbers with the path count reaching them. When a new number comes and find its position, simply count all the previous path which is less than the current number and add them up as the total number of paths.

/// <summary>

/// Leet code #673. Number of Longest Increasing Subsequence

///

/// Given an unsorted array of integers, find the number of longest

/// increasing subsequence.

///

/// Example 1:

/// Input: [1,3,5,4,7]

/// Output: 2

/// Explanation: The two longest increasing subsequence are [1, 3, 4, 7]

/// and [1, 3, 5, 7].

///

/// Example 2:

/// Input: [2,2,2,2,2]

/// Output: 5

/// Explanation: The length of longest continuous increasing subsequence

/// is 1, and there are 5 subsequences' length is 1, so output 5.

///

/// Note: Length of the given array will be not exceed 2000 and the answer

/// is guaranteed to be fit in 32-bit signed int.

/// </summary>

int LeetCodeSort::findNumberOfLIS(vector<int>& nums)

{

vector<int> seq;

vector<map<int, int>> dp;

// first we build the value from nums into a sequence,

for (size\_t i = 0; i < nums.size(); i++)

{

int j = lower\_bound(seq.begin(), seq.end(), nums[i]) - seq.begin();

if (j < (int)seq.size())

{

seq[j] = nums[i];

}

else

{

seq.push\_back(nums[i]);

dp.push\_back(map<int, int>());

}

// calculate dp value based on current count on seq length - 1;

if (j == 0) dp[j][nums[i]] += 1;

else

{

for (auto itr : dp[j - 1])

{

if (itr.first < nums[i]) dp[j][nums[i]] += itr.second;

}

}

}

int result = 0;

if (!dp.empty())

{

for (auto itr : dp[dp.size() - 1])

{

result += itr.second;

}

}

return result;

}

## 674. Longest Continuous Increasing Subsequence

Easy

Given an unsorted array of integers, find the length of longest continuous increasing subsequence (subarray).

**Example 1:**

**Input:** [1,3,5,4,7]

**Output:** 3

**Explanation:** The longest continuous increasing subsequence is [1,3,5], its length is 3.

Even though [1,3,5,7] is also an increasing subsequence, it's not a continuous one where 5 and 7 are separated by 4.

**Example 2:**

**Input:** [2,2,2,2,2]

**Output:** 1

**Explanation:** The longest continuous increasing subsequence is [2], its length is 1.

**Note:** Length of the array will not exceed 10,000.

### Analysis:

This one you do not need to use LIS, just greedily track longest sequence.

/// <summary>

/// Leet code #674. Longest Continuous Increasing Subsequence

///

/// Given an unsorted array of integers, find the length of longest

/// continuous increasing subsequence.

///

/// Example 1:

/// Input: [1,3,5,4,7]

/// Output: 3

/// Explanation: The longest continuous increasing subsequence is [1,3,5],

/// its length is 3.

/// Even though [1,3,5,7] is also an increasing subsequence, it's not a

/// continuous one where 5 and 7 are separated by 4.

///

/// Example 2:

/// Input: [2,2,2,2,2]

/// Output: 1

/// Explanation: The longest continuous increasing subsequence is [2],

/// its length is 1.

///

/// Note: Length of the array will not exceed 10,000.

/// </summary>

int LeetCodeSort::findLengthOfLCIS(vector<int>& nums)

{

int count = 0, result = 0;

for (size\_t i = 0; i < nums.size(); i++)

{

if ((i == 0) || (nums[i] > nums[i - 1]))

{

count++;

}

else

{

result = max(result, count);

count = 1;

}

}

result = max(result, count);

return result;

}

## 354. Russian Doll Envelopes

Hard

You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

**Note:**  
Rotation is not allowed.

**Example:**

**Input:** [[5,4],[6,4],[6,7],[2,3]]

**Output:** 3

**Explanation: T**he maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

### Analysis:

We sort the envelopes by width from low to high, then for same width, search the heights from high to low. This is because to put an envelop into another we need the width and height for the following one bigger than current one.

Then we use LIS to get the longest increasing height. The length of the longest increasing sequence on height is answer.

/// <summary>

/// Leet code #354. Russian Doll Envelopes

///

/// You have a number of envelopes with widths and heights given as a

/// pair of integers (w, h). One envelope can fit into another if and

/// only if both the width and height of one envelope is greater than

/// the width and height of the other envelope.

///

/// What is the maximum number of envelopes can you Russian doll?

/// (put one inside other)

///

/// Note:

/// Rotation is not allowed.

///

/// Example:

///

/// Input: [[5,4],[6,4],[6,7],[2,3]]

/// Output: 3

/// Explanation: The maximum number of envelopes you can Russian doll

/// is 3 ([2,3] => [5,4] => [6,7]).

/// </summary>

int LeetCodeSort::maxEnvelopes(vector<pair<int, int>>& envelopes)

{

vector<int> collector;

struct envelope\_order

{

bool operator() (pair<int, int> a, pair<int, int> b)

{

return ((a.first < b.first) || ((a.first == b.first) && (a.second > b.second)));

}

};

std::sort(envelopes.begin(), envelopes.end(), envelope\_order());

for (pair<int, int>envelope : envelopes)

{

vector<int>::iterator iterator =

lower\_bound(collector.begin(), collector.end(), envelope.second);

if (iterator == collector.end())

{

collector.push\_back(envelope.second);

}

else if (\*iterator > envelope.second)

{

\*iterator = envelope.second;

}

}

return (int)collector.size();

}