Leetcode\_300\_LongestIncreasingSubsequence\_最长上升子序列\_Medium\_Hard

# Leetcode\_300\_LongestIncreasingSubsequence\_最长上升子序列\_Medium\_Hard

## 题目介绍

\* 难度：Medium/Hard -->O(n^2)/O(n\*logn)

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

\* 题目分析：

\* 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

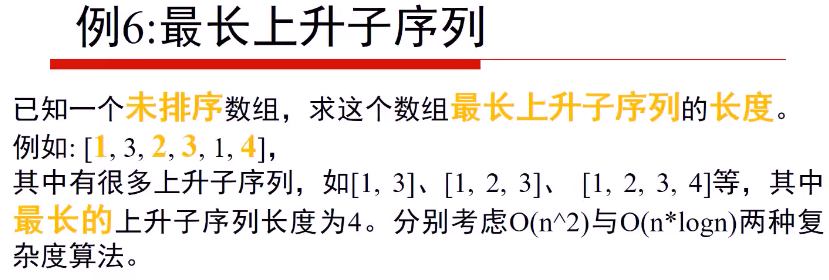
\* Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

\* 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?



## 思路分析

\* 思路分析：

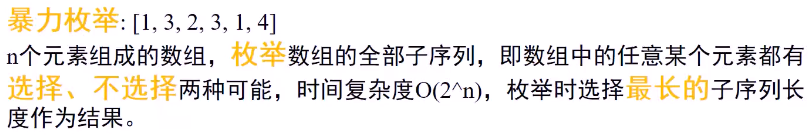
\* 1. O(n^2)复杂度:动态规划思想。

\* 状态转移方程：dp[i] = max{dp[k]}+1; 条件：nums[i]>nums[k],k = i-1,...,0;

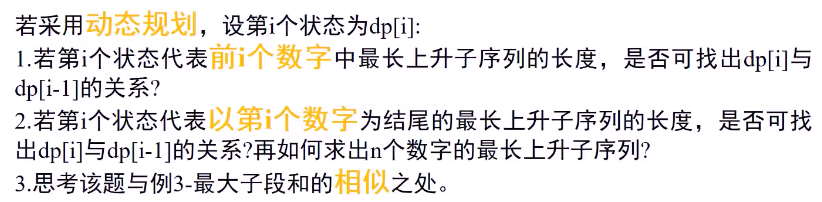
\* dp[i] = max{dp[k]}; 条件：nums[i]=nums[k],k = i-1,...,0;

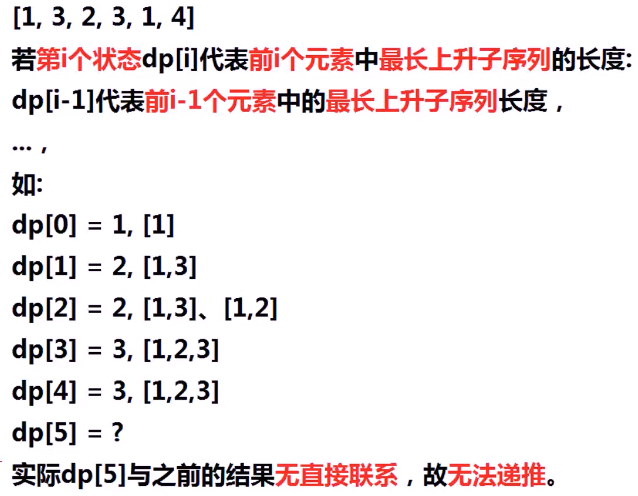
\* 初始化：dp[i] = 1;

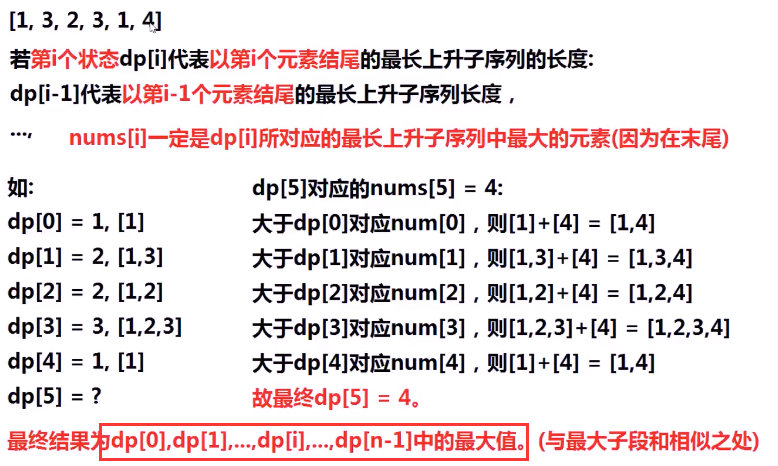
### 暴力枚举

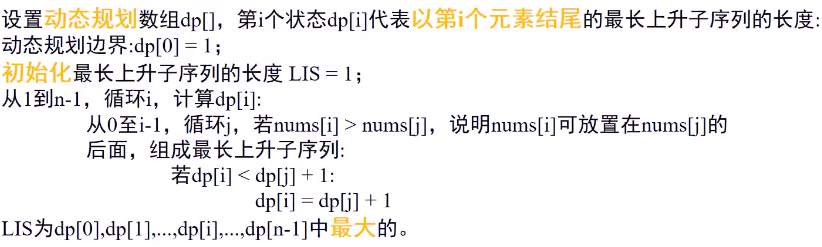


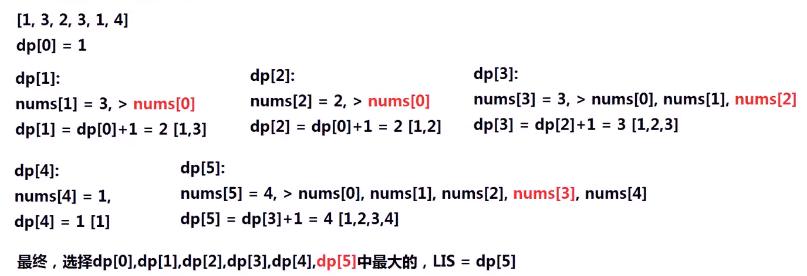
### 动态规划











### O(nlogn)算法

**\* 2. O(nlogn)复杂度：基于二分查找算法。**

**\* 利用一个数组容器dp存放最长上升子序列长度为i的末尾数字num；**

**\* 下一个数字元素x:若x大于容器顶部数字元素，则直接push到顶部；**

**\* 否则寻找恰好大于等于该元素的值；由于从底部至顶部已经是排序的数字，**

**\* 因此借助二分查找就可以快速查找到需要替换的元素值。**

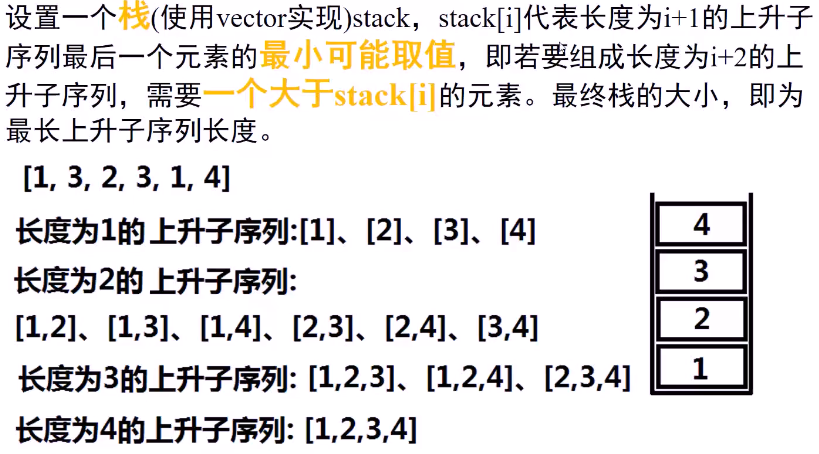
**\* 最终，前面存放数值的长度就是最长子序列的长度。**

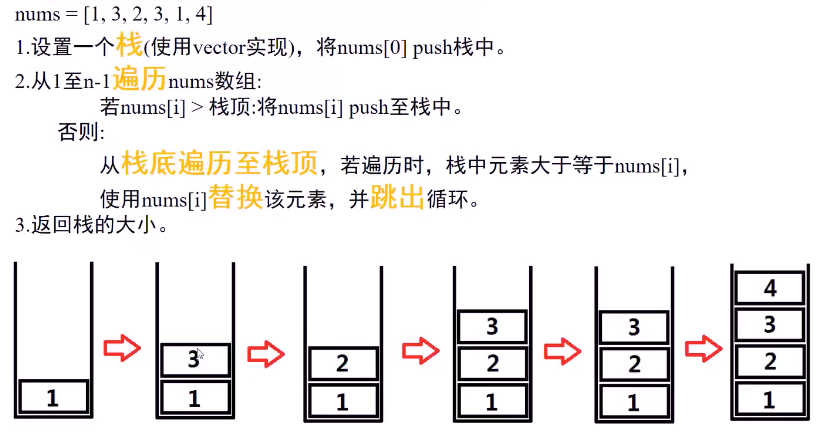
**\* 复杂度分析：最外层为O(N);**

**\* 但是对于新元素大于顶部元素的，复杂度是O(1);**

**\* 新元素小于顶部元素的，复杂度就是O(NlogN);**

**\* 因此，最终复杂度为O(NlogN).**





## Java代码

### 动态规划O(n^2)

public int **lengthOfLIS**(int[] nums) {

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

if(nums.length == 1) return 1;

//开辟dp数组，存放以nums[i]结尾的上升子序列的长度，初始化为1

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

**for(int i = 0;i < nums.length;i++) dp[i] = 1;**

int maxLen = dp[0];

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

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

//dp[i]已经大于dp[j]了，直接continue；

//这里不能包括等号，因为dp[i]=dp[j]时若nums[i]>nums[j]，仍然需要更新,dp[i]=dp[j]+1;

if(dp[i] > dp[j]) continue;

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

else if(nums[i] == nums[j]) dp[i] = dp[j];

}

//更新最大长度

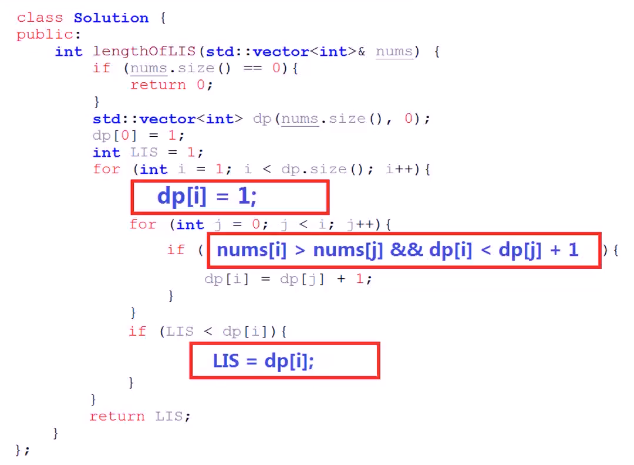
if(dp[i] > maxLen)

maxLen = dp[i];

}

return maxLen;

}



### O(NlogN)算法：基于二分查找

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*方法一\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* 动态规划思路

\* 复杂度为O(n^2)

\*/

public int lengthOfLIS(int[] nums) {

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

if(nums.length == 1) return 1;

//开辟dp数组，存放以nums[i]结尾的上升子序列的长度，初始化为1

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

for(int i = 0;i < nums.length;i++) dp[i] = 1;

int maxLen = dp[0];

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

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

//dp[i]已经大于dp[j]了，直接continue；

//这里不能包括等号，因为dp[i]=dp[j]时若nums[i]>nums[j]，仍然需要更新,dp[i]=dp[j]+1;

if(dp[i] > dp[j]) continue;

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

else if(nums[i] == nums[j]) dp[i] = dp[j];

}

//更新最大长度

if(dp[i] > maxLen)

maxLen = dp[i];

}

return maxLen;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*方法二\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* 采用栈结构

\* 复杂度为O(NlogN)

\*/

public int lengthOfLIS2(int[] nums) {

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

if(nums.length == 1) return 1;

//开辟dp数组，存放以nums[i]结尾的上升子序列的长度，初始化为1

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

dp[0] = nums[0];

int dpLen = 1;

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

if(nums[i] > dp[dpLen-1]){

dp[dpLen] = nums[i];

dpLen++;

}else{

//用二分查找到刚好大于等于nums[i]的元素的索引，用替换掉

int index = binarySearch(dp,0,dpLen-1,nums[i]);

dp[index] = nums[i];

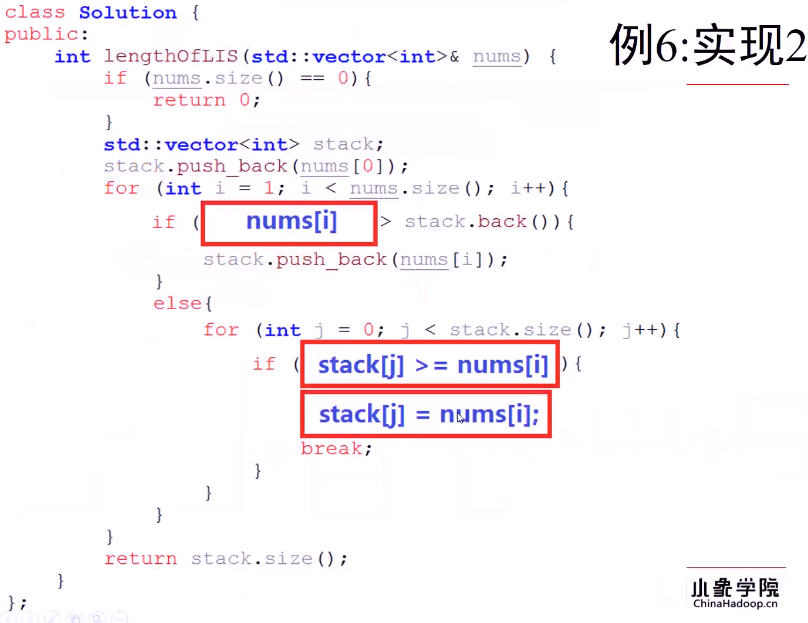
}

}

return dpLen;

}

未利用二分查找遍历查询待替换的元素：



利用二分查找优化：

