11. Container With Most Water

这道题标示为中等。

方法一：暴力破解

但是时间复杂度为O(n​2​​)，空间复杂度为O(1)。

因为时间复杂度过大，导致程序最终报了[Time Limit Exceeded]

/\*\*

\* @param {number[]} height

\* @return {number}

\*/

var maxArea = function(height) {

let contain;

let max=0;

for(let i=0;i<height.length-1;i++){

for(let j=i+1;j<height.length;j++){

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

contain=(j-i)\*height[i];

}else{

contain=(j-i)\*height[j];

}

if(contain>max){

max=contain;

}

}

}

return max;

};

方法二：两点接近法

这种方法一开始写出来只是凭一种直觉，但是最后被别人给证明出来了，证明会在后面详写。

这种方法背后的直觉是：线之间形成的区域总是受到短线高度的限制，而且两条线相隔越大，容积也会越大。

我们选取两个指针，一个指向数组的最开始，一个指向数组的末尾。在接下来的每一步中，我们选取指针值较小的那个指针，然后将它向中间移动一个。

在这样的过程中，就会找到容积最大的两个点。

证明：

假设返回的结果不是最优解，那么一定存在一个最优解，假设当两个指针同时指向ol和or（一个左一个右）时，它们之间的容积最大。所以我们的算法能成功的关键在于能够让这两个点同时被两个指针所指向。因为我们在算法结束的时候会遍历一遍数组，所以肯定能遍历到ol和or，但是否同时能遍历到，这就需要我们证明了。

假设我们遍历到ol而不是or，当一个指针停在ol，它不会移动，直到：

1. 另一个指针最终也指向ol；

在这种情况下，迭代结束，但另一个指针一定指向过or。

1. 另一个指针到达rr，但还没有到达or时（rr比ol大，所以下一步，指针将离开ol）

在这种情况下，我们移动了ol，但是注意ol和rr会比ol和or大（因为无论高度还是宽度ol和or都比不上ol和rr），所以ol和rr才是最优解。

/\*\*

\* @param {number[]} height

\* @return {number}

\*/

var maxArea = function(height) {

let i=0;

let j=height.length-1;

let max=0;

let count;

while(i<j){

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

count=(j-i)\*height[i];

i++;

}else{

count=(j-i)\*height[j];

j--;

}

if(count>max){

max=count;

}

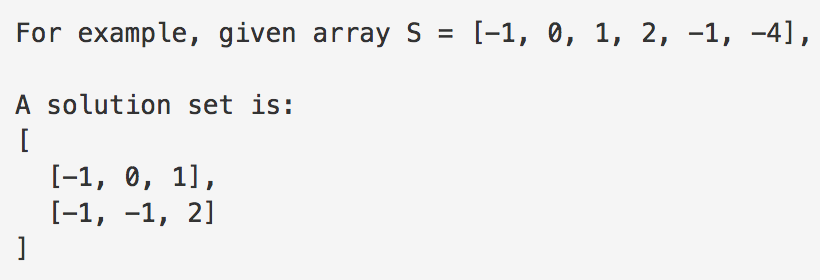
}

return max

};

15. 3Sum

要求：给了一个有n个元素的数组S，算出在S中有多少个a,b,c组合，可以使a + b + c = 0?找出S中所有的满足这样条件的不重复组合。



思路一：

这是我的思路，这个思路的产生源于611题，求在一个数组有多少种组合可以组成三角形三边。

我先将数组排序，然后设置i、j、k三个指针，i、j从前面遍历，k从最后开始遍历，并且在i固定，遍历j的情况下，k也只会遍历这一遍。同时还要注意最后得到的数组不能有重复项，所以我使用的方法是从i、j、k三个方面都进行了考虑：1、针对i是如果i不为0，且nums[i]===nums[i-1]，则continue；

2、针对j是如果j的前面一个位置不是i，且nums[j]===nums[j-1]，则continue；

1. 针对k是如果找到一个k使nums[i]+nums[j]+nums[k]===0，则不再继续下去。

/\*\*

\* @param {number[]} nums

\* @return {number[][]}

\*/

var threeSum = function(nums) {

let result=[];

let len=nums.length;

nums.sort((a,b) => {

return a-b;

});

for(let i=0;i<len-2;i++){

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

continue;

}

let k=len-1;

for(let j=i+1;j<len-1;j++){

if((j-1)!==i&&nums[j]===nums[j-1]){

continue;

}

while(k>j&&nums[i]+nums[j]+nums[k]>0){

k--;

}

if(k>j&&nums[i]+nums[j]+nums[k]===0){

let array=[nums[i],nums[j],nums[k]];

result.push(array);

}

}

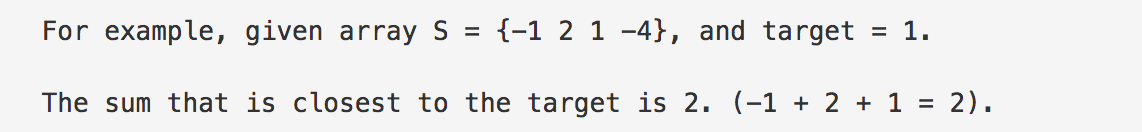
}

return result;

};

16. 3Sum Closest

要求：给一个长度为n的整数数组S，从S中找到三个整数，使它们加起来的总数最接近一个给定的值，返回这三个整数的总数。



这道题标识为middle。

这道题使用了一种复杂度为O(n^2)的方法。

先将数组排序。然后标记i、j和k。在一次i循环中，j从i+1开始，k从nums.length-1开始，如果nums[i]+nums[j]+nums[k]-target取绝对值小于当前存储的closestSum，则将当前的nums[i]+nums[j]+nums[k]赋值给closestSum。如果nums[i]+nums[j]+nums[k]>target，则将k--，来减小nums[k]的值;如果nums[i]+nums[j]+nums[k]<=target，则将j++，来增加nums[j]的值。

/\*\*

\* @param {number[]} nums

\* @param {number} target

\* @return {number}

\*/

var threeSumClosest = function(nums, target) {

nums.sort((a,b) => {

return a-b;

})

let closestSum = nums[0]+nums[1]+nums[2];

let curSum;

for(let i=0;i<nums.length-2;i++){

j=i+1;

k=nums.length-1;

while(j<k){

curSum=nums[i]+nums[j]+nums[k];

if(Math.abs(closestSum-target)>Math.abs(curSum-target)){

closestSum=curSum;

}

target>curSum?j++:k--;

}

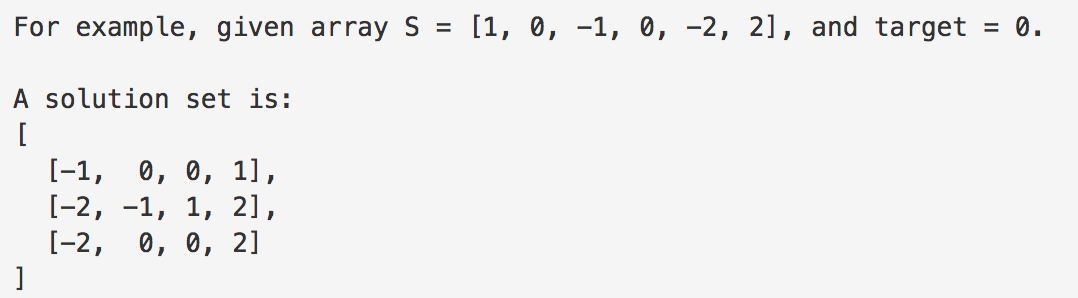
}

return closestSum;

};

18. 4Sum

要求：给出一个有n个长度的数组S，找出数组中所有4个元素a、b、c、d的组合，使a+b+c+d=target，找出所有不重复的组合。比如：



这道题标识为middle。

这道题我没做出来。

这道题目的思路和3Sum差不多，先将数组排序，再定义两个指针i和j，然后用两个指针low和high指向剩下数组的头和尾，根据nums[a]+nums[b]+nums[c]+nums[d]和target的大小关系，移动low或high指针。

这样的时间复杂度为O(n^3)，但是这样做会爆出超时错误，所以还需要排除一些情况，减小时间复杂度。

1. 排除重复的情况；
2. 排除从nums[i]+nums[i+1]+nums[i+2]+nums[i+3]>target的情况；
3. 排除nums[i]+nums[len-3]+nums[len-2]+nums[len-1]<target的情况。

还要注意的点是，如果有多个if判断条件，一定要把break的放在第一个。

/\*\*

\* @param {number[]} nums

\* @param {number} target

\* @return {number[][]}

\*/

var fourSum = function(nums, target) {

let len=nums.length;

let low,high,sum,result=[];

nums.sort((a,b)=>{

return a-b;

})

if(len<4){

return result;

}

for(let i=0;i<len-3;i++){

if(nums[i]+nums[i+1]+nums[i+2]+nums[i+3]>target){break;}

if(nums[i]+nums[len-1]+nums[len-2]+nums[len-3]<target){continue;}

if(i>0&&nums[i-1]===nums[i]){continue;} //因为在nums[i-1]时已经将所有情况都列出来了，如果nums[i]===nums[i-1]，nums[i]不会再有更新的情况出现了

for(let j=i+1;j<len-2;j++){

if(nums[i]+nums[j]+nums[j+1]+nums[j+2]>target){break;}

if(nums[i]+nums[j]+nums[len-1]+nums[len-2]<target){continue;}

if(j>i+1&&nums[j-1]===nums[j]){continue;}

low=j+1;

high=len-1;

while(low<high){

sum=nums[i]+nums[j]+nums[low]+nums[high];

if(sum===target){

result.push([nums[i],nums[j],nums[low],nums[high]]);

while(low<high&&nums[low]===nums[low+1]){low++;}

while(low<high&&nums[high]===nums[high-1]){high--;}

low++;

high--;

}else if(sum>target){

high--;

}else{

low++;

}

}

}

}

return result;

};

26. Remove Duplicates from Sorted Array

要求：给定一个排序的数组，删除重复的位置，使每个元素只显示一次并返回新的长度。

不要为另一个数组分配额外的空间，您必须使用常量内存来执行此操作。

例如，

给定输入数组nums = [1,1,2]，你的函数应该返回length = 2，num的前两个元素分别为1和2。

这道题标识为easy。

这道题一开始理解错题目意思，感觉返回的数组而不是数组长度，再细细读题，才发现，我return一个长度，然后后台会检查这个长度内的数组元素是否符合要求。

这道题要解决的问题是删去数组中的重复元素，我在答案中看到一个人的解法觉得很不错，特在此记录。

先定义了一个count，用来记录数组中重复项的总数。每次出现重复项，count就加一，因此count记录的是到当前为止出现的重复项总数，当出现一个非重复项时，就将它移动到i-count的位置。

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var removeDuplicates = function(nums) {

let count=0;

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

if(nums[i]===nums[i-1]){

count++

}else{

nums[i-count]=nums[i];

}

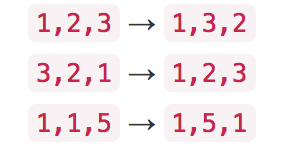
}

return nums.length-count;

};

31. Next Permutation

要求：给出一个数组，找出比这个数组组成的整数大的数组排序，如果没有则返回最小的数组排序。比如：



这道题标识为middle。

这道题的思路如下:从最后向前找，找到第一个小于后一个数的数字nums[i]比如：13954，3就是第一个小于后一个数的数字。找到之后，再从后开始找出第一个大于nums[i]的数字，记为nums[tmp]，例子中就是4。然后交换这两个数字，即3和4，此时为14953。最后将nums[i]（4）之后的数字逆置（这样才能取到最小值），即14359。

代码如下：

/\*\*

\* @param {number[]} nums

\* @return {void} Do not return anything, modify nums in-place instead.

\*/

var nextPermutation = function(nums) {

let len=nums.length;

let i=len-2;

let tmp=len-1;

if(len===1){

return;

}

//找到第一个小于后一个数的数字

while(i>=0&&nums[i]>=nums[i+1]){

i--

}

if(i>=0){

//再从后开始找出第一个大于nums[i]的数字:nums[tmp]

while(i<=tmp){

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

break;

}

tmp--;

}

//交换min和tmp

swap(nums,tmp,i);

//将nums[i]后面的数逆置

reverse(nums,i+1,len-1);

}else{

reverse(nums,0,len-1);

}

return;

};

function swap(arr,i,j){

let tmp;

tmp = arr[i];

arr[i]=arr[j];

arr[j]=tmp;

}

//逆置

function reverse(arr,start,end){

let tmp;

if(start>end){

return

}

for(let i=start;i<=(end+start)/2;i++){

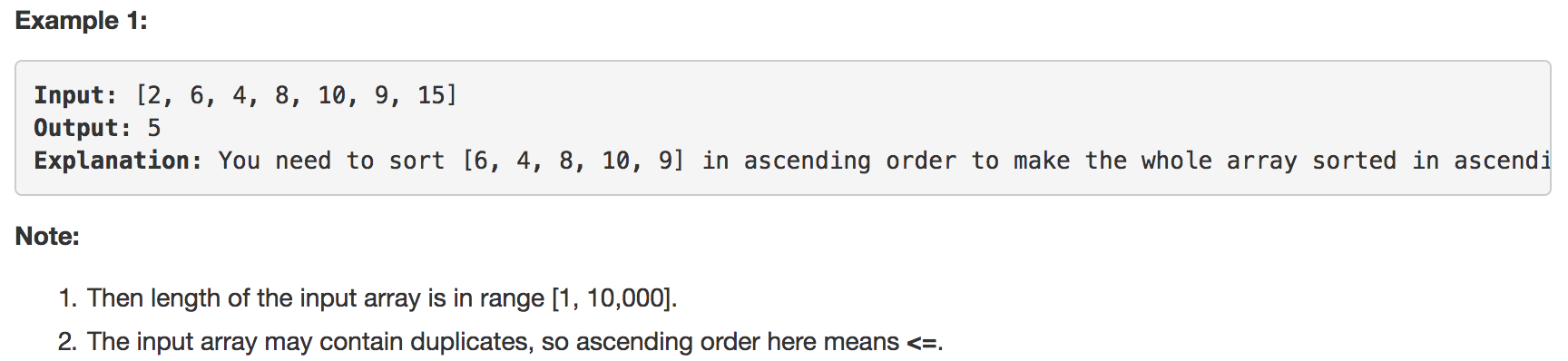
swap(arr,i,start+end-i);

}

}

581. Shortest Unsorted Continuous Subarray

要求：给了一个整数数组，我们需要找到一个子数组，当排序好子数组，原数组就也会被排序好。返回子数组的长度。



这道题标示为简单。

这道题我的思路是只要将原数组排序生成一个新的数组，然后比较原数组与新数组，记录下第一个元素不同的位置和最后一个元素不同的位置。将这两个相减加一，就得到字数组的长度。

代码如下：

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var findUnsortedSubarray = function(nums) {

let len=nums.length;

let sorted=nums.slice(0,len);

let tmp=[];

sorted.sort((a,b)=>a-b);

let start=0;

let end=len-1

while(start<len&&sorted[start]===nums[start]){start++;}

while(end>start&&sorted[end]===nums[end]){end--;}

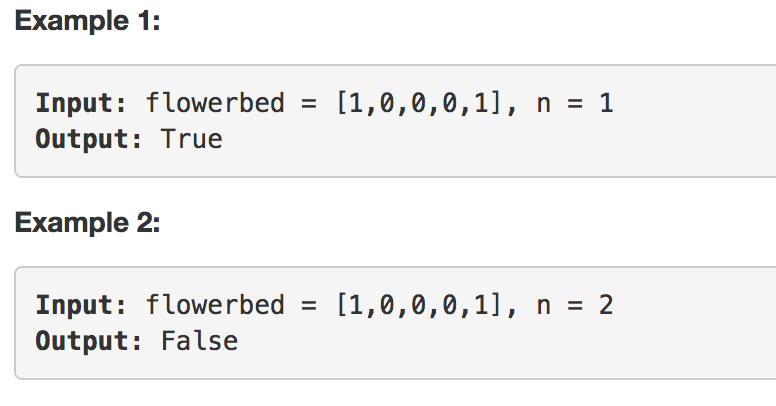
return end-start+1;

};

605. Can Place Flowers

要求：假设你有一个很长的花坛，一部分种了花，一部分空着。然而，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个包含0和1的数组，其中0是指空，1指不空），和一个数N，N返回在不违反规则的情况下可以种植花的个数。如：



思路一：将其分为两个情况：数组中有1的情况和没有1的情况。当数组有1时，需要把三个可以插花的部分都算上：第一个1出现之前可以插花的个数、两个1

之间可以插花的个数、最后一个1到最后位置之间可以插花的个数。（这种方法很容易考虑不完全，不好）

代码如下：

/\*\*

\* @param {number[]} flowerbed

\* @param {number} n

\* @return {boolean}

\*/

var canPlaceFlowers = function(flowerbed, n) {

let flag;

let count=0;

let count0=0;

for(let i=0;i<flowerbed.length;i++){

if(flowerbed[i]===1){

if(flag===undefined){

count+=Math.floor(i/2);

}else{

count+=Math.floor((i-flag-2)/2);

}

flag=i;

}else{

count0++;

}

}

if(flowerbed.length-1-flag){

count+=Math.floor((flowerbed.length-1-flag)/2)

}

if(flag===undefined){

if(count0===flowerbed.length){

count+=Math.floor((count0+1)/2);

}else{

return false;

}

}

if(count>=n){

return true;

}else{

return false;

}

};

思路二：  
一个标识为0的位置，如果前后都不为1，则这个位置可以插入1，计算这样位置的个数。

代码如下：

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {number}

\*/

var findSecondMinimumValue = function(root) {

if(root.right.val>root.val){

return root.right.val;

}else if(root.left.val>root.val){

return root.left.val;

}else{

return -1;

}

};

611. Valid Triangle Number

方法一：暴力破解（但是超时了）

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var triangleNumber = function(nums) {

let max;

let min;

let count=0;

for(let i=0;i<nums.length-2;i++){

for(let j=i+1;j<nums.length-1;j++){

max=nums[i]+nums[j];

min=Math.abs(nums[i]-nums[j]);

for(let k=j+1;k<nums.length;k++){

if(nums[k]>min&&nums[k]<max){

count++;

}

}

}

}

return count;

};

方法二：二分查找

先将数组排序，然后标记i和j，我们要做的是找到第一个大于等于nums[i]+nums[j]的数nums[k]，然后在这次查找中可以使nums[i]、nums[j]和某数称为三角形三边的个数为k-j-1个。在这种方法中使用了二分查找法，关于二分查找法，我会另外写一篇文章。

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var triangleNumber = function(nums) {

let left;

let right;

let count=0;

nums.sort((a,b) => {

return a-b;

})

for(let i=0;i<nums.length-2;i++){

for(let j=i+1;j<nums.length-1;j++){

left=j+1;

right=nums.length-1

while(left<=right){

let mid=Math.floor((right+left)/2)

if(nums[mid]>=(nums[i]+nums[j])){

right=mid-1;

}else{

left=mid+1;

}

}

count+=left-j-1;

}

}

return count;

};

方法三：线性搜索

也是先将数组排序。

这种方法其主要作用的就是k=i+2，这样在一次i循环中，增加j的值时，k还从上一次找到的地方向后继续寻找，因为在前一次找到的k，使nums[k] 第一个大于等于nums[i]+nums[j]的数，在增加j之后，nums[i]+nums[j]变大，所以这次第一个大于等于nums[i]+nums[j]的数的位置，肯定在上一次的k之后。

还要注意0的情况，因为三角形边长不能为0，所以需要排除。

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var triangleNumber = function(nums) {

let count=0;

let k;

nums.sort((a,b) => {

return a-b;

})

for(let i=0; i<nums.length-2;i++){

k=i+2

for(let j=i+1;j<nums.length-1&& nums[i] != 0;j++){

while(nums[k]<nums[i]+nums[j]&&k<nums.length){

k++

}

count+=k-j-1

}

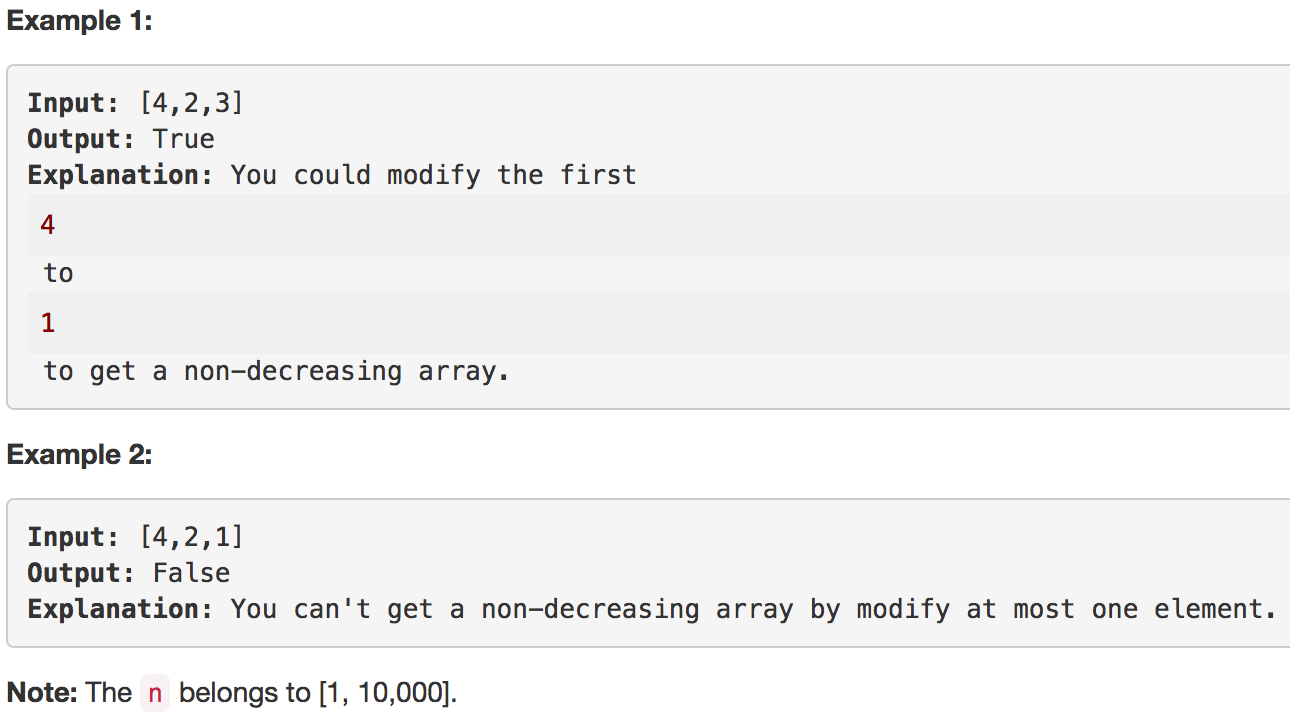
}

return count;

};

665. Non-decreasing Array

要求：给了一个包含n个整数的数组，我们要判断这个数组，是否可以通过修改至多一个元素，使之成为一个非递减数组。比如：



这道题标识为简单。

这道题我想了很久，一开始的想法是遍历数组，找出所有前一个元素大于后一个元素的情况，如果这种情况count大于1，则说明需要修改两个以上的元素，才能成为一个非递减数组。但是这种想法并不全面，

当a[i]>a[i+1]时，调整起来只有两种情况：一、减小a[i]，使a[i]=a[i+1]。但这时候因为减小了a[i]，所以有可能会出现a[i-1]>a[i](=a[i+1])的情况，再次形成递减序列的特殊情况，如[3,4,2]；二、增加a[i+1]，使a[i+1]=a[i]，但因为增加了a[i+1]，有可能出现a[i+1](=a[i])>a[i+2]的特殊情况，如[4,2,3]。

当情况一出现时，可以用情况二来解决；当情况二出现时，可以用情况一来解决。当两种情况同时出现，才不行。

代码如下：

/\*\*

\* @param {number[]} nums

\* @return {boolean}

\*/

var checkPossibility = function(nums) {

let len=nums.length;

let count=0;

for(let i=0;i<len;i++){

if(nums[i]>nums[i+1]){

if(nums[i-1]>nums[i+1]&&nums[i]>nums[i+2]){

return false;

}

count++;

if(count>1){

return false;

}

}

}

return true;

};

674. Longest Continuous Increasing Subsequence

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var findLengthOfLCIS = function(nums) {

let start=0;

let end=0;

let count=0;

if(nums.length===1){

return 1;

}

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

if(nums[i-1]<nums[i]){

end=i;

if(count<end-start+1){

count=end-start+1;

}

}else if(nums[i-1]>=nums[i]){

if(count<end-start+1){

count=end-start+1;

}

start=i;

end=i-1;

}

}

return count;

};