2019-11 IEEEBIBM 2019 Award Committee Chairs Best Student Paper

//binary-tree-level-order-traversal   
/\*\*2   
 \* Definition for a binary tree node.   
 \* struct TreeNode {   
 \* int val;   
 \* TreeNode \*left;   
 \* TreeNode \*right;   
 \* TreeNode(int x) : val(x), left(NULL), right(NULL) {}   
 \* };   
 \*/   
class Solution {   
public:   
 vector<vector<int>> levelOrder(TreeNode\* root) {   
 if(root==NULL)   
 return {};   
 else   
 {   
 queue<TreeNode\*> q;   
 q.push(root);   
 vector<int> inter;   
 vector<vector<int>> result;   
 while(q.size()!=NULL)   
 {   
 int count=q.size();   
 while(count>0)   
 { TreeNode\* curr=q.front();   
 q.pop();   
 inter.push\_back(curr->val);   
 if(curr->left!=NULL)   
 q.push(curr->left);   
 if(curr->right!=NULL)   
 q.push(curr->right);   
 count--;   
 }   
 result.push\_back(inter);   
 inter={};   
 }   
 return result;   
 }   
 }   
};   
   
   
\* https://leetcode-cn.com/problems/binary-tree-level-order-traversal-ii   
\* 题目描述：   
\* 给定一个二叉树，返回其节点值自底向上的层次遍历。   
\* 即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历   
\*   
\* 例如：   
\* 给定二叉树 [3,9,20,null,null,15,7]   
\* 3   
\* / \   
\* 9 20   
\* / \   
\* 15 7   
\* 返回其自底向上的层次遍历为：   
\* [   
\* [15,7],   
\* [9,20],   
\* [3]   
\* ]   
\*   
\* 解题思路：   
\* 由于需要按层进行组成，且是vector<vector<int>>   
\* 所以必须把每个层分开记录，另一点就是还要实现逆序   
\* 分层可以通过单独记录每层节点来完成，而逆序有两种方式   
\* 第一种是直接正序记录，然后再颠倒顺序   
\* 第二种是进行递归记录每层节点，最后在每层中添加数据   
\* 这样第二种就省去了第一种方法中颠倒顺序这步的操作   
\*/   
/\*\*   
 \* Definition for a binary tree node.   
 \* struct TreeNode {   
 \* int val;   
 \* TreeNode \*left;   
 \* TreeNode \*right;   
 \* TreeNode(int x) : val(x), left(NULL), right(NULL) {}   
 \* };   
 \*/   
class Solution {   
public:   
 vector<vector<int>> levelOrderBottom(TreeNode\* root) {   
 if (root == NULL)   
 return {};   
   
 vector<vector<int>> res;   
 vector<TreeNode \*> nodes;   
 nodes.push\_back(root);   
 do\_order(nodes, res);   
   
 return res;   
 }   
   
 void do\_order(vector<TreeNode \*> &nodes, vector<vector<int>> &res) {   
 if (nodes.size() == 0)   
 return;   
   
 vector<TreeNode \*> next\_level;   
 for (int i=0; i<nodes.size(); i++) {   
 if (nodes[i]->left)   
 next\_level.push\_back(nodes[i]->left);   
 if (nodes[i]->right)   
 next\_level.push\_back(nodes[i]->right);   
 }   
   
 do\_order(next\_level, res);   
 vector<int> temp;   
 for (int i=0; i<nodes.size(); i++) {   
 temp.push\_back(nodes[i]->val);   
 }   
 res.push\_back(temp);   
 }   
};   
   
/\*4   
//minimum-depth-of-binary-tree   
   
/\*   
\* https://leetcode-cn.com/problems/minimum-depth-of-binary-tree   
\* 题目描述：   
\* 给定一个二叉树，找出其最小深度。   
\* 最小深度是从根节点到最近叶子节点的最短路径上的节点数量。   
\*   
\* 说明:   
\* 叶子节点是指没有子节点的节点。   
\*   
\* 示例:   
\* 给定二叉树 [3,9,20,null,null,15,7],   
\* 3   
\* / \   
\* 9 20   
\* / \   
\* 15 7   
\* 返回它的最小深度 2.   
\*   
\* 解题思路：   
\* 使用递归的方法，求最短路径，也就是左右子树的最短路径   
\* 主要考虑几种特殊情况：   
\* 1. 递归终止条件，即无子节点时终止，即空指针   
\* 2. 没有左子树或右子树时如何处理   
\* 3. 左右子树都存在时如何处理   
\* 4. 左右子树都不存在时如何处理   
\* 总之，只要进入递归的不是空指针，当节点都算一个深度值   
\* 这也就是为什么返回值后都 +1 的原因   
\*/   
   
/\*\*   
 \* Definition for a binary tree node.   
 \* struct TreeNode {   
 \* int val;   
 \* TreeNode \*left;   
 \* TreeNode \*right;   
 \* TreeNode(int x) : val(x), left(NULL), right(NULL) {}   
 \* };   
 \*/   
class Solution {   
public:   
 int minDepth(TreeNode\* root) {   
 if (root == nullptr)   
 return 0;   
   
 int left, right;   
 left = minDepth(root->left);   
 right = minDepth(root->right);   
   
 if (right == 0 && left == 0)   
 return 1;   
 if (right == 0)   
 return left+1;   
 if (left == 0)   
 return right+1;   
   
 return left>right ? (right+1) : (left+1);   
 }   
};   
   
/\*5   
//two-sum   
/\*   
 https://leetcode-cn.com/problems/two-sum   
 给定一个整数数组 nums 和一个目标值 target，   
 请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。   
 你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。   
   
 示例:   
 给定 nums = [2, 7, 11, 15], target = 9   
 因为 nums[0] + nums[1] = 2 + 7 = 9   
 所以返回 [0, 1]   
   
 解题思路：   
 1. 最简单的方法就是暴力搜索了，两个嵌套循环就搞定了   
 2. 为了降低复杂度，努力的方向就是在有了第一个数之后   
 如何快速的判断是否有另一个数存在，这里就可以看出其实是一种映射   
 3. 映射方式可以使用 map、hash table 等，而 hash table 更快   
 所以这里使用 unordered\_map，因为其底层实现就是 hash table   
\*/   
class Solution {   
public:   
 vector<int> twoSum(vector<int>& nums, int target) {   
 unordered\_map<int, int> data;   
 construct\_map(nums, data);   
   
 int i=0;   
 vector<int> res;   
 while (i < nums.size()) {   
 /\*   
 这里是不对的，两个数的和不一定月假越大，比如负数！   
 if (nums[i] > target) {   
 i++;   
 continue;   
 }   
 \*/   
 int left = target - nums[i];   
 if (data.count(left) && data[left] != i) {   
 res.push\_back(i);   
 res.push\_back(data[left]);   
 break;   
 }   
   
 i++;   
 }   
   
 return res;   
 }   
   
 void construct\_map(vector<int> &nums, unordered\_map<int, int> &data) {   
 int i = 0;   
 while (i < nums.size()) {   
 data[nums[i]] = i;   
 i++;   
 }   
 }   
};   
   
/\*6   
\* https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array   
\* 题目描述：   
\* 给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素只出现一次，   
\* 返回移除后数组的新长度。   
\* 不要使用额外的数组空间，你必须在原地修改输入数组   
\* 并在使用 O(1) 额外空间的条件下完成。   
\*   
\* 示例 1:   
\* 给定数组 nums = [1,1,2],   
\* 函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。   
\* 你不需要考虑数组中超出新长度后面的元素。   
\*   
\* 示例 2:   
\* 给定 nums = [0,0,1,1,1,2,2,3,3,4],   
\* 函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。   
\*   
\* 注：   
\* 你不需要考虑数组中超出新长度后面的元素。   
\*   
\* 解题思路：   
\* 1. 题目的意思其实就是将不同的数字向前紧密排列，因此涉及到下一个新的数字   
\* 放在哪里，即需要一个索引值指定，还需要知道最新的数字在哪里，即又需要   
\* 一个索引值   
\* 2. 因此使用两个索引值，相同时一个前移、一个不动；   
\* 遇到不同时，前边的指针前移并将新的数字复制过来，然后走的快的   
\* 指针继续前移，如此反复即可   
\*/

class Solution {   
public:   
 int removeDuplicates(vector<int>& nums) {   
 if (nums.size() <= 1)   
 return nums.size();   
   
 int head = 0, tail = 0;   
 while (tail < nums.size()) {   
 if (nums[head] == nums[tail]) {   
 tail++;   
 } else {   
 head++;   
 if (head < tail) {   
 nums[head] = nums[tail];   
 tail++;   
 }   
 }   
 }   
   
 nums.resize(head+1);   
   
 return head+1;   
 }   
};   
   
/\*7   
\* https://leetcode-cn.com/problems/remove-element   
\* 题目描述：   
\* 给定一个数组 nums 和一个值 val，你需要原地移除所有数值等于 val 的元素，   
\* 返回移除后数组的新长度。   
\* 不要使用额外的数组空间，你必须在原地修改输入数组   
\* 并在使用 O(1) 额外空间的条件下完成。   
\* 元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。   
\*   
\* 示例 1:   
\* 给定 nums = [3,2,2,3], val = 3,   
\* 函数应该返回新的长度 2, 并且 nums 中的前两个元素均为 2。   
\* 你不需要考虑数组中超出新长度后面的元素。   
\*   
\* 示例 2:   
\* 给定 nums = [0,1,2,2,3,0,4,2], val = 2,   
\* 函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。   
\* 注意这五个元素可为任意顺序。   
\*   
\* 注：   
\* 你不需要考虑数组中超出新长度后面的元素。   
\*   
\* 解题思路：   
\* 1. 解题思路与题目 0026\_remove\_duplicates\_from\_sorted\_array 类似   
\* 不同的是这里需要覆盖的是指定数字，同样需要双指针   
\* 2. 注意指针的含义即可   
\*/   
   
class Solution {   
public:   
 int removeElement(vector<int>& nums, int val) {   
 if (nums.size() == 0)   
 return 0;   
   
 int head = 0, tail = 0;   
 while (tail < nums.size()) {   
 if (nums[tail] == val) {   
 tail++;   
 } else {   
 if (head < tail) {   
 nums[head] = nums[tail];   
 }   
 head++;   
 tail++;   
 }   
 }   
   
 nums.resize(head);   
   
 return head;   
 }   
};   
   
/\*8   
\* https://leetcode-cn.com/problems/search-insert-position   
\* 题目描述：   
\* 给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。   
\* 如果目标值不存在于数组中，返回它将会被按顺序插入的位置。   
\* 你可以假设数组中无重复元素。   
\*   
\* 示例 1:   
\* 输入: [1,3,5,6], 5   
\* 输出: 2   
\*   
\* 示例 2:   
\* 输入: [1,3,5,6], 2   
\* 输出: 1   
\*   
\* 示例 3:   
\* 输入: [1,3,5,6], 7   
\* 输出: 4   
\*   
\* 示例 4:   
\* 输入: [1,3,5,6], 0   
\* 输出: 0   
\*   
\* 解题思路：   
\* 1. 最简单粗暴的方法就是逐个遍历，直到找到目标数字或者第一个   
\* 大于目标数的值，不过时间复杂度为 O(N)   
\* 2. 由于给定的数组是有序数组，所以可以使用二分法进行查找   
\* 这样时间复杂度就可以变成 O(logN)   
\*/   
   
class Solution {   
public:   
 int searchInsert(vector<int>& nums, int target) {   
 if (nums.size() == 0)   
 return 0;   
   
 int start = 0, end = nums.size() - 1;   
 if (nums[start] > target)   
 return 0;   
 if (nums[end] < target)   
 return end + 1;   
   
 int mid;   
 while (start < end) {   
 /\*   
 本来想优化一下，结果这句造成了死循环   
 按理说有符号的正数不应该出现这种情况的呀！   
 mid = start + (end - start) >> 1;   
 \*/   
 mid = start + (end - start) / 2;   
 if (nums[mid] > target) {   
 end = mid - 1;   
 } else if (nums[mid] < target) {   
 start = mid + 1;   
 } else {   
 return mid;   
 }   
 }   
   
 if (nums[start] >= target)   
 return start;   
 return start + 1;   
 }   
};   
   
/\*9   
jump-game-ii   
题目：   
 给定一个非负整数数组，你最初位于数组的第一个位置。   
 数组中的每个元素代表你在该位置可以跳跃的最大长度。   
 你的目标是使用最少的跳跃次数到达数组的最后一个位置。   
示例:   
 输入: [2,3,1,1,4]   
 输出: 2   
 解释: 跳到最后一个位置的最小跳跃数是 2。   
 从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。   
说明:   
 假设你总是可以到达数组的最后一个位置。   
\*/   
   
/\*   
 贪婪算法(Greedy Algorithm), 时间复杂度 O(n)   
\*/   
class Solution {   
public:   
 int jump(vector<int>& nums) {   
 int step = 0, cur = 0, prev = 0;   
 while (cur < nums.size()-1) {   
 step++;   
 int temp = cur;   
 int index = 0;   
 // 因为当前点可达，则它前方的点中上次的产生最大可达点之间都可达了   
 // 故检查这段中的元素最大的可达距离   
 for (int i=prev; i<=temp; i++) {   
 if (cur < nums[i] + i) {   
 // 记录上次最大可达 与 当前点 之间的点中产生最大可达的点坐标   
 index = i;   
 cur = nums[i] + i;   
 }   
 }   
 if (cur == temp)   
 return -1;   
   
 prev = index;   
 }   
   
 return step;   
 }   
};   
 /\*11   
jump-game   
题目：   
 给定一个非负整数数组，你最初位于数组的第一个位置。   
 数组中的每个元素代表你在该位置可以跳跃的最大长度。   
 判断你是否能够到达最后一个位置。   
示例 1:   
 输入: [2,3,1,1,4]   
 输出: true   
 解释: 从位置 0 到 1 跳 1 步, 然后跳 3 步到达最后一个位置。   
示例 2:   
 输入: [3,2,1,0,4]   
 输出: false   
 解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0，   
 所以你永远不可能到达最后一个位置。   
\*/   
   
/\*   
 动态规划(DP)方法，复杂度 O(n^2)   
 解题思路：   
 1. 该问题要解决的是最后一个元素是否可达，故需要判断最后一个元素   
 前边可达的元素是否可达最后一个元素   
 2. 由此问题变为从 第 1 个元素开始扫描，判断当前元素是否可达   
 直到扫描完整个数组   
\*/   
class Solution {   
public:   
 bool canJump(vector<int>& nums) {   
 if (nums.size() <= 1)   
 return true;   
   
 bool can[nums.size()];   
 can[0] = true;   
   
 for (int i=1; i<nums.size(); i++) {   
 for (int j=0; j<i; j++) {   
 // 判断当前元素前方所有的元素是否有可达到当前元素的   
 if (can[j] && j+nums[j]>=i) {   
 can[i] = true;   
 break;   
 }   
 }   
 }   
   
 return can[nums.size()-1];   
 }   
};   
   
/\*   
 贪婪算法(Greedy Algorithm), 时间复杂度 O(n)   
\*/   
class Solution {   
public:   
 bool canJump(vector<int>& nums) {   
 if (nums.size() <= 1)   
 return true;   
   
 int reach = 0;   
 for (int i=0; i<nums.size(); i++) {   
 // 第一个条件表示当前点不可达，即前方元素的最远可达无法到达该元素   
 // 第二个条件为最后一个元素可达   
 if (i > reach || reach >= nums.size() - 1)   
 break;   
   
 reach = max(reach, i+nums[i]);   
 }   
   
 return reach >= nums.size() - 1;   
 }   
};

ll

# -\*- coding:utf-8 -\*-   
class Solution:   
 def minNumberInRotateArray(self, rArr):   
 if not rArr:   
 return 0   
 low,hi = 0,len(rArr)-1   
 while low<hi :   
 mid = low + (hi-low)/2   
 if rArr[mid]<rArr[hi]:   
 hi = mid   
 elif rArr[mid]>rArr[hi]:   
 low = mid +1   
 else:   
 hi-=1   
 return rArr[low]   
   
 # write code here