### 1. Two Sum

class Solution(object):

def twoSum(self, nums, target):

"""

:type nums: List[int]

:type target: int

:rtype: List[int]

"""

list1=[]

for x in range(len(nums)):

if ((target-nums[x]) in nums) and (nums.index(target-nums[x])!=x):

list1.append(x)

list1.append(nums.index(target-nums[x]))

break

else:

continue

return list1

public class Solution {

public int[] twoSum(int[] nums, int target) {

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

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

if(nums[i]+nums[j]==target){

int[] list={i,j};

return list;

}

}

}

return null;

}

}

public class Solution {

public int[] twoSum(int[] nums, int target) {

Map<Integer, Integer> numb=new HashMap<>();

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

numb.put(nums[i],i);

}

numb.get(nums[i])

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

if(numb.containsKey(target-nums[i])&&(i!=numb.get(target-nums[i]))){

return new int[] {i,numb.get(target-nums[i])};

}

}

return null;

}

}

### 242. Valid Anagram

public class Solution {

public boolean isAnagram(String s, String t) {

if(s.length()!=t.length()){

return false;

}

char[] s1=s.toCharArray();

char[] t1=t.toCharArray();

Arrays.sort(s1);

Arrays.sort(t1);

return Arrays.equals(s1,t1);

}

}

public class Solution {

public boolean isAnagram(String s, String t) {

if(s.length()!=t.length()){

return false;

}

int[] counter=new int[26];

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

counter[s.charAt(i)-'a']++;

}

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

counter[t.charAt(i)-'a']--;

if( counter[t.charAt(i)-'a']<0){

return false;

}

}

return true;

}

}

### 14. Longest Common Prefix

思路：一个字母一个字母的比较

public class Solution {

public String longestCommonPrefix(String[] strs) {

if(strs.length==0) return "";

String prefix=strs[0];

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

char a=prefix.charAt(i);

for(int j=1;j<strs.length;j++){

if(i==strs[j].length()||strs[j].charAt(i)!=a){ //注意先后顺序

return prefix.substring(0,i); //substring(0,0),返回空串””

}

}

}

return prefix;

}

}

思路：字符串整个比较是否为子串，然后逐个去掉字母

public class Solution {

public String longestCommonPrefix(String[] strs) {

if(strs.length==0) return "";

String prefix=strs[0];

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

while(strs[i].indexOf(prefix)!=0){ //保证是prefix

prefix=prefix.substring(0,prefix.length()-1);

if(prefix.isEmpty()) return "";

}

}

return prefix;

}

}

思路：分治，

二分查找

思路：将候选人分两半

public class Solution {

public String longestCommonPrefix(String[] strs) {

if(strs==null||strs.length==0) return "";

int minlen=Integer.MAX\_VALUE;

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

if(strs[i].length()<minlen)

minlen=strs[i].length();

}

int low=1;

int high=minlen;

while(low<=high){

int mid=(low+high)/2;

if(iscommonPrefix(strs,mid)){

low=mid+1;

}else{

high=mid-1;

}

}

return strs[0].substring(0,(low+high)/2);

}

private boolean iscommonPrefix(String[] strs,int mid){

String str=strs[0].substring(0,mid);

for(int j=1;j<strs.length;j++){

if(!strs[j].startsWith(str)){

return false;

}

}

return true;

}

}

### 19 Remove Nth Node From End of List

注：链表中指针指向同一个内存区域

思路：获得链表长度，找到删除点的位置

public class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

ListNode copyhead=new ListNode(0);//初始化了一个头

copyhead.next=head; //注意next

int len=0;

ListNode first=head;

while(first!=null){

len++;

first=first.next;

}

first=copyhead;

len-=n;

while(len>0){

first=first.next;

len--;

}

first.next=first.next.next;

return copyhead.next;

}

}

思路2：两个指针，间隔为n，一个指针在null时候，另一个指针刚好在n前一个位置

### 83 Remove Duplicates from Sorted List

问题：超时

public class Solution {

public ListNode deleteDuplicates(ListNode head) {

if(head==null){return null;}

ListNode first= head.next;

ListNode start= new ListNode(head.val);

ListNode result=start;

while(first!=null){

if(first.val!=start.val){

ListNode mid=new ListNode(first.val);

first=first.next;

start.next=mid;

start=start.next;

}

}

return result;

}

}

思路：直接修改链表后继，既可以有效

public class Solution {

public ListNode deleteDuplicates(ListNode head) {

if(head==null){return null;}

ListNode first= head;

while(first!=null&&first.next!=null){

if(first.val!=first.next.val){

first=first.next;

}else{

first.next=first.next.next;

}

}

return head;

}

}

### 141. Linked List Cycle

是否存在环

思路：一个指针走一步，另一个指针走两步

public class Solution {

public boolean hasCycle(ListNode head) {

if(head==null) return false;

ListNode first=head;

ListNode second=new ListNode(head.val);

if(first.next!=null&&first.next.next!=null){

second=head.next.next;

while(second.next!=null&&second.next.next!=null){ //while 循环条件 条件1

if (first==second){ //判断链表两个指针是否指向同一个节点 条件2 条件1,2 互换作为while条件

return true;

}

first=first.next;

second=second.next.next;

}

return false;

}else{

return false;

}

}

}

思路2：使用hashtable 判断这个节点是否被访问过

public class Solution {

public boolean hasCycle(ListNode head) {

Set<ListNode> nodeSeed =new HashSet<>();

while(head!=null){

if(nodeSeed.contains(head)){

return true;

}else{

nodeSeed.add(head);

}

head=head.next;

}

return false;

}

}

public class Solution {

public boolean isSymmetric(TreeNode root) {

return Mirror(root.left,root.right);

}

public boolean Mirror (TreeNode root1,TreeNode root2){

if(root1==null&&root2==null) return true;

if(root1==null||root2==null) return false;

return root1.val==root2.val&&Mirror(root1.left,root2.right)&&Mirror(root1.right,root2.left);

}

}

### 101. Symmetric Tree

public class Solution {

public boolean isSymmetric(TreeNode root) {

return Mirror(root.left,root.right);

}

public boolean Mirror (TreeNode root1,TreeNode root2){

if(root1==null&&root2==null) return true;

if(root1==null||root2==null) return false; //设定巧妙，上一句已经判断两个都为null情况

return root1.val==root2.val&&Mirror(root1.left,root2.right)&&Mirror(root1.right,root2.left);

}

}

### 160. Intersection of Two Linked Lists

思路1：hashtable：空间复杂度超了

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA==null&&headB==null) return null;

if (headA==null||headB==null) return null;

Set a=new HashSet();

ListNode first=headA;

while(first!=null){

a.add(first);

first=first.next;

}

while(headB!=null){

if(a.contains(headB)){

return headB;

}

}

return null;

}

}

思路2：让两个指针走的步数相同，如果存在intersection的话就会相遇。不存在intersection的话，两个链表最后一个元素不同。

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA==null&&headB==null) return null;

if (headA==null||headB==null) return null;

ListNode pA=headA;

ListNode pB=headB;

int LastA=Integer.MAX\_VALUE;

int LastB=Integer.MAX\_VALUE;

while(pA!=pB){

if(pA.next==null){

LastA=pA.val;

pA=headB;

}else{

pA=pA.next;

}

if(pB.next==null){

LastB=pB.val;

pB=headA;

}else{

pB=pB.next;

}

if(LastA!=Integer.MAX\_VALUE&&LastB!=Integer.MAX\_VALUE&&LastA!=LastB) return null;

}

return pA;

}

}

### 206. Reverse Linked List

思路1：递归

public class Solution {

public ListNode reverseList(ListNode head) {

if(head==null||head.next==null) return head; //递归的结束条件

ListNode p=reverseList(head.next);

head.next.next=head; //改变节点后继，并不改变节点内容，只是一个指针 //递归过程

head.next=null;

return p;

}

}

思路2：迭代

public class Solution {

public ListNode reverseList(ListNode head) {

ListNode h1=null; //考虑空链表的问题

ListNode h2=head;

while(h2!=null){

ListNode h3=h2.next;

h2.next=h1;

h1=h2;

h2=h3;

}

return h1;

}

}

### 27. Remove Element

思路：先排序，确定val个数

public class Solution {

public int removeElement(int[] nums, int val) {

Arrays.sort(nums);

int count=0;

int findex=0;

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

if(nums[i]==val){

if(count==0){

findex=i;

}

count++;

}

}

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

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

}

return nums.length-count;

}

}

思路2：头尾两个指针，头指针<尾指针，重点注意一下头==尾的情况，时间复杂度降为o（n）

public class Solution {

public int removeElement(int[] nums, int val) {

int count=0;

int lindex=nums.length-1;

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

if(nums[i]==val){

count+=1;

while(nums[lindex]==val){

if(lindex==i) break;

count+=1;

lindex=lindex-1;

}

nums[i]=nums[lindex];

lindex=lindex-1;

if(lindex==i) break;

}

}

return nums.length-count;

}

}

巧妙： n设置为整个数组长度

public class Solution {

public int removeElement(int[] nums, int val) {

int i = 0;

int n = nums.length;

while (i < n) {

if (nums[i] == val) {

nums[i] = nums[n - 1];

// reduce array size by one

n--; //直到nums[i]为一个非val的值得时候，i++；

} else {

i++;

}

}

return n;

}

}

### 189. Rotate Array （other）

思路：新增了一个数组作为中间介，时间O（n），1pass空间O(n)

public class Solution {

public void rotate(int[] nums, int k) {

int len=nums.length;

int[] nums1=new int[len];

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

nums1[(i+k)%len]=nums[i];

}

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

nums[i]=nums1[i];

}

}

}

思路：右移转换为反转，可以节省空间，但是几乎3pass

public class Solution {

public void rotate(int[] nums, int k) {

k%=nums.length;

reverse(nums,0,nums.length-1);

reverse(nums,0,k-1);

reverse(nums,k,nums.length-1);

}

public void reverse(int[] nums,int start, int end){

while(start<end){

int temp=nums[start];

nums[start]=nums[end];

nums[end]=temp;

start++;

end--;

}

}

}

思路：需要确定有环的存在，一个一个的替换，需要一个保存中间点的space

public class Solution {

public void rotate(int[] nums, int k) {

k = k % nums.length;

int count = 0;

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

int current = start;

int prev = nums[start];

do {

int next = (current + k) % nums.length;

int temp = nums[next];

nums[next] = prev;

prev = temp;

current = next;

count++;

} while (start != current);

}

}

}

### 191. Number of 1 Bits （other）

思路1：时间复杂度高

public class Solution {

// you need to treat n as an unsigned value

public int hammingWeight(int n) {

int count=0;

while(n==0){

int bo=n%2;

n=n/2;

if(bo==1) count+=1;

}

return count;

}

}

思路2：每个bit的处理，进行位运算。

public class Solution {

// you need to treat n as an unsigned value

public int hammingWeight(int n) {

int count=0;

int mask=1;

for(int i=0;i<32;i++){

if((n&mask)!=0){ //注意运算符优先级

count++;

}

mask<<=1;

}

return count;

}

}

### 217. Contains Duplicate

思路：先排序，再前后比较

public class Solution {

public boolean containsDuplicate(int[] nums) {

Arrays.sort(nums);

for(int i=0;i<(nums.length-1);i++){

if(nums[i]==nums[i+1]) return true;

}

return false;

}

}

思路2：哈希表，发现这个数在hash表中是否存在，数组中的数一个一个添加到hash表中

public class Solution {

public boolean containsDuplicate(int[] nums) {

Set<Integer> num=new HashSet<>(nums.length);

for(int x : nums){

if(num.contains(x)) return true;

num.add(x);

}

return false;

}

}

### 219. Contains Duplicate II

思路：map中不允许有重复元素，元素下标是递增的顺序。Map边插入put边进行比较

public class Solution {

public boolean containsNearbyDuplicate(int[] nums, int k) {

Map<Integer,Integer> a= new HashMap<>(nums.length);

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

if(a.containsKey(nums[i])){

int j=a.get(nums[i]);

if(i-j<=k){

return true;

}else{ //多个重复元素

a.put(nums[i],i);

}

}else{

a.put(nums[i],i);

}

}

return false;

}

}

public class Solution {

public boolean containsNearbyDuplicate(int[] nums, int k) {

if(nums.length<2) return false;

Set temp= new HashSet(k+1);

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

for (int j=i;j<(i+k+1);j++){

if(temp.contains(nums[j])) {

return true;

}

temp.add(nums[j]);

}

}

return false;

}

}

### 198. House Robber

思路：动态规划

public class Solution {

public int rob(int[] nums) {

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

if(nums.length==1) return nums[0];

if(nums.length==2) return Math.max(nums[0],nums[1]);

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

a[0]=nums[0];

a[1]=Math.max(nums[0],nums[1]);

for(int i=2;i<nums.length;i++){

a[i]=Math.max(a[i-2]+nums[i],a[i-1]);

}

return a[nums.length-1];

}

}

### 225. Implement Stack using Queues

思路：查找函数，各种接口实现,<>泛型 push: O(1),pop: O(n)

class MyStack {

Queue<Integer> a=new LinkedList<>();

Queue<Integer> b=new LinkedList<>();

int top;

// Push element x onto stack.

public void push(int x) {

a.add(x);

top=x;

}

// Removes the element on top of the stack.

public void pop() {

while(a.size()>1){

top=a.remove();

b.add(top);

}

a.remove();

Queue<Integer> c=new LinkedList<>();

c=a;

a=b;

b=c;

}

// Get the top element.

public int top() {

return top;

}

// Return whether the stack is empty.

public boolean empty() {

return a.isEmpty();

}

}

思路2：队列中存放顺序和stack相同，两个队列，push: O(n),pop: O(1)

class MyStack {

Queue<Integer> a=new LinkedList<>();

Queue<Integer> b=new LinkedList<>();

// Push element x onto stack.

public void push(int x) {

b.add(x);

while(a.size()>0){

int tmp=a.remove();

b.add(tmp);

}

Queue<Integer> c=a;

a=b;

b=c;

}

// Removes the element on top of the stack.

public void pop() {

a.remove();

}

// Get the top element.

public int top() {

return a.peek();

}

// Return whether the stack is empty.

public boolean empty() {

return a.isEmpty();

}

}

思路3：只使用一个队列，减少空间， push: O(n),pop: O(1)

class MyStack {

Queue<Integer> a=new LinkedList<>();

// Push element x onto stack.

public void push(int x) {

int num=a.size();

a.add(x);

while(num>0){

int c=a.remove();

a.add(c);

num--;

}

}

// Removes the element on top of the stack.

public void pop() {

a.remove();

}

// Get the top element.

public int top() {

return a.peek();

}

// Return whether the stack is empty.

public boolean empty() {

return a.isEmpty();

}

}

### 226. Invert Binary Tree

思路：递归方法解决，判断一个树是空的

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode invertTree(TreeNode root) {

if(root==null||root.left==null&&root.right==null) return root;

//根是空的，递归结束条件 if(root==null)

TreeNode a=root.left;

root.left=root.right;

root.right=a;

invertTree(root.left);

invertTree(root.right);

return root;

}

}

思路2：广度优先搜索，队列，

public class Solution {

public TreeNode invertTree(TreeNode root) {

if(root==null) return root;

Queue<TreeNode> a=new LinkedList<>();

a.add(root);

while(!a.isEmpty()){ //队列中为空时，循环结束

TreeNode r=a.poll();

if(r.left!=null){

a.add(r.left);

}

if(r.right!=null){

a.add(r.right);

}

TreeNode temp= r.left;

r.left=r.right;

r.right=temp;

}

return root;

}

}

### 232. Implement Queue using Stacks

思路1：stack存储的顺序和queue相同，栈不能直接赋值

class MyQueue {

Stack<Integer> a=new Stack<>();

Stack<Integer> b=new Stack<>();

int front;

// Push element x to the back of queue.

public void push(int x) {

while(a.size()>0){

b.push(a.pop());

}

b.push(x);

while(b.size()>0){

a.push(b.pop());

}

}

// Removes the element from in front of queue.

public void pop() {

a.pop();

}

// Get the front element.

public int peek() {

return a.peek();

}

// Return whether the queue is empty.

public boolean empty() {

return a.isEmpty();

}

}

思路2:2个Stack，pop，peek麻烦

class MyQueue {

Stack<Integer> a=new Stack<>();

Stack<Integer> b=new Stack<>();

int top;

// Push element x to the back of queue.

public void push(int x) {

a.push(x);

}

// Removes the element from in front of queue.

public void pop() {

while(a.size()>1){

b.push(a.pop());

}

int t=a.pop();

while(b.size()>0){

a.push(b.pop());

}

}

// Get the front element.

public int peek() {

while(a.size()>1){

b.push(a.pop());

}

int t=a.pop();

b.push(t);

while(b.size()>0){

a.push(b.pop());

}

return t;

}

// Return whether the queue is empty.

public boolean empty() {

return a.isEmpty();

}

}

### 237. Delete Node in a Linked List

public class Solution {

public void deleteNode(ListNode node) {

node.val = node.next.val;

node.next = node.next.next;

}

}

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

ListNode dummyHead = new ListNode(0);

ListNode p = l1, q = l2, curr = dummyHead;

int carry = 0;

while (p != null || q != null) {

int x = (p != null) ? p.val : 0;

int y = (q != null) ? q.val : 0;

int sum = carry + x + y;

carry = sum / 10;

curr.next = new ListNode(sum % 10);

curr = curr.next;

if (p != null) p = p.next;

if (q != null) q = q.next;

}

if (carry > 0) {

curr.next = new ListNode(carry);

}

return dummyHead.next;

}

ListNode rr=new ListNode(0);

ListNode kk=rr;

int reverse=0;

while(l1!=null||l2!=null){

int val1=(l1==null)? 0:l1.val;

int val2=(l2==null)? 0:l2.val;

int mid=val1+val2+reverse;

reverse=mid/10;

rr.next=new ListNode(mid);

rr=rr.next;

if(l2!=null) l1=l1.next;

if(l1!=null) l2=l2.next;

}

if(reverse==1){

rr.next=new ListNode(1);

rr=rr.next;

}

return kk.next;

### 3. Longest Substring Without Repeating Characters

思路1（self）：查找相同元素，定位上一次相同元素位置。

public class Solution {

public int lengthOfLongestSubstring(String s) {

int len=0;

int fromIndex=0;

Set<Character> tmp=new HashSet<>();

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

if(tmp.contains(s.charAt(i))){

fromIndex=s.indexOf(s.charAt(i),fromIndex)+1;

tmp.clear();

for(int j=fromIndex;j<i;j++)

tmp.add(s.charAt(j));

}

tmp.add(s.charAt(i));

if(tmp.size()>len) len=tmp.size();

}

return len;

}

}

思路2:sliding window【两个指针】：当不确定循环几次的时候，while，两个位置变量定位位置。时间复杂度O(2n)

public class Solution {

public int lengthOfLongestSubstring(String s) {

int len=s.length();

int fromIndex=0,currIndex=0,lens=0;

Set<Character> tmp=new HashSet<>();

while(fromIndex<len&&currIndex<len){

if(tmp.contains(s.charAt(currIndex))){

tmp.remove(s.charAt(fromIndex++));

}else{

tmp.add(s.charAt(currIndex++));

lens=Math.max(lens,currIndex-fromIndex);

}

}

return lens;

}

}

思路：sliding window optimized，使用Map。

注意头尾，若不想边界出错，头尾指向window的头尾

public class Solution {

public int lengthOfLongestSubstring(String s) {

int len=s.length();

int fromIndex=0,currIndex=0,lens=0;

Map<Character,Integer> tmp=new HashMap<>();

if(s.length()==0) return 0;

for(currIndex=0;currIndex<len;currIndex++){

if(tmp.containsKey(s.charAt(currIndex))){

fromIndex=Math.max(tmp.get(s.charAt(currIndex)),fromIndex);

}

lens=Math.max(lens,currIndex-fromIndex+1);

tmp.put(s.charAt(currIndex),currIndex+1);

}

return lens;

}

}

### 5. Longest Palindromic Substring(最长回文子串)（Manacher's Algorithm

### ）

废品1：

public class Solution {

public String longestPalindrome(String s) {

int len=s.length();

int max=0;

int maxl=0;

int k=0,j=0;

String p="";

if(len==0) return null;

if(len==1) return s;

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

if(s.charAt(i)==s.charAt(i+1)){

maxl=2;

int t=Math.min(i+1,len-i-1);

if(t>1){

k=i+2;

for(j=i-1;j>i-t+1;j--){

if(s.charAt(j)==s.charAt(k)){maxl+=2;k++;}else{

break;

}

}

}

}

max=Math.max(max,maxl);

if(maxl>max){

p=s.substring(k,i);

}

}

return p;

}

}

思路：动态规划核心思想：1.最优子问题 2.重叠子问题。最长回文子串的每一个子串都是都是回文子串，主要是如何找到最长回文子串，每次都进行判断。

注：动态规划注重方向，根据递归表达式，在后一个算出结果之前的所有结果都要算出。

public String longestPalindrome(String s) {

int n = s.length();

String res = null;

boolean[][] dp = new boolean[n][n];

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

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

dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i + 1][j - 1]);

if (dp[i][j] && (res == null || j - i + 1 > res.length())) {

res = s.substring(i, j + 1);

}

}

}

return res;

}

以下代码存在的问题：dp[5][8]计算的时候，dp[6][7]并没有进行计算

**int** len =s.length();

**int** strlen=0;

**int** strf=0;

**int** strl=0;

**boolean**[][] dp=**new** **boolean**[len+1][len+1];

**for**(**int** i=1;i<len+1;i++){

**for**(**int** j=1;j<len+1;j++){

**if**(s.charAt(i-1)==s.charAt(j-1)){

**if**(j>i){

dp[i][j]=**true**&&(j-i<3||dp[i+1][j-1]);

**if**(j-i+1>=strlen){

strf=i;

strl=j;

strlen=j-i+1;

}

}**else**{

dp[i][j]=**true**&&(i-j<3||dp[i-1][j+1]);

**if**(i-j+1>=strlen){

strf=j;

strl=i;

strlen=i-j+1;

}

}

}

}

}

**if**(strf==strl) **return** s;

System.***out***.println("start"+strf+"finish"+strl);

**return** s.substring(strf-1,strl);

}

}

思路2：从中心往外面扩展，递归方式实现

递归：1.问题可以转化为多个相同求解方案的子问题求解。2.递归次数有限。3.结束递归的条件。

递归的缺点：运行效率较低 方案：将递归方程求解得到递推式

### 6. ZigZag Conversion

思路：每一行进行缓存，然后拼接。StringBuffer，对象声明，实例化与初始化

public class Solution {

public String convert(String s, int numRows) {

StringBuffer[] ss=new StringBuffer[numRows];

int i=0;

for(int k=0;k<numRows;k++){

ss[k]=new StringBuffer();

} //这是必须的

while(i<s.length()){

for(int j=0;j<numRows&&i<s.length();j++){

ss[j].append(s.charAt(i));

i++;

}

for(int j=numRows-2;j>0&&i<s.length();j--){

ss[j].append(s.charAt(i));

i++;

}

}

for(i=1;i<numRows;i++){

ss[0].append(ss[i]);

}

return ss[0].toString();

}

}

### 11. Container With Most Water（策略Strategy）

思路：two pointers，暴力破解n^2,会超时，想技巧将一个变量设置为相对固定

public class Solution {

public int maxArea(int[] height) {

int fir=0;

int las=height.length-1;

int maxarea=0;

while(fir<las){ fir！=las不会超时，之前一定抽风

maxarea=Math.max(maxarea,Math.min(height[fir],height[las])\*(las-fir));

if(height[fir]<height[las]){

fir++;

}else{

las--;

}

}

return maxarea;

}

}

### 7. Reverse Integer

思路：主要考虑是否越界溢出，int类型存在的问题。

巧妙：使用long型，如果大于int的最大值返回0

public class Solution {

public int reverse(int x) {

int b=Math.abs(x);

long c=0;

while(b>0){

c=b%10+c\*10;

b=b/10;

if (c>Integer.MAX\_VALUE) return 0;

}

if(x<0) return (int)-c;

return (int)c;

}

}

### 9. Palindrome Number

思路：借鉴7

public class Solution {

public boolean isPalindrome(int x) {

if(x<0) return false;

if(reverse(x)==x) return true;

return false;

}

public int reverse(int x){

long c=0;

while(x>0){

c=c\*10+x%10;

x=x/10;

if(c>Integer.MAX\_VALUE) return 0;

}

return (int)c;

}

}

### 20. Valid Parentheses

思路：没有考虑顺序

public class Solution {

public boolean isValid(String s) {

int[] f=new int[3];

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

if(s.charAt(i)=='('){

f[0]++;

}else if(s.charAt(i)==')'){

f[0]--;

if(f[0]<0) return false;

}

if(s.charAt(i)=='{'){

f[0]++;

}else if(s.charAt(i)=='}'){

f[0]--;

if(f[0]<0) return false;

}

if(s.charAt(i)=='['){

f[0]++;

}else if(s.charAt(i)==']'){

f[0]--;

if(f[0]<0) return false;

}

}

if(f[0]==0&&f[1]==0&&f[2]==0) return true;

return false;

}

}

思路：stack可以保证顺序，栈中保存的是对象，需要强制类型转换

public class Solution {

public boolean isValid(String s) {

Stack p=new Stack();

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

if(s.charAt(i)=='('||s.charAt(i)=='{'||s.charAt(i)=='['){p.push(s.charAt(i));}

else{

if(p.empty()) return false;

char tmp=(char)p.pop();

if(s.charAt(i)==')'&&tmp!='(') return false;

if(s.charAt(i)=='}'&&tmp!='{') return false;

if(s.charAt(i)==']'&&tmp!='[') return false;

}

}

if(p.empty()) return true;

return false;

}

}

### 22. Generate Parentheses

思路：回溯方法（定义：深度优先搜索问题解的方法）

明确问题的解空间：解空间树/状态空间

画解空间树：只存储根节点到当前节点的路径（画树）

1. 子集树：n个元素集合S找出满足某种性质的子集
2. 排列树：n个元素满足某种性质排列

回溯采用深度优先搜索方式

实现方式:自定义栈；递归方式

提高回溯效率：剪枝函数（约束函数，界限函数）

回溯步骤：

1. 确定解空间
2. 确定节点扩展搜索规则
3. 深度优先方式搜索解空间树，剪枝函数避免无效搜索

实现上问题：

1. 没有完全回溯，回溯时要恢复环境
2. 具体问题逻辑上考虑周全，扩展搜索规则要正确

本题思路：

约束条件1：开始必须是“（”

约束条件2：“（”数量<=”）”，保证逻辑上括号的匹配

约束条件3: 结束的条件，两边括号的数量都为0

Eclipse中测试的方案

**package** test123;

**import** java.net.PortUnreachableException;

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** track22 {

**public** **void** generateParenthesis(**int** n) {

StringBuffer s=**new** StringBuffer();

s.append('(');

**int** num1=n-1; //左括号

**int** num2=n; //右括号

List<String> f=**new** LinkedList<>();

track(s, num1, num2,f);

**for**(**int** i=0;i<f.size();i++){

System.***out***.println(f.get(i));

}

}

**public** **void** track(StringBuffer s,**int** num1,**int** num2,List f) {

**if** (num1==0&&num2==0) {

System.***out***.println("正确序列：");

String p=s.toString();

f.add(p);

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

System.***out***.print(p.charAt(i));

}

System.***out***.println("\n");

}

// if(num1==0){

// s.append(')');

// int in=s.length();

// track(s, num1, num2-1);

// s.deleteCharAt(in-1);

// }

**if**(num1<=num2&&num1>=0){

s.append(")");

**int** l1=s.length();

track(s, num1, num2-1,f);

s.deleteCharAt(l1-1);

s.append("(");

**int** l2=s.length();

track(s, num1-1, num2,f);

s.deleteCharAt(l2-1);

}

// else {

// System.out.println("剪枝序列：");

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

// System.out.println(s.charAt(i));

// System.out.println("\n");

// }

// }

}

**public** **static** **void** main(String[] args) {

track22 p=**new** track22();

p.generateParenthesis(3);

}

}

AC答案：

public class Solution {

public List<String> generateParenthesis(int n) {

StringBuffer s=new StringBuffer();

s.append('(');

int num1=n-1; //左括号

int num2=n; //右括号

List<String> f=new LinkedList<>();

track(s, num1, num2,f);

return f;

}

public void track(StringBuffer s,int num1,int num2,List f) {

if (num1==0&&num2==0) {

String p=s.toString();

f.add(p);

}

if(num1<=num2&&num1>=0){ //逻辑要正确,可能的错误情况“())”

s.append(")");

int l1=s.length();

track(s, num1, num2-1,f);

s.deleteCharAt(l1-1); //恢复回溯环境

s.append("(");

int l2=s.length();

track(s, num1-1, num2,f);

s.deleteCharAt(l2-1);

}

}

}

### 13. Roman to Integer

思路：理解罗马数字记数方法，大的在前面则加，小的在前面则减法

public class Solution {

public int romanToInt(String s) {

Map<Character,Integer> a=new HashMap<>();

，。a.put('I', 1);

a.put('V', 5);

a.put('X', 10);

a.put('L', 50);

a.put('C', 100);

a.put('D', 500);

a.put('M', 1000);

int res=a.get(s.charAt(s.length()-1));

for(int i=s.length()-1;i>0;i--){

if(a.get(s.charAt(i))>a.get(s.charAt(i-1))){

res=res-a.get(s.charAt(i-1));

}else{

res=res+a.get(s.charAt(i-1));

}

}

return res;

}

}

### 21. Merge Two Sorted Lists

public class Solution {

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode a=l1;

ListNode b=l2;

ListNode c=new ListNode(0);

ListNode c1=c;

while(a!=null||b!=null){

if(a==null){

c.next=b;

return c1.next;

}

if(b==null){

c.next=a;

return c1.next;

}

if(a.val>b.val){

c.next=b;

b=b.next;

c=c.next; //指针移动

}else{

c.next=a;

a=a.next;

c=c.next;

}

}

if(a==null&&b==null){

return c1.next;

}

return c1;

}

}

递归：尽量减少变量

**public** ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode l3;

**if**(l1==**null**){

**return** l2;

}

**if** (l2==**null**) {

**return** l1;

}

**if**(l1.val>l2.val){

l3=l2;

l3.next=mergeTwoLists(l1, l2.next);

}**else** {

l3=l1;

l3.next=mergeTwoLists(l1.next, l2);

}

**return** l3;

}

更简便的递归：

public ListNode mergeTwoLists(ListNode l1, ListNode l2){

if(l1 == null) return l2;

if(l2 == null) return l1;

if(l1.val < l2.val){

l1.next = mergeTwoLists(l1.next, l2);

return l1;

} else{

l2.next = mergeTwoLists(l1, l2.next);

return l2;

}

}

### 24. Swap Nodes in Pairs

思路：神马时候使用递归。

1. 定义是递归的
2. 数据结构是递归的，比如链表
3. 问题的求解方法是递归的。

public class Solution {

public ListNode swapPairs(ListNode head) {

if(head==null||head.next==null) return head;

ListNode sl=head;

ListNode fa=head.next;

sl.next=fa.next;

fa.next=sl;

head=fa;

sl.next=swapPairs(sl.next);

return head;

}

}

public int strStr(String haystack, String needle) {

if(haystack==null||needle==null||haystack.length()<needle.length()) return -1;

if(needle.equals("")) return 0;

int result = 0;

for(int i=0,j=0;i<haystack.length()-needle.length()+1;i++){

if(haystack.charAt(i)==needle.charAt(j))

if(haystack.substring(i,needle.length()+i).equals(needle))

return i;

}

return -1;

}

### 15. 3Sum

思路：确定一个值进行求2sum

public class Solution {

public List<List<Integer>> threeSum(int[] nums) {

Arrays.sort(nums);

List<List<Integer>> res =new LinkedList<>();

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

if(i==0||nums[i]!=nums[i-1]){ //去掉重复的元素，确定第一个不重复

int l=i+1,r=nums.length-1,sum=0-nums[i];

while(l<r){ //一个循环求2sum

if(nums[l]+nums[r]==sum){

res.add(Arrays.asList(nums[i],nums[l],nums[r]));

while(l<r&&nums[l]==nums[l+1]) l++;

while(l<r&&nums[r]==nums[r-1]) r--;

l++;r--;

}else if (nums[l]+nums[r]<sum) l++;

else r--;

}

}

}

return res;

}

}

### 16. 3Sum Closest

思路：数组考虑two pointer，减少遍历次数，注意指针如何移动

public class Solution {

public int threeSumClosest(int[] nums, int target) {

int gap=Integer.MAX\_VALUE,result=0;

Arrays.sort(nums);

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

int j=i+1,k=nums.length-1,sum=0,cha=0;

while(j<k){

if(nums[i]+nums[j]+nums[k]==target) return target;

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

cha=Math.abs(sum-target);

if(cha<gap){

gap=cha;

result=sum;

}

if(sum<target) j++;

else k--;

}

}

return result;

}

}

### 17. Letter Combinations of a Phone Number

**思路：回溯**

**注：arrayList,List是接口，不能实例化**

**public** **class** test17 {

**public** List<String> letterCombinations(String digits) {

**int** len=digits.length();

String[] a=**new** String[]{"0","1","abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

String[] bStrings=**new** String[len];

**for** (**int** i = 0; i < len; i++) {

//System.out.println(digits.charAt(i));

bStrings[i]=a[Character.*getNumericValue*(digits.charAt(i))];

}

StringBuffer p=**new** StringBuffer();

ArrayList<String> result = **new** ArrayList<>();

**if** (digits==**null** || digits.length()==0) **return** result;

result=combine1(0,bStrings,p,result);

System.***out***.println(result);

**return** result;

}

**public** ArrayList<String> combine1(**int** k,String[] s,StringBuffer p, ArrayList<String> result) {

//List<String> r=new List<String>();

//ArrayList<String> result = new ArrayList<>();

**if**(k==s.length){

result.add(p.toString());

//System.out.println(p.toString());

**return** result;

}

**for**(**int** i=0;i<s[k].length();i++){

**int** f=p.length();

p.append(s[k].charAt(i));

result=combine1(k+1,s,p,result);

p.delete(f, f+1);

}

**return** result;

}

**public** **static** **void** main(String[] args) {

test17 rr=**new** test17();

String digits="2";

rr.letterCombinations(digits);

}

}

### 18. 4Sum

**思路：数组，两个指针，4sum变3sum，注意排除相同情况（红色标出）**

**public** List<List<Integer>> fourSum(**int**[] nums, **int** target) {

List<List<Integer>> re=**new** ArrayList<>();

Arrays.*sort*(nums);

**for**(**int** i=0;i<nums.length-3;i++){

**if(i>0&&nums[i]==nums[i-1]) continue;**

**for**(**int** j=i+1;j<nums.length-2;j++){

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

**int** sum=target-nums[i]-nums[j];

**int** m=j+1,n=nums.length-1;

**while**(m<n){

**int** tmp=nums[m]+nums[n];

**if** (tmp==sum) {

re.add(Arrays.*asList*(nums[i],nums[j],nums[m],nums[n]));

**while**(m<n&&nums[m]==nums[m+1]) m++;

**while**(m<n&&nums[n]==nums[n-1]) n--;

m++;

n--;

}**else** **if** (tmp>sum) {

n--;

}**else** {

m++;

}

}

}

}

**return** re;

}

### 31. Next Permutation

思路：找到要交换的点，然后两个指针，对后面的序列:由从大到小，变成从小到大排序

**import** java.util.Arrays;

**public** **class** test31 {

**public** **void** nextPermutation(**int**[] nums) {

**for**(**int** i=nums.length-1;i>=0;i--){

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

**int** tmp=nums[i-1];

**for**(**int** k=nums.length-1;k>i-1;k--){

**if**(nums[k]>tmp){

*swap*(nums, k, i-1);

**break**;

}

}

*rerange*(nums, i, nums.length-1);

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

System.***out***.print(nums[m]);

}

**break**;

}

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

Arrays.*sort*(nums);

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

System.***out***.print(nums[m]);

}

**break**;

}

}

}

**private** **static** **void** rerange(**int**[] s, **int** i, **int** j) {

**while**(i<j){

*swap*(s, i, j);

i++;

j--;

}

}

**private** **static** **void** swap(**int**[] s, **int** i, **int** j) {

**int** temp = s[i];

s[i] = s[j];

s[j] = temp;

}

**public** **static** **void** main(String[] args) {

test31 rr=**new** test31();

**int**[] nums=**new** **int**[]{1,3,2};

rr.nextPermutation(nums);

}

}

### 35. Search Insert Position

思路：画图发现最终指针停在的位置，减少代码量

Good！

public int searchInsert(int[] nums, int target) {

if (nums==null || nums.length==0) return -1;

int l = 0, r = nums.length-1;

int m = 0;

while (l<=r){

m = l+(r-l)/2;

if (target<nums[m]) r = m-1;

else if (nums[m]<target) l = m+1;

else return m;

}**// 如果没有找到目标元素，那么l一定停在恰好比target大的index上,r一定停在恰好比target小的index上**

return l;

}

Me！

public int searchInsert(int[] nums, int target) {

int i=0,j=nums.length-1,mid=0;

while(i<=j){

mid=(i+j)/2;

if (i==j) {

if(nums[mid]==target){

return mid;

}else if (nums[mid]<target){

return mid+1;

}else {

return mid;

}

}

if(nums[mid]==target){

return mid;

}else if (nums[mid]<target) {

i=mid+1;

}else {

j=mid-1;

}

}

return mid;

}

### 36. Valid Sudoku

思路：读懂题目

public class Solution {

public boolean isValidSudoku(char[][] board) {

for(int i=0;i<9;i++){

if (!partVaild(board, i, i,0, 8)) return false;

if (!partVaild(board, 0, 8,i, i)) return false;

}

for (int i = 0; i < 3; i++) {

for(int j=0;j<3;j++){

if (!partVaild(board,i\*3,i\*3+2,j\*3,j\*3+2)) return false;

}

}

return true;

}

public boolean partVaild(char[][] board, int x1,int x2,int y1,int y2) {

HashSet<Character > middle= new HashSet<>();

for(int i=x1;i<=x2;i++){

for(int j=y1;j<=y2;j++){

if(board[i][j]!='.')if (!middle.add(board[i][j])) return false;

}

}

return true;

}

}

### 38. Count and Say

**注：注意标红的两句，在循环结束最后一个计数没有被加入stringbuffer中**

**public** String countAndSay(**int** n) {

String tmp="1";

**for**(**int** i=1;i<n;i++){

StringBuffer ttBuffer=**new** StringBuffer();

**char** s=tmp.charAt(0);

**int** count=0;

**for** (**int** j = 0; j < tmp.length(); j++) {

**if**(tmp.charAt(j)==s){

count=count+1;

}**else**{

ttBuffer.append(count);

ttBuffer.append(s);

s=tmp.charAt(j);

count=1;

}

}

ttBuffer.append(count);

ttBuffer.append(s);

tmp=ttBuffer.toString();

}

**return** tmp;

}

### 39. Combination Sum

注：

1.trackback无返回值

2. **new** ArrayList<>(res);

**public** List<List<Integer>> combinationSum(**int**[] candidates, **int** target) {

List<List<Integer>> res = **new** ArrayList<>();

*backtrack*(target, candidates, res, **new** ArrayList<>(),0);

**return** **new** ArrayList<>(res);

}

**public** **static** **void** backtrack(**int** target,**int**[] candidates,List<List<Integer>> res,List<Integer> tmp,**int** ss) {

**if** (target==0) {

res.add(**new** ArrayList<>(tmp));

**return**;

}

**for**(**int** i=ss;i<candidates.length;i++){

**if** (target-candidates[i]>=0) {

tmp.add(candidates[i]);

*backtrack*(target-candidates[i], candidates, res,tmp,i);

tmp.remove(tmp.size()-1);

}

}

**return**;

}

### 40. Combination Sum II

思路：重点是Set中插入相同的元素的时候，该元素只存在一个。

**public** List<List<Integer>> combinationSum2(**int**[] candidates, **int** target) {

Set<List<Integer>> res = **new** HashSet<>();

Arrays.*sort*(candidates);

*backtrack*(target, candidates, res, **new** ArrayList<>(),0);

**return** **new** ArrayList<>(res);

}

**public** **static** **void** backtrack(**int** target,**int**[] candidates,Set<List<Integer>> res,List<Integer> tmp,**int** ss) {

**if** (target==0) {

res.add(**new** ArrayList<>(tmp));

**return**;

}

**for**(**int** i=ss;i<candidates.length;i++){

**if** (target-candidates[i]>=0) {

tmp.add(candidates[i]);

*backtrack*(target-candidates[i], candidates, res,tmp,i+1);

tmp.remove(tmp.size()-1);

}

}

**return**;

}

### 46. Permutations

思路：回溯方法进行排列。

深度表示排列中的第x个元素

每x的位置上有n种元素的可能，要进行交换操作

public class Solution {

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> rest=new ArrayList<>();

backtrack(nums,0,rest);

return rest;

}

public void backtrack(int[] nums,int i, List<List<Integer>> res){

if (i==nums.length){

List<Integer> sol = new ArrayList<>();

for (int x:nums) {

sol.add(x);

}

res.add(sol);

}

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

swap(nums,i,j);

backtrack(nums,i+1,res);

swap(nums,i,j);

}

}

public void swap(int[] nums,int i,int j){

int tmp=nums[i];

nums[i]=nums[j];

nums[j]=tmp;

}

}

### 47. Permutations II

**思路：注意集合的使用---除去重复的排列，continue和排序可以排除部分重复的排列**

**public** List<List<Integer>> permuteUnique(**int**[] nums) {  
 Set<List<Integer>> rest=**new** HashSet<>(); *//集合,为啥需要集合 0019，只能保证第一位不重复，不能保证后面几位不重复  
 // List<List<Integer>> rest=new ArrayList<>();* Arrays.*sort*(nums);  
 backtrack(nums,0,rest);  
 **return new** LinkedList<>(rest);  
}  
**public void** backtrack(**int**[] nums,**int** i, Set<List<Integer>> res){  
 **if** (i==nums.**length**){  
 List<Integer> sol = **new** ArrayList<>();  
 System.***out***.println();  
 **for** (**int** x:nums) {  
 sol.add(x);  
 System.***out***.print(x);  
 }  
 System.***out***.println();  
 res.add(sol);  
 }  
 **for** (**int** j=i;j<nums.**length**;j++){  
 System.***out***.print(j);  
 **if** (j>i&&nums[j]==nums[j-1]) **continue**; *//continue* swap(nums,i,j);  
 backtrack(nums,i+1,res);  
 swap(nums,i,j);  
  
  
 }  
  
}  
**public void** swap(**int**[] nums,**int** i,**int** j){  
 **int** tmp=nums[i];  
 nums[i]=nums[j];  
 nums[j]=tmp;  
}

### 100. Same Tree

思路：深度优先搜索，结束条件，左子树，右子树

class Solution {

public:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

if(p==null &&q==null) return true;

if(p==null ||q==null) return false;

if(p.val==q.val)

return isSameTree(p.left,q.left) &&isSameTree(p.right,q.right);

return false;

}

};

### 107. Binary Tree Level Order Traversal II

**思路：广度优先搜索，每层level。**

**注意：**

**1. 在list中还有list的时候，需要new LinkedList**

2.

public class Solution {

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> wrapList = new LinkedList<List<Integer>>();

levelMaker(wrapList, root, 0);

return wrapList;

}

public void levelMaker(List<List<Integer>> list, TreeNode root, int level) {

if(root == null) return;

if(level >= list.size()) {

list.add(0, new LinkedList<Integer>());

}

levelMaker(list, root.left, level+1);

levelMaker(list, root.right, level+1);

list.get(list.size()-level-1).add(root.val); 先取出来再添加

}

}

### 110. Balanced Binary Tree

public boolean isBalanced(TreeNode root) {

if(root==null){

return true;

}

return height(root)!=-1;

}

public int height(TreeNode node){

if(node==null){

return 0; //叶子结点

}

int lH=height(node.left);

if(lH==-1){

return -1; //左子树深度超过1

}

int rH=height(node.right);

if(rH==-1){

return -1;

}

if(lH-rH<-1 || lH-rH>1){

return -1;

}

return Math.max(lH,rH)+1;

}

### 108. Convert Sorted Array to Binary Search Tree

public class Solution {

public TreeNode sortedArrayToBST(int[] nums) {

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

TreeNode head= finder(nums,0,nums.length-1);

return head;

}

public TreeNode finder(int[] num,int low,int high){

if(low>high) return null;

int mid=(low+high)/2;

TreeNode node =new TreeNode(num[mid]);

node.left=finder(num,low,mid-1);

node.right=finder(num,mid+1,high);

return node;

}

}

### 111. Minimum Depth of Binary Tree

思路：如果有一棵子树是空的，那么树的深度就是非空的那棵子树

public class Solution {

public int minDepth(TreeNode root) {

if(root==null) return 0;

int left=minDepth(root.left);

int right=minDepth(root.right);

return (left==0||right==0)?left+right+1:Math.min(left,right)+1;

}

}

### 112. Path Sum

思路：与其每次计算和，不如每次减去一个值。

public class Solution {

public boolean hasPathSum(TreeNode root, int sum) {

if(root==null) return false;

if(root.left==null&&root.right==null &&sum-root.val==0) return true;

return hasPathSum(root.left,sum-root.val)||hasPathSum(root.right,sum-root.val);

}

}

### 122. Best Time to Buy and Sell Stock II

思路：贪心算法

public class Solution {

public int maxProfit(int[] prices) {

int total =0;

for (int i=0;i< prices.length-1;i++){

if( prices[i+1]>prices[i]) total+=prices[i+1]-prices[i];

}

return total;

}

}

### 118. Pascal's Triangle

1.数组使用ArrayList

public class Solution {

public List<List<Integer>> generate(int numRows)

{

List<List<Integer>> allrows = new ArrayList<List<Integer>>();

ArrayList<Integer> row = new ArrayList<Integer>();

for(int i=0;i<numRows;i++)

{

row.add(0, 1);

for(int j=1;j<row.size()-1;j++)

row.set(j, row.get(j)+row.get(j+1));

allrows.add(new ArrayList<Integer>(row));

}

return allrows;

}

### 119. Pascal's Triangle II

public List<Integer> getRow(int rowIndex) {

List<Integer> list = new ArrayList<Integer>();

if (rowIndex < 0)

return list;

for (int i = 0; i < rowIndex + 1; i++) {

list.add(0, 1);

for (int j = 1; j < list.size() - 1; j++) {

list.set(j, list.get(j) + list.get(j + 1));

}

}

return list;

}

### 125. Valid Palindrome

java.lang.Character里有相应的判断方法  
char c = 'a';  
isDigit(c)：判断字符是否数字  
isLetter(c)：判断字符是否字母   
isLetterOrDigit(c)：判断字符是否字母或数字  
isLowerCase(c)：判断字符是否小写字母  
isUpperCase(c)：判断字符是否大写字母  
isWhitespace(c)：判断字符是否空格

思路：回文字串，使用两个指针

public class Solution {

public boolean isPalindrome(String s) {

if (s.isEmpty()) {

return true;

}

int head = 0, tail = s.length() - 1;

char cHead, cTail;

while(head <= tail) {

cHead = s.charAt(head);

cTail = s.charAt(tail);

if (!Character.isLetterOrDigit(cHead)) {

head++;

} else if(!Character.isLetterOrDigit(cTail)) {

tail--;

} else {

if (Character.toLowerCase(cHead) != Character.toLowerCase(cTail)) {

return false;

}

head++;

tail--;

}

}

return true;

}

}

### 136. Single Number

思路：异或操作：相同为0，不同为1.

如果数字中有结果只出现两次，异或会全部为0

public class Solution {

public int singleNumber(int[] nums) {

int ans=0;

int len=nums.length;

for(int i=0;i!=len;i++)

ans^=nums[i];

return ans;

}

}

### 155. Min Stack

思路：

变量min记录最小值

Stack中记录x-min的差值，如果差值大于0说明x比较大，如果差值小于0说明x比较小，要更新最小值。

public class MinStack {

long min;

Stack<long> stack;

/\*\* initialize your data structure here. \*/

public MinStack() {

stack =new Stack<>();

}

public void push(int x) {

if(stack.isEmpty()){

stack.push(0L);

min=x;

}else{

stack.push(x-min);

if(x<min) min=x;

}

}

public void pop() {

if(stack.isEmpty()) return;

long pop =stack.pop();

if(pop<0) min=min-pop;

}

public int top() {

long pop=stack.peek();

if(top>0){

return (int)(top+min);

}else{

return (int)min;

}

}

public int getMin() {

return (int)min;

}

}

### 167. Two Sum II - Input array is sorted

思路：两个指针从两边开始找

public int[] twoSum(int[] num, int target) {

int[] indice = new int[2];

if (num == null || num.length < 2) return indice;

int left = 0, right = num.length - 1;

while (left < right) {

int v = num[left] + num[right];

if (v == target) {

indice[0] = left + 1;

indice[1] = right + 1;

break;

} else if (v > target) {

right --;

} else {

left ++;

}

}

return indice;

}