# 101、判断是否为对称二叉树

**boolean isSymmetric(TreeNode root)**

1 **if** root==null **return** true

2 **return** Aux(root.left,root.right)

**boolean Aux(TreeNode L,TreeNode R)**

1 **if** L==null **and** R=null **return** true

2 **if** L==null **or** R=null **return** false

3 **if** L.val≠R.val **return** false

4 **return** Aux(L.left,R.right) **and** Aux(L.right,R.left)

# 102、二叉树的层遍历

**递归**

**public List<List<Integer>> levelOrder(TreeNode root)**

1 let res be a new List<List<Integer>>

2 helper(root,1,res)

3 **return** res

**private void helper(TreeNode root,int curLevel,List<List<Integer>> res)**

1 **if** root==null **return**

2 **if** res.size()<curLevel res.add(a new ArrayList<Integer>)

3 res[curLevel].add(root.val)

4 helper(root.left,curLevel+1,res)

5 helper(root.right,curLevel+1,res)

**队列**

**public List<List<Integer>> levelOrder(TreeNode root)**

1 let res be a new List<List<Integer>>

2 let queue be a new Queue<TreeNode>

3 **if** root==null **return** res

4 queue.offer(root)

5 **while** **not** queue.isEmpty()

6 curLevelNum=queue.size()

7 let curLevel be a new ArrayList<Integer>

8 **for** i=1 **to** curLevelNum

9 peek=queue.poll()

10 **if** peek.left≠null queue.offer(peek.left)

11 **if** peek.right≠null queue.offer(peek.right)

12 curLevel.add(peek.val)

13 res.add(curLevel)

14 **return** res

# 103、二叉树的ZigZag层遍历

**递归**

**public List<List<Integer>> zigzagLevelOrder(TreeNode root)**

1 let res be a new List<List<Integer>>

2 helper(root,1,true,res)

3 **return** res

**private void helper(TreeNode root,int curLevel,boolean isOdd,List<List<Integer>> res)**

1 **if** root==null **return**

2 **if** res.size()<curLevel res.add(a new LinkedList<Integer>)

3 **if** isOdd res[curLevel].add(root.val)**//插到尾部**

4 **else** res[curLevel].add(1,root.val)**//插到头部**

5 helper(root.left,curLevel+1,**not** isOdd,res)

6 helper(root.right,curLevel+1,**not** isOdd,res)

**队列**

**public List<List<Integer>> zigzagLevelOrder(TreeNode root)**

1 let res be a new List<List<Integer>>

2 let queue be a new Queue<TreeNode>

3 **if** root==null **return** res

4 queue.offer(root)

5 isOdd=true

6 **while** **not** queue.isEmpty()

7 curLevelNum=queue.size()

8 let curLevel be a new LinkedList<Integer>

9 **for** i=1 **to** curLevelNum

10 peek=queue.poll()

11 **if** peek.left≠null queue.offer(peek.left)

12 **if** peek.right≠null queue.offer(peek.right)

13 **if** isOdd curLevel.add(peek.val) **//插到尾部**

14 **else** curLevel.add(1,peek.val) **//插到头部**

15 res.add(curLevel)

16 isOdd=**not** isOdd

17 **return** res

# 104、二叉树的最大深度

**递归**

**int maxDepth(TreeNode root)**

1 if root==null return 0

2 return max(maxDepth(root.left),maxDepth(root.right))+1

**队列**

**public int maxDepth(TreeNode root)**

1 let queue be a new Queue<TreeNode>

2 **if** root==null **return** 0

3 queue.offer(root)

4 count=0

5 **while** **not** queue.isEmpty()

6 curLevelNum=queue.size()

7 **for** i=1 **to** curLevelNum

8 peek=queue.poll()

9 **if** peek.left≠null queue.offer(peek.left)

10 **if** peek.right≠null queue.offer(peek.right)

11 count++

12 **return** count

# 105、根据前序遍历和中序遍历构建二叉树

**Map<Integer,Integer> map**//保存inorder[]中的值-索引对

**TreeNode buildTree(int[] preorder, int[] inorder)**

1 let map be a new Map<Integer,Integer>

2 **for** i=1 **to** preorder.length

3 map[inorder[i]]=i

4 **return** Aux(preorder,1,inorder,1,preorder.length)

**TreeNode Aux(int[] preorder,int prebegin,int[] inorder,int inbegin,int inend)**

1 **if** prebegin==preorder.length **return** null

2 dexOfInorder=map[preorder[prebegin]]

3 **if** dexOfInorder<inbegin **or** dexOfInorder>inend **return** null

4 let cur be a new TreeNode with val=preorder[prebegin]

5 cur.left=Aux(preorder,prebegin+1,inorder,inbegin,dexOfInorder-1)

6 cur.right=Aux(preorder,**prebegin+dexOfInorder-inbegin+1**,inorder,dexOfInorder+1,inend)

7 **return** cur

**前序遍历特点：1.访问节点cur 2.递归访问节点cur的左子树 3.递归访问cur的右子树**

**如果前序遍历中连续一段恰好能够成一颗子树，那么该段的首端点为该子树的根节点**

**中序遍历的特点：以某个节点为根节点的子树所构成的中序遍历子序列，一定位于中序遍历序列中的某连续的一段**

Aux解释：对于以preorder[prebegin]（记为cur）为根节点的子树，该子树位于中序遍历序列的inorder[inbegin...inend]位置上

cur在中序遍历序列中的位置为dexOfInorder

那么说明cur的左子树位于**inorder[instart, dexOfInorder-1]**

右子树位于**inorder[dexOfInorder+1,inend]**

**cur的左子树的大小为dexOfInorder-1-instart+1=dexOfInorder-instart**

**cur的左子树的根节点为preorder[prebegin+1]**

**cur的右子树的根节点为preorder[prebegin+dexOfInorder-instrat+1]**

# 106、根据后续遍历和中序遍历重构二叉树

**Map<Integer,Integer> map**//保存inorder[]中的值-索引对

**TreeNode buildTree(int[] inorder, int[] postorder)**

1 let map be a new Map<Integer,Integer>

2 **for** i=1 **to** inorder.length

3 map[inorder[i]=i

4 **return** Aux(postorder,postorder.length,inorder,1,inorder.length)

**TreeNode Aux(int[]postorder,int postend,int[] inorder,int inbegin,int inend)**

1 **if** postend<1 **return** null

2 dexOfInorder=map[postorder[postend]]

3 **if** dexOfInorder<inbegin **or** dexOfInorder>inend **return** null

4 let cur be a new TreeNode with val= postorder[postend]

5 cur.right=Aux(postorder,postend-1,inorder,dexOfInorder+1,inend)

6 cur.left=Aux(postorder,postend-(inend-dexOfInorder)-1,inorder,inbegin, dexOfInorder-1)

7 **return** cur

**后序遍历特点：1.递归访问节点cur的左子树 2.递归访问cur的右子树 3.问节点cur**

**如果后续遍历中连续一段恰好能够成一颗子树，那么该段的末端点为该子树的根节点**

**因此我们沿着反向的后续遍历，进行重构，首先最后一个节点为根节点**

**中序遍历的特点：以某个节点为根节点的子树所构成的中序遍历子序列，一定位于中序遍历序列中的某连续的一段**

Aux解释：对于以postorder[postend]（记为cur）为根节点的子树，该子树位于中序遍历序列的inorder[inbegin...inend]位置上

cur在中序遍历序列中的位置为dexOfInorder

那么说明cur的左子树位于**inorder[instart, dexOfInorder-1]**

右子树位于**inorder[dexOfInorder+1,inend]**

**cur的右子树的大小为inend-(dexOfInorder+1)+1=inend-dexOfInorder**

**cur的左子树的根节点为postorder[postend-1]**

**cur的右子树的根节点为postorder[postend-(inend-dexOfInorder)-1]**

# 107、二叉树的自底向上的层遍历

**递归**

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

1 let res be a new List<List<Integer>>

2 helper(root,1,res)

3 **return** res

**private void helper(TreeNode root,int curLevel,List<List<Integer>> res)**

1 **if** root==null **return**

2 **if** res.size()<curLevel res.add(1,a new ArrayList<Integer>)

3 res[res.size()-curLevel+1].add(root.val)

4 helper(root.left,curLevel+1,res)

5 helper(root.right,curLevel+1,res)

**队列**

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

1 let res be a new LinkedList<List<Integer>>

2 let queue be a new Queue<TreeNode>

3 **if** root==null **return** res

4 queue.offer(root)

5 **while** **not** queue.isEmpty()

6 curLevelNum=queue.size()

7 let curLevel be a new ArrayList<Integer>

8 **for** i=1 **to** curLevelNum

9 peek=queue.poll()

10 **if** peek.left≠null queue.offer(peek.left)

11 **if** peek.right≠null queue.offer(peek.right)

12 curLevel.add(peek.val)

13 res.add(1,curLevel)

14 **return** res

# 108、已排序数组构建平衡搜索二叉树

**TreeNode sortedArrayToBST(int[] nums)**

1 **return** helper(nums,1,nums.length)

**TreeNode helper(int[] nums,int begin,int end)**

1 **if** begin>end **return** null

2 mid=⌊begin+end⌋

3 let root be a new TreeNode with val=nums[mid]

4 root.left=helper(nums,begin,mid-1)

5 root.right=helper(nums,mid+1,end)

6 **return** root

# 109、已排序链表构建平衡搜索二叉树（随机访问很慢！）

**思路1：将链表转化为数组，然后按108进行构建**

**思路2：**

**TreeNode sortedListToBST(ListNode head)**

1 **return** helper(head,null)

**TreeNode helper(ListNode head, ListNode tail)**

1 **if** head==tail **return** null

2 slow=head ,fast=head

3 **while** fast.next≠tail **and** fast.next.next≠tail

4 fast=fast.next.next

5 slow=slow.next

6 let root be a new TreeNode with val equals slow.val

8 root.left=helper(head,slow)

9 root.right=helper(slow.next,tail)

10 **return** root

# 110、检查一棵搜索二叉树是否为平衡树（每个节点左右子树高度差不超过1）

**boolean isBalanced(TreeNode root)**

1 **if** root==null **return** true

2 **if** |Aux(root.left)-Aux(root.right)|>1 **return** false

3 **return** isBalanced(root.left) **and** is Balanced(root.right)

**int Height(TreeNode cur)**

1 **if** cur==null **return** 0

2 **return** 1+max(Height(cur.left),Height(cur.right))

**boolean isBalanced(TreeNode root)**

1 **if** root==null

2 **return** true

3 **return** height(root)≠-1 **//由于返回的是树根节点的高度，因此当有节点不满足平衡性时，需要将这个错误信息传递至根节点**

**int height(TreeNode node)//一旦发现不平衡的节点，必须将这个属性传递至根节点**

1 **if** node==null

2 **return** 0

3 LH=height(node.left)

**//当某节点左孩子有问题时，将这种错误信息传递至当前节点**

4 **if** LH==-1 **return** -1

5 RH=height(node.right)

6 **if** RH==-1 **return** -1

7 **if** |LH-RH|>1 **return** -1

8 **return** max(LH,RH)+1

# 111、求一棵树的最小深度（从根到叶的距离）

**minDepth=0**

**int minDepth(TreeNode root)**

1 **if** root==null **return** 0

2 minDepth=+∞

3 Aux(root,1)

4 **return** minDepth

**void Aux(TreeNode cur,int curDepth)**

1 **if** cur==null **return**

2 **if** cur.left==null **and** cur.right==null

3 minDepth=min(minDepth,curDepth)

4 **return**

5 **if** cur.left≠null Aux(cur.left,curDepth+1)

6 **if** cur.right≠null Aux(cur.right,curDepth+1)

# 112、判断二叉树是否存在一条从根到叶的路径，满足所有元素之和为指定的sum

# 113、求二叉树中满足路径元素之和为指定sum的所有路径

**boolean hasPathSum(TreeNode root, int sum)**

1 **if** root==null **return** false

2 **return** Aux(root,0,sum)

**boolean Aux(TreeNode cur,int presum,int sum)**

1 cursum=presum+cur.val

2 **if** cur.left==null **and** cur.right==null **and** cursum==sum **return** true

3 **if** cur.left==null **and** cur.right==null **return** false

4 **if** cur.left==null **return** Aux(cur.right,cursum,sum)

5 **if** cur.right==null **return** Aux(cur.left,cursum,sum)

6 **return** Aux(cur.left,cursum,sum) **or** Aux(cur.right,cursum,sum)

**注意点：不能以当前节点为空来判断其父节点为叶节点，一个节点只有其左右孩子都为空，才能判定其为叶节点，因此Aux的输入节点cur不能为空，否则无法判断（不传入其父节点的情况下）**

**List<List<Integer>> pathSum(TreeNode root, int sum)**

1 let Res be a new List<List<Integer>>

2 **if** root==null **return** Res

3 let Pre be a new List<Integer>

4 Aux(root,0,sum,Pre,Res)

5 **return** Res

**void Aux(TreeNode cur,int presum,int sum,List<Integer> Pre,List<List<Integer>> Res)**

1 cursum=presum+cur.val

2 **if** cur.left==null **and** cur.right==null//cur is a leaf node

3 **if** cursum=sum

4 let Cur be a new List equals to Pre

5 Cur.add(cur.val)

6 Res.add(Cur)

7 **else**//cur is not a leaf node

8 **if** cur.left≠null

9 Pre.add(cur.val)

10 Aux(cur.left,cursum,sum,Pre,Res)

11 Pre.remove(Pre.size())

12 **if** cur.right≠null

13 Pre.add(cur.val)

14 Aux(cur.right,cursum,sum,Pre,Res)

15 Pre.remove(Pre.size())

# 114、以前序遍历的顺序将二叉树改造为全右树（链表，每个节点只有右孩子）

let pre be a TreeNode

**void flatten(TreeNode root)**

1 pre=null

2 Aux(root)

**void Aux(TreeNode cur)**

1 **if** cur==null **return**

2 Left=cur.left

3 Right=cur.right

4 **if** pre==null pre=cur

5 **else** pre.right=cur

6 pre.left=null

7 pre=pre.right //这里pre与cur指向同一个对象

8 Aux(Left)//这里会改变当前cur的结构，因此在之前要保留孩子节点，即Line2、3

9 Aux(Right)

# 115、两个字符串S，T，求T在S中的所有不同子字符串的个数

**不同指：T中所有元素出现在S中的位置不同，这些元素在S中的顺序一致，但是可以间断**

**如S="abbba" T="aba" 则输出为3**

**int numDistinct(String s, String t)**

1 let M[0...s.length][0...t.length] be a new array

2 **for** i=0 **to** s.length

3 M[i][0]=1

4 **for** i=1 **to** s.length

5 **for** j=1 **to** t.length

6 **if** s[i]==t[j] M[i][j]=M[i-1][j]+M[i-1][j-1]

7 **else** M[i][j]=M[i-1][j]

8 **return** M[s.length][t.length]

**其中M[i][j]代表S[1...i]中包含T[1...j]的字串的总数，显然M[i][0]=1**

****

**①若S[i]==T[j]**

**A、S[i]属于字串中的一个元素，那么S[i]选择与T[j]对应，因此总数为M[i-1][j-1]**

**B、S[i]不属于字串中的一个元素，那么S[i]选择不与T[j]对应，因此T[j]应该与S[1...i-1]中的一个元素对应，因此此时总数为M[i-1][j]**

**②若S[i] ≠T[j]**

**此时T[j]应与S[1...i-1]中的一个对应，因此总数为M[i-1][j]**

# 116、将满二叉树同深度的孩子串联起来，从左到右的顺序

**public void connect(TreeLinkNode root)**

1 let res be a new List<List<TreeLinkNode>>

2 helper(root,1,res)

**private void helper(TreeLinkNode root,int level,List<TreeLinkNode> res)**

1 **if** root==null **return**

2 **if** res.size()<level res.add(root)

3 **else** res[level].next=root

4 res[level]=root

5 helper(root.left,level+1,res)

6 helper(root.right,level+1,res)

**public void connect(TreeLinkNode root)**

1 **if** root==null **return**

2 TreeLinkNode pre=root

3 TreeLinkNode cur=null

4 **while** pre.left≠null

5 cur=pre

6 **while** cur≠null

7 cur.left.next=cur.right

8 **if** cur.next≠null cur.right.next=cur.next.left

9 cur=cur.next

10 pre=pre.left

# 117、将非满二叉树同深度的孩子串联起来，从左到右的顺序

**void connect(TreeLinkNode root)**

1 let res be a new List<List<TreeLinkNode>>

2 helper(root,1,res)

**void helper(TreeLinkNode root,int level,List<TreeLinkNode> res)**

1 **if** root==null **return**

2 **if** res.size()<level res.add(root)

3 **else** res[level].next=root

4 res[level]=root

5 helper(root.left,level+1,res)

6 helper(root.right,level+1,res)

# 118、119、帕斯卡三角形 (a+b)n的系数

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

1 let Res be a new List<List<Integer>>

2 **if** numRows==0 **return** Res

3 let Tem be a new List<Integer>

4 Tem.add(1)

5 Res.add(Tem)

6 **for** i=1 **to** numRows-1

7 let CurLine be a new List<Integer>

8 PreLine=Res[i-1]

9 **for** j=1 **to** i+1

10 **if** j==1 **or** j==i+1 CurLine.add(1)

11 **else** CurLine.add(PreLine[j-1]+PreLine[j])

12 Res.add(CurLine)

13 **return** Res



# 120、给出一个三角形，求从顶点到底部的和最小的路径，每个位置只能去往相邻的位置

**DP1：O(n2) M[i][j]：第i行第j列的元素为终点的最小和**

**int minimumTotal(List<List<Integer>> triangle)**

1 **if** triangle==null **or** triangle.size==0 **return** 0

2 n=triangle.size

3 let M[1...n][1...n] be a new Array

4 M[1][1]=triangle[1][1]

5 **for** row=2 **to** n

6 **for** col=1 **to** row

7 **if** col==1 M[row][col]=M[row-1][col]+triangle[row][col]

8 **else** **if** col==row M[row][col]=M[row-1][col-1]+ triangle[row][col]

9 **else** M[row][col]=min(M[row-1][col], M[row-1][col-1])+ triangle[row][col]

10 minimum=-∞

11 **for** i=1 **to** n

12 minimum=min(minimum,M[n][i])

13 **return** minimum

**DP2：O(n)**

**int minimumTotal(List<List<Integer>> triangle)**

1 **if** triangle==null **or** triangle.size==0 **return** 0

2 n=triangle.size

3 let M[1...n] be a new Array

4 M[1]=triangle[1][1]

5 **for** i=2 **to** n

6 **for** j=i **downto** 1

7 **if** j==0 M[j]=M[j]+val

8 **else** **if** j==i M[j]=M[j-1]+val

9 **else** M[j]=min(M[j],M[j-1])+val

10 minimum=-∞

11 **for** i=1 **to** n

12 minimum=min(minimum,M[i])

13 **return** minimum

**DP1中M[i][j]只与M[i-1][j-1] M[i-1][j] 有关，因此不同行的M[i][j]可以共用**

**从而M[i][j]-->M[j],计算到哪一层就代表哪一层对应元素的最小和**

**每层从右向左更新M[j]，不会影响到之前元素的计算**

# 121、买卖一次的最大利润

**int maxProfit(int[] prices)**

1 **if** prices==null **or** prices.length==0 **return** 0

2 buy=1,sell=1,profits=-∞

3 **for** data=1 **to** prices.length

4 **if** prices[data]<prices[buy] buy=data

5 **if** profits<prices[data]-prices[buy]

6 sell=data

7 profits=prices[data]-prices[buy]

8 **return** profits

# 122、买卖不限次数的最大利润

**int maxProfit(int[] prices)**

1 profits=0

2 **for** i=2 **to** prices.length

3 **if** prices[i]>prices[i-1] profits=profits+prices[i]-prices[i-1]

4 **return** profits

# 123、买卖限制2次的最大利润

**int maxProfit(int[] prices)**

1 firstBuy=-∞,firstSell=0

2 secondBuy=-∞,secondSell=0

3 **for** i=1 **to** prices.length

4 **if** firstBuy<-prices[i] firstBuy=-prices[i]

**//第一次购买时的最大利润**

5 **if** firstSell<firstBuy+prices[i] firstSell=firstBuy+prices[i]

**//第一次出售时的最大利润**

6 **if** secondBuy<firstSell-prices[i] secondBuy=firstSell-prices[i]

**//第二次购买时的最大利润**

7 **if** secondSell<secondBuy+prices[i] secondSell=secondBuy+prices[i]

**//第二次出售时的最大利润**

8 **return** secondSell

# 124、一棵树的最大和路径（该路径从一个节点出发，至另一个节点，每个节点可以去往左右孩子节点以及双亲节点）

**Map<TreeNode,Integer> map//备忘录**

**int maxPathSum(TreeNode root)**

1 let map be a new Map<TreeNode,Integer>

2 **return** Independent(root)

**int Independent(TreeNode cur)//最大和路径在以cur为根节点的子树中**

1 **if** cur==null **return** **-**∞

2 left=Independent(cur.left)//最大路径和在cur的左子树中

3 right=Independent(cur.right)//最大路径和在cur的右子树中

4 cross=Cross(cur.left)+Cross(cur.right)+cur.val//包含cur的最大和路径

5 **return** max(max(left,right),cross)

**int Cross(TreeNode cur)//以cur的父节点为一个端点的最大路径**

1 **if** map.contains(cur) **return** map[cur]

2 **if** cur==null **return** 0

3 left=Cross(cur.left)

4 right=Cross(cur.right)

5 map[cur]=max(max(left,right)+cur.val,0)

6 **return** map[cur]

**改进方案：遍历该树所有节点，更新以该节点为根节点的最大路径和（包含该根节点）**

**maximum=-**∞

**int maxPathSum(TreeNode root)**

1 Aux(root)

2 **return** maximum

**int Aux(TreeNode cur)//该函数返回值是以cur为端点（根节点）的路径最大和**

1 **if** root==null **return** **-**∞

2 left=max(0,Aux(cur.left))//左子树的

3 right=max(0,Aux(cur.right))

4 maximum=max(maximum,cur.val+left+right)**//这里更新以该节点为根节点的最大路径和（包含该根节点，双边都要计算）**

5 **return** cur.val+max(left,right)**//只返回了较大的单边路径和**

# 125、合法的回文字符串（合法字符包括大小写字母以及0-9，标点符号以及其他忽略）

**boolean isPalindrome(String s)**

1 **if** s==null **or** s.length==0 **return** true

2 left=1,right=s.length

3 **while** left<right

4 **while** left<right **and** **not** IsValid(s,left) left++

5 **while** left<right **and** **not** IsValid(s,right) right--

6 **if** **not** Match(s[left++],s[right--]) **return** false

7 **return** true

**boolean IsValid(String s,int dex)**

1 num1=s[dex]-'a'

2 **if** num1≥0 **and** num1<26 **return** true

3 num2=s[dex]-'A'

4 **if** num2≥0 **and** num2<26 **return** true

5 num3=s[dex]-'0'

6 **if** num3≥0 **and** num3<10 **return** true

7 **return** false

**boolean Match(char a,char b)**

1 **if** a==b **or** (a≥65 **and** b≥65 **and** |a-b|==32) **return** true

2 **return** false

# 126、

# 127、

# 128、返回一个数组中最长连续整数的长度

**DP：不过要利用备忘录，而且需要利用Map而不是数组来减少空间使用量**

**public int longestConsecutive(int[] num)**

1 res=0

2 let map be a new Map<Integer,Integer> **//包含关键字的最长长度**

3 **for** n:num

4 **if** map.contains(n) **continue**

5 left=map.contain(n-1)?map.get(n-1):0

6 right=map.contain(n+1)?map.get(n+1):0

7 map[n]=left+right+1

8 res=max(res,map[n])

9 map[n-left]=map[n]**//只需要改变端点处即可，区间[n-left...n+right]被再次访问时，只有端点处的值才会被利用，其他时候都会continue**

10 map[n+right]=map[n]

11 **return** res

**public int longestConsecutive(int[] nums)**

1 let set be a new Set<Integer>()

2 **for** **each** num:nums

3 set.add(num)

4 res=0

5 **for** **each** num:set

6 **if** not set.contains(num-1)

**//只对统计以num为起点的最长序列，因为会遍历每个元素，因此必然会包含那个最长序列的开头元素，因此没有必要向两边遍历。**

7 m=num+1

8 **while** set.contains(m)

9 m++

10 res=max(res,(m-1)-num+1)

11 **return** res

# 129、从根到叶的最大和，每下一层增大10倍然后加上当前节点的值

**sum=0**

**int sumNumbers(TreeNode root)**

1 **if** root==null **return** 0

2 sum=0

3 Aux(root,0)

4 **return** sum

**void Aux(TreeNode cur,int sumcur)**

1 **if** cur.left==null **and** cur.right==null

2 sum=sum+sumcur\*10+cur.val

3 **return**

4 **if** cur.left==null Aux(cur.right,sumcur\*10+cur.val)

5 **elseif** cur.right==null Aux(cur.left,sumcur\*10+cur.val)

6 **else** Aux(cur.right,sumcur\*10+cur.val)

7 Aux(cur.left,sumcur\*10+cur.val)

# 130、吃子,将被'X'包围的'O'改为'X'

**void solve(char[][] board)**

1 **if** board==null **or** board.length<3 **or** board[1].length<3 **return**

2 row=board.length

3 col=board[1].length

**//由边界的'O'向内探测，将这些相连的'0'联通区域改为'1'，这些区域是无法被X包围的，因为处于边界**

4 **for** i=1 **to** row

5 check(board,i,0,row,col)

6 check(board,i,col-1,row,col)

7 **for** j=1 **to** col

8 check(board,0,j,row,col)

9 check(board,row-1,j,row,col)

10 **for** i=1 **to** row

11 **for** j=1 **to** col

12 **if** board[i][j]=='O' board[i][j]='X'

13 **for** i=0 **to** row

14 **for** j=0 **to** col

15 **if** board[i][j]=='1' board[i][j]='O'

**check(char[][] board,int i,int j,int row,int col)**

1 **if** board[i][j]='O'

2 board[i][j]='1'

//向非边界的四周继续探测

3 **if** i>1 check(board,i-1,j,row,col)

4 **if** j>1 check(board,i,j-1,row,col)

5 **if** i+1<row check(board,i+1,j,row,col)

6 **if** j+1<col check(board,i,j+1,row,col)

# 131、字符串的所有回文子序列组abb->a b b a bb

**回溯法**

**public List<List<String>> partition(String s)**

1 let Res be a new List<List<String>>

2 let Pre be a new ArrayList<String>

3 Aux(s,0,Pre,Res)

4 **return** Res

**void Aux(String s,int dex,ArrayList<String> Pre,List<List<String>> Res)**

1 **if** dex≥s.length

2 let Cur be a new ArrayList<String> equals to Pre

3 Res.add(Cur)

4 **return**

5 **for** i=dex **to** s.length

6 **if** IsPalindrome(s,dex,i)

7 Pre.add(s[dex...i+1])

8 Aux(s,i+1,Pre,Res)

9 Pre.remove(Pre.size())

**boolean IsPalindrome(String s,int start,int end)**

1 **if** start==end **return** true

2 **while** start<end

3 **if** s[start++]≠s[end--] **return** false

4 **return** true

# 132、将字符串分割成回文字串所需要的最少切割次数

**双重动态规划**

**同时计算M与IsPalindrome**

**int minCut(String s)**

1 let M[1...s.length] be a new array

2 let IsPalindrome[1...s.length][1...s.length] be a new array

3 **for** i=1 **to** s.length

4 M[i]=i-1

5 **for** j=1 **to** i

6 **if** s[j]==s[i] **and** (j+1>i-1 **or** IsPalindrome[j+1][i-1]

7 IsPalindrome[j][i]=true

8 M[i]=(j==0? 0: min(M[i],M[j-1]+1))

9 **return** M[s.length]

# 133、克隆无向图

**DFS: depth first search**

**Map<Integer,UndirectedGraphNode> map**

**UndirectedGraphNode cloneGraph(UndirectedGraphNode node)**

1 let map be a new Map<Integer,UndirectedGraphNode>

2 **return** Aux(node)

**UndirectedGraphNode Aux(UndirectedGraphNode node)**

1 **if** node==null **return** null

2 **if** map.contain(node.label) **return** map[node.label]

3 let curCopy be a new UndirectedGraphNode with same label as node and a empty adjacent List

4 map[node.label]=curCopy

5 **for** UndirectedGraphNode n:node.neighbors

6 curCopy.neighbors.add(Aux(n))

7 **return** curCopy

**BFS: breadth first search**

**UndirectedGraphNode cloneGraph(UndirectedGraphNode node)**

1 **if** node==null **return** null

2 let Copynode be a new UndirectedGraphNode with same label as node and a empty adjacent List

3 let queue be a new Queue stored UndirectedGraphNode

4 let map be a new Map<Integer,UndirectedGraphNode>

5 map[Copynode.label]=Copynode

6 queue.offer(node)

7 **while** **not** queue.isEmpty()

8 cur=queue.poll()

9 **for** neighbor:cur.neighbors//遍历当前cur的邻接链表

10 **if** not map.contain(neighbor)//若该元素尚未添加到新的图中，则进行添加

11 map[neighbor.label]= a new UndirectedGraphNode with same label as neighbor and a empty adjacent List

12 queue.add(neighbor)

13 map[cur.label].neighbors.add(map[neighbor.label])//添加邻接链表中的该元素

14 **return** Copynode

# 134、环形加油站问题

**int canCompleteCircuit(int[] gas, int[] cost)**

1 start=0

2 total=0//存储总的欠下的油量

3 tank=0//邮箱剩余油量

4 **for** i=1 **to** gas.length

5 tank=tank+gas[i]-cost[i]

6 **if** tank<0

7 start=i+1**//由于此时从上一个起点开始假设为k（开始时油箱为空）无法到达加油i+1，那么即便从k+1开始也一定无法到达i+1，因为从k到k+1必然会有油量剩余，最少为0，必然不会比从k+1起始开的更近，因此递归推断得到无论从k...i的哪个加油站开始，都无法到达i+1**

8 total=total+tank//保存从第一个加油站到目前所欠下的油量

9 tank=0

10 **return** (total+tank<0)? -1:start//如果邮箱中剩余的油量不足以弥补之前欠下的油量，那么无法循环

# 135、分糖果，每人至少一颗，相邻的小朋友，等级高的必须必等级少的至少多一颗

**int candy(int[] ratings)**

1 n=ratings.length

2 **if** n≤1 **return** n

3 let Candy[1...n] be a new array stored the number of candy each child get

4 **for** i=1 **to** n//首先每人发一颗糖

5 Candy[i]=1

6 **for** i=2 **to** n//从左到右满足性质

7 **if** Candy[i]>Candy[i-1]

8 Candy[i]=Candy[i-1]+1

9 **for** i=n-1 **downto** 1//从右到左满足性质

10 **if** Candy[i]>Candy[i+1]

11 Candy[i]=max(Candy[i+1]+1,Candy[i]);

12 res=0

13 **for** i=1 **to** n

14 res+=Candy[i]

15 **return** res

# 136、数组中找出只出现一次的数（其余均出现了两次）

**int singleNumber(int[] nums)**

1 curnum=nums[1]

2 **for** i=1 **to** nums.length

3 curnum=curnum **BitNotOr** nums[i]

4 **return** curnum

**137、数组中找出只出现1次或2此的数（其余均出现了3次）**

**int singleNumber(int[] nums)**

1 len=nums.length

2 res=0

3 **for** i=1 **to** 32 //整型的32位bit，从低位到高位

4 sum=0

5 **for** j=1 **to** len

6 sum=sum+(nums[j]>>i)&1//将第j个数的第i位bit移到最低位，与0x 00000001与运算，然后累计到sum中，sum存储的就是当前第i位出现次数

7 res=res |( sum%3)<<i

8 **return** res

**与上述思路相似，更为简洁的一种写法，只需遍历一次，而非32次**

**以这种思路分析正确重复次数为456...可能会稍显复杂，但上一中思路清晰，可以适应任何情况**

**int singleNumber(int[] nums)**

1 CntOne=0

2 CntTwo=0

3 **for** i=1 **to** nums.length

4 CntOne=(**~** CntTwo) **&** (CntOne **^** nums[i])

5 CntTwo=(**~** CntOne) **&** (CntTwo **^** nums[i])

6 **return** CntOne|CntTwo

由于模3的状态只有3种，用二进制bit可以表示为00->01>10>00...

CntOne代表两个bit中的低位bit,CntTwo 代表两个bit中的高位bit

**现取出32位中的第i位来分析，CntOne和CntTwo的第i位记录该位1出现的次数（状态）**

若初始状态为00，该位又来一个1，CntOne变为1，CntTwo变为0，计数状态变为01

若初始状态为01，该位又来一个1，CntOne变为0，CntTwo变为1，计数状态变为10

若初始状态为10，该位又来一个1，CntOne变为0，CntTwo变为0，计数状态变为00

**因此最后的状态要么为01 要么为 10，故返回CntOne|CntTwo**

# 138、复制一个包含随机指针的链表

**RandomListNode copyRandomList(RandomListNode head)**

1 **if** head=null **return** null

2 iter=head

//copy every node,and insert just behind the original one, and link them together

3 **while** iter≠null

4 tem=iter.next

5 let copy be a new RandomListNode with same label as iter

6 iter.next=copy

7 copy.next=tem

8 iter=tem

9 iter=head

//copy every node,and insert just behind the original one, and link them together

10 **while** iter≠null

11 **if** iter.random!=null

12 iter.next.random=iter.random.next

13 iter=iter.next.next

14 copyhead=head.next

15 iter=head, itercopy=copyhead

//extract the whole copy link

16 **while** iter≠null

17 iter.next=iter.next.next

18 **if** itercopy.next≠null itercopy.next=itercopy.next.next //pay attention!,details

19 iter=iter.next

20 itercopy=itercopy.next

21 **return** copyhead

# 139、字符串是否可以分解为单词表中的单词

**boolean wordBreak(String s, Set<String> wordDict)**

1 let M[1...s.length] be a new array stored bool

2 **for** i=1 **to** s.length

3 **if** wordDict.contains(s[0...i]) M[i]=true

4 **else**

5 **for** j=1 **to** i-1

6 **if** not M[j] continue

7 **if** wordDict.contains(s[j+1...i])

8 M[i]=true

9 **break**

10 **return** M[s.length]

# 140、求字符串可能的分解方式

# 141、判断链表是否存在环状结构（不用额外空间）

**boolean hasCycle(ListNode head)**

1 slow=head,fast=head

2 **while** fast!=null **and** fast.next!=null//不同速度遍历链表的判断条件

3 fast=fast.next.next

4 slow=slow.next

5 **if** fast=slow **return** true//fast追上了slow

6 **return** false

# 142、返回链表环状结构的的入口（若非环状，则返回null）

**ListNode detectCycle(ListNode head)**

1 slow=head,fast=head,Entry=head

2 **while** fast≠null **and** fast.next≠null

3 fast=fast.next.next

4 slow=slow.next

5 **if** fast==slow//存在环状结构

6 **while** Entry≠slow

7 slow=slow.next

8 Entry=Entry.next

9 **return** Entry

10 **return** null

fast与slow相遇时，fast走过的距离是slow的两倍，并且slow尚未完成一圈

设从head到入口的距离为L，从Entry到Entry的距离为L1

**由于fast走过的距离为slow的两倍，那么从Encounter position继续行走（出发时n=0）第n（n≥1）次到Encounter position的路程为L+L1，因此从Encounter position第n次到达Entry的路程为L。**

**因此当slow经过L会到达Entry，head经过L会到达Entry**



# 143、重排链表，依次为 1->n->2->n-1->3->n-2>...

**O(n)的空间：将所有节点存在数组中，然后按要求重连**

**O(1)空间：**

**void reorderList(ListNode head)**

1 **if** head==null **or** head.next==null **return**//长度小于2的链表重排之后与重排之前相同

2 iter1=head,iter2=head

**//find the middle of the list**

3 **while** iter2.next≠null **and** iter2.next.next≠null//找中点的常用判断条件

4 iter1=iter1.next

5 iter2=iter2.next

6 mid=iter1//链表长为奇数时，指向中间元素，偶数时，指向第一段尾元素

7 cur=iter1.next//无论链表长为奇数或偶数，都会指向第二段的头元素

//Reverse the half after middle 1->2->3->4->5->6 to 1->2->3->6->5->4

8 **while** cur.next≠null

9 tem=cur.next

10 cur.next=tem.next

11 tem.next=mid.next

12 mid=next=tem

13 iter1=head

14 iter2=mid.next

//reorder one by one 1->2->3->6->5->4 to 1->6->2->5->3->4

15 **while** iter1≠mid

16 mid.next=iter2.next

17 iter2.next=iter1.next

18 iter1.next=iter2;

19 iter1=iter2.next

20 iter2=mid.next

Line 8-14: 1->2->3->4->5->6 TO 1->2->3->5->4->6 TO 1->2->3->6->5->4

# 144、145 前序后续遍历，递归以及非递归算法

**前序遍历：**

**List<Integer> preorderTraversal(TreeNode root)**

1 let Res be a new List<Integer>//stored the value of each node

2 let S be a Stack stored TreeNode

3 cur=root

4 **while** cur≠null **or** not S.isEmtpy()

5 **while** cur≠null

6 Res.add(cur.val)

7 S.push(cur)

8 cur=cur.left

9 **if** **not** S.isEmtpy()

10 cur=S.pop().right

11 **return** Res

**中序遍历：**

**List<Integer> InorderTraversal(TreeNode root)**

1 let Res be a new List<Integer>//stored the value of each node

2 let S be a Stack stored TreeNode

3 cur=root

4 **while** cur≠null **or** not S.isEmtpy()

5 **while** cur≠null

6 S.push(cur)

7 cur=cur.left

8 **if** **not** S.isEmtpy()

9 cur=S.pop()

10 Res.add(cur.val)

11 cur=cur.right

11 **return** Res

**后序遍历：**

**List<Integer> preorderTraversal(TreeNode root)**

1 let Res be a new List<Integer>//stored the value of each node

2 let S be a Stack stored TreeNode

3 cur=root

4 **while** cur≠null **or** not S.isEmtpy()

5 **while** cur≠null

6 Res.add(1,cur.val)

7 S.push(cur)

8 cur=cur.right

9 **if** **not** S.isEmtpy()

10 cur=S.pop().left

11 **return** Res

# 146、缓存实现（LRU）（实现一个类）

**由于当对一个对象进行操作，读取、写入、或者更新的时候，会将其活跃度提到最高，首先想到的是优先队列，问题是：用什么作为优先队列的关键字**

**换一种思路：链表，可以很好的解决这个问题，将每次读取、写入或者更新时都讲该节点插入到链表的头部（复杂度为O(1),而且非常简单）**

class LRUCache{

**class Node**{

int value,**key//key仅用于协助删除least recent used 元素**

Node pre,next

**Node(int key,int value){**

this.key=key

this.value=value

}

**int capacity**

**Node head,tail**//哨兵头尾节点很方便！

**Map<Integer,Node> map**  
 **LRUCache(int capacity)**

**int get(int key)**

**void set(int key,int value)**

**update(Node cur)**

**delete(Node cur)**

**headinsert(Node cur)**

}

# 147、链表插入排序的实现

**ListNode insertionSortList(ListNode head)**

1 **if** head==null **or** head.next==null **return** head

2 let pseudohead be a new ListNode

3 pseudohead.next=head//设置伪头，避免讨论

4 tail=head//已排序部分的尾节点

5 iter1=head.next

6 iter2=iter3=null//指向未排序部分的首节点

7 tail.next=null

8 **while** iter1≠null

9 tem=iter1.next

10 iter2=pseudohead.next

11 iter3=pseudohead

//找到iter1应该插入的位置：iter3之后，iter2之前

12 **while** iter2≠null **and** iter1.val>iter2.val

13 iter3=iter2

14 iter2=iter2.next

15 iter3.next=iter1

16 iter1.next=iter2

17 iter1=tem

18 **return** pseudohead

# 148、O(nlogn)的链表排序实现（利用归并排序）

**ListNode sortList(ListNode head)**

1 **return** Aux(head,null)[1]

**ListNode[] Aux(ListNode head,ListNode tail)** //返回已排序的头尾节点

1 **if** head==tail **or** head.next==tail **return** {head,tail}

2 slow=head,fast=head

3 **while** fast.next≠null **and** fast.next.next≠null

4 slow=slow.next

5 fast=fast.next.next

6 mid=slow.next

7 left=Aux(head,mid)

8 right=Aux(mid,tail)

9 **return** Merge(left,right)

**ListNode[] Merge(ListNode[] left,ListNode[] right)**

1 iter1=left[1],tail1=left[2]

2 iter2=right[1],tail2=right[2]

3 let pseudohead be a new ListNode

4 iter= pseudohead

5 while iter1≠tail1 and iter2≠tail2

6 if iter1.val<iter2.val

7 iter.next=iter1

8 iter1=iter1.next

9 else iter.next=iter2

10 iter2=iter2.next

11 iter=iter.next

12 while iter1≠tail1

13 iter.next=iter1

14 iter1=iter1.next

15 iter=iter.next

16 while iter2≠tail2

17 iter.net=iter2

18 iter2=iter2.next

19 iter=iter.next

20 iter.next=null

21 return {pseudohead.next,null}

# 150、实现四则运算（不用考虑优先级，略简单）

**{"4", "13", "5", "/", "+"}=4+13/5=6**

**{"2", "1", "+", "3", "\*"}=(2+1)\*3**

**set={"+","-","\*","/"}**

**int evalRPN(String[] tokens)**

1 **if** tokens==null **or** tokens.length==0 **return** 0

2 **if** tokens.length==1 **return** StringtoInt(tokens[1])

3 **return** Aux(tokens,tokens.length)

**int[] Aux(String[] tokens,int dex)**

1 **if** set.contains(tokens[dex-1]) //右运算子又是一个表达式

2 output=Aux(tokens,dex-1)

3 right=output[1]

4 rightbegin=output[2]

5 **else** right=StringtoInt(tokens[dex-1]) //右运算子只是一个数字

6 rightbegin=dex-1

7 **if** set.contains(tokens[rightbegin-1])//左运算子又是一个表达式

8 output=Aux(tokens,rightbegin-1)

9 left=output[1]

10 leftbegin=output[1]

11 **else** left=StringtoInt(tokens[rightbegin-1])

12 leftbegin=rightbegin-1

13 res=0

14 **switch**(tokens[dex])

15 **case** "+": res=left+right **break**

16 **case** "-": res=left-right **break**

17 **case** "\*": res=left\*right **break**

18 **case**"/": res=left/right **break**

19 **default**: **break**

20 **return** {res,leftbegin}

# 151、反序输出单词 ("AB DCE FWE"-->"FWE DCE AB")

**String reverseWords(String s)**

1 dex=0,begin=0,end=0

2 let S be a Stack<String>

3 **while** dex≤s.length()

4 **while** dex≤s.length() **and** s[dex]==' ' dex++

5 **if** dex==s.length() **break**

6 **while** dex≤s.length() **and** s[dex]≠' ' dex++

7 end=dex

8 S.push(s[begin...end-1])

9 let sb be a new StringBuilder

10 **while** **not** S.IsEmpty()

11 sb.append(S.pop()+" ")

12 **if** sb.length>0 sb.delete(sb.length-1)

13 **return** sb.toString()

# 152、最大子数组积

**M1[i]：以nums[i]结尾的子数组的最大积**

**M2[i]：以nums[i]结尾的子数组的最小积**

**int maxProduct(int[] nums)**

1 **if** nums==null **or** nums.length==0 **return** 0

2 let M1[1...nums.length] be a new array

3 let M2[1...nums.length] be a new array

6 maximum=M1[1]=M2[1]=nums[1]

7 **for** i=2 **to** nums.length

8 **if** nums[i]==0 M1[i]=M2[i]=0

9 **else** **if** nums[i]>0

10 **if** M1[i-1]≤0 M1[i]=nums[i]

11 **else** M1[i]=M1[i-1]\*nums[i]

12 **if** M2[i-1]≥0 M2[i]=nums[i]

13 **else** M2[i]=M2[i-1]\*nums[i]

14 **else**

15 **if** M2[i-1]≥0 M1[i]=nums[i]

16 **else** M1[i]=M2[i-1]\*nums[i]

17 **if** M1[i-1]≤0 M2[i]=nums[i]

18 **else** M2[i]=M1[i-1]\*nums[i]

19 maximum=Math.max(maximum,M1[i])

19 **return** maximum

# 153、154、有序链表经过一次旋转（[1....n]变为 [m+1...n 1...m]）找出最小值

**int findMin(int[] nums)**

1 dex=0

2 **while** dex≤nums.length

3 **if** dex+1≤nums.length and nums[dex]>nums[dex+1]

4 **return** nums[dex+1]

5 dex++

6 **return** nums[1]

# 155、实现最小栈（既能满足一般栈的压入弹出，返回栈顶元素值的功能，又能返回最小值的功能）

**class MinStack**{

class Node{

int val

int min//存储以该节点作为表头时的最小值

Node next

Node(int val){this.val=val;}

}

Node head=new Node(0)//pseudohead

void push(int x){

Node cur=new Node(x)

Node next=head.next

head.next=cur

cur.next=next

cur.min=min(x,next==null? +∞:next.min)

}

public void pop(){head.next=head.next.next}

public int top(){**return** head.next.val}

public int getMin(){**return** head.next.min}

}

法二：

private class MinStackNode{

int value;

MinStackNode pre;

MinStackNode next;

**MinStackNode large;//若该节点为最小节点，那么large指向上一个最小节点，即该节点压入前的最小节点，若该节点不为最小节点，该字段为null**

MinStackNode(int value){this.value=value;}

}

# 160、找到两个链表合并之处

**ListNode getIntersectionNode(ListNode headA, ListNode headB)**

1 lenA=0,lenB=0

2 iterA=headA,iterB=headB

3 **while** iterA≠null

4 iterA=iterA.next

5 lenA++

6 **while** iterB≠null

7 iterB=iterB.next

8 lenB++

9 iterA=headA,iterB=headB

10 **while** iterA≠null **or** iterB≠null

11 **if** lenA<lenB

12 iterB=iterB.next

13 lenA++

14 **elseif** lenA<lenB

15 iterA=iterA.next

16 lenB++

17 **else**

18 **if** iterA==iterB **return** iterA

19 iterB=iterB.next

20 iterA=iterA.next

21 **return** null

# 162 找到数组中的峰值，即该元素比其邻元素要大

**int findPeakElement(int[] nums)**

1 **if** nums==null **or** nums.length<2 return 0

2 left=0,right=nums.length

3 **while** left<right

4 mid= ⌊left+right/2⌋

5 **if** nums[mid]<nums[mid+1] left=mid+1

6 **else** right=mid

7 **if** mid==nums.length **or** nums[mid]>nums[mid+1] return mid

8 **return** mid+1

# 164、找到数字组中最大的gap

**int maximumGap(int[] num)**

1 **if** nums.length<2 return 0

2 max=-∞,min=+∞

3 for i:num

4 max=max(max,i)

5 min=min(min,i)

6 **if** max==min return 0

7 interval=⌈(max-min)/(num.length-1)⌉

8 let BucketMin[1...num.length] be a new array initialized to +∞

9 let BucketMax[1...num.length] be a new array initialized to -∞

10 for i:num

11 dex=⌊(i-min)/interval ⌋

12 BucketMin[dex]=min(BucketMin[dex],i)

13 BucketMax[dex]=max(BucketMax[dex],i)

14 previous=BucketMax[1]//不可能为+∞，因为至少包含min

15 gap=interval//gap no smaller than interval

16 **for** i=2 **to** num.length

17 **if** BucketMin[i]==+∞ **and** BucketMax[i]== -∞ **continue**//跳过空桶

18 cur=BucketMin[i]

19 gap=max(gap,cur-previous)

20 previous=BucketMax[i]

21 **return** gap

interval=⌈ (max-min)/(n-1)⌉

第i个桶的范围：**[**min+(i-1)\*interval,min+i\*interval**)**  i from 1 to n

**为何是n：由于max处于边界位置，即使向上取整，max仍有可能处于边界位置，因此桶的数量是n个（第n个桶只可能放值等于max的元素）**

**为什么gap必然不小于interval?**由于interval= interval=⌈ (max-min)/(n-1)⌉,那么小于interval的必然是gap=⌈ (max-min)/(n-1)⌉-1≤⌊(max-min)/(n-1)⌋,由于总共有n个元素，因此max≤gap\*(n-1)

max≤⌊(max-min)/(n-1)⌋\*n-1≤max（特殊情况等号才成立，显然矛盾）

# 165、比较版本数字

"1.0"="1" "1.0.0.1">"1.0" "2.5.2">"1.9"

**int compareVersion(String version1, String version2)**

1 dex1=1,dex2=1

2 **while** dex1≤version1.length **or** dex2≤version2.length

3 num1=0,num2=0

4 **while** dex1≤version1.length **and** version1[dex1]≠'.'

5 num1=num1\*10+(version1[dex1]-'0')

6 **while** dex2≤version2.length **and** version2[dex2]≠'.'

7 num2=num2\*10+(version2[dex]-'0')

8 **if** num1>num2 return 1

9 **if** num1<num2 return -1

10 dex1++,dex2++

11 **return** 0

# 166、给定分子分母，写出小数的形式，循环用()表示

**String fractionToDecimal(int numerator, int denominator)**

1 **if** numerator==0 return "0"

2 let Res be a new StringBuilder

3 Res.append(((numerator>0)^(denominator>0))?"-":"")//相乘判断正负会移除

4 long n=|numerator|//防止转为正数时溢出，因为最小负数比最大正数的绝对值大1

5 long d=|denominator|

6 Res.append(n/d)//integral part

7 long remain=n%d

8 **if** remain==0 **return** Res.toString()

9 Res.append('.')

10 let map be a new Map<Long,Integer>

11 **while** remain≠0

12 **if** map.containsKey(remain) break

13 **else** map.put(remain,Res.length()+1)

14 remain=remain\*10

15 Res.append(remain/d)

16 remain=remain%d

17 **if** remain≠0

18 Res.insert(map.get(remain),"(")

19 Res.append(")")

20 **return** Res.toString()

# 168、数字转为Excel表格的编号

**String convertToTitle(int n)**

1 let Res be a new StringBuilder

2 **while** n>0

3 n--

4 Res.insert(0,n%26+'A')

5 cur=⌊n/26⌋

6 **return** Res.toString(

**本质就是十进制转化为26进制，但是对应关系有偏移，原本应该是0-25对应A-Z，现在是1-26，因此每一位的对应关系都发生了偏移，将每一位减一取消偏移，就能对应取模的关系**

**一般的M进制数，除了最高位可以有M种情况(0...M-1)，最高位只有M-1种情况(1...M-1)**

**而此处的26进制，每一位都可以有26种情况**

**另外：若要使得2-A 3-B ..... 27-Z 28AA 这种为整体偏移，只需要在开头取消偏移即可，每位的偏移仍然是1**

# 171、Excel编号转化为数字

**int titleToNumber(String s)**

1 int res=0

2 for i=1 to s.length

3 res=res\*26+(s[i]-'A'+1)

4 return res

# 169、找出数组中的主元（出现次数多余一半）

**int majorityElement(int[] nums)**

1 res=nums[1]

2 count=1

3 **for** i=2 **to** nums.length

4 **if** res==nums[i] ++count

5 else --count

6 **if** count==0

7 res=nums[i]

8 count=1

9 **return** res

**所有被设定为主元的元素，若它并非主元，必然会进入Line6-Line8，将其替换为下一个假定的主元，最终保留的一定是真正的主元**

# 172、判断n的阶乘结果尾部的0的个数

**出现一次因子5就多一次**

**int trailingZeroes(int n)**

1 long m=5//防止Line5溢出

2 res=0

3 **while** n≥m

4 res=res+n/m

5 m=m\*5

6 **return** res

# 173、搜索二叉树的迭代器

**将栈的中序遍历过程进行拆分**

public class BSTIterator {

Stack<TreeNode> s = new Stack<TreeNode>();

public BSTIterator(TreeNode root) {

helper(root);

}

public boolean hasNext() {

return !s.isEmpty();

}

public int next() {

TreeNode cur = s.pop();

helper(cur.right);

return cur.val;

}

public void helper(TreeNode t) {

while (t != null) {

s.push(t);

t = t.left;

}

}

}

# 174、骑士救公主，最少需要多少生命值

**int calculateMinimumHP(int[][] dungeon)**

1 **if** dungeon==null **or** dungeon.length==0 **or** dungeon[1].length==0

2 **throw** Exception

3 row=dungeon.length,col=dungeon[1].length

4 let Health[1...row][1...col] be a new Array

5 Health[row][col]=max(1-dungeon[row][col],1)

6 **for** i= row-1 **downto** 1

7 Health[i][col]=max(Health[i+1][col]-dungeon[i][col],1)

8 **for** i=col-1 **downto** 1

9 Health[row][i]=max(Health[row][i+1]-dungeon[row][i],1)

10 **for** i=row-1 **downto** 1

11 **for** j=col-1 **downto** 1

12 Health[i][j]=min(

max(Health[i+1][j]-dungeon[i][j],1),max(Health[i][j+1]-dungeon[i][j],1)

13 **return** Health[1][1]

核心递归式:

Health[i][j]=min(max(Health[i+1][j]-dungeon[i][j],1),max(Health[i][j+1]-dungeon[i][j],1)

**从i+1,j到达终点**，最少需要Health[i+1][j]点生命值，在(i,j)损耗的生命值为dungeon[i][j],因此**从(i,j)经过(i+1,j)到达终点，**最少需要max(Health[i+1][j]-dungeon[i][j],1)点生命值，同理，从(i,j)经过(i,j+1)到达终点，最少需要max(Health[i][j+1]-dungeon[i][j],1)点生命值

# 179、将一堆数组串联成一个最大的数字

**问题的关键，如何对不同长度的数字进行排序**

**String largestNumber(int[] nums)**

1 let Ary[1...nums.lenght] be a new array stored String

2 for i=1 to nums.length

3 Ary[i]=Integer.toString(nums[i])

4 QuickSort(Ary)

5 if Ary[1][1]=='0' return 0 //boundary condition

6 let Res be a new StringBuilder

7 for String s:Ary

8 Res.append(s)

9 return Res.toString()

**int Compare(String s1,String s2)**

1 Res1=s1+s2

2 Res2=s2+s1

3 return Res1<Res2?-1:(Res1==Res2?0:1)

# 187、重复的DNA序列

**List<String> findRepeatedDnaSequences(String s)**

1 let set and repeat be new Set<String>

2 **for** i=1 **to** s.length-9

3 cur=s[i...i+9]

4 **if** **not** set.add(cur)

5 repeat.add(cur)

6 **return** a new List<String> equals torepeat

# 188、买卖k次的最大利润

**int maxProfit(int k, int[] prices)**

1 **if** k<1 return 0

2 **if** k>prices.length/2//实际上最大的买卖次数不会超过该值

3 Maxprofit=0

4 **for** i=2 **to** prices.length

5 **if** prices[i]>prices[i-1]

6 Maxprofit+=prices[i]-prices[i-1]

7 **return** Maxprofit

8 let Buy[1...k] be a new Array initialized to -∞

9 let Sell[1...k] be a new Array initialized to 0

10 **for** day=1 **to** prices.length

11 **for** i=1 **to** k

12 **if** i==1 Buy[i]=max(Buy[i],0-prices[day])

13 **else** Buy[i]=max(Buy[i],Sell[i-1]-prices[day])

14 Sell[i]=max(Sell[i],Buy[i]+prices[i])

15 **return** Sell[k]

**DP:M[0...k][1...n]**

**M[i][j] represents the max profit up until prices[j] using at most i transcation**

**int maxProfit(int k, int[] prices)**

1 n=prices.length

2 **if** k>prices.length/2//实际上最大的买卖次数不会超过该值

3 Maxprofit=0

4 **for** i=2 **to** prices.length

5 **if** prices[i]>prices[i-1]

6 Maxprofit+=prices[i]-prices[i-1]

7 **return** Maxprofit

8 let M[0...k][1...n] be a new Array initialized to zero

9 **for** i=1 **to** k

10 localMax=M[i-1][1]-price[1]

11 **for** j=2 **to** n

12 M[i][j]=max(M[i][j-1],prices[j]+localMax)

13 localMax=math.max(localMax,M[i-1][j]-prices[j])

14 **return** M[k][n]

localMax 代表i次买入，i-1次卖出的最大收益（随着day进行迭代）

M[i][j]=max(M[i][j-1],prices[j]+localMax)

M[i][j-1]：在第j-1日前（包括第j-1日）最多进行i次买卖的最大收益**（即第j日不交易）**

prices[j]+localMax：在j-1日之前（包括j-1日）进行i此买入，i-1次卖出的最大收益**外加**在第j日卖出的最大收益**（即在第j日进行卖出）**

localMax=math.max(localMax,M[i-1][j]-prices[j]) **更新lacalMax**

M[i-1][j]-prices[j]：在j日之前进行i-1次买卖外加一次买入的最大收益

# 189、数组向右循环以为k位

**O(n)complexity，O(n)extra space**

**void rotate(int[] nums, int k)**

1 k=k%nums.length

2 let copy[0...nums.length-1] be a copy of array nums

3 **for** i=0 **to** nums.length-1

4 int j=(i+k)%nums.length

5 nums[j]=copy[i]

上述索引从0开始计算，为了使得与Line4的模值对应，免得要添加偏移

**O(1) extra space**

**void rotate(int[] nums, int k)**

1 k=k%nums.length

2 Reverse(nums,1,nums.length-k)

3 Reverse(nums,nums.length-k+1,nums.length)

4 Reverse(nums,1,nums.length)

**Reverse(int[] nums,int begin,int end)**

1 for i=begin to ⌊(begin+end)/2⌋

2 int tem=nums[i]

3 nums[i]=nums[end-(i-begin)]

4 nums[end-(i-begin)]=tem

# 190、反转整数的32bit位bit序列

**int reverseBits(int n)**

1 let Bits[1...32] be a new Array

2 **for** i=1 **to** 32

3 Bits[i]=n>>(i-1)&1

4 **for** i=1 **to** 16

5 int tem=Bits[i]

6 Bits[i]=Bits[32-i+1]

7 Bit[32-i+1]=tem

8 res=0

9 **for** i=32 **downto** 1

10 res=(res<<1)+Bit[i]

11 **return** res

# 191、整数的汉明重量（统计比特为1的个数）

**int hammingWeight(int n)**

1 res=0

2 **for** i=1 **to** 32

3 res=res+(n>>(i-1))&1

4 **return** res

# 198、小偷行窃（不能偷窃相邻的两个房子）

**DP1: M[i]代表前i个房子，并且对第i个房子行窃的最大收益**

**int rob(int[] nums)**

1 **if** nums==null **or** nums.length==0 **return** 0

2 **if** nums.length==1 **return** nums[1]

3 **if** nums.length==2 **return** max(nums[1],nums[2])

4 **if** nums.length==3 **return** max(nums[2],nums[1]+nums[3])

5 let M[1...nums.length] be a new Array

6 M[1]=nums[1]

7 M[2]=nums[2]

8 M[3]=nums[1]+nums[3]

9 maximum=max(M[3],M[2])

10 **for** i=4 **to** nums.length

11 M[i]=max(nums[i]+M[i-2],nums[i]+M[i-3])

12 maximum=max(maximum,M[i])

13 **return** M[i]

**DP2:M[i]代表前i个房子中，行窃所得最大收益**

**int rob(int[] nums)**

1 **if** nums==null **or** nums.length==0 **return** 0

2 **if** nums.length==1 **return** nums[1]

3 let M[1...nums.length] be a new array

4 M[1]=nums[1]

5 M[2]=max(nums[1],nums[2])

6 **for** i=3 **to** nums.length

7 M[i]=max(nums[i]+M[i-2],M[i-1])

8 **return** M[nums.length]

# 199、从右边看一个搜索二叉树所能看到的所有节点（每一层最右边的节点）

**List<Integer> rightSideView(TreeNode root)**

1 let Res be a new List<Integer>

2 Aux(root,1,Res)

3 **return** Res

**Aux(TreeNode cur,int height,List<Integer> Res)**

1 **if** cur==null **return**

2 **if** Res.size()<height Res.add(cur.val)

3 Aux(cur.right,height+1,Res)//right first

4 Aux(cur.left,height+1,Res)

**200、求岛屿的个数，与题130不同，本题中边界的'1'被认为是被'0'包围的（不用单独讨论边界），但是130题中，边界的'O'不被认为是被'X'包围的（需要先处理边界）**

**private int row,col**

**int numIslands(char[][] grid)**

1 res=0

2 **if** grid==null **or** grid.length==0 **or** grid[1].length==0 **return** 0

3 row=grid.length,col=grid[1].length

4 **for** i=1 **to** row

5 **for** j=1 **to** col

6 **if** grid[i][j]=='1'

7 res++;

8 search(grid,i,j)

9 **return** res

**void search(char[][] grid,int x,int y)**

1 grid[x][y]='2'

2 **if** x+1≤row **and** grid[x+1][y]=='1' search(grid,x+1,y)

3 **if** x-1≥1 **and** grid[x-1][y]=='1' search(grid,x-1,y)

4 **if** y+1≤col **and** grid[x][y+1]=='1' search(grid,x,y+1)

5 **if** y-1≥1 **and** grid[x][y-1]=='1' search(grid,x,y-1)