# 201、求m到n的位与

**等价于 求 m 与 n 二进制编码中 同为1的前缀.**

**int rangeBitwiseAnd(int m, int n)**

1 **if** m==0 **return** 0

2 moveFactor=0

3 **while** m≠n

4 m=m>>1

5 n=n>>1

6 moveFactor++

7 **return** m<<moveFactor//此时m为前缀部分，将其还原到其原有的位置上

**int rangeBitwiseAnd(int m, int n)**

1 r=Integer.MAX\_VALUE(111111...111)//32个1

2 **while** m&r≠n&r

3 r=r<<1

4 **return** n&r

# 202、Happy Number

**boolean isHappy(int n)**

1 let set be a new Set<Integer>

2 **while** set.add(n)

3 **if** n==1 **return** true

4 n=Aux(n)

5 **return** false

**int Aux(int n)**

1 res=0

2 **while** n≠0

3 remain=n%10

4 res=res+remain\*remain

5 n=n/10

6 **return** res

# 203、删除链表中指定值的元素

**ListNode removeElements(ListNode head, int val)**

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

2 let pseudohead be a new List with any value

3 pseudohead.next=head

4 pre=pseudohead,cur=head

5 **while** cur≠null

6 **if** cur.val==val pre.next=cur.next

7 **else** pre=cur

8 cur=cur.next

9 **return** pseudohead.next

# 204、求所有小于n的素数

**利用O(n)的额外空间来提升运算效率，从O(n2)->O(n)**

**int countPrimes(int n)**

1 let notPrime[1...n] be a new Array stored boolean initialized to false

2 count=0

3 **for** i=2 **to** n-1

4 **if** notPrime[i]==false

5 count++

6 **for** j=2 **to** ⌊ n/i ⌋//i\*j<n

7 notPrime[i\*j]=true

8 **return** count

# 205、判断两个字符串是否同构

"add" 与 "egg"同构

**boolean isIsomorphic(String s, String t)**

1 **if** s==null **or** t==null **throw** Exception

2 **if** s.length≠t.length **return** false

3 let m1[1...128] m2[1...128] be two new Array//128代表128个ASCII

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

5 **if** m1[s[i]] ≠m2[t[i]] **return** false

6 m1[s[i]]=i//代码中要写成i+1，以区分数组初始化原始的0

7 m2[t[i]]=i

8 **return** true

**注意点，m1[i]表示字符i出现的最后一次的位置**

# 206、反转链表

**public ListNode reverseList(ListNode head)**

1 let pseudohead be a new ListNode with arbitrary value

2 cur=head

3 **while** cur≠null

4 tem=cur.next

5 cur.next=pseudohead.next

6 pseudohead.next=cur

7 cur=tem

8 **return** pseudohead.next

# 207、能否完成课程（即判断有向图是否存在环结构）

**DFS法：**

**boolean canFinish(int numCourses, int[][] prerequisites)**

1 let Graph[1...numCourses] be a new Array stored Adjacent List initialized to empty List

2 **for** **each** edge of prerequisites

3 Graph[edge[2]].add(edge[1])//from course edge[2] point to course edge[1]

4 let visiting[1...numCourses] visited[1...numCourses] be Arrays stored boolean initialized to false

5 **for** i=1 **to** numCourses

6 **if** **not** visited[i]

7 **if** **not** DFS(Graph,visiting,visited,i) **return** false

8 **return** true

**boolean DFS(ArrayList<Integer>[] Graph,boolean[] visiting,boolean[] visited,int dex)**

1 **if** visiting[dex] **return** false

2 curNodeAdj=Graph[dex]

3 **visiting[dex]=true**

4 **for** **each** i of curNodeAdj

5if not visited[i]

6 **if** **not** DFS(Graph,visiting,visited,i) **return** false

7 **visiting[dex]=false**

8 **visited[dex]=true//必须放在最后，否则Line5可能有问题**

9 **return** true

**visited代表DFS中代表颜色的参数，true代表已经搜索过**

**visiting代表当前搜索中已经经过的节点，需要在DFS遍历Adj后重置为false**

**BSF法：**

**boolean canFinish(int numCourses, int[][] prerequisites)**

1 let Graph[1...numCourses] be a new Array stored Adjacent List initialized to empty List

2 let Degree[1...numCourses] be a new Array stored int initialized to zero

//Build Graph G(V,E)

3 **for** **each** edge of prerequisites

4 Degree[edge[1]]++

5 Graph[edge[2]].add(edge[1])

//Begin BFS from those nodes with zero Degree

6 let queue be a new Queue

7 visitcnt=1

8 **for** i=1 **to** numCourses

9 **if** Degree[i]==0

10 queue.add(i)

11 visitcnt ++

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

13 curnode=queue.poll()

14 curAdj=Graph[curnode]

15 **for** each edgeEnd:Adj

16 Degree[edgeEnd]--

17 **if** Degree[edgeEnd]==0

18 queue.offer(edgeEnd)

19 visitcnt++

20 **return** visitcnt==numCouses+1?true:false

**Degree[i]存储的是第i个课程的先修课程的个数，即以该节点为**

# 208、单词查找树

**class TrieNode**

public TrieNode[] Children

boolean IsWord=false

char val

**public TrieNode()**

1 this.val='\0'

2 let Children[1...26] be a new Array stored TrieNode

**TrieNode(char val)**

1 this.val=val

2 let Children[1...26] be a new Array stored TrieNode

**public class Trie**

private TrieNode root

**public Trie()**

1 let root be a new TrieNode()

**public void insert(String word)**

1 curNode=root

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

3 curChar=word[i]

4 **if** curNode.Children[curChar-'a'+1]==null

5 let curNode.Children[curChar-'a'+1] be a new TrieNode(curChar)

6 curNode=curNode.Children[curChar-'a'+1]

7 curNode.IsWord=true

**public boolean search(String word)**

1 curNode=root

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

3 curChar=word[i]

4 **if** curNode.Children[curChar-'a'+1]==null **return** false

5 curNode=curNode.Children[curChar-'a'+1]

6 **return** curNode.IsWord

**public boolean startsWith(String prefix)**

1 curNode=root

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

3 curChar=word[i]

4 **if** curNode.Children[curChar-'a']==null **return** false

5 curNode=curNode.Children[curChar-'a']

6 **return** AnyWord(curNode)

**private boolean AnyWord(TrieNode cur)**

1 **if** cur.IsWord **return** true

2 **for** **each** trienode:cur.Children)

3 **if** trienode ==null **continue**

4 **if** AnyWord(trienode) **return** true

5 **return** false

# 209、求长度最小的子数组，满足子数组的和不小于指定值

**public int minSubArrayLen(int s, int[] nums)**

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

2 begin=1,end=1,sum=0 minimum=+∞

3 **while** end≤nums.length

4 **while** end≤nums.length **and** sum<s

5 sum=sum+nums[end++]

6 **while** begin<end **and** sum≥s

7 minimum=min(minimum,(end-1)-begin+1)

8 if minimum==1 return 1

9 sum=sum-nums[begin++]

10 **return** minimum==+∞?0:minimum

**任意时刻,sum为子数组nums[begin...end-1]的和**

# 210、有向无环图的Toplogical sort

**BFS算法：**

**public int[] findOrder(int numCourses, int[][] prerequisites)**

1 let Graph[1...numCourses] be a new Array stored Adjacent List initialized to empty List

2 let Degree[1...numCourses] be a new Array stored int initialized to zero

//Build Graph G(V,E)

3 **for** **each** edge of prerequisites

4 Degree[edge[1]]++

5 Graph[edge[2]].add(edge[1])

//Begin BFS from those nodes with zero Degree

6 let Res[1...numCourses] be a new Array

7 let queue be a new Queue

8 visitcnt=1

9 **for** i=1 **to** numCourses

10 **if** Degree[i]==0

11 queue.offer(i)

12 Res[visitcnt++]=i

13 **while** **not** queue.isEmtpy()

14 curnode=queue.poll()

15 curAdj=Graph[curnode]

16 **for** each edgeEnd:Adj

17 Degree[edgeEnd]--

18 **if** Degree[edgeEnd]==0

19 queue.offer(edgeEnd)

20 Res[visitcnt++]=edgeEnd

21 **return** visitcnt==numCouses+1?Res:null

**DFS1:**

**class Vertex{**

**static** int time=0

int end=0

int dex

boolean visited=false

boolean visiting=false

**public Vertex(int dex){**this.dex=dex**}**

**}**

**int[] findOrder(int numCourses, int[][] prerequisites)**

1 let Vertexs[1...numCourses] be a new Array stored Vertex

2 **for** i=1 **to** numCourses

3 Vertexs[i]=new Vertex(i)

4 let Graph[1...numCourses] be a new Array stored Ajacent List initialized to empty List

5 **for** **each** edge:prerequisites

6 Graph[edge[2]].add(Vertexs[edge[1])

7 **for** i=1 **to** numCourses

8 **if** **not** Vertexs[i].visited

9 **if** **not** DFS(Graph,Vertexs,i) **return** new int[0]

10 Descend Sort Vertexs by end

12 let Res[1...numCourses] be a new Array

13 cnt=0

14 **for** **each** v:Vertexs

15 Res[cnt++]=v.dex

16 **return** Res

**boolean DFS(ArrayList<Vertex>[] Graph,Vertex[] Courses,int dex)**

1 **if** Vertexs[dex].visiting **return** false

2 **Vertexs[dex].visiting=true**

3 curAdj=Graph[dex]

4 **for** Vertex v:curAdj

5 **if** **not** v.visited

6 **if** **not** DFS(Graph,Vertexs,v.dex)

7 **return** false

8 **Vertexs[dex].visiting=false**

9 **Vertexs[dex].visited=true//必须放在最后，否则Line可能有问题**

10 Vertex.time++

11 Vertexs[dex].end=Vertex.time

12 **return** true

# 211、单词查找树2

**public class WordDictionary {**

**private class TrieNode{**

TrieNode[] children;

boolean isWord;

**TrieNode()**

1 children=new TrieNode[26];

2 isWord=false;

}

**private TrieNode root;**

WordDictionary()

1 root=new TrieNode();

// Adds a word into the data structure.

**public void addWord(String word)**

1TrieNode iter=root;

2 **for** **each** c **of** word

3 index=c-'a';

4 **if** iter.children[index]==null

5 iter.children[index]=new TrieNode();

6 iter=iter.children[index];

7 iter.isWord=true;

// Returns if the word is in the data structure. A word could

// contain the dot character '.' to represent any one letter.

**public boolean search(String word)**

1 **return** searchHelper(word,root);

**private boolean searchHelper(String subWord,TrieNode root)**

1 **if** subWord.equals("") **return** root.isWord;

2 first=subWord[1]

3 **if** first=='.'

4 **for** int i=1 **to** 26

5 **if** root.children[i]≠null

6 **if** searchHelper(substring[2...end],root.children[i]) **return** true;

7 **return** false;

8 **else**

9 index=first-'a';

10 **if** root.children[index]≠null

11 **return** searchHelper(substring[2...end],root.children[index]);

12 **else**

13 **return** false;

# 212、

# 213、小偷偷房子（房子首尾相接）

**拆分成两个部分[1...n-1] 或 [2...n]**

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

1 **if** nums.length==1 **return** nums[0]

2 **return** max(Aux(nums,1,nums.length-1),Aux(nums,2,nums.length))

**private int Aux(int[] num, int begin, int end)**

1 preInclude=0,preExclude=0

2 **for** i=begin **to** end

3 tem1=preInclude,tem2=preExclude

4 preInclude=tem2+num[i]

5 preExclude=max(tem1,tem2)

6 **return** max(preInclude,preExclude)

**DP算法见198**

# 214、补充一个字符串，使其为回文序列，且长度最短

**KMP算法：十分巧妙**

sNew:aabbcde#edcbbaa

π :010000000000012

**KMP算法可以求出一个字符串的从字符串起始位置开始的最长回文序列**

**String shortestPalindrome(String s)**

1 sNew=s+'#'+s.reverse()

2 let π[1...sNew.length] be a new Array

3 π[1]=0//首字符定义为0（longgest pre-postfix 不包括自身）

4 k= π[1]

5 **for** q=2 **to** sNew.length

6 **while** k>0 **and** sNew[q]≠sNew[k+1]

7 k= π[k]

8 **if** sNew[q]==sNew[k+1]

9 k=k+1

10 π[q]=k

11 **return** s[π[sNew.length]+1...s.length].reverse+s

**Recursive Solution:**

# 215、线性时间选择算法

**int findKthLargest(int[] nums, int k)**

1 **return** Select(nums,1,nums.length,nums.length-k+1)

**int Select(int[] nums,int p,int r,int k)**

1 **if** p==r **return** nums[p]

2 q=Partition(nums,p,r)

3 n=q-p+1

4 **if** n==k **return** nums[q]

5 **if** k<n

6 **return** Select(nums,p,q-1,k)

7 **else**

8 **return** Select(nums,q+1,r,k-n)

# 216、指定长度子集和问题

**public List<List<Integer>> combinationSum3(int k, int target)**

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

2 let Pre be a new List<Integer>

3 nums={1,2,3,4,5,6,7,8,9}

4 Aux(nums,k,1,target,Res,Pre)

5 **return** Res

**void Aux(int[] nums,int k,int dex,int target,List<List<Integer>> Res,List<Integer> Pre)**

1 **if** Pre.size()==k **and** target==0

2 Res.add(Copy(Pre))

3 **return**

4 **for** start=dex **to** nums.length

5 Pre.add(nums[start])

6 Aux(nums,k,start+1,target-nums[start],Res,Pre)

7 Pre.remove(Pre.size())

# 217、判断是否含有相同的元素(元素大小与数组长度无关）

**public boolean containsDuplicate(int[] nums)**

1 let set be a new Set<Integer>

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

3 **if** not set.add(nums[i]) **return** true

4 **return** false

# 218、城市天际线问题

**Point的第一种存储方式**

**public List<int[]> getSkyline(int[][] buildings)**

1 let skyLine be a new ArrayList<int[]>

2 let points be a new ArrayList<int[]>

//一个建筑出现时为正，消失时为负

3 **for** **each** building:buildings

4 points.add({building[1],building[3]})

5 points add({building[2],-building[3]})

6 sort(points,new Comparator<int[]>(){

public int compare(int[] obj1,int[] obj2){

**if** obj1[1] ≠obj2[1] **return** obj1[1]-obj2[1]

**else** **return** obj2[2]-obj1[2]

}

7 let maxHeap be a MAXHEAP

8 curHeight=0,preHeight=0

9 **for** **each** point:points

10 **if** point[2]>0//当出现正值时，说明有一个建筑需要进入队列

11 maxHeap.offer(point[2])

12 **else** maxHeap.remove(-point[2])//出现负值说明该建筑已经到右边缘，将其退出队列

13 curHeight=maxHeap.peek()==null?0:maxHeap.peek()

14 **if** curHeight≠preHeight

15 skyLine.add({point[1],curHeight})

16 preHeight=curHeight

17 **return** skyLine

Line6:为什么当obj[1]相同时，要把高度**大**的放在前面？

情况1：若都为**正数**，即都是大楼刚出现的时刻，孰先孰后无关紧要

**情况2：若一正一负，若负在先，先将楼弹出最大堆，会导致以下情况**

**输入(0,2,3) 和 (2,5,3) 输出为(0,3) (2,0) (2,3) (5,0)**

**因此要求若在某一处即是某栋楼的开始，又是某栋楼的结束，先压入新楼，再弹出旧楼**

**Point的另一种存储方式**

**public List<int[]> getSkyline(int[][] buildings)**

1 let skyLine be a new ArrayList<int[]>

2 let points be a new ArrayList<int[]>

//一个建筑出现时为负，消失时为正

3 **for** **each** building:buildings

4 points.add({building[1],-building[3]})

5 points add({building[2],building[3]})

6 sort(points,new Comparator<int[]>(){

public int compare(int[] obj1,int[] obj2){

**if** obj1[1] ≠obj2[1] **return** obj1[1]-obj2[1]

**else** **return** obj1[2]-obj2[2]

}

7 let maxHeap be a MAXHEAP

8 curHeight=0,preHeight=0

9 **for** **each** point:points

10 **if** point[2]<0//当出现负值时，说明有一个建筑需要进入队列

11 maxHeap.offer(-point[2])

12 **else** maxHeap.remove(point[2]) //出现正值说明该建筑已经到右边缘，将其退出队列

13 curHeight=maxHeap.peek()==null?0:maxHeap.peek()

14 **if** curHeight≠preHeight

15 skyLine.add({point[1],curHeight})

16 preHeight=curHeight

17 **return** skyLine

Line6:为什么当obj[1]相同时，要把高度**小**的放在前面？

情况1：若都为**负数**，即都是大楼刚出现的时刻，孰先孰后无关紧要

**情况2：若一正一负，若正在先，先将楼弹出最大堆，会导致以下情况**

**输入(0,2,3) 和 (2,5,3) 输出为(0,3) (2,0) (2,3) (5,0)**

**因此要求若在某一处即是某栋楼的开始，又是某栋楼的结束，先压入新楼，再弹出旧楼**

**219、判断是否含有距离不超过k的相同元素**

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

1 let set be a new Set<Integer>

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

3 **if** i>k set.remove(nums[i-k-1])

4 **if** **not** set.add(nums[i]) **return** true

5 **return** false

# 220、判断是否含有距离不超过k的且差值不超过t的一对元素(share solution)

**boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t)**

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

2 Long maximum=Integer.MIN\_VALUE,minimum=Integer.MAX.VALUE

3 **for** **each** i:nums

4 maximum=max(maximum,i)

5 minimum=min(minimum,i)

6 Long numBucket=(maximum-minimum)/(t==0?1:t)**+1**

7 let buckets be a new Map<Long,LinkedList<Integer>> buckets

8 let queueNumBucket be a new Queue stroed Long

9 **for** i=1 **to** numBucket

10 buckets.put(i,a new LinkedList)

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

12 Long dexBucket=(nums[i]-minimum)/(t==0?1:t)

13 queueNumBucket.offer(dexBucket)

14 **if** queueNumBucket.size()>k+1//队列中有k+2个元素时，弹出队头元素

15 buckets[queueNumBucket.poll()].clear()

16 curbucket=buckets[dexBucket]

17 **if** curbucket.size()≠0 **return** true

18 curbucket.add(nums[i])

19 **if** dexBucket>1 **and** buckets[dexBucket-1].size()≠0

**and** |buckets[dexBucket-1].getFirst()-long(nums[i])|≤t

20 **return** true

21 **if** dexBucket<numBucket **and** buckets[dexBucket+1].size()≠0

**and** |buckets[dexBucket+1].getFirst()-long(nums[i])|≤t

22 **return** true

23 **return** false

类似利用桶的还有Code164

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

nums[1...n]

A、**规定桶的个数为n个**：那么间隔为⌈(max-min)/(num.length-1)⌉

**因为只有最大值会放到第n个桶中（特殊情况）**

B、**规定桶的长度为len**：那么桶的个数为⌊(max-min)/len⌋**+1**

**其中+1是为了补偿max-min恰能被len整除的时候，由于桶的范围是左闭右开的，需要将最大值放入额外的一个桶**

# 221、最大的正方形区域（标记为'1'的区域）

**public int maximalSquare(char[][] matrix)**

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

2 maximum=0

3 **for** row=1 **to** matrix.length

4 **for** col=1 **to** matrix[1].length

5 **if** matrix[row][col]=='0' **continue**

6 maximum=max(maximum,Aux(maximum,row,col)

**private int Aux(char[][] matrix,int row,int col)**

1 i=row+1,j=col+1

2 **while** i<matrix.length **and** j<matrix[1].length

3 **for** k=col **to** j

4 **if** matrix[i][k]=='0' **return** ((i-1)-row+1)\* ((i-1)-row+1)

5 **for** k=row **to** i-1

6 **if** matrix[k][j]=='0' **return**((i-1)-row+1)\* ((i-1)-row+1)

7 i++ j++

8 **return** ((i-1)-row+1)\* ((i-1)-row+1)

**动态规划：**

**dp[row][col]=min(dp[row-1][col-1],dp[row][col-1],dp[row-1][col])+1**

dp[row][col]存储的是，以(row,col)为右下端点的正方形的边长

**public int maximalSquare(char[][] matrix)**

1 rows=matrix.length

2 cols=rows>0?matrix[1].length:0

3 let dp[0...rows][0...cols] be a new array

4 res=0

5 **for** row=1 **to** rows

6 **for** col=1 **to** cols

7 **if** matrix[row][col]=='1'

8 dp[row][col]=min(dp[row-1][col-1],dp[row-1][col],dp[row][col-1])+1

9 res=max(res,dp[row][col]\*dp[row][col])

10 **return** res

**public int maximalSquare(char[][] matrix)**

1 rows=matrix.length

2 cols=rows>0?matrix[1].length:0

3 let dp[0...cols] be a new array

4 res=0,pre=0

5 **for** row=1 **to** rows

6 **for** col=1 **to** cols

7 **tmp=dp[col]**

8 **if** matrix[row][col]=='1'

9 dp[col]=min(pre,dp[col-1],dp[col])

10 **else** **dp[col]=0**

11 res=Math.max(res,dp[col]\*dp[col])

12 **pre=tmp**

13 **return** res



1、每次迭代会从左到右计算dp[col]，在更新dp[col]之前，dp[col]的值为**上一行**该列的值，即代表了红色的区域

2、而dp[col-1]已经是**本行**该列的新值，因此无法通过dp来表示pre部分，而pre却是dp[col-1]更新之前的值，因此用额外的一个量来存储即可

# 222、完全二叉树节点个数(share solution) 复杂度0(lg2(n))

**public int countNodes(TreeNode root)**

1 LeftMostHeight=0

2 iter=root

3 **while** iter≠null

4 LeftMostHeight++

5 iter=iter.left

6 RightMostHeight=0

7 iter=root

8 **while** iter≠null

9 RightMostHeight++

10 iter=iter.right

11 **if** LeftMostHeight==RightMostHeight return (1<<LeftMostHeight)-1

12 left=0,right=(1<<RightMostHeight)-1

13 leafNum=Aux(root,left,right,RightMostHeight)

14 notleafNum=(1<<RightMostHeight)-1

15 **return** leafNum+notleafNum

**private int Aux(TreeNode root,int left,int right,int len)**

1 **if** left==right **return** left+1

2 **if** left==right-1 **return** left+1

3 mid=⌊(left+right)/2⌋

4 midNode=root

5 **for** i=1 **to** len

6 midNode=((mid>>(len-i))&1)==0?midNode.left:midNode.right

7 **if** midNode==null **return** Aux(root,left,mid,len)

8 **else** **return** Aux(root,mid,right,len)

**public int countNodes(TreeNode root)**

1 LeftMostHeight=0

2 iter=root

3 **while** iter≠null

4 LeftMostHeight++

5 iter=iter.left

6 RightMostHeight=0

7 iter=root

8 **while** iter≠null

9 RightMostHeight++

10 iter=iter.right

11 **if** LeftMostHeight==RightMostHeight

12 return (1<<LeftMostHeight)-1

13 **else** **return** 1+countNode(root.left)+countNodes(root.right)

2的n次幂: 1<<n

# 223、两个矩形的面积

**public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H)**

1 left=max(A,E)

2 bottom=max(B,F)

3 right=min(C,G)

4 top=min(D,H)

5 **if** left≥right **or** bottom≥top overlap=0

6 **else** overlap=(right-left)\*(top-bottom)

7 areaA=(C-A)\*(D-B),areaB=(G-E)\*(H-F)

8 **return** areaA+areaB-overlap

# 224、含有加减法的运算表达式的实现(brillient!)

**public int calculate(String s)**

1 res=0

2 sign='+'

3 let stackVal and stackSign be two Stacks

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

5 **if** s[i]==' ' **continue**

6 **elseif** Character.isDigit(s[i])

7 curVal=0

8 **while** i≤s.length() **and** Character.isDigit(s[i])

9 curVal=curVal\*10+s[i++]-'0'

10 i--

11 **if** sign=='+' res=res+curVal

12 **elseif** sign=='-' res=res-curVal

13 **elseif** s[i]=='('

14 stackVal.push(res)

15 stackSign.push(sign)

16 res=0

17 sign='+'

18 **elseif** s[i]==')'

19 res=stackVal.poll()+(stackSign.poll()=='+'?1:-1)\*res

20 **else** sign=s[i]

21 **return** res

**优化方案：**

**public int calculate(String s)**

1 sign=1,res=0//sign最初赋值为1，即正号

2 let stack be a new Stack

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

4 **if** Character.isDigit(s[i])

5 curVal=0

6 **while** i≤s.length() **and** Character.isDigit(s[i])

7 curVal=curVal\*10+(s[i++]-'0')

8 i--

9 res=res+curVal\*sign

10 **elseif** s[i]=='+' sign=1

11 **elseif** s[i]=='-' sign=-1

12 **elseif** s[i]=='('

13 stack.push(res)

14 stack.push(sign)

15 res=0

16 sign=1

17 **elseif** s[i]==')'

18 res=res\*stack.pop()+stack.pop()

//空格部分什么也不做，直接跳过

19 **return** res

# 225、用队列实现栈

**思路：在压入时，使得满足栈的性质**

**不变式：压入开始前，队列满足栈的性质，即队列头尾栈顶元素，队列尾为栈底元素**

**压入结束后，队列头元素为刚压入的元素，也满足栈的性质**

**class MyStack {**

//one Queue solution

**private Queue<Integer> queue = new LinkedList<Integer>()**

// Push element x onto stack.

**public void push(int x)**

1 queue.add(x)

2 **for** i=2 **to** queue.size()

3 queue.add(queue.poll())

// Removes the element on top of the stack.

**public void pop()**

1 queue.poll()

// Get the top element.

**public int top()**

1 **return** queue.peek()

// Return whether the stack is empty.

**public boolean empty()**

1 **return** queue.isEmpty()

**}**

# 226、反转一颗BST（二叉搜索树）

**public TreeNode invertTree(TreeNode root)**

1 helper(root)

2 **return** root

**private void helper(TreeNode cur)**

1 **if** cur≠null

2 tem=cur.left

3 cur.left=cur.right

4 cur.right=tem

5 helper(cur.left)

6 helper(cur.right)

# 227、加减乘除运算表达式的计算（不含括号）类似的有224

**Solution1: Using Stack**

**public int calculate(String s)**

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

2 let stack be a new Stack

3 sign='+'

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

5 **if** s[i]==' ' **continue**

6 **elseif** Character.isDigit(s[i])

7 curVal=0

8 **while** i≤s.length() **and** Character.isDigit(s[i])

9 curVal=curVal\*10+s[i++]-'0'

10 i--

11 **if** sign=='+'

12 stack.push(curVal)

13 **elseif** sign=='-'

14 stack.push(-curVal)

15 **elseif** sign=='\*'

16 stack.push(stack.pop()\*curVal)

17 **elseif** sign=='/'

18 stack.push(stack.pop()/curVal)

19 **else** sign=s[i]

20 res=0

21 **while** **not** stack.isEmtpy()

22 res=res+stack.pop()

23 **return** res

**给第一个数附上初始的符号'+'，将 符号-数字 视为一对**

**当数字出现后，根据与该数字配对的符号进行计算，将当前计算结果压进栈**

**另外当为乘法除法时，当前结果与上一次的结果有关，一次与栈顶元素计算后再压入栈**

**Solution 2:Not Using Stack**

**public int calculate(String s)**

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

2 res=0

3 preVal=0,curVal=0

4 sign='+'

5 **for** i=1 **to** s.length()

6 **if** s[i]==' ' **continue**

7 **elseif** Character.isDigit(s[i])

8 curVal=0

9 **while** i≤s.length() **and** Character.isDigit(s[i])

10 curVal=curVal\*10+s[i++]-'0'

11 i--

12 **if** sign=='+'//当前数值与前一个数值无关，将前一个数值更新到总和中

13 res=res+preVal

14 preVal=curVal

15 **elseif** sign=='-'//当前数值与前一个数值无关，将前一个数值更新到总和中

16 res=res+preVal

17 preVal=-curVal

18 **elseif** sign=='\*'//当前数值与前一个数值有关，不更新总和

19 preVal=preVal\*curVal

20 **elseif** sign=='/'//当前数值与前一个数值有关，不更新总和

21 preVal==preVal/curVal

22 **else** sign=s[i]

23 res=res+preVal//需要加上最后一个结果

24 **return** res

**带有括号的加减乘除表达式的运算**

**public int calculate(String s)**

1 let stkVal,stkRes be two new Stack stored int

2 let stkSign be new Stack stored sign

3 preVal=0

4 res=0

5 sign='+'

6 **for** i=1 **to** s.length()

7 **if** s[i]==' ' **continue**

8 **elseif** Character.isDigit(s[i])

9 curVal=0

10 **while** i≤s.length() Character.isDigit(s[i])

11 curVal=curVal\*10+s[i++]-'0'

12 i--

13 **if** sign=='+'//当前数值与前一个数值无关，将前一个数值更新到总和中

14 res=res+preVal,preVal=curVal

15 **elseif** sign=='-' //当前数值与前一个数值无关，将前一个数值更新到总和中

16 res=res+preVal,preVal=-curVal

17 **elseif** sign=='\*' preVal=preVal\*curVal//当前数值与前一个数值有关，不更新总和

18 **elseif** sign=='/' preVal/curVal//当前数值与前一个数值有关，不更新总和

19 **elseif** s[i]=='('

20 stkRes.push(res)

21 stkVal.push(preVal)

22 stkSign.push(sign)

23 preVal=0,sign='+',res=0

24 **elseif** s[i]==')'

25 res=res+preVal//括号内是一个完整表达式，需要加上括号内最后一个数值

26 **if** stkSign.peek()=='+'//当前数值与前一个数值无关，将前一个数值更新到总和中

27 preVal=**res**,res=stkRes.poll()+stkVal.poll()

28 **elseif** stkSign.peek()=='-'//当前数值与前一个数值无关，将前一个数值更新到总和

29 preVal=-**res**,res=stkRes.poll()+stkVal.poll()

30 **elseif** stkSign.peek()=='\*'//当前数值与前一个数值有关，不更新总和

31 preVal=stkVal.poll()\***res**,res=stkRes.poll()

32 **elseif** stkSign.peek()=='/'//当前数值与前一个数值有关，不更新总和

33 preVal=stkVal.poll()/**res**,res=stkRes.poll()

34 stkSign.pop()

35 **else** sign=s[i]

36 **return** res+preVal

**Line 27-33 黄色的res的值是括号内表达式的值**



# 228、归纳数字的范围

**[0,1,2,3,5,6,7,9]-> ["0->3","5->7","9"]**

**public List<String> summaryRanges(int[] nums)**

1 let res be a new LinkedList<String>

2 **if** nums==null **or** nums.length==0 return res

3 begin=1,end=2

4 **while** end≤nums.length

5 **if** nums[end] ≠nums[end-1]+1

6 **if** end-1-begin+1==1

7 res.add(Integer.toString(nums[begin]))

8 **else**

9 res.add(Integer.toString(nums[begin])+"->"+ Integer.toString(nums[end-1])

10 end++

11 **if** end-1-begin+1==1

12 res.add(Integer.toString(nums[begin]))

13 **else**

14 res.add(Integer.toString(nums[begin])+"->"+ Integer.toString(nums[end-1])

15 **return** res

# 229、找出主元（出现次数多于⌊n/3⌋)

**思路：主元最多只有两个**

**public List<Integer> majorityElement(int[] nums)**

1 element1=∞,element2=∞,cnt1=0,cnt2=0

2 **for** **each** i:nums

3 **if** i==element1 cnt1++

4 **elseif** i==element2 cnt2++

5 **elseif** cnt1==0 element1=i,cnt1=1

6 **elseif** cnt2==0 element2=i,cnt2=1

7 **else** cnt1--,cnt2--

8 cnt1=cnt2=0

9 **for** **each** i:nums

10 **if** i==element1 cnt1++

11 **elseif** i==element2 cnt2++

12 let res be a new List

13 **if** cnt1>nums.length/3 res.add(element1)

14 **if** cnt2>nums.length/3 res.add(element2)

15 **return** res

**Line 5、6行保证每次只更替一个**

# 230、二叉搜索树的第k顺序数

**int cnt=0**

**int res=0**

**boolean founded=false**

**public int kthSmallest(TreeNode root, int k)**

1 helper(root,k)

2 **return** res

**private void helper(TreeNode cur,int k){**

1 **if** **not** founded **and** cur≠null

2 helper(cur.left,k)

3 cnt++

4 **if** cnt==k res=cur.val,founded=true,**return**

5 helper(cur.right,k)

# 231、判断是否是2的幂次

**public boolean isPowerOfTwo(int n)**

1 **if** n≤0 **return** false

2 cnt=0

3 **for** i=0 **to** 30

4 cnt=cnt+(n>>i&1)

5 **return** cnt==1? true:false

# 232、用堆栈实现队列

**思路：与利用队列实现栈一样，在压入时，使得其满足队列的性质**

**不变式：压入前，栈中元素满足队列的性质（栈顶为队列头，栈底为队列尾）**

**压入后，压入的元素位于栈底（也就是队列尾），满足队列的性质**

**class MyQueue {**

LinkedList<Integer> queue = new LinkedList<Integer>()

// Push element x to the back of queue.

**public void push(int x)**

1 LinkedList<Integer> temp = new LinkedList<Integer>()

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

3 temp.push(queue.pop())

4 queue.push(x)

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

6 queue.push(temp.pop())

// Removes the element from in front of queue.

**public void pop()**

1 queue.pop()

// Get the front element.

**public int peek()**

1 **return** queue.peek()

// Return whether the queue is empty.

**public boolean empty()**

1 **return** queue.isEmpty()

**}**

# 233、1的计数???

**public int countDigitOne(int n)**

1 onts=0

2 m=1

3 **while** m≤n

4 ones=ones+(n/m+8)/10\*m+(n/m%10==1?n%m+1:0)

5 m=m\*10

6 **return** ones

# 234、判断链表是否为Palindrome O(n)复杂度 O(1)空间

**public boolean isPalindrome(ListNode head)**

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

2 slow=head,fast=head

3 **while** fast.next≠null **and** fast.next.next≠null//寻找中点的判断条件

4 fast=fast.next.next

5 slow=slow.next

6 slow=reverse(slow.next) //无论长度为奇数还是偶数，slow.next是又半部分的开始

7 iter1=head,iter2=slow

8 **while** iter2≠null

9 **if** iter1.val≠iter2.val **return** false

10 iter1=iter1.next

11 iter2=iter2.next

12 **return** true

**ListNode reverse(ListNode head)**

1 headNew=null

2 iter=head

3 **while** iter≠null

4 tem=iter.next

5 iter.next=headNew

6 headNew=iter

7 iter=tem

8 **return** headNew

循环条件为fast.next≠null **and** fast.next.next≠null



循环条件为fast ≠null **and** fast.next ≠null



# 235、搜索二叉树两个节点的公共祖先

**public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)**

1 iter=root

2 parent=root

3 **while** iter≠null

4 **if** iter.val==p.val **or** iter.val=q.val **return** iter

5 parent=iter

6 **if** iter.val>p.val **and** iter.val>q.val

7 **elseif** iter.val<p.val and iter.val<q.val

8 **else** break

9 **return** parent

**236、一般二叉树两个节点的公共祖先**

**TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)**

1 let path1,path2 be two new empty List<TreeNode>

2 findPath(root,p,path1)

3 findPath(root,q,path2)

4 i=0

5 parent=root

6 **while** i≤path1.size() **and** i≤path2.size()

7 **if** path1[i].val≠path2[i].val break

8 parent=path1[i++]

9 **return** parent

**private boolean findPath(TreeNode cur,TreeNode p,List<TreeNode> path)**

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

2 path.add(cur)

3 **if** cur==p **return** true

4 **if** findPath(cur.left,p,path) **return** true

5 **if** findPath(cur.right,p,path) **return** true

6 path.remove(path.size())

7 **return** false

**public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)**

1 **if** root==null **or** root==p **or** root==q **return** root

2 left= lowestCommonAncestor(root.left, p, q)

3 right= lowestCommonAncestor(root.right, p, q)

4 **if** left≠ null **and** right≠null **return** root//左右都不为空说明，p和q分别位于root的左右子树中，因此他们的公共祖先LSA为root

5 **return** left≠null? left:right//左为空说明，q、p不存在与root的左子树中,右为空说明p、q不存在与root的右子树中

**//注意，left，right并不是root.left和root.right，而是left位于root.left，right位于root.right**

# 237、删除链表中指定的节点（只给了这个节点）

**由于无法获取当前节点的前一个节点，因此只能通过改变节点的值来达到删除的目的**

**public void deleteNode(ListNode node)**

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

2 iter=node,pre=null

3 **while** iter.next≠null

4 pre=iter

5 iter.val=iter.next.val

6 iter=iter.next

7 pre.next=null

# 238、计算数组中每一项的积（除了该项之外的积）

例如：{1，2，3，4}-->{2\*3\*4，1\*3\*4，1\*2\*4，1\*2\*3}

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

1 let res[1...nums.length] be a new Array

2 tolProduct=help(nums,1)

3 res[1]=tolProduct

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

5 **if** nums[i]==0 //recalculate tolProduct

6 tolProduct=helper(nums,i,)

7 **else**

8 tolProduct=tolProduct\*nums[i-1]/nums[i]

9 res[i]=tolProduct

10 **return** res

**private int helper(int[] nums,int dex)**

1 tolProduct=1

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

3 **if** i==dex **continue**

4 **if** nums[i]==0

5 tolProduct=0,**break**

6 **else** tolProduct=tolProduct\*nums[i]

7 **return** tolProduct

**Brilliant Solution:**

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

1 res[0]=1

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

3 res[i]=res[i-1]\*nums[i-1]

4 right=1

5 **for** i=nums.length **downto** 1

6 res[i]=res[i]\*right

7 right=right\*nums[i]

8 **return** res

# 239、移动窗口的最大值数组

**例如数组{1,2,3,4,5} 窗口长度为3**

**窗口第一次在{1,2,3}，最大值3，第二次在{2,3,4}，最大值4，第三次在{3,4,5}，最大值5**

**输出为{3,4,5}，即每次窗口最大值所构成的数组**

**public int[] maxSlidingWindow(int[] nums, int k)**

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

2 let queue be a PriorityQueue<Integer>

3 res[1...nums.length-k+1] be a new Array

4 **for** i=1 **to** k

5 queue.offer(nums[i])

6 res[1]=queue.peek()

7 **for** i=k+1 **to** nums.length

8 queue.remove(nums[i-k])

9 queue.offer(nums[i])

10 res[i-k+1]=queue.peel()

11 **return** res

**Brilliant Solution:**

**public int[] maxSlidingWindow(int[] nums, int k)**

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

2 res[1...nums.length-k+1] be a new Array

3 dex=1

4 let queue be a new Queue<Integer>

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

// remove numbers out of range k

6 **while** **not** queue.isEmtpy() **and** queue.peek()<i-k+1

7 queue.poll()

// remove smaller numbers in k range as they are useless

//保证队列的尾部（当前值）是队列的最小值

8 **while** not queue.isEmtpy() **and** nums[queue.getLast()]<nums[i]

9 queue.removeLast()

10 queue.offer(i)

11 **if** i≥k

12 res[dex++]=nums[queue.peek()]

13 **return** res

**始终保持队头为当前窗口值最大的索引**

# 240、搜索有序矩阵中搜索给定值

**public boolean searchMatrix(int[][] matrix, int target)**

1 **if** matrix==null **or** matrix.length==0 **or** matrix[0].length==0 **return** false

2 col=matrix[1].length

3 row=1

4 **while** col≥1 **and** row ≤matrix.length

5 **if** target==matrix[row][col] **return** true

6 **elseif** target<matrix[row][col] col--

7 **elseif** target>matrix[row][col] row++

8 **return** false

# 241、求一个包含"+ - \*"的运算表达式添括号所得的所有结果（结果可能重复）

**问题转化为：两个子问题（两个子问题 由 选择先算哪个运算符号所得）**

**public List<Integer> diffWaysToCompute(String input)**

1 let res be a new empty List<Integer>

2 **if** input==null **or** input.length==0 **return** res

3 isSingleValue=true

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

5 **if** "+-\*".indexOf(input[i]) ≠-1

6 isSingleValue=false

7 String inputLeft=input[1...i-1]

8 String inputRight=input[i+1...end]

9 resLeft=diffWaysToCompute(inputLeft)

10 resRight=diffWaysToCompute(inputRight)

11 **for** **each** Integer:resLeft

12 **for** **each** Integer:resRight

13 **if** input[i]=='+'

14 res.add(left+right)

15 **elseif** input[i]=='-'

16 res.add(left-right)

17 **else**

18 res.add(left\*right)

19 **if** isSingleValue res.add(Integer.parseInt(input))

20 **return** res

# 242、判断两个字符串是否包含相同的元素

**public boolean isAnagram(String s, String t)**

1 **if** s==null **or** t==null **return** false

2 **if** s.length≠t.length **return** false

3 aryS=s.toCharArray()

4 aryT=t.toCharArray()

5 sort(aryS),sort(aryT)

6 **for** i=1 **to** aryS.length

7 **if** aryS[i] ≠aryT[i] **return** false

8 **return** true

# 257、二叉树的所有路径（从根到叶节点）

**public List<String> binaryTreePaths(TreeNode root)**

1 let res be a new ArrayList<String>

2 **if** root≠null helper(root,"",res)

3 **return** res

**void helper(TreeNode root, String path, List<String> res)**

1 **if** root.left==null **and** root.right==null res.add(path+root.val)

2 **if** root.left≠null helper(root.left,path+root.val+"->",res)

3 **if** root.right≠null helper(root.right,path+root.val+"->",res)

**关键：1. "->"放置位置**

由于第一个节点会直接填写，而其余节点均要用->链接

path+root.val作为通用表达式，既要适用于根节点，又要适用于其他节点，因此"->"应该放在path中

# 258、数字各个位相加，直至和为个位数

**public int addDigits(int num)**

1 **while** num≥10

2 num=helper(num)

3 **return** num

**private int helper(int num)**

1 res=0

2 **while** num≠0

3 res=res+num%10

4 num=num/10

5 **return** res

# 260、找到两个只出现过一次的数（在一个数组中，其余元素均出现两次）

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

1 xor=0

2 **for** **each** num of nums

3 xor=xor^num

4 offset=0

5 **while** (xor&(1<<offset))==0

6 offset++

\* 4 offset=31

\* 5 while (xor&(1<<offset))==0

\* 6 offset--

7 divide=1<<offset

8 rets={0,0}

9 **for** **each** num of nums

10 **if** num&divide==0

11 rets[0]=rets[0]^num

12 **else**

13 rets[1]=rets[1]^num

14 **return** rets

由于xor最后只是这两个只出现一次的元素的异或结果，由于这两个元素不同，因此异或结果xor必然包含1（bit）任选其中一个1bit作为两组元素的分隔

在两组中各自进行异或，每一组只含有一个出现一次的元素，因此异或结果就是该元素

# 263、判断是否是为ugly数字

ugly满足其因式分解只包含2、3、5，且定义1为ugly数字

**public boolean isUgly(int num)**

1 **if** num==1 **return** true

2 **if** num==0 **return** false

3 **while** num≠1

4 **if** num%2==0

5 num=num/2

6 **elseif** num%3==0

7 num=num/3

8 **elseif** num%5==0

9 num=num/5

10 **else** **return** false

11 **return** true

# 264、第n个ugly数字

1\*2 2\*2 3\*2 4\*2 5\*2...

1\*3 2\*3 3\*3 4\*3 5\*3...

1\*5 2\*5 3\*5 4\*5 5\*5...

**DP:**

**public int nthUglyNumber(int n)**

1 let ugly[1...n] be a new array

2 ugly[1]=1

3 index2=1,index3=1,index5=1

4 factor2=2,factor3=3,factor5=5

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

6 minimum=min(min(factor2,factor3),factor5)

7 ugly[i]=minimum

8 **if** factor2==minimum//并列的判断

9 factor2=2\*ugly[++index2]

10 **if** factor3==minimum//并列的判断

11 factor3=3\*ugly[++index3]

12 **if** factor5==minimum//并列的判断

13 factor5=5\*ugly[++index5]

14 **return** ugly[n]

往前推进的方式并不是(1 2 3 4 5 6 7...)\*2(or3or5): 2\*(++index2)

而是下一个ugly数字乘以2(or3or5): 2\*ugly[++index2]

# 268、丢失的数字

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

1 res=nums.length

2 i=0

3 for each num of nums

4 res=res^num

5 res=res^i++

6 return res

输入只能缺少一个数字，而不能为一串

若没有缺少数字，即第一个丢失的数字为下一个数字n

0 1 2 3...n-1

若丢失了数字k

那么数组为

0 1 2 ...k-1 k+1...n-1 n

**对比两个数组，除了数字k与数字n之外，剩余数字出现了2次，用异或便可滤去这些数字**

# 273、阿拉伯数字变英文表达

**public String numberToWords(int num)**

1 if num==0 return "Zero"

2 return helper(num)

**private String helper(int num)**

1 **if** num<1000 **return** lessThousandToWord(num)

2 **if** num<1000\*1000 **return** lessThousandToWord(⌊num/1000⌋)+

" Thousand"+(num%1000==0?"":" "+helper(num%1000))

3 **if** num<1000\*1000\*1000 **return** lessThousandToWord(⌊num/(1000\*1000) ⌋)+

" Million"+(num%(1000\*1000)==0?"":" "+helper(num%(1000\*1000)))

4 **return** lessThousandToWord(⌊num/(1000\*1000\*1000) ⌋)+

" Billion"+(num%(1000\*1000\*1000)==0?"":" "+helper(num%(1000\*1000\*1000)))

**private String lessThousandToWord(int n){**

1 **if** n<10 **return** digitToWord(n)

2 **if** n≥10&&n≤19 **return** tenToWord(n)

3 **if** n≥20&&n<100 **return** tensToWord(n)+(n%10==0?"":" "+digitToWord(n%10))

4 **return** digitToWord(⌊n/100⌋)+" Hundred"+(n%100==0?"":" "+lessThousandToWord(n%100))

**private String digitToWord(int n)**

1 **switch**(n)

2 **case** 1:**return** "One"

3 **case** 2:**return** "Two"

4 **case** 3:**return** "Three"

5 **case** 4:**return** "Four"

6 **case** 5:**return** "Five"

7 **case** 6:**return** "Six"

8 **case** 7:**return** "Seven"

9 **case** 8:**return** "Eight"

10 **case** 9:**return** "Nine"

11 **default**: **return** ""

**private String tensToWord(int n)**

1 **switch**(n/10)

2 **case** 2: **return** "Twenty"

3 **case** 3: **return** "Thirty"

4 **case** 4: **return** "Forty"

5 **case** 5: **return** "Fifty"

6 **case** 6: **return** "Sixty"

7 **case** 7: **return** "Seventy"

8 **case** 8: **return** "Eighty"

9 **case** 9: **return** "Ninety"

10 **default**: **return** ""

**private String tenToWord(int n)**

1 **switch**(n)

2 **case** 10: **return** "Ten"

3 **case** 11: **return** "Eleven"

4 **case** 12: **return** "Twelve"

5 **case** 13: **return** "Thirteen"

6 **case** 14: **return** "Fourteen"

7 **case** 15: **return** "Fifteen"

8 **case** 16: **return** "Sixteen"

9 **case** 17: **return** "Seventeen"

10 **case** 18: **return** "Eighteen"

11 **case** 19: **return** "Nineteen"

12 **default**: **return** ""

# 274、论文h索引

学者的文论索引h定义如下：

**N篇paper中，至少被引用h次的论文有h篇，剩余N-h篇论文被引用次数不多于h次**

**public int hIndex(int[] citations)**

1 len=citations.length

2 let count[0...len] be a new array//利用了线性时间排序算法的思路

3 for each c of citations

4 if c≥len count[len]++

5 else count[c]++

6 total=0

7 for i=len downto 0

8 total=total+count[i]

9 if total≥i

10 return i

11 return 0

**为什么第9行是大于等于而不是等于：**

**{1,1}当i为2是total为0，total为1时total为2**

**7-9行迭代过程中：若当前迭代total没有发生改变，而i相比于上次迭代减少了1，此时若条件成立则一定是取等号**

# 275、论文h索引（对于已排序的情况）（二分法）

**public int hIndex(int[] citations)**

1 left=0,len=citations.length,right=len-1

2 **while** left**<**right

3 mid=⌊(left+right)/2⌋

4 **if** citations[mid] <len-mid left=mid+1

5 **else** right=mid-1

6 **if** len==0 **return** 0

7 **return** citations[left] <len-left?len-left-1:len-left

**public int hIndex(int[] citations)**

1 left=0,len=citations.length,right=len-1

2 **while** left**≤**right

3 mid=⌊(left+right)/2⌋

4 **if** citations[mid] <len-mid left=mid+1

5 **else** right=mid-1

6 **return** len-left

# 278、找到第一个错误的版本（二分搜索）

**public int firstBadVersion(int n)**

1 left=1,right=n

2 **while** left**<**right

3 mid=⌊(left+right)/2⌋

4 **if** **not** isBadVersion(mid) left=mid+1

5 **else** right=mid-1

6 **return** isBadVersion(left)?left:left+1

**public int firstBadVersion(int n)**

1 left=1,right=n

2 **while** left**≤**right

3 mid=⌊(left+right)/2⌋

4 **if** **not** isBadVersion(mid) left=mid+1

5 **else** right=mid-1

6 **return** left

**细节：判断条件为left≤right时，其结果往往不需要讨论**

**判断条件为left<right时，其结果往往需要讨论**

**共性：进行二分时，要么取右边，要么取左边，都不包括mid，换言之，新的迭代区域的边界情况（若取右边，[mid+1,right]中的mid+1，若取左边[left,mid-1]中的mid-1）并不确定**

**原因：当left<right时，终止时left==right，但这点的情况并不明确，因此需要对该点进行判断后才能输出最后结果；当left≤right时，终止时left>right，并且left的情况是明确的，因为当left==right时的迭代会使得left处于正确的位置**

# 279、整数分解成平方数的最少个数

**DP: dp[i+j\*j]=min(dp[i+j\*j],dp[i]+1)**

**public int numSquares(int n)**

1 let dp[0...n] be a new array initialized to +∞

2 dp[0]=0**//使得平方数也是正确的结果：1**

3 **for** i=0 **to** n

4 **for** j=1 **to** i+j\*j≤n

5 dp[i+j\*j]=min(dp[i+j\*j],dp[i]+1)

6 **return** dp[n]

**很神奇的一点，在你求dp[i+j\*j]时，所访问的dp[i]一定是值为i时的最小值，也就是说更新一定会发生在被调用之前**

**被更新时为 i1+j1\*j1，被调用时为i2，满足i2= i1+j1\*j1**

**当被调用时,i==i2，因此i2+j\*j>i2，因此不可能再被更新了**

# 282、结果为设定值得运算表达式

**public List<String> addOperators(String num, int target)**

1 let res be a new List<String>

2 let sb be a new StringBuilder

3 **if** num==null **or** num.length==0 **return** res

4 helper(res,num,target,sb,1,0,0)

5 **return** res

**void helper(List<String> res,String num,int target,StringBuilder sb, int pos,long sum,long preVal)**

1 **if** pos==num.length+1

2 sum+=preVal

3 **if** sum==target

4 res.add(sb.toString())

5 **return**

6 **for** i=pos **to** num.length

7 **if** i≠pos and num[pos]=='0' **break** **//首位是0的话不可以多个数合并成一个数，只能单独作为一位**

8 curVal=Long.parseLong(num[pos...i])

9 originalLength=sb.length()

10 **if** pos==1**//整个表达式的第一个数字不需要携带符号**

11 sb.append(curVal)

12 helper(res,num,target,sb,i+1,sum+preVal,curVal)

13 sb.setLength(originalLength)

14 **else**

15 sb.append("+").append(curVal)

16 helper(res,num,target,sb,i+1,sum+preVal,curVal)

17 sb.setLength(originalLength)

18 sb.append("-").append(curVal)

19 helper(res,num,target,sb,i+1,sum+preVal,**-**curVal)

20 sb.setLength(originalLength)

21 sb.append("\*").append(curVal)

22 helper(res,num,target,sb,i+1,sum,preVal\*curVal)

23 sb.setLength(originalLength)

**关键：将[+-\*/][digit] 作为一个整体**

# 283、将零元素移动到最后面，并且保持其余元素相对顺序

**public void moveZeroes(int[] nums)**

1 moves=0

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

3 **if** nums[i]==0 moves++

4 **else** nums[i-moves]=nums[i]

5 **for** i=nums.length+1-moves **to** nums.length

6 nums[i]=0

# 284、定义带有peek功能的迭代器

**class PeekingIterator implements Iterator<Integer> {**

private Integer peek

private Iterator<Integer> iterator

**public PeekingIterator(Iterator<Integer> iterator)**

1 this.iterator=iterator

2 peek=null

**public Integer peek()**

1 **if** peek==null

2 peek=iterator.next()

3 **return** peek

**public Integer next()**

1 **if** peek≠null

2 int res=peek

3 peek=null

4 **return** res

5 **else** **return** iterator.next()

**public boolean hasNext() {**

1 **if** peek≠null **return** true

2 **else** **return** iterator.hasNext()

# 287、找到重复的元素（n+1个元素，[1-n]）

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

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

2 **if** nums[i]==i **continue**

3 **while** nums[i]≠i

4 **if** nums[nums[i]]==nums[i] **return** nums[i]

5 tem=nums[i]

6 nums[i]=nums[nums[i]]

7 nums[tem]=tem//此时nums[i]已经改变了

8 **return** 0**//if input is right,never return this**

# 289、生死游戏

**Rules:**

1.Any live cell with fewer than two live neighbors dies, as if caused by under-population.

2.Any live cell with two or three live neighbors lives on to the next generation.

3.Any live cell with more than three live neighbors dies, as if by over-population..

4.Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

**状态-->状态转移-->状态**

**\* live-live:2**

**\* live-dead:3**

**\* dead-live:4**

**\* dead-dead:5**

**public void gameOfLife(int[][] board)**

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

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

3 **for** j=1 **to** board[1].length

4 board[i][j]=state(board,i,j)

5 **for** i=1 **to** board.length

6 **for** j=1 **to** board[1].length

7 **if** board[i][j]==2 **or** board[i][j]==4

8 board[i][j]=1

9 **elseif** board[i][j]==3 **or** board[i][j]==5

10 board[i][j]=0

**private int state(int[][] board,int row,int col)**

1 left=row>1? row-1:row

2 right=row<board.length?row+1:row

3 top=col>1? col-1:col

4 bottom=col<board[1].length? col+1:col

5 liveCnt=0

6 **for** i=left **to** right

7 **for** j=top **to** bottom

8 **if** i==row **and** j==col **continue**

9 **else** liveCnt+= (board[i][j]==1||board[i][j]==2||board[i][j]==3)?1:0

10 **if** board[row][col]==1

11 **if** liveCnt<2 **or** liveCnt>3 **return** 3 **//rule1、3**

12 **else** return 2**//rule 2**

13 **else**

14 **if** liveCnt==3 **return** 4 **//rule4**

15 **else** **return** 5

# 290、模式匹配（同构）与205类似

**public boolean wordPattern(String pattern, String str)**

1 strAry=str.split(" ")

2 **if** pattern.length()≠strAry.length **return** false

3 let charPos be a new HashMap<Character,Integer>

4 let wordPos be a new HashMap<String,Integer>

5 **for** i=1 **to** pattern.length

6 c=pattern[i]

7 word=strAry[i]

8 **if** charPos.containsKey(c)

9 lastPos=charPos.get(c)

10 **if** (not wordPos.containsKey(word)) **or** wordPos.get(word) ≠lastPos **return** false

11 charPos.put(c,i)

12 wordPos.put(word,i)

13 **else**

14 **if** wordPos.containsKey(word) **return** false

15 charPos.put(c,i)

16 wordPos.put(word,i)

17 **return** true

# 292、Nim游戏（搬运石块）

**public boolean canWinNim(int n)**

1 **if** n<3 **return** true

2 let dp[1...n] be a new array stored boolean

3 dp[1]=dp[2]=dp[3]=true

4 **for** i=4 **to** n

5 dp[i]=(dp[i-1]&&dp[i-2]&&dp[i-3])? false:true

6 **return** dp[n]

(dp[i-1]&&dp[i-2]&&dp[i-3])表示对于i-1,i-2,i-3块石头，先手必赢时，那么i块石头必输

**public boolean canWinNim(int n)**

1 **if** n%4==0 **return** false

2 **else** **return** true

# 295、实现可以输出中位数的容器

**让较大的一半与较小的一半分开存储**

**class MedianFinder**

**Queue<Integer> large= new PriorityQueue<Integer>()**

**Queue<Integer> small= new PriorityQueue<Integer>(Collections.reverseOrder())**

**public void addNum(int num)**

1large.add(num)

2 small.add(large.poll())**//这里弹出的是large中的最小值**

3 **if** (large.size()<small.size())

4 large.add(small.poll())**//保证large的元素个数不少于small的元素个数**

**public double findMedian()**

**1return** large.size()>small.size()?

large.peek():(large.peek()+small.peek()) / 2.0

# 296、实现树到字符串，字符串到树的转化(对应关系可自己定义，确保可逆即可）

**方法1：若一个节点只有左子树,val+"R"**

**若一个节点只有右子树,val+"L"**

**若一个节点为叶节点,val+"A"**

**private String final leftNull="L",rightNull="R",allNull="A"**

**public String serialize(TreeNode root)**

1 let sb be a new StringBuilder

2 **if** root≠null helper1(root,sb) **//R,L,A必须通过存在的节点的孩子节点的状况进行判断**

3 **return** sb.toString()

**private void helper1(TreeNode root,StringBuilder sb)//需要保证root不为null**

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

2 sb.append(root.val).append(',').append(allNull).append(',')

3 **elseif** root.left==null

4 sb.append(root.val).append(',').append(leftNull).append(',')

5 helper1(root.right,sb)

6 **elseif** root.right=null

7 sb.append(root.val).append(',').append(rightNull).append(',')

8 helper1(root.left,sb)

9 **else** sb.append(root.val).append(',')

10 helper1(root.left,sb)

12 helper2(root.right,sb)

**private int iter//基本类型无法像类对象可以以引用的方式传递，共享唯一一份数据**

**public TreeNode deserialize(String data)**

1 iter=0

2 **if** data.equals("") return null

3 **return** helper2(data.split(","))**//可以保证第一个字符串为数字**

**private TreeNode helper2(String[] strAry)//需要保证iter当前指向的字符串是数子**

1 curVal=Integer.parseInt(strAry[iter++])

2 root=new TreeNode(curVal)

3 **if** strAry[iter].equals(allNull)

4 iter++//skip "A"

5 **elseif** strAry[iter].equals(leftNull)

6 iter++

7 root.right=helper2(strAry)

8 **elseif** strAry[iter].equals(rightNull)

9 iter++

10 root.left=helper2(strAry)

11 **else** root.left=helper2(strAry)

12 root.right=helper2(strAry)

13 **return** root

**另一种思路按前序遍历，当前节点是null就返回添加'X'**

**private final String spliter=",",Null="N"**

**public String serialize(TreeNode root)**

1 let sb be a new StringBuilder

2 helper1(root,sb)

3 **return** sb.toString()

**private void helper1(TreeNode root, StringBuilder sb)**

1 **if** root==null

2 sb.append(Null).append(spliter)

3 **else**

4 sb.append(root.val).append(spliter)

5 helper1(root.left,sb)

6 helpter2(root.right,sb)

**public TreeNode deserialize(String data)**

1 iter=0

2 **return** helper2(data.split(spliter))

**private int iter//基本类型无法像类对象可以以引用的方式传递，共享唯一一份数据**

**private TreeNode helper2(String[] strAry)**

1 **if** strAry[iter].equals(Null)

2 iter++

3 **return** null

4 **else**

5 root=new TreeNode(Integer.parseInt(strAry[iter++]))

6 root.left=helper2(strAry)

7 root.right=helper2(strAry)

8 **return** root

**只有前序遍历无法构建唯一二叉树，但是给出Null节点后便可以了**

# 299、指出完全相同（数值和位置）的数字个数，以及数字相同位置不同的数字个数

**public String getHint(String secret, String guess)**

1 let secretAry[1...10] guessAry[1...10] be new Arrays

2 countA=0

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

4 **if** secret[i]==guess[i] countA++

5 secret[secret[i]-'0']++

6 guess[guess[i]-'0']++

7 countB=0

8 **for** i=1 **to** 10

9 countB+=min(secretAry[i],guessAry[i])

10 countB=countB-countA**//countB包含所有数值相同的数字个数，要减去位置也相同的才是位置不同数值相同的数字个数**

11 **return** countA+"A"+countB+"B"

# 300、最长单调递增子序列

**O(n2)**

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

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

2 let dp[1...nums.length] be a new Array initialized to 1

3 maximum=1

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

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

6 **if** nums[i]>nums[j] dp[i]=max(dp[i],dp[j]+1)

7 maximum=max(maximum,dp[i])

8 **return** maximum

**O(nlgn)**

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

1 let dp[0...nums.length-1] be a new Array

2 len=0

3 **for** each n of nums

4 i=binarySearch(dp,0,len,n)

5 **if** i<0 i=-i-1

6 dp[i]=n

7 **if** i==len len++

8 **return** len