301、除去非法括号

```
public List<String> removeInvalidParentheses(String s)
1 let res be a new List<String>
2 helperOrder(s,1,1,res)
3 return res

private void helperOrder(String s,int pos,int afterLastRemovePos,List<String>res )
1 count=0
2 for i=pos to s.length()
3    if s[i]=='(' count++
4    if s[i]==')' count--
5    if count≥0 continue
6    for j= afterLastRemovePos to i
7       if s[j]==')' and (j==afterLastRemovePos or s[j-1] ≠')')
8          helperOrder(s[1...j-1]+s[j+1...end],i,j,res)
9    return
10 helperReverseOrder(s,s.length(),s.length(),res)

private void helperReverseOrder(String s,int pos,int beforeLastRemovePos,List<String> res)
1 count=0
2 for i=pos downto 1
3    if s[i]==')' count++
4    if s[i]=='(' count--
5    if count≥0 continue
6    for j=beforeLastRemovePos downto i
7       if s[j]=='(' and (j== beforeLastRemovePos or s[j+1] ≠'(')
8          helperReverseOrder(s[1...j-1]+s[j+1...end],i-1,j-1,res)
9    return
10 res.add(s)
```

其中：

对于正向遍历时 afterLastRemovePos，为删除节点 j 的下一个索引，由于删除导致下一个索引与删除前一致，因此递归代入 j

对于反向遍历时 beforeLastRemovePos，为删除节点 j 的前一个节点的索引，由于删除不影响前一个节点的序号，因此递归代入 j-1

合并一下

```
public List<String> removeInvalidParentheses(String s)
1 let res be a new List<String>
2 helper(s,1,1,res,{'(',')'})
3 return res

private void helper(String s,int pos,int afterLastRemovePos,List<String>res,char[] par)
1 count=0
2 for i=pos to s.length()
3     if s[i]==par[1] count++
4     if s[i]==par[2] count--
5     if count≥0 continue
6     for j= afterLastRemovePos to i
7         if s[j]=par[2] and (j==afterLastRemovePos or s[j-1] ≠par[2])
8             helper(s[1...j-1]+s[j+1...end],i,j,res)
9     return
10 reverse=s.reverse()
11 if par[1]==')'
12     res.add(reverse)
13 else
14     helper(reverse,1,1,res,{')','('}
```

302、求区间和

private int[] sumAry

public NumArray(int[] nums)

1 **if** nums==null throw Exception

2 let sumAry[0...nums.length] be a new Arrray

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

4 sumAry[i]=sumAry[i-1]+nums[i]

public int sumRange(int i, int j)//索引 i,j 从 0 开始

1 **return** sumAry[j+1]-sumAry[i]

304、矩阵区间和

```
private int[][] sumAry
public NumMatrix(int[][] matrix)
1 if matrix==null throw Exception
2 row=matrix.length
3 col=matrix.length==0?0:matrix[1].length
4 let sumAry[0...row][0...col] be a new Array
5 for i=1 to row
6    for j=1 to col
7       sumAry[i][j]=sumAry[i][j-1]+sumAry[i-1][j]-sumAry[i-1][j-1]+matrix[i][j]

public int sumRegion(int row1, int col1, int row2, int col2)//索引从 0 开始
1 return sumAry[row2+1][col2+1]-sumAry[row2+1][col1]-
          sumAry[row1][col2+1]+sumAry[row1][col1]
```

第 7 行可以换成

```
sumAry[i][j]+=sumAry[i][j-1]
for k=1 to i
      sumAry[i][j]+=matrix[k][j]
```

或

```
sumAry[i][j]+=sumAry[i-1][j-1]
for k=1 to i
      sumAry[i][j]+=matrix[k][j]
for k=1 to j-1
      sumAry[i][j]+=matrix[i][k]
```

306、加性序列（除了第一个第二个元素外，其余元素均为前两个元素之和）

```
public boolean isAdditiveNumber(String s)
1 n=s.length
2 for i=1 to n-2
3     for j=i+1 to n-1
4         a=parse(s[1...i])
5         b=parse(s[i+1...j])
6         if a==-1 or b==-1 continue
7         if dfs(s[j+1...end],a,b) return true
8 return false
//Line2-7 保证至少可以分成 3 个元素，相当于前两个元素的初始化


boolean dfs(String s, long a, long b) {
1 if s.length==0 return true
2 for i=1 to s
3     c=parse(s[1...i])
4     if c==-1 continue
5     if c==a+b and dfs(s[i+1...end],b,c) return true
6 return false

long parse(String s)
1 if s≠"0" and s[1]=='0' return -1
2 long result=0
3 try
4     result=Long.parseLong(s)
5 catch
6     return -1
7 return result
```

307、NumArray（数据结构，可以随时更新元素，并快速返回范围和）

```
public class NumArray {
    public class RangeSumTreeNode{
      int val
        int start,end,index//范围左右端点序号以该节点
        int sum//以该节点为子树的和
        RangeSumTreeNode left,right
        RangeSumTreeNode(int start,int end)
            this.start=start
            this.end=end
            this.index=start+(end-start>>1)
    }

    private RangeSumTreeNode root

    public NumArray(int[] nums)
    1 root=buildRangeSumTreeNode(nums,0,nums.length-1);

    private RangeSumTreeNode buildRangeSumTreeNode(int[] nums,int start,int end)
    1 if start>end return null
    2 RangeSumTreeNode root=new RangeSumTreeNode(start,end)
    3 root.val=nums[root.index]
    4 RangeSumTreeNode left=buildRangeSumTreeNode(nums,start,root.index-1)
    5 RangeSumTreeNode right=buildRangeSumTreeNode(nums,root.index+1,end)
    6 root.left=left
    7 root.right=right
    8 root.sum=root.val+(root.left==null?0:root.left.sum)+
                          (root.right==null?0:root.right.sum)
    9 return root

    void update(int i, int val)
    1 updateHelper(root,i,val)

    void updateHelper(RangeSumTreeNode root,int i,int val)
    1 if root.index==i
    2   root.val=val
    3 elseif root.index<i
    4   updateHelper(root.right,i,val)
    5 else
    6   updateHelper(root.left,i,val)
    7 root.sum=root.val+(root.left==null?0:root.left.sum)+
                          (root.right==null?0:root.right.sum)



    public int sumRange(int i, int j)
    1 return sumRangeHelper(root,i,j)

    public int sumRangeHelper(RangeSumTreeNode root, int start, int end)
    1 if root==null return 0
    2 if root.end == end and root.start == start
    3    return root.sum
    4 else
```

```
    5    if end < root.index
    6        return sumRangeHelper(root.left, start, end)
    7    elseif start > root.index
    8        return sumRangeHelper(root.right, start, end)
    9    else
    10    return sumRangeHelper(root.right, root.index+1, end) + root.val+
                            sumRangeHelper(root.left, start, root.index-1)
}
```

309、最佳买卖收益（买卖之前必须间隔一天）

public int maxProfit(int[] prices)

```
1 let buy[0...prices.length] be a new Array
2 let sell[0...prices.length] be a new Array
3 let rest[0...prices.length] be a new Array
4 buy[0]=- ∞
5 for day=1 to prices.length
6     buy[day]=max(buy[day-1],rest[day-1]-prices[day])
7     sell[day]=max(sell[day-1],buy[day-1]+prices[day])
8     rest[day]=sell[day-1]
9 return sell[prices.length]
```

buy[i],sell[i],rest[i]分别代表前 i 天以最后一次交易为买，卖，休息的最大收益

public int maxProfit(int[] prices)

```
1 recentBuy=- ∞,secondRecentBuy=0
2 recentSell=0,secondRecentSell=0
3 for each price : prices
4     secondRecentBuy=recentBuy
5     recentBuy=max(recentBuy,recendRecentSell-price)
6     secondRecentSell=recentSell
7     recentSell=max(recentSell,secondRecentBuy+price)
8 return recentSell
```

310、给定图 G(V,E)，给出树高最小的根节点列表

```
public List<Integer> findMinHeightTrees(int n, int[][] edges)
1 let graph[0...n-1] be a new Array stored Set<Integer> initialized with an empty Set<Integer>
2 for each edge: edges
3     graph[edge[0]].add(edge[1])
4     graph[edge[1]].add(edge[0])
5 let leaves be a new ArrayList<Integer>
6 for i=0 to n-1
7     if graph[i].size()==1 leaves.add(i)
8 while n>2
9   n=n-leaves.size()
10  let newLeaves be a new ArrayList<Integer>
11  for each i :leaves
12      j=graph[i].iterator().next()//取得邻接链表中的下一个元素
13      graph[j].remove(i)//为了删除更方便，Adj 设置成 Set
14      if graph[j].size()==1 newLeaves.add(j)
15  leaves=newLeaves
16 return leaves
```

312、依次引爆气球的最大得分

由于当前气球引爆所得到的分数不取决于之前已经引爆的气球，因此我们对问题进行的选择为：选择哪一个气球最后引爆，而不是选择哪一个气球最先引爆

```
public int maxCoins(int[] nums)
1 n=nums.length
2 let expendNums[0...n-1] be a new Array//用于存放边界
3 expendNums[0]= expendNums[n-1]=1
4 for i=1 to nums.length
5    expendNums[i]=nums[i]
6 let dp[0...n-1][0...n-1] be a new Array
7 for len=3 to n
8    for begin=0 to n-len
9       end=begin+len-1
10       for lastBurstDex=begin+1 to end-1
11          dp[begin][end]=max(dp[begin][end],
                        expendNums[begin]*expendNums[lastBurstDex+expendNums[end]+
                        dp[begin][lastBurstDex]+dp[lastBurstDex][end]
13 return dp[0][n-1]
```

begn 与 end 为当前区域的边界，当前区域的范围为 begin+1...end-1

313、Ugly 序列的第 n 个

```
public int nthSuperUglyNumber(int n, int[] primes)
1 let ugly[1...n] be a new Array
2 ugly[1]=1
3 let indexs[1...primes.length] be a new Array
4 let factors[1...primes.length] be a new Array
5 for i=1 to primes.length
6     factors[i]=primes[i]
7 for i=2 to n
8     minimum=minOfFactors(factors)
9     ugly[i]=minimum
10    for j=1 to factors.length
11        if factors[j]==minimum
12            factors[j]=primes[j]*ugly[indexs[j]+1]
13            indexs[j]=index[j]+1
13 return ugly[n]
```

Line10-13：对于每个素数的递增序列进行更新，因为有可能同时多个值为最小值每个需要更新素数序列的值为 ugly 的下一个值（该素数序列所存储的索引+1）

与 Code264 类似

315、第 i 个元素后小于元素 i 值得元素的个数

利用含有额外属性的搜索二叉树

```
class Node{
      Node left,right
      int val,cnt,repeat//cnt 表示左子树的节点总数，repeat 表示值为 val 的元素的重复次数
      public Node(int val)
      1 this.val=val this.cnt=0 this.repeat=1
}
private Node root
public List<Integer> countSmaller(int[] nums)
1 let res be a new List<String>
2 root=null
3 for i=nums.length downto 1
4    insert(nums[i],res)
5 return res
```

必须在插入的过程中不断更新 res，而不是全部插入完后才更新

```
private void insert(int num,List<Integer> res)
1 Node cur=root,pre=null
2 count=0//存储搜索路径中小于当前 num 的元素个数（这些元素个数不包括在新建或查找到的节点的 cnt 中）
3 while cur≠null
4    pre=cur
5    if num==cur.val
6       cur.repeat++
7    elseif num<cur.val
8       cur.cnt++
9       cur=cur.left
10   else
11      count=count+cur.cnt+cut.repeat
12      cur=cur.right
13 if cur≠null
14    res.add(0,cur.cnt+count)
15 else
16    if pre==null root=new Node(num)
17    elseif (num<pre.val) pre.left=new Node(num)
18    else pre.right=new Node(num)
19    res.add(0,count)
```

利用二叉索引树（BIT）

```
class BITNode{
    BITNode left,right
    int val,num,repeat//num 为以该节点为根的子树的节点总数（包括该节点）repeat 为重复次数
    BITNode BITNode(int val)
    1 this.val=val,this.num=1,this.repeat=1
}
private BITNode root
public List<Integer> countSmaller(int[] nums)
1 let res be a new List<Integer>
2 root=null
3 for i=nums.length downto 1
4    insert(nums[i],res)
5 return res

private void insert(int num,List<Integer> res)
1 BITNode cur=root,pre=null
2 count=0
3 while cur≠null
4    pre=cur
5    cur.num++
6    if cur.val==num
8       cur.repeat++
9       break
10   elseif num<cur.val
11      cur=cur.left
12   else
13      count=count+(cur.left==null?0:cur.left.num)+cur.repeat
14      cur=cur.right
15 if cur≠null
16    res.add(0, (cur.left==null?0:cur.left.num)+count
17 else
18    if pre==null root=new BITNode(num)
19    elseif num<pre.val pre.left=new BITNode(num)
20    else pre.right=new BITNode(num)
21    res.add(0,count)
```

316、除去重复字母使得余下的字母组合最小

贪婪递归求解

```
public String removeDuplicateLetters(String s)
1 if s==null or s.length()<2 return s
2 let cnt[1...26] be a new Array
3 pos=0
4 for i=1 to s.length() cnt[s[i]-'a']++
5 for i=1 to s.length()
6    if s[i]<s[pos] pos=i
7    if --cnt[s[i]-'a']==0 break //这个字母在 s[i+1...end]不再会出现，因此必须添加，如果不退出循环，可能 pos 会定位到 i 之后的地方，就把这个字符给忽略了。
8  return  s[pos]+  removeDuplicateLetters(s[pos+1...end].replaceAll(s[pos]+"",""))// 由于选择了 pos 位置上的字符（如果该字符有重复，那么一定是第次出现的位置，即贪婪），因此应该把 s[pos+1...end]中出现的该字符都除去
```

```
public String removeDuplicateLetters(String s)
1 let count[1...26] be a new Array
2 let visited[1...26] be a new Array Stored boolean initialized to false
3 charAry=s.toCharArray()
4 for each c:charAry
5    count[c-'a']++
6 let stack be a new empty Stack<Character>
7 for each c:charAry
8    count[c-'a']--//即使该元素已经存在于栈中，也需要执行该语句，因为 count 是表明在剩余序列中该字符出现的次数
9    if visited[c-'a'] continue
10   while not stack.isEmpty() and stack.peek()>c and count[stack.peek()-'a']>0
     //当栈顶元素比当前元素大，并且栈顶元素还存在于剩余部分时，将其弹出
11      visited[stack.pop()-'a']=false
12   stack.push(c)//每一次循环，只要当前字符不在栈中，必将其压入栈
13   visited[c-'a']=true
14 let sb be a new StringBuilder
15 while not stack.isEmpty()
16    sb.append(stack.pop())
17 return sb.reverse().toString()
```

318、不包含重复字母的单词的最大积

```
public int maxProduct(String[] words)
1 let letterState[1...words.length] be a new Array
2 for i=1 to words.length
3     for each c:words[i]
4         letterState[i]|=1<<c-'a'
5 maximum=0
6 for i=1 to words.length
7     for j=i+1 to words.length
8         if (letterState[i]&letterState[j])==0
9             maximum=max(maximum,words[i].length*words[j].length
10 return maximum
```

由于需要判断任意两个单词是否含有重复字母，如果直接对单词中每个字母进行判断，复杂度至少为 O(N)，但是如果借助于 bit 操作，判断可在 O(1)的时间内完成，即可以同时判断

letterState[i]存储的是第 i 个单词的字母情况，由于单词为 26 个字母，因此 int 型（32 位 bit）整数即可存储该单词的 26 个字母的出现情况，最低位代表 a，第 26 位代表 z

319、n 次循环后，亮灯数量

最初时候，灯泡全灭

第 i 次循环：变换第 i,2*i,3*i...位置上的灯泡的状态（开变关，关变开）

也就是说，对在第 i 个位置的灯泡

前 i 次循环到该位置上次数为奇数时，灯泡状态为开

对于一个整数 n，将其分解成两个数的乘积（若乘积因子不同，可以交换顺序）

若 n 不为平方数，那么可分解的数量为偶数次

如 2=1*2=2*1　6=1*6=6*1=2*3=3*2

若 n 为平方数，由于平方分解乘积因子不可交换顺序，因此可分解的数量为奇数

如 4=1*4=4*1=2*2

对于 i+1 开始的循环，不可能再改变第 i 个灯泡的状态

因此，结果为 sqrt(n)

int bulbSwitch(int n)

1 return sqrt(n)

321、两个数组求合并后的 k 位最大数组

public int[] maxNumber(int[] nums1, int[] nums2, int k)

```
1 len1=nums1.length
2 len2=nums2.length
3 start=Math.max(k-len2,0)
4 end=Math.min(len1,k)
5 let res[1...k] be a new Array
6 for i=start to end
7     tem=merge(maxNumberOfSingleArray(nums1,i),maxNumberOfSingleArray(nums2,k-i))
8     if isLarge(tem,res,1,1) res=tem
9 return res
```

从 nums1 取出一个最大的数组长为 x，nums2 中取出一个最大的数组长为 y，满足 x+y=k, 0≤x≤len1, 0≤y≤len2 ---> Math.max(k-len2,0) ≤x≤ Math.min(len1,k)

private int[] merge(int[]nums1, int[] nums2)//合并两个未排序数组，使得合并后的数组最大

```
1 let res[1...nums1.length+nums2.length] be a new Array
2 iter1=1,iter2=1,i=1
3 while i≤res.length
4     res[i++]=isLarge(nums1,nums2,iter1,iter2)?nums1[iter1++]:nums2[iter2++]
5 return res
```

private boolean isLarge(int[] nums1,int[] nums2,int i,int j)

```
1 while i≤nums1.length and j≤nums2.length and nums1[i]==nums2[j]
2     i++,j++
3 return j==nums2.length or i<nums1.length and nums1[i]>nums[j]
```

private int[] maxNumberOfSingleArray(int[] nums, int k)//求最大非连续子数组（元素相对顺序不变，但子数组中的元素在原数组中可以不连续）

```
1 n=nums.length
2 let res[1...k] be a new Array
3 j=1
4 for i=1 to n
5     while n-i+j>k and j>1 and res[j-1]<nums[i] j--
6     if j≤k res[j++]=nums[i] //j 可能会大于 k，此时，不用更新
7 return res
```

伪代码中：位置索引从 1 开始

当前 res 中的"指针"：j（位置 j 尚未填写），剩余 k-j+1 个元素需要填

nums 中剩余的元素（包括第位置 i 上的元素）共有 n-i+1 个

因此必须满足 n-i+1>k-j+1 --> n-i+j>k

为什么是大于而不是大于等于，因为循环条件成立后会递减 j，需要保证递减后也能满足 nums 剩余元素能填满 res 剩余位置，因此当前判断需要使得 nums 中剩余元素至少比 res 剩余位置多一个，才有递减 j 的资本

322 兑换零钱的最少数量

```
public int coinChange(int[] coins, int amount)
1 let dp[0...amount] be a new Array initialized to +∞
2 dp[0]=0
3 for i=1 to amount
4     for j=1 to coins.length
5         if coins[j] ≤ i
6             dp[i]=min(dp[i],dp[i-coins[j]]+1)
7 return dp[amount]== +∞? -1:dp[amount]
```

注意：伪代码中假设+∞+1=+∞，即不考虑溢出

程序中可以将 dp 先赋值为 amount+1，因为可兑换的张数不会超过这个数量，防止当前类型的最大值+1 后溢出

324、将数组排列成 a<b>c<d>e...

326、判断是否是 3 的幂次

public boolean isPowerOfThree(int n)

```
1 while n≥3
2     if n%3≠0 return false
3     n=⌊ n/3 ⌋
4 return n==1
```

331、判断前序遍历（包含空节点）是否是一颗合法的二叉树

将一颗二叉树的叶节点定义为空节点

定义度：每一个非空节点包含 2 个 outdegress 和 1 个 indegree（2 孩子 1 双亲）

每一个空节点包含 0 个 outdegree 和 1 个 indegree（0 孩子 1 双亲）

每当遇到一个节点，如果它是空节点，提供的度为-1，如果它是非空节点提供度为 1

```
public boolean isValidSerialization(String preorder)
1 if preorder==null or preorder.length==0 return false
2 dif=1
3 strAry=preorder.split(",")
4 for each s:strAry
5    if --dif<0 return false//无论什么节点，先-1，不用讨论边界情况
6    if not s.equals("#") dif+=2
7 return dif==0
//如果换成这种写法，与利用高度来判断是完全一样的
public boolean isValidSerialization(String preorder)
1 if preorder==null or preorder.length==0 return false
2 dif=0//这里 dif 可以代表另一种含义，即高度
3 strAry=preorder.split(",")
4 for int i=1 to strAry.length-1
5    if strAry[i].equals("#")
6       if --dif<0 return false
7    else dif++
8 return dif==0 and strAry[strAry.length].equals("#")

public boolean isValidSerialization(String preorder)
1 if preorder==null or preorder.length==0 return false
2 let stack be a new Stack stored String
3 strAry=preorder.split(",")
4 for each s:strAry
5    while s.equals("#") and not stack.isEmpty() and stack.top().equals("#")
//为什么要用 while,参考例子"9,3,4,#,#,1,#,#,2,#,6,#,#"理解
6       stack.pop()//这个弹出的空节点与 s 的双亲为同一个非空节点
7       if stack.isEmpty() return false
8       stack.pop()//将这个非空节点弹出，随后压入当前的 s（空节点），替换
9    stack.push(s)
10 return stack.size()==1 and stack.top().equals("#")
```