# ****1、TwoSum****

**int[] twoSum(int[] nums, int target)**

1 n=nums.length

2 let result[1...2] be a new array

3 let map be a new Map stored <int,int>

4 **for** i=1 **to** n

5 **if** map.containsKey(nums[i])

6 result[1]=map.get(nums[i])

7 result[2]=i

8 **else** map.put(target-nums[i],i)

9 **return** result

对于每个i，将目标值与nums[i]相减后的值作为key，存入map

# 2、addTwoNumbers

**public ListNode addTwoNumbers(ListNode l1, ListNode l2)**

1 let pesudoHead be a new ListNode

2 carry=0

3 iter1=l1,iter2=l2,iter=pseudoHead

4 **while** iter1≠null **and** iter2≠null

5 sum=carry+iter1.val+iter2.val

6 carry=sum/10

7 sum=sum%10

8 iter.next=a new ListNode with val=sum

9 iter=iter.next

10 iter1=iter1.next,iter2=iter2.next

11 **while** iter1≠null

12 sum=carry+iter1.val

13 carry=sum/10

14 sum=sum%10

15 iter.next=a new ListNode with val=sum

16 iter=iter.next

17 iter1=iter1.next

18 **while** iter2≠null

19 sum=carry+iter2.val

20 carry=sum/10

21 sum=sum%10

22 iter.next=a new ListNode with val=sum

23 iter=iter.next

24 iter2=iter2.next

25 **if** carry≠0

26 iter.next=a new ListNode with val=carry

27 **return** pesudoHead.next

# ****3、最长不重复子数组（求包含元素的个数）****

**lengthOfLongestSubstring(String s)**

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

2 last=1,M=1

3 **for** i=2 **to** Length

4 **for** j=i-1 **to** last

5 **if**(s[j]==s[i])

6 last=j+1 **and** break

7 M=max(M,i-last+1)

8 **return** M

**外层循环的循环不变式：**

**每次迭代开始时,s[last...i-1]都是一个不重复子数组，故只需要检查s[i]与s[last...i-1]是否有重复即可，若有则更新last**

**每次迭代结束后,s[last...i]都是一个不重复子数组**

**Brilliant**

**public int lengthOfLongestSubstring(String s)**

1 right=0,left=0

2 let cnt be a new array with size=128

3 maximum=0

4 **while** right<s.length()

5 c=s[right]

6 cnt[c]++

7 **while** cnt[c]>1

8 cnt[s[left++]]--

9 maximum=max(maximum,right-left+1)

10 **return** maximum

**动态规划**

**设计的子问题形式，可以与原问题不同，可以将问题改变为以求S[i]元素为结尾的最长不重复子数组的长度）**

**设计思路：定义C[i]为以S[i]结尾的最长不重复子数组的长度**

**lengthOfLongestSubstring(s)**

1 L=s.length

2 let c[1...L] be new tables

3 **for** i=1 **to** L c[i]=1

4 Max=1

5 **for** i=2 **to** L

6 j=i-1

7 **while** j≥ i-c[i-1]

8 **if** s[j]==s[i] **break**

9 **else** j--

10 c[i]=i-j//从j+1...i为以s[i]结尾的最长不重复子数组

11 Max=max(Max,c[i])

11 **return** Max

# ****4、findMedianSortedArrays****

4.1 问题陈述：两个已排序的数组A1A2，长为m和n，在O(lg(m+n))的时间内返回中位数

**① ②**

 

**对于情况①：证明顺序数为k的元素必定存在于L1[s1+t1...e1]和L[s2...s2+t2-1]中**

A、假设在L1[s1...s1+t1-1]中存在顺序数为k的元素L1[s1+j-1] ,1≤j≤t1，故L1[s1+j-1]≤L1[s1+t1-1]。**由于L2[s2+t2-1]> L1[s1+t1-1]≥L1[s1+j-1],因此L2[s2...e2]中顺序数比L1[s1+j-1]小的元素只可能存在L2[s2...s2+t2-2]中，最多为t2-1个。而L1[s1...e1]中顺序数比L1[s1+j-1]小的元素存在L1[s1...s1+j-2]中，为j-1个。t2-1+j-1≤t2-1+t1-1=t1+t2-2=k-2，因此L[s1+j-1] 最多为顺序数k-1的元素,不可能是顺序数为k的元素。**

B、假设在L2[s2+t2...e2]中存在顺序数为k的元素L2[s2+j-1], t2+1≤j≤e2，故L2[s2+j-1]≥L2[s2+t2-1]。**由于L1[s1+t1-1]< L2[s2+t2-1]≤L2[s2+j-1],因此L1[s1...e1]中顺序数比L2[s2+j-1]小的元素可能存在于L1[s1...e1]中，最少为t1个。而L2[s2...e2]中顺序数比L2[s2+j-1]小的元素为t2个。t1+t1=k，因此L2[s2+j-1]最少为顺序数k+1的元素，不可能是顺序为k的元素。**

**对于情况②：同理**

4.6 中位数：（A[(n+1)/2]+A[(n+2)/2]）/2 //默认向下取整

**findMedianSortedArrays(nums1,nums2)**

1 n1=nums1.length,n2=nums2.length

2 **return** (Aux(nums1,1,n1,nums2,1,n2,(n1+n2+1)/2)+

Aux(nums1,1,n1,nums2,1,n2,(n1+n2+2)/2))/2//利用4.6的公式

**Aux(nums1,s1,e1,nums2,s2,e2,k)**

1 n1=e1-s1+1,n2=e2-s2+1

2 **if**(n1>n2) **return** Aux(nums2,s2,e2,nums1,s1,e1,k)**//保证表1长度小于表2，便于line 5计算**

3 **if**(n1==0) **return** nums2[s2+k-1]

4 **if**(k==1) **return** min(nums1[s1],nums2[s2])**//为什么这句是必须的，k=1会导致s1+t1-1越界**

5 t1=min(k/2,n1),t2=k-t1//若表12长度不定，那么这里需要分类讨论

6 **if** nums1[s1+t1-1]<nums2[s2+t2-1]

7 **return** Aux(nums1,**s1+t1**,e1,nums2,s2,**s2+t2-1**,k-t1)

8 **elseif** nums1[s1+t1-1]>ums2[s2+t2-1]

9 **return** Aux(nums1,s1,**s1+t1-1**,nums2,**s2+t2**,e2,k-t2)//详见情况①的分析

10 **return** nums1[s1+t1-1]//相等说明顺序数为k的数存在重复，返回这俩任意一个都行

**findMedianSortedArrays(nums1,nums2)**

1 n1=nums1.length,n2=nums2.length

2 **return** (Aux(nums1,1 ,nums2,1,(n1+n2+1)/2)+

Aux(nums1,1,nums2,1,(n1+n2+2)/2))/2//利用4.6的公式

**Aux(nums1,s1,nums2,s2,k)**

1 **if** s1==nums1.length+1 **return** nums2[s2+k-1]

2 **if** s2==nums2.length+1 **return** nums1[s1+k-1]

3 **if** k==1 **return** min(nums1[s1],nums2[s2])

4 Mid1=∞，Mid2=∞

5 **if** s1+k/2-1<=nums1.length Mid1=nums1[s1+k/2-1]//5,6两行至少有一行为ture

6 **if** s2+k/2-1<=nums2.length Mid2=nums2[s2+k/2-1]

7 **if** Mid1<Mid2 **return** Aux(nums1,**s1+k/2**,B,s2,k-k/2)

8 **else** **return** Aux(nums1,s1,B,**s2+k/2**,k-k/2)//不会出现Mid1=Mid2=∞的情况

**//这里为什么不讨论Mid1==Mid2：因为k/2+k/2不一定等于k**

# ****5、回文序列****

**String longestPalindrome(String s)**

1 Maximum=0,left=1

2 **if** s.length==0 return “”

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

4 m=i,n=i;

5 **while** n<=s.length **and** s[i]==s[n] n++//5、6两行找到与s[i]相同的边界

6 **while** m>=0 **and** s[i]==s[m] m--

7 **while** m>=0 **and** n<=s.length and s[m]==s[n]//向两边扩展

8 m--

9 n++

10 **if** Maximum<n-m-1 //(n-1)-(m+1)+1=n-m-1

11 Maximum=n-m-1

12 left=m+1

13 **return** s[left...left+Maximum-1]

**start=0,len=0**

**String longestPalindrome(String s)**

1 **if** s.length<2 **return** s

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

3 Aux(s,i,i)**//对称中心为一个元素**

4 Aux(s,i,i+1)**//对称中心为两个元素**

5 **return** s[start...start+len]

**Aux(String s,int left, int right)**

1 **while** left>=1 **and** right<=s.length **and** left]==s[right]

2 left--,right++

3 **if** right-left-1>len

4 len=right-left-1

5 start=left+1

**String longestPalindrome(String s) DP:**

1 **if** s==null **return** null

2 start=-1,len=-1

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

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

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

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

7 M[j][i]=true

8 **if** i-j+1>len

9 len=i-j+1

10 start=j

11 **return** s[start...start+len-1]

**动态规划：与最长单调子序列相同，逆序求最长公共子序列**

**如果子序列可以不连续取，那么与最长单调子序列相似，逆序后求最长公共序列即可**

**如果子序列必须连续取值，子问题设计：c[i]保存以s[i]结尾的回文序列**

**c[i]=c[i-1]+2或c[i]重新计算**

**① ②**

 

**String longestPalindrome(String s)**

1 n=s.length

2 let c[1...n] be a new array

3 c[1]=1

4 **for** i=2 **to** n

5 j=i-1-c[i-1]// ①中左边红色处或②中蓝色处的索引 (i-1-c[i-1]+1)-1

6 **if** j>0 **and** s[j]==s[i] c[i]=c[i-1]+2 //情况①

7 **else** //情况②

8 **for** k=j+1 to i//由于不是情况①，所以向右寻找以s[i]结束的回文序列

10 left=k,right=i

11 **while** left<right **and** s[left]==s[right]

12 left++,right--

13  **if** left≥right//当s[k...i]为回文序列时

14 c[i]=i-k+1

15 **break**

16 **return** s[left...right]//这里根据最大的c[index]来取出回文序列

# **7、整数反转**

**for** (; x != 0; x /= 10)

res = res \* 10 + x % 10;

7.1 要处理旋转后溢出的情况

# ****9、回文数字（并非子序列，而是整个序列）****

9.1 一般思路：转为数组后，比较对称的两项是否相等（注意负数不属于回文）

9.2 较好思路：参考整数反转7，直接比较反转前后是否相等

# 10、包含“.”和“\*”的正则表达式

**isMatchRecursion(S,P,i,j);//从S(i)与P(j)开始比较，默认之前已经匹配成功**

1 **while**( i==S.length **and** **(**getChar(P,j)==’\*’ **or** getChar(P,j+1)==’\*’**)**) j++//’\*’可匹配0个，故跳过

2 if i==S.length **and** j==P.length return true//如果都到了末端，则匹配成功

3 if i==S.length **or** j==P.length return false//如果只有一个到了末端，则匹配失败

3 if getChar(P,j+1)==’\*’ and **(**getChar(P,j)==’.’ || getChar(P,j)==getChar(S,i)**)**

4 return isMatchRecursion(S,P,**i+1,j**)**||** isMatchRecursion(S,P,**i,j+2**)

5 if getChar(P,j+1)==’\*’ return isMatchRecursion(S,P,i,j+2)// 隐含条件getChar(P,j)≠getChar(S,i)

6 if **getChar(P,j)==’.’**||getChar(P,j)== getChar(S,i) return isMatchRecursion(S,P,i+1,j+1)

7 return false

**getChar(S,i)**

1 **if** i>=1 **and** i<=S.length

2 **return** S[i]

3 **else return** ‘\0’//否则返回空字符

①若，否则继续往下走

 OR 

此时：S已经全部匹配完，但是正则表达式尚未结束，并且后跟\*，循环递增j，若不满足上述两个条件则跳出循环，继续往下走。

②若，否则继续往下走



此时S与正则表达式都到了边界点，即完全匹配，返回ture。

③若，否则继续往下走

 OR 

此时不能匹配成功，返回false。

④若，否则继续往下走

 OR 

若S(i)与P(j)匹配，则有两种可能，该正则表达式成功匹配S(i)，继续与S(i+1)尝试匹配;或者，该正则表达式选择不匹配S(i)**(尽管是可以匹配的,例如”aaa”,”a\*aa”)**，则s(i)与p(j+2)尝试匹配，即返回isMatchRecursion(i+1,j)|| isMatchRecursion(i,j+2)。

⑤若，否则继续往下走



S(i)与P(j)不匹配，则继续尝试匹配S(i)与P(j+2),即返回isMatchRecursion(i,j+2)

⑥若，否则继续往下走

 OR 

S(i)与P(j)匹配，则继续尝试匹配S(i+1)与P(j+1)，即返回isMatchRecursion(i+1,j+1)

⑦返回false

**动态规划求解：效率更高**

**子问题设计：match[i][j]代表以索引i为起始的匹配子串与以索引j为起始的模式字串是否匹配，其中mat[s.length+1][p.length+1]=true**

**boolean isMatch(String s, String p)**

1 let match[1...s.length+1][1...p.length+1] be a new array initialized to flase

2 match[s.length+1][p.length+1]=true

3 i=p.length()

4 **while** getChar(p,i)==’\*’ **and** getChar(p,i-1)!=’0’ //先处理”” 与a\*b\*c\*...z\*匹配的问题

5 match[s.length()+1][i-1]=true

6 i=i-2

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

8 **for** j=p.length **to** 1

9 **if** getChar(p,j)==’\*’ j—

10 **if** getChar(p,j+1)==’\*’ **and** (getChar(p,j)==’.’ **or** getChar(p,j)==getChar(s,i))

11 match[i][j]=match[i][j+2] **or** match[i+1][j]

12 **elseif** getChar(p,j+1)==’\*’

13 match[i][j]=match[i][j+2]

14 **elseif** getChar(p,j)==’.’ **or** getChar(p,j)==getChar(s,i)

15 match[i][j]=match[i+1][j+1]

16 **else** match[i][j]=false

17 **return** match[1][1]

Line10 的两种情况：



s[i]与p[j]是可以匹配的，但是由于\*的存在

**①s[i]与p[jj+1]匹配，此时match[i][j]=match[i+1][j]&&(s[i] matchs p[(j)(j+1)])**

**②s[i]不与p[jj+1]匹配，此时match[i][j]=match[i][j+2]&&(none matchs p[(j)(j+1)])**

**子问题设计：match[i][j]代表以索引i为终点的匹配子串与以索引j为终点的模式字串是否匹配，其中mat[0][0]=true**

**boolean isMatch(String s, String p)**

1 let match[0...s.length][0...p.length] be a new array initialized to false

2 match[0][0]=true

3 i=1

4 **while** getChar(p,i+1)==’\*’

5 match[0][i+1]=true

6 i+=2

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

8 **for** j=1 **to** p.length

9 **if** getChar(p,j+1)==’\*’ j++

10 **if** getChar(p,j)==’\*’ and (getChar(p,j-1)==’.’ or getChar(p,j-1)==getChar(s,i))

11 match[i][j]=match[i-1][j] or match[i][j-2]

12 **elseif** getChar(p,j)==’\*’

13 match[i][j]=match[i][j-2]

14 **elseif** getChar(p,j)==’.’ or getChar(p,j)==getChar(s,i)

15 match[i][j]=match[i-1][j-1]

16 **else** match[i][j] false

17 **return** match[s.length][p.length]

# 11、最大灌水量

11.1 简单思路：遍历所有种可能n2

11.2 **两端向中间循环，遍历一次**

若当前左右索引分别为left,right。若height[left]<height[right]，如果固定左端点left，向左移动右端点right，**所能够得到的最大容量就是height[left]\*(right-left)（因为容器的有效高度不会超过height[left]）**。因此，需要向右移动左端点，可能获取更大的容量（容器的高度可能会高于height[left]）；同理若height[left]>=height[right]，则左移右端点

**Container With Most Water(height)**

1 Maximum=0

2 left=1,right=height.length

3 **while**(left<right)

4 Maximum=max(Maximum,(right-left)\*min(height[left],height[right])

5 **if** height(left)<height(right) left++;

6 **else** right--;

**return** Maximum

# 12、intToRoman

# 13、romanToInt

13.1 受到正则表达式的影响，采用相似的结构进行转换

# 14、longestCommonPrefix（String数组的最长公共前缀）

for index=1 to MimLength

for i=1 to strs.length

...

# 15、threeSum(nums)

1 let List<List<int>> L be new List

2 **sort(nums)**

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

4 **if i>1 and nums[i]==nums[i-1] continue//与前一个元素相同，跳过**

5 j=i+1,k=nums.length

6 **while**(j<k)

7 **if** nums[i]+nums[j]+nums[k]==0

8 L.add(nums[i],nums[j],nums[k])

9 ++j,--k

10 **while(j<k and nums[j]==nums[j-1])++j//跳过相同元素**

11 **while(j<k and nums[k]==nums[k+1])—k//跳过相同元素**

12 **elseif** nums[i]+nums[j]+nums[k] > 0 --k

13 **else** ++j

14 return L

# 16、threeSumClosest与15类似

# 18、4Sum与15类似

# 17 手机数字转字母组合

word= {"abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};

**letterCombinations(String digits)**

1 List<String> Res be a new List

2 **if** digits.length==0 **return** lst

3 let Pre be a new StringBuilder

4 Aux(digits,1,Pre,Res)

5 **return** Res

**Aux(String digits,int dex,StringBuilder Pre,List<String> Res)**

1 if dex==digits.length+1

2 Cur=Pre.toString()

3 Res.add(Cur)

4 return

5 num=charToInt(digits[dex]) //将对应的’2’’3’...’9’等等转为1 2...8

6 for i=1 to word[num].length

7 Pre.append(word[num][i])

8 Aux(digits,dex+1,Pre,Res)

9 Pre.remove(Pre.size())

# 20、合法的括号

**isValid(String s)**

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

2 let stk be a new Stack**//栈**

3 **return** Aux(stk,s,1)

Aux(stk,s,index)

1 **if** index>s.length **and** stk.length==0 **return** true//当已经匹配完，且栈为空

2 **if** index>s.length **return** false//已经匹配完，但是此时栈中还有未匹配完的元素

3 **if** isLeftHalf(s[index])

4 stk.push(s[index])

5 **return** Aux(stk,s,index+1)

6 **if** isRightHalf(s[index]) **and** stk.length==0 **return** false//当出现右半括号，但栈为空

7 **if** isRightHalf(s[index]) **and** isMatch(stk.topelement,s[index])//出现右半括号且匹配栈顶元素

8 stk.pop

9 **return** Aux(stk,s,index+1)

10 **return** false

# 21、合并两个有序链表，太简单，略

**22、给定括号对数n，列出所有可能的括号,与17题手机字母组合类似**

**public List<String> generateParenthesis(n)**

1 let lst be a new List

2 if n==0 return L

3 left=0,right=0

4 s=””

5 Aux(lst,s,left,right,n)

6 **return** lst

**Aux(lst,s,left,right,n)**

1 **if** left==n **and** right==n lst.add(s) //此时已经打完所有的括号

2 **elseif** left==n Aux(lst,s+”)”,left,right+1,n)

3 **elseif** left==right Aux(lst,s+”(”,left+1,right,n)

4 **else** Aux(lst,s+”(”,left+1,right,n)

5 Aux(lst,s+”)”,left,right+1,n)

**//由于String比较特殊，不必考虑回溯后状态的恢复**

# 23、K个有序链表的链接

**每个链表取一个元素放在数组lst中，每次取出lst的最小值，然后补上这个最小值对应的ListNode的下一个元素next，用最小堆来组织这个lst**

**ListNode mergeKLists(ListNode[] lists)**

1 L=lists.length

2 **if**(L==0) **return** null

3 let lst[1...L] be a new array storing ListNode

4 **for** i=1 **to** L

5 lst[i]=lists[i]

6 **for** i=L/2 **to** 1

7 MinHeap(lst,i)

8 result=null,cur=null

9 **while** lst[0]!=null

10 **if** result==null

11 result=lst[0]

12 cur=result

13 Replace(lst,lst[0].next)

14 **else** cur.next=lst[0]

15 cur=cur.next;

16 Replace(lst,lst[0].next)

17 **return** result

**GetVal(ListNode LN)**//只要有利用lst读取val的时候，就用该函数读取，因为lst的元素可能为null，需要让null的元素排在最后面

1 **if** LN==null return ∞

2 **return** LN.val

**MinHeap(ListNode[] lst, int i)**

1 L=2\*i,R=2\*i+1

2 **if** L<=lst.length **and** GetVal(lst[L])<GetVal(lst[i]) Small=L

3 **else** Small=i

4 **if** R<=lst.length **and** GetVal(lst[R])< GetVal(lst[Small]) Small=R

5 **if** Small≠i

6 Exchange(lst,Small,i)

7 MinHeap(lst,Small)

**Replace(ListNode[] lst,ListNode LN)**

1 lst[0]=LN

2 MinHeap(lst,1)

**关于最大\小堆，将根节点移除的话，需要再进行一次构建最小堆，仅仅对新的根节点维护堆的性质是不对的，因为对应关系都变了**

**还可以用优先队列组织元素**

# 25 链表每k个元素反转（包括24题在内）

**ListNode reverseKGroup(ListNode head, int k)**

1 **if** head==null **or** k<2 **return** head

2 let ary[1...k] be a new array stored ListNode

3 cur=head,pre=null

4 **while**(cur≠null)

5 i=1

6 **while** cur≠null **and** i<=k //每k个元素处理一次

7 ary[i++]=cur

8 cur=cur.next

9 **if** pre==null **and** i>k //若这k个元素为第一组，需要处理表头

10 head=ary[k]

11 **for** j=k **to** 2 ary[j].next=ary[j-1]

12 ary[1].next=cur

13 pre=ary[0]

14 **elseif** i>k //若这k个元素不为第一组

15 pre.next=ary[k]

16 **for** j=k **to** 2 ary[j].next=ary[j-1]

17 ary[1].next=cur

18 pre=ary[1]

19 **else** **break** //没有取满k个元素，已经到了表尾，退出循环即可

20 **return** head

# 26、不利用额外空间，计算出有序数组的不重复元素的个数L，并且处于数组前L个位置

**int removeDuplicates(int[] nums)**

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

2 swapped=2

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

4 **if** nums[i-1]<nums[i] //当前元素与前一个元素不同，那么必定为一个新的元素

5 nums[swapped++]=nums[i]

6 **return** swapped-1//1...swapped-1 才是不重复元素

**同理，如果数组不是有序的话，首先排个序，当然得用原址排序（插入，快排等等）**

# 27 不利用额外空间，除去所有指定的元素，返回剩余元素个数L，并且占据前L个位置

**removeElement(int[] nums, int val)**

1 if nums.length==0 return 0

2 swapped=1

3 for i=1 to nums.length

4 if nums[i]≠val

5 nums[swapped++]=nums[i]

6 return swaped-1//1...swapped-1 才是非val的元素

**特点：swapped所指向的位置总是不大于循环变量i所指向的位置，因此可以改变i之前的数据（因为已经进行判断过）**

# 28、字符串匹配（KMP算法）

**int strStr(String T, String P)**

1 n=T.length

2 m=P.length

3 π=Aux(P)

4 k=0

5 for q=1 to n

6 while k>0 and P[k+1]≠T[q]

7 k=π[k]

8 if P[k+1]==T[q]

9 k++

10 if k==m

11 return i-m+1

**//模式字符串P的子串Pk=P[1...k]的最大前后缀长度**

**Aux(String P)**

1 m=P.length

2 let π[1...m] be a new array

3 π[1]=0

4 k=0

5 for q=2 to m

6 while k>0 and P[k+1]≠P[q]

7 k=π[k]

8 if P[k+1]==P[q]

9 k++

10 π[q]=k

11 return π

for循环：每次迭代开始时，满足如下图形：即k代表π[q-1]



# 31、求数组的下一字典序（若当前为最大，则返回最小）

**nextPermutation(int[] nums)**

1 L=nums.length

2 for i=L-1 to 1//高位

3 for j=L to i+1//低位

4 if nums[j]>nums[i]//第一个比高位数值大的低位

5 change(nums,i,j)

6 QuickSort(nums,i+1,L)

7 return

8 QuickSort(nums,1,L)

Line 6：//因为当i与j交换后，需要将i之后的数字变为字典序才能使得交换后是最小值

# 32、最长有效括号的长度

**longestValidParentheses(String s)**

1 n=s.length

2 let stk be a stack stored int

3 let data[1...n] be a new array initialized to zero

4 **for** i=1 **to** n

5 **if** s[i]==’(‘ stk.push(i)

6 **else\_if** !stk.empty() //若该右括号有与之匹配的左括号，将这两处标记为1

7 data[i]=1

8 data[stk.pop()]=1

9 int res=0,tep=0

10 **for** i=1 **to** n //最大长度额有效括号，必定是被连续标记的区域

11  **if** data[i]==1 tep++

12 **else** res=max(tep,res)//间断后，计算最大值，将tep置零，以便计算新区域的长度

13 tep=0

14 **return** max(tep,res)//最后一块区域可能全部被标记，需要再次比较一下

**动态规划：最长有效括号区域必然是以“（”起始“）”结尾**

**子问题设计：以s[i]结尾（包含s[i]的最大有效括号长度）。b[i]保存以s[i]结尾的最大有小括号长度，显然，若s[i]为“（”，那么b[i]=0**

**longestValidParentheses(String s)**

1 n=s.length,Maximum=0

2 let b[1...s] be new array initialized to zero

3 **for** right=2 **to** n//因为s[1]不可能为有效括号区域的右端点

4 **if** s[right]==”)”

5 left=right-1-b[right-1]//以s[right-1]为右端点的区域的左端点再的左边一个

6 **if** left≥1 **and** s[left]==”(“

7 b[right]=b[right-1]+2

8 **if** left-1≥2//当line成立后，会将之前由于s[left]间断的部分连接起来

9 b[right]=b[right]+b[left-1]

10 Maximum=max(Maximum,b[right])

11 **return** Maximum



# 33、search(int[] nums, int target) 略

# 34、在O(lgn)的时间内在有序表中查找给定目标所在的范围，查找失败则返回[-1,-1]

searchRange(int[] nums, int target)

1 left=AuxLeft(nums,1,nums.length,target)

2 **if** left==-1 **return** [-1,-1]

3 right=AuxRight(nums,left,nums.length,target)

4 **return** [left,right]

**AuxLeft(int[]nums,int left,int right, int target)**

1 **if** left<right

2 mid=⌊ (left+right)/2 ⌋

3 **if** nums[mid]==target

4 **if** mid==1 **or** muns[mid-1]<target **return** mid

5 **else** **return** AuxLeft(nums,left,mid-1,target)

6 **elseif** nums[mid]<target **return** AuxLeft(nums,mid+1,right,target)

7 **else** **return** AuxLeft(nums,left,mid-1,target)

8 **elseif** left==right **and** nums[left]==target **return** left

9 **else** **return** -1//查找不到或者left>right

**AuxRight(int[]nums,int left,int right, int target)**

1 **if** left<right

2 mid=⌊ (left+right)/2 ⌋

3 **if** nums[mid]==target

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

5 **else return** AuxRight(nums,mid+1,right,target)

6 **elseif** nums[mid]<target **return** AuxRight(nums,mid+1,right,target)

7  **else return** AuxRight(nums,left,mid-1,target)

8 **elseif** left==right a**nd** nums[left]==target **return** left

9 **else return** -1

# 35、在有序表中找到指定元素的位置或者该元素应该插入的位置，同34，用二分法

**注意 当 left>right 时，返回left : [0,-1]说明在开始处插入 [length+1,length]说明在尾部插入**

# 36、合法的数独

**合法的定义：每个元素所在的行，列，九宫格都满足唯一性**

**boolean isValidSudoku(char[][] board)**

1 let ArySet[1...27] be a new array stored HashSet

2 **for** row=1 **to** 9

3 **for** col=1 **to** 9

4 tem=board[row][col]

5 **if** tem≠’.’

6 **if** ArySet[row].contains(tem) **return** false

7 **else** ArySet[row].add(tem)

8 **if** ArySet[9+col].contains(tem) **return** false

9 **else** ArySet[9+col].add(tem)

10 **if** ArySet[18+⌊(row-1)/3⌋\*3+⌊(col-1)/3⌋+1].contains(tem) **return** false

11 **else** ArySet[18+⌊(row-1)/3⌋\*3+⌊(col-1)/3⌋+1].add(tem)

12 **return** true

若索引从0开始，红色部分改为：**18+⌊row/3⌋\*3+⌊col/3⌋**

# 37、数独Backtracking（回溯法）

**solveSudoku(char[][] board)**

1 solveSudoku(board)

**boolean solveSudokuAux(char[][] board)**

1 **for** row=1 **to** 9

2 **for** col=1 **to** 9

3 **if** board[row][col]=\0

4 **for** num=1 **to** 9

5 char[row][col]=num

6 **if** IsValid(board,row,col) **and** solveSudokuAux(board)

7 **return** true;

8 **else** char[row][col]=\0

9 **else return** false;

**boolean IsValid(char[][] board ,int row,int col)**

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

2 **if** i==row **continue**

3 **if** board[i][col]==board[row][col] **return** false

4 **for** j=1 **to** 9

5  **if** j==col **continue**

6 **if** board[row][j]==board[row][col] **return** false

7 **for** i=⌊(row-1)/3⌋\*3+1 **to** ⌊(row-1)/3+1⌋\*3

8  **for** j=⌊(col-1)/3⌋\*3+1 **to** ⌊(col-1)/3+1⌋\*3

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

10  **if** board[i][j]=board[row][col] **return** false

11 **return** true

# 38、Count And Say

“1” “11” “21” “1211” “111221” “312211”...

“1” : 1个1---> ”11”

“11” : 2个1---> ”21”

“21” : 1个2，1个1---> “1211” ......

**String countAndSay(int n)//返回第n个**

1 if (n==0) return “”

2 pre=”1”,cur=””

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

4 cur=Say(pre)

5 pre=cur

6 **return** pre

**String Say(String pre)**

1 i=1

2 len=pre.length

3 let sb be a new StringBuilder

4 **while** i≤len

5 count=1

6 **while** i+1≤len **and** pre[i]==pre[i+1]

7 i++

8 count++

9 sb.append(count)

10 sb.append(pre[i])

11 i++

12 **return** sb.toString()

# 39、重复子集和问题

# 40、不重复子集和问题

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

1 sort(canditates)

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

3 let Cur be a new List<Integer>

4 Aux(Res, Cur,candidates,1,target)

5 return Res

递归思路：回溯法，candidates中第left个位置选择取或者不取

**Aux(List<List<Integer>>Res,List<Integer> Pre,int[] candidates,int left,int target)**

1 **if** left>candidates.length **return**

2 **if** candidates[left]==target

3 let Cur be a new List equals to Pre

4 Cur.add(target)

5 Res.add(Cur)

6 return

7 **if** canditates[left]<target

8 Aux(Res,Pre,candidates,left+1,target)

9 Pre.add(candidates[left])

10 Aux (Res, Pre,candidates,left,target-candidates[left])//可以重复

11 //Aux(Res, Cur2,candidates,left+1,target-candidates[left])//不能重复

12 Pre.remove(Pre.size())

**为什么Line9、Line10 两句不能同时存在：Line8和Line10 可以包含情况Line11，再加上Line11 会导致结果重复（先Line10递归调用Aux再调用Line8，就等价于直接调用Line11）**

**动态规划（仅仅求个数）：C[i][j]保存前i个和为j的个数**

**SubSumNum(int[] candidates, int target)**

1 n=canditates.length

2 let C[1...n][0...target] be a new array

3 k=0

4 **while** candidates[1]\*k≤target//初始化,若元素不可重复用，再加个条件k<2

5 C[1][ candidates[1]\*k]=1

6 k++

7 **for** i=2 **to** n

8 **for** j=0 **to** target//j为0意味着前i-1项之和为0

9 **if** C[i-1][j]>0//若前i-1项存在和j

10 k=0

11 while candidates[0]\*k≤target-j//若元素不可重复用，再加个条件k<2

12 **C[i][j+ candidates[i]\*k]+= C[i-1][j]**

13 k++

14 **return** C[n][target]

C[i][j] 前i项的组合（可以重复），之和为j，

另一种递归思路：在剩余可选范围内，依次选择每个元素（若满足条件），放入子集的下一个位置上，该回溯循环所添加的元素都位于子集的同一个位置，可以跳过相同的值

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

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

3 let Cur be a new List<Integer>

4 Aux(Res, Cur,candidates,1,target)

5 return Res

**Aux(List<List<Integer>>Res,List<Integer> Pre,int[] candidates,int Mostleft,int target)**

1 for left=Mostleft to candidates.length

2 if left>Mostleft and candidates[left]==candidates[left-1] continue

3 elseif candidates[left]==target

4 let Cur be a new List equals to Pre

5 Cur.add(candidates[left])

6 Res.add(Cur)

7 elseif candidates[left]<target//当前和小于目标

8 Pre.add(candidates[left])

9 Aux(Res,Pre,candidates,left,target- candidates[left]) //可以重复

//Aux(Res,Cur,candidates,left+1,target- candidates[left]) //不能重复

10 Pre.remove(Pre.size())

11 else break //当前和已经大于目标，故此组合不可能

# 41、O(n)时间内在无序数组中找到第一个遗漏的正整数

思路：利用数组索引和元素值之间的对应关系

int firstMissingPositive(int[] nums)

1 i=1

2 end=nums.length

3 while i≤end

4 if nums[i]=i

5 i++

6 elseif nums[i]>0 and nums[i]<=end and nums[nums[i]]!=nums[i]

7 exchange(nums,i,nums[i])

8 else

9 exchange(nums,i,end--)

10 if end<1 return 1

11 return nums[end]+1

类似于快速排序中的Partition，将不符合的那部分挪到后面去，并相应缩小边界

索引为i的值应该与i相等 num[i]=i

当满足该条件时，递增i

当不满足且0<num[i]≤end and nums[nums[i]!=nums[i] 时，交换索引为i与nums[i]的元素，将值为num[i]的元素放置到正确的位置上，即放置到索引为num[i]的地方

当不满足所有条件时，说明num[i]不在所需要的值得范围内，放到边界处，并缩减边界

下标从0开始的话，索引为i处的值，应该为i+1，即nums[i]=i+1

故而索引从0...end 放置的值为1...end+1

nums[i]应该放置到索引为nums[i]-1的地方，即nums[nums[i]-1]=nums[i]

42、Trapping Rain Water

int trap(int[] height)

1 if height==null or height.lenght==0 return 0

2 leftMax=0,rightMax=0,trapped=0,left=0,right=height.length

3 while left<right

4 if leftMax<height[left] leftMax=height[left]

5 if rightMax<height[right] rightMax=height[right]

6 if leftMax<rightMax

7 trapped+=leftMax-height[left]

8 left++

9 else trapped+=rightMax-height[right]

10 trapped+=rightMax-height[right]

11 right—

12 return trapped

与11题最大灌水量相似，都保存一个左右最大值，并且较小的一边向中间移动

43、乘法（可以突破位数的限制）

44、正则表达式： ?可以匹配任意单个字符 \*可以匹配任意长度的任意字符

与第10题相似，可以采用相似的递归结构，但是效率较低

与第10题相似，可以采用相似的DP算法

isMatch(String s, String p)

1 let match[1...s.length+1][1...p.length+1] be a new array

2 match[s.length+1][p.length+1]=true

3 i=s.length

4 while i>0 and p[i]==’\*’

5 match[s.length+1][i--]=true

6 for i=s.length to 1

7 for j=p.length to 1

8 if s[i]==p[j] or p[j]==’?’

9 match[i][j]=match[i+1][j+1]

10 elseif p[j]==’\*’

11 match[i][j]=match[i+1][j] or match[i][j+1]

12 else match[i][j]=false

13 return match[1][1]

Line 11会产生两种情况



**①\*s[i]与p[j]匹配，match[i][j]=match[i+1][j]&&(s[i] matchs p[j])**

**②\* s[i]不与p[j]匹配，match[i][j]=match[i][j+1]&&(none matches p[j])**

isMatch(String s, String p)

1 let match[0...s.length][0...p.length] be a new array

2 match[0][0]=true

3 i=1

4 while i<=p.length and p[i]==’\*’

5 match[0][i++]=true

6 for i=1 to s.length

7 for j=1 to p.length

8 if s[i]==p[j] or p[j]==’?’

9 match[i][j]=match[i-1][j-1]

10 elseif p[j]==’\*’

11 match[i][j]=match[i-1][j] or match[i][j-1]

12 else match[i][j]=false

13 return match[s.length][p.length]

# **45、从起始位置调到终点最少跳跃次数，每个位置上的值代表在该位置时最大跳跃距离**

# **55、能否跳到终点**

贪心算法

int jump(int[] nums)

1 jumps=0,curEnd=1,curFarthest=1

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

3 curFarthest=max(curFarthest,i+nums[i])

4 **if** i==curEnd

5 jumps++

6 **if** curEnd==curFarthest **and** i==nums.length **return** false;//55题

7 curEnd=curFarthest

8 **return** jumps

//很奇怪，curEnd并不是跳跃点



**curEnd: jumps次跳跃所能覆盖的最大范围**

**curFarthest:jumps+1次跳跃所能覆盖的最大范围[]**

**jumps=1时,所能到达的区域是（curEnd[0]---curEnd[1] ]**

**jumps=i 时,所能到达的区域是（curEnd[i-1]---curEnd[i] ]，为什么左边是curEnd[i-1]，如果到达的区域在curEnd[i-1]及其左边，那么最多需要i-1跳即可**

**因此第i跳的起跳点会在（curEnd[i-2]---curEnd[i-1] ] 落点在 （curEnd[i-1]---curEnd[i] ]**

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

1 jumps=0

2 curFarthest=1+nums[0]

3 curEnd=0

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

5 **if** curEnd<i

6 jumps++

7 curEnd=curFarthest

8 curFarthest=max(curFarthest,i+nums[i])

9 **return** jumps

这种写法与前一种写法的区别在于更新跳跃次数的位置：

前一种：当curEnd==i的时候就更新jumpTimes，**因此当curEnd恰好等于终点的时候，需要特殊处理一下，采用的方法是只遍历到倒数第二个**

这一种：当curEnd<i的时候更新jumpTimes，**但是curFarthest需要滞后，因为更新curEnd的时候需要用到i-1位置及其之前的最远距离，不能把位置i的也包括进去**

# 46、47、所有可能排列方式： 回溯法：backtrack

**思路1：选择剩余的数组中的第j个放在后一位**

**List<List<Integer>> permute(int[] nums)**

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

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

3 let Used[1...nums.length] be a new array stored boolean//标记使用情况，不必创建新数组

4 let Cur be a new List stored Integer

5 sort(nums)//服务于Aux中的Line7

6 Aux(nums,Used,Cur,Res)

7 **return** Res

**Aux(int[] nums, boolean[] Used, List<Integer> Pre, List<List<Integer>> Res)**

1 **if** Pre.size()==nums.length

2 Cur=Pre

3 Res.add(Cur)

4  **return**

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

6 **if** Used[i] continue

7 **if** i>1 **and** nums[i-1]==nums[i] **and** Used[i-1]==false **continue**

8 Used[i]=true

9 Pre.add(nums[i])//考虑这里为什么不用新建一个Cur

10 Aux(nums,Used,Pre,Res);

11 Used[i]=false

12 Pre.remove(Pre.size())**//这里的复杂度是O(1) 相比于思路2中的remove高效很多**

**思路2：选择下一位数字放在Pre第j位**

**List<List<Integer>> permute(int[] nums)**

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

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

3 Aux(nums,1,Res,Cur)

4 return Res

**Aux(int[] nums,int i,List<List<Integer>> Res,List<Integer>Pre)**

1 **if** i==nums.length

2 Cur=Pre

3 Res.add(Cur)

4 **return**

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

7 Pre.add(j,nums[i]) //考虑这里为什么不用新建一个Cur

8 Aux(nums,i+1,Res,Cur)

9 Pre.remove(j)//这里复杂度较高

# 48、方阵顺时针旋转90度

外层循环:



内层循环



# 49、字符串分类，具有相同字符以及数量的字符串视为同类

**List<List<String>> groupAnagrams(String[] strs)**

1 HashMap<String, List<String>> map

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

3 ar=strs[i]

4 sort(ar)**//将构成该字符串的所有字符排序后，变为该类的一个唯一标识**

5 list=map.get(ar)

6 **if** list=null let list be a new List<String>

7 list.add(strs[i])

8 map.put(ar,list)

9 List<List<String>> Res

10 **for** L:map.values()

11 sort(L)

12 Res.add(L)

13 **return** Res

# 50、求指数运算

**double myPow(double x, int n)**

1 **if** n==0 **return** 1

2 **elseif** n==1 **return** x

3 **elseif** n==-1 **return** 1/x

4 **else**

5 num=myPow(x,n/2)

6 **return** num\*num\*myPow(x,mod(n,2))

# 51、52 女王问题

**List<List<String>> solveNQueens(int n)**

1 let Res be a new List stored all the possible board

2 let board[1...n][1...n] be a new array stored boolean//true代表该位置放置queen

3 let Col[1...n],Dig1[1...2n-1],Dig2[1...2n-1] be new arrays stored boolean

4 Aux(board,n,1, Col,Dig1,Dig2,Res)

5 **return** Res

**Aux(board,n,row,Row,Col,Dig1,Dig2,Res)**//尝试在第dex行放置一个皇后

1 **if** row>board.length

2 Res.add(board)

3 **return**

4 **for** col=1 **to** board.length//尝试dex行的每一列，否是可以放置皇后

5 dig1=row+col,dig2=col-row+ board.length//计算出改点对应的两条对角线的索引

6 **if** Col[col] **or** Dig1[dig1] **or** Dig2[dig2] **continue**

7 Col[col]= Dig1[dig1]= Dig2[dig2]=true

8 board[row][col]=true

9 Aux(board,n,row+1, Col,Dig1,Dig2,Res)

10 Col[col]= Dig1[dig1]= Dig2[dig2]=false//回到原来的状态

11 board[row][col]=false//回到原来的状态

**基于board的check复杂度较高，必须检测位于行列以及两条对角线上的所有元素**

**利用Col Dig1 Dig2就能够很大程度降低检测所消耗的时间**

# 53、最大子数组

**思路一：归并算法，算法导论**

注意点：分为三种情况 [left,mid] [left,mid,right] [mid+1,right]

其中情况[left,mid,right]:right>left，且必然包含mid和mid+1

**思路二：动态规划**

**子问题设计：M[i]表示子数组nums[1...i]中以nums[i]元素结尾的子数组的最大值**

**int maxSubArray(int[] nums)**

1 n=nums.length

2 let M[1...n] be a new array initialized to zero

3 M[1]=nums[1]

4 Max=M[1]

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

6 **if** M[i-1]<0 M[i]=nums[i]//若上一个最大自数组小于0，那么舍弃这部分

7 **else** M[i]=M[i-1]+nums[i]

8 Max=max(Max,M[i])

9 **return** Max

# 54、螺旋读取数组

**List<Integer> spiralOrder(int[][] matrix)**

1 let Res be a new list

2 n=matrix.length

3 **if** n==0 return Res

4 m=matrix[1].length

5 **if** m==0 return Res

6 left=1,bottom=n,right=m,top=1

7 type=0

8 **while** left<=right **and** top<=bottom

9 **if** type==0

10 **for** i=left **to** right

11 Res.add(matrix[top][i])

12 top++

13 **elseif** type==1

14 **for** i=top **to** bottom

15 Res.add(matrix[i][right])

16 right--;

17 **elseif** type==2

18 **for** i=right **to** left

19 Res.add(matrix[bottom][i])

20 bottom—

21 **else**

22 **for** i=bottom **to** top

23 Res.add(matrix[i][left])

24 left++

25 type=(type+1)%4

26 **return** Res

# 59、螺旋填充正整数，同理

# 56、合并区间

**merge(List<Interval> intervals)**

1 **if** intervals.isEmpty() **return i**ntervals

2 **sort intervals according to the start of the interval**

3 let Res be a new list

4 Pre=intervals[1]

5 **for** i=2 **to** intervals.size

6 **if** Pre.end<Cur.start//此时无交叠

7 Res.add(Pre)

8 Pre=Cur

9 **else**

10 Pre.end=max(Cur.end,Pre.end) //由于Cur.start>Pre.start

11 Res.add(Pre)

12 **return** Res

# 57、在已经排好序的区间中插入新区间

**方法1：**

**insert(List<Interval> intervals, Interval newInterval)**

1 insert newInterval to intervals

2 call merge(intervals)

**方法2：**

**insert(List<Interval> intervals, Interval newInterval)**

1 let Res be a new List

2 cnt=1

3 **while** cnt<=intervals.size

4 Cur=intervals[cnt]

5 **if** newInterval.end<Cur.start //无交叠，且与后续也无交叠

6 break

7 **elseif** newInterval.start>Cur.end //无交叠，可能与后续有交叠

8 Res.add(Cur)

9 cnt++

10 **else**

11 newInterval.start=min(newInterval.start,Cur.start)

12 newInterval.end=max(newInterval.end,Cur.end)

13 cnt++

14 Res.add(newInterval)

14 **for** i=cnt **to** intervals.size

15 Res.add(intervals[i])

16 **return** Res

# 58、找出最后一个单词，太简单了，略

# 60、给定正整数1…n，在所有n！种排列方式中找到第k个

**backtrack可以用于求解所有的组合，但是用于求第k个台浪费时间**

**getPermutation(int n, int k)**

1 k=k-1//由于取模后从0开始计算

2 let Used be a new array stored Boolean

3 let sb be a new StringBuilder

4 m=n

5 **while**(m>0)

6 dex=k/((m-1)!)

7 k=k mod (m-1)!

7 sb.append(find(Used,dex))

8 m--

9 **return** sb

**find(Boolean[] Used,int dex)**

1 cnt=0

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

3 **if** Used[i]==flase

4 **if** dex==cnt

5 Used[i]=true

6 return i

7 **else** cnt++

8 **return** -1

# 61、将一个单链表向左/右循环移位k位

**ListNode rotateRight(ListNode head, int k)**

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

2 Cur=head

3 len=1

4 **while** **Cur.next!=null**//无需用Pre来保留前一个合法元素，改变条件即可

5 Cur=Cur.next

6 len++

7 **Cur.next=head**//将单链表改造为单环链表

8 dex=len-k%len //若循环左移则为dex=k%len

9 Cur=head

10 Pre=null

11 **for** i=1 **to** dex

12 Pre=Cur

13 Cur=Cur.next

14 head=Cur

15 Pre.next=null

16 **return** head

# 62、简单路径总数

**思路：动态规划**

**uniquePaths(int m, int n)**

1 let M[1…m][1…n] be a new array

2 **for** i=1 **to** m

3 **for** j=1 **to** n

4 **if** i==0 **or** j==0 M[i][j]=0

5 **else** M[i][j]=M[i-1][j]+M[i][j-1]

6 **return** M[m][n]

# 63、有障碍的简单路径总数

**int uniquePathsWithObstacles(int[][] obstacleGrid)**

1 **if** obstacleGrid==null **return** 0

2 m=obstacle.length

3 **if** m==0 **return** 0

4 n=obstacle[0].length

5 **if** n==0 **return** 0

6 let M[1…m][1…n] be a new array

7 **for** i=1 **to** m

8 **for** j=1 **to** n

9  **if** obstalceGrid[i][j]==1 M[i][j]=0

10 **else if** i==1 **and** j==1 M[i][j]=1

11 **else if** i==1 M[i][j]=M[i][j-1]

12 **else if** j==1 M[i][j]=M[i-1][j]

13 **else** M[i][j]=M[i-1][j]+M[i][j-1]

14 **return** M[m][n]

# 64、到终点的最小总和

**int minPathSum(int[][] grid)**

1 **if** grid=null **return** 0

2 m=grid.length

3 **if** m==0 **return** 0

4 n=grid[0].length

5 **if** n==0 **return** 0

6 let M[1…m][1…n]

7 **for** i=1 **to** m

8 **for** j=1 **to** n

9 **if** i==1 **and** j==1 M[i][j]=grid[i]

10 **else if** i==1 M[i][j]=M[i][j-1]+grid[i][j]

11 **else if** j==1 M[i][j]=M[i-1][j]+grid[i][j]

12 **else** M[i][j]=min(M[i-1][j],M[i][j-1])+grid[i][j]

13 **return** M[m][n]

# 65、合法的数字

000 1.1 1. .1 1e2 +1e-2 -2e0 视为合法数字

1e e2 视为非法数字

**isNumber(String s)**

1 n=s.length

2 **if** n==0 return false

3 i=1

4 count\_num=0,count\_point=0

5 **while** getChar(s,i)==’ ’ i++

6 **if** getChar(s,i)==’+’ or getChar(s,i)==’-’ i++

7 **while** isdigit(getChar(s,i)) **or** getChar(s,i)==’.’

8 **if** getChar(s,i++) count\_point++

9 **else** count\_num++

10 **if** count\_point>1 **or** count\_num<1 **return** false//若点超过一个，或数字少于1个，则不合法

若有点，点与数字的组合中，点可以出现在该组合的任意位置

若下面有e那么e之前必须含有数字，若下面没有e那么至少有一个数字才合法

11 **if** getChar(s,i)==’e’

12 i++

13 count\_num=0,count\_point=0

14 **if** getChar(s,i)==’+’ or getChar(s,i)==’-’ i++

15 **while** isdigit(getChar(s,i)) **or** getChar(s,i)==’.’

16 **if** getChar(s,i++) count\_point++

17  **else** count\_num++

18 **if** count\_point>0 **or** count\_num<1 **return** false//e之后不能有点，但必须有数字

19 **while** getChar(s,i)==’ ’ i++

20 **return** i==n+1

**isdigit(char c)**

1 **if** c>=’0’ and c<=’9’ **return** true

2 **return** false

**getChar(String s,int i)**

1 if i>=1 **and** i<=s.length **return** s[i]

2 **return** ‘\0’

# 66、67相加

**处理好进位即可,这里N代表进制数**

1 carry=0

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

3 cur=a[i]+b[i]+carry

4 carry=cur/N

5 cur=mod(cur,N)

6 c[i]=cur

# 68、文本排布：保持顺序的前提下，尽可能多得将元素排在一行，每行中首元素紧靠左边，若有尾元素，紧靠右边。最后一行各单词见空一格

**List<String> fullJustify(String[] words, int maxWidth)**

1 dex=1

2 **while** dex≤words.length

3 let sb be a new StringBuilder

4 sb.append(words[dex])

5 curdex=dex+1

6 curlen=sb.length()

7 **while** curdex≤words.length **and** words[curdex].length+1≤maxWidth-curlen

8 curlen=curlen+words[curdex++].length+1

9 int space=1,extra=0

10 **if** curdex<words.length **and** curdex-dex>1 //非最后一行，且该行单词数至少2个

11 space=⌊(maxWidht-curlen)/(curdex-dex-1) ⌋+1

12 extra=(maxWidht-curlen)%(curdex-dex-1)

13 dex++//

14 **while** extra-->0

15 sb.append(space+1,’ ’)

16 sb.append(words[dex++])

17 **while** dex<curdex

18 sb.append(space,’ ’)

19 sb.append(words[dex++])

20 sb.append(maxWidth-sb.length(),’ ’)

21 Res.add(sb.toString())

22 **return** Res

# 69、开方

**public int mySqrt(int x)**

1 **if** x<4 **return** x==0? 0:1

2 res=2\*mySqrt(x/4)

3 **if** (res+1)\*(res+1)≤x&&(res+1)\*(res+1)≥0 return res+1

4 **return** res

**public int mySqrt(int x)**

1 r=x

2 **while**(r\*r>x)

3 r=(r+x/r)/2

4 **return** r

# 70、上楼梯的可能方案，可以跳1步或2两步

**DP:类似于斐波那契数列**

**int climbStairs(int n)**

1 let M[1…n] be a new array

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

3 **if** i==1 M[i]=1

4 **elseif** i==2 M[i]=2

5 **else** M[i]=M[i-1]+M[i-2]

6 **return** M[n]

# 71、简单路径

**"/"、"/."无作用**

**"/.."退回上一个目录**

**"/somethingt" 进入指定子目录**

**public String simplifyPath(String path)**

1 strs=path.split("/")

2 let stack be a new Stack

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

4 **if** strs[i].equals("") **or** strs[i].equals(".") **continue**

5 **elseif** strs[i].equals("..")

6 **if** **not** stack.isEmpty()

7 stack.pop()

8 **else** stack.push(strs[i])

9 let sb be a new StringBuilder

10 **while** **not** stack.isEmpty()

11 sb.append(new StringBuilder(stack.pop()).reverse()+"/")

12 **if** sb.length()==0 sb.append('/')

13 **return** sb.reverse().toString()

# 72、两字符串间最短距离

**int minDistance(String word1, String word2)**

1 m=word1.length

2 n=word2.length

3 **if** n==0 return m

4 **if** m==0 return n

5 let M[0…m][0…n] be a new array

6 **for** i=0 **to** m M[i][0]=i

7 **for** j=0 **to** n M[0][j]=j

8 **for** i=1 **to** m

9 **for** j=1 **to** n

10 M[i][j]=min( M[i-1][j-1]+(word1[i]==word[j]?0:1),M[i-1][j]+1,M[i][j-1]+1)

11 **return** M[m][n]

[1…i-1] i

[1…j-1] j

若word1[i]==word2[j]

**①那么可以选择直接匹配（不用替换）**，则M[i][j]=M[i-1][j-1]

**②选择插入一个**

若word1序列插入，使得word2[j]与\*匹配，那么word1[1…i]需要与word2[1…j-1]匹配

因此：M[i][j]=M[i][j-1]+1

[1…i-1] i \*

[1…j-1] j

若word2序列插入，使得word1[i]与\*匹配，那么word1[1…i-1]需要与word2[1…j]匹配

因此：M[i][j]=M[i-1][j]+1

[1…i-1] i

[1…j-1] j \*

**③若选择删除一个**

若删除word1[i]，那么word1[1…i-1]需要与word2[1…j]匹配

因此：M[i][j]=M[i-1][j]+1

[1…i-1]

[1…j-1] j

若删除word2[j]，那么word1[1…i]需要与word2[1…j-1]匹配

因此：M[i][j]=M[i][j-1]+1

[1…i-1] i

[1…j-1]

若word1[i]≠word2[j]

**①若替换一个**，无论替换word[i]还是word[j]

M[i][j]=M[i-1][j-1]+1

②③同理

综上M[i][j]=min( M[i-1][j-1]+(word1[i]==word[j]?0:1),M[i-1][j]+1,M[i][j-1]+1)

# 73、m×n的矩阵中，将所有零元素所在的行列全部置零

注意：必须标记出原来是非零元素，后来被置零的元素

# 74、一个矩阵，每一行从小到大排列，且下一行最小元素大于上一行的最大元素，查找指定元素

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

1 m=matrix.length

2 **if** m==0 return false

3 n=matrix[0].length

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

5 dex=1

6 **while** dex<m **and** target>matrix[dex][1] dex++ //找到首元素不小于k的行号

7 **if** dex ≤ m **and** target==matrix[dex][1] **return** true//若该行首元素恰等于k，否则在上一行找

8 dex--

9 **if** dex<1 **return** false

10 **for** i=1 **to** n

11 **if** target==matrix[dex][i] **return** true

12 **return** false

# 75、大量重复元素的排序，基数排序或者快速排序

# 76、O(n)时间内求长字符串中包含短字符串所有元素（顺序可不以一样）的最小窗口

**String minWindow(String s, String t)**

1 sLen=s.length

2 tLen=t.length

3 begin=1,end=1//terminal of current field

4 head=1//begin terminal of final output substring

5 len=+∞//the length of final output substring

6 count=tLen//counting elements to be matched

7 let map be a new Map<char,int>//initialized to zero,stored the times of occured element

8 **for** i=1 **to** tLen

9 map[t[i]]++;

10 **while** end<sLen

11 **if** map.contain(s[end]) **and** map[s[end]]>0

12 count--

13 map[s[end]]--

14 end++

//若s[end]对应的计数值已经为0，说明，该元素过剩了

15 **while** count==0

16 **if** end-begin<d

17 head=begin

18 len=end-begin

19 **if** map.contain(s[begin]) and map[s[begin]]++==0

20 count++

21 map[s[begin]]++

22 begin++

//去掉当前区域[being,end)中第一个匹配t重元素的字符，当该元素再次出现的时候，又生成了一个新区域

23 return d==+∞? "":s.substring[head,head+len)

# 77、求1...n中包含k个元素的所有子集，回溯法

**List<List<Integer>> combine(int n, int k)**

1 if n==0 or k==0 or n<k return null

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

3 let Pre be a new List<Integer>

4 combineAux(n,k,1,1,Pre,Res)

5 return Res

**combineAux(int n,int k,int dex,int start,List<Integer> Pre,List<List<Integer>> Res)**

1 if dex==k+1

2 let Cur be a new List equals to Pre

3 Res.add(Cur)

4 return

5 for i=start to n-(k-dex)//start represents the minimum number can be chosen during this loop,and dex represents the index of the k numbers

6 Pre.add(i)

7 combineAux(n,k,dex+1,i+1,Pre,Res)

8 Pre.remove(Pre.size())//state restore/recover

# 78、不重复数组的所有子集，与77题类似，但有另一种回溯思路

**List<List<Integer>> subsets(int[] nums)**

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

2 let pre be a new List<Integer>

3 helper(nums,1,pre,res)

4 **return** res

**该回溯思路是沿着nums进行的，对于nums[i]选择取或不取**

**private void helper(int[] nums,int dex,List<Integer> pre,List<List<Integer>> res)**

1 **if** dex==nums.length+1

2 let cur be a new List equals to pre

3 res.add(cur)

4 **return**

5 helper(nums,dex+1,pre,res)//not contain nums[start]

6 pre.add(nums[dex])

7 helper(nums,dex+1,pre,res)//contain nums[start]

8 pre.removeLast()//state recover

**该回溯思路是在剩余可选范围内，依次选择每个元素，放置到子集的下一个位置上，每个循环中，添加的元素都位于子集的相同位置**

**private void helper(int[] nums,int begin,List<Integer> pre,List<List<Integer>> res)**

***//每一种临时状态都是一种可行解***

**1 let cur be a new List equals to pre**

**2 res.add(cur)**

3 for i=begin to nums.length

4 Pre.add(nums[i])

5 helper(nums,i+1,pre,res)

6 Pre.removeLast()

# 79、搜索单词，回溯法

boolean exist(char[][] board, String word)

1 if board=null or board.length==0 or word==null or word.length==0 return false

2 m=board.length

3 n=board[0].length

4 if n==0 return false;

5 let Used[1...m][1...n] be a new array stored boolean initialized to zero

6 len=word.length

7 for row=1 to m

8 for col=1 to n

9 if existAux(board,m,n,row,col,Used,word,1,len)

10 return true

11 return false

existAux(char[][] board,int m,int n,int row,int col,boolean[][] Used,String word,int dex,int len){

1 if board[row][col]≠word[dex] **return** false//check whether current position matches word[dex]

2 **if** dex==len **return** true

3 Used[row][col]=true

4 **if** col>1 **and** !Used[row][col-1] //left

5 **if** existAux(board,m,n,row,col-1,Used,word,dex+1,len) **return** true

6 if row>1 and !Used[row-1]col]//top

7 if existAux(board,m,n,row-1,col,Used,word,dex+1,len) return true

8 if col<n and !Used[row][col+1]//right

9 if existAux(board,m,n,row,col+1,Used,word,dex+1,len) return true

10 if row<m and !Used[row+1][col]//bottom

11 if existAux(board,m,n,row+1,col,Used,word,dex+1,len) return true

12 Used[row][col]=false //state recover

13 return false

# **80、求有序数组中元素个数N（多于两个的相同元素按2个计算），且位于数组前N个位置**

int removeDuplicates(int[] nums)

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

2 let tem be a new array whose size equals to nums

3 if nums.length==1 return 1

4 begin=1,end=2

5 dex=1

6 while end<nums.length

7 while end<nums.length and nums[end]==nums[end-1] end++

8 len=end-begin

9 if len>2 len=2

10 for i=1 to len

11 tem[dex++]=nums[begin]

12 begin=end++

13 if begin==nums.length-1 tem[dex++]=nums[begin]//boundary condition

14 nums=tem

15 return dex-1

# **81、在已排序的循环数组中搜索指定元素**

boolean search(int[] nums, int target)

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

2 minimum=nums[1] //minimum must be the first element of the array

3 i=1

4 while i≤nums.length

5 **if** target==nums[i] **return** true

6 **if** i>1 **and** nums[i]<nums[i-1] **and** nums[i]==minimum **return** false

7 **return** false

# 82、83 有序链表中除去重复的元素（一个不留和留下一个两种情况）

**ListNode deleteDuplicates(ListNode head)**

1 ListNode begin=null,end=head//begin point at the last noneduplicate element

2 while end≠null

3 Pre=end

4 while end.next≠null and end.next.val==end.val

5 end=end.next

6 if Pre≠end

7 if begin==null head=end.next

8 else begin.next=end.next

9 else

10 begin=end

11 end=end.next

12 return head

ListNode deleteDuplicates(ListNode head)

1 ListNode begin=head,end=head

2 while end≠null

3 while end.next≠null **and** end.next.val==end.val

4 end=end.next

5 begin.next=end.next

6 end=end.next

7 begin=end

8 return head

# **84、面积最大的长方形**

public int largestRectangleArea(int[] heights)

1 let stack be a new Stack

2 maximum=0,left=0,pos=0,iter=1

3 while i≤heights.length

4 **while not** stack.isEmpty() **and** heights[stack.peek()]>heights[iter]

5 **pos=stack.pop()**

6 **left=stack.isEmpty()?1:stack.peek()+1**

7 maximum=max(maximum,heights[pos]\*(iter-1-left+1))

8 stack.push(iter++)

9 **while** **not** stack.isEmpty()

10 **pos=stack.pop()**

11 **left=stack.isEmpty()?1:stack.peek()+1**

12 maximum=max(maximum,heights[pos]\*(heights.length-1-left+1))

13 **return** maximum



堆栈中保留的为索引号，从栈底到栈顶，索引号对应的高度依次增高（栈顶索引对应的高度必定是栈中最高的）

每个下标(iter)都会入栈，入栈时会保证该下标所对应的高度height[iter]是栈中最大值

当heights[stack.peek()]>heights[iter]时：将栈顶元素弹出，并计算，以height[stack.peek()]为高的矩形的面积（未必是最大的！）原因如下：由于此时，height[stack.peek()]是最大的，但有可能存在这样一种情况：heights[stack.secondpeek()]==heights[stack.peek()]，即栈顶与次栈顶的下标所对应的高度相同，那么即便本次出栈得到的矩形面积不是最大，当次栈顶元素出栈时会进一步扩大矩形面积，最终maximum会与以height[stack.peek()]为高度的最大矩形面积相等。因此每次迭代只需计算次顶下标的下一个位置（即弹出一次后，取left=stack.peek()+1，[left...iter-1]范围内的高度必然是大于栈顶元素的，原因看绿色部分），特殊情况是，栈中只有一个栈顶元素（说明：在栈顶元素入栈前，之前的所有位置都已经弹出，即高度比栈顶元素对应的高度要高，此时的边界就是数组边界）

pos=stack.pop()之后：假设此时栈不为空，栈顶下标为pos\_pre=stack.peek()

[pos\_pre+1...iter-1]：该范围内的高度必然是大于或等于heights[pos]的（反证：如果不成立，那么此时栈顶元素就将会是这些小于heights[pos]的下标，而不是pos\_pre）

# **85、最大的矩形区域（该区域全部被标记为1）**

int maximalRectangle(char[][] matrix)

1 if matrix==null or matrix.length==0 return 0

2 m=matrix.length

3 n=matrix[0].length

4 if n==0 return 0

5 let H[1...m][1...n] W[1...m][1...n] S[1...m][1...n] be new arrays

6 Maximum=0

7 if matrix[1][1]=='1'

8 Matrixmum=H[1][1]=W[1][1]=S[1][1]=1

9 else H[1][1]=W[1][1]=S[1][1]=0

10 for row=1 to m

11 for col=1 to n

12 if row==1 and col==1 continue

13 if matrix[row][col]=='0'

14 H[row][col]=W[row][col]=S[row][col]=0

15 continue

16 i1=i2=row

17 j1=j2=col

18 if row>1 and col>1 and S[row-1][col-1]≠0

19 **while** i1≥row-H[row-1][col-1] **and** matrix[i1][col]=='1' i1--

20 **while** j1≥col-W[row-1][col-1] **and** matrix[row][j1]=='1' j1--

21 **while** i2≥1 **and** matrix[i2][col]=='1' i2--

22 **while** j2≥1 **and** matrix[row][j2]=='1' j2--

23 **if** (row-i1)\*(col-j1) ≥row-i2 **and** (row-i1)\*(col-j1) ≥col-j2

24 H[row][col]=row-i1

25 W[row][col]=col-j1

26 **elseif** row-i2≥(row-i1)\*(col-j1) **and** row-i2≥col-j2

27 H[row][col]=row-i2

28 W[row][col]=1

29 **else** H[row][col]=1

30 W[row][col]=col-j2

31 S[row][col]=H[row][col]\*W[row][col]

32 Maximum=max(Maximum,S[row][col])

33 **return** Maximum

以(row,col)为右下端点的矩形（该点必须是1，否则不可能），必然是A、B、C三类中的最大者，其中情形A：以(row-1,col-1)为右下端点的矩形必须存在，否则不存在情形A

思路2

**int maximalRectangle(char[][] matrix)**

1 if matrix==null or matrix.length==0 return 0

2 m=matrix.length

3 n=matrix[0].length

4 if n==0 return 0

5 let left[1...n] right[1...n] height[1...n] be new arrays

6 fill(left,0),fill(right,n),fill(height,0)

7 Maximum=0

8 **for** i=1 **to** m

9 cur\_left=0 cur\_right=n???

10 **for** j=1 **to** n

11 **if** matrix[i][j]=='1' height[j]++

12 **else** height[j]=0

13 **for** j=1 **to** n

14 **if** matrix[i][j]=='1' left[j]=max(left[j],cur\_left)

15 **else** left[j]=0,cur\_left=j+1

16 **for** j=n **downto** 1

17 **if** matrix[i][j]=='1' right[j]=min(right[j],cur\_right)

18 **else** right[j]=n,cur\_right=j

19 **for** j=1 **to** n

20 Maximum=max(Maximum,(right[j]-left[j])\*height[j])

21 **return** Maximum

思路3：根据84题的结论，对矩形进行逐行扫描，并更新高度矩阵heights

* 当出现'1'的时候，递增对应的高度
* 当出现'0'的时候，说明出现了断点，高度置0

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

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

2 m=matrix.length,n=matrix[0].length

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

4 maximum=0

5 **for** row=1 **to** m

6 **for** i=1 **to** n

7 **if** matrix[row][i]=='1' heights[i]++

8 **else** heights[i]=0

9 maximum=max(maximum,maximalRectangle(heights))

10 **return** maximum

# 86、对链表进行类似于快排的Partition

**ListNode partition(ListNode head, int x)**

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

2 Left=null,Right=null

3 Lefthead=null,Righthead=null

4 cur=head

5 **while** cur≠null

6 **if** cur.val<x

7 **if** Lefthead==null

8 Left=Lefthead=cur

9 **else** Left.next=cur

10 Left=cur

11 **else**

12 **if** Right==null

13 Right=Righthead=cur

14 **else** Right.next=cur

15 Right=cur

16 cur=cur.next

17 **if** Right≠null Right.next=null

18 **if** Lefthead==null **return** Righthead

19 Left.next=Righthead

20 **return** Lefthead

# 87、单词的Scramble

**boolean isScramble(String s1, String s2)**

1 **if** s1.equals(s2) **return** true

2 let letter[1...26] be a new array

3 **for** i=1 **to** s1.length() letter[s1[i]]++

4 **for** i=1 **to** s2.length() letter[s2[i]]--

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

6 if letter[i]≠0 return false

7 **for** i=1 **to** **s1.length()-1**

8 **if** isScramble(s1[1..i],s2[1...i]) **and** isScramble(s1[i+1...end],s2[i+1...end])

10 **return** true

11 **if** isScramble(s1[1..i],s2[end-i+1...end] **and** isScramble(s1[i+1...end],s2[1...end-i])

12 **return** true

13 **return** false



# 88、合并有序数组，太简单，略

**不加哨兵，就用三个循环**

**while i**≤m **and** j≤n

**while** i≤m

**while** j≤n

# 89、n位格雷码

**List<Integer> grayCode(int n)**

1 let Res be a new List

2 Res.add(0)

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

4 size=Res.size()

5 **for** j=size **downto** 1

6 Res.add(Res.get(j)|1<<i-1)

7 **return** Res

# 90、（包含重复）数组的所有子集

**List<List<Integer>> subsetsWithDup(int[] nums)**

1 sort(nums)

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

3 let pre be a new List<Integer>

4 helper(nums,1,pre,res)

5 **return** res

**private void helper(int[] nums,int pos,List<Integer> pre,List<List<Integer>> res)**

1 let cur be a new List<Integer> equals to pre

2 pes.add(cur)

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

5 **if** i>pos **and** nums[i]==nums[i-1] **continue**

6 pre.add(nums[i])

7 helper(nums,i+1,pre,res)

8 pre.removeLast()

**在剩余可选范围内，依次选择每个元素，放入子集的下一个位置上**

**上述回溯循环中添加的元素都位于同一位置，因此可以跳过相同的值**

# 91、所有可行非异前缀码译码方案

**numDecodings(String s)**

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

2 n=s.length

3 let M[0...n] be a new array initialized to 0

4 M[0]=1

5 **if** s.[1]=='0' **return** 0

6 M[1]=1

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

8 **if** s[i]≠'0' M[i]+=M[i-1]

9 **if** IsValid(s,i) M[i]+=M[i-2]

10 **if** M[i]==0 **return** 0

11 **return** M[n]

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

1 **if** s[i-1]=='1' **return** true

2 **if** s[i-1]=='2' and s[i]<'7' **return** true

3 **return** false



# 92、反转链表指定区间

**ListNode reverseBetween(ListNode head, int m, int n)**

1 **if** head==null **or** m==n **return** head

2 Left=null,Right=null,M=null

3 cur=head,pre=null

4 **for** i=1 **to** m-1

5 pre=cur

6 cur=cur.next

7 Right=M=cur

8 Left=pre

9 cur=cur.next

10 **for** i=0 **to** n-m-1

11 **if** Left==null head=cur

12 **else** Left.next=cur

13 M.next=cur.next

14 cur.next=Right

15 Right=cur

16 cur=M.next

17 **return** head



# 93、合法IP，32位，四个整数（0-255）

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

1 let Res be a new List<String>

2 let Pre be a new StringBuilder

3 Aux(s,1,1,Pre,Res)

4 **return** Res

**Aux(String s,int dex,int start,StringBuilder Pre,List<String> Res){**

1 **if** dex==5 **and** start==s.length

2 let Cur be a new StringBuilder equals to Pre

3 Cur.delete(1)//delete the first '.'

4 Res.add(Cur.toString())

5 **return**

6 **if** s[start]=='0' //must be a single number

7 Pre.append('.'+s[start])

8 Aux(s,dex+1,start+1,Pre,Res)

9 **return**

10 **for** len=1 to 3

11 **if** start+len-1>s.length **break**

12 int num=getNum(s,start,start+len-1)

13 **if** num>255 **break**

14 Pre.append('.'+s[start...start+len-1])

15 Aux(s,dex+1,start+len,Pre,Res)

16 Pre.delete(Pre.length-(len+1)+1,Pre.length)

**getNum(String s,int start,int end)**

1 num=0

2 **for** i=start **to** end

3 num=num\*10+(s[i]-'0')

4 **return** num

# 94、二叉树（无父亲指针）中序遍历，递归 or 栈

# 95、1...n n个数构成的所有二叉树

**List<TreeNode> generateTrees(int n)**

1 if n==0 return a new List<TreeNode> that is empty

2 **return** Aux(1,n)

**List<TreeNode> Aux(int start,int end)**

1 let Cur be a new List<TreeNode>

2 if start>end Cur.add(null)

3 **for** k=start **to** end //k represents the index of root

4 **for** L:Aux(start,k-1)

5 **for** R:Aux(k+1,end)

6 let Root be a new TreeNode with val=k

7 Root.left=L

8 Root.right=R

9 Cur.add(Root)

10 **return** cur

**动态规划**：M[i][j]存储i...j为元素构成的二叉树的所有根节点

**List<TreeNode> generateTrees(int n)**

1 **if** n==0 return a new List<TreeNode> that is empty

2 let M[0...n+1][0...n+1] be a new array

3 let None be a new List<TreeNode>

4 None.add(null)

5 **for** i=1 **to** n

6 let Tem be a new List<TreeNode>

7 let Root be a new TreeNode with val=i

8 Tem.add(Root)

9 M[i][i]=Tem

10 M[i][i-1]=a new List<TreeNode> equals to None //prepare for Line18

11 M[i+1][i]=a new List<TreeNode> equals to None //prepare for Line17

12 **for** L=2 **to** n

13 **for** start=1 **to** n-L+1

14 end=start+L-1

15 let Tem be a new List<TreeNode>

16 **for** k=start **to** end// index of root

17 **for** R:M[k+1][end]

18 **for** L:M[start][k-1]

19 let Root be a new TreeNode with val=k

20 Root.left=L

21 Root.right=R

22 Tem.add(Root)

23 M[strat][end]=Tem

24 **return** M[1][n]

# 96、求1...n n个元素构成的所有不同搜索二叉树的数量

**public int numTrees(int n)**

1 **if** n==0 return n

2 let M[0...n] be a new array initialized to zero

3 M[0]=1

4 **for** L=1 **to** n

5 **for** k=0 **to** L-1//number of leftSubTree's Node

6 M[L]=M[L]+M[k]\*M[L-1-k]

7 **return** M[n]

# 97、检查s3是否为s1和s2的交织

**boolean isInterleave(String s1, String s2, String s3)**

1 **if** s1==null **or** s2==null **or** s3==null **throw** RuntimeException

2 **if** s1.length+s2.length!=s3.length **return** false

3 let M[0...s1.length][0...s2.length] be a new array stored boolean and initialized to false

4 M[0][0]=true

5 **for** len1=0 **to** s1.length

6 **for** len2=0 **to** s2.length

7 **if** len1==len2==0 **continue**

8 **elseif** len1==0

9 M[len1][len2]=M[len1][len2-1] and s2[len2]==s3[len1+len2]

10 **elseif** len2==0

11 M[len1][len2]=M[len1-1][len2] and s1[len1]==s3[len1+len2]

12 **else** M[len1][len2]= M[len1][len2-1] and s2[len2]==s3[len1+len2]

or M[len1-1][len2] and s1[len1]==s3[len1+len2]

13 **return** M[s1.length][s2.length]

# 98、检查搜索二叉树是否合法（是否满足值域区间）

**boolean isValidBST(TreeNode root)**

1 **return** Aux(root,Long.MIN\_VLAUE,Long.MAX\_VLAUE)

**boolean Aux(TreeNode x,long left,long right)**

1 **if** x≠null

2 **if** x.val≤left and x.val≥right **return** false

3 **return** Aux(x.left,left,x.val) **and** Aux(x.right,x.val,right)

4 **return** true

**public boolean isValidBST(TreeNode root)**

1 let stack be a new Stack stored TreeNode

2 let maxStack be a new Stack stored Long

3 cur=root

4 minimum=-∞

5 maxStack.push(+∞)

6 **while** cur≠null **or** **not** stack.isEmpty()

7 **while** cur≠null

8 **if** cur.val≥maxStack.peek() or cur.val≤minimum **return** false

9 stack.push(cur)

10 maxStack.push(cur.val)

11 cur=cur.left

12 **if** **not** stack.isEmpty()

13 peek=stack.pop()

14 minimum=peek.val

15 maxStack.pop()

16 cur=peek.right

17 **return** true

**最大边界的维护需要依靠栈，因为之前的值都是有效的，而最小边界的维护只与弹出元素的值有关**

# 99、恢复搜索二叉树（有两个元素被交换了位置）

**void recoverTree(TreeNode root)**

1 InorderTraversal(root)

2 tem=first.val

3 first.val=second.val

4 second.val=tem

**void InorderTraversal(TreeNode cur)**

1 **if** cur≠null

2 InorderTraversal(cur.left)

3 **if** Pre≠null **and** pre.val>=cur.val

4 first=pre

5 **if** first≠null **and** pre.val>cur.val

6 second=cur

7 pre=cur

8 InorderTraversal(x.right)



# 100、比较两棵搜索二叉树是否相同

**boolean isSameTree(TreeNode p, TreeNode q)**

1 **if** p==null **and** q==null **return** true

2 **if** p==null **or** q==null **return** false

3 **if** p.val≠q.val **return** false

4 **return** isSameTree(p.left,q.left) **and** isSameTree(p.right,q.right)