1. Two sum

把数组本身数字和下标分别作为key和value放进map里，时间复杂度o(n)

1. **public** **class** Solution {
2. **public** **int**[] twoSum(**int**[] nums, **int** target) {
3. **int**[] result= **new** **int**[2];
4. Map<Integer,Integer> difference = **new** HashMap<Integer,Integer>();
5. **for** (**int** i=0;i<nums.length;i++) {
6. **if** (difference.containsKey(target-nums[i])){
7. result[0] = difference.get(target-nums[i]);
8. result[1]=i;
9. **break**;
10. } **else** {
11. difference.put(nums[i],i);
12. }
13. **throw** **new** IllegalArgumentException("No two sum solution");
14. }
15. **return** result;
16. }
17. }

7. Reverse Integer

注意结果的溢出问题

1. 先使结果为long型再转换成int

long result；

if (result>Integer.MAX\_VAULE||result <Integer.MIN\_VALUE) return 0;

2. 每次计算结果后，判断结果除以10后是否和原来没乘10时结果相等，若溢出，关系改变已不再是10倍关系：

Int result =y;

  y = 10\*y+a;

x = x / 10;

if (y/10!=result)  return 0;

1. **public** **class** Solution {
2. **public** **int** reverse(**int** x) {
3. **long** result = 0;
4. **while** (x!=0) {
5. result = 10\*result+x%10;
6. x /= 10;
7. }
8. **if** (result>Integer.MAX\_VALUE||result<Integer.MIN\_VALUE) **return** 0;
9. **return** (**int**)result;
10. }
11. }

9. Palindrome Number

time法：

判断回文，前后依次取一位比较，time用来表示推进

1. **public** **class** Solution {
2. **public** **boolean** isPalindrome(**int** x) {
3. **int** bit = 0;
4. **int** time = 0;
5. **if** (x<0) {
6. **return** **false**;
7. }
8. **int** y = x;
9. **while**(y>0) {
10. bit ++;
11. y/=10;
12. }
13. **int** l = bit;
14. **while** (time<(l/2)) {
16. **int** i = (**int**)Math.pow(10,time);
17. **int** j=(**int**)Math.pow(10,bit-time-1);
18. **if** ((x/j)%10 != (x/i)%10) {
19. **return** **false**;
20. }
21. time ++;
22. }
23. **return** **true**;
24. }
25. }

13. Roman to Integer

1. **public** **class** Solution {
2. **public** **int** romanToInt(String s) {
3. **int** result = 0;
4. **char**[] roman = {'I','V','X','L','C','D','M'};
5. **int**[] integer ={1,5,10,50,100,500,1000};
6. Map<Character,Integer> map = **new** HashMap<Character,Integer>();
7. **for** (**int** i=0;i<roman.length;i++) {
8. map.put(roman[i],integer[i]);
9. }
10. **for** (**int** i=0;i<s.length()-1;i++) {
11. **int** a = map.get(s.charAt(i));
12. **int** b = map.get(s.charAt(i+1));
13. **if** (a<b) {
14. result = result -a;
15. } **else** {
16. result = result + a;
17. }
19. }
20. result += map.get(s.charAt(s.length()-1));
21. **return** result;
22. }
23. }

14. Longest Common Prefix

验证第一个字符串从0到leng－1的字符是不是其他字符串都有，String有startsWith()方法（是否以某个字符串开始), 不要使用StringBuilder复制, 指向同一字符串，若有不同操作使用string＋

1. **public** **class** Solution {
2. **public** String longestCommonPrefix(String[] strs) {
3. **if** (strs == **null**||strs.length==0) {
4. **return** "";
5. }
6. StringBuilder result = **new** StringBuilder("");
7. **int** len = strs[0].length();
8. **for** (**int** i=0;i<len;i++) {
9. StringBuilder curr = result;
10. result.append(strs[0].charAt(i));
11. **for** (**int** j=1;j<strs.length;j++) {
12. **if** (!strs[j].startsWith(result.toString())) {
13. **return** curr.toString();
14. }
15. }
17. }
18. **return** result.toString();
19. }
20. }

20. Valid Parentheses

1. **public** **class** Solution {
2. **public** **boolean** isValid(String s) {
3. Stack<Character> stack = **new** Stack<Character>();
4. **int** len = s.length();
5. **for** (**int** i=0;i<len;i++) {
6. **char** curr = s.charAt(i);
7. **if** (curr == '(' || curr =='['||curr=='{' ) {
8. stack.push(curr);
9. } **else** **if**(stack.isEmpty()) {
10. **return** **false**;
11. } **else** **if** (curr==')') {
12. **if** (stack.pop()!= '(') {
13. **return** **false**;
14. }
15. } **else** **if** (curr==']') {
16. **if** (stack.pop()!= '['){
17. **return** **false**;
18. }
19. } **else** **if** (curr=='}') {
20. **if** (stack.pop()!= '{') {
21. **return** **false**;
22. }
23. }
25. }
26. **if** (stack.isEmpty()) {
27. **return** **true**;
28. } **else** **return** **false**;
29. }
30. }
31. Merge Two Sorted Lists

定义两个指针，一个head永远指头且返回该指针，另外一个end根据两个list一直指到最后

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** ListNode mergeTwoLists(ListNode l1, ListNode l2) {
11. **if** (l1==**null**||l2==**null**) **return** l1==**null**?l2:l1;
12. ListNode head=**null**;
13. ListNode end= **null**;
15. **while**(l1!=**null**&&l2!=**null**) {
16. **if** (l1.val<l2.val) {
17. **if** (head==**null**) {
18. head = l1;
19. end = head;
20. } **else** {
21. end.next = l1;
22. end = end.next;
23. }
24. l1 = l1.next;
26. } **else** {
27. **if** (head==**null**) {
28. head = l2;
29. end =head;
30. } **else** {
31. end.next= l2;
32. end = end.next;
33. }
34. l2 = l2.next;
35. }
36. }
37. **if** (l2==**null**) {
38. end.next=l1;
39. } **else** {
40. end.next = l2;
41. }
42. **return** head;
44. }
45. }

26. Remove Duplicates from Sorted Array

从数组中删除重复元素，返回无重复长度的数组，要使用in place算法，把要保留的元素填入重复的位置即可

1. **public** **class** Solution {
2. **public** **int** removeDuplicates(**int**[] nums) {
3. **if** (nums==**null**||nums.length==0) **return** 0;
4. **int** result = 1;
5. **for** (**int** i=1;i<nums.length;i++) {
6. **if**(nums[i]!=nums[i-1]) {
7. nums[result]=nums[i];
8. result++;
9. }
11. }
12. **return** result;
13. }
14. }

27. Remove Element

1. **public** **class** Solution {
2. **public** **int** removeElement(**int**[] nums, **int** val) {
3. **if** (nums.length==0) **return** 0;
4. **int** result = 0;
5. **for** (**int** i=0;i<nums.length;i++) {
6. **if** (nums[i]!=val) {
7. nums[result++] = nums[i];
8. }
9. }
10. **return** result;
12. }
13. }

28. Implement strStr()

1. **public** **class** Solution {
2. **public** **int** strStr(String haystack, String needle) {
3. **for** (**int** i=0;i<haystack.length()-needle.length()+1;i++) {
4. **if** (haystack.substring(i,i+needle.length()).equals(needle)) {
5. **return** i;
6. }
7. }
8. **return** -1;
9. }
10. }

35. Search Insert Position

第一次依次查找，时间复杂度O(n)：

1. **public** **class** Solution {
2. **public** **int** searchInsert(**int**[] nums, **int** target) {
3. **for** (**int** i=0;i<nums.length;i++) {
4. **if** (nums[i]==target||nums[i]>target) {
5. **return** i;
6. } **else** {
7. **continue**;
8. }
10. }
11. **return** nums.length;
12. }
13. }

后来看答案想起应该用二分法，**有序数组查找记得二分**！

1. **public** **class** Solution {
2. **public** **int** searchInsert(**int**[] nums, **int** target) {
3. **int** low = 0,high=nums.length-1;
4. **while**(low<=high) {
5. **int** mid = low+(high-low)/2;
6. **if** (nums[mid]==target) {
7. **return** mid;
8. } **else** **if**(nums[mid]<target) {
9. low = mid+1;
10. } **else** {
11. high = mid-1;
12. }
13. }
14. **return** low;
15. }
16. }

38. Count and Say

从第一个数1开始，根据前一个数一直计算到n

两层循环，第一个是从1到n个数，第二个是每个String从前到后，注意出了内层循环要把最后一次count and say的值append进来

1. **public** **class** Solution {
2. **public** String countAndSay(**int** n) {
3. StringBuilder result  = **new** StringBuilder("1");
4. StringBuilder prev;
5. **char** say;
7. **for** (**int** i=1;i<n;i++) {
8. prev = result;
9. result = **new** StringBuilder("");
10. **int** count=1;
11. say = prev.charAt(0);
12. **for** (**int** j=1;j<prev.length();j++) {
13. **if** (prev.charAt(j)!= say) {
14. result.append(count).append(say);
15. say = prev.charAt(j);
16. count = 1;
17. } **else** {
18. count++;
19. }
20. }
21. result.append(count).append(say);
22. }
23. **return** result.toString();
24. }
25. }

53. Maximum Subarray

最开始的想法是，数组最大元素为最大值，先检查数组中每一个数，如果是负数，下一次循环，如果是正数，检查每个以该元素开始的序列的和，是否大于最大值。时间复杂度o(n^2),**容易超时**

1. **public** **class** Solution {
2. **public** **int** maxSubArray(**int**[] nums) {
3. **if** (nums==**null**||nums.length==0) **return** 0;
4. **int** lar = nums[0];
5. **for** (**int** i=0;i<nums.length;i++) {
6. **if** (nums[i]>lar) {
7. lar = nums[i];
8. }
9. }
10. **for** (**int** i=0;i<nums.length;i++) {
11. **if** (nums[i]>0) {
12. **int** sum = nums[i];
13. **for** (**int** j=i+1;j<nums.length;j++) {
14. sum += nums[j];
15. **if** (sum>lar) {
16. lar = sum;
17. }
18. }
19. } **else** {
20. **continue**;
21. }
22. }
23. **return** lar;
24. }
25. }
26. }

后来看答案，解法如下：

第一个元素为最大值，起始的和为0，遍历数组时如果和已经为负，从当前元素开始计算和（负数越加越小）。时间复杂度o(n)

1. **public** **class** Solution {
2. **public** **int** maxSubArray(**int**[] nums) {
3. **int** max = nums[0], sum = 0;
4. **for** (**int** i = 0; i < nums.length; i++) {
5. **if** (sum < 0) {
6. sum = nums[i];
7. } **else** {
8. sum += nums[i];
9. }
10. **if** (sum > max)
11. max = sum;
12. }
13. **return** max;
14. }
15. }

58. Length of Last Word

**注意所有可能出现的情况**：’Hello World’ , ’a ’, ‘b a ’, ‘ a’

应该从最后一个不为空格的元素开始计算

1. **public** **class** Solution {
2. **public** **int** lengthOfLastWord(String s) {
3. **if** (s==**null**||s.length()==0) **return** 0;
4. **int** count =0;
5. **for** (**int** i=s.length()-1;i>=0;i--) {
6. **if** (count==0&&s.charAt(i)==' ') {
7. **continue**;
8. } **else** **if**(s.charAt(i)!=' '){
9. count++;
10. } **else** {
11. **return** count;
12. }
13. }
14. **return** count;
15. }
16. }

66. Plus One

一开始的思路是把数组各位转化成整数，加1之后再放进新数组，如下，但是会发生整数溢出！另外，把Array 各元素print出来使用Arrays.toString()方法

1. **public** **class** Solution {
2. **public** **int**[] plusOne(**int**[] digits) {
3. **int** len = digits.length;
4. **int** sum = 0;
5. **for** (**int** i=0;i<len;i++) {
6. sum = 10\*sum+digits[i];
7. }
8. **int** result = sum+1;
9. **int** bit=0;
10. **int** x = result;
11. **while**(x>0) {
12. bit++;
13. x/=10;
14. }
15. **int**[] answer = **new** **int**[bit];
16. **int** l = bit-1;
17. **while**(result>0) {
18. answer[l]=result%10;
19. result /= 10;
20. l--;
22. }
23. **return** answer;
24. }
25. }

这道题正确解法是，先从后往前看各位，如小于9，直接加1返回，如等于9，此位置0，继续看前一位，如果所有位都是9，建一个新数组，第一位为1，其余为0

1. **public** **class** Solution {
2. **public** **int**[] plusOne(**int**[] digits) {
3. **int** len=digits.length;
4. **for** (**int** i=len-1;i>=0;i--) {
5. **if** (digits[i]<9) {
6. digits[i]++;
7. **return** digits;
8. } **else** {
9. digits[i]=0;
10. }
11. }
12. **int**[] result = **new** **int**[len+1];
13. System.arraycopy(digits,0,result,1,len);
14. result[0]=1;
15. **return** result;
16. }
17. }

67. Add Binary

把两个字符串长度补齐，从最低位开始做加法，分情况讨论，注意记录进位信息

1. **public** **class** Solution {
2. **public** String addBinary(String a, String b) {
3. String result ="";
4. **int** al = a.length();
5. **int** bl = b.length();
6. **int** diff = Math.abs(al-bl);
7. **int** len = Math.max(al,bl);
8. **for** (**int** i=1;i<=diff;i++) {
9. **if** (al>bl) {
10. b= "0"+b;
11. } **else** {
12. a= "0"+a;
13. }
14. }
15. **boolean** add = **false**;
16. **for** (**int** i=len-1;i>=0;i--) {
17. **if** (a.charAt(i)=='1'&&b.charAt(i)=='1') {
18. **if** (add) {
19. result = "1"+result;
21. } **else** {
22. result = "0"+result;
23. }
24. add =**true**;
25. } **else** **if** ((a.charAt(i)=='1'&&b.charAt(i)=='0')||(a.charAt(i)=='0'&&b.charAt(i)=='1')) {
27. **if** (add) {
28. result = "0"+result;
29. add=**true**;
30. } **else** {
31. result = "1"+result;
32. add=**false**;
33. }
34. } **else** **if**(a.charAt(i)=='0'&&b.charAt(i)=='0') {
35. **if** (add) {
36. result = "1"+result;
37. } **else** {
38. result = "0"+result;
39. }
40. add=**false**;
41. }
42. }
43. **if** (add) {
44. result ="1"+result;
45. }
46. **return** result;
47. }
48. }

69. Sqrt(x)

二分法要防止溢出，**int** mid=lo+(hi-lo)/2;  mid>x/mid

1. **public** **class** Solution {
2. **public** **int** mySqrt(**int** x) {
3. **if** (x<=1) **return** x;
4. **int** lo=0;
5. **int** hi=x;
6. **while**(lo<=hi) {
7. **int** mid=lo+(hi-lo)/2;
8. **if** (mid>x/mid) {
9. hi=mid-1;
10. } **else** **if**(mid<x/mid) {
11. lo = mid+1;
12. } **else** {
13. **return** mid;
14. }
15. }
16. **return** hi;
18. }
19. }

70. Climbing Stairs

一开始用的递归，超时了

1. **public** **class** Solution {
2. **public** **int** climbStairs(**int** n) {
3. **if** (n<=1) **return** 1;
4. **if** (n==2) **return** 2;
5. **return** climbStairs(n-1) + climbStairs(n-2);
6. }
7. }

因为形式与斐波那契数列相同，即最后一个数等于前两个数之和，可以仿照斐波那契写出求和表达式

1. **public** **class** Solution {
2. **public** **int** climbStairs(**int** n) {
3. **if** (n<=1) **return** 1;
4. **if** (n==2) **return** 2;
5. **int** one=1;
6. **int** two=2;
7. **int** result=0;
8. **for**(**int** i=2;i<n;i++) {
9. result =one+two;
10. one = two;
11. two =result;
12. }
13. **return** result;
14. }
15. }

83. Remove Duplicates from Sorted List

两个指针，一个当前指针，一个先前指针

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** ListNode deleteDuplicates(ListNode head) {
12. **if** (head==**null**||head.next==**null**) {
13. **return** head;
14. }
15. ListNode prev=head;
16. ListNode curr=head.next;
17. **while**(curr!=**null**) {
18. **if** (prev.val==curr.val) {
19. prev.next=curr.next;
20. } **else** {
21. prev = curr;
22. }
23. curr = curr.next;
25. }
26. **return** head;
27. }
28. }

88. Merge Sorted Array

一开始想的是建一个新的数组，把所有元素都复制过来，但是这样太麻烦了

1. **class** Solution {
2. **public** **void** merge(**int**[] nums1, **int** m, **int**[] nums2, **int** n) {
3. **int**[] result = **new** **int**[n+m];
4. **int** i=0,j=0,k=0;
5. **while**(i<m&&j<n) {
6. **if** (nums1[i]<nums2[j]) {
7. result[k]=nums1[i];
8. i++;
9. } **else** {
10. result[k] = nums2[j];
11. j++;
12. }
13. k++;
14. }
15. **int** z=k;
16. **if** (i==m) {
17. **for** (**int** x=k-m;x<n;x++) {
18. result[z++]=nums2[x];
19. }
20. } **else** {
21. **for** (**int** y=k-n;y<m;y++) {
22. result[z++]=nums1[y];
23. }
24. }
25. **for** (i=0;i<m+n;i++) {
26. nums1[i]=result[i];
27. }
29. }
30. }

此题应该用原地算法，从两个数组最后一个元素开始比较，大者放入第一个数组的相应位置，依次前推，如果第二个数组已经到头则停止（第一个数组元素位置已经排列好），如果第一个数组到头则把第二个数组余下元素都复制进来

1. **class** Solution {
2. **public** **void** merge(**int**[] nums1, **int** m, **int**[] nums2, **int** n) {
3. **int** i=m-1,j=n-1,k=n+m-1;
4. **while**(i>=0&&j>=0) {
5. **if** (nums1[i]>nums2[j]) {
6. nums1[k--]=nums1[i--];
7. } **else** {
8. nums1[k--]=nums2[j--];
9. }
10. }
11. **while**(j>=0) {
12. nums1[k--] = nums2[j--];
13. }
14. }
15. }

100. Same Tree

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **boolean** isSameTree(TreeNode p, TreeNode q) {
12. **if** (p==**null**&&q==**null**) **return** **true**;
13. **if**(p==**null**||q==**null**) **return** **false**;
14. **if** (p.val==q.val) {
15. **return** isSameTree(p.left,q.left)&&isSameTree(p.right,q.right);
16. }
17. **return** **false**;
18. }
19. }

101. Symmetric Tree

递归的方法：

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **boolean** isSymmetric(TreeNode root) {
12. **return** root==**null**||isSymmetricHelp(root.left,root.right);
14. }
15. **public** **boolean** isSymmetricHelp(TreeNode left,TreeNode right) {
16. **if** (left==**null**||right==**null**) {
17. **return** left==right;
18. }
19. **if** (left.val!=right.val) {
20. **return** **false**;
21. }
22. **return** isSymmetricHelp(left.left,right.right)&&isSymmetricHelp(left.right,right.left);
23. }
24. }

迭代的方法：

tree的迭代一般都会用Queue或Stack。此题中构建一个队列，分别把左节点的左节点和右节点的右节点，左节点的右节点和右节点的左节点，两两成对装进队列做比较，循环条件是队列个数大于1。注意null进队列个数也会加1.

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **boolean** isSymmetric(TreeNode root) {
12. **if** (root==**null**) **return** **true**;
14. Queue<TreeNode> queue = **new** LinkedList<TreeNode>();
15. queue.add(root.left);
16. queue.add(root.right);
17. **while**(queue.size()>1){
18. TreeNode left = queue.poll();
19. TreeNode right = queue.poll();
20. **if** (left==**null**&&right==**null**) **continue**;
21. **if** (left==**null**||right==**null**) **return** **false**;
22. **if** (left.val!=right.val) **return** **false**;
23. queue.add(left.left);
24. queue.add(right.right);
25. queue.add(left.right);
26. queue.add(right.left);

29. }
30. **return** **true**;
32. }
33. }

104. Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **int** maxDepth(TreeNode root) {
12. **if** (root==**null**) **return** 0;
13. **return** 1+Math.max(maxDepth(root.left),maxDepth(root.right));
14. }
15. }

107. Binary Tree Level Order Traversal II

Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:  
Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its bottom-up level order traversal as:

[

[15,7],

[9,20],

[3]

]

此题考查对数据结构Queue和List的应用， Queue先进先出，Stack后进先出

注意初始化List时要具化到ArrayList/LinkedList：

List<List<Integer>> result = **new** LinkedList<List<Integer>>();

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** List<List<Integer>> levelOrderBottom(TreeNode root) {
12. List<List<Integer>> result = **new** LinkedList<List<Integer>>();
13. **if** (root==**null**) **return** result;
14. Queue<TreeNode> queue = **new** LinkedList<TreeNode>();
15. queue.offer(root);
16. **while**(!queue.isEmpty()) {
17. List<Integer> list = **new** LinkedList<Integer>();
18. **int** size = queue.size();
19. **for** (**int** i=0;i<size;i++) {
20. TreeNode curr = queue.poll();
21. list.add(curr.val);
22. **if** (curr.left!=**null**) {
23. queue.offer(curr.left);
24. }
25. **if** (curr.right!=**null**) {
26. queue.offer(curr.right);
27. }
28. }
29. result.add(0,list);
30. }
31. **return** result;
33. }
34. }

108. Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

**Example:**

Given the sorted array: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

0

/ \

-3 9

/ /

-10 5

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** TreeNode sortedArrayToBST(**int**[] nums) {
12. **if** (nums == **null** || nums.length == 0) {
13. **return** **null**;
14. }
15. **return** sortedArrayToBSTHelper(nums,0,nums.length-1);
16. }
18. **public** TreeNode sortedArrayToBSTHelper(**int**[] nums,**int** low, **int** high) {
19. **if** (low > high) **return** **null**;
21. **int** mid = low + (high - low)/2;
22. TreeNode root = **new** TreeNode(nums[mid]);
23. root.left = sortedArrayToBSTHelper(nums,low,mid-1);
24. root.right = sortedArrayToBSTHelper(nums,mid+1,high);
25. **return** root;
26. }
27. }

109. Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

**Example:**

Given the sorted linked list: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

0

/ \

-3 9

/ /

-10 5

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. /\*\*
10. \* Definition for a binary tree node.
11. \* public class TreeNode {
12. \*     int val;
13. \*     TreeNode left;
14. \*     TreeNode right;
15. \*     TreeNode(int x) { val = x; }
16. \* }
17. \*/
18. **class** Solution {
20. **public** TreeNode sortedListToBST(ListNode head) {
21. **if** (head == **null**) **return** **null**;
22. ListNode slow =  head;
23. ListNode fast = head;
24. ListNode prev = **null**;
25. **while** (fast.next != **null**) {
26. fast = fast.next;
27. **if** (fast.next != **null**) {
29. prev = slow;
30. slow = slow.next;
31. fast = fast.next;
32. }
33. }
34. **if** (prev != **null**) {
35. prev.next = **null**;
36. } **else** {
37. head = **null**;
38. }
39. TreeNode root = **new** TreeNode(slow.val);
40. root.left = sortedListToBST(head);
41. root.right =  sortedListToBST(slow.next);
42. **return** root;
43. }
44. }

110. Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

一开始直接计算左右子树高度绝对值，发现是错的，因为每个节点都要求balanced

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **boolean** isBalanced(TreeNode root) {
12. **if** (root==**null**) {
13. **return** **true**;
14. }
15. **return** Math.abs(height(root.left)-height(root.right))<=1;
16. }
17. **public** **int** height(TreeNode node) {
18. **if** (node==**null**) {
19. **return** 0;
20. }
21. **return** 1+Math.max(height(node.left),height(node.right));
22. }
23. }

要注意需要记录每一个节点高度，用-1表示已经有节点不满足balanced的条件

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **boolean** isBalanced(TreeNode root) {
12. **if** (root==**null**) {
13. **return** **true**;
14. }
15. **return** height(root)!=-1;
16. }
17. **public** **int** height(TreeNode node) {
18. **if** (node==**null**) {
19. **return** 0;
20. }
21. **int** left = height(node.left);
22. **int** right = height(node.right);
23. **if** (left==-1||right==-1) **return** -1;
24. **if** (Math.abs(left-right)>1) **return** -1;
25. **return** 1+Math.max(left,right);
26. }
27. }

111. Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

这道题和Maximum Depth的区别是，求Maximun可以直接递归，Minimum一定要先判断下左右子树是否为空，为空的话不符合down to the nearest leaf node的条件

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **int** minDepth(TreeNode root) {
12. **if** (root==**null**) **return** 0;
13. **if** (root.left!=**null**&&root.right==**null**) {
14. **return** 1+minDepth(root.left);
15. }
16. **if** (root.right!=**null**&&root.left==**null**) {
17. **return** 1+minDepth(root.right);
18. }
19. **return** 1+Math.min(minDepth(root.left),minDepth(root.right));
20. }
22. }

112. Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:  
Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

此题递归，要理解题意root-to-leaf,一定要从上至下，不能中间结束，root.left==null&&root.right==null 即是限制条件

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **boolean** hasPathSum(TreeNode root, **int** sum) {
12. **if** (root==**null**) **return** **false**;
13. **int** remain = sum-root.val;
14. **if** (remain==0&&root.left==**null**&&root.right==**null**) {
15. **return** **true**;
16. }
17. **return** hasPathSum(root.left,remain)||hasPathSum(root.right,remain);
18. }
19. }

113. Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:  
Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ / \

7 2 5 1

return

[

[5,4,11,2],

[5,8,4,5]

]

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** List<List<Integer>> pathSum(TreeNode root, **int** sum) {
12. List<List<Integer>> res = **new** ArrayList<>();
13. pathSumHelper(root,res,**new** ArrayList(),sum);
14. **return** res;
15. }
16. **public** **void** pathSumHelper(TreeNode root, List<List<Integer>> res, List<Integer> subList, **int** remain) {
17. **if** (root == **null**) **return**;
18. subList.add(root.val);
19. **if** (remain == root.val && root.left == **null** && root.right == **null**) {
20. res.add(**new** ArrayList(subList));
21. }
22. pathSumHelper(root.left,res,subList,remain-root.val);
23. pathSumHelper(root.right,res,subList,remain-root.val);
24. subList.remove(subList.size() - 1);
25. }
26. }

118. Pascal's Triangle

Given *numRows*, generate the first *numRows* of Pascal's triangle.

For example, given *numRows* = 5,  
Return

[

[1],

[1,1],

[1,2,1],

[1,3,3,1],

[1,4,6,4,1]

]

1. **public** **class** Solution {
2. **public** List<List<Integer>> generate(**int** numRows) {
3. List<List<Integer>> result = **new** ArrayList<List<Integer>>();
4. List<Integer> temp = **new** ArrayList<Integer>();
5. **for** (**int** i=1;i<=numRows;i++) {
6. List<Integer> sublist = **new** ArrayList<Integer>();
7. sublist.add(1);
8. **if** (temp.size()>0) {
9. **for** (**int** j=0;j<temp.size()-1;j++) {
10. sublist.add(temp.get(j)+temp.get(j+1));
11. }
12. sublist.add(1);
13. }
14. temp = sublist;
15. result.add(sublist);
16. }
17. **return** result;
18. }
19. }

119. Pascal's Triangle II

Given an index *k*, return the *k*th row of the Pascal's triangle.

For example, given *k* = 3,  
Return [1,3,3,1].

此题为保证空间复杂度为o(k), 不能如上题每次都创建一个list，需要只建一个list每次都增加一个值，根据三角形规律计算list中每个元素的值

1. **public** **class** Solution {
2. **public** List<Integer> getRow(**int** rowIndex) {
3. List<Integer> result = **new** ArrayList<Integer>();
4. **for** (**int** i=0;i<=rowIndex;i++) {
5. result.add(0,1);
6. **int** size = result.size();
7. **for** (**int** j=1;j<size-1;j++) {
8. result.set(j,result.get(j)+result.get(j+1));
9. }
10. }
11. **return** result;
12. }
13. }

121. Best Time to Buy and Sell Stock

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

**Example 1:**

Input: [7, 1, 5, 3, 6, 4]

Output: 5

max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)

**Example 2:**

Input: [7, 6, 4, 3, 1]

Output: 0

In this case, no transaction is done, i.e. max profit = 0.

一开始用的bruteforce，时间复杂度O(n^2), 超时

1. **public** **class** Solution {
2. **public** **int** maxProfit(**int**[] prices) {
3. **int** len=prices.length;
4. **int** result=0;
5. **if** (len<=1) **return** 0;
6. **for** (**int** i=0;i<len-1;i++) {
7. **for** (**int** j=i+1;j<len) {
8. **if** (prices[i]>=prices[j]) {
9. **continue**;
10. } **else** {
11. diff = prices[j]-prices[i];
12. **if** (diff>result) {
13. result = diff;
14. }
15. }
16. }
17. }
18. **return** result;
19. }
20. }

注意以后遇到这种问题要想办法优化时间复杂度为O(n), 此题遍历时用两个变量：最小值与最大差值，每次遇到一个最小值则保存，否则比较此时的最大差值和之前保存值，取较大者。

1. **public** **class** Solution {
2. **public** **int** maxProfit(**int**[] prices) {
3. **int** len=prices.length;
4. **int** max=0;
5. **if** (len<1) **return** 0;
6. **int** minPrice = prices[0];
7. **for** (**int** i=0;i<len;i++) {
8. **if** (prices[i]<minPrice) {
9. minPrice = prices[i];
10. } **else** {
11. max = Math.max(max,prices[i]-minPrice);
12. }
13. }
14. **return** max;
15. }
16. }

122. Best Time to Buy and Sell Stock II

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

和上一题相似

1. **public** **class** Solution {
2. **public** **int** maxProfit(**int**[] prices) {
3. **int** max = 0;
4. **int** len = prices.length;
5. **if** (len<1) **return** 0;
6. **int** minPrice = prices[0];
7. **for** (**int** i=0;i<len;i++){
8. **if** (prices[i]<minPrice) {
9. minPrice = prices[i];
10. } **else** {
11. max += prices[i]-minPrice;
12. minPrice = prices[i];
13. }
14. }
15. **return** max;
16. }
17. }

125. Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,  
"A man, a plan, a canal: Panama" is a palindrome.  
"race a car" is *not* a palindrome.

注意String的操作，正则表达式［^a-zA-Z］是所有非字母，［^0-9a-zA-Z］是所有非数字和字母

1. **public** **class** Solution {
2. **public** **boolean** isPalindrome(String s) {
3. **if** (s==**null**||s.length()==0) **return** **true**;
4. s = s.toUpperCase().replaceAll("[^0-9a-zA-Z]","");
5. **for** (**int** i=0;i<s.length()/2;i++) {
6. **if** (s.charAt(i)!=s.charAt(s.length()-1-i)) {
7. **return** **false**;
8. }
9. }
10. **return** **true**;
11. }
12. }

136. Single Number

Given an array of integers, every element appears *twice* except for one. Find that single one.

**Note:**  
Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

这道题因为要保证时间复杂度O(n)且不使用多余的空间，最简单的方法是用异或的位运算，即^, 一个数与本身做异或是0，与0做异或是本身

1. **public** **class** Solution {
2. **public** **int** singleNumber(**int**[] nums) {
3. **int** result  = 0;
4. **for** (**int** i=0;i<nums.length;i++) {
5. result = result^nums[i];
6. }
7. **return** result;
8. }
9. }

141. Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up:  
Can you solve it without using extra space?

这道题是典型的双指针，注意起始一个指向head，一个指向head.next

1. /\*\*
2. \* Definition for singly-linked list.
3. \* class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) {
7. \*         val = x;
8. \*         next = null;
9. \*     }
10. \* }
11. \*/
12. **public** **class** Solution {
13. **public** **boolean** hasCycle(ListNode head) {
14. **if** (head==**null**||head.next==**null**) **return** **false**;
15. ListNode first = head;
16. ListNode second = head.next;
17. **while**(first!=second) {
18. **if** (second==**null**||second.next==**null**) {
19. **return** **false**;
20. }
21. first = first.next;
22. second = second.next.next;
23. }
24. **return** **true**;
25. }
26. }

155. Min Stack

第一种方法：只用一个栈，push时需要注意记录倒数第二小的数，以免在每次pop（）最小数后最小值丢失

1. **public** **class** MinStack {
2. List<Integer> list;
3. **int** min = Integer.MAX\_VALUE;
4. **public** MinStack(){
5. list = **new** ArrayList<Integer>();
6. }
7. **public** **void** push(**int** x) {
8. **if** (x<=min) {
9. list.add(min);
10. min = x;
11. }
12. list.add(x);
13. }
14. **public** **void** pop() {
15. **int** size = list.size();
16. **if** (list.get(size-1)==min) {
18. list.remove(size-1);
20. min=list.remove(size-2);
22. } **else** {
23. list.remove(size-1);
24. }
26. }
27. **public** **int** top() {
28. **return** list.get(list.size()-1);
29. }
30. **public** **int** getMin() {
31. **return** min;
32. }

35. }

第二种方法：用两个栈，第二个栈存放最小值，注意由于Stack中的元素用了泛型，而泛型不支持基本类型，所以需要基本类型的包装类。而包装类不同于基本类型的是包装类是引用，要比较相等不能简单使用==，要使用equals方法。

1. **public** **class** MinStack {
3. Stack<Integer> stack;
4. Stack<Integer> minStack;
5. **public** MinStack(){
6. stack  = **new** Stack<Integer>();
7. minStack = **new** Stack<Integer>();
9. }
10. **public** **void** push(**int** x) {
12. **if** (minStack.isEmpty()||x<=minStack.peek()) {
13. minStack.push(x);
14. }
15. stack.push(x);
17. }
18. **public** **void** pop() {
19. **if** (stack.peek().equals(minStack.peek())) {
20. minStack.pop();
21. }
22. stack.pop();
23. }
24. **public** **int** top() {
25. **return** stack.peek();
26. }
27. **public** **int** getMin() {
28. **return** minStack.peek();
29. }
30. }

160. Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

A: a1 → a2

↘

c1 → c2 → c3

↗

B: b1 → b2 → b3

begin to intersect at node c1.

**Notes:**

* If the two linked lists have no intersection at all, return null.
* The linked lists must retain their original structure after the function returns.
* You may assume there are no cycles anywhere in the entire linked structure.
* Your code should preferably run in O(n) time and use only O(1) memory.

这道题两种思路

1. 当A链走完去走B,B链走完去走A, 若两个链表相交，则走到某一处两节点指向会一致，否则最后会同时为空
2. /\*\*
3. \* Definition for singly-linked list.
4. \* public class ListNode {
5. \*     int val;
6. \*     ListNode next;
7. \*     ListNode(int x) {
8. \*         val = x;
9. \*         next = null;
10. \*     }
11. \* }
12. \*/
13. **public** **class** Solution {
14. **public** ListNode getIntersectionNode(ListNode headA, ListNode headB) {
15. **if** (headA==**null**||headB==**null**) {
16. **return** **null**;
17. }
18. ListNode hA = headA;
19. ListNode hB = headB;
20. **while**(hA!=hB) {
21. hA = (hA==**null**?headB:hA.next);
22. hB = (hB==**null**?headA:hB.next);
24. }
25. **return** hA;
27. }
28. }
29. 若两链表长度不相等（需先写函数算长度），使它们起始长度一致，分别比较下一个节点即可

167. Two Sum II - Input array is sorted

Given an array of integers that is already ***sorted in ascending order***, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have *exactly* one solution and you may not use the *same* element twice.

**Input:** numbers={2, 7, 11, 15}, target=9  
**Output:** index1=1, index2=2

不用HashMap的方法：

因为是有序的，把最小值与最大值相加，与target比较，若小于target，最左端（最小值）递增，若大于target，最右端（最大值）递减

1. **public** **class** Solution {
2. **public** **int**[] twoSum(**int**[] numbers, **int** target) {
3. **int**[] result = **new** **int**[2];
4. **int** low = 0;
5. **int** high = numbers.length-1;
6. **while**(numbers[low]+numbers[high]!=target) {
7. **if** (numbers[low]+numbers[high]>target) {
8. high--;
9. } **else** {
10. low++;
11. }
12. }
13. result[0] = low+1;
14. result[1] = high+1;
15. **return** result;
16. }
17. }

168. Excel Sheet Column Title

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

1 -> A

2 -> B

3 -> C

...

26 -> Z

27 -> AA

28 -> AB

将10进制转化成26进制，因为下表是从1开始，所以要先减1操作1

1. **public** **class** Solution {
2. **public** String convertToTitle(**int** n) {
3. String result = "";
4. **while**(n>0) {
5. n--;
6. result  = (**char**)('A'+n%26)+result;
7. n=n/26;
8. }
9. **return** result;
10. }
11. }

169. Majority Element

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

自己的解法如下，需要建立Map, 时间复杂度O(n),但空间复杂度较高

1. **public** **class** Solution {
2. **public** **int** majorityElement(**int**[] nums) {
3. **int** len = nums.length;
4. Map<Integer,Integer> map = HashMap<Integer,Integer>();
5. **for** (**int** i=0;i<len;i++) {
6. **if** (map.containsKey(nums[i])) {
7. **int** value = map.get(nums[i]);
8. map.put(nums[i],value+1);
9. **if** (map.get(nums[i])>n/2) {
10. result = nums[i];
11. **break**;
12. }
13. } **else** {
14. map.put(nums[i],1);
15. }
16. }
17. **return** result;
18. }
19. }

后来了解到Moore voting alogrithm， 此算法讲两个不相同的数成对消除，因为要找的数超过半数，最后剩下的一定是该数

1. **public** **class** Solution {
2. **public** **int** majorityElement(**int**[] nums) {
3. **int** len = nums.length;
4. **int** count = 0;
5. **int** result = nums[0];
6. **for** (**int** i=0;i<len;i++) {
7. **if** (nums[i]!= result) {
8. **if** (count==0) {
9. result = nums[i];
10. count=1;
11. } **else** **if**(count>0){
12. count--;
13. }
14. } **else** {
15. count++;
16. }
17. }
18. **return** result;
19. }
20. }

171. Excel Sheet Column Number

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

A -> 1

B -> 2

C -> 3

...

Z -> 26

AA -> 27

AB -> 28

1. **public** **class** Solution {
2. **public** **int** titleToNumber(String s) {
3. **int** result = 0;
4. **for**(**int** i = 0 ; i < s.length(); i++) {
5. result = result \* 26 + (s.charAt(i) - 'A' + 1);
6. }
7. **return** result;
8. }

172. Factorial Trailing Zeroes

Given an integer *n*, return the number of trailing zeroes in *n*!.

**Note:**Your solution should be in logarithmic time complexity.

求一个数阶乘后末尾的0， 因为2和5可以出0，2比5多，只要计算含有多少个5，注意25这样的数含有两个5

1. **public** **class** Solution {
2. **public** **int** trailingZeroes(**int** n) {
3. **int** result = 0;
4. **while**(n>0) {
5. n = n/5;
6. result+=n;
7. }
8. **return** result;
9. }
10. }

190. Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as **00000010100101000001111010011100**), return 964176192 (represented in binary as **00111001011110000010100101000000**).

此题用到位运算，左移<<,右移 >>,带符号右移>>>(没有<<<)

一个数n与1做与运算即可得到最右1位的数， 将n不断右移一位，再将结果一直左移，可以实现翻转

1. **public** **class** Solution {
2. // you need treat n as an unsigned value
3. **public** **int** reverseBits(**int** n) {
4. **int** result=0;
5. **for** (**int** i=0;i<32;i++) {
6. result = (result<<1)+(n&1);
7. n = n >>> 1;
8. }
9. **return** result;
10. }
11. }

191. Number of 1 Bits

Write a function that takes an unsigned integer and returns the number of ’1' bits it has (also known as the [Hamming weight](http://en.wikipedia.org/wiki/Hamming_weight)).

For example, the 32-bit integer ’11' has binary representation 00000000000000000000000000001011, so the function should return 3.

自己的解法如下：

1. **public** **class** Solution {
2. // you need to treat n as an unsigned value
3. **public** **int** hammingWeight(**int** n) {
4. **int** result = 0;
5. **for** (**int** i=0;i<32;i++) {
6. result = result +(n&1);
7. n = n>>>1;
8. }
9. **return** result;
10. }
11. }

剑指offer解法如下：

把一个数减去1，再和原数做与运算，会把该数最右边一个1变成0，有多少个1就可以进行多少次操作，不用循环32次

1. **public** **class** Solution {
2. // you need to treat n as an unsigned value
3. **public** **int** hammingWeight(**int** n) {
4. **int** result = 0;
5. **while**(n!=0) {
6. result++;
7. n=(n-1)&n;
8. }
9. **return** result;
10. }
11. }

198. House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

动态规划题，每个房子有两种情况，抢或者不抢，如果抢，则前一个房子一定不抢；如果不抢，前一个房子可以抢也可以不抢。用两个变量分别记录前一个房子的情况，从头到尾遍历。

1. **public** **class** Solution {
2. **public** **int** rob(**int**[] nums) {
3. **int** preIn = 0;
4. **int** preOut = 0;
5. **for** (**int** i=0;i<nums.length;i++) {
6. **int** temp = preIn;
7. preIn = preOut+nums[i];
8. preOut = Math.max(temp,preOut);
9. }
10. **return** Math.max(preIn,preOut);
11. }
12. }

202. Happy Number

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

**Example:**19 is a happy number

* 12 + 92 = 82
* 82 + 22 = 68
* 62 + 82 = 100
* 12 + 02 + 02 = 1

1. **public** **class** Solution {
2. **public** boolean isHappy(**int** n) {
3. Set<Integer> set = **new** HashSet<Integer>();
4. **while**(n!=1) {
5. n = bitSum(n);
6. **if** (set.contains(n)) {
7. **return** **false**;
8. } **else** {
9. set.add(n);
10. }
11. }
12. **return** **true**;
13. }
14. **public** **int** bitSum(**int** x) {
15. **int** sum=0;
16. **while**(x>0) {
17. sum+=(x%10)\*(x%10);
18. x/=10;
19. }
20. **return** sum;
21. }
22. }

203. Remove Linked List Elements

Remove all elements from a linked list of integers that have value ***val***.

**Example**  
***Given:*** 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, ***val*** = 6  
***Return:*** 1 --> 2 --> 3 --> 4 --> 5

这道题可以用一个fakeHead放在head前，这样可以避免考虑很多边界条件

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** ListNode removeElements(ListNode head, **int** val) {
11. ListNode fakeHead = **new** ListNode(-1);
12. fakeHead.next = head;
13. ListNode prev = fakeHead;
14. ListNode curr = head;
15. **while**(curr!=**null**) {
16. **if** (curr.val==val) {
17. prev.next=curr.next;
18. } **else** {
19. prev=prev.next;
20. }
21. curr = curr.next;
22. }
23. **return** fakeHead.next;
24. }
25. }

递归方法更简单：

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** ListNode removeElements(ListNode head, **int** val) {
11. **if**(head==**null**) **return** **null**;
12. head.next = removeElements(head.next,val);
13. **return** head.val==val?head.next:head;
14. }
15. }

204. Count Primes

**Description:**

Count the number of prime numbers less than a non-negative number, ***n***.

所有数为一个是否是质数的布尔数组，开始都是false，从2开始到n，分别将所有整数倍位置的布尔数组变为true，遍历时若该元素为true，结果加1

1. **public** **class** Solution {
2. **public** **int** countPrimes(**int** n) {
3. **boolean**[] notPrime = **new** **boolean**[n];
4. **int** count=0;
5. **for** (**int** i=2;i<n;i++) {
6. **if** (notPrime[i]==**false**) {
7. count++;
8. }
9. **for** (**int** j=2;i\*j<n;j++) {
10. notPrime[i\*j]=**true**;
11. }
12. }
13. **return** count;
14. }
15. }

205. Isomorphic Strings

Given two strings ***s*** and ***t***, determine if they are isomorphic.

Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example,  
Given "egg", "add", return true.

Given "foo", "bar", return false.

Given "paper", "title", return true.

需要注意的是，两个不同字母不能map同一值，因此要加上限制条件，检查map中的value

1. **public** **class** Solution {
2. **public** **boolean** isIsomorphic(String s, String t) {
3. **int** len=s.length();
4. Map<Character,Character> map = **new** HashMap<Character,Character>();
5. **for** (**int** i=0;i<len;i++) {
6. **if** (map.containsKey(s.charAt(i))) {
7. **if** (map.get(s.charAt(i))!=t.charAt(i)) {
8. **return** **false**;
9. } **else** {
10. **continue**;
11. }
12. } **else** {
13. **if** (map.containsValue(t.charAt(i))) {
14. **return** **false**;
15. }
16. map.put(s.charAt(i),t.charAt(i));
17. }
18. }
19. **return** **true**;
20. }
21. }

206. Reverse Linked List

Reverse a singly linked list.

分别用迭代和递归两种方法：

1. 迭代：从头到尾遍历，先保存下一个节点，使当前节点指向newHead, 再使newHead指向当前的head，之前保存的节点为新的head
2. /\*\*
3. \* Definition for singly-linked list.
4. \* public class ListNode {
5. \*     int val;
6. \*     ListNode next;
7. \*     ListNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** ListNode reverseList(ListNode head) {
12. ListNode newHead = **null**;
13. **while**(head!=**null**) {
14. ListNode next = head.next;
15. head.next = newHead;
16. newHead = head;
17. head =next;
18. }
19. **return** newHead;
20. }
21. }

递归：

1. **public** **class** Solution {
2. **public** ListNode reverseList(ListNode head) {
3. **if** (head==**null**) **return** head;
4. **return** reverseHelp(head,**null**);
6. }
7. **public** ListNode reverseHelp(ListNode curr,ListNode prev) {
8. **if** (curr.next==**null**) {
9. curr.next=prev;
10. **return** curr;
11. }
12. ListNode next = curr.next;
13. curr.next=prev;
14. **return** reverseHelp(next,curr);
15. }
16. }

217. Contains Duplicate

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

1. **public** **class** Solution {
2. **public** **boolean** containsDuplicate(**int**[] nums) {
3. Set<Integer> set = **new** HashSet<Integer>();
4. **for** (**int** i=0;i<nums.length;i++) {
5. **if** (set.contains(nums[i])) {
6. **return** **true**;
7. } **else** {
8. set.add(nums[i]);
9. }
10. }
11. **return** **false**;
12. }
13. }

219. Contains Duplicate II

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the **absolute** difference between *i* and *j* is at most *k*.

1. **public** **class** Solution {
2. **public** **boolean** containsNearbyDuplicate(**int**[] nums, **int** k) {
3. Map<Integer,Integer> map = **new** HashMap<Integer,Integer>();
4. **for**(**int** i=0;i<nums.length;i++) {
5. **if** (map.containsKey(nums[i])) {
6. **if** (i-map.get(nums[i])<=k) {
7. **return** **true**;
8. }
9. }
10. map.put(nums[i],i);
11. }
12. **return** **false**;
13. }
14. }

225. Implement Stack using Queues

Implement the following operations of a stack using queues.

* push(x) -- Push element x onto stack.
* pop() -- Removes the element on top of the stack.
* top() -- Get the top element.
* empty() -- Return whether the stack is empty.

**Notes:**

* You must use *only* standard operations of a queue -- which means only push to back, peek/pop from front, size, and is empty operations are valid.
* Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
* You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

用队列实现栈，这里用LinkedList实现，注意的地方是新元素进队列时要把前面元素重新装进队列以使新元素成为队列头

1. **public** **class** MyStack {
2. Queue<Integer> queue;
3. **public** MyStack() {
4. queue = **new** LinkedList<Integer>();
5. }
6. **public** **void** push(**int** x) {
7. queue.add(x);
8. **for** (**int** i=0;i<queue.size()-1;i++) {
9. queue.add(queue.poll());
10. }
11. }
12. **public** **int** pop() {
13. **return** queue.poll();
14. }
15. **public** **int** top() {
16. **return** queue.peek();
17. }
18. **public** **boolean** empty() {
19. **return** queue.isEmpty();
20. }
21. }
23. /\*\*
24. \* Your MyStack object will be instantiated and called as such:
25. \* MyStack obj = new MyStack();
26. \* obj.push(x);
27. \* int param\_2 = obj.pop();
28. \* int param\_3 = obj.top();
29. \* boolean param\_4 = obj.empty();
30. \*/

226. Invert Binary Tree

Invert a binary tree.

4

/ \

2 7

/ \ / \

1 3 6 9

to

4

/ \

7 2

/ \ / \

9 6 3 1

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** TreeNode invertTree(TreeNode root) {
12. **if** (root==**null**) **return** root;
13. TreeNode left = invertTree(root.right);
14. TreeNode right = invertTree(root.left);
15. root.left = left;
16. root.right = right;
17. **return** root;
18. }
19. }

231. Power of Two

Given an integer, write a function to determine if it is a power of two.

1. **public** **class** Solution {
2. **public** **boolean** isPowerOfTwo(**int** n) {
3. **if** (n==0) **return** **false**;
4. **while**(n%2==0) n/=2;
5. **return** n==1;
6. }
7. }

232. Implement Queue using Stacks

Implement the following operations of a queue using stacks.

* push(x) -- Push element x to the back of queue.
* pop() -- Removes the element from in front of queue.
* peek() -- Get the front element.
* empty() -- Return whether the queue is empty.

**Notes:**

* You must use *only* standard operations of a stack -- which means only push to top, peek/pop from top, size, and is empty operations are valid.
* Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
* You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

和225相似。

用队列实现栈，进队列时要想办法把新元素放进栈底，这样才能保证后进后出，push前用一个临时栈存放已有元素，push后再把原元素依次加进来

1. **public** **class** MyQueue {
2. Stack<Integer> stack;
3. **public** MyQueue() {
4. stack = **new** Stack<Integer>();
6. }
7. **public** **void** push(**int** x) {
8. Stack<Integer> stack2 = **new** Stack<Integer>();
10. **while**(!stack.isEmpty()) {
11. stack2.push(stack.pop());
12. }
13. stack.push(x);
14. **while**(!stack2.isEmpty()) {
15. stack.push(stack2.pop());
16. }
17. }
18. **public**  **int** pop() {
19. **return** stack.pop();
20. }
21. **public** **int** peek() {
22. **return** stack.peek();
23. }
25. **public** **boolean** empty() {
26. **return** stack.isEmpty();
27. }
29. }
30. /\*\*
31. \* Your MyQueue object will be instantiated and called as such:
32. \* MyQueue obj = new MyQueue();
33. \* obj.push(x);
34. \* int param\_2 = obj.pop();
35. \* int param\_3 = obj.peek();
36. \* boolean param\_4 = obj.empty();
37. \*/

234. Palindrome Linked List

Given a singly linked list, determine if it is a palindrome.

**Follow up:**  
Could you do it in O(n) time and O(1) space?

因为对时间复杂度，空间复杂度有要求，可以使后一半链表反转再和前一半比较，用两指针法定位链表中点，分别为一步和两步，这样当快指针走到头时慢指针走到一半处

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** **boolean** isPalindrome(ListNode head) {
11. **if** (head==**null**||head.next==**null**) {
12. **return** **true**;
13. }
14. ListNode slow = head;
15. ListNode fast = head.next;
16. **while**(fast.next!=**null**&&fast.next.next!=**null**) {
17. slow = slow.next;
18. fast = fast.next.next;
19. }
20. **if** (fast.next==**null**) {
21. slow = slow.next;
22. } **else** {
23. slow = slow.next.next;
24. }
25. ListNode second = reverse(slow);
26. **while**(second!=**null**) {
27. **if** (head.val!=second.val) {
28. **return** **false**;
29. }
30. second = second.next;
31. head = head.next;
32. }
33. **return** **true**;
34. }
36. **public** ListNode reverse(ListNode head) {
37. ListNode newHead = **null**;
38. **while**(head!=**null**) {
39. ListNode next = head.next;
40. head.next = newHead;
41. newHead = head;
42. head =next;
43. }
44. **return** newHead;
45. }
46. }

237. Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function.

这道题只知道节点信息，不知道前指针，实质上并没有删除节点而是把要删除节点下一个节点的值与删除节点替换，再把指向下一个节点的指针指向下下个节点，从而看上去好像删除了此节点

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** **void** deleteNode(ListNode node) {
11. node.val = node.next.val;
12. node.next = node.next.next;
13. }
14. }

235. Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”

\_\_\_\_\_\_\_6\_\_\_\_\_\_

/ \

\_\_\_2\_\_ \_\_\_8\_\_

/ \ / \

0 \_4 7 9

/ \

3 5

For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

这道题利用二叉搜索数的性质和递归，当两个节点都小于根节点时，最小公共节点一定在根的左子树，当两个节点都大于根节点时最小公共节点一定在根的右子树，否则在根节点一左一右，那么根节点就是最小公共节点

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
12. **if** (Math.max(p.val,q.val)<root.val) {
13. **return** lowestCommonAncestor(root.left,p,q);
14. } **else** **if** (Math.min(p.val,q.val)>root.val) {
15. **return** lowestCommonAncestor(root.right,p,q);
16. } **else** {
17. **return** root;
18. }
19. }
20. }

242. Valid Anagram

Given two strings *s* and *t*, write a function to determine if *t* is an anagram of *s*.

For example,  
*s* = "anagram", *t* = "nagaram", return true.  
*s* = "rat", *t* = "car", return false.

**Note:**  
You may assume the string contains only lowercase alphabets.

**Follow up:**  
What if the inputs contain unicode characters? How would you adapt your solution to such case?

一开始用的是Map,把s字符串每个字母和出现次数放进Map，再减去t字符串每个字母出现次数，检查最后的Map是不是全为0

1. **public** **class** Solution {
2. **public** **boolean** isAnagram(String s, String t) {
3. **int** sl = s.length();
4. **int** tl = t.length();
5. **if** (s==**null**||t==**null**||sl!=tl) **return** **false**;
6. Map<Character,Integer> map = **new** HashMap<Character,Integer>();
7. **for** (**int** i=0;i<sl;i++) {
8. **if** (map.containsKey(s.charAt(i))) {
9. map.put(s.charAt(i),map.get(s.charAt(i))+1);
10. } **else** {
11. map.put(s.charAt(i),1);
12. }
13. }
14. **for** (**int** j=0;j<tl;j++) {
15. **if** (map.containsKey(t.charAt(j))) {
16. map.put(t.charAt(j),map.get(t.charAt(j))-1);
17. **if** (map.get(t.charAt(j))<0) **return** **false**;
18. } **else** {
19. **return** **false**;
20. }
21. }
22. **for** (Character c:map.keySet()) {
23. **if** (map.get(c)!=0) {
24. **return** **false**;
25. }
26. }
27. **return** **true**;
28. }
29. }

其实可用更简单的数据结构，因为用26个字母，建一个长度为26的整形数组，遍历时每次找到s字符串字母在数组的相应位置，加1，遍历t字符串时减1，检查数组是否每个元素为0；

1. **public** **class** Solution {
2. **public** **boolean** isAnagram(String s, String t) {
3. **int** sl = s.length();
4. **int** tl = t.length();
5. **if** (sl!=tl) **return** **false**;
6. **int**[] letter = **new** **int**[26];
7. **for** (**int** i=0;i<sl;i++) letter[s.charAt(i)-'a']++;
8. **for** (**int** i=0;i<tl;i++) letter[t.charAt(i)-'a']--;
9. **for** (**int** j:letter) {
10. **if** (j!=0) {
11. **return** **false**;
12. }
13. }
14. **return** **true**;
15. }
16. }

还有一种方法可以拓展至unicode characters，将字符串转化成字符数组，排序后比价是否相等

1. **public** **class** Solution {
2. **public** **boolean** isAnagram(String s, String t) {
3. **int** sl = s.length();
4. **int** tl = t.length();
5. **if** (sl!=tl) **return** **false**;
6. **char**[] sc = s.toCharArray();
7. **char**[] tc = t.toCharArray();
8. Arrays.sort(sc);
9. Arrays.sort(tc);
10. **for** (**int** i=0;i<sl;i++) {
11. **if** (sc[i]!=tc[i]) {
12. **return** **false**;
13. }
14. }
15. **return** **true**;
16. }
17. }

257. Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

1

/ \

2 3

\

5

All root-to-leaf paths are:

["1->2->5", "1->3"]

遇到树的问题要想到递归，注意递归的结束条件

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** List<String> binaryTreePaths(TreeNode root) {
12. List<String> result = **new** ArrayList<String>();
13. **if** (root==**null**) **return** result;
15. **if** (root.left==**null**&&root.right==**null**) {
16. String string = Integer.toString(root.val);
17. result.add(string);
18. **return** result;
19. }
21. List<String> left = binaryTreePaths(root.left);
23. **for** (String path:left) {
24. result.add(Integer.toString(root.val)+"->"+path);
25. }
27. List<String> right = binaryTreePaths(root.right);
29. **for** (String path:right) {
30. result.add(Integer.toString(root.val)+"->"+path);
31. }
33. **return** result;
34. }
35. }

258. Add Digits

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

For example:

Given num = 38, the process is like: 3 + 8 = 11, 1 + 1 = 2. Since 2 has only one digit, return it.

**Follow up:**  
Could you do it without any loop/recursion in O(1) runtime?

1. **public** **class** Solution {
2. **public** **int** addDigits(**int** num) {
3. **if** (num==0) **return** 0;
4. **return** num%9==0?9:num%9;
5. }
6. }

263. Ugly Number

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Note that 1 is typically treated as an ugly number.

1. **public** **class** Solution {
2. **public** **boolean** isUgly(**int** num) {
3. **if** (num<=0) **return** **false**;
4. **if** (num==1) **return** **true**;
5. **while**(num%2==0) num/= 2;
6. **while**(num%3==0) num/= 3;
7. **while**(num%5==0) num/= 5;
8. **return** num==1;
9. }
10. }

268. Missing Number

Given an array containing *n* distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

**Example 1**

**Input:** [3,0,1]

**Output:** 2

**Example 2**

**Input:** [9,6,4,2,3,5,7,0,1]

**Output:** 8

1. **public** **class** Solution {
2. **public** **int** missingNumber(**int**[] nums) {
3. **int** sum=0;
4. **for** (**int** i=0;i<nums.length;i++) {
5. sum += nums[i];
6. }
7. **return** (1+nums.length)\*nums.length/2-sum;
8. }
9. }

此题可以用位运算，异或(^)

1. **public** **int** missingNumber(**int**[] nums) { //xor
2. **int** res = nums.length;
3. **for**(**int** i=0; i<nums.length; i++){
4. res ^= i;
5. res ^= nums[i];
6. }
7. **return** res;
8. }
9. 283. Move Zeroes

278. First Bad Version

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

因为要尽可能减少调用API的次数，用二分查找最快，需要注意的是循环条件没有“＝”,找到一个bad version后使其成为high，开始下一次查找

1. /\* The isBadVersion API is defined in the parent class VersionControl.
2. boolean isBadVersion(int version); \*/
4. **public** **class** Solution **extends** VersionControl {
5. **public** **int** firstBadVersion(**int** n) {
6. **int** low=1,high=n;
7. **while**(low<high) {
8. **int** mid = low+(high-low)/2;
9. **if** (isBadVersion(mid)==**true**) {
10. high =mid;
11. } **else** {
12. low = mid+1;
13. }
14. }
15. **return** low;
16. }
17. }

283. Move Zeroes

Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0].

**Note**:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.
3. **public** **class** Solution {
4. **public** **void** moveZeroes(**int**[] nums) {
5. **int** j = 0;
6. **for**(**int** i = 0; i < nums.length; i++) {
7. **if**(nums[i] != 0) {
8. **int** temp = nums[j];
9. nums[j] = nums[i];
10. nums[i] = temp;
11. j++;
12. }
13. }
14. }
15. }

290. Word Pattern

Given a pattern and a string str, find if str follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in pattern and a **non-empty** word in str.

**Examples:**

1. pattern = "abba", str = "dog cat cat dog" should return true.
2. pattern = "abba", str = "dog cat cat fish" should return false.
3. pattern = "aaaa", str = "dog cat cat dog" should return false.
4. pattern = "abba", str = "dog dog dog dog" should return false.

**Notes:**  
You may assume pattern contains only lowercase letters, and str contains lowercase letters separated by a single space.

注意出现abba—>dog dog dog dog的情况，字符串比较相等用equals()!

1. **public** **class** Solution {
2. **public** **boolean** wordPattern(String pattern, String str) {
3. String[] list=str.split(" ");
4. **int** pl = pattern.length();
5. **int** sl = list.length;
6. **if** (pl!=sl) **return** **false**;
7. Map<Character,String> map = **new** HashMap<Character,String>();
8. **for** (**int** i=0;i<pl;i++) {
9. **if** (map.containsKey(pattern.charAt(i))) {
10. **if** (!list[i].equals(map.get(pattern.charAt(i)))) {
11. **return** **false**;
12. }
13. } **else** **if**(map.containsValue(list[i])){
14. **return** **false**;
15. } **else** {
16. map.put(pattern.charAt(i),list[i]);
17. }
18. }
19. **return** **true**;
20. }
21. }

292. Nim Game

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

1. **public** **class** Solution {
2. **public** **boolean** canWinNim(**int** n) {
3. **return** n%4!=0;
4. }
5. }

303. Range Sum Query - Immutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

**Example:**

Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1

sumRange(2, 5) -> -1

sumRange(0, 5) -> -3

**Note:**

1. You may assume that the array does not change.
2. There are many calls to *sumRange* function.

题目中说有大量调用，因此不能每次都循环去计算，正确的方法是将每个数之前所有和保存下来，调用函数时做一次减法即可

1. **public** **class** NumArray {
2. **int**[] nums;
3. **public** NumArray(**int**[] nums) {
4. **for** (**int** i=1;i<nums.length;i++) {
5. nums[i]+=nums[i-1];
6. }
7. **this**.nums=nums;
8. }
9. **public** **int** sumRange(**int** i,**int** j) {
10. **if** (i==0) **return** nums[j];
11. **return** nums[j]-nums[i-1];
12. }
13. }
14. /\*\*
15. \* Your NumArray object will be instantiated and called as such:
16. \* NumArray obj = new NumArray(nums);
17. \* int param\_1 = obj.sumRange(i,j);
18. \*/

326. Power of Three

Given an integer, write a function to determine if it is a power of three.

**Follow up:**  
Could you do it without using any loop / recursion?

任何一个3的x次方一定能被int型里最大的3的x次方（3的19次方1162261467）整除

1. **public** **class** Solution{
2. **public** **boolean** isPowerOfThree(**int** n) {
3. **return** (n>0&&1162261467%n==0);
4. }
5. }

342. Power of Four

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

**Example:**  
Given num = 16, return true. Given num = 5, return false.

**Follow up**: Could you solve it without loops/recursion?

判断是否是4的n次方，首先必须是2的n次方，一个数如果是2的n次方那么有

num & (num - 1) = 0

如果是4的n次方，必须是奇数位为1，和0x55555555(奇数位都为1)做与运算后，值不变

1. **public** **class** Solution {
2. **public** **boolean** isPowerOfFour(**int** num) {
3. **return** num>0 && (num & (num-1))==0 && ((num & 0x55555555)==num);
4. }
5. }

345. Reverse Vowels of a String

Write a function that takes a string as input and reverse only the vowels of a string.

**Example 1:**  
Given s = "hello", return "holle".

**Example 2:**  
Given s = "leetcode", return "leotcede".

判断是否是原音字母最开始想到的是一个函数依次判断是否等于aeiou，也可使用数据结构Set

1. **public** **class** Solution {
2. **public** String reverseVowels(String s) {
3. **char**[] chars = s.toCharArray();
4. **int** i=0,j=chars.length-1;
5. Set<Character> set = **new** HashSet<Character>(){
6. {add('a');add('e');add('i');add('o');add('u');add('A');add('E');add('I');add('O');add('U');}
7. };
8. **while**(i<j) {
9. **while** (i<=j&&!set.contains(chars[i])) {
10. i++;
11. }
12. **while** (i<=j&&!set.contains(chars[j])) {
13. j--;
14. }
15. **if** (i<j) {
16. **char** temp = chars[i];
17. chars[i] = chars[j];
18. chars[j] = temp;
19. i++;
20. j--;
21. }
22. }
23. **return** **new** String(chars);
24. }
25. }

349. Intersection of Two Arrays

Given two arrays, write a function to compute their intersection.

**Example:**  
Given *nums1* = [1, 2, 2, 1], *nums2* = [2, 2], return [2].

**Note:**

* Each element in the result must be unique.
* The result can be in any order.

以后如遇到无序数组，也可用Arrays.sort()排序，也是一种思路

1. **public** **class** Solution {
2. **public** **int**[] intersection(**int**[] nums1, **int**[] nums2) {
3. Set<Integer> set = **new** HashSet<Integer>();
4. Set<Integer> intersection = **new** HashSet<Integer>();
5. **for** (**int** i=0;i<nums1.length;i++) {
6. set.add(nums1[i]);
7. }
8. **for** (**int** j=0;j<nums2.length;j++) {
9. **if** (set.contains(nums2[j])) {
10. intersection.add(nums2[j]);
11. }
12. }
13. **int**[] result = **new** **int**[intersection.size()];
14. **int** k=0;
15. **for** (**int** num:intersection) {
16. result[k++] = num;
17. }
18. **return** result;
19. }
20. }

350. Intersection of Two Arrays II

Given two arrays, write a function to compute their intersection.

**Example:**  
Given *nums1* = [1, 2, 2, 1], *nums2* = [2, 2], return [2, 2].

**Note:**

* Each element in the result should appear as many times as it shows in both arrays.
* The result can be in any order.

两种方法，一种是用map记录出现次数，一种是排序后用two pointers方法

**Follow up:**

* What if the given array is already sorted? How would you optimize your algorithm? **用两指针方法**
* What if *nums1*'s size is small compared to *nums2*'s size? Which algorithm is better?**如果是用map的方法，用size小的构建map**
* What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

**If only nums2 cannot fit in memory, put all elements of nums1 into a HashMap, read chunks of array that fit into the memory, and record the intersections.**

**If both nums1 and nums2 are so huge that neither fit into the memory, sort them individually (external sort), then read 2 elements from each array at a time in memory, record intersections.**

1. **public** **class** Solution {
2. **public** **int**[] intersect(**int**[] nums1, **int**[] nums2) {
3. **int** i=0,j=0;
4. **int** l1 = nums1.length,l2 = nums2.length;
5. List<Integer> list = **new** ArrayList<Integer>();
6. Arrays.sort(nums1);
7. Arrays.sort(nums2);
8. **while** (i<=l1-1&&j<=l2-1) {
9. **if** (nums1[i]>nums2[j]) {
10. j++;
11. } **else** **if** (nums1[i]<nums2[j]) {
12. i++;
13. } **else** {
14. list.add(nums1[i]);
15. i++;
16. j++;
17. }
18. }
19. **int**[] result = **new** **int**[list.size()];
20. **int** z=0;
21. **for** (**int** num:list) {
22. result[z++] = num;
23. }
24. **return** result;
25. }
26. }

367. Valid Perfect Square

Given a positive integer *num*, write a function which returns True if *num* is a perfect square else False.

**Note:** **Do not** use any built-in library function such as sqrt.

**Example 1:**

Input: 16

Returns: True

**Example 2:**

Input: 14

Returns: False

二分查找，一定要注意涉及到平方时防止溢出，这里用long型

1. **public** **class** Solution {
2. **public** **boolean** isPerfectSquare(**int** num) {
3. **if** (num<0) **return** **false**;
4. **if** (num<=1) **return** **true**;
5. **long** low=0;
6. **long** high = num;
7. **while**(low<=high) {
8. **long** mid = low+(high-low)/2;
9. **long** t = mid\*mid;
10. **if** (t==num) {
11. **return** **true**;
12. } **else** **if**(t<num){
13. low = mid+1;
14. } **else** {
15. high = mid-1;
16. }
17. }
18. **return** **false**;
19. }
20. }

371. Sum of Two Integers

Calculate the sum of two integers *a* and *b*, but you are **not allowed** to use the operator + and -.

**Example:**  
Given *a* = 1 and *b* = 2, return 3.

本题用到了位运算，异或(^)是无符号加法，需要再加上进位值，只有两个1相加才会进位，可以用&，因为进位是往前一位，再左移。

递归：

1. **public** **class** Solution {
2. **public** **int** getSum(**int** a, **int** b) {
3. **if** (a==0) **return** b;
4. **if** (b==0) **return** a;
5. **int** sum=a^b;
6. **int** carry = (a&b)<<1;
7. **return** getSum(sum,carry);
8. }
9. }

迭代：

1. **public** **class** Solution {
2. **public** **int** getSum(**int** a, **int** b) {
3. **if** (a==0) **return** b;
4. **if** (b==0) **return** a;
5. **while**(b!=0) {
6. **int** carry = (a&b)<<1;
7. a = a^b;
8. b = carry;
9. }
10. **return** a;
11. }
12. }

374. Guess Number Higher or Lower

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to ***n***. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0):

-1 : My number is lower

1 : My number is higher

0 : Congrats! You got it!

**Example:**

n = 10, I pick 6.

Return 6.

二分查找。

1. /\* The guess API is defined in the parent class GuessGame.
2. @param num, your guess
3. @return -1 if my number is lower, 1 if my number is higher, otherwise return 0
4. int guess(int num); \*/
6. **public** **class** Solution **extends** GuessGame {
7. **public** **int** guessNumber(**int** n) {
8. **int** low = 1,high =n;
9. **while**(low<=high) {
10. **int** mid = low+(high-low)/2;
11. **if** (guess(mid)==0) {
12. **return** mid;
13. } **else** **if**(guess(mid)==1){
14. low = mid+1;
15. } **else** {
16. high  =mid-1;
17. }
18. }
19. **return** low;
20. }
21. }

383. Ransom Note

Given an arbitrary ransom note string and another string containing letters from all the magazines, write a function that will return true if the ransom note can be constructed from the magazines ; otherwise, it will return false.

Each letter in the magazine string can only be used once in your ransom note.

**Note:**  
You may assume that both strings contain only lowercase letters.

canConstruct("a", "b") -> false

canConstruct("aa", "ab") -> false

canConstruct("aa", "aab") -> true

两种方法，分别用map和array作为数据结构

用Array:

1. **public** **class** Solution {
2. **public** **boolean** canConstruct(String ransomNote, String magazine) {
3. **char**[] mc = magazine.toCharArray();
4. **char**[] rc = ransomNote.toCharArray();
5. **int**[] alphbet = **new** **int**[26];
6. **for** (**int** i=0;i<mc.length;i++) {
7. alphbet[mc[i]-'a']++;
8. }
9. **for** (**int** j=0;j<rc.length;j++) {
10. alphbet[rc[j]-'a']--;
11. **if** (alphbet[rc[j]-'a']<0) **return** **false**;
12. }
13. **return** **true**;
14. }
15. }

用Map:

1. **public** **class** Solution {
2. **public** **boolean** canConstruct(String ransomNote, String magazine) {
3. **if** (ransomNote.length()==0) **return** **true**;
4. **int** rl = ransomNote.length();
5. **int** ml = magazine.length();
6. Map<Character,Integer> map = **new** HashMap<Character,Integer>();
7. **for** (**int** i=0;i<ml;i++) {
8. **char** c = magazine.charAt(i);
9. **if** (map.containsKey(c)) {
10. map.put(c,map.get(c)+1);
11. } **else** {
12. map.put(c,1);
13. }
14. }
15. **for** (**int** j=0;j<rl;j++) {
16. **char** t = ransomNote.charAt(j);
17. **if** (map.containsKey(t)&&map.get(t)>0) {
18. map.put(t,map.get(t)-1);
19. } **else** {
20. **return** **false**;
21. }
22. }
23. **return** **true**;
24. }
25. }

387. First Unique Character in a String

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

**Examples:**

s = "leetcode"

return 0.

s = "loveleetcode",

return 2.

这道题也用两种方法，分别是用map和字母表array两种数据结构

用Map:

1. **public** **class** Solution {
2. **public** **int** firstUniqChar(String s) {
3. Map<Character,Integer> map = **new** HashMap<Character,Integer>();
4. **for** (**int** i=0;i<s.length();i++) {
5. **if** (map.containsKey(s.charAt(i))) {
6. map.put(s.charAt(i),map.get(s.charAt(i))+1);
7. } **else** {
8. map.put(s.charAt(i),1);
9. }
10. }
11. **for** (**int** i=0;i<s.length();i++) {
12. **if** (map.get(s.charAt(i))==1) {
13. **return** i;
14. }
15. }
16. **return** -1;
17. }
18. }

字母表array:

1. **public** **class** Solution {
2. **public** **int** firstUniqChar(String s) {
3. **int**[] freq = **new** **int**[26];
4. **for** (**int** i=0;i<s.length();i++) {
5. freq[s.charAt(i)-'a']++;
6. }
7. **for** (**int** i=0;i<s.length();i++) {
8. **if** (freq[s.charAt(i)-'a'] == 1) {
9. **return** i;
10. }
11. }
12. **return** -1;
13. }
14. }

389. Find the Difference

Given two strings ***s*** and ***t*** which consist of only lowercase letters.

String ***t*** is generated by random shuffling string ***s*** and then add one more letter at a random position.

Find the letter that was added in ***t***.

**Example:**

Input:

s = "abcd"

t = "abcde"

Output:

e

Explanation:

'e' is the letter that was added.

这道题可以用字母表array方法，更简单的方法是把char转化成int，加上t的int总和减去s的int总和，剩余的数就是多出来的字符的int值，再转换过来

1. **public** **class** Solution {
2. **public** **char** findTheDifference(String s, String t) {
3. **int** code=(**int**)t.charAt(s.length());
4. **for** (**int** i=0;i<s.length();i++) {
5. code -= (**int**) s.charAt(i);
6. code += (**int**) t.charAt(i);
7. }
8. **return** (**char**) code;
9. }
10. }

400. Nth Digit

Find the *n*th digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

**Note:**  
*n* is positive and will fit within the range of a 32-bit signed integer (*n* < 231).

**Example 1:**

**Input:**

3

**Output:**

3

**Example 2:**

**Input:**

11

**Output:**

0

1. **public** **class** Solution {
2. **public** **int** findNthDigit(**int** n) {
3. **int** len = 1;
4. **long** count = 9;
5. **int** start = 1;
6. **while**(n > len\*count) {
7. n -= len\*count;
8. len += 1;
9. count \*= 10;
10. start \*= 10;
11. }
12. start += (n-1)/len;
13. String s = String.valueOf(start);
14. **return** Character.getNumericValue(s.charAt((n-1)%len));
15. }
16. }

401. Binary Watch

A binary watch has 4 LEDs on the top which represent the **hours** (**0-11**), and the 6 LEDs on the bottom represent the **minutes** (**0-59**).

Each LED represents a zero or one, with the least significant bit on the right.



For example, the above binary watch reads "3:25".

Given a non-negative integer *n* which represents the number of LEDs that are currently on, return all possible times the watch could represent.

**Example:**

Input: n = 1  
Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]

**Note:**

* The order of output does not matter.
* The hour must not contain a leading zero, for example "01:00" is not valid, it should be "1:00".
* The minute must be consist of two digits and may contain a leading zero, for example "10:2" is not valid, it should be "10:02".

这道题用Integer.bitCount()的方法最快， 即计算一个数转化成二进制后1的个数

1. **public** **class** Solution {
2. **public** List<String> readBinaryWatch(**int** num) {
3. List<String> result = **new** ArrayList<String>();
4. **for** (**int** n=0;n<12;n++) {
5. **for** (**int** m=0;m<60;m++) {
6. **if** (Integer.bitCount(n)+Integer.bitCount(m)==num) {
7. result.add(String.format("%d:%02d",n,m));
8. }
9. }
10. }
11. **return** result;
12. }
13. }

404. Sum of Left Leaves

Find the sum of all left leaves in a given binary tree.

**Example:**

3

/ \

9 20

/ \

15 7

There are two left leaves in the binary tree, with values **9** and **15** respectively. Return **24**.

树的问题，两种方法：

递归：

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **int** sumOfLeftLeaves(TreeNode root) {
12. **if** (root==**null**) **return** 0;
13. **if** (root.left==**null**) {
14. **return** sumOfLeftLeaves(root.right);
15. }
16. **if** (root.left.left==**null**&root.left.right==**null**) {
17. **return** root.left.val+sumOfLeftLeaves(root.right);
18. }
19. **return** sumOfLeftLeaves(root.right)+sumOfLeftLeaves(root.left);
21. }
22. }

迭代：

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **int** sumOfLeftLeaves(TreeNode root) {
12. **if** (root==**null**) **return** 0;
13. Deque<TreeNode> queue = **new** LinkedList<TreeNode>();
14. queue.offer(root);
15. **int** res = 0;
16. **while**(!queue.isEmpty()) {
17. TreeNode curr = queue.poll();
18. **if** (curr.left!=**null**) {
19. **if** (curr.left.left==**null**&&curr.left.right==**null**) {
20. res += curr.left.val;
21. } **else** {
22. queue.offer(curr.left);
23. }
24. }
25. **if** (curr.right!=**null**) {
26. queue.offer(curr.right);
27. }
28. }
29. **return** res;
30. }
31. }

405. Convert a Number to Hexadecimal

Given an integer, write an algorithm to convert it to hexadecimal. For negative integer, [two’s complement](https://en.wikipedia.org/wiki/Two%27s_complement) method is used.

**Note:**

1. All letters in hexadecimal (a-f) must be in lowercase.
2. The hexadecimal string must not contain extra leading 0s. If the number is zero, it is represented by a single zero character '0'; otherwise, the first character in the hexadecimal string will not be the zero character.
3. The given number is guaranteed to fit within the range of a 32-bit signed integer.
4. You **must not use *any* method provided by the library** which converts/formats the number to hex directly.

尽管题目说负数是用2的补数，但是计算机里数的二进制表示是不变的，用位操作&可以正负数直接操作不用预处理，过程是每次处理最低4位，将对应的16进制表示添加到字符串开头(注意StringBuilder insert方法)，循环移位时注意是无符号移位才能结束循环，用>>>

1. **public** **class** Solution {
2. **public** String toHex(**int** num) {
3. **if** (num==0) **return** "0";
4. StringBuilder result= **new** StringBuilder("");
5. String[] array = {"0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f"};
6. **while**(num!=0) {
7. result.insert(0,array[num&15]);
8. num = num >>>4;
9. }
10. **return** result.toString();
11. }
12. }

409. Longest Palindrome

Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes that can be built with those letters.

This is case sensitive, for example "Aa" is not considered a palindrome here.

**Note:**  
Assume the length of given string will not exceed 1,010.

**Example:**

Input:

"abccccdd"

Output:

7

Explanation:

One longest palindrome that can be built is "dccaccd", whose length is 7.

最开始的错误：map存各字符出现次数，偶数次直接放进去，奇数次取最大放进去。其实奇数次可以减1变成偶数后放进去

1. **public** **class** Solution {
2. **public** **int** longestPalindrome(String s) {
3. **int** ans = 0;
4. Map<Character,Integer> map = **new** HashMap<Character,Integer>();
5. **for** (**int** i=0;i<s.length();i++) {
6. **if** (map.containsKey(s.charAt(i))) {
7. map.put(s.charAt(i),map.get(s.charAt(i))+1);
8. } **else** {
9. map.put(s.charAt(i),1);
10. }
11. }
12. **int** odd = 0;
13. **for** (Character c:map.keySet()) {
14. **if** (map.get(c)%2==0) {
15. ans += map.get(c);
16. } **else** {
17. **if**(map.get(c)>odd) {
18. **if** (odd>1) {
19. ans += odd-1;
20. }
21. odd = map.get(c);
22. } **else** {
23. **if** (map.get(c)>1) {
24. ans += map.get(c)-1;
25. }
26. }
27. }
28. }
29. ans += odd;
30. **return** ans;
31. }
32. }

一种更简单的方法：看出现的对数，用set即可

1. **public** **class** Solution {
2. **public** **int** longestPalindrome(String s) {
4. **if** (s==**null**||s.length()==0) **return** 0;
5. Set<Character> set = **new** HashSet<Character>();
6. **int** count=0;
7. **for** (**int** i=0;i<s.length();i++) {
8. **if** (set.contains(s.charAt(i))) {
9. set.remove(s.charAt(i));
10. count++;
11. } **else** {
12. set.add(s.charAt(i));
13. }
14. }
15. **if** (set.isEmpty()) **return** count\*2;
16. **return** count\*2+1;
17. }
18. }

412. Fizz Buzz

Write a program that outputs the string representation of numbers from 1 to *n*.

But for multiples of three it should output “Fizz” instead of the number and for the multiples of five output “Buzz”. For numbers which are multiples of both three and five output “FizzBuzz”.

**Example:**

n = 15,

Return:

[

"1",

"2",

"Fizz",

"4",

"Buzz",

"Fizz",

"7",

"8",

"Fizz",

"Buzz",

"11",

"Fizz",

"13",

"14",

"FizzBuzz"

]

1. **public** **class** Solution {
2. **public** List<String> fizzBuzz(**int** n) {
3. List<String> result = **new** ArrayList<String>();
4. **if** (n<1) **return** result;
5. **for** (**int** i=1;i<=n;i++) {
6. **if** ((i%3==0)&&(i%5==0)) {
7. result.add("FizzBuzz");
8. } **else** **if** (i%3==0) {
9. result.add("Fizz");
10. } **else** **if** (i%5==0) {
11. result.add("Buzz");
12. } **else** {
13. result.add(String.valueOf(i));
14. }
15. }
16. **return** result;
17. }
18. }

414. Third Maximum Number

Given a **non-empty** array of integers, return the **third** maximum number in this array. If it does not exist, return the maximum number. The time complexity must be in O(n).

**Example 1:**

**Input:** [3, 2, 1]

**Output:** 1

**Explanation:** The third maximum is 1.

**Example 2:**

**Input:** [1, 2]

**Output:** 2

**Explanation:** The third maximum does not exist, so the maximum (2) is returned instead.

**Example 3:**

**Input:** [2, 2, 3, 1]

**Output:** 1

**Explanation:** Note that the third maximum here means the third maximum distinct number.

Both numbers with value 2 are both considered as second maximum.

这道题要注意的地方是，初始化要用Long.MIN\_VALUE, 否则如果数组中有Integer.MIN\_VALUE,无法判断返回max1还是max3

1. **public** **class** Solution {
2. **public** **int** thirdMax(**int**[] nums) {
3. **long** max1 = Long.MIN\_VALUE;
4. **long** max2 = Long.MIN\_VALUE;
5. **long** max3 = Long.MIN\_VALUE;
6. **for** (**int** i=0;i<nums.length;i++) {
7. **if** (nums[i]== max1||nums[i]==max2||nums[i]==max3) {
8. **continue**;
9. }
10. **if** (nums[i]>max1) {
11. max3=max2;
12. max2= max1;
13. max1 = nums[i];
14. } **else** **if**(nums[i]>max2) {
15. max3 = max2;
16. max2 = nums[i];
17. } **else** **if**(nums[i]>max3) {
18. max3=nums[i];
19. }
20. }
21. **return** max3==Long.MIN\_VALUE?(**int**)max1:(**int**)max3;
22. }
23. }

415. Add Strings

Given two non-negative integers num1 and num2 represented as string, return the sum of num1 and num2.

**Note:**

1. The length of both num1 and num2 is < 5100.
2. Both num1 and num2 contains only digits 0-9.
3. Both num1 and num2 does not contain any leading zero.
4. You **must not use any built-in BigInteger library** or **convert the inputs to integer** directly.

此题中注意题目限制条件及对边界条件的处理

1. **public** **class** Solution {
2. **public** String addStrings(String num1, String num2) {
3. StringBuilder s = **new** StringBuilder();
4. **int** carry = 0;
5. **for**(**int** i=num1.length()-1,j=num2.length()-1;i>=0||j>=0||carry==1;i--,j--) {
6. **int** x = i<0?0:num1.charAt(i)-'0';
7. **int** y = j<0?0:num2.charAt(j)-'0';
8. s.append((x+y+carry)%10);
9. carry = (x+y+carry)/10;
10. }
11. **return** s.reverse().toString();
12. }
13. }

434. Number of Segments in a String

Count the number of segments in a string, where a segment is defined to be a contiguous sequence of non-space characters.

Please note that the string does not contain any **non-printable** characters.

**Example:**

**Input:** "Hello, my name is John"

**Output:** 5

当一个字符非空格且如果前一位不是第一位是空格或者是第一位，结果都加1

1. **public** **class** Solution {
2. **public** **int** countSegments(String s) {
3. **int** count=0;
4. **for**(**int** i=0; i<s.length(); i++)
5. **if**(s.charAt(i)!=' ' && (i==0 || s.charAt(i-1)==' '))
6. count++;
7. **return** count;
8. }
9. }

正则表达式：

1. **public** **class** Solution {
2. **public** **int** countSegments(String s) {
3. **return** ("a "+s).split(" +").length-1;
4. }
5. }

437. Path Sum III

You are given a binary tree in which each node contains an integer value.

Find the number of paths that sum to a given value.

The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.

**Example:**

root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

10

/ \

**5** **-3**

**/** **\** **\**

**3** **2** **11**

/ \ **\**

3 -2 **1**

Return 3. The paths that sum to 8 are:

1. 5 -> 3

2. 5 -> 2 -> 1

3. -3 -> 11

一道DFS的题，注意跟节点可以包含可以不包含

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **public** **int** pathSum(TreeNode root, **int** sum) {
12. **if** (root==**null**) **return** 0;
13. **return** findPath(root,sum)+pathSum(root.left,sum)+pathSum(root.right,sum);
14. }
15. **public** **int** findPath(TreeNode root,**int** sum) {
16. **if** (root==**null**) **return** 0;
17. **return** sum==root.val?1+findPath(root.right,sum-root.val)+findPath(root.left,sum-root.val):findPath(root.right,sum-root.val)+findPath(root.left,sum-root.val);
18. }
19. }

438. Find All Anagrams in a String

Given a string **s** and a **non-empty** string **p**, find all the start indices of **p**'s anagrams in **s**.

Strings consists of lowercase English letters only and the length of both strings **s** and **p** will not be larger than 20,100.

The order of output does not matter.

**Example 1:**

**Input:**

s: "cbaebabacd" p: "abc"

**Output:**

[0, 6]

**Explanation:**

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

**Example 2:**

**Input:**

s: "abab" p: "ab"

**Output:**

[0, 1, 2]

**Explanation:**

The substring with start index = 0 is "ab", which is an anagram of "ab".

The substring with start index = 1 is "ba", which is an anagram of "ab".

The substring with start index = 2 is "ab", which is an anagram of "ab".

这道题用hash和滑动窗口的思想。

第一个注意定义的hash数组长度，因为ASCII码长度256，每一位索引表示一个字符的计数值。

其次是窗口的滑动，用两个指针表示窗口的左右边界，用count计算窗口里面的字符串和p字符串相差的个数。

窗口右移是如果一个位置hash值大于0说明这个字符在p中出现，count值减1，如果count为0，说明完全重合，左边界为一个答案，窗口长度为p长度时左边界移动，如果hash值非负说明这个位置在p中出现，count值要加1

1. **public** **class** Solution {
2. **public** List<Integer> findAnagrams(String s, String p) {
3. List<Integer> result = **new** ArrayList<Integer>();
4. **int**[] array = **new** **int**[256];
5. **for** (**char** c:p.toCharArray()) {
6. array[c]++;
7. }
8. **int** left=0,right=0,count=p.length();
9. **while**(right<s.length()) {
10. **if** (array[s.charAt(right)]>0) {
11. count--;
12. }
13. array[s.charAt(right)]--;
14. right++;
15. **if** (count==0) {
16. result.add(left);
17. }
18. **if** (right-left==p.length()) {
19. **if** (array[s.charAt(left)]>=0) {
20. count++;
21. }
23. array[s.charAt(left)]++;
24. left++;
25. }
26. }
27. **return** result;
28. }
29. }

441. Arranging Coins

You have a total of *n* coins that you want to form in a staircase shape, where every *k*-th row must have exactly *k* coins.

Given *n*, find the total number of **full** staircase rows that can be formed.

*n* is a non-negative integer and fits within the range of a 32-bit signed integer.

**Example 1:**

n = 5

The coins can form the following rows:

¤

¤ ¤

¤ ¤

Because the 3rd row is incomplete, we return 2.

**Example 2:**

n = 8

The coins can form the following rows:

¤

¤ ¤

¤ ¤ ¤

¤ ¤

Because the 4th row is incomplete, we return 3.

1. **public** **class** Solution {
2. **public** **int** arrangeCoins(**int** n) {
3. **return** (**int**)((Math.sqrt((**long**)8\*n + 1) - 1)/2);
4. }
5. }

447. Number of Boomerangs

Given *n* points in the plane that are all pairwise distinct, a "boomerang" is a tuple of points (i, j, k) such that the distance between i and j equals the distance between i and k (**the order of the tuple matters**).

Find the number of boomerangs. You may assume that *n* will be at most **500** and coordinates of points are all in the range **[-10000, 10000]** (inclusive).

**Example:**

**Input:**

[[0,0],[1,0],[2,0]]

**Output:**

2

**Explanation:**

The two boomerangs are **[[1,0],[0,0],[2,0]]** and **[[1,0],[2,0],[0,0]]**

以空间换时间，分别求出到一个点距离相等的所有点用map保存距离和计数，对同一个距离，所有点可以排列组合

1. **public** **class** Solution {
2. **public** **int** numberOfBoomerangs(**int**[][] points) {
3. **int** result = 0;
4. **for** (**int** i=0;i<points.length;i++) {
5. Map<Integer,Integer> map = **new** HashMap<Integer,Integer>();
6. **for** (**int** j=0;j<points.length;j++) {
7. **int** dis = (**int**)(Math.pow(points[i][0]-points[j][0],2)+Math.pow(points[i][1]-points[j][1],2));
8. map.put(dis,map.getOrDefault(dis,0)+1);
9. }
10. **for** (**int** dis:map.keySet()) {
11. result +=map.get(dis)\*(map.get(dis)-1);
12. }
13. map.clear();
14. }
15. **return** result;
16. }
17. }

448. Find All Numbers Disappeared in an Array

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements of [1, *n*] inclusive that do not appear in this array.

Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.

**Example:**

**Input:**

[4,3,2,7,8,2,3,1]

**Output:**

[5,6]

这道题注意对时间和空间复杂度的要求。每遍历一个数，把数组中属于该数位置的数置为负数，表示数组中包含此数，最后只要找到数组中非负数，即为缺失数

1. **public** **class** Solution {
2. **public** List<Integer> findDisappearedNumbers(**int**[] nums) {
3. List<Integer> ans = **new** ArrayList<Integer>();
4. **for** (**int** i=0;i<nums.length;i++) {
5. **int** val = Math.abs(nums[i])-1;
6. **if** (nums[val]>0) nums[val] = -nums[val];
7. }
8. **for** (**int** i=0;i<nums.length;i++) {
9. **if** (nums[i]>0) {
10. ans.add(i+1);
11. }
12. }
13. **return** ans;
14. }
15. }

453. Minimum Moves to Equal Array Elements

Given a **non-empty** integer array of size *n*, find the minimum number of moves required to make all array elements equal, where a move is incrementing *n* - 1 elements by 1.

**Example:**

**Input:**

[1,2,3]

**Output:**

3

**Explanation:**

Only three moves are needed (remember each move increments two elements):

[1,2,3] => [2,3,3] => [3,4,3] => [4,4,4]

这是一道数学题，基于以下两个关系式

sum(Array)+x\*(len-1)=finalValue\*len

finalValue = min + x （最小值每次都要加1）

1. **public** **class** Solution {
2. **public** **int** minMoves(**int**[] nums) {
3. **int** len = nums.length;
4. **int** sum = 0;
5. **int** min = nums[0];
6. **for** (**int** i=0;i<len;i++) {
7. sum += nums[i];
8. min = Math.min(min,nums[i]);
9. }
10. **return** sum-len\*min;
11. }
12. }

455. Assign Cookies

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie. Each child i has a greed factor gi, which is the minimum size of a cookie that the child will be content with; and each cookie j has a size sj. If sj >= gi, we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

**Note:**  
You may assume the greed factor is always positive.   
You cannot assign more than one cookie to one child.

**Example 1:**

**Input:** [1,2,3], [1,1]

**Output:** 1

**Explanation:** You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3.

And even though you have 2 cookies, since their size is both 1, you could only make the child whose greed factor is 1 content.

You need to output 1.

**Example 2:**

**Input:** [1,2], [1,2,3]

**Output:** 2

**Explanation:** You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2.

You have 3 cookies and their sizes are big enough to gratify all of the children,

You need to output 2.

双指针法

1. **public** **class** Solution {
2. **public** **int** findContentChildren(**int**[] g, **int**[] s) {
3. **int** i=0,j=0;
4. **int** ans=0;
5. Arrays.sort(g);
6. Arrays.sort(s);
7. **while**(i<g.length&&j<s.length) {
8. **if** (g[i]<=s[j]) {
9. ans++;
10. i++;
11. }
12. j++;
13. }
14. **return** ans;
15. }
16. }

459. Repeated Substring Pattern

Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000.

**Example 1:**

**Input:** "abab"

**Output:** True

**Explanation:** It's the substring "ab" twice.

**Example 2:**

**Input:** "aba"

**Output:** False

**Example 3:**

**Input:** "abcabcabcabc"

**Output:** True

**Explanation:** It's the substring "abc" four times. (And the substring "abcabc" twice.)

这道题注意字符串复制对StringBuilder的使用

1. **public** **class** Solution {
2. **public** **boolean** repeatedSubstringPattern(String s) {
4. **int** flag =0;
5. **int** l = s.length();
6. **for** (**int** i=0;i<l/2;i++) {
7. String tem = s.substring(0,i+1);
8. **int** tl = tem.length();
9. **if** (l%tl==0) {
10. StringBuilder sb = **new** StringBuilder("");
11. **for** (**int** j=1;j<=l/tl;j++) {
12. sb.append(tem);
13. }
14. **if** (sb.toString().equals(s)) **return** **true**;
15. }
16. }
17. **return** **false**;
18. }
19. }

461. Hamming Distance

The [Hamming distance](https://en.wikipedia.org/wiki/Hamming_distance) between two integers is the number of positions at which the corresponding bits are different.

Given two integers x and y, calculate the Hamming distance.

**Note:**  
0 ≤ x, y < 231.

**Example:**

**Input:** x = 1, y = 4

**Output:** 2

**Explanation:**

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

The above arrows point to positions where the corresponding bits are different.

对异或运算的应用

1. **public** **class** Solution {
2. **public** **int** hammingDistance(**int** x, **int** y) {
3. **int** res = x^y;
4. **int** ans = 0;
5. **while** (res!=0) {
6. **if** ((res&1)==1) {
7. ans++;
8. }
9. res = res>>1;
10. }
11. **return** ans;
12. }
13. }

也可以直接用Java提供的方法， Integer.bitCount()计算有几个1

1. **public** **class** Solution {
2. **public** **int** hammingDistance(**int** x, **int** y) {
3. **return** Integer.bitCount(x ^ y);
4. }
5. }

463. Island Perimeter

You are given a map in form of a two-dimensional integer grid where 1 represents land and 0 represents water. Grid cells are connected horizontally/vertically (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells). The island doesn't have "lakes" (water inside that isn't connected to the water around the island). One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

**Example:**

[[0,1,0,0],

[1,1,1,0],

[0,1,0,0],

[1,1,0,0]]

Answer: 16

Explanation: The perimeter is the 16 yellow stripes in the image below:



每出现一个1，就有四条边，检查是否上边和左边有1，如果有就减少了两条边

1. **public** **class** Solution {
2. **public** **int** islandPerimeter(**int**[][] grid) {
3. **int** row = grid.length;
4. **int** col = grid[0].length;
5. **int** result = 0;
6. **for** (**int** i=0;i<row;i++) {
7. **for** (**int** j=0;j<col;j++) {
8. **if** (grid[i][j]==1) {
9. result +=4;
10. **if** (i>0&&grid[i-1][j]==1) result-=2;
11. **if** (j>0&&grid[i][j-1]==1) result-=2;
12. }
13. }
14. }
15. **return** result;
16. }
17. }

475. Heaters

Winter is coming! Your first job during the contest is to design a standard heater with fixed warm radius to warm all the houses.

Now, you are given positions of houses and heaters on a horizontal line, find out minimum radius of heaters so that all houses could be covered by those heaters.

So, your input will be the positions of houses and heaters seperately, and your expected output will be the minimum radius standard of heaters.

**Note:**

1. Numbers of houses and heaters you are given are non-negative and will not exceed 25000.
2. Positions of houses and heaters you are given are non-negative and will not exceed 10^9.
3. As long as a house is in the heaters' warm radius range, it can be warmed.
4. All the heaters follow your radius standard and the warm radius will the same.

**Example 1:**

**Input:** [1,2,3],[2]

**Output:** 1

**Explanation:** The only heater was placed in the position 2, and if we use the radius 1 standard, then all the houses can be warmed.

**Example 2:**

**Input:** [1,2,3,4],[1,4]

**Output:** 1

**Explanation:** The two heater was placed in the position 1 and 4. We need to use radius 1 standard, then all the houses can be warmed.

双指针方法，对每一个house而言要找到离house最近的heater，每次都把最大半径找到，查找也可以用二分法

1. **public** **class** Solution {
2. **public** **int** findRadius(**int**[] houses, **int**[] heaters) {
3. Arrays.sort(houses);
4. Arrays.sort(heaters);
5. **int** ans=0;
6. **for** (**int** i=0,j=0;i<houses.length;i++) {
7. **while**(j<heaters.length-1&&Math.abs(heaters[j+1]- houses[i])<=Math.abs(heaters[j]-houses[i])) {
8. j++;
9. }
10. ans = Math.max(ans,Math.abs(heaters[j]-houses[i]));
11. }
12. **return** ans;
13. }
14. }

476. Number Complement

Given a positive integer, output its complement number. The complement strategy is to flip the bits of its binary representation.

**Note:**

1. The given integer is guaranteed to fit within the range of a 32-bit signed integer.
2. You could assume no leading zero bit in the integer’s binary representation.

**Example 1:**

**Input:** 5

**Output:** 2

**Explanation:** The binary representation of 5 is 101 (no leading zero bits), and its complement is 010. So you need to output 2.

**Example 2:**

**Input:** 1

**Output:** 0

**Explanation:** The binary representation of 1 is 1 (no leading zero bits), and its complement is 0. So you need to output 0

位运算

1. **public** **class** Solution {
2. **public** **int** findComplement(**int** num) {
3. **int** count=0;
4. **int** n = num;
5. **while**(n>0) {
6. count++;
7. n = n >>1;
8. }
9. **return** (**int**)(Math.pow(2,count) -1 - num);
10. }
11. }

479. Largest Palindrome Product

Find the largest palindrome made from the product of two n-digit numbers.

Since the result could be very large, you should return the largest palindrome mod 1337.

**Example:**

Input: 2

Output: 987

Explanation: 99 x 91 = 9009, 9009 % 1337 = 987

**Note:**

The range of n is [1,8].

当n=1时最大回文数是9，其他情况回文数都是偶位整数，可以利用字符串反转得到所有回文数，构建回文数后检查该数是否可用两数相乘得到，除法运算时若得到大于n位数的最大值，跳出此次循环，寻找下一个回文数。可用StringBuilder类reverse()方法直接反转，从最大数开始寻找可减少时间复杂度。

1. **class** Solution {
2. **public** **int** largestPalindrome(**int** n) {
3. **if** (n==1) **return** 9;
4. **int** ub = (**int**)(Math.pow(10,n)-1);
5. **int** lb = ub/10;
6. **for** (**int** i=ub;i>lb;i--) {
7. **long** tem = createPalin(i);
8. **for** (**int** j=ub;j>lb;j--) {
9. **if** (tem/j>ub) {
10. **break**;
11. }
12. **if** (tem%j==0) {
13. **return** (**int**)(tem%1337);
14. }
15. }
16. }
17. **return** -1;
18. }
19. **public** **long** createPalin(**int** num) {
20. String res = num + **new** StringBuilder().append(num).reverse().toString();
21. **return** Long.parseLong(res);
22. }
23. }

485. Max Consecutive Ones

Given a binary array, find the maximum number of consecutive 1s in this array.

**Example 1:**

**Input:** [1,1,0,1,1,1]

**Output:** 3

**Explanation:** The first two digits or the last three digits are consecutive 1s.

The maximum number of consecutive 1s is 3.

**Note:**

* The input array will only contain 0 and 1.
* The length of input array is a positive integer and will not exceed 10,000

1. **public** **class** Solution {
2. **public** **int** findMaxConsecutiveOnes(**int**[] nums) {
3. **int** count = 0;
4. **int** max = 0;
5. **for** (**int** i=0;i<nums.length;i++) {
6. **if** (nums[i]==1) {
7. count++;
8. } **else** {
9. max = Math.max(max,count);
10. count  = 0;
11. }
12. }
13. **return** Math.max(max,count);
14. }
15. }

492. Construct the Rectangle

For a web developer, it is very important to know how to design a web page's size. So, given a specific rectangular web page’s area, your job by now is to design a rectangular web page, whose length L and width W satisfy the following requirements:

1. The area of the rectangular web page you designed must equal to the given target area.

2. The width W should not be larger than the length L, which means L >= W.

3. The difference between length L and width W should be as small as possible.

You need to output the length L and the width W of the web page you designed in sequence.

**Example:**

**Input:** 4

**Output:** [2, 2]

**Explanation:** The target area is 4, and all the possible ways to construct it are [1,4], [2,2], [4,1].

But according to requirement 2, [1,4] is illegal; according to requirement 3, [4,1] is not optimal compared to [2,2]. So the length L is 2, and the width W is 2.

**Note:**

1. The given area won't exceed 10,000,000 and is a positive integer
2. The web page's width and length you designed must be positive integers.
3. **public** **class** Solution {
4. **public** **int**[] constructRectangle(**int** area) {
5. **int**[] res=**new** **int**[2];
6. **int** val = (**int**)Math.sqrt(area);
7. **while**(area%val!=0) {
8. val--;
9. }
10. res[1] = val;
11. res[0] = area/val;
12. **return** res;
13. }
14. }

496. Next Greater Element I

You are given two arrays **(without duplicates)** nums1 and nums2 where nums1’s elements are subset of nums2. Find all the next greater numbers for nums1's elements in the corresponding places of nums2.

The Next Greater Number of a number **x** in nums1 is the first greater number to its right in nums2. If it does not exist, output -1 for this number.

**Example 1:**

**Input:** **nums1** = [4,1,2], **nums2** = [1,3,4,2].

**Output:** [-1,3,-1]

**Explanation:**

For number 4 in the first array, you cannot find the next greater number for it in the second array, so output -1.

For number 1 in the first array, the next greater number for it in the second array is 3.

For number 2 in the first array, there is no next greater number for it in the second array, so output -1.

**Example 2:**

**Input:** **nums1** = [2,4], **nums2** = [1,2,3,4].

**Output:** [3,-1]

**Explanation:**

For number 2 in the first array, the next greater number for it in the second array is 3.

For number 4 in the first array, there is no next greater number for it in the second array, so output -1.

**Note:**

1. All elements in nums1 and nums2 are unique.
2. The length of both nums1 and nums2 would not exceed 1000.

两种方法

1. 时间复杂度O(N^M)，常规解法，对每一个nums1值找到nums2中相等值位置，再从后开始检查更大值
2. **public** **class** Solution {
3. **public** **int**[] nextGreaterElement(**int**[] findNums, **int**[] nums) {
4. **int**[] res = **new** **int**[findNums.length];
5. **for** (**int** i=0;i<findNums.length;i++) {
6. **int** j=0;
7. **while**(nums[j]!=findNums[i]) j++;
8. **for** (j=j+1;j<nums.length;j++) {
9. **if** (nums[j] > findNums[i]) {
10. res[i] = nums[j];
11. **break**;
12. }
13. }
14. **if** (j==nums.length) res[i]=-1;
15. }
16. **return** res;
17. }
18. }

2. 时间复杂度O(N),空间复杂度O(N)，用一个map记录num2每一个数与对应的下一个更大值的key-value映射，stack可以用来找更大值，遍历num1检查map中是否有对应值

1. **public** **class** Solution {
2. **public** **int**[] nextGreaterElement(**int**[] findNums, **int**[] nums) {
3. **int**[] ans = **new** **int**[findNums.length];
4. Map<Integer,Integer> map = **new** HashMap<Integer,Integer>();
5. Stack<Integer> stack = **new** Stack<Integer>();
6. **for** (**int** i=0;i<nums.length;i++) {
7. **while**(!stack.isEmpty()&&stack.peek()<nums[i]) {
8. map.put(stack.pop(),nums[i]);
9. }
10. stack.push(nums[i]);

13. }
14. **for** (**int** j=0;j<findNums.length;j++) {
15. ans[j]=map.getOrDefault(findNums[j],-1);
16. }
17. **return** ans;
18. }
19. }

500. Keyboard Row

Given a List of words, return the words that can be typed using letters of **alphabet** on only one row's of American keyboard like the image below.

**Example 1:**

**Input:** ["Hello", "Alaska", "Dad", "Peace"]

**Output:** ["Alaska", "Dad"]

**Note:**

1. You may use one character in the keyboard more than once.
2. You may assume the input string will only contain letters of alphabet.

看了答案发现这种解法最简洁

注意：List和Array转换时，要想使List转array返回指定类型的数组，toArray()方法要加上一个数组类型的参数，

1. **public** **class** Solution {
2. **public** String[] findWords(String[] words) {
3. String[] strs = {"QWERTYUIOP","ASDFGHJKL","ZXCVBNM"};
4. Map<Character, Integer> map = **new** HashMap<Character, Integer>();
5. **for**(**int** i = 0; i<strs.length; i++){
6. **for**(**char** c: strs[i].toCharArray()){
7. map.put(c, i);//put <char, rowIndex> pair into the map
8. }
9. }
10. List<String> ans = **new** ArrayList<String>();
11. **for**(String word: words){
12. **if**(word.equals("")) **continue**;
13. **int** index = map.get(word.toUpperCase().charAt(0));
14. **for**(**char** c: word.toUpperCase().toCharArray()){
15. **if**(map.get(c)!=index){
16. index = -1; //don't need a boolean flag.
17. **break**;
18. }
19. }
20. **if**(index!=-1) ans.add(word);//if index != -1, this is a valid string
21. }
22. **return** ans.toArray(**new** String[0]);
23. }
24. }

501. Find Mode in Binary Search Tree

Given a binary search tree (BST) with duplicates, find all the [mode(s)](https://en.wikipedia.org/wiki/Mode_(statistics)) (the most frequently occurred element) in the given BST.

Assume a BST is defined as follows:

* The left subtree of a node contains only nodes with keys **less than or equal to** the node's key.
* The right subtree of a node contains only nodes with keys **greater than or equal to** the node's key.
* Both the left and right subtrees must also be binary search trees.

For example:  
Given BST [1,null,2,2],

1

\

2

/

2

return [2].

**Note:** If a tree has more than one mode, you can return them in any order.

**Follow up:** Could you do that without using any extra space? (Assume that the implicit stack space incurred due to recursion does not count).

这道题考察DFS和中序遍历（因为中序是按大小顺序查找），注意help function的使用和全局变量的设置

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. Integer prev = **null**;
12. **int** max = 0;
13. **int** count = 1;
14. **public** **int**[] findMode(TreeNode root) {
15. **if** (root==**null**) **return** **new** **int**[0];
16. List<Integer> list = **new** ArrayList<Integer>();
17. inorder(root,list);
18. **int**[] ans = **new** **int**[list.size()];
20. **for** (**int** i=0;i<list.size();i++) {
21. ans[i]=list.get(i);
22. }
23. **return** ans;
24. }
25. **public** **void** inorder(TreeNode root, List<Integer> list) {
26. **if** (root.left!=**null**) inorder(root.left,list);
27. **if** (prev!=**null**) {
28. **if** (prev==root.val) {
29. count++;
30. } **else** {
31. count=1;
32. }
33. }
34. **if** (count>max) {
35. max = count;
36. list.clear();
37. list.add(root.val);
38. } **else** **if** (count==max) {
39. list.add(root.val);
40. }
41. prev = root.val;
42. **if** (root.right!=**null**) inorder(root.right,list);
43. }
44. }

504. Base 7

Given an integer, return its base 7 string representation.

**Example 1:**

**Input:** 100

**Output:** "202"

**Example 2:**

**Input:** -7

**Output:** "-10"

**Note:** The input will be in range of [-1e7, 1e7].

1. **public** **class** Solution {
2. **public** String convertToBase7(**int** num) {
3. StringBuilder ans = **new** StringBuilder();
4. **if** (num==0) **return** "0";
5. **boolean** pos = **true**;
6. **if** (num<0) {
7. num  = -num;
8. pos = **false**;
9. }
10. **while** (num>0) {
11. ans.append(String.valueOf(num%7));
12. num /= 7;
13. }
14. **return** pos==**true**?ans.reverse().toString():ans.append("-").reverse().toString();
15. }
16. }

506. Relative Ranks

Given scores of **N** athletes, find their relative ranks and the people with the top three highest scores, who will be awarded medals: "Gold Medal", "Silver Medal" and "Bronze Medal".

**Example 1:**

**Input:** [5, 4, 3, 2, 1]

**Output:** ["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]

**Explanation:** The first three athletes got the top three highest scores, so they got "Gold Medal", "Silver Medal" and "Bronze Medal".   
For the left two athletes, you just need to output their relative ranks according to their scores.

**Note:**

1. N is a positive integer and won't exceed 10,000.
2. All the scores of athletes are guaranteed to be unique.

注意复制数组的两种方法，clone() 和System.arraycopy()

1. **public** **class** Solution {
2. **public** String[] findRelativeRanks(**int**[] nums) {
3. String[] ans = **new** String[nums.length];
4. Map<Integer,Integer> map = **new** HashMap<Integer,Integer>();
5. **int**[] newNums = **new** **int**[nums.length];
6. System.arraycopy(nums,0,newNums,0,nums.length);
7. Arrays.sort(newNums);
8. **int** rank=1;
9. **for** (**int** i=newNums.length-1;i>=0;i--) {
10. map.put(newNums[i],rank);
11. rank++;
13. }
14. **for** (**int** i=0;i<nums.length;i++) {
15. **int** val=map.get(nums[i]);
16. **if** (val==1) {
17. ans[i]="Gold Medal";
18. } **else** **if** (val==2) {
19. ans[i] = "Silver Medal";
20. } **else** **if**(val==3) {
21. ans[i]="Bronze Medal";
22. } **else** {
23. ans[i]=String.valueOf(val);
24. }
25. }
26. **return** ans;
27. }
28. }

507. Perfect Number

We define the Perfect Number is a **positive** integer that is equal to the sum of all its **positive** divisors except itself.

Now, given an **integer** n, write a function that returns true when it is a perfect number and false when it is not.

**Example:**

**Input:** 28

**Output:** True

**Explanation:** 28 = 1 + 2 + 4 + 7 + 14

**Note:** The input number **n** will not exceed 100,000,000. (1e8)

1. **public** **class** Solution {
2. **public** **boolean** checkPerfectNumber(**int** num) {
3. **if** (num<=1) **return** **false**;
4. **int** sum  = 1;
5. **int** n =2;
6. **while**(n<Math.sqrt(num)) {
7. **if** (num%n==0) {
8. sum += n;
9. sum += num/n;
10. }
11. n++;
12. }
13. **if** (n==Math.sqrt(num)) {
14. sum+=n;
15. }
16. **return** sum==num;
17. }
18. }

520. Detect Capital

Given a word, you need to judge whether the usage of capitals in it is right or not.

We define the usage of capitals in a word to be right when one of the following cases holds:

1. All letters in this word are capitals, like "USA".
2. All letters in this word are not capitals, like "leetcode".
3. Only the first letter in this word is capital if it has more than one letter, like "Google".

Otherwise, we define that this word doesn't use capitals in a right way.

**Example 1:**

**Input:** "USA"

**Output:** True

**Example 2:**

**Input:** "FlaG"

**Output:** False

**Note:** The input will be a non-empty word consisting of uppercase and lowercase latin letters.

三种情况符合条件，第一种长度为1，第二种全是大写字母，第三种不管第一个字母大小写，后面全是小写，按照这三种思路分别用字符串方法处理

1. **public** **class** Solution {
2. **public** **boolean** detectCapitalUse(String word) {
3. **if** (word.length()==1) **return** **true**;
4. **if** (word.toUpperCase().equals(word)) **return** **true**;
5. **if** (word.substring(1).toLowerCase().equals(word.substring(1))) **return** **true**;
6. **return** **false**;
7. }
8. }

正则表达式方法：

1. **public** **class** Solution {
2. **public** **boolean** detectCapitalUse(String word) {
3. **return** word.matches("[A-Z]+|[a-z]+|[A-Z][a-z]+");
4. }
5. }

521. Longest Uncommon Subsequence I

Given a group of two strings, you need to find the longest uncommon subsequence of this group of two strings. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be **any**subsequence of the other strings.

A **subsequence** is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.

The input will be two strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1.

**Example 1:**

**Input:** "aba", "cdc"

**Output:** 3

**Explanation:** The longest uncommon subsequence is "aba" (or "cdc"),   
because "aba" is a subsequence of "aba",   
but not a subsequence of any other strings in the group of two strings.

**Note:**

1. Both strings' lengths will not exceed 100.
2. Only letters from a ~ z will appear in input strings.

这道题很简单也很tricky，要理解题目的意思。

1. **public** **class** Solution {
2. **public** **int** findLUSlength(String a, String b) {
3. **return** a.equals(b)?-1:Math.max(a.length(),b.length());
4. }
5. }

530. Minimum Absolute Difference in BST

Given a binary search tree with non-negative values, find the minimum [absolute difference](https://en.wikipedia.org/wiki/Absolute_difference) between values of any two nodes.

**Example:**

**Input:**

1

\

3

/

2

**Output:**

1

**Explanation:**

The minimum absolute difference is 1, which is the difference between 2 and 1 (or between 2 and 3).

**Note:** There are at least two nodes in this BST.

这道题关键要知道用的是中序遍历，因为中序是排序的数组，只要求两个相邻数中差值最小的即可

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **int** min = Integer.MAX\_VALUE;
12. TreeNode prev = **null**;
13. **public** **int** getMinimumDifference(TreeNode root) {
14. inorder(root);
15. **return** min;
17. }
18. **public** **void** inorder(TreeNode root) {
19. **if** (root==**null**) **return** ;
20. inorder(root.left);
21. **if** (prev!=**null**) {
22. min = Math.min(min,root.val-prev.val);
23. }
24. prev = root;
25. inorder(root.right);
26. }
27. }

532. K-diff Pairs in an Array

Given an array of integers and an integer **k**, you need to find the number of **unique** k-diff pairs in the array. Here a **k-diff** pair is defined as an integer pair (i, j), where **i** and **j** are both numbers in the array and their [absolute difference](https://en.wikipedia.org/wiki/Absolute_difference) is **k**.

**Example 1:**

**Input:** [3, 1, 4, 1, 5], k = 2

**Output:** 2

**Explanation:** There are two 2-diff pairs in the array, (1, 3) and (3, 5).  
Although we have two 1s in the input, we should only return the number of **unique** pairs.

**Example 2:**

**Input:**[1, 2, 3, 4, 5], k = 1

**Output:** 4

**Explanation:** There are four 1-diff pairs in the array, (1, 2), (2, 3), (3, 4) and (4, 5).

**Example 3:**

**Input:** [1, 3, 1, 5, 4], k = 0

**Output:** 1

**Explanation:** There is one 0-diff pair in the array, (1, 1).

**Note:**

1. The pairs (i, j) and (j, i) count as the same pair.
2. The length of the array won't exceed 10,000.
3. All the integers in the given input belong to the range: [-1e7, 1e7].

空间换时间，用map记录每个数字出现次数，遍历map检查每个key的value加上k是否存在于map中，如果k是0的话，每个数至少出现两次，注意对Map.Entry<T,T>的应用

1. **public** **class** Solution {
2. **public** **int** findPairs(**int**[] nums, **int** k) {
3. **if** (k<0) **return** 0;
4. **int** ans = 0;
5. Map<Integer,Integer> map = **new** HashMap<Integer,Integer>();
6. **for** (**int** i=0;i<nums.length;i++) {
7. map.put(nums[i],map.getOrDefault(nums[i],0)+1);
8. }
9. **for** (Map.Entry<Integer,Integer> set:map.entrySet()) {
10. **if** (k==0) {
11. **if** (set.getValue()>=2) ans++;
12. } **else** {
13. **if** (map.containsKey(set.getKey()+k)) ans++;
14. }
15. }
16. **return** ans;
17. }
18. }

还有一种方法是双指针遍历，先排序，设置头指针和尾指针，如果头指针所指数加k小于尾指针所指，增加头指针，如果头指针所指数加k大于尾指针所指，增加尾指针，直至找到两数之差等于k，结果加1，记得要去掉重复数字

1. **public** **class** Solution {
2. **public** **int** findPairs(**int**[] nums, **int** k) {
3. Arrays.sort(nums);
5. **int** start = 0, end = 1, result = 0;
6. **while** (start < nums.length && end < nums.length) {
7. **if** (start == end || nums[start] + k > nums[end]) {
8. end++;
9. } **else** **if** (nums[start] + k < nums[end]) {
10. start++;
11. } **else** {
12. start++;
13. result++;
14. **while** (start < nums.length && nums[start] == nums[start - 1]) start++;
15. end = Math.max(end + 1, start + 1);
16. }
17. }
18. **return** result;
19. }
20. }

538. Convert BST to Greater Tree

Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

**Example:**

**Input:** The root of a Binary Search Tree like this:

5

/ \

2 13

**Output:** The root of a Greater Tree like this:

18

/ \

20 13

这道题先遍历右子数，再根节点求和，最后左子树

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public class** Solution {
11. **int** sum = 0;
12. **public** TreeNode convertBST(TreeNode root) {
13. reverse(root);
14. **return** root;
15. }
16. **public** **void** reverse(TreeNode root) {
17. **if** (root==**null**) **return**;
18. reverse(root.right);
19. root.val += sum;
20. sum = root.val;
21. reverse(root.left);
22. }
23. }

541. Reverse String II

Given a string and an integer k, you need to reverse the first k characters for every 2k characters counting from the start of the string. If there are less than k characters left, reverse all of them. If there are less than 2k but greater than or equal to k characters, then reverse the first k characters and left the other as original.

**Example:**

**Input:** s = "abcdefg", k = 2

**Output:** "bacdfeg"

**Restrictions:**

1. The string consists of lower English letters only.
2. Length of the given string and k will in the range [1, 10000]

注意字符串中位置的计算

1. **public** **class** Solution {
2. **public** String reverseStr(String s, **int** k) {
3. StringBuilder res = **new** StringBuilder("");
4. **int** i;
5. **int** sl =s.length();
6. **for** (i=0;i+2\*k<sl;i=i+2\*k) {
7. res.append(**new** StringBuilder(s.substring(i,i+k)).reverse());
8. res.append(s.substring(i+k,i+2\*k));
10. }
11. **if** (sl-i<=k) {
12. res.append(**new** StringBuilder(s.substring(i,sl)).reverse());
13. } **else** {
14. res.append(**new** StringBuilder(s.substring(i,i+k)).reverse());
15. res.append(s.substring(i+k,sl));
16. }
17. **return** res.toString();
18. }
19. }

543. Diameter of Binary Tree

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.

**Example:**  
Given a binary tree

1

/ \

2 3

/ \

4 5

Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].

**Note:** The length of path between two nodes is represented by the number of edges between them.

这道题一开始的想法是左右子数最大高度＋1， 但是注意最大直径可以不包含根节点，所以DFS时对每一个节点都要计算下左右子树最大高度（经过验证，左右子数高度相加即为这个根节点下最大的直径），将最大的值保存（要设全局变量）

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **public** **class** Solution {
11. **int** max = 0;
12. **public** **int** diameterOfBinaryTree(TreeNode root) {
13. getMax(root);
15. **return** max;
16. }
17. **public** **int** getMax(TreeNode root) {
18. **if** (root == **null**) **return** 0;
19. **int** left = getMax(root.left);
20. **int** right = getMax(root.right);
21. max = Math.max(max,left+right);
22. **return** 1+ Math.max(left,right);
23. }
24. }

551. Student Attendance Record I

You are given a string representing an attendance record for a student. The record only contains the following three characters:

1. **'A'** : Absent.
2. **'L'** : Late.
3. **'P'** : Present.

A student could be rewarded if his attendance record doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.

You need to return whether the student could be rewarded according to his attendance record.

**Example 1:**

**Input:** "PPALLP"

**Output:** True

**Example 2:**

**Input:** "PPALLL"

**Output:** False

自己一开始的解法是判断连续三个字符是否都是L,这样要考虑的边界条件很多，巧妙的解法是如果前面一个是L,就继续加不然就重新开始计算L, 每次循环时都判断A和L条件，要注意这种直接遍历到底的解法

1. **public** **class** Solution {
2. **public** **boolean** checkRecord(String s) {
3. **int** countA = 0;
4. **int** countL = 0;
5. **for** (**int** i=0;i<s.length();i++) {
6. **if** (s.charAt(i)=='A') countA++;
7. **if** (s.charAt(i)=='L') {
8. **if** (i>=1&&s.charAt(i-1)=='L') {
9. countL++;
10. } **else** {
11. countL = 1;
12. }
13. }
14. **if** (countA>1||countL>2) {
15. **return** **false**;
16. }
18. }
19. **return** **true**;
20. }
21. }

557. Reverse Words in a String III

Given a string, you need to reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.

**Example 1:**

**Input:** "Let's take LeetCode contest"

**Output:** "s'teL ekat edoCteeL tsetnoc"

**Note:** In the string, each word is separated by single space and there will not be any extra space in the string.

考察String和StringBuilder的转化和使用，注意使用String.split()方法后得到string数组

1. **public** **class** Solution {
2. **public** String reverseWords(String s) {
3. StringBuilder ans = **new** StringBuilder("");
4. String[] array = s.split(" ");
5. **for**(**int** i=0;i<array.length;i++) {
6. ans.append(**new** StringBuilder(array[i]).reverse());
7. **if** (i!=array.length-1) ans.append(" ");
8. }
9. **return** ans.toString();
10. }

561. Array Partition I

Given an array of **2n** integers, your task is to group these integers into **n** pairs of integer, say (a1, b1), (a2, b2), ..., (an, bn) which makes sum of min(ai, bi) for all i from 1 to n as large as possible.

**Example 1:**

**Input:** [1,4,3,2]

**Output:** 4

**Explanation:** n is 2, and the maximum sum of pairs is 4 = min(1, 2) + min(3, 4).

**Note:**

1. **n** is a positive integer, which is in the range of [1, 10000].
2. All the integers in the array will be in the range of [-10000, 10000].

数学问题，排序后最小的两个数一组才能避免使用第二小的数

1. **class** Solution {
2. **public** **int** arrayPairSum(**int**[] nums) {
3. **int** sum = 0;
4. Arrays.sort(nums);
5. **for** (**int** i=0;i<nums.length;i=i+2) {
6. sum += nums[i];
7. }
8. **return** sum;
9. }
10. }

563. Binary Tree Tilt

Given a binary tree, return the tilt of the **whole tree**.

The tilt of a **tree node** is defined as the **absolute difference** between the sum of all left subtree node values and the sum of all right subtree node values. Null node has tilt 0.

The tilt of the **whole tree** is defined as the sum of all nodes' tilt.

**Example:**

**Input:**

1

/ \

2 3

**Output:** 1

**Explanation:**

Tilt of node 2 : 0

Tilt of node 3 : 0

Tilt of node 1 : |2-3| = 1

Tilt of binary tree : 0 + 0 + 1 = 1

**Note:**

1. The sum of node values in any subtree won't exceed the range of 32-bit integer.
2. All the tilt values won't exceed the range of 32-bit integer.

自己的解法是，每一个节点的tilt都是左右子树分别求总和后之差的绝对值，再左右节点递归

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** **int** findTilt(TreeNode root) {
12. **if** (root==**null**) **return** 0;
14. **return** findTilt(root.left)+findTilt(root.right)+Math.abs(sumNode(root.left)-sumNode(root.right));
15. }
16. **public** **int** sumNode(TreeNode root) {
17. **if** (root==**null**) **return** 0;
18. **return** sumNode(root.left)+sumNode(root.right)+root.val;
19. }
20. }

其实就是一个DFS后续遍历，只需遍历一次，记录每次遍历左右子数和之差，每次返回值为节点值之和，用一个全局变量存tilt

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **int** ans = 0;
12. **public** **int** findTilt(TreeNode root) {
13. postOrder(root);
14. **return** ans;
16. }
17. **public** **int** postOrder(TreeNode root) {
18. **if** (root==**null**) **return** 0;
19. **int** left = postOrder(root.left);
20. **int** right = postOrder(root.right);
21. ans += Math.abs(left-right);
22. **return** left+right+root.val;
23. }
24. }

566. Reshape the Matrix

In MATLAB, there is a very useful function called 'reshape', which can reshape a matrix into a new one with different size but keep its original data.

You're given a matrix represented by a two-dimensional array, and two **positive** integers **r** and **c** representing the **row** number and **column** number of the wanted reshaped matrix, respectively.

The reshaped matrix need to be filled with all the elements of the original matrix in the same **row-traversing** order as they were.

If the 'reshape' operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

**Example 1:**

**Input:**

nums =

[[1,2],

[3,4]]

r = 1, c = 4

**Output:**

[[1,2,3,4]]

**Explanation:**  
The **row-traversing** of nums is [1,2,3,4]. The new reshaped matrix is a 1 \* 4 matrix, fill it row by row by using the previous list.

**Example 2:**

**Input:**

nums =

[[1,2],

[3,4]]

r = 2, c = 4

**Output:**

[[1,2],

[3,4]]

**Explanation:**  
There is no way to reshape a 2 \* 2 matrix to a 2 \* 4 matrix. So output the original matrix.

**Note:**

1. The height and width of the given matrix is in range [1, 100].
2. The given r and c are all positive.
3. **class** Solution {
4. **public** **int**[][] matrixReshape(**int**[][] nums, **int** r, **int** c) {
5. **int** row = nums.length;
6. **int** col = nums[0].length;
7. **if** (r\*c!=row\*col) **return** nums;
8. **int**[][] ans = **new** **int**[r][c];
9. **int** x=0,y=0;
10. **for** (**int** i=0;i<r;i++) {
11. **for** (**int** j=0;j<c;j++) {
12. ans[i][j]=nums[x][y];
13. y++;
14. **if** (y==col) {
15. x++;
16. y=0;
17. }
18. }
19. }
20. **return** ans;
21. }
22. }

572. Subtree of Another Tree

Given two non-empty binary trees **s** and **t**, check whether tree **t** has exactly the same structure and node values with a subtree of **s**. A subtree of **s** is a tree consists of a node in **s** and all of this node's descendants. The tree **s** could also be considered as a subtree of itself.

**Example 1:**  
Given tree s:

3

/ \

4 5

/ \

1 2

Given tree t:

4

/ \

1 2

Return **true**, because t has the same structure and node values with a subtree of s.

**Example 2:**  
Given tree s:

3

/ \

4 5

/ \

1 2

/

0

Given tree t:

4

/ \

1 2

Return **false**.

树的应用，先判断根节点是否相同树，再依次左子树右子数判断

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** **boolean** isSubtree(TreeNode s, TreeNode t) {
12. **if** (s==**null**) **return** **false**;
14. **if** (isSameTree(s,t)) **return** **true**;
15. **return** (isSubtree(s.left,t)||isSubtree(s.right,t));
16. }
17. **public** **boolean** isSameTree(TreeNode t1,TreeNode t2) {
18. **if** (t1==**null**&&t2==**null**) **return** **true**;
19. **if** (t1==**null**||t2==**null**) **return** **false**;
20. **if** (t1.val!=t2.val) **return** **false**;
21. **return** isSameTree(t1.left,t2.left)&&isSameTree(t1.right,t2.right);
22. }
23. }

575. Distribute Candies

Given an integer array with **even** length, where different numbers in this array represent different **kinds** of candies. Each number means one candy of the corresponding kind. You need to distribute these candies **equally** in number to brother and sister. Return the maximum number of **kinds** of candies the sister could gain.

**Example 1:**

**Input:** candies = [1,1,2,2,3,3]

**Output:** 3

**Explanation:**

There are three different kinds of candies (1, 2 and 3), and two candies for each kind.

Optimal distribution: The sister has candies [1,2,3] and the brother has candies [1,2,3], too.

The sister has three different kinds of candies.

**Example 2:**

**Input:** candies = [1,1,2,3]

**Output:** 2

**Explanation:** For example, the sister has candies [2,3] and the brother has candies [1,1].

The sister has two different kinds of candies, the brother has only one kind of candies.

**Note:**

1. The length of the given array is in range [2, 10,000], and will be even.
2. The number in given array is in range [-100,000, 100,000].
3. **class** Solution {
4. **public** **int** distributeCandies(**int**[] candies) {
5. Set<Integer> set = **new** HashSet<Integer>();
6. **for** (**int** i=0;i<candies.length;i++) {
7. set.add(candies[i]);
8. }
9. **return** set.size()>=candies.length/2?candies.length/2:set.size();
10. }
11. }

581. Shortest Unsorted Continuous Subarray

Given an integer array, you need to find one **continuous subarray** that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order, too.

You need to find the **shortest** such subarray and output its length.

**Example 1:**

**Input:** [2, 6, 4, 8, 10, 9, 15]

**Output:** 5

**Explanation:** You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole array sorted in ascending order.

**Note:**

1. Then length of the input array is in range [1, 10,000].
2. The input array may contain duplicates, so ascending order here means **<=**.

自己的解法是常规解法，时间复杂度O(n\*logn), 空间复杂度O(n)

1. **class** Solution {
2. **public** **int** findUnsortedSubarray(**int**[] nums) {
3. **int**[] sortArray = **new** **int**[nums.length];
4. System.arraycopy(nums,0,sortArray,0,nums.length);
5. Arrays.sort(sortArray);
6. **int** i=0,j=nums.length-1;
7. **while**(i<nums.length&&nums[i]==sortArray[i]) i++;
8. **while**(j>=0&&nums[j]==sortArray[j]) j--;
9. **if** (i<j) {
10. **return** j-i+1;
11. } **else** {
12. **return** 0;
13. }
14. }
15. }

答案有一种很巧的解法，时间复杂度O(n),空间复杂度O(1)

1. **class** Solution {
2. **public** **int** findUnsortedSubarray(**int**[] nums) {
3. **int** start=-1,end=-1,max = nums[0],min=nums[nums.length-1];
4. **for** (**int** i=0;i<nums.length;i++){
5. max = Math.max(max,nums[i]);
6. min = Math.min(min,nums[nums.length-i-1]);
7. **if** (nums[i]<max) end=i;
8. **if** (nums[nums.length-1-i]>min) start = nums.length-i-1;
9. }
10. **if** (start==-1) **return** 0;
11. **return** end-start+1;
12. }
13. }

674. Longest Continuous Increasing Subsequence

Given an unsorted array of integers, find the length of longest continuous increasing subsequence (subarray).

**Example 1:**

**Input:** [1,3,5,4,7]

**Output:** 3

**Explanation:** The longest continuous increasing subsequence is [1,3,5], its length is 3.

Even though [1,3,5,7] is also an increasing subsequence, it's not a continuous one where 5 and 7 are separated by 4.

**Example 2:**

**Input:** [2,2,2,2,2]

**Output:** 1

**Explanation:** The longest continuous increasing subsequence is [2], its length is 1.

1. **class** Solution {
2. **public** **int** findLengthOfLCIS(**int**[] nums) {
3. **if** (nums.length <= 1) **return** nums.length;
4. **int** left = 0, right = 1;
5. **int** max = 0, count = 1;
6. **while** (right < nums.length) {
7. **if** (nums[left] >= nums[right]) {
8. max = Math.max(count,max);
9. count = 1;
10. } **else** {
11. count++;
12. }
13. left++;
14. right++;
15. }
16. max = Math.max(count,max);
17. **return** max;
18. }
19. }

# **Medium**

2. Add Two Numbers

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order** and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example**

**Input:** (2 -> 4 -> 3) + (5 -> 6 -> 4)

**Output:** 7 -> 0 -> 8

**Explanation:** 342 + 465 = 807.

注意“假头”的应用！

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** ListNode addTwoNumbers(ListNode l1, ListNode l2) {
11. ListNode ans = **new** ListNode(0);
12. ListNode res= ans;
13. **int** carry = 0;
14. **while**(l1!=**null**&&l2!=**null**) {
15. **int** val = (l1.val+l2.val+carry)%10;
16. carry = (l1.val+l2.val+carry>=10?1:0);
17. ans.next = **new** ListNode(val);
18. l1 = l1.next;
19. l2 = l2.next;
20. ans = ans.next;
21. }
22. ListNode l = (l1==**null**?l2:l1);
23. **while**(l!=**null**) {
24. **int** val = (l.val+carry)%10;
25. carry = (l.val+carry>=10?1:0);
26. ans.next = **new** ListNode(val);
27. l = l.next;
28. ans = ans.next;
29. }
30. **if** (carry==1) {
31. ans.next = **new** ListNode(1);
32. }
33. **return** res.next;
34. }
35. }

5. Longest Palindromic Substring

Given a string **s**, find the longest palindromic substring in **s**. You may assume that the maximum length of **s** is 1000.

**Example:**

**Input:** "babad"

**Output:** "bab"

**Note:** "aba" is also a valid answer.

**Example:**

**Input:** "cbbd"

**Output:** "bb"

第一种解法：中心扩展法，遍历每个字母时依次以其为中心，用两指针判断前面一位后面一位是否相等，相等就继续扩展，注意要考虑回文长度为奇数和偶数的两种情况，时间复杂度O(n^2)

1. **public** **class** Solution {
2. **int** low=0,high=0,max=0;
3. **public** String longestPalindrome(String s) {
4. **for** (**int** i=0;i<s.length();i++) {
5. findPalindrome(s,i,i);
6. findPalindrome(s,i,i+1);
7. }
8. **return** s.substring(low,high+1);
10. }
11. **public** **void** findPalindrome(String s,**int** start,**int** end) {
12. **while**(start>=0&&end<s.length()&&s.charAt(start)==s.charAt(end)) {
13. start--;
14. end++;
15. }
16. **if** (max<end-start-1) {
17. max = end-start-1;
18. low = start+1;
19. high = end-1;
20. }
21. }
22. }

动态规划的解法：

dp[i][j] = s.charAt(i)==s.charAt(j)&&(dp[i+1][j-1]

1. **public** **class** Solution {
2. **public** String longestPalindrome(String s) {
3. **if**(s.length()<=1){
4. **return** s;
5. }
6. **int** start=0, end=0;
7. **int** maxLen = 0;
8. **boolean**[][] plain = **new** **boolean**[s.length()][s.length()];
9. **for**(**int** i=s.length()-1;i>=0;i--){
10. **for**(**int** j=i;j<s.length();j++){
11. **if**(s.charAt(i)==s.charAt(j)&&(j-i<=2||plain[i+1][j-1])){
12. plain[i][j]=**true**;
13. **if**(maxLen<j-i+1){
14. maxLen = j-i+1;
15. start = i;
16. end = j;
17. }
18. }
19. }
20. }
21. **return** s.substring(start, end+1);
22. }
23. }

3. Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.

**Examples:**

Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a **substring**, "pwke" is a *subsequence*and not a substring.

可以用Set或者Map, 思想都是用分别用i和j两个指针分别指向当前遍历的和重复字母处。字符放入set中遇到新的就放入set更新结果，如果重复就从左边开始删字符直到删除重复为止

Set:

1. **public** **class** Solution {
2. **public** **int** lengthOfLongestSubstring(String s) {
3. **int** max = 0;
4. Set<Character> set = **new** HashSet<Character>();
5. **for** (**int** i=0,j=0;i<s.length();) {
6. **if** (!set.contains(s.charAt(i))) {
7. set.add(s.charAt(i));
8. i++;
9. max = Math.max(max,set.size());
10. } **else** {
11. set.remove(s.charAt(j++));
12. }
14. }
15. **return** max;
17. }
18. }

Map: 记录每个字符出现位置，j,i 分别指向头尾，如果出现重复那么跨过该重复字符开始下一次搜索

1. **public** **class** Solution {
2. **public** **int** lengthOfLongestSubstring(String s) {
3. **if** (s==**null**) **return** 0;
4. **int** max =0;
5. Map<Character,Integer> map = **new** HashMap<Character,Integer>();
6. **for** (**int** i=0,j=0;i<s.length();i++) {
7. **if** (map.containsKey(s.charAt(i))) {
8. j = Math.max(j,map.get(s.charAt(i))+1);
9. }
10. map.put(s.charAt(i),i);
11. max = Math.max(max,i-j+1);
12. }
13. **return** max;
14. }
15. }

6. ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

首先要搞清楚什么是ZigZag,见下图



The method is, to make every row a single StringBuilder and connect them together to be one final result String. Each line StringBuilder consist of two parts: Vertical row and oblique row. Calculate them respectively.

One point needs to notice is how to initial StringBuilder array!

1. **public** **class** Solution {
2. **public** String convert(String s, **int** numRows) {
3. **if** (s == **null** || s.length() == 1) {
4. **return** s;
5. }
6. **char**[] c = s.toCharArray();
7. **int** len = c.length;
8. StringBuilder[] sb = **new** StringBuilder[numRows];
9. **for** (**int** i = 0;i<sb.length;i++) {
10. sb[i] = **new** StringBuilder();   **// 注意这里！！**
11. }
12. **int** indexC = 0;
13. **while**(indexC<len) {
14. **for** (**int** i=0;i<numRows&&indexC<len;i++) {
15. sb[i].append(c[indexC++]);
16. }
17. **for** (**int** i=numRows-2;i>=1&&indexC<len;i--) {
18. sb[i].append(c[indexC++]);
19. }
20. }
21. **for** (**int** i=1;i<sb.length;i++) {
22. t
23. }
24. **return** sb[0].toString();
25. }
26. }

8. String to Integer (atoi)

Implement atoi to convert a string to an integer.

**Hint:** Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

**Notes:** It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

**Update (2015-02-10):**  
The signature of the C++ function had been updated. If you still see your function signature accepts a const char \* argument, please click the reload button  to reset your code definition.

[spoilers alert... click to show requirements for atoi.](https://leetcode.com/problems/string-to-integer-atoi/description/)

**Requirements for atoi:**

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT\_MAX (2147483647) or INT\_MIN (-2147483648) is returned.

这道题要考虑四种情况，即空格开始，正负号，非数字字符，整型越界

1. **class** Solution {
2. **public** **int** myAtoi(String str) {
3. **if** (str == **null** || str.length() == 0) **return** 0;
4. str = str.trim();
5. **int** sign=1;
6. **int** start = 0;
7. **int** len = str.length();
8. **long** sum = 0;
9. **if** (str.charAt(0)=='+') {
10. sign = 1;
11. start++;
12. } **else** **if**(str.charAt(0)=='-') {
13. sign = -1;
14. start++;
15. }
16. **for** (**int** i=start;i < len;i++) {
17. **if** (str.charAt(i) < '0' || str.charAt(i) > '9') {
18. **return** (**int**)sum\*sign;
19. }
20. sum = sum\*10 + str.charAt(i)-'0';
21. **if** (sign == 1 && sum > Integer.MAX\_VALUE) {
22. **return** Integer.MAX\_VALUE;
23. }
24. **if** (sign == -1 && -sum < Integer.MIN\_VALUE) {
25. **return** Integer.MIN\_VALUE;
26. }
28. }
29. **return** (**int**) sum\*sign;
30. }
31. }

11. Container With Most Water

Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and *n* is at least 2.

Brute force may result in time limit exceed, this problem should use greedy algorithm and two pinters, which point to head and tail respectively. Compare the max value and the area surround by current two lines. Than move the less value pointer into inner array direction.

1. **class** Solution {
2. **public** **int** maxArea(**int**[] height) {
3. **int** max = 0;
4. **int** left = 0, right = height.length - 1;
5. **while**(left < right) {
7. **if** (Math.min(height[right],height[left]) \* (right - left) > max) {
8. max = Math.min(height[right],height[left]) \* (right - left);
9. }
10. **if** (height[right] >= height[left]) {
11. left++;
12. } **else** {
13. right--;
14. }
15. }
16. **return** max;
18. }
19. }

215. Kth Largest Element in an Array

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example,  
Given [3,2,1,5,6,4] and k = 2, return 5.

**Note:**  
You may assume k is always valid, 1 ≤ k ≤ array's length.

Based on Quick Sort, when an array is in descending order, only care about if the position of partition is Kth.If larger than K-1 , meaning the Kth element is in the first half part(Notice it’s in desceding order), thus next step is to get the partition of the first half part. To in this priciple until the partition value is exactly the same as K-1.

1. **class** Solution {
2. **public** **int** findKthLargest(**int**[] nums, **int** k) {
4. **int** low = 0, high = nums.length - 1;
5. **int** p = partition(nums,low,high);
6. **while**(p != k - 1) {
7. **if** (p > k - 1) {
8. high = p - 1;
9. } **else** {
10. low =  p + 1;
11. }
12. p = partition(nums, low,high);
13. }
14. **return** nums[p];
15. }
16. **private** **int** partition(**int**[] arr, **int** low, **int** high) {
18. **int** wall = low-1;
19. **int** pivot = arr[high];
20. **for** (**int** i = low;i < high;i++) {
21. **if** (arr[i] >= pivot) {
22. wall++;
23. swap(arr,i,wall);
24. }
25. }
26. swap(arr,high,wall+1);
27. **return** wall+1;
29. }
30. **public** **void** swap(**int**[] arr , **int** i, **int** j) {
31. **int** temp = arr[i];
32. arr[i] = arr[j];
33. arr[j] = temp;
34. }
35. }

12. Integer to Roman

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

1. **class** Solution {
2. **public** String intToRoman(**int** num) {
3. **int**[] values = {1000,900,500,400,100,90,50,40,10,9,5,4,1};
4. String[] romans = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
5. StringBuilder ans = **new** StringBuilder("");
6. **int** i=0;
7. **while** (num >0) {
8. **while** ( num >= values[i]) {
9. num -= values[i];
10. ans.append(romans[i]);
11. }
12. i++;
13. }
14. **return** ans.toString();
15. }
16. }

15. 3Sum

Given an array *S* of *n* integers, are there elements *a*, *b*, *c* in *S* such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

**Note:** The solution set must not contain duplicate triplets.

For example, given array S = [-1, 0, 1, 2, -1, -4],

A solution set is:

[

[-1, 0, 1],

[-1, -1, 2]

]

My original solution is to add another for loop based on Twosum problem. While not consider the duplicate issue.

One good method is use two pointers, head and tail. Sort first. As long as head is smaller than tail, compare the sum values of head and tail to outer value. Move till find the unduplicate value;

1. **class** Solution {
2. **public** List<List<Integer>> threeSum(**int**[] nums) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
4. **int** len = nums.length;
5. Arrays.sort(nums);
6. **for** (**int** i = 0;i < len-2;i++) {
7. **if** (i==0 || (i>0 && nums[i]!=nums[i-1])) {
8. **int** lo = i+1, hi = len-1, val = -nums[i];
9. **while**(lo < hi) {
10. **if** (nums[lo] + nums[hi] == val) {
11. ans.add(Arrays.asList(nums[i], nums[lo], nums[hi]));
12. **while**(lo < hi && nums[lo] == nums[lo+1]) lo++;
13. **while**(lo < hi && nums[hi] == nums[hi-1]) hi--;
14. lo++;
15. hi--;
16. } **else** **if**(nums[lo]+nums[hi] > val) {
17. hi--;
18. } **else** {
19. lo++;
20. }
21. }
22. }
23. }
24. **return** ans;
25. }
26. }

16. 3Sum Closest

Given an array *S* of *n* integers, find three integers in *S* such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array S = {-1 2 1 -4}, and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Similar algorithm and data structure with 3sum. Two pointers where head and tail move to inner array meanwhile.

1. **class** Solution {
2. **public** **int** threeSumClosest(**int**[] nums, **int** target) {
3. **int** ans = nums[0] + nums[1] + nums[nums.length-1];
4. Arrays.sort(nums);
5. **for** (**int** i = 0;i < nums.length-2;i++) {
6. **int** lo = i+1, hi = nums.length - 1;
7. **while** (lo < hi) {
8. **int** val = nums[i] + nums[lo] + nums[hi];
9. **if** (val < target) {
10. lo++;
11. } **else** **if**(val > target) {
12. hi--;
13. } **else** {
14. **return** val;
15. }
16. **if** (Math.abs(val - target) < Math.abs(ans - target)) {
17. ans = val;
18. }
19. }
20. }
21. **return** ans;
22. }
23. }

17. Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.



**Input:**Digit string "23"

**Output:** ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

**Note:**  
Although the above answer is in lexicographical order, your answer could be in any order you want.

Iterative Solution:

Use the idea of FIFO

1. **public** **class** Solution {
2. **public** List<String> letterCombinations(String digits) {
3. LinkedList<String> ans  = **new** LinkedList<String>();
4. **if** (digits.length() == 0) **return** ans;
5. String[] map = {"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
6. ans.add("");
7. **for** (**int** i = 0;i < digits.length();i++) {
8. **int** mapIndex = digits.charAt(i) - '0';
9. **while**(ans.peek().length() == i) {
10. String curr = ans.remove();
11. **for** (**char** c:map[mapIndex].toCharArray()) {
12. ans.add(curr+c);
13. }
14. }
15. }
16. **return** ans;
17. }
18. }

Recursive Solution:

Figure out what paramter needs to be passed in recursion

1. **public** **class** Solution {
2. String[] map = {"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
3. **public** List<String> letterCombinations(String digits) {
4. List<String> ans = **new** ArrayList<String>();
5. **if** (digits.length() == 0) **return** ans;
6. combination("", digits, 0, ans);
7. **return** ans;
8. }
9. **private** **void** combination(String prefix, String digits, **int** start,List<String> ans) {
10. **if** (start >= digits.length()) {
11. ans.add(prefix);
12. **return**;
13. }
14. String letters = map[(digits.charAt(start) - '0')];
15. **for** (**int** i = 0;i<letters.length();i++) {
16. combination(prefix+letters.charAt(i), digits, start+1,ans);
17. }
18. }
19. }

18. 4Sum

Given an array *S* of *n* integers, are there elements *a*, *b*, *c*, and *d* in *S* such that *a* + *b* + *c* + *d* = target? Find all unique quadruplets in the array which gives the sum of target.

**Note:** The solution set must not contain duplicate quadruplets.

For example, given array S = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is:

[

[-1, 0, 0, 1],

[-2, -1, 1, 2],

[-2, 0, 0, 2]

]

Similat idea with 3Sum

1. **class** Solution {
2. **public** List<List<Integer>> fourSum(**int**[] nums, **int** target) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
4. Arrays.sort(nums);
5. **for** (**int** i = 0; i < nums.length-3; i++) {
6. **if** (i!=0&&nums[i]==nums[i-1]) **continue**;
7. **for** (**int** j=i+1;j < nums.length-2;j++) {
8. **if** (j!=i+1&&nums[j]==nums[j-1]) **continue**;
9. **int** lo = j+1,hi=nums.length-1;
10. **int** val = target - nums[i] - nums[j];
11. **while**(lo < hi) {
12. **if** (nums[lo] + nums[hi] == val) {
13. ans.add(Arrays.asList(nums[i],nums[j],nums[lo],nums[hi]));
14. **while**(lo < hi && nums[lo] == nums[lo+1]) lo++;
15. **while**(lo < hi && nums[hi] == nums[hi-1]) hi--;
16. hi--;
17. lo++;
18. } **else** **if**(nums[lo] + nums[hi] > val) {
19. hi--;
20. } **else** {
21. lo++;
22. }
23. }
24. }
25. }
26. **return** ans;
27. }
28. }

19. Remove Nth Node From End of List

Given a linked list, remove the *n*th node from the end of list and return its head.

For example,

Given linked list: **1->2->3->4->5**, and ***n* = 2**.

After removing the second node from the end, the linked list becomes **1->2->3->5**.

**Note:**  
Given *n* will always be valid.  
Try to do this in one pass.

Two pointers method.

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **class** Solution {
10. **public** ListNode removeNthFromEnd(ListNode head, **int** n) {
11. ListNode fast = head;
12. ListNode slow = head;
13. **while**(n>0) {
14. fast = fast.next;
15. n--;
16. }
17. **if** (fast == **null**) {
18. **return** head.next;
19. }
20. **while** (fast.next!= **null**) {
21. fast = fast.next;
22. slow = slow.next;
23. }
24. slow.next = slow.next.next;
25. **return** head;
27. }
28. }

22. Generate Parentheses

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

]

Backtracking problem.

1. **class** Solution {
2. **public** List<String> generateParenthesis(**int** n) {
3. List<String> ans = **new** ArrayList<String>();
4. backTracking(ans,"",0,0,n);
5. **return** ans;
7. }
8. **private** **void** backTracking(List<String> list, String subString, **int** left, **int** right,**int** n) {
9. **if** (left == n && right==n) {
10. list.add(subString);
11. **return**;
12. }
13. **if** (left < n) {
14. backTracking(list, subString+"(",left+1, right,n);
15. }
16. **if** (right < left) {
17. backTracking(list, subString+")",left,right+1,n);
18. }
20. }
21. }

75. Sort Colors

Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

**Note:**  
You are not suppose to use the library's sort function for this problem.

To solve this problem in one pass, use three pointers which point to red, while and blue color repectively. Swap the pair to make the every value appear in the right position of the array.

1. **class** Solution {
2. **public** **void** sortColors(**int**[] nums) {
3. **int** start = 0, mid = 0, end = nums.length - 1;
4. **int** midValue = 1;
5. **while**(mid <= end) {
6. **if**(nums[mid] == midValue) {
7. mid++;
8. } **else** **if**(nums[mid] > midValue) {
9. swap(nums, mid, end);
10. end--;
11. } **else** {
12. swap(nums, start, mid);
13. start++;
14. mid++;
15. }
16. }
18. }
19. **public** **void** swap(**int**[] arr, **int** i, **int** j) {
20. **int** temp = arr[i];
21. arr[i] = arr[j];
22. arr[j] = temp;
23. }
24. }

24. Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.

For example,  
Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may **not** modify the values in the list, only nodes itself can be changed.

LinkedList, the useage of fake head!

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **class** Solution {
10. **public** ListNode swapPairs(ListNode head) {
11. **if** (head==**null** || head.next == **null**) **return** head;
12. ListNode newHead = **new** ListNode(0);
13. ListNode curr = newHead;
14. newHead.next = head;
15. **while** (curr.next!=**null** && curr.next.next!=**null**) {
16. ListNode first = curr.next;
17. ListNode second = curr.next.next;
18. first.next = second.next;
19. curr.next = second ;
20. curr.next.next = first;
21. curr = curr.next.next;
22. }
23. **return** newHead.next;
24. }
25. }

29. Divide Two Integers

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX\_INT.

In this problem, we are asked to divide two integers. However, we are not allowed to use division, multiplication and mod operations. So, what else can we use? Yeah, bit manipulations.

Let's do an example and see how bit manipulations work.

Suppose we want to divide 15 by 3, so 15 is dividend and 3 is divisor. Well, division simply requires us to find how many times we can subtract the divisor from the the dividend without making the dividend negative.

Let's get started. We subtract 3 from 15 and we get 12, which is positive. Let's try to subtract more. Well, we **shift** 3 to the left by 1 bit and we get 6. Subtracting 6 from 15 still gives a positive result. Well, we shift again and get 12. We subtract 12 from 15 and it is still positive. We shift again, obtaining 24 and we know we can at most subtract 12. Well, since 12 is obtained by shifting 3 to left twice, we know it is 4 times of 3. How do we obtain this 4? Well, we start from 1 and shift it to left twice at the same time. We add 4 to an answer (initialized to be 0). In fact, the above process is like 15 = 3 \* 4 + 3. We now get part of the quotient (4), with a remainder 3.

Then we repeat the above process again. We subtract divisor = 3 from the remaining dividend = 3 and obtain 0. We know we are done. No shift happens, so we simply add 1 << 0 to the answer.

Now we have the full algorithm to perform division.

According to the problem statement, we need to handle some exceptions, such as overflow.

Well, two cases may cause overflow:

1. divisor = 0;
2. dividend = INT\_MIN and divisor = -1 (because abs(INT\_MIN) = INT\_MAX + 1).

Of course, we also need to take the sign into considerations, which is relatively easy.

1. **class** Solution {
2. **public** **int** divide(**int** dividend, **int** divisor) {
3. **if**(divisor==0 || (dividend ==Integer.MIN\_VALUE && divisor == -1)) **return** Integer.MAX\_VALUE;
4. **int** ans = 0;
5. **int** sign = ((dividend < 0)^(divisor<0)?-1:1);
6. **long** dvd = Math.abs((**long**)dividend);
7. **long** dvs = Math.abs((**long**)divisor);
9. **while** (dvd >= dvs) {
11. **long** temp = dvs, multiple = 1;
12. **while**(dvd >= (temp<<1)) {
13. temp <<= 1;
14. multiple <<= 1;
15. }
16. dvd -= temp;
17. ans += multiple;
18. }
19. **return** sign == 1? ans:-ans;
20. }
22. }

31. Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.  
1,2,3 → 1,3,2  
3,2,1 → 1,2,3  
1,1,5 → 1,5,1

To find the next number, the key point is to find out which two numbers need to swap. From the end of the array, check whether the array is ascending. From where array stops ascending, swap the value in this postion and the value which is smallest among all the numbers to the end bigger than this number. After that, ascending the array after this postion.

Notice Arrays.sort can sort part of an array given the start and end index.

1. **class** Solution {
2. **public** **void** nextPermutation(**int**[] nums) {
3. **if** (nums.length == 0 || nums.length == 1) **return**;
4. **int** end = nums.length-1;
5. **while** (end > 0 && nums[end] <= nums[end-1]) {
6. end--;
7. }
8. **if** (end == 0) {
9. Arrays.sort(nums);
10. } **else** {
11. **int** k = end-1;
12. **int** val = nums[k];
13. **while**(k < nums.length-1) {
14. **if** (val >= nums[k+1]) {
15. **break**;
16. }
17. k++;
18. }
19. swap(nums,k,end-1);
20. Arrays.sort(nums,end,nums.length);
21. }
22. }
23. **public** **void** swap(**int**[] arr, **int** i,**int** j) {
24. **int** temp = arr[i];
25. arr[i] = arr[j];
26. arr[j] = temp;
27. }
28. }

33. Search in Rotated Sorted Array

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Use Binary Search, find which part of the array is ascending. Notice the loop conditin is (low +1 ) < high which can avoid the corner case in like (3,1). Also add two addition lines in the end.

1. **class** Solution {
2. **public** **int** search(**int**[] nums, **int** target) {
3. **if** (nums == **null** || nums.length == 0) **return** -1;
4. **int** low = 0, high = nums.length - 1;
5. **while**(low + 1< high) {
6. **int** mid = low + (high - low)/2;
7. **if** (nums[mid] == target) {
8. **return** mid;
9. }
10. **if** (nums[low] < nums[mid]) {
11. **if** (target >= nums[low] && target <= nums[mid]) {
12. high = mid;
13. } **else** {
14. low = mid;
15. }
16. } **else** {
17. **if** (target >= nums[mid] && target<=nums[high]) {
18. low = mid;
19. } **else** {
20. high = mid;
21. }
22. }
23. }
24. **if** (nums[low] == target) **return** low;
25. **if** (nums[high] == target) **return** high;
26. **return** -1;
27. }
28. }

34. Search for a Range

Given an array of integers sorted in ascending order, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

If the target is not found in the array, return [-1, -1].

For example,  
Given [5, 7, 7, 8, 8, 10] and target value 8,  
return [3, 4].

Another form of Binary Search. The first search is to find the left bound of target(if exists), second search is to find the right bound of the target, notice the move condition.

1. **class** Solution {
2. **public** **int**[] searchRange(**int**[] nums, **int** target) {
3. **int**[] ans = {-1,-1};
4. **if** (nums == **null** || nums.length == 0) {
5. **return** ans;
6. }
7. **int** start = 0, end = nums.length - 1;
8. **int** left = findLeft(nums, target);
9. **if** (left == -1) **return** ans;
10. **int** right = findRight(nums, target);
11. ans[0] = left;
12. ans[1] = right;
13. **return** ans;
15. }
16. **public** **int** findLeft(**int**[] nums, **int** target) {
17. **int** start = 0, end = nums.length - 1;
18. **while** (start +1 < end) {
19. **int** mid = start + (end - start)/2;
20. **if** (nums[mid] >= target) {
21. end = mid;
22. } **else** {
23. start = mid;
24. }
25. }
26. **if** (nums[start] == target) **return** start;
27. **if** (nums[end] == target) **return** end;
28. **return** -1;
29. }
30. **public** **int** findRight(**int**[] nums, **int** target) {
31. **int** start = 0, end = nums.length - 1;
32. **while** (start +1 < end) {
33. **int** mid = start + (end - start)/2;
34. **if** (nums[mid] <= target) {
35. start = mid;
36. } **else** {
37. end = mid;
38. }
39. }
40. **if** (nums[end] == target) **return** end;
41. **if** (nums[start] == target) **return** start;
42. **return** -1;
43. }
44. }

36. Valid Sudoku

Determine if a Sudoku is valid, according to: [Sudoku Puzzles - The Rules](http://sudoku.com.au/TheRules.aspx).

The Sudoku board could be partially filled, where empty cells are filled with the character '.'.



A partially filled sudoku which is valid.

**Note:**  
A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

My original method is more complicated than this answer, which is pretty clean and easy to explain.

1. **class** Solution {
2. **public** **boolean** isValidSudoku(**char**[][] board) {
3. Set seen = **new** HashSet();
4. **for** (**int** i=0; i< 9; i++) {
5. **for** (**int** j=0; j<9; j++) {
6. **char** number = board[i][j];
7. **if** (number != '.')
8. **if** (!seen.add(number + " in row " + i) ||
9. !seen.add(number + " in column " + j) ||
10. !seen.add(number + " in block " + i/3 + "-" + j/3))
11. **return** **false**;
12. }
13. }
14. **return** **true**;
15. }
16. }

39. Combination Sum

Given a **set** of candidate numbers (***C***) **(without duplicates)** and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

The **same** repeated number may be chosen from ***C*** unlimited number of times.

**Note:**

* All numbers (including target) will be positive integers.
* The solution set must not contain duplicate combinations.

For example, given candidate set [2, 3, 6, 7] and target 7,   
A solution set is:

[

[7],

[2, 2, 3]

]

Useage of BackTracking, contrast with 40,

1. **class** Solution {
2. **public** List<List<Integer>> combinationSum(**int**[] candidates, **int** target) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
4. **if** (candidates == **null** || candidates.length == 0) **return** ans;
5. Arrays.sort(candidates);
6. backTracking(candidates,target,0,ans,**new** ArrayList());
7. **return** ans;
8. }
9. **private** **void** backTracking(**int**[] candidates,**int** target, **int** start,List<List<Integer>> ans,List<Integer> list) {
10. **if** (target < 0) **return**;
11. **if** (target == 0) ans.add(**new** ArrayList<Integer>(list));
12. **for** (**int** i = start;i<candidates.length;i++) {
13. **if** (target < candidates[i]) **break**;
14. list.add(candidates[i]);
15. backTracking(candidates, target-candidates[i],i,ans,list);
16. list.remove(list.size()-1);
17. }
18. }
19. }

40. Combination Sum II

Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

Each number in ***C*** may only be used **once** in the combination.

**Note:**

* All numbers (including target) will be positive integers.
* The solution set must not contain duplicate combinations.

For example, given candidate set [10, 1, 2, 7, 6, 1, 5] and target 8,   
A solution set is:

[

[1, 7],

[1, 2, 5],

[2, 6],

[1, 1, 6]

]

1. **class** Solution {
2. **public** List<List<Integer>> combinationSum2(**int**[] candidates, **int** target) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
4. **if** (candidates == **null** || candidates.length == 0) **return** ans;
5. Arrays.sort(candidates);
6. backTracking(candidates,target,0,ans,**new** ArrayList());
7. **return** ans;
8. }
9. **private** **void** backTracking(**int**[] candidates,**int** target, **int** start,List<List<Integer>> ans,List<Integer> list) {
10. **if** (target < 0) **return**;
11. **if** (target == 0) ans.add(**new** ArrayList<Integer>(list));
12. **for** (**int** i = start;i<candidates.length;i++) {
13. **if** (i != start && candidates[i] == candidates[i-1]) **continue**;
14. list.add(candidates[i]);
15. backTracking(candidates, target-candidates[i],i+1,ans,list);
16. list.remove(list.size()-1);
17. }
18. }
19. }

43. Multiply Strings

Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2.

**Note:**

1. The length of both num1 and num2 is < 110.
2. Both num1 and num2 contains only digits 0-9.
3. Both num1 and num2 does not contain any leading zero.
4. You **must not use any built-in BigInteger library** or **convert the inputs to integer** directly.

Key point:

`num1[i] \* num2[j]` will be placed at indices `[i + j`, `i + j + 1]`

1. **class** Solution {
2. **public** String multiply(String num1, String num2) {
3. **int** len1 = num1.length();
4. **int** len2 = num2.length();
5. **if** (len1 == 0 || len2 == 0) **return** "0";
6. **int**[] res = **new** **int**[len1+len2];
7. **for** (**int** i=len1-1;i>=0;i--) {
8. **for** (**int** j=len2-1;j>=0;j--) {
9. **int** po1 = i+j,  po2=i+j+1;
10. **int** mul = (num1.charAt(i)-'0') \* (num2.charAt(j)-'0');
12. res[po1] = (res[po2]+mul) / 10 + res[po1];
13. res[po2] = (res[po2]+mul) % 10;
14. }
15. }
16. StringBuilder ans = **new** StringBuilder();
17. **for** (**int** num:res) {
18. **if** (!(num==0 && ans.length()==0)) ans.append(num);
19. }
20. **return** ans.length() == 0?"0":ans.toString();
21. }
22. }

46. Permutations

Given a collection of **distinct** numbers, return all possible permutations.

For example,  
[1,2,3] have the following permutations:

[

[1,2,3],

[1,3,2],

[2,1,3],

[2,3,1],

[3,1,2],

BackTracking , together with 47!

1. **class** Solution {
2. **public** List<List<Integer>> permute(**int**[] nums) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
4. **if** (nums == **null** || nums.length == 0) **return** ans;
5. backTracking(nums,ans,**new** ArrayList<Integer>(),**new** HashSet<Integer>());
6. **return** ans;
7. }
8. **public** **void** backTracking(**int**[] nums,List<List<Integer>> ans,List<Integer> subList , HashSet<Integer> set) {
9. **if** (subList.size() == nums.length) {
10. ans.add(**new** ArrayList(subList));
11. } **else** {
12. **for** (**int** i=0;i<nums.length;i++) {
13. **if** (!set.contains(nums[i])) {
14. subList.add(nums[i]);
15. set.add(nums[i]);
16. backTracking(nums,ans,subList,set);
17. set.remove(nums[i]);
18. subList.remove(subList.size()-1);
19. }
21. }
22. }
23. }
24. }

47. Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,  
[1,1,2] have the following unique permutations:

[

[1,1,2],

[1,2,1],

[2,1,1]

]

1. **class** Solution {
2. **public** List<List<Integer>> permuteUnique(**int**[] nums) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
4. **if** (nums == **null** || nums.length == 0) **return** ans;
5. Arrays.sort(nums);
6. backTracking(nums,ans,**new** ArrayList(),**new** **boolean**[nums.length]);
7. **return** ans;
8. }
9. **public** **void** backTracking(**int**[] nums, List<List<Integer>> ans,List<Integer> subList,**boolean**[] used) {
10. **if** (subList.size() == nums.length) {
11. ans.add(**new** ArrayList(subList));
12. } **else** {
13. **for** (**int** i=0;i<nums.length;i++) {
14. **if** ((i !=0 && nums[i] == nums[i-1] && !used[i-1]) || used[i] ) **continue**;
15. subList.add(nums[i]);
16. used[i] =**true**;
17. backTracking(nums, ans, subList, used);
18. used[i] = **false**;
19. subList.remove(subList.size()-1);
20. }
21. }
22. }
23. }

48. Rotate Image

You are given an *n* x *n* 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

**Note:**  
You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**

Given **input matrix** =

[

[1,2,3],

[4,5,6],

[7,8,9]

],

rotate the input matrix **in-place** such that it becomes:

[

[7,4,1],

[8,5,2],

[9,6,3]

]

**Example 2:**

Given **input matrix** =

[

[ 5, 1, 9,11],

[ 2, 4, 8,10],

[13, 3, 6, 7],

[15,14,12,16]

],

rotate the input matrix **in-place** such that it becomes:

[

[15,13, 2, 5],

[14, 3, 4, 1],

[12, 6, 8, 9],

[16, 7,10,11]

]

Use the symmetrical property. First switch rows and columns and then reverse each row.Be careful to switch since repeat may occur.

1. **class** Solution {
2. **public** **void** rotate(**int**[][] matrix) {
3. **int** row = matrix.length;
4. **int** col = matrix[0].length;
5. **for** (**int** i=0;i<row;i++) {
6. **for** (**int** j=i+1;j<col;j++) {
7. **int** temp = matrix[i][j];
8. matrix[i][j] = matrix[j][i];
9. matrix[j][i] = temp;
10. }
11. }
12. **for** (**int** i=0;i<row;i++) {
13. **for** (**int** j=0;j<col/2;j++) {
14. **int** temp = matrix[i][j];
15. matrix[i][j] = matrix[i][col-1-j];
16. matrix[i][col-1-j] = temp;
17. }
18. }
19. }
20. }

49. Group Anagrams

Given an array of strings, group anagrams together.

For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],   
Return:

[

["ate", "eat","tea"],

["nat","tan"],

["bat"]

]

1. **class** Solution{
2. **public** List<List<String>> groupAnagrams(String[] strs) {
3. **if** (strs == **null** || strs.length == 0) **return** **new** ArrayList<List<String>>();
4. Map<String, List<String>> map = **new** HashMap<String, List<String>>();
5. **for** (String s : strs) {
6. **char**[] ca = s.toCharArray();
7. Arrays.sort(ca);
8. String keyStr = String.valueOf(ca);
9. **if** (!map.containsKey(keyStr)) map.put(keyStr, **new** ArrayList<String>());
10. map.get(keyStr).add(s);
11. }
12. **return** **new** ArrayList<List<String>>(map.values());
13. }
14. }

50. Pow(x, n)

Implement [pow(*x*, *n*)](http://www.cplusplus.com/reference/valarray/pow/).

**Example 1:**

**Input:** 2.00000, 10

**Output:** 1024.00000

**Example 2:**

**Input:** 2.10000, 3

**Output:** 9.26100

The realization of Math.pow() function, pay attenation to the boundary conditions

1. **class** Solution {
2. **public** **double** myPow(**double** x, **int** n) {
4. **if** (n == 0) **return** 1;
5. **if** (n == Integer.MIN\_VALUE) {
6. x = x\*x;
7. n = n/2;
8. }
9. **if** (n < 0) {
10. x = 1/x;
11. n = -n;
12. }
13. **return** (n%2 == 0)? myPow(x\*x,n/2):x\*myPow(x\*x,n/2);
14. }
15. }

54. Spiral Matrix

Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.

For example,  
Given the following matrix:

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

You should return [1,2,3,6,9,8,7,4,5].

Traverse in turn from right, down, left ,up. Tricky part is when traverse left or up, need to check whether row and column still exists to prevent duplicates

1. **class** Solution {
2. **public** List<Integer> spiralOrder(**int**[][] matrix) {
3. List<Integer> ans = **new** ArrayList<Integer>();
4. **if** (matrix == **null** || matrix.length == 0 ) **return** ans;
5. **int** rowStart = 0;
6. **int** colStart = 0;
7. **int** rowEnd = matrix.length-1;
8. **int** colEnd = matrix[0].length - 1;
9. **while** (rowStart <= rowEnd && colStart <= colEnd) {
10. // go right
11. **for** (**int** i=colStart; i<=colEnd;i++) {
12. ans.add(matrix[rowStart][i]);
13. }
14. rowStart++;
16. // go down
17. **for** (**int** i=rowStart;i<=rowEnd;i++) {
18. ans.add(matrix[i][colEnd]);
19. }
20. colEnd--;
22. // go left
23. **if** (rowStart <= rowEnd) {
24. **for** (**int** i=colEnd;i>=colStart;i--) {
25. ans.add(matrix[rowEnd][i]);
26. }
27. rowEnd--;
28. }
30. // go up
31. **if** (colStart <= colEnd) {
32. **for** (**int** i=rowEnd;i>=rowStart;i--) {
33. ans.add(matrix[i][colStart]);
34. }
35. colStart++;
36. }
37. }
38. **return** ans;
39. }
40. }

55. Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example:  
A = [2,3,1,1,4], return true.

A = [3,2,1,0,4], return false.

Greedy algorithm, find out the maximum range every time and make sure every step is within the reach.

1. **class** Solution {
2. **public** **boolean** canJump(**int**[] nums) {
3. **if** (nums.length<2) **return** **true**;
4. **int** reach = 0;
5. **for** (**int** i=0;i<nums.length && i<= reach ;i++) {
6. reach = Math.max(reach,i+nums[i]);
7. **if** (reach >= nums.length-1) **return** **true**;
8. }
9. **return** **false**;
10. }
11. }

56. Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,  
Given [1,3],[2,6],[8,10],[15,18],  
return [1,6],[8,10],[15,18].

Usage of Comparator

1. /\*\*
2. \* Definition for an interval.
3. \* public class Interval {
4. \*     int start;
5. \*     int end;
6. \*     Interval() { start = 0; end = 0; }
7. \*     Interval(int s, int e) { start = s; end = e; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** List<Interval> merge(List<Interval> intervals) {
12. **if** (intervals.size() == 0) **return** intervals;
13. List<Interval> ans = **new** ArrayList<Interval>();
14. Collections.sort(intervals, **new** Comparator<Interval>() {
15. @Override
16. **public** **int** compare(Interval i1, Interval i2) {
17. **if** (i1.start != i2.start) {
18. **return** i1.start - i2.start;
19. }
20. **return** i1.end - i2.end;
21. }
22. });
23. **int** lo = intervals.get(0).start;
24. **int** hi = intervals.get(0).end;
25. **for** (**int** i=1;i<intervals.size();i++) {
26. **if** (intervals.get(i).start <= hi) {
27. hi = Math.max(hi,intervals.get(i).end);
28. } **else** {
29. ans.add(**new** Interval(lo,hi));
30. lo = intervals.get(i).start;
31. hi = intervals.get(i).end;
32. }
33. }
34. ans.add(**new** Interval(lo,hi));
35. **return** ans;
36. }
37. }

59. Spiral Matrix II

Given an integer *n*, generate a square matrix filled with elements from 1 to *n*2 in spiral order.

For example,  
Given *n* = 3,

You should return the following matrix:

[

[ 1, 2, 3 ],

[ 8, 9, 4 ],

[ 7, 6, 5 ]

]

The same idea as Spiral Matrix (54)

1. **class** Solution {
2. **public** **int**[][] generateMatrix(**int** n) {
3. **int**[][] ans = **new** **int**[n][n];
4. **int** rowStart = 0;
5. **int** rowEnd = n-1;
6. **int** colStart = 0;
7. **int** colEnd = n-1;
8. **int** num = 1;
9. **while**(rowStart <= rowEnd && colStart <= colEnd) {
10. // go right
11. **for** (**int** i=colStart; i<=colEnd;i++) {
12. ans[rowStart][i] = num;
13. num++;
14. }
15. rowStart++;
17. // go down
18. **for** (**int** i=rowStart; i<= rowEnd;i++) {
19. ans[i][colEnd] = num;
20. num++;
21. }
22. colEnd--;
24. // go left
25. **if** (rowStart <= rowEnd) {
26. **for** (**int** i=colEnd;i>=colStart;i--) {
27. ans[rowEnd][i] = num;
28. num++;
29. }
30. rowEnd--;
31. }
33. // go up
34. **if** (colStart <= colEnd) {
35. **for** (**int** i=rowEnd;i>=rowStart;i--) {
36. ans[i][colStart] = num;
37. num++;
38. }
39. colStart++;
40. }
41. }
42. **return** ans;
44. }
45. }

94. Binary Tree Inorder Traversal

Given a binary tree, return the *inorder* traversal of its nodes' values.

For example:  
Given binary tree [1,null,2,3],

1

\

2

/

3

return [1,3,2].

A example lead by BITTIGER. A template in tree’s traversal.

Use Iterative other than Recursive

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** List<Integer> inorderTraversal(TreeNode root) {
12. List<Integer> ans = **new** ArrayList<Integer>();
13. Deque<Guide> path = **new** ArrayDeque<Guide>();
14. path.add(**new** Guide(0,root));
16. **while**(!path.isEmpty()) {
17. Guide current = path.removeFirst();
18. **if** (current.node == **null**) **continue**;
19. **if** (current.ope == 1) {
20. ans.add(current.node.val);
21. } **else** {
22. path.addFirst(**new** Guide(0,current.node.right));
23. path.addFirst(**new** Guide(1,current.node));
24. path.addFirst(**new** Guide(0,current.node.left));
25. }
27. }
28. **return** ans;
30. }
31. **private** **class** Guide {
32. **int** ope;
33. TreeNode node;
34. **private** Guide(**int** ope, TreeNode node){
35. **this**.ope = ope;
36. **this**.node = node;
37. }
38. }
39. }

144. Binary Tree Preorder Traversal

Given a binary tree, return the *preorder* traversal of its nodes' values.

For example:  
Given binary tree [1,null,2,3],

1

\

2

/

3

return [1,2,3].

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. // class Solution {
11. //     public List<Integer> preorderTraversal(TreeNode root) {
12. //         List<Integer> ans = new ArrayList<Integer>();
13. //         preorder(root, ans);
14. //         return ans;
15. //     }
16. //     public void preorder(TreeNode root, List<Integer> list) {
17. //         if (root == null) return;
18. //         int value = root.val;
19. //         list.add(value);
20. //         preorder(root.left,list);
21. //         preorder(root.right,list);
22. //     }
23. // }
24. **class** Solution {
25. **public** List<Integer> preorderTraversal(TreeNode root) {
26. List<Integer> res = **new** ArrayList<>();
27. Deque<Guide> path = **new** ArrayDeque<>();
28. path.addFirst(**new** Guide(0,root));
30. **while** (!path.isEmpty()) {
31. Guide current = path.removeFirst();
32. **if** (current.node == **null**) **continue**;
34. **if** (current.ope == 1) {
35. res.add(current.node.val);
36. } **else** {
37. path.addFirst(**new** Guide(0,current.node.right));
38. path.addFirst(**new** Guide(0,current.node.left));
39. path.addFirst(**new** Guide(1,current.node));
40. }
41. }
42. **return** res;
43. }
44. **private** **class** Guide {
45. **int** ope; // 0:visit  1: print
46. TreeNode node;
47. **private** Guide(**int** ope, TreeNode node) {
48. **this**.ope = ope;
49. **this**.node = node;
50. }
51. }
52. }

145. Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

For example:  
Given binary tree [1,null,2,3],

1

\

2

/

3

return [3,2,1].

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
11. **class** Solution {
12. **public** List<Integer> postorderTraversal(TreeNode root) {
13. List<Integer> res = **new** ArrayList<>();
14. Deque<Guide> path = **new** ArrayDeque<>();
15. path.addFirst(**new** Guide(0,root));
17. **while** (!path.isEmpty()) {
18. Guide current = path.removeFirst();
19. **if** (current.node == **null**) **continue**;
21. **if** (current.ope == 1) {
22. res.add(current.node.val);
23. } **else** {
24. path.addFirst(**new** Guide(1,current.node));
25. path.addFirst(**new** Guide(0,current.node.right));
26. path.addFirst(**new** Guide(0,current.node.left));
27. }
28. }
29. **return** res;
30. }
31. **private** **class** Guide {
32. **int** ope; // 0:visit  1: print
33. TreeNode node;
34. **private** Guide(**int** ope, TreeNode node) {
35. **this**.ope = ope;
36. **this**.node = node;
37. }
38. }
39. }

60. Permutation Sequence

The set [1,2,3,…,*n*] contains a total of *n*! unique permutations.

By listing and labeling all of the permutations in order,  
We get the following sequence (ie, for *n* = 3):

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given *n* and *k*, return the *k*th permutation sequence.

A math problem, divide into groups

1. **class** Solution {
2. **public** String getPermutation(**int** n, **int** k) {
3. List<Integer> list = **new** ArrayList<Integer>();
4. **for** (**int** i=1;i<=n;i++) {
5. list.add(i);
6. }
8. **int**[] fact = **new** **int**[n];
9. fact[0] = 1;
10. **for**(**int** i=1;i< n;i++) {
11. fact[i] = fact[i-1]\*i;
12. }
14. k = k-1;
16. StringBuilder res = **new** StringBuilder();
17. **for** (**int** i = n;i>0;i--) {
18. **int** index = k / fact[i-1];
19. res.append(list.get(index));
20. k = k % fact[i-1];
21. list.remove(index);
22. }
23. **return** res.toString();
24. }
25. }

61. Rotate List

Given a list, rotate the list to the right by *k* places, where *k* is non-negative.

**Example:**

Given **1->2->3->4->5->NULL** and *k* = **2**,

return **4->5->1->2->3->NULL**.

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **class** Solution {
10. **public** ListNode rotateRight(ListNode head, **int** k) {
11. **if** (head == **null**) {
12. **return** head;
13. }
14. **int** count = 1;
15. ListNode fast = head;
16. **while** (fast.next != **null**) {
17. count++;
18. fast = fast.next;
19. }
20. // k = k % count;
21. // if (k == 0) return head;
22. ListNode slow = head;
24. **for** (**int** i= count - k % count;i>1;i--) {
25. slow = slow.next;
26. }
28. fast.next = head;
29. head = slow.next;
30. slow.next = **null**;
31. **return** head;
33. }
34. }

62. Unique Paths

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?



The most fundamental problem of DP.

1. **class** Solution {
2. **public** **int** uniquePaths(**int** m, **int** n) {
3. **int**[][] dp = **new** **int**[m][n];
4. **for** (**int** i=0;i<m;i++) {
5. **for** (**int** j=0;j<n;j++) {
6. **if** (i == 0 || j == 0) {
7. dp[i][j] = 1;
8. **continue**;
9. }
10. dp[i][j] = dp[i-1][j]+dp[i][j-1];
11. }
12. }
13. **return** dp[m-1][n-1];
14. }
15. }

63. Unique Paths II

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

[

[0,0,0],

[0,1,0],

[0,0,0]

]

The total number of unique paths is 2.

1. **class** Solution {
2. **public** **int** uniquePathsWithObstacles(**int**[][] obstacleGrid) {
3. **int** row = obstacleGrid.length;
4. **int** col = obstacleGrid[0].length;
5. **int** dp[][] = **new** **int**[row][col];
6. **for** (**int** i=0;i<row;i++) {
7. **for** (**int** j=0;j<col;j++) {
8. **if** (obstacleGrid[i][j] != 1) {
9. **if** (i==0 && j == 0) {
10. dp[i][j] = 1;
11. **continue**;
12. }
13. **if** (i==0 && j >= 1) {
14. dp[i][j] = dp[i][j-1];
15. **continue**;
16. }
17. **if** (j==0 && i >= 1) {
18. dp[i][j] = dp[i-1][j];
19. **continue**;
20. }
21. dp[i][j] = dp[i-1][j] + dp[i][j-1];
22. } **else** {
23. dp[i][j] = 0;
24. }
25. }
26. }
27. **return** dp[row-1][col-1];
28. }
29. }

64. Minimum Path Sum

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

**Note:** You can only move either down or right at any point in time.

**Example 1:**

[[1,3,1],

[1,5,1],

[4,2,1]]

Given the above grid map, return 7. Because the path 1→3→1→1→1 minimizes the sum.

Dynamic Programming, use old 2D array and new 2D array properly

1. **class** Solution {
2. **public** **int** minPathSum(**int**[][] grid) {
3. **int** row = grid.length;
4. **if** (grid == **null** || row == 0) **return** 0;
5. **int** col = grid[0].length;
7. **int**[][] dp = **new** **int**[row][col];
9. **for** (**int** i=0;i<row;i++) {
10. **for** (**int** j=0;j<col;j++) {
11. **if** (i==0 && j==0) {
12. dp[i][j] = grid[i][j];
13. } **else** **if** (i==0){
14. dp[i][j] = grid[i][j]+dp[i][j-1];
15. } **else** **if** (j==0) {
16. dp[i][j] = grid[i][j]+ dp[i-1][j];
17. } **else** {
18. dp[i][j] = grid[i][j]+ Math.min(dp[i][j-1],dp[i-1][j]);
19. }
20. }
21. }
22. **return** dp[row-1][col-1];
23. }
24. }

71. Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,  
**path** = "/home/", => "/home"  
**path** = "/a/./b/../../c/", => "/c"

**Corner Cases:**

* Did you consider the case where **path** = "/../"?  
  In this case, you should return "/".
* Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/".  
  In this case, you should ignore redundant slashes and return "/home/foo".

1. **class** Solution {
2. **public** String simplifyPath(String path) {
3. StringBuilder ans = **new** StringBuilder("/");
4. Deque<String> queue = **new** LinkedList<String>();
5. **for** (String s:path.split("/")) {
6. **if** (s.equals("..")) {
7. **if** (!queue.isEmpty()) {
8. queue.removeLast();
9. }
10. } **else** **if** (!s.equals("") && !s.equals(".")) {
11. queue.add(s);
12. }
13. }
14. **for** (String q:queue) {
15. ans.append(q).append("/");
16. }
17. **if** (!queue.isEmpty()) {
18. ans.setLength(ans.length()-1);
19. }
20. **return** ans.toString();
22. }
23. }

73. Set Matrix Zeroes

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in place.

**Follow up:**

Did you use extra space?  
A straight forward solution using O(*mn*) space is probably a bad idea.  
A simple improvement uses O(*m* + *n*) space, but still not the best solution.  
Could you devise a constant space solution?

Use the first row and first column as markers. Mark respective row and col marker as 0 if matrix[i][j] = 0, indicating later this row and col must be marked as 0. Because the first row and col status has been altered, use two variables to track down their own status.

1. **class** Solution {
2. **public** **void** setZeroes(**int**[][] matrix) {
3. **int** row = matrix.length;
4. **if** (row == 0) **return**;
5. **int** col = matrix[0].length;
6. **boolean** firstRow = **false**;
7. **boolean** firstCol = **false**;
8. **for** (**int** i = 0 ; i < row ; i++ ) {
9. **for** (**int** j = 0; j < col ; j++) {
10. **if** (matrix[i][j] == 0) {
11. **if** (i == 0) {
12. firstRow = **true**;
13. }
14. **if** (j == 0) {
15. firstCol = **true**;
16. }
17. matrix[i][0] = 0;
18. matrix[0][j] = 0;
19. }
20. }
21. }
22. **for** (**int** i = 1; i < row ; i++) {
23. **for** (**int** j = 1 ; j < col ; j++) {
24. **if** (matrix[i][0] == 0 || matrix[0][j] == 0) {
25. matrix[i][j] = 0;
26. }
27. }
28. }
29. **if** (firstRow) {
30. **for** (**int** i = 0 ; i < col ; i++) {
31. matrix[0][i] = 0;
32. }
33. }
34. **if** (firstCol) {
35. **for** (**int** i = 0 ; i < row ; i++) {
36. matrix[i][0] = 0;
37. }
38. }
40. }
41. }

74. Search a 2D Matrix

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

[

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

Given **target** = 3, return true.

Treat the 2D matrix as array and then use Binary search.

1. **class** Solution {
2. **public** **boolean** searchMatrix(**int**[][] matrix, **int** target) {
3. **if** (matrix == **null** || matrix.length == 0) {
4. **return** **false**;
5. }
6. **int** row = matrix.length;
7. **int** col = matrix[0].length;
8. **int** start = 0, end = row\*col - 1;
9. **while**(start <= end) {
10. **int** mid = start + (end - start)/2;
11. **int** value = matrix[mid / col][mid % col];
12. **if** (value == target) {
13. **return** **true**;
14. } **else** **if** (value > target) {
15. end = mid - 1;
16. } **else** {
17. start = mid + 1;
18. }
19. }
20. **return** **false**;
21. }
22. }

77. Combinations

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

For example,  
If *n* = 4 and *k* = 2, a solution is:

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],

]

Backtracking problem

1. **class** Solution {
2. **public** List<List<Integer>> combine(**int** n, **int** k) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
5. backTracking(ans,**new** ArrayList<Integer>(),n,k,1);
6. **return** ans;
7. }
8. **private** **void** backTracking(List<List<Integer>> ans,List<Integer> subList, **int** n, **int** k, **int** start) {
9. **if** (subList.size() == k) {
10. ans.add(**new** ArrayList(subList));
11. } **else** {
12. **for** (**int** i=start;i <= n ;i++) {
13. subList.add(i);
14. backTracking(ans,subList,n,k,i+1);
15. subList.remove(subList.size()-1);
16. }
17. }
18. }
19. }

78. Subsets

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

**Note:** The solution set must not contain duplicate subsets.

For example,  
If ***nums*** = [1,2,3], a solution is:

[

[3],

[1],

[2],

[1,2,3],

[1,3],

[2,3],

[1,2],

[]

]

回溯法应用：

1. **class** Solution {
2. **public** List<List<Integer>> subsets(**int**[] nums) {
3. List<List<Integer>> ans = **new** ArrayList<List<Integer>>();
4. **if** (nums == **null**) {
5. **return** **null**;
6. }
7. **if** (nums.length == 0) {
8. **return** ans;
9. }
10. Arrays.sort(nums);
12. backTracking(nums,**new** ArrayList(),ans,0);
13. **return** ans;
15. }
16. **private** **void** backTracking(**int**[] nums, List<Integer> subList,List<List<Integer>> ans,**int** startIndex) {
17. ans.add(**new** ArrayList(subList));
18. **for** (**int** i=startIndex;i<nums.length;i++) {
19. subList.add(nums[i]);
20. backTracking(nums,subList,ans,i+1);  **// 注意这里不是startIndex+1**
21. subList.remove(subList.size()-1);
22. }
23. }
25. }

79. Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,  
Given **board** =

[

['A','B','C','E'],

['S','F','C','S'],

['A','D','E','E']

]

**word** = "ABCCED", -> returns true,  
**word** = "SEE", -> returns true,  
**word** = "ABCB", -> returns false.

Backtracking: use a 2D array to track down the status of each board (whether used or not), do not forget to restore the status to originals. (Pith of backtrack)

1. **class** Solution {
2. **public** **boolean** exist(**char**[][] board, String word) {
3. **if** (board == **null** || board.length == 0 || word == **null** || word.length() == 0) **return** **false**;
5. **int** row = board.length;
6. **int** col = board[0].length;
7. **boolean**[][] used = **new** **boolean**[row][col];
8. **for** (**int** i = 0; i < row; i++) {
9. **for** (**int** j = 0;j < col; j++) {
10. **if** (backTrack(board, word.toCharArray(),used,i,j,0)) {
11. **return** **true**;
12. }
13. }
14. }
15. **return** **false**;
17. }
18. **private** **boolean** backTrack (**char**[][] board, **char**[] word,**boolean**[][] used, **int** row, **int** col, **int** index)    {
19. **if** (index == word.length) **return** **true**;
20. **if** (row < 0 || col < 0 || row >= board.length || col >= board[0].length) {
21. **return** **false**;
22. }
23. **if** (used[row][col] == **true** || board[row][col] != word[index]) {
24. **return** **false**;
25. }
26. used[row][col] = **true**;
28. // right
29. **boolean** right = backTrack(board, word, used,row, col+1,index+1);
30. **if** (right) **return** **true**;
32. // left
33. **boolean** left = backTrack(board, word, used,row, col-1,index+1);
34. **if** (left) **return** **true**;
36. //up
37. **boolean** up = backTrack(board, word, used,row-1, col,index+1);
38. **if** (up) **return** **true**;
40. // down
41. **boolean** down = backTrack(board, word, used,row+1, col,index+1);
42. **if** (down) **return** **true**;

45. used[row][col] = **false**;
47. **return** **false**;
49. }
50. }

80. Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":  
What if duplicates are allowed at most *twice*?

For example,  
Given sorted array *nums* = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of *nums* being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length.

First time I used a map to keep track down how many times a number has occurred in the array which totally ignores the array is sorted and is a bad idea.

1. **class** Solution {
2. **public** **int** removeDuplicates(**int**[] nums) {
3. **int** res = 0;
4. Map<Integer,Integer> map = **new** HashMap<Integer,Integer>();
5. **for** (**int** i = 0; i < nums.length; i++) {
6. **if** (map.getOrDefault(nums[i],0) < 2) {
7. nums[res++] = nums[i];
8. map.put(nums[i],map.getOrDefault(nums[i],0)+1);
10. }
11. }
12. **return** res;
13. }
14. }

Actually one very concise is to compare the current number with the second former one.

1. **class** Solution {
2. **public** **int** removeDuplicates(**int**[] nums) {
3. **int** i = 0;
4. **for** (**int** n : nums)
5. **if** (i < 2 || n > nums[i-2])
6. nums[i++] = n;
7. **return** i;
8. }
9. }

81. Search in Rotated Sorted Array II

*Follow up* for "Search in Rotated Sorted Array":  
What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Write a function to determine if a given target is in the array.

The array may contain duplicates.

The only case needs to handle is (1,1,1,3,1) where start and mid are the same. In this case, let start point increase by 1.

1. **class** Solution {
2. **public** **boolean** search(**int**[] nums, **int** target) {
3. **if** (nums == **null** || nums.length == 0) **return** **false**;
4. **int** low = 0;
5. **int** high = nums.length-1;
6. **while** (low + 1 < high) {
7. **int** mid = low + (high - low)/2;
9. **if** (nums[mid] == target) {
10. **return** **true**;
11. } **else** **if** (nums[low] < nums[mid]){
12. **if** (nums[low] <= target && target <= nums[mid]) {
13. high = mid;
14. } **else** {
15. low = mid;
16. }
17. } **else** **if** (nums[low] > nums[mid]){
18. **if** (nums[mid] <= target && target <= nums[high]) {
19. low =  mid;
20. } **else** {
21. high = mid;
22. }
23. } **else** {
24. low++;
25. }
26. }
27. **if** (nums[low] == target || nums[high] == target) **return** **true**;
28. **return** **false**;
29. }
30. }

82. Remove Duplicates from Sorted List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.

For example,  
Given 1->2->3->3->4->4->5, return 1->2->5.  
Given 1->1->1->2->3, return 2->3.

In my own method, I used a flag to mark if there exists duplicate. In fact, it is not necessary by checking if the current node is the very next of the previous one. Since if duplicate values exist, it cannot be the exact next one.

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **class** Solution {
10. **public** ListNode deleteDuplicates(ListNode head) {
11. **if** (head == **null** || head.next == **null**) **return** head;
12. ListNode newHead = **new** ListNode(0);
13. newHead.next = head;
14. ListNode prev = newHead;
16. **while** (head!= **null**) {
17. **while**(head.next != **null** && head.next.val == head.val) {
18. head = head.next;
19. }
20. **if** (prev.next == head) {
21. prev = prev.next;
23. } **else** {
24. prev.next = head.next;
25. }
26. head = head.next;
27. }
28. **return** newHead.next;
29. }
30. }

86. Partition List

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,  
Given 1->4->3->2->5->2 and *x* = 3,  
return 1->2->2->4->3->5.

Think of dummy head when it comes to link list. One tricky is Case (1,1) and x is 2, this case may cause exception. Lines 21 -26 is setting left boundary.

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **class** Solution {
10. **public** ListNode partition(ListNode head, **int** x) {
11. **if** (head == **null** || head.next == **null**) **return** head;
13. ListNode dummy = **new** ListNode(0);
14. dummy.next = head;
15. ListNode curr = head;
16. ListNode prev = dummy;
17. ListNode left = dummy;

20. **while**(curr != **null**) {
21. **if** (left == prev) {
22. **if** ( curr.val < x) {
23. left = left.next;
24. }
25. prev = curr;
26. curr = curr.next;
27. } **else** {
28. **if** (curr.val < x) {
29. prev.next = curr.next;
30. curr.next = left.next;
31. left.next = curr;
32. left = left.next;
33. curr = prev.next;
34. } **else** {
35. prev = curr;
36. curr = curr.next;
37. }
39. }
41. }
43. **return** dummy.next;
45. }
46. }

89. Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer *n* representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given *n* = 2, return [0,1,3,2]. Its gray code sequence is:

00 - 0

01 - 1

11 - 3

10 - 2

Grey code formula

G(i) = i ^ (i/2)

1. **class** Solution {
2. **public** List<Integer> grayCode(**int** n) {
3. List<Integer> ans = **new** ArrayList<Integer>();
4. **for** (**int** i = 0 ;i < 1 << n ;i++) {
5. ans.add(i^(i >> 1));
6. }
7. **return** ans;
8. }
9. }

90. Subsets II

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets (the power set).

**Note:** The solution set must not contain duplicate subsets.

For example,  
If ***nums*** = [1,2,2], a solution is:

[

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

]

**Notice how to remove duplicates!**

1. **class** Solution {
2. **public** List<List<Integer>> subsetsWithDup(**int**[] nums) {
3. List<List<Integer>> res = **new** ArrayList<>();
4. **if** (nums == **null** || nums.length == 0) {
5. **return** res;
6. }
7. Arrays.sort(nums);
8. backTracking(nums,**new** ArrayList(),0,res);
9. **return** res;
10. }
12. **public** **void** backTracking(**int**[] nums,ArrayList<Integer> subList,**int** start,List<List<Integer>> res){
13. res.add(**new** ArrayList(subList));
14. **for** (**int** i = start;i < nums.length;i++) {
15. ***if (i != start && nums[i] == nums[i-1]) {***
16. ***continue;***
17. ***}***
18. subList.add(nums[i]);
19. backTracking(nums,subList,i+1,res);
20. subList.remove(subList.size()-1);
21. }
22. }

}

91. Decode Ways(重要)

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1

'B' -> 2

...

'Z' -> 26

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,  
Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

**DP problem**

1. **class** Solution {
2. **public** **int** numDecodings(String s) {
3. **if** (s == **null** || s.length() == 0) {
4. **return** 0;
5. }
6. **int** len = s.length();
7. **int**[] dp = **new** **int**[len+1];
8. dp[0] = 1;
9. dp[1] = (s.charAt(0) == '0'?0:1);
10. **for** (**int** i = 2;i <= len;i++) {
11. **int** first = Integer.valueOf(s.substring(i-1,i));
12. **int** second = Integer.valueOf(s.substring(i-2,i));
13. **if** (first <= 9 && first >= 1) {
14. dp[i] += dp[i-1];
15. }
16. **if** (second <= 26 && second >= 10) {
17. dp[i] += dp[i-2];
18. }
19. }
20. **return** dp[len];
21. }
22. }

92. Reverse Linked List II

Reverse a linked list from position *m* to *n*. Do it in-place and in one-pass.

For example:  
Given 1->2->3->4->5->NULL, *m* = 2 and *n* = 4,

return 1->4->3->2->5->NULL.

**Note:**  
Given *m*, *n* satisfy the following condition:  
1 ≤ *m* ≤ *n* ≤ length of list.

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **class** Solution {
10. **public** ListNode reverseBetween(ListNode head, **int** m, **int** n) {
12. ListNode newHead = **new** ListNode(0);
13. newHead.next = head;
14. ListNode prev = newHead;
16. **for** (**int** i = 1;i < m;i++) {
17. prev = prev.next;
18. }
19. ListNode sta = prev.next;
20. ListNode nex = sta.next;

23. **for** (**int** i = 1;i <= n-m;i++) {
24. sta.next = nex.next;
25. nex.next = prev.next;
26. prev.next = nex;
27. nex = sta.next;
28. }
30. **return** newHead.next;
31. }
32. }

93. Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:  
Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

1. **class** Solution {
2. **public** List<String> restoreIpAddresses(String s) {
3. List<String> res = **new** ArrayList<>();
4. backTracking(res,s,0,0,"");
5. **return** res;
6. }
7. **public** **void** backTracking(List<String> res, String s, **int** count,**int** start,String curr){
8. **if** (count > 4) **return**;
9. **if** (count == 4 && start == s.length()) {
10. res.add(curr);
11. **return**;
12. }
13. **for** (**int** i = 1;i <= 3;i++) {
14. **if** (start + i > s.length()) **break**;
15. String temp = s.substring(start,start+i);
16. **if** ((temp.startsWith("0") && temp.length() > 1) || (i == 3 && Integer.parseInt(temp) >=256)) **continue**;
17. backTracking(res,s,count+1,start+i,curr+temp+(count==3?"":"."));
18. }
19. }
20. }

95. Unique Binary Search Trees II

Given an integer *n*, generate all structurally unique **BST's** (binary search trees) that store values 1...*n*.

For example,  
Given *n* = 3, your program should return all 5 unique BST's shown below.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

递归：

从最小值到最大值，每个点以此作为根结点，构建左右子树

判断左右子树为空的情况，分别把左右子数的子集与根结点构建新树，加入结果

base case: 最小值大于最大值，此时return

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** List<TreeNode> generateTrees(**int** n) {
12. **return** generateTreesHelper(1,n);
13. }
15. **private** List<TreeNode> generateTreesHelper(**int** min,**int** max){
16. List<TreeNode> res = **new** ArrayList<>();
17. **if** (min > max) **return** res;
18. **for** (**int** root = min; root <= max; root++) {
19. List<TreeNode> leftList = generateTreesHelper(min,root-1);
20. List<TreeNode> rightList = generateTreesHelper(root+1,max);
21. **if** (leftList.size() == 0 && rightList.size() == 0){
22. TreeNode rt = **new**  TreeNode(root);
23. res.add(rt);
24. } **else** **if** (leftList.size() == 0) {
25. **for** (TreeNode right:rightList){
26. TreeNode rt = **new** TreeNode(root);
27. rt.right = right;
28. res.add(rt);
29. }
30. } **else** **if** (rightList.size() == 0) {
31. **for** (TreeNode left:leftList){
32. TreeNode rt = **new** TreeNode(root);
33. rt.left = left;
34. res.add(rt);
35. }
36. } **else** {
37. **for** (TreeNode right:rightList){
38. **for** (TreeNode left:leftList){
39. TreeNode rt = **new** TreeNode(root);
40. rt.left = left;
41. rt.right = right;
42. res.add(rt);
43. }
44. }
45. }
46. }
47. **return** res;
48. }
49. }

96. Unique Binary Search Trees

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?

For example,  
Given *n* = 3, there are a total of 5 unique BST's.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

动态规划：

数组rootNums为该结点数目下binary tree的数目

base case：rootNums[0] = rootNums[1] = 1

状态转移方程：rootNums[i] += rootNums[j]\*rootNums[i-1-j];

1. **class** Solution {
2. **public** **int** numTrees(**int** n) {
3. **int**[] rootNums = **new** **int**[n+1];
4. rootNums[0] = rootNums[1] = 1;
5. **for** (**int** i = 2;i <= n;i++) {
6. **for** (**int** j = 0; j < i; j++) {
7. rootNums[i] += rootNums[j]\*rootNums[i-1-j];
8. }
9. }
10. **return** rootNums[n];
11. }
12. }

98. Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

**Example 1:**

2

/ \

1 3

Binary tree [2,1,3], return true.

**Example 2:**

1

/ \

2 3

Binary tree [1,2,3], return false.

最大值 最小值要用long型以免越界

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** **boolean** isValidBST(TreeNode root) {
12. **return** isValidBSTHelper(root,Long.MIN\_VALUE,Long.MAX\_VALUE);
13. }
15. **private** **boolean** isValidBSTHelper(TreeNode root, **long** min, **long** max) {
16. **if** (root == **null**) **return** **true**;
17. **if** (root.val >= max || root.val <= min) {
18. **return** **false**;
19. }
20. **return** isValidBSTHelper(root.left, min, root.val) && isValidBSTHelper(root.right,root.val,max);
21. }
22. }

迭代用stack：

1. **public** **boolean** isValidBST(TreeNode root) {
2. **if** (root == **null**) **return** **true**;
3. Stack<TreeNode> stack = **new** Stack<>();
4. TreeNode pre = **null**;
5. **while** (root != **null** || !stack.isEmpty()) {
6. **while** (root != **null**) {
7. stack.push(root);
8. root = root.left;
9. }
10. root = stack.pop();
11. **if**(pre != **null** && root.val <= pre.val) **return** **false**;
12. pre = root;
13. root = root.right;
14. }
15. **return** **true**;
16. }

102. Binary Tree Level Order Traversal

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

For example:  
Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its level order traversal as:

[

[3],

[9,20],

[15,7]

]

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** List<List<Integer>> levelOrder(TreeNode root) {
12. List<List<Integer>> res = **new** ArrayList<>();
13. Queue<TreeNode> queue = **new** LinkedList<>();
14. **if** (root == **null**) **return** res;
15. queue.offer(root);
16. queue.offer(**null**);
17. List<Integer> subList = **new** ArrayList<>();
18. **while**(!queue.isEmpty()) {
19. TreeNode curr = queue.poll();
20. **if** (curr != **null**) {
21. subList.add(curr.val);
22. **if** (curr.left != **null**) {
23. queue.offer(curr.left);
24. }
25. **if** (curr.right != **null**) {
26. queue.offer(curr.right);
27. }
28. } **else** {
29. res.add(subList);
30. **if** (queue.isEmpty()) **break**;
31. subList = **new** ArrayList<>();
32. queue.offer(**null**);
33. }
34. }
35. **return** res;
36. }
37. }

103. Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:  
Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its zigzag level order traversal as:

[

[3],

[20,9],

[15,7]

]

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** List<List<Integer>> zigzagLevelOrder(TreeNode root) {
12. List<List<Integer>> res = **new** ArrayList<>();
13. **if** (root == **null**) **return** res;
14. **boolean** changeOrder = **false**;
15. Queue<TreeNode> queue = **new** LinkedList<>();
16. Stack<TreeNode> stack = **new** Stack<>();
17. queue.offer(root);
18. queue.offer(**null**);
19. List<Integer> subList = **new** ArrayList<>();
20. **while**(!queue.isEmpty()) {
21. TreeNode curr = queue.poll();
22. **if** (curr != **null**) {
23. **if** (!changeOrder) {
24. subList.add(curr.val);
25. } **else** {
26. stack.push(curr);
27. }
28. **if** (curr.left != **null**) queue.offer(curr.left);
29. **if** (curr.right != **null**) queue.offer(curr.right);
30. } **else** {
31. **while** (!stack.isEmpty()) {
32. subList.add(stack.pop().val);
33. }
34. res.add(subList);
35. **if** (queue.isEmpty()) **break**;
36. changeOrder = !changeOrder;
37. subList = **new** ArrayList<>();
38. queue.offer(**null**);
39. }
40. }
41. **return** res;
42. }
43. }

105. Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** TreeNode buildTree(**int**[] preorder, **int**[] inorder) {
12. **if** (preorder == **null** || preorder.length == 0 || preorder.length != inorder.length) **return** **null**;
13. **return** helper(preorder, 0, inorder,0, inorder.length - 1);


17. }
18. **public** TreeNode helper(**int**[] preorder,**int** preStart,**int**[] inorder, **int** inStart, **int** inEnd) {
20. **if** (preStart > preorder.length || inStart > inEnd) **return** **null**;
21. TreeNode root = **new** TreeNode(preorder[preStart]);
23. **int** i = inStart;
24. **while** (i <= inEnd) {
25. **if** (inorder[i] == root.val) {
26. **break**;
27. } **else** {
28. i++;
29. }
30. }

33. root.left = helper(preorder, preStart+1, inorder, inStart, i-1);
34. root.right = helper(preorder, preStart+(i-inStart+1), inorder,i+1,inEnd);
35. **return** root;
36. }

106. Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

**Note:**  
You may assume that duplicates do not exist in the tree.

For example, given

inorder = [9,3,15,20,7]

postorder = [9,15,7,20,3]

Return the following binary tree:

3

/ \

9 20

/ \

15 7

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** TreeNode buildTree(**int**[] inorder, **int**[] postorder) {
12. **if** (inorder == **null** || inorder.length == 0) **return** **null**;
13. **return** buildTreeHelper(inorder,0,inorder.length-1,postorder,0,postorder.length-1);
15. }
16. **private** TreeNode buildTreeHelper(**int**[] inorder, **int** inStart, **int** inEnd, **int**[] postorder, **int** postStart,**int** postEnd){
17. **if** (inStart > inEnd || postStart > postEnd || inEnd >= inorder.length || postEnd >= postorder.length) **return** **null**;
18. TreeNode root = **new** TreeNode(postorder[postEnd]);
19. **int** i = inStart;
20. **while**(i <= inEnd) {
21. **if** (inorder[i] == root.val){
22. **break**;
23. } **else** {
24. i++;
25. }
26. }
27. root.left = buildTreeHelper(inorder,inStart,i-1,postorder,postStart,postStart+(i-inStart-1));
28. root.right = buildTreeHelper(inorder,i+1,inEnd,postorder,postStart+(i-inStart),postEnd-1);
29. **return** root;
30. }
31. }

114. Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example,  
Given

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

6

This is to reverse preorder. Everytime visting a node, make previous visted node as the right child and left child as null, change previous one to current node.

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. TreeNode prev = null;
12. **public** **void** flatten(TreeNode root) {
13. **if** (root == null) **return**;
14. flatten(root.right);
15. flatten(root.left);
16. root.right = prev;
17. root.left = null;
18. prev = root;
19. }
20. }

116. Populating Next Right Pointers in Each Node

Given a binary tree

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**Note:**

* You may only use constant extra space.
* You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,  
Given the following perfect binary tree,

1

/ \

2 3

/ \ / \

4 5 6 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

1. /\*\*
2. \* Definition for binary tree with next pointer.
3. \* public class TreeLinkNode {
4. \*     int val;
5. \*     TreeLinkNode left, right, next;
6. \*     TreeLinkNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** **void** connect(TreeLinkNode root) {
11. TreeLinkNode start = root;
12. **while**(start != **null**) {
13. TreeLinkNode curr = start;
14. **while** (curr != **null**) {
15. **if** (curr.left != **null**) {
16. curr.left.next = curr.right;
17. }
18. **if** (curr.next != **null** && curr.right != **null**) {
19. curr.right.next = curr.next.left;
20. }
21. curr = curr.next;
22. }
23. start = start.left;
24. }
25. }
26. }

117. Populating Next Right Pointers in Each Node II

Follow up for problem "*Populating Next Right Pointers in Each Node*".

What if the given tree could be any binary tree? Would your previous solution still work?

**Note:**

* You may only use constant extra space.

For example,  
Given the following binary tree,

1

/ \

2 3

/ \ \

4 5 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ \

4-> 5 -> 7 -> NULL

1. /\*\*
2. \* Definition for binary tree with next pointer.
3. \* public class TreeLinkNode {
4. \*     int val;
5. \*     TreeLinkNode left, right, next;
6. \*     TreeLinkNode(int x) { val = x; }
7. \* }
8. \*/
9. **public** **class** Solution {
10. **public** **void** connect(TreeLinkNode root) {
11. TreeLinkNode childHead = **null**;
12. TreeLinkNode child = root;
13. TreeLinkNode parent = **null**;
14. **while** (child != **null**) {
15. **while**(child != **null**) {
16. **if** (child.left != **null**) {
17. **if** (parent != **null**) {
18. parent.next = child.left;
19. } **else** {
20. childHead = child.left;
21. }
22. parent = child.left;
23. }
24. **if** (child.right != **null**) {
25. **if** (parent != **null**) {
26. parent.next = child.right;
27. } **else** {
28. childHead = child.right;
29. }
30. parent = child.right;
31. }
32. child = child.next;
33. }
34. child = childHead;
35. parent = **null**;
36. childHead = **null**;
37. }
38. }
39. }

120. Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

[

[2],

[3,4],

[6,5,7],

[4,1,8,3]

]

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

**Note:**  
Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

1. **class** Solution {
2. **public** **int** minimumTotal(List<List<Integer>> triangle) {
3. **if** (triangle == **null** || triangle.size() == 0) **return** 0;
5. **int** size = triangle.size();
6. **int**[] min = **new** **int**[size];
7. **for** (**int** i = 0; i < size; i++) {
8. min[i] = triangle.get(size-1).get(i);
9. }
11. **for** (**int** i = size - 2; i >= 0; i--) {
12. **for** (**int** j = 0; j <= i;j++) {
13. min[j] = triangle.get(i).get(j) + Math.min(min[j], min[j+1]);
14. }
15. }
16. **return** min[0];
17. }
18. }

127. Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

For example,

Given:  
*beginWord* = "hit"  
*endWord* = "cog"  
*wordList* = ["hot","dot","dog","lot","log","cog"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",  
return its length 5.

**Note:**

* Return 0 if there is no such transformation sequence.
* All words have the same length.
* All words contain only lowercase alphabetic characters.
* You may assume no duplicates in the word list.
* You may assume *beginWord* and *endWord* are non-empty and are not the same.

1. **class** Solution {
2. **public** **int** ladderLength(String beginWord, String endWord, List<String> wordList) {
3. Set<String> reached = **new** HashSet<String>();
4. Set<String> list = **new** HashSet<String>();
5. **for** (**int** i =0;i<wordList.size();i++) {
6. list.add(wordList.get(i));
7. }
8. reached.add(beginWord);
9. **int** distance = 1;
10. **while**(!reached.isEmpty()) {
11. Set<String> toAdd = **new** HashSet<String>();
12. **for** (String reach:reached) {
13. **for** (**int** i = 0;i < reach.length();i++) {
14. **char**[] chars = reach.toCharArray();
15. **for** (**char** c = 'a';c<='z';c++) {
16. chars[i] = c;
17. String word = **new** String(chars);
19. **if** (list.contains(word)) {
20. **if** (word.equals(endWord)) {
21. **return** distance + 1;
22. }
23. toAdd.add(word);
24. list.remove(word);
25. }
26. }
27. }
28. }
29. reached = toAdd;
31. distance++;
32. }
34. **return** 0;
35. }
36. }

129. Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,

1

/ \

2 3

The root-to-leaf path 1->2 represents the number 12.  
The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
12. **public** **int** sumNumbers(TreeNode root) {
13. **return** sumNumbersHelper(root,0);
14. }
15. **public** **int** sumNumbersHelper(TreeNode root, **int** value) {
16. **if** (root == **null**) **return** 0;
17. **if** (root.left == **null** && root.right == **null**) **return** 10 \* value + root.val;
18. **return** sumNumbersHelper(root.left, 10 \* value + root.val) + sumNumbersHelper(root.right, 10 \* value + root.val);
19. }
20. }

130. Surrounded Regions

Given a 2D board containing 'X' and 'O' (the **letter** O), capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

X X X X

X O O X

X X O X

X O X X

After running your function, the board should be:

X X X X

X X X X

X X X X

X O X X

1. **class** Solution {
2. **public** **void** solve(**char**[][] board) {
3. **if** (board == **null** || board.length == 0) **return**;
4. **if** (board.length < 3 || board[0].length < 3) **return**;
5. **int** r = board.length;
6. **int** c = board[0].length;
7. **for** (**int** i = 0; i < r;i++) {
8. **if** (board[i][0] == 'O') {
9. helperDFS(board, i, 0);
10. }
11. **if** (board[i][c-1] == 'O') {
12. helperDFS(board, i, c - 1);
13. }
14. }
15. **for** (**int** j = 1;j < c-1; j++) {
16. **if** (board[0][j] == 'O') {
17. helperDFS(board, 0,j);
18. }
19. **if** (board[r-1][j] == 'O') {
20. helperDFS(board, r-1, j);
21. }
22. }
24. **for** (**int** i = 0; i < r;i++) {
25. **for** (**int** j = 0; j < c;j++) {
26. **if** (board[i][j] == 'O') {
27. board[i][j] = 'X';
28. }
29. **if** (board[i][j] == '\*') {
30. board[i][j] = 'O';
31. }
33. }
34. }
35. }
37. **public** **void** helperDFS(**char**[][] board, **int** i, **int** j) {
38. **if** (i < 0 || i >= board.length || j < 0 || j >= board[0].length || board[i][j] != 'O') {
39. **return**;
40. }
41. board[i][j] = '\*';
42. helperDFS(board, i + 1, j);
43. helperDFS(board, i - 1, j);
44. helperDFS(board, i, j + 1);
45. helperDFS(board, i, j - 1);
46. }
47. }

131. Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

**Example:**

**Input:** "aab"

**Output:**

[

["aa","b"],

["a","a","b"]

]

1. **class** Solution {
2. **public** List<List<String>> partition(String s) {
3. List<List<String>> res = **new** ArrayList<>();
4. **if** (s == **null** || s.length() == 0) **return** res;
5. partitionHelper(s, res, **new** ArrayList<>());
6. **return** res;
7. }
8. **public** **void** partitionHelper(String s, List<List<String>> res,List<String> subList) {
9. **if** (s.length() == 0) {
10. res.add(**new** ArrayList(subList));
11. **return**;
12. }
13. **for** (**int** i = 0; i < s.length();i++) {
14. **if** (isPalin(s.substring(0,i+1))) {
15. subList.add(s.substring(0,i+1));
16. partitionHelper(s.substring(i+1),res,subList);
17. subList.remove(subList.size()-1);
19. }
20. }
21. }
23. **public** **boolean** isPalin(String s) {
24. **for** (**int** i = 0; i < s.length()/2;i++) {
25. **if** (s.charAt(i) != s.charAt(s.length() - 1 - i)) {
26. **return** **false**;
27. }
28. }
29. **return** **true**;
30. }
31. }

133. Clone Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

**OJ's undirected graph serialization:**

Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

1. First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
2. Second node is labeled as 1. Connect node 1 to node 2.
3. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

1

/ \

/ \

0 --- 2

/ \

\\_/

1. /\*\*
2. \* Definition for undirected graph.
3. \* class UndirectedGraphNode {
4. \*     int label;
5. \*     List<UndirectedGraphNode> neighbors;
6. \*     UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }
7. \* };
8. \*/
9. **public** **class** Solution {
11. **public** UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
12. HashMap<Integer, UndirectedGraphNode> map = **new** HashMap<>();
13. **return** dfs(node,map);
14. }
15. **public** UndirectedGraphNode dfs(UndirectedGraphNode node, HashMap<Integer, UndirectedGraphNode> map) {
16. **if** (node == **null**) **return** **null**;
17. **if** (map.containsKey(node.label)) {
18. **return** map.get(node.label);
19. }
20. UndirectedGraphNode clone = **new** UndirectedGraphNode(node.label);
21. map.put(clone.label, clone);
22. **for** (UndirectedGraphNode neighbor: node.neighbors) {
23. clone.neighbors.add(dfs(neighbor,map));
24. }
25. **return** clone;
26. }
27. }

134. Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

**Note:**

* If there exists a solution, it is guaranteed to be unique.
* Both input arrays are non-empty and have the same length.
* Each element in the input arrays is a non-negative integer.

**Example 1:**

**Input:**

gas = [1,2,3,4,5]

cost = [3,4,5,1,2]

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

如果一个数组总和非负，那么一定可以找到一个数，从此开始绕数组一圈，累加和一直都是非负

1. **class** Solution {
2. **public** **int** canCompleteCircuit(**int**[] gas, **int**[] cost) {
3. **int** total = 0;
4. **int** sum = 0;
5. **int** start = 0;
6. **for** (**int** i = 0; i < gas.length;i++) {
7. total += gas[i] - cost[i];
8. **if** (sum < 0) {
9. sum = gas[i] - cost[i];
10. start = i;
11. } **else** {
12. sum += gas[i] - cost[i];
13. }
14. }
15. **return** total >= 0?start:-1;
16. }
17. }

138. Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

1. **public** **class** Solution {
2. **public** RandomListNode copyRandomList(RandomListNode head) {
3. Map<RandomListNode,RandomListNode> map = **new** HashMap<>();
4. RandomListNode cur = head;
5. **while**(cur != **null**) {
6. map.put(cur, **new** RandomListNode(cur.label));
7. cur = cur.next;
8. }
9. cur = head;
10. **while**(cur != **null**) {
11. map.get(cur).next = map.get(cur.next);
12. map.get(cur).random = map.get(cur.random);
13. cur = cur.next;
14. }
15. **return** map.get(head);
16. }
17. }

139. Word Break

Given a **non-empty** string s and a dictionary wordDict containing a list of **non-empty** words, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

**Note:**

* The same word in the dictionary may be reused multiple times in the segmentation.
* You may assume the dictionary does not contain duplicate words.

**Example 1:**

**Input:** s = "leetcode", wordDict = ["leet", "code"]

**Output:** true

**Explanation:** Return true because "leetcode" can be segmented as "leet code".

**Example 2:**

**Input:** s = "applepenapple", wordDict = ["apple", "pen"]

**Output:** true

**Explanation:** Return true because "applepenapple" can be segmented as "apple pen apple".

  Note that you are allowed to reuse a dictionary word.

**Example 3:**

**Input:** s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]

**Output:** false

1. **class** Solution {
2. **public** **boolean** wordBreak(String s, List<String> wordDict) {
3. **boolean**[] dp = **new** **boolean**[s.length()+1];
4. dp[0] = **true**;
6. **for** (**int** i = 1; i <= s.length();i++) {
7. **for** (**int** j = 0; j < i;j++) {
8. **if** (dp[j] && wordDict.contains(s.substring(j,i))) {
9. dp[i] =**true**;
10. **break**;
11. }
12. }
13. }
14. **return** dp[s.length()];
15. }
16. }

142. Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

**Note:** Do not modify the linked list.

1. **public** **class** Solution {
2. **public** ListNode detectCycle(ListNode head) {
3. **if** (head == **null** || head.next == **null**) {
4. **return** **null**;
5. }
6. ListNode fast = head;
7. ListNode slow = head;
8. **while**(fast != **null**) {
9. fast = fast.next;
10. **if** (fast != **null**) {
11. fast = fast.next;
12. slow = slow.next;
13. }
14. **if** (fast == slow) {
15. ListNode slow2 = head;
16. **while**(slow != slow2) {
17. slow = slow.next;
18. slow2 = slow2.next;
19. }
20. **return** slow;
21. }
22. }
23. **return** **null**;
24. }
25. }

148. Sort List

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

1. /\*\*
2. \* Definition for singly-linked list.
3. \* public class ListNode {
4. \*     int val;
5. \*     ListNode next;
6. \*     ListNode(int x) { val = x; }
7. \* }
8. \*/
9. **class** Solution {
10. **public** ListNode sortList(ListNode head) {
11. **if** (head == **null** || head.next == **null**) **return** head;
13. ListNode prev = **null**, slow = head, fast = head;
14. **while**(fast != **null** && fast.next != **null**) {
15. prev = slow;
16. slow = slow.next;
17. fast = fast.next.next;
18. }
19. prev.next = **null**;
21. ListNode l1 = sortList(head);
22. ListNode l2 = sortList(slow);
23. **return** merge(l1, l2);
24. }
26. **public** ListNode merge(ListNode l1, ListNode l2) {
27. ListNode res = **new** ListNode(0);
28. ListNode cur = res;
29. **while**(l1 != **null** && l2 != **null**) {
30. **if** (l1.val < l2.val) {
31. cur.next = l1;
32. l1 = l1.next;
33. } **else** {
34. cur.next = l2;
35. l2 = l2.next;
36. }
37. cur = cur.next;
38. }
39. **if** (l1 != **null**) {
40. cur.next = l1;
41. } **else** {
42. cur.next = l2;
43. }
44. **return** res.next;
45. }
46. }

147. Insertion Sort List

1. **class** Solution {
2. **public** ListNode insertionSortList(ListNode head) {
3. **if** (head == **null** || head.next == **null**) **return** head;
4. ListNode newHead = **new** ListNode(0);
5. newHead.next = head;
6. ListNode curr = head;
7. ListNode temp = **null**;
8. ListNode prev = **null**;
9. **while**(curr != **null** && curr.next != **null**) {
10. **if** (curr.val <= curr.next.val) {
11. curr = curr.next;
12. } **else** {
13. temp = curr.next;
14. curr.next = temp.next;
15. prev = newHead;
16. **while**(prev.next.val <= temp.val) {
17. prev = prev.next;
18. }
19. temp.next = prev.next;
20. prev.next = temp;
21. }
23. }
24. **return** newHead.next;
25. }
26. }

150. Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in [Reverse Polish Notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation).

Valid operators are +, -, \*, /. Each operand may be an integer or another expression.

**Note:**

* Division between two integers should truncate toward zero.
* The given RPN expression is always valid. That means the expression would always evaluate to a result and there won't be any divide by zero operation.

**Example 1:**

**Input:** ["2", "1", "+", "3", "\*"]

**Output:** 9

**Explanation:** ((2 + 1) \* 3) = 9

**Example 2:**

**Input:** ["4", "13", "5", "/", "+"]

**Output:** 6

**Explanation:** (4 + (13 / 5)) = 6

**Example 3:**

**Input:** ["10", "6", "9", "3", "+", "-11", "\*", "/", "\*", "17", "+", "5", "+"]

**Output:** 22

**Explanation:**

((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5

= ((10 \* (6 / (12 \* -11))) + 17) + 5

= ((10 \* (6 / -132)) + 17) + 5

= ((10 \* 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

1. **class** Solution {
2. **public** **int** evalRPN(String[] tokens) {
3. Stack<Integer> stack = **new** Stack<>();
4. **for** (String s:tokens) {
5. **if** (s.equals("+")) {
6. stack.push(stack.pop() + stack.pop());
7. } **else** **if** (s.equals("-")) {
8. **int** a = stack.pop();
9. **int** b = stack.pop();
10. stack.push(b - a);
11. } **else** **if** (s.equals("\*")) {
12. stack.push(stack.pop() \* stack.pop());
13. } **else** **if** (s.equals("/")) {
14. **int** a = stack.pop();
15. **int** b = stack.pop();
16. stack.push(b/a);
17. } **else** {
18. stack.push(Integer.parseInt(s));
19. }
20. }
21. **return** stack.pop();
22. }
23. }

151. Reverse Words in a String

**Example:**

**Input:** "the sky is blue",

**Output:**"blue is sky the".

**Notes:**

* A word is defined as a sequence of non-space characters.
* Input string may contain leading or trailing spaces. However, your reversed string should not contain leading or trailing spaces.
* You need to reduce multiple spaces between two words to a single space in the reversed string.

1. **public** **class** Solution {
2. **public** String reverseWords(String s) {
3. StringBuilder res = **new** StringBuilder("");
4. String[] array = s.trim().split(" ");
5. **for** (**int** i = array.length - 1;i >= 0;i--) {
6. **if** (array[i].length() != 0) {
7. res.append(array[i]).append(" ");
8. }
9. }
10. **return** res.toString().trim();
11. }
12. }

152. Maximum Product Subarray

Given an integer array nums, find the contiguous subarray within an array (containing at least one number) which has the largest product.

**Example 1:**

**Input:** [2,3,-2,4]

**Output:** 6

**Explanation:** [2,3] has the largest sum = 6.

**Example 2:**

**Input:** [-2,0,-1]

**Output:** 0

**Explanation:** The result cannot be 2, because [-2,-1] is not a subarray.

1. **class** Solution {
2. **public** **int** maxProduct(**int**[] nums) {
3. **int** res = nums[0];
4. **int** max = nums[0];
5. **int** min = nums[0];
6. **for** (**int** i = 1; i < nums.length; i++) {
7. **int** temp = max;
8. max = Math.max(Math.max(max \* nums[i], min \* nums[i]), nums[i]);
9. min = Math.min(Math.min(temp \* nums[i], min \* nums[i]), nums[i]);
10. res = Math.max(res,max);
11. }
12. **return** res;
13. }
14. }

153. Find Minimum in Rotated Sorted Array

1. **class** Solution {
2. **public** **int** findMin(**int**[] nums) {
3. **int** left = 0;
4. **int** right = nums.length - 1;
5. **while**(left + 1 < right) {
6. **int** mid = (right - left)/2 + left;
7. **if** (nums[mid] < nums[right]) {
8. right = mid;
9. } **else** {
10. left = mid;
11. }
12. }
13. **if** (nums[left] < nums[right]) {
14. **return** nums[left];
15. } **else** {
16. **return** nums[right];
17. }
18. }
19. }

154. Find Minimum in Rotated Sorted Array II

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e.,  [0,1,2,4,5,6,7] might become  [4,5,6,7,0,1,2]).

Find the minimum element.

The array may contain duplicates.

**Example 1:**

**Input:** [1,3,5]

**Output:** 1

**Example 2:**

**Input:** [2,2,2,0,1]

**Output:** 0

1. **class** Solution {
2. **public** **int** findMin(**int**[] nums) {
3. **int** low = 0, high = nums.length - 1;
4. **while** (low + 1 < high) {
5. **int** mid = (high - low) / 2 + low;
6. **if** (nums[mid] < nums[high]) {
7. high = mid;
8. } **else** **if** (nums[mid] > nums[high]){
9. low = mid;
10. } **else** {
11. high --;
12. }
13. }
14. **if** (nums[low] < nums[high]) {
15. **return** nums[low];
16. } **else** {
17. **return** nums[high];
18. }
19. }
20. }

161. One Edit Distance

Given two strings ***s*** and ***t***, determine if they are both one edit distance apart.

**Note:**

There are 3 possiblities to satisify one edit distance apart:

1. Insert a character into **s** to get **t**
2. Delete a character from **s** to get **t**
3. Replace a character of **s** to get **t**
4. **class** Solution {
5. **public** **boolean** isOneEditDistance(String s, String t) {
6. **for** (**int** i = 0; i < Math.min(s.length(), t.length());i++) {
7. **if** (s.charAt(i) != t.charAt(i)) {
8. **if** (s.length() == t.length()) {
9. **return** (s.substring(i+1).equals(t.substring(i+1)));
10. } **else** **if** (s.length() > t.length()) {
11. **return** s.substring(i+1).equals(t.substring(i));
12. } **else** {
13. **return** t.substring(i+1).equals(s.substring(i));
14. }
15. }
16. }
17. **return** Math.abs(s.length() - t.length()) == 1;
18. }
19. }

162. Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array nums, where num[i] ≠ num[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞.

**Example 1:**

**Input:** **nums** = [1, 2, 3, 1]

**Output:** 2

**Explanation:** 3 is a peak element and your function should return the index number 2.

**Example 2:**

**Input:** **nums** = [1, 2, 1, 3, 5, 6, 4]

**Output:** 1 or 5

**Explanation:** Your function can return either index number 1 where the peak element is 2,

  or index number 5 where the peak element is 6.

1. **class** Solution {
2. **public** **int** findPeakElement(**int**[] nums) {
3. **int** left = 0;
4. **int** right = nums.length - 1;
5. **while** (left + 1 < right) {
6. **int** mid = (right - left) / 2 + left;
7. **if** (nums[mid] < nums[mid+1]) {
8. left = mid;
9. } **else** {
10. right = mid;
11. }
12. }
13. **if** (nums[left] > nums[right]) {
14. **return** left;
15. } **else** {
16. **return** right;
17. }
18. }
19. }

163. Missing Ranges

Given a sorted integer array **nums**, where the range of elements are in the **inclusive range [*lower*, *upper*]**, return its missing ranges.

**Example:**

**Input:** **nums** = [0, 1, 3, 50, 75], ***lower*** = 0 and ***upper*** = 99,

**Output:** ["2", "4->49", "51->74", "76->99"]

1. **class** Solution {
2. **public** List<String> findMissingRanges(**int**[] nums, **int** lower, **int** upper) {
4. List<String> res = **new** ArrayList<>();
5. **long** low = (**long**) lower;
6. **long** up = (**long**) upper;
7. **for** (**int** num:nums) {
8. **if** (num == low) {
9. low++;
10. } **else** **if** (num > low){
11. **if** (low + 1 == num) {
12. res.add(String.valueOf(low));
13. } **else** {
14. res.add(low + "->" + (num - 1));
15. }
16. low = (**long**) num + 1;
17. }
18. }
19. **if** (low == up) {
20. res.add(String.valueOf(low));
21. } **else** **if** (low < up){
22. res.add(low + "->" + up);
23. }
24. **return** res;
25. }
26. }

157. Read N Characters Given Read4

A stupid question.

1. /\* The read4 API is defined in the parent class Reader4.
2. int read4(char[] buf); \*/
4. **public** **class** Solution **extends** Reader4 {
5. /\*\*
6. \* @param buf Destination buffer
7. \* @param n   Maximum number of characters to read
8. \* @return    The number of characters read
9. \*/
10. **public** **int** read(**char**[] buf, **int** n) {
11. **char**[] temp = **new** **char**[4];
12. **int** index = 0;
13. **while**(**true**) {
14. **int** count = read4(temp);
15. count = Math.min(count, n - index);
16. **for** (**int** i = 0; i < count; i++) {
17. buf[index++] = temp[i];
18. }
19. **if** (index == n || count < 4) **return** index;
20. }
21. }
22. }

165. Compare Version Numbers

Compare two version numbers version1 and version2.  
If version1 > version2 return 1, if version1 < version2 return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character.  
The . character does not represent a decimal point and is used to separate number sequences.  
For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

0.1 < 1.1 < 1.2 < 13.37

1. **class** Solution {
2. **public** **int** compareVersion(String version1, String version2) {
3. String[] v1 = version1.split("\\.");
4. String[] v2 = version2.split("\\.");
5. **for** (**int** i = 0;i < Math.max(v1.length, v2.length);i++) {
6. **int** num1 = i < v1.length ? Integer.parseInt( v1[i] ) : 0;
7. **int** num2 = i < v2.length ? Integer.parseInt( v2[i] ) : 0;
8. **if** (num1 > num2) {
9. **return** 1;
10. } **else** **if** (num2 > num1) {
11. **return** -1;
12. }
13. }
14. **return** 0;
15. }
16. }

166. Fraction to Recurring Decimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

**Example 1:**

**Input:** numerator = 1, denominator = 2

**Output:** "0.5".

**Example 2:**

**Input:** numerator = 2, denominator = 1

**Output:** "2".

**Example 3:**

**Input:** numerator = 2, denominator = 3

**Output:** "0.(6)".

1. **class** Solution {
2. **public** String fractionToDecimal(**int** numerator, **int** denominator) {
3. **if** (numerator == 0) **return** "0";
4. StringBuilder res = **new** StringBuilder();
5. res.append(((numerator > 0) ^ (denominator > 0) ? "-":""));
6. **long** num = Math.abs((**long**) numerator);
7. **long** den = Math.abs((**long**) denominator);
8. res.append(num / den);
9. num %= den;
10. **if** (num == 0) {
11. **return** res.toString();
12. }
13. res.append(".");
14. Map<Long, Integer> map = **new** HashMap<>();
15. map.put(num, res.length());
16. **while**( num != 0) {
17. num \*= 10;
18. res.append(num / den);
19. num %= den;
20. **if** (map.containsKey(num)) {
21. **int** index = map.get(num);
22. res.insert(index,"(");
23. res.append(")");
24. **break**;
25. } **else** {
26. map.put(num,res.length());
27. }
28. }
29. **return** res.toString();
30. }
31. }

170. Two Sum III - Data structure design

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure.  
find - Find if there exists any pair of numbers which sum is equal to the value.

**Example 1:**

add(1); add(-1);

find(0) -> true

**Example 2:**

add(1); add(3); add(5);

find(4) -> true

find(7) -> false

1. **class** TwoSum {
3. /\*\* Initialize your data structure here. \*/
4. Map<Integer, Integer> map;
5. **public** TwoSum() {
6. map = **new** HashMap<>();
7. }
9. /\*\* Add the number to an internal data structure.. \*/
10. **public** **void** add(**int** number) {
11. map.put(number, map.getOrDefault(number,0)+1);
12. }
14. /\*\* Find if there exists any pair of numbers which sum is equal to the value. \*/
15. **public** **boolean** find(**int** value) {
16. **for** (**int** key : map.keySet()) {
17. **int** other = value - key;
18. **if** ((key == other && map.get(key)  > 1) || (key != other && map.containsKey(other))) {
19. **return** **true**;
20. }
21. }
22. **return** **false**;
23. }
24. }

173. Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

**Note:**next() and hasNext() should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

1. **public** **class** BSTIterator {
3. **private** TreeNode curr;
4. Stack<TreeNode> stack;
5. **public** BSTIterator(TreeNode root) {
6. curr = root;
7. stack = **new** Stack<>();
8. }
10. /\*\* @return whether we have a next smallest number \*/
11. **public** **boolean** hasNext() {
12. **if** (!stack.isEmpty() || curr != **null**) {
13. **return** **true**;
14. }
15. **return** **false**;
17. }
19. /\*\* @return the next smallest number \*/
20. **public** **int** next() {
21. **while**(curr != **null**) {
22. stack.push(curr);
23. curr = curr.left;
24. }
25. curr = stack.pop();
26. **int** value = curr.val;
27. curr = curr.right;
28. **return** value;
29. }
30. }
32. /\*\*
33. \* Your BSTIterator will be called like this:
34. \* BSTIterator i = new BSTIterator(root);
35. \* while (i.hasNext()) v[f()] = i.next();
36. \*/

179. Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

**Example 1:**

**Input:** [10,2]

**Output:** "210"

**Example 2:**

**Input:** [3,30,34,5,9]

**Output:** "9534330"

1. **class** Solution {
2. **public** String largestNumber(**int**[] nums) {
3. **if** (nums == **null** || nums.length == 0) **return** "";
4. String[] array = **new** String[nums.length];
5. **for** (**int** i = 0; i < nums.length;i++) {
6. array[i] = String.valueOf(nums[i]);
7. }
8. Arrays.sort(array,**new** Comparator<String>(){
9. @Override
10. **public** **int** compare(String str1, String str2) {
11. String s1 = str1 + str2;
12. String s2 = str2 + str1;
13. **return** s2.compareTo(s1);
14. }
15. });
16. **if** (array[0].charAt(0) == '0') {
17. **return** "0";
18. }
19. StringBuilder res = **new** StringBuilder();
20. **for** (String s:array) {
21. res.append(s);
22. }
23. **return** res.toString();
24. }
25. }

186. Reverse Words in a String II

Given an input string, reverse the string word by word.

**Example:**

**Input:** ["t","h","e"," ","s","k","y"," ","i","s"," ","b","l","u","e"]

**Output:** ["b","l","u","e"," ","i","s"," ","s","k","y"," ","t","h","e"]

**Note:**

* A word is defined as a sequence of non-space characters.
* The input string does not contain leading or trailing spaces.
* The words are always separated by a single space.

**Follow up:**Could you do it *in-place* without allocating extra space?

1. **class** Solution {
2. **public** **void** reverseWords(**char**[] str) {
3. reverse(str, 0, str.length - 1);
4. **int** start = 0;
5. **for** (**int** i = 0; i < str.length; i++) {
6. **if** (str[i] == ' ') {
7. reverse(str, start, i - 1);
8. start = i+1;
9. }
10. }
11. reverse(str, start, str.length - 1);
12. }
14. **private** **void** reverse(**char**[] str, **int** left, **int** right) {
15. **while**(left < right) {
16. **char** temp = str[right];
17. str[right] = str[left];
18. str[left] = temp;
19. left++;
20. right--;
21. }
22. }
23. }

187. Repeated DNA Sequences

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

**Example:**

**Input:** s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

**Output:** ["AAAAACCCCC", "CCCCCAAAAA"]

1. **class** Solution {
2. **public** List<String> findRepeatedDnaSequences(String s) {
3. HashSet<String> set = **new** HashSet<>();
4. HashSet<String> repeated = **new** HashSet<>();
5. **for** (**int** i = 0; i < s.length() - 9;i++) {
6. String temp = s.substring(i, i + 10);
7. **if** (!set.add(temp)) {
8. repeated.add(temp);
9. }
10. }
11. **return** **new** ArrayList(repeated);
12. }
13. }

199. Binary Tree Right Side View

1. **class** Solution {
2. **public** List<Integer> rightSideView(TreeNode root) {
3. List<Integer> res = **new** ArrayList<>();
4. **if** (root == **null**) {
5. **return** res;
6. }
7. Queue<TreeNode> queue = **new** LinkedList<>();
8. queue.offer(root);
9. **while**(!queue.isEmpty()) {
10. **int** size = queue.size();
11. **for** (**int** i = 0; i < size;i++) {
12. TreeNode curr = queue.poll();
13. **if** (curr.left != **null**) {
14. queue.add(curr.left);
15. }
16. **if** (curr.right != **null**) {
17. queue.add(curr.right);
18. }
19. **if** (i == size - 1) {
20. res.add(curr.val);
21. }
22. }
23. }
24. **return** res;
25. }
26. }

201. Bitwise AND of Numbers Range

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

**Example 1:**

**Input:** [5,7]

**Output:** 4

**Example 2:**

**Input:** [0,1]

**Output:** 0

1. **class** Solution {
2. **public** **int** rangeBitwiseAnd(**int** m, **int** n) {
3. **int** offset = 0;
4. **while**( m != n) {
5. m >>= 1;
6. n >>= 1;
7. offset++;
8. }
9. **return** m <<= offset;
10. }
11. }

207. Course Schedule

There are a total of *n* courses you have to take, labeled from 0 to n-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

**Example 1:**

**Input:** 2, [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0. So it is possible.

**Example 2:**

**Input:** 2, [[1,0],[0,1]]

**Output:** false

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0, and to take course 0 you should

  also have finished course 1. So it is impossible.

**Note:**

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).
2. You may assume that there are no duplicate edges in the input prerequisites.

**Hints:**

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. [Topological Sort via DFS](https://class.coursera.org/algo-003/lecture/52) - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via [BFS](http://en.wikipedia.org/wiki/Topological_sorting#Algorithms).
4. **class** Solution {
5. **public** **boolean** canFinish(**int** numCourses, **int**[][] prerequisites) {
6. **int**[] inDegree = **new** **int**[numCourses];
7. **int** res = numCourses;
8. **for** (**int** i = 0; i < prerequisites.length; i++) {
9. inDegree[prerequisites[i][0]]++;
10. }
11. Queue<Integer> queue = **new** LinkedList<>();
12. **for** (**int** i = 0; i < numCourses;i++) {
13. **if** (inDegree[i] == 0) {
14. queue.offer(i);
15. }
16. }
17. **while**(!queue.isEmpty()) {
18. **int** course = queue.poll();
19. res--;
20. **for** (**int**[] pair: prerequisites) {
21. **if** (pair[1] == course) {
22. inDegree[pair[0]]--;
23. **if** (inDegree[pair[0]] == 0) {
24. queue.offer(pair[0]);
25. }
26. }
27. }
28. }
29. **return** res == 0;
30. }
31. }

210. Course Schedule II

1. **class** Solution {
2. **public** **int**[] findOrder(**int** numCourses, **int**[][] prerequisites) {
3. **int**[] inDegree = **new** **int**[numCourses];
4. **int**[] res = **new** **int**[numCourses];
5. **for** (**int** i = 0; i < prerequisites.length; i++) {
6. inDegree[prerequisites[i][0]]++;
7. }
8. Queue<Integer> queue = **new** LinkedList<>();
9. **for** (**int** i = 0; i < inDegree.length; i++) {
10. **if** (inDegree[i] == 0) {
11. queue.offer(i);
12. }
13. }
14. **int** j = 0;
15. **while**(!queue.isEmpty()) {
16. **int** course = queue.poll();
17. res[j++] = course;
18. **for** (**int**[] pair:prerequisites) {
19. **if** (pair[1] == course) {
20. inDegree[pair[0]]--;
21. **if** (inDegree[pair[0]] == 0) {
22. queue.offer(pair[0]);
23. }
24. }
25. }
26. }
27. **return** j == numCourses ? res: **new** **int**[0];
28. }
29. }

208. Implement Trie (Prefix Tree)

1. **class** Trie {
3. **private** TrieNode root;
5. /\*\* Initialize your data structure here. \*/
6. **public** Trie() {
7. root = **new** TrieNode();
8. }
10. /\*\* Inserts a word into the trie. \*/
11. **public** **void** insert(String word) {
12. TrieNode node = root;
13. **for** (**int** i = 0;i < word.length(); i++) {
14. **int** j = word.charAt(i) - 'a';
15. **if** (node.children[j] == **null**) {
16. node.children[j] = **new** TrieNode();
17. }
18. node = node.children[j];
19. }
20. node.isWord = **true**;
21. }
23. /\*\* Returns if the word is in the trie. \*/
24. **public** **boolean** search(String word) {
25. TrieNode node = root;
26. **for** (**int** i = 0; i < word.length();i++) {
27. **int** j = word.charAt(i) - 'a';
28. **if** (node.children[j] == **null**) {
29. **return** **false**;
30. }
31. node = node.children[j];
32. }
33. **return** node.isWord;
34. }
36. /\*\* Returns if there is any word in the trie that starts with the given prefix. \*/
37. **public** **boolean** startsWith(String prefix) {
38. TrieNode node = root;
39. **for** (**int** i = 0; i < prefix.length(); i++) {
40. **int** j = prefix.charAt(i) - 'a';
41. **if** (node.children[j] == **null**) {
42. **return** **false**;
43. }
44. node = node.children[j];
45. }
46. **return** **true**;
47. }
48. }
50. **class** TrieNode {
51. TrieNode[] children;
52. **boolean** isWord;
54. **public** TrieNode() {
55. children = **new** TrieNode[26];
56. isWord = **false**;
57. }
58. }

211. Add and Search Word - Data structure design

Design a data structure that supports the following two operations:

void addWord(word)

bool search(word)

search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

**Example:**

addWord("bad")

addWord("dad")

addWord("mad")

search("pad") -> false

search("bad") -> true

search(".ad") -> true

search("b..") -> true

1. **class** WordDictionary {
3. **private** TrieNode root;
5. /\*\* Initialize your data structure here. \*/
6. **public** WordDictionary() {
7. root = **new** TrieNode();
8. }
10. /\*\* Adds a word into the data structure. \*/
11. **public** **void** addWord(String word) {
12. TrieNode node = root;
13. **for** (**int** i = 0; i < word.length();i++) {
14. **int** j = word.charAt(i) - 'a';
15. **if** (node.children[j] == **null**) {
16. node.children[j] = **new** TrieNode();
17. }
18. node = node.children[j];
19. }
20. node.isWord = **true**;
21. }
23. /\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/
24. **public** **boolean** search(String word) {
25. **return** find(word, root, 0);
26. }
28. **public** **boolean** find(String word, TrieNode node, **int** index) {
29. **if** (index == word.length()) {
30. **return** node.isWord;
31. }
32. **if** (word.charAt(index) == '.') {
33. **for** (TrieNode temp : node.children) {
34. **if** (temp != **null** && find(word, temp, index + 1)) **return** **true**;
35. }
36. **return** **false**;
37. } **else** {
38. **int** j = word.charAt(index) - 'a';
39. TrieNode temp = node.children[j];
40. **return** temp != **null** && find(word, temp, index + 1);
41. }
42. }
43. }
45. **class** TrieNode{
46. TrieNode[] children;
47. **boolean** isWord;
49. **public** TrieNode() {
50. children = **new** TrieNode[26];
51. isWord = **false**;
53. }
54. }
56. /\*\*
57. \* Your WordDictionary object will be instantiated and called as such:
58. \* WordDictionary obj = new WordDictionary();
59. \* obj.addWord(word);
60. \* boolean param\_2 = obj.search(word);
61. \*/

213. House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

**Example 1:**

**Input:** [2,3,2]

**Output:** 3

**Explanation:** You cannot rob house 1 (money = 2) and then rob house 3 (money = 2),

  because they are adjacent houses.

**Example 2:**

**Input:** [1,2,3,1]

**Output:** 4

**Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3).

  Total amount you can rob = 1 + 3 = 4.

1. **class** Solution {
2. **public** **int** rob(**int**[] nums, **int** low, **int** high) {
3. **int** preIn = 0;
4. **int** preOut = 0;
5. **for** (**int** i = low; i <= high; i++) {
6. **int** temp = preIn;
7. preIn = nums[i] + preOut;
8. preOut = Math.max(preOut, temp);
9. }
10. **return** Math.max(preIn, preOut);
11. }
13. **public** **int** rob(**int**[] nums) {
14. **if** (nums.length == 1) {
15. **return** nums[0];
16. }
17. **return** Math.max(rob(nums, 0, nums.length - 2), rob(nums, 1, nums.length - 1));
18. }
19. }

216. Combination Sum III

Find all possible combinations of ***k*** numbers that add up to a number ***n***, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

**Note:**

* All numbers will be positive integers.
* The solution set must not contain duplicate combinations.

**Example 1:**

**Input:** ***k*** = 3, ***n*** = 7

**Output:** [[1,2,4]]

**Example 2:**

**Input:** ***k*** = 3, ***n*** = 9

**Output:** [[1,2,6], [1,3,5], [2,3,4]]

1. **class** Solution {
2. **public** List<List<Integer>> combinationSum3(**int** k, **int** n) {
3. List<List<Integer>> res = **new** ArrayList<>();
4. helper(res, **new** ArrayList(), k, n, 1);
5. **return** res;
6. }
8. **private** **void** helper(List<List<Integer>> res, List<Integer> subList, **int** k, **int** n, **int** start) {
9. **if** (k == 0 && n == 0) {
10. res.add(**new** ArrayList(subList));
11. **return**;
12. }
13. **for** (**int** i = start; i <= 9; i++) {
14. subList.add(i);
15. helper(res, subList, k - 1, n - i, i + 1);
16. subList.remove(subList.size() - 1);
18. }
19. }
20. }

230. Kth Smallest Element in a BST

Given a binary search tree, write a function kthSmallest to find the **k**th smallest element in it.

**Note:**  
You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

**Follow up:**  
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Why static variable count doesn’t work?

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** **int** count = 0;
12. **public** **int** res = 0;
13. **public** **int** kthSmallest(TreeNode root, **int** k) {
14. inorder(root,k);
15. **return** res;
16. }
17. **public** **void** inorder(TreeNode root, **int** k) {
18. **if** (root == **null**) **return**;
19. inorder(root.left,k);
20. count++;
21. System.out.println(count);
22. **if** (count == k) {
23. res = root.val;
24. System.out.println(res);
25. **return**;
26. }
27. inorder(root.right,k);
28. }
29. }

299. Bulls and Cows

You are playing the following [Bulls and Cows](https://en.wikipedia.org/wiki/Bulls_and_Cows) game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

For example:

Secret number: "1807"

Friend's guess: "7810"

Hint: 1 bull and 3 cows. (The bull is 8, the cows are 0, 1 and 7.)

Write a function to return a hint according to the secret number and friend's guess, use A to indicate the bulls and B to indicate the cows. In the above example, your function should return "1A3B".

Please note that both secret number and friend's guess may contain duplicate digits, for example:

Secret number: "1123"

Friend's guess: "0111"

In this case, the 1st 1 in friend's guess is a bull, the 2nd or 3rd 1 is a cow, and your function should return "1A1B".

You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

The idea is to iterate over the numbers in secret and in guess and count all bulls right away. For cows maintain an array that stores count of the number appearances in secret and in guess. Increment cows when either number from secret was already seen in guest or vice versa.

1. **class** Solution {
2. **public** String getHint(String secret, String guess) {
3. **int** bull=0,cow = 0;
4. **char**[] sec = secret.toCharArray();
5. **char**[] gue = guess.toCharArray();
6. **int**[] table = **new** **int**[10];
8. **for** (**int** i=0;i<sec.length;i++) {
9. **if** (sec[i] == gue[i]) {
10. bull++;
11. } **else** {
12. **if** (table[sec[i]-'0']++<0) {
13. cow++;
14. }
15. **if** (table[gue[i]-'0']-->0) {
16. cow++;
17. }
18. }
19. }
20. **return** bull+"A"+cow+"B";
21. }
22. }

300. Longest Increasing Subsequence

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example,  
Given [10, 9, 2, 5, 3, 7, 101, 18],  
The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4. Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(*n2*) complexity.

**Follow up:** Could you improve it to O(*n* log *n*) time complexity?

1. **class** Solution {
2. **public** **int** lengthOfLIS(**int**[] nums) {
3. **if** (nums == **null** || nums.length == 0) **return** 0;
4. **int** n = nums.length;
5. **int**[] res = **new** **int**[nums.length];
6. **int** max = 1;
7. **for** (**int** i = 0; i < n; i++) {
8. res[i] = 1;
9. }
10. **for** (**int** i = 1; i < n; i++) {
11. **for** (**int** j = 0; j < i; j++) {
12. **if** (nums[i] > nums[j]) {
13. res[i] = Math.max(res[i],res[j] + 1);
14. max = Math.max(max,res[i]);
15. }
16. }
17. }
18. **return** max;
19. }
20. }

322. Coin Change(DP)

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

**Example 1:**  
coins = [1, 2, 5], amount = 11  
return 3 (11 = 5 + 5 + 1)

**Example 2:**  
coins = [2], amount = 3  
return -1.

1. **class** Solution {
2. **public** **int** coinChange(**int**[] coins, **int** amount) {
3. **if** (amount == 0) **return** 0;
4. **int**[] res = **new** **int**[amount+1];
5. **for** (**int** i = 1; i<= amount; i++) {
6. res[i] = Integer.MAX\_VALUE;
7. **for** (**int** j = 0; j < coins.length;j++) {
8. **if** (coins[j] <= i && res[i-coins[j]] != Integer.MAX\_VALUE) {
9. res[i] = Math.min(res[i],res[i-coins[j]]+1);
10. }
11. }
12. }
13. **return** res[amount] == Integer.MAX\_VALUE?-1:res[amount];
14. }
15. }

450. Delete Node in a BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

**Note:** Time complexity should be O(height of tree).

**Example:**

root = [5,3,6,2,4,null,7]

key = 3

5

/ \

3 6

/ \ \

2 4 7

Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the following BST.

5

/ \

4 6

/ \

2 7

Another valid answer is [5,2,6,null,4,null,7].

5

/ \

2 6

\ \

4 7

Three cases in deleting the nodes: no children, one single child, two children

1. /\*\*
2. \* Definition for a binary tree node.
3. \* public class TreeNode {
4. \*     int val;
5. \*     TreeNode left;
6. \*     TreeNode right;
7. \*     TreeNode(int x) { val = x; }
8. \* }
9. \*/
10. **class** Solution {
11. **public** TreeNode deleteNode(TreeNode root, **int** key) {
12. **if** (root == **null**) **return** root;
13. **if** (root.val > key) {
14. root.left = deleteNode(root.left, key);
15. } **else** **if** (root.val < key) {
16. root.right =  deleteNode(root.right, key);
17. } **else** {
18. **if** (root.left == **null** && root.right == **null**) {
19. root = **null**;
20. } **else** **if** (root.left == **null**) {
21. **return** root.right;
22. } **else** **if** (root.right == **null**) {
23. **return**  root.left;
24. } **else** {
25. root.val = successorValue(root.right);
26. root.right = deleteNode (root.right, root.val);
27. }
28. }
29. **return** root;
31. }
32. **private** **int** successorValue(TreeNode root) {
33. **int** min = root.val;
34. **while** (root.left != **null**) {
35. min = root.left.val;
36. root = root.left;
37. }
38. **return** min;
39. }
40. }

777. Swap Adjacent in LR String

In a string composed of 'L', 'R', and 'X' characters, like "RXXLRXRXL", a move consists of either replacing one occurrence of "XL" with "LX", or replacing one occurrence of "RX" with "XR". Given the starting string start and the ending string end, return True if and only if there exists a sequence of moves to transform one string to the other.

**Example:**

**Input:** start = "RXXLRXRXL", end = "XRLXXRRLX"

**Output:** True

**Explanation:**

We can transform start to end following these steps:

RXXLRXRXL ->

XRXLRXRXL ->

XRLXRXRXL ->

XRLXXRRXL ->

XRLXXRRLX

1. **class** Solution {
2. **public** **boolean** canTransform(String start, String end) {
3. **int** s = 0, e = 0;
4. **int** len = start.length();
5. **while**(s < len && e < len) {
6. **while**( s < len && start.charAt(s) == 'X') {
7. s++;
8. }
9. **while**(e< len && end.charAt(e) == 'X') {
10. e++;
11. }
12. **if** (s==len && e==len) {
13. **return** **true**;
14. }
16. **if** (s==len || e==len) {
17. **return** **false**;
18. }
20. **if** (start.charAt(s) != end.charAt(e)) {
21. **return** **false**;
22. }
23. **if** (start.charAt(s) == 'R' && s > e) {
24. **return** **false**;
25. }
26. **if** (start.charAt(s) == 'L' && s < e) {
27. **return** **false**;
28. }
29. s++;
30. e++;
31. }
32. **return** **true**;
33. }
34. }

Hard

4. Median of Two Sorted Arrays

There are two sorted arrays **nums1** and **nums2** of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

**Example 1:**

nums1 = [1, 3]

nums2 = [2]

The median is 2.0

**Example 2:**

nums1 = [1, 2]

nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5

Time complexity O(log(min(m,n)))

1. **class** Solution {
2. **public** **double** findMedianSortedArrays(**int**[] nums1, **int**[] nums2) {
3. **if** (nums1.length > nums2.length) {
4. **return** findMedianSortedArrays(nums2,nums1); // always look at the shorter array
5. }
6. **int** len = nums1.length + nums2.length;
7. **int** cut1 = 0; // left elements in nums1
8. **int** cut2 = 0; // left elements in nums2
9. **int** cutL = 0; // the start in nums1
10. **int** cutR = nums1.length; // the end in nums1
12. **while**(cut1 <= nums1.length) {
14. cut1 = cutL + (cutR - cutL)/2;
15. cut2 = len/2 - cut1;
17. **double** L1 = (cut1 == 0)? Integer.MIN\_VALUE:nums1[cut1-1];
18. **double** R1 = (cut1 == nums1.length)? Integer.MAX\_VALUE:nums1[cut1];
19. **double** L2 = (cut2 == 0)? Integer.MIN\_VALUE:nums2[cut2-1];
20. **double** R2 = (cut2 == nums2.length)? Integer.MAX\_VALUE:nums2[cut2];
22. **if** (L1 > R2) {
23. cutR = cut1 - 1;
24. } **else** **if** (L2 > R1) {
25. cutL = cut1 + 1;
26. } **else** {
27. **if** (len % 2 == 0) {
28. L1 = L1 > L2 ? L1:L2;
29. R1 = R1 > R2 ? R2:R1;
30. **return** (L1+R1)/2;
32. } **else** {
33. R1 = R1 > R2 ? R2:R1;
34. **return** R1;
35. }
36. }
38. }
39. **return** -1;
41. }
43. }

45. Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:  
Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

**Note:**  
You can assume that you can always reach the last index.

Contrast with 55

1. **class** Solution {
2. **public** **int** jump(**int**[] nums) {
3. **if** (nums == **null** || nums.length <= 1) **return** 0;
4. **int** start = 0;
5. **int** currMax = 0;
6. **int** nextMax = 0;
7. **int** ans = 0;
8. **while** (start <= currMax) {
9. **while** (start <= currMax) {
10. nextMax = Math.max(nextMax,start+nums[start]);
11. start++;
12. }
13. currMax = nextMax;
14. ans++;
15. **if** (currMax >= nums.length - 1) **return** ans;
17. }
18. **return** -1;
19. }
20. }

282. Expression Add Operators(backtrack)

Given a string that contains only digits 0-9 and a target value, return all possibilities to add **binary** operators (not unary) +, -, or \*between the digits so they evaluate to the target value.

Examples:

"123", 6 -> ["1+2+3", "1\*2\*3"]

"232", 8 -> ["2\*3+2", "2+3\*2"]

"105", 5 -> ["1\*0+5","10-5"]

"00", 0 -> ["0+0", "0-0", "0\*0"]

"3456237490", 9191 -> []

1. **class** Solution {
2. **public** List<String> addOperators(String num, **int** target) {
3. List<String> res = **new** ArrayList<>();
4. **if** (num == null || num.length() == 0) **return** res;
5. addOperatorsHelper(res, num, "",target,0,0,0);
6. **return** res;
7. }
8. **public** **void** addOperatorsHelper(List<String> res,String num, String path,**int** target, **int** index,**long** tempValue,**long** prev) {
9. **if** (index == num.length() && tempValue == target) {
10. res.add(path);
11. **return**;
12. }
13. **for** (**int** i = index;i < num.length();i++) {
14. **if** (i != index && num.charAt(index) == '0') **break**;
15. **long** curr = Long.parseLong(num.substring(index,i+1));
16. **if** (index == 0) {
17. addOperatorsHelper(res, num, path + curr, target, i + 1, curr, curr);
18. } **else** {
19. addOperatorsHelper(res, num, path + "+" + curr, target, i + 1, tempValue + curr, curr);
20. addOperatorsHelper(res, num, path + "-" + curr, target, i + 1, tempValue - curr, -curr);
21. addOperatorsHelper(res, num, path + "\*" + curr, target, i + 1, tempValue - prev + prev \* curr, prev \* curr);
22. }
23. }
24. }
25. }

312. Burst Balloons

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] \* nums[i] \* nums[right] coins. Here left and rightare adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

**Note:**   
(1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.  
(2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

**Example:**

Given [3, 1, 5, 8]

Return 167

nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167

DP problem, 关键是找到状态转移方程，dp[i][j] = dp[i][k-1]+nums[i-1]\*nums[k]\*nums[j+1]+dp[k+1][j], 这里k是i,j之间最后扎破的气球

1. **class** Solution {
2. **public** **int** maxCoins(**int**[] nums) {
3. **int** n = nums.length;
4. **int**[] fullNums = **new** **int**[n+2];
5. fullNums[0] = 1;
6. fullNums[n+1] = 1;
7. **for** (**int** i=0;i<nums.length;i++) {
8. fullNums[i+1] = nums[i];
9. }
11. **int**[][] coins = **new** **int**[n+2][n+2];
13. **for** (**int** l=1;l<=n;l++) {
14. **for** (**int** i=1;i<= n-l+1;i++) {
15. **int** j=i+l-1;
16. **for** (**int** k=i;k<=j;k++) {
17. coins[i][j] = Math.max(coins[i][j], coins[i][k-1]+fullNums[i-1]\*fullNums[k]\*fullNums[j+1]+coins[k+1][j]);
18. }
20. }
21. }
22. **return** coins[1][n];
23. }
24. }