**排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**快排**

**public int**[] quickSort(**int**[] nums){  
 quickSort(nums, 0, nums.**length** - 1);  
 **return** nums;  
}

**static void** quickSort(**int** arr[], **int** left, **int** right) {  
 **int** index = *partition*(arr, left, right);  
 **if** (left < index-1) *quickSort*(arr, left, index-1); *//****TODO 注意边界* if** (index < right) *quickSort*(arr, index, right);  
}  
**static int** partition(**int** arr[], **int** left, **int** right) {  
 **int** l = left, r = right, pivot = arr[(left+right)/2];  
 **while** (l<=r) {  
 **while** (arr[l]<pivot) l++;  
 **while** (arr[r]>pivot) r--;  
 **if** (l<=r) {  
 *swap*(arr, l, r);  
 l++; r--;  
 }  
 };*// 最后退出的情况应该是右指针在左指针左边一格* **return** l;  
}

**快排（另一种写法）**

**public int**[] quickSort(**int**[] nums){  
 quickSort(nums, 0, nums.**length** - 1);  
 **return** nums;  
}  
**private void** quickSort(**int**[] nums, **int** left, **int** right){  
 **if** (left < right) {  
 **int** index = partition(nums, left, right);  
 quickSort(nums, left, index - 1);  
 quickSort(nums, index + 1, right);  
 }  
}  
**private int** partition(**int** nums[], **int** l, **int** r){  
 **int** pivot = nums[l]; *//枢轴记录* **while** (l < r){  
 **while** (l<r && nums[r]>=pivot) r--;  
 nums[l] = nums[r]; *//交换比枢轴小的记录到左端* **while** (l<r && nums[l]<=pivot) l++;  
 nums[r] = nums[l]; *//交换比枢轴小的记录到右端* }  
 nums[l] = pivot; *//扫描完成，枢轴到位* **return** l; *//返回枢轴的位置*}

**快选（第K大元素）**

**public int** findKthLargest(**int**[] nums, **int** k) {  
 **return** quickSelect(nums, k-1, 0, nums.**length** - 1);  
}  
**private int** quickSelect(**int**[] arr, **int** k, **int** left, **int** right){ *//返回第k大的索引* **int** pivot = arr[left+(right-left)/2], l = left, r = right;  
 **while**(l<=r){  
 **while**(arr[l]>pivot) l++; *// 从左向右找到第一个小于枢纽值的数* **while**(arr[r]<pivot) r--; *// 从右向左找到第一个大于枢纽值的数* **if**(l<=r){ *// 将两个数互换* swap(arr, l, r);  
 l++;  
 r--;  
 }  
 }*// 最后退出的情况应该是右指针在左指针左边一格* **if**(left<r && k<=r) **return** quickSelect(arr, k, left, r);*// 这时如果右指针还大于等于k，说明kth在左半边* **if**(l<right && k>=l) **return** quickSelect(arr, k, l, right);*// 这时如果左指针还小于等于k，说明kth在右半边* **return** arr[k];  
}

**归并排序**

**public int**[] mergeSort(**int**[] nums){  
 **int**[] tempSorted = **new int**[nums.**length**];  
 mergeSort(nums, 0, nums.**length** - 1, tempSorted);  
 **return** nums; *//或return tempSorted; 最终两个数组都是排好序的*}  
**private void** mergeSort(**int**[] unsorted, **int** left, **int** right, **int**[] tempSorted){  
 **if** (left < right){  
 **int** mid = (left + right) / 2;  
 mergeSort(unsorted, left, mid, tempSorted); *//左边有序* mergeSort(unsorted, mid + 1, right, tempSorted); *//右边有序* merge(unsorted, left, mid, right, tempSorted); *//再将二个有序数列合并* }  
}  
**private void** merge(**int**[] origin, **int** left, **int** mid, **int** right, **int**[] tempSorted){  
 **int** i = left, j = mid + 1, k = 0;  
 **while** (i <= mid && j <= right){ *//真正排序的地方* **if** (origin[i] < origin[j]) tempSorted[k++] = origin[i++];  
 **else** tempSorted[k++] = origin[j++];  
 }  
 **while** (i <= mid) tempSorted[k++] = origin[i++];  
 **while** (j <= right) tempSorted[k++] = origin[j++];  
 *//更新origin数组* **for** (**int** x = 0; x < k; x++) origin[left + x] = tempSorted[x];  
}

SortList 归并排序 快慢指针

**public class** SortList { *\* O(nlogn)可以采用快速排序/归并排序/堆排序,对于链表用归并排序方便(配合合并两个有序链表).* **public** ListNode sortList(ListNode head) {  
 **return** mergeSort(head);  
 }  
 **private** ListNode mergeSort(ListNode head){  
 **if** (head==**null** || head.**next**==**null**) **return** head;  
 ListNode slow = head, fast = head;  
 **while** (fast.**next** != **null** && fast.**next**.**next** != **null**){  
 slow = slow.**next**;  
 fast = fast.**next**.**next**;  
 }  
 ListNode head2 = slow.**next**;  
 slow.**next** = **null**;  
 ListNode l1 = mergeSort(head);  
 ListNode l2 = mergeSort(head2);  
 **return** mergeTwoSortedLinkedList(l1, l2);  
 }  
}

插入排序

*\* 插入排序就是每一步都将一个待排数据按其大小插入到已经排序的数据中的适当位置，直到全部插入完毕。***public int**[] insertionSort(**int**[] nums) {  
 **for** (**int** i=1; i<nums.**length**; i++)  
 **for** (**int** j=i; j>0 && nums[j-1]>nums[j]; j--) *swap*(nums, j-1, j);  
 **return** nums;  
}

**public int**[] insertionSort(**int**[] nums){  
 **for** (**int** i=1; i<nums.**length**; i++){  
 **if** (nums[i-1]>nums[i]){  
 **int** temp = nums[i];j = i;  
 **while** (j>0 && nums[j-1]>temp) {  
 nums[j] = nums[j-1]; j--;  
 }  
 nums[j] = temp;  
 }  
 }  
 **return** nums;  
}

InsertionSortList

**public class** InsertionSortList {  
 *//很容易出错* **public** ListNode insertionSortList0(ListNode head) {  
 ListNode dummy = **new** ListNode(0);  
 ListNode pre = dummy;  
 ListNode cur = head;  
 **while**(cur!=**null**) {  
 pre = dummy;  
 **while**(pre.**next**!=**null** && pre.**next**.**val**<cur.**val**) {  
 pre = pre.**next**;  
 }  
 ListNode next = cur.**next**;  
 cur.**next** = pre.**next**;  
 pre.**next** = cur;  
 cur = next;  
 }  
 **return** dummy.**next**;  
 }

}

**二叉树先中后序迭代遍历范式\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**public** List<Integer> preorderTraversal(TreeNode root) {  
 List<Integer> result = **new** ArrayList<>();  
 Deque<TreeNode> stack = **new** ArrayDeque<>();  
 TreeNode p = root;  
 **while**(!stack.isEmpty() || p != **null**) {  
 **if**(p != **null**) {  
 stack.push(p);  
 result.add(p.**val**); *// Add before going to children* p = p.**left**;  
 } **else** {  
 TreeNode node = stack.pop();  
 p = node.**right**;  
 }  
 }  
 **return** result;  
}  
**public** List<Integer> inorderTraversal(TreeNode root) {  
 List<Integer> result = **new** ArrayList<>();  
 Deque<TreeNode> stack = **new** ArrayDeque<>();  
 TreeNode p = root;  
 **while**(!stack.isEmpty() || p != **null**) {  
 **if**(p != **null**) {  
 stack.push(p);  
 p = p.**left**;  
 } **else** {  
 TreeNode node = stack.pop();  
 result.add(node.**val**); *// Add after all left children* p = node.**right**;  
 }  
 }  
 **return** result;  
}  
**public** List<Integer> postorderTraversal(TreeNode root) {  
 LinkedList<Integer> result = **new** LinkedList<>();//注意定义是LinkedList  
 Deque<TreeNode> stack = **new** ArrayDeque<>();  
 TreeNode p = root;  
 **while**(!stack.isEmpty() || p != **null**) { //TODO 注意是||  
 **if**(p != **null**) {  
 stack.push(p);  
 result.addFirst(p.**val**); *// Reverse the process of preorder* p = p.**right**; *// Reverse the process of preorder* } **else** {  
 TreeNode node = stack.pop();  
 p = node.**left**; *// Reverse the process of preorder* }  
 }  
 **return** result;  
}

**二分搜索类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

FirstBadVersion 找第一个满足某性质的二分搜索标准范式

**public class** FirstBadVersion {  
 */\* The isBadVersion API is defined in the parent class VersionControl.  
 boolean isBadVersion(int version); \*/* **private class** VersionControl {  
 **boolean** isBadVersion(**int** version){  
 **return false**;  
 }  
 }**public class** Solution **extends** VersionControl {  
 *//简洁* **public int** firstBadVersion(**int** n) {  
 **int** l = 1, r = n;  
 **while** (l<=r){  
 int m = l+(r-l)/2;  
 **if** (isBadVersion(m)) r = m-1;  
 **else** l = m+1;  
 }  
 **return** l;  
 }

}

*/\*\*Given a sorted array of integers, 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].*

SearchForARange找第一个满足某性质的二分搜索标准范式

SearchInseartPosition

**public class** SearchForARange { *//简洁, 巧妙* **public int**[] searchRange(**int**[] nums, **int** target) {  
 **int** start = *firstGeq*(nums, target);  
 **if** (start==nums.**length** || nums[start] != target)  
 **return new int**[]{-1, -1};  
 **return new int**[]{start, *firstGeq*(nums, target+1)-1};  
 }  
 **private static int** firstGeq(**int**[] nums, **int** target) {  
 **int** l = 0, r = nums.**length**-1;  
 **while** (l<=r) {  
 **int** m = l+((r-l)>>1);  
 **if** (nums[m]>=target) r = m-1;  
 **else** l = m+1;  
 }  
 **return** l;  
 }

}

**LIS 含标准DP解法**

**public class** LongestIncreasingSubsequence {  
 *//****TODO 较难理解,注意Arrays.binarySearch的用法****  
 \* O(nlogn) DP与二分搜索,tricky* **public int** lengthOfLIS(**int**[] nums) {  
 **int**[] ascendings = **new int**[nums.**length**];  
 **int** len = 0;  
 **for**(**int** x: nums) {  
 **int** i = Arrays.*binarySearch*(ascendings, 0, len, x);  
 **if**(i<0) i = -(i+1); *//****TODO -(i+1)表示要维持原序列顺序,待插入的位置*** ascendings[i] = x; *//更新/增加最后一个值* **if**(i==len) len++;  
 }  
 **return** len;  
 } *\* O(n^2) brute force(DP): dp[i]为以nums[i]结尾的LIS  
 dp[j]=max(dp[j], dp[i]+1), i<j and nums[i]<nums[j]. 再找出dp中的最大值.* **public int** lengthOfLIS1(**int**[] nums) {  
 **if** (nums==**null** || nums.**length**==0) **return** 0;  
 **int** len = nums.**length**;  
 **int**[] dp = **new int**[len];  
 Arrays.*fill*(dp, 1);  
 **for** (**int** j=0; j<len; j++){  
 **for** (**int** i=0; i<j; i++){  
 **if** (nums[i]<nums[j])  
 dp[j] = Math.*max*(dp[j], dp[i]+1);  
 }  
 }  
 **int** max = 0;  
 **for** (**int** x: dp) max = Math.*max*(max, x);  
 **return** max;  
 }  
}

*/\*\*Follow up for H-Index: What if the citations array is sorted in ascending order? Could you optimize your algorithm?  
 Hint: Expected runtime complexity is in O(log n) and the input is sorted.  
 \* Created by eugene on 16/5/18.*

HIndexII

**public class** HIndexII {  
 */\*\*  
 \* https://segmentfault.com/a/1190000003794831  
 \* 二分法, O(log n)* **public int** hIndex(**int**[] citations) {  
 **int** N = citations.**length**;  
 **if**(N==0) **return** 0;  
 **int** l = 0, r = citations.**length**-1;  
 **while**(l <= r){  
 **int** m = (l+r)/2;  
 **if**(citations[m] < N-m){ *// 如果该点是有效的H指数，则最大H指数一定在右边* l = m+1;  
 } **else if** (citations[m] == N-m)  
 **return** N-m;  
 **else** { *// 否则最大H指数在左边* r = m-1;  
 }  
 }  
 **return** N-l; *//* ***TODO 注意返回值是N-l, N-l是l点的H指数*** }

}

*/\*\*Suppose a sorted array 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.  
 You may assume no duplicate exists in the array.  
 \* Created by eugene on 16/3/12.*

FindMinimumInRotatedSortedArray 另一种二分形式

**public class** FindMinimumInRotatedSortedArray {  
 *//简洁,但是非标准二分形式* **public int** findMin(**int**[] nums) {  
 **int** l = 0, r = nums.**length**-1;  
 **while**(l<r){ *//****TODO 此处不是l<=r***  
 **int** m = l+(r-l)/2;  
 **if**(nums[m]<nums[r]) r = m; *//****TODO 此处不是r = m+1***  
 **else** l = m+1; }  
 **return** nums[l];  
 }

*}*

*/\*\*Follow up for "Find Minimum in Rotated Sorted Array":  
 What if duplicates are allowed?  
 Would this affect the run-time complexity? How and why?  
 \* Created by eugene on 16/3/12.*

FindMinimumInRotatedSortedArrayII HARD

**public class** FindMinimumInRotatedSortedArrayII {  
 *//当nums[m] = nums[r]时，无法排除一半的序列，而只能排除掉nums[r],此时只能搜寻nums[l:r-1]* **public int** findMin(**int**[] nums) {  
 **int** l = 0, r = nums.**length**-1;  
 **while**(l<r){ *//****TODO 此处不是l<=r***  
 **int** m = l+(r-l)/2;  
 **if**(nums[m]<nums[r]) r = m; *//****TODO 此处不是r = m+1***  
 **else if** (nums[m]<nums[r]) l = m+1; **else** r--;  
 }  
 **return** nums[l];  
 }

*}*

*/\*\*  
 \* Suppose a sorted array 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.  
 \* Created by Eugene on 11/16/2015.*

SearchInRotatedSortedArray HARD

**public class** SearchInRotatedSortedArray {

*\* 假设数组是A，每次左边缘为l，右边缘为r，还有中间位置是m。在每次迭代中，分三种情况：  
 （1）如果target==A[m]，那么m就是我们要的结果，直接返回；  
 （2）如果A[m]<A[r]，那么说明从m到r一定是有序的（没有受到rotate的影响），  
 那么我们只需要判断target是不是在m到r之间，如果是则把左边缘移到m+1，  
 否则就target在另一半，即把右边缘移到m-1。  
 （3）如果A[m]>=A[r]，那么说明从l到m一定是有序的，同样只需要判断target是否在这个范围内，  
 相应的移动边缘即可。  
 \* 根据以上方法，每次我们都可以切掉一半的数据，所以算法的时间复杂度是O(logn)，空间复杂度是O(1)。\* 4 5 6 7 0 1 2* **public int** search(**int**[] nums, **int** target) {  
 **if** (nums==**null** || nums.**length**==0) **return** 0;  
 **int** l = 0, r = nums.**length**-1;  
 **while** (l<=r){  
 int m = l+(r-l)/2;  
 **if** (target==nums[m]) **return** m;  
 **if** (nums[m]<nums[r]) {  
 **if** (nums[m]<target && target<=nums[r]) l = m+1;  
 **else** r = m-1;  
 } **else** {  
 **if** (nums[l]<=target && target<nums[m]) r = m-1;  
 **else** l = m+1;  
 }  
 }  
 **return** -1;  
 }  
}

*/\*\*  
 \* Follow up for "Search in Rotated Sorted Array":  
 What if duplicates are allowed?  
 Would this affect the run-time complexity? How and why?  
 Write a function to determine if a given target is in the array.  
 \* Created by DCLab on 1/2/2016.*

SearchInRotatedSortedArrayII HARD

**public class** SearchInRotatedSortedArrayII {  
 */\*\*  
 \* 当有重复数字，会存在A[m] = A[r]的情况。此时右半序列可能是sorted，也可能并没有sorted，如下例子。  
 3 1 2 3 3 3 3  
 3 3 3 3 1 2 3  
 所以当A[m] = A[r] != target时，无法排除一半的序列，而只能排除掉A[r]，此时只能搜寻A[l : r-1] 正因为这个变化，在最坏情况下，算法的复杂度从O(logn)退化成了O(n)：例如序列 2 2 2 2 2 2 2 中寻找target = 1。* **public boolean** search(**int**[] nums, **int** target) {  
 **if** (nums == **null** || nums.**length** == 0) **return false**;  
 **int** l = 0, r = nums.**length**-1;  
 **int** m = 0;  
 **while** (l<=r){  
 m = l+(r-l)/2;  
 **if** (target==nums[m]) **return true**;  
 **if** (nums[m]<nums[r]) { *//m~r有序* **if** (nums[m]<target && target<=nums[r]) l = m+1;  
 **else** r = m-1;  
 } **else if** (nums[m] > nums[r]){ *//l~m有序* **if** (nums[l]<=target && target<nums[m]) r = m-1;  
 **else** l = m+1;  
 } **else** { *//nums[m] == nums[r]时* r--;  
 }  
 }  
  
 **return false**;  
 }  
}

Search2DMatrix Search2DMatrixII

*/\*\*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 in ascending from left to right.  
 Integers in each column are sorted in ascending from top to bottom.  
 For example, Consider the following matrix:  
 [ [1, 4, 7, 11, 15],  
 [2, 5, 8, 12, 19],  
 [3, 6, 9, 16, 22],  
 [10, 13, 14, 17, 24],  
 [18, 21, 23, 26, 30] ]  
 Given target = 5, return true.  
 Given target = 20, return false.*

**public class** Search2DMatrixII {  
 *//均从右上角开始搜索* **public boolean** searchMatrix(**int**[][] matrix, **int** target) {  
 **int** m = matrix.**length**, n = matrix[0].**length**;  
 **int** i = 0, j = n-1;  
 **while** (i<m && j>=0){  
 **if** (matrix[i][j]==target) **return true**;  
 **else if** (matrix[i][j]<target) i++;  
 **else** j--;  
 }  
 **return false**;  
 }  
}

**BestTimeToBuyAndSellStock \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*Say you have an array for which the ith 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.*

BestTimeToBuyAndSellStock DP 局部全局

**public class** BestTimeToBuyAndSellStock { *\* 最一般的方法是嵌套遍历找到最大买入卖出差,但超时.  
 一维动态规划(双重):记录出现过的最低价作为买入价格,计算当天售出的收益.  
 P(i)=Max(P(i-1), price[i]-B(i))  
 B(i)=Min(B(i-1), price[i])* **public int** maxProfit(**int**[] prices) {  
 **if**(prices.**length**<=1) **return** 0;  
 **int** min = prices[0], profit = 0;  
 **for**(**int** i=1; i<prices.**length**; i++){  
 min = Math.*min*(min, prices[i]);  
 profit = Math.*max*(profit, prices[i]-min);  
 }  
 **return** profit;  
 }  
}

*/\*\*Say you have an array for which the ith 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).  
 \* Created by eugene on 16/2/10.*

BestTimeToBuyAndSellStockII 贪心

**public class** BestTimeToBuyAndSellStockII {  
 */\*\*  
 \* 贪心方法:递增序前后元素差之和.* **public int** maxProfit(**int**[] prices) {  
 **int** profit = 0;  
 **for** (**int** i=1; i<prices.**length**; i++){  
 **if** (prices[i]>prices[i-1]) profit += prices[i]-prices[i-1];  
 }  
 **return** profit;  
 }  
}

*/\*\*Say you have an array for which the ith element is the price of a given stock on day i.  
 Design an algorithm to find the maximum profit. You may complete at most two transactions.  
 Note: You may not engage in multiple transactions at the same time  
 (ie, you must sell the stock before you buy again).  
 \* Created by eugene on 16/2/10.*

BestTimeToBuyAndSellStockIII *HARD DP*

**public class** BestTimeToBuyAndSellStockIII {

*两次一维动态规划组合:以第i天为分界线，计算第i天之前进行一次交易的最大收益preProfit[i]，  
 和第i天之后进行一次交易的最大收益postProfit[i]，  
 遍历的同时，max{preProfit[i] + postProfit[i]}(0≤i≤n-1)就是最大收益。  
 第i天之前和第i天之后进行一次的最大收益求法同Best Time to Buy and Sell Stock I。  
 上述方法时间复杂度O(n^2),超时.  
 优化:用数组暂存结果提速,时间复杂度O(n).* **public int** maxProfit(**int**[] prices) {  
 **if** (prices.**length** < 2) **return** 0;  
 **int** len = prices.**length**;  
 **int**[] preP = **new int**[len];  
 **int**[] postP = **new int**[len];  
 **int** min = prices[0];  
 **for** (**int** i=1; i<len; i++){  
 min = Math.*min*(min, prices[i]);  
 preP[i] = Math.*max*(preP[i-1], prices[i]-min); *//注意是i-1* }  
 **int** max = prices[len-1], profit = 0;  
 **for** (**int** i=len-2; i>=0; i--){ *//****TODO 巧妙点在于从后往前计算*** max = Math.*max*(max, prices[i]);  
 postP[i] = Math.*max*(postP[i+1], max-prices[i]); *//注意是i+1* profit = Math.*max*(profit, preP[i]+postP[i]); *//****TODO 同时计算最大收益*** }  
 **return** profit;  
 }  
}

*/\*\*Say you have an array for which the ith element is the price of a given stock on day i.  
 Design an algorithm to find the maximum profit. You may complete at most k transactions.  
 Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).  
 \* Created by eugene on 16/3/25.*

BestTimeToBuyAndSellStockIV *HARD DP*

**public class** BestTimeToBuyAndSellStockIV {  
 *//****TODO 重温*** */\*\*http://liangjiabin.com/blog/2015/04/leetcode-best-time-to-buy-and-sell-stock.html  
 \* 二维DP(可简化为一维):  
 错误的递推式:profit[i][j] = max(profit[i – 1][j], profit[i – 1][j – 1] + diff)  
 注意两次交易合并的情况,于是引入局部与全局变量,定义global[i][j]为在到达第i天时最多可进行j次交易的最大利润,  
 其中局部最优值是比较前一天并少交易一次的全局最优加上大于0的差值，和前一天的局部最优加上差值后相比，  
 两者之中取较大值，而全局最优比较局部最优和前一天的全局最优。  
 local[i][j] = max(global[i-1][j-1] + max(diff,0), local[i-1][j]+diff)  
 global[i][j] = max(local[i][j], global[i-1][j])  
 另外,当k大于天数时，其实就退化成Best Time to Buy and Sell Stock II,就不能用DP来做了.**\*/* **public int** maxProfit(**int** k, **int**[] prices) {  
 **if** (prices.**length** < 2) **return** 0;  
 **int** days = prices.**length**;  
 **if** (k >= days) **return** maxProfit(prices);  
 **int**[][] local = **new int**[days][k + 1];  
 **int**[][] global = **new int**[days][k + 1];  
 **for** (**int** i = 1; i < days ; i++) {  
 **int** diff = prices[i] - prices[i - 1];  
 **for** (**int** j = 1; j <= k; j++) {  
 local[i][j] = Math.*max*(global[i-1][j-1]+Math.*max*(diff, 0), local[i-1][j]+diff);  
 global[i][j] = Math.*max*(global[i-1][j], local[i][j]);  
 }  
 }  
 **return** global[days - 1][k];  
 }  
 *//退化成Best Time to Buy and Sell Stock II* **private int** maxProfit(**int**[] prices) {  
 **int** maxProfit = 0;  
 **for** (**int** i = 1; i < prices.**length**; i++) {  
 **if** (prices[i] > prices[i - 1]) {  
 maxProfit += prices[i] - prices[i - 1];  
 }  
 }  
 **return** maxProfit;  
 }  
}

*/\*\*Say you have an array for which the ith 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) with the following restrictions:  
 You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).  
 After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)  
 Example:  
 prices = [1, 2, 3, 0, 2]  
 maxProfit = 3  
 transactions = [buy, sell, cooldown, buy, sell]  
 \* Created by eugene on 16/6/10.*

BestTimeToBuyAndSellStockWithCooldown DP

**public class** BestTimeToBuyAndSellStockWithCooldown {  
 */\*\*https://leetcode.com/discuss/71354/share-my-thinking-process  
 \* DP: 列出初始转移状态再根据规律简化:  
 buy[i] = max(rest[i-1]-price[i], buy[i-1])  
 sell[i] = max(buy[i-1]+price[i], sell[i-1])  
 rest[i] = max(sell[i-1], buy[i-1], rest[i-1]) = sell[i-1]  
 最终简化为:  
 buy[i] = max(sell[i-2]-price, buy[i-1])  
 sell[i] = max(buy[i-1]+price, sell[i-1])  
 DP solution only depending on i - 1 and i - 2 can be optimized using O(1) space.  
 Let b1, b0 represent buy[i - 1], buy[i]  
 Let s2, s1, s0 represent sell[i - 2], sell[i - 1], sell[i]  
 First we define the initial states at i = 0:  
 We can buy. The max profit at i = 0 ending with a buy is -prices[0].  
 We cannot sell. The max profit at i = 0 ending with a sell is 0.**\*/* **public int** maxProfit(**int**[] prices) {  
 **if**(prices == **null** || prices.**length** <= 1) **return** 0;  
 **int** b0 = -prices[0], b1 = b0;  
 **int** s0 = 0, s1 = 0, s2 = 0;  
 **for**(**int** i = 1; i < prices.**length**; i++) {  
 b0 = Math.*max*(b1, s2 - prices[i]);  
 s0 = Math.*max*(s1, b1 + prices[i]);  
 b1 = b0; s2 = s1; s1 = s0;  
 }  
 **return** s0;  
 }  
}

**递归类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

ScrambleString HARD

**public class** ScrambleString { *\* 递归 O(3^n) 但是比下面的DP快* **public boolean** isScramble(String s1, String s2) {  
 **if** (s1.equals(s2)) **return true**;  
 **int**[] letters = **new int**[26];  
 **for** (**int** i=0; i<s1.length(); i++) {  
 letters[s1.charAt(i)-**'a'**]++;  
 letters[s2.charAt(i)-**'a'**]--;  
 }  
 **for** (**int** i=0; i<26; i++) **if** (letters[i]!=0) **return false**;  
 **for** (**int** i=1; i<s1.length(); i++) { //TODO i=1开始  
 **if** (isScramble(s1.substring(0,i), s2.substring(0,i))  
 && isScramble(s1.substring(i), s2.substring(i))) **return true**;  
 **if** (isScramble(s1.substring(0,i), s2.substring(s2.length()-i))  
 && isScramble(s1.substring(i), s2.substring(0,s2.length()-i))) **return true**;  
 }  
 **return false**;  
 }

*}*

*/\*\*Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are +, - and \*.  
 Example Input: Input: "2\*3-4\*5"  
 (2\*(3-(4\*5))) = -34  
 ((2\*3)-(4\*5)) = -14  
 ((2\*(3-4))\*5) = -10  
 (2\*((3-4)\*5)) = -10  
 (((2\*3)-4)\*5) = 10  
 Output: [-34, -14, -10, -10, 10]  
 \* Created by eugene on 16/5/5.*

DifferentWaysToAddParentheses

**public class** DifferentWaysToAddParentheses {  
 */\*\*  
 \* http://blog.csdn.net/foreverling/article/details/49742089  
 \* 分治法:对于输入字符串，若其中有运算符，则将其分为两部分，分别递归计算其值，然后将左值集合与右值集合进行交叉运算，  
 将运算结果放入结果集中；若没有运算符，则直接将字符串转化为整型数放入结果集中。* **public** List<Integer> diffWaysToCompute(String input) {  
 List<Integer> res = **new** ArrayList<>();  
 **for** (**int** i=0; i<input.length(); i++) {  
 **char** ch = input.charAt(i);  
 **if** (ch==**'+'** || ch==**'-'** || ch==**'\*'**) {  
 List<Integer> left = diffWaysToCompute(input.substring(0, i));  
 List<Integer> right = diffWaysToCompute(input.substring(i + 1));  
 **for** (**int** n : left) {  
 **for** (**int** m : right) {  
 **if** (ch==**'+'**) res.add(n+m);  
 **else if** (ch==**'-'**) res.add(n-m);  
 **else** res.add(n\*m);  
 }  
 }  
 }  
 }  
 **if** (res.size()==0) res.add(Integer.*parseInt*(input));  
 **return** res;  
 }  
}

**二叉树递归类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

PathSum

**public class** PathSum {  
 *//http://www.programcreek.com/2013/01/leetcode-path-sum/* **public boolean** hasPathSum(TreeNode root, **int** sum) {  
 **if** (root == **null**) **return false**;  
 **if** (root.**val** == sum && root.**left** == **null** && root.**right** == **null**) **return true**;  
 **return** hasPathSum(root.**left**, sum-root.**val**) || hasPathSum(root.**right**, sum-root.**val**);  
 }

*}*

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

PathSumII 树中回溯 与PathSum思路一致

**public class** PathSumII {*/\*\*https://leetcode.com/discuss/16980/dfs-with-one-linkedlist-accepted-java-solution \* dfs,递归,与Path Sum思路一致* **public** List<List<Integer>> pathSum(TreeNode root, **int** sum){  
 List<List<Integer>> result = **new** LinkedList<>();  
 pathSum(root, sum, **new** LinkedList<>(), result);  
 **return** result;  
 }  
 **public void** pathSum(TreeNode root, **int** sum, List<Integer> path, List<List<Integer>> result) {  
 **if** (root == **null**) **return**; *//统一判空处理* path.add(root.**val**);  
 **if** (root.**left**==**null** && root.**right**==**null** && sum==root.**val**) {  
 result.add(**new** LinkedList<>(path));//这里不能马上return  
 } **else** {  
 pathSum(root.**left**, sum-root.**val**, path, result);  
 pathSum(root.**right**, sum-root.**val**, path, result);  
 }  
 path.remove(path.size()-1);  
 }

*}*

*/\*\*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"]  
 \* Created by eugene on 16/5/10.*

BinaryTreePaths

**public class** BinaryTreePaths {**public** List<String> binaryTreePaths(TreeNode root) {  
 List<String> result = **new** ArrayList<>();  
 **if** (root!=**null**) searchBT(root, **""**, result);  
 **return** result;  
 }  
 **private void** searchBT(TreeNode root, String path, List<String> result) {  
 **if** (root.**left**==**null** && root.**right**==**null**) result.add(path+root.**val**);  
 **if** (root.**left**!=**null**) searchBT(root.**left**, path+root.**val**+**"->"**, result);  
 **if** (root.**right**!=**null**) searchBT(root.**right**, path+root.**val**+**"->"**, result);  
 }

*}*

*/\*\*Given a binary tree, find the maximum path sum.  
 For this problem, a path is defined as any sequence of nodes from  
 some starting node to any node in the tree along the parent-child connections.  
 The path does not need to go through the root.  
 For example: Given the below binary tree,  
 1  
 / \  
 2 3  
 Return 6.*

BinaryTreeMaximumPathSum HARD

**public class** BinaryTreeMaximumPathSum { *\* 递归:递归计算左,右子树最大路径和,从左,右子树最大路径和与根节点中选出单侧最大值,  
 再从单侧最大值,当前全路径和,全局最大路径和中选出最大值.  
 \* 注意节点值可能出现负值,比较时不能漏掉单节点情况.* **private int maxSum** = Integer.***MIN\_VALUE***;  
 **public int** maxPathSum(TreeNode root) {  
 maxPath(root);  
 **return maxSum**;  
 }  
 **private int** maxPath(TreeNode root) {  
 **if** (root==**null**) **return** 0;  
 **int** maxLeft = maxPath(root.**left**);  
 **int** maxRight = maxPath(root.**right**);**int** maxSide = Math.*max*(root.**val**, Math.*max*(root.**val**+maxLeft, root.**val**+maxRight)); *//****TODO 因为节点可能有负值,不能漏掉比较root.val***  
 *//* ***TODO 因为节点可能有负值,不能漏掉比较maxCurrent* maxSum** = Math.*max*(**maxSum**, Math.*max*(maxSide, maxLeft+root.**val**+maxRight));  
 **return** maxSide;//T***ODO 返回maxSide*** }  
}

*/\*\*Given a binary tree containing digits from 0-9 only,  
 each root-to-leaf path could represent a number.  
 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.  
 \* Created by eugene on 16/2/17.  
 \*/*

SumRootToLeafNumbers

**public class** SumRootToLeafNumbers {**public int** sumNumbers(TreeNode root) {  
 **return** helper(root, 0);  
 }  
 **private int** helper(TreeNode root, **int** sum) {  
 **if**(root == **null**) **return** 0;  
 **int** curSum = sum\*10+root.**val**;  
 **if**(root.**left**==**null** && root.**right**==**null**) **return** curSum; *//叶子节点* **return** helper(root.**left**, curSum) + helper(root.**right**, curSum);  
 }  
}

MaximumDepthOfBinaryTree

**public class** MaximumDepthOfBinaryTree {  
 **public int** maxDepth(TreeNode root) {  
 **if** (root==**null**) **return** 0;  
 **int** maxL = maxDepth(root.**left**);  
 **int** maxR = maxDepth(root.**right**);  
 **return** Math.*max*(maxL, maxR) + 1;  
 }  
}

MinimumDepthOfBinaryTree 类MaximumDepth

**public class** MinimumDepthOfBinaryTree {  
 *//类似Maximum Depth Of Binary Tree, 最后一行不同* **public int** minDepth(TreeNode root) {  
 **if**(root==**null**) **return** 0;  
 **int** left = minDepth(root.**left**);  
 **int** right = minDepth(root.**right**);  
 **return** (left==0 || right==0)? left+right+1 : Math.*min*(left,right)+1;  
 }

}

BalancedBinaryTree 类MaximumDepth

**public class** BalancedBinaryTree {  
 **public boolean** isBalanced(TreeNode root) {  
 **if** (root==**null**) **return true**;  
 **if** (maxBalancedDepth(root)==-1) **return false**;  
 **return true**;  
 }  
 **private int** maxBalancedDepth(TreeNode root){  
 **if** (root==**null**) **return** 0;  
 **int** lDepth = maxBalancedDepth(root.**left**);  
 **int** rDepth = maxBalancedDepth(root.**right**);  
 *//****TODO 增加以下剪枝条件提速 与****Maximum Depth Of Binary Tree不同处***if** (lDepth==-1 || rDepth==-1) **return** -1;  
 **if** (Math.*abs*(lDepth-rDepth)>1) **return** -1;  
 **return** Math.*max*(lDepth, rDepth)+1;  
 }

}

SameTree

**public class** SameTree {  
 **public boolean** isSameTree(TreeNode p, TreeNode q) {  
 **if**(p == **null** && q == **null**) **return true**;  
 **else if**(p != **null** && q != **null**){  
 **if**(p.**val** == q.**val**) **return** isSameTree(p.**left**, q.**left**) && isSameTree(q.**right**, p.**right**);  
 }   
 **return false**;  
 }

}

SymmetricTree 类SameTree

**public class** SymmetricTree { *\* 递归:思路类似Same Tree* **public boolean** isSymmetric(TreeNode root) {  
 **if** (root==**null**) **return true**;  
 **return** isSymmetricHelper(root.**left**, root.**right**);  
 }  
 **private boolean** isSymmetricHelper(TreeNode p, TreeNode q) {  
 **if**(p == **null** && q == **null**) **return true**;  
 **else if**(p != **null** && q != **null**){  
 **if**(p.**val** == q.**val**) *//****TODO 注意交换次序* return** isSymmetricHelper(p.**left**, q.**right**) && isSymmetricHelper(p.**right**, q.**left**);

else return false;  
 }**else return false**;  
 }

}

/\*\*

\* Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

1 The left subtree of a node contains only nodes with keys less than the node's key.

2 The right subtree of a node contains only nodes with keys greater than the node's key.

3 Both the left and right subtrees must also be binary search trees.

\* Created by eugene on 16/1/18.

ValidateBinarySearchTree

public class ValidateBinarySearchTree {

public boolean isValidBST(TreeNode root) {

return dfs(root, Double.NEGATIVE\_INFINITY, Double.POSITIVE\_INFINITY);

}

private boolean dfs(TreeNode root, double min, double max){

if (root == null) return true;

if (min>=root.val || root.val>=max) return false; //TODO 注意等于情况

return dfs(root.left, min, root.val) && dfs(root.right, root.val, max);

}

}

*/\*\*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: “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).”  
 \* Created by eugene on 16/4/29.*

LowestCommonAncestorOfBinarySearchTree

**public class** LowestCommonAncestorOfBinarySearchTree {  
 *//简洁版* **public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
 **while** ((root.**val** - p.**val**) \* (root.**val** - q.**val**) > 0)  
 root = p.**val** < root.**val** ? root.**left** : root.**right**;  
 **return** root;  
 }

}

LowestCommonAncestorOfBinaryTree 类MaximumDepth 较难分析

**public class** LowestCommonAncestorOfBinaryTree { *\* 若当前子树同时含p和q,根节点为LCA;若只有其一,则LCA为其一;若都不含,则为null.***public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
 **if** (root == **null** || root == p || root == q) **return** root;  
 TreeNode left = lowestCommonAncestor(root.**left**, p, q);  
 TreeNode right = lowestCommonAncestor(root.**right**, p, q);  
 **if**(left!=**null** && right!=**null**) **return** root;  
 **else if**(left==**null**) **return** right;  
 **else return** left;  
}

}

*/\*\*https://leetcode.com/problems/invert-binary-tree/  
 \* Invert a binary tree.  
 4  
 / \  
 2 7  
 / \ / \  
 1 3 6 9  
 to  
 4  
 / \  
 7 2  
 / \ / \  
 9 6 3 1  
 \* Created by eugene on 16/4/21.*

InvertBinaryTree

**public class** InvertBinaryTree {  
 **public** TreeNode invertTree(TreeNode root) {  
 invertHelper(root);  
 **return** root;  
 }  
 **private void** invertHelper(TreeNode root){  
 **if** (root==**null**) **return**;  
 TreeNode temp = root.**left**;  
 root.**left** = root.**right**;  
 root.**right** = temp;  
 invertHelper(root.**left**);  
 invertHelper(root.**right**);  
 }  
}

*/\*\*Given a binary tree, flatten it to a linked list in-place.  
 \* Created by eugene on 16/2/2.*

FlattenBinaryTreeToLinkedList 先序

**public class** FlattenBinaryTreeToLinkedList {  
 **private** TreeNode **prev** = **null**;  
 **public void** flatten(TreeNode root) {  
 **if** (root == **null**) **return**;  
 flatten(root.**right**);  
 flatten(root.**left**);  
 root.**right** = **prev**;//这行与下行可以交换秩序  
 root.**left** = **null**;  
 **prev** = root;  
 }

}

*/\*\*Given an array where elements are sorted in ascending order, convert it to a height balanced BST.  
 \* Created by eugene on 16/1/27.*

ConvertSortedArrayToBinarySearchTree 二分

**public class** ConvertSortedArrayToBinarySearchTree { *\* 递归:类似二分搜索.因为中序序列的中间数为根.* **public** TreeNode sortedArrayToBST(**int**[] nums) {  
 **if** (nums.**length** == 0) **return null**;  
 **return** sortedArrayToBST(nums, 0, nums.**length**-1);  
 }  
 **private** TreeNode sortedArrayToBST(**int**[] nums, **int** l, **int** r){  
 **if** (l>r) **return null**;  
 **int** m = (l+r)/2;  
 TreeNode root = **new** TreeNode(nums[m]);  
 root.**left** = sortedArrayToBST(nums, l, m-1);  
 root.**right** = sortedArrayToBST(nums, m+1, r);  
 **return** root;  
 }  
}

*/\*\*Given a singly linked list where elements are sorted in ascending order,  
 convert it to a height balanced BST.  
 \* Created by eugene on 16/1/28.*

ConvertSortedListToBinarySearchTree 快慢指针 二分

**public class** ConvertSortedListToBinarySearchTree {  
*//https://leetcode.com/discuss/83856/share-my-java-solution-1ms-very-short-and-concise***public** TreeNode sortedListToBST(ListNode head) {  
 **if**(head==**null**) **return null**;  
 **return** toBST(head, **null**);  
}  
**public** TreeNode toBST(ListNode head, ListNode tail){  
 ListNode slow = head, fast = head;  
 **if**(head==tail) **return null**;  
 **while**(fast!=tail && fast.**next**!=tail){ //TODO 注意是tail，不是null  
 fast = fast.**next**.**next**;  
 slow = slow.**next**;  
 }  
 TreeNode node = **new** TreeNode(slow.**val**);  
 node.**left** = toBST(head,slow);  
 node.**right** = toBST(slow.**next**,tail);  
 **return** node;  
}

}

ConstructBinaryTreeFromInorderAndPostorderTraversal

**public class** ConstructBinaryTreeFromInorderAndPostorderTraversal {*//****TODO 无重复*** *\* 递归:后序序列最后一个为根.中序序列的根的左侧序列为左子树,右侧序列为右子树.  
 根据左右序列长度可以在后序序列中找到左右子树.  
 \* 示例:  
 in-order: 4 2 5 (1) 6 7 3 8  
 post-order: 4 5 2 6 7 8 3 (1)* **public** TreeNode buildTree(**int**[] inorder, **int**[] postorder) {  
 **return** rebuild(postorder, inorder, 0, postorder.**length**-1, 0, inorder.**length**-1);  
 }  
 **private** TreeNode rebuild(**int**[] postorder, **int**[] inorder, **int** postL, **int** postR, **int** inL, **int** inR){  
 **if** (postL>postR || inL>inR) **return null**;  
 TreeNode root = **new** TreeNode(postorder[postR]);  
 **int** i = 0;  
 **for** (; i<inorder.**length**; i++){  
 **if** (inorder[i] == postorder[postR]) **break**;  
 }  
 **int** postLeftLen = i - inL;  
 root.**left** = rebuild(postorder, inorder, postL, postL+postLeftLen-1, inL, i-1);  
 root.**right** = rebuild(postorder, inorder, postL+postLeftLen, postR-1, i+1, inR);  
 **return** root;  
 }  
}

ConstructBinaryTreeFromPreorderAndInorderTraversal

**public class** ConstructBinaryTreeFromPreorderAndInorderTraversal {*//****TODO 无重复*** *\* 递归:先序序列第一个为根.中序序列的根的左侧序列为左子树,右侧序列为右子树.  
 根据左右序列长度可以在先序序列中找到左右子树.  
 \* 示例:  
 in-order: 4 2 5 (1) 6 7 3 8  
 pre-order: (1) 2 4 5 3 7 6 8* **public** TreeNode buildTree(**int**[] preorder, **int**[] inorder) {  
 **return** rebuild(preorder, inorder, 0, preorder.**length**-1, 0, inorder.**length**-1);  
 }  
 **private** TreeNode rebuild(**int**[] preorder, **int**[] inorder, **int** preL, **int** preR, **int** inL, **int** inR){  
 **if** (preL>preR || inL>inR) **return null**;  
 TreeNode root = **new** TreeNode(preorder[preL]);  
 **int** i = 0;  
 **for** (; i<inorder.**length**; i++){  
 **if** (inorder[i] == preorder[preL]) **break**;  
 }  
 **int** preLeftLen = i - inL;  
 root.**left** = rebuild(preorder, inorder, preL+1, preL+preLeftLen, inL, i-1);  
 root.**right** = rebuild(preorder, inorder, preL+preLeftLen+1, preR, i+1, inR);  
 **return** root;  
 }  
}

*/\*\*Two elements of a binary search tree (BST) are swapped by mistake.  
 Recover the tree without changing its structure.  
 Note: A solution using O(n) space is pretty straight forward.  
 Could you devise a constant space solution?  
 \* Created by eugene on 16/1/19.*

RecoverBinarySearchTree HARD

**public class** RecoverBinarySearchTree { *\* 空间复杂度O(1),注意pre存的是中序次序的上一个节点  
 \* 示例:The inorder traversal is : 1 3 4 6 7 8 10 13 14  
 \* If we change the value 4 and 8: 1 3 (8) 6 7 (4) 10 13 14* **private** TreeNode **first**=**null**, **second**=**null**, **pre**=**null**;  
 **public void** recoverTree(TreeNode root) {  
 inOrder(root);  
 **int** temp = **first**.**val**;  
 **first**.**val** = **second**.**val**;  
 **second**.**val** = temp;  
 }  
 **private void** inOrder(TreeNode root){  
 **if** (root == **null**) **return**;  
 inOrder(root.**left**);  
 **if** (**pre** == **null**) **pre** = root;  
 **else** {  
 **if** (**pre**.**val** > root.**val**){  
 **if** (**first** == **null**) **first** = **pre**; *//第一个错误节点:当前节点的上一节点* **second** = root;*//第二个错误节点:当前节点* }  
 **pre** = root;  
 }  
 inOrder(root.**right**);  
 }

*}*

*/\*\*Given n, generate all structurally unique BST's (binary search trees) that  
 store values 1...n.  
 \* Created by eugene on 16/1/16.*

UniqueBinarySearchTreesII DFS 交叉组合

**public class** UniqueBinarySearchTreesII { *\* I start by noting that 1..n is the in-order traversal for any BST with nodes 1 to n. So if I pick i-th node as my root, the left subtree will contain elements 1 to (i-1), and the right subtree will contain elements (i+1) to n. I use recursive calls to get back all possible trees for left and right subtrees and combine them in all possible ways with the root.* **public** List<TreeNode> generateTrees(**int** n) {  
 **if** (n==0) **return new** ArrayList<>();  
 **return** genTrees(1, n);  
 }  
 **private** List<TreeNode> genTrees(**int** start, **int** end) {  
 List<TreeNode> list = **new** ArrayList<>();  
 **if**(start>end) {  
 list.add(**null**);//TODO 注意空节点也要添加  
 **return** list;  
 } **else if**(start==end){  
 list.add(**new** TreeNode(start));  
 **return** list;  
 }  
 **for**(**int** i=start;i<=end;i++) {  
 List<TreeNode> left = genTrees(start, i-1);  
 List<TreeNode> right = genTrees(i+1, end);  
 **for**(TreeNode l: left) {  
 **for**(TreeNode r: right) {  
 TreeNode root = **new** TreeNode(i);  
 root.**left** = l;  
 root.**right** = r;  
 list.add(root);  
 }  
 }  
 }  
 **return** list;  
 }  
}

**二叉树类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*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.  
 \* Created by eugene on 16/1/16.*

UniqueBinarySearchTrees DP 卡特兰数

**public class** UniqueBinarySearchTrees {  
 *\* 基于以下原则的BST建树具有唯一性：  
 以i为根节点的树，其左子树由[0, i-1]构成， 其右子树由[i+1, n]构成。  
 \* 一维动态规划:二叉查找树可以任意取根，只要满足中序遍历有序的要求就可以。  
 从处理子问题的角度来看，选取一个结点为根，就把结点切成左右子树，  
 以这个结点为根的可行二叉树数量就是左右子树可行二叉树数量的乘积，  
 所以总的数量是将以所有结点为根的可行结果累加起来。  
 i=0, count[0]=1 //empty tree  
 i=1, count[1]=1 //one tree  
 i=2, count[2]=count[0]\*count[1] // 1 is root  
 + count[1]\*count[0] // 2 is root  
 Count[i] = ∑ Count[0...k] \* [ k+1....i], 0<=k<=i-1 (卡特兰数)  
 \* 时间上每次求解i个结点的二叉查找树数量的需要一个i步的循环，总体要求n次，  
 所以总时间复杂度是O(1+2+...+n)=O(n^2)。  
 空间上需要一个数组来维护，并且需要前i个的所有信息，所以是O(n)。* **public int** numTrees(**int** n) {  
 **int**[] count = **new int**[n+1]; //额外分配一个  
 count[0] = 1; count[1] = 1;  
 **for** (**int** i=2; i<=n; i++){  
 **for** (**int** k=0; k<=i-1; k++)  
 count[i] += count[k] \* count[i-k-1];  
 }  
 **return** count[n];  
 }  
}

BinaryTreeZigzagLevelOrderTraversal

**public class** BinaryTreeZigzagLevelOrderTraversal {**public** List<List> zigzagLevelOrder(TreeNode root) {  
 List<List> res = **new** ArrayList<>();  
 **if**(root == **null**) **return** res;  
 Queue<TreeNode> q = **new** LinkedList<>();  
 q.add(root);  
 **boolean** order = **true**;  
 **while**(!q.isEmpty()) {

int size = q.size();  
 List<Integer> tmp = **new** ArrayList<>();  
 **for**(**int** i = 0; i < size; ++i) {  
 TreeNode n = q.poll();  
 **if**(order) tmp.add(n.**val**);  
 **else** tmp.add(0, n.**val**);  
 **if**(n.**left** != **null**) q.add(n.**left**);  
 **if**(n.**right** != **null**) q.add(n.**right**);  
 }  
 res.add(tmp);  
 order = !order;  
 }  
 **return** res;  
 }

}

PopulatingNextRightPointersInEachNode 完全二叉

PopulatingNextRightPointersInEachNodeII 普通二叉 通用解法

*/\*\*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.*

**public class** PopulatingNextRightPointersInEachNodeII {  
 *//https://leetcode.com/discuss/67291/java-solution-with-constant-space 通用解法* **public void** connect(TreeLinkNode root) {  
 TreeLinkNode levelHead = **new** TreeLinkNode(0);  
 TreeLinkNode pre = levelHead, cur = root;  
 **while** (cur != **null**) {  
 **if** (cur.**left** != **null**) {  
 pre.**next** = cur.**left**;  
 pre = pre.**next**;  
 }  
 **if** (cur.**right** != **null**) {  
 pre.**next** = cur.**right**;  
 pre = pre.**next**;  
 }  
 cur = cur.**next**;  
 **if** (cur == **null**) {  
 pre = levelHead;  
 cur = levelHead.**next**;  
 levelHead.**next** = **null**;  
 }  
 }  
 }

}

**链表逆转类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

reverseList **递归 迭代**

**public** ListNode reverseList(ListNode head) {  
 **if**(head==**null** || head.**next**==**null**) **return** head;  
 ListNode second = head.**next**; *//pin the second node* ListNode nHead = reverseList(second);  
 second.**next** = head;  
 head.**next** = **null**; *//****TODO set first's next to be null* return** nHead;  
}

*//***链表逆转(***迭代)***public** ListNode reverseList1(ListNode head) {  
 **if** (head==**null**) **return null**;  
 ListNode pre = head, curr = head.**next**;  
 **while** (curr!=**null**){  
 ListNode temp = curr.**next**;  
 curr.**next** = pre;  
 pre = curr;  
 curr = temp;  
 }  
 head.**next** = **null**; *//****TODO 易漏* return** pre;  
}

*/\*\*https://leetcode.com/discuss/25580/simple-java-solution-with-clear-explanation  
 dummy-> 1 -> 2 -> 3 -> 4 -> 5 m=2 n=4 有 pre = 1, last= 2, cur= 3  
 first reversing : dummy->1 - 3 - 2 - 4 - 5; pre = 1, last= 2, cur= 4  
 second reversing: dummy->1 - 4 - 3 - 2 - 5; pre = 1, last= 2, cur= 5 (finish)*

reverseBetween

**public** ListNode reverseBetween(ListNode head, **int** m, **int** n) {  
 **if**(head == **null**) **return null**;  
 ListNode dummy = **new** ListNode(0);  
 dummy.**next** = head;  
 ListNode pre = dummy; *// make a pointer pre as a marker for the node before reversing* **for**(**int** i=0; i<m-1; i++) pre = pre.**next**;  
 ListNode last = pre.**next**; *// a pointer to the beginning of a sub-list that will be reversed* ListNode cur = last.**next**;   
 **for**(**int** i=0; i<n-m; i++) { //注意迭代次数  
 last.**next** = cur.**next**;  
 cur.**next** = pre.**next**;//不是cur.next=last.next;  
 pre.**next** = cur;  
 cur = last.**next**;  
 }  
 **return** dummy.**next**;  
}

*/\*\*  
 \* Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.  
 If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.  
 You may not alter the values in the nodes, only nodes itself may be changed.  
 Only constant memory is allowed.  
 For example,  
 Given this linked list: 1->2->3->4->5  
 For k = 2, you should return: 2->1->4->3->5  
 For k = 3, you should return: 3->2->1->4->5  
 \* Created by DCLab on 11/8/2015.*

ReverseNodesInKGroup *HARD*

**public class** ReverseNodesInKGroup {**public** ListNode reverseKGroup(ListNode head, **int** k) {  
 **if**(head == **null** || k == 1) **return** head;  
 ListNode dummy = **new** ListNode(0); *//因为有“放到链表首位”的操作* dummy.**next** = head;  
 ListNode pre = dummy;  
 **int** i = 0;  
 **while**(head != **null**){  
 i++;  
 **if**(i % k ==0){  
 pre = reverse(pre, head.**next**);  
 head = pre.**next**;  
 }**else** {  
 head = head.**next**;  
 }  
 }  
 **return** dummy.**next**;  
 }

*/\*\*****TODO 区间逆转,注意last,pre,next指针不变,cur指针移动*** *\* 首先，搞清楚怎么逆转一个单链表。其实O(n)就可以了。  
 \* 第一个肯定是last one，然后我们每遍历到一个node，就把它放到最链表的首位，最后一个就成为第一个了。  
 \* Reverse a link list between pre and next exclusively  
 \* an example:  
 \* a linked list:  
 \* 0->1->2->3->4->5->6  
 \* | |  
 \* pre next  
 \* after call pre = reverse(pre, next)  
 \* 0->3->2->1->4->5->6  
 \* | |  
 \* pre next**\** ***@return*** *the reversed list's last node, which is the precedence of parameter next  
 \*/***public** ListNode reverse(ListNode pre, ListNode next){  
 ListNode last = pre.**next**; *//where first will be doomed "last"* ListNode cur = last.**next**;  
 **while**(cur != next){  
 last.**next** = cur.**next**;  
 cur.**next** = pre.**next**;  
 pre.**next** = cur;  
 cur = last.**next**;  
 }  
 **return** last;  
}

}

*/\*\*  
 \* 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.*

SwapNodesInPairs 递归

**public class** SwapNodesInPairs { **public** ListNode swapPairs(ListNode head) {  
 **if** (head==**null** || head.**next**==**null**) **return** head;  
 ListNode second = head.**next**;  
 head.**next** = swapPairs(second.**next**);  
 second.**next** = head;  
 **return** second;  
 }

}

**链表类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*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.  
 Follow up: Can you solve it without using extra space*

LinkedListCycleII 快慢指针

**public class** LinkedListCycleII { *\* 假设链表有环，环长Y，环以外的长度是X。快慢两指针从起点出发,最后相遇在K处,设环起点到K处长度为K. 有Y-K=X,意思是两指针相遇后再走X就回到了环的起点.* **public** ListNode detectCycle(ListNode head) {  
 ListNode slow = head， fast = head;  
 **while** (fast!=**null** && fast.**next**!=**null**){  
 slow = slow.**next**;  
 fast = fast.**next**.**next**;  
 **if** (slow==fast) **break**;  
 }  
 **if** (fast==**null** || fast.**next**==**null**) **return null**; *//****TODO 易漏,判断无环情况*** slow = head;  
 **while** (fast!=slow){  
 slow = slow.**next**;  
 fast = fast.**next**;  
 }  
 **return** slow;  
 }  
}

*/\*\*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?  
 \* Created by eugene on 16/4/29.*

PalindromeLinkedList 快慢指针

**public class** PalindromeLinkedList {  
 */\*\*图 https://leetcode.com/discuss/78152/java-easy-to-understand  
 \* 先找到重点,逆转后半部分,比较* **public boolean** isPalindrome(ListNode head) {  
 ListNode fast = head;  
 ListNode slow = head;  
 **while**(fast != **null** && fast.**next** != **null**) {  
 slow = slow.**next**;  
 fast = fast.**next**.**next**;  
 }  
 slow = reverseList(slow);//此方法省略，自行补全  
 **while**(slow != **null** && head.**val** == slow.**val**) {  
 head = head.**next**;  
 slow = slow.**next**;  
 }  
 **return** slow == **null**;  
 }

}

*/\*\*Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.  
 You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.  
 Example:  
 Given 1->2->3->4->5->NULL,  
 return 1->3->5->2->4->NULL.  
 Note: The relative order inside both the even and odd groups should remain as it was in the input.  
 The first node is considered odd, the second node even and so on ...  
 \* Created by eugene on 16/6/22.*

OddEvenLinkedList 快慢指针

**public class** OddEvenLinkedList {  
 *//注意边界* **public** ListNode oddEvenList(ListNode head) {  
 **if** (head==**null** || head.**next**==**null**) **return** head;  
 ListNode odd = head, even = head.**next**, evenHead = even;  
 **while** (even!=**null** && even.**next**!=**null**){  
 odd.**next** = odd.**next**.**next**;  
 even.**next** = even.**next**.**next**;  
 odd = odd.**next**;  
 even = even.**next**;  
 }  
 odd.**next** = evenHead;  
 **return** head;  
 }

}

*/\*\*Given a linked list, remove the nth 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.*

RemoveNthFromEnd 快慢指针 onepass

**public class** RemoveNthFromEnd {  
 *//one pass* **public** ListNode removeNthFromEnd(ListNode head, **int** n) {  
 ListNode dummy = **new** ListNode(0);  
 dummy.**next** = head;  
 ListNode slow = dummy, fast = dummy;  
 *//Move fast in front so that the gap between slow and fast becomes n* **for**(**int** i=1; i<=n+1; i++) { *//****TODO 注意边界*** fast = fast.**next**;  
 }  
 **while**(fast != **null**) {*//Move fast to the end, maintaining the gap* slow = slow.**next**;  
 fast = fast.**next**;  
 }  
 slow.**next** = slow.**next**.**next**;*//Skip the desired node* **return** dummy.**next**;  
 }

}

*/\*\*  
 \* 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.*

PartitionList 双指针

**public class** PartitionList { *\* maintain two queues, the first one stores all nodes with val less than x ,  
 and the second queue stores all the rest nodes. Then concat these two queues.  
 Remember to set the tail of second queue a null next, or u will get TLE.* **public** ListNode partition(ListNode head, **int** x) {  
 ListNode dummy1 = **new** ListNode(0), dummy2 = **new** ListNode(0);  
 ListNode curr1 = dummy1, curr2 = dummy2; *//current tails of the two queues;* **while** (head!=**null**){  
 **if** (head.**val**<x) {  
 curr1.**next** = head;  
 curr1 = head;  
 }**else** {  
 curr2.**next** = head;  
 curr2 = head;  
 }  
 head = head.**next**;  
 }  
 curr2.**next** = **null**; *//* ***avoid cycle in linked list. otherwise u will get TLE.*** curr1.**next** = dummy2.**next**;  
 **return** dummy1.**next**;  
 }

}

*/\*\*Given a list, rotate the list to the right by k places, where k is non-negative.  
 把后k个rotate到list前面去，k可以超过list本身长度。  
 For example:  
 Given 1->2->3->4->5->NULL and k = 2,  
 return 4->5->1->2->3->NULL.  
 \* Created by DCLab on 12/14/2015.*

RotateList 首尾连

**public class** RotateList {  
 *//因为k可以超过list本身长度，可以先首尾连起来，然后找到该断开的地方断开。* **public** ListNode rotateRight(ListNode head, **int** k) {  
 **if** (head == **null** || k == 0) **return** head;  
 ListNode p = head;  
 **int** len = 1; *//since p is already point to head* **while** (p.**next** != **null**) {  
 len++;  
 p = p.**next**;  
 }  
 p.**next** = head; *//form a loop* **for** (**int** i=0; i<len-k%len; i++) p = p.**next**;  
 head = p.**next**; *//now p points to the prev of the new head* p.**next** = **null**;  
 **return** head;  
 }  
}

MergeTwoSortedLists 递归

**public** ListNode mergeTwoLists(ListNode l1, ListNode l2) {  
 **if**(l1==**null**) **return** l2;  
 **if**(l2==**null**) **return** l1;  
 **if**(l1.**val**<l2.**val**) {  
 l1.**next** = mergeTwoLists(l1.**next**, l2);  
 **return** l1;  
 } **else**{  
 l2.**next** = mergeTwoLists(l1, l2.**next**);//最好不要写成l2.**next**, l1  
 **return** l2;  
 }  
}

*/\*\*二分法（分治法）  
 \* http://blog.csdn.net/linhuanmars/article/details/19899259  
 \* 时间复杂度：nklogk \* 分析：假设总共有k个list，每个list的最大长度是n，那么运行时间满足递推式T(k) = 2T(k/2)+O(n\*k)。 \* 根据主定理，可以算出算法的总复杂度是O(nklogk)*

*MergeKSortedLists HARD 归并排序模式*

**public class** MergeKSortedLists { *\* 二分法（分治法）* **public** ListNode mergeKLists(ListNode[] lists) {  
 **if**(lists.**length**==0) **return null**;  
 **return** mergeSort(lists, 0, lists.**length**-1);  
 }  
 **private** ListNode mergeSort(ListNode[] lists, **int** l, **int** r){ *//归并排序范式* **if**(l>=r) **return** lists[l];  
 **else**{ *//l<r* **int** m = l+(r-l)/2;  
 **return** mergeTwoLists(mergeSort(lists, l, m), mergeSort(lists, m+1, r));  
 }  
 }

}

CopyListWithRandomPointer HARD

**public class** CopyListWithRandomPointer {**class** RandomListNode {  
 **int label**;  
 RandomListNode **next**, **random**;  
 RandomListNode(**int** x) { **this**.**label** = x; }  
 };  
 *//仅使用哈希表和双指针,速度更快* **public** RandomListNode copyRandomList(RandomListNode head) {  
 **if** (head==**null**) **return null**;  
 Map<RandomListNode, RandomListNode> map = **new** HashMap<>();  
 RandomListNode newHead = **new** RandomListNode(head.**label**);  
 map.put(head, newHead);  
 RandomListNode p = head, q = newHead;  
 p = p.**next**;  
 **while** (p!=**null**){  
 RandomListNode newNode = **new** RandomListNode(p.**label**);  
 map.put(p, newNode);  
 q.**next** = newNode;  
 q = newNode;  
 p = p.**next**;  
 }  
 p = head; q = newHead;  
 **while** (p!=**null**){  
 **if** (p.**random**!=**null**) q.**random** = map.get(p.**random**);  
 p = p.**next**;  
 q = q.**next**;  
 }  
 **return** newHead;  
 }

}

**数组操作类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

RotateArray

**public class** RotateArray {  
 */\*\*  
 \* O(1) space and in O(n) time.  
 Assuming we are given {1,2,3,4,5,6} and order 2. The basic idea is:  
 1. Divide the array two parts: 1,2,3,4 and 5, 6  
 2. Rotate first part: 4,3,2,1,5,6  
 3. Rotate second part: 4,3,2,1,6,5  
 4. Rotate the whole array: 5,6,1,2,3,4  
 或者 6,5,4,3,2,1 -> 5,6|1,2,3,4 (代码简洁)* **public void** rotate(**int**[] nums, **int** k) {  
 k = k%nums.**length**;  
 **if** (k==0) **return**;  
 reverse(nums, 0, nums.**length** - 1);  
 reverse(nums, 0, k - 1);  
 reverse(nums, k, nums.**length** - 1);  
 }  
 **private void** reverse(**int**[] nums, **int** start, **int** end){  
 **while** (start < end) {  
 **int** temp = nums[start];  
 nums[start] = nums[end];  
 nums[end] = temp;  
 start++;  
 end--;  
 }  
 }  
}

*/\*\*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.  
 \* 对序列大小的比较做出定义：两个长度相同的序列，从两者的第一个元素开始向后比较， 直到出现一个不同元素（也可能就是第它们的第一个元素），该元素较大的序列为大， 反之序列为小；若一直到最后一个元素都相同，那么两个序列相等。 对于一个任意序列，最小的序列是增序，最大的序列为减序。  
 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*

NextPermutation

**public class** NextPermutation { *\* 假设一个有m个元素的序列pn，其下一个较大序列为pn+1。(这里序号从1开始)  
 \* 1} 若pn最右端的2个元素构成一个增序子序列，那么直接反转这2个元素使该子序列成为减序，即可得到pn+1。  
 \* 2} 若pn最右端一共有连续的s个元素构成一个减序子序列，令i = m-s，则有pn(i) < pn(i+1)，其中pn(i)表示排列pn的第i个元素。  
 1. 要保证是下一个较大的序列，必须保持pn(i)左边的元素不动，并在子集s{pn(i+1), pn(i+2), ..., pn(m)}中找出所有比pn(i)大的元素中最小的一个pn(j)，然后将二者调换位置。  
 2. 接着,需要使新子集{pn(i+1), pn(i+2), ..., pn(i), ...,pn(m)}成为最小序列即得到pn+1,然而新子集仍保持减序，那么直接将其反转即可得到.  
 ex. 3 6 4 2  
 4 6 3 2 (step1 swap)  
 4 2 3 6 (step2 reverse)* **public static void** nextPermutation(**int**[] nums) {  
 **if** (nums==**null** || nums.**length**<=1) **return**;  
 **int** len = nums.**length**, i = 0;  
 **if** (nums[len-1]>nums[len-2]){ *//pn最右端的2个元素构成一个增序子序列  
 swap*(nums, len-1, len-2); **return**;  
 }  
 **for** (i=len-2; i>=0; i--){ *//pn最右端一共有连续的s个元素构成一个减序子序列* **if** (i==0 && nums[0]>=nums[1]){ *//pn是一个减序列, 全体逆转返回  
 reverse*(nums, 0, len-1); **return**;  
 }  
 **if** (nums[i]<nums[i+1]) **break**; *//找到置换点i* }  
 **int** theI = i;  
 **for** (i = len-1; i>theI; i--){  
 **if** (nums[theI]<nums[i]) **break**; *//找到第二个置换点i* }  
 *swap*(nums, theI, i);  
 *reverse*(nums, theI+1, len-1); *//逆转theI后的部分* }  
 **private static void** reverse(**int**[] nums, **int** l, **int** r) {  
 **for**(**int** i=l; i<=(l+r)/2; i++) *//奇偶通用  
 swap*(nums, i, l+r-i);  
 }

}

*/\*\*  
 \* Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array Note: You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1 and nums2 are m and n respectively.  
 \* Created by DCLab on 1/9/2016.*

MergeSortedArray 从尾部开始

**public class** MergeSortedArray {  
 **public void** merge(**int**[] nums1, **int** m, **int**[] nums2, **int** n) {  
 **int** i = m-1, j = n-1, k = m+n-1;  
 **while**(i>=0 && j>=0) {  
 **if**(nums1[i]>nums2[j]) nums1[k--] = nums1[i--];  
 **else** nums1[k--] = nums2[j--];  
 }  
 **while**(j>=0) nums1[k--] = nums2[j--];  
 }

}

*/\*\*  
 \* 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.  
 Could you come up with an one-pass algorithm using only constant space?*

SortColors onepass 双指针

**public class** SortColors { *//1-pass every elem left low pointer is 0, elem right high pointer is 2* **public void** sortColors(**int**[] nums) {  
 **if**(nums==**null** || nums.**length**<2) **return**;  
 **int** l = 0, r = nums.**length**-1;  
 **for**(**int** i=0; i<=r; ) {  
 **if**(nums[i]==0) { *// swap nums[i] and nums[l] and i,l both ++  
 swap*(nums, i, l);  
 i++; l++; *//****TODO i也++*** }**else if**(nums[i]==2) { *//swap nums[i] and nums[r] and r--;  
 swap*(nums, i, r);  
 r--; *//****TODO i不++*** }**else** {  
 i++;  
 }  
 }  
 }

}

*/\*\*Given an unsorted integer array, find the first missing positive integer.  
 For example, Given [1,2,0] return 3, and [3,4,-1,1] return 2.  
 Your algorithm should run in O(n) time and uses constant space.*

FirstMissingPositive HARD inplace tricky

**public class** FirstMissingPositive {  
 */\*\*交换数组元素，使得数组中第i位存放数值(i+1)。最后遍历数组，寻找第一个不符合此要求的元素，  
 \* 返回这个元素。整个过程需要遍历两次数组，复杂度为O(n)。* **public int** firstMissingPositive(**int**[] nums) {  
 **int** n = nums.**length**;  
 **for**(**int** i=0; i<n; i++) {  
 **while**(nums[i] > 0 && nums[i] <= n && nums[i] != nums[nums[i] - 1])  
 swap(nums, i, nums[i]-1);  
 }  
 **for**(**int** i=0; i<n; i++) **if**(nums[i] != i+1) **return** i+1;  
 **return** n+1;  
 }

}

*/\*\*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.  
 \* Created by eugene on 16/3/19.*

MajorityElement

**public class** MajorityElement {  
 *//O(n) time and O(1) space* **public int** majorityElement(**int**[] num) {  
 **int** major=num[0], count = 1;  
 **for**(**int** i=1; i<num.**length**;i++){  
 **if**(count==0){  
 count++;  
 major=num[i];  
 }**else if**(major==num[i]){  
 count++;  
 }**else** count--;  
 }  
 **return** major;  
 }

*}*

*/\*\*Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.  
 The algorithm should run in linear time and in O(1) space.  
 \* Created by eugene on 16/4/24.*

MajorityElementII

**public class** MajorityElementII {  
 */\*\*  
 \* Moore's Voting Algorithm:根据题目有两个候选者,分别为这两个数设置count计数,遍历数组若出现相同的数则加一,否则都减一.  
 当任一count为0时,候选者替换为当前数.再遍历一次数组,确认出现次数超过⌊ n/3 ⌋.* **public** List<Integer> majorityElement(**int**[] nums) {  
 ArrayList<Integer> res = **new** ArrayList<>();  
 **int** len = nums.**length**;  
 **if** (len==0) **return** res;  
 **int** fir = 0, sec = 1; *//****TODO 使fir,sec两数保持不同* int** firCount = 0, secCount = 0;  
 **for** (**int** i=0; i<len; i++) {  
 **if** (fir == nums[i]) firCount++;  
 **else if** (sec == nums[i]) secCount++;  
 **else if** (firCount == 0) {  
 fir = nums[i];  
 firCount = 1;  
 } **else if** (secCount == 0) {  
 sec = nums[i];  
 secCount = 1;  
 } **else** {  
 firCount--;  
 secCount--;  
 }  
 }  
 firCount = 0; secCount = 0;  
 **for** (**int** i : nums) { *// Count again for x1, x2* **if** (i == fir) firCount++;  
 **else if** (i == sec) secCount++;  
 }  
 **if** (firCount>len/3) res.add(fir);  
 **if** (secCount>len/3) res.add(sec);  
 **return** res;  
 }  
}

*/\*\*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]  
 ]*

PascalTriangle PascalTriangleII 三角 简洁

**public class** PascalTriangle { *\* 简洁. 与Pascal's Triangle II方法一致.* **public** List<List<Integer>> generate(**int** numRows) {  
 List<List<Integer>> res = **new** ArrayList<>();  
 ArrayList<Integer> row = **new** ArrayList<>();  
 **for**(**int** i=0; i<numRows; i++) {  
 row.add(0, 1);  
 **for**(**int** j=1; j<row.size()-1; j++)  
 row.set(j, row.get(j)+row.get(j+1));  
 res.add(**new** ArrayList<>(row));  
 }  
 **return** res;  
 }

}

SetMatrixZeroes 2D

**public class** SetMatrixZeroes { *\* O(1)空间解法：用第0行和第0列来记录第1~m-1行和第1~n-1列是否需要置0。  
 \* 而用两个变量记录第0行和第0列是否需要置0。  
 \** ***TODO 注意合并写法,不易出错*****public void** setZeroes(**int**[][] matrix) {  
 **int** m = matrix.**length**, n = matrix[0].**length**;  
 **boolean** rZero = **false**, cZero = **false**;  
 **for**(**int** i=0; i<m; i++){  
 **for**(**int** j=0; j<n; j++){  
 **if**(matrix[i][j]==0) {  
 **if**(i==0) rZero = **true**;  
 **if**(j==0) cZero = **true**;  
 matrix[0][j] = 0;  
 matrix[i][0] = 0;  
 }  
 }  
 }  
 **for**(**int** i=1; i<m; i++){ *//合并写法* **for**(**int** j=1; j<n; j++){  
 **if**(matrix[i][0]==0 || matrix[0][j]==0) matrix[i][j] = 0;  
 }  
 }  
 **if**(rZero) **for**(**int** j=0; j<n; j++) matrix[0][j] = 0;  
 **if**(cZero) **for**(**int** i=0; i<m; i++) matrix[i][0] = 0;  
 }  
}

*/\*\*Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.  
 For example, Given n = 3,  
 You should return the following matrix:  
 [ [ 1, 2, 3 ],  
 [ 8, 9, 4 ],  
 [ 7, 6, 5 ] ]*

SpiralMatrix SpiralMatrixII 2D

**public class** SpiralMatrixII {  
 *//与SpiralMatrix思路一致,此法不易出错* **public int**[][] generateMatrix(**int** n){  
 **int**[][] res = **new int**[n][n];  
 n -= 1;  
 **int** m = n, r = 0, c = 0, k = 1;  
 **while** (r<=m && c<=n){  
 **for** (**int** j=c; j<=n; j++) res[r][j] = k++;  
 r++;  
 **for** (**int** i=r; i<=m; i++) res[i][n] = k++;  
 n--;  
 **if** (r<=m){//TODO 矩形需要加上此条件  
 **for** (**int** j=n; j>=c; j--) res[m][j] = k++;

m--;  
 }  
 **if** (c<=n){//TODO矩形需要加上此条件  
 **for** (**int** i=m; i>=r; i--) res[i][c] = k++;

c++;  
 }  
 }  
 **return** res;  
 }

}

*/\*\*You are given an n x n 2D matrix representing an image.  
 Rotate the image by 90 degrees (clockwise). 方阵  
 Follow up: Could you do this in-place?*

RotateImage

**public class** RotateImage {  
 */\*\*先行逆转再转置  
 \* 1 2 3 7 8 9 7 4 1  
 \* 4 5 6 => 4 5 6 => 8 5 2  
 \* 7 8 9 1 2 3 9 6 3* **public void** rotate(**int**[][] matrix) {  
 **int** len = matrix.**length**;  
 **for**(**int** i=0; i<len/2; i++){  
 **int**[] temp = matrix[i];  
 matrix[i] = matrix[len-i-1];  
 matrix[len-i-1] = temp;  
 }  
 **for**(**int** i=0; i<len; i++){  
 **for**(**int** j=i; j<len; j++){  
 **int** temp = matrix[i][j];  
 matrix[i][j] = matrix[j][i];  
 matrix[j][i] = temp;  
 }  
 }  
 }  
}

InsertInterval HARD 清晰思路

**public class** InsertInterval {

//*the intervals were initially sorted according to their start times.*  
 **public static class** Interval {  
 **int start**;  
 **int end**;  
 Interval() { **start** = 0; **end** = 0; }  
 Interval(**int** s, **int** e) { **start** = s; **end** = e; }  
 }**public** List<Interval> insert(List<Interval> intervals, Interval newInterval) {  
 Interval span = newInterval;  
 List<Interval> result = **new** LinkedList<>();  
 **int** i = 0;  
 **while** (i<intervals.size() && intervals.get(i).**end**<span.**start**)  
 result.add(intervals.get(i++));  
 **while** (i<intervals.size() && intervals.get(i).**start**<=span.**end**) {*//merge all overlapping intervals* span = **new** Interval(  
 Math.*min*(span.**start**, intervals.get(i).**start**),  
 Math.*max*(span.**end**, intervals.get(i).**end**));  
 i++;  
 }  
 result.add(span); *// add the union of intervals we got* **while** (i<intervals.size()) result.add(intervals.get(i++));*// add all the rest* **return** result;  
 }

}

MergeIntervals HARD 清晰思路

**public class** MergeIntervals {  
**public** List<Interval> merge(List<Interval> intervals) {  
 **if** (intervals.size() <= 1) **return** intervals;  
 Collections.*sort*(intervals, **new** Comparator<Interval>() {  
 @Override  
 **public int** compare(Interval i1, Interval i2) {  
 **return** Integer.*compare*(i1.**start**, i2.**start**);  
 }  
 });  
 List<Interval> result = **new** LinkedList<>();  
 **int** preL = intervals.get(0).**start**, preR = intervals.get(0).**end**;  
 **for** (Interval span : intervals) {  
 **if** (span.**start** <= preR) *// Overlapping intervals, move the preR if needed* preR = Math.*max*(preR, span.**end**);  
 **else** { *// Disjoint intervals, add the previous one and reset bounds* result.add(**new** Interval(preL, preR));  
 preL = span.**start**;  
 preR = span.**end**;  
 }  
 }  
 result.add(**new** Interval(preL, preR)); *// Add the last interval* **return** result;  
}

}

**特殊题型 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

GasStation 贪心 tricky

**public class** GasStation {  
 */\*\*贪心:关键是要可以总结出来这道题目的属性，注意Note这个地方，其属性主要有两个：  
 1 如果总的gas - cost小于零的话，那么没有解返回-1;  
 2 如果前面所有的gas - cost加起来小于零，那么前面所有的点都不能作为出发点。* **public int** canCompleteCircuit(**int**[] gas, **int**[] cost) {  
 **int** start = 0, preRemain = 0, totalRemain = 0;  
 **for** (**int** i=0; i<gas.**length**; i++){  
 **int** dif = gas[i]-cost[i];  
 **if** (preRemain>=0) preRemain += dif;  
 **else**{  
 preRemain = dif;  
 start = i;  
 }  
 totalRemain += dif;  
 }  
 **if** (totalRemain < 0) **return** -1;  
 **else return** start;  
 }  
}

*/\*\*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.*

JumpGame 贪心 DP会超时

**public class** JumpGame { *\* 注意与Jump Game II的区别。  
 \* 如果采用DP（复杂度为O(n^2)，大数据测试用例将无法通过。  
 \* 贪心思路：（复杂度O(n)）* **public boolean** canJump(**int**[] nums) {  
 **int** maxReach = 0;  
 **for**(**int** i=0; i<nums.**length**; i++){  
 **if**(i>maxReach) **return false**;  
 maxReach = Math.*max*(nums[i]+i, maxReach);*//每次都更新当前最大可达值* }  
 **return true**;  
 }

}

*/\*\*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.)*

JumpGameII HARD 贪心

**public class** JumpGameII {  
 */\*\*首先明白，这个题只要我们求跳数，怎么跳、最后距离是多少都没让求，不要做无必要的计算。  
 \* 如果采用DP，大数据测试用例将无法通过。  
 \* 分析图：http://www.cnblogs.com/lichen782/p/leetcode\_Jump\_Game\_II.html  
 \* 贪心思路：lastMax:用最小跳数jumps可达的最大距离；  
 \* maxReach:用jumps + 1跳可达的最大距离；（贪心值）  
 \* maxReach = max(i+nums[i], maxReach), 0 <= i <= lastMax* **public int** jump(**int**[] nums) {  
 **int** minJumps = 0, lastMax = 0, maxReach = 0;  
 **for** (**int** i=0; i<nums.**length**; ++i) {  
 **if** (i>lastMax) { *//****TODO 注意此处的巧妙*** minJumps++;  
 lastMax = maxReach; *//更新* }  
 maxReach = Math.*max*(maxReach, i+nums[i]); *//更新* }  
 **return** minJumps;  
 }  
}

**运算类 进位 Math \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*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].*

GrayCode Math

**public class** GrayCode {  
 *//G(i) = i^(i/2)* **public** List<Integer> grayCode(**int** n) {  
 List<Integer> result = **new** LinkedList<>();  
 **for** (**int** i=0; i < 1<<n; i++) result.add(i ^ i>>1);  
 **return** result;  
 }

}

ReverseInteger

**public class** ReverseInteger {

// assume that your function returns 0 when the reversed integer overflows.**public int** reverse(**int** x) {  
 **long** rev = 0;//TODO long类型  
 **while**(x!= 0){  
 rev = rev\*10 + x%10;  
 x = x/10;  
 **if**(rev>Integer.***MAX\_VALUE*** || rev<Integer.***MIN\_VALUE***) **return** 0;//TODO  
 }  
 **return** (**int**)rev;  
 }

}

*/\*\*Given a non-negative number represented as an array of digits, plus one to the number. The digits are stored such that the most significant digit is at the head of the list. 大数加1*

PlusOne

**public class** PlusOne {**public int**[] plusOne(**int**[] digits) {  
 **int** n = digits.**length**;  
 **for**(**int** i=n-1; i>=0; i--) {  
 **if**(digits[i] < 9) {  
 digits[i]++;  
 **return** digits;  
 }  
 digits[i] = 0;  
 }  
 **int**[] newRes = **new int** [n+1];  
 newRes[0] = 1;  
 **return** newRes;  
 }

}

AddBinary 双指针 类AddTwoNumbers

**public class** AddBinary {  
 *//简洁代码, Computation from string usually can be simplified by using a carry as such 从低位开始进位相加* **public** String addBinary(String a, String b) {  
 StringBuilder sb = **new** StringBuilder();  
 **int** i = a.length()-1, j = b.length()-1, carry = 0;  
 **while** (i>=0 || j>=0) {  
 **int** sum = carry;  
 **if** (i>=0) sum += a.charAt(i--)-**'0'**;  
 **if** (j>=0) sum += b.charAt(j--)-**'0'**;  
 sb.append(sum % 2);  
 carry = sum / 2;  
 }  
 **if** (carry!=0) sb.append(carry);  
 **return** sb.reverse().toString();  
 }  
}

*/\*\*（注意大整数）  
 \* You are given two linked lists representing two non-negative numbers.  
 \* 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.  
 Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)  
 Output: 7 -> 0 -> 8*

AddTwoNumbers 双指针 链表 类AddBinary

**public class** AddTwoNumbers {  
 **public** ListNode addTwoNumbers(ListNode l1, ListNode l2) {  
 ListNode dummy = **new** ListNode(0), p = dummy;  
 **int** carry = 0;  
 **while**(l1!=**null** || l2!=**null**){  
 **if** (l1!=**null**){  
 carry += l1.**val**;  
 l1 = l1.**next**;  
 }  
 **if** (l2!=**null**){  
 carry += l2.**val**;  
 l2 = l2.**next**;  
 }  
 p.**next** = **new** ListNode(carry%10);  
 p = p.**next**; *//****TODO 易漏*** carry = carry/10;  
 }  
 **if** (carry!=0) p.**next** = **new** ListNode(carry); *//****TODO 易漏* return** dummy.**next**;  
 }  
}

MultiplyStrings

**public class** MultiplyStrings { *\* 从个位起,我们把每一位相乘，得到一个没有进位的临时结果，然后把临时结果从低位起依次进位。  
 \* 对于一个m位整数乘以n位整数的结果，最多只有m+n位。* **public** String multiply(String num1, String num2) {  
 **if** (num1.equals(**"0"**) || num2.equals(**"0"**)) **return "0"**; *//****TODO 不能漏* char**[] l = **new** StringBuilder(num1).reverse().toString().toCharArray();  
 **char**[] r = **new** StringBuilder(num2).reverse().toString().toCharArray();  
 **int**[] nums = **new int**[l.**length** + r.**length**];  
 **for** (**int** i=0; i<l.**length**; i++){  
 **for** (**int** j=0; j<r.**length**; j++){  
 nums[i+j] += (l[i]-**'0'**) \* (r[j]-**'0'**);  
 }  
 }  
 StringBuilder builder = **new** StringBuilder();  
 **for** (**int** i=0; i<nums.**length**; i++){  
 builder.append(nums[i] % 10);  
 **if** (i+1<nums.**length**) nums[i+1] += nums[i]/10;  
 }  
 *//****TODO 注意删除最后的0* if** (builder.charAt(nums.**length**-1)==**'0'**) builder.deleteCharAt(nums.**length**-1);  
 **return** builder.reverse().toString();  
 }

}

DivideTwoIntegers 增量减

**public class** DivideTwoIntegers {

// *Divide two integers without using multiplication, division and mod operator* **public int** divide(**int** dividend, **int** divisor) {  
 **if**(divisor==0 || dividend==Integer.***MIN\_VALUE*** && divisor==-1) **return** Integer.***MAX\_VALUE***;  
 **int** res = 0;  
 **int** sign = (dividend<0)^(divisor<0)? -1 : 1;  
 **long** a = Math.*abs*((**long**)dividend), b = Math.*abs*((**long**)divisor);  
 **while**(b<=a){  
 **long** sum = b, multiple = 1;  
 **while**(sum<<1 <= a){  
 sum <<= 1;  
 multiple <<= 1;*//moving with stride 1, 2, 4, 8, 16...2^n for performance reason.* }  
 a -= sum;  
 res += multiple;  
 }  
 **return** sign==1? res : -res;  
 }

}

*/\*\*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 kth permutation sequence.  
 Note: Given n will be between 1 and 9 inclusive.*

PermutationSequence Math

**public class** PermutationSequence {  
 */\*\* 发现数学规律进行解题：  
 \* 1. 以某一数字开头的排列有(n-1)!个，先计算(n-1)!；k=k-1将k转换为从0开始（原来k从1开始）； 2. 第一位数字是“原字符串”位置k/(n-1)!处的字符；更新k=k%(n-1)!；  
 3. 第二位数字是“原字符串”位置k/(n-2)!处的字符；更新k=k%(n-2)!；  
 4. 不断重复2、3，直至第n位数字。（注意：每确定一位数字，“原字符串”应移除该数字）  
 时间复杂度：O(n^2)；空间复杂度：O(n)；* **public** String getPermutation(**int** n, **int** k) {  
 StringBuilder sb = **new** StringBuilder();  
 List<Integer> origin = **new** ArrayList<>();*//生成原字符串* **for**(**int** i = 1; i <= n; i++) origin.add(i);  
 **int** factorial = 1;*//计算(n-1)!* **for** (**int** i = 1; i < n; i++) factorial \*= i;  
 k = k - 1;*//k转换为从0开始* **for** (**int** i = 0; i < n; i++) {*//计算Kth全排列串中第i位(0<i<n)数字* **int** index = k / factorial; *//得到第i位数字字符* k = k % factorial; *//更新k* sb.append(origin.remove(index));*//每确定一位数字，“原字符串”应移除该数字* **if** (i < n - 1) { *//更新factorial* factorial = factorial / (n - 1 - i);  
 }  
 }  
 **return** sb.toString();  
 }

}

**2Sum/3Sum/4Sum \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*2Sum 哈希*

**public int**[] twoSum(**int**[] nums, **int** target) {  
 HashMap<Integer, Integer> map = **new** HashMap<>();  
 **for**(**int** i=0; i<nums.**length**; i++){  
 **if**(!map.containsKey(nums[i])) map.put(target-nums[i], i);  
 **else return new int**[]{map.get(nums[i]), i};  
 }  
 **return new int**[2];  
}

*3Sum 类二分(夹逼) 排序 去重*

*\* O(n^2) 先快排，再枚举第一个数，然后对余下数的求2sum(由于数组有序,采用二分法,同时注意去重)***public** List<List<Integer>> threeSum(**int**[] num) {  
 Arrays.*sort*(num);  
 List<List<Integer>> res = **new** LinkedList<>();  
 **for** (**int** i = 0; i < num.**length**-2; i++) {  
 **if** (i==0 || num[i]!=num[i-1]) { *// Skip same results* **int** l = i+1, r = num.**length**-1, sum = 0-num[i];  
 **while** (l<r) {  
 **if** (num[l]+num[r]==sum) {  
 res.add(Arrays.*asList*(num[i], num[l], num[r]));  
 **while** (l<r && num[l]==num[l+1]) l++; *// Skip same results* **while** (l<r && num[r]==num[r-1]) r--; *// Skip same results* l++; r--;  
 } **else if** (num[l]+num[r]<sum) l++;  
 **else** r--;  
 }  
 }  
 }  
 **return** res;  
}

*3SumClosest 类二分(夹逼) 排序 去重*

**public int** threeSumClosest(**int**[] nums, **int** target) {  
 Arrays.*sort*(nums);  
 **int** closestDiff = Integer.***MAX\_VALUE***;  
 **int** result = 0;  
 **for** (**int** i = 0; i < nums.**length** - 2; i++){  
 **if** (i > 0 && nums[i] == nums[i - 1]) **continue**; *//去重* **int** l = i + 1, r = nums.**length** - 1;  
 **while** (l < r){  
 **int** sum = nums[l] + nums[r] + nums[i];  
 **if** (sum == target) **return** sum;  
 **else if** (sum > target) r--;  
 **else** l++;  
 **int** diff = Math.*abs*(sum - target);  
 **if** (diff < closestDiff){  
 result = sum;  
 closestDiff = diff;  
 }  
 }  
 }  
 **return** result;  
}

*4Sum 类二分(夹逼) 排序 去重*

**public** List<List<Integer>> fourSum(**int**[] nums, **int** target) {  
 ArrayList<List<Integer>> res = **new** ArrayList<>();  
 **if**(nums.**length**<4) **return** res;  
 Arrays.*sort*(nums);  
 **for**(**int** i=0; i<nums.**length**-3; i++){  
 **if**(i>0 && nums[i]==nums[i-1]) **continue**; *// Skip same results* **for**(**int** j=i+1; j<nums.**length**-2; j++){  
 **if**(j>i+1 && nums[j]==nums[j-1]) **continue**; *// Skip same results* **int** l = j+1, r = nums.**length**-1;  
 **while**(l<r){  
 **int** sum = nums[i]+nums[j]+nums[l]+nums[r];  
 **if**(sum==target){  
 res.add(Arrays.*asList*(nums[i], nums[j], nums[l], nums[r]));  
 **while**(l<r && nums[l]==nums[l+1]) l++; *// Skip same results* **while**(l<r && nums[r]==nums[r-1]) r--; *// Skip same results* l++; r--;  
 } **else if**(sum<target)l++;  
 **else** r--;  
 }  
 }  
 }  
 **return** res;  
}

**Power Ugly \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Sqrtx 二分法 二分返回值为r

**public class** Sqrtx { *\* 二分搜索：对于一个非负数n，它的平方根不会大于（n/2+1）。  
 \* 在[0, n/2+1]这个范围内可以进行二分搜索，求出n的平方根。* **public int** mySqrt(**int** x) {  
 **int** l = 0, r = x/2 + 1; *//r=x也可以* **while** (l<=r){  
 **int** m = l+(r-l)/2;  
 **long** square = (**long**)m\*m; *//****TODO 强制转化为long非常重要,否则溢出* if** (square==x) **return** m;  
 **else if** (square<x) l = m+1;  
 **else** r = m-1;  
 }  
 **return** r;  
 }  
}

*/\*\*  
 \* Implement myPow(x, n).  
 \* Created by DCLab on 12/3/2015.*

Pow 二分法递归

**public class** Pow {  
 *//二分法（递归）* **public double** myPow(**double** x, **int** n) {  
 **if** (n < 0) **return** 1/power(x, -n); *//****TODO 注意这种写法防溢出* else return** power(x, n);  
 }  
 **private double** power(**double** x, **int** n) {  
 **if** (n==0) **return** 1;  
 **double** v = power(x, n/2);  
 **return** n%2==0? v\*v : x\*v\*v;//注意n%2不要写成n/2  
 }

}

isPowerOfTwo

**public boolean** isPowerOfTwo(**int** n) {  
 **return** n>0 && (n&(n-1))==0;  
}

isPowerOfThree

**public boolean** isPowerOfThree(**int** n) {*//Math*  
 **return** ( n>0 && 1162261467%n==0); *// 1162261467 is 3^19, 3^20 is bigger than int*}**public boolean** isPowerOfThree1(**int** n) {*//简单易懂*  
 **if**(n>1) **while**(n%3==0) n /= 3;  
 **return** n==1;  
}

isPowerOfFour

**public boolean** isPowerOfFour(**int** num) {  
 *//****TODO return num>0 && 1073741824%num==0 无法排除2的情况* return** num>0 && (num&(num-1))==0 && (num-1)%3==0; *//还要检测2的幂*}  
**public boolean** isPowerOfFour1(**int** num) {  
 **if** (num>1) **while** (num%4==0) num /= 4;  
 **return** num==1;  
}

*/\*\*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.  
 \* Created by eugene on 16/5/14.*

UglyNumber

**public class** UglyNumber { *\* Just divide by 2, 3 and 5 as often as possible and then check whether we arrived at 1. Also try divisor 4 if that makes the code simpler / less repetitive.* **public boolean** isUgly1(**int** num) {  
 **if** (num<=0) **return false**;  
 **for** (**int** i=2; i<6; i++){ *//2,3,4,5* **while** (num%i==0) num /= i;  
 }  
 **return** num==1;  
 }

*}*

*/\*\*Write a program to find the n-th ugly number.  
 Ugly numbers are positive numbers whose prime factors only include 2, 3, 5.  
 For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers. Note that 1 is typically treated as an ugly number.  
 \* Created by eugene on 16/5/14.*

UglyNumberII DP多重

**public class** UglyNumberII {  
 */\*\*https://leetcode.com/discuss/52905/my-16ms-c-dp-solution-with-short-explanation  
 \* DP: We have an array k of first n ugly number. We only know, at the beginning, the first one, which is 1. Then  
 k[1] = min( k[0]x2, k[0]x3, k[0]x5). The answer is k[0]x2. So we move 2's pointer to 1.  
 Then we test:  
 k[2] = min( k[1]x2, k[0]x3, k[0]x5). And so on.  
 Be careful about the cases such as 6, in which we need to forward both pointers of 2 and 3.* **public int** nthUglyNumber(**int** n) {  
 **int**[] primes = {2,3,5};  
 **int**[] idx = **new int**[3];  
 **int**[] uglys = **new int**[n];  
 uglys[0] = 1;  
 **for** (**int** i=1; i<n; i++) {  
 uglys[i] = Integer.***MAX\_VALUE***;  
 **for** (**int** j=0; j<3; j++)  
 uglys[i] = Math.*min*(uglys[i], uglys[idx[j]]\*primes[j]);  
 **for** (**int** j=0; j<3; j++) *// bump up index for the current minimum ugly number* **if** (uglys[i]==uglys[idx[j]]\*primes[j]) idx[j]++;  
 }  
 **return** uglys[n-1];  
 }  
}

SuperUglyNumber DP多重

**public class** SuperUglyNumber { *\* 与UglyNumberII思路一致, O(kN)* **public int** nthSuperUglyNumber(**int** n, **int**[] primes) {  
 **int** k = primes.**length**;  
 **int**[] idx = **new int**[k];  
 **int**[] uglys = **new int**[n];  
 uglys[0] = 1;  
 **for** (**int** i=1; i<n; i++){  
 uglys[i] = Integer.***MAX\_VALUE***;  
 **for** (**int** j=0; j<k; j++)  
 uglys[i] = Math.*min*(uglys[i], uglys[idx[j]]\*primes[j]);  
 **for** (**int** j=0; j<k; j++)  
 **if** (uglys[i]==uglys[idx[j]]\*primes[j]) idx[j]++;  
 }  
 **return** uglys[n-1];  
 }  
}

**trap/柱形（从左从右遍历）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*\* Given n non-negative integers representing an elevation map where the width of  
 \* each bar is 1, compute how much water it is able to trap after raining.  
 For example, Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.  
 Ref: https://leetcode.com/problems/trapping-rain-water/  
 \* Created by DCLab on 11/25/2015.*

TrappingRainWater HARD 从左从右

**public class** TrappingRainWater {  
 *\* 思路：对某个值A[i]来说，能trapped的最多的water取决于在i之前最高的值leftMostHeight[i] 和在i右边的最高的值rightMostHeight[i]（均不包含自身）。  
 如果min(left,right) > A[i]，那么在i这个位置上能trapped的water就是min(left,right) – A[i]。 \* 第一遍从左到右计算数组leftMostHeight，第二遍从右到左计算rightMostHeight，  
 在第二遍的同时就可以计算出i位置的结果了，而且并不需要开空间来存放rightMostHeight数组。  
 \* 时间复杂度是O(n)，只扫了两遍。空间O(n).***public int** trap1(**int**[] height) {  
 **if** (height == **null** || height.**length** <= 2) **return** 0;  
 **int**[] lMax = **new int**[height.**length**];  
 **int** len = height.**length**, max = height[0], trapped = 0;  
 **for** (**int** i = 0; i<len; i++){ *//left -> right* lMax[i] = max;  
 **max = Math.max(max, height[i]);**  
 }  
 max = height[len-1];  
 **for** (**int** i = len-1; i>=0; i--){ *//right -> left* **if** (Math.*min*(lMax[i], max) > height[i]){  
 trapped += Math.*min*(lMax[i], max) - height[i];  
 }  
 **max = Math.max(max, height[i]);**  
 }  
 **return** trapped;  
}  
}

*/\*\*There are N children standing in a line. Each child is assigned a rating value.  
 You are giving candies to these children subjected to the following requirements:  
 Each child must have at least one candy. Children with a higher rating get more candies than their neighbors. What is the minimum candies you must give?*

Candy HARD 从左从右 贪心

**public class** Candy {  
 */\*\*涉及相邻元素问题,类似Trapping Rain Water,总体思路也是从左从右双向遍历.  
 \* 贪心:只要遇到更大的相邻值,则可以加一;但要总数最小,则需按升序来加一.  
 首先从左向右遍历,升序加一;再从右向左遍历,升序加一.* **public int** candy(**int**[] ratings) {  
 **int**[] candies = **new int**[ratings.**length**];  
 candies[0] = 1;  
 **int** total = candies[0];  
 **for** (**int** i=1; i<ratings.**length**; i++){  
 **if** (ratings[i-1]<ratings[i]){  
 candies[i] = candies[i-1]+1;  
 } **else** {  
 candies[i] = 1;  
 }  
 total += candies[i];  
 }  
 **for** (**int** i=ratings.**length**-2; i>=0; i--){  
 **if** (ratings[i]>ratings[i+1]){  
 **if** (! (candies[i]>candies[i+1])) {  
 **int** old = candies[i]; *//****TODO 易错(用例[5,3,1]),并非新增值都是1,应存储原值,用以计算新增值*** candies[i] = candies[i+1]+1;  
 total += candies[i]-old;  
 }  
 }  
 }  
 **return** total;  
 }  
}

*/\*\*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.  
 \* Created by eugene on 16/6/1.*

ContainerWithMostWater 类二分 无序

**public class** ContainerWithMostWater {  
 *//规律证明参见:https://leetcode.com/discuss/59635/easy-concise-java-o-n-solution-with-proof-and-explanation //类似Trapping Rain Water的双指针解法* **public int** maxArea(**int**[] height) {  
 **if** (height==**null** || height.**length**<=1) **return** 0;  
 **int** l = 0, r = height.**length**-1;  
 **int** maxArea = 0;  
 **while** (l<r){  
 maxArea = Math.*max*(maxArea, Math.*min*(height[l], height[r])\*(r-l));  
 *//核心方法:Every time move the smaller value pointer to inner array* **if** (height[l]<height[r]) l++;  
 **else** r--;  
 }  
 **return** maxArea;  
 }

}

*/\*\*Given an array of n integers where n > 1, nums, return an array output such that output[i]  
 is equal to the product of all the elements of nums except nums[i].  
 Solve it without division and in O(n).  
 For example, given [1,2,3,4], return [24,12,8,6].  
 Follow up: Could you solve it with constant space complexity?  
 (Note: The output array does not count as extra space for the purpose of space complexity analysis.)  
 \* Created by eugene on 16/5/2.*

ProductOfArrayExceptSelf

**public class** ProductOfArrayExceptSelf {  
 *\* 先从左往右求积,再从右往左求积.* **public int**[] productExceptSelf(**int**[] nums) {  
 **int** len = nums.**length**;  
 **int**[] res = **new int**[len];  
 res[0] = 1;  
 **for** (**int** i=1; i<len; i++) {  
 res[i] = res[i-1] \* nums[i-1];  
 }  
 **int** right = 1;  
 **for** (**int** i = len-1; i>=0; i--) {  
 res[i] \*= right;  
 right \*= nums[i];  
 }  
 **return** res;  
 }  
}

*/\*\*Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, \* find the area of largest rectangle in the histogram.  
 \* For example, Given height = [2,1,5,6,2,3], return 10.  
 \* Created by DCLab on 1/5/2016.*

LargestRectangleInHistogram HARD 栈

**public class** LargestRectangleInHistogram { *\* 方法1：维护一个栈，栈内只存高度递增的条形索引。这个动态栈的特点是：栈内元素一定比当前i指向的元素小。 \* 另一个关键点是弹栈过程中面积的计算（高\*宽）：h[t] \* (stack.isEmpty() ? i : i - stack.peek() - 1)。 \* 时间复杂度：O(n)。  
 \* 方法2：对当前条形，计算其前面所有情况的面积。时间复杂度：O(n\*n)。* **public int** largestRectangleArea(**int**[] height) {  
 Stack<Integer> stack = **new** Stack<>();  
 **int** maxArea = 0, i = 0;  
 **int**[] h = Arrays.*copyOf*(height, height.**length** + 1); *//末尾添加dummy元素0* **while**(i < h.**length**){  
 **if**(stack.isEmpty() || h[stack.peek()] <= h[i]){ *//维护高度递增条形索引栈* stack.push(i++);//TODO 注意对比的不是h[i-1]<=h[i]，注意stack存索引  
 }**else** { *//当前元素小于栈顶元素* **int** p = stack.pop();  
 *//高h[p]，宽stack.isEmpty() ? i : i - stack.peek() - 1* maxArea = Math.*max*(maxArea, h[p] \* (stack.isEmpty() ? i : i - stack.peek() - 1));//TODO 注意stack为空时宽度为i  
 }  
 }  
 **return** maxArea;  
 }  
}

*/\*\*  
 \* Given a 2D binary matrix filled with 0's and 1's,  
 \* find the largest rectangle containing all ones and return its area.  
 \* Created by DCLab on 1/6/2016.*

MaximalRectangle HARD

**public class** MaximalRectangle {  
 */\*\*DP方法也行,但相对复杂难以理解  
 \* 高效的方法：转化为Largest Rectangle in Histogram的问题，时间复杂为O(n^2).* **public int** maximalRectangle(**char**[][] matrix) {  
 **if** (matrix.**length**==0) **return** 0;  
 **int** m = matrix.**length**, n = matrix[0].**length**;  
 *//实际上height可以分配一维数组存储* **int**[][] height = **new int**[m][n+1]; *//末尾多加一个dummy元素0* **int** maxArea = 0;  
 **for** (**int** i=0; i<m; i++) {*//矩阵按行转化为柱状图* **for** (**int** j=0; j<n; j++) {  
 **if** (matrix[i][j] == **'0'**) height[i][j] = 0;  
 **else** height[i][j] = (i==0)? 1 : height[i-1][j]+1;  
 }  
 }  
 **for** (**int** i=0; i<m; i++) maxArea = Math.max(maxArea, maxAreaInHist(h[i]));  
 **return** maxArea;  
 }  
 *//O(n) Largest Rectangle in Histogram算法，传入的height数组最后一个元素是多加的dummy元素* **private int** maxAreaInHist(**int**[] height) {  
 Stack<Integer> stack = **new** Stack<>();  
 **int** i = 0;  
 **int** maxArea = 0;  
 **while** (i < height.**length**) {  
 **if** (stack.isEmpty() || height[stack.peek()] <= height[i]) {  
 stack.push(i++);  
 } **else** {  
 **int** t = stack.pop();  
 maxArea = Math.*max*(maxArea, height[t] \* (stack.isEmpty() ? i : i - stack.peek() - 1));  
 }  
 }  
 **return** maxArea;  
 }  
}

**堆\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*Given an array nums, there is a sliding window of size k which is moving from the very left of  
 the array to the very right. You can only see the k numbers in the window.  
 Each time the sliding window moves right by one position.  
 For example, Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.  
 Window position Max  
 --------------- -----  
 [1 3 -1] -3 5 3 6 7 3  
 1 [3 -1 -3] 5 3 6 7 3  
 1 3 [-1 -3 5] 3 6 7 5  
 1 3 -1 [-3 5 3] 6 7 5  
 1 3 -1 -3 [5 3 6] 7 6  
 1 3 -1 -3 5 [3 6 7] 7  
 Therefore, return the max sliding window as [3,3,5,5,6,7].  
 Note: You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.  
 Follow up: Could you solve it in linear time?  
 \* Created by eugene on 16/5/3.*

SlidingWindowMaximum HARD 含双端队列法

**public class** SlidingWindowMaximum {  
 */\*\*https://segmentfault.com/a/1190000003903509  
 \* 双端队列维护窗口(存下标). 时间O(N) 空间O(K)* **public int**[] maxSlidingWindow(**int**[] nums, **int** k) {  
 **if**(nums == **null** || nums.**length** == 0) **return new int**[0];  
 **int** len = nums.**length**;  
 **int**[] res = **new int**[len-k+1];  
 ArrayDeque<Integer> deque = **new** ArrayDeque<>();  
 **for**(**int** i = 0; i<len; i++){  
 *// 每当新数进来时，如果发现队列头部的数的下标，是窗口最左边数的下标，则扔掉* **if**(!deque.isEmpty() && deque.peekFirst() == i-k) deque.poll();  
 *// 把队列尾部所有比新数小的都扔掉，保证队列是降序的* **while**(!deque.isEmpty() && nums[deque.peekLast()]<nums[i]) deque.removeLast();  
 *// 加入新数* deque.offerLast(i);  
 *// 队列头部就是该窗口内第一大的* **if**(i+1>=k) res[i-k+1] = nums[deque.peek()];  
 }  
 **return** res;  
 }  
 *//优先队列(堆)维护窗口. 时间O(NlogK) 空间O(K)* **public int**[] maxSlidingWindow1(**int**[] nums, **int** k) {  
 **int** len = nums.**length**;  
 **if** (len==0 || k==0) **return new int**[0];  
 **int**[] result = **new int**[len-k+1];  
 PriorityQueue<Integer> queue = **new** PriorityQueue<>(Collections.*reverseOrder*());  
 **for** (**int** i=0; i<k; i++){  
 queue.offer(nums[i]);  
 }  
 result[0] = queue.peek();  
 **for** (**int** i=k; i<len; i++){  
 queue.remove(nums[i-k]);  
 queue.offer(nums[i]);  
 result[i-k+1] = queue.peek();  
 }  
 **return** result;  
 }  
}

*/\*\*https://leetcode.com/problems/the-skyline-problem/  
 \* A key point is the left endpoint of a horizontal line segment.  
 \* Created by eugene on 16/4/13.*

TheSkylineProblem HARD

**public class** TheSkylineProblem {  
 */\*\*时间O(NlogN) 空间O(N)  
 \* https://segmentfault.com/a/1190000003786782  
 \* 1 把这些矩形拆成两个点，一个左上顶点，一个右上顶点。将所有顶点按照横坐标排序后，开始遍历这些点。  
 2 遍历时，通过一个大顶堆来得知当前图形的最高位置。堆顶是所有顶点中最高的点，只要这个点没被移出堆，说明这个最高的矩形还没结束。  
 3 对于左顶点，我们将其加入堆中。对于右顶点，我们找出堆中其相应的左顶点，然后移出这个左顶点，同时也意味这这个矩形的结束。  
 4 代码中，为了区分左右顶点，左顶点高度值是负数，而右顶点高度值则存的是正数。  
 注意:堆中先加入一个0高度，帮助我们在只有最矮的建筑物时选择最低值**\*/* **public** List<**int**[]> getSkyline(**int**[][] buildings) {  
 List<**int**[]> result = **new** ArrayList<>();  
 List<**int**[]> height = **new** ArrayList<>();  
 **for**(**int**[] b:buildings) { *// 构建左右顶点混合列表* height.add(**new int**[]{b[0], -b[2]}); *// 左高度为负数* height.add(**new int**[]{b[1], b[2]}); *// 右高度为正数* }  
 Collections.*sort*(height, (a, b) -> { *// 按横坐标排序，相同横坐标其纵坐标小的在前* **if**(a[0] != b[0]) **return** a[0] - b[0];  
 **else return** a[1] - b[1];  
 });  
 Queue<Integer> pq = **new** PriorityQueue<>(11, (i1,i2) -> (i2-i1)); *// 构建大顶堆，按照纵坐标来判断大小* pq.offer(0); *// 将地平线值0先加入堆中* **int** prev = 0; *// prev用于记录上次keypoint的高度* **for**(**int**[] h:height) {  
 **if**(h[1] < 0) pq.offer(-h[1]); *// 将左顶点加入堆中* **else** pq.remove(h[1]); *// 将右顶点对应的左顶点移去* **int** cur = pq.peek();  
 **if**(prev != cur) { *// 如果堆的新顶部和上个keypoint高度不一样，则加入一个新的keypoint* result.add(**new int**[]{h[0], cur});  
 prev = cur;  
 }  
 }  
 **return** result;  
 }  
}

*/\*\*Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.  
 Examples: [2,3,4] , the median is 3  
 [2,3], the median is (2 + 3) / 2 = 2.5  
 Design a data structure that supports the following two operations:  
 void addNum(int num) - Add a integer number from the data stream to the data structure.  
 double findMedian() - Return the median of all elements so far.  
 For example:  
 add(1) add(2) findMedian() -> 1.5  
 add(3) findMedian() -> 2  
 \* Created by eugene on 16/5/29.*

FindMedianFromDataStream HARD 设计

**public class** FindMedianFromDataStream {  
 */\*\*  
 \* 使用两个堆,注意使用long防止溢出.  
 \*/* **class** MedianFinder {  
 PriorityQueue<Long> **small**, **large**;  
 **public** MedianFinder(){  
 **small** = **new** PriorityQueue<>();  
 **large** = **new** PriorityQueue<>();  
 }  
 *// Adds a number into the data structure.* **public void** addNum(**int** num) {  
 **large**.add((**long**)num);  
 **small**.add(-**large**.poll());  
 **if** (**large**.size()<**small**.size()) *//保持large个数总不少于small* **large**.add(-**small**.poll());  
 }  
 *// Returns the median of current data stream* **public double** findMedian() {  
 **return large**.size()>**small**.size()? **large**.peek() :  
 (-**small**.peek()+**large**.peek())/2.0;  
 }  
 }}

**DP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**ClimbingStairs** *Fibonacci标准DP范式*

**public int** climbStairs(**int** n) {*//动态规划：res[i]=res[i-1]+res[i-2]*  
 **if** (n == 1) **return** 1;  
 **int**[] dp = **new int**[n];  
 dp[0] = 1;  
 dp[1] = 2;  
 **for** (**int** i=2; i<n; i++){  
 dp[i] = dp[i-1]+dp[i-2];  
 }  
 **return** dp[n-1];  
}

*/\*\*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.  
 \* Created by DCLab on 1/12/2016.*

DecodeWays 类ClimbingStairs

**public class** DecodeWays {  
 *\* 动态规划（类似Climbing Stairs问题，注意题目的输入是合法的encode，无需考虑不合法编码）： \* 对于当前位i(number数组头部额外分配一个)，考察前两位i-1, i-2：  
 递推式number[i] = number[i-1] + number[i-2]，但有以下限制：****TODO 注意限制条件的思维方式，如何做到简明不易错*** *1. s[i-1]!=0，否则number[i]=number[i-2]；  
 2. 在s[i-1]!=0基础上，s[i-2]!=0 && (1<=s[i-2,i-1]<=26)，否则number[i]=number[i-1]。* **public int** numDecodings(String s) {  
 **if**(s == **null** || s.length() == 0) **return** 0;  
 **int** n = s.length();  
 **int**[] dp = **new int**[n+1];  
 dp[0] = 1; *//dp[0] means an empty string will have one way to decode* dp[1] = s.charAt(0)!=**'0'**? 1 : 0;  
 **for**(**int** i=2; i<=n; i++) {  
 **int** first = Integer.*valueOf*(s.substring(i-1, i));  
 **int** second = Integer.*valueOf*(s.substring(i-2, i));  
 **if**(1<=first && first<=9) dp[i] += dp[i-1];  
 **if**(10<=second && second<=26) dp[i] += dp[i-2];  
 }  
 **return** dp[n];  
 }

*}*

*/\*\*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.*

Triangle 三角->1D 自底向上

**public class** Triangle { *\* 二维动态规划(自顶向下):在某一个元素i，j的最小路径和就是它上层对应的相邻两个元素的最小路径和加上自己的值，  
 动态规划(自底向上):sum[i][j]=min(sum[i+1][j],sum[i+1][j+1])+triangle[i][j],  
 自底向上的方式省去了对每层首尾元素的特殊处理,更简洁.* **public int** minimumTotal(List<List<Integer>> triangle) {  
 **int** m = triangle.size();  
 **if** (m==0) **return** 0;  
 **int**[] dp = **new int**[m];  
 **for** (**int** i=0; i<triangle.get(m-1).size(); i++)  
 dp[i] = triangle.get(m-1).get(i);  
 **for** (**int** i=m-2; i>=0; i--){  
 **for** (**int** j=0; j<triangle.get(i).size(); j++)  
 dp[j] = Math.*min*(dp[j], dp[j+1]) + triangle.get(i).get(j);  
 }  
 **return** dp[0];  
 }  
}

*/\*\*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?  
 Note: m and n will be at most 100.*

UniquePaths 2D->1D缩减范式

**public class** UniquePaths { //另一种方法是Math 求组合数 *\* 动态规划：res[i][j]=res[i-1][j]+res[i][j-1]   
 \* 为简化存储，只需要用一个一维数组存上一行的信息即可。* **public int** uniquePaths(**int** m, **int** n) {  
 **int**[] res = **new int**[n]; *//存上一行的信息* res[0] = 1;  
 **for**(**int** i=0; i<m; i++) *//m-1+1，多出一行存初始值* **for**(**int** j=1; j<n; j++) *//n-1* res[j] += res[j-1]; *//更新新一行的信息* **return** res[n-1];  
 }

*}*

UniquePathsII 2D->1D缩减

**public class** UniquePathsII { //含障碍情况  
 **public int** uniquePathsWithObstacles(**int**[][] obstacleGrid) {  
 **int** n = obstacleGrid[0].**length**;  
 **int**[] dp = **new int**[n];  
 dp[0] = 1;  
 **for** (**int**[] row : obstacleGrid) {  
 **for** (**int** j = 0; j < n; j++) {  
 **if** (row[j]==1) dp[j] = 0;  
 **else if** (j>0) dp[j] += dp[j-1];  
 }  
 }  
 **return** dp[n-1];  
 }

*}*

*/\*\*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.  
 \* Created by eugene on 16/3/28.*

HouseRobber

**public class** HouseRobber {  
 *//DP, dp(i)=max{dp(i-1), dp(i-2)+nums(i)}* **public int** rob(**int**[] nums) {  
 **if** (nums.**length**==0) **return** 0;  
 **if** (nums.**length**==1) **return** nums[0];  
 **int**[] dp = **new int**[nums.**length**];  
 dp[0] = nums[0];  
 dp[1] = Math.*max*(dp[0], nums[1]);  
 **for** (**int** i=2; i<nums.**length**; i++){  
 dp[i] = Math.*max*(dp[i-1], dp[i-2]+nums[i]);  
 }  
 **return** dp[nums.**length**-1];  
 }  
}

HouseRobberII

**public class** HouseRobberII { //回环情形  
 *//DP, s(i)=max{s(i-1), s(i-2)+nums(i)}  
 //分两种情况,遍历数组两次,1.包含第一个,不含最后一个;2.包含最后一个,不含第一个.  
 //http://www.programcreek.com/2014/05/leetcode-house-robber-ii-java/* **public int** rob(**int**[] nums) {  
 **int** len = nums.**length**;  
 **if** (len==0) **return** 0;  
 **if** (len==1) **return** nums[0];  
 **if** (len==2) **return** Math.*max*(nums[0], nums[1]);  
 **int**[] dp = **new int**[len];  
 *//剔除最后一个元素* dp[0] = nums[0];  
 dp[1] = Math.*max*(dp[0], nums[1]); *//****TODO 易错* for** (**int** i=2; i<len-1; i++){  
 dp[i] = Math.*max*(dp[i-1], dp[i-2]+nums[i]);  
 }  
 **int** temp = dp[len-2];  
 *//剔除第一个元素* dp[1] = nums[1];  
 dp[2] = Math.*max*(dp[1], nums[2]); *//****TODO 易错* for** (**int** i=3; i<len; i++){  
 dp[i] = Math.*max*(dp[i-1], dp[i-2]+nums[i]);  
 }  
 **return** Math.*max*(temp, dp[len-1]);  
 }  
}

*/\*\*Find the contiguous subarray within an array (containing at least one number)  
 \* which has the largest sum.  
 For example, given the array [−2,1,−3,4,−1,2,1,−5,4],  
 the contiguous subarray [4,−1,2,1] has the largest sum = 6.*

MaximumSubarray 局部全局

**public class** MaximumSubarray {  
 **public int** maxSubArray(**int**[] nums) {  
 **if** (nums == **null** || nums.**length** == 0) **return** 0;  
 **int** local = nums[0], global = nums[0];  
 **for** (**int** k = 1; k < nums.**length**; k++){  
 *//****TODO 注意不是Math.max(local+nums[k], local); 若local+nums[k]<nums[k]"应该"直接抛弃k前面的所有数*** local = Math.*max*(local+nums[k], nums[k]);  
 global = Math.*max*(local, global);  
 }  
 **return** global;  
 }  
}

*/\*\*Find the contiguous subarray within an array (containing at least one number) which has the largest product.  
 For example, given the array [2,3,-2,4],  
 the contiguous subarray [2,3] has the largest product = 6.  
 \* Created by eugene on 16/3/11.*

MaximumProductSubarray 局部全局

**public class** MaximumProductSubarray {  
 *// 一维动态规划，考虑到乘积符号问题，同时记录最大值（正）与最小值（负），  
 // 采用局部和全局变量追踪局部与最终结果。* **public int** maxProduct(**int**[] nums) {  
 **if**(nums==**null** || nums.**length**==0) **return** 0;  
 **int** max = nums[0], min = nums[0], globalM = nums[0];  
 **for**(**int** i=1; i<nums.**length**; i++){  
 **int** t1 = max\*nums[i], t2 = min\*nums[i];  
 max = Math.*max*(nums[i], Math.*max*(t1, t2));  
 min = Math.*min*(nums[i], Math.*min*(t1, t2));  
 globalM = Math.*max*(globalM, max);  
 }  
 **return** globalM;  
 }

}

*/\*\*Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n.  
 For example, given n = 12, return 3 because 12 = 4 + 4 + 4; given n = 13, return 2 because 13 = 4 + 9.  
 \* Created by eugene on 16/5/20.*

PerfectSquares

**public class** PerfectSquares {  
 */\*\*https://leetcode.com/discuss/62526/an-easy-understanding-dp-solution-in-java  
 dp[0] = 0  
 dp[1] = dp[0]+1 = 1  
 dp[2] = dp[1]+1 = 2  
 dp[3] = dp[2]+1 = 3  
 dp[4] = Min{ dp[4-1\*1]+1, dp[4-2\*2]+1 }  
 = Min{ dp[3]+1, dp[0]+1 }  
 = 1  
 dp[5] = Min{ dp[5-1\*1]+1, dp[5-2\*2]+1 }  
 = Min{ dp[4]+1, dp[1]+1 }  
 = 2  
 ...  
 dp[13] = Min{ dp[13-1\*1]+1, dp[13-2\*2]+1, dp[13-3\*3]+1 }  
 = Min{ dp[12]+1, dp[9]+1, dp[4]+1 }  
 = 2  
 \* DP: dp[k] = Min{ dp[k-i\*i]+1 }, n-i\*i>=0 && i>=1* **public int** numSquares(**int** n) {  
 **int**[] dp = **new int**[n+1];  
 dp[0] = 0;  
 **for** (**int** k=1; k<=n; k++){  
 dp[k] = dp[k-1]+1;  
 **int** i = 2;  
 **while** (k-i\*i>=0){  
 **int** temp = dp[k-i\*i]+1;  
 **if** (temp<dp[k]) dp[k] = temp;  
 i++;  
 }  
 }  
 **return** dp[n];  
 }  
}

*/\*\*  
 \* Implement wildcard pattern matching with support for '?' and '\*'.*

*'?' Matches any single character.*

*'\*' Matches any sequence of characters (including the empty sequence).  
 \* Some examples:  
 isMatch("aa","a") → false  
 isMatch("aa","aa") → true  
 isMatch("aaa","aa") → false  
 isMatch("aa", "\*") → true  
 isMatch("aa", "a\*") → true  
 isMatch("ab", "?\*") → true  
 isMatch("aab", "c\*a\*b") → false  
 \* Created by DCLab on 11/27/2015.*

WildcardMatching HARD

**public class** WildcardMatching {  
 */\*\*https://leetcode.com/discuss/54278/my-java-dp-solution-using-2d-table  
 \* DP* ***TODO 理解状态转移 注意初始化*** *dp[i][j] = dp[i-1][j-1] 当s.charAt(i-1)==p.charAt(j-1) || p.charAt(j-1)=='?'  
 dp[i][j] = dp[i-1][j] || dp[i][j-1] 当p.charAt(j-1)=='\*'* **public boolean** isMatch(String s, String p) {  
 **int** m = s.length(), n = p.length();  
 **boolean**[][] dp = **new boolean**[m+1][n+1];  
 dp[0][0] = **true**;  
 **for**(**int** j=1; j<=n; j++) {  
 **if**(p.charAt(j-1)==**'\*'**) dp[0][j] = **true**;  
 **else break**;  
 }  
 **for**(**int** i=1; i<=m; i++) {  
 **for**(**int** j=1; j<=n; j++) {  
 **if** (p.charAt(j-1)!=**'\*'**) {  
 **if** (s.charAt(i-1)==p.charAt(j-1) || p.charAt(j-1)==**'?'**)  
 dp[i][j] = dp[i-1][j-1];  
 } **else** {  
 dp[i][j] = dp[i-1][j] || dp[i][j-1];  
 }  
 }  
 }  
 **return** dp[m][n];  
 }

**}**

*/\*\*Implement regular expression matching with support for '.' and '\*'.  
 '.' Matches any single character.  
 '\*' Matches zero or more of the preceding element.  
 The matching should cover the entire input string (not partial).  
 The function prototype should be:  
 bool isMatch(const char \*s, const char \*p)  
 \* Created by eugene on 16/5/31.*

RegularExpressionMatching HARDER

**public class** RegularExpressionMatching {  
 */\*\*****TODO 有一定难度,注意理解记忆*** *\* https://leetcode.com/discuss/93024/easy-dp-java-solution-with-detailed-explanation  
 \* 1. If p[j]==s[i] : dp[i][j] = dp[i-1][j-1];  
 2. If p[j]=='.' : dp[i][j] = dp[i-1][j-1];  
 3. If p[j]=='\*': here are two sub conditions:  
 (1) if p[j-1]!=s[i] : dp[i][j] = dp[i][j-2] //in this case, a\* only counts as empty  
 (2) if p[i-1]==s[i] or p[i-1]=='.':  
 dp[i][j] = dp[i-1][j] //in this case, a\* counts as multiple a  
 || dp[i][j-1] // in this case, a\* counts as single a  
 || dp[i][j-2] // in this case, a\* counts as empty* **public boolean** isMatch(String s, String p) {  
 **if** (s == **null** || p == **null**) **return false**;  
 **boolean**[][] dp = **new boolean**[s.length()+1][p.length()+1];  
 dp[0][0] = **true**;  
 **for** (**int** j=0; j<p.length(); j++) {  
 **if** (p.charAt(j)==**'\*'** && dp[0][j-1]) dp[0][j+1] = **true**;  
 }  
 **for** (**int** i=0 ; i<s.length(); i++) {  
 **for** (**int** j=0; j<p.length(); j++) {  
 **if** (p.charAt(j)==**'.'** || p.charAt(j)==s.charAt(i)) { *//情况1,2* dp[i+1][j+1] = dp[i][j];  
 }  
 **if** (p.charAt(j) == **'\*'**) { *//情况3* **if** (p.charAt(j-1)!=s.charAt(i) && p.charAt(j-1)!=**'.'**) {  
 dp[i+1][j+1] = dp[i+1][j-1];  
 } **else** {  
 dp[i+1][j+1] = dp[i+1][j] || dp[i][j+1] || dp[i+1][j-1];  
 }  
 }  
 }  
 }  
 **return** dp[s.length()][p.length()];  
 }

**}**

*/\*\*Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.  
 For example, given  
 s = "leetcode",  
 dict = ["leet", "code"].  
 Return true because "leetcode" can be segmented as "leet code".  
 \* Created by eugene on 16/2/27.*

WordBreak 1D 回溯会超时

**public class** WordBreak {  
 *\* 看到这题第一反应是DFS枚举查找，直到“探”到string尾部则算成功。但题目并不要求给出是如何break的,而只要判断是否能break。对这类判断“是”与“否”的可以尝试用DP，避免DFS的反复搜索。  
 \* 定义dp[i]为区间[0,i-1]能否被分割,注意初始状态dp[0]=True.  
 dp[i+1]=True时应满足:1.s[k,i]在字典中,0<=k<=i;2.dp[k]==True.* **public boolean** wordBreak(String s, Set<String> wordDict) {  
 **boolean**[] dp = **new boolean**[s.length()+1];  
 dp[0] = **true**;  
 **for** (**int** i=0; i<s.length(); i++){  
 **for** (**int** k=0; k<=i; k++){ *//****TODO 边界极易出错* if** (dp[k] && wordDict.contains(s.substring(k, i+1))) dp[i+1] = **true**;  
 }  
 }  
 **return** dp[s.length()];  
 }  
}

*/\*\*Given a string S and a string T, count the number of distinct subsequences of T in S. Here is an example:S = "rabbbit", T = "rabbit", Return 3.*

DistinctSubsequences HARD

**public class** DistinctSubsequences { *\* 图解:http://blog.csdn.net/abcbc/article/details/8978146  
 \* 二维动态规划(与Interleaving String思路一致):W(i,j)表示S(0,i)的子串与T(0,j)相同的个数, 递推式:W(i,j)=W(i-1,j-1)+W(i-1,j),当S(i)==T(j)时;  
 W(i,j)=W(i-1,j),当S(i)!=T(j)时.  
 边界条件:S,T均为空串,W(0,0)=1;  
 仅S为空串,W(0,j)=0;  
 仅T为空串,W(i,0)=1;* **public int** numDistinct(String s, String t) {  
 **int** lenS = s.length(), lenT = t.length();  
 **int**[][] result = **new int**[lenS+1][lenT+1];  
 result[0][0] = 1;  
 *//for (int j=1; j<=lenT; j++) result[0][j] = 0; //冗余* **for** (**int** i=1; i<=lenS; i++) result[i][0] = 1;  
 **for** (**int** i=1; i<=lenS; i++)  
 **for** (**int** j=1; j<=lenT; j++){  
 result[i][j] = result[i-1][j];  
 **if** (s.charAt(i-1) == t.charAt(j-1)) result[i][j] += result[i-1][j-1];  
 }  
 **return** result[lenS][lenT];  
 }  
}

*/\*\*Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.  
 For example,  
 Given:s1 = "aabcc",s2 = "dbbca",  
 When s3 = "aadbbcbcac", return true.  
 When s3 = "aadbbbaccc", return false.*

InterleavingString HARD

**public class** InterleavingString { *\* 二维动态规划:matched[l1][l2]表示s1取l1长度，s2取l2长度，是否能匹配s3的l1+12长度。  
 递推式:matched[i][j] = s1.charAt(i-1)==s3.char(i+j-1) && matched[i-1][j] ||  
 s2.charAt(j-1)==s3.charAt(i+j-1) && matched[i][j-1];  
 边界条件:l1=0 && l2=0时，matched[0][0] = true;  
 i=0时，s3[j-1]==s2[j-1], matched[0][j] = matched[0][j-1]  
 s3[j-1]!=s2[j-1], matched[0][j] = false;  
 j=0时，s3[i-1]==s1[i-1], matched[i][0] = matched[i-1][0]  
 s3[i-1]!=s1[i-1], matched[i][0] = false;* **public boolean** isInterleave(String s1, String s2, String s3) {  
 **int** len1 = s1.length(), len2 = s2.length(), len3 = s3.length();  
 **if** (len1==0 && len2==0 && len3==0) **return true**;  
 **if** (len1 + len2 != len3) **return false**;  
 **boolean**[][] matched = **new boolean**[len1+1][len2+1];  
 matched[0][0] = **true**; *//二维初始* **for** (**int** j=1; j<=len2; j++){ *//j一维初始* **if** (s2.charAt(j-1) == s3.charAt(j-1)) matched[0][j] = matched[0][j-1];  
 }  
 **for** (**int** i=1; i<=len1; i++){ *//i一维初始* **if** (s1.charAt(i-1) == s3.charAt(i-1)) matched[i][0] = matched[i-1][0];  
 }  
 **for** (**int** i=1; i<=len1; i++){  
 **for** (**int** j=1; j<=len2; j++){  
 matched[i][j] = s1.charAt(i-1)==s3.charAt(i+j-1) && matched[i-1][j] ||  
 s2.charAt(j-1)==s3.charAt(i+j-1) && matched[i][j-1];  
 }  
 }  
 **return** matched[len1][len2];  
 }  
}

*/\*\*Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2.  
 (each operation is counted as 1 step.)  
 You have the following 3 operations permitted on a word:  
 a) Insert a character b) Delete a character c) Replace a character*

EditDistance HARD

**public class** EditDistance {  
 */\*\*https://leetcode.com/discuss/50807/java-dp-solution-o-nm  
 \* f(i, j) := minimum cost (or steps) required to convert first i characters of word1 to first j characters of word2  
 Case 1: word1[i] == word2[j], i.e. the ith the jth character matches.  
 f(i, j) = f(i - 1, j - 1)  
 Case 2: word1[i] != word2[j], then we must either insert, delete or replace, whichever is cheaper  
 f(i, j) = 1 + min { f(i, j - 1), f(i - 1, j), f(i - 1, j - 1) }  
 f(i, j - 1) represents insert operation  
 f(i - 1, j) represents delete operation  
 f(i - 1, j - 1) represents replace operation  
 Here, we consider any operation from word1 to word2. It means, when we say insert operation, we insert a new character after word1 that matches the jth character of word2. So, now have to match i characters of word1 to j - 1 characters of word2. Same goes for other 2 operations as well.  
 Note that the problem is symmetric. The insert operation in one direction (i.e. from word1 to word2)  
 is same as delete operation in other. So, we could choose any direction.  
 Above equations become the recursive definitions for DP.  
 Base Case: f(0, k) = f(k, 0) = k  
 \* D[i,j]: s1的前i个字符和 s2的前j个字符的最短编辑距离  
 \* 注意置换也是一种基本操作，置换明显比 删除+增加 操作更短；  
 \* 如果置换不是基本操作，则是另一种算法（参考：http://blog.csdn.net/abcjennifer/article/details/7735272）。  
 \* s1[i]!=s2[j]时，D[i,j]=min(D[i-1,j]+1,D[i,j-1]+1,D[i-1,j-1]+1);  
 \* s1[i]==s2[j]时，D[i,j]=D[i-1,j-1];* **public int** minDistance(String word1, String word2) {  
 **if** (word1.equals(word2)) **return** 0;  
 **int** m = word1.length(), n = word2.length();  
 **int**[][] dp = **new int**[m+1][n+1];  
 **int** i = 0, j = 0;  
 **for** (i=0; i<=m; i++) dp[i][0] = i;  
 **for** (j=0; j<=n; j++) dp[0][j] = j;  
 **for** (i=1; i<=m; i++)  
 **for** (j=1; j<=n; j++){  
 **if** (word1.charAt(i-1) == word2.charAt(j-1))  
 dp[i][j] = dp[i-1][j-1];  
 **else** {  
 **int** replace = dp[i-1][j-1] + 1;  
 **int** insert = dp[i][j-1] + 1;  
 **int** delete = dp[i-1][j] + 1;  
 dp[i][j] = Math.*min*(Math.*min*(replace, insert), delete);  
 }  
 }  
 **return** dp[m][n];  
 }  
}

**回溯DFS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*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] ]*

CombinationSum CombinationSumUnique 重复与不重复 回溯增量i或i+1

**public class** CombinationSumUnique {  
 **public** List<List<Integer>> combinationSum2(**int**[] candidates, **int** target) {  
 Arrays.*sort*(candidates);//TODO 不重复情况要先排序  
 Set<List<Integer>> res = **new** HashSet<>();  
 dfs(candidates, target, res, **new** LinkedList<>(), 0);  
 **return new** LinkedList<>(res);  
 }  
 **private void** dfs(**int**[] candidates, **int** target, Set<List<Integer>> res, List<Integer> sol, **int** start){  
 **if**(target==0){  
 res.add(**new** LinkedList<>(sol));  
 **return**;  
 }  
 **for**(**int** i=start; i<candidates.**length**; i++){  
 **if**(target-candidates[i]>=0){  
 sol.add(candidates[i]);  
 dfs(candidates, target-candidates[i], res, sol, i+1);  
 sol.remove(sol.size()-1);  
 }  
 }  
 }  
}

*/\*\* Given a collection of 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], and [3,2,1].  
 \* Created by DCLab on 11/29/2015.*

Permutations 无重复全排列 回溯增量为start+1

**public class** Permutations {  
 */\*\*****TODO 不含重复*** *\* 回溯法：从集合依次选出每一个元素，作为排列的第一个元素，然后对剩余的元素进行全排列，如此递归处理。 \* 时间复杂度：n! 空间复杂度：（in place置换）  
 \* 以abc为例子：  
 1. a和a交换(固定a), 求后面bc的全排列： abc, acb。 求完后，a和b交换； 得到bac,开始第二轮 2. b和b交换(固定b), 求后面ac的全排列： bac, bca。 求完后，b和c交换； 得到cab,开始第三轮 3. c和c交换(固定c), 求后面ba的全排列： cab, cba.  
 \* 分析图：http://segmentfault.com/a/1190000002710424* **public** List<List<Integer>> permute(**int**[] nums) {  
 List<List<Integer>> res = **new** LinkedList<>();  
 dfs(nums, res, 0);  
 **return** res;  
 }  
 **private void** dfs(**int**[] nums, List<List<Integer>> res, **int** start){  
 **if**(start==nums.**length**){  
 List<Integer> sol = **new** LinkedList<>();  
 **for**(**int** n: nums) sol.add(n);  
 res.add(sol);  
 **return**;  
 }  
 **for**(**int** i=start; i<nums.**length**; i++){  
 swap(nums, start, i);  
 dfs(nums, res, start+1); *//****TODO 注意特殊的地方,不是i+1而是start+1*** swap(nums, start, i);  
 }  
 }//swap操作省略

}

PermutationsII 含重复全排列 先排序 用集合除重复

**public class** PermutationsII { //T***ODO 注意先排序***

// Set<List<Integer>> res = **new** HashSet<>(); *//****TODO 一定要用集合***

//添加 **if** (i>start && nums[i]==nums[i-1]) **continue**; *//跳过再次重复的数字*

}

Subsets 无重复全组合 无任何剪枝 含另一种*复制插入方法*

**public class** Subsets {  
 *//回溯* **public** List<List<Integer>> subsets(**int**[] nums) {  
 List<List<Integer>> result = **new** ArrayList<>();  
 result.add(**new** ArrayList<>());  
 dfs(nums, result, **new** ArrayList<>(), 0);  
 **return** result;  
 }  
 **private void** dfs(**int**[] nums, List<List<Integer>> result, List<Integer> sol, **int** start){  
 **for** (**int** i=start; i<nums.**length**; i++){  
 sol.add(nums[i]);  
 result.add(**new** ArrayList<>(sol));  
 dfs(nums, result, sol, i+1);  
 sol.remove(sol.size()-1);  
 }  
 }  
 *//另一种快速解法 复制插入方法* **public** List<List<Integer>> subsets0(**int**[] nums) {  
 List<List<Integer>> res = **new** ArrayList<>();  
 res.add(**new** ArrayList<>());  
 **for** (**int** num: nums){  
 **int** n = res.size();  
 **for** (**int** i=0; i<n; i++){  
 ArrayList<Integer> sol = **new** ArrayList<>(res.get(i));  
 sol.add(num);  
 res.add(sol);  
 }  
 }  
 **return** res;  
 }

*}*

SubsetsII 含重复全组合 先排序

**public class** SubsetsII { //T***ODO 注意先排序***

//添加 **if** (i>start && nums[i]==nums[i-1]) **continue**; *//跳过再次重复的数字 注意不是i>0*

}

*/\*\*  
 \* 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] ]  
 \* Created by DCLab on 12/29/2015.*

Combinations 限定组合

**public class** Combinations {  
 **public** List<List<Integer>> combine(**int** n, **int** k) {  
 List<List<Integer>> res = **new** LinkedList<>();  
 **if** (n==0 || k==0) **return** res;  
 dfs(n, k, 1, res, **new** LinkedList<>());  
 **return** res;  
 }  
 **private void** dfs(**int** n, **int** k, **int** start, List<List<Integer>> res, List<Integer> sol){  
 **if**(sol.size()==k){  
 res.add(**new** LinkedList<>(sol));  
 **return**;  
 }  
 **for**(**int** i=start; i<=n; i++){  
 sol.add(i);  
 dfs(n, k, i+1, res, sol);  
 sol.remove(sol.size()-1);  
 }  
 }  
}

RestoreIPAddresses 标准回溯 字符合法判断

**public** List<String> restoreIpAddresses(String s) {  
 List<String> result = **new** ArrayList<>();  
 List<String> list = **new** ArrayList<>();  
 dfs(result, list, s, 0);  
 **return** result;  
 }  
 **private void** dfs(List<String> result, List<String> list, String s, **int** start){  
 **if**(list.size()==4){  
 **if**(start != s.length()) **return**;  
 StringBuilder builder = **new** StringBuilder();  
 **for** (**int** i=0; i<3; i++){  
 builder.append(list.get(i)).append(**"."**);  
 }  
 builder.append(list.get(3));  
 result.add(builder.toString());  
 **return**;  
 }  
 **for**(**int** i=start; i<s.length() && i<start+3; i++){  
 String tmp = s.substring(start, i+1);  
 **if**(isValid(tmp)){  
 list.add(tmp);  
 dfs(result, list, s, i + 1);  
 list.remove(list.size()-1);  
 }  
 }  
 }  
 **private boolean** isValid(String s){  
 **if**(s.charAt(0) == **'0'**)  
 **return** s.equals(**"0"**); *// to eliminate cases like "00", "10"* **int** digit = Integer.*valueOf*(s);  
 **return** digit >= 0 && digit <= 255;  
 }  
}

*/\*\*  
 \* 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.  
 \* Created by DCLab on 12/31/2015.*

WordSearch DFS(2D) boolean

**public class** WordSearch {  
 *//回溯法* **public boolean** exist(**char**[][] board, String word) {  
 **for**(**int** i=0; i<board.**length**; i++){  
 **for**(**int** j=0; j<board[0].**length**; j++){  
 **if**(dfs(board, word, i, j, 0)) **return true**;  
 }  
 }  
 **return false**;  
 }  
 **private boolean** dfs(**char**[][] board, String word, **int** i, **int** j, **int** k){  
 **if**(k==word.length()) **return true**;  
 **if**(i<0 || j<0 || i>=board.**length** || j>=board[0].**length**) **return false**;  
 **if**(board[i][j] == word.charAt(k)){  
 **char** temp = board[i][j];  
 board[i][j]=**'#'**; *//mark* **if**(dfs(board, word, i-1, j, k+1)  
 ||dfs(board, word, i+1, j, k+1)  
 ||dfs(board, word, i, j-1, k+1)  
 ||dfs(board, word, i, j+1, k+1))  
 **return true**; //不能直接return dfs()…  
 board[i][j]=temp; //不匹配的需要重置，匹配的不重置  
 }  
 **return false**;  
 }  
}

*/\*\*Given an integer n, return all distinct solutions to the n-queens puzzle.  
 Each solution contains a distinct board configuration of the n-queens' placement,  
 where 'Q' and '.' both indicate a queen and an empty space respectively.  
 \* 每一个的横竖斜都没有其他皇后*

NQueens NQueensII HARD 2D 非典型回溯 回溯增量为start+1

**public class** NQueens { *\* 回溯法 关键：每一行只能有一个皇后，用一维数组存每行皇后所在列。  
 \* 用一维数组则在回溯的时候不需进行remove重置操作，因为回溯正好就回到上一行，可以接着找下一个合法列坐标。 \* 走完最后一行，坐标（row，cols[row]）即皇后坐标。* **public** List<List<String>> solveNQueens(**int** n) {  
 ArrayList<List<String>> res = **new** ArrayList<>();  
 **if**(n<=0) **return** res;  
 dfs(n, res, 0, **new int**[n]);  
 **return** res;  
 }  
 **private void** dfs(**int** n, List<List<String>> res, **int** start, **int**[] cols){  
 **if**(start==n){  
 List<String> sol = **new** ArrayList<>(n);  
 **for**(**int** i=0; i<n; i++){  
 StringBuilder s = **new** StringBuilder();  
 **for**(**int** j=0; j<n; j++){  
 **if**(j==cols[i]) s.append(**"Q"**);  
 **else** s.append(**"."**);  
 }  
 sol.add(s.toString());  
 }  
 res.add(sol);  
 **return**;  
 }  
 **for**(**int** i=0; i<n; i++){//TODO 不是int i=start;  
 cols[start] = i; *//(start,i)* **if**(isColCrossValid(start, cols)) dfs(n, res, start+1, cols);//不是i+1  
 }  
 }*//判断列和对角线,对角线就是两点行差值和列差值是否相同* **private boolean** isColCrossValid(**int** row, **int**[] cols){  
 **for**(**int** r=0; r<row; r++){  
 **if**(cols[row]==cols[r] || Math.*abs*(cols[row]-cols[r]) == Math.*abs*(row-r)) **return false**;  
 }  
 **return true**;  
 }

}

*/\*\*Write a program to solve a Sudoku puzzle by filling the empty cells.  
 Empty cells are indicated by the character '.'.  
 You may assume that there will be only one unique solution.*

SudokuSolver HARD 2D 非典型回溯 小方阵判断

**public class** SudokuSolver {  
 */\*\*回溯法* ***TODO 注意此回溯法的特点*** *\* 和Valid Sudoku这题相比，此题中的输入是合法的。这样用回溯法我们只要检查新加入的值  
 \* 能否在行、列以及小方块里有效即可，没有必要检查整个矩阵。* **public void** solveSudoku(**char**[][] board) {  
 solve(board);  
 }  
 **private boolean** solve(**char**[][] board){  
 **for** (**int** r=0; r<9; ++r) {  
 **for** (**int** c=0; c<9; ++c) {  
 **if** (board[r][c]!=**'.'**) **continue**;  
 **for** (**char** ch=**'1'**; ch<=**'9'**; ch++){  
 board[r][c] = ch;  
 **if** (isValidSudoku(board, r, c)){ *//限制条件* **if** (solve(board)) **return true**;  
 }  
 board[r][c] = **'.'**;  
 }  
 **return false**;//TODO 记得return false;  
 }  
 }  
 **return true**;  
 } *//****TODO 注意如何计算所在3\*3 block的起始行号/列号* private boolean** isValidSudoku(**char**[][] board, **int** x, **int** y) { *//x,y为坐标* **int** r, c;  
 **for** (r = 0; r<9; ++r) { *// Same value in the same c?* **if** ((x!=r) && (board[r][y]==board[x][y])) **return false**;  
 }  
 **for** (c = 0; c<9; ++c) { *// Same value in the same r?* **if** ((y!=c) && (board[x][c]==board[x][y])) **return false**;  
 }  
 **for** (r = (x/3)\*3; r < x/3\*3+3; ++r) { *// Same value in the 3 \* 3 block?* **for** (c = (y/3)\*3; c < y/3\*3+3; ++c) {  
 **if** ((x!=r) && (y!=c) && (board[r][c]==board[x][y])) **return false**;  
 }  
 }  
 **return true**;  
 }  
}

*/\*\*Given a string s and a dictionary of words dict, add spaces in s  
 to construct a sentence where each word is a valid dictionary word.  
 Return all such possible sentences.  
 For example, given  
 s = "catsanddog",  
 dict = ["cat", "cats", "and", "sand", "dog"].  
 A solution is ["cats and dog", "cat sand dog"].  
 \* Created by eugene on 16/2/28.*

WordBreakII HARD DFS 额外剪枝 哈希

**public class** WordBreakII { *\* Using DFS directly will lead to TLE,  
 so used HashMap to save the previous results to prune duplicated branches* **public** List<String> wordBreak(String s, Set<String> wordDict) {  
 **return** dfs(s, wordDict, **new** HashMap<>());  
 }  
 List<String> dfs(String s, Set<String> wordDict, HashMap<String, LinkedList<String>> map) {  
 **if** (map.containsKey(s)) **return** map.get(s);  
 LinkedList<String> res = **new** LinkedList<>();  
 **if** (s.length() == 0) {  
 res.add(**""**);  
 **return** res;  
 }  
 **for** (String word : wordDict) {  
 **if** (s.startsWith(word)) {  
 List<String> sublist = dfs(s.substring(word.length()), wordDict, map);  
 **for** (String sub: sublist)  
 res.add(word + (sub.isEmpty()? **""** : **" "**) + sub);  
 }  
 }  
 map.put(s, res);  
 **return** res;  
 }

}

*/\*\*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 -> []  
 \* Created by eugene on 16/5/21.*

ExpressionAddOperators HARD

**public class** ExpressionAddOperators {  
 */\*\*  
 \* https://leetcode.com/discuss/58614/java-standard-backtrace-ac-solutoin-short-and-clear  
 \* 回溯:注意边界情况1.用long存储计算结果;2.零开头的数字应忽略. 注意乘法优先级,用一个变量暂存相乘的结果.* **public** List<String> addOperators(String num, **int** target) {  
 ArrayList<String> result = **new** ArrayList<>();  
 **if**(num==**null** || num.length()==0) **return** result;  
 dfs(num, target, 0, **""**, result, 0, 0);  
 **return** result;  
 }  
 **private void** dfs(String num, **int** target, **int** start, String path, List<String> result, **long** eval, **long** multed){  
 **if** (start==num.length()){  
 **if** (target==eval) result.add(path);  
 **return**;  
 }  
 **for** (**int** i=start; i<num.length(); i++){  
 **if** (num.charAt(start)==**'0'**){  
 **if** (i!=start) **break**; *//零开头的数字应忽略* }  
 **long** x = Long.*parseLong*(num.substring(start, i+1));  
 **if** (start==0)  
 dfs(num, target, i+1, path+x, result, x, x);  
 **else** {  
 dfs(num, target, i+1, path+**"+"**+x, result, eval+x, x);  
 dfs(num, target, i+1, path+**"-"**+x, result, eval-x, -x);  
 *//减去之前操作的,再加上相乘值* dfs(num, target, i+1, path+**"\*"**+x, result, eval-multed+multed\*x, multed\*x);  
 }  
 }  
 }  
}

**分支限界BFS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*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. https://leetcode.com/problems/letter-combinations-of-a-phone-number/  
 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.*

LetterCombinationsOfPhoneNumber 与Combination的区别

**public class** LetterCombinationsOfPhoneNumber {  
 **static** String[] *mapping* = **new** String[] {**"0"**, **"1"**, **"abc"**, **"def"**, **"ghi"**, **"jkl"**, **"mno"**, **"pqrs"**, **"tuv"**, **"wxyz"**};  
 *//简洁, 广度(队列* **public** List<String> letterCombinations(String digits) {  
 LinkedList<String> q = **new** LinkedList<>();  
 **if** (digits==**null** || digits.length()==0) **return** q;  
 q.add(**""**);  
 **for**(**int** i=0; i<digits.length(); i++){  
 **int** k = Character.*getNumericValue*(digits.charAt(i));  
 **while**(q.peek().length()==i){  
 String old = q.remove();  
 **for**(**char** c : *mapping*[k].toCharArray())  
 q.offer(old+c);  
 }  
 }  
 **return** q;  
 }

*}*

*/\*\*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 intermediate word must exist in the word list  
 For example,  
 Given: beginWord = "hit" endWord = "cog"  
 wordList = ["hot","dot","dog","lot","log"]  
 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.  
 \* Created by eugene on 16/2/14.*

WordLadder 图BFS

**public class** WordLadder {  
 **class** WordNode{  
 String **word**;  
 **int numSteps**;  
 **public** WordNode(String word, **int** numSteps){  
 **this**.**word** = word;  
 **this**.**numSteps** = numSteps;  
 }  
 } *\* 分支限界法(BFS,最短路径),算法引入WordNode数据结构* **public int** ladderLength(String beginWord, String endWord, Set<String> wordList) {  
 Queue<WordNode> queue = **new** LinkedList<>();  
 queue.add(**new** WordNode(beginWord, 1));  
 wordList.add(endWord);  
 **while**(!queue.isEmpty()){  
 WordNode node = queue.remove();  
 String word = node.**word**;  
 **if**(word.equals(endWord)) **return** node.**numSteps**;  
  
 **char**[] chars = word.toCharArray();  
 **for**(**int** i=0; i<chars.**length**; i++){  
 **for**(**char** c=**'a'**; c<=**'z'**; c++){  
 **char** temp = chars[i];  
 **if**(chars[i]!=c) chars[i] = c;  
 String newWord = **new** String(chars);  
 **if**(wordList.contains(newWord)){  
 queue.add(**new** WordNode(newWord, node.**numSteps**+1));  
 wordList.remove(newWord);  
 }  
 chars[i] = temp;  
 }  
 }  
 }  
 **return** 0;  
 }  
}

SurroundedRegions 图BFS(2D) 高出错率

**public class** SurroundedRegions { *\* 边界处的O不能被包围,可以先遍历一次将与边界相连的所有O找出并替换成#,再遍历一次将O换成X,#换回O. 如果采用DFS可能会产生栈溢出,因此用BFS.* **private** Queue<Integer> **queue** = **new** LinkedList<>();  
 **public void** solve(**char**[][] board) {  
 **if** (board == **null** || board.**length** == 0) **return**;  
 **int** m = board.**length**, n = board[0].**length**;  
 **for** (**int** i = 0; i < m; i++) {*// merge O's on left & right boarder* **if** (board[i][0] == **'O'**) bfsO(board, i, 0);  
 **if** (board[i][n - 1] == **'O'**) bfsO(board, i, n - 1);  
 }  
 **for** (**int** j = 0; j < n; j++) {*// merge O's on top & bottom boarder* **if** (board[0][j] == **'O'**) bfsO(board, 0, j);  
 **if** (board[m - 1][j] == **'O'**) bfsO(board, m - 1, j);  
 }  
 **for** (**int** i = 0; i < m; i++) {*// process the board* **for** (**int** j = 0; j < n; j++) {  
 **if** (board[i][j] == **'O'**) board[i][j] = **'X'**;  
 **else if** (board[i][j] == **'#'**) board[i][j] = **'O'**;*//****TODO 用else if*** }  
 }  
 }  
 **private void** bfsO(**char**[][] board, **int** i, **int** j) {  
 **int** n = board[0].**length**;  
 processO(board, i, j);*// fill current first and then its neighbors* **while** (!**queue**.isEmpty()) {  
 **int** k = **queue**.poll();  
 **int** x = k/n;  
 **int** y = k%n;  
 processO(board, x - 1, y);  
 processO(board, x + 1, y);  
 processO(board, x, y - 1);  
 processO(board, x, y + 1);  
 }  
 }  
 **private void** processO(**char**[][] board, **int** i, **int** j) {  
 **int** m = board.**length**, n = board[0].**length**;  
 **if** (i < 0 || i >= m || j < 0 || j >= n) **return**;  
 **if** (board[i][j]!=**'O'**) **return**;;  
 **queue**.offer(i \* n + j);*// add current cell in queue & then process its neighbors in bfsO* board[i][j] = **'#'**;  
 }  
}

**Trie \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*Given a 2D board and a list of words from the dictionary, find all words in the board. Each word must 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 in a word.  
 For example,  
 Given words = ["oath","pea","eat","rain"] and board =  
 [ ['o','a','a','n'],  
 ['e','t','a','e'],  
 ['i','h','k','r'],  
 ['i','f','l','v'] ]  
 Return ["eat","oath"].  
 Note: You may assume that all inputs are consist of lowercase letters a-z.  
 \* Created by eugene on 16/4/8.*

WordSearchII HARD 回溯

**public class** WordSearchII { *\* https://leetcode.com/discuss/77851/java-15ms-easiest-solution-100-00%25  
 \* 若采用跟Word Search一样的思路,将超时.  
 \* 回溯中引入Trie,当前候选字符串不在所有单词的前缀中时,剪枝.* **public** List<String> findWords(**char**[][] board, String[] words) {  
 List<String> res = **new** ArrayList<>();  
 TrieNode root = buildTrie(words);  
 **for** (**int** i = 0; i < board.**length**; i++) {  
 **for** (**int** j = 0; j < board[0].**length**; j++)  
 dfs(board, i, j, root, res);  
 }  
 **return** res;  
 }  
 **public void** dfs(**char**[][] board, **int** i, **int** j, TrieNode p, List<String> res) {  
 **if** (i < 0 || j < 0 || i >= board.**length** || j >= board[0].**length**) **return**;  
 **char** c = board[i][j];  
 **if** (c == **'#'** || p.**children**[c - **'a'**] == **null**) **return**;  
 p = p.**children**[c - **'a'**];  
 **if** (p.**word** != **null**) { *// found one* res.add(p.**word**);  
 p.**word** = **null**; *// de-duplicate* }  
 board[i][j] = **'#'**;  
 dfs(board, i - 1, j, p, res);  
 dfs(board, i, j - 1, p, res);  
 dfs(board, i + 1, j, p, res);  
 dfs(board, i, j + 1, p, res);  
 board[i][j] = c;  
 }  
 **class** TrieNode {  
 TrieNode[] **children** = **new** TrieNode[26];  
 String **word**;  
 }  
 **public** TrieNode buildTrie(String[] words) {  
 TrieNode root = **new** TrieNode();  
 **for** (String w : words) {  
 TrieNode node = root;  
 **for** (**char** c : w.toCharArray()) {  
 **int** i = c - **'a'**;  
 **if** (node.**children**[i] == **null**) node.**children**[i] = **new** TrieNode();  
 node = node.**children**[i];  
 }  
 node.**word** = w;  
 }  
 **return** root;  
 }  
}

**哈希类 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.  
 The Sudoku board could be partially filled, where empty cells are filled with the character '.'.*

ValidSudoku 2D 简洁合并

**public class** ValidSudoku {  
 *//简洁,不易出错* **public boolean** isValidSudoku(**char**[][] board) {  
 **for** (**int** i=0; i<9; i++) {  
 **if** (!partValid(board,i,0,i,8)) **return false**; *//每行* **if** (!partValid(board,0,i,8,i)) **return false**; *//每列* }  
 **for** (**int** i=0;i<3;i++){ *//每3\*3单元* **for**(**int** j=0;j<3;j++){  
 **if** (!partValid(board,i\*3,j\*3,i\*3+2,j\*3+2)) **return false**;  
 }  
 }  
 **return true**;  
 }  
 **private boolean** partValid(**char**[][] board, **int** x1, **int** y1, **int** x2, **int** y2){  
 HashSet set = **new** HashSet();  
 **for** (**int** i=x1; i<=x2; i++){  
 **for** (**int** j=y1; j<=y2; j++){  
 **if** (board[i][j]!=**'.'**) **if**(!set.add(board[i][j])) **return false**;  
 }  
 }  
 **return true**;  
 }

*}*

*/\*\*Given an unsorted array of integers, find the length of  
 the longest consecutive elements sequence.  
 For example, Given [100, 4, 200, 1, 3, 2],  
 The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.  
 Your algorithm should run in O(n) complexity.*

LongestConsecutiveSequence HARD 中间向两边发散 图(DFS)

**public class** LongestConsecutiveSequence { *\* 因为排序至少要O(nlogn),先排序是不行的.这里使用哈希表. 中间向两边发散法* **public int** longestConsecutive(**int**[] nums) {  
 **if** (nums.**length**==0) **return** 0;  
 Set<Integer> set = **new** HashSet<>();  
 **int** longest = 0;  
 **for** (**int** e : nums) set.add(e);  
 **for** (**int** e : nums) {  
 **int** left = e-1, right = e+1, count = 1;  
 **while** (set.contains(left)){  
 count++;  
 set.remove(left);  
 left--;  
 }  
 **while** (set.contains(right)){  
 count++;  
 set.remove(right);  
 right++;  
 }  
 longest = Math.*max*(longest, count);  
 }  
 **return** longest;  
 }  
}

*/\*\*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.  
 \* Created by eugene on 16/5/6.*

ValidAnagram 数组映射

**public class** ValidAnagram {**public boolean** isAnagram(String s, String t) {  
 **int**[] map = **new int**[26];  
 **for** (**int** i=0; i<s.length(); i++) map[s.charAt(i)-**'a'**]++;  
 **for** (**int** i=0; i<t.length(); i++) map[t.charAt(i)-**'a'**]--;  
 **for** (**int** i: map) **if** (i != 0) **return false**;  
 **return true**;  
 }

}

*/\*\*  
 \* Given an array of strings, group anagrams together.  
 For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],  
 Return: [ ["ate", "eat","tea"],  
 ["nat","tan"],  
 ["bat"] ]  
 Note: For the return value, each inner list's elements must follow the lexicographic order.  
 All inputs will be in lower-case.  
 \* Created by DCLab on 12/2/2015.*

GroupAnagrams

**public class** GroupAnagrams {  
 *//易位构词的排序结果是一样的* **public** List<List<String>> groupAnagrams(String[] strs) {  
 Arrays.*sort*(strs);  
 HashMap<String, List<String>> map = **new** HashMap<>();  
 **for**(String str: strs){  
 **char**[] chars = str.toCharArray();  
 Arrays.*sort*(chars);  
 String s = String.*valueOf*(chars);  
 **if**(!map.containsKey(s)) map.put(s, **new** LinkedList<>()); *//****TODO 简化*** map.get(s).add(str);  
 }  
 **return new** LinkedList<>(map.values()); *//****TODO 简化*** }  
}

*/\*\*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:  
 pattern = "abba", str = "dog cat cat dog" should return true.  
 pattern = "abba", str = "dog cat cat fish" should return false.  
 pattern = "aaaa", str = "dog cat cat dog" should return false.  
 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.  
 \* Created by eugene on 16/5/26.*

WordPattern

**public class** WordPattern {  
 *//简洁* **public boolean** wordPattern(String pattern, String str) {  
 String[] strings = str.split(**" "**);  
 **if** (strings.**length** != pattern.length()) **return false**;  
 HashMap index = **new** HashMap(); *//****TODO 没有指定类型* for** (Integer i=0; i<strings.**length**; ++i){ *//****TODO Integer不能写成int* if** (index.put(pattern.charAt(i), i) != index.put(strings[i], i))  
 **return false**;  
 }  
 **return true**;  
 }

}

LRUCache HARD 双向链表 设计

**public class** LRUCache {  
 **class** CacheEntry {  
 **int key**, **value**;  
 CacheEntry **pre**, **next**;  
 **public** CacheEntry(**int** key, **int** value){  
 **this**.**key** = key;  
 **this**.**value** = value;  
 }  
 }  
 **private int capacity**;  
 **private** Map<Integer, CacheEntry> **map** = **new** HashMap<>();  
 **private** CacheEntry **head**, **end**;  
 */\*\*  
 \* http://www.programcreek.com/2013/03/leetcode-lru-cache-java/  
 \* 如果要O(1)的查找复杂度，肯定要用哈希表.  
 \* 如果要O(1)的cache替换复杂度,可以采用链表,保证从头到尾的顺序就是cache从新到旧的顺序.  
 对于任何一个节点，如果被访问了，那么就将该节点移至头部; 如果cache已满，那么就把尾部的删掉，从头部插入新节点。  
 \** ***@param capacity*** *\*/* **public** LRUCache(**int** capacity) {  
 **this**.**capacity** = capacity;  
 }  
 **public int** get(**int** key) {  
 **if**(**map**.containsKey(key)){  
 CacheEntry entry = **map**.get(key);  
 remove(entry);*//对于任何一个节点，如果被访问了，那么就将该节点移至头部* setHead(entry);  
 **return** entry.**value**;  
 }  
 **return** -1;  
 }  
 **public void** set(**int** key, **int** value) {  
 **if** (**map**.containsKey(key)){  
 CacheEntry entry = **map**.get(key);  
 entry.**value** = value;  
 remove(entry);*//对于任何一个节点，如果被访问了，那么就将该节点移至头部* setHead(entry);  
 } **else** {  
 CacheEntry entry = **new** CacheEntry(key, value);  
 **if** (**map**.size()>=**capacity**){  
 **map**.remove(**end**.**key**);*//如果cache已满，那么就把尾部的删掉，从头部插入新节点* remove(**end**); *//****TODO 此操作不能位于map.remove之前,否则end被垃圾回收为null*** }  
 setHead(entry);  
 **map**.put(key, entry);  
 }  
 }  
 **private void** remove(CacheEntry entry){  
 **if**(entry.**pre**!=**null**) entry.**pre**.**next** = entry.**next**;  
 **else head** = entry.**next**;  
 **if**(entry.**next**!=**null**) entry.**next**.**pre** = entry.**pre**;  
 **else end** = entry.**pre**;  
 }  
 **private void** setHead(CacheEntry entry){  
 entry.**next** = **head**;  
 entry.**pre** = **null**;  
 **if**(**head**!=**null**) **head**.**pre** = entry;  
 **head** = entry;  
 **if**(**end**==**null**) **end** = entry;  
 }  
}

MaxPointsOnLine HARD 几何

**public class** MaxPointsOnLine {  
 **class** Point {  
 **int x**;  
 **int y**;  
 Point() { **x** = 0; **y** = 0; }  
 Point(**int** a, **int** b) { **x** = a; **y** = b; }  
 }  
 */\*\*****TODO 注意一个极其容易出错的基础知识*** *\* http://www.programcreek.com/2014/04/leetcode-max-points-on-a-line-java/  
 \* 为每个点计算其与其他点的斜率,斜率相同则在同一直线上.注意额外处理重复点和垂直线的情况.* **public int** maxPoints(Point[] points) {  
 **if**(points.**length**<=2) **return** points.**length**;  
 Map<Double, Integer> kCount = **new** HashMap<>();  
 **int** max = 0;  
 **for** (**int** i=0; i<points.**length**; i++){  
 Point p1 = points[i];  
 **int** dup = 1, vertical = 0;  
 **for** (**int** j=i+1; j<points.**length**; j++){  
 Point p2 = points[j];  
 **if** (p1.**x**==p2.**x**) {  
 **if** (p1.**y**==p2.**y**) dup++;  
 **else** vertical++;  
 } **else** { *//****TODO 错误:double k = (1.0\*(p1.y-p2.y))/(p1.x-p2.x);* double** k = (p1.**y**-p2.**y**)==0? 0.0 : 1.0\*(p1.**y**-p2.**y**)/(p1.**x**-p2.**x**);  
 kCount.put(k, kCount.containsKey(k)? kCount.get(k)+1 : 1);  
 }  
 }  
 **for** (Integer n: kCount.values()) **if** (n+dup>max) max = n+dup;  
 max = Math.*max*(vertical+dup, max);  
 kCount.clear();  
 }  
 **return** max;  
 }  
}

**Duplicate \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

RemoveElement

**public class** RemoveElement {  
 *//与RemoveDuplicatesFromSortedArray* **public int** removeElement(**int** nums[], **int** val) {  
 **int** index = 0;  
 **for**(**int** i=0; i<nums.**length**; i++)  
 **if**(nums[i]!=val) nums[index++] = nums[i];  
 **return** index;  
 }  
 *//快速* **public int** removeElement1(**int**[] nums, **int** val) {  
 **int** i = 0, j = nums.**length**-1;  
 **while**(i<=j){  
 **if**(nums[i]==val) {  
 *swap*(nums, i, j);  
 j--;  
 }**else**{  
 i++;  
 }  
 }  
 **return** j+1;  
 }

}

*/\*\*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 difference between i and j is at most k.  
 \* Created by eugene on 16/4/14.*

ContainsDuplicateII

**public class** ContainsDuplicateII {  
 **public boolean** containsNearbyDuplicate(**int**[] nums, **int** k) {  
 HashMap<Integer, Integer> map = **new** HashMap<>();  
 **for** (**int** i=0; i<nums.**length**; i++){  
 **if** (!map.containsKey(nums[i])) map.put(nums[i], i);  
 **else** {  
 **if** (i-map.get(nums[i])<=k) **return true**;  
 **else** map.put(nums[i], i);  
 };  
 }  
 **return false**;  
 }  
}

*/\*\*Given an array of integers, find out whether there are two distinct indices i and j  
 in the array such that the difference between nums[i] and nums[j] is at most t  
 and the difference between i and j is at most k.  
 \* Created by eugene on 16/4/14.*

ContainsDuplicateIII

**public class** ContainsDuplicateIII {  
 */\*\*  
 \* O(nlog(k)),借助TreeSet维护窗口k  
 \* http://www.programcreek.com/2014/06/leetcode-contains-duplicate-iii-java/* **public boolean** containsNearbyAlmostDuplicate(**int**[] nums, **int** k, **int** t) {  
 **if** (k < 1 || t < 0) **return false**;  
 TreeSet<Integer> set = **new** TreeSet<>();  
 **for** (**int** i=0; i<nums.**length**; i++) {  
 **int** n = nums[i];  
 **if** ((set.floor(n)!=**null** && n-set.floor(n)<=t)  
 || (set.ceiling(n)!=**null** && set.ceiling(n)-n<=t))  
 **return true**;  
 set.add(n);  
 **if** (i>=k) set.remove(nums[i-k]);  
 }  
 **return false**;  
 }

}

RemoveDuplicatesFromSortedArray

**public class** RemoveDuplicatesFromSortedArray {  
 **public int** removeDuplicates(**int**[] nums) {  
 **int** i = 0;  
 **for**(**int** n : nums)  
 **if**(i<1 || n!=nums[i-1])  
 nums[i++] = n;  
 **return** i;  
 }

}

*/\*\*  
 \* Follow up for "Remove Duplicates":  
 What if duplicates are allowed at most twice?  
 Note: Do not allocate extra space for another array, you must do this in place with constant memory. 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.  
 \* Created by DCLab on 1/1/2016.*

RemoveDuplicatesFromSortedArrayII

**public class** RemoveDuplicatesFromSortedArrayII {**public int** removeDuplicates(**int**[] nums) {  
 **int** i = 0;  
 **for** (**int** n: nums)  
 **if** (i<2 || n!=nums[i-2])  
 nums[i++] = n;  
 **return** i;  
 }

}

*/\*\*  
 \* Given a sorted linked list, delete all duplicates such that each element appear only once. For example,  
 Given 1->1->2, return 1->2.  
 Given 1->1->2->3->3, return 1->2->3.*

RemoveDuplicatesFromSortedList 含递归法

**public class** RemoveDuplicatesFromSortedList {  
 *//简洁 递归 https://leetcode.com/discuss/37323/3-line-java-recursive-solution* **public** ListNode deleteDuplicates(ListNode head) {  
 **if**(head == **null** || head.**next** == **null**)**return** head;  
 head.**next** = deleteDuplicates(head.**next**);  
 **return** head.**val**==head.**next**.**val**? head.**next** : head;  
 }  
 *//****TODO 记忆此方法* public** ListNode deleteDuplicates1(ListNode head) {  
 **if** (head == **null**) **return** head;  
 ListNode p = head;  
 **while** (p.**next** != **null**){  
 **if** (p.**val** == p.**next**.**val**){  
 p.**next** = p.**next**.**next**; *//****TODO 这里不能添加p = p.next;***  
 } **else** {  
 p = p.**next**; *//****TODO 只有值不同时才往下走*** }  
 }  
 **return** head;  
 }

}

*/\*\*  
 \* 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.*

RemoveDuplicatesFromSortedListII 前向探照两节点

**public class** RemoveDuplicatesFromSortedListII {  
 */\*\*  
 \* 遍历中探测当前节点的下两个节点* **public** ListNode deleteDuplicates1(ListNode head) {  
 ListNode dummy = **new** ListNode(0);  
 dummy.**next** = head;  
 ListNode p = dummy;  
 **while**(p.**next**!=**null** && p.**next**.**next**!=**null**){  
 **if**(p.**next**.**val**==p.**next**.**next**.**val**){  
 **int** dup = p.**next**.**val**;  
 **while**(p.**next**!=**null** && p.**next**.**val**==dup){  
 p.**next** = p.**next**.**next**; *//****TODO 这里不能添加p = p.next;***  
 }  
 }**else**{  
 p = p.**next**;  
 }  
 }  
 **return** dummy.**next**;  
 }  
}

*/\*\*Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.  
 Note:  
 You must not modify the array (assume the array is read only).  
 You must use only constant, O(1) extra space.  
 Your runtime complexity should be less than O(n2).  
 There is only one duplicate number in the array, but it could be repeated more than once.  
 \* Created by eugene on 16/5/24.*

FindTheDuplicateNumber HARD FastSlow

**public class** FindTheDuplicateNumber {  
 */\*\*  
 \* 详细解释参考 https://segmentfault.com/a/1190000003817671  
 \* 时间O(N),空间 O(1).  
 \* 将数组下标与其值映射起来,若有重复值,则连续映射下去会出现环路,环路的起点就是重复值.  
 用快慢两个下标都从0开始，快下标每轮映射两次，慢下标每轮映射一次，直到两个下标再次相同。保持慢下标位置不变，再用一个新的下标从0开始，这两个下标都继续每轮映射一次，当这两个下标相遇时，就是环的起点(重复的数)。* **public int** findDuplicate(**int**[] nums) {  
 **int** slow = nums[0];  
 **int** fast = nums[nums[0]];  
 **while** (slow!=fast){  
 slow = nums[slow];  
 fast = nums[nums[fast]];  
 }  
 **int** i = 0;  
 **while** (slow!=i){  
 slow = nums[slow];  
 i = nums[i];  
 }  
 **return** i;  
 }

}

**Parenthese \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*  
 \* Given a string containing just the characters '(', ')', '{', '}', '[' and ']',  
 \* determine if the input string is valid.  
 The brackets must close in the correct order, "()" and "()[]{}" are all valid  
 but "()" and "([])" are not.  
 \* Created by DCLab on 11/3/2015.*

ValidParentheses

**public class** ValidParentheses {  
 *//巧妙法,但速度很慢(常规法是利用栈)* **public boolean** isValid(String s) {  
 **int** preLen = -1;  
 **while** (preLen!=s.length()){  
 preLen = s.length();  
 s = s.replace(**"()"**, **""**).replace(**"{}"**, **""**).replace(**"[]"**, **""**);  
 }  
 **return** s.length()==0;  
 }

*}*

**public boolean** isValid1(String s) {  
 LinkedList<Character> stack = **new** LinkedList<>();  
 **for**(**char** c: s.toCharArray()){  
 **switch**(c){  
 **case '('**: **case '['**: **case '{'**: stack.push(c); **break**;  
 **case ')'**: **if**(stack.isEmpty() || stack.pop()!=**'('**) **return false**; **break**; *//****TODO 判空易漏（短路求值）* case ']'**: **if**(stack.isEmpty() || stack.pop()!=**'['**) **return false**; **break**;  
 **case '}'**: **if**(stack.isEmpty() || stack.pop()!=**'{'**) **return false**; **break**;  
 }  
 }  
 **return** stack.isEmpty(); *//****TODO 易漏 如"("这种情况应排除***)

*/\*\*Given a string containing just the characters '(' and ')', find the length of  
 the longest valid (well-formed) parentheses substring.  
 For "(()", the longest valid parentheses substring is "()", which has length = 2.  
 Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.*

LongestValidParentheses HARD DP(open判断) 或栈方法

**public class** LongestValidParentheses {  *\* V[i] represents number of valid parentheses matches from S[j to i] (j<i)  
V[i] = V[i-1]+2+previous matches V[i-(V[i-1]+2)], if S[i]=='}' and open'('count> 0* **public int** longestValidParentheses(String s) {  
 **char**[] chars = s.toCharArray();  
 **int**[] dp = **new int**[chars.**length**];  
 **int** open = 0, max = 0;  
 **for** (**int** i=0; i<chars.**length**; i++) {  
 **if** (chars[i]==**'('**) open++;  
 **if** (chars[i]==**')'** && open>0) {  
 dp[i] = 2+dp[i-1]; *// matches found* **if**(i-dp[i]>0) *// add matches from previous* dp[i] += dp[i-dp[i]];  
 open--;  
 }  
 **if** (dp[i]>max) max = dp[i];  
 }  
 **return** max;  
 }

*}*

*/\*\*  
 \* 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:  
 "((()))", "(()())", "(())()", "()(())", "()()()"  
 \* Created by DCLab on 11/5/2015.*

GenerateParentheses 回溯

**public class** GenerateParentheses { *\* 简洁,易理解与记忆*

*\* 剪枝条件1，左<=num(左右<=2n)  
\* 剪枝条件2，右<=左* **public** List<String> generateParenthesis(**int** n) {  
 List<String> list = **new** ArrayList<>();  
 backtrack(list, **""**, 0, 0, n);  
 **return** list;  
 }  
 **public void** backtrack(List<String> list, String str, **int** open, **int** close, **int** n){  
 **if**(str.length() == n\*2){  
 list.add(str);  
 **return**;  
 }  
 **if**(open < n) backtrack(list, str+**"("**, open+1, close, n);  
 **if**(close < open) backtrack(list, str+**")"**, open, close+1, n);  
 }

}

*/\*\*Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are +, - and \*.  
 Example Input: Input: "2\*3-4\*5"  
 (2\*(3-(4\*5))) = -34  
 ((2\*3)-(4\*5)) = -14  
 ((2\*(3-4))\*5) = -10  
 (2\*((3-4)\*5)) = -10  
 (((2\*3)-4)\*5) = 10  
 Output: [-34, -14, -10, -10, 10]  
 \* Created by eugene on 16/5/5.*

DifferentWaysToAddParentheses 分治递归 左右交叉

**public class** DifferentWaysToAddParentheses {  
 */\*\*  
 \* http://blog.csdn.net/foreverling/article/details/49742089  
 \* 分治法:对于输入字符串，若其中有运算符，则将其分为两部分，分别递归计算其值，然后将左值集合与右值集合进行交叉运算，  
 将运算结果放入结果集中；若没有运算符，则直接将字符串转化为整型数放入结果集中。* **public** List<Integer> diffWaysToCompute(String input) {  
 List<Integer> res = **new** ArrayList<>();  
 **for** (**int** i=0; i<input.length(); i++) {  
 **char** ch = input.charAt(i);  
 **if** (ch==**'+'** || ch==**'-'** || ch==**'\*'**) {  
 List<Integer> left = diffWaysToCompute(input.substring(0, i));  
 List<Integer> right = diffWaysToCompute(input.substring(i + 1));  
 **for** (**int** n : left) {  
 **for** (**int** m : right) {  
 **if** (ch==**'+'**) res.add(n+m);  
 **else if** (ch==**'-'**) res.add(n-m);  
 **else** res.add(n\*m);  
 }  
 }  
 }  
 }  
 **if** (res.size()==0) res.add(Integer.*parseInt*(input));  
 **return** res;  
 }  
}

*/\*\*Remove the minimum number of invalid parentheses in order to make the input string valid.  
 Return all possible results.  
 Note: The input string may contain letters other than the parentheses ( and ).  
  
 Examples:  
 "()())()" -> ["()()()", "(())()"]  
 "(a)())()" -> ["(a)()()", "(a())()"]  
 ")(" -> [""]  
 \* Created by eugene on 16/6/2.*

RemoveInvalidParentheses HARD 回溯 多剪枝

**public class** RemoveInvalidParentheses {  
 */\*\*https://leetcode.com/discuss/72208/easiest-9ms-java-solution  
 \* DFS, 剪枝条件较多,涉及rmL(待移除左括号数),rmR(待移除右括号数),open(尚未匹配的左括号数)* **public** List<String> removeInvalidParentheses(String s) {  
 Set<String> res = **new** HashSet<>(); *//去除重复* **int** rmL = 0, rmR = 0;  
 **for**(**int** i = 0; i < s.length(); i++) { *//注意计算rmL与rmR的巧妙方法* **if**(s.charAt(i)==**'('**) rmL++;  
 **else if**(s.charAt(i)==**')'**) {  
 **if**(rmL!=0) rmL--;  
 **else** rmR++;  
 }  
 }  
 DFS(res, s, 0, rmL, rmR, 0, **new** StringBuilder());  
 **return new** ArrayList<>(res);  
 }  
 **public void** DFS(Set<String> res, String s, **int** start, **int** rmL, **int** rmR, **int** open, StringBuilder sb) {  
 **if**(start==s.length() && rmL==0 && rmR==0 && open==0) {  
 res.add(sb.toString());  
 **return**;  
 }  
 **if**(start==s.length() || rmL<0 || rmR<0 || open<0) **return**;  
 **char** c = s.charAt(start);  
 **int** len = sb.length();  
 **if**(c==**'('**) {  
 DFS(res, s, start+1, rmL-1, rmR, open, sb);  
 DFS(res, s, start+1, rmL, rmR, open+1, sb.append(c));  
 } **else if**(c==**')'**) {  
 DFS(res, s, start+1, rmL, rmR-1, open, sb);  
 DFS(res, s, start+1, rmL, rmR, open-1, sb.append(c));  
 } **else** {  
 DFS(res, s, start+1, rmL, rmR, open, sb.append(c));  
 }  
 sb.setLength(len); *//恢复原长度* }  
}

**String \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**ImplementStrStr**

**public int** strStr(String haystack, String needle) {  
 **int** l1 = haystack.length(), l2 = needle.length();  
 **if**(l2==0) **return** 0; *//****TODO 不能漏* for**(**int** start=0; start<=l1-l2; start++){  
 **for**(**int** j=0; j<l2; j++){  
 **if**(haystack.charAt(start+j)!=needle.charAt(j)) **break**;  
 **if**(j==l2-1) **return** start;  
 }  
 }  
 **return** -1;  
}

*/\*\*Given an input string, reverse the string word by word.  
 For example,  
 Given s = "the sky is blue",  
 return "blue is sky the".  
 \* Created by eugene on 16/3/10.*

ReverseWordsInString

**public class** ReverseWordsInString {  
 *//神奇的写法* **Changes to the returned list "write through" to the array****public** String reverseWords(String s) {  
 String[] words = s.trim().split(**" +"**); //TODO 注意trim()  
 Collections.*reverse*(Arrays.*asList*(words));*//****TODO words被in place修改了?* return** String.*join*(**" "**, words);  
 }  
 *//注意split对""的结果返回长度为1的数组,且第一个元素为""* **public** String reverseWords1(String s) {  
 **if** (s.length()==0) **return ""**;  
 String trimed = s.trim();  
 **if** (trimed.length()==0) **return ""**;  
 String[] strings = trimed.split(**"\\s+"**);  
 StringBuilder builder = **new** StringBuilder();  
 **for** (**int** i=strings.**length**-1; i>=0; i--){  
 builder.append(strings[i]+**" "**);  
 }  
 **return** builder.toString().trim();  
 }  
}

StringToInteger

**public class** StringToInteger {  
 */\*\*****TODO 此题用例极易出错,注意各种条件*****public int** myAtoi(String str) {  
 **int** i = 0, sign = 1;  
 **long** total = 0; *//****TODO 使用long* if**(str.length()==0) **return** 0;  
 **while**(str.charAt(i)==**' '** && i<str.length()) i++; *//1. Remove Spaces* **if**(str.charAt(i)==**'+'** || str.charAt(i)==**'-'**){ *//2. Handle signs* sign = str.charAt(i)==**'+'** ? 1 : -1;  
 i++;  
 }  
 **while**(i<str.length()){ *//3. Convert number and avoid overflow* **int** digit = str.charAt(i)-**'0'**;  
 **if**(digit<0 || digit>9) **break**;  
 total = 10\*total + digit;  
 **if** (total\*sign>=Integer.***MAX\_VALUE***) **return** Integer.***MAX\_VALUE***; *//4. Check* **else if** (total\*sign<=Integer.***MIN\_VALUE***) **return** Integer.***MIN\_VALUE***;  
 i++;  
 }  
 **return** (**int**)total\*sign;  
 }

}

*/\*\*You are given a string, s, and a list of words,  
 \* words, that are all of the same length.  
 \* Find all starting indices of substring(s) in s that is a concatenation of each word in words \* exactly once and without any intervening characters.  
 \* 注意：words中的word可以重复  
 For example, given: s: "barfoothefoobarman" words: ["foo", "bar"]  
 You should return the indices: [0,9]. (order does not matter).*

SubstringWithConcatenationOfAllWords HARD 哈希

**public class** SubstringWithConcatenationOfAllWords { *//****TODO HARD*****public static** List<Integer> findSubstring(String s, String[] words) {  
 List<Integer> res = **new** ArrayList<>();  
 **if** (s == **null** || words == **null** || words.**length** == 0) **return** res;  
 **int** wL = words[0].length(); *// length of each word* HashMap<String, Integer> map = **new** HashMap<>(); *// map for words* **for** (String w : words) map.put(w, map.containsKey(w) ? map.get(w) + 1 : 1);  
 **for** (**int** i = 0; i <= s.length() - wL\*words.**length**; i++) {  
 HashMap<String, Integer> copy = **new** HashMap<>(map);  
 **for** (**int** j = 0; j < words.**length**; j++) { *// checkc if match* String str = s.substring(i + j\*wL, i + j\*wL + wL); *// next word* **if** (copy.containsKey(str)) { *// is in remaining words* **int** count = copy.get(str);  
 **if** (count == 1) copy.remove(str);  
 **else** copy.put(str, count - 1);  
 } **else break**; *// not in words* **if** (copy.isEmpty()) { *// matches* res.add(i); **break**;  
 }  
 }  
 }  
 **return** res;  
 }

}

*/\*\*Given a string, find the length of the longest substring without repeating characters. \* For example, the longest substring without repeating letters for "abcabcbb" is "abc", \* which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.*

LongestSubstringWithoutRepeatingCharacters 窗口 哈希

**public class** LongestSubstringWithoutRepeatingCharacters { *\* 简洁清晰,滑动窗口法O(n),借助HashMap* **public int** lengthOfLongestSubstring(String s) {  
 **if** (s.length()==0) **return** 0;  
 HashMap<Character, Integer> map = **new** HashMap<>();  
 **int** max = 0, r = 0, l = 0;  
 **for** (; r<s.length(); ++r){  
 **if** (map.containsKey(s.charAt(r))){  
 l = Math.*max*(l, map.get(s.charAt(r))+1); //TODO l收缩易错,l只往前走}  
 map.put(s.charAt(r), r); *//update pos* max = Math.*max*(max, r-l+1);  
 }  
 **return** max;  
 }

*}*

*/\*\*Given a string S and a string T, find the minimum window in S which will contain \* all the characters in T in complexity O(n).  
 For example, S = "ADOBECODEBANC", T = "ABC" Minimum window is "BANC".  
 Note: If there is no such window in S that covers all characters in T, return the empty string "".  
 If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.*

MinimumWindowSubstring HARD 窗口 模板 substring类 双指针

**public class** MinimumWindowSubstring { *\** ***TODO 此法可作为substring类题目解答模板*** *\* 简洁. 双指针，动态维护一个区间。尾指针不断往后扫，当扫到有一个窗口包含了所有T的字符后，  
 然后再收缩头指针，直到不能再收缩为止。最后记录所有可能的情况中窗口最小的。* **public** String minWindow(String s, String t) {  
 **char**[] S = s.toCharArray(), T = t.toCharArray();  
 **int**[] map = **new int**[256];  
 **int** j = 0, i = 0, count = T.**length**, minLen = Integer.***MAX\_VALUE***, minI = 0;  
 **for**(**char** c: T) map[c]++;  
 **while**(j<s.length()) {  
 **if**(map[S[j]]>0) count--; *//in t* map[S[j]]--; *//****TODO 此行不能移入上述if块* while**(count==0) { *//valid* **if**((j-i+1) < minLen) { *//update min len and min start* minLen = j-i+1;  
 minI = i;  
 }  
 map[S[i]]++;  
 **if**(map[S[i]]>0) count++; *//make it invalid 收缩* i++;  
 }  
 j++;  
 }  
 **if**(minI+minLen > s.length()) **return ""**;  
 **return** s.substring(minI, minI+minLen);  
 }

}

**Palindrome \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s.  
 For example, given s = "aab",  
 Return [ ["aa","b"], ["a","a","b"] ]  
 \* Created by eugene on 16/2/19.*

PalindromePartitioning 标准回溯范式

**public class** PalindromePartitioning { *\* 回溯法(dfs),当然DP也可以(可参考Palindrome Partitioning II).* **public** List<List<String>> partition(String s) {  
 List<List<String>> res = **new** ArrayList<>();  
 dfs(res, s, 0, **new** ArrayList<>());  
 **return** res;  
 }  
 **private void** dfs(List<List<String>> res, String s, **int** start, List<String> sol){  
 **if**(start==s.length()){  
 res.add(**new** ArrayList<>(sol));  
 **return**;  
 }  
 **for**(**int** i=start; i<s.length(); i++){  
 String str = s.substring(start, i+1);  
 **if**(isPalindrome(str)){  
 sol.add(str);  
 dfs(res, s, i+1, sol);  
 sol.remove(sol.size()-1);  
 }  
 }  
 }  
 **private boolean** isPalindrome(String s){  
 **int** l = 0, r = s.length()-1;  
 **while**(l<r){  
 **if**(s.charAt(l++)!=s.charAt(r--)) **return false**;  
 }  
 **return true**;  
 }  
}

*/\*\*Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s.  
 For example, given s = "aab",  
 Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut. \* Created by eugene on 16/2/20.*

PalindromePartitioningII HARD 双重DP(1D+2D) 回溯超时

**public class** PalindromePartitioningII {*/\*\*http://www.programcreek.com/2014/04/leetcode-palindrome-partitioning-ii-java/ \* DP(双重二维),也可以用回溯法(参考Palindrome Partitioning).  
 \* 定义cutNum(i)为[0,i]间最小切割数, 有  
 cutNum(0,i)=min[curNum(0,j-1)+curNum(j,i)],  
 即cut[i] = min{cut[j-1]+1} (j<=i), if [j,i] is palindrome.  
 核心思想在于"j从0到i的过程中每找到一个回文,若要切割则在前次切割的基础上加1次".  
 \* 判断[j,i]间是否为回文也是DP问题,定义palindrome(j,i)为[j,i]间是否为回文,  
 palindrome(j,i)= str(j)==str(i) && (i-j<=1 || palindrome(j+1, i-1)).  
 i-j<=1因为一个字符一定也是回文(这个条件使得边界情况初始化可以省略).* **public int** minCut(String s) {  
 **int** len = s.length();  
 **boolean**[][] palindrome = **new boolean**[len][len];  
 **int**[] cut = **new int**[len];  
 **for** (**int** i=0; i<len; i++){  
 cut[i] = i;  
 **for** (**int** j=0; j<=i; j++){  
 **if** (s.charAt(j)==s.charAt(i) && (i-j<=1 || palindrome[j+1][i-1])){  
 palindrome[j][i] = **true**;  
 **if** (j==0) cut[i] = 0; *//无需切割* **else** cut[i] = Math.*min*(cut[i], cut[j-1]+1);  
 }  
 }  
 }  
 **return** cut[len-1];  
 }  
}

*/\*\*最长连续回文串  
 \* Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.  
 \* Created by DCLab on 11/25/2015.*

LongestPalindromicSubstring 中间发散 DP会超时

**public class** LongestPalindromicSubstring { *\* Time O(n^2), Space O(1)  
 \* 注意使用DP会在特殊用例超时,这里采用从中间向两边扩展的方法.* **public** String longestPalindrome(String s) {  
 **if** (s==**null**) **return null**; **if** (s.length()<=1) **return** s;   
 String res = s.substring(0, 1);  
 **for** (**int** i=0; i<s.length(); i++) {  
 String p = extendPalindrome(s, i, i); *//奇数扩展* **if** (p.length()>res.length()) res = p;  
 p = extendPalindrome(s, i, i+1); *//偶数扩展* **if** (p.length()>res.length()) res = p;  
 }  
 **return** res;  
 }  
 *// get longest palindrome with center of (l,r)* **private** String extendPalindrome(String s, **int** l, **int** r) {  
 **while** (l>=0 && r<=s.length()-1 && s.charAt(l)==s.charAt(r)) {  
 l--; r++;//TODO 注意不是l++; r--;   
 }  
 **return** s.substring(l + 1, r);  
 }

*}*

*/\*\*Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.  
 For example:  
 Given "aacecaaa", return "aaacecaaa".  
 Given "abcd", return "dcbabcd".  
 \* Created by eugene on 16/4/10.*

ShortestPalindrome HARD

**public class** ShortestPalindrome {  
 */\*\* 此解易理解,但3%接近超时  
 \* https://leetcode.com/discuss/61416/my-9-lines-three-pointers-java-solution-with-explanation  
 \* 寻找从第一个字符起的最长回文串,再逆转剩余部分并拼接到字符串首部* **public** String shortestPalindrome(String s) {  
 **int** i = 0, end = s.length()-1, j = end;  
 **char** chs[] = s.toCharArray();  
 **while**(i < j) {  
 **if** (chs[i] == chs[j]) {  
 i++; j--;  
 } **else** { *//重置* i = 0; end--; j = end;  
 }  
 }  
 **return new** StringBuilder(s.substring(end+1)).reverse().append(chs).toString();  
 }

}

PalindromeNumber

**public class** PalindromeNumber {  
 *//先逆转整数,注意overflow* **public boolean** isPalindrome(**int** x) {  
 **if** (x==0) **return true**;  
 **if** (x<0) **return false**;  
 **if** (*reverse*(x)==x) **return true**;  
 **return false**;  
 }  
 **private static int** reverse(**int** x){  
 **long** res = 0;  
 **while** (x!=0){  
 res = res\*10 + x%10;  
 x = x/10;  
 **if** (res>Integer.***MAX\_VALUE*** || res<Integer.***MIN\_VALUE***) **return** 0;  
 }  
 **return** (**int**)res;  
 }  
}

*/\*\*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.(amanaplanacanalpanama)  
 "race a car" is not a palindrome.(raceacar)  
 Note: Have you consider that the string might be empty?  
 For the purpose of this problem, we define empty string as valid palindrome.  
 \* Created by eugene on 16/2/13.*

ValidPalindrome

**public class** ValidPalindrome {  
 *//不借助正则,最快* **public boolean** isPalindrome(String s) {  
 **if** (s.isEmpty()) **return true**;  
 **int** l = 0, r = s.length()-1;  
 **char** c1, c2;  
 **while**(l<r) {  
 c1 = s.charAt(l); c2 = s.charAt(r);  
 **if** (!Character.*isLetterOrDigit*(c1)) l++;  
 **else if**(!Character.*isLetterOrDigit*(c2)) r--;  
 **else** {  
 **if** (Character.*toLowerCase*(c1) != Character.*toLowerCase*(c2)) **return false**;  
 l++; r--;  
 }  
 }  
 **return true**;  
 }  
 *//取巧* **public boolean** isPalindrome1(String s) {  
 String actual = s.replaceAll(**"[^A-Za-z0-9]"**, **""**).toLowerCase();  
 String rev = **new** StringBuffer(actual).reverse().toString();  
 **return** actual.equals(rev);  
 }

}

**Graph \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

CloneGraph BFS 哈希

**public class** CloneGraph {  
 **class** UndirectedGraphNode {  
 **int label**;  
 List<UndirectedGraphNode> **neighbors**;  
 UndirectedGraphNode(**int** x) { **label** = x; **neighbors** = **new** ArrayList<UndirectedGraphNode>(); }  
 } *\* 无向图深拷贝,BFS***public** UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {  
 **if**(node==**null**) **return null**;  
 LinkedList<UndirectedGraphNode> q = **new** LinkedList<>();  
 HashMap<UndirectedGraphNode, UndirectedGraphNode> map = **new** HashMap<>();  
 q.offer(node);  
 UndirectedGraphNode newHead = **new** UndirectedGraphNode(node.**label**);  
 map.put(node, newHead);  
 **while**(!q.isEmpty()){  
 UndirectedGraphNode old = q.poll();  
 **for**(UndirectedGraphNode oldN: old.**neighbors**){  
 **if**(!map.containsKey(oldN)){  
 map.put(oldN, **new** UndirectedGraphNode(oldN.**label**));  
 q.offer(oldN);  
 }  
 map.get(old).**neighbors**.add(map.get(oldN));*//****TODO 注意是add(map.get(oldN))，不是add(oldN)*** }  
 }  
 **return** newHead;  
}

*}*

*/\*\*For a undirected graph with tree characteristics, we can choose any node as the root.The result graph is then a rooted tree. Among all possible rooted trees,  
 those with minimum height are called minimum height trees (MHTs).  
 Given such a graph, write a function to find all the MHTs and return a list of their root labels.  
 Format： The graph contains n nodes which are labeled from 0 to n - 1.  
 You will be given the number n and a list of undirected edges (each edge is a pair of labels). You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.  
 Note: (1) According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices  
 are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.”  
 (2) The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.  
 \* Created by eugene on 16/6/11.*

MinimumHeightTrees BFS

**public class** MinimumHeightTrees {  
 */\*\*https://leetcode.com/discuss/71763/share-some-thoughts  
 \* Remove the leaves, update the degrees of inner vertexes. Then remove the new leaves. Doing so level by level until there are 2 or 1 nodes left. What's left is our answer! The time complexity and space complexity are both O(n).* **public** List<Integer> findMinHeightTrees(**int** n, **int**[][] edges) {  
 **if** (n==1) **return** Collections.*singletonList*(0);  
 ArrayList<HashSet<Integer>> adjs = **new** ArrayList<>(n);  
 **for** (**int** i=0; i<n; i++) adjs.add(**new** HashSet<>());  
 **for** (**int**[] e: edges){  
 adjs.get(e[0]).add(e[1]);  
 adjs.get(e[1]).add(e[0]);  
 }  
 ArrayList<Integer> leaves = **new** ArrayList<>();  
 **for** (**int** i=0; i<n; i++)  
 **if** (adjs.get(i).size()==1) leaves.add(i);  
 **while** (n>2){  
 n -= leaves.size();  
 ArrayList<Integer> newLeaves = **new** ArrayList<>();  
 **for** (**int** i: leaves){  
 **int** adjOfLeaf = adjs.get(i).iterator().next();  
 adjs.get(adjOfLeaf).remove(i);  
 **if** (adjs.get(adjOfLeaf).size()==1) newLeaves.add(adjOfLeaf);  
 }  
 leaves = newLeaves;  
 }  
 **return** leaves;  
 }  
}

**Bit \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*/\*\*Given an array of integers, every element appears three times except for one. Find that single one. Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?*

SingleNumberII

**public class** SingleNumberII {  
 *\* 由于x^x^x = x，无法直接利用I的方法来解。  
 但可以应用类似的思路，即利用位运算来消除重复3次的数。以一个数组[14 14 14 9]为例，  
 将每个数字以二进制表达：  
 1110  
 1110  
 1110  
 1001  
 \_\_\_\_\_  
 4331 对每一位进行求和  
 1001 对每一位的和做%3运算，来消去所有重复3次的数* **public int** singleNumber(**int**[] nums) {  
 **int** result = 0;  
 **for** (**int** i=31; i>=0; i--){  
 **int** sum = 0, mask = 1<<i;  
 **for**(**int** n: nums) **if**((n&mask)!=0) sum++;  
 result = (result<<1) + sum%3;  
 }  
 **return** result;  
 }

*}*

*/\*\*Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.  
 For example, Given nums = [0, 1, 3] return 2.  
 Note: Your algorithm should run in linear runtime complexity.  
 Could you implement it using only constant extra space complexity?  
 \* Created by eugene on 16/5/15.*

MissingNumber

**public class** MissingNumber {  
 *//位操作(i与num[i]异或)* **public int** missingNumber(**int**[] nums) {  
 **int** xor = 0, i = 0;  
 **for** (i=0; i<nums.**length**; i++) {  
 xor ^= i^nums[i];  
 }  
 **return** xor^i;  
 }  
 *//另一种方法：先求[1,n]间的和,再减去给定数组之和*

*}*

*/\*\*Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.  
 Example 1: Given ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"] Return 16  
 The two words can be "abcw", "xtfn".  
 Example 2: Given ["a", "ab", "abc", "d", "cd", "bcd", "abcd"] Return 4  
 The two words can be "ab", "cd".  
 Example 3: Given ["a", "aa", "aaa", "aaaa"] Return 0  
 No such pair of words.  
 \* Created by eugene on 16/6/15.*

MaximumProductOfWordLengths

**public class** MaximumProductOfWordLengths {  
 */\*\*****TODO 注意理解题意,length(word[i])\*length(word[j])表示两个单词完整长度乘积*** *\* https://leetcode.com/discuss/82920/java-easy-version-to-understand  
 \* 位操作, O(n^2)* **public int** maxProduct(String[] words) {  
 **if** (words == **null** || words.**length** <= 1) **return** 0;  
 **int** len = words.**length**;  
 **int**[] masks = **new int**[len];  
 **for** (**int** i=0; i<len; i++) { *//用一个int的26位去保存每个单词所包含的字母的信息* masks[i] = 0;  
 **for** (**int** j=0; j<words[i].length(); j++) {  
 masks[i] |= (1 << words[i].charAt(j)-**'a'**);  
 }  
 }  
 **int** max = 0;  
 **for** (**int** i = 0; i < len; i++) {  
 **for** (**int** j = i + 1; j < len; j++) {  
 **if** ((masks[i] & masks[j]) == 0) {  
 max = Math.*max*(max, words[i].length()\*words[j].length());  
 }  
 }  
 }  
 **return** max;  
 }

}