Table of Contents

Google Inteview 120+

Try to solve the 120+ LeetCode problems labeled as Google Interview.

简述各题

4.Median of Two Sorted Arrays：二分法，注意细节，各种-1， + 1。

10.Regular Expression Matching: 注意为*时的三种表达，为多个字母，一个字母和空。

66.Plus One: 注意carry = 0的时候就可以结束了。

31.Next Permutation: 原理我始终没搞明白，但是好神奇，有空问问。reverse的时候要包括second。

274.H-Index: 注意j=len-i-1,注意条件是j>=citations[i]。注意各种细节。

360.Sort Transformed Array: two pointers，注意 a >=0 和a<0时输入到result的时候顺序得变，两个得数的判定情况也得变。

280.Wiggle Sort: 算法精妙，注意in-place。注意观察数的规律，奇数位需要比左边的偶数位的大，偶数位的要比左边奇数位的小。

281.Wiggle Sort II:

139.Word Break:

140.Word Break II:!!!

133.Clone Graph:

208.Implement Trie:

314.Binary Tree Vertical Order Traversal

146.LRU Cache:

460.LFU Cache:

312.Burst Balloons:

200.Number of Islands:

201.Number of IslandsII:

293.Flip Game:

294.Flip Game II:

375.Guess Number Higher or Lower:

376.Guess Number Higher or LowerII: !!!

224.Basic Calculator:

282.Expression Add Operators:

320.Generalized Abbreviation:注意怎样递归，没算复杂度

22.Generate Parentheses: !!!注意怎样递归，没算复杂度

239.Search a 2D Matrix II： 二分法 还没做

240.Search a 2D Matrix II: 从右上或左下角开始。

288.Unique Word Abbreviation: 注意各种情况，字典里本来就有重复，以及字典里只有一个相同的。

329.Longest Increasing Path in a Matrix: 用DP，要用一个数组记住已经算出节点的最大值。普通dfs会超时。

246.Strobogrammatic Number: 注意string的创建方法， 可用Arrays.asList("00", "11", "88"，“696”) .

247.Strobogrammatic Number II：一层一层剥，可以重写一遍，注意0的加入。

20.Valid Parentheses: 注意stack是空的情况。

22.Generate Parentheses: 注意右括号要比左括号少才能加。

490.The Maze: 注意dfs的while里多加一次，要减掉。

505.The Maze II:visited变为distance，当值更小时更新。bfs可用PQ。

323.Number of Connected Components in an Undirected Graph

314.Binary Tree Vertical Order Traversal

394.Decode String: 俩stack，先放空的，push结果时要加上之前stack里的。

377.Combination Sum IV:仔细想想怎么做，公式！初始化！

316.Remove Duplicate Letters:注是subsequence，注意判断条件。

354.Russian Doll Envelopes：注意Arrays.binarySearch的用法。

474.Ones and Zeroes: DP,注意公式，注意一定要从后往前推。

503.Next Greater Element II:stack, 走两遍，第一遍倒着走第二遍顺着走。

501.Find Mode in Binary Search Tree: 注意list.clear()。

298.Binary Tree Longest Consecutive Sequence

494.Target Sum

485.Max Consecutive Ones

487.Max Consecutive Ones II

475.Heaters

498.Diagonal Traverse

486.Predict the Winner

310.Minimum Height Trees: 循环：n>2,注意iterator.next()，set不能转int。

239.Sliding Window Maximum: deque用法，先判断deque是否为空。

54.Spiral Matrix

56.Merge Intervals

23.Merge K sorted Lists

284.Peeking Iterator: 注意全局变量和next()方法的区别。

480.Sliding Window Median: 注意(double)加法和随时平衡两个堆。

361.Bomb Enemy:行不用数组因为可一直更新，注意算行列敌人时初始化。

341.Flatten Nested List Iterator: 用stack，注意while

281.Zigzag Iterator:注意iterator用法。

269.Alien Dictionary: 没写完太长，回忆topo sort,前一个数建图

380.Insert Delete GetRandom O(1)

228.Summary Ranges

370.Range Addition

295.Find Median from Data Stream:用Collections.reverseOrder或“-”

259.3Sum Smaller: res += right - left,中间一串数都是。

346.Moving Average from Data Stream

50.Pow(x, n)

286.Walls and Gates

351.Android Unlock Patterns

200.Number of Islands

305.Number of Islands II

271.Encode and Decode Strings

173.Binary Search Tree Iterator

57.Insert Interval

163.Missing Ranges

218.The Skyline Problem

42.Trapping Rain Water

407.Trapping Rain Water II：从外往里,新加点要和当前cell.height比较取大。

297.Serialize and Deserialize Binary Tree

406.Queue Reconstruction by Height:排好序后顺序按p[2]位置插入。

391.Perfect Rectangle

471.Encode String with Shortest Length:看一下kmp算法优化

465.Optimal Account Balancing: dfs，一个一个来,前一个人加完就不用算了

159.Longest Substring with At Most Two Distinct Characters: 类似topo sort

340.Longest Substring with At Most K Distinct Characters

411.Minimum Unique Word Abbreviation:太长没写 有空看

336.Palindrome Pairs

408.Valid Word Abbreviation:num的表示,最后的相加,比较对应位。

353.Design Snake Game: 注意吃进去和减掉，从头吃从尾巴减，先减。

362.Design Hit Counter: 注意循环变换，timestamp-times[i]< 300

155.Min Stack: 注意minStack为空的情况。

251.Flatten 2D Vector: 注意hasNext和next运行完后x和y的值。

276.Paint Fence:当前不能和前一个一样或不能和再前一个一样。

307.Range Sum Query - Mutable:学习建线段树再写一次。

308.Range Sum Query 2D - Mutable:看update和sum中计算最低位的方法。http://www.cnblogs.com/grandyang/p/4985506.html

348.Design Tic-Tac-Toe

400.Nth Digit???没做好

253.Meeting Rooms II：注意heap的初始化。

345.Reverse Vowels of a String:看了一眼没做可以想想

228.Summary Ranges

293.Flip Game

294.Flip Game II:注意map方法递归if (!helper(t, map))

128.Longest Consecutive Sequence: 注意set的删减

289.Game of Life:在左边放第二状态然后右移。注意计算时不能出界。

369.Plus One Linked List:注意一定要新建Node,carry放node.next

162.Find Peak Element:num[-1]=num[n]=-∞,和左右比较即可。

309.Best Time to Buy and Sell Stock with Cooldown:再做。

356.Line Reflection: 看清定义，map的value为set。

447.Number of Boomerangs: res+=val*(val - 1)和map.clear();

358.Rearrange String k Distance Apart:waitQueue先加后判断count是否为0

417.Pacific Atlantic Water Flow

326.Power of Three: 机智的做法，取最大取余，可以记一下

231.Power of Two: Integer.bitCount(n) == 1

261.Graph Valid Tree:看一下树的定义，注意判断是否为cycle。

257.Binary Tree Paths: 注意stringBuilder的backtracking方法

359.Logger Rate Limiter

389.Find the Difference: 可以再想一下，用亦或做。

317.Shortest Distance from All Buildings

425.Word Squares

331.Verify Preorder Serialization of a Binary Tree:while循环外push。

422.Valid Word Square: 注意先看有没有超过length。

493.Reverse Pairs:有空看merge sort做法。

315.Count of Smaller Numbers After Self

482.License Key Formatting

Arrays

Plus one

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.

Tips: 从数组的最后一位开始运算，先赋值原数组。遇到需要进位从而比原数组多一位的情况，记得重建数组。

代码：

```
public class Solution {
    public int[] plusOne(int[] digits) {
        int carry = 1;
        for (int i = digits.length - 1; i >= 0 && carry > 0; i--
) {
            int sum = digits[i] + carry;
            digits[i] = sum % 10;
            carry = sum / 10;
        }
        if (carry == 0) {
            return digits;
        }
        int[] result = new int[digits.length + 1];
        result[0] = carry;
        for (int i = 0; i < digits.length; i++) {
            result[i + 1] = digits[i];
        }
        return result;
    }
}
```

Next Permutation

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

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

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

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

Tips:

题意： 比如1，2，3，4，就是这四个数所有的permutations，按order排序，然后这个permutation在排序中的下一个。

O(n)

1. 在当前序列中，从尾端往前寻找两个相邻元素，前一个记为first，后一个记为second，并且满足first < second。
2. 然后再从尾端寻找另一个元素number，如果满足first 小于number，即将第first个元素与number元素对调.
3. 并将second元素之后（包括second）的所有元素颠倒排序，即求出下一个序列

example:

6，3，4，9，8，7，1

1. 此时 first ＝ 4，second = 9
2. 从尾巴到前找到第一个大于first的数字，就是7
3. 交换4和7，即上面的swap函数，此时序列变成6，3，7，9，8，4，1

4. 再将second＝9以及以后的序列重新排序，让其从小到大排序，使得整体最小，即reverse一下（因为此时肯定是递减序列）

得到最终的结果：6，3，7，1，4，8，9

1. Start from its last element, traverse backward to find the first one with index i that satisfy num[i-1] < num[i]. So, elements from num[i] to num[n-1] is reversely sorted.
2. To find the next permutation, we have to swap some numbers at different positions, to minimize the increased amount, we have to make the highest changed position as high as possible. Notice that index larger than or equal to i is not possible as num[i,n-1] is reversely sorted. So, we want to increase the number at index i-1, clearly, swap it with the smallest number between num[i,n-1] that is larger than num[i-1]. For example, original number is 121543321, we want to swap the '1' at position 2 with '2' at position 7.
3. The last step is to make the remaining higher position part as small as possible, we just have to reversely sort the num[i,n-1]

Code:

```
public class Solution {
    public void nextPermutation(int[] nums) {
        // find two adjacent elements, n[i-1] < n[i]
        int i = nums.length - 1;
        for (; i > 0; i --) {
            if (nums[i] > nums[i-1]) {
                break;
            }
        }
        if (i != 0) {
            // swap (i-1, x), where x is index of the smallest n
umber in [i, n)
            int x = nums.length - 1;
            for (; x >= i; x --) {
                if (nums[x] > nums[i-1]) {
                    break;
                }
            }
            swap(nums, i - 1, x);
        }
        reverse(nums, i, nums.length - 1);
    }

    void swap(int[] a, int i, int j) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
    // reverse a[i, j]
    void reverse(int[] a, int i, int j) {
        for (; i < j; i ++, j --) {
            swap(a, i, j);
        }
    }
}
```

H-Index

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: A scientist has index h if h of his/her N papers have at least h citations each, and the other N − h papers have no more than h citations each.

For example, given citations = [3, 0, 6, 1, 5], which means the researcher has 5 papers in total and each of them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, his h-index is 3.

Note: If there are several possible values for h, the maximum one is taken as the h-index.

Hint:

An easy approach is to sort the array first. What are the possible values of h-index? A faster approach is to use extra space.

Tips

可以按照如下方法确定某人的H指数：

1、将其发表的所有SCI论文按被引次数从高到低排序；

O(nlogn)时间，**O(1)**空间。现将数组排序。然后从大到小遍历（traverse），一边计数一边比较计数与当前的引用数，直到计数大于引用数。

2、从前往后查找排序后的列表，直到某篇论文的序号大于该论文被引次数。所得序号减一即为H指数。

只是用多余的一个空间记了个数，也是相当于排序了，其实和1是一样的。

O(n)时间，**O(n)**空间。

使用一个额外的数组，其下标为引用数，置为为具有该引用数的文章数量。注意，根据定义，H-index的上限不可能超过文章总数n。因此我们只需要额外开一个长度为n的数组即可。然后对新数组按引用数从大到小遍历，一边计数一边比较计数与当前的引用数，直到计数大于引用数。

Code:

方法一：

```
public class Solution {
    public int hIndex(int[] citations) {
        int len = citations.length;
        Arrays.sort(citations);
        for (int i = citations.length - 1; i >= 0; i--) {
            int j = citations.length - i - 1;
            if (j >= citations[i]) {
                return j;
            }
        }
        return citations.length;
    }
}
```

方法二；

```
public class Solution {
    public int hIndex(int[] citations) {
        int len = citations.length;
        if(citations == null || len == 0) return 0;
        int[] counts = new int[len + 1];
        for(int c : citations){
            if(c > len) counts[len]++;
            else counts[c]++;
        }
        if(counts[len] >= len) return len;
        for(int i = len - 1; i >= 0; i--){
            counts[i] += counts[i + 1];
            if(counts[i] >= i) return i;
        }
        return 0;
    }
}
```

Sort Transformed Array

Given a sorted array of integers nums and integer values a, b and c. Apply a function of the form f(x) = ax2 + bx + c to each element x in the array.

The returned array must be in sorted order.

Expected time complexity: O(n)

Example:

```
nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5,

Result: [3, 9, 15, 33]

nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5

Result: [-23, -5, 1, 7]
```

Tips:

用two pointers做非常容易，但是要注意 a >=0 和a < 0两种情况，输入到result的时候顺序得变，之后两个得到的数的判定情况也得变。

Code:

```
public class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b,
int c) {
        int[] result = new int[nums.length];
        if (nums == null || nums.length == 0) {
            return result;
        }
        int left = 0, right = nums.length - 1;
        int i = a >= 0 ? nums.length - 1 : 0;
        while(left <= right) {
            int l = a * nums[left] * nums[left] + b * nums[left]
 + c;
            int r = a * nums[right] * nums[right] + b * nums[rig
ht] + c;
            if (a >= 0) {
                if (l <= r) {
                    result[i--] = r;
                    right--;
                } else {
                    result[i--] = l;
                    left++;
                }
            } else {
                if (l > r) {
                    result[i++] = r;
                    right--;
                } else {
                    result[i++] = l;
                    left++;
                }
            }
        }
        return result;
    }
}
```

Wiggle Sort

Given an unsorted array nums, reorder it in-place such that nums[0] <= nums[1] >= nums[2] <= nums[3]....

For example, given nums = [3, 5, 2, 1, 6, 4], one possible answer is [1, 6, 2, 5, 3, 4].

Tips:

题意：

给定一个数组，要求进行如下排序： 奇数的位置要大于两边。 如 nums[1] > nums[0] ，nums[1] > nums[2]

- 方法一

复杂度 时间 O(NlogN) 空间 O(1)

思路

根据题目的定义，摇摆排序的方法将会很多种。我们可以先将数组排序，这时候从第3个元素开始，将第3个元素和第2个元素交换。然后再从第5个元素开始，将第5个元素和第4个元素交换，以此类推。就能满足题目要求。

- 方法二：

复杂度 时间 O(N) 空间 O(1)

思路

题目对摇摆排序的定义有两部分：

如果i是奇数，nums[i] >= nums[i - 1]

如果i是偶数，nums[i] <= nums[i - 1]

所以我们只要遍历一遍数组，把不符合的情况交换一下就行了。具体来说，如果nums[i] > nums[i - 1]， 则交换以后肯定有nums[i] <= nums[i - 1]。

Code:

Naive:

```
public class Solution {
    public void wiggleSort(int[] nums) {
        // 先将数组排序
        Arrays.sort(nums);
        // 将数组中一对一对交换
        for(int i = 2; i < nums.length; i+=2){
            int tmp = nums[i-1];
            nums[i-1] = nums[i];
            nums[i] = tmp;
        }
    }
}
```

O(n):

```
public class Solution {
    public void wiggleSort(int[] nums) {
        for(int i = 1; i < nums.length; i++){
            // 需要交换的情况：奇数时nums[i] < nums[i - 1]或偶数时num
s[i] > nums[i - 1]
            if((i % 2 == 1 && nums[i] < nums[i-1]) || (i % 2 ==
0 && nums[i] > nums[i-1])){
                int tmp = nums[i-1];
                nums[i-1] = nums[i];
                nums[i] = tmp;
            }
        }
    }
}
```

Wiggle Sort II

Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....

Example: (1) Given nums = [1, 5, 1, 1, 6, 4], one possible answer is [1, 4, 1, 5, 1, 6]. (2) Given nums = [1, 3, 2, 2, 3, 1], one possible answer is [2, 3, 1, 3, 1, 2].

Note: You may assume all input has valid answer.

Follow Up: Can you do it in O(n) time and/or in-place with O(1) extra space?

Credits: Special thanks to @dietpepsi for adding this problem and creating all test cases.

Tips:

average O(n) time and O(1) space.

Assume your original array is {6,13,5,4,5,2}. After you get median element, the 'nums' is partially sorted such that the first half is larger or equal to the median, the second half is smaller or equal to the median, i.e

```
13   6   5   5   4   2

         M
```

to get wiggle sort, you want to put the number in the following way such that

(1) elements smaller than the 'median' are put into the last even slots

(2) elements larger than the 'median' are put into the first odd slots

(3) the medians are put into the remaining slots.

```
Index :       0   1   2   3   4   5
Small half:   M       S       S
Large half:       L       L       M
```

M - Median, S-Small, L-Large. In this example, we want to put {13, 6, 5} in index 1,3,5 and {5,4,2} in index {0,2,4}

The index mapping, (1 + 2*index) % (n | 1)（这个是位运算里的或） combined with 'Color sort', will do the job.

After selecting the median element, which is 5 in this example, we continue as the following

```
Mapped_idx[Left] denotes the position where the next smaller-tha
n median element  will be inserted.

Mapped_idx[Right] denotes the position where the next larger-tha
n median element  will be inserted.
```

Step 1:

```
Original idx: 0    1    2    3    4    5
Mapped idx:   1    3    5    0    2    4
Array:        13   6    5    5    4    2
             Left
              i
                                       Right
```

nums[Mapped_idx[i]] = nums[1] = 6 > 5, so it is ok to put 6 in the first odd index 1. We increment i and left.

Step 2:

```
Original idx: 0    1    2    3    4    5
Mapped idx:   1    3    5    0    2    4
Array:        13   6    5    5    4    2
                  Left
                   i
                                       Right
```

nums[3] = 5 = 5, so it is ok to put 6 in the index 3. We increment i.

Step 3:

```
Original idx: 0    1    2    3    4    5
Mapped idx:   1    3    5    0    2    4
Array:        13   6    5    5    4    2
                  Left
                        i
                                      Right
```

nums[5] = 2 < 5, so we want to put it to the last even index 4 (pointed by Right). So, we swap nums[Mapped_idx[i]] with nums[Mapped_idx[Right]], i.e. nums[5] with nums[4], and decrement Right.

Code:

```
public class Solution {

    public void wiggleSort(int[] nums) {
        int median = findKthLargest(nums, (nums.length + 1) / 2)
;
        int n = nums.length;

        int left = 0, i = 0, right = n - 1;

        while (i <= right) {

            if (nums[newIndex(i,n)] > median) {
                swap(nums, newIndex(left++,n), newIndex(i++,n));
            }
            else if (nums[newIndex(i,n)] < median) {
                swap(nums, newIndex(right--,n), newIndex(i,n));
            }
            else {
                i++;
            }
        }
    }

    private int findKthLargest(int[] nums, int k) {
        k = nums.length - k;
        int lo = 0;
        int hi = nums.length - 1;
        while (lo < hi) {
            final int j = partition(nums, lo, hi);
            if(j < k) {
                lo = j + 1;
            } else if (j > k) {
                hi = j - 1;
            } else {
                break;
            }
        }
        return nums[k];
    }

    private int partition(int[] nums, int left, int right) {

        int i = left;
        int j = right + 1;
        while(true) {
            while(i < right && nums[++i] < nums[left]);
            while(j > left && nums[left] < nums[--j]);
            if(i >= j) {
                break;
            }
            swap(nums, i, j);
        }
        swap(nums, left, j);
        return j;
    }

    private int newIndex(int index, int n) {
        return (1 + 2*index) % (n | 1);
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
```

```
            nums[j] = temp;
        }

    }
```

Spiral Matrix

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

For example, Given the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
```

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

Tips:

回旋增加，顺序是向右，向下，向左，向上为一轮。注意判断边界条件即可。

There are 4 modes of changing index. It can go up, down, left or right. Use an array to store the boundaries of 4 modes. Read out one number each time and change the index based on mode and boundaries.

Code:

```
public class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix == null || matrix.length == 0) {
            return result;
        }
        int left = 0;
        int right = matrix[0].length - 1;
        int top = 0;
        int bottom = matrix.length -1;
        while (left <= right && top <= bottom) {
            // traverse right
            for (int i = left; i <= right; i++) {
                result.add(matrix[top][i]);
            }
            top++;
            // traverse down
            for (int j = top; j <= bottom; j++) {
                result.add(matrix[j][right]);
            }
            right--;
            // traverse left: should judge first
            // after changes it can be left > right || top > bot
tom,
            // should return in this case
            if (top <= bottom) {
                for (int i = right; i >= left; i--) {
                    result.add(matrix[bottom][i]);
                }
                bottom--;
            }
            // traver right: also should judge first
            if (left <= right) {
                for (int j = bottom; j >= top; j--) {
                    result.add(matrix[j][left]);
                }
                left++;
            }
        }
        return result;
    }
}
```

Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

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

Tips:

注意要先按照interval的start排序，要重写compareTo方法。

先把第一个加进去，然后判断下一个interval的开始如果小于前一个的end的话，就不加这次的interval，改result里面的end为当前interval的end和result已存最后一个interval的end的最大值。

Code:

```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
    public List<Interval> merge(List<Interval> intervals) {
        List<Interval> result = new ArrayList<>();
        if (intervals == null || intervals.size() < 2) {
            return intervals;
        }
        Collections.sort(intervals, new Comparator<Interval>() {
            public int compare(Interval a, Interval b) {
                return a.start - b.start;
            }
        });
        for (int i = 0; i < intervals.size(); i++) {
            if (result.size() == 0) {
                result.add(intervals.get(i));
                continue;
            }
            if (intervals.get(i).start <= result.get(result.size
() - 1).end) {
                result.get(result.size() - 1).end = Math.max(res
ult.get(result.size() - 1).end, intervals.get(i).end);
            } else {
                result.add(intervals.get(i));
            }
        }
        return result;
    }
}
```

Insert Interval

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1: Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Example 2: Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

Tips:

和Merge Intervals差不多，不断比较interval的start和end，进行插入或合并，分各种情况讨论。

Code:

```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
    public List<Interval> insert(List<Interval> intervals, Inter
val newInterval) {
        List<Interval> result = new ArrayList<>();
        if (intervals == null || intervals.size() == 0) {
            result.add(newInterval);
            return result;
        }
        int index = 0;
        boolean inserted = false;
        while (index < intervals.size()) {
            Interval current = intervals.get(index);
            if (newInterval.start < current.start) {
                if (newInterval.end < current.start) {
                    result.add(newInterval);
                    inserted = true;
                    break;
                }
                else {
                    if (newInterval.end <= current.end) {
                        result.add(new Interval(newInterval.star
t, current.end));
                        index++;
                        inserted = true;
                        break;
                    }
                    else {
                        index++;
                    }
                }
            }
            else {
                if (newInterval.start > current.end) {
                    result.add(current);
                    index++;
                }
                else {
                    if (newInterval.end <= current.end) {
                        inserted = true;
                        break;
                    }
                    else {
                        if (index == intervals.size() - 1) {
                            result.add(new Interval(current.star
t, newInterval.end));
                            inserted = true;
                        }
                        else {
```

```
                        newInterval.start = current.start;
                    }
                    index++;
                }
            }
        }
    }
    while (index < intervals.size()) {
        result.add(intervals.get(index));
        index++;
    }
    if (!inserted) {
        result.add(newInterval);
    }
    return result;
}

}
```

Meeting Rooms

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), determine if a person could attend all meetings.

For example,

Given [[0, 30],[5, 10],[15, 20]],

return false.

Tips:

先sort，然后比较每个interval，只要前一个的end大于后一个的start，就return false。

Code:

```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
    public boolean canAttendMeetings(Interval[] intervals) {
        Arrays.sort(intervals, new Comparator<Interval>() {
            public int compare(Interval a, Interval b) {
                    return a.start - b.start;
            }
        });
        for (int i = 0; i < intervals.length - 1; i++) {
            if (intervals[i].end > intervals[i + 1].start) {
                return false;
            }
        }
        return true;
    }
}
```

Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.

For example,

Given [[0, 30],[5, 10],[15, 20]],

return 2.

Tips:

O(nlogn),因为要sort。

用heap来做，遍历interval，把end放进heap，count为1。如果后面有start的值大于heap.peek(),则count + 1。

Code:

```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
    public int minMeetingRooms(Interval[] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }
        Arrays.sort(intervals, new Comparator<Interval>() {
            public int compare(Interval a, Interval b) {
                return a.start - b.start;
            }
        });
        int count = 1;
        PriorityQueue<Integer> heap = new PriorityQueue<>();
        heap.offer(intervals[0].end);
        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i].start < heap.peek()) {
                count++;
            }
            else {
                heap.poll();
            }
            heap.offer(intervals[i].end);
        }
        return count;
    }
}
```

Range Addition

ssume you have an array of length n initialized with all 0's and are given k update operations.

Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc.

Return the modified array after all k operations were executed.

Example:

Given:

```
length = 5,
updates = [
    [1,  3,  2],
    [2,  4,  3],
    [0,  2, -2]
]
```

Output:

```
[-2, 0, 3, 5, 3]
```

Explanation:

Initial state: [0, 0, 0, 0, 0]

After applying operation [1, 3, 2]: [0, 2, 2, 2, 0]

After applying operation [2, 4, 3]: [0, 2, 5, 5, 3]

After applying operation [0, 2, -2]: [-2, 0, 3, 5, 3]

Hint:

```
Thinking of using advanced data structures?
You are thinking it too complicated.
For each update operation, do you really need to update all elem
ents between i and j?
Update only the first and end element is sufficient.
The optimal time complexity is O(k + n) and uses O(1) extra spac
e.
```

Tips:

注意看hint，每次更新时只更新 start 和 end + 1 位，start加上value代表从这里开始变， end + 1位减去value代表到这里变回来。iterate一遍之后算前向和，存入答案数组即为答案。这个解法真妙！

tricky one. For each update, increment the start index by inc, decrement the end index + 1 by inc. Then do a moving sum at last.

Just store every start index for each value and at end index plus one minus it

for example it will look like:

```
[1 , 3 , 2] , [2, 3, 3] (length = 5)

res[ 0, 2, ,0, 0 -2 ]

res[ 0 ,2, 3, 0, -5]

sum 0, 2, 5, 5, 0

res[0, 2, 5, 5, 0]
```

Code:

```
public class Solution {
    public int[] getModifiedArray(int length, int[][] updates) {

        int[] res = new int[length];
         for(int[] update : updates) {
            int value = update[2];
            int start = update[0];
            int end = update[1];

            res[start] += value;

            if(end < length - 1)
                res[end + 1] -= value;

        }

        int sum = 0;
        for(int i = 0; i < length; i++) {
            sum += res[i];
            res[i] = sum;
        }

        return res;
    }
}
```

Search a 2D Matrix

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

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

Consider the following matrix:

```
[
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

Given target = 3, return true.

Tips:

把2D坐标转化为1D然后用binary search做即可。

Code：

```
public class Solution {
    /**
     * @param matrix, a list of lists of integers
     * @param target, an integer
     * @return a boolean, indicate whether matrix contains targe
t
     */
    public boolean searchMatrix(int[][] matrix, int target) {
        // write your code here
        if (matrix == null || matrix.length == 0){
            return false;
        }
        if (matrix[0] == null || matrix[0].length == 0){
            return false;
        }
        int row = matrix.length;
        int column = matrix[0].length;
        int start = 0;
        int end = row * column -1;

        while(start + 1 < end){
            int mid = start + (end - start)/2;
            int first = mid / column;
            int second = mid % column;
            if (matrix[first][second] == target){
                return true;
            }
            else if (matrix[first][second] < target){
                start = mid;
            }
            else{
                end = mid;
            }
        }

        if (matrix[0][0] == target){
            return true;
        }
        if (matrix[row-1][column-1] == target){
            return true;
        }
        else{
            return false;
        }
    }
}
```

Search a 2D Matrix II

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.

Tips:

复杂度要求——**O(m+n)** time and **O(1) extra space**，同时输入只满足自顶向下和自左向右的升序，行与行之间不再有递增关系，与上题有较大区别。

从右上角开始搜索，由于左边的元素一定不大于当前元素，而下面的元素一定不小于当前元素，因此每次比较时均可排除一列或者一行元素（大于当前元素则排除当前行，小于当前元素则排除当前列，由矩阵特点可知）.

另附求数量版本的Code。

Code:

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0) {
            return false;
        }
        if (matrix[0] == null || matrix[0].length == 0) {
            return 0;
        }
        int m = matrix.length;
        int n = matrix[0].length;
        int row = 0, col = n - 1;
        while (row < m && col >=0) {
            if (matrix[row][col] == target) {
                return true;
            } else if (matrix[row][col] < target) {
                row++;
            } else {
                col--;
            }
        }
        return false;
    }
}
```

求数量版本：

```
public class Solution {
    /**
     * @param matrix: A list of lists of integers
     * @param: A number you want to search in the matrix
     * @return: An integer indicate the occurrence of target in
the given matrix
     */
    public int searchMatrix(int[][] matrix, int target) {
        // write your code here
        if (matrix == null || matrix.length == 0) {
            return 0;
        }
        if (matrix[0] == null || matrix[0].length == 0) {
            return 0;
        }
        int n = matrix.length;
        int m = matrix[0].length;
        int count = 0;
        int x = n - 1;
        int y = 0;
        while (x >= 0 && y < m) {
            if (matrix[x][y] < target) {
                y++;
            } else if (matrix[x][y] > target) {
                x--;
            } else {
                count++;
                x--;
                y++;
            }
        }
        return count;
    }
}
```

Summary Ranges

Given a sorted integer array without duplicates, return the summary of its ranges.

For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

Tips:

遍历一遍就好啦， O(n)

Code:

```
public class Solution {
    public List<String> summaryRanges(int[] nums) {
    ArrayList<String> list=new ArrayList();
    if(nums.length==1){
        list.add(nums[0]+"");
        return list;
    }
    for(int i=0;i<nums.length;i++){
        int a=nums[i];
        while(i+1<nums.length&&(nums[i+1]-nums[i])==1){
            i++;
        }
        if(a!=nums[i]){
            list.add(a+"->"+nums[i]);
        }else{
            list.add(a+"");
        }
    }
    return list;
    }
}
```

3Sum Smaller

Given an array of n integers nums and a target, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition nums[i] + nums[j] + nums[k] < target.

For example, given nums = [-2, 0, 1, 3], and target = 2.

Return 2. Because there are two triplets which sums are less than 2:

```
[-2, 0, 1]
[-2, 0, 3]
```

Follow up: Could you solve it in O(n2) runtime?

Tips:

O(n^2). 2 points. 先固定一个数i, 然后设left和right搜，如果满足条件则说明left 和 [left + 1, right]之间的任意数都满足，left++再看； 如果不满足则right--再看。

Code：

```
public class Solution {
    int count;

    public int threeSumSmaller(int[] nums, int target) {
        count = 0;
        Arrays.sort(nums);
        int len = nums.length;

        for(int i=0; i<len-2; i++) {
            int left = i+1, right = len-1;
            while(left < right) {
                if(nums[i] + nums[left] + nums[right] < target)
{
                    count += right-left;
                    left++;
                } else {
                    right--;
                }
            }
        }

        return count;
    }
}
```

Missing Ranges

Given a sorted integer array where the range of elements are [lower, upper] inclusive, return its missing ranges.

For example, given [0, 1, 3, 50, 75], lower = 0 and upper = 99, return ["2", "4->49", "51->74", "76->99"].

Tips:

增加pre和after来判断！

Code：

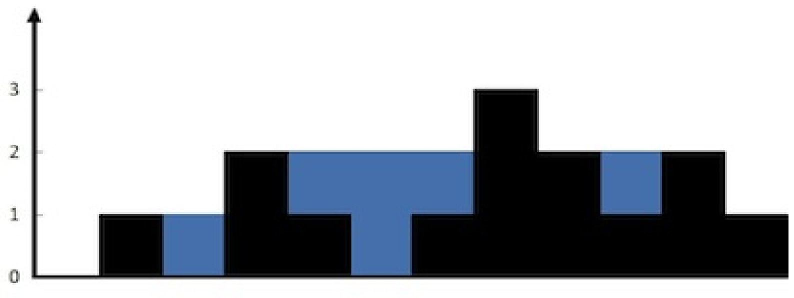
```
public class Solution {
    public List<String> findMissingRanges(int[] A, int lower, in
t upper) {
        List<String> result = new ArrayList<String>();
        int pre = lower - 1;
        for(int i = 0 ; i <= A.length  ; i++){
            int after = i == A.length ? upper + 1 : A[i];
            if(pre + 2 == after){
                result.add(String.valueOf(pre + 1));
            }else if(pre + 2 < after){
                result.add(String.valueOf(pre + 1) + "->" + Stri
ng.valueOf(after - 1));
            }
            pre = after;
        }
        return result;
    }
}
```

Trapping Rain Water

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.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Tips:

I calculated the stored water at each index a and b in my code. At the start of every loop, I update the current maximum height from left side (that is from A[0,1...a]) and the maximum height from right side(from A[b,b+1...n-1]). if(leftmaxleftmax). On the other side, we cannot store more water than (leftmax-A[a]) at index a since the barrier at left is of height leftmax. So, we know the water that can be stored at index a is exactly (leftmax-A[a]). The same logic applies to the case when (leftmax>rightmax). At each loop we can make a and b one step closer. Thus we can finish it in linear time.

Code:

```
public class Solution {
    public int trap(int[] A){
        int a=0;
        int b=A.length-1;
        int max=0;
        int leftmax=0;
        int rightmax=0;
        while(a<=b){
            leftmax=Math.max(leftmax,A[a]);
            rightmax=Math.max(rightmax,A[b]);
            if(leftmax<rightmax){
                max+=(leftmax-A[a]);       // leftmax is smaller
 than rightmax, so the (leftmax-A[a]) water can be stored
                a++;
            }
            else{
                max+=(rightmax-A[b]);
                b--;
            }
        }
        return max;
    }
}
```

Trapping Rain Water II

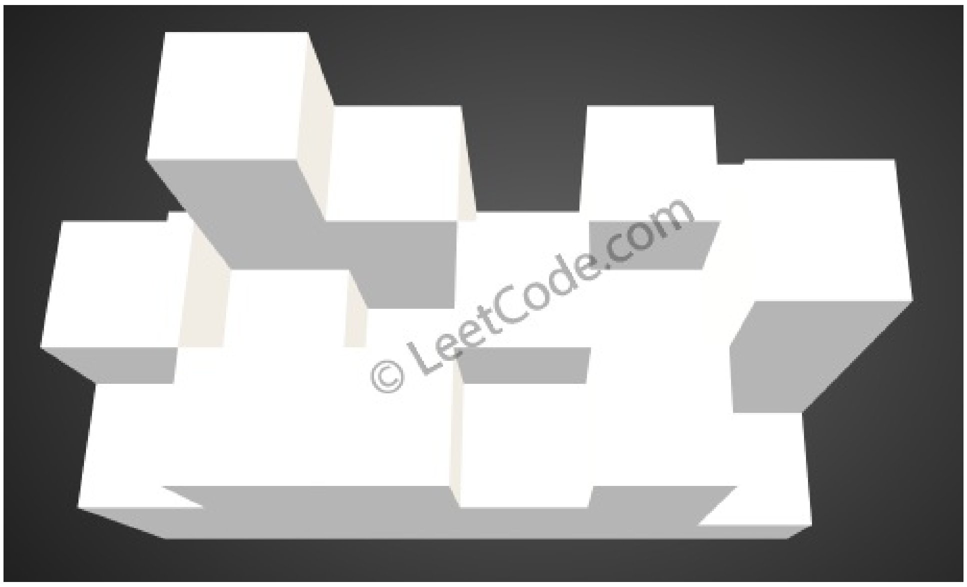
Given an m x n matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining.

Note:

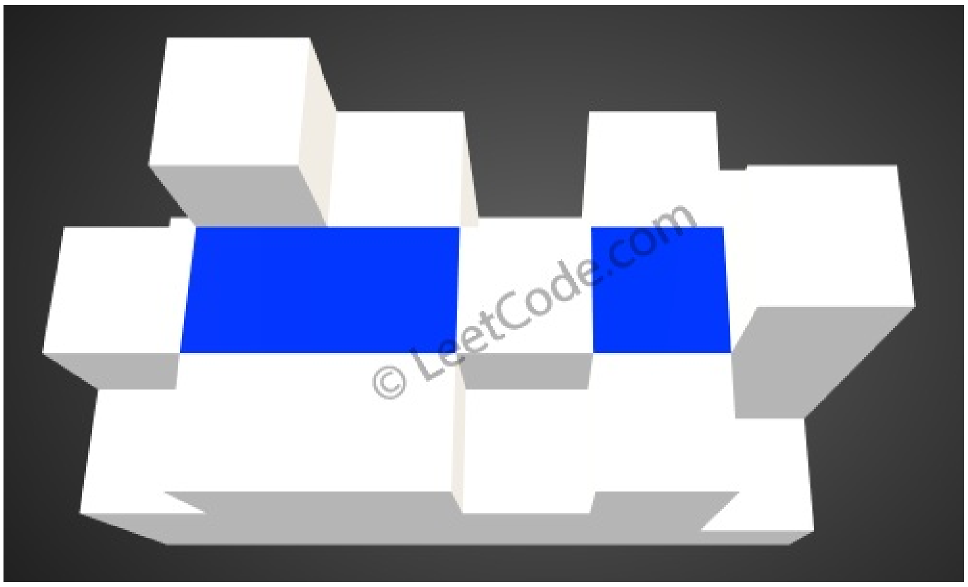
Both m and n are less than 110. The height of each unit cell is greater than 0 and is less than 20,000.

Example:

```
Given the following 3x6 height map:
[
  [1,4,3,1,3,2],
  [3,2,1,3,2,4],
  [2,3,3,2,3,1]
]

Return 4.
```



The above image represents the elevation map [[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]] before the rain.



After the rain, water are trapped between the blocks. The total volume of water trapped is 4.

Tips:

O(nmlog(n+m)) 建堆：**O((2n+2m)log(2(n+m)))**较少，可忽略

用PriorityQueue把选中的height排序。为走位，create class Cell {x,y, height}.

注意几个理论：

1. 从matrix四周开始考虑，发现matrix能Hold住的水，取决于height低的block。
2. 必须从外围开始考虑，因为水是被包裹在里面，外面至少需要现有一层。

以上两点就促使我们用min-heap: 也就是natural order的PriorityQueue.

process的时候，画个图也可以搞清楚，就是四个方向都走走，用curr cell的高度减去周围cell的高度。 若大于零，那么就有积水。

每个visited的cell都要mark. 去到4个方向的cell,加进queue里面继续process.

这里，有一点，和trapping water I 想法一样。刚刚从外围，只是能加到跟外围cell高度一致的水平面。往里面，很可能cell高度变化。

这里要附上curr cell 和 move-to cell的最大高度。

1. Same idea as the trap Rain Water I.
2. Since this is not 1-way run through a 1D array (2D array can go 4 directions...), need to mark visted spot.
3. Use PriorityQueue, sort lowest on top, because the lowest surroundings

determines the best we can get.

4. Bukkit theory: the lowest bar determines the height of the bukkit water. So, we always process the lowest first.
5. Therefore, we use a min-heap, a natural order priorityqueue based on height.
6. Note: when adding a new block into the queue, comparing with the checked origin, we still want to add the higher height into queue. (The high bar will always exist and hold the bukkit.)

   Step:

   i. Create Cell (x,y,h)
   ii. Priorityqueue on Cell of all 4 borders
   iii. Process each element in queue, and add surrounding blocks into queue.
   iv. Mark checked block

Code:

```
public class Solution {

    public class Cell {
        int row;
        int col;
        int height;
        public Cell(int row, int col, int height) {
            this.row = row;
            this.col = col;
            this.height = height;
        }
    }

    public int trapRainWater(int[][] heights) {
        if (heights == null || heights.length == 0 || heights[0]
.length == 0)
            return 0;

        PriorityQueue<Cell> queue = new PriorityQueue<>(1, new C
omparator<Cell>(){
            public int compare(Cell a, Cell b) {
                return a.height - b.height;
            }
        });

        int m = heights.length;
        int n = heights[0].length;
        boolean[][] visited = new boolean[m][n];

        // Initially, add all the Cells which are on borders to
the queue.
        for (int i = 0; i < m; i++) {
            visited[i][0] = true;
            visited[i][n - 1] = true;
            queue.offer(new Cell(i, 0, heights[i][0]));
            queue.offer(new Cell(i, n - 1, heights[i][n - 1]));
        }

        for (int i = 0; i < n; i++) {
            visited[0][i] = true;
            visited[m - 1][i] = true;
            queue.offer(new Cell(0, i, heights[0][i]));
            queue.offer(new Cell(m - 1, i, heights[m - 1][i]));
        }

        // from the borders, pick the shortest cell visited and
check its neighbors:
        // if the neighbor is shorter, collect the water it can
trap and update its height as its height plus the water trapped
       // add all its neighbors to the queue.
        int[][] dirs = new int[][]{{-1, 0}, {1, 0}, {0, -1}, {0,
 1}};
        int res = 0;
        while (!queue.isEmpty()) {
            Cell cell = queue.poll();
            for (int[] dir : dirs) {
                int row = cell.row + dir[0];
                int col = cell.col + dir[1];
                if (row >= 0 && row < m && col >= 0 && col < n &
& !visited[row][col]) {
                    visited[row][col] = true;
                    res += Math.max(0, cell.height - heights[row
][col]);
```

```
                    queue.offer(new Cell(row, col, Math.max(heig
hts[row][col], cell.height)));
                }
            }
        }

        return res;
    }
}
```

Longest Consecutive Sequence

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.

Tips:

O(n)

1. first step is to use a hashset to store all numbers and remove redundant numbers.
2. Then iterate through the array, for each number in the array, go up and down to find its consecutive numbers, delete them in the set if found.

后面附了UnionFind

Code:

HashSet:

```
public class Solution {
    public int longestConsecutive(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        Set<Integer> set = new HashSet<>();
        for (int i = 0; i < nums.length; i++) {
            set.add(nums[i]);
        }
        int result = -1;
        for (int i = 0; i < nums.length; i++) {
            int current = nums[i];
            if (!set.contains(current)) {
                continue;
            }
            set.remove(current);
            int down = current;
            while (set.contains(current - 1)) {
                down--;
                set.remove(current - 1);
                current -= 1;
            }
            current = nums[i];
            int up = current;
            while (set.contains(current + 1)) {
                up++;
                set.remove(current + 1);
                current += 1;
            }
            result = Math.max(result, up - down + 1);
        }
        return result;
    }
}
```

UnionFind:

```
public class Solution {
        public int longestConsecutive(int[] nums) {
            UF uf = new UF(nums.length);
            Map<Integer,Integer> map = new HashMap<Integer,Integ
er>(); // <value,index>
            for(int i=0; i<nums.length; i++){
                if(map.containsKey(nums[i])){
                    continue;
                }
                map.put(nums[i],i);
                if(map.containsKey(nums[i]+1)){
                    uf.union(i,map.get(nums[i]+1));
                }
                if(map.containsKey(nums[i]-1)){
                    uf.union(i,map.get(nums[i]-1));
                }
            }
            return uf.maxUnion();
        }
    }

    class UF{
        private int[] list;
        public UF(int n){
            list = new int[n];
            for(int i=0; i<n; i++){
                list[i] = i;
            }
        }

        private int root(int i){
            while(i!=list[i]){
                list[i] = list[list[i]];
                i = list[i];
            }
            return i;
        }

        public boolean connected(int i, int j){
            return root(i) == root(j);
        }
        public void union(int p, int q){
          int i = root(p);
          int j = root(q);
          list[i] = j;
        }

        // returns the maxium size of union
        public int maxUnion(){ // O(n)
            int[] count = new int[list.length];
            int max = 0;
            for(int i=0; i<list.length; i++){
                count[root(i)] ++;
                max = Math.max(max, count[root(i)]);
            }
            return max;
        }
    }
```

Median of Two Sorted Arrays

There are two sorted arrays nums1 and nums2 of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Example 1:

```
nums1 = [1, 3]
nums2 = [2]

The median is 2.0
```

Example 2:

```
nums1 = [1, 2]
nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5
```

Tips:

要用二分法 **O(log (m+n))**

Code：

```
class Solution {
    /**
     * @param A: An integer array.
     * @param B: An integer array.
     * @return: a double whose format is *.5 or *.0
     */
    public double findMedianSortedArrays(int[] A, int[] B) {
        // write your code here
        int len = A.length + B.length;
        if (len % 2 == 1){
            return findMedian(A, 0, B, 0, len / 2 + 1);
        } else {
            return (findMedian(A, 0, B, 0, len/2) + findMedian(A
, 0, B, 0, len / 2 + 1)) / 2.0;
        }
    }
    private static int findMedian(int[] A, int A_start, int[] B,
 int B_start, int k){
        if (A_start >= A.length){
            return B[B_start + k -1];
        }
        if (B_start >= B.length){
            return A[A_start + k -1];
        }
        if (k == 1){
            return Math.min(A[A_start], B[B_start]);
        }
        int A_key = A_start + k / 2 -1 < A.length ? A[A_start +
k / 2 -1] : Integer.MAX_VALUE;
        int B_key = B_start + k / 2 -1 < B.length ? B[B_start +
k / 2 -1] : Integer.MAX_VALUE;
        if (A_key < B_key){
            return findMedian(A, A_start + k/2, B, B_start, k -
k/2);
        } else {
            return findMedian(A, A_start, B, B_start + k/2, k -
k/2);
        }

    }
}
```

Target Sum

You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol.

Find out how many ways to assign symbols to make sum of integers equal to target S.

Example 1:

```
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3
```

There are 5 ways to assign symbols to make the sum of nums be target 3.

Note:

1. The length of the given array is positive and will not exceed 20.
2. The sum of elements in the given array will not exceed 1000.
3. Your output answer is guaranteed to be fitted in a 32-bit integer.

Tips:

用简单的DFS可以做，类似Expression Add Operators，可以进行优化，用一个sum数组记录后面数的和，如果这个和小于target - 当前和的绝对值，则可以return。复杂度是**O(2^n)**.

也可以用DP，非常快。（没看）

The original problem statement is equivalent to: Find a subset of nums that need to be positive, and the rest of them negative, such that the sum is equal to target

Let P be the positive subset and N be the negative subset For example:

```
Given nums = [1, 2, 3, 4, 5] and target = 3◦
Then one possible solution is +1-2+3-4+5 = 3
Here positive subset is P = [1, 3, 5] and negative subset is N =
 [2, 4]

Then let's see how this can be converted to a subset sum problem
:

                  sum(P) - sum(N) = target
sum(P) + sum(N) + sum(P) - sum(N) = target + sum(P) + sum(N)
                       2 * sum(P) = target + sum(nums)
```

So the original problem has been converted to a subset sum problem as follows: Find a subset P of nums such that sum(P) = (target + sum(nums)) / 2

Note that the above formula has proved that target + sum(nums) must be even

Codes:

DFS:

```
public class Solution {
    int res = 0;
    public int findTargetSumWays(int[] nums, int S) {
        if (nums == null || nums.length == 0) return 0;
        int[] sums = new int[nums.length];
        sums[nums.length - 1] = nums[nums.length - 1];
        for (int i = nums.length - 2; i >= 0; i--) {
            sums[i] = sums[i + 1] + nums[i];
        }
        helper(nums, S, 0, 0, sums);
        return res;
    }

    private void helper(int[] nums, int S, int pos, long eval, i
nt[] sums) {
        if (pos == nums.length) {
            if (eval == S) res++;
            return;
        }
        if (sums[pos] < Math.abs(S - eval)) return;
        helper(nums, S, pos + 1, eval + nums[pos], sums);
        helper(nums, S, pos + 1, eval - nums[pos], sums);
    }
}
```

DP:

```
public int findTargetSumWays(int[] nums, int s) {
    int sum = 0;
    for (int n : nums)
        sum += n;
    return sum < s || (s + sum) % 2 > 0 ? 0 : subsetSum(nums, (s
 + sum) >>> 1);
}

public int subsetSum(int[] nums, int s) {
    int[] dp = new int[s + 1];
    dp[0] = 1;
    for (int n : nums)
        for (int i = s; i >= n; i--)
            dp[i] += dp[i - n];
    return dp[s];
}
```

Max Consecutive Ones

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

Example 1:

```
Input: [1,1,0,1,1,1]
Output: 3
Explanation: The first two digits or the last three digits are c
onsecutive 1s.
    The maximum number of consecutive 1s is 3.
```

Note:

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

Tips:

别忘了最后也要比较一次local_max和res。

Code：

```
public class Solution {
    public int findMaxConsecutiveOnes(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        int res = 0, temp = 0;
        for (int i : nums) {
            if (i == 1) {
                temp++;
                continue;
            } else {
                res = Math.max(res, temp);
                temp = 0;
            }
        }
        res = Math.max(res, temp);
        return res;
    }
}
```

Max Consecutive Ones II

Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.

Example 1:

```
Input: [1,0,1,1,0]
Output: 4
Explanation: Flip the first zero will get the the maximum number
 of consecutive 1s.
    After flipping, the maximum number of consecutive 1s is 4.
```

Note:

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

Follow up: What if the input numbers come in one by one as an infinite stream? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

Tips:

Time： **O(n)**

用2pointers走两遍就可以，left和right，先走right，当0的个数大于两个的时候把left移到第一个0后一位，算出结果，然后接着走。 follow up部分用queue或者arraylist记住每个0的位置即可。

Code:

Time: O(n) Space: O(1)

```
public class Solution {
    public int findMaxConsecutiveOnes(int[] nums) {
        int max = 0, zeros = 0;
        for (int l = 0, r = 0; r < nums.length; r++) {
            if (nums[r] == 0) zeros++;
            while (zeros == 2) {
                while (nums[l] != 0) l++;
                l++;
                zeros--;
            }
            max = Math.max(max, r - l + 1);
        }
        return max;
    }
}
```

Follow Up: Time: O(n) Space: O(k)

```
public int findMaxConsecutiveOnes(int[] nums) {
    int max = 0, k = 1; // flip at most k zero
    Queue<Integer> zeroIndex = new LinkedList<>();
    for (int l = 0, h = 0; h < nums.length; h++) {
        if (nums[h] == 0)
            zeroIndex.offer(h);
        if (zeroIndex.size() > k)

            l = zeroIndex.poll() + 1;
        max = Math.max(max, h - l + 1);
    }
    return max;
}
```

Heaters

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

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

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

Note:

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

Example 1:

```
Input: [1,2,3],[2]
Output: 1
Explanation: The only heater was placed in the position 2, and i
f we use the radius 1 standard, then all the houses can be warme
d.
```

Example 2:

```
Input: [1,2,3,4],[1,4]
Output: 1
Explanation: The two heater was placed in the position 1 and 4.
We need to use radius 1 standard, then all the houses can be war
med.
```

Tips:

用 **heaters[i]+heaters[i+1]<=house*2** 来判断哪个heater离这个house最近。

No, sorting of houses **O(nlogn)** and heaters **O(mlogm)**, and the main part of this algorithm with two pointer is **O(n + m)**, you can think of each element in houses and heaters are visited once.

如果提前没有排序，则为O(mn),会超时。

Binary Search:

The idea is to leverage decent Arrays.binarySearch() function provided by Java.

1. For each house, find its position between those heaters (thus we need the heaters array to be sorted).
2. Calculate the distances between this house and left heater and right heater, get a MIN value of those two values. Corner cases are there is no left or right heater.
3. Get MAX value among distances in step 2. It's the answer.

Time complexity: **max(O(nlogn), O(mlogn))** - m is the length of houses, n is the length of heaters.

Code:

```
public class Solution {
    public int findRadius(int[] houses, int[] heaters) {
        Arrays.sort(houses);
        Arrays.sort(heaters);

        int i = 0, res = 0;
        for (int house : houses) {
            while (i < heaters.length - 1 && heaters[i] + heater
s[i + 1] <= house * 2) {
                i++;
            }
            res = Math.max(Math.abs(heaters[i] - house), res);
        }
        return res;
    }
}
```

Binary Search:

```
public class Solution {
    public int findRadius(int[] houses, int[] heaters) {
        Arrays.sort(heaters);
        int result = Integer.MIN_VALUE;

        for (int house : houses) {
            int index = Arrays.binarySearch(heaters, house);
            if (index < 0) {
            index = -(index + 1);
            }
            int dist1 = index - 1 >= 0 ? house - heaters[index -
 1] : Integer.MAX_VALUE;
            int dist2 = index < heaters.length ? heaters[index]
- house : Integer.MAX_VALUE;

            result = Math.max(result, Math.min(dist1, dist2));
        }

        return result;
    }
}
```

Diagonal Traverse

Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

Example:

```
Input:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
Output:  [1,2,4,7,5,3,6,8,9]\
```

Note:

The total number of elements of the given matrix will not exceed 10,000.

Tips:

设置一个方向dir,如果碰到四边的壁就转向。注意转向时要先判断是否超出m和n，在判断是否小于0，不然会出错。

O(n)

Code：

```
public class Solution {
    public int[] findDiagonalOrder(int[][] matrix) {
        if (matrix == null || matrix.length == 0) return new int
[0];
        int[] res = new int[matrix.length * matrix[0].length];
        int m = matrix.length, n = matrix[0].length;
        int col = 0, row = 0, d = 0;
        int[][] dir = {{-1, 1},{1, -1}};

        for (int i = 0; i < m * n; i++) {
            res[i] = matrix[row][col];
            row = row + dir[d][0];
            col = col + dir[d][1];
            if (row >= m) {
                row = m - 1;
                col += 2;
                d = 1 - d;
            }
            if (col >= n) {
                col = n - 1;
                row += 2;
                d = 1 - d;
            }
            if (row < 0) {
                row = 0;
                d = 1 - d;
            }
            if (col < 0) {
                col = 0;
                d = 1 - d;
            }
        }
        return res;
    }
}
```

Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers (h, k), where h is the height of the person and k is the number of people in front of this person who have a height greater than or equal to h. Write an algorithm to reconstruct the queue.

Note: The number of people is less than 1,100.

Example

```
Input:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
```

Tips:

1. Pick out tallest group of people and sort them in a subarray (S). Since there's no other groups of people taller than them, therefore each guy's index will be just as same as his k value.
2. For 2nd tallest group (and the rest), insert each one of them into (S) by k value. So on and so forth.

E.g.

```
input: [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
subarray after step 1: [[7,0], [7,1]]
subarray after step 2: [[7,0], [6,1], [7,1]]
```

Code:

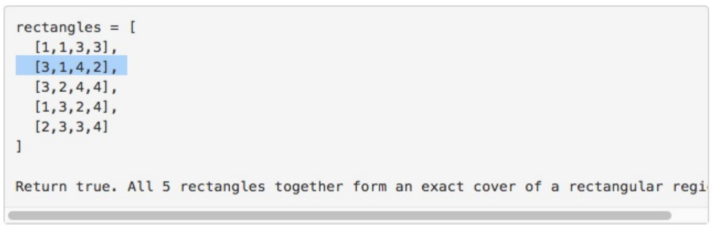
```
public class Solution {
    public int[][] reconstructQueue(int[][] people) {
        Arrays.sort(people, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                if (a[0] != b[0]) return b[0] - a[0];
                else return a[1] - b[1];
            }
        });
        ArrayList<int[]> tmp = new ArrayList<>();
        for (int i = 0; i < people.length; i++) {
            tmp.add(people[i][1], people[i]);
        }
        int[][] res = new int[people.length][2];
        for (int i = 0; i < tmp.size(); i++) {
            res[i][0] = tmp.get(i)[0];
            res[i][1] = tmp.get(i)[1];
        }
        return res;
    }
}
```

Perfect Rectangle

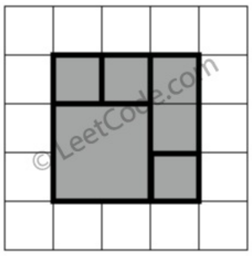
Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region.

Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)).

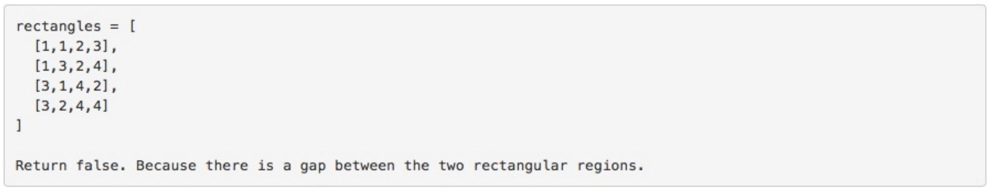
Example 1:



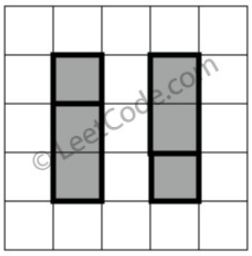
```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [3,2,4,4],
  [1,3,2,4],
  [2,3,3,4]
]

Return true. All 5 rectangles together form an exact cover of a rectangular regi
```



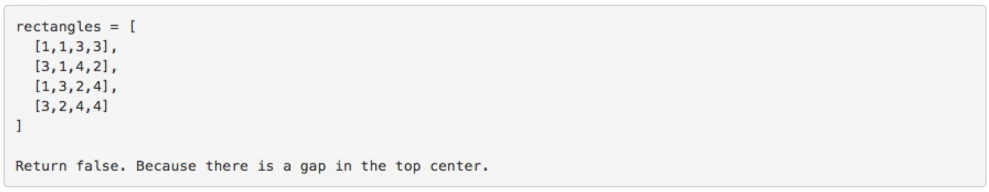
Example 2:



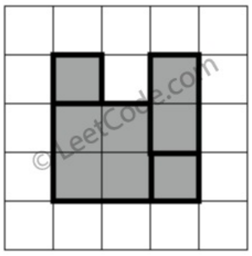
```
rectangles = [
  [1,1,2,3],
  [1,3,2,4],
  [3,1,4,2],
  [3,2,4,4]
]

Return false. Because there is a gap between the two rectangular regions.
```



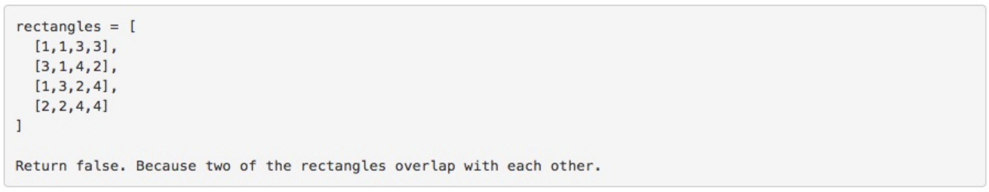
Example 3:



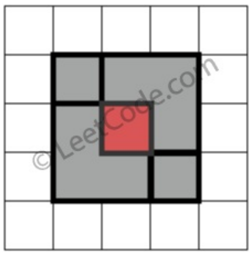
```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [3,2,4,4]
]

Return false. Because there is a gap in the top center.
```



Example 4:



```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [2,2,4,4]
]

Return false. Because two of the rectangles overlap with each other.
```



Tips：

O(n)

两条规则： 1.大的长方形面积必须等于所有小长方形面积之和2. 2.正方形四个角必须只出现过一次

Code：

```
public class Solution {
    public boolean isRectangleCover(int[][] rectangles) {

        if (rectangles.length == 0 || rectangles[0].length == 0)
 return false;

        int x1 = Integer.MAX_VALUE;
        int x2 = Integer.MIN_VALUE;
        int y1 = Integer.MAX_VALUE;
        int y2 = Integer.MIN_VALUE;

        HashSet<String> set = new HashSet<String>();
        int area = 0;

        for (int[] rect : rectangles) {
            x1 = Math.min(rect[0], x1);
            y1 = Math.min(rect[1], y1);
            x2 = Math.max(rect[2], x2);
            y2 = Math.max(rect[3], y2);

            area += (rect[2] - rect[0]) * (rect[3] - rect[1]);

            String s1 = rect[0] + " " + rect[1];
            String s2 = rect[0] + " " + rect[3];
            String s3 = rect[2] + " " + rect[3];
            String s4 = rect[2] + " " + rect[1];

            if (!set.add(s1)) set.remove(s1);
            if (!set.add(s2)) set.remove(s2);
            if (!set.add(s3)) set.remove(s3);
            if (!set.add(s4)) set.remove(s4);
        }

        if (!set.contains(x1 + " " + y1) || !set.contains(x1 + "
 " + y2) || !set.contains(x2 + " " + y1) || !set.contains(x2 + "
 " + y2) || set.size() != 4) return false;

        return area == (x2-x1) * (y2-y1);
        }
    }
```

Optimal Account Balancing

A group of friends went on holiday and sometimes lent each other money. For example, Alice paid for Bill's lunch for $10. Then later Chris gave Alice $5 for a taxi ride. We can model each transaction as a tuple (x, y, z) which means person x gave person y $z. Assuming Alice, Bill, and Chris are person 0, 1, and 2 respectively (0, 1, 2 are the person's ID), the transactions can be represented as [[0, 1, 10], [2, 0, 5]].

Given a list of transactions between a group of people, return the minimum number of transactions required to settle the debt.

Note:

1. A transaction will be given as a tuple (x, y, z). Note that x ≠ y and z > 0.
2. Person's IDs may not be linear, e.g. we could have the persons 0, 1, 2 or we could also have the persons 0, 2, 6.

Example 1:

```
Input:
[[0,1,10], [2,0,5]]

Output:
2

Explanation:
Person #0 gave person #1 $10.
Person #2 gave person #0 $5.

Two transactions are needed. One way to settle the debt is perso
n #1 pays person #0 and #2 $5 each.
```

Example 2:

```
Input:
[[0,1,10], [1,0,1], [1,2,5], [2,0,5]]

Output:
1

Explanation:
Person #0 gave person #1 $10.
Person #1 gave person #0 $1.
Person #1 gave person #2 $5.
Person #2 gave person #0 $5.

Therefore, person #1 only need to give person #0 $4, and all deb
t is settled.
```

Tips:

这道题给了一堆某人欠某人多少钱这样的账单，问我们经过优化后最少还剩几个。其实就相当于一堆人出去玩，某些人可能帮另一些人垫付过花费，最后结算总花费的时候可能你欠着别人的钱，其他人可能也欠你的欠。我们需要找出简单的方法把所有欠账都还清就行了。

这道题的思路跟之前那道Evaluate Division有些像，都需要对一组数据颠倒顺序处理。我们使用一个哈希表来建立每个人和其账户的映射，其中账户若为正数，说明其他人欠你钱；如果账户为负数，说明你欠别人钱。我们对于每份账单，前面的人就在哈希表中减去钱数，后面的人在哈希表中加上钱数。这样我们每个人就都有一个账户了，然后我们接下来要做的就是合并账户，看最少需要多少次汇款。

我们先统计出账户值不为0的人数，因为如果为0了，表明你既不欠别人钱，别人也不欠你钱，如果不为0，我们把钱数放入一个数组accnt中，然后调用递归函数。在递归函数中，我们初始化结果res为整型最大值，然后我们跳过为0的账户，然后我们开始遍历之后的账户，如果当前账户和之前账户的钱数正负不同的话，我们将前一个账户的钱数加到当前账户上，这很好理解，比如前一个账户钱数是-5，表示张三欠了别人五块钱，当前账户钱数是5，表示某人欠了李四五块钱，那么张三给李四五块，这两人的账户就都清零了。然后我们调用递归函数，此时从当前改变过的账户开始找，num表示当前的转账数，需要加1，然后我们用这个递归函数返回的结果来更新res，后面别忘了复原当前账户的值。遍历结束后，我们看res的值如果还是整型的最大值，说明没有改变过，我们返回num，否则返回res即可。

Code:

```
public class Solution {
    public int minTransfers(int[][] transactions) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int[] t : transactions) {
            if (!map.containsKey(t[0])) map.put(t[0], t[2]);
            else map.put(t[0], map.get(t[0]) + t[2]);
            if (!map.containsKey(t[1])) map.put(t[1], -t[2]);
            else map.put(t[1], map.get(t[1]) - t[2]);
        }
        int[] account = new int[map.size()];
        int i = 0;
        for (int p : map.keySet()) {
            if (map.get(p) != 0) account[i++] = (map.get(p));
        }
        return dfs(0, 0, account);
    }
    private int dfs(int index, int num, int[] account) {
        int res = Integer.MAX_VALUE;
        int size = account.length;
        while (index < size && account[index] == 0) index++;
        for (int i = index + 1; i < size; i++) {
            if (account[i] * account[index] < 0) {
                account[i] += account[index];
                res = Math.min(res, dfs(index + 1, num + 1, acco
unt));
                account[i] -= account[index];
            }
        }
        return res == Integer.MAX_VALUE ? num : res;
    }
}
```

Number of Boomerangs

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

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

Example: Input: [[0,0],[1,0],[2,0]]

Output: 2

Explanation: The two boomerangs are [[1,0],[0,0],[2,0]] and [[1,0],[2,0],[0,0]]

Tips:

O(n^2)

不需要存具体的点，只需要知道本轮里到这个点距离相同的点的个数，然后res += val * (val - 1)即可。注意每轮结束后要清空map。

Code：

```
public class Solution {
    public int numberOfBoomerangs(int[][] points) {
        if(points == null || points.length == 0) return 0;
        int res = 0;
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < points.length; i++) {
            for (int j = 0; j < points.length; j++) {
                if (i == j) continue;
                int d = getDistance(points[i], points[j]);
                map.put(d, map.getOrDefault(d, 0) + 1);
            }
            for(int val : map.values()) {
                res += val * (val-1);
            }
            map.clear();
        }
        return res;
    }
    private int getDistance(int[] a, int[] b) {
        int dx = a[0] - b[0];
        int dy = a[1] - b[1];

        return dx*dx + dy*dy;
    }
}
```

LinkedList

Plus One Linked List

Given a non-negative number represented as a singly linked list of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example:

```
Input:
1->2->3

Output:
1->2->4
```

Tips:

Reverse, +1, Reverse

Code:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode plusOne(ListNode head) {
        if (head == null) {
            return head;
        }
        head = reverse(head);
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode result = dummy;
        int carry = 1;
        while (result.next != null) {
            result = result.next;
            int sum = result.val + carry;
            result.val = sum % 10;
            carry = sum / 10;
        }
        //System.out.println(carry);
        if (carry != 0) {
            result.next  = new ListNode(carry);
        }
        return reverse(dummy.next);
    }

        private ListNode reverse(ListNode head) {
        if (head == null) {
            return head;
        }
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        while (head.next != null) {
            ListNode temp = head.next;
            head.next = head.next.next;
            temp.next = dummy.next;
            dummy.next = temp;
        }
        return dummy.next;
    }
}
```

Linked List Random Node

Given a singly linked list, return a random node's value from the linked list. Each node must have the same probability of being chosen.

Follow up:

What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently without using extra space?

Example:

```
// Init a singly linked list [1,2,3].
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
Solution solution = new Solution(head);

// getRandom() should return either 1, 2, or 3 randomly. Each el
ement should have equal probability of returning.
solution.getRandom();
```

Tips:

Reservoir sampling

水塘抽样

由于限定了head一定存在，所以我们先让返回值res等于head的节点值，然后让cur指向head的下一个节点，定义一个变量i，初始化为1，若cur不为空我们开始循环，我们在[0, i]中取一个随机数，如果取出来0，那么我们更新res为当前的cur的节点值，然后此时i自增一，cur指向其下一个位置，这里其实相当于我们维护了一个大小为1的水塘，然后我们随机数生成为0的话，我们交换水塘中的值和当前遍历到底值，这样可以保证每个数字的概率相等

Suppose we see a sequence of items, one at a time. We want to keep a single item in memory, and we want it to be selected at random from the sequence.

If we know the total number of items (n), then the solution is easy: select an index i between 1 and n with equal probability, and keep the i-th element.

The problem is that we do not always know n in advance. A possible solution is the following:

```
* Keep the first item in memory.
* When the i-th item arrives (for i>1):
* with probability 1/i, keep the new item instead of the current
 item; or equivalently
* with probability 1-1/i, keep the current item and discard the
new item.
```

So:

```
* when there is only one item, it is kept with probability 1;
* when there are 2 items, each of them is kept with probability
1/2;
* when there are 3 items, the third item is kept with probabilit
y 1/3,
  and each of the previous 2 items is also kept with probability
 (1/2)(1-1/3) = (1/2)(2/3) = 1/3;
* by induction, it is easy to prove that when there are n items,
 each item is kept with probability 1/n.
```

Code:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */

import java.util.Random;
```

```
public class Solution {
    ListNode head;
    Random random;
    /** @param head The linked list's head.
        Note that the head is guaranteed to be not null, so it c
ontains at least one node. */
    public Solution(ListNode head) {
        this.head = head;
        random = new Random();
    }

    /** Returns a random node's value. */
    public int getRandom() {
        ListNode result = head;
        ListNode cur = head;
        int size = 1;
        while (cur != null) {
            if (random.nextInt(size) == 0) {
                result = cur;
            }
            size++;
            cur = cur.next;
        }
        return result.val;
    }
}

/**
 * Your Solution object will be instantiated and called as such:
 * Solution obj = new Solution(head);
 * int param_1 = obj.getRandom();
 */
```

Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Tips:

建个堆，哪个小往里压

Code:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }
        ListNode result = new ListNode(0);
        ListNode copy = result;
        Queue<ListNode> queue = new PriorityQueue<>(lists.length
, new Comparator<ListNode>() {
            public int compare(ListNode a, ListNode b) {
                if (a == null) {
                    return 1;
                }
                else if (b == null) {
                    return -1;
                }
                else return a.val - b.val;
            }
        });
        for (int i = 0; i < lists.length; i++) {
            if (lists[i] != null) {
                queue.add(lists[i]);
            }
        }
        while (!queue.isEmpty()) {
            ListNode temp = queue.poll();
            result.next = new ListNode(temp.val);
            result = result.next;
            if (temp.next != null) {
                queue.add(temp.next);
            }
        }
        return copy.next;
    }
}
```

Design

Zigzag Iterator

Given two 1d vectors, implement an iterator to return their elements alternately.

For example, given two 1d vectors:

```
v1 = [1, 2]
v2 = [3, 4, 5, 6]
```

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1, 3, 2, 4, 5, 6].

Follow up: What if you are given k 1d vectors? How well can your code be extended to such cases?

Clarification for the follow up question - Update (2015-09-18): The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic". For example, given the following input:

```
[1,2,3]
[4,5,6,7]
[8,9]
```

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

Tips:

- 对iterator的定义和写法不是很清晰，算法倒是比较简洁明了。借鉴了discuss做出来，但是并不是很难。
- 两个数组的合并，穿插进行。
- Follow up 要新建arraylist iterator。

//解法1：

```
public class ZigzagIterator {
    Iterator<Integer> it1;
    Iterator<Integer> it2;
    int turns;

    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        this.it1 = v1.iterator();
        this.it2 = v2.iterator();
        turns = 0;
    }

    public int next() {
        // 如果没有下一个则返回0
        if(!hasNext()){
            return 0;
        }
        turns++;
        // 如果是第奇数个，且第一个列表也有下一个元素时，返回第一个列表的下
一个
        // 如果第二个列表已经没有，返回第一个列表的下一个
        if((turns % 2 == 1 && it1.hasNext()) || (!it2.hasNext())
){
            return it1.next();
        // 如果是第偶数个，且第二个列表也有下一个元素时，返回第二个列表的下
一个
        // 如果第一个列表已经没有，返回第二个列表的下一个
        } else if((turns % 2 == 0 && it2.hasNext()) || (!it1.has
Next())){
            return it2.next();
        }
        return 0;
    }

    public boolean hasNext() {
        return it1.hasNext() || it2.hasNext();
    }
}
```

//解法**2**：

```
public class ZigzagIterator {
int index;
    boolean count;
    List<Integer> v1;
    List<Integer> v2;

    public ZigzagIterator (List<Integer> v1, List<Integer> v2) {
        this.index = 0;
        this.count = true;
        this.v1 = v1;
        this.v2 = v2;
    }

    public int next() {
        int result;
        if (count && index < v1.size()) {
            result = v1.get(index);
            if (index < v2.size()) {
                count = !count;
            } else {
                index++;
            }
        } else {
            result = v2.get(index++);
            if (index < v1.size()) {
                count = !count;
            }
        }
        return result;
    }

    public boolean hasNext() {
        return index < v1.size() || index < v2.size();
    }
}
```

/**

- Your ZigzagIterator object will be instantiated and called as such:

- ZigzagIterator i = new ZigzagIterator(v1, v2);
- while (i.hasNext()) v[f()] = i.next(); */

## follow up:

```
public class ZigzagIterator implements Iterator<Integer> {
    List<Iterator<Integer>> itlist;
    int turns;

    public ZigzagIterator(List<Iterator<Integer>> list) {
        this.itlist = new LinkedList<Iterator<Integer>>();
        // 将非空迭代器加入列表
        for(Iterator<Integer> it : list){
            if(it.hasNext()){
                itlist.add(it);
            }
        }
        turns = 0;
    }

    public Integer next() {
        if(!hasNext()){
            return 0;
        }
        Integer res = 0;
        // 算出本次使用的迭代器的下标
        int pos = turns % itlist.size();
        Iterator<Integer> curr = itlist.get(pos);
        res = curr.next();
        // 如果这个迭代器用完，就将其从列表中移出
        if(!curr.hasNext()){
            itlist.remove(turns % itlist.size());
            // turns变量更新为上一个下标
            turns = pos - 1;
        }
        turns++;
        return res;
    }

    public boolean hasNext() {
        return itlist.size() > 0;
    }
}
```

Zigzag Iterator II

Follow up Zigzag Iterator: What if you are given k 1d vectors? How well can your code be extended to such cases? The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic".

Have you met this question in a real interview? Yes Example Given k = 3 1d vectors:

```
[1,2,3]
[4,5,6,7]
[8,9]
Return [1,4,8,2,5,9,3,6,7].


public class ZigzagIterator2 {

    public List<Iterator<Integer>> its;
    public int turns;

    /**
     * @param vecs a list of 1d vectors
     */
    public ZigzagIterator2(ArrayList<ArrayList<Integer>> vecs) {
        // initialize your data structure here.
        this.its = new ArrayList<Iterator<Integer>>();
        for (List<Integer> vec : vecs) {
            if (vec.size() > 0)
                its.add(vec.iterator());
        }
        turns = 0;
    }

    public int next() {
        // Write your code here
        int elem = its.get(turns).next();
        if (its.get(turns).hasNext())
```

```
            turns = (turns + 1) % its.size();
        else {
            its.remove(turns);
            if (its.size() > 0)
                turns %= its.size();
        }
        return elem;
    }

    public boolean hasNext() {
        // Write your code here
        return its.size() > 0;
    }
}
```

Design Tic-Tac-Toe

Design a Tic-tac-toe game that is played between two players on a n x n grid.

You may assume the following rules:

A move is guaranteed to be valid and is placed on an empty block.

Once a winning condition is reached, no more moves is allowed.

A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

```
Given n = 3, assume that player 1 is "X" and player 2 is "O" in
the board.

TicTacToe toe = new TicTacToe(3);

toe.move(0, 0, 1); -> Returns 0 (no one wins)
|X| | |
| | | |    // Player 1 makes a move at (0, 0).
| | | |

toe.move(0, 2, 2); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 2 makes a move at (0, 2).
| | | |

toe.move(2, 2, 1); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 1 makes a move at (2, 2).
| | |X|

toe.move(1, 1, 2); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 2 makes a move at (1, 1).
| | |X|

toe.move(2, 0, 1); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 1 makes a move at (2, 0).
|X| |X|

toe.move(1, 0, 2); -> Returns 0 (no one wins)
|X| |O|
|O|O| |    // Player 2 makes a move at (1, 0).
|X| |X|

toe.move(2, 1, 1); -> Returns 1 (player 1 wins)
|X| |O|
|O|O| |    // Player 1 makes a move at (2, 1).
|X|X|X|
```

Follow up: Could you do better than O(n2) per move() operation?

Hint:

Could you trade extra space such that move() operation can be done in O(1)? You need two arrays: int rows[n], int cols[n], plus two variables: diagonal, anti_diagonal.

Tips:

O(1) Time, **O(n)** extra space. 把player订成1和-1； 建立rows[n]和cols[n]数组和变量diagonal, anti_diagonal来记录这一行/列/对角线的情况。

当有行/列/对角线/反对角线的绝对值达到size的时候，就返回当前player，不然返回0。

The key observation is that in order to win Tic-Tac-Toe you must have the entire row or column. Thus, we don't need to keep track of an entire n^2 board. We only need to keep a count for each row and column. If at any time a row or column matches the size of the board then that player has won.

To keep track of which player, I add one for Player1 and -1 for Player2. There are two additional variables to keep track of the count of the diagonals. Each time a player places a piece we just need to check the count of that row, column, diagonal and anti-diagonal.

Code:

```
public class TicTacToe {
    private int[] rows;
    private int[] cols;
    private int diagonal;
    private int antiDiagonal;
    /** Initialize your data structure here. */
    public TicTacToe(int n) {
        rows = new int[n];
        cols = new int[n];
    }

    /** Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins. */
    public int move(int row, int col, int player) {
        int toAdd = player == 1 ? 1 : -1;
        rows[row] += toAdd;
        cols[col] += toAdd;
        if (row == col) {
            diagonal += toAdd;
        }
        if (row + col == rows.length + 1) {
            antiDiagonal += toAdd;
        }
        int size = rows.length;
        if (Math.abs(rows[row]) == size ||
            Math.abs(cols[col]) == size ||
            Math.abs(diagonal) == size ||
            Math.abs(antiDiagonal) == size) {
                return player;
            }
        return 0;
    }
}

/**
 * Your TicTacToe object will be instantiated and called as such
:
 * TicTacToe obj = new TicTacToe(n);
 * int param_1 = obj.move(row,col,player);
 */
```

Design Snake Game

Design a Snake game that is played on a device with screen size = width x height. Play the game online if you are not familiar with the game.

The snake is initially positioned at the top left corner (0,0) with length = 1 unit.

You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both increase by 1.

Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake.

When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake.

Example:

```
Given width = 3, height = 2, and food = [[1,2],[0,1]].

Snake snake = new Snake(width, height, food);

Initially the snake appears at position (0,0) and the food at (1
,2).

|S| | |
| | |F|

snake.move("R"); -> Returns 0

| |S| |
| | |F|

snake.move("D"); -> Returns 0

| | | |
| |S|F|

snake.move("R"); -> Returns 1 (Snake eats the first food and rig
ht after that, the second food appears at (0,1) )

| |F| |
| |S|S|

snake.move("U"); -> Returns 1

| |F|S|
| | |S|

snake.move("L"); -> Returns 2 (Snake eats the second food)

| |S|S|
| | |S|

snake.move("U"); -> Returns -1 (Game over because snake collides
 with border)
```

Tips:

看起来难实际上比较简单，重点是用Deque，可以两头加减蛇。加入HashSet可以快速读取蛇有没有咬到自己。先计算移动后新蛇头的位置，判断是否出局，然后看是否吃了食物，吃了则蛇头入队，不减蛇尾，没吃则是普通移动，蛇头入队，减去老蛇尾。

Code：

```
public class SnakeGame {

    /** Initialize your data structure here.
        @param width - screen width
        @param height - screen height
        @param food - A list of food positions
        E.g food = [[1,1], [1,0]] means the first food is positi
oned at [1,1], the second is at [1,0]. */
        HashSet<Integer> set;
        Deque<Integer> body;
        int score;
        int[][] food;
        int foodIndex;
        int width;
        int height;
    public SnakeGame(int width, int height, int[][] food) {
        this.width = width;
        this.height = height;
        this.food = food;
        set = new HashSet<Integer>();
        set.add(0);
        body = new LinkedList<Integer>();
        body.addLast(0);
    }

    /** Moves the snake.
        @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D
' = Down
        @return The game's score after the move. Return -1 if ga
me over.
        Game over when snake crosses the screen boundary or bite
```

```
s its body. */
    public int move(String direction) {
        if (score == -1) {
            return -1;
        }
        //calculate head's position
        int rowHead = body.peekFirst() / width;
        int colHead = body.peekFirst() % width;
        // move
        switch (direction) {
            case "U" : rowHead--;
                       break;
            case "D" : rowHead++;
                       break;
            case "L" : colHead--;
                       break;
            case "R" : colHead++;
                       break;
        }
        set.remove(body.peekLast()); // since the snake moved, i
t is no longer in the old tail's position.
        int headPos = rowHead * width + colHead; // recaluate ne
w head position
        //Game Over
        if (rowHead < 0 || rowHead == height || colHead < 0 || c
olHead == width || set.contains(headPos)) {
            return score = -1;
        }
        //add head
        set.add(headPos);
        body.addFirst(headPos);
        // eating food, keep tail, add head
        if (foodIndex < food.length && rowHead == food[foodIndex
][0] && colHead == food[foodIndex][1]) {
            set.add(body.peekLast()); // old tail does not chang
e, so add it back to set
            foodIndex++;
            return ++score;
        }
        // if didnt eat food, remove old tail,just normal move
        body.removeLast();
        return score;
    }
}

/**
 * Your SnakeGame object will be instantiated and called as such
:
 * SnakeGame obj = new SnakeGame(width, height, food);
 * int param_1 = obj.move(direction);
 */
```

Flatten 2D Vector

Implement an iterator to flatten a 2d vector.

For example, Given 2d vector =

```
[
  [1,2],
  [3],
  [4,5,6]
]
```

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,2,3,4,5,6].

Hint:

How many variables do you need to keep track?

Two variables is all you need. Try with x and y.

Beware of empty rows. It could be the first few rows.

To write correct code, think about the invariant to maintain. What is it?

The invariant is x and y must always point to a valid point in the 2d vector. Should you maintain your invariant ahead of time or right when you need it?

Not sure? Think about how you would implement hasNext(). Which is more complex?

Common logic in two different places should be refactored into a common method.

Tips:

就是判断有没有到这一行的尾部以及有没有到最后。注意边界。

Code：

```
public class Vector2D implements Iterator<Integer> {
    int x;
    int y;
    List<List<Integer>> vec2d;
    public Vector2D(List<List<Integer>> vec2d) {
        y = 0;
        x = 0;
        this.vec2d = vec2d;
    }

    @Override
    public Integer next() {
        return vec2d.get(y).get(x++);
    }

    @Override
    public boolean hasNext() {
        while (y < vec2d.size()) {
            if (x < vec2d.get(y).size()) {
                return true;
            } else {
                y++;
                x = 0;
            }
        }
        return false;
    }
}

/**
 * Your Vector2D object will be instantiated and called as such:
 * Vector2D i = new Vector2D(vec2d);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

Design Hit Counter

Design a hit counter which counts the number of hits received in the past 5 minutes.

Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.

It is possible that several hits arrive roughly at the same time.

Example: HitCounter counter = new HitCounter();

// hit at timestamp 1. counter.hit(1);

// hit at timestamp 2. counter.hit(2);

// hit at timestamp 3. counter.hit(3);

// get hits at timestamp 4, should return 3. counter.getHits(4);

// hit at timestamp 300. counter.hit(300);

// get hits at timestamp 300, should return 4. counter.getHits(300);

// get hits at timestamp 301, should return 3. counter.getHits(301); Follow up: What if the number of hits per second could be very large? Does your design scale?

Tips:

由于Follow up中说每秒中会有很多点击，下面这种方法就比较巧妙了，定义了两个大小为300的一维数组times和hits，分别用来保存时间戳和点击数，在点击函数中，将时间戳对300取余，然后看此位置中之前保存的时间戳和当前的时间戳是否一样，一样说明是同一个时间戳，那么对应的点击数自增1，如果不一样，说明已经过了五分钟了，那么将对应的点击数重置为1。那么在返回点击数时，我们需要遍历times数组，找出所有在5分中内的位置，然后把hits中对应位置的点击数都加起来即可.

Code:

```
public class HitCounter {

    private int[] times;
    private int[] hits;

    /** Initialize your data structure here. */
    public HitCounter() {
        times = new int[300];
        hits = new int[300];
    }

    /** Record a hit.
        @param timestamp - The current timestamp (in seconds gra
nularity). */
    public void hit(int timestamp) {
        int index = timestamp % 300;
        if (times[index] != timestamp) {
            times[index] = timestamp;
            hits[index] = 1;
        } else {
            hits[index]++;
        }
    }

    /** Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds gra
nularity). */
    public int getHits(int timestamp) {
        int total = 0;
        for (int i = 0; i < 300; i++) {
            if (timestamp - times[i] < 300) {
                total += hits[i];
            }
        }
        return total;
    }
}
```

```
/**
 * Your HitCounter object will be instantiated and called as suc
h:
 * HitCounter obj = new HitCounter();
 * obj.hit(timestamp);
 * int param_2 = obj.getHits(timestamp);
 */
```

Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:

Given the list [[1,1],2,[1,1]],

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,1,2,1,1].

Example 2:

Given the list [1,[4,[6]]],

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,4,6].

Tips:

这道题让我们建立压平嵌套链表的迭代器，用栈来做。By calling next repeatedly until hasNext returns false, the order of elements returned by next.

由于栈的后进先出的特性，我们在对向量遍历的时候，从后往前把对象压入栈中，那么第一个对象最后压入栈就会第一个取出来处理，我们的hasNext()函数需要遍历栈，并进行处理，如果栈顶元素是整数，直接返回true，如果不是，那么移除栈顶元素，并开始遍历这个取出的list，还是从后往前压入栈，循环停止条件是栈为空，返回false。

Code:

```
/**
 * // This is the interface that allows for creating nested list
s.
 * // You should not implement it, or speculate about its implem
entation
 * public interface NestedInteger {
```

```
 *
 *     // @return true if this NestedInteger holds a single inte
ger, rather than a nested list.
 *     public boolean isInteger();
 *
 *     // @return the single integer that this NestedInteger hol
ds, if it holds a single integer
 *     // Return null if this NestedInteger holds a nested list
 *     public Integer getInteger();
 *
 *     // @return the nested list that this NestedInteger holds,
 if it holds a nested list
 *     // Return null if this NestedInteger holds a single integ
er
 *     public List<NestedInteger> getList();
 * }
 */
public class NestedIterator implements Iterator<Integer> {
    Stack<NestedInteger> stack = new Stack<>();
    public NestedIterator(List<NestedInteger> nestedList) {
        for (int i = nestedList.size() - 1; i >= 0; i--) {
            stack.push(nestedList.get(i));
        }
    }

    @Override
    public Integer next() {
        this.hasNext();
        return stack.pop().getInteger();
    }

    @Override
    public boolean hasNext() {
        while (!stack.isEmpty()) {
            NestedInteger current = stack.peek();
            if (current.isInteger()) {
                return true;
            }
            stack.pop();
            List<NestedInteger> list = current.getList();
            for (int i = list.size() - 1; i >= 0; i--) {
                stack.push(list.get(i));
            }
        }
        return false;
    }
}

/**
 * Your NestedIterator object will be instantiated and called as
 such:
 * NestedIterator i = new NestedIterator(nestedList);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

Peeking Iterator

Given an Iterator class interface with methods: next() and hasNext(), design and implement a PeekingIterator that support the peek() operation -- it essentially peek() at the element that will be returned by the next call to next().

Here is an example. Assume that the iterator is initialized to the beginning of the list: [1, 2, 3].

Call next() gets you 1, the first element in the list.

Now you call peek() and it returns 2, the next element. Calling next() after that still return 2.

You call next() the final time and it returns 3, the last element. Calling hasNext() after that should return false.

Hint:

```
Think of "looking ahead". You want to cache the next element.
Is one variable sufficient? Why or why not?
Test your design with call order of peek() before next() vs next
() before peek().
For a clean implementation, check out Google's guava library sou
rce code.
```

Follow up: How would you extend your design to be generic and work with all types, not just integer?

Tips:

iterator加peek()功能就是 use one integer to cache the first element。注意类型，不能是int，不然 = null会出错，得是Integer。 Follow up就是变一下Iterator it的T。

Code：

```
// Java Iterator interface reference:
// https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.
html
```

```
class PeekingIterator implements Iterator<Integer> {
    Integer next;
    Iterator<Integer> it;
    public PeekingIterator(Iterator<Integer> iterator) {
        // initialize any member here.
        this.next = null;
        this.it = iterator;
        if (it.hasNext()) {
            next = it.next();
        }
    }

    // Returns the next element in the iteration without advanci
ng the iterator.
    public Integer peek() {
        return next;
    }

    // hasNext() and next() should behave the same as in the Ite
rator interface.
    // Override them if needed.
    @Override
    public Integer next() {
        int res = next;
        if (it.hasNext()) {
            next = it.next();
        } else {
            next = null;
        }
        return res;
    }

    @Override
    public boolean hasNext() {
        return next != null;
    }
}
```

Follow up:

```
class PeekingIterator<T> implements Iterator<T> {
    T store;
    Iterator<T> it;
    public PeekingIterator(Iterator<T> iterator) {
        // initialize any member here.
        this.it = iterator;
        this.store = null;
        if (it.hasNext()) {
            store = it.next();
        }
    }

    // Returns the next element in the iteration without advanci
ng the iterator.
    public T peek() {
        if (store != null) {
            return store;
        }
        return null;
    }
    @Override
    public T next() {
        if (store == null) {
            return null;
        }
        T result = store;
        if (it.hasNext()) {
            store = it.next();
        }
        else store = null;
        return result;
    }
    @Override
    public boolean hasNext() {
        return store != null;
    }
}
```

Encode and Decode Strings

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {
  // ... your code
  return encoded_string;
}
```

Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {
  //... your code
  return strs;
}
```

So Machine 1 does:

```
string encoded_string = encode(strs);
```

and Machine 2 does:

```
vector<string> strs2 = decode(encoded_string);
```

strs2 in Machine 2 should be the same as strs in Machine 1.

Implement the encode and decode methods.

Note: The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters.

Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

Do not rely on any library method such as eval or serialize methods. You should implement your own encode/decode algorithm.

Tips:

用slash隔开就好啦,格式是

The encoded form of strings is length + '/' +str + length + '/' + str.

If you say, there is a '/' in a str. Then it is counted for the length of that str but the added slash is not.

For "ab/cd", the encoded one should be "5/ab/cd". The decode function will read the length first, then skip the slash between '5' and 'a', starting from the next character and get the substring of that length which is "ab/cd".

Code:

```
public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder sb = new StringBuilder();
        for(String s : strs) {
            sb.append(s.length()).append('/').append(s);
        }
        return sb.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> res = new ArrayList<String>();
        int i = 0;
        while(i < s.length()) {
            int slash = s.indexOf('/', i);
            int size = Integer.valueOf(s.substring(i, slash));
            res.add(s.substring(slash + 1, slash + size + 1));
            i = slash + size + 1;
        }
        return res;
    }
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.decode(codec.encode(strs));
```

Streams:

```
// Encodes a list of strings to a single string.
public String encode(List<String> strs) {
    return strs.stream()
               .map(s -> s.replace("/", "//").replace("*", "/*")
 + "*")
               .collect(Collectors.joining());
}

// Decodes a single string to a list of strings.
public List<String> decode(String s) {
    List<String> res = new ArrayList<>();
    StringBuilder str = new StringBuilder();

    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == '/') {
            str.append(s.charAt(++i));
        } else if (s.charAt(i) == '*') {
            res.add(str.toString());
            str.setLength(0);
        } else {
            str.append(s.charAt(i));
        }
    }

    return res;
}
```

Design Phone Directory

Design a Phone Directory which supports the following operations:

1. get: Provide a number which is not assigned to anyone.
2. check: Check if a number is available or not.
3. release: Recycle or release a number. Example:

   // Init a phone directory containing a total of 3 numbers: 0, 1, and 2. PhoneDirectory directory = new PhoneDirectory(3);

   // It can return any available phone number. Here we assume it returns 0. directory.get();

   // Assume it returns 1. directory.get();

   // The number 2 is available, so return true. directory.check(2);

   // It returns 2, the only number that is left. directory.get();

   // The number 2 is no longer available, so return false. directory.check(2);

   // Release number 2 back to the pool. directory.release(2);

   // Number 2 is available again, return true. directory.check(2);

Tips:

1.在get()是O(n)，check O(1) release O(1)的方法里没有什么破绽

2.在全是是O(1)的方法里

For method check, what if the number passed in is larger than or equal to the maxNumbers

In that case, release function works properly, but check function does not. Boundary check is needed in release function.

Code：

复杂度高：

```
public class PhoneDirectory {

    /** Initialize your data structure here
        @param maxNumbers - The maximum numbers that can be stor
ed in the phone directory. */
        boolean[] bitSet;
        int smallestFreeIndex;
    public PhoneDirectory(int maxNumbers) {
        this.bitSet = new boolean[maxNumbers];
    }

    /** Provide a number which is not assigned to anyone.
        @return - Return an available number. Return -1 if none
is available. */
    public int get() {
        if (smallestFreeIndex == bitSet.length) {
            return -1;
        }
        int num = smallestFreeIndex;
        bitSet[num] = true;
        for (int i = smallestFreeIndex + 1; i < bitSet.length; i
++) {
            if (!bitSet[i]) {
                smallestFreeIndex = i;
                break;
            }
        }
        if (num == smallestFreeIndex) {
            smallestFreeIndex = bitSet.length;
        }
        return num;
    }

    /** Check if a number is available or not. */
    public boolean check(int number) {
        return !bitSet[number];
    }

    /** Recycle or release a number. */
    public void release(int number) {
        if (bitSet[number] = false) {
            return;
        }
        bitSet[number] = false;
        if (number < smallestFreeIndex) {
            smallestFreeIndex = number;
        }
    }
}

/**
 * Your PhoneDirectory object will be instantiated and called as
 such:
 * PhoneDirectory obj = new PhoneDirectory(maxNumbers);
 * int param_1 = obj.get();
 * boolean param_2 = obj.check(number);
 * obj.release(number);
 */
```

复杂度低：

```
Set<Integer> used = new HashSet<Integer>();
Queue<Integer> available = new LinkedList<Integer>();
int max;
public PhoneDirectory(int maxNumbers) {
    max = maxNumbers;
    for (int i = 0; i < maxNumbers; i++) {
        available.offer(i);
    }
}

public int get() {
    Integer ret = available.poll();
    if (ret == null) {
        return -1;
    }
    used.add(ret);
    return ret;
}

public boolean check(int number) {
    if (number >= max || number < 0) {
        return false;
    }
    return !used.contains(number);
}

public void release(int number) {
    if (used.remove(number)) {
        available.offer(number);
    }
}
```

Insert Delete GetRandom O(1)

Design a data structure that supports all following operations in average O(1) time.

insert(val): Inserts an item val to the set if not already present.

remove(val): Removes an item val from the set if present.

getRandom: Returns a random element from current set of elements. Each element must have the same probability of being returned.

Example:

```
// Init an empty set.
RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successf
ully.
randomSet.insert(1);

// Returns false as 2 does not exist in the set.
randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].
randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.
randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].
randomSet.remove(1);

// 2 was already in the set, so return false.
randomSet.insert(2);

// Since 1 is the only number in the set, getRandom always retur
n 1.
randomSet.getRandom();
```

Tips:

for O(1) insert and remove we have to use something utilizing hashcode, like hashmap or hashset.

And for constant time getRandom, we will need something that has random access like array or arraylist.

We can combine them together, using a hashmap to store a map from value to their index in the arraylist.

Code:

```
public class RandomizedSet {
    Map<Integer, Integer> map;
    List<Integer> vals;
    /** Initialize your data structure here. */
    public RandomizedSet() {
        map = new HashMap<>();
        vals = new  ArrayList<>();
    }

    /** Inserts a value to the set. Returns true if the set did
not already contain the specified element. */
    public boolean insert(int val) {
        if (map.containsKey(val)) {
            return false;
        }
        map.put(val, vals.size());
        vals.add(val);
        return true;
    }

    /** Removes a value from the set. Returns true if the set co
ntained the specified element. */
    public boolean remove(int val) {
        if (!map.containsKey(val)) {
            return false;
        }
        int last = vals.get(vals.size() - 1);
        vals.set(map.get(val), last);
        map.put(last, map.get(val));
        map.remove(val);
        vals.remove(vals.size() - 1);
        return true;
    }

    /** Get a random element from the set. */
    public int getRandom() {
        Random rand = new Random();
        int r = rand.nextInt(vals.size());
        return vals.get(r);
    }
}
```

```
/**
 * Your RandomizedSet object will be instantiated and called as
such:
 * RandomizedSet obj = new RandomizedSet();
 * boolean param_1 = obj.insert(val);
 * boolean param_2 = obj.remove(val);
 * int param_3 = obj.getRandom();
 */
```

Logger Rate Limiter

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is not printed in the last 10 seconds.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

```
Logger logger = new Logger();

// logging string "foo" at timestamp 1
logger.shouldPrintMessage(1, "foo"); returns true;

// logging string "bar" at timestamp 2
logger.shouldPrintMessage(2,"bar"); returns true;

// logging string "foo" at timestamp 3
logger.shouldPrintMessage(3,"foo"); returns false;

// logging string "bar" at timestamp 8
logger.shouldPrintMessage(8,"bar"); returns false;

// logging string "foo" at timestamp 10
logger.shouldPrintMessage(10,"foo"); returns false;

// logging string "foo" at timestamp 11
logger.shouldPrintMessage(11,"foo"); returns true;
```

Tips:

来个map结束了

Code：

```
public class Logger {

    /** Initialize your data structure here. */
    private Map<String, Integer> map = new HashMap<>();
    public Logger() {
        map=new HashMap<>();
    }

    /** Returns true if the message should be printed in the giv
en timestamp, otherwise returns false.
        If this method returns false, the message will not be pr
inted.
        The timestamp is in seconds granularity. */
    public boolean shouldPrintMessage(int timestamp, String mess
age) {
        if(!map.containsKey(message) || timestamp - map.get(mess
age) >= 10){
            map.put(message,timestamp);
            return true;
        }
        return false;
    }
}

/**
 * Your Logger object will be instantiated and called as such:
 * Logger obj = new Logger();
 * boolean param_1 = obj.shouldPrintMessage(timestamp,message);
 */
```

Moving Average from Data Stream

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

For example,

```
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3
```

Tips:

The idea is to keep the sum so far and update the sum just by replacing the oldest number with the new entry.

O(1)

Code:

```
public class MovingAverage {
    private int [] window;
    private int n, insert;
    private long sum;

    /** Initialize your data structure here. */
    public MovingAverage(int size) {
        window = new int[size];
        insert = 0;
        sum = 0;
    }

    public double next(int val) {
        if (n < window.length)  n++;
        sum -= window[insert];
        sum += val;
        window[insert] = val;
        insert = (insert + 1) % window.length;

        return (double)sum / n;
    }
}

/**
 * Your MovingAverage object will be instantiated and called as
such:
 * MovingAverage obj = new MovingAverage(size);
 * double param_1 = obj.next(val);
 */
```

String

Given an encoded string, return it's decoded string.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there won't be input like 3a or 2[4].

Examples:

```
s = "3[a]2[bc]", return "aaabcbc".
s = "3[a2[c]]", return "accaccacc".
s = "2[abc]3[cd]ef", return "abcabccdcdcdef".
```

Tips:

先建立count和result栈，注意result要先入栈"";注意每次result要push的情况下可能要 result.push(result.pop() + 当前要入栈的元素)

```
流程：
1.遇到数字换算一下存入count栈；
2.遇到[，result入栈“”；
3.遇到字母，result入栈字母；result.push(result.pop() + s.charAt(i))
；
4.遇到]，将result.pop()入栈count.pop()次；
5.return result.pop()。
```

Code:

```
public class Solution {
    public String decodeString(String s) {
        Stack<Integer> count = new Stack<>();
        Stack<String> result = new Stack<>();
        result.push("");
        int i = 0;
        while (i < s.length()) {
           if (s.charAt(i) >= '0' && s.charAt(i) <= '9') {
                int start = i;
                while (s.charAt(i + 1) >= '0' && s.charAt(i + 1)
 <= '9') {
                    i++;
                }
                count.push(Integer.parseInt(s.substring(start, i
 + 1)));
            } else if (s.charAt(i) == '[') {
                result.push("");
            } else if (s.charAt(i) == ']') {
                String temp = result.pop();
                StringBuilder sb = new StringBuilder();
                int index = count.pop();
                for (int j = 0; j < index; j++) {
                    sb.append(temp);
                }
                result.push(result.pop() + sb.toString());
            } else {
                result.push(result.pop() + s.charAt(i));
            }
            i++;
        }
        return result.pop();
    }
}
```

Unique Word Abbreviation

An abbreviation of a word follows the form . Below are some examples of word abbreviations:

```
a) it                      --> it    (no abbreviation)

     1
b) d|o|g                   --> d1g

              1    1  1
     1---5----0----5--8
c) i|nternationalizatio|n  --> i18n

              1
     1---5----0
d) l|ocalizatio|n          --> l10n
```

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no other word from the dictionary has the same abbreviation.

Example:

Given dictionary = ["deer", "door", "cake", "card"]

```
isUnique("dear") -> false;
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true
```

Tips:

1. 注意全局变量map，以及在哪里new map;
2. 需要注意，字典里可能有重复选项，也可能这个词在字典里没有，所以

```
if (map.containsKey(abb)) {
    if (!map.get(abb).equals(str)) {
        map.put(abb,"");
    }
}

还有一段：
return !map.containsKey(getAbb(word))||map.get(getAbb(word)).equu
als(word)
```

另外需要注意toString方法；

代码：

```
 public class ValidWordAbbr {
    HashMap<String, String> map;
    public ValidWordAbbr(String[] dictionary) {
        map = new HashMap<String, String>();
        for (String str : dictionary) {
            String abb = getAbb(str);
            if (map.containsKey(abb)) {
                if (!map.get(abb).equals(str)) {
                    map.put(abb,"");
                }
            } else {
                map.put(abb, str);
            }
        }
    }

    public boolean isUnique(String word) {
        return !map.containsKey(getAbb(word))||map.get(getAbb(wo
rd)).equals(word);
    }

    private String getAbb(String word) {
        if (word.length() <= 2) {
            return word;
        }
        String abb = word.charAt(0) + Integer.toString(word.leng
th() - 2) + word.charAt(word.length() - 1);
        return abb;
    }
}

// Your ValidWordAbbr object will be instantiated and called as
such:
// ValidWordAbbr vwa = new ValidWordAbbr(dictionary);
// vwa.isUnique("Word");
// vwa.isUnique("anotherWord");
```

Generalized Abbreviation

Write a function to generate the generalized abbreviations of a word.

Example:

```
Given word = "word", return the following list (order does not m
atter):
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1
o1d", "1or1", "w1r1" "1o2",
"2r1", "3d", "w3", "4"]
```

Tips:

Backtracking，关键是string的某一位char缩写还是不缩写

就是用dfs把每一位缩写和不缩写的所有情况遍历出来，缩写就变数字撒，然后变之前检查前一位是不是数字，前一位是数字就合并数字

方法2：

For each char c[i], either abbreviate it or not.

```
Abbreviate: count accumulate num of abbreviating chars, but don'
t append it yet.
Not Abbreviate: append accumulated num as well as current char c
[i].
In the end append remaining num.
Using StringBuilder can decrease 36.4% time.
This comes to the pattern I find powerful:
int len = sb.length(); // decision point
... backtracking logic ...
sb.setLength(len);     // reset to decision point
Similarly, check out remove parentheses and add operators.
```

Code:

```
    public List<String> generateAbbreviations(String word) {
        List<String> result = new ArrayList<>();
        if (word == null || word.length() == 0) {
            result.add("");
            return result;
        }
        helper(result, word, new StringBuilder(), 0);
        return result;
    }
    private void helper(List<String> result, String word, String
Builder build, int index) {
        if (index == word.length()) {
            result.add(build.toString());
            return;
        }
        String copy = build.toString();
        if (build.length() > 0 && build.charAt(build.length() -
1) >= '0' && (build.charAt(build.length() - 1) <= '9')) {
            char previous = build.charAt(build.length() - 1);
            if (previous == '9') {
                build.setCharAt(build.length() - 1, '1');
                build.append('0');
            }
            else build.setCharAt(build.length() - 1, (char)(prev
ious + 1));
        }
        else {
            build.append('1');
        }
        helper(result, word, build, index + 1);
        build = new StringBuilder(copy);
        build.append(word.charAt(index));
        helper(result, word, build, index + 1);
    }
```

解法2：

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
        List<String> res = new ArrayList<>();
        DFS(res, new StringBuilder(), word.toCharArray(), 0, 0);
        return res;
    }

    public void DFS(List<String> res, StringBuilder sb, char[] c
, int i, int num) {
        int len = sb.length();
        if(i == c.length) {
            if(num != 0) sb.append(num);
            res.add(sb.toString());
        } else {
            DFS(res, sb, c, i + 1, num + 1);              // ab
br c[i]

            if(num != 0) sb.append(num);                  // no
t abbr c[i]
            DFS(res, sb.append(c[i]), c, i + 1, 0);
        }
        //sb.setLength(len);
        if(sb.length() > 0) {
            sb.delete(len, sb.length());
        }
    }
}
```

Longest Absolute File Path

Suppose we abstract our file system by a string in the following manner:

The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:

```
dir
    subdir1
    subdir2
        file.ext
```

The directory dir contains an empty sub-directory subdir1 and a sub-directory subdir2 containing a file file.ext.

The string "dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext" represents:

```
dir
    subdir1
        file1.ext
        subsubdir1
    subdir2
        subsubdir2
            file2.ext
```

The directory dir contains two sub-directories subdir1 and subdir2. subdir1 contains a file file1.ext and an empty second-level sub-directory subsubdir1. subdir2 contains a second-level sub-directory subsubdir2 containing a file file2.ext.

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is "dir/subdir2/subsubdir2/file2.ext", and its length is 32 (not including the double quotes).

Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return 0.

Note: The name of a file contains at least a . and an extension. The name of a directory or sub-directory will not contain a .. Time complexity required: O(n) where n is the size of the input string.

Notice that a/aa/aaa/file1.txt is not the longest file path, if there is another path aaaaaaaaaaaaaaaaaaaaa/sth.png.

TIPS:

stack 和 array的解法差不多，下面是理解：都是O(n)因为所有的东西只会进一次出一次。

I'm assuming that "\t"will appear only before the directory or file name.

For example, if s = "\t\t\tdirname", then s.lastIndexOf("\t") will be 2, the number of "\t" will be 3. If a diretory name does not contain"\t", then s.lastIndexOf("\t") will be -1, the number of "\t" will be 0.

However, if "\t" is allowed within the directory or file name, then this way does not work.

用\t的个数来代表层数，\n来区分文件。

Code:

Array:(更快）

```
public class Solution {
    public int lengthLongestPath(String input) {
        String[] files = input.split("\n");
        int[] stack = new int[files.length + 1];
        int maxLength = 0;
        stack[0] = 0;
        for (String s : files) {
            int level = s.lastIndexOf("\t") + 1;
            int curLength = stack[level] + s.length() - level +
1;
            stack[level + 1] = curLength;
            if (s.contains(".")) {
                maxLength = Math.max(maxLength, curLength - 1);
            }
        }
        return maxLength;
    }
}
```

stack：

```
  public int lengthLongestPath(String input) {
          Deque<Integer> stack = new ArrayDeque<>();
          stack.push(0); // "dummy" length
          int maxLen = 0;
          for(String s:input.split("\n")){
              int lev = s.lastIndexOf("\t")+1; // number of "\t"
              while(lev+1<stack.size()) stack.pop(); // find par
ent
              int len = stack.peek()+s.length()-lev+1; // remove
 "/t", add"/"
              stack.push(len);
              // check if it is file
              if(s.contains(".")) maxLen = Math.max(maxLen, len-
1);
          }
          return maxLen;
      }
```

Fraction to Recurring Decimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

```
Given numerator = 1, denominator = 2, return "0.5".
Given numerator = 2, denominator = 1, return "2".
Given numerator = 2, denominator = 3, return "0.(6)".
```

Hint:

No scary math, just apply elementary math knowledge. Still remember how to perform a long division?

Try a long division on 4/9, the repeating part is obvious. Now try 4/333. Do you see a pattern?

Be wary of edge cases! List out as many test cases as you can think of and test your code thoroughly.

Tips:

- 个人觉得这道题难死了，找循环的部分，用括号括起来。
- 拟除法运算。关键就是处理循环小数部分。。。用hash表记录余数的位置。如果余数重复出现的话，那就是出现了循环小数。

  Use HashMap to store a remainder and its associated index while doing the division so that whenever a same remainder comes up, we know there is a repeating fractional part.
- 但是，测试用例真心碉堡了。会出现（INT_MIN）/(-1)这样的测试case，因为INT_MIN的绝对值比INT_MAX要大1，所以即使输入的都是int，但是中间结果很可能越界。所以需要使用long long。

- The important thing is to consider all edge cases while thinking this problem through, including: negative integer, possible overflow, etc.
- 这样记录每一位小数，放入map，直到出现重复，则加括号。

  ```
  num *= 10;
  res.append(num / den);
  num %= den;
  ```

Code:

```
public class Solution {
    public String fractionToDecimal(int numerator, int denominat
or) {
        if (numerator == 0) {
            return "0";
        }
        StringBuilder res = new StringBuilder();
        // "+" or "-"
        res.append((numerator < 0 == denominator < 0 || numerato
r == 0) ? "" : "-");
        long num = Math.abs((long)numerator);
        long den = Math.abs((long)denominator);

        // integral part
        res.append(num / den);
        num %= den;
        if (num == 0) {
            return res.toString();
        }

        // fractional part
        res.append(".");
        HashMap<Long, Integer> map = new HashMap<Long, Integer>(
);
        map.put(num, res.length());
        System.out.println(res.length());
        while (num != 0) {
            num *= 10;
            res.append(num / den);
```

```
            System.out.println(res.length());
            num %= den;
            if (map.containsKey(num)) {
                int index = map.get(num);
                res.insert(index, "(");
                res.append(")");
                break;
            }
            else {
                map.put(num, res.length());
            }
        }
        return res.toString();
    }
}
```

Flip Game

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to compute all possible states of the string after one valid move.

For example, given s = "++++", after one move, it may become one of the following states:

```
[
  "--++",
  "+--+",
  "++--"
]
```

If there is no valid move, return an empty list [].

Tips:

很简单啦，注意charAt后面用==而不是.equals()和substring的index就好

Code:

```
public class Solution {
    public List<String> generatePossibleNextMoves(String s) {
        List<String> result = new ArrayList<>();
        for (int i = 1; i < s.length(); i++) {
            if (s.charAt(i - 1) == '+'  && s.charAt(i) =='+'){
                result.add(s.substring(0, i - 1) + "--" + s.subs
tring(i + 1, s.length()));
            }
        }
        return result;
    }
}
```

Strobogrammatic Number

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

For example, the numbers "69", "88", and "818" are all strobogrammatic.

Tips:

走一遍，只要左右不是00 11 88 696就是错的。

Code：

```
public class Solution {
    public boolean isStrobogrammatic(String num) {
        if (num == null || num.length() == 0) {
            return true;
        }
        int left = 0, right = num.length() - 1;
        while (left <= right) {
            if (!"00 11 88 696".contains(num.charAt(left) + "" +
 num.charAt(right))) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }
}
```

Strobogrammatic Number II

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

For example, Given n = 2, return ["11","69","88","96"].

Hint:

Try to use recursion and notice that it should recurse with n - 2 instead of n - 1.

Tips:

我觉得还是挺简单的，就是把n-2的结果的外层 + 可能的几个数。

Code：

```
public class Solution {
    public List<String> findStrobogrammatic(int n) {
        List<String> answer = new ArrayList<String>();
        helper(answer, n, n);
        return answer;
    }

    public void helper(List<String> list, int n, int targetLen){
        if(n == 0){
            list.add("");
            return;
        }
        if(n == 1){
            list.add("0");
            list.add("1");
            list.add("8");
            return;
        }
        helper(list, n - 2, targetLen);
        int size = list.size();
        int i = 0;
        while(i < size){
            String cur = list.get(i);
            if(n != targetLen){
                list.add("0" + cur + "0");
            }
            list.add("1" + cur + "1");
            list.add("6" + cur + "9");
            list.add("8" + cur + "8");
            list.add("9" + cur + "6");
            list.remove(0);
            size--;
        }
    }
}
```

Strobogrammatic Number III

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high.

For example,

Given low = "50", high = "100", return 3. Because 69, 88, and 96 are three strobogrammatic numbers.

Note: Because the range might be a large number, the low and high numbers are represented as string.

Tips:

Construct char array from lenLow to lenHigh and increase count when s is between low and high. Add the stro pairs from outside to inside until left > right.

Code:

}

```
public class Solution {
    char[][] pairs = {{'0', '0'}, {'1', '1'}, {'6', '9'}, {'8',
'8'}, {'9', '6'}};
    int count = 0;

    public int strobogrammaticInRange(String low, String high) {
        for(int len = low.length(); len <= high.length(); len++)
 {
            dfs(low, high, new char[len], 0, len - 1);
        }
        return count;
    }

    public void dfs(String low, String high, char[] c, int left,
 int right) {
        if(left > right) {
            String s = new String(c);
            if((s.length() == low.length() && s.compareTo(low) <
 0) ||
               (s.length() == high.length() && s.compareTo(high)
 > 0)) {
                   return;
               }
            count++;
            return;
        }

        for(char[] p : pairs) {
            c[left] = p[0];
            c[right] = p[1];
            if(c.length != 1 && c[0] == '0') continue;
            if(left < right || left == right && p[0] == p[1]) {
                dfs(low, high, c, left + 1, right - 1);
            }
        }
    }
}
```

Decode String

Given an encoded string, return it's decoded string.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there won't be input like 3a or 2[4].

Examples:

```
s = "3[a]2[bc]", return "aaabcbc".
s = "3[a2[c]]", return "accaccacc".
s = "2[abc]3[cd]ef", return "abcabccdcdcdef".
```

Tips:

建立count和result两个stack，result先push(""),然后判断如下情况：

```
1. 数字：转化完成后放入count栈代用；
2.  [ ：result入栈”“；
3.  ] ：取出result.pop()的结果，入栈count.pop()次，注意要用StringBuil
der, 可变长度；
    注意最后：result.push(result.pop() + sb.toString());result里之
前的内容也要加。
4. 字母：result.push(result.pop() + s.charAt(i));
```

Code:

```
public class Solution {
    public String decodeString(String s) {
        if (s == null || s.length() == 0) return s;
        Stack<String> res = new Stack<>();
        Stack<Integer> count = new Stack<>();
        StringBuilder tempCount = new StringBuilder();
        res.push("");
        for (Character c : s.toCharArray()) {
            if (Character.isDigit(c)) {
                tempCount.append(c);
            } else if (c == '[') {
                count.push(Integer.parseInt(tempCount.toString()
));
                tempCount = new StringBuilder();
                res.push("");
            } else if (c == ']') {
                String temp = res.pop();
                StringBuilder sb = new StringBuilder();
                int index = count.pop();
                for (int i = 0; i < index; i++) {
                    sb.append(temp);
                }
                res.push(res.pop() + sb.toString());
            } else {
                res.push(res.pop() + c);
            }
        }
        return res.pop();
    }
}
```

2.

```
public class Solution {
    public String decodeString(String s) {
        Stack<Integer> count = new Stack<>();
        Stack<String> result = new Stack<>();
        result.push("");
        int i = 0;
        while (i < s.length()) {
           if (s.charAt(i) >= '0' && s.charAt(i) <= '9') {
                int start = i;
                while (s.charAt(i + 1) >= '0' && s.charAt(i + 1)
 <= '9') {
                    i++;
                }
                count.push(Integer.parseInt(s.substring(start, i
 + 1)));
            } else if (s.charAt(i) == '[') {
                result.push("");
            } else if (s.charAt(i) == ']') {
                String temp = result.pop();
                StringBuilder sb = new StringBuilder();
                int index = count.pop();
                for (int j = 0; j < index; j++) {
                    sb.append(temp);
                }
                result.push(result.pop() + sb.toString());
            } else {
                result.push(result.pop() + s.charAt(i));
            }
            i++;
        }
        return result.pop();
    }
}
```

Valid Parentheses

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.

Tips:

用stack， 看到左括号就入栈，看到右括号就出栈看能不能匹配到左括号。

Code：

```
public class Solution {
    public boolean isValid(String s) {
        if (s == null || s.length() == 0) {
            return true;
        }
        Stack<Character> stack = new Stack<>();

        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                stack.push('(');
                continue;
            }
            if (s.charAt(i) == '{') {
                stack.push('{');
                continue;
            }
            if (s.charAt(i) == '[') {
                stack.push('[');
                continue;
            }
            if (s.charAt(i) == ']') {
                if (stack.isEmpty()) {
                    return false;
                }
                if (stack.pop() != '[') {
                    return false;
                }
            }
            if (s.charAt(i) == '}') {
                if (stack.isEmpty()) {
                    return false;
                }
                if (stack.pop() != '{') {
                    return false;
                }
            }
            if (s.charAt(i) == ')') {
                if (stack.isEmpty()) {
                    return false;
                }
                if (stack.pop() != '(') {
                    return false;
                }
            }
        }

        if (stack.isEmpty()) {
            return true;
        }
        return false;
    }
}
```

Generate Parentheses

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

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

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

Tips:

典型的递归。一步步构造字符串。当左括号出现次数<n时，就可以放置新的左括号。当右括号出现次数小于左括号出现次数时，就可以放置新的右括号。

给定的n为括号对，所以就是有n个左括号和n个右括号的组合。

按顺序尝试知道左右括号都尝试完了就可以算作一个解。

注意，左括号的数不能大于右括号，要不然那就意味着先尝试了右括号而没有左括号，类似“)(” 这种解是不合法的。

Code:

```
public class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        if (n <= 0) {
            return result;
        }
        helper(result, "", n, n);
        return result;
    }
    private void helper(List<String> result, String build, int l
eft, int right) {
        if (left == 0 && right == 0) {
            result.add(build);
            return;
        }
        if (left > 0) {
            helper(result, build + "(", left - 1, right);
        }
        if (right > 0 && left < right) {
            helper(result, build + ")", left, right - 1);
        }
    }
}
```

Wildcard Matching

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

```
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequ
ence).

The matching should cover the entire input string (not partial).

The function prototype should be:
bool isMatch(const char *s, const char *p)

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
```

Tips:

O(n)

two pointer，两个字串对应两个指针。i， j 如果能匹配则index都加1。用一个starIndex来储存曾遇到过得star的位置，初始为-1，遇到star就赋值starindex，也用iIndex记住此时i的位置。遇到不能匹配时，如果starindex不等于-1说明遇到过star。则重置ij到iindex+ 1和starindex + 1，iindex++意思是把这一位i匹配到记录中的star里去。

Code：

```
public boolean isMatch(String s, String p) {
    int i = 0;
    int j = 0;
    int starIndex = -1;
    int iIndex = -1;

    while (i < s.length()) {
        if (j < p.length() && (p.charAt(j) == '?' || p.charAt(j)
 == s.charAt(i))) {
            ++i;
            ++j;
        } else if (j < p.length() && p.charAt(j) == '*') {
            starIndex = j;
            iIndex = i;
            j++;
        } else if (starIndex != -1) {
            j = starIndex + 1;
            i = iIndex+1;
            iIndex++;
        } else {
            return false;
        }
    }

    while (j < p.length() && p.charAt(j) == '*') {
        ++j;
    }

    return j == p.length();
```

Remove Duplicate Letters

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

```
Given "bcabc"
Return "abc"

Given "cbacdcbc"
Return "acdb"
```

Tips:

we need to know if we can replace a letter with some letter that is smaller in lexical order, so we should go through the string first, use a map to record the times one letter appears in total. Then we can go through the string again to build the result. We also need a set to record the letters that we believe currently have been correctly arranged in result.

Code:

```
public class Solution {
    public String removeDuplicateLetters(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }
        Map<Character, Integer> map = new  HashMap<>();
        for (int i = 0; i < s.length(); i++) {
            if (!map.containsKey(s.charAt(i))) {
                map.put(s.charAt(i), 1);
            }
            else map.put(s.charAt(i), map.get(s.charAt(i)) + 1);
        }
        Set<Character> visited = new HashSet<>();
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {
            char current = s.charAt(i);
            map.put(current, map.get(current) - 1);
            if (result.length() == 0) {
                result.append(current);
                visited.add(current);
            }
            else if (!visited.contains(current)) {
                while (result.length() > 0 && current <  result.
charAt(result.length() - 1) && map.get(result.charAt(result.leng
th() - 1)) > 0) {
                    char previous = result.charAt(result.length(
) - 1);
                    result.deleteCharAt(result.length() - 1);
                    visited.remove(previous);
                }
                result.append(current);
                visited.add(current);
            }
        }
        return result.toString();
    }
}
```

Regular Expression Matching

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)

Some examples:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "a*") → true
isMatch("aa", ".*") → true
isMatch("ab", ".*") → true
isMatch("aab", "c*a*b") → true
```

Tips:

1. If p.charAt(j) == s.charAt(i) : dp[i][j] = dp[i-1][j-1];
2. If p.charAt(j) == '.' : dp[i][j] = dp[i-1][j-1];
3. If p.charAt(j) == '*':

   here are two sub conditions:

```
   1.if p.charAt(j-1) != s.charAt(i) : dp[i][j] = dp[i][j-2]
 //in this case, a* only counts as empty
   2.if p.charAt(i-1) == s.charAt(i) or p.charAt(i-1) == '.':
                     dp[i][j] = dp[i-1][j]    //in this ca
se, a* counts as multiple a
                  or dp[i][j] = dp[i][j-1]   // in this ca
se, a* counts as single a
                  or dp[i][j] = dp[i][j-2]   // in this ca
se, a* counts as empty
```

Code:

```
 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 i = 0; i < p.length(); i++) {
        if (p.charAt(i) == '*' && dp[0][i-1]) {
            dp[0][i+1] = true;
        }
    }
    for (int i = 0 ; i < s.length(); i++) {
        for (int j = 0; j < p.length(); j++) {
            if (p.charAt(j) == '.') {
                dp[i+1][j+1] = dp[i][j];
            }
            if (p.charAt(j) == s.charAt(i)) {
                dp[i+1][j+1] = dp[i][j];
            }
            if (p.charAt(j) == '*') {
                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()];
}
```

Palindrome Permutation

Given a string, determine if a permutation of the string could form a palindrome.

For example, "code" -> False, "aab" -> True, "carerac" -> True.

Hint:

1. Consider the palindromes of odd vs even length. What difference do you notice?
2. Count the frequency of each character.
3. If each character occurs even number of times, then it must be a palindrome. How about 4. character which occurs odd number of times?

Tips：

用一个set来做，存取。

Code：

```
public class Solution {
    public boolean canPermutePalindrome(String s) {
        Set<Character> set=new HashSet<Character>();
        for(int i=0; i<s.length(); ++i){
            if (!set.contains(s.charAt(i)))
                set.add(s.charAt(i));
            else
                set.remove(s.charAt(i));
        }
        return set.size()==0 || set.size()==1;
    }
}
```

Nth Digit

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

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

Example 1:

```
Input:
3

Output:
3
```

Example 2:

```
Input:
11

Output:
0

Explanation:
The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
, ... is a 0, which is part of the number 10.
```

Tips:

Straight forward way to solve the problem in 3 steps:

```
find the length of the number where the nth digit is from
find the actual number where the nth digit is from
find the nth digit and return
```

Code:

```
public class Solution {
    public int findNthDigit(int n) {
        int len = 1;
        long count = 9;
        int start = 1;

        while (n > len * count) {
            n -= len * count;
            len += 1;
            count *= 10;
            start *= 10;
        }

        start += (n - 1) / len;
        String s = Integer.toString(start);
        return Character.getNumericValue(s.charAt((n - 1) % len)
);
    }
}
```

Reverse Vowels of a String

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

Example 1:

Given s = "hello", return "holle".

Example 2:

Given s = "leetcode", return "leotcede".

Note:

The vowels does not include the letter "y".

Tips:

Two Pointers;

In the inner while loop, don't forget the condition "start less than end" while incrementing start and decrementing end.

May use a HashSet to reduce the look up time to O(1)

Code:

```
public class Solution {
    public String reverseVowels(String s) {
        if(s == null || s.length()==0) return s;
        String vowels = "aeiouAEIOU";
        char[] chars = s.toCharArray();
        int start = 0;
        int end = s.length()-1;
        while(start<end){

            while(start<end && !vowels.contains(chars[start]+"")
){
                start++;
            }

            while(start<end && !vowels.contains(chars[end]+"")){
                end--;
            }

            char temp = chars[start];
            chars[start] = chars[end];
            chars[end] = temp;

            start++;
            end--;
        }
        return new String(chars);
    }
}
```

Group Shifted Strings

Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" -> "bcd". We can keep "shifting" which forms the sequence:

```
"abc" -> "bcd" -> ... -> "xyz"
```

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

For example, given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"], A solution is:

```
[
  ["abc","bcd","xyz"],
  ["az","ba"],
  ["acef"],
  ["a","z"]
]
```

Tips:

O(n2)

每个字母和首字母的相对距离都是相等的，比如abc和efg互为偏移，对于abc来说，b和a的距离是1，c和a的距离是2，对于efg来说，f和e的距离是1，g和e的距离是2。

再来看一个例子，az和yx，z和a的距离是25，x和y的距离也是25(直接相减是-1，这就是要加26然后取余的原因)，那么这样的话，所有互为偏移的字符串都有个unique的距离差，我们根据这个来建立映射就可以很好的进行单词分组了

Code:

```
public class Solution {
    public List<List<String>> groupStrings(String[] strings) {
        List<List<String>> result = new ArrayList<>();
        Map<String, List<String>> map = new HashMap<>();
        for(String s: strings){
            String key = getBitMap(s);
            if(!map.containsKey(key))
                map.put(key, new ArrayList<String>());
            map.get(key).add(s);
        }
        for(String key: map.keySet()){
            List<String> list = map.get(key);
            //Collections.sort(list);
            result.add(list);
        }
        return result;
    }
    private String getBitMap(String s){
        int[] arr = new int[s.length()];
        arr[0] = 0;
        for(int i = 1; i < s.length(); i++){
            arr[i] = s.charAt(i)-s.charAt(0) < 0?
                    ((s.charAt(i)-s.charAt(0))%26 + 26): (s.cha
rAt(i)-s.charAt(0));
        }
        return Arrays.toString(arr);
    }
}
```

Rearrange String k Distance Apart

Given a non-empty string str and an integer k, rearrange the string such that the same characters are at least distance k from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".

Example 1:

```
str = "aabbcc", k = 3

Result: "abcabc"

The same letters are at least distance 3 from each other.
```

Example 2:

```
str = "aaabc", k = 3

Answer: ""

It is not possible to rearrange the string.
```

Example 3:

```
str = "aaadbbcc", k = 2

Answer: "abacabcd"

Another possible answer is: "abcabcda"

The same letters are at least distance 2 from each other.
```

Tips:

这道题给了我们一个字符串str，和一个整数k，让我们对字符串str重新排序，使得其中相同的字符之间的距离不小于k，这道题的难度标为Hard，看来不是省油的灯。的确，这道题的解法用到了哈希表，堆，和贪婪算法。

我们需要一个哈希表来建立字符和其出现次数之间的映射，然后需要一个堆来保存这每一堆映射，按照出现次数来排序。然后如果堆不为空我们就开始循环，我们找出k和str长度之间的较小值，然后从0遍历到这个较小值，对于每个遍历到的值，如果此时堆为空了，说明此位置没法填入字符了，返回空字符串，否则我们从堆顶取出一对映射，然后把字母加入结果res中，此时映射的个数减1，如果减1后的个数仍大于0，则我们将此映射加入临时集合v中，同时str的个数len减1，遍历完一次，我们把临时集合中的映射对由加入堆中。

The greedy algorithm is that in each step, select the char with highest remaining count if possible (if it is not in the waiting queue). PQ is used to achieve the greedy. A regular queue waitQueue is used to "freeze" previous appeared char in the period of k.

In each iteration, we need to add current char to the waitQueue and also release the char at front of the queue, put back to maxHeap. The "impossible" case happens when the maxHeap is empty but there is still some char in the waitQueue.

Code:

```
public class Solution {
    public String rearrangeString(String str, int k) {

        StringBuilder rearranged = new StringBuilder();
        //count frequency of each char
        Map<Character, Integer> map = new HashMap<>();
        for (char c : str.toCharArray()) {
            if (!map.containsKey(c)) {
                map.put(c, 0);
            }
            map.put(c, map.get(c) + 1);
        }

        //construct a max heap using self-defined comparator, wh
ich holds all Map entries, Java is quite verbose
```

```
        Queue<Map.Entry<Character, Integer>> maxHeap = new Prior
ityQueue<>(new Comparator<Map.Entry<Character, Integer>>() {
            public int compare(Map.Entry<Character, Integer> ent
ry1, Map.Entry<Character, Integer> entry2) {
                return entry2.getValue() - entry1.getValue();
            }
        });

        Queue<Map.Entry<Character, Integer>> waitQueue = new Lin
kedList<>();
        maxHeap.addAll(map.entrySet());

        while (!maxHeap.isEmpty()) {

            Map.Entry<Character, Integer> current = maxHeap.poll
();
            rearranged.append(current.getKey());
            current.setValue(current.getValue() - 1);
            waitQueue.offer(current);

            if (waitQueue.size() < k) { // intial k-1 chars, wai
tQueue not full yet
                continue;
            }
            // release from waitQueue if char is already k apart
            Map.Entry<Character, Integer> front = waitQueue.poll
();
            //note that char with 0 count still needs to be plac
ed in waitQueue as a place holder
            if (front.getValue() > 0) {
                maxHeap.offer(front);
            }
        }

        return rearranged.length() == str.length() ? rearranged.
toString() : "";
    }

}
```

Minimum Unique Word Abbreviation

A string such as "word" contains the following abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Given a target string and a set of strings in a dictionary, find an abbreviation of this target string with the smallest possible length such that it does not conflict with abbreviations of the strings in the dictionary.

Each number or letter in the abbreviation is considered length = 1. For example, the abbreviation "a32bc" has length = 4.

Note:

1. In the case of multiple answers as shown in the second example below, you may return any one of them.
2. Assume length of target string = m, and dictionary size = n. You may assume that m ≤ 21, n ≤ 1000, and log2(n) + m ≤ 20.

Examples:

```
"apple", ["blade"] -> "a4" (because "5" or "4e" conflicts with "
blade")

"apple", ["plain", "amber", "blade"] -> "1p3" (other valid answe
rs include "ap3", "a3e", "2p2", "3le", "3l1").
```

Tips:

这道题实际上是之前那两道Valid Word Abbreviation和Generalized Abbreviation的合体，我们的思路其实很简单，首先找出target的所有的单词缩写的形式，然后按照长度来排序，小的排前面，我们用优先队列来自动排序，里面存一个pair，保存单词缩写及其长度，然后我们从最短的单词缩写开始，跟dictionary中所有的单词一一进行验证，利用Valid Word Abbreviation中的方法，看其是否是合法的单词的缩写，如果是，说明有冲突，直接break，进行下一个单词缩写的验证。

Similar to the bit manipulation idea, I use "" *to replace the a char in the target string. Then, we start with putting a string with all "**" which has length =1 into the heap. When pop, check whether it is an overlap with other words in the dictionary. If not, we get the result and return it. Otherwise, we generate the next abbr from current one by replacing* "" with a char of the target string.

Code:

```
public class Solution {
    class Abbr{
        String abbr;
        int len;
        Abbr(String abbr, int len) {
            this.abbr = abbr;
            this.len = len;
        }
    }
    public String minAbbreviation(String target, String[] dictio
nary) {
        Set<String> visited = new HashSet();
        PriorityQueue<Abbr> q = new PriorityQueue(1, new CompAbb
r() );
        int len=target.length();
        String first = "";
        for (int i=0; i<len; i++) first+="*";
        q.offer(new Abbr(first, 1) );
        while (! q.isEmpty() ) {
            Abbr ab = q.poll();
            String abbr = ab.abbr;
            boolean conflict = false;
            for (String word: dictionary) {
                if ( (word.length() == len) && isConflict(abbr,
word) ) {
                    conflict = true;
                    break;
                }
            }
            if (conflict) {
                generateAbbr(target, abbr, visited, q);
            } else {
                return NumAbbr(abbr);
            }
        }
        return null;
    }
    int abbrLength(String abbr){
        int ret = 0, star = 0;
        for (char c: abbr.toCharArray()){
            if (c >= 'a' && c <= 'z') {
                ret += 1 + star;
                star = 0;
            } else if (c=='*'){
                star = 1;
            }
        }
        return ret+star;
    }

    class CompAbbr implements Comparator<Abbr> {
        public int compare(Abbr s1, Abbr s2){
            return Integer.compare( s1.len, s2.len );
        }
    }

    void generateAbbr(String str, String abbr, Set<String> visit
ed, PriorityQueue<Abbr> q){
        char[] temp = abbr.toCharArray();
        for (int i=0; i<temp.length;i++){
            if (temp[i] == '*') {
                temp[i] = str.charAt(i);
                String next = new String(temp);
                if (! visited.contains(next) ) {
                    q.offer( new Abbr(next, abbrLength(next) ) )
;

                    visited.add( next );
                }
                temp[i] = '*';
            }
        }
    }
    boolean isConflict(String abbr, String str){
        for (int i=0; i<abbr.length(); i++){
            if ( abbr.charAt(i) != '*' &&  str.charAt(i) != abbr
.charAt(i) ) return false;
        }
        return true;
    }
    String NumAbbr(String abbr){
        String ret="";
        int count=0;
        for (char c : abbr.toCharArray() ){
            if (c >='a' && c <='z') {
                if (count > 0) {
                    ret += count;
                    count=0;
                }
                ret+=c;
            } else {
                count++;
            }
        }
        if (count > 0) ret += count;
        return ret;
    }
}
```

CPP:

```
class Solution {
public:
    string minAbbreviation(string target, vector<string>& dictio
nary) {
        if (dictionary.empty()) return to_string((int)target.siz
e());
        priority_queue<pair<int, string>, vector<pair<int, strin
g>>, greater<pair<int, string>>> q;
        q = generate(target);
        while (!q.empty()) {
            auto t = q.top(); q.pop();
            bool no_conflict = true;
            for (string word : dictionary) {
                if (valid(word, t.second)) {
                    no_conflict = false;
                    break;
                }
            }
            if (no_conflict) return t.second;
        }
        return "";
    }
    priority_queue<pair<int, string>, vector<pair<int, string>>,
 greater<pair<int, string>>> generate(string target) {
        priority_queue<pair<int, string>, vector<pair<int, strin
g>>, greater<pair<int, string>>> res;
        for (int i = 0; i < pow(2, target.size()); ++i) {
            string out = "";
            int cnt = 0, size = 0;
            for (int j = 0; j < target.size(); ++j) {
                if ((i >> j) & 1) ++cnt;
                else {
                    if (cnt != 0) {
                        out += to_string(cnt);
                        cnt = 0;
                        ++size;
                    }
                    out += target[j];
                    ++size;
                }
            }
            if (cnt > 0) {
                out += to_string(cnt);
                ++size;
            }
            res.push({size, out});
        }
        return res;
    }
    bool valid(string word, string abbr) {
        int m = word.size(), n = abbr.size(), p = 0, cnt = 0;
        for (int i = 0; i < abbr.size(); ++i) {
            if (abbr[i] >= '0' && abbr[i] <= '9') {
                if (cnt == 0 && abbr[i] == '0') return false;
                cnt = 10 * cnt + abbr[i] - '0';
            } else {
                p += cnt;
                if (p >= m || word[p++] != abbr[i]) return false
;
                cnt = 0;
            }
        }
        return p + cnt == m;
    }
};
```

Longest Substring with At Most Two Distinct Characters

Given a string, find the length of the longest substring T that contains at most 2 distinct characters.

For example, Given s = “eceba”,

T is "ece" which its length is 3.

Tips：

这道题给我们一个字符串，让我们求最多有两个不同字符的最长子串。

用哈希表来做，记录每个字符的出现次数，如果哈希表中的映射数量超过两个的时候，我们需要删掉一个映射。

比如此时哈希表中e有2个，c有1个，此时把b也存入了哈希表，那么就有三对映射了，这时我们的left是0，先从e开始，映射值减1，此时e还有1个，不删除，left自增1。这是哈希表里还有三对映射，此时left是1，那么到c了，映射值减1，此时e映射为0，将e从哈希表中删除，left自增1，然后我们更新结果为i - left + 1，以此类推直至遍历完整个字符串。

Code:

```
public class Solution {
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        int res = 0, left = 0;
        Map<Character, Integer> map = new HashMap<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (!map.containsKey(s.charAt(i))) map.put(c, 0);
            map.put(c, map.get(c) + 1);
            while (map.size() > 2) {
                char l = s.charAt(left);
                map.put(l, map.get(l) - 1);
                if (map.get(l) == 0) map.remove(l);
                left++;
            }
            res = Math.max(res, i - left + 1);
        }
        return res;
    }
}
```

Longest Substring with At Most K Distinct Characters

Given a string, find the length of the longest substring T that contains at most k distinct characters.

For example, Given s = “eceba” and k = 2,

T is "ece" which its length is 3.

Tips:

和上题一毛一样，把2换成k就行。

Code：

```
public class Solution {
    public int lengthOfLongestSubstringKDistinct(String s, int k
) {
        int res = 0, left = 0;
        Map<Character, Integer> map = new HashMap<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (!map.containsKey(s.charAt(i))) map.put(c, 0);
            map.put(c, map.get(c) + 1);
            while (map.size() > k) {
                char l = s.charAt(left);
                map.put(l, map.get(l) - 1);
                if (map.get(l) == 0) map.remove(l);
                left++;
            }
            res = Math.max(res, i - left + 1);
        }
        return res;
    }
}
```

Palindrome Pairs

Given a list of unique words, find all pairs of distinct indices (i, j) in the given list, so that the concatenation of the two words, i.e. words[i] + words[j] is a palindrome.

Example 1:

```
Given words = ["bat", "tab", "cat"]
Return [[0, 1], [1, 0]]
The palindromes are ["battab", "tabbat"]
```

Example 2:

```
Given words = ["abcd", "dcba", "lls", "s", "sssll"]
Return [[0, 1], [1, 0], [3, 2], [2, 4]]
The palindromes are ["dcbaabcd", "abcddcba", "slls", "llssssll"]
```

Tips:

Trie: O(n*k^2)

1. For word "ba", starting from the first character 'b', index into the root.next array with index ('b' - 'a' = 1). The corresponding node is null, then we know there are no words ending at this character, so the search is terminated;
2. For word "a", again indexing into array root.next at index ('a' - 'a' = 0) will yield node n1, which is not null. We then check the value of n1.isWord. If it is true, then it is possible to obtain a palindrome by appending this word to the one currently being examined (a.k.a. word "a"). Also note that the two words should be different, but the n1.isWord field provides no information about the word itself, which makes it impossible to distinguish the two words. So we need to modify the fields of the TrieNode so we can identify the word it represents. One easy way is to have an integer field to remember the index of the word in the "words" array. For non-word nodes, this integer will take negative values (-1 for example) while for those representing a word, it will be non-negative values. Suppose we have made this modification, then we know

the two words are the same, so we discard this pair combination. Since the word "a" has only one letter, it seems we are done with it. Or do we? Not really. What if we have words with suffix "a" ("aaa" in this case)? We need to continue check the rest part of these words (such as "aa" for the word "aaa") and see if the rest forms a palindrome. If it is, then appending this word ("aaa" in this case) to the original word ("a") will also form a palindrome ("aaaa"). Here I take another strategy: add an integer list to each TrieNode; the list will record the indices of all words satisfying the following two conditions: each word has a suffix represented by the current Trie node; the rest of the word forms a palindrome.

Two Pointers：**O(n*k^2)**

Code:

```
public class Solution {
    class TrieNode {
        TrieNode[] next;
        int index;
        List<Integer> list;

        TrieNode() {
            next = new TrieNode[26];
            index = -1;
            list = new ArrayList<>();
        }
    }

    public List<List<Integer>> palindromePairs(String[] words) {
        List<List<Integer>> res = new ArrayList<>();

        TrieNode root = new TrieNode();
        for (int i = 0; i < words.length; i++) addWord(root, wor
ds[i], i);
        for (int i = 0; i < words.length; i++) search(words, i,
root, res);

        return res;
    }
    private void addWord(TrieNode root, String word, int index)
{
        for (int i = word.length() - 1; i >= 0; i--) {
            int j = word.charAt(i) - 'a';
            if (root.next[j] == null) root.next[j] = new TrieNod
e();
            if (isPalindrome(word, 0, i)) root.list.add(index);
            root = root.next[j];
        }

        root.list.add(index);
        root.index = index;
    }

    private void search(String[] words, int i, TrieNode root, Li
st<List<Integer>> res) {
        for (int j = 0; j < words[i].length(); j++) {
            if (root.index >= 0 && root.index != i && isPalindro
me(words[i], j, words[i].length() - 1)) {
                res.add(Arrays.asList(i, root.index));
            }

            root = root.next[words[i].charAt(j) - 'a'];
              if (root == null) return;
        }

        for (int j : root.list) {
            if (i == j) continue;
            res.add(Arrays.asList(i, j));
        }
    }

    private boolean isPalindrome(String word, int i, int j) {
        while (i < j) {
            if (word.charAt(i++) != word.charAt(j--)) return fal
se;
        }

        return true;
    }
}
```

Two Pointers:

```
public List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> pairs = new LinkedList<>();
    if (words == null) return pairs;
    HashMap<String, Integer> map = new HashMap<>();
    for (int i = 0; i < words.length; ++ i) map.put(words[i], i)
;
    for (int i = 0; i < words.length; ++ i) {
        int l = 0, r = 0;
        while (l <= r) {
            String s = words[i].substring(l, r);
            Integer j = map.get(new StringBuilder(s).reverse().t
oString());
            if (j != null && i != j && isPalindrome(words[i].sub
string(l == 0 ? r : 0, l == 0 ? words[i].length() : l)))
                pairs.add(Arrays.asList(l == 0 ? new Integer[]{i
, j} : new Integer[]{j, i}));
            if (r < words[i].length()) ++r;
            else ++l;
        }
    }
    return pairs;
}

private boolean isPalindrome(String s) {
    for (int i = 0; i < s.length()/2; ++ i)
        if (s.charAt(i) != s.charAt(s.length()-1-i))
            return false;
    return true;
}
```

License Key Formatting

Now you are given a string S, which represents a software license key which we would like to format. The string S is composed of alphanumerical characters and dashes. The dashes split the alphanumerical characters within the string into groups. (i.e. if there are M dashes, the string is split into M+1 groups). The dashes in the given string are possibly misplaced.

We want each group of characters to be of length K (except for possibly the first group, which could be shorter, but still must contain at least one character). To satisfy this requirement, we will reinsert dashes. Additionally, all the lower case letters in the string must be converted to upper case.

So, you are given a non-empty string S, representing a license key to format, and an integer K. And you need to return the license key formatted according to the description above.

Example 1:

```
Input: S = "2-4A0r7-4k", K = 4

Output: "24A0-R74K"

Explanation: The string S has been split into two parts, each pa
rt has 4 characters.
```

Example 2:

```
Input: S = "2-4A0r7-4k", K = 3

Output: "24-A0R-74K"

Explanation: The string S has been split into three parts, each
part has 3 characters except the first part as it could be short
er as said above.
```

Note:

1. The length of string S will not exceed 12,000, and K is a positive integer.
2. String S consists only of alphanumerical characters (a-z and/or A-Z and/or 0-9) and dashes(-).
3. String S is non-empty.

Tips:

注意算的时候用已经做好的sb算。

Code：

```
public class Solution {
    public String licenseKeyFormatting(String S, int K) {
        if (S == null || S.length() == 0) return S;
        StringBuilder sb = new StringBuilder();
        for (int i = S.length() - 1; i >= 0; i--) {
            if (S.charAt(i) != '-')
                sb.append (sb.length() % (K + 1) == K ? '-' : ""
).append(S.charAt(i));
        }
        return sb.reverse().toString().toUpperCase();
    }
}
```

Given a rows x cols screen and a sentence represented by a list of non-empty words, find how many times the given sentence can be fitted on the screen.

Note:

1. A word cannot be split into two lines.
2. The order of words in the sentence must remain unchanged.
3. Two consecutive words in a line must be separated by a single space.
4. Total words in the sentence won't exceed 100.
5. Length of each word is greater than 0 and won't exceed 10.
6. 1 ≤ rows, cols ≤ 20,000.

Example 1:

```
Input:
rows = 2, cols = 8, sentence = ["hello", "world"]

Output:
1

Explanation:
hello---
world---

The character '-' signifies an empty space on the screen.
```

Example 2:

```
Input:
rows = 3, cols = 6, sentence = ["a", "bcd", "e"]

Output:
2

Explanation:
a-bcd-
e-a---
bcd-e-

The character '-' signifies an empty space on the screen.
```

Example 3:

```
Input:
rows = 4, cols = 5, sentence = ["I", "had", "apple", "pie"]

Output:
1

Explanation:
I-had
apple
pie-I
had--

The character '-' signifies an empty space on the screen.
```

Tips: O(Row * MaxWordLen) 直接的方法是每次扫描一行，尝试能放几个，这样时间复杂度会高一点．另外一种方法是把所有的字符串都加起来，然后每次看如果位移一整行的距离是否正好落在这个字符串的空格位置，如果不是的话就退后，直到遇到一个空格．

Code:

```
public class Solution {
    public int wordsTyping(String[] sentence, int rows, int cols
) {
        String str = "";
        for (String s : sentence) {
                str += s + " ";
        }
        int len = str.length();
        int start = 0;
        for (int i = 0; i < rows; i++) {
            start += cols;
            if (str.charAt(start % len) == ' ') {
                start++;
                continue;
            }
            while(start > 0 && str.charAt((start - 1) % len) !='
 ') start--;
        }
        return start / len;
    }
}
```

Optimized:

```
public int wordsTyping(String[] sentence, int rows, int cols) {
    String s = String.join(" ", sentence) + " ";
    int len = s.length(), count = 0;
    int[] map = new int[len];
    for (int i = 1; i < len; ++i) {
        map[i] = s.charAt(i) == ' ' ? 1 : map[i-1] - 1;
    }
    for (int i = 0; i < rows; ++i) {
        count += cols;
        count += map[count % len];
    }
    return count / len;
}
```

Add Strings

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

Note:

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

Tips:

注意判断0。

Code:

```
public class Solution {
    public String addStrings(String num1, String num2) {
        StringBuilder sb = new StringBuilder();
        int carry = 0;
        for(int i = num1.length() - 1, j = num2.length() - 1; i
>= 0 || j >= 0 || carry == 1; i--, j--){
            int x = i < 0 ? 0 : num1.charAt(i) - '0';
            int y = j < 0 ? 0 : num2.charAt(j) - '0';
            sb.append((x + y + carry) % 10);
            carry = (x + y + carry) / 10;
        }
        return sb.reverse().toString();
    }
}
```

DP

Guess Number Higher or Lower II

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

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

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

However, when you guess a particular number x, and you guess wrong, you pay $x. You win the game when you guess the number I picked.

Example:

```
n = 10, I pick 8.

First round:  You guess 5, I tell you that it's higher. You pay
$5.
Second round: You guess 7, I tell you that it's higher. You pay
$7.
Third round:  You guess 9, I tell you that it's lower. You pay $
9.

Game over. 8 is the number I picked.

You end up paying $5 + $7 + $9 = $21.
```

Given a particular n ≥ 1, find out how much money you need to have to guarantee a win.

Hint:

```
  The best strategy to play the game is to minimize the maximum
loss you could possibly face.
  Another strategy is to minimize the expected loss.
  Here, we are interested in the first scenario.Show More Hint
```

Tips:

求**MinMax!!**

It looks like the complexity is **O(n^3)** because there are **n^2** subproblems, and each subproblem takes O(n) time to compute.

这题要求我们在猜测数字y未知的情况下（1~n任意一个数），要我们在最坏情况下我们支付最少的钱。也就是说要考虑所有y的情况。

我们假定选择了一个错误的数x，（1<=x<=n && x!=y ）那么就知道接下来应该从[1,x-1] 或者[x+1,n]中进行查找。 假如我们已经解决了[1,x-1] 和 [x+1,n]计算问题，我们将其表示为solve(L,x-1) 和solve(x+1,n)，那么我们应该选择max(solve(L,x-1),solve(x+1,n)) 这样就是求最坏情况下的损失。总的损失就是 f(x) = x + max(solve(L,x-1),solve(x+1,n))

那么将x从1~n进行遍历，取使得 f(x) 达到最小，来确定最坏情况下最小的损失，也就是我们初始应该选择哪个数。

具体：

我们计算从i开始到i + （l-1）的interval的最小花销。 其中设置l为interval的length，例如[2，3，4]的length为3，所以这个interval是[i, i + (l - 1)];

我们需要建立一个二维的dp数组，其中dp[i][i + (l-1)]表示从数字i开始长度为L的interval之间猜中任意一个数字最少需要花费的钱数。

我们需要遍历每一段区间[i,i+(l-1)]，维护一个全局最小值dp[i][i + (l - 1)]变量，然后遍历该区间中的每一个数字，计算局部最大值costForThisGuess = g + max(dp[j][g - 1], dp[g + 1][i])。其中g为在这个区间内猜测的数。

最后，为了找出这段区间的全局最小值，计算dp[i][i + (l - 1)] = Math.min(dp[i][i + (l - 1)], costForThisGuess)。

值得注意的是，当g==i + (l - 1)，即g为这个区间的最后一个数时，我们是没有第二段区间的，所以在计算costForThisGuess时，需要先判定，如果g==i + (l -1), costForThisGuess = dp[i][g - 1] + g。

最后其实我们要求的区间为[1，n]，所以return dp[1][n];

Code:

```
public class Solution {
    public int getMoneyAmount(int n) {
        // all intervals are inclusive
        // uninitialized cells are assured to be zero
        // the zero column and row will be uninitialized
        // the illegal cells will also be uninitialized
        // add 1 to the length just to make the index the same a
s numbers used
        int[][] dp = new int[n + 1][n + 1]; // dp[i][j] means th
e min cost in the worst case for numbers (i...j)

        // iterate the lengths of the intervals since the calcul
ations of longer intervals rely on shorter ones
        for (int l = 2; l <= n; l++) {
            // iterate all the intervals with length l, the star
t of which is i. Hence the interval will be [i, i + (l - 1)]
            for (int i = 1; i + (l - 1) <= n; i++) {
                dp[i][i + (l - 1)] = Integer.MAX_VALUE;
                // iterate all the first guesses g
                for (int g = i; g <= i + (l - 1); g++) {
                    int costForThisGuess;
                    // since if g is the last integer, g + 1 doe
s not exist, we have to separate this case
                    // cost for [i, i + (l - 1)]: g (first guess
) + max{the cost of left part [i, g - 1], the cost of right part
 [g + 1, i + (l - 1)]}
                    if (g == i + (l-1)) {
                        costForThisGuess = dp[i][g - 1] + g;
                    } else {
                        costForThisGuess = g + Math.max(dp[i][g
- 1], dp[g + 1][i + (l - 1)]);
                    }
                    dp[i][i + (l - 1)] = Math.min(dp[i][i + (l -
 1)], costForThisGuess); // keep track of the min cost among all
 first guesses
                }
            }
        }
        return dp[1][n];
    }
}
```

Count Numbers with Unique Digits

Given a non-negative integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.

Example: Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])

Tips:

排列组合题。

设i为长度为i的各个位置上数字互不相同的数。

```
i==1 : 1 0（0~9共10个数，均不重复）
i==2: 9 * 9 （第一个位置上除0外有9种选择，第2个位置上除第一个已经选择的数
，还包括数字0，也有9种选择）
i ==3: 9* 9 * 8 （前面两个位置同i==2，第三个位置除前两个位置已经选择的数
还有8个数可以用）
……
i== n: 9 * 9 * 8 *…… (9-i+2)
```

需要注意的是，9- i + 2 >0 即 i < 11，也就是i最大为10，正好把每个数都用了一遍。

Code:

```
public class Solution {
    public int countNumbersWithUniqueDigits(int n) {
        int result = 1;
        int curlev = 9;
        for (int i = 1; i <= n; i++) {
            result = result + curlev;
            curlev = curlev * (10 - i);
        }
        return result;
    }
}
```

Largest Divisible Subset

Given a set of distinct positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 0.

If there are multiple solutions, return any subset is fine.

Example 1:

```
nums: [1,2,3]

Result: [1,2] (of course, [1,3] will also be ok)
```

Example 2:

```
nums: [1,2,4,8]

Result: [1,2,4,8]
```

Tips:

DP，复杂度O(n^2)。

两个循环，第二个循环是for(int j = i; j>=0; j--),也就是对i之前的所有数求count的最大值，以便求出遍历到i是可得到的最大值。

除了count数组外，还需要用一个pre数组来存储当前节点的最大因数。

Code：

```
public class Solution {
    public List<Integer> largestDivisibleSubset(int[] nums) {
        int n = nums.length;
        int[] count = new int[n];
        int[] pre = new int[n];
        Arrays.sort(nums);
        int max = 0, index = 0;
        for (int i = 0; i < n; i++) {
            count[i] = 1;
            pre[i] = -1;
            for (int j = i; j >= 0; j--) {
                if (nums[i] % nums[j] == 0) {
                    if (count[i] < count[j] + 1) {
                        count[i] = count[j] + 1;
                        pre[i] = j;
                    }
                }
            }
            if (count[i] > max) {
                max = count[i];
                index = i;
            }
        }
        List<Integer> res = new ArrayList<>();
        for (int i = 1; i < max; i++) {
            res.add(nums[index]);
            index = pre[index];
        }
        return res;
    }
}
```

Best Time to Buy and Sell Stock with Cooldown

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]
```

Tips:

- 基础：要额外O(n)空间 有了cooldown之后需要设置buy和sell两个数组，buy[i]表示第i天的时候买入股份的最大利益，sell[i]表示第i天的时候卖出股份的最大利益。

初始化：

```
        buy[0] = -prices[0];
        buy[1] = prices[1] > prices[0] ? -prices[0] : -prices[1]
;
        sell[0] = 0;
        sell[1] = prices[1] > prices[0] ? prices[1] - prices[0]
: 0;
```

状态转移方程为：

```
        buys[i] = max(sells[i - 2] - prices[i], buys[i - 1])
        sells[i] = max(buys[i - 1] + prices[i], sells[i - 1])
```

上述方程的含义为：

第i天买入的最大累积收益 = max(第i-2天卖出，第i天买入；第i-1天买入，第i天持有) 第i天卖出的最大累积收益 = max(第i-1天买入~第i天卖出的最大累积收益；第i-1天卖出，第i天没交易)

- 优化： 只需要额外O(1)空间。

如果不满足于空间复杂度，可进行优化，用prev_sell和prev_buy优化，因为sell和buy只与他们的前两天有关。

Code：

基础：

```
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        if (prices.length == 1) {
            return 0;
        }
        int n = prices.length;
        int[] buy = new int[n];
        int[] sell = new int[n];
        buy[0] = -prices[0];
        buy[1] = prices[1] > prices[0] ? -prices[0] : -prices[1]
;
        sell[0] = 0;
        sell[1] = prices[1] > prices[0] ? prices[1] - prices[0]
: 0;
        for (int i = 2; i < n; i++) {
            buy[i] = Math.max(sell[i - 2] - prices[i], buy[i - 1
]);
            sell[i] = Math.max(buy[i - 1] + prices[i], sell[i -
1]);
        }
        return sell[n - 1];
    }
}
```

优化：

```
public int maxProfit(int[] prices) {
    int sell = 0, prev_sell = 0, buy = Integer.MIN_VALUE, prev_b
uy;
    for (int price : prices) {
        prev_buy = buy;
        buy = Math.max(prev_sell - price, prev_buy);
        prev_sell = sell;
        sell = Math.max(prev_buy + price, prev_sell);
    }
    return sell;
}
```

Flip Game II

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

For example, given s = "++++", return true. The starting player can guarantee a win by flipping the middle "++" to become "+--+".

Follow up: Derive your algorithm's runtime complexity.

Tips:

第一题的变形，和拿硬币类似，用DP+记忆化搜索做。

- 复杂度为O(n!!)的做法是一个一个把所有可以换成”--“的”++“都换好，然后看对手能不能赢，如果都不行，就return true。

The idea is try to replace every "++" in the current string s to "--" and see if the opponent can win or not, if the opponent cannot win, great, we win!

For the time complexity, here is what I thought, let's say the length of the input string s is n, there are at most n - 1 ways to replace "++" to "--" (imagine s is all "+++..."), once we replace one "++", there are at most (n - 2) - 1 ways to do the replacement, it's a little bit like solving the N-Queens problem, the time complexity is (n - 1) x (n - 3) x (n - 5) x ..., so it's O(n!!), double factorial.

- 复杂度为O(n^2)的优化是加一个HashMap，把之前所有的结果都记录下来，可以直接调用。

Code：

naive：

```
public boolean canWin(String s) {
  if (s == null || s.length() < 2) {
    return false;
  }

  for (int i = 0; i < s.length() - 1; i++) {
    if (s.startsWith("++", i)) {
      String t = s.substring(0, i) + "--" + s.substring(i + 2);

      if (!canWin(t)) {
        return true;
      }
    }
  }

  return false;
}
```

optimization：

```
public boolean canWin(String s) {
    if (s == null || s.length() < 2) {
        return false;
    }
    HashMap<String, Boolean> winMap = new HashMap<String, Boolea
n>();
    return helper(s, winMap);
}

public boolean helper(String s, HashMap<String, Boolean> winMap)
 {
    if (winMap.containsKey(s)) {
        return winMap.get(s);
    }
    for (int i = 0; i < s.length() - 1; i++) {
        if (s.startsWith("++", i)) {
            String t = s.substring(0, i) + "--" + s.substring(i+
2);
            if (!helper(t, winMap)) {
                winMap.put(s, true);
                return true;
            }
        }
    }
    winMap.put(s, false);
    return false;
}
```

Perfact Squares

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.

Tips:

dp[n] indicates that the perfect squares count of the given n, and we have:

```
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[n] = Min{ dp[n - i*i] + 1 },  n - i*i >=0 && i >= 1
```

Code:

```
public class Solution {
    public int numSquares(int n) {
        int[] dp = new int[n + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        dp[0] = 0;
        for(int i = 1; i <= n; ++i) {
            int min = Integer.MAX_VALUE;
            int j = 1;
            while(i - j*j >= 0) {
                min = Math.min(min, dp[i - j*j] + 1);
                ++j;
            }
            dp[i] = min;
        }
        return dp[n];
    }
}
```

Combination Sum IV

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

Example:

nums = [1, 2, 3] target = 4

The possible combination ways are: (1, 1, 1, 1) (1, 1, 2) (1, 2, 1) (1, 3) (2, 1, 1) (2, 2) (3, 1)

Note that different sequences are counted as different combinations.

Therefore the output is 7.

Follow up:

```
What if negative numbers are allowed in the given array?
How does it change the problem?
What limitation we need to add to the question to allow negative
 numbers?
```

Tips:

原题：比较简单的DP，初始化是result[0] = 1; 转移公式是 result[i] += result[i - num]。

Follow up：

加一个visted吧，一个数只能用一次

I think if there are negative numbers in the array, we must add a requirement that each number is only used one time, or either positive number or negative number should be used only one time, otherwise there would be infinite possible combinations. For example, we are given:

{1, -1}, target = 1,

it's obvious to see as long as we choose n 1s and (n-1) -1s, it always sums up to 1, n can be any value >= 1.

I don't think recursion will work if we don't add any extra requirement. Basically, DP is to memorize the results of sub-problems, which is exactly what recursion will re-calculate instead. They are substantially the same. So if one of the them is not working, neither is the other.

For this problem itself, still use the {-1, 1} example, if we can use any number more than one time, it actually equals to we are given {all negative integers, 0, all positive integers}, ie we are given an array with infinite length because -1 can be used to compose all negative integers and similarly for 1. So there will be infinite number of combinations.

Code：

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int[] result = new int[target + 1];
        result[0] = 1;
        for (int i = 1; i <= target; i++) {
            for (int num : nums) {
                if (num <= i) {
                    result[i] += result[i - num];
                }
            }
        }
        return result[target];
    }
}
```

Bomb Enemy

Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0' (the number zero), return the maximum enemies you can kill using one bomb.

The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.

Note that you can only put the bomb at an empty cell.

Example:

For the given grid

```
0 E 0 0
E 0 W E
0 E 0 0
```

return 3. (Placing a bomb at (1,1) kills 3 enemies)

Tips:

O(mn):

I think it is O(m * n). Although there is another for loop k inside for loops of i and j. It just calculates the enemies in advance. In the end, it will traverse this grid once to compute the enemies that are killed.

Every O(mn) algorithm is also O(mn*(m+n))

1. 需要一个row变量，用来记录到下一个墙之前的敌人个数。还需要一个数组col，其中col[j]表示第j列到下一个墙之前的敌人个数。
2. 算法思路是遍历整个数组grid，对于一个位置grid[i][j]，对于水平方向，如果当前位置是开头一个或者前面一个是墙壁，我们开始从当前位置往后遍历，遍历到末尾或者墙的位置停止，计算敌人个数。对于竖直方向也是同样，如果当前位置是开头一个或者上面一个是墙壁，我们开始从当前位置向下遍历，遍历到末尾或者墙的位置停止，计算敌人个数。
3. 有了水平方向和竖直方向敌人的个数，那么如果当前位置是0，表示可以放炸

弹，我们更新结果res即可.

Code:

```
public class Solution {
    public int maxKilledEnemies(char[][] grid) {
        if(grid == null || grid.length == 0 ||  grid[0].length =
= 0) return 0;
        int max = 0;
        int row = 0;
        int[] col = new int[grid[0].length];
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (grid[i][j] == 'W') continue;
                int m = i, n = j;
                if (j == 0 || grid[i][j - 1] == 'W') {
                    row = 0;
                    while (n < grid[0].length && grid[m][n] != '
W') {
                        if (grid[m][n] == 'E') row++;
                        n++;
                    }
                }
                m = i; n = j;
                if (i == 0 || grid[i - 1][j] == 'W') {
                    col[j] = 0;
                    while (m < grid.length && grid[m][n] != 'W')
 {
                        if (grid[m][n] == 'E') col[j]++;
                        m++;
                    }
                }
                if (grid[i][j] == '0') max = Math.max(max, row +
 col[j]);
            }
        }
        return max;
    }
}
```

别人版

```
public class Solution {
    public int maxKilledEnemies(char[][] grid) {
        if(grid == null || grid.length == 0 ||  grid[0].length =
= 0) return 0;
        int max = 0;
        int row = 0;
        int[] col = new int[grid[0].length];
        for(int i = 0; i<grid.length; i++){
            for(int j = 0; j<grid[0].length;j++){
                if(grid[i][j] == 'W') continue;
                if(j == 0 || grid[i][j-1] == 'W'){
                     row = killedEnemiesRow(grid, i, j);
                }
                if(i == 0 || grid[i-1][j] == 'W'){
                     col[j] = killedEnemiesCol(grid,i,j);
                }
                if(grid[i][j] == '0'){
                    max = (row + col[j] > max) ? row + col[j] :
max;
                }
            }

        }

        return max;
    }

    //calculate killed enemies for row i from column j
    private int killedEnemiesRow(char[][] grid, int i, int j){
        int num = 0;
        while(j <= grid[0].length-1 && grid[i][j] != 'W'){
            if(grid[i][j] == 'E') num++;
            j++;
        }
        return num;
    }
    //calculate killed enemies for  column j from row i
    private int killedEnemiesCol(char[][] grid, int i, int j){
        int num = 0;
        while(i <= grid.length -1 && grid[i][j] != 'W'){
            if(grid[i][j] == 'E') num++;
            i++;
        }
        return num;
    }
}
```

Remove K Digits

Given a non-negative integer num represented as a string, remove k digits from the number so that the new number is the smallest possible.

Note:

1. The length of num is less than 10002 and will be ≥ k.
2. The given num does not contain any leading zero.

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new
 number 1219 which is the smallest.
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note th
at the output must not contain leading zeroes.
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is lef
t with nothing which is 0.
```

Tips:

首先这是一个贪心问题，即我们可以将问题转化为，一个长度为N的数字里面，删除哪个数可以使得数变得最小。

我们从头开始找，找到第一个下降的数，如 1234553，那么最后一个3前面的5就是，删除它得到的数字是最小的。

Code:

```
public class Solution {
    public String removeKdigits(String num, int k) {
        int digits = num.length() - k;
        char[] stk = new char[num.length()];
        int top = 0;
        // k keeps track of how many characters we can remove
        // if the previous character in stk is larger than the c
urrent one
        // then removing it will get a smaller number
        // but we can only do so when k is larger than 0
        for (int i = 0; i < num.length(); ++i) {
            char c = num.charAt(i);
            while (top > 0 && stk[top-1] > c && k > 0) {
                top -= 1;
                k -= 1;
            }
            stk[top++] = c;
        }
        // find the index of first non-zero digit
        int idx = 0;
        while (idx < digits && stk[idx] == '0') idx++;
        return idx == digits? "0": new String(stk, idx, digits -
 idx);
    }
}
```

Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers (h, k), where h is the height of the person and k is the number of people in front of this person who have a height greater than or equal to h. Write an algorithm to reconstruct the queue.

Note:

The number of people is less than 1,100.

Example

```
Input:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
```

Tips：

1. Pick out tallest group of people and sort them in a subarray (S).
2. Since there's no other groups of people taller than them, therefore each guy's index will be just as same as his k value.
3. For 2nd tallest group (and the rest), insert each one of them into (S) by k value. So on and so forth.

E.g.

```
input: [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
subarray after step 1: [[7,0], [7,1]]
subarray after step 2: [[7,0], [6,1], [7,1]]
```

Code:

```
public class Solution {
    public int[][] reconstructQueue(int[][] people) {
        //pick up the tallest guy first
        //when insert the next tall guy, just need to insert him
 into kth position
        //repeat until all people are inserted into list
        Arrays.sort(people,new Comparator<int[]>(){
           @Override
           public int compare(int[] o1, int[] o2){
               return o1[0]!=o2[0]?-o1[0]+o2[0]:o1[1]-o2[1];
           }
        });
        List<int[]> res = new LinkedList<>();
        for(int[] cur : people){
            if(cur[1]>=res.size())
                res.add(cur);
            else
                res.add(cur[1],cur);
        }
        return res.toArray(new int[people.length][]);
    }
}
```

Paint Fence

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note:

n and k are non-negative integers.

Tips:

注意题目的意思是不能超过两个相邻的颜色一致。 这种方案总数问题很多都是用dp。 因为超过相邻两个颜色一致，即不能三个颜色一致,那么x的涂色不能和前一个一致 | | 不能和前前个涂色一致。

即f[x] = f[x - 1] K - 1 + f[x - 2] k - 1; 除了递推，还要考虑base 情况。 如果n 或者k 任意一个为0， 那么f[x] ＝ 0。 如果n == 1, 那么就是k。

时间复杂度： O(n)空间复杂度： O(n)

改进：空间复杂度O(1)

Code:

```
public class Solution {
    public int numWays(int n, int k) {
        int[] f = new int[n + 1];
        if (n == 0 || k == 0) {
            return 0;
        }
        if (n == 1) {
            return k;
        }
        f[0] = k;
        f[1] = k * k;
        for (int i = 2; i < n; i++) {
            f[i] = f[i - 1] * (k - 1) + f[i - 2] * (k - 1);
        }
        return f[n - 1];
    }
}
```

改进：

```
public class Solution {
    public int numWays(int n, int k) {
        int[] f = new int[3];
        if (n == 0 || k == 0) {
            return 0;
        }
        if (n == 1) {
            return k;
        }
        f[0] = k;
        f[1] = k * k;
        for (int i = 2; i < n; i++) {
            f[i % 3] = f[(i - 1) % 3] * (k - 1) + f[(i - 2) % 3]
 * (k - 1);
        }
        return f[(n - 1) % 3];
    }
}
```

Max Sum of Rectangle No Larger Than K

Given a non-empty 2D matrix matrix and an integer k, find the max sum of a rectangle in the matrix such that its sum is no larger than k.

Example:

```
Given matrix = [
  [1,  0, 1],
  [0, -2, 3]
]
k = 2
```

The answer is 2. Because the sum of rectangle [[0, 1], [-2, 3]] is 2 and 2 is the max number no larger than k (k = 2).

Note:

The rectangle inside the matrix must have an area > 0.

What if the number of rows is much larger than the number of columns?

Tips:

题意：求矩阵里K*K的矩阵里数和的最大值。

朴素的思想为，枚举起始行，枚举结束行，枚举起始列，枚举终止列。。。。。O(m^2 * n^2)

这里用到一个技巧就是，进行求和时，我们可以把二维的合并成一维，然后就变为求一维的解。

比如对于矩阵：

[1, 0, 1], [0, -2, 3]

进行起始行为0，终止行为1时，可以进行列的求和，即[1, -2, 4]中不超过k的最大值。

求和的问题解决完，还有一个是不超过k. 这里我参考了 https://leetcode.com/discuss/109705/java-binary-search-solution-time-complexity-min-max-log-max 的方法

使用了二分搜索。对于当前的和为sum，我们只需要找到一个最小的数x，使得 sum – k <=x，这样可以保证sum – x <=k。

这里需要注意，当行远大于列的时候怎么办呢？转换成列的枚举 即可。

在代码实现上，我们只需要让 m 永远小于 n即可。这样复杂度总是为O(m^2*n*log n)

```
/* first  consider the situation matrix is 1D
    we can save every sum of 0~i(0<=i<len) and binary search pre
vious sum to find
    possible result for every index, time complexity is O(NlogN)
.

    so in 2D matrix, we can sum up all values from row i to row
j and create a 1D array
    to use 1D array solution.

    If col number is less than row number, we can sum up all val
ues from col i to col j
    then use 1D array solution.
*/
```

Code:

```
public int maxSumSubmatrix(int[][] matrix, int target) {
    int row = matrix.length;
    if(row==0)return 0;
    int col = matrix[0].length;
    int m = Math.min(row,col);
    int n = Math.max(row,col);
    //indicating sum up in every row or every column
    boolean colIsBig = col>row;
    int res = Integer.MIN_VALUE;
    for(int i = 0;i<m;i++){
        int[] array = new int[n];
        // sum from row j to row i
        for(int j = i;j>=0;j--){
            int val = 0;
            TreeSet<Integer> set = new TreeSet<Integer>();
            set.add(0);
            //traverse every column/row and sum up
            for(int k = 0;k<n;k++){
                array[k]=array[k]+(colIsBig?matrix[j][k]:matrix[
k][j]);
                val = val + array[k];
                //use  TreeMap to binary search previous sum to
get possible result
                Integer subres = set.ceiling(val-target);
                if(null!=subres){
                    res=Math.max(res,val-subres);
                }
                set.add(val);
            }
        }
    }
    return res;
}
```

Burst Balloons

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] *nums[i]* nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:

(1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.

(2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

Example:

Given [3, 1, 5, 8]

Return 167

```
nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
```

coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

Tips:

这道题提出了一种打气球的游戏，每个气球都对应着一个数字，我们每次打爆一个气球，得到的金币数是被打爆的气球的数字和其两边的气球上的数字相乘，如果旁边没有气球了，则按1算，以此类推，求能得到的最多金币数。像这种求极值问题，我们一般都要考虑用动态规划Dynamic Programming来做，我们维护一个二维动态数组dp，其中dp[i][j]表示打爆区间[i,j]中的所有气球能得到的最多金币。题目中说明了边界情况，当气球周围没有气球的时候，旁边的数字按1算，这样我们可以在原数组两边各填充一个1，这样方便于计算。这道题的最难点就是找递归式，如下所示：

dp[i][j] = max(dp[i][j], nums[i - 1]*nums[k]*nums[j + 1] + dp[i][k - 1] + dp[k + 1][j]) (i ≤ k ≤ j)

有了递推式，我们可以写代码，我们其实只是更新了dp数组的右上三角区域，我们最终要返回的值存在dp[1][n]中，其中n是两端添加1之前数组nums的个数。

Reverse thinking. Like I said the coins you get for a balloon does not depend on the balloons already burst. Therefore, instead of divide the problem by the first balloon to burst, we divide the problem by the last balloon to burst.

Why is that? Because only the first and last balloons we are sure of their adjacent balloons before hand!

For the first we have nums[i-1]*nums[i]*nums[i+1] for the last we have nums[-1]*nums[i]*nums[n].

OK. Think about n balloons if i is the last one to burst, what now?

We can see that the balloons is again separated into 2 sections. But this time since the balloon i is the last balloon of all to burst, the left and right section now has well defined boundary and do not affect each other! Therefore we can do either recursive method with memoization or dp.

Final

Here comes the final solutions. Note that we put 2 balloons with 1 as boundaries and also burst all the zero balloons in the first round since they won't give any coins.

The algorithm runs in **O(n^3)** which can be easily seen from the 3 loops in dp solution.

Code:

DP:

public class Solution { public int maxCoins(int[] nums) { int balloons[] = new int[nums.length + 2]; for (int i = 0; i < nums.length; i++) { balloons[i + 1] = nums[i]; } int n = nums.length + 2; balloons[0] = balloons[n - 1] = 1; int[][] dp = new int[n][n]; for (int k = 2; k < n; k++) { for (int left = 0; left < n - k; left++) { int right = left + k; for (int i = left + 1; i < right; i++) { dp[left][right] = Math.max(dp[left][right], balloons[left]*balloons[i]* balloons[right] + dp[left][i] + dp[i][right]); } } } return dp[0][n - 1]; } }

Java D&C with Memoization:

```
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] memo = new int[n][n];
    return burst(memo, nums, 0, n - 1);
}

public int burst(int[][] memo, int[] nums, int left, int right)
{
    if (left + 1 == right) return 0;
    if (memo[left][right] > 0) return memo[left][right];
    int ans = 0;
    for (int i = left + 1; i < right; ++i)
        ans = Math.max(ans, nums[left] * nums[i] * nums[right]
        + burst(memo, nums, left, i) + burst(memo, nums, i, righ
t));
    memo[left][right] = ans;
    return ans;
}
```

Patching Array

Given a sorted positive integer array nums and an integer n, add/patch elements to the array such that any number in range [1, n] inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.

```
Example 1:
nums = [1, 3], n = 6
Return 1.
```

Combinations of nums are [1], [3], [1,3], which form possible sums of: 1, 3, 4.

Now if we add/patch 2 to nums, the combinations are: [1], [2], [3], [1,3], [2,3], [1,2,3]. Possible sums are 1, 2, 3, 4, 5, 6, which now covers the range [1, 6].

So we only need 1 patch.

```
Example 2:
nums = [1, 5, 10], n = 20
Return 2.
The two patches can be [2, 4].

Example 3:
nums = [1, 2, 2], n = 5
Return 0.
```

Tips:

道题给我们一个有序的正数数组nums，又给了我们一个正整数n，问我们最少需要给nums加几个数字，使其能组成[1,n]之间的所有数字，注意数组中的元素不能重复使用，否则的话只有要有1，就能组成所有的数字了

我们定义一个变量miss，用来表示[0,n]之间最小的不能表示的值，那么初始化为1，为啥不为0呢，因为n=0没啥意义，直接返回0了。那么此时我们能表示的范围是[0, miss)，表示此时我们能表示0到miss-1的数，如果此时的num <= miss，那么我

们可以把我们能表示数的范围扩大到[0, miss+num)，如果num>miss，那么此时我们需要添加一个数，为了能最大限度的增加表示数范围，我们加上miss它本身，以此类推直至遍历完整个数组，我们可以得到结果。下面我们来举个例子说明：

给定nums = [1, 2, 4, 11, 30], n = 50，我们需要让[0, 50]之间所有的数字都能被nums中的数字之和表示出来。

首先使用1, 2, 4可能表示出0到7之间的所有数，表示范围为[0, 8)，但我们不能表示8，因为下一个数字11太大了，所以我们要在数组里加上一个8，此时能表示的范围是[0, 16)，那么我们需要插入16吗，答案是不需要，因为我们数组有1和4，可以组成5，而下一个数字11，加一起能组成16，所以有了数组中的11，我们此时能表示的范围扩大到[0, 27)，但我们没法表示27，因为30太大了，所以此时我们给数组中加入一个27，那么现在能表示的范围是[0, 54)，已经满足要求了，我们总共添加了两个数8和27，所以返回2即可。

Explanation

Let miss be the smallest sum in [0,n] that we might be missing. Meaning we already know we can build all sums in [0,miss). Then if we have a number num <= miss in the given array, we can add it to those smaller sums to build all sums in [0,miss+num). If we don't, then we must add such a number to the array, and it's best to add miss itself, to maximize the reach.

Example:

Let's say the input is nums = [1, 2, 4, 13, 43] and n = 100. We need to ensure that all sums in the range [1,100] are possible.

Using the given numbers 1, 2 and 4, we can already build all sums from 0 to 7, i.e., the range [0,8). But we can't build the sum 8, and the next given number (13) is too large. So we insert 8 into the array. Then we can build all sums in [0,16).

Do we need to insert 16 into the array? No! We can already build the sum 3, and adding the given 13 gives us sum 16. We can also add the 13 to the other sums, extending our range to [0,29).

And so on. The given 43 is too large to help with sum 29, so we must insert 29 into our array. This extends our range to [0,58). But then the 43 becomes useful and expands our range to [0,101). At which point we're done.

Code:

```
public class Solution {
    // note the array is sorted
    public int minPatches(int[] nums, int n) {
        int m = nums.length, miss = 1, patch = 0;
        for (int i = 0; miss > 0 && miss <= n;) { //still run lo
op @ i >= m
            if (i < m && nums[i] <= miss)
                miss += nums[i++];
            else {
                miss += miss; //path an element with value miss
                patch++;
            }
        }
        return patch;
    }
}
```

Russian Doll Envelopes

You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:

Given envelopes = [[5,4],[6,4],[6,7],[2,3]], the maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

Tips:

给定一些信封的宽和长，当且仅当信封x的宽和长均小于另一个信封y时，x可以装入y，求最多可以嵌套的装几个？

求LIS可以直接简单的dp，设dp[i]为以i为结尾的LIS长度，则dp[i] = max(0,dp[j] | j<i && A[j] < A[i]) + 1

复杂度为O(n^2)，但可以优化到**O(nlogn)**，排序然后二分。

做二分法的时候要注意Arrays.binarySearch方法：

```
Arrays.binarySearch() returns ( - insertion_index - 1) in cases
where the element was not found in the array.
Initially the dp array is all zeroes.
For all zeroes array the insertion index for any element greater
 than zero is equal to the length of the array (dp.length in thi
s case).
This means that the number needs to be added to the end of the a
rray to keep the array sorted.
```

Sort the array. Ascend on width and descend on height if width are same.

Find the longest increasing subsequence based on height.

Since the width is increasing, we only need to consider height.

[3, 4] cannot contains [3, 3], so we need to put [3, 4] before [3, 3] when sorting otherwise it will be counted as an increasing number if the order is [3, 3], [3, 4]

Code:

O(n^2) 简单dp

```
public class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        if(envelopes == null || envelopes.length == 0 || envelop
es[0] == null || envelopes[0].length != 2) return 0;
        Arrays.sort(envelopes, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                if (a[0] == b[0]) return b[1] - a[1];
                else return a[0] - b[0];
            }
        });
        int dp[] = new int[envelopes.length];
        for (int i = 0; i < envelopes.length; i++) dp[i] = 1;
        int max = 1;
        for (int i = 0; i < envelopes.length; i++) {
            for (int j = 0; j < i; j++) {
                if (envelopes[i][1] > envelopes[j][1]) dp[i] = M
ath.max(dp[j] + 1, dp[i]);
            }
            max = Math.max(max, dp[i]);
        }
        return max;
    }
}
```

O(nlogn) 二分

```
public class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        if(envelopes == null || envelopes.length == 0
           || envelopes[0] == null || envelopes[0].length != 2)
            return 0;
        Arrays.sort(envelopes, new Comparator<int[]>(){
            public int compare(int[] arr1, int[] arr2){
                if(arr1[0] == arr2[0])
                    return arr2[1] - arr1[1];
                else
                    return arr1[0] - arr2[0];
           }
        });
        int dp[] = new int[envelopes.length];
        int len = 0;
        for(int[] envelope : envelopes){
            int index = Arrays.binarySearch(dp, 0, len, envelope
[1]);
            if(index < 0)
                index = -(index + 1);
            dp[index] = envelope[1];
            if(index == len)
                len++;
        }
        return len;
    }
}
```

Word Break

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. You may assume the dictionary does not contain duplicate words.

For example, given s = "leetcode", dict = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".

Code:

```
    public class Solution {
        public boolean wordBreak(String s, Set<String> wordDict)
 {
            boolean[] result = new boolean[s.length() + 1];
            result[0] = true;
            for (int i = 1; i <= s.length(); i++) {
                for (int j = 0; j < i; j++) {
                    String sub = s.substring(j,i);
                    result[i] = result[i] || (result[j] && wordD
ict.contains(sub));
                }
            }
            return result[s.length()];
        }
    }
```

Word Break II

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, add spaces in s to construct a sentence where each word is a valid dictionary word. You may assume the dictionary does not contain duplicate words.

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"

Tips:

如果要返回所有解法，则需要用到DFS。而直接DFS会超时，因为每一个位置都要dfs下去看有没有解。所以用hashmap来记录已经得到过的结果。把s分为左边和右边，如果左边在字典中，就DFS右边，DFS完返回一个sublist，然后result里加入左边的word和右边的sublist。需要注意的是，如果右边为空，则需要加入“”，不然结果会为空。

Complexity: O(len(wordDict) ^ len(s / minWordLenInDict)) There're len(wordDict) possibilities for each cut

Code：

```
    public class Solution {
        public List<String> wordBreak(String s, List<String> wordDict) {
            return DFS(s, wordDict, new HashMap<String, List<String>>());
        }
        private List<String> DFS(String s, List<String> wordDict, HashMap<String, List<String>> map) {
            if (map.containsKey(s)) return map.get(s);
            List<String> res = new ArrayList<>();
            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;
        }
    }
```

DP:

DP + DFS. Very similar to Word Break I, but instead of using a boolean dp array, I used an array of Lists to maintain all of the valid start positions for every end position. Then just do classic backtracking to find all solutions. The time complexity is **O(n*m*) + *O(n* number of solutions)**, where n is the length of the input string, m is the length of the longest word in the dictionary. The run time was 6ms. It is very efficient because DP is used to find out all the valid answers, and no time is wasted on doing the backtracking.

```
public List<String> wordBreak(String s, Set<String> wordDict) {
    List<Integer>[] starts = new List[s.length() + 1]; // valid
start positions
    starts[0] = new ArrayList<Integer>();

    int maxLen = getMaxLen(wordDict);
    for (int i = 1; i <= s.length(); i++) {
        for (int j = i - 1; j >= i - maxLen && j >= 0; j--) {
            if (starts[j] == null) continue;
            String word = s.substring(j, i);
            if (wordDict.contains(word)) {
                if (starts[i] == null) {
                    starts[i] = new ArrayList<Integer>();
                }
                starts[i].add(j);
            }
        }
    }

    List<String> rst = new ArrayList<>();
    if (starts[s.length()] == null) {
        return rst;
    }

    dfs(rst, "", s, starts, s.length());
    return rst;
}


private void dfs(List<String> rst, String path, String s, List<I
nteger>[] starts, int end) {
    if (end == 0) {
        rst.add(path.substring(1));
        return;
    }

    for (Integer start: starts[end]) {
        String word = s.substring(start, end);
        dfs(rst, " " + word + path, s, starts, start);
    }
}

private int getMaxLen(Set<String> wordDict) {
    int max = 0;
    for (String s : wordDict) {
        max = Math.max(max, s.length());
    }
    return max;
}
```

Brute force:**(2^n)**

```
    public class Solution {
        public List<String> wordBreak(String s, Set<String> word
Dict) {
            List<String> res = new ArrayList<String>();
            if(s == null || s.length() == 0) {
                return res;
            }
            dfshelper(res , "" , 0 , wordDict , s);
            return res;
        }

        protected void dfshelper(List<String> res , String path
, int start , Set<String> wordDict , String s) {
            if(start == s.length()) {
                res.add(path);
                return;
            }
            for(int i = start ; i < s.length() ; i++) {
                String t = s.substring(start , i+1);
                if(wordDict.contains(t)) {
                    if(i+1 != s.length()) {
                        path += t + " ";
                        dfshelper(res , path , i+1 , wordDict ,
s);
                        path = path.substring(0 , path.length()
- t.length() - 1);
                    }else {
                        path += t;
                        dfshelper(res , path , i+1 , wordDict ,
s);
                        path = path.substring(0 , path.length()
- t.length());
                    }
                }
            }
        }
    }
```

Ones and Zeroes

In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.

Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used at most once.

Note:

1. The given numbers of 0s and 1s will both not exceed 100
2. The size of given string array won't exceed 600.

Example 1:

```
Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
Output: 4
```

Explanation: This are totally 4 strings can be formed by the using of 5 0s and 3 1s, which are “10,”0001”,”1”,”0”

Example 2:

```
Input: Array = {"10", "0", "1"}, m = 1, n = 1
Output: 2
```

Explanation: You could form "10", but then you'd have nothing left. Better form "0" and "1".

Tips:

Time Complexity: **O(kl + kmn)**, Space Complexity: **O(mn)**

DP, 公式为dp[i][j] = Math.max(1 + dp[i-count[0]][j-count[1]], dp[i][j])。

注意for (int i=m;i>=count[0];i--)，必须从后往前dp，因为从前往后的话会有重复使用字符串的问题。

You can try a simple example yourself, basically the difference is, if you loop in the forward way, then you may use a item more than once, for example Array = {"10", ...}, m = 5, n = 3, for the first element, you dp[1][1] = 1, then dp[2][2] = 2, dp[3][3] = 3. If the problem specifies you can use the strings more than once, then it would the algorithm to do it. **Code:**

```
public class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        int[][] dp = new int[m + 1][n + 1];
        for (String s : strs) {
            int[] count = count(s);
            for (int i = m; i >= count[0]; i--) {
                for (int j = n; j >= count[1]; j--) {
                    dp[i][j] = Math.max(dp[i][j], 1 + dp[i - cou
nt[0]][j - count[1]]);
                }
            }
        }
        return dp[m][n];
    }
    private int[] count(String s) {
        int[] res = new int[2];
        for (char c : s.toCharArray()) {
            if (c == '0') res[0]++;
            else res[1]++;
        }
        return res;
    }
}
```

Predict the Winner

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

Example 1:

```
Input: [1, 5, 2]
Output: False
Explanation: Initially, player 1 can choose between 1 and 2.
If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) a
nd 5. If player 2 chooses 5, then player 1 will be left with 1 (
or 2).
So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
Hence, player 1 will never be the winner and you need to return
False.
```

Example 2:

```
Input: [1, 5, 233, 7]
Output: True
Explanation: Player 1 first chooses 1. Then player 2 have to cho
ose between 5 and 7. No matter which number player 2 choose, pla
yer 1 can choose 233.
Finally, player 1 has more score (234) than player 2 (12), so yo
u need to return True representing player1 can win.
```

Note:

1. 1 <= length of the array <= 20.
2. Any scores in the given array are non-negative integers and will not exceed

10,000,000.
3. If the scores of both players are equal, then player 1 is still the winner.

Tips:

第一种解法是**divide and conquer**, 用一个helper函数，注意最后的返回是>=0,而在recursive的过程中，第二个选手选的数字被巧妙地减掉了。优化是加一个memory记录已经选掉的部分。这个减法太精髓了。

So assuming the sum of the array it SUM, so eventually player1 and player2 will split the SUM between themselves. For player1 to win, he/she has to get more than what player2 gets. If we think from the prospective of one player, then what he/she gains each time is a plus, while, what the other player gains each time is a minus. Eventually if player1 can have a > 0 total, player1 can win.

Helper function simulate this process. In each round: if e==s, there is no choice but have to select nums[s] otherwise, this current player has 2 options: --> nums[s]-helper(nums,s+1,e): this player select the front item, leaving the other player a choice from s+1 to e --> nums[e]-helper(nums,s,e-1): this player select the tail item, leaving the other player a choice from s to e-1 Then take the max of these two options as this player's selection, return it.

第二种解法是**DP**：思路其实是一致的， 注意初始化问题即可。dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1]);

The dp[i][j] saves how much more scores that the first-in-action player will get from i to j than the second player. First-in-action means whomever moves first. You can still make the code even shorter but I think it looks clean in this way.

Code：

优化：

```
public class Solution {
    public boolean PredictTheWinner(int[] nums) {
        return helper(nums, 0, nums.length - 1, new Integer[nums
.length][nums.length]) >= 0;

    }
    private int helper(int[] nums, int s, int e, Integer[][] mem
) {
        if (mem[s][e] == null) {
            mem[s][e] = s == e ? nums[s] : Math.max(nums[s] - he
lper(nums, s+1, e, mem), nums[e] - helper(nums, s, e-1, mem));
        }
        return mem[s][e];
    }
}
```

原始

```
public class Solution {
    public boolean PredictTheWinner(int[] nums) {
        return helper(nums, 0, nums.length-1)>=0;
    }
    private int helper(int[] nums, int s, int e){
        return s==e ? nums[e] : Math.max(nums[e] - helper(nums,
s, e-1), nums[s] - helper(nums, s+1, e));
    }
}
```

DP：

```
public class Solution {
    public boolean PredictTheWinner(int[] nums) {
        int n = nums.length;
        int[][] dp = new int[n][n];
        for (int i = 0; i < n; i++) dp[i][i] = nums[i];
        for (int len = 1; len < n; len++) {
            for (int i = 0; i < n - len; i++) {
                int j = i + len;
                dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums
[j] - dp[i][j - 1]);
            }
        }
        return dp[0][n - 1] >= 0;
    }
}
```

Encode String with Shortest Length

Given a non-empty string, encode the string such that its encoded length is the shortest.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times.

Note:

1. k will be a positive integer and encoded string will not be empty or have extra space.
2. You may assume that the input string contains only lowercase English letters. The string's length is at most 160.
3. If an encoding process does not make the string shorter, then do not encode it. If there are several solutions, return any of them is fine.

Example 1:

```
Input: "aaa"
Output: "aaa"
Explanation: There is no way to encode it such that it is shorte
r than the input string, so we do not encode it.
```

Example 2:

```
Input: "aaaaa"
Output: "5[a]"
Explanation: "5[a]" is shorter than "aaaaa" by 1 character.
```

Example 3:

```
Input: "aaaaaaaaaa"
Output: "10[a]"
Explanation: "a9[a]" or "9[a]a" are also valid solutions, both o
f them have the same length = 5, which is the same as "10[a]".
```

Example 4:

```
Input: "aabcaabcd"
Output: "2[aabc]d"
Explanation: "aabc" occurs twice, so one answer can be "2[aabc]d
".
```

Example 5:

```
Input: "abbbabbbcabbbabbbc"
Output: "2[2[abbb]c]"
Explanation: "abbbabbbc" occurs twice, but "abbbabbbc" can also
be encoded to "2[abbb]c", so one answer can be "2[2[abbb]c]".
```

Tips:

DP, 公式为

```
dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j])
```

注意每个循环内要找这个字符串自己是不是可以被简写。

复杂度: **O(n^4)** replaceAll 需要O(n)

Find pattern可以用KMP，算法附在后面

Codes：

```
public class Solution {
    public String encode(String s) {
        String[][] dp = new String[s.length()][s.length()];
        for (int l = 0; l < s.length(); l++) {
            for (int i = 0; i < s.length() - l; i++) {
                int j = i + l;
                String substr = s.substring(i, j + 1);
                if (l < 4) dp[i][j] = substr;
                else {
                    dp[i][j] = substr;
                    for (int k = i; k < j; k++) {
                        if ((dp[i][k] + dp[k + 1][j]).length() <
 dp[i][j].length()) dp[i][j] = dp[i][k] + dp[k + 1][j];
                    }
                    for (int k = 0; k < l; k++) {
                        String repeatStr = substr.substring(0, k
 + 1);
                        if (repeatStr != null && substr.length()
 % repeatStr.length() == 0
                            && substr.replaceAll(repeatStr, "").
length() == 0) {
                            String ss = substr.length() / repeat
Str.length() + "[" + dp[i][i+k] + "]";
                            if(ss.length() < dp[i][j].length())
{
                            dp[i][j] = ss;
                            }
                        }
                    }
                }
            }
        }
        return dp[0][s.length()-1];
    }
}
```

KMP：

```
public class Solution {
    public String encode(String s) {
        if(s == null || s.length() <= 4) return s;

        int len = s.length();

        String[][] dp = new String[len][len];

        // iterate all the length, stay on the disgnose of the d
p matrix
        for(int l = 0; l < len; l ++) {
            for(int i = 0; i < len - l; i ++) {
                int j = i + l;
                String substr = s.substring(i, j + 1);
                dp[i][j] = substr;
                if(l < 4) continue;

                for(int k = i; k < j; k ++) {
                    if(dp[i][k].length() + dp[k + 1][j].length()
 < dp[i][j].length()) {
                        dp[i][j] = dp[i][k] + dp[k + 1][j];
                    }
                }

                String pattern = kmp (substr);
                if(pattern.length() == substr.length()) continue
; // no repeat pattern found
                String patternEncode = substr.length() / pattern
.length() + "[" + dp[i][i + pattern.length() - 1] + "]";
                if(patternEncode.length() < dp[i][j].length()) {
                    dp[i][j] = patternEncode;
                }
            }
        }

        return dp[0][len - 1];
    }

    private String kmp (String s) {
        int len = s.length();
        int[] LPS = new int[len];
        int i = 1, j = 0;
        LPS[0] = 0;
        while(i < len) {
            if(s.charAt(i) == s.charAt(j)) {
                LPS[i ++] = ++ j;
            } else if(j == 0) {
                LPS[i ++] = 0;
            } else {
                j = LPS[j - 1];
            }
        }

        int patternLen = len - LPS[len - 1];
        if(patternLen != len && len % patternLen == 0) {
            return s.substring(0, patternLen);
        } else {
            return s;
        }
    }
}
```

Find Permutation

By now, you are given a secret signature consisting of character 'D' and 'I'. 'D' represents a decreasing relationship between two numbers, 'I' represents an increasing relationship between two numbers. And our secret signature was constructed by a special integer array, which contains uniquely all the different number from 1 to n (n is the length of the secret signature plus 1). For example, the secret signature "DI" can be constructed by array [2,1,3] or [3,1,2], but won't be constructed by array [3,2,4] or [2,1,3,4], which are both illegal constructing special string that can't represent the "DI" secret signature.

On the other hand, now your job is to find the lexicographically smallest permutation of [1, 2, ... n] could refer to the given secret signature in the input.

Example 1:

```
Input: "I"
Output: [1,2]
Explanation: [1,2] is the only legal initial spectial string can
 construct secret signature "I", where the number 1 and 2 constr
uct an increasing relationship.
```

Example 2:

```
Input: "DI"
Output: [2,1,3]
Explanation: Both [2,1,3] and [3,1,2] can construct the secret s
ignature "DI",
but since we want to find the one with the smallest lexicographi
cal permutation, you need to output [2,1,3]
```

Note:

1. The input string will only contain the character 'D' and 'I'.
2. The length of input string is a positive integer and will not exceed 10,000

Tips: Greedy, **O(n)**

```
设置一个min。
1. 如果是I，就res[i++] = min++;
2. 如果是D，就遍历到不是D的地方，然后从大到小加数字。
```

Code:

```
public class Solution {
    public int[] findPermutation(String s) {
        s = s + ".";
        int[] res = new int[s.length()];
        int min = 1, i = 0;
        while (i < res.length) {
            while (s.charAt(i) == 'I') res[i++] = min++;
            int j = i;
            while (s.charAt(j) == 'D') j++;
            for (int k = j; k >= i; k--) res[k] = min++;
            i = j + 1;
        }
        return res;
    }
}
```

Data Stucture

Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- getMin() -- Retrieve the minimum element in the stack.

Example:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> Returns -3.
minStack.pop();
minStack.top();      --> Returns 0.
minStack.getMin();   --> Returns -2.
```

Tips: 需要两个stack，可以判断*stack.peek().equals(minStack.peek())*，之后pop minStack，（注意是equals！！）也可以*minStack.push(Math.min(number, minStack.peek()));* (push x 和min之间的最小值）来免去pop时的判断。

代码：

```
public class MinStack {

    /** initialize your data structure here. */
    Stack<Integer> stack;
    Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<Integer>();
        minStack = new Stack<Integer>();
    }
    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty()) {
            minStack.push(x);
        } else if (minStack.peek() >= x){
            minStack.push(x);
        }
    }

    public void pop() {
        if (stack.peek().equals(minStack.peek())) {
            minStack.pop();
        }
        stack.pop();
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
```

Kth Smallest Element in a Sorted Matrix

Given a n x n matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.

Note that it is the kth smallest element in the sorted order, not the kth distinct element.

Example:

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,
```

return 13. Note: You may assume k is always valid, 1 ≤ k ≤ n2.

Tips:

思路1（Heap）：

- Build a minHeap of elements from the first row.
- Do the following operations k-1 times : Every time when you poll out the root(Top Element in Heap), you need to know the row number and column number of that element(so we can create a Node class here), replace that root with the next element from the same column.

  这种思路下先将第一行加入heap，然后取heap里的最小值，加入矩阵中这个最小值下面的数，要注意的是要新建一个node类，并重写compareTo方法，不然无法获取这个点的坐标。注意在重写compareTo那里一定要public，不然会出错！

思路2（binary search）：

这题我们也可以用二分查找法来做，我们由于是有序矩阵，那么左上角的数字一定是最小的，而右下角的数字一定是最大的，所以这个是我们搜索的范围，然后我们算出中间数字mid，由于矩阵中不同行之间的元素并不是严格有序的，所以我们要在每一行都查找一下mid，我们使用upper_bound，这个函数是查找第一个大于目标数的元素，如果目标数在比该行的尾元素大，则upper_bound返回该行元素的个数，如果目标数比该行首元素小，则upper_bound返回0, 我们遍历完所有的行可以找出中间数是第几小的数，然后k比较，进行二分查找，本解法的整体时间复杂度为**O(nlgn*lgX)**，其中X为最大值和最小值的差值。

思路3（Heap）：

和思路一类似，但是不需要提前加入第一行，而是改为在找到当前最小值后，加入最小值的右边和下边两个元素。需要注意的是这个时候需要判断要加入heap的元素是否已经在别的地方加入到heap里了。所以需要新建n*n的matrix hash[][]来判断。

Code:

解法1：

```
public class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        if (k == n * n) {
            return matrix[n - 1][n - 1];
        }
        PriorityQueue<Node> heap = new PriorityQueue<>();
        for (int i = 0; i < n; i++) {
            heap.add(new Node(0, i, matrix[0][i]));
        }
        for (int i = 0; i < k - 1; i++) {
            Node curt = heap.remove();
            if (curt.x == n - 1) {
                continue;
            }
            heap.add(new Node(curt.x + 1, curt.y, matrix[curt.x
+ 1][curt.y]));
        }
        return heap.remove().val;
    }
}

class Node implements Comparable<Node> {
    int x;
    int y;
    int val;
    public Node(int x, int y, int val) {
        this.x = x;
        this.y = y;
        this.val = val;
    }
    public int compareTo (Node that) {
        return this.val - that.val;
    }
}
```

解法2：

```
public int kthSmallest(int[][] matrix, int k) {
    int m=matrix.length;

    int lower = matrix[0][0];
    int upper = matrix[m-1][m-1];

    while(lower<upper){
        int mid = lower + ((upper-lower)>>1);
        int count = count(matrix, mid);
        if(count<k){
            lower=mid+1;
        }else{
            upper=mid;
        }
    }

    return upper;
}

private int count(int[][] matrix, int target){
    int m=matrix.length;
    int i=m-1;
    int j=0;
    int count = 0;

    while(i>=0&&j<m){
        if(matrix[i][j]<=target){
            count += i+1;
            j++;
        }else{
            i--;
        }
    }

    return count;
}
```

解法3：

```
class Number {
    public int x, y, val;
    public Number(int x, int y, int val) {
        this.x = x;
        this.y = y;
        this.val = val;
    }
}
class NumComparator implements Comparator<Number> {
    public int compare(Number a, Number b) {
        return a.val - b.val;
    }
}

public class Solution {
    /**
     * @param matrix: a matrix of integers
     * @param k: an integer
     * @return: the kth smallest number in the matrix
     */
    private boolean valid(int x, int y, int[][] matrix, boolean[
][] hash) {
        if(x < matrix.length && y < matrix[x].length && !hash[x]
[y]) {
            return true;
        }
        return false;
    }

    int dx[] = {0,1};
    int dy[] = {1,0};
    public int kthSmallest(int[][] matrix, int k) {
        // write your code here
        if (matrix == null || matrix.length == 0) {
            return -1;
        }
        if (matrix.length * matrix[0].length < k) {
            return -1;
        }
        PriorityQueue<Number> heap = new PriorityQueue<Number>(k
, new NumComparator());
        boolean[][] hash = new boolean[matrix.length][matrix[0].
length];
        heap.add(new Number(0, 0, matrix[0][0]));
        hash[0][0] = true;

        for (int i = 0; i < k - 1; i ++) {
            Number smallest = heap.poll();
            for (int j = 0; j < 2; j++) {
                int nx = smallest.x + dx[j];
                int ny = smallest.y + dy[j];
                if (valid(nx, ny, matrix, hash)) {
                    hash[nx][ny] = true;
                    heap.add(new Number(nx, ny, matrix[nx][ny]))
;
                }
            }
        }
        return heap.peek().val;
    }
}
```

Find K Pairs with Smallest Sums

You are given two integer arrays nums1 and nums2 sorted in ascending order and an integer k.

Define a pair (u,v) which consists of one element from the first array and one element from the second array.

Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.

Example 1:

```
Given nums1 = [1,7,11], nums2 = [2,4,6],  k = 3

Return: [1,2],[1,4],[1,6]

The first 3 pairs are returned from the sequence:
[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

Example 2:

```
Given nums1 = [1,1,2], nums2 = [1,2,3],  k = 2

Return: [1,1],[1,1]

The first 2 pairs are returned from the sequence:
[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
```

Example 3:

```
Given nums1 = [1,2], nums2 = [3],  k = 3

Return: [1,3],[2,3]

All possible pairs are returned from the sequence:
[1,3],[2,3]
```

Tips:

- 和Kth Smallest Element in a Sorted Matrix几乎一样，唯一的区别就是这题给了两个array而那题给了一个矩阵。
- 想象一下，把这题的两个数组变成num1.length * num2.length的矩阵，例如将exp1变为

  ```
  matrix =
      [ 1*2, 1*4 , 1*6 ],
      [ 7*2, 7*4 , 7*6 ],
      [11*2, 11*4, 11*6]
  ```

- 与那题一样，先将第一行加入heap，然后取heap里的最小值，加入矩阵中这个最小值下面的数，并在result里添加当前最小值。（注意添加的是值不是坐标哟）
- 仍然要注意的是要新建一个node类，并重写compareTo方法，不然无法获取这个点的坐标。注意在重写compareTo那里一定要public，不然会出错！

Code:

```
public class Solution {
    public List<int[]> kSmallestPairs(int[] nums1, int[] nums2,
int k) {
        List<int[]> result = new ArrayList<>();
        PriorityQueue<Node> heap = new PriorityQueue<>();
        if (nums1 == null || nums1.length == 0 || nums2 == null
|| nums2.length == 0 || k <= 0) {
            return result;
        }
        for (int i = 0; i < nums2.length; i++) {
            heap.add(new Node(0, i, nums1[0] + nums2[i]));
        }
        for (int i = 0; i < Math.min(k, nums1.length * nums2.len
gth); i++) {
            Node curt = heap.remove();
            int[] temp = {nums1[curt.x], nums2[curt.y]};
            result.add(temp);
            if (curt.x == nums1.length - 1) {
```

```
                continue;
            }
            heap.add(new Node(curt.x + 1, curt.y, nums1[curt.x +
 1] + nums2[curt.y]));
        }
        return result;
    }
    class Node implements Comparable<Node>{
        int x;
        int y;
        int val;
        public Node(int x, int y, int val) {
            this.x = x;
            this.y = y;
            this.val = val;
        }
        public int compareTo(Node that) {
            return this.val - that.val;
        }
    }
}
```

Kth Smallest Element in a BST

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

Note: You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up: What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Hint:

Try to utilize the property of a BST. What if you could modify the BST node's structure? The optimal runtime complexity is O(height of BST).

Tips:

The first solution is actually the worst, but good for the follow up. it is an O(n^2) worst, O(nlogn) average case algorithm.

the second one is O(n) but it doesn't terminate when solution is found, instead it continues to scan the remaining nodes which is a waste without question.

The third one is a traditional iterative in order traversal and should solve the problem.

Follow Up:

Hi I think the first solution is for the follow up question. To find the kth element in the BST, we'll definitely need o(k), that is o(n) complexity. But if we are allowed to modify the node structure of BST, we can simply store the count value in every node. And thus we can achieve the search in o(logn)

Code:

解法1：Binary Search (dfs): most preferable

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        int count = countNodes(root.left);
        if (count == k - 1){
            return root.val;
        } else if (count + 1 < k) {
            return kthSmallest(root.right, k - count - 1);
        } else {
            return kthSmallest(root.left, k);
        }
    }
    private int countNodes(TreeNode node) {
        if (node == null) {
            return 0;
        }
        return countNodes(node.left) + countNodes(node.right) +
1;
    }
}
```

解法2：DFS in-order recursive:

```
// better keep these two variables in a wrapper class
private static int number = 0;
private static int count = 0;

public int kthSmallest(TreeNode root, int k) {
    count = k;
    helper(root);
    return number;
}

public void helper(TreeNode n) {
    if (n.left != null) helper(n.left);
    count--;
    if (count == 0) {
        number = n.val;
        return;
    }
    if (n.right != null) helper(n.right);
}
```

解法3：DFS in-order iterative:

```
public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> st = new Stack<>();

        while (root != null) {
            st.push(root);
            root = root.left;
        }

        while (k != 0) {
            TreeNode n = st.pop();
            k--;
            if (k == 0) return n.val;
            TreeNode right = n.right;
            while (right != null) {
                st.push(right);
                right = right.left;
            }
        }

        return -1; // never hit if k is valid
  }
```

Clone Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization: Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node. As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

```
1.First node is labeled as 0. Connect node 0 to both nodes 1 and
 2.
2.Second node is labeled as 1. Connect node 1 to node 2.
3.Third node is labeled as 2. Connect node 2 to node 2 (itself),
 thus forming a self-cycle.
```

Visually, the graph looks like the following:

```
   1
  / \
 /   \
0 --- 2
     / \
     \_/
```

Tips:

先用bfs遍历取出所有节点存在Node里，然后map复制所有节点（不包括邻居），最后复制完成邻居。

可用BFS或DFS，我是用了BFS。先把所有点用BFS搞在ArrayList里，然后用map搞好，一一对应成新的，然后在把边搞了。

也有别的dfs和bfs的做法，都附上了。

Code：

解法1：自己的解法：

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     List<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new A
rrayList<UndirectedGraphNode>(); }
 * };
 */
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode no
de) {
        if (node == null) {
            return node;
        }
        HashMap<UndirectedGraphNode, UndirectedGraphNode> map =
new HashMap<>();
        ArrayList<UndirectedGraphNode> nodes = bfs(node);
        for (UndirectedGraphNode cur : nodes) {
            map.put(cur, new UndirectedGraphNode(cur.label));
        }
        for (UndirectedGraphNode cur: nodes) {
            UndirectedGraphNode newNode = map.get(cur);
            for (UndirectedGraphNode neighbor : cur.neighbors) {
                UndirectedGraphNode newNeighbor = map.get(neighb
or);
                newNode.neighbors.add(newNeighbor);
            }
        }
        return map.get(node);
    }
    private ArrayList<UndirectedGraphNode> bfs(UndirectedGraphNo
de node) {
        ArrayList<UndirectedGraphNode> result = new ArrayList<>(
);
        Queue<UndirectedGraphNode> queue = new LinkedList<>();
        HashSet<UndirectedGraphNode> set = new HashSet<Undirecte
dGraphNode>();
        queue.add(node);
        set.add(node);
        while (!queue.isEmpty()) {
            UndirectedGraphNode cur = queue.remove();
            result.add(cur);
            for (UndirectedGraphNode neighbor : cur.neighbors) {
                if (!set.contains(neighbor)) {
                    queue.add(neighbor);
                    set.add(neighbor);
                }
            }
        }
        return result;
    }
}
```

解法2/3：

```
 private Map<UndirectedGraphNode, UndirectedGraphNode> map = new
 HashMap<UndirectedGraphNode, UndirectedGraphNode>();
// DFS
public UndirectedGraphNode cloneGraph(UndirectedGraphNode node)
{
    if (node == null) return null;
    if (map.containsKey(node)) return map.get(node);
    UndirectedGraphNode copy = new UndirectedGraphNode(node.labe
l);
    map.put(node, copy);
    for (UndirectedGraphNode n : node.neighbors)
        copy.neighbors.add(cloneGraph(n));
    return copy;
}

// BFS
public UndirectedGraphNode cloneGraph(UndirectedGraphNode node)
```

```
{
    if (node == null) return null;
    Queue<UndirectedGraphNode> q = new LinkedList<UndirectedGrap
hNode>();
    q.add(node);
    UndirectedGraphNode copy = new UndirectedGraphNode(node.labe
l);
    map.put(node, copy);
    while (!q.isEmpty()) {
        UndirectedGraphNode cur = q.poll();
        for (UndirectedGraphNode neigh : cur.neighbors) {
            if (map.containsKey(neigh)) map.get(cur).neighbors.a
dd(map.get(neigh));
            else {
                UndirectedGraphNode neighCopy = new UndirectedGr
aphNode(neigh.label);
                map.put(neigh, neighCopy);
                map.get(cur).neighbors.add(neighCopy);
                q.add(neigh);
            }
        }
    }
    return copy;
}
```

Super Ugly Number

Write a program to find the nth super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k. For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4.

Note:

```
(1) 1 is a super ugly number for any given primes.
(2) The given numbers in primes are in ascending order.
(3) 0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000.
```

Tips:

和ugly numberII类似，只是此时如果用set来存已经加过的数，此时因为prime的数量巨大，可能会超出内存。II的代码在最下方。

heap方法:要先定义一个新类Num(int index, int prime, int val),代表这个数理这个prime乘了几次之后又这个val，ugly[i]代表有这是第几个丑数。所以可以while(heap.peek.val() == ugly[i]) 来判断当前最小丑数是不是把所有的prime都拿出来乘了放进heap里。

直接方法：这道题让我们求超级丑陋数，是之前那两道Ugly Number 丑陋数和Ugly Number II 丑陋数之二的延伸，质数集合可以任意给定，这就增加了难度。但是本质上和Ugly Number II 丑陋数之二没有什么区别，由于我们不知道质数的个数，我们可以用一个idx数组来保存当前的位置，然后我们从每个子链中取出一个数，找出其中最小值，然后更新idx数组对应位置，注意有可能最小值不止一个，要更新所有最小值的位置.

Code:

Heap: O(log(k)N)

```
public class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        PriorityQueue<Num> heap = new PriorityQueue<>();
        int[] ugly = new int[n];
        ugly[0] = 1;
        for (int i = 0; i < primes.length; i++) {
            heap.add(new Num(1, primes[i], primes[i]));
        }
        for (int i = 1; i < n; i++) {
            ugly[i] = heap.peek().val;
            while (heap.peek().val == ugly[i]) {
                Num cur = heap.remove();
                heap.add(new Num(cur.index + 1, cur.prime, ugly[
cur.index] * cur.prime));
            }
        }
        return ugly[n - 1];
    }

    private class Num implements Comparable<Num>{
        int index;
        int prime;
        int val;

        public Num(int index, int prime, int val) {
            this.index = index;
            this.prime = prime;
            this.val = val;
        }

        public int compareTo(Num that) {
            return this.val - that.val;
        }
    }
}
```

直接：O(kN)

```
public class Solution {
    /**
     * @param n a positive integer
     * @param primes the given prime list
     * @return the nth super ugly number
     */
    public int nthSuperUglyNumber(int n, int[] primes) {
        int[] times = new int[primes.length];
        int[] uglys = new int[n];
        uglys[0] = 1;

        for (int i = 1; i < n; i++) {
            int min = Integer.MAX_VALUE;
            for (int j = 0; j < primes.length; j++) {
                min = Math.min(min, primes[j] * uglys[times[j]])
;
            }
            uglys[i] = min;

            for (int j = 0; j < times.length; j++) {
                if (uglys[times[j]] * primes[j] == min) {
                    times[j]++;
                }
            }
        }
        return uglys[n - 1];
    }
}
```

Upgly Number II:

// version 1: O(n) scan class Solution { /**

```
     * @param n an integer
     * @return the nth prime number as description.
     */
    public int nthUglyNumber(int n) {
        List<Integer> uglys = new ArrayList<Integer>();
        uglys.add(1);

        int p2 = 0, p3 = 0, p5 = 0;
        // p2, p3 & p5 share the same queue: uglys

        for (int i = 1; i < n; i++) {
            int lastNumber = uglys.get(i - 1);
            while (uglys.get(p2) * 2 <= lastNumber) p2++;
            while (uglys.get(p3) * 3 <= lastNumber) p3++;
            while (uglys.get(p5) * 5 <= lastNumber) p5++;

            uglys.add(Math.min(
                Math.min(uglys.get(p2) * 2, uglys.get(p3) * 3),
                uglys.get(p5) * 5
            ));
        }

        return uglys.get(n - 1);
    }
};
```

// version 2 O(nlogn) HashMap + Heap

```
class Solution {
    /**
     * @param n an integer
     * @return the nth prime number as description.
     */
    public int nthUglyNumber(int n) {
        // Write your code here
        Queue<Long> Q = new PriorityQueue<Long>();
        HashSet<Long> inQ = new HashSet<Long>();
        Long[] primes = new Long[3];
        primes[0] = Long.valueOf(2);
        primes[1] = Long.valueOf(3);
        primes[2] = Long.valueOf(5);
        for (int i = 0; i < 3; i++) {
            Q.add(primes[i]);
            inQ.add(primes[i]);
        }
        Long number = Long.valueOf(1);
        for (int i = 1; i < n; i++) {
            number = Q.poll();
            for (int j = 0; j < 3; j++) {
                if (!inQ.contains(primes[j] * number)) {
                    Q.add(number * primes[j]);
                    inQ.add(number * primes[j]);
                }
            }
        }
        return number.intValue();
    }
};
```

Implement Trie (Prefix Tree)

Implement a trie with insert, search, and startsWith methods.

Note: You may assume that all inputs are consist of lowercase letters a-z.

Tips:

Trie的原理：tree的每一层代表单词的每个letter，每一层最多有26个node（26个字母），还有一个boolean用来store到此层位置是不是一个单词的结尾。

一定要先copy root。

Code:

```
class TrieNode {
    // Initialize your data structure here.
    TrieNode[] children;
    public boolean isKey;
    public TrieNode() {
        children = new TrieNode[26];
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        TrieNode copy = root;
        for (int i = 0; i < word.length(); i++) {
            if (copy.children[word.charAt(i) - 'a'] == null) {
                copy.children[word.charAt(i) - 'a'] = new TrieNo
de();
                copy = copy.children[word.charAt(i) - 'a'];
            }
            else copy = copy.children[word.charAt(i) - 'a'];
        }
        copy.isKey = true;
    }

    // Returns if the word is in the trie.
    public boolean search(String word) {
        TrieNode copy = root;
        for (int i = 0; i < word.length(); i++) {
            if (copy.children[word.charAt(i) - 'a'] == null) {
                return false;
            }
            else copy = copy.children[word.charAt(i) - 'a'];
        }
        return copy.isKey;
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        TrieNode copy = root;
        for (int i = 0; i < prefix.length(); i++) {
            if (copy.children[prefix.charAt(i) - 'a'] == null) {
                return false;
            }
            else copy = copy.children[prefix.charAt(i) - 'a'];
        }
        return true;
    }
}

// Your Trie object will be instantiated and called as such:
// Trie trie = new Trie();
// trie.insert("somestring");
// trie.search("key");
```

Line Reflection

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given points.

Example 1: Given points = [[1,1],[-1,1]], return true.

Example 2: Given points = [[1,1],[-1,-1]], return false.

Follow up: Could you do better than O(n2)?

Hint:

1. Find the smallest and largest x-value for all points.
2. If there is a line then it should be at y = (minX + maxX) / 2.
3. For each point, make sure that it has a reflected point in the opposite side.

Tips:

O(n)

1. If there exists a line reflecting the points, then each pair of symmetric points will have their x coordinates adding up to the same value, including the pair with the maximum and minimum x coordinates.
2. So, in the first pass, I iterate through the array, adding each point to the hash set, and keeping record of the minimum and maximum x coordinates.
3. Then, in the second pass, I check for every point to the left of the reflecting line, if its symmetric point is in the point set or not.
4. If all points pass the test, then there exists a reflecting line. Otherwise, not.
5. Use String to encode the coordinates.

Code:

```
public class Solution {
    public boolean isReflected(int[][] points) {
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        HashMap<Integer, HashSet<Integer>> map = new HashMap<>()
;
        for(int[] p:points){
            max = Math.max(max,p[0]);
            min = Math.min(min,p[0]);
            if (!map.containsKey(p[0])) map.put(p[0], new HashSe
t<>());
            HashSet<Integer> set = map.get(p[0]);
            set.add(p[1]);
            map.put(p[0], set);
        }
        int sum = max+min;
        for(int[] p:points){
            if( !map.containsKey(sum - p[0]) || !map.get(sum - p
[0]).contains(p[1]))
                return false;
        }
        return true;
    }
}
```

String，无set

```
public class Solution {
    public boolean isReflected(int[][] points) {
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        HashSet<String> set = new HashSet<>();
        for(int[] p:points){
            max = Math.max(max,p[0]);
            min = Math.min(min,p[0]);
            String str = p[0] + "a" + p[1];
            set.add(str);
        }
        int sum = max+min;
        for(int[] p:points){
            String str = (sum-p[0]) + "a" + p[1];
            if( !set.contains(str))
                return false;

        }
        return true;
    }
}
```

Sliding Window Maximum

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?

Hint:

1. How about using a data structure such as deque (double-ended queue)?
2. The queue size need not be the same as the window’s size.
3. Remove redundant elements and the queue should store only elements that need to be

Tip：

双向队列，**O(N)** ，窗口里存的是index；

当我们遇到新的数时，将新的数和双向队列的末尾比较，如果末尾比新数小，则把末尾扔掉，直到该队列的末尾比新数大或者队列为空的时候才住手。这样，我们可以保证队列里的元素是从头到尾降序的，由于队列里只有窗口内的数，所以他们其实就是窗口内第一大，第二大，第三大...的数。保持队列里只有窗口内数的方法和上个解法一样，也是每来一个新的把窗口最左边的扔掉，然后把新的加进去。然而由于我们在加新数的时候，已经把很多没用的数给扔了，这样队列头部的数并不一定是窗口最左边的数。这里的技巧是，我们队列中存的是那个数在原数组中的下标，这样我们既可以直到这个数的值，也可以知道该数是不是窗口最左边的数。这里为什么时间复杂度是O(N)呢？因为每个数只可能被操作最多两次，一次是加入队列的时候，一次是因为有别的更大数在后面，所以被扔掉，或者因为出了窗口而被扔掉。

we scan the array from 0 to n-1, keep "promising" elements in the deque. The algorithm is amortized O(n) as each element is put and polled once.

At each i, we keep "promising" elements, which are potentially max number in window [i-(k-1),i] or any subsequent window. This means

If an element in the deque and it is out of i-(k-1), we discard them. We just need to poll from the head, as we are using a deque and elements are ordered as the sequence in the array

Now only those elements within [i-(k-1),i] are in the deque. We then discard elements smaller than a[i] from the tail. This is because if a[x] <a[i] and x<i, then a[x] has no chance to be the "max" in [i-(k-1),i], or any other subsequent window: a[i] would always be a better candidate.

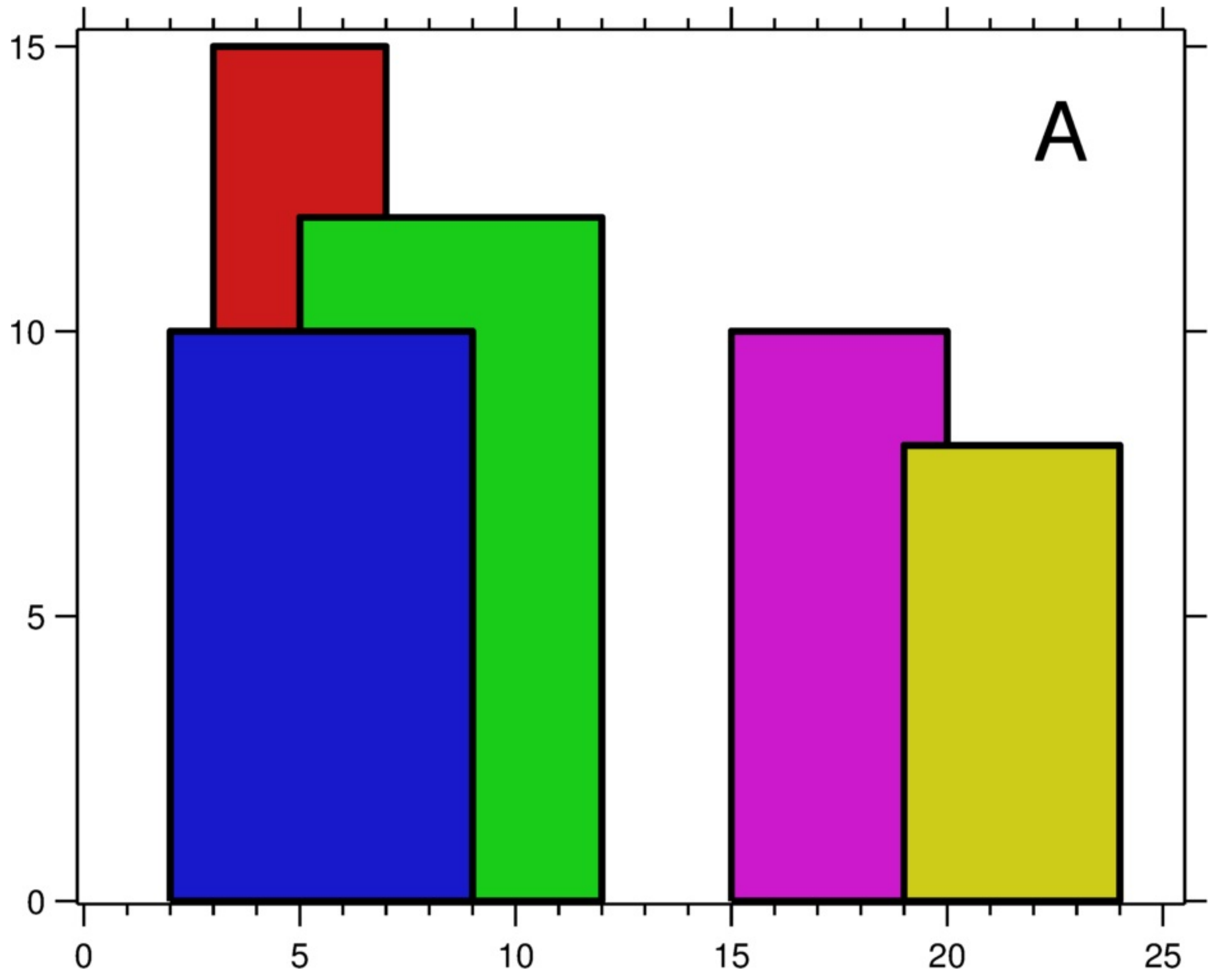
As a result elements in the deque are ordered in both sequence in array and their value. At each step the head of the deque is the max element in [i-(k-1),i]

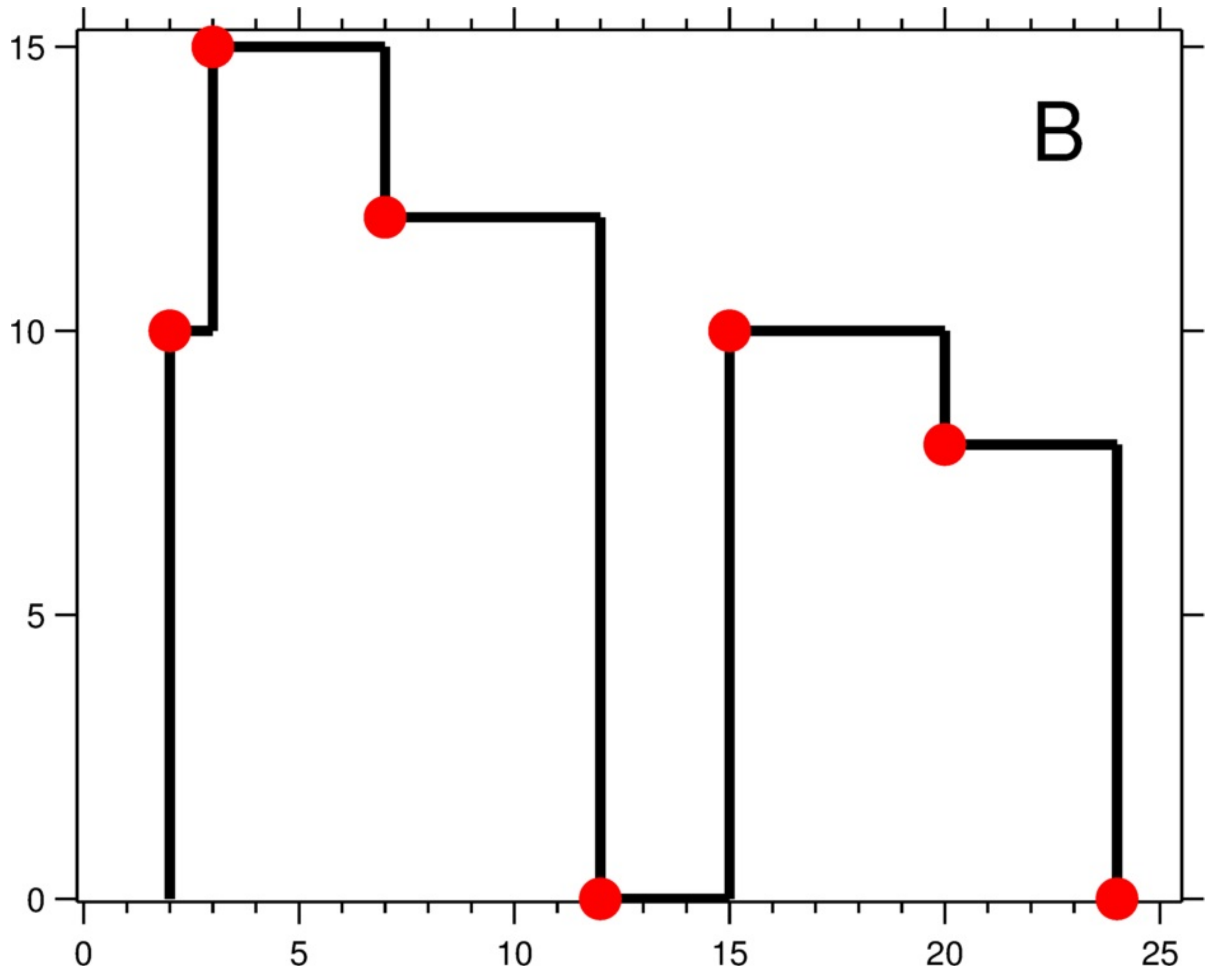
Code:

```
public class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
            if(nums == null || nums.length == 0) return new int[
0];
            Deque<Integer> deque = new LinkedList<Integer>();
            int[] res = new int[nums.length + 1 - k];
            for(int i = 0; i < nums.length; 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 + 1 - k] = nums[deque.pee
k()];
            }
            return res;
        }
}
```

The Skyline Problem

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are given the locations and height of all the buildings as shown on a cityscape photo (Figure A), write a program to output the skyline formed by these buildings collectively (Figure B).





The geometric information of each building is represented by a triplet of integers [Li, Ri, Hi], where Li and Ri are the x coordinates of the left and right edge of the ith building, respectively, and Hi is its height. It is guaranteed that 0 ≤ Li, Ri ≤ INT_MAX, 0 < Hi ≤ INT_MAX, and Ri - Li > 0. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

For instance, the dimensions of all buildings in Figure A are recorded as: [[2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8]] .

The output is a list of "key points" (red dots in Figure B) in the format of [[x1,y1], [x2, y2], [x3, y3], ...] that uniquely defines a skyline. A key point is the left endpoint of a horizontal line segment. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as:[[2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0]].

Notes:

1. The number of buildings in any input list is guaranteed to be in the range [0, 10000].
2. The input list is already sorted in ascending order by the left x position Li.
3. The output list must be sorted by the x position.
4. There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...[2 3], [4 5], [7 5], [11 5], [12 7]...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...[2 3], [4 5], [12 7], ...]

Tips:

O(nlogn)

如果按照一个矩形一个矩形来处理将会非常麻烦，我们可以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，我们开始遍历这些点。遍历时，通过一个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。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。具体代码中，为了在排序后的顶点列表中区分左右顶点，左顶点的值是正数，而右顶点值则存的是负数。

Code:

```
public class Solution {
    public List<int[]> getSkyline(int[][] buildings) {
        List<int[]> result = new ArrayList<>();
        List<int[]> height = new ArrayList<>();
        // 拆解矩形，构建顶点的列表
        for(int[] b:buildings) {
            // 左顶点存为负数
            height.add(new int[]{b[0], -b[2]});
            // 右顶点存为正数
            height.add(new int[]{b[1], b[2]});
        }
        // 根据横坐标对列表排序，相同横坐标的点纵坐标小的排在前面
        Collections.sort(height, new Comparator<int[]>(){
            public int compare(int[] a, int[] b){
                if(a[0] != b[0]){
                    return a[0] - b[0];
                } else {
                    return a[1] - b[1];
                }
            }
        });
        // 构建堆，按照纵坐标来判断大小
        Queue<Integer> pq = new PriorityQueue<Integer>(11, new C
omparator<Integer>(){
            public int compare(Integer i1, Integer i2){
                return i2 - i1;
            }
        });
        // 将地平线值9先加入堆中
        pq.offer(0);
        // prev用于记录上次keypoint的高度
        int prev = 0;
        for(int[] h:height) {
            // 将左顶点加入堆中
            if(h[1] < 0) {
                pq.offer(-h[1]);
            } else {
            // 将右顶点对应的左顶点移去
                pq.remove(h[1]);
            }
            int cur = pq.peek();
            // 如果堆的新顶部和上个keypoint高度不一样，则加入一个新的key
point
            if(prev != cur) {
                result.add(new int[]{h[0], cur});
                prev = cur;
            }
        }
        return result;
    }
}
```

Read N Characters Given Read4

The API: int read4(char *buf) reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the read4 API, implement the function int read(char *buf, int n) that reads n characters from the file.

Note: The read function will only be called once for each test case.

Code:

```
public int read(char[] buf, int n) {
  boolean eof = false;      // end of file flag
  int total = 0;            // total bytes have read
  char[] tmp = new char[4]; // temp buffer

  while (!eof && total < n) {
    int count = read4(tmp);

    // check if it's the end of the file
    eof = count < 4;

    // get the actual count
    count = Math.min(count, n - total);

    // copy from temp buffer to buf
    for (int i = 0; i < count; i++)
      buf[total++] = tmp[i];
  }

  return total;
}
```

Read N Characters Given Read4 II - Call multiple times

The API: int read4(char *buf) reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the read4 API, implement the function int read(char *buf, int n) that reads n characters from the file.

Note:

The read function may be called multiple times.

Tips:

上一题的followup，可以多次调用read

所以上次调用可能有读多了的，要先存下来。

复杂度**O(n)**

Code:

/ *The read4 API is defined in the parent class Reader4. int read4(char[] buf);* /

```
public class Solution extends Reader4 {
    /**
     * @param buf Destination buffer
     * @param n   Maximum number of characters to read
     * @return    The number of characters read
     */
    Queue<Character> remain = new LinkedList<Character>();
    public int read(char[] buf, int n) {
        int i = 0;
        // 队列不为空时，先读取队列中的暂存字符
        while(i < n && !remain.isEmpty()){
            buf[i] = remain.poll();
            i++;
        }
        for(; i < n; i += 4){
            char[] tmp = new char[4];
            int len = read4(tmp);
            // 如果读到字符多于我们需要的字符，需要暂存这些多余字符
            if(len > n - i){
                System.arraycopy(tmp, 0, buf, i, n - i);
                // 把多余的字符存入队列中
                for(int j = n - i; j < len; j++){
                    remain.offer(tmp[j]);
                }
            // 如果读到的字符少于我们需要的字符，直接拷贝
            } else {
                System.arraycopy(tmp, 0, buf, i, len);
            }
            // 同样的，如果读不满4个，说明数据已经读完，返回总所需长度和目
前已经读到的长度的较小的
            if(len < 4) return Math.min(i + len, n);
        }
        // 如果到这里，说明都是完美读取，直接返回n
        return n;
    }
}
```

LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

Tips:

考察对基本数据结构哈希表、链表的掌握，

为了使查找、插入和删除都有较高的性能，首先定义一个Node节点存储key, value, prev, next。使用一个哈希表。每次进行操作以后把最近操作的放到队尾。

复杂度：

get（）采用哈希表查找，时间复杂度**O**（**1**），set（）采用双向链表，删除插入时间复杂度**O(1)**

Code:

```
public class LRUCache {
    private class Node{
        Node prev;
        Node next;
        int key;
        int value;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
            this.prev = null;
            this.next = null;
        }
    }

    private int capacity;
    private HashMap<Integer, Node> hs = new HashMap<Integer, Nod
e>();
    private Node head = new Node(-1, -1);
    private Node tail = new Node(-1, -1);

    public LRUCache(int capacity) {
        this.capacity = capacity;
        tail.prev = head;
        head.next = tail;
    }

    public int get(int key) {
        if( !hs.containsKey(key)) {
            return -1;
        }

        // remove current
        Node current = hs.get(key);
        current.prev.next = current.next;
        current.next.prev = current.prev;

        // move current to tail
        move_to_tail(current);

        return hs.get(key).value;
    }

    public void set(int key, int value) {
        if( get(key) != -1) {
            hs.get(key).value = value;
            return;
        }

        if (hs.size() == capacity) {
            hs.remove(head.next.key);
            head.next = head.next.next;
            head.next.prev = head;
        }

        Node insert = new Node(key, value);
        hs.put(key, insert);
        move_to_tail(insert);
    }

    private void move_to_tail(Node current) {
        current.prev = tail.prev;
        tail.prev = current;
        current.prev.next = current;
        current.next = tail;
    }
}
```

LFU Cache

Design and implement a data structure for Least Frequently Used (LFU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

put(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least recently used key would be evicted.

Follow up:

Could you do both operations in O(1) time complexity?

Example:

```
LFUCache cache = new LFUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.get(3);       // returns 3.
cache.put(4, 4);    // evicts key 1.
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

Tips:

需要用两个map存放keys和values，用一个linkedhashset存放出现的顺序。而创造node的时候以（count）为输入，同样count的放在一个node，进行每次操作时increaseCount。每次将新点加入成为第一个点（addToHead)。如果capacity不

够，则选择head里的第一个点进行删除，如果head里只有一个key，那么移除head使下一位变成head。

head --- FreqNode1 ---- FreqNode2 ---- ... ---- FreqNodeN | | |
first first first
| | |
KeyNodeA KeyNodeE KeyNodeG
| | |
KeyNodeB KeyNodeF KeyNodeH
| | |
KeyNodeC last KeyNodeI
| |
KeyNodeD last | last

Code:

```
    public class LFUCache {
        private Node head = null;
        private int cap = 0;
        private HashMap<Integer, Integer> valueHash = null;
        private HashMap<Integer, Node> nodeHash = null;

        public LFUCache(int capacity) {
            this.cap = capacity;
            valueHash = new HashMap<Integer, Integer>();
            nodeHash = new HashMap<Integer, Node>();
        }

        public int get(int key) {
            if (valueHash.containsKey(key)) {
                increaseCount(key);
                return valueHash.get(key);
            }
            return -1;
        }

        public void put(int key, int value) {
            if ( cap == 0 ) return;
            if (valueHash.containsKey(key)) {
                valueHash.put(key, value);
            } else {
                if (valueHash.size() < cap) {
                    valueHash.put(key, value);
                } else {
                    removeOld();
                    valueHash.put(key, value);
                }
                addToHead(key);
            }
            increaseCount(key);
        }

        private void addToHead(int key) {
            if (head == null) {
                head = new Node(0);
                head.keys.add(key);
            } else if (head.count > 0) {
                Node node = new Node(0);
                node.keys.add(key);
                node.next = head;
                head.prev = node;
                head = node;
            } else {
                head.keys.add(key);
            }
            nodeHash.put(key, head);
        }

        private void increaseCount(int key) {
            Node node = nodeHash.get(key);
            node.keys.remove(key);

            if (node.next == null) {
                node.next = new Node(node.count+1);
                node.next.prev = node;
                node.next.keys.add(key);
            } else if (node.next.count == node.count+1) {
                node.next.keys.add(key);
            } else {
                Node tmp = new Node(node.count+1);
                tmp.keys.add(key);
                tmp.prev = node;
                tmp.next = node.next;
                node.next.prev = tmp;
                node.next = tmp;
            }

            nodeHash.put(key, node.next);
            if (node.keys.size() == 0) remove(node);
        }

        private void removeOld() {
            if (head == null) return;
            int old = head.keys.iterator().next();
            head.keys.remove(old);
            if (head.keys.size() == 0) remove(head);
            nodeHash.remove(old);
            valueHash.remove(old);
        }

        private void remove(Node node) {
            if (node.prev == null) {
                head = node.next;
            } else {
                node.prev.next = node.next;
            }
            if (node.next != null) {
                node.next.prev = node.prev;
            }
        }

        class Node {
            public int count = 0;
            public LinkedHashSet<Integer> keys = null;
            public Node prev = null, next = null;

            public Node(int count) {
                this.count = count;
                keys = new LinkedHashSet<Integer>();
                prev = next = null;
            }
        }
    }
```

Word Search II

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.

Tips:

We mark visited as "#" (pound)

We need to use DFS to search words in board. And we need to be very smart on stop recursion. We can use a hashmap which map characters to their all positions in the board. Then search each word in board char by char using the map. This may consumes a lot time to search word prefix. So it’s better to use Trie, to store all words and do searching starts with every position in the board with the trie.

Intuitively, start from every cell and try to build a word in the dictionary. Backtracking (dfs) is the powerful way to exhaust every possible ways. Apparently, we need to do pruning when current character is not in any word.

```
How do we instantly know the current character is invalid? HashM
ap?
How do we instantly know what's the next valid character? Linked
List?
But the next character can be chosen from a list of characters.
"Mutil-LinkedList"?
Combing them, Trie is the natural choice. Notice that:

TrieNode is all we need. search and startsWith are useless.
No need to store character at TrieNode. c.next[i] != null is eno
ugh.
Never use c1 + c2 + c3. Use StringBuilder.
No need to use O(n^2) extra space visited[m][n].
No need to use StringBuilder. Storing word itself at leaf node i
s enough.
No need to use HashSet to de-duplicate. Use "one time search" tr
ie.
```

1. You **do not need Set** res to remove duplicate.
2. You could just use List res.
3. Because the **Trie has the ability of remove duplicate**.

   check validity, e.g., if(i > 0) dfs(...), before going to the next dfs. De-duplicate c - a with one variable i. Remove HashSet completely. dietpepsi's idea is awesome.(解释见上)

Code:

```
public class Solution {
    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, Li
st<String> res) {
        char c = board[i][j];
        if(c == '#' || p.next[c - 'a'] == null) return;
        p = p.next[c - 'a'];
        if(p.word != null) {   // found one
            res.add(p.word);
            p.word = null;     // de-duplicate
        }

        board[i][j] = '#';
        if(i > 0) dfs(board, i - 1, j ,p, res);
        if(j > 0) dfs(board, i, j - 1, p, res);
        if(i < board.length - 1) dfs(board, i + 1, j, p, res);
        if(j < board[0].length - 1) dfs(board, i, j + 1, p, res)
;
        board[i][j] = c;
    }

    public TrieNode buildTrie(String[] words) {
        TrieNode root = new TrieNode();
        for(String w : words) {
            TrieNode p = root;
            for(char c : w.toCharArray()) {
                int i = c - 'a';
                if(p.next[i] == null) p.next[i] = new TrieNode()
;
                p = p.next[i];
           }
           p.word = w;
        }
        return root;
    }

    class TrieNode {
        TrieNode[] next = new TrieNode[26];
        String word;
    }
```

```
}
```

Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open (and closing parentheses), the plus + or minus sign -, non-negative integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

```
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23
```

Note: Do not use the eval built-in library function.

Tips:

Simple iterative solution by identifying characters one by one. One important thing is that the input is valid, which means the parentheses are always paired and in order.

Only 5 possible input we need to pay attention:

```
digit: it should be one digit from the current number

'+': number is over, we can add the previous number and start a
new number
'-': same as above
'(': push the previous result and the sign into the stack,
     set result to 0, just calculate the new result within the p
arenthesis.
')': pop out the top two numbers from stack,
     first one is the sign before this pair of parenthesis,
     second is the temporary result before this pair of parenthe
sis. We add them together.

Finally if there is only one number, from the above solution,
we haven't add the number to the result, so we do a check see if
 the number is zero.
```

Code:

```
public int calculate(String s) {
    Stack<Integer> stack = new Stack<Integer>();
    int result = 0;
    int number = 0;
    int sign = 1;
    for(int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        if(Character.isDigit(c)){
            number = 10 * number + (int)(c - '0');
        }else if(c == '+'){
            result += sign * number;
            number = 0;
            sign = 1;
        }else if(c == '-'){
            result += sign * number;
            number = 0;
            sign = -1;
        }else if(c == '('){
            //we push the result first, then sign;
            stack.push(result);
            stack.push(sign);
            //reset the sign and result for the value in the par
enthesis
            sign = 1;
            result = 0;
        }else if(c == ')'){
            result += sign * number;
            number = 0;
            result *= stack.pop();    //stack.pop() is the sign
before the parenthesis
            result += stack.pop();   //stack.pop() now is the re
sult calculated before the parenthesis

        }
    }
    if(number != 0) result += sign * number;
    return result;
}
```

The Maze

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, determine whether the ball could stop at the destination.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

Example 1

```
    Input 1: a maze represented by a 2D array

    0 0 1 0 0
    0 0 0 0 0
    0 0 0 1 0
    1 1 0 1 1
    0 0 0 0 0

    Input 2: start coordinate (rowStart, colStart) = (0, 4)
    Input 3: destination coordinate (rowDest, colDest) = (4, 4)

    Output: true
    Explanation: One possible way is : left -> down -> left -> d
own -> right -> down -> right.
```

Example 2

```
           Input 1: a maze represented by a 2D array

           0 0 1 0 0
           0 0 0 0 0
           0 0 0 1 0
           1 1 0 1 1
           0 0 0 0 0

           Input 2: start coordinate (rowStart, colStart) = (0,
4)
           Input 3: destination coordinate (rowDest, colDest) =
(3, 2)

           Output: false
           Explanation: There is no way for the ball to stop at
the destination.
```

Note:

```
    1. There is only one ball and one destination in the maze.
    2. Both the ball and the destination exist on an empty space
, and they will not be at the same position initially.
    3. The given maze does not contain border (like the red rect
angle in the example pictures), but you could assume the border
of the maze are all walls.
    4. The maze contains at least 2 empty spaces, and both the w
idth and height of the maze won't exceed 100.
```

Tips: 比较简单的dfs，需要注意球只能在碰壁的时候改变方向。在dfs的while中，会多加一次distance，所以在进入下一层时应该减掉。

Time Complexity: **O(N)**

Code:

```
    public class Solution {
        private static final int[][] directions = {{0, 1}, {0, -
1}, {1, 0}, {-1, 0}};
        public boolean hasPath(int[][] maze, int[] start, int[]
destination) {
            boolean[][] flag = new boolean[maze.length][maze[0].
length];
            return dfs(maze, start, flag, destination);
        }
        private boolean dfs(int[][] maze, int[] start, boolean[]
[] flag, int[] destination) {
            if (flag[start[0]][start[1]]) return false;
            if (start[0] == destination[0] && start[1] == destin
ation[1]) return true;
            flag[start[0]][start[1]] = true;
            for (int[] d : directions) {
                int row = start[0], col = start[1];
                while(row >= 0 && col >= 0 && row < maze.length
&& col < maze[0].length && maze[row][col] == 0) {
                    row += d[0];
                    col += d[1];
                }
                int[] newStart = {row - d[0], col - d[1]};
                if (dfs(maze, newStart, flag, destination)) retu
rn true;
            }
            return false;
        }
    }
```

bfs:

```
public class Solution {
    class Point {
        int x,y;
        public Point(int _x, int _y) {x=_x;y=_y;}
    }
    public boolean hasPath(int[][] maze, int[] start, int[] dest
ination) {
        int m=maze.length, n=maze[0].length;
        if (start[0]==destination[0] && start[1]==destination[1]
) return true;
        int[][] dir=new int[][] {{-1,0},{0,1},{1,0},{0,-1}};
        boolean[][] visited=new boolean[m][n];
        LinkedList<Point> list=new LinkedList<>();
        visited[start[0]][start[1]]=true;
        list.offer(new Point(start[0], start[1]));
        while (!list.isEmpty()) {
            Point p=list.poll();
            int x=p.x, y=p.y;
            for (int i=0;i<4;i++) {
                int xx=x, yy=y;
                while (xx>=0 && xx<m && yy>=0 && yy<n && maze[xx
][yy]==0) {
                    xx+=dir[i][0];
                    yy+=dir[i][1];
                }
                xx-=dir[i][0];
                yy-=dir[i][1];
                if (visited[xx][yy]) continue;
                visited[xx][yy]=true;
                if (xx==destination[0] && yy==destination[1]) re
turn true;
                list.offer(new Point(xx, yy));
            }
        }
        return false;

    }
}
```

The Maze II

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

Example 1

```
         Input 1: a maze represented by a 2D array

         0 0 1 0 0
         0 0 0 0 0
         0 0 0 1 0
         1 1 0 1 1
         0 0 0 0 0

         Input 2: start coordinate (rowStart, colStart) = (0, 4)
         Input 3: destination coordinate (rowDest, colDest) = (4
, 4)

         Output: 12
         Explanation: One shortest way is : left -> down -> left
 -> down -> right -> down -> right.
                      The total distance is 1 + 1 + 3 + 1 + 2 +
2 + 2 = 12.
```

Example 2

```
        Input 1: a maze represented by a 2D array

        0 0 1 0 0
        0 0 0 0 0
        0 0 0 1 0
        1 1 0 1 1
        0 0 0 0 0

        Input 2: start coordinate (rowStart, colStart) = (0, 4)
        Input 3: destination coordinate (rowDest, colDest) = (3
, 2)

        Output: -1
        Explanation: There is no way for the ball to stop at th
e destination.
```

Note:

1. There is only one ball and one destination in the maze.
2. Both the ball and the destination exist on an empty space, and they will not be at the same position initially.
3. The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.
4. The maze contains at least 2 empty spaces, and both the width and height of the maze won't exceed 100.

Tips:

和maze1很相似，不同的是mazeI只需要判断是否能到，mazeII要计算最短距离。此时只需要把代表visited数组变成distance数组，保存到这点的最小距离即可，当需要更新最小距离时，再往下dfs。

bfs的做法可以直接用heap来找到到每个点的最小距离。

Code:

dfs:

```
public class Solution {
    private static final int[][] directions = {{0, 1}, {0, -1},
{1, 0}, {-1, 0}};
    public int shortestDistance(int[][] maze, int[] start, int[]
 destination) {
        int[][] distance = new int[maze.length][maze[0].length];
        for (int i = 0; i < maze.length; i++) {
            for (int j = 0; j < maze[0].length; j++) {
                distance[i][j] = Integer.MAX_VALUE;
            }
        }
        distance[start[0]][start[1]] = 0;
        dfs(maze, start, destination, distance);
        return distance[destination[0]][destination[1]] == Integ
er.MAX_VALUE ? -1 : distance[destination[0]][destination[1]];
    }

    private void dfs(int[][] maze, int[] start, int[] destinatio
n, int[][] distance) {
        if (start[0] == destination[0] && start[1] == destinatio
n[1]) return;
        for (int[] d : directions) {
            int row = start[0], col = start[1];
            while (row >= 0 && row < maze.length && col >= 0 &&
col < maze[0].length && maze[row][col] == 0) {
                row += d[0];
                col += d[1];
            }
            row -= d[0];
            col -= d[1];
            int[] newStart = {row, col};
            int dis = distance[start[0]][start[1]] + Math.abs(st
art[0] - row) + Math.abs(start[1] - col);
            if (distance[row][col] > dis) {
                distance[row][col] = dis;
                dfs(maze, newStart, destination, distance);
            }
        }
    }
}
```

bfs:

```
public class Solution {
    class Point {
        int x,y,l;
        public Point(int _x, int _y, int _l) {x=_x;y=_y;l=_l;}
    }
    public int shortestDistance(int[][] maze, int[] start, int[]
 destination) {
        int m=maze.length, n=maze[0].length;
        int[][] length=new int[m][n]; // record length
        for (int i=0;i<m*n;i++) length[i/n][i%n]=Integer.MAX_VAL
UE;
        int[][] dir=new int[][] {{-1,0},{0,1},{1,0},{0,-1}};
        PriorityQueue<Point> list=new PriorityQueue<>((o1,o2)->o
1.l-o2.l); // using priority queue
        list.offer(new Point(start[0], start[1], 0));
        while (!list.isEmpty()) {
            Point p=list.poll();
            if (length[p.x][p.y]<=p.l) continue; // if we have a
lready found a route shorter
            length[p.x][p.y]=p.l;
            for (int i=0;i<4;i++) {
                int xx=p.x, yy=p.y, l=p.l;
                while (xx>=0 && xx<m && yy>=0 && yy<n && maze[xx
][yy]==0) {
                    xx+=dir[i][0];
                    yy+=dir[i][1];
                    l++;
                }
                xx-=dir[i][0];
                yy-=dir[i][1];
                l--;
                list.offer(new Point(xx, yy, l));
            }
        }
        return length[destination[0]][destination[1]]==Integer.M
AX_VALUE?-1:length[destination[0]][destination[1]];
    }
}
```

Find Median from Data Stream

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:

1. void addNum(int num) - Add a integer number from the data stream to the data structure.
2. double findMedian() - Return the median of all elements so far.

For example:

```
add(1)
add(2)
findMedian() -> 1.5
add(3)
findMedian() -> 2
```

Tips:

I keep two heaps (or priority queues):

Max-heap small has the smaller half of the numbers.

Min-heap large has the larger half of the numbers.

This gives me direct access to the one or two middle values (they're the tops of the heaps), so getting the median takes O(1) time. And adding a number takes O(log n) time.

Supporting both min- and max-heap is more or less cumbersome, depending on the language, so I simply negate the numbers in the heap in which I want the reverse of the default order. To prevent this from causing a bug with -231 (which negated is itself, when using 32-bit ints), I use integer types larger than 32 bits.

Using larger integer types also prevents an overflow error when taking the mean of the two middle numbers. I think almost all solutions posted previously have that bug.

Update: These are pretty short already, but by now I wrote even shorter ones.

Code:

```
class MedianFinder {

    private Queue<Long> small = new PriorityQueue(),
                        large = new PriorityQueue();

    public void addNum(int num) {
        large.add((long) num);
        small.add(-large.poll());
        if (large.size() < small.size())
            large.add(-small.poll());
    }

    public double findMedian() {
        return large.size() > small.size()
               ? large.peek()
               : (large.peek() - small.peek()) / 2.0;
    }
};
```

Create Maximum Number

Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k <= m + n from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits. You should try to optimize your time and space complexity.

```
Example 1:
nums1 = [3, 4, 6, 5]
nums2 = [9, 1, 2, 5, 8, 3]
k = 5
return [9, 8, 6, 5, 3]

Example 2:
nums1 = [6, 7]
nums2 = [6, 0, 4]
k = 5
return [6, 7, 6, 0, 4]

Example 3:
nums1 = [3, 9]
nums2 = [8, 9]
k = 3
return [9, 8, 9]
```

Tips:

题意：给定两个长度分别为m与n的数组，数组元素为0-9，每个数组代表一个数字。从这两个数组中选出一些数字，组成一个最大的数，其长度k <= m + n。数组中选出数字的相对顺序应当与原数组保持一致。返回一个包含k个数字的数组。你应当最优化时间和空间复杂度。

递归写法TLE了。

枚举第一个数组A的个数i，那么数组B的个数就确定了 k - i。

然后枚举出A和B长度分别为i和k-i的最大子数组，（可以用栈，类似leetcode Remove Duplicate Letters）

最后组合看看哪个大。

组合的过程类似合并排序，看看哪个数组大，就选哪个。

枚举数组长度复杂度O(k)，找出最大子数组O(n)，合并的话O(k^2)

而k最坏是m+n,所以总的复杂度就是O((m+n)^3)

To find the maximum ,we can enumerate how digits we should get from nums1 , we suppose it is i.

So , the digits from nums2 is K - i.

And we can use a stack to get the get maximum number(x digits) from one array.

OK, Once we choose two maximum subarray , we should combine it to the answer.

It is just like merger sort, but we should pay attention to the case: the two digital are equal.

we should find the digits behind it to judge which digital we should choose now.

In other words,we should judge which subarry is bigger than the other.

That's all.

update:use stack to find max sub array and it runs 21ms now.

Code:

```
public class Solution {
    public int[] maxNumber(int[] nums1, int[] nums2, int k) {
        int get_from_nums1 = Math.min(nums1.length, k);
        int[] ans = new int[k];
        for (int i = Math.max(k - nums2.length, 0); i <= get_fro
m_nums1; i++) {
            int[] res1 = new int[i];
            int[] res2 = new int[k - i];
            int[] res = new int[k];
            res1 = solve(nums1, i);
            res2 = solve(nums2, k - i);
            int pos1 = 0, pos2 = 0, tpos = 0;

            while (res1.length > 0 && res2.length > 0 && pos1 <
res1.length && pos2 < res2.length) {
                if (compare(res1, pos1, res2, pos2))
                    res[tpos++] = res1[pos1++];
                else
                    res[tpos++] = res2[pos2++];
            }
            while (pos1 < res1.length)
                res[tpos++] = res1[pos1++];
            while (pos2 < res2.length)
                res[tpos++] = res2[pos2++];

            if (!compare(ans, 0, res, 0))
                ans = res;
        }

        return ans;
    }

    public boolean compare(int[] nums1, int start1, int[] nums2,
 int start2) {
        for (; start1 < nums1.length && start2 < nums2.length; s
tart1++, start2++) {
            if (nums1[start1] > nums2[start2]) return true;
            if (nums1[start1] < nums2[start2]) return false;
        }
        return start1 != nums1.length;
    }

    public int[] solve(int[] nums, int k) {
        int[] res = new int[k];
        int len = 0;
        for (int i = 0; i < nums.length; i++) {
            while (len > 0 && len + nums.length - i > k && res[l
en - 1] < nums[i]) {
                len--;
            }
            if (len < k)
                res[len++] = nums[i];
        }
        return res;
    } }
```

Next Greater Element II

Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element.

The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.

Example 1:

```
Input: [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2;
The number 2 can't find next greater number;
The second 1's next greater number needs to search circularly, w
hich is also 2.
Note: The length of given array won't exceed 10000.
```

Tips:

stack: O(n)

1. 先从倒数第二个数开始，创建一个递减栈，把栈内比当前数小的数都pop后push当前数。
2. 从最后一个数开始，在递减栈中将比自己小的数都扔掉，创建新的递减栈。因为栈先入后出，所以第二遍等于从头开始找比自己大的第一个数。此时res[i] = stack.peek()，然后再将当前数入栈创建新的递减栈。

brute forth: O(n^2)

Code:

stack: 屌得飞

```
public class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int[] res = new int[nums.length];
        Stack<Integer> stack = new Stack<>();
        for (int i = nums.length - 2; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek()  <= nums[i])
 stack.pop();
            stack.push(nums[i]);
        }
        for (int i = nums.length - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek()  <= nums[i])
 stack.pop();
            if (stack.isEmpty()) res[i] = -1;
            else res[i] = stack.peek();
            stack.push(nums[i]);
        }
        return res;
    }
}
```

brute forth:

```
public int[] nextGreaterElements2(int[] nums) {
    int  n = nums.length;
    int[] res = new int[n];
    for (int i = 0; i < n; i++) {
        res[i] = -1;
        for (int k = i+1; k < i + n; k++) {
            if (nums[k%n] > nums[i]){
                res[i] = nums[k%n];
                break;
            }
        }
    }
    return res;
}
```

Find Mode in Binary Search Tree

Given a binary search tree (BST) with duplicates, find all the mode(s) (the most frequently occurred element) in the given BST.

Assume a BST is defined as follows:

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

For example:

Given BST [1,null,2,2],

```
   1
    \
     2
    /
   2
```

return [2].

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

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

Tips: Time: **O(n)** Space: **extra O(1)** 利用BST的特性，recursive中序遍历然后找众数。注意list.clear();

Code:

```
public class Solution {
    int curCount = 1;
    int maxCount = 0;
    TreeNode prev;
    public int[] findMode(TreeNode root) {
        if (root == null) return new int[0];
        List<Integer> list = new ArrayList<>();
        inorder(root, list);
        int[] res = new int[list.size()];
        for (int i = 0; i < list.size(); i++) res[i] = list.get(
i);
        return res;
    }
    private void inorder(TreeNode root, List<Integer> list) {
        if (root == null) return;
        inorder(root.left, list);
        if (prev != null) {
            if (root.val == prev.val) curCount++;
            else curCount = 1;
        }
        prev = new TreeNode(root.val);
        if (curCount > maxCount) {
            maxCount = curCount;
            list.clear();
            list.add(root.val);
        } else if (curCount == maxCount) list.add(root.val);
        inorder(root.right, list);
    }
}
```

Sliding Window Median

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

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. Your job is to output the median array for each window in the original array.

For example, Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.

```
Window position                Median
---------------               -----
[1  3  -1] -3  5  3  6  7       1
 1 [3  -1  -3] 5  3  6  7       -1
 1  3 [-1  -3  5] 3  6  7       -1
 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]      6
```

Therefore, return the median sliding window as [1,-1,-1,3,5,6].

Note: You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

Tips:

建一个最大堆一个最小堆，最大堆存放小的数最小堆存放大的数，二者堆顶的平均值即为这个区间的median。 注意时刻更新两个堆，保持最小堆的大小始终为最大堆的大小或最大堆+1。有新数加入时先删除再入堆。 注意(double)的用法，不然达到Integer.MAX_VALUE的时候回报错。 注意重写的compare方法要用Integer和compareTo,直接用int和-不对，会遇到int最大值会出错。

Code:

```
public class Solution {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(new Com
parator<Integer>() {
        public int compare(Integer a, Integer b) {
            return b.compareTo(a);
        }
    });
    public double[] medianSlidingWindow(int[] nums, int k) {
        int n = nums.length - k + 1;
        if (n <= 0) return new double[0];
        double[] res = new double[n];
        for (int i = 0; i <= nums.length; i++) {
            if (i >= k) {
                res[i - k] = getMedian();
                remove(nums[i - k]);
            }
            if (i < nums.length) {
                add(nums[i]);
            }
        }
        return res;
    }
    private double getMedian() {
        if (minHeap.isEmpty() && maxHeap.isEmpty()) return 0;
        if (minHeap.size() == maxHeap.size()) return ((double)ma
xHeap.peek() + (double)minHeap.peek())/2.0;
        else return (double)minHeap.peek();
    }
    private void remove(int x) {
        if (x < getMedian()) maxHeap.remove(x);
        else minHeap.remove(x);
        if (maxHeap.size() > minHeap.size()) minHeap.add(maxHeap
.poll());
        if (minHeap.size() > maxHeap.size() + 1) maxHeap.add(min
Heap.poll());
    }
    private void add(int x) {
        if (x < getMedian()) maxHeap.add(x);
        else minHeap.add(x);
        if (maxHeap.size() > minHeap.size()) minHeap.add(maxHeap
.poll());
        if (minHeap.size() > maxHeap.size() + 1) maxHeap.add(min
Heap.poll());
    }
}
```

Shortest Distance from All Buildings

You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values 0, 1 or 2, where:

1. Each 0 marks an empty land which you can pass by freely.
2. Each 1 marks a building which you cannot pass through.
3. Each 2 marks an obstacle which you cannot pass through.

For example, given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2):

```
1 - 0 - 2 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

The point (1,2) is an ideal empty land to build a house, as the total travel distance of 3+3+1=7 is minimal. So return 7.

Note: There will be at least one building. If it is not possible to build such house according to the above rules, return -1.

Tips:

BFS，先找出所有building，然后算他们到每个building的距离存入dist数组，最后得出最近的点。有个trick就是用原数组grid来记录这个点能到几个building，能到一个building就+1。bfs中传入k代表遍历之前一个building时没有去过的点不用去了，相当于visited数组。

Code:

```
public class Solution {
    class Tuple {
        public int x;
        public int y;
        public int dist;
        public Tuple(int x, int y, int dist) {
            this.x = x;
            this.y = y;
            this.dist = dist;
        }
    }
    int[][] dir = new int[][] {{1, 0},{-1, 0},{0, 1},{0, -1}};
    public int shortestDistance(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        int[][] dist = new int[m][n];
        List<Tuple> buildings = new ArrayList<>();
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) buildings.add(new Tuple(i,
j, 0));
                grid[i][j] = -grid[i][j];
            }
        }
        for (int k = 0; k < buildings.size(); k++) {
            bfs(buildings.get(k), k, dist, grid, m, n);
        }
        int res = -1;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == buildings.size() && (res < 0 |
| dist[i][j] < res)) res = dist[i][j];
            }
        }
        return res;
    }
    private void bfs(Tuple root, int k, int[][] dist, int[][] gr
id, int m, int n) {
        Queue<Tuple> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            Tuple b = queue.poll();
            dist[b.x][b.y] += b.dist;
            for (int[] d : dir) {
                int x = b.x + d[0], y = b.y + d[1];
                if (x >= 0 && y >= 0 && x < m && y < n && grid[x
][y] == k) {
                    grid[x][y] = k + 1;
                    queue.add(new Tuple(x, y, b.dist + 1));
                }
            }
        }
    }
}
```

Third Maximum Number

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

Example 1:

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

Output: 1

Explanation: The third maximum is 1.
```

Example 2:

```
Input: [1, 2]

Output: 2

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

Example 3:

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

Output: 1

Explanation: Note that the third maximum here means the third ma
ximum distinct number.
Both numbers with value 2 are both considered as second maximum.
```

Tips:

用heap或者普通做都行。

Code：

```
public class Solution {
    public int thirdMax(int[] nums) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        Set<Integer> set = new HashSet<>();
        for (int i : nums) {
            if (!set.contains(i)) {
                pq.offer(i);
                set.add(i);
                if (pq.size() > 3) {
                    set.remove(pq.poll());
                }
            }
        }
        if (pq.size() < 3) {
            while (pq.size() > 1) {
                pq.poll();
            }
        }
        return pq.peek();
    }
}
```

Binary Search

Guess Number Higher or Lower

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

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

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

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

```
-1 : My number is lower
 1 : My number is higher
 0 : Congrats! You got it!
```

Example:

```
n = 10, I pick 6.

Return 6.
```

Tips:

非常简单，用简单的二分法就能做，需要注意的是先算guess(n) == 0的情况，不然会超时。还有就是注意guess(n)的定义。

Code:

```
/* The guess API is defined in the parent class GuessGame.
   @param num, your guess
   @return -1 if my number is lower, 1 if my number is higher, o
therwise return 0
      int guess(int num); */

public class Solution extends GuessGame {
    public int guessNumber(int n) {
        if (guess(n) == 0) {
            return n;
        }
        int start = 1;
        int end = n;
        while (start < end) {
            int mid = start + (end - start) / 2;
            if (guess(mid) == 0) {
                return mid;
            } else if (guess(mid) == -1) {
                end = mid;
            } else {
                start = mid;
            }
        }
        return start;
    }
}
```

Pow(x, n)

Implement pow(x, n).

Tips:

就是算x的n次方，注意如果n是负的就n=-n然后x=1/x， 然后判断n的奇偶性。因为是double所以注意边界条件。

Code:

```
public class Solution {
    public double myPow(double x, int n) {
        if(n == 0)
            return 1;
        if (n == -2147483648) {
            n = 2147483647;
            x = 1 / x;
            return x * x * myPow(x * x, (n - 1)/2);
        }
        if(n < 0){
            n = -n;
            x = 1 / x;
        }
        return (n % 2 == 0) ? myPow (x * x, n / 2) : x * myPow(x
 * x, n / 2);
    }
}
```

别人的做法：

```
1.double myPow
double myPow(double x, int n) {
    if(n==0) return 1;
    double t = myPow(x,n/2);
    if(n%2) return n<0 ? 1/x*t*t : x*t*t;
    else return t*t;
}
2.double x
double myPow(double x, int n) {
    if(n==0) return 1;
    if(n<0){
        n = -n;
        x = 1/x;
    }
    return n%2==0 ? myPow(x*x, n/2) : x*myPow(x*x, n/2);
}
3.iterative one
double myPow(double x, int n) {
    if(n==0) return 1;
    if(n<0) {
        n = -n;
        x = 1/x;
    }
    double ans = 1;
    while(n>0){
        if(n&1) ans *= x;
        x *= x;
        n >>= 1;
    }
    return ans;
}
4.bit operation
```

Find Peak

A peak element is an element that is greater than its neighbors.

Given an input array where num[i] ≠ num[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞.

For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

click to show spoilers.

Note:

Your solution should be in logarithmic complexity.

Tips:

binary search. For each middle position, we compare it with its neighbors, if it is larger than both of its neighbors, just return it.

Code:

```
public class Solution {
    public int findPeakElement(int[] nums) {
        if (nums == null || nums.length ==0) {
            return -1;
        }
        int start = 0;
        int end = nums.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] > nums[mid - 1] && nums[mid] > nums[mi
d + 1]) {
                return mid;
            } else if (nums[mid] < nums[mid - 1]) {
                end = mid;
            } else {
                start = mid;
            }
        }
        return (nums[start] > nums[end] ? start : end);
    }
}
```

Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:

Given target value is a floating point.

You are guaranteed to have only one unique value in the BST that is closest to the target.

Tips:

注意BST的性质，递归来做。

Code:

```
public class Solution {
    public int closestValue(TreeNode root, double target) {
        int a = root.val;
        TreeNode kid = target < a ? root.left : root.right;
        if (kid == null) return a;
        int b = closestValue(kid, target);
        return Math.abs(a - target) < Math.abs(b - target) ? a :
 b;
    }
}
```

Smallest Rectangle Enclosing Black Pixels

An image is represented by a binary matrix with 0 as a white pixel and 1 as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location (x, y) of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.

For example, given the following image:

```
[
  "0010",
  "0110",
  "0100"
]
```

and x = 0, y = 2, Return 6.

Tips:

找到包含所有black pixel的最小矩形。这里我们用二分查找。因为给定black pixel点(x，y)，并且所有black pixel都是联通的，以row search为例， 所有含有black pixel的column，映射到row x上时，必定是连续的。这样我们可以使用binary search，在0到y里面搜索最左边含有black pixel的一列。接下来可以继续搜索上下和右边界。搜索右边界和下边界的时候，其实我们要找的是第一个'0'，所以要传入一个boolean变量searchLo来判断。

Suppose we have a 2D array

```
"000000111000000"
"000000101000000"
"000000101100000"
"000001100100000"
```

Imagine we project the 2D array to the bottom axis with the rule "if a column has any black pixel it's projection is black otherwise white". The projected 1D array is

"000001111100000" Theorem

If there are only one black pixel region, then in a projected 1D array all the black pixels are connected.

Proof by contradiction

Assume to the contrary that there are disconnected black pixels at i and j where i < j in the 1D projection array. Thus there exists one column k, k in (i, j) and and the column k in the 2D array has no black pixel. Therefore in the 2D array there exists at least 2 black pixel regions separated by column k which contradicting the condition of "only one black pixel region".

Therefore we conclude that all the black pixels in the 1D projection array is connected. This means we can do a binary search in each half to find the boundaries, if we know one black pixel's position. And we do know that.

To find the left boundary, do the binary search in the [0, y) range and find the first column vector who has any black pixel.

To determine if a column vector has a black pixel is O(m) so the search in total is O(m log n)

We can do the same for the other boundaries. The area is then calculated by the boundaries. Thus the algorithm runs in **O(m log n + n log m)**

Code:

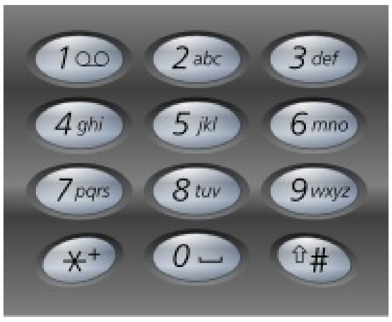
```
private char[][] image;
    public int minArea(char[][] iImage, int x, int y) {
        image = iImage;
        int m = image.length, n = image[0].length;
        int left = searchColumns(0, y, 0, m, true);
        int right = searchColumns(y + 1, n, 0, m, false);
        int top = searchRows(0, x, left, right, true);
        int bottom = searchRows(x + 1, m, left, right, false);
        return (right - left) * (bottom - top);
    }
    private int searchColumns(int i, int j, int top, int bottom,
 boolean opt) {
        while (i != j) {
            int k = top, mid = (i + j) / 2;
            while (k < bottom && image[k][mid] == '0') ++k;
            if (k < bottom == opt)
                j = mid;
            else
                i = mid + 1;
        }
        return i;
    }
    private int searchRows(int i, int j, int left, int right, bo
olean opt) {
        while (i != j) {
            int k = left, mid = (i + j) / 2;
            while (k < right && image[mid][k] == '0') ++k;
            if (k < right == opt)
                j = mid;
            else
                i = mid + 1;
        }
        return i;
    }
```

Graph

Letter Combinations of a Phone Number

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

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



Input:Digit string "23" Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
Note: Although the above answer is in lexicographical order, your answer could be in any order you want.

Tips:

FIFO queue, faster: 以234为例，先加入abc，然后提出a，加d，e，f存，再提出b,c,加d,e,f存，再提出ad，存adg，adh，adi,以此类推。

DFS：就是普通的dfs。。。

Code：

FIFO queue：

```
public class Solution {
    public List<String> letterCombinations(String digits) {
        List<String> ans = new ArrayList<String>();
        if (digits.length() == 0) {
            return ans;
        }
        String[] mapping = new String[] {"0", "1", "abc", "def",
 "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
        ans.add("");
        for (int i = 0; i < digits.length(); i++) {
            int cur = Character.getNumericValue(digits.charAt(i)
);
            while (ans.get(0).length() == i) {
                String prev = ans.remove(0);
                for (char c : mapping[cur].toCharArray()) {
                    ans.add(prev + c);
                }
            }
        }
        return ans;
    }
}
```

DFS：

```
   public class Solution {
        private static final String[] KEYS = { "", "", "abc", "d
ef", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

        public List<String> letterCombinations(String digits) {
            List<String> ret = new LinkedList<String>();
            combination("", digits, 0, ret);
            return ret;
        }

        private void combination(String prefix, String digits, i
nt offset, List<String> ret) {
            if (offset >= digits.length()) {
                ret.add(prefix);
                return;
            }
            String letters = KEYS[(digits.charAt(offset) - '0')]
;
            for (int i = 0; i < letters.length(); i++) {
                combination(prefix + letters.charAt(i), digits,
offset + 1, ret);
            }
        }
    }
```

Binary Tree Paths

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

For example, given the following binary tree:

```
   1
 /   \
2     3
 \
  5
```

All root-to-leaf paths are:

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

Tips:

简单的DFS，不需要backtracking

Code：

recursion：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> result = new ArrayList<>();
        if (root == null) {
            return result;
        }
        dfs(root, result, "");
        return result;
    }
    private void dfs (TreeNode root, List<String> result, String
 s) {
        if(root.left == null && root.right==null){
            result.add(s+String.valueOf(root.val));
        }
        if(root.left != null) {
            dfs(root.left, result, s+String.valueOf(root.val)+"-
>");
        }
        if(root.right !=null) {
            dfs(root.right, result, s+String.valueOf(root.val)+"
->");
        }
    }
}
```

stack：

```
public class Solution {
    public List<String> binaryTreePaths(TreeNode root) {

        List<String> res = new ArrayList<String>();
        Map<TreeNode, String> nodeMap = new HashMap<TreeNode, St
ring>();

        if (null == root) return res;

        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root);
        nodeMap.put(root, String.valueOf(root.val));

        while(!stack.isEmpty()) {
            TreeNode node = stack.pop();

            if ((null == node.left) && (null == node.right)) {
                res.add(nodeMap.get(node));
            }
            if (null != node.left) {
                stack.push(node.left);
                nodeMap.put(node.left, nodeMap.get(node)+"->" +
String.valueOf(node.left.val));
            }
            if (null != node.right) {
                stack.push(node.right);
                nodeMap.put(node.right, nodeMap.get(node)+"->" +
 String.valueOf(node.right.val));
            }
            nodeMap.remove(node);

        }
        return res;
    }
}
```

Walls and Gates

You are given a m x n 2D grid initialized with these three possible values.

```
-1 - A wall or an obstacle.
0 - A gate.
INF - Infinity means an empty room.
```

We use the value 231 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

For example, given the 2D grid:

```
INF  -1  0  INF
INF INF INF  -1
INF  -1 INF  -1
  0  -1 INF INF
```

After running your function, the 2D grid should be:

```
  3  -1   0   1
  2   2   1  -1
  1  -1   2  -1
  0  -1   3   4
```

Tips

- 这里附上了BFS和DFS的解法，但是显然BFS更快。最先找到gate，然后以gate为root进行BFS遍历，叶子节点为四个方向。
- 最巧妙地部分是这里定义了static final d，来确定四个方向的位置，即通过用i，j +/- 1的方式来得到[i, j+1],[i+1,j],[i, j-1], [i-1, j]。
- 注意在遍历四个方向时不要出界。 *

Code:

先附上bfs，dfs在下面。

```
public class Solution {
    // use this d we can get the pos of the four room adjacent t
o x
    public static final int[] d = {0, 1, 0, -1, 0};

    public void wallsAndGates(int[][] rooms) {
        if (rooms.length == 0 || rooms[0].length == 0) {
            return;
        }
        Deque<Integer> queue = new ArrayDeque<>();
        // put the gate into the queue
        for (int i = 0; i < rooms.length; ++i) {
            for (int j = 0; j < rooms[0].length; ++j) {
                if (rooms[i][j] == 0) {
                    queue.add(i * rooms[0].length + j);
                }
            }
        }

        while (!queue.isEmpty()) {
            int x = queue.remove();
            //define the position of the gate
            //i is the row index and j is the col index
            int i = x / rooms[0].length, j = x % rooms[0].length
;
            for (int k = 0; k < 4; ++k) {
                int m = i + d[k], n = j + d[k + 1];

                // keep the room inside the border and if it's e
mpty, put the distance
                if (m >= 0 && m < rooms.length && n >= 0 && n <
rooms[0].length && rooms[m][n] == Integer.MAX_VALUE) {
                    rooms[m][n] = rooms[i][j] + 1;
                    queue.add(m * rooms[0].length + n);
                }
            }
        }
    }
}
```

也附上dfs解法：

```
private static int[] d = {0, 1, 0, -1, 0};

public void wallsAndGates(int[][] rooms) {
    for (int i = 0; i < rooms.length; i++)
        for (int j = 0; j < rooms[0].length; j++)
            if (rooms[i][j] == 0) dfs(rooms, i, j);
}

public void dfs(int[][] rooms, int i, int j) {
    for (int k = 0; k < 4; ++k) {
        int p = i + d[k], q = j + d[k + 1];
        if (0<= p && p < rooms.length && 0<= q && q < rooms[0].l
ength && rooms[p][q] > rooms[i][j] + 1) {
            rooms[p][q] = rooms[i][j] + 1;
            dfs(rooms, p, q);
        }
    }
}
```

Minimum Height Trees

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.

Example 1:

Given n = 4, edges = [[1, 0], [1, 2], [1, 3]]

```
    0
    |
    1
   / \
  2   3
```

return [1]

Example 2:

Given n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

```
 0  1  2
  \ | /
    3
    |
    4
    |
    5
```

return [3, 4]

Tips:

思路

```
1.类似剥洋葱的方法，就是一层一层的褪去叶节点，最后剩下的一个或两个节点就是我
们要求的最小高度树的根节点。

2.先建立 List<HashSet<Integer>> adjacent， 来存储所有点的edge。
  实际是建立一个类似hashmap的东西，用ArrayList里面的位置（index）来作为k
ey, 所以可以使用get(edge[0])这种来添加edge。

3.然后找出所有的叶子节点，即set.size() == 1的点，将这些点加入叶子节点的Ar
rayList。

4.之后进入循环，在n > 2的条件下一层一层把叶子节点剥掉。ArrayList newLeav
e来作为下一层叶子节点，取代leaves，再剥。
```

注意点

```
 1.return Collections.singletonList(0);   singletonList

 2.int neighbor = adjacent.get(leave).iterator().next(); 这是hash
set里面往下一个的办法
```

The time complexity and space complexity are both O(n).

We start from every end, by end we mean vertex of degree 1 (aka leaves). We let the pointers move the same speed. When two pointers meet, we keep only one of them, until the last two pointers meet or one step away we then find the roots.

It is easy to see that the last two pointers are from the two ends of the longest path in the graph.

The actual implementation is similar to the BFS topological sort. 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! Note that for a tree we always have V = n, E = n-1.

Code:

```
public class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges
) {
        if (n == 1) {
            return Collections.singletonList(0);
        }
        List<HashSet<Integer>> adjacent = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjacent.add(new HashSet<Integer>());
        }
        //add edges to each node
        for (int[] edge : edges) {
            adjacent.get(edge[0]).add(edge[1]);
            adjacent.get(edge[1]).add(edge[0]);
        }
        // Through set.size() to judge if the node is a leave
        List<Integer> leaves = new ArrayList<>();
        for (int i = 0; i < n; ++i) {
            if (adjacent.get(i).size() == 1) {
                leaves.add(i);
            }
        }
        while (n > 2) {
            List<Integer> newLeaves = new ArrayList<>();
            for (int leave : leaves) {
                int neighbor = adjacent.get(leave).iterator().ne
xt();
                adjacent.get(neighbor).remove(leave);
                n--;
                if (adjacent.get(neighbor).size() == 1) {
                    newLeaves.add(neighbor);
                }
            }
            leaves = newLeaves;
        }
        return leaves;
    }
}
```

Verify Preorder Serialization of a Binary Tree

One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.

```
     _9_
    /   \
   3     2
  / \   / \
 4   1  #  6
/ \ / \   / \
# # # #   # #
```

For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node.

Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.

Each comma separated value in the string must be either an integer or a character '#' representing null pointer.

You may assume that the input format is always valid, for example it could never contain two consecutive commas such as "1,,3".

Tips:

思路：

- 栈
- 入度和出度的差

1.栈

这个方法简单的说就是不断的砍掉叶子节点。最后看看能不能全部砍掉。

已例子一为例，：”9,3,4,#,#,1,#,#,2,#,6,#,#” 遇到x # #的时候，就把它变为 #

我模拟一遍过程：

```
9,3,4,#,# => 9,3,# 继续读
9,3,#,1,#,# => 9,3,#,# => 9,# 继续读
9,#2,#,6,#,# => 9,#,2,#,# => 9,#,# => #
```

2.In a binary tree, if we consider null as leaves, then

all non-null node provides 2 outdegree and 1 indegree (2 children and 1 parent), except root all null node provides 0 outdegree and 1 indegree (0 child and 1 parent).

Suppose we try to build this tree. During building, we record the difference between out degree and in degree diff = outdegree - indegree. When the next node comes, we then decrease diff by 1, because the node provides an in degree. If the node is not null, we increase diff by2, because it provides two out degrees. If a serialization is correct, diff should never be negative and diff will be zero when finished.

翻译：对于二叉树，我们把空的地方也作为叶子节点（如题目中的#），那么有

```
所有的非空节点提供2个出度和1个入度(2 childern and 1 parent)（根除外）
所有的空节点但提供0个出度和1个入度(1 parent)
```

我们在遍历的时候，计算diff = outdegree – indegree. 当一个节点出现的时候，diff – 1，因为它提供一个入度；当节点不是#的时候，diff+2(提供两个出度) 如果序列式合法的，那么遍历过程中diff >=0 且最后结果为0.

Stack:

```
public class Solution {
    public boolean isValidSerialization(String preorder) {
        if (preorder == null || preorder.length() == 0){
            return true;
        }
        String[] nodes = preorder.split(",");
        Stack<String> stack = new Stack<>();
        for (String node : nodes) {
            while (node.equals("#") && !stack.isEmpty() && stack
.peek().equals("#")) {
                stack.pop();
                if (stack.isEmpty()) {
                    return false;
                }
                stack.pop();
            }
            stack.push(node);
        }
        return stack.size() == 1 && stack.peek().equals("#");
    }
}
```

dietpepsi:

```
public boolean isValidSerialization(String preorder) {
    String[] nodes = preorder.split(",");
    int diff = 1;
    for (String node: nodes) {
        if (--diff < 0) return false;
        if (!node.equals("#")) diff += 2;
    }
    return diff == 0;
}
```

Binary Tree Longest Consecutive Sequence

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

For example,

```
   1
    \
     3
    / \
   2   4
        \
         5
```

Longest consecutive sequence path is 3-4-5, so return 3.

```
   2
    \
     3
    /
   2
  /
 1
```

Longest consecutive sequence path is 2-3,not3-2-1, so return 2.

Tips:

简单的DFS，重点是local_max,即先算出这次遍历的最大长度，然后再和总最大长度进行比较。

Code：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int longestConsecutive(TreeNode root) {
        int result = dfs(root, -1, 0);
        return result;
    }
    private int dfs(TreeNode root, int prev, int local_max) {
        if (root == null) {
            return local_max;
        }
        if (root.val != prev + 1) {
            local_max = 1;
        } else {
            local_max++;
        }
        int left = dfs(root.left, root.val, local_max);
        int right = dfs(root.right, root.val, local_max);
        int result = Math.max(local_max, Math.max(left, right));
        return result;
    }
}
```

Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:

```
     0          3
     |          |
     1 --- 2    4
```

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.

Example 2:

```
     0           4
     |           |
     1 --- 2 --- 3
```

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1.

Note: 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.

LintCode 类似题：Find the Connected Component in the Undirected Graph

http://www.jiuzhang.com/solutions/find-the-connected-component-in-the-undirected-graph/

Tips:

- 解法1：并查集（UNION-FIND)

  for(int[] edge : edges) 查找每个edge两边的nodes的father，如果发现还没有合并，就Union，把第一个的father设为第二个，即map.put(fa_x, fa_y);

用n来代表最后的联通块个数，所以合并一次就要n--。最后返回n即可。

- 解法2：DFS
- 解法3：BFS

Code:

解法1（并查集）：

```
public class Solution {
    public int countComponents(int n, int[][] edges) {
        if (n <= 1) {
            return n;
        }
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            map.put(i, i);
        }
        for (int[] edge : edges) {
            int fa_x = find(map, edge[0]);
            int fa_y = find(map, edge[1]);
            if (fa_x != fa_y) {
                map.put(fa_x, fa_y);
                n--;
            }
        }
        return n;
    }

    private int find(HashMap<Integer, Integer> map, int x) {
        int father = map.get(x);
        while (father != map.get(father)) {
            father = map.get(father);
        }
        return father;
    }
}
```

Union Find 模板：

```
HashMap<Integer, Integer> map = new HashMap<>();

int find(HashMap<Integer, Integer> map, int x) {
     int father = map.get(x);
        while (father != map.get(father)) {
            father = map.get(father);
        }
     return father;
}

void union(int x, int y) {
    int fa_x = find(map, edge[0]);
    int fa_y = find(map, edge[1]);
    if (fa_x != fa_y) {
       map.put(fa_x, fa_y);
    }
}
```

解法2（DFS）：

```
  public class Solution {
    public int countComponents(int n, int[][] edges) {
        if (n <= 1)
            return n;
        Map<Integer, List<Integer>> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            map.put(i, new ArrayList<>());
        }
        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
            map.get(edge[1]).add(edge[0]);
        }
        Set<Integer> visited = new HashSet<>();
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (visited.add(i)) {
                dfsVisit(i, map, visited);
                count++;
            }
        }
        return count;
    }

    private void dfsVisit(int i, Map<Integer, List<Integer>> map
, Set<Integer> visited) {
        for (int j : map.get(i)) {
            if (visited.add(j))
                dfsVisit(j, map, visited);
        }
    }
}
```

解法3（BFS）：

```
 public class Solution {

    public int countComponents(int n, int[][] edges) {
        if (n <= 1) {
            return n;
        }
        List<List<Integer>> adjList = new ArrayList<List<Integer
>>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<Integer>());
        }
        for (int[] edge : edges) {
            adjList.get(edge[0]).add(edge[1]);
            adjList.get(edge[1]).add(edge[0]);
        }
        boolean[] visited = new boolean[n];
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                count++;
                Queue<Integer> queue = new LinkedList<Integer>()
;
                queue.offer(i);
                while (!queue.isEmpty()) {
                    int index = queue.poll();
                    visited[index] = true;
                    for (int next : adjList.get(index)) {
                        if (!visited[next]) {
                            queue.offer(next);
                        }
                    }
                }
            }
        }

        return count;
    }
}
```

Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

Hint:

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], what should your return? Is this case a valid tree? 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.” Note: 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.

Tips:

UF方法：复杂度 O(n)

- 与Number of connected components in an Undirected Graph 非常类似，只是输出结果是boolean。
- 仍旧计算无向联通块的个数，如果最后结果为1则返回true，否则返回false。
- 不同的是在对edge做UF操作的时候需要加一个判断来判断是否构成cycle，如果构成则不是树。

```
    if (n == 1) {
        return true;
    }
```

DFS和BFS方法简单贴了一下。

Code:

UF:

```
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        if (n <= 1) {
            return true;
        }
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            map.put(i, i);
        }
        for (int[] edge : edges) {
            int fa_x = find(edge[0], map);
            int fa_y = find(edge[1], map);
            if (fa_x == fa_y) {
                return false;
            }
            if (fa_x != fa_y) {
                map.put(fa_x, fa_y);
                n--;
            }
        }
        if (n == 1) {
            return true;
        }
        return false;
    }
    private int find(int x, HashMap<Integer, Integer> map) {
        int father = map.get(x);
        while (father != map.get(father)) {
            father = map.get(father);
        }
        //compressed:
        while (x != map.get(x)) {
            temp = map.get(x);
            map.put(x, father);
            x = temp;
        }
        //end
        return father;
    }
}
```

DFS:

```
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        // initialize adjacency list
        List<List<Integer>> adjList = new ArrayList<List<Integer
>>(n);

        // initialize vertices
        for (int i = 0; i < n; i++)
            adjList.add(i, new ArrayList<Integer>());

        // add edges
        for (int i = 0; i < edges.length; i++) {
            int u = edges[i][0], v = edges[i][1];
            adjList.get(u).add(v);
            adjList.get(v).add(u);
        }

        boolean[] visited = new boolean[n];

        // make sure there's no cycle
        if (hasCycle(adjList, 0, visited, -1))
            return false;

        // make sure all vertices are connected
        for (int i = 0; i < n; i++) {
            if (!visited[i])
                return false;
        }

        return true;
    }

    // check if an undirected graph has cycle started from verte
x u
```

```
    boolean hasCycle(List<List<Integer>> adjList, int u, boolean
[] visited, int parent) {
        visited[u] = true;

        for (int i = 0; i < adjList.get(u).size(); i++) {
            int v = adjList.get(u).get(i);

            if ((visited[v] && parent != v) || (!visited[v] && h
asCycle(adjList, v, visited, u)))
                return true;
        }

        return false;
    }
}
```

BFS:

```
private boolean valid(int n, int[][] edges)
{
    // build the graph using adjacent list
    List<Set<Integer>> graph = new ArrayList<Set<Integer>>();
    for(int i = 0; i < n; i++)
        graph.add(new HashSet<Integer>());
    for(int[] edge : edges)
    {
        graph.get(edge[0]).add(edge[1]);
        graph.get(edge[1]).add(edge[0]);
    }

    // no cycle
    boolean[] visited = new boolean[n];
    Queue<Integer> queue = new ArrayDeque<Integer>();
    queue.add(0);
    while(!queue.isEmpty())
    {
        int node = queue.poll();
        if(visited[node])
            return false;
        visited[node] = true;
        for(int neighbor : graph.get(node))
        {
            queue.offer(neighbor);
            graph.get(neighbor).remove((Integer)node);
        }
    }

    // fully connected
    for(boolean result : visited)
    {
        if(!result)
            return false;
    }

    return true;
}
```

Number if Islands

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
11110
11010
11000
00000
Answer: 1
```

Example 2:

```
11000
11000
00100
00011
Answer: 3
```

Tips:

- 解法1 UnionFind:

  这个和找无向联通快差不多，就是要把二维的matrix转换成一维的然后放入map，公式是

  ```
  1d = n * i + j
  ```

  也要设置一个二维数组distance，使得用i + d[0] 和 j + d[1]能找到这个岛中间的四块区域，公式在代码里找。

对于for(int[] edge : edges) 且要是岛屿（=='1'），查找每个岛屿两边的nodes(i + d[0], j + d[1])的father，如果发现还没有合并，就Union，把第一个的father设为第二个，即map.put(fa_x, fa_y)，count--;

- 解法2 DFS：

  找到一个岛之后result++, 再往四周进行DFS，把dfs过程中碰到的岛都设为0。然后再找下一个岛，再去DFS，这样保证加的岛都不相邻。

- 解法3 BFS：

  就是把DFS换成BFS呗

Code:

UnionFind:

```
public class Solution {
    int m, n;
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length =
= 0)  {
            return 0;
        }
        int[][] distance = {{1,0},{-1,0},{0,1},{0,-1}};
        HashMap<Integer, Integer> map = new HashMap<>();
        m = grid.length;
        n = grid[0].length;
        int count = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    count++;
                    map.put(i * n + j, i * n + j);
                }
            }
        }
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] != '1') {
                    continue;
                }
                for (int[] d : distance) {
                    int x = i + d[0];
                    int y = j + d[1];
                    if (x >= 0 && y >= 0 && x < m && y < n && gr
id[x][y] == '1') {
                        int fa_x = find(i * n + j, map);
                        int fa_y = find(x * n + y, map);
                        if (map.get(fa_x) != map.get(fa_y)) {
                            map.put(fa_x, fa_y);
                            count--;
                        }
                    }
                }
            }
        }
        return count;
    }
    private int find (int pos, HashMap<Integer, Integer> map) {
        int father = map.get(pos);
        while (father != map.get(father)) {
            father = map.get(father);
        }
        return father;
    }
}
```

DFS:

```
public class Solution {
    private int m;
    private int n;
    public int numIslands(char[][] grid) {
        m = grid.length;
        if (m == 0) {
            return 0;
        }
        n = grid[0].length;
        if (n == 0) {
            return 0;
        }
        int result = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    result++;
                    dfs(i, j, grid);
                }
            }
        }
        return result;
    }
    private void dfs(int i, int j, char[][] grid) {
        if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] !=
'1') {
            return;
        }
        grid[i][j] = 0;
        dfs(i + 1, j, grid);
        dfs(i, j + 1, grid);
        dfs(i - 1, j, grid);
        dfs(i, j - 1, grid);
    }
}
```

BFS:

```
public int numIslands(char[][] grid) {
        int count=0;
        for(int i=0;i<grid.length;i++)
            for(int j=0;j<grid[0].length;j++){
                if(grid[i][j]=='1'){
                    bfsFill(grid,i,j);
                    count++;
                }
            }
        return count;
    }
    private void bfsFill(char[][] grid,int x, int y){
        grid[x][y]='0';
        int n = grid.length;
        int m = grid[0].length;
        LinkedList<Integer> queue = new LinkedList<Integer>();
        int code = x*m+y;
        queue.offer(code);
        while(!queue.isEmpty())
        {
            code = queue.poll();
            int i = code/m;
            int j = code%m;
            if(i>0 && grid[i-1][j]=='1')    //search upward and mark
 adjacent '1's as '0'.
            {
                queue.offer((i-1)*m+j);
                grid[i-1][j]='0';
            }
            if(i<n-1 && grid[i+1][j]=='1')  //down
            {
                queue.offer((i+1)*m+j);
                grid[i+1][j]='0';
            }
            if(j>0 && grid[i][j-1]=='1')  //left
            {
                queue.offer(i*m+j-1);
                grid[i][j-1]='0';
            }
            if(j<m-1 && grid[i][j+1]=='1')  //right
            {
                queue.offer(i*m+j+1);
                grid[i][j+1]='0';
            }
        }
    }
```

Number of Islands II

Number of Islands II

A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example:

Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]]. Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).

```
0 0 0
0 0 0
0 0 0
```

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

```
1 0 0
0 0 0   Number of islands = 1
0 0 0
```

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

```
1 1 0
0 0 0   Number of islands = 1
0 0 0
```

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

```
1 1 0
0 0 1   Number of islands = 2
0 0 0
```

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

```
1 1 0
0 0 1   Number of islands = 3
0 1 0
```

We return the result as an array: [1, 1, 2, 3]

TIPS:

和 Number of Islands 一毛一样，只是这次DFS和BFS的复杂度都变很高，要**O(knm)**。所以用UF做法，操作第K个点的复杂度为**O(K)**。 设count为0，每加一个岛做一次UF，如果发现与别的岛相连就count--。注意这里用了compress find，时间复杂度大大降低哟，因为compress的平摊复杂度是**O(1)**。

Code:

```
public class Solution {
    public List<Integer> numIslands2(int m, int n, int[][] posit
ions) {
        List<Integer> result = new ArrayList<Integer>();
        int count = 0;
        HashMap<Integer, Integer> map = new HashMap<>();
        int[][] distance = {{1,0},{-1,0},{0,1},{0,-1}};
        for (int[] pos : positions) {
            int x = pos[0];
            int y = pos[1];
            map.put(x * n + y, x * n + y);
            count++;
            for (int[] d : distance) {
                int nx = x + d[0];
                int ny = y + d[1];
                int npos = nx * n + ny;
                if (nx < 0 || ny < 0 || nx >= m || ny >= n || !m
ap.containsKey(npos)) {
                    continue;
                }
                int fa_x = find(x * n + y, map);
                int fa_y = find(npos, map);
                if (fa_x != fa_y) {
                    map.put(fa_x, fa_y);
                    count--;
                }
            }
            result.add(count);
        }
        return result;
    }

    private int find(int pos, HashMap<Integer, Integer> map) {
        int father = map.get(pos);
        while (father != map.get(father)) {
            father = map.get(father);
        }
        // find compress here
        int temp = -1;
        while (pos != map.get(pos)) {
            temp = map.get(pos);
            map.put(pos, father);
            pos = temp;
        }
        //end of the compress
        return father;
    }
}
```

Evaluate Division

Equations are given in the format A / B = k, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.

Example:

```
Given a / b = 2.0, b / c = 3.0.
queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x =
 ? .
return [6.0, 0.5, -1.0, 1.0, -1.0 ].
```

The values are **positive**. This represents the equations..

According to the example above:

```
equations = [ ["a", "b"], ["b", "c"] ],
values = [2.0, 3.0],
queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x"
, "x"] ].
```

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

Tips:

- 解法1：

最直白的解法。先过滤掉没有见过的字符，形成字典，然后开始算结果。如果发现问题中设计的两个字串有一个没有出现在字典里，就返回-1，其他情况则进入DFS。

DFS中建立一个新的stack，针对每个queary依次搜索每个equation，如果equation中有queary种两个字串的任意一个，就接着往下搜，temp = helper * value 或 helper / value直到搜出结果。

值得注意的是，这里有 if(temp > 0) return; else stack pop()的语句，是因为题中表示所有等式的值都是positive的，所以如果出现负数，则一定是在这次DFS中直到叶子都没找到结果，返回了-1。所以此时要pop之后接着搜答案。

- 解法2：

先把所有equation都存进map里，这样在queary大的时候比解法一快。

- 解法3：并查集

应该就是在找老大哥合并的过程，只是在compress find的时候和union的时候带着value呗~

Code:

解法1：

```
public class Solution {
    public double[] calcEquation(String[][] equations, double[]
values, String[][] query) {
        double[] result = new double[query.length];
        // filter unexpected words
        // 过滤掉没有遇见过的字符
        Set<String> words = new HashSet<>();
        for (String[] strs : equations) {
            words.add(strs[0]);
            words.add(strs[1]);
        }
        for (int i = 0; i < query.length; ++i) {
            String[] keys = query[i];
            if (!words.contains(keys[0]) || !words.contains(keys
[1])) result[i] = -1.0d;
            else {
                Stack<Integer> stack = new Stack<>();
                result[i] = helper(equations, values, keys, stac
k);
            }
        }
        return result;
    }
    public double helper(String[][] equations, double[] values, 
String[] keys, Stack<Integer> stack) {
        // 直接查找，key的顺序有正反
        // look up equations directly
        for (int i = 0; i < equations.length; ++i) {
            if (equations[i][0].equals(keys[0]) && equations[i][
1].equals(keys[1])) return values[i];
            if (equations[i][0].equals(keys[1]) && equations[i][
1].equals(keys[0])) return 1 / values[i];
        }
        // lookup equations by other equations
        // 间接查找，key的顺序也有正反
        for (int i = 0; i < equations.length; ++i) {
            if (!stack.contains(i) && keys[0].equals(equations[i
][0])) {
                stack.push(i);
                double temp = values[i] * helper(equations, valu
es, new String[]{equations[i][1], keys[1]}, stack);
                if (temp > 0) return temp;
                else stack.pop();
            }
            if (!stack.contains(i) && keys[0].equals(equations[i
][1])) {
                stack.push(i);
                double temp = helper(equations, values, new Stri
ng[]{equations[i][0], keys[1]}, stack) / values[i];
                if (temp > 0) return temp;
                else stack.pop();
            }
        }
        // 查不到，返回-1
        return -1.0d;
    }
}
```

解法2：

```
public double[] calcEquation(String[][] equations, double[] valu
es, String[][] query) {
        // use table save string to integer
        Map<String, Integer> table = new HashMap<>();
        int len = 0;
        for (String[] strings : equations)
            for (String string : strings)
                if (!table.containsKey(string)) table.put(string
, len++);

        // init map by direct equation
        double[][] map = new double[len][len];
        for (int i = 0; i < len; ++i)
            for (int j = 0; j < len; ++j)
                map[i][j] = (i == j ? 1.0d : -1.0d);
        for (int i = 0; i < equations.length; ++i) {
            String[] keys = equations[i];
            int row = table.get(keys[0]);
            int col = table.get(keys[1]);
            map[row][col] = values[i];
            map[col][row] = 1 / values[i];
        }

        // floyd-warshall like algorithm
        for (int i = 0; i < len; ++i) {
            for (int j = 0; j < len; ++j) {
                for (int k = 0; k < len; ++k) {
                    if (map[j][i] >= 0d && map[i][k] >= 0d) {
                        map[j][k] = map[j][i] * map[i][k];
                        }
                }
            }
        }

        // query now
        double[] result = new double[query.length];
        for (int i = 0; i < query.length; ++i) {
            String[] keys = query[i];
            Integer row = table.get(keys[0]);
            Integer col = table.get(keys[1]);
            if (row == null || col == null) result[i] = -1.0d;
            else result[i] = map[row][col];
        }
        return result;
    }
```

解法3：

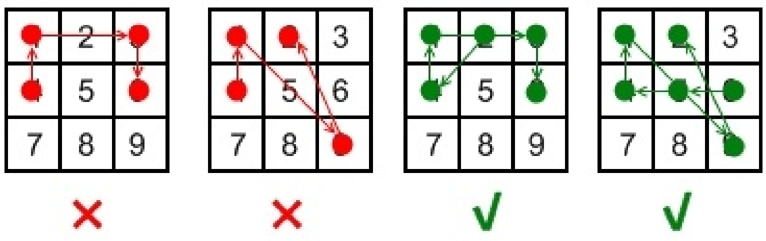
```
public double[] calcEquation(String[][] equations, double[] valu
es, String[][] query) {

        // map string to integer
        Map<String, Integer> mIdTable = new HashMap<>();
        int len = 0;
        for (String[] words : equations)
            for (String word : words)
                if (!mIdTable.containsKey(word)) mIdTable.put(wo
rd, len++);

        // init parent index and value
        Node[] nodes = new Node[len];
        for (int i = 0; i < len; ++i) nodes[i] = new Node(i);

        // union, you can take an example as follows
        // (a/b=3)->(c/d=6)->(b/d=12)
        for (int i = 0; i < equations.length; ++i) {
            String[] keys = equations[i];
            int k1 = mIdTable.get(keys[0]);
            int k2 = mIdTable.get(keys[1]);
            int groupHead1 = find(nodes, k1);
            int groupHead2 = find(nodes, k2);
            nodes[groupHead2].parent = groupHead1;
            nodes[groupHead2].value = nodes[k1].value * values[i
] / nodes[k2].value;
        }

        // query now
        double[] result = new double[query.length];
        for (int i = 0; i < query.length; ++i) {
            Integer k1 = mIdTable.get(query[i][0]);
            Integer k2 = mIdTable.get(query[i][1]);
            if (k1 == null || k2 == null) result[i] = -1d;
            else {
                int groupHead1 = find(nodes, k1);
                int groupHead2 = find(nodes, k2);
                if (groupHead1 != groupHead2) result[i] = -1d;
                else result[i] = nodes[k2].value / nodes[k1].val
ue;
            }
        }
        return result;
    }

    public int find(Node[] nodes, int k) {
        int p = k;
        while (nodes[p].parent != p) {
            p = nodes[p].parent;
            // compress
            nodes[k].value *= nodes[p].value;
        }
        // compress
        nodes[k].parent = p;
        return p;
    }

    private static class Node {
        int    parent;
        double value;

        public Node(int index) {
            this.parent = index;
            this.value = 1d;
        }
    }
```

Android unlock patterns

Given an Android 3x3 key lock screen and two integers m and n, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.

Rules for a valid pattern:

```
1.Each pattern must connect at least m keys and at most n keys.
2.All the keys must be distinct.
3.If the line connecting two consecutive keys in the pattern pas
ses through any other keys,
  the other keys must have previously selected in the pattern.
  No jumps through non selected key is allowed.
4.The order of keys used matters.
```



Explanation:

```
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
```

Invalid move: 4 - 1 - 3 - 6 Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: 4 - 1 - 9 - 2 Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: 2 - 4 - 1 - 3 - 6 Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: 6 - 5 - 4 - 1 - 9 - 2 Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example: Given m = 1, n = 1, return 9.

Tips:

这道题说的是安卓机子的解锁方法，有9个数字键，如果密码的长度范围在[m, n]之间，问所有的解锁模式共有多少种，注意题目中给出的一些非法的滑动模式。

那么我们先来看一下哪些是非法的，首先1不能直接到3，必须经过2，同理的有4到6，7到9，1到7，2到8，3到9，还有就是对角线必须经过5，例如1到9，3到7等。

我们建立一个二维数组jumps，用来记录两个数字键之间是否有中间键，然后再用一个一位数组visited来记录某个键是否被访问过，然后我们用递归来解，

我们先对1调用递归函数，在递归函数中，我们遍历1到9每个数字next，然后找他们之间是否有jump数字，如果next没被访问过，并且jump为0，或者jump被访问过，我们对next调用递归函数。

数字1的模式个数算出来后，由于1,3,7,9是对称的，所以我们乘4即可，然后再对数字2调用递归函数，2,4,6,9也是对称的，再乘4，最后单独对5调用一次，然后把所有的加起来就是最终结果了.

Code:

```
public class Solution {
    // cur: the current position
    // remain: the steps remaining
    int DFS(boolean vis[], int[][] skip, int cur, int remain) {
        if(remain < 0) return 0;
        if(remain == 0) return 1;
        vis[cur] = true;
        int rst = 0;
        for(int i = 1; i <= 9; ++i) {
            // If vis[i] is not visited and (two numbers are adj
acent or skip number is already visited)
            if(!vis[i] && (skip[cur][i] == 0 || (vis[skip[cur][i
]]))) {
                rst += DFS(vis, skip, i, remain - 1);
            }
        }
        vis[cur] = false;
        return rst;
    }

    public int numberOfPatterns(int m, int n) {
        // Skip array represents number to skip between two pair
s
        int[][] skip = new int[10][10];
        skip[1][3] = skip[3][1] = 2;
        skip[1][7] = skip[7][1] = 4;
        skip[3][9] = skip[9][3] = 6;
        skip[7][9] = skip[9][7] = 8;
        skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip
[3][7] = skip[7][3] = skip[4][6] = skip[6][4] = 5;
        boolean vis[] = new boolean[10];
        int rst = 0;
        // DFS search each length from m to n
        for(int i = m; i <= n; ++i) {
            rst += DFS(vis, skip, 1, i - 1) * 4;    // 1, 3, 7,
9 are symmetric
            rst += DFS(vis, skip, 2, i - 1) * 4;    // 2, 4, 6,
8 are symmetric
            rst += DFS(vis, skip, 5, i - 1);        // 5
        }
        return rst;
    }
}
```

Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

Tips:

用stack，先向左一路push到底，next()时pop一个出来，return its value 然后如果右边有再把右边push，右边的左边还有就把右边的左边push到底

I use Stack to store directed left children from root. When next() be called, I just pop one element and process its right child as new root. The code is pretty straightforward.

So this can satisfy **O(h) memory**, hasNext() in **O(1) time**, But next() is **O(h)** time.

The average time complexity of next() function is O(1) indeed in your case. As the next function can be called n times at most, and the number of right nodes in self.pushAll(tmpNode.right) function is maximal n in a tree which has n nodes, so the amortized time complexity is O(1).

Code:

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```
public class BSTIterator {
    private Stack<TreeNode> stack = new Stack<TreeNode>();

    public BSTIterator(TreeNode root) {
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode temp = stack.pop();
        pushAll(temp.right);
        return temp.val;
    }

    private void pushAll(TreeNode node) {
        while (node != null) {
            stack.push(node);
            node = node.left;
        }
    }
}

/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = new BSTIterator(root);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

Binary Tree Vertical Order Traversal

Given a binary tree, return the vertical order traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from left to right.

Examples:

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

```
   3
  /\
 /  \
 9  20
    /\
   /  \
  15   7
```

return its vertical order traversal as:

```
[
  [9],
  [3,15],
  [20],
  [7]
]
```

Given binary tree [3,9,8,4,0,1,7],

```
     3
    /\
   /  \
   9   8
  /\  /\
 /  \/  \
 4  01   7
```

return its vertical order traversal as:

```
[
  [4],
  [9],
  [3,0,1],
  [8],
  [7]
]
```

Given binary tree [3,9,8,4,0,1,7,null,null,null,2,5] (0's right child is 2 and 1's left child is 5),

```
     3
    /\
   /  \
   9   8
  /\  /\
 /  \/  \
 4  01   7
    /\
   /  \
   5   2
```

return its vertical order traversal as:

```
[
  [4],
  [9,5],
  [3,0,1],
  [8,2],
  [7]
]
```

Tips:

BFS，需要两个queue和一个HashMap。min和max存最小和最大的列数。两个queue同时FIFO，分别存这个node的值和他的col。

The following solution takes 5ms.

BFS, put node, col into queue at the same time Every left child access col - 1 while right child col + 1 This maps node into different col buckets Get col boundary min and max on the fly Retrieve result from cols Note that TreeMap version takes 9ms.

Code：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> verticalOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        Queue<Integer> cols = new LinkedList<>();
        HashMap<Integer, ArrayList<Integer>> map = new HashMap<>
();
        queue.offer(root);
        cols.offer(0);
        int min = 0;
        int max = 0;
        while (!queue.isEmpty()) {
            TreeNode curt = queue.poll();
            int col = cols.poll();
            if (!map.containsKey(col)) {
                map.put(col, new ArrayList<Integer>());
            }
            map.get(col).add(curt.val);
            if (curt.left != null) {
                cols.offer(col - 1);
                queue.offer(curt.left);
                min = Math.min(min, col - 1);
            }
            if (curt.right != null) {
                cols.offer(col + 1);
                queue.offer(curt.right);
                max = Math.max(max, col + 1);
            }
        }
        for (int i = min; i <= max; i++) {
            result.add(map.get(i));
        }
        return result;
    }
}
```

TreeMap Version:

```
public int index = 0;
public TreeMap<Integer, List<Integer>> tm;

public class Pair {
    TreeNode node;
    int index;
    public Pair(TreeNode n, int i) {
        node = n;
        index = i;
    }
}

public List<List<Integer>> verticalOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    tm = new TreeMap<Integer, List<Integer>>();
    if (root == null) return res;

    Queue<Pair> q = new LinkedList<Pair>();
    q.offer(new Pair(root, 0));

    while (!q.isEmpty()) {
        Pair cur = q.poll();
        if (!tm.containsKey(cur.index)) tm.put(cur.index, new Ar
rayList<Integer>());
        tm.get(cur.index).add(cur.node.val);

        if (cur.node.left != null) q.offer(new Pair(cur.node.lef
t, cur.index-1));
        if (cur.node.right != null) q.offer(new Pair(cur.node.ri
ght, cur.index+1));
    }

    for (int key : tm.keySet()) res.add(tm.get(key));
    return res;
}
```

Reconstruct Itinerary

Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

Note:

1.If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].

2.All airports are represented by three capital letters (IATA code).

3.You may assume all tickets form at least one valid itinerary.

Example 1:

tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]

Return ["JFK", "MUC", "LHR", "SFO", "SJC"].

Example 2:

tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

Return ["JFK","ATL","JFK","SFO","ATL","SFO"].

Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"]. But it is larger in lexical order.

Tips:

题意：给定一些飞机票，让你从中找出一个序列可以全部用完。要求如下：

1.从JFK出发。

2.如果有多个解，输出字典序最小的。

思路：

一直DFS直到arrival的pq取完，然后加最后一个城市，addFirst。

All the airports are vertices and tickets are directed edges. Then all these tickets form a directed graph.

The graph must be Eulerian since we know that a Eulerian path exists.

Thus, start from "JFK", we can apply the Hierholzer's algorithm to find a Eulerian path in the graph which is a valid reconstruction.

Since the problem asks for lexical order smallest solution, we can put the neighbors in a min-heap. In this way, we always visit the smallest possible neighbor first in our trip.

Code：

Recursive：

```
public class Solution {
    public List<String> findItinerary(String[][] tickets) {
        LinkedList<String> result = new LinkedList<>();
        if (tickets == null || tickets.length == 0) {
            return result;
        }
        HashMap<String, PriorityQueue<String>> map = new HashMap
<>();
        for (String[] ticket : tickets) {
            /*if (!map.containsKey(ticket[0])) {
                map.put(ticket[0], new PriorityQueue<String>());
            }*/
            map.putIfAbsent(ticket[0], new PriorityQueue<>());
            map.get(ticket[0]).add(ticket[1]);
        }
        dfs(result, map, "JFK");
        return result;
    }

    private void dfs(LinkedList<String> result, HashMap<String,
PriorityQueue<String>> map, String current) {
        while (map.containsKey(current) && !map.get(current).isE
mpty()) {
            dfs(result, map, map.get(current).poll());

        }
        result.addFirst(current);
    }
}
```

Stack:

```
public List<String> findItinerary(String[][] tickets) {
    Map<String, PriorityQueue<String>> targets = new HashMap<>()
;
    for (String[] ticket : tickets)
        targets.computeIfAbsent(ticket[0], k -> new PriorityQueu
e()).add(ticket[1]);
    List<String> route = new LinkedList();
    Stack<String> stack = new Stack<>();
    stack.push("JFK");
    while (!stack.empty()) {
        while (targets.containsKey(stack.peek()) && !targets.get
(stack.peek()).isEmpty())
            stack.push(targets.get(stack.peek()).poll());
        route.add(0, stack.pop());
    }
    return route;
}
```

Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

For example, Given the following words in dictionary,

```
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
```

The correct order is: "wertf".

Note: You may assume all letters are in lowercase.

If the order is invalid, return an empty string.

There may be multiple valid order of letters, return any one of them is fine.

Tips:

Topology Sort

首先简单介绍一下拓扑排序，这是一个能够找出有向无环图顺序的一个方法

假设我们有3条边：A->C, B->C, C->D，先将每个节点的计数器初始化为0。然后我们对遍历边时，每遇到一个边，把目的节点的计数器都加1。然后，我们再遍历一遍，找出所有计数器值还是0的节点，这些节点就是有向无环图的“根”。然后我们从根开始广度优先搜索。具体来说，搜索到某个节点时，将该节点加入结果中，然后所有被该节点指向的节点的计数器减1，在减1之后，如果某个被指向节点的计数器变成0了，那这个被指向的节点就是该节点下轮搜索的子节点。在实现的角度来看，我们可以用一个队列，这样每次从队列头拿出来一个加入结果中，同时把被这

个节点指向的节点中计数器值减到0的节点也都加入队列尾中。需要注意的是，如果图是有环的，则计数器会产生断层，即某个节点的计数器永远无法清零（有环意味着有的节点被多加了1，然而遍历的时候一次只减一个1，所以导致无法归零），这样该节点也无法加入到结果中。所以我们只要判断这个结果的节点数和实际图中节点数相等，就代表无环，不相等，则代表有环。

对于这题来说，我们首先要初始化所有节点（即字母），一个是该字母指向的字母的集合（被指向的字母在字母表中处于较后的位置），一个是该字母的计数器。然后我们根据字典开始建图，但是字典中并没有显示给出边的情况，如何根据字典建图呢？其实边都暗藏在相邻两个词之间，比如abc和abd，我们比较两个词的每一位，直到第一个不一样的字母c和d，因为abd这个词在后面，所以d在字母表中应该是在c的后面。所以每两个相邻的词都能蕴含一条边的信息。在建图的同时，实际上我们也可以计数了，对于每条边，将较后的字母的计数器加1。计数时需要注意的是，我们不能将同样一条边计数两次，所以要用一个集合来排除已经计数过的边。最后，我们开始拓扑排序，从计数器为0的字母开始广度优先搜索。为了找到这些计数器为0的字母，我们还需要先遍历一遍所有的计数器。

最后，根据结果的字母个数和图中所有字母的个数，判断时候有环即可。无环直接返回结果。

Code:

```
public class Solution {
    public String alienOrder(String[] words) {
        HashMap<Character, HashSet<Character>> graph = new HashM
ap<>();
        HashMap<Character, Integer> indegrees = new HashMap<>();
        Queue<Character> queue = new LinkedList<>();
        StringBuilder orders = new StringBuilder();
        //initialize
        for (String word : words) {
            for (char c : word.toCharArray()) {
                if (!indegrees.containsKey(c)) {
                    indegrees.put(c, 0);
                }
            }
        }
        //count indegrees
        for (int i = 0; i < words.length - 1; i++) {
            String word1 = words[i];
            String word2 = words[i + 1];
            for (int j = 0; j < Math.min(word1.length(), word2.l
ength()); j++) {
                char c1 = word1.charAt(j);
                char c2 = word2.charAt(j);
                if (c1 == c2) {
                    if (j + 1 <= word1.length() - 1 && j + 1 > w
ord2.length() - 1) return "";
                    continue;
                }
                if (!graph.containsKey(c1)) {
                    graph.put(c1, new HashSet<Character>());
                }
                HashSet<Character> set = graph.get(c1);
                if (!set.contains(c2)) {
                    set.add(c2);
                    graph.put(c1, set);
                    indegrees.put(c2, indegrees.get(c2) + 1);
                }
                break;
            }
        }
        // topological sort
        for (char c : indegrees.keySet()) {
            if (indegrees.get(c) == 0) {
                queue.offer(c);
            }
        }
        while (!queue.isEmpty()) {
            char cur = queue.poll();
            orders.append(cur);
            if (graph.containsKey(cur)) {
                for (char c : graph.get(cur)) {
                    if (indegrees.get(c) == 1) {
                        queue.offer(c);
                    }
                    indegrees.put(c, indegrees.get(c) - 1);
                }
            }
        }
        return orders.length() == indegrees.size() ? orders.toSt
ring() : "";
    }
}
```

别人的

```
public class Solution {
    public String alienOrder(String[] words) {
        // 节点构成的图
        Map<Character, Set<Character>> graph = new HashMap<Chara
cter, Set<Character>>();
        // 节点的计数器
        Map<Character, Integer> indegree = new HashMap<Character
, Integer>();
        // 结果存在这个里面
        StringBuilder order = new StringBuilder();
        // 初始化图和计数器
        initialize(words, graph, indegree);
        // 建图并计数
        buildGraphAndGetIndegree(words, graph, indegree);
        // 拓扑排序的最后一步，根据计数器值广度优先搜索
        topologicalSort(order, graph, indegree);
        // 如果大小相等说明无环
        return order.length() == indegree.size() ? order.toStrin
g() : "";
    }

    private void initialize(String[] words, Map<Character, Set<C
haracter>> graph, Map<Character, Integer> indegree){
        for(String word : words){
            for(int i = 0; i < word.length(); i++){
                char curr = word.charAt(i);
                // 对每个单词的每个字母初始化计数器和图节点
                if(graph.get(curr) == null){
                    graph.put(curr, new HashSet<Character>());
                }
                if(indegree.get(curr) == null){
                    indegree.put(curr, 0);
                }
            }
        }
    }

    private void buildGraphAndGetIndegree(String[] words, Map<Ch
aracter, Set<Character>> graph, Map<Character, Integer> indegree
){
        Set<String> edges = new HashSet<String>();
        for(int i = 0; i < words.length - 1; i++){
        // 每两个相邻的词进行比较
            String word1 = words[i];
            String word2 = words[i + 1];
            for(int j = 0; j < word1.length() && j < word2.lengt
h(); j++){
                char from = word1.charAt(j);
                char to = word2.charAt(j);
                // 如果相同则继续，找到两个单词第一个不相同的字母
                if(from == to) continue;
                // 如果这两个字母构成的边还没有使用过，则
                if(!edges.contains(from+""+to)){
                    graph.get(from).add(to);
                    // 将后面的字母加入前面字母的Set中
                    indegree.put(to, indegree.get(to) + 1);
                    // 更新后面字母的计数器，+1
                    // 记录这条边已经处理过了
                    edges.add(from+""+to);
                    break;
                }
            }
        }
    }

    private void topologicalSort(StringBuilder order, Map<Charac
ter, Set<Character>> graph, Map<Character, Integer> indegree){
        // 广度优先搜索的队列
        Queue<Character> queue = new LinkedList<Character>();
        // 将有向图的根，即计数器为0的节点加入队列中
        for(Character key : indegree.keySet()){
            if(indegree.get(key) == 0){
                queue.offer(key);
            }
        }
        // 搜索
        while(!queue.isEmpty()){
            Character curr = queue.poll();
            // 将队头节点加入结果中
            order.append(curr);
            Set<Character> set = graph.get(curr);
            if(set != null){
                // 对所有该节点指向的节点，更新其计数器，-1
                for(Character c : set){
                    if (indegree.get(c) == 1) {
                        queue.offer(c);
                    }
                    indegree.put(c, indegree.get(c) - 1);
                }
            }
        }
    }
}
```

Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following tree

```
    1
   / \
  2   3
     / \
    4   5
```

as "[1,2,3,null,null,4,5]", just the same as how LeetCode OJ serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

Tips:

just use comma and null, BFS, O(n)

Code:

```
public String serialize(TreeNode root) {
    if (root == null) {
        return "null";
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    StringBuilder result = new StringBuilder();
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();
            if (current == null) {
                result.append("null,");
                continue;
            }
            result.append(current.val).append(",");
            queue.offer(current.left);
            queue.offer(current.right);
        }
    }
    System.out.println(result);
    return result.toString();
}

// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    String[] nodes = data.split(",");
    int index = 0;
    if (nodes[index].equals("null")) {
        return null;
    }
    TreeNode root = new TreeNode(Integer.parseInt(nodes[index++]
));
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();
            if (!nodes[index].equals("null")) {
                current.left = new TreeNode(Integer.parseInt(nod
es[index]));
                queue.offer(current.left);
            }
            index++;
            if (!nodes[index].equals("null")) {
                current.right = new TreeNode(Integer.parseInt(no
des[index]));
                queue.offer(current.right);
            }
            index++;
        }
    }
    return root;
}
```

Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

```
nums = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
Return 4
The longest increasing path is [1, 2, 6, 9].
```

Example 2:

```
nums = [
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
Return 4
The longest increasing path is [3, 4, 5, 6]. Moving diagonally i
s not allowed.
```

Tips:

To get max length of increasing sequences:

```
Do DFS from every cell
Compare every 4 direction and skip cells that are out of boundar
y or smaller
Get matrix max from every cell's max
Use matrix[x][y] <= matrix[i][j] so we don't need a visited[m][n
] array
The key is to cache the distance because it's highly possible to
 revisit a cell
```

复杂度：

the DFS here is basically like DP with memorization. Every cell that has been computed will not be computed again, only a value look-up is performed. So this is **O(mn)**, m and n being the width and height of the matrix.

To be exact, each cell can be accessed 5 times at most: 4 times from the top, bottom, left and right and one time from the outermost double for loop. 4 of these 5 visits will be look-ups except for the first one. So the running time should be lowercase o(5mn), which is of course O(mn).

Code:

```
public static final int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1
, 0}};

public int longestIncreasingPath(int[][] matrix) {
    if(matrix.length == 0) return 0;
    int m = matrix.length, n = matrix[0].length;
    int[][] cache = new int[m][n];
    int max = 1;
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            int len = dfs(matrix, i, j, m, n, cache);
            max = Math.max(max, len);
        }
    }
    return max;
}

public int dfs(int[][] matrix, int i, int j, int m, int n, int[]
[] cache) {
    if(cache[i][j] != 0) return cache[i][j];
    int max = 1;
    for(int[] dir: dirs) {
        int x = i + dir[0], y = j + dir[1];
        if(x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] <=
 matrix[i][j]) continue;
        int len = 1 + dfs(matrix, x, y, m, n, cache);
        max = Math.max(max, len);
    }
    cache[i][j] = max;
    return max;
}
```

普通DFS：(会超时)

```
    public class Solution {
        final int[][] dis = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
        public int longestIncreasingPath(int[][] matrix) {
            if (matrix.length == 0) return 0;
            int[][] flag = new int[matrix.length][matrix[0].leng
th];
            int m = matrix.length, n = matrix[0].length;
            int max = 1;
            for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                    max = Math.max(max, helper(matrix, i, j, m,
n, flag, 1));
                    System.out.print(max);
                }
            }
            return max;
        }

        private int helper(int[][] matrix, int i, int j, int m,
int n, int[][] flag, int cur) {
            if (flag[i][j] != 0) return cur;
            flag[i][j] = 1;
            int max = cur;
            for (int[] d : dis) {
                int x = i + d[0], y = j + d[1];
                if (x < 0 || y < 0 || x >= m || y >= n || matrix
[x][y] <= matrix[i][j]) continue;
                int len = helper(matrix, x, y, m, n, flag, cur +
 1);
                System.out.println(len);
                max = Math.max(max, len);

            }
            flag[i][j] = 0;
            return max;
        }
    }
```

Game of Life

According to the Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

Tips:

O(mn)

lives -= board[i][j] & 1; 这句是因为算了九个邻居（包括了自己），所以要减去自己。

To solve it in place, we use 2 bits to store 2 states:

[2nd bit, 1st bit] = [next state, current state]

```
- 00  dead (next) <- dead (current)
- 01  dead (next) <- live (current)
- 10  live (next) <- dead (current)
- 11  live (next) <- live (current)
```

1. In the beginning, every cell is either 00 or 01.
2. Notice that 1st state is independent of 2nd state.
3. Imagine all cells are instantly changing from the 1st to the 2nd state, at the same time.
4. Let's count # of neighbors from 1st state and set 2nd state bit.
5. Since every 2nd state is by default dead, no need to consider transition 01 -> 00.
6. In the end, delete every cell's 1st state by doing >>
7. For each cell's 1st bit, check the 8 pixels around itself, and set the cell's 2nd bit.

```
Transition 01 -> 11: when board == 1 and lives >= 2 && lives <=
3.
Transition 00 -> 10: when board == 0 and lives == 3.
```

To get the current state, simply do

```
board[i][j] & 1
```

To get the next state, simply do

```
board[i][j] >> 1
```

Code:

```
public class Solution {
    public void gameOfLife(int[][] board) {
        if(board == null || board.length == 0) return;
        int m = board.length, n = board[0].length;
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                int lives = liveNeighbors(board, m, n, i, j);

                // In the beginning, every 2nd bit is 0;
                // So we only need to care about when the 2nd bi
t will become 1.
                if(board[i][j] == 1 && lives >= 2 && lives <= 3)
 {
                    board[i][j] = 3; // Make the 2nd bit 1: 01 -
--> 11
                }
                if(board[i][j] == 0 && lives == 3) {
                    board[i][j] = 2; // Make the 2nd bit 1: 00 -
--> 10
                }
            }
        }

        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                board[i][j] >>= 1;  // Get the 2nd state.
            }
        }
    }

    public int liveNeighbors(int[][] board, int m, int n, int i,
 int j) {
        int lives = 0;
        for(int x = Math.max(i - 1, 0); x <= Math.min(i + 1, m -
 1); x++) {
            for(int y = Math.max(j - 1, 0); y <= Math.min(j + 1,
 n - 1); y++) {
                lives += board[x][y] & 1;
            }
        }
        lives -= board[i][j] & 1;
        return lives;
    }
```

```
  }
```

不用位操作：

Pacific Atlantic Water Flow

Given an m x n matrix of non-negative integers representing the height of each unit cell in a continent, the "Pacific ocean" touches the left and top edges of the matrix and the "Atlantic ocean" touches the right and bottom edges.

Water can only flow in four directions (up, down, left, or right) from a cell to another one with height equal or lower.

Find the list of grid coordinates where water can flow to both the Pacific and Atlantic ocean.

Note: 1.The order of returned grid coordinates does not matter. 2.Both m and n are less than 150.

Example:

```
 Given the following 5x5 matrix:

   Pacific ~   ~   ~   ~   ~
        ~  1   2   2   3  (5) *
        ~  3   2   3  (4) (4) *
        ~  2   4  (5)  3   1  *
        ~ (6) (7)  1   4   5  *
        ~ (5)  1   1   2   4  *
           *   *   *   *   * Atlantic

 Return:

 [[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (posit
ions with parentheses in above matrix).
```

Tips:

O(MN)

分别从两个海往里走，如果走得到就在两个海中标记true，最后把两个海都标记了true的加入res。

Code：

dfs：

```
public class Solution {
    int[][]dir = new int[][]{{0,1},{0,-1},{1,0},{-1,0}};

    public List<int[]> pacificAtlantic(int[][] matrix) {
        List<int[]> res = new ArrayList<>();
        if(matrix == null || matrix.length == 0 || matrix[0].len
gth == 0){
            return res;
        }
        int m = matrix.length, n = matrix[0].length;
        boolean[][] p = new boolean[m][n];
        boolean[][] a = new boolean[m][n];
        for (int i = 0; i < m; i++) {
            dfs(p, matrix, i, 0);
            dfs(a, matrix, i, n - 1);
        }
        for (int i = 0; i < n; i++) {
            dfs(p, matrix, 0, i);
            dfs(a, matrix, m - 1, i);
        }
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (a[i][j] && p[i][j]) res.add(new int[] {i, j}
);
            }
        }
        return res;
    }

    private void dfs(boolean[][] visited, int[][] matrix, int i,
 int j) {
        int m = matrix.length, n = matrix[0].length;
        visited[i][j] = true;
        for (int[] d : dir) {
            int x = i + d[0];
            int y = j + d[1];
            if (x < 0 || y < 0 || x >= m || y >= n || visited[x]
[y] || matrix[x][y] < matrix[i][j]) continue;
            dfs(visited, matrix, x, y);
        }
    }
}
```

bfs：

```
public class Solution {
    int[][]dir = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    public List<int[]> pacificAtlantic(int[][] matrix) {
        List<int[]> res = new LinkedList<>();
        if(matrix == null || matrix.length == 0 || matrix[0].len
gth == 0){
            return res;
        }
        int n = matrix.length, m = matrix[0].length;
        //One visited map for each ocean
        boolean[][] pacific = new boolean[n][m];
        boolean[][] atlantic = new boolean[n][m];
        Queue<int[]> pQueue = new LinkedList<>();
        Queue<int[]> aQueue = new LinkedList<>();
        for(int i=0; i<n; i++){ //Vertical border
            pQueue.offer(new int[]{i, 0});
            aQueue.offer(new int[]{i, m-1});
            pacific[i][0] = true;
            atlantic[i][m-1] = true;
        }
        for(int i=0; i<m; i++){ //Horizontal border
            pQueue.offer(new int[]{0, i});
            aQueue.offer(new int[]{n-1, i});
            pacific[0][i] = true;
            atlantic[n-1][i] = true;
        }
        bfs(matrix, pQueue, pacific);
        bfs(matrix, aQueue, atlantic);
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                if(pacific[i][j] && atlantic[i][j])
                    res.add(new int[]{i,j});
            }
        }
        return res;
    }
    public void bfs(int[][]matrix, Queue<int[]> queue, boolean[]
[]visited){
        int n = matrix.length, m = matrix[0].length;
        while(!queue.isEmpty()){
            int[] cur = queue.poll();
            for(int[] d:dir){
                int x = cur[0]+d[0];
                int y = cur[1]+d[1];
                if(x<0 || x>=n || y<0 || y>=m || visited[x][y] |
| matrix[x][y] < matrix[cur[0]][cur[1]]){
                    continue;
                }
                visited[x][y] = true;
                queue.offer(new int[]{x, y});
            }
        }
    }
}
```

Word Squares

Given a set of words (without duplicates), find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where 0 ≤ k < max(numRows, numColumns).

For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.

```
b a l l
a r e a
l e a d
l a d y
```

Note:

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet a-z.

Example 1:

```
Input:
["area","lead","wall","lady","ball"]

Output:
[
  [ "wall",
    "area",
    "lead",
    "lady"
  ],
  [ "ball",
    "area",
    "lead",
    "lady"
  ]
]

Explanation:
The output consists of two word squares. The order of output doe
s not matter (just the order of words in each word square matter
s).
```

Example 2:

```
Input:
["abat","baba","atan","atal"]

Output:
[
  [ "baba",
    "abat",
    "baba",
    "atan"
  ],
  [ "baba",
    "abat",
    "baba",
    "atal"
  ]
]

Explanation:
The output consists of two word squares. The order of output doe
s not matter (just the order of words in each word square matter
s).
```

Tips:

简直难。。。**26^(n*n)**

By considering the word squares as a symmetric matrix, my idea is to go through the top right triangular matrix in left-to-right and then down order. For example, with the case of ["area","lead","wall","lady","ball"] where length = 4, we start with 4 empty string "" "" "" "" Next, [0,0] , "a","b", "l", "w" can be placed, we start with "a" "a" "" "" "" [0,1] go right, "r" can be placed after "a", but no words start with "r" at [1,0], so this DFS ends. "ar" "" "" "" Now, start with "b" at [0,0] "b" "" "" "" We can have "ba" at [0,1] and there is a word start with "a" "ba" "a" "" "" Next "bal" "a" "l" "" Next "ball" "a" "l" "l" When finish the first row, go down to next row and start at [1,1] "ball" "ar" "l" "l" ..... so on and so forth until reaching [4,4]

Details:

if ((rows[row].nodes[i] != null) && (rows[col].nodes[i] != null)) { rows[row] = rows[row].nodes[i]; if (col != row) rows[col] = rows[col].nodes[i]; helper(row, col+1, len, rows, ret); rows[row] = pre_row; if (col != row) rows[col] = pre_col; } Input: ["area","lead","wall","lady","ball"] let's assume that row = 2, col= 2 and we have rows[0] = w a l l rows[1] = a r e a rows[2] = l e ? rows[3] = l a Then, we find a letter from a ~ z which after rows[2] to fill the "?" mark. This is the case of row==col. rows[2] is a Trie node currently representing prefix "le" when i=0, 'a' is the letter after this prefix. rows[2] move to lea To prevent from moving the same pointer twice, we have the condition if (col != row) Therefore, we move to the "right" by giving col+1 for the next DFS step.

Now, we have row = 2, col= 3 and rows[0] = w a l l rows[1] = a r e a rows[2] = l e a ? rows[3] = l a ? Then, we find a letter from a ~ z which after rows[2] and rows[3] to fill the "?" mark. rows[2] is a Trie node currently representing prefix "lea" rows[3] is a Trie node currently representing prefix "la" when i=3, 'd' is the letter which after these two prefixes. rows[2] move to lea'd' rows[3] move to la'd'

The last two lines are the backtracking. rows[row] = pre_row; if (col != row) rows[col] = pre_col;

Code:

```
public class Solution {
    class TrieNode{
        TrieNode[] children;
        String word;
        TrieNode() {
            this.children = new TrieNode[26];
            this.word = null;
        }
    }
    private void add(TrieNode root, String word) {
        TrieNode node = root;
        for (char c : word.toCharArray() ) {
            int idx = c-'a';
            if (node.children[idx] == null) node.children[idx] =
 new TrieNode();
            node = node.children[idx];
        }
        node.word = word;
    }
    public List<List<String>> wordSquares(String[] words) {
        List<List<String>> res = new ArrayList<>();
        if (words == null || words.length == 0) return res;
        TrieNode root = new TrieNode();
        for (String word : words) add(root, word);
        int len = words[0].length();
        TrieNode[] rows = new TrieNode[len];
        for (int i = 0; i < len; i++) rows[i] = root;
        helper(0, 0, len, rows, res);
        return res;
    }

    private void helper(int row, int col, int len, TrieNode[] ro
ws, List<List<String>> res) {
        if (col == row && row == len) {
            List<String> curt = new ArrayList<>();
            for (int i = 0; i < len; i++) {
                curt.add(new String(rows[i].word));
            }
            res.add(curt);
            return;
        }

        if (col < len) {
            TrieNode pre_row = rows[row];
            TrieNode pre_col = rows[col];
            for (int i = 0; i < 26; i++) {
                if (rows[row].children[i] != null && rows[col].c
hildren[i] != null) {
                    rows[row] = rows[row].children[i];
                    if (col != row) rows[col] = rows[col].childr
en[i];
                    helper(row, col + 1, len, rows, res);
                    rows[row] = pre_row;
                    rows[col] = pre_col;
                }
            }
        } else {
            helper(row + 1, row + 1, len, rows, res);
        }
    }
}
```

Minimum Absolute Difference in BST

Given a binary search tree with non-negative values, find the minimum absolute difference between values of any two nodes.

Example:

Input:

```
1
 \
  3
 /
2
```

Output: 1

Explanation: The minimum absolute difference is 1, which is the difference between 2 and 1 (or between 2 and 3). Note: There are at least two nodes in this BST.

Tips: Recursive做法，可以试着重做一遍。**O(n)**

Code:

```
public class Solution {
    int min = Integer.MAX_VALUE;
    Integer prev = null;

    public int getMinimumDifference(TreeNode root) {
        if (root == null) return min;

        getMinimumDifference(root.left);

        if (prev != null) {
            min = Math.min(min, root.val - prev);
        }
        prev = root.val;

        getMinimumDifference(root.right);

        return min;
    }

}
```

Island Perimeter

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

Example:

```
[[0,1,0,0],
 [1,1,1,0],
 [0,1,0,0],
 [1,1,0,0]]

Answer: 16
Explanation: The perimeter is the 16 yellow stripes in the image
 below:
https://leetcode.com/problems/island-perimeter/?tab=Description
```

Tips:

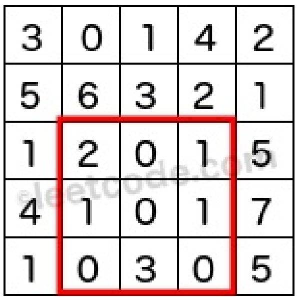
1. loop over the matrix and count the number of islands;
2. if the current dot is an island, count if it has any right neighbour or down neighbour;
3. the result is islands *4 - neighbours* 2

Code:

```
public class Solution {
    public int islandPerimeter(int[][] grid) {
        int islands = 0, neighbours = 0;

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == 1) {
                    islands++; // count islands
                    if (i < grid.length - 1 && grid[i + 1][j] ==
 1) neighbours++; // count down neighbours
                    if (j < grid[i].length - 1 && grid[i][j + 1]
 == 1) neighbours++; // count right neighbours
                }
            }
        }

        return islands * 4 - neighbours * 2;
    }
}
```

Range Sum Query 2D - Mutable

Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).



The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
update(3, 2, 2)
sumRegion(2, 1, 4, 3) -> 10
```

Note:

1. The matrix is only modifiable by the update function.
2. You may assume the number of calls to update and sumRegion function is distributed evenly.
3. You may assume that row1 ≤ row2 and col1 ≤ col2.

Tips:

O(log(m) * log(n))

这道题让我们求二维区域和检索，而且告诉我们数组中的值可能变化，这是之前那道Range Sum Query 2D - Immutable的拓展，由于我们之前做过一维数组的可变和不可变的情况Range Sum Query - Mutable和Range Sum Query - Immutable，那么为了能够通过OJ，我们还是需要用到树状数组Binary Indexed Tree(参见Range Sum Query - Mutable)，其查询和修改的复杂度均为O(logn)，那么我们还是要建立树状数组，我们根据数组中的每一个位置，建立一个二维的树状数组，然后还需要一个getSum函数，以便求得从(0, 0)到(i, j)的区间的数字和，然后在求某一个区间和时，就利用其四个顶点的区间和关系可以快速求出。

Code:

```
public class NumMatrix {

    int[][] tree;
    int[][] nums;
    int m;
    int n;

    public NumMatrix(int[][] matrix) {
        if (matrix.length == 0 || matrix[0].length == 0) return;
        m = matrix.length;
        n = matrix[0].length;
        tree = new int[m+1][n+1];
        nums = new int[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                update(i, j, matrix[i][j]);
            }
        }
    }

    public void update(int row, int col, int val) {
        if (m == 0 || n == 0) return;
        int delta = val - nums[row][col];
        nums[row][col] = val;
        for (int i = row + 1; i <= m; i += i & (-i)) {
            for (int j = col + 1; j <= n; j += j & (-j)) {
                tree[i][j] += delta;
            }
        }
    }

    public int sumRegion(int row1, int col1, int row2, int col2)
 {
        if (m == 0 || n == 0) return 0;
        return sum(row2+1, col2+1) + sum(row1, col1) - sum(row1,
 col2+1) - sum(row2+1, col1);
    }

    public int sum(int row, int col) {
        int sum = 0;
        for (int i = row; i > 0; i -= i & (-i)) {
            for (int j = col; j > 0; j -= j & (-j)) {
                sum += tree[i][j];
            }
        }
        return sum;
    }
}
```

Count of Smaller Numbers After Self

You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to the right of nums[i].

Example:

```
Given nums = [5, 2, 6, 1]

To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
```

Return the array [2, 1, 1, 0].

Tips:

1. we should build an array with the length equals to the max element of the nums array as BIT.
2. To avoid minus value in the array, we should first add the (min+1) for every elements (It may be out of range, where we can use long to build another array. But no such case in the test cases so far.)
3. Using standard BIT operation to solve it.

用max.length建立的数组，每个数都减去了最小值+1，所以这个tree数组的index的是从1到max的所有数。然后从最后一位开始更新，找到这个数在tree里面的位置，取他的parent就是比他大一个的数的在tree里的地方算prefix sum

https://en.wikipedia.org/wiki/Fenwick_tree

https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

Code:

```
public class Solution {
    public List<Integer> countSmaller(int[] nums) {
        List<Integer> res = new ArrayList<>();
        if (nums == null || nums.length == 0) return res;
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;
        for (int num : nums) {
            min = Math.min(min, num);
        }
        for (int i = 0; i < nums.length; i++) {
            nums[i] = nums[i] - min + 1;
            max = Math.max(max, nums[i]);
        }
        int[] tree = new int[max + 1];
        for (int i = nums.length - 1; i >= 0; i--) {
            res.add(0, getSum(nums[i] - 1, tree));
            update(nums[i], tree);
        }
        return res;
    }
    private int getSum (int i, int[] tree) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i  -= i & (-i);
        }
        return sum;
    }
    private void update (int i, int[] tree) {
        while (i < tree.length) {
            tree[i]++;
            i += i & (-i);
        }
    }
}
```

Reverse Pairs

Given an array nums, we call (i, j) an important reverse pair if i < j and nums[i] > 2*nums[j].

You need to return the number of important reverse pairs in the given array.

Example1:

```
Input: [1,3,2,3,1]
Output: 2
```

Example2:

```
Input: [2,4,3,5,1]
Output: 3
```

Note:

1. The length of the given array will not exceed 50,000.
2. All the numbers in the input array are in the range of 32-bit integer.

Tips:

BIT: 和count of smaller number after itself 类似， 就是这次因为要大于两倍，所以取BIT中的index用二分搜索搜一下。这次从原数列的头开始，找到比当前数的两倍还要大的第一个数，取出他之后的所有数（因为都满足要求）并更新数组，更新数组时是更新比当前数小的所有数。

Merge Sort: 分两组，左右各自有序，如果右边的小于左边/2，就加上右边到最后所有的数。

In each round, we divide our array into two parts and sort them. So after "int cnt = mergeSort(nums, s, mid) + mergeSort(nums, mid+1, e); ", the left part and the right part are sorted and now our only job is to count how many pairs of number (leftPart[i], rightPart[j]) satisfies leftPart[i] <= 2*rightPart[j].

For example, left: 4 6 8 right: 1 2 3 so we use two pointers to travel left and right parts. For each leftPart[i], if j<=e && nums[i]/2.0 > nums[j], we just continue to move j to the end, to increase rightPart[j], until it is valid. Like in our example, left's 4 can match 1 and 2; left's 6 can match 1, 2, 3, and left's 8 can match 1, 2, 3. So in this particular round, there are 8 pairs found, so we increases our total by 8.

Code:

BIT:

}

```
public class Solution {
    public int reversePairs(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        int[] copy = Arrays.copyOf(nums, nums.length);
        int[] tree = new int[nums.length + 1];
        Arrays.sort(copy);
        int res = 0;
        for (int num : nums) {
            res += get(index(copy, 2L * num + 1), tree);
            update(index(copy, num), tree);
        }
        return res;
    }
    private int get(int i, int[] tree) {
        int res = 0;
        while (i < tree.length) {
            res += tree[i];
            i += i & (-i);
        }
        return res;
    }
    private void update(int i, int[] tree) {
        while (i > 0) {
            tree[i]++;
            i -= i & (-i);
        }
    }
    private int index(int[] copy, long val) {
        int l = 0, r = copy.length - 1, m = 0;
        while (l <= r) {
            m = l + (r - l) / 2;
            if (copy[m] >= val) r = m - 1;
            else l = m + 1;
        }
        return l + 1;
    }
}
```

Merge Sort:

```
public class Solution {
    public int reversePairs(int[] nums) {
        return mergeSort(nums, 0, nums.length-1);
    }
    private int mergeSort(int[] nums, int s, int e){
        if(s>=e) return 0;
        int mid = s + (e-s)/2;
        int cnt = mergeSort(nums, s, mid) + mergeSort(nums, mid+
1, e);
        for(int i = s, j = mid+1; i<=mid; i++){
            while(j<=e && nums[i]/2.0 > nums[j]) j++;
            cnt += j-(mid+1);
        }
        Arrays.sort(nums, s, e+1);
        return cnt;
    }
}
```

九章：

```
public class Solution {
    /**
     * @param A an array
     * @return total of reverse pairs
     */
    public long reversePairs(int[] A) {
        return mergeSort(A, 0, A.length - 1);
    }

    private int mergeSort(int[] A, int start, int end) {
        if (start >= end) {
            return 0;
        }

        int mid = (start + end) / 2;
        int sum = 0;
        sum += mergeSort(A, start, mid);
        sum += mergeSort(A, mid+1, end);
        sum += merge(A, start, mid, end);
        return sum;
    }

    private int merge(int[] A, int start, int mid, int end) {
        int[] temp = new int[A.length];
        int leftIndex = start;
        int rightIndex = mid + 1;
        int index = start;
        int sum = 0;

        while (leftIndex <= mid && rightIndex <= end) {
            if (A[leftIndex] <= A[rightIndex]) {
                temp[index++] = A[leftIndex++];
            } else {
                temp[index++] = A[rightIndex++];
                sum += mid - leftIndex + 1;
            }
        }
        while (leftIndex <= mid) {
            temp[index++] = A[leftIndex++];
        }
        while (rightIndex <= end) {
            temp[index++] = A[rightIndex++];
        }

        for (int i = start; i <= end; i++) {
            A[i] = temp[i];
        }

        return sum;
    }
}
```

Valid Word Square

Given a sequence of words, check whether it forms a valid word square.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where 0 ≤ k < max(numRows, numColumns).

Note:

1. The number of words given is at least 1 and does not exceed 500.
2. Word length will be at least 1 and does not exceed 500.
3. Each word contains only lowercase English alphabet a-z.

Example 1:

```
Input:
[
  "abcd",
  "bnrt",
  "crmy",
  "dtye"
]

Output:
true

Explanation:
The first row and first column both read "abcd".
The second row and second column both read "bnrt".
The third row and third column both read "crmy".
The fourth row and fourth column both read "dtye".

Therefore, it is a valid word square.
```

Example 2:

```
Input:
[
  "abcd",
  "bnrt",
  "crm",
  "dt"
]

Output:
true

Explanation:
The first row and first column both read "abcd".
The second row and second column both read "bnrt".
The third row and third column both read "crm".
The fourth row and fourth column both read "dt".

Therefore, it is a valid word square.
```

Example 3:

Input: ["ball", "area", "read", "lady"]

Output: false

Explanation: The third row reads "read" while the third column reads "lead".

Therefore, it is NOT a valid word square.

Tips:

一个一个比，注意先判断有没有出界。

Code：

```
public class Solution {
    public boolean validWordSquare(List<String> words) {
       if(words == null || words.size() == 0){
            return true;
        }
        int n = words.size();
        for(int i=0; i < n; i++){
            for(int j=0; j < words.get(i).length(); j++){
                if(j >= n || words.get(j).length() <= i || words
.get(j).charAt(i) != words.get(i).charAt(j))
                    return false;
            }
        }
        return true;
    }
}
```

Maximum Product of Word Lengths

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.
```

Tips:

其实，因为每个字母都是小写字母，所以我们可以用一个int[] masks的26位去保存每个单词所包含的字母的信息：因为我们不关心每个单词中各个字母出现的个数，只关心是否出现，所以可以用1表示出现，0表示未出现。这样的好处是，经过这样的预处理之后，当需要判断两个单词是否有相同字母时，做个与计算就知道结果了（是否为0）。这样，我们就得到了O(n*n)的解法。

在优化过的解法中，还按照word的长度给这些word排序，这样一旦找到最大值可以立刻停止。很快。而总复杂度是O(n*n)，所以可以sort。第一个版本为优化过，第二个为普通，第三个为没有bit manipulate的版本

没有bit的tip：the core of this problem is to figure out a data structure that can store what letters a word has and make the comparison of whether two words has same letter easier. We can either store the info in an 26 array or with a 32-bit integer.

Code: