Leetcode 自推

E2 之後, 只要能想到方法, 基本上就能寫出來, 所以重點是想到方法, 細節不重要.

DS^N 拿到 Amazon 實習(2016 年 2 月底), 更說明了刷題的重要性, 和 OMSCS 的可行性.
MY 3 月 1 日拿到 Amazon offer, 說明不要著急, 也說明我沒有理由找不到工作.
五遍哥最終拿到 offer(也是 3 月初拿到 offer, 共面了六家), 說明 CS 行業付出總會有回報, 也說明形勢沒他們說的那麼不好.
Yicun 被 Amazon 拒了, 但還是拿到了 data science 的 offer.

mitbbs 上的消極情緒實在有點誤導人, 弄得我整個二月份沒怎麼投, 把時間都花到學知識上去了, 錯過了投公司的黃金時間, 直到一大堆人拿到 offer, 我才醒悟過來. 以後不要管那些人的消極帖子了, 好多是故意發的. 反正我身邊的人都找到工作了.

想想我之前的四一五事件, 西樹事件, 英報事件, 再 lēng 都沒事, 一定要沉得住氣. 我之後一定比計劃更好.

若面一家, 拿 offer 的概率為 10%, 則面試 22 家後, 有 offer 的概率為 1 - 0.9^22 = 90.2%.
若投簡歷拿面試的概率為 10%, 則要拿到 22 家面試, 就要投 220 家簡歷.
刷題的意義在於使 面一家拿 offer 的概率大於 10%.
讀 Gatech CS 的意義在於使 投簡歷拿面試的概率大於 10%.

一切都是熟能生巧, 想一想我考駕照, 最開始覺得很難, 很多人都考了兩三次才過, 結果我一次考過且得了 93 分. 我現在開車這麼熟練, 是學車時不敢相信的. 另一個例子是我考 pku, 全國第一, 當時電動力學自推, 最開始都忘了後面講的甚麼, 都不敢看, 結果 E4 後, 倒背如流. 還比如**我考 GRE** 等. 所以一切都是熟能生巧, 我最終也能成為**刷題王**. 從考 pku, 到考 GRE, 到發文章, 到申請出國, 事實證明, 只要我想做的事, 還沒有沒做到的.

E1 的要求: E1 沒有 key 可以看, 所以是自己想方法. 但寫的時候可以在我電腦上用 main 函數和我自己的 test case 來試 . 之後(E2, E3...)的要求是: 寫的時候不能在我電腦上用 main 函數來試, 必須直接寫好拿到 OJ 上去試, 但是可以用我電腦上的編譯器來測試是否有語法錯誤, 若第一次沒通過, 後面改的時候可以在我電腦上用 main 函數來試.
E2 的要求: 可以看 key.
E3 的要求: 不能看 key 或任何提示, 必須自己想(或回憶)方法. 並要求每題都在代碼中寫出時間和空間複雜度並核對. 若沒做出來, 默寫答案時不能用在我電腦上編繹查語法錯, 只能全寫好了再查.

Paste 最好的方法:
Ctrl+Alt+Shift+V to paste unformatted text, 或 Ctrl+Shift+V to choose, 或直接 Ctrl+V 保持原來格式, 反正後面也要自己選格式. 現在基本上都直接 Ctrl+V.
paste 了之後直接在左上角選 Default Style, 就可以 paste 成我想要的格式, 有時要自已選 Liberation Serif, 12, 黑色.

本 note 按題號排, 而不按分類排的原因: 分類經常會調整, 所以將本 note 中的解答也調整順序會比較麻煩, 就在我的那個題目分類的文件中調整順序會簡單些.

遞歸 和 動態規劃 的本質都是 遞推

遞歸就是數學歸納法: 已假定好 helper(i)俱有某個功能, 用 helper(i)來實現 helper(i+1), 而且要處理好初始情況, 比如 i=0 的時候. 很多循環也可以這樣想.

我感覺絶大多數遞歸都是 DFS.

我的總節: DFS 就是遞歸, BFS 就是用 queue.

(Code Ganker 在 133 和 130 題中都說了 DFS 可以不用遞歸)

每道題的 converntion 還是要記一下, 因為若是 OA, 則沒有面試官可以問.

動態規劃中，要用 res[i]來表示 's 的前 i 個字符' 能否怎麼樣；而不要用 res[i]來表示 s[0,...i]能否怎麼樣，因為後者沒法表示 's 的前 0 個字符' 能否怎麼樣: 139. Word Break, Medium.

LinkedList<Integer> stack1 = stack2 中, 左邊的 stack1 只是一個指針, 右邊的 stack2 是一個俱體的 object. 這句話的作用就是將 stack1 指向 「stack2 所指的那個 object」.

List 題目中, 賦值式右邊都是 object, 左邊的變量都是指針. 如 p1.next = p2.next 中, 右邊的 p2.next 是一個俱體的 ListNode 的 object, 左邊的 p1.next 只是一個指針(注意 ListNode 類中有 next 這麼個成員). 再如 cur = p2.next 中左邊的 cur 也是一個指針.

List 題目中, 寫循環的終止條件時, 直接看循環結束時是什麼樣的(如 cur=null), 然後就將終止條件寫成不這麼樣, 即 while(cur != null).

實參時用 ArrayList<String>，形參時用 List<String>: 131. Palindrome Partitioning, Medium

new 後不能跟 List, 而要跟 ArrayList(包括 res.add(new ArrayList<Integer>(item))): 113. Path Sum II, Medium

二分法的 九章算法模版: 35. Search Insert Position

item-res 遞歸法模版: 46. Permutations 和 77. Combinations

85. Maximal Rectangle:
當二維數組 matrix 作為輸入時, 不要按以下這樣寫:

```
if(matrix == null) return 0;
int n = matrix.length, m = matrix[0].length;
if(n == 0 || m == 0) return 0;
```

這樣寫的錯誤之處在於, 若輸入數組為[], 即 matrix.length =0, 此時 matrix[0].length 根本就不存在, 故 m = matrix[0].length 一句會報錯. 所以以後二維數組都不要像上面那樣寫, 而應該像下面這樣寫:

```
if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
int n = matrix.length, m = matrix[0].length;
```

66. Plus One:
E2 幾分鍾寫好, 但改了幾次才通過, 因為 E2 最開始是這樣寫的:

```
digits[i] = (digits[i] + carry) % 10;
carry = (digits[i] + carry) / 10;
```

導致第二行的 digits[i]已經被新值覆蓋了, 若交換這兩句的順序, carry 也會被覆蓋, 後來改為以下的才通過, 以後都按以下那樣寫:

```
sum = (digits[i] + carry);
carry = sum / 10;
digits[i] = sum % 10;
```

為何遞歸的空間複雜度為 log n: 23. Merge k Sorted Lists
對遞歸回溯的最好解釋, 可當公式用: 113. Path Sum II, Medium
遞歸回溯之最好理解: 78. Subsets, Medium
BFS 典型題, 可當公式用: 102. Binary Tree Level Order Traversal, Easy 還可看看 104 題中的 level
Inorder 等 traversal 的典型題, 可當公式用: 94. Binary Tree Inorder Traversal, Medium
動態歸劃基本思路總結: 139. Word Break, Medium
List<List<String>> res = new ArrayList<List<String>>(): 131. Palindrome Partitioning, Medium
動態歸劃中將二維數組 res[i][j]降低為一維數組的方法: 97. Interleaving String, Hard
按引用傳遞: 109. Convert Sorted List to Binary Search Tree, Medium
DFS 和 BFS 比較並體會: 133. Clone Graph, Medium
iteration(迭代法)之思想: 156. Binary Tree Upside Down, Medium
if 和{在同一行的格式只有好處沒有壞處：159. Longest Substring with At Most Two Distinct Characters, Hard
左右為不同類型(Queue 和 LinkdedList)的時候, 右邊仍可省略<>中的內容. 199. Binary Tree Right Side View, Medium
若要用 list.peek(), 則不能寫成 List<Integer> list = new LinkedList<Integer>(), 而要寫成 LinkedList<Integer> list = new LinkedList<Integer>(). 227. Basic Calculator II, Medium 這四句實際上形成了一個環, 每句右邊的變量就是下一句左邊的, 所以他們的順序一下就記住了, 其它情況(如互換兩個數)也可以類似記憶). 86. Partition List, Medium
Code Ganker:
二分查找树在面试中非常常见: 98. Validate Binary Search Tree, Medium
回文串的判断和搜索是面试中经常遇到的基本问题，我在 Facebook 的面试中就遇到了这道题: 125. Valid Palindrome, Easy
一個 BST 最小值的那個節點為: 沿著該 BST 最左邊緣一直按 cur = cur.left 走到底的那個節點. 173. Binary Search Tree Iterator, Medium
給定 BST 中的一個節點, 要求比它大的下一個節點(即中序遍歷的下一個)不是那麼直接的, 要像本題那樣弄個 stack. 173. Binary Search Tree Iterator, Medium

HashTable 和 HashMap:

http://blog.sina.com.cn/s/blog_93dc666c0101hbib.html

http://blog.csdn.net/zhangerqing/article/details/8193118

Font colors: Red 3, Yellow 4

1. Two Sum. Medium.

http://blog.csdn.net/linhuanmars/article/details/19711387

Given an array of integers, find two numbers such that they add up to a specific target number.
The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.
You may assume that each input would have exactly one solution.
Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2
Subscribeto see which companies asked this question

Key: 用 HashMap. Key 為 nums[i], value 為 i. 用 map.containsKey(target-numbers[i])). 注意題目已經更新為 The return format had been changed to zero-based indices, 將答案中的+1 去掉後, 即可通過.

N-sum 係列題目方法快速記憶:
1. 非 sorted 2sum: 用 map(nums[i], i)
2. sorted 2sum: 用 l--, r++
3. 3sum: 寫一個 2Sum(nums, end, target), 3sum 中調用 2Sum(nums, i - 1, target - nums[i]), 即另兩個數只在 nums[i]左邊找.

History:
E1 沒做出來.
E2 通過.
E3 改了幾次後才通過. 原因是不知道 numbers 中元素可能有重復的, 且對 numbers 中只有一個 target/2 元素時處理得不好, 後來專門處理了這兩種情況後通過. 注意以下答案的代碼很巧, 將 map.put() 放在 map.containsKey 同一個循環中, 且在 map.containsKey 後面, 這樣即自動處理好了上面兩種情況, 又讓代碼簡短.

我: 要掌握的是以下第一種方法. 第二種方法其實就是 167. Two Sum II 的解答.
这是一道非常经典的题目，brute force 时间复杂度为 O(n^2), 对每一对 pair 两两比较。 优化的方法一般有两种，第一种是用哈希表，时间复杂度为 O(n),同时空间复杂度也是 O(n),代码如下：

```
public int[] twoSum(int[] numbers, int target) {
    int[] res = new int[2];
    if(numbers==null || numbers.length<2)
        return null;
    HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();
    for(int i=0;i<numbers.length;i++)
    {
        if(map.containsKey(target-numbers[i]))
        {
            res[0]=map.get(target-numbers[i])+1;
            res[1]=i+1;
            return res;
```

```
        }
        map.put(numbers[i],i);
    }
    return null;
}
```

第二种解法是先对数组进行排序，然后使用夹逼的方法找出满足条件的 pair，原理是因为数组是有序的，那么假设当前结果比 target 大，那么左端序号右移只会使两个数的和更大，反之亦然。所以每次只会有一个选择，从而实现线性就可以求出结果。该算法的时间复杂度是 O(nlogn+n)=O(nlogn)，空间复杂度取决于排序算法。代码如下：

```
public int[] twoSum(int[] numbers, int target) {
    int[] res = new int[2];
    if(numbers==null || numbers.length<2)
        return null;
    Arrays.sort(numbers);
    int l = 0;
    int r = numbers.length-1;
    while(l<r)
    {
        if(numbers[l]+numbers[r]==target)
        {
            res[0] = number[l];
            res[1] = number[r];
            return res;
        }
        else if(numbers[l]+numbers[r]>target)
        {
            r--; //為何不 l--? 因為兩個 pointer 是從兩端向中間移動的. l--就退了回去, where has already been visited earlier.
        }
        else
        {
            l++;
        }
    }
    return null;
```

}

注意，在这里，输出结果改成了满足相加等于 target 的两个数，而不是他们的 index。因为要排序，如果要输出 index，需要对原来的数的 index 进行记录，方法是构造一个数据结构，包含数字的值和 index，然后排序。所以从这个角度来看，这道题第二种解法并没有第一种解法好，空间也是 O(n). 在 LeetCode 原题中是假设结果有且仅有一个的，一般来说面试时会要求出所有的结果，这个时候会涉及到重复 pair 的处理，相关的内容会在 3Sum 那道题目中涉及到，请参见 3Sum -- LeetCode.

2. Add Two Numbers, Medium

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

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8

Key: 題目的意思是輸入的 list 的頭表示數字的最低位, 即 2->4->3 表示 342(三百四十二). 結果也是按這個順序寫的, 即結果的 7->0->8 也表示 807(八百零七). 方法還是常規的加法算法寫:

```
sum = l1.val + l2.val + carry;
digit = sum % 10;
carry = sum / 10;
```

從左向右(即低位到高位)算. 跟 67 題一樣, 要注意輸入的一個數比另一個長時, 以及結果中最高位還要进一位時之情況. 用一個 falseHead, 可使處理頭部時比較方便.

由題目給的 ListNode class 的代碼可以看出, 一個 class 的 constructor 不一定要初始代所有 memeber, 比如 ListNode class 的 constructor 就只初始化了 val, 而沒初始化 next (已 record 至 Java p348).

History:
E1 把題目意思理解錯了, 以為要求 加數 和 結果 的順序是相反的，這用單向 linked list 基本上是不可能實現的，害得我想了一下午. 看了答案後才知道題目是甚麼意思，然後自己做出來了，但也借用了答案中處理进位的方法, 不過這個方法也不難想到. 由於看了答案, 故算沒做出來.

E2 幾分鍾寫好, 一次通過, 且 E1 沒留下任何 key, 也沒帖出任何答案, E2 是甚麼都沒看, 直接寫的. 以上的 key 是 E2 寫的. E2 的代碼跟 Code Ganker 的大體一樣, 即都是分 while(l1 != null && l2 != null), while(l1 != null), while(l2 != null), if(carry > 0)幾鍾情況, 但 E2 由於用了個 falseHead, 反而處理頭部比 Code Ganker 的簡潔.

E3 較快寫好, 本來可以一遍通過的, 結果不小心筆誤將 l1 寫成了 l2, l2 寫成了 l1, 改後即通過. E3 的代碼最簡短, 以後用 E3 的代碼.

E3 的代碼(用它):

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    if(l1 == null) return l2;
    if(l2 == null) return l1;
    ListNode fakeHead = new ListNode(0);
    ListNode cur = fakeHead;
```

```
        int carry = 0;

        while(l1 != null && l2 != null) {
            int sum = l1.val + l2.val + carry;
            int digit = sum % 10;
            carry = sum / 10;
            cur.next = new ListNode(digit);
            cur = cur.next;
            l1 = l1.next;
            l2 = l2.next;
        }

        while(l1 != null) {
            int sum = l1.val + carry;
            int digit = sum % 10;
            carry = sum / 10;
            cur.next = new ListNode(digit);
            cur = cur.next;
            l1 = l1.next;
        }

        while(l2 != null) {
            int sum = l2.val + carry;
            int digit = sum % 10;
            carry = sum / 10;
            cur.next = new ListNode(digit);
            cur = cur.next;
            l2 = l2.next;
        }

        if(carry > 0) cur.next = new ListNode(carry);

        return fakeHead.next;
    }
```

3. Longest Substring Without Repeating Characters, Medium

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.
Subscribe to see which companies asked this question

Key: 用 start 來表示當前的 '無重復字母的子串' 的左端, i 為其右端, i 往右走, 例如以下的 bpdefghojk 即為這樣一個 '無重復字母的子串' :
a a b(start) p d e f g h o j k(i) e l m n

用字符 c 來表示 s[i]. 用數組 charMap[c] 記錄上一個 c 在 s 中的位置 (c 為這個 char 的 ASCII 碼(共 256 個), 直接將 char 類型的 c 用作數組 index 即可, 可自動轉換為 int).

若 charMap[c] >= start, 例如:
a a b(start) p d e(charMap[c]) f g h o j k e(i) l m n

則說明該 '無重復字母的子串' bpdefghojke 中已有兩個 e 字符, 故它不再是 '無重復字母的子串', 要將其左邊的一段砍掉, 才能重新變為 '無重復字母的子串'. 俱體地砍哪一段? 答: 砍 bpde 這段, 即[start, charMap[c]]這段. 剩下的 fghojke 這段照樣是 '無重復字母的子串', 即[charMap[c]+1, ...i]這段. 為何不砍其它地方? 因為若砍 bp 這段, 則剩下的 defghojke 還是有重復字符啊.

實際寫的時候, 砍 即為 將 start 移到 c 的下一個(即 charMap[c]+1), 此時的[start, i]就是一個新的無重復的子串, 最後在所有的這樣的子串中找最長的即可.

History:
E1 沒做出來.
E2 很快寫好, 本來可以一次通過, 但粗心有個地方該 array index 越界了, 改後即通過, 但 E2 用的我自己的 O(N^2)方法(E1 基本上沒留下甚麼 key, E2 方法是自己想的, 本題的答案是 E2 弄的), 方法就是考查[l, r]子串, l 從 string 的最左到最右, 對每一個 l, 都找到最長不重復子串對應的 r(會用到 Set), 然後在所有的[l, r]子串中找到最長的, 由於這不是最優解, 以後不用, 以後都用 William 的代碼.
E3 改了幾次後通過, 但還是用的 E2 的 O(N^2)方法. 所以 E3 還是將 William 的第二種方法(O(N))默寫了一遍. E3 還重寫了 key.

我: William 給了兩種方法, 其實方法都差不多, 要當握的是第二種方法, 因為時間復雜度最小.

William:
1. Two pointer + Hash Table: Time ~ O(2N), Space ~ O(1)
类似上题解法 2。
start - the start index;
i - the curr (end) index.
Move j from 0 to N - 1, update the maxLen = max{i - start + 1, maxLen} at every step.
If we find a visited character (exist[c] == true), then shrink start pointer i until find the same character c, and then update the maxLen.

```
public int lengthOfLongestSubstring(String s) {
    boolean[] exist = new boolean[256]; // ASCII
    int start = 0, maxLen = 0;
    for (int i = 0; i < s.length(); i++) {
        while (exist[s.charAt(i)])
            exist[s.charAt(start++)] = false;
        exist[s.charAt(i)] = true;
        maxLen = Math.max(i - start + 1, maxLen);
    }
    return maxLen;
}
```

2. Hash Table: Time ~ O(N), Space ~ O(1)
Hash Table charMap[c] 记录上一个 c 出现的 index。
如果 charMap[c] >= start，表明从 start 到 i 已出现过 c，故 c 为重复字符，取 substring s[start .. i] 的长度 start - i + 1，并更新 maxLen。

```
public int lengthOfLongestSubstring(String s) {
    int[] charMap = new int[256]; // ASCII
```

```
    Arrays.fill(charMap, -1);
    int start = 0, maxLen = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
         if (charMap[c] >= start)    start = charMap[c] + 1; // exist a repeat of
s.charAt(j) from i to j
        charMap[c] = i;
        maxLen = Math.max(maxLen, i - start + 1);
    }
    return maxLen;
}
```

4. Median of Two Sorted Arrays. Hard

http://blog.csdn.net/linhuanmars/article/details/19905515

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

Key: 本題方法要記住. 注意題目要求時間為 O(log(m+n)), 所以用每次丟一半之方法. 寫一個 helper(A, B, i, i2, j, j2, k), 其作用為返回 A[i, ...i2]和 B[j,...j2]總共的第 k 個數(即第 k 小的數, 從 k=1 開始). helper 遞歸調用自己. helper 的實現方法為: 若 A[k/2] == B[k/2], 則說明它們就是 'A 和 B 的第 k 個數' (此時別證, 先看完後面); 若 A[k/2]>B[k/2], 則說明 B 的前 k/2 個數都不是 'A 和 B 的第 k 個數', 故 B 將它們都丟掉(A 也只留前 k/2 個); 同理若 A[k/2]<B[k/2], 則將 A 的前 k/2 個都丟掉(B 也只留前 k/2 個)．注意前面 A[k/2]等中寫的 k/2，實際上准確的寫法遠比它複雜，如何寫出？要背住以下兩式：
posA = Math.min(k/2,m); //m 為 A[i, ...i2]中元素個數
posB = k-posA; //記住了上式，此式就好記了
然後 A[k/2]等中寫 k/2 可用它們表示出．另外，helper 中是假定 A 比 B 短的．另外 helper 中還要考慮 m=0 和 k=1 兩個特殊情況．

Median 的定義為(參考了 wiki): 若總共有寄數個數, 則取中間那個數; 若總共有偶數個數, 則取中間的兩個數的平均值. 所以本題返回的是一個 double.

本題用 Code Ganker 的答案, 是因為他的算法比 William 的(和一個網上的)更好懂，代碼也更短．

History:
E1 沒做出來.

E2 看了 posA = Math.min(k/2,m)和 posB = k-posA 兩式後, 代碼對都對了, except for m==0 和 k==1 兩個特殊情況沒寫好. 後來難得查錯, 就看了答案. 算我沒做出來. E1 沒寫 key, 也沒弄答案, 所有都是 E2 弄的. E3 沒做出來. E3 將本題方法忘得一幹二淨了, 故直接看的答案.

这道题比较直接的想法就是用 Merge Sorted Array 这个题的方法把两个有序数组合并，当合并到第(m+n)/2 个元素的时候返回那个数即可，而且不用把结果数组存起来。算法时间复杂度是 O(m+n)，空间复杂度是 O(1)。因为代码比较简单，就不写出来了，跟 Merge Sorted Array 比较类似，大家可以参照这个题目的解法。

(我: 以下 Code Ganker 說的第 k 大, 實際上應該為第 k 小)

接下来我们考虑有没有优化的算法。优化的思想来源于 order statistics，在算法导论 10.3 节中提到。问题等价于求两个 array 的第 k=(m+n)/2（假设 m 和 n 分别是两个数组的元素个数）大的数是多少。基本思路是每次通过查看两个数组的第 k/2 大的数(假设是 A[k/2],B[k/2])，如果两个 A[k/2]=B[k/2]，说明当前这个数即为两个数组剩余元素的第 k 大的数，如果 A[k/2]>B[k/2], 那么说明 B 的前 k/2 个元素都不是我们要的第 k 大的数，反之则排除 A 的前 k/2 个，如此每次可以排除 k/2 个元素，最终 k=1 时即为结果。总的时间复杂度为 O(logk),空间复杂度也是 O(logk)，即为递归栈大小。在这个题目中因为 k=(m+n)/2,所以复杂度是 O(log(m+n))。比起第一种解法有明显的提高，代码如下：

```
public double findMedianSortedArrays(int A[], int B[]) {
    if((A.length+B.length)%2==1) //我寫的時候, 可以定義兩個字母來代表 A 和 B 的長度
        return helper(A,B,0,A.length-1,0,B.length-1,(A.length+B.length)/2+1);
    else
        return (helper(A,B,0,A.length-1,0,B.length-1,(A.length+B.length)/2)
              +helper(A,B,0,A.length-1,0,B.length-1,(A.length+B.length)/2+1))/2.0;
}

private int helper(int A[], int B[], int i, int i2, int j, int j2, int k)
{
    int m = i2-i+1;
    int n = j2-j+1;
    if(m>n)
        return helper(B,A,j,j2,i,i2,k);
    if(m==0)
        return B[j+k-1];
    if(k==1)
        return Math.min(A[i],B[j]);
    int posA = Math.min(k/2,m);
    int posB = k-posA;
    if(A[i+posA-1]==B[j+posB-1])
        return A[i+posA-1];
    else if(A[i+posA-1]<B[j+posB-1])
        return helper(A,B,i+posA,i2,j,j+posB-1,k-posA);
    else
        return helper(A,B,i,i+posA-1,j+posB,j2,k-posB);
}
```

实现中还是有些细节要注意的，比如有时候剩下的数不足 k/2 个，那么就得剩下的，而另一个数组则需要多取一些数。但是由于这种情况发生的时候，不是把一个数组全部读完，就是可以切除 k/2 个数，所以不会影响算法的复杂度。

这道题的优化算法主要是由 order statistics 派生而来，原型应该是求 topK 的算法，这个问题是非常经典的

问题，一般有两种解法，一种是用 quick select(快速排序的 subroutine),另一种是用 heap。 复杂度是差不多的，有兴趣可以搜一下，网上资料很多，topK 问题在海量数据处理中也是一个非常经典的问题，所以还是要重视。

5. Longest Palindromic Substring. Medium.

Given a string *S*, find the longest palindromic substring in *S*. You may assume that the maximum length of *S* is 1000, and there exists one unique longest palindromic substring.
Subscribe to see which companies asked this question

Key: 從中心往兩邊掃描. 分兩種情況: 中心為一個字符(如 xycabacwz 中的 b) 以及 中心為兩字符的間隙(如 xycaacwz 中的 aa 之間的間隙). 中心本身也要取遍整個 s 的所有字符(或間隙). 將掃描出的字符拿來對比, 直到不同為止, 這樣便得到一個 palindrome, 然後在所有 palindrome 中找出最長的即可.

以上 key 中此方法時間是 O(n^2)的, 如果要按暴力的 sliding window 做, 即用[l,..r]掃描所有 substring, 然後對每個 substring 調用 125. Valid Palindrome 的 isPalindrom 函數, 則可知時間是 O(n^3)的. 這就是為何要按以上 key 的方法做.

History:
E1 直接看的答案.
E2 最開始將 Java 的 substring()函數放在循環中, 每輪循環都調用一下 substring(), 結果超時了, 後來改了一下, 循環中不調用 substring(), 而是改為保存 substring 的兩端的位置, 最後返回時才調用 substring(), 就通過了, 由此可知 String.substring()函數還是挺耗時的.
E3 沒多久就寫好了, 改了一次後通過. E3 的方法跟以上 key 一樣, 代碼比 E2 略囉嗦. E3 改的那個錯誤在於, 當整個 s 都為 palindromic 時, 我最初的代碼不能返回 s, 改正的方法不是專門處理這種情況, 而是將某幾句移出 while, 改後即通過.

我: E1 沒給本題弄答案, 本題所有內容(包括 key)都是 E2 弄的. Code Ganker 給了兩種方法, 第一種是從中心往兩邊掃描, 第二種是動態規劃(E2 沒看). 兩種方法時間複雜度一樣, 空間複雜度是第一種方法更好, 所以要掌握的是第一種方法(即從中心往兩邊掃描的方法). 我 E2 的代碼也是 Code Ganker 的第一種方法, 算法一樣, 但比 Code Ganker 的簡明好懂, 以後都用我 E2 的代碼(在最後).

Code Ganker:
这道题是比较常考的题目，求回文子串，一般有两种方法。 第一种方法比较直接，实现起来比较容易理解。基本思路是对于每个子串的中心（可以是一个字符，或者是两个字符的间隙，比如串 abc,中心可以是 a,b,c,或者是 ab 的间隙，bc 的间隙）往两边同时进行扫描，直到不是回文串为止。假设字符串的长度为 n,那么中心的个数为 2*n-1(字符作为中心有 n 个，间隙有 n-1 个）。对于每个中心往两边扫描的复杂度为 O(n),所以时间复杂度为 O((2*n-1)*n)=O(n^2),空间复杂度为 O(1)，代码如下：

```
public String longestPalindrome(String s) {
    if(s == null || s.length()==0)
        return "";
    int maxLen = 0;
    String res = "";
    for(int i=0;i<2*s.length()-1;i++)
    {
```

```
        int left = i/2;
        int right = i/2;
        if(i%2==1)
            right++;
        String str = lengthOfPalindrome(s,left,right);
        if(maxLen<str.length())
        {
            maxLen = str.length();
            res = str;
        }
    }
    return res;
}
private String lengthOfPalindrome(String s, int left, int right)
{

    while(left>=0 && right<s.length() && s.charAt(left)==s.charAt(right))
    {
        left--;
        right++;
    }
    return s.substring(left+1,right);
}
```

而第二种方法是用动态规划，思路比较复杂一些，但是实现代码会比较简短。基本思路是外层循环 i 从后往前扫，内层循环 j 从 i 当前字符扫到结尾处。过程中使用的历史信息是从 i+1 到 n 之间的任意子串是否是回文已经被记录下来，所以不用重新判断，只需要比较一下头尾字符即可。这种方法使用两层循环，时间复杂度是 O(n^2)。而空间上因为需要记录任意子串是否为回文，需要 O(n^2)的空间，代码如下：

```
public String longestPalindrome(String s) {
    if(s == null || s.length()==0)
        return "";
    boolean[][] palin = new boolean[s.length()][s.length()];
    String res = "";
    int maxLen = 0;
    for(int i=s.length()-1;i>=0;i--)
    {
        for(int j=i;j<s.length();j++)
        {
            if(s.charAt(i)==s.charAt(j) && (j-i<=2 || palin[i+1][j-1]))
            {
                palin[i][j] = true;
```

```
            if(maxLen<j-i+1)
            {
               maxLen=j-i+1;
               res = s.substring(i,j+1);
            }
         }
      }
   }
   return res;
}
```

综上所述，两种方法的时间复杂度都是 O(n^2)。而空间上来看第一种方法是常量的，比第二种方法优。这个题目中假设最长回文子串只有一个，实际面试中一般不做这种假设，如果要返回所有最长回文串，只需要稍做变化就可以，维护一个集合，如果等于当前最大的，即加入集合，否则，如果更长，则清空集合，加入当前这个。实际面试会有各种变体，感觉平常还是要多想才行。

我 E2 的代碼:

```
   public String longestPalindrome(String s) {
      if(s == null || s.length() <= 1) return s;

      int maxLen = 0;
      int lReal = 0, rReal = 0;
      String res = "";

      for(int i = 0; i < s.length(); i++) {
         //中心為一個字符(如 xycabacwz 中的 b) 之情況:
         int l = i - 1, r = i + 1;
         //以下 while 和 if 可以到一個單獨的函數裡, 這樣可以避免重復代碼, E2 試過但沒寫對, 就放棄了
         while(l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {l--; r++;}
         if(r - l - 1 > maxLen) {
            maxLen = r - l - 1;
            lReal = l; rReal = r;
         }

         //中心為兩字符的間隙(如 xycaacwz 中的 aa 之間的間隙)之情況:
         l = i; r = i + 1;
         while(l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {l--; r++;}
         if(r - l - 1 > maxLen) {
            maxLen = r - l - 1;
            lReal = l; rReal = r;
         }
      }

      return s.substring(lReal + 1, rReal);
   }
```

6. ZigZag Converstion, Easy

http://wlcoding.blogspot.com/2015/03/zigzag-conversion.html?view=sidebar

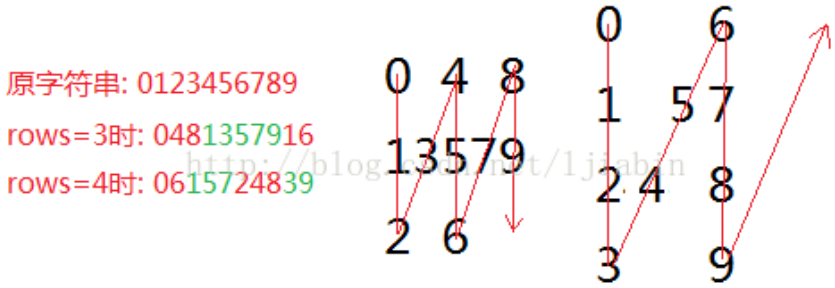
The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)
P A H N
A P L S I I G
Y I R
And then read line by line: "PAHNAPLSIIGYIR"
Write the code that will take a string and make this conversion given a number of rows:
string convert(string text, int nRows);
convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".
Subscribe to see which companies asked this question

Key : 看下面 William 說的. 記住公式 2 * nRows – 2.

History:
E1 沒做出來.
E2 改了幾次後通過, 代碼跟 William 差不多.
E3 沒單獨處理 nRows == 1 之情況(詳見下面紅字), 改後即通過, 代碼跟 Code Ganker 的差不多.

Tao: William 的代碼跟 Code Ganker 差不多, 但 Code Ganker 的更簡潔, 以後都用 Code Ganker 的代碼, 但 William 說的看一看也有幫助.

以下圖片來自 http://blog.csdn.net/ljiabin/article/details/40477429



William:

Time ~ O(N), Space ~ O(1)

红色为主元素，黑色为斜线上的元素：
P A H N
A P L S I I G
Y I R
skip1：主元素之间的距离 = 2 * nRows – 2 (我: 易用數學歸納法證之, 以後就住即可)；
skip2：主元素和斜线元素之间的距离 = 主元素间距逐次递减 2（除第一行和最后一行外，其他行每个主元素之后都跟一个斜线元素）。
注意：该方法不兼容 nRows == 1 的情况，所以必须作为 edge case 单独处理 (我: 因為 nRows == 1 時,

主元素之間的距離=0, 會出現死循環, 詳見代碼中) ← 要記住單獨處理 nRows == 1 之情況!

(我: 代碼省)

Code Ganker:

这道题是 cc150 里面的题目了，其实比较简单，只要看出来他其实每个 zigzag 是 2*m-2 个字符就可以，这里 m 是结果的行的数量。接下来就是对于每一行先把往下走的那一列的字符加进去，然后有往上走的字符再加进去即可。时间复杂度是 O(n),空间复杂度是 O(1),代码如下：

实现上要注意最后如果字符长度到了，就不需要添加了。 这道题也没有什么扩展，我觉得面试中考到的几率不大，基本就是字符串的操作。

```
public String convert(String s, int nRows) {
    if(s == null || s.length()==0 || nRows <=0)
        return "";
    if(nRows == 1)
        return s;
    StringBuilder res = new StringBuilder();
    int size = 2*nRows-2;
    for(int i=0;i<nRows;i++)
    {
        for(int j=i;j<s.length();j+=size) //若 nRows == 1, 則 size = 0, 故 for 中為 j+=0, 即死循環.
        {
            res.append(s.charAt(j));
            if(i!=0 && i!=nRows-1 && j+size-2*i<s.length())
                res.append(s.charAt(j+size-2*i));
        }
    }
    return res.toString();
}
```

---

7. Reverse Integer, Easy

http://blog.csdn.net/linhuanmars/article/details/20024837

Reverse digits of an integer.
Example1: x = 123, return 321
Example2: x = -123, return -321
click to show spoilers.

Have you thought about this?
Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!
If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.
Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?
For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.
Update (2014-11-10):
Test cases had been added to test the overflow behavior.
Subscribe to see which companies asked this question

Key: 用 res = res * 10 + x % 10 就可以反轉. 本題主要是要注意越界. res * 10 + x % 10 的值可能會越界, 所以要判斷是否 res * 10 + x % 10 > Integer.MAX_VALUE , 而算 res * 10 + x % 10 時, res*10 本身也可能越界, 所以要這樣判斷: (res > (Integer.MAX_VALUE - x % 10) / 10.

當 x 為負時, 先反轉其絕對值, 再乘-1. 當 x 為 Integer.MIN_VALUE 時, 直接返回 0(否則前面取絕對值時會越界), 因為 Integer.MIN_VALUE 反轉必然要越界, 而絕對值小於 Integer.MIN_VALUE 的數, 就可按正常地按照前面處理負數的方法處理了.

History:
E1 直接看的答案.

E2 小心寫好, 一次通過, res = res * 10 + x % 10 是自己想出來的, 代碼跟以下答案差不多. 以上代碼是 E2 寫的.

E3 本來可以一次通過的, 結果不小心范了兩個弱智錯誤: 一是最後本來該 return res, 我寫成了 return 0; 二是 (Code Ganker 代碼沒這問題)本來應該寫成 if(x < 0) return -reverse(-x), 結果我忘了 reverse 前的負號, 寫成了 if(x < 0) return reverse(-x). 改後即通過.

这道题思路非常简单，就是按照数字位反转过来就可以，基本数字操作。但是这种题的考察重点并不在于问题本身，越是简单的题目越要注意细节，一般来说整数的处理问题要注意的有两点，一点是符号，另一点是整数越界问题。代码如下. 代码为了后面方便处理，先将数字转为正数。注意 Integer.MIN_VALUE 的绝对值是比 Integer.MAX_VALUE 大 1 的，所以经常要单独处理。如果不先转为正数也可以，只是在后面要对符号进行一下判断。这种题目考察的就是数字的基本处理，面试的时候尽量不能错，而且对于 corner case 要尽量进行考虑，一般来说都是面试的第一道门槛。

```
public int reverse(int x) {
    if(x==Integer.MIN_VALUE)
        return 0;
    int num = Math.abs(x);
    int res = 0;
    while(num!=0)
    {
        if(res>(Integer.MAX_VALUE-num%10)/10)
//本来我们是要判断 res*10+num%10>MAX_VALUE 对吧？ 因为如果乘起来越界那么久错了，为了防止越界，就把他们移到右边去,就变成 res>(MAX_VALUE-num%10)/10 了~
            return 0;
        res = res*10+num%10;
        num /= 10;
    }
```

```
    return x>0?res:-res;
}
```

8. String to Integer (atoi), Easy

http://blog.csdn.net/linhuanmars/article/details/21145129

Implement atoi to convert a string to an integer.
Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.
Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.
Update (2015-02-10):
The signature of the C++ function had been updated. If you still see your function signature accepts a const char * argument, please click the reload button to reset your code definition.
spoilers alert... click to show requirements for atoi.
Requirements for atoi:
The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.
The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.
If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.
If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.
Subscribe to see which companies asked this question

Key: 本題主要是考 corner case, 沒甚麼別的技巧. 最開始可以用 string.trim()去掉前後的空格(已 record 到 Java 書). 所有的 orner case 都在題目中的 Requirements for atoi 中提到過了, 所有的 corner case 都可以總結在這個例子中: " -008573876636825392helo ". 如何正確地寫出 對溢出的判斷 是難點.

以下結論記住:
判斷語句 value * 10 + d > 2147483647
跟 value > (2147483647 - d) / 10) 是完全等價的.
原因很簡單, 詳見 E2 代碼中.

History:
E1 沒做出來.
E2 改了很多次, 最後總算通過, 主要是越界那裡老是出錯.
E3 改了兩個粗心小錯誤(忘了要求 i 小於 str 長度, 負數越界時忘了在 value 前加負號)後通過. 所以本題其實並不難, 不要被本題的低通過率嚇到了.

我: 本題雖然是 Easy, 但通過率只有 13.1%, 比絕大多數 Hard 都低, 本題的通過率在 leetcode 中排第二低, 僅次於 65. Valid Number (11.8%). 我 E2 的代碼(在最後)比 Code Ganker 的更簡明, 以後都用 E2 的代碼. 以下 Code Ganker 說的可以不看.

这道题还是对于 Integer 的处理，在 Reverse Integer 这道题中我有提到，这种题的考察重点并不在于问题本身，而是要注意 corner case 的处理，整数一般有两点，一个是正负符号问题，另一个是整数越界问题。思路比较简单，就是先去掉多余的空格字符，然后读符号（注意正负号都有可能，也有可能没有符号），接下来按顺序读数字，结束条件有三种情况：（1）异常字符出现（按照 C 语言的标准是把异常字符起的后面全部截去，保留前面的部分作为结果）；（2）数字越界（返回最接近的整数）；（3）字符串结束。代码如下.

这道题主要考察整数处理，注意点上面已经提到过，因为这个问题是 C 语言的一个基本问题，面试中还是有可能出现，相对比较底层，边缘情况的处理是关键，可能面试 tester 的职位会更常见一些。

(以下是 Code Ganker 的代碼, 以後都看我 E2 的代碼, 在最後)

```
public int atoi(String str) {
    if(str==null)
    {
        return 0;
    }
    str = str.trim();
//Tao: String.trim(): It returns a copy of this string with leading and trailing white space removed, or this string if it has no leading or trailing white space.
    if(str.length()==0)
        return 0;
    boolean isNeg = false;
    int i = 0;
    if(str.charAt(0)=='-' || str.charAt(0)=='+')
    {
        i++;
        if(str.charAt(0)=='-')
            isNeg = true;
    }
    int res = 0;
    while(i<str.length())
    {
        if(str.charAt(i)<'0'||str.charAt(i)>'9')
            break;
```

```
        int digit = (int)(str.charAt(i)-'0');
        if(isNeg && res>-((Integer.MIN_VALUE+digit)/10))
//Tao: 注意 Math.abs(Integer.MIN_VALUE) =  Integer.MIN_VALUE, 即還是負數
            return Integer.MIN_VALUE;
        else if(!isNeg && res>(Integer.MAX_VALUE-digit)/10)
            return Integer.MAX_VALUE;
        res = res*10+digit;
        i++;
    }
    return isNeg?-res:res;
}
```

E2 的代碼:

```
    public static int myAtoi(String str) {
        if(str == null) return 0;
        str = str.trim();
        if(str.length() == 0) return 0;

        int sign = 1;
        int value = 0; // The absolute value

        for(int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);

            if(i == 0 && c == '+') {
                continue;
            } else if(i == 0 && c == '-') {
                sign = -1;
                continue;
            } else if(!Character.isDigit(c)) {
                break;
            } else {
                int d = c – '0';

// Integer.MAX_VALUE = 2147483647, Integer.MIN_VALUE = -2147483648.
//以下的記住:
判斷語句  value * 10 + d >  2147483647 (判斷句 A)
跟 value > (2147483647 - d) / 10) (判斷句 B) 是完全等價的. 原因很簡單:
此問題顯然在 value = 214748364 時我們才 care, 此時

當 d 小於 7 時, value * 10 + d 小於 2147483647, 即  判斷句 A  為 false
```

而此時(2147483647 - d) / 10 = 214748364, 它跟 value 是相等的, 即 判斷句 B 也為 false

當 d 大於 7 時, value * 10 + d 大於 2147483647, 即 判斷句 A 為 true
而此時(2147483647 - d) / 10 = 214748363, value 大於它, 即 判斷句 B 也為 true.

```
            if(sign == 1 && value > (Integer.MAX_VALUE - d) / 10) return Integer.MAX_VALUE;
            if(sign == -1 && -value < (Integer.MIN_VALUE + d) / 10) return Integer.MIN_VALUE;

            value = value * 10 + d;
        }

    }

    return sign * value;
}
```

---

9. Palindrome Number, Easy

http://wlcoding.blogspot.com/2015/03/palindrome-i-valid-num-ii-valid-str-iii.html?view=sidebar

Determine whether an integer is a palindrome. Do this without extra space.
click to show spoilers.
Some hints:
Could negative integers be palindromes? (ie, -1)
If you are thinking of converting the integer to string, note the restriction of using extra space.
You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?
There is a more generic way of solving this problem.
Subscribe to see which companies asked this question

Key: 每次將 x 的首尾兩位去掉, 將首尾兩位拿來比較. 得到尾位的方法是 532%10=2, 而得到首位的方法是 532/100=5, 即先要算出 100 這個數(代碼中叫 div).

History:
E1 直接看的答案.
E2 方法跟 William 的第一種差不多, 但在算 div 時, while 中的條件沒寫好, 導致 div 老是溢出, 改了幾次後才通過, 以上 key 是 E2 寫的.
E3 幾分鍾寫好, 改了一個錯誤後才通過. E3 方法跟 key 中一模一樣. 這個錯誤在於, 我算 div 時, 是寫成的 while (x / (div * 10) > 0), 所以當 x 較大時, div*10 會越界, 後來改成了 while((x / 10) / div > 0)才通過, 不過還是沒有 William 的 while (x / div >= 10)好.

palindrome: 回文（指顺读和倒读都一样的词语）, [美]['pælɪndroʊm] ← 注意讀法和重音

我: 以下 William 給了兩種方法, 要掌握的是第一種, 因為第二種是 將數反轉 的方法, 而題目的 click to show

spoilers 中說反轉的方法可能會溢出. 但實際上反轉的方法並不會出現溢出, 且第二種方法也很簡單, 一眼就能看完, 所以第二種方法也可以看看, 順便學學如何反轉一個數. Code Ganker 也用的 William 的第一種方法.

William:

1. Time ~ O(2N) where N is number of digits, Space ~ O(1)

```
public boolean isPalindrome(int x) {
    if (x < 0)  return false;
    int div = 1;
    while (x / div >= 10) {
        div *= 10;
    }
    int num = x; //E2 沒用 num, 直接對 x 操作的, 也能通過. 當然最後還是對 num 操作.
    while (num != 0) {
        int l = num / div;
        int r = num % 10;
        if (l != r) return false;
        num = (num % div) / 10;
        div /= 100;
    }
    return true;
}
```

2. Time ~ O(N), Space ~ O(1)
Reverse the integer and compare if it's equal to the original one.
注意：这道题不用担心 overflow 的问题，因为 overflow 的数不会是 palindrome ← 我不贊同, 因為 test case 可以不是 palindrome 的, 例如 1111111889, 雖然本代碼也能在 OJ 上通過, 但應該是因為運氣好, 沒有出現 1111111889 這樣的 test case.

```
public boolean isPalindrome(int x) {
    int num = x, val = 0;
    if (x < 0)  return false;
    while (num != 0) {
        val = val * 10 + num % 10; //val 是 num 反轉後的數. 反轉的方法即不停地將 num 的尾位放到 val 的尾位: {val, num} = {0, 123} → {3, 12} → {32, 1} → {321, 0}
        num = num / 10;
    }
    if (val == x)   return true;
    else            return false;
}
```

---

10. Regular Expression Matching, Hard

http://blog.csdn.net/linhuanmars/article/details/21145563

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
Subscribe to see which companies asked this question

注意題目表述得不清楚. 實際上應當這麼說: 'The preceding element of * in p' and * together matches zero or more of the preceding element of * in p. 所以題目中的例子 isMatch("aab", "c*a*b") → true, 是因為"c*a*b" could be 0 c 2 a and 1 b, it matched aab (from leetcode discussioin forum)

Key: DP. 連 NB 的 E3 都表示本題很難, key 也不好懂. 本題方法跟 44. Wildcard Matching 類似. 用 d(i, j)表示 's 的前 i 個字符(即 s[0, ..i-1])' 和 'p 的前 j 個字符(即 p[0, j-1])' 是否 match(即 isMatch(s[0, ..i-1], p[0, j-1])之結果).

若 p[j-1]可日 s[i-1], 則 d(i, j) = d(i-1, j-1), (p[j]可日 s[i] 的意思就是 p[j]==s[i] || p[j] == '.')

否則, 若 p[j-1]為*, 則有兩種情況:

一是 p[j-2]不可日 s[i-1], 這種情況就比較簡單, 直接 d(i, j) = d(i, j-2), 例如 s = a b(i-2) c(i-1), p = a b(j-3) e(j-2) *(j-1), 其中 p 中的 e*一起表示取 0 個 e, 即看 s = a b(i-2) c(i-1)和 p = a b(j-3)是否 match, 即 d(i, j) = d(i, j-2).

二是 p[j-2]可日 s[i-1], 則 d(i, j) = d(i, j-2) || d(i-1, j). 等於 d(i, j-2), 例如 s = a b(i-2) c(i-1), p = a b c(j-3) c(j-2) *(j-1), 其中 p 中的 c*一起表示取 0 個 c, 即看 s = a b(i-2) c(i-1)和 p = a b c(j-3)是否 match, 此即 d(i, j-2). 而||後的 d(i-1, j), 例如 s = a b c e(i-2) e(i-1), p = a b c(j-3) e(j-2) *(j-1), 此時就要看 s 去掉最後一個字符後, 是否仍和 p match, 即若 s = a b c e(i-2)和 p = a b c(j-3) e(j-2) *(j-1) match, 則 s 後面再加一個 e, 即 s = a b c e(i-2) e(i-1)和 p = a b c(j-3) e(j-2) *(j-1)也 match, 所以此時 d(i, j)=d(i-1, j). 別忘了本情況假定 p[j-2]可日 s[i-1]. E2.5 表示 d(i, j) = d(i, j-2)好像被上面"一是"包括了, 要改. E3 試了下, 去掉 d(i, j) = d(i, j-2)後通不過, 實際上這是必要的, 注意那個||表示只要有一種 OK 即可以, 此時雖然 p[j-2]可日 s[i-1], 但我們也有權不用這點, 而像上面"一是"那樣, 將 p 中的 c*一起表示取 0 個 c.

初始化 d:
d[0][0]=true,
d(0, j), 即 s 只第一個字母, 若 p[j-1] = '*' 則 d(0, j) = d(0, j-2), 否則 d(0, j)=false. 實際上, d(0,j)為 true 時只有一種情況, 即 s=a, p=aaa*(當然 p 也可為****).

History:
E1 直接看的 Code Ganker 答案.
E2 為了與 44. Wildcard Matching 一致, 就按 William 的 2-d DP 算法寫的, 寫好後改了兩個小錯(j=1 和<=)

後即通過. Key 是 E2 寫的, 有 key 就好寫, 最好不要看 key. 以後就參考 William 的 2-d DP 代碼. 與 44 題一樣, 本題的 1d-DP 也要用 prev 和 cur 保存歷結果, 所以不要求掌握 1d-DP.
E3 沒做出來.

William:
Solution
•'*' 代表前一个字母（包括为空）的一个或多个 copy，或删除前一个字母！
•p 不会以 '*' 开头。
•".*" 可代表一切 String。

1. DFS: Time ~ O(?)
s[.. i, I + 1 ...]
p[.. j, j + 1 ...]
两种情况：
If p[j + 1] != '*', 只有一种情况 match: p[j] == s[i] || p[j] == '.'
If p[j + 1] == '*', 又分两种情况 match（可以 implement 在一起，不是非常 straightforward，需要看一下）：
1) '*' is deletion, check if s[i] == p[j + 2];
2) '*' is repetition, 从 s[i]向前找，找到第一个 s[i + k] != p[j], check if s[i + k] == p[j + 2]

```
public boolean isMatch(String s, String p) {
    if (p.length() == 0)    return s.length() == 0;

     if (p.length() == 1 || p.charAt(1) != '*') { // s[i] == p[j] || p[j] == '.' must hold for matching
        if (s.length() > 0 && (p.charAt(0) == s.charAt(0) || p.charAt(0) == '.'))
            return isMatch(s.substring(1), p.substring(1));
        else    return false;
    } else { // p.charAt(1) == '*'
          while (s.length() > 0 && (p.charAt(0) == s.charAt(0) || p.charAt(0) == '.')) {
                // 1st run: * is deletion, >= 2nd run: * is repetition; there is redundancy here!!
            if (isMatch(s, p.substring(2))) return true;
            s = s.substring(1);
        }
        return isMatch(s, p.substring(2));
    }
}
```

2. 2-d DP: Time ~ O(SP), Space ~ O(SP)
Let d(i, j) = true if s[0, i - 1] matches p[0, j - 1] (i, j are string lengths).
Initialize:

•d(0, 0) = true,
•d(0, j): if p[j - 1] == '*', d(j) = d(0, j - 2) // deletion; else d(j) = false.

Fill up the table:
if **p[j - 1] matches s[i - 1],** d(i, j) = d(i - 1, j - 1);
else if **p[j - 1] == '*',** two cases:
if **p[j - 2] matches s[i - 1],** d(i, j) = deletion: d(i, j - 2) || repetition: d(i - 1, j);

else d(i, j) = deletion: d(i, j - 2);

Note: “p[j] matches s[i]” means p[j] == s[i] || p[j] == '.'.

Return d(M, N).

```
public boolean isMatch(String s, String p) {
    int lenS = s.length(), lenP = p.length();
     boolean[][] d = new boolean[lenS + 1][lenP + 1]; // i, j are the lengths of s[0..i-1] and p[0..j-1]
    d[0][0] = true;
    // initialize the first row
    for (int j = 1; j <= lenP; j++) {
        if (p.charAt(j - 1) == '*') d[0][j] = d[0][j - 2]; // * is deletion. 例如 s = “”, p = “a*”
    }
    // fill up the table
    for (int i = 1; i <= lenS; i++) {
        for (int j = 1; j <= lenP; j++) {
            if (p.charAt(j - 1) == s.charAt(i - 1) || p.charAt(j - 1) == '.')
                d[i][j] = d[i - 1][j - 1];
            else if (p.charAt(j - 1) == '*') {
                if (p.charAt(j - 2) == s.charAt(i - 1) || p.charAt(j - 2) == '.')
                        d[i][j] = d[i][j - 2] || d[i - 1][j]; // * is deletion or repetition
                else
                    d[i][j] = d[i][j - 2]; // * is deletion
            }
        }
    }
    return d[lenS][lenP];
}
```

3. 1-d DP (滚动数组) : Time ~ O(SP), Space ~ O(P)

```
public boolean isMatch(String s, String p) {
    int lenS = s.length(), lenP = p.length();
    boolean[] d = new boolean[lenP + 1];
    d[0] = true;
    // initialize the first row
    for (int j = 1; j <= lenP; j++) {
        if (p.charAt(j - 1) == '*')
            d[j] = d[j - 2]; // * is deletion
    }
    // fill up the table
    for (int i = 1; i <= lenS; i++) {
        boolean prev = d[0];    // prev stores d[i-1][j-1]
        d[0] = false;   // add this line for 1D reduction!!
        for (int j = 1; j <= lenP; j++) {
            boolean curr = d[j]; // curr stores d[i-1][j]
            if (p.charAt(j - 1) == s.charAt(i - 1) || p.charAt(j - 1) == '.')
                d[j] = prev;
            else if (p.charAt(j - 1) == '*') {
                if (p.charAt(j - 2) == s.charAt(i - 1) || p.charAt(j - 2) == '.')
                    d[j] = d[j - 2] || curr; // * is deletion or repetition
                else
                    d[j] = d[j - 2]; // * is deletion
            } else
                d[j] = false; // add this line for 1D reduction!!
```

```
                prev = curr;
            }
        }
        return d[lenP];
}
```

Code Ganker:

这个题目比较常见，但是难度还是比较大的。我们先来看看 brute force 怎么解决。基本思路就是先看字符串 s 和 p 的从 i 和 j 开始的子串是否匹配，用递归的方法直到串的最后，最后回溯回来得到结果。假设现在走到 s 的 i 位置，p 的 j 位置，情况分为下列两种：

(1)p[j+1]不是'*'。情况比较简单，只要判断当前 s 的 i 和 p 的 j 上的字符是否一样（如果有 p 在 j 上的字符是'.',也是相同），如果不同，返回 false，否则，递归下一层 i+1，j+1;

(2)p[j+1]是'*'。那么此时看从 s[i]开始的子串，假设 s[i],s[i+1],...s[i+k]都等于 p[j]那么意味着这些都有可能是合适的匹配，那么递归对于剩下的(i,j+2),(i+1,j+2),...,(i+k,j+2)都要尝试（j+2 是因为跳过当前和下一个'*'字符）。

实现代码如下：

```
public boolean isMatch(String s, String p) {
   return helper(s,p,0,0);
}
private boolean helper(String s, String p, int i, int j)
{
   if(j==p.length())
      return i==s.length();

   if(j==p.length()-1 || p.charAt(j+1)!='*')//Tao:  j==p.length()-1 是對 charAt(j+1)之約束
   {
      if(i==s.length()|| s.charAt(i)!=p.charAt(j) && p.charAt(j)!='.')
         return false;
      else
         return helper(s,p,i+1,j+1); //Tao: 若 j+1=s.length(), 則在運行 helper(s,p,i+1,j+1)時會在最開始的 if(j==p.length())中處理
   }
   //Tao: 以下的語句是上面兩個 if(j==p.length())和 if(j==p.length()-1 || p.charAt(j+1)!='*')剩下來的, 即只有 j < p.length() - 1 且 p.charAt(j+1) == '*'才會執行
```

```
    //p.charAt(j+1)=='*'
    while(i<s.length() && (p.charAt(j)=='.' || s.charAt(i)==p.charAt(j)))// Tao: i<s.length()是對 i++的約束
    {
        if(helper(s,p,i,j+2))
            return true;
        i++;
    }
    return helper(s,p,i,j+2); //Tao: 此句之是 while 剩下來的, 比如 i=s.length()時, 會在運行 helper(s,p,i+1,j+2)時在最開始的 if(j==p.length())中處理. 又如 s.charAt(i)!=p.charAt(j)時, 運行 helper(s,p,i,j+2), 這是考慮到這種情況: s=abc, p=abd*c, 它使得 s.charAt(i)=b, p.charAt(j)=d, 但我們還是認為 s=p, 因為 d*可表示 0 個 d
}
```

---

11. Container With Most Water, Medium

Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.
Note: You may not slant the container.
Subscribe to see which companies asked this question

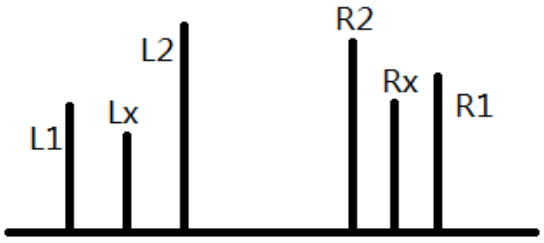
Key: 本題知道方法了後還是很簡單的. 先把 l 設為最左, r 設為最右, 若 height[l] < height[r], 則 l++, 否則 r--. 在這樣得出的所有面積中找最大的. 下面有我寫的證明. 但最好直接把方法記住.

History:
E1 直接看的答案.
E2 幾分鍾寫好一次通過.
E3 記得方法, 所以幾分鍾寫好, 本來可以一次通過的, 結果范了個弱智錯誤, 就是不小心將 r 初始化為了 height.length, 而應該為 height.length - 1. 改後即通過.

以下來自:
http://www.cnblogs.com/lautsie/p/3219461.html
此题一开始看错了，后来才发现由小的一根（短板）决定高度。中间看过网上的解答，思路是 O(n)的，略巧妙。其实就是基于这样一个直觉，先把 L1 设为最左，R1 设为最右，这样宽度最大，然后缩小宽度，那么高度就要比原来高。缩小宽度的逻辑是，如果 L1 比 R1 小，那么 L1 往右移，否则反之。可以做简单证明 (Tao: 上面那個網址有證明, E2 難得看)。

Tao: 以下是我寫的證明, 參考了 http://blog.csdn.net/maverick1990/article/details/37110525

證明: 設本題之解為 left, right. 注意 L 從左往右掃, R 從右往左掃, 最後 L 和 R 只差 1. 說明 left 和 right 總是會被 L 或 R 掃到. 用反證法: 假設用以上方法不能找出解. 此假設說詳細點, 即為: 不能同時讓 left 被 L 掃到, right 被 R 掃到.
若 left 先被 L 掃到 (right 先被掃到的情況類似可證), 而此時的 R 不在 right 上, 設 R 在 arr 上(arr 必在 right 之右, 否則就成了 right 先被掃到的情況了), 並且根据假設, 即由假設可知: 下一步是 L 移動, 然後我們就永遠失去了 L. 之後就算 R 掃到 right 上, 也再也不能讓同時讓 left 被 L 掃到, right 被 R 掃到 了.
可以知道以上是不可能的. 因為若下一步是 L 移動, 則說明 height[L] < height[R]. 那麼 L 和 R 形成的面積為(R – L) * L (注意 L 是短板). 由於 R 在 right 之右, 可知 R – L 大於 right – L, 即 L 和 R 形成的面積 大於 L(即 left)和 right 形成的面積(which is (right – L) * L). 設與前面說的 left 和 right 為解 矛盾. 故下一步不可能是 L 移動, 即 L 會守在 left 的位置上, 等著 R 走到 right 上. E3 表示以上暗含了 height[L]<height[right] ← 這個命題為真嗎? E3 沒認真想.



以下來自:

http://harrifeng.github.io/algo/leetcode/container-with-most-water.html

这个题目很容易联想到动态规划:max(i) = max{max(i-1), size introduced by j}, 其实动态规划也是可以解决这个问题的, 但是动态规划就要求记住每一步选择的结果,对于 这种求最大值的问题,往往意味着很多的冗余.

最优解法非常巧妙,利用了常见的"双指针"法, 两个指针一个指向最前,一个指向最后, 然后 "数值小"的指针向里运动, 这里利用了"水桶的容积只取决于最断的那个木板"的原理!,最终 把复杂度降到了 O(n)

```
public class Solution {
    public int maxArea(int[] height) {
        int start = 0;
        int end = height.length - 1;
        int maxV = Integer.MIN_VALUE;

        while (start < end) {
            int contains = Math.min(height[end], height[start]) * (end - start); //tao: 循環中也定義 int 數, 可能不大好
            maxV = Math.max(maxV, contains);
```

```
                if (height[start] < height[end]) {
                    start ++;
                } else {
                    end--;
                }
            }
            return maxV;
        }
}
```

---

12. Integer to Roman, Medium

https://stupidcodergoodluck.wordpress.com/2014/03/31/leetcode-integer-to-roman/

Given an integer, convert it to a roman numeral.
Input is guaranteed to be within the range from 1 to 3999.
Subscribe to see which companies asked this question

Key: 記住方法即可, 不用證明. 注意題目只要求輸入的數在 3999 以內. 將輸入的數(num)跟這樣一堆數比較: {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1}, 若 num 大於 1000, 則在 result 中加入 1000 對應的羅馬數字(M), 並在 num 中減去 1000; 然後再比較, 比如若此時的 num 在 90 和 50 之間, 就加入 50 的羅馬數字, 並在 num 中減去 50,... 如此重復.

新記憶法(E3 想出的):
上面那一堆數即{M, CM, D, CD, C, XC, L, XL, X, IX, V, IV, I}, 它們其實是 Roman 數字的 building block. 它們可以分為兩類:

第一類是單個字母的, 即{M, D, C, L, X, V, I}, 這就是 Roman 數字的基本字母啊(13 題中按 I View X-ray, Lucy Cows Drink Milk 來記的).

第二類是兩個字母的, 即{CM, CD, XC, XL, IX, IV}, 它們其實就是列舉了所有「小數在左即減」的情況. 如何將它們列出來? 可以看以下(其中紅的(IXC)是可左減的, 13 題中有記憶法):

```
I = 1;
V = 5;
X = 10;
L = 50;
C = 100;
D = 500;
M = 1000;
```

{CM, CD, XC, XL, IX, IV}可以這樣得來:
右邊的字母(如 CM 中的 M)即將 Roman 數字的 基本字母 遍歷一遍(I 除外, 因為 I 不能左減誰), 且每個 基本字母 只能出現一次, 故為: M, D, C, L, X, V,
左邊的字母(如 XC 中的 X)是 小於 右邊的字母的 最大的 左減數 (如 XC 中的 X 是小於 C 的最大的左減數).

老記憶法(E3 因沒記住 而沒做出來): 記憶以上那堆數的方法為: 它們都是 1, 4, 5, 9 的整 10 倍, 而 1, 4, 5, 9 按 伊(1)斯(4)蘭無(5)酒(9) 來記. 還可用這樣來 確認記對了: 1 5 10 分別減 1, 得 0 4 9, 而 0 不能用, 故為 49, 故總體為 1459(注意只能用此來確認, 而不能用它來記, 否則會不確定是否有 100 減 1, 即 99). 注意還要記住是十倍十倍地變, 還要記住是 1459 四個數, 而不是三個數. E3 就因為 這兩個地方沒記清楚 而沒做出來: 一是不確定到底是 400, 還是 499, 還是 444, 二是誤記為了 伊(1)斯(4)蘭教(9), 而少了個 5.

History:
E1 直接看的答案.
E2 比較快寫好, 一次通過. 代碼跟以下答案的不同之處在於, 我是 while 在外, for 在內, 這樣效率比答案略低.
E3 沒做出來, 因為有兩個地方沒記清楚: 一是不確定到底是 400, 還是 499, 還是 444, 二是誤記為了 伊(1)斯(4)蘭教(9), 而少了個 5.

Roman Numeral Table

1	I	**14**	XIV	**27**	XXVII	**150**	CL
2	II	**15**	XV	**28**	XXVIII	**200**	CC
3	III	**16**	XVI	**29**	XXIX	**300**	CCC
4	IV	**17**	XVII	**30**	XXX	**400**	CD
5	V	**18**	XVIII	**31**	XXXI	**500**	D
6	VI	**19**	XIX	**40**	XL	**600**	DC
7	VII	**20**	XX	**50**	L	**700**	DCC
8	VIII	**21**	XXI	**60**	LX	**800**	DCCC
9	IX	**22**	XXII	**70**	LXX	**900**	CM
10	X	**23**	XXIII	**80**	LXXX	**1000**	M
11	XI	**24**	XXIV	**90**	XC	**1600**	MDC
12	XII	**25**	XXV	**100**	C	**1700**	MDCC
13	XIII	**26**	XXVI	**101**	CI	**1900**	MCM

MathATube.com

对老中来说，不熟悉罗马数字的规律是个问题。其实就是那么几个符号，然后规律就是 1,4,5,9 然后每次十倍循环。算法就是给一个数，大于 1000 的时候就写上一个 M， 还大于 1000，再写一个 M… 终于小于 1000 了，看看大于 900 不？，大于 500 不…如此类推。

```
/*
I = 1;
```

```
V = 5;
X = 10;
L = 50;
C = 100;
D = 500;
M = 1000;
*/

public class Solution {
    public String intToRoman(int num) {
        String[] symbols =
{"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
        int[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
        StringBuilder ret = new StringBuilder();
        for (int i=0; i<values.length; i++) {
            while (num >= values[i]) { //Tao: 注意是 >=, 不是>. 記憶, 比如 num=5 時. 另外,
 while 不要寫成 if 了.
                ret.append(symbols[i]);
                num -= values[i];
            }
        }
        return new String(ret);
    }
}
```

13. Roman to Integer. Easy

http://blog.csdn.net/wzy_1988/article/details/17057929

Given a roman numeral, convert it to an integer.
Input is guaranteed to be within the range from 1 to 3999.
Subscribe to see which companies asked this question

Key: 先看下面的計數規則. 記住方法即可: 從左往右掃描, 若當前數(cur)小於等於它左邊的數(pre), 則在結果中加上 cur; 否則在結果中減去 pre 的兩倍, 再加上 cur , 這是因為此情況時應當減去 pre, 而之前掃到 pre 時, 已經加了 pre, 所以要減 pre 的兩倍. 建議用 switch, 不用 Map, 因為 switch 可以不用空間.

Roman Numeral Table

1	I	14	XIV	27	XXVII	150	CL
2	II	15	XV	28	XXVIII	200	CC
3	III	16	XVI	29	XXIX	300	CCC
4	IV	17	XVII	30	XXX	400	CD
5	V	18	XVIII	31	XXXI	500	D
6	VI	19	XIX	40	XL	600	DC
7	VII	20	XX	50	L	700	DCC
8	VIII	21	XXI	60	LX	800	DCCC
9	IX	22	XXII	70	LXX	900	CM
10	X	23	XXIII	80	LXXX	1000	M
11	XI	24	XXIV	90	XC	1600	MDC
12	XII	25	XXV	100	C	1700	MDCC
13	XIII	26	XXVI	101	CI	1900	MCM

MathATube.com

History:
E1 通過.

E2 幾分鍾寫好, 一次通過. 用的是 Map. 用 Map 的代碼要簡明很多, 但以下的答案和 Code Ganker 都是用的 switch, 不知道為甚麼(因為用 switch 可以達到空間 O(1)啊, Code Ganker 自己也說空間是 O(1)). 我還是將我的 E2 代碼帖在最後.

E3 五分鍾寫好, 一遍通過. E3 也用的 Map.

首先，学习一下罗马数字，参考罗马数字

罗马数字是最古老的数字表示方式，比阿拉伯数组早 2000 多年，起源于罗马

罗马数字有如下符号：

基本字符	I	V	X	L	C	D	M
对应阿拉伯数字	1	5	10	50	100	500	1000

以上記憶(from online): 字母記法為: I view X-rays. Lucky cows drink milk.
數字記法為: 以 1 和 5 為基本數, 然後不停地乘 10, 即: 1, 5 → 10, 50 → 100, 500 → 1000

计数规则(第 1 條是癈話, 第 4 和 5 條本題用不著, 故只有第 2 和 3 條要記住)：

1. 相同的数字连写，所表示的数等于这些数字相加得到的数，例如：III = 3
2. 小的数字在大的数字右边，所表示的数等于这些数字相加得到的数，例如：VIII = 8
3. 小的数字，限于 I、X 和 C(即 1, 10, 100, 好記, 注意沒有 M, 因為沒有比 M 更大的了）在大的数字左边，所表示的数等于大数减去小数所得的数，例如：IV = 4
4. 正常使用时，连续的数字重复不得超过三次 (tao: 注意本題和 12. Integer to Roman 都只要求數字在 3999 以內, 4000 就是 MMMM, 但不管)
5. 在一个数的上面画横线，表示这个数扩大 1000 倍（本题只考虑 3999 以内的数，所以用不到这条规则）

其次，罗马数字转阿拉伯数字规则（仅限于 3999 以内）：

从前向后遍历罗马数字，如果某个数比前一个数小，则加上该数。反之，减去前一个数的两倍然后加上该数

答案代碼(用它):

```
import java.util.Scanner;

public class RomanToInt {
```

```
public static int charToInt(char c) {
    int data = 0;

    switch (c) {
        case 'I':
            data = 1;
            break;
        case 'V':
            data = 5;
            break;
        case 'X':
            data = 10;
            break;
        case 'L':
            data = 50;
            break;
        case 'C':
            data = 100;
            break;
        case 'D':
            data = 500;
            break;
        case 'M':
            data = 1000;
            break;
    }

    return data;
}

public static int romanToInt(String s) {
    int i, total, pre, cur;

    total = charToInt(s.charAt(0));

    for (i = 1; i < s.length(); i++) {
        pre = charToInt(s.charAt(i - 1));
        cur = charToInt(s.charAt(i));

        if (cur <= pre) {
            total += cur;
        } else {
            total = total - pre * 2 + cur;
        }
    }

    return total;
}
```

```
    public static void main(String[] args) {
        String s;
        int d;
        Scanner cin = new Scanner(System.in);

        while (cin.hasNext()) {
            s = cin.nextLine();

            d = romanToInt(s);

            System.out.println(d);
        }

        cin.close();
    }
}
```

E2 的代碼(不用. 因為雖然用 Map 會使代碼更簡明, 但要用空間, 所以還是應該像答案那樣用 switch):

```
    public int romanToInt(String s) {
        if(s == null || s.length() == 0) return 0;
        Map<Character, Integer> map = new HashMap<>();
        map.put('I', 1);
        map.put('V', 5);
        map.put('X', 10);
        map.put('L', 50);
        map.put('C', 100);
        map.put('D', 500);
        map.put('M', 1000);

        int res = map.get(s.charAt(0));

        for(int i = 1; i < s.length(); i++) {
            int cur = map.get(s.charAt(i));
            int pre = map.get(s.charAt(i - 1));
            if(cur <= pre) res += cur;
            else res = res - 2 * pre + cur;
        }

        return res;
    }
```

14. Longest Common Prefix, Easy

http://blog.csdn.net/linhuanmars/article/details/21145733

Write a function to find the longest common prefix string amongst an array of strings.
Subscribe to see which companies asked this question

Key: brute force. 以第一个字符串为标准(longest common prefix 必定也是第一个字符串的 prefix)，对于每个字符串从第一个字符开始，看看是不是和标准一致：

(我 E2 的方法)寫一個 int compare(String standard, String str)函數求 standard 和 str 的 common prefix 所在的位置(的下一個)，然後對所有 string 都調用 compare, 然後 compare 返回值中最小的即為要求的 longest common prefix 所在的位置．

Convention: 當無 longest common prefix 時(如輸入數組為空: []), 則應返回"", 而不是 null.

History:
E1 沒做出來.
E2 很快寫好, 但改了兩次後才通過, 第一次改是因為沒考慮到 strs 數組只有一個元素之情況, 第二次改是因為 最開始我在各 'strs[0]和 strs[i]的 common prefix' 中找的最長的, 實際上應當找最短的.
E3 很快寫好, 一次通過.

我： Code Ganker 的方法是橫向比較: 若所有 string 的第 idx 個字符都相同，都將這個字符加入到 結果字符串 res 中．我 E2 的方法是縱向比較．兩種方法時間復雜度都為 O(m*n), m 為所有 string 最大長度, n 為 string 總個數，空間復雜度應該都是 O(1) (不算結果佔的空間的話，Code Ganker 說的是 O(m), 應該是要算結果佔的空間)．但我 E2 的代碼更清新好懂，以後都用我 E2 的代碼．

Code Ganker:

这道题属于字符串处理的题目，思路比较简单，就是 brute force 的想法，以第一个字符串为标准，对于每个字符串从第一个字符开始，看看是不是和标准一致，如果不同，则跳出循环返回当前结果，否则继续下一个字符。时间复杂度应该是 O(m*n)，m 表示字符串的最大长度，n 表示字符串的个数，空间复杂度应该是 O(m),即字符串的长度

代码设置了一个标记，来标记是否结束。细心的读者可能发现中间那个循环 index 是从 0 开始，按理说不用对第一个字符串进行判断了，因为他是标准，这么做的目的其实是因为 leetcode 有一个测试集是空串，如果不对 0 进行判断，那么就没有设置 flag 为 false，跑到第二层就会越界。当然也可以手动判断一下，这个其实是小问题。

Code Ganker 之代碼：

```
public String longestCommonPrefix(String[] strs) {
    StringBuilder res = new StringBuilder();
    if(strs == null || strs.length==0)
        return res.toString();
    boolean sameFlag = true;
```

```
    int idx = 0;
    while(sameFlag)
    {
        for(int i=0;i<strs.length;i++)
        {
            if(strs[i].length()<=idx || strs[i].charAt(idx)!=strs[0].charAt(idx))
            {
                sameFlag = false;
                break;
            }
        } //end of for
        if(sameFlag)
            res.append(strs[0].charAt(idx));
        idx++;
    } //end of while
    return res.toString();
}
```

我 E2 的代碼：

```
    public String longestCommonPrefix(String[] strs) {
        if(strs.length == 0 || strs[0].length() == 0)
            return "";

        if(strs.length == 1)
            return strs[0];

        int min = Integer.MAX_VALUE; //E3 是將 min 初始化為 strs[0].length(), 這樣不不需要前面那個 if(strs.length == 1)  return strs[0]了

        for(int i = 1; i < strs.length; i++) {
            min = Math.min(min, compare(strs[0], strs[i]));
        }

        return strs[0].substring(0, min);
```

```
    }

    private int compare(String standard, String str) {
        int minLen = Math.min(standard.length(), str.length());

        int i = 0;

        while(i < minLen) {
            if(standard.charAt(i) != str.charAt(i))
                break;
            i++;
        }

        return i;
    }
```

---

15. 3Sum, Medium

http://blog.csdn.net/linhuanmars/article/details/19711651

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

- Elements in a triplet (*a*,*b*,*c*) must be in non-descending order. (ie, *a* ≤ *b* ≤ *c*)
- The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:
(-1, 0, 1)
(-1, -1, 2)
Subscribe to see which companies asked this question

Key: 先 sort num. 調用 twoSum(int[] num, int end, int target), 即在 num[0,..end]中找和為 target 的數.twoSum 函數要求輸入的數組是 sorted, twoSum 函數中用 twoSum 這題中一樣的兩頭夾逼法. 在 threeSum 中, i 從右往左掃, 並在 num[i]左邊找 '和為-num[i]的數', 即調用 twoSum(num,i-1,-num[i]). 注意跳過相同的元素.

History:
E1 直接看的答案.
E2 電腦上 test case 可通過, 但 OJ 中超時, 原因尚不清楚.
E3 最開始調用 twoSum 時不小心將 twoSum(num, i-1,-num[i])中的負號忘了, 且在 threeSum 函數中沒跳過重復元素(twoSum 中跳了), 改後即通過, 代碼跟 Code Ganker 差不多.

这道题是 Two Sum 的扩展，brute force 时间复杂度为 O(n^3), 对每三个数进行比较。这道题和 Two Sum 有所不同，使用哈希表的解法并不是很方便，因为结果数组中元素可能重复，如果不排序对于重复的处理将会比较麻烦，因此这道题一般使用排序之后夹逼的方法，总的时间复杂度为 O(n^2+nlogn)=(n^2),空间复杂度是 O(n),代码如下.注意，在这里为了避免重复结果，对于已经判断过的数会 skip 掉，这也是排序带来的方便。 这道题考察的点其实和 Two Sum 差不多，Two Sum 是 3Sum 的一个 subroutine, 不过更加综合一些，实现上更加全面，需要注意细节，面试中比较常见的一道题。此题更加复杂的扩展是 4Sum，请参见 4Sum – LeetCode。

```
public ArrayList<ArrayList<Integer>> threeSum(int[] num)
{
   ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
   if(num==null || num.length<=2)
      return res;
   Arrays.sort(num); //Tao: 不要忘了 sort! 自己寫的時候老是忘
   for(int i=num.length-1;i>=2;i--)
   {
      if(i<num.length-1 && num[i]==num[i+1])
         continue;
       ArrayList<ArrayList<Integer>> curRes = twoSum(num,i-1,-num[i]);
       for(int j=0;j<curRes.size();j++)
       {
          curRes.get(j).add(num[i]);
       }
       res.addAll(curRes);
   }
   return res;
}

//Tao: 以下是 twoSum 函數, 要求輸入的數組是 sorted, 然後用兩頭夾逼法. 此人(Code Ganker)對 leetcode 中 twoSum 這題的解法與下面是一樣的(稍有改動), 也先 sort 了一下, 所以前面我對 twoSum 一題中用的其它解法. 3Sum 的解法中, 絶大多數都用了 sort.
private ArrayList<ArrayList<Integer>> twoSum(int[] num, int end,int target)
{
   ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
   if(num==null || num.length<=1)
      return res;
   int l = 0;
   int r = end;
   while(l<r)
   {
      if(num[l]+num[r]==target)
      {
         ArrayList<Integer> item = new ArrayList<Integer>();
         item.add(num[l]);
         item.add(num[r]);
         res.add(item);
         l++;
         r--;
```

```
            while(l<r&&num[l]==num[l-1])
                l++;
            while(l<r&&num[r]==num[r+1])
                r--;
        }
        else if(num[l]+num[r]>target)
        {
            r--; //Tao: 為何此處不 l--? 因為 num[l-1] + num[r+1]已經等於 target 了故 num[l-1] + num[r]還是會>target
        }
        else
        {
            l++;
        }
    }
    return res;
}
```

---

16. 3Sum Closest, Medium

http://blog.csdn.net/linhuanmars/article/details/19712011

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

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

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
Subscribe to see which companies asked this question

Key: 用 twoSum 函數. twoSum 函數要求輸入的數組是 sorted, 然後用兩頭夾逼法. twoSum(int[] num, int target, int start)返回 closest. 在 threeSum 中調用 twoSum(num,target-num[i],i+1). 本題不用跳過相同的元素, 原因不清楚, 可能是 OJ 對運行的時間要求要松點.
當然本題還可以像前面的題那樣, twoSum 寫成 twoSum(int[] nums, int end, int target), 然後在 threeSum 中調用 twoSum(nums, i - 1, target – nums[i]). E3 就是這樣寫的, 代碼在最後.

History:
E1 直接看的答案.
E2 通過.
E3 改了幾次後通過.

这道题跟 3Sum 很类似，区别就是要维护一个最小的 diff，求出和目标最近的三个和。brute force 时间复杂度为 O(n^3)，优化的解法是使用排序之后夹逼的方法，总的时间复杂度为 O(n^2+nlogn)=(n^2),空间复杂度是 O(n),代码如下.这道题具体的考察点可以参见 3Sum，稍微变体一下，其实区别不大。此题更加复杂的扩展是 4Sum，请参见 4Sum – LeetCode.

```
public int threeSumClosest(int[] num, int target) {
```

```
    if(num == null || num.length<=2)
        return Integer.MIN_VALUE;
    Arrays.sort(num); //Tao: 不要忘了 sort! 自己寫的時候老是忘
    int closest = num[0]+num[1]+num[2]-target;
    for(int i=0;i<num.length-2;i++) // Tao: 為何是 num.length-2? 見下頁注釋. E2 的代碼是從後往前掃描的(即從 num.length-1 到 0), 可能好懂點.
    {
        int cur = twoSum(num,target-num[i],i+1); //Tao: 為何是 i+1(即 num[i]右邊的), 而不是 num[i]左邊的? 因為沒必要考慮左邊的, 比如***1***2**3**, 若此時要選中 123, 則當 num[i]為 1 時找出它右邊的 23 就可以了, 沒必要 num[i]為 2 時還往左邊找
        if(Math.abs(cur)<Math.abs(closest))
            closest = cur;
    }
    return target+closest;
}
//Tao: 以下是 twoSum 函數, 要求輸入的數組是 sorted, 然後用兩頭夾逼法.
private int twoSum(int[] num, int target, int start)
{
    int closest = num[start]+num[start+1]-target; //Tao: start+1 不會超出數組大小, 因為 threeSum 中的 for(int i=0;i<num.length-2;i++)中已保證 i 最大為 num.length-3
    int l = start;
    int r = num.length-1;
    while(l<r)
    {
        if(num[l]+num[r]==target)
            return 0;
        int diff = num[l]+num[r]-target; //Tao: 這裡又在循環中定義 int, 看來這樣應該沒事
        if(Math.abs(diff)<Math.abs(closest))
            closest = diff;
        if(num[l]+num[r]>target)
        {
            r--;
        }
        else
        {
            l++;
        }
    }
    return closest;
}
```

E3 的代碼:

```
    public int threeSumClosest(int[] nums, int target) {
        int minDiffAbs = Integer.MAX_VALUE;
        int minDiff = 0;
        Arrays.sort(nums);

        for(int i = 2; i < nums.length; i++) {
            int diff = twoSum(nums, i - 1, target - nums[i]);
```

```
            if(Math.abs(diff) < minDiffAbs) {
                minDiffAbs = Math.abs(diff);
                minDiff = diff;
            }
        }

        return (target - minDiff);
    }

    private int twoSum(int[] nums, int end, int target) {
        int l = 0, r = end;
        int minDiffAbs = Integer.MAX_VALUE;
        int minDiff = 0;

        while(l < r) {
            int sum = nums[l] + nums[r];

            if(sum == target) {
                return 0;
            } else if(sum < target) {
                l++;
            } else {
                r--;
            }

            int diff = target - sum;
            int diffAbs = Math.abs(diff);
            if(diffAbs < minDiffAbs) {
                minDiffAbs = diffAbs;
                minDiff = diff;
            }
        }
        return minDiff;
    }
```

17. Letter Combinations of a Phone Number, Medium

http://blog.csdn.net/linhuanmars/article/details/19743197

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.
Subscribe to see which companies asked this question

Key: 本題沒必要用遞歸, 用循環即可. for(int i = 0; i < digits.length(); i++), 對第 i 輪循環, 將上一輪的結果(即 List<String> res)中的每一個 String 都拿出來重新加工, 將 digits[i]所對應的字母都加到 '上一輪的結果中的每一個 String(即 res)' 後面, 然後將 res 更新為本輪的. 用一個 map 來表示手機鍵盤.

History:
E1 直接看的答案.
E2 沒看 key, 直接寫的(NB!). 最開始按遞歸寫的, 後來發現沒必要用遞歸, 改用迭代, 寫出來後, 本來可以一次通過, 但忘了初始時在 res 中加一個空的 String, 導致整個結果為空. 改後即通過. 後來發現我的代碼跟以下答案基本一樣, 所有地方都不謀而合. E2 重寫了 key. 後來發現 E2 是用 map 來表示手機鍵盤的, 比以下答案要好, 所以以後用 E2 的代碼(在最後).
E3 用的遞歸, 本來可以一次通過的, 結果不小心將 substring(0, n - 1)寫成了 substring(n - 1). 十來天沒用 Java(用 C 語言寫操作係統 project3 去了), Java 又不熟練了(但我覺得還是很熟練啊, 可能不是因為這個原因, 可能只是因為粗心). 改後即通過.

这道题目和求组合的思路差不多，比较简单。依次读取数字，然后把数字可以代表的字符依次加到当前的所有结果中，然后进入下一次迭代。假设总共有 n 个 digit，每个 digit 可以代表 k 个字符，那么时间复杂度是 O(k^n) (可用數學規納法求)，就是结果的数量，空间复杂度也是一样。代码如下 这道题个人觉得没有太多算法和数据结构的思想，但是自己在 facebook 的面试中遇到了，所以还是要重视一下，就是一些数组操作的小细节。相关的扩展是这道题如何用递归来解，思路其实类似，就是对于当前字符，递归剩下的数字串，然后得到结果后加上自己返回回去，大家可以试试。如果有问题，欢迎留言讨论哈。
Tao: 若輸入為"23", 則以下我加的三個 System.out.println()輸出結果為:
initial: 1
res.size()=1
newRes=[a]
newRes=[a, b]
newRes=[a, b, c]
res.size()=3
newRes=[ad]
newRes=[ad, ae]
newRes=[ad, ae, af]
newRes=[ad, ae, af, bd]
newRes=[ad, ae, af, bd, be]
newRes=[ad, ae, af, bd, be, bf]
newRes=[ad, ae, af, bd, be, bf, cd]
newRes=[ad, ae, af, bd, be, bf, cd, ce]

newRes=[ad, ae, af, bd, be, bf, cd, ce, cf]

Code Ganker 的代碼(不用):

```
public ArrayList<String> letterCombinations(String digits) {
    ArrayList<String> res = new ArrayList<String>();
    if(digits==null || digits.length()==0)
        return res; // tao: 在原答案中, 這兩句在 res.add("") 之後. 原答案在 leetcode OJ 中被判為: Wrong Answer. Input: “”, output: [“”], expected: []. 我作此改動後就 accept 了.
        res.add(""); //tao: 此時 res.size()已為 1 (見前面的輸出結果 initial), 否則後面的 for(int j=0;j<res.size();j++)循環運行不了
//System.out.println("initial: "+res.size()); //Added by Tao

    for(int i=0;i<digits.length();i++)
    {
        String letters = getLetters(digits.charAt(i));
        ArrayList<String> newRes = new ArrayList<String>();
        // System.out.println("res.size()="+res.size()); // Added by Tao
        for(int j=0;j<res.size();j++)
        {
            for(int k=0;k<letters.length();k++)
            {
                newRes.add(res.get(j)+Character.toString(letters.charAt(k)));
              //System.out.println("newRes="+newRes); //Added by Tao
            }
        }
        res = newRes;
    }
    return res;
}
private String getLetters(char digit)
{
    switch(digit)
    {
        case '2':
            return "abc";
        case '3':
            return "def";
        case '4':
            return "ghi";
        case '5':
            return "jkl";
        case '6':
            return "mno";
        case '7':
            return "pqrs";
        case '8':
            return "tuv";
```

```
        case '9':
            return "wxyz";
        case '0':
            return " ";
        default:
            return "";
    }
}
```

E2 的代碼:

```
    public static List<String> letterCombinations(String digits) {
        List<String> res = new ArrayList<>();
        if(digits == null || digits.length() == 0) return res;

        Map<Character, String> map = new HashMap<>();
        map.put('2', "abc");
        map.put('3', "def");
        map.put('4', "ghi");
        map.put('5', "jkl");
        map.put('6', "mno");
        map.put('7', "pqrs");
        map.put('8', "tuv");
        map.put('9', "wxyz");

        res.add("");

        for(int i = 0; i < digits.length(); i++) {
            char digit = digits.charAt(i);
            char[] letters = map.get(digit).toCharArray();
            List<String> newRes = new ArrayList<>();

            for(String s : res) {
                StringBuilder sb = new StringBuilder(s);
                for(char c : letters) {
                    sb.append(c);
                    newRes.add(new String(sb.toString()));
                    sb.delete(sb.length() - 1, sb.length());
                }
            }

            res = newRes;
        }

        return res;
    }
```

---

18. 4Sum, Medium

http://blog.csdn.net/linhuanmars/article/details/24826871

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

- Elements in a quadruplet (*a*,*b*,*c*,*d*) must be in non-descending order. (ie, $a \le b \le c \le d$)
- The solution set must not contain duplicate quadruplets.

For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is:
(-1, 0, 0, 1)
(-2, -1, 1, 2)
(-2, 0, 0, 2)
Subscribe to see which companies asked this question

Key: 思想跟 3Sum 一樣. 調用 3Sum 和 2Sum. 本題也要跳過相同元素. twoSum 函數跟前面第 15 題 3Sum 中調用的 twoSum 是一模一樣的. 本題的 threeSum 函數 跟 本題的 fourSum 函數 基本上一模一樣. 本題算是 NSum 這類題目的一個總結性的復習.

History:
E1 直接看的答案.
E2 的 twoSum 中 跳過重復元素 這一點沒弄好(詳見以下代碼中), 所以沒寫出來.
E3 是將之前題目中(我寫的)的代碼拷過來改的, 幾分鍾改好, 一次通過.

这道题要求跟 3Sum 差不多，只是需求扩展到四个的数字的和了。我们还是可以按照 3Sum 中的解法，只是在外面套一层循环，相当于求 n 次 3Sum。我们知道 3Sum 的时间复杂度是 O(n^2)，所以如果这样解的总时间复杂度是 O(n^3)。代码如下：

```
public ArrayList<ArrayList<Integer>> fourSum(int[] num, int target) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num==null||num.length==0)
        return res;
    Arrays.sort(num);
    for(int i=num.length-1;i>2;i--) //Tao: 此處要求 i>2 是為了保證後面調用 twoSum(num,i-1,target-num[i])時, i-1>=2, 即能有至少三個元素可以和 target-num[i]比較
    {
        if(i==num.length-1 || num[i]!=num[i+1])
        {
            ArrayList<ArrayList<Integer>> curRes = threeSum(num,i-1,target-num[i]);
            for(int j=0;j<curRes.size();j++)
            {
                curRes.get(j).add(num[i]);
            }
            res.addAll(curRes);
        }
    }
    return res;
}
```

```
//Tao: 以下是 threeSum 函數, 跟上面的 fourSum 函數基本上一模一樣
private ArrayList<ArrayList<Integer>> threeSum(int[] num, int end, int target)
{
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    for(int i=end;i>1;i--) //Tao: 此處要求 i>1 是為了保證後面調用 twoSum(num,i-1,target-num[i])時, i-1>=1, 即能有至少兩個元素可以和 target-num[i]比較
    {
        if(i==end || num[i]!=num[i+1])
        {
            ArrayList<ArrayList<Integer>> curRes = twoSum(num,i-1,target-num[i]);
            for(int j=0;j<curRes.size();j++)
            {
                curRes.get(j).add(num[i]);
            }
            res.addAll(curRes);
        }
    }
    return res;
}

//Tao: 以下是 twoSum 函數, 跟前面第 15 題 3Sum 中調用的 twoSum 是一模一樣的
private ArrayList<ArrayList<Integer>> twoSum(int[] num, int end, int target)
{
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    int l=0;
    int r=end;
    while(l<r)
    {
        if(num[l]+num[r]==target)
        {
            ArrayList<Integer> item = new ArrayList<Integer>();
            item.add(num[l]);
            item.add(num[r]);
            res.add(item);
            l++;
            r--;
            //E2 將以下兩個 while 放在了前面, 結果死活沒搞出來.
            while(l<r&&num[l]==num[l-1])
                l++;
            while(l<r&&num[r]==num[r+1])
                r--;
        }
        else if(num[l]+num[r]>target)
        {
            r--;
        }
        else
        {
```

```
            l++;
        }
    }
    return res;
}
```

19. Remove Nth Node From End of List, Easy

Given a linked list, remove the *n*th node from the end of list and return its head.
For example,
Given linked list: **1->2->3->4->5**, and ***n* = 2**.

After removing the second node from the end, the linked list becomes **1->2->3->5**.
Note:
Given *n* will always be valid.
Try to do this in one pass.
Subscribe to see which companies asked this question

Key: Two pointers. pre 和 cur 都從 fakeHead 出發, cur 先走 n 步, 然後兩人一起走, 等 cur 走到末尾時, pre 所在的位置就是我們要找的了.

History:
E1 是先用循環找出 list 的長度, 所以不是 one pass 的(注意題目要 one pass).
E2 用標准的 two pointers 方法, 很快寫好, 本來可以一次通過的, 但粗心將 return fakeHead 寫成了 return head, 改後即通過.
E3 較快寫好, 一次通過, 方法跟 key 一樣.

Tao: 以下用我 E2 的代碼, 方法跟 William 和 Code Ganker 的一樣, 只是我的代碼看起來要好懂清新些, Code Ganker 沒用 fakeHead, 但也把刪首節點這個情況處理得很好. William 用了 fakeHead. 注意這個方法中, pre 和 cur 走的步數總共就是 list 長度, 所以可以算是 one pass.

Code Ganker 說得還可以, 可以看看:

这道题是链表基本操作，主要问题就是如何得到链表的倒数第 n 个结点，应该是一个比较重要的 routine，因为很多题目都要用这个 subroutine。思路就是先用一个 runner 指针走 n 步，然后再来一个 walker 从头开始和 runner 同时向后走，当 runner 走到链表末尾的时候，walker 指针即为倒数第 n 个结点。算法的时间复杂度是 O(链表的长度)，空间复杂度是 O(1)。原题中假设 n 总是合法的，但是面试的时候最好不要做这种假设，或者至少要和面试官讨论一下。以上代码是如果 n 不合法，就直接返回链表头，也就是什么都不做。对于这种 corner case，大家还是尽量考虑，因为题目比较简单，更多的只能去关注这些问题。

以下是 E2 的代碼:

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    if(head == null)
        return head;
```

```
    ListNode fakeHead = new ListNode(0);
    fakeHead.next = head;

    ListNode pre = fakeHead;
    ListNode cur = fakeHead;

    for(int i = 0; i < n; i++)
        cur = cur.next;

    while(cur.next != null) {
        pre = pre.next;
        cur = cur.next;
    }

    //if 是考慮到 n=0 之情況
    if(pre.next != null)
        pre.next = pre.next.next;

    return fakeHead.next;
}
```

---

20. Valid Parentheses, Easy

http://www.raychase.net/2639#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. Subscribe to see which companies asked this question

Key: 用栈来记录前半个括号的情况

Corner case: (, ((, [)), ([])

History:
E1 直接看的答案.
E2 通過, 但被 corner case 檔了好幾次.
E3 幾分鍾寫好, 一次通過. E3 的方法跟以下答案一樣, 但代碼要簡潔些, 以後用 E3 的代碼.

用栈来记录前半个括号的情况，注意不同情况下如果栈为空的处理：

以下答案代碼不用:

```
public class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (int i=0; i<s.length(); i++) {
            char ch = s.charAt(i);

            if (ch=='(' || ch=='[' || ch=='{')
                stack.push(ch);

            if (ch==')' && (stack.empty() || stack.pop()!='('))
                return false;

            if (ch==']' && (stack.empty() || stack.pop()!='[')) //tao: 對於[()]之情況, 當 ch=[ 時, ( 已經在上一個 if 中 pop 出來了, 故此時看到的還是[]
                return false;

            if (ch=='}' && (stack.empty() || stack.pop()!='{'))
                return false;

        } // for

//E2 中, 我將以下三句寫成了一句: return stack.isEmpty(), 也通過了.
        if (!stack.empty())
            return false;

        return true; //tao: 前面的 if 返回都是 false, 如果那些 if 都沒通過, 就返回 true 了. 此句同時也考慮到了 s 長度為 0 之情況.
    }
}
```

E3 的代碼:

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

        for(int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if(c == '(' || c == '[' || c == '{') {
                stack.push(c);
```

```
        } else {
            if(stack.isEmpty()) return false; //防止 ()) 情況
            char l = stack.pop();
            if((l == '(' && c == ')') || (l == '[' && c == ']') || (l == '{' && c == '}')) continue;
            else return false; //防止 ([)] 情況
        }
    }

    return stack.isEmpty(); //防止 (() 情況
}
```

21. Merge Two Sorted Lists, Easy

http://blog.csdn.net/linhuanmars/article/details/19712593

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.
Subscribe to see which companies asked this question

Key: splice: 接合. 直接在 l1 中插入 l2 的 node. 用了一個 pre 指針, 從 l1 的開頭開始掃描, 當 pre 的下一個數大於 l2 的值時(當然 l2 也要在自己的 list 中掃描), 就將 l2 的 node 插到 pre 後面去.

Corner case: l2 中有些 node 要加到 l1 頭元素之前, 有些要加到 l1 尾元素之後.

注意在我電腦上不用寫 ListNode 這個 class, 每個 List 題目的文件夾中都已經有 ListNode.class 了.

History:
E1 通過, 但不是 in-place 的.
E2 本來可以一次通過的, 但某處少寫了一句 pre = pre.next, 添了後才通過, 代碼基本上和 Code Ganker 是樣的.
E3 很快寫好, 但不小心將 while 中的 l2 !=null 寫成了 l2.next != null (見代碼中), 結果對於 l2 只有一個元素的情況沒通過. 改後即通過.

Tao:

我的代碼也能通過, 我的代碼是先創建一個新的 list, l1 和 l2 分別拿出一個元素來比，小的就放入新的 list 裡面。但這不是 inplace 的, 所以不用我的代碼(注意題目中要求是直接 splice nodes). Code Ganker 的代碼是 inplaced, 但他的原代碼移動了 l1 和 pre 兩個指針, 稍微有點囉嗦, 我將 l1 改為了 pre.next, 而只移動 pre 一個指針, 要簡潔些, 以下用我改进後的 Code Ganker 代碼.

Tao 的感想: 本 note 最初是用的我的代碼, 因為我看了半天都看不懂 Code Ganker 的代碼. 而等我刷題刷到 100 多題時, 再回過來看, 不僅看得懂 Code Ganker 的代碼了, 而且還對它进行了改进. 看來刷題真的能提高水平.

第 148. Sort List 題中, Code Ganker 解法後的評論(那裡也有的 merge sort, 實際上就是調用本題代碼):

回复 linhuanmars：我生成了一个新的头，然后把左右链表接到新的头上面，最后也 AC 了，不知道算不算常数空间复杂度呢？

回复 qweyxy_2：恩，这样还是常数的，只不过接的时候多点操作而已~ 数量级没变哈~

Code Ganker:

这道题目比较简单，经典的链表基本操作。维护两个指针对应两个链表，因为一般会以一条链表为基准，比如说 l1, 那么如果 l1 当期那的元素比较小，那么直接移动 l1 即可，否则将 l2 当前的元素插入到 l1 当前元素的前面。算法时间复杂度是 O(m+n) (我: 注意標准的 merge sort 算法的時間複雜度為 O(n logn), CLRS p171),m 和 n 分别是两条链表的长度，空间复杂度是 O(1)，代码如下.

这个题类似的有 Merge Sorted Array，只是后者是对数组进行合并操作，面试中可能会一起问到。扩展题目 Merge k Sorted Lists, 这是一个在分布式系统中比较有用的基本操作，还是需要重视，面试中可以发散出很多问题。

Code Ganker: 这里是有构造一个 helper 的虚指针的，一般来说需要改动到链表头的题目就会构造虚表头，l1 和 l2 是第一个结点，不过后来会被改变，往后迭代. helper 是一种链表比较常见的技巧，就是在链表头构造一个空节点，这样是有利于链表操作中需要改动链表头时不需要分情况讨论，可以多看几道链表的题目就明白了哈 ~

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode helper = new ListNode(0);
    helper.next = l1;
    ListNode pre = helper;

    while(pre.next !=null && l2 != null) //E3 最開始將&&後寫成了 l2.next != null.
    {
        if(pre.next.val>l2.val)
        {
            ListNode next = l2.next;
            //以下兩句是將 l2 這個 node 插到 pre 後面
            l2.next = pre.next;
            pre.next = l2;

            l2 = next;
        }
```

```
        pre = pre.next;


    }
    //以下是當 l1 這個 list 被掃描完了後(pre.next ==null), 若 l2 還有剩餘, 則將 l2 剩下那段全部接到 l1 後面. 注意此時 l2 剩下的元素都比 l1 尾元素大.
    if(l2!=null)
    {
        pre.next = l2;  //注意是直接將 l2 剩下的那段接到 l2 後面. E2 是一個一個 node 接到 l1 後的, 沒必要.
    }
    return helper.next;
}
```

---

22. Generate Parentheses, Medium

http://www.cnblogs.com/springfor/p/3886559.html

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:
"((()))", "(()())", "(())()", "()(())", "()()()"
Subscribe to see which companies asked this question

Key: 遞歸. 寫一個 dfs(ArrayList<String> res, String item, int left, int right), 其作用為: 將 left 個左括號 和 right 個右括號 弄成合法的 string(合法的意思是 string 中的左括號個數 >= 右括號個數), 加到 item 中, 若加完, 則將 item 放入 res 中.
在 dfs 函數中:
```
if (left>0)  dfs(res,item+'(',left-1,right);
if (right>0)  dfs(res,item+')',left,right-1);
```
且剩下的左括號的數量不能大於右括號(即已 string 中的左括號個數要 >= 右括號個數, 如不能生成")("): if(left > right) return; 注意 left 是指剩下的左括號數. 此要求是用來確保生成的組合是合法的. 注意本方法不用保護現場, 因為上面的 item+'('並沒改變 item, 注意它跟 item.apped("(") 之區別.

History:
E1 直接看的答案.

E2 很快寫好, 但范了個小錯, 改後即通過. 這個小錯就是, 我將 item 弄成的一個 StringBuilder, 而在 dfs 中弄的 item.apped("("). 而答案中的 item 是一個 String, 而在 dfs 中的是 item+'('. 注意它們之間是有區別的: item.apped("(")改變了 item, 而 item+'('沒改變 item.

E3 沒多久就寫好, 一次通過. 但 E3 的方法較麻煩, E3 寫了一個 StringNum 這麼個 class, 其中包含現有的括號 string, 如"(())(", 以及這個 string 中的 左括號個數 和 右括號個數. 然後寫一個 helper(len, n)來產生所有的

合法的 string(合法的意思是左括號個數 >= 右括號個數), 在 helper 中遞歸調用 helper(len - 1, n). 這樣雖然通過, 但寫起來的代碼就較長(但也不是很長, 注意我一次通過). 主要還是我沒想到 key 中那樣, 用 dfs(...left, right) 來將 left 個左括號和 right 個右括號 弄成合法的 string, 這點很重要, 要記住.

(以下"左括号的数不能"有點 confusing/misleading, 不看)
这道题跟 unique binary tree ii 是类似的。如果是只求个数的话是类似 unique binary tree，用到了卡特兰数。
这里也是用到了类似的模型。
不过这道题按照 DFS 那种递归想法解决还是比较容易想到的。
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。

```
public ArrayList<String> generateParenthesis(int n) {
        ArrayList<String> res = new ArrayList<String>();
        String item = new String();

        if (n<=0)
            return res;

        dfs(res,item,n,n);  //res 已經在 dsf()中被賦值了. Java 的函數原來也有 Fortran 裡 subroutine 的功能! 能這樣做是因為 res 是 reference.
        return res;
    }

//以下代碼可考慮 left = right = 2 時之情況,  首先會運行下面 A 行的 dfs(res,item+'(',1, 2) 會產生(()) 和()() 兩個結果. 然後會運行 面 B 行的 dfs(res,item+')', 2, 1), 這就是 left > right 時之情況, 要丟掉.

    public void dfs(ArrayList<String> res, String item, int left, int right){ //tao: left 是左括號的數量, right 是右括號的數量
        if(left > right) //deal with ")("
            return;

        if (left == 0 && right == 0){
            res.add(new String(item));
            return;
        }

        if (left>0)
            dfs(res,item+'(',left-1,right);  //A 行
          //tao:先不停地通過上面和下面這兩個 dsf()走到底(即 left=right=0 時), 最後才由上面的 if(left == 0 && right ==0)來 return. 這就是 DFS 之思想. 注意這兩個 dsf()後都沒跟 return 語句.
        if (right>0)
            dfs(res,item+')',left,right-1);  //B 行
    }
```

23. Merge k Sorted Lists, Hard

http://blog.csdn.net/linhuanmars/article/details/19899259

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.
Subscribe to see which companies asked this question

key: 遞歸. 寫一個 helper(ListNode[] lists, int l, int r), 其作用是將 ListNode[l]到 listNode[r] inclusive 這些 lists merge 好, 並返回 merge 後的 list 首元素. helper 中調用自己, 來每次 merge lists 中的前一半和後一半, 這兩半 merge 後的兩個 list, 用 21. Merge Two Sorted Lists 中我的 mergeTwoLists 函數來 merge 好. 另外, 本題的複雜度應該也是個考點, 不要忽略.

History:
E1 用 brute force 超時了.
E2 一次通過, 而且是在 E1 的 key 寫得不清楚的情況下做出來的, NB.
E3 很快寫好, 一次通過. 方法跟 key 中一模一樣. 我寫的時候其實已經對 key 沒甚麼映像了, 該方法基本上是我自己獨立想出來的, 由於很快就寫好了(mergeTwoLists(list1, list2)函數是調用之前的題目中的), 所以不相信 hard 的題目會這麼簡單, 覺得此方法應該會超時, 結果代碼帖到 OJ 中一試, 居然一次就通過了. 但我 E3 將複雜度算錯了.

Tao: 我試過 brute force 的方法：for(int i = 1; i < lists.length; i++) res = mergeTwoLists(res, lists[i]), 但在 leetcode OJ 中是 time limit exceeded. 注意以下 Code Ganker 的代碼中，我稍做了修改：將 mergeKLists()和 helper()的參數 lists 類型由他的 ArrayList 改成了 leetcode 中建議的 ListNode[]，函數內也作了相應的小修改. 改後仍是 accepted.

这道题目在分布式系统中非常常见，来自不同 client 的 sorted list 要在 central server 上面 merge 起来。这个题目一般有两种做法，下面一一介绍并且分析复杂度。 第一种做法比较容易想到，就是有点类似于 MergeSort 的思路,就是分治法，不了解 MergeSort 的朋友，请参见归并排序-维基百科，是一个比较经典的 O(nlogn)的排序算法，还是比较重要的。思路是先分成两个子任务，然后递归求子任务，最后回溯回来。这个题目也是这样，先把 k 个 list 分成两半，然后继续划分，知道剩下两个 list 就合并起来，合并时会用到 Merge Two Sorted Lists 这道题，不熟悉的朋友可以复习一下。代码如下。

我们来分析一下上述算法的时间复杂度。假设总共有 k 个 list，每个 list 的最大长度是 n，那么运行时间满足递推式 T(k) = 2T(k/2)+O(n*k)。(O(n*k)是因為 mergeTwoLists(helper(lists, l, m), helper(lists, m + 1, r))外層調用了 mergeTwoLists) 根据主定理，可以算出算法的总复杂度是 O(nklogk) (我: 原因見下面 CLRS 截圖的最後一段, 此即 case 2, 故時間複雜度為 Θ(f(n) lg n), 即 O(nklogk))。如果不了解主定理的朋友，可以参见主定理-维基百科。

我: From CLRS p115:

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

空间复杂度的话是递归栈的大小 O(logk)。Tao: 每個遞歸棧都是從那個 binary tree 的頂到底的單根線，故長度為 binary tree 之高度 O(logk)。每一個遞歸棧運行完後就清空了(往回由底到頂走的時候會 pop)，空間留給下一條線。

```
public static ListNode mergeKLists(ListNode[] lists) {
  if(lists == null || lists.length == 0)
   return null;

  return helper(lists, 0, lists.length - 1);
 }

 public static ListNode helper(ListNode[] lists, int l, int r) {
  if(l < r) {
   int m = (l + r) / 2;
   return mergeTwoLists(helper(lists, l, m), helper(lists, m + 1, r));
  }
   return lists[l];
 }

 public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
 // Tao: 將前面 21. Merge Two Sorted Lists 中我的 mergeTwoLists 代碼抄過來
 }
```

接下来我们来看第二种方法。这种方法用到了堆的数据结构，思路比较难想到，但是其实原理比较简单。维护一个大小为 k 的堆，每次取堆顶的最小元素放到结果中，然后读取该元素的下一个元素放入堆中，重新维

护好。因为每个链表是有序的，每次又是去当前 k 个元素中最小的，所以当所有链表都读完时结束，这个时候所有元素按从小到大放在结果链表中。这个算法每个元素要读取一次，即是 k*n 次，然后每次读取元素要把新元素插入堆中要 logk 的复杂度，所以总时间复杂度是 O(nklogk)。空间复杂度是堆的大小，即为 O(k)。代码如下(我: 已省略).

可以看出两种方法有着同样的时间复杂度，都是可以接受的解法，但是却代表了两种不同的思路，数据结构也不用。个人觉得两种方法都掌握会比较好哈，因为在实际中比较有应用，所以也是比较常考的题目。

---

24. Swap Nodes in Pairs, Medium

http://www.cnblogs.com/springfor/p/3862030.html

Given a linked list, swap every two adjacent nodes and return its head.
For example,
Given 1->2->3->4, you should return the list as 2->1->4->3.
Your algorithm should use only constant space. You may **not** modify the values in the list, only nodes itself can be changed.
Subscribe to see which companies asked this question

Key: 寫一個函數 void swap(ListNode left, ListNode right), 其作用為交換 left.next 和 right, 例如 1->2(left)->3->4(right)->5, 調用此函數後成為 1->2->4->3->5. 然後在主函數 swapPairs 中調用 swap 函數. 注意 swap 在調用前, 已假定這樣一個不變量: left 和 right 之間隔一個 node. 所以每次調用 swap 前, 要先將這個不變量弄好.

History:
E1 直接看的答案.
E2 通過, 但改了幾次, 因為是我自己多用了一個指針來表示下一對要交換的 pair 的第一個 node, 並用這個指針來控製循環, 結果反而多出了 test case, 給弄複雜了.
E3 的思想跟以下的答案差不多, 但俱體實現有點區別, E3 的方法更好懂, 所以以後就用 E3 的代碼, key 是 E3 寫的. E3 最初寫的時候, 沒將調用 swap 前需要的那個不變量弄好(詳細解釋見代碼中), 改後即通過.

这道题考察了基本的链表操作，注意当改变指针连接时，要用一个临时指针指向原来的 next 值，否则链表丢链，无法找到下一个值。

本题的解题方法是：
我: 用两个指针，ptr1 始终指向需要交换的 pair 的前面一个 node，ptr2 始终指向需要交换的 pair 的第一个 node, 例如 2134 中, 要交换 34 時, 有 21(ptr1)3(ptr2)4.

需要运用 fakehead(以後不要叫 falseHead, 但 denture 的 wiki 中確實說又叫 false teeth)来指向原指针头，防止丢链，用两个指针，ptr1 始终指向需要交换的 pair 的前面一个 node，ptr2 始终指向需要交换的 pair 的第一个 node。

然后就是进行链表交换。

需要用一个临时指针 nextstart， 指向下一个需要交换的 pair 的第一个 node，保证下一次交换的正确进行。

然后就进行正常的链表交换，和指针挪动就好。

当链表长度为奇数时，ptr2.next 可能为 null；

当链表长度为偶数时，ptr2 可能为 null。

所以把这两个情况作为终止条件，在 while 判断就好，最后返回 fakehead.next。

代码如下：

Code Ganker 的方法也與以下的代碼相似，這裡就不列出來了。但 Code Ganker 有兩句話說得好：这道题中用了一个辅助指针作为表头，这是链表中比较常用的小技巧，因为这样可以避免处理 head 的边界情况，一般来说要求的结果表头会有变化的会经常用这个技巧，大家以后会经常遇到。
因为这是一遍过的算法，时间复杂度明显是 O(n)，空间复杂度是 O(1)。实现中注意细节就可以了，不过我发现现在面试中链表操作的题目出现并不多，所以个人觉得大家练一下就好了，不用花太多时间哈。

```
//以下代碼不用
//Tao:設輸入為 1 → 2 → 3 → 4
public ListNode swapPairs(ListNode head) {
    if(head == null || head.next == null)
      return head;

    ListNode fakehead = new ListNode(-1);
    fakehead.next = head; //Tao: -1(fakehead) → 1 (head) → 2 → 3 → 4

    ListNode ptr1 = fakehead;
    ListNode ptr2 = head; //Tao: -1(fakehead, ptr1) → 1 (head, ptr2) → 2 → 3 → 4

    while(ptr2!=null && ptr2.next!=null){//Tao: 記住 while 中是 pt2. 看下一句就記住了。
       ListNode nextstart = ptr2.next.next;
       //Tao:以上使得: -1(fakehead, ptr1) → 1 (head, ptr2) → 2 → 3 (nextstart) → 4
       ptr2.next.next = ptr2;
       //Tao: 以上使得: -1(ptr1) → 1 (ptr2) → 2 → (指回前面那個 1, 形成一個環)
       //Tao: 另一條為 3 (nextstart) → 4
       ptr1.next = ptr2.next;
       //Tao: 以上使得: -1(ptr1) →  2 → 1(ptr2) → (not interested)
       //Tao: 另一條為 3 (nextstart) → 4
```

```
        ptr2.next = nextstart;
        //Tao: 以上使得: -1(ptr1) →  2 → 1(ptr2) → 3 (nextstart) → 4
        ptr1 = ptr2;
        //Tao: 以上使得: -1→ 2 → 1 (ptr1, ptr2) → 3 (nextstart)→ 4
        ptr2 = ptr2.next;
        //Tao: 以上使得: -1→ 2 → 1 (ptr1) → 3 (nextstart, ptr2)→ 4
      }
    return fakehead.next;
  }
```

E3 的代碼:

```
    public ListNode swapPairs(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode fakeHead = new ListNode(0);
        fakeHead.next = head;
        ListNode left = fakeHead, right = head.next;

        while(true) {
            swap(left, right);
            right = right.next; //E3 最初沒有這句, 給出錯誤結果. 此句理解起來並不難, 先看 swap 函數前的黃字. 若調用 swap 前為 1->2(left)->3->4(right)->5, 調用 swap 後即成為 1->2(left)->4(right)->3->5, 若直接運行以下的兩句(即將 left 和 right 往右移兩位的那兩句), 則下一輪循環後, left 和 right 之間沒間隔一個 node, 這不滿足 swap 運行前的不變量, 所以這裡要先將 right 往右移一位.
            if(right.next == null || right.next.next == null) break;
            left = left.next.next;
            right = right.next.next;
        }

        return fakeHead.next;
    }

//swap(left, right)之作用為交換 left.next 和 right, 例如 1->2(left)->3->4(right)->5, 調用此函數後成為 1->2->4->3->5. 注意 swap 在調用前, 已假定這樣一個不變量: left 和 right 之間隔一個 node. 所以每次調用 swap 前, 要先將這個不變量弄好.
    private void swap(ListNode left, ListNode right) {
        ListNode leftNext = left.next, rightNext = right.next;
        left.next = right;
        right.next = leftNext;
        leftNext.next = rightNext;
    }
```

---

25. Reverse Nodes in k-Group, Hard

http://www.cnblogs.com/springfor/p/3864530.html

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.
If the number of nodes is not a multiple of *k* then left-out nodes in the end should remain as it is.
You may not alter the values in the nodes, only nodes itself may be changed.
Only constant memory is allowed.
For example,
Given this linked list: 1->2->3->4->5
For *k* = 2, you should return: 2->1->4->3->5
For *k* = 3, you should return: 3->2->1->4->5
Subscribe to see which companies asked this question

Key: 題目的意思是每次都要反轉 k 個, 直到剩下的元素個數不够 k 個, 如題目中的 For k = 2, you should return: 2->1->4->3->5, 即 1->2 反轉了, 3->4 也要反轉, 而剩下的只有一個 5, 不够反轉, 所以就不繼續反轉了. 另外, 當 k 大於 list 長度時, 則甚麼都不做, 返回原 list (尼瑪, 題目中不說這個 convention, 誰 TM 知道).

寫一個 reverse(pre, next)使得夾在 pre 和 next 之間的那部分(不包括 pre 和 next)反轉. 主函數 reverseKgroup(head, k)定出 pre 和 next 的位置, 然後調用 reverse. reverse 返回的是反轉後的 list 的最後一個元素, 它好作為下一次調用 reverse(pre, next)時的 pre.

History:
E1 直接看的答案.
E2 寫了五個小時, 終於通過了, 花了很長時間弄清題目意思.
E3 一次通過, NB, 比 E2 強多了. 方法跟以下答案差不多, 都是寫了個 reverse(pre, next)使得夾在 pre 和 next 之間的那部分(不包括 pre 和 next)反轉, 寫法也差不多, 但主函數 reverseKgroup(head, k)函數的寫法差別較大, 我的代碼沒有答案那麼簡明, 但也不複雜.

这道题主要是利用 reverse 链表的方法，reverse 的方法就是维护三个指针，然后别忘了保存 next 指针就行。Tao: Code Ganker 的解法也與這個差不多.

```
/**
 * Reverse a link list between pre and next exclusively
 * an example:
 * a linked list:
 * 0->1->2->3->4->5->6
 * |                 |
 * pre              next
 * after call pre = reverse(pre, next)
 *
```

```
 * 0->3->2->1->4->5->6
 *               |    |
 *              pre  next
 * @param pre
 * @param next
 * @return the reversed list's last node, which is the precedence of parameter next
 */
private static ListNode reverse(ListNode pre, ListNode next){
    //Tao:  輸入: 0 (pre) ->1->2->3->4 (next)
    ListNode last = pre.next;//where first will be doomed "last"
    ListNode cur = last.next;
   //Tao: 以上使得:  0 (pre) ->1 (last) ->2 (cur)->3->4 (next)
  // Tao: 記憶 1/3: 記住上一行的鏈中 pre, las, cur 三個的位置: last 最初為 pre 後一個, cur 一直都是 last 後一個. 一個完整的循環步驟為: 先從 cur 後面那個元素開刀, 使得把 cur 從鏈中刪掉, cur 再擠到 pre 後面去. 最後讓 cur 這個指針重新指向 last 之後那個元素. 簡記起來就兩句話: last 最初為 pre 後面的那個, 要不停地把 last 後面那個放到 pre 後面去, 就可以實現反轉.
    while(cur != next){    // Tao: 記憶 2/3: while 循環條件為 cur ! = next, 即 cur 沒有到輸入鏈之末尾(每次要移動的就是 cur)
            // Tao: 以下顏色表示:
            循環第一輪,
            循環第二輪 (Turqoise 4),
            循環第三輪 (dark violet),
            循環第四輪 (sky blue)
        last.next = cur.next;
        //Tao: 以上語句運行前為 0 (pre) ->1 (last) ->2 (cur)->3->4 (next) , 運行後為:
        //Tao: 0 (pre) ->1 (last) ->3->4 (next)
        //Tao: 另一條: 2 (cur)->3->4 (next)
        //Tao: 0 (pre) -> 2 ->1 (last) ->4 (next)
        // Tao: 另一條: 3 (cur) ->4 (next)
        cur.next = pre.next;
```

```
        //Tao: 0 (pre) ->1 (last) ->3->4 (next)
        //Tao: 2 (cur)->1 (last) ->3->4 (next)
        //Tao: 0 (pre) -> 2 ->1 (last) ->4 (next)
         //Tao: 3 (cur) ->2 ->1 (last) ->4 (next)

         pre.next = cur;
        //Tao: 0 (pre) -> 2 (cur)->1 (last) ->3->4 (next)
         //Tao: 0 (pre) ->3 (cur) ->2 ->1 (last) ->4 (next)
         cur = last.next;
        //Tao: 0 (pre) -> 2 ->1 (last) ->3 (cur) ->4 (next)
        //Tao: 0 (pre) ->3 ->2 ->1 (last) ->4 (next, cur)
    }
    return last;  // Tao: 記憶 3/3: return last.
}

public static ListNode reverseKGroup(ListNode head, int k) {
        //Tao: 設輸入 1 (head) ->2->3->4->5->6,  k=3
        if(head == null || k == 1)
            return head;

        ListNode dummy = new ListNode(0); //Tao: 0 ->1 (head) ->2->3->4->5->6
        dummy.next = head; //Tao: 0 (dummy) ->1 (head) ->2->3->4->5->6
        int count = 0;
        ListNode pre = dummy;
        ListNode cur = head; //Tao: 0 (dummy, pre) ->1 (head, cur) ->2->3->4->5->6
        while(cur != null){
            count ++;
            ListNode next = cur.next;
           //Tao: 0 (pre) ->1 (cur) ->2 (next) ->3->4->5->6,  count = 1
```

```
        //Tao: 0 (pre) ->1 ->2 (cur) ->3 (next) ->4->5->6,  count = 2
        //Tao: 0 (pre) ->1 ->2 ->3 (cur) ->4 (next)->5->6,  count = 3
        //Tao: 0 ->3 ->2 ->1 (pre) ->4 (cur) ->5 (next)->6,  count = 1
         if(count == k){
            pre = reverse(pre, next);
    //Tao: reverse(pre, next)前: 0 (dummy, pre) ->1 (head) ->2 ->3 (cur) ->4 (next)->5->6,  count = 3
    //Tao: reverse(pre, next)後: 0 ->3 ->2 ->1 (pre) ->4 (next) ->5->6,  count = 3
            count = 0;
    //Tao: 0 ->3 ->2 ->1 (pre) ->4 (next) ->5->6,  count = 0
         } //end of if
         cur = next;
         //Tao: 0 (dummy, pre) ->1 (head) ->2 (next, cur) ->3->4->5->6,  count = 1
         //Tao: 0 (dummy, pre) ->1 (head) ->2 ->3 (next, cur) ->4->5->6,  count = 2
         //Tao: 0 ->3 ->2 ->1 (pre) ->4 (next, cur) ->5->6,  count = 0
       }//end of while
     return dummy.next;
    }
```

---

26. Remove Duplicates from Sorted Array, Easy

http://blog.csdn.net/linhuanmars/article/details/20023993

Given a sorted array, remove the duplicates in place such that each element appear only *once* and return the new length.
Do not allocate extra space for another array, you must do this in place with constant memory.
For example,
Given input array *nums* = [1,1,2],
Your function should return length = 2, with the first two elements of *nums* being 1 and 2 respectively.
It doesn't matter what you leave beyond the new length.
Subscribe to see which companies asked this question

Key: 若 A[i] != A[i-1]則把 A[i]放入正確的位置, 這樣基本上沒有 corner case, 比 E1 的好多了.

History:
E1 的做法是若 A[i] == A[i-1]則跳過, 這樣會弄出很多 corner case, 最後沒通過.
E2 幾分鍾寫好一次通過, 代碼除了變量名之外, 跟 Code Ganker 的一模一樣.

E3 很快寫好, 一次通過. E3 是按 '相等則跳過' 的方法寫的(且能一次通過, NB), 沒有 corner case. E3 的代碼帖在最後, 不用, 只是為了說明按 '相等則跳過' 的方法寫也可以沒有 corner case, 且代碼也短.

这道题跟 Remove Element 类似，也是考察数组的基本操作，属于面试中比较简单的题目。做法是维护两个指针，一个保留当前有效元素的长度，一个从前往后扫，然后跳过那些重复的元素。因为数组是有序的，所以重复元素一定相邻，不需要额外记录。时间复杂度是 O(n)，空间复杂度 O(1)。代码如下. 类似的题目有 Remove Duplicates from Sorted List，那道题是在数组中操作，还有 Remove Duplicates from Sorted Array II，这个题目操作有所不同，不过难度也差不多，有兴趣的朋友可以看看。

Tao: 函數 removeDupliactes 返回的雖然是新的長度, 但原數組 A 也被改成了需要的樣子. 又是一個 Fortran subroutine 功能的例子.

Code Ganker 的代碼(用它):

```
public int removeDuplicates(int[] A) {
    if(A == null || A.length==0)
        return 0;
    int index = 1;
    for(int i=1;i<A.length;i++)
    {
        if(A[i]!=A[i-1])
        {
            A[index]=A[i];
            index++;
        }
    }
    return index;
}
```

E3 的代碼(可以看看, 但不用):

```
    public int removeDuplicates(int[] nums) {
        if(nums == null) return 0;
        int n = nums.length;
        if(n <= 1) return n;

        int pos = 0, l = 0, r = 0;

        while(r < n) {
            while(r < n && nums[l] == nums[r]) r++;
            nums[pos] = nums[l];
            pos++;
            l = r;
        }

        return pos;
    }
```

---

27. Remove Element, Easy

Given an array and a value, remove all instances of that value in place and return the new length.
The order of elements can be changed. It doesn't matter what you leave beyond the new length.
Subscribe to see which companies asked this question

Key: for(int i = 0; i < nums.length; i++) {
  if(nums[i] != val) {
   nums[put] = nums[i];
   put++; } }

History:
E1 幾分鐘寫好, 一次通過.
E2 幾分鐘寫好, 一次通過.
E3 幾分鐘寫好, 一次通過.

Tao: 這是目前見過的最簡單的一道題, 幾分鐘寫出來, 一次通過. Code Ganker 的解法還是有點繞, 因為他想盡量少賦值. 以下用我的代碼.

```
public int removeElement(int[] nums, int val) {
  if(nums == null || nums.length == 0)
   return 0;

  int put = 0;

  for(int i = 0; i < nums.length; i++) {
   if(nums[i] != val) {
    nums[put] = nums[i];
    put++;
   }
  }

 return put;
 }
```

---

28. Implement strStr, Easy

http://blog.csdn.net/linhuanmars/article/details/20276833

Implement strStr().
Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.
Subscribe to see which companies asked this question

Key: brute force. 假设原串的长度是 n，匹配串的长度是 m。思路很简单，就是对原串的每一个长度为 m 的字串都判断是否跟匹配串一致

```
 if(haystack.charAt(i+j)!=needle.charAt(j)) {
            successFlag = false; break;}
```

Convention:
當輸入的兩個 String 中有 null 時，返回 0 或-1 都可以.
當輸入的兩個 String 都為""時, 返回 0.
needle 可能比 haystack 長.

若有時間, 可以看看 KMP 算法.

History:
E1 直接看的答案.
E2 一次通過, 代碼跟 Code Ganker 基本上一模一樣.
E3 很快寫好, 方法跟 Code Ganker 的一樣, 對很多例子都能給出正確結果, 但在 OJ 中居然超時了. 後來看了 Code Ganker 的代碼, 將我的代碼中的 begin 掃描的范圍由 begin < haystack.length()改為了 begin <= haystack.length() - needle.length(), 就通過了. 感覺 OJ 完全沒必要作這樣的要求, 所以實際上我 E3 的代碼基本上是對的. 但還是算我此題沒做出來. E3 改進後的代碼比較好懂, 以後都用它.

这是算法中比较经典的问题，判断一个字符串是否是另一个字符串的子串。这个题目最经典的算法应该是 KMP 算法，不熟悉的朋友可以参见 Knuth–Morris–Pratt algorithm。KMP 算法是最优的线性算法，复杂度已经达到这个问题的下限。但是 KMP 算法比较复杂，很难在面试的短时间里面完整正确的实现。所以一般在面试中并不要求实现 KMP 算法。

下面我们先说说 brute force 的算法，假设原串的长度是 n，匹配串的长度是 m。思路很简单，就是对原串的每一个长度为 m 的字串都判断是否跟匹配串一致。总共有 n-m+1 个子串，所以算法时间复杂度为 O((n-m+1)*m)=O(n*m)，空间复杂度是 O(1)。代码如下：

```
public int strStr(String haystack, String needle) {
    if(haystack==null || needle == null || needle.length()==0)
        return 0; //E2 返回的-1, 也通過了 .
    if(needle.length()>haystack.length())
        return -1;
    for(int i=0;i<=haystack.length()-needle.length();i++) //tao: 注意是<=
    {
        boolean successFlag = true; //tao: 這句話要放 for(i)裡面. 若是放外面了, 則一旦後面 if 中把 successFlag 弄成 false, 那麼下一輪 i 循環中就算遇到相同的字串, successFlag 的值也是 false.
        for(int j=0;j<needle.length();j++)
        {
            if(haystack.charAt(i+j)!=needle.charAt(j))
            {
                successFlag = false;
                break;
            }
        }
        if(successFlag)
            return i;
    }
    return -1;
}
```

E3 的(改進後的)代碼:

```
    public static int strStr(String haystack, String needle) {
        if(needle.length() == 0) return 0;
```

```
        if(haystack.length() == 0) return -1;

        int begin = 0;

        while(begin <= haystack.length() - needle.length()) {
            if(haystack.charAt(begin) == needle.charAt(0)) {
                int ihay = begin;
                int ineedle = 0;
                while(ihay < haystack.length() && ineedle < needle.length() && haystack.charAt(ihay) ==
needle.charAt(ineedle)) {
                    if(ineedle == needle.length() - 1) return begin;
                    ihay++;
                    ineedle++;
                }
            }

            begin++;
        }

        return -1;
    }
```

29. Divide Two Integers, Medium

http://blog.csdn.net/linhuanmars/article/details/20024907

Divide two integers without using multiplication, division and mod operator.
If it is overflow, return MAX_INT.
Subscribe to see which companies asked this question

Key: 本題雖然為 Medium, 其實本題很難, leetcode 中通過率只有 15%, 在 Hard 中都算低的.

用 E3 的方法, 好處是不用死記(記 dividend=a, divisor=b), E3 的方法跟 Code Ganker 的思想差不多, 但俱體寫法有區別:

題目中說不能用乘除, 最容易想到的就是在 a 不停地減去 b, 結果就是減的次數. 這樣做的壞處是效率太低(think of Integer.MAX_VALUE / 1).

E3 表示, 除了可以用 加減 外, 還可以用移位(移位就跟乘法差不多了), 故效率高的方法為: 每次將 b 乘 2(即左移一位), 直到 b*2^n < a < b*2^(n+1). 既然此時 b*2^n < a, 那麼是否有 a/b=2^n? 當然不是, 因為 a 和 b*2^n 之間還有差距, 只有當 a 減去這個差距(稱其為 gap)後, 才能這樣, 即 a-gap = b*2^n, (a-gap)/b = 2^n. 故最終的 a/b = 2^n + gap/b. 如何算 gap/b? 遞歸調用 divide 函數. 為了防止越界(如 a=Integer.MIN_VALUE 且取 a 絕對值時), 變量都用 long.

Avadoles, 一定要看: 另外注意, 實踐表明:
若 int x=Integer.MIN_VALUE, 則

long y = Math.abs(x) => y = -2147483648, 顯然不是我們想要的. 這是因為: 若 x 為 int, 則 Math.abs(x)也

是一個 int, 因為實際上調用的 int Math.abs(int x)這個函數. 此時 Math.abs(x)這個 int 越界, 其值為負, 然後 Math.abs(x)再由 int 轉為 long.

long y = Math.abs((long) x) => y = 2147483648, 這下就對了. 這是因為: 若 x 為 long, 則 Math.abs(x)也為 long, 因為實際上調用的 long Math.abs(int long)這個函數. 故此時 Math.abs(x)不會越界.

Conventions:
5/0 = Integer.MAX_VALUE, -5/0 = Integer.MIN_VALUE

Code Ganker 的方法:
記住算法即可, 不用證明. 以下方法的出發點為: 任何一個數都可以寫為 2 的冪方之和:
num = a_0 * 2^0 + a_1 * 2^1 + a_2 * 2^2 +...+ a_n * 2^n

方法(如果忘了, 可用俱體例子來幫助回憶, 以及確認回憶出來的是正確的): 以 40 / 3 = 13 為例. 先由小到大試不同的 n, 使得 2^n 剛好<=40, 然後:

substract	diff	resAppend
3 * 2^3 = 24	40 – substract = 40 – 24 = 16	2^3 = 8
3 * 2^2 = 12	16 – substract = 16 – 12 = 4	2^2 = 4
3 * 2^1 = 6	4 – substract = 4 – 6 < 0	0 (do not append to res)
3 * 2^0 = 3	4 – substract = 4 – 3 = 1	2^0 = 1

最終結果即為 resAppend 一列的總和, 即為 8+4+1=13.

寫的時候, 注意左移一位相當於乘以 2.

History:
E1 沒做出來.

E2 改了很多次後通過, E2 方法應該跟 Code Ganker 差不多, 但沒仔細看 Code Ganker 的代碼, 所以不知道 E2 還是 Code Ganker 的代碼好. E2 的代碼帖在最後. 以上的 Key 是按 E2 的代碼寫的. 由於 E2 代碼是自已寫的, 由方法跟 Code Ganker 差不多, 區別可能只在 corner case 上, 所以以後建議用 E2 代碼.

E3 也改了很多次才通過. E3 以為本題很簡單, 故 '改了很多次才通過後' 還有點鬱悶, 後來才發現我之前對本題的評價是 “其實這道題很難, leetcode 中通過率只有 15%, 在 Hard 中都算低的”. E3 只大概記得本題好像要用跟二進製相關的東西, 故大部分內容是 E3 獨立想到的. 後來才發現 E3 的方法跟 Code Ganker 的思想差不多, 但俱體寫法有區別. 但 E3 的方法的好處是不用死記, 故以後用 E3 的代碼(簡化過).

这道题属于数值处理的题目，对于整数处理的问题，在 Reverse Integer 中我有提到过，比较重要的注意点在于符号和处理越界的问题。对于这道题目，因为不能用乘除法和取余运算，我们只能使用位运算和加减法。比较直接的方法是用被除数一直减去除数，直到为 0。这种方法的迭代次数是结果的大小，即比如结果为 n，算法复杂度是 O(n)。Tao: 我用過這個方法, 但在 leetcode OJ 中是 time limit exceeded.
那么有没有办法优化呢？ 这个我们就得使用位运算。我们知道任何一个整数可以表示成以 2 的幂为底的一组基的线性组合，即 num=a_0*2^0+a_1*2^1+a_2*2^2+...+a_n*2^n。基于以上这个公式以及左移一位相当于乘以 2，我们先让除数左移直到大于被除数之前得到一个最大的基。然后接下来我们每次尝试减去这个基，如果可以则结果增加加 2^k,然后基继续右移迭代，直到基为 0 为止。因为这个方法的迭代次数是按 2 的幂直到超过结果，所以时间复杂度为 O(logn)。代码如下. 这种数值处理的题目在面试中还是比较常见的，类似的题目有 Sqrt(x)，Pow(x, n)等。上述方法在其他整数处理的题目中也可以用到，大家尽量熟悉实现这些问题。

Code Ganker 的代碼(不用):

```
public int divide(int dividend, int divisor) {
    if(divisor == 0)
    {
        return Integer.MAX_VALUE;
    }
    boolean isNeg = (dividend^divisor)>>>31 == 1;
    //Tao: ^是按位异或, 只有在两个比较的位不同时其结果是 1。否则，结果是零。0^0=0, 0^1=1, 1^0=1, 1^1=0. >>> 是补零右移
    //Code Ganker: 这里是判断两个整数是不是同号的简便写法， 你也可以写成乘法， 但是那样又有越界的可能， 而且乘法的计算复杂度比位运算要高。//Tao: 乘法 is not allowed in this problem

    int res = 0;
    if(dividend == Integer.MIN_VALUE)
    //leetcodeOJ 有一個 test case 是-2147483648 / (-1) = 2147483647
    //還有一個是 -2147483648 / 1 =  -2147483648
    //dividend==Integer.MIN_VALUE 这个条件的处理也不是很懂
回复 u012296380：这是比较细节的问题， 我们知道对于 int 类型最小的整数比最大的整数绝对值大 1， 所以如果要取绝对值进行统一处理， 那么就要单独一下处理最小整数的情况， 上面代码的做法是把它加一个除数让他可以取绝对值。
回复 linhuanmars：但是如果除数为负值，是否应该是减去一个除数？
回复 sddxhwj1：是的, 所以这个语句是 dividend += Math.abs(divisor); 如果除数是负数, 加上绝对值就相当于减去一个除数了哈~
    {
        dividend += Math.abs(divisor); //注意不是-=
        if(divisor == -1)
        {
            return Integer.MAX_VALUE;
        }
        res++;
    }
    if(divisor == Integer.MIN_VALUE)
    {
        return res; //Tao: a / 負無窮 = 0
    }
    dividend = Math.abs(dividend);
    divisor = Math.abs(divisor);
    int digit = 0;
    //Tao : 如 20/3, 二进製為 10100/11
    while(divisor <= (dividend>>1)) //注意 dividend>>1 並沒改變 dividend 的值, 若要改變, 就用 dividend>>=1. 為何是 dividend>>1 而不是 dividend?下面我會解釋.
    {
        divisor <<= 1; //Tao: 此即 divisor = divisor << 1, 即 divisor = divisor * 2.
        digit++;
dividend   dividend>>1        divisor                    digit
10100      1010               11(原 divisor)               0
10100      1010               110(=原 divisor * 2)         1
10100      1010               1100(=原 divisor * 2 * 2)    2
//以上最後一行中, divisor 再乘個 2 就要大於 dividend 了, 這就是為什麼 while 中要求的是 divisor<=(dividend>>1)而不是 divisor<=dividend
//以上的 while 循環結束後, 有 divisor*(2 的 digit 次方) < dividend, 且 divisor*(2 的(digit+1)次方) > dividend, 記要算的為 a/b, 記 digit 為 n, 則循環結束後 b 變成了 b*2^n, 則有 b*2^n < a, b*2^(n+1) < a, 即
```

2^n < a/b < 2^(n+1), 即在二进製數的權表(...8421)中, a/b 必在右起第 n 位為 1, 這就是為何下面那個 while 循環中會讓 res += 1<<digit. 而在下面那個 while 循環的第二輪中, i 中要求 dividend>=divisor, 即 a > b*2^(n-1) , 即 a/b > 2^(n-1), 即在權表中 a/b 必在右起第 n-1 位為 1, 故又來個 res += 1<<digit.

```
    }
    while(digit>=0) //記憶: 除了 if 之外, 兩個 while 的 body 內的操作正好是相反的.
    {
        if(dividend>=divisor)
        {
            res += 1<<digit;
            dividend -= divisor;
        }
        divisor >>= 1; //即 divisor = divisor / 2
        digit--;
  dividend   divisor   digit  res
  10100       1100      2    0
  1000        110       1    100     (10100-1100=1000)
  10          11        0    110     (1000-110=10)
  10          1        -1    110     (if 中的沒執行)
    }
    return isNeg?-res:res;
}
```

E2 的代碼(不用):

```
    public static int divide(int dividend, int divisor) {
        // Corner cases
        if(divisor == 0) return Integer.MAX_VALUE;
        if(divisor == Integer.MIN_VALUE) return dividend == Integer.MIN_VALUE ? 1 : 0;

        //以下是專門處理 dividend == Integer.MIN_VALUE 之情況. 這個情況之所以複雜, 是因為若將其轉化為 divide(-dividend, divisor)來算, 則-dividend 越界. 以下的方法(跟 Code Ganker 應該一樣)是先將 dividend 的絕對值中扣除 divisor 這麼多, 如此則 divide(-dividend, divisor)就不越界了, 然後在最後結果中补 1 就是.
        int divisorSign = divisor > 0 ? 1 : - 1;

        if(dividend == Integer.MIN_VALUE) {
            dividend += divisorSign * divisor;
            if(divisor > 0) return -divide(-dividend, divisor) – 1;
            //以下的句子雖然長, 但不難懂, 可以先看這句:
            else return divide(-dividend, -divisor) + 1;
            若只用以上這句, 則對 divide(-dividend, -divisor)= Integer.MAX_VALUE 時會越界, 即-2147483648 / -1 時, divide(-dividend, -divisor)= 2147483647(注意前面 dividend 中已扣除了 divisor 這麼多). 而越界時, 最後結果只需要返回 Integer.MAX_VALUE 就可以了.
            else return divide(-dividend, -divisor) == Integer.MAX_VALUE ? Integer.MAX_VALUE : divide(-dividend, -divisor) + 1;
        }

        if(dividend < 0 && divisor < 0) return divide(-dividend, -divisor);
        if(dividend < 0 || divisor < 0) return -divide(Math.abs(dividend), Math.abs(divisor));
```

```
        // Do normal things
        int res = 0, diff = dividend, resAppend = 1;

        while(divisor * resAppend <= (dividend >>> 1)) resAppend <<= 1;

        while(diff > 0 && resAppend > 0) {
            int substract = divisor * resAppend;
            if(diff >= substract) {
                diff -= substract;
                res += resAppend;
            }
            resAppend >>>= 1;
        }

        return res;
    }
```

E3 的代碼(用它):

```
    public int divide(int dividend, int divisor) {
        long sign = 1;
        long a = (long) dividend, b = (long) divisor;

        if(a == 0) return 0;
        if(a > 0 && b == 0) return Integer.MAX_VALUE;
        if(a < 0 && b == 0) return Integer.MIN_VALUE;
        if((a > 0 && b > 0) || (a < 0 && b < 0)) sign = 1;
        if((a > 0 && b < 0) || (a < 0 && b > 0)) sign = -1;

        a = Math.abs(a);
        b = Math.abs(b);

        if(a < b) return 0;

        long bTry = b;
        long n = 0;

        // bTry 不停地乘以 2, 直到 bTry < a < 2*bTry
        while((bTry << 1) < a) {
            bTry <<= 1;
            n++;
        }
        //此時 bTry < a < 2*bTry, 且 bTry = b*2^n
        //即有 b*2^n < a < b*2^(n+1)

        //以下兩句执行後, 就有 power=2^n. 為何要算它? 因為 b*2^n < a, 故 a/b=2^n <- 不對, 因為 a 和
b*2^n 之間還有差距, 只有當 a 減去這個差距(稱其為 gap)後, 才能這樣, 即 a-gap = b*2^n, (a-gap)/b = 2^n.
最終的 a/b = 2^n + gap/b.
        long twoPower = 1;
```

```
        for(int i = 0; i < n; i++) twoPower <<= 1;

        long gap = a - bTry;
        long left = (long) divide((int) gap, (int) b);

        long resAbs = twoPower + left;
        long res = (sign > 0 ? resAbs : -resAbs); //注意不能寫為 res = sign * res, 因為不能用乘法
        if(res < Integer.MIN_VALUE) return Integer.MIN_VALUE;
        if(res > Integer.MAX_VALUE) return Integer.MAX_VALUE;
        return (int) res;
    }
```

---

30. Substring with Concatenation of All words, Hard

http://segmentfault.com/a/1190000002625580

You are given a string, **s**, and a list of words, **words**, that are all of the same length. Find all starting indices of substring(s) in **s** that is a concatenation of each word in **words**exactly once and without any intervening characters.
For example, given:
s: "barfoothefoobarman"
words: ["foo", "bar"]
You should return the indices: [0,9].
(order does not matter).
Subscribe to see which companies asked this question

Key: 本題其實很難. 用兩個 HashMap 和一個 sliding window. 一個 HashMap 叫 map, 用來放 words 中的單詞, key 是單詞, value 是該單詞在 words 中出現的次數. 另一個 HashMap 叫 curMap, 用來放掃描出來的(可以理解是 sliding window 中的)單詞, key 是單詞, value 是該單詞在該 sliding window 中出現的次數.

本算法之重點是如何循環. 有兩層循環, 內層的 j 循環是按以下方式循環的(先 forget sliding window):
若 s = "0123456789", wordLen = 3 (wordLen 為 words 中每個單詞的長度),
則 j 循環為: 012 345 678 9, 即 j = 0, 3, 6, 9, 即掃描出一堆以 wordLen 為長度的子串: 012 345 678 9. 所以 j 是控製掃描的變量, j 很重要.

外層循環的變量 i 表示 j 的起始位置, 比如以上的 j 循環為 i=0 時的, 而 i=1 時的 j 循環為: 123 456 789, 即 j = 1, 4, 7. 這樣兩層循環一起, 即可保證所有可 能都被掃描到了.

而 sliding window 又是怎樣弄出來的? 我們再定義一個變量 left, 它是 sliding window 的左端(沒有記錄右端的變量, 原因馬上可見), left 的起始值為 i. j 在該 sliding window 中掃. 用 count 來記錄 sliding window 中出現的合法單詞個數(合法的意思就是存在於 words 中的), 當 count 等於 words 中單詞個數時[注 1], 說明已經找到一個題目中要求的子串(其末尾自然就是該 sliding window 的右端, 但不重要), 此時就將 left 的值放入結果 List 中, 然後 left 右移一個 wordLen 長度(注意要將 count 減 1, 並且在 curMap 中將被移出 sliding window 的那個單詞(即之前最左端那個單詞)次數減 1), 開始下一個 sliding window(再說一遍, sliding window 的右端不重要).

有兩種重要的情況要考慮: 一是掃描出來的單詞不在 words 中怎麼辦? 此時說明當前 sliding window 已經不可能是題意要求的了, 注意題意要求的是 barfoo 這樣的子串, 而不是 barcaofoo 這樣的. 此時要放棄當前 sliding window, 開始下一個, 即 left 右移一個單詞的長度, 並且清空 curMap, 清零 count.

第二種情況是掃著掃著, sliding window 中某個單詞出現的次數大於它在 words 中的次數時, 此時的 sliding window 也不可能滿足題意要求了, 要改造當前 sliding window. 此時要做的是 left 逐次右移一個單詞的長度, 直到移至該單詞(次數大的那個單詞)右邊, 沿途的單詞都要在 curMap 中減 1, count 也要減逐個減 1.

[注 1] 為何只根据 count 和 words 中單詞個數 是否相等, 就能判斷是否已找到一個題目中要求的子串? 答: 由上面'第二種情況', 可知, 已保證了 sliding window 中每個單詞出現的次數不會大於它在 words 中的次數, 所以可以根据總數 count 來判斷.

History:
E1 直接看的答案.
E2 是看著下面的步驟寫的, 花了一上午的時間, 還是沒寫出來, 後來發現我的循環方式都跟它不一樣. 所以以下的步驟不能當作自推時的提示用, 只能當作看代碼時的注釋用, 而且說得不太清楚, 所以不要看以下的步驟. 看我上面的紅字就可以了. 後來看以下代碼並默寫, 又花了差不多一個晚上的時間. 以上紅字都是 E2 寫的, E1 基本上甚麼都沒寫.
E3 基本上把 key 中大部分內容都回憶出來了, 寫好後在我電腦上能得出正確結果, 但在 OJ 上對大輸入超時. 然後又在時間效率上做了很多改進, 最後還是超時. 後來看答案, 才發現是我對循環記得不准確: 我將 j 也寫了個循環, 實際上 j 是不用單獨循環的. 然後按這樣 將我的代碼去掉那層循環後(又花了不少時間), 就通過了. 其實, 我最開始在寫的時候就感覺到多了一層循環, 還以為是答案寫得不好, 打算本題做完後自己來去掉一層循環, 優化一下, 結果答案其實並沒有那層循環, 而是我記錯了, 弄得我的代碼也一直超時沒通過(注意我的代碼去掉那層循環後可通過). 所以本題我雖然沒做出來, 但其實離做出來已經很接近了.

Tao: Code Ganker 和 William 也用的類似的方法, 用了兩個 HashMap, 他們的兩層循環的 for 抬頭的寫法也跟這裡一樣.
比较复杂的一题，首先是要明确用滑块的概念来解决，始终保持 L 集合中的字符串在滑块中都只出现了一次，当然设置一个总计数 count，当 cout 等于 L 集合长度时，即使找了一段符合要求的字符串。
需要用到的内存空间：
•两张哈希表，一张保存 L 集合中的单词，一张用来保存当前滑块中的单词，key 为单词，value 为出现次数
•cout 计数，保存当前滑块中的单词总数
•left 标记，记录滑块左起点
实现的步骤：
1.遍历一遍单词数组 L 集合，构造总单词表
2.以单词长度为步长，遍历目标字符串，如果当前单词在总单词表内，则进入步骤 3；反之，则清空当前滑块单词表，将 cout 置零，将 left 移动到下一位置
3.当前滑块档次表中的相应单词计数加 1，检查该单词的计数是否小于等于总单词表中该单词的总数，如果是，则将 count 计数加 1，进入步骤 5；反之，进入步骤 4
4.根据左起点 left 收缩滑块，直到收缩到与当前单词相同的字符串片段，将其剔除之后，滑块的收缩工作完成. Tao: 看後面作者的解釋.
5.如果当前 count 计数等于单词集合长度，记录下 left 左起点的位置后，将 left 右移，当前滑块中相应单词计数减 1，总计数减 1，继续循环
这里解释下步骤 4 中的收缩滑块，这是因为当前滑块中有单词的出现次数超过了额定的出现次数，那么就是需要收缩滑块来剔除这个单词，相当于是从滑块的左起点开始寻找该单词，找到之后，将该单词的右端点作为滑块新的左起点，这样就保证了滑块中所有单词都是小于等于额定出现次数，这样也保证了 count 计数的有效性。
遇到总单词表中不存在的单词的情况，在步骤 2 中已经说明，清空当前数据之后继续循环，也就是保证了滑

块中是不会出现不存在单词表中的单词的。
最后，考虑最外圈循环，如果是从 0 开始作为滑块的初始起点，那么其实并没有遍历字符串中的所有可能子串，因为步长是单词长度，所以移动滑块的时候会跨过很多可能子串，所以要在外圈再加一层循环，这个循环的作用就是移动滑块的初始起点，所以循环次数就是单词的长度。

```
//Tao: 以下代碼太長, 看不清楚, 可以看我 java 文件中的 read-code.java, 那裡有高亮, 還可以查括號管的范圍.
public class Solution {
    public List<Integer> findSubstring(String S, String[] L) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (S == null || S.length() == 0 || L == null || L.length == 0)
            return result;
        int strLen = S.length();
        int wordLen = L[0].length();
        HashMap<String, Integer> map = new HashMap<String, Integer>();

       for (int i = 0; i < L.length; i++) {
            if (map.containsKey(L[i])) {
                map.put(L[i], map.get(L[i]) + 1);
            } else {
                map.put(L[i], 1);
            }
        }

        for (int i = 0; i < wordLen; i++) {
            HashMap<String, Integer> curMap = new HashMap<String, Integer>();
            int count = 0, left = i;

            for (int j = i; j <= strLen - wordLen; j += wordLen) {
                String curStr = S.substring(j, j + wordLen);

                if (map.containsKey(curStr)) {

                    if (curMap.containsKey(curStr)) {
                        curMap.put(curStr, curMap.get(curStr) + 1);
                    } else {
                        curMap.put(curStr, 1);
                    }

                    if (curMap.get(curStr) <= map.get(curStr)) {
                        count++;
                    } else {
                        while (true) {
                            String tmp = S.substring(left, left + wordLen);
                            curMap.put(tmp, curMap.get(tmp) – 1);
                            left += wordLen;
                            if (curStr.equals(tmp)) {
                                break;
                            } else {
                                count--;
```

```
                }
            }
        }

        if (count == L.length) {
            result.add(left);
            String tmp = S.substring(left, left + wordLen);
            curMap.put(tmp, curMap.get(tmp) - 1);
            left += wordLen;
            count--;
        }

    } else {
        curMap.clear();
        count = 0;
        left = j + wordLen;
    }

  }

 }
 return result;
 }
}
```

---

31. Next Permutation, Medium

http://wlcoding.blogspot.com/2015/03/next-permutation.html?view=sidebar

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
Subscribe to see which companies asked this question

Key: 本題之關鍵在於理解題目意思.
題目的意思是：123 的全排列按字典顺序为(Tao: 即將 123 當作一個單詞, 132 當作另一個單詞, 則 123 在字典中是排在 132 前面的)：
123 132 213 231 312 321
如果输入其中某一个序列，返回它的下一个序列。如：输入：213 输出：231 ；输入：321 输出：123

方法要記住:
先從右往左掃, 找出右端最長的降序列(即從左往右看是降序), 記此降序列左邊那個數為小明. 然後將此降序列逆轉(逆轉後為昇序列). 再在 此逆轉後的昇序列 中從左到右找出第一個大於小明的數(記為小紅), 交換小明和小紅, 即可. 例子(下劃線的 3 是小明, 加粗的 **4** 是小紅): (2,3,6,5,4,1) → (2,3,1,**4**,5,6) → (2,**4**,1,3,5,6)

History:
E1 直接看的答案.
E2 通過, 但 E2 的 rerverse 函數寫得有點囉嗦, 還是 William 的 reverse 寫得好.
E3 很快寫好, 本來可以一次通過, 結果不小心將 r = n - 1 寫成了 r = nums[n -1], 改後即通過.

E1 用的 Code Ganker 的代碼. E2 用的 William 的, 因為 William 講得簡明些, 代碼也簡明些. 以後都看 William 的.

William:
Time ~ O(2N), Space ~ O(1)
这道题是全排列中的一步，即给定一个排列，求下一个排列是什么。
方法(我: 看我下面的綠字例子)：
•从右向左扫，找到右端最长的降序列(Tao: 從左往右看是降序)（记录该序列的开头为 num[curr]，则 num[curr - 1] 为分割数，即在 curr - 1 和 curr 间分割）；
•反该转序列成升序；
•在反转后的序列中从左至右找第一个大于分割数的元素，并与分割数交换位置。

Tao: 對算法的理解(不是證明, 只是為了幫助記憶. 不用證, 記住算法即可): 以下面的 (2,3,6,5,4,1) → (2,4,1,3,5,6)為例, 首先, 原來的 3 必須換成比它大的數中最接近 3 的, 所以把 3 換為 4. 其次, 要在詞典中最接近原來的排列, 則 3(現在是 4)後的數都要昇序, 所以將 3 後倒序.

```
public void nextPermutation(int[] num) {
    // Time: O(N), Space: O(1)
    // E.g.: 6 8 7 4 3 2 -> 7 2 3 4 6 8 (partition number: 6, swap with 7)
    //我: 另一個更好的例子是(2,3,6,5,4,1) → (2,3,1,4,5,6) → (2,4,1,3,5,6)

    // find longest descending tail and reverse it, num[curr - 1] is the partition number
    int curr = num.length - 1;
    while (curr > 0 && num[curr - 1] >= num[curr])  curr--;
    reverse(num, curr, num.length - 1);
    // swap num[curr - 1] and the first larger element on its right side
    if (curr > 0) {
        int next = curr;
        curr--;
        while (num[curr] >= num[next])  next++;
        swap(num, curr, next);
    }
}

//此 reverse 函數寫得好, 記住寫法. (reverse 就是反轉數組中的元素)
private void reverse(int[] num, int start, int end) {
    while (start < end) {
        swap(num, start++, end--);
    }
}
```

```
private void swap(int[] num, int a, int b) {
    int temp = num[a];
    num[a] = num[b];
    num[b] = temp;
}
```

Code Ganker:

这道题是给定一个数组和一个排列，求下一个排列。算法上其实没有什么特别的地方，主要的问题是经常不是一见到这个题就能马上理清思路。下面我们用一个例子来说明，比如排列是(2,3,6,5,4,1)，求下一个排列的基本步骤是这样：//Tao: 下面說的"後"都是指右, "前"都是指左

1) 先从后往前找到第一个不是依次增长的数，记录下位置 p。比如例子中的 3，对应的位置是 1;

2) 接下来分两种情况：

(1) 如果上面的数字(Tao: 即前面的例子(2,3,6,5,4,1))都是依次增长的(從右往左看的話)，那么说明这是最后一个排列，下一个就是第一个，其实把所有数字反转过来即可(比如(6,5,4,3,2,1)下一个是(1,2,3,4,5,6));

(2) 否则，如果 p 存在，从 p 开始往后找，找到下一个数就比 p 对应的数小的数字，然后两个调换位置，比如例子中的 4。调换位置后得到(2,4,6,5,3,1)。最后把 p(tao: 更准確地說是 p 之前所在位置上的數)之后的所有数字倒序，比如例子中得到(2,4,1,3,5,6), 这个即是要求的下一个排列。

以上方法中，最坏情况需要扫描数组三次，所以时间复杂度是 O(3*n)=O(n)，空间复杂度是 O(1)。代码如下：

这个问题感觉主要是理清求下一个排列的思路，如果有朋友有更好的解法或者思路，欢迎一起来讨论哈，可以留言或者发邮件到 linhuanmars@gmail.com 给我。

敢问楼主是怎么找到这样的规律的。。。

回复 qweyxy_2：这个只能通过观察得到了。。(Tao: 網上很多其它人(不只是 William)也是用的這個方法, 所以應該不是 Code Ganker 原創的)

```
public void nextPermutation(int[] num) {
    if(num==null || num.length==0)
        return;
    int i = num.length-2;
    while(i>=0 && num[i]>=num[i+1]) //注意是>=
    {
        i--;
    }
    if(i>=0)
    {
```

```
        int j=i+1;
        while(j<num.length && num[j]>num[i])
        {
           j++;
        }
        j--;
        int temp = num[i];
        num[i] = num[j];
        num[j] = temp;
    }
    reverse(num, i+1);
}
private void reverse(int[] num, int index)//tao: 此函數之作用是反轉 num 數組中從 index 開始到最後的元素, 注意記住以下的方法
{
    int l = index;
    int r = num.length-1;
    while(l<r)
    {
        int temp = num[l];
        num[l] = num[r];
        num[r] = temp;
        l++;
        r--;
    }
}
```

---

32. Longest Valid Parentheses, Hard

http://blog.csdn.net/linhuanmars/article/details/20439613

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.
For "(()", the longest valid parentheses substring is "()", which has length = 2.
Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.
Subscribe to see which companies asked this question

Key: 注意題目意思是找連續的 valid 的子串. 比如()(()的輸出應為 2, 而不是 4. 用一個 Integer Stack 來記錄每個左括號的位置. 遇到右括號時, 若 stack 為空, 則將該右括號下一位設為 start; 若 stack 不為空, 則出棧, 並計算 local max 長度(分出棧後 stack 是否為空兩種情況).

History:
E1 直接看的答案.
E2 沒做出來.
E3 沒做出來. E3 的方法是 sliding window, i 為左端, j 為右端, i 和 j 都動, 考查以 s[i]開始的子串最長能廷伸多遠. 結果總是不停地有 test case 通不過. 還是 Code Ganker 的方法好. 此方法不容易想到, 要記住.

我: 以下 Code Ganker 表述有點不清楚, 所以先看代碼(clear), 再看 Code Ganker 說的(也可以不看).

原题链接:http://oj.leetcode.com/problems/longest-valid-parentheses/
这道题是求最长的括号序列，比较容易想到用栈这个数据结构。基本思路就是维护一个栈，遇到左括号就进栈，遇到右括号则出栈，并且判断当前合法序列是否为最长序列。不过这道题看似思路简单，但是有许多比较刁钻的测试集。具体来说，主要问题就是遇到右括号时如何判断当前的合法序列的长度。比较健壮的方式如下：
(1) 如果当前栈为空，则说明加上当前右括号没有合法序列（有也是之前判断过的）；
(2) 否则弹出栈顶元素，如果弹出后栈为空，则说明当前括号匹配，我们会维护一个合法开始的起点 start，合法序列的长度即为当前元素的位置-start+1；否则如果栈内仍有元素，则当前合法序列的长度为当前栈顶元素的位置下一位到当前元素的距离，因为栈顶元素后面的括号对肯定是合法的，而且左括号出过栈了。
因为只需要一遍扫描，算法的时间复杂度是 O(n)，空间复杂度是栈的空间，最坏情况是都是左括号，所以是 O(n)。
实现的代码如下.这种用剩余栈的栈顶元素位置信息作为当前合法数据的判断依据是比较重要的技巧，在 Largest Rectangle in Histogram 这道题里面也用到了，有兴趣的朋友可以看看哈。

大牛，我一直关注你的博客，一直有个问题，斗胆请教一下，为什么不用 Java 内置的 Stack 类，而是用 LinkedList 模拟 Stack，这两个有什么区别吗，正常使用哪个更优，面试的时候也尽量用 linkedlist 吗？谢谢！

回复 shileicomeon：这两个是等价的哈~ 纯粹属于个人习惯，面试中用哪种都是可以的哇~

```
public int longestValidParentheses(String s) {
    if(s==null || s.length()==0)
        return 0;
    LinkedList<Integer> stack = new LinkedList<Integer>(); //不是 ArrayList
    int start = 0;
    int max = 0;
    for(int i=0;i<s.length();i++)
    {
        if(s.charAt(i)=='(')
        {
```

```
            stack.push(i);
        }
        else
        {
            if(stack.isEmpty()) //如())()
            {
                start = i+1;
            }
            else
            {
                stack.pop();
                max = stack.isEmpty()?Math.max(max,i-start+1):Math.max(max,i-stack.peek());
                //第一個 Math.max()可用"))(())"輸入來理解, 第二個 Math.max()可用"(()("輸入來理解.
            }
        }
    }
    return max;
}
```

---

33. Search in Rotated Sorted Array, Hard

http://blog.csdn.net/linhuanmars/article/details/20525681

Suppose a sorted array is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
You are given a target value to search. If found in the array return its index, otherwise return -1.
You may assume no duplicate exists in the array.
Subscribe to see which companies asked this question

Key: 用二分查找. 本題雖然是 Hard, 但只要知道方法了, 就很簡單. 題目的意思是數組的後一段移到前面去了, 如題目中的 0 1 2 4 5 6 7 的 4 5 6 7 這段移到前面去, 就成了 4 5 6 7 0 1 2. 本題的元素無重復, 本題的 II 即元素有重復的情況.

方法是:
若 A[m]==target，則返回 m.

若 A[m]<A[r], 則说明 A[m, … r]是有序的(後面的 Extra 會詳講). 因為 A[m, … r]是有序的, 所以我們可以用 A[m]和 A[r]來判斷 target 是否在 A[m, ...r]內. 我們不能用 A[l]和 A[m]來判斷 target 是否在 A[l, ...m]內, 因為 A[l, ...m]不是有序的, 但若由前面已知 target 不在 A[m, ...r]內, 則可知 target 在 A[l, ...m]內.

若 A[m]>A[r], 同上. 注意不用擔心 A[m]=A[r], 因為元素無重復.

Extra: 對於 rotated sorted array 這類題目, 對於輸入數組 A 的某段 A[start, ...end], 記住以下結論:

若元素無重復, 若 A[start] < A[end], 則該段是 sorted (用手機信號那樣的柱狀圖稍想就知道(若有坎坎, 則必有 A[start]>A[end])), 否則若 A[start] > A[end], 則坎坎必在此段.
若元素有重復(本題 II 之情況), 則用本題 II 的 移動邊界 的 trick. 做本題 II 時再看.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 本來可以一次通過的, 結果粗心將 r 初始化為了 nums.length, 改為 nums.length – 1 後即通過. E2 是用的九章模版, 代碼跟九章的基本一模一樣.
E3 本來可以一次通過的, 但忘了專門處理數組兩端, 改後即通過. E3 的代碼更接近九章模版風格, 以後用 E3 代碼.

这道题是二分查找 Search Insert Position 的变体，看似有点麻烦，其实理清一下还是比较简单的。因为 rotate 的缘故，当我们切取一半的时候可能会出现误区，所以我们要做进一步的判断。具体来说，假设数组是 A，每次左边缘为 l，右边缘为 r，还有中间位置是 m。在每次迭代中，分三种情况：

（1）如果 target==A[m]，那么 m 就是我们要的结果，直接返回；
（2）如果 A[m]<A[r]，那么说明从 m 到 r 一定是有序的（没有受到 rotate 的影响），那么我们只需要判断 target 是不是在 m 到 r 之间，如果是则把左边缘移到 m+1，否则就 target 在另一半，即把右边缘移到 m-1。
（3）如果 A[m]>=A[r]，那么说明从 l 到 m 一定是有序的，同样只需要判断 target 是否在这个范围内，相应的移动边缘即可。

根据以上方法，每次我们都可以切掉一半的数据，所以算法的时间复杂度是 O(logn)，空间复杂度是 O(1)。代码如下.

注意在这道题中我们假设了这个数组中不会出现重复元素。如果允许出现重复元素，那么我们要对中间和边缘的相等的情况继续处理，进而影响到时间复杂度，这个问题会在 Search in Rotated Sorted Array II 中具体讨论，大家有兴趣可以看看。

lz，你的代码只考虑了升序吧，如果是 3,2,1,10,9,8,7,6,5，查找 9，你的代码就不行了哟

回复 bupt010：是的哈 ~ 在 leetcode 中这些有序数组是假设成升序的，所以代码中实现的只是升序的情况。你提出的问题很重要哈，是更加通用的一种情况。我想了一下实现起来主要是要先判断是升序还是降序，...

Code Ganker 的代碼(不用):

```
public int search(int[] A, int target) {
    if(A==null || A.length==0)
        return -1;
    int l = 0;
    int r = A.length-1;
    while(l<=r)
    {
        int m = (l+r)/2;
        if(target == A[m])
            return m;
        if(A[m]<A[r])
        {
            if(target>A[m] && target<=A[r])
                l = m+1;
            else //else 後面只有一句話的時候, 就不用{}. 上面 if 後也只有一句話, 也不用{}.
                r = m-1;
        }
        else
        {
            if(target>=A[l] && target<A[m])
                r = m-1;
```

```
            else
                l = m+1;
        }
    }
    return -1;
}
```

E2 的代碼(不用):

```
    public int search(int[] nums, int target) {
        if(nums == null || nums.length == 0) return -1;
        int l = 0, r = nums.length - 1;

        while(l + 1 < r) {
            int m = (l + r) / 2;

            if(nums[m] == target) return m;

            if(nums[m] < nums[r]) {
                if(nums[m] < target && target <= nums[r]) l = m; // nums[m] == target 的情況在上面考慮過了
                else r = m;
            } else {
                if(nums[l] <= target && target < nums[m]) r = m;
                else l = m;
            }
        }

        if(nums[l] == target) return l;
        if(nums[r] == target) return r;

        return -1;
    }
```

E3 的代碼(用它):

```
    public int search(int[] nums, int target) {
        if(nums == null || nums.length == 0) return -1;
        int n = nums.length;
        int l = 0, r = n - 1;

        //以下是專門處理數組兩端. 只有一個元素的數組 和 只有兩個元素的數組 也一並被處理了.
        if(target == nums[0]) return 0;
        else if(target == nums[n - 1]) return n - 1;

        while(l + 1 < r) {
            int m = (l + r) / 2;
            if(target == nums[m]) return m;

            if(nums[l] > nums[m]) {
                if(nums[m] < target && target < nums[r]) l = m;
                else r = m;
```

```
        } else {
            if(nums[l] < target && target < nums[m]) r = m;
            else l = m;
        }
    }

    return -1;
}
```

---

34. Search for a Range, Medium

http://blog.csdn.net/linhuanmars/article/details/20593391

Given a sorted array of integers, find the starting and ending position of a given target value.
Your algorithm's runtime complexity must be in the order of *O*(log *n*).
If the target is not found in the array, return [-1, -1].
For example,
Given [5, 7, 7, 8, 8, 10] and target value 8,
return [3, 4].
Subscribe to see which companies asked this question

Key: 用兩次二分查找. 一次用 if(A[m]<target)可以找出右邊界, 一次用 if(A[m]<=target)可以找出左邊界. 用例子一想即知. 此方法不易想到, 要記住.

History:
E1 直接看的答案.

E2 改了很多次後通過. E2 是按九章模版寫的(方法是以上 key 的), 這個模版還有個好處是不用擔心到底應該是 l=m, r=m 還是 l=m+1, r=m-1, which is what happened in the codes of Code Ganker below, 當然也有可能是我結果取的[r1, l2], 而 Code Ganker 取的[l1, r2]有關, 但我試著將我的也改成 l=m+1, r=m-1 且取[l1, r2], 但沒通過(本題可能會受[r1, l2]還是[l1, r2]之影响, 只用 l=m, r=m 這點不是很明顯, 在 74. Search a 2D Matrix 中, 比較 Code Ganker 和九章的代碼, 就明顯了). 以後用 E2 的代碼(在最後). 九章算法的代碼太長太囉嗦, 所以不用.

E3 改了一次後通過(A[m]=target 後忘了 return, 但 key 中方法不用 return). E3 是用的一次二分查找, 找到 target 後, 用一兩個指針從 target 開始 左右擴展, 直到遇到不為 target 的數. E3 方法沒有 key 好, 因為 E3 方法最壞情況(所有元素都等於 target)時間為 O(N).

这道题是二分查找 Search Insert Position 的变体，思路并不复杂，就是先用二分查找找到其中一个 target，然后再往左右找到 target 的边缘。找边缘的方法跟二分查找仍然是一样的，只是切半的条件变成相等，或者不等（往左边找则是小于，往右边找则是大于）。这样下来总共进行了三次二分查找，所以算法的时间复杂度仍是 O(logn)，空间复杂度是 O(1)。 代码如下：

(tao: 這段代碼有點 redundant, 所以我沒用)

有朋友在留言中提到这里可以只用两次二分查找就足够了，确实如此。 如果我们不寻找那个元素先，而是直接相等的时候也向一个方向继续夹逼，如果向右夹逼，最后就会停在右边界，而向左夹逼则会停在左边界，如此用停下来的两个边界就可以知道结果了，只需要两次二分查找。代码如下：

```
public int[] searchRange(int[] A, int target) {
    int[] res = {-1,-1};
    if(A==null || A.length==0)
    {
        return res;
    }
    int ll = 0;
    int lr = A.length-1;
    while(ll<=lr)
    {
        int m = (ll+lr)/2;
        if(A[m]<target)
        {
            ll = m+1;
        }
        else
        {
            lr = m-1; //循環結束後, lr 就是左邊界. 當然最後返回的是 ll, 因為 循環結束後它和 lr 相等
        }
    }
    int rl = 0;
    int rr = A.length-1;
    while(rl<=rr)
    {
        int m = (rl+rr)/2;
        if(A[m]<=target)
        {
            rl = m+1; //循環結束後, rl 就是左邊界. 當然最後返回的是 rr, 因為 循環結束後它和 rl 相等
        }
        else
        {
            rr = m-1;
        }
    }
    if(ll<=rr)
    {
        res[0] = ll;
        res[1] = rr;
    }
    return res;
}
```

E2 的代碼:

```
    public int[] searchRange(int[] nums, int target) {
        int[] res = {-1, -1};
        if(nums == null || nums.length == 0) return res;

        int n = nums.length;
```

```
    int l1 = 0, r1 = n - 1;
    int l2 = 0, r2 = n - 1;

    //找出右邊界
    while(l1 + 1 < r1) {
        int m1 = (l1 + r1) / 2;
        if(nums[m1] < target) l1 = m1;
        else r1 = m1;
    }

    //找出左邊界
    while(l2 + 1 < r2) {
        int m2 = (l2 + r2) / 2;
        if(nums[m2] <= target) l2 = m2;
        else r2 = m2;
    }

    if(nums[r1] == target) res[0] = r1;
    if(nums[l2] == target) res[1] = l2;

    if(nums[0] == target) res[0] = 0;
    if(nums[n - 1] == target) res[1] = n - 1;

    return res;
}
```

---

35. Search Insert Position, Medium

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
You may assume no duplicates in the array.
Here are few examples.
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0
Subscribe to see which companies asked this question

Key: 二分查找, 直接寫, 注意邊界. 題意是若找不到, 則返回 'target 插入後, target 在數組中的位置'.

二分法有時是 while(l < r), 有時是 while(l <= r), 關於這個問題, 群裡好幾個人都說九章算法的模版很好使, 基本萬用. 也們帖出的一個帖子中說:
“永遠都用 low + 1 < high 當結束條件 就可以了
循環結束之後判斷兩個數值一個 low index 的, 一個 high index 的是否符合你的條件”
九章模版還有個好處是不用擔心到底應該是 l=m, r=m 還是 l=m+1, r=m-1 (群裡有人這樣說, 可見 34. Search for a Range, 但那裡不明顯, 在 74. Search a 2D Matrix 中更明顯)

注意九章模版中,
1. 對於有三個或三個以上元素的數組, m 是到不了數組兩頭的(即 m 的取值是取不到 0 和 nums.length-1 的), 所以最後可能要特殊討論數組的兩頭.
2. while 循環結束後, l 比 r 小 1, 可能要特殊討論 A[l]和 A[r].
3. A 只有兩個元素時, 無法进入 while.

History:

E1 幾分鐘寫好, 一次通過. Code Ganker 的基本上是一樣的. 但還是要看看 Code

E2 幾分鐘寫好, 本來可以一次通過的, 結果最後處理邊界情況時, 有個 if 寫到另一個 if 前面去了, 使得它搶了另一個 if 的風頭, 改後即通過. E2 的代碼是按九章模版寫的, 但不及 E1 的簡明, 尼瑪 E2 還不如 E1! 九章算法的代碼也帖在後面, 可以看看. 九章代碼跟 E2 幾乎一模一樣,.

E3 幾分鍾寫好, 改了幾次才通過, 這是因為我把題意理解錯了, 題意是若找不到, 則返回 'target 插入後, target 在數組中的位置', 我以為是返回 '插入前, 比 target 小的那個數的位置'. E3 的代碼最簡短(比九章代碼還短), 且用的九章模版, 故以後用 E3 的代碼.

Ganker 說的:

算法复杂度是 O(logn)，空间复杂度 O(1). 二分查找是一个非常经典的方法，不过一般在面试中很少直接考二分查找，会考一些变体，例如 Search in Rotated Sorted Array，Search for a Range 和 Search a 2D Matrix，思路其实是类似的，稍微变体一下即可，有兴趣可以练习一下哈。

E1 的代碼(不用):
```
public static int searchInsert(int[] nums, int target) {
  if(nums == null || nums.length == 0)
   return -1;

  int l = 0;
  int r = nums.length - 1;
  while(l <= r) {
   int m = (l + r) / 2;

   if(target == nums[m])
    return m;

   if(target > nums[m])
    l = m + 1;
   else
    r = m - 1;
  }

 return l;
```

E2 的代碼(不用):
```
    public static int searchInsert(int[] nums, int target) {
        if(nums == null || nums.length == 0) return -1;
```

```
        if(nums[0] == target) return 0; //此句是必須的, 否則通不過.

        int l = 0, r = nums.length - 1;

        while(l + 1 < r) {
            int m = (l + r) / 2;

            if(nums[m] == target) return m;
            else if(nums[m] < target) l = m;
            else r = m;
        }

        if(target < nums[l]) return l;
        if(target > nums[r]) return r + 1;
        if(target > nums[l]) return l + 1;

        return -1;
    }
```

E3 的代碼(用它, 因為最簡短, 且用的九章模版):

```
    public int searchInsert(int[] nums, int target) {
        if(nums == null || nums.length == 0) return -1;
        int n = nums.length;
        int l = 0, r = n - 1;

        if(target > nums[n - 1]) return n; //例如題目中的[1,3,5,6], 7 → 4
        if(target <= nums[0]) return 0; //例如題目中的[1,3,5,6], 0 → 0

        while(l + 1 < r) {
            int m = (l + r) / 2;
            if(target == nums[m]) return m;
            else if(target < nums[m]) r = m;
            else l = m;
        }

        return l + 1; //沒找到 target 時, 返回 l+1.  這是因為循環結束後, target 就在 nums[l]和 nums[r]之間. 用題目中的[1,3,5,6], 2 → 1 這個例子一想, 就明白了.
    }
```

九章算法:

```
public class Solution {
    public int searchInsert(int[] A, int target) {
        if (A == null || A.length == 0) {
            return 0;
        }
        int start = 0, end = A.length - 1;

        while (start + 1 < end) {
```

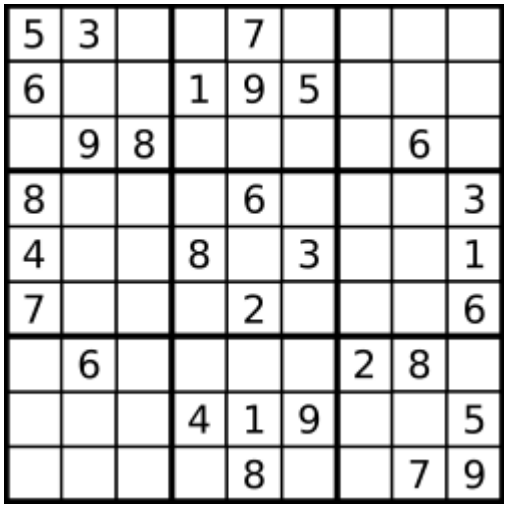
```
            int mid = start + (end - start) / 2;
            if (A[mid] == target) {
                return mid;
            } else if (A[mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }

        //以下三個 if 跟 '我 E2 最後的三個 if' 其實是一樣的
        if (A[start] >= target) {
            return start;
        } else if (A[end] >= target) {
            return end;
        } else {
            return end + 1;
        }
    }
}
```

---

36. Valid Sudoku, Easy

http://blog.csdn.net/linhuanmars/article/details/20748171

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



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

validated.
Subscribe to see which companies asked this question

Key: Sudoku Rules 見下. 本題直接用 brute force, 就是檢查每行, 每列, 每個 sub-box 中的數字有沒有重復. 對每一行, 用一個叫 map 的 boolean 數組, map[i]表示 i 這個數已經在本行出現過了; 對每一列或 block, 可以同樣這樣做. 注意 board 數組中的元素是 char 類型的, 而不是 int. 記住處理 sub-box 時的循環技巧:

```
for(int block=0;block<9;block++)
    for(int i=block/3*3;i<block/3*3+3;i++)
        for(int j=block%3*3;j<block%3*3+3;j++)
```

對以此循環的理解見下面兩個表:

block	0 1 2	3 4 5	6 7 8
block/3*3 block/3*3+3	0 3	3 6	6 9
block%3*3 block%3*3+3	0 3 6 3 6 9	0 3 6 3 6 9	0 3 6 3 6 9

block 循環時的棋盤為(以下每格表示實際棋盤中的九格):

block = 0 i = 0 1 2 j = 0 1 2	block = 1 i = 0 1 2 j = 3 4 5	block = 2 i = 0 1 2 j = 6 7 8
block = 3 i = 3 4 5 j = 0 1 2	block = 4 i = 3 4 5 j = 3 4 5	block = 5 i = 3 4 5 j = 6 7 8
block = 6 i = 6 7 8 j = 0 1 2	block = 7 i = 6 7 8 j = 3 4 5	block = 8 i = 6 7 8 j = 6 7 8

History:
E1 幾分鐘寫好, 一次通過(按 E2 的標准一次通過的, 沒事先自己 test). 我自己的代碼也是幾分鐘寫好, 一次通

過(按 E2 的標准一次通過的!). 但在判斷 sub-box 的時候, 我的代碼要用個四重 for 循環, 而 Code Ganker 的用三重, 所以代碼比我的簡潔. 時間復雜度應該跟我是一樣的.

E2 反而因為沒看清楚 board 數組中的元素是 char 類型的而沒有一次通過, 改了兩次才通過.

E3 花了好一會兒才寫好, 本來可以一次通過的, 結果不小心將 r 和 c 寫成了 i(完全是 typo), 改後即通過.

以下來自題目中的鏈接: Sudoku Puzzles - The Rules.

There are just 3 rules to Sudoku.

Each row must have the numbers 1-9 occuring just once.

Each column must have the numbers 1-9 occuring just once.

And the numbers 1-9 must occur just once in each of the 9 sub-boxes of the grid.

Code Ganker:

这道题是 Sudoku Solver 的一个子问题，在解数独的时候我们需要验证当前数盘是否合法。其实思路比较简单，也就是用 brute force。对于每一行，每一列，每个九宫格进行验证，总共需要 27 次验证，每次看九个元素。所以时间复杂度就是 O(3*n^2), n=9。代码如下. 这道题其实没有什么好的算法，基本上就是遍历去检查，实现上就是数组的操作，属于 Sudoku Solver 的 subroutine，但是在 Sudoku Solver 中实现上又可以进行优化，没必要每次检查整个 board，只需要看当前加进去的数就可以，有兴趣的朋友可以看看哈。

```
public boolean isValidSudoku(char[][] board) {
    if(board==null || board.length!=9 || board[0].length!=9)
        return false;

    //Check each line:
    for(int i=0;i<9;i++)
    {
        boolean[] map = new boolean[9]; //注意 map 是一個數組, 不是 HashMap!
        for(int j=0;j<9;j++)
        {
            if(board[i][j] != '.')
            {
                if(map[(int)(board[i][j]-'1')])
                {
                    return false;
                }
                map[(int)(board[i][j]-'1')] = true;
            }
        }
    }
     //Check each column:
    for(int j=0;j<9;j++)
    {
        boolean[] map = new boolean[9];
        for(int i=0;i<9;i++)
```

```
        {
            if(board[i][j] != '.')
            {
                if(map[(int)(board[i][j]-'1')])
                {
                    return false;
                }
                map[(int)(board[i][j]-'1')] = true;
            }
        }
    }

    //Check each sub-box:
    for(int block=0;block<9;block++)
    {
        boolean[] map = new boolean[9];
        for(int i=block/3*3;i<block/3*3+3;i++)
        {
            for(int j=block%3*3;j<block%3*3+3;j++)
            {
                if(board[i][j] != '.')
                {
                    if(map[(int)(board[i][j]-'1')])
                    {
                        return false;
                    }
                    map[(int)(board[i][j]-'1')] = true;
                }
            }
        }
    }
    return true;
}
```

37. Sudoku Solver, Hard

http://blog.csdn.net/linhuanmars/article/details/20748761

Write a program to solve a Sudoku puzzle by filling the empty cells.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Empty cells are indicated by the character '.'.
You may assume that there will be only one unique solution.

A sudoku puzzle...

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

...and its solution numbers marked in red.
Subscribe to see which companies asked this question

Key: DFS. 用 William 的代碼(在最後), 因為比 Code Ganker 的簡明很多.

寫一個用來遞歸的函數 boolean isSolvable(char[][] board), 其作用為將 board 按題目要求 solve 掉, 當然若不可解則返回 false, 可解則返回 true. 在 isSolvable 中寫 i,j,k 三層循環, i 為 board 行循環, j 為 board 列循環, k 為要填入 board[i][j]的數(1 到 9). 在 k 循環中要用 History 中那句紅字.

本題的 isValide 函數跟本題 I 中的 isValidSudoku 函數寫法不太一樣, 故可以不用跳回去看. 不同之處在於, I 中要判斷整個盤是不是 valid, 而本題中只需判斷新加的那個數是不是 valid, 之前已存在的數都已知是 valid 的了.

History:
E1 直接看的答案.

E2 是按 William 的算法寫的, 最開始寫出來是死循環, 想了好久, 後來改為 if(isValid(i, j, board) && isSolvable(board)) return true, 且 k 循環結束後 if(board[i][j] == '.') return false; 然後通過. 後來發現這跟 William 的不謀而合(William 第二句沒有 if(board[i][j] == '.')而直接 return false, 但等價).

E3 沒做出來. E3 寫出的代碼老是是死循環, 試了用 visited[][]數組, 還試了 if(nEmpty == 0) return (其中 nEmpty 表示 空格子 的個數), 結果還是死循環, 改了半天 後放棄了.

```
//   j      0 1 2 | 3 4 5 | 6 7 8
// j / 3    0 0 0 | 1 1 1 | 2 2 2
// j % 3    0 1 2 | 0 1 2 | 0 1 2
// if j / 3 = 1, then:
// (j / 3) * 3 + c (c = 0, 1, 2) = 3, 4, 5
```

这道题的方法就是用在 N-Queens 中介绍的常见套路。简单地说思路就是循环处理子问题，对于每个格子，带入不同的 9 个数，然后判合法，如果成立就递归继续，结束后把数字设回空。大家可以看出代码结构和 N-Queens 是完全一样的。判合法可以用 Valid Sudoku 做为 subroutine，但是其实在这里因为每次进入时已经保证之前的 board 不会冲突，所以不需要判断整个盘，只需要看当前加入的数字和之前是否冲突就可以，这样可以大大提高运行效率，毕竟判合法在程序中被多次调用。实现的代码如下. 再强调一下哈，以上方法是一个非常典型的套路，大部分 NP 问题的都是可以这个方法，比如 N-Queens，Combination Sum，Combinations，Permutations 等。

Code Ganker 的代碼(不用):

```
public void solveSudoku(char[][] board) {
    if(board == null || board.length != 9 || board[0].length !=9)
        return;
    helper(board,0,0);
}
//tao: 以下是 helper 函數
private boolean helper(char[][] board, int i, int j)
{
    if(j>=9)
        return helper(board,i+1,0);
    if(i==9)
    {
        return true;
    }
    if(board[i][j]=='.')
    {
        for(int k=1;k<=9;k++) //注意不要寫成了 for(int k=0;k<9;k++)
        {
            board[i][j] = (char)(k+'0');
            if(isValid(board,i,j))
            {
                if(helper(board,i,j+1))
                    return true;
            }
            board[i][j] = '.';
```

//不是太明白第 24 行的作用。不是无论如何 for 循环在下一轮都会尝试下一个数字吗？为什么要显式地把它设为'.'？谢谢

//回复 FMZW：这里是要设回"."，因为这是一个深度优先搜索，有可能会递归先走到后边的结点，那么如果不设回去，在后面的回溯中那个结点就会一直是"9"哈~

//回复 linhuanmars：好的，谢谢指点。所以其实可以把这一行拿到 for loop 外面，这样就好理解了。//tao: 其實我覺得還是在 for loop 裡面比較好理解, 相當於 if(isValid(board,i,j))沒成功, 即在 i,j 設的數不是 valid 時, 此時還是要設回'.', 不然 borad[i,j]就會是 k.

//回复 FMZW：恩，是的，确实那样比较好理解哈~

//我觉得 helper 的 boolean 返回值是有必要的 否则在 21 行处无法判定是否需要 return 如果已经找到 valid solution 应该直接 return 以跳过 24 行的 backtrack

//回复 chuhany：确实， 非常感谢您指出我的问题哈 ~ 当时回答得有问题， 代码太久了自己都忘记了 ~ 这里 boolean 确实是必要的 ~ 否则无法判断那一行是不是需要返回。

```
        }
    }
    else
    {
        return helper(board,i,j+1);
    }
    return false;
}
//tao:以下是 isValid 函數, 與上一題 Valid Sudoku 的代碼不太一樣. 這裡只判斷 board[i][j]是合 valid
private boolean isValid(char[][] board, int i, int j)
{
   //tao:以下是判斷 board[i][j]所在的同一列中是否有跟他重復的
    for(int k=0;k<9;k++)
    {
        if(k!=j && board[i][k]==board[i][j])
            return false;
    }
  //tao:以下是判斷 board[i][j]所在的同一行中是否有跟他重復的
    for(int k=0;k<9;k++)
    {
        if(k!=i && board[k][j]==board[i][j])
            return false;
    }
  //tao: 以下是判斷 board[i][j]所在的同一個 sub-box 中是否有跟他重復的
    for(int row = i/3*3; row<i/3*3+3; row++)
   // i = 0, 1, 2 時,  i/3*3 = 0; i/3*3+3 = 3,
   // i = 3, 4, 5 時,  i/3*3 = 3; i/3*3+3 = 6
   // i = 6, 7, 8 時,  i/3*3 = 6; i/3*3+3 = 9
  //故 i/3*3 到 i/3*3+3,  j/3*3 到 j/3*3+3 就 board[i][j]所在的那個 sub-box
    {
        for(int col=j/3*3; col<j/3*3+3; col++)
        {
            if((row!=i || col!=j) && board[row][col]==board[i][j])
                return false;
        }
    }
    return true;
}
```

William:

DFS: Time ~ O(N!), Space ~ O(1)
对所有的空格 '.' 逐个填补，从 '1' 选到 '9' 依次尝试，判断：
•此时的 board 是否 valid：用上题的方法 isValid()，区别为这里只要判断当前行、当前列、和当前所在的 subbox 是否含有和自身相同的元素即可，不需要用 Hashtable；
•接下去的 board 是否 solvable：DFS，call recursive method isSolvable()，注意其需要返回 boolean 值，如果联合上面两个条件都成立，则返回 true；如果从 '1' 到 '9' 尝试都不成功，则返回 false。

William 的代碼(用它):

```java
public void solveSudoku(char[][] board) {
    if (!isSolvable(board))
        throw new IllegalArgumentException("Not solvable!");
}

// DFS
private boolean isSolvable(char[][] board) {
    for (int i = 0; i < 9; i++) { //由此可知, 按9*9的board寫, 也能通過.
        for (int j = 0; j < 9; j++) {
            if (board[i][j] == '.') {
                for (int k = 0; k < 9; k++) {
                    board[i][j] = (char)(k + '1');
                    if (isValid(i, j, board) && isSolvable(board)) return true;
                    board[i][j] = '.';
                }
                return false;
            }
        }
    }
    return true;
}

private boolean isValid(int i, int j, char[][] board) {
    // check ith row
    for (int c = 0; c < 9; c++)
        if (c != j && board[i][c] == board[i][j])   return false;
    // check jth col
    for (int r = 0; r < 9; r++)
        if (r != i && board[r][j] == board[i][j])   return false;
    // check (i,j)'s subbox
    for (int k = 0; k < 9; k++) {
        int row = i / 3 * 3 + k / 3, col = j / 3 * 3 + k % 3;
        if (row != i && col != j && board[row][col] == board[i][j]) return false;
    }
    return true;
}
```

---

38. Count and Say, Easy

http://blog.csdn.net/linhuanmars/article/details/20679963

The count-and-say sequence is the sequence of integers beginning as follows:
1, 11, 21, 1211, 111221, ...
1 is read off as "one 1" or 11.
11 is read off as "two 1s" or 21.
21 is read off as "one 2, then one 1" or 1211.
Given an integer *n*, generate the *n*th sequence.
Note: The sequence of integers will be represented as a string.
Subscribe to see which companies asked this question

Key: 寫一個函數 next(pre)用來返回下一個字串, 比如 next(21)=1211, 然後調用 n 次 next(). 在 next()中, 用一個量(position)記錄當前位置, 用一個量(sublen)記錄有相同字符的字串的長度.

Convention: n 是從 1 開始的, 即 sequence 1, 11, 21...中的 1 是第 1 個, 不是第 0 個. 輸入的 n 不會小於 1.

History:
E1 一次通過.
E2 一次通過.
E3 很快寫好, 一次通過.

Tao: 我自己的代碼也是一次通過, 跟 Code Ganker 都一樣的簡明, 方法也差不多, 所以用我自己的代碼. 但還是要看看 Code Ganker 說的：

这道题属于字符串操作的类型，算法上提高空间不大，只能是对于某一个数的串，读过去然后得到计数并生成下一个串。时间复杂度是 O(n*串的长度)，空间复杂度是 O(串的长度)。代码如下.小陷阱就是跑完循环之后记得把最后一个字符也加上，因为之前只是计数而已。这道题属于字符串或者说数组操作的类型，考察对于明确问题和算法的实现能力，一般会在电面，或者最容易的第一道题中出现，力求一遍搞定，不留 bug 哈。

E1 的代碼:

```
public static String countAndSay(int n) {
  String string = "1";

  for(int i=0; i<n-1; i++)
  {
   string = next(string);
  }

  return string;
 }

 private static String next(String pre) {
  String result = ""; //E2 用的 StringBuilder, 以後都用 StringBuilder 比較好.
  int position = 0;

  while(position<pre.length())
  {
   int i = position;
   int sublen = 1;

   while(i < pre.length()-1 && pre.charAt(i) == pre.charAt(i+1))
   {
    sublen++;
    i++;
   }

   position+=sublen;
   result += sublen+""+pre.charAt(position-1);
  }

 return result;
 }
```

E3 的代碼:

```
public String countAndSay(int n) {
    if(n <= 0) return "";
    String s = "1";
    for(int i = 1; i < n; i++) s = getNext(s);
    return s;
}

private String getNext(String s) {
    StringBuilder sb = new StringBuilder();
    int pre = 0, cur = 0;

    while(cur < s.length()) {
        while(cur < s.length() && s.charAt(cur) == s.charAt(pre)) cur++;
        sb.append(Integer.toString(cur - pre));
        sb.append(s.charAt(pre));
        pre = cur;
    }

    return sb.toString();
}
```

---

39. Combination Sum, Medium

http://blog.csdn.net/linhuanmars/article/details/20828631

Given a set of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.
The **same** repeated number may be chosen from ***C*** unlimited number of times.
Note:

- All numbers (including target) will be positive integers.
- Elements in a combination ($a1$, $a2$, … , ak) must be in non-descending order. (ie, $a1 \le a2 \le \ldots \le ak$).
- The solution set must not contain duplicate combinations.

For example, given candidate set 2,3,6,7 and target 7,
A solution set is:
[7]
[2, 2, 3]
Subscribe to see which companies asked this question

Key: item-res 模版遞歸法.
本題輸入數組中的元素既然可以重復選, 那麼輸入數組中的元素就沒必需重復了, 所以可以 assume 輸入數組中的元素無重復. 本題的二與本題正好相反, 輸入數組中的元素既然不能重復選, 所以輸入數組中的元素是可以重復的.

本題用遞歸. 方法的實質為: 先決定要取 candidates[i], 再在 candidates[i]及其後的那段數組中, 找出和為 target - candidates[i]的一堆元素. 其中 i 取所有可能的遍歷值.

方法為: 先將 candidates 排序, 排序唯一的作用(已驗證)是使最後得到的結果是 sorted 的(但 E3 表明, Code Ganker 和 E2 的代碼去掉 sort 也能通過, E3 代碼沒 sort, 也能通過, 應該是題目將結果格式的要求放寬了). 然後寫一個 helper(int[] candidates, int start, int target, List<Integer> item, List<List<Integer>> res), 其作用為: 在 '從 candidates[start]到 candidates 末尾 (inclusive)' 這麼一段子數組中, 按題目要求, 找出和為 target 的一堆數, 放入 item 中. 若 target 為 0, 則表明已放滿, 則將 item 放入 res 中. helper 函數中, 調用自己的方式為: 對 i 從 start 到 candidates.length, 調用 helper(candidates, i, target - candidates[i], item, res), 注意 helper 第二個參數是 i 而不是 i+1, 這是為了允許結果中有重復元素. 另外, 調用 helper 前, 要要求 target>0, 否則會死循環, 因為若輸入的 target 就小於 0, 則 target-candidates[i]求遠不可能為 0(which is the condition of terminating the loop) ← 這樣做最好的方法是像 E3 那樣, 在 helper 最開始, 直接 if(target < 0) return.

History:
E1 直接看的答案, 覺得本題很難, 代碼看都很難看懂, 就更不用說自己寫了(結果 E3 將 key 忘得差不多了, 還是做出來了). 其它人的代碼好像也差不多.

E2 由於 E1 沒寫 key, 而 Code Ganker 也甚麼都沒講, 所以就看的 Wiliam 的 key, 代碼寫好後, 改了好幾次就通過了. 雖然看的 William 的 key, 但我的代碼比 William 的更簡明. 以上的 key 都是 E2 寫的(E3 表示寫得好).

E3 最開始沒在 helper 開始寫 if(target < 0) return, 而是用的其它終止條件, 導致死循環, 添了此句後即通過. E3 也將 key 忘得差不多了(但記得好像是一個一個往後弄, 但不確定), 基本上是自己獨立寫出來的. E3 的代碼跟 Code Ganker 和 E2 差不多, 但最清新簡潔, 以後用 E3 代碼.

这个题是一个 NP 问题，方法仍然是 N-Queens 中介绍的套路。基本思路是先排好序，然后每次递归中把剩下的元素一一加到结果集合中，并且把目标减去加入的元素，然后把剩下元素（包括当前加入的元素）放到下一层递归中解决子问题。算法复杂度因为是 NP 问题，所以自然是指数量级的。代码如下. 注意在实现中 for 循环中第一步有一个判断，那个是为了去除重复元素产生重复结果的影响，因为在这里每个数可以重复使用，所以重复的元素也就没有作用了，所以应该跳过那层递归。这道题有一个非常类似的题目 Combination Sum II，有兴趣的朋友可以看看，一次搞定两个题哈。

Code Ganker 的代碼(不用):

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(candidates == null || candidates.length==0)
        return res;
    Arrays.sort(candidates); //不要此句, 也能通過
    helper(candidates,0,target,new ArrayList<Integer>(),res);
    return res;
}
private void helper(int[] candidates, int start, int target, ArrayList<Integer> item, List<List<Integer>>
res)
{
    if(target<0) //這個 if 和下個 if 是遞歸結束之條件
        return;
    if(target==0)
    {
        res.add(new ArrayList<Integer>(item));
        return;
    }
```

```
    for(int i=start;i<candidates.length;i++)
    {
        if(i>0 && candidates[i]==candidates[i-1])
            continue;
        item.add(candidates[i]);
        //System.out.println(i+" "+(target-candidates[i])+" "+item+" "+res);
        helper(candidates,i,target-candidates[i],item,res);
        //System.out.println("item="+item);
        item.remove(item.size()-1);
    }
}
```

E2 的代碼(不用):

```
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(candidates == null || candidates.length == 0) return res;
        Arrays.sort(candidates);  //不要此句, 也能通過
        helper(candidates, 0, target, new ArrayList<Integer>(), res);
        return res;
    }

    private void helper(int[] candidates, int start, int target, List<Integer> item, List<List<Integer>> res)
{
        if(target == 0) {
            res.add(new ArrayList<Integer>(item));
            return;
        }

        if(target > 0) {
            for(int i = start; i < candidates.length; i++) {
                item.add(candidates[i]);
                helper(candidates, i, target - candidates[i], item, res);
                item.remove(item.size() - 1);
            }
        }

    }
```

E3 的代碼(用它):

```
    public static List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(candidates == null || candidates.length == 0) return res;
        helper(candidates, target, 0, new ArrayList<Integer>(), res);
        return res;
    }

    private static void helper(int[] candidates, int target, int start, List<Integer> item,
List<List<Integer>> res) {
        if(target < 0) return;
```

```
        if(target == 0) {
            res.add(new ArrayList<Integer>(item));
            return;
        }

        for(int i = start; i < candidates.length; i++) {
            item.add(candidates[i]);
            helper(candidates, target - candidates[i], i, item, res);
            item.remove(item.size() - 1);
        }
    }
```

---

40. Combination Sum II, Medium

http://blog.csdn.net/linhuanmars/article/details/20829099

Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.
Each number in ***C*** may only be used **once** in the combination.
Note:

- All numbers (including target) will be positive integers.
- Elements in a combination ($a1, a2, \dots, ak$) must be in non-descending order. (ie, $a1 \le a2 \le \dots \le ak$).
- The solution set must not contain duplicate combinations.

For example, given candidate set 10,1,2,7,6,1,5 and target 8,
A solution set is:
[1, 7]
[1, 2, 5]
[2, 6]
[1, 1, 6]
Subscribe to see which companies asked this question

Key: item-res 模版遞歸法. 本題與 I 的不同在於, 一是要先將輸入數組排序, 好讓重復的元素在一起, 二是本題中輸入數組中的元素只能用一個, 三是輸入數組中可以有重復元素(參見 I 的 key 的). 所以本題的代碼就是在 I 的代碼上作了三處相應的修改, 對於第二點不同, 遞歸調用 helper 時, 將 i 改為 i+1, 即: helper(candidates, i + 1, target - candidates[i], item, res), 對於第三點不同, 在 i 循環的最初, 加一句 if(i > start && candidates[i] == candidates[i - 1]) continue, 好跳過重復元素, 可這樣理解: 例如輸入數組為{1,1,2}, target 為 3, 此 if 可以防止結果中出現兩個{1,2} ← 原因: 在輸入數組{1,2,2}中, 當 start 為 1 時, 即{1,2(start),2}時, i 取 start 時, 產生一個{1,2}結果, 當 i 取 start 下一個時, 就要跳過了.

History:
E1 直接看的答案.

E2 改了好多遍才通過. 最後我 E2 代碼中的 if(i > start ...) continue 一句中的 i>start 與以下答案不謀而合. 整個 E2 的代碼也跟答案基本一樣.

E3 是在本題 I 的 E3 代碼基础上改的, 改後改了幾遍後通過. E3 沒用 i > start 跳過, 而是寫的一句稍不同的句子(見

E3 代碼中), 以下帖出的 E3 代碼將此句換為 Code Ganker 的 i > start 一句. E3 代碼跟 Code Ganker 和 E2 都差不多, 包括那句 i > start. 為了與本題的 I 一致, 以後還是用 E3 的代碼.

这道题跟 Combination Sum 非常相似，不了解的朋友可以先看看，唯一的区别就是这个题目中单个元素用过就不可以重复使用了。乍一看好像区别比较大，但是其实实现上只需要一点点改动就可以完成了，就是递归的时候传进去的 index 应该是当前元素的下一个。代码如下. 在这里我们还是需要在每一次 for 循环前做一次判断，因为虽然一个元素不可以重复使用，但是如果这个元素重复出现是允许的，但是为了避免出现重复的结果集，我们只对于第一次得到这个数进行递归，接下来就跳过这个元素了，因为接下来的情况会在上一层的递归函数被考虑到，这样就可以避免重复元素的出现。这个问题可能会觉得比较绕，大家仔细想想就明白了哈。

Code Ganker 的代碼(不用):

```
public List<List<Integer>> combinationSum2(int[] num, int target) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(num == null || num.length==0)
        return res;
    Arrays.sort(num);
    helper(num,0,target,new ArrayList<Integer>(),res);
    return res;
}
private void helper(int[] num, int start, int target, ArrayList<Integer> item,
List<List<Integer>> res)
{
    if(target == 0)
    {
        res.add(new ArrayList<Integer>(item));
        return;
    }
    if(target<0 || start>=num.length)
        return;
    for(int i=start;i<num.length;i++)
    {
        if(i>start && num[i]==num[i-1]) continue;
        item.add(num[i]);
        helper(num,i+1,target-num[i],item,res);
        item.remove(item.size()-1);
    }
}
```

E2 的代碼(不用):

```
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(candidates == null || candidates.length == 0) return res;
        Arrays.sort(candidates);
        helper(candidates, 0, target, new ArrayList<Integer>(), res);
        return res;
    }

    private void helper(int[] candidates, int start, int target, List<Integer> item, List<List<Integer>> res)
{
        if(target == 0) {
```

```
            res.add(new ArrayList<Integer>(item));
            return;
        }

        if(target > 0) {
            for(int i = start; i < candidates.length; i++) {
                if(i > start && candidates[i] == candidates[i - 1]) continue; //Modification 1
                item.add(candidates[i]);
                helper(candidates, i + 1, target - candidates[i], item, res);//Modification 2
                item.remove(item.size() - 1);
            }
        }

    }
```

E3 的代碼(用它, 按 Code Ganker 的 i > start 改進過):

```
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(candidates == null || candidates.length == 0) return res;
        Arrays.sort(candidates); //Modification 1
        helper(candidates, target, 0, new ArrayList<Integer>(), res);
        return res;
    }

    private void helper(int[] candidates, int target, int start, List<Integer> item, List<List<Integer>> res)
{
        if(target < 0) return;

        if(target == 0) {
            res.add(new ArrayList<Integer>(item));
            return;
        }

        for(int i = start; i < candidates.length; i++) {
            if(i > start && candidates[i] == candidates[i - 1]) continue; //Modification 2
            //E3 原代碼中上句是這樣寫的(也能通過): if(i >= 1 && candidates[i] == candidates[i - 1] && i - 1
>= start) continue;
            item.add(candidates[i]);
            helper(candidates, target - candidates[i], i + 1, item, res); //Modification 3
            item.remove(item.size() - 1);
        }
    }
```

---

41. First Missing Positive, Hard

http://blog.csdn.net/linhuanmars/article/details/20884585

Given an unsorted integer array, find the first missing positive integer.
For example,

Given [1,2,0] return 3,
and [3,4,-1,1] return 2.
Your algorithm should run in *O*(*n*) time and uses constant space.
Subscribe to see which companies asked this question

Key: 題目中的 positive integer 是指從 1 開始的.

先按 非 in-place 的方法寫一遍，然後再按 非 in-place 的方法寫．非 in-place 的方法可以幫助想到 in-place 的方法．

非 in-place 的寫起來很簡單，可以算 Easy 級別的，方法為：用一個新數組 B，其中 B[0]放 1, B[1]放 2, B[2]放 3...若哪個元素違反了 B[i] = i+1, 則說明 i+1 即我們要求的那個 missing positive. 實際寫的時候，掃描原數組 A, 將 A[i]放入 B[A[i] - 1]中即可．

而 in-place 的方法，即將 A 本身也拿來當 B 用，即让 A[0]=1, A[1]=2, A[2]=3, ... ．實際寫的時候，是交換 A[i]和 A[A[i]-1]，交換後的新 A[i]若還是不能滿足 A[i] = i+1, 則繼續保持交換 新的 A[i]和新的 A[A[i]-1], 直到沒法交換(即 A[A[i]-1]的 index (A[i]-1)越界，或 A[i]=A[A[i]-1])為止．

Corner case: 輸入為[]時, 輸出應為 1
History:
E1 直接看的答案.
E2 另外用了一個數組, 可通過. 也試了試寫 in-place 的, 但對跳過的情況沒處理好.
E3 非 in-place 和 in-place 都寫出來並通過了．以上 key 基本上是 E3 寫的．E3 的代碼比較好懂，且 in-place 的方法跟 Code Ganker 一樣，故以後非 in-place 和 in-place 都用 E3 的代碼．

这道题要求用线性时间和常量空间，思想借鉴到了 Counting sort 中的方法，不了解的朋友可以参见 Counting sort - Wikipedia。既然不能用额外空间，那就只有利用数组本身，跟 Counting sort 一样，利用数组的 index 来作为数字本身的索引，把正数按照递增顺序依次放到数组中。即让 A[0]=1, A[1]=2, A[2]=3, ...，这样一来，最后如果哪个数组元素违反了 A[i]=i+1 即说明 i+1 就是我们要求的第一个缺失的正数。对于那些不在范围内的数字，我们可以直接跳过，比如说负数，0，或者超过数组长度的正数，这些都不会是我们的答案。代码如下. 实现中还需要注意一个细节，就是如果当前的数字所对应的下标已经是对应数字了，那么我们也需要跳过，因为那个位置的数字已经满足要求了，否则会出现一直来回交换的死循环。这样一来我们只需要扫描数组两遍，时间复杂度是 O(2*n)=O(n)，而且利用数组本身空间，只需要一个额外变量，所以空间复杂度是 O(1)。这道题个人还是比较喜欢的，既有一点算法思想，在实现中又有一些注意细节，而且整体来说模型比较简单，很适合在面试中出现。

看来出题者的本意是先有这个解法然后编的这道题(tao: 嚴重同意)，所谓 first missing positive integer 是指必须从 1 开始的，有一个 case 是[2]，然后正确答案是 1，但单纯从字面来说，first missing positive integer 对单个元素的情况实在是没有清晰定义。

有个疑问，我相信楼主的 code 肯定是通过测试的，不过单纯从题的描述来看，正整数不一定从 1 开始吧，那么比如从 5 开始，数组长度为 3，就可能出现比数组长度大的数了，这样的话这种对应方式就会出现问题。
当然要解决也很简单，先扫描一遍记录最小最大值，然后相应减去偏差就是了，可能 leetcode 准备的 test case 没有考虑这种情况。

回复 dracodoc：按照题目要求，都是从 1 考虑起，如果没有 1，就直接返回 1 了哈~

```
public int firstMissingPositive(int[] A) {
    if(A==null || A.length==0)
    {
```

```
        return 1; //注意是 1
    }
    for(int i=0;i<A.length;i++)
    { //print(i)  ← a 處
      //print(A)  ← b 處
        if(A[i]<=A.length && A[i]>0 && A[A[i]-1]!=A[i]) //tao: 我們要先把數組重新排列, 使得 A[0]=1,
A[1]=2, A[2]=3, ... , 即 A[A[i]-1] = A[i].
        { //以下為交換 A[A[i]-1] 和 A[i]. 以下的方法其實不容易理解, 可見代碼後的輸出.
            int temp = A[A[i]-1];
            A[A[i]-1] = A[i];
            A[i] = temp;
            i--;
        }
    }
    for(int i=0;i<A.length;i++)
    {
        if(A[i]!=i+1)
            return i+1;
    }
    return A.length+1;
}
```

E3 的代碼(非 in-place):

```
    public int firstMissingPositive(int[] nums) {
        if(nums == null || nums.length == 0) return 1;
        int n = nums.length;
        int[] newNums = new int[n];

        for(int i = 0; i < n; i++)
            if(nums[i] > 0 && nums[i] <= n) newNums[nums[i] - 1] = nums[i];

        for(int i = 0; i < n; i++)
            if(newNums[i] != i + 1) return (i + 1);

        return (n + 1);
    }
```

E3 的代碼(in-place):

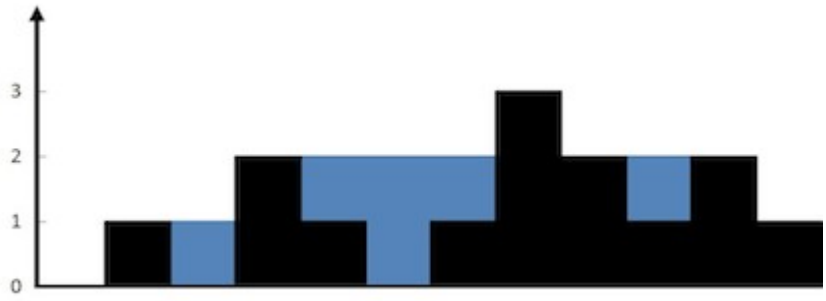
```
    public static int firstMissingPositive(int[] nums) {
        if(nums == null || nums.length == 0) return 1;
        int n = nums.length;

        for(int i = 0; i < n; i++) {
            while(nums[i] != (i + 1)) {
                if(nums[i] <= 0 || nums[i] > n || nums[i] == nums[nums[i] - 1]) break;//if 中前面兩個要求是為
了確保下面 nums[nums[i] - 1]中的 index (nums[i] - 1)不越界
                int temp = nums[i];
                nums[i] = nums[temp - 1];
                nums[temp - 1] = temp;
```

```
        }
    }

    for(int i = 0; i < n; i++)
        if(nums[i] != i + 1) return (i + 1);

    return (n + 1);
}
```

42. Trapping Rain Water, Hard.

http://blog.csdn.net/linhuanmars/article/details/20888505

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. **Thanks Marcos** for contributing this image!
Subscribe to see which companies asked this question

Key: 本題只要知道方法了, 其實不難. 注意題目的意思是求整個數組中的總水量, 而不是單個坑中能放的最大水量.

用兩個指針 l 和 r 從數組兩端往中間掃. l 最初在左端, r 最初在右端. 若某時刻, A[l] < A[r], 則說明 A[l]為短板, 則 A[l]右側附近的積水量由 A[l]決定(此時不管 A[r], 太遠了, 不感興趣). 為何是附近? 因為 A[l]右側的柱子中, 可能有比 A[l]高的(叫它小明, 即小明是 A[l]右側第一個>=A[l]的), 所以 A[l]和小明就構成了一個坑, 將此坑中的水量求出來(如何求? 見下). 然後 A[l]移到小明處, 將 '新的 A[l]' 和 A[r]比較. 若 A[l]仍比 A[r]小, 則重復上面步驟. 若 A[l]比 A[r]大, 則說明 A[r]為短板, 則 A[r]左側附近的積水量由 A[r]決定. 此時在 A[r]左側中找出第一個>=A[r]的(A[r]日出的小明), 則此小明和 A[r]就構成了一個坑, 將此坑中的水量求出來, 然後 A[r]移到小明此處, 再比較 A[l]和 A[r]大小.

算坑中水的方法: 若 A[l]為短板, 則記此 A[l]的值為 min, 然後不停地 l++, 將當前 A[l](注意是 l++後的)頭頂上的水(min-A[l])加到總水量中, 若當前 A[l]大於 min 時(遇到小明了), 不再 l++. 這樣新算出了此坑中的水量. 若 A[r]為短板, 算坑中水的方法一樣, 只是要 r--.

Code Ganker 的兩種方法都是將 height[i]頭頂上的水量累加起來. 關建在於找坑兩側邊緣的短板.

History:

E1 沒寫出來.
E2 直接看的答案.
E3 沒做出來. E3 知道是加頭頂上的水, 但是沒想出來如何移動 l 和 r. E3 基本上重寫了 key.

我: 以下有兩種方法, 要掌握的是第二種. 第一種也要看看, 第一種的方法其實很好懂, 我在代碼中的黃字注釋比 Code Ganker 的解釋要好懂些. 第二種看我上面的紅字也好懂.

这道题比较直接的做法类似 Longest Palindromic Substring 中的第一种方法，对于每一个 bar 往两边扫描，找到它能承受的最大水量，然后累加起来即可。每次往两边扫的复杂度是 O(n)，对于每个 bar 进行处理，所以复杂度是 O(n^2)，空间复杂度是 O(1)。思路比较清晰，就不列代码了，有兴趣的朋友可以实现一下哈。

下面我们说说优化的算法。这种方法是基于动态规划的，基本思路就是维护一个长度为 n 的数组，进行两次扫描，一次从左往右，一次从右往左。第一次扫描的时候维护对于每一个 bar 左边最大的高度是多少，存入数组对应元素中，第二次扫描的时候维护右边最大的高度，并且比较将 左边和右边 小的 最大高度（我们成为瓶颈）存入数组对应元素中。这样两遍扫描之后就可以得到每一个 bar 能承受的最大水量，从而累加得出结果。这个方法只需要两次扫描，所以时间复杂度是 O(2*n)=O(n)。空间上需要一个长度为 n 的数组，复杂度是 O(n)。代码如下：

(以下代碼不用)

```
public int trap(int[] A) {
    if(A==null || A.length==0)
        return 0;
    int max = 0;
    int res = 0;
    int[] container = new int[A.length];

    //第一次掃描, container[i]中放入 A[i]左邊(不包括 A[i])的最大高度
    for(int i=0;i<A.length;i++)
    {
        container[i]=max;
        max = Math.max(max,A[i]);
    }
    max = 0;
    //第二次掃描, container[i]中放入 min(A[i]左邊的最大高度, A[i]右邊的最大高度), 即 A[i]所在坑兩側邊緣的短板.
    for(int i=A.length-1;i>=0;i--)
    {
        container[i] = Math.min(max,container[i]);
        max = Math.max(max,A[i]);
        //以下是算水量. 對於 A[i], 它頭頂上的水量為 短板高度-A[i]. 總水量就是將所有 A[i]頭頂上的水量累加起來.
        res += container[i]-A[i]>0?container[i]-A[i]:0;
```

```
    }
    return res;
}
```

这个方法算是一种常见的技巧，从两边各扫描一次得到我们需要维护的变量，通常适用于当前元素需要两边元素来决定的问题，非常类似的题目是 Candy，有兴趣的朋友可以看看哈。
上面的方法非常容易理解，实现思路也很清晰，不过要进行两次扫描，复杂度前面的常数得是 2，接下来我们要介绍另一种方法，相对不是那么好理解，但是只需要一次扫描就能完成。基本思路是这样的，用两个指针从两端往中间扫，在当前窗口下，如果哪一侧的高度是小的，那么从这里开始继续扫，如果比它(tao: 它指矮點的那側窗口)还小的，肯定装水的瓶颈就是它(tao: 它指矮點的那側窗口)了，可以把装水量加入结果，如果遇到比它大的，立即停止，重新判断左右窗口的大小情况，重复上面的步骤。(tao: 後面的是廢話且表述不清, 可以不看) 这里能作为停下来判断的窗口，说明肯定比前面的大了，所以目前肯定装不了水（不然前面会直接扫过去）。这样当左右窗口相遇时，就可以结束了，因为每个元素的装水量都已经记录过了。代码如下：

用以下代碼:

```
public int trap(int[] A) {
    if(A==null || A.length ==0)
        return 0;
    int l = 0;
    int r = A.length-1;
    int res = 0;
    while(l<r)
    {
        int min = Math.min(A[l],A[r]);
        if(A[l] == min)
        {
            l++;
            while(l<r && A[l]<min)
            {
                res += min-A[l];
                l++;
            }
        }
        else
        {
            r--;
            while(l<r && A[r]<min)
            {
                res += min-A[r];
                r--;
            }
        }
    }
    return res;
}
```

这个算法每个元素只被访问一次，所以时间复杂度是 O(n)，并且常数是 1，比前面的方法更优一些，不过理解起来需要想得比较清楚。
这种两边往中间夹逼的方法也挺常用的，它的核心思路就是向中间夹逼时能确定接下来移动一侧窗口不可能使结果变得更好，所以每次能确定移动一侧指针，直到相遇为止。这种方法带有一些贪心，用到的有 Two

Sum，Container With Most Water，都是不错的题目，有兴趣的朋友可以看看哈。

43. Multiply Strings, Medium

http://blog.csdn.net/fightforyourdream/article/details/17370495

Given two numbers represented as strings, return multiplication of the numbers as a string.
Note: The numbers can be arbitrarily large and are non-negative.
Subscribe to see which companies asked this question

Key: 先反轉輸入的 string(轉化為 StringBuilder 來反轉). 將相乘的結果先放入數組 res 中. 雙層循環遍歷兩個 string. 算法就按平時手算乘法的方法. num1[i] * num[j]就是結果的第 i+j 位, 例如 385*597 中, 8*7 就是結果中的十位(即第 1 位, 一列竪式算就知道了). 將 num1[i] * num[j]加上之前的 res[i+j]和之前的进位 放入 res[i+j]中. res 數組表示的是竪式中的一行, 並不段更新 res, 所以本方法是動態規劃. 最將結果數組轉化為 StringBuilder 然後再反轉回來. 還別忘了移除結果中的前導 0(用 stringBuilder.deleteCharAt(0)). 本題沒甚麼 corner case.
History:
E1 直接看的答案.
E2 改了多次後通過, 方法跟以下代碼差不多, E2 代碼前半部份要比以下代碼簡潔些, 但移除前導 0 的部份要囉嗦些, 下面(在最後)帖出我改進後的 E2 的代碼(即移除前導 0 的部份用的以下代碼的), 以後都用我改進後的 E2 的代碼. 以上的 key 是 E2 寫的.
E3 改了幾次後通過, 方法跟 E2 一樣. 范的錯誤有兩個, 一是 竪式中的每行的最左邊一位應當設為 carry, 我忘了設(見代碼中). 二是最後移除前導 0 時, 對"0*0"的情況沒處理好.

tao: Code Ganker 的代碼不太易懂, 糊到糊到的. 所以用以下很清新的代碼.

这道题一开始觉得很麻烦，后来参考了 http://leetcodenotes.wordpress.com/2013/10/20/leetcode-multiply-strings-%E5%A4%A7%E6%95%B4%E6%95%B0%E7%9A%84%E5%AD%97%E7%AC%A6%E4%B8%B2%E4%B9%98%E6%B3%95/comment-page-1/#comment-122

才知道有如此优雅的写法，这样的写法一遍就可以写出 bug free 的代码。

思路：

1 翻转 string

2 建立数组，双层循环遍历两个 string，把单位的乘积累加到数组相应的位置

3 处理进位并输出

4 注意前导零的 corner case
package Level4;

tao: 上面鏈接所指的網頁中給出的說明為

这题如果直接乘，那肯定会溢出，所以为了解决溢出这个问题，那能考虑的方法就是一位一位相乘，也许自己小学基础没有打好，居然无法直接想到一位一位相乘的算法，最后 google 后发现这个版本的最清晰易懂。

- 直接乘会溢出，所以每次都要两个 single digit 相乘，最大 81，不会溢出。

•比如 385 * 97, 就是个位=5 * 7，十位=8 * 7 + 5 * 9 ，百位=3 * 7 + 8 * 9 ...
可以每一位用一个 Int 表示，存在一个 int[]里面。

•这个数组最大长度是 num1.len + num2.len，比如 99 * 99，最大不会超过 10000，所以 4 位就够了。

•这种个位在后面的，不好做（10 的 0 次方，可惜对应位的数组 index 不是 0 而是 n-1），
所以干脆先把 string reverse 了代码就清晰好多。

•最后结果前面的 0 要清掉。

另外在用 StringBuilder 插 character 的时候记得用 insert(int offset, int num) 这个 function，这样 insert(0, num);就可以从右往左插，可以省了如果从左往右插最后需要 reverse 的那步操作。

```
public class S43 {

    public static void main(String[] args) {
        String num1 = "0";
        String num2 = "0";
        System.out.println(multiply(num1, num2));
    }

    public static String multiply(String num1, String num2) {
        // 先把 string 翻转
        String n1 = new StringBuilder(num1).reverse().toString();
        String n2 = new StringBuilder(num2).reverse().toString();

        int[] d = new int[n1.length()+n2.length()];              // 构建数组存放乘积
        for(int i=0; i<n1.length(); i++){
            for(int j=0; j<n2.length(); j++){
                d[i+j] += (n1.charAt(i)-'0') * (n2.charAt(j)-'0');                  // 在正确位置累加乘积
            }
        }

        StringBuilder sb = new StringBuilder();
        for(int i=0; i<d.length; i++){
            int digit = d[i]%10;              // 当前位
```

```
                int carry = d[i]/10;              // 进位
                if(i+1<d.length){
                    d[i+1] += carry;
                }
                sb.insert(0, digit);              // prepend
            }

            // 移除前导零
            while(sb.charAt(0)=='0' && sb.length()>1){
                sb.deleteCharAt(0);
            }
            return sb.toString();
        }

}
```

以下是我改進後的 E2 的代碼

```
    public String multiply(String num1, String num2) {
        if(num1 == null || num2 == null) return null;
        int n1 = num1.length(), n2 = num2.length();
        if(n1 == 0 || n2 == 0) return "";

        StringBuilder sb1  = new StringBuilder(num1);
        StringBuilder sb2  = new StringBuilder(num2);

        sb1.reverse();
        sb2.reverse();

        int[] resInt = new int[n1 + n2 + 1];
        int carry = 0;

        for(int i2 = 0; i2 < n2; i2++) {

            for(int i1 = 0; i1 < n1; i1++) {
                int d1 = sb1.charAt(i1) - '0';
                int d2 = sb2.charAt(i2) - '0';
```

```
            int temp = resInt[i1 + i2] + d1 * d2 + carry;
            resInt[i1 + i2] = temp % 10;
            carry = temp / 10;
        }
        resInt[i2 + n1] = carry; //竖式中的每行的最左边一位應當設為 carry. E3 最開始沒寫這句.
        carry = 0;
    }

    char[] resChar = new char[resInt.length];

    for(int i = 0; i < resInt.length; i++)
        resChar[i] = (char) (resInt[i] + '0');

    String resString = new String(resChar);
    StringBuilder resSb = new StringBuilder(resString);
    resSb.reverse();

    while(resSb.charAt(0) == '0' && resSb.length() > 1)
        resSb.deleteCharAt(0);

    return resSb.toString();
}
```

---

44. Wildcard Matching, Hard

http://blog.csdn.net/linhuanmars/article/details/21198049

Implement wildcard pattern matching with support for '?' and '*'.
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).

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
Subscribe to see which companies asked this question

Key: DP. 用 d(i, j)表示 's 的前 i 個字符(即 s[0, ..i-1])' 和 'p 的前 j 個字符(即 p[0, j-1])' 是否 match(即 isMatch(s[0, ..i-1], p[0, j-1])之結果).
若 p[j-1]可日 s[i-1], 則 d(i, j) = d(i-1, j-1), (p[j]可日 s[i]的意思是: p[j]==s[i] || p[j] == '?')
否則, 若 p[j-1]為*, 則這樣做(本題之核心): 看是否能在 0 到 i 之間(inclusive)找出一個 k, 使得 s[0, k-1] matches p[0, j-2](match 意思同上). 若能找出, 則 d(i, j)為 ture, 因為 s 的後面那段(即 s[k]到 s 末尾)都被 p 中的*對應了. 比如: s = abcdefghijk, p = abc*
初始化 d:
d[0][0]=true,
d(0, j), 即 s 只第一個字母, 若 p[j-1] = '*' 則 d(0, j) = d(0, j-1), 否則 d(0, j)=false. 實際上, d(0,j)為 true 時只有一種情況, 即 s=a, p=a****** (當然 p 也可為******).

More example from Leetcode test cases: isMatch("abefcdgiescdfimde", "ab*cd?i*de")=true
Example by running William's code: isMatch(“abc”, “ab*dlkjgod”) = false.

Corner case: isMatch(“a”, “”), isMatch(“”, “*”)=true, isMatch(“a”, “a*”)=true, isMatch(“b”, “?*?”)=false

History:
E1 直接看的 Code Ganker 答案.
E2 按 Code Ganker 的方法做(E1 的 key 就是抄的 Code Ganker 原話, 沒表述清楚), 做了很久, 很多 Corner case. 最終沒通過, 然後看的 William 的代碼.
E3 沒做出來, 因為 E3 沒用動態規劃, 而是用的雙指針法, 寫出來的代碼對有些 test case 通不過.

我: E2 覺得 Code Ganker 的代碼不好懂, 所以以後用 William 的代碼, 掌握 2d-DP 即可, 有時間了再看看 1-d DP(E2 沒看, 注意 1d-DP 也要用 prev 和 cur 保存歷結果, 所以不要求掌握 1d-DP). Key 是 E2 寫的.

William:
Solution
•'*' 代表任意 sequence（包括为空），注意与上一题的区别，此处 '*' 不是之前字符的 copy， 因此既没有 deletion，也没有 repetition 的作用。
•p 可以 '*' 开头。
•"?*" 可代表一切 String。

注意：这道题的 test case 中有一个很长的 case 会导致超时，可以先比较 p （去除 * ）和 s 的长度，从而节省时间。

1. 2-d DP: Time ~ O(SP), Space ~ O(SP)
Let d(i, j) = true if s[0, i - 1] matches p[0, j - 1] (i, j are string lengths).
Initialize:

•d(0, 0) = true,
•d(0, j): if p[j - 1] == '*', d(j) = d(0, j - 1) // deletion; else d(j) = false.

Fill up the table:
if **p[j - 1] matches s[i - 1],** d(i, j) = d(i - 1, j - 1);
else if **p[j - 1] == '*',** find if there is a s[0, k - 1] that matches p[0, j - 1]
for (k : 0 to i) { if **d(k, j - 1) == true**, d(i, j) = true; }

Note: “p[j] matches s[i]” means p[j] == s[i] || p[j] == '?'.
Return d(M, N).

```
public boolean isMatch(String s, String p) {
    int lenS = s.length(), lenP = p.length();

    // deal with the exceeding time limit case
    //E3 發現若將以下的 int, for, if 三句去掉, 也能通過. 可能是 OJ 將那個 test case 刪掉了
    int count = 0;
    for (int i = 0; i < lenP; i++) {
        if (p.charAt(i) != '*') count++;
    }
    if (count > lenS) return false;

     boolean[][] d = new boolean[lenS + 1][lenP + 1]; // i, j are the lengths of
s[0..i-1] and p[0..j-1]
    d[0][0] = true;
    // initialize the first row
    for (int j = 1; j <= lenP; j++) {
        if (p.charAt(j - 1) == '*') d[0][j] = d[0][j - 1];
    }
    // fill up the table
    for (int i = 1; i <= lenS; i++) {
        for (int j = 1; j <= lenP; j++) {
            if (p.charAt(j - 1) == s.charAt(i - 1) || p.charAt(j - 1) == '?')
                d[i][j] = d[i - 1][j - 1];
            else if (p.charAt(j - 1) == '*') {
                for (int k = 0; k <= i; k++) {
                    if (d[k][j - 1] == true) {
                        d[i][j] = true;
                        break;
                    }
                }
            }
        }
    }
    return d[lenS][lenP];
}
```

2. 1-d DP (滚动数组) : Time ~ O(SP), Space ~ O(P)

```
public boolean isMatch(String s, String p) {
    int lenS = s.length(), lenP = p.length();

    // deal with the exceeding time limit case
    int count = 0;
    for (int i = 0; i < lenP; i++) {
        if (p.charAt(i) != '*') count++;
    }
    if (count > lenS) return false;

    boolean[] d = new boolean[lenP + 1];
    boolean[] isTrue = new boolean[lenP + 1]; // isTrue[j-1] == T iff any d[m][j-1]
== T (0 <= m <= i)
    d[0] = true;
    isTrue[0] = true;
    // initialize the first row
    for (int j = 1; j <= lenP; j++) {
```

```
        if (p.charAt(j - 1) == '*') d[j] = d[j - 1];
        if (d[j] == true)   isTrue[j] = true;
    }
    // fill up the table
    for (int i = 1; i <= lenS; i++) {
        boolean prev = d[0];    // prev stores d[i - 1][j - 1]
        d[0] = false;   // add this line for 1D reduction!!
        for (int j = 1; j <= lenP; j++) {
            boolean curr = d[j];    // curr stores d[i - 1][j]
            if (p.charAt(j - 1) == s.charAt(i - 1) || p.charAt(j - 1) == '?')
                d[j] = prev;
            else if (p.charAt(j - 1) == '*') {
                if (isTrue[j - 1] == true)  d[j] = true;
            } else
                d[j] = false;   // add this line for 1D reduction!!
            if (d[j] == true)   isTrue[j] = true;
            prev = curr;
        }
    }
    return d[lenP];
}
```

Code Ganker:

这道题目其实是 Regular Expression Matching 的简化版，在这里'?'相当于那边的'.'，而'*'相当于那边的'.*'，因为这里'*'就可以代替任何字符串，不需要看前面的字符，所以处理起来更加简单。
brute force 的方法就不重新列举代码了，有兴趣实现的朋友可以参考一下 Regular Expression Matching，代码结构一样，只是处理情况变一下就可以，不过 leetcode 过不了（因为超时）。
我们主要还是说一下动态规划的方法。跟 Regular Expression Matching 一样，还是维护一个假设我们维护一个布尔数组 res[i],代表 s 的前 i 个字符和 p 的前 j 个字符是否匹配(这里因为每次 i 的结果只依赖于 j-1 的结果，所以不需要二维数组，只需要一个一维数组来保存上一行结果即可），递推公式分两种情况：
(1)p[j]不是'*'。情况比较简单，只要判断如果当前 s 的 i 和 p 的 j 上的字符一样（如果有 p 在 j 上的字符是'?'，也是相同），并且 res[i]==true，则更新 res[i+1]为 true，否则 res[i+1]=false;
(2)p[j]是'*'。因为'*'可以匹配任意字符串，所以在前面的 res[i]只要有 true，那么剩下的 res[i+1], res[i+2],...,res[s.length()]就都是 true 了。
算法的时间复杂度因为是两层循环，所以是 O(m*n), 而空间复杂度只用一个一维数组，所以是 O(n)，假设 s 的长度是 n，p 的长度是 m。代码如下. 这种模糊匹配的题目是面试中的一类题目，一般在 onsite 的时候会遇到，如果没有熟悉思路有时候当场可能很难做得很好。

```
public boolean isMatch(String s, String p) {

    if(p.length()==0)

        return s.length()==0;


   if(s.length()>300 && p.charAt(0)=='*' && p.charAt(p.length()-1)=='*') //這一個 if 見後面說明.

    return false;


    boolean[] res = new boolean[s.length()+1];

    res[0] = true;
```

```
    for(int j=0;j<p.length();j++) //p 循環
    {
        if(p.charAt(j)!='*')
        {
            for(int i=s.length()-1;i>=0;i--) //s 循環. 注意 i 的循環是倒過來的. 我試過順過來的循環, 沒成功.
            {
//System.out.println("j="+j+", i="+i+", res[0]="+res[0]+", res[1]="+res[1]+", res[2]="+res[2]+",
res[3]="+res[3]+", s_i="+s.charAt(i)+", p_j="+p.charAt(j)+", ");
                res[i+1] = res[i]&&(p.charAt(j)=='?'||s.charAt(i)==p.charAt(j));
            }
        }
        else
        {
            int i = 0;
            while(i<=s.length() && !res[i])
                i++;
            for(;i<=s.length();i++)
            {
                res[i] = true;
            }
        }
        res[0] = res[0]&&p.charAt(j)=='*';
```

一维的数组理解起来好吃力啊。。。有一行不明白，请问那个:
res[0] = res[0]&&p.charAt(j)=='*';
是什么意思呢。。。lz 有时间的话提醒一下啊。。
谢啦

回复 toefl784580：这是对 res[0]进行更新， res[0]是 true 的条件就是本来就是 true，并且当前字符还是一个星号， 因为 0 表示目标串没有字符， 能够是 true 的原因就是都是星号， 只要出现字符就会是 false 哈。

tao: res[i]代表 s 的前 i 个字符是否匹配. 以上說的就是 s="", p="******"之情況.

```
    }
    return res[s.length()];
}
```

这里有个问题，就是如果把以上代码直接放到 LeetCode 中去测试，会有最后一个 test case 过不了，说超时了，这道题的 AC 率这么低就是因为这个 case，从难度来说，这个题比 Regular Expression Matching 简单一些。个人觉得这其实是 LeetCode 的问题，把测试超时的槛设得太低了，好像用 C++就能过，因为效率比较高，而 java 可能要抠各种细节，这其实意义不是很大，既然算法复杂度已经到位了，就应该可以过，甚至

觉得时间应该设得更高一些，连同 brute force 也让过，这样方便大家测试一道题的不同解法，至少检验正确性，时间上大家自己去分析就可以。所以如果想要过最后一个 case 可以在代码的第一个 if 以后加上以下两行跳过那个 case(tao: 已加)：

```
if(s.length()>300 && p.charAt(0)=='*' && p.charAt(p.length()-1)=='*')
    return false;
```

tao: 若以上的 println 沒有被 comment out, 且輸入為 s="abc", p="abc", 則輸出為:

```
//System.out.println("j="+j+", i="+i+", res[0]="+res[0]+", res[1]="+res[1]+", res[2]="+res[2]+",
res[3]="+res[3]+", s_i="+s.charAt(i)+", p_j="+p.charAt(j)+", ");
//res[i+1] = res[i]&&(p.charAt(j)=='?'||s.charAt(i)==p.charAt(j));
```

j=0, i=2, res[0]=true, res[1]=false, res[2]=false, res[3]=false, s_i=c, p_j=a,

j=0, i=1, res[0]=true, res[1]=false, res[2]=false, res[3]=false, s_i=b, p_j=a,

j=0, i=0, res[0]=true, res[1]=false, res[2]=false, res[3]=false, s_i=a, p_j=a,

j=1, i=2, res[0]=false, res[1]=true, res[2]=false, res[3]=false, s_i=c, p_j=b,

j=1, i=1, res[0]=false, res[1]=true, res[2]=false, res[3]=false, s_i=b, p_j=b,

j=1, i=0, res[0]=false, res[1]=true, res[2]=true, res[3]=false, s_i=a, p_j=b,

j=2, i=2, res[0]=false, res[1]=false, res[2]=true, res[3]=false, s_i=c, p_j=c,

j=2, i=1, res[0]=false, res[1]=false, res[2]=true, res[3]=true, s_i=b, p_j=c,

j=2, i=0, res[0]=false, res[1]=false, res[2]=false, res[3]=true, s_i=a, p_j=c,

---

45. Jump Game II, Hard

http://blog.csdn.net/linhuanmars/article/details/21356187

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.
For example:
Given array A = [2,3,1,1,4]
The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)
Note:
You can assume that you can always reach the last index.
Subscribe to see which companies asked this question

Key: 由於 E3 代碼比 Code Ganker 的好懂, 以後還是用 E3 代碼. E3 方法為: 將本題 I 的 E3 代碼稍加修改, 即在 vrangh 中找出 reach 最大的那個元素(比如 nums[5])時, 將 numJump++即可(即下一步就是跳到 vrangh 中 reach 最大的那個元素).

Code Ganker 的方法: 貪心法. 本題方法之思想為: 設某一步的(合法)落腳點為數組中的某(連續的)范圍(不要問甚麼范圍, 看了後面的自然就知道了), 則選擇落腳點的原則就是: 在這個范圍中選出一個點(稱此點為 Островский 同志), 使得這個點(以及之前的點)所弄出的 reach 最遠, Островский 同志就是本步的落腳點. 下

一步的落腳點的合法范圍即為[Островский 同志的下一個, ... reach] (當然, 此處的 reach 應當叫 lastReach 了, 因為它是上一步的落腳點日出來的), 然後同樣地, 在此范圍找出一個本步的落腳點, 它使得這個點(以及之前的點)所弄出的 reach 最遠(注 1)... ...

注 1：實踐中, 找本步的落腳點的方法為: i 在這個范圍中掃描, 而 reach = Math.max(reach, A[i] + i)自動就能找出本步落腳點日出來的 reach, 並自動將其值保存在 reach 變量中. 然後當 i 大於 lastReach 時, 便開始下一步(step++並更新 lastReach). 實際中, i 的掃描就用一個 for 循環就可以了, 循環條件跟本題的 I 一模一樣.

注意本題仍有走不到數組最後的情況, 此時應返回 0.

History:
E1 直接看的答案.
E2 沒做出來, 主要是因為 E1 沒寫 key. 但 E2 看了答案後, 也花了好一會兒才看懂. 以上 key 是 E2 寫的.
E3 是將本題 I 的 E3 代碼稍加修改寫出來的, 最開始把 numJumps++寫到 for 循環裡面去了, 改後即通過. 由於 E3 代碼比 Code Ganker 的好懂, 以後還是用 E3 代碼. 方法見 key 中.

这道题是 Jump Game 的扩展，区别是这道题不仅要看能不能到达终点，而且要求到达终点的最少步数。其实思路和 Jump Game 还是类似的，只是原来的全局最优现在要分成 step 步最优(tao: 即 reach)和 step-1 步最优(tao: 即 lastReach)（假设当前步数是 step）。当走到超过 step-1 步最远的位置时(tao: 即代碼中的 if(i>lastReach))，说明 step-1 不能到达当前一步，我们就可以更新步数，将 step+1。时间复杂度仍然是 O(n)，空间复杂度也是 O(1)。代码如下. 动态规划是面试中特别是 onsite 中非常重要的类型，一般面试中模型不会过于复杂，所以大家可以熟悉一下比较经典的几个题，比如 Jump Game，Maximum Subarray 等。

感觉楼主的算法非常简单，但这个还有 jump game 都用的是贪心法而不是动态规划啊。我一开始用动态规划，算的是要达到某点的最短步数，结果需要之前所有能跳到这个点的步数，计算起来就超时了。
另外假设题目要求最短步数的具体解法，似乎就没法写这么简单了，我用的也是贪心法，点 i 的 reach 范围就是 i+A[i]，在 i 的 reach 范围内找 reach 范围最大的点，这个点就一定是最短步数的跳转点，继续找下去，这样能构建最短解法，但扫描的时候会有点重复，虽然还应该是常数级的 O(n).

回复 dracodoc：恩，确实~ 大家普遍说这个问题都是说贪心解~ 不过这里我觉得取决于我怎么定义递推量~ 事实上这里贪心的主要步奏就在于用 reach 来跳跃，所以如果我动态规划的递推量就定义 reach 包含那个步奏~ 我觉得说是动态规划也未尝不可哈~

Code Ganker 的代碼(不用):

```
public int jump(int[] A) {
    if(A==null || A.length==0)
        return 0;
    int lastReach = 0;
    int reach = 0;
    int step = 0;

    for(int i=0;i<=reach&&i<A.length;i++)
    {
        if(i>lastReach)
        {
            step++;
            lastReach = reach;
        }
        reach = Math.max(reach,A[i]+i); //E2.5: 歷史上最大的那個 reach 對應的 A[i]就是 Островский 同志, 當之後的某個 i 在 Островский 同志日出的 reach 之外時, 就要 jump 了(step++).
    } //End of for
```

```
    //别忘了以下的 if
    if(reach<A.length-1)
        return 0;
    return step;
}
```

E3 的代碼(用它):

```
    public int jump(int[] nums) {
        if(nums == null || nums.length <= 1) return 0;
        int n = nums.length;
        int[] reach = new int[n];
        for(int i = 0; i < n; i++) reach[i] = nums[i] + i;

        int i = 0, maxReach = 0, numJumps = 0;

        while(i < n) {
            if(reach[i] >= n - 1) return (numJumps + 1);
            int iNext = 0;
            for(int j = i + 1; j <= reach[i] && j < n; j++) {
                if(reach[j] > maxReach) {
                    maxReach = reach[j];
                    iNext = j;
                }
            }
            i = iNext;
            numJumps++;
            maxReach = 0;
        }

        return 0; // 實踐表明, 此句隨便返回甚麼值都能通過, 因為題目中說了 you can always reach the last index, 所以並不會執行到此句處.
    }
```

46. Permutations, Medium

http://blog.csdn.net/linhuanmars/article/details/21569031

Given a collection of **distinct** numbers, return all possible permutations.
For example,
[1,2,3] have the following permutations:
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].
Subscribe to see which companies asked this question

key: item-res 模版遞歸法. 寫一個 void helper(int[] num, boolean[] used, ArrayList<Integer> item, List<List<Integer>> res), 其作用為(即維護這麼一個不變量): 調用 helper(num, used, item, res)前(稱為 A 時刻), num 中有一些(或沒有)元素的 used 為 true, item 是這些為 used 為 true 的元素的一個(為何是一個? 見下面注)可能的排列. 而 helper(num, used, item, res)之作用即為: 將 A 時刻那些 used 為 false 的元素 的所有可能的排列都加到 item 後面去(當 item 中元素個數等於 num 中元素個數時, 將 item 復製一份加入到 res 中

去), 然後將 item 恢復原狀 保護現場 (即 A 時刻的樣子, 即將 item 後面加入的那些 A 時刻那些 used 為 false 的元素都刪掉, 並將 used 變組也恢復為 A 時刻的樣子). helper 中調用自己的方式為: 遍歷 '所有 A 時刻的 used 為 false 的元素', 取一個這樣的元素, 將其 used 設為 true, 然後將此元素加到 item 中, 然後調用 helper(num, used, item, res), 最後保護現場(從 item 中刪此元素, 並將此元素的 used 設為 fase).

[注] 為何此時只需要 item 為這些為 used 為 true 的元素的一個(而不是所有)可能的排列? 因為沒法做到讓 item 為所有可能的排列, 也沒必要那樣去做. 實際上 'item 為這些為 used 為 true 的元素的一個(而不是所有)可能的排列' 不是 helper 的功能, 而是 helper 在执行前所需要的狀態(當然 helper 在执行後, 也要保證還是這個狀態, 故准確地說, 應該叫 helper 要維持這麼一個不變量, 易知 helper 在执行後, 此不變量是成立的, 因為我們做了保護現場). 而 'helper 在执行前所需要的此狀態' 是很容易滿足的, 我們想想, helper 是在甚麼時候被調用的? 一是在 permute 函數中, 此時 num 中沒有 used 為 true 的元素, 且 item 為空, 故此狀態自然滿足. 二是在 helper 中調用自己, 由於每次 helper 做了保護現場, 做此狀態也滿足.

後面這類型的排列組合題都是用類似的 item-res 遞歸法, 代碼比較相似, 本題和 77. Combinations 題可以作為兩個代表模版記住.

History:
E1 直接看的答案.

E2 很快寫好, 本來可以一次通過, 給果在給 helper 傳 res 實參時, 我不小心新建了一個 List<List<Integer>> 傳进去, 導致結果沒放在 res 中. 改後即通過. 上面的 key 都是 E2 寫的.

E3 改了一次後通過. E3 的方法沒 Code Ganker 的方法好, E3 是寫一個 List<List<Integer>> helper(int[] nums, int start), 其作用為返回 nums[start, ...]這個子數組的所有 permutation. 在 helper 中調用 helper(nums, start + 1), 然後在得到的結果中, 每次取一個 List, 將 nums[start]插入此 List 中的每一個鏠, 所有插法組成的結果即為 helper(nums, start)之返回值. 在 permute 函數中調用 helper(nums, 0)即得到最終結果. 這樣做不好的地方在於 要在 List 中間插入一個數, 不太好. 此方法可以將遞歸改寫為循環的, 但由於此方法不好, 故沒改寫. E3 沒想出 Code Ganker 的方法, 原因是我 E2 的 key 對 Code Ganker 的方法根本沒理解清楚, Code Ganker 的方法是 E3 看了 Code Ganker 代碼後才理解清楚的, 故 E3 重寫了 key. E3 又默寫了一遍 Code Ganker 的代碼.

这道题跟 N-Queens，Sudoku Solver，Combination Sum，Combinations 等一样，也是一个 NP 问题。方法还是原来那个套路，还是用一个循环递归处理子问题。区别是这里并不是一直往后推进的，前面的数有可能放到后面，所以我们需要维护一个 used 数组来表示该元素是否已经在当前结果中，因为每次我们取一个元素放入结果，然后递归剩下的元素，所以不会出现重复。时间复杂度还是 NP 问题的指数量级。代码如下.

注意在实现中有一个细节，就是在递归函数的前面，我们分别设置了 used[i]的标记，表明改元素被使用，并且把元素加入到当前结果中，而在递归函数之后，我们把该元素从结果中移除，并把标记置为 false，这个我们可以称为“保护现场”，递归函数必须保证在进入和离开函数的时候，变量的状态是一样的，这样才能保证正确性。当然也可以克隆一份结果放入递归中，但是那样会占用大量空间。所以个人认为这种做法是最高效的，而且这种方法在很多题目（几乎所有 NP 问题）中一直用到，不熟悉的朋友可以练习一下哈。
对于 NP 问题我们一直都是用递归方法，也是一个很成熟的套路，可以举一反三。不过有时候面试官会刻意让我们实现一下迭代的做饭，所以这里我们就介绍一下迭代的实现方法。(tao: 迭代的实现方法 ignored)

这种 NP 问题的求解在 LeetCode 中非常常见，类似的有 N-Queens，Sudoku Solver，Combination Sum，Combinations，不过思路差不多，掌握套路就不难了。这道题还有一个扩展就是如果元素集合中会出现重复，那么意味着我们需要跳过一些重复元素，具体的细节可以参见 Permutations II。

LZ, leetcode 原题返回值是 List<List<Integer>>,是否应该?
Line 1. public List<List<Integer>> permute(int[] num) {

Line 2. List<List<Integer>> res = new ArrayList<List<Integer>>();
Line 10: helper(int[] num, boolean[] used, ArrayList<Integer> item, List<List<Integer>> res)
当然，返回值变成 ArrayList 可以通过。但违背对接口编程的原则。

回复 jingyuzhang：原版的 leetcode 是用 ArrayList 作为返回值的， 最近新的改版之后才都变成 List 的抽象接口的哈 ~ 我就没一一改过来了，因为很多题目都是这样的 ~ 其实原理是一样的， 就按照你写的修复一下就行哈 ~

(tao: 我已在下面的代碼中按上面說的改成了 List, leetcode 中可以通過)

```
public List<List<Integer>> permute(int[] num) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(num==null || num.length==0)
        return res;
    helper(num, new boolean[num.length], new ArrayList<Integer>(), res);
    return res;
}

private void helper(int[] num, boolean[] used, ArrayList<Integer> item, List<List<Integer>> res)
{
    if(item.size() == num.length)
    {
        res.add(new ArrayList<Integer>(item));
        return;
    }

    for(int i=0;i<num.length;i++)
    {
        if(!used[i])
        {
            used[i] = true;
            item.add(num[i]);
            helper(num, used, item, res);
            item.remove(item.size()-1); //保護現場
            used[i] = false; //保護現場
        }
    }
}
```

---

47. Permutations II, Hard

http://blog.csdn.net/linhuanmars/article/details/21570835

Given a collection of numbers that might contain duplicates, return all possible unique permutations.
For example,
[1,1,2] have the following unique permutations:
[1,1,2], [1,2,1], and [2,1,1].
Subscribe to see which companies asked this question

Key: item-res 模版遞歸法. 只比上題多了兩句:
Arrays.sort(num);

if(i>0 && !used[i-1] && num[i]==num[i-1]) continue;
以上 used[i-1]為 true(不是 false 嗎?)時 表示第 i-1 個元素已經加到 item 中了, 為了避免重復, 就要跳過第 i 個元素.

History:
E1 直接看的答案.
E2 很快寫好, 一次通過. 多加的那句 if 的條件中跟 Code Ganker 有點不同. 我的是 used[i-1], 而 Code Ganker 的是!used[i – 1], 我試過, 兩者都可以通過, 但我的更好理解, 所以用我的.
E3 先將本題 I 的代碼默寫好(之前在 I 中也默寫了一遍), 然後跟 I 的答案對比, 發現不小心將 used[i] = true 寫成了 used[nums[i]] = true. 改後, 並添了兩句, 即通過. E3 添的第二句跟 Code Ganker 稍有不同, 見 Code Ganker 代碼中.

`tao: 我同樣對 ArrayList → List 做了相應改動.`

这个题跟 Permutations 非常类似，唯一的区别就是在这个题目中元素集合可以出现重复。这给我们带来一个问题就是如果不对重复元素加以区别，那么类似于{1,1,2}这样的例子我们会有重复结果出现。那么如何避免这种重复呢？方法就是对于重复的元素循环时跳过递归函数的调用，只对第一个未被使用的进行递归，我们那么这一次结果会出现在第一个的递归函数结果中，而后面重复的会被略过。如果第一个重复元素前面的元素还没在当前结果中，那么我们不需要进行递归。想明白了这一点，代码其实很好修改。首先我们要对元素集合排序，从而让重复元素相邻，接下来就是一行代码对于重复元素和前面元素使用情况的判断即可。代码如下.

这样的解法是带有一般性的，把这个代码放到 Permutations 中也是正确的，所以如果熟悉的话，面试时如果碰到这个题目建议直接实现这个代码，不要假设元素没有重复，当然可以跟面试官讨论，不过一般来说都是要考虑这个情况的哈。

```
public List<List<Integer>> permuteUnique(int[] num) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(num==null && num.length==0)
        return res;
    Arrays.sort(num); //多出的句子 1/2
    helper(num, new boolean[num.length], new ArrayList<Integer>(), res);
    return res;
}
private void helper(int[] num, boolean[] used, ArrayList<Integer> item, List<List<Integer>> res)
{
    if(item.size() == num.length)
    {
        res.add(new ArrayList<Integer>(item));
        return;
    }
    for(int i=0;i<num.length;i++)
    {
        if(i>0 && used[i-1] && num[i]==num[i-1]) continue; //多出的句子 2/2. used[i-1]為 true 時 表示第 i-1 個元素已經加到 item 中了, 為了避免重復, 就要跳過第 i 個元素. 原代碼為 !used[i-1], 可通過, 但不太好懂, 我去掉了!, 也可通過, 但更好懂. 其實改了後也不好懂, 還是看下面我 E3 添的那句, 好懂些.
        if(!used[i])
        {
            used[i] = true;
            item.add(num[i]);
```

```
            helper(num, used, item, res);
            item.remove(item.size()-1);
            used[i] = false;
            //while(i < nums.length - 1 && nums[i] == nums[i + 1]) i++; E3 是在這裡添的這麼一句, 此句跟前面 Arrays.sort(num)一起, 也能通過. 例如對於輸入為[1, 2, 2, 2, 3], 對第一個 2 作了遞歸後, 後面的 2 就不必再做遞歸了, 要跳過.
        }
    }
}
```

---

48. Rotate Image, Medium

http://www.lifeincode.net/programming/leetcode-rotate-image-java/

You are given an *n* x *n* 2D matrix representing an image.
Rotate the image by 90 degrees (clockwise).
Follow up:
Could you do this in-place?
Subscribe to see which companies asked this question

Key: (x, y) → (y, n – x – 1) → (n – x – 1, n – y – 1) → (n – y – 1, x) → (x, y). 此式意思為:
1. 將 matrix[x][y]這個元素移到 matrix[y][n – x – 1]處 (即將 matrix[x][y]的值賦給 matrix[y][n – x – 1]),
2. 將 matrix[y][n – x – 1]這個元素移到 matrix[n – x – 1][n – y – 1]處,
3. 將 matrix[n – x – 1][n – y – 1]這個元素移到 matrix[n – y – 1][x]處,
4. 將 matrix[n – y – 1][x]這個元素移到 matrix[x][y]處,
實際中, 賦值時順序 按以上順序倒過來(4321) 賦比較方便. 這樣一次移四個點, 所以循環時不是每個點都要弄到, 故要注意循環的起始點.

History:
E1 一次通過了, 也想到了 (x, y) → (y, n – x – 1). 但我的不是 in place 的, 我是把 matrix 按(x, y) → (y, n – x – 1)弄到了一個新數組中, 再把新數組拷到 matrix 中. 所以以下不用我的代碼.
E2 通過.
E3 沒多久就寫好, 本來可以一次通過的, 但在賦值時最後一句忘了 matrix[y][n - 1 – x] = temp 不小心寫成了 matrix[y][n - 1 - x] = matrix[x][y]. 改後即通過. E3 的代碼比答案的簡明, 故以後用 E3 的代碼.

Code Ganker 的方法跟以下一模一樣, 但他的解釋不太好懂, 評論中的人也說看不懂. 由此可見 Code Ganker 的代碼也不都是原創, 因為只有可能是 Code Ganker 抄以下代碼(中抄英), 不可能英抄中(英看不懂中). 但 Code Ganker 很多題會講些思路, 心得, 面試經驗甚麼的, 還是比較用有, 以後還是以 Code Ganker 為主.

Wiki for in-place: 在計算機科學中，一個**原地算法（in-place algorithm）**是一種使用小的，固定數量的額外之空間來轉換資料的算法。當算法執行時，輸入的資料通常會被要輸出的部份覆蓋掉。不是原地算法有時候稱為**非原地（not-in-place）**或**不得其所（out-of-place）**。

If we want to do this in-place, we need to find the regular pattern of rotating a point. We need to modify four points. For example, we have the matrix like the following.
1 2 3 4 5
6 7 8 9 10

11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
Now we want to rotate “2”, the element we need to move is “2”, “10”, “24”, “16”. Like the following.

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
The position of them is (0, 1), (1, 4), (4, 3), (3, 0).
In fact, for any point (x, y), the affected point is (y, n – x – 1), (n – x – 1, n – y – 1) (Tao: 繼續使用(x, y) → (y, n – x – 1)將(y, n – x – 1)旋轉得出的, 這四個點每個都是由上個點旋轉得出的), (n – y – 1, x), in which “n” is the size of the matrix. In other words, we need to move point like (x, y) -> (y, n – x – 1) -> (n – x – 1, n – y – 1) -> (n – y – 1, x) -> (x, y).
However, every time we visit a point, we move 4 points. So we need to decide which points we need to visit.
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
When we process (0, 0), the last element in the first line is moved. In fact, for the first line, we only need to process (0, 0) to (0, n – 2) (Tao: 即最後一個不用移). The affected points are like the following.
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
For the second line, we need to start from the second element. And ends one element earlier comparing to the first line, which is from (1, 1) to (1, n – 3).
In conclusion, for the ith line, we start from (i, i), and ends at (i, n – 2 – i).
We don’t need to process every line. We only need process half of them. Because it will affect the other half. But we need to be careful about the parity of the number of line.
The complexity is O(n2).

以下代碼不用:

```
public class Solution {
    public void rotate(int[][] matrix) {
        int n = matrix.length;
        int halfN;
//以下幾句有點 redundant. 其實可以不用判斷 n 的奇偶. 全部用 halfN = n / 2 就可以, Leetcode OJ 上能通過. 如果一定要判斷 n 的奇偶, 也可以寫為更簡潔的 int halfN = (n % 2 == 0 ? n / 2 : n / 2 + 1), leetcode 上可以通過.
        if (n % 2 == 0)
            halfN = n / 2;
        else
            halfN = n / 2 + 1;
```

```
        for (int i = 0; i < halfN; i++) {
            for (int j = i; j < n - i - 1; j++) {
                int tmp = matrix[i][j];
                matrix[i][j] = matrix[n - j - 1][i];
                matrix[n - j - 1][i] = matrix[n - i - 1][n - j - 1];
                matrix[n - i - 1][n - j - 1] = matrix[j][n - i - 1];
                matrix[j][n - i - 1] = tmp;
            }
        }
    }
}
```

E3 的代碼(用它):

```
    public void rotate(int[][] matrix) {
        if(matrix == null || matrix.length <= 1) return;
        int n = matrix.length;
        int nLayer = n / 2;

        for(int layer = 0; layer < nLayer; layer++)
            for(int j = layer; j <= n - 2 - layer; j++)
                rotateOne(matrix, layer, j);
    }

    //以下函數名字叫 rotateOne, 實際上是移動四個元素
    private void rotateOne(int[][] matrix, int x, int y) {
        int n = matrix.length;
        int temp = matrix[x][y];
        matrix[x][y] = matrix[n - 1 - y][x];
        matrix[n - 1 - y][x] = matrix[n - 1 - x][n - 1 - y];
        matrix[n - 1 - x][n - 1 - y] = matrix[y][n - 1 - x];
        matrix[y][n - 1 - x] = temp;
    }
```

---

49. Group Anagrams, Medium

http://blog.csdn.net/linhuanmars/article/details/21664747

Given an array of strings, group anagrams together.
For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],
Return:

```
[
  ["ate", "eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

Note:

1.For the return value, each *inner* list's elements must follow the lexicographic order.

2.All inputs will be in lower-case.
Subscribe to see which companies asked this question

Key: 如果用 242 題的 hashmap 的方法，然后两两匹配，再分组会比较麻烦(E3 實踐表明這樣寫會超時)。更好的方法是建立一个 HashMap<String, ArrayList<String>>, 對輸入數組中的每一個 string 都先排序, 排序后的 string 作为一个 key，而 value 是所有属于这个 key 类的 string，这样就可以比较简单的进行分类。 sort 單個 String 的方法是: 先將這個 String 轉化為 char array, 然後對 char array sort, 然後再轉化回 String. 注意可以放心地用 map.containsKey(String), 它是直接比較 string 是不是相等(即字符是否相同), 而不是比較是 string 的引用是不是相同(已 record 到 Java 書).對 map 的遍歷可以用 map.keySet().

History:
E1 直接看的答案.
E2 本來可以一次通過的, 但忘了對最後結果 sort, 改後即通過(E3 沒 sort, 也通過了), 以上的 key 基本都是 E2 寫的.
E3 最開始被 242. Valid Anagram 題中 History 的 E1 中的「 我看了 49. Group Anagrams 中 Code Ganker 講的算法後, 自己的代碼很快寫好並通過. 」一句話誤導了, 以為本題還是要像 242 題中那樣將每個 string 轉化成一個 map 來比, 即調用 242 題中的 isAnagram 函數, 將每個 string 拿去跟結果中每個 list 的第一個 string 比較是否為 anagram, 若是則加入. 我按這樣很快寫好, 但在 OJ 中對大輸入超時了(在我電腦上沒改就能對題目中的例子給出正確結果). 後來想到了本題 key 中方法, 但要 sort 每一個 string, 覺得這樣的時間複雜度也挺高的, 就沒打算寫, 正打算看答案, 就看到了本題 key 中的 HashMap<String, ArrayList<String>>和排序一詞, 才知道這個方法是被 OJ 接受的, 然後按這樣寫, 很快寫好, 一次通過. 由於我最初被誤導(242 題中那句話已刪), 並且也想到了正確方法, 所以本題還是算我做出來了.

我: 注意 E2 時題目可能變化了, Code Ganker 的代碼並不是按最新要求寫的(見我在代碼前說的), 以後都用 E2 的代碼. E2 對 map 的遍歷沒像 Code Ganker 那樣用 iterator, 而是用的 map.keySet(), which is more 順手 to use, 以後我也都這樣用, 但 Code Ganker 代碼中用 iterator 遍歷的方法也看看, 好熟悉一下 iterator 的操作.

这是一道很经典的面试题了，在 cc150 里面也有，就是把一个数组按照易位构词分类。易位构词其实也很好理解，就是两个单词所包含的字符和数量都是一样的，只是顺序不同。
这个题简单的版本是判断两个单词是不是 anagram，一般来说有两种方法。第一种方法是用 hashmap，key 是字符，value 是出现的次数，如果两个单词构成的 hashmap 相同，那么就是 anagram。实现起来就是用一个构建 hashmap，然后另一个在前面的 hashmap 中逐个去除，最后如果 hashmap 为空，即返回 true。这个方法时间复杂度是 O(m+n)，m，n 分别是两个单词的长度。而空间复杂度是 O(字符集的大小)。第二种方法是将两个单词排序，如果排序之后结果相同，就说明两个单词是 anagram。这种方法的时间复杂度取决于排序算法，一般排序算法是 O(nlogn)，如果字符集够小，也可以用线性的排序算法。不过总体来说，如果是判断两个单词的，第一种方法要直接简单一些。
接下来我们看看这道题，是在很多字符串里面按照 anagram 分类，如果用 hashmap 的方法，然后两两匹配，在分组会比较麻烦。而如果用排序的方法则有一个很大的优势，就是排序后的字符串可以作为一个 key，也就是某一个 class 的 id，如此只要对每一个字符串排序，然后建立一个 hashmap，key 是排序后的串，而 value 是所有属于这个 key 类的字符串，这样就可以比较简单的进行分类。假设我们有 n 个字符串，字符串最大长度是 k，那么该算法的时间复杂度是 O(nklogk)，其中 O(klogk)是对每一个字符串排序（如果用线性算法也可以提高）。空间复杂度则是 O(nk)，即 hashmap 的大小。实现代码如下.
理清了思路，实现起来还是比较简单的，这道题考察排序，hashmap，字符串处理，比较全面，在面试中非常常见，大家一定要熟悉哈。

我: 以下是 Code Ganker 的代碼, 注意題目可能有變化了, E2 時的題目要求返回的是 List<List<String>>, 而以下代碼返回的是 ArrayList<String>. 而且題目要求返回的 List<List<String>>中的每一個 List<String>

都是 sorted, 而以下代碼沒有 sort (代碼中那個 Arrays.sort(charArr)不是幹這件事的).

```
public ArrayList<String> anagrams(String[] strs) {
    ArrayList<String> res = new ArrayList<String>();
    if(strs == null || strs.length == 0)
        return res;
    HashMap<String, ArrayList<String>> map = new HashMap<String, ArrayList<String>>();
    for(int i=0;i<strs.length;i++)
    {
        char[] charArr = strs[i].toCharArray();
        Arrays.sort(charArr);
        String str = new String(charArr);
        if(map.containsKey(str))
        {
            map.get(str).add(strs[i]);
            //注意此用法, 參見 Java 書 p834. 注意 map.get(str)是一個 ArrayList<String>, 因為前面的定義為
              HashMap<String, ArrayList<String>> map = new HashMap<String, ArrayList<String>>();
        }
        else
        {
            ArrayList<String> list = new ArrayList<String>();
            list.add(strs[i]);
            map.put(str,list);
        }
    }
    Iterator iter = map.values().iterator();  //注意 iter.next()返回的是一個 ArrayList, 因為 map 的各個 value 就是 ArrayList.
    //見 Java 書 p834 for values(), p787 for Iterator Interface, p787 頂 for iterator(),
    while(iter.hasNext()) //注意不要寫成 if(iter.hasNext())了
    {
```

```
        ArrayList<String> item = (ArrayList<String>)iter.next();
        if(item.size()>1)
            res.addAll(item);
      //注意是 addAll, 而不是 add, 因為我們要加入的是 item 的元素, 而不是 item 自已 (addAll(c)見 Java 書 p787)
    }
    return res;
}
```

以下是我 E2 的代碼:

```
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> res = new ArrayList<List<String>>();
        if(strs == null || strs.length == 0)
            return res;

        Map<String, List<String>> map = new HashMap<String, List<String>>();

        for(String s : strs) {
            char[] ca = s.toCharArray();
            Arrays.sort(ca);
            String k = new String(ca);

            if(!map.containsKey(k)) { //由此可知 map.containsKey(String)是直接比較 string 是不是相等(即字符是否相同), 而不是比較是 string 的引用是不是相同.
                map.put(k, new ArrayList<String>());
                map.get(k).add(s);
            } else {
                map.get(k).add(s);
            }

        }

        for(String k : map.keySet()) {
            List<String> item = map.get(k);
            Collections.sort(item); //E3 沒 sort, 也通過了.
            res.add(item);
        }
```

```
    return res;
  }
```

50. Pow(x, n), Medium

http://blog.csdn.net/linhuanmars/article/details/20092829

Implement pow(*x*, *n*).
Subscribe to see which companies asked this question

Key: 遞歸調用 myPow(x, n/2). 注意 n 分奇偶. 注意 n 可為負, 此時可按 myPow(x, -n)算, 但注意 n = Integer.MIN_VALUE 時, -n 會越界, 故要專門處理 n = Integer.MIN_VALUE 之情況.

History:
E1 是一個一個乘的, 在 leetcode OJ 中超時了.

E2 幾分鍾寫好, 但我的代碼用的 if(n < 0) return 1.0 / myPow(x, -n), 在輸入的 n 為 Integer.MIN_VALUE 時 -n 越界了. 後來專門處理了 n 為 Integer.MIN_VALUE 時之情況, 才通過. 注意 Code Ganker 的代碼不用專門處理這個情況, 原因是他從頭到尾都沒用-n.

E3 在 OJ 上幾分鍾寫好, 也是 n = Integer.MIN_VALUE 時沒通過, 專門處理了後即通過. 雖然 Code Ganker 沒專門處理 n = Integer.MIN_VALUE 的情況, 但其代碼其實沒有 E3 好懂. 故以後還是用 E3 代碼.

Code Ganker:
这道题是一道数值计算的题目，因为指数是可以使结果变大的运算，所以要注意越界的问题。如同我在Sqrt(x)这道题中提到的，一般来说数值计算的题目可以用两种方法来解，一种是以 2 为基进行位处理的方法，另一种是用二分法。这道题这两种方法都可以解，下面我们分别介绍。
第一种方法在 Divide Two Integers 使用过，就是把 n 看成是以 2 为基的位构成的，因此每一位是对应 x 的一个幂数，然后迭代直到 n 到最高位。比如说第一位对应 x，第二位对应 x*x,第三位对应 x^4,...,第 k 位对应 x^(2^(k-1)),可以看出后面一位对应的数等于前面一位对应数的平方，所以可以进行迭代。因为迭代次数等于 n 的位数，所以算法的时间复杂度是 O(logn)。代码如下(我: 太長, 已省略)：

以上代码中处理了很多边界情况，这也是数值计算题目比较麻烦的地方。比如一开始为了能够求倒数，我们得判断倒数是否越界，后面在求指数的过程中我们也得检查有没有越界。所以一般来说求的时候都先转换为正数，这样可以避免需要双向判断（就是根据符号做两种判断）。
接下来我们介绍二分法的解法，如同我们在 Sqrt(x)的方法。不过这道题用递归来解比较容易理解，把 x 的 n 次方划分成两个 x 的 n/2 次方相乘，然后递归求解子问题，结束条件是 n 为 0 返回 1。因为是对 n 进行二分，算法复杂度和上面方法一样，也是 O(logn)。代码如下：

Code Ganker 的代碼(不用):

```
double pow(double x, int n) {
    if (n == 0) return 1.0;
    double half = pow(x, n/2); //注意一開始就將 n 除以 2, 且從頭到尾都沒用-n, 這樣的好處是 n 為 Integer.MIN_VALUE 時沒有-n 越界的問題.
    if (n%2 == 0)
    {
```

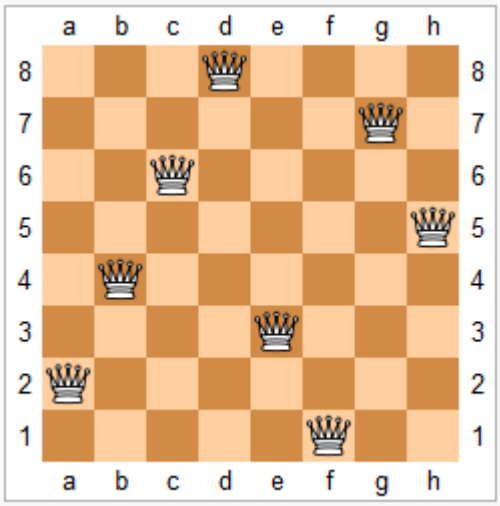
```
        return half*half; //n 為 負偶數 的情況也被本式算了.
    }
    else if (n>0)
    {
        return half*half*x;
    }
    else
    {
        return half/x*half; //n 為 負奇數 的情況
    }
}
```

以上代码比较简洁，不过这里有个问题是没有做越界的判断，因为这里没有统一符号，所以越界判断分的情况比较多，不过具体也就是在做乘除法之前判断这些值会不会越界，有兴趣的朋友可以自己填充上哈，这里就不写太啰嗦的代码了。不过实际应用中健壮性还是比较重要的，而且递归毕竟会占用递归栈的空间，所以我可能更推荐第一种解法。

E3 的代碼(用它):

```
    public double myPow(double x, int n) {
        if(n == 0) return 1.0;

        if(n == Integer.MIN_VALUE) {
            double halfPower = myPow(x, n / 2);
            return halfPower * halfPower;
        }

        if(n < 0) return 1.0 / myPow(x, -n);

        double halfPower = myPow(x, n / 2);

        if(n % 2 == 0) return halfPower * halfPower;
        else return halfPower * halfPower * x;
    }
```

51. N-Queens, Hard



One solution to the eight queens puzzle

http://blog.csdn.net/linhuanmars/article/details/20667175

The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.
Each solution contains a distinct board configuration of the *n*-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.
For example,
There exist two distinct solutions to the 4-queens puzzle:

```
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

Subscribe to see which companies asked this question

Key: 規則就是要求 每個皇后所在的橫豎斜上(不只是一步, 而是整個橫竖斜大直線上)都沒有其它皇后.

NP 間題, 直接 brute force. 用 columnForRow[row]表示第 row 行的皇后在第 columnForRow[row]列(因為每行只能有一個皇后).

寫一個 check(int row, int[] columnForRow)來檢查第 row 行的皇后是否合法, 即是否與所有其它行的皇后沖突(要檢查橫豎兩條再加兩條斜的). check 中用 if(columnForRow[row]==columnForRow[i] || Math.abs(columnForRow[row]-columnForRow[i])==row-i)

寫一個 helper(int n, int row, int[] columnForRow, List<List<String>> res)把 columnForRow 這個數組從第 row 個到第 n 個元素填好, 當然填的都是合法的東西, 然後把它轉換成題目要求的棋盤的形式放入 res 中. 遞歸調用 helper.
注意題目中的栱盤大小是 n×n, 而不是固定的 8×8.

History:
E1 直接看的答案.

E2 改了很多遍才通過, 代碼跟答案差不多, 但 E2 另外還專門寫了一個函數來把已填好的 columnForRow 轉換成題目要求的棋盤的形式放入 res 中. 以上 key 中 check 那句 if 是 E2 加的, 之前沒有. E2 的 check 寫得比答案麻

煩.

E3 改了很多遍才通過, 方法跟答案一樣, 代碼稍有不同, 我的 check 函數寫得沒答案好.

以下對題意的解釋來自 http://www.cnblogs.com/springfor/p/3870944.html
这道题很经典，网上有很多讲解实例。
在国际象棋中，皇后最强大，可以横竖斜的走（感谢 AI 课让我稍微对国际象棋了解了一下）。而八皇后问题就是让 8 个皇后中的每一个的横竖斜(tao: 不只是一步, 而是整個橫竖斜大直線上)都没有其他皇后，这样其实感觉是一种和棋的状态。。不知道说的对不对。。。毕竟真实棋盘中不能放 8 个皇后。。
额。。这道题是 N 皇后，我说着说着就说到了 8 皇后去了。。其实是一样的。。。我这道题的解法也是参考了网上的一些人的讲法。经典的 DFS 递归回溯解法，大体思路就是对每一行，按每一列挨个去试，试到了就保存结果没试到就回溯。

Code Ganker:
N 皇后问题是非常经典的问题了，记得当时搞竞赛第一道递归的题目就是 N 皇后。因为这个问题是典型的 NP 问题，所以在时间复杂度上就不用纠结了，肯定是指数量级的。下面我们来介绍这个题的基本思路。
主要思想就是一句话：用一个循环递归处理子问题。这个问题中，在每一层递归函数中，我们用一个循环把一个皇后填入对应行的某一列中，如果当前棋盘合法，我们就递归处理先一行，找到正确的棋盘我们就存储到结果集里面。
这种题目都是使用这个套路，就是用一个循环去枚举当前所有情况，然后把元素加入，递归，再把元素移除，这道题目中不用移除的原因是我们用一个一维数组去存皇后在对应行的哪一列，因为一行只能有一个皇后，如果二维数组，那么就需要把那一行那一列在递归结束后设回没有皇后，所以道理是一样的。
这道题最后一个细节就是怎么实现检查当前棋盘合法性的问题，因为除了刚加进来的那个皇后，前面都是合法的，我们只需要检查当前行和前面行是否冲突即可。检查是否同列很简单，检查对角线就是行的差和列的差的绝对值不要相等就可以。代码如下. 这道题实现的方法是一个非常典型的套路，有许多题都会用到，基本上大部分 NP 问题的求解都是用这个方式，比如 Sudoku Solver，Combination Sum，Combinations，Permutations，Word Break II，Palindrome Partitioning 等，所以大家只要把这个套路掌握熟练，那些题就都不在话下哈。

Tao: 原代碼是 ArrayList<String[]> solveNQueens(int n), 在 leetcode 中通不過, 我將其改成了 leetcode 要求的 List<List<String>>, 其它地方也作了相應修改, leetcode 可以通過了. 其它人的代碼好像也是 ArrayList<String[]>, 說明應改是 leetcode 的要求後來改成的 List<List<String>>.
為何是 List<List<String>>, 而不是<List<String>? 因為<List<String>只能保存一個棋盤, 而此間題可能有多個解, 所以把所有解的棋盤都放入 List<List<String>>中.

```
public List<List<String>> solveNQueens(int n) {
  List<List<String>> res = new ArrayList<List<String>>();
    helper(n,0,new int[n], res);
    return res;
}
//以下 row 表示在第 row 行放一個皇后(每行只能放一個皇后), columnForRow[row]表示第 row 行的皇后在第 columnForRow[row]列
//helper 的作用就是把 columnForRow 這個數組從第 row 行到第 n 行填好, 當然填的都是合法的東西, 然後把它轉換成題目要求的棋盤的形式放入 res 中 (本句話居然是 E1 寫的)
private void helper(int n, int row, int[] columnForRow, List<List<String>> res)
{
  if(row == n)
  {
   List<String> item = new ArrayList<String>();
```

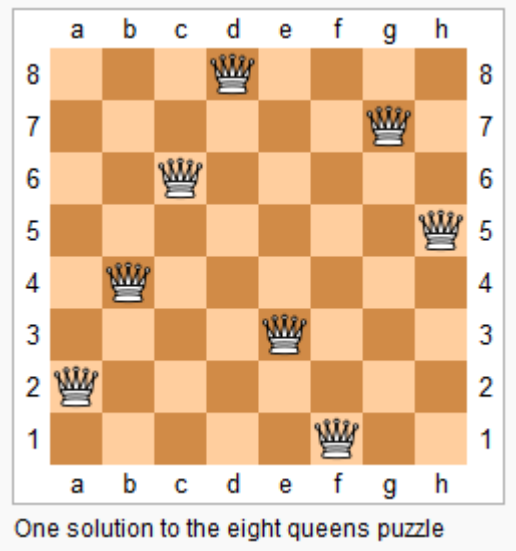
```
        //以下的循環是把已填好的 columnForRow 轉換成題目要求的棋盤的形式放入 res 中
        for(int i=0;i<n;i++) //i 代表行
        {
            StringBuilder strRow = new StringBuilder();
            for(int j=0;j<n;j++) //i 代表列
            {
                if(columnForRow[i]==j)
                    strRow.append('Q');
                else
                    strRow.append('.');
            }
             item.add(strRow.toString());
        }
        res.add(item);
        return;
    }
 //以下的循環是把 columnForRow 這個數組從第 row 行到第 n 行填好, 當然填的都是合法的東西
    for(int i=0;i<n;i++)
    {
        columnForRow[row] = i;
        if(check(row,columnForRow))
        {
            helper(n,row+1,columnForRow,res);
        }
    }
}
//以下是檢查第 row 行的皇后是否合法, 即是否與所有其它行的皇后沖突.
private boolean check(int row, int[] columnForRow)
{
    for(int i=0;i<row;i++) //注意是 i<row, 不是 i<n, 因為上面 helper(n,row+1,columnForRow,res)是按次序加的
    {
        if(columnForRow[row]==columnForRow[i] || Math.abs(columnForRow[row]-columnForRow[i])==row-i)
            return false;
    }
    return true;
}
```

52. N-Queens II, Hard

http://blog.csdn.net/linhuanmars/article/details/20668017

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.



One solution to the eight queens puzzle

Subscribe to see which companies asked this question

Key:这道题跟 N-Queens 算法是完全一样的，只是把输出从原来的结果集变为返回结果数量而已。用 ArrayList<Integer>保存結果, 不要用 int.

History:
E1 直接看的答案.
E2 是在本題 I 的代碼上改的, 兩分鍾改好, 本來可以一次通過, 但最初忘了在保存結果的 ArrayList 中忘了先加入一個 0, 改後即通過.
E3 是在本題 I 的 E3 代碼上改的, 瞬間改好(此時間沒意義, 由於 E2 寫了, E3 才寫的), 一次通過. E3 做法跟 key 稍有不同, E3 是返回的結果(但不是最終結果)的長度, 並將結果轉換為棋盤那段刪了. 當然 key 中方法更好.

这道题跟 N-Queens 算法是完全一样的，只是把输出从原来的结果集变为返回结果数量而已。思路我们就不在赘述了，大家可以参见 N-Queens，算法的时间复杂度仍然是指数量级的，空间复杂度是 O(n)。代码如下． 这道题目我个人没有看到从输出结果集变为输出结果数量有什么可提升的空间，不像 Unique Paths，输出结果集还是数量是有不同复杂度的解法的。如果这个题大家有什么更优的解法，可以留言或者发邮件到 linhuanmars@gmail.com 给我交流一下哈。

请问您为何不用 int res 来储存结果，采用 ArrayList<Integer> res 的形式是否有些繁琐？ 谢谢！

回复 sinat_17501975：这是因为 java 中所有的传参都是按值传递，所以如果你直接传 int res 在里面改变 res 的值是不会影响 res 在外面的值的，而我们这个 res 是希望每层递归都能知道最新的值，所以采用传进一个一个元素的数组，这样 res 的值才能真正得到传递 ~ 你可以试试直接传 int 看看是什么情况分析一下哈 ~

```
public int totalNQueens(int n) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    res.add(0);
    helper(n,0,new int[n],res);
    return res.get(0);
}
private void helper(int n, int row, int[] columnForRow, ArrayList<Integer> res)
{
```

```
    if(row==n)
    {
        res.set(0,res.get(0)+1);
        return;
    }
    for(int i=0;i<n;i++)
    {
        columnForRow[row]=i;
        if(check(row,columnForRow))
        {
            helper(n,row+1,columnForRow,res);
        }
    }
}
private boolean check(int row, int[] columnForRow)
{
    for(int i=0;i<row;i++)
    {
        if(columnForRow[i]==columnForRow[row] || Math.abs(columnForRow[row]-
columnForRow[i])==row-i)
            return false;
    }
    return true;
}
```

53. Maximum Subarray, Medium

http://blog.csdn.net/linhuanmars/article/details/21314059

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.
For example, given the array [−2,1,−3,4,−1,2,1,−5,4],
the contiguous subarray [4,−1,2,1] has the largest sum = 6.
click to show more practice.
More practice:
If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.
Subscribe to see which companies asked this question

E3: 注意題目中說 try using divide and conquer, 我查了下, William 是用的跟 Code Ganker 一模一樣的 O(N)方法(公式都一樣). Leetcode Discussion 裡面的帖子也基本上都是 O(N)方法, 有人問如何用 divide and conquer(此帖閱讀量為一萬四千, 其它帖子基本都是幾百), 有人提供了代碼, 但很多人(包括代碼作者自己)都說時間複雜度是 O(N log N), 有人說沒有 O(log N)的方法. 所以本題應該不存在好的 divide and conquer 方法, 出題人應該是水平較低, 才在題目中這樣說. 但是注意 209. Minimum Size Subarray Sum 中, 題目也這樣問了: If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log n). 所以用空了還是看看本題的 O(N log N)的 divide and conquer. Code Ganker(和 William)的 O(N)方法應該是本題的標准方法.

Wikipedia 也有這題(Maximum subarray problem), 裡面給出的算法就是 Code Ganker(和 William)的 O(N)方法, 此方法叫 Kadane's algorithm, 是 CMU 的 J. Kadane 於 70 年代提出的. 面試問此題的, 都是 SB. 老子打死也不相信, 沒見過此題的人能在面試時間內做出. 所以 Leetcode 中有些題的解法是要背的. 網上有人說甚麼多久做刷完 Hard 題 blabla (注意本題是 Medium), 實際上他要麼是在裝 B, 要麼就是以前背過這些算法.

Key: O(N)方法: 注意題意有點表述不清, 要返回的結果是 largest sum, 而不是這個 subarray. 以下兩式其實並不好理解(見後面我寫的兩大段理解), 所以要直接把以下兩式記住:

i 遍歷數組:
local = Math.max(A[i],local+A[i]); //注意 max 中第一項為 A[i], 這個要記住. 本題的 II(152 題)第一項也為 A[i].
global = Math.max(local,global);
global 就是要的結果.

以上算法之理解: local 表示當前(即 A[i]所在的)subarray 的 sum, 而 global 就是所有 subarray 的 sum 中最大的.第一式中, 若 local 為正, 則 max(A[i],local+A[i])為 local+A[i]; 若 local 為負, 則 max(A[i],local+A[i])為 A[i]. 這意味著甚麼? 意味著從 subarray 的左端向右開始加數組中的元素到 local 中, 加到加到, 設加到 A[k]時(即 i 掃到 k 時)local 變為負, 則可知 A[k]必為負(否則為何 local 是由正變負的?), 此時 local 為負, 則 max(A[k+1],local+A[k+1])為 A[k+1], 這意味著之前的 subarray 已到盡頭, 新的 subarray 從 A[k+1]開始了(因為 local=A[k+1]), 然後又像之前那樣, 從新的 subarray 的左端開始加, 直到 local 又變為負... 新的 subarray 為何要從 A[k+1]開始? 因為 A[k]為負, 如果將 A[k]也算在新的 subarray 中, 那所有 sum 全都變小 A[k]的絕對值那麼多, 當然不是我們希望的. 而新的 subarray 為何不從 A[k+1]的下一個(即 A[k+2])開始? 因為若 A[k+1]不爭氣, 即若它為負, 則它自然會在算 max(A[k+1],local+A[k+1])時被淘汰, 即跟之前 A[k]一樣的命運, 而使得這個 subarray 只有 A[k+1]一個人, 而下一個 subarray 從 A[k+2]開始.

一個重要的問題: 為何要以 local 變負作為一個 subarray 結束之標志? 因為 global 只對正的 local 感興趣, 對負的 local 不感興趣, 所以一旦 local 為負, 就開始一個新的 subarray.

上面討論了那麼多 subarray 的左端, 為何不討論 subarray 的右端? 因為右端無所謂, 反正右邊一直往當前 subarray 中加入更多的元素(只要 local 不為負), 若加入的元素為正, local 就增加, 而這個新的 local 值會被 global 記錄下來; 若加入的元素為負, local 減小, global 不記錄就是; 若加入的元素為負且使得 local 也為負, 則就回到了上一段中說的, subarray 到此為止, 開始下一個 subarray(即當前 subarray 的右端就是下一個 subarray 的左端, which has already been discussed in 上一段, 所以還討論個毛的右端?).

注意 local 和 global 的初始值要設為 A[0] (這是因為 i 是從 1 開始的), 而不是 0 (若輸入為{-1}會得出錯誤結果), 也不能是 Interger.MIN_VALUE(因為若輸入為{-1}會 overflow).

History:
E1 直接看的答案.
E2 很快寫好, 但 local 和 global 的初值沒設為 A[0], 所以改了好幾次才通過.
E3 沒做出來, E3 主要是受題目誤導, 想 divide and conquer 方法去了, 沒怎麼想 O(N)方法.

这是一道非常经典的动态规划的题目，用到的思路我们在别的动态规划题目中也很常用，以后我们称为”局部最优和全局最优解法“。
基本思路是这样的，在每一步，我们维护两个变量，一个是全局最优，就是到当前元素为止最优的解是，一个是局部最优，就是必须包含当前元素的最优的解。接下来说说动态规划的递推式（这是动态规划最重要的步骤，递归式出来了，基本上代码框架也就出来了）。假设我们已知第 i 步的 global[i]（全局最优）和 local[i]（局部最优），那么第 i+1 步的表达式是：
local[i+1]=Math.max(A[i], local[i]+A[i])，就是局部最优是一定要包含当前元素，所以不然就是上一步的局部最优 local[i]+当前元素 A[i]（因为 local[i]一定包含第 i 个元素，所以不违反条件），但是如果 local[i]是负的，那么加上

他就不如不需要的，所以不然就是直接用 A[i]；
global[i+1]=Math(local[i+1],global[i])，有了当前一步的局部最优，那么全局最优就是当前的局部最优或者还是原来的全局最优（所有情况都会被涵盖进来，因为最优的解如果不包含当前元素，那么前面会被维护在全局最优里面，如果包含当前元素，那么就是这个局部最优）。

接下来我们分析一下复杂度，时间上只需要扫描一次数组，所以时间复杂度是 O(n)。空间上我们可以看出表达式中只需要用到上一步 local[i]和 global[i]就可以得到下一步的结果，所以我们在实现中可以用一个变量来迭代这个结果，不需要是一个数组，也就是如程序中实现的那样，所以空间复杂度是两个变量（local 和 global），即 O(2)=O(1)。代码如下. 这道题虽然比较简单(簡單你媽逼, 此算法是 CMU 的 J. Kadane 於 70 年代提的, 好像真的是你自己想出來似的. 此逼裝得, 我給 10 分)，但是用到的动态规划方法非常的典型，我们在以后的题目中还会遇到，大家还是要深入理解一下哈。我现在记得的用到的题目是 Jump Game，以后有统计一下再继续更新。

```
public int maxSubArray(int[] A) {
    if(A==null || A.length==0)
        return 0;
    int global = A[0];
    int local = A[0];
    for(int i=1;i<A.length;i++)
    {
        local = Math.max(A[i],local+A[i]);//注意 max 中第一項為 A[i], 這個要記住. 本題的 II(152 題)第一項也為 A[i].
        global = Math.max(local,global);
    }
    return global;
}
```

54. Spiral Matrix, Medium

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].
Subscribe to see which companies asked this question

Key: 本題本質就是旋轉遍歷一個任意數組. 本題沒甚麼算法, 就是 brute force 地按 spiral 的順序放. 像剝洋葱一樣, 由外向內一層一層(即一圈一圈)地剝. 總層數為 Math.min(nRow, nCol)/2 (最好記住, 不要像 E3 那樣用別的式子浪費時間). 注意題目沒說輸入一定是方陣. 另外當 min(rowLength, columnLength)為奇數時, 要專門多加一行或一列, 這個情況別忘了. 注意每個 for 循環的起始點別弄錯了, 爭取一次寫對. 轉的方式用對稱的(見下面的黃字).

History:
E1 參考了 Code Ganker 後才通過(算我沒做出來).
E2 通過.

E3 寫了兩個多小時才通過, 原因是 E3 將層數設為了 nLayer = (Math.min(m, n) + 1) / 2, 然後循環中 layer 為[0,...nLayer-2], 其中 m,n 為輸入數組的行數和列數, 這樣寫的後果是最後要專門多加的那一行或一列很難弄對, 它是互相幹撓, corner case 很多. 所以還是像 Code Ganker 那樣寫比較好 nLayer = Math.min(m, n) / 2, 循環中 layer 為[0,...nLayer-1].

```
/*
00 01 02 03 04 05
10 11 12 13 14 15
20 21 22 23 24 25
30 31 32 33 34 35
40 41 42 43 44 45
50 51 52 53 54 55
*/
```

以下是我 E2 轉的方式:

```
*  *  *  *
*        *
*        *
*  *  *  *
```

以下是 Code Ganker 轉的方式, 比較比稱, 以後就用這個方式.

```
*  *  *  *
*        *
*        *
*  *  *  *
```

Code Ganker:
这道题是比较简单的数组操作，按螺旋顺序把数组读取并且放到结果数组中即可。基本思路跟 Rotate Image 有点类似，就是一层一层的处理，每一层都是按照右下左上的顺序进行读取就可以。实现中要注意两个细节，一个是因为题目中没有说明矩阵是不是方阵，因此要先判断一下行数和列数来确定螺旋的层数。另一个是因为一层会占用两行两列，如果是单数的，最后要将剩余的走完。所以最后还要做一次判断。因为每个元素访问一次，所以时间复杂度是 O(m*n)，m，n 是分别是矩阵的行数和列数，空间复杂度是 O(1)。代码如下.
因为这道题是比较简单的题目，所以上面提到的两个细节还是比较重要的，面试中遇到了一定要注意，尽量做到没有 bug 哈。

Code Ganker 的代碼(用它):

```
public ArrayList<Integer> spiralOrder(int[][] matrix) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    if(matrix == null || matrix.length==0 || matrix[0].length==0)
        return res;
    int min = Math.min(matrix.length, matrix[0].length);
    int levelNum = min/2;
    for(int level=0;level<levelNum;level++)
```

```
    {
        for(int i=level;i<matrix[0].length-level-1;i++)
        {
            res.add(matrix[level][i]);
        }
        for(int i=level;i<matrix.length-level-1;i++)
        {
            res.add(matrix[i][matrix[0].length-1-level]);
        }
        for(int i=matrix[0].length-1-level;i>level;i--)
        {
            res.add(matrix[matrix.length-1-level][i]);
        }
        for(int i=matrix.length-1-level;i>level;i--)
        {
            res.add(matrix[i][level]);
        }
    }
    if(min%2==1)
    {
        if(matrix.length < matrix[0].length)
        {
            for(int i=levelNum; i<matrix[0].length-levelNum;i++)
            {
                res.add(matrix[levelNum][i]);
            }
        }
        else
        {
            for(int i=levelNum; i<matrix.length-levelNum;i++)
            {
                res.add(matrix[i][levelNum]);
            }
        }
    }
    return res;
}
```

Tao: 以下是我的代碼(leetcode 已通過). min(rowLength, columnLength)為奇數的情況參考了 Code Ganker 的, 其餘的是自已寫的. 方法與 Code Ganker 一樣. 思路就是按 spiral 的順序放. 注意題目沒說輸入一定是方陣. 另外當 min(rowLength, columnLength)為奇數時, 要專門多加一行或一列, 這個情況別忘了.

以下代碼用的其它轉的方式, 不用:

```
public static List<Integer> spiralOrder(int[][] matrix) {
  List<Integer> res = new ArrayList<Integer>();
  if(matrix == null || matrix.length == 0 || matrix[0].length == 0)
  //leetcode 的 test case 還真有輸入為[]的.
   return res;
```

```
  int lenRow=matrix.length;
  int lenCol=matrix[0].length;
  int minLen = Math.min(lenRow, lenCol);
  int depth = minLen/2;

  List<Integer> list = new ArrayList<Integer>();

  for(int start=0; start<depth; start++) {

  //讀一行(方形的上):
   for(int i=start; i<lenCol-start; i++) {
    list.add(matrix[start][i]);
   }

  //讀一列(方形的右):
   for(int i=start+1; i<lenRow-start; i++) {
    list.add(matrix[i][lenCol-start-1]);
   }

  //讀一行(方形的下):
   for(int i=lenCol-start-2; i>start; i--) {
    list.add(matrix[lenRow-start-1][i]);
   }

  //讀一列(方形的左):
   for(int i=lenRow-start-1; i>start; i--) {
    list.add(matrix[i][start]);
   }

   res.addAll(list);
   list.clear();
  }

//以下為當 min(rowLength, columnLength)為奇數時, 要專間多加一行或一列
  if(minLen%2 == 1) {
   if(lenRow>lenCol) {

    //多加一列:
    for(int i=depth; i<lenRow-depth; i++) {
    list.add(matrix[i][depth]);
    }

   } else {

   //多加一行:
    for(int i=depth; i<lenCol-depth; i++) {
    list.add(matrix[depth][i]);
    }
   }
```

```
  System.out.println(list);
  res.addAll(list);
  list.clear();
 }

 return res;
}
```

55. Jump Game, Medium

http://blog.csdn.net/linhuanmars/article/details/21354751

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Determine if you are able to reach the last index.
For example:
A = [2,3,1,1,4], return true.
A = [3,2,1,0,4], return false.
Subscribe to see which companies asked this question

Key: E3 的方法比較好懂好寫, 很容易想到, 且稍加修改後可適用於本題的 II, 故以後都用 E3 代碼. E3 方法為: 先定義每個元素 nums[i]能輻射到的最遠處(稱為 reach[i]), 可知 reach[i] = nums[i] + i. 最開始對元素 nums[0], 考查其輻射范圍內的所有元素(稱這些元素們為 vrangh), 在 vrangh 中找出 reach 最大的那個元素(比如 nums[5]), 下一步便跳到 nums[5]處, 然後考查 nums[5]輻射范圍內的所有元素(vrangh), 在它的 vrangh 中找出 reach 最大的元素, 然後跳到此元素那裡去...如此循環, 直到某元素的 reach 超過了數組長度.

E3 的方法跟 Code Ganker 類似, 但有差別. E3 的代碼比較好懂好寫, 且很容易想到, 沒用 reach = Math.max(nums[i] + i, reach)這個公式

Code Ganker 方法: 貪心法. for(int i=0;i<=reach && i<A.length;i++) reach= Math.max(A[i]+i,reach). 等式右邊的 A[i]+i 是 A[i]使得能 夠到最遠的位置, 這是 A[i]的功勞. 等式右邊的 reach 是 A[i]之前的所有元素使得能 夠到最遠的位置, 這是 A[i]之前所有的元素的功勞. 本算法之正確性易用數學歸納法證明. 注意 i<=reach 這個要求很重要, 因為它保證了 i 是能 够得著 的.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過. E2 的代碼比 Code Ganker 的更好.
E3 改了個小錯誤(updated variable unexpeted, 詳見代碼中)後通過. E3 的方法跟 key 類似, 但有差別. E3 的代碼比較好懂好寫, 且很容易想到, 沒用 reach = Math.max(nums[i] + i, reach)這個公式 , 故也帖在了後面, 可以看看. E3 代碼雖然有兩層循環, 但實際上時間複雜度為 O(N), 跟 key 一樣. 當然 reach 數組要佔 O(N)空間, 但應該可以改為不用數組, 我這裡用數組, 主要還是為了好懂好寫.

这道题是动态规划的题目，所用到的方法跟是在 Maximum Subarray 中介绍的套路，用“局部最优和全局最优解法”。我们维护一个到目前为止能跳到的最远距离 (tao: 即 reach)，以及从当前一步出发能跳到的最远距离(tao: 即 A[i] + i)。局部最优 local=A[i]+i，而全局最优则是 global=Math.max(global, local)。递推式出来了，代码就比较容易实现了。因为只需要一次遍历时间复杂度是 O(n)，而空间上是 O(1)。代码如下. 这也是一道比较经典的动态规划的题目，不过不同的切入点可能会得到不同复杂度的算法，比如如果维护的历史

信息是某一步是否能够到达，那么每一次需要维护当前变量的时候就需要遍历前面的所有元素，那么总的时间复杂度就会是 O(n^2)。所以同样是动态规划，有时候也会有不同的角度，不同效率的解法。这道题目还有一个扩展 Jump Game II，有兴趣的朋友可以看看。

鹅鹅 2014-11-30 06:16 发表

greedy algorithm

wx738986 2014-08-27 08:07 发表

lz 我觉得你这个算法是贪心算法不是动态规划

wiki: **貪心法**，又稱**貪心演算法**，是一種在每一步選擇中都採取在當前狀態下最好或最優（即最有利）的選擇，從而希望導致結果是最好或最優的算法。一般人換零錢的時候也會應用到貪心算法。把$36 換散：$20 > $10 > $5 > $1. 貪心算法與動態規劃的不同在於它每對每個子問題的解決方案都做出選擇，不能回退。動態規劃則會保存以前的運算結果，並根據以前的結果對當前進行選擇，有回退功能。

Code Ganker 的代碼(不用):

```
public boolean canJump(int[] A) {
    if(A==null || A.length==0)
        return false;
    int reach = 0;
    for(int i=0;i<=reach && i<A.length;i++) //注意 i<=reach 這個要求很重要, 因為它保證了 i 是能 够得著的.
    {
        reach = Math.max(A[i]+i,reach);
        //System.out.println(i+" "+A[i]+" "+(A[i]+i)+" "+reach);
    }
    if(reach<A.length-1)
        return false;
    return true;
}
//若輸入為{2, 3, 1, 1, 4}, 則以上 println 輸出結果為
0 2 2 2
1 3 4 4
2 1 3 4
3 1 4 4
4 4 8 8
最後 canJump ()結果為 true, 即該數組能到最後.
```

E2 的代碼(不用):

```
    public boolean canJump(int[] nums) {
        if(nums == null || nums.length == 0) return false;

        int reach = 0;

        for(int i = 0; i <= reach && i < nums.length; i++) {
            reach = Math.max(nums[i] + i, reach);
            if(reach >= nums.length - 1) return true;
        }
```

```
        return false;
    }
```

E3 的代碼(用它):

```
    public boolean canJump(int[] nums) {
        if(nums == null || nums.length == 0) return false;
        int n = nums.length;
        int[] reach = new int[n];
        for(int i = 0; i < n; i++) reach[i] = nums[i] + i;

        int i = 0, maxReach = 0;

        while(i < n) {
            if(reach[i] >= n - 1) return true;
            if(nums[i] == 0) return false;
            int iNext = 0;
            for(int j = i + 1; j <= reach[i] && j < n; j++) {
                if(reach[j] > maxReach) {
                    maxReach = reach[j];
                    iNext = j; //E3 最開始是寫成的 i=j, 結果出錯. 這是因為這樣的話, 就將 for(wutzer)中的 wutzer 中的 i 的值改變了, 但我們並不想改變 wutzer 中的 i 值, 我們只想得到下一個要跳到的 i 值(即 iNext).
                }
            }
            i = iNext;
            maxReach = 0;
        }

        return false;
    }
```

56. Merge Intervals, Hard

http://blog.csdn.net/linhuanmars/article/details/21857617

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].
Subscribe to see which companies asked this question

Key: 見 Java p795 的 Comparator. 先自定義一個 Comparator<Interval>, 然後用 Java p797 的 Collections.sort(intervals,comp) 對 intervals 排序. 剩下的就好辦了. merge 時新的 end 選 overlap 的兩個 interval 的 end 中較大的一個. 用一個 List res 來放結果, 設 res 的最後一個元素為 intervals[i-1], 則比較它和 intervals[i], 若它們有 overlap, 則將 res 最後一個元素的 end 設為它們兩個較大的那個 end, 否則(即沒有 overlap)將 intervals[i]放入 res 中.

interval 的一些 convention 跟 57 題是一樣的.

History:
E1 直接看的答案.
E2 通過, 但代碼比 Code Ganker 的略囉嗦, 原因是 Code Ganker 用了 res.get(res.size()-1).end.
E3 沒多久寫好, 一次通過(本題是 Hard, 略 NB). E3 是直接在 intervals 這個 List 中操作的, 最後也返回操作後的 intervals. 即: 若 a 和 b 是 intervals 中的兩個相鄰的 interval, 若它們有 overlap, 則將 a.end 設為 b.end, 然後從 intervals 中刪除 b. 這樣做的缺點是要不停地刪東西. key 中方法不用刪.

这是一道关于 interval 数组结构的操作，在面试中也是一种比较常见的数据结构。假设这些 interval 是有序的（也就是说先按起始点排序，然后如果起始点相同就按结束点排序），那么要把它们合并就只需要按顺序读过来，如果当前一个和结果集中最后一个有重叠，那么就把结果集中最后一个元素设为当前元素的结束点（不用改变起始点因为起始点有序，因为结果集中最后一个元素起始点已经比当前元素小了）。那么剩下的问题就是如何给 interval 排序，在 java 实现中就是要给 interval 自定义一个 Comparator，规则是按起始点排序，然后如果起始点相同就按结束点排序。整个算法是先排序，然后再做一次线性遍历，时间复杂度是 O(nlogn+n)=O(nlogn)，空间复杂度是 O(1)，因为不需要额外空间，只有结果集的空间。代码如下. 自定义 Comparator 有时候在面试中也会要求实现，不熟悉的朋友还是要熟悉一下哈。LeetCode 中关于 interval 的题目还有 Insert Interval，有兴趣的朋友可以看看哈。

LZ, 再看这道题的时候有个困惑，就是在做比较器的时候，我们可以实例化一个比较器之后再 block 里 override 比较方法，想问下这个只是在 leetcode 中可以，还是现实编程中也能这样？如果我想用重建 class 实现 Comparator 接口，貌似在 leetcode 里不行。。。

回复 LostBank：现实中经常这样～ 在写匿名类的时候经常就是 new 出来后直接再 block 里面定义方法哈～(tao: 此句重要)

回复 linhuanmars：明白啦，原来这就是传说中的匿名类（没有名字的类），感谢 LZ，受教~

LZ 第 10，11 行是否可以去掉。因为 22 行判断了？

回复 jingyuzhang：从比较函数的定义来说， 是需要那么写的， 而且我觉得二十二行做的事情似乎和前面没有冲突， 排序还是得正常执行的哈～ 有问题继续交流哇～

第 10，11 行去掉是通过 Leetcode 的因为 2 个 start 相同的数一定会走 22 行。LZ 是否提供一个 test case,证明 10，11 行不能去掉？ 谢谢。

回复 jingyuzhang：这个问题如果要过 leetcode 确实是代码可以去掉那两行，但是从 Interval 的比较函数定义来说，还是要先比较 start，再比较 end，这样独立来看是健壮的，有时候面试并不是做对就好，代码的规范和健壮性还是能加分的哈～

回复 linhuanmars：谢谢回复。 面试时， 我也许会说这个是 start interval Comparator .因为这个问题，出性能上考虑，我们只需要 compare start interval. 别的问题还会有 end interval Comparator 和 interval Comparator。当然，这是个人习惯。再次感谢 LZ 解答。

Tao: 將原代碼中的 ArrayList<Interval>改成了 List<Interval>, 其它地方也作了相應改動, 否則通不過, 改了後就能通過了. 由此可知為何 Java 書中老是愛用 List<Interval> res = new ArrayList<Interval>()這樣的形或了, 因為 leetcode 要求這樣的習慣啊.

```
public List<Interval> merge(List<Interval> intervals) {
   List<Interval> res = new ArrayList<Interval>();
   if(intervals==null || intervals.size()==0)
      return intervals;
   Comparator<Interval> comp = new Comparator<Interval>() //别忘了()
   {
      @Override //注意 O 要大寫
      public int compare(Interval i1, Interval i2)
      {
         if(i1.start==i2.start)
            return i1.end-i2.end;
```

```
            return i1.start-i2.start;
        }
    }; //别忘了分號! 整個 Comparator<Interval> comp = new Comparator<Interval>() {...}; 是一句話.
    Collections.sort(intervals,comp);

    res.add(intervals.get(0));
    for(int i=1;i<intervals.size();i++) //注意起始點是 i=1, 不是 i=0
    {
        if(res.get(res.size()-1).end>=intervals.get(i).start) //注意是>=, 不是>. 另外, 此處用
res.get(res.size()-1).end, 這點比較巧, 這樣點出的代碼比 E2 的簡明.
        {
            res.get(res.size()-1).end = Math.max(res.get(res.size()-1).end, intervals.get(i).end);
        }
        else
        {
            res.add(intervals.get(i));
        }
    }
    return res;
}
```

57. Insert Interval, Hard

http://blog.csdn.net/linhuanmars/article/details/22238433

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].
Subscribe to see which companies asked this question

Key: E3 的方法跟 Code Ganker 都是 O(N), Code Ganker 是一次遍歷, E3 是幾次遍歷, E3 方法跟 Code Ganker 類似, 但 E3 的方法好懂好記好想到, 故以後用 E3 的方法.

E3 的方法: 先從左到右掃描 intervals, 找到 newInterval.start 所在的那個 interval[i] (即 newInterval.start 在 interval[i]的兩端之間, 用 newInterval.start <= intervals[i].end 來判斷, 稱此 intervals[i]為小明). 然後從右到左掃描 intervals, 找到 newInterval.end 所在的那個 interval[i] (用 newInterval.end >= intervals[i].start 來判斷, 稱此 intervals[i]為小紅). 然後建一個新的 Interval 叫 merged, 將 merged.start 設為 newInterval.start 和 小明.start 中較小的那個, 將 merged.end 設為 newInterval.end 和 小紅.end 中較大的那個. 然後將 interval 們放入 res 這個 List 中: 將小明之前的(不包括小明)放入 res 中, 然後將 merged 放入 res 中, 然後將小紅之後的(不包括小紅)放入 res 中. 返回 res 即可. 這樣做幾乎可以包括所有 case(稍想即知, 也可以不想): newInterval 兩端都在別的 interval 內部, newInterval 兩端中某端不在別的 interval 內部, newInterval 兩端都不在別的 interval 內部, newInterval 完全將某 interval 包含在體內, 等等. 只有一種情況沒包括: 就是小明(或小紅)不存在时(即 newInterval 應插入的位置在 intervals 最右端(或最左端), 專門處理這種情況即可.

Code Ganker 的方法：基本思路就是先扫描走到新的 interval 应该插入的位置，接下来就是插入新的 interval, 一直到新的 interval 的 end 小于下一个 interval 的 start，然后取新 interval 和当前 interval 中 end 大的即可。根据 intervals.get(i).end<newInterval.start 來找 newInterval 應插入的位置.
實踐證明, 一些 conventions:
[1, 2]和[3, 4]不算有 overlap, 要[1, 2]和[2, 4]才算有 overlap
[2, 2]也算一個 interval

History:
E1 直接看的答案.
E2 通過, 但不是用 key 中方法來找 newInterval 應插入的位置, 結果太多 corner case, 改了好多次.
E3 改了一次後通過, 原因是沒考慮到小明和小紅不存在之情況(見上面 key).

Code Ganker:

这道题跟 Merge Intervals 很类似，都是关于数据结构 interval 的操作。事实上，Merge Intervals 是这道题的子操作，就是插入一个 interval，如果出现冲突了，就进行 merge。跟 Merge Intervals 不一样的是，这道题不需要排序，因为插入之前已经默认这些 intervals 排好序了。简单一些的是这里最多只有一个连续串出现冲突，因为就插入那么一个。基本思路就是先扫描走到新的 interval 应该插入的位置，接下来就是插入新的 interval 并检查后面是否冲突，一直到新的 interval 的 end 小于下一个 interval 的 start，然后取新 interval 和当前 interval 中 end 大的即可。因为要进行一次线性扫描，所以时间复杂度是 O(n)。而空间上如果我们重新创建一个 ArrayList 返回，那么就是 O(n)。有朋友可能会说为什么不 in-place 的进行操作，这样就不需要额外空间，但是如果使用 ArrayList 这个数据结构，那么删除操作是线性的，如此时间就不是 O(n)的。如果这道题是用 LinkedList 那么是可以做到 in-place 的，并且时间是线性的。代码如下.

这道题有一个变体，就是如果插入的时候发现冲突，那就返回失败，不插入了。看起来好像比上面这道题还要简单，但是要注意的是，如此我们就不需要进行线性扫描了，而是进行二分查找，如果不冲突，则进行插入，否则直接返回失败。这样时间复杂度可以降低到 O(logn)。当然这里需要用二分查找树去维护这些 intervals。所以一点点变化可能可以使复杂度降低，还是应该多做思考哈。
同时，这种题目还可以问一些关于 OO 设计的东西，比如就直接问你要实现一个 intervals 的类，要维护哪些变量，实现哪些功能，用什么数据结构，等等。这些你可以跟面试官讨论，然后根据他的功能要求用相应的数据结构。所以扩展性还是很强的，大家可以考虑的深入一些。

请教下 linkedlist remove()是 O(n)吧，还有用 linked'list 的 in-place 算法总体的复杂度为什么是线性的呢？

回复 diwu001：如果是 remove 头尾的元素就是 O(1)的，但是如果 remove 某个 index 对应的元素就是 O(n)的，用 linkedlist 操作和 arraylist 一样，只是 in-place 方便，因为 linkedlist 操作不需要移动元素，只要接指针就可以了 ~

Code Ganker 代碼(不用):
tao: 以下已將原代碼中 的 ArrayList 改成了 List, 並通過.

```
public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
    List<Interval> res = new ArrayList<Interval>();
    if(intervals.size()==0)
    {
        res.add(newInterval);
        return res;
    }
    int i=0;
//以下是找 newInterval 應插入的位置
    while(i<intervals.size() && intervals.get(i).end<newInterval.start)
    {
```

```
        res.add(intervals.get(i));
        i++;
    }
//此時已找到 newInterval 應插入的位置, 以下的 if 是設定 newInterval 的 start.
    if(i<intervals.size())
        newInterval.start = Math.min(newInterval.start, intervals.get(i).start);
    res.add(newInterval);
//以下的 while 是設定 newInterval 的 end.
    while(i<intervals.size() && intervals.get(i).start<=newInterval.end) //注意是<=
    {
        newInterval.end = Math.max(newInterval.end, intervals.get(i).end);
        i++;
    }
    while(i<intervals.size())
    {
        res.add(intervals.get(i));
        i++;
    }
    return res;
}
```

E3 代碼(用它):

```
    public static List<Interval> insert(List<Interval> intervals, Interval newInterval) {
        List<Interval> res = new ArrayList<>();

        if(intervals == null || intervals.size() == 0) {
            res.add(newInterval);
            return res;
        }

        int startPos = -1, endPos = -1; //小明和小紅

        for(int i = 0; i < intervals.size(); i++) {
            Interval cur = intervals.get(i);
            if(newInterval.start <= cur.end) {
                startPos = i;
                break;
            }
        }

        for(int i = intervals.size() - 1; i >= 0; i--) {
            Interval cur = intervals.get(i);
            if(newInterval.end >= cur.start) {
                endPos = i;
                break;
            }
        }

        //小明不存在時(即 newInterval 應插入的位置在 intervals 最右端):
```

```
    if(startPos == -1) {
        res = new ArrayList<>(intervals);
        res.add(newInterval);
        return res;
    }

    //小紅不存在時(即 newInterval 應插入的位置在 intervals 最左端):
    if(endPos == -1) {
        res.add(newInterval);
        for(Interval i : intervals) res.add(i);
        return res;
    }

    int mergedStart = Math.min(newInterval.start, intervals.get(startPos).start);
    int mergedEnd = Math.max(newInterval.end, intervals.get(endPos).end);

    Interval merged = new Interval(mergedStart, mergedEnd);

    for(int i = 0; i < startPos; i++) res.add(intervals.get(i));
    res.add(merged);
    for(int i = endPos + 1; i < intervals.size(); i++) res.add(intervals.get(i));

    return res;
}
```

58. Length of Last Word, Easy

http://blog.csdn.net/linhuanmars/article/details/21858067

Given a string *s* consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.
If the last word does not exist, return 0.
Note: A word is defined as a character sequence consists of non-space characters only.
For example,
Given *s* = "Hello World",
return 5.
Subscribe to see which companies asked this question

Key: 本題很簡單. 先去掉尾部的空格, 然後從後往前數.

Convertion: 輸入"a ", 注意 a 後有一個空格, 輸出 1.

每道題的 converntion 還是要記一下, 因為若是 OA, 則沒有面試官可以問.

History:
E1 通過.
E2 幾分鍾寫好, 一次通過, 但 E2 調用了 string.trim()函數來去空格, 更方便, 再將結合 Code Ganker 的代碼改進後, 就可以得到很短的代碼, 帖在了最後, 都可以看看.
E3 兩分金鍾寫好, 算一次通過. 本來 E3 沒一次通過, 因為 E3 寫代碼想到了"a "這種情況, 並將其理解為 the

last word does not exist(題目中這樣說了, 所以稍被誤導), 所以返回的 0, 提交後, 才知道應當返回 1. 由於這只是一個 convention, 且我寫代碼時也想到過這種情況, 所以算一次通過.

Tao: 我的代碼也通過了(瞄了一眼 Code Ganker 要去掉尾部空格的那句話), 但我是簌前往後數的, 沒有 Code Ganker 的簡明, 所以下面用 Code Ganker 的代碼.

这道题比较简单，就是进行一下字符串操作。唯一的细节就是要去掉尾部的空格，然后读到下一个空格，记录下长度。时间复杂度是 O(n)，n 是字符串的长度，空间复杂度是 O(1)。代码如下. 这样子的题目比较简单，考查最简单的代码实现，一般也就是面试的第一道最基础的题目，这种题目要细心，实现不可以有 bug 哈。

```
public int lengthOfLastWord(String s) {
    if(s==null || s.length()==0)
        return 0;
    int idx = s.length()-1;
    while(idx>=0 && s.charAt(idx)==' ') idx--;
    int idx2 = idx;
    while(idx2>=0 && s.charAt(idx2)!=' ') idx2--;
    return idx-idx2;
}
```

以下是我 E2 改進後的代碼:

```
    public int lengthOfLastWord(String s) {
        if(s == null || s.length() == 0) return 0;
        s = s.trim();
        int i = s.length() - 1;
        while(i >= 0 && s.charAt(i) != ' ') i--;
        return s.length() - 1 - i;
    }
```

E3 的代碼更簡潔:

```
    public int lengthOfLastWord(String s) {
        if(s == null || s.length() == 0) return 0;
        s = s.trim();
        int res = 0;

        for(int i = s.length() - 1; i >= 0; i--) {
            if(s.charAt(i) == ' ') break;
            res++;
        }

        return res;
```

}

59. Spiral Matrix II, Medium

Given an integer *n*, generate a square matrix filled with elements from 1 to *n*2 in spiral order.
For example,
Given *n* = 3,
You should return the following matrix:
[
[1, 2, 3],
[8, 9, 4],
[7, 6, 5]
]
Subscribe to see which companies asked this question

Key: 本題本質就是旋轉遍歷一個正方形數組. 本題沒甚麼算法, 就是 brute force 地按 spiral 的順序放. 像剝洋葱一樣, 由外向內一層一層(即一圈一圈)地剝. 總層數為 n/2 (最好記住, 不要像本題 I 中 E3 那樣用別的式子浪費時間). 若 n 為奇數, 最後還要專門加上數組中心那個數. 轉的方式用對稱的(見下面的黃字).

History:
E1 通過.
E2 通過.
E3 幾分鍾寫好(抄了我本題 I 的代碼), 一次通過. E3 在題目分類中 將本題調到了 I 前面, 是因為本題是 I 的特殊情況, 先做本題, 可以幫助推出 I 中的公式等.

Tao: 這道題比 Spiral Matrix 還要簡單些. 以下用我自己的代碼(leetcode 已通過), CodeGanker 的方法跟我差不多.

```
/*
00 01 02 03 04 05
10 11 12 13 14 15
20 21 22 23 24 25
30 31 32 33 34 35
40 41 42 43 44 45
50 51 52 53 54 55
*/
```

E2 和 Code Ganker 都用的以下轉的方式(Both E2 和 Code Ganker 在 n 為奇數時都要單獨處理一個情況):

```
*  *  *  *
*        *
*        *
*  *  *  *
```

建議還是用以下對稱的方式:

```
*  *  *  *
```

```
*       *

*       *

*  *  *  *
```

Code Ganker:
这道题跟 Spiral Matrix 很类似，只是这道题是直接给出 1 到 n^2，然后把这些数按照螺旋顺序放入数组中。思路跟 Spiral Matrix 还是一样的，就是分层，然后按照上右下左的顺序放入数组中。每个元素只访问一次，时间复杂度是 O(n^2)。代码如下.
这种题目就是简单的数组操作，主要是得细心，注意数组下标即可。

Code Ganker 的代碼(用它)

```
public int[][] generateMatrix(int n) {
    if(n<0)
        return null;
    int[][] res = new int[n][n];
    int levelNum = n/2;
    int num = 1;
    for(int l=0;l<levelNum;l++)
    {
        for(int i=l;i<n-l;i++)
        {
            res[l][i] = num++;
        }
        for(int i=l+1;i<n-l;i++)
        {
            res[i][n-1-l] = num++;
        }
        for(int i=n-2-l;i>=l;i--)
        {
            res[n-1-l][i] = num++;
        }
        for(int i=n-2-l;i>l;i--)
        {
            res[i][l] = num++;
        }
    }
    if(n%2==1)
    {
        res[levelNum][levelNum] = num;
    }
    return res;
}
```

以下代碼用的別的方式, 雖然 n 為奇數時可以不單獨處理那個情況, 但這種方式不好記憶, 故不用以下代碼

```
 public static int[][] generateMatrix(int n) {
  int[][] res = new int[n][n];
  int num = 1;
```

```
 for(int start=0; start<=n/2; start++) {

  for(int i=start; i<n-start; i++) {
  //以下兩句可以直接寫成 res[start][i] = num++;
   res[start][i] = num;
   num++;
  }

  for(int i=start+1; i<n-start; i++) {
   res[i][n-start-1] = num;
   num++;
  }

  for(int i=n-start-2; i>start-1; i--) {
   res[n-start-1][i] = num;
   num++;
  }

  for(int i=n-start-2; i>start; i--) {
   res[i][start] = num;
   num++;
  }
 }

 return res;
}
```

60. Permutation Sequence, Medium

http://blog.csdn.net/linhuanmars/article/details/22028697

The set [1,2,3,…,*n*] contains a total of *n*! unique permutations.
By listing and labeling all of the permutations in order,
We get the following sequence (ie, for *n* = 3):

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

Given *n* and *k*, return the *k*th permutation sequence.
Note: Given *n* will be between 1 and 9 inclusive.
Subscribeto see which companies asked this question

Key: 題意: 排列順序是按數字大小排的, 例如 132 比 123 大, 故 132 在 123 之後. 題目的 convention 為: k 是從 1 開始的, 比如 n=3, k=1 時, 應返回"123".

E3 和 Code Ganker 的非遞歸法:
例如 n=4 時前幾個排列為(p=k-1):
1234, k=1, p=0

1243, k=2, p=1
1324, k=3, p=2
1342, k=4, p=3
1423, k=5, p=4
1432, k=6, p=5

2134, k=7, p=6
2143, k=8, p=7
2314, k=9, p=8
2341, k=10, p=9
2413, k=11, p=10
2431, k=12, p=11
...
若 k=10, 則結果為 2341.
首先定義一個 list=1234.
令 fac=3!=6, 注意前 fac 個排列的首位都是 1, 之後又有 fac 個排列的首位是 2...故用(k-1)/fac 就可以得出結果的第一位數: (k-1)/fac=9/6=1, list[1]=2, 2 即為結果的首位數. 此時有人要問, 為何要用(k-1), 為何不直接用 k: k/fac=10/6=1, list[1]=2, 同樣可以得出正確的結果首位數? 答: 當 k=6 時, 結果的首位數應該為 1, 若用(k-1), 則(k-1)/fac=5/6=0, list[0]=1, 但若用 k, 則 k/fac=6/6=1, list[1]=2, 故用 k 是錯的. 實際寫代碼時, 可在最先就定義一個 p=k-1, 以後就只用 p, 即用 p/fac 即可得到 list 的 index.

得到了結果的首位後, 如何得後面的數字? 答: 先將結果的首位(2)從 list 中刪掉, 此時 list=134, 然後更新 p=p%fac=9/6=3, fac=fac/i(此時 i 即前面 3!中的那個 3)=6/3=2, 然後用新的 p 和 fac 值: p/fac=1, 即結果的下一位為 list[1]=3. 然後再將 3 從 list 中刪掉, 更新 p 和 fac 的值, ...注意結果的最後一位不能這樣得出, 要專門加(即 list 中最後僅剩的那個數). 若要看得更清楚, 可見 E3 代碼後面 對 E3 代碼的理解.

E2 的遞歸法: 寫一個 helper(int n, int k, StringBuilder sb, List<Integer> remains), 甚作用為在 sb 中放入[1,2,3...n]的第 k+1 個 permutation (用題目的 convention). 所以 getPermutation 函數中應調用 helper(n, k-1, sb, remains). remain 用來記錄本次要 permutate 的數, 要每輪 helper 函數內部要從 remain 中刪掉本輪中放入 sb 中的那個元素. 可寫一個函數專門來算 n!, 但也可以不寫.

History:
E1 直接看的答案.
E2 用的遞歸, 但應該跟 Code Ganker 的算法是等價的. 遞歸要好寫些.
E3 用的非遞歸法, 花了好一會兒小心地寫, 一遍通過. E2.5 看的是 E2 的遞歸法 key, 但 E3 還是忘了用遞歸法, 而是用非遞歸法寫的, 做完後才發現 E3 的代碼跟 Code Ganker 幾乎完全一樣, 好多地方都不謀而合, 比如要先將 k 減 1, 以及要用一個 list. 而且 E3 覺得 非遞歸法 比 遞歸法 好, 且也不難寫, 所以以後都用 非遞歸法. E3 代碼比 Code Ganker 的略簡潔好懂, 以後用 E3 代碼.

这道题目算法上没有什么特别的，更像是一道找规律的数学题目。我们知道，n 个数的 permutation 总共有 n 阶乘个，基于这个性质我们可以得到某一位对应的数字是哪一个。思路是这样的，比如当前长度是 n，我们知道每个相同的起始元素对应(n-1)!个 permutation，也就是(n-1)!个 permutation 后会换一个起始元素。(tao: 所以 k/(n-1)!即為每一輪可以放入的元素) . 因此，只要当前的 k 进行(n-1)!取余，得到的数字就是当前剩余数组的 index (tao: 即下一輪的 k)，如此就可以得到对应的元素。如此递推直到数组中没有元素结束。实现中我们要维护一个数组来记录当前的元素，每次得到一个元素加入结果数组，然后从剩余数组中移除，因此空间复杂度是 O(n)。时间上总共需要 n 个回合，而每次删除元素如果是用数组需要 O(n),所以总共是 O(n^2)。这里如果不移除元素也需要对元素做标记，所以要判断第一个还是个线性的操作。代码如下. 上面代码还有有个小细节，就是一开始把 k--，目的是让下标从 0 开始，这样下标就是从 0 到 n-1，不用考虑 n 时去取余，更好地跟数组下标匹配。如果有朋友有更好的思路来实现线性的时间复杂度，欢迎指教哈，可以留言或者发邮件到 linhuanmars@gmail.com 给我。

Code Ganker 代碼(不用):

```
public String getPermutation(int n, int k) {
    if(n<=0)
        return "";
    k--; //這裡 k--是想使 k 從 0 開始, 這是為了後面可以直接算 k %= factorial
    StringBuilder res = new StringBuilder();
    int factorial = 1;
    ArrayList<Integer> nums = new ArrayList<Integer>();

   for(int i=2;i<n;i++)
    {
        factorial *= i;
    } //此時 factorial = (n-1)!

    for(int i=1;i<=n;i++)
    {
        nums.add(i);
    } //此時 nums=1, 2, 3 … n

    int round = n-1;
    while(round>=0)
    {
        int index = k/factorial;
//此乃本題之關鍵一
//index 為每一輪的第一個數, 例如 n=3 時,
  第一輪為 123, 132,
  第二輪為 213, 231,
  第三輪為 312, 321.
  每一輪有 factorial 個排列.
  (別忘了前面最開始有 k--)

        res.append(nums.get(index));
        nums.remove(index);

       k %= factorial;
 //這是算下一輪要用到的 k, 此乃本題之關鍵二, 稍想即能理解.
 //原代碼中此句在 int index = k/factorial 之後, 我把它移到這裡, 要好理解些, leetcode 可通過.

        if(round>0)
            factorial /= round;  //這是算下一輪要用到的 factorial
        round--; //注意此句在 if(round>0)之外
    }
    return res.toString();
}
```

E2 的代碼(不用):

```
    private Map<Integer, Integer> facMap = new HashMap<>(); //value 是(key!)

    public String getPermutation(int n, int k) {
```

```
        StringBuilder sb = new StringBuilder();
        if(n < 0 || k < 0) return sb.toString();
        List<Integer> remains = new ArrayList<>();
        facMap.put(0, 1);
        //for(int i = 1; i <= 9; i++) facMap.put(i, i * facMap.get(facMap.size() - 1)); //若用這句代替下句, 則不用調用 fac()函數, 會省點計算, 我電腦上能給出正確結果, 但 leetcode 上通不過, 不知道為甚麼.
        for(int i = 1; i <= 9; i++) facMap.put(i, fac(i));
        for(int i = 1; i <= n; i++) remains.add(i);
        helper(n, k - 1, sb, remains); //注意此處的 k-1 與 Code Ganker 的 k—不謀而合. 我 E2 試著修改 helper 的寫法, 好讓這裡不減 1, 但反而弄得容易數組越界, 所以還是這樣寫.
        return sb.toString();
    }

//helper(int n, int k, StringBuilder sb, List<Integer> remains), 甚作用為在 sb 中放入[1,2,3...n]的第 k+1 個 permutation (用題目的 convention). 所以 getPermutation 函數中應調用 helper(n, k-1, sb, remains). remain 用來記錄本次要 permutate 的數, 要每輪 helper 函數內部要從 remain 中刪掉本輪中放入 sb 中的那個元素. 可寫一個函數專門來算 n!, 但也可以不寫.
    private void helper(int n, int k, StringBuilder sb, List<Integer> remains) {
        if(n <= 0) return;
        int facVal = facMap.get(n - 1);
        int addIdx = k / facVal;
        sb.append(remains.get(addIdx));
        k = k % facVal;
        remains.remove(addIdx);
        helper(n - 1, k, sb, remains);
    }

    //算 n!
    private int fac(int n) {
        if(n <= 0) return 1;
        int res = 1;
        for(int i = 1; i <= n; i++) res *= i;
        return res;
    }

E3 的代碼(用它):
    public String getPermutation(int n, int k) {
        if(n <= 0 || k <= 0) return null;
        StringBuilder res = new StringBuilder();
        List<Integer> list = new ArrayList<>();
        for(int i = 1; i <= n; i++) list.add(i);

        int fac = 1;
        for(int i = 2; i < n; i++) fac *= i;
        int p = k - 1; //為何要用 p=k-1? 見下面對 E3 代碼的理解

        for(int i = n - 1; i >= 1; i--) {
            int idx = p / fac;
            res.append(list.get(idx));
```

```
            p = p % fac;
            fac = fac / i;
            list.remove(idx);
        }

        res.append(list.get(0));

        return res.toString();
    }
```

對 E3 代碼的理解:

若 n = 4, k = 10, 則:
1234, k=1, p=0
1243, k=2, p=1
1324, k=3, p=2
1342, k=4, p=3
1423, k=5, p=4
1432, k=6, p=5 ← 由此行, 可知為何要用 p, 而不用 k, 這是因為此時若用 k, 則 idx=k/fac=6/6=1, 而不是 0

2134, k=7, p=6
2143, k=8, p=7
2314, k=9, p=8
2341, k=10, p=9
2413 , k=11, p=10
2431, k=12, p=11
…

每輪循環各變量值為:
initial: n=4, p=9, fac=6, list=1234

i=3,
idx=1, append 2
p=3, fac=2, list=134

i=2
idx=1, append 3
p=1, fac=1, list=14

i=1
idx=1, append 4
p=1, fac=1, list=1

最後專門 append 1

61. Rotate List, Medium

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

For example:
Given 1->2->3->4->5->NULL and *k* = 2,
return 4->5->1->2->3->NULL.
Subscribe to see which companies asked this question

Key: 用 walker-runner 定位到要旋转的那个结点. 方法是 walker 和 runner 最開始都在 list 頭, 然後 runner 走 k 步, 然後 walker 和 runner 一起走(速度一樣, 注意跟之前題目中的不同), 當 runner 走到 list 末尾時, walker 就在要旋轉的那個結點. 然后将 walker 下一个结点设为新表头，并且把 walker 设为表尾. 注意 k 大於 list 長度時, 要取餘.

Corner cases: k=0 的情況, k>list 長度的情況.

History:
E1 通過, 方法是寫一個 rotateOnce 函數, 對 list 作一次 rotate, 然後調用 k 次 rotateOnce.
E2 通過, 用的 walker-runner 方法, 只看了一下下面那句(即要用 walker-runner 方法), 然後就自己寫出來了(當時的 key 是按我 E1 的代碼寫的), 第一次沒過, 因為沒將 k % list_length = 0 的情況單獨考慮(Code Ganker 的代碼不用單獨考慮, 原因尚不清楚), 改後就通過了.
E3 也用的 key 中方法, 也改了幾次才通過, 主要是因為沒考慮到 k=0 的情況, 沒考慮到 k>list 長度的情況. E3 的代碼比 Coder Ganker 的簡明好懂, 以後用 E3 的代碼(在最後).

Code Ganker 的方法是用 walker-runner 定位到要旋转的那个结点，然后将下一个结点设为新表头，并且把当前结点设为表尾。

要掌握的是 Code Ganker 的方法(即下面第二種方法). 因為我的方法計算次數要比 Code Ganker 的多.

E1 的代碼(不用):

```
 public static ListNode rotateRight(ListNode head, int k) {
  if(head == null || k < 0)
   return null;

  if(head.next == null)
   return head;

  int length = 0;
  ListNode current = head;

  while(current != null) {
   current = current.next;
   length++;
  }

  k %= length; //取餘後可以減少很多運算, 特別是 k 很大的時候. 如果不要這句, leetcode 中是超時.

  for(int i=0; i<k; i++) {
   head = rotateOnce(head);
  }

  return head;
 }
 //以下是 rotateOnce 函數
 private static ListNode rotateOnce(ListNode head) {
```

```
 ListNode newEnd = head;

 while(newEnd.next.next != null) {
  newEnd = newEnd.next;
 }

 ListNode newHead = newEnd.next;
 newHead.next = head;

 newEnd.next = null;

 return newHead;
}
```

Code Ganker:

这是一道链表操作的题目，基本思路是用 walker-runner 定位到要旋转的那个结点，然后将下一个结点设为新表头，并且把当前结点设为表尾。需要注意的点就是旋转的结点数可能超过链表长度，所以我们要对这个进行取余。定位旋转的尾结点的不超过链表的长度，所以时间复杂度是 O(n)，空间复杂度是 O(1)。代码如下. 上面的实现中采取的方式是直接走到那个结点，如果没超过长度就直接旋转了，如果超过了，就进行取余，并且重新跑到结尾点。其实也可以先直接求长度，然后取余之后再走。其实各有利弊(我: 當然是前者好)，所以大家根据喜好实现即可哈。

(不用以下代碼)

```
public ListNode rotateRight(ListNode head, int n) {
    if(head == null)
    {
        return null;
    }
    ListNode walker = head;
    ListNode runner = head;
    int idx = 0;
     //求 list 長度:
    while(runner!=null && idx<n)
    {
        runner = runner.next;
        idx++;
    }
    if(runner == null)
    {
        n %= idx;
        runner = head;
        idx=0;
        while(runner!=null && idx<n)
        {
            runner = runner.next;
            idx++;
        }
    }
    while(runner.next!=null)
```

```
    {
        walker = walker.next;
        runner = runner.next;
    }
    runner.next = head;
    ListNode newHead = walker.next;
    walker.next = null;
    return newHead;
}
```

我 E3 的代碼(用這個):

```
    public ListNode rotateRight(ListNode head, int k) {
        if(head == null || head.next == null || k == 0) return head;
        ListNode l = head, r = head;
        int size = 0;
        ListNode cur = head;

        while(cur != null) {
            cur = cur.next;
            size++;
        }

        if(k >= size) k = k % size;
        if(k == 0) return head;

        for(int i = 1; i <= k; i++) r = r.next;

        while(r.next != null) {
            r = r.next;
            l = l.next;
        }

        ListNode newHead = l.next;
        r.next = head;
        l.next = null;

        return newHead;
    }
```

62. Unique Paths, Medium

http://blog.csdn.net/linhuanmars/article/details/22126357

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).
The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).
How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?
Note: *m* and *n* will be at most 100.
Subscribe to see which companies asked this question

Key: DP. 用 res[i][j]表示從棋盤的[0][0]位置到[i][j]的路徑數. 遞推式是 res[i][j] = res[i-1][j] + res[i][j-1]. 寫為 1d-DP. 答案是將 res 的長度定義為 n 的, E2 也是這樣寫的. E3 最好將其定為 min(m, n). 注意使用 res[0] = 1 這個技巧, 這使得不用專門初始化首行和首列.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 沒用 res[0] = 1 的技巧, 在初始化 res 時, 即在上邊緣和左邊緣上時, 路徑應當只有一種, 我給 res 初始化的不是一, 另有指標小錯誤, 改後即通過.
E3 幾分鍾寫好, 一遍通過. E3 用的 2d-DP.

我们先说说 brute force 的解法，比较容易想到用递归，到达某一格的路径数量等于它的上面和左边的路径数之和，结束条件是走到行或者列的边缘。因为每条路径都会重新探索，时间复杂度是结果数量的量级，不是多项式的复杂度。代码如下. (tao: 此處省去代碼, 因為遞推或與以下代碼的遞推式不同, 看多了有點 confusing, 還是只看一個好)

上面的代码放到 LeetCode 中跑会超时，因为不是多项式量级的。其实上一步中递推式已经出来了，就是 res[i][j]=res[i-1][j]+res[i][j-1] (tao: 此式中 res[i][j]表示從初始的[0][0]位置到[i][j]的路徑數)，这样我们就可以用一个数组来保存历史信息，也就是在 i 行 j 列的路径数，这样每次就不需要重复计算，从而降低复杂度。用动态规划我们只需要对所有格子进行扫描一次，到了最后一个得到的结果就是总的路径数，所以时间复杂度是 O(m*n)。而对于空间可以看出我们每次只需要用到上一行当前列，以及前一列当前行的信息，我们只需要用一个一维数组存上一行的信息即可，然后扫过来依次更替掉上一行对应列的信息即可（因为所需要用到的信息都还没被更替掉），所以空间复杂度是 O(n)（其实如果要更加严谨，我们可以去行和列中小的那个，然后把小的放在内层循环，这样空间复杂度就是 O(min(m,n))，下面的代码为了避免看起来过于繁杂，就不做这一步了，有兴趣的朋友可以实现一下，比较简单，不过代码有点啰嗦）。实现的代码如下：

```
public int uniquePaths(int m, int n) {
    if(m<=0 || n<=0)
        return 0;
    int[] res = new int[n];
    res[0] = 1;
    for(int i=0;i<m;i++)
    {
        for(int j=1;j<n;j++) //注意起始為 j=1, 這也是為了保證下面 res[j-1]中 不會出現 res[-1]
        {
            res[j] += res[j-1]; //見後面我的解釋.
        }
```

```
    }
    return res[n-1];
}
```

上面的方法用动态规划来求解，如果我们仔细的看这个问题背后的数学模型，其实就是机器人总共走 m+n-2 步，其中 m-1 步往下走，n-1 步往右走，本质上就是一个组合问题，也就是从 m+n-2 个不同元素中每次取出 m-1 个元素的组合数。根据组合的计算公式，我们可以写代码来求解即可。代码如下. (tao: irrelevant, 已省去代碼)

63. Unique Paths II, Medium

http://blog.csdn.net/linhuanmars/article/details/22135231

Follow up for "Unique Paths":
Now consider if some obstacles are added to the grids. How many unique paths would there be?
An obstacle and empty space is marked as 1 and 0 respectively in the grid.
For example,
There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2.
Note: *m* and *n* will be at most 100.
Subscribe to see which companies asked this question

Key: DP. 遞推式還是 res[i][j]=res[i-1][j]+res[i][j-1]. 只是每次要判斷下 obstacleGrid[i][j]是否為 1, 否是, 則 res[i][j] = 0, 因為這樣的話, 在算到[i][j]右邊或下邊的格子的路徑數時, 就相當於沒加 res[i][j]. 注意使用 res[0] = 1 這個技巧, 這使得不用專門初始化首行和首列.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 沒用 res[0] = 1 的技巧, 改了很多次才通過, 這是因為對 '首行和首列上有 1 之情況' 沒處理好.
E3 幾分鍾寫好, 一次通過. E3 用的 2d-DP.

这道题跟 Unique Paths 非常类似，只是这道题给机器人加了障碍，不是每次都有两个选择（向右，向下）了。因为有了这个条件，所以 Unique Paths 中最后一个直接求组合的方法就不适用了，这里最好的解法就是用动态规划了。递推式还是跟 Unique Paths 一样，只是每次我们要判断一下([i][j], tao)是不是障碍，如果是，则 res[i][j]=0 (tao: 這樣的話, 在算到[i][j]右邊或下邊的格子的路徑數時, 就相當於沒加 res[i][j], 自己想, 很容易理解)，否则还是 res[i][j]=res[i-1][j]+res[i][j-1]。实现中还是只需要一个一维数组，因为更新的时候所需要的信息足够了。这样空间复杂度是是 O(n)（如同 Unique Paths 中分析的，如果要更加严谨，我们可以去行和列中小的那个，然后把小的放在内层循环，空间复杂度就是 O(min(m,n))，时间复杂度还是 O(m*n)。代码如下. 这里就不列出 brute force 递归方法的代码了，递归式和结束条件跟动态规划很近似，有兴趣的朋友可以写一下哈。

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    if(obstacleGrid == null || obstacleGrid.length==0 || obstacleGrid[0].length==0)
        return 0;
```

```
    int[] res = new int[obstacleGrid[0].length];
    res[0] = 1;
    for(int i=0;i<obstacleGrid.length;i++)
    {
        for(int j=0;j<obstacleGrid[0].length;j++)
        {
            if(obstacleGrid[i][j]==1)
            {
                res[j]=0;
            }
            else
            {
                if(j>0) //別忘了 j>0 這個要求, 這相當於 Unique Paths 這道題中 j 循環中起始值為 j=1
                    res[j] += res[j-1];
            }
        }
    }
    return res[obstacleGrid[0].length-1];
}
```

64. Minimum Path Sum, Medium

http://blog.csdn.net/linhuanmars/article/details/22257673

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.
Note: You can only move either down or right at any point in time.
Subscribe to see which companies asked this question

Key: 動態規劃. 本題要返回的是 path sum 最小的那條路徑 之 path sum. 递推式是 res[i][j]=min(res[i-1][j], res[i][j-1])+grid[i][j]。要先處理 grid 的第一行和第一列.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 但改了幾次才通過, 原因就是 grid 的第一行和第一列沒處理好. 好久沒刷題了, 手生了啊.
E3 按 2-d DP 寫的, 幾分鍾寫好, 一次通過.

这道题跟 Unique Paths，Unique Paths II 是相同类型的。事实上，这道题是上面两道题的 general 版本，是寻找带权重的 path。在 Unique Paths 中，我们假设每个格子权重都是 1，而在 Unique Paths II 中我们假设障碍格子的权重是无穷大，所以永远不会选择。剩下的区别就是这道题寻找这些路径中权重最小的，而不是总路径数。其实方法是一样的，还是使用动态规划，只是现在维护的不是路径数，而是到达某一个格子的最短路径（也就是权重之和最小）。而这个当前最短路径可以取前面一格和上面一格中小的最短路径长度得到，因此递推式是 res[i][j]=min(res[i-1][j], res[i][j-1])+grid[i][j]。总共进行两层循环，时间复杂度是 O(m*n)。而空间复杂度仍是使用 Unique Paths 中的方法来省去一维，是 O(m)，不了解的朋友可以看看哈。代码如下. 这道题是动态规划的经典题型，模型足够简单，所以可能在简单的面试（比如电面）中出现。总体来说还是比较容易的，递推式比较直观。

```
public int minPathSum(int[][] grid) {
    if(grid == null || grid.length==0 || grid[0].length==0)
        return 0;
```

```
    int[] res = new int[grid[0].length];
    res[0] = grid[0][0];
    for(int i=1;i<grid[0].length;i++)  //起始點為 i=1
    {
        res[i] = res[i-1]+grid[0][i]; //這是先處理 grid 的第一行, 在 Unique Paths 這道題中, 只先處理了[0][0]一個元素, 然後運氣好, 第一行和第一列都不用先處理了. 此題是 general 的情況, 要先把第一行和第一列都先處理了.
    }
    for(int i=1;i<grid.length;i++) //前面已經處理過第一行了, 做這裡起始點為 i=1
    {
        for(int j=0;j<grid[0].length;j++)
        {
            if(j==0)
                res[j] += grid[i][j]; //這是先處理 grid 的第一列, 若不先處理, 則下面 res[j-1]中會出現 res[-1]
            else
                res[j] = Math.min(res[j-1], res[j])+grid[i][j];
        }
    }
    return res[grid[0].length-1];
}
```

65. Valid Number, Hard

https://leetcodenotes.wordpress.com/2013/11/23/leetcode-valid-number/

Validate if a given string is numeric.
Some examples:
"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.
Update (2015-02-10):
The signature of the C++ function had been updated. If you still see your function signature accepts a const char * argument, please click the reload button to reset your code definition.
Subscribe to see which companies asked this question

Key: 本題實在太多 corner case. 自己很難寫對, 還是直接背答案好(背本 key 中的步驟).

寫一個 valid(String s, boolean hasDot)函數, 其作用為: 若 hasDot 為 true(已確認不是 false. 這點要注意, 因為跟語義相反, 這樣做也是被逼的), 則判斷 s 是否是一個 valid 的整數(即不含'.'), 若是則返回 true, 否則返回 false; 若 hasDot 為 false, 則判斷 s 是否是一個 valid 的小數(即含一個'.'), 若是則返回 true, 否則返回 false. hasDot 的意思即字面意思, 表示 s 中是否已有'.'

在 isNumber(s)函數(即主函數)中, 先按 e 將輸入的 String 分為兩部分(這兩部分放在數組 t 中). 然後調用 valid(t[0], false)和 valid(t[1], true). 更細節地講, 即在 isNumber 函數中, 首先 trim → 排除 s 為"3e"這樣的

→ 按 e 分 → 排除分得的不為兩個(即有多個 e)之情況 → 調用 valid(t[0], false)和 valid(t[1], true).

在 valid(s, hasDot)函數中, 跳過最前的+- → 排除 s 為空或 s 只有'.' → 掃描 s, 看當前字符是否為'.' 若是'.', 則 return 或 update hasDot; 否不是'.', 則判斷當前字符是否為數字(此處數字的意思為 0 到 9 inclusive 的整數).

2e+02
003
-003
00.
2e
e2
00
“1 ” (1 後有個空格), true
“ ” (即空格), false
“. .” (兩.之間有個空格), false
“. 1”(.和 1 之間有個空格), false
“1 .”(1 和.之間有個空格), false

History:
E1 直接看的答案. E1 在大吵架的過程中做的此題(本題很煩), 以此作記念. 以後找到工作了不要忘了現在的生活.
E2 寫得比較長, 但總是通不過, 後來看的答案.
E3 時間不夠, 故直接看的答案. 本題是 E3 唯一一道直接看答案的題目.

tao: 本題通過率只有 11.4%, 為 leetcode 中最低. Code Ganker 的代碼不太簡潔, 故不用 Code Ganker 的
Code Ganker: 这种题目一般在面试 tester 职位中比较常见，需要考虑比较多的边界情况，更多的是在寻找 bug 的感觉

这就是典型的 try and fail，尼玛谁知道什么算是数什么不算啊？让我用眼睛看我也不知道啊！总之，试出来的规则是这样的：
•AeB 代表 A * 10 ^ B
•A 可以是小数也可以是整数，可以带正负号
•.35, 00.神马的都算 valid 小数；就”.”单独一个不算
•B 必须是整数，可以带正负号
•有 e 的话，A,B 就必须同时存在
算法就是按 e 把字符串 split 了，前面按 A 的法则做，后面按 B 做。

1 | Wei
March 25, 2014 at 8:51 pm
for 循环里的 hasDot 写的真是太好了，我还单独开了个 int 来计算 A 部分中 dots 的数量。。。这个是我看到的比较简洁好理解的版本，谢谢分享。

```
public boolean isNumber(String s) {
    s = s.trim(); //trim 見 java 書 p411 注釋. 注意是 s=s.strim(), 而不是直接一句 s.trim(), 這與下面的 s=s.substring(1)一樣.

   if (s.length() > 0 && s.charAt(s.length() - 1) == 'e')
```

```
        return false; //avoid "3e" which is false

    String[] t = s.split("e"); //注意是"e", 而不是'e', 因為 split()的參數是 regular expression, 而不是 char. 參見 Java 書 p412.

    if (t.length == 0 || t.length > 2)
        return false;

    boolean res = valid(t[0], false); //此處和下面 valid(t[1], true)中的 false 和 true 的直觀意義不明顯, 看了後面的 valid 函數就知道甚麼意思了.

    if (t.length > 1)
        res = res && valid(t[1], true);

    return res;
}

//valid(String s, boolean hasDot)函數, 其作用為: 若 hasDot 為 true(已確認不是 false. 這點要注意, 因為跟語義相反, 這樣做也是被逼的), 則判斷 s 是否是一個 valid 的整數(即不含'.'), 若是則返回 true, 否則返回 false; 若 hasDot 為 false, 則判斷 s 是否是一個 valid 的小數(即含一個'.'), 若是則返回 true, 否則返回 false.  hasDot 的意思即字面意思, 表示 s 中是否已有'.'
private boolean valid(String s, boolean hasDot) {

    if (s.length() > 0 && (s.charAt(0) == '+' || s.charAt(0) == '-')) //avoid "1+", "+", "+."
    s = s.substring(1);//substring 見下面注釋, 注意不是直接一句 s.substring(1);因為 substring()返回的是 a new string.

    char[] arr = s.toCharArray();

    if (arr.length == 0 || s.equals("."))
        return false;
//注意 leetcode 中必須用 s.equals("."), 若用 s==".", 則通不過(即輸入為"."時, 得到的最終結果為 true). 而在我電腦上得到的結果為 false. Java 書 p410 的說法是與我電腦的行為是一致的. leetCode 應該是用不同的 Java 標准.

    for (int i = 0; i < arr.length; i++) {
//以下的 if(hasDot)確實寫得很聰明. 可分上面 valid(t[0], false)和 valid(t[1], true)兩種情況來想: 對於 valid(t[0], false), 下面實際上是在判斷'.'的數量, 若多於一個'.', 則 return false. 對於 valid(t[1], true), 只要有一個'.'則 return false

        if (arr[i] == '.') {
            if (hasDot)
                return false;
            hasDot = true;
        } else if (!('0' <= arr[i] && arr[i] <= '9')) { //這樣寫不等式還蛮符合手寫習慣的, 我以後也可以這樣寫

            return false;
        }
```

```
    }

    return true;
}
```

66. Plus One, Easy

http://blog.csdn.net/linhuanmars/article/details/22365957

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.
Subscribe to see which companies asked this question

Key: 簡單. 題目的意思是 輸入數組代表的數 是十进製的, 不是二进製. digit[0]是最高位. 注意可以直接在輸入數組裡改, 這樣空間複雜度一般情況為 O(1). 只有一種情況要用新數組來保存結果, 那就是輸入數組每個元素都是 9, 此時結果比輸入的數多一位.

History:
E1 的代碼很快寫好, 一次通過. 但我用了一個額外的數組放結果, 空間複雜度為 O(n). 而 Code Ganker 是直接在輸入數組裡改, 空間複雜度一般情況為 O(1). 所以下面用 Code Ganker 的代碼.
E2 幾分鍾寫好, 但改了幾次才通過, 因為 E2 最開始是這樣寫的:

```
digits[i] = (digits[i] + carry) % 10;
carry = (digits[i] + carry) / 10;
```

導致第二行的 digits[i]已經被新值覆蓋了, 若交換這兩句的順序, carry 也會被覆蓋, 後來改為以下的才通過, 以後都按以下那樣寫:

```
sum = (digits[i] + carry);
carry = sum / 10;
digits[i] = sum % 10;
```

E3 幾分鍾寫好, 本來可以一次通過, 結果不小心在 for 中將 i--寫為了 i++, 改後即通過. E3 沒寫成 in-place 的, 但若要求寫為 in-place, 也可以很容易寫出.

这是一道比较简单的题目，对一个数组进行加一操作。但是可不要小看这个题，这个题被称为“Google 最喜欢的题”，因为在 google 面试中出现的频率非常高。我们先说说这道题的解法。思路是维护一个进位，对每一位进行加一，然后判断进位，如果有继续到下一位，否则就可以返回了，因为前面不需要计算了。有一个小细节就是如果到了最高位进位仍然存在，那么我们必须重新 new 一个数组，然后把第一个为赋成 1（因为只是加一操作，其余位一定是 0，否则不会进最高位）。因为只需要一次扫描，所以算法复杂度是 O(n)，n 是数组的长度。而空间上，一般情况是 O(1)，但是如果数是全 9，那么是最坏情况，需要 O(n)的额外空间。代码如下. 我自己在 Google 的电面中就遇到了这道题，我觉得为什么 Google 喜欢的原因是后续可以问一些比较基本的扩展问题，比如可以扩展这个到两个数组相加，或者问一些 OO 设计，假设现在要设计一个 BigInteger 类，那么需要什么构造函数，然后用什么数据结构好，用数组和链表各有什么优劣势。这些问题虽然不是很难，但是可以考到一些基本的理解，所以平时准备有机会还是可以多想想哈。

楼主你好，是不是先把 digits 这个数组复制一下比直接把参数数组 digits 改了稍微更好些呢？

回复阿斯顿福吖：这个操作一般是服务于 BigInteger 的高精度类的，所以省空间还是主要出发点，如果先复制就意味着每次都要线性(tao: 原來他經常說的線性就是 O(n))的额外空间，还是不好的哈～ 只有在逼不得已再进行申请空间会好一些～

```
public int[] plusOne(int[] digits) {
    if(digits == null || digits.length==0)
        return digits;
    int carry = 1; //注意將 carry 初始化為 1 的技巧. 這樣就不用單獨處理 i=digits.length-1 時的情況(which was what E2 did). 這是因為將要加的那個 1 當成 carry 了.
    for(int i=digits.length-1;i>=0;i--)
    {
        int digit = (digits[i]+carry)%10;
        carry = (digits[i]+carry)/10;
        digits[i] = digit;
        if(carry==0)
            return digits;
    }
    int [] res = new int[digits.length+1];
    res[0] = 1;
    return res;
}
```

67. Add Binary, Easy

Given two binary strings, return their sum (also a binary string).
For example,
a = "11"
b = "1"
Return "100".
Subscribe to see which companies asked this question

Key: 按常規的加法算法寫, 即若算 c=a+b:

```
sum = aDigit + bDigit + carry;
cDigit = sum % 2;
carry = sum / 2;
```

按從右到左的順序加. 注意輸入的一個數比另一個長時, 以及結果中最高位還要进一位時(如 11+10).

History:
E1 的代碼在 leetcode 中是一次通過, 但有點繁鎖, 因為我先把輸入轉成 StringBuilder 再轉成 charArray 再轉成 intArray, 再把結果轉回 char. 後來看了 Code Ganker 的代碼, 才知道 StringBuilder 原來可以直接 append(int), 就把我最初的代碼改进了. Code Ganker 的方法稍有不同, 我最開始就把輸入 inverse 了, 而他沒有, 他是直接反序處理的. 不過差別不大, 我的反而更好理解. 另外, 對於輸入的一個數比另一個長時的處理, 我的方法比 Code Ganker 要簡潔. 所以整體上我的代碼比 Code Ganker 的短. Code Ganker 說的也沒甚麼比較有用的, 所以沒 copy 他的話過來.

E2 較快寫好, 本來可以一次通過的, 結果粗心大意, 把一個本來應該在括號內的數 寫到括號外去了. 改後即通過. E2 的代碼是從右往左加的. 而 E1 改进後的代碼是從左往右加的, 所以最開始要 reverse a, b 兩個數. 所以 E2 的效率要比 E1 高.

E3 幾分鍾寫好, 改了一次才通過, 原因是 E3 最開始沒考慮到兩個數長度不一樣, 故對短的那個 String 取字符時 index 越界了. 改後即通過. E3 也是從右往左加的, 跟 E2 一樣. E3 的代碼比 E2 短, 但 E3 用了額外的 char 數組, 故空間為 O(N). E2 一個數組都沒用, 故空間為 O(1). 雖然 E3 需要的空間比 E2 多, 但 E3 的代碼更短更

好寫, 故以後用 E3 代碼.

E1 改进後的代碼(不用):

```
 public static String addBinary(String a, String b) {
  StringBuilder asb = new StringBuilder(a);
  StringBuilder bsb = new StringBuilder(b);
  StringBuilder res = new StringBuilder();
  asb.reverse();
  bsb.reverse();

  int maxLen = Math.max(a.length(), b.length());
  int minLen = Math.min(a.length(), b.length());
  int digit;
  int carry = 0;
  int temp;

  for(int i=0; i<maxLen; i++) {

   if(i<minLen) {
     temp = (int) ((asb.charAt(i) - '0') + (bsb.charAt(i) - '0'));
     digit = (temp + carry) % 2;
     carry = (temp + carry) / 2;
   } else { //短的那個數已經讀完了的情況
    temp = (int) ((a.length() > b.length()) ? asb.charAt(i) - '0' : bsb.charAt(i) - '0');
    digit = (temp + carry) % 2;
    carry = (temp + carry) / 2;
   }

    res.append(digit);
  }

  if(carry == 1) //結果中最高位還要进一位時(如 11+10)
   res.append('1');

  return res.reverse().toString();
 }
```

E2 的代碼(不用):

```
    public String addBinary(String a, String b) {
        if(a == null || b == null) return "0";
        if(a.length() == 0 || b.length() == 0) return "";
        if(a.length() < b.length()) return addBinary(b, a);
        // Now a is longer than b
        int lenDiff = a.length() - b.length();
        int carry = 0;
        StringBuilder res = new StringBuilder();

        //處理 a 和 b 都有的那段
        for(int i = b.length() - 1; i >= 0; i--) {
```

```
            int bDigit = (int) (b.charAt(i) - '0');
            int aDigit = (int) (a.charAt(i + lenDiff) - '0');

            int sum = aDigit + bDigit + carry;
            int cDigit = sum % 2;
            carry = sum / 2;
            res.insert(0, cDigit);
        }

        //b 已經走完了, a 還沒走完
        for(int i = a.length() - b.length() - 1; i >= 0; i--) {
            int aDigit = (int) (a.charAt(i) - '0');
            int sum = aDigit + carry;
            int cDigit = sum % 2;
            carry = sum / 2;
            res.insert(0, cDigit);
        }

        //最高位有进位時
        if(carry > 0) res.insert(0, carry);

        return res.toString();
    }
```

E3 的代碼(用它):

```
    public String addBinary(String a, String b) {
        if(a == null || b == null || a.length() == 0 || b.length() == 0) return "";
        if(a.length() < b.length()) return addBinary(b, a);
        int n = a.length();
        int lenDiff = a.length() - b.length(); //E2 也用的一模一樣的變量名: lenDiff
        char[] A = new char[n], B = new char[n], C = new char[n];

        for(int i = n - 1; i >= 0 ; i--) {
            A[i] = a.charAt(i);
            B[i] = (i >= lenDiff ? b.charAt(i - lenDiff) : '0');
        }

        int carry = 0;

        for(int i = n - 1; i >= 0; i--) {
            int ADigit = ((int) (A[i] - '0'));
            int BDigit = ((int) (B[i] - '0'));
            int sum = ADigit + BDigit + carry;
            carry = sum / 2;
            C[i] = (char) (sum % 2 + '0');
        }

        String res = new String(C);
```

```
    if(carry == 1) return ("1" + res);
    return res;
  }
```

68. Text Justification, Hard

http://blog.csdn.net/whuwangyi/article/details/21503207
Given an array of words and a length *L*, format the text such that each line has exactly *L* characters and is fully (left and right) justified.
You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly *L*characters.
Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.
For the last line of text, it should be left justified and no extra space is inserted between words.
For example,
words: ["This", "is", "an", "example", "of", "text", "justification."]
L: 16.
Return the formatted lines as:
[
  "This is an",
  "example of text",
  "justification. "
]
Note: Each word is guaranteed not to exceed *L* in length.
click to show corner cases.
Corner Cases:

- A line other than the last line might contain only one word. What should you do in this case? In this case, that line should be left-justified.

Subscribe to see which companies asked this question

Key: E3 的方法:
寫一個 split 函數, 用來得到每行的單詞; 再寫一個 put 函數, 用來在每個單詞之間加恰當的空格. 俱體的, 就是:

void split(String[] words, int maxWidth, int begin, List<List<String>> lines)的作用為: 將 words 數組從 begin 開始(inclusive)到末尾的這個子數組中的單詞, 分成一行一行的, 每行中的單詞就是最終結果中的每行的單詞(只是沒放空格的), 每一行是一個 List<String>, 比如題目中例子, 弄出來的一行就是[This, is, an], 然後將所有行放入 lines 中, 成為一個 List<List<String>>.

String put(List<String> line, int maxWidth)的作用就是, line 是 split 函數得到的一行, 如[This, is, an], put 函數將 line 中每個單詞之間加入恰當個數的空格, 使它成為最終結果中的一行那個樣子(以 String 的引式表示), 比如"This is an", 返回這個 String.

Corner case(除上面的兩步中說的之外):
輸入{""}, 0, 輸出要是[""], 而不是[].
輸入{""} , 2, 輸出要是[" "]
輸入{"a"}, 1.

words[i].length == maxWidth 時(且分 i 是否為 0),

History:
E1 直接看的答案.

E2 改了很多遍, 終於通過了, 臨時出現的 corner case 確實很多, 可能是我代碼不好,當時 E1 留下的 key 就是看以下的作者的文字, 我 E2 的 pack 函數返回的是一個 List<List<Integer>>呢, 以後還是按答案中的 pack 那樣寫.

E3 寫好後, 基本上只改了兩個地方(詳見代碼中)後就通過了, 一是最後一行左對齊, 所以要在右邊加一片空格, E3 忘了加; 二是 split 函數中, 每個單詞按最緊密的方式放, 互相之間還應有個空格, E3 在計數時忘了考慮這個空格. E3 的思想跟 Code Ganker 差不多, 但 Code Ganker 的 split 函數返回的是單詞在 words 數組中的指標, 而 E3 返回的是 List<List<String>>. 雖然如此, 但 E3 的代碼思路比較清晰, 比 Code Ganker 的好懂很多, 再加上本題是出了名難題, 所以以後都用 E3 的代碼.

群裡有人面試被問了此題, 還報怨了很久, 但我 E3 還是並沒太為難地做出來了.

Code Ganker 的方法的 key:
本題思路簡單但 corner case 多. 思路分为两步:
第一步(即 pack 函數)：将 words 分行，获得每一行的起始词和结尾词的索引；
第二步(即 convert 函數)：将分好行的词群转换为字符串行，主要就是要计算中间插入的空格数。这里需要对于只含单个词的行和最后一行都要进行特殊处理，因为空格分布的规则不一样。

int pack(String[] words, int start, int maxWidth)函數: start 即為輸出中某一行(如題目例子中的 example of text 這一行)的起始指標(如 example 在 words[]中的指標), pack(words, start, L)返回的是這一行的末尾單詞的指標(即 text 在 words[]中的指標)
String convert(String[] words, int start, int end, int maxWidth)函數 : 功能自明. 其中可用 end == words.length – 1 來判斷是否是最後一行(即題目例子中的 justification 一行, 想想即知: 最後一行不只一個單詞時, 也可以這樣判斷)

tao: 這道題 Code Ganker 的代碼不大好懂, 所以用以下比較好懂的代碼. Code Ganker 說的話也沒用甚麼比較有用的. Leetcode 題目下面還有個藍色的 clike to show corner cases, 別忘了點開看看.

这道题目说实在挺无聊了(tao: 我覺得不無聊啊, 本題就是 Word 裡面的文字兩邊對齊, 很實用啊, 但 corner case 確實 TMD 多)。断断续续地刷 LeetCode，发现其实里面 AC Rate 很低的题不一定都很难(tao: 本題 AC Rate 為 14.6%)，有的可能是题意本身比较模糊，各种 corner case 比较多而已。这题就是典型的思路简单但 AC Rate 低的题目。
我的思路分为两步，分别用自定义的两个函数 pack 和 convert 搞定，感觉这样代码结构会清晰些。
第一步(tao: 即 pack 函數)：将 words 分行，获得每一行的起始词和结尾词的索引；
第二步(tao: 即 convert 函數)：将分好行的词群转换为字符串行，主要就是要计算中间插入的空格数。这里需要对于只含单个词的行和最后一行都要进行特殊处理，因为空格分布的规则不一样。

Code Ganker 的代碼(不用):

```
public ArrayList<String> fullJustify(String[] words, int L) {
        ArrayList<String> ret = new ArrayList<String>();
        if (words == null || words.length == 0)
            return ret;
```

```
            int start = 0, end = pack(words, start, L);
            ret.add(convert(words, start, end, L));
            while (end != words.length - 1) { //注意 end 最大只能是 words.length-2, 因為下一行有 start=end+1. 當然此句也可以寫為 while(end < words.length -1)
                start = end + 1;
                end = pack(words, start, L);
                ret.add(convert(words, start, end, L));
            }
            return ret;
        }

        //以下為 int pack(String[] words, int start, int L)函數, start 即為輸出中某一行(如例子中的 example of text 這一行)的起始指標(如 example 在 words[]中的指標), pack(words, start, L)返回的是這一行的末尾單詞的指標(即 text 在 words[]中的指標)
        // pack a line and return the end of this line
        private int pack(String[] words, int start, int L) { //start 是 words 的指標
            int next = start; // the index of the next word, 返回的就是 next
            int length = words[next].length();

            // always try to include the next word plus a padding space (greedy packing)
            while (next + 1 < words.length
                    && length + 1 + words[next + 1].length() <= L)
                length += words[++next].length() + 1; //++i 返回改變後的值, i++返回改變前的值
            return next;
        }

        //以下為 convert 函數
        // convert multiple words along with extra padding space into a string of length L
        private String convert(String[] words, int start, int end, int L) { //start 和 end 是 words 的指標
            StringBuilder sb = new StringBuilder();

            // if this line only contains one word
            if (start == end) {
                sb.append(words[start]);
                for (int i = 0; i < L - words[start].length(); i++) {
                    sb.append(" "); //Java 書 p417 說 append(' ')也可以, 實踐也表明可以.
                }
                return sb.toString();
            }
            // if the line is the last line, the space distribution rule is different
            else if (end == words.length - 1) {
```

```
                int curLen = 0;
                for (int i = start; i < end; i++) {
                        sb.append(words[i]);
                        sb.append(" ");
                        //以上兩句可寫為 sb.append(words[i]+" ");
                        curLen += words[i].length() + 1;
                }
                sb.append(words[end]);
                curLen += words[end].length();

                for (int i = 0; i < L - curLen; i++) {
                        sb.append(" ");
                }
                return sb.toString();
        }

        // calculate the lengths of padding space
        int totalLen = 0, numOfSpaces = end - start;
        for (int i = start; i <= end; i++)
                totalLen += words[i].length();
        int lenOfPaddingSpace = (L - totalLen) / numOfSpaces;
        int numOfExtraSpaces = (L - totalLen) % numOfSpaces;

        // construct the line
        int count = 0; // count of the extra spaces
        for (int i = start; i < end; i++) {
                sb.append(words[i]);
                for (int j = 0; j < lenOfPaddingSpace; j++)
                        sb.append(" ");
                if (count < numOfExtraSpaces)
                        sb.append(" ");
                count++;
        }
        sb.append(words[end]);

        return sb.toString();
    }
```

E3 的代碼:

```
  public List<String> fullJustify(String[] words, int maxWidth) {
```

```
        List<String> res = new ArrayList<>();
        if(words.length == 0) return res;

        List<List<String>> lines = new ArrayList<List<String>>();
        split(words, maxWidth, 0, lines);

        for(int i = 0; i < lines.size() - 1; i++)
            res.add(put(lines.get(i), maxWidth));

        //處理最後一行:
        StringBuilder sb = new StringBuilder();
        List<String> lastLine = lines.get(lines.size() - 1);

        for(int i = 0; i < lastLine.size(); i++) {
            sb.append(lastLine.get(i));
            if(i < lastLine.size() - 1) sb.append(" ");
        }

        int moreSpaces = maxWidth - sb.length();

        for(int i = 0; i < moreSpaces; i++)
            sb.append(" "); //最後一行左對齊, 所以這是在加右邊的那片空格. E3 最開始忘了加. (E3 error 1/2)

        res.add(sb.toString());

        return res;
    }

//void split(String[] words, int maxWidth, int begin, List<List<String>> lines)的作用為: 將 words 數組從 begin 開始(inclusive)到末尾的這個子數組中的單詞, 分成一行一行的, 每行中的單詞就是最終結果中的每行的單詞(只是沒放空格的), 每一行是一個 List<String>, 比如題目中例子, 弄出來的一行就是[This, is, an], 然後將所有行放入 lines 中, 成為一個 List<List<String>>.
    private void split(String[] words, int maxWidth, int begin, List<List<String>> lines) {
        if(begin >= words.length) return;
        int i = begin;
        int lineLen = 0;
        List<String> line = new ArrayList<>();

        while(i < words.length && lineLen + words[i].length() <= maxWidth) {
            line.add(words[i]);
            lineLen += words[i].length() + 1; //加 1 是因為每個單詞按最緊密的方式放, 互相之間還應有個空格. E3 最開始忘了加 1. (E3 error 2/2)
            i++;
        }

        if(!line.isEmpty()) lines.add(line); //此行是必需的, 否則通不過.

        split(words, maxWidth, i, lines);
    }
```

```
//String put(List<String> line, int maxWidth)的作用就是, line 是 split 函數得到的一行, 如[This, is, an], put 函數將 line 中每個單詞之間加入恰當個數的空格, 使它成為最終結果中的一行那個樣子(以 String 的引式表示), 比如"This     is     an", 返回這個 String.
   private String put(List<String> line, int maxWidth) {
      StringBuilder sb = new StringBuilder();

      int nChars = 0;
      for(String s : line) nChars += s.length();
      int nSpacesTotal = maxWidth - nChars;
      int nGaps = line.size() - 1;

      sb.append(line.get(0));

      if(line.size() == 1) {
         for(int i = 0; i < nSpacesTotal; i++) sb.append(" ");
         return sb.toString();
      }

      int nSpacesEachGap = nSpacesTotal / nGaps;
      int nSpacesExtra = nSpacesTotal % nGaps;

      for(int i = 1; i < line.size(); i++) {
         for(int j = 0; j < nSpacesEachGap; j++) sb.append(" ");
         if(i <= nSpacesExtra) sb.append(" ");
         sb.append(line.get(i));
      }

      return sb.toString();
   }
```

69. Sqrt(x), Medium

Implement int sqrt(int x).
Compute and return the square root of *x*.
Subscribe to see which companies asked this question

Key: 用二分法，注意 if(m*m<=x)會越界，可寫為 if(m<=x/m).

History:

E1 最初的代碼在 leetcode 也通過, 但我最初寫成 if(m*m<=x), 在 leetcode 中越界了, 後來只好把 l, r 和 m 都改成了 long, 才通過(E2 發現九章算法也是這樣做的). 後來看了 Code Ganker 的代碼, 才知道寫成 if(m <= x/m)這個技巧. 下面的 if(m <=x/m && (m+1)>x/(m+1)) 也一樣, 下面用我 E2 被改进後的代碼(實際上是九章的代碼)(E2 的代碼跟 E1 基本一樣, 但更簡練, 下面帖的其實是 E2 的代碼). Code Ganker 的方法和我的差不多. 但 code Ganker 說的還是要看看.

E2 幾分鍾寫好, 但改了很多次才通過, 原因是忘了寫 '當 m 滿足條件時直接返回 m' 這點, 可能是時間久了, 對

二分法不熟了.

E3 在 OJ 上也很快寫好, 改了很多次才通過, 原因是被對越界沒處理好.

Code Ganker:

这是一道数值处理的题目，和 Divide Two Integers 不同，这道题一般采用数值中经常用的另一种方法：二分法。基本思路是跟二分查找类似，要求是知道结果的范围，取定左界和右界，然后每次砍掉不满足条件的一半，知道左界和右界相遇。算法的时间复杂度是 O(logx)，空间复杂度是 O(1)。代码如下(tao: 已省略)．

二分法在数值计算中非常常见，还是得熟练掌握。其实这个题目还有另一种方法，称为牛顿法。不过他更多的是一种数学方法，算法在这里没有太多体现，不过牛顿法是数值计算中非常重要的方法，所以我还是介绍一下。不太了解牛顿法基本思想的朋友，可以参考一下牛顿法-维基百科。一般牛顿法是用数值方法来解一个 f(y)=0 的方程（为什么变量在这里用 y 是因为我们要求的开方是 x，避免歧义）。对于这个问题我们构造 f(y)=y^2-x，其中 x 是我们要求平方根的数，那么当 f(y)=0 时，即 y^2-x=0,所以 y=sqrt(x),即是我们要求的平方根。f(y)的导数 f'(y)=2*y，根据牛顿法的迭代公式我们可以得到 y_(n+1)=y_n-f(n)/f'(n)=(y_n+x/y_n)/2。最后根据以上公式来迭代解以上方程。（tao: 代碼已省略）．

实际面试遇到的题目可能不是对一个整数开方，而是对一个实数。方法和整数其实是一致的，只是结束条件换成左界和右界的差的绝对值小于某一个 epsilon（极小值）即可。这里注意一个小问题，就是在 java 中我们可以用==来判断两个 double 是否相等，而在 C++中我们则需要通过两个数的绝对值差小于某个极小值来判断两个 double 的相等性。实际上两个 double 因为精度问题往往是不可能每一位完全相等的，java 中只是帮我们做了这种判定。

比较典型的数值处理的题目还有 Divide Two Integers，Pow(x,n)等，其实方法差不多，一般就是用二分法或者以 2 为基进行位处理的方法。

```
   public int mySqrt(int x) {
      if(x <= 0) return 0;
      if(x == 1) return 1;

      int l = 0, r = x;

      while(l < r) { //寫成 while(l <= r)也能通過. 另外, 寫為 while(l + 1 < r)也能通過, 此即九章算法之模版, 以後都這樣寫.
         int m = (l + r) / 2;

         if(m <= x / m && m + 1 > x / (m + 1)) return m;

         if(m > x / m) r = m;
         else l = m;
      }

      return l;
   }
}
```

下面是九章算法對本題的答案(可通過), 以後都按這個模版寫 (William 和 Code Ganker 的模版都跟這個不一樣, 不用他們的):

```
public int sqrt(int x) {
    // find the last number which square of it <= x
    long start = 1, end = x; //寫成 start = 0 也能通過.

    while (start + 1 < end) {
        long mid = start + (end - start) / 2; //即 mid = (start + end) / 2
        if (mid * mid <= x) {
            start = mid;
        } else {
            end = mid;
        }
    } //end of while

    if (end * end <= x) {
        return (int) end; //必須將 long 轉換為 int, 否則 leetcode 中報錯: incompatible type
    }

    return (int) start;
}
```

70. Climbing Stairs, Easy

You are climbing a stair case. It takes *n* steps to reach to the top.
Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?
Subscribe to see which companies asked this question

Key: 題意是樓梯共有 n 階, 而不是說人總共必須跨 n 次.
遞推式為 f(n) = f(n-1) + f(n-2), 遞歸方法會超時, 用動態規劃就好. 遞推式為 f(1) = 1 , f(2) = 2 , f(3) = 3 , f(4) = 5 , f(n) = f(n-1) + f(n-2) (我想出來的), 因為最後一跨只有兩種可能: 要麼是這一跨只有一步(此時前面共跨了 f(n-1)次), 要麼是這一跨有兩步(此時前面共跨了 f(n-2)次). 跟 Fibonacci 數列的遞推式一樣, 但前幾項不同.

History:
E1: 我用遞歸方法寫過, 但在 leetcode 裡超時了. 然後改用動態規劃, 幾分鐘就寫好, 一次通過了. 以下用我的代碼. Code Ganker 的方法和我差不多, 以下是他說的, 可以看看.
E2 五分鍾寫好, 一次通過.
E3 最開始用的一個數組(知道可以按兩個變量寫), 三分鍾寫好, 居然沒一次通過, 因為在 res[1] = 2 前忘了加 if(n >= 2), 改後即通過. 然後又重新按兩個變量寫, 三分鍾寫好, 一次通過.

Code Ganker:

这道题目是求跑楼梯的可行解法数量。每一步可以爬一格或者两个楼梯，可以发现，递推式是 f(n)=f(n-1)+f(n-2)，也就是等于前一格的可行数量加上前两格的可行数量。熟悉的朋友可能发现了，这个递归式正是斐波那契数列的定义，不熟悉的朋友可以看看 Wiki – 斐波那契数列。根据这个定义，其实很容易实现，可以用递归或者递推都是比较简单的，下面列举一下递推(tao: 注意是递推不是递归, 其實就是我那種動態規劃的

方法)的代码 (tao: 已省略).

可以很容易判断，上面代码的时间复杂度是 O(n)，面试一般都会实现一下，不过还没完，面试官会接着问一下，有没有更好的解法？还真有，斐波那契数列其实是有 O(logn)的解法的。根据 wiki 我们知道，斐波那契数列是有通项公式的，如下：

$$a_n = \frac{\sqrt{5}}{5} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

所以如果我们用 Pow(x, n)中介绍的分治法来求解这个 n 次幂的话可以完成 O(logn)的求解。还有另一种理解方法就是斐波那契数列的线性代数解法（参见 Wiki - 斐波那契数列），可以看到迭代是一个二乘二的简单矩阵，数列的第 n 个数就是求解这个矩阵的 n-2 次幂，同样用分治法就可以完成 O(logn)的求解。
这是对于斐波那契数列问题的一般面试过程，先实现一下通常的 O(n)的解法，然后再了解一下是否知道有 O(logn)的解法，一般不要求实现，知道就行，不过其实实现也不是很难，有兴趣的朋友可以练习一下哈。

```
 public static int climbStairs(int n) {
  if(n<=0)
   return 0;

  if(n <= 3)
   return n;

  int prepre = 1;
  int pre = 2;
  int cur = 0;

  for(int i=3; i<=n; i++) {
   cur = pre + prepre;
   prepre = pre;
   pre = cur;
  }

  return cur;
 }
//f(1) = 1, prepre
//f(2) = 2, pre, prepre
//f(3) = 3, cur, pre
//f(4) = 5,        cur
//f(n) = f(n-1) + f(n-2)
```

71. Simplify Path, Medium

https://leetcodenotes.wordpress.com/2013/10/31/leetcode-simplify-path-unix%E7%BB%9D%E5%AF%B9%E8%B7%AF%E5%BE%84simplify/

Given an absolute path for a file (Unix-style), simplify it.
For example,
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"

click to show corner cases.
Corner Cases:

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

Subscribe to see which companies asked this question

Key:
1.先用/来 split string: String[] split = path.split("/") (見 java 書 p412, 可以不見)
2.然后看每一小段(設叫 s)，若是”.”或者是“”（说明两个/连着），s 不入栈；若是”..”，pop；若是正常，push s. 注意用 str.equals(), 不要用== (見 java 書 p410).
3.最后再用 string builder 把”/”和栈中元素一个一个连起来。

Corner case: 除上面提到的之外, 還有”/”, “///”, 都返口”/”
“/../” 返回”/..”
"/home/../../..", 返回”/../.."
"/."返回"/", "/.."返回"/", "/..."返回"/...", ("/.."返回"/"的事, 我也不理解為甚麼要這麼返回, 但確實是這樣)

History:
E1 直接看的答案.
E2 改了很多遍後通過.
E3 沒多久就寫好, 最開始用的 LinkedList, 且同時用 Queue 的 offer()函數和 Stack 的 pop()函數, 結果 pop()函數的作用跟我想像的不一樣(詳見 Java p806), 後來改用 Deque, 就通過了(其它地方沒改). 答案中用 Stack 也可以, 所以以後還是按答案那樣寫.

Tao: 對題意的解釋可以先看下面 Code Ganker 說的.
这个题一开始自己做死了，后来 bf 给讲用 stack 之后就一遍过了，代码也简单很多。思路
1 先用/来 split string
2.然后看每一小段，若是”.”或者是“”（说明两个/连着），不入栈；若是”..”，pop；若是正常，push
3.最后再用 string builder 把”/”和栈中元素一个一个连起来。

Tao: 別忘了看題目中的”click to show corner cases”.

Code Gankder 的代碼太長, 故不用他的代碼. 但他的思路和這裡是差不多的, 可以看看他說的:
Code Ganker:
这道题目是 Linux 内核中比较常见的一个操作，就是对一个输入的文件路径进行简化。思路比较明确，就是维护一个栈，对于每一个块（以‘/’作为分界）进行分析，如果遇到‘../’则表示要上一层，那么就是进行出栈操作，如果遇到‘./’则是停留当前，直接跳过，其他文件路径则直接进栈即可。最后根据栈中的内容转换成路径即可（这里是把栈转成数组，然后依次添加）。时间上不会超过两次扫描（一次是进栈得到简化路径，一次是出栈获得最后结果），所以时间复杂度是 O(n)，空间上是栈的大小，也是 O(n)。代码如下(tao: 已省略)
这道题其实还有一种做法，不需要维护栈，也就是不用额外空间，但是要对字符位置进行比较好的记录或者回溯，可能会多扫描一次，但是不会增加时间复杂度的量级。不过那个方法虽然对于空间上有提高，但是有很多细节的操作，并且没有什么算法思想，属于纯字符串操作，个人觉得意义不是很大，而且在面试中也很难正确写出来，还是比较推荐上述解法。

```
public String simplifyPath(String path) {
    Stack<String> s = new Stack<String>();
    String[] split = path.split("/");
    for (String a : split) {
```

```
        if (!a.equals(".") && !a.isEmpty()) {
            if (a.equals("..")) {
                if (!s.isEmpty()) {
                    s.pop();
                }
            } else {
                s.push(a);
            }
        }
    }
    StringBuilder sb = new StringBuilder();
    while (!s.isEmpty()) {
        sb.insert(0, s.pop());
        sb.insert(0, "/");
    }
    return sb.length() == 0 ? "/" : sb.toString();
}
```

72. Edit Distance, Hard

http://blog.csdn.net/linhuanmars/article/details/24213795

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

Key: DP. 用 res[i][j]表示 word1 的前 i 個字符(即 word1[0, 1… i-1])轉化成 word2 的前 j 個字符要編輯的最少步數. 遞推式為以下兩種情況: 若 word1[i-1] = word2[j-1]，則 res[i][j]=res[i-1][j-1], 因為新加入的字符(即 word1[i-1]和 word2[j-1])不用編輯.

若 word1[i-1]!=word2[j-1]，則 res[i][j] = min(res[i-1][j], res[i][j-1], res[i-1][j-1])+1. E3 對此式之理解為: 若 word1[i-1]!=word2[j-1], 則為何這兩個字符不同? 可以有以下三種理解: 一是 word1 append 了一個 word1[i-1]字符(此時 res[i][j] = res[i-1][j] + 1), 二是 word2 append 了一個 word2[j-1]字符(此時 res[i][j] = res[i][j-1] + 1), 三是將 word1[i-1]替換成了 word[j-1](此時 res[i][j] = res[i-1][j-1] + 1). 實際上到底是這三種的哪一種, 是說不清楚的, 即這三種都可以理解為對的, 按這三種理解會得出三個不同的 edit distance, 我們要找的是最小 edit distance 的那種. 實踐中, 直接記住此公式即可.

用 1d-DP, 注意 res 數組的長度要弄成 word1 和 word2 中較小的長度. Code Ganker 的 1d-DP 比 William 的 prev+curr 的 1d-DP 要好寫, 即遞推的左邊用 newRes[j], 即 newRes[j]=..., 而上一行的信自保存在 res 中. 而且更新數組時, 直接賦值引用, 不要一個一個元素地拷.

History:
E1 直接看的答案.
E2 寫的 1d-DP, 但用的一個 resPrev 來保存上一行的信息, 在更新 resPrev 時范了個錯誤, 查了好久才查出來,

改後即通過. 以後按 Code Ganker 那樣寫.
E3 最開始求出的遞推式為 res[i][j] = res[i-1][j-1], 結果通不過, 後來覺得可能應為 res[i][j] = min(res[i-1][j], res[i][j-1], res[i-1][j-1])+1, 一試居然通過了. E3 按 2d-DP 寫的.

这道题求一个字符串编辑成为另一个字符串的最少操作数，操作包括添加，删除或者替换一个字符。这道题难度是比较大的，用常规思路出来的方法一般都是 brute force，而且还不一定正确。这其实是一道二维动态规划的题目，模型上确实不容易看出来，下面我们来说说递推式。
我们维护的变量 res[i][j]表示的是 word1 的前 i 个字符(tao: 即 word1[0], word1[1], … word1[i-1])和 word2 的前 j 个字符编辑的最少操作数是多少。假设我们拥有 res[i][j]前的所有历史信息，看看如何在常量时间内得到当前的 res[i][j]，我们讨论两种情况：
1）如果 word1[i-1]=word2[j-1]，也就是当前两个字符相同，也就是不需要编辑，那么很容易得到 res[i][j]=res[i-1][j-1]，因为新加入的字符不用编辑；
2）如果 word1[i-1]!=word2[j-1]，那么我们就考虑三种操作(tao: 後面的 word1 和 word2 應該是寫反了, 評論中也這麼說)，如果是插入 word1，那么 res[i][j]=res[i-1][j]+1，也就是取 word1 前 i-1 个字符和 word2 前 j 个字符的最好结果，然后添加一个插入操作；如果是插入 word2，那么 res[i][j]=res[i][j-1]+1，道理同上面一种操作；如果是替换操作，那么类似于上面第一种情况，但是要加一个替换操作（因为 word1[i-1]和 word2[j-1]不相等），所以递推式是 res[i][j]=res[i-1][j-1]+1。上面列举的情况包含了所有可能性，有朋友可能会说为什么没有删除操作，其实这里添加一个插入操作永远能得到与一个删除操作相同的效果，所以删除不会使最少操作数变得更好，因此如果我们是正向考虑，则不需要删除操作。取上面几种情况最小的操作数，即为第二种情况的结果，即 res[i][j] = min(res[i-1][j], res[i][j-1], res[i-1][j-1])+1。
接下来就是分析复杂度，算法时间上就是两层循环，所以时间复杂度是 O(m*n)，空间上每一行只需要上一行信息，所以可以只用一维数组即可，我们取 m, n 中小的放入内层循环，则复杂度是 O(min(m,n))。代码如下.

上面代码用了 minWord, maxWord 是为了使得空间是 min(m,n)，细节做得比较细，面试的时候可能不用刻意这么做，提一下就好。
这道题目算是难度比较大的题目，所以在短时间的面试可能时间太紧了，所以也有变体。我自己在面试 Google 的时候，问的是如何判断 edit distance 是不是在 1 以内，返回 true 或 false 就可以了。这样一改其实就没有必要动态规划了，只需要利用距离只有 1 这一点进行判断就可以，大概思路就是只要有一个不同，接下来就不能再有不同了，有兴趣的朋友可以想想哈。

“如果是插入 word1，那么 res[i][j]=res[i-1][j]+1，也就是取 word1 前 i-1 个字符和 word2 前 j 个字符的最好结果，然后添加一个插入操作；如果是插入 word2，那么 res[i][j]=res[i][j-1]+1”
这里插入 word1 和插入 word2 写反了吧

回复 lzheng8：应该是没错的，res[i][j]=res[i-1][j]+1, 就是 word1 少一个字符，此时添加一个字符，则可以得到 i 和 j 的最好结果~

回复 Code_Ganker：楼主，res[i][j]=res[i-1][j]+1 代表 word1 的前 i-1 个字符转换到 word2 的前 j 个字符的最少步数，那么 word1 的第 i 个字符应该被删掉。所以文中是写反了哈

回复 yanyan300300：我也觉得楼主写反了

版主，能不能稍详细介绍一下 Google 判断 edit distance 是不是在 1 以内的思路？或者有代码的话我参考代码也可以，谢谢了！理论上是比这个简单，但是我确实没明白，sorry.

回复 shileicomeon：先用长度作为标准，如果长度相同，那么允许的就是一次修改操作， 如果长度差 1，那么就是一次插入操作，如果差得更多就肯定不是 1 以内了~。 接下来就是进行一次扫描~ 然后如果遇到有不同的，那么说明一次修改是无可避免了~ 按照上面的长度分情况，我们就知道是进行哪一种操作，接下来就继续匹配，不允许再出现不同了，违背就返回 false~ 代码我没有留了~ 这个思路应该比较清楚了哈~

回复 Code_Ganker：Very 清楚了！谢谢！

请问下 res[i] 的初始化为什么是 res[i] = i？谢谢~

回复 sinat_17730719：初始化的时候我们是在第 0 行，也就是另一个字符串一个字符都不取，即空字符串。 res[i]表示如何将当前字符串（前 i 个元素）变成一个空字符串， 所以我们可以进行 i 次删除（或者插入）操作，就可以使得当前字符串变成空字符串（或者空字符串变成当前字符串）， 有什么问题继续交流哈 ~

```
public int minDistance(String word1, String word2) {
    if(word1.length()==0)
        return word2.length();
    if(word2.length()==0)
        return word1.length();
    String minWord = word1.length()>word2.length()?word2:word1;  //後面就可以直接用 minWord.charAt(j), 我暈, 還可以這樣
    String maxWord = word1.length()>word2.length()?word1:word2;
    int[] res = new int[minWord.length()+1]; //為何是 minWord.length()+1?下面我馬上說.
//以下循環是處理 maxWord 取 0 個字符, minWord 取前 i 個字符時之情況. 可看出 i 可以等於 minWord.length(), 故前面分配 res 大小時為 minWord.length()+1. 注意 res[helo]中的 helo 表示 minWord 的指標, 後面也一直是這樣的約定.
    for(int i=0;i<=minWord.length();i++)
    {
        res[i] = i;
    }
//以下循環中 i 即為 maxWord 的指標, j 即為 minWord 的標. newRes[helo]中的 helo 也是 minWord 的指標. 這裡實際上是通過動態規劃之方法, 用一維數組表示二維數組(以前有道題也這樣用過, 注意體會): 即另一維的值保存在上一輪循環中.
//第 i 輪(即若 i=5 時, 我給它叫第 5 輪)時的 newRes[j+1]表示將 maxWord[0, 1, … i]和 minWord[0, 1, … j] convert 成一樣的之步數.
    for(int i=0;i<maxWord.length();i++)
    {
        int[] newRes = new int[minWord.length()+1];
        newRes[0] = i+1; //按我上面寫的"newRes[j+1]表示將...", 可知 newRes[0]是無意義的, 這裡的 newsRes[0]=i+1 完全是為下面的 newRes[j+1] = Math.min(res[j],Math.min(res[j+1],newRes[j]))+1 一句話在 j=0 時服務, 因為此時右邊會出現 newRes[0]. 在下面的二重循環中, 第 i 輪(即若 i=5 時, 我給它叫第 5 輪)中, 當 j=0 時, 我們要算 newRes[1], 即將 maxWord[0, 1, … i]和 minWord[0] convert 成一樣的之步數. 顯然 newRes[1]=i+1, 所以我們先在這裡把 newRes[0]賦值為 i+1, 然後在式子 newRes[j+1] = Math.min(res[j],Math.min(res[j+1],newRes[j]))+1 中就可以得出 newRes[1]=i+1.
        for(int j=0;j<minWord.length();j++)
        {
            if(minWord.charAt(j)==maxWord.charAt(i))
```

```
            {
                newRes[j+1]=res[j]; //此句話即表示 res[i][j+1] = res[i-1][j]
            }
            else
            {
                newRes[j+1] = Math.min(res[j],Math.min(res[j+1],newRes[j]))+1;

            }
        }
        res = newRes;
    }
    return res[minWord.length()];
}
```

73. Set Matrix Zeroes, Medium

http://blog.csdn.net/linhuanmars/article/details/24066199

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in place.
click to show follow up.
Follow up:
Did you use extra space?
A straight forward solution using O(*mn*) space is probably a bad idea.
A simple improvement uses O(*m* + *n*) space, but still not the best solution.
Could you devise a constant space solution?
Subscribe to see which companies asked this question

Key: 使用輸入數組的第 0 行和第 0 列來記錄所有 0 的位置, 空間複雜度為 O(1).

History:
E1 的代碼在 leetcode 中是一次通過, 但不是 in place 的.
E2 通過, 代碼跟以下 Code Ganker 的基本差不多. 最開始范了點小錯(見下面代碼中的黃色注釋), 不然可以一次通過的(注意 E1 的一次通過和 E2 的一次通過意思不一樣).
E3 很快寫好, 但改了兩個小錯誤後才通過. 一是把 matrix[0][i]寫成了 matrix[i][0], 二是在根据第一行和第一列 來在其它行列中設 0 時(即爆泡泡), 不小心將第一行和第一列也爆了.

Tao: 我的代碼在 leetcode 中是一次通過. 我的方法是先用一個 list 存放原數組中 0 的位置, 然後根据這個 list 去將輸入數組設 0. 空間複雜度為 O(0 的個數), 一般情況下比 O(mn)和 O(m+n)要好, 但最壞情況下(輸入數組全是 0)為 O(mn), 所以雖然通過, 但還是不太滿意. 注意題目要求 in place. Code Ganker 的 空間複雜度為 O(1), 所以還是用 Code Ganker 的代碼.

Tao: 這就是泡泡堂啊.

Gode Ganker:

这是一个矩阵操作的题目，目标很明确，就是如果矩阵如果有元素为 0，就把对应的行和列上面的元素都置为 0。这里最大的问题就是我们遇到 0 的时候不能直接把矩阵的行列在当前矩阵直接置 0，否则后面还没访问到的会被当成原来是 0，最后会把很多不该置 0 的行列都置 0 了。
一个直接的想法是备份一个矩阵，然后在备份矩阵上判断，在原矩阵上置 0，这样当然是可以的，不过空间复杂度是 O(m*n)，不是很理想。
上面的方法如何优化呢？我们看到其实判断某一项是不是 0 只要看它对应的行或者列应不应该置 0 就可以，所以我们可以维护一个行和列的布尔数组(tao: 意思可能是指兩個數組, 一個長度為輸入數組的行數, 另一個長度為輸入數組的列數, 當然也可以把它們合並為一個數組)，然后扫描一遍矩阵记录那一行或者列是不是应该置 0 即可，后面赋值是一个常量时间的判断。这种方法的空间复杂度是 O(m+n)。
其实还可以再优化，我们考虑使用第一行和第一列来记录上面所说的行和列的置 0 情况，这里问题是那么第一行和第一列自己怎么办？想要记录它们自己是否要置 0，只需要两个变量（一个是第一行，一个是第一列）就可以了。然后就是第一行和第一列，如果要置 0，就把它的值赋成 0（反正它最终也该是 0，无论第一行或者第一列有没有 0），否则保留原值。然后根据第一行和第一列的记录对其他元素进行置 0。最后再根据前面的两个标记来确定是不是要把第一行和第一列置 0 就可以了。这样的做法只需要两个额外变量，所以空间复杂度是 O(1)。
时间上来说上面三种方法都是一样的，需要进行两次扫描，一次确定行列置 0 情况，一次对矩阵进行实际的置 0 操作，所以总的时间复杂度是 O(m*n)。代码如下.

这道题也是 cc150 里面比较经典的题目，看似比较简单，却可以重重优化，最终达到常量空间。其实面试中面试官看重的是对于算法时间空间复杂度的理解，对优化的概念，这些常常比题目本身的难度更加重要，平常做题还是要对这些算法分析多考虑哈。

```
public void setZeroes(int[][] matrix) {
    if(matrix==null || matrix.length==0 || matrix[0].length==0)
        return;
    //rowFlag 和 colFlag 是用來決定最後是否要將第 0 行或第 0 列全設為 0:
    boolean rowFlag = false;
    boolean colFlag = false;
    //Set colFlag:
    for(int i=0;i<matrix.length;i++)
    {
        if(matrix[i][0]==0)
        {
            colFlag = true;
            break;
        }
    }
    //Set rowFlag:
    for(int i=0;i<matrix[0].length;i++)
    {
        if(matrix[0][i]==0)
```

```
        {
            rowFlag = true;
            break;
        }
    }
    //Set the first row and column:
    for(int i=1;i<matrix.length;i++)
    {
        for(int j=1;j<matrix[0].length;j++)
        {
            if(matrix[i][j]==0)
            {
                matrix[i][0] = 0;
                matrix[0][j] = 0;
//如果第 0 行或第 0 列本來就有 0 怎麼辦? 不用擔心, 因為那些 0 對應的行或列本來就要設為 0 的.
            }
        }
    }
    //根据第 0 行和第 0 列的信息來將 matrix 中對應的元素設為 0:
    for(int i=1;i<matrix.length;i++) //E2 最開始是寫成的 i 從 0 開始, 結果沒通過. 因為不能將第 0 行全抹成 0 了. 以下 j=1 同理. 否則 E2 可以一次通過的.
    {
        for(int j=1;j<matrix[0].length;j++)
        {
            if(matrix[i][0]==0 || matrix[0][j]==0)
                matrix[i][j] = 0;
        }
    }
    //若 colFlag 為 true, 則將第 0 列所有元素設為 0
    if(colFlag)
    {
        for(int i=0;i<matrix.length;i++)
        {
```

```
            matrix[i][0] = 0;
        }
    }
    //若 rowFlag 為 true, 則將第 0 行所有元素設為 0
    if(rowFlag)
    {
        for(int i=0;i<matrix[0].length;i++)
        {
            matrix[0][i] = 0;
        }
    }
}
```

74. Search a 2D Matrix, Medium.

http://blog.csdn.net/linhuanmars/article/details/24216235

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.
Subscribe to see which companies asked this question

Key: 二分查找. 用兩次二分查找. 第一次二分查找用來 找出 target 所在的行, 第二次二分查找用來 定出此行中 target 是否存在. 本題題意跟 240. Search a 2D Matrix II 之區別為: 本題中 '本行第一個數' 比 '上一行最後一個數' 大, 而 240 題的數組沒有這個限制. 240 題不是用二分查找, 跟本題其實沒有甚麼卵關係.

Corner case: {{1}}, {{1, 1}}

History:
E1: 這道題看起來簡單, 一看就是二分法, 我自己寫的時候是把 start 和 end 的間隔 offset 算出來, 然後 middle = offset/2, 再把 middle 對應的位置找出來. 寫了一上午, 結果最後在 leetcode 中是超時. 我的時間複雜度應該也是 log(m*n)=log(m)+log(n), 但可能太多其它計算了. 而且我的代碼中有不少比較複雜的公式, corner case 也多, 故不好. 以下用 Code Ganker 的小清新代碼.

E2 按九章模版寫好, 本來可以一次通過的, 但沒單獨考慮輸入數組只有一行之情況, 以及第二次二分查找後沒有考查邊界的情況. 改後即通過. E2 方法是用的以上 key 的方法, 九章也是這樣做的(還有一個只用一次 二分查找 的方法, 為了好懂, 我還是用兩次 二分查找 的方法), 但我 E2 的代碼更簡明好懂, 以後用我 E2 的代碼(在最後).

E3 改了兩個粗心錯誤後通過, 一是不小心將 m 寫成了 n, 二是 while 中的 < 不小心寫成了 >.

Code Ganker:

这道题是二分查找 Search Insert Position 的题目，因为矩阵是行有序并且列有序，查找只需要先按行查找，定位出在哪一行之后再进行列查找即可，所以就是进行两次二分查找。时间复杂度是 O(logm+logn)，空间上只需两个辅助变量，因而是 O(1)，代码如下.

二分查找是面试中出现频率不低的问题，但是很少直接考二分查找，会考一些变体，除了这道题，还有 Search in Rotated Sorted Array 和 Search for a Range，思路其实差不多，稍微变化一下即可，有兴趣可以练习一下哈。

Code Ganker 的代碼(不用):

```
public boolean searchMatrix(int[][] matrix, int target) {
    if(matrix == null || matrix.length==0 || matrix[0].length==0)
        return false;
    int l = 0;
    int r = matrix.length-1;
    while(l<=r)    // 最好寫成 l<=r, 不要寫成 l<r, 其它二分法也注意
    {
        int mid = (l+r)/2;
        if(matrix[mid][0] == target) return true;
        if(matrix[mid][0] > target)
        {
            r = mid-1;//注意沒必要寫成 r=mid, 因為前面已大 if(matrix[mid][0]==target)，故這裡它們不會相等了．其它二分法中也注意．這樣也記住了要先判斷 if(matrix[mid][0]==target), 以及前面用 matrix[mid][0]>target, 而不是 matrix[mid][0]>=target
        }
        else
        {
            l = mid+1;
        }
    }
    int row = r;
    if(row<0)
        return false;
    l = 0;
    r = matrix[0].length-1;
    while(l<=r)
    {
        int mid = (l+r)/2;
        if(matrix[row][mid] == target) return true;
        if(matrix[row][mid] > target)
        {
            r = mid-1;
```

```
        }
        else
        {
            l = mid+1;
        }
    }
    return false;
}
```

E2 的代碼(用它):

```
    public static boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return false;
        int m = matrix.length, n = matrix[0].length;
        if(target < matrix[0][0] || target > matrix[m - 1][n - 1]) return false;

        //First binary search, find the row
        int l1 = 0, r1 = m - 1;
        int row = 0;

        while(l1 + 1 < r1) {
            int m1 = (l1 + r1) / 2;

            if(matrix[m1][0] <= target && target <= matrix[m1][n - 1]) row = m1;

            if(matrix[m1][0] > target) r1 = m1;
            else l1 = m1;
        }

        //以上 while 結束後, l1 比 r1 小 1. 那麼 target 到底在第 l1 行, 還是 r1 行？以下 if-else 就是找出到底在 l1 還是 r1 行.
        if(target < matrix[r1][0]) row = l1;
        else row = r1;

        //Second binary search, find the column
        int l2 = 0, r2 = n - 1;

        while(l2 + 1 < r2) {
            int m2 = (l2 + r2) / 2;

            if(matrix[row][m2] == target) return true;

            if(matrix[row][m2] > target) r2 = m2;
            else l2 = m2;
        }

        //以上 while 沒有 cover 本行首尾, 以下 if 即處理本行首尾
        if(matrix[row][l2] == target || matrix[row][r2] == target) return true;

        return false;
```

}

75. Sort Colors, Medium

http://blog.csdn.net/linhuanmars/article/details/24286349

Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.
Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.
Note:
You are not suppose to use the library's sort function for this problem.
click to show follow up.
Subscribe to see which companies asked this question

Key: 用兩個指針, 一個(idx0)指在結果數組中的那堆 0 的下一個元素, 別一個(idx1)指在那堆 1 的下一個元素. 另外用一個指針從左向右掃描. 若掃到 0, 則 idx0 和 idx1 都右移一步, 並更新 0 區和 1 區的元素值. 若掃到 1, 則只有 idx1 右移一步, 並更新 1 區的元素值. 注意不需要 idx2 這麼個指針, 因為 0 區和 1 區之右的就是 2 區了.

History:
E1 沒通過.
E2 通過.
E3 很快寫好, 本來可以一遍通過的, 但不小心將 end0++寫在了 if(end1 > end0) nums[end1] = 1 前面, 導致 if 中的 end0 是++之後的值, 調換這兩句順序後即通過. E3 的方法跟 Code Ganker 一樣, 寫法稍有不同, 見 Code Ganker 代碼中.

这道题也是数组操作的题目，其实就是要将数组排序，只是知道数组中只有三个元素 0,1,2。熟悉计数排序的朋友可能很快就发现这其实就是使用计数排序，元素空间只需要三个元素即可。代码如下(tao: 已省略)

上面的代码是计数排序的标准解法，可以看到总共进行了三次扫描，其实最后一次只是把结果数组复制到原数组而已，如果不需要 in-place 的结果只需要两次扫描。
其实就算返回元素组也可以是两次扫描，这需要用到元素只有 0,1,2 的本质。我们知道 helper[i]中是包含着 0,1,2 的元素数量，我们只需要按照 helper[0,1,2]的数量依次赋值过来即可（每层循环把 helper[i]--，如果 helper[i]到 0 就 i++就可以了），只是这样就不是计数排序比较标准的解法，我希望还是复习一下。
这种方法的时间复杂度是 O(2*n)，空间是 O(k)，k 是颜色的数量，这里是 3。
上述方法需要两次扫描，我们考虑怎么用一次扫描来解决。其实还是利用了颜色是三种这一点，道理其实也简单，就是搞两个指针，一个指在当前 0 的最后一个下标，另一个是指在当前 1 的最后一个下标（2 不需要指针因为剩下的都是 2 了）。进行一次扫描，如果遇到 0 就两个指针都前进一步并进行赋值，如果遇到 1 就后一个指针前进一步并赋值。代码如下 .

上述方法时间复杂度还是 O(n)，只是只需要一次扫描，空间上是 O(1)（如果颜色种类是已知的话）。
这道题我觉得主要还是熟悉一下计数排序，计数排序是线性排序中比较重要的一种，关于排序要搞个专题专门的复习一下，很多排序的基本思想都对解题有帮助哈。

Tao: Code Ganker 上面給出的計收排序看不太懂，所以這裡沒 copy 下來. Leetcode 的 follow up 提到的 two-pass algorithm using counting sort 應該更接近以下網頁給出的第一種方法（看看即可）：
http://wlcoding.blogspot.com/2015/03/sort-colors.html?view=sidebar

以下是 Code Ganker 的第二種方法：

```
public void sortColors(int[] A) {
    if(A==null || A.length==0)
        return;
    int idx0 = 0;
    int idx1 = 0;
    for(int i=0;i<A.length;i++)
    {
        if(A[i]==0)
        {
            A[i] = 2;
            A[idx1++] = 1;
            A[idx0++] = 0;    //注意此句一定要在上句之後，否則結果是錯的，想想即知. E2 就是寫到上一句之前的, 代價是多寫了幾句, 其實問題不大, 也比較好懂. E3 也是寫到前面的, 只多寫了個 if, 參見下面 E3 代碼.
        }
        else if(A[i]==1)
        {
            A[i] = 2;
            A[idx1++] = 1;
        }
    }
}
```

E3 的代碼(只參考, 不用)

```
    public void sortColors(int[] nums) {
        if(nums == null || nums.length <= 1) return;
        int end0 = 0, end1 = 0; //end0 和 end1 跟 Code Ganker 的 idx0 和 idx1 意思一樣.

        for(int i = 0; i < nums.length; i++) {
            if(nums[i] == 0) {
                nums[end0] = 0;
                if(end1 > end0) nums[end1] = 1;
                end0++;
                end1++;
            } else if(nums[i] == 1) {
                nums[end1] = 1;
                end1++;
            }
        }

        for(int i = end1; i < nums.length; i++) nums[i] = 2;

    }
```

76. Minimum Window Substring, Hard

http://wlcoding.blogspot.com/2015/03/minimum-window-substring.html?view=sidebar

Given a string S and a string T, find the minimum window in S which will contain all the characters in

T in complexity O(n).
For example,
S = "ADOBECODEBANC"
T = "ABC"
Minimum window is "BANC".
Note:
If there is no such window in S that covers all characters in T, return the empty string "".
If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.
Subscribe to see which companies asked this question

Key: 難. 以下 William 說的很好, 表述能力跟我差不多.

History:
E1 直接看的答案.
E2 改了很多遍才通過. E2 表示本題自己寫還是有點難, 很多細節要注意.
E3 想到了類似方法, 但還是沒寫出來. E3 也表示本題很難.

tao: Williman 的代碼比 Code Ganker 的短, 而且代碼中有解釋, 故用 William 的. Code Ganker 說的也沒甚讨有用的, 故不 copy 到這裡.

William:
Time ~ O(M+N), Space ~ O(M) where N = lenS, M = lenT
先用一个 Hash Table 记录 T 里所有 char 的出现次数。
然后从左向右扫描 S：
•用两个指针（开头 prev 和结尾 i(即 S 循環中的 i)），i 在前找(即 i 在 prev 右邊)，到包含了所有 T 的 char 时停下；
•然后往前移动 prev，当碰到一个 T 中的元素，且从 prev 到 i 的 substring 中包含的该元素个数等于 T 中包含该元素的个数时(即 count == T.length()時)停下（解释如下）；
•此时得到一个符合条件的 substring，比较其长度和之前找到的最小长度，如果更小，则记录该长度和开头位置。

这里第二步的实现比较技巧，只用了一个 Hash Table：
在第一步每次找到 T 的 char 时就在 Hash Table 中将其对应的次数减 1，如果次数减为负，则表明从 prev 到 i 的 substring 中包含了多于 T 所包含的 char（比如，T 中只有 1 个 A，但可能目前的 substring 中有 5 个 A），所以当找到所有的 T 的 char 后，Hash Table 中所有 key 对应的次数都会小于等于 0。
接着，在第二步移动 prev 时，每遇到 T 的 char 后，又会将 Hash Table 中的次数再加 1（恢复过程），当某一个 key 的次数由非正变成正时(即 **if** (map.get(prevChar) > 0)時)，表明从 prev 到 i 的 substring 中包含的 key 的数量与 T 所包含的一致，也就是说 prev 不能再往前移动，此时得到了以 i 结尾的最短 substring。

```
public String minWindow(String S, String T) {
    if (S.length() == 0 || T.length() == 0 || S.length() < T.length())  return "";

    Map<Character, Integer> map = new HashMap<>();

    //以下為 T 循環
    for (int i = 0; i < T.length(); i++) {
        char c = T.charAt(i);
        if (!map.containsKey(c))    map.put(c, 1);
        else                        map.put(c, map.get(c) + 1);
```

```
    }

    //以下為S循環
    int count = 0, prev = 0, minStart = 0, minLen = Integer.MAX_VALUE;

    for (int i = 0; i < S.length(); i++) {
        char currChar = S.charAt(i);
        if (map.containsKey(currChar)) { // if currChar exists in T (這是指左邊的if)

            //(tao: 本題中作者的注釋都是管下面的)
            // record currChar (when all T's chars are found, all map's value <= 0)
            map.put(currChar, map.get(currChar) - 1);
             // stop adding length (count) if we have more currChar's than T does (map's value < 0)
            if (map.get(currChar) >= 0)  count++;

             //最開始的幾輪不會執行以下的while, 只有等S[prev, i] contains all T's chars時才會執行
            while (count == T.length()) {
// S[prev, i] contains all T's chars, now shrink the left end (prev) (tao: 此時i是正確的right end) (tao: 此注釋原來在while上一行, 我把它移到這裡來的)
                char prevChar = S.charAt(prev);
                if (map.containsKey(prevChar)) {   // if prevChar exists in T
                    map.put(prevChar, map.get(prevChar) + 1); // recover map by adding the value back
                    if (map.get(prevChar) > 0) {   // S[pre, i] is the min substring ending at i (tao: 因為若此時不是最短, 即prev可以再往右移, 那麼S[pre, i]中preChar的個數就沒有T那麼多了)
                        if (minLen > i - prev + 1) {
                            minLen = i - prev + 1;
                            minStart = prev;
                        }
                         count--;    // reduce count to end the while loop. tao: 此count--的作用完全就是想跳出while循環, 沒有別的用. 因為if (map.get(prevChar) > 0)成立即表示已經找到了正確的prev了(由下面知prev是while的循環變量), 故此時可以結束while循環了, 所以count--在if (map.get(prevChar) > 0)中. E3表示此count--是有俱體作含義的, 因為後面有prev++, 此即window中仍掉了S[prev]這個字符, 故計數(count)要減1.
                    }//end of if(map.get(prevChar) > 0)
                }//end of if(map.containsKey(prevChar))
                 prev++;//prev在while中開始定義賦值0的, 相當於是while的循環變量, 故在while最後結束前++
            } //end of while
        }//end of if (map.containsKey(currChar))
    }//end of for

    if (minLen == Integer.MAX_VALUE)    return "";
    else    return S.substring(minStart, minStart + minLen);
}
```

77. Combinations, Medium

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.
For example,
If *n* = 4 and *k* = 2, a solution is:
[

  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
Subscribe to see which companies asked this question

Key: item-res 模版遞歸法. 寫一個 void helper(int n, int k, int start, List<Integer> item, List<List<Integer>> res), 其作用為(即維護這樣一個不變量): 調用 helper(n, k, start, item, res)之前, item 中已經有了[1, start-1]中的一個可能的組合(為何不是所有可能的組合? 見注), 此組合中元素個數不一定是 k 個, 但少於 k 個. helper(n, k, start, item, res)之作用為: 從[start, n]中選出一些數加到 item 中, 當 item 中元素個數等於 k 時, 將 item 復製一份 放入 res 中. helper 中調用自己的方法是: i 遍歷[start, n], 將 i 加入到 item 中, 然後調用 helper(n,k,i+1,item,res), 最後將 item 最後一個元素刪掉 好保護現場.

本題代碼跟 46. Permutations 題相比, 基本上只有兩點不同, 一是少了個 used 數組, 二是 i 遍歷的范圍, 46 題是遍歷整個輸入數組, 而本題是遍歷[start, n]. 實際上, 對於本題, 第二點(i 遍歷 start 之後的)就是起了第一點(used 數組)的作用, 即 start 之前是 used 的.

[注] item 中為何要求已經有了[1, start-1]中的一個可能的組合, 原因跟 46. Permutations 中一樣: helper 是在甚麼時候被調用的? 一是在 combine 函數中, 此時調用的是 helper(n,k,1,new ArrayList<Integer>(), res), 顯然 1 之前沒有元素, 且 item 為空, 故此狀態(item 中已有[1, start-1]中的一個可能組合)自然滿足. 二是在 helper 中調用自己, 由於每次 helper 做了保護現場, 做此狀態也滿足.

History:

E1 的代碼很快寫好並在 leetcode 中一次通過, 用的是遞歸, 用了公式$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$. 直接在 combine(n, k)中遞歸調用 combine(n-1, k)和 combine(n-1, k)即可. 這樣做不好之處在於 要遞歸調用 combine 兩次.

E2 按 E1 的方法寫的, 很快寫好, 沒檢查, 一次通過. Code Ganker 的代碼確實不好理解(不是因為回溯), 或不容易想到.

E3 也是按 E1 的方法寫的, 代碼比 E1 的簡短. E3 想了半天 Key 中方法, 還是沒想出來, 原因跟 46. Permutations 題一樣, 我之前的 key 對 Code Ganker 代碼根本就沒理解清楚. E3 重寫了 key, 且默寫了 Code Ganker 的代碼.

William 的代碼不簡明, 還用了 sort (但注意 William 給此題標注的是 DFS).

Code Ganker:

这道题是 NP 问题的题目，时间复杂度没办法提高，用到的还是 N-Queens 中的方法：用一个循环递归处理子问题。实现的代码跟 Combination Sum 非常类似而且更加简练，因为不用考虑重复元素的情况，而且元素只要到了 k 个就可以返回一个结果。代码如下(tao: 已省略). NP 问题在 LeetCode 中出现的频率很高，例如 N-Queens，Sudoku Solver，Combination Sum，Permutations 等等。不过这类题目都是用一种方法而且也没有办法有时间上的提高，所以还是比较好掌握的。

Code Ganker 的代碼(已將 ArrayList 改為 List, 改後可通過):

```
public class Solution {
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(n<=0 || n<k)
        return res;
    helper(n,k,1,new ArrayList<Integer>(), res);
    return res;
}

private void helper(int n, int k, int start, List<Integer> item, List<List<Integer>> res)
{
    if(item.size()==k)
    {
        res.add(new ArrayList<Integer>(item));
        return;
    }
    for(int i=start;i<=n;i++)
    {
        item.add(i);
        helper(n,k,i+1,item,res);
        item.remove(item.size()-1);
    }
}
}
```

78. Subsets, Medium

http://wlcoding.blogspot.com/2015/03/subsets-i-ii.html?view=sidebar

Given a set of distinct integers, *nums*, return all possible subsets.
Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,
If ***nums*** = [1,2,3], a solution is:

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

Subscribe to see which companies asked this question

Key: item-res 模 版 遞 歸 法 . 寫 一 個 void helper(int[] nums, int start, List<Integer> item,

List<List<Integer>> res), 其作用為: 在 item 中加入 nums[start,...]中的元素, 每加一次都將 item 復製一份加入到 res 中, 然後調用 helper(nums, i + 1, item, res), 並刪除 item 最後一個元素 以保護現場.

History:
E1 沒做出來.

E2 沒按以上的 key 寫, 很快寫好, 本來可以一次通過的, 但沒初始化 res(本來初始化了的, 但被注釋掉了), 取消注釋後即通過. E2 沒用遞歸, 而是用的循環, 方法跟 17. Letter Combinations of a Phone Number 一樣. 但 E2 代碼沒有 William 的遞歸代碼簡潔. E2 的代碼不帖出來. E2 又重新了 key. E2 又默寫了一遍 William 的代碼.

E3 幾分鍾寫好, 改了一次後通過, 原因是最開始忘了在 res 中加一個空 List. E3 代碼跟 William 差不多. 為了跟前面的題一致, 以後用 E3 代碼.

感想:
E1: 連續兩天早起各種奔波, 睡眠不足, 現在是晚上一點, 又累又困又餓, 全身無力, 還要看這個遞歸回溯, 還與 for 循環糾纏在一起, 看了四個小時都沒看懂, 好難受(難字打了半天才打出來)
E2(准備 Google 時): 晚上一點算個毛, 老子昨晚六點睡, 十點多起來, 還沒說甚麼呢, 老子現在晚上好久沒三點前睡了.
E2(做本題時): 覺得本題很簡單啊. 不管是我自己的循環法, 還是 William 的遞歸法, 都很好理解啊.

Tao: William 的代碼比 Code Ganker 的短, 故用 William 的代碼, 但 Code Ganker 說的話也要看看.

William:

注意与 permutation 的区别：
permutation 可以取 i 之前的元素（所以每次要遍历 array 的所有元素，并跳过元素 i ），而 subset 只能取 i 之后的元素（所以只要遍历 array 中 start 到 end 的元素）。

DFS: Time ~ O(N!), Space ~ O(N)
•Must sort first (to generate non-descending order subset)
•将元素逐个加入到 list 中，每增加一个即得到一个 subset；取元素 i 加入 List，然后递归从 i + 1 开始，完成后从 List 中删除元素 i (tao: 此句話對理解遞歸回溯很有幫助)，再取元素 i + 1，直至元素取完为止。
•No need to check duplicate lists here, since it assumes no duplicates in the array.

Code Ganker:

求子集问题是经典的 NP 问题，复杂度上我们就无法强求了哈，肯定是非多项式量级的。一般来说这个问题有两种解法：递归和非递归。
我们先来说说递归解法，主要递推关系就是假设函数返回递归集合，现在加入一个新的数字，我们如何得到包含新数字的所有子集。其实就是在原有的集合中对每集合中的每个元素都加入新元素得到子集，然后放入原有集合中（原来的集合中的元素不用删除，因为他们也是合法子集）。而结束条件就是如果没有元素就返回空集（注意空集不是 null，而是没有元素的数组）就可以了。时间和空间都是取决于结果的数量，也就是 O(2^n)，代码如下(tao: 已省略). 其实非递归解法的思路和递归是一样的，只是没有回溯过程，也就是自无到有的一个个元素加进来，然后构造新的子集加入结果集中，代码如下(tao: 已省略). 这种 NP 问题算法上都没有什么大的提高，基本上就是考察对于递归和非递归解法的理解，都是和 N-Queens 一个套路，掌握之后就没什么难度哈。

William 代碼(不用):

```
//在我電腦上運行時, 別忘了也要給以下兩句加 static, 當然所有函數也要加 static(加了後就可以輸出正確答案). 注意 leetcode 中不能加這些 static, 否則說 wrong answer.

private List<List<Integer>> listSet = new ArrayList<List<Integer>>();
private List<Integer> list = new ArrayList<Integer>();

public List<List<Integer>> subsets(int[] S) {
    Arrays.sort(S);
    addUp(S, 0);
    return listSet;
}

private void addUp(int[] S, int start) {
    // if (!listSet.contains(list))         // No need to check duplicate lists!!
        listSet.add(new ArrayList<Integer>(list));
//上句即為將上一次遞歸得到的 list(已經执行了 list.add(S[i])的)拷到一個新的 list, 並將這個新的 list 加入 listSet.add 中. 此句第一次执行時即加入一個空集到 listSet 中
//若上句改為 listSet.add(list), 則得到的結果全是空集, 即[[], [], …]
//System.out.println(listSet);
    for (int i = start; i < S.length; i++) {
        list.add(S[i]);

//tao: 遞歸回溯之最好理解(1/4): 不要去想下一句的 addUp 是怎麼實現的, 不要去 track 每一輪循環的結果, 我以前就进入了這個思維誤區, 反而不好理解. 只要相信 addUp 的作用就可以了.
        addUp(S, i + 1);
//tao: 遞歸回溯之最好理解(2/4): addUp(S, i + 1)的作用為: 在 listSet 中加入一些子集(即 list), 這些子集為{S[i], S[i+1],... S[length-1]}的所有子集, 例如以下的輸出中, S[i]=1 時, 這些子集為 [1, 2], [1, 2, 3], [1, 3]]
        list.remove(list.size() - 1);
//tao: 遞歸回溯之最好理解(3/4): 以上一句中的 list 實際上是上上句 list.add[S[i]]中的 list, 因為 addUp()每次被执行之初就會自己弄(定義)一個新的 list, 這個新的 list 只在 addUp()函數內部有定義, 即從 addUP()跳出來之後,我們看到的是 addUp(S, i+1)之前的那個 list, 即 list.add(S[i])中的 list. list.remove(list.size() - 1)這句話的作用就是刪除 list.addS[i]中的 S[i], 然後进入下一輪 for 循環, 再加入下一個 S[i], 再調用 addUp, ....就這樣日下去.

//tao: 遞歸回溯之最好理解(4/4): 總結: 其實遞歸回溯也沒甚麼難的, 核心思想是要弄清楚被遞歸的函數是甚麼作用. 回溯的句子一般是出現在調用遞歸函數之後, 所以它起的作用是與前面調用的遞歸函數無關(因為遞歸函數已跳出, 遞歸函數的函數中 block 定義的變量現在沒定義了), 它起的作用是在遞歸函數之前的那些句子.

    }
}
```

E3 代碼(用它):

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(nums == null || nums.length == 0) return res;
    helper(nums, 0, new ArrayList<Integer>(), res);
    res.add(new ArrayList<Integer>());
    return res;
}

private void helper(int[] nums, int start, List<Integer> item, List<List<Integer>> res) {
```

```
        for(int i = start; i < nums.length; i++) {
            item.add(nums[i]);
            res.add(new ArrayList<Integer>(item));
            helper(nums, i + 1, item, res);
            item.remove(item.size() - 1);
        }
    }
```

79. Word Search, Medium.

http://wlcoding.blogspot.com/2015/03/word-search.html?view=sidebar

Given a 2D board and a word, find if the word exists in the grid.
The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.
For example,
Given **board** =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]
word = "ABCCED", -> returns true,
word = "SEE", -> returns true,
word = "ABCB", -> returns false.
Subscribe to see which companies asked this question

Key: 遞歸回溯. 難點在於處理重復使用過的字符. 寫一個 dfs(board, boolean[][] visited, i, j, index, word)函數, 其作用為判斷 word 以 index 開始(包括 index)的 substring 能否在 board 中(以 board[i][j]開始)中找到 (可以通過判斷對 board[i][j]前後左右是否跟 word 的字符 match 來實現). 這裡用一個 visited[][]數組來標記是否使用過. 所以另一個不變量是 '在調用 dfs(i, j)之前, 數組 visited 就已經記好了 board 上路徑 i,j(不含 i,j)之前的各點是否已被訪問過', 故 dfs 調用完後, 要將 visited 恢復為調用前的狀態. 多用 '不符合則 return false' 的方法. 在 dfs()中遞歸調用它自已. 在 exist 函數中, 對 board 上各點都調用 dfs.

History:
`E1` 的代碼其它的都還可以, 就是在處理重復使用過的字符上怎麼都弄不好.
`E2` 的代碼在我電腦上對各種 `test case` 都能給出正確結果, 但在 `OJ` 上, 大輸入時超時, 原因可能是我對 `visit` 的回溯更新處理得不好, 我是在 `exist` 函數中每次對 `visit` 整個數組重設為 `0` 的, 實際上在 `dfs` 中回溯更新 `visit[i][j]`就可以了, `E2` 的代碼其餘地方都跟 `William` 一樣.
E3 改了幾次後通過, 方法跟 William 一樣.

Tao: 以下用 William 的代碼. 不用 Code Ganker 的, 因為 Code Ganker 沒有 Word Search II. Code Ganker 說的也沒甚麼用有的, 所以沒有 copy.

William:
DFS: Time ~ O(M^2*N^2), Space ~ O(M*N)
Time: DFS ~ O(MN = number of edges), loop ~ O(MN)
•DFS letter 的上下左右。
•递归停止条件：

return ture：index 达到 word 的长度；
return false：越界、遇到之前的 letter、遇到不 match 的 letter
•DFS 返回后要将访问过的 letter unmark 掉。

```
public boolean exist(char[][] board, String word) {
    int m = board.length;
    int n = board[0].length;
    boolean[][] visited = new boolean[m][n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (dfs(board, visited, i, j, 0, word))
                return true;
        }
    }
    return false;
}

private boolean dfs(char[][] board, boolean[][] visited, int i, int j, int index,
String word) {
    if (index == word.length())   return true;
     if (i < 0 || i >= board.length || j < 0 || j >= board[0].length)     return
false;   // stop if cross the board
    if (visited[i][j])   return false;  // stop if visited
    if (board[i][j] != word.charAt(index))  return false;   // stop if not match
    visited[i][j] = true;   // marked as visited
    boolean match = dfs(board, visited, i - 1, j, index + 1, word)
                 || dfs(board, visited, i, j - 1, index + 1, word)
                 || dfs(board, visited, i + 1, j, index + 1, word)
                 || dfs(board, visited, i, j + 1, index + 1, word);
    visited[i][j] = false;//回溯, 保護現場
    return match;
}
```

80. Remove Duplicates from Sorted Array II, Medium

http://blog.csdn.net/linhuanmars/article/details/24343525

Follow up for "Remove Duplicates":
What if duplicates are allowed at most *twice*?
For example,
Given sorted array *nums* = [1,1,1,2,2,3],
Your function should return length = 5, with the first five elements of *nums* being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length.
Subscribe to see which companies asked this question

Key: 维护一个 counter，当 counter>=3 时，就直接跳过即可，否则说明元素出现次数没有超，继续放入结果数组，若遇到新元素则重置 counter. 本題解法與 26.Remove Duplicates from Sorted Array 不同點在於, 本題就是若 A[i] == A[i-1](當然還要 count>=3)則跳過, 其餘的都差不多. 不要以為本題按 A[i] == A[i-1]寫, 就複雜多少, 其實不複雜, 也沒有 corner case.

History:
E1 直接看的答案.
E2 通過.
E3 最開始按其它方式寫, 沒通過, 後來在本題 I 中 E3 代碼中加了幾行, 一次通過. E3 代碼帖在最後, 可以看看.

E3 的也是按 相等則跳過 的方法寫的, 但俱體寫法跟 Code Ganker 的不同, E3 沒用 count, 故沒有 Code Ganker 的好懂(但也不難懂). 以後還是用 Code Ganker 的代碼.

这道题跟 Remove Duplicates from Sorted Array 比较类似，区别只是这里元素可以重复出现至多两次，而不是一次。其实也比较简单，只需要维护一个 counter，当 counter 是 2 时(tao: 實際上是>=3)，就直接跳过即可，否则说明元素出现次数没有超，继续放入结果数组，若遇到新元素则重置 counter。总体算法只需要扫描一次数组，所以时间上是 O(n)，空间上只需要维护一个 index 和 counter，所以是 O(1)。代码如下. 这种简单数组操作的问题在电面中比较常见，既然简单，所以出手就要稳，不能出错，还是要保证无误哈。

Code Ganker 的代碼(用它):

```
public int removeDuplicates(int[] A) {
    if(A==null || A.length==0)
        return 0;
    int idx = 0;
    int count = 0; //此處可將 count 初始化為任意值(count=1, 2, 892 都可以通過), 因為後面每輪循環都會給 count 賦值為 1
    for(int i=0;i<A.length;i++)
    {
        if(i>0 && A[i]==A[i-1])
        {
            count++;
            if(count>=3)
                continue;
        }
        else
        {
            count = 1;
        }
        A[idx++]=A[i];
    }
    return idx;
}
```

E3 的代碼(可以看看, 但不用):

```
    public int removeDuplicates(int[] nums) {
        if(nums == null) return 0;
        int n = nums.length;
        if(n <= 1) return n;
        int pos = 0, l = 0, r = 0;

        while(r < n) {
            while(r < n && nums[l] == nums[r]) r++;
            nums[pos] = nums[l];
            pos++;

          //跟本題 I 的 E3 代碼相比, 就多加了以下的 if
            if(r - l > 1) {
                nums[pos] = nums[l];
                pos++;
```

```
            }

            l = r;
        }

        return pos;
    }
```

---

81. Search in Rotated Sorted Array II, Medium

http://blog.csdn.net/linhuanmars/article/details/20588511

Follow up for "Search in Rotated Sorted Array":
What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

Subscribe to see which companies asked this question

Key: 本題跟 33.Search in Rotated Sorted Array 差不多, 代碼就多一兩句. 即多了一個有重復元素相等時, 對邊緣的移動(見我 E3 的那兩個紅色 while). 看下面 Code Ganker 說的. 注意本題只要求判斷 target 在不在數組中, 沒要求返回其位置(本題的 I 要求返回位置). 由於我在本題的 I 中習慣先比較 A[m]和 A[r], 正好跟 Code Ganker 的說的相反, 所以可用以下例子來想:

{1,1,1,1,1,2,3} rotate 成以下兩種情況:
{2,3,1,1,1,1,1}
{1,1,1,1,2,3,1}

Extra: 對於 rotated sorted array 這類題目, 對於輸入數組 A 的某段 A[start, ...end], 記住以下結論:

若元素無重復, 若 A[start] < A[end], 則該段是 sorted (用手機信號那樣的柱狀圖稍想就知道(若有坎坎, 則必有 A[start]>A[end])), 否則若 A[start] > A[end], 則坎坎必在此段.
若元素有重復, 則用 移動邊界 的 trick.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過. E2 用的九章模版, 且跟本題的 I 一致, 所以以後用 E2 的代碼. 九章算法是直接用暴力一個一個試的, 因為二分法和暴力的時間復雜度都是 O(n). 但我覺得還是用二分法好, 因為二分法只有在最壞情況下才是 O(n), 還是比暴力法好.
E3 改了幾次後通過. E3 的代碼最好懂好想到的, 且跟本題 I 一致, 以後都用 E3 的代碼.

这道题是二分查找 Search Insert Position 的变体，思路在 Search in Rotated Sorted Array 中介绍过了，不了解的朋友可以先看看那道题哈。和 Search in Rotated Sorted Array 唯一的区别是这道题目中元素会有重复的情况出现。不过正是因为这个条件的出现，出现了比较复杂的 case，甚至影响到了算法的时间复杂度。原来我们是依靠中间和边缘元素的大小关系，来判断哪一半是不受 rotate 影响，仍然有序的。而现在因为重复的出现，如果我们遇到中间和边缘相等的情况，我们就丢失了哪边有序的信息，因为哪边都有可能是有序的结果。假设原数组是{1,2,3,3,3,3,3}，那么旋转之后有可能是{3,3,3,3,3,1,2}，或者{3,1,2,3,3,3,3}，这样

的我们判断左边缘和中心的时候都是 3，如果我们要寻找 1 或者 2，我们并不知道应该跳向哪一半。解决的办法只能是对边缘移动一步，直到边缘和中间不在相等或者相遇，这就导致了会有不能切去一半的可能。所以最坏情况（比如全部都是一个元素，或者只有一个元素不同于其他元素，而他就在最后一个）就会出现每次移动一步，总共是 n 步，算法的时间复杂度变成 O(n)。代码如下.
以上方法和 Search in Rotated Sorted Array 是一样的，只是添加了中间和边缘相等时，边缘移动一步，但正是这一步导致算法的复杂度由 O(logn)变成了 O(n)。个人觉得在面试中算法复杂度还是很重要的考察点，因为涉及到对算法的理解，大家还是要尽量多考虑哈。

Code Ganker 的代碼(不用):

```
public boolean search(int[] A, int target) {
    if(A==null || A.length==0)
        return false;
    int l = 0;
    int r = A.length-1;
    while(l<=r)
    {
        int m = (l+r)/2;
        if(A[m]==target)
            return true;
        if(A[m]>A[l])
        {
            if(A[m]>target && A[l]<=target)
            {
                r = m-1;
            }
            else
            {
                l = m+1;
            }
        }
        else if(A[m]<A[l])
        {
            if(A[m]<target && A[r]>=target)
            {
                l = m+1;
            }
            else
            {
                r = m-1;
            }
        }
        else
        {
            l++;
        }
    }
    return false;
}
```

E2 的代碼(不用):

```
public boolean search(int[] nums, int target) {
    if(nums == null || nums.length == 0) return false;
    int l = 0, r = nums.length - 1;

    while(l + 1 < r) {
        int m = (l + r) / 2;

        if(nums[m] == target) return true;

        if(nums[m] < nums[r]) {
            if(nums[m] < target && target <= nums[r]) l = m;
            else r = m;
        } else if(nums[m] > nums[r]) {
            if(nums[l] <= target && target < nums[m]) r = m;
            else l = m;
        } else {
            while(r != m && nums[r] == nums[m]) r--;
        }
    }

    if(nums[l] == target) return true;
    if(nums[r] == target) return true;

    return false;
}
```

E3 的代碼(用它, 跟 I 相比, 就多了下面兩個紅色的 while):

```
public boolean search(int[] nums, int target) {
    if(nums == null || nums.length == 0) return false;
    int n = nums.length;
    int l = 0, r = n - 1;

    if(target == nums[0]) return true;
    else if(target == nums[n - 1]) return true;

    while(l + 1 < r) {
        while(l <= n - 2 && nums[l + 1] == nums[l]) l++;
        while(r >= 1 && nums[r - 1] == nums[r]) r--;

        int m = (l + r) / 2;
        if(target == nums[m]) return true;

        if(nums[l] > nums[m]) {
            if(nums[m] < target && target < nums[r]) l = m;
            else r = m;
        } else {
            if(nums[l] < target && target < nums[m]) r = m;
            else l = m;
```

```
            }
        }

        return false;
    }
```

九章算法的代碼:

```
// it ends up the same as sequential search
// We used linear search for this question just to indicate that the
// time complexity of this question is O(n) regardless of binary search is applied or not.
public class Solution {
    public boolean search(int[] A, int target) {
        for (int i = 0; i < A.length; i ++) {
            if (A[i] == target) {
                return true;
            }
        }
        return false;
    }
}
```

82. Remove Duplicates from Sorted List II, Medium

http://blog.csdn.net/linhuanmars/article/details/24389429

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.
For example,
Given 1->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.
Subscribe to see which companies asked this question

Key: 找到重復的那堆元素的首尾, 然後把它們全刪掉. 還是用 pre 和 cur 兩個指針. 若 cur 和它下一個不相等, 則 pre=cur. 否則就 cur 就一直往後跳, 直到跳到 cur 和它下一個不相等, 然後把這堆相等的都刪掉.

History:
E1 沒做出來.
E2 改了幾次後通過, 方法跟 Code Ganker 差不多, 代碼有些不同.
E3 一次通過, 方法跟 key 中一樣. E3 最開始把題目意思理解複雜了, 以為對於 1->2->3->3->3->2->4 的情況, 將 3 全刪了後, 對剩下的 1->2->2->4 中的 2 也要全刪掉, 所以多花出了時間想這個怎麼弄, 後來一想, 題意顯然不要求刪 2. 理解對了後, 即一次通過(也花了點時間寫).

Tao: 我的代碼沒處理好前驅指針. 用 Code Ganker 的代碼.

这道题跟 Remove Duplicates from Sorted List 比较类似，只是这里要把出现重复的元素全部删除。其实道理还是一样，只是现在要把前驱指针指向上一个不重复的元素中，如果找到不重复元素，则把前驱指针知道该

元素，否则删除此元素。算法只需要一遍扫描，时间复杂度是 O(n)，空间只需要几个辅助指针，是 O(1)。代码如下. 可以看到，上述代码中我们创建了一个辅助的头指针，是为了修改链表头的方便。前面介绍过，一般会改到链表头的题目就会需要一个辅助指针，是比较常见的技巧。

1 楼 阳光明媚 2015 2015-03-28 16:02 发表

哎，我没意识到单向链表，也可以实现前驱指针的概念。。。。

```
public ListNode deleteDuplicates(ListNode head) {
    if(head == null)
        return head;
    ListNode helper = new ListNode(0);
    helper.next = head;
    ListNode pre = helper;
    ListNode cur = head;
    while(cur!=null)
    {
        //以下 while 結束後, cur 指向的是重復元素中的最後一個.
        while(cur.next!=null && pre.next.val==cur.next.val) //為何不像 83. Remove Duplicates from Sorted List 中那樣寫成 pre.val==cur.val? 因為此處要讓 pre 指向上一个不重复的元素(Code Ganker 前面說了)
        {
            cur = cur.next;
        }
        if(pre.next==cur) //即沒發現重復的元素
        {
            pre = cur; //原代碼中為 pre = pre.next; 為了保持與 83. Remove Duplicates from Sorted List 的代碼一致, 我將其改成了 pre=cur
        }
        else
        {
            pre.next = cur.next;
        }
        cur = cur.next;
    }

    return helper.next;
```

}

83. Remove Duplicates from Sorted List, Easy

http://blog.csdn.net/linhuanmars/article/details/24354291

Given a sorted linked list, delete all duplicates such that each element appear only *once*.
For example,
Given 1->1->2, return 1->2.
Given 1->1->2->3->3, return 1->2->3.
Subscribe to see which companies asked this question

Key: 维护两个指针，一个(pre)指向当前不重复的最后一个元素，一个(cur)进行依次扫描:
if(cur.val == pre.val) pre.next = cur.next; //若 cur 與 pre 相等, 則刪除 cur
else pre = cur; //否則更新 pre 為 cur

History:
E1 很快寫好, 一次通過.
E2 幾分鍾寫好, 一次通過, 代碼跟 Code Ganker 基本上一模一樣.
E3 幾分鍾寫好, 一次通過, 方法跟 key 中一樣.

Tao: 我的代碼也是很快寫好, 並在 leetcode 中一次通過. 但我後來做 82. Remove Duplicates from Sorted List II 時, 沒處理好前驅指針, 用了 Code Ganker 的代碼, 所以本題也用 Code Ganker 的代碼.
Code Ganker:

这是一道比较简单的链表操作的题目，要求是删去有序链表中重复的元素。方法比较清晰，维护两个指针，一个指向当前不重复的最后一个元素，一个进行依次扫描，遇到不重复的则更新第一个指针，继续扫描，否则就把前面指针指向当前元素的下一个（即把当前元素从链表中删除）。时间上只需要一次扫描，所以是 O(n)，空间上两个额外指针，是 O(1)。代码如下. 链表操作在 LeetCode 中占有一定的比例，不过面试中出现的频率并不是特别高，不过基本的操作还是要熟练的。这道题目还可以求于另一个数据结构数组中，也比较简单，有兴趣可以看看 Remove Duplicates from Sorted Array。

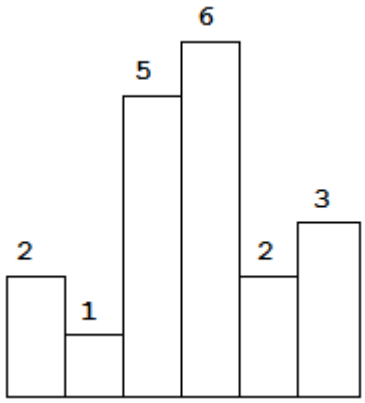
```
public ListNode deleteDuplicates(ListNode head) {
    if(head == null)
        return head;
    ListNode pre = head;
    ListNode cur = head.next;
    while(cur!=null)
    {
        if(cur.val == pre.val)
            pre.next = cur.next; //若 cur 與 pre 相等, 則刪除 cur
//若执行上句前為 1->2(pre)->2(next)->3->4,
//則执行上句後為 1->2(pre)->3->4, 另一條鏈為 2(next)->3->4
        else
            pre = cur; //否則更新 pre 為 cur
```

```
        cur = cur.next;
    }
    return head;
}
```

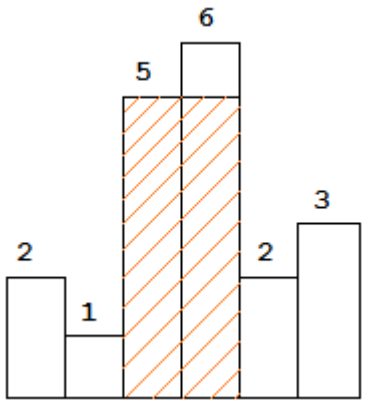
84. Largest Rectangle in Histogram, Hard

http://blog.csdn.net/doc_sgl/article/details/11805519

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].



The largest rectangle is shown in the shaded area, which has area = 10 unit.
For example,
Given heights = [2,1,5,6,2,3],
return 10.
Subscribe to see which companies asked this question

Key: 熱列慶祝本題於 2016 年 8 月 11 日被我(E2.5)評為 Leetcode 第一難題! 每次看答案解釋都要看半天. 尼瑪 85 題還要用本題.

William 說的不太容易看懂, 如果看不懂, 則看 hackersun007 的解釋. 以下是我總結出的 key(盡量看懂): 將 index 放入棧中, 按高度從小到大的順序放. 若還到比棧頂元素高的, 當然入棧. 若還到比棧頂元素還矮的(設這個 比棧頂元素還矮的元素 的 index 為 i), 則出棧(出的不是 i, 而是棧頂的那個高子 index(設此 index 為 t)), 然後算面積: height[t] * (i - stack.top() - 1), 此面積實際上即為以 height[t]為高度, 寬度為 't 的 left edge 到 i 的 left edge', 的這樣一個面積. 然後繼續, 取這些面積中最大的. 若前面出棧後, 棧為空, 則面積按 height[t] * i 算, 因為當前元素 i 左邊所有的元素都比元素 i 大(也都比 h[t]大), 所以這時的面積可以是從最左邊開始, 所以是乘以 i. 注意在幹所有事之前, 要在 heights 最後加一個 0, 但由於 heights 為數組, 不好加東西, 克服的方法是: 要用到 heights[heights.length]時, 將 heights[heights.length]換為 0 即可.

History:
E1 直接看的答案.
E2 也直接看的答案.
E3 沒做出來, 原因是 E3 沒有在 heights 最後加 0, 且 E3 以為只要棧為空, 就按 heights[t]*i 來算面積, 實際上是當出棧後為空時, 才這樣算, 若沒出棧時 棧就為空, 則只 push(i). E3 在 key 中加入了這兩點.

Tao: 以下來自 hackersun007, 他的含圖的解釋說得很好懂. 但他的代碼是 C++的. William 的 Java 代碼跟他的一模一樣(Code Ganker 的太長不用). 下面先貼上 William 的 Java 版本. 將 hackersun007 的代碼看懂了後, William 的代碼就很好懂了, 最終要以 William 的為准.

William:

这道题非常巧妙，只用扫一遍 array。
方法：对于某一个 bar i，将其左边所有比起低的 bar 入栈，对其左边所有比其高的 bar 计算能形成的最大面积。假设 bar i 左边有一个比其高的 bar j，计算 bar j 能形成的矩形面积时，高度取 bar j 的 height[j]；宽度取 i - stack.peek() - 1（注意这里用 peek() 不用 pop()，不是 bar j 的 index，而是 bar j 左边第一个低于其的 bar 的 index），即在 bar j 的左右两边分别找到第一个低于其的 bar，这样形成 bar j 可以取到的最大矩形。
实现用一个 Stack（栈内的元素只呈升序），每次比较栈顶与当前元素。

•如果栈顶元素 <= 当前元素，则入栈；
•否则合并现有栈，依次往回计算面积（不包括当前元素），直至栈顶元素小于当前元素。

另外要注意扫到数组结尾时计算面积的边界情况，解决方法是在 array 的结尾加一个 0 元素，比如当只有一个元素 height = {1}，Area = 1。

William 代碼(用它):

```
public int largestRectangleArea(int[] height) {

    Stack<Integer> stack = new Stack<Integer>();
    int maxArea = 0;
    int i = 0;
    while(i <= height.length){
        if(stack.isEmpty() || height[stack.peek()] <= (i == height.length ?
0 : height[i])){ // add 0 at end //人为添加的 dummy 元素 0 (Java 方法), 後面會講.
            stack.push(i++);
        }else {
```

```
                int t = stack.pop();
                 maxArea = Math.max(maxArea, height[t] * (stack.isEmpty() ? i :
i - stack.peek() - 1)); //注意不要寫成了 i-t-1
            }
        }
        return maxArea;
}
```

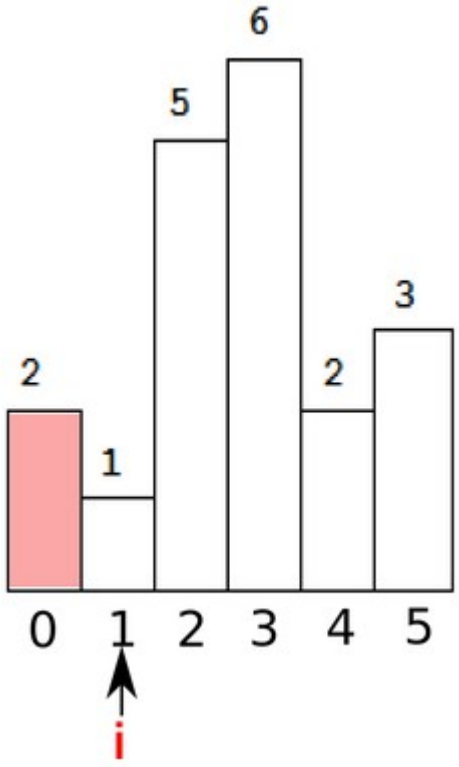
hackersun007 的 C++代碼(不用, 但要看他的解釋)

```
class Solution {
public:
    int Max(int a, int b){return a > b ? a : b;}
    int largestRectangleArea(vector<int> &height) {
            height.push_back(0); //人为添加的 dummy 元素 0 (C++方法), 後面會講.
        stack<int> stk;
        int i = 0;
        int maxArea = 0;
        while(i < height.size()){
            if(stk.empty() || height[stk.top()] <= height[i]){ //寫成 height[i] >= height[stk.top()]更好懂
                stk.push(i++);
            }else {
                int t = stk.top();
                stk.pop();
                maxArea = Max(maxArea, height[t] * (stk.empty() ? i : i - stk.top() - 1));
            }
        }
        return maxArea;
    }
};
```

hackersun007 的解釋:

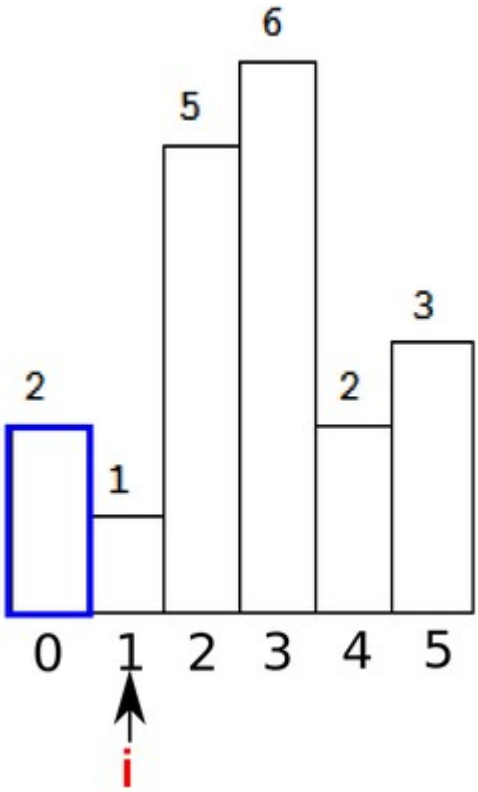
就用题目中的[2,1,5,6,2,3]来解释一下这段代码吧。

首先，如果栈是空的，那么索引 i 入栈。那么第一个 i=0 就进去吧。注意栈内保存的是索引，不是高度。然后 i++。

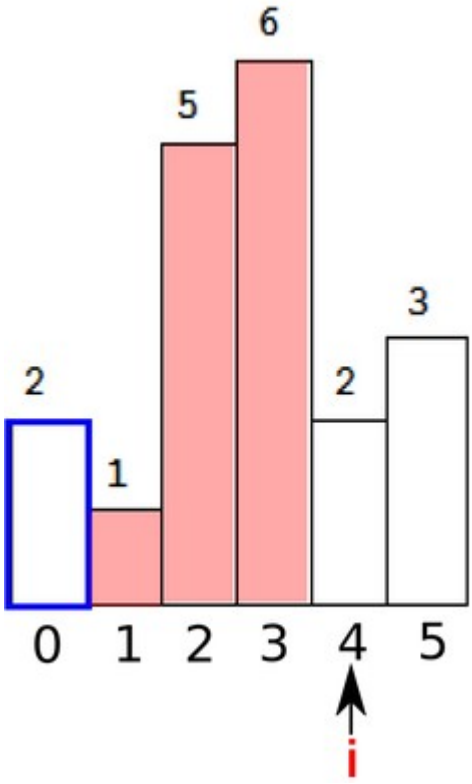


然后继续，当 i=1 的时候，发现 h[i]小于了栈内的元素，于是出栈。（由此可以想到，哦，看来 stack 里面只存放 height 单调递增的索引）

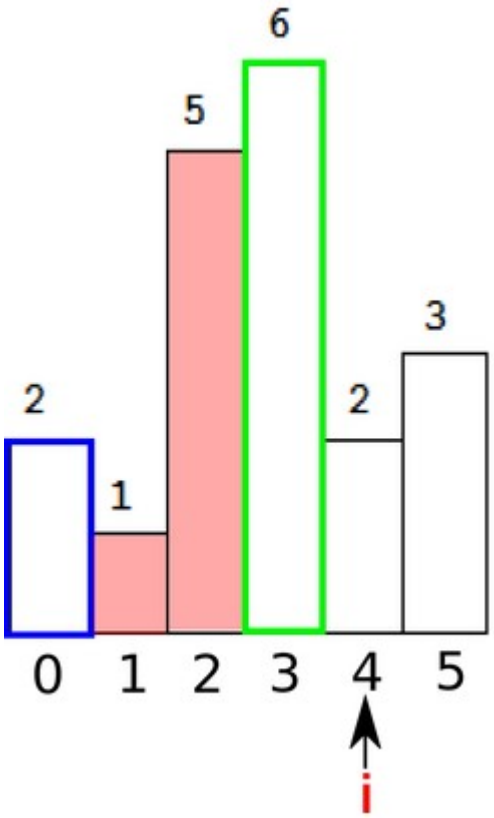
这时候 stack 为空，所以面积的计算是 h[t] * i。t 是刚刚弹出的 stack 顶元素。也就是蓝色部分的面积。



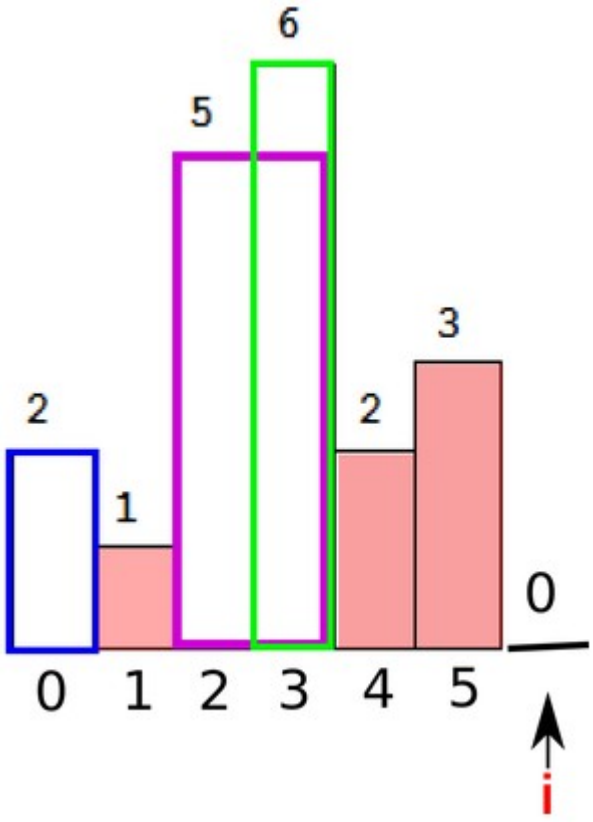
继续。这时候 stack 为空了，继续入栈。注意到只要是连续递增的序列，我们都要 keep pushing，直到我们遇到了 i=4，h[i]=2 小于了栈顶的元素。



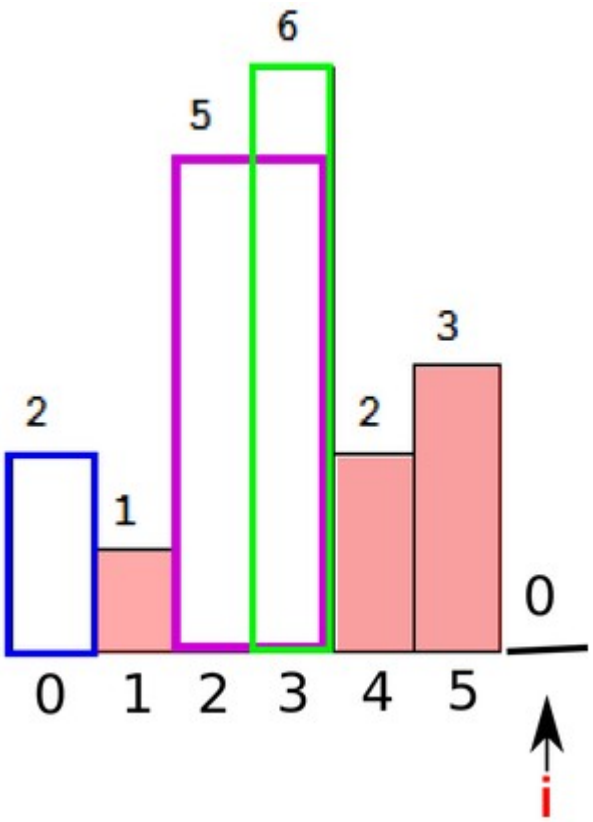
这时候开始计算矩形面积。首先弹出栈顶元素，t=3 (t 為 stack.top())。即下图绿色部分(並且還算了綠色部分的面積, 注意按代碼中的 i - stk.top() - 1 算時, stk.top()=2, 而不是 3)。



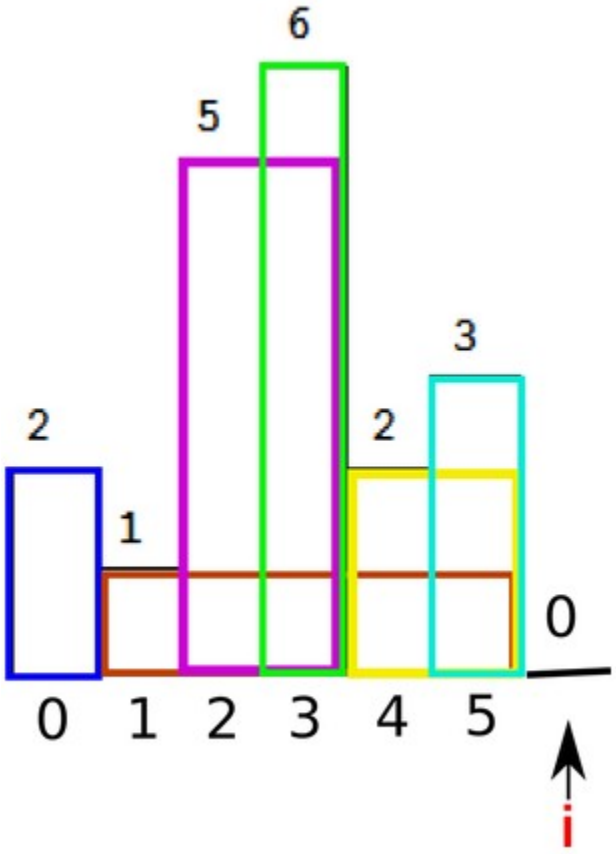
接下来注意到栈顶的（索引指向的，← tao 覺得此括號裡的內容看不懂, 還是別看好）元素还是大于当前 i 指向的元素，于是出栈，并继续计算面积，粉色部分。



最后，栈顶的（索引指向的，← tao 覺得此括號裡的內容看不懂，還是別看好）元素小于了当前 i 指向的元素，循环继续，入栈并推动 i 前进(tao: 下圖中粉色的部分都是棧中的)。直到我们再次遇到下降的元素，也就是我们最后人为添加的 dummy 元素 0.



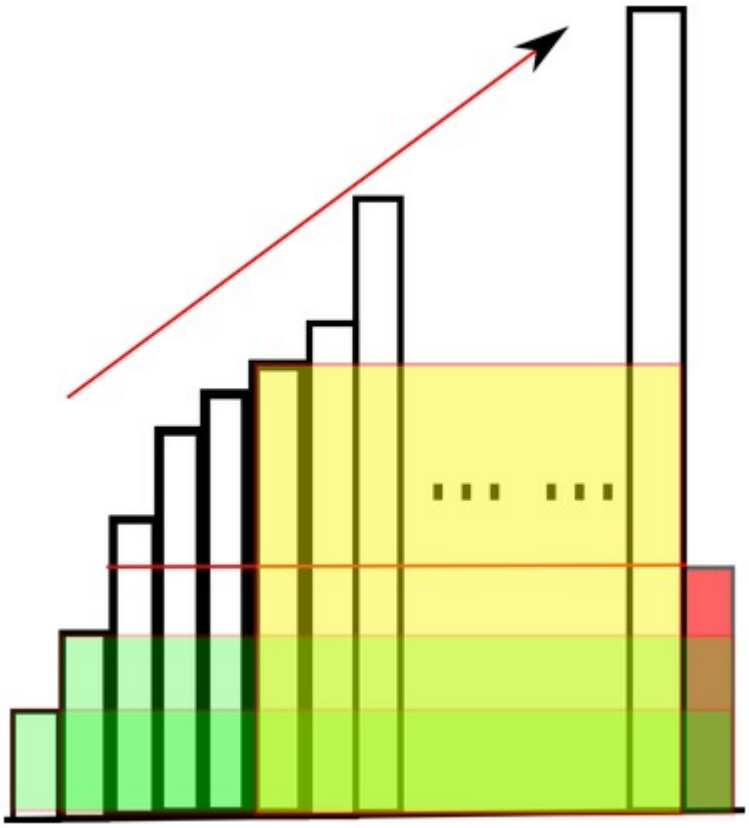
同理，我们计算栈内的面积。由于当前 i 是最小元素，所以所有的栈内元素都要被弹出并参与面积计算。



注意我们在计算面积的时候已经更新过了maxArea。

总结下，我们可以看到，stack 中总是保持递增的元素的索引，然后当遇到较小的元素后，依次出栈并计算栈中 bar 能围成的面积，直到栈中元素小于当前元素。

可以这样理解这个算法，看下图。

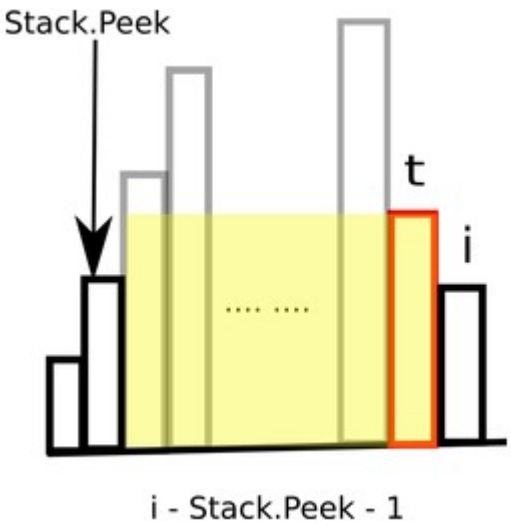


例如我们遇到最后遇到一个递减的 bar（红色）。高度位于红线上方的（也就是算法中栈里面大于最右 bar 的）元素，他们是不可能和最右边的较小高度 bar 围成一个比大于在弹栈过程中的矩形面积了（黄色面积），因为红色的 bar 对他们来说是一个短板，和红色 bar 能围成的最大面积也就是红色的高度乘以这些“上流社会”所跨越的索引范围。但是“上流社会”的高度个个都比红色 bar 大，他们完全只计算彼此之间围成的面积就远远大于和红色 bar 围成的任意面积了。所以红色 bar 是不可能参与“上流社会”的 bar 的围城的。因为虽然长度不占优势，但是团结的力量是无穷的。它还可以参与“比较远的”比它还要屌丝的 bar 的围城(tao: 此時那些上流社會中的都已經移民出國(棧)了, 棧內只剩下紅色 bar 和比他還要屌絲的 bar. 原來出棧的意義就是高富帥出國, 留屌絲在國內竞爭啊!)。他们的面积是有可能超过上流社会的面积的，因为距离啊！所以弹栈到比红色 bar 小就停止了。

另外一个细节需要注意的是，弹栈过程中面积的计算。

h[t] * (stack.empty() ? i : i - stack.top() - 1)

h[t]是刚刚弹出的栈顶端元素。此时的面积计算是 h[t]和前面的“上流社会”能围成的最大面积。这时候要注意哦，栈内索引指向的元素都是比 h[t]小的，如果 h[t]是目前最小的，那么栈内就是空哦(tao: 稍想即知, 當棧為空時, 當前元素 i 左邊所有的元素都比元素 i 大(也都比 h[t]大), 所以這時的面積可以是從最左邊開始, 所以是乘以 i)。(本句跟前面內容無關) 而在目前栈顶元素和 h[t]之间（不包括 h[t]和栈顶元素），都是大于他们两者的。如下图所示：



那 h[t]无疑就是 Stack.top()和 t 之间那些上流社会的短板啦，而它们的跨越就是 i - Stack.top() - 1。

所以说，这个弹栈的过程也是维持程序不变量的方法啊：栈内元素一定是要比当前 i 指向的元素小的。

評論:

倒数第三个图的黄色和红色部分都画错了，红色应该是所有的底边，面积是 6，而黄色面积应该是 8。程序没问题但是图错了。
倒数第三个图的黄色矩形表错了，左边应该标到 index 的 1 与 2 之间

他们是不可能和最右边的较小高度 bar 围成一个比大于在弹栈过程中的矩形面积了（黄色面积）？？
那假如是{1,6,7,5}呢？6*2<5*3 and 7*1<5*2

回复 cow__sky：赞成，博主在这点表述上是存在缺陷的。

85. Maximal Rectangle, Hard

http://blog.csdn.net/doc_sgl/article/details/11832965

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.
Subscribe to see which companies asked this question

Key: DP. 本題題目的意思為, 若有以下輸入數組, 則以下紅色區域即為 larges rectangle containing all 1, 故返回紅色區域之面積.

0	0	1	0
0	0	0	1
0	1	1	1
0	0	1	1

雖然本題後面的網上解說很長, 其實本題很簡單(如果直接調用 84 題的函數), 不要浪費時間去看後面的網上解說(表述不清), 直接看我的 key 即可. 本題解法之核心在於 將本題轉化成 84. Largest Rectangle in Histogram 題. 轉化的方法為: 本題的輸入數組 matrix 的每一行可導出一個 histogram. 例如上圖的 matrix 可以這樣弄:

matrix[0]可導出 0 0 1 0 這樣一個 histogram, 即從左到右有四個柱, 它們高度分別為 0, 0, 1, 0.

matrix[1]可導出 0 0 0 1 這樣一個 histogram,
matrix[2]可導出 0 1 1 2 這樣一個 histogram, 下面馬上解釋
matrix[3]可導出 0 0 2 3 這樣一個 histogram,
這些 histogram 是怎樣導出的呢? 是這樣導出的: matrix[i]形成的 histogram 的第 j 個柱之高度為 'matrix 第 j 列從第 0 行到第 i 行的連續 1 的個數'. 這裡的'連續 1'的意思是 這條連續的 1 形成的線段的下端要在 matrix 的第 i 行上(即這條連續的 1 不能懸空).

然後將 matrix 每一行的 histogram(如最後一行的{0, 0, 2, 3})當作 height[]數組, 作為 84. Largest Rectangle in Histogram 題中的 largestRectangleArea(int[] height)函數的輸入數組, 得到該行形成的最大面積, 最後在所用面積中找出最大的, 就是本題之答案.

實際中, 可用 d[i][j]來表示 matrix[i]形成的 histogram 的第 j 個柱之高度, 遞推式為:
若 matrix[i][j] = 1, 則 d[i][j] = d[i-1][j] + 1
若 matrix[i][j] = 0, 則 d[i][j] = 0

E3 表示 d[i][j]可以寫成一維數組(像 1d-DP 那樣), 且不難寫, 即用一個一維數組 heights[j]來表示第 i 行的 heights(此處 heights 的意思跟 84 題一樣).

History:
E1 直接看的答案.

E2 很快寫好, 本來可以一次通過的, 但寫遞推式時出了點小錯誤(E1 的 key 沒有遞推式, 以上的 key 都是 E2 寫的), 改後即通過. 我 E2 的代碼跟答案的步驟一樣, 但更簡明.

E3 幾分鍾寫好(largestRectangleArea 函數是抄的 84 題的 E3 代碼), 但改了一次才通過, 原因是 E3 開頭部分最開始是這樣寫的:

```
if(matrix == null) return 0;
int n = matrix.length, m = matrix[0].length;
if(n == 0 || m == 0) return 0;
```

這樣寫的錯誤之處在於, 若輸入數組為[], 即 matrix.length =0, 此時 matrix[0].length 根本就不存在, 故 m = matrix[0].length 一句會報錯. 所以以後二維數組都不要像上面那樣寫, 而應該像下面這樣寫:

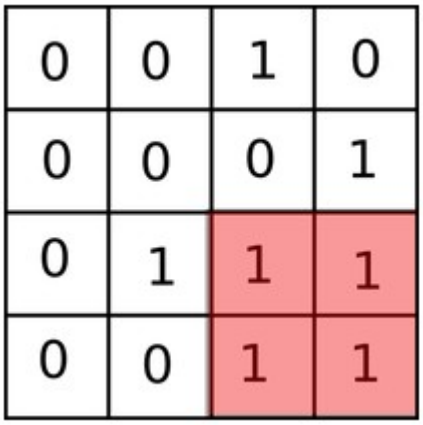
```
if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
int n = matrix.length, m = matrix[0].length;
```

另外, E3 代碼中的 d[][]是用的一個一維數組(heights[]), 故空間上比答案和 E2 都好, 故以後都用 E3 代碼.

Tao: 以下用 hackersun007 的算法. 他的原代碼是用 C++寫的, 我將其改寫為了 Java (leetcode 已通過), 以下用我改寫後的代碼.

hackersun007:

例如矩阵是下图这样的，结果应该是图中红色区域的面积：



一般人拿到这个题目，除非做过类似的，很难一眼找出一个方法来，更别说找一个比较优化的方法了。

首先一个难点就是，你怎么判断某个区域(tao: 含 1 的區域)就是一个矩形呢？

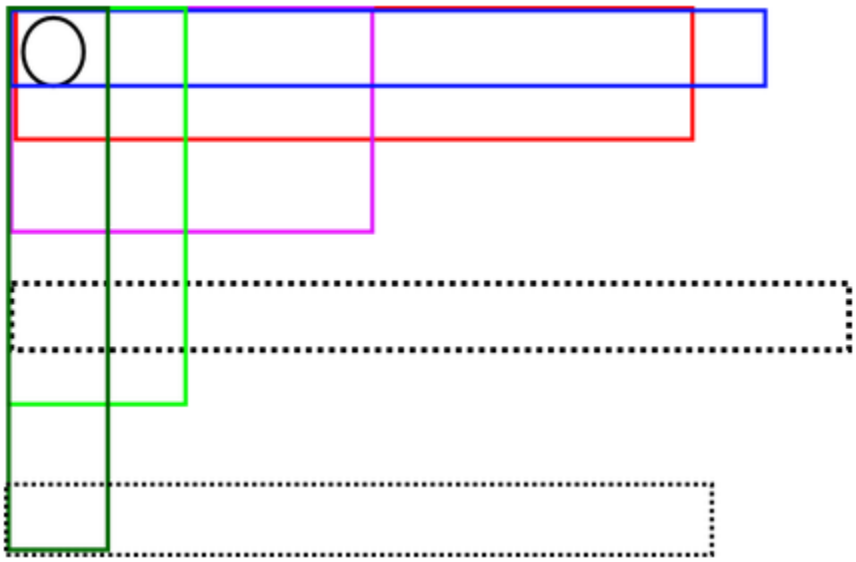
其次，以何种方式来遍历这个 2D 的 matrix 呢？

一般来说，对这种“棋盘式”的题目，像什么 Queen 啦，象棋啦，数独啦，如果没有比较明显的遍历方式，可以采用一行一行地遍历。

然后，当遍历到(i, j)的时候，该做什么样的事情呢？想想，嗯，那我可不可以简单看看，以(i,j)为矩形左上角，能不能形成一个矩形(tao: 此矩形都是由 1 組成)，能不能形成多个矩形？那形成的矩形中，我们能不能找一个最大的呢？

首先，如果(i, j)是 0，那肯定没法是矩形了。

如果是 1，那么我们怎么找以它为左上角的矩形(tao: 此矩形都是由 1 組成)呢？呼唤画面感！



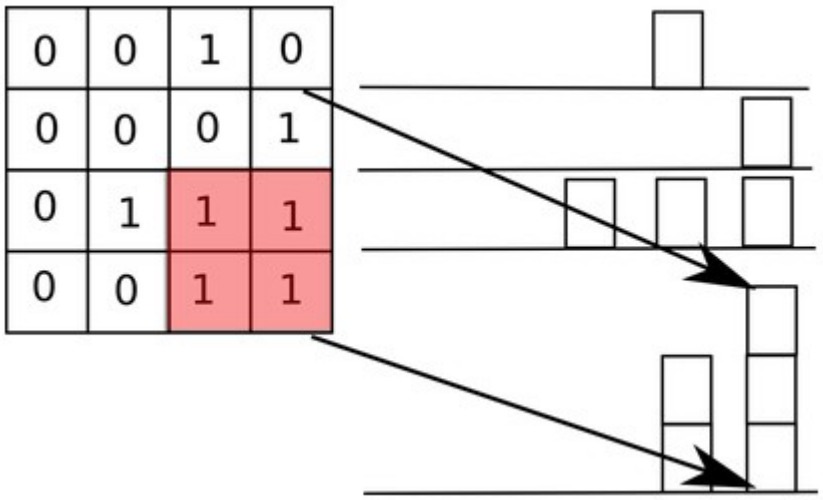
图中圈圈表示左上角的 1，那么矩形的可能性是。。。太多啦，怎么数呢？

我们可以试探地从左上角的 1 所在的列开始，往下数数，然后呢，比如在第一行，例如是蓝色的那个矩形，我们看看在列

上，它延伸了多远，这个面积是可以算出来的。然后继续，第二行，例如是那个红色的矩形，再看它延伸到多远，哦，我们知道，比第一行近一些，我们也可以用当前离第一行的行数，乘以延伸的距离，得到当前行表示的矩形面积。但是到了第一个虚线的地方(tao: 此虛線框中全為 1, 注意此時已經不限於找以圖中圈圈為左上角的矩形, 此時是找任意最大矩形)，它远远超过了上面的其他所有行延伸的距离了，注意它的上方都是空心(tao: 即有些元素是 0)的哦，所以，我们遇到这种情况，计算当前行和左上角(tao: 為甚麼是左上角? 因為我們是從那裡開始看起走的, 比如輸入的矩陣的[0][0]元素就是圖中的左上角)1 围成的面积的时候，只能取 所有前面最小的延伸距离 乘以 当前离第一行的行数。其实，这对所有情况都是这样的，是吧？于是，我们不是就有方法遍历这些所有的矩形了嘛。

但是，当我们在数这个矩形的时候越来越像 leetcode_question_85 Largest Rectangle in Histogram 这道题了，不是吗? (tao: 此處的 histogram 是横著放的). 我们讨论了柱状图的最大矩形面积，那可以 O(n)的，学以致用呀！
btw，leetcode 的这两题也是挨一块儿的，用心良苦。。。。

(tao: 以下開始, 又研究豎著的 histogram 了. 下圖中的兩個箭頭的指向沒有特別的意義, 可以不管那兩個箭頭.)



上面的矩阵就成这样了(tao: 以下即 dp 矩陣)：

0	0	1	0
0	0	0	1
0	1	1	2
0	0	2	3

每一行都是一次柱状图的最大矩形面积了。dp[i][j]就是当前的第 j 列(時第 0 行)到第 i 行连续 1 的个数。

以下代碼不用:

```
public static int largestRectangleArea(int[] height) {
(將 84. Largest Rectangle in Histogram 中的代碼抄過來)
}

 public static int maximalRectangle(char[][] matrix) {
  if(matrix == null || matrix.length == 0 || matrix[0].length == 0)
   return 0;

  int m = matrix.length;
  int n = matrix[0].length;

  int[][] dp = new int[m][n];

  for(int j = 0; j < n; j++) {
   if(matrix[0][j] == '1')
    dp[0][j] = 1;
  }

  for(int j = 0; j < n; j++) {
   for(int i = 1; i < m; i++) {
    if(matrix[i][j] == '1')
     dp[i][j] = dp[i-1][j] + 1;
   }
  }

  int maxRect = 0;

  for(int i = 0; i < m; i++) {
   int temp = largestRectangleArea(dp[i]);
   if(temp > maxRect)
    maxRect = temp;
  }

  return maxRect;
 }
```

leetcode 的讨论组还给出了一个比较难理解的方法。。。

我 E2 的代碼(不用):

```
    public int largestRectangleArea(int[] height) {
        (將 84. Largest Rectangle in Histogram 中的代碼抄過來)
    }

    public int maximalRectangle(char[][] matrix) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
        int m = matrix.length, n = matrix[0].length;
```

```
        int[][] d = new int[m][n];

        for(int j = 0; j < n; j++) {
            d[0][j] = (matrix[0][j] == '1' ? 1 : 0);
        }

        for(int i = 1; i < m; i++) {
            for(int j = 0; j < n; j++) {
                d[i][j] = (matrix[i][j] == '1' ? d[i - 1][j] + 1 : 0);
            }
        }

        int max = 0;

        for(int i = 0; i < m; i++) {
            max = Math.max(max, largestRectangleArea(d[i]));
        }

        return max;
    }
```

E3 代碼(用它):

```
    public int maximalRectangle(char[][] matrix) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
        int n = matrix.length, m = matrix[0].length;
        int[] heights = new int[m];
        int max = 0;

        for(int i = 0; i < n; i++) {
            for(int j = 0; j < m; j++) {
                if(matrix[i][j] == '0') heights[j] = 0;
                else heights[j] += 1;
            }

            max = Math.max(max, largestRectangleArea(heights));
        }

        return max;
    }

    public int largestRectangleArea(int[] heights) {
            (將 84. Largest Rectangle in Histogram 中的代碼抄過來)
    }
```

86. Partition List, Medium

http://blog.csdn.net/linhuanmars/article/details/24446871

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.
You should preserve the original relative order of the nodes in each of the two partitions.
For example,
Given 1->4->3->2->5->2 and *x* = 3,
return 1->2->2->4->3->5.
Subscribe to see which companies asked this question

Key: 用 pre 和 cur 兩個指針. pre 先向右掃, 使得 pre 指向那一片小於 x(如 3)的元素中的最後一個,如 1->2->1->4->3->2->5->2 中的 1. 然後使 cur 指向第一個大於或等於 x 的元素, 如 1->2->1->4->3->2->5->2 中的 4. 此時有 1->2->1(pre)->4(cur)->3->2->5->2. 然後寫一個循環, 每輸循環中, 考查 cur.next 是否小於 x, 若小於, 則將 cur.next 移到 pre 之後, 並更新 pre. 若不小於, 則 cur 往右走.

History:
E1 直接看的答案.
E2 通過, 但被 corner case 檔了好多次(尤其是開頭就是幾個比 k 小的 node 之情情況), 代碼雖不長, 但邏輯有點繞(用 E3 的代碼就不覺得繞).
E3 最開始想錯方法了, 導致死循環, 後來想對了方法(即 key 中的, key 是 E3 寫的, 跟 code ganker 方法一樣), 寫好後改了一次(沒考慮到所有元素都小於 x)就通過了. E3 的代碼比 code ganker 好懂, 以後用 E3 的代碼.

这是一道链表操作的题目，要求把小于 x 的元素按顺序放到链表前面。我们仍然是使用链表最常用的双指针大法，一个(tao: 我稱為 A 指針)指向当前小于 x 的最后一个元素，一个(tao: 我稱為 B 指針)进行往前扫描。如果元素大于 x，那么(B 指針)继续前进，否则，要把元素移到前面(tao: 即 A 指針後)，并更新第一个指针(A 指針)。这里有一个小细节，就是如果不需要移动（也就是已经是接在小于 x 的最后元素的后面了 ← 此括號裡的不好懂, 可以不看. 此即 A 指針=B 指針的時候），那么只需要继续前进(B 指針, 當然最後也要統一前进 A 指針)即可。算法时间复杂度是 O(n)，空间只需要几个辅助变量，是 O(1)。代码如下. 这道题思路比较清晰，不过还是有点细节的，第一次写可能不容易完全写对，可以练习练习。

Code Ganker 的代碼(不用):

```
public ListNode partition(ListNode head, int x) {
    if(head == null)
        return null;
    ListNode helper = new ListNode(0);
    helper.next = head;
    ListNode walker = helper;
    ListNode runner = helper;
    while(runner.next!=null)
    {
        if(runner.next.val<x)
        {
```

```
        if(walker!=runner)
        {  //以下的四句是將 runner.next 移到 walker 後面
            ListNode next = runner.next.next; //此句是為刪除(即移走)原位置的 runner.next 作准備
            runner.next.next = walker.next; //此句和下一句是將 runner.next 插到 walker 後面. (記憶: 上一句把 runner.next.next 放在了別的變量中, 那麼緊接著就要給 runner.next.next 賦新值).
            walker.next = runner.next; // (記憶: 上一句把 walker.next 放在了別的變量中, 那麼緊接著就要給 next 賦新值).
            runner.next = next; //此句是刪除(即移走)原位置的 runner.next. (記憶: 前面弄一個 next 是何用? 這四句實際上形成了一個環, 每句右邊的變量就是下一句左邊的, 所以他們的順序一下就記住了, 其它情況(如互換兩個數)也可以類似記憶)
        }
        else //即 walker = runner 時, 這是處理開頭就是幾個比 k 小的 node 之 corner case. 注意此 else 沒有 {}, 故只管得住下一句 runner = runner.nex, 管不住 walker = walker.next
            runner = runner.next; //注意緊接著下面 walker 也要後移一位, 即 runner 和 walker 一起後移一位, 然後看 runner 下一位的值(即回到 if(runner.next.val < x)那裡去), 若 runner 下一位比 x 小, 則繼續將 runner 和 walker 一起後移一位, 否則若 runner 下一位比 x 大, 則執行下面 A 處, 即只將 runner 後移一位, walker 不動, 然後後面的就可以正常弄了. 這個技巧不好想到. E2 沒想到, 用了一種有點繞的方法. 不過這個技巧也有點繞.
        walker = walker.next;
      }
      else
      {
        runner = runner.next; //A 處
      }
    }
    return helper.next;
}
```

E3 的代碼:

```
    public ListNode partition(ListNode head, int x) {
        if(head == null || head.next == null) return head;
        ListNode fakeHead = new ListNode(0);
        fakeHead.next = head;
        ListNode pre = fakeHead, cur = fakeHead;
```

```
        //以下 while 执行完後, pre 指向那一片小於 x 的元素中的最後一個, 如 1->2->1->4->3->2->5->2 中的
1
        while(pre != null && pre.next != null && pre.next.val < x) {
            pre = pre.next;
        }

        //下句使得 cur 指向第一個大於或等於 x 的元素 如 1->2->1->4->3->2->5->2 中的 4
        cur = pre.next; //此時有 1->2->1(pre)->4(cur)->3->2->5->2

        //若所有元素都小於 x, 則直接返回原 list:
        if(cur == null) return head;

        //每輪循環中, 考查 cur.next 是否小於 x, 若小於, 則將 cur.next 移到 pre 之後
        while(cur.next != null) {
            if(cur.next.val < x) {
                ListNode preNext = pre.next, curNext = cur.next, curNextNext = cur.next.next;

                //以下三句是將 cur.next 移到 pre 之後
                pre.next = curNext;
                curNext.next = preNext;
                cur.next = curNextNext;

                pre = pre.next;
            } else {
                cur = cur.next;
            }

        }

        return fakeHead.next;
    }
```

87. Scramble String, Hard

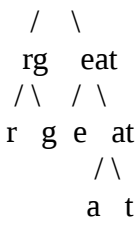
http://blog.csdn.net/linhuanmars/article/details/24506703

Given a string *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.
Below is one possible representation of *s1* = "great":

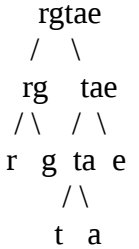
```
    great
   /    \
  gr    eat
 / \    / \
g   r  e   at
           / \
          a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.
For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

```
    rgeat
```



We say that "rgeat" is a scrambled string of "great".
Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".



We say that "rgtae" is a scrambled string of "great".
Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.
Subscribe to see which companies asked this question

Key: 本題是 3-d DP, 要用四重循環. 但只要知道遞推式, 就不怎麼難了. 題目的意思是對一個 string, 可以分出(構造出)很多個樹, 只要有一個分法能讓兩個 string 互為 scramble, 則稱這兩個 string 互為 scramble.

核心, 記住: 用一個三維數組 res[i][j][len]表示 '以 s1[i]開始的長度為 len 的子串' 和 '以 s2[j]開始的長度為 len 的子串' 是否互為 scramble. 要遞推的變量是 res[i][j][len]中的 len, 若已知 res[所有 i][所有 j][0, ...len-1], 如何得出 res[i][j][len]? 方法是將前面提到的那兩個長度為 len 的子串都分別 一刀劈為兩部分(即分成左子樹和右子樹), 有以下兩種情況:

s1: ------ (長度為 k) ---------- ----------(長度為 len-k)----------

i i+k-1 i+k i+len-1

s2: ------ (長度為 k) ---------- ----------(長度為 len-k)----------

j j+k-1 j+k j+len-1

以上情況, 若 's1 左段和 s2 左段互為 scramble' 且 's1 右段和 s2 右段互為 scramble', 則這兩個長度為 len 的子串就互為 scramble.

s1: ------ (長度為 k) ---------- ----------(長度為 len-k)----------

i i+k-1 i+k i+len-1

s2: ----------(長度為 len-k)---------- ------ (長度為 k) ----------

j j+len-k-1 j+len-k j+len-1

以上情況, 若 's1 左段和 s2 右段互為 scramble' 且 's1 右段和 s2 左段互為 scramble', 則這兩個長度為 len 的子串就互為 scramble.

以上兩種情況 只要有任一情況為 scramble, 則這兩個長度為 len 的子串就互為 scramble. 然後注意有很多種

劈法(即 k 有很多種取法), 只要有一種劈法能讓 這兩個長度為 len 的子串互為 scramble, 則 這兩個長度為 len 的子串就互為 scramble, 即 res[i][j][len]就為 true.

爭取不看遞推式能寫出代碼. 遞推式為
res[i][j][len] |= ((res[i][j][k] && res[i + k][j + k][len - k]) || (res[i][j + len - k][k] && res[i + k][j][len – k])).
注意 Java 中沒有||=, 已 record 到 Java p116.

實際寫代碼時, 要寫四重循環, 由外向內為: len, i, j, k (記憶: 跟 res[i][j][len]順序相反), 如果遞推式正確, 主要的工作就是將循環條件寫對了.

Convention: 實踐表明: Leetcode 中的 test case 默認兩個輸入 String 長度是相等的(若有長度不相等的 test case, 可以簡單地直接返回 false). 若兩個輸入 String 是一樣的(如 s1="helo", s2="helo"), 則它們也算是互為 scramble 的 (即要返回 true).

History:
E1 直接看的答案.
E2 寫好提交後, 一些 for 循環的循環條件中的 index 越界了, 改後即通過. 對本題這種難度的題目, 只改動了幾個 for 的循環條件就通過了, 都沒在我電腦上 test 過, 其實是很 NB 的.
E3 最開始用遞歸, 對題目中例子能給出正確結果, 但在 OJ 中果然超時. 然後 E3 覺得一定可以用動態規劃, 但就是想不到規劃一個甚麼樣的數組, 所以就沒做出來. 答案中的那個數組不容易想到, 要記住.

Tao: 本題雖然複雜, 但理解起來並不難. (scramble: 把…搅乱). 解法其實是維護一個三維數組, 所有臨時結果都放在數組中, 並沒有再利用這些空間, 所以我個人覺得這不算是動態歸劃. 後來發現其實也可以算是動態歸劃, 因為遞推式右邊的幾個 res[][][]值還會是在不同的循環輸中被多次用到, 而且下面我解釋 |= 時也有點動態歸劃的意思.

这道题看起来是比较复杂的，如果用 brute force，每次做切割，然后递归求解，是一个非多项式的复杂度，一般来说这不是面试官想要的答案。
这其实是一道三维动态规划的题目，我们提出维护量 res[i][j][n]，其中 i 是 s1 的起始字符，j 是 s2 的起始字符，而 n 是当前的字符串长度，res[i][j][len]表示的是以 i 和 j 分别为 s1 和 s2 起点的长度为 len 的字符串是不是互为 scramble。
有了维护量我们接下来看看递推式，也就是怎么根据历史信息来得到 res[i][j][len]。判断这个是不是满足，其实我们首先是把当前 s1[i...i+len-1]字符串劈一刀分成两部分，然后分两种情况：第一种是左边和 s2[j...j+len-1]左边部分是不是 scramble，以及右边和 s2[j...j+len-1]右边部分是不是 scramble；第二种情况是左边和 s2[j...j+len-1]右边部分是不是 scramble，以及右边和 s2[j...j+len-1]左边部分是不是 scramble。
(tao: 第一種情況是劈成:
s1: ----- ----------
s2: ----- ----------
第二種情況是劈成:
s1: ----- ----------
s2: ---------- -----
即 s2 的劈法是跟到 s1 來的. 以上的 visual 劈法對理解下面的遞推式很有幫助)
如果以上两种情况有一种成立，说明 s1[i...i+len-1]和 s2[j...j+len-1]是 scramble 的。而对于判断这些左右部分是不是 scramble 我们是有历史信息的，因为长度小于 n 的所有情况我们都在前面求解过了（也就是长度是最外层循环）。
上面说的是劈一刀的情况，对于 s1[i...i+len-1]我们有 len-1 种劈法，在这些劈法中只要有一种成立，那么两个串就是 scramble 的。
总结起来递推式是 res[i][j][len] = || (res[i][j][k]&&res[i+k][j+k][len-k] || res[i][j+len-k][k]&&res[i+k][j][len-k]) 对于所有 1<=k<len，也就是对于所有 len-1 种劈法的结果求或运算。因为信息都是计算过的，对于

每种劈法只需要常量操作即可完成，因此求解递推式是需要 O(len)（因为 len-1 种劈法）。
如此总时间复杂度因为是三维动态规划，需要三层循环，加上每一步需要线行时间求解递推式，所以是 O(n^4)。虽然已经比较高了，但是至少不是指数量级的，动态规划还是有很大有事的，空间复杂度是 O(n^3)。代码如下.
个人觉得这是 LeetCode 中最难的动态规划的题目了，要进行一次三维动态规划，对于维护量的含义也比较讲究。有朋友会讨论这个维护量是怎么提出来的，我自己也没什么绝对的方法，还是熟能生巧，靠"感觉"，做的题目多了就自然来了，这个做高中数学题有点类似哈，辅助线是靠"灵感"的哈。面试中如果遇到就是 top 难度的了，不过即使如此，只要思路清晰，还是可以记住的。如果没做过，个人觉得比较难当场想出来，不过算法大牛就另说了，这种题很经常出现在编程比赛中，ACM 高手还是不在话下的哈。

再次仔细阅读了这个帖子。继续惊叹于解法的完美。
同时有以下疑问：
1. 代码中 第 22 行 用了 |= 赋值， 其实想要表示的是 ||=的意思吗？（当然 Java 看来没有||=的用法，会编译不通过），这里恰好因为是 boolean 类型，所以|=的效果是一样的， 想要跟博主确认一下；
2. 还有我觉得最内一层循环里，可以只要一旦等式右边的部分为 true，就可以赋值并且 break 出那一层循环，可以稍微节约一点，不知道对吗？
再次感谢 code ganker

回复 jiaojialin_csdn：是的，因为是布尔操作，所以按位或是可以的哈 ~

Tao: 我自已寫時, 可以統一把 s1.length()和 s2.length()用一個變量(如 m)來表示, 這樣可以少打字.

```
public boolean isScramble(String s1, String s2) {
    if(s1==null || s2==null || s1.length()!=s2.length())
        return false;
    if(s1.length()==0)
        return true;
    boolean[][][] res = new boolean[s1.length()][s2.length()][s1.length()+1]; //為何第三維的長度為 s1.length()+1, 而不是 s1.length()? 因為後面的 for(int len=2;len<=s1.length();len++)中的 len 可以取 s1.length(), 而對於數組, 它的 index 最大值一旦可以取 a, 那麼這個數組的長度就要為 a+1, 因為最開始有個 index=0 的元素. 本題中 res[][][0]實際上並沒用到, 我們一直是對 res[][][1]到 res[][][s1.length()]賦值的.

//處理 len = 1 時之情況:
    for(int i=0;i<s1.length();i++)
    {
        for(int j=0;j<s2.length();j++)
        {
            res[i][j][1] = s1.charAt(i)==s2.charAt(j);
        }
    }

    for(int len=2;len<=s1.length();len++)
    {
        for(int i=0;i<s1.length()-len+1;i++) //為何是 i<s1.length()-1? 因為下面 res[i][j][len]要處理以 i 為起點的長度為 len 的字符串, 這個字符串最右的 index 為 i+len-1, 它必須小於 s1.length(), 即 i<s1.length()-len+1
        {
            for(int j=0;j<s2.length()-len+1;j++)
            {
                for(int k=1;k<len;k++)
                {
```

```
                res[i][j][len] |= res[i][j][k]&&res[i+k][j+k][len-k] || res[i][j+len-k][k]&&res[i+k][j][len-k];
 //為何要用 |= ? 因為它即 res[i][j][len] = res[i][j][len] | …., 而右邊的 res[i][j][len]是在上一個 k 時算出來的, 若它為 true, 而左邊的 res[i][j][len] (在本 k 時算出來的)為 false, 最後 res[i][j][len]還應當為 true.
              }
           }
        }
     }
     return res[0][0][s1.length()];
}
```

88. Merge Sorted Array, Easy

http://blog.csdn.net/linhuanmars/article/details/19712333

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.
Note:
You may assume that *nums1* has enough space (size that is greater or equal to *m* + *n*) to hold additional elements from *nums2*. The number of elements initialized in *nums1* and*nums2* are *m* and *n* respectively.
Subscribe to see which companies asked this question

Key: 由後往前取 A 和 B 中較大的元素放，放到 A 中，放的位置也是從 A 的後面往前放．這樣可以實現 in-place. 注意 A 已弄完， B 還沒弄完的時候.

History:
E1 直接看的答案.
E2 通過.
E3 幾分鍾寫好, 本來可以一次通過的, 但在 A 弄完, B 還沒弄完時, 在 while 中不小心忘了寫個 j--, 導致死循環(弱智錯誤), 改後即通過.

我: 這裡用 Code Ganker 的代碼．William 的方法也是一樣的，但代碼沒 Code Ganker 這麼對稱．

这是一道数组操作的题目，思路比较明确，就是维护三个 index，分别对应数组 A，数组 B，和结果数组。然后 A 和 B 同时从后往前扫，每次迭代中 A 和 B 指向的元素大的便加入结果数组中，然后 index-1，另一个不动。代码如下．

这里从后往前扫是因为这个题目中结果仍然放在 A 中，如果从前扫会有覆盖掉未检查的元素的可能性。算法的时间复杂度是 O(m+n),m 和 n 分别是两个数组的长度，空间复杂度是 O(1)。这个题类似的有 Merge Two Sorted Lists，只是后者是对于 LinkedList 操作，面试中可能会一起问到。

楼主，调用 java 类库的话怎么算时间复杂度和空间复杂度呢？比如这个题用
System.arraycopy(B, 0, A, m, n);
Arrays.sort(A);

回复 zhangwxly：排序是 O(nlogn)的复杂度，一般这种问题问出来面试官一般也不会接受用类库解决的方案哈~ 还是想看看 coding 的能力~

回复 linhuanmars：恩，我知道了，谢谢楼主了~

我：以下倒著放的方法，不用擔心 A 中放不下的問題．因為若只放 B 的元素，則 A 的末尾正好有那麼多空間給 B 放．若只放 A 的元素，則每放一個到 A 的末尾，A 非零的那段就會多出一個空間來．混放 AB 時也一樣能放下．

Code Ganker 的代碼(不用):

```
public void merge(int A[], int m, int B[], int n) {
    if(A==null || B==null)
        return;
    int idx1 = m-1;
    int idx2 = n-1;
    int len = m+n-1;
    while(idx1>=0 && idx2>=0)
    {
        if(A[idx1]>B[idx2])
        {
            A[len--] = A[idx1--];
        }
        else
        {
            A[len--] = B[idx2--];
        }
    }
    while(idx2>=0)    //即 idx1< 0 且 idx2>=0 的情況（即 A 已弄完，B 還沒弄完的時候）．注意 idx2<0 且 idx1>=0 時（即 B 已弄完，A 還沒弄完的時候）不用管，因為剩下的沒弄完的 A 元素本來就在 A 中
    {
        A[len--] = B[idx2--];
    }
//末尾加不加 return 都能在 leetcode 中通過．應該是（好像記得）返回值為 void 的函數可以沒有 return 語句，所有句子執行完了後自動 return.
}
```

E3 的代碼(用它, 注釋可看上面 Code Ganker 代碼中的, Code Ganker 代碼跟 E3 意思一樣):

```
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        if(n == 0) return;

        int i = m - 1, j = n - 1;

        while(i >= 0 && j >= 0) {
            if(nums1[i] > nums2[j]) {
                nums1[i + j + 1] = nums1[i];
                i--;
            } else {
                nums1[i + j + 1] = nums2[j];
                j--;
            }
        }

        while(j >= 0) {
            nums1[j] = nums2[j];
```

```
        j--;
      }
    }
```

89. Gray Code, Medium

http://wlcoding.blogspot.com/2015/03/gray-code.html?view=sidebar

The gray code is a binary numeral system where two successive values differ in only one bit.
Given a non-negative integer *n* representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.
For example, given *n* = 2, return [0,1,3,2]. Its gray code sequence is:
00 - 0
01 - 1
11 - 3
10 - 2
Note:
For a given *n*, a gray code sequence is not uniquely defined.
For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.
For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.
Subscribe to see which companies asked this question

Key: 看下面 Wiliam 說的.

Convention: Input 0 expected [0]

History:

E1 直接看的答案.

E2 最開始犯了點小錯, 改後即通過.

我: 以下用 William 的代碼．Code Ganker 的解釋太囉嗦, 沒有 William 的解釋好懂. Code Ganker 說的只有一句适有用:

Code Ganker: 感觉更接近于一道寻找规律的题目，实现上用到一点位运算会比较简单，不过位运算是这道题的考察点之一，面试中还是有关于位运算的题目，需要熟悉一下哈。

William(我: 我作了 minor 改動):

Time ~ O(N^2), Space ~ O(1)
可以用以下规则实现：
n = 1: 0 | 1
n = 2: 00 01 | 11 10
n = 3: 000 001 011 010 | 110 111 101 100
...
红线左边的为上一行序列从左往右每个码前加 0，
红线右边的为上一行序列从右往左每个码前加 1。
用两层循环：外层记录码的位数: 从 1 到 n (bits). 内层记录上一行码序列的 index，从 list.size() - 1 到 0。flipper 用于将码的最高位从 0 变为 1（用 | 或 + 都可）。

```
public List<Integer> grayCode(int n) {
```

```
    List<Integer> list = new ArrayList<>();
    list.add(0);
    for (int i = 0; i < n; i++) {//從i=0想起走,比較好懂. 第i輪(starts from第0輪)得到的list中含n=i+1對應的所有數.
        int flipper = 1 << i;
//為何以下的for要用倒序的? 因為以下的for若改為順序的, 即for(int j = 0; j < list.size(); j++)則在leetcode中是Time Limit Exceeded (input: 1). 應該是进入死循環了. 原因是循環體中每次要在list中加一個元素, 加了後list.size()就變了, 再作j < list.size()判斷時, list.size()就不是原來那個值了. 所以用for (int j = list.size() - 1; j >= 0; j--), 进入循環時只在初始化j時用一下list.size(), 之後不再用list.size()了.
        for (int j = list.size() - 1; j >= 0; j--)  // scan the previous sequence backward
//以下的list是上一輪(第i-1輪)留下來的, 它含n=i對應的所有數. 注意下面只需要在list中添加首位為1的數就可以了, 首位為0的數已經在上一輪中都得到了. 在做|運算時, 不足的高位應該是自動添0了, 比如10 | 100 = 010 | 100. 當然若拿不准, 也可以用William前面所說的+, leetcode中也能通過
            list.add(list.get(j) | flipper);        // flip the current highest bit from 0 to 1
    }
    return list;
}
```

90. Subsets II, Medium

http://wlcoding.blogspot.com/2015/03/subsets-i-ii.html?view=sidebar

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.
Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,
If ***nums*** = [1,2,2], a solution is:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
Subscribe to see which companies asked this question

Key: item-res 模版遞歸法. 代碼只比 Subsets 一題多一個 排序 和 if(i > start && nums[i] == nums[i - 1]) continue. 可以這樣理解: 例如題中的例子, 此 if 可以防止結果中出現兩個{1,2} ← 原因: 在輸入數組{1,2,2}中, 當 start 為 1 時, 即{1,2(start),2}時, i 取 start 時, 產生一個{1,2}結果, 當 i 取 start 下一個時, 就要跳過了.

本方法跟 40. Combination Sum II 中的一樣.

History:
E1 直接看的答案.

E2 一分鍾寫好, 一次通過.
E3 在本題 I 的 E3 代碼上改的, 本來可以一次通過的, 結果忘了排序, 改後即通過. E3 代碼跟 William 差不多. 為了跟前面一致, 以後用 E3 代碼.

Tao: 以下用 William 的代碼. Code Ganker 說的也沒甚麼用, 故沒 copy.

Backtracking: Time ~ O(N!), Space ~ O(N)
•剪枝条件：发现相同元素，且前一个相同元素没有用过（巧妙地使用了 i > start，表明前一元素 i - 1 已经被用过）。
•No need to check duplicate lists here, since the pruning already takes out the duplicates.

William 代碼(不用):

```
private List<List<Integer>> listSet = new ArrayList<List<Integer>>();
private List<Integer> list = new ArrayList<Integer>();

public List<List<Integer>> subsetsWithDup(int[] S) {
//以上我已經將函數名改為了 leetcode 認可的 subsetsWithDup
    Arrays.sort(S);
    addUp(S, 0);
    return listSet;
}

private void addUp(int[] S, int start) {
    // if (!listSet.contains(list))         // No need to check duplicate lists!!
        listSet.add(new ArrayList<Integer>(list));
    for (int i = start; i < S.length; i++) {
           if (i > start && S[i - 1] == S[i]) continue; // skip the duplicated
elements (important pruning step to increase speed)!!
        list.add(S[i]);
        addUp(S, i + 1);
        list.remove(list.size() - 1);
    }
}
```

E3 代碼(用它):

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(nums == null || nums.length == 0) return res;
    Arrays.sort(nums);
    helper(nums, 0, new ArrayList<Integer>(), res);
    res.add(new ArrayList<Integer>());
    return res;
}

private void helper(int[] nums, int start, List<Integer> item, List<List<Integer>> res) {
    for(int i = start; i < nums.length; i++) {
        if(i > start && nums[i] == nums[i - 1]) continue;
        item.add(nums[i]);
        res.add(new ArrayList<Integer>(item));
        helper(nums, i + 1, item, res);
        item.remove(item.size() - 1);
```

```
    }
  }
```

91. Decode Ways

http://blog.csdn.net/linhuanmars/article/details/24570759

A message containing letters from A-Z is being encoded to numbers using the following mapping:
'A' -> 1
'B' -> 2
...
'Z' -> 26
Given an encoded message containing digits, determine the total number of ways to decode it.
For example,
Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).
The number of ways decoding "12" is 2.
Subscribe to see which companies asked this question

Key: 用 res[i]來表示 s[0, ...i]這段有多少個 decode way. 若已知 res[i]之前的信息, 如何求出 res[i]? 此時 s[i]是新加进來的, 則有兩種情況, 一是 s[i]自己可對應一個字母, 此時有 res[i-1]個 decode way, 二是 s[i-1, i]這兩個數字一起對應一個字母(如 25 對應 Y), 此時有 res[i-2]個 decode way. 對於一個俱體的輸入, 有時這兩種情況都成立, 有時只有一種成立, 到底哪種成立, 由 s[i-1, i]決定(稱 s[i-1, i]為 n):
若 n = 10 或 20, 則 res[i] = res[i-2]
若 n= 00, 30, 40, 50, ...90, 則 res[i] = 0 (如輸入為 230 時, 無論如何都沒有 decode 的方法, 2, 30 或 23, 0 都不行)
若 11<=n<=19, 或 21<=n<=26, 則 res[i] = res[i-1] + res[i-2]
若 01<=n<=09, 或 27<=n<=99(不包括上面那些 30, 40 等), 則 res[i] = res[i-1]
由於只用到了 res[i], res[i-1], res[i-2]三個數, 所以實際代碼中可以不用數組, 而只有三個數來記錄 res[i], res[i-1], res[i-2]就可以了, 注意每輪循環最後要 update 它們的值. 如果要用數組來表示 res, 則返而有個細節比較 tricky, 就是要判斷 i 是否 i>=2, 若感興趣, 可看後面帖出來的 E3 代碼(不用此代碼).
Code Ganker 的 res[i]跟我不同, 不看.

History:
E1 直接看的答案.
E2 本來可以一次通過的, 但受 Code Ganker 的公式誤導, 沒有一次通過, 改後即通過. 我 E2 的代碼是用的一個數組 res[i], 但 Code Ganker 只用了三個數來表示, 我按他的方式改进了我 E2 的代碼, 改进後的 E2 代碼比 Code Ganker 的簡潔好懂, 以後用改进後的 E2 代碼(在最後).
E3 做了兩個多小時, 改了無數次才通過. 這是因為我最開始方法想錯了兩次, 並按這兩次錯方法寫了代碼. 第一次是只考查 s[i], 第二次是考查 s[i-2, i-1]和 s[i]. 最後才想到考查 s[i-1,i], 且是用數組來表示 res 的, 這樣寫起來反而有點 tricky, 見上面 key 中說的.

这道题要求解一个数字串按照字符串编码方式可解析方式的数量。看到这种求数量的，我们很容易想到动态规划来存储前面信息，然后迭代得到最后结果。
我们维护的量 res[i]是表示前 i 个数字有多少种解析的方式(我: 跟我 key 中不同)，接下来来想想递归式，有两种方式：第一种新加进来的数字不然就是自己比较表示一个字符，那么解析的方式有 res[i-1]种，第二种就是新加进来的数字和前一个数字凑成一个字符，解析的方式有 res[i-2]种（因为上一个字符和自己凑成了一个）。当然这里要判断前面说的两种情况能否凑成一个字符，也就是范围的判断，如果可以才有对应的解析方式，如果不行，那么就是 0。最终结果就是把这两种情况对应的解析方式相加。这里可以把范围分成几

个区间：
我: 以下的公式對 30, 40 等的情況不對, 別看, 看我 key 中的
我: 以下情況中, 例如 10 中的 0 表示第 i 個數字(即新加的那個), 1 表示第 i-1 個數字(即原有數字中最後那個), 其餘的(如 11, 27 等)也一樣.
（1）00：res[i]=0（无法解析，没有可行解析方式）；
（2）10, 20：res[i]=res[i-2]（只有第二种情况成立）；
（3）11-19, 21-26：res[i]=res[i-1]+res[i-2]（两种情况都可行）；
（4）01-09, 27-99：res[i]=res[i-1]（只有第一种情况可行）；
递推式就是按照上面的规则来得到，接下来我们只要进行一遍扫描，然后依次得到维护量就可以了，算法的时间复杂度是 O(n)。空间上可以看出我们每次只需要前两位的历史信息，所以只需要维护三个变量然后迭代赋值就可以了，所以空间复杂度是 O(1)。代码如下.
这道题是一维动态规划的题目，递推式关系来说是比较容易得到的，主要是要对这些两位数进行划分有一些细节，容易出小错误。

Code Ganker 的代碼(不用):

```
public int numDecodings(String s) {
    if(s==null || s.length()==0 || s.charAt(0)=='0')
    {
        return 0;
    }
    int num1=1; //num1 即 res[i-2]
    int num2=1; //num2 即 res[i-1]
    int num3=1; //num3 即 res[i]
    for(int i=1;i<s.length();i++)
    {
        if(s.charAt(i)=='0')
        {
            if(s.charAt(i-1)=='1' || s.charAt(i-1)=='2')
                num3 = num1;
            else
                return 0;
        }
        else
        {
            if(s.charAt(i-1)=='0' || s.charAt(i-1)>='3')
                num3 = num2;
            else
            {
                if(s.charAt(i-1)=='2' && s.charAt(i)>='7' && s.charAt(i)<='9')
                    num3 = num2;
                else
                    num3 = num1+num2;
            }
        }
        //在下一輪 for 循環開始前先更新:
        num1 = num2;
        num2 = num3;
    }
```

```
    return num2; //由於 for 的末尾已做了 num2=num3, 故返回 num2
}
```

改进後的 E2 代碼(用它):

```
    public int numDecodings(String s) {
        if(s == null || s.length() == 0 || s.charAt(0) == '0') return 0;
        int resIMinus2 = 1, resIMinus1 = 1, resI = 1;

        for(int i = 1; i < s.length(); i++) {
            int n = Integer.parseInt(s.substring(i - 1, i + 1));

            if(n == 10 || n == 20) resI = resIMinus2;
            else if(n % 10 == 0) resI = 0;
            else if((11 <= n && n <= 19) || (21 <= n && n <= 26)) resI = resIMinus1 + resIMinus2;
            else resI = resIMinus1;

            resIMinus2 = resIMinus1;
            resIMinus1 = resI;
        }

        return resIMinus1;
    }
```

E3 的代碼(不用):

```
    public int numDecodings(String s) {
        if(s == null || s.length() == 0) return 0;
        if(s.charAt(0) == '0') return 0;
        int n = s.length();
        if(n == 1) return 1;
        int[] d = new int[n];
        d[0] = 1;

        for(int i = 1; i < n; i++) {
            int cur = Integer.parseInt(s.substring(i - 1, i + 1));
            if(cur == 10 || cur == 20) d[i] = (i >= 2 ? d[i - 2] : 1); //注意這裡對 i>=2 的判斷. 為何不在 for 前面專門處理 d[1]? 因為那樣也很麻煩.
            else if(cur %10 == 0) return 0;
            else if(1 <= cur && cur <= 9) d[i] = d[i - 1];
            else if(11 <= cur && cur <= 26) d[i] = d[i - 1] + (i >= 2 ? d[i - 2] : 1);
            else d[i] = d[i - 1];
        }

        return d[n - 1];
    }
```

92. Reverse Linked List II, Medium

Reverse a linked list from position *m* to *n*. Do it in-place and in one-pass.
For example:

Given 1->2->3->4->5->NULL, *m* = 2 and *n* = 4,
return 1->4->3->2->5->NULL.
Note:
Given *m*, *n* satisfy the following condition:
1 ≤ *m* ≤ *n* ≤ length of list.
Subscribe to see which companies asked this question

Key: 本題中用一個 cur, 意思跟 206. Reverse Linked List 中一樣, 即 cur 圖定在 m 處, 除此之外還要用個 pre, 固定在 m 之前的那個元素處, 用一個 start 表示上一輪移動過的那個元素(即上一輪的 cur.next), 此輪要將 cur.next 移到 pre 之後 start 之前. 還要弄個 falseHead.

若輸入為 123456789, m=3, n=6, 則:
12(pre)3(cur,start)456789
12(pre)4(start)3(cur)56789
12(pre)5(start)43(cur)6789
12(pre)6(start)543(cur)789

History:
E1 的代碼在 leetcode 中是一次通過. 以下用我 E1 的代碼(因為 206. Reverse Linked List 也是用的我的代碼, 此處保待一致).
E2 居然還改了好幾次才通過.
E3 一次通過, 代碼跟 E1 差不多.

Code Ganker 的代碼好像比我 E1 的還長一點點, 沒看. 他說的只有一句話有用:

Code Ganker: 链表的题目就是这样，想起来道理很简单，实现中可能会出些小差错，还是熟能生巧哈。

E1 的代碼:
```
public static ListNode reverseBetween(ListNode head, int m, int n) {
 if(head == null || head.next == null)
  return head;

 ListNode falseHead = new ListNode(0);
 falseHead.next = head;

 ListNode pre = falseHead; //pre 用來表示 m 之前的那個元素
 ListNode cur = falseHead; //cur 的意思跟 206. Reverse Linked List 中一樣, 即依次將 cur.next 移到前面去, 即 pre 之後

//以下為得到 pre 和 cur, 注意 cur = pre.next 一句在 for 循環之外
 for(int i = 1; i < m; i++)
  pre = pre.next;
```

```
  cur = pre.next;

  ListNode start = cur; //start 的意思見下面

  for(int i = m; i < n; i++) {
   //以下兩句是刪除 cur.next
   ListNode temp = cur.next;
   cur.next = cur.next.next;

   //以下幾句是把 cur.next 添到前面去, 即 pre 之後. start 表示上一輪移動過的那個元素(即上一輪的 cur.next), 此輪要將 cur.next 移到 start 之前
   temp.next = start;
   pre.next = temp;
   start = temp; //更新 start, 使 start 等於本輪的 cur.next (前面給 temp 賦的值為 cur.next)
  }

  return falseHead.next;
 }
```

93. Restore IP Addresses, Medium

http://wlcoding.blogspot.com/2015/03/restore-ip-addresses.html?view=sidebar

Given a string containing only digits, restore it by returning all possible valid IP address combinations.
For example:
Given "25525511135",
return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)
Subscribe to see which companies asked this question

Key:
題意: 一個有效的 IP 地址由 4 個數字(我給每個這樣數字叫 vrangh)組成, 每個 vrangh 在 0 到 255 之間 inclusively. 若 vrangh 只有一位數, 則它可以為 0; 否則若 vrangh 位數大於一, 則不能以 0 開頭.

E3 的方法: 遞歸. 寫一個 List<String> helper(String s, int numDots), 其作用為: 將 s 弄成 '有 numDots 個點的 IP 地址(如 s=2135, numDots = 1, 則 2.135 就是一個這樣的 IP 地址)', 並將所有這樣的 IP 地址放入一個 list 中, 返回這個 list. 在 helper 中, 先取 s 前端的一個 substring(叫它小明), 其長度為 1, 2, 或 3, 若小明是一個 vrangh, 則遞歸調用 helper(s 除了小明後剩下的 substring, numDots -1), 再將小明加一點 和 'helper 得到的 list 中所有的 string' 連接起來即可. 遞歸的終止條件即為 numDots = 0.

注意本題的時間複雜度.

William 的方法:

寫一個 boolean isValidIP(String s)來判斷 s 是不是一個合法的 vrangh.

寫一個 dfs(String s, int numPt, StringBuilder path, List<String> list), 其作用為: 輸入的 s 是原 s 剩下的沒弄的那段, numPt 用來記錄加了多少個點, 在 s 前端取不同長度的 vrangh, 若其為合法 vrangh, 則將它加入到 path 中, 若原 s 都加 path 中了(由 numPt 知, 比如 numPt 為 3 時, 則加現在的 s 到 path 中, 然後將 path 加到 list 中), 則將 path 加到 list 中去. 注意每次要先加 vrangh 再加點. E2 是先加點再加 vrangh, 這樣要額外處理 path 中的第一個 vrangh(因為此 vrangh 之前不用加點), which is 不好. 最後, 注意回溯保護現場.

Corner cases:
輸入 0000, 要求輸出["0.0.0.0"], 而不是[]
輸入的 String 長度 小於 4 或 大於 12, 要求輸出[]
輸入"", 要求輸出[], 而不是[""]

History:
E1 直接看的答案.
E2 改了很多遍才通過. 以上 key 都是 E2 寫的.
E3 改了兩三次即通過. E3 的方法跟 William 的類似, 但俱體寫法還是有較大不同, 我的方法沒用 William 那個臨時放結果的 path, 故不用保護現場, 所以我的代碼要好懂點, 以後用我 E3 的代碼 .

我: William 的代碼要簡潔些, 做用 William 的代碼. Code Ganker 說的也沒甚麼有用的, 故沒 copy.

William:
DFS: Time ~ O(N^4), Space ~ O(N)
•isValidIp(String s)：判断 s 是否是 0 ~ 255 的数字，且不以 0 开头（!s.startsWith("0")）；
•dfs(String s, int numPt, StringBuilder path, List<String> list>：分解剩下的 s（每次递归送入 s.substring(i)（我：即下一輪的 s 就從 s.substring(i)之後的開始，i=1 或 2 或 3），类似 Regular Expression (Wildcard) Matching 里的用法）；用 numPt 记录已经加入的 dot 的数目，若为 3 则只需考虑 isValidIp(s)，否则需要分别取 s 的 1、2、3 位来进行递归。

```
public List<String> restoreIpAddresses(String s) {
    List<String> list = new ArrayList<>();
    dfs(s, 0, new StringBuilder(), list);
    return list;
}

//以下的 path 是前面(上一輪)已經弄好了的(valid 的)IP 地址(如 233.131.122.12 中的 223.131), s 為上一輪中沒弄完的(剩下的)那一段(如 233.131.122.12 中的 122.12)
//dfs()的功能是將輸入的 s 弄成合理的 IP(子)地址並加到輸入的 path 後, 這對理解下面的回溯有幫助
private void dfs(String s, int numPt, StringBuilder path, List<String> list) {
    if (numPt == 3) {
        if (isValidIp(s)) list.add(path.toString() + s);
    } else {
        int len = path.length();
        for (int i = 1; i <= 3 && i <= s.length(); i++) {//i <= 3 是因為每一節不能超過
```

三位數

```
                String num = s.substring(0, i);//string.substring 函數見 Java 書 p418(已record 到 p409)
                if (isValidIp(num))
                    dfs(s.substring(i), numPt + 1, path.append(num).append('.'), list);
                      path.delete(len, path.length());// 回溯. len 為 path 之前的長度, path.length()是上一句 dfs 中改變了 path 後的 path 長度. path.delete 之目的就是刪掉上一句 dfs 中新加的那一段. 為何要這樣做?例如當 i=1 時, 上一句的 dfs 處理了 s.substring(0, 1) (e.g., 233.131.122.12 中的 233.1), 但我們接著要處理 i=2 時的 s.substring(0, 2) (e.g., 233.131.122.12 中的 233.13), 而在處理前, 顯然要先將 233.1 中的 1 刪掉, 再重新加入 13, 成為 233.13. path.delete 就是起這個作用(stringBuilder.delete 函數見 Java 書 p416)
            }
        }
    }

    private boolean isValidIp(String s) {
            if (s.length() == 1 || s.length() >= 1 && s.length() <= 3 && !s.startsWith("0")) { //注意 public boolean startsWith(String prefix)的用法, self-explanatory, 已 recod 至 java p409. E3 沒用這個函數.
            int num = Integer.parseInt(s); //注意 Integer.parseInt(string)的用法
            if (num >= 0 && num <= 255) return true;
        }
        return false;
    }
```

E3 的代碼:

```
    public List<String> restoreIpAddresses(String s) {
        List<String> res = new ArrayList<>();
        if(s == null || s.length() < 4 || s.length() > 12) return res;
        return helper(s, 3);
    }

//List<String> helper(String s, int numDots) 的作用為: 將 s 弄成 '有 numDots 個點的 IP 地址(如 s=2135, numDots = 1, 則 2.135 就是一個這樣的 IP 地址)', 並將所有這樣的 IP 地址放入一個 list 中, 返回這個 list.
    private List<String> helper(String s, int numDots) {
        List<String> res = new ArrayList<>();

        if(numDots == 0 && isVrangh(s)) {
            res.add(s);
            return res;
        }

        for(int i = 1; i <= 3 && i < s.length(); i++) {
            String front = s.substring(0, i);
            if(isVrangh(front)) {
                List<String> resRest = helper(s.substring(i, s.length()), numDots - 1);
                if(resRest.isEmpty()) continue;
                for(String rest : resRest) res.add(front+"."+rest);
            }
        }
```

```
        return res;
    }

    private boolean isVrangh(String s) {
        if(s.equals("0")) return true;
        if(s.charAt(0) == '0') return false;
        if(s.length() > 3) return false;
        int sInt = Integer.parseInt(s);
        if(1 <= sInt && sInt <= 255) return true;
        return false;
    }
```

94. Binary Tree Inorder Traversal, Medium

http://blog.csdn.net/linhuanmars/article/details/20187257

Given a binary tree, return the *inorder* traversal of its nodes' values.
For example:
Given binary tree `{1,#,2,3}`,

```
  1
   \
    2
   /
  3
```

return `[1,3,2]`.
Note: Recursive solution is trivial, could you do it iteratively?
confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key: E3 的迭代法(iteratively): 將輸入樹的左邊緣的節點都放入 stack 中. 然後依次將 stack 中的節點 pop 出來, 若 pop 出來的節點含有右兒子, 則從該右兒子開始, 一路向左全加入 stack 中, 好留給下一輪循環用. 自己畫一個有好幾層的滿樹就想清楚了. 由 iteratively 方法知, DFS 不一定要用遞歸. 要看看遞歸法.

但我後來又覺得 Code Ganker 的代碼好些.
Tree traversal 迭代法以後統一用 LeetCode Discussion 上的代碼(已帖在後面).

Code Ganker 的迭代法(iteratively)：用棧，對每一個 node, 都從 node 開始沿著左邊路走到盡頭並把沿途元素加到棧中, 然後 pop 出一個, 將其附給 node, 加到 res 中，再將 node 換成 node.right. 只用一個 while, 注意 while 中的條件. 本題其實並不簡單, 可能需要將代碼背下來(應當按模板背下來).

History:
E1 直接看的答案.
E2 沒做出來.
E3 的遞歸法很快寫好, 一次通過, 代碼跟 Code Ganker 的幾乎一模一樣. 對於迭代法, E3 試著回憶 Code Ganker 的方法, 但沒想出來, 所以按自己的方法寫的, 寫好後沒一次通過, 改了下某幾行順序, 即通過. 後來看 Code Ganker 的代碼, 發現既不好懂, 也不好記憶. 而 E3 的代碼雖然沒 Code Ganker 的簡短, 但由於是我自己想出來的, 所以要好懂些, 以後用 E3 的代碼. 後來發現 William 的代碼結構跟我的很相似, 即一個 while, 然後一個雙重 while.

Code Ganker:

通常，实现二叉树的遍历有两个常用的方法：一是用递归，二是使用栈实现的迭代方法。下面分别介绍。
递归应该最常用的算法，相信大家都了解，算法的时间复杂度是 O(n), 而空间复杂度则是递归栈的大小，即 O(logn)。代码如下：

Code Ganker 代碼(不用):

我：以下代碼看看就可以了，因為題目中說了不喜歡遞歸法．我要記住的是後面的迭代法．迭代法是由遞歸法演變來的（關係不是很接近），所以要看看這裡的遞歸法．

```
public ArrayList<Integer> inorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    helper(root, res);
    return res;
}
```

我：以下 helper(root, res)函數的作用就是把 root 以及它的所有後代都按 inorder 加入到 res 中．可以很容易用數學歸納法證明這點，初始情況可以用一個只有三個元素的樹（root-left-right）．在理解下面遞歸時，只要抓住 helper(root, res)的作用就好理解了．

```
private void helper(TreeNode root, ArrayList<Integer> res)
{
    if(root == null)
        return;
    helper(root.left,res);
    res.add(root.val);
    helper(root.right,res);
}
```

接下来是迭代的做法，其实就是用一个栈来模拟递归的过程。所以算法时间复杂度也是 O(n)，空间复杂度是栈的大小 O(logn)。过程中维护一个 node 表示当前走到的结点（不是中序遍历的那个结点），实现的代码如下：
(我: William 只用了迭代法(inorder 也用了 Morris 法的, preorder, postorder 等沒用), 代碼跟 Code Ganker 差不多)

我：以下代碼用一個只有三個元素的樹（root-left-right）想一想，就明白了．

Tree traversal 迭代法以後統一用 LeetCode Discussion 上的代碼:

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode p = root;
    while(!stack.isEmpty() || p != null) {
        if(p != null) {
            stack.push(p);
            p = p.left;
        } else {
```

```
            TreeNode node = stack.pop();
            result.add(node.val);  // Add after all left children
            p = node.right;
        }
    }
    return result;
}
```

Code Ganker 的代碼(不用):

```
public ArrayList<Integer> inorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    while(root!=null || !stack.isEmpty()) //注意此處是||, 不是&&, 可用一個只有三個元素的樹（root-left-right）想一想．另外，root!=null 不能去掉，否則 while 循環永遠执行不了，因為 stack 最初都是空的．
    {
        if(root!=null)
        {
         //以下為沿著左邊路走到盡頭
            stack.push(root);
            root = root.left;
        }
        else
        {
            root = stack.pop();
            res.add(root.val);
            root = root.right; //注意此行右邊的 root 就是上一行中加入 res 的那個 root, 不要想成其它的了
        }
    }
    return res;
}
```

最后我们介绍一种比较复杂的方法，这个问题我有个朋友在去 google onsite 的时候被问到了，就是如果用常量空间来中序遍历一颗二叉树。这种方法叫 Morris Traversal。想用 O(1)空间进行遍历 (我：以下的省略，因為有的人認為不大容易被考到，也沒看．見 144. Binary Tree Preorder Traversal 的評論)

評論：

您好！请问迭代算法那儿：res 是用 ArrayList 实例化的，而 stack 是用 LinkedList 实例化。其中，stack 是要经常 push，pop 所以用 LinkedList 好些是吗？那 res 为什么用 ArrayList 呢？

回复 saorenyu：从用栈的角度两个是一样的~ res 用 ArrayList 是因为以前 leetcode 输出的接口是 ArrayList 哈~ 没什么特别的~

大牛，这个方法为啥不弄 public 呢？
private void helper(TreeNode root, ArrayList<Integer> res)

回复 youngtoleetcode：内部使用的算法只需要 private 就行，从设计的角度不允许外界调用~

E3 的代碼(不用):

```
    public List<Integer> inorderTraversal(TreeNode root) {
```

```
        List<Integer> res = new ArrayList<>();
        if(root == null) return res;
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode l = root;
        stack.push(l);

        //將輸入樹的左邊緣的節點都放入 stack 中
        while(l.left != null) {
            l = l.left;
            stack.push(l);
        }

        while(!stack.isEmpty()) {
            TreeNode cur = stack.pop();
            res.add(cur.val);

            if(cur.right != null) {
                stack.push(cur.right);

                //以下幾句是處理 cur.right 存在時之情況, 此時也要將 以 cur.right 為根的子樹 的左邊緣的節點都放入 stack 中, 好留給下一輪循環(while(!stack.isEmpty()))用
                TreeNode curL = cur.right;
                while(curL.left != null) {
                    curL = curL.left;
                    stack.push(curL);
                }
            }
        }

        return res;
    }
```

95. Unique Binary Search Trees II, Medium

http://blog.csdn.net/linhuanmars/article/details/24761437

Given *n*, generate all structurally unique **BST's** (binary search trees) that store values 1...*n*.
For example,
Given *n* = 3, your program should return all 5 unique BST's shown below.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key: 遞歸．寫一個 generate(int start, int end)，其作用是返回一堆 BST（當然是它們的根組成的一個 List），這些樹都是由 start 到 end 的數填起來的．對於 start 到 end 之間的某個數 k, 用 generae(start, k-1)得到一個 list, 用 generate(k+1, end)得到一個 list. 在這兩個 list 中各取一個, 設為 node k 的左右子樹.

Corner case: 當 n=0 時, 要求返回一個空的 list, 而不是一個含有 null 的 list.

History:
E1 直接看的答案.
E2 最開始在 star > end 時沒加 null(詳見我在代碼中說的), 改後可通過.
E3 沒做出來, 因為我想的方向不對, 我想的是從 n 個節點的所有樹中添一個節點, 得到 n+1 個節點的所有樹, 但這樣會弄出來很多重復的樹, 所以行不通.

我：注意理解題意. 題目中要求返回一個 List<TreeNode>, 並不是要求返回一個{1, #, 2, 3}這樣的東西, 而是一個由 TreeNode 組成的 List, 這些 TreeNode 每一個都是 一個可能的樹的 root.

William Tag: DFS.
我: William 的代碼跟 Code Ganker 的基本上一模一樣. 但 Code Ganker 代碼中變量取的名字容易讓人誤解, 所以以下用 William 的代碼.

William:
DFS: Time ~ O(?), Space ~ O(?) (複雜度看下面 Code Ganker 說的)
(我： 以下偽碼很對理解代碼很有幫助)
Call recursive method that returns all the unique BST's from start to end:
generate(start, end);
Given node k (from start to end),
leftSubs = generate(start, k - 1);
rightSubs = generate(k+ 1, end);
for (left : leftSubs)
  for (right : rightSubs)
   { TreeNode node = new TreeNode(k); node.left = left; node.right = right; }

代碼：
```
public static class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int x) { val = x; }
}

public List<TreeNode> generateTrees(int n) {
    if(n <= 0) return new ArrayList<TreeNode>(); //本句是 E3 加的, 不加通不過. 這是因為當n=0 時, OJ 要求返回一個空的 list. 而若不要此句, 則按後面的那句 if(start > end), 返回的是一個含有 null 的 list, 這不符合要求.
    return generate(1, n);
}

//generate(int start, int end)的作用就是返回一堆樹 ( 當然是它們的根組成的一個 List ) , 這些樹都是由 start 到 end 的數填起來的 .
private List<TreeNode> generate(int start, int end) {
    List<TreeNode> subTree = new ArrayList<>();
//E3 表示, 對以下 if 最好的理解, 就是用個例子: generate(1, 3). 當 k=2 時, 我們要建一個以 2 為根的樹, 此時很簡單, 因為所以左子樹為 1, 右子樹 3. 但當 k=1 時, 左子樹為 null, 右子樹為 2 和 3, 此 null 就是在执行 leftSubs = generate(1, 1 – 1)時, 此 generate(1, 1-1)中的 if(start > end)弄出來的, 即 leftSubs=null, 這正是我們想要的.
//E2 最開始寫的 if(start > end) return subTree, 沒有加 null, 結果給出錯誤結果. 這是因為若不加 null, 則 subTree 就是一個空集, 下面的 for(TreeNode left : leftSubs)就不會执行, 而實際上我們是想 for 执行並將 node.left 設為 null. 這就是我們為何必須在 subTree 中加 null.
    if (start > end) {
        subTree.add(null);
```

```
        return subTree;
    }

    for (int k = start; k <= end; k++) {//k 為 leftSubs 和 rightSubs 的 parent, 我們的目的是產生所有可能的子樹，所以 k 當然要從 start 到 end 試完．
        List<TreeNode> leftSubs = generate(start, k - 1);
        List<TreeNode> rightSubs = generate(k + 1, end);
        for (TreeNode left : leftSubs) {
            for (TreeNode right : rightSubs) {
                TreeNode node = new TreeNode(k);
                node.left = left;
                node.right = right;
                subTree.add(node);
            }
        }
    }

    return subTree;
}
```

Code Ganker 說的也看看：

这道题是求解所有可行的二叉查找树，从 Unique Binary Search Trees 中我们已经知道，可行的二叉查找树的数量是相应的卡特兰数，不是一个多项式时间的数量级，所以我们要求解所有的树，自然是不能多项式时间内完成的了。算法上还是用求解 NP 问题的方法来求解，也就是 N-Queens 中介绍的在循环中调用递归函数求解子问题。思路是每次一次选取一个结点为根，然后递归求解左右子树的所有结果，最后根据左右子树的返回的所有子树，依次选取然后接上（每个左边的子树跟所有右边的子树匹配，而每个右边的子树也要跟所有的左边子树匹配，总共有左右子树数量的乘积种情况），构造好之后作为当前树的结果返回。代码如下（我：已省之）

实现中还是有一些细节的，因为构造树时两边要遍历所有左右的匹配，然后接到根上面。当然我们也可以像在 Unique Binary Search Trees 中那样存储所有的子树历史信息，然后进行拼合，虽然可以省一些时间，但是最终还是逃不过每个结果要一次运算，时间复杂度还是非多项式的，并且要耗费大量的空间，感觉这样的意义就不是很大了。

我：以下的討論都是然並卵，還沒我對遞歸的理解好．其實只要清楚被遞歸函數的作用，甚麼都好理解了．

请教一个思路：
什么时候使用循环中嵌套递归呢？
什么时候仅使用递归？
因为递归本身也是在穷举， 加上循环有什么意义呢？
我碰到好几次用循环套递归的问题了，一直越不过这个坎。。。
谢谢了

回复 thendore：通常来说循环中嵌套递归时对于 NP 问题会比较常用，其实还是看问题需求，有时候也不能说递归是在穷举，其实递归只是实现的手段，跟算法无关哈~

回复 linhuanmars：因为你在 N – queen 问题里说了其他的几个问题是同种套路，是这个意思么

回复 thendore：一般 NP 问题都可以用这个套路来做~ 不过其实就是一种 brute force 的实现方式~

回复 linhuanmars：单次递归是因为当前问题比较复杂，才递推至和当前问题一样性质但规模稍小一点的一个子问题去解决，那递归外面套循环的意思是当前问题是由多个子问题共同决定，是这个意思吗？

回复 thendore：也不是共同决定，只是问题更加复杂，每个循环都对应着子问题的解决~

96. Unique Binary Search Trees, Medium

http://blog.csdn.net/linhuanmars/article/details/20187257

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?
For example,
Given *n* = 3, there are a total of 5 unique BST's.

```
  1         3     3      2     1
   \       /     /      / \     \
    3     2     1      1   3     2
   /     /       \                \
  2     1         2                3
```

Subscribe to see which companies asked this question

Key: 本題實際上是求 有 n 個節點的不同的 tree(不要求是 binary search tree)的數量(解釋見下). 見下面的遞推式. 動態歸劃. 维护量 res[i]表示含有 i 个结点的二叉查找树的数量。根据递推式依次求出 1 到 n 的的结果即可。

History:
E1 直接看的答案.
E2 幾分鍾寫好, 本來可以一次通過的, 但在遞推式右邊忘了減 1, 改後即通過.
E3 沒做出來. E3 對本題解法點映像都沒有了, 根本沒想到用動態歸劃.

我: 本題實際上是求 有 n 個節點的不同的二叉樹(不要求是二叉查找樹, 可以想象成所有節點都沒有值)的數量, 不用管題目中說的節點的值 1,2..n. 因為只要 一個 n 節點的二叉樹 存在, 那就一定有且只有一種方法把 1,2..n 全填入節點使其成為一個二叉查找樹. 此方法就是: 中序遍歷此二叉樹(注意二叉查找樹的性質是中序遍歷有序), 在遍歷的過程中依次填入 1,2...n, 就可以了.

Code Ganker:
这道题要求可行的二叉查找树的数量，其实二叉查找树可以任意取根，只要满足中序遍历有序的要求就可以。从处理子问题的角度来看，选取一个结点为根，就把结点切成左右子树，以这个结点为根的可行二叉树数量就是左右子树可行二叉树数量的乘积，所以总的数量是将以所有结点为根的可行结果累加起来。写成表达式如下：

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i\, C_{n-i} \quad \text{for } n \geq 0.$$

(我: 以上公式左邊是 n+1, 右邊加起來是 i + n-i = n, 左邊比右邊多一個, 多出來那一個就是根)
熟悉卡特兰数的朋可能已经发现了，这正是卡特兰数的一种定义方式，是一个典型的动态规划的定义方式（根据其实条件和递推式求解结果）。所以思路也很明确了，维护量 res[i]表示含有 i 个结点的二叉查找树的数量。根据上述递推式依次求出 1 到 n 的的结果即可。
时间上每次求解 i 个结点的二叉查找树数量的需要一个 i 步的循环，总体要求 n 次，所以总时间复杂度是 O(1+2+...+n)=O(n^2)。空间上需要一个数组来维护，并且需要前 i 个的所有信息，所以是 O(n)。代码如下.
这种求数量的题目一般都容易想到用动态规划的解法，这道题的模型正好是卡特兰数的定义。当然这道题还可以用卡特兰数的通项公式来求解，这样时间复杂度就可以降低到 O(n)。因为比较直接，这里就不列举代码了。
如果是求解所有满足要求的二叉树（而不仅仅是数量）那么时间复杂度是就取决于结果的数量了，不再是一

个多项式的解法了，有兴趣的朋友可以看看 Unique Binary Search Trees II。

```
public int numTrees(int n) {
    if(n<=0)
        return 0;
    int[] res = new int[n+1]; //res[i]表示含有 i 个结点的二叉查找树的数量. 因為 res 要從 res[0]開始, 所以總共是 n+1 個
    res[0] = 1; //res[0]只是用來初始化, 沒有俱體的意義.
    res[1] = 1;
    for(int i=2;i<=n;i++)
    {
        for(int j=0;j<i;j++)
        {
            res[i] += res[j]*res[i-j-1]; //注意前面遞推式中左邊是 res[n+1], 而此處左邊是 res[i], 所以右邊是 res[j]和 res[i-j-1], 加起來為 i-1
        }
    }
    return res[n];
}
```

97. Interleaving String, Hard

http://blog.csdn.net/linhuanmars/article/details/24683159

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.
For example,
Given:
s1 = "aabcc",
s2 = "dbbca",
When *s3* = "aadbbcbcac", return true.
When *s3* = "aadbbbaccc", return false.
Subscribe to see which companies asked this question

interleaving: 交织

以下是 William 對題意的解釋:

For example,
Given: s1 = "aabcc", s2 = "dbbca",
When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.

Key: 動態歸劃. 用 boolean res[i][j]表示 's1 前 i 個字符(即 s1[0,..i-1])' 和 's2 前 j 個字符(即 s2[0,...j-1])' interleave 成 's3 的前 i+j 個字符(即 s3[0,...i+j-1])'. [注 1]. 若已知 res[i][j]之前的信息, 如何算 res[i][j]? 有兩種情況, 第一種情況是若 's1 的前 i-1 個字符' 和 's2 前 j 個字符' 已經 interleave 成了 's3 的前 i+j-1 個字符', 則可比較 's1 的第 i 個字符(注意此處的第是從第 1 開始的, 所以是 s1[i-1])' 和 's3 的第 i+j 個字符(同左, 即 s3[i+j-1])' 是否相符; 第二種情況是若 's1 的前 i 個字符' 和 's2 前 j-1 個字符' 已經 interleave 成了 's3 的前 i+j-1 個字符', 則可比較... 兩種情況只要有一種是 interleave, 則 res[i][j]即為 true.

遞推式為(爭取不看): res[i][j] = res[i-1][j] && s1[i] == s3[i+j] || res[i][j-1] && s2[j] == s3[i+j], 注意 res[0][j]和

res[i][0]都要初始化.
要寫成 1d-DP, 即爭取用一維數組(William 叫的滾动数组, William 的代碼跟 Code Ganker 一模一樣)．注意, 雖然是 1d-DP, 但仍要用兩層循環(即 i 循環和 j 循環)．

Corner case:
1. 輸入的三個 String 有一個或多個 String 的長度為 0(即"").
2. s3 的長度不等於 s1 和 s2 長度之和
Convention:
若 s1 取 0 個字符, s2=s3, 這樣也算 interleave. 若 s2 取 0 個字符...

[注 1]: E2 最開始是這樣寫的: 用 res[i][j]表示 s1[0,..i]和 s2[0,...j] interleave 成 s3[0,...i+j+1], 但後來發現這樣不好, 因為它沒法表示這種情況: s1 的前 0 個字符(即 s1 不取字符) 和 s2 的一些字符 interleave 成 s3.

History:
E1 直接看的答案.
E2 直接按 1-d DP 寫的, 改了幾次後通過, 代碼跟 Code Ganker 基本一樣.
E3 按 2-d DP 寫的, 改了幾次後通過.

这是一道关于字符串操作的题目，要求是判断一个字符串能不能由两个字符串按照他们自己的顺序，每次挑取两个串中的一个字符来构造出来。
像这种判断能否按照某种规则来完成求是否或者某个量的题目，很容易会想到用动态规划来实现。
先说说维护量，res[i][j]表示用 s1 的前 i 个字符和 s2 的前 j 个字符能不能按照规则表示出 s3 的前 i+j 个字符，如此最后结果就是 res[s1.length()][s2.length()]，判断是否为真即可。接下来就是递推式了，假设知道 res[i][j]之前的所有历史信息，我们怎么得到 res[i][j]。可以看出，其实只有两种方式来递推，一种是选取 s1 的字符作为 s3 新加进来的字符，另一种是选 s2 的字符作为新进字符。而要看看能不能选取，就是判断 s1(s2)的第 i(j)个字符是否与 s3 的 i+j 个字符相等。如果可以选取并且对应的 res[i-1][j](res[i][j-1])也为真，就说明 s3 的 i+j 个字符可以被表示。这两种情况只要有一种成立，就说明 res[i][j]为真，是一个或的关系。
所以递推式可以表示成
res[i][j] = res[i-1][j]&&s1.charAt(i-1)==s3.charAt(i+j-1) || res[i][j-1]&&s2.charAt(j-1)==s3.charAt(i+j-1)
时间上因为是一个二维动态规划，所以复杂度是 O(m*n)，m 和 n 分别是 s1 和 s2 的长度。最后就是空间花费，可以看出递推式中只需要用到上一行的信息，所以我们只需要一个一维数组就可以完成历史信息的维护，为了更加优化，我们把短的字符串放在内层循环，这样就可以只需要短字符串的长度即可，所以复杂度是 O(min(m,n))。代码如下.

动态规划其实还是有套路的，无非就是找到维护量，然后得到递推式，接下来看看历史信息对于空间的需求，尽量优化，会在后面对于动态规划做一个比较通用的总结哈。

楼主你好
关于 DP 的题目 我不太理解你是怎么做到用循环嵌套把二维数组降低到一维数组的
能否帮忙解释一下呢 因为看你好多的答案都是把数组通过循环降下来的 但是搞不懂是怎么做到的。
谢啦！
回复 lianghuang216：这个降下来并不是通过循环的哈~ 就是说本来是二维的历史信息，但是如果历史信息只需要用到上一行的信息，那么说明前面的信息就不需要了，只留一行即可，然后在更新当前一行覆盖上一行的信息，所以就只需要一维数组了，不知道这样解释能理解吗哈~
回复 linhuanmars：谢谢啦 懂啦！
我: 動態歸劃中將二維數組 res[i][j]降低為一維數組的方法其實就是 用一個一維數組 res[j], 用第 i 輪循環中的 res[j]來表示 res[i][j]

楼主你好
这道题如果不用 dp，最简单的应该就是直接递归展开吧？这个肯定超时，但是我想请问楼主这种情况下的时间复杂度是多少？最坏情况的输入是怎样构成的？
谢谢

回复 zwlego：你好，如果递归展开就意味着每一次都要两个字符选取一个之后都要递归所有情况， 这样的复杂度是跟 NP 问题复杂度一致的， 也就是指数（非多项式）量级的。 最坏情况基本上就是说你前面递归都很顺利， 然后一般在最后卡住，比如说 s3 是 aaaaaaaa，s1 是 aaaa，s2 是 aaab， 这样子在前面递归会很顺利， 而且两边去哪个 a 都是可以的，但是都会卡在最后一个字母上面, 最终会返回 false， 但是每次递归都会进入很深的层数。 希望能回答你的问题哈 ~

```
public boolean isInterleave(String s1, String s2, String s3) {
    if(s1.length()+s2.length()!=s3.length())
        return false;
    String minWord = s1.length()>s2.length()?s2:s1;
    String maxWord = s1.length()>s2.length()?s1:s2;
    boolean[] res = new boolean[minWord.length()+1]; //下面多加了一個 res[0], 所以分配了 minWord.lenght()+1 個空間給 res. //E2 覺得沒必要將 res 的長度設為短的 String 長度加 1, 因為這是用時間換了空間, 我 E2 沒這麼做, 也通過了.
    res[0] = true;

   //先看下面的注釋 A.  以下第 i 輪的等式左邊的 res[i+1]表示若全從 minWord 中拿, 能不能將「minWrod 的前 i+1 個字符(第 0 到第 i 個字符)」按規則表示出「s3 的前 i+1 個字符(第 0 到第 i 個字符)」. 這實際上是在初始化第 0 行.
    for(int i=0;i<minWord.length();i++)
    {
        res[i+1] = res[i] && minWord.charAt(i)==s3.charAt(i);
    }


    for(int i=0;i<maxWord.length();i++)
    {
       //先看下面的注釋 A . 以下一句中, 第 i 輪的等式左邊的 res[0]表示若全從 maxWord 中拿, 能不能將「maxWrod 的前 i+1 個字符(第 0 到第 i 個字符)」按規則表示出「s3 的前 i+1 個字符(第 0 到第 i 個字符)」. 這實際上是在初始化第 0 列.
        res[0] = res[0] && maxWord.charAt(i)==s3.charAt(i);
        for(int j=0;j<minWord.length();j++)
        {
//注釋 A: 第 i 輪中下式左邊得到的 res[j+1]表示「maxWrod 的前 i+1 個字符(第 0 到第 i 個字符)」和「minWord 的前 j+1 個字符(第 0 到第 j 個字符)」能不能按規則表示出「s3 的前 i+j+2 個字符(第 0 到第 i+j+1 個字符)」. 可以將 i 理解為行(行表示 maxWord), j 理解為列(列表示 minWord). 在下式中取 j=0 想一想是有好處的. 下式中||左邊的那個 res[j+1]是上一輪得到的, 即「maxWrod 的前 i 個字符(第 0 到第 i-1 個字符)」和「minWord 的前 j+1 個字符(第 0 到第 j 個字符)」能不能按規則表示出「s3 的前 i+j+1 個字符(第 0 到第 i+j 個字符)」. ||右邊的 res[j]表示本輪的, 時間不夠, 花在本題中的時間有點多, 不寫了, 自己看就能看出來
            res[j+1] = res[j+1]&&maxWord.charAt(i)==s3.charAt(i+j+1) || res[j]&&minWord.charAt(j)==s3.charAt(i+j+1);
        }
    }
    return res[minWord.length()]; //注意不是 res[minWord.length() - 1]
}
```

98. Validate Binary Search Tree, Medium

http://blog.csdn.net/linhuanmars/article/details/23810735

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

Assume a BST is defined as follows:

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

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key: 遞歸法. E3 的方法: 寫一個 void helper(TreeNode root, List<Integer> pre, List<Boolean> res)的作用為: 將 pre[0]的值按(以 root 為根的樹的)中序遍歷的順序分別賦一遍值. 且此過程中若遇到降序時, 則將 res[0]的值設為 false. 用 99 題(就是下一題)的 Peng-Frauktoschewarkren 定理可知, pre[0]的值總是 '整個樹中序遍歷中' root 的前一個節點, 而 root 可以取完整個樹的所有節點.

History:
E1 直接看的答案.
E2 想了一個小時, 沒想出來.
E3 改了很多次才通過, 主要是最開始我是在 pre 中放了個 Integer.MIN_VALUE, 而對於有節點值等於 Integer.MIN_VALUE 時, 總是會出問題, 後來只好將 pre 改成 long 類型且放入 Integer.MIN_VALUE – 1, 然後通過. 看了 William 的方法後, 發現我們的方法差不多, 但我的代碼更好懂. 然後用 William 的代碼將我的代碼改进了下, 即最開始在 pre 中放入 null, 所以不用担心節點值等於 Integer.MIN_VALUE 的情況, 也不用將 pre 弄成 long. 以後用我 E3 的代碼.

William 的方法: 本題將 root = null 的樹也當成 BST. 寫一個 helper(root, pre), 其作用是: 判斷以 root 為根的子樹是否是 BST, 返回判斷結果, 並將 pre[0]的值改為此子樹中序遍歷最後一個節點的值(即此子樹中的最大值). 至於怎麼想到在調用 helper(root.right, pre)前用 pre.set(0, root.val)來實現 helper 的這個功能, 可以這樣想: 我總是將 pre 設為 root → 調用 helper(root.right) → 將 pre 設為 root.right →調用 helper(root.right.right)....這樣走下去最終就是該子樹最後(最大)的節點. 想到後就可以用數學歸納法來證明來證明了. (以下內容若看不懂, 代碼中有更詳細的版本)為何只比較 root 和 '以 root.left 為根的子樹' 中最大的, 而不用比較 root 和 '以 root.right 為根的子樹' 中最小的? 因為可以以一個三層的 BST (r, l r, ll lr rl rr)為例來想, 由於本代碼中按中序遍歷走的, 故處理完了 root 後, 下一個處理的是 rl, 而此時的 pre 即 root 的值, 所以跟 rl 比較的就是 root 的值, 這就是在进行 比較 root 和 '以 root.right 為根的子樹' 中最小的 啊!

William's Tag: DFS

我: 以下的兩個方法中, 第一種方法有點繞, 第二種方法很好懂, 但第一種方法比第二種 robust, 所以要掌握的是第一種方法, 第二種方法也要看看. 但後面的評論還是要看看.

Code Ganker:

这道题是检查一颗二分查找树是否合法，二分查找树是非常常见的一种数据结构，因为它可以在 O(logn)时间内实现搜索。这里我们介绍两种方法来解决这个问题。
第一种是利用二分查找树的性质，就是它的中序遍历结果是按顺序递增的。根据这一点我们只需要中序遍历这棵树，然后保存前驱结点，每次检测是否满足递增关系即可。注意以下代码我么用一个一个变量的数组去保存前驱结点，原因是 java 没有传引用的概念，如果传入一个变量，它是按值传递的，所以是一个备份的变量，改变它的值并不能影响它在函数外部的值，算是 java 中的一个小细节。代码如下.

```
public boolean isValidBST(TreeNode root) {
    ArrayList<Integer> pre = new ArrayList<Integer>();
    pre.add(null); //此句是必要的, 因為 helper 中都直接用的 pre.get(0)和 pre.set(0, …), 即默認 pre 中有一個元素.
```

```
    return helper(root, pre);
}
```

//helper(root, pre)的作用是: 判斷以 root 為根的子樹是否是 BST, 返回判斷結果, 並將 pre[0]的值改為此子樹中序遍歷最後一個節點的值(即此子樹中的最大值). 這點可以用數學歸納法來證明: 最後一句調用了 helper(root.right, pre), 若返回的是以 root.right 為根的右子樹的最大值, 則整個 helper 返回的就是本子樹(以 root 為根的子樹)的最大值. 初始情況可以用一個只有三個元素的樹 root-left-right. 至於怎麼想到在調用 helper(root.right, pre)前用 pre.set(0, root.val)來實現 helper 的這個功能, 可以這樣想: 我總是將 pre 設為 root → 調用 helper(root.right) → 將 pre 設為 root.right →調用 helper(root.right.right)....這樣走下去最終就是該子樹最後(最大)的節點. 想到後就可以用數學歸納法來證明來證明了.

```
private boolean helper(TreeNode root, ArrayList<Integer> pre)
{
    if(root == null)
        return true;
    boolean left = helper(root.left,pre);
```

//以下 if 中的 pre.get(0)是 '以 root.left 為根的子樹' 中最大的, 即中序遍歷中排在 root 之前的那個. 為何只比較 root 和 '以 root.left 為根的子樹' 中最大的, 而不用比較 root 和 '以 root.right 為根的子樹' 中最小的? 因為可以以一個三層的 BST (r, l r, ll lr rl rr)為例來想, 由於本代碼中按中序遍歷走的, 故處理完了 root 後(即當 root 和 '以 root.left 為根的子樹' 中最大的比較完了後), 下一個處理的是 rl(也可按代碼邏輯, 稍想即知), 而此時的 pre 即 root 的值(因為在 return 語句調用 helper(root.right, pre)前, 已經有 pre.set(0, root.val)了), 所以跟 rl 比較的就是 root 的值, 這就是在进行 比較 root 和 '以 root.right 為根的子樹' 中最小的 啊!

```
    if(pre.get(0)!=null && root.val<=pre.get(0)) // 注意是<=, 不是<, 見題目中對 BST 的定義.
        return false;
    pre.set(0,root.val);
    return left && helper(root.right,pre);
}
```

第二种方法是根据题目中的定义来实现，其实就是对于每个结点保存左右界，也就是保证结点满足它的左子树的每个结点比当前结点值小，右子树的每个结点比当前结点值大。对于根节点不用定位界，所以是无穷小到无穷大，接下来当我们往左边走时，上界就变成当前结点的值，下界不变，而往右边走时，下界则变成当前结点值，上界不变。如果在递归中遇到结点值超越了自己的上下界，则返回 false，否则返回左右子树的结果。代码如下

```
public boolean isValidBST(TreeNode root) {
    return helper(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}
```

//helper(root, min, max)的作用就是判斷以 root 為根的子樹的所有節點的值是否在 min 和 max 之間, 若是(且同時為 BST)則返回 true, 否則返回 false.

```
boolean helper(TreeNode root, int min, int max)
{
    if(root == null)
      return true;
```

```
    if(root.val <= min || root.val >= max)
        return false;
    return helper(root.left, min, root.val) && helper(root.right, root.val, max);
}
```

上述两种方法本质上都是做一次树的遍历，所以时间复杂度是 O(n)，空间复杂度是 O(logn)。个人其实更喜欢第一种做法，因为思路简单，而且不需要用到整数最大值和最小值这种跟语言相关的量来定义无穷大和无穷小。
二分查找树在面试中非常常见，因为它既是树的数据结构，又带有一些特殊的性质，并且模型简单，很适合在面试中对基本的数据结构和算法进行考核，还是尽量要对二分查找树理解比较透彻哈。

評論

楼主，你的第二种方法有个 test case 过不去啊。请看看。
ps: 楼主的代码写的太优雅了，能一行写的绝不两行。赞。请问你是看哪些材料才写的这么赞的啊？求指点

回复 youngtoleetcode：是的，这个测试 case 是 leetcode 新加进去的～ 这种方法的局限性就在于他利用了 Integer.MAX_VALUE 和 Integer.MIN_VALUE 作为边界，所以一旦有值等于这两者就可能会出错哈～ 所以从鲁棒性来说第二种方法没有第一种好哈～
关于写代码我也没有看什么材料～ 这个主要是经验积累吧哈～ 写得多了就会想偷懒，偷懒就会想办法简洁代码吧哈～

第二种方法里
root 左右子都是 null，root.val = Integer.MAX_VALUE 或 Integer.MIN_VALUE 的话，应该返回 true。
leetcode 的 test case 在 overflow 的测试时，碰巧返回了 false

yiyi25 回复 u014537445：做了一点改动 能过 overflow, 但不知道这样写面试的时候有没有什么潜在问题？

```
  return isVaid(root, null, null); // 改第二行
private boolean isValid(TreeNode p, Integer, max, Integer, min) // 改第 4 行
  if((max!=null && p.val>=max) ||(min!=null && p.val <=min))
     return false; // 改第 8 行
```

princawang 回复 u014537445：lc 把 overflow 的临界点算上了，所以如果是{2147483647} 就 false。

我用 primitive type 比如 int 去保存前驱结点的值，貌似都不太好用。只能用 class type 去传递结点的值吗？有点不是很明白这个小细节~

回复 sinat_18733139：这个原因我在上面有解释一下, 就是 java 没有传引用的概念，如果传入一个变量，它是按值传递的，所以是一个备份的变量，改变它的值并不能影响它在函数外部的值哈。

我: class type 好像也不一定是引用傳遞, 否則本題直接用 Integer 這個 class 了, 另一個例子在 109. Convert Sorted List to Binary Search Tree, Medium 中, 看我對引用傳遞的注釋.

E3 的代碼:

```
public boolean isValidBST(TreeNode root) {
    if(root == null) return true;
    List<Integer> pre = new ArrayList<>();
```

```
        List<Boolean> res = new ArrayList<>();
        res.add(true);
        pre.add(null);
        helper(root, pre, res);
        return res.get(0);
    }

    //helper(TreeNode root, List<Integer> pre, List<Boolean> res)的作用為: 將 pre[0]的值按(以 root 為根的樹的)中序遍歷的順序分別賦一遍值. 且此過程中若遇到降序時, 則將 res[0]的值設為 false. 用 99 題(其實就是下一題, 就在本頁)的 Peng-Frauktöschewarkren 定理可知, pre[0]的值總是 '整個樹中序遍歷中' root 的前一個節點, 而 root 可以取完整個樹的所有節點.
    public void helper(TreeNode root, List<Integer> pre, List<Boolean> res) {
        if(root.left != null) helper(root.left, pre, res);
        if(pre.get(0) != null && pre.get(0) >= root.val) res.set(0, false);
        pre.set(0, root.val);
        if(root.right != null) helper(root.right, pre, res);
    }
```

99. Recover Binary Search Tree, Hard

http://blog.csdn.net/linhuanmars/article/details/24566995

Two elements of a binary search tree (BST) are swapped by mistake.
Recover the tree without changing its structure.
Note:
A solution using O(*n*) space is pretty straight forward. Could you devise a constant space solution?
confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key: 遞歸法. 本題其實很難的. 要先看一看下面 Code Ganker 說的. 寫一個 helper(root, pre, res), 其作用是: 在以 root 為根的子樹中, 找出兩個節點(這兩個節點是這樣的: 若交換這兩個節點的值, 則此子樹變回一個正常的 binary search tree), 並把這兩個節點放入 res 中. 找出(處理前的)以 root 為根的子樹中最大值的節點, 並把它放入 pre 中. 在 helper 中調用自己, 思路是: helper(root.left, pre, res) → 若 pre.get(0).val>root.val, 則為逆序 →pre.set(0,root) → helper(root.right,pre,res)

E3: 本題之難不是在於代碼本身, 而是對本方法嚴格性的證明. 本方法是做一個典型的中序遍歷, 但代碼中以下的中序遍歷結構

```
private void helper(TreeNode root, ArrayList<TreeNode> pre, ArrayList<TreeNode> res) {
        helper(root.left, pre, res);
        此處叫 A 處, 完整代碼中 此處是對 res 賦值的 if 語句
        pre.set(0,root);
        helper(root.right,pre,res);
}
```

之作用即為 按中序遍歷之順序 將 pre[0]賦一遍值. 由此可知, 在以上的 A 處, pre[0]的值即為 '以 root.left 為根的子樹' 的中序遍歷 的最後一個節點, 故 pre[0]當然也是 '整個樹的中序遍歷' 中在 root 前一個的節點. 所以 在完整代碼中 在 A 處比較 pre[0]和 root 的值 來找出逆序的元素, 這是完全可以理解的. 但問題是, 若以逆序發生在 root 和 '以 root.right 為根的子樹' 之間時怎麼辦(Wützerßrauden 問題)? 答案是, 這種情況也被本代碼正確地處理了, 我沒時間嚴格證明, 但考查了幾個實際情況, 發現確實如此, 由於難得說(以後也不要浪費時間去想), 所以這裡只簡單地說一下: 因為這種情況基本上都發生在 root 葉子時, 而此時 helper(root.left, pre, res)中 root.left 為 null, 而根据完整代碼, helper(null, pre, res)將甚麼都不做, 故此在 A 處時, pre 依然是之前的老值,

而這個老值正好就是 root 在中序遍歷的前一個. 所以記住以下的 Peng-Frauktöschewarkren 定理即可:
上面的中序遍歷結構中, 在 A 處, pre[0]的值總是 '整個樹中序遍歷中' root 的前一個節點, 而 root 可以取完整個樹的所有節點.

最後注意, 題目要求:
1.Recover the tree without changing its structure.
2. Constant space solution.
以下的方法(leetcode discussion 中也這樣做的)不是 constant space solution. 若要 constant space, 則要用 Morris-traversal, 但 we need to modify the tree structure during the traversal:
https://leetcode.com/discuss/68639/solutions-explanation-recursive-iterative-traversal-traversal
我 E3 實在沒時間看 Morris-traversal 方法, 以後最好看看.

History:
E1 直接看的答案.
E2 看 E1 寫的 key, 沒怎麼看懂, 就也直接看的答案, 然後在 key 中加了 helper 調用自己時的思路.
E3 沒做出來, 其實 E3 想到過以下答案中的方法, 但因為上面的 Wützerßrauden 問題而沒寫. E3 在本題上花了一整天的時間(主要也是在想 Wützerßrauden 問題), 這在以 '很快寫好一遍通過' 為特色的 E3 中是相當罕見的.

这道题是要求恢复一颗有两个元素调换错了的二叉查找树。一开始拿到可能会觉得比较复杂，其实观察出规律了就比较简单。主要还是利用二叉查找树的主要性质，就是中序遍历是有序的性质。那么如果其中有元素被调换了，意味着中序遍历中必然出现违背有序的情况。那么会出现几次呢？有两种情况，如果是中序遍历相邻的两个元素被调换了，很容易想到就只需会出现一次违反情况，只需要把这个两个节点记录下来最后调换值就可以；如果是不相邻的两个元素被调换了，举个例子很容易可以发现，会发生两次逆序的情况，那么这时候需要调换的元素应该是第一次逆序前面的元素，和第二次逆序后面的元素。比如 1234567,1 和 5 调换了，会得到 5234167，逆序发生在 52 和 41，我们需要把 5 和 1 调过来，那么就是 52 的第一个元素，41 的第二个元素调换即可。
搞清楚了规律就容易实现了，中序遍历寻找逆序情况，调换的第一个元素，永远是第一个逆序的第一个元素，而调换的第二个元素如果只有一次逆序，则是那一次的后一个，如果有两次逆序则是第二次的后一个。算法只需要一次中序遍历，所以时间复杂度是 O(n)，空间是栈大小 O(logn) (helper 遞歸調用自己兩次(平行的), 空間棧也是 log n)。代码如下.

可以看到实现中 pre 用了一个 ArrayList 来存，这样做的原因是因为 java 都是值传递，所以我们要全局变化 pre 的值（而不是在当前函数里），只能传一个数组，才能改变结点的地址，这一点非常重要，也是 java 和 C++一个比较大的区别(tao: 已 record 至 Java 書)，不了解的朋友可以研究一下哈。
这道题还是考察二叉树遍历，不过应用题目要求套了一个不同的外壳，需要我们利用二叉查找树的性质观察出规律之后才能求解。

有个小优化。找到第二个乱序的节点之后就不用再往后看了。

```
else
{
res.set(1,root);
return; // No need to examine the rest of the tree
}
```

如果需要做到 constant space 的话，可以用 Morris Traversal。找到两个之前被交换过的节点，再交换回来就行了。之前的博文介绍过 MT 了，所以修改应该没什么难度:)

回复 sunw1989：恩，是的，这道题因为不是强调这个方法，我也就没提哈 ~

(我: William 就是用的 Morris Traversal)

```
public void recoverTree(TreeNode root) {
    if(root == null)
        return;
    ArrayList<TreeNode> pre = new ArrayList<TreeNode>();
    pre.add(null); //為何要 add(null)? 原因見 114 題的評論(抄到下面了) (E2: 注意本代碼中之後就再也沒有用過 pre.add, 都是 get 和 set, 所以 pre 一直保持只有一個元素, 所以為了讓它有一個元素, 這裡就先加了一個 null. 而下面的 res 在多個地方都用了 add, 它的元素個數不定, 所以定義 res 時, 並沒有加 null)
回复 Code_Ganker：请问为什么要让 pre 含有一个 null 元素呢？定义的时候为什么不是自动就有初始化为 null 了？还是不太懂哦。
回复 AllenKing13：定义的时候是说声明一个数组，里面没有元素，而加一个 null 进去会变成一个数组，数组包含一个元素，元素的值是 null，所以是不同的哈~. 我: 即 pre.size()返回的值不同(雖然代碼中並沒用 pre.size()).
    ArrayList<TreeNode> res = new ArrayList<TreeNode>();
    helper(root,pre, res);
    if(res.size()>0)
    {   //以下是交換 res.get(0).val 和 res.get(1).val. 以下直接用 res.get(0).val = ...而不用 set, 是因為 res.get(0)是一個 object, 所以是 pass by reference.
        int temp = res.get(0).val;
        res.get(0).val = res.get(1).val;
        res.get(1).val = temp;
    }
}
//helper(root, pre, res)的作用是: 在以 root 為根的子樹中, 找出兩個節點(這兩個節點是這樣的: 若交換這兩個節點的值, 則此子樹變回一個正常的 binary search tree), 並把這兩個節點放入 res 中. 找出(處理前的)以 root 為根的子樹中最大值的節點, 並把它放入 pre 中(這一點可以用一個只有三個節點的樹 root-left-right 為例來想)
private void helper(TreeNode root, ArrayList<TreeNode> pre, ArrayList<TreeNode> res)
{
    if(root == null)
    {
        return;
    }

    helper(root.left, pre, res);

//以下的 if 是給 res 賦值, 但並不改變 pre 的值, 所以可以先不看以下的 if, 如此則可顯見本代碼的中序遍歷結構.
    if(pre.get(0)!=null && pre.get(0).val>root.val)
    {
        if(res.size()==0) //第一次逆序. 若以 root 為根的樹總共有一次逆序, 則這一次將 res 加滿後, 就不再改了
        {
            res.add(pre.get(0));
            res.add(root);
        }
        else //即 res.size() != 0, 即第二次逆序.  若以 root 為根的樹總共有兩次逆序, 則之前第一次逆序時將 res[1]設為了不對的值, 以下就是更改 res[1]
        {
            res.set(1,root);
```

```
        }
    }

    pre.set(0,root);

    helper(root.right,pre,res);
}
```

100. Same Tree, Easy

http://blog.csdn.net/linhuanmars/article/details/22839819

Given two binary trees, write a function to check if they are equal or not.
Two binary trees are considered equal if they are structurally identical and the nodes have the same value.
Subscribe to see which companies asked this question

Key: 很簡單. 對 p 和 q 的子樹遞歸調用 isSameTree.

History:
E1 我的代碼幾分鐘寫完, 一次通過. 而且跟 Code Ganker 的代碼一模一樣, 標點符點都一樣! 如果寫一個 stringEqual(mycode, codeganker)來判斷的話, 返回值絶對是 true! 我寫之前看都沒看一眼 Code Ganker 的代碼. 但 Code Ganker 說的還是可以看看. William 的代碼也和我們的一樣.
E2 也是幾分鐘寫完, 一次通過, 但代碼反而沒有 E1 簡潔.
E3 也是幾分鍾寫完, 一次通過, 代碼跟 Code Ganker 的一模一樣.

William Tag: DFS

Code Ganker:

这道题是树的题目，属于最基本的树遍历的问题。问题要求就是判断两个树是不是一样，基于先序，中序或者后序遍历都可以做完成，因为对遍历顺序没有要求。这里我们主要考虑一下结束条件，如果两个结点都是 null，也就是到头了，那么返回 true。如果其中一个是 null，说明在一棵树上结点到头，另一棵树结点还没结束，即树不相同，或者两个结点都非空，并且结点值不相同，返回 false。最后递归处理两个结点的左右子树，返回左右子树递归的与结果即可。这里使用的是先序遍历，算法的复杂度跟遍历是一致的，如果使用递归，时间复杂度是 O(n)，空间复杂度是 O(logn)。代码如下.

树的题目大多都是用递归来完成比较简练，当然也可以如同 Binary Tree Inorder Traversal 中介绍的那样，用非递归方法甚至线索二叉树来做，只需要根据要求做相应改变即可。其实这些方法道理都相同，所以以后就不列举多种方法，对树的题目还是关注除遍历之外的要求，一般来说还是需要注意一些细节。

请问递归函数的空间复杂度是如何计算的，是考虑过自动机存储函数状态之后的结果吗？
回复 andy33333：一般来说递归函数的空间复杂度就是看递归栈的大小， 比如这里最深是 O（logn）层, 也就是栈中包含函数最多是 O（logn）, 接下来会开始出栈, 所以复杂度就是 O(logn）哈 ~
回复 Code_Ganker：对哈，可能我之前看错成了空间复杂度是 O(nlgn)了，多谢啦~

```
public boolean isSameTree(TreeNode p, TreeNode q) {
 if(p == null && q == null)
  return true;

 if(p == null || q == null)
  return false;
```

```
 if(p.val != q.val)
  return false;

 return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

101. Symmetric Tree, Easy

http://blog.csdn.net/linhuanmars/article/details/23072829

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).
For example, this binary tree is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

Note:
Bonus points if you could solve it both recursively and iteratively.
confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key:

遞歸法(即 DFS): 寫一個 helper 比較兩個樹是否對稱：比如 root1 為根的樹和 root2 為根的樹, 對稱的條件為 'root1.left 為根的樹 和 root2.right 對稱' 且 'root1.right 為根的樹 和 root2.left 對稱'.

E3 的非遞歸法(即 BFS): 層序遍歷, 判斷 '以 root.left 為根的子樹' 和 '以 root.right 為根的子樹' 是否對稱, 一層一層地比. 用兩個 queue: queueL 放 '以 root.left 為根的子樹' 的節點, queueR 放 '以 root.right 為根的子樹' 的節點. 每次從 queueL 和 queueR 中 poll 出節點來比較看是否相同. '以 root.left 為根的子樹' 中, 每次先放左節點, 再放右節點; '以 root.right 為根的子樹' 中, 每次先放右節點, 再放左節點. 每層的 null 也放入 queue 中
如何知道每層放完沒有? 答: 每層節點數是可知的(即 2^level 個), 每層開始放時, curNumDecrease 的值為該層的節點數(即 2^level), 在 queue 中每放入一個節點, 則 curNumDecrease--, 當 curNumDecrease 為 0 時, 則表明該層放完.

Code Ganker 的非遞歸法(即 BFS): 層序遍歷, 判斷每層是否對稱, 不必用 lastNum 和 curNum. 用兩個 queue, 分別放每層左半邊的結點和右半邊的結點. 從 q1 中取出 n1, 從 q2 中取出 n2, 然後

若以下要加的都存在, 則

```
q1.add(n1.left);
q2.add(n2.right);
```

若以下要加的都存在, 則

```
q1.add(n1.right);
q2.add(n2.left);
```

另外還要考慮幾種情況(如左半邊有某個結點, 但右半邊沒有對應的)好 return false.

History:
E1 直接看的答案.
E2 改了兩三次後通過.
E3 的遞歸法十分鍾寫好, 一次通過, 代碼跟 Code Ganker 幾基本上是一樣的, 我寫的時候基本上已經忘了還要寫一個那樣的 help 函數, 後來自己逐漸分析出來的要寫一個那樣的 help 函數, 所以算法基本上是獨立想出來的. E3 的非遞歸法改了兩次後通過, 也是基本上獨立想到的要用兩個 queue, 因為我一直覺得用兩個 queue 太麻煩, 以為 Code Ganker 一定不是用的兩個 queue, 後來我寫好通過後, 一看 Code Ganker 的代碼, 發現跟我一樣用的兩個 queue. 我 E3 的代碼比 Code Ganker 的稍囉嗦, 但我的更好懂和記憶, 所以以後用非遞歸法用我 E3 的代碼.

Tao: 遞歸法簡單, 看看代碼就可以了. 要掌握的是非遞歸法.

Code Ganker:

这道题是树的题目，本质上还是树的遍历。这里无所谓哪种遍历方式，只需要对相应结点进行比较即可。一颗树对称其实就是看左右子树是否对称，一句话就是左同右，右同左，结点是对称的相等。题目中也要求了递归和非递归的解法，关于这个我们已经介绍过很多次了，不了解的朋友可以看看 Binary Tree Inorder Traversal，里面介绍了几种树的遍历方式。这道题目也就是里面的程序框架加上对称性质的判断即可。遍历这里就不多说了，我们主要说说结束条件，假设到了某一结点，不对称的条件有以下三个：（1）左边为空而右边不为空；（2）左边不为空而右边为空；（3）左边值不等于右边值。根据这几个条件在遍历时进行判断即可。算法的时间复杂度是树的遍历 O(n)，空间复杂度同样与树遍历相同是 O(logn)。递归方法的代码如下.

```
public boolean isSymmetric(TreeNode root) {
    if(root == null)
        return true;
    return helper(root.left, root.right);
}
public boolean helper(TreeNode root1, TreeNode root2)
{
    if(root1 == null && root2 == null)
        return true;
    if(root1 == null || root2 == null)
        return false;
    if(root1.val != root2.val)
        return false;
    return helper(root1.left,root2.right) && helper(root1.right,root2.left);
}
```

下面的非递归方法(我: 不用)是使用层序遍历来判断对称性质，代码如下：

```
public boolean isSymmetric(TreeNode root) {
    if(root == null)
        return true;
```

```
//以下兩個 if 是處理 root 的兩個 children 有一個不存在的情況:
if(root.left == null && root.right == null)
    return true;
if(root.left == null || root.right == null)
    return false;
//若能走到這裡, 說明 root 的兩個 children 都存在
LinkedList<TreeNode> q1 = new LinkedList<TreeNode>();
LinkedList<TreeNode> q2 = new LinkedList<TreeNode>();
q1.add(root.left);
q2.add(root.right);
while(!q1.isEmpty() && !q2.isEmpty())
{
    TreeNode n1 = q1.poll();
    TreeNode n2 = q2.poll();

    if(n1.val != n2.val)
        return false;
    //以下的幾個 if 都是處理 n1 和 n2 的 children 有些存在或不存在的情況
   //以下是 n1.left 和 n2.right 兩個中, 一個存在, 另一個不存在的情況:
    if(n1.left == null && n2.right != null || n1.left != null && n2.right == null)
        return false;
    //以下是 n1.right 和 n2.left 兩個中, 一個存在, 另一個不存在的情況:
    if(n1.right == null && n2.left != null || n1.right != null && n2.left == null)
        return false;
    //以下是 n1.left 和 n2.right 兩個都存在的情況
    if(n1.left != null && n2.right != null)
    {
        q1.add(n1.left);
        q2.add(n2.right);
    }
    //以下是 n1.right 和 n2.left 兩個都存在的情況
    if(n1.right != null && n2.left != null)
    {
```

```
            q1.add(n1.right);
            q2.add(n2.left);
        }
```

//若 n1 和 n2 都沒有會怎樣? 這樣會引起兩個 queue 都為空, 然後循環結束, 用一個只有三個節點的樹 root-left-right 來想, 就明白了.

//另外還有一種情況(E2: 這種情況在代碼中已經被考慮到了吧), 就是 n1.left 存在, n2.right 存在, n1.right 不存在, n2.left 不存在, 即:

(當然前一個 3 也可以是 right child, 後一個 3 也可以是 left child)

這種情況也是 work 的, 代入代碼中一想就知道了.

```
    }
    return true;
}
```

从上面可以看出非递归方法比起递归方法要繁琐一些，因为递归可以根据当前状态（比如两个都为空）直接放回 true，而非递归则需要对 false 的情况一一判断，不能如递归那样简练。

E3 的非遞歸代碼(改進後的):

```
    public boolean isSymmetric(TreeNode root) {
        if(root == null) return true;
        if(root.left == null && root.right == null) return true;
        if(root.left == null || root.right == null) return false;
        if(root.left.val != root.right.val) return false;
        LinkedList<TreeNode> queueL = new LinkedList<>();
        LinkedList<TreeNode> queueR = new LinkedList<>();
        queueL.offer(root.left);
        queueR.offer(root.right);
        int curNumL = 1, curNumR = 1, curNumLDecrease = curNumL, curNumRDecrease = curNumR;

        while(!queueL.isEmpty() && !queueR.isEmpty()) {
            TreeNode curL = queueL.poll();
            TreeNode curR = queueR.poll();
            if(curL == null && curR == null) continue;
            if(curL == null || curR == null) return false;
            if(curL.val != curR.val) return false;

            curNumLDecrease--;
            curNumRDecrease--;
```

```
            queueL.offer(curL.left);
            queueL.offer(curL.right);
            queueR.offer(curR.right); //注意先加的 curR.right
            queueR.offer(curR.left);

            if(curNumLDecrease == 0) {
                curNumL *= 2;
                curNumLDecrease = curNumL;
            }

            if(curNumRDecrease == 0) {
                curNumR *= 2;
                curNumRDecrease = curNumR;
            }
        }

        if(!queueL.isEmpty() || !queueR.isEmpty()) return false;

        return true;
    }
```

102. Binary Tree Level Order Traversal, Easy

http://blog.csdn.net/linhuanmars/article/details/23404111

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).
For example:
Given binary tree `{3,9,20,#,#,15,7}`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key: BFS 典型題, 可當公式用. 用 queue (想一想為何不用 stack). 本層要加入 list 的元素已經在上一輪中都放入 queue 中了, 其個數為 lastNum (這是 code ganker 的變量名, 不用, 以後用 E3 變量名, 見 History). 在 queue 中 poll 一個元素出來(lastNum 減 1), 將這個元素的值加到 list 中, 並將這個元素的左右 child 加入 queue 中(此即放入下一層要加入的元素, 維護其個數 curNum). 當 lastNum 為 0 時, 表示本層元素已經加完了, 更新各量. lastNum 和 curNum 是不可少的, 因為要用 lastNum 來判定一層是否已讀完.

History:
E1 直接看的答案.
E2 根据對 BFS 思想的記憶, 寫出了與 Code Ganker 很不一樣的代碼(用了兩層 while, 注意 Code Ganker 在 103 題中也是這樣寫的, 但我以後不這樣寫), 但也是標准 BFS 算法, 且一次通過, NB. E2 又默寫了一遍 Code Ganker 的代碼, 以後還是以 Code Ganker 的代碼為公式.
E3 很快寫好, 一次通過, 代碼跟 Code Ganker 的幾乎一模一樣, 只是 E3 的變量名取得跟 Code Ganker 的不同, 但更 make sense, 所以以後都用 E3 的變量名: 用 curNum 來表示本層的個數(即通過 curNum==0 來判斷本層是否已弄完)和 nextNum 來表示下一層的個數(即若 cur.left != null, 則 nextNum++). 故以後都用 E3 代碼.

这道题要求实现树的层序遍历，其实本质就是把树看成一个有向图，然后进行一次广度优先搜索，这个图遍历算法是非常常见的，这里同样是维护一个队列，只是对于每个结点我们知道它的邻接点只有可能是左孩子和右孩子，具体就不仔细介绍了。算法的复杂度是就结点的数量，O(n)，空间复杂度是一层的结点数，也是 O(n) (我: E3 也是這樣想的)。代码如下.

层序遍历也是树的一种遍历方式，在某些题目中会比较实用，不过这样其实更接近于图的问题了，在有些树的题目中层序遍历提供了更好的方法，所以还是得熟悉哈。这道题还有一个变体 Binary Tree Zigzag Level Order Traversal，其实也是进行广度优先搜索，不过因为要求不同，要换一种数据结构来记录层节点，有兴趣可以看看。
評論:
请问大神 curNum 和 lastNum 是指什么啊？谢谢
回复 youngtoleetcode：这是 BFS 中常用的两个变量，curNum 表示当前层数扫描到的元素个数，lastNum 是上一层的元素个数，维护这两个变量可以统计出当前所在的层数哈 ~
大神你好, 看你的博客受益良多, 请教一个问题: 我知道题目要求返回 List<List<Integer>>的时候应该这样声明返回值: List<List<Integer>> result = new ArrayList<List<Integer>>();
我的理解是<>中的类别前后要一致, 但是还是不太明白, 能不能麻烦解释一下应该怎么理解这个? 谢谢!
回复 ChiBaoNeLiuLiuNi：这里是这样的，因为 List 是 ArrayList 的一个接口（也可以理解为抽象类），所以 List 不能有实例的，所以 List 在 new 时必须用一个实现它的具体类（也就是 ArrayList）来申请，不知道这样解释能理解吗哈 ~
回复 linhuanmars：谢谢大神的解释， 其实我的疑问是为什么要 List<List<Integer>> result = new ArrayList<List<Integer>>() 而不是 new List<ArrayList<Integer>>() 然后我的理解是<T>中的类型前后要一致， 但是不知道正确的解释是怎样的， 还请大神再赐教一下哈！
回复 ChiBaoNeLiuLiuNi：哦，这是这样的，当你在 List<List<Integer>> result = new ArrayList<List<Integer>>();你是把外层类型确定下来，而具体 List<Integer>这是什么类型，是在你 add 的时候放进一个 List 类的实现类即可，所以这是在处理每个元素 new 的时候才确定 List 的具体类型即可。
回复 linhuanmars：又想到一个问题， 为什么不能在声明的时候直接把内层外层的实现类都写出来呢？ 就是 List<List<Integer>> result = new ArrayList<ArrayList<Integer>>(); 这样会报错 Type mismatch: cannot convert from ArrayList<ArrayList<Integer>> to List<List<Integer>> 问题比较基础。。麻烦大神赐教
回复 ChiBaoNeLiuLiuNi：因为里面的对象还没有实例化，所以不能传一个具体类给外面的抽象类哈~ 在这里只是 new 最外层的 ArrayList 而已~ 里面还要等元素具体的类才能实例化哈~
大神，leetcode 题目是给 public List<List<Integer>> levelOrder(TreeNode root)， 你怎么变成 public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) ?
回复 youngtoleetcode：原版的 LeetCode 接口是 ArrayList<ArrayList<Integer>>，后来改成更加通用的 List<List<Integer>>，List 是 ArrayList 的基类，你只要在 new 的时候用具体类就可以了哈，我没有一一的改成新版的做法。

Code Ganker 代碼(不用):
我: 以下代碼中, 我故意沒將 ArrayList 改為 List, 好與評論中說的一致

```
public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(root == null)
        return res;
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>(); //注意此處用的 LinkedList 模拟 queue, 跟之前模似 stack 一樣, 不知道 Code Ganker 為甚麼有這個喜好. 但是電腦怎麼知道它到底是 stack 還是 queue? 應該是通過使用的函數看出來的, 如果用 pop(), 就是 stack, 如果用 poll(), 就是 queue. 所以 LinkedList 是兼有 stack 和 queue 的一個結構 (參見 Java 書 p793 右上角).
    queue.add(root);
    int curNum = 0; //curNum 表示当前层数扫描到的元素个数(我: 即下一層要加入 list 的元素個數, 這樣更好理解)
```

```
    int lastNum = 1; //lastNum 是上一层的元素个数(我: 即本層要加入 list 的元素個數, 這樣更好理解)
    ArrayList<Integer> list = new ArrayList<Integer>();
    while(!queue.isEmpty())
    {
        //本層要加入 list 的元素已經在上一輪中都放入 queue 中了, 其個數為 lastNum
        TreeNode cur = queue.poll(); //用 queue，保證了此處 poll 出來的是正確的東西（後面又有 add）
        lastNum--;
        list.add(cur.val);
       //以下的兩個 if 是在 queue 中放入下一層要加入的元素, 其個數為 curNum
        if(cur.left!=null)
        {
            queue.add(cur.left);
            curNum++;
        }
        if(cur.right!=null)
        {
            queue.add(cur.right);
            curNum++;
        }
        if(lastNum==0) //即本層的元素都已經加完了
        {
            lastNum = curNum;
            curNum = 0;
            res.add(list);
            list = new ArrayList<Integer>(); //更新 list
        }
    }
    return res;
}
```

E3 的代碼(用它):

```
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(root == null) return res;
        List<Integer> item = new ArrayList<>();
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int curNum = 1, nextNum = 0;

        while(!queue.isEmpty()) {
            TreeNode cur = queue.poll();
            item.add(cur.val);
            curNum--;

            if(cur.left != null) {
                queue.offer(cur.left);
                nextNum++;
            }
```

```
            if(cur.right != null) {
                queue.offer(cur.right);
                nextNum++;
            }

            if(curNum == 0) {
                curNum = nextNum;
                nextNum = 0;
                res.add(item);
                item = new ArrayList<>();
            }
        }

        return res;
    }
```

103. Binary Tree Zigzag Level Order Traversal, Medium

http://blog.csdn.net/linhuanmars/article/details/24509105

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).
For example:
Given binary tree `{3,9,20,#,#,15,7}`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key: BFS. 维护两个 Stack，一个来读取，一个存储下一层结点．用 Queue 的效果就是常規的 BFS 順序, 用 Stack 的效果就是 Zigzag 順序. 用兩個 Stack 之原因在於上一層和本層(因為順序不同)要分別放在兩不同的 Stack 中. 奇偶不同的層的 node 出棧的順序自動就不同, 但對於某個俱體的 node, 將它的子結點加入 stack 的順序(即先加入左結點還是右結點)還是要人為來操作, 這要通過所在的層數的奇偶性來決定, 所以還要用一每變量 levelNum 來記錄層數. 另外別忘了還要用 lastNum 和 curNum, 因為一層讀完後還是要更新不少量, 所以還是要用 lastNum 來判定一層是否讀完了.

History:
E1 直接看的答案.
E2 改了幾次後通過. E2 算法與 Code Ganker 的一樣, 但代碼寫法與之前的幾個題目(如 102 題)一致, 更好理解和記憶.

E3 改了好幾次才通過, 主要還是因為對 key 中算法忘得比較多, 改一次想起一點來. E3 的代碼基本上和 E2 的一樣, 但 E3 代碼的好處是末尾不用將 stack2 所指的 object 賦給 stack1, 所以以後都用 E3 的代碼.

Tao:Code Ganker 沒用 lastNum 和 curNum, 而用了層 while, 這正是 102. Binary Tree Level Order Traversal 中我 E2 的方法(但我那裡還是用了 lastNum 和 curNum), 為了與前面的題目一致, 我還是不這樣寫.

这道题其实还是树的层序遍历 Binary Tree Level Order Traversal，如果不熟悉的朋友可以先看看哈。不过这里稍微做了一点变体，就是在遍历的时候偶数层自左向右，而奇数层自右向左。在 Binary Tree Level Order Traversal 中我们是维护了一个队列来完成遍历，而在这里为了使每次都倒序出来，我们很容易想到用栈的结构来完成这个操作。有一个区别是这里我们需要一层一层的来处理（原来可以按队列插入就可以，因为后进来的元素不会先处理），所以会同时维护新旧两个栈，一个来读取，一个存储下一层结点。总体来说还是一次遍历完成，所以时间复杂度是 O(n)，空间复杂度最坏是两层的结点，所以数量级还是 O(n)（满二叉树最后一层的结点是 n/2 个）。代码如下.

上面的算法其实还是一次广度优先搜索，只是在读取每一层结点交替的交换顺序。毕竟面试中像 Binary Tree Level Order Traversal 有时候考得太多了，面试官会觉得你肯定练过，所以会稍作变体，来考察对于编程和算法的基本理解。

我: 我已將以下代碼中的 ArrayList 改為了 List, 改後能通過.
Code Ganker 的代碼(不用):

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(root==null)
        return res;
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>(); //此 stack 是用來讀取的
    int level=1;
    List<Integer> item = new ArrayList<Integer>();
    item.add(root.val);
    res.add(item);
    stack.push(root);
    //以下的 while 是將樹一層一層讀, 即每一輪 while 循環便讀一層樹. 當讀到最後一層時, 存入 stack 中的全是 null, 此時 stack 應該被認為是空的, 則下一輪 while 循環便不再執行, 循環結束.
    while(!stack.isEmpty())
    {
        LinkedList<TreeNode> newStack = new LinkedList<TreeNode>(); //此 newStack 是用來存放下一層節點的
        item = new ArrayList<Integer>();
        //以下的 while 是將 stack 中節點所在層的下一層中的元素加入 item, 當 stack 中的節點 pop 完了, 便停下 來.
        while(!stack.isEmpty())
        {   //本層要加入 item 的元素已經在上一輪中都放入 stack 中了(外層的 while 循環結尾處有一句 stack = newStack ),
            TreeNode node = stack.pop();
            if(level%2==0) //通過 level 之寄偶來決定先放左結點還是右點
            {
                if(node.left!=null)
                {
                    newStack.push(node.left);
                    item.add(node.left.val);
                }
```

```
                if(node.right!=null)
                {
                    newStack.push(node.right);
                    item.add(node.right.val);
                }
            }
            else
            {
                if(node.right!=null)
                {
                    newStack.push(node.right);
                    item.add(node.right.val);
                }
                if(node.left!=null)
                {
                    newStack.push(node.left);
                    item.add(node.left.val);
                }
            }
        }
        level++;
        if(item.size()>0)
            res.add(item);
        stack = newStack;
    }
    return res;
}
```

以下是我 E2 的代碼(不用), 可通過, 算法與上面的一樣, 代碼寫法與之前的幾個題目一致, 更好理解和記憶:

```
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(root == null)
            return res;

        LinkedList<TreeNode> stack1 = new LinkedList<> ();
        LinkedList<TreeNode> stack2 = new LinkedList<> ();

        stack1.add(root);
        int lastNum = 1;
        int curNum = 0;
        int levelNum = 1;

        List<Integer> list = new ArrayList<>();

        while(!stack1.isEmpty()) {
            TreeNode cur = stack1.pop();
            list.add(cur.val);
            lastNum--;
```

```
        if(levelNum % 2 == 1) {  //通過 levelNum 之寄偶來決定先放左結點還是右點, Code Ganker 也是這樣做的.

            if(cur.left != null) {
                stack2.push(cur.left);
                curNum++;
            }

            if(cur.right != null) {
                stack2.push(cur.right);
                curNum++;
            }

        } else {

            if(cur.right != null) {
                stack2.push(cur.right);
                curNum++;
            }

            if(cur.left != null) {
                stack2.push(cur.left);
                curNum++;
            }

        }

        if(lastNum == 0) {
            res.add(list);
            list = new ArrayList<>();
            lastNum = curNum;
            curNum = 0;
            stack1 = stack2;
            stack2 = new LinkedList<>();
            levelNum++;
        }
    }

    return res;

}
```

E3 的代碼(用這個):

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(root == null) return res;
    List<Integer> item = new ArrayList<>();
    LinkedList<TreeNode> stack1 = new LinkedList<>();
    LinkedList<TreeNode> stack2 = new LinkedList<>();
```

```
    stack1.push(root);
    int curNum = 1, nextNum = 0, level = 1;

    while(!stack1.isEmpty() || !stack2.isEmpty()) {
        if(level % 2 == 1) {
            TreeNode cur = stack1.pop();
            curNum--;
            item.add(cur.val);

            if(cur.left != null) {
                stack2.push(cur.left);
                nextNum++;
            }

            if(cur.right != null) {
                stack2.push(cur.right);
                nextNum++;
            }

        } else {
            TreeNode cur = stack2.pop();
            curNum--;
            item.add(cur.val);

            if(cur.right != null) {
                stack1.push(cur.right);
                nextNum++;
            }

            if(cur.left != null) {
                stack1.push(cur.left);
                nextNum++;
            }

        }

        if(curNum == 0) {
            res.add(item);
            item = new ArrayList<Integer>();
            curNum = nextNum;
            nextNum = 0;
            level++;
        }

    }

    return res;
}
```

104. Maximum Depth of Binary Tree, Easy

http://blog.csdn.net/linhuanmars/article/details/19659525

Given a binary tree, find its maximum depth.
The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.
Subscribe to see which companies asked this question

Key: 層序遍歷．與 102. Binary Tree Level Order Traversal 方法基本上是一樣的，只需多維護一個 level 變量．

History:
E1 的遞歸法代碼幾分鍾寫好，一次通過的，而且代碼和 Code Ganker 的一模一樣．但遞歸法太簡單了，還是要掌握下面的層序遍歷, 但遞歸法還是要看看．
E2 層序遍歷法幾分鍾寫好, 本來可以一次通過的, 結果不小心把 if(lastNum == 0)中的 lastNum = curNum 和 curNum = 0 兩句順序寫反了, 查了好一會兒才查出來, 然後通過.
E3 層序遍歷法幾分鍾寫好, 一次通過. E3 代碼和 Code Ganker 的基本上一模一樣．

Code Ganker:

这是一道比较简单的树的题目，可以有递归和非递归的解法，递归思路简单，返回左子树或者右子树中大的深度加1，作为自己的深度即可，代码如下：

```
public int maxDepth(TreeNode root) {
    if(root == null)
        return 0;
    return Math.max(maxDepth(root.left),maxDepth(root.right))+1;
}
```

非递归解法一般采用层序遍历(相当于图的 BFS)，因为如果使用其他遍历方式也需要同样的复杂度 O(n). 层序遍历理解上直观一些，维护到最后的 level 便是树的深度。代码如下：

我：可以先看 102. Binary Tree Level Order Traversal 的注釋

```
public int maxDepth(TreeNode root) {
    if(root == null)
        return 0;
    int level = 0;
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
    queue.add(root);

//以下的 curNum 即本層要加入 list 的元素個數，相當於 102. Binary Tree Level Order Traversal 中的 lastNum
//以下的 nextNum 即下一層要加入 list 的元素個數，相當於 102. Binary Tree Level Order Traversal 中的 curNum
    int curNum = 1; //num of nodes left in current level
    int nextNum = 0; //num of nodes in next level
    while(!queue.isEmpty())
    {
        TreeNode n = queue.poll();
        curNum--;
        if(n.left!=null)
```

```
        {
            queue.add(n.left);
            nextNum++;
        }
        if(n.right!=null)
        {
            queue.add(n.right);
            nextNum++;
        }
        if(curNum == 0)
        {
            curNum = nextNum;
            nextNum = 0;
            level++;
        }
    }
    return level;
}
```

总体来说我觉得这道题可以考核树的数据结构，也可以看看对递归和非递归的理解。相关的扩展可以是 Minimum Depth of Binary Tree.

105. Construct Binary Tree from Preorder and Inorder Traversal, Medium

http://blog.csdn.net/linhuanmars/article/details/24389549

Given preorder and inorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.
Subscribe to see which companies asked this question

Key: 遞歸. 假设树的先序遍历是 1 245 3687，中序遍历是 425 1 6837, 我们知道 1 是根, 且 425 是左子树，而 6837 是右子树. 寫一個 helper(preorder, preL, preR, inorder, inL, inR, map), 其作用是: 將輸入中的 preorder[preL, … preR]和 inorder[inL, … inR]作為一個子樹的 preorder 遍歷和 inorder 遍歷, 構造出這個子樹, 並返回這個子樹的 root. map 的作用是記錄 所有數 在中序遍历中的 index, 即 for(i) map.put(inorder[i],i).

History:
E1 直接看的答案.
E2 改了好幾次, 主要是最開始調用 helper 時的實參沒寫對, 改後通過, 且代碼跟 Code Ganker 的基本一樣.
E3 改了一次後通過, 原因是調用 helper 時第一個實參應該為 preL+1, 而我不小心寫成了 preL. 其實的實參都和 Code Ganke 一模一樣.

这道题是树中比较有难度的题目，需要根据先序遍历和中序遍历来构造出树来。这道题看似毫无头绪，其实梳理一下还是有章可循的。下面我们就用一个例子来解释如何构造出树。
假设树的先序遍历是 12453687，中序遍历是 42516837 。这里最重要的一点就是先序遍历可以提供根的所在，而根据中序遍历的性质知道根的所在就可以将序列分为左右子树。比如上述例子，我们知道 1 是根，所以根据中序遍历的结果 425 是左子树，而 6837 就是右子树。接下来根据切出来的左右子树的长度又可以在先序便利中确定左右子树对应的子序列（先序遍历也是先左子树后右子树）。根据这个流程，左子树的先序

遍历和中序遍历分别是 245 和 425，右子树的先序遍历和中序遍历则是 3687 和 6837，我们重复以上方法，可以继续找到根和左右子树，直到剩下一个元素。可以看出这是一个比较明显的递归过程，对于寻找根所对应的下标，我们可以先建立一个 HashMap，以免后面需要进行线行搜索，这样每次递归中就只需要常量操作就可以完成对根的确定和左右子树的分割。
算法最终相当于一次树的遍历，每个结点只会被访问一次，所以时间复杂度是 O(n)。而空间我们需要建立一个 map 来存储元素到下标的映射，所以是 O(n)。代码如下.

可以看出上面实现结果还是非常接近于一次树的遍历的，只是我们是以一个构造树的形式，在遍历中把树创建出来。这种题目算是树中的难题了，不过理清思路，其实也不过如此哈~

我: 上面例子中講的樹是:

楼主可以详细解释一下 helper 这个函数吗，没看太懂

回复 jamesji927777：这个函数就是利用先序遍历和中序遍历的特性，确定好各自的窗口的下标，然后递归的把树构建出来，用一个例子试试就知道了哈~

(我: 下面有我對 helper 作用的解釋, 比 Code Ganker 講得好)

不知楼主你还在否？ 我一直在看你的 blog 收获不小 哈哈。

有点迷惑，因为我们可以直接用 inorder 的数组就能构建树，为什么题目还要给我们 preorder 的数组。 还是我题目看不明白。。。 有点虚。。。谢谢你回复 (我: 省略了代帖出來的代碼)

回复 thendore：用 inorder 是构建不出来唯一的树的哈~ 因为一个序列比如 123，他的数可以是 2 为根，然后左右子树是 1 和 3，也可以是 1 为根，然后 2 是 1 的右子树，3 是 2 的右子树，所以无法唯一确定，需要两种遍历序列才能确定树的结构~

回复 linhuanmars：谢谢，我犯傻了 哈哈

这种题还要不要写非递归啊？

回复 u012296380：这种题本身思路比较复杂一些，所以可以不写非递归，不过有时间还是写写好~ 本质都是一样的~

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
    if(preorder==null || inorder==null)
        return null;
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for(int i=0;i<inorder.length;i++)
```

```
    {
        map.put(inorder[i],i);
    }
    return helper(preorder,0,preorder.length-1,inorder,0,inorder.length-1, map);
}

//helper(preorder, preL, preR, inorder, inL, inR, map)的作用是: 將 preorder[preL, … preR]和 inorder[inL, … inR]作為一個子樹的 preorder 遍歷和 inorder 遍歷, 構造出這個子樹, 並返回這個子樹的 root.
private TreeNode helper(int[] preorder, int preL, int preR, int[] inorder, int inL, int inR, HashMap<Integer, Integer> map)
{
    //假设树的先序遍历(preorder)是 12453687，中序遍历(inorder)是 42516837
    if(preL>preR || inL>inR) //走到葉子時的情況. 比如分來分去, 最後只剩一個 node 4, 即調用 helper 時的輸入為 helper(preorder, 2, 2, inorder, 0, 0, map)的時候(此時 index=0), 在此函數體中, 會执行 root.left = helper(preorder, 3, 2, inorder, 0, -1, map), 然後返回 null, 即 root.left = null, 正是我們想要的.
        return null;
    TreeNode root = new TreeNode(preorder[preL]);
    int index = map.get(root.val);
    root.left = helper(preorder, preL+1, index-inL+preL, inorder, inL, index-1, map); //index-inL 是(inorder[]中的)左子樹那段的長度, 如前面例子中的 425 那段
    root.right = helper(preorder, preL+index-inL+1, preR, inorder, index+1, inR,map);
    return root;
}
```

106. Construct Binary Tree from Inorder and Postorder Traversal, Medium

Given inorder and postorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.
Subscribe to see which companies asked this question

Key: 與 105. Construct Binary Tree from Preorder and Inorder Traversal 方法是一樣的. 用上題中我畫的樹為例, 它的中序遍历是 425 1 6837, 後序遍历是 452 8673 1. 我们知道 1 是根, 且 425 是左子树，而 6837 是右子树.

History:
E1 的代碼最開始在 leetcode 中報錯, 參考了 Code Ganker 的代碼後, 找出了一點小錯誤, 就通過了. Code Ganker 的代碼和我的差不多.
E2 很快寫好, 一次通過, 代碼跟 E1 的基本一樣.
E3 很快寫好, 一次通過, helper 中的實參跟 E1 的一樣.

我: 以下用我的代碼, 因為與 105. Construct Binary Tree from Preorder and Inorder Traversal 的代碼比較一致.

Code Ganker:

这道题和 Construct Binary Tree from Preorder and Inorder Traversal 思路完全一样，如果有朋友不了解，请先看看 Construct Binary Tree from Preorder and Inorder Traversal 哈。这里的区别是要从中序遍历和后序遍

历中构造出树，算法还是一样，只是现在取根是从后面取（因为后序遍历根是遍历的最后一个元素）。思想和代码基本都是差不多的，自然时间复杂度和空间复杂度也还是 O(n)。代码如下.

这道题和 Construct Binary Tree from Preorder and Inorder Traversal 是树中难度比较大的题目了，有朋友可能会想根据先序遍历和后序遍历能不能重新构造出树来，答案是否定的。只有中序便利可以根据根的位置切开左右子树，其他两种遍历都不能做到，其实先序遍历和后序遍历是不能唯一确定一棵树的，会有歧义发生，也就是两棵不同的树可以有相同的先序遍历和后序遍历，有兴趣的朋友可以试试举出这种例子哈。

我: 我的代碼:

```
public TreeNode buildTree(int[] inorder, int[] postorder) {
 if(inorder == null || postorder == null)
  return null;

 HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

 for(int i = 0; i < inorder.length; i++) {
  map.put(inorder[i], i);
 }

 return helper(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1, map);
}

private TreeNode helper(int[] inorder, int inL, int inR, int[] postorder, int postL, int postR,
HashMap<Integer, Integer> map) {
 if(inL > inR || postL > postR)
  return null;

 TreeNode root = new TreeNode(postorder[postR]);
 int index = map.get(root.val);

 root.left = helper(inorder, inL, index - 1, postorder, postL, postL + index - inL - 1, map);
 root.right = helper(inorder, index + 1, inR, postorder, postL + index - inL, postR - 1, map); //Code
Ganker 的代碼中, postL + index – inL 是寫成的 postR-(inR-index), 與我的是等價的

 return root;
}
```

107. Binary Tree Level Order Traversal II, Easy

http://blog.csdn.net/linhuanmars/article/details/23414711

Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).
For example:
Given binary tree {3,9,20,#,#,15,7},

```
   3
  / \
 9  20
```

/ \
15 7
return its bottom-up level order traversal as:
[
[15,7],
[9,20],
[3]
]
confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.
Subscribe to see which companies asked this question

Key: 代碼與 102. Binary Tree Level Order Traversal 一模一樣, 唯一的區別就是最後對結果进行一次 reverse.

History:
E1 直接看的答案.
E2 本來可以一次通過的, 結果不小心把 list 定義在了 while 裡面, 改後通過.
E3 是在本題 I 的代碼上直接加的一句 reverse, 一次通過.

这道题和 Binary Tree Level Order Traversal 很类似，都是层序遍历一棵树，只是这道题要求从最底层往最上层遍历。这道题我没有想到什么好的做法可以一次的自底向上进行层序遍历，能想到的就是进行 Binary Tree Level Order Traversal 中的遍历，然后对结果进行一次 reverse。时间上和空间上仍是 O(n)。代码如下.

上述做法确实没有什么特殊的地方，如果是这样做，那么这道题并没有什么考究，跟 Binary Tree Level Order Traversal 完全一样，没有太大意义。如果有朋友有更好的解法或者思路，欢迎交流哈，可以留言或者发邮件到 linhuanmars@gmail.com 给我哈。

why do we need level here?
I mean in the Binary Tree Level Order 1, we don't use level information in it..

回复 shashoujj1991：恩，这里确实不用维护 level，只是作为 bfs 的标准写法哈~\

我: William 也是這樣做的. William: “先用 Top-down level-order，再将结果反转”.

我: 以下代碼稍看看就可以了. 實際上可以用 102. Binary Tree Level Order Traversal 一模一樣的代碼, 只是在 return res 前加一句 Collections.reverse(res)就可以了, leetcode 能通過, 這樣與下面的代碼意思其實差不多, 但更好記憶.

```
public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(root == null)
        return res;
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
    int lastNum = 1;
    int curNum = 0;
```

```
    int level = 0;
    queue.add(root);
    while(!queue.isEmpty())
    {
        TreeNode n = queue.pop(); //此處用了 pop(), 應該是把 queue 當 stack 在用
        if(res.size()<=level)
            res.add(new ArrayList<Integer>());
        res.get(level).add(n.val);
        lastNum--;
        if(n.left!=null)
        {
            queue.add(n.left);
            curNum++;
        }
        if(n.right!=null)
        {
            queue.add(n.right);
            curNum++;
        }
        if(lastNum==0)
        {
            level++;
            lastNum = curNum;
            curNum = 0;

        }
    }
    Collections.reverse(res);
    return res;
}
```

108. Convert Sorted Array to Binary Search Tree, Medium

http://blog.csdn.net/linhuanmars/article/details/23904883

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.
Subscribe to see which companies asked this question

Key: 遞歸. 寫一個 helper(num, l, r), 其作用是: 將 num[l...r]這個子數組轉化為一個 BST, 並返回該 BST 的根. 用 divide and conquer: m = (l + r)/2.

History:
E1 直接看的答案.
E2 很快寫好, 一次通過, 代碼跟 Code Ganker 的差不多.
E3 幾分鍾寫好, 一次通過, 代碼跟 Code Ganker 的一模一樣, 字母一個都不差.

William Tag: DFS

这道题是二分查找树的题目，要把一个有序数组转换成一颗二分查找树。其实从本质来看，如果把一个数组看成一棵树（也就是以中点为根，左右为左右子树，依次下去）数组就等价于一个二分查找树。所以如果要构造这棵树，那就是把中间元素转化为根，然后递归构造左右子树。所以我们还是用二叉树递归的方法来实现，以根作为返回值，每层递归函数取中间元素，作为当前根和赋上结点值，然后左右结点接上左右区间的递归函数返回值。时间复杂度还是一次树遍历 O(n)，总的空间复杂度是栈空间 O(logn)加上结果的空间 O(n)，额外空间是 O(logn)，总体是 O(n)。代码如下.

这是一道不错的题目，模型简单，但是考察了遍历和二分查找树的数据结构，比较适合在电面中出现，类似的题目有 Convert Sorted List to Binary Search Tree，是这道题非常好的后续问题，不同数据结构，实现上也会有所不同，有兴趣的朋友可以看看哈。

您好！请问空间复杂度中 O(logn)是如何来的？
回复 AllenKing13：空间复杂度是递归的栈大小，也就是树的高度，所以是 O(logn)~

问一下，这个题目的时间复杂度 为什么是 O(N),有点想不明白。 还有，如果我用同样的方法去做 Convert Sorted List to Binary Search Tree 的时间复杂度又是多少？谢谢
回复既会易经还会易筋经：这里都是 O(n)，可以简单的这样想，每个元素访问到了几次，这里其实就是一次遍历，每个元素访问一次，所以复杂度是 O(n)，这和树的遍历复杂度是一样的哈 ~

我: 根据 BST 的「中序遍歷有序」的性質, 可知以下構造出的確實是 BST

```
public TreeNode sortedArrayToBST(int[] num) {
    if(num==null || num.length==0)
        return null;
    return helper(num,0,num.length-1);
}

//helper(num, l, r)的作用是: 將 num[l...r]這個子數組轉化為一個 BST, 並返回該 BST 的根
private TreeNode helper(int[] num, int l, int r)
{
    //以下的 if 其實是在結 BST 的葉子加 null. 此 if 的條件 l>r 中有當 num[l...r]只有兩個或一個元素時會出現(不會立刻出現, 而是在下面調用 helper 時出現), 而此時正好有空節點, 要加 null 了. 當 num[l...r]有三個或三個以上元素時, 就至少會構成 root-left-right 這樣的樹, 所以還不用馬上加 null.
    if(l>r)
        return null;

    int m = (l+r)/2;
```

```
    TreeNode root = new TreeNode(num[m]);
    root.left = helper(num,l,m-1);
    root.right = helper(num,m+1,r);
    return root;
}
```

109. Convert Sorted List to Binary Search Tree, Medium

http://blog.csdn.net/linhuanmars/article/details/23904937

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.
Subscribe to see which companies asked this question

Key: 遞歸. 寫一個 helper(l, r)，其作用是: 將輸入的鏈表 list_helo 的 list_helo[l,...r]子鏈表 (取名 list_helo 是為了與後面的變量 list 區別開來) 轉化為一個 BST, 返回該 BST 的根, 並將原本指向 list_helo[l](是字母 l, 不是數字 1)處的 list(list 是一個 ListNode 類型的指針, list 是全局的, 它最初指向鏈表的 head)移到指向 list_helo[r+1]處．用 divide and conquer: m = (l + r)/2.

History:
E1 直接看的答案.
E2 本來完全可以做出來的, 但 E1 將 key 寫錯了, 將 list_helo[l]寫為了 list_helo[l-1], 導致我代碼中少了一句 list = list.next, 就沒做出來, 再加上晚上熬夜寫的, 腦殼不怎麼動了, 所以後來就看了答案.
E3 按 108 題完全一樣的方法做的, 用 walker-runner 方法找 list 的中點, 我寫好後, 改了一個 typo(不小心將 r 寫成 l 了)即通過. 但此方法效率很低. 所以 E3 又默寫了一遍以下的答案.

William Tag: DFS (use a static list). William 的方法跟 Code Ganker 一樣.

Code Ganker:
这个题是二分查找树的题目，要把一个有序链表转换成一棵二分查找树。其实原理还是跟 Convert Sorted Array to Binary Search Tree 这道题相似，我们需要取中点作为当前函数的根。这里的问题是对于一个链表我们是不能常量时间访问它的中间元素的。这时候就要利用到树的中序遍历了，按照递归中序遍历的顺序对链表结点一个个进行访问，而我们要构造的二分查找树正是按照链表的顺序来的。思路就是先对左子树进行递归，然后将当前结点作为根，迭代到下一个链表结点，最后在递归求出右子树即可。整体过程就是一次中序遍历，时间复杂度是 O(n)，空间复杂度是栈空间 O(logn)。代码如下.

这道题是不错的题目，不过这种构造的方式比较绕，因为一般来说我们都是对于存在的树进行遍历，这里是模拟一个中序遍历的过程把树从无到有地构造出来。过程比较不常规，不过多想想就明白了哈。

这里的空间复杂度不用算上建立树的 n 个节点么？应该是 O（n）+O(logn)=O(n)?

回复 jijizhan：计算空间复杂度应该是只看额外空间，而存储结果所用的空间是不用计算在内的哈~

欢哥 这题是时间 O(N) 空间 O(N)的话，那我直接创建一个数组，然后遍历赋值，然后用数组做，就跟那道有序数组转换树 一样了，也是时间 O(N) 空间 O(N). 这样行不行呢？

回复 szl198811：这里的空间复杂度是 O(logn)，原来写错了，改过来了哈~

博主你好，helper 的 ArrayList<ListNode> list 参数可以采用 arraylist 和数组，目的是为了在递归的过程中可以更改这个值。但

是如果以 ListNode 类型作为参数就不能更改了。什么时候传入的参数值会被实际改变，什么时候不变呢？请问这叫做 java 的什么机制呢？我没系统的学过 java。恳请博主帮助解释一下，多谢了！

回复 ashash1990：这个是 java 的按值传递机制，就是传入变量的自身值改变，在外部是不会被改变的，因为是赋值了一份传进来~ 但是如果是该对象指向的元素就可以被改变，因为是改变地址指向的值，这部分对没学过 C++的可能比较难理解~ 可以花点精力看看哈~

多谢回复哈。code 第 11 行里，list 是一个 ArrayList，我想用一个只包含一个元素的 Array 也行了。ArrayList 的动态空间特点在这类题目里面似乎没有用途

回复 jasonexcel：哦，是的哈 ~ 这里用数组还是用 ArrayList 都可以的 ~

ArrayList<ListNode> list = new ArrayList<ListNode>();

请问，如果新建一个 Array 而不是 arraylist 也可以实现同样的效果，你为何在 code 的时候选择 arraylist 呢，有什么特别的讲究吗

回复 jasonexcel：请问你说的 Array 是指常用的数组吗？ ArrayList 有个好处是不用自己维护所需的空间，他会自适应控制空间，而数组需要提前指定元素个数 ~ 不知道有没有理解错你的意思哈 ~

private ListNode list; //Code Ganker 的原代碼中無此句．list 由 ListNode 的 default constructor 來初始化．

```
public TreeNode sortedListToBST(ListNode head) {
    if(head == null)
        return null;
    ListNode cur = head;
    int count = 0;

    //以下的 while 使 count 等於這個鏈表中元素的個數. 注意 cur 走到最後一個元素後, 還要再走一步, 最後停在 null 上.
    while(cur!=null)
    {
        cur = cur.next;
        count++;
    }

    list = head; //別忘了這句！Code Ganker 原代碼中是以下兩句:ArrayList<ListNode> list = new ArrayList<ListNode>().和 list.add(head). 其目的正如前面的評論中所說, 是想改變 list. 但這種方法還是有點繁鎖. 我將代碼按照 William 的方法將 list 改為了一個 ListNode, 並將其定義放在 sortedListToBST 函數外(即本代碼第一句, 注意 list 任是 Solution 這個類的一個 member), 其它地方也改了相應修改. 改後可通過. 注意我還試過了將 list 作為一個 ListNode 定義在 sortedListToBST 函數裡面, 然後通過 helper 的參數傳遞, 但 leetcode 中沒通過, 所以應該還是 Code Ganker 說的那樣, ListNode 是按值傳遞的, 即復製了一份傳进來的. 注意 Java 中自定義 class(如 ListNode, 以及 98. Validate Binary Search Tree 中的 Integer)的 object 在函數參數中仍然是按值傳遞的(即復製了一份傳进來的), 當然 built-in 類型(如 int)也是按值傳遞. 若要按引用傳遞, 要麼就像 Code Ganke 那樣用一個只有一個元素的 ArrayList<ListNode>(或 ArrayList<Integer>等), 或者像 William 這樣, 將變量放在所有函數外, 成為 Solution 這個 class 的一個 member.
```

```
    return helper(0,count-1); //由此可知 helper(l, r)中的 l 和 r 都是 zero based 的(即從 0 開始)
}

//helper(l, r)的作用是: 將輸入的鏈表 list_helo 的 list_helo[l,...r]子鏈表 (取名 list_helo 是為了與後面的變量 list 區別開來) 轉化為一個 BST, 返回該 BST 的根, 並將原本指向 list_helo[l](是字母 l, 不是數字 1)處的 list(list 是一個 ListNode 類型的指針, list 是全局的, 它最初指向鏈表的 head)移到指向 list_helo[r+1]處．(這一點可用數學歸納法證明(可先看下面我的注釋)，初始情況可用一個只有三個元素的鏈表 0->1->2 來驗證)．
private TreeNode helper(int l, int r)
{
    if(l>r)
        return null;

    int m = (l+r)/2;
    TreeNode left = helper(l,m-1); //William 說的: 完成后 list 会自动移到 m 处
    TreeNode root = new TreeNode(list.val);
    root.left = left;
    list = list.next; //將 list 移到 m+1 處，為下面調用 helper 構造好不變量. E2 少寫了這一句.
    root.right = helper(m+1,r); //此句执行完後，list 處於 r+1 處，正好維護了我們想要的不變量
    return root;
}
```

110. Balanced Binary Tree, Easy

http://blog.csdn.net/linhuanmars/article/details/23731355

Given a binary tree, determine if it is height-balanced.
For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.
Subscribe to see which companies asked this question

Key: 遞歸. 寫一個 helper(root), 返回以 root 為根的子樹的 depth (即 return Math.max(left,right)+1), 若這個子樹不平衡, 則返回-1 (E3 是將 res[0]設為 false, helper 是這樣一個函數: int helper(TreeNode root, List<Boolean> res), 其餘的跟 Code Gankre 一樣).

History:
E1 直接看的答案.
E2 很快寫好, 但少考慮到了一種 if 情況(見代碼中), 改了幾遍後弄好了, 通過.
E3 沒多久就寫好, 一次通過. E3 沒像 key 中那樣 helper 返回-1, 而是 helper 將 res[0]設為 false, 這樣不易錯 (Code Ganker 的代碼中, 那個 if(left<0 || right<0)容易忘了寫, E2 就忘了). 以後建議用 E3 的代碼.

William Tag: DFS

这道题是树操作的题目，还是老套路用递归。这道题是求解树是否平衡，还是有一些小技巧的。要判断树是否平衡，根据题目的定义，深度是比需的信息，所以我们必须维护深度，另一方面我们又要返回是否为平衡树，那么对于左右子树深度差的判断也是必要的。这里我们用一个整数来做返回值，而 0 或者正数用来表示树的深度，而-1 则用来比较此树已经不平衡了，如果已经不平衡，则递归一直返回-1 即可，也没有继续比较的必要了，否则就利用返回的深度信息看看左右子树是不是违反平衡条件，如果违反返回-1，否则返回左右子树深度大的加一作为自己的深度即可。算法的时间是一次树的遍历 O(n)，空间是栈高度 O(logn)。代码如下.
可以看出树的题目万变不离其宗，都是递归遍历，只是处理上保存量，递归条件和结束条件会有一些变化。

```
public boolean isBalanced(TreeNode root)
{
    return helper(root)>=0;
}
private int helper(TreeNode root)
{
    if(root == null)
        return 0;
    int left = helper(root.left);
    int right = helper(root.right);
    if(left<0 || right<0) //E2 最開始沒寫這個 if, 改了幾遍後弄好了. 這個 if 還是必需的, 若無此 if, 則 left 和 right 都為-1 時(或 left=-1 且 right=0, 即 right 為 null 時), 下面的 abs(left-right)會<2, 最終得出 true 之錯誤結果.
        return -1;
    if(Math.abs(left-right)>=2)
        return -1;
    return Math.max(left,right)+1;
}
```

E3 的代碼:

```
    public boolean isBalanced(TreeNode root) {
        if(root == null) return true;
        List<Boolean> res = new ArrayList<>();
        res.add(true);
        height(root, res);
        return res.get(0);
    }

    private int height(TreeNode root, List<Boolean> res) {
        if(root == null) return 0;
        int heightLeft = height(root.left, res);
        int heightRight = height(root.right, res);
        if(Math.abs(heightLeft - heightRight) > 1) res.set(0, false);
        return Math.max(heightLeft, heightRight) + 1;
    }
```

111. Minimum Depth of Binary Tree, Easy

http://blog.csdn.net/linhuanmars/article/details/19660209

Given a binary tree, find its minimum depth.
The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.
Subscribe to see which companies asked this question

Key: 層序遍歷．與 104. Maximum Depth of Binary Tree 代碼基本一樣, 只是在檢測到第一個葉子的時候就可以返回了.

History:
E1 的遞歸代碼也很快寫好並通過, 但比 Code Ganker 多出了幾句, 所以遞歸法還是用 Code Ganker 的代碼.
我的非遞歸代碼也很快寫好且一次通過, 代碼跟 Code Ganker 的基本一樣, 但我的跟 104. Maximum Depth of

Binary Tree 的代碼更一致, 所以非遞歸法用我的代碼(為了保持與 104. Maximum Depth of Binary Tree 一致, 我作了點小修改). 要掌握的是非遞歸法, 但遞歸法也要看看.
E2 的非遞歸代碼幾分鍾寫好, 一次通過, 代碼跟 Code Ganker 的除了變量名之外, 其餘的一模一樣.
E3 的非遞歸代碼幾分鍾寫好, 一次通過, 代碼以下的答案基本一模一樣.

我: 我自已

Code Ganker:
这道题是树的题目，其实跟 Maximum Depth of Binary Tree 非常类似，只是这道题因为是判断最小深度，所以必须增加一个叶子的判断（因为如果一个节点如果只有左子树或者右子树，我们不能取它左右子树中小的作为深度，因为那样会是 0，我们只有在叶子节点才能判断深度，而在求最大深度的时候，因为一定会取大的那个，所以不会有这个问题）。这道题同样是递归和非递归的解法，递归解法比较常规的思路，比 Maximum Depth of Binary Tree 多加一个左右子树的判断，代码如下：

```
public int minDepth(TreeNode root) {
    if(root == null)
        return 0;
    if(root.left == null)
        return minDepth(root.right)+1;
    if(root.right == null)
        return minDepth(root.left)+1;
    return Math.min(minDepth(root.left),minDepth(root.right))+1;
}
```

非递归解法同样采用层序遍历(相当于图的 BFS），只是在检测到第一个叶子的时候就可以返回了，代码如下(我: 用的我的代碼)

```
 public int minDepth(TreeNode root) {
  if(root == null)
   return 0;

  LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
  queue.add(root);
  int curNum = 1;
  int nextNum = 0;
  int level = 0;

  while(!queue.isEmpty()) {
   TreeNode n = queue.poll();
   curNum--;

   if(n.left == null && n.right == null)
    return level + 1;

   if(n.left != null) {
    queue.add(n.left);
    nextNum++;
   }
```

```
  if(n.right != null) {
   queue.add(n.right);
   nextNum++;
  }

  if(curNum == 0) {
   curNum = nextNum;
   nextNum = 0;
   level++;
  }

 }

  return 0; //也可以寫成 return level, 也能通過. 其實我的代碼最開始就是 return level, 後來看了 Code Ganker 的, 才改成的 return 0.
 }
```

(Code Ganker 的代碼中把 queue.add(cur.left)寫成了 queue.offer(cur.left))

原来用的 queue.add 这里改称 offer 有什么讲究么？

回复 u012296380：从本质上来说是没有区别的，两个做的都是相同的操作，只是 add 属于 list 的操作，而 poll 和 offer 则是队列的操作 (我: 見 Java 書 p806)。所以如果规范一些，应该是在把它当成什么数据结构的时候就用那个数据结构对应的操作比较好哈~

112. Path Sum, Easy

http://blog.csdn.net/linhuanmars/article/details/23654413

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.
For example:
Given the below binary tree and sum = 22,

```
          5
         / \
        4   8
       /   / \
      11  13  4
     / \     \
    7   2     1
```

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.
Subscribe to see which companies asked this question

Key: 簡單. 遞歸. 對 root.left 和 root.right 調用 hasPathSum.

可以不看: 注意題意是要走一條完整的 path. 對於以下情況, 總共只有兩條 path: 1->2->4 和 1->3. 注意 1->2->null 不算一條 path. E2 將 1->2->null 也理解為了一條 path, 所以多花出了時間.

```
    1
   / \
  2   3
```

/ \

4 null

William Tag: DFS

History:
E1 的代碼雖然也通過了, 但在處理某個 node 只有一個 child 的情況時多寫了幾句, 實際上跟本就不用專門考慮這種情況, 所以我比 Code Ganker 的多了幾句, 其餘的代碼都一樣. 不過也是 binary tree 這類題中，我第一次寫自己的代碼並通過了．以下用 Code Ganker 的代碼.
E2 居然沒一次通過, 原因主要是將 return 中的||寫為了&&, 另一個原因如上所述, 改了幾次後通過, 且代碼比 Code Ganker 的稍好.
E3 很快寫好, 一次通過, 代碼跟 Code Ganker 的基本一樣.

Code Ganker:
这道题是树操作的题目，判断是否从根到叶子的路径和跟给定 sum 相同的。还是用常规的递归方法来做，递归条件是看左子树或者右子树有没有满足条件的路径，也就是子树路径和等于当前 sum 减去当前节点的值。结束条件是如果当前节点是空的，则返回 false，如果是叶子，那么如果剩余的 sum 等于当前叶子的值，则找到满足条件的路径，返回 true。算法的复杂度是输的遍历，时间复杂度是 O(n)，空间复杂度是 O(logn)。代码如下.

树的题目在 LeetCode 中出现的频率很高，不过方法都很接近，掌握了就都差不多。这类求解树的路径的题目是一种常见题型，类似的还有 Binary Tree Maximum Path Sum，那道题更加复杂一些，路径可以起始和结束于任何结点（把二叉树看成无向图），有兴趣的朋友可以看看哈。

```
public boolean hasPathSum(TreeNode root, int sum) {
    if(root == null)
        return false;

    //E2 中以下兩句是寫為:
    if(root.left == null && root.right == null)
            return root.val == sum;
    //按 E2 寫更好.

    if(root.left == null && root.right==null && root.val==sum) //root 為 leaf 的情況
        return true;

    return hasPathSum(root.left, sum-root.val) || hasPathSum(root.right, sum-root.val);
}
```

113. Path Sum II, Medium

http://blog.csdn.net/linhuanmars/article/details/23655413

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.
For example:
Given the below binary tree and sum = 22,

```
              5
             / \
            4   8
           /   / \
          11  13  4
```

/ \ / \
7 2 5 1
return
[
[5,4,11,2],
[5,8,4,5]
]
Subscribeto see which companies asked this question

Key: 遞歸. 用我 E2 的方法, 比 Code Ganker 的更簡明清新好懂, 代碼在最後. Code Ganker 的代碼和 key 可以不看. 寫一個 helper(root, sum, item, res)函數, 其作用為: 在以 root 為根的子樹中, 找出所有 path sum 為 sum 的 path, 將 path 中的結點值加入到 item 中, 當一條 path 走完時, 將 item 加入到 res 中. 注意保護現場.

Code Ganker 的方法: 跟我 E2 方法的區別就在於 helper(root, sum, item, res)是在 root 以下的子樹（即以 root.left 為根的子樹或以 root.right 為根的子樹）中找 path. 俱體實現時, 跟我的方法在囉輯上也有些區別.

寫一個 helper(root, sum, item, res), 其作用是：在 root 以下的子樹（即以 root.left 為根的子樹或以 root.right 為根的子樹）中找到一些 path, 這些 path 每一條的和都為 sum, 並將每一條 path 中的節點值都放入一個 item 中，並把這些 item 放入 res 中．注意保護現瑒．

William Tag: DFS

History:
E1 直接看的答案.
E2 看了 E1 按 Code Ganker 方法寫的 key, 反而不是太好寫, 改了幾次也通不過, 就按我自己的方法寫了, 改了一次後就通過了. 代碼比 Code Ganker 的更簡明清新好懂. 這是我第一次自己做出含回溯的遞歸題(E1 做出過不含回溯的遞歸題), 而且還沒按 key 來寫, NB!
E3 沒多久寫好, 一次通過. 但是是直接遞歸調用的 pathSum 函數自己. 方法沒 E2 的好.

Code Ganker:

这道题是树的题目，跟 Path Sum 的要求很接近，都是寻找从根到叶子的路径。这道题目的要求是求所有满足条件的路径，所以我们需要数据结构来维护中间路径结果以及保存满足条件的所有路径。这里的时间复杂度仍然只是一次遍历 O(n)，而空间复杂度则取决于满足条件的路径和的数量（假设是 k 条），则空间是 O(klogn)。代码如下.

这类求解树的路径的题目是一种常见题型，类似的有 Binary Tree Maximum Path Sum，那道题更加复杂一些，路径可以起始和结束于任何结点（把二叉树看成无向图），有兴趣的朋友可以看看哈。

我在想一个问题，大神你怎么解释保护现场这个概念（add then remove），用英语，用几句话- 最近 onsite 各种被三哥调戏口语。。
回复 wangtongd1：应该是 after recursion we remove the element to recover the state of the item before recursive call 吧~
回复 linhuanmars：Precisely
（我：後面還有我對保護現瑒的理解，也是紅字）

我：我已將原代碼中的 ArrayList 改成了 List, 改後能通過

```
public List<List<Integer>> pathSum(TreeNode root, int sum) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(root==null)
        return res;
```

```
    List<Integer> item = new ArrayList<Integer>();
    item.add(root.val);
    helper(root,sum-root.val,item,res);
    return res;
}

//helper(root, sum, item, res)的作用是：在 root 以下的子樹（即以 root.left 為根的子樹或以 root.right 為根的子樹）中找到一些 path, 這些 path 每一條的和都為 sum, 並將每一條 path 中的節點值都放入一個 item 中，並把這些 item 放入 res 中．
private void helper(TreeNode root, int sum, List<Integer> item, List<List<Integer>> res)
{
    if(root == null) //找不到要求的 path
        return;
    if(root.left==null && root.right==null && sum==0)    //找到一條要求的 path
    {
        res.add(new ArrayList<Integer>(item)); //注意此處是新拷貝了一個 item 放入 res 中，因為以前的 item 要在回溯中刪元素
        return; //別忘了 return
    }
    if(root.left!=null)
    {
        item.add(root.left.val); //因為下一行中的 helper 處理的是 root 以下的子樹，故此處要先將 root 的值放入 item 中，這個 path 才完整．
        helper(root.left,sum-root.left.val,item,res); //helper 之作用見前面
        item.remove(item.size()-1); //回溯．調用 helper 之前，item 中有 root.left.val，調用 helper 之後，要將 root.left.val 刪掉，使 item 復原，此即評論中的保護現場．注意在每輪調用 helper 的時候，該輪 helper 在末尾都執行了 item.remove(item.size()-1)，即將自己對 item 的影响抹掉了．所以稍前面我寫”調用 helper 之後，要將 root.left.val 刪掉，使 item 復原”這幾個字處，刪掉的真的就是本輪的 root.left.val. 保護現場實際上就是對遞歸的負責啊！把 item 中元素刪掉了怎麼放入 res 中？前面在放 res 中元素時是在刪之前就新拷貝了一個 item 放入 res 中．
    }
    if(root.right!=null)
    {
        item.add(root.right.val);
        helper(root.right,sum-root.right.val,item,res);
        item.remove(item.size()-1);
    }
}
```

我 E2 的代碼:

```
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(root == null)
            return res;

        List<Integer> item = new ArrayList<Integer>();
        helper(root, sum, item, res);
        return res;
    }
```

//helper(root, sum, item, res)函數, 其作用為: 在以 root 為根的子樹中, 找出所有 path sum 為 sum 的 path, 將整條 path 中的結點值加入到 item 中, 並將 item 加入到 res 中, 然後保護現場, 即刪掉本 helper(root, sum, item, res) 在 item 中所加入的所有值, 即從 root 到 path 結尾的所有值. 以上是對遞歸及回溯最好的解釋, 本題(E2 代碼)也可以當作遞歸及回溯的公式用.

```
private void helper(TreeNode root, int sum, List<Integer> item, List<List<Integer>> res) {
    item.add(root.val);

    if(root.left == null && root.right == null && root.val == sum)
        res.add(new ArrayList<Integer>(item)); //注意 new 後不能跟 List, 而要跟 ArrayList. 已 record 到 Java p791

    if(root.left != null)
        helper(root.left, sum - root.val, item, res);

    if(root.right != null)
        helper(root.right, sum - root.val, item, res);

    item.remove(item.size() - 1); //回溯, 消除前面加入的 root.val
}
```

114. Flatten Binary Tree to Linked List, Medium

http://blog.csdn.net/linhuanmars/article/details/23717703

Given a binary tree, flatten it to a linked list in-place.
For example,
Given

```
      1
     / \
    2   5
   / \   \
  3   4   6
```

The flattened tree should look like:

```
 1
  \
   2
    \
     3
      \
       4
        \
         5
          \
           6
```

click to show hints.
Hints:
If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.
Subscribe to see which companies asked this question

Key: 遞歸 .
注意題目意思:
1. 要按先序遍歷 faltten 成 linked list.
2. 最終得到這樣一個樹: 所有結點的 left 都為 null, right 即相當於 linked list 中的 next.
3. 輸入[1, #, 2]這樣一個樹, 輸出也是[1, #, 2]這樣一個樹

對 root.left 和 root.right 遞歸調用 flatten 函數(flatten 函數的作用在題目中已經說清楚了). 調用後, 可知 '以 root.left 為根的子樹' 和 '以 root.right 為根的子樹' 都已被 flatten 成 linked list 了, 再將它們 merge 成一個 linked list 就可以了, 注意此處說的 liked list 的意思見上面. 注意 merge 的時候不用一個一個元素地去弄, 而是直接將 '以 root.left 為根的子樹' 的這個 linked list 插到 root 和 '以 root.right 為根的子樹' 的這個 linked list 之間就可以了, 且只處理首尾元素就可以了.

History:
E1 直接看的答案.
E2 看的 E1 的老 key(用的 Code Ganker 的方法), whose 思維不是很清晰(見下面黃字), 故沒做出來.
E3 沒多久就寫好, 寫好後 OJ 說超時了, 後來發現題意要求左兒子全為 null, 改後即通過, 由於這是題意沒說清楚, 故仍算我一次通過. E3 的方法跟 William 一樣, E3 的代碼要邏嗦些(代碼見最後), 但 E3 的好處是不用像 William 那樣為了找末尾, 每次往右走到末尾去. E3 解決這個問題的方法是, 不像 William 那樣直接遞歸調用 flatten 函數自己, 而是寫一個 helper, helper 中遞歸調用 helper, TreeNode helper(TreeNode root)的作用為, 將以 root 為根的子樹按題目要求 flatten 成一個 list, 並返回這個 list 的最後一個節點. E3 好像要比 William 好點, 但 William 的代碼要短不少, 更好記憶, 所以以後還是用 William 的代碼.

我: E2 表示以後都用 William 的方法. 要掌握的是他的 recursion 方法, 他的 iteration 的方法也要看看(E2 沒時間看, E3 看了). E1 用的 Code Ganker 的方法(代碼被 E3 刪了), 不太好懂, 且沒體現出遞歸的精神, 故以後不用.

William:
1. Recursion: Time ~ O(N), Space ~ O(logN)
Two steps:
1) recursively flatten leftSubTree and flatten rightSubTree;
2) merge leftLink and rightLink（注意这部分）:
•connect the rightmost node of leftLink to the rightLink;
•point root.right to root.left, and point root.left to null.

```
public void flatten(TreeNode root) {
    if (root == null)   return;
    // recursion
    flatten(root.left);
    flatten(root.right);
    if (root.left == null)  return;
    // merge: root-left-right
    TreeNode p = root.left;
    //以下的 while 只管一句, 作用是讓 p 走到左半條的最右端.
    while (p.right != null) p = p.right;
    //此時左半條首尾為: root.left → … → p. 以下即為將左半條插入到 root 和 右半條 之間.
    p.right = root.right;
    root.right = root.left;
    root.left = null;
}
```

2. Iteration (no Stack): Time ~ O(N), Space ~ O(1)
类似 Recursion 的第二步：

•connect the rightmost node of leftSubTree to the rigthSubTree;
•point curr.right to curr.left, and point curr.left = null;
•move curr to curr.right (Iteration 需要这步，直至 curr == null 停止).

```
public void flatten(TreeNode root) {
    TreeNode curr = root;
    while (curr != null) {
        if (curr.left != null) {
            // connect the rightmost node in the left subtree to the right node
            if (curr.right != null) {
                TreeNode next = curr.left;
                while (next.right != null)  next = next.right;
                next.right = curr.right;
            }
            // move left node to right
            curr.right = curr.left; //將題目中的例子中的將 2 和 4 按进去了
            curr.left = null;
         } //題目中的例子中的 3 怎麼沒按进去？ 答： 這個在下一輪循環中做． 注意下一句 curr =
curr.right, which 使得 curr = 2, 然後下一輪就要弄 2 了
        curr = curr.right;
    }
}
```

E3 的代碼:

```
  public void flatten(TreeNode root) {
    if(root == null) return;
    helper(root);
  }
```

//TreeNode helper(TreeNode root)的作用為, 將以 root 為根的子樹按題目要求 flatten 成一個 list, 並返回這個 list 的最後一個節點.

```
  private TreeNode helper(TreeNode root) {
    if(root == null) return root;
    if(root.left == null && root.right == null) return root;

    TreeNode rootLeft = root.left;
    TreeNode rootRight = root.right;

    if(root.left == null && root.right != null) return helper(rootRight);

    if(root.left != null && root.right == null) {
      root.right = rootLeft;
      root.left = null;
      return helper(root.right);
    }

    TreeNode leftEnd = helper(root.left);
    TreeNode rightEnd = helper(root.right);
    root.right = rootLeft;
```

```
        root.left = null;
        leftEnd.right = rootRight;
        return rightEnd;
    }
```

115. Distinct Subsequences, Hard

http://blog.csdn.net/linhuanmars/article/details/23589057

Given a string **S** and a string **T**, count the number of distinct subsequences of **T** in **S**.
A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).
Here is an example:
S = "rabbbit", **T** = "rabbit"
Return 3.
Subscribe to see which companies asked this question

Key: 題目的意思就是在 s 中選出 t, 並保持順序, 有多少種選法(全選（s=t 時）也算一種選法).

用 res[i][j]表示 's 的前 i 個字符(即 s[0, ...i-1])' 中有多少個 't 的前 j 個字符(即 t[0, ...j-1])' 的序列. 若已知 res[i+1][j+1] 之前的信息, 如何求 res[i+1][j+1](即 s 的前 i+1 個字符 中有多少個 t 的前 j+1 個字符)? 方法是比較 s[i]和 t[j](注意 s[0, ...i]即 s 的前 i+1 個字符), 若它們不相等，則要在 s 的前 i 個 中找 t 的前 j+1 個，即 res[i+1][j+1]=res[i][j+1]; 若 s[i]和 t[j]相等，則可以有兩種找法，一是在 s 的前 i 個 中找 t 的前 j 個，然後在它們 both 後面加上 s[i]和 t[j]，二是直接在 s 的前 i 個 中找 t 的前 j+1 個．所以遞推式為(爭取不看)：res[i+1][j+1] = (s[i]==t[j] ? res[i][j] : 0) + res[i][j+1]. 如何初始化 res[0]? 想一想 s="a", t="a"就知道了．Code Ganker 說的有點 confusing, 不看.

爭取寫成 1-d DP.

History:
E1 直接看的答案．
E2 改了幾次後通過，但 E2 的代碼比 Code Ganker 的囉嗦不少，因為 E2 用的 res[i][j]表示 's[0, ...i]中有多少個 t[0, ...j]的序列'，這樣使得初始化 res[i][j]比較麻煩.
E3 沒做出來. E3 想到了遞推變量, 但沒想到遞推式.

这道题应该很容易感觉到是动态规划的题目。还是老套路，先考虑我们要维护什么量。这里我们维护 res[i][j]，对应的值是 S 的前 i 个字符(我: 為了與後面的講解一致(但與代碼不一致), 應為前 i-1 個字符, 即 S[0,1...i-1])和 T 的前 j 个字符(我: 為了與後面的講解(但與代碼不一致), 應為前 j-1 個字符, 即 T[0,1...j-1])有多少个可行的序列（注意这道题是序列，不是子串，也就是只要字符按照顺序出现即可，不需要连续出现）。下面来看看递推式，假设我们现在拥有之前的历史信息，我们怎么在常量操作时间内得到 res[i][j]。假设 S 的第 i 个字符和 T 的第 j 个字符不相同(我: i, j 都是子串末尾)，那么就意味着 res[i][j]的值跟 res[i-1][j]是一样的(我: 因為不管怎麼選出 T, 反正不會選 S 的第 i 個字符)，前面该是多少还是多少，而第 i 个字符的加入也不会多出来任何可行结果。如果 S 的第 i 个字符和 T 的第 j 个字符相同，那么所有 res[i-1][j-1]中满足的结果都会成为新的满足的序列，当然 res[i-1][j]的也仍是可行结果，所以 res[i][j]=res[i-1][j-1]+res[i-1][j] 。所以综合上面两种情况，递推式应该是 res[i][j]=(S[i]==T[j]?res[i-1][j-1]:0)+res[i-1][j]。算法进行两层循环，时间复杂度是 O(m*n)，而空间上只需要维护当前 i 对应的数据就可以，也就是 O(m)。代码如下.

可以看到代码跟上面推导的递推式下标有点不同，因为下标从 0 开始，这种细节在实现的时候比较能想清楚，这里 res[j+1]相当于 T 的前 j(我: 應該是 j+1)个字符对应的串，少看一个。而 res[0]表示一个字符都没有时的结果，最后结果返回 res[T.length()]，对应于整个字符串的可行序列的数量。

楼主文章都非常棒，我受益匪浅。 楼主能分析下 brute force 的复杂度吗？我想的是找出所有的 subsequence, 然后对每个

subsequence 查有没有 T, 因为有 2^n 个这样的 subsequence, T 的长度 m,所以感觉是(2^n)*m.但看到有说 n^m 的，怎么分析啊？谢谢

回复 jujuangela：请问你能解释一下你的想法吗哈~ 我理解你的意思如果取出所有的 subsequence 你去判断的话，是会有很多重复的，那你如果去去除这些重复呢？

回复 Code_Ganker：楼主理解得对。我不明白为什么有重复呢？每个 subsequence 不是都不一样吗？能举个例吗？如果 n^m 是对的答案，楼主能分析分析吗？不好意思如果这些问题都太基础。。

回复 jujuangela：比如你 rabbbit 这个 subsequencen 你应该会取到吧，那么里面有三个 rabbit，然后去到 rabbit 应该又会重新取到。要保证不重复应该是取所以长度是 T 的长度的 subsequence, 那么复杂度应该是 C(n,m), 所以这个应该是 O(n^m)的复杂度哈~

回复 Code_Ganker：谢谢！理解了！

```
public int numDistinct(String S, String T) {
    if(T.length()==0)
    {
        return 1;
    }
    if(S.length()==0)
        return 0;

    int[] res = new int[T.length()+1]; //下面多了一個 res[0], 所以是 T.length()+1. res[0]並沒有俱體的意思, 只起輔助作用.
    res[0] = 1; //為何要初始化為 1? 我後面會講.

    for(int i=0;i<S.length();i++)
    {
        for(int j=T.length()-1;j>=0;j--) //注意若改為 j 從小到大變, 則通不過. j 之所以要從大到小變, 是因為 2d 遞推式中右邊要用到 res[i][j], in terms of 1d, 此即上一輪 i 循環中的 res[j], 即 res[j]不能被覆蓋了
        {
            //2d 遞推式為：res[i+1][j+1] = (s[i]==t[j] ? res[i][j] : 0) + res[i][j+1]
            res[j+1] = (S.charAt(i)==T.charAt(j)?res[j]:0)+res[j+1]; //第 i 輪的左邊的 res[j+1]表示在 S[0,1...i]中能選出多少個 T[0,1..j]. 可以理解為 i 為行, j 為列. 右邊的 res[j] 和 res[j+1]是上一輪的值(這就是為何 j 循環要倒著來), 即右邊的 res[j]表示在 S[0,1...i-1]中能選出多少個 T[0,1..j-1], 右邊的 res[j+1]表示在 S[0,1...i-1]中能選出多少個 T[0,1..j].  為何前面要初始化 res[0]=1? 想一想 S 和 T 只有一個字母時, 算 res[1]的值過程, 就知道了.
        }
    }
    return res[T.length()]; //注意不是 res[T.length() - 1]
}
```

116. Populating Next Right Pointers in Each Node, Medium

http://wlcoding.blogspot.com/2015/03/populating-next-right-pointers-in-each.html?view=sidebar

Given a binary tree

```
struct TreeLinkNode {
  TreeLinkNode *left;
  TreeLinkNode *right;
  TreeLinkNode *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.
Initially, all next pointers are set to NULL.
Note:

- You may only use constant extra space.
- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,
Given the following perfect binary tree,

```
       1
     /  \
    2    3
   / \  / \
  4  5  6  7
```

After calling your function, the tree should look like:

```
       1 -> NULL
     /  \
    2 -> 3 -> NULL
   / \  / \
  4->5->6->7 -> NULL
```

Subscribe to see which companies asked this question

Key:
Code Ganker 的方法: 總共用四個指針: 上一層的指針 lastHead, lastCur, 和本層的指針 curHead, pre. 若要連本層的節點, 則用 lastHead 表示上一層的 head, 用 curHead 表示本層的 head, 用 lastCur 表示上一層的某個節點, 用 pre 表示本層的要連的那個->的左端, 則若 lastCur.left != null, 則將 pre.next 設為 lastCur.left, 若 lastCur.right != null, 則將 pre.next 設為 lastCur.right.

William 的方法: 用 curr 指向 A 層的結點, 並從左向右移動. 用 prev 指向 A 層的一下層(稱 B 層)的結點, 並從左向右移動. A 層已連好, 要連的是 B 層. 用一個 dummy 作為 B 層的 fakeHead. prev 最開始指向 dummy, 然後 prev.next = cur.left 或 prev.next = cur.right. 當 cur 走完 A 層後, 遞歸調用 connect(dummy.next), 自動就運 B 的下一層. 對於遞歸, 不要按我習慣的去想 connect 的作用, 這樣反而想不清楚. 由於是最後調用 connecct 的, 所以直接回到代碼開頭去想就是了.

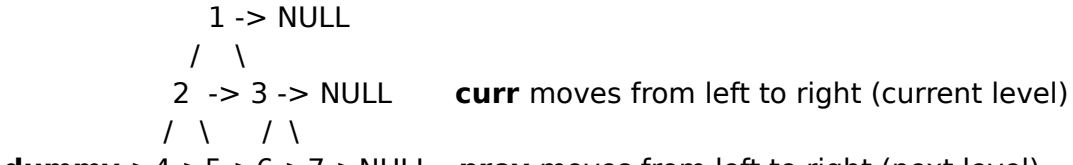
History:
E1 直接看的答案.
E2 通過, 但沒調用 connect, 且代碼只適用於本題, 不適用於本題的 II.
E3 改了一次後通過, 但同樣只適用於本題, 不適用於本題的 II. 我本題的 II 的 E3 代碼是在本題的 E3 代碼基础上改的, 但寫好後改了很多次才通過, 且代碼看起來有點亂, 所以沒將 E3 的代碼帖出來. William 的代碼遞歸調用了 connect, 空間複雜度可能不為 O(1)(注意題目要求用 constant extra space), 所以以後不用 William 的代碼. Code Ganker 的空間複雜度為 O(1), 且比較清新, 以後用 Code Ganker 的代碼.

我: 以下用 William 的代碼, Code Ganker 的代碼太長, 且不太好懂. Code Ganker 說的也沒甚麼有用的, 故沒 Copy.

William:

两道题(我: 指本題和第 117 題, 117 題和本題代碼一模一樣)可以用一个算法解决。

Time ~ O(N), Space ~ O(1)
逐行完成 connection：



注意：

- prev = dummy 和接下来的 prev.next = curr.left or curr.right 使 dummy 停留在 next level 的开头，然后用 prev = prev.next 将 prev 从左向右移动。
- connect 完一行（next level）后，调用 connect(dummy.next) 进入下一行（记得此时 dummy.next 为 next level 的开头）。

我: 以下對使 dummy(所指的 node)的下一個也為本層實際的開頭 的方法很巧妙.
我: 以下代碼不用

```
public void connect(TreeLinkNode root) {
    if (root == null)   return;
    // dummy is the node above curr
    // dummy.next stores the leftmost node in the next level
    TreeLinkNode dummy = new TreeLinkNode(0);
    //以下的 prev = dummy 使得 prev 和 dummy 這兩個變量指向同一個 node(此處的 prev 和 dummy 都是指這兩個變量, 而不是指它們所指的 object)
    TreeLinkNode prev = dummy, curr = root;
    while (curr != null) {
        // when curr has any child:
        // if it's the leftmost node, connect dummy.next to it
        // otherwise. connect previous left node to it
        if (curr.left != null) {
            prev.next = curr.left;//此句使得 prev(所指的 node)的下一個為本層實際的開頭, 注意前面已使 prev 和 dummy 指向同一個 node, 所以此時 dummy(所指的 node)的下一個也為本層實際的開頭
            prev = prev.next; //此時 prev 等於 curr.left. 注意此句已將 prev 這個變量所指的 node 改變了, 但 dummy 沒有, 故 dummy(所指的 node)的下一個仍為本層實際的開頭
        }
        if (curr.right != null) {
            prev.next = curr.right;//此時 prev 仍等於 curr.left,注意是 pass by reference
            prev = prev.next;
        }
        curr = curr.next; //若 curr 為本層最右, 則 curr.next 為 null, 這是題目給出的
    }
     connect(dummy.next);     // go down to the next level //此時將 dummy.next 帶入 connect 中, 再回到本代碼開頭, 將 dummy.next 當作 root, 從頭開始看, 就明白了.
}
```

Code Ganker 的代碼(用它):

```
public void connect(TreeLinkNode root) {
   if(root == null)
      return;
   TreeLinkNode lastHead = root;
   TreeLinkNode pre = null;
   TreeLinkNode curHead = null;
   while(lastHead!=null)
```

```
    {
        TreeLinkNode lastCur = lastHead;
        while(lastCur != null)
        {
            if(lastCur.left!=null)
            {
                if(curHead == null)
                {
                    curHead = lastCur.left;
                    pre = curHead;
                }
                else
                {
                    pre.next = lastCur.left;
                    pre = pre.next;
                }
            }
            if(lastCur.right!=null)
            {
                if(curHead == null)
                {
                    curHead = lastCur.right;
                    pre = curHead;
                }
                else
                {
                    pre.next = lastCur.right;
                    pre = pre.next;
                }
            }
            lastCur = lastCur.next;

        }
        lastHead = curHead;
        curHead = null;
    }
}
```

117. Populating Next Right Pointers in Each Node II, Hard

http://wlcoding.blogspot.com/2015/03/populating-next-right-pointers-in-each.html?view=sidebar

Follow up for problem "*Populating Next Right Pointers in Each Node*".
What if the given tree could be any binary tree? Would your previous solution still work?
Note:

- You may only use constant extra space.

For example,
Given the following binary tree,

```
       1
     /  \
```

```
     2    3
    / \    \
   4   5    7
```
After calling your function, the tree should look like:
```
       1 -> NULL
     /  \
     2 -> 3 -> NULL
    / \    \
   4-> 5 -> 7 -> NULL
```
Subscribe to see which companies asked this question

Key: Code Ganker 的方法: 代碼和本題的 I 的一模一樣. LeetCode 中試過, 可以在本題中通過.
William 的方法: 代碼也和本題的 I 的一模一樣.

History:
E1 直接看的答案.
E2 在我 116 題 E2 的代碼的基础上改的, 但太多 coner case, 改了幾遍都通不過, 就放棄了, 看的答案.
E3 改了很多遍才通過, 詳見本題的 I 中.

118. Pascal's Triangle, Easy

http://blog.csdn.net/linhuanmars/article/details/23311527

Given *numRows*, generate the first *numRows* of Pascal's triangle.
For example, given *numRows* = 5,
Return
```
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```
Subscribe to see which companies asked this question

Key: 用兩個 list: cur 用來表示當前行, pre 用來表示上一行. cur 中的數根据 pre 中的得到, 即 cur 中的數由 pre 中兩個相鄰數相加得到.

History:
E1 的代碼也能通過, 方法其 Code Ganker 一樣, 但我的代碼有點囉嗦(與 Code Ganker 的不同之處我會在下面的代碼中說), 故按照 Code Ganker 的改進了下, 改後基本上和 Code Ganker 的代碼一樣了, 以下用改進後的我的代碼.
E2 八分鍾寫好, 本來可以一次通過的, 結果筆誤將 j 寫為了 i, 改後即通過.
E3 幾分鍾寫好, 一次通過.

这道题比较简单，属于基础的数组操作。基本思路是每层保存前一行的指针，然后当前航数据根据上一行来得到，每个元素就是上一行两个相邻元素相加（第一个和最后一个元素是 1）。算法时间复杂度应该是 O(1+2+3+...+n)=O(n^2)，空间上只需要二维数组来存储结果，不需要额外空间。代码如下.

这道题因为是求解每一行结果，所以空间上没什么好讲究的，Pascal's Triangle II 只求解某一行的数据，反而可以在空间上进行精简，有兴趣的朋友可以看看哈。

楼主您好！请问 pre 和 cur 这两个 ArrayList 不是额外的空间吗？(我: pre 即我的 item, cur 即我的 newItem)

回复 saorenyu：这里严谨一点说应该是 O(1)的空间哈~ 因为 cur 的内容会放入 res 中，所以结果的空间不算在空间复杂度中~ (我: 的確是這樣的, 注意 pre 和 cur 後指的對象其實都放入了 res 中, 所以並沒用出多餘的空間)

```
public static List<List<Integer>> generate(int numRows) {
 List<List<Integer>> res = new ArrayList<List<Integer>>();
 List<Integer> item = new ArrayList<Integer>();

 if(numRows <= 0)
  return res;

 item.add(1);
 res.add(item);

 for(int i = 2; i <= numRows; i++) {
  List<Integer> newItem = new ArrayList<Integer>();
  newItem.add(1);
  //我的原代碼中, 以下 for 的條件寫成的 j < i - 1, 結果會指標溢出(不用去想原因), 弄得我只好把 res 的前兩行都單獨考慮了. 當然也可以將 for 的條件也可寫成 j < i -2 (不用單獨考慮 res 第二行, 可通過), 時間上可能會節約一點(size()函數的時間), 但也節約不到多少, 但意思不如 Code Ganker 的 item.size()-1 那樣明確. 以下還是用 Code Ganker 的 item.size()-1
  for(int j = 0; j < item.size() - 1; j++) {
   newItem.add(item.get(j)+item.get(j+1));
  }

  newItem.add(1);
 //我: 我最開始不放心, 在我的原代碼中把以下兩句寫成的
 res.add(new ArrayList<Integer>(newItem));
 item = new ArrayList<Integer>(newItem);
 其實按下面的 Code Ganker 的寫法是安全的, 而且不會用出多餘的空間.
  res.add(newItem);
  item = newItem;
 }

 return res;
}
```

119. Pascal's Triangle II, Easy

http://blog.csdn.net/linhuanmars/article/details/23311629

Given an index *k*, return the *k*th row of the Pascal's triangle.
For example, given *k* = 3,
Return [1,3,3,1].
Note:
Could you optimize your algorithm to use only *O*(*k*) extra space?

Subscribe to see which companies asked this question

Key: 只用一個 list, 最好寫成 inplace 的(題目沒這麼要求). 比如若當前該 list 為[1,3,3,1], 要將它改為[1,4,6,4,1]. 方法是從該 list 的右邊往左邊改: res[i+1]=res[i]+res[i+1], i 從 res.size-2 開始. 每輪 i 循環後的 res 為:

[1,3,3,1], 這是初始的 res. 此時 res[i+1]=res[i]+res[i+1]寫成值即為 4=3+1, 其中的 3 和 1 為 res 最右端的 3 和 1, 然後得到以下的 res, 後同.
[1,3,3,4] ,
[1,3,6,4] ,
[1,4,6,4] ,
最後在最右端添 1:
[1,4,6,4,1]

Convention: 注意此題的 row convention 與數組是一樣的: [1]為第 0 行, [1,3,3,1]是第 3 行, 而
[1],
[1,1],
[1,2,1],
[1,3,3,1],
[1,4,6,4,1]
是 first 5 rows (第 118 題就是弄的 first n rows).

History:
E1 直接看的答案.

E2 改了幾次後通過. E2 的方法是從左往右改, 比 Code Ganker 的方法要繞, 所以以後還是用 Code Ganker 的方法(即以上的 Key). 但 E2 的方法體現了 按遞歸的思想(每輪循環開始時有一個不變量, 每輪循環結束前保護現場, 即使不變量依然為真的)寫循環, 所以我也將 E2 的代碼帖出來, 在後面, 最好也看看.

E3 是在本題 I 的代碼上改的, 幾分鍾改好, 一遍通過. 所以 E3 是跟本題的 I 一樣, 用的兩個 List, 故不是 in-place 的.

这道题跟 Pascal's Triangle 很类似，只是这里只需要求出某一行的结果。Pascal's Triangle 中因为是求出全部结果，所以我们需要上一行的数据就很自然的可以去取。而这里我们只需要一行数据，就得考虑一下是不是能只用一行的空间来存储结果而不需要额外的来存储上一行呢？这里确实是可以实现的。对于每一行我们知道如果从前往后扫，第 i 个元素的值等于上一行的 res[i]+res[i+1]，可以看到数据是往前看的，如果我们只用一行空间，那么需要的数据就会被覆盖掉。所以这里采取的方法是从后往前扫，这样每次需要的数据就是 res[i]+res[i-1]，我们需要的数据不会被覆盖，因为需要的 res[i]只在当前步用，下一步就不需要了(我: 看下面的評論)。这个技巧在动态规划省空间时也经常使用，主要就是看我们需要的数据是原来的数据还是新的数据来决定我们遍历的方向。时间复杂度还是 O(n^2)，而空间这里是 O(k)来存储结果，仍然不需要额外空间。代码如下.

这道题相比于 Pascal's Triangle 其实更有意思一些，因为有一个考点就是能否省去额外空间，在面试中出现的可能性大一些，不过总体比较简单，电面中比较合适。

我: 以下評論說得很有道理:

我感觉从前往后是 res[i] = res[i-1]+res[i], res[i+1] = res[i]+res[i+1]

所以 res[i]被覆盖了。而从后往前是 res[i] = res[i]+res[i-1], res[i-1]=res[i-1]+res[i-2]，所以 res[i]只用了一次

(我: William 使用了兩個 list, 細節沒看, 應該沒 Code Ganker 的好)

```
public ArrayList<Integer> getRow(int rowIndex) {
   ArrayList<Integer> res = new ArrayList<Integer>();
   if(rowIndex<0) //注意不能像 118 題中那樣寫成 rowIndex<=0, 否則第 0 行就是[], 但正確的第 0 行應該是[1]
       return res;
   res.add(1);
   for(int i=1;i<=rowIndex;i++)
   {
      for(int j=res.size()-2;j>=0;j--)
      {
         res.set(j+1,res.get(j)+res.get(j+1));
      }
      res.add(1);
   }
   return res;
}
```

E2 的方法和代碼:

比如若當前該 list 為[1,3,3,1], 要將它改為[1,4,6,4,1]. 方法是從該 list 的從左邊往右邊改(跟 Code Ganker 是反過來的): 所以公式是 res[i]=res[i-1]+res[i], i 從 1 開始. 每輪 i 循環後的 res 為:

[1,3,3,1], 這是初始的 res. res[i]=res[i-1]+res[i]寫成值即為 4=3+1, 其中的 3 和 1 為最左端的 3 和 1, 然後得到以下的 res:
[1,4,3,1], res[i]=res[i-1]+res[i]寫成值即為 6=3+3, 其中的兩個 3 為上一個 res(即[1,3,3,1])中的兩個 3, 然後得到以下的 res. 複雜之處就在於要用上一個 res. 後同.
[1,4,6,1],
[1,4,6,4],
最後再添一個 1:
[1,4,6,4,1],

```
   public List<Integer> getRow(int rowIndex) {
      List<Integer> res = new ArrayList<>();
      if(rowIndex < 0) return res;
      res.add(1);
      if(rowIndex == 0) return res;
      int resGetIMinus1 = 1, temp = 1;

      for(int count = 1; count <= rowIndex; count++) {
         for(int i = 1; i < res.size(); i++) {
//每輪 i 循環開始時(即每輪走到此處時), 我們假定有以下不變量: resGetIMinus1 的值為公式 res[i]=res[i-1]+res[i]中正確的 res[i-1], 即上一輪的 res(看我代碼之前的方法)中的 res[i-1]. 有了這個假定後, 下面 res.set(i, resGetIMinus1 + res.get(i))一句就可以放心用.
            temp = res.get(i);
            res.set(i, resGetIMinus1 + res.get(i));
            resGetIMinus1 = temp; //此句可以理解為遞歸中的保護現場 使不變量依然為真的 , 即將
```

```
resGetIMinus1 設為循環開始時假定的那個樣子.
            }

            res.add(1);
        }

        return res;
    }

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

120. Triangle, Medium

http://blog.csdn.net/linhuanmars/article/details/23230657

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.
For example, given the following triangle

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

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).
Note:
Bonus point if you are able to do this using only *O*(*n*) extra space, where *n* is the total number of rows in the triangle.
Subscribe to see which companies asked this question

Key: 動態規劃. 用一個數組 res, 其長度為 triangle 最後一行之長度. res[i]用來記錄從 triangle 某一行的第 i 個元素出發 往下走到底的 Minimum Path Sum. Triangle 的每一行都對應一個 res 數組, triangle 最後一行對應的 res 數組就等於 triangle 最後一行, 然後由此可得出 triangle 倒數第二行對應的 res(此 res 已將 triangle 最後一行 對應的 res 覆蓋), 然後可以得出更上層對應的 res, 直到得到最高一層對應的 res. 最終結果即為 res[0]. 第 i 層(最上層為第 0 層)的 res 可由以下遞推式得出(j 表示第 j 列, 最左列為第 0 列):
res[j] = Math.min(res[j],res[j+1])+triangle.get(i).get(j)

```
題目中的 triangle:      各層對應的 res 數組:
    [2],                    [11,...]
   [3,4],                  [9,10,...]
  [6,5,7],                [7,6,10,...]
 [4,1,8,3]                [4,1,8,3]
```

History:
E1 直接看的答案.

E2 先按 2-d DP 寫的(Solution2), 本來可以一次通過, 但不小心將遞推式中的 sum 寫成了 triangle, 改後即通過. 然後改為 1-d DP(Solution2b), 最開始循環方向寫反了, 使得數組被覆蓋, 而且不小心將>=寫成了<=, 改後即通過. 然後又按由下到上搜索的方式改进 1-d DP(Solution2c), 最開始也把循環方向寫反了, 使得數組被覆蓋, 改後即通過.

E3 沒多久就寫好, 一次通過. E3 的方法跟 Code Ganker 的 1-d DP 一樣, 代碼稍有區別(Code Ganker 的代更好懂). E3 基本上是獨立做出來的, 因為 E3 將 key 中方法基本上忘光了, 最開始一直是按遞歸在想, 但發現一個反例, 說明遞歸法是錯的, 然後才一步步想出了 key 中方法. 之前的 key 寫得有點 confusing, 故 E3 還重寫了 key.

我: Code Ganker 的第一種方法看看就可以了, 要掌握的是 Code Ganker 第二種方法(看了第一種方法, 第二種才好懂, 但 E3 表示直接就獨立想出第二種方法了). William 也是用的 Code Ganker 的第二種方法, William 說這是 1-d DP (滚动数组 ← E3 將這四個字看成 滚粗 了)

这是一道动态规划的题目，求一个三角形二维数组从顶到低端的最小路径和。思路是维护到某一个元素的最小路径和(我: 即 sum)，那么在某一个元素 i，j(我: i 為行, j 為列)的最小路径和就是它上层对应的相邻两个元素的最小路径和加上自己的值，递推式是 sum[i][j]=min(sum[i-1][j-1],sum[i-1][j])+triangle[i][j]。最后扫描一遍最后一层的路径和，取出最小的即可。每个元素需要维护一次，总共有 1+2+...+n=n*(n+1)/2 个元素 (我: 題目中說了, n 為總行數)，所以时间复杂度是 O(n^2)。而空间上每次只需维护一层即可（因为当前层只用到上一层的元素），所以空间复杂度是 O(n) (我: 最後一行的元素個數也等於總行數 n)。代码如下.

Code Ganker 代碼二(不用):

```
public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
    if(triangle == null || triangle.size() == 0)
        return 0;
    if(triangle.size()==1)
        return triangle.get(0).get(0);
    int[] sums = new int[triangle.size()]; //sum[i]表示當前行第 i 個元素的最小路徑, 由於最後一行的元素個數也等於總行數 n ,所以要分配 triangle.size()這麼空間多給 sum
    sums[0] = triangle.get(0).get(0); //初始化第 0 行的第 0 個元素的最小路徑, 即最頂端那個元素
    for(int i=1;i<triangle.size();i++) // i 為行指標,  從第 1 行開始, 到最後一行, 計算每行每個元素的最小路徑
    {
        sums[i] = sums[i-1]+triangle.get(i).get(i); //當前行(第 i 行)的最後一個元素(第 i 個元素)要單獨處理. 左邊的 sum[i]為當前行的最後一個元素的最小路徑, 右邊的 sum[i-1]是上一行的最後一個元素的最小路徑.
        for(int j=i-1;j>=1;j--)
        {
            sums[j] = (sums[j]<sums[j-1]?sums[j]:sums[j-1]) + triangle.get(i).get(j); //處理當前行(第 i 行)的第 j 個元素. 左邊的 sum 為當前行的, 右邊的 sum 都是上一行的.
        }
        sums[0] = sums[0]+triangle.get(i).get(0); // //當前行(第 i 行)的最開頭的元素(第 0 個元素)也要單獨處理.  左邊的 sum 為當前行的, 右邊的 sum 都是上一行的.
       //注意以上是從右到左處理的, 因為若從左到右處理, 則左邊的元素的 sum 值會被覆盖. Think of 為何 sum[0]要最後處理就知道了.

    }
   //以下是掃描最後一行的每個元素的最小路徑, 找出最小的.
    int minimum = sums[0];
```

```
    for(int i=1;i<sums.length;i++)
    {
        if(sums[i]<minimum)
        {
            minimum = sums[i];
        }
    }
    return minimum;
}
```

上述代码实现时要注意每层第一个和最后一个元素特殊处理一下。

换个角度考虑一下，如果这道题不自顶向下进行动态规划，而是放过来自底向上来规划(我: 本題的路徑 從上往下走 和 從下往上走 結果是一樣的)，递归式只是变成下一层对应的相邻两个元素的最小路径和加上自己的值，原理和上面的方法是相同的，这样考虑到优势在于不需要最后对于所有路径找最小的，而且第一个元素和最后元素也不需要单独处理了，所以代码简洁了很多。代码如下：

Code Ganker 代碼二(用它):

```
public int minimumTotal(List<List<Integer>> triangle) { //注意若輸入的就是 List<List<Integer>>, 則不用把它轉換成 ArrayList<List<Integer>>, 直接用就是了. 其它的也一樣, 比如輸入為 Set<String>
    //自己寫的時候可以定義一個 n = triangle.size()
    if(triangle.size() == 0)
        return 0;
    int[] res = new int[triangle.size()]; //res 即相當於第一種方法中的 sum, 即: res[i]表示當前行第 i 個元素的最小路徑, 由於最後一行的元素個數也等於總行數 n ,所以要分配 triangle.size()這麼空間多給 res
    for(int i=0;i<triangle.size();i++) //掃描最後一行, 好初始代 res. i 為列指標. 記住最後一行的元素個數也等於總行數 n.
    {
        res[i] = triangle.get(triangle.size()-1).get(i); //最後一行即第 triangle.size()-1 行
    }
    // 以下處理當前行(第 i 行)的第 j 個元素
    for(int i=triangle.size()-2;i>=0;i--) //倒數第二行即第 triangle.size()-2 行
    {
        for(int j=0;j<=i;j++) //為何 j<=i, 想一想第 0 和第 1 行就知道了
        {
            res[j] = Math.min(res[j],res[j+1])+triangle.get(i).get(j); //左邊的 res 為當前行的, 右邊的 res 都是上一行的. 注意每行要從左往右掃描(與第一種方法不同), 否則 res[j]會被覆盖. 這裡用 min()比第一種方法自己比較要好.
        }
    }
    return res[0];
}
```

这道题是不错的动态规划题目，模型简单，比较适合在电面中出现。

回复 linhuanmars：sorry 有点儿语无伦次。。
我的意思是说：

```
List<Integer> last = null;
List<Integer> cur = null;
```

last = triangle.get(i-1);
cur = triangle.get(i);
我声明的 cur 和 last 算分配空间了么？
因为我貌似只是声明了一个指针，然后让他指向 triangle list 中的某个 element，所以好像没有分配空间？？所以我的空间复杂度是 O(1)？？
抱歉哈 评论内容里不让我贴代码，只能断章取义啦。。。再谢！

回复 massagecream：恩，这样只是分配指针，没有线性空间的分配哈~

回复 linhuanmars：多谢牛牛指点！

121. Best Time to Buy and Sell Stock, Medium

http://blog.csdn.net/linhuanmars/article/details/23162793

Say you have an array for which the *i*th element is the price of a given stock on day *i*.
If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.
Subscribe to see which companies asked this question

From online: 本題(I)題目的意思是整个过程中只能买一只股票然后卖出，也有可能不买不卖。也就是我们要找到一对最低价和最高价，最低价在最高价前面，以最低价买入股票，以最低价卖出股票。

E3 表示以下內容不看也能想到, 故不是很重要, 但也可以看看:
我對股票係列題目(I-IV)題意的理解(若本題用不到, 後面幾題也會用到):
prices[i]中的 i 就是日期, 即第 i 天. LeetCode 和 Code Ganker 說的交易(transaction)就是指買和賣操作, 即買賣算一次交易. 每一次交易的買要在賣之前. 注意題目中說了交易是不能重疊的, 即要本次交易的先賣了, 才能进行下一次交易的買. 本次交易的賣和下一次交易的買可以是同一天, 只是要前者的時間比後者早, 如一個上午, 一個下午. 但如果這樣幹了, 實際上這兩次交易可以合併為一次. 這是為何 III(IV)的題目中會說 at most 2(k) trasactions 的原因之一(really?).
買和賣要在不同天完成(若硬要在同一天完成, 由於當天股票價格是一樣的, 所以相當於沒買沒賣). 同一天買賣的做法是充許的, 但題目中並不認為這是一次交易(just a matter of 定義). 所以 III(IV)的題目中會說 at most 2(k) trasactions(原因之二, really?), 即 2 次交易我可以這樣做: 一次交易的買賣不在同一天, 另一次交易的買賣在同一天, 這種做法是允許的, 但題目只認其做了一次交易. 題目的精確意是: 最多 2(k)次「買賣不在同一天的交易」, 除此之外, 你可以隨便再做多少次「買賣在同一天的交易」, 因為它們根本就沒有影响.

Key: 動態歸劃. 遞推式為:

```
local = Math.max(local+prices[i+1]-prices[i],0);
global = Math.max(local, global);
```

若要知道原理, 可看我在代碼中的解釋(那個波形圖及它下面的解釋). 但最好直接記住. 由 E3 代碼中可知, Code Ganker 的遞推式其實是一語雙關, 故只能記住, 不易想到. 所以建議看看 E3 代碼(沒用 Code Ganker 的遞推式), 以防忘了遞推式之情況.

History:
E1 直接看的答案.
E2 兩分鍾寫好, 一次通過.
E3 很快寫好, 一次通過. E3 忘了遞推式, 但記得我代碼中畫那個波形圖(及方法), 所以寫出了跟遞推式等價的代碼, 由此可知 記住波形圖(及方法)之重要性. E3 的代碼也帖在最後, 可以看看, 以防忘了遞推式之情況.

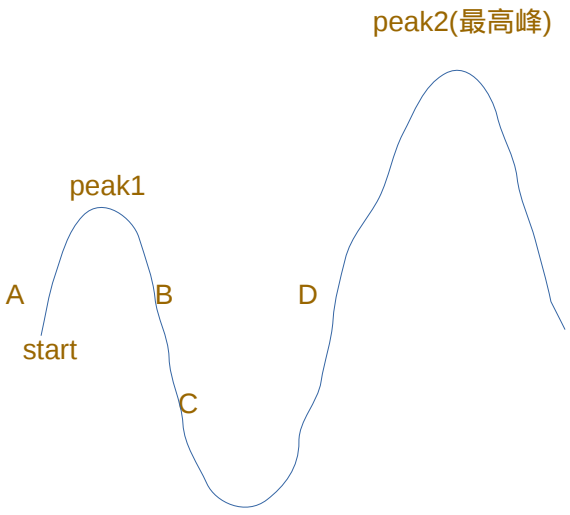
这道题求进行一次交易能得到的最大利润。如果用 brute force 的解法就是对每组交易都看一下利润，取其中最大的，总用有 n*(n-1)/2 个可能交易，所以复杂度是 O(n^2)。

很容易感觉出来这是动态规划的题目，其实跟 Maximum Subarray 非常类似，用“局部最优和全局最优解法”。思路是维护两个变量，一个是到目前为止最好的交易，另一个是在当前一天卖出的最佳交易（也就是局部最优）。递推式是 local[i+1]=max(local[i]+prices[i+1]-price[i],0), global[i+1]=max(local[i+1],global[i])。这样一次扫描就可以得到结果，时间复杂度是 O(n)。而空间只需要两个变量，即 O(1)。代码如下.

这种题目的解法非常经典，不过是比较简单的动态规划。这道题目还有两个变体，Best Time to Buy and Sell Stock II 和 Best Time to Buy and Sell Stock III，虽然题目要求比较像，但是思路却变化比较大，Best Time to Buy and Sell Stock II 可以交易无穷多次，思路还是比较不一样，而 Best Time to Buy and Sell Stock III 则限定这道题交易两次，其实还可以 general 到限定交易 k 次，会在这道题目中仔细讲解，有兴趣的朋友可以看看哈。

Code Ganker 的代碼(用它):

```
public int maxProfit(int[] prices) {
    if(prices==null || prices.length==0)
        return 0;
    int local = 0;
    int global = 0;
    for(int i=0;i<prices.length-1;i++)
    {   //以下兩句的解釋見圖下面
        local = Math.max(local+prices[i+1]-prices[i],0);
        global = Math.max(local, global);
```



valley1(最低谷)

//上圖中豎坐標為價格, 橫坐標為日期. 從上圖中的 start 出發, 在 A 段時, local 的值為 start 到當前天的價格差, local 不斷增加. 到 peak1 時, local 達到極大, global 即等於這個極大值. 在 B 段時(B 段為比 start 高的那段), local 仍為 start 到當前天的價格差, 但 local 不斷減少. 在 C 段時(C 段為比 start 低的那段), local 一直為 0(因為 local+prices[i+1]-prices[i] < 0), 到 valley1 時, local 還是為 0. 在 D 段時, local 又從 0 開始增加, 實際上此時已經將最低谷 valley1 作為了新的 start 開始計數, 直到達到最高峰 peak2 (自動找到了最低谷和最高峰), max 再次被更新為 valley1(最低谷)和 peak2(最高峰)間的價格差, 這就是我們想要的結果. 當然 valley1(最低谷)和 peak2(最高峰)之間還可能有其它峰谷, 這個以後再想. 其實以上只是幫助理解和記憶遞推式, 沒必要去嚴格證明, 把所有情況都考慮到.

```
    }
    return global;
}
```

E3 的代碼(不用, 但可以看看, 以防忘了遞推式之情況):

```
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length <= 1) return 0;
        int n = prices.length;
        int profit = 0, maxPro = 0; //profit 即 Code Ganker 的 local, maxPro 即 Code Ganker 的 global.

        for(int i = 1; i < n; i++) {
            int diff = prices[i] - prices[i - 1];
            if(profit > 0 || (profit <= 0 && diff > 0)) profit += diff; // profit<=0 && diff<0 時, 不做 profit += diff, 這其實際上已在遞推式中暗含了, 所以 Code Ganker 的遞推式是一語雙關, 故只能記住, 不易想到. 實踐表明, 不要此句, 也能通過, 但這其實就等價於 Code Ganker 的遞推式了(含一言雙關), 這樣雖然簡單, 但不易想到, 故還是保留此句.
            if(profit < 0) profit = 0;
            maxPro = Math.max(maxPro, profit);
        }

        return maxPro;
    }
```

122. Best Time to Buy and Sell Stock II, Medium

http://blog.csdn.net/linhuanmars/article/details/23164149

Say you have an array for which the *i*th element is the price of a given stock on day *i*.
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
Subscribe to see which companies asked this question

Key: 本題其實比 I 更簡單. 既然交易次數沒有限定, 所以若 當天比前一天的 prices 大, 就在這兩天交易一次, 即

在 121 題的圖中, 在上坡的段都交易. 最後算出總的 profit 就可以了.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過.
E3 兩分鍾寫好, 一次通過.

这道题跟 Best Time to Buy and Sell Stock 类似，求最大利润。区别是这里可以交易无限多次（当然我们知道交易不会超过 n-1 次(我: 看看代碼中的循環條件就知道甚麼意思了)，也就是每天都进行先卖然后买）。既然交易次数没有限定，可以看出我们只要对于每次两天差价大于 0 的都进行交易，就可以得到最大利润。因此算法其实就是累加所有大于 0 的差价既可以了，非常简单。如此只需要一次扫描就可以了，时间复杂度是 O(n)，空间上只需要 O(1)存一个累加结果即可。代码如下.

这道题其实比 Best Time to Buy and Sell Stock 更加简单，只需要看透背后的模型就很 OK 了哈。

```
public int maxProfit(int[] prices) {

    if(prices == null || prices.length==0)
        return 0;
    int res = 0;
    for(int i=0;i<prices.length-1;i++)
    {
        int diff = prices[i+1]-prices[i]; //在第 i 天買, 在第 i+1 天賣
        if(diff>0) //只对于每次两天差价大于 0 的进行交易, 即在 121 題的圖中只在上坡的段加
            res += diff;
    }
    return res;
}
```

123. Best Time to Buy and Sell Stock III, Hard

Say you have an array for which the *i*th element is the price of a given stock on day *i*.
Design an algorithm to find the maximum profit. You may complete at most *two* transactions.
Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
Subscribe to see which companies asked this question

Key: 本題其實是 188. Best Time to Buy and Sell Stock IV, Hard 的特殊情況, 將其代碼中的 k 定為 2 就能在本題中通過. 不用專門再寫代碼.

History:
E1 直接用的本題 IV 的代碼(令 k=2), 通過.
E1 直接用的本題 IV 的代碼(令 k=2), 通過.
E3 直接用的本題 IV 我 E3 的代碼(令 k=2), 通過.

我: 以下是稍改後的 188 題的 William 的代碼(即直接令 k=2 即可, 其它的都可以不動), 可以在本題中通過. Code Ganker 對本題的解答就是用的 k 情況的代碼並讓其等於 2 的. 但這裡不用 Code Ganker 的代碼, 原因見 188 題.

William 對本題還有另外一個不錯的解法, 有空了可以看看. 但我覺得對本題麼難的題目, 掌握 188 題的算法就夠了(Code Ganker 也是這樣做的).

```
public int maxProfit(int[] prices) {
 int k = 2;
 int len = prices.length;
 int profit = 0;

 if(k > len / 2) {
  for(int i = 0; i < len - 1; i++) {
   int diff = prices[i + 1] - prices[i];
   if(diff > 0) profit += diff;
  }
  return profit;
 }

 int[][] d = new int[k + 1][len];

 for(int i = 1; i <= k; i++) {
  int temp = d[i - 1][0] - prices[0];

  for(int j = 1; j < len; j++) {
   temp = Math.max(temp, d[i - 1][j - 1] - prices[j]);
   d[i][j] = Math.max(d[i][j - 1], prices[j] + temp);
  }

 }

 return d[k][len - 1];
}
```

124. Binary Tree Maximum Path Sum, Hard

http://blog.csdn.net/linhuanmars/article/details/22969069

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

```
      1
     / \
    2   3
```

Return 6.
Subscribeto see which companies asked this question

Key: 遞歸. 本方法記住兩個關鍵詞: 放入, 返回. 本方法其實很好理解, 關鍵是不容易想到.

寫一個 helper(root, res), 其作用是: 1. 把 '以 root 為根的子樹' 的最大路徑和 放入 res 中, 2. 返回 '以 root 為根的子樹' 中, '從 root 開始(包括 root)向下走出的路徑' 中的最大路徑和. 這個返回值是計算 '以 root 的父節點為根的子樹的最大路徑和' 所需要的.

注意若輸入為[2, -1], 則結果應為 2, 即 path 不一定是要到底的, 由於這點, root.left < 0(或 root.right < 0)時, 不要將其加入 path.

William Tag: DFS

History:
E1 直接看的答案.
E2 寫了一上午, 沒通過, 後來看的答案.
E3 沒通過. 其實 E3 的方法很多地方都跟 Code Ganker 的一樣, 只是沒 organize 好, 好幾個地方都用了遞歸, 導致超時了(在我電腦上對題目中給的那個 1-2-3 樹能給出正確答案). Code Ganker 的方法其實很好理解, 關鍵是不容易想到.

这道题是求树的路径和的题目，不过和平常不同的是这里的路径不仅可以从根到某一个结点，而且路径可以从左子树某一个结点，然后到达右子树的结点(我: 從右到左是符價的, 給出一樣的和, 所以沒必要考慮)，就像题目中所说的可以起始和终结于任何结点。在这里树没有被看成有向图，而是被当成无向图来寻找路径。因为这个路径的灵活性，我们需要对递归返回值进行一些调整，而不是通常的返回要求的结果。在这里，函数的返回值定义为以自己为根的一条从根到子结点的最长路径（这里路径就不是当成无向图了，必须往单方向走）(我: 這裡的最長路徑應該是最大路徑和的意思)。这个返回值是为了提供给它的父结点计算自身的最长路径用，而结点自身的最长路径（也就是可以从左到右那种）则只需计算然后更新即可。这样一来，一个结点自身的最长路径就是它的左子树返回值（如果大于 0 的话），加上右子树的返回值（如果大于 0 的话），再加上自己的值。而返回值则是自己的值加上左子树返回值，右子树返回值或者 0（注意这里是“或者”，而不是“加上”，因为返回值只取一支的路径和(我: 見代碼中我的解釋)）。在过程中求得当前最长路径时比较一下是不是目前最长的，如果是则更新。算法的本质还是一次树的遍历，所以复杂度是 O(n)。而空间上仍然是栈大小 O(logn)。代码如下.

树的题目大多是用递归方式，但是根据要求的量还是比较灵活多变的，这道题是比较有难度的，他要用返回值去维护一个中间量，而结果值则通过参数来维护，需要一点技巧。

为什么不设一个 static 全局变量 res，维护这个变量，要用 LinkedList?linkedlist 只是用了第一个值啊。
回复 u012296380：一般来说面试中的题目是不用全局变量的，因为一般不会当成一个类来看。 而且全局变量是一种不是很好的编程习惯， 所以一般不采用。 这个题目用 static 也是不合理的， 因为 static 是指一个会被所有 instance 共用的变量， 而这里没有那种含义。 有时候面试还是要注意编程规范， 比解决问题更重要哈 ~ java 的引用只能通过传一个一个变量的 arraylist 或者数组来解决， 这是由 java 永远按值传递决定的。
回复 linhuanmars：那可以用 wrap 成类的 Integer 吗？

回复 strisunshine：在 java 中用 wrap 也是按值传递的，并没有按引用传递的效果~
回复 linhuanmars：懂了 ~ 谢谢啦，很受用。还是太不规范了。

```
public int maxPathSum(TreeNode root) {
    if(root==null)
        return 0;
    ArrayList<Integer> res = new ArrayList<Integer>();
    res.add(Integer.MIN_VALUE); //E3 默寫時忘了這句.
    helper(root,res);
    return res.get(0);
}

//helper(root, res)的作用是: 1. 把 '以 root 為根的子樹' 的最大路徑和放入 res 中, 2. 返回 '以 root 為根的子樹' 中, '從 root 開始(包括 root)向下走出的路徑' 中的最大路徑和. 這個返回值是計算 '以 root 的父節點為根的子樹的最大路徑和' 所需要的.
private int helper(TreeNode root, ArrayList<Integer> res)
{
    if(root == null)
        return 0;
    int left = helper(root.left, res);
    int right = helper(root.right, res);
    int cur = root.val + (left>0?left:0)+(right>0?right:0); //cur 的作用是把 cur 的值放入的 res 中, 所以它是真正的以 root 為根的子樹的最大路徑和. 之所以要 root.val, left, right 三者相加, 是因為可以從 left 走到 root 再走到 right, 這條路徑是可以的. 為何要用(left>0?left:0), 而不直接用 left? 因為實踐表明 test case 中的節點值可以為負, 若 left<0, 那這條路徑中還不如不要 left 這段, 若不要, 這段的貢獻就是 0.
    if(cur>res.get(0))
        res.set(0,cur); //set 函數見 Java 書 p791
    return root.val+Math.max(left, Math.max(right,0)); //return 的作用是把 return 的值提供給調用 helper 的人, 即本行的 root 的父節點, 所以 return 的不是真正的 root 為根的子樹的最大路徑和, 而是 '不能越過 root 的路徑' 中的最大路徑和, 而這正是計算 '以 root 的父節點為根的子樹的最大路徑和' 所需要的. 前面 maxPathSum 函數中沒有直接 return helper(root,res) 也是這個原因.
}
```

125. Valid Palindrome, Easy

http://wlcoding.blogspot.com/2015/03/palindrome-i-valid-num-ii-valid-str-iii.html?view=sidebar

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.
For example,
"A man, a plan, a canal: Panama" is a palindrome.
"race a car" is *not* a palindrome.
Note:
Have you consider that the string might be empty? This is a good question to ask during an interview.
For the purpose of this problem, we define empty string as valid palindrome.
Subscribe to see which companies asked this question

Key: 两头夹逼. 注意跳過不是字母或數字的字符. 注意使用 **Character**.isLetterOrDigit(char)和 **Character**.toLowerCase(char) (Java 書中已有).

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過.
E3 幾分鍾寫好, 但改了兩個粗心錯誤後才通過, E3 方法跟 William 一樣, 但代碼稍有不同. E3 連 E2 都不如!

alphanumeric: 包括文字与数字的

我: Code Ganker 的方去跟 William 一樣, 但代碼要比 William 的長很多, 原因是 Code Ganker 是自己寫的 Character 的函數, 其實沒必要. 以下用 William 的代碼.

William:
Time ~ O(N), Space ~ O(1)
两头夹逼。

```
public boolean isPalindrome(String s) {
    int i = 0, j = s.length() - 1;
    while (i < j) {

      while (i < j && !Character.isLetterOrDigit(s.charAt(i)))  i++; //while 只管這句
      while (i < j && !Character.isLetterOrDigit(s.charAt(j)))  j--; //while 只管這句

      if (Character.toLowerCase(s.charAt(i)) != Character.toLowerCase(s.charAt(j)))
            return false;

        i++; j--;
    }
    return true;
}
```

但 Code Ganker 說的還是有幾句有用的:

回文串的判断和搜索是面试中经常遇到的基本问题，我在 Facebook 的面试中就遇到了这道题。在 LeetCode 中类似的题目还有 Palindrome Number，Longest Palindromic Substring，Palindrome Partitioning 等，Palindrome Number 跟这道题比较类似，只是判断的是整数，而后面两道还是有一定难度的，有兴趣的朋友可以看看哈。

你好！请问在面试中不能用 Character.isLetterOrDigit 和 Character.toLowerCase 吗？

回复 SequoiaYoung：如果程序的主体不是要求的话应该是没问题的，这些其实可以跟面试官沟通的哈~

126. Word Ladder II, Hard

http://wlcoding.blogspot.com/2015/03/word-ladder-i-ii.html?view=sidebar

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,
Given:

beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]
Return
[
["hit","hot","dot","dog","cog"],
["hit","hot","lot","log","cog"]
]
Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

Subscribe to see which companies asked this question

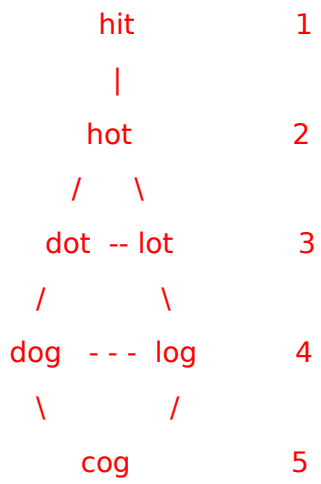
Key: 本題只要知道算法, 其實不算難. BFS + DFS.

BFS 的任務是在 map 中放這樣一些東西: map = {{hit, 1}, {hot, 2}, {dot, 3}, {lot, 3}, {dog, 4}, {log, 4}, ...}, 其中在同一層的單詞有同樣的層數. 第 3 層中的單詞是從第 2 層的單詞導出來的, 這就是層的意思. 而一層取一個就可以構成一個 path, 比如可以取 hit->hot->dot->..., 也可以取 hit->hot->lot->... 而 DFS 的任務就是在 map 中把所有 path 的取法都弄出來. 127 題中的 '那個放 depth 的 Queue' 和 '用來記錄是否已訪問的 Set' 在本題中都可以省去, 因為 map 都有這些功能;

DFS 的實現方法是: 寫一個 dfs(start, end, map, path, pathset)函數, 其作用是(即維護這樣一個不變量): path 裡面已經包括了 start 之前的路徑(包括 start), dfs 函數將從「start 的下一個」開始的路徑從 map 中找出來並接到 path 上, 接著接著若到 end 了, 就將這個 path 放入 pathSet. 最後保護現場, 即將 path 還原為最開始那個樣子(path 裡面已經包括了 start 之前的路徑(包括 start)), 即刪掉「start 的下一個」. 在 DFS 中也要將 string 的每個字母改為 26 個字母來得到該 string 在圖中的 neighbor. (BFS 當然中也要這樣做)

注意: 題目的例子中的 wordList 中不包括 beginWord 和 endWord, 但實際的 test case 中, wordList 是包括 beginWord 和 endWord 的!

題目中的例子的圖為:



History:
E1 直接看的答案.
E2 寫了一晚上, 通過了.
E3(見本題 I 中的 History).

我: 為了與 127. Word Ladder 一致, 本題仍用 William 的代碼. Code Ganker 的代碼很長, 沒看, 也不清楚他的方法. 但他說的要看看, 貼在最後.

William:

BFS + DFS: Time ~ O(26MN), Space ~ O(26MN)
这道题比上题复杂一些，上题用 BFS 只要找到 一个 transformation 即可返回 min depth，而这题需要找到所有的 transformations with min depth。
BFS：用一个 HashMap 存放 key - word, value - depth，如果 current depth >= min depth 则停止搜索，否则循环直到 Queue 为空，这样能把所有的 path 中的 transformation word 放入 HashMap，并标记 depth。
DFS：根据 HashMap 搜索出所有的 path，从 start 搜到 end；依然是每个单词的逐个字母尝试 26 个字母替换，如果新词在 HashMap 中，且其 depth 是目前的 current depth + 1，则将其放入 List 中，直至到达 end。

```
public List<List<String>> findLadders(String start, String end, Set<String> dict) {
    Queue<String> word = new LinkedList<>();
    Map<String, Integer> map = new HashMap<>(); // hashmap<word, depth>
    word.add(start);
    map.put(start, 1);

    // BFS: find all the paths of transformation and store the depths in a HashMap
    int minDepth = Integer.MAX_VALUE;
    while (!word.isEmpty()) {
        String currWord = word.poll();
        int currDepth = map.get(currWord);

//以下的是設定 minDepth 的值, E3 建議將以下的 Ava--döles 之間的代碼 換為 Pre--cémone 之間的代碼(可通過).因為後者意的算法更有道理, 更好理解.

//Ava-\
        //以下的代碼的意思是: 從 queue(即 word)中取出來單詞中, 最小的層數即賦給 minDepth. 但有個問題, 即 minDepth 是動態變化的, 它是逐漸變為最小層數的. 這樣讓人不是很舒服. 以下的 E3 建議的代碼是該 minDepth 一次性地變為最小層數.
        if (currDepth >= minDepth)   continue;   // skip adding words to queue
        if (currWord.equals(end))   {
            minDepth = Math.min(minDepth, currDepth);
            continue;
        }
//-döles


//Pre-
        //以下的代碼的意思是: 在 queue(即 word)中取出來單詞一旦是 end, 則它的層數即為最小層數, 即將其賦給 minDepth. 因為是一層一層地走地, 這個「一層一層地走」本身就足以保證這樣弄出來的是最小層數: 若有一條更迂回的路通向 end, 則我第一次遇到達 end 時, 對應的路當然是最短的, 因為 BFS 每掃一層, 實際上就將所有路徑都碰到了.
        //以下兩個 if 使得 queue(即 word)中 minDepth 層只有 end 一個單詞, 因為最開始 minDepth 為 Integer.MAX_VALUE, 誰都可以加入 queue, 直到 currDepth 為 end 時(此時此處 minDepth仍為 Integer.MAX_VALUE), 然後下面那個 if 給 minDepth 賦了最小層數, 之後的層數大於等於 minDepth 的節點就都不能放进 queue 了.
        if (currDepth >= minDepth)   continue;   // skip adding words to queue

        boolean endWordAlreadyFound = false;
```

```
        if (currWord.equals(end) && !endWordAlreadyFound)   {
                //此處為何不將 end 放入 queue(即 word)中? 因為在上一輪循環中, 訪問 end 的爸爸(如題目中例子中的 dog)時, 已經將在 B 處將 end 作為 newWord 放入 queue 中了
                minDepth = currDepth;
                endWordAlreadyFound = true;
                continue;
        }
//-cémone

        for (int i = 0; i < currWord.length(); i++) {
            char[] currWordArr = currWord.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                currWordArr[i] = c;
                String newWord = new String(currWordArr);
                if (dict.contains(newWord) && !map.containsKey(newWord)) {
                    word.add(newWord);//B 處
                        map.put(newWord, currDepth + 1);//注意是 currDepth+1, 而不是 currDepth++. map 中放的是這樣一些東西: map = {{hit, 1}, {hot, 2}, {dot, 2}, {dog, 2}, {cog, 3}, {hog, 3}...}, 其中在同一層的單詞有同樣的層數. 第 3 層中的單詞是從第 2 層的單詞導出來的, 這就是層的意思. 而一層取一個就可以構成一個 path, 比如可以取 hit->hot->cog->..., 也可以取 hit->dot->hog->... 下面 dfs 函數就是在 map 中把所有 path 的取法都弄出來.
                }
            }
        }
    }

    // DFS: backtracking from the start to put all the paths into the List
    //以下就是「哪裡要用變量, 就在哪裡定義那個變量」的例子:
    List<List<String>> pathSet = new ArrayList<List<String>>();
    List<String> path = new ArrayList<>();
    path.add(start);//A 處. 別忘了這一句. 要加這一句, 是因為下面 dfs 的性質的原因, 看看下面我的解說就知道了
    dfs(start, end, map, path, pathSet);
    return pathSet;
}

// DFS (backtracking)
//dfs(start, end, map, path, pathset)的作用是(即維護這樣一個不變量): path 裡面已經包括了 start 之前的路徑(包括 start), dfs 函數將從「start 的下一個」開始的路徑從 map 中找出來並接到 path 上, 接著接著若到 end 了, 就將這個 path 放入 pathSet. 最後保護現場, 即將 path 還原為最開始那個樣子(path 裡面已經包括了 start 之前的路徑(包括 start)), 即刪掉「start 的下一個」.
private void dfs(String start, String end, Map<String, Integer> map, List<String> path, List<List<String>> pathSet) {
    if (start.equals(end)) {
        //這裡為何不在 path 中加入 end? 這是因為 end 已經在前面 A 處加入 path 了, 注意此時 start = end.
        pathSet.add(new ArrayList<String>(path));    // need to initialize a new ArrayList!! //注意不是 new List<String>(path)
        return;
    }
    int currDepth = map.get(start);
    for (int i = 0; i < start.length(); i++) {
        char[] currWordArr = start.toCharArray();
        for (char c = 'a'; c <= 'z'; c++) {
            currWordArr[i] = c;
            String newWord = new String(currWordArr);
```

```
            if (map.containsKey(newWord) && map.get(newWord) == currDepth + 1) { //
map.get(newWord) == currDepth + 1的目的是確認newWord是start的下一層中的單詞
                int size = path.size();
                path.add(newWord);
                dfs(newWord, end, map, path, pathSet);
                 path.remove(size);//注意對於list, remove函數為list.remove(index), 不
要寫成path.remove(path.get(size));
            }
        }
    }
}
```

我: William另外還有一個方法, 代碼也一樣長. 由於本題太難(見下面Code Ganker說的), 所以掌握上面那一種方法就可以了.

Code Ganker:

这道题是 LeetCode 中 AC 率最低的题目(我: 65. Valid Number 和 8. String to Integer (atoi)比本題還低)，确实是比较难。一方面是因为对时间有比较严格的要求（容易超时），另一方面是它有很多细节需要实现。思路上和 Word Ladder 是比较类似的，但是因为是要求出所有路径，仅仅保存路径长度是不够的...(我: 此處省去八百字)

这道题目实现中有很多细节和技巧，个人觉得如果在面试中遇到很难做到完整，而且考核的算法思想也不是很精妙，更多的是繁琐的操作。在面试中时间上花费太大，也不是很合适，大家主要还是了解一下思路哈。

这么复杂。。。onsite 也不见得会考吧？ - - 我还是有侥幸心理。。。

Code_Ganker 回复 ChiBaoNeLiuLiuNi：确实，这种题目考察点不如 word ladder 那道题~ 我也觉得面试靠这种没意思~

回复 linhuanmars：
大神。。最近看面经，简单的也是出 leetcode hard 题目， 碰到原题就算运气好的了， 本来打算跳过几道 hard， 现在看来还是老实做题了

127. Word Ladder, Medium

http://wlcoding.blogspot.com/2015/03/word-ladder-i-ii.html?view=sidebar

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1.Only one letter can be changed at a time
2.Each intermediate word must exist in the word list

For example,
Given:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]
As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.
Note:

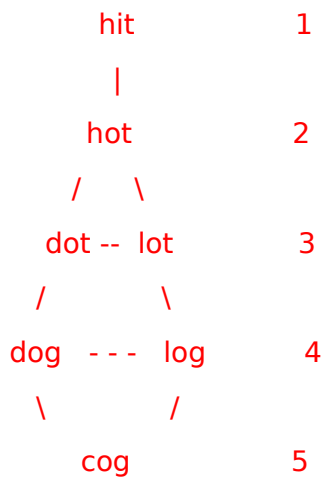
- Return 0 if there is no such transformation sequence.
- All words have the same length.

•All words contain only lowercase alphabetic characters.
Subscribe to see which companies asked this question

Key: 本題其實是一個圖的題目(題目中的例子的圖見 下). 一個 String 即圖的一個 node, 它的 neighbor 即為與它差一個字母的 String. 比如題目中 hit 的 neighbor 為 hot. 另外注意, 只有當這個 String 在 wordList 中時, 我們才把它當作一個 node, 否則這個 node 不存在, 比如 hat 就不是 hit 的 neighbor(即圖中根本就沒有 hat 這麼個 node). 這些 String 們是如何產生的? 比如對於 beginWord: hit, 我們將 h 改為 26 個字母都試一下, 看新的 String 在不在 wordList 中, 若在, 則是一個 node(即 hit 的一個 neighbor), 然後再將 i 改為 26 個字母都試一下... 我們要做的, 就是在圖中找出一從 beginWord 到 endWord 的最短路徑的長度.

算法就是用普通的 BST 來遍歷這個圖. 用一個 queue 來放 node, 再用另一個 queue 來放對應的 node 所在的層數(這個跟 lastNum curNum 方法, 以及每層末尾放 null 的方法, 作用一樣都是 BFS). 當遍歷中發現 endWord 時, 就返回它的層數, 否則返回 0. 注意由於是圖, 所以 hit 和 hot 互為 neighbor, 為了防止倒過來走形成 cycle, 要再用一個 Set 來記錄已訪問過的 node. 即: 當一個 String 在 wordList 中且沒被訪問過時, 才放入 queue 中.

題目中的例子的圖為:



History:
E1 直接看的答案.
E2 本來可以一次通過, 但 level 的初始值設錯了, 改後即通過, E2 的思想跟 William 一樣, 但代碼實現上有點區別, E2 是單獨寫一個函數來得到某個 string 在圖中的 neighbor, 其實沒必要, 別外, E2 是像 Code Ganker 的 BST 那樣維護的層數(即用 lastNum, curNum, level 這三個數), 而 William 用的一個 Queue<Integer>來放層數(不用 lastNum, curNum). 以上 key 是 E2 寫的.
E3 的本題和 II 都能對題目中例子給出正確的結果, 但在 OJ 都對一個較長的輸入(不是很長)超時, 該輸入在我電腦上試了下, 好像不是死循環, 能給出正確的中間結果(即不是最短的那些路徑), 但程序總是運行不完, 應該是我代碼效率太低. 我是(本題和 II 中)用的一個 Map<String, List<String>> , key 為一個 word, value 為這個 word 的所有 next 節點結成的 list. 我 E3 實際上是用的 DFS. 所以對圖的題目(純 graph, 棋盤-code ganker 在 130 題中也這樣說, word ladder 等), 都盡量用 BFS, 因為 DFS 效率太低.

我: 本題用 William 的代碼, 他的方法跟 Code Ganker 差不多(代碼實現上有小差別), 但代碼比 Code Ganker 簡潔且好懂(主要是注釋多些). 但 Code Ganker 說的也要看看, 在最後.

William:

BFS: Time ~ O(26MN) where M is the min steps and N is the word length, Space ~ O(26MN)

Data Structure:
•Use two Queues: one stores the words in transformation, and the other stores the corresponding depth (steps of transformation).
•To avoid cycle in transformation, we need a Hash table to store the visited word.

Procedure:

•For each word, check every character; for each character, replace it with 26 characters and check if it's in the dictionary.
•If the word is in the dictionary (只有當這個 word 在 dictionary 中, 我們才把它當作一個 node, 否則這個 node 不存在), and it hasn't been visited yet, then we add it to the transformation path and Hash table, and increase the depth by 1.
•Need to return 0 in the end (no transformation is found); if a transformation exists, min depth will be returned inside the while loop.

```
public int ladderLength(String start, String end, Set<String> dict) {
    Set<String> map = new HashSet<>(); // record visited words in dict so as not to modify dict //stores the visited word to avoid cycle in transformation. William 居然把這個 Set 取名叫 map.
        Queue<String> word = new LinkedList<String>(); //stores the words in transformation. William 也喜歡用 LinkedList 來摸拟 queue 的功能, 跟 Code Ganker 一樣的習慣. 只是 Code Ganker 左邊也寫成 LinkedList.
     Queue<Integer> depth = new LinkedList<Integer>(); //stores the corresponding depth (steps of transformation)
    word.add(start);
    depth.add(1);

    // BFS
    while (!word.isEmpty()) {
        String currWord = word.poll();//(A行)
        int currDepth = depth.poll();
        // return depth if a match is found
        if (currWord.equals(end))    return currDepth;
        // change each letter of currWord
        for (int i = 0; i < currWord.length(); i++) {
            char[] currWordArr = currWord.toCharArray();
            // try every possible char and check if there's a match in dict
            for (char c = 'a'; c <= 'z'; c++) { //注意 char c 也可以 c++
                currWordArr[i] = c;
                String newWord = new String(currWordArr);
                if (dict.contains(newWord) && !map.contains(newWord)) {
                    word.add(newWord);//在 queue 中加入上面 A 行中 currWord 的所有邻居(如何將此題轉換為一個圖, 參見後面 Code Ganker 說的).
                    depth.add(currDepth + 1);//維護層數, 此 trick 比 Code Ganker 的 lastNum, curNum 要好點. 注意是 currDepth+1, 而不是 currDepth++, 所以 depth 中可能有多個相同的數, 每一個 newWord 都在 depth 中有一個層數對應, 相同層的 newWord 對應的層數的值也是一樣的.
                    map.add(newWord);
                }
            }
        }
    }


    return 0; // no match is found in dict
```

```
}
```

Code Ganker:

这道题看似一个关于字符串操作的题目，其实要解决这个问题得用图的方法。我们先给题目进行图的映射，顶点则是每个字符串，然后两个字符串如果相差一个字符则我们进行连边。接下来看看这个方法的优势，注意到我们的字符集只有小写字母，而且字符串长度固定，假设是 L。那么可以注意到每一个字符可以对应的边则有 25 个（26 个小写字母减去自己），那么一个字符串可能存在的边是 25*L 条。接下来就是检测这些边对应的字符串是否在字典里，就可以得到一个完整的图的结构了。根据题目的要求，等价于求这个图一个顶点到另一个顶点的最短路径，一般我们用广度优先搜索（不熟悉搜索的朋友可以看看 Clone Graph）即可。这个算法中最坏情况是把所有长度为 L 的字符串都看一下，或者把字典中的字符串都看一下，而长度为 L 的字符串总共有 26^L，所以时间复杂度是 O(min(26^L, size(dict))，空间上需要存储访问情况，也是 O(min(26^L, size(dict))。代码如下（我：已省之）

可以看出代码框架其实就是广度优先搜索的基本代码，就是判断边的时候需要换字符和查字典的操作，对于这些图的搜索等基本算法，还是要熟悉哈。

128. Longest Consecutive Sequence, Hard

http://wlcoding.blogspot.com/2015/03/longest-consecutive-sequence.html?view=sidebar

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.
Subscribe to see which companies asked this question

Key: 看下面 William 說的. 本題其實挺簡單的, 雖然是 Hard, 但通過率跟有的 Easy 差不多高.

我: 本題用 William 的代碼, 他的方法跟 Code Ganker 是一模一樣的, 但代碼比 Code Ganker 簡潔很多.

History:
E1 直接看的答案.
E2 很快寫好並通過, 但我的代碼比 William 羅嗦很多.
E3 居然沒做出來. E3 將 key 中方法忘得一幹二淨了.

William:

Time ~ O(N), Space ~ O(N)
因为要 O(n) Time，所以不能够 sort。
方法：用一个 HashSet 存放所有元素。然后对每个元素 i，分别搜索其邻近的左 (i--) 右 (i++) 是否存在 HashSet 中，若找到相应元素则长度加 1，直至搜索不到. 得到的长度为包含 i 的最长连续序列长度。通过遍历每个元素可找到最长的连续序列长度。
注意：每次在 HashSet 中找到一个元素后，必须将该元素删除。如果没有这步，之后该元素还可能被再次搜索到，从而导致 O(N^2) Time。

```
public int longestConsecutive(int[] num) {
    Set<Integer> set = new HashSet<>();
    int max = 0;
    for (int i : num) set.add(i); // 不用寫成 if(!set.contains(i)) set.add(i); 因為若有重復, Set 自動無視, 只加入非重復的.已 record 至 Java p822.
    for (int i : num) {
        int len = 1;
        for (int j = i + 1; set.contains(j); j++) { //將 set.contains(j)放入 for 的條件中, 好巧的寫法, 使代碼簡潔了很多.
            len++;
            set.remove(j);
        }
        for (int j = i - 1; set.contains(j); j--) {
            len++; //E3 默寫時居然寫成了 len--. E3 啊 E3.
            set.remove(j);
        }
        set.remove(i);
        max = Math.max(max, len);
    }
    return max;
}
```

Code Ganker 說的也可以看看:

这道题是要求出最长的整数连续串。我们先说说简单直接的思路，就是先排序，然后做一次扫描，记录当前连续串长度，如果连续串中断，则比较是否为当前最长连续串，并且把当前串长度置 0。这样时间复杂度是很明确，就是排序的复杂度加上一次线性扫描。如果不用特殊的线性排序算法，复杂度就是 O(nlogn)。
其实这个题看起来是数字处理，排序的问题，但是如果要达到好的时间复杂度，还得从图的角度来考虑。思路是把这些数字看成图的顶点，而边就是他相邻的数字，然后进行深度优先搜索。通俗一点说就是先把数字放到一个集合中，拿到一个数字，就往其两边搜索，得到包含这个数字的最长串，并且把用过的数字从集合中移除（因为连续的关系，一个数字不会出现在两个串中）。最后比较当前串是不是比当前最大串要长，是则更新。如此继续直到集合为空。如果我们用 HashSet 来存储数字，则可以认为访问时间是常量的，那么算法需要一次扫描来建立集合，第二次扫描来找出最长串，所以复杂度是 O(2*n)=O(n)，空间复杂度是集合的大小，即 O(n)。代码如下(我: 已省略)

这是一个非常不错的题目，有比较好的算法思想，看起来是一个排序扫描的题目，其实想要优化得借助图的算法，模型也比较简单，很适合在面试中提问。

能不能麻烦再解释一下这道题如何利用图的思想呢？代码看懂了，但是没想明白和图的关系，十分感谢！

回复 qweyxy_2：我的理解是把数字看做顶点， 如果有相邻数字则和相邻数字之间有一条边

回复 qweyxy_2：就是楼上理解的这样哈~

129. Sum Root to Leaf Numbers, Medium

http://blog.csdn.net/linhuanmars/article/details/22913699

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.
An example is the root-to-leaf path 1->2->3 which represents the number 123.
Find the total sum of all root-to-leaf numbers.
For example,

```
  1
 / \
```

2 3
The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Return the sum = 12 + 13 = 25.
Subscribe to see which companies asked this question

Key: 遞歸. 本題的方法很巧, 且不易想到, 所以要記好. 寫一個 int helper(TreeNode root, int sum), 甚返回值為經過 root(此處的 root 不一定是根)的所有 '根到葉路徑形成的數' 之和. 其中 sum 表示 '從根到 root 的爸爸(inclusive)' 這條路徑所形成的數(只是數, 不是和), 這是一個不變量, 即每次調用 helper(root, sum)時, 都要把 sum 的值弄成那樣的.

William Tag: DFS

History:
E1 直接看的答案.
E2 將 sum*10+寫到 helper()外面去了, 導致沒做出來.
E3 早把答案的方法忘了, 自己寫了一個完全不同的 DFS 代碼出來, 改了一次後通過. 雖然也通過了, 但方法顯然沒有答案的方法好. E3 方法大概是將 root.left 子樹的所有 root to leaf numbers 放入一個 list 中, 將 root.right 子樹的所有 root to leaf numbers 放入一個 list 中, 再將 root 的值加到這兩個 list 中每一個元素最前面去(用的 String 合併, 因為不知道每個數有多少位). E3 又將答案的方法默寫了一遍.

这是一道树的题目，一般使用递归来做，主要就是考虑递归条件和结束条件。这道题思路还是比较明确的，目标是把从根到叶子节点的所有路径得到的整数都累加起来，递归条件即是把当前的 sum 乘以 10 并且加上当前节点传入下一函数，进行递归，最终把左右子树的总和相加。结束条件的话就是如果一个节点是叶子，那么我们应该累加到结果总和中，如果节点到了空节点，则不是叶子节点，不需要加入到结果中，直接返回 0 即可。算法的本质是一次先序遍历，所以时间是 O(n)，空间是栈大小，O(logn)。代码如下.

树的题目在 LeetCode(我: 不是面試)中还是有比较大的比例的，不过除了基本的递归和非递归的遍历之外，其他大部分题目都是用递归方式来求解特定量，大家还是得熟悉哈。

```
public int sumNumbers(TreeNode root) {
    return helper(root,0);
}
//int helper(TreeNode root, int sum)的返回值為經過 root(此處的 root 不一定是根)的所有 '根到葉路徑形成的數' 之和. 其中 sum 表示 '從根到 root 的爸爸(inclusive)' 這條路徑所形成的數(只是數, 不是和), 這是一個不變量, 即每次調用 helper(root, sum)時, 都要把 sum 的值弄成那樣的.
private int helper(TreeNode root, int sum)
{
    if(root == null) //這是處理某個 node 只有一個 child(另一個 child 為 null, if 中的 root 即這個 null)之情況. 對於普通的走到某條路盡頭時(即某結點 A 沒有 child, 此處 root 即它的 left 時)的情況, 在之前走到 A 處時, 就已經由下面的那個 if 處理過了.
        return 0;
    if(root.left==null && root.right==null) //這是處理某個 node 無 child 之情況, 即 leaf. 注意根据前面說到的不變量, 此句中的 sum 表示 '從根到 root 的爸爸(inclusive)' 這條路徑所形成的數, 故以下的 sum*10+root.val 表示 '從根到 root' 這條路徑所形成的數. 由於本 root 無兒子, 故可直接返回 sum*10+root.val
        return sum*10+root.val;
```

```
    return helper(root.left,sum*10+root.val)+helper(root.right,sum*10+root.val); //注意根据前面說到的不變量, 此句中的 sum 表示 '從根到 root 的爸爸(inclusive)'  這條路徑所形成的數. 故 sum*10+root.val 表示 '從根到 root(即 root.left 和 root.right 的爸爸)'  這條路徑所形成的數, 不變量在此句中得以維持.
}
```

130. Surrounded Regions, Medium

http://wlcoding.blogspot.com/2015/03/surrounded-regions.html?view=sidebar

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.
A region is captured by flipping all 'O's into 'X's in that surrounded region.
For example,
X X X X
X O O X
X X O X
X O X X
After running your function, the board should be:
X X X X
X X X X
X X X X
X O X X
Subscribe to see which companies asked this question

注意題意： 沿海連通到內地的 O 最後不用換為 X. 注意題目中是大寫字母 O, 不是數字 0, 也不是小寫字母 o, 害得老子查半天.

Key: BFS. 將注意力由內地的O轉移到沿海的O．用 BFS 處理沿海的O．將沿海的（包括連通到內地的）O全換為＃，現在剩下的 O 則是中间被包围的部分，将其全部变为 X。再将原先的 ＃ 恢复成 O 即可。
寫以下兩個函數:

visit(board, i, j, q)的作用為：若 board[i][j]為O，則將其換為#, 並將 i, j 放入 q 中(q 是一個 Queue)．若 board[i][j]不為O，則甚麼都不做．注意本函數並不要求 board[i][j]在邊界上．將 i,j 放入 q 時, 是將坐標 i,j 加密為了一個數存到了 q 中, 此加密的方法其實很 stragith forward, 即將 board 中的所有位置按 從左到右 從上到下 的順序按 0,1,2...賦值, 它們就是加密後的值.

bfs(board, i, j)的作用為：look at board[i][j], 若它不為O，則甚麼都不做；若只有它一個人是O，即四周（它在邊界上就是三周甚麼的）都不是O, 則將它一個人換為＃；若它四周有O，則將那一堆連通的O都換為＃．注意並不會遞歸調用 bfs()函數(注意不叫 dfs()函數), 本題並沒有用到遞歸, 本題是 BFS 題. 在 bfs()中, 先對 i,j 調用 visite, 然後不停地從 q 中取一個 ij, 對它的四周調用 visit().

History:
E1 直接看的答案.
E2 看了 E1 寫的 key, 還是不會做, 就直接看的答案, 然後又添了點 key.
E3 用的跟 200. Number of Islands 題一模一樣的 DFS 方法，對題目中例子能給出正確結果, 但在 OJ 上出現了 StackOverflowError, 網上很多人對本題的 DFS 都出現了這個 StackOverflowError, Code Ganker 在本題中也建議對於棋盤類題都用 BFS 而不是 DFS.

我: William 和 Code Ganker 的方法是一樣的，但 William 的代碼中注釋多些，所以以下用 William 的代碼．但 Code Ganker 說的也要看看(在最後面)．本題代碼雖然長, 但看了我的注釋, 理解起來還是很容易的.

William:

1. BFS: Time ~ O(N^2), Space ~ O(N^2)
对四周的边界的每一个点做 BFS：用一个 Queue，将访问的格子为 O 的变成 #（注意必须区别与 X），然后放入 Queue 中，然后对其上、下、左、右做同样的检查，循环直至 Queue 为空，这样可以从边界的某一点找到所有连续的 O 并变成 # (我： 注意這點，沿海連通到內地的 O 最後不用換為 X)。
现在剩下的 O 则是中间被包围的部分，将其全部变为 X。
再将原先的 # 恢复成 O 即可。

```
public void solve(char[][] board) {
    int m = board.length;
    if (m < 1)  return;
    int n = board[0].length;
    if (m <= 2 || n <= 2)   return;//因為此時就算有O，也必在邊界上，所以不用換

    // run flood fill algorithm (BFS) on every boundry point
    for (int i = 0; i < m; i++) {
        bfs(board, i, 0);       // first column
        bfs(board, i, n - 1);   // last column
    }
    for (int j = 1; j < n - 1; j++) {
        bfs(board, 0, j);       // first row;
        bfs(board, m - 1, j);   // last row;
    }

    // flip 'O' to 'X' and recover '#' to 'O'
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++) {
            if (board[i][j] == 'O')         board[i][j] = 'X';
            else if (board[i][j] == '#')    board[i][j] = 'O';
        }
}


// BFS
//bfs(board, i, j)的作用為：look at board[i][j], 若它不為O，則甚麼都不做；若只有它一個人是O，即四周（它在邊界上就是三周甚麼的）都不是O，則將它一個人換為#；若它四周有O，則將那一堆連通的O都換為#．注意並不會遞歸調用bfs()函數.
private void bfs(char[][] board, int i, int j) {
    Queue<Integer> q = new LinkedList<>();
    visit(board, i, j, q);
    while (!q.isEmpty()) { //不會無限循環下去, 因為已 visit 過的點都已換成了#, 不會再进入queue了
        int pos = q.poll();
        int x = pos / board[0].length;//think of 十进製和二进製
        int y = pos % board[0].length;
        visit(board, x - 1, y, q);
        visit(board, x, y + 1, q);
        visit(board, x + 1, y, q);
        visit(board, x, y - 1, q);
    }
}
```

```
// Mark visited 'O' to '#' and add its coordinates to the queue
//visit(board, i, j, q)的作用為：若 board[i][j]為O，則將其換為#, 並將 i, j 放入 q 中．若board[i][j]不為O，則甚麼都不做．注意本函數並不要求 board[i][j]在邊界上
private void visit(char[][] board, int i, int j, Queue<Integer> q) {
     if (i < 0 || i > board.length - 1 || j < 0 || j > board[0].length - 1 || board[i][j] != 'O')
     return;
    board[i][j] = '#'; // mark 'O' on the boundary to be '#' (we can recover it to 'X' later)
    q.add(i * board[0].length + j);//將坐標 i,j 換化為了一個數存到了 q 中，相當於一個加密過程，後面用到 q 中的元素時會有相應的解密過程．此加密的方法其實很 stragith forward, 即將 board 中的所有位置按 從左到右 從上到下 的順序按 0,1,2...賦值:
  0    1    2 … n-1
  n   n+1 n+2 … 2n-1
  ...
}
```

2. DFS: Time ~ O(N^2), Space ~ O(N^2) StackOverflowError!
棋盘类的题目建议用 BFS 而非 DFS：因为当棋盘的 size 很大时，用 DFS 时可能某一条 path 非常深入，导致 Stack overflow；而用 BFS 时每一个格子的 neighbor 只有 4 个，Queue 是逐渐增大，且上限为棋盘的格数。

(我：DFS 代碼已省略)

Code Ganker:

这个题目用到的方法是图形学中的一个常用方法：Flood fill 算法，其实就是从一个点出发对周围区域进行目标颜色的填充。背后的思想就是把一个矩阵看成一个图的结构，每个点看成结点，而边则是他上下左右的相邻点，然后进行一次广度或者深度优先搜索。
接下来我们看看这个题如何用 Flood fill 算法来解决。首先根据题目要求，边缘上的'O'是不需要填充的，所以我们的办法是对上下左右边缘做 Flood fill 算法，把所有边缘上的'O'都替换成另一个字符，比如'#'。接下来我们知道除去被我们换成'#'的那些顶点，剩下的所有'O'都应该被替换成'X'，而'#'那些最终应该是还原成'O'，如此我们可以做最后一次遍历，然后做相应的字符替换就可以了。复杂度分析上，我们先对边缘做 Flood fill 算法，因为只有是'O'才会进行，而且会被替换成'#'，所以每个结点改变次数不会超过一次，因而是 O(m*n)的复杂度，最后一次遍历同样是 O(m*n)，所以总的时间复杂度是 O(m*n)。空间上就是递归栈（深度优先搜索）或者是队列（广度优先搜索）的空间，同时存在的空间占用不会超过 O(m+n)（以广度优先搜索为例，每次队列中的结点虽然会往四个方向拓展，但是事实上这些结点会有很多重复，假设从中点出发，可以想象最大的扩展不会超过一个菱形，也就是 n/2*2+m/2*2=m+n，所以算法的空间复杂度是 O(m+n)）。代码如下(我: 已省之)

可以看到上面代码用的是广度优先搜索，用一个队列来维护，当然也可以用深度优先搜索，但是如果使用递归，会发现 LeetCode 过不了，这是因为在图形中通常图片（或者说这里的矩阵）一般会很大，递归很容易导致栈溢出，所以即使要用深度优先搜索，也最好使用非递归的实现方式哈。

最后提到如果用非递归的 DFS 就可能不溢出，同样是用 stack 存储，因为图的深度大而溢出，不明白非递归 DFS 怎么能避免这个问题？

回复 derekxu518：因为递归用的是程序运行时的堆空间，一般来说分配给程序的堆空间比较小，而非递归则是动态分配空间，相对来说可以使用内存中的空间，所以在实践中非递归更加常用，因为可以避免递归栈溢出的问题哈~

四个角是不是重复算了？

回复 jimboned：是的~ 确切来说应该避开重复，不过这里因为进去马上回返回，所以实际增加的操作很少，也可以不那么计较哈~

还有个问题比较弱智的哈，为什么在 Leetcode，所有的方程都不用加 static？为什么都是 instant method 呢？

回复 LostBank：因为 leetcode 都是把代码做成一个 solution 类 ~ 而且因为要跑很多个 test case 所以做成成员函数会比 static 要好 ~ 当然其实只要没有成员变量的介入， 就都是一样的哈 ~

好思路，明天照你的思路写。

回复 denganliang825：恩， 这是图形学中比较常用的算法， 我在做图形的时候经常用到哈 ~

回复 linhuanmars：顺利 ac，谢谢

回复 denganliang825：有什么问题可以多交流哈 ~

131. Palindrome Partitioning, Medium

http://blog.csdn.net/linhuanmars/article/details/22777711

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.
Return all possible palindrome partitioning of *s*.
For example, given *s* = "aab",
Return
```
  [
    ["aa","b"],
    ["a","a","b"]
  ]
```
Subscribe to see which companies asked this question

Key: 遞歸．難. 本題實際上比本題的 II 要難些(雖然 II 是 Hard). (E3 表示難個 JB, 我沒多久就寫好, 一遍通過). 寫一個函數 getDict, 它與 II 中的 getDict 一模一樣．寫一個 helper(s, dict, start, item, res)的作用是(其實是維護這樣一個不變量)：將「s 從 start 開始到 s 末尾的這個子串(如 aabcb)」中所有可能的回文組合（如[a, a, bcb], [aa, bcb]）分別加到 item 中（比如若 item 為[efe], 則加後為[efe, a, a, bcb], [efe, aa, bcb]），並將這些加後的 item 全放入 res 中，然後將該 helper 自己對 item 的影响消除，即將其加入 item 中的東西刪掉，即該 helper 执行完後，item 恢復為[efe]. helper 中, 用一個 i 從 start 到 s 末尾掃描, 若 s[start,... i]是回文, 則對 s 剩下的遞歸調用 helper.

History:
E1 直接看的答案.
E2 最開始忘了在 partition 函數中算出 dict 數組, 且在 helper 中回溯時, remove 的參數寫成了元素的值, 而不是 index. 看來是好久沒刷題了, 這麼基本的都忘了. 改後即通過.
E3 沒多久就寫好, 一遍通過. E3 方法和代碼基本上跟 Code Ganker 一樣, 只是 E3 沒用 dic[][]數組, 而是對每一個子串都調用一下 125. Valid Palindrome 題的 isPalindrome(s)函數. 後來比較了下這兩種做法, Code Ganker 的方法要花空間, 我的做法可能會有一些重復計算(稍想即知, 比如最終結果中不同 item 中會有一些一樣的 palindrome, 它們就是重復計算出來的), 總體來講, 還是 Code Ganker 的做法好些, 注意動態規劃的

目的就是用空間來避免重復計算, 所以 Code Ganke 那樣用空間還是值得.

William Tag: DFS+DP. William 的 DFS 方去跟 Code Ganker 的方法很像．但 Code Ganker 在 getDict 中顯然還用了 DP(後來想了想，他用了一個大數組 getDict 來存結果, 其實不算是 DP), 所以 Code Ganker 的方法其實是 DFS+DP. Wiliiam 也有 DFS+DP 方法，但好像跟 Code Ganker 不同了，沒看．上面說一大堆只是為了強調這是 DFS.

Code Ganker:

这道题是求一个字符串中回文子串的切割，并且输出切割结果，其实是 Word Break II 和 Longest Palindromic Substring 结合，该做的我们都做过了。首先我们根据 Longest Palindromic Substring 中的方法建立一个字典，得到字符串中的任意子串是不是回文串的字典，不熟悉的朋友可以先看看哈。接下来就跟 Word Break II 一样，根据字典的结果进行切割，然后按照循环处理递归子问题的方法，如果当前的子串满足回文条件，就递归处理字符串剩下的子串。如果到达终点就返回当前结果。算法的复杂度跟 Word Break II 一样，取决于结果的数量，最坏情况是指数量级的。代码如下.

同样，这里同 Word Break II 一样也可以使用动态规划的方法，但是要对所有中间结果进行存储，花费大量的空间，这里就不列举代码了。这道题扩展还有 Palindrome Partitioning II，虽然求解的问题类似，但是因为一些细节的不同，复杂度会有很大的变化，有兴趣的朋友可以看看哈。

楼主，你好，这个递归调用感觉很精妙，自己想想不到的，像写这种递归的代码，有什么规律可循吗？

回复 kelmt：像这种 NP 问题基本上都是用这种循环中递归的方法，可以看看 N-Queen 那道题哈~ 多做几道就掌握规律了哈~

res.add(new ArrayList<String>(item));
您好，好多地方看到类似句子，然后自己写的时候总忘记 new 直接在 add()中加 item.为什么这样加不进去，一定要 new 一个呢？

回复 wuyuan100：可以加进去，但是这样加进去是把 item 的指针加进去，因为 item 是在递归中一直用这个对象的，如果只是把指针加进去，那么其他地方改动到 item，加进去的也就被改动了~ 这并不是我们想要的效果哈~

LZ,可以解释一下字典的横纵坐标都是什么意思么？
“根据 Longest Palindromic Substring 重大方法建立一个字典”-没看懂是什么意思。
就以 aba 为例， 建成的字典是
1. TFT
2. FTF
3. FFT
谢谢！

回复 jingyuzhang：字典 dict[i][j]的意思表示从 i 到 j 组成的字符串是不是回文串， 可以看看 Longest Palindromic Substring 这道题里面的解释哈， 比较详细 ~

我：我已將以下代碼中的 ArrayList 改為了 List, 改後可通過．

```
public List<List<String>> partition(String s) {
    List<List<String>> res = new ArrayList<List<String>>();
    if(s==null || s.length()==0)
```

```
        return res;
    helper(s, getDict(s),0,new ArrayList<String>(), res); //注意作為實參時，應寫為 ArrayList<String>
    return res;
}
//helper(s, dict, start, item, res)的作用是(其實是維護這樣一個不變量)：將「s 從 start 開始到 s 末尾的這個子串(如 aabcb)」中所有可能的回文組合（如[a, a, bcb], [aa, bcb]）分別加到 item 中（比如若 item 為[efe], 則加後為[efe, a, a, bcb], [efe, aa, bcb]），並將這些加後的 item 全放入 res 中，然後將該 helper 自己對 item 的影响消除，即將其加入 item 中的東西刪掉，即該 helper 執行完後，item 恢復為[efe].
private void helper(String s, boolean[][] dict, int start, List<String> item, List<List<String>> res) //注意作為形參時，應寫為 List<String>
{
    if(start==s.length())
    {
        res.add(new ArrayList<String>(item));
        return;
    }
    for(int i=start;i<s.length();i++)
    {
        if(dict[start][i]) //參數中已經提供了可用的 dict[][], 就不要再將此句寫為 if(getDict(s)[start][i])，否則太浪費空間
        {
            item.add(s.substring(start,i+1));
            helper(s,dict,i+1,item,res);
            item.remove(item.size()-1);//回溯．將本 helper 自己對 item 的影响消除，即保護現瑒．注意不要寫成了 item.remove(item.get(item.size() - 1))，因為 arrayList.remove()的參數是 index
        }
    }
}

//以下的 getDict 函數與 132. Palindrome Partitioning II 中的 getDict 一模一樣．
//字典 dict[i][j]的意思表示从 i 到 j 组成的字符串是不是回文串
private boolean[][] getDict(String s)
{
    boolean[][] dict = new boolean[s.length()][s.length()];
    for(int i=s.length()-1;i>=0;i--)
```

```
    {
        for(int j=i;j<s.length();j++)
        {
            if(s.charAt(i)==s.charAt(j) && ((j-i<2)||dict[i+1][j-1]))
            {
                dict[i][j] = true;
            }
        }
    }
    return dict;
}
```

132. Palindrome Partitioning II, Hard

http://blog.csdn.net/linhuanmars/article/details/22837047

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.
Return the minimum cuts needed for a palindrome partitioning of *s*.
For example, given *s* = "aab",
Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.
Subscribe to see which companies asked this question

Key: 動態歸劃．寫一個函數 getDict[s], 其返回值為一個 boolean[][] dict, dict[i][j]表示从 i 到 j 组成的字符串是不是回文串．其寫法也是 DP. 用 i, j 來掃描, 若有 s[i]=s[j]且 dict[i+1][j-1]=true, 則 dic[i][j]=true. 想想 i, j 要怎樣掃描, 才不能保證算 dic[i][j]時, dict[i+1][j-1]沒有沒覆蓋, 即 dict[i+1][j-1]仍是歷史信息.

Code Ganker 的 minCut 函數寫法(不好懂, 故不推薦): 動態歸劃, 用 res[i]表示 s 前 i 個字符的子串(即 s[0, … i-1])能分成多少個回文(不是多少次 cut, 要減 1 才是 cut)．遞推式是若 dict[j][i]=true, 則 res[i+1] = Math.min(res[i+1],res[j]+1) ，外層 i 掃描整個 s, 內層 j 從 0 到 i 掃描．res[i+1] = Math.min(res[i+1],res[j]+1)意思就是: 不同的 j 值可以通過本公式得出不同的 res[i+1], 然後在在這些不同的 res[i+1]中找最小的. 本公式 中 res[j]+1 之理解: 若 dict[j][i]=true, 即 s[j, ...i]是回文, 則整個 s 回文數量等於 s[0, ..j-1]的回文數(即 res[j])加 1(這個 1 是指 s[j, ...i]這個回文).

E3 的 minCut 函數寫法(好懂, 且思想跟本題的 I 一致, 故推薦): 動態歸劃, 用 res[start]來表將 's 從 s[start]開始(inclusive)到 s 結束' 的這個子串切成 palindrome 要切多少刀. i 從 start 開始向右掃描, 若 s[start, ...i]為 palindrome, 則此時有 1+res[i+1]那麼多種切法, 那個 1 表示將 s[start, ...i]切下來. 然後對所有 i 求切法最小值, 即可得出 res[start]. 當然最外層 start 還要從右往左掃描. 最終要返回的結果為 res[0].

History:
E1 直接看的答案.

E2 看不懂 E1 留的 key, 想了好久都沒想出來, 所以也直接看的答案. 以上 key 基本都是 E2 寫的.

E3 通過. E3 最開始 minCut 是用遞歸寫的, 能在我電腦上結出正確結果, 但在 OJ 上對稍大的輸入超時了, 此輸入在我電腦上也要運行很久. 後來又改為動態歸劃(即以上 key 中 E3 的 minCut 寫法), 還是超時, 後來發是因為我的 getDict 函數效率太低, 我是按原始的 sliding window 寫的(即 s[i, ..j]中的 i 和 j 都是從左往右掃, 然後用 125. Valid Palindrome 中的夾逼法判斷 s[i, ..j]是否為 palindrome), 後來改為 key 中那樣寫(只是思想一樣, 但我的掃描方式還是不同, 所以我的代碼要長不少), 即通過. 由此可見本題對時間效率的要求是很高的, 這可能就是為何本題為 hard 之原因.

这道题跟 Palindrome Partitioning 非常类似，区别就是不需要返回所有满足条件的结果，而只是返回最小的切割数量就可以。做过 Word Break 的朋友可能马上就会想到，其实两个问题非常类似，当我们要返回所有结果（Palindrome Partitioning 和 Word Break II）的时候，使用动态规划会耗费大量的空间来存储中间结果，所以没有明显的优势。而当题目要求是返回某个简单量（比如 Word Break 是返回能否切割，而这道题是返回最小切割数）时，那么动态规划比起 brute force 就会有明显的优势。这道题先用 Palindrome Partitioning 中的方法建立字典，接下来动态规划的方式和 Word Break 是完全一样的，我们就不再细说了，不熟悉的朋友可以看看 Word Break 的分析哈。因为保存历史信息只需要常量时间就能完成，进行两层循环，时间复杂度是 O(n^2)。空间上需要一个二维数组来保存字典和一个线性数组来保存动态规划信息，所以是 O(n^2)。代码如下．

这个问题和 Word Break 可以说是一个题目，这里多了一步求解字典。如果求解所有结果时，他们没有多项式时间的解法，复杂度取决于结果数量，而当求解某一种统计的特殊量时，用动态规划就会很大的优势，可以降低时间复杂度。

hey, 最近怎么样？ return res[s.length()]-1; 这句话我理解的不透彻，为啥要减一，能给个 physical comprehension 吗？

回复 wangtongd1：因为动态规划得到的是切割成多少份，而他求得是几个 cut，所以要减 1 哈~

Code Ganker 的 minCut 函數(不用):

```
public int minCut(String s) {
    if(s == null || s.length()==0)
        return 0;
    boolean[][] dict = getDict(s);
    int[] res = new int[s.length()+1]; //注意下面多加了個 res[0]這個元素, 它不表示甚麼意思, 只起初始的作用, 所以總共需要分配 s.length()+1 個空間給 res. 由於開頭多了一個元素, 故 res[i]即表示 s 從開頭到第 i-1 字符为止的子串能分成多少個回文
    res[0] = 0;
    for(int i=0;i<s.length();i++)
    {
        res[i+1] = i+1; //注意 res[i+1]即表示 s 從開頭到第 i 字符为止的子串能分成多少個回文. 想一想為何 res[1]=1 是有好處的．這裡其實就是將 res[i+1]的初值賦為最大可能的值, 好給下面求 min 用.
        for(int j=0;j<=i;j++)
        {
            if(dict[j][i])
            {
                res[i+1] = Math.min(res[i+1],res[j]+1);
            }
        }
    }
    return res[s.length()]-1;//res 表示的是分成多少份，而我們要求的是多少個 cut, 所以要減 1.  注意不是 res[s.length()-1]，原因見上面說的多了個 res[0].
```

```
}

private boolean[][] getDict(String s)
{
   boolean[][] dict = new boolean[s.length()][s.length()];
   for(int i=s.length()-1;i>=0;i--)
   {
      for(int j=i;j<s.length();j++)
      {
         if(s.charAt(i)==s.charAt(j) && (j-i<2 || dict[i+1][j-1]))
            dict[i][j] = true;
      }
   }
   return dict;
}
```

E3 的 minCut 函數(用它), 和 Code Ganker 的 getDict 函數一起, 能通過:

```
   public int minCut(String s) {
      if(s == null || s.length() <= 1) return 0;
      int n = s.length();

      boolean[][] dict = getDict(s);
      int[] res = new int[n + 1];
      res[n] = -1;

      for(int start = n - 2; start >= 0; start--) {
         int startMin = Integer.MAX_VALUE;

         for(int i = start; i < n; i++) {
            if(dict[start][i]) {
               startMin = Math.min(startMin, 1 + res[i + 1]);
            }
         }

         res[start] = startMin;
      }

      return res[0];
   }
```

133. Clone Graph, Medium

http://wlcoding.blogspot.com/2015/03/clone-graph.html?view=sidebar

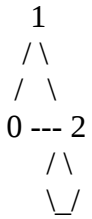
Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

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:



Subscribe to see which companies asked this question

Key: DFS 方法: 寫一個 dfsClone(node)函數, 其作用為: 復製 以 node 及其所有 neighbors(以及 neighbors 的 neighbors)組成的子圖, 並返回這個復製版子圖中 node 的復製版. 用 HashMap 來避免重復訪問. HashMap 的 key 為 visited node, value 為 its copy.

History:
E1 直接看的答案.
E2 改了一次後通過.
E3 沒做出來. E3 沒想到用 key 中那樣的 map, 而是用的一個 map 來放<一個 node 的值, 這個 node>, 結果在 OJ 中大輸入時超時, 我沒去驗證對題目中例子能否給出正確結果.

我: William 的代碼解釋比 Code Ganker 的多, 所以用 William 的代碼. William 用了 DFS 和 BFS, 我要掌握 DFS, 但 BFS 也要看熟悉, 好好體會 DFS 和 BFS 的思想. Code Ganker 說的也可以看看, 在最後面.

From LeetCode:

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     List<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }
 * };
 */
```

William:

遍历 Graph 的 node，create a copy。

1. DFS: Time ~ O(|V| + |E|), Space ~ O(|V| + |E|)
Recursion: call method dfsClone(node)，return 该 node 的 copy；
遇到每一个 node，create 一个 copy，放入其 label，然后逐个加入其 neighbor w 的 copy：copy.neighbors.add(dfsClone(w))。

注：HashMap 记录 key - visited node, value - its copy，避免重复访问。

```
private Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();//注意此map 在所有函數之外(是 Solution 這個 class 的一個 mememberl, 這顯然是為了後面的函數好修改它

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
    if (node == null)   return null;
    return dfsClone(node);
}

//dfsClone(node)的作用為: 復製以 node 及其所有 neighbors(以及 neighbors 的 neighbors)組成的子圖, 並返回這個復製版子圖中 node 的復製版
private UndirectedGraphNode dfsClone(UndirectedGraphNode node) {
     if (map.containsKey(node))  return map.get(node); /* need to add visited node to neighbor list */
     UndirectedGraphNode copy = new UndirectedGraphNode(node.label); //注意復製一個節點的方法. 不要直接 copy = node, 這樣只是復製了 reference.

    map.put(node, copy);
    for (UndirectedGraphNode w : node.neighbors)
       // if (!map.containsKey(w))                /* wrong for self-loop case */
            copy.neighbors.add(dfsClone(w)); //往深度走, 可以看到鮮明的 DFS 特色. 另外 copy.neighbors 是一個 list, 所以可以 add

    return copy;
}
```

2. BFS: Time ~ O(|V|+ |E|), Space ~ O(|V|+ |E|)
我: 以下解說不用看, 直接看代碼就可以了
用一个 Queue 来记录 visited 的 node。每次从 Queue 中取一个 node（Queue 中的 node 已经建立过 copy 并放入 HashMap），访问其所有 neighbor：
如果 visit 过，则直接将其 copy 从 HashMap 中调出放入 copy 的 neighbor list 里；
如果没有 visit 过，则要 create 一个 neighbor copy 并加入 node copy 的 neighbor list 里，然后再将该 node neighbor 放入 Queue 中。
循环直至 Queue 为空。
注：HashMap 记录 key - visited node, value - its copy，避免在 Queue 中加入重复的 node。

```
public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
    if (node == null)   return null;
    UndirectedGraphNode copy = new UndirectedGraphNode(node.label);
    Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();
    map.put(node, copy);
    Queue<UndirectedGraphNode> q = new LinkedList<>();//尼瑪, 左邊是 Queue, 右邊是 LinkedList, 這也可以...
//BFS 要用 queue, 比如典型的 BFS 題 102. Binary Tree Level Order Traversal, Easy 就是用的 queue. Code Ganker 本題(133 題)的 BFS 解法也用的 queue.
    q.add(node);
    while (!q.isEmpty()) {
        UndirectedGraphNode v = q.poll();
        for (UndirectedGraphNode w : v.neighbors) {
            if (map.containsKey(w)) {   /* need to add the visited node to neighbor list */
                map.get(v).neighbors.add(map.get(w));
            } else {
```

```
                UndirectedGraphNode wCopy = new UndirectedGraphNode(w.label);
                map.get(v).neighbors.add(wCopy);
                map.put(w, wCopy);
                q.add(w);//這是把節點一層一層地加到 queue 中(先加一層, 再加下一層, 這裡的「一層」即某個節點的所有 neighbors), 在 queue 的另一端一層一層地取, 由此可以看出鮮明的 BST 特色
            }
        }
    }
    return copy;
}
```

Code Ganker:

这道题是 LeetCode 中为数不多的关于图的题目，不过这道题还是比较基础，就是考察图非常经典的方法：深度优先搜索和广度优先搜索。这道题用两种方法都可以解决，因为只是一个图的复制，用哪种遍历方式都可以。具体细节就不多说了，因为两种方法太常见了。这里恰好可以用旧结点和新结点的 HashMap 来做 visited 的记录。下面是广度优先搜索的代码(我: 已省略)

深度优先搜索的代码如下(我: 已省略)

当然深度优先搜索也可以用递归来实现，代码如下(我: 已省略)(評論中有些人認為這是廣度優先,此評論沒 copy)

这几种方法的时间复杂度都是 O(n)（每个结点访问一次），而空间复杂度则是栈或者队列的大小加上 HashMap 的大小，也不会超过 O(n)。图的两种遍历方式是比较经典的问题了，虽然在面试中出现的不多，但是还是有可能出现的，而且如果出现了就必须做好，所以大家还是得好好掌握哈。

LinkedList 能转化为 Stack 类型吗？LinkedList<UndirectedGraphNode> stack = new LinkedList<UndirectedGraphNode>();在这里初始化的时候。

java 中 LinkedList 实现了栈所需要的函数和功能，所以是可以当成栈来用的哈~

回复 linhuanmars：哈哈，谢谢！

深度优先搜索 的时间复杂度是 O(n)吗？

回复 xiah_sunny：是的，深度和广度搜索的最坏时间都是 O(n)的哈~

134. Gas Station, Medium

http://blog.csdn.net/linhuanmars/article/details/22706553

There are *N* gas stations along a circular route, where the amount of gas at station *i* is gas[i].
You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.
Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.
Note:
The solution is guaranteed to be unique.

Subscribe to see which companies asked this question

Key: travel around the circuit 意思是回到最初的出發點上. 先看下面黃字中我對負序列和正序到的解釋. 然後看 Code Ganker 所說的兩點藍色字(不用看證明), 然後就明白怎麼做了.

History:
E1 直接看的答案.
E2 很快寫好, 本來可以一次通過的, 但粗心將 gas_i – cost_i 寫成了 gas_i + cost_i. 改後即通過. 我暈.
E3 對回憶的內容還不是很確定(後來證明我回憶出的是正確的), 就試著寫, 結果寫好後, 一遍通過. E3 方法跟 key 中一樣, 但寫法有較大差別.

(William 的代碼跟 Code Ganker 基本上是一樣的, William 說說這是貪心法)

Code Ganker:

这是一道具体问题的题目，brute force 的方法比较容易想到，就是从每一个站开始，一直走一圈，累加过程中的净余的油量，看它是不是有出现负的，如果有则失败，从下一个站开始重新再走一圈；如果没有负的出现，则这个站可以作为起始点，成功。可以看出每次需要扫描一圈，对每个站都要做一次扫描，所以时间复杂度是 O(n^2)。代码比较直接，这里就不列举了。

我:

以下說的負序列和正序列是這樣的序列:

負序列是這樣一個序列: 以 上一個負序列(為何一定是負序列? 後面我會講)的終點的下一站為起點開始走, 直到出現負的油量的站(油量定義為 Σ(gas_i – cost_i), 其中求和是對所走過的站求和), 以這個站為終點的序列.

正序列是這樣一個序列: 以 上一個負序列的終點的下一站為起點開始走, 直到走到最初的出發點. 正序列並不一定存在, 但若存在, 最多只有一個, 因為若有兩個, 那麼第一個正序列為何要停下來？別忘了只有油量為負的時候才能停下來.

所以 visualize 地講, 只有兩種可能: 一種是從起點開始, 經過若干個負序列後, 回到最初起點. 另一種是從起點開始, 經過若干個負序列, 再經過一個正負序列, 回到最初起點. 這也是為何前面我說「為何一定是負序列? 後面我會講」.

Code Ganker:

接下来说说如何提高这个算法。方法主要思想是把这个圈划分成一个个的负序列，以及一个正序列（如果存在的话）。从任意一个站出发，我们可以累加油的净余量，如果出现负的，序列结束，开启一个新的，并且证明旧的这个序列的起点不能作为起点，因为会出现负油量，不能继续前进。下面我们证明
不仅这个负序列的起点不能作为起点，负序列中的任意一点都不能作为起点。
证明：假设我们取定负序列中的一个站作为起点，因为一个序列一旦遇到负的净余量就会结束并且开启新的，那么说明在这个起点前的累加结果必然是正数（否则会结束这个序列，则前面不会是这个序列的一部分）。如此我们从当前序列出发必然会使走到序列终点时负的油量更大，本来已经是负的，所以不能去负序列的任意一个结点作为起点。
根据上面的划分方式，我们会把圈分成一段段的序列，而且其中最多只有一个正序列，那就是绕一圈回到起点的那个序列（当然也有可能整个圈是一个正序列，就是油量一直为正，那么我们测的开始点就可以作为起点了）。接下来我们证明
如果将全部油量累计起来(我: 即 Σ(gas_i – cost_i), 求和是從任意一點開始加(總和是個不變量), 加到加到, 這個和可能為負, 但不管, 繼續加, 直到所有 station 加完)，总量为正，那么一定能找到一个起点(我: 這個起點即為正序列的起點)，使得可以走完一圈，也就是一定有解(我: 否則無解, 因為總和是個不變量)。
证明：按照我们之前的划分，整个圈会被划分成有累积量为 s1, s2, ..., sk 的负序列，以及一个正序列拥有油量 sp（这里正序列一定存在因为全部累加和是正的，如果全是负序列那么结果不会是正的）。而且我们知道

s1+s2+...+sk+sp>0 (我: 因為命題中說的全部油量累计起来总量为正)，也就是说 sp>-s1-s2-...-sk。换句话说，如果我们从 sp 对应的站的起点出发，在 sp 对应的序列会一直是正的，并且，当他走到负序列时，因为 sp 的正油量大于所有负油量的总和，所以累加油量会一直正，完整整个圈的行驶。这证明了只要累加油量是正的，一定能找到一个起点来完成任务。
根据上面的两个命题，我们可以来实现代码，需要维护两个量，一个是总的累积油量 total，另一个是当前序列的累计油量 sum，如果出现负的，则切换起点，并且将 sum 置 0。总共是需要扫描所有站一次，时间复杂度是 O(n)。而只需要两个额外变量，空间复杂度是 O(1)。代码如下.

这个题目的优化解法更像是一个数学题，需要定义数学模型并证明命题正确性，比较需要数学逻辑的功底。通过定义的模型以及证明的命题来做一部分贪心。

```
public int canCompleteCircuit(int[] gas, int[] cost) {
    if(gas==null || gas.length==0 || cost==null || cost.length==0 || gas.length!=cost.length)
        return -1;
    int sum = 0; //sum 是当前序列的累计油量
    int total = 0; //total 是总的累积油量
    int pointer = -1;
    for(int i=0;i<gas.length;i++)
    {
        int diff = gas[i]-cost[i];
        sum += diff;
        total += diff;
        if(sum<0)
        {
            sum=0;
            pointer=i;
        }
    }
    return total>=0?pointer+1:-1;
}
```

135. Candy, Hard

http://wlcoding.blogspot.com/2015/03/candy.html?view=sidebar

There are *N* children standing in a line. Each child is assigned a rating value.
You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

Subscribe to see which companies asked this question

Key: 看下面 William 說的掃描三次.

我: 注意理解題意, 不要想太複雜了. 題目只是要求 Children with a higher rating get more candies than their neighbors. 注意是 neighbors, 即只要和自己左右相鄰的比較就可以了(即按 local 的順序給糖, 而不是按 global 的順序給糖)

History:
E1 沒做出來
E2 很快寫好, 但第二遍掃描時, 忘了'且该人的糖果没有后面那人的多'這點, 改後即通過.
E3 幾分鍾寫好, 但第二遍掃描時, 忘了'且该人的糖果没有后面那人的多'這點, 改後即通過.

我: William 的方法和 Code Ganker 差不多(有點小區別), 但 William 代碼比 Code Ganker 的簡潔好懂, 所以以下用 William 的代碼. Code Ganker 說的也沒甚有有用的, 所以沒 copy.

William:

Greedy Algorithm: Time ~ O(3N), Space ~ O(N)
扫描三次：
•从前往后扫，遇到比前面大的人，多分一个糖果给他(我: 多分一個的意思是比前一個人多一個糖果, 此次掃描是處理上坡時之情況), 否則(即沒前一個大)只分一個糖果给他;
•从后往前扫，遇到比后面大的人，且该人的糖果没有后面那人的多 (我: 即糖果數小於等於後面那人的) ，则多分一个糖果给他 (我: 多分一個的意思是比後一個人多一個糖果, 此次掃描是處理下坡時之情況. 比如 ratings = {2, 1}時之情況, 注意此時要給的糖果數是 3 個.)
•再从头扫一遍，累计所有的糖果数。

以上算法可用以下例子來想:
Ratings = 4 2 3 4 1
Candies after first pass = 1 1 2 3 1
Candies after fsecond pass = 2 1 2 3 1
Total candies = 9

```
public int candy(int[] ratings) {
    int n = ratings.length;
    int[] candyNum = new int[n];
    Arrays.fill(candyNum, 1);

    for (int i = 1; i < n; i++)
        if (ratings[i - 1] < ratings[i])    // ascending
         //看看人家[]中都加了空格的:i - 1
            candyNum[i] = candyNum[i - 1] + 1;

    for (int i = n - 2; i >= 0; i--)
         if (ratings[i] > ratings[i + 1] && candyNum[i] <= candyNum[i + 1])     //
descending
            candyNum[i] = candyNum[i + 1] + 1;

    int total = 0;
    for (int i = 0; i < n; i++)
        total += candyNum[i];
    return total;
}
```

我: 我後來又將以上代碼作了點修改(改後可通過), 將後面兩次掃描合並了, Time ~ O(2N), 但代價是代碼沒有以上代碼那樣清新, 時間上也沒有提高多少, 所以還按以上代碼寫比較好. 修改後的代碼就不貼上來了.

136. Single Number, Medium

http://blog.csdn.net/linhuanmars/article/details/22648829
Given an array of integers, every element appears *twice* except for one. Find that single one.
Note:
Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?
Subscribe to see which companies asked this question

key: 兩個相等的數異或(a^b 相當於 a!=b,詳見 Java 書 p116)結果為 0. 故可以將數組中所有元素都異或起來, 相等的數就抵消了, 最終剩下的就是只出現一次的數. 注意輸入數組中的 相等的數 不一定是挨著的, 但異或應該滿足交換律(沒證), 所以沒有影响.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過, 代碼跟答案一樣.
E3 幾分鍾寫好, 本來可以一次通過的, 結果不小心將 nums[i]寫為了 i, 改後即通過.

我: 以下第一種方法基本上跟 137 題是一樣的, 看看就可以. 第二種方法很巧, 要掌握的是第二種方法.
Code Ganker:
这道题目跟 Single Number II 比较类似，区别只是这道题目每个元素出现两次，而不是三次。我们仍然可以按照 Single Number II 中的两种方法来，都只是把取余 3 改成取余 2 即可。我们就列举一下第二种方法的代码如下.

第一種方法(不用):
```
public int singleNumber(int[] A) {
    int[] digits = new int[32];
    for(int i=0;i<32;i++)
    {
        for(int j=0;j<A.length;j++)
        {
            digits[i] += (A[j]>>i)&1;
        }
    }
    int res = 0;
    for(int i=0;i<32;i++)
    {
        res += (digits[i]%2)<<i;
    }
    return res;
}
```

上面方法的时间复杂度仍然是 O(n)，空间是一个 32 个元素的数组，是 O(1)的复杂度。

按照上面的方法，虽然空间复杂度是 O(1)的，不过还是需要一个 32 个元素的数组。还有另一种方法是利用每个元素出现两次，以及位操作异或的性质来解决这个问题。因为两个相同的元素异或(我: a^b 相當於 a!=b, 詳見 Java 書 p116)结果是 0，利用这个特点我们可以对所有数组元素进行异或，如果出现两次的元素就会自行抵消，而最终剩下的元素则是出现一次的元素。这个方法只需要一次扫描，即 O(n)的时间复杂度，而空间上也不需要任何额外变量，比起上面的方法更优。代码如下：

第二種方法(用它):

```
public int singleNumber(int[] A) {
    if(A==null || A.length==0)
        return 0;
    int res = A[0]; //注意不要寫成 int res = 0 並在下面的 for 中使初始為 i=0, 因為這樣的話就相當於給 A 多引入了一個 0 元素
    for(int i=1;i<A.length;i++)
    {
        res ^= A[i];
    }
    return res;
}
```

上面的方法实现非常简练，不过也相当取巧，可能没有准备过比较难在面试中想到。相对而言第一种方法是比较通用的，无论出现多少次都是适用的。第二种方法属于对于出现两次这种特殊情况才能使用，不过方法巧妙，有点体现位运算的精髓，所以个人还是挺喜欢的哈。

137. Single Number II, Medium

http://blog.csdn.net/linhuanmars/article/details/22645599

Given an array of integers, every element appears *three* times except for one. Find that single one.
Note:
Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?
Subscribe to see which companies asked this question

Key: 位運算. 如果某個數出現三次, 則該數的每一位出現 1 的次數也會是 3 的倍數. 方法就是將所有數的某位都加走來, 然後取餘 3, 結果就是單身狗的該位數. a 的第 i 位可這樣得到: (a>>i)&1. 若要將某數 d(d 為 0 或 1)加到 a 的第 i 位上, 可以這樣: a += d<<i. 可定義一個長度為 32 的數組來用, 這也算是 O(1)空間複雜度. 若 digits[i]是用數組表示的一個二進製數, 則將其轉化為一個真正的二進製數的方法為: for(int i...) res += digits[i] << i. 這樣做比每次乘 2 的冪次方要直觀好寫.

Corner case: input [1], output 1. 負數元素(負數的表示見 Java p55, 本題代碼中不用專門考慮負數).

History:
E1 直接看的答案.

E2 改了幾次後通過, 方法是用的答案的方法, 但在代碼實現上很不一樣, 我 E2 的代碼沒有答案的好.

E3 最開始是按十進製寫的, 即 將所有元素的十進製的某位數加起來 然後取餘 3. 這樣的問題是, 任何數取餘 3 後, 都只能等於 0, 1, 2. 即最終結果的每位數只能為 0, 1, 2, 而最終結果是十進製的, 所以這樣做不對. 後來改為按二進製寫, 即通過. E3 的代碼也是 按求和取餘 3 來做的, 但俱體寫法跟 Code Ganker 不一樣, 尤其是 E3

沒像 Code Ganker 那樣用一個額外的數組, 故 E3 代碼實際上比 Code Ganker 的好, 但由於 Code Ganker 代碼中對位的操作要記住, 且 Code Ganker 比 E3 的代碼直觀好懂, 故還是用 Code Ganker 的代碼.

这个题比较直接的想法是用一个 HashMap 对于出现的元素进行统计，key 是元素，value 是出现个数，如果元素出现三次，则从 HashMap 中移除，最后在 HashMap 剩下来的元素就是我们要求的元素（因为其他元素都出现三次，有且仅有一个元素不是如此）。这样需要对数组进行一次扫描，所以时间复杂度是 O(n)，而需要一个哈希表来统计元素数量，总共有(n+2)/3 个元素，所以空间复杂度是 O((n+2)/3)=O(n)。这个方法非常容易实现，就不列举代码了。
在 LeetCode 的题目中要求我们不要用额外空间来实现，也就是 O(1)空间复杂度。实现的思路是基于数组的元素是整数，我们通过统计整数的每一位来得到出现次数。我们知道如果每个元素重复出现三次，那么每一位出现 1 的次数也会是 3 的倍数，如果我们统计完对每一位进行取余 3，那么结果中就只剩下那个出现一次的元素。总体只需要对数组进行一次线性扫描，统计完之后每一位进行取余 3 并且将位数字赋给结果整数，这是一个常量操作（因为整数的位数是固定 32 位），所以时间复杂度是 O(n)。而空间复杂度需要一个 32 个元素的数组，也是固定的，因而空间复杂度是 O(1)。代码如下.
这个题目主要是对整数数组的操作，用到的位统计是整数中经常使用的技巧，利用位的固定性来省去一定的时间或者空间复杂度。
请教下
digits[i] += (A[j] >> i) & 1;
这个最后面为什么加&1 ？ 直接把位置上的 0、1 数字相加不可以吗？
回复 yuzouzhijie：这里我们只需要看第 i 位的数据，所以是移位后取最低位~ (我: %1 就是把高位全弄成 0)

我有个疑问哦，"不要用额外空间来实现"这个要求，是等于 要求 O(1)空间复杂度 吗？
回复 yuzouzhijie：是的哈~

我用 hashmap 写的，遍历 hashmap 的时候有个疑问，我是用 for 遍历的，为什么我看别人都用 iterator 然后再复制给 map.entery 遍历呢？一般遍历 hashmap 用什么
回复 u012296380：这个其实都是可以的，用你熟练的就可以 ~ 不过有时候只要访问一部分数据就要用 iterator 了 ~

Code Ganker 的代碼(用它):

```
public int singleNumber(int[] A) {
    int[] digits = new int[32]; //int 為 32 bit, 見 java 書 p68.
    for(int i=0;i<32;i++) //i 代表第 i 位
    {
        for(int j=0;j<A.length;j++) //j 代表 A 的第 j 個元素, 即有 A[j].
        {
            digits[i] += (A[j]>>i)&1;
//&&和&之區別見 Java 書 p116.
//(A[j]>>i)&1 結果為: A[j]的第 i 位保持(是 0 就是 0, 是 1 就是 1), 但高位全變為 0. digits[i]為 A 數組中所有元素的第 i 位的總和. 注意>>和>>>的區別, 見 java p1300. 注意 A[j]>>i 後, A[j]的值是不變的.
//另外, 此式中必須用+=, 否則用=通不過, 因為這是一個求和的過程.
        }
    }
    int res = 0; //最後要返回的就是 res, 即下面要使 res 等於那個單身狗的值
    for(int i=0;i<32;i++)
    {
        res += (digits[i]%3)<<i; // digits[i]%3 即為單身狗的第 i 位上的數字, (digits[i]%3)<<i 是將其移回原位(第 i 位).  res += (digits[i]%3)<<i 即把每一位都加入到 res 中對應的位上, 所以 res 的值就等於那個單身狗.
    }
    return res;
}
```

E3 代碼(不用, 但可參考, 實際上比 Code Ganker 的好, 因為空間為 O(1), 沒用數組):

```
public int singleNumber(int[] nums) {
    if(nums == null || nums.length == 0) return 0;
    if(nums.length == 1) return nums[0];
    int resDigit = 0, res = 0, twoPower = 1;

    for(int pos = 0; pos < 32; pos++) {
        for(int i = 0; i < nums.length; i++) {
            int lastDigit = nums[i] % 2;
            resDigit += lastDigit;
            nums[i] = nums[i] >>> 1;
        }
        resDigit = resDigit % 3;
        res = resDigit * twoPower + res;
        twoPower *= 2;
        resDigit = 0;
    }

    return res;
}
```

138. Copy List with Random Pointer, Hard

http://blog.csdn.net/linhuanmars/article/details/22463599

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.
Return a deep copy of the list.
Subscribe to see which companies asked this question

Key: 本題只要知道方法了, 其實也不難. 想避免使用额外空间，我们只能通过利用链表原来的数据结构来存储结点. 对链表进行三次扫描. 第一次掃描: 把新鏈表插到舊鏈表中去, 使新舊更替. 第二次掃描: 把旧结点的随机指针赋给新节点的随机指针, 即 node.next.random (node 為舊節點, node.next 為新節點) = node.random.next (我: 為何不是 node.random? 因為 node.random 還是舊節點, node.random.next 才是對應的新節點). 第三次扫描: 把链表拆成两个. 注意最後要將舊 list 恢復成原來的樣子.

History:
E1 直接看的答案.
E2 第一遍沒通過, 然後改了一句話就通過了, 第一遍時范了一個很容易范的錯誤, 即第二次掃描時忘了考慮 node.random == null 時之情況.
E3 的方法跟 key 中一樣, 但改了好幾次後才通過, 原因是范了兩個錯, 一是第二次掃描時忘了考慮 node.random == null 時之情況(同 E2), 二是最後沒將舊 list 恢復成原來的樣子(我 E3 時不知道還有這個要求).

我: 注意每個 node 有三個 member: int label, Node next, Node random. 注意 label 就相當於其它鏈表題中的 val, 不要誤以為 label 就是那個隨機指針.

以下來自: http://www.itzhai.com/java-based-notebook-the-object-of-deep-and-shallow-copy-copy-copy-implement-the-cloneable-interface-serializing-deep-deep-copy.html

深复制与浅复制：

浅复制(shallow clone)：

被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

深复制(deep clone)：

被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了一遍。

我: 即 deep clone 把將引用的對象(如本題中的 random 所指的對象)也復製了. 而 shallow clone 不會復製引用的對象, 所以引用的還是之前的老對象. 比如本題中, 若是 shallow clone, 則新鏈表中的每個節點的 random 會指向老鏈表中的節點. 若是 deep clone, 則指向新鏈表中的節點.

我: 以下 Code Ganker 介紹了兩種方法, 我記住第二種, 但第一種也看看(E2 和 E3 沒時間看, 其實可以不看). William 也是用的這兩種方法.

Code Ganker:

这是一道链表操作的题目，要求复制一个链表，不过链表的每个结点带有一个随机指针，指向某一个结点。我们先介绍一种比较直接的算法，思路是先按照复制一个正常链表的方式复制，复制的时候把复制的结点做一个 HashMap，以旧结点为 key，新节点为 value。这么做的目的是为了第二遍扫描的时候我们按照这个哈希表把结点的随机指针接上。这个算法是比较容易想到的，总共要进行两次扫描，所以时间复杂度是 O(2*n)=O(n)。空间上需要一个哈希表来做结点的映射，所以空间复杂度也是 O(n)。代码如下.

```
public RandomListNode copyRandomList(RandomListNode head) {
    if(head == null)
        return head;
    HashMap<RandomListNode,RandomListNode> map = new HashMap<RandomListNode,RandomListNode>();
    RandomListNode newHead = new RandomListNode(head.label);
    //以下(包括 while)是將新舊鏈表放入 map 中, 新鏈表中只包含了老鏈表中的 label 和 next(next 是通過 pre 指針實現的), 不包括 random.
    map.put(head,newHead);
    RandomListNode pre = newHead;  //pre 是新鏈表的指針, pre 的作用是把新鏈表串起來
    RandomListNode node = head.next; //node 是老鏈表的指針
    while(node!=null)
    {
        RandomListNode newNode = new RandomListNode(node.label);
        map.put(node,newNode);
        pre.next = newNode;//pre 的作用是把新鏈表串起來
        pre = newNode;
        node = node.next;
```

```
    }
    //以下(包括 while)是將舊鏈表的 random 也弄到新鏈表去
    node = head;
    RandomListNode copyNode = newHead;
    while(node!=null)
    {
        copyNode.random = map.get(node.random); //map 的作用就是查詢 node.random 所對應的新鏈表節點
        copyNode = copyNode.next; //注意新鏈表在前面就已經串好了
        node = node.next;
    }
    return newHead;
}
```

那么有没有办法可以不用额外空间来完成这个任务呢？还是有的，前面我们需要一个哈希表的原因是当我们访问一个结点(我: 新鏈表中的節點)时可能它的随机指针指向的结点还没有访问过，结点还没有创建，所以我们需要线性的额外空间。想避免使用额外空间，我们只能通过利用链表原来的数据结构来存储结点。基本思路是这样的，对链表进行三次扫描，第一次扫描对每个结点进行复制，然后把复制出来的新节点接在原结点的 next，也就是让链表变成一个重复链表，就是新旧更替；第二次扫描中我们把旧结点的随机指针赋给新节点的随机指针，因为新结点都跟在旧结点的下一个，所以赋值比较简单，就是 node.next.random (node 為舊節點, node.next 為新節點) = node.random.next (我: 為何不是 node.random? 因為 node.random 還是舊節點, node.random.next 才是對應的新節點)，其中 node.next 就是新结点，因为第一次扫描我们就是把新结点接在旧结点后面。现在我们把结点的随机指针都接好了，最后一次扫描我们把链表拆成两个，第一个还原原链表，而第二个就是我们要求的复制链表。因为现在链表是旧新更替，只要把每隔两个结点分别相连，对链表进行分割即可。这个方法总共进行三次线性扫描，所以时间复杂度是 O(n)。而这里并不需要额外空间，所以空间复杂度是 O(1)。比起上面的方法，这里多做一次线性扫描，但是不需要额外空间，还是比较值的。实现的代码如下：

```
public RandomListNode copyRandomList(RandomListNode head) {
    if(head == null)
        return head;
    //以下是第一次掃描: 把新鏈表插到舊鏈表中去, 使新舊更替
    RandomListNode node = head;
    while(node!=null)
    {
        RandomListNode newNode = new RandomListNode(node.label);
        //以下兩句是把 newNode 插到 node 後面
        newNode.next = node.next;
        node.next = newNode;
```

```
        //以下是 node 跳到下一個舊節點
        node = newNode.next;
    }
    //以下是第二次掃描: 把旧结点的随机指针赋给新节点的随机指针
    node = head;
    while(node!=null)
    {
        if(node.random != null) //當 node.random == null 時, node.next.random 自動就是默認值 null. E2 和 E3 都沒考慮到 node.random 為 null 時之情況
            node.next.random = node.random.next; //把旧结点的随机指针赋给新节点的随机指针, 看前面講解中我的注釋
        node = node.next.next;
    }
    //第三次扫描: 把链表拆成两个
    RandomListNode newHead = head.next;
    node = head;
    while(node != null)
    {
        RandomListNode newNode = node.next;
        node.next = newNode.next; //此句是將舊 list 恢復為原來的樣子. 注意別把此句放到下面的 if 中去了, 否則分開的舊鏈表末尾沒有 null
        if(newNode.next!=null)
            newNode.next = newNode.next.next;
        node = node.next;
    }
    return newHead;
}
```

上面介绍了两种方法来解决这个问题，第二种方法利用了原来的链表省去了额外空间，虽然多进行一次扫描，不过对时间复杂度量级没有影响，还是对算法有提高的。这个题目算是比较有难度的链表题目，既有基本操作，也需要一些算法思想。

139. Word Break, Medium

http://blog.csdn.net/linhuanmars/article/details/22358863

Given a string *s* and a dictionary of words *dict*, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

For example, given
s = "leetcode",
dict = ["leet", "code"].
Return true because "leetcode" can be segmented as "leet code".
Subscribe to see which companies asked this question

Key: 動態歸劃. 用數組 res[i]表示 s 的 '前 i 個字符組成的子串 (即 s[0, ...i])' 能否用字典中的詞表示出來(即若輸入字符串為 s[0, ...i]時本題的結果). 若已知 res[0, … i-1], 如何求 res[i]? 方法是: 若存在一個 j, 使得以下一句成立:
" res[j - 1]為 true, 且 s[j, ...i](包括 i 在內) 這個子串在字典中", 則 res[i]為 true. 下面 Code Ganker 說的可以不看.

History:
E1 直接看的答案.

E2 很快寫好, 本來可以一次通過的, 但不小心數組 index 越界了, 改後即通過. E2 的代碼比 Code Ganker 的更好懂. 後來(E3 時)發現 E2 並不是按 key 中的 res[i]寫的，且沒有我 E3 的直接好懂，故以後不用 E2 代碼

E3 最開始用遞歸試試，代碼能對題目中例子結出正確答案，結果在 OJ 中果然超時．後來又改用動態規劃，最開始用的 res[i]來表示 s[0,...i]能否分成 wordDict 中的詞，但發現這樣無法表示 s 中取 0 個字符時能否分成 wordDict 中的詞．故又改為用 res[i]來表示 's 的前 i 個字符' 能否分成 wordDict 中的詞, 然後就通過了．多道題目的經驗表示：動態規劃中，要用 res[i]來表示 's 的前 i 個字符' 能否怎麼樣；而不要用 res[i]來表示 s[0,...i]能否怎麼樣，因為後者沒法表示 's 的前 0 個字符' 能否怎麼樣.

我: 我以前有個錯誤的認識, 就是 本輪循環要覆蓋上一輪的原數組 才算動態歸劃, 現在發現其實不一定要覆蓋原數組(比如本題). 動態歸劃的思想就是 通過保存歷史結果來避免重復計算, 所以不一定要覆蓋原數組.

Code Ganker:

这道题仍然是动态规划的题目，我们总结一下动态规划题目的基本思路。首先我们要决定要存储什么历史信息以及用什么数据结构来存储信息。然后是最重要的递推式，就是如从存储的历史信息中得到当前步的结果。最后我们需要考虑的就是起始条件的值。

接下来我们套用上面的思路来解这道题。首先我们要存储的历史信息 res[i]是表示到字符串 s 的第 i 个元素为止能不能用字典中的词来表示，我们需要一个长度为 n(我: n 為 s 的長度)的布尔数组来存储信息。然后假设我们现在拥有 res[0,...,i-1]的结果，我们来获得 res[i]的表达式。思路是对于每个以 i 为结尾的子串(我: 即下面公式中的 s.substring[j, i+1])，看看他是不是在字典里面以及他之前的元素对应的 res[j]是不是 true，如果都成立，那么 res[i]为 true，写成式子是

(我: 以下公式的 res[j]應該為 res[j-1]. 另注意 substring(a, b)返回的是 a 到 b-1 的 substring, 詳見 Java 書 p418. 另注意下式中的 U 表示或)

$$\bigcup_{j=0}^{i} res[j] \;\&\&\; s.substring[j, i+1] \in dict$$

假设(我: s 中)总共有 n 个字符串，并且字典是用 HashSet 来维护，那么总共需要 n 次迭代，每次迭代需要一个取子串的 O(i)操作，然后检测 i 个子串，而检测是 constant 操作。所以总的时间复杂度是 O(n^2)（i 的累加仍然是 n^2 量级），而空间复杂度则是字符串的数量，即 O(n)。代码如下.

动态规划的题目在 LeetCode 中占有相当的比例，不过却没有什么通法，因为每道题会有不同的性质和获取信息的角度。但是总体来说基本思路就如同我上面介绍的那样，根据步骤出来之后基本上问题也就解决了，大家可以多练习熟悉一下哈。

我: Code Ganker 的原代碼是在 j 循環以外寫的 StringBuilder str = new StringBuilder(s.substring(0,i+1)), 並寫了一個不太好看懂的 str.deleteCharAt(0), 來刪前 j 個元素. 他以為這樣可以減少時間複雜度, 但有人在評論中指出, 這樣並不能減少時間複雜度, 不管是 Code Ganker 的原代碼, 還是以下的 評論中他們提出的代碼, 在考慮了 substring 函數的時間後, 時間複雜度都是 O(n^3). 所以本題用以下的 評論中他們提出的代碼(好懂些). 原代碼我看了好一會了才看懂, 所以也貼在後面, 但可以不看, 貼出來只是不想讓我的那此注釋白寫了. 評論太長, 不貼出來了. 另外, William 的代碼也跟以下的 評論中他們提出的代碼 差不多, 但 William 說時間複雜度都是 O(n^2), 應該是沒考慮 substring 函數的時間.

```
 public boolean wordBreak(String s, Set<String> dict) {//注意若輸入的就是 Set<String>, 則不用把它轉換成 HashSet<String>, 直接用就是了. 這跟 120 題中的輸入 List<List<Integer>>一樣．（見首頁的實參和形參）
        if(s==null || s.length()==0){
            return false;
        }
        boolean[] res = new boolean[s.length()+1];//注意下面多加了個 res[0]這個元素, 它不表示甚麼意思, 只起初始的作用, 所以總共需要分配 s.length()+1 個空間給 res. 由於開頭多了一個元素, 故 res[i]即表示 s 的第 i-1 个元素为止能不能用字典中的词来表示
        res[0]=true;
        for(int i=0;i<s.length();i++){
            for(int j=0;j<=i;j++){
                String str = s.substring(j, i+1);
                if(res[j]&&dict.contains(str)){ //Code Ganker 的公式中為 res[j-1] (見公式上面我的糾正), 但此處為何成了 res[j]? 還是那個原因:res[i]即表示 s 的第 i-1 个元素为止能不能用字典中的词来表示
                    res[i+1]=true; //為何是 res[i+1]而不是 res[i]? 見上面我說的
                    break;
                }
            }
        }
        return res[s.length()]; //注意不是 res[s.length()-1]
    }
```

Code Ganker 的原代碼(不用看):

```
public boolean wordBreak(String s, Set<String> dict) {
    if(s==null || s.length()==0)
        return true;
    boolean[] res = new boolean[s.length()+1]; //注意下面多加了個 res[0]這個元素, 它不表示甚麼意思, 只起初始的作用, 所以總共需要分配 s.length()+1 個空間給 res. 故 res[i+1]即表示上面講解中的 res[i]
    res[0] = true;
    for(int i=0;i<s.length();i++)
    {
```

```
        StringBuilder str = new StringBuilder(s.substring(0,i+1)); //為何是從 0 開始? 看下面我的注釋就知道了
        for(int j=0;j<=i;j++)
        {
            if(res[j] && dict.contains(str.toString()))
            {
                res[i+1] = true;
                break;
            }
            str.deleteCharAt(0); //j 等於多少, 就刪了多少+1 那麼多個, 所以每輪 j 循環開始時, s 的就已經被刪了 j 個
        }
    }
    return res[s.length()];
}
```

E2 的代碼 (不用):

```
    public boolean wordBreak(String s, Set<String> wordDict) {
        int n = s.length();
        boolean[] res = new boolean[n];

        for(int i = 0; i < n; i++) {
            for(int j = 0; j <= i; j++) {
                String cur = s.substring(j, i + 1);
                boolean pre = j - 1 >= 0 ? res[j - 1] : true;
                res[i] = res[i] || ( pre && wordDict.contains(cur));
            }
        }

        return res[n - 1];
    }
```

E3 的代碼(用它):

```
    public static boolean wordBreak(String s, Set<String> wordDict) {
        if(s.length() == 0) return true;
        if(wordDict.isEmpty()) return false;
        int n = s.length();

        boolean[] res = new boolean[n + 1]; //res[i]表示  's 前 i 個字符組成的子串' 能否分成 wordDict 中的詞
        res[0] = true;

       //以下是考查子串 s.substring(l, r)是否在 wordDict 中，以及 s.substring(l, r)左邊的那堆字符 能否分為分成 wordDict 中的詞
        for(int r = 1; r <= n; r++) {
            for(int l = 0; l < r; l++) {
                res[r] = res[r] || (res[l] && wordDict.contains(s.substring(l, r)));
            }
        }
```

```
        return res[n];
    }
```

140. Word Break II, Hard

http://blog.csdn.net/linhuanmars/article/details/22452163

Given a string *s* and a dictionary of words *dict*, add spaces in *s* to construct a sentence where each word is a valid dictionary word.
Return all such possible sentences.
For example, given
s = "catsanddog",
dict = ["cat", "cats", "and", "sand", "dog"].
A solution is ["cats and dog", "cat sand dog"].
Subscribe to see which companies asked this question

Key:

E3 的方法見 History 中, 建議用我 E3 的代碼.

Code Ganker 的方法: 遞歸. 本題不難. 寫一個 helper(String s, Set<String> dict, int start, String item, List<String> res)，作用是: 把 s 的從 start 開始的子串按 dict 分好, 接到 item 中去(item 是之前得到的結果), 若 start 大於 s 的長度, 就將 item 放入 res 中.

注意：LeetCode 的 test case 中有一个很长的 string 容易造成超时，在搜索所有 partition 之前，需要判断整个 string 是否 breakable 再决定是否进行。

History:
E1 直接看的答案.

E2 沒多久就寫好, 小改後即通過.

E3 沒多久就寫好, 小改一次(即加了句若 resRest 為空則結果中要加入 rear)後即通過. E3 也是用的遞歸, 但寫起來跟 Code Ganker 有較大的不同. E3 是遞歸調用 wordBreak 自己. wordBreak 中, 從 s 的右邊往左邊看, 先在最右端扣出一個子串(叫它小明), 若 wordDict 中含此小明 且 小明之左的部分是 breakable, 則調用 wordBreak(小明之左的部分, wordDict), 將它返回的所有 string 右邊加上小明即可. 當然要使用本題 I 中的函數來判斷小明之左的部分是否為 breakable. 注意 Code Ganker 的代碼中, 若題目沒有那個較長的 non-breakable 的 test case, 則不用調用本題 I 中的函數. Code Ganker 的代碼也不用保護現場. 但我 E3 的代碼比較直接好寫, 空間上也沒多佔用多少, 故以後還是建議用我 E3 的代碼.

我: 本題要掌握第一種方法(遞歸), 第二種方法(動態歸劃)不用看.

Code Ganker 後面說的:
可以看出，用动态规划的代码复杂度要远远高于 brute force 的解法，而且本质来说并没有很大的提高，甚至空间上还是一个暴涨的情况。所以这道题来说并不是一定要用动态规划，有一个朋友在面 Amazon 时遇到这道题，面试官并没有要求动态规划，用 brute force 即可，不过两种方法时间上和空间上的优劣还是想清楚比较好，面试官可能想听听理解。实现的话可能主要是递归解法。

还有一点需要指出的是，上面的两个代码放到 LeetCode 中都会超时，原因是 LeetCode 中有一个非常 tricky 的测试 case，其实是不能 break 的，但是又很长，出现大量的记录和回溯，因此一般通过这个的解法是把 Word Break 先跑一遍，判断是不是能 break，如果可以再跑上面的代码。这样做法其实比较傻，但是为了过这个只能这样了，这一点我觉得 LeetCode 没必要把超时设得这么严格，实际意义不大，只是把 AC 率给拉了下来哈。我: William 也這樣說了:
William:

注意：LeetCode 的 test case 中有一个很长的 string 容易造成超时，在搜索所有 partition 之前，需要判断整个 string 是否 breakable 再决定是否进行。

我:

我處理以上 test case 的做法是: 將 139. Word Break, Medium 中的 wordBreak(String s, Set<String> dict) 函數拷過來, 將其改名為 canBreak(String s, Set<String> dict), 然後在本題的 wordBreak 函數中 if(s == null || s.length() == 0) return res; 的後面加一句: if(!canBreak(s, wordDict)) return res; 就可以通過了.

Code Ganker:

这道题目要求跟 Word Break 比较类似，不过返回的结果不仅要知道能不能 break，如果可以还要返回所有合法结果。一般来说这种要求会让动态规划的效果减弱很多，因为我们要在过程中记录下所有的合法结果，中间的操作会使得算法的复杂度不再是动态规划的两层循环，因为每次迭代中还需要不是 constant 的操作，最终复杂度会主要取决于结果的数量，而且还会占用大量的空间，因为不仅要保存最终结果，包括中间的合法结果也要一一保存，否则后面需要历史信息会取不到。所以这道题目我们介绍两种方法，一种是直接 brute force 用递归解，另一种是跟 Word Break 思路类似的动态规划。

对于 brute force 解法，代码比较简单，每次维护一个当前结果集，然后遍历剩下的所有子串，如果子串在字典中出现，则保存一下结果，并放入下一层递归剩下的字符。思路接近于我们在 N-Queens 这些 NP 问题中经常用到的套路。代码如下：

```
public ArrayList<String> wordBreak(String s, Set<String> dict) {
    ArrayList<String> res = new ArrayList<String>();
    if(s==null || s.length()==0)
        return res;
    helper(s,dict,0,"",res);
    return res;
}
//helper 的作用是: 把 s 的從 start 開始的子串按 dict 分好, 接到 item 中去(item 是之前得到的結果), 若 start 大於 s 的長度, 就將 item 放入 res 中.
private void helper(String s, Set<String> dict, int start, String item, ArrayList<String> res)
{
    if(start>=s.length())
    {
        res.add(item);
        return;
    }
    StringBuilder str = new StringBuilder();
    for(int i=start;i<s.length();i++)
    {
        str.append(s.charAt(i));
        if(dict.contains(str.toString()))
        {
```

```
            String newItem = item.length()>0?(item+" "+str.toString()):str.toString();
            helper(s,dict,i+1,newItem,res);
        }
    }
}
```

接下来我们列出动态规划的解法，递推式跟 Word Break 是一样的，只是现在不只要保存 true 或者 false，还需要保存 true 的时候所有合法的组合，以便在未来需要的时候可以得到。不过为了实现这点，代码量就增大很多，需要一个数据结构来进行存储，同时空间复杂度变得非常高，因为所有中间组合都要一直保存。时间上还是有提高的，就是当我们需要前面到某一个元素前的所有合法组合时，我们不需要重新计算了。不过最终复杂度还是主要取决于结果的数量。代码如下：

```
class Interval {
    int start;
    int end;
    public Interval(int start, int end) {
        this.start = start; this.end = end;
    }
    public Interval(Interval i) {
        start = i.start;
        end = i.end;
 }
}
ArrayList<ArrayList<Interval>> deepCopy(ArrayList<ArrayList<Interval>> paths) {
    if (paths==null) return null;
    ArrayList<ArrayList<Interval>> res = new ArrayList<ArrayList<Interval>>(paths.size());
    for (int i=0; i<paths.size(); i++) {
     ArrayList<Interval> path = paths.get(i);
     ArrayList<Interval> copy = new ArrayList<Interval>(path.size());
        for (int j=0; j<path.size(); j++) {
         copy.add(new Interval(path.get(j)));
     }
     res.add(copy);
    }
    return res;
}
public ArrayList<String> wordBreak(String s, Set<String> dict) {
    ArrayList<String> res = new ArrayList<String>();
```

```
    if (s==null || s.length()==0) return res;
    Map<Integer, ArrayList<ArrayList<Interval>>> dp = new HashMap<Integer,
ArrayList<ArrayList<Interval>>>();
    dp.put(0, new ArrayList<ArrayList<Interval>>());
    dp.get(0).add(new ArrayList<Interval>());
    for (int i=1; i<=s.length(); i++) {
        for (int j=0; j<i; j++) {
            String cur = s.substring(j, i);
            ArrayList<ArrayList<Interval>> paths = null;
            if (dp.containsKey(j) && dict.contains(cur)) {
                paths = deepCopy(dp.get(j));
                Interval interval = new Interval(j, i);
                for (ArrayList<Interval> path:paths) {
                    path.add(interval);
                }
            }
            if (paths != null) {
                if (!dp.containsKey(i)) {
                    dp.put(i, paths);
                }
                else {
              dp.get(i).addAll(paths);
             }
            }
        }
    }
    if (!dp.containsKey(s.length())) {
        return res;
    }
    ArrayList<ArrayList<Interval>> paths = dp.get(s.length());
    for (int j=0; j<paths.size(); j++) {
        ArrayList<Interval> path = paths.get(j);
        StringBuilder str = new StringBuilder();
```

```
        for (int i=0; i<path.size(); i++) {
            if (i!=0) {str.append(" ");}
            int start = path.get(i).start;
            int end = path.get(i).end;
            str.append(s.substring(start, end));
        }
        res.add(str.toString());
    }
    return res;
}
```

可以看出，用动态规划的代码复杂度要远远高于 brute force 的解法，而且本质来说并没有很大的提高，甚至空间上还是一个暴涨的情况。所以这道题来说并不是一定要用动态规划，有一个朋友在面 Amazon 时遇到这道题，面试官并没有要求动态规划，用 brute force 即可，不过两种方法时间上和空间上的优劣还是想清楚比较好，面试官可能想听听理解。实现的话可能主要是递归解法。
还有一点需要指出的是，上面的两个代码放到 LeetCode 中都会超时，原因是 LeetCode 中有一个非常 tricky 的测试 case，其实是不能 break 的，但是又很长，出现大量的记录和回溯，因此一般通过这个的解法是把 Word Break 先跑一遍，判断是不是能 break，如果可以再跑上面的代码。这样做法其实比较傻，但是为了过这个只能这样了，这一点我觉得 LeetCode 没必要把超时设得这么严格，实际意义不大，只是把 AC 率给拉了下来哈。我: William 也這樣說了:

William:

注意：LeetCode 的 test case 中有一个很长的 string 容易造成超时，在搜索所有 partition 之前，需要判断整个 string 是否 breakable 再决定是否进行。

我:

我處理以上 test case 的做法是: 將 139. Word Break, Medium 中的 wordBreak(String s, Set<String> dict) 函數拷過來, 將其改名為 canBreak(String s, Set<String> dict), 然後在本題的 wordBreak 函數中 if(s == null || s.length() == 0) return res; 的後面加一句: if(!canBreak(s, wordDict)) return res; 就可以通過了.

Code Ganker 後面的評論:

神，看了你不少帖子，请问你的变量 res 一般是什么单词的缩写啊，指什么意思呢？rest?
还有，神，你代码怎么都不注释啊？虽说好的代码不需注释。
最后，神，面试一般需要写几个方法就可以了，还是完整的类写写来（包括 public static void main?）？面试一般需要自己主动说出时间复杂度和空间复杂度吗？
谢谢神

回复 youngtoleetcode：res 一般是 result 的意思，就是结果的意思哈 ~
一般来说面试时没有时间写注释的，这是按照面试环境来写的 ~ 面试会一边写一边解释给面试官听，所以很少写注释。
面试的问题一般都是比较独立的功能问题，所以写方法也就是函数就可以了 ~ 空间复杂度和时间复杂度主动说出来是最好的，不要想着去逃避哈 ~ 因为不说面试官还是会问的 ~

大神，关于递归什么时候需要“保护现场”，小弟我有自己的理解，但是不知道对不对，能不能再解释一下呢？

回复 qweyxy_2：就是维护递归时维护一个变量，而这个变量是需要全局变化的，也就是递归前把元素加进去，递归出来时需要把它移除，否则在出这层递归时就会变化了，所以要还原一下~

回复 linhuanmars：哦， 我好像明白了， 因为这里每次进入下一次递归是用的 newItem 这个新的 String， 而不是一直用 item， 所以不用回溯， 不知是不是理解正确了

回复 ChiBaoNeLiuLiuNi：是的哈~ 你的理解是对的~

回复 linhuanmars：请问大神， brute force 解法中并没有维护现场， 是不是因为每次递归都是 new 一个新的 StringBuilder str， 而这个 str 并不是全局变量， 所以无需回溯呢？ 跟大神讨论确认一下哈， 谢谢。

楼主问一下，brute force 的解法和 dp 解法的时间和空间复杂度分别是多少？我没搞清楚。。多谢！

回复 jhbxlx：从这道题来说，因为是一个 NP 问题，结果数量有可能就是指数量级的，所以复杂度来说都是指数的，用 DP 只不过是循环时间复杂度减少了，但是到了赋值给结果的时候还是指数量级的 ~ 像这种题目就是时间和空间复杂度都取决于结果数量的量级哈 ~

回复 linhuanmars：嗯，好的，谢谢

E3 的代碼:

```
public static List<String> wordBreak(String s, Set<String> wordDict) {
    List<String> res = new ArrayList<>();
    if(s.length() == 0 || wordDict.isEmpty()) return res;
    int n = s.length();
    boolean[] can = canBreak(s, wordDict);
    if(!can[n]) return res; //不要此句也能通過, 因為對於那個 non-breakable 的長輸入, 以下的 if(can[i]...) 已經判斷了

    for(int i = 0; i <= n; i++) {
        String rear = s.substring(i, n); //rear = s[i, n-1]
        String rest = s.substring(0, i); //rest = s[0, i-1]

        if(can[i] && wordDict.contains(rear)) { //若不要 if 中的那個 can[i], 則通不過
            List<String> resRest = wordBreak(rest, wordDict);
            if(resRest.isEmpty()) res.add(rear);
            for(String sRest : resRest) res.add(sRest + " " + rear);
        }
    }

    return res;
}

//以下 canBreak 函數是抄自本題 I 的 E3 代碼, 稍改了下返回值類型. 返回的 res 數組中, res[i]表示 s 前 i 個字符組成的子串 能否分成 wordDict 中的詞
public static boolean[] canBreak(String s, Set<String> wordDict) {
    int n = s.length();

    boolean[] res = new boolean[n + 1];
    res[0] = true;

    for(int r = 1; r <= n; r++) {
```

```
            for(int l = 0; l < r; l++) {
                res[r] = res[r] || (res[l] && wordDict.contains(s.substring(l, r)));
            }
        }

        return res;
    }
```

141. Linked List Cycle, Medium

http://blog.csdn.net/linhuanmars/article/details/21200601

Given a linked list, determine if it has a cycle in it.
Follow up:
Can you solve it without using extra space?
Subscribe to see which companies asked this question

Key: walker-runner 方法. runner 速度是 walker 兩倍. 若 walker 和 runner 能相遇, 則是 cycle, 否則不是.

History:
E1 一次通過.
E2 兩三分鍾寫好, 一次通過, 代碼跟 Code Ganker 的一模一樣.
E3 幾分鍾寫好, 一次通過, 方法跟 key 一樣.

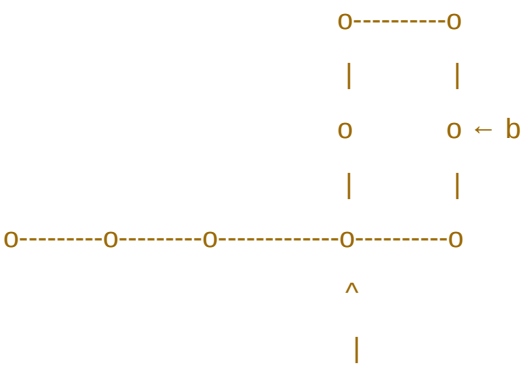
我: 我的代碼是用 HashMap, 若出現重復則返回 true, 寫好後一次通過, 後來看了 Code Ganker 說的, 又改成了 HashSet, 也能通過. 但這用了 extra space. Leetcode OJ 建議不用 extra space. 所以以下還是用 Code Ganker 的代碼. (William 的方法也跟 Code Ganker 一樣)

Code Ganker:

这道题是 cc150 里面的题目，算是链表里面比较经典的题目。
我们先讲一下比较直接容易想到的方法，就是用一个 hashset，然后把扫过的结点放入 hashset 中，如果出现重复则返回 true。想法比较简单，也很好实现，这里就不放代码了，有兴趣的朋友可以写写。
下面我们来考虑如何不用额外空间来判断是否有 cycle，用到的方法很典型，就是那种 walker-runner 的方法，基本想法就是维护两个指针，一个走的快，一个走得慢，当一个走到链表尾或者两个相见的时候，能得到某个想要的结点，比如相遇点，中点等等。

我:

以下一段是證明: 若為 cycle, 則 walker 和 runner 總是可以相遇的. 不用看, 知道這個結論就可以了 (還是要看看, 因為 142 題要用).



cycle 起始點

以上鏈表中, a = 3, c =6, 則 k = 1, b 按以下說的取 b = kc – a = 3. 則 m 取 0, n 取 k=1. 則 a + b + mc = 6, 即 s=6, 可驗證 a + b + nc = 12 = 2s. 由於總是取 m=0, 故以下代碼中兩個指針最先會相遇在上面算出的 b 處.

介绍完方法，剩下的主要就是数学了，假设两个指针 walker 和 runner，walker 一倍速移动，runner 两倍速移动。有一个链表，假设他在 cycle 开始前有 a 个结点，cycle 长度是 c，而我们相遇的点在 cycle 开始后 b 个结点。那么想要两个指针相遇，意味着要满足以下条件：(1) a+b+mc=s; (2) a+b+nc=2s; 其中 s 是指针走过的步数，m 和 n 是两个常数。这里面还有一个隐含的条件，就是 s 必须大于等于 a，否则还没走到 cycle 里面，两个指针不可能相遇。假设 k 是最小的整数使得 a<=kc，也就是说(3) a<=kc<=a+c (我: 這個式子表示: 第一, kc 是大於 a 的, 第二, (k-1)c 是小於 a 的, 即 kc<=a+c, 所以 k 是最小整數)。接下来我们取 m=0, n=k，用(2)-(1)可以得到 s=kc 以及 a+b=kc。由此我们可以知道，只要取 b=kc-a(我: 並用 m=0, n=k)(由(3)可以知道 b 不会超过 c)，那么(1)和(2)便可以同时满足，使得两个指针相遇在离 cycle 起始点 b 的结点上。
因为 s=kc<=a+c=n，其中 n 是链表的长度(我: 與前面的 n 不同)，所以走过的步数小于等于 n，时间复杂度是 O(n)。并且不需要额外空间，空间复杂度 O(1)。代码如下.

这道题是链表中比较有意思的题目，基于这个方法，我们不仅可以判断链表中有没有 cycle，还可以确定 cycle 的位置，有兴趣的朋友可以看看 Linked List Cycle II。

```
public boolean hasCycle(ListNode head) {
    if(head == null)
        return false;
    ListNode walker = head;
    ListNode runner = head;
    while(runner!=null && runner.next!=null) //注意其它 walker-runner 的題目中是 while(runner.next!=null && runner.next.next!=null)
    {
        walker = walker.next;
        runner = runner.next.next;
        if(walker == runner) //注意此 if 要寫在 while 裡面, 否則死循環!
            return true;
    }
    return false;
}
```

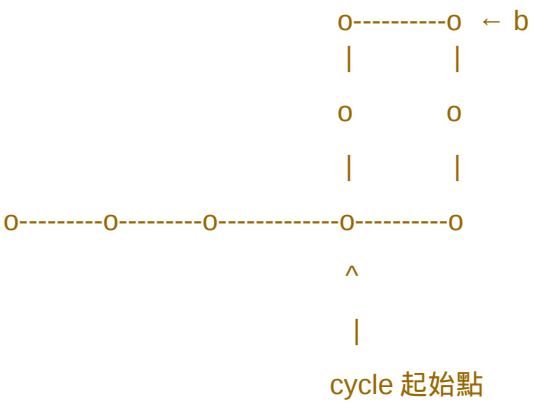
142. Linked List Cycle II, Medium

http://blog.csdn.net/linhuanmars/article/details/21260943

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.
Note: Do not modify the linked list.
Follow up:
Can you solve it without using extra space?
Subscribe to see which companies asked this question

Key: walker-runner 方法. 先將 walker 和 runner 都移到 b 點(即 walker 和 runner 的相遇點, 方法同 141 題), 然後 walker 從 list 起始點出發, runner 從 b 點出發, 兩個指針每次都走一步(就是移到下一個 node, runner 也這樣走), 當再相遇時, 即為 cycle 起始點.

我:



以上鏈表中, a = 3, c =6, 則 k = 1, b 按以下說的取 b = kc – a = 3, 可驗證第二個節點走過 kc-b (即 a) 步後會停在 cycle 起始點.

History:
E1 直接看的答案.
E2 幾分鍾寫好通過, 本來可以一次通過, 但第二次走時, walker 本應從 list 起點出發, 但我粗心寫成了從 b 點出發, 改後通過, 代碼基本上跟 Code Ganker 一模一樣.
E3 很快寫好, 但在 walker 和 runner 第一次相遇時, 忘了用 break 跳出循環, 導致了死循環, 改後即通過.

这道题是 Linked List Cycle 的扩展，就是在确定是否有 cycle 之后还要返回 cycle 的起始点的位置。从 Linked List Cycle 中用的方法我们可以得知 a=kc-b（不了解的朋友可以先看看 Linked List Cycle ）。现在假设有两个结点，一个从链表头出发，一个从 b 点出发，经过 a 步之后，第一个结点会到达 cycle 的出发点，而第二个结点会走过 kc-b，加上原来的 b 刚好也会停在 cycle 的起始点。如此我们就可以设立两个指针，以相同速度前进知道相遇，而相遇点就是 cycle 的起始点。算法的时间复杂度是 O(n+a)=O(2n)=O(n)，先走一次确定 cycle 的存在性并且走到 b 点，然后走 a 步找到 cycle 的起始点。空间复杂度仍是 O(1)。代码如下.

这道题目更多的是数学上相遇点的推导，有了理论基础之后就是比较简单的链表操作。

我: 為了保持與 141. Linked List Cycle 一致, 我對 Code Ganker 的代碼作了較大改進, 改後可通過. 方法仍是 Code Ganker 上面講的方法, 且代碼也不比 Code Ganker 的原代碼長, 而且看起來要清新些. 以下用我改後的代碼.

```
public static ListNode detectCycle(ListNode head) {
 if(head == null)
  return null;

 ListNode walker = head;
 ListNode runner = head;
//以下的 while 是將 walker 和 runner 指針移到 b 處
 while(runner != null && runner.next != null) {
  walker = walker.next;
  runner = runner.next.next;

  if(walker == runner)
   break;
 }
```

```
//以下是 list 無 cycle 之情況
 if(runner == null || runner.next == null)
  return null;

//walker 從 list 起始點出發, runner 從 b 點出發, 兩個指針每次都走一步, 當相遇時, 即為 cycle 起始點.
 walker = head;

 while(walker != runner) {
  walker = walker.next;
  runner = runner.next;
 }

 return runner;
}
```

143. Reorder List, Medium

Given a singly linked list L: $L0 \rightarrow L1 \rightarrow \ldots \rightarrow Ln\text{-}1 \rightarrow Ln$,
reorder it to: $L0 \rightarrow Ln \rightarrow L1 \rightarrow Ln\text{-}1 \rightarrow L2 \rightarrow Ln\text{-}2 \rightarrow \ldots$
You must do this in-place without altering the nodes' values.
For example,
Given {1,2,3,4}, reorder it to {1,4,2,3}.
Subscribe to see which companies asked this question

Key: 分三步完成：（1）将链表切成两半，也就是找到中点，然后截成两条链表；（2）将后面一条链表进行 reverse 操作，就是反转过来；（3）将两条链表按顺序依次 merge 起来。

History:
E1 在瞄了一眼 Code Ganker 的思路後, 自己很快寫出代碼並一次通過. 我的代碼基本上跟 Code Ganker 一樣, 但我的還是好理解一點, 以下用 E1 的代碼. 但 Code Ganker 說的也要看看:
E2 通過, 本來可以一次通過, 但粗心寫錯了兩個地方, 一是 runner = runner.next.next 寫成了 runner = runner.next, 二是第二條鏈表反轉後, 它的頭應該是 falseHead.next, 而我寫成了 head2.
E3 的方法跟 key 一樣, 但 E3 中某處的 cur = cur.next(E1 的代碼中無此句)之前沒用 if(cur != null), 改後即通過. 由於 E1 的代碼中無此句, 所以 E1 根本都不用擔心此句, 故 E3 的代碼還沒 E1 寫得好.

Code Ganker:

这是一道比较综合的链表操作的题目，要按照题目要求给链表重新连接成要求的结果。其实理清思路也比较简单，分三步完成：（1）将链表切成两半，也就是找到中点，然后截成两条链表；（2）将后面一条链表进行 reverse 操作，就是反转过来；（3）将两条链表按顺序依次 merge 起来。
这几个操作都是我们曾经接触过的操作了，第一步找中点就是用 Linked List Cycle 中的 walker-runner 方法，一个两倍速跑，一个一倍速跑，知道快的碰到链表尾部，慢的就正好停在中点了。第二步是比较常见的 reverse 操作，在 Reverse Nodes in k-Group 也有用到了，一般就是一个个的翻转过来即可。第三步是一个 merge 操作，做法类似于 Sort List 中的 merge，只是这里不需要比较元素打小了，只要按顺序左边取一个，右边取一个就可以了。
接下来看看时间复杂度，第一步扫描链表一遍，是 O(n)，第二步对半条链表做一次反转，也是 O(n)，第三部对两条半链表进行合并，也是一遍 O(n)。所以总的时间复杂度还是 O(n)，由于过程中没有用到额外空间，所以空间复杂度 O(1)。

```
public static void reorderList(ListNode head) {
  // 123 456 -> 162534
  // 1234 567 -> 1726354
 if(head == null || head.next == null)
  return;

 ListNode walker = head;
 ListNode runner = head;
//以下是找中點, 無論是 123 456 情況, 還是 1234 567 情況, while 執行完後, walker 都指向前一部分的最後一個(即第一種情況的 3 或第二種情況的 4)
 while(runner.next != null && runner.next.next != null) {
  walker = walker.next;
  runner = runner.next.next;
 }

 //把 list 一分為二
 ListNode head1 = head;
 ListNode head2 = walker.next;
 walker.next = null;

 head2 = reverseList(head2);
  // 123 654 -> 162534
  // 1234 765 -> 1726354

 ListNode pre = head1;
 ListNode cur = head2;

 while(cur != null) {
  ListNode next = cur.next;

  //以下兩句是將 cur 插到 pre 後面去
  cur.next = pre.next;
  pre.next = cur;
```

```
    cur = next;
    pre = pre.next.next;
  }
 }

 private static ListNode reverseList(ListNode head) {
  將 206. Reverse Linked List 中的 reverseList 函數抄過來. //E2 沒抄, 是自己寫的.
}
```

144. Binary Tree Preorder Traversal, Medium

http://blog.csdn.net/linhuanmars/article/details/21428647

Given a binary tree, return the *preorder* traversal of its nodes' values.
For example:
Given binary tree {1,#,2,3},

```
   1
    \
     2
    /
   3
```

return [1,2,3].
Note: Recursive solution is trivial, could you do it iteratively?
Subscribe to see which companies asked this question

Key: E3 的迭代法(iteratively): 同相要先將輸入樹的左邊緣的節點都放入 stack 中. 在 94. Binary Tree Inorder Traversal 的代碼上, 刪了一句 res.add(), 在其它 幾個地方加了幾句 res.add(), 其餘的跟 94 題相同.

但後來我又覺得 Code Ganker 的代碼好些.

Code Ganker 的迭代法(iteratively)：與 94. Binary Tree Inorder Traversal 維一的區別就是 res.add 的位置不一樣.

History:
E1 直接看的答案.
E2 很快寫好一次通過, 主要是因為記得 94. Binary Tree Inorder Traversal 的代碼.
E3 的遞歸和迭代法都是在我 94. Binary Tree Inorder Traversal 的 E3 代碼的基础上改的, 改好後一次通過. 同樣, 以後不用 Code Ganker 的迭代法代碼, 而用 E3 的迭代法代碼.

跟 Binary Tree Inorder Traversal 一样，二叉树的先序遍历我们仍然介绍三种方法，第一种是递归，第二种是迭代方法，第三种是用线索二叉树。
递归是最简单的方法，算法的时间复杂度是 O(n), 而空间复杂度则是递归栈的大小，即 O(logn)。代码如下：

```
public ArrayList<Integer> preorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    helper(root, res);
    return res;
}
private void helper(TreeNode root, ArrayList<Integer> res)
{
    if(root == null)
        return;
    res.add(root.val);
    helper(root.left,res);
    helper(root.right,res);
}
```

接下来是迭代的做法，其实就是用一个栈来模拟递归的过程。所以算法时间复杂度也是 O(n)，空间复杂度是栈的大小 O(logn)。实现的代码如下：

我：以下代碼用一個只有三個元素的樹（root-left-right）想一想，就明白了．

Tree traversal 迭代法以後統一用 LeetCode Discussion 上的代碼:

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode p = root;
    while(!stack.isEmpty() || p != null) {
        if(p != null) {
            stack.push(p);
            result.add(p.val);  // Add before going to children
            p = p.left;
        } else {
            TreeNode node = stack.pop();
            p = node.right;
        }
    }
    return result;
}
```

Code Ganker 的代碼(不用):

```
public ArrayList<Integer> preorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    if(root == null) //不要此 if, leetcode 中也能通過.
        return res;
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    while(root!=null || !stack.isEmpty())
    {
        if(root!=null)
        {
            stack.push(root);
            res.add(root.val);
```

```
            root = root.left;
        }
        else
        {
            root = stack.pop();
            root = root.right;
        }
    }
    return res;
}
```

最后使用 Morris 遍历方法(我: 已省略)

E3 的代碼(不用):

```
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null) return res;
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode l = root;
        stack.push(l);
        res.add(l.val);

        //將輸入樹的左邊緣的節點都放入 stack 中
        while(l.left != null) {
            l = l.left;
            res.add(l.val); //與 94 題(inorder)的代碼相比, 加了此句
            stack.push(l);
        }

        while(!stack.isEmpty()) {
            TreeNode cur = stack.pop();//與 94 題(inorder)的代碼相比, 刪掉了此句後的 res.add(cur.val)一句

            if(cur.right != null) {
                stack.push(cur.right);
                res.add(cur.right.val);//與 94 題(inorder)的代碼相比, 加了此句

                TreeNode curL = cur.right;
                while(curL.left != null) {
                    curL = curL.left;
                    res.add(curL.val);//與 94 題(inorder)的代碼相比, 加了此句
                    stack.push(curL);
                }
            }
        }

        return res;
    }
```

145. Binary Tree Postorder Traversal, Hard

http://blog.csdn.net/linhuanmars/article/details/22009351

Given a binary tree, return the *postorder* traversal of its nodes' values.
For example:
Given binary tree {1,#,2,3},

```
   1
    \
     2
    /
   3
```

return [3,2,1].
Note: Recursive solution is trivial, could you do it iteratively?
Subscribe to see which companies asked this question

Key: William 的方法: 本題的 postorder 的順序是 left-right-root. 故先做一個 pre-order, 其順序為 root-right-left(注意跟之前的 pre-order 不同, 此處 right 在 left 前), 最後將所得的結果 list 反轉過來, 即為本題之結果. 而做 root-right-left 的 pre-order 的方法是, 將之前 144. Binary Tree Preorder Traversal(注意該題是 root-left-right)的代碼中出現的所有 left 和 right 互換即可. 由於本解法中, 是將結果 list(即 res)作為了中間結果來用, 故空間複雜度由之前的 O(log n)變為了 O(n).

Code Ganker 的方法:
本題還是比較難的. 本黃字是 E2 寫的, 最後還有 E2 劃的圖(其實不看 E2 的圖, 只看本黃字, 反而好理解些, 因為沒那麼多細節), 可見 E2 真是漚心瀝血, E2 就是紅太陽! 本題基本思想還是和 preorder 和 inorder 一樣, 都是先沿樹的左邊沿走, 沿途的 node 加到 stack 中, 然後在 stack 中取 node 出來處理, 並加入到 res 中. 本題跟 preorder 和 inorder 相比, 複雜的地方就在於以下:
考慮一個只有三個 node 的樹 1(root)-2(left)-3(right), 沿樹的左邊沿走完後, stack 中是 1 和 2, 將 2 取出加入到 res 中後, stack 中剩下 1, 此為狀態 A, 然後根据 post order 之要求, 就要處理 1 的右結點 3.
處理 3 時, 其實就是重復上面的步驟, 即 「沿 以 3 為 root 的子樹 的左邊沿走, 沿途的 node 加到 stack 中, 然後在 stack 中取 node 出來處理, 並加入到 res 中」. 將 以 3 為 root 的子樹 處理完後, stack 中又只剩下 1, 此為狀態 B, 然後根据 post order 之要求, 就要處理 1 自己了.
注意上面狀態 A 和狀態 B 時, stack 中的東西都是一樣的, 但下一步要做的事不同, 所以只靠 stack 來指導下一步做什麼 是不夠的. 這就是本題複雜的地方.
解決的辦法就是, 在狀態 A 和狀態 B 時, 通過 判斷 1 的右結點是否已被訪問過(peekNode.right 是否等於 pre, 而 pre 就是上一輪的 peekNode) 來區分到底是狀態 A 還是狀態 B.
然後就可以看以下 Code Ganker 說的了:
1）如果当前栈顶元素的右结点存在并且还没访问过（也就是右结点不等于上一个访问结点），那么就把当前结点移到右结点继续循环；
2）如果栈顶元素右结点是空或者已经访问过，那么说明栈顶元素的左右子树都访问完毕，应该访问自己继续回溯了。

History:
E1 直接看的答案.
E2 沒做出來, 基本上也是直接看的答案.
E3 的遞歸法是在我 94. Binary Tree Inorder Traversal 的 E3 代碼的基础上改的, 改好後一次通過. E3 的迭代法沒做出來, 後來看了 William 的方法提示後, 由我自己寫出來的(William 的代碼跟我差不多), 一次通過, 但由於還是看了 William 的提示, 還是算我本題沒做出來. William 的方法空間複雜度可能沒 Code Ganker 的好, 但方法比 Code Ganker 的好懂太多了, 以後都用 William 的方法(當然是我 E3 的代碼, 在最前, 另 William 本題只給了這一種方法).

Tree traversal 迭代法以後統一用 LeetCode Discussion 上的代碼:

```
public List<Integer> postorderTraversal(TreeNode root) {
    LinkedList<Integer> result = new LinkedList<>();
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode p = root;
    while(!stack.isEmpty() || p != null) {
        if(p != null) {
            stack.push(p);
            result.addFirst(p.val);  // Reverse the process of preorder
            p = p.right;             // Reverse the process of preorder
        } else {
            TreeNode node = stack.pop();
            p = node.left;           // Reverse the process of preorder
        }
    }
    return result;
}
```

E3 的代碼(用了 William 的方法, 不用):

```
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null) return res;
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode l = root;
        stack.push(l);
        res.add(l.val);

        while(l.right != null) {
            l = l.right;
            res.add(l.val);
            stack.push(l);
        }

        while(!stack.isEmpty()) {
            TreeNode cur = stack.pop();

            if(cur.left != null) {
                stack.push(cur.left);
                res.add(cur.left.val);

                TreeNode curL = cur.left;
                while(curL.right != null) {
                    curL = curL.right;
                    res.add(curL.val);
                    stack.push(curL);
                }
            }
        }
```

```
        Collections.reverse(res);

        return res;
    }
```

Code Ganker:

跟 Binary Tree Inorder Traversal 以及 Binary Tree Preorder Traversal 一样，二叉树的后序遍历我们还是介绍三种方法，第一种是递归，第二种是迭代方法，第三种是用线索二叉树。 递归还是那么简单，算法的时间复杂度是 O(n), 而空间复杂度则是递归栈的大小，即 O(logn)。代码如下：

```
public ArrayList<Integer> postorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    helper(root, res);
    return res;
}
private void helper(TreeNode root, ArrayList<Integer> res)
{
    if(root == null)
        return;
    helper(root.left,res);
    helper(root.right,res);
    res.add(root.val);
}
```

接下来是迭代的做法，本质就是用一个栈来模拟递归的过程，但是相比于 Binary Tree Inorder Traversal 和 Binary Tree Preorder Traversal，后序遍历的情况就复杂多了。我们需要维护当前遍历的 cur 指针和前一个遍历的 pre 指针来追溯当前的情况（注意这里是遍历的指针，并不是真正按后序访问顺序的结点）。具体分为几种情况：
（1）如果 pre 的左孩子或者右孩子是 cur，那么说明遍历在往下走，按访问顺序继续，即如果有左孩子，则是左孩子进栈，否则如果有右孩子，则是右孩子进栈，如果左右孩子都没有，则说明该结点是叶子，可以直接访问并把结点出栈了。
（2）如果反过来，cur 的左孩子是 pre，则说明已经在回溯往上走了，但是我们知道后序遍历要左右孩子走完才可以访问自己，所以这里如果有右孩子还需要把右孩子进栈，否则说明已经到自己了，可以访问并且出栈了。
（3）如果 cur 的右孩子是 pre，那么说明左右孩子都访问结束了，可以轮到自己了，访问并且出栈即可。
算法时间复杂度也是 O(n)，空间复杂度是栈的大小 O(logn)。实现的代码如下(己省略).

上面迭代实现的思路虽然清晰，但是实现起来还是分情况太多，不容易记忆。我后来再看 wiki 的时候发现有跟 Binary Tree Inorder Traversal 和 Binary Tree Preorder Traversal 非常类似的解法，容易统一进行记忆，思路可以参考其他两种，区别是最下面在弹栈的时候需要分情况一下：
1）如果当前栈顶元素的右结点存在并且还没访问过（也就是右结点不等于上一个访问结点），那么就把当前结点移到右结点继续循环；
2）如果栈顶元素右结点是空或者已经访问过，那么说明栈顶元素的左右子树都访问完毕，应该访问自己继续回溯了。
下面列举一下代码：

我: 以下代碼要在紙上劃才想得清楚, 就以只有三個元素的樹（root-left-right）為例想一想 (最後有我 E2 劃的圖)

以下代碼不用:

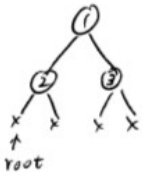
```
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    if(root == null)
    {
        return res;
    }
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    TreeNode pre = null;
    while(root != null || !stack.isEmpty())
    {
        if(root!=null)
        {
            stack.push(root);
            root = root.left;
        }
        else
        {
            TreeNode peekNode = stack.peek();
            if(peekNode.right != null && peekNode.right != pre) //E2 做了 minor change, 改後更清楚, 可通過
            {
                root = peekNode.right; //注意若执行了此 if, 就不會再执行下面的 else 了, E1 好幾次都看錯了
            }
            else
            {
                stack.pop();
                res.add(peekNode.val);
                pre = peekNode;
            }
        }
    }
    return res;
}
```

評論:

您好！wiki 上的迭代方法那一块，TreeNode peekNode = stack.peek(); peekNode 的定义移到 while 循环外面会不会好一点呢？
回复 AllenKing13：理论上是要好一些哈~ 不用反复的申请变量~
(我: 應該不可以放到 while 外面, 我在 leetcode 中試了一下, 通不過)

以下是圖 1/2:

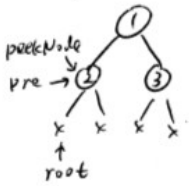
先從root出發，沿著樹的左邊沿走，沿途的node加入到stack中，然後得到下圖：



res：空

左圖在一次while循環後，變為下圖：

此句話簡寫為"urangps"。



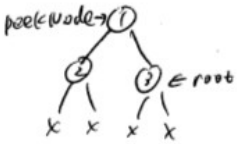
res：2

(即 pre ≠ 1.right)

此時棧頂元素1的左結點存在但還沒被訪問過，

則下一步要將root移到1左結點繼續循環。

urangps.



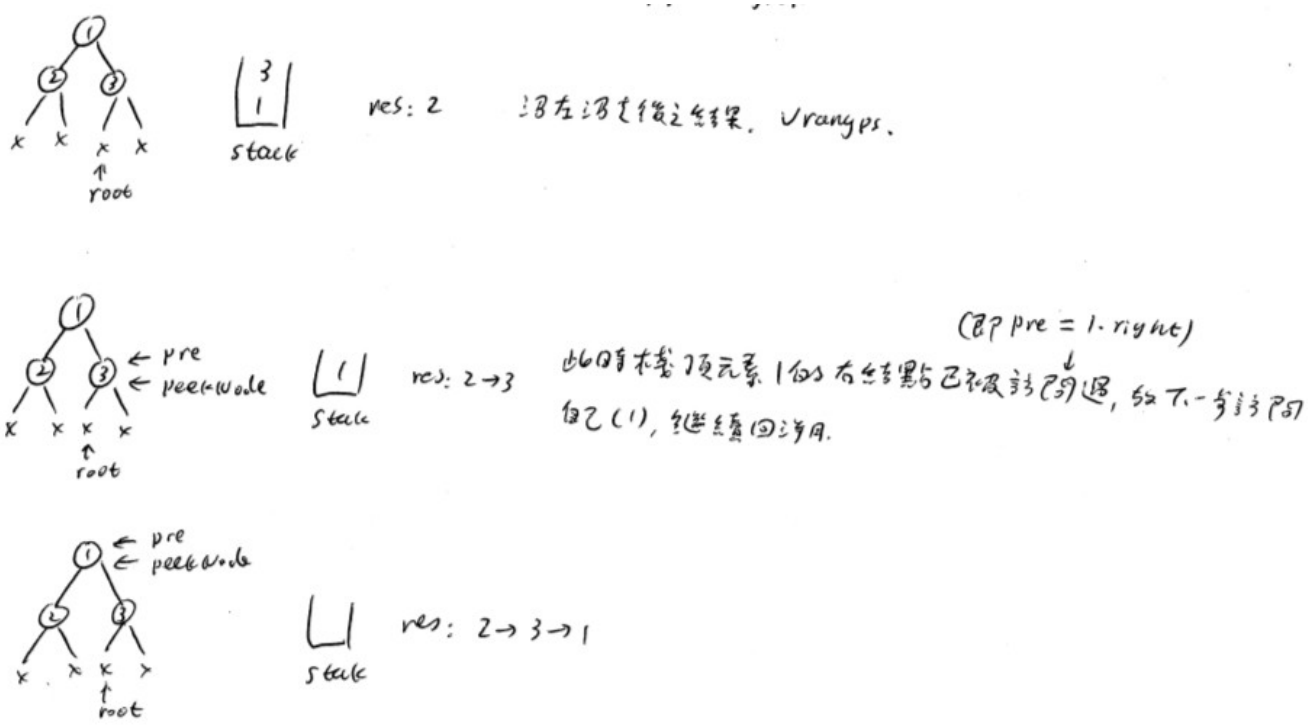
res：2

此即將new root，~~urangps~~，然後以3為新的root，重新開始，

即重復上面的步驟(即先沿以3為root的子樹的左沿走，並加到stack中，

然後…)，~~urangps~~。

以下是圖 2/2:



最后介绍 Morris 遍历方法，这个方法不需要为每个节点额外分配指针指向其前驱和后继结点，而是利用叶子节点中的右空指针指向中序遍历下的后继节点就可以了，所以优势在于不需要额外空间。不过同样相比于 Binary Tree Inorder Traversal 和 Binary Tree Preorder Traversal，后序遍历的情况要复杂得多，因为后序遍历中一直要等到孩子结点访问完才能访问自己，需要一些技巧来维护。

在这里，我们需要创建一个临时的根节点 dummy，把它的左孩子设为树的根 root。同时还需要一个 subroutine 来倒序输出一条右孩子路径上的结点。跟迭代法一样我们需要维护 cur 指针和 pre 指针来追溯访问的结点。具体步骤是重复以下两步直到结点为空：
1. 如果 cur 指针的左孩子为空，那么 cur 设为其右孩子。
2. 否则，在 cur 的左子树中找到中序遍历下的前驱结点 pre（其实就是左子树的最右结点）。接下来分两种子情况：
（1）如果 pre 没有右孩子，那么将他的右孩子接到 cur。将 cur 更新为它的左孩子。
（2）如果 pre 的右孩子已经接到 cur 上了，说明这已经是回溯访问了，可以处理访问右孩子了，倒序输出 cur 左孩子到 pre 这条路径上的所有结点，并把 pre 的右孩子重新设为空（结点已经访问过了，还原现场）。最后将 cur 更新为 cur 的右孩子。
空间复杂度同样是 O(1)，而时间复杂度也还是 O(n)，倒序输出的过程只是加大了常数系数，并没有影响到时间的量级。如果对这个复杂度结果不是很熟悉的朋友，可以先看看 Binary Tree Inorder Traversal 中的分析，在那个帖子中讲得比较详细。实现的代码如下：

(以下 Morris 代碼不用, 我有專門的 Morris 代碼文件)

```
public ArrayList<Integer> postorderTraversal(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    TreeNode dummy = new TreeNode(0);
    dummy.left = root;
    TreeNode cur = dummy;
    TreeNode pre = null;
```

```
    while(cur!=null)
    {
        if (cur.left==null)
        {
            cur = cur.right;
        }
        else
        {
            pre = cur.left;
            while (pre.right!=null && pre.right!=cur)
                pre = pre.right;
            if (pre.right==null)
            {
                pre.right = cur;
                cur = cur.left;
            }
            else
            {
                reverse(cur.left, pre);
                TreeNode temp = pre;
                while (temp != cur.left)
                {
                    res.add(temp.val);
                    temp = temp.right;
                }
                res.add(temp.val);
                reverse(pre, cur.left);
                pre.right = null;
                cur = cur.right;
            }
        }
    }
    return res;
}
void reverse(TreeNode start, TreeNode end)
{
    if (start == end)
        return;
    TreeNode pre = start;
    TreeNode cur = start.right;
    TreeNode next;
    while (pre != end)
    {
        next = cur.right;
        cur.right = pre;
        pre = cur;
        cur = next;
    }
}
```

到目前为止，二叉树的三种遍历的三种方法都介绍过了，后序遍历相比于前面两种，还是要复杂一些，个人觉得面试中可能倾向于靠其他两种遍历，特别是像 Morris 遍历方法，如果没有准备过很难在面试中写出这种方法的后序遍历，主要还是要有概念，就是知道方法的存在性以及优劣的分析就可以了，不过递归法和迭代法还是需要掌握一下的。

146. LRU Cache, Hard

http://blog.csdn.net/linhuanmars/article/details/21310633

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.
Subscribe to see which companies asked this question

Key: 見 Code Ganker 的 "用一个 hash 表加上一个双向链表(tao: 自己定義一個 Node class)来实现。基本思路是这样的" 直到那一段結束. 注意 Code Ganker 的 convention 是刚刚用到的应该放在队尾，然后当需要踢掉元素的时候从队头踢出 (which Tao thinks is consistent with Queue operations).

我: 由此可以深刻理解 cache: cache 就是用來暫存用得頻繁的東西. 用得最頻繁的放在隧尾, 好方便下次用的時候訪問. 用得最不頻繁的放在隧首, 若隧尾有新東西放入了, 則隧首的被踢掉. 注意 cache 只是在內存中, 所以踢出的那個並不是永久刪除了, 而是依然在硬盤中, 只是被踢出了 cache 中.

Corner case: 注意調用 set(key)或 get(key, value)時, 當 key 存在時, 都要把 key 對應的 node 移到隧尾.

History: E1 直接看的答案. E2 通過, 花了點時間, 但 E2 的代碼實際上比 Code Ganker 的代碼稍簡潔, 因為我引入了首尾兩個節點, 並事先將它們連好了.

我: 本題也是我目前見過的代碼最長的. 代碼雖然長, 但看了我的注釋後其實也不難理解. 注意本題在 Leetcode 的 Top 10 Poplular 中排首位. 後來發現 Top 10 Poplular 經常在變, 有時此題都进不了 Top 10 Poplular. 所以我覺得此 Top 10 可能不是網友報的面試 Top 10 Poplular, 而是大家刷題的 Top 10 Poplular .

Code Ganker:

这是一道非常综合的题目，主要应用在操作系统的资源管理中。
按照题目要求，要实现 get 和 set 功能，为了满足随机存储的需求我们首先想到的一般是用数组，如果用链表会有 O(n)的访问时间。然而他又有另一个条件就是要维护 least used 的队列，也就是说经常用的放在前面，用的少的放在后面。这样当资源超过 cache 的容积的时候就可以把用得最少的资源删掉。这就要求我们要对节点有好的删除和插入操作，这个要求又会让我们想到用链表，因为数组的删除和插入是 O(n)复杂度的。
那么我们能不能维护一个数据结构使得访问操作和插入删除操作都是 O(1)复杂度的呢？答案是肯定的。这个数据结构比较复杂，是用一个 hash 表加上一个双向链表(tao: 自己定義一個 Node class)来实现。基本思路是这样的，用一个 hash 表来维护结点的位置关系，也就是 hash 表的 key 就是资源本身的 key，value 是资源的结点(我: 當然是對節點的引用)（包含 key 和 value 的信息）。然后把结点维护成一个双向链表构成的队列，这样子如果我们要访问某一个结点，那么可以通过 hash 表和 key 来找到结点，从而取到相应的 value。而当我们想要删除或者插入结点时，我们还是通过 hash 表找到结点，然后通过双向链表和队列的尾结点把自己删除同时插入到队尾。通过 hash 表访问结点我们可以认为是 O(1)的操作（假设 hash 函数足够好），然后双向链表的插入删除操作也是 O(1)的操作(我: 不用去遍歷鏈表, 因為已經通過 hash 知道要插入或刪除哪個節點了)。如此我们便实现了用 O(1)时间来完成所有 LRU cache 的操作。空间上就是对于每一个资源有一个 hash

表的的项以及一个对应的结点（包含前后指针和资源的<key, value>）。代码如下.

实现的时候还是有很多细节的，因为我们不经常使用双向链表，插入删除操作要维护前后指针，并且同时要维护成队列，增加了许多注意点。这道题是一道很实际的题目，思路和数据结构都是很适合面试的题目，但是代码量有些偏大，所以一般只在 onsite 的时候有可能遇到，可能也不会让你完整地写出全部代码，主要还是讲出维护的数据结构和各种操作复杂度的分析。

在面试时还被问道，如果要实现多线程 LRU，应该在什么地方加锁？然后如何优化？我就很模糊的说只在 set 时给 hashmap 与 linkedlist 加锁。如何优化就更不知道应该从哪个方向回答了。。望指点！谢谢了

: Code_Ganker 2015-07-23 08:44 发表 回复 mikeTong830：这个应该还可以从读写锁的概念来，get 是读锁，set 是写锁，写锁优先级更高哈~ 可以多个读一个，不可以写一个~

请教一下如果 LRUCache 中只有一个元素的时候，假设这个元素是 node， 是不是 first， last 都等于 node？ 我看 lz 代码好像是这个意思。

回复议题一玩到：是的，一个元素就是 first 和 last 都是它~

回复 Code_Ganker：谢谢大牛回复， 其实 set 方法中 if(first!=null) first.pre = null; else last = null; 这里不对 first， last 设成 null 也没有什么影响的， 代码测试过是没问题的， 之前 get 方法中对新的 first 元素的 pre 指针都没有设成 null， 当然大牛这样写更加严谨。

求问 31-42 行那里，为什么 get 到 node 的时候要把 node 加入到队尾呢，不是访问后加入队头表示最晚访问到，优先级最高么？难道我理解错了？求指点。

回复 qweyxy_2：这个队列其实是在确定下一个踢掉的是哪个哈 ~ 也就是 least used 是哪个，所以刚刚用到的应该放在队尾，然后当需要踢掉元素的时候从队头踢出 ~

回复 Code_Ganker：哦哦，感觉理解不一样，我写的是队头是最新的 node， 队尾是 least 的，也通过了。感谢，哈哈！

除了 hash，还有其他高级数据结构能满足这个要求吗？求指点。

回复 iuvsamuel：就我所知，好像没有数据结构能实现所有操作都是 O(1)的了哈 ~

```
public class LRUCache {
    class Node //class 中的 class
    {
        Node pre, next;
        int key;
        int val;
        public Node(int key, int value)
        {
            this.key = key;
            this.val = value;
```

```
    }
}

private int capacity; //别忘了 private
private int num; //num 意思見下
private HashMap<Integer, Node> map; //不用將其賦值為 new HashMap<>(), 下面 constructor 中會賦值. 我甚至懷疑 Java 中的 class 能不能在 constructor 之外給 memeber 賦值. E2 實踐表明 在 constructor 之外, 可以這樣: int a = 0; 但不能這樣: int a; a=0; (已 record 到 Java p348). E2 我仍寫了 Map<Integer, Node> map = new HashMap<>() 和 int numberOfElements = 0 的, 都 OK.
private Node first, last;

public LRUCache(int capacity) { //這是 constructor
    //以下是初始化 LRUCache 這個 class 中的數据成員: capacity, num, map, first, last.
    this.capacity = capacity;
    num = 0; //num 即為 cache 中節點的個數. 此題中的 cache 就是指我們維護的那個隧列(當然還有個起輔助作用的 map). 最開始 cache 是空的, 當調用了 set 函數後, cache 裡面才開始有元素, 所以 num 是在 set 函數裡被加 1 的
    map = new HashMap<Integer, Node>();
    first = null;
    last = null;
}

//get(key)函數的作用是返回 key 所對應的節點的 value, 並且將此節點移到鏈表末尾
public int get(int key) {
    Node node = map.get(key);
    if(node == null)
        return -1;
    if(node!=last) //原代碼是 else if(node!=last), 這裡我把 else 去掉了, 好與下面 set 函數一致. 改後能通過
    {   //以下是刪除 node. 注意對於雙向鏈表, 刪除一個節點(不是鏈表首尾節點)需要重搭兩條線, 即 node 前的那個節點的 next, 和 node 後那個節點的 pre. 而加入一個節點(不是鏈表首尾節點)需要重搭四條線, 即 node 前的那個節點的 next, node 的 pre, node 的 next,  和 node 後那個節點的 pre
        if(node == first)
            first = first.next;
        else
            node.pre.next = node.next; //注意 else 只能管到這句
```

```
            node.next.pre = node.pre;
            //以下是將 node 接到鏈表末尾
            last.next = node;
            node.pre = last;
            node.next = null;
            //以下是更新 last 指針
            last = node;
        }
        return node.val;
    }

    public void set(int key, int value) {
        Node node = map.get(key); //注意這是 map 的 get 函數, 不是前面自己寫的 get 函數. set 函數並不調用前面自己寫的 get 函數(所以後面會重復 get 函數的代碼)
        if(node != null) //當 node 在 map 中存在的時候, 刪除 node, 並將 node 接到鏈表末尾
        {
            node.val = value;
            if(node!=last) //此 if 中的內容跟 get 函數中 else if(node!=last)中的內容是一模一樣的, 目的就是刪除 node, 並將 node 接到鏈表末尾, 最後更新 last 指針
            {
                if(node == first)
                    first = first.next;
                else
                    node.pre.next = node.next;
                node.next.pre = node.pre;
                last.next = node;
                node.pre = last;
                node.next = null;
                last = node;
            }
        }
        else //當 node 在 map 中不存在的時候, 新建 node, 並將 node 接到鏈表末尾
        {
            Node newNode = new Node(key,value);
```

```
        //以下的 if 是當隧列滿了時, 踢掉隧首的節點, 只管刪節點, 不管加節點
        if(num>=capacity)
        {
            map.remove(first.key);  //隧尾有新東西放入了, 故隧首的被踢掉. 別忘了 map 和隧列中都要刪
            first = first.next;
            if(first!=null)
                first.pre = null;
            else //小細節, 即 first 為 null 時, 實際上此時隧列為空
                last = null; //注意 else 只管此句
            num--; //上面已經刪了一個元素了, 故 num--
        }

        //以下是將新節點加入隧尾, 只管加節點, 不管刪節點.
        if(first == null || last == null)  //只要 first 和 last 中有一個為 null, 整個鏈表就為空, 此時就把 first 和 last 都 設為 newNode. first 在下面馬上設, last 在 if 和 else 之後設
        {
            first = newNode;
        }
        else
        {
            last.next = newNode;
        }
        newNode.pre = last;
        last = newNode;
        map.put(key,newNode); //同樣別忘了 map 和隧列中都要加
        num++; //上面已經加了一個元素了, 故 num++
    }

  }
}
```

147. Insertion Sort List, Medium

http://blog.csdn.net/linhuanmars/article/details/21144553

Sort a linked list using insertion sort.
Subscribe to see which companies asked this question

Key: 我最開始以為本題一定有一次遍歷的方法, 但自己又想不出來, 就沒自己寫. 結果看了 Code Ganker 的解法後發現也不是一次遍歷(E3 曰: insertion sort 當然不能一次遍歷). 以下代碼中, cur 是當前節點, 是要插入前面去的. 這裡用了一個 helper 節點(即 fasehead), 注意從一開始, helper 就沒和 head 相連, 實際上是為結果建了一個新的 list, 它以 helper 為開頭. pre 每次從新 list 的開頭開始掃描(所以 cur 和 pre 都做了遍歷, 沒事, insertion sort 的時間復雜度本來就是 O(n^2)), 直到 pre 的下一個數比 cur 大, 然後把 cur 插到 pre 後面. 重建一個新 list 的好處就是新 list 的末尾自動就是 null, 不會受老 list 中每個 node 的 next 的影响.

History:
E1 直接看的答案.
E2 很快寫好, 一次通過.
E3 一次通過. E3 沒有重建一個新的 list, 而是強行在原 list 中移動 node 的, 是最正綜的 in-place. 而 Code Ganker 的代碼雖然重建了一個新的 list, 但並沒有 create 新的 node, 而是將原 node 搬到新的 list 中去, 所以也沒多用出空間, 而且 Code Ganker 的方法要好寫些, 所以以後還是以 Code Ganker 的方法為准.

这道题跟 Sort List 类似，要求在链表上实现一种排序算法，这道题是指定实现插入排序。插入排序是一种O(n^2)复杂度的算法，基本想法相信大家都比较了解，就是每次循环找到一个元素在当前排好的结果中相对应的位置，然后插进去，经过 n 次迭代之后就得到排好序的结果了。了解了思路之后就是链表的基本操作了，搜索并进行相应的插入。时间复杂度是排序算法的 O(n^2)，空间复杂度是 O(1)。代码如下.
这道题其实主要考察链表的基本操作，用到的小技巧也就是在 Swap Nodes in Pairs 中提到的用一个辅助指针来做表头避免处理改变 head 的时候的边界情况。作为基础大家还是得练习一下哈。

这题还有个地方，helper 先不和 head 相连，也是个边界

```
public ListNode insertionSortList(ListNode head) {
    if(head == null)
        return null;
    ListNode helper = new ListNode(0);
    ListNode pre = helper;
    ListNode cur = head;
    while(cur!=null)
    {
        ListNode next = cur.next;
        pre = helper; //pre 每次從新 list 的開頭開始掃描
        while(pre.next!=null && pre.next.val<=cur.val) //用<也可以通過
        { //E2 總共只用了一個 while, 但複雜度是一樣的 (這裡只是想說明只寫一個 while 也是可能的). E3 也只用了一個 while.
            pre = pre.next;
        }
        //以下兩句是把 cur 插到 pre 後面
        cur.next = pre.next;
        pre.next = cur;

        cur = next;
    }
    return helper.next;
}
```

148. Sort List, Medium

http://blog.csdn.net/linhuanmars/article/details/21133949

Sort a linked list in $O(n \log n)$ time using constant space complexity.
Subscribe to see which companies asked this question

Key: 用 merge sort. 用 walker 和 runner 來找 list 中點, 其中 runner 跑的速度是是 walker 的兩倍. 中點不一定要找得很精確. 找到中點後, 把 list 分為兩點, 分別遞歸調用本 sortList 函數. 最後 merge 兩個 list, merge 函數調用 21. Merge Two Sorted Lists 的代碼.

Corner case: 只有兩個元素的 list. 見 History 中的 E3.

(Tao: William 也是用的 merge sort)

History:
E1 直接看的答案.
E2 本來可以一次通過, 但有個細節寫錯了(見代碼中黃字), 改了兩次才通過. 另外 E2 在 sortList 函數了還單獨處理了 list 只有兩個元素之情況, 其實沒必要, mergeTwoLists 函數其實會中處理這個情況.
E3 也用的 merge sort (merge 函數是自己寫的, 沒調用之前的), 最初寫好後, 還是對 '只有兩個元素的 list' 通不過, 專門處理了這種情況後即通過(E2 也專門處理了). 這是因為我最初將移動 walker 和 runner 的循環條件寫成了 while(runner != null && runner.next != null), 這使得對 '只有兩個元素的 list' 的情況, runner = null, walker 等於第二個元素, 故分裂後的第二條鏈的 head(=walker.next)為 null, 分裂後的第一條鏈還是原來的那條老的 '只有兩個元素的 list', 這其實並沒起到分裂的作用, 最後對 分出來的兩條鏈 遞歸調用 sortList 函數時, sortList 函數處理第一條鏈 實際上又回到了原問題(sort 一個 '只有兩個元素的 list'), 因而死循環. 後來發現 Code Ganker 的代碼寫成的是 while(runner.next!=null && runner.next.next!=null), 這樣可以將 '只有兩個元素的 list' 均分成兩等份, 故不用專門考慮 '只有兩個元素的 list' 的情況.

这道题跟 Insertion Sort List 类似，要求我们用 O(nlogn)算法对链表进行排序，但是并没有要求用哪一种排序算法，我们可以使用归并排序，快速排序，堆排序等满足要求的方法来实现。对于这道题比较容易想到的是归并排序，因为我们已经做过 Merge Two Sorted Lists，这是归并排序的一个 subroutine。剩下我们需要做的就是每次找到中点，然后对于左右进行递归，最后用 Merge Two Sorted Lists 把他们合并起来。代码如下.
不过用归并排序有个问题就是这里如果把栈空间算上的话还是需要 O(logn)的空间的。对于其他排序算法，用兴趣的同学可以实现一下哈。
排序是面试中比较基础的一个主题，所以对于各种常见的排序算法大家还是要熟悉，不了解的朋友可以参见排序算法 - Wiki。特别是算法的原理，很多题目虽然没有直接考察排序的实现，但是用到了其中的思想，比如非常经典的 topK 问题，就用到了快速排序的原理，关于这个问题在 Median of Two Sorted Arrays 中有提到，有兴趣的朋友可以看看。

这道题使用 merge sort 可以做到 O(1) 的空间复杂度(我: 我仔細分析了 Code Ganker 的代碼, 空間復雜度也是 O(1). 我在群裡跟 DS^N 等人討論過, 一致認為(DS^N 有 reference 可作證)merge sort 的空間復雜度取決於數据結構, 若為 list, 就是 O(1), 若為數組, 就是 O(n) (E3: 當然不算遞歸棧的空間)。思路是把输入的链表当作已排序的长度为 1 的若干个链表的合，然后从头至尾做 log(n) 次 pass，每次 pass 把相邻的两个链表 merge。由于链表的特点，可以获得非常好的时间、空间复杂度。
详细介绍可以看：
http://t.cn/RAvuKoI
当然实现上会稍复杂一点

每次都需要找中点,时间复杂度还是 O(nlogn), 这点想不明白, 博主有空的时候能不能解释一下呢? 谢谢啦！
每道题做好都要看看博主怎么解的，受益匪浅！

回复 github_21795029：因为这个找中点过程和 merge 是并列的，也就是复杂度严格来说是 O((n+n)*logn)=O(2nlogn)，所以还是 O(nlogn)哈 ~
回复 github_21795029：另外如果像数组 mergesort 那样用 bottom up 迭代的办法可不可以做呢,
但是那样的话,每次要遍历到一个组的开头,好像比 bottom down 递归时间复杂度更高
回复 github_21795029：按理来说也是可以的 ~ 时间复杂度有没有增加取决于这个线性操作是否与 merge 并列，如果是并列，虽然常数变大，但是复杂度的量级还是一样的 ~

这道题目还有一个比较好的实现方式，STL 里面对 list 进行 sort 的时候采用的方法。

http://blog.csdn.net/flymu0808/article/details/38128477 这个最后面有一个介绍。 感觉你的 blog 对我用处很大，希望这个对你有所帮助，算是礼尚往来，哈哈。

回复 Superbluesummer：好的，多谢，我看看哈 ~

constant space complexity 如何理解？ 如何保证？

回复 ttc9082：按理来说，space complexity 应该要计算上栈空间的大小的，在这里正如我最后讨论的，需要 O(logn)的栈空间。如果用其他 O(nlogn)的排序算法，快排也是 O(logn)(同样是栈空间)，而堆排序则需要 O(n)。所以我没有想到能够真正做到 constant 空间的算法，如果您有什么好算法，欢迎继续讨论哈 ~

我: Code Ganker 的原代碼有點囉嗦，寫了三個函數，我將其改簡明了點，改後可通過．

```
public ListNode sortList(ListNode head)
{
    if(head == null || head.next == null)
        return head;
    //以下是讓 walker 指向 list 的中間, runner 用來是輔助 walker 的, 因為 runner 跑的速度是 walker 的兩倍, 所以若 runner 到末尾了, 就說明 walker 就到中間了. walker 也不是一定要精確地在中間位置, 大差不差就可以了.
    ListNode walker = head;
    ListNode runner = head;
    while(runner.next!=null && runner.next.next!=null) //判斷 runner.next != null 還是有必要的, 否則若 runner.next 為 null 時, 後面判斷 runner.next.next 時就會報錯, 因為 runner.next 自己就已經是 null, 哪來的 next. //E2 此句最開始寫成了 while(walker != null && runner.next != null), 改了兩次才通過.
    {
        walker = walker.next;
        runner = runner.next.next;
    }
    ListNode head2 = walker.next; //也可以該 head2 = walker, 我上面說了的, 不一定要精確地在中間位置. 這裡讓 head2 = walker.next 只是為了下面寫代碼方便.
    walker.next = null;
    ListNode head1 = head;
    head1 = sortList(head1);
    head2 = sortList(head2);
    return mergeTwoLists(head1, head2);
}

private ListNode mergeTwoLists(ListNode l1, ListNode l2)
{
```

```
    將 21. Merge Two Sorted Lists 的代碼抄過來
}
```

149. Max Points on a Line, Hard

http://blog.csdn.net/linhuanmars/article/details/21060933

Given *n* points on a 2D plane, find the maximum number of points that lie on the same straight line.
Subscribe to see which companies asked this question

Key: 方法很直接(Code Ganker 和 William 都這麼做的). 每次選一個點(稱為小明)為基准, 掃描其它點, 用一個 Map(即小明的 Map), key 為其它點跟小明的斜率, value 為 跟小明有此斜率 的點的個數. 其它點掃完後, 找出 小明的 Map 中 value 的最大值. 然後換下一點(稱為小紅)為基准, 也找出 小紅的 Map 中 value 的最大值,... 最後在這些最大值中取最大的即可.

注意 Double 也可以作 Map 的 key(如 LC-149), Java 好像有判斷 Double 相等的方法. 已 record 至 Java p834.

Corner case: 斜率為無限大時; 點重合時; 算斜率時, (double)(2 - 3) / (double)(8 - 4) = -0.0, 跟 0.0 被當成不同的數.

History:
E1 直接看的答案.
E2 被 corner case 擋了好幾次, 最後通過.
E3 較快寫好, 稍改了兩三次即通過. E3 的 Map 的 value 為 斜率相同的 Point 組成 的 List, 這樣做沒有答案中好, 答案是只放的 Point 數量.

我: Code Ganker 的第一種方法的代碼(很好懂)看看就可以, 要記住的是他的第二種方法. William 也是用的跟他第二種差不多的方法. 別看以下兩種方法的代碼都有點長, 其實很好看懂.

Code Ganker:
这道题属于计算几何的题目，要求给定一个点集合，是求出最多点通过一条直线的数量。我的思路是 n 个点总共构成 n*(n-1)/2 条直线，然后对每条直线看看是有多少点在直线上，记录下最大的那个，最后返回结果。而判断一个点 p_k 在 p_i, p_j 构成的直线上的条件是(p_k.y-p_i.y)*(p_j.x-p_i.x)==(p_j.y-p_i.y)*(p_k.x-p_i.x)。算法总共是三层循环，时间复杂度是 O(n^3),空间复杂度则是 O(1),因为不需要额外空间做存储。代码如下.

Code Ganker 第一種方法(不用):

```
public int maxPoints(Point[] points) {
    if(points==null || points.length==0)
    {
        return 0;
    }
    if(allSamePoints(points))//所有點都在同一條直線上時
        return points.length;
    int max = 0;
    //以下是判斷第 k 個點在不在 「以第 i 個點 和 以第 j 個點 構成的直線」上
    for(int i=0;i<points.length-1;i++)
    {
        for(int j=i+1;j<points.length;j++)
        {
            if(points[j].y==points[i].y && points[j].x==points[i].x)
                continue;
```

```
            int cur = 2;
            for(int k=0;k<points.length;k++)
            {
                if(k!=i && k!=j
                && (points[k].y-points[i].y)*(points[j].x-points[i].x)
                    ==(points[j].y-points[i].y)*(points[k].x-points[i].x))
                    cur++;
            }
            max = Math.max(max,cur);
        }
    }
    return max;
}
private boolean allSamePoints(Point[] points)
{
    for(int i=1;i<points.length;i++)
    {
        if(points[0].y!=points[i].y || points[0].x!=points[i].x)
            return false;
    }
    return true;
}
```

大家看到代码中还写了一个 allSamePoints 的函数，这是一个非常 corner 的情况，就是所有点都是一个点的情况，因为下面我们要跳过重复的点（否则两个重合点会认为任何点都在他们构成的直线上），但是偏偏当所有点都重合时，我们需要返回所有点。除了这种情况，只要有一个点不重合，我们就会从那个点得到结果，这是属于比较 tricky 的情况。

经过一个朋友的提醒， 这道题还可以有另一种做法， 基本思路是这样的， 每次迭代以某一个点为基准， 看后面每一个点跟它构成的直线， 维护一个 HashMap， key 是跟这个点构成直线的斜率的值， 而 value 就是该斜率对应的点的数量， 计算它的斜率， 如果已经存在， 那么就多添加一个点， 否则创建新的 key。 这里只需要考虑斜率而不用考虑截距是因为所有点都是对应于一个参考点构成的直线， 只要斜率相同就必然在同一直线上。 最后取 map 中最大的值， 就是通过这个点的所有直线上最多的点的数量。 对于每一个点都做一次这种计算， 并且后面的点不需要看扫描过的点的情况了， 因为如果这条直线是包含最多点的直线并且包含前面的点， 那么前面的点肯定统计过这条直线了。 因此算法总共需要两层循环， 外层进行点的迭代， 内层扫描剩下的点进行统计， 时间复杂度是 O（n^2), 空间复杂度是哈希表的大小， 也就是 O（n), 比起上一种做法用这里用哈希表空间省去了一个量级的时间复杂度。 代码如下：

用 double 做斜率比较直线是否是同一条，是不是不很精确？ 我看网上有人用一对整数表示斜线。。。

回复 linkedin_21530339：在 java 里面是可以的，它考虑了数值问题哈~ C++就不是很好，得用数值方法判断相等~

回复 linhuanmars：能具体说说两者的区别吗，为何 c++不行呢，两者的 double 都是 8 个字节的精确度啊？

回复 linkedin_21530339：java 的编译器会判断两个 double 是否相等是判断了类似于 fabs(a,b)<10e-10 这样的语句，而 c++则直接比他们的位，所以 double 在 C++中不能作为 key~

Code Ganker 第二種方法(用它):

```
public int maxPoints(Point[] points) {
    if (points == null || points.length == 0) return 0;
    int max = 1;
    double ratio = 0.0;
    for (int i = 0; i < points.length - 1; i++) //注意是 i < points.length – 1, 而不是 i < points.length, 因為總是要留出一個 point_j 來, 兩點才能定一條直線.
    {
```

```
        HashMap<Double, Integer> map = new HashMap<Double, Integer>();
        int numofSame = 0;
        int localMax = 1;
        //以下注意學習多個 else if 的寫法
        for (int j = i + 1; j < points.length; j++) //注意 j 是從 i+1 開始的. 我 E3 是從 0 開始的, 這樣會有重復計算.
        {
            //以下是計算 ratio. E2 是專間寫的一個函數來算 ratio, 看起來要簡潔點
            if(points[j].x == points[i].x && points[j].y == points[i].y) {
                numofSame++;
                continue;
            }
            else if(points[j].x == points[i].x)
            {
                ratio = (double) Integer.MAX_VALUE; //最好像這樣 cast 一下
            }
            else if(points[j].y == points[i].y)
            {
                ratio = 0.0;
            }
            else
            {
                ratio = (double) (points[j].y - points[i].y) / (double) (points[j].x – points[i].x);
            }

            //以下是放 map
            int num; //num 為 ratio 在 map 中對應的 value, 即有相同斜率的點的個數
            if (map.containsKey(ratio))
            {
                num = map.get(ratio)+1;

            }
            else
            {
                num = 2;
            }
            map.put(ratio, num);
        } //end of j loop

        for (Integer value : map.values()) //map.values()函數見 java 書 p834
        {
            localMax = Math.max(localMax, value);
        }
        localMax += numofSame; //別忘了這句
        max = Math.max(max, localMax);
    } //end of i looop
    return max;
}
```

计算几何的题目在现在的面试中挺常见的，可能因为有些问题比较实用的缘故，而且实现中一般细节比较多，容易出 bug，所以还是得重视。如果有朋友有更好的解法或者思路，欢迎交流哈，可以留言或者发邮件到 linhuanmars@gmail.com 给我。

150. Evaluate Reverse Polish Notation, Medium

http://blog.csdn.net/linhuanmars/article/details/21058857

Evaluate the value of an arithmetic expression in Reverse Polish Notation.
Valid operators are +, -, *, /. Each operand may be an integer or another expression.
Some examples:
  ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
  ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
Subscribe to see which companies asked this question

Key: 维护一个运算数栈，读到运算数的时候直接进栈，而每得到一个运算符，就从栈顶取出两个运算数，运算之后将结果做为一个运算数放回栈中，直到式子结束，此时栈中唯一一个元素便是结果。

History:

E1 代碼能通過(寫之前描了一眼 Code Ganker 說的用棧放數), 且跟 Code Ganker 的代碼基本上是一樣的.

E2 代碼在我電腦上可以得出正確結果, 在 LeetCode OJ 中卻總是報 java.lang.NumberFormatException, 應該是 compiler 版本的問題, 算我通過.

E3 在 OJ 上較快寫好, 本來可以一次通過的, 結果把 a, b 寫反了(見代碼中), 改後即通過. E3 代碼比 Code Ganker 的看起來要清新些, 故以後用 E3 代碼.

Wiki: 逆波蘭記法中，操作符置於操作數的後面。例如表達「三加四」時，寫作「3 4 +」，而不是「3 + 4」。如果有多個操作符，操作符置於第二個操作數的後面，所以常規中綴記法的「3 - 4 + 5」在逆波蘭記法中寫作「3 4 - 5 +」：先 3 減去 4，再加上 5。使用逆波蘭記法的一個好處是不需要使用括號。

我: 我的代碼也能通過(寫之前描了一眼 Code Ganker 說的用棧放數), 且跟 Code Ganker 的代碼基本上是一樣的, 以下用我的代碼.

Code Ganker:
这道题是逆波兰式的求解，不了解逆波兰式的朋友可以参考一下逆波兰表示法 – wiki。逆波兰式有个优点就是他不需要括号来表示优先级，直接根据式子本身就可以求解。思路是维护一个运算数栈，读到运算数的时候直接进栈，而每得到一个运算符，就从栈顶取出两个运算数，运算之后将结果做为一个运算数放回栈中，直到式子结束，此时栈中唯一一个元素便是结果。代码如下.
以上代码中有一个没有周全的地方是没有对逆波兰式错误的情况进行出错处理，其实也不难，就是每次 pop 操作检查栈空情况，如果栈空，则说明出错。还有就是最后检查一下栈的 size，如果不是 1 也说明运算数多了，返回错误。有兴趣的朋友可以自己补充一下哈。
和这道题类似的，有波兰式求解，中缀表达式求解，这几个其实是表达式的不同表达方式。既然这里出现了逆波兰式，大家还是看看其他两种的求解方法，原理其实近似，都是通过维护栈来实现，网上也有不少材料，这里就不细说了。

Code Ganker 代碼(不用):

```
 public static int evalRPN(String[] tokens) {
  if(tokens == null || tokens.length == 0)
   return 0;

  Stack<Integer> stack = new Stack<>();
```

```
  for(String str : tokens) {

   if(str.equals("+")) //注意我最開始用 str == “+”, 雖然在我電腦上可以得出正確結果, 但在 leetcode 中通不過. 記住 leetcode 中不要用 string == “....”, 而要用 str.equals(“....”). 見 java 書 p410.
    stack.push(stack.pop() + stack.pop());

   else if(str.equals("-"))
    stack.push(-stack.pop() + stack.pop());

   else if(str.equals("*"))
    stack.push(stack.pop() * stack.pop());

   else if(str.equals("/")) {
    int b = stack.pop();
    int a = stack.pop();
    stack.push(a / b); //我最開始還寫了處理 b=0 時之情況, 但看到 Code Ganker 沒寫, 就把我的也刪了, 可以通過.

   } else {
    stack.push(Integer.parseInt(str));
   }

  }

  return stack.pop();
 }
```

E3 代碼(用它):

```
    public int evalRPN(String[] tokens) {
        if(tokens == null || tokens.length == 0) return 0;
        LinkedList<Integer> stack = new LinkedList<>();

        for(String s : tokens) {
            if(s.equals("+") || s.equals("-") || s.equals("*") || s.equals("/")) {
                int a = stack.pop();
                int b = stack.pop();
                stack.push(calculate(b, a, s)); //E3 最開始將它寫誤為了 calculate(a, b, s).
            } else {
                stack.push(Integer.parseInt(s));
            }
        }

        return stack.pop();
    }

    private int calculate(int a, int b, String operator) {
        if(operator.equals("+")) return a + b;
        else if(operator.equals("-")) return a - b;
        else if(operator.equals("*")) return a * b;
        else return a / b;
```

}

151. Reverse Words in a String, Medium

http://blog.csdn.net/linhuanmars/article/details/20982463

Given an input string, reverse the string word by word.
For example,
Given s = "the sky is blue",
return "blue is sky the".
Update (2015-02-12):
For C programmers: Try to solve it *in-place* in *O*(1) space.
click to show clarification.
Clarification:

- What constitutes a word?
  A sequence of non-space characters constitutes a word.
- Could the input string contain leading or trailing spaces?
  Yes. However, your reversed string should not contain leading or trailing spaces.
- How about multiple spaces between two words?
  Reduce them to a single space in the reversed string.

Subscribe to see which companies asked this question

Key: 遞歸. 寫一個 helper(s, index)函數, 其作用為: 返回的是將「s 以 index 開始(包括 index)的子串按題目要求返轉」. 在 helper 中 遞歸調用自己(即對以下一個單詞為開始的子串調用 helper). E3 不建議用遞歸, 而就用本題 II 的方法, 見下面紅字. 但也要看看遞歸, 防止面試時 follow up 要求用遞歸.

本題與 186. Reverse Words in a String II, Medium 的題意不同之處在於(看題目中的 Clarification):

1. 本題的輸入字符串有 leading or trailing spaces, 而 186 題沒有.

2. 本題的輸入字符串有 multiple spaces between two words, 而 186 題沒有.

3. 本題空簡複雜度可以為 O(n), 而 186 題要求為 O(1).

History:
E1 直接看的答案.
E2 沒花多久就寫好了, 改了兩次就通過了, 沒完全按照 E1 留的 key 寫(以上的 key 是 E2 寫的), 但方法跟 Code Ganker 差不多, 但代碼比 Code Ganker 的簡短好懂, 以後用 E2 的代碼(在最後).
E3 改了幾次後通過. E3 沒用遞歸, 而是用的本題 II 中一樣的方法, 唯一的不同就是 r 為空格且下一個也為空格時, 就將它下一個空格刪掉, 直到它下一個不再是空格. 以後還是盡量不用遞歸, 一來遞歸不好記憶, 二來一般人也覺得遞歸不好. E3 代碼沒帖, 因為亂且略長, 自己應該能寫出來. 另外, 本題應該還可以用 string.split(“多個空格”), 然後將 split 出來的 string 一個個反轉, 但此方法看起來用點像耍賴, 所以沒試.

我: William 有一個很短的代碼, 但我看不大懂, 以下還是用 Code Ganker 的代碼, 思路上要直接些.

Code Ganker:
这道题是字符串处理的题目，我们先介绍一种很直接的做法，就是类似于 java 中 String::split 函数做的操作，把字符串按空格分开，不过我们把重复的空格直接忽略过去。接下来就是把得到的结果单词反转过来得到结果。因为过程中就是一次扫描得到字符串，然后再一次扫描得出结果，所以时间复杂度是 O(n)。空间上要用一个数组来存，所以是 O(n) (我: 注意不是所有遞歸法的空間都是 O(log n), 本題即 O(n+nlogn)=O(n), 若不算

結果佔的空間, 就應當是 O(log n))。实现思路比较清晰，这里就不列举迭代实现的代码了，有兴趣的朋友可以练习一下哈。下面的代码是这种方法的递归实现：

```
public String reverseWords(String s) {
    s = s.trim(); //trim 見 Java 書 p409
    return helper(s,0).toString();
}

//helper(s, index)的作用為: 返回的是將「s 以 index 開始(包括 index)的子串按題目要求返轉」, 並在返轉後加一個空格(若 s=0 則不加) 後的字符串.
private StringBuilder helper(String s, int index)
{
    if(index>=s.length())
        return new StringBuilder(); //Code Ganker: 这个是结束条件，就是走到字符串末尾了，只需要返回一个空串就可以了，后面就可以倒序 append 到这个上面就行了 ~
    StringBuilder cur = new StringBuilder();
    int lastIndex = index; //記錄最初輸入的 index, 因為後面 index 的值為改變

    //此時 index 指向一個單詞的開始, 如 how   are you 中的 h. 注意遞歸不變量保證了 index 指向的不是空格, 下面我會 show
    while(index < s.length() && s.charAt(index)!=' ')
    {
        cur.append(s.charAt(index++));
    }

    //此時 index 指向 how   are you 中 w 後面那個空格
    while(index < s.length() && s.charAt(index)==' ')
        index++;

    //此時 index 指向 how   are you 中的 a. 注意這裡維護了「下面調用 helper 時, index 指向的不是空格」這樣一個遞歸不變量
//先看下下行(即 return helper(s,index).append(cur).append(' ');)的注釋, 再看以下的 if. lastIndex 等於最初輸入的 index. 此 if 是在處理最初輸入的 index 為 0(即 reverseWords 中調用時)之情況. 即加入原字符串(輸入的那個字符串 s)的第一個單詞, 此時顯然不能在後面加空格, 本 if 就這麼個作用.
    if(lastIndex == 0)
        return helper(s,index).append(cur);

//以下一句的作用: 根据 helper()的作用, helper(s, index)返回的是將「s 以 index 開始(包括 index)的子串按題目要求返轉」, 並在返轉後加一個空格 後的字符串. 以上面的 how   are you 為例, 以下 helper(s, index)返回的就是 you are . 以下再加一個 cur (即 how)再加一個空格後, 即為 you are how .
    return helper(s,index).append(cur).append(' ');
}
```

接下来我们再介绍另一种方法，思路是先把整个串反转并且同时去掉多余的空格，然后再对反转后的字符串对其中的每个单词进行反转，比如"the sky is blue"，先反转成"eulb si yks eht"，然后在对每一个单词反转，得到"blue is sky the"。这种方法先反转的时间复杂度是 O(n)，然后再对每个单词反转需要扫描两次（一次是得到单词长度，一次反转单词）,所以总复杂度也是 O(n)，比起上一种方法并没有提高，甚至还多扫描了一次，不过空间上这个不需要额外的存储一份字符串，不过从量级来说也还是 O(n)。代码如下 (我: 我覺得本題沒必要掌握第二種方法, 故省之)

可以看出，两种方法并没有哪种有明显优势。之所以列出第二种方法是因为有些题用这种方法确实不错，有时候可以

借鉴一下这个想法，就是先反转整体，再反转局部。比如说左旋转一个数组，用这种方法可以用常量空间搞定，大家可以想想哈。

評論:

```
if(index>=s.length())
    return new StringBuilder();
```

第一个方法，这个应该是收敛条件吧？结束是新建一个 stringbuilder?
回复亚 ISUer：这个是结束条件，就是走到字符串末尾了，只需要返回一个空串就可以了，后面就可以倒序 append 到这个上面就行了 ~
回复 Code_Ganker：嗯，每次加完要清空。。谢啦 ~ ~

E2 的代碼:

```
public String reverseWords(String s) {
    s = s.trim();
    return helper(s, 0);
}

private String helper(String s, int index) {
    while(index < s.length() && s.charAt(index) == ' ') index++; //若 index 所在的位置可能是空格, 則往後跳到非空格的字符上去. 然後以 index 作為 要反轉的 substring 的開始
    int i = index;
    while(i < s.length() && s.charAt(i) != ' ') i++; //i 跳過本單詞, 此時 i 是下一個單詞的開始(實際上是本單詞後面的空格)
    if(i == s.length()) return s.substring(index, i);
    return helper(s, i) + " " + s.substring(index, i);
}
```

152. Maximum Product Subarray, Medium

http://blog.csdn.net/linhuanmars/article/details/39537283

Find the contiguous subarray within an array (containing at least one number) which has the largest product.
For example, given the array [2,3,-2,4],
the contiguous subarray [2,3] has the largest product = 6.
Subscribe to see which companies asked this question

Key: 本題實際上就是 53. Maximum Subarray 的 II. 方法跟 53 題差不多. 區別是 local 和 nums[i]都有可能為負, 當它們都為負時, local * nums[i]就得到一個局部最大. 所以要用兩個 local: localMax 和 localMin. 式子相當於是將 53 題的 local = Math.max(nums[i], local + nums[i])中 max 的第二項 update 一下, 第一項仍要保持為 nums[i]. 以下劃線兩式要記住:

```
for(int i=1;i<A.length;i++) {
    int max_copy = max_local;
    max_local = {A[i], A[i]*max_local, A[i]*min_local} 這三者最大的;
```

```
        min_local = {A[i], A[i]*max_copy, A[i]*min_local} 這三者最小的;
        global = Math.max(global, max_local);
    }
```

可用以下兩例來想(俱體的證明不要想).

例一:

```
// nums       -4  -3  -2
// localMax -4  12   6
// localMin -4  -3 -24
// return 12
```

例二:

```
// nums       -2   0  -1
// localMax  -2  0   0
// localMin  -2   0  -1
// return 0
```

History:
E1 直接看的答案.
E1 改了幾次後通過, 代碼跟 Code Ganker 的基本一樣. E1 基本上沒留下甚麼 key, 以上的 key 都是 E2 寫的.
E3 通過, 是試對的, 原理不是完全清楚, 但也知道大概. E3 代碼跟 Code Ganker 有點區別, 帖在最後, 可參考.

这道题跟 Maximum Subarray 模型上和思路上都比较类似，还是用一维动态规划中的“局部最优和全局最优法”。这里的区别是维护一个局部最优不足以求得后面的全局最优，这是由于乘法的性质不像加法那样，累加结果只要是正的一定是递增，乘法中有可能现在看起来小的一个负数，后面跟另一个负数相乘就会得到最大的乘积。不过事实上也没有麻烦很多，我们只需要在维护一个局部最大的同时，在维护一个局部最小(我: 為何是局部最小, 見下面我的講解)，这样如果下一个元素遇到负数时，就有可能与这个最小相乘得到当前最大的乘积和，这也是利用乘法的性质得到的。代码如下.

这道题是一道很不错的面试题目，因为 Maximum Subarray 这道题过于常见了，所以可能大部分人都做过，这道题模型类似，但是又有一些新的考点，而且总体还是比较简单，无论是思路上还是实现上，又能考察动态规划，个人还是比较喜欢的哈。

Code Ganker 代碼(用它):

```
public int maxProduct(int[] A) {
    if(A==null || A.length==0)
        return 0;
    if(A.length == 1)
        return A[0];

    int max_local = A[0];
    int min_local = A[0];
    int global = A[0];

    for(int i=1;i<A.length;i++) //注意是從 i=1 開始. 為何? 因為下面要用上一輪的 max_local (即 A[0])等, which 已經在前面初始化過了.
    {
```

```
        int max_copy = max_local; //max_copy 是什麼鬼? 見下面我的解說. E2 的代碼寫此不謀而合
        max_local = Math.max(Math.max(A[i]*max_local, A[i]), A[i]*min_local); //這是在 A[i],  A[i]*max_local, A[i]*min_local 三者中找最大的, 為何還是 A[i]乘以 max_local 和 min_local?  因為 max_local 和 min_local(若 minl_local 為負)是絶對值最大的兩個數. 注意跟 A[i]相乘的 max_local 和 min_local 都應當是上一輪的, 所以前面才會弄個 max_copy = max_local, 給下面 min_local 用
        min_local = Math.min(Math.min(A[i]*max_copy, A[i]), A[i]*min_local); //這是為下一輪算的 min_local, 本輪用不上
        global = Math.max(global, max_local);
    }
    return global;
}
```

E3 代碼(不用, 可參考):

```
    public int maxProduct(int[] nums) {
        if(nums == null || nums.length == 0) return 0;
        int localPos = nums[0], localNeg = nums[0], global = nums[0];
        int localPosPrev = localPos;

        for(int i = 1; i < nums.length; i++) {
            localPos = (nums[i] > 0 ? Math.max(nums[i], localPos * nums[i]) : nums[i] * localNeg);
            localNeg = (nums[i] > 0 ? localNeg * nums[i] : Math.min(nums[i], nums[i] * localPosPrev));
            global = Math.max(global, localPos);
            localPosPrev = localPos;
            System.out.println(i+" "+localPos+" "+localNeg);
        }

        return global;
    }
```

153. Find Minimum in Rotated Sorted Array, Medium

http://codeganker.blogspot.com/2014/10/find-minimum-in-rotated-sorted-array.html

Suppose a sorted array is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
Find the minimum element.
You may assume no duplicate exists in the array.
Subscribe to see which companies asked this question

Key: 二分法. 本題中的元素是沒有重復的, 本題的 II 即元素有重復之情況 最小值總是在坎坎那裡. 但 Code Ganker 的思想是不管這段中有沒有坎坎, 都有一個最小值在那裡, 最後所有最小值中最小那個. 如果左半边有序，那么左半边最小就是左边第一个元素，可以和当前最小相比取小的(實際上相當於扔掉了左半邊, 因為左半邊的最小值已經掌在手裡了)，然后考查向右半边。否则(即若左半邊無序)，那么就是右半半边第一个元素，然后考查左半边。这样子每次可以截掉一半元素，所以最后复杂度等价于一个二分查找

Extra: 對於 rotated sorted array 這類題目, 對於輸入數組 A 的某段 A[start, ...end], 記住以下結論(以下的 focus 是找坎坎):

若元素無重復, 若 A[start] < A[end], 則該段是 sorted (用手機信號那樣的柱狀圖稍想就知道(若有坎坎, 則必有 A[start]>A[end])), 否則若 A[start] > A[end], 則坎坎必在此段.

若元素有重復(本題 II 之情況), 則用 81. Search in Rotated Sorted Array II 題一樣的 trick(即移動邊界). 做本題 II 時再看.

History:
E1 幾分鍾寫好, 一次通過. 但我的方法基本上是 brute forece 的: 從頭開始搜索, 直到走到坎坎了, 就返回那裡的數, 就是最小數. 本方法的時間複雜度最壞時為 O(n). 所以以下不用我的代碼.
E2 幾分鍾寫好, 一次通過, 用的以上 key 中算法, 且用的九章模版. 九章用的不同的算法, 難得看, 就沒看.
E3 很快寫好, 本來可以一次通過, 但不小心將 nums[r]寫成了 r, 暈, 改後即通過. E3 跟 E2 代碼基本上一樣, 但為了跟前面和後面的題一致, 還是用 E3 代碼.

Code Ganker 的頁面今天打不開, 提示是「服務器維護中」, 所以我每道題都把解答 copy 到本 word 中是多麼明智. 後來在 http://codeganker.blogspot.com/中找到了 Code Ganker 的代碼(但無評論). 等會兒後 Code Ganker 的頁面可以打開了. 但好處是我發現 http://codeganker.blogspot.com/中的代碼複製過來是有顏色的.

我最開始因為 Code Ganker 頁面打不開而用的 William 的代碼, William 的代碼比 Code Ganker 的短很多, 但它必須是 l = m +1, r = m, 否則通不過, 不好太好理解和記憶. Code Ganker 的代碼要清新些, 統一是 l = m + 1, r = m - 1. 而且也可以和 33. Search in Rotated Sorted Array, Hard 和 81. Search in Rotated Sorted Array II, Medium 一致, 所以還是換成了用 Code Ganker 的代碼. 所以還是九章模版好, 一律都是 l=m, r=m.

Code Ganker:

这道题是 Search in Rotated Sorted Array 的扩展，区别就是现在不是找一个目标值了，而是在 bst 中找最小的元素。主要思路还是跟 Search in Rotated Sorted Array 差不多，还是通过左边界和中间的大小关系来得到左边或者右边有序的信息，如果左半边有序，那么左半边最小就是左边第一个元素，可以和当前最小相比取小的，然后走向右半边。否则，那么就是右半半边第一个元素，然后走向左半边。这样子每次可以截掉一半元素，所以最后复杂度等价于一个二分查找，是 O(logn)，空间上只有四个变量维护二分和结果，所以是 O(1)。代码如下.

有朋友可能注意到上面的实现还有第三种情况，就是左边界和中间是相等的情况，这道题目其实是不用发生的，因为元素没有重复，而 Find Minimum in Rotated Sorted Array II 这道题这考虑了元素重复的情况，这里只是为了算法更加一般性，就把那个情况也包含进来(我: 所以 154 題的代碼與本題一模一樣)，有兴趣的朋友可以看看 Find Minimum in Rotated Sorted Array II 对重复元素的分析哈。

大牛，想问下为什么条件判断是 while(l<r-1)？Binary Search 本身方法很简单，主要是不停的去除不符合的一半，但是每次条件判断都很迷茫。求大牛点拨！
回复 rubyhuang87：这里是特殊情况哈~ 是为了让它停留在有可能最小的两个元素上，所以条件放松了一步，这样 l 和 r 就停在二分查找最后的两个元素，最小的元素肯定在其中之一~
我: 154 題評論中同樣的問題:
回复 sinat_22785087：这里的问题是这样的，原先因为是搜索某一个元素，所以可以夹逼到找到元素或者他们交错为止，现在的主要问题是如果 l 和 r 相差 1 以内，那么 m 就会等于 l，如此下面的条件就会跳过 r，而最小元素可能就在 r 里面，所以终止条件才改成 l<r-1 使得夹逼就停止在最后两个元素中哈 ~
回复 linhuanmars：哎，为不美的程序找借口。需要在循环外（之前，之后）单独处理某些元素几乎都是不美的程序，就是逻辑没好透。既然知道(l + r) / 2 会偏向 l，循环中改用 r 做判断条件就可以避免跳过 r 的问题了不是么。

不用维护 min，二分之后只有两种情况，一旦发现子数组都有序，右子数组的第一个元素就是解，直接返回它（考虑到给的数组可能压根没 rotate 过，那也只需要取左右子数组的第一个元素小的那个返回）；一边有序另外一边无序，去无序的那边找。循环退出后的停在哪就是解。(我: 這就是 Wiliam 的方法, 還有人貼了 William 的代碼, 但我還是覺得按 Code Ganker 這樣寫比較好)

Code Ganker 的代碼(不用):

```
public int findMin(int[] num) {
    if(num == null || num.length==0)
        return 0;
    int l = 0;
    int r = num.length-1;
    int min = num[0];
    while(l<r-1)//為何循環條件不像 33 題那樣寫成 l<=r? 見下面我說的.
    {
        int m = (l+r)/2;
        if(num[l]<num[m])
        {
            min = Math.min(num[l],min);//此時坎坎在右半段(注意最小值總是在坎坎那裡), 但為何還要取左半段的最小值? 因為 Code Ganker 的思想是不管這段中有沒有坎坎, 都最一個最小值放那裡, 最後所有最小值中最小那個, 就是正確答案了.
            l = m+1; //不要寫成了 l++
        }
        else if(num[l]>num[m])
        {
            min = Math.min(num[m],min); //不要寫成了 Math.min(nums[r], min)
            r = m-1;
        }
        else
        {
            l++;
        }
    }
//為何還要做以下兩個比較? 因為循環結束後, l 和 r 是不相等的(這點與 33 題不同), 此時 l=r-1, 注意此時不會再进入循環了, 即 l 到 r 那段(實際上就 l 和 r 兩個數)沒有被處理. 所以最後要再拿它們跟 min 比較一下. 至於為何循環條件不像 33 題那樣寫成 l<=r, 想一想特殊情況[2,1]就知道了. 其實 William 的代碼「必須是 l = m +1, r = m, 否則通不過」部分也是因為這個原因.
    min = Math.min(num[r],min);
    min = Math.min(num[l],min);
    return min;
}
```

E2 的代碼(不用):

```
public int findMin(int[] nums) {
    if(nums == null || nums.length == 0) return 0;
    int min = Integer.MAX_VALUE;
    int l = 0, r = nums.length - 1;

    while(l + 1 < r) {
        int m = (l + r) / 2;

        if(nums[m] < nums[r]) {
            min = Math.min(min, nums[m]);
            r = m;
```

```
            } else {
                min = Math.min(min, nums[l]);
                l = m;
            }
        }

        return Math.min(min, Math.min(nums[l], nums[r]));
    }
```

E3 的代碼(用它. 其實跟 E2 代碼基本上一樣, 但為了跟前面和後面的題一致, 還是用 E3 代碼):

```
    public int findMin(int[] nums) {
        if(nums == null || nums.length == 0) return 0;
        int n = nums.length;
        int l = 0, r = n - 1;
        int min = Integer.MAX_VALUE;

        while(l + 1 < r) {
            int m = (l + r) / 2;

            if(nums[l] > nums[m]) {
                min = Math.min(min, nums[m]);
                r = m;
            } else {
                min = Math.min(min, nums[l]);
                l = m;
            }
        }

        return min = Math.min(min, Math.min(nums[l], nums[r]));
    }
```

154. Find Minimum in Rotated Sorted Array II, Hard

http://codeganker.blogspot.com/2014/10/find-minimum-in-rotated-sorted-array-ii.html

Follow up for "Find Minimum in Rotated Sorted Array":
What if *duplicates* are allowed?
Would this affect the run-time complexity? How and why?
Suppose a sorted array is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
Find the minimum element.
The array may contain duplicates.
Subscribe to see which companies asked this question

Key: 二分法. 用 81. Search in Rotated Sorted Array II 題一樣一樣的 trick, 移動邊界即可. 即只加兩句, 就是跟 81. Search in Rotated Sorted Array II 中一模一樣的兩句紅色 while.

History:

E1 直接看的答案.
E2 最開始受 'E1 寫的對重復元素的討論(已刪)' 的誤導, 想錯了方向, 搞了半天還沒寫出來. 後來發現就用 81 題一樣的 trick 即可, 然後通過, 代碼只比本題的 I 多了一兩句. 為了與本題的 I 一致, 以後都用 E2 的代碼.
E3 是在本題 I 的代碼中加了那兩個紅色的 while, 一次通過.

这道题是 Search in Rotated Sorted Array 的扩展，思路在 Find Minimum in Rotated Sorted Array 中已经介绍过了，和 Find Minimum in Rotated Sorted Array 唯一的区别是这道题目中元素会有重复的情况出现。不过正是因为这个条件的出现，影响到了算法的时间复杂度。原来我们是依靠中间和边缘元素的大小关系，来判断哪一半是不受 rotate 影响，仍然有序的。而现在因为重复的出现，如果我们遇到中间和边缘相等的情况，我们就无法判断哪边有序，因为哪边都有可能有序。假设原数组是{1,2,3,3,3,3,3}，那么旋转之后有可能是{3,3,3,3,3,1,2}，或者{3,1,2,3,3,3,3}，这样的我们判断左边缘和中心的时候都是 3，我们并不知道应该截掉哪一半。解决的办法只能是对边缘移动一步，直到边缘和中间不在相等或者相遇，这就导致了会有不能切去一半的可能。所以最坏情况就会出现每次移动一步，总共移动 n 此，算法的时间复杂度变成 O(n)。代码如下(我: 與 153 題一模一樣, leetcode 中兩題都可通過)

在面试中这种问题还是比较常见的，现在的趋势是面试官倾向于从一个问题出发，然后 follow up 问一些扩展的问题，而且这个题目涉及到了复杂度的改变，所以面试中确实是一个好题，自然也更有可能出现哈。

E2 的代碼(不用):

```
public int findMin(int[] nums) {
    if(nums == null || nums.length == 0) return 0;
    int min = Integer.MAX_VALUE;
    int l = 0, r = nums.length - 1;

    while(l + 1 < r) {
        int m = (l + r) / 2;

        if(nums[m] < nums[r]) {
            min = Math.min(min, nums[m]);
            r = m;
        } else if(nums[m] > nums[r]) {
            min = Math.min(min, nums[l]);
            l = m;
        } else {
            while(r != m && nums[r] == nums[m]) r--;
        }
    }

    return Math.min(min, Math.min(nums[l], nums[r]));
}
```

E3 的代碼(用它, 跟本題 I 相比, 就多了以下兩個紅色的 while):

```
public int findMin(int[] nums) {
    if(nums == null || nums.length == 0) return 0;
    int n = nums.length;
    int l = 0, r = n - 1;
    int min = Integer.MAX_VALUE;

    while(l + 1 < r) {
        while(l <= n - 2 && nums[l + 1] == nums[l]) l++;
        while(r >= 1 && nums[r - 1] == nums[r]) r--;
```

```
            int m = (l + r) / 2;

            if(nums[l] > nums[m]) {
                min = Math.min(min, nums[m]);
                r = m;
            } else {
                min = Math.min(min, nums[l]);
                l = m;
            }
        }

        return Math.min(min, Math.min(nums[l], nums[r]));
    }
```

155. Min Stack, Easy

http://blog.csdn.net/linhuanmars/article/details/41008731

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.

Subscribe to see which companies asked this question

Key: 維護兩個棧(可以用 ArrayList 來表示棧), 一個是本題需要的正常棧, 一個是保存最小值的棧. 寫 push 函數時, 若小於(此時想一想等於時怎麼辦(不用太在這點花工夫想))最小棧頂元素, 則也要放入最小棧(當然也要放入正常棧). 寫 pop 函時, 若 pop 出的數等於最小棧中的棧頂元素, 則在正常棧和最小棧中都將這個元素刪掉, 否則只在正常棧中將這個元素刪掉. 其它的函數都很簡單. 注意 ArrayList 不像 LinkedList, 沒有 push 等函數(參見 Java p793). 注意兩個棧都有可能為空.

Convention: 若 Min Stack 為空, 則 top()返回 0, pop()直接 return, getMin()返回 0.

History: E1 直接看的答案; E2 一直通不過, 後來發現是因為寫 pop 時, 沒有先來 if(stack.isEmpty()) return, 而是後面兩次判斷 stack 是否為空, 這樣導致了一個很難發現的邏輯 bug, 花了不少時間才查出來.

我: William 的方法跟 Code Ganker 差不多, 也是用的兩個棧, 用其中一個保存最小值. 但 William 是直接用的 Stack, Code Ganker 是用的 ArrayList.

Code Ganker:
这道题是关于栈的题目，实现一个栈的基本功能，外加一个获得最小元素的操作。正常情况下 top，pop 和 push 操作都是常量时间的，而主要问题就在于这个 getMin 上面，如果遍历一遍去找最小值，那么 getMin 操作就是 O(n)的，既然出出来了这道题就肯定不是这么简单的哈。比较容易想到就是要追溯这个最小值，在 push 的时候维护最小值，但是如果 pop 出最小值的时候该如何处理呢，如何获得第二小的值呢？如果要去寻找又不是常量时间了。解决的方案是再维护一个栈，我们称为最小栈，如果(push 函數)遇到更小的值则插入最小栈，否则就不需要插入最小栈（注意这里正常栈是怎么都要插进去的）。这里的正确性在于，如果(pop 函數)后来得到的值(即 pop 出來的)是大于当前最小栈顶的值的，那么接下来 pop 都会先出去(意思是在正常棧中將這個元素刪掉)，而最小栈顶的值会一直在，而当 pop 到最小栈顶的值时，一起出去(意思是在正常棧和最小棧中都將這個元素刪掉)后接下来第二小的就在 pop 之后最小栈

的顶端了。如此 push 时最多插入两个栈一个元素，是 O(1)，top 是取正常栈顶，自然是 O(1)，而 pop 时也是最多抛出两个栈的栈顶元素，O(1)。最后 getMin 只需要 peek 最小栈顶栈顶即可，所以仍是 O(1)，实现了所有操作的常量操作，空间复杂度是 O(n)，最小栈的大小。代码如下.

这道题在理清思路之后代码还是比较简单的，这里用 ArrayList 来实现栈，当然也可以用链表，不过对于时间复杂度要求比较高，所以重点是想出维护最小栈顶做法，属于比较考察站的性质的题目，是很不错的面试题目，在面经中也经常出现。

我很想知道 为什么第六行 minStack.get(minStack.size()-1)要大于等于 x？大于 x 不行吗？
回复 feelxia777：大于等于是必要的，否则(我: 此處不應該用「否則」，而應該用「因為」)有可能最小元素有两个，但是因为(若在 pop 函數中)当前弹出其中一个，minStack 也跟着弹出，最小元素的信息就丢失了~
回复 linhuanmars：明白了，讲的真的很好

为什么不直接用 stack 而用 ArrayList 那？我看很多题都是不用 stack，用 LinkedList 或者 ArrayList 来做和 stack 一样的事情。。。求解。
回复 princawang：这个都是可以的哈 ~ 因为 ArrayList 实现了 Stack 的接口，所以可以当成 Stack 来使用，是我的个人习惯而已哈 ~

楼主你好，我也用了你的这种方法，但是 leetcode 提示 Memory Limit Exceeded，刚才我直接用楼主的代码依旧是 Memory Limit Exceeded。。。是不是最近 leetcode 更改了题目的通过标准？(我: Code Ganker 的代碼在 leetcode 中可以通過)
回复 xumyselfcn：我也遇到了这个问题，后来我把 ArrayList 换成 Stack 再跑就好了，不知道是为什么，求博主解答一下哈。。。
回复 zhch1990：恩，可能是 Leetcode 标准严格了，ArrayList 空间相对 stack 可能会大点，因为他先分配了多一点的空间哈 ~

刚刚人森第一次 phone screen 给了 google。。。好紧张，45 分钟只做了一个题。
面试官给了个问题，我想要先想一想的，但是又怕没有交流会很不好，就直接写了。因为没想清楚，写的时候出了好多 bug....
求 lz 给一些应对 phone screen 的建议。。我很想把问题想清楚了再动手写，但是又担心木有交流会很不好。。
回复 toefl784580：最好还是思路想清楚，然后举一个例子跟面试官交流，然后再写代码会好一些~ 如果有把握也可以直接说思路然后写，不过写的时候最好边跟面试官沟通解释哈~

qweyxy_2 大神的讲解瞬间解决了我的问题~！
qweyxy_2 回复 qweyxy_2：感觉第六行的小于等于很重要

大神好节奏~！为啥这个问题用两个 LinkedList 就不行，会显示超过内存限制？还有，测试了下 throw exception 也不行额
回复 LostBank：恩，这是 leetcode 的问题，设置的空间限制太紧了~ LinkedList 其实会比 arrayList 多一份指针，不过我觉得不是大问题~ 不过 leetcode 过不了就没法了哈~ leetcode 应该不支持 throu exception 的吧~
回复 linhuanmars：神一般的回复速度，解释一如既往的给力!
大神四 11 快乐哇~！

```
class MinStack {
    ArrayList<Integer> stack = new ArrayList<Integer>();
    ArrayList<Integer> minStack = new ArrayList<Integer>();
    public void push(int x) {
        stack.add(x);
        if(minStack.isEmpty() || minStack.get(minStack.size()-1)>=x)//評論中 refer 到的第六行. 此句中為何是>=? 看上面評論.
//另外, 此句不要寫成了 if(!minStack.isEmpty() && minStack.get(minStack.size() - 1) >= x)
//注意 ArrayList 的 index 和數組一樣, 也是從 0 開始的, 已 record 到 Java 書 p791.
        {
```

```
            minStack.add(x);
        }
    }

    public void pop() {
        if(stack.isEmpty())
        {
            return;
        }
        int elem = stack.remove(stack.size()-1);
        if(!minStack.isEmpty() && elem == minStack.get(minStack.size()-1))
        {
            minStack.remove(minStack.size()-1);
        }
    }

    public int top() {
        if(!stack.isEmpty())
            return stack.get(stack.size()-1);
        return 0;
    }

    public int getMin() {
        if(!minStack.isEmpty())
            return minStack.get(minStack.size()-1);
        return 0;
    }
}
```

後面的題, Gode Ganker 就沒有解答了. Goodbye, Code Ganker!

156. Binary Tree Upside Down, Medium, Locked

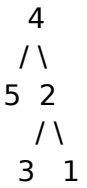
http://wlcoding.blogspot.com/2015/03/binary-tree-upside-down.html?view=sidebar

題目(題目的意思不是 CLRS 中的 left-rotation 或 right-rotation, 題目的意思看看下面 William 的圖(不是題目中的圖)就知道了):

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.
For example:
Given a binary tree {1,2,3,4,5},

```
  1
 / \
2  3
/ \
4  5
```

return the root of the binary tree [4,5,2,#,#,3,1].

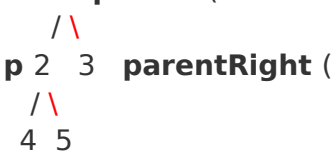


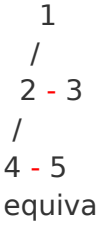
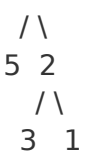
Key: 用迭代法. 看以下"Solution"一詞後的圖, 迭代 parent, parentRight 和 p 三個指針

History:
E1 直接看的答案.
E2 對題目中的 case 給出正確結果, 且代碼跟 William 的幾乎一樣, 算通過.
E3 迭代法較快寫好, 改了一次(即 while 條件中應為 哪一個節點不為 null 寫錯了)後通過(即對題目中給出的例子可以得到正確結果), 代碼跟 Wiliam 的基本上一模一樣(變量名除外).

我: Code Ganker 從此題開始就沒有解答了. 以下用 William 的代碼. William 用了迭代和遞歸兩種方法, 要掌握的是迭代法, 遞歸法也要看一看.

Solution

1. Iteration: Time ~ O(N), Space ~ O(1)
original tree:


remove right links, and add horizontal links, change to:

equivalent to:


```
// Top-down
public TreeNode UpsideDownBinaryTree(TreeNode root) {
    TreeNode p = root, parent = null, parentRight = null;
    while (p != null) {
//以下我的講解都是以 p=root(即上圖中的 1)為例. 注意此時 p, parent, 和 pranetRight 都不在上圖所示的位置. 但後面會將它們設為上圖中所示的位置.
        TreeNode left = p.left; //left 為上圖中的 2
        TreeNode right = p.right;//right 為上圖中的 3
    //假定已經將 parent, parentRight 和 p 三個指針的指向改為了上圖所示的位置(後面會改), 以下是將 p 的左右 child 設為正確的節點
```

```
            p.left = parentRight;
            p.right = parent;
         //以下是更新 parent, parentRight 和 p 三個指針, 將它們設為圖示位置, 以便下一輪使用. 這便是
iteration(迭代法)之思想.
            parent = p;
            parentRight = right;
            p = left;
        }
        return parent;
    }
```

2. Recursion: Time ~ O(N), Space ~ O(logN)

```
// Bottom-up
public TreeNode UpsideDownBinaryTree(TreeNode root) {
    return dfsBottomUp(root, null);
}

//dfsBottomUp(p, parent)的作用是: 將以 parent 為根, 以 p 為 left child 的子樹按題目要求反轉好,
並返回新樹的根.
private TreeNode dfsBottomUp(TreeNode p, TreeNode parent) {
    if (p == null)  return parent;
    TreeNode root = dfsBottomUp(p.left, p);
    p.left = (parent == null) ? null : parent.right;
    p.right = parent;
    return root;
}
```

157. Read N Characters Given Read4, Easy, Locked

http://wlcoding.blogspot.com/2015/03/read-n-characters-given-read4-i-ii.html?view=sidebar

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.

Key: 我覺得本題和 158 題難度都應當是 Medium. 本題的關鍵在於看懂題目是甚麼意思. 見我下面的黃字. 看懂題意後就不難了.
以下兩種情況時, 將不再讀入：
1. 到達文件末尾時(eof 為 true)
2. 總共讀取的字符數達到 n 個

注意 read4(buffer)中的 buffer 和 read(buf, n)中的 buf 不是同一個數組, buf 是從 buffer 中讀入的, using System.arraycopy(sourceArray, srcPos, targetArray, tarPos, length), arraycopy 函數見 Java 書 p280, 也可不看 Java 書, 看這裡就夠了.

```
size = read4(buffer)
```
每次讀入 `Math.min(n - numRead, size)`個

Reader4 這個類我 E3 已經寫好了, 直接調用就是. 測試時注意文件末尾都有一個 ascii=10 的 newline 作為結

束符.

History:
E1 直接看的答案.
E2 的代碼跟 William 的在囉輯上一樣, 本題無法在我電腦上驗證, 故算通過.
E3 改了幾次後能在我電腦上給出正確結果, 代碼跟 William 的差不多, 故算通過. E3 為了在電腦上驗證, 花了不少時間寫 read4 那個 API.

我的文件 Solution2 中已經寫好了一個虛拟的 Reader4 class, 其中含有 read4()函數, 這個 class 就在同一個文件夾中, 所以以後寫的時候直接 extend Reader4 就可以了.

我:題目的意思是: 已經提供了一個 read4(buffer)函數, 作用是讀入的字符放入數組 buffer 中, 從某哪個文件中讀不重要(在 read4 函數中已自動弄好了), 每次最多只能讀 4 個字符, 若遇到文件尾(eof), 少於 4 個字符時, 有幾個字符讀幾個. 要求寫一個 int read(char[] buf, int n)函數, 讀入的字符全放入數組 buf 中. 注意不是從 buf 中讀, 而是從某個文件中讀, 是哪個文件不重要(在 read4 函數中自動弄好了). n 為要求讀入的字符數. 若文件中的總字符數多於 n 個時, 就讀 n 個. 若遇到文件尾(eof), 即文件中的總字符數不足 n 個時, 文件中有多少個字符, 就讀多少個. 返回的是實際總共讀入的字符數. 另外, read4 在讀文件時, 會自動設一個文件的 offset(已在 API 中寫好, 不用自己寫), 比如第一次調用 read4 時, 讀到文件中的 a 處, 第二次調用 read4 時, 就從文件中 a 的下一個字符開始讀.

已有的 API：**read4(buffer)**，将读取的字符存放于 buffer 中；每次读 4 bytes，若是遇到文件尾 (eof) 少于 4 个字符时，有几个字符读几个。
要求设计 API：**read(buf, n)**，将读取的字符存放于 buf 中，一共读 n bytes，返回实际一共读到的 byte 数（当遇到 eof，可能少于 n bytes）；考虑两种情况：

- read() 只被调用一次；
- read() 会被调用多次。

I: Call read() once (我: 此 read()是指自己寫的那個 read(buf, n)函數, 而不是系統提供的 read4(buffer)函數)

Note: The read function will only be called once for each test case.

Solution

Time ~ O(N), Space ~ O(1)
考虑两个条件：

•read4(buffer) 是否到达文件末尾 (eof) ？
•一共读取的字符数是否到达 n (numRead < n)？

```
/* 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
     */
    public int read(char[] buf, int n) {//讀入的字符全放入 buf 中. 注意不是從 buf 中讀, 而
```

```
是從某個文件中讀, 是哪個文件不重要(在 read4 函數中自動弄好了). n 為要求讀入的字符數.
        char[] buffer = new char[4]; //將每次 read4 讀的字符暫時存入 buffer 中
        int numRead = 0;//numRead 是總共已經讀入的字符數
        boolean eof = false;//eof 即 end of file
        while (!eof && numRead < n) {
            int size = read4(buffer);//從文件中向 buffer 中讀入. 讀入的字符同樣放入 buffer
中, 從某哪個文件中讀不重要(在 read4 函數中自動弄好了), 每次最多只能讀 4 個字符
            if (size < 4) eof = true;
            int bytes = Math.min(n - numRead, size);//bytes 為實際要讀入的字符個數. n 為
剩下還要讀的字符個數. 當 n - numRead < size 的時候, 就只從 buffer 中讀 n - numRead 個字符,
buffer 中剩下的字符就放那裡不讀了. 注意不要將 bytes 寫成了 byte, 因為 byte 在 Java 中是一個數據類型,
 跟 int 一樣, 見 Java 書 p68.
            System.arraycopy(buffer, 0, buf, numRead, bytes);//從 buffer 中向 buf 中讀
入. arraycopy 函數見 Java 書 p280, 也可不看 Java 書, 看這個就明白了: arraycopy(sourceArray,
srcPos, targetArray, tarPos, length). 注意在 buf 中, 是從 numRead 開始放的, E2 也獨立想到了.
            numRead += bytes;
        }
        return numRead;//最後返回的 numRead 應該是等於 n 的
    }
}
```

158. Read N Characters Given Read4 II - Call multiple times, Hard, Locked

http://wlcoding.blogspot.com/2015/03/read-n-characters-given-read4-i-ii.html?view=sidebar

本題題目: 見 157 題

Key: 本題的關鍵仍是在於看懂題目是甚麼意思. 見我下面的黃字. 方法見下面 William 講的(Time 和 Space 下面那段).

History:
E1 直接看的答案.
E2 的代碼跟 William 的在囉輯上一樣, 本題無法在我電腦上驗證, 故算通過.
E3 改了好幾次才通過(在我電腦上能給出正確結果), E3 的代碼沒有 William 的好, 因為 E3 沒像 William 那樣用一個 buffersize 變量來表示 read4 弄出來的 buffer 數組是否已被 read 讀完, E3 是用其它方法判斷的, 不太直接, 別人看的話也不太好懂.

II. Call read() multiple times (我: 此 read()是指自己寫的那個 read(buf, n)函數, 而不是系統提供的 read4(buffer)函數)
Note: The read function may be called multiple times.

我:題目的意思是: 比如要讀入的文件內容為: "12345671234567abc", 調用兩次 read 函數: read(buf, 7), read(buf, 7), 其中 buf 是在 read 之外定義的一個空數組, 所以這兩次調用完 read 後, buf 中的內容應為"12345671234567" .

Solution

Time ~ O(N), Space ~ O(1)
多次调用 read() 时可能会发生：之前一次调用(read())中从 read4() 读取的字符(即 buffer 中的字符)未使用完（已经到达 n bytes），所以要将 buffer 设为全局变量，并记录其大小 buffersize(即 buffer 中還有多少個字符沒讀)和上次读到的位置 offset(的下一個, 即 offset 表示下一次開始讀的位置)，当 buffer 为空 (buffersize == 0) 时才调用 read4()，并且每次从 buffer 的 offset 开始拷贝到 buff。上一題中的 size 變量就用不著了, 就用 buffersize 就可以了.

```
/* 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
     */
    private char[] buffer = new char[4];
     private int offset = 0, bufsize = 0;//上一次調用 read 函數後, 有可能使 offset 和 bufsize 的值為非零, 這一次調用 read 函數時, 用的就是那個非零的 offset 和 bufsize 值. offset 即為上一次調用 read 函數時(在 buffer 中, 注意不是 buf 中)讀到的位置的下一個位置(即這一次就要從 offset 開始讀起走), bufsize 即上一次調用 read 函數後, buffer 中還剩下的沒讀完的字符個數, 即從 offset 開始到 buffer 結束的字符個數.

    public int read(char[] buf, int n) {
        int numRead = 0;
        boolean eof = false;
        while (!eof && numRead < n) {
            if (bufsize == 0) {
                bufsize = read4(buffer);//從文件中往 buffer 中讀入.
                eof = bufsize < 4;
            }
             int bytes = Math.min(n - numRead, bufsize); //bytes 為本次調用 read(不是 read4)實際要讀入的字符個數.
            System.arraycopy(buffer, offset, buf, numRead, bytes);//從 buffer 中向 buf 中讀入.
             offset = (offset + bytes) % 4;//(offset + bytes)其實不會超過 4, 最多就等於 4(最餘也就是想讓 4%4=0), 因為上上行給 bytes 賦的值本來就不超過 bufsize
            bufsize -= bytes;//前面已經從 buffer 中讀了 bytes 個字符, 所以 bufsize 中就要減去 bytes. 注意 bufsize 並不代表 buffer 中字符的個數, 因為除非到了文件尾, buffer 中的字符數都會是 4 個, bufsize 是用來記錄 buffer 中還有多少個字符沒讀
            numRead += bytes;
        }
        return numRead;
    }
}
```

159. Longest Substring with At Most Two Distinct Characters, Hard, Locked

http://wlcoding.blogspot.com/2015/03/longest-substring-i-ii.html?view=sidebar

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. (我： 即 e 和 c 為那兩個 distinct characters)

Key: 看以下第二種方法的講解

History:
E1 沒做出來.
E2 沒做出來, 主要是因為沒在 i-start+1 中選最大的(E1 的 key(即第二種方法的講解中的)中沒寫), 後來看了答案後加入了選最大的, 能給出正確結果及每輪的打印值, 算代碼正確, 但由於是看了答案後改的, 所以算我沒做出來.
E3 按 O(N^2)寫的, 即用 s[i,..j]掃描所有 substring, j 掃到出現三個不同字符時, 即記錄此 substring 長度並跳出 j 循環. 代碼在我電腦上對好幾個 test case 都給出正確答案, 且從原理上講也沒有不對的地方, 題目也沒說不讓 O(N^2), 所以算通過. 然後又將我的 O(N^2)代碼改了下, 成了 O(N)的 (後面我說的看不懂就算了), 即 j 掃到出現三個不同字符時, i 跳到 j 的位置. 此代碼也能對那幾個 test case 都給出正確答案, 但後來想了想, 此代碼不對, 因為 i 不能直接跳到 j 去, 因為在 j 前就有 i 應該呆的位置. 所以我還是將 William 的 O(2N)代碼默寫了一遍.

我: 以下 William 給了兩種方法, 第一種是 O(N)的, 但不好記憶. 第二種方法是 O(2N), 時間上跟第一種差別不大, 但要稍稍好懂一點, 要好記些, 且可以適用於任意個不同字符數的情況, 故以後還是要掌握第二種.

Solution

1. Three pointers: Time ~ O(N), Space ~ O(1)

我: 以下三個指針的位置為: a(start) a a a(last) b(curr)

•start - start index of substring
•curr - current (end) index of substring (我: 就是 end index of substring)
•last - index of the latest distinct character (我: last 即 curr 左側離 curr 最近的那個跟 cur 不同的元素)

curr 从 1 到 N - 1 扫，如果遇到相邻不同元素 s[curr] != s[curr-1]，则记录前一不同元素的位置 last = curr – 1 (我: 如 a a a a a(last) b(curr), 這個 last 的位置實際上是上一輪設好了的，a a a a a(last) b(curr)的狀態其實就是本輪的開始狀態)，然后(curr)继续往前走，再次碰到相邻不同元素时，有两种情况:
如果 s[curr] == s[last]，表明目前只遇到两种元素 (我: 如 a a a a a(last) b b b b a (curr))，更新 last 的位置(我: 即 last=curr-1, 如 a a a a a a b b b b(last) a (curr))后继续往前；
如果 s[curr] != s[last]，表明目前遇到了三种元素(我: 如 a(start) a a a a(last) b b b b c (curr))，则要取 substring s[start .. curr - 1] 的长度(我: 注意不能把 curr 算进去了，否則有三個不同的字符了)为 curr - start，然后将开始位置设为 last + 1(我: 如 a a a a a(last) b(start) b b b c (curr))，更新 last 的位置(我: 即 last=curr-1, 如 a a a a a a b(start) b b b(last) c (curr))后继续往前。
注意：String 到结尾的情况，最后要比较 s.length() - start 和 之前记录的 maxLen 的大小(我: 因為此時從 start 到 string 末尾都是可以的, 即不同的字符數沒超過兩個．有三種情況, 一眼就可以看出來都是這樣的：一是 a(start) a a a a(last) b b b b(current, end of string), 二是上面第一種情況運行完該輪 for 後的狀態, 即 a(start) a a a a b b b b(last) a (curr), 三是上面第二種情況運行完該輪 for 後的狀態, 即 a a a a a b(start) b b b(last) c (curr), 注意此時 start 的位置)。

```
// two distinct chars
```

```
public int lengthOfLongestSubstringTwoDistinct(String s) {
    int n = s.length();
    int start = 0, last = -1, max = 0;
    for (int curr = 1; curr < n; curr++) {//注意 curr 是從 1 開始
        if (s.charAt(curr) != s.charAt(curr - 1)) {
            if (last >= 0 && s.charAt(curr) != s.charAt(last)) {
                max = Math.max(max, curr - start);
                start = last + 1;
            }
            last = curr - 1;
        }//用這種格式（即 if 和{在同一行）也可以清楚地看到 if 包括了哪些范圍，因為可以找跟 if 對齊的那個}．好處是代碼看起來短了很多．所以這種格式只有好處，沒有壞處．
    }
    return Math.max(n - start, max);
}
```

2. Two pointer Hash Table: Time ~ O(2N), Space ~ O(1)

我: 本方法簡單地說, 就是用 start 和 i 兩個指針, 來表示 sliding window 的左端和右端, 即[start, ...i]. i 不停地往右移, 移到移到, 當發現此 sliding window 中不同的字符數大於 k 時, 就將 start 往右移, 好讓此 sliding window 中不同的字符數 回到 k.

以下解法可以推广到最多 k 个不同的字符。
用一个 Hash Table 记录每个字符出现的个数(我: 此 Hash Table 即下面的數組 count, count[c]的 index c 為字符的 ASCII 碼(注意輸入數組可能含有任何 Extended ASCII 字符, which has 256 個字符), count[c]的值為該字符出現的次數. 由此可知 Hash Table 不一定就是 HashMap)
当 count[c] == 0，表明是第一次遇到的新字符，则 numDistinct++，并更新 count[c]；(我: numDistinct 為目前為止出現的不同字符的個數)
当 numDistinct > k 时，表明已扫过的 substring 中包含了 k + 1 种不同字符，此时收缩头指针 start(我: 找出一個「只出現了一次的子符」的下一個字符作為新的 start, 即扔掉了那個「只出現了一次的子符」)，同时更新 numDistinct 和 count[c]，當 numDistinct == k 时得到符合条件的 substring s[start .. i]，取长度为 i - start + 1。這樣便得到了一個 numDistinct == k 的字串.

當然, 若對若個 i, numDistinct 本來就為 k, 也要取長度 i - start + 1.

最終在所有的 i - start + 1 中取最大的, 即為結果.

例如, 對於題目中的例子, 輸入 s = "eceba", k=2, 則每輪中的 start 和 i 打印出來為(以下每一行都是 numDistinct=k 的):

```
//0 1
//0 2 (最後選中了這個)
//2 3
//3 4
```

```
// k distinct chars
public int lengthOfLongestSubstringKDistinct(String s, int k) {
     int[] count = new int[256]; // ASCII //There are 128 in the Standard ASCII character set, and there are 256 characters in the Extended ASCII character set.
    int start = 0, numDistinct = 0, maxLen = 0;
    for (int i = 0; i < s.length(); i++) {
        if (count[s.charAt(i)] == 0) numDistinct++;//注意 if 管不到下一句
        count[s.charAt(i)]++;

//實際上只要前面 numDistinct++中使得 numDistinct 等於 k+1, 就會进入下面的 while 循環, 所以下面
```

```
的 while 中, numDistinct 總是等於 k+1 的. 一旦當 numDistinct 等於 k+1 時, 就要開始右移 start 了,
好讓 numDistinct 回到 k. while 中的幾句的目的就是找出一個「只出現了一次的子符」的下一個字符作為新
的 start, 即扔掉了那個「只出現了一次的子符」,這樣才能使當前的子串中不同字符的個數減少一個(重回 k).
這樣是可以找出所有 numDistinct=k 的子串的, 證明為(可以不看): 若本 numDistinct=k 的子串已找到,
則下一個 numDistinct=k 的子一定可以按這樣找到, 細節自己想.
若下面的 AB 兩句沒有沒 comment out, 則對於輸入 s = "eceba", k=2, 打印出來的為:
//0 1
//0 2 (最後選中了這個)
//2 3
//3 4
注意 A 句其實可以不要, 因為在 B 句處, 總是有 numDistinct=k 的.
        while (numDistinct > k) {
            count[s.charAt(start)]--;//另忘了這句
            if (count[s.charAt(start)] == 0) numDistinct--;//注意 if 管不到下一句
            start++;
        }

        //if(numDistinct == k) //A 句
            //System.out.println(start+" "+i); //B 句

        maxLen = Math.max(i - start + 1, maxLen);
    }
    return maxLen;
}
```

160. Intersection of Two Linked Lists, Easy

http://wlcoding.blogspot.com/2015/03/intersection-of-two-linked-lists.html?view=sidebar

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

For example, the following two linked lists:

A: a1 → a2
↘
c1 → c2 → c3
↗
B: b1 → b2 → b3

begin to intersect at node c1.

Notes:

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

Credits:

Special thanks to @stellari for adding this problem and creating all test cases.

Subscribe to see which companies asked this question

Key: 先分别找到 list A 和 list B 的长度，得到它们的长度差 diff。然后长的 list 先走 diff，然后再 A 和 B 一起走，A 和 B 相等时为相遇点

History:

E1 幾分鍾寫好, 一次通過.
E2 很快寫好, 一次通過.
E3 很快寫好, 方法跟 key 一樣, 本來可以一次通過的, 結果不小心將 headA = null 的 corner case 沒處理對, 此時本來應當返回 null, 我腦殼有包, 返回了 headB. 改後即通過. E3 還不如 E1 和 E2.

我寫代碼前瞄了一眼 William 說的「先分别找到 list A 和 listB 的长度，得到它们的长度差 diff」, 然後我的代碼幾分鍾寫好並一次通過, 方法跟 William 一樣, 但代碼比 William 的略囉嗦, 以下用 William 的代碼.

Solution

Two pointers: Time ~ O(n), Space ~ O(1)
先分别找到 list A 和 list B 的长度，得到它们的长度差 diff。
然后长的 list 先走 diff，然后再 A 和 B 一起走，两个 node 相等时为相遇点。

```
public static class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    ListNode pA = headA, pB = headB;
    int lenA = 0, lenB = 0;
    while (pA != null) {
        pA = pA.next;
        lenA++;
    }
    while (pB != null) {
        pB = pB.next;
        lenB++;
    }

    pA = headA;
    pB = headB;
    for (int i = 0; i < Math.abs(lenA - lenB); i++) {
        if (lenA > lenB)    pA = pA.next;
        else                pB = pB.next;
    }
    while (pA != pB) {//注意使用 pA != pB. E2 用的 pA != null, 也能寫出來. E3 也用的 pA != null.
        pA = pA.next;
        pB = pB.next;
    }
    return pA;
}
```

161. One Edit Distance, Medium, Locked

http://wlcoding.blogspot.com/2015/03/edit-distance.html?view=sidebar

Given two strings S and T, determine if they are both one edit distance apart.

Key: 不是很難. 考查是不是 append, modify(改一個字符)或 insert 這三種情況的一種:
Append: S = "abcde", T = "abcdeX"
Modify: S = "abcde", T = "abXde"
Insert: S = "abcde", T = "abcXde"
先走到不同的字符處(若 S 已走完, 則為 append), 然後要判斷是否為 modify 或 insert. 盡量不要看下面 William 說的, 否則就沒意思了(E2 看過).
Corner case: s 和 t 相同的時候.

History: E1 直接看的答案.
E2 一次通過(所有我自己的 test case), 代碼跟 William 差不多, 比 William 的略簡明, 但由於本題是 locked 不能判斷是否完全正確, 所以以後還是用 William 的代碼.
E3 一次通過(所有我自己的 test case), 方法跟 William 的不同, 我是先分'S 和 T 長度相等'和'長度差 1'兩種情況, 然後分別單獨處理這兩種情況, 然後用一個 isFirst 變量來判斷是否為首次出現不同. 感覺沒有 William 的方法好.

Solution

Time ~ O(N), Space ~ O(1)
One Edit Distance 有三种情况：
Append: S = "abcde", T = "abcdeX"
Modify: S = "abcde", T = "abXde"
Insert: S = "abcde", T = "abcXde"
从左右向右扫短的 String，如果字符相同则向前走，循环跳出时先判断是不是到达该 String 的结尾，如果到达且另一个 String 只有一个字符未读，则是 Append case；
如果没到结尾，则找到不同的 characters，这时有两种情况：
如果是 Modify case，则两个 String 的长度必须相等，且跳过该字符继续往后扫，其他字符都相同；
如果是 Insert case，则两个 String 的长度差 1，且跳过长 String 的该字符后，其他字符都相同。
扫完后判断是否到该 String 的结尾，如果到则表示是上述两种的 case 中的一种，否则则不是。

```
public boolean isOneEditDistance(String s, String t) {
    int M = s.length(), N = t.length();
    if (M > N)  return isOneEditDistance(t, s); //所以以下都只用考慮 M <= N
    if (N - M > 1)  return false;

    int i = 0;
    while (i < M && s.charAt(i) == t.charAt(i)) i++; //注意 while 只能管到 i++. 此 while 運行完了後, 要麼有 i=M, 要麼有 s.charAt(i) != t.charAt(i)
    if (i == M) return N - M == 1;   // Append case //E2 的此 if 與此一模一樣, 注意不能直接 return true, 因為有 s 和 t 相等的情況(此時 N-M=0).
//以下都是 i<M(否則上一個 if 中就 return 了)且 s.charAt(i) != t.charAt(i)
    if (M == N) {                    // Modify case
        i++;//跳過不相等的那個字符
        while (i < M && s.charAt(i) == t.charAt(i)) i++;//注意 while 只能管到 i++
    } else /* if (N - M == 1) */ {   // Insert case//注意以下必有 N-M=1, 否則前面 if(N-M>1)處已經 return 了
        while (i < M && s.charAt(i) == t.charAt(i + 1)) i++;
    }
```

```
    return i == M;
}
```

162. Find Peak Element, Medium

http://siddontang.gitbooks.io/leetcode-solution/content/array/find_peak_element.html

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 (tao: already clicked).
Note:
Your solution should be in logarithmic complexity.
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 二分查找. 題意是數組元素無重復. 若 nums[m]兩邊的數(即 m-1 和 m+1 處的數)都比它小, 則返回 m. 方法是: 若 m 左邊的比它大(即 nums[m-1] > nums[m]), 則左半段必有 peak(因為 num[-1]=-∞), 此時丟掉右半段(即 r=m). 否則若 m 右邊的比它大, 則右半段必有 peak, 此時丟掉左半段.

Corner case:
1. 整個數組都是單調遞增或單調遞減 之情況. 注意題目中說的是 num[-1] = num [n] = -∞(不是 num[0] = num [n-1] = -∞), 所以當整個數組都是單調遞增時, peak 就是 num[n-1], 當整個數組都是單調遞減時, peak 就是 num[0].
2. {2,1,2}

History:
E1 的思路基本上是對的(跟 William 和其他人是一樣的), 但我的代碼對 整個數組單調的情況 沒處理好, 所以老是通不過.
E2 很快寫好, 本來可以一次通過, 但不小心讓一個 if 搶了另一個 if 的風頭, 使得代碼對{2,1,2}這種情況給出錯誤結果. 後來添了一個 else 單詞, 即通過. E2 的代碼不用考門考慮 整個數組單調的情況(嚴格地講, 最後的 return 語句中包含了對這種情況的討論), 且 E2 用的九章模版. 九章的代碼跟我 E2 相似, 尤其是最後的 return 語句, 跟我不謀而合, 以下不帖九章代碼.
E3 十分鍾寫好, 一次通過. E3 的代碼跟 E2 基本一樣, 但比 E2 略簡潔, 故以後用 E3 代碼.

我: 以下用 siddontang 的代碼, 他的原代碼是 C++的, 以下是我改編後的 Java 版. William 的方法是一樣的(他說 Time ~ O(logN), Space ~ O(1)), 但他的代碼中是 l=m+1, r=m, 我一直不大喜歡這樣的二分查找代碼風格, 所以沒用 William 的代碼.

以下是 siddontang 說的:

这题要求我们在一个无序的数组里面找到一个 peak 元素，所谓 peak，就是值比两边邻居大就行了。

对于这题，最简单地解法就是遍历数组，只要找到第一个元素，大于两边就可以了，复杂度为 O(N)。但这题还可以通过二分来做。

首先我们找到中间节点 mid，如果大于两边返回当前 index 就可以了，如果左边的节点(我: 即 mid-1 位置的

節點)比 mid 大，那么我们可以继续在左半区间查找，这里面一定存在一个 peak，为什么这么说呢(我: 簡單地說就是因為 num[-1]=-∞, 本段後面說的都很弱, 不用看)？假设此时的区间范围为[0, mid - 1]， 因为 num[mid – 1]一定大于 num[mid]了(我: 這就是為何代碼中是 r=m-1, 而不是 r=m)，如果 num[mid - 2] <= num[mid - 1]，那么 num[mid - 1]就是一个 peak。如果 num[mid - 2] > num[mid - 1]，那么我们就继续在[0, mid - 2]区间查找，因为 num[-1]为负无穷，所以最终我们绝对能在左半区间找到一个 peak。同理右半区间一样。

Code Ganker 的代碼(不用):

```
 public int findPeakElement(int[] nums) {
  int len = nums.length;

  if(len == 1)
   return 0;

  int l = 0, r = len - 1;

  while(l <= r) {
   int m = (l + r) / 2;

   if((m == 0 || nums[m - 1] < nums[m]) && (m == len - 1 || nums[m] > nums[m + 1])) //此句寫得很好, m=0 && nums[m] > nums[m + 1]就是 整個數組都是單調遞增 之情況. 注意若將 整個數組都是單調遞增或遞減 的情況單獨分出來寫, 也不容易通過, 因為 nums[m-1] < nums[m]中的 m-1 可能小於 0. 所以這句是一舉多得.
    return m;
   else if(m > 0 && nums[m - 1] > nums[m])
    r = m - 1;
   else
    l = m + 1;
  }

  return l;
 }
```

E2 的代碼(不用):

```
    public int findPeakElement(int[] nums) {
        if(nums == null || nums.length == 0) return -1;

        int l = 0, r = nums.length - 1;

        while(l + 1 < r) {
            int m = (l + r) / 2;

            if(nums[m] > nums[m - 1] && nums[m] > nums[m + 1]) return m;
            if(nums[m - 1] > nums[m]) r = m;
            else if(nums[m + 1] > nums[m]) l = m;
        }
```

```
        return nums[l] > nums[r] ? l : r;
    }
```

E3 的代碼(用它):

```
    public int findPeakElement(int[] nums) {
        if(nums == null || nums.length == 0) return -1;
        int n = nums.length;
        int l = 0, r = n - 1;

        if(n == 1) return 0;
        if(n == 2) return nums[0] > nums[1] ? 0 : 1;

        while(l + 1< r) {
            int m = (l + r) / 2;
            if(nums[m - 1] > nums[m]) r = m;
            else if(nums[m + 1] > nums[m]) l = m;
            else return m; //即 nums[m]比它左右都大
        }

        return nums[l] > nums[r] ? l : r;
    }
```

163. Missing Ranges, Medium, Locked

http://wlcoding.blogspot.com/2015/03/intervals-find-missing-insert-merge.html?view=sidebar

題目(我: `array` 和 `lower, upper` 都是輸入):
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"].

Key:

E3 方法: 寫一個 getRange(from, to)函數返回 from 到 to 的 inclusive range 的字符串(注意主函數輸出是 List<String>), 分 from 等於和不等於 to 兩種情況. 掃描 A 數組, 若出現不連續的元素(A[i] != A[i + 1] - 1), 則通過 getRange(A[i] + 1, A[i + 1] – 1)得到一個 missing range 並加入到結果中. lower 和 upper 要單獨處理(不難處理).

William 方法: 寫一個並調用 getRange(from, to)函數返回 from 到 to 的 inclusive range 的字符串(注意主函數輸出是 List<String>), 分 from 等於和不等於 to 兩種情況. 主函數中, 用 pre 和 cur 兩個指針(它們是數組元素, 不是 index), cur 在輸入數組中掃描, pre 是 cur 的前一個. 若 cur – pre >= 2, 則調用 getRange(pre + 1, cur – 1). 注意邊界情況.

Corner case: 要將 lower 和 upper 也包括进去; 輸入數組只有一個元素的時候.

History:

E1 直接看的答案.

E2 在我電腦上能得出正確結果, 算通過, 但代碼在處理兩端邊界時比 William 的略囉嗦.

E3 沒多久就寫好, 本來可以一次通過的, 結果不小心將代碼中 A 處的 else if 寫成了 if, 改後即通過(對題目中例子能給出正確結果). E3 的方法跟 William 差不多, E3 代碼比 William 長, 因為 E3 對兩端(即 lower 和 upper)是單獨處理的, 而 William 是硬將其和其它 i 寫到一起, 我覺得沒必要, E3 的代碼更好懂好寫, 不易出錯, 且為了跟 228. Summary Ranges 一致, 所以以後還是用 E3 代碼. 雖然本題是 lock 的, 但由於 E3 代碼跟 William 代碼維一的區別就是如何處理 upper 和 lower, 且本題比較簡單, 所以 E3 代碼應該是對的.

Solution

It assumes that lower <= min and upper >= max.

Time ~ O(N), Space ~ O(1)
Add an interval 的条件：curr - prev >= 2
注意边界值：

•prev 的初始为 lower - 1；
•循环 i 从 0 到 N (N + 1)，当 i == N，curr = upper + 1。

William 代碼(不用):

```
// find missing ranges in start -> end
public List<String> findMissingRanges(int[] A, int lower, int upper) {
    List<String> ranges = new ArrayList<>();
    int prev = lower - 1;
    for (int i = 0; i <= A.length; i++) {
        int curr = (i == A.length) ? upper + 1 : A[i];
        if (curr - prev >= 2)   ranges.add(getRange(prev + 1, curr - 1));
        prev = curr;
    }
    return ranges;
}

private String getRange(int from, int to) {
         return  (from  ==  to)  ?  Integer.toString(from)  :  from  +  "->"  +
to;//Integer.toString(int)見 Java 書 p404
}
```

E3 的代碼(用它):

```
   public List<String> findMissingRanges(int[] A, int lower, int upper) {
      List<String> res = new ArrayList<>();
      if(A == null || A.length == 0 || lower > upper) return res;

      if(lower < A[0]) res.add(getRange(lower, A[0] - 1));

      for(int i = 0; i < A.length; i++) {
         if(i == A.length - 1) {
            if(A[i] < upper) res.add(getRange(A[i] + 1, upper));
         } else if(A[i] != A[i + 1] - 1) { //A 處
            res.add(getRange(A[i] + 1, A[i + 1] - 1));
         }
```

```
        }

        return res;
    }

    private String getRange(int l, int r) {
        if(l == r) return Integer.toString(l);
        else {
            StringBuilder sb = new StringBuilder();
            sb.append(l);
            sb.append("->");
            sb.append(r);
            return sb.toString();
        }
    }
```

164. Maximum Gap, Hard

http://wlcoding.blogspot.com/2015/03/maximum-gap.html?view=sidebar

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.
Try to solve it in linear time/space.
Return 0 if the array contains less than 2 elements.
You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.
Credits:
Special thanks to @porker2008 for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 題目的意思是找輸入數組被排序後的最大跳躍, 這裡的跳躍是指 '排序後的數組' 中相鄰兩個數的差. 比如 排序後的數組 為{1, 2, 5, 87, 90}, 則 相鄰兩個數的差們 為: 1, 3, 82, 3, 故本題最終結果(最大跳躍)為 82.

用 Bucket sort (不用去看 CLRS). Bucket sort 中的 bucket 的意思其實就是 histogram 中的 bin. 本題比 Bucket sort 要簡化些, 因為不用 sort. 本題的思想是: 每個 bucket 中只需存入該 bucket 中所有實際元素的最大值和最小值. In terms of bucket 中所有實際元素的最大值和最小值, 為了方便看, 我將這樣一個 bucket: [min_of_bucket_i, max_of_bucket_i], 簡記為[min_1, max_1]. 如此, 則所有 buckets 為:
[min_1, max_1], [min_2, max_2], [min_3, max_3], [min_4, max_4], …
相鄰 buckets 之間有 gap, 比如 bucket_2 和 bucket_3 之間的 gap 為 min_3 – max_2. 本題的最終結果即為所有 gap 中最大的. 本題之結果之所以是所有 gap 中最大的, 是因為本題之結果不可能為同一 bucket 中某兩數之差, 後面有證明, 我只要記住這個結論即可.

實際中, 一些細節:
1. 總共有 n-1 個 buckets, 其中 n 為輸入數組的元素個數. 記住即可. 記憶: n 個數之間的 gap 有 n-1 個.
2. 每個 bucket 的 size 為 ceil((max - min) / (n – 1)) (後面有證明, 我只要記住此公式即可. 另 Math.ceil(125.9)=126.0), 其中 max 和 min 是輸入數組中的最大值和最小值. size 是甚麼意思? 意思就是, 我們有以下 buckets(written in terms of their boudaries):
[min, min + size – 1], [min + size, min + 2 * size – 1]...
若輸入數組中某數 a 滿足 min + size <= a <= min + 2 * size – 1 (都是<=是因為相鄰 bucket 之間沒

有挨著), 則將 a 放入[min + size, min + 2 * size – 1]這個 bucket 中. 注意這裡的 buckets 是 written in terms of their boudaries, 而不是像前面那樣 in terms of bucket 中所有實際元素的最大值和最小值.
3. Buckets 可用兩個數組來表示: int[] maxBucket 和 int[] minBucket. 這是 in terms of bucket 中所有實際元素的最大值和最小值. 第 i 個 bucket 即為[minBucket[i], maxBucket[i]], in terms of bucket 中所有實際元素的最大值和最小值.
4. 本題要是記住了以上方法, 其實是很好寫的.

History:
E1 直接看的答案.
E2 沒做出來, 主要是因為 E1 的 key 沒寫清楚, 看不懂是甚麼意思. 以上 key 基本都是 E2 重寫的.
E3 全忘了, 連用 bucket sort 這點都忘了, 故直接看的答案.

我: 其它人的代碼也都是在排序. 有兩個網頁都說桶排序是 leetcode 的官方解法, 所以我要掌握的是下面第一種方法, 有時間的話也可以看看第二種方法(今天沒看). 網上還有個人說(記住這個結論): 那么，线性的排序算法有哪些？计数排序、基数排序、桶排序。

leetcode 上给出的参考解法(的說明)(講得好)：

Suppose there are N elements and they range from A to B (我: inclusive).

Then the maximum gap will be no smaller than ceiling[(B - A) / (N - 1)] (我: ceil(125.9) = 126.0. 此結果的推導: 我們要找的是 maximum gap 的最小可能值, 這只有當所以元素盡量均勻分佈(間隔都相等)時會出現最小可能的 maximum gap(因為若不均勻份佈, 比如某個元素往右偏了一點, maximum gap 就等於那個元素左邊的那個 gap, 它比其它 gap 大, 即該元素的偏離使得 maximum gap 變大了). 實際上完全均勻是不一定能做到的, 當(B-A)/(N-1)能除盡時, 就能完全均勻, 此時最小可能的 maximum gap 就等於(B-A)/(N-1). 當(B-A)/(N-1)不能除盡時, 比如(B-A)/(N-1)餘 1 時, 那麼最好的分佈是這樣的: 先把前 N-2 個 gap 設為能除盡的(B-A-1)/(N-1), 最後一個 gap 比前面的 gap 大 1, 此時最小可能的 maximum gap 就是最後那個 gap, 它等於(B-A-1)/(N-1) + 1, 這實際上就等於 ceiling[(B - A) / (N – 1)] (think of ceiling(4/3)))

Let the length of a bucket to be len = ceiling[(B - A) / (N - 1)], (我: 後面那半句話(即本黃字後面的)沒用且錯, 不要看. 代碼中直接分配 N-1 個 bucket 就可以了. 可以看出, 如果 len 為那麼多, 則需要的 bucket 數不會超過 N-1 個, 因為若(B - A) / (N – 1)能除盡, 則需要(B-A)/len = N-1 個 bucket. 若不能除盡, 則需要得更少) then we will have at most num = (B - A) / len + 1 of bucket

for any number K in the array, we can easily find out which bucket it belongs by calculating loc = (K - A) / len (我: loc 為 int) and therefore maintain the maximum and minimum elements in each bucket.

Since the maximum difference between elements in the same buckets will be at most len - 1, so the final answer will not be taken from two elements in the same buckets.

For each non-empty buckets p, find the next non-empty buckets q, then q.min - p.max could be the potential answer to the question. Return the maximum of all those values.

William:
1. Bucket Sort: Time ~ O(2N), Space ~ O(2N)
Bucket Sort 一般适用于 uniform distribution，虽然这道题不能保证 uniform，但是由于是找 max Gap，不需要 sort 每个 bucket 里的元素，只要记录每个 bucket 的范围 [min, max] 即可，所以仍能保证 linear time。
方法：
•Construct n - 1 buckets, where the bucket size = ceil((max - min) / (n – 1))

Bucket 0: [min, min + size – 1] (我: min 和 min+size-1 是這個 bucket(即 bin)的左右限, 即在 min 和 min+size+1 之間的數(或等於 min 或 min+size-1 的數)會被於入這個 bucket 中)
Bucket 1: [min + size, min + 2 * size - 1]
...
Bucket n - 2: [min + (n - 2) * size, max]
•Put n - 2 numbers (except min and max) into the n - 1 buckets, and update each bucket's range [minBucket, maxBucket].
We know that there must be at least one bucket is empty (我: 因為要把 n-2 個數於入 n-1 個 bucket 中). That means the max gap is on the two sides of empty buckets(此句表述不清, 可以不看, leetcode 的管方講解對這點說得清楚些).
•Go through the n - 1 buckets and find the largest gap between two non-empty buckets.

第一種方法(用它):

```
public int maximumGap(int[] num) {
    int n = num.length;
    if (n < 2) return 0;

    // find the max and min
    int max = Integer.MIN_VALUE, min = Integer.MAX_VALUE;
    for (int i : num) {
        max = Math.max(max, i);
        min = Math.min(min, i);
    }
    if (max == min) return 0; //别忘了這句

    // create n - 1 buckets with their own min and max values
    // there must be at least one is empty
    int numBucket = n - 1;
    int sizeBucket = (int) Math.ceil((double) (max - min) / numBucket); //Math.ceil 見 Java 書 p144, 也可以不看 Java 書, 直接看此例: Math.ceil(125.9)=126.0
    int[] maxBucket = new int[numBucket];
    int[] minBucket = new int[numBucket];
    Arrays.fill(maxBucket, Integer.MIN_VALUE);
    Arrays.fill(minBucket, Integer.MAX_VALUE);
    // put n - 2 numbers (except max and min) into n - 1 buckets
    // by updating the corresponding bucket's min and max value
    for (int i = 0; i < n; i++) {
        if (num[i] != max && num[i] != min) { //此 if 的意思是 max 和 min 不放. 為何不放? 因為後面會專門處理它們兩個.
            int index = (num[i] - min) / sizeBucket;//想一想如何決定將一個數放到 histogram 的哪個 bin 中, 就明白了
            maxBucket[index] = Math.max(maxBucket[index], num[i]);
            minBucket[index] = Math.min(minBucket[index], num[i]);
        }
    }
    // find the max gap
    int maxGap = Integer.MIN_VALUE, prev = min;//由於 min 沒有放入 bucket 中, 所以 prev 要從 min 開始. prev 的意思是當前 bucket 之前的那個 bucket 的 maxBucket.
    for (int i = 0; i < numBucket; i++) {
        if (maxBucket[i] != Integer.MIN_VALUE && maxBucket[i] != Integer.MAX_VALUE) {//此 if 的意思是 要 bucket 不為空. 實踐證明, 將&&右邊的刪掉 也能通過, 實際上刪掉更 make sense.
            maxGap = Math.max(maxGap, minBucket[i] - prev);
            prev = maxBucket[i];
        }
    }
```

```
    maxGap = Math.max(maxGap, max - prev);//由於 max 沒有於入 bucket 中, 所以要單獨考慮
    return maxGap;
}
```

2. LSD (我: LSD 意思下面會說) Radix sort: Time ~ O(MN) where M << N (still linear), Space ~ O(N)
这里不适合直接用 counting sort，因为其 time ~ O(N + R)，R 为 num 的范围，可能 R >> N，导致很慢。
我们选用 least-significant-digit-first (LSD) radix sort，其算法对每一位进行 counting sort，这里的 R = 10（0 ~ 9)，从而大大提高了速度。从低位到高位，循环次数 M = max number of digits（最大位数）。
LSD radix sort 是 distribution-based sorting algorithm，不需要任何 compare，这道题经测试比 Bucket sort 要快。
注意：

•取 integer 的位数可以用 log：Math.log10(val) + 1，也可用 Integer.toString(val).length() 或 String.valueOf(val).length()，用 log 更高效。
•Counting sort 中 1) count size 是 R + 1（因为是 CDF，要多留一维）；2) compute frequency 的一步要注意 count[a[i] + 1]++ 的 index 里要加 1；3) move item 的一步 index 要 ++（这样之后的相同元素会往后放）。

第二種方法(不用):
```
public int maximumGap(int[] num) {
    int n = num.length;
    if (n < 2)  return 0;

    // find the max
    int max = Integer.MIN_VALUE;
    for (int i : num)
        max = Math.max(max, i);
    int m = (int) (Math.log10(max) + 1); // number of digits

    int div = 1; // divider
    int R = 10;
    int[] aux = new int[n];
    for (int d = m - 1; d >= 0; d--) {
        int[] count = new int[R + 1];
        // compute frequency of each digit using key as index
        for (int i = 0; i < n; i++)
            count[(num[i] / div) % 10 + 1]++;
        // compute frequency cumulates which specifies the destinations
        for (int r = 0; r < R; r++)
            count[r + 1] += count[r];
        // access cumulates using key as index to move items
        for (int i = 0; i < n; i++)
            aux[count[(num[i] / div) % 10]++] = num[i];
        // copy back into original array
        for (int i = 0; i < n; i++)
            num[i] = aux[i];
        // update divider to move to higher digit
        div *= 10;
    }

    // find the max gap
```

```
    int maxGap = Integer.MIN_VALUE;
    for (int i = 1; i < n; i++) {
        maxGap = Math.max(maxGap, num[i] - num[i - 1]);
    }
    return maxGap;
}
```

165. Compare Version Numbers, Easy

http://wlcoding.blogspot.com/2015/03/compare-version-numbers.html?view=sidebar

Compare two version numbers *version1* and *version2*.
If *version1* > *version2* return 1, if *version1* < *version2* return -1, otherwise return 0.
You may assume that the version strings are non-empty and contain only digits and the . character.
The . character does not represent a decimal point and is used to separate number sequences.
For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.
Here is an example of version numbers ordering:
0.1 < 1.1 < 1.2 < 13.37
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 對兩個 version number 用 .split("\\.") 分割的到兩個 String[]，然后取长的那个 String[]的长度(作為循環條件, 短的 String[]补 0)，比较每一段，相同段位大的大，若相等进入下一段比较

Convention: 輸入的 version 有可能為"1", "1.0", "0.0.1"

History:
E1 的代碼也通過了, 但沒有 William 的簡潔, 方法跟 William 差不多. 以下用 William 的代碼.
E2 很快寫好, 本來可以一次通過, 但忘了加一個負號(William 的代碼中不需要), 方法跟 William 差不多, 添了後即通過.
E3 最開始不知道輸入會有"1", "1.0", "0.0.1"這些情況, 所以改了很多次, 最終才弄清題意(題目不說清楚, 這應該就是為何本題為 Easy, 但通過率只有 17.9%). E3 的方法跟 William 的一樣, 都是取長的那個 String[]的長度作為循環條件, 然後給短的补 0. 但俱體寫法上稍有不同, 最後參考了 William 的代碼將我 E3 的代碼改進了下, 比 William 的稍微好懂好寫, 以後用我 E3 改進後的代碼.

來自網路: 题目意思是说给定两个版本号。比如 1.4.2 表示第一个版本的第四个版本的第二个版本，另外一个版本号比如 1.3.7 第一个版本的第三个版本的第七个版本。问你哪个比较新。很明显 1.4.2 比较新返回 1 如果一样新返回 0 比较旧返回-1. 注: 记得判断情况 1.4.0000 和 1.4 是一样的！

William:

Time ~ O(max{Na, Nb}), Space ~ O(Na + Nb)
用 .split("\\.") 分割的到 char[](我: 應該是 String[])，然后取长的那个的长度(作為循環條件, 短的 String[]补 0)，比较每一段，相同段位大的大，若相等进入下一段比较。

```
public int compareVersion(String version1, String version2) {
    String[] v1 = version1.split("\\.");    // "." means any character //注意用\\.
```

見 Java 書 p1301

```
    String[] v2 = version2.split("\\.");

    int longest = v1.length > v2.length ? v1.length : v2.length;

    for (int i = 0; i < longest; i++) {
        //注意以下的补 0 技巧, 而且可以不用弄清楚到底 v1 長還是 v2 長
        int ver1 = i < v1.length ? Integer.parseInt(v1[i]) : 0;
        int ver2 = i < v2.length ? Integer.parseInt(v2[i]) : 0;

        if (ver1 > ver2)    return 1;
        if (ver1 < ver2)    return -1;
    }
    return 0;
}
```

E3 改進後的代碼:

```
public static int compareVersion(String version1, String version2) {
    String[] v1 = version1.split("\\.");
    String[] v2 = version2.split("\\.");

    if(v1.length < v2.length) return -compareVersion(version2, version1); //注意負號

    //Now v1.length >= v2.length
    for(int i = 0; i < v1.length; i++) {
        int v1i = Integer.parseInt(v1[i]);
        int v2i = i < v2.length ? Integer.parseInt(v2[i]) : 0;
        if(v1i > v2i) return 1;
        else if(v1i < v2i) return -1;
    }

    return 0;
}
```

166. Fraction to Recurring Decimal, Medium

http://wlcoding.blogspot.com/2015/03/fraction-to-recurring-decimal.html?view=sidebar

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:

1. No scary math, just apply elementary math knowledge. Still remember how to perform a *long division*?
2. Try a long division on 4/9, the repeating part is obvious. Now try 4/333. Do you see a pattern?
3. Be wary of edge cases! List out as many test cases as you can think of and test your code

thoroughly.

Credits:
Special thanks to @Shangrila for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 注意本題雖然為 Medium, 但通過率也只有 16.0%, 在 Hard 中也算低的.
本題代碼中可以用除法(a/b)運算, 只是 a 和 b 都是 int, (a/b)也是 int, 而我們要做的就是算出小數結果. 先在結果中加入可能的負號, 然後將 numerator 和 denominator 取絕對值來算. 直接按手算除法的方法寫即可, 寫起來其實一點都不難, 用 4/333 作例子即知, 就是將每次得到的 remainder 乘以 10 再當作 numerator 來除. 當 remainder 出現重復時, 即表明為循環小數. 用 Map<remainder, position> 來記錄 remainder, 以及此 remainder 相應的商在結果 String 中的位置(map.put(remainder, res.length())), 這樣好知道在哪裡放入"(". 最後再放入")"即可.

注意 numerator 和 denominator 都可能為 Integer.MIN_VALUE, 最開始取絕對值時會越界, 所以 numerator, denominator, 以及 remainder 都要弄成 long (remainder 為何也要弄成 long, 見 History 中的 E2).

Conventions:
5/0 返回"2147483647",
-5/0 返回"-2147483648".
400/333=1.0120120120...=1.(012)

注意小數點後不一定就馬上開始循環, 比如 73/15=4.86666...=4.8(6)

History:
E1 直接看的答案.

E2 范了兩個小錯: 一是得到 remainder 後, 本應該用 remainder*10 再除, 我寫成了用 numerator*10 再除. 二是不知道 long 類型也可以放在 Map 中(即 Map<Long, Integer>), 而將 remainder 定義成了 int, 放在 Map<Integer, Integer> 中, 然後報僥倖心理, 以為 remainder 不會越界, 給果當 (-1) / (-2147483648)時, remainder 作為 int 還是越界了. 把以上兩個錯誤改了後, 即通過. 注意基本上所有 primitive type 都有相應的 wrapper class (即 Integer 等, Java 書 p403). 以上的 key 是 E2 寫的.

E3 改了很多遍才通過, E3 代碼比 William 的長不少, 主要是因為 E3 不是用的 map.put(remainder, res.length()), 而是 map.put(remainder, pos), 而 pos 是通過較複雜的方法算出來的, 這反而使得要考慮一些特殊情況.

Time ~ O(N), Space ~ O(N)
1) First add sign, integer part, and digit;
2) Then add the fractional part:
循环小数开始的条件：当 remainder 出现重复，从上一个 reminder 出现的位数到当前为循环的部分。
用 Hash Table 存放 <remainder, string index>，以便找出开始循环的位置插入 "("。
注意：这里 numerator 或 denominator 可能为 Integer.MIN_VALUE，取 Math.abs 会 overflow(我: 因為 Integer.MIN_VALUE 的絕對值比 Integer.MAX_VALUE 大 1, 詳細理由見下)，所以要转换成 long。

我:
為何不能取 Math.abs(Integer.MIN_VALUE):

網上說: `Math.abs(int)` should behave like the following Java code:

```
public static int abs(int x){
    if (x >= 0) {
        return x;
    }
    return -x; //注意此處返回了個 -Integer.MIN_VALUE, 而返回值要求是 int 類型的, 所以 overflow 了
}
```

而以下代碼中的 long n = numerator, n = Math.abs(n), 應該是調用了 Math.abs 的 overload 版本, 即 long abs (long x), 所以不會出現以上問題.

```
public String fractionToDecimal(int numerator, int denominator) {
    if (numerator == 0) return "0";

    StringBuilder str = new StringBuilder();

    // add sign
    if (numerator < 0 ^ denominator < 0)    str.append("-"); //注意用^的技巧

    long n = numerator, d = denominator; // use long to avoid overflow of -Integer.MIN_VALUE
    n = Math.abs(n);    // Math.abs can't be directly used on Integer.MIN_VALUE
    d = Math.abs(d);
//注意以上三句不能寫為 long n = Math.abs(numerator), d = Math.abs(denominator);否則換湯不換藥, 因為 abs 中還是 int 類型的 numerator

    // add integer part
    str.append(n / d); //可以直接 append(int 等 type), 詳見 Java 書 p416

    // add digit or stop
      long r = n % d; // remainder, must use long type. 因為若 n = 1, d = Math.abs(Integer.MIN_VALUE)時, r 也可以為 Math.abs(Integer.MIN_VALUE)
    if (r == 0) return str.toString();
    else        str.append(".");

    // add fractional part
     Map<Long, Integer> map = new HashMap<>(); // <remainder, position>//注意 Long 大寫
    while (r > 0) {
        if (map.containsKey(r)) {
                str.insert(map.get(r), "(");//先看 else 中的, 再看這裡. insert 即在 map.get(r)處插入, 詳見 Java 書 p416. 比如 else 中的 4/3, 此時 str 已經是"1.3"了. 此 insert 執行後, str 就是"1.(3"
            str.append(")");//此句執行後, str 就是"1.(3)"
            break;
        } else {
              map.put(r, str.length());  // no need to -1 on str.length() since r hasn't been added to str//比如說 4/3, 此時 str 已經是"1."了,此時就是 map.put(3, 2) ← 將(3, 2)放入 map 中, 是為了後面發現 3 為循環數(重復數時)好在適當的位置加括號.
            r *= 10;//想一想手算 4/3 的過程就明白了
            str.append(r / d);//此時 str 已經是"1.3"了, 下一輪就要在 3 前後加括號了
            r = r % d;
        }
    }
```

```
    return str.toString();
}
```

167. Two Sum II - Input array is sorted, Medium, Locked

http://wlcoding.blogspot.com/2015/03/n-sum-2sum-i-ii-iii-3sum-i-ii-4sum-i.html?view=sidebar

Given an array of integers that is already ***sorted in ascending order***, find two numbers such that they add up to a specific target number.
The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.
You may assume that each input would have exactly one solution.

Key: 簡單. Two pointers, 两头夹逼，大于 target 则 hi--(為何不 lo++?見代碼中)，小于 target 则 lo++. 别證.

History:
E1 時, 我的方法跟 William 是一樣的, 在我電腦上可以得出正確結果, 但 leetcode 上是 locked 所以沒法測試, 可以算我的代碼已通過.
E2 可以算通過.
E3 很快寫好, 但第一次在我電腦上測試時进入了死循環, 原因是 sum = numbers[lo] + numbers[hi]時, 忘了 return, 改後即通過(即在我電腦上能對好幾種 test case 給出正確答案). E3 表示 E1 居然也能想到 key 中方法, NB.

我: 以下用 William 的代碼.

Solution
Two pointers: Time ~ O(N), Space ~ O(1)
两头夹逼，大于 target 则 hi--，小于 target 则 lo++。(我: 此算法沒時間證明, 記住就可以了)

```
public int[] twoSum(int[] numbers, int target) {
    // Assume input array is sorted
    int N = numbers.length, lo = 0, hi = N - 1;
    while (lo < hi) {
        int sum = numbers[lo] + numbers[hi];
        if (sum > target)       hi--;//為何不 lo--? 因為兩個 pointer 是從兩端向中間移動的. lo--就退了回去, where has already been visited earlier.
        else if (sum < target)  lo++;
        else                    return new int[] {lo + 1, hi + 1}; //注意此句寫法, 注意不能直接 return {lo + 1, hi + 1}
    }
     throw new IllegalArgumentException("No two sum solution"); //添了此句後就不用加 return 語句了, 否則 compiler 說沒有 return
}
```

168. Excel Sheet Column Title, Easy

http://wlcoding.blogspot.com/2015/03/excel-sheet-column-title-number.html?view=sidebar

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

1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB

Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 看一看 BAC = 2*26^2 + 1*26 + 3 = 1381 就知道怎麼寫了. 設輸入的數叫 n, 則每次 n % 26 即可得到一位數(如 C), 將該位數加到結果中, 然後從 n 中減去該數(C, 即 3), 再除以 26, 即可重復以上步驟, 得到更多的位數.

注意要記住每輪循環都要先 n--. 這樣做的原因是: 本題跟一般的 26 进製還有點不同, 一般的 26 进製是
A ↔ 0, B ↔ 1, C ↔ 2, ...Z ↔ 25,
而本題是
A ↔ 1, B ↔ 2, C ↔ 3, ...Z ↔ 26,
即最大可取到 26.

n--後就可以用 A ↔ 0, B ↔ 1, C ↔ 2, ...Z ↔ 25 的對應關係. 為何一定要用這個對應關係? 因為如下例:

若 n 不減 1, 則
AZC= 1*26^2 + 26*26 + 3 = 1*26^2 + 26^2 + 3 = 2*26^2 + 3, 就會得出錯誤的結果.

若每輪 n 減 1, 則該數減 1 後變為
1*26^2 + 26*26 + 2, 由 A ↔ 0, B ↔ 1, C ↔ 2, ...Z ↔ 25 知, 最右一位為 C
然後除以 26, 得
1*26 + 26 = 2*26, 會得出錯誤的結果, 但將其減 1 後為:
1*26 + 25, 由 A ↔ 0, B ↔ 1, C ↔ 2, ...Z ↔ 25 知, 右起第二位為 Z, 就得出正確結果了.

History:
E1 的代碼也能通過, 但比 William 的囉嗦點, 原因是沒有想到 William 那個先做 n--的技巧, 所以弄得我 n %26 == 0 時要單獨考慮. 以下用 William 的代碼.
E2 幾分鍾寫好, 一次通過.
E3 改了好多次才通過, 原因是 E3 只知道要減 1, 但忘了在哪裡減 1, 後來仔細想了, 才分析出來該在哪裡減 1.

來自網路: 这道题是我微软 onsite 时遇到的一道题，没做过遇到这道题确实有点难一下子理得很清楚（我当时这道题应该做的不好，从 most significant digit 做，而且忘了 n 要-1）。这道题说白了其实就是十进制转换 26 进制，而且是从 1 开始的 1 十进制的转换

以下例子可以幫助我想:

A=1, B=2, C=3, D=4, E=5...

BAC = 2*26^2 + 1*26 + 3 = 1381
AEBZ = 1*26^3 + 5*26^2 + 2*26 + 26 = 21034

William:
Time ~ O(N), Space ~ O(N)
从低位到高位生成 Column Title，最后反转整个 String。

```
public String convertToTitle(int n) {
    StringBuilder str = new StringBuilder();
    while (n > 0) {
        n--; // since 'A' corresponds to 1 not 0 //注意這裡先做 n--的技巧. 否則對於上面 AEBZ 的情況, 可以正確得到 Z, 但在 n/26 後进入下一輪, 此時 n= 1*26^2 + 5*26 + 2 + 1 = 1*26^2 + 5*26 + 3, 得到的是 C, 而不是 B!
        str.append((char) (n % 26 + 'A'));
        n = n / 26;
    }
    return str.reverse().toString();
}
```

169. Majority Element, Easy

http://wlcoding.blogspot.com/2015/03/majority-element.html?view=sidebar

Given an array of size *n*, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times.
You may assume that the array is non-empty and the majority element always exist in the array.
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: Moore's voting algorithm. 遍歷數組, 用 count 記錄當前 major 出現的次數減去被別人 cancel 的次數, 若 count 減為 0 了, 就將 major 設為 num[i], 且 count 設為 1. 記住, 別證.

History:
E1 幾分鍾寫好並一次通過, 用的 Hash Table.
E2 幾分鍾寫好並一次通過, 用的上面 key 中的方法.
E3 忘了還有 cancel 的方法, 故先用的 Hash Table, 兩分鍾寫好, 一次通過. 然後瞟到了 key 中說要用甚麼 cancel, 就回想起來了, 然後按 cancel 那樣寫, 改了幾次後通過, 是試出來的, 原理不清楚. E3 又將答案代碼默寫了一遍, 過了一個兒才寫的, 故寫出來的代碼跟答案不完全一樣, 比答案的簡短, 也帖在後面, 可參考.

我的代碼幾分鍾寫好並一次通過, 用的 Hash Table, 記錄每個數出現的個數. 網上說了, Hash Table 的方法時間為 O(n), 空間也為 O(n). 網上還說了幾種其它方法, 比如排序. 以下用 William 的代碼. 由於我對 Moore's voting algorithm 理解不深, 所以最好也熟悉一下我的 Hash Table 做法(代碼在最後).

Moore's voting algorithm: Time ~ O(N), Space ~ O(1)
A straightforward way is to use a Hashtable, and then iterate through it (map.entrySet()) to find the majority element. However it requires extra space.
Instead, we can do it in-place with one-pass.
Idea: Since the majority is more than ⌊ n/2 ⌋ times, it can cancel out all the other elements,

and the remaining one will be the majority element. (我: 理由沒這麼簡單, 文獻中的證明有兩大段長, 沒時間看. 記住這個算法就可以了. 若面試中問到了為甚麼這個算法是對的, 就按 cancel 這麼答)
major: stores the current candidate of the majority element;
count: (我: count 是記錄當前 major 出現的次數減去被別人 cancel 的次數)three cases:

•if (count == 0), set current candidate to num[i] and set counter to 1;
•if num[i] == major, count++;
•if num[i] != major, count--.

```
public int majorityElement(int[] num) {
    // Moore's voting algorithm
    int major = num[0], count = 1;
    for (int i = 1; i < num.length; i++) {//注意是從 1 開始
        if (count == 0) {
            major = num[i];
            count = 1;
        } else if (num[i] == major) {
            count++;
        } else
            count--;
    }

    return major; // no need to verify since it assumes majority always exists
}
```

E3 的 cancel 方法代碼(不用, 可參考):

```
public int majorityElement(int[] nums) {
    int count = 0;
    int major = nums[0];

    for(int i = 0; i < nums.length; i++) {
        if(count == 0) major = nums[i];
        if(major == nums[i]) count++;
        else count--;
    }

    return major;
}
```

170. Two Sum III - Data structure design, Easy, Locked

http://wlcoding.blogspot.com/2015/03/n-sum-2sum-i-ii-iii-3sum-i-ii-4sum-i.html?view=sidebar

Design and implement a TwoSum class. It should support the following operations: add and find.
Add - Add the number to an internal data structure.
Find - Find if there exists any pair of numbers which sum is equal to the value.
For example,
add(1); add(3); add(5);
find(4) -> true
find(7) -> false

Key: 注意輸入數組中的元素可能有重復的. 用一個 HashMap, key 存放數組中的數, value 存放該數在數組中出現的次數. 對 HashMap 用 foreach loop, 用 map.entrySet(), 見 Java 書 p834 和 p835.

History:
E1 直接看的答案.
E2 的方法略有小差別, 代碼在我電腦上的各種 case 可以通過, 由於是 locked, 沒法在 OJ 上測試, 故也算通過.
E3 幾分鍾寫好, 最開始對 '只 add 了一個 target/2 之情況' 給出 true 這個錯誤答案, 改後即通過(在我電腦上對各種 test case 能給出正確答案, 且代碼邏輯跟以下答案基本一樣).

Solution

Hash Table: Time: add ~ O(1), find ~ O(N), Space ~ O(N)
用一个 Hash Table，key 存放 number，value 存放出现次数 count。
add()：update Hash Table 中相应的 count；
find()：遍历 Hash Table，对于每一个 num，有两种情况：

•num == value - num: check if count >= 2;
•num != value - num: check if containsKey(value - num).

```
public class TwoSum {
    private Map<Integer, Integer> map = new HashMap<>();

    public void add(int number) {
        //以下兩句是在 map 中放東西的簡潔寫法
        int count = map.containsKey(number) ? map.get(number) : 0;
        map.put(number, count + 1);//由此可知, 無論 Map 中是無已有 key, 都可以用 put. 已記入 Java 書中.
    }

    public boolean find(int value) {
        for (Map.Entry<Integer, Integer> entry : map.entrySet()) { //由此可知 Foreach loop 也可用於 HashMap(已記入 Java 書中). Map.Entry 神馬的見 Java 書 p834 和 p835. 注意此處的 entry 就是一個 Set 中的一個元素, 它本身又是一個 Map 的一個元素.
            int num = entry.getKey();//entry 作為一個 Map 的一個元素. getKey()見 Java 書 p835
            int y = value - num;
            if (y == num) {
                if (entry.getValue() >= 2)  return true; //getValue()見 Java 書 p835
            }
            else if (map.containsKey(y))
                return true;
        }
        return false;
    }
}
```

171. Excel Sheet Column Number, Easy

Related to question Excel Sheet Column Title
Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28

Credits:
Special thanks to @tsfor adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 簡單, 直接寫. 26 进製. 它跟 10 进製的轉換關係是我熟知的 pattern: 比如 BAC = 2*26^2 + 1*26^1 + 3*26^0 = 1381. 記住此公式即可.

History:
E1 的代碼很快寫好, 一次通過. 我的代碼跟 William 的基本一樣(有括號的差別, 但我覺得我的更好). 以下用我 E1 的代碼. William 說的 Time ~ O(N), Space ~ O(1), 顯然我的也一樣.
E2 幾分鍾寫好, 一次通過.
E3 兩 分鍾寫好, 一次通過.

題目意思為(來自網路, 本來是 168 題的題意, 但由於先做本題比較好, 所以 copy 到本題來了):
excel 中的序是这样排的：A~Z，AA~ZZ，AAA~ZZZ.....
本质是进制转换，将 n 转化为 26 进制，转化过程如下（括号里的是 26 进制数）：
1->(1)->A
2->(2)->B
...
26->(10)->Z
27->(11)->AA
28->(12)->AB
.....
52->(20)->AZ
53->(21)->BA

公式為: BAC = (2*26+1)*26+3 = 1381

```
 public static int titleToNumber(String s) {
  int res = 0;

  for(int i = 0; i < s.length(); i++)
   res = res * 26 + (int) (s.charAt(i) - 'A') + 1;

  return res;
 }
```

172. Factorial Trailing Zeroes, Easy

http://wlcoding.blogspot.com/2015/03/factorial-trailing-zeroes.html?view=sidebar

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

Note: Your solution should be in logarithmic time complexity.
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 注意题目的 n!表示階乘的意思，而不是感嘆號. 題目的意思是找 n!的右起連續 0 的個數，如若 n!=23050078000, 則 Trailing Zeroes 為 3 個.

易知 n! 的 trailing zeroes 的個數即為 [1,...n]范圍內所有數的 5 因子個數(馬上說理由), 例如若 n=17, 由下面的數表知, [1,...17]范圍內的 5, 10, 15 總共貢獻三個 5 因子, 故 17! 的 trailing zeroes 為三個. trailing zeroes 的個數為何等於 5 因子個數? 因為一個 5 因子和一個 2 因子就會生成一個 trailing zero, 而 2 因子總是比 5 因子多, 故只數 5 因子個數就可以了. 當 n 小於 25 時, n!的 trailing zero(即 5 因子)個數為 n/5 個.

當 n 大於 25 時, 比如 n=27 時, n! 的 trailing zero(即 5 因子)個數為多少? 由下面的數表知, 5, 10, 15, 20, 25 都會貢獻 5 因子, 但 25 貢獻的是兩個 5 因子, 故總共的 5 因子數為 6 個. 同理可知, 50 也要貢獻兩個 5 因子, 125 要貢獻三個 5 因子.

故實際的方法為: trailing zero(即 5 因子)個數 = 貢獻一個 5 因子的數 的個數 + 貢獻兩個 5 因子的數 的個數 + 貢獻三個 5 因子的數 的個數 = n / 5 + n / 25 + n / 125 ...

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

History:
E1 的代碼是挨個數數 5 的個數, 最後累加, 結果超時了(題目要求 log(n)). 以下用 William 的代碼.
E2 幾分鍾寫好, 一次通過, 代碼跟 William 基乎一模一樣.
E3 兩分鍾寫好, 一次通過, 代碼跟 William 基乎一模一樣. E3 是想了一會兒才想到方法, 然後才開始寫的. E3 通過後, 發現之前留的 key 其實並不好懂, 所以 E3 又重寫了 key.

Time ~ O(log5N), Space ~ O(1)
每一个包含 5 的数都会生成一个 0 (因为 even number 的数量要大于 5 的数量, see more detail below)
5 生成 1 个 0
25 生成 2 个 0 (我: 因為包含兩個 5 因子, 每個 5 因子遇到一個 2 因子, 就日出一個 10)
125 生成 3 个 0 (我: 同上)
...
先计算所有包含 5 的个数，累加到 0 的个数中；
(我: 以下這句話先不看, 等看了公式再看, 自然就明白了)
注意 25, 125, ... 也包含了 5，所以在 25, 125 中不用算 2 个 0 或 3 个 0，即不用多加，所以之后每遇到一个 5 的倍数，也就累加一个 0。

```
public int trailingZeroes(int n) {
    // count how many 5's
    // # of zeros = floor(n/5) + floor(n/25) + floor(n/125) + ...(此句乃 William 方法之關鍵, 比如若 n=30, 則 floor(n/5)即以上藍色數字(即 5 的倍數)的個數, floor(n/25)即 25 的倍數的個數)
    int count = 0;
```

```
    while (n > 0) {
        n = n / 5; // use divide instead of multiplication to avoid overflow
        count += n;
    }
    return count;
}
```

173. Binary Search Tree Iterator, Medium

http://wlcoding.blogspot.com/2015/03/binary-search-tree-iterator.html?view=sidebar

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.
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 本題其實不難, 不要被複雜的題目和本 key 嚇到了. 中序遍歷. 用 stack. 寫一個 pushLeftChildren(curr), 作用為將「以 curr 為根的子樹」的最左邊 edge 的所有節點(包括 curr)放入 stack 中, 最後放入的那個節點就是「以 curr 為根的子樹」的最小值, ready for use. 當調用下面的 next()函數時, pop 掉這個節點(即最小值節點, 即 res), 返回這個最小值, 並 pushLeftChildren(res.right). 注意 BST 的性質告訴我們, 右子樹中所有節點的值都大於根的值, 所以 pushLeftChildren(res.right)後, stack 的棧頂元素最比 res 大的下一個值(當然, 若 res.right 不存在, 則之前的 stack 中的下一個就是比 res 大的下一個值), ready for next use. 在 constructor 中調用 pushLeftChildren(root), 作用就是在 stack 中放東西, 即初始化 stack. 注意題目的意思仍然是 在調用 next()之前, 要先調用 hasNext(), 不然要 hasNext()有卵用.

本節論可以記住(跟本題沒多大關係): 易證明, 一個 BST 最小值的那個節點為: 沿著該 BST 最左邊緣一直按 cur = cur.left 走到底的那個節點.

最後, 由本題可知, 給定 BST 中的一個節點, 要求比它大的下一個節點(即中序遍歷的下一個)不是那麼直接的, 要像本題那樣弄個 stack.

History:
E1 直接看的答案.
E2 本來可以一次通過的, 但 hasNext()中不小心少寫了一個!符號, 改後即通過.
E3 一次通過, 方法跟 William 的一樣. 最開始沒有頭緒, 後來看到題目中要求 時間 O(1) 和 空間 O(h), 就想到了用迭代法中序遍歷, 然後看了看之前的迭代法中序遍歷代碼, 就寫出來了.

我: William 給出了三種方法. 只有中序遍歷(即第一種方法)是對的. 另外兩種都是錯的, 不要看.

William:

Inorder: left-root-right

Solution

Time ~ O(1), Space ~ O(logN)

•constructor：找到最左节点，recursively call pushLeftChildren()。
•next()： 要先 check hasNext()；
然后从 Stack 中 pop 一个 node，将其右子树入栈，再返回该 node 的 val。
•hasNext()：Iterator 要注意 call hasNext() 不能影响 next() 的结果。
用一个 Stack，hasNext() 中只需 check 该 Stack 是否为空。

```
public class BSTIteratorInorder {//要將 BSTIteratorInorder 改為 BSTIterator, leetcode 才能通過, 此處為了與其它兩種方法一致, 故沒改
    private Stack<TreeNode> stack = new Stack<>();

       public  BSTIteratorInorder(TreeNode  root)  {// 要 將 BSTIteratorInorder 改 為 BSTIterator, leetcode 才能通過, 此處為了與其它兩種方法一致, 故沒改
          pushLeftChildren(root); // find the first node (leftmost) to start, and store the trace //此 constructor 的作用就是在 stack 中放東西, 即初始化 stack
    }

    // push all the left subnodes to stack until reaching the first node in inorder (the leftmost node)
//pushLeftChildren(curr)的作用為將「以 curr 為根的子樹」的最左邊 edge 的所有節點(包括 curr)放入 stack 中, 最後放入的那個節點就是「以 curr 為根的子樹」的最小值, ready for use. 當調用下面的 next() 函數時, pop 掉這個節點(即最小值節點, 即 res), 返回這個最小值, 並 pushLeftChildren(res.right). 注意 BST 的性質告訴我們, 右子樹中所有節點的值都大於根的值, 所以 pushLeftChildren(res.right)後, stack 的棧頂元素最比 res 大的下一個值(當然, 若 res.right 不存在, 則之前的 stack 中的下一個就是比 res 大的下一個值), ready for next use.
    private void pushLeftChildren(TreeNode curr) {
        while (curr != null) {
             stack.add(curr); //stack.add 作用跟 stack.push 一樣(網上已驗證), 將此句改為 stack.push(curr)也能通過
            curr = curr.left;
        }
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
           if (!hasNext()) throw new NoSuchElementException("All nodes have been visited");//E1: leetcode 不認識 NoSuchElementException 這個 exception, 將此 if 刪了後就能通過. E2: 我 E2 的代碼就沒有這個 if, 也通過了, 注意題目的意思仍然是 在調用 next()之前, 要先調用 hasNext(), 不然要 hasNext()有卵用.

        TreeNode res = stack.pop();
        pushLeftChildren(res.right);//當 res.right 為 null 時怎麼辦? 看 pushLeftChildren 函數裡面的 while(curr != null), 就知道此時甚麼都不做.
        return res.val;
    }
}
```

Preorder: root-left-right

Solution

Solution 已省之. 因為代碼是錯的(可以明顯從調用 stack.add 的順序中看出來), leetcode 給出的結果為:

Submission Result: Wrong Answer. Input: [2,1] . Output: [2,1]. Expected: [1,2]

不要浪費時間去看代碼!

Postorder: left-right-root

Solution

Solution 已省之. 因為代碼也是錯的(沒看代碼), leetcode 給出的結果為:

Submission Result: Wrong Answer. Input: [1,null,2]. Output: [2,1]. Expected: [1,2]

不要浪費時間去看代碼!

174. Dungeon Game, Hard

http://wlcoding.blogspot.com/2015/03/dungeon-game.html?view=sidebar

The demons had captured the princess (**P**) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (**K**) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.
The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.
Some of the rooms are guarded by demons, so the knight loses health (*negative* integers) upon entering these rooms; other rooms are either empty (*0's*) or contain magic orbs that increase the knight's health (*positive* integers).
In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.
For example, given the dungeon below, the initial health of the knight must be at least **7** if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

Notes:

- The knight's health has no upper bound.
- Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

Credits:
Special thanks to @stellari for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 動態歸劃. 看下面的的遞推式. 注意要倒著弄, 即從公主到武士掃描. 注意處理在邊界或公主住置的情況. 盡量把代碼寫成 1-d DP.

History:
E1 直接看的答案.

E2 直接寫的 1-d DP, 沒多久就寫好了, 本來可以一次通過的, 但初始化最後一行時少加了個 1, 改後即通過. William 的代碼中太多 if-else, 沒有我 E2 的代碼好懂, 以後都看 E2 的代碼.

E3 最開始按是順著(即從武士到公主)弄的, 寫出的代碼怎麼也通不過, 後來發現順著弄是不可能做出來的, 因為它缺乏之前的 health 信息. 後來改為倒著弄, 寫好後改了一個小錯誤(返回為 0 時, 要實際返回 1, William 代碼沒這個問題), 即通過. E3 代碼比 William 的長不少, 原因是 E3 沒寫出一個簡潔的遞推式, 而是用的一堆 if-else(語義上跟 William 差不多, 但有小不同, 應該跟 William 的等價, 沒證). 以後還是要記住遞推式. E3 是按 2d-DP 寫的.

dungeon: 地牢
我: William 給了 2-d 和 1-d 兩種 DP. 我要掌握的是 1-d DP. 但要先看看 2-d DP, 才好看懂 1-d DP.

William:
注意：1) 武士血不能为 0 2) 有些房间可以加血 (grid[i][j] > 0)

1. 2-d DP: Time ~ O(M*N), Space ~ O(M*N)
Let d(i, j) be the min health from **grid[i, j] to** grid[M - 1][N – 1]. //modified by Tao
d(i, j) = max{min{d(i + 1, j), d(i, j + 1)}- grid[i][j], 1 } (我: 以下是對此式的解釋)
•underline part: min health to proceed after taking possession of grid[i][j] (我: grid[i][j]即 dungeon[i][j]. 若 grid[i][j]為正, 則有 grid[i][j]那麼多是不需要的, 所以要減, 若 grid[i][j]為負, 可類似想)
(two cases: go down or go right)
•max{x, 1}: guarantee health >= 1

Return d(0, 0).
Use bottom-up approach.

```
public int calculateMinimumHP(int[][] dungeon) {
        int m = dungeon.length;
        int n = dungeon[0].length;

        int[][] d = new int[m][n];  // min health to enter dungeon(i, j)
        for (int i = m - 1; i >= 0; i--)
            for (int j = n - 1; j >= 0; j--) {//以下的幾個 else, 從最後的 else 往前看, 就明白了
                if (i == m - 1 && j == n - 1)
                    d[i][j] = Math.max(1 - dungeon[i][j], 1);//若 dungeon[i][j]為 0, 則只要 1 活命. 若 dungeon[i][j]為正, 則有 dungeon[i][j]那麼多是不需要的, 所以要減. 若 grid[i][j]為負, 可類似想 .其實就是 Math.min(d[i + 1][j], d[i][j + 1])換成了 1
                else if (i == m - 1)    // go right
                    d[i][j] = Math.max(d[i][j + 1] - dungeon[i][j], 1);
                else if (j == n - 1)    // go down
                    d[i][j] = Math.max(d[i + 1][j] - dungeon[i][j], 1);
                else                    // go down or right
                    d[i][j] = Math.max(Math.min(d[i + 1][j], d[i][j + 1]) - dungeon[i][j], 1);
```

```
            }
        return d[0][0];
    }
```

2. 1-d DP (滚动数组) : Time ~ O(M*N), Space ~ O(min{M, N})

```
public int calculateMinimumHP(int[][] dungeon) {
        int m = dungeon.length;
        int n = dungeon[0].length;
        int min = Math.min(m, n), max = Math.max(m, n);
        int[] d = new int[min];
        int value;
//我自己寫的時候,可以直接按前面 1-d DP 那樣: for(int i = m - 1; …), for(int j = n - 1; …),不用 min 和 max(當然前面也不用求 min 和 max)了, 這樣的空間為 O(N). 用 min 和 max 的方法太繞, 容易出錯.
        for (int i = max - 1; i >= 0; i--)
            for (int j = min - 1; j >= 0; j--) {
//以下兩行先不看, 直接看下面的 if 和多個 else. 看完了再回來看這兩行. 由於那幾個 else 按照 min=n, max=m 來看, 是對的(代碼跟上面的 1-d DP 基本上是一樣). 那麼當 min=m, max=n 時, 就將 ij 互換一下就可以了.
                if (n == min)           value = dungeon[i][j];
                else /* (m == min) */   value = dungeon[j][i];
//以下的幾個 else, 從最後的 else 往前看, 就明白了. 看的時候就按 min=n, max=m 來看, 如此的話就跟上面的 1-d DP 基本上是一樣的了.
                if (i == max - 1 && j == min - 1)
                    d[j] = Math.max(1 - value, 1);
                else if (i == max - 1)          // go right
                    d[j] = Math.max(d[j + 1] - value, 1);
                else if (j == min - 1)          // go down
                    d[j] = Math.max(d[j] - value, 1);
                else                            // go down or right
                      d[j] = Math.max(Math.min(d[j], d[j + 1]) - value, 1);//右邊的 d[j]是上一輪的, 即 2-d DP 中的 d[i+1][j]. 右邊的 d[j+1]是本輪的, 即 2-d DP 中的 d[i][j+1];
            }
        return d[0];
    }
```

E2 的代碼:

```
public class Solution2 {
   public int calculateMinimumHP(int[][] dungeon) {
      if(dungeon == null || dungeon.length == 0 || dungeon[0].length == 0)
         return 0;

      int m = dungeon.length, n = dungeon[0].length;

      int[] d = new int[n]; //當然最好還是像 William 那樣, 將 d 的長度設為 min(m, n). 但實際上, 在本代碼中不大好改, 因為下面 dungeon[m-1][n-1]中哪個是 m, 哪個是 n 是固定的

      d[n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);
```

```
        for(int j = n - 2; j >= 0; j--) {
            d[j] = Math.max(d[j + 1] - dungeon[m - 1][j], 1);
        }

        for(int i = m - 2; i >= 0; i--) {
            d[n - 1] = Math.max(d[n - 1] - dungeon[i][n - 1], 1);

            for(int j = n - 2; j >= 0; j--) {
                d[j] = Math.max(Math.min(d[j], d[j + 1]) - dungeon[i][j], 1);
            }
        }

        return d[0];
    }
}
```

179. Largest Number, Medium

http://wlcoding.blogspot.com/2015/03/largest-number.html?view=sidebar

Given a list of non negative integers, arrange them such that they form the largest number.
For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330.
Note: The result may be very large, so you need to return a string instead of an integer.
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 將輸入數組排序(要轉換成 String 來排(因為後面要比較 ab 和 ba), 並且 implement Comparator interface, 比較 String a 和 String b 時, 要較 ab 和 ba, 因為只有這樣, 9 才能排到 34 前面, 其它人也是這樣做的, 注意若用 int 表示 ab 和 ba, 則 ab 和 ba 可能越界). 然後把它們結到 StringBuilder 上. 注意 Array.sort()默認是由小到大排的.

Corner case: {0, 0}, 結果應為 0, 而不是 00

History:
E1 直接看的答案. William 的代碼是用 new 直接建一個 Comparator 的 instance, 不大好理解. 網上很多其它人都是 implement 的 Comparator interface (Java 書 p532), 要好理解些. 而且 William 的代碼在語法上和邏輯上都有錯誤, 改了語法也通不過. 以下用我改進後(參考了網上其它人的代碼)後的 William 的代碼, 可以通過.
E2 改了幾次後通過.
E3 改了好幾次才通過, 主要是語法上的. E3 的方法跟 key 中一樣, 但比改進後的 William 代碼更簡明, 以後用 E3 改進後的代碼.

William:
Time ~ O(NlogN), Space ~ O(N)
Use Arrays.sort() and define a Comparator with the compare(a, b) method:
given two integer a and b,
if ab < ba, return -1;
if ab > ba, return 1;
if ab == ba, return 0.
After sort, append the strings from right to left to form a String, and delete leading zeros.
注意：
•两个 Integer append 后可能会 overload，所以必须不能直接比较 ab 和 ba 的值，要将两个 String 从高

位到低位依次比较。
•Comparator 用于 Integer 时，使用 Arrays.sort() 会有错，因为 num[i] 是 int 不是 Integer。
•要记得删除开头的 0。

我改進後的 William 代碼:

```
class StringComparator implements Comparator<String> { // Comparator 見 Java 書 p795. 一個典型的用 Comparator 的題見 56. Merge Intervals
 @Override
 public int compare(String a, String b) {
  String ab = a + b, ba = b + a;

  for(int i = 0; i < ab.length(); i++) {
   char c1 = ab.charAt(i), c2 = ba.charAt(i);
   if(c1 < c2) return -1;
   else if(c1 > c2) return 1;
   //若 c1 == c2, 則跳到下一個字符去
  }

  //以下是若所有字符都比完了, 說明 ab 和 ba 相符, 返回 0.
  return 0;
 }
}

public class Solution {
 public String largestNumber(int[] nums) {
  int n = nums.length;
  String[] numStr = new String[n];

  for(int i = 0; i < n; i++)
   numStr[i] = Integer.toString(nums[i]);

  Arrays.sort(numStr, new StringComparator());

  StringBuilder sb = new StringBuilder();
 // append numStr from right to left
  for(int i = n - 1; i >= 0; i--)
   sb.append(numStr[i]);
  // delete leading zeros
  int start = 0;

  while(start < n && sb.charAt(start) == '0')
   start++;

  String res = sb.substring(start);

  if(res.length() == 0)//處理輸入為[0]或[0,0]等情況
  return "0";
```

```
  return res;
 }
}
```

我 E3 改進後的代碼:

```
    public static String largestNumber(int[] nums) {
        String[] numsStr = new String[nums.length];


            //以下是將 int 數組轉化為 String 數組(上面 William 改進後的代碼也是這樣做的), 然後用 Comparator<String>來 sort. 我 E3 還按另一種寫法寫了並通過了, 就是這裡不將 int 數組轉化為 String 數組, 而 Comparator<String>按 Comparator<Integer>寫(注意, 但要將 int 數組轉化為 Integer 數組, 參見 Java p795 右邊角). 但是最後結果要用 String 表示, 所以最後 StringBuilder append 結果時, 還是要內部將 Integer 元素一個個轉化為 String, 這樣也沒簡化多少, 反而看起來不自然. 所以以後還是采用第一種寫法.
        for(int i = 0; i < nums.length; i++) numsStr[i] = Integer.toString(nums[i]);

        Comparator<String> comp = new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                String ab = a + b; //我 E3 最開始是將 a 和 b 轉化為 StringBuilder 然後 append 的, 這樣弄得代碼很長, 且運行時反而報錯, 後來就幹脆直接加. 上面 William 改進後的代碼也是這樣做的.
                String ba = b + a;
                return ab.compareTo(ba);
            }
        };

        Arrays.sort(numsStr, comp);

        StringBuilder res = new StringBuilder();
        for(int i = numsStr.length - 1; i >= 0; i--) res.append(numsStr[i]);

        while(res.length() >= 2 && res.charAt(0) == '0') res.deleteCharAt(0);
        return res.toString();
    }
```

186. Reverse Words in a String II, Medium, Locked

http://wlcoding.blogspot.com/2015/03/reverse-words-in-string-i-ii.html?view=sidebar

Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.
The input string does not contain leading or trailing spaces and the words are always separated by a single space.
For example,
Given s = "the sky is blue",
return "blue is sky the".
Could you do it *in-place* without allocating extra space?

Key: 不難. 先将整个 String reverse，再将每个单词逐个 reverse (可寫一個 reverse 函數)。判断单词只要遇到空格 i 即是该单词的 end，而 i + 1 则为下个单词的 start

History:
E1 直接看的答案.
E2 很快寫好, 但忘了反轉最後一個單詞, 改後即對我自己的例子給出正確答案, 算通過.
E3 很快寫好, 本來可以一次通過的, 但在寫循環時, i<=s.length 不小心寫成了 i<s.length, 改後即通過(在我電腦上結出正確結果).

Solution
Time ~ O(2N), Space ~ O(1)
先将整个 String reverse，再将每个单词逐个 reverse。
判断单词只要遇到空格 i 即是该单词的 end，而 i + 1 则为下个单词的 start。

```
//以下的 reverseWords 的參數確實是 char[], 而不是 String. 另外有兩個網站的答案也是 char[], 說明題目中就是 char[].
public void reverseWords(char[] s) {
    reverse(s, 0, s.length - 1);//反轉整個 string

    //以下是將每個單詞逐個反轉
    int start = 0;
    for (int i = 0; i <= s.length; i++) {
        if (i == s.length || s[i] == ' ') {//i == s.length 是為了反轉最後一個單詞
            reverse(s, start, i - 1);
            start = i + 1;
        }
    }
}

//reverse(c, start, end)的作用是將 c[start, ...end]這段反轉
private void reverse(char[] c, int start, int end) {
    while (start < end) {
       //以下三句的作用是交換 c[start]和 c[end], 然後 start++, end--
        char temp = c[start];
        c[start++] = c[end]; //start++返回的是增加前的值
        c[end--] = temp;
    }
}
```

187. Repeated DNA Sequences, Medium

http://wlcoding.blogspot.com/2015/03/repeated-dna-sequences.html?view=sidebar

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:
["AAAAACCCCC", "CCCCCAAAAA"].

Key: Hash Table + Bit Manipulation. 將 ACGT 分別 map 成 0123(可用數組實現 map)(0123 分別即二进制中的 00 01 10 11), 然後對給每個「10 個字符的子串」編碼(加密),如： AAACCC = 00 00 00 01 01 01（二进制） = 21 （十进制）. 密碼就存在 int v 中, 每一個俱體的 10 個字符的子串就有一個對應的 v 的值. 判斷 v 是否出現了兩次(用兩個 Set), 若是, 則加到結果的 list 中去. 算 v 時可用 v <<= 2. 實踐表明, 不對子串加密也可以, 直接將子串放入 Set 中判斷也能通過(Solution2b), 但 E3 表示這樣不好, 因為 Set 判斷它是否含某個子串時, 時間複雜度應該為子串長度; 而加密成一個數後, Set 判斷是否含此數時, 時間複雜度為 O(1), 加密之作用即在此. E3 也是將輸入 String 加密, 但是將其加密成一個四进製的數, 此方法比較容易想到, 以後也可以這樣寫.

Corner case: input "AAAAAAAAAAA" (11 個 A), output "AAAAAAAAAA" (10 個 A)

History:
E1 直接看的答案.
E2 最開始只用了一個 Set, 若 Set 中已含有某個 substring, 則將此 substring 加入結果中, 這樣做是不對的, 因為若有某個 substring 出現了三次, 則會在結果中加入兩次該 substring, 後來改為了用兩個 Set, 即通過.
E3 改了兩個粗心小錯誤後通過. E3 也是將輸入 String 加密, 但是將其加密成一個四进製的數. E3 改進後的代碼也帖在了最後, 有時間可以看看. E3 全是獨立想出來的, E3 都不知道 key 也是用的加密法, E3 寫之前還專門驗證過 10 位的四進製數不會溢出(顯然不會溢出, 因為 10 位的十進製數才會溢出).

我: 以下的方法中: 方法 1 空間超出了, 不看. 要掌握的是方法 2. 要看一看方法 3(變化不大, 好懂).

1. Suffix sort: Time ~ O(NlogN), Space ~ O(N^2) Memory Limit Exceeded!
(我： 代碼已省)

2. Hash Table + Bit Manipulation: Time ~ O(MN) where M = 10, Space(2N)

```
public List<String> findRepeatedDnaSequences(String s) {
    List<String> list = new ArrayList<>();
    Set<Integer> firstVisited = new HashSet<>();
    Set<Integer> secondVisited = new HashSet<>();
    char[] map = new char[26]; //注意 map 是一個數組, 而不是一個 HashMap, 這裡是用數組來實現 HashMap 的(網上有的人就是直接用的 HashMap). 另外, 此句改為 int[] map = new int[26]也能通過, 而且意思還明確些(E2 就是用的 int[] map).
    map['A' - 'A'] = 0;//即二进製的 00
    map['C' - 'A'] = 1;//即二进製的 01
    map['G' - 'A'] = 2;//即二进製的 10
    map['T' - 'A'] = 3;//即二进製的 11

    int len = 10;
    //以下 i 為長度為 10 的子串的開始
    for (int i = 0; i < s.length() - len + 1; i++) {//i 的上限取這個, 是為了保證下面 j 最大能取到 s.length()-1
        int v = 0;
        //以下是掃描從 i 開始的, 長度為 10 的子串
        for (int j = i; j < i + len; j++) {
            v <<= 2;//即 v=v<<2, 即 v 左移兩位
            v |= map[s.charAt(j) - 'A'];//這是將 v 的最右兩位(總是 0)換成 map[s.charAt(j) - 'A']的二进製值(比如 101101100 | 10 = 101101110). 這實際上是在給這個 10 個字符的子串編碼(加
```

```
密),如： AAACCC = 00 00 00 01 01 01 （二进制）= 21 （十进制）. 密碼就存在 v 中, 每一個俱體的
10 個字符的子串就有一個對應的 v 的值. E2 不是用的 |=, 而是用的+=, 也能通過.
        }
        //以下是判斷 v 是否出現了兩次, 若是, 則加到結果的 list 中去
                if  (!firstVisited.add(v)  &&  secondVisited.add(v))// 注 意 若 ！
firstVisited.add(v)返回為 false, 即 firstVisited.add(v)返回為 true, 就不會执行&&後的那個,
這正是我們想要的: 只在 firstVisited 中加, 不在 secondVisited 中加, 略巧.
            list.add(s.substring(i, i + len));//注意 substring 返回的是從 i 到 i+len-1 的
子串
    }
    return list;
}
```

3. Hash Table + Bit Manipulation: Time ~ O(N), Space(2N)
在方法二的基础上稍作了改动，得到 linear time performance。
给 DNA 编码（A - 00, C - 01, G - 10, T - 11），把 String s 变成 Binary Integer v。

•读取 sequence：首次连续读取 10 个 char；之后每次读取 1 个 char，将 v 左移两位（v << 2），在最后两位加上新读的 char（v |= map[s.charAt(i) - 'A']），然后舍弃最高的两位（v &= ~(3 << 2*10)，注意是左移 20 不是 10，因为一个 char 对应两位. 我: 即 v = v & 11000...0）。
•判断得到的 v 是否已经读过：这里巧妙地用了一行代码解决避免结果出现 duplicates
if (!firstVisited.add(v) && secondVisited.add(v))
注意：HashSet 的 add() 返回 boolean，若元素存在则返回 false；
所以上述的条件是指：之前访问过该元素，且之后尚未第二次访问到，这样就得到一个 repeated sequene，将其放入 list，同时这样也避免了之后再次遇到同样的 sequence，不会再将其放入 list，导致 duplicates。

```
public List<String> findRepeatedDnaSequences(String s) {
    List<String> list = new ArrayList<>();
    Set<Integer> firstVisited = new HashSet<>();
    Set<Integer> secondVisited = new HashSet<>();
    char[] map = new char[26];
    map['A' - 'A'] = 0;
    map['C' - 'A'] = 1;
    map['G' - 'A'] = 2;
    map['T' - 'A'] = 3;

    int len = 10, v = 0;
    for (int i = 0; i < s.length(); i++) {
        v <<= 2;
        v |= map[s.charAt(i) - 'A']; // add 2 bits at the end
        if (i == len - 1)   firstVisited.add(v);
        else if (i > len - 1) {
            v &= ~(3 << 2*len); // drop 2 bits at the highest place
            if (!firstVisited.add(v) && secondVisited.add(v))
                list.add(s.substring(i - len + 1, i + 1));
        }
    }
    return list;
}
```

也可用一个 Hash Table 记录 sequence 和其出现的次数，不用像上面用两个 Hash Sets：

```
public List<String> findRepeatedDnaSequences(String s) {
```

```
    List<String> list = new ArrayList<>();
     Map<Integer, Integer> visited = new HashMap<>(); // key - sequence, value -
count
    char[] map = new char[26];
    map['A' - 'A'] = 0;
    map['C' - 'A'] = 1;
    map['G' - 'A'] = 2;
    map['T' - 'A'] = 3;

    int len = 10, v = 0;
    for (int i = 0; i < s.length(); i++) {
        v <<= 2;
        v |= map[s.charAt(i) - 'A']; // add 2 bits at the end
        if (i == len - 1)   visited.put(v, 1);
        else if (i > len - 1) {
            v &= ~(3 << 2*len); // drop 2 bits at the highest place
            if (!visited.containsKey(v))    visited.put(v, 1);
            else                            visited.put(v, visited.get(v) + 1);
             if (visited.get(v) == 2)        list.add(s.substring(i - len + 1, i +
1));
        }
    }
    return list;
}
```

E3 改進後的代碼:

```
  public static List<String> findRepeatedDnaSequences(String s) {
    List<String> res = new ArrayList<>();
    if(s == null || s.length() < 10) return res;
    Set<Integer> values = new HashSet<>();
    int[] charInt = new int[26];
    charInt['A' - 'A'] = 0;
    charInt['C' - 'A'] = 1;
    charInt['G' - 'A'] = 2;
    charInt['T' - 'A'] = 3;

    int curVal = 0;

    for(int i = 0; i <= 9; i++)
      curVal = curVal * 4 + charInt[s.charAt(i) - 'A'];

    values.add(curVal);

    int fourPowerNine = 1;
    for(int i = 1; i <= 9; i++) fourPowerNine *= 4;

    for(int i = 1; i <= s.length() - 10; i++) {
      curVal = (curVal - charInt[s.charAt(i - 1) - 'A'] * fourPowerNine) * 4 + charInt[s.charAt(i + 9) -
'A'];

      if(!values.contains(curVal)) values.add(curVal);
      else if(!res.contains(s.substring(i, i + 10))) res.add(s.substring(i, i + 10));
    }
```

```
        return res;
    }
```

188. Best Time to Buy and Sell Stock IV, Hard

http://wlcoding.blogspot.com/2015/03/best-time-to-buy-and-sell-stock.html?view=sidebar

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

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

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

Subscribe to see which companies asked this question

Key: 2-d DP. Let d(i, j) be the max profit of i transactions in [0, j] (1 <= i <= k, 0 <= j <= N – 1). 分 There is no sell at j 和 There is a sell at j (there must be a buy at l < j)兩種情況, 盡量不要看下面的遞推式. William 是按 2-d DP 寫的, 我檢查了他 solution 的網頁, 沒有 1-d DP.
注意 William 的代碼時間是 O(N^2), 所以只有 ij 兩層循環, 沒有 l 循環. 這樣做的方法是: 將 l 循環和 j 循環合併, 用一個變量 tmpMax 來記錄 max_{1 <= l <= j} {d(i - 1, l - 1) – p[l]}.
另外, 要把對 d 的定義(和空間分配)放到處理 'k 超過數組的一半' 那段之後, 否則會說 space limit exceeded.

History:
E1 直接看的答案.

E2 寫得跟答案已經很接近了, 但 space limit exceeded, 為了查出這是甚麼原因, 看了答案, 並發現我還要對 j=0 時初始化 tmpMax. 將 space limit exceeded(which I think is more like just a convention)解決了後, 再將初始化 tmpMax 那句添上, 我的代碼就通過了, 但由於已看答案中的初始化 tmpMax 那句, 所以仍算此題我沒做出來. 實際上要是沒被迫看到初始化 tmpMax 那句, 我自己應該也能寫出來. 以上 key 大多數是 E2 寫的.

E3 最開始寫為的 i, j, l 三層循環, 結果超時了, 後來又將 l 跟 j 循環合併(跟 key 不謀而合), 改了幾下, 即通過. E3 代碼跟 William 稍有不同, 沒有初始化 tmpMax 那句.

我: 本題用 William 的代碼. 千萬不要看 Code Ganker 在 123. Best Time to Buy and Sell Stock III 中的代碼 (Code Ganker 沒有本題(188 題)的答案), 他說可以用於 k 的情況, 但實際上把他的代碼修改後在本題的 leetcode 中通不過, 而且那個代碼很不好懂, 他表述也不清楚, 花了我幾個小時的時間去證明, 還沒完全證出來. 評論裡也有很多人看不懂. 所以他的代碼有可能是運氣好撞對了. 不要浪費時間去看. 用以下 William 的清新代碼就是了.

William:
Solution: 2-d DP, Time ~ O(N^2)
Let d(i, j) be the max profit of i transactions in [0, j] (1 <= i <= k, 0 <= j <= N – 1). (我: N 為 prices 的長

度)

Two case:

- There is no sell at j: d(i, j) = d(i, j - 1)
- There is a sell at j (there must be a buy at l < j): d(i, j) = max_{1 <= l < j} {d(i - 1, l - 1) - p[l] + p[j]} (我: 此 max 是取各種 l 時的最大值. l 要從 1 開始, 否則 d 中的 l-1 小於 0)

d(i, j) = max{d(i, j - 1), max_{1 <= l <= j} {d(i - 1, l - 1) - p[l] + p[j]}}
(我: 以上最外層的 max 是取兩項中較大的一項)
= max{d(i, j - 1), p[j] + max_{1 <= l <= j} {d(i - 1, l - 1) - p[l]}} // move p[j] out of max

profit_max = d[k][N - 1], 我: 取 k 是因為當然交易越多越好.

注意：

- int[][] d = new int[k + 1][N]
- 如果 k 超过数组的一半（k > len / 2），即为 as many transactions as you like 的 case，可简化。
  我: 注意題目是 at most k transactions, 而不是 exact k transactions.

```
public int maxProfit(int k, int[] prices) {
        int len = prices.length;
        int profit = 0;

        // as many transactions as we want
        if (k > len / 2) {
            for (int i = 0; i < len - 1; i++) {
                int diff = prices[i + 1] - prices[i];
                if (diff > 0)   profit += diff;
            }
            return profit;
        }

        // at most k transactions
        // d(i, j) = max{d(i, j - 1), max_{1<=l<=j} {d(i - 1, l - 1) - p[l] (buy) + p[j] (sell)}}
        int[][] d = new int[k + 1][len]; //注意這裡自動將 d[i][0]初始化為了 0, 其中 i 為任意值. 且後面也從來沒有給 d[i][0]賦過值, 所以 d[i][0]求遠等於 0.

        for (int i = 1; i <= k; i++) {
            int tmpMax = d[i - 1][0] - prices[0]; //先看下面我對 tmpMax 的意思的解釋. 此句中若刪掉 d[i - 1][0]也能通過(因為 d[i - 1][0]等於 0). 這句實際上是在處理一個殊情況: 當下面那個 for 中的 j 取 1 的時候, 下面那個 for 中的 d[i-1][0]代表的是在第 0 天做交易的 profit, 若按下面那樣將其加入到第 1 天買後來某天賣所賺的錢中, 這樣算出來的結果是不對的. 實際上第 0 天做交易是無效的, 沒有獨立的行動能力, 後面的交易必須來接管它, 即要將後面的交易買入的時間提前到第 0 天, 所以此式中右邊是 prices[0], 然後再將結果跟第 1 天買入的那個交易 PK. 本段是 E1 寫的. E2 沒看, 但 E2 想到的是一樣的. E1 就能想到, 不容易啊. E3 的寫法不同, 所以不是必須要這樣寫.
            for (int j = 1; j < len; j++) {
                tmpMax = Math.max(tmpMax, d[i - 1][j - 1] - prices[j]);
//想一想即知, tmpMax 其實就是 max_{1 <= l <= j} {d(i - 1, l - 1) - p[l]}
                d[i][j] = Math.max(d[i][j - 1], prices[j] + tmpMax);
            }
        }
        return d[k][len - 1];
    }
```

189. Rotate Array, Easy

http://wlcoding.blogspot.com/2015/03/rotate-array.html?view=sidebar

Rotate an array of *n* elements to the right by *k* steps.
For example, with *n* = 7 and *k* = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].
Note:
Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.
[show hint]
Hint:
Could you do it in-place with O(1) extra space?
Related problem: Reverse Words in a String II
Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 題目中的數組一次 rotate 就是: 所有元素右移一位, 最右的元素移到最左邊. Reversal algorithm: rev(rev(A) rev(B)) = BA
History:
E1 用的 brute force, 真的是 rotate k 次, 每次所有元素右移一位, 結果超時了, 複雜度是 O(kN), 沒得出正確結果, 因為知道必超時, 就沒細改.
E2 用的以上 Key 的方法, 通過.
E3 沒做出來. E3 最開始也是用的 E1 的 brute force 方法(這是 in-place 的), 幾分鍾寫好, 能得出正確結果(E1 這麼簡單的都沒寫對), 能對題目中例子給出正確結果, 但在 OJ 上超時了. E3 後來又想了一會兒別的 in-place 方法, 沒想出來.

我: 以下 William 給出的 4 種方法, 要掌握的是方法 4. Reversal algorithm. 方法 1 超時不看, 方法 2 和 3 要看看. 方法 2 和方法 3 太繞, 沒必要花太多時間. 方法 2 看看算法就可以, 代碼和證明可以不管. 方法 3 大概看看算法和代碼就可以了. (E2 沒看方法 2 和方法 3). 注意題目要求是 Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

William:
注意：k 可能大于 nums.length，为提高效率（避免多次遍历 array）要先取模 k = k % num.length。

1. Rotate one by one: Time Exceed Limit! (我: 我的代碼跟 William 的差不多, 但沒得出正確結果, 因為知道必超時, 就沒細改, 但代碼跟 William 只有很小差別. 由於超時, 所以不看, 代碼已省)
1) One direction: Time ~ O(KN), Space ~ O(1)

2) Two directions (improved version): Time ~ O(min{K, N-K}*N), Space ~ O(1) (我: 由於时間復雜度還是不是 O(n), 所以不看, 代碼已省)

Below there are three linear time algorithms:

2. Juggling algorithm: Time ~ O(N), Space ~ O(1)
外层循环：将 array 分成若干个 block，每个 block 的 size 为 n 和 k 的最大公约数 gcd(n, k)，即外层循环数；
内层循环： l 每次跳跃 k 格，将新元素 l 替代老元素 j： num[j] = num[l]，并将 j = l，直至 j 回到出发点停止。
(我: 以下是偽碼)

```
for ( i = 0; i < gcd(n, k); i++) {
   while ( j != jStart) {
     A[j] = A[l];
     // move right: j = n - 1 - i, l = j - k
     // move left:  j = i, l = j+ k
    }
}
```

Example: n = 7, k = 3 (move right)

gcd(7, 3) = 1，仅循环一遍即可完成：

- 依次向左走，每次跳 3 格 l = j - 3，用左边的取代右边的值：num[j] = num[l];
- 走到左端后，再从右端返回： if (l < 0) l+= n;
- 直到 l 再次回到出发点 n - 1 - i = 6 停止。

```
 0  1  2  3  4  5  6
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 4] j = 6, l = 3
[1, 2, 3, 1, 5, 6, 4] j = 3, l = 0
[5, 2, 3, 1, 5, 6, 4] j = 0, l = -3 -> 4
[5, 2, 3, 1, 2, 6, 4] j = 4, l = 1
[5, 6, 3, 1, 2, 6, 4] j = 1, l = -2 -> 5
[5, 6, 3, 1, 2, 3, 4] j = 5, l = 2
[5, 6, 3, 1, 2, 3, 4] j = 2, l = -1 -> 6  break
[5, 6, 7, 1, 2, 3, 4]
```

```
public void rotate(int[] nums, int k) {
    int n = nums.length;
    k = k % n;
    if (k < n - k)  rotateRight(nums, k);
    else            rotateLeft(nums, n - k);
}

public void rotateRight(int[] nums, int k) {
    int n = nums.length;
     for (int i = 0; i < gcd(n, k); i++) { // there're gcd(n,k) elements in each
block
        int last = nums[n - 1 - i];
        int j = n - 1 - i;
        while (true) {
            int l = j - k;
            if (l < 0) l += n;
            if (l == n - 1 - i) break;
            nums[j] = nums[l];
            j = l;
        }
        nums[j] = last;
    }
}

public void rotateLeft(int[] nums, int k) {
    int n = nums.length;
    for (int i = 0; i < gcd(n, k); i++) {
        int first = nums[i];
        int j = i;
        while (true) {
            int l = j + k;
            if (l >= n) l -= n;
```

```
            if (l == i) break;
            nums[j] = nums[l];
            j = l;
        }
        nums[j] = first;
    }
}

// greatest common divisor
private int gcd(int n, int k) {
    if (k == 0) return n;
    else        return gcd(k, n % k);
}
```

3. Block-Swap algorithm: Time ~ O(N), Space ~ O(1)
AB -> BA, assume A is shorter, divide the B into Bl Br (Br.len = A.len);
A Bl Br -> divide B and swap(A, Br): Br Bl A -> divide Br and swap... -> Bl Br A

```
public void rotate(int[] nums, int k) {
    k = nums.length - k % nums.length;
    if (k == 0 || k == nums.length)   return;

    // move to the Left
    int i = k, j = nums.length - k; // A (len = i), B (len = j)
    while (i != j) {
        if (i < j) {    // A is shorter
            swap(nums, k - i, k - i + j, i);    // A,Bl,Br -> Br,Bl,A (A=[k-i..k-
1], Br=[k-i+j...k-1+j])
            j -= i;
        } else {        // B is shorter
            swap(nums, k - i, k, j);            // Al,Ar,B -> B,Ar,Al (Al=[k-i..k-
i+j-1], B=[k..k+j-1])
            i -= j;
        }
    }
    // i == j: A and B are with equal length
    swap(nums, k - i, k, i);    // A,B -> B,A (A=[k-i..k-1], B=[k..k+i-1])
}

// swap d elements starting at left and starting at right
private void swap(int[] nums, int left, int right, int d) {
    for (int i = 0; i < d; i++) {
        //以下三句是交 nums[left+1]和 nums[right+1]
        int temp = nums[left + i];
        nums[left + i] = nums[right + i];
        nums[right + i] = temp;
    }
}
```

4. Reversal algorithm: Time ~ O(N), Space ~ O(1)
rev(rev(A) rev(B)) = BA

rev 運算跟矩陣轉置差不多: rev(rev(A) rev(B)) = rev(rev(B)) rev(revA) = BA. 可用例如題目中的例子驗證(真接算, 不用我對 rev 的展開式): A=[1,2,3,4], B=[5,6,7], 則 rev(rev(A) rev(B)) = rev([4,3,2,1,7,6,5]) = [5,6,7,1,2,3,4]

用以下代碼:

```
public void rotate(int[] nums, int k) {
    //可先寫一句int n = nums.length
    k = k % nums.length;
    // A (len = n-k), B (len = k)
    // rev(rev(A)rev(B)) = BA
    reversal(nums, 0, nums.length - k - 1);
    reversal(nums, nums.length - k, nums.length - 1);
    reversal(nums, 0, nums.length - 1);
}

//reversal(nums, start, end)的作用是將nums[start, ...end]這段反轉
private void reversal(int[] nums, int start, int end) {
    while (start < end) {
        //以下三句的作用是交換nums[start]和nums[end]. 也可按「下面貼出來的186. Reverse Words in a String II中的相同功能的代碼寫法」來寫
        int temp = nums[start];
        nums[start] = nums[end];
        nums[end] = temp;
        start++;
        end--;
    }
}
```

190. Reverse Bits, Easy

http://wlcoding.blogspot.com/2015/03/reverse-integer-bits.html?view=sidebar

Reverse bits of a given 32 bits unsigned integer.
For example, given input 43261596 (represented in binary as **00000010100101000001111010011100**), return 964176192 (represented in binary as**00111001011110000010100101000000**).
Follow up:
If this function is called many times, how would you optimize it?
Related problem: Reverse Integer
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 定義一個 val 初值設為 0, 每次左移一位, 然後將 n 最右邊一位的數(n&1)插到 val 最右邊去(用 |), 然後 n 右移一位 . 當 n 為負數時之處理的約定, 見下面我說的和 William 說的. 若要輸出來檢查, 則可用 Integer.toString(13, 2), 結果為 1101, 即 13 的二进製碼(已 record 到 java p404).

History: E1 直接看的答案. E2 幾分鍾好, 但將循環條件寫為了 while(n > 0), 得出了錯誤的結果, 這是因為 n 可能有前導 0(因為要填滿 32 位), 如 n = 000001101..., 當 n 右移為 000001 時, 再右移就成為 0 了, 循環結束, 那幾個前導 0 沒法弄到 val 中去, 正確的作法是用 for 指明右移 32 次, 改後即通過.

我: 注意本題的代碼欄中要求的是 you need treat n as an unsigned value , 但函數中要求的是 int: public int reverseBits(int n). 這是因為 Java 中沒有 usigned 類型, 所以我們要把 int 當成一個 unsigned (這是題目的要求, 因為 leetcode 代碼框中有一句: “you need treat n as an unsigned value”), 即 William 下面所說的 . 若在 leetcode 中切換成 C++, 代碼欄中的函數變為 uint32_t reverseBits(uint32_t n), 這裡用 uint32_t 是因為 C++中的 int(包括 unsigned)是 16 位的.

William:
注意是 unsigned integer，int 定义的是 signed integer，首位为 1 时表示 -1 (我: 詳見 java p55)，而在这里要把这一位当成数值进行 reverse。

Bit Manipulation: Time ~ O(N), Space ~ O(1)
n & 1：取最后一位的值；
val << 1：进位；
|：加上末尾值，注意用 + 会出错，由符号位引起，区别 -1 + 1 = 0 和 -1 | 1 = -1。

```
// you need treat n as an unsigned value
public int reverseBits(int n) {
    int val = 0;
    for (int i = 0; i < 32; i++) {//題目中說了的是 32 bits unsigned integer
    //以下一句是將 n 最右邊一位的數插到 val 最右邊去
        val = val << 1 | n & 1; // use | instead of +
        n = n >> 1; //若將此句寫為 n>>1, 則 compiler 會提示: error: not a statement //E2 用的 n >>>= 1, 雖然都能通過(因為總共只移 32 次, 所以最左加 0 或 1 都一樣), 但用>>>=意義要好些.
    }
    return val;
}
```

191. Number of 1 Bits, Easy

Write a function that takes an unsigned integer and returns the number of ’1' bits it has (also known as the Hamming weight).

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

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

Subscribe to see which companies asked this question

Key: 定義一個 val 初值設為 0, 每次在 val 中加上 n 的最右一位數, 然後 n 右移一位.

History: E1 的代碼幾分鍾寫好並一次通過, 這是第一個我自己做出來的位運算題目. 我的方法和 William 的一樣, 只是在語句表述上有點小差別, 我的更簡練點. 以下用我的代碼. William 說的 Time ~ O(N), Space ~ O(1), 我的顯然也一樣. E2 一分鍾寫好(不是幾分鍾), 一次通過.

```
// you need treat n as an unsigned value
 public static int hammingWeight(int n) {
  int val = 0;

  for(int i = 0; i < 32; i++) {
   val += n & 1; //在 val 中加上 n 的最右一位數
   n >>= 1; //n 右移一位. 與 190 題一樣, 注意不要將此句寫成了 n>>1, 否則則 compiler 會提示: error: not a
```

statement. //E2 用的 n >>>= 1, 雖然都能通過(因為總共只移 32 次, 所以最左加 0 或 1 都一樣), 但用>>>=意義要好些.
 }

 return val;

198. House Robber, Easy

http://wlcoding.blogspot.com/2015/03/house-robber.html?view=sidebar

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.
Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.
Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases. Also thanks to @ts for adding additional test cases.
Subscribe to see which companies asked this question

Key: 動態歸劃. 題目的意思就是不能在數組中相鄰的兩個元素中拿. 遞推式為 d(i) = max{d(i - 1), d(i - 2) + s[i]}, 其中 d(i)表示從 nums[0,...i]中拿到的最多錢.

History:
E1 的代碼也一次通過, 這是我自己做出的第一個動態歸劃題. 我的遞推式是 d[i][0] = Math.max(d[i - 1][0], d[i - 1][1]); d[i][1] = d[i - 1][0] + nums[i]; 其中 d[i][0]表示從 nums[0,...i]中拿到的最多錢且不拿第 i 個, d[i][1]表示從 nums[0,...i]中拿到的最多錢且要拿第 i 個. 我的方法應該算是 2-d DP, 盡管第二維長度只有 2. 時間複雜度為 O(N), 空間複雜度為 O(2N). 我試過將代碼改成 1-D DP, 但好像不大好改. William 用的 1-d DP, 所以以下用 William 的代碼.
E2 六分鍾寫好, 一次通過.
E3 幾分鍾寫好, 本來可以一次通過的, 結果在初始化 d[1]時, 忘了加 if(n > 1), 改後即通過.

stash: 隱藏

William:
1-d DP: Time ~ O(N), Space ~ O(N)
Let d(i) be the max money that the robber can get from house 0 to i.
d(i) = max{d(i - 1), d(i - 2) + s[i]}; (我: max 中第一項 d(i-1)表示不拿第 i 個, 第二項 d(i-2) + s[i])表示要拿第 i 個. 由此式可知, DP 不一定要覆蓋前面的, 普通的遞推也可以叫 DP)
d(0) = s[0];
d(1) = max{d(0), s[1]}; (我: 或寫成 d(1) = max{s[0], s[1]}比較好看)
Return d(N - 1).

我: William 的原代碼寫法不大符合我(probably 很多其他人)的習慣, 在 for 中用了幾個 if-else 來初始化 d[0]等, 看起來有點別扭, 以下用我改寫後的代碼, 改後可以通過.

```
public int rob(int[] nums) {
 if(nums == null || nums.length == 0)
```

```
   return 0;

  int n = nums.length;

  int[] d = new int[n];
  d[0] = nums[0];
  if(n > 1)
   d[1] = Math.max(nums[0], nums[1]);

  for(int i = 2; i < n; i++)
   d[i] = Math.max(d[i - 1], d[i - 2] + nums[i]);

  return d[n - 1];
 }
```

199. Binary Tree Right Side View, Medium

http://wlcoding.blogspot.com/2015/03/binary-tree-right-side-view.html?view=sidebar

Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.
For example:
Given the following binary tree,

```
   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

You should return [1, 3, 4].
Credits:
Special thanks to @amrsaqr for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 簡單. BFS. 每層結束時(if(lastNum == 0)), 將結點的值放入 res 中即可.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過. E2 是按 Code Ganker 的 BFS 的風格寫的, 沒有像 William 那樣每層結束時放 null. 我覺得 William 的 BFS 的風格沒 Code Ganke 的好, 因為每層都放 null, 佔用的空間要多點. 為了與之前的 BFS 題一致及方便記憶, 以後都用 E2 的代碼(在最後). 但 William 的代碼也要看看.
E3 幾分鍾寫好, 一次通過, 代碼跟 E2 的一模一樣(變量名除外).

William's Tag: Tree Traversal (BFS)
William:
Level-order Traversal: Time ~ O(N), Space ~ O(N)
因为可能存在右边比左边长的部分，所以要遍历所有节点找到每行的最后一个 Node（q.peek() == null）。

我: 原代碼中的 TreeNode 類已省

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> list = new ArrayList<>();
```

```
    if (root == null)   return list;
    Queue<TreeNode> q = new LinkedList<>();//左右為不同類型(Queue 和 LinkdedList)的時候,
右邊仍可省略<>中的內容.
    q.add(root);
    q.add(null);

    while (!q.isEmpty()) {
         TreeNode curr = q.poll();//因為下面的 while 結束時, curr 必為 null, 然後再來到本
while 的此句時, 要再 poll 一個非 null 的節點出來才得行
        while (curr != null) {
            if (curr.left != null)  q.add(curr.left);
            if (curr.right != null) q.add(curr.right);
             if (q.peek() == null)   list.add(curr.val);//q.peek()即 curr 在 queue 中的
下一個
            curr = q.poll();//將本層末尾的那個 null 刪掉
        }
        if (!q.isEmpty()) q.add(null);//此句意思是: 若本層已掃描完(即前面 while 遇上了 null
而終止, null 標志一層結束), 則在 queue 中「本層的所有子節點」(前面的 while 放的就是本層的所有子節
點 )的那段也加一個 null, 表示那層也結束了. 為何 if 中的條件是!q.isEmpty()?以 while 中的 curr 為
「此樹最後一層的最右一個節點」為例想一想就知道了, 這其實是防此前面的 while(!q.isEmpty())成為死循
環, 注意若 queue 中只有一個 null 節點, queue 也算是非空.
    }
    return list;
}
```

E2 的 Code Ganker 風格代碼:

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> res = new ArrayList<>();

    if(root == null)
        return res;

    LinkedList<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    int lastNum = 1;
    int curNum = 0;

    while(!queue.isEmpty()) {
        TreeNode cur = queue.poll();
        lastNum--;

        if(cur.left != null) {
            queue.offer(cur.left);
            curNum++;
        }
```

```
            if(cur.right != null) {
                queue.offer(cur.right);
                curNum++;
            }

            if(lastNum == 0) {
                res.add(cur.val);
                lastNum = curNum;
                curNum = 0;
            }
        }

        return res;
    }
```

200. Number of Islands, Medium

http://wlcoding.blogspot.com/2015/03/number-of-islands.html?view=sidebar

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

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: DFS. 寫一個 dfs(grid, i, j), 其作用為：look at grid[i][j], 若它不為 1，則甚麼都不做；若只有它一個人是 1，即四周（它在邊界上就是三周甚麼的）都不是 1, 則將它一個人換為＃；若它四周有 1，則將那一堆連通的 1 都換為#. 用 dfs 函數就可以探測出哪片區域是 island, 然後用一個 count 來記數就可以了.

History:
E1 直接看的答案.
E2 很快寫好, 本來可以一次通過的, 但不小心將 grid[i][j] == '1'寫為了 grid[i][j] == 1, 改後通過.
E3 一次通過, 代碼跟 William 的基本一樣. E3 早就把 key 忘了, 方法基本上是自己獨立想出來的, E3 是將'1'換為'2', 這跟 key 中的'#'不謀而合.

我: William 給了 BFS 和 DFS 兩種方法, 要掌握的是 DFS(因為簡單些. 注意是方法 2, 不要看錯成方法 1 了), BFS 也要看一看, 但由於 BFS 中的 bfs 函數和 visit 函數跟 130 題的一模一樣, 所以這裡先不看, 做了 130 題後可以再回來掃一眼.
E3 建議以後都用 BFS 方法, 因為我 E3 在 130. Surrounded Regions 題中用本題一模一樣的 DFS 方法出現了 StackOverflowError(對題目中例子能給出正確結果), 網上很多人的 DFS 都這樣, Code Ganker 在 130 題中也建議對於棋盤類題都用 BFS 而不是 DFS.

1. BFS: Time ~ O(MN), Space ~ O(MN)

```
public int numIslands(char[][] grid) {
    int m = grid.length;
    if (m == 0) return 0;
    int 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') {//由於 bfs 會將 1 換為#, 所以不會重復計算
                bfs(grid, i, j);
                count++;
            }
        }
    // recover the grid
    //不要以下的 recover 這段也能通過
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            if (grid[i][j] == '#')
                grid[i][j] = '1';
    return count;
}

//以下的 bfs 函數跟 130. Surrounded Regions, Medium 中的 visit 一模一樣(除了變量名字外).
//bfs(grid, i, j)的作用為：look at grid[i][j], 若它不為 1，則甚麼都不做；若只有它一個人是 1，即四周（它在邊界上就是三周甚麼的）都不是 1, 則將它一個人換為 # ；若它四周有 1，則將那一堆連通的 1 都換為 # . 注意並不會遞歸調用 bfs()函數.
private void bfs(char[][] grid, int i, int j) {
    Queue<Integer> q = new LinkedList<>();
    visit(grid, i, j, q);
    while (!q.isEmpty()) { //不會無限循環下去, 因為已 visit 過的點都已換成了#, 不會再进入 queue 了
        int pos = q.poll();
        int x = pos / grid[0].length;
        int y = pos % grid[0].length;
        visit(grid, x - 1, y, q);
        visit(grid, x + 1, y, q);
        visit(grid, x, y - 1, q);
        visit(grid, x, y + 1, q);
    }
}
```

```
//以下的visit函數跟130. Surrounded Regions, Medium中的visit一模一樣(除了變量名字外).
//visit(grid, i, j, q)的作用為：若grid[i][j]為1，則將其換為#, 並將i, j放入q中．若grid[i][j]不為1，則甚麼都不做．注意本函數並不要求grid[i][j]在邊界上
private void visit(char[][] grid, int i, int j, Queue<Integer> q) {
    if (i < 0 || i > grid.length - 1 || j < 0 || j > grid[0].length - 1 || grid[i][j] != '1') return;
    grid[i][j] = '#';
    q.add(i * grid[0].length + j);//將坐標i,j換化為了一個數存到了q中，相當於一個加密過程，後面用到q中的元素時會有相應的解密過程．巧．
}
```

2. DFS: Time ~ O(MN), Space ~ O(MN) (我: 注意 space)
对每个为 '1' 的点使用 DFS，访问所有与其连接的 '1'，并将 '1' 标记为 '#'。每完成一次 DFS 就得到一个 island，最后将所有的 '#' 变为 '1' 恢复 grid。

//以下的 numIslands 函數除了 dfs(grid, i, j)一句外, 其餘的和 BFS 方法(那裡為 bfs(grid, i, j))中的 numIslands 一模一樣

```
public int numIslands(char[][] grid) {
    if (grid.length == 0)   return 0;

    int 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') {//由於dfs會將1換為#, 所以不會重復計算
                dfs(grid, i, j);
                count++;
            }
    // recover the grid
    //不要以下的recover這段也能通過
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            if (grid[i][j] == '#')
                grid[i][j] = '1';
    return count;
}

//dfs(grid, i, j)的作用為：look at grid[i][j], 若它不為1，則甚麼都不做；若只有它一個人是1，即四周（它在邊界上就是三周甚麼的）都不是1, 則將它一個人換為#；若它四周有1，則將那一堆連通的1都換為#．之所以要換為#, 是為了防止多次走到同一點上時引起的額外運算(甚至死循環).

private void dfs(char[][] grid, int i, int j) {
    if (grid[i][j] == '0')  return; //若將if改為if (grid[i][j] == '0' || grid[i][j] == '#'), 意思會更明確(功能是一樣的), 也能通過.
    if (grid[i][j] == '1') {
        grid[i][j] = '#'; //別把此句忘了
        int m = grid.length, n = grid[0].length;
        if (i - 1 >= 0)   dfs(grid, i - 1, j);
        if (i + 1 < m)    dfs(grid, i + 1, j);
        if (j - 1 >= 0)   dfs(grid, i, j - 1);
        if (j + 1 < n)    dfs(grid, i, j + 1);
    }
}
```

E2 的 dfs 函數是寫為以下這樣的, 也能通過, 但更簡潔:

```
public void dfs(char[][] grid, int i, int j) {
    if(i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] != '1')
        return;

    grid[i][j] = '#';

    dfs(grid, i - 1, j);
    dfs(grid, i + 1, j);
    dfs(grid, i, j - 1);
    dfs(grid, i, j + 1);
}
```

201. Bitwise AND of Numbers Range, Medium

http://www.cnblogs.com/grandyang/p/4431646.html

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.
For example, given the range [5, 7], you should return 4.
Credits:
Special thanks to @amrsaqr for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 返回結果就是 m 和 n 左起的共同部分(該共同部分的數碼可以為 1, 也可以為 0), 後面的部分就都是 0. 平移 m 和 n，每次向右移一位，直到 m 和 n 相等，记录下所有平移的次数 i，然后再把 m 左移 i 位即为最终结果.

History: E1 直接看的答案. E2 幾分鍾寫好, 一次通過.

我: William 的代碼不太簡明, 而且不太好懂. 以下用 Grandyang 的代碼. Grandyang 給了兩種方法, 我要掌握的是第二種方法, 第一種也要看一看.

Grandyang:
又是一道考察位操作 Bit Operation 的题，相似的题目在 LeetCode 中还真不少，比如 Repeated DNA Sequences 求重复的 DNA 序列，Single Number 单独的数字，Single Number II 单独的数字之二，Grey Code 格雷码，和 Reverse Bits 翻转位等等，那么这道题其实并不难，我们先从题目中给的例子来分析，[5, 7]里共有三个数字，分别写出它们的二进制为：
101　　110　　111
相与后的结果为 100，仔细观察我们可以得出，最后的数是该数字范围内所有的数的左边共同的部分(我: 該共同部分的數碼可以為 1, 也可以為 0, 見下面網上給的例子)，如果上面那个例子不太明显，我们再来看一个

范围[26, 30]，它们的二进制如下：
11010 11011 11100 11101 11110
(我: 結果即為 11000). 发现了规律后，我们只要写代码找到左边公共的部分即可.

我: 網上還有人給了兩個例子:
Num1: 110010
Num2: 110111
Result: 110000 (我: 左起第三位, 都是 0)

和
Num1: 0110
Num2: 1100
Result: 0

我: 為何可以直接比 m 和 n, 而不用管它們之間的數? 因為: 對於 m 和 n(相同位上)不相同的數碼, 不管它們之間的數為多少, 至少 m 和 n 該位上的數不同了, 所以 m&n 後該位上為 0, 而這個 0 跟其它數&後還是 0, 所以只用看 m 和 n. 對於 m 和 n(相同位上)相同的數碼, 它們之間的數在該位上的數碼也必然相同, 否則不在 m 到 n 的范圍內, 所以也只用看 m 和 n.

Grandyang:
我们可以从建立一个 32 位都是 1 的 mask (我: 即 Integer.MAX_VALUE)，然后每次向左移一位，比较 m 和 n 是否相同，不同再继续左移一位，直至相同，然后把 m 和 mask 相与就是最终结果，代码如下(我: 原代碼是 C++, 以下是我改寫後的 Java 代碼, 可通過):

```
public int rangeBitwiseAnd(int m, int n) {
 int d = Integer.MAX_VALUE;

 while ((m & d) != (n & d)) {
  d <<= 1;
 }

 return m & d;
}
```

此题还有另一种解法，不需要用 mask，直接平移 m 和 n，每次向右移一位，直到 m 和 n 相等，记录下所有平移的次数 i，然后再把 m 左移 i 位即为最终结果，代码如下：
解法二(我: 原代碼是 C++, 以下是我改寫後的 Java 代碼, 可通過)

```
public int rangeBitwiseAnd(int m, int n) {
```

```
 int i = 0;

 while(m != n) {
  m >>= 1;
  n >>= 1;
  i++;
 }

 return m <<= i;
}
```

202. Happy Number, Easy

http://wlcoding.blogspot.com/2015/04/happy-number.html?view=sidebar

Write an algorithm to determine if a number is "happy".
A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.
Example: 19 is a happy number

- 1^2 + 9^2 = 82
- 8^2 + 2^2 = 68
- 6^2 + 8^2 = 100
- 1^2 + 0^2 + 0^2 = 1

Credits:
Special thanks to @mithmatt and @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 其實就是 brute force 地一個一個試. 用 HashSet 存儲每次得到的数位平方和，如果遇到出现过的结果则表明不是 happy number，如果遇到 1 则表明是 happy number，除此之外则继续循环. 用 n%10 可以得到 n 最低位的數.

History:
E1 覺得 brute force 一定超時, 就沒寫.
E2 通過了.
E3 幾分鍾寫好, 本來可以一次通過的, 結果忘了每次循環結束時將 sum 清零(E3 代碼跟 William 的略有區別), 改後即通過.

我: 以下做法其實就是 brute force 地一個一個試(網上很多其他人也是這樣做的), 我最開始也想到過這樣做, 但覺得一定超時, 就沒寫.

William:
Time ~ O(?), Space ~ O(?)
用 Hash Set 存储每次得到的数位平方和，如果遇到出现过的结果则表明不是 happy number，如果遇到

1 则表明是 happy number，除此之外则继续循环。

```
public boolean isHappy(int n) {
    Set<Integer> set = new HashSet<>();
      int sum = n;// 此處將 sum 初始賦為 n, 是為了第一次进入 while 時, 在 sum = sumDigitSquare(sum)中好計算 sumDigitSquare(n). 在 set 中多加一個 n 的值, 並不會影响結果.
    while (sum != 1) {
        if (!set.contains(sum)) set.add(sum);
        else                    return false;
        sum = sumDigitSquare(sum);
    }
    return true;
}

private int sumDigitSquare(int n) {
    int sum = 0;
    while (n > 0) {
        sum += Math.pow(n % 10, 2); //此句也可寫為 sum += (n % 10) * (n % 10), 可通過.
        n = n / 10;
    }
    return sum;
}
```

203. Remove Linked List Elements, Easy

Remove all elements from a linked list of integers that have value ***val***.
Example
Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, ***val*** = 6
Return: 1 --> 2 --> 3 --> 4 --> 5
Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 簡單. 用一個 cur 指針, 若 cur.next 的值等於 val, 則將 cur.next 刪掉. 要用一個 falseHead. 注意相同值的 node 連在一起時之情況, 如 1-2-2-2-3-4.

History:
E1 幾分鍾寫好並通過, 而且代碼跟 William 幾乎是一模一樣的.
E2 試了好多次才通過.
E3 三分鍾寫好, 為了追 qiu speed, I did not check carefuly. 結果粗心范了兩個小錯誤, 一是我心裡想的是 if(cur.next.val == val) , 結果不小心寫成了 if(cur.val == val). The second was that after deleting a node, I moved pre to its next while I actually should not move it because of possible multiple nodes with value val. E3 的代碼方法跟 E1(so William)一樣, 但代碼更簡單, 以後用 E3 的代碼.

E1 的代碼(不用) : 用 pre 和 cur 兩個指針, cur 指向當前要刪的, pre 是它前一個.

```
public static ListNode removeElements(ListNode head, int val) {
  ListNode falseHead = new ListNode(0);
  falseHead.next = head;
  ListNode pre = falseHead;
  ListNode cur = head;
```

```
 while(cur != null) {
  if(cur.val == val)
   pre.next = cur.next;
  else
   pre = pre.next;

  cur = cur.next;
 }

 return falseHead.next;
}
```

E3 的代碼: 用一個 cur 指針, 若 cur.next 的值等於 val, 則將 cur.next 刪掉

```
    public ListNode removeElements(ListNode head, int val) {
        if(head == null) return head;
        ListNode fakeHead = new ListNode(0);
        fakeHead.next = head;
        ListNode cur = fakeHead;

        while(cur.next != null) {
            if(cur.next.val == val) cur.next = cur.next.next;
            else cur = cur.next;
        }

        return fakeHead.next;
    }
```

204. Count Primes, Easy

Description:
Count the number of prime numbers less than a non-negative number, ***n***.
Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.
Show Hint(我: Hint 太囉嗦, 還不如看我的 key, 所以就沒把 Hint copy 過來)
Subscribe to see which companies asked this question

Key: 我的 key 寫得比 leetcode 的 hint 要簡明多了, 不用看 leetcode 的 hint. 用 Sieve of Eratosthenes 方法. 將 2 的倍數, 3 的倍數...都 mark off. 剩下的數就全是 prime 了. 用一個 Hash Table(用數組實現)表示從 2 到 n 的數是否被 marked off (由於用的數組來記錄, 所以本題還可以輸出所有素數, 本題之所以不要求這樣做, 應該是因為 n 大時, 輸出會比較長, 我電腦上已驗證很長).

以下三點可避免重復運算:

第一，本算法之步驟為先將 2 的倍數 mark off, 再將 3 的倍數 mark off, 4 的倍數就不用 mark off 了, 因為包括在了 2 的倍數中, 再將 5 的倍數 mark off, ... 再將 p 的倍數 mark off, ...

第二, 當 p>sqrt(n)時, 是不用將 p 的倍數 mark off 的. 因為這個 p 的倍數 等於 p*某數, 這個「某數」是要小於 sqrt(n)的 (否則 p * 某數 就大於 n 了, 不是我們感興趣的), 即然這個「某數」小於 sqrt(n), 那麼 p*某數 就已經在之前作為「某數」的倍數被 marked off 了.

第三, 前面說將 2 的倍數 mark off, 3 的倍數 mark off, ...p 的倍數 mark off, ..., 這裡的 「p 的倍數」實際上不用從 2*p, 3*p...開始, 而是可以從 p^2 開始, 因為 2*p 已經在前面作為 2 的倍數被 mark off 掉了.

Convention: 最小的素數是 2, 而不是 1.

History:
E1 看了 leetcode 的 hint 後寫出的代碼能通過, 且跟 leetcode 的 hint 中的代碼幾乎一模一樣. 以下用我的代碼. leetcode 的 hint 中前面還有一個效率較低的代碼(即寫一個 isPrime(n)函數, 然後從 1 到 n 一個一個試), 我自已也寫出來了, 但 leetcode 中是超時. 我在我電腦上比較了這兩種方法, Sieve of Eratosthenes 方法明顯要快很多.
E2 幾分鍾寫好, 一次通過. 後來發現 E2 其實忘了寫 key 中的第二點, 但也通過了.
E3 改了好幾次才通過. E3 沒看 hint, 將方法全回憶出來了, 但 E3 最開始用的 Set, 結果超時, 不知道為甚麼, 所以又看了下 hint, 也沒多大幫助, 後來將 Set 改為數組, 就通過了. 在電腦上運行時, 用數組確實比 Set 快不少, 所以以後能用數組就用數組, 不要用 Set.

E1 改進後的代碼:

```
 public int countPrimes(int n) {
    boolean[] markedOff = new boolean[n];

    for(int p = 2; p * p < n; p++) { //E2 是寫成的<=, 也通過了, 因為多一些 p 的值的 後果是 最多就重復 mark 一下, 並不會影响最後結果.
       if(!markedOff[p]) { //E2 忘了寫這個 if, 但也通過了
          for(int m = p; m * p < n; m++) markedOff[m * p] = true;  //此句是 E3 的寫法, 更好懂.
       }
    }

    //Count number of primes in the table:
    int count = 0;

    for(int i = 2; i < n; i++) {
       if(!markedOff[i])  count++;
    }

    return count;
 }
```

205. Isomorphic Strings, Easy

Given two strings ***s*** and ***t***, determine if they are isomorphic.
Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.
All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.
For example,
Given "egg", "add", return true.
Given "foo", "bar", return false.
Given "paper", "title", return true.
Note:
You may assume both ***s*** and ***t*** have the same length.
Subscribe to see which companies asked this question

Key: 先看下面我說的, 再看題目. 用 Hash Table, 直接寫. 題目的意思就是字母的一一映射(每個字母可以映射到自己), 沒有別的意思. 注意 s="ab", t="cc"時之情況, 此時可用 if(oneSide(s, t) && oneSide(t, s)) return true; 題目將""和""(即兩個長度為 0 的 String)也定義為 isomorphic.

History:
E1 很快寫好並通過.
E2 居然沒做出來, 主要是沒想到 if(oneSide(s, t) && oneSide(t, s)).
E3 幾分鍾寫好, 結果第一次沒通過, 因為只考慮了 oneSide(s, t), 沒考慮 oneSide(t, s). 後來意識到後, 改後即通過. E3 表示 E2 好弱.

來自網路:

題目的意思就是找一个字母的一一映射，通过映射能够把其中一个单词变化成另一个单词，比如 paper title, p->t, a->i, e->l,r->e, 映射关系不能相交,即 不能出现一对多和多对一的关系,比如不能出现 a->b, a->c 或者 b->a, c->a.

我: 題目的意思就是字母的一一映射(每個字母可以映射到自己), 沒有別的意思. 題目中故弄玄虛, 尤其是"while preserving the order of characters", 不要看這句, 整個題目都可以不看, 不然想起來浪費時間.

我: 我的代碼很快寫好並通過, 方法跟 William 一樣, 但代碼比 William 的囉嗦點, 以下用參考 William 的代碼精簡後的我的代碼. William 的代碼是 Time ~ O(2N), Space ~ O(26), 我的應該也一樣.

```
public boolean isIsomorphic(String s, String t) {
 if(s == null || s.length() == 0) //Test case 中還真有["", ""]
  return true; //實踐表明, return false 通不過.

//以下幾句可以寫為一句: return oneSide(s, t) && oneSide(t, s), E3 就是這麼寫的.
 if(oneSide(s, t) && oneSide(t, s)) //此句很重要, 我自推第二遍就沒想到, 結果居然沒搞出來.
  return true;

 return false;
}

public boolean oneSide(String s, String t) {
 Map<Character, Character> map = new HashMap<>();

 for(int i = 0; i < s.length(); i++) {
  char cs = s.charAt(i), ct = t.charAt(i);

  if(!map.containsKey(cs))
   map.put(cs, ct);
  else {
   if(ct != map.get(cs))
    return false;
  }
 }
```

```
 return true;
}
```

206. Reverse Linked List, Easy

Reverse a singly linked list.
click to show more hints.
Hint:
A linked list can be reversed either iteratively or recursively. Could you implement both?
Subscribe to see which companies asked this question

Key: 簡單. Iterative 方法為: 把每個元素依次換到最前面. 即使 cur 固定在 head 處, 然後每次把 cur.next 移到最前面去. Recursive 方法為: 寫一個 recursive_reverse(ListNode current, ListNode next)之作用為: 反轉以 next 為頭的 list, 並將 current 加到反轉後的 list 末尾, 返回反轉後的 list 的新頭.

History:
E1 很快寫好, 一次通過. 以下用我 E1 的代碼. Code Ganker 沒有這道題的答案, William 的代碼沒細看, 看起來和我的差不多.

E2 幾分鍾寫好, 本來可以一次通過, 但忘了考慮輸入為 null 的情況, 添了才通過. E2 還用了一個 fakeHead, 其實沒必要. E2 還沒 E1 寫得好.

E3 的 iterative 代碼很快寫好, 一次通過. E3 不僅用了一個 fakeHead, 還多用了一個 last 指針. E3 也沒 E1 寫得好. E2 和 E3 怎麼反而沒 E1 爭氣. E3 的 recursive 的代碼也較快寫好, 一次通過, 方法跟 Code Ganker 的很不一樣, 在遞歸調用前要遍歷一次整個 list 找出它的末尾, 所以雖然可能總的時間複雜度跟 Code Ganker 差不多, 但看起來還是沒 Code Ganker 的好.

Tao: 我的代碼很快寫好並一次通過.
Tao: 234. Palindrome Linked List 中用了另一個反轉的算法, 即將 list 中每一個箭頭反向, 時間和空間複雜度跟本題一樣, 在那裡我將練習那個算法.

E1 的 iterative 代碼:

```
public static ListNode reverseList(ListNode head) {
 if(head == null || head.next == null)
  return head;

 ListNode cur = head;

 while(cur.next != null) {
  //以下兩句是刪除 cur.next
  ListNode temp = cur.next;
  cur.next = cur.next.next;

  //以下兩句是把 cur.next 添到最前面去
  temp.next = head;
  head = temp; //注意最後不要添一句 cur = cur.next, 否則不能把每個元素都換到前面去.
 }

 return head;
}
```

Code Ganker 在 143. Reorder List 的解答中提供了一個本題的遞歸解法, 可以看看(注意題目要求最好 iteratively 和 recursively 都會做):

Code Ganker (http://blog.csdn.net/linhuanmars/article/details/21503215):

这道题(第 143. Reorder List 題)看起来比较复杂，其实理清思路之后就都是链表常见的几个基本操作。这里我想多说一下 reverse 操作，因为这是链表最常见的操作。有时候在第一轮电面这种比较基础的面试中，可能会要求实现 reverse 操作，但是因为有点过于简单，面试官会要求递归和非递归都实现一下。上面的代码使用非递归的方式实现 reverse(我: 非遞歸法不比我上面的代碼好, 故沒 copy)。下面我们列举一下递归的代码，有兴趣的朋友可以看看哈。

Code Ganker 的 recursive 代碼:

```
public ListNode recursive_reverse(ListNode head) {
    if(head == null || head.next==null)
        return head;
    return recursive_reverse(head, head.next);
}
```

//recursive_reverse(ListNode current, ListNode next)之作用為: 反轉以 next 為頭的 list, 並將 current 加到反轉後的 list 末尾, 返回反轉後的 list 的新頭.

```
private ListNode recursive_reverse(ListNode current, ListNode next)
{
    if (next == null) return current;
    ListNode newHead = recursive_reverse(current.next, next.next);
    next.next = current;
    current.next = null;
    return newHead;
}
```

207. Course Schedule, Medium

http://wlcoding.blogspot.com/2015/05/course-schedule.html?view=sidebar

There are a total of *n* courses you have to take, labeled from `0` to `n - 1`.
Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]
Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?
For example:
2, [[1,0]]
There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.
2, [[1,0],[0,1]]
There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.
Note:
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
click to show more hints.
Hints:

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Subscribe to see which companies asked this question

Key: 題意就是判斷 一個有向圖 是否有環. 要掌握的是 DFS, William 的 DFS 詳細版代碼(及注釋)不看, 直接看後面的精簡版代碼(有足夠的注釋). BFS 代碼也要看看.

先構造一個 graph, 然後判斷此 graph 是否有 cycle. 即寫一個 Digraph 的 class, 它的 attribute 為:
int V: vertex 個數
int E: edge 個數
數組 adj: 表示節點的連接關係, adj 其每個元素為一個 list, adj[i]表示第 i 個 vertex 的所有 neighbors
boolean hasCyle
boolean[] onStack: 標記本次調用 dfs(v)所形成的軌跡, 用來判斷軌跡是否到碰到自己身體, 即是否有 cycle.
boolean[] marked: 標記歷史上所有形成的軌跡, 好讓以後的掃描不再訪問這些軌跡上的點, 避免重復運算.

在 Digraph 的 constructor 中初始化 V, E, adj.

在 Digraph 中, 寫一個 void dfs(v), 其作用為: 掃描所有 v 的兒子(即 v->w 中的 w), 對每一個俱體的兒子 w, 都一路深度往後代方向走出一條軌跡(貪吃蛇)來, 這條軌跡上所有點對應的 marked 和 onStack 都被設為 true, 當軌跡延伸到碰到自己的身體時(onStack 為 true), 就說明有一個環了, 就將 hasCycle 設為 true. 最後保護現場, 即將軌跡上所有點對應的 onStack 全都還原為 false, marked 仍保留為 true.

在 Digraph 中, 寫一個 boolean hasTopologicalOrder(), 其作用為對所有的點調用 dfs(v), 最後返回是否有 cycle.

History:
E1 直接看的答案. E1 本題和 210 題光看懂就花了一下午加一晚上的時間(看 210 題基本上沒花甚麼時間, 主要在看本題), 昨晚只睡了三個小時, 今晚又不能早睡, 後天還要開組會.
E2 直接看的答案. E2 表示 E1 寫的注釋很詳細好懂, 很 helpful, E1 的時間沒白花, E1 基本上是紅太陽.
E3 改了兩三次後通過. 方法跟 William 的類似, 也是 DFS, 用了 marked 和 onStack 兩個數組來標記是否已訪問, 但沒寫 Digraph 這麼個 class, 而是用的 Map 來表示有向圖的. 但本題 II 在此代碼的基础上改的, 能對題目中的三個例子都能給出正確結果, 但對大輸入時超時了, 不知道為何超時. 所以 E3 還是將本題的 Wiliam 的代碼默寫了一遍. 但也將 E3 的代碼帖在後面, 有空了可以看看.

我: William 給了 DFS 和 BFS 兩種方法, 要掌握的是 DFS 的方法 (簡化後的代碼, 但要先看看簡化前的代碼和我的注釋, 才看得懂簡化後的代碼), BFS 的方法也看看(E2 沒時間看, E3 也沒看).

本题判断是否可以按序完成课程，可以把课程 (vertex) 和次序 (edge) 看成一个 directed graph，如果存在 cycle，则无法完成课程。根据 directed graph 的性质：没有 cycle 等同于该 graph 存在 topological order，所以本题即转化为判断是否存在 topological order (即课程完成次序)。
首先需要实现一个 Digraph (我: 即 Directed graph)的 class，使用 adjacency lists: List<Integer>[] adj 存放 edges。
Topological sort 的实现有两种形式，可以用 DFS 或 BFS。本题是判断 topological order 是否存在，而不需要做 topological sort，所以 code 可稍简化。

1. DFS: Time ~ O(|V| + |E|), Space ~ O(|V|)
Topological sort (DFS) 的实现需要使用一个 Stack，当某个 vertex 所有的 adjacent vertices 都入栈之后，将其入栈，DFS 结束后所有元素出栈的顺序 (reverse DFS postorder) 即为 topological order。
但此处稍有不同：因为是 detect cycle，只要判断 topological order 是否存在，而并不需要得到具体的 order，所以不用使用 Stack(我: 以下第一個代碼還是用了 Stack 的)，只用一个 boolean[] onStack 记录入栈过的元素即可。另外，用 boolean[] marked 记录访问过的 vertices，如果之前访问过则不再访问。DFS 的具体规则如下：
•在搜索每条 path 的过程中将访问过的 vertices 的 marked[v] 和 onStack[v] 标记为 true，当该条 path 搜索完返回时将 onStack[v] 恢复成 false，而 marked[v] 不变；
•在搜索一条新 path 时 之前搜过的 path 上的 vertices 都不用再访问，只要看当前访问到的 vertex 和当前 path 上的某一点是否相连，如果相连则构成 cycle。

注意以下代碼全都是被包在 Solution 這個 class 裡面的(後面的簡化版和本題的 II 也一樣). 當然若這樣弄, 以下 public class Digraph 中的 public 應去掉.

詳細版代碼, 可以得到 cycle 中元素(不看, 直接看下面精簡版的, 那裡重新寫了注釋):

```
// directed graph class
//注意有向圖 class 的寫法(三個 memeber: V, E, adj)
public class Digraph {
   private int V;  //number of vertices. 一門課就是一個 vertex. 注意 V 是 vertex 的數量, 而不是 vertex.
   private int E; //number of edges. 一個 prerequisite pair 就是個 edge, 如題目中的"2, [[1,0]]"中的[0, 1]. 注意 E 是 edge 的數量, 而不是 edge.
   private List<Integer>[] adj;    // adjacency list //adj 是一個數組, 每個元素是一個 list. adj[i]是一個 list, 它表示第 i 個 vertex 的所有 neighbors

   public Digraph(int n, int[][] edges) {//注意後面調用 Digraph 時, edges 對應的實參就是題目中的 canFinish(int numCourses, int[][] prerequisites)中的 int[][] prerequisites, 即題目中的 '2, [[1,0],[0,1]]' 中的[[1,0],[0,1]]. 其俱體意思見後面.

      this.V = n;
      this.E = edges.length;
      adj = (List<Integer>[]) new List[V];
//以上是 array of lists 的寫法, 注意不能寫成 adj = new List<Integer>[V]. 詳見代碼後面的注. 我的電腦會對此句提示一些 note, 但在 leetcode 中可以通過.
      for (int i = 0; i < V; i++)
         adj[i] = new ArrayList<Integer>();  // initialize adj //此處要將 adj[i]建為一個 new ArrayList, 是因為下面的那個 for 中要直接調用 adj[v].add()
      for (int i = 0; i < E; i++) { //由此可知, 所有的課程都是用 0,1,...E-1 這些數字來表示的.
         int v = edges[i][1];//注意 edges 數組是輸入變量, 它是用來初始化 v 和 w(which 進一步用來初始化 adj). 若 edges[i][1] = b, edges[i][0] = a (即 edges[i]=[a, b], edges[i]表示第 i 個 edge), 則要先上了課 b, 才能上課 a. 所以本行和下行的意思是: 要先上了課 v, 才能上課 w. //注意後面調用 Digraph 時, edges 對應的實參就是題目中的 canFinish(int numCourses, int[][] prerequisites)中的 int[][] prerequisites, 即題目中的 '2, [[1,0],[0,1]]' 中的[[1,0],[0,1]]. 其俱體意思見後面.
         int w = edges[i][0];
         adj[v].add(w);  //符號: adj[v].add(w)表示要先上 v 再上 w, 劃出來就是 v->w, 稱為 v 是 w 的爸爸//add edge v->w//這是在用 edges[][]數組來構造 adj. adj 是一個數組 , 它的一個元素 adj[i]是一個 List, adj[i]表示從第 i 個 vertex 出發的 edge 們. 若 adj[v]這個 List 中有一個元素 w(當然還可能有其它元素), 則說明從 v
```

這個 vertex 出發, 有 v->w 這麼個 edge, 這個 edge 表示要先上了 v 這課, 才能上 w 這課, 即先上爸再上兒.

```
        }
    }
}

// directed cycle class
public class DirectedCycle {
    private Stack<Integer> cycle;
    //以下三行中定義的 marked, onStack, 和 edgeTo 的意思(作用)下面我會講
    private boolean[] marked;
    private boolean[] onStack;
    private int[] edgeTo;

    public DirectedCycle(Digraph G) {
        marked = new boolean[G.V];
        onStack = new boolean[G.V];
        edgeTo = new int[G.V];
        for (int v = 0; v < G.V; v++) {//小寫的 v 是 marked[v]的指標
            if (!marked[v]) dfs(G, v);  // call DFS to find directed cycles
        }
    }

    public boolean hasCycle() {
        return cycle != null; //cycle 是前面定義的一個 Stack
    }

    // DFS
    //注意下面的 dfs(G, v)也是 DirectedCycle 的 member function
    //dfs(G, v)的作用為: 掃描所有 v 的兒子(即 v->w 中的 w), 對每一個俱體的兒子 w, 都一路深度往後代方向走, 走出一條軌跡來, 這條軌跡上每個人都對應的 marked 和 onStack 都被設為 true, 當軌跡延伸到碰到自己的身體時(如同貪吃蛇. 判斷軌跡是否碰到自己的身體的方法就是看當前那個點的 onStack 是否為 true, 若為 true, 則說明碰到了(當前那個點為交點), onStack 就是這個作用), 就說明有一個環了. 然後將這個環從 交點的爸爸 到 交點 開始逆序(逆序的意思我下面會說)放入 cycle 中. 最後將軌跡上每個人都對應的 onStack 全都還原為 false, marked 仍保留為 true. 這就是 dfs(G, v)的作用. marked 和 onStack 的作用是標記所形成的軌跡(貪吃蛇), 它們最開始一直是一樣的, 但最後 dfs 會將 onStack 都還原為 false, 而保留 marked 的值. 所以 onStack 的作用是標記本次調用 dfs(G, v)所形成的軌跡(貪吃蛇), 好讓本次的貪吃蛇知道甚麼時候碰到自己了. 而 marked 的作用是標記歷史上所有形成的軌跡(貪吃蛇), 好讓以後的掃描不再訪問這些軌跡(貪吃蛇)上的點, 以避免重復運算或死循環.


    private void dfs(Digraph G, int v) {
        marked[v] = true; //標記 v 已經被訪問到了, 即 v 為過 dfs 的參數
        onStack[v] = true;
        for (int w : G.adj[v]) {//w 是 v 的兒子, 即 v->w. 本 for 是掃描所有 v 的兒子
            if (cycle != null)  return; // stop when directed cycle found //即若之前已有其它 cycle, 則不再繼續.
            else if (!marked[w]) {      // recursion when new vertex found //注意是!marked[w], 而不是!
```

```
onStack[w], 因為歷史上所有形成的軌跡(貪吃蛇)身上的點都不能碰, 否則會重復運算或死循環.
            edgeTo[w] = v;//此句不是此 else 的主要任務, 要把注意力放到下句(dfs 那句). 此句的意思是記綠 v->w 的關係, 即 v 是 w 前一個, 此 edgeTo[]不是前面的 edges[][], 此 edgeTo[]是提供給後面的 for(x)循環用的
            dfs(G, w);
        } else if (onStack[w]){// trace back directed cycle //此時 onStack[w]=true, 表明出現一個環了, w 為交點(即貪吃蛇碰到自己的那一點).
            cycle = new Stack<Integer>();
            //以下幾句是將 w 所在的環放入 cycle 中, 從 v 開始, 逆序放, 即放 了 v 後, 再放他爸(for 中的 x = edgeTo[x]中的 edgeTo[x]就是 x 的爸爸, 因為前面的 edgeTo[w] = v 是記綠 v->w 的關係), 這樣逆序轉一圈, 最後放入 w. 注意此處的 v 不是最前面開始掃描的起點 v, 而是環的交點 w 的爸爸
            for (int x = v; x != w; x = edgeTo[x]) //本 for 只能管到下一句. 注意此處的 v 不是最前面開始掃描的起點 v, 而是環的交點 w 的爸爸
                cycle.push(x);
            cycle.push(w);
            cycle.push(v);//不要本句也能通過. 實際上應當將本句刪除, 因為前面 for 中首先就加了 v 的.
          }
       }//end of for(w)
       onStack[v] = false;
    }//end of dfs() function
}

public boolean canFinish(int numCourses, int[][] prerequisites) {
    Digraph G = new Digraph(numCourses, prerequisites);
    DirectedCycle finder = new DirectedCycle(G);
    return !finder.hasCycle();   // can finish courses if no directed cycle exists
} //最後別忘了, 我要背下來的是後面簡化版的代碼
```

我: 關於 array of list, 網上說的:

You can't create arrays of generic types, generally.

The reason is that the JVM has no way to check that only the right objects are put into it (with ArrayStoreExceptions), since the difference between List<String> and List<Integer> are nonexistent at runtime.

Of course, you can trick the compiler by using the raw type List or the unbound wildcard type List<?>, and then cast it (with a unchecked cast) to List<String>. But then it is your responsibility to put only List<String> in it and no other lists.

另一個按 William 的 array of list 的寫法的例子如下:

http://stackoverflow.com/questions/7810074/array-of-generic-lis

William:

上面的 code 不但能 detect cycle，还将 cycle 记录在了一个 Stack 中，具体实现是当发现 cycle (onStack[v] == true)，trace back cycle，把所有 cycle 上的 vertices 放入一个 Stack 中。如果把这部分删掉可精简 code 如下：

精簡版代碼(看它):

```
public class Digraph {
    //記憶: 一個 graph 的三要素: vertex 個數(V), edge 個數(E), 連接關係(adj, 即 neighbor 數組).
constructor 中就是初始化這三個. 剩下的變量 hasCycle, marked, onStack 都是和本題相關的, 它們在 constructor 之外處理的.
```

```
    private int V;
    private int E;
    private List<Integer>[] adj; //adj 是一個數組, 每個元素是一個 list. adj[i]表示第 i 個 vertex 的所有兒子.
    private boolean hasCycle;
    private boolean[] marked;
    private boolean[] onStack;

    //Constructor, 作用是初始化 V, E 和 adj.
    public Digraph(int n, int[][] edges) {//edges 對應的實參就[[1,0],[0,1]].
        this.V = n;
        this.E = edges.length;
        adj = (List<Integer>[]) new List[V]; //為何要這樣寫, 參見前面詳細版代碼的黃字
        for (int i = 0; i < V; i++)
            adj[i] = new ArrayList<Integer>(); //下面的 for 中要直接調用 adj[v].add()
        for (int i = 0; i < E; i++) {
            int v = edges[i][1];
            int w = edges[i][0];
            adj[v].add(w);  // add edge v->w, 表示要先上 v, 再上 w
        }
    }

    public boolean hasTopologicalOrder() {
        marked = new boolean[V];
        onStack = new boolean[V];
        for (int v = 0; v < V; v++)
            if (!marked[v]) dfs(v); // call DFS to find directed cycles
        return !hasCycle;
    }

    // DFS
    //dfs(v)的作用為: 掃描所有 v 的兒子(即 v->w 中的 w), 對每一個俱體的兒子 w, 都一路深度往後代方向走出一條軌跡(貪吃蛇)來, 這條軌跡上所有點對應的 marked 和 onStack 都被設為 true, 當軌跡延伸到碰到自己的身體時(onStack 為 true), 就說明有一個環了, 就將 hasCycle 設為 true. 最後保護現場, 即將軌跡上所有點對應的 onStack 全都還原為 false, marked 仍保留為 true. 所以 onStack 的作用是標記本次調用 dfs(v)所形成的軌跡. 而 marked 的作用是標記歷史上所有形成的軌跡, 好讓以後的掃描不再訪問這些軌跡上的點, 以避免重復運算或死循環.
    private void dfs(int v) {
        marked[v] = true;
        onStack[v] = true;
        for (int w : adj[v]) {
            if (hasCycle)           return;
            else if (onStack[w])    hasCycle = true;//我交換了本句和下句順序, 更好理解, 可通過.
            else if (!marked[w])    dfs(w);
        }
        onStack[v] = false;
    }
}
```

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    Digraph G = new Digraph(numCourses, prerequisites);
    return G.hasTopologicalOrder(); // can finish courses if no directed cycle exists
}
```

2. BFS: Time ~ O(|V| + |E|), Space ~ O(|V|)

我: 本方法用一個 無cycle的圖 和 有cycle的圖 的例子, 一想即知.

Topological sort (BFS) 的实现需要使用一个 Queue 存放入度为 0 的 vertices，vertex 的入度 (指向它的 vertices 的数量) 用 int[] indegree 记录。

我: indegree[v]即有多少個vertex指向v, 注意a->b->c中c的indegree為1而不是2(即indegree不是level的意思), 而a->c, b->c中c的indegree為2.

从 indegree 为 0 的 vertices 开始，将其出栈(我: 本段中的棧指Queue)，然后将其所有 adjacent vertices 入栈，并将它们的 indegree 都减 1 (即在 graph 中删除之前的 vertex)，重复此过程直至 Queue 为空。

此时如果 graph 中所有的 vertices 都遍历过，则不存在 cycle；如果有 vertices 没有入栈 (最后的 indegree 不为 0)， 则存在 cycle；具体实现用一个 count 记录出栈的 vertex 的数量，判断最后的 count 是否等于 vertex 的总数。
注意：如果要得到 topological order，即为 Queue 出栈的顺序。

```
public class Digraph {
    private int V;
    private int E;
    private List<Integer>[] adj;
    private int[] indegree;
    private Queue<Integer> q;   // store all vertices with 0 indegree

    public Digraph(int n, int[][] edges) {
        this.V = n;
        this.E = edges.length;
        adj = (List<Integer>[]) new List[V];
        for (int i = 0; i < V; i++)
            adj[i] = new ArrayList<Integer>();
        indegree = new int[V];
        for (int i = 0; i < E; i++) {
            int v = edges[i][1];
            int w = edges[i][0];
            adj[v].add(w);  // add edge v->w
            indegree[w]++;  // update indegree of w
        }
    }

    // BFS
    public boolean hasTopologicalOrder() {
        q = new LinkedList<Integer>(); //q 是一個 queue
        for (int i = 0; i < V; i++)
```

```
            if (indegree[i] == 0)   q.add(i);   // add all vertices with 0 indegree

        int count = 0;  // number of vertices polled from queue
        while (!q.isEmpty()) {
            int v = q.poll();
            count++;
            // 以下的 for 是將 v 的所有兒子的 indegree 減少 1, 若減少後等於 0, 則加入 queue 中.
            for (int w : adj[v]) {
                indegree[w]--; //原代碼這兩句是用的--i 形式, 我將其分為了兩句, 更好懂, 可通過.
                if (indegree[w] == 0) q.add(w); // add vertex if its indegree is 0
            }
        }

        if (count < V)  return false;
        else    return true;    // has topological order if all vertices polled from queue
    }
}

public boolean canFinish(int numCourses, int[][] prerequisites) {
    Digraph G = new Digraph(numCourses, prerequisites);
    return G.hasTopologicalOrder(); // can finish courses if no directed cycle (has topological order)
}
```

E3 的代碼:

```
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        if(numCourses <= 0 || prerequisites == null || prerequisites.length == 0) return true;
        Map<Integer, List<Integer>> map = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < prerequisites.length; i++) {
            if(!map.containsKey(prerequisites[i][0]))
                map.put(prerequisites[i][0], new ArrayList<Integer>());
        }

        for(int i = 0; i < prerequisites.length; i++) {
            map.get(prerequisites[i][0]).add(prerequisites[i][1]); //key 為兒子, value 為爸爸們 (key 為爸爸, value
為兒子們也能通過)是為了跟本題的 II 一致.
        }

        boolean[] markedGlobal = new boolean[numCourses];
        boolean[] markedLocal = new boolean[numCourses];

        for(int course = 0; course < numCourses; course++) {
            if(!dfs(course, map, markedGlobal, markedLocal)) return false;
        }

        return true;
    }

    private boolean dfs(int cur, Map<Integer, List<Integer>> map, boolean[] markedGlobal, boolean[]
```

```
markedLocal) {
        if(markedLocal[cur]) return false;
        if(markedGlobal[cur]) return true;
        if(!map.containsKey(cur)) return true;

        markedGlobal[cur] = true;
        markedLocal[cur] = true;

        for(int i = 0; i < map.get(cur).size(); i++) {
            int next = map.get(cur).get(i);
            if(!dfs(next, map, markedGlobal, markedLocal)) return false;
        }

        markedLocal[cur] = false;

        return true;
    }
```

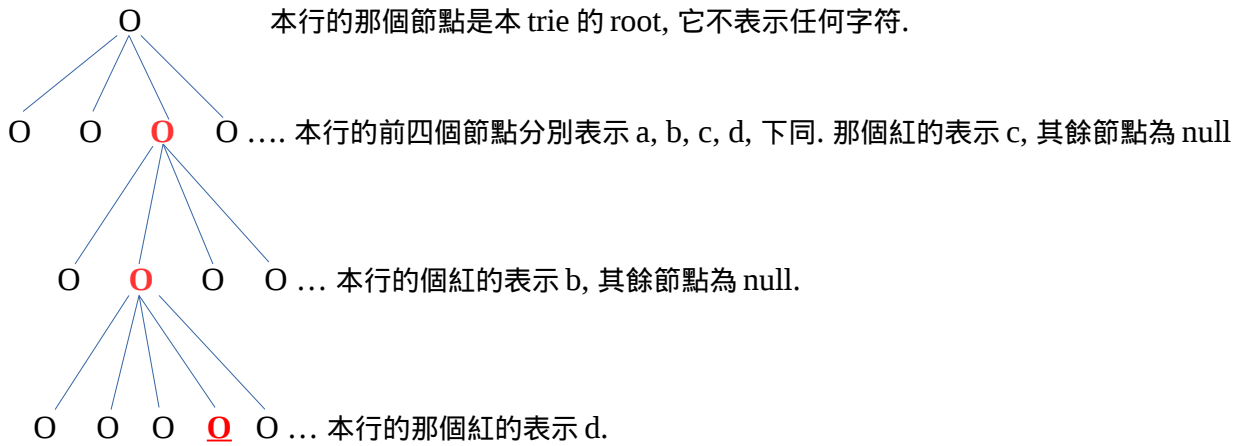
208. Implement Trie (Prefix Tree), Medium

http://wlcoding.blogspot.com/2015/05/implement-trie-prefix-tree.html?view=sidebar

Implement a trie with insert, search, and startsWith methods.
Note:
You may assume that all inputs are consist of lowercase letters a-z.
Subscribe to see which companies asked this question

Key: 遞歸. 寫一個 put(TrieNode x, String str, int d)函數, 作用是將 str 的「以 str[d]開始的子 string」 插入以 x 為 root 的 trie 中並返回 x, 若中途遇到 trie 中沒有的字符, 則為該字符自建一個 node. 若 d==str.length(), 則說明 str 已插完, 返回 x. put 通過遞歸來實現: x.next[c] = put(x.next[c], str, d + 1). 再寫一個 get(TrieNode x, **String** str, **int** d)函數, 作用是返回 str 的「以 str[d]開始的子 string」 的最後一個字符在 trie(以 x 為 root)中對應的那個 node . get 可以遞歸調用它自己. 其餘的都可以通過調用 put 和 get 來實現. insert 調用 put, search 調用 get, startsWith 調用 get, 看其返回值是否為 null.

例如, 加入"cbd"這個單詞後的 trie 為如下所示, 其中紅色的節點分別表示 c, b, d, 加下劃線的節點的 eow 為 true. 注意不是所有節點都有子節點的(意思是有的節點的 next 為 null), 若下圖中某個節點無子節點, 即表示它的 next 為 null.



History:
E1 直接看的答案. DS^N 面試問到此題, 然後跪了.
E2 在晚上很晚時寫的, 半個腦袋已經睡著了, 而且 key 也沒這麼詳細, 代碼仍然通過, NB, 代碼很多地方跟 William 的不同, 比 William 的囉嗦.
E3 改了幾次後通過, 方法和 William 的差不多, 雖然代碼沒有 Wiliam 的簡明, 但由於完全是我自己想出來的, 所以比較好懂和好記憶. 但我做 211. Add and Search Word 題時, 在本題 E3 代碼基础上改的, 改的方法跟 William 一樣, 在我電腦上也能得出正確結果, 但在 OJ 上大輸入時總是超時, 一直沒想出超時的原因, 所以以後本題和 211 題還是用 William 的代碼, E3 默寫過本題的 William 代碼.

Tao: http://taop.marchtea.com/06.09.html 對 Trie 的講解很好, 將最重要的部分 copy 到下面(其餘部份可以不看):

Trie 树（字典树）

方法介绍

1.1、什么是 Trie 树

Trie 树，即字典树，又称单词查找树或键树，是一种树形结构。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是最大限度地减少无谓的字符串比较，查询效率比较高。

Trie 的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

它有 3 个基本性质：

1.根节点不包含字符，除根节点外每一个节点都只包含一个字符。
2.从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
3.每个节点的所有子节点包含的字符都不相同。

1.2、树的构建

咱们先来看一个问题：假如现在给你 10 万个长度不超过 10 的单词，对于每一个单词，我们要判断它出没出现过，如果出现了，求第一次出现在第几个位置。对于这个问题，我们该怎么解决呢？

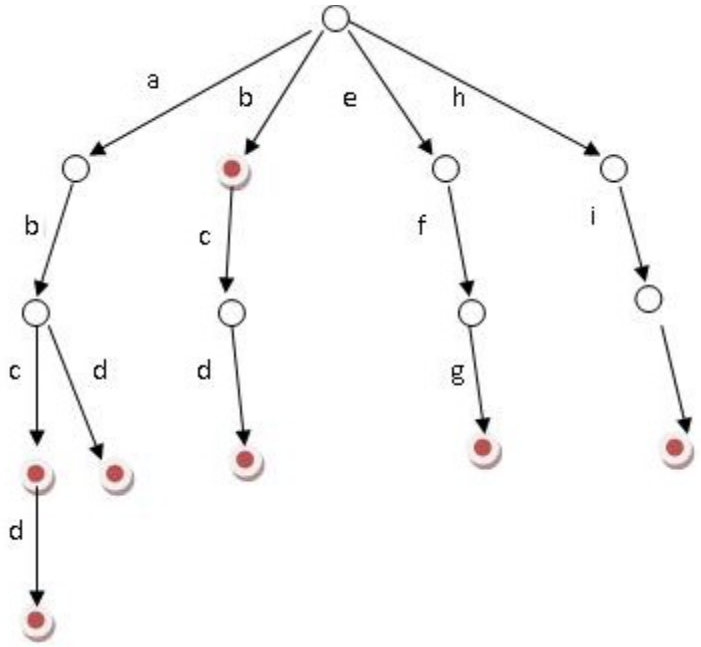
如果我们用最傻的方法，对于每一个单词，我们都要去查找它前面的单词中是否有它。那么这个算法的复杂度就是 O(n^2)。显然对于 10 万的范围难以接受。

换个思路想：

•假设我要查询的单词是 abcd，那么在它前面的单词中，以 b，c，d，f 之类开头的显然不必考虑，而只要找以 a 开头的中是否存在 abcd 就可以了。

•同样的，在以 a 开头中的单词中，我们只要考虑以 b 作为第二个字母的，一次次缩小范围和提高针对性，这样一个树的模型就渐渐清晰了。

即如果现在有 b，abc，abd，bcd，abcd，efg，hii 这 6 个单词，我们可以构建一棵如下图所示的树：



如上图所示，对于每一个节点，从根遍历到他的过程就是一个单词，如果这个节点被标记为红色，就表示这个单词存在，否则不存在。

那么，对于一个单词，只要顺着他从根走到对应的节点，再看这个节点是否被标记为红色就可以知道它是否出现过了。把这个节点标记为红色，就相当于插入了这个单词。

这样一来我们查询和插入可以一起完成，所用时间仅仅为单词长度（在这个例子中，便是 10）。这就是一棵 trie 树。

我们可以看到，trie 树每一层的节点数是 26^i 级别的。所以为了节省空间，我们还可以用动态链表，或者用数组来模拟动态。而空间的花费，不会超过单词数×单词长度。

William:

Time ~ O(L), Space ~ O(RN) to O(RNL) where R = 26, N is number of words and L is word length
Implement an R-way trie:
•String ST: key - word, value - boolean eow
•DFS methods: put() for **insert**, and get() for **search** (check eow) and **startsWith** (no need to check eow).
Note: variables in class TrieNode should be public.

```
class TrieNode {
    boolean eow; // end of word
    TrieNode[] next; //注意 next 是數組
```

```
    // Initialize your data structure here.
    public TrieNode() {
        eow = false;
        next = new TrieNode[26]; //注意此句执行後, next 數組中每個元素都是 null. From online:
```
Everything in Java not explicitly set to something, is initialized to a zero value.

- For references (anything that holds an object) that is `null`.
- For int/short/byte that is a 0.
- For float/double that is a 0.0
- For booleans that is a false.

```
    }
}


public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode(); //不要這句也能通過
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        root = put(root, word, 0); //本句能不能寫成 put(root, word, 0), 而不要"root="? 實踐表明答案是, 若前面 constructor 中有 root = new TrieNode()一句, 則本句可以寫成 put(root, word, 0); 若前面 constructor 沒有 root = new TrieNode()一句, 則本句不能寫成 put(root, word, 0). 這是因為若 constructor 中有那句, 則 root 是一個確定的 object, 用 put 對它进行操作即可, 若 constructor 中沒有那句, 則 root 為 null, 它不是一個確定的 object, 而是一個 unversal 的 null, 則 put 對它的操作並非是在我們想像的那個 root 上, 所以操作完了後要將它重新賦給 root. 後面的 x.next[c] = put(x.next[c], str, d + 1)一句也是這個原因.
    }

//put(x, str, d)的作用是: 將 str 的「以 str[d]開始的子 string」插入以 x 為 root 的 trie 中並返回 x, 若中途遇到 trie 中沒有的字符, 則為該字符自建一個 node. 若 d==str.length(), 則說明 str 已插完, 返回 x.
//注意每新建一個 TrieNode x, constructor 自動就給 x 弄了 26 個兒子(但每個兒子都是 null). 而 put(x, str, d)的作用就是將 x 的某個兒子賦為非 null, 哪個兒子這麼幸運? 就是第 str.charAt(d) - 'a'個兒子這麼幸運.這個兒子被賦的甚麼東西? 賦的還是它自己(x 那個幸運兒子), 但是全新的自我了, 因為它的後代也已經像它一樣被弄了(即 str 的「以 str[d+1]開始的子 string」被 插入了以 x 那個幸運兒子為 root 的 trie 中)
    private TrieNode put(TrieNode x, String str, int d) {
        if (x == null)  x = new TrieNode(); //此話重要. 它的意思是, 要是 trie 的 path 中無 str 中的某個字母(即以下 put 的第一個參量 x.next[c]==null), 則自己建這麼一個 node.
        if (d == str.length()) {//注意 trie 的 root 不包含字符, 所以當 d==str.length()時即弄完了.
            x.eow = true;
            return x;
        }
        int c = str.charAt(d) - 'a';
        x.next[c] = put(x.next[c], str, d + 1);
        return x;
    }

    // Returns if the word is in the trie.
```

```
    public boolean search(String word) {
        TrieNode x = get(root, word, 0);
        if (x == null)  return false;
        else            return x.eow;
    }
//以下 get 函數的作用是返回 str 的「以 str[d]開始的子 string」 的最後一個字符在 trie(以 x 為 root)中對應的那個 node
    private TrieNode get(TrieNode x, String str, int d) {
        if (x == null)  return null;
        if (d == str.length())  return x;
        int c = str.charAt(d) - 'a';
        return get(x.next[c], str, d + 1);
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        TrieNode x = get(root, prefix, 0);
        if (x == null)  return false;
        else            return true;
    }
}

// Your Trie object will be instantiated and called as such:
// Trie trie = new Trie();
// trie.insert("somestring");
// trie.search("key");
```

E3 的代碼:

```
class TrieNode {
   public boolean eow; //表示該節點對應的字符是否為 end of word,
   public boolean addedThisLetter; //表示該節對應的字符是否已被加入到該 trie 中(即是否在某個單詞中包含該字符, 即在加某個單詞到 trie 中時是否訪問過該節點),
   public TrieNode[] next; //即該節點的子節點組成的數組, 共 26 個, 根据數組的指標即可知對應的是哪個字符, 比如 next[0]對應的是 a, 所以不再需要一個 val member.
   public TrieNode() {}
}

public class Trie {
   private TrieNode root;

   public Trie() {
      root = new TrieNode();
   }

   public void insert(String word) {
      insertHelper(root, word, 0);
   }

   public boolean search(String word) {
      return searchHelper(root, word, 0);
   }
```

```
    public boolean startsWith(String prefix) {
        return startsWithHelper(root, prefix, 0);
    }


    //void insertHelper(TrieNode cur, String word, int d)之作用:
    //將 word[d, ...]這個 substring 放入以 cur 為根的 trie 中. word[d]這個字符是放入 cur.next[d]中的.
    private void insertHelper(TrieNode cur, String word, int d) {
        if(cur.next == null) {
            cur.next = new TrieNode[26];
            for(int i = 0; i < 26; i++) cur.next[i] = new TrieNode();
        }

        if(d >= word.length()) return;
        int index = word.charAt(d) - 'a';
        cur.next[index].addedThisLetter = true;
        if(d == word.length() - 1) cur.next[index].eow = true;
        insertHelper(cur.next[index], word, d + 1);
    }

    //boolean searchHelper(TrieNode cur, String word, int d)之作用:
    //返回 word[d, ...]這個 substring 是否是 以 cur 為根的 trie 中的一個 word, 即 word[d, ...]的最後一個字符在該 trie 中對應的節點的 eow 是否為 true. 注意 word[d]這個字符是放入 cur.next[d]中的.
    private boolean searchHelper(TrieNode cur, String word, int d) {
        if(cur.next == null) return false;
        int index = word.charAt(d) - 'a';
        if(d == word.length() - 1 && !cur.next[index].eow) return false;
        if(d == word.length() - 1 && cur.next[index].eow) return true;
        return searchHelper(cur.next[index], word, d + 1);
    }

    //boolean startsWithHelper(TrieNode cur, String word, int d)之作用:
    //返回 word[d, ....]這個 substring 是否在 以 cur 為根的 trie 中, 即 word[d, ...]中所有字符在該 trie 中所對應的節點的 addedThisLetter 是否為 true. 注意 word[d]這個字符是放入 cur.next[d]中的.
    private boolean startsWithHelper(TrieNode cur, String word, int d) {
        if(cur.next == null) return false;
        int index = word.charAt(d) - 'a';
        if(!cur.next[index].addedThisLetter) return false;
        if(d == word.length() - 1) return true;
        return startsWithHelper(cur.next[index], word, d + 1);
    }
}
```

209. Minimum Size Subarray Sum, Medium

http://wlcoding.blogspot.com/2015/05/minimum-size-subarray-sum.html?view=sidebar

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a subarray of

which the sum ≥ **s**. If there isn't one, return 0 instead.
For example, given the array [2,3,1,2,4,3] and s = 7,
the subarray [4,3] has the minimal length under the problem constraint.
click to show more practice (tao: already clicked).
More practice:
If you have figured out the *O*(*n*) solution, try coding another solution of which the time complexity is *O*(*n* log *n*).
Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 用 Two Pointers. 右指針向右掃. 當 sum >= s 時, 左指針向右移 直到 sum < s 為止. 途中記錄最短長度.

Corner case: {1, 1}, 3

History:

E1 直接看的答案.

E2 改了很多次後通過, 且代碼比 William 的囉嗦.

E3 改了幾次後通過. E3 用的 two pointers, 沒看 binary search 的代碼.

我: William 給了 Two Pointers 和 Binary Search 兩種方法, Two Pointers 時間和空間複雜度都要低些, 且代碼也要簡單些, 故要掌握 Two Pointer 的方法. 但題目中要要求 O(n log n)的方法也要會做, 所以 Binary Search 的方法也要看看(E2 沒時間看).

1. Two Pointers: Time ~ O(N), Space ~ O(1)
前指针向前扫，当 sum >= s 时向前移动后指针 prev (shrink the subarray)，只到 sum < s 为止，并记录最短的 subarray 的长度。

```
public int minSubArrayLen(int s, int[] nums) {
    int prev = 0, sum = 0, len = Integer.MAX_VALUE;
    for (int i = 0; i < nums.length; i++) {
     //以下就是求「以 i 為 end 的」「和大於等於 s 的 subarray」的最小長度. 即[prev, prev+1, … i]中 prev 的最右時之 subarray. 所有 最小長度 中的最小值當然就是本題答案了.
        sum += nums[i];
        while (sum >= s) {
            len = Math.min(len, i - prev + 1);
            sum -= nums[prev++];//注意 prev++返回的是加 1 前的 prev
        }
    }
    return len == Integer.MAX_VALUE ? 0 : len; //0 is for the corner case: {1,1}, 3
}
```

2. Binary Search: Time ~ O(NlogN), Space ~ O(N)

我: 由於我不太習慣 William 的二分法寫法, 所以不用他的代碼, 以下用 Jyuan 的代碼, 方法跟 William 是一樣的.

Jyuan (我: 以下的 Chinglish 有的地方沒表述清楚, 我作了小修改):

http://www.jyuan92.com/blog/leetcode-minimum-size-subarray-sum/

Another solution is by using binary search algorithm, we initial another array (我: sum) with

the length `nums.length + 1`, and `sum[i]` is equal to the sum from `nums[0]` to `num[i – 1].` When we meet an element in the array “sum”, for example `sum[i]`, if sum[i] is larger or equal to `s`, then we find the largest element (let's say sum[j]) that is 小於等於 `sum[i] - target` by using binary search. 我: 之所以要找這樣的 sum[j], 是因為 num[j, j+1, ... i-1]的元素和即為 sum[i] – sum[j]. 而我們又有 sum[j] <= sum[i] – target, 即 sum[i] – sum[j] >= target. 注意 i 是定的, j 是變的. 定出 j 後, then we calculate the length of the subarray num[j, j+1, ... i-1], which is `i - j`, 然後變 i, 每一個 i 都對應這樣一個 length, 然後在所有 length 中找最小一個, 就是結果

Time Complexity: `O(nlogn).`

```
public int minSubArrayLen(int s, int[] nums) {


if (null == nums || nums.length == 0) {
    return 0;
}
int min = nums.length + 1;
int[] sum = new int[nums.length + 1];
for (int i = 0; i < nums.length; i++) {
    sum[i + 1] = sum[i] + nums[i];
    if (sum[i + 1] >= s) {
        int left = binarySearch(sum, sum[i + 1] - s, 0, i + 1); //注意 i+1 即為上面解說中的 sum[j] <= sum[i] – target 中的 i
        min = Math.min(min, i + 1 - left); //注意 i+1 即為上面解說中的 sum[j] <= sum[i] – target 中的 i
    }
}
return min == nums.length + 1 ? 0 : min;
}
```

//binarySearch(sum, target, start, end)的作用為: 在 sum[start, start+1...end]中找出 最大的 小於等於 target 的元素, 返回那個元素的 index. 注意 sum 中的元素是遞增的. 原代碼中 binarySearch 的第一個參數(sum)取的名字為 nums, 很 misleading, 我將其改為了 sum, 改後可通過.

```
private int binarySearch(int[] sum, int target, int start, int end) {
    while (start + 1 < end) {//while 的條件為何是 start + 1 < end? 下面我會說
        int mid = (start + end) / 2; //原代碼中為 int mid = start + (end - start) / 2, 我將其改為了這個簡明寫法, 另原代碼還有幾處我看起來不習慣的地方, 也作了小修改, 改後可通過.
        //注意此處沒有別的二分法中的 if(sum[mid] == target) return mid;
```

```
        if (target <= sum[mid]) {
            end = mid;
        } else {
            start = mid;
        }
    }
//上面 while 循環運行的條件為 start + 1 < end , 即 start <= end - 2, 即 start 和 end 中間至少夾了一個元素. 循環結束後, start 就和 end 緊挨(即 start = end – 1).  所以此時若 target >= sum[end], 就返回 end, 若 target < sum[end], 就返回 start(注意 sum[start]總是小於 target 的, 因為前面 while 中移動 start 的條件即為 target>sum[mid], 然後就將 start 移到 mid 去了, 故總有 target>sum[start] )
    if (target >= sum[end]) {
        return end;
    } else {
        return start;
    }
}
```

210. Course Schedule II, Medium

http://wlcoding.blogspot.com/2015/05/course-schedule.html?view=sidebar

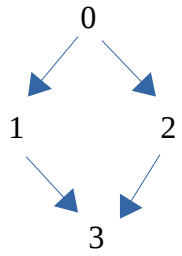
There are a total of *n* courses you have to take, labeled from `0` to `n - 1`.
Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`
Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.
There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.
For example:
2, [[1,0]]
There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is `[0,1]`
4, [[1,0],[2,0],[3,1],[3,2]]
There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is `[0,1,2,3]`. Another correct ordering is`[0,2,1,3]`.
Note:
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
click to show more hints.
Hints:

1. This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Subscribe to see which companies asked this question

Key: 對 207 題的簡化 DFS 代碼 增加四處就可以了(有兩處還是 trivial 的). 在 dfs 中產生軌迹的時候, 回溯時將 course 加入一個 stack 中(注意此時的順序是反的)(為何要這樣做? 可以看 dfs 函數前的那一大段黃字), 然後再在 stack 中讀入到另一個數組中就可以了.

本題中的 4 門課的例子的 graph 為:



History:
E1 直接看的答案.
E2 很快寫好並通過, E2 的代碼中 '從 stack 中讀入到另一個數組' 是在 findOrder 函數中做的, 而 William 是在 topologicalOrder 函數中做的, 我覺得還是 E2 的更自然好懂.
E3 是在本題 I 中我 E2 的代碼基础上改的, 我的代碼對題目中的三個例子都能給出正確結果, 但對大輸入時超時了, 不知道為何超時. E3 的方法跟 William 的類似, 見本題 I 的 History.

我: 本題 William 給了 DFS 和 BFS 兩種方法, 要掌握的是 DFS, 但 BFS 也要看看(E2 沒時間看).

本題是求 order，所以需要做 topological sort。只需在上面的代码中稍作改动即可。

1. DFS: Time ~ O(|V| + |E|), Space ~ O(|V|)
DFS 可以得到 postorder（即每次访问完某一 node 的所有 adjacent nodes 之后，才将该 node 入栈），而出栈的结果就是 reverse postorder = topological order。

```
public class Digraph {
   private int V, E; // V: vertex 個數, E: edge 個數
   private List<Integer>[] adj; //adj 是一個數組, 每個元素是一個 list. adj[i]表示第 i 個 vertex 的所有兒子.
   private boolean hasCycle;
   private boolean[] marked, onStack;
   private Stack<Integer> postorder;  //增加的(1/4, trivial): 定義 postorder

   //Constructor, 作用是初始化 V, E 和 adj.
   public Digraph(int n, int[][] edges) { //edges 對應的實參就[[1,0],[0,1]].
      this.V = n;
      this.E = edges.length;
      adj = (List<Integer>[]) new List[V];
      for (int i = 0; i < V; i++)
         adj[i] = new ArrayList<Integer>(); //下面的 for 中要直接調用 adj[v].add()
```

```
        for (int i = 0; i < E; i++) {
            int v = edges[i][1];
            int w = edges[i][0];
            adj[v].add(w);  // add edge v->w, 表示要先上 v, 再上 w
        }
    }

    public int[] topologicalOrder() {
        marked = new boolean[V];
        onStack = new boolean[V];
        postorder = new Stack<Integer>();  //增加的(2/4, trivial ): 新建一個 postorder

        for (int v = 0; v < V; v++) //for 只管下面一句
            if (!marked[v]) dfs(v); // call DFS to find directed cycles

        //增加的(3/4, 以下 if-else 都是): 將 postorder 中的結果提取出來.
        if (hasCycle)   return new int[0];
        else {
            int[] order = new int[V];
            for (int i = 0; i < V; i++)
                order[i] = postorder.pop();
            return order;
        }
    }

    // DFS
    //dfs(v)的作用為: 掃描所有 v 的兒子(即 v->w 中的 w), 對每一個俱體的兒子 w, 都一路深度往後代方向走出一條軌跡(貪吃蛇)來, 這條軌跡上所有點對應的 marked 和 onStack 都被設為 true, 當軌跡延伸到碰到自己的身體時(onStack 為 true), 就說明有一個環了, 就將 hasCycle 設為 true. 最後保護現場, 即將軌跡上所有點對應的 onStack 全都還原為 false, marked 仍保留為 true. 所以 onStack 的作用是標記本次調用 dfs(v)所形成的軌跡. 而 marked 的作用是標記歷史上所有形成的軌跡, 好讓以後的掃描不再訪問這些軌跡上的點, 以避免重復運算或死循環.
    //注意由 topologicalOrder 函數(或 E2 代碼中的 findOrder 函數)可知, 若有 cycle, 則直接就直接返回空數組了, 此時 dfs 函數並不起作用. 我們要認真考慮的是不含 cycle 時之情況(此時下面的 A 句是永遠不會執行的), 此時 dfs(v)的作用為: 除了俱有 207 題中的功能外, 還能將 v 和 v 的所有兒子按 '先兒後爸' 之順序放入 postorder 這個棧中(故 v 放在 postorder 的棧頂).
//若 v 並非真正的 應當最先上的課, 即以下這種情況: a->b->c->d->e(即 a 為應當最先上的課), 即若此時 v 為 c, 則 dfs 放入 postorder 這個棧中的東西就為(左為棧底, 右為棧頂): edc, 而 topologicalOrder 函數中用 for 循環對所有課 v 都調用 dfs(v), 故 for 中當 v 走到 a 時(即 v 為 a 時), 新加入 postorder 這個棧中的東西就為 ba(為何不加 c 了? 因為已經被 mark 了啊), 然後整個 postorder 棧就是 edcba.
    private void dfs(int v) {
        marked[v] = true;
        onStack[v] = true;

        for (int w : adj[v]) {
            if (hasCycle)           return; //A 句
            else if (onStack[w])    hasCycle = true;//我交換了本句和下句順序, 更好理解, 可通過.
            else if (!marked[w])    dfs(w);
```

```
        }

        postorder.push(v);  //增加的(4/4): 將 v 放入 postorder 中
        onStack[v] = false;
    }
}

public int[] findOrder(int numCourses, int[][] prerequisites) {
    Digraph G = new Digraph(numCourses, prerequisites);
    return G.topologicalOrder();
}
```

2. BFS: Time ~ O(|V| + |E|), Space ~ O(|V|)
BFS 中访问所有 indegree 为 0 的 node 的顺序（dequeue 的顺序）= topological order。
From Leetcode discussion: We observe that if a node has incoming edges, it has prerequisites. Therefore, the first few in the order must be those with no prerequisites, i.e. no incoming edges.

```
public class Digraph {
    private int V, E;
    private List<Integer>[] adj;
    private int[] indegree;
    private Queue<Integer> q;  // store all vertices with 0 indegree

    //Constructor, 作用是初始化 V, E, adj, indegree
    public Digraph(int n, int[][] edges) {
        this.V = n;
        this.E = edges.length;
        adj = (List<Integer>[]) new List[V];
        for (int i = 0; i < V; i++)
            adj[i] = new ArrayList<Integer>();
        indegree = new int[V];
        for (int i = 0; i < E; i++) {
            int v = edges[i][1];
            int w = edges[i][0];
            adj[v].add(w);  // add edge v->w, 表示要先上 v, 再上 w
            indegree[w]++;  // update indegree of w
        }
    }

    // BFS
    public int[] topologicalOrder() {
        q = new LinkedList<Integer>(); //q 是一個 queue
        for (int i = 0; i < V; i++) //此 for 只管以下一句
            if (indegree[i] == 0)   q.add(i);  // add all vertices with 0 indegree

        int count = 0;  // number of vertices polled from queue
        int[] order = new int[V];   // 增加的(1/4, trivial): 定義 order 數組, 用來放結果

        while (!q.isEmpty()) {
```

```
            int v = q.poll();
            order[count] = v; // 增加的(2/4): 將 poll 出來的 加入到 order 中
            count++; //原代碼上句和本句是寫到一起的, 我將其分為了兩句, 更好懂, 可通過.

            // 以下的 for 是將 v 的所有兒子的 indegree 減少 1, 若減少後等於 0, 則加入 queue 中.
            for (int w : adj[v]) {
               indegree[w]--; //原代碼這兩句是用的--i 形式, 我將其分為了兩句, 更好懂, 可通過.
               if (indegree[w] == 0) q.add(w); // add vertex if its indegree is 0
            }
         } // end of while

         if (count < V)  return new int[0];  // 增加的(3/4): 返回空數組
         else   return order;    // 增加的(4/4): 返回 order 數組
      } // end of function topologicalOrder()

} // end of class Digraph

public int[] findOrder(int numCourses, int[][] prerequisites) {
    Digraph G = new Digraph(numCourses, prerequisites);
    return G.topologicalOrder();
}
```

211. Add and Search Word - Data structure design, Medium

http://wlcoding.blogspot.com/2015/05/add-and-search-word-data-structure.html?view=sidebar

Design a data structure that supports the following two operations:
void addWord(word)
bool search(word)
search(word) can search a literal word or a regular expression string containing only letters `a-z` or `.`
A `.` means it can represent any one letter.
For example:
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
Note:
You may assume that all words are consist of lowercase letters `a-z`.
click to show hint.
You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.
Subscribe to see which companies asked this question

Key: 本題代碼跟 208. Implement Trie (Prefix Tree)唯一的區別就是 get 函數中, 若 str.charAt(d)=='.'時, 要將 c=0,

1..R-1 都放入 get(x.next[c], str, d+1)中弄一下, 而此時面對多個可以 return 的結點, 怎麼辦? 答: 只要找到一個 eow 的結點, 就返回它, 因為只有 search 會調用 get. 當然, 本題中沒有 startsWith 函數. TrieNode class 是寫在 WordDictionary 這個 class 裡面的. 在 WordDictionary 這個 class 中還要定義一個 root, 以及 WordDictionary 的 constructor.

History:
E1 直接看的答案.
E2 通過, 而且 'get 中面對多個可以 return 的結點時，只要找到一個 eow 的結點, 就返回它' 的想法與 William 不謀而合.
E3 在 208. Implement Trie 的 E3 代碼基础上改的, 改的方法跟 William 一樣, 在我電腦上也能得出正確結果, 但在 OJ 上大輸入時總是超時, 一直沒想出超時的原因, 所以以後本題和 208 題還是用 William 的代碼.

我: William 給出了兩種方法, 只看方法 1. 方法 2 超時, 不看.

William Tag: DFS.

William:
1. Add: Time ~ O(L), Search: Time ~ O(L) to O(RL); Space ~ O(RN) to O(RNL) where R = 26
Use tire to store dictionary: addWord() and search() are similar to insert() and search() in Trie().
Only difference is to implement the regular expression '.' here:
•a slight modification on search(): if char c == '.', check R branches; otherwise only check the cth branch.

//注意以下哪些 member 是 private, 哪些是 public: 只有 constructor 和用户要用的 addWord 函數和 search 函數是 public. 其餘都是 private, 包括 member 變量和 member class (inner class?).

```
public class WordDictionary {
    private final static int R = 26;    // lowercase letters a through z
    private TrieNode root;  // root of trie

    // R-way trie node
    private class TrieNode {
        boolean eow;
        TrieNode[] next;//注意 next 是數組
        public TrieNode() {//這一個是 public, 應該是語法要求
            eow = false;
            next = new TrieNode[R];
        }
    }

    //constructor
    public WordDictionary() {
        // root = new TrieNode();//若此句不注釋掉, 也能通過
    }

    // Adds a word into the data structure.
    public void addWord(String word) {
        root = put(root, word, 0);
    }

    private TrieNode put(TrieNode x, String str, int d) {
        if (x == null)  x = new TrieNode();
        if (d == str.length()) {
            x.eow = true;
```

```
            return x;
        }
        int c = str.charAt(d) - 'a';
        x.next[c] = put(x.next[c], str, d + 1);
        return x;
    }

    // Returns if the word is in the data structure. A word could
    // contain the dot character '.' to represent any one letter.
    public boolean search(String word) {
        TrieNode x = get(root, word, 0);
        if (x == null)  return false;
        else            return x.eow;
    }

    private TrieNode get(TrieNode x, String str, int d) {
        if (x == null)  return null;
        if (d == str.length())  return x;
        if (str.charAt(d) == '.') {         // '.' represents any letter
            for (int c = 0; c < R; c++) {   // search R branches
                 TrieNode y = get(x.next[c], str, d + 1); //注意 y 是最後一個字符對應的
node.
                if (y != null && y.eow)  return y; //E2 的想法與此不謀而合
            }
            return null;
        } else {                            // search cth branch
            int c = str.charAt(d) - 'a';
            return get(x.next[c], str, d + 1);
        }
    }
}

// Your WordDictionary object will be instantiated and called as such:
// WordDictionary wordDictionary = new WordDictionary();
// wordDictionary.addWord("word");
// wordDictionary.search("pattern");
```

2. Add: Time ~ O(2^L), Search: Time ~ O(L); Space ~ O(2^L) where R = 27 Exceed Time Limit!
Another idea is to add additional last branch for '.' in the trie, and store every char in the last branch besides the cth branch. Although it saves time for search(), it will lead to exponential time for addWord().

(我: 由於本方法超時, 故代碼已省)

212. Word Search II, Hard

http://wlcoding.blogspot.com/2015/03/word-search.html?view=sidebar

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.
click to show hint.
You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?
If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.
Subscribe to see which companies asked this question

Key: 遞歸回溯. 要在 Solution 中寫一個 class Node, 其 member 為 String val 和 Node[] next(無 eow member), val 的作用為: 當 trie 中某個 Node 為 eow 時, 這個 Node 的 val 即為這個 word, 否則 val 為空(在 put 函數中實現). 還要在 Solution 中定義一個全局的 Node root. 再加上 insert(String key)和 put(Node x, String key, int d), 這樣就形成了一個 trie. 最初將所有 words 都加入到 trie 中. 然後要做的就是用 dfs 函數查看 'board[i][j]在 trie 中對應的 Node' 是否為 eow, 若是, 則將其對應的 word 加入到 list 中, 更俱體的:

寫一個 void dfs(char[][] board, boolean[][] visited, int i, int j, Node x, List<String> list), 其作用為: 查看 x.next 中的第 board[i][j]-'a'個 Node, 若其 val 非 null(當 trie 中某個 Node 為 eow 時, 這個 Node 的 val 即為這個 word, 否則 val 為空), 則將這個 val 加入到 list 中. dfs 和 79 題一樣, 要遞歸調用自己來考查 ij 的上下左右, 對 visited 的回溯也跟 79 題一樣.

History:
E1 直接看答案.
E2 沒做出來, 主要是因為 E1 寫的 key 太簡略了, 基本上甚麼都沒說, E2 重新寫了 key.
E3 的代碼對題目中的例子和幾個 OJ 的例子能給出正確結果, 在但 OJ 上超時了, 改了好久還是超時. E3 的方法大多數跟 William 一樣, 不知道為何超時, 可能是因為 William 的 TrieNode 節點弄了個 val 表示對應的單詞, 在 dfs 函數中直接加 val. 而我 E3 沒弄 val, 而是用的一個 StringBuilder 沿路加的.

Tao: 先做 208. Implement Trie (Prefix Tree)和 79. Word Search

DFS + Trie: Time ~ O(?), Space ~ O(?)
此题是 Boggle Game 的一种（这里只能上下左右，不能斜角走），因为需要对每一个格点进行 DFS，所以必须用有效的 pruning 实现 backtracking 来提高速度，否则程序无法通过 large datasets。
解决方法：将 dictionary 中所有的 word 存放在 Trie 中，在 DFS 中搜索到当前的 path 可以看成一个 prefix，dictionary 中不存在这样的 prefix 即停止搜索。

Tao: 原代碼有幾處比較 redundant, 我作了以下改进, 改後 leetcode 中可以通過. 以下用改後的代碼.
1. 原代碼 put()中定義了個 Node xCopy = x, 然後一直用的 xCopy, 我把 xCopy 刪掉了, 全改為了 x.
2. 原代碼還在幾個函數中同時用了個 val 和 key 兩個變量(不是 Node 中的那個 member val), val 其值和 key 相等, 我把 val 刪掉了, 全改用 key
3. 在 words[i]那裡有點重復的話, 我把它簡化了.
4. use DFS to search 那裡有點重復的話, 我把它簡化了.

5.把最前面的 R=26 也去掉了.
6. words.length 那裡也簡化了一點.

```
private Node root;

// R-way trie node
private static class Node { //此句中的 static 去掉,在 leetcode 中也可以通過
    private String val;//當 trie 中某個 Node 為 eow 時, 這個 Node 的 val 即為這個 word, 否則 val 為空
    private Node[] next = new Node[26];
}

// insert key-value pair into trie
private void insert(String key) {
    root = put(root, key, 0); //若將本句寫為 put(root, key, 0), 則通不過.
}


private Node put(Node x, String key, int d) {
//key 為 dictionary 中的一個單詞(如 words[i]), 它是用來一個一個搜索它 的字母的. d 與 208. Implement Trie 中的 d 一樣, 沒有實質意義, 它的作用是控製遞歸的結束

    if (x == null)  x = new Node();
    if (d == key.length()) {
        x.val = key;//此 val 其實也起了 208. Implement Trie 中 eow 的作用
        return x;
    }
    int c = key.charAt(d) - 'a';
    x.next[c] = put(x.next[c], key, d + 1);
    return x;
}

public List<String> findWords(char[][] board, String[] words) {
    List<String> list = new ArrayList<>();
    int M = board.length;
    if (M == 0) return list;
    int N = board[0].length;

    // convert dictionary to a trie
    if (words.length == 0) return list;
    for (int i = 0; i < words.length; i++)
        insert(words[i]);

    // use DFS to search all valide words
    boolean[][] visited = new boolean[M][N];
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            int c =  board[i][j] - 'a';
             dfs(board, visited, i, j, root.next[c], list);//此處的調用和下面 dfs 調用自己的方式是一樣的, 為何要是 root.next[c], 可以看下面我對 dfs 作用的描述.
//注意是 root.next[c], 而不是 root. 若改為 root, 則報錯:Input:["a"], ["a"] Output:[] Expected:["a"]. 這是因為後面 dfs 中的實參 x.next[board[i - 1][j] - 'a']表明, 搜索是從 board[i][j]的下一個開始的, 而當 board 中只有一個字母時, 這個字母的下一個根本就不存在. (think of 上面雙 for 循環中調用 dfs(board, visited, 0, 0, root, list)時)
```

```
        }
    }

    // sort the list
    Collections.sort(list); //另忘了最後要 sort list!
    return list;
}

// DFS
//dfs(char[][] board, boolean[][] visited, int i, int j, Node x, List<String> list) 的作用就是, 從 board[i][j]出發, 查看 x.next 中的第 board[i][j]-'a'個 Node, 若其 val 非 null(當 trie 中某個 Node 為 eow 時, 這個 Node 的 val 即為這個 word, 否則 val 為空), 則將這個 val 加入到 list 中.
private void dfs(char[][] board, boolean[][] visited, int i, int j, Node x, List<String> list) {
    if (x == null)  return;
    if (x.val != null && !list.contains(x.val)) { //注意一個 path 中間可能也出現完整的單詞, 即 x.val != null, 當然 path 結尾處也可以出現.
        list.add(x.val);
    }
    visited[i][j] = true;
//以下的思想是: 實參 x.next[board[i - 1][j] - 'a']是為了保證每個 path 中的所有字每都是 board 中的. 如果這樣一個 path 中某個地方出現了完整的單詞, 則說明 words[i]中的某個單語的確可以在 borad 中找到它所有的字母.
//以下若像 79 題那樣, 將 if 中對 ij 和 visited 的要求提到外面去寫個總的:
 if(i < 0 || i >= board.length || j < 0 || j >= board[0].length)       return;
 if (visited[i][j]) return;
然後以下幾個 dfs()都不要 if. 這樣做看起來簡潔, 但實踐表明通不過, 因為上面 if(i < 0 || ...)是要进入了 dfs 函數後才能执行, 而下面調用 dfs(...x.next[board[i - 1][j] - 'a'])一句中, 在进入這個 dfs 函數之前, 要先算實參 x.next[board[i - 1][j] - 'a']的值, 而此處的[i-1][j]可能越界, 而由於還沒进入 dif 函數, 故 if(i < 0 || ...)對它無能為力.


    if (i > 0 && !visited[i - 1][j])                    // up
        dfs(board, visited, i - 1, j, x.next[board[i - 1][j] - 'a'], list);
    if (i < board.length - 1 && !visited[i + 1][j])     // down
        dfs(board, visited, i + 1, j, x.next[board[i + 1][j] - 'a'], list);
    if (j > 0 && !visited[i][j - 1])                    // left
        dfs(board, visited, i, j - 1, x.next[board[i][j - 1] - 'a'], list);
    if (j < board[0].length - 1 && !visited[i][j + 1])  // right
        dfs(board, visited, i, j + 1, x.next[board[i][j + 1] - 'a'], list);
    visited[i][j] = false;
}
```

213. House Robber II, Medium

http://wlcoding.blogspot.com/2015/03/house-robber.html?view=sidebar

Note: This is an extension of House Robber.
After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are**arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.
Given a list of non-negative integers representing the amount of money of each house, determine the

maximum amount of money you can rob tonight **without alerting the police**.
Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 本題跟 I 唯一的區別就是, 本題中的房子形成了一個圈. 還是用 DP. 本題之思想是將圈的問題轉化成 I 中的直線問題. 用 liear(start, end)表示房子按 I 中那樣直線排時, 搶 nums[start, … end] (inclusive) 的結果. 分兩種情況: 一是要搶 nums[0], 則 nums[1]和 nums[n-1]不能搶, 此時結果為 max1 = nums[0] + linear(2, n-2); 二是不搶 nums[0], 則 nums[1]和 nums[n-1]能搶, 此時結果為 max2 = liear(1, n-1). 最後結果為 max1 和 max2 中的較大者. 可專門為 liear 寫一個函數(即 William 的 dp 函數), liear 中用到的 DP 方法跟 I 一樣.

History:
E1 的代碼在 leetcode 中沒通過, 因為給出了錯誤結果. 我的思想跟 William 差不多, 也是分兩種情況: 一是是 d[0] = nums[0], d[1] = 0, 然後弄其它的, 二是 d[0] = 0, d[1] = nums[1], 然後弄其它的. 錯誤之處在於: 第二種情況中我要求 d[1]的錢一定要拿, 但實際上不一定. 這應該是一個常見錯誤, 所以在這裡寫下來. 以下用 William 的代碼.

E2 在初始化時范了一個小錯誤, 改後即通過. E1 沒留下甚麼 key, 都是 E2 寫的.
E3 范了一個不易查覺的小錯誤(我馬上說), 改後即通過. E3 的方法跟 key 中一樣, 但 E3 為了節省空間, 只給數組 d 分配了(end - start + 1)這麼多空間(實際上沒必要這樣做, 因為這樣做的空間複雜度還是 O(N), 空間沒佔便宜, 反而不好寫. E3 若不這麼做, 應該能一次通過), 故 res[i]中的 i 是減去 start 後的值, 所以在 res[i]的遞推式時, 要用 nums[i + start], 而不是 nums[i]. 我 E3 范的那個錯誤就是將其寫成了 nums[i].

1-d DP: Time ~ O(2N), Space ~ O(N)
Two cases:

•If s[0] is robbed, then s[1] and s[N - 1] cannot be robbed, so the max1 = s[0] + max of s[2 .. N - 2];
•If s[0] is not robbed, then s[1] and s[N - 1] can be robbed, so the max2 = max of s[1 .. N - 1].

Compare max1 and max2, and take the larger one as the final max.

我: 以下照樣用我改後的代碼(改後可通過), 原因同 198. House Robber 題

```
public int rob(int[] nums) {
 if(nums == null || nums.length == 0)
  return 0;

 int n = nums.length;

 int max1 = nums[0] + dp(nums, 2, n - 2);
 int max2 = dp(nums, 1, n - 1);

 return Math.max(max1, max2);
}

//dp(nums, start, end)的作用是找出 若輸入數組是 nums[start, end]這段子數組時, 本題之答案
private int dp(int[] nums, int start, int end) {
 if(start > end)
```

```
   return 0;

  int[] d = new int[nums.length];
  d[start] = nums[start];
  if(end - start >= 1)
   d[start + 1] = Math.max(nums[start], nums[start + 1]);

  for(int i = start + 2; i <= end; i++)
   d[i] = Math.max(d[i - 1], d[i - 2] + nums[i]);

  return d[end];
 }
```

214. Shortest Palindrome, Hard

http://segmentfault.com/a/1190000003059361

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.
For example:
Given "aacecaaa", return "aaacecaaa".
Given "abcd", return "dcbabcd".
Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases. Thanks to @Freezen for additional test cases.
Subscribe to see which companies asked this question

Key: 见下面<Key Starts>和<Key Ends>之間的內容.

History:
E1 直接看的答案.

E2 最開始用遞歸寫的 getCommonLength 函數, 在我電腦裡可以給出正確結果, 且我電腦測得裡對長 String 運行起來也很快, 但在 leetcode 中有個大輸入(一個四萬個 a 的 String)下報錯: stackOverFlowError (不是 Time Limit Exceeded), 不知道甚麼原因, 可能是遞歸棧的空間超出了, 總體來講, 應該是空間複雜度沒合符要求, 後來又重寫代碼, 不用遞歸, 改用類似動能歸劃的方法(即不更新數組 p[k]值的動能歸劃, 後來發現以下答案也這樣做的), 總算通過了, 本題花了老子四五個小時, 不過通過 E2 的表現, 可看出我對算法(遞歸和動能歸劃等)已經掌握得很熟練了, 想用哪個就用哪個, 而且本題還是 Hard. E2 弄了 key.

E3 沒做出來, E3 大多數步驟都想出來了, 就是在求公共前綴後綴上卡住了, 主要原因是 E3 以為 s[i]在廷續 s[i-1]的公共前綴後綴時, 一定要把廷續出來的 s[i]的公共前綴後綴弄成 parindrome, 所以沒想出來. 這也是受了 E2 的 key 的誤導. 實際上, 求公共前綴後綴那部分, 是不要求弄出來的公共前綴後綴為 parindrome 的. E3 重寫了 E2 的 key 的 求公共前綴後綴那部分. 以下答案的代碼不用了不太好懂, 我 E3 將其改寫為了比較好懂的形式, 以後用 E3 改寫的答案代碼(能通過).

我: 網上很多人用的 KMP 算法, 也有很多人用的 Manacher's Algorithm(比如 William, William 的另一個 O(n^2)的方法超時了, 所以我不用 O(n^2)的方法, 他的 Manacher's Algorithm 跟之前的題目有關, 我難得看)
我: 不要被下面的 KMP 算法嚇到, 本題除了 '本題的「公共前綴後綴長度」和 KMP 算法中的「部分匹配表」是同一回事' 之外, 跟 KMP 算法並沒有甚麼卵關係. 注意 KMP 算法也沒講如何求「公共前綴後綴長度」(當然有的 KMP 版本可能講了的, 但無所謂, 因為本題中答案中也要講如何找, 而不用專門去看 KMP 算法), 所以還是

要自己想算法來求「公共前綴後綴長度」(which is the key point of this problem), 所以做本題的話, 完全可以不看 KMP 算法, 但為了擴大我的知識面, 還是可以看看後面網址裡對 KMP 算法的講解.

用 Java 暴力是可以过的，思路也很简单：补充完成之后的回文串中心必定在原字符串中，所以原字符串以第一个字符为起点必然存在至少一个回文串（长度可以为 1），那么任务就变为找到原字符串中以第一个字符为起点最长的回文串，找到之后剩下的工作就是把剩余部分的翻转补充到原字符串头部即可。
这样代码逻辑就很简单，就是从原字符串的头部开始截取子串，长度递减，直到获取到第一个是回文串的子串，此时就找到了需要截断的部分，从该位置开始到原字符串末尾就是需要截取并翻转拼接的部分。算法复杂度是 O(n^2)。实现代码(我: 已省).

LeetCode 做多了也就知道 O(n^2)的算法必然有改进版，自己思考了下没有悟出来，就参考了这篇文章：[LeetCode] Shortest Palindrome 最短回文串(我: 即下面的)。

<Key Starts>
其实思路也很简单：
1. 求字符串 s 的翻转 s_rev
2. 将两个字符串进行拼接：{s}#{s_rev}(我: 此步的目地就是找 comLen)
3. 找出新字符串中最长公共前缀后缀长度 comLen(我: 「公共前缀后缀」意思就是{s}#{s_rev}的 前綴和後綴 它們兩個的最長的公共字符串, 其實就是「s 的從 s[0]開始的最長回文子串」, 如下面我例 1 中的 abcba. 本答案作者將其叫做 「最長公共前缀后缀」, 我為了避免囉嗦, 我將「最長」兩字去掉, 並不改變此短語的意思)
4. s_rev.substring(0, s.length() - comLen)就是在原字符串头部插入的子串部分 (我: 其實就是將 s 中除去「s 的從 s[0]開始的最長回文子串」剩下的那串(如下面我例 1 中的 ef)反轉後 加到 s 前去.) (我: 注意 s.substring(start, end)返回的是 s[start, end-1])

我: 兩個例子(空格只是為了看起來清楚些, 空格不是 string 的一個字符):
s = abcba ef ,
s_rev = fe abcba,
{s}#{s_rev} = abcba ef#fe abcba
len = 7, comLen = 5 (即 comLen 為 abcba)
result = fe abcbaef

s = gh abcba ef,
s_rev = feabcbah g,
{s}#{s_rev} = gh abcba ef# feabcbah g
len = 9, comLen = 1 (即 comLen 為 g)
result = feabcbah ghabcbaef

我: 由此可知, 本方法的難點在於如何找出一個 string(本題中即為{s}#{s_rev})的「公共前綴後綴長度」. 這個「公共前綴後綴長度」即 KMP 算法中的「部分匹配表」, 或本答案作者稱的「next 數組」, 這也是本題唯一能與 KMP 算法搭上關係的地方. 除此之外便跟 KMP 算法沒有甚麼卵關係, 而 KMP 算法中並沒講如何找出「公共前綴後綴長度」, 所以總體來講, 本題跟 KMP 算法並沒有甚麼卵關係. 以下的(即 key Ends 之後的)幾段就是講如何找出這個 公共前綴後綴長度, 寫得有點不好懂, 不用看, 看我下面寫的:

用數組 p[k] = m 表示 s[0,...k]這個子串的 公共前綴後綴長度 為 m, 即表示 s[0,...m-1]和 s[k-m+1, k]是相等的. 若已知 p[i-1], 如何求 p[i] (i 要在 s 中掃描)? (先大概看看下面例子, 知道 i 和 j (j=i-1)的位置):

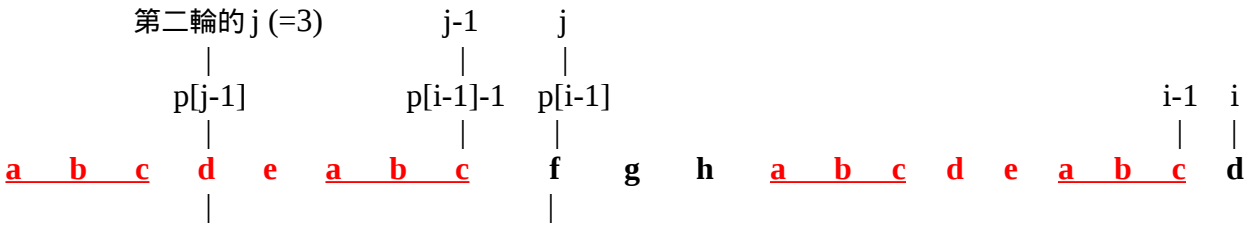
要先比較 s[i]和 s[p[i-1]], 若記 j=i-1, 則為比較 s[i]和 s[j], 這裡分兩種情況:

1. 若 s[i] = s[j], 則說明 s[i]廷續了 s[0,...i-1]的公共前綴後綴(如下面第一例中的 abcdeabc, 但該例中 s[i] != s[j]), 故此時 p[i] = p[i-1] + 1.
2. 若 s[i] != s[j] (如下面第一例之情況), 則說明 s[i]廷續無望. 此時想到 s[0,...i-1]公共前綴後綴(如下面第一例中的 abcdeabc)本身也可能有 公共前綴後綴(如 abc), 就像分形一樣, 故 s[i]還是有可能廷續 '公共前綴後綴的 公共前綴後綴' (如 abc). 此時就應當拿 s[i]跟 p[j-1] (即 p[p[i-1]-1])位置上的比較, 若不相同, 則再找 '公共前綴後綴 的 公共前綴後綴 的 公共前綴後綴', 即 s[i]跟 p[p[p[i-1]-1]-1]位置上的比較, …

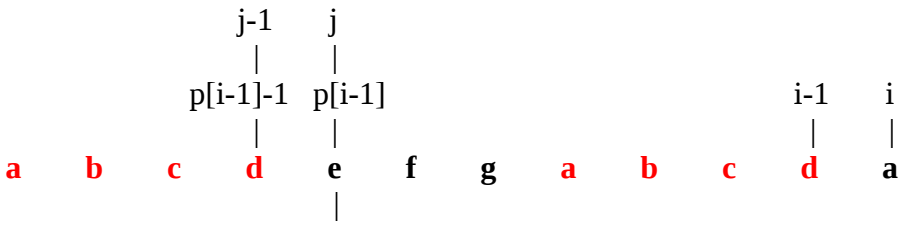
例如下例(本題的公共前綴後綴是 palindrome 的, 但以下方法也適用於公共前綴後綴不是 palindrome 之情況, 故以下例子是 general 的 公共前綴後綴不是 palindrome 之情況):

a b c d e a b c f g h a b c d e a b c

以上的 string 的公共前綴後綴為 a b c d e a b c, 而 '這個公共前綴後綴' 自己還有一個公共前綴後綴: a b c. 如果以上的 string 是 s[0, ..i - 1], 若 s[i]為 d, 即(從右往左看):



有一種情況要注意, 就是比著比著, j=0 了, 如下例, 要專門處理:



注意以下做法 是行不通的:
用 p[k]表示 s[0,...k]的公共前綴後綴為 s[0,...p[k]] (即 p[k]表示的不是長度), 即 s[0,...p[k]] = s[n-p[k], ...n]. 行不通的原因是: 它沒法表示 公共前綴後綴 個數為 0 時之情況, 因為即使 p[k]=0, 它表示的也是 s[0]為公共前綴後綴. 這點花了我 E3 幾個小時的時間.

<Key Ends>

我: 以下的不用看:

1. 若 s[i]=s[j] (下圖中的 b 即 s[i], a 即 s[j])，也就是当前字符延续了之前的公共前缀后缀，那么 p[i]=p[i-1]+1 即可
2. 若 s[i]!=s[j], 則:

以下兩行中，左右的紅色表示公共前綴後綴, 紅色部分也有它自己的公共前綴後綴, 即藍色部分(think of 分形), 注意圖中的四個藍色部分都是相同的子串.

****
****

以下兩行中, a 和 b 代表黑色*. (b 即 s[i], a 即 s[j]). 若 a != b, 則可以比較 b 和 c, 若 b=c, 則*****c 和*****b 仍為公共前綴後綴.

********** a********** b
*****c********** a********** b

到目前位置，期望********** a 為最长公共前缀后缀的期望已经失败，那我是否可以期望下缩短长度之后能有匹配的公共前缀后缀呢？答案是肯定的，因為上圖中的***** c 即為最长公共前缀后缀. 对于位置 i-1 而言，公共前缀后缀的长度依次为：p[i-1],p[p[i-1]-1],p[p[p[i-1]-1]-1],..... 在此基础上，对于位置 i 而言，只要比对某几个特定的位置，看 s[i]是否能符合条件（即是否和当前公共前缀后缀后的第一个字符相等）就能求得 p[i]的值.当然，如果比对某个位置的时候 p[x](我: p[x]的一個例子為 p[p[p[i-1]-1]-1])已经为 0，那么就可以马上结束比较跳出循环，然后只要和首字母比对下就行了.

举个例子：

(我: 我對本段原文的 typo 作了修改)对于字符串 s：babcd，先求 rev_s：dcbab，拼接之后：babcd#dcbab。上文已经解释过，s 的前缀必然是一个回文串（长度可能为 1），任务就是求这个回文串的最长长度，因此拼接之后的{s}#{s_rev}必然有公共前缀后缀，任务就是求这个公共前缀后缀的最长长度，那么这个时候就需要祭出 KMP 算法了。有了解的同学，估计一看就看出这个就是求 KMP 里的 next 数组。由于之前学 KMP 的时候也只学了个一知半解，所以这次又重新学习了下从头到尾彻底理解 KMP（2014 年 8 月 22 日版），这下对 KMP 又有更好的理解了。我: 那篇文章很長且不好懂, 實際上這篇更簡明好懂: http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm.html

详细的 KMP 算法上面提到的文章里讲的非常详细，就不从头说了。这里讲一讲我之前一直困惑现在理解了的点(我: 以下幾段實際上是講如何找出 公共前綴後綴長度, 主要意思就是我下面畫的那兩個星號圖, 先看星號圖就好懂了)。

(我: 本段我作了順序調節)对于 KMP 算法，核心的地方就是求 next 数组(我: 即上面我帖的那個網址中說的「部分匹配表」)，而求 next 数组中比较难理解的地方就是当当前位置的字符和目标字符不匹配的时候。用 p 表示 next 数组，其中 p[k]表示从 0 到 k 位置为止公共前缀后缀的长度，例如：abacaba，公共前缀后缀长度是 3. (我: p[k] = m 表示 s[0,...k]這個子串的 公共前綴後綴長度 為 m, 即表示 s[0,...m-1]和 s[k-m+1, k]是相等的) 当 p[k]=m 则表示 s.substring(0,m)和 s.substring(k-m+1,k+1)是相等的. 对于字符串 s，已经有 p[0]到 p[i-1]，且 p[i-1]=j (我: 即表示 s[0,...i-1]這個子串的 s[0,...j-1]和 s[i-j, i-1]是相等的)，

求 p[i]：

1.若 s[i]=s[j]，也就是当前字符延续了之前的公共前缀后缀，那么 p[i]=p[i-1]+1 即可

2.若 s[i]!=s[j]，即 s.substring(0,j)和 s.substring(i-j+1,i+1)是不匹配的，但是仍然可能存在 s.substring(0,x)和 s.substring(i-x+1,i+1)，这一点就是我以前最不能理解的地方，这次结题的经历加深了我这部分的理解。

到目前位置，期望 i 位置的最长公共前缀后缀为 j+1 的期望已经失败，那我是否可以期望下缩短长度之后能有匹配的公共前缀后缀呢？答案是肯定的，因为对于位置 i-1 来说，其实是可能存在多个公共前缀后缀的，只是 p[i-1]只记录其中最长的，那么次长的是多少呢，答案就在 p[j-1]里。对于位置 i-1 来说，已知 0 到 j-1 的子串和 i-j+1(我: 應該是 i-j)到 i-1 子串是相等的，而对于位置 j-1 来说，从 0 到 p[j-1]-1 的子串(我: 即上面第一段藍色)和从 j-p[j-1]到 j-1 的子串是相同的(我: 即上面第二段藍色)，更进一步和 i-p[j-1]到 i-1 的子串(我: 即上面第四段藍色)也是相同的，那如果现在 比较一下 i 和 p[j-1]是否相等(我: 嚴格地說, 即 s[i]和 s[p[j-1]]是否相等) 同样可以求出最长公共前缀后缀的值（因为 p 中记录是到每个位置为止的最长公共前缀后缀，所以这样每次递推下去每次得到都是当前可能的最长公共前缀后缀）。

梳理一下，就是对于位置 i-1 而言，公共前缀后缀的长度依次为：p[i-1],p[p[i-1]-1],p[p[p[i-1]-1]-1],.....(回憶: p[k] = m 表示 s[0,...k]這個子串的 公共前綴後綴長度 為 m, 即表示 s[0,...m-1]和 s[k-m+1, k]是相等的)。在此基础上，对于位置 i 而言，只要比对某几个特定的位置，看 s[i]是否能符合条件（即是否和当前公共前缀后缀后的第一个字符相等）就能求得 p[i]的值。当然，如果比对某个位置的时候 p[x](我: p[x]的一個例子為 p[p[p[i-1]-1]-1])已经为 0，那么就可以马上结束比较跳出循环，然后只要和首字母比对下就行了（因为这种情况说明可能的公共前缀后缀都已经被比对完了，s[i]依然不符合条件，那么只能从头开始了）。

应用了 KMP 之后的实现代码(不用)：

```
public class Solution {
    public String shortestPalindrome(String s) {
        StringBuilder builder = new StringBuilder(s);
        return builder.reverse().substring(0, s.length() - getCommonLength(s)) + s;
    }

    private int getCommonLength(String str) {
        StringBuilder builder = new StringBuilder(str);
        String rev = new StringBuilder(str).reverse().toString();
        builder.append("#").append(rev);
        int[] p = new int[builder.length()];
```

```
        for (int i = 1; i < p.length; i++) {
            int j = p[i - 1];
            while (j > 0 && builder.charAt(i) != builder.charAt(j)) j = p[j - 1];
            p[i] = j == 0 ? (builder.charAt(i) == builder.charAt(0) ? 1 : 0) : j + 1;
        }
        return p[p.length - 1];
    }
}
```

E3 改寫的答案代碼(能通過, 用它):

```
    public String shortestPalindrome(String s) {
        StringBuilder sb = new StringBuilder(s);
        String sRev = sb.reverse().toString();
        int commonLen = getCommonLen(s + "#" + sRev);
        int addLen = s.length() - commonLen;
        return sRev.substring(0, addLen) + s;
    }

    private int getCommonLen(String s) {
        int n = s.length();
        int[] p = new int[n];

        for(int i = 1; i < n; i++) { //i 從 1 開始 是因為 下面會用到 p[i-1]
            int j = p[i - 1];

            while(j >= 1 && s.charAt(i) != s.charAt(j)) j = p[j - 1]; //要求 j>=1 是因為會用到 p[j-1]

            if(j == 0) {
                if(s.charAt(i) == s.charAt(0)) p[i] = 1;
                else p[i] = 0;
                continue;
            }

            p[i] = j + 1;
        }

        return p[n - 1];
    }
```

215. Kth Largest Element in an Array, Medium

http://pisxw.com/algorithm/Kth-Largest-Element-in-an-Array.html

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

Given [3,2,1,5,6,4] and k = 2, return 5.
Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.
Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: Quick Select 算法. 先看下面截圖 CLRS 的 Quick Sort (簡單易懂). 代碼看我 E2 代碼, 不要看答案代碼. 我 E2.5 覺得本題絕對可以算 Hard.

寫一個 Quick Sort 中的 partition 函數:partition(nums, p, r), 其作用為: 將 nums[p, ...r]這個子數組元素重新 arrange. 設 arrange 前的 nums[r]為 x. 則 arrange 後的數組為: {比 x 小或等於 x 的, x, 比 x 大的}, 並返回 x 在 arrange 後的數組(即{比 x 小或等於 x 的, x, 比 x 大的}這個數組)中的 index(**設此 index 為 m**).

易知, **nums[m]是在整個 nums 數組中第 m 小(從第 0 小, 第 1 小... 這樣排的)的**. 這是因為以下兩點:
1. nums[p,..m-1]都比 m 小, 這是 arrange 造成的.
2. nums[0,...p - 1]都比 nums[p]小, 這是調用 partition(nums, p, r)之前就已經 被我們假定 成立的, 即這是一個不變量.

再寫一個 findK 函數(利用類似二分法的方法 , 並在它當中遞歸調用 findK 函數). findK(nums, k, p, r)的作用為: 在子數組 nums[p, ...r]中找出整個 nums 數組(即 nums[0,...nums.length-1])中第 k 小的數, 並返回這個數. 其中「第 k 小」是按第 0 小, 第 1 小, 第 2 小...的 convention 來的. 注意題目中的「第 k 大」是按第 1 大, 第 2 大...的 convention 來的.

History:
E1 直接看的答案.
E2 沒做出來, 主要是因為 E1 的 key 有問題. E2 改寫了 key. E2.5 又較大地改寫了 key.
E3 沒想好一會兒, 沒想出來, 直接看的答案.

From CLRS p192 (也可見文件"數据結構和排序總結", randomizedSelect 函數, 它比以下代碼要稍複雜):

Quicksort, like merge sort, applies the divide-and-conquer paradigm introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide: Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q - 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

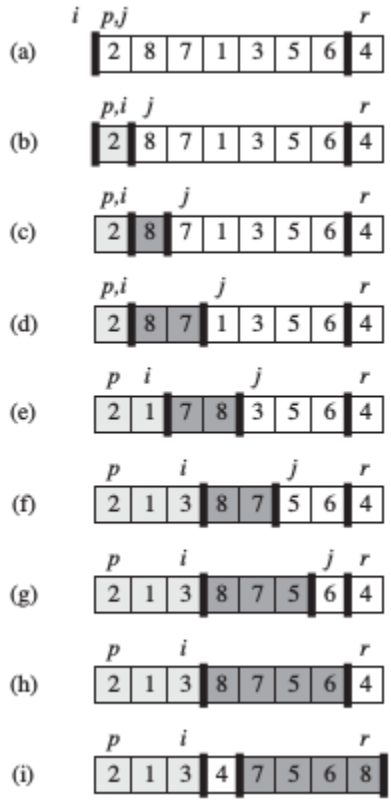
Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

本算法之思想為:
1. A[i]的左邊(包括A[i])都是比A[r]小的數, 稱為1區
2. A[i]的右邊(不包括A[i])都是比A[r]大的數, 稱為2區
3. A[j]的右邊(包括A[j])是還沒訪問的, 稱為3區
(以上三句可以看作是一個不變量)
所以若if使得 互換A[i+1]和A[j], 其作用即為擴大1區, 縮小3區,
若if使得 沒互換A[i+1]和A[j], 其作用即為擴大2區, 縮小3區
最終目的是只有 1區(即比A[r]小的數) 和 2區(即比A[r]大的數)

Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION always selects an element $x = A[r]$ as a *pivot* element around which to partition the subarray $A[p..r]$. As the procedure runs, it partitions the array into four (possibly empty) regions. At the start of each iteration of the **for** loop in lines 3–6, the regions



我: 網上絶大多數人都是用的 Quick Select 算法(Quick Sort 的變種). William 是用的非主流的 Priority Queue 的方法(效率要低些), 所以不用 William 的代碼. 以後的題都不要只看 William 的代碼. 以下的算法經我跟其它人比較, 已證實就是 Quick Select 算法.

该题的思路在每个算法书里基本上都会有，就是找出一个无序数组中的第 k 大的元素。

首先最容易想到的就是对这个数组进行排序，然后直接得到结果，但是时间复杂度为 O(nlogn)。但是由于我们关心的是第 K 个，所以没必要对整个数组进行排序，此时我们就会想到快速排序中的 partition 功能，将一个数组进行分割，得到两个部分，中间元素为 x,前一部分元素都小于 x,后一部分元素都大于 x。根据每一部分的长度，就可以看出第 K 个元素在哪一部分中。这样一个 partition 下去，就能得到结果，时间复杂度为 O(n)。

算法代码

代码采用 JAVA 实现：

答案代碼(符號跟 CLRS 不同, 不用):

```
public class Solution {
  // findKthLargest(nums, k)的作用是: 返回 nums 數組的第 k 大的元素. 其中「第 k 大」是按第 1 大, 第 2
```

```
大...的 convention 來的. 注意與下面的「第 k 小」的 convention 不同.
    public int findKthLargest(int[] nums, int k) {
        return findK(nums,nums.length-k,0,nums.length-1);
    }
   //以下要遞歸調用 findK 函數. findK(nums, k, i, j)的作用為: 在子數組 nums[i, ...j]中找出整個 nums 數組
(即 nums[0,...nums.length-1])中第 k 小的數, 並返回這個數. 其中「第 k 小」是按第 0 小, 第 1 小, 第 2 小...
的 convention 來的.
    public int findK(int[] nums,int k,int i,int j)
    {
        if(i>=j) return nums[i]; //此句不要也能通過. 建議不要, 因為有點 confusing.
        int m=partition(nums,i,j);
        if(m==k) return nums[m];
        if(m<k)
            return findK(nums,k,m+1,j); 整個 nums 數組中第 k 小
        else
            return findK(nums,k,i,m-1); 整個 nums 數組中第 k 小
    }

    // partition(nums, i, j)的作用是將 nums[i, ...j]這個子數組分好: {比 nums[m]小或等的, nums[m], 比
nums[m]大的}, 並返回 m 的值. nums[m]是整個 nums 數組中第 m 小的(按第 0 小, 第 1 小, 第 2 小...的
convention). 注意 m 是整個 nums 數組中的 index.  nums[m]是整個 nums 數組中第 m 小的的原因為: 已經
假定 nums[i, ..j]前的數都比 nums[i, ..j]中的小(或理解為這麼一個不變量). 在 findK 調用自己時, 這個不變量
也是成立的. 而在 findKthLargest 調用 findK 時, 由於 nums[0, nums.length-1]前面已經沒有數了, 故這個不
變量也是成立的.
//此處符號跟前面 CLRS partition 函數中符號的對應關係為:
//此處        i          j          m          n
//CLRS      p          r          i+1        j
    public int partition(int[] nums,int i,int j){
        int x=nums[i];
        int m=i;
        int n=i+1;
        while(n<=j){
            if(nums[n]<x)
            {
                m++;
                //以下三句是將 nums[m]和 nums[n]互換. 記憶: 互換的寫法其實就是個環(tmp-m-m-n-n-tmp)
                int tmp=nums[m];
                nums[m]=nums[n];
                nums[n]=tmp;
            }
            n++;
        }

        //将 x 与 nums[m]进行互换
        int tmp=nums[m];
        nums[m]=nums[i];
        nums[i]=tmp;
```

```
        return m;
    }
}
```

E2 將以上的代碼改為 CLRS 的符號和習慣(能通過, 用它):

```
    public int findKthLargest(int[] nums, int k) {
        return findK(nums, nums.length - k, 0, nums.length - 1);
    }
```

findK(nums, k, p, r)的作用為: 在子數組 nums[p, ...r]中找出整個 nums 數組(即 nums[0,...nums.length-1])中第 k 小的數, 並返回這個數. 其中「第 k 小」是按第 0 小, 第 1 小, 第 2 小...的 convention 來的.

```
    private int findK(int[] nums, int k, int p, int r) {
        int m = partition(nums, p, r);

        if(m == k) return nums[m];
        else if(m < k) return findK(nums, k, m + 1, r);
        else return findK(nums, k, p, m - 1);
    }
```

//partition(nums, p, r)之作用跟 CLRS 中的 partition(A, p, r)函數作用一樣, 即為: 將 nums[p, ...r]這個子數組元素重新 arrange. 設 arrange 前的 nums[r]為 x. 則 arrange 後的數組為: {比 x 小或等於 x 的, x, 比 x 大的}, 並返回 x 在 arrange 後的數組(即{比 x 小或等於 x 的, x, 比 x 大的}這個數組)中的 index(設此 index 為 m).

```
    private int partition(int[] nums, int p, int r) {
        int x = nums[r];
        int i = p - 1;

        for(int j = p; j < r; j++) {
            if(nums[j] <= x) {
                i++;
                //swap nums[i] and nums[j]
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
            }

        }

        //swap nums[i + 1] and nums[r]
        int temp2 = nums[i + 1];
        nums[i + 1] = nums[r];
        nums[r] = temp2;

        return i + 1;
    }
```

216. Combination Sum III, Medium

http://wlcoding.blogspot.com/2015/03/combination-sum-i-ii.html?view=sidebar

Find all possible combinations of ***k*** numbers that add up to a number ***n***, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.
Ensure that numbers within the set are sorted in ascending order.

Example 1:
Input: ***k*** = 3, ***n*** = 7
Output:
[[1,2,4]]

Example 2:
Input: ***k*** = 3, ***n*** = 9
Output:
[[1,2,6], [1,3,5], [2,3,4]]
Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: item-res 模版遞歸法. DFS, 寫一個用來遞歸調用的 addUp(start, k, n), 其作用為: 若 k 和 n 都為 0, 則將 list(復製一個新的)添加到 listSet 中. 否則在{start, start+1, … 9}中找出 k 個數, which can sum up to n. 並將這 k 個數添加到 list 中(list 中之前可能還有別的數). 最後, 無論是否 k 和 n 都為 0, 都要將 list 還原成調用本 addUp 之前的樣子.

History:
E1 直接看的答案.

E2 幾分鍾寫好, 一次通過.

E3 沒多久就寫好, 最開始將遞歸結束的條件寫為了 if(k == 0 || n == 0), 結果中會出現 k 個數的組合但和不為 n, 然後又改為 if(n == 0), 結果中會出現 非 k 個數 且和為 n 的組合, 然後改為 if(k == 0 && n == 0), 即通過. 後來發現 William 也是這樣寫的. 同樣, 做本題時我已經將 key 全忘了, 都不確定本題是否還能用遞歸法, 所以本題也是 E3 獨立做出來的. E3 的代碼跟 William 差不多, 但風格更接近 Code Ganker 和我前幾題的 E3 風格, 故以後用 E3 的代碼.

我: William 的解說不好懂, 不看.

DFS: Time ~ O(N!), Space ~ O(N)
Recursively call:

•addUp(i + 1, k - 1, n - i), i = start ... 9
i + 1: candidate
k - 1: number of elements to sum up
n - i: target

William 的代碼(不用):

```
private List<List<Integer>> listSet = new ArrayList<List<Integer>>();
private List<Integer> list = new ArrayList<Integer>();
```

```
public List<List<Integer>> combinationSum3(int k, int n) {
    addUp(1, k, n);
    return listSet;
}

//addUp(start, k, n)的作用為: 若k和n都為0, 則將list添加到listSet中. 否則在{start, start+1, … 9}中找出k個數, which can sum up to  n. 並將這k個數添加到list中(list中之前可能還有別的數). 最後, 無論是否k和n都為0, 都要將list還原成調用本addUp之前的樣子.
private void addUp(int start, int k, int n) {
    if (k < 0 || n < 0) return; //不考慮這種情況, 也能通過. 但為了嚴謹, 還是考慮比較好(後面addUp(i+1, k-1, n-i)中的n-i可能為負)
    else if (k == 0 && n == 0) { //剛好找到
        listSet.add(new ArrayList<Integer>(list));//注意是將list復製一個新的加进去. 注意anonymous ArrayList的用法.
    } else {
        for (int i = start; i <= 9; i++) {
            //以下兩句是加入含i的這麼個組合
            list.add(i);
            addUp(i + 1, k - 1, n – i); //注意是n-i, 不是n-1
            list.remove(list.size() - 1);//為何不是list.remove(0)?因為前面list.add(i)之前, list中可能還有別的數
        }
    }
}
```

E3 的代碼(用它):

```
public List<List<Integer>> combinationSum3(int k, int n) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(k <= 0 || n <= 0) return res;
    helper(k, n, 1, new ArrayList<Integer>(), res);
    return res;
}

private void helper(int k, int n, int start, List<Integer> item, List<List<Integer>> res) {
    if(k == 0 && n == 0) {
        res.add(new ArrayList<Integer>(item));
        return;
    }

    for(int i = start; i <= 9; i++) {
        item.add(i);
        helper(k - 1, n - i, i + 1, item, res);
        item.remove(item.size() - 1);
    }
}
```

217. Contains Duplicate, Easy

http://wlcoding.blogspot.com/2015/05/contains-duplicate.html?view=sidebar

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

Subscribe to see which companies asked this question

Key: 用 Set. 簡單, 直接寫.

History:
E1 此題是最先做的幾道之一, 根本都不知道用 Set, 所以看的答案.
E2 一次通過.
E3 兩分鍾寫好, 一次通過.

William:

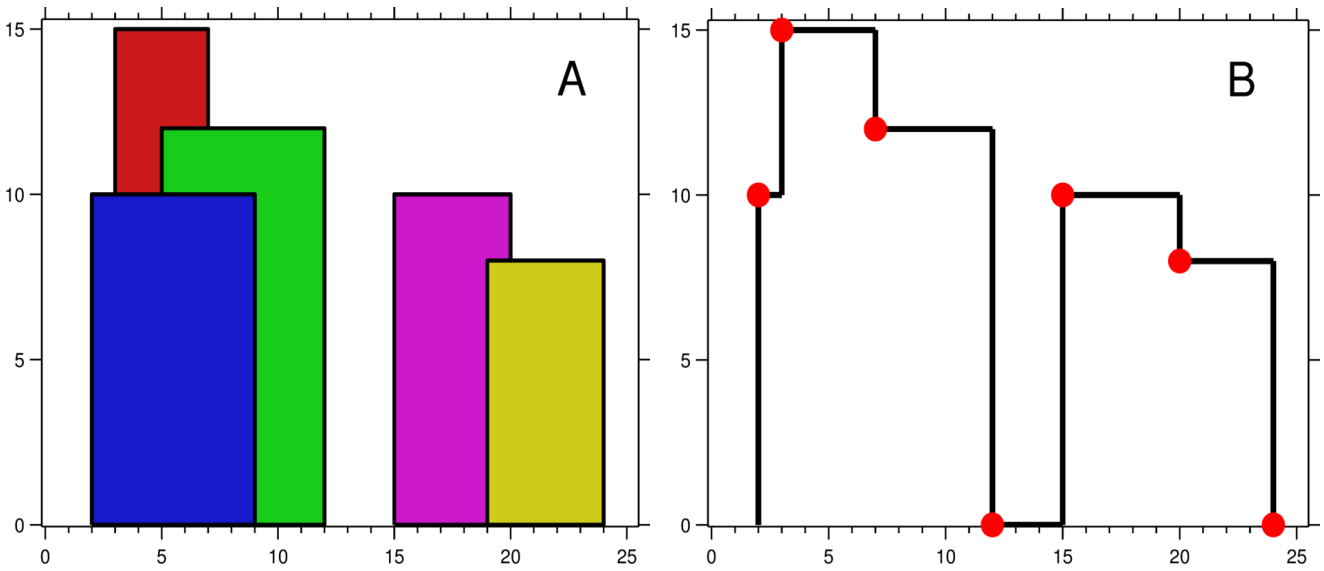
Hash Set: Time ~ O(N), Space ~ O(N)

```
public boolean containsDuplicate(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int i : nums) {
        if (set.contains(i))    return true;
        else                    set.add(i);
    }
    return false;
}
```

218. The Skyline Problem, Hard

http://codechen.blogspot.com/2015/06/leetcode-skyline-problem.html

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:

- The number of buildings in any input list is guaranteed to be in the range [0, 10000].
- The input list is already sorted in ascending order by the left x position Li.
- The output list must be sorted by the x position.
- 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], ...]

Credits:
Special thanks to @stellari for adding this problem, creating these two awesome images and all test cases.
Subscribe to see which companies asked this question

Key: 本方法不易想到, 要記住. buildings[i][j]的 i 表示第 i 個 building, 即 building[i]. 而 building[i]是一個有三個元素的數組, 比如 building[i] = {Li, Ri, Hi}.

方法:
寫一個 Height 類, 成員數据是 index 和 height. 每個 building(如(Li, Ri, Hi))都可以轉換為兩個 Height object: Height(Li, -Hi)和 Height(Ri, Hi). 其中 Height(Li, -Hi)專門將 Hi 寫為負的, 是為了好在這兩個 Height 中辨證哪個為 Height(Li, -Hi), 哪個為 Height(Ri, Hi).

本方法要用到一個 List 和一個 PriorityQueue (當前還要用一個 List 來放結果, 這個 List 不算).

List 中放入所有的 Height object, Height(Li, -Hi)和 Height(Ri, Hi)都要放. 然後寫一個 Comparator 對這個 List 進行昇序排序, 先按 index(即 Li 和 Ri)排, index 相同時按 height 排.

PriorityQueue 中放入所有的高度的絕對值. 用 Collections.reverseOrder() 作為定義 PriorityQueue 時的 Comparator, 使得最大值最優先出來(Java p808-top). 在遍歷(放 Height 的)List 前, 要在 PriorityQueue 中放入一個 0, 這是必要的, 否則後面可能把 PriorityQueue 搞空, 就 remove 不出東西來了. 這個 0 也巧妙地解決了一堆 building 結束後的地上那個紅點.

遍歷(放 Height 的)List, 遇到高度為負的 Height 時, 將此高度(的絕對值)加入 PriorityQueue 中; 遇到高度為正的 Height 時, 將此高度從 PriorityQueue 中刪除. 加入或刪除之後, 從 PriorityQueue 中 peek 一個高度出來(稱為 cur), 將 cur 與上一輪中 peek 出來的高度(稱為 prev, 遍歷之前初始化為 0)比較, 若 cur 和 prev 不同, 則將 cur 這個高度(及當前所遍歷到的 Height 的 index)加入到結果中 並且更新 prev 為 cur.

History:

E1 直接看的答案.
E2 花了一晚上小心翼翼地寫, 結果一次通過, NB!
E3 沒認真想, 就直接看的答案. E3 重寫了 key, 感覺寫得好清楚.

silhouette: 轮廓

Tao: 上面的紅字要比以下的准確好懂.

(1) 自建一个名为 Height 的数据结构(我: 即一個 Height 類, 成員數据是 index 和 height)，保存一个 building 的 index 和 height。约定，当 height 为负数时表示这个高度为 height 的 building 起始于 index；height 为正时表示这个高度为 height 的 building 终止于 index。

(2) 对 building 数组进行处理，每一行[Li, Ri, Hi]，根据 Height 的定义，转换为两个 Height 的对象，即，Height(Li, -Hi) 和 Height(Ri, Hi)。 将这两个对象存入 heights 这个 List 中。

(3) 写个 Comparator 对 heights 进行升序排序，首先按照 index 的大小排序，若 index 相等，则按 height 大小排序，以保证一栋建筑物的起始节点一定在终止节点之前。

(4) 将 heights 转换为结果。使用 PriorityQueue 对高度值进行暂存。遍历 heights，遇到高度为负值的对象时，表示建筑物的起始节点，此时应将这个高度加入 PriorityQueue。遇到高度为正值的对象时，表示建筑物的终止节点，此时应将这个高度从 PriorityQueue 中除去。且在遍历的过程中检查，当前的 PriorityQueue 的 peek()是否与上一个 iteration 的 peek()值（prev）相同，若否，则应在结果中加入[当前对象的 index, 当前 PriorityQueue 的 peek()]，并更新 prev 的值。

思路是照抄这个链接的：http://www.cnblogs.com/easonliu/p/4531020.html
（脑洞：C++全忘光了，这个链接的代码好久才看懂，赶快补！）

Java Code:

```
public class Solution {
    public List<int[]> getSkyline(int[][] buildings) {
        List<int[]> result = new ArrayList<int[]>();
        if (buildings == null || buildings.length == 0 || buildings[0].length == 0)
{
            return result;
        }

        List<Height> heights = new ArrayList<Height>();
        for (int[] building : buildings) { //循環變量居然還可以是數組
            heights.add(new Height(building[0], -building[2]));
            heights.add(new Height(building[1], building[2]));
        }

        Collections.sort(heights, new Comparator<Height>() {
            @Override
            public int compare(Height h1, Height h2) {
                return h1.index != h2.index ? h1.index - h2.index : h1.height -
h2.height;
            }
        });
```

//以下的已 record 到 Java p808.

```
//PriorityQueue 見 Java 書 p808. Collections.reverseOrder()Java 書中 p797 也講到過, 但很簡略, 看下面我寫的就可以了.
//以下一句也可以寫為如下兩句(可以通過):
        Comparator<Integer> cmp = Collections.reverseOrder();
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>(1000, cmp);
//Collections.reverseOrder(): It is used to get a comparator that imposes the reverse of the natural ordering on a collection of objects that implement the Comparable interface.

        PriorityQueue<Integer> pq = new PriorityQueue<Integer>(1000, Collections.reverseOrder()); //pq 中 height 最大的最優先(peek)出來
        pq.offer(0);//這句是必要的, 否則後面可能把 pq 搞空, 就 remove 不出東西來了. 這個 0 也巧妙地解決了一堆 building 結束後的地上那個紅點.
        int prev = 0;
        for (Height h : heights) {
            if (h.height < 0) {
                pq.offer(-h.height);//另忘了負號
            } else {
                pq.remove(h.height);//注意 remove 的不是 pq 中最大的, 因為已經指定 remove 誰了. 注意是 remove, 而不是 poll.
            }
            //注意 pq.offer 或 pq.remove 後, 都會執行以下幾句:
            int cur = pq.peek();//注意 peek 出的是 pq 中最大的
            if (cur != prev) {
                result.add(new int[]{h.index, cur});//注意 anonymous array 的用法
                prev = cur;
            }
        }

        return result;
    }

    class Height {
        int index;
        int height;
        Height(int index, int height) {
            this.index = index;
            this.height = height;
        }
    }
}
```

219. Contains Duplicate II, Easy

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i]** = **nums[j]** and the difference between *i* and *j* is at most *k*.
Subscribe to see which companies asked this question

Key: 用 HashMap. 注意用 map.put(nums[i], i), 而不是 map.put(i, nums[i]), 否則沒法通過 nums[i]找到 i. 雖然此方法空間沒有 E3 好, 但此方法好想到, 細節少, 且 '用 update 了之後的(nums[i], I)' 這點還是有點巧的, 所以以後還是用此方法, E3 方法只作參考, 看一看.

History:
E1 一次通過.
E2 一次通過.

E3 十分鍾寫好, 一次通過. E3 是用的 sliding window, 時間為 O(N), 空間為 O(k), 空間比 key 中方法好,但沒有 key 中方法那樣直觀好懂, 所以以後還是用 key 中方法, 但最好也看看我 E3 的方法. E3 的方法為: 將以 nums[i]為右端的 長度為 k+1 的 sliding window 放入 Set 中, 若有此 Set 中有跟 nums[i]重復的, 則返回 true. 實際寫的時候, 還有一些細節要注意, 所以 E3 能十分鍾寫好並一次通過, 還是有點 NB 的.

我: 我的代碼一次通過, 方法跟 William 的一樣, 但我的代碼略囉嗦, 以下是我根据 William 精簡後的我的代碼, 改後跟 William 的基本一樣, 但我的代碼看起來要順眼些, 以下用我的精簡後的代碼. William 說的 Time ~ O(N), Space ~ O(N), 故我的也一樣. 注意 map.containsKey()的時間一般是 O(1) (見 Java 書 834 頂的紅字).

E1 的代碼(用它):

```
 public boolean containsNearbyDuplicate(int[] nums, int k) {
  if(nums == null || nums.length == 0 || k <= 0)
   return false;

  Map<Integer, Integer> map = new HashMap<Integer, Integer>();

  for(int i = 0; i < nums.length; i++) {

   if(!map.containsKey(nums[i]))
    map.put(nums[i], i);
   else {
    if(i - map.get(nums[i]) <= k)
     return true;

    map.put(nums[i], i); //此時 map 中已有 nums[i]這個 key, 此處再用 map.put(nums[i], i), 其作用是 update nums[i]對應的 value(即 i). 但我們實際上只需要這個 update 了的 i, 因為我們是從左往右走的.
   }

  }

  return false;
 }
```

E3 的代碼(可以看看 , sliding window 法, 空間 O(k)):

```
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        if(nums == null || nums.length <= 1 || k <= 0) return false;
        Set<Integer> set = new HashSet<>();
        int n = nums.length;

        // 將最前面的 k+1 個元素(k+1 即 sliding window 長度)放入 set 中(set 即 sliding window), 此時只管放, 不用從 set 中刪.
        for(int i = 0; i < Math.min(k + 1, n); i++) {
            if(!set.contains(nums[i])) set.add(nums[i]);
            else return true;
        }

        // 若數組比 sliding window 還短, 則直接返回.
        if(n <= k + 1) return false;
```

```
        // 若數組比 sliding window 長, 則放元素時, 放一個之前, 要先從 set 中刪一個, 好保待 set(即 sliding window)的長度.
        for(int i = k + 1; i < n; i++) {
            set.remove(nums[i - k - 1]);
            if(!set.contains(nums[i])) set.add(nums[i]);
            else return true;
        }

        return false;
    }
```

220. Contains Duplicate III, Medium

http://wlcoding.blogspot.com/2015/05/contains-duplicate.html?view=sidebar

Given an array of integers, find out whether there are two distinct indices *i* and *j* in the array such that the difference between **nums[i]** and **nums[j]** is at most *t* and the difference between *i* and *j* is at most *k*.
Subscribe to see which companies asked this question

Key: 用 TreeSet 的 floor()和 ceiling()(Java p826). 對每一個 nums[i], 在數組 nums 中保持一個長度為 k 的 slide window, 這個 slide window 位於 nums[i]左邊且緊挨 nums[i]. 這個 slide window 中的元素都放入 TreeSet 中. 若 slide window 中有一個在[num[i] - t, num[i] + t] 中(用 floor()和 ceiling(),Java 書 p826), 則返回 true(它和 nums[i]的 index 差必小於 k), 否則向右滑動 slide window.

Corner case: 輸入數組的元素可能為 2147483647

History:
E1 直接看的答案.
E2 方法略有不同, 只取了 ceiling 或 floor, 沒同時取, 所以出現了 int 溢出, 後來花了點時日解決了, 通過.
E3 改了幾次後通過. E3 本來也想不起用 TreeSet 了, 還打算強行將 num[i]跟 slide window 中一個個比較, 後來想起在 leetcode 分類中, 本題題目後好像有個 TreeSet 一詞(想到時已刪, 刪了後一會兒想到的), 然後就按 TreeSet 做出來了.

BST + Slide Window: Time ~ O(NlogK), Space ~ O(K) (我: TreeSet 中查找元素是 O(log n), 見 Java 書 p822 底)
Maintain a TreeSet (red-black tree implementation) with size K.
For each num[i], check if there is a element in the TreeSet that is within [num[i] - t, num[i] + t] (using TreeSet.floor() andTreeSet.ceiling()).
If yes, return true, since the difference between index i and all the indices within the TreeSet will not exceed t;
If not, add num[i] to the TreeSet, and once the TreeSet exceeds its size K, we remove num[i - k], which is the furthest element to num[i] (keep a sliding window containing K elements of nums).
Note: The condition of TreeSet size exceeding K is i >= K instead of i > K.

```
public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {//index 的差不超過 k, 元素值的差不超過 t
```

```
    if (k < 1 || t < 0) return false;

     TreeSet<Integer> set = new TreeSet<>(); //實踐和 Java p786 都證明,本句也可寫成
Set<Integer> set = TreeSet<>(); 但不能用 set.floor(i), 若要用 floor, 則要
TreeSet<Integer> set = new TreeSet<>(); 因為 Set 中沒有 floor 函數, 詳見 Java p822 的表
    for (int i = 0; i < nums.length; i++) {
        int c = nums[i];
        if (set.floor(c) != null && set.floor(c) >= c - t
         || set.ceiling(c) != null && set.ceiling(c) <= c + t) {
            return true;
        } else {
            set.add(c);
            if (i >= k)  set.remove(nums[i - k]);
        }
    }
    return false;
}
```

221. Maximal Square, Medium

http://wlcoding.blogspot.com/2015/06/maximal-square.html?view=sidebar

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.
For example, given the following matrix:
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
Return 4.
Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: DP. 用 d(i, j)表示 以 matrix[i][j]為右下角端點的 maximal square' 的邊長. 遞推式是若 matrix[i][j]=='0', 則 d(i, j)=0, 否則 d(i, j) = min{d(i - 1, j - 1), d(i - 1, j), d(i, j – 1)} + 1 . 注意左邊界和上邊界 . 寫成 2-d DP 就可以了 .

Code Ganker 在 72. Edit Distance 中的 1d-DP 的寫法比 William 的 prev+curr 的 1d-DP 要好寫, 即遞推的左邊用 newRes[j],即遞推的左邊用 newRes[j], 即 newRes[j]=..., 而上一行的信自保存在 res 中. 我可以按 Code Ganker 的寫法來寫 1d-DP.

History:
E1 直接看的答案.

E2 直接按 1d-DP 寫的, 改了好多次才通過. 1-d DP 的難點在於 d(i-1, j-1)不能直接用一維 d 數組的 existing element 來表示, 要用兩個變量 prev 和 cur 來記錄和更新, 而且它們兩個的初始化也有一些 subtlety, 要非常細想才行. 在面試時應該想不對, 所以不建議 E3 按 1d-DP 做, 將 2d-DP 做熟練即可, 然後知道 1d-DP 的大概方法(即要知道 1d-DP 是可能的, 然後知道是用 prev 和 cur 兩個變量來實現的即可, 不必想細節).

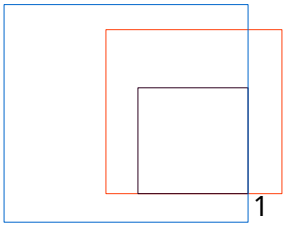
E3 按 2d-DP 寫的, 較快寫好, 改了兩次後通過. 一次是看錯了題目, 以為 matrix 的元素是 int 類型的, 其實是 char 類型的. 第二次是初始化 d[0][j]和 d[i][0]時忘了取 max(我是單獨初始化的 d[0][j]和 d[i][0], William

代碼沒這個問題).

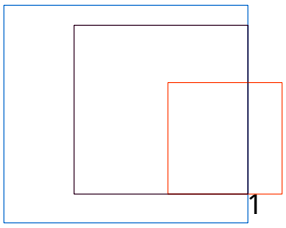
注意 matrix 的元素是 char, 而不是 int. Leetcode 的 test case 中輸入數組是{“11”, “11”}之類的, 但 leetcode 會自動將它轉換為 char[][].

我： 本題 William 給了 2-d DP 和 1-d DP，我掌握 2 -d DP 就可以了，因為 1-d DP 太繞，面試時不一定寫得出來．

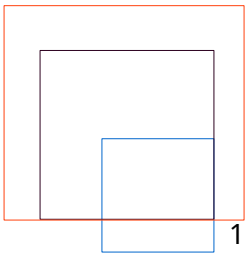
1. 2-d DP: Time ~ O(MN), Space ~ O(MN)
Let **d(i, j)** be the **length** (我: 即邊長) of the maximal square ending at matrix[i][j] (我： 即這個 square 的右下角端點就是 matrix[i][j]).
Fill up the table:
if matrix[i][j] == ' 0 ', d(i, j) = 0;
else if matrix[i][j] == ' 1 ',
d(0, j) = 1; 我： 即在左邊界的時候，此時 square 邊長只能是 1, 再往左就走不動了
d(i, 0) = 1; 我： 即在上邊界的時候，此時 square 邊長只能是 1, 再往上就走不動了
d(i, j) = min{d(i - 1, j - 1), d(i - 1, j), d(i, j – 1)} + 1. 我：原文中是 max, 我將其改為了 min, 好與代碼一致（代碼可通過）．為何要是 min? 以下我用圖解釋．黑色代表 square ending at matrix[i-1][j-1], 藍色代表 square ending at matrix[i][j-1], 紅色代表 square ending at matrix[i-1][j]. 比如第一種情況，即黑色的 d(i-1, j-1)為 min，此時右下角([i][j]處)為 1 時，必有 d(i, j) = d(i-1, j-1) + 1, 因為藍色和紅色的都比黑色的大，可以保證最右和最下的邊足夠長．另外兩種情況可類似想．



黑色的 d(i-1, j-1)為 min



紅色的 d(i-1, j)為 min



藍色的 d(i, j-1)為 min

Return: maxArea = (max{d(i, j)})^2. 我：別忘了這步

```
public int maximalSquare(char[][] matrix) {
    int m = matrix.length;
    if (m == 0) return 0;
    int n = matrix[0].length;

    int[][] d = new int[m][n];
    int max = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == '0')    d[i][j] = 0;
            else /* if  (matrix[i][j] == '1') */ {
                if (i == 0 || j == 0)
```

```
                    d[i][j] = 1;
                else
                    d[i][j] = Math.min(Math.min(d[i - 1][j - 1], d[i - 1][j]), d[i]
[j - 1]) + 1;
                }
                max = Math.max(max, d[i][j]);
            }
        }
        return max * max;//别忘了平方一下
}
```

2. 1-d DP (滚动数组) : Time ~ O(MN), Space ~ O(min{M, N})

```
public int maximalSquare(char[][] matrix) {
    int m = matrix.length;
    if (m == 0) return 0;
    int n = matrix[0].length;

    int[] d = new int[Math.min(m, n)];
    int max = 0, prev = d[0], curr = d[0];
    for (int i = 0; i < Math.max(m, n); i++) {
        for (int j = 0; j < Math.min(m, n); j++) {
            curr = d[j];//注意此d[j]是下面d[j]=Math.min...賦值前的，即此d[j]是在上一輪i
循環中賦值的
             int entry = m > n ? matrix[i][j] : matrix[j][i]; //entry用int類型居然能
通過, 用char也能通過(試過)
            if (entry == '0')    d[j] = 0;
            else /* if  (entry == '1') */ {
                if (i == 0 || j == 0)   d[j] = 1;
                else    d[j] = Math.min(Math.min(prev, d[j]), d[j - 1]) + 1;//左邊的
d[j]即為d[i][j], 右邊的d[j]即d[i-1][j], d[j-1]即d[i][j-1], prev即d[i-1][j-1]
            }
            prev = curr;
            max = Math.max(max, d[j]);
        }
    }
    return max * max;
}
```

222. Count Complete Tree Nodes, Medium

http://wlcoding.blogspot.com/2015/06/count-complete-tree-nodes.html?view=sidebar

Given a **complete** binary tree, count the number of nodes.
Definition of a complete binary tree from Wikipedia:
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h.
Subscribe to see which companies asked this question

Key: 遞歸. 題目的意思是樹的底層所有節點都靠左, 其餘每層都滿.
给定一个节点，找到达 leftmost node 和 rightmost node 的 depth，
如果相同，则表明该节点下的 tree 为 perfect tree，其节点数为 2^n - 1（ n = tree height ），这里

2^n 不用 Math.pow(2, n) 是因为其结果为 double, 而使用 bit manipulation.
如果不同，则遞歸調用自己, 分别求左、右 subtree 的 node 个数，相加再加 1 即为总数.

History:
E1 沒有專門考慮左右 depth 相等之情況, 而是直接遞歸, 結果超時了(代碼中我有詳說).
E2 幾分鍾寫好, 本來可以一次通過的, 結果移位時忘了減 1 和忘了加括號, 改後通過.
E3 較快寫好, 一次通過. 代碼和 William 的基本一樣.

我： heap 就是 complete tree 的一種（另外 max-heap 和 min-heap 還是有序的），參見 CLRS p172(可以不看).

William Tag: DFS

Recursion: Time ~ O(NlogN), Space ~ O(1)
Complete Tree:除底层外所有的节点都满（为 perfect tree），底层所有的节点都靠左。
方法：
给定一个节点，找到达 leftmost node 和 rightmost node 的 depth，

•如果相同，则表明该节点下的 tree 为 perfect tree，其节点数为 2^n - 1（ n = tree height ）；
•如果不同，则使用 Recursion 分别求左、右 subtree 的 node 个数，相加再加 1 即为总数。

Complexity:worst-case is only one node on the bottom level,
2 (logN + log(N-1) + ... + log2 + log1) ≈ NlogN.（我： 每個節點都要走到底，有的是 logN, 有的是 log(N-1), 有的是 log(N-2)...注意這只是 worst-case, 最好情況下可以達到 O(2logN)，即 perfect tree 的情況）
注意：这里 2^n 不用 Math.pow(2, n) 是因为其结果为 double (我：Avadöles, 己 record 到 Java 書)，而使用 bit manipulation: 1 << n 更快。

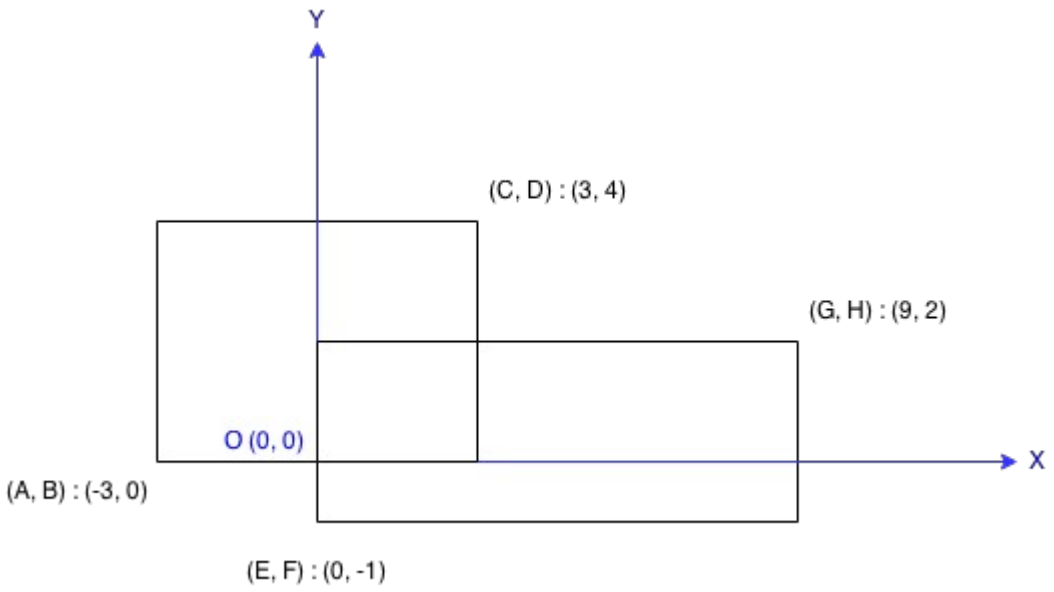
```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public int countNodes(TreeNode root) {
    int leftHeight = 0, rightHeight = 0;
    TreeNode left = root, right = root;
    while (left != null) {
        left = left.left;
        leftHeight++; //走到最底後（即 left 為葉子時），還要再執行一次循環，因為葉子也是有 left 的，只是這個 left 為 null. 所以最後得到的 leftHeight 是對的.
    }
    while (right != null) {
        right = right.right;
        rightHeight++;
    }
    if (leftHeight == rightHeight)  return (1 << leftHeight) - 1;//注意括號是必需的, 因為移位的優先級比加法還低, 已 record 至 Java 書
    else    return 1 + countNodes(root.left) + countNodes(root.right);//我的代碼就是直接用的這麼一句（再加 if(root == null)  return 0. 網上也有人這樣寫的當然他們也說超時），結果超時了. 注意 William 的代碼的好處在於，若 leftmost node 和 rightmost node 的 depth 相等，就不再
```

算（即不用再遞歸）了．而且被遞歸的 countNodes 也可以用這個 advantage, 所以效率比我的暴力遞歸要高多了．注意暴力遞歸並沒用到 complete tree 這個條件．
}

223. Rectangle Area, Easy

http://www.programcreek.com/2014/06/leetcode-rectangle-area-java/

Find the total area covered by two **rectilinear** rectangles in a **2D** plane.
Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.



Assume that the total area is never beyond the maximum possible value of **int**.
Credits:
Special thanks to @mithmatt for adding this problem, creating the above image and all test cases.
Subscribe to see which companies asked this question

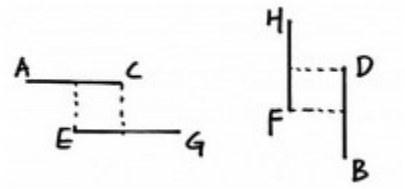
Key: 下面的圖可以不看, 只看題目中的圖. 若水平方向沒有 overlap, 則就沒有 overlap 的 area. 若豎直方向沒有 overlap, 也沒有 overlap 的 area. 注意矩形 ABCD 不是一定在矩形 EFGH 左邊, 也可以在它右邊. 若有 overlap, 則 right = Math.min(C,G), left = Math.max(A,E), top 和 bottom 類似. 此方法 E3 沒想出來, 可能是因為此方法不易想到, 所以要注意記憶.

History:

E1 直接看的答案.
E2 很快寫好並通過.
E3 居然沒做出來, 可能是因為 E3 是很晚做的, 太困了. E3 稍看了眼 key 後, 就自己寫出來了.

rectilinear: 直線的;直進的

我： William 的代碼（和方法）很囉嗦，不用他的代碼（這也說明每道題不要只看 William 的解答）. 網上的人都是用的以下解法.
This problem can be converted as a overlap internal problem. On the x-axis, there are (A,C) and (E,G); on the y-axis, there are (F,H) and (B,D). If they do not have overlap, the total area is the sum of 2 rectangle areas. If they have overlap, the total area should minus the overlap area.




```
public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {

    if(C<E||G<A )
        return (G-E)*(H-F) + (C-A)*(D-B); //可以定義一個 areaNoOverlap = (G-E)*(H-F) +
(C-A)*(D-B)

    if(D<F || H<B)
        return (G-E)*(H-F) + (C-A)*(D-B);

    int right = Math.min(C,G);
    int left = Math.max(A,E);
    int top = Math.min(H,D);
    int bottom = Math.max(F,B);

    return (G-E)*(H-F) + (C-A)*(D-B) - (right-left)*(top-bottom);
}
```

224. Basic Calculator, Medium

https://leetcode.com/discuss/39553/iterative-java-solution-with-stack

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.

Subscribe to see which companies asked this question

Key: E3 的方法:
先按沒有括號寫好, 方法是用兩個變量 pre 和 cur, 若讀到+-號, 就開始算, 比如算以下式子:
123 + 678 - 256
當讀到那個減號時, pre 和 cur 的值為:
123(pre) + 678(cur) -(此時讀到此減號) 256
此時(即讀到那個減號時) 就执行 pre = pre + sign * cur, 其中 sign 為 1(之前設好的), 然後將 sign 設為-1(因為此時讀到的是減號), 並將 cur 清零, 好供下一次算的时候用.

若有括號, 則怎麼弄? 此時要用一個 stack, 來保存'('左邊的結果 以及緊挨'('左邊的加減號. 即若算以下式子:
2 + 3 + (77 - 12) + 46
讀到'('時, 在 stack 中放入 2+3 的值(即 5), 然後放入 1(表示加號), 然後將 pre, cur, sign 這三個清零(當然 sign 是清為 1). 這樣做的意思就是: 將'('前的結果存入 stack 中, 然後重新開始算括號中的內容, 相當於將 括號中的內容 當作一個新的表達式來算.

讀到')'時, 將括號內的表達式的值賦給 cur, 將 stack 中的 5 取出來並將其賦給 pre, 將 stack 中的 1 取出來並將其賦給 sign. 即有:
5 (pre) + (sign) 65 (cur) + 46,
這樣又可以按照沒有括號時之情況算了.

當 i 等於 s.length() (不是 s.length()-1) 時, 再算一次 pre = pre + sign * cur, 然後返回 pre, 即最終結果.

History:
E1 直接看的答案.

E2 一次通過, 但我在自己的編譯器上測試了半天各種 case, 改了幾次才弄出來,搞了好一陣子, 寫出的代碼居然跟下面的基本一樣, 只有處理')'時稍有點不同, 連最後處理 s 以數字結尾之情況都一樣, 盡管下面的英文也有些細節沒寫.

E3 花了一會兒在 OJ 上寫好, 改了幾次後才通過. E3 其本上將本題方法忘完了, 所以是自己寫出來的, 後來發現代碼跟以下答案差不多, 但 E3 的代碼好像要好懂點. 以後用 E3 代碼. E3 還重新寫了 key, 因為 E2 的 key 就是叫我看下面的英文.

我：William 此題是和 227. Basic Calculator II 共用的代碼, which is lengthy and 不好懂. 我在網上看了好幾個人的代碼, 就以下的最清新好懂.

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:

1.digit: it should be one digit from the current number (即數字不只是一位的, 比如可以為 324)(Character.isDigit()見 Java 書 p151). 並且用 sign = 1 或-1 來表示符號.

2.'+': number is over, we can add the previous number and start a new number

3.'-': same as above

4.'(': push the previous result and the sign into the stack, set result to 0, just calculate the new result within the parenthesis.

5.')': 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 parenthesis. We add them together.

Finally if there is only one number (我: 即輸入只有一個數, E2 還發現這實際上還處理了 s 以數字結尾之情況(如 s="2+34"), 其實也包括了輸入只有一個數之情況), from the above solution, we haven't add the number to the result, so we do a check see if the number is zero.

以下代碼不用:

```
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)){ //Character.isDigit()見 Java 書 p151
            number = 10 * number + (int)(c - '0');
        }else if(c == '+'){
            result += sign * number; //此 number 是該+號前的 number, 此 sign 是該 number 前的 sign
            number = 0; //清零 number, 下一輪循環會重新讀入給 number 賦值
            sign = 1; //表示本 c 是+號, 留著後面用
        }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); //此 sign 是 ( 前的那個 sign

                //reset the sign and result for the value in the parenthesis

                sign = 1;   //此 sign 是 ( 後的那個 sign, 必為+

                result = 0; //注意不是 number=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 result calculated before the parenthesis

            } //注意空格就自動跳過了(還有點巧)

        }

        if(number != 0) result += sign * number; //若輸入只有一個數(E2 還發現這實際上還處理了 s 以數字結尾之情況(如 s=”2+34”), 其實也包括了輸入只有一個數之情況)

        return result;

}
```

E3 的代碼(用它):

```
    public int calculate(String s) {
        if(s == null || s.length() == 0) return 0;
        s = s.trim();
        LinkedList<Integer> stack = new LinkedList<>();

        int cur = 0;
        int pre = 0;
        int sign = 1;

        for(int i = 0; i <= s.length(); i++) {
            if(i == s.length()) {
                pre = pre + sign * cur;
                return pre;
            }

            char c = s.charAt(i);

            if(c == ' ') {
                continue;
            } else if(Character.isDigit(c)) {
                cur = cur * 10 + (int) (c - '0');
```

```
        } else if(c == '+' || c == '-') {
            pre = pre + sign * cur;
            cur = 0;
            sign = (c == '+' ? 1 : -1);
        } else if(c == '(') {
            stack.push(pre);
            stack.push(sign);
            pre = 0;
            cur = 0;
            sign = 1;
        } else if(c == ')') {
            pre = pre + sign * cur;
            cur = pre;
            sign = stack.pop();
            pre = stack.pop();
        }

    }

    return pre;
}
```

225. Implement Stack using Queues, Easy

http://wlcoding.blogspot.com/2015/06/implement-stack-using-queues.html?view=sidebar

Implement the following operations of a stack using queues.

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

Notes:

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

Update (2015-06-11):
The class name of the **Java** function had been updated to **MyStack** instead of Stack.
Credits:
Special thanks to @jianchao.li.fighter for adding this problem and all test cases.
Subscribe to see which companies asked this question

Key: 本題其實沒那麼簡單．用兩個 queue: q1 和 q2. q2 用來存 stack 中最有權出來的那個元素(要 保持 q2 中只有一個元素. 如何保持? 後面會說), q1 用來存剩下的元素. q1 和 q2 一起, 總共構成一個 stack. 跟 232

題一樣, 先寫 peek()(本題叫 top()), 再寫其它的. 寫的順序: empty, top, pop, push. 寫 top 時, 當 q2 為空時怎麼辦? 答: 將 q1 入口處的元素(即剛放进去的元素, 我將它叫 A)弄出來, 但 q1 是 queue, 不是 stack, 怎麼弄? 方法是: 將 q1 出口處的元素移到 q1 入口處, 移 q1.size()-1 次, 則 A 就在出口處了. 寫 push 時, 當 push 到 q2 後, q2 中的元素不只一個時應該怎麼辦? 答: 當 q2 中有不只一個元素時, 將多餘的元素加入到 q1 入口處, 好保持 q2 中只有一個元素.

History: E1 幾分鍾寫好並一次通過, 但我只用的一個 queue. E2 通過, 用的兩個 queue 但代碼比此處的囉嗦不少.

我: 看我說的就可以了, William 說的比較囉嗦, 不用看.

我: 我的代碼幾分鍾寫好並一次通過, 但我只用的一個 queue, 實現 pop 的方法是 revese queue->poll queue->reverse queue, 這樣雖然通過了, 但時間效率應該稍低. 以下用 William 的代碼, William 和網上其它人都用的兩個 queue. 一個 queue(q2)存 stack 中最有權出來的那個元素, 另一個 queue(q1)存剩下的元素. q1 和 q2 一起, 總共構成一個 stack.

William:

Time: push() ~ O(1), pop() ~ O(N), top() ~ O(1) and O(N) in worst-case (if pop() is called twice), empty() ~ O(1)
Space ~ O(N)
Use two Queues: q2 stores the latest element (only one), and q1 stores the rest elements.

•**Push():** enqueue elements to q2, and if there're more than one elements in q2, then dequeue them to q1;
•**pop():** 1) do top() first. If q2 is empty (if pop() is continuously called more than twice), then move the last element in q1 to q2;
2) dequeue the element in q2.
•**top():** 1) if q2 is empty, then circulate q1 to move its last element to front, and dequeue it to q2;
2) return q2.peek();
•**empty():** MyStack is empty iff both q1 and q2 are empty.

```
class MyStack {
    private Queue<Integer> q1 = new LinkedList<>(); //為何要這樣寫?見 Java 書 p806
    private Queue<Integer> q2 = new LinkedList<>(); // store the latest element

    // Push element x onto stack.
    public void push(int x) {
        q2.add(x);
        while (q2.size() > 1)//當 q2 中有不只一個元素時, 將多餘的元素加入到 q1 入口處, 好保持 q2 中只有一個元素.
            q1.add(q2.poll());
    }

    // Removes the element on top of the stack.
    public void pop() {
        top();  // if q1 is not empty but q2 is empty, need update q2 first. 注意調用本類中函數的寫法
        q2.poll();
    }
```

```
    // Get the top element.
    public int top() {
        if (q2.isEmpty()) { // move the last element in q1 to q2
//以下是想將 q1 入口處的元素(即剛放进去的元素, 我將它叫 A)弄出來, 但 q1 是 queue, 不是 stack, 怎麼弄? 以下的方法是將 q1 出口處的元素移到 q1 入口處, 移 q1.size()-1 次, 則 A 就在出口處了. 注意這樣弄完了後, 其它元素在 q1 中還是原來的順序.
            for (int i = 0; i < q1.size() - 1; i++)//此 for 自動解決了 q1 為空的情況.也可以像 232. Implement Queue using Stacks 中那樣用 while
                q1.add(q1.poll());
            q2.add(q1.poll());
        }
        return q2.peek();
    }

    // Return whether the stack is empty.
    public boolean empty() {
        return q1.isEmpty() && q2.isEmpty();
    }
}
```

226. Invert Binary Tree, Easy

https://leetcode.com/problems/invert-binary-tree/

Invert a binary tree.

```
   4
 /  \
 2   7
/\   /\
1  3 6  9
```

to

```
   4
 /  \
 7   2
/\   /\
9  6 3  1
```

Trivia:

This problem was inspired by this original tweet by Max Howell:

Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

(Tao: if I could meet such fucking easy problem, I will be in Google)

Subscribe to see which companies asked this question

Key: DFS. 直接遞歸調用 invertTree(root.left), invertTree(root.right), 然後交換 root.left 和 root.right, 交換時使用 invertTree 的返回值.

History:

E1 幾分鐘寫好, 一次通過. 我用的 BFS, 比 William 的 DFS 稍長. 要掌握 William 的 DFS, 我自己的 BFS 也要看熟.

E2 DFS 幾分鐘寫好, 一次通過.

E3 DFS 兩分鍾寫好, 一次通過.

我: 我在網上查了下, 題目中那句話的意思是 Homebrew 的开发者是 Max Howell, 他面 Google 被拒了, 理由是: 虽然我们 90% 工程师都在用你写的软件（Homebrew），但你不能在白板上反转二叉树，所以滚蛋.

William:

William Tag: DFS

Time ~ O(N), Space ~ O(N) (我覺得 Space 應該是 O(log N))

```
public TreeNode invertTree(TreeNode root) {
    if (root == null)    return root;
    TreeNode left = invertTree(root.left);
    TreeNode right = invertTree(root.right);
    root.left = right;
    root.right = left;
    return root;
}
```

我的 BFS 代碼, 方法是將每個結點的左右子結點互換:

```
 public TreeNode invertTree(TreeNode root) {
  Queue<TreeNode> queue = new LinkedList<TreeNode>();
  queue.offer(root);

  while(!queue.isEmpty()) {
   TreeNode cur = queue.poll();

   if(cur != null) { //因為葉子的 children(null)也會被放进 queue 中
    //以下是交换 cur.right 和 cur.left
    TreeNode temp = cur.right;
    cur.right = cur.left;
    cur.left = temp;

    queue.offer(cur.left);
    queue.offer(cur.right);
   }

  }

  return root;
 }
```

227. Basic Calculator II, Medium

http://blog.csdn.net/xudli/article/details/46644317

Implement a basic calculator to evaluate a simple expression string.
The expression string contains only **non-negative** integers, `+`, `-`, `*`, `/` operators and empty spaces . The integer division should truncate toward zero.
You may assume that the given expression is always valid.
Some examples:
"3+2*2" = 7
" 3/2 " = 1

" 3+5 / 2 " = 5
Note: Do not use the eval built-in library function.
Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: E3 的方法:
方法跟本題的 I 類似, 先按沒有乘除寫好, 方法是用兩個變量 pre 和 cur, 寫法跟 I 一樣. 若有乘除, 則用一個 stack, 來保存 乘除式子(如下面的 77*12) 左邊的結果 以及緊挨 乘除式子 左邊的加減號, 即若算以下式子:
2 + 3 + 77 * 12 + 46

讀到*時, 在 stack 中放入 2+3 的值(即 5), 然後放入 1(表示加號). 要用一個 boolean 變量 multDiv 來做括號之作用, 即當 multDiv 為 true 時 表示已进入 乘除式子, 當 multDiv 為 false 時 表示已出 乘除式子. 算乘除式子時, 還要用一個變量 preOperator 來表示之前的乘除號.

當 乘除式子 開始時, 將 pre(即左邊加減運算之結果)和 sign 放入 stack 中, 好開始算乘除. 當 乘除式子 結束時, 將 stack 中的 pre 和 sign 取出來, 將它們與 乘除之結果 一起作加減運算.

當 i 等於 s.length() (不是 s.length()-1) 時, 要先判斷 乘除式子 是否已結束, 若已結束(即最右的運算為加減), 則再算一次 pre = pre + sign * cur, 然後返回 pre, 即最終結果. 若乘除還沒結束(即最右的運算為乘除), 則要先將那個乘除式子算完後, 再跟它之前的結果一起再算一次 pre = pre + sign * cur, 然後返回 pre, 即最終結果.

History:
E1 沒做出來.
E2 通過, 但寫了較長時間.
E3 在 OJ 上沒多久寫好, 改了幾個語法錯誤, 然後再改了一次後通過. 原因是當讀到乘除號時, 忘了更新 preOperator (見代碼中 A 處), 加上此句後即通過. 由於 E3 的代碼跟本題的 I 一致, 且只需一次遍歷, 且比以下代碼好懂, 故以後用 E3 的代碼.

以下代碼的 key:
pass 兩遍, 第一遍, 先解決乘除, 第二遍, 做加減. 做乘除時, 在 stack 中放入乘除號(數子化)和數字, 然後 pop 出來算, 弄成 23(opl) *(op) 12(cur)的樣子, 先算好乘除, 將結果放入 stack 中, 同時該 stack 中也要放入加減號(第一遍 pass 完後別忘了 reverse stack), 好給第二遍 pass 時用. 第二遍 pass 中算加減的方法和第一遍中算乘除相似.

以下代碼不用:
```
public class Solution {
    public int calculate(String s) {
        if(s==null || s.length()==0) return 0;

        LinkedList<Integer> list = new LinkedList<Integer>(); //以下是將此 list 當 stack 用. 注意不能寫成 List<Integer> list = new LinkedList<Integer>(), 若單獨寫這樣一句, 則沒錯. 但若寫這樣一句, 並在後面使用 list.peek(), 則 compiler 報錯說找不到 peek 函數. 由此可以 peek 不是 List 中的, 而是 LinkedList 中的.

        //第一遍 pass, 做乘除.
        for(int i=0; i<s.length(); i++) {
            char c = s.charAt(i);
            if(Character.isDigit(c)) {
```

```
            int cur = c-'0'; //沒有 cast 成 int,  不是好習慣
            while(i+1<s.length() && Character.isDigit(s.charAt(i+1))) {
               cur = cur * 10 + s.charAt(i+1) - '0';
               ++i; //第一次看到 Java 程序員還用++i, 明顯是受 C++影响
            }//end of while
            if(!list.isEmpty() && (list.peek() == 2 || list.peek()==3)) {
              //以下意思為: 23(opl) *(op) 12(cur)
               int op = list.pop();
               int opl = list.pop(); //opl 中的 l 表示 left
               int res = 0;
               if(op==2) res = opl * cur;
               else res = opl / cur; //注意不是 cur / opl
               list.push(res);
            } else {
               list.push(cur); //若輸入為"1-2", 則 for 循環結束後, list 為[1, 1(表示減), 2]
            }
         } else if(c==' ') continue;
         else {
            switch (c) {
               case '+': list.push(0);
               break; //別忘了 break
               case '-': list.push(1);
               break;
               case '*': list.push(2);
               break;
               case '/': list.push(3);
               break;
               default: return -1;
            }
         }
      }

      if(list.isEmpty()) return 0; //此句是然並卵, 刪了一樣能通過

     //第二遍 pass, 做加減. 原代碼以下用的 poll, 有點 confusing, 我將其改為了 pop, 改後能通過.
      Collections.reverse(list); //為何要 reverse? Think of 輸入為"1-2"之情況, 此時 list 為[1, 1(表示減), 2],
按若無此 reverse, 則以下的代碼算的就是 2-1, 而不是 1-2

      int res = list.pop(); //這裡是重新定義的一個 res, 跟前面的不同
     //reverse 後, list 為[2, 1(表示減), 1], 以下的 res=1, op=1, opr=2
      while(!list.isEmpty()) {
         int op = list.pop();
         int opr = list.pop();
         if(op==0) res += opr; //opl 中的 l 表示 right
         else res -= opr;
      }
      return res;
   }
```

```
}
```

E3 的代碼(用它):

```
    public int calculate(String s) {
        if(s == null) return 0;
        s = s.trim();
        int n = s.length();
        if(n == 0) return 0;
        int pre = 0, cur = 0, sign  = 1;
        char preOperator = ' ';
        boolean multDiv = false;

        LinkedList<Integer> stack = new LinkedList<>();

        for(int i = 0; i <= n; i++) {

            if(i == n) {
                if(multDiv) {
                    cur = (preOperator == '*' ? pre * cur : pre / cur);
                    sign = stack.pop();
                    pre = stack.pop();
                }

                pre = pre + sign * cur;
                return pre;
            }

            char c = s.charAt(i);
            if(c == ' ') continue;
            else if(Character.isDigit(c)) {
                cur = cur * 10 + (int) (c - '0');
            } else if(c == '+' || c == '-') {
                if(multDiv) {
                    multDiv = false;
                    cur = (preOperator == '*' ? pre * cur : pre / cur);
                    sign = stack.pop();
                    pre = stack.pop();
                }
                pre = pre + sign * cur;
                cur = 0;
                sign = (c == '+' ? 1 : -1);
            } else if(c == '*' || c == '/') {
                if(!multDiv) {
                    multDiv = true;
                    stack.push(pre);
                    stack.push(sign);
                    pre = cur;
                    cur = 0;
                    preOperator = c; //A 處. E3 最開始忘了寫此句.
```

```
                continue;
            } else {
                pre = (preOperator == '*' ? pre * cur : pre / cur);
                cur = 0;
                preOperator = c;
            }

        }

    }

    return pre;
}
```

228. Summary Ranges, Easy

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"].
Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 用 cur 掃描輸入數組, 用 l 表示 range 的左端, r 表示 range 的右端. 當不連續的時候, 就得到 l 和 r, 並調用自己寫的 getRange(l, r)將 range 放进結果中. 注意處理邊界情況.

History:
E1 一次通過.
E2 試了幾次才通過.
E3 幾分鍾寫好, 改了兩次才通過: 一是寫 nums[i] != nums[i + 1] - 1 時, 不小心忘了寫-1. 二是 StringBuilder sb = new StringBuilder(Integer.toString(nums[l]))一句寫為了 StringBuilder sb = new StringBuilder(nums[l]), 注意後者不是將將 nums[l]轉化為 String 然後初始化給 sb, 而是指 sb 這個 StringBuilder 的 capacity 是 nums[l]. 已 record 到 Java p416. E3 代碼比 William 的好懂好寫, 以後用 E3 代碼.

William 的方法跟我的差不多, William 的代碼是 Time ~ O(N), Space ~ O(1), 所以我 E1 複雜度應該也是這麼多. William 有一句話比較有用:

William:
注意：这里判断不能颠倒顺序（if difference > 1 ... else ...），否则会出错。对如下例子：[-2147483648, 2147483647]，因为 difference 为负，导致判断为一个 range 的错误。
建议：判断条件时应用明确的情况（e.g., difference == 1)。

以下是我的代碼:

E1 代碼(不用):
```
 public List<String> summaryRanges(int[] nums) {
  List<String> ranges = new ArrayList<String>();

  if(nums == null || nums.length == 0) //leetcode test case 還真有輸入為[]的情況
   return ranges;
```

```
 int l = nums[0]; //l 是小寫字母 el

 for(int i = 1; i <= nums.length; i++) {

  int cur = (i < nums.length ? nums[i] : Integer.MAX_VALUE);

  if(cur - nums[i - 1] != 1) {
   int r = nums[i - 1];
   ranges.add(getRange(l, r));
   l = cur;
  }

 }

 return ranges;
}

private String getRange(int from, int to) {
 return from == to ? Integer.toString(from) : from + "->" + to;
}
```

E3 代碼(用它):

```
    public static List<String> summaryRanges(int[] nums) {
        List<String> res = new ArrayList<>();
        if(nums == null || nums.length == 0) return res;
        int start = 0;

        for(int i = 0; i < nums.length; i++) {
            if(i == nums.length - 1 || nums[i] != nums[i + 1] - 1) {
                res.add(getRange(nums, start, i));
                start = i + 1;
            }
        }

        return res;
    }

    public static String getRange(int[] nums, int l, int r) {
        if(l == r) {
            return Integer.toString(nums[l]);
        } else {
            StringBuilder sb = new StringBuilder(Integer.toString(nums[l]));
            sb.append("->");
            sb.append(nums[r]);
            return sb.toString();
        }
    }
```

229. Majority Element II, Medium

http://wlcoding.blogspot.com/2015/03/majority-element.html?view=sidebar

Given an integer array of size *n*, find all elements that appear more than ⌊ n/3 ⌋ times. The algorithm should run in linear time and in O(1) space.
Hint:

1.How many majority elements could it possibly have?
2.Do you have a better hint? Suggest it!

Subscribe to see which companies asked this question

Key: majority 最多有兩個. 所以用兩個 majority, 兩個 count, 代碼基本上是 169. Majority Element 的 double copy. 由於題目中沒說一定有 majority(或多少個 majority) (169 題說的一定有), 所以最後還要驗證一下這兩個 majority 到底是不是 majority(再遍歷一次數組, 數出現的個數). 注意 majority1 和 majority2 都初始化為 0. if (count1 == 0) {...} else if (count2 == 0 && majority1 != nums[i]) {...} else if...

History:
E1 的代碼幾分鍾寫好並一次通過, 但是用的 HashMap.
E2 用以上 Key 中的方法, 沒通過. 原因是因為我對 Moore's voting algorithm 本來理解就不深. 在 Key 中加了幾句, 希望以後能寫出來.
E3 沒做出來.

我: 我的代碼幾分鍾寫好並一次通過(將 169. Majority Element 中我的代碼稍加改動), 但是用的 HashMap, 記錄每個數出現的個數. 網上說了, 時間為 O(n), 空間也為 O(n). 注意題目要求空間為 O(1), 雖然我的代碼也能通過. 以下用 William 的代碼, 是改进的 Moore's voting algorithm(網上其他人也是這樣做的), 空間為 O(1). 由於我對 Moore's voting algorithm 理解不深, 所以最好也熟悉一下我的 Hash Table 做法(代碼在最後).

Time ~ O(N), Space ~ O(1)
Modify Moore's voting algorithm: use two variable **num1** and **num2** (since there are at most two possible majority elements, 我: 因為題目說 majority 至少要出現⌊ n/3 ⌋ + 1 次, 若有 3 個 majority, 則總元數個數比 n 還多了, 因為 3 * (⌊ n/3 ⌋ + 1) = 3 * (n/3 + 1) > 3 * (n/3) = n), and two counter **count1** and **count2**; check if num1 and num2 is the one-third majority in the end.

```
public List<Integer> majorityElement(int[] nums) {
    List<Integer> list = new ArrayList<>();
    int n = nums.length;
    if (n == 0) return list;

    int num1 = 0, num2 = 0; //num1 和 num2 都是 majority
    int count1 = 0, count2 = 0;
    for (int i = 0; i < n; i++) {//注意是從 0 開始, 因為 num1 和 num2 初始化與 169 題不同
        int val = nums[i];
        //if (count1 == 0) {//此句為 William 的原代碼. E3 時發現在 OJ 上通不過(William 網站上也有人說通不過. 我將其加了一個 num2 != val 後, 即通過, 如下句所示)
        if (count1 == 0 && num2 != val) {
            num1 = val;
            count1 = 1;
        } else if (count2 == 0 && num1 != val) {
            num2 = val;
            count2 = 1;
```

```
        } else if (num1 == val) {
            count1++;
        } else if (num2 == val) {
            count2++;
        } else /* if (num1 != val && num2 != val) */ {
            count1--;
            count2--;
        }
    }

    if (isOneThirdMajority(num1, nums)) list.add(num1);
    if (num1 != num2 && isOneThirdMajority(num2, nums)) list.add(num2);
    return list;
}

//最後要驗證一下 num1 和 num2 到底是不是 majority
private boolean isOneThirdMajority(int val, int[] nums) {
    int n = nums.length;
    int count = 0;
    for (int i = 0; i < n; i++)
        if (nums[i] == val) count++;
    //以下幾句應該可以簡寫為 return count > n / 3;
    if (count > n / 3)  return true;
    else                return false;
}
```

230. Kth Smallest Element in a BST, Medium

http://wlcoding.blogspot.com/2015/07/kth-smallest-element-in-bst.html?view=sidebar

Given a binary search tree, write a function kthSmallest to find the **k**th smallest element in it.
Note:
You may assume k is always valid, 1 ≤ k ≤ BST's total elements.
Follow up:
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?
Hint:

1.Try to utilize the property of a BST.
2.What if you could modify the BST node's structure?
3.The optimal runtime complexity is O(height of BST).

Credits:
Special thanks to @ts for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 遞歸+Binary Search. 寫一個 size(TreeNode x)函數, 用 DFS 求以 x 為根的樹的節點數. size 遞歸調用 size, kthSmallest 中要調用 size, 也要調用 kthSmallest 自己.
Binary Search Tree 的性質是: x 左子樹所有結點的值 <= x 的值 <= x 右子樹所有結點的值

History:
E1 沒做出來.
E2 很快寫好, 一次通過, 代碼跟 William 差不多.
E3 較快寫好, 一次通過, 代碼跟 William 的方法二(Binary Search)基本一樣. 由於此方法中用了兩次(層)遞歸, 我寫好後還以為會超時, 結果一次通過了, 且發現 William 也是這樣寫的.

William Tag: 1) DFS (Inorder traversal) 2) Use BST size

我: 以下有三種方法, 我要掌握的是第二種(Binary Search). 另外兩種也看看(注意第二種方法的時間複雜度沒有第一種小, 所以第一種方法也要看熟, 其實第一種方法還要簡單些). 第三種方法正是題目中 Hint 所要求的, 只能寫偽代碼, 但也要看熟.

1. Inorder Traversal: Time ~ O(N), Space ~ O(N)
Inorder traversal of BST yields an increasing order.

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

private Queue<TreeNode> q = new LinkedList<>(); //注意雖然用了 Queue, 但本方法仍然是 DFS, 只是將 DFS 中序遍歷得到的 node 序列存入 q 中, 好留到後面用而已.

public int kthSmallest(TreeNode root, int k) {
    if (k == 0)         throw new IllegalArgumentException("k is zero!");
    inorder(root);
    if (q.size() < k)   throw new IllegalArgumentException("k exceeds tree size!");

    for (int i = 0; i < k - 1; i++) {
        q.poll();
    }
    return q.poll().val;
}

//inorder(x)的作用為: 將以 x 為根的子樹的所有節點 以中序遍歷的順序 放入 q 中
private void inorder(TreeNode x) {
    if (x == null)  return;
    if (x.left != null) inorder(x.left);
    q.add(x);
    if (x.right != null) inorder(x.right);
}
```

2. Binary Search: Time ~ O(NlogN), Space ~ O(1) (Tao: skeptical, 遞歸棧難道不佔用空間嗎?)

(我: 本方法直接看代碼, 不用看 William 囉嗦的)
Binary search based on the size of each left node, where the size is the number of subtree nodes (including the root itself):

•if n + 1 == k, the kth node is the root;
•if n + 1 < k, the kth node is in the right subtree;
•if n + 1 > k, the kth node is in the left subtree.

Binary search takes O(logN); obtaining size takes O(N). 實際上應該用 Master Theorem.

```
public int kthSmallest(TreeNode root, int k) {
    int n = size(root.left);
    if (n + 1 == k) {
        return root.val;
    } else if (n + 1 < k) {
```

```
        return kthSmallest(root.right, k - n - 1);
    } else /* if (n + 1 > k) */ {
        return kthSmallest(root.left, k);
    }
}

private int size(TreeNode x) {
    if (x == null)  return 0;
    return size(x.left) + size(x.right) + 1;
}
```

3. Modify BST node's structure

我: William 的代碼已省, 因為完全是亂來, 跟網上其它人的做法完全不同, leetcode 中通不過, 改了幾次後還是通不過. 以下用網上其它人的代碼. 由於本方法要改 TreeNode 的結構, which is intrinsic to leetcode, 所以沒法改, 只能寫偽代碼. 以下我帖出網上兩個人的偽代碼(他們的方法是一樣的). 本方法的本質還是 Binary Search, 只是 size()函數相當於是已經自帶了.

第一個人的:

(來自 http://www.jyuan92.com/blog/leetcode-kth-smallest-element-in-a-bst/)

If we can change the BST node structure, We can add a new Integer to mark the number of element in the left sub-tree.

when the node is not null.

1.If `k == node.leftNum + 1`, return node
2.if `k > node.leftNum + 1`, make `k -= node.leftNum + 1`, and then `node = node.right`
3.otherwise, `node = node.left`

The time complexity of algorithm above will be `O(h)` (我: 因為 `height h` 和 `binary search` 的時間複雜度都等於 `logN)`, h is the height of the input tree.

第二個人的:

(來自 http://www.fgdsb.com/2015/01/19/kth-smallest-node-in-bst/)

如果想更优化效率，可以引入 order statistic tree。
Order statistic tree 是一种 BST 的变种，往每个 node 里面加上一个 field 来存左子树的节点数，在树平衡时可以达到 O(logN)复杂度。
这种树在很多题目中都可以用到。
伪代码（摘自 GeeksforGeeks）：

```
start:
if K = root.leftElement + 1
    root node is the K th node.
    goto stop
else if K > root.leftElements
    K = K - (root.leftElements + 1)
```

```
7       root = root.right
8       goto start
9   else
10      root = root.left
11      goto start
12  stop:
```

231. Power of Two, Easy

http://wlcoding.blogspot.com/2015/07/power-of-two.html?view=sidebar

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

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

Subscribe to see which companies asked this question

Key: 2 的 power 的二進製 必定是 100..00 這樣的數, 即首位是 1, 其餘是 0. 所以可以先將這個數右移到只剩 1, 再左移同樣的次數, 若還跟原數相等, 就是 2 的 power.

William 還給了幾鍾方法, 也要看看(他的方法 3 很巧, E2 表示: 我靠, 好巧, 要記住!).

Convention: 0 和負數都不算 power of two.

History:
E1 幾分鍾寫好並通過.
E2 三分鍾寫好, 一次通過, 且代碼比 E1 略簡明, 所以以後用 E2 的代碼.
E3 在 OJ 上兩三分鍾寫好, 本來可以一次通過的, 結果我對 n=0 為負數時的對題目的理解不對, 導致沒通過. 改後即通過. 由於是 convention 的問題, 故本題還是算我一次通過了.

我 E2 的代碼(用它):

```
public boolean isPowerOfTwo(int n) {
    if(n < 1) return false;

    int m = n;
    int count = 0;

    while(m > 1) {
        m >>>= 1;
        count++;
    }

    m <<= count;

    return m == n;
}
```

William:

Power of two must be 100...000 (one follows by all zeros).

1. Time ~ O(2N) where N is the bit length, Space ~ O(1)
Create 100...000 and compare with n (need to get the bit length first). 我: 方法跟我的一樣, 代碼沒我的看起來順眼, 故代碼已省.

2. Time ~ O(N), Space ~ O(1)
Check each bit after 1 if it's zero.

```
public boolean isPowerOfTwo(int n) {
    if (n <= 0) return false;
    while (n > 0) {
         if (n != 1 && (n & 1) != 0) return false;    // stop if any bit after 1 is not 0
        n >>= 1;
    }
    return true;
}
```

3. Time ~ O(1), Space ~ O(1)
Use property: n & (n - 1) == 0.
Example: n = 1000, n - 1 = 111, 1000 & 0111 = 0000.
Note: Be careful of the edge case when n <= 0.

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

232. Implement Queue using Stacks, Easy

http://wlcoding.blogspot.com/2015/06/implement-stack-using-queues.html?view=sidebar

Implement the following operations of a queue using stacks.

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

Notes:

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

Subscribe to see which companies asked this question

Key: 看我說的就可以了, William 說的比較囉嗦, 不用看. 方法很巧, 就是用兩個 Stack, 放東西 (push)時放入 stack1, 拿東西時(peek/pop)從 stack2 拿. 若 stack2 為空, 則將 stack1 中所有的元素都弄到 stack2 中去(在 peek()中弄的).這其實相當於是將 stack1 的棧底和 stack2 的棧底連通了, 整個成了一個 Queue. 先寫 peek(), 再寫其它的.

History: E1 沒寫出來. E2 一次通過, 代碼跟 William 的一模一樣, 標點都一樣.

William:

Time: push() ~ O(1), pop() ~ O(N), top() ~ O(1) and O(N) in worst-case, empty() ~ O(1)
Space ~ O(N)

Use two Stacks: push new elements to stack1, and pop old elements from stack2; if stack2 become empty, pop all the elements in stack1 to stack2.

- **Push():** push elements to stack1;
- **pop():**
  1) do peek() first. If stack2 is empty, then pop all the elements in stack1 to stack2 (the elements on the top in stack2 are the older ones);
  2) pop the element in stack2.
- **Peek():**
  1) if stack2 is empty, then pop all the elements in stack1 to stack2;
  2) return stack2.peek();
- **empty():** MyQueue is empty iff both stack1 and stack2 are empty.

```
class MyQueue {
    private Stack<Integer> stack1 = new Stack<>();
    private Stack<Integer> stack2 = new Stack<>();  // store the old element

    // Push element x to the back of queue.
    public void push(int x) {
        stack1.push(x);
    }

    // Removes the element from in front of queue.
    public void pop() {
        peek(); // if stack2 is empty, need to update stack2 first
        stack2.pop();
    }

    // Get the front element.
    public int peek() {
        if (stack2.isEmpty()) { // move all the elements in stack1 to stack2
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }

    // Return whether the queue is empty.
    public boolean empty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```

233. Number of Digit One, Medium

http://blog.csdn.net/xudli/article/details/46798619

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.
For example:
Given n = 13,
Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.
Hint:

1.Beware of overflow.

Subscribe to see which companies asked this question

Key: 本題之關健在於求出公式. 以算百位上 1(即 比 n 小的所有數 的總共的 百位上 1 的個數)为例子: 分 n 百位上是 0, 1, 和 >=2 三种情况: n=3141092, n=3141192, n=3141592. 然後求出最終公式. 可將 n 分成兩段想, 如 a= 31410, b=92, 公式是用 a 和 b 表示出來的. 為了防止 overflow, 要將有的數定義為 long.

History:
E1 的代碼是直接將數字轉化為 String, 然後暴力數 1 的個數, 在 leetcode 中果然超時.
E2 改了幾次後通過, E2 是按 key(即分 0, 1, >=2)的分法做的, 但公式是自已想的(即只看了 key, 沒看以下的文字), 實際上我 E2 沒寫出一個統一的公式, 而是用的 if-else, 但跟以下的統一公式等價. E2 還把 key 看錯了, 對於 n=3141092, E2 的 a=3141, 而不是 key 中那樣 a=31410, E2 雖然也做出來了, 但這樣會更難(或不可能)寫出統一公式. 以後注意別看錯了. 以後最好是將以下統一公式記住, 若忘記了, 也可以像 E2 這樣寫成 if-else.
E3 沒做出來, 主要是因為忘得太厲害了.

網上的人(包括 William)用的方法基本都一樣, 就是求出那個公式. 以下的解說看起來最好懂. William 說的 Time ~ O(logN), Space ~ O(1), 所以以下代碼的複雜度也一樣.

[思路]

reference: https://leetcode.com/discuss/44281/4-lines-o-log-n-c-java-python

intuitive(我: 本句話不要太當回事, 對理解後面的幫助不大): 每 10 个数, 有一个个位是 1, 每 100 个数, 有 10 个十位是 1, 每 1000 个数, 有 100 个百位是 1. 做一个循环, 每次计算单个位上 1 得总个数(个位,十位, 百位).

例子:

以算百位上 1(我: 即 比輸入數字小的所有數 的總共的 百位上 1 的個數, 這是注釋 A)为例子: 假设(輸入數字的)百位上是 0, 1, 和 >=2 三种情况:

- case 1(我: 即輸入數字 n 的百位上是 0): n=3141092, a= 31410, b=92. 计算百位上 1 的个数(我: 看注釋 A)应该为 3141 *100 次(我: 因為比 n 小的且百位為 1 的數可以是 31401**, 31391**, 31381**,... 1**, 從 0 到 3140 總共是 3141 個, 而每個的**又有 100 個可能(00, 01, ...99), 所以是 3141*100).
- case 2(我: 即輸入數字 n 的百位上是 1): n=3141192, a= 31411, b=92. 计算百位上 1 的个数(我: 看注釋 A)应该为 3141 *100 + (92+1) 次(我: 31401**, 31391**, 31381**,... 1**, 總共是 3141 個, 而 31411**總共是 92+1 個, 即 3141100, 3141101, 3141102, … 3141192).

- case 3(我: 即輸入數字 n 的百位上>=2): n=3141592, a= 31415, b=92. 计算百位上 1 的个数(我: 看注釋 A)应该为 (3141+1) *100 次(我: 31401**, 31391**, 31381**,... 1**, 總共是 3141 個, 而 31411**總共是 100 個, 即 3141100, 3141101, 3141102, … 3141199).

以上三种情况可以用 一个公式概括:

`(a + 8) / 10 * m + (a % 10 == 1) * (b + 1);` 我: 若求百位 1 的個數, 則 m=100, 其餘類似.

[CODE]

```
public class Solution {
    public int countDigitOne(int n) {
        int ones = 0;
        for (long m = 1; m <= n; m *= 10) { //若將 m 和 a, b 改為 int, 則對 n=1410065408 通不過, 這是因為循環結束時 m > n, 而此時的 m 越界了. 另注意 m *= 10 不要寫成了 m * 10, 因為必須是一個 statement
            long a = n/m, b = n%m;
            ones += (a + 8) / 10 * m;
            if(a % 10 == 1) ones += b + 1;
        }
        return ones;
    }
}
```

234. Palindrome Linked List, Easy

http://www.programcreek.com/2014/07/leetcode-palindrome-linked-list-java/

Given a singly linked list, determine if it is a palindrome.
Follow up:
Could you do it in O(n) time and O(1) space?
Subscribe to see which companies asked this question

Key: We can use a fast and slow pointer to get the center of the list, then reverse the second list and compare two sublists. 注意反轉第二條鏈的方法, 此處用的是與 206. Reverse Linked List, Easy 中不同的方法, 時間和空間複雜度不變, 但為了練習, 我還是用本題反轉的方法. 反轉的方法即為將 list 中每一個箭頭方向反轉(以下 123456 是要反轉的 list)
1->2p, 3c->4n->5->6 (前面已讓 2 指向 1)
1->2<-3p, 4c->5n->6
Conventions: 若 head 為 null, 或只有一個 node, 則也算是 palindrome.

History:
E1 直接看的答案.
E2 改了一次後通過, 第一次沒通過是因為將 reverse 函數的返回類型寫成了 void, 所以在主函數中反轉了第二個 list 後 忘了 update 新的 head2.
E3 較快寫好, 本來可以一次通過, 但不小心把 if-else 寫成了兩個 if, 結果第一個 if 滿足了的還是要进入第二個 if, 使得在第二個 if 中出現了 null, 改後即通過, 以後要多加小心類似情況, 若以後看不懂我這裡在說甚麼, 則不管. E3 也用的反轉後一半的方法, 其中 reverse 函數調用的 206. Reverse Linked List 中的我 E3 的 iterative 方式的 reverse 函數.

我: 以下給了兩種方法, 我要掌握的是第二種(最好看我帖在最後的 E2 的代碼, 算法一樣, 但代碼要清楚些),

但第一種也要看看.

Java Solution 1

We can create a new list in reversed order and then compare each node. The time and space are O(n). (我: 我也想過這個方法, 但覺得這樣效率應該不高, 所以就沒寫)

```
public boolean isPalindrome(ListNode head) {
    if(head == null)
        return true;
//以下是將 list 反轉, prev 的名字取得有點 misleading. 其
實就是新建了一個鏈表, 用來保存反轉後的 list, prev 其實就是
這個新建鏈表的頭. 反轉的方法就是不停地將 list 中的節點值拷
到新鏈表的最前面
    ListNode p = head;
    ListNode prev = new ListNode(head.val);

    while(p.next != null){
        ListNode temp = new ListNode(p.next.val);
        temp.next = prev;
        prev = temp;
        p = p.next;
    }
//以下是比較原 list 和反轉後的 list 是否相等
    ListNode p1 = head;
    ListNode p2 = prev;

    while(p1!=null){
        if(p1.val != p2.val)
            return false;

        p1 = p1.next;
        p2 = p2.next;
    }

    return true;
}
```

Java Solution 2

We can use a fast and slow pointer to get the center of the list, then reverse the second list and compare two sublists. The time is O(n) and space is O(1). (我: William 和很多其它人就是用這個方法)

```
public boolean isPalindrome(ListNode head) {

    if(head == null || head.next==null)
        return true;

    //find list center
    ListNode fast = head;
    ListNode slow = head;

    while(fast.next!=null && fast.next.next!=null){
        fast = fast.next.next;
```

```
            slow = slow.next;
        }
//若 list 有奇數個元素, 如:
1->2->3->4->5,
則最後為
1->2->3(slow)->4->5(fast)

若 list 有偶數個元素, 如:
1->2->3->4->5->6,
則最後為
1->2->3(slow)->4->5(fast)->6

//以下是將 list 一分為二
        ListNode secondHead = slow.next;
        slow.next = null;

        //reverse second part of the list
//以下是反轉第二條鏈, 注意是反轉是 inplace 的, NB!
        ListNode p1 = secondHead;
        ListNode p2 = p1.next;

        while(p1!=null && p2!=null){
            ListNode temp = p2.next;
 //若第二條鏈為 1->2->3->4->5->6, 則
 //上一句使得:1(p1)->2(p2)->3(temp)->4->5->6
            p2.next = p1;
 //上一句使得:1(p1)->2(p2), 3(temp)->4->5->6
                    ^       |
                    |_______|

            p1 = p2;
            p2 = temp;
 //上兩句使得:  1  -> 2(p1), 3(p2)->4->5->6
                    ^       |
                    |_______|
下一輪到這個時, 就成了
                    1  ->  2 <-3(p1), 4(p2)->5->6
                    ^       |
                    |_______|
最後一輪到這裡時, 就成了
                1  ->  2 <-3 <-4 <-5 <-6(p1) <-null(p2)
                ^       |
                |_______|
        }//end of while


        secondHead.next = null;
//上一句使得: null<-1<-2<-3<-4<-5<-6(p1)<-null(p2)

        //compare two sublists now
        ListNode p = (p2==null?p1:p2);
        ListNode q = head;
        while(p!=null){ //由前面知, 當原 list 有偶數個節點時,
最後弄出的兩條鏈長度一樣. 當原 list 有奇數個節點時, 第一條鏈
```

```
比第二條鏈多一個節點(如 1->2->3, 4->5), 此時只要比較第二條
鏈那麼多個節點就可以了(即第一條鏈的最後一個節點不用比)
        if(p.val != q.val)
            return false;

        p = p.next;
        q = q.next;

    }

    return true;
}
```

E2 的代碼:

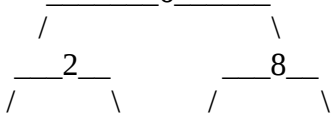
```
  public boolean isPalindrome(ListNode head) {
    if(head == null || head.next == null)
      return true;

    if(head.next.next == null) {
      return head.val == head.next.val;
    }

    ListNode walker = head;
    ListNode runner = head;

    // 123w 21r
    // 123w 32r1
    while(runner.next != null && runner.next.next != null) {
      walker = walker.next;
      runner = runner.next.next;
    }

    // break up
    ListNode head1 = head;
    ListNode head2 = walker.next;
    walker.next = null;

    head2 = reverse(head2);

    // 123 12
    // 123 123

    // Compare the two lists
    while(head2 != null) {
      if(head1.val != head2.val)
        return false;

      head1 = head1.next;
      head2 = head2.next;
    }
```

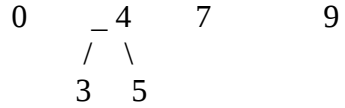
```
    return true;
}

// reverse
private ListNode reverse(ListNode head) {
    if(head == null || head.next == null) // The list has 0 or 1 node
        return head;

    ListNode pre = head;
    ListNode cur = pre.next;
    ListNode next = cur.next;

    while(next != null) {
        cur.next = pre;
        pre = cur;
        cur = next;
        next = next.next;
    }
   //1p->2c->3n->4->5->6
    //1p->2c->1, 3n->4->5->6, for the followings, 2 always points to 1, and I will omit this notation.
    //1->2p, 3c->4n->5->6
    //1->2<-3p, 4c->5n->6
    //1->2<-3<-4p, 5c->6n
    //1->2<-3<-4<-5p, 6c n=null, loop stops

    cur.next = pre;
    head.next = null;

    //1->2<-3<-4<-5p<-6c n=null
    //null<-1<-2<-3<-4<-5p<-6c n=null

    return cur;
}
```

235. Lowest Common Ancestor of a Binary Search Tree, Easy

http://wlcoding.blogspot.com/2015/08/lowest-common-ancestor-of-tree-i-ii.html?view=sidebar

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”





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

Key: DFS. 遞歸調用 lowestCommonAncestor 自己. 若 p 和 q 都比 root 大，則它們都在 root 的右子樹, 則可以在以 root.right 為根的子樹中繼續搜; 若 p 和 q 都比 root 小，則它們都在 root 的左子樹, 則可以在以 root.left 為根的子樹中繼續搜；若 root 在 p 和 q 之間, 則 root 必為 p 和 q 的 lowest common ancestor(證明見我下面的講解).

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過.
E3 雖然一次通過, 但把本題寫麻煩了. 方法跟 key 中方法一樣. 但在判斷 p 在 root 的左子樹還是右子樹時, 我是判斷的 p 是否在 root.left, 以及 p 是否在 root.right 中, which 也是用遞歸做的. 我沒想到可以直接比較 p 和 root 的大小, 腦殼短路了.

我: 以下 William 給了兩個 code, 第一個囉嗦些, 不要浪費時間去看, 直接看第二個. 以下 William 的講解也不用看(只看下時間和空間複雜度就可以了), 看我下面的講解.

DFS: Time ~ O(logN) (我: 因為每次丟一半, 跟 binary search 類似), Space ~ O(logN)
Going from top down to find p and q, the LCA is the node when split happens.
If root.val is larger than both p.val and q.val, then go left;
If root.val is smaller than both p.val and q.val, then go right;
otherwise, there is a split.

我: 以下 code 不看 !

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if      (p.val == q.val)        return p;
    else if (p.val > q.val)         return lowestCommonAncestor(root, q, p);
    else /* if (p.val < q.val) */ {
        if      (root.val < p.val)  return lowestCommonAncestor(root.right, p, q);
        else if (root.val > q.val)  return lowestCommonAncestor(root.left, p, q);
        else                        return root;
    }
}
```

我: 看以下 code

我的講解: 注意 binary search tree 的性質是左子樹中的值全部<=root, 右子樹中的值全部>=root (見 CLRS p308). 所以如果將這個樹中序遍歷, 則是: ....(比 root 小的)..., root, ...(比 root 大的).... 比 root 小的那些全剖在左子樹, 比 root 大的那些全部在右子樹.

所以對於本題:

1. 若 p 和 q 都比 root 大，則它們都在 root 的右子樹, 則可以在以 root.right 為根的子樹中繼續搜;

2. 若 p 和 q 都比 root 小，則它們都在 root 的左子樹, 則可以在以 root.left 為根的子樹中繼續搜;

3. 若 root 在 p 和 q 之間, 則可知兩點: 第一, root 必為 p 和 q 的 common ancestor, 因為若不是, 則必是這樣一個 scenario: 設 helo 是 p 和 q 的某個 common ancestor(一定存在, 比如最高那個根), 而 p 和 q 在 helo 的一側(如左子樹), root 在 helo 的另一側(如右子樹), 而這種 scenario 中, root 的值都不可能在 p 和 q 之間. 第二, root 必為 p 和 q 的 lowest common ancestor, 因為 此時 p 和 q 在 root 的兩側(左子樹和右子樹), 而 root 的兩側是沒有公共節點的, 所以 p 和 q 不會有別的 common ancestor.

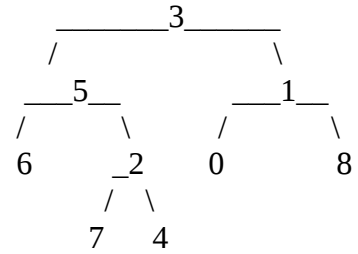
The code can be simplified as follows (there is no need to compare p.val and q.val):

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if      (root.val < p.val && root.val < q.val)
       return lowestCommonAncestor(root.right, p, q);
    else if (root.val > p.val && root.val > q.val)
       return lowestCommonAncestor(root.left, p, q);
    else
       return root;
}
```

236. Lowest Common Ancestor of a Binary Tree, Medium

http://wlcoding.blogspot.com/2015/08/lowest-common-ancestor-of-tree-i-ii.html?view=sidebar

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”



For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. Another example is LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition. Subscribe to see which companies asked this question

Key: DFS. 遞歸調用 lowestCommonAncestor 自己. E3 表示本 key 對該方法理解得不好, 以下黃字理解得好些. lowestCommonAncestor 函數的作用是找 lowest common ancestor, 但也可以根据 lowestCommonAncestor(root, p, q)返回的是否為 null 來判斷以 root 為根的子樹是否同時含有 p 和 q. 本題方法是: 對 root.left 調用 lowestCommonAncestor, 對 root.right 調用 lowestCommonAncestor, 若結果都是 null, 則說明 root 不是 LCA(當然前提是 root 自己不是 p 或 q) (E3 表示: 若 root.left 樹含 p, roo.right 樹含 q 時, root 就是 LCA 啊, 所以看以下黃字好些); 若對 root.left 的結果不是 null, 則顯然 LCA 在 root.left 子樹中, 此時返回這個結果; 若對 root.right ....; 若對 root.left 和 root.right 的結果都不是 null, 則 root 必為 LCA, 因為此時 root.left 和 root.right 兩人必定是 p 和 q 兩人, 不然 不可能 結果都不是 null, 此時顯然 root 就是 LCA.

E3 說的: 本題的解法(William 的代碼, which is the same as the leetcode discussion solution)要完全理解還挺難的. 該代碼對以下情況

```
    1
   / \
  2   3
```

lowestCommonAncestor(2, 2, 3)給出的結果是 2, 我 E2 的代碼給出的結果也是 2, 這顯然不對, 但它們都能通過, 原因是題目中說了: find the lowest common ancestor (LCA) of two given nodes in the tree. 即題目的 test case 中要求 lowestCommonAncestor(root, p, q)中 p 和 q 都在以 root 為根的樹中, 對於不在樹中的情況不 test. 但是我們在寫代碼時要考慮不在樹中的情況, 因為 lowestCommonAncestor 在調用自己時會出現這種情況. 所以本代碼其實是本題的一個解析延拓, 即 lowestCommonAncestor(root, p, q)的功能比題目要求要多, 其完整功能為以下. 以下的都是不變量, 在檢查 lowestCommonAncestor(root, p, q)的功能時, 也要檢查以下三個功能是不是全有.
若 p, q 都不在 root 為根的樹中, 則返回 null. E2 代碼中, 此時只有(left == null && right == null)這一個可能.
若 p, q 都在以 root 為根的樹中(即題目的情況), 則返回 p, q 的 LCA. E2 代碼中, 此時只有(left != null && right == null)和(left == null && right != null) 兩種可能.
若 p, q 有一個在以 root 為根的樹中, 則返回在樹中的那個, 比如 , 若 p 在以 root 為根的樹中 則返回 p, 若 q 在以 root 為根的樹中 則返回 q. E2 代碼中, 此時有(left != null && right == null), (left == null && right != null) , 和(left == null && right == null)三種可能.

History:
E1 直接看的答案.
E2 很快寫好, 但最開始沒有將遞歸調用的結果保存在變量中, 在其它地方重復調用了, 所以超時了, 改後通過.
E3 改了幾次後通過, 但代碼比較囉嗦, 我寫了一個函數 boolean hasNode(TreeNode root, TreeNode cur)來判斷 cur 是否在以 root 為根的子樹中, hasNode 通過遞歸調用自己來實現的. 這樣顯然沒 William 的好.

DFS: Time ~ O(N), Space ~ O(N) (我: 我覺得 Space 應該是 O(logN), 235 題就是)
Recursively search for p an q from bottom up, return p or q if founded, otherwise return null.
If left and right both return null (both left and right subtree doesn't contain p and q), then return null;
If left returns p or q (left subtree contain p or q) and right returns null (right subtree doesn't contain p and q), then return left;
If right returns p or q (right subtree contains p or q) and left returns null (left subtree doesn't contain p and q), then return right;
If both left and right return p or q (both left and right subtree contains p or q), then the root is the LCA.

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) return root;
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);
    return left == null ? right : (right == null ? left : root);
}
```

我: 上面代碼中的 return 不是很容易看清楚, 我 E2 是按以下寫的, 以下幾句跟上面的 return 一句等價, 但更好懂:

```
if(left == null && right == null)
   return null;
else if(left != null && right == null)
```

```
        return left;
    else if(left == null && right != null)
        return right;
    else
        return root;
```

後面的題, William 就沒有解答了. Goodbye, William!

237. Delete Node in a Linked List, Easy

http://www.changyu496.com/?p=162

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.
Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function.
Subscribe to see which companies asked this question

Key: 把传入节点(設叫 A)的后面一个节点(設叫 B)的值赋给自己(即 A)，然后把自己后面的节点(即 B)删掉即可. 網上其他人也是這樣做的. 注意題目中說了 except the tail (即最後一個非 null 節點). 注意輸入的是 ListNode, 而不是 value. 第一次看到 list 的題目的輸入居然不是 head. 本題代碼只有兩句, 應該是 leetcode 中代碼最短的.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過.
E3 一分鍾寫好, 一次通過, 代碼跟以下的兩行代碼一模一樣.

我: 本題文字看以下的, 代碼用最後那個短的.

翻译一下：
我们需要这样一个函数：从一个单链表中删除指定元素(不能是尾部, 我: 即最後一個非 null 節點, 因為在這種輸入下, 尾部根本沒法刪, 詳見代碼後我說的)，这个函数的参数只有这个被删除的节点。
假设这个单链表为 1 -> 2 -> 3 -> 4 。然后你传入的参数节点为第三个，值为 3 的那个，执行了这个函数之后，这个链表将变为 1 -> 2 -> 4
开始做这道题的时候，我有点懵，只传入一个被删除的节点？有那么一瞬间我都觉得这道题是不是出错了？哈哈
不过稍微冷了一下，想了一下，这个传入的是一个节点，不是值。而节点本身可是有前后关系的呀，而且这道题算是很基础的数据结构知识了吧。
通过一次“李代桃僵”就可以了。以上面的题目中例子做解释吧
1 -> 2 -> 3 -> 4
cur next
我得到 3 这个节点作为参数的时候，把这个节点的指针命名为 cur，然后也可以得到 4 这个节点，把这个节点的指针命名为 next，这个时候，我也可以把 3 这个节点的值改为 4，这时链表变为
1 -> 2 -> 4 -> 4
cur next
这时，接着把 cur 当前的 next 指针直接指向 next 的 next 指针。也就是说之间删掉 next 的这个节点即可。

这样对于这个链表就变为
1 -> 2 -> 4
结束了！
其实简单来说就是把传入节点的后面一个节点的值赋给自己，然后把自己后面的节点删掉即可。
当然也要注意一下异常检查，代码非常非常简单，这道题也非常非常基础，更是一种思维能力上的考验其实，而我是使用 Java 来写的，所以可以先不用考虑节点的“删除”问题。

我: 以下代碼不用

```
/**
* Definition for singly-linked list.
* public class ListNode {
* int val;
* ListNode next;
* ListNode(int x) { val = x; }
* }
*/
public class Solution {
public void deleteNode(ListNode node) {
if(node==null||node.next==null) return; //最後一個節點不刪
ListNode temp = node.next;
node.val = temp.val;
node.next = temp.next;
}
}
```

我: 以下的代碼也能通過, 用以下代碼(來自 http://www.programcreek.com/2014/07/leetcode-delete-node-in-a-linked-list-java/):

```
public void deleteNode(ListNode node) {
    node.val = node.next.val;
    node.next = node.next.next;
}
```

以上的代碼若在我電腦上測試, 若輸入為最後一個非 null 節點, 則報錯: java.lang.NullPointerException. 但在 leetcode 中卻能通過, 說明 leetcode 的 test case 中確實沒有最後一個非 null 節點, 跟題目中說的 except the tail 一致. 本題明顯是先有解答了, 再根据解答來編的題目.

238. Product of Array Except Self, Medium

http://www.programcreek.com/2014/07/leetcode-product-of-array-except-self-java/

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

Subscribe to see which companies asked this question

Key: 兩次遍歷. 第一次遍歷從後往前, 使得 result[i]等於 nums[i]之後的元素(不包括 nums[i])乘積. 第二次遍歷從前往後, 把 nums[i]之前的元素(不包括 nums[i])乘到 result[i]上去. 注意題意不是要 in-place, 而是要新建一個數組 result.

Hisotry:
E1 直接看的答案.
E2 通過, 代碼比以下的略囉嗦.
E3 沒做出來(暈), 原因是卡在一個地方了: 第一次遍歷時, E3 寫的是 result[i] = nums[i + 1] * result[i + 1], 居然沒想到將 nums 的 i+1 改為 i, 即 result[i] = nums[i] * result[i + 1], E3 看到答案此句後, 後面(第二次遍歷)全是自己寫的, 寫好後一次通過. 第二次遍歷時用的 left 變量跟答案不謀而合, 且第二次遍歷的代碼跟答案一模一樣.

我: 網上其他人也是這個方法.

Space is O(1).

```
public int[] productExceptSelf(int[] nums) {

    int[] result = new int[nums.length];
    result[result.length-1] = 1;

    //以下的 for 使得 result[i] = nums[i+1] * nums[i+2] … * nums[result.length-1] (但 result[result.length-1]仍為 1)
    for(int i=nums.length-2; i>=0; i--) {
        result[i] = result[i+1] * nums[i+1];
    }

    int left = 1;
    //以下的 for 使得 result[i] = nums[1] * nums[2] * … nums[i-1] * nums[i+1] * nums[i+2] … *nums[result.length-1]
    for(int i=0; i<nums.length; i++) {
        result[i] *= left; //left = nums[1] * nums[2] * … nums[i-1]
        left *= nums[i];
    }

    return result;
}
```

239. Sliding Window Maximum, Hard

http://www.shuatiblog.com/blog/2014/07/27/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 considered.

Subscribe to see which companies asked this question

Key: 本題解答易看懂, 但自己寫不一定容易. 用 Deque(Java p806)來表示 window, Deque 中放入的是 nums 的 index. 注意 Deque 中的元素個數是不定的(如不理解, 可看下面 Analysis 下的第二段英文). 遍歷 nums, 對於 nums[i], 要保持兩點: 一是 Deque 中所有元素(若 Deque 中有一個數是 i, 則此處的"元素"是指 nums[i])都在以 nums[i]為右端的 window 中(其餘的刪之), 二是 Deque 中元素(注意是指 nums[i])是降序排列的(不要用 sort, 每次加入 nums[i]前將 Deque 中<=nums[i]的元素都刪掉(要 removeLast(), 而只有 Deque 有這個功能, Queue 不行)就可以了, 這樣刪的主要目的是為了很方便地維持 Deque 是降序的, 而保持降序的目的就是為了好刪, 因為小元素都在同一端去了). 這樣對於每一個 window (即 Deque), 這個 Deque 的首元素就是這個 window 的 max.
Corner case: nums 較短時, k 較小時

History:
E1 直接看的答案.
E2 通過, 但花了點時間.
E3 沒多久就寫好, 一次通過(NB! key 中還說自己寫不一定容易). E3 的方法沒 key 好, E3 的方法跟 key 之區別在於, E3 沒有將 Deque 弄成降序的, 而只是保持 Deque 的頭元素(getFirst()弄出來那個)為當前 window 中最大的. E3 方法是(若看不懂就算了, 沒帖 E3 代碼): 保持 Deque 的頭元素(getFirst()弄出來那個)為當前 window 中最大的(稱此元素為小明), Deque 中位於小明前面(getFirst 那個方向)的元素都刪掉(好保證 Deque 的頭元素為當前 window 中最大的). 每次往 Deque 尾部(addLast())加 nums 中元素. 當新加入 Deque 的元素大於 Deque 的頭元素時, 將 Deque 中 除新加入那個大元素外 的所有元素都刪掉(同樣是為了保證 Deque 的頭元素為當前 window 中最大的). 若新加入 Deque 一個元素後 Deque 的長度大於 k, 且新加入那個元素沒有 Deque 頭元素大, 則強行找出當前 Deque 中最大的元素(即一個一個比, 就因為這點, 此方法沒 key 的好), 然後將 Deque 中最大元素前面的都刪掉(同樣是為了保證 Deque 的頭元素為當前 window 中最大的). 每次將 Deque 頭元素加入結果數組中. E3 能一次通過, 還是有點 NB 的.

Analysis
The naive approach is using a Heap. This time complexity is O(n*logn). However, there is a better way using a (double-ended) queue.
We do not need to keep all numbers. For example, suppose numbers in a window of size 4 are (1, 3, -1, 2). Obviously, no matter what next numbers are, 1 and -1 are never going to be a maximal as the

window moving. The queue should look like (3, 2) in this case.

Solution
When moves to a new number, iterate through back of the queue, removes all numbers that are not greater than the new one, and then insert the new one to the back.
FindMax only need to take the first one of the queue.
To remove a number outside the window, only compare whether the current index is greater than the front of queue. If so, remove it.
A natural way most people would think is to try to maintain the queue size the same as the window's size. Try to break away from this thought and think out of the box.
Code
written by me
我: 原代碼是定義的一個純 LinkedList, 而且所有操作都是按 List 來的, 如 q.get(0), q.remove(q.size() - 1)等, 我將其全改為了 Deque 的定義和 Deque 的超作(Java p806), 改後可通過.

```
public int[] slidingWindowMax(int[] array, int w) { //注意函數名和參數名都和 leetcode 不同, 自己寫的時候改一下就是了


    if(array.length == 0) return array;//此句是我加的, 否則通不過
    int[] ans = new int[array.length - w + 1]; //先記住: 以 i 為右端的 window 的左端為 i-(w-1), 下面會再用. 故以最後一個元素(array.length-1)為右端的 window 的左端為(array.length-1) – (w-1) = array.length – w. 如此, 則窗口左端在 0 時產生一個 max, 在 1 時產生一個 max, … 在 array.length-w-1 時產生一個 max, 在 array.length-w 時產生一個 max, 所以總共 array.length-w+1 個 max
     Deque<Integer> q = new LinkedList<Integer>(); //注意 Deque 和 Queue 一樣, 都是 interface, 所以寫法和 Queue 一樣, 右邊都要用 LinkedList. 也可以寫為 LinkedList<Integer> q = new LinkedList<>() ← 以後按這樣寫, 已記入 Java p806
    // Queue stores indices of array, and values are in decreasing order.
    // In this way, the top element in queue is the max in window
    //例如: q = 5(head/top), 2, 7, 3 (bottom), 這些 5, 2, 7, 3 全是 array 的 index, 且 array[5]>array[2]>array[7]>array[3]. 「q 是降序 sort 了的」之原因見下面.
    for (int i = 0; i < array.length; i++) {
        // 1. remove element from head until first number within window. 即保持 q 中所有元素都在以 i 為右端的 window 中
        if (!q.isEmpty() && q.getFirst() + w <= i) { //即 q.getFirst() <= i-w. 注意以 i 為右端的 window 為[i-w+1, i-w+2, … i]. 注意由於是將 array 中的指标從左到右的順序(即從小到大)於入 q 中的, 所以 q 中於的那些 index 其實也是 sort 了的, 是按昇序 sort 了的
            // it's OK to change 'while' to 'if' in the line above
            // cuz we actually remove 1 element at most
            q.removeFirst();
        }
        // 2. before inserting i into queue, remove from the tail of the
        // queue indices with smaller value they array[i]
        //注意以下是 while, 而不是 if. 要保持不停地刪 q 中的尾元素, 直到 q 中的尾元素比 array[i]大, 由於 q 是降序 sort 了的(可用歸納法證之), 故實際上 q 中所有元素都比 array[i]大, 這也正保持了「q 是降序 sort 了的」這個不變量.
        while (!q.isEmpty() && array[q.getLast()] <= array[i]) {
            q.removeLast();
        }
        q.add(i);
        // 3. set the max value in the window (always the top number in
        // queue)
```

```
            if (i + 1 >= w) { //i+1=w 時, 以下 ans[i+1-w]中的 index i+1-w=0. 而由 for 中可以 i 的最大取值為 array.length, 故 ans[i+1-w]中的 index i+1-w 的最大取值為 array.length+1-w, 這就是前面定義 ans 時分配的空間.
                ans[i + 1 - w] = array[q.getFirst()];
            }
        }
        return ans;
}
```

240. Search a 2D Matrix II. Medium

http://blog.csdn.net/xudli/article/details/47015825

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.
Subscribe to see which companies asked this question

Key: 從右上角開始, 若 target 小於該格的數, 則向左走一步; 若 target 大於該格的數, 則向下走一步. 這可以用歸納法證明的: 若某格的數能用這個方法找到, 那麼它左邊的數和下邊的數也可以用這個方法找到. 或這樣說: 記某格的數為 A, '它左邊的數或下邊的數' 為 B, 則要找 B, 可以先找到 A, 而根据假設, A 是能用這個方法找到的, 設 A 已找到, 即此時光標已在 A 處, 那麼用這個方法顯然能找到 B. 下面原作者的解釋不嚴密, 可以不看. 實際做題時, 若想驗證此方法的正確性, 可以在數組中選一個數作為 target, 看能不能找到它.

History:
E1 直接看的答案.
E2 幾分鍾寫好, 一次通過.
E3 幾分鍾寫好, 本來可以一次通過的, 結果不小心在 j--和 i++中把 i 和 j 寫反了, 改後即通過.

我: 本題題意跟 74. Search a 2D Matrix 的區別在於, 74 題中 '本行第一個數' 比 '上一行最後一個數' 大, 而本題的數組沒有這個限制. 74 題是用二分查找, 跟本題其實沒有甚麼卵關係.

[思路]
这道题是经典题, 我在微软和 YELP 的 onsite 和电面的时候都遇到了.
从右上角开始, 比较 target 和 matrix[i][j]的值. 如果小于 target, 则该行不可能有此数, 所以 i++; 如果大于 target, 则该列不可能有此数, 所以 j--. 遇到边界则表明该矩阵不含 target.

我: 網上其他人基本上也都是用的這個方法. 他們的時間複雜度是 O(m+n), 本解答顯然也是.
[CODE]

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix.length==0 || matrix[0].length==0) return false;

        int i=0, j=matrix[0].length-1;

        while(i<matrix.length && j>=0) {
            int x = matrix[i][j];
            if(target == x) return true;
            else if(target < x) --j;
            else ++i;
        }
        return false;
    }
}
```

241. Different Ways to Add Parentheses, Medium

https://leetcode.com/discuss/48477/a-recursive-java-solution-284-ms

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are `+`,`-` and `*`.

Example 1
Input: `"2-1-1"`.
((2-1)-1) = 0
(2-(1-1)) = 2
Output: `[0, 2]`

Example 2
Input: `"2*3-4*5"`
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
Output: `[-34, -14, -10, -10, 10]`
Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.
Subscribe to see which companies asked this question

Key: 本題還是比較簡單的. 不要被本題的括號嚇到了, 注意輸入中是沒有括號的, 本題的括號就表示一個計算的先後順序, 可以簡單地用遞歸來實現. 掃描到某一個運算符處時, 遞歸調用 diffWaysToCompute 函數來算運算符 左邊的值 和 右邊的值, 然後再用這個運算符算得最後結果.

Corner cases: 輸入只有一個數, 沒有運算符

History:
E1 直接看的答案.
E2 沒多久就寫好了, 但忘了考慮'輸入只有一個數, 沒有運算符'之 corner case, 改後即通過.
E3 沒多欠就寫好了, 一遍通過.

我: 網上另一個人也是用的類似的遞歸法, 但代碼要長很多.

```
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {
        List<Integer> ret = new LinkedList<Integer>();
        for (int i=0; i<input.length(); i++) {
            if (input.charAt(i) == '-' || input.charAt(i) == '*' || input.charAt(i) == '+' ) {
                String part1 = input.substring(0, i);
                String part2 = input.substring(i+1);
                List<Integer> part1Ret = diffWaysToCompute(part1);
                List<Integer> part2Ret = diffWaysToCompute(part2);
                for (Integer p1 :   part1Ret) {
                    for (Integer p2 :   part2Ret) {
                        int c = 0;
                        switch (input.charAt(i)) {
                            case '+': c = p1+p2;
                                break;
                            case '-': c = p1-p2;
                                break;
                            case '*': c = p1*p2;
                                break;
                        }
                        ret.add(c);
                    }
                }
            } //end of if
        } //end of for
        if (ret.size() == 0) { //輸入只有一個數, 沒有運算符之情況. E3 沒想到這樣做, E3 是專門掃描的一遍.
            ret.add(Integer.valueOf(input)); //valueOf 見 java p405
        }
        return ret;
    }
}
```

242. Valid Anagram, Easy

Given two strings *s* and *t*, write a function to determine if *t* is an anagram of *s*.
For example,
s = "anagram", *t* = "nagaram", return true.
s = "rat", *t* = "car", return false.
Note:
You may assume the string contains only lowercase alphabets.

Follow up:
What if the inputs contain unicode characters? How would you adapt your solution to such case?
Subscribe to see which companies asked this question

Key: 用 hashmap，key 是字符，value 是出现的次数，如果两个单词构成的 hashmap 相同，那么就是 anagram。实现起来就是用一个构建 hashmap，然后另一个在前面的 hashmap 中逐个去除，最后如果 hashmap 为空，即返回 true.

History:
E1 很快寫好並通過. 網上基本上也是用的這個算法(Hash Table)或排序(Code Ganker 也提到了). 以下用我的代碼.
E2 很快寫好, 一次通過.
E3 幾分鍾寫好, 一次通過, 代碼跟 E1 基本一樣.

Anagram：单词里的字母的种类和数目没有改变，只是改变了字母的排列顺序。

我: Code Ganker 在 49 題中講的相關部分也抄過來:

Code Ganker:

这是一道很经典的面试题了，在 cc150 里面也有，就是把一个数组按照易位构词分类。易位构词其实也很好理解，就是两个单词所包含的字符和数量都是一样的，只是顺序不同。
这个题简单的版本是判断两个单词是不是 anagram，一般来说有两种方法。第一种方法是用 hashmap，key 是字符，value 是出现的次数，如果两个单词构成的 hashmap 相同，那么就是 anagram。实现起来就是用一个构建 hashmap，然后另一个在前面的 hashmap 中逐个去除，最后如果 hashmap 为空，即返回 true。这个方法时间复杂度是 O(m+n)，m，n 分别是两个单词的长度。而空间复杂度是 O(字符集的大小)。第二种方法是将两个单词排序，如果排序之后结果相同，就说明两个单词是 anagram。这种方法的时间复杂度取决于排序算法，一般排序算法是 O(nlogn)，如果字符集够小，也可以用线性的排序算法。不过总体来说，如果是判断两个单词的，第一种方法要直接简单一些。

```
public static boolean isAnagram(String s, String t) {
    Map<Character, Integer> map = new HashMap<>();

    for(int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if(!map.containsKey(c))
            map.put(c, 1);
        else
            map.put(c, map.get(c) + 1);
    }

    for(int i = 0; i < t.length(); i++) {
```

```
            char c = t.charAt(i);

            if(!map.containsKey(c))
                return false;
            else if(map.get(c) == 1)
                map.remove(c);
            else
                map.put(c, map.get(c) - 1);
        }

        return map.isEmpty();
    }
```

243. Shortest Word Distance, Easy, Locked

https://leetcode.com/discuss/50234/ac-java-clean-solution

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.
For example,
Assume that words = ["practice", "makes", "perfect", "coding", "makes"].
Given word1 = "coding", word2 = "practice", return 3.
Given word1 = "makes", word2 = "coding", return 1.
Note:
You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

Key: 簡單, 直接寫. 一次遍歷, 掃描到 word1 時記下它的 index, 掃描到 word2 時記下它的 index, 兩個相減即可. 這樣可以保證 word1 和 word2 是相鄰的(即它們之間不會有別的 word1 和 word2). 注意 words 中可能用重復的 word, 所以 distance 要求個最小值.

History:
E1 應該能很快做出來, 但由於 locked, 沒法 judge, 所以沒做, 直接看的答案.
E2 很快寫出來, 代碼跟以下的基本差不多, 在我電腦上能結出正確結果, 算通過.
E3 是將每個 word 和它所在的所有位置都放入 Map 中, 然後取出 word1 和 word2 的所有位置相減取最小值. 幾分鐘寫好, 一次通過(給出正確答案). 這其實就是本題 II 的方法, 但用在本題中, 顯然沒有答案中方法好, 故又默寫了一遍答案.

```
public int shortestDistance(String[] words, String word1, String word2) {
    int p1 = -1, p2 = -1, min = Integer.MAX_VALUE; //將 p1 和 p2 初始化為-1 之理由見下. E2 也是這樣寫的.

    for (int i = 0; i < words.length; i++) {
        if (words[i].equals(word1))
            p1 = i;

        if (words[i].equals(word2))
            p2 = i;

        if (p1 != -1 && p2 != -1) //注意 if 中的這個要求, 這是保證 p1 和 p2 都被賦了值的. 否則可能給出錯誤結果(比如 p1=0, p2=-1 時, 錯誤地返回 1)
```

```
            min = Math.min(min, Math.abs(p1 - p2));
        }

        return min;
    }
```

244. Shortest Word Distance II, Medium, Locked

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

For example,
Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “coding”, word2 = “practice”, return 3.
Given word1 = "makes", word2 = "coding", return 1.

Note:
You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

```
// Your WordDistance object will be instantiated and called as such:
// WordDistance wordDistance = new WordDistance(words);
// wordDistance.shortest("word1", "word2");
// wordDistance.shortest("anotherWord1", "anotherWord2");
```

以下的 key 是一個更好的理解的方法, 且與本題 I, III 的解法一致, 第二遍刷題時用以下方法:
http://sbzhouhao.net/LeetCode/LeetCode-Shortest-Word-Distance-II.html

Key: 用一個 Map<String, List<Integer>>來存 words[i]和 該單詞出現的所有位置, shortest(String word1, String word2) 中取出 word1 和 word2 對應的 list, 然後比較找出最小間隔. 找最小間隔的方法就是用最 straight forward 的, 兩層循環, 一個一個比.

History:
E1 直接看的答案.
E2 按以上方法, 很快寫好, 我電腦給出正確結果, 算通過.
E3 是將我本題 I 的代碼(Map 法)拷幾句到本題中, 然後通過(能給出正確結果, 且代碼跟以下答案差不多).

```
public class WordDistance {
    private Map<String, List<Integer>> map = new HashMap<>();
    public WordDistance(String[] words) {
        for (int i = 0; i < words.length; i++) {
            String s = words[i];
            List<Integer> list;
            if (map.containsKey(s)) {
                list = map.get(s);
            } else {
```

```
            list = new ArrayList<>();
        }
        list.add(i);
        map.put(s, list);
      }
    }
    public int shortest(String word1, String word2) {
      List<Integer> l1 = map.get(word1);
      List<Integer> l2 = map.get(word2);
      int min = Integer.MAX_VALUE;
      for (int a : l1) {
        for (int b : l2) {
          min = Math.min(Math.abs(b - a), min);
        }
      }
      return min;
    }
}
```

老方法(大概看看算法(代碼中黃色注釋)就可以了):

http://likesky3.iteye.com/blog/2236157

Key: 不要看下面原作者的中文分析, 那個說起看不懂, 看我的: 定義一個 HashMap, 在 constructor 中將每個單詞的位置放入 HashMap 中, 俱體的, 是將單詞作為 Map 的 key, 該單詞出現的所有位置(注意 words 中的單詞有重復的)組成一個 list 作為 Map 的 value. 在 shortest(String word1, String word2) 中取出 word1 和 word2 對應的 list, 然後比較找出最小間隔(找的算法其實也有難度的, 但說不清楚, 最好記住).

[分析]
在构造函数(我: 即 constructor)将各单词出现的所有位置保存在 HashMap 中，调用 shortest()方法时，从 HashMap 中获取两单词的位置列表，两个列表是已排序的，因此可使用类似 MergeSort 的思路求解最短距离。
[ref]
https://leetcode.com/discuss/50190/java-solution-using-hashmap

Java 代码

```
public class WordDistance {
    //以下 map 中的 String 是放 words 中的一個單詞, List<Integer>是放這個單詞在 words 中的位置, 由於 words 中的單詞可能有重復的, 所以用的 List.
    private HashMap<String, List<Integer>> indexer = new HashMap<String, List<Integer>>();
    //以下是 constructor
    public WordDistance(String[] words) {
        if (words == null) return; //constructor 中居然也可以 return
        for (int i = 0; i < words.length; i++) {
            if (indexer.containsKey(words[i])) { //如果 indexer 這個 Map 中含有 words[i], 則將 i 加到 indexer.get(words[i])這個 List 中去
                indexer.get(words[i]).add(i);
```

```
            } else { //如果 indexer 這個 Map 中沒有 words[i], 則新建一個 List 並加到 indexer 中去
                List<Integer> positions = new ArrayList<Integer>();
                positions.add(i);
                indexer.put(words[i], positions);
            }
        }
    }

    public int shortest(String word1, String word2) {
        List<Integer> posList1 = indexer.get(word1);
        List<Integer> posList2 = indexer.get(word2);
        int i = 0, j = 0;
        int diff = Integer.MAX_VALUE;
//以下的方法就是:
A A A A B B B B A A A A B B B B B
要在以上找 A 和 B 最小間隔(不是眼中看到的那個 nominal 間隔), 則在 B 左邊的所有 A 興 B 的間隔中取最小的, 然後在 A 左邊的所有 B 興 A 的間隔中取最小的, 然後求總的最小的. 本題算法其實很難, 不大好說清楚, 就是這麼個意思, 剩下的自己想.
        while (i < posList1.size() && j < posList2.size()) { //注意 i 是 List 中的指標, 而不是 words 中的指標
            int pos1 = posList1.get(i), pos2 = posList2.get(j);
            if (pos1 < pos2) {
                diff = Math.min(diff, pos2 - pos1);
                i++;
            } else {
                diff = Math.min(diff, pos1 - pos2);
                j++;
            }
        }
        return diff;
    }
}
```

245. Shortest Word Distance III, Medium, Locked

http://buttercola.blogspot.com/2015/08/leetcode-shortest-word-distance-iii.html

This is a **follow up** of Shortest Word Distance. The only difference is now *word1* could be the same as *word2*.

Given a list of words and two words *word1* and *word2*, return the shortest distance between these two words in the list.
word1 and *word2* may be the same and they represent two individual words in the list.
For example,

Assume that words = ["practice", "makes", "perfect", "coding", "makes"].
Given *word1* = “makes”, *word2* = “coding”, return 1.
Given *word1* = "makes", *word2* = "makes", return 3.
Note:
You may assume *word1* and *word2* are both in the list.Understand the problem:

Key: 本題的解法是 在本題 I 的解法上作改動. 改動的地方是考慮 word1=word2 之情況. 若 word1=word2=haha, 則對於輸入數組: * * haha * * * * * haha * * * *

E2(用這個): 只做一次遍歷. 第一次遇到 haha 時, 將 p1 和 p2 都設為第一個 haha 的位置: * * haha (p1, p2) * * * * * haha * * * *. 第二次遇到 haha 時, 只將 p2 設為第二個 haha 的位置: * * haha (p1) * * * * * haha(p2) * * * *

E3(最好懂, 代碼沒帖出來): 只做一次遍歷. 最開始就先判斷 word1 和 word2 是否相同, 若不相同, 則直接用本題 I 中的代碼(p1, p2 法). 若 word1 和 word2 相同, 則用一個 pre 和 cur 指針, cur 指向當前等於 word1 的詞, pre 指向上一個等 於 word1 的詞, 在各個(cur - pre)中取最小值.

Buttercola(不用這個): 只做一次遍歷. * * haha (posA) * * * * * haha * * * *, * * haha (posA, posB) * * * * * haha * * * *, 下一次遇到 haha 時, * * haha (posB) * * * * * haha(posA) * * * *, 如此可以給出正確結果了.

Understand the problem:
The problem is an extension of the previous one. The only diff is the word1 and word2 can be the same. It can still be easily solved by using a hash map. The question is, can we solve it by using the one-pass of the array?

The key is we cannot update the two pointers simultaneously, if they are the same. We could update one, compare the distance, and then update the other.

History:
E1 直接看的答案.
E2 在我電腦上結出正確結果, 代碼和方法跟 buttercola 的有點不同, 但比 buttercola 的更好理解, 所以把我 E2 的代碼也帖出來, 以後可以參考我 E2 的代碼, 不用看 buttercola 的代碼.
E3 很快寫好, 在 equals 前加了個!後即通過(能在我電腦上給出正確結果). E3 是一開始就判斷 word1 和 word2 是否相同, 若相同, 則用本題 I 的代碼, 若不同, 則用 pre 和 cur 法, 詳見上面紅字.

E2(用這個):

```
public static int shortestWordDistance(String[] words, String word1, String word2) {
    if(words == null || words.length == 0)
        return 0;

    int p1 = -1, p2 = -1;
    int minDistance = Integer.MAX_VALUE;

    for(int i = 0; i < words.length; i++) {
        if(!word1.equals(word2) && words[i].equals(word1)) {
            p1 = i;
        }
```

```
            if(word1.equals(word2) && p1 == -1 && words[i].equals(word1)) {
                p1 = i;
            }

            if(words[i].equals(word2)) {
                p2 = i;
            }

            if(p1 != -1 && p2 != -1 && p1 != p2) {
                minDistance = Math.min(minDistance, Math.abs(p1 - p2));
            }

        }

        return minDistance;
    }
```

Buttercola(不用這個):

Code (Java):

```
public class Solution {
    public int shortestWordDistance(String[] words, String word1, String word2) {
        int posA = -1;
        int posB = -1;
        int minDistance = Integer.MAX_VALUE;

        for (int i = 0; i < words.length; i++) {
            String word = words[i];
//若 word1=word2=haha, 則以下 if 執行完後為:
* * haha (posA) * * * * * haha * * * *
注意 else if 使得 posB 沒被賦值, posB 仍等於-1.
            if (word.equals(word1)) {
                posA = i;
            } else if (word.equals(word2)) {
                posB = i;
            }
//此時 posB = -1, 以下 if 不執行
            if (posA != -1 && posB != -1 && posA != posB) {
                minDistance = Math.min(minDistance, Math.abs(posA - posB));
            }

//以下 if 執行完後為:
* * haha (posA, posB) * * * * * haha * * * *
下一次遇到 haha 時, 以上第一個 if 執行完後為:
* * haha (posB) * * * * * haha(posA) * * * *
如此則上面第二個 if 中的 minDistance 可以給出正確結果了, 下面那個 if 算了後也沒甚麼影响了.
            if (word1.equals(word2)) {
                posB = posA;
```

```
            }
        }

        return minDistance;
    }
}
```

246. Strobogrammatic Number, Easy, Locked

https://leetcode.com/discuss/50594/4-lines-in-java

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.
Hints:
two pointers

Key: 比較兩頭的數字是不是 strobogrammatic, 然後往中間走. 要用到 string.contains()函數, 見最後. 本題之技巧在於用了個"00 11 88 696".contains(), 注意 00 和 11 之間有個空格, 其餘也是.

Strobo 只能含這些數字: 01869

History:
E1 直接看的答案.
E2 幾分鍾寫好, 能對各種 test case 結出正確結果, 故算通過, 但 E2 沒用"00 11 88 696".contains()這個技巧, 而是硬比較的兩頭數字是不是 01869.
E3 幾分鍾寫好, 一次通過(即在我電腦上能結出正確結果). E3 用的以上 key 中方法, 但 E3 專門處理了 num 含寄數個字母之情況, 實際上不必專門處理, 見代碼中我的解釋.

我: strobogrammatic 這個單詞查不到, 但題目中解釋得很清楚, 意思就是以自己中心為軸逆(或順)時針轉 180 度後和之前的一樣. 注意意思不是 '正放和倒放(即頭下腳上)都一樣', 因為 81 正放和倒放都一樣, 但不是題目所說的 strobogrammatic.

Just checking the pairs, going inwards from the ends.

//contains 函數見下. 696 寫到一起, 即可以 test 69, 也可以 test 96. 注意 num.charAt(i) +””+num.charAt(j)中的””是必要的, 好將 char 轉化為 String, 然後應該是自動轉化為了 CharSequence. 另外注意 00 和 11 之間有個空格, 即那個 string 為”00 空格 11 空格 88 空格 696”, 這樣可以防止 18 這樣的數被返回 true.

//若 num 為寄數個字母, 如”689”時, 怎麼辦? 注意此情況下, 最終 i=j=1, 即 num.charAt(i) +””+num.charAt(j) = “88”, 同樣可以用”00 11 88 696”來判斷. 巧哉!

```
public boolean isStrobogrammatic(String num) {
    for (int i=0, j=num.length()-1; i <= j; i++, j--)
        if (!"00 11 88 696".contains(num.charAt(i) + "" + num.charAt(j)))
            return false;
    return true;
}
```

public boolean contains(CharSequence s): The **java.lang.String.contains()** method returns true if

and only if this string contains the specified sequence of char values. 我: 由答案中可知, 參數也可以是 String 類型的, 應該是自動轉換為了 CharSequence 類型.

例子:

```
CharSequence cs1 = "int";

boolean retval = str1.contains(cs1);
```

247. Strobogrammatic Number II, Medium, Locked

https://leetcode.com/discuss/50412/ac-clean-java-solution

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.

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

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

Key: 遞歸調用 helper(n, m), 其作用是: 將所有長度為 n 的 strobogrammatic number 放入一個 List 中, 返回這個 List. 當 n != m 時, 返回的 strobogrammatic number 首尾可為 0; 當 n=m 時, 返回的 strobogrammatic number 首尾不為 0. helper(n, m) 中調用 helper(n - 2, m), 返回的 list 中包含的是長度為 n-2 的 strobogrammatic number, 再在這些 strobogrammatic number 首尾加數. 注意一個成品數的首位不能為 0. 注意 findStrobogrammatic 函數中調用 helper 時, 給 m 的值就是 n (findStrobogrammatich 的 n, 即題目中的那個 n), 所以整個程序運行時, m 的值都是 findStrobogrammatic 中的那個 n.
Strobo 只能含這些數字: 01869

文件 correct-output-for-check 中有正確輸出(即 Code Ganker 代碼的輸出), 以供檢查.

History:
E1 直接看的答案.

E2 很快寫好, 但 helper 中沒考慮 n=0 之情況, 且遞歸調用 helper 的位置不小心寫到了初始情況(n=0 等)之前, 所以導致 helper 參數越界, 改後即通過(即能給出以下答案代碼(我電腦上 copy 為 Solution2b)相同的結果), E2 代碼跟以下答案基本一樣, 只是 E2 並沒用 Arrays.asList()函數, 其實沒必要用, 以後也建議別用.

E3 後來用 key 中方法, 沒多久就寫好了, 本來可以一次通過的(即在我電腦上能給出正確結果), 但不小心在 n=1 時少輸出了一個 8, 改後即通過. E3 大多數時間都花在了這個上面: 最開始用了這麼一個錯誤的方法(很容易进入此誤區): 在 findStrobogrammatic(n)函數中調用 findStrobogrammatic(n-2), 然後在其結果中的每一個 String 的首尾加(非 0 的)數字(如加 8,8 或 9,6), 使其成為一個長度為 n 的 strobo. 另外, 由於 findStrobogrammatic(n-2)結果中所有 String 頭部都非 0, 所以還要將其首尾改為 0 後, 再在新的 String 的首

尾加(非 0 的)數字(如加 8,8 或 9,6), 使其成為一個長度為 n 的 strobo. 這兩種加的, 得到的結果組成的 List 即為最終結果. 注意本方法的錯誤之處在於, 在首尾改為 0 時, 若 findStrobogrammatic(n-2)結果中有如下 String: 818, 619, 916 等, 則在首尾改為 0 後, 就都成了 010, 再在它們首尾加(非 0 的)數字(如加 8,8 或 9,6), 得到的 strobo 就都是一樣的, 所以最終結果中有很多重複, 且 n 增加時會累加. 當 n=10 時, 正確結果中有 2500 個數, 而此方法得到的是 10240 個數. 當然也可以在加入 res 前先判斷 res 中是否已有此 String, 但這樣做會讓時間複雜度高不少. 所以此方法不可行.

```
public List<String> findStrobogrammatic(int n) {
    return helper(n, n);
}

 //helper(n, m)的作用是: 將所有長度為 n 的 strobogrammatic number 放入一個 List 中, 返回這個 List. 當 n != m 時, 返回的 strobogrammatic number 首尾可為 0; 當 n=m 時,  返回的 strobogrammatic number 首尾不為 0.
List<String> helper(int n, int m) { //注意 findStrobogrammatic 函數中調用 helper 時, 給 m 的值就是 n, 所以整個程序運行時, m 的值都是 findStrobogrammatic 中的那個 n.
    if (n == 0) return new ArrayList<String>(Arrays.asList(""));
    if (n == 1) return new ArrayList<String>(Arrays.asList("0", "1", "8")); //這個將數組弄成 ArrayList 的方法已經 record 到了 Java p790. Arrays.asList()函數見最後. 也不一定要用這個函數, 直接往 list 中 add 也可以.

    List<String> list = helper(n - 2, m); //整個程序運行時, m 的值都是 findStrobogrammatic 中的那個 n. list 中包含的是長度為 n-2 的 strobogrammatic number. 以下就是在這些 strobogrammatic number 首尾加數.

    List<String> res = new ArrayList<String>();

    for (int i = 0; i < list.size(); i++) {
        String s = list.get(i);

        //以下注意, 整個程序運行時, m 的值都是 findStrobogrammatic 中的那個 n
        if (n != m) res.add("0" + s + "0"); //當 n != m(即 n<m)時, 返回的 strobogrammatic number 首尾可為 0, 因為此時還要在首尾加其它數; 當 n=m 時,  返回的 strobogrammatic number 首尾不為 0.

        res.add("1" + s + "1");
        res.add("6" + s + "9");
        res.add("8" + s + "8");
        res.add("9" + s + "6");
    }

    return res;
}
```

The java.util.Arrays.asList(T... a) returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs.

Following is the declaration for java.util.Arrays.asList() method
public static <T> List<T> asList(T... a)

String a[] = new String[]{"abc","klm","xyz","pqr"};
List list1 = Arrays.asList(a);

248. Strobogrammatic Number III, Hard, Locked

https://leetcode.com/discuss/50628/ac-java-solution-with-explanation

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.

Key: 遞歸. 寫一個 helper(int n, Map<Integer, List<String>> map, String low, String high), 返回一個 list, 它包含所有長度為 n 且在 low 和 high 之間的 strobogrammatic number. 而 map 中的<n, list> 中的 list 中放的是 所有 n 位 strobo (不需要在 low 和 high 之間). helper 還要全局變量 count 設為正確的值(即本題結果). 在 helper(n....)中調用 helper(n-2...), 然後在兩頭加字符. 可以用 string.compareTo()函數(見 Java p410), 但注意它比較的是字典順序, 而不是數的大小, 故 E2 沒用, 而是自己寫的(答案用了的, 當然要保證兩個 string 長度相同, compareTo()給的結果才是我們想要的)

strobo 只能含這些數字: 01869

文件 correct-output-for-check 中有正確輸出(即 Code Ganker 代碼的輸出), 以供檢查.

History:
E1 直接看的答案.

E2 邊寫邊在我電腦上試結果, 最後能給出和以下答案(我電腦上 copy 為 Solution2b)一樣的結果, 且對於大輸入, 運行時間也差不多(注 1), 所以算通過, E2 的其本方法跟以下答案一樣, 但在代碼實現上有較大差別, E2 的代碼比較容易想到. 由於 E2 對以下 high 再大 10 倍之情況空間超出, 所以以後不用 E2 代碼, 還是用答案代碼.

注 1: 對於輸入為 low = "50", high = "500000000000000000"的情況, 以下答案的運行時間為約 3 秒, E2 的運行時間為 5 秒, 差別不大, 時間複雜度應該只是 O(kN)中的 k 不同. 但對於 high 再大 10 倍之情況, 即 low = "50", high = "5000000000000000000", 我的代碼就報錯: java.lang.OutOfMemoryError, 應該是遞歸棧空間超出了, 但答案的代碼能花約 8 秒的時間得到結果.

E3 通過(在我電腦上對幾個例子能給出正確結果). E3 沒用遞歸. 最開始用的方法為: 把所有長度的 strobo 都放在 map 裡, 在我電腦上運行時, 對 correct-output-for-check 中的例子, 都能給出正確結果, 但運行時間約為 7

秒(Code Ganker 的為 3 秒). E3 後來又改進了, 寫一個 void helper(int lowLen, int highLen), 其作用為: 將全局變量 count 設為 lowLen 和 highLen 之間(lowLen < len < highLen)的 strobo 個數, 並將全局變量 lowList 中放入長度為 lowLen 的 strobo, 將全局變量 highList 中放入長度為 highLen 的 strobo. 然後在 strobogrammaticInRange 函數中數比 low 大的 strobo 數量, 及比 high 小的 strobo 數量, 將它們加到 count 上, 即結果. 這樣就只保存了長度跟 low 和 high 相同的 strobo 放在 list 裡面, 其餘 strobo, 只有運行暫時保存了上一個 strobo. 這樣對 correct-output-for-check 中的例子, 都能給出正確結果, 運行時間減少到了約 5 秒, 且大大節省了空間. 但對於 E2 中提到那個 high 再大 10 倍之情況, 即 low = "50", high = "5000000000000000000", 我的 E3 代碼還是報錯:java.lang.OutOfMemoryError: GC overhead limit exceeded, , 但答案的代碼能花約 8 秒的時間得到結果. 所以我 E2 和 E3 的方法都在空間上效率太低. 但後來看了以下答案, 好像它在空間上還要奢侈一些, 答案是將改有 strobo 都放到 map 中(即我 E3 最開始那個方法), 但不知道為甚麼答案運行起來就比我 E2 和 E3 的都快, 且對 high 大 10 倍那個情況也沒有空間超出.

答案代碼(用它):

```
public class Solution {

  int count = 0;
  String[][] pairs = {{"0", "0"}, {"1", "1"}, {"6", "9"}, {"8", "8"}, {"9", "6"}};

  public int strobogrammaticInRange(String low, String high) {
    // use a look-up table to return the strobogrammatic list of length n
    Map<Integer, List<String>> map = new HashMap<Integer, List<String>>();
    map.put(0, new ArrayList<String>(Arrays.asList("")));
    map.put(1, new ArrayList<String>(Arrays.asList("0", "1", "8")));

    // loop through all possible lengths
    for (int len = low.length(); len <= high.length(); len++)
  //注意此處保證了 len<=high.length(). 這是 A 處.
      helper(len, map, low, high);

    return count;  //count 是在下面 helper 中算的
  }

  // return the strobogrammatic list of length n
//helper(int n, Map<Integer, List<String>> map, String low, String high), 返回一個 list, 它包含所有長度為 n 且在 low 和 high 之間的 strobogrammatic number. 而 map 中的<n, list> 中的 list 中 放的是 所有 n 位 strobo (不需要在 low 和 high 之間). helper 還要全局變量 count 設為正確的值(即本題結果). 在 helper(n....)中調用 helper(n-2...), 然後在兩頭加字符.
  List<String> helper(int n, Map<Integer, List<String>> map, String low, String high) {
    List<String> res = new ArrayList<String>();

    if (map.containsKey(n)) {
      res = map.get(n);
    } else {
      // found in look-up table? return it, otherwise do the recursion by n - 2
      List<String> list = map.containsKey(n - 2) ? map.get(n - 2) : helper(n - 2, map, low, high);

      for (int i = 0; i < list.size(); i++) {
        String s = list.get(i);
```

```
      for (int j = 0; j < pairs.length; j++) {
        // form the new strobogrammatic number
        String v = pairs[j][0] + s + pairs[j][1];

        // if it's larger than high already, no need to proceed
        if (v.length() == high.length() && v.compareTo(high) > 0)
//前面 A 處保證了 v.length()<=high.length().
          break;

        res.add(v);
       }
      }

     // put the new list to look-up table
     map.put(n, res);
    }

    // if current length is longer than low
    // we start to count
    if (n >= low.length()) {
     count += res.size();

     for (String s : res) {
      // eliminate the number that is outside [low, high] range
      if ((s.length() > 1 && s.charAt(0) == '0') ||
          (s.length() == low.length() && s.compareTo(low) < 0) ||
          (s.length() == high.length() && s.compareTo(high) > 0))
        count--;
     }
    }

    return res;
  }

}
```

(不用!) E2 的代碼(改進後):

```
public class Solution2 {
    private static int numStrobo; //E2 為了 numSrobo 能按 reference 轉遞, 是將其放在一個 ArrayList 中的, 其實沒必要這樣做, 因為這裡的 numStrobo 是 Solution 的一個 member, 所以 Solution 的 member function 是可以改變 numStrobo 的值的, 這跟之前的 pass by reference 情況還不一樣(那裡的 numStrobo 是在某個函數中定義的). 後來看了答案後, 改進成了這個樣子.
    private static Map<Integer, List<String>> map = new HashMap<>();
```

```
public static int strobo(String low, String high) {
    if(compareStrings(low, high) > 0) return 0;
    numStrobo = 0;
    int highLen = high.length();

    //為何要調用兩次 helper? 因為一次處理 0, 2, 4...位的數, 另一次處理 1, 3, 5...位的數
    helper(highLen, map, low, high).size();
    helper(highLen - 1, map, low, high).size();

    return numStrobo;
}

private static List<String> helper(int n, Map<Integer, List<String>> map, String low, String high) {
    List<String> res = new ArrayList<>();
    List<String> listForMap = new ArrayList<>();

    if(n <= 0) {
        listForMap.add("");
        map.put(0, new ArrayList<String>(listForMap));
        return res;
    }

    if(n == 1) {
        listForMap.add("0");
        listForMap.add("1");
        listForMap.add("8");
        map.put(1, new ArrayList<String>(listForMap));
        return res;
    }

    helper(n - 2, map, low, high);
    List<String> listBefore = map.get(n - 2);

    String[][] pairs = {{"0", "0"}, {"1", "1"}, {"6", "9"}, {"8", "8"}, {"9", "6"}};

    for(String s : listBefore) {
        for(String[] pair : pairs) { //數組也可以做 for 循環的 循環變量!
            String sNew = pair[0] + s + pair[1];
            listForMap.add(new String(sNew));
            if(!pair[0].equals("0"))
                if(compareStrings(low, sNew) <= 0 && compareStrings(sNew, high) <= 0) res.add(new
```

```
String(sNew));
            }

            map.put(n, listForMap);
        }

        if(res.size() > 0)
            numStrobo += res.size();

        return res;
    }

    private static int compareStrings(String s, String t) {
        int sn = s.length(), tn = t.length();
        if(sn != tn) return sn - tn;

        for(int i = 0; i < sn; i++) {
            int sd = s.charAt(i) - '0';
            int td = t.charAt(i) - '0';
            if(sd != td) return sd - td;
        }

        return 0;
    }

    public static void main(String[] args) {
        System.out.println(strobo("50", "500000000000000000"));
    }
}
```

249. Group Shifted Strings, Easy, Locked

https://leetcode.com/discuss/50358/my-concise-java-solution

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 (我： 注意不含大寫， 也不含數字), group all strings that belong to the same shifting sequence.
For example, given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],
Return:

```
[
  ["abc","bcd","xyz"],
```

  ["az","ba"],
  ["acef"],
  ["a","z"]
]
Note: For the return value, each *inner* list's elements must follow the lexicographic order.

Key: 看下面我說的(E2 表示寫得好).

History:
E1 直接看的答案.
E2 很快寫好, 但在算向左移動越界時沒寫對, 且忘了將代表也放入答案中, 改後即對題目中的例子給出正確結果, 且代碼跟以下答案差不多, 所以算通過.
E3 的情況跟 E2 基本一樣: E3 幾分鍾寫好, 但改了兩個錯誤才在我電腦上結出正確答案(算通過), 一是在左移時, 忘了考慮移後比 a 小之情況, 二是放 map 時, 忘了將代表放入到 map 的 list 中去.

我: 以下的方法就是: 若一個 String 為 fuck, 則可以按以下表示:
a b c d e f g h i j k l m n o p q r s t u v w x y z
a b c d e f g h i j k l m n o p q r s t u v w x y z
a b c d e f g h i j k l m n o p q r s t u v w x y z
a b c d e f g h i j k l m n o p q r s t u v w x y z
其中首字母 f 離 a 的距離為 f-a=5, 所以 fuck 的所有字母都向左移 5 位, 成為 apxf:
a b c d e f g h i j k l m n o p q r s t u v w x y z
a b c d e f g h i j k l m n o p q r s t u v w x y z
a b c d e f g h i j k l m n o p q r s t u v w x y z
a b c d e f g h i j k l m n o p q r s t u v w x y z
顯然 apxf 和 fuck 是同一個 group 中的, 我們可以將 apxf 作為這個 group 的代表, 其餘的 String 都可以按照同樣的方法左移(即將該 String 的首字母移到 a 要移多少位, 然後將該 String 的所有字母都左移這麼多位), 若移後為 apxf, 則它們也是這個 group 中的. 實現時可用一個 HashMap<String, List<String>>, 將 apxf 當 key, 若發現同一個 group 中的 String, 則將其加入到 List 中.

```
public class Solution {
    public List<List<String>> groupStrings(String[] strings) {
        List<List<String>> result = new ArrayList<List<String>>();
        Map<String, List<String>> map = new HashMap<String, List<String>>();
        for (String str : strings) {
            //以下是找 str 對應的 key
            int offset = str.charAt(0) - 'a'; //offset 為要左移的位數
            String key = "";
            for (int i = 0; i < str.length(); i++) {
                char c = (char) (str.charAt(i) - offset); //將該 string 中的所有字母都左移 offset 那麼多位
                if (c < 'a') { //若移出左邊界時. 例如上面 fuck 中的 c 最後移成了 x
                    c += 26;
                }
                key += c;
            }
            if (!map.containsKey(key)) {
                List<String> list = new ArrayList<String>();
                map.put(key, list);
```

```
            //以上兩句應該可以簡寫為 map.put(key, new ArrayList<String>());
        }
        map.get(key).add(str);
    }
    //map 已弄好, 以下是排序
    for (String key : map.keySet()) { //遍歷 map 的一個好方法, 已加入 Java 書 p834
        List<String> list = map.get(key);
        Collections.sort(list); //由此可知, 一堆 String 也可以用 Collections.sort()來排序
        result.add(list);
    }
    return result;
  }
}
```

250. Count Univalue Subtrees, Medium, Locked

https://leetcode.com/discuss/50357/my-concise-java-solution

Given a binary tree, count the number of uni-value subtrees.
A Uni-value subtree means all nodes of the subtree have the same value.
For example:
Given binary tree,

```
          5
         / \
        1   5
       / \   \
      5   5   5
```

return 4.

Key: 遞歸. 寫一個 helper(node, count), 作用就是返回 以 node 為根的子樹 是不是 univalue 的, 若是返回 true 並且給 count[0]加 1(count 為數組), 若不是則返回 false. 若 以 node.left 為根的字樹 和 以 node.right 為根的子樹 都是 univalue 的, 則判斷 node 的值跟 node.left 和 node.right 是否相等. Null 樹被認為是 univalue 的.

History:
E1 直接看的答案.
E2 對題目中的輸入能得到正確的結, 且代碼跟答案幾本上差不多, 算通過, E2 最開始范了個 &&只执行一邊 的錯誤, 詳見代碼中.
E3 沒多久一次寫好, 對題目中的例子可以給出正確結果, 且方法和以下答案一樣, 所以算一次通過. E3 對 key 一點映像都沒有了, 想到 key 中一樣的方法並寫好後, 還以為我的方法(和代碼)太麻煩, 想看看答案有甚麼高見, 結果 suprisingly, 答案方法跟我的居然一樣, 所以該方法完全可以算是我獨立想出來的.

```
public class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        int[] count = new int[1]; //把 count 弄成一個只有一個元素的數組, 應該就是為了 pass by reference
        helper(root, count);
        return count[0];
    }

    //helper(node, count)的作用就是返回 以 node 為根的子樹 是不是 univalue 的, 若是返回 true 並且給
```

count[0]加 1, 若不是則返回 false.

```
    private boolean helper(TreeNode node, int[] count) {
        if (node == null) {
            return true;
        }

//以下幾句為何不寫為 if(helper(node.left, count) && helper(node.right, count)) {...}? 因為根据&&之性質, 若左邊的 helper(node.left, count)返回 false, 則不执行右邊的 helper(node.right, count), which is not what we want (we want to visit every node). E2 最開始就是這樣寫的, 返回了錯誤的結果, 後來查出就是這個原因, 後來改成多定義了 left 和 right 兩個 boolean 變量, which 與答案不謀而合.

        boolean left = helper(node.left, count);
        boolean right = helper(node.right, count);

        if (left && right) {
            if (node.left != null && node.val != node.left.val) {
                return false;
            }
            if (node.right != null && node.val != node.right.val) {
                return false;
            }
            count[0]++;
            return true;
        } //end of if (left && right)

        return false;
    }
}
```

251. Flatten 2D Vector, Medium, Locked

http://buttercola.blogspot.com/2015/08/leetcode-flatten-2d-vector.html

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:

1.How many variables do you need to keep track?
2.Two variables is all you need. Try with `x` and `y`.
3.Beware of empty rows. It could be the first few rows.
4.To write correct code, think about the invariant to maintain. What is it?
5.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*?
6.Not sure? Think about how you would implement `hasNext()`. Which is more complex?
7.Common logic in two different places should be refactored into a common method.

Key: 用兩個 Iterator, 一個是管 輸入的二維數組 的外層的, 一個是管 輸入的二維數組 的內層的: Iterator<List<Integer>>outerIterator; Iterator<Integer> innerIterator; 函數 hasNext() 要比 next() 複雜很多. 在 hasNext()中可遞歸調用 hasNext(), 不必專門處理空白行.

History:
E1 直接看的答案.
E2 沒通過, 對末尾有空白行的情況時之 hasNext()沒處理好.
E3 改了一兩次後通過(即能對以下輸入得出正確結果). E3 的方法比答案的好懂, 好想到, 且沒用遞歸, 故最好也看看 E3 的代碼(由於好想到, 故實際上以後可以就用 E3 代碼). 輸入: [[], [], [], [1, 2], [], [], [3], [4, 5, 6], [], [], []] , E3 輸出: 1 2 3 4 5 6.

我: 以下給了兩種方法, 要掌握的是第二種方法, 第一種也要看看(我沒時間, 沒看)

Understand the problem:
The question itself is very easy to solve. Just several corner cases need to think of:
-- What if the 2d vector contains empty arrays, e.g. [ ], [ ], 1 2 3 ? In this case, the next() should not output anything, but the return type is int. There the hasNext() should be more complicated in which it handles this situation.
-- What if the 2d vector itself is empty? Again, handle it in hasNext()

Code (Java):

以下代碼不用:

```
public class Vector2D {

    private List<List<Integer>> vec2d;
    private int rowId;
    private int colId;
    private int numRows;

    public Vector2D(List<List<Integer>> vec2d) {
        this.vec2d = vec2d;
        rowId = 0;
        colId = 0;
        numRows = vec2d.size();
    }

    public int next() {
```

```
        int ans = 0;

        if (colId < vec2d.get(rowId).size()) {
            ans = vec2d.get(rowId).get(colId);
        }

        colId++;

        if (colId == vec2d.get(rowId).size()) {
            colId = 0;
            rowId++;
        }

        return ans;
    }

    public boolean hasNext() {
        while (rowId < numRows && (vec2d.get(rowId) == null || vec2d.get(rowId).isEmpty())) {
            rowId++;
        }

        return vec2d != null &&
        !vec2d.isEmpty() &&
        rowId < numRows;
    }
}
```

Followup:
As an added challenge, try to code it using only iterators in C++or iterators in Java.

Code (Java):

用以下代碼:

```
public class Vector2D {
    private Iterator<List<Integer>>outerIterator; //Iterator 見 Java p787.
    private Iterator<Integer> innerIterator;

    //Constructor
    public Vector2D(List<List<Integer>> vec2d) {
        outerIterator = vec2d.iterator();
        innerIterator = Collections.emptyIterator(); //此句是初始化 innerIterator 這個變量, 若不要此句, 則下面 A 處直接用沒初始化的 innerIterator 會報錯. E3 實踐表明的這點. E3 代碼是這樣初始化的: innerIterator = list.iterator(), 其中 list 是一個空 List(參見下面的 E3 代碼). Collections.emptyIterator(): Returns an iterator that has no elements. 已記入 Java p787.
    }

    public int next() {
        return innerIterator.next(); //OJ does while (i.hasNext()) cout << i.next();, i.e., always calls
```

```
hasNext before next (來自 https://leetcode.com/discuss/50292/7-9-lines-added-java-and-c-o-1-space). 這也符合我們平時之習慣, 調用 next 前都要先用 hasNext 來檢查一下, 否則若是 next 把 hasNext 的功能也包括了, 那要 hasNext 有卵用.
      //另外注意 Constructor 中給 innerIterator 賦的一個空 iterator. 但由於默認每次調用 next()前都要先調用 hasNext(), 所以 innerIterator 的非空值實際上是在 hasNext()中賦的.
   }

   public boolean hasNext() {
      if (innerIterator.hasNext()) {  //A 處
         return true;
      }

      if (!outerIterator.hasNext()) {
         return false;
      }

      //以下即 innerIterator 沒有 next, 且 outerIterator 有 next 時之情況 (因為若能從上面兩個 if 走出來, 則就是這種情況, 有點巧)
      innerIterator = outerIterator.next().iterator();

      return hasNext(); //遞歸. 對上句得到的 innerIterator 執行 hasNext(). 注意此處把空白數組也一並處理了.
   }
}

E3 的代碼(僅作參考, 比答案好懂, 好想到, 沒用遞歸):
class Vector2D {
   Iterator<List<Integer>> iterOut;
   Iterator<Integer> iterIn;

   public Vector2D(List<List<Integer>> vec2d) {
      iterOut = vec2d.iterator();
      List<Integer> list = new ArrayList<>();
      iterIn = list.iterator();
   }

   public int next() {
      return iterIn.next();
   }

   public boolean hasNext() {
      if(iterIn.hasNext()) return true;
      else {
         while(iterOut.hasNext()) {
            List<Integer> curList = iterOut.next();
            if(curList.isEmpty()) continue; //跳過空白數組
            iterIn = curList.iterator(); //Set new iterIn, for future use
            return true;
         } //end of while
```

```
        } // end of else

        return false;
    }
}
```

252. Meeting Rooms, Easy, Locked

https://aquahillcf.wordpress.com/2015/08/30/leetcode-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.

Key: 本題本質就是判斷這些 interval 有沒有 overlap. 題目的意思是, 若有兩個 interval 有 overlap, 則就算這個人不能 attend all meetings. 寫一個比較 interval 的 Comparator, 然後用 Arrays.sort 給這些 interval 排序 (按 start, 若 start 相同則按 end), 若兩個相鄰的 interval 有 overlap 則 return false.

History:
E1 直接看的答案.
E2 代碼跟以下的基本一樣, 算通過.
E3 幾分鍾寫好(copy 了 56. Merge Intervals 中我 E3 的 comparator), 算通過. E3 的代碼跟以下答案基本上完全一樣, 唯一的不同就是 E3 不小心將 start < end 寫為了 start > end. 如果本題沒鎖, E3 應該能通過 test case 查出此錯誤(E3 對題目中的 case 能給出正確答案, 應為巧合), 故仍算此題通過(若算沒通過, 以後要重點看此題, 完全沒必要).

在做过 II 之后，回来看着道题就显得非常的简单。因为这只是判断可不可以全参加，其实只要有个 meeting 的 end 比另外一个 meeting 的 start 要晚，那么就是 false。

Solution:

```
public boolean canAttend(Interval[] meetings){ //Interval 是 leetcode 自定義的類, 可見 56. Merge Intervals 的 OJ 中的定義: https://leetcode.com/problems/merge-intervals/
    if(meetings == null || meetings.length == 0){
        return true;
    }
//以下的寫法是 Arrays.sort(meetings, comparator); 其中 comparator 部分是定義了一個新的 Comparator. 這是簡寫, 完整的寫法見 Java 795 的紅字.
    Arrays.sort(meetings, new Comparator<Interval>(){
        public int compare(Interval i1, Interval i2){ //253 題中的 compare 是 private 的
            if(i1.start == i2.start) return i1.end-i2.end;
            else return i1.start-i2.start;
        }
    }); //注意末尾是分號, 因為這是一句話:  Arrays.sort(meetings, comparator);
    for(int i = 1; i < meetings.length; i++){
        if(meetings[i].start < meetings[i-1].end){
            return false;
        }
    }
```

```
    return true;
}
```

253. Meeting Rooms II, Medium, Locked

https://aquahillcf.wordpress.com/2015/08/30/leetcode-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.

Key: 本題本質就是求 有共同部分 的 interval 的最多個數(如下面我紅圖中的右邊那三條, 它們三個就是有共同部分的 interval). 注意 overlap 在一起的 interval 不一定都有共同部分, 比如題目中的那三個 interval. 本方法不容易相到, 要記住. 先將所有 interval 排序(按 start, 若 start 相同則按 end). 掃描這些 interval, 用一個 PriorityQueue (Java p807)保存 end (PriorityQueue 中總 end 數即所需房間數), 若當前 interval 的 start 小於 'PriorityQueue 中 peek 出來的最小 end', 則說明有共同部分, 則將當前 interval 的 end 加入 PriorityQueue 中, 否則則將最小 end 從 PriorityQueue 中刪掉(但仍要加入當前 interval 的 end). 可以用以下例子想:

History:
E1 直接看的答案.
E2 代碼跟以下差不多, 算通過.
E3 沒做出來, 原因是將 key 基本上都忘了.

(我: 癈話開始)想法： 按照起始时间来 sort，假设上面是 sort 好的，那么我们看 m1.end > m2.start => meeting room+1
往后走，m2.end < m3.start => meeting room 不变。=> 只需要将每一个 meeting 的 end 和下一个 meeting 的 start 相比即可。
但是反例： [[0,20], [5,40], [25,40]] 第一个可以和第三个在一起开，如果按照上面的算法，是需要三间的，但正确答案需要两间。
然后我的想法变为我要维护两个变量，即一个 minEnd time, 一个 maxEnd time，可是还是不对万一我们有三个不同的区间呢(我: 癈話結束)。后来看了别人的想法之后才知道我们需要用一个 PriorityQueue 来维护这个 end time，如果有 start 比我们的最小的 endtime 小的话，我们就可以 remove 掉原来的，然后再加入新的。这是一个非常有序的迭代！这里用了一个 priorityqueue 来进行维护这个 queue。而最后自然也很明朗，我们拿到多少个最终的 end time 就是我们需要几间 meeting room。

Solution:

```
public int getMeetingRooms(Interval[] meetings){
    if(meetings == null || meetings.length == 0){
        return 0;
    }
```

```
    Arrays.sort(meetings, new Compararator<Interval>(){
       private int compare(Interval i1, Interval i2){ //252 題中的 compare 是 public 的
          if(i1.start == i2.start) return i1.end-i2.end;
          else return i1.end-i2.end; //應該是 i1.start-i2.start
       }
       });

    PriorityQueue<Integer> queue = new PriorityQueue<Integer>();
    queue.add(meetings[0].end);

    for(int i = 1; i < meetings.length; i++){
       Interval curr = meetings[i];
       int value = queue.peek();
       if(curr.start > value){
          queue.remove(value);
       }
       queue.add(curr.end);
    }
    return queue.size();
}
```

254. Factor Combinations, Medium, Locked

https://leetcode.com/discuss/51250/my-recursive-dfs-java-solution

Numbers can be regarded as product of its factors. For example,

```
8 = 2 x 2 x 2;
  = 2 x 4.
```

Write a function that takes an integer *n* and return all possible combinations of its factors.

Note:

1. Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is `[2, 6]`, not `[6, 2]`.
2. You may assume that *n* is always positive.
3. Factors should be greater than 1 and less than *n*.

Examples:

input: `1`

output:

```
[]
```

input: 37

output:

```
[]
```

input: 12

output:

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

input: 32
output:

```
[
  [2, 16],
  [2, 2, 8],
  [2, 2, 2, 4],
  [2, 2, 2, 2, 2],
  [2, 4, 4],
  [4, 8]
]
```

Key: item-res 模版遞歸法. 寫一個 helper(List<List<Integer>> result, List<Integer> item, int n, int start), 作用是: 將 n 分成 >=start 的因子 的組合, 每一種分法放入 item 中(item 中原來可能有別的數(設叫 A)). 當 item 中的數足夠成為一個組合時(下面用 n<=1 來判斷的), 將此 item 拷一份放入 result 中. 最後將本 helper 對 item 的影响消除, 即刪除加入到 item 中的元素(使 item 只剩 A). 在 helper(result, item, n, start) 中遞歸調用 helper(result, item, n/i, i) (i 的取值范圍為[start, n], i 即為 n 的一個因子, 在調用 helper 之前已將 i 加入到 item 中), 注意重點在參數 n/i 上, 而不是最後一個參數 i 上. 最後一個參數取 i, 及它取值范圍為 [start, n], 共同的作用就是使得 item 中的數是從小到大排好序的(E3 表示, 這就是本方法之目的, 因為昇序排列可以避免重復的 item), 這也是 start 這個參數的作用.

History:
E1 直接看的答案.

E2 改了一兩次後, 對題目中的輸例子能給出正確結果, 算通過. E2 代碼跟以下答案有一個小地方不同, 已寫入以下答案中. 以上 key 有很多是 E2 寫的(E3 表示寫得好).

E3 改了幾次後, 對題目中的例子能給出正確結果, 且對幾個附加的 test case(見 leetcode 分類中本題處)也能給出正確結果, 故算通過. E3 寫好後, 跟答案一比, 才發現 E3 代碼居然跟答案代碼基本一模一樣(除了變量名和參數順序外). E3 代碼已帖在最後, 供欣賞. E3 寫的時候完全忘了 key 中方法, 一點映像都沒有了, 故 E3 完全是獨立做出來的, NB! 忘之證据: E3 寫好後, 雖然能對題目中的例子能給出正確結果, 但還不確定此方法是對的, 甚至都不確定是否方向(以昇序為目的)都想錯了(雖然當時覺得應該沒錯, 但還是擔心有沒考慮周全的地方), 故又附加了幾個 test case, 看到對它們也給出正確結果後, 且看到 E3 代碼跟答案一模一樣後, 才確信我 E3 代碼和方向是正確的. E3 代碼寫法(變量名&參數順序)比較符合我的習慣, 故可以看看 E3 代碼, 幫助看懂答案代碼.

答案代碼(用它):

```
public List<List<Integer>> getFactors(int n) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    helper(result, new ArrayList<Integer>(), n, 2);
    return result;
}

//helper(result, item, n, start)的作用是: 將 n 分成 >=start 的因子 的組合, 每一種分法放入 item 中
```

(item 中原來可能有別的數(設叫 A)). 當 item 中的數足夠成為一個組合時(下面用 n<=1 來判斷的), 將此 item 拷一份放入 result 中. 最後將本 helper 對 item 的影响消除, 即刪除加入到 item 中的元素(使 item 只剩 A).

```
public void helper(List<List<Integer>> result, List<Integer> item, int n, int start){
    if (n <= 1) {
        if (item.size() > 1) { //E2 的代碼沒有此 if. 導致的結果就是 getFactors(n)最終的結果 result 中會包含[n]這麼個 List. E2 的處理方法是在 getFactors()函數中 return result 前直接將該 List 刪掉. E3 居然連這句都跟答案一樣.
            result.add(new ArrayList<Integer>(item)); //注意此處是復製的 item, 即對 item 沒做任何操作, 所以不用擔心如何消除對 item 的影响.
        }
        return; //即然 n<=1 了, 後面的 for 中的就沒意義了, 所以此時 return.
    }

    for (int i = start; i <= n; ++i) {
        if (n % i == 0) {
            item.add(i);
            helper(result, item, n/i, i); //以 i 為 start, 也保證了 item 中的元素是昇序排列的, 這正是題目中要求
            item.remove(item.size()-1); //回溯, 保護現場. 上一句 helper()加入 item 中的元素己經被 helper()自己刪掉了, 此處是刪前面加的 i, 好消除本 helper()對 item 的影响.
        }
    }
}
```

E3 代碼(不用, 只是用來欣賞它跟答案的相似性):

```
    public  List<List<Integer>> getFactors(int n) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(n <= 3) return res;
        helper(n, 2, new ArrayList<Integer>(), res);
        return res;
    }

    private void helper(int n, int start, List<Integer> item, List<List<Integer>> res) {
        if(n == 1) {
            if(item.size() > 1) res.add(new ArrayList<Integer>(item));
            return;
        }

        for(int factor = start; factor <= n; factor++) {
            if(n % factor == 0) {
                item.add(factor);
                helper(n / factor, factor, item, res);
                item.remove(item.size() - 1);
            }
        }
    }
```