**Chapter 2 Greedy Algorithm(6/6)**

**605. Can Place Flowers(Easy):**  
Suppose you have a long flowerbed in which some of the plots are planted and some are not. However, flowers cannot be planted in adjacent plots - they would compete for water and both would die.

Given a flowerbed (represented as an array containing 0 and 1, where 0 means empty and 1 means not empty), and a number n, return if n new flowers can be planted in it without violating the no-adjacent-flowers rule.

Example 1:

Input: flowerbed = [1,0,0,0,1], n = 1

Output: True

Example 2:

Input: flowerbed = [1,0,0,0,1], n = 2

Output: False

Note:

1. The input array won't violate no-adjacent-flowers rule.
2. The input array size is in the range of [1, 20000].
3. n is a non-negative integer which won't exceed the input array size.

Sol:

如果當前位子有花的話，就跳過。

頭尾要另外處理，其餘的就檢查前後是否有種花。

例外是當長度為0 or 1時要特別處理。



**452. Minimum Number of Arrows to Burst Balloons(Medium)**  
There are a number of spherical balloons spread in two-dimensional space. For each balloon, provided input is the start and end coordinates of the horizontal diameter. Since it's horizontal, y-coordinates don't matter and hence the x-coordinates of start and end of the diameter suffice. Start is always smaller than end. There will be at most 104 balloons.

An arrow can be shot up exactly vertically from different points along the x-axis. A balloon with xstart and xend bursts by an arrow shot at x if xstart ≤ x ≤ xend. There is no limit to the number of arrows that can be shot. An arrow once shot keeps travelling up infinitely. The problem is to find the minimum number of arrows that must be shot to burst all balloons.

Example:

Input:

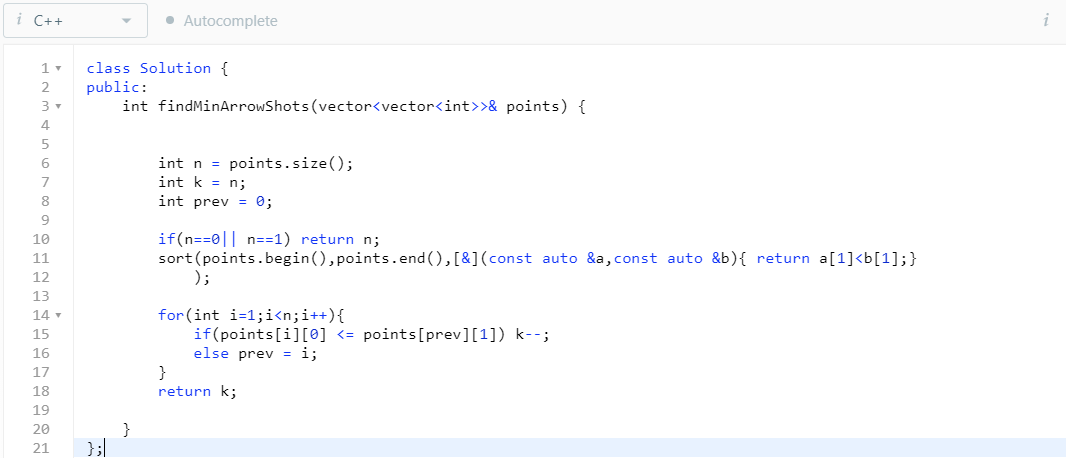
[[10,16], [2,8], [1,6], [7,12]]

Output:

2

Explanation:

One way is to shoot one arrow for example at x = 6 (bursting the balloons [2,8] and [1,6]) and another arrow at x = 11 (bursting the other two balloons).

Sol:  
跟435類似，先把區間按照結尾的大小進行增序排列，若和前一段結尾有重疊，就可以少射出一發弓箭(--k)。若都不重疊 原本需要k把弓箭，k為points得長度。  


**763. Partition Labels(Medium)**

A string S of lowercase letters is given. We want to partition this string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

Example 1:

Input: S = "ababcbacadefegdehijhklij"

Output: [9,7,8]

Explanation:

The partition is "ababcbaca", "defegde", "hijhklij".

This is a partition so that each letter appears in at most one part.

A partition like "ababcbacadefegde", "hijhklij" is incorrect, because it splits S into less parts.

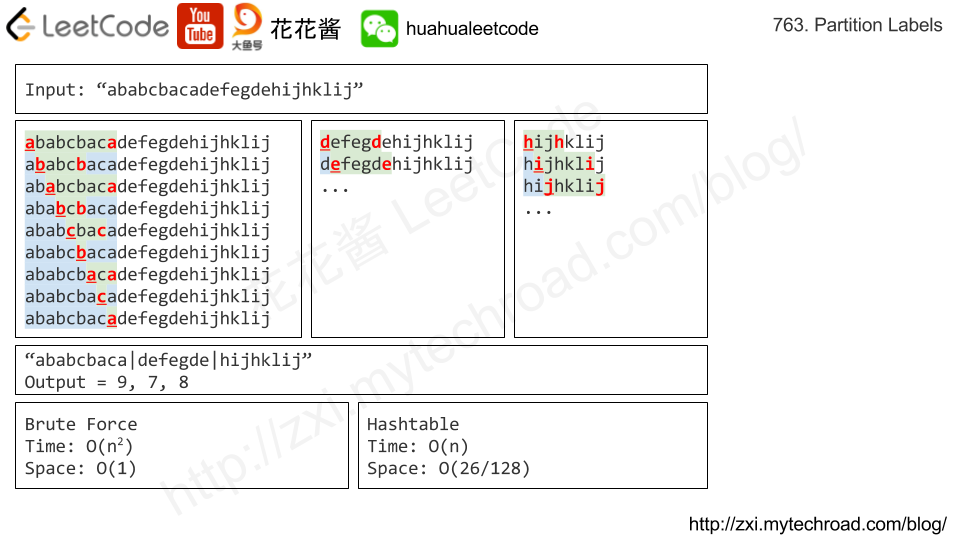
Note:

1. S will have length in range [1, 500].
2. S will consist of lowercase letters ('a' to 'z') only.

Sol:

先創建一個陣列 last 紀錄每個字母出現過的。

接著更新當前遇到的段落結尾。(段落結尾為max(end,last[S[i]-'a'])  
因為若當前出現的字母最後一次出現的位置比目前end還長，代表目前的end不能當作段落結尾。而若更新完後結尾和當前位置相同，代表目前為一個段落，便紀錄當前長度和更新起始點





122. Best Time to Buy and Sell Stock II(Easy)

Say you have an array prices 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 (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

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

Output: 7

Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Example 2:

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

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are

engaging multiple transactions at the same time. You must sell before buying again.

Example 3:

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

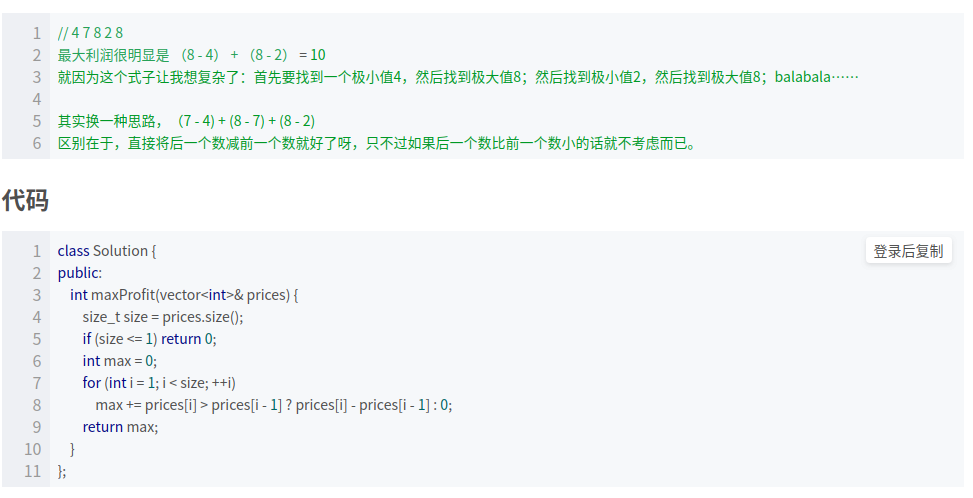
Output: 0

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

Constraints:

* 1 <= prices.length <= 3 \* 10 ^ 4
* 0 <= prices[i] <= 10 ^ 4

Sol:



**406. Queue Reconstruction by Height(Medium)**

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

Note:

The number of people is less than 1,100.

Example

Input:

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:

[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

Sol:

題目的主要意思為，給你一串陣列，讓你對其進行“排序”，每個元素由兩個值構成(a,b)，a代表這個字元的大小，b代表有多少個比他大或者等於他的數字排在他前面，這就是最後生成的陣列需要滿足的條件。

首先我們可以對陣列按照a進行從大到小的排序，如果a相同的話，就按照b來排，按照題目的要求，應該是b小的排在前面。定義一個結果陣列res，排好序的陣列在這個res裡面的順序，就是b的值的大小了

比如此時的順序為：

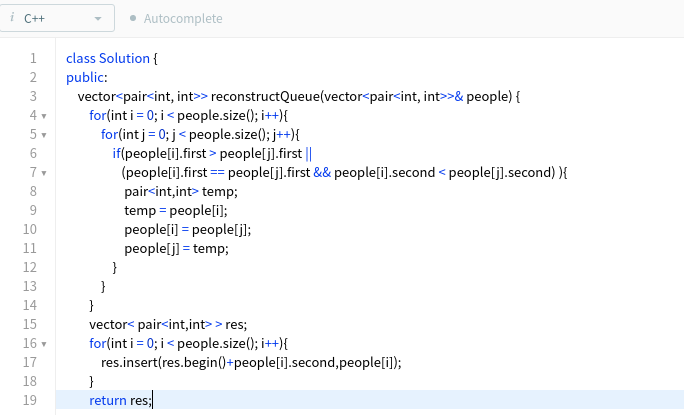
[[7,0], [7,1] , [6,1] , [5,0]] , [5,2] , [4,4] ]

那麼首先插入[7,0]在res[0]的位置，[7,1]在res[1]，

[6,1]由於b是1，那麼他就在res[1]，此時[7,1]向後挪一位，所以res=[[7,0],[6,1],[7,1]]

接著是[5,0]因為b=0，所以他插在res[0]的位置，以此類推。

主要是參考了discuss



**665. Non-decreasing Array(Easy)**

Given an array nums with n integers, your task is to check if it could become non-decreasing by modifying at most 1 element.

We define an array is non-decreasing if nums[i] <= nums[i + 1] holds for every i (0-based) such that (0 <= i <= n - 2).

Example 1:

Input: nums = [4,2,3]

Output: true

Explanation: You could modify the first 4 to 1 to get a non-decreasing array.

Example 2:

Input: nums = [4,2,1]

Output: false

Explanation: You can't get a non-decreasing array by modify at most one element.

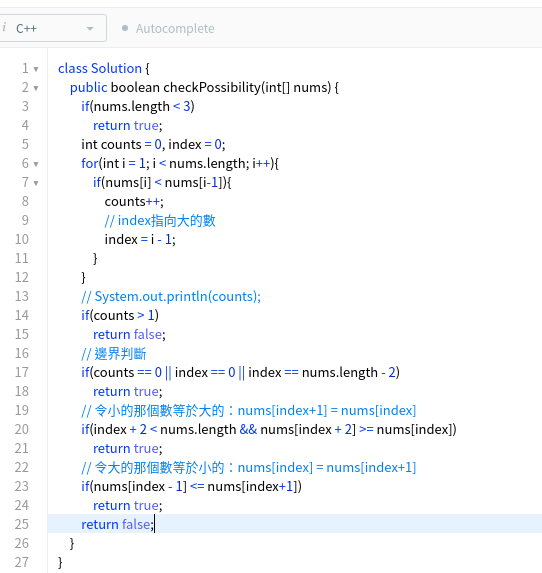
Constraints:

* 1 <= n <= 10 ^ 4
* - 10 ^ 5 <= nums[i] <= 10 ^ 5

Sol:

最多修改陣列中一個數的值，判斷是否可以使它是單調不減陣列，array[i] <= array[i + 1] ，如果可以就返回true，否則false

思路也很簡單，統計反序的次數。 如果次數大於一次，那麼肯定是不可以通過修改一個數的值，使之有序； 而對於只出現一次反序的情況，處理方式是讓反序的兩個數相等，此時，我們就需要判斷等於誰，才可以使陣列有序，分兩種情況： ①讓他們都等於大的數； ②等於小的那個數。 只要其中之一操作可以使陣列有序，就返回true，否則，就返回false。



Time: O(N)

Space:O(1)

**Chapter.3 Two pointer(2/4)**

**633. Sum of Square Numbers(Easy)**

Given a non-negative integer c, your task is to decide whether there're two integers a and b such that a^2 + b^2 = c.

Example 1:

Input: 5

Output: True

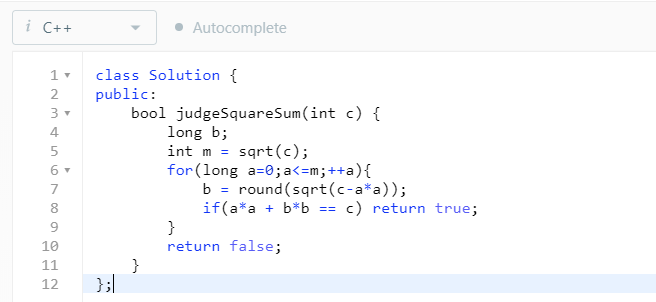
Explanation: 1 \* 1 + 2 \* 2 = 5

Example 2:

Input: 3

Output: False

從0~sqrt(c) 一個一個嘗試，直到剛好有可以滿足a^2 + b^2 =c為止。



**680. Valid Palindrome II(Easy)**

Given a non-empty string s, you may delete at most one character. Judge whether you can make it a palindrome.

Example 1:

Input: "aba" Output: True

Example 2:

Input: "abca" Output: True

Explanation: You could delete the character 'c'.

Note:

1. The string will only contain lowercase characters a-z. The maximum length of the string is 50000.

Sol:  
當遇到不匹配的時候，我們到底是刪除左邊的字符，還是右邊的字符呢，我們的做法是兩種情況都要算一遍，只要有一種能返回true，那麼結果就返回true。我們可以寫一個子函數來判斷字符串的某一個範圍內的子字符串是否為回文串



**524. Longest Word in Dictionary through Deleting(Medium)**

Given a string and a string dictionary, find the longest string in the dictionary that can be formed by deleting some characters of the given string. If there are more than one possible results, return the longest word with the smallest lexicographical order. If there is no possible result, return the empty string.

Example 1:

Input:

s = "abpcplea", d = ["ale","apple","monkey","plea"]

Output:

"apple"

Example 2:

Input:

s = "abpcplea", d = ["a","b","c"]

Output:

"a"

Note:

1. All the strings in the input will only contain lower-case letters.
2. The size of the dictionary won't exceed 1,000.
3. The length of all the strings in the input won't exceed 1,000.

Sol:

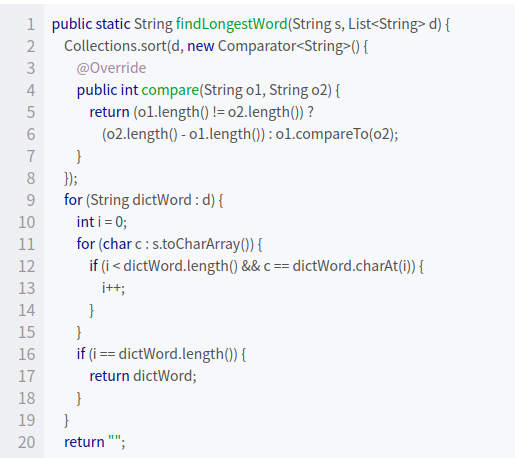
题目意思是要找出给出的输入字符串去除某些字符后能拼接成的在字典中的最长字典字符串。

如果按照题目给的意思，通过删除字符来判断剩下的字符串是否在字典里无疑是个非常繁重的工作。这个题应该反过来想：对于字典中的每个字典单词串，判断能否由输入字符串中的某些字符元素组成。方法分为两步：1.对字典字符串排序；2.查找到能最优先被输入字符串匹配的字典字符串。

题目中对结果有以下要求：返回最长或长度一样时返回字母表中最前的。那么可以对字典中的字符串按照这两个要求排序：长度降序、长度相同时字母表升序。这样遍历字典字符串列表，第一个能被输入字符串去掉某些字符表示出的字典字符串即为所求。

如何判断字典字符串能被输入字符串去除某些元素后表示出来？方法是对输入字符串的每个字符从左到右遍历，如果输入字符串的某位和字典字符串的第i位相同，那么字典字符串的指针i++指向下一位字符，再判断输入字符串此后是否存在和字典字符串当前位相同的字符，以此类推。

循环结束的i是字典字符串和输入字符串匹配的字符个数，如果i等于字典字符串的长度说明已经输入字符串能通过去掉若干字符的方式表示出字典字符串，循环结束，返回此字典字符串。



**Chapter.4 Binary Search**

**154. Find Minimum in Rotated Sorted Array II(Medium)**Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

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

Find the minimum element.

The array may contain duplicates.

Example 1:

Input: [1,3,5]

Output: 1

Example 2:

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

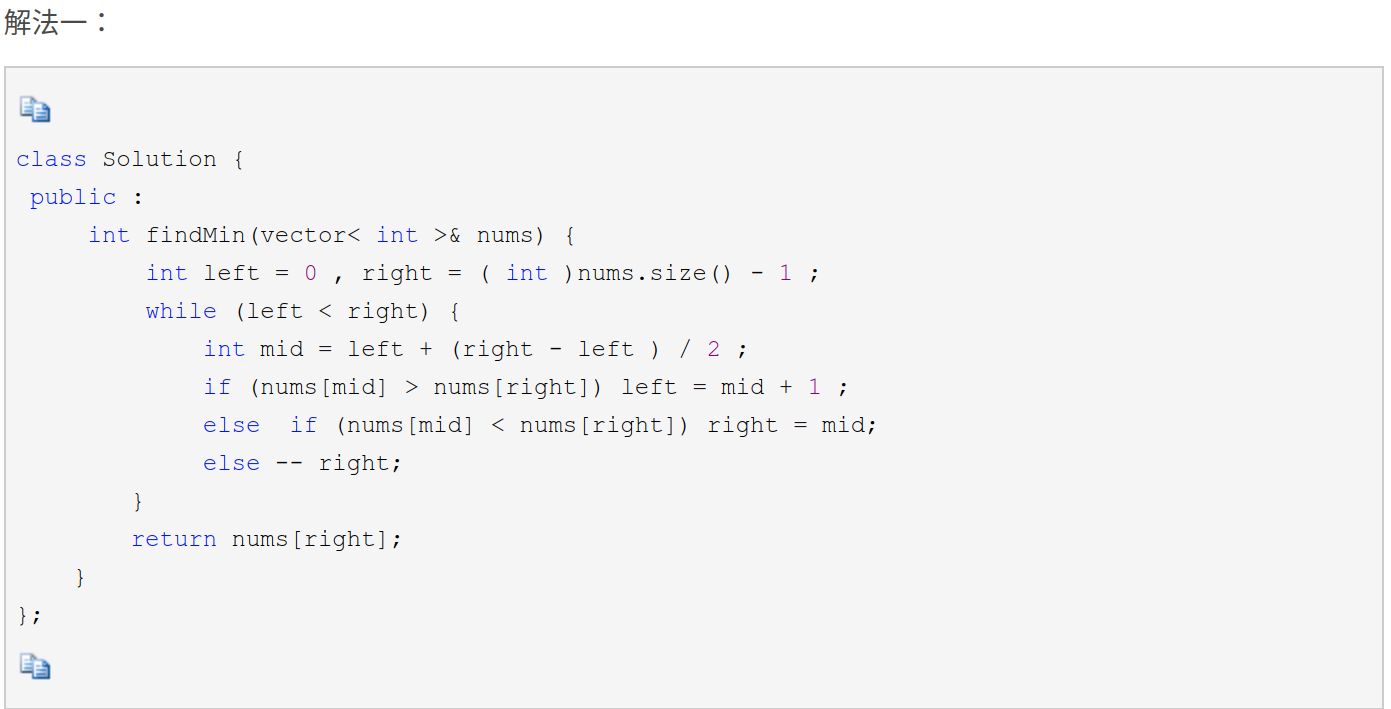
Output: 0

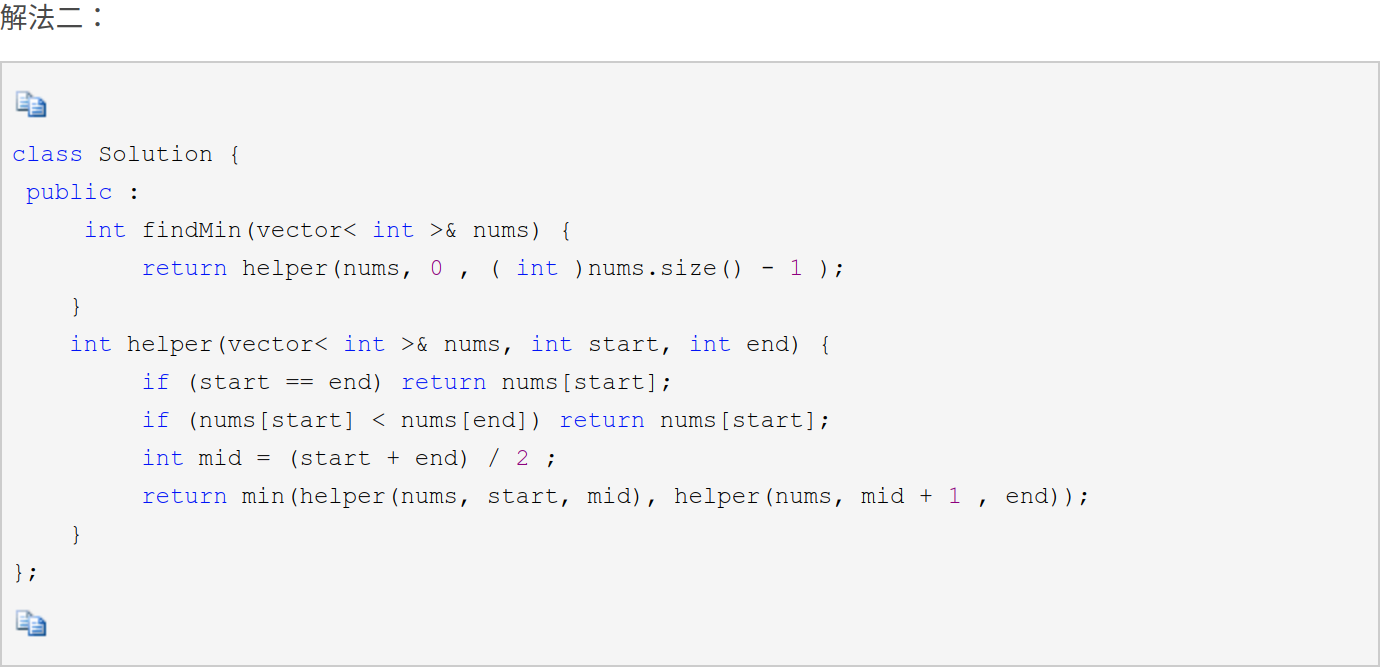
Note:

* This is a follow up problem to [Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/description/).
* Would allow duplicates affect the run-time complexity? How and why?

Sol:

這道尋找旋轉有序重複數組的最小值是之前那道[Find Minimum in Rotated Sorted Array](http://www.cnblogs.com/grandyang/p/4032934.html) 的拓展，當數組中存在大量的重複數字時，就會破壞二分查找法的機制，將無法取得O(lgn)的時間複雜度，又將會回到簡單粗暴的O(n)，比如這兩種情況：{2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2}和{2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}，可以發現，當第一個數字和最後一個數字，還有中間那個數字全部相等的時候，二分查找法就崩潰了，因為它無法判斷到底該去左半邊還是右半邊。這種情況下，將右指針左移一位（或者將左指針右移一位），略過一個相同數字，這對結果不會產生影響，因為只是去掉了一個相同的，然後對剩餘的部分繼續用二分查找法，在最壞的情況下，比如數組所有元素都相同，時間複雜度會升到O(n)，參見代碼如下：



跟之前那道[Find Minimum in Rotated Sorted Array](http://www.cnblogs.com/grandyang/p/4032934.html) 一樣，還是可以用分治法Divide and Conquer來解，還是由熱心網友 [howard144](https://www.cnblogs.com/grandyang/p/4040438.html#4313406) 提供，不過寫法跟之前那道略有不同，只有在nums[start] < nums[ end]的時候，才能返回nums[start]，等於的時候不能返回，比如[3, 1, 3]這個數組，或者當start等於end成立的時候，也可以直接返回nums[start]，後面的操作跟之前那道題相同，每次將區間[start, end]從中間mid位置分為兩段，分別調用遞歸函數，並比較返回值，每次取返回值較小的那個即可，參見代碼如下：  


**540. Single Element in a Sorted Array(Medium)**

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Find this single element that appears only once.

Follow up: Your solution should run in O(log n) time and O(1) space.

Example 1:

Input: nums = [1,1,2,3,3,4,4,8,8]

Output: 2

Example 2:

Input: nums = [3,3,7,7,10,11,11]

Output: 10

Constraints:

* 1 <= nums.length <= 10^5
* 0 <= nums[i] <= 10^5

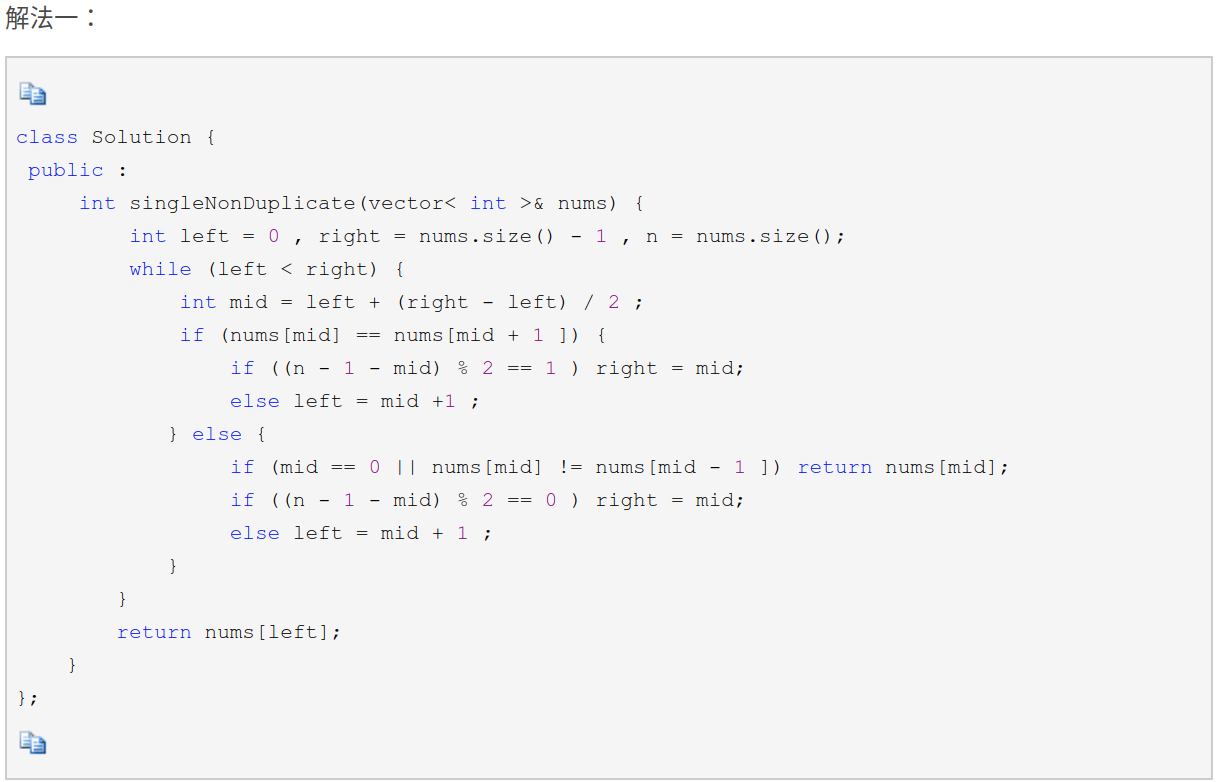
Sol:

這道題給我們了一個有序數組，說是所有的元素都出現了兩次，除了一個元素，讓我們找到這個元素。如果沒有時間複雜度的限制，我們可以用多種方法來做，最straightforward的解法就是用個雙指針，每次檢驗兩個，就能找出落單的。也可以像[Single Number](http://www.cnblogs.com/grandyang/p/4130577.html)裡的方法那樣，將所有數字亦或起來，相同的數字都會亦或成0，剩下就是那個落單的數字。那麼由於有了時間複雜度的限制，需要為O(logn)，而數組又是有序的，不難想到要用二分搜索法來做。二分搜索法的難點在於折半了以後，如何判斷將要去哪個分支繼續搜索，而這道題確實判斷條件不明顯，比如下面兩個例子：

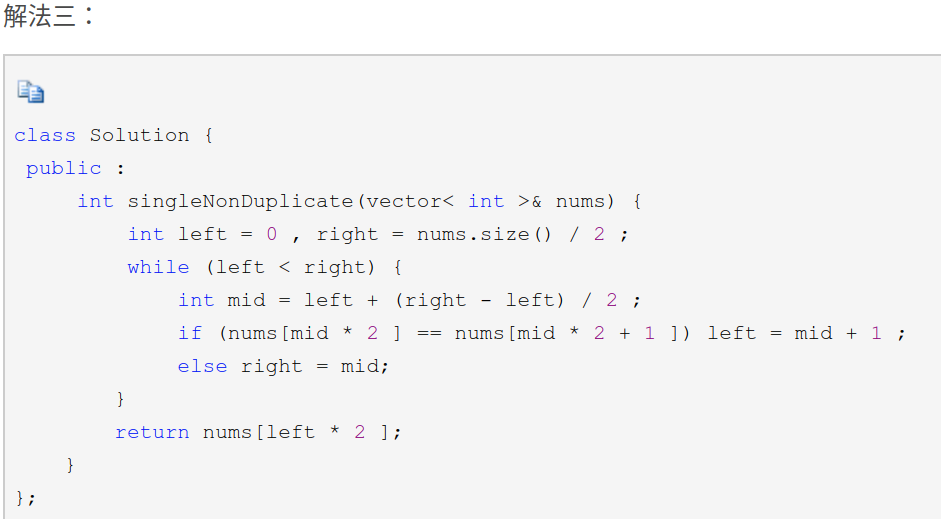
1 1 2 2 3

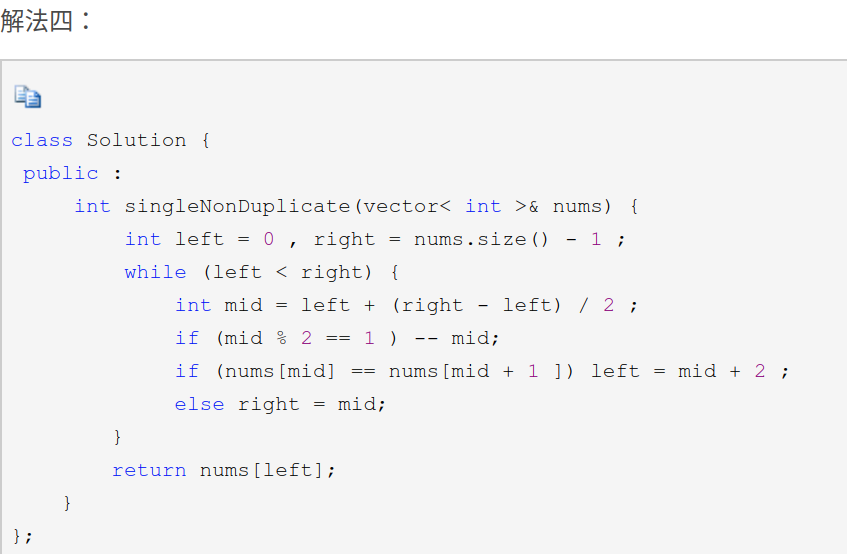
1 2 2 3 3

這兩個例子初始化的時候left=0, right=4一樣，mid算出來也一樣為2，但是他們要去的方向不同，如何區分出來呢？仔細觀察我們可以發現，如果當前數字出現兩次的話，我們可以通過數組的長度跟當前位置的關係，計算出右邊和當前數字不同的數字的總個數，如果是偶數個，說明落單數左半邊，反之則在右半邊。有了這個規律就可以寫代碼了，為啥我們直接就能跟mid+1比呢，不怕越界嗎？當然不會，因為left如何跟right相等，就不會進入循環，所以mid一定會比right小，一定會有mid+1存在。當然mid是有可能為0的，所以此時當mid和mid+1的數字不等時，我們直接返回mid的數字就可以了，參見代碼如下：



下面這種解法是對上面的分支進行合併，使得代碼非常的簡潔。使用到了亦或1這個小技巧，為什麼要亦或1呢，原來我們可以將坐標兩兩歸為一對，比如0和1，2和3，4和5等等。而亦或1可以直接找到你的小伙伴，比如對於2，亦或1就是3，對於3，亦或1就是2。如果你和你的小伙伴相等了，說明落單數在右邊，如果不等，說明在左邊，這方法，太叼了有木有，參見代碼如下：****

下面這種解法其實跟上面的方法其實有些類似，雖然沒有亦或1，但是將right縮小了一倍，但是在比較的時候，是比較mid\*2和mid\*2+1的關係的，這樣還是能正確的比較原本應該相等的兩個小伙伴的值的，其實核心思路和上面一樣，參見代碼如下：  


下面這種方法其實跟解法二很像，沒有用亦或1，但是對mid進行了處理，強制使其成為小伙伴對兒中的第一個位置，然後跟另一個小伙伴比較大小，參見代碼如下：  


**4. Median of Two Sorted Arrays(Hard)**  
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)).

You may assume nums1 and nums2 cannot be both empty.

Example 1:

nums1 = [1, 3]

nums2 = [2]

The median is 2.0

Example 2:

nums1 = [1, 2]

nums2 = [3, 4]

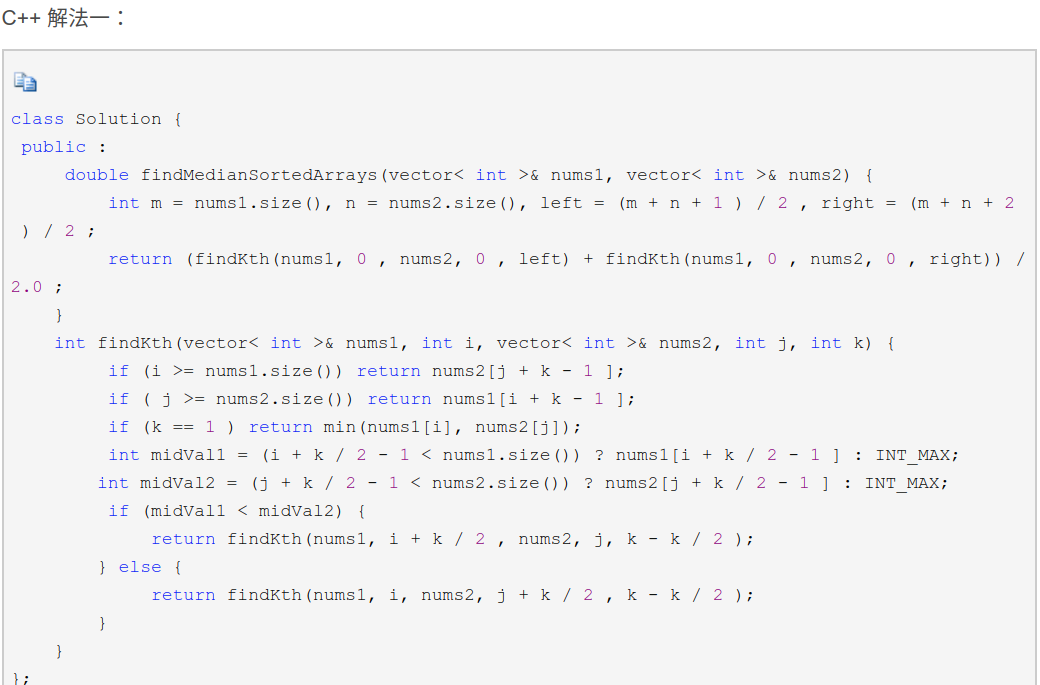
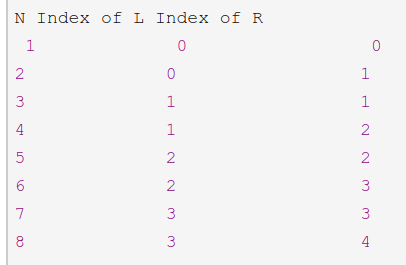
The median is (2 + 3)/2 = 2.5

Sol:

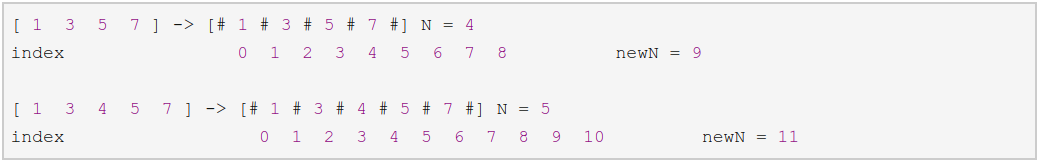
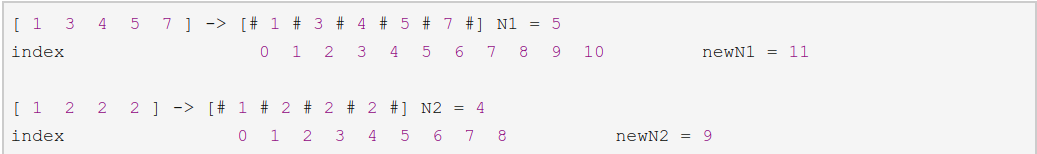
這道題讓我們求兩個有序數組的中位數，而且限制了時間複雜度為O(log (m+n))，看到這個時間複雜度，自然而然的想到了應該使用二分查找法來求解。但是這道題被定義為Hard 也是有其原因的，難就難在要在兩個未合併的有序數組之間使用二分法，如果這道題只有一個有序數組，讓求中位數的話，估計就是個Easy 題。對於這道題來說，可以將兩個有序數組混合起來成為一個有序數組再做嗎，圖樣圖森破，這個時間複雜度限制的就是告訴你金坷垃別想啦。還是要用二分法，而且是在兩個數組之間使用，感覺很高端啊。回顧一下中位數的定義，如果某個有序數組長度是奇數，那麼其中位數就是最中間那個，如果是偶數，那麼就是最中間兩個數字的平均值。這裡對於兩個有序數組也是一樣的，假設兩個有序數組的長度分別為m和n，由於兩個數組長度之和m+n 的奇偶不確定，因此需要分情況來討論，對於奇數的情況，直接找到最中間的數即可，偶數的話需要求最中間兩個數的平均值。為了簡化代碼，不分情況討論，使用一個小trick，分別找第(m+n+1) / 2 個，和(m+n+2) / 2 個，然後求其平均值即可，這對奇偶數均適用。若m+n 為奇數的話，那麼其實(m+n+1) / 2 和(m+n+2) / 2 的值相等，相當於兩個相同的數字相加再除以2，還是其本身。

好，這裡需要定義一個函數來在兩個有序數組中找到第K個元素，下面重點來看如何實現找到第K個元素。首先，為了避免拷貝產生新的數組從而增加時間複雜度，使用兩個變量i和j分別來標記數組nums1 和nums2 的起始位置。然後來處理一些corner cases，比如當某一個數組的起始位置大於等於其數組長度時，說明其所有數字均已經被淘汰了，相當於一個空數組了，那麼實際上就變成了在另一個數組中找數字，直接就可以找出來了。還有就是如果K=1 的話，只要比較nums1 和nums2 的起始位置i和j上的數字就可以了。難點就在於一般的情況怎麼處理？因為需要在兩個有序數組中找到第K個元素，為了加快搜索的速度，可以使用二分法，那麼對誰二分呢，數組麼？其實要對K二分，意思是需要分別在nums1 和nums2 中查找第K/2 個元素，注意這裡由於兩個數組的長度不定，所以有可能某個數組沒有第K/2 個數字，所以需要先check 一下，數組中到底存不存在第K/2 個數字，如果存在就取出來，否則就賦值上一個整型最大值（目的是要在nums1 或者nums2 中先淘汰K/2 個較小的數字，判斷的依據就是看midVal1 和midVal2 誰更小，但如果某個數組的個數都不到K/2 個，自然無法淘汰，所以將其對應的midVal 值設為整型最大值，以保證其不會被淘汰），若某個數組沒有第K/2 個數字，則淘汰另一個數組的前K/2 個數字即可。舉個例子來說吧，比如nums1 = {3}，nums2 = {2, 4, 5, 6, 7}，K=4，要找兩個數組混合中第4個數字，則分別在nums1 和nums2中找第2個數字，而nums1 中只有一個數字，不存在第二個數字，則nums2 中的前2個數字可以直接跳過，為啥呢，因為要求的是整個混合數組的第4個數字，不管nums1 中的那個數字是大是小，第4個數字絕不會出現在nums2 的前兩個數字中，所以可以直接跳過。

有沒有可能兩個數組都不存在第K/2 個數字呢，這道題裡是不可能的，因為K不是任意給的，而是給的m+n 的中間值，所以必定至少會有一個數組是存在第K/2 個數字的。最後就是二分法的核心啦，比較這兩個數組的第K/2 小的數字midVal1 和midVal2 的大小，如果第一個數組的第K/2 個數字小的話，那麼說明要找的數字肯定不在nums1 中的前K/2 個數字，可以將其淘汰，將nums1 的起始位置向後移動K/2 個，並且此時的K也自減去K/2，調用遞歸，舉個例子來說吧，比如nums1 = {1, 3}，nums2 = {2, 4, 5}，K=4，要找兩個數組混合中第4個數字，那麼分別在nums1 和nums2 中找第2個數字， nums1 中的第2個數字是3，nums2 中的第2個數字是4，由於3小於4，所以混合數組中第4個數字肯定在nums2 中，可以將nums1 的起始位置向後移動K/ 2 個。反之，淘汰nums2 中的前K/2 個數字，並將nums2 的起始位置向後移動K/2 個，並且此時的K也自減去K/2，調用遞歸即可，參見代碼如下：

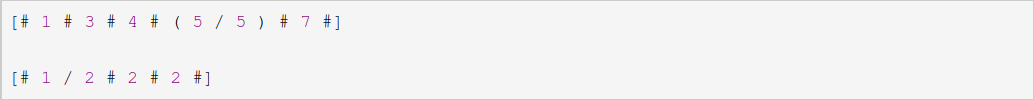
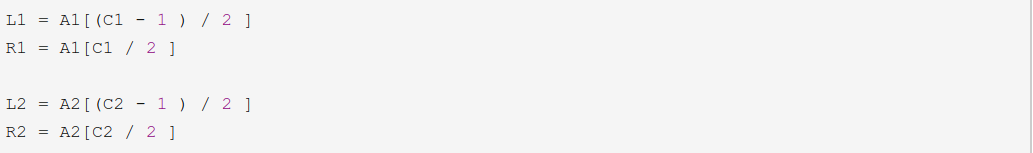
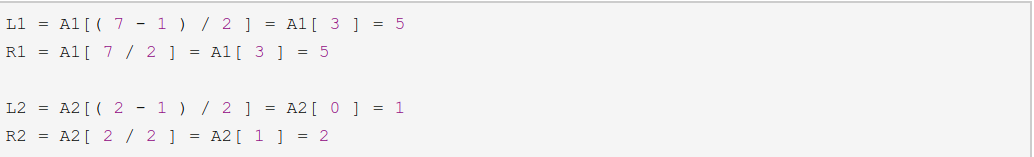
  
上面的解法一直使用的是原數組，同時用了兩個變量來分別標記當前的起始位置。我們也可以直接生成新的數組，這樣就不要用起始位置變量了，不過拷貝數組的操作可能會增加時間複雜度，也許會超出限制，不過就算當個思路拓展也是極好的。首先要判斷數組是否為空，為空的話，直接在另一個數組找第K個即可。還有一種情況是當K = 1 時，表示要找第一個元素，只要比較兩個數組的第一個元素，返回較小的那個即可。這里分別取出兩個數組的第K/2 個數字的位置坐標i和j，為了避免數組沒有第K/2 個數組的情況，每次都和數組長度做比較，取出較小值。這裡跟上面的解法有些許不同，上面解法直接取出的是值，而這裡取出的是位置坐標，但是思想都是很類似的。不同在於，上面解法中每次固定淘汰K/2 個數字，而這裡由於取出了合法的i和j，所以每次淘汰i或j個。評論區有網友提出，可以讓j = ki，這樣也是對的，可能還更好一些，收斂速度可能會更快一些，參見代碼如下：  
  
此題還能用迭代形式的二分搜索法來解，是一種相當巧妙的應用，這裡就參照[stellari大神的帖子](https://leetcode.com/problems/median-of-two-sorted-arrays/discuss/2471/very-concise-ologminmn-iterative-solution-with-detailed-explanation) 來講解吧。所謂的中位數，換一種角度去看，其實就是把一個有序數組分為長度相等的兩段，中位數就是前半段的最大值和後半段的最小值的平均數，也就是離分割點相鄰的兩個數字的平均值。比如說對於偶數個數組[1 3 5 7]，那麼分割開來就是[1 3 / 5 7]，其中'/'表示分割點，中位數就是3和5的平均值。對於奇數個數組[1 3 4 5 7]，可以分割為[1 3 4 / 4 5 7]，可以發現左右兩邊都有個4，則中位數是兩個4的平均數，還是4。這裡使用L表示分割點左邊的數字，R表示分割點右邊的數字，則對於[1 3 5 7]來說，L=3，R=5。對於[1 3 4 5 7]來說，L=4，R=4。那麼對於長度為N的數組來說，可以分別得到L和R的位置，如下所示：  


觀察上表，可以得到規律，Idx(L)= (N-1)/2，idx(R) = N/2，所以中位數可以用下式表示：  
(L + R) / 2 = (A[(N - 1 ) / 2 ] + A[N / 2 ]) / 2

為了統一數組長度為奇數和偶數的情況，可以使用一個小tricky，即在每個數字的兩邊都加上一個特殊字符，比如井號，這個tricky其實在馬拉車算法中也使用過，可以參見博主之前的帖子 [Manacher's Algorithm馬拉車算法](http://www.cnblogs.com/grandyang/p/4475985.html)。這樣做的好處是不管奇數或者偶數，加井號後數組的長度都是奇數，並且切割點的位置也是確定的，比如：  
  
這裡的N是原數組的長度，newN 是添加井號後新數組的長度，可以發現newN = 2N+1，而且切割點永遠都在新數組中坐標為N的位置，且idx(L) = ( N-1)/2，idx(R) = N/2，這裡的N就可以換成分割點的位置，豈不美哉（注意這裡的idx(L) 和idx(R) 表示的是在未填充#號的坐標位置）！現在假設有兩個數組：  
  
跟只有一個數組的情況類似，這裡需要找到一個切割點，使得其分別可以將兩個數組分成左右兩部分，需要滿足的是兩個左半邊中的任意一個數字都要小於兩個右半邊數組的數字，注意這裡可能有的左半邊或右半邊會為空，但是兩個左半邊數字的個數和應該等於兩個右半邊的個數和。這裡還可以觀察出一些規律：

1. 總共有2N1 + 2N2 + 2 個位置，那麼除去兩個分割點，兩個左右半邊應該各有N1 + N2 個數字。

2. 因此，對於一個在A2 數組中的分割點位置C2 = K，在A1 數組中的位置應該為C1 = N1 + N2 - K，比如假如在A2 中的分割點位置為C2 = 2，那麼在A1 中的位置為C1 = 4 + 5 - C2 = 7。

  
3. 假如兩個數組都被分割了，那麼就應該會有兩個L和R，分別是：  
  
對於上面的例子就有：  
  
現在需要檢測這個切割點是否是正確的中位數的切割點，那麼根據之前的分析，任意的左半邊的數字都需要小於等於右半邊的數字，L1 和L2 是左半邊的最大的數字，R1和R2 是右半邊的最小的數字，所以需要滿足下列關係：  
  
由於兩個數組都是有序的，所以L1 <= R1 和L2 <= R2 都是滿足的，那麼就只需要滿足下列的不等式即可：  
  
這樣的話就可以利用二分搜索了，假如L1 > R2 的話，說明數組A1 的左半邊的數字過大了，需要把切割點C1 往左移動。假如L2 > R1，說明數組A2 的左半邊數字過大，需要把分割點C2 左移。若滿足上面的條件，說明當前切割點就是正確的，那麼中位數就可以求出來了，即為：  
  
最後還有兩點注意事項：  
1. 由於C1 和C2 是可以互相計算而得，即一個確定了，另一個就可以計算出來了。所以盡量去移動較短的那個數組，這樣得到的時間複雜度為O(lg(min(N1, N2)))。  
2. 對於corner case 的處理，當切割點在0 或者2n 的位置時，將L或R的值分別賦值為整型最小值和最大值，這不會改變正確的切割點的位置，會使得代碼實現更加方便。



**Chapter.5 Sorting**

**451. Sort Characters By Frequency(Medium)**Given a string, sort it in decreasing order based on the frequency of characters.  
Example 1:

Input:"tree"

Output:"eert"

Explanation:

'e' appears twice while 'r' and 't' both appear once.So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

Example 2:

Input:"cccaaa"

Output:"cccaaa"

Explanation:

Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.Note that "cacaca" is incorrect, as the same characters must be together.

Example 3:

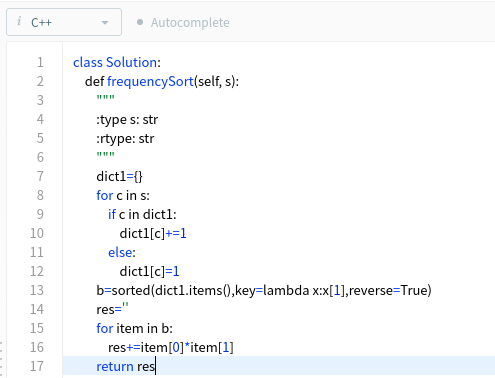
Input:"Aabb"

Output:"bbAa"

Explanation:

"bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

Sol: 字典儲存 字典排序 遍歷排序後的list  


75. Sort Colors(Medium)  
Given an array with *n* objects colored red, white or blue, sort them [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) 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.

Example:

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

Output: [0,0,1,1,2,2]

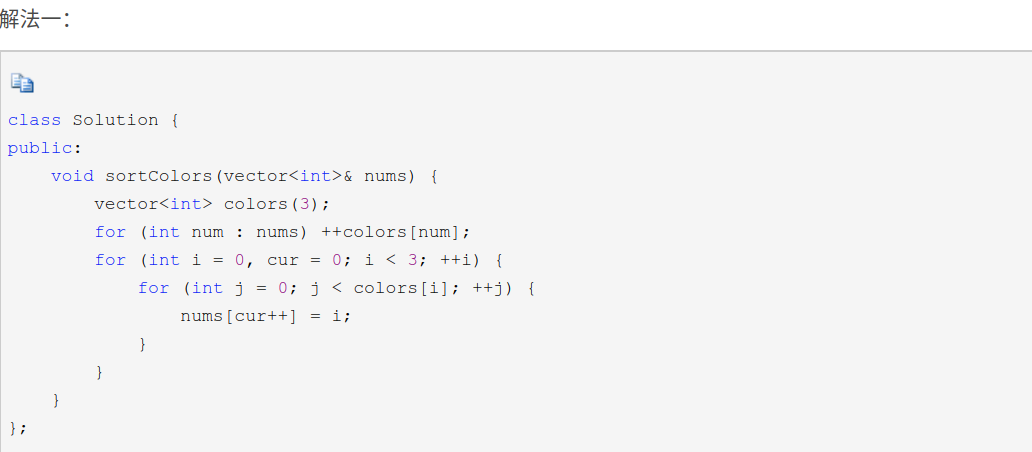
Follow up:

* A rather straight forward solution is a two-pass algorithm using counting sort.  
  First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
* Could you come up with a one-pass algorithm using only constant space?

Sol:  
這道題的本質還是一道排序的題，題目中給出提示說可以用計數排序，需要遍歷數組兩遍，那麼先來看這種方法，因為數組中只有三個不同的元素，所以實現起來很容易。

- 首先遍歷一遍原數組，分別記錄 0，1，2 的個數。

- 然後更新原數組，按個數分別賦上 0，1，2。

  
題目中還要讓只遍歷一次數組來求解，那麼就需要用雙指針來做，分別從原數組的首尾往中心移動。

- 定義 red 指針指向開頭位置，blue 指針指向末尾位置。

- 從頭開始遍歷原數組，如果遇到0，則交換該值和 red 指針指向的值，並將 red 指針後移一位。若遇到2，則交換該值和 blue 指針指向的值，並將 blue 指針前移一位。若遇到1，則繼續遍歷。

  
當然我們也可以使用 while 循環的方式來寫，那麼就需要一個變量 cur 來記錄當前遍歷到的位置，參見代碼如下：  
  
  
  
Chapter.6 BFS & DFS

**130. Surrounded Regions(Medium)**  
Given a 2D board containing 'X' and 'O' (the letter O), capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

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:

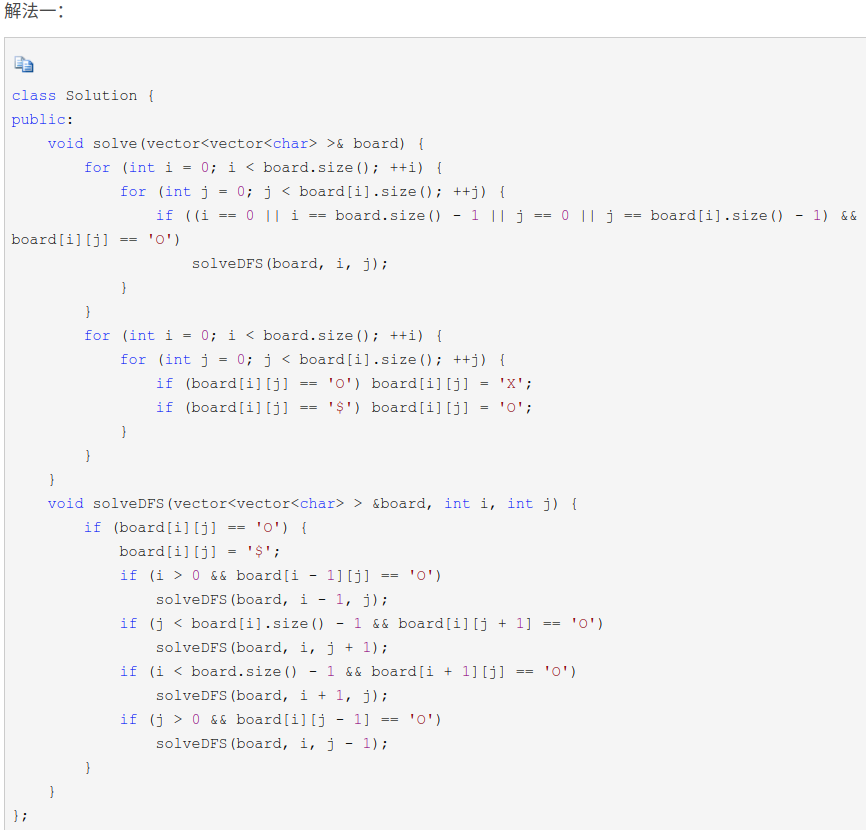
X X X X

X X X X

X X X X

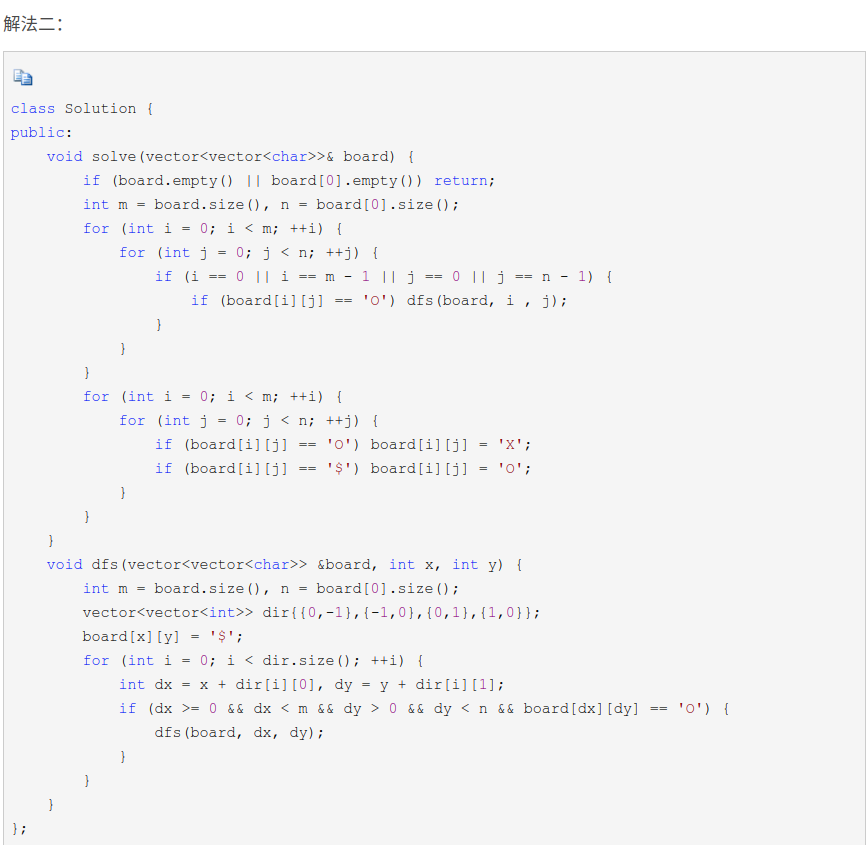
X O X X

Explanation:

Surrounded regions shouldn’t be on the border, which means that any 'O' on the border of the board are not flipped to 'X'. Any 'O' that is not on the border and it is not connected to an 'O' on the border will be flipped to 'X'. Two cells are connected if they are adjacent cells connected horizontally or vertically.  
  
Sol:  
這是道關於XXOO 的題，有點像圍棋，將包住的O都變成X，但不同的是邊緣的O不算被包圍，跟之前那道Number of Islands 很類似，都可以用DFS 來解。剛開始我的思路是 DFS 遍歷中間的O，如果沒有到達邊緣，都變成X，如果到達了邊緣，將之前變成X的再變回來。但是這樣做非常的不方便，在網上看到大家普遍的做法是掃矩陣的四條邊，如果有O，則用DFS 遍歷，將所有連著的O都變成另一個字符，比如$，這樣剩下的O都是被包圍的，然後將這些O變成X，把$變回O就行了。代碼如下：  


很久以前，上面的代碼中最後一個if 中必須是j > 1 而不是j > 0，為啥j > 0 無法通過OJ 的最後一個大數據集合，博主開始也不知道其中奧秘，直到被另一個網友提醒在本地機子上可以通過最後一個大數據集合，於是博主也寫了一個程序來驗證，請參見驗證LeetCode Surrounded Regions 包圍區域的DFS方法，發現j > 0 是正確的，可以得到相同的結果。神奇的是，現在用 j > 0 也可以通過 OJ 了。

下面這種解法還是 DFS 解法，只是遞歸函數的寫法稍有不同，但是本質上並沒有太大的區別，參見代碼如下：



我們也可以使用迭代的解法，但是整體的思路還是一樣的，在找到邊界上的O後，然後利用隊列 queue 進行 BFS 查找和其相連的所有O，然後都標記上美元號。最後的處理還是先把所有的O變成X，然後再把美元號變回O即可，參見代碼如下：



257. Binary Tree Paths  
Given a binary tree, return all root-to-leaf paths.  
Note: A leaf is a node with no children.  
Example:

Input:

1

/ \

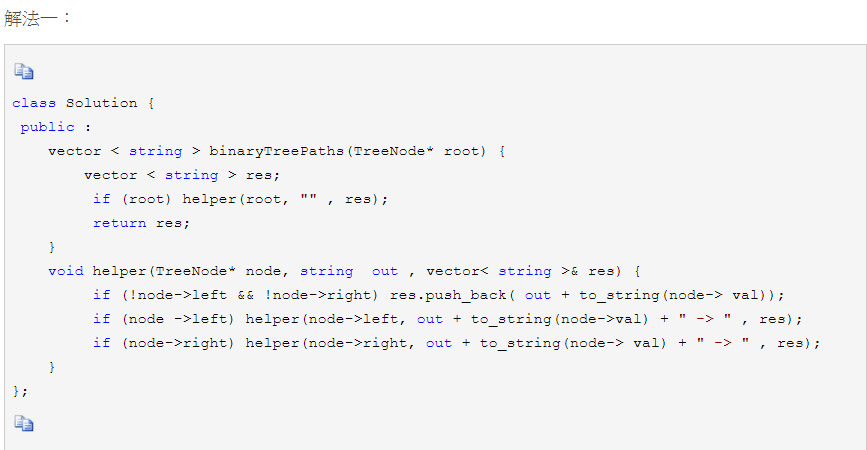
2 3

\

5

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

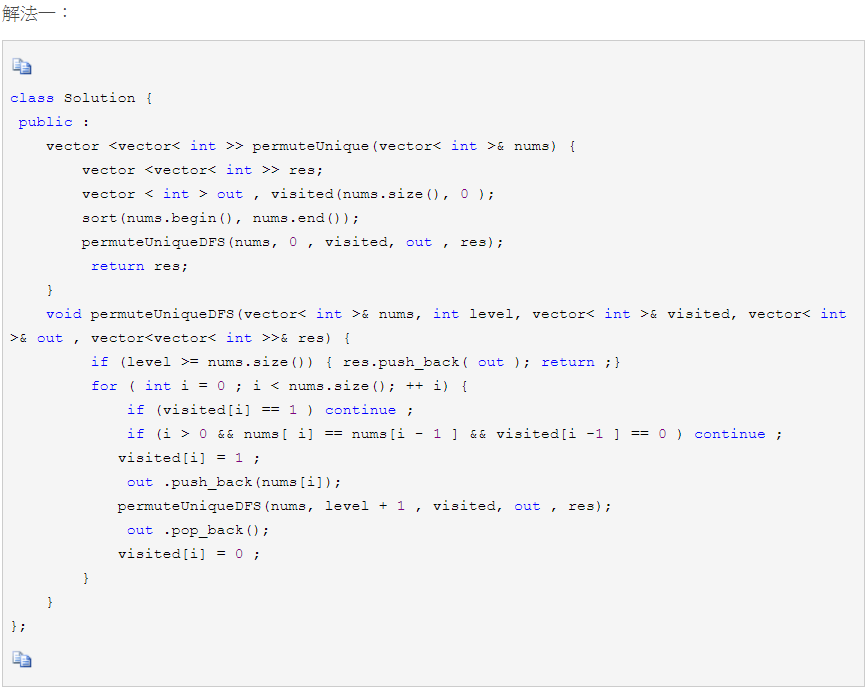
Explanation: All root-to-leaf paths are: 1->2->5, 1->3  
這道題給我們一個二叉樹，讓我們返回所有根到葉節點的路徑，跟之前那道[Path Sum II](http://www.cnblogs.com/grandyang/p/4042156.html)很類似，比那道稍微簡單一些，不需要計算路徑和，只需要無腦返回所有的路徑即可，那麼思路還是用遞歸來解，博主之前就強調過，玩樹的題目，十有八九都是遞歸，而遞歸的核心就是不停的DFS到葉結點，然後在回溯回去。在遞歸函數中，當我們遇到葉結點的時候，即沒有左右子結點，那麼此時一條完整的路徑已經形成了，我們加上當前的葉結點後存入結果res中，然後回溯。注意這裡結果res需要reference，而out是不需要引用的，不然回溯回去還要刪除新添加的結點，很麻煩。為了減少判斷空結點的步驟，我們在調用遞歸函數之前都檢驗一下非空即可，代碼而很簡潔，參見如下：

  
下面再來看一種遞歸的方法，這個方法直接在一個函數中完成遞歸調用，不需要另寫一個helper函數，核心思想和上面沒有區別，參見代碼如下：  
  
還是遞歸寫法，從論壇中扒下來的解法，核心思路都一樣啦，寫法各有不同而已，參見代碼如下：  
  
  
47. Permutations II  
Given a collection of numbers that might contain duplicates, return all possible unique permutations.

Example:

Input: [1,1,2]

Output:

[  
 [1,1,2],  
 [1,2,1],  
 [2,1,1]  
]  
  
這道題是之前那道 Permutations 的延伸，由於輸入數組有可能出現重複數字，如果按照之前的算法運算，會有重複排列產生，我們要避免重複的產生，在遞歸函數中要判斷前面一個數和當前的數是否相等，如果相等，且其對應的visited中的值為1，當前的數字才能使用（下文中會解釋這樣做的原因），否則需要跳過，這樣就不會產生重複排列了，代碼如下：  
  
  
40. Combination Sum II  
  
Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

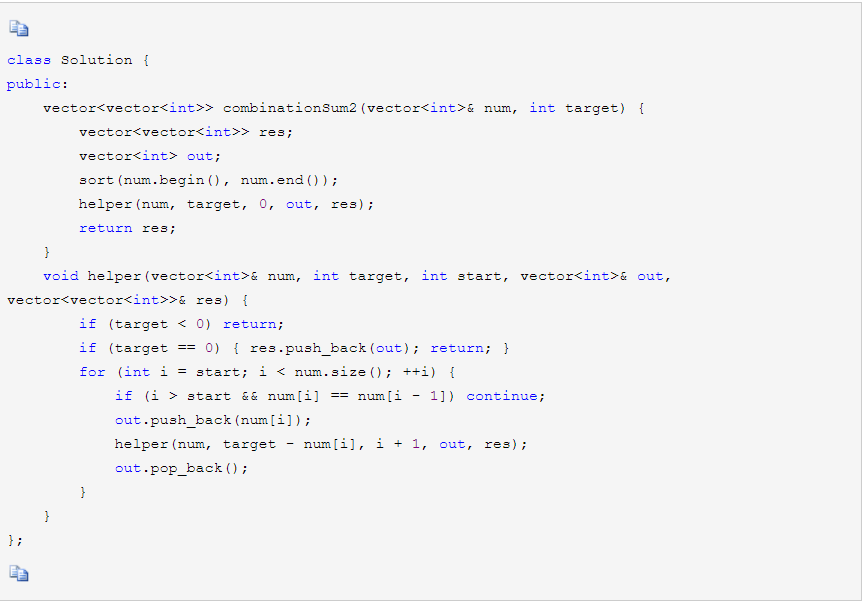
Each number in candidates may only be used once in the combination.

Note:  
All numbers (including target) will be positive integers.  
The solution set must not contain duplicate combinations.  
  
  
  
Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8,  
A solution set is:

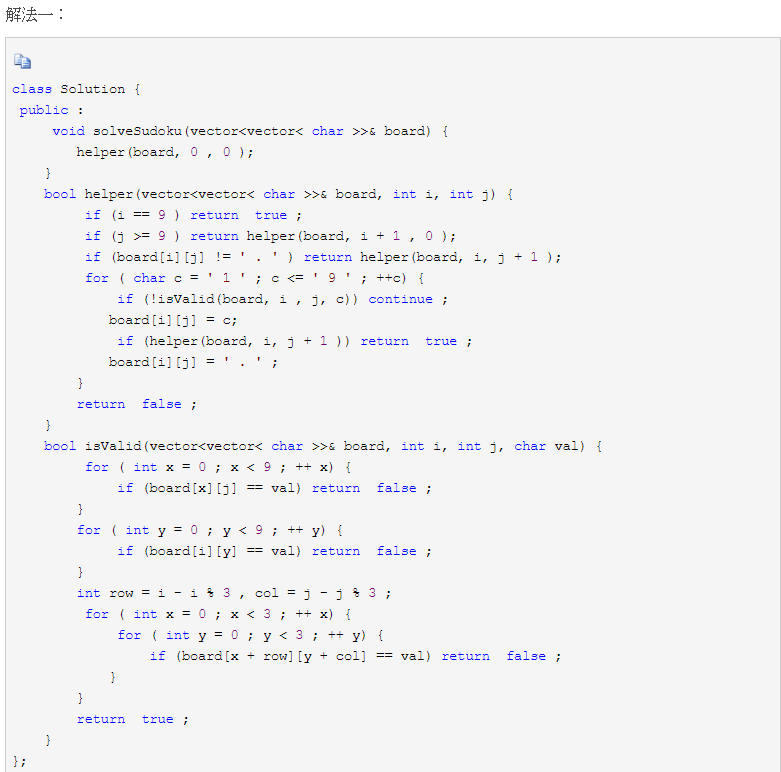
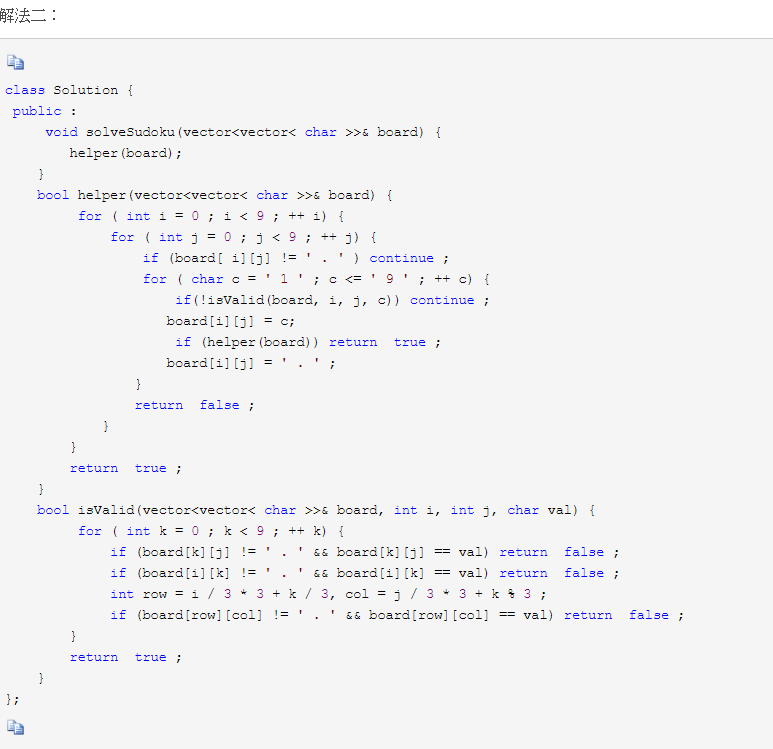
[  
 [1, 7],  
 [1, 2, 5],  
 [2, 6],  
 [1, 1, 6]  
]

Example 2:  
Input: candidates = [2,5,2,1,2], target = 5,  
A solution set is:

[  
 [1,2,2],  
 [5]  
]  
这道题跟之前那道 Combination Sum 本质没有区别，只需要改动一点点即可，之前那道题给定数组中的数字可以重复使用，而这道题不能重复使用，只需要在之前的基础上修改两个地方即可，首先在递归的 for 循环里加上 if (i > start && num[i] == num[i - 1]) continue; 这样可以防止 res 中出现重复项，然后就在递归调用 helper 里面的参数换成 i+1，这样就不会重复使用数组中的数字了，代码如下：  


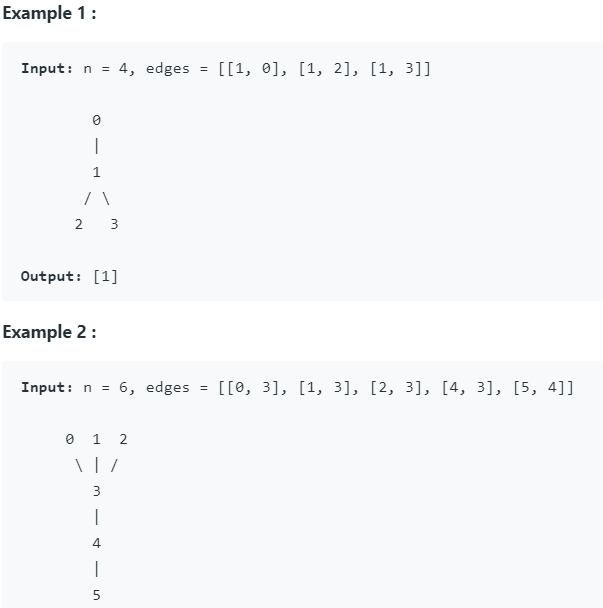
37. Sudoku Solver  
Write a program to solve a Sudoku puzzle by filling the empty cells.  
A sudoku solution must satisfy all of the following rules:  
Each of the digits 1-9 must occur exactly once in each row.  
Each of the digits 1-9 must occur exactly once in each column.  
Each of the the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.  
Empty cells are indicated by the character '.'.

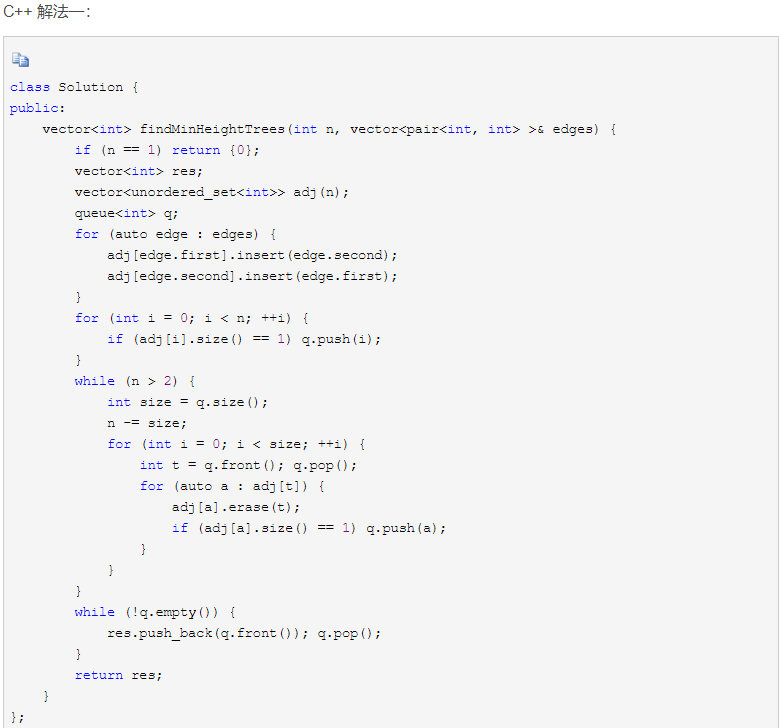
|  |  |
| --- | --- |
|  |  |

這道求解數獨的題是在之前那道 Valid Sudoku 的基礎上的延伸，之前那道題讓我們驗證給定的數組是否為數獨數組，這道讓求解數獨數組，跟此題類似的有 Permutations，Combinations， N-Queens 等等，其中尤其是跟 N-Queens 的解題思路及其相似，對於每個需要填數字的格子帶入1到9，每代入一個數字都判定其是否合法，如果合法就繼續下一次遞歸，結束時把數字設回'.'，判斷新加入的數字是否合法時，只需要判定當前數字是否合法，不需要判定這個數組是否為數獨數組，因為之前加進的數字都是合法的，這樣可以使程序更加高效一些，整體思路是這樣的，但是實現起來可以有不同的形式。一種實現形式是遞歸帶上橫縱坐標，由於是一行一行的填數字，且是從0行開始的，所以當i到達9的時候，說明所有的數字都成功的填入了，直接返回ture 。當j大於等於9時，當前行填完了，需要換到下一行繼續填，則繼續調用遞歸函數，橫坐標帶入i+1。否則看若當前數字不為點，說明當前位置不需要填數字，則對右邊的位置調用遞歸。若當前位置需要填數字，則應該嘗試填入1到9內的所有數字，讓c從1遍歷到9，每當試著填入一個數字，都需要檢驗是否有衝突，使用另一個子函數isValid來檢驗是否合法，假如不合法，則跳過當前數字。若合法，則將當前位置賦值為這個數字，並對右邊位置調用遞歸，若遞歸函數返回true，則說明可以成功填充，直接返回true。不行的話，需要重置狀態，將當前位置恢復為點。若所有數字都嘗試了，還是不行，則最終返回false。檢測當前數組是否合法的原理跟之前那道 Valid Sudoku 非常的相似，但更簡單一些，因為這裡只需要檢測新加入的這個數字是否會跟其他位置引起衝突，分別檢測新加入數字的行列和所在的小區間內是否有重複的數字即可，參見代碼如下：  
  
還有另一種遞歸的寫法，這裡就不帶橫縱坐標參數進去，由於遞歸需要有boolean 型的返回值，所以不能使用原函數。因為沒有橫縱坐標，所以每次遍歷都需要從開頭0的位置開始，這樣無形中就有了大量的重複檢測，導致這種解法雖然寫法簡潔一些，但擊敗率是沒有上面的解法高的。這裡的檢測數組衝突的子函數寫法也比上面簡潔不少，只用了一個for 循環，用來同時檢測行列和小區間是否有衝突，注意正確的坐標轉換即可，參見代碼如下：  
  
  
310. Minimum Height Trees  
For an undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format

The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges (each edge is a pair of labels).

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.  


这道题虽然是树的题目，但是跟其最接近的题目是Course Schedule 课程清单和Course Schedule II 课程清单之二。由于LeetCode中的树的题目主要都是针对于二叉树的，而这道题虽说是树但其实本质是想考察图的知识，这道题刚开始在拿到的时候，我最先想到的解法是遍历的点，以每个点都当做根节点，算出高度，然后找出最小的，但是一时半会又写不出程序来，于是上网看看大家的解法，发现大家推崇的方法是一个类似剥洋葱的方法，就是一层一层的褪去叶节点，最后剩下的一个或两个节点就是我们要求的最小高度树的根节点，这种思路非常的巧妙，而且实现起来也不难，跟之前那到课程清单的题一样，我们需要建立一个图g，是一个二维数组，其中g[i]是一个一维数组，保存了i节点可以到达的所有节点。我们开始将所有只有一个连接边的节点(叶节点)都存入到一个队列queue中，然后我们遍历每一个叶节点，通过图来找到和其相连的节点，并且在其相连节点的集合中将该叶节点删去，如果删完后此节点也也变成一个叶节点了，加入队列中，再下一轮删除。那么我们删到什么时候呢，当节点数小于等于2时候停止，此时剩下的一个或两个节点就是我们要求的最小高度树的根节点啦，参见代码如下：  


Chapter.7 Dynamic Programing

213. House Robber II  
You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

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

Example 1:

Input: [2,3,2]

Output: 3

Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2),

because they are adjacent houses.

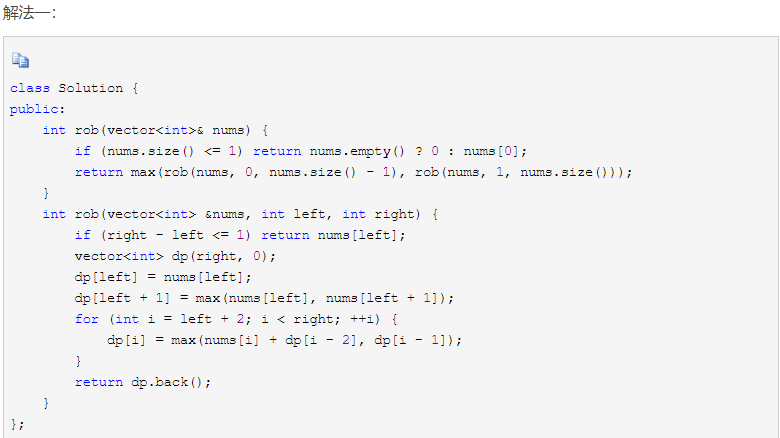
Example 2:

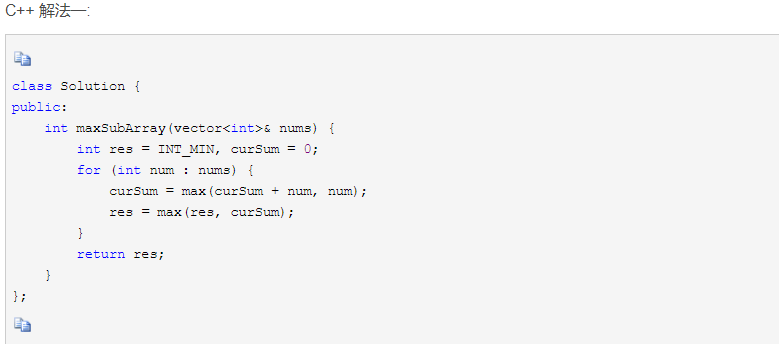
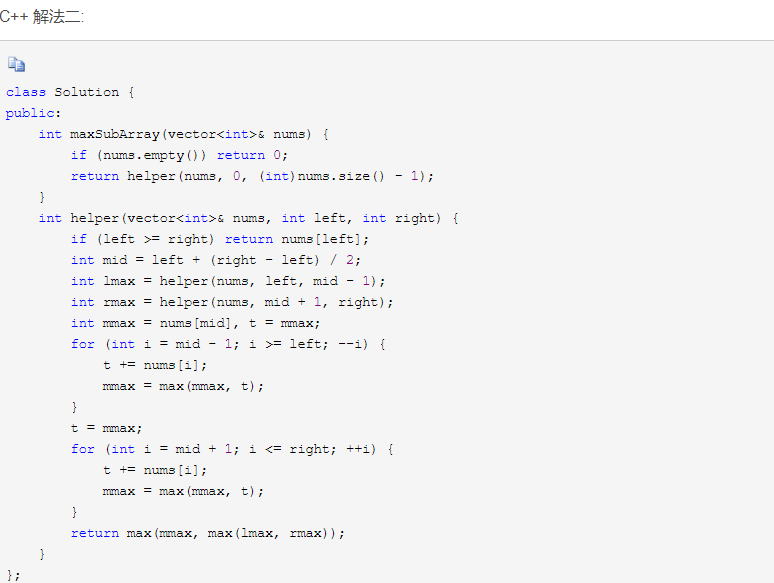
Input: [1,2,3,1]

Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

这道题是之前那道 House Robber 的拓展，现在房子排成了一个圆圈，则如果抢了第一家，就不能抢最后一家，因为首尾相连了，所以第一家和最后一家只能抢其中的一家，或者都不抢，那这里变通一下，如果把第一家和最后一家分别去掉，各算一遍能抢的最大值，然后比较两个值取其中较大的一个即为所求。那只需参考之前的 House Robber 中的解题方法，然后调用两边取较大值，代码如下  


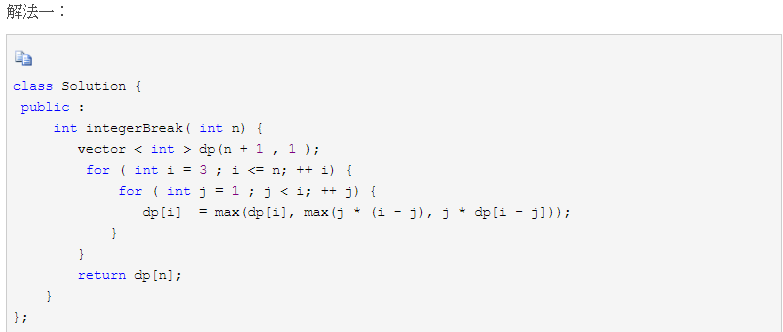
53. Maximum Subarray  
这道题让求最大子数组之和，并且要用两种方法来解，分别是 O(n) 的解法，还有用分治法 Divide and Conquer Approach，这个解法的时间复杂度是 O(nlgn)，那就先来看 O(n) 的解法，定义两个变量 res 和 curSum，其中 res 保存最终要返回的结果，即最大的子数组之和，curSum 初始值为0，每遍历一个数字 num，比较 curSum + num 和 num 中的较大值存入 curSum，然后再把 res 和 curSum 中的较大值存入 res，以此类推直到遍历完整个数组，可得到最大子数组的值存在 res 中，代码如下：  
  
题目还要求我们用分治法 Divide and Conquer Approach 来解，这个分治法的思想就类似于二分搜索法，需要把数组一分为二，分别找出左边和右边的最大子数组之和，然后还要从中间开始向左右分别扫描，求出的最大值分别和左右两边得出的最大值相比较取最大的那一个，代码如下：  
  
  
  
343. Integer Break

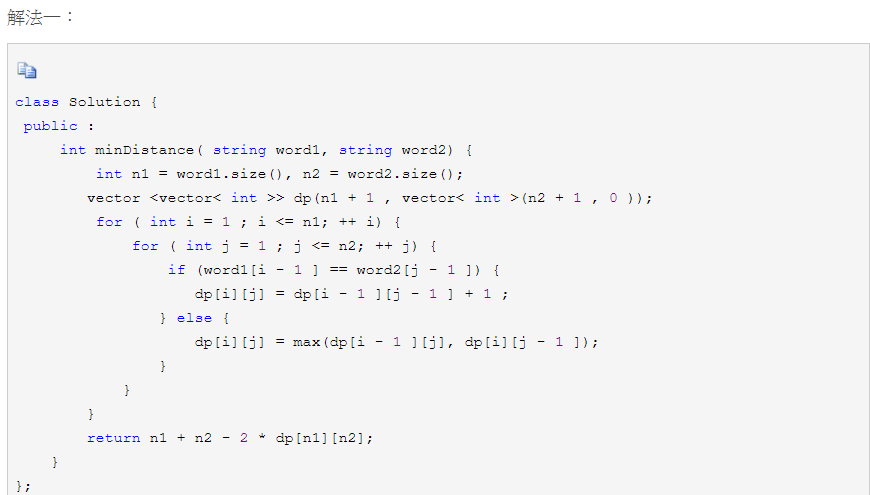
Given a positive integer n, break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.

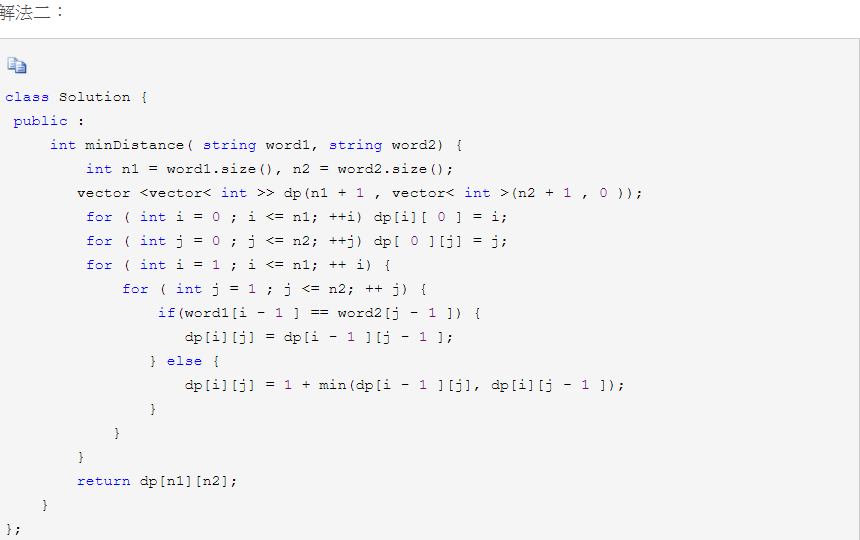
Example 1:

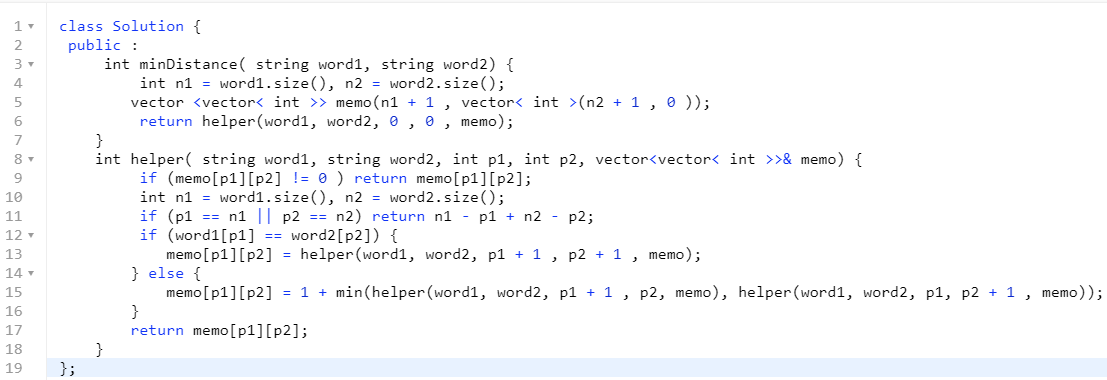
Input: 2  
Output: 1  
Explanation: 2 = 1 + 1, 1 × 1 = 1.

Example 2:

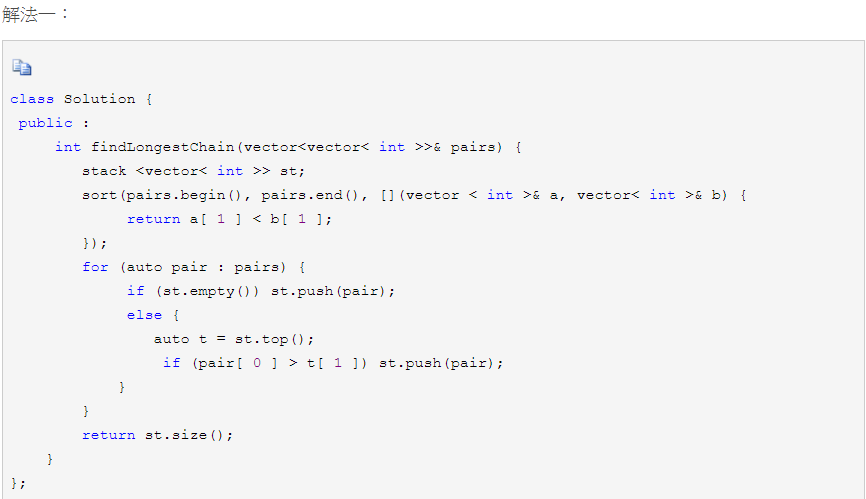
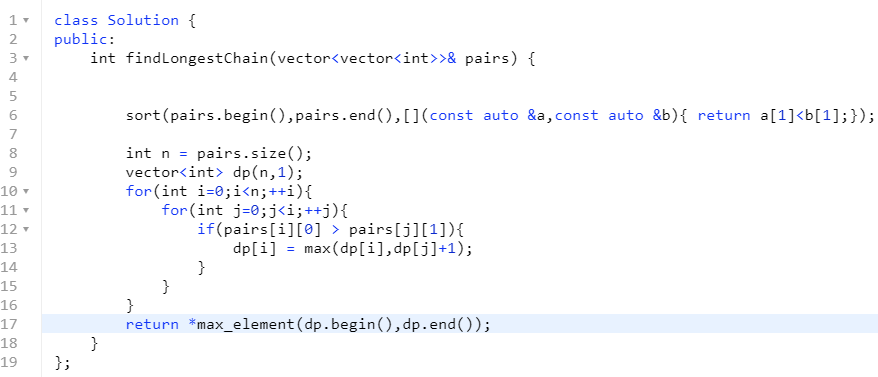
Input: 10  
Output: 36  
Explanation: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36.  
Note: You may assume that n is not less than 2 and not larger than 58.  
  
這道題給了我們一個正整數n，讓拆分成至少兩個正整數之和，使其乘積最大。最簡單粗暴的方法自然是檢查所有情況了，但是拆分方法那麼多，怎麼才能保證能拆分出所有的情況呢？感覺有點像之前那道 Coin Change，當前的拆分方法需要用到之前的拆分值，這種重現關係就很適合動態規劃Dynamic Programming來做，我們使用一個一維數組dp，其中dp[i ]表示數字i拆分為至少兩個正整數之和的最大乘積，數組大小為n+1，值均初始化為1，因為正整數的乘積不會小於1。可以從3開始遍歷，因為n是從2開始的，而2只能拆分為兩個1，乘積還是1。i從3遍歷到n，對於每個i，需要遍歷所有小於i的數字，因為這些都是潛在的拆分情況，對於任意小於i的數字j，首先計算拆分為兩個數字的乘積，即j乘以ij，然後是拆分為多個數字的情況，這裡就要用到dp[ij]了，這個值表示數字ij任意拆分可得到的最大乘積，再乘以j就是數字i可拆分得到的乘積，取二者的較大值來更新dp[i]，最後返回dp[n]即可，參見代碼如下：  
  
  
  
583. Delete Operation for Two Strings  
Given two words word1 and word2, find the minimum number of steps required to make word1 and word2 the same, where in each step you can delete one character in either string.

Example 1:  
Input: "sea", "eat"  
Output: 2  
Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".  
Note:  
The length of given words won't exceed 500.  
Characters in given words can only be lower-case letters.  
  
這道題給了我們兩個單詞，問我們最少需要多少步可以讓兩個單詞相等，每一步我們可以在任意一個單詞中刪掉一個字符。那麼我們分析怎麼能讓步數最少呢，是不是知道兩個單詞最長的相同子序列的長度，並乘以2，被兩個單詞的長度之和減，就是最少步數了。其實這道題就轉換成求Longest Common Subsequence最長相同子序列的問題，令博主意外的是，LeetCode中竟然沒有這道題，這與包含萬物的LeetCode的作風不符啊。不過沒事，有這道題也行啊，對於這種玩字符串，並且是求極值的問題，十有八九都是用dp來解的，曾經有網友問博主，如何確定什麼時候用greedy，什麼時候用dp？其實博主也不不太清楚，感覺dp要更tricky一些，而且出現的概率大，所以博主一般會先考慮dp，如果實在想不出遞推公式，那麼就想想greedy能做不。如果有大神知道更好的區分方法，請一定留言告知博主啊，多謝！那麼決定了用dp來做，就定義一個二維的dp數組，其中dp[i][j]表示word1的前i個字符和word2的前j個字符組成的兩個單詞的最長公共子序列的長度。下面來看遞推式dp[i][j]怎麼求，首先來考慮dp[i][j]和dp[i-1][j-1]之間的關係，我們可以發現，如果當前的兩個字符相等，那麼dp[i][j] = dp[i-1][j-1] + 1，這不難理解吧，因為最長相同子序列又多了一個相同的字符，所以長度加1。由於我們dp數組的大小定義的是(n1+1) x (n2+1)，所以我們比較的是word1[i-1]和word2[j-1]。那麼我們想如果這兩個字符不相等呢，難道我們直接將dp[i-1][j-1]賦值給dp[i][j]嗎，當然不是，我們還要錯位相比嘛，比如就拿題目中的例子來說，"sea"和"eat"，當我們比較第一個字符，發現's'和'e'不相等，下一步就要錯位比較啊，比較sea中第一個's '和eat中的'

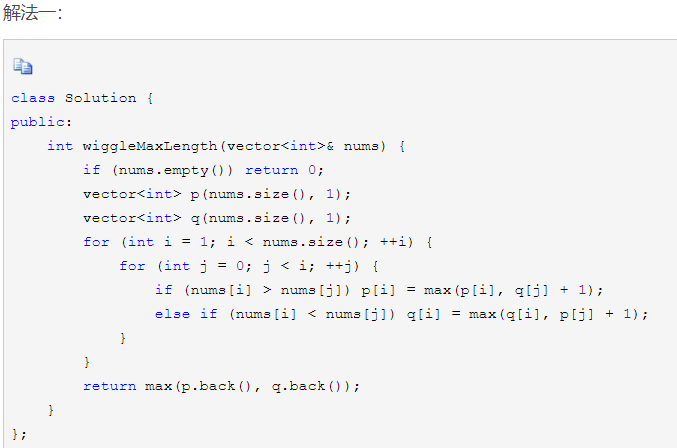
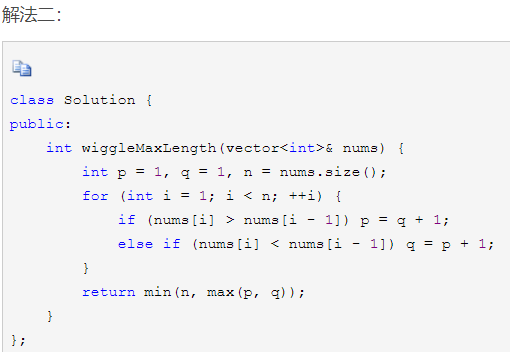
下面這種方法也是用的dp，但是和上面的dp思路不太一樣，這種算法是跟之前那道Edit Distance相同的思路。那道題問我們一個單詞通過多少步修改可以得到另一個單詞，其實word2刪除一個字符，和跟在word1對應的地方加上那個要刪除的字符，達到的效果是一樣的，並不影響最終的步驟數，所以這道題完全可以按照那道題的解法來做，一點都不需要變動，定義一個二維的dp數組，其中dp[i][j]表示word1的前i個字符和word2的前j個字符組成的兩個單詞，能使其變相同的最小的步數，講解可以參看那篇帖子，參見代碼入下：  


下面這種方法是解法二的遞歸寫法，用的優化的dfs的方法，用memo數組來保存中間計算結果，以避免大量的重複計算，參見代碼如下：  
  
  
  
646. Maximum Length of Pair Chain  
You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.  
Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.  
Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

Example 1:  
Input: [[1,2], [2,3], [3,4]]  
Output: 2  
Explanation: The longest chain is [1,2] -> [3,4]  
Note:  
The number of given pairs will be in the range [1, 1000].

這道題給了我們一些鏈對，規定瞭如果後面鏈對的首元素大於前鏈對的末元素，那麼這兩個鏈對就可以鏈起來，問我們最大能鏈多少個。那麼我們想，由於規定了鏈對的首元素一定小於尾元素，我們需要比較的是某個鍊錶的首元素和另一個鍊錶的尾元素之間的關係，如果整個鏈對數組是無序的，那麼就很麻煩，所以我們需要做的是首先對鏈對數組進行排序，按鏈對的尾元素進行排序，小的放前面。這樣我們就可以利用Greedy算法進行求解了。我們可以用一個棧，先將第一個鏈對壓入棧，然後對於後面遍歷到的每一個鏈對，我們看其首元素是否大於棧頂鏈對的尾元素，如果大於的話，就將當前鏈對壓入棧，這樣最後我們返回棧中元素的個數即可，參見代碼如下：  
  
解法二:先排序後，用DP來記錄結尾為第i個時候最長的鏈結長度  
  
  
376. Wiggle Subsequence  
  
A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.  
For example, [1,7,4,9,2,5] is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, [1,4,7,2,5] and [1,7,4,5,5] are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.  
Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

Examples:  
Input: [1,7,4,9,2,5]  
Output: 6  
The entire sequence is a wiggle sequence.  
Input: [1,17,5,10,13,15,10,5,16,8]  
Output: 7  
There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8].  
Input: [1,2,3,4,5,6,7,8,9]  
Output: 2  
Follow up:Can you do it in O(n) time?

这道题给我了我们一个数组，让我们求最长摆动子序列，关于摆动Wiggle数组，可以参见LC上之前的两道题Wiggle Sort和Wiggle Sort II。题目中给的tag说明了这道题可以用DP和Greedy两种方法来做，那么我们先来看DP的做法，我们维护两个dp数组p和q，其中p[i]表示到i位置时首差值为正的摆动子序列的最大长度，q[i]表示到i位置时首差值为负的摆动子序列的最大长度。我们从i=1开始遍历数组，然后对于每个遍历到的数字，再从开头位置遍历到这个数字，然后比较nums[i]和nums[j]，分别更新对应的位置，参见代码如下  
  
题目中有个Follow up说要在O(n)的时间内完成，而Greedy算法正好可以达到这个要求，这里我们不在维护两个dp数组，而是维护两个变量p和q，然后遍历数组，如果当前数字比前一个数字大，则p=q+1，如果比前一个数字小，则q=p+1，最后取p和q中的较大值跟n比较，取较小的那个，参见代码如下：  


494. Target Sum  
  
You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol.  
Find out how many ways to assign symbols to make sum of integers equal to target S.  
Example 1:

Input: nums is [1, 1, 1, 1, 1], S is 3.

Output: 5

Explanation:

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

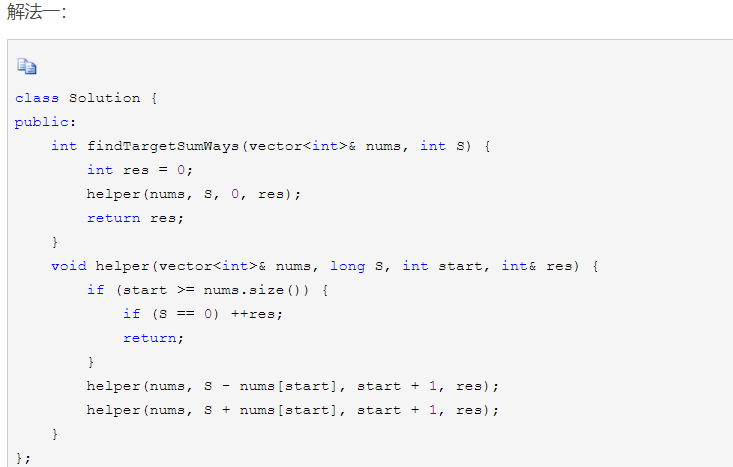
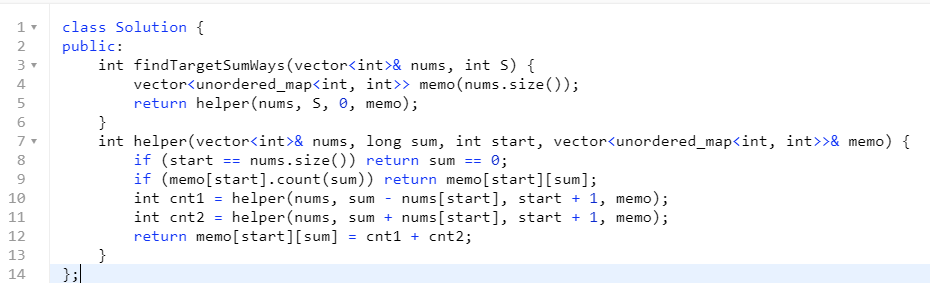
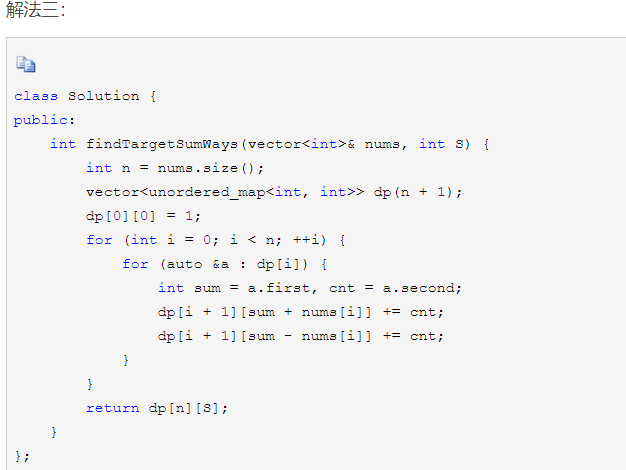
+1-1+1+1+1 = 3

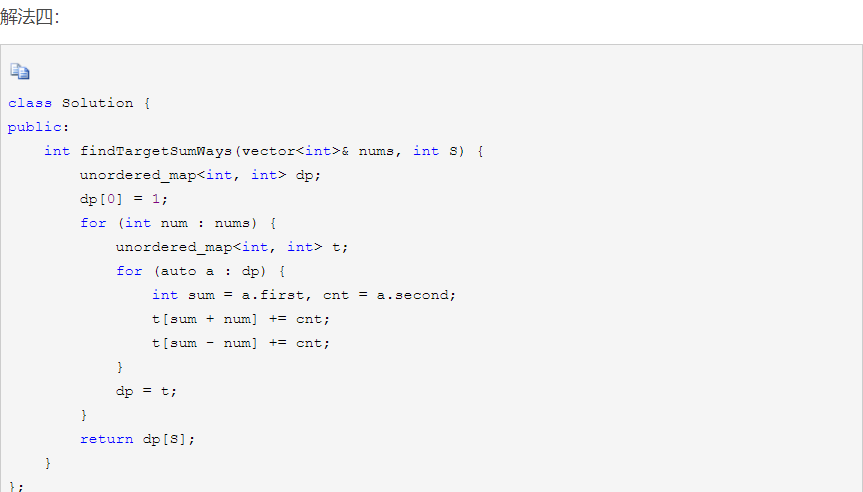
+1+1-1+1+1 = 3

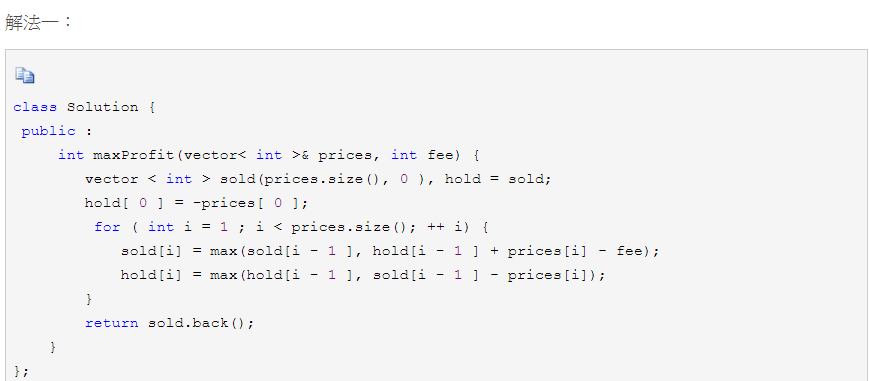
+1+1+1-1+1 = 3

+1+1+1+1-1 = 3  
There are 5 ways to assign symbols to make the sum of nums be target 3.  
Note:  
The length of the given array is positive and will not exceed 20.  
The sum of elements in the given array will not exceed 1000.  
Your output answer is guaranteed to be fitted in a 32-bit integer.

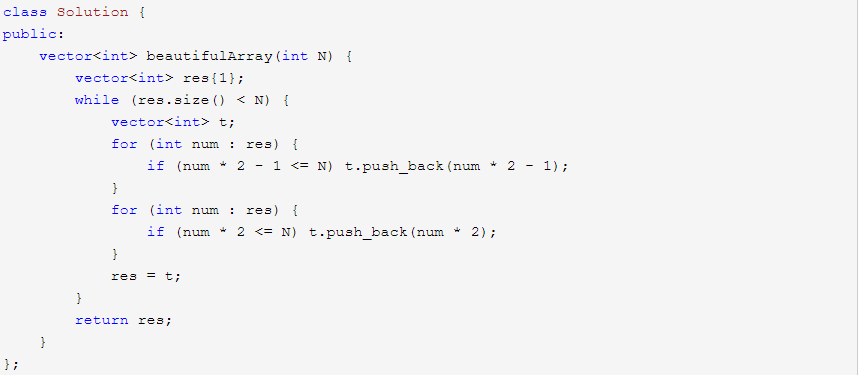
这道题给了我们一个数组，和一个目标值，让给数组中每个数字加上正号或负号，然后求和要和目标值相等，求有多少中不同的情况。那么对于这种求多种情况的问题，博主最想到的方法使用递归来做。从第一个数字，调用递归函数，在递归函数中，分别对目标值进行加上当前数字调用递归，和减去当前数字调用递归，这样会涵盖所有情况，并且当所有数字遍历完成后，若目标值为0了，则结果 res 自增1，参见代码如下：

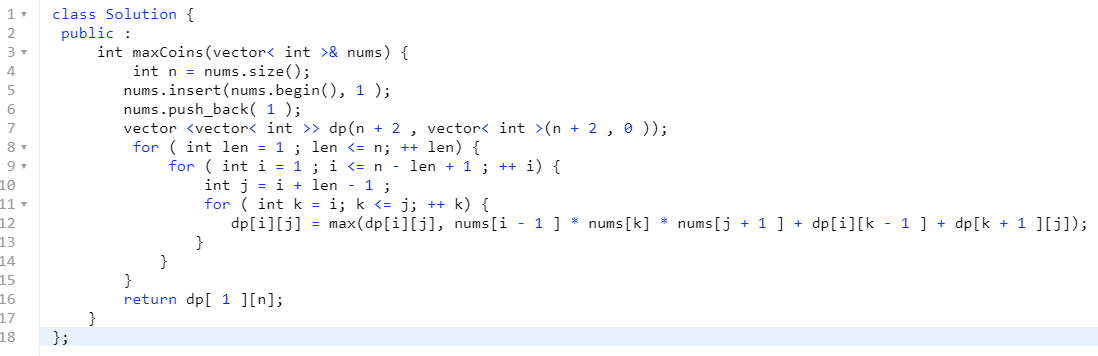
  
我们对上面的递归方法进行优化，使用 memo 数组来记录中间值，这样可以避免重复运算，参见代码如下：  
  
我们也可以使用迭代的方法来解，使用一个 dp 数组，其中 dp[i][j] 表示到第 i-1 个数字且和为j的情况总数，参见代码如下：  
  


我们也可以对上面的方法进行空间上的优化，只用一个 HashMap，而不是用一个数组的哈希表，在遍历数组中的每一个数字时，新建一个 HashMap，在遍历原 HashMap 中的项时更新这个新建的 HashMap，最后把新建的 HashMap 整个赋值为原 HashMap，参见代码如下：  
  
  
714. Best Time to Buy and Sell Stock with Transaction Fee  
You are given an array of integers prices, for which the i-th element is the price of a given stock on day i; and a non-negative integer fee representing a transaction fee.  
You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction. You may not buy more than 1 share of a stock at a time (ie. you must sell the stock share before you buy again. )  
Return the maximum profit you can make.  
Example 1:  
Input: prices = [1, 3, 2, 8, 4, 9], fee = 2  
Output: 8  
Explanation: The maximum profit can be achieved by:  
Buying at prices[0] = 1  
Selling at prices[3] = 8  
Buying at prices[4] = 4  
Selling at prices[5] = 9  
The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.  
Note:  
0 < prices.length <= 50000.  
0 < prices[i] < 50000.  
0 <= fee < 50000.

又是一道股票交易的題，之前已經有過類似的五道題了，fun4LeetCode大神的帖子做了amazing的歸納總結，有時間的話博主也寫個總結。這道題跟Best Time to Buy and Sell Stock II其實最像，但是由於那道題沒有交易費的限制，所以我們就無腦貪婪就可以了，見到利潤就往上加。但是這道題有了交易費，所以當賣出的利潤小於交易費的時候，我們就不應該賣了，不然虧了。所以這道題還是還是得用動態規劃來做，按照fun4LeetCode大神的理論，本質其實是個三維dp數組，由於第三維只有兩種情況，賣出和保留，而且第二維交易的次數在這道題中沒有限制，所以我們用兩個一維數組就可以了，sold[i]表示第i天賣掉股票此時的最大利潤，hold[i]表示第i天保留手裡的股票此時的最大利潤。那麼我們來分析遞推公式，在第i天，如果我們要賣掉手中的股票，那麼此時我們的總利潤應該是前一天手裡有股票的利潤(不然沒股票賣毛啊)，加上此時的賣出價格，減去交易費得到的利潤總值，跟前一天賣出的利潤相比，取其中較大值，如果前一天賣出的利潤較大，那麼我們就前一天賣了，不留到今天了。然後來看如果第i天不賣的利潤，就是昨天股票賣了的利潤然後今天再買入股票，得減去今天的價格，得到的值和昨天股票保留時的利潤相比，取其中的較大值，如果昨天保留股票的利潤大，那麼我們就繼續保留到今天，所以遞推時可以得到：  
sold[i] = max(sold[i - 1], hold[i - 1] + prices[i] - fee);  
hold[i] = max(hold[i - 1], sold[i - 1] - prices[i]);  
參見代碼如下：  
  


Chapter.8 Divide and Conquer

932. Beautiful Array  
  
For some fixed `N`, an array `A` is \*beautiful\* if it is a permutation of the integers `1, 2, ..., N`, such that:  
For every i < j, there is no k with i < k < j such that A[k] \* 2 = A[i] + A[j].  
Given N, return any beautiful array A. (It is guaranteed that one exists.)  
Example 1:  
Input: 4  
Output: [2,1,4,3]  
Example 2:  
Input: 5  
Output: [3,1,2,5,4]  
Note:  
1 <= N <= 1000  
  
這道題定義了一種漂亮數組，說的是在任意兩個數字之間，不存在一個正好是這兩個數之和的一半的數字，現在讓返回長度是N的一個漂亮數組，注意這裡長度是N的漂亮數組一定是由1到N之間的數字組成的，每個數字都會出現，而且一定存在這樣的漂亮數組。博主剛開始時是沒什麼頭緒的，想著總不會是要遍歷所有的排列情況，然後對每個情況去驗證是否是漂亮數組的吧，想想都覺得很不高效，於是就放棄掙扎，直接逛論壇了。不出意外，最高票的還是你李哥，居然提出了逆天的線性時間的解法，獻上膝蓋，怪不得有網友直接要Venmo 號立馬打錢，LOL～ 這道題給了提示說是要用分治法來做，但是怎麼分是這道題的精髓，若只是普通的對半分，那麼在merge 的時候還是要驗證是否是漂亮數組，麻煩！但若按奇偶來分的話，那就非常的叼了，因為奇數加偶數等於奇數，就不會是任何一個數字的2倍了。這就是奇偶分堆的好處，這時任意兩個數字肯定不能分別從奇偶堆裡取了，那可能你會問，奇數堆會不會有三個奇數打破這個規則呢？只要這個奇數堆是從一個漂亮數組按固定的規則變化而來的，就能保證一定也是漂亮數組，因為對於任意一個漂亮數組，若對每個數字都加上一個相同的數字，或者都乘上一個相同的數字，則一定還是漂亮數組，因為數字的之間的內在關係並沒有改變。明白了上面這些，基本就可以解題了，假設此時已經有了一個長度為n的漂亮數組，如何將其擴大呢？可以將其中每個數字都乘以2並加1，就都會變成奇數，並且這個奇數數組還是漂亮的，然後再將每個數字都乘以2，那麼都會變成偶數，並且這個偶數數組還是漂亮的，兩個數組拼接起來，就會得到一個長度為2n 的漂亮數組。就是這種思路，可以從1開始，1本身就是一個漂亮數組，然後將其擴大，注意這裡要卡一個N，不能讓擴大的數組長度超過N，只要在變為奇數和偶數時加個判定就行了，將不大於N的數組加入到新的數組中，參見代碼如下：  
  
  
312. Burst Balloons  
Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon [i] you will get nums[left] \* nums[i] \* nums[right]coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.  
Find the maximum coins you can collect by bursting the balloons wisely.  
Note:  
You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.  
0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100  
Example:  
Input:   
Output: nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> [][3,1,5,8]167   
Explanation:   
 coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167  
  
  
  
  
這道題提出了一種打氣球的遊戲，每個氣球都對應著一個數字，每次打爆一個氣球，得到的金幣數是被打爆的氣球的數字和其兩邊的氣球上的數字相乘，如果旁邊沒有氣球了，則按1算，以此類推，求能得到的最多金幣數。參見題目中給的例子，題意並不難理解。那麼大家拿到題後，總是會習慣的先去想一下暴力破解法吧，這道題的暴力搜索將相當的複雜，因為每打爆一個氣球，斷開的地方又重新挨上，所有剩下的氣球又要重新遍歷，這使得分治法不能work，整個的時間複雜度會相當的高，不要指望可以通過OJ。而對於像這種求極值問題，一般都要考慮用動態規劃Dynamic Programming 來做，維護一個二維動態數組dp，其中dp[i][j] 表示打爆區間[i, j] 中的所有氣球能得到的最多金幣。題目中說明了邊界情況，當氣球周圍沒有氣球的時候，旁邊的數字按1算，這樣可以在原數組兩邊各填充一個1，方便於計算。這道題的最難點就是找狀態轉移方程，還是從定義式來看，假如區間只有一個數，比如dp[i][i]，那麼計算起來就很簡單，直接乘以周圍兩個數字即可更新。如果區間裡有兩個數字，就要算兩次了，先打破第一個再打破了第二個，或者先打破第二個再打破第一個，比較兩種情況，其中較大值就是該區間的dp 值。假如區間有三個數呢，比如dp[1][3]，怎麼更新呢？如果先打破第一個，剩下兩個怎麼辦呢，難道還要分別再遍曆算一下嗎？這樣跟暴力搜索的方法有啥區別呢，還要dp 數組有啥意思。所謂的狀態轉移，就是假設已知了其他狀態，來推導現在的狀態，現在是想知道dp[1][3] 的值，那麼如果先打破了氣球1，剩下了氣球2和3，若之前已經計算了dp[2][3] 的話，就可以使用其來更新dp[1][3] 了，就是打破氣球1的得分加上dp[2][3]。那假如先打破氣球2呢，只要之前計算了dp[1][1] 和dp[3][3]，那麼三者加起來就可以更新dp[1][3]。同理，先打破氣球3，就用其得分加上dp[1][2] 來更新dp[1][3]。說到這裡，是不是感覺豁然開朗了^.^  
那麼對於有很多數的區間[i, j]，如何來更新呢？現在是想知道dp[i][j] 的值，這個區間可能比較大，但是如果知道了所有的小區間的dp 值，然後聚沙成塔，逐步的就能推出大區間的dp 值了。還是要遍歷這個區間內的每個氣球，就用k來遍歷吧，k在區間[i, j] 中，假如第k個氣球最後被打爆，那麼此時區間[i, j] 被分成了三部分，[i, k-1]，[k]，和[k+1, j]，只要之前更新過了[i, k-1] 和[k+1, j] 這兩個子區間的dp值，可以直接用dp[i][k-1] 和dp[k+1][j]，那麼最後被打爆的第k個氣球的得分該怎麼算呢，你可能會下意識的說，就乘以周圍兩個氣球被nums[k-1] \* nums[k] \* nums[k+1]，但其實這樣是錯誤的，為啥呢？dp[i][k-1] 的意義是什麼呢，是打爆區間[i, k-1] 內所有的氣球後的最大得分，此時第k-1 個氣球已經不能用了，同理，第k+1 個氣球也不能用了，相當於區間[i, j] 中除了第k個氣球，其他的已經爆了，那麼周圍的氣球只能是第i-1 個，和第j+ 1 個了，所以得分應為nums[i-1] \* nums[k] \* nums[j+1]，分析到這裡，狀態轉移方程應該已經躍然紙上了吧，如下所示：  
dp[i][j] = max(dp[i][j], nums[i - 1] \* nums[k] \* nums[j + 1] + dp[i][k - 1] + dp[k + 1][j]) ( i ≤ k ≤ j )  
有了狀態轉移方程了，就可以寫代碼，下面就遇到本題的第二大難點了，區間的遍歷順序。一般來說，遍歷所有子區間的順序都是i從0到n，然後j從i到n，然後得到的[i, j] 就是子區間。但是這道題用這種遍歷順序就不對，在前面的分析中已經說了，這裡需要先更新完所有的小區間，然後才能去更新大區間，而用這種一般的遍歷子區間的順序，會在更新完所有小區間之前就更新了大區間，從而不一定能算出正確的dp值，比如拿題目中的那個例子[3, 1, 5, 8] 來說，一般的遍歷順序是：  
[3] -> [3, 1] -> [3, 1, 5] -> [3, 1, 5, 8] -> [1] -> [1, 5] -> [1, 5, 8 ] -> [5] -> [5, 8] -> [8]   
顯然不是我們需要的遍歷順序，正確的順序應該是先遍歷完所有長度為1的區間，再是長度為2的區間，再依次累加長度，直到最後才遍歷整個區間：  
[3] -> [1] -> [5] -> [8] -> [3, 1] -> [1, 5] -> [5, 8] -> [3, 1, 5] -> [1, 5, 8] -> [3, 1, 5, 8]  
這裡其實只是更新了dp 數組的右上三角區域，最終要返回的值存在dp[1][n] 中，其中n是兩端添加1之前數組nums 的個數。參見代碼如下：

  
對於題目中的例子[3, 1, 5, 8]，得到的dp數組如下：

0 0 0 0 0 0

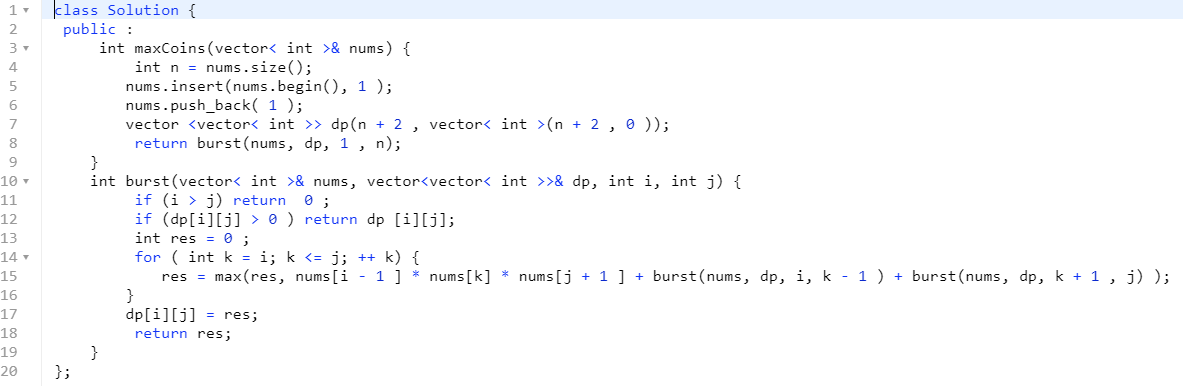
0 3 30 159 167 0

0 0 15 135 159 0

0 0 0 40 48 0

0 0 0 0 40 0

0 0 0 0 0 0  
  
這題還有遞歸解法，思路都一樣，就是寫法略有不同，參見代碼如下：



Chapter.9 Math Question

168. Excel Sheet Column Title  
  
Given a positive integer, return its corresponding column title as appear in an Excel sheet.  
For example:

1 -> A

2 -> B

3 -> C

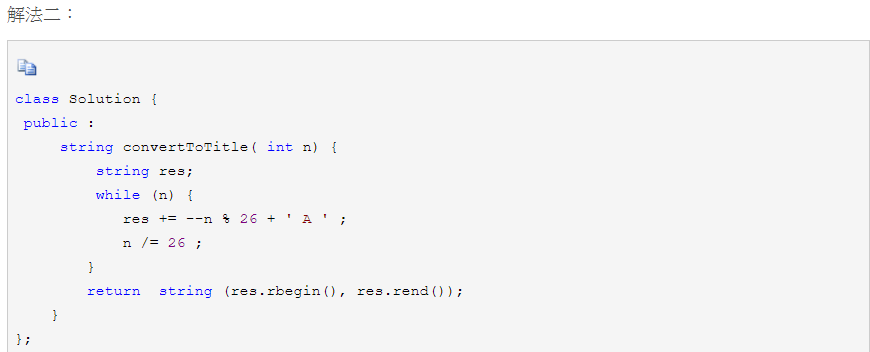
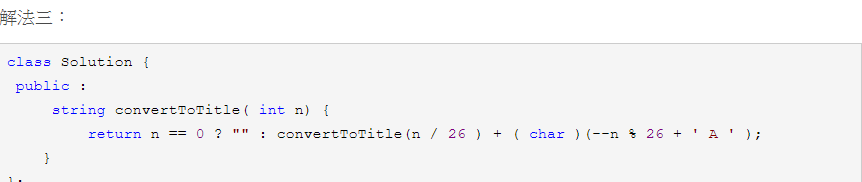
...

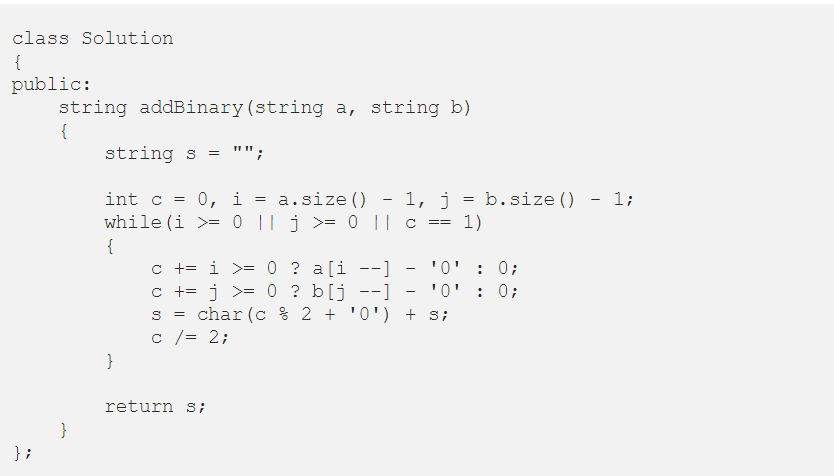
26 -> Z

27 -> AA

28 -> AB

…  
Example 1: Input: 1 ,Output: "A"  
Example 2:Input: 28, Output: "AB"  
Example 3:Input: 701,Output: "ZY"

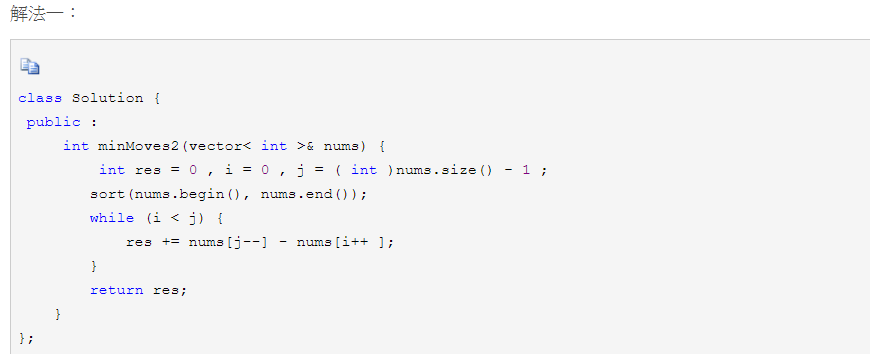
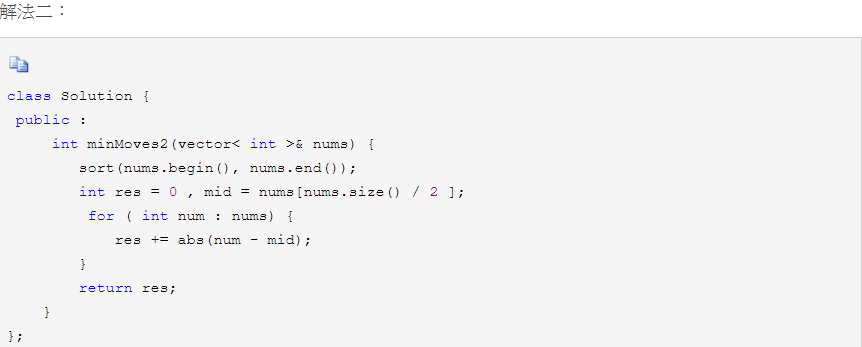
此題和 Excel Sheet Column Number 是一起的，但是我在這題上花的時間遠比上面一道多，其實原理都一樣，就是一位一位的求，此題從低位往高位求，每進一位，則把原數縮小26倍，再對26取餘，之後減去餘數，再縮小26倍，以此類推，可以求出各個位置上的字母。最後只需將整個字符串翻轉一下即可。我們先從最簡單的來看，對於小於26的數字，那麼我們只需要對26取餘，然後減去1，加上字符A即可，但是對於26來說，如果還是這麼做的話就會出現問題，因為對26取餘是0，減去1後成為-1，加上字符A後，並不等於字符Z。所以對於能被26整除的數我們得分開處理，所以就分情況討論一下吧，能整除26的，直接在結果res上加上字符Z，然後n自減去26；不能的話，就按照一般的處理，n要減去這個餘數。之後n要自除以26，繼續計算下去，代碼如下：  
  
然後我們可以對上面對方法進行下優化，合併if和else，寫的更簡潔一些。從上面的講解中我們得知，會造成這種不便的原因是能被26整除的數字，無法得到字符Z。那麼我們用一個小trick，比如對於26來說，我們先讓n自減1，變成25，然後再對26取餘，得到25，此時再加上字符A，就可以得到字符Z了。叼就叼在這對其他的不能整除26的數也是成立的，完美解決問題，參見代碼如下：  
  
這道題還可以用遞歸來解，而且可以喪心病狂的壓縮到一行代碼來解：  


67. Add Binary  
Given two binary strings, return their sum (also a binary string).  
The input strings are both non-empty and contains only characters 1 or 0.  
Example 1: Input: a = "11", b = "1",Output: "100"  
Example 2: Input: a = "1010", b = "1011",Output: "10101"  
  
雖然是 easy，並且看起來也很 easy，但是很容易出小錯誤，而且能用的方式很多，不好決定要用那種。  
解這題有三個重點:  
(1) ‘0’ 和 ‘1’ 轉成數字: 雖然也可以不轉，但是計算時候加上進位的情形很多， 所以還是轉成數字比較方便，可以簡單用這個式子: a[i] - '0'  
(2) 加法是從後面加到前面，所以計算出的結果可能需要 reverse。  
(3) 考慮到進位，最重要的是兩個字串相加完之後，可能還會有進位! 我就是遺漏了這個最前頭的進位。  
這裡直接貼網路上的解法，很乾淨漂亮，直接把進位考慮在終止條件裡面。  
並且進位也直接當作該次運算的加總，最後在除以 2 變成下一次的進位。  


238. Product of Array Except Self  
  
Given an array nums of n integers where n > 1, return an array output such that output[i] is equal to the product of all the elements of numsexcept nums[i].  
Example:  
Input:   
Output:[1,2,3,4][24,12,8,6]  
Note: Please solve it without division and in O( n ).  
Follow up:  
Could you solve it with constant space complexity? (The output array does not count as extra space for the purpose of space complexity analysis.)  
  
這道題給定我們一個數組，讓我們返回一個新數組，對於每一個位置上的數是其他位置上數的乘積，並且限定了時間複雜度O(n)，並且不讓我們用除法。如果讓用除法的話，那這道題就應該屬於Easy，因為可以先遍歷一遍數組求出所有數字之積，然後除以對應位置的上的數字。但是這道題禁止我們使用除法，那麼我們只能另闢蹊徑。我們想，對於某一個數字，如果我們知道其前面所有數字的乘積，同時也知道後面所有的數乘積，那麼二者相乘就是我們要的結果，所以我們只要分別創建出這兩個數組即可，分別從數組的兩個方向遍歷就可以分別創建出乘積累積數組。參見代碼如下  

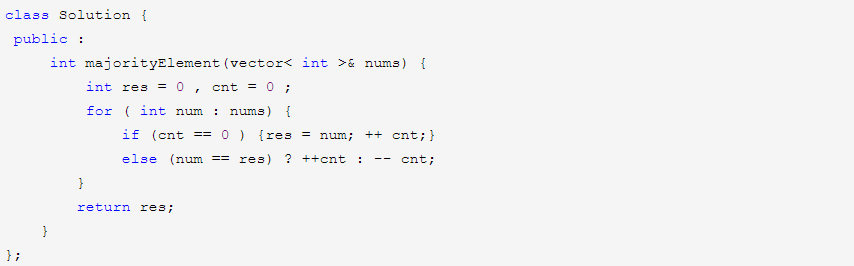
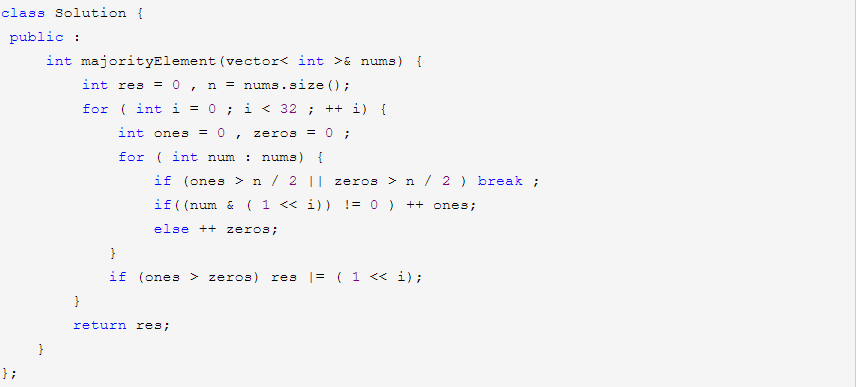

462. Minimum Moves to Equal Array Elements II

Given a non-empty integer array, find the minimum number of moves required to make all array elements equal, where a move is incrementing a selected element by 1 or decrementing a selected element by 1.  
You may assume the array's length is at most 10,000.

Example:  
Input: [1,2,3]  
Output:2  
Explanation:  
Only two moves are needed (remember each move increments or decrements one element):  
[1,2,3] => [2,2,3] => [2,2,2]  
  
這道題是之前那道Minimum Moves to Equal Array Elements的拓展，現在我們可以每次對任意一個數字加1或者減1，讓我們用最少的次數讓數組所有值相等。一般來說這種題目是不能用暴力方法算出所有情況，因為OJ一般是不會答應的。那麼這道題是否像上面一道題一樣，有巧妙的方法呢？答案是肯定的。下面這種解法實際上利用了之前一道題Best Meeting Point的思想，是不感覺很amazing，看似完全不相干的兩道題，居然有著某種內部聯繫。我們首先給數組排序，那麼我們最終需要變成的相等的數字就是中間的數，如果數組有奇數個，那麼就是最中間的那個數字；如果是偶數個，那麼就是中間兩個數的區間中的任意一個數字。而兩端的數字變成中間的一個數字需要的步數實際上就是兩端數字的距離，講到這裡發現是不是就和這道題Best Meeting Point的思路是一樣了。那麼我們就兩對兩對的累加它們的差值就可以了，參見代碼如下：  
  
既然有了上面的分析，我們知道實際上最後相等的數字就是數組的最中間的那個數字，那麼我們在給數組排序後，直接利用坐標定位到中間的數字，然後算數組中每個數組與其的差的絕對值累加即可，參見代碼如下：  
  


169. Majority Element  
  
Given an array of size n , find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times.  
You may assume that the array is non-empty and the majority element always exist in the array.  
Example 1:  
Input: [3,2,3]  
Output: 3  
Example 2:  
Input: [2,2,1,1,1,2,2]  
Output: 2

這是到求大多數的問題，有很多種解法，其中我感覺比較好的有兩種，一種是用哈希表，這種方法需要O(n) 的時間和空間，另一種是用一種叫摩爾投票法Moore Voting，需要O(n) 的時間和O(1) 的空間，比前一種方法更好。這種投票法先將第一個數字假設為過半數，然後把計數器設為1，比較下一個數和此數是否相等，若相等則計數器加一，反之減一。然後看此時計數器的值，若為零，則將下一個值設為候選過半數。以此類推直到遍歷完整個數組，當前候選過半數即為該數組的過半數。不仔細弄懂摩爾投票法的精髓的話，過一陣子還是會忘記的，首先要明確的是這個叼炸天的方法是有前提的，就是數組中一定要有過半數的存在才能使用，下面來看本算法的思路，這是一種先假設候選者，然後再進行驗證的算法。現將數組中的第一個數假設為過半數，然後進行統計其出現的次數，如果遇到同樣的數，則計數器自增1，否則計數器自減1，如果計數器減到了0，則更換下一個數字為候選者。這是一個很巧妙的設定，也是本算法的精髓所在，為啥遇到不同的要計數器減1呢，為啥減到0了又要更換候選者呢？首先是有那個強大的前提存在，一定會有一個出現超過半數的數字存在，那麼如果計數器減到0了話，說明目前不是候選者數字的個數已經跟候選者的出現個數相同了，那麼這個候選者已經很weak，不一定能出現超過半數，此時選擇更換當前的候選者。那有可能你會有疑問，那萬一後面又大量的出現了之前的候選者怎麼辦，不需要擔心，如果之前的候選者在後面大量出現的話，其又會重新變為候選者，直到最終驗證成為正確的過半數，佩服算法的提出者啊，代碼如下：

  
下面這種解法利用到了位操作Bit Manipulation 來解，將這個大多數按位來建立，從0到31位，每次統計下數組中該位上0和1的個數，如果1多，那麼將結果res 中該位變為1，最後累加出來的res 就是過半數了，相當讚的方法，參見代碼如下：  


470. Implement Rand10() Using Rand7()  
  
Given a function rand7 which generates a uniform random integer in the range 1 to 7, write a function rand10 which generates a uniform random integer in the range 1 to 10.  
Do NOT use system's Math.random().

Example 1: Input: 1,Output: [7]  
Example 2:Input: 2,Output: [8,4]  
Example 3:Input: 3,Output: [8,1,10]

Note:  
rand7 is predefined.  
Each testcase has one argument: n, the number of times that rand10 is called.  
Follow up:  
What is the expected value for the number of calls to rand7() function?  
Could you minimize the number of calls to rand7()?

這道題給了我們一個隨機生成[1, 7] 內數字的函數rand7()，需要利用其來生成一個能隨機生成[1, 10] 內數字的函數rand10()，注意這裡的隨機生成的意思是等概率生成範圍內的數字。這是一道很有意思的題目，由於rand7() 只能生成1到7之間的數字，所以8，9，10 這三個沒法生成，那麼怎麼辦？大多數人可能第一個想法就是，再用一個唄，然後把兩次的結果加起來，範圍不就擴大了麼，擴大成了[2, 14] 之間，然後如果再減去1，範圍不就是[1, 13] 了麼。想法不錯，但是有個問題，這個範圍內的每個數字生成的概率不是都相等的，為啥這麼說呢，我們來舉個簡單的例子看下，就比如說rand2()，我們知道其可以生成兩個數字1和2，且每個的概率都是1/2。那麼對於(rand2() - 1) + rand2()呢，看一下：  
rand2() - 1 + rand() 2 = ?  
1 1 1   
1 2 2   
2 1 2   
2 2 3

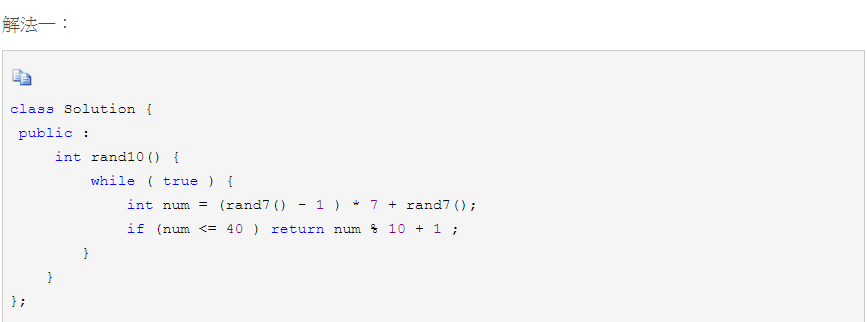
我們發現，生成數字範圍[1, 3] 之間的數字並不是等概率大，其中2出現的概率為1/2，1和3分別為1/4。這就不隨機了。問題出在哪裡了呢，如果直接相加，不同組合可能會產生相同的數字，比如1+2 和2+1 都是3。所以需要給第一個rand2() 升一個維度，讓其乘上一個數字，再相加。比如對於(rand2() - 1) \* 2 + rand2()，如下：  
（rand2() - 1） \* 2 + rand() 2 = ?  
1 1 1   
1 2 2   
2 1 3   
2 2 4

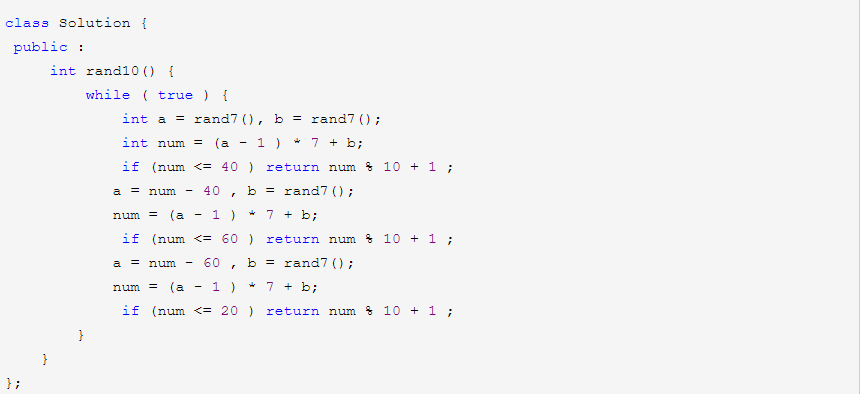
這時右邊生成的1，2，3，4 就是等概率出現的了。這樣就通過使用rand2()，來生成rand4()了。那麼反過來想一下，可以通過rand4() 來生成rand2()，其實更加簡單，我們只需通過rand4() % 2 + 1 即可，如下：

rand4() % 2 + 1 = ?  
1 2   
2 1   
3 2  
4 1

同理，我們也可以通過rand6() 來生成rand2()，我們只需通過rand6() % 2 + 1 即可，如下  
rand6() % 2 + 1 = ?  
1 2   
2 1   
3 2   
4 1   
5 2   
6 1

所以，回到這道題，我們可以先湊出rand10\*N()，然後再通過rand10\*N() % 10 + 1來獲得rand10()。那麼，只需要將rand7()轉化為rand10\*N()即可，根據前面的講解，我們轉化也必須要保持等概率，那麼就可以變化為(rand7() - 1) \* 7 + rand7() ，就轉為了rand49()。但是49不是10的倍數，不過49包括好幾個10的倍數，比如40，30，20，10等。這裡，我們需要把rand49()轉為rand40()，需要用到 拒絕採樣Rejection Sampling，總感覺名字很奇怪，之前都沒有聽說過這個採樣方法，刷題也是個不停學習新東西的過程呢。簡單來說，這種採樣方法就是隨機到需要的數字就接受，不是需要的就拒絕，並重新採樣，這樣還能保持等概率，具體的證明這裡就不講解了，博主也不會，有興趣的童鞋們可以去Google一下～這裡直接用結論就好啦，當用rand49()生成一個[1, 49]範圍內的隨機數，如果其在[1, 40]範圍內，我們就將其轉為rand10()範圍內的數字，直接對10去餘並加1，返回即可。如果不是，則繼續循環即可，參見代碼如下：



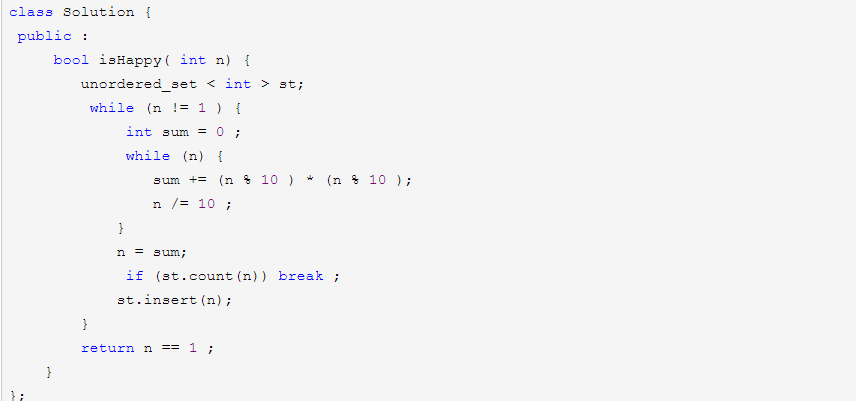
我們還可以對上面的解法進行一下優化，因為說實話在[1, 49] 的範圍內隨機到[41, 49] 內的數字概率還是挺高的，我們可以做進一步的處理，就是當循環到這九個數字的時候，我們不重新採樣，而是做進一步的處理，將採樣到的數字減去40，這樣就相當於有了個rand9()，那麼通過(rand9() - 1) \* 7 + rand7()，可以變成rand63()，對rand63() 進行拒絕採樣，得到rand60()，從而又可以得到rand10()了，此時還會多餘出3個數字，[61, 63]，通過減去60，得到rand3()，再通過變換(rand3() - 1) \* 7 + rand7() 得到rand21()，此時再次調用拒絕採樣，得到rand20()，進而得到rand10()，此時就只多餘出一個21，重複整個循環的概率就變的很小了，參見代碼如下：  


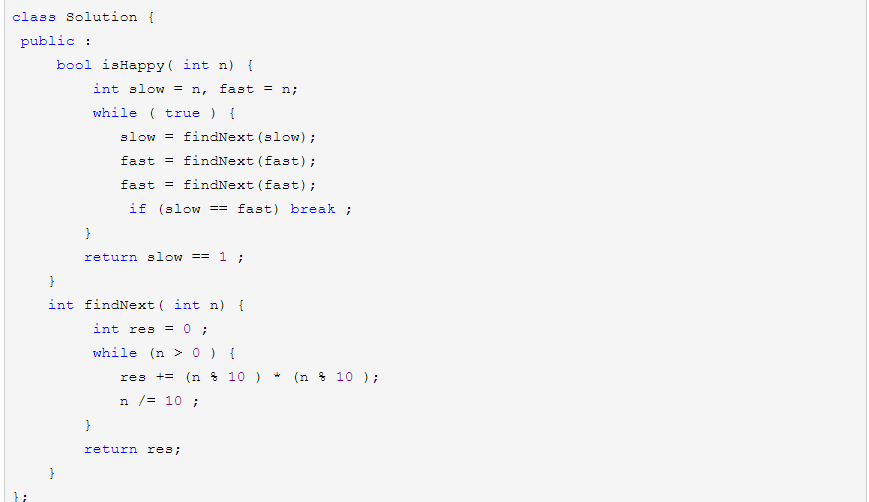
202. Happy Number

Write an algorithm to determine if a number is "happy".  
A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.  
Example:   
Input: 19  
Output: true

Explanation:   
1^2 + 9^2 = 82  
8^2 + 2^2 = 68  
6^2 + 8^2 = 100  
1^2 + 0^2 + 0^2 = 1

這道題定義了一種快樂數，就是說對於某一個正整數，如果對其各個位上的數字分別平方，然後再加起來得到一個新的數字，再進行同樣的操作，如果最終結果變成了1，則說明是快樂數，如果一直循環但不是1的話，就不是快樂數，那麼現在任意給我們一個正整數，讓我們判斷這個數是不是快樂數，題目中給的例子19是快樂數，那麼我們來看一個不是快樂數的情況，比如數字11有如下的計算過程：  
1^2 + 1^2 = 2  
2^2 = 4  
4^2 = 16  
1^2 + 6^2 = 37  
3^2 + 7^2 = 58  
5^2 + 8^2 = 89  
8^2 + 9^2 = 145  
1^2 + 4^2 + 5^2 = 42  
4^2 + 2^2 = 20  
2^2 + 0^2 = 4  
我們發現在算到最後時數字4又出現了，那麼之後的數字又都會重複之前的順序，這個循環中不包含1，那麼數字11不是一個快樂數，發現了規律後就要考慮怎麼用代碼來實現，我們可以用HashSet 來記錄所有出現過的數字，然後每出現一個新數字，在HashSet 中查找看是否存在，若不存在則加入表中，若存在則跳出循環，並且判斷此數是否為1 ，若為1返回true，不為1返回false，代碼如下：



這道題還有一種快慢指針的解法，由熱心網友喵團團提供，跟之前那道 Linked List Cycle 檢測環的方法類似，不同的是這道題環一定存在，不過有的環不符合題意，只有最後slow停在了1的位置，才表明是一個快樂數。而且這裡每次慢指針走一步，快指針走兩步，不是簡單的指向next，而是要調用子函數計算各位上數字的平方和，當快慢指針相等時，跳出循環，並且判斷慢指針是否為1即可，參見代碼如下：  


**Chapter 10 Bit Operation**

**268. Missing Number (Easy)**

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

Example 1:

Input: [3,0,1]

Output: 2

Example 2:

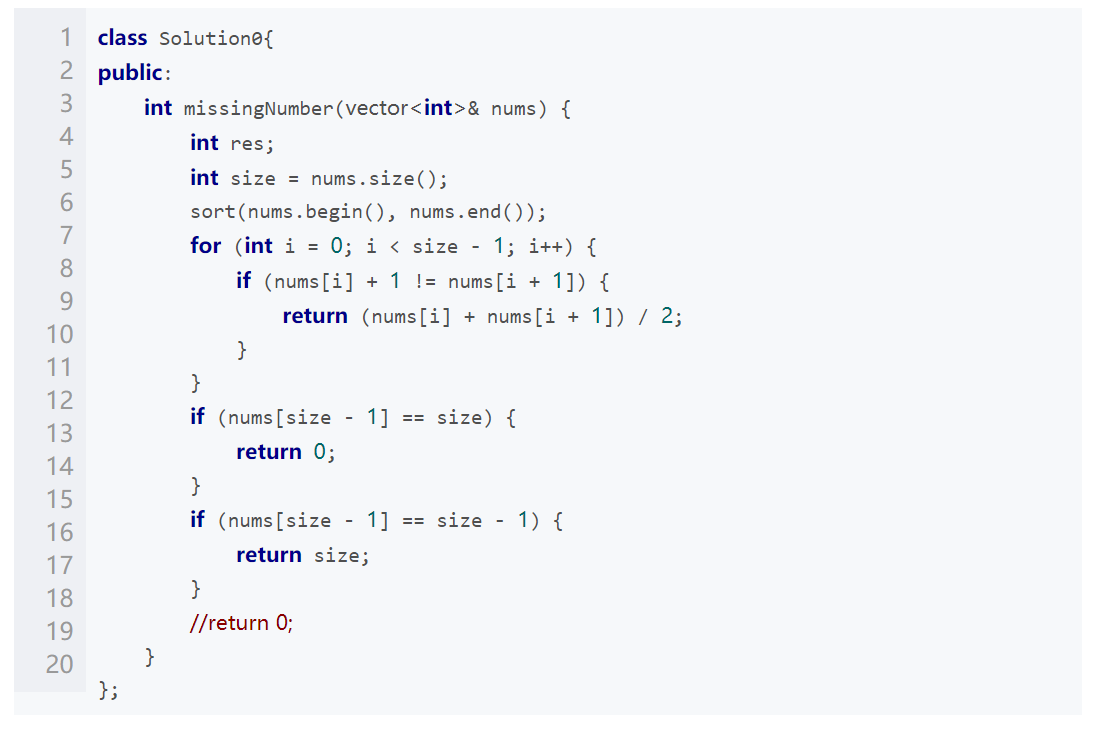
Input: [9,6,4,2,3,5,7,0,1]

Output: 8

## **解题思路**

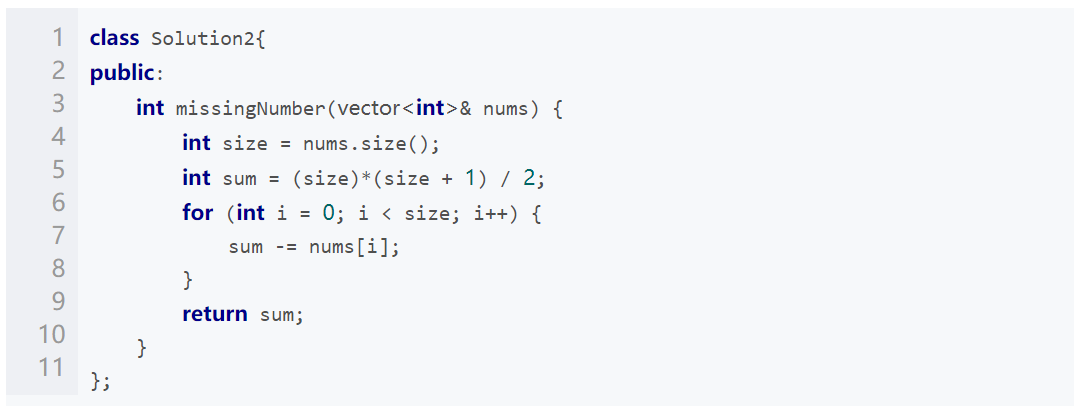
方法1:

* 暴力解法
* 先排序,再遍历找缺少的值. 遍历时候有三种情况:
* 前后不相等. 则缺少中间这个数.
* 遍历结束
* arr[n-1]=n;,明,缺少0; arr[n-1] =n-1,说明缺少n



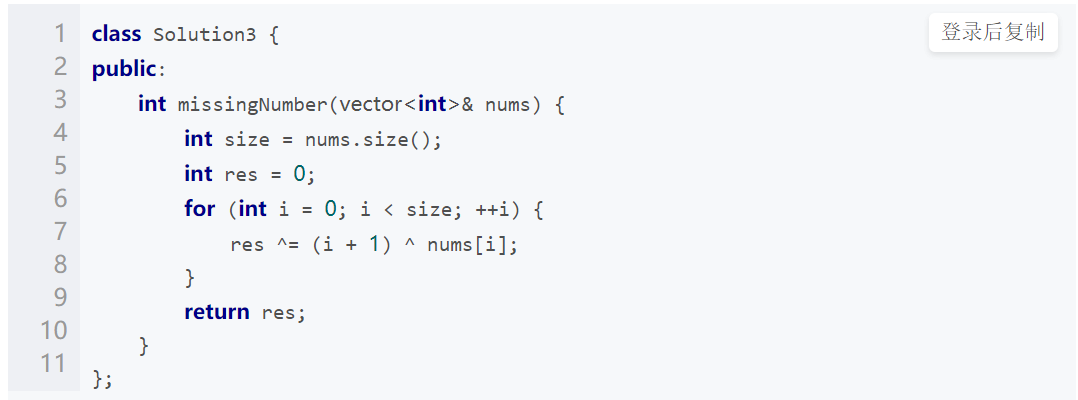
方法2:

* [0.1..n]是一个等差数列. 可以直接对其求和.
* 再遍历数组,逐一减去数组中元素.剩下的即为丢失的值.



方法3:

* 异或
* 将这个数组和 完整的[0..n]的数组进行异或.
* 余下的值则为 missing number.
* 注意: 1. 0 ^ A =A. 2. [1..n]直接有索引产生.



**693. Binary Number with Alternating Bits (Easy)**

Given a positive integer, check whether it has alternating bits: namely, if two adjacent bits will always have different values.

Example 1:

Input: 5

Output: True

Explanation:

The binary representation of 5 is: 101

Example 2:

Input: 7

Output: False

Explanation:

The binary representation of 7 is: 111.

Example 3:

Input: 11

Output: False

Explanation:

The binary representation of 11 is: 1011.

Example 4:

Input: 10

Output: True

Explanation:

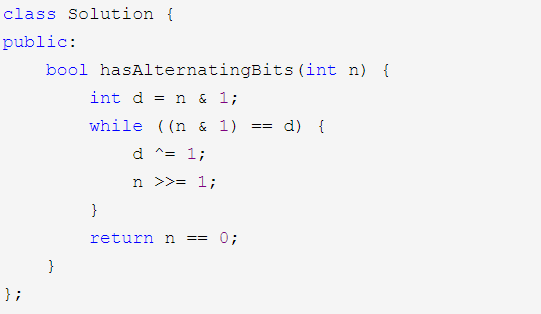
The binary representation of 10 is: 1010.

这道题让我们判断一个二进制数的1和0是否是交替出现的，博主开始也美想到啥简便方法，就一位一位来检测呗，用个变量bit来记录上一个位置的值，初始化为-1，然后我们用‘与’1的方法来获取最低位的值，如果是1，那么当此时bit已经是1的话，说明两个1相邻了，返回false，否则bit赋值为1。同理，如果是0，那么当此时bit已经是0的话，说明两个0相邻了，返回false，否则bit赋值为0。判断完别忘了将n向右移动一位。如果while循环退出了，返回true，参见代码如下



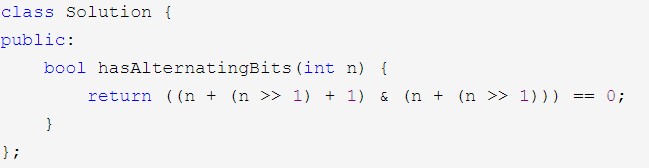
下面这种解法写的更加简洁了，我们不需要用if条件来判断，而是可以通过‘亦或’1的方式来将0和1互换，当然我们也可以通过d = 1 - d 来达到同样的效果，但还是写成‘亦或’1比较叼，while循环的条件是最低位等于d，而d不停的在0和1之间切换，n每次也向右平移一位，这样能交替检测0和1，循环退出后，如果n为0，则返回true，反之则返回false，参见代码如下：

解法二：



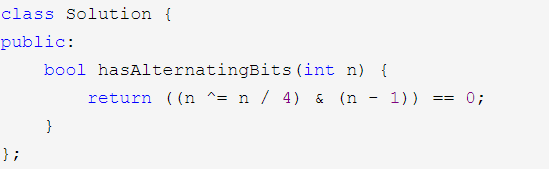
下面这种解法就十分的巧妙了，利用了0和1的交替的特性，进行错位相加，从而组成全1的二进制数，然后再用一个检测全1的二进制数的trick，就是‘与’上加1后的数，因为全1的二进制数加1，就会进一位，并且除了最高位，其余位都是0，跟原数相‘与’就会得0，所以我们可以这样判断。比如n是10101，那么n>>1就是1010，二者相加就是11111，再加1就是100000，二者相‘与’就是0，参见代码如下：

解法三：



下面这种解法也很巧妙，先将n右移两位，再和原来的n亦或，得到的新n其实就是除了最高位，其余都是0的数，然后再和自身减1的数相‘与’，如果是0就返回true，反之false。比如n是10101，那么n/4是101，二者相‘亦或’，得到10000，此时再减1，为1111，二者相‘与’得0，参见代码如下：

解法四：



**476. Number Complement (Easy)**

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

Note:

1，The given integer is guaranteed to fit within the range of a 32-bit signed integer.

2，You could assume no leading zero bit in the integer’s binary representation.

Example 1:

Input: 5

Output: 2

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

Example 2:

Input: 1

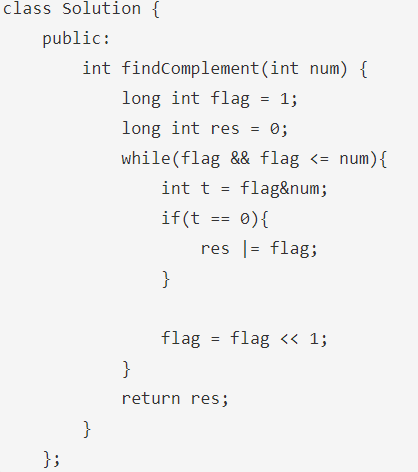
Output: 0

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

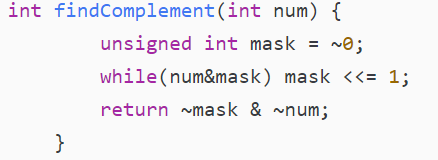
**思路**

按位取反，即可

解法一:



解法二:



**Advanced:**

**260. Single Number III (Medium)**

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

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

**Output:** [3,5]

Follow up:

* The order of the result is not important. So in the above example, [5, 3] is also correct.
* Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

**我的思路**

我的解法使用了之前已经使用过的set容器，具体内容请参照

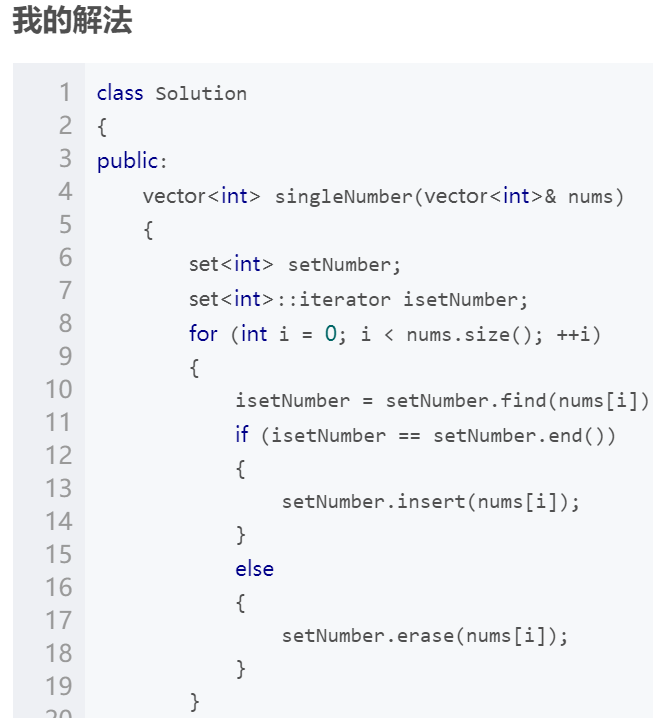
LeetCode 136. Single Number 题解（C++）

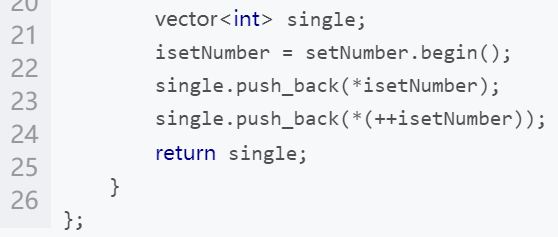
**最佳解法**

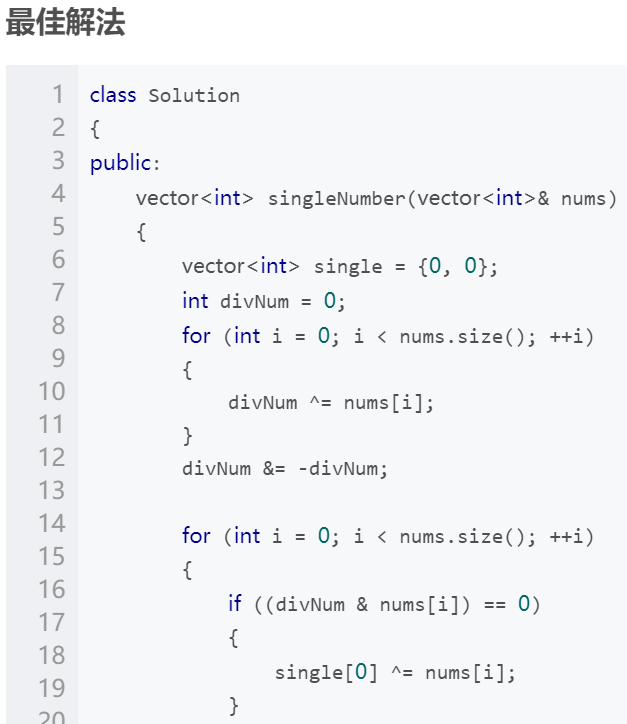
最佳解法使用了异或位运算符。首先将所有的元素异或，得到的结果divNum为两个只出现一次的数字（即所求的答案）的异或。

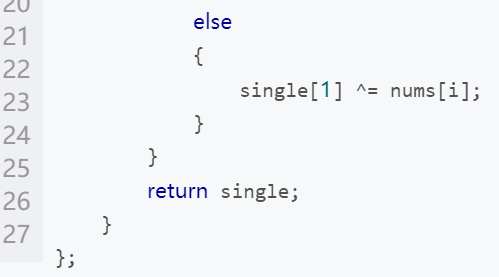
这里假设两个数字分别为A和B，即A^B=divNum，因为A和B是两个不同的数字，所以它们的二进制中至少有一位是不一样的，这里我们需要先找到它们不一样的位置(这里必然有一个是1，另一个为0）。通过divNum与自己的负数-divNum相与，可以得到A和B的二进制中不一样的且最靠近右侧的位数的位置（该位置被置为1），并且其它的位置都为0。比如divNum是0110，则-divNum是1010，divNum&-divNum=0010

这里需要将A和B分为两组元素并分别进行异或位操作。将divNum与每个元素相与，若为0，则该元素属于同一组，若为1，则元素属于另外一组。之后再将两组元素分别进行异或操作，便可得到结果。









**Chapter 11 Data Structure**

566. Reshape the Matrix(Easy)

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

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

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

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

Input:

nums =

[[1,2],

[3,4]]

r = 1, c = 4

Output:

[[1,2,3,4]]

Explanation:

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

Note:

The height and width of the given matrix is in range [1, 100].

The given r and c are all positive.

分析：

给定一个矩阵，根据参数返回一个新的矩阵。

可以发现，只有给的r \* c和原来矩阵行列乘积相同才可以将矩阵正确的变形，否则返回原来的矩阵。用f来确定原矩阵中元素的索引。直接两重循环将元素填进新矩阵中。



225. Implement Stack using Queues

Implement the following operations of a stack using queues.

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

Example:

MyStack stack = new MyStack();

stack.push(1);

stack.push(2);

stack.top(); // returns 2

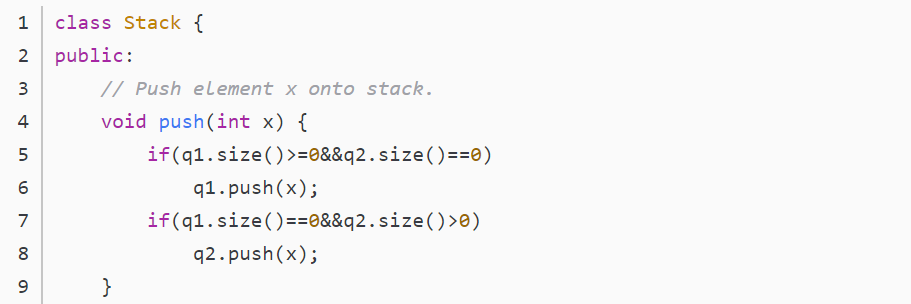
stack.pop(); // returns 2

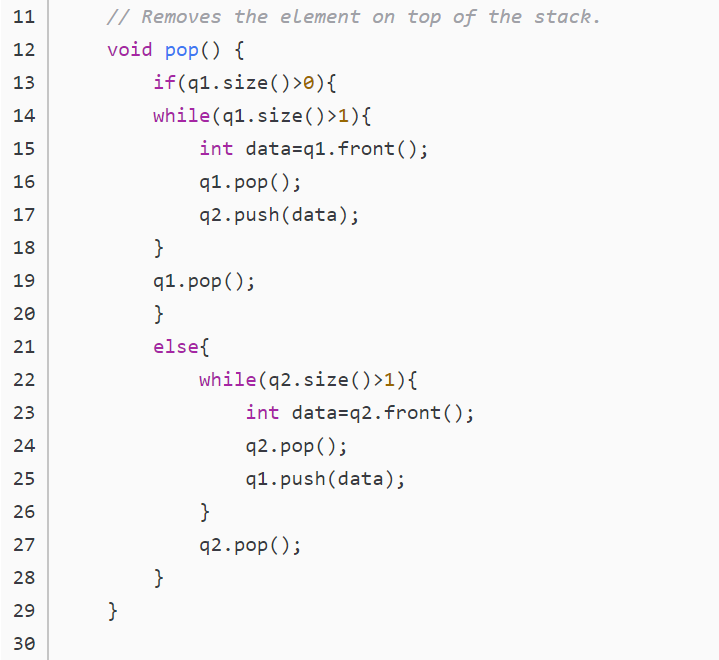
stack.empty(); // returns false

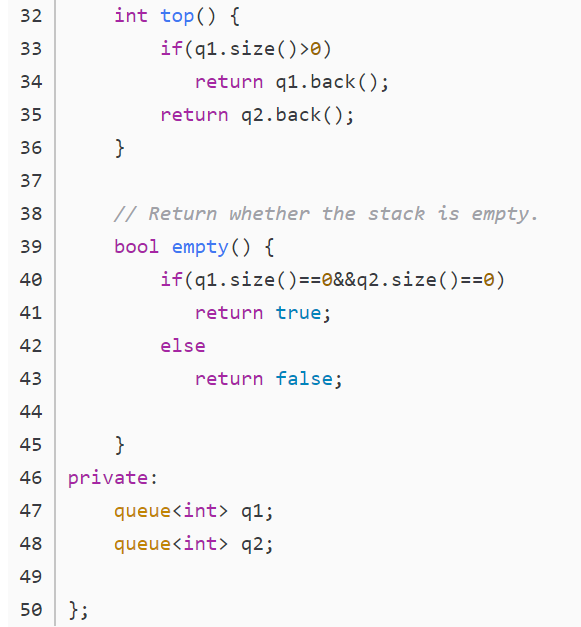
Notes:

* 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).

解题思路：用两个队列来实现。







503. Next Greater Element II

Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.

**Example 1:**

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

**Output:** [2,-1,2]

**Explanation:** The first 1's next greater number is 2;

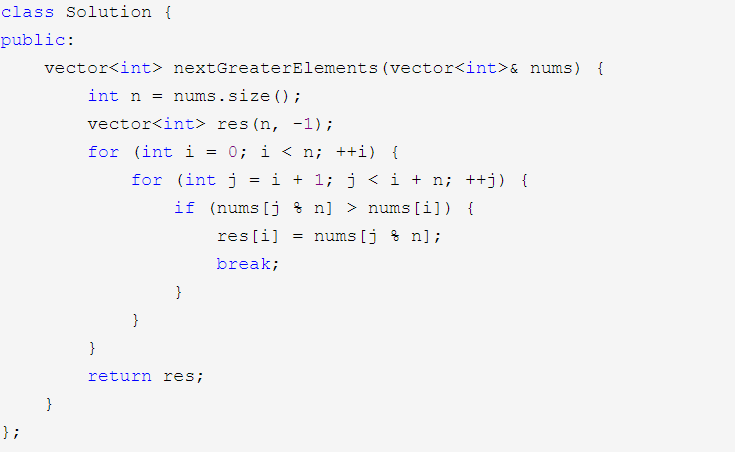
The number 2 can't find next greater number;

The second 1's next greater number needs to search circularly, which is also 2.

**Note:** The length of given array won't exceed 10000.

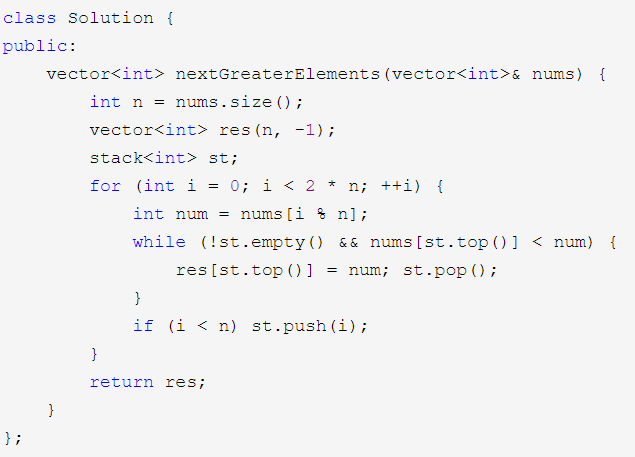
这道题是之前那道Next Greater Element I的拓展，不同的是，此时数组是一个循环数组，就是说某一个元素的下一个较大值可以在其前面，那么对于循环数组的遍历，为了使下标不超过数组的长度，我们需要对n取余，下面先来看暴力破解的方法，遍历每一个数字，然后对于每一个遍历到的数字，遍历所有其他数字，注意不是遍历到数组末尾，而是通过循环数组遍历其前一个数字，遇到较大值则存入结果res中，并break，再进行下一个数字的遍历，参见代码如下：

解法一：



我们可以使用栈来进行优化上面的算法，我们遍历两倍的数组，然后还是坐标i对n取余，取出数字，如果此时栈不为空，且栈顶元素小于当前数字，说明当前数字就是栈顶元素的右边第一个较大数，那么建立二者的映射，并且去除当前栈顶元素，最后如果i小于n，则把i压入栈。因为res的长度必须是n，超过n的部分我们只是为了给之前栈中的数字找较大值，所以不能压入栈，参见代码如下：

解法二：



**217. Contains Duplicate**

Given an array of integers, find if the array contains any duplicates.

Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

Example 1:

Input: [1,2,3,1]

Output: true

Example 2:

Input: [1,2,3,4]

Output: false

Example 3:

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

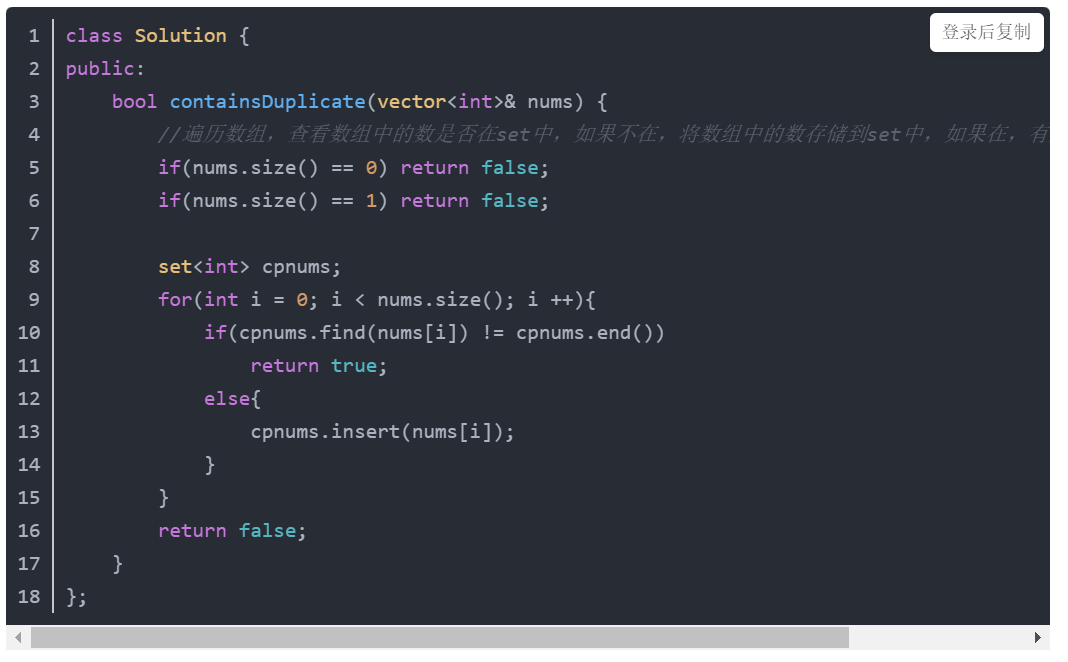
Output: true

思路分析：

方法一：

判断数组中的数是否重复，有重复返回true。无重复返回false。遍历数组，将数组中的数存储到set中，每次插入nums[ i ]之前判断是否已在set中存在，如果已经存在，说明存在重复，返回true。

时间复杂度：O(n) 空间复杂度：O(n)

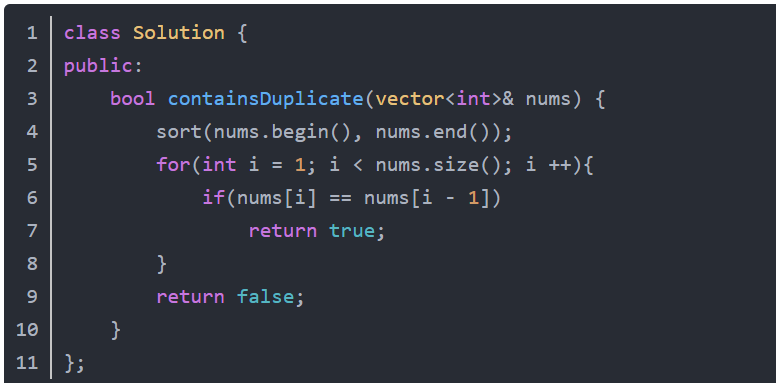


如果要求空间复杂度为O(1)呢？

方法二：

可以先将数组排序，然后判断相邻的两个数是否有相等的，有，说明有重复，返回true。

时间复杂度：O(n) 空间复杂度：O(1)



697. Degree of an Array

Given a non-empty array of non-negative integers nums, the degree of this array is defined as the maximum frequency of any one of its elements.

Your task is to find the smallest possible length of a (contiguous) subarray of nums, that has the same degree as nums.

Example 1:

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

Output: 2

Explanation:

The input array has a degree of 2 because both elements 1 and 2 appear twice.

Of the subarrays that have the same degree:

[1, 2, 2, 3, 1], [1, 2, 2, 3], [2, 2, 3, 1], [1, 2, 2], [2, 2, 3], [2, 2]

The shortest length is 2. So return 2.

Example 2:

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

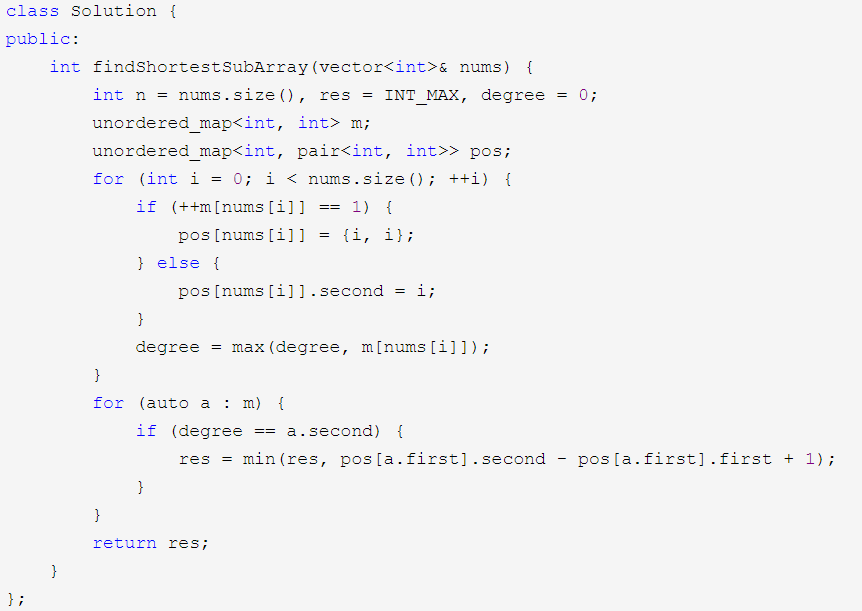
Output: 6

Note:

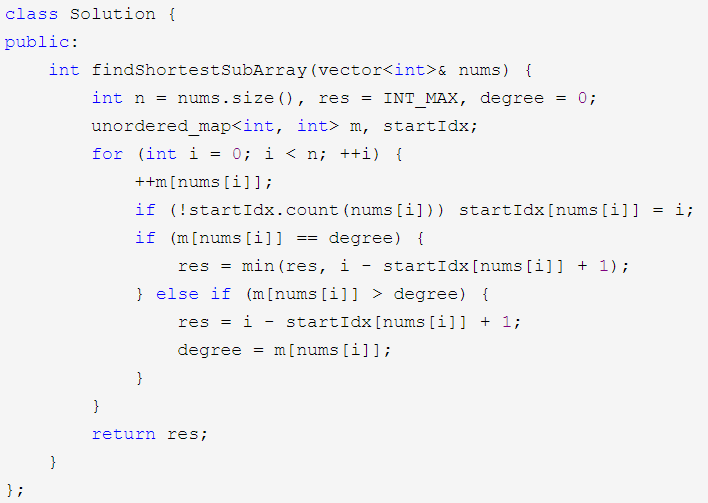
nums.length will be between 1 and 50,000.

nums[i] will be an integer between 0 and 49,999.

这道题给了我们一个数组，定义数组的度为某个或某些数字出现最多的次数，要我们找最短的子数组使其和原数组拥有相同的度。那么我们肯定需要统计每个数字出现的次数，就要用哈希表来建立每个数字和其出现次数之间的映射。由于我们要求包含原度的最小长度的子数组，那么最好的情况就是子数组的首位数字都是统计度的数字，即出现最多的数字。那么我们肯定要知道该数字的第一次出现的位置和最后一次出现的位置，由于我们开始不知道哪些数字会出现最多次，所以我们统计所有数字的首尾出现位置，那么我们再用一个哈希表，建立每个数字和其首尾出现的位置。我们用变量degree来表示数组的度。好，现在我们遍历原数组，累加当前数字出现的次数，当某个数字是第一次出现，那么我们用当前位置的来更新该数字出现的首尾位置，否则只更新尾位置。每遍历一个数，我们都更新一下degree。当遍历完成后，我们已经有了数组的度，还有每个数字首尾出现的位置，下面就来找出现次数为degree的数组，然后计算其首尾位置差加1就是candidate数组的长度，由于出现次数为degree的数字不一定只有一个，我们遍历所有的，找出其中最小的即可，参见代码如下



下面这种方法只用了一次遍历，思路跟上面的解法很相似，还是要建立数字出现次数的哈希表，还有就是建立每个数字和其第一次出现位置之间的映射，那么我们当前遍历的位置其实可以看作是尾位置，还是可以计算子数组的长度的。我们遍历数组，累加当前数字出现的次数，如果某个数字是第一次出现，建立该数字和当前位置的映射，如果当前数字的出现次数等于degree时，当前位置为尾位置，首位置在startIdx中取的，二者做差加1来更新结果res；如果当前数字的出现次数大于degree，说明之前的结果代表的数字不是出现最多的，直接将结果res更新为当前数字的首尾差加1的长度，然后degree也更新为当前数字出现的次数。参见代码如下



594. Longest Harmonious Subsequence

We define a harmonious array is an array where the difference between its maximum value and its minimum value is exactly 1.

Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible [subsequences](https://en.wikipedia.org/wiki/Subsequence).

Example 1:

Input: [1,3,2,2,5,2,3,7]

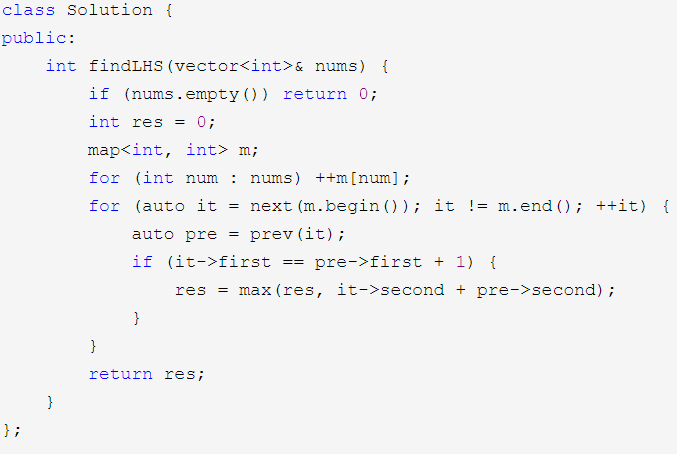
Output: 5

Explanation: The longest harmonious subsequence is [3,2,2,2,3].

Note: The length of the input array will not exceed 20,000.

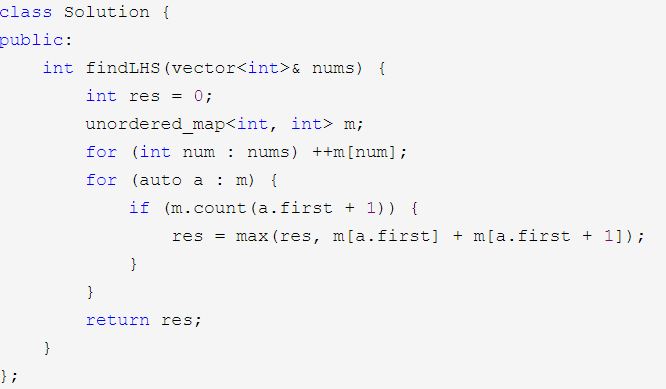
这道题给了我们一个数组，让我们找出最长的和谐子序列，关于和谐子序列就是序列中数组的最大最小差值均为1。由于这里只是让我们求长度，并不需要返回具体的子序列。所以我们可以对数组进行排序，那么实际上我们只要找出来相差为1的两个数的总共出现个数就是一个和谐子序列的长度了。明白了这一点，我们就可以建立一个数字和其出现次数之间的映射，利用 TreeMap 的自动排序的特性，那么我们遍历 TreeMap 的时候就是从小往大开始遍历，我们从第二个映射对开始遍历，每次跟其前面的映射对比较，如果二者的数字刚好差1，那么就把二个数字的出现的次数相加并更新结果 res 即可，参见代码如下：

解法一:



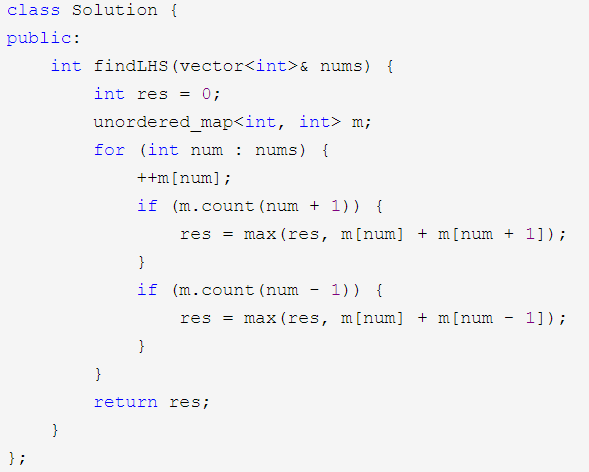
其实我们并不用向上面那种解法那样用 next 和 prev 来移动迭代器，因为其用到了 TreeMap 的自动排序功能，所以才可以利用 next 和 prev。其实我们还可以用 HashMap 来做，先遍历一遍，建立每个数字跟其出现次数之间的映射，然后再遍历每个数字的时候，只需在 HashMap 中查找该数字加1是否存在，存在就更新结果 res，这样更简单一些，参见代码如下：

解法二：



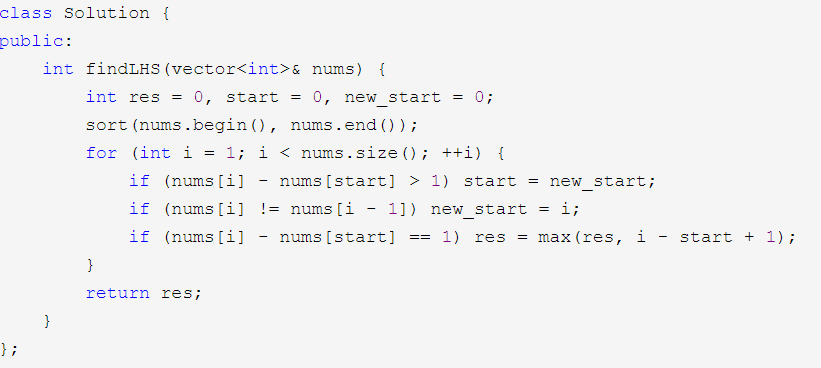
我们其实也可以在一个 for 循环中搞定，遍历每个数字时，先累加其映射值，然后查找该数字加1是否存在，存在的话用 m[num] 和 m[num+1] 的和来更新结果 res，同时，还要查找该数字减1是否存在，存在的话用 m[num] 和 m[num-1] 的和来更新结果 res，这样也是可以的，参见代码如下：

解法三：



下面方法不用任何 map，但是需要对数组进行排序，当数组有序了之后，我们就可以一次遍历搞定了。这实际上用到了滑动窗口 Sliding Window 的思想，用变量 start 记录当前窗口的左边界，初始化为0。用 new\_start 指向下一个潜在窗口的左边界，初始化为0。i为当前窗口的右边界，从1开始遍历，首先验证当前窗口的差值是否小于1，用 nums[i] 减去 nums[start]，若不满足，则将 start 赋值为 new\_start，即移动到下一个窗口。然后看当前数字跟之前一个数字是否相等，若不相等，说明当前数字可能是下一个潜在窗口的左边界，将 new\_start 赋值为i。然后再看窗口的左右边界值是否刚好为1，因为题目中说了差值必须正好为1，由于我们对数组排序了，所以只要左右边界差值正好为1，那么这个窗口包含的数字就可以组成满足题意的子序列，用其长度来更新结果 res 即可，参见代码如下：

解法四:



287. Find the Duplicate Number

Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Example 1:

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

Output: 2

Example 2:

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

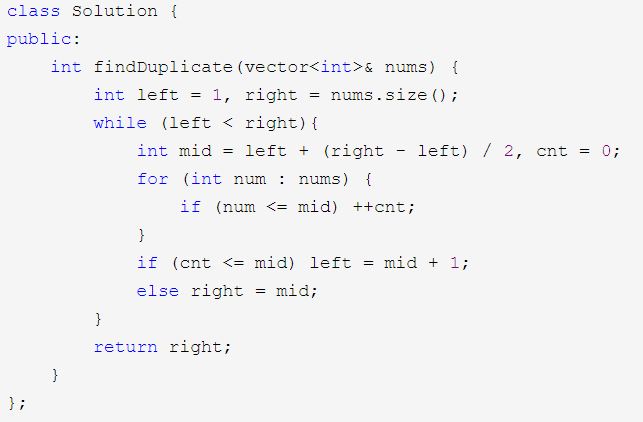
Output: 3

Note:

1. You must not modify the array (assume the array is read only).
2. You must use only constant, *O*(1) extra space.
3. Your runtime complexity should be less than *O*(*n*2).
4. There is only one duplicate number in the array, but it could be repeated more than once.

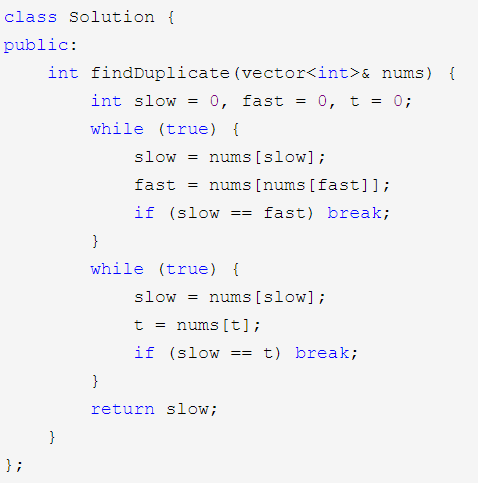
这道题给了我们 n+1 个数，所有的数都在 [1, n] 区域内，首先让证明必定会有一个重复数，这不禁让博主想起了小学华罗庚奥数中的抽屉原理(又叫[鸽巢原理](https://zh.wikipedia.org/wiki/%E9%B4%BF%E5%B7%A2%E5%8E%9F%E7%90%86))，即如果有十个苹果放到九个抽屉里，如果苹果全在抽屉里，则至少有一个抽屉里有两个苹果，这里就不证明了，直接来做题吧。题目要求不能改变原数组，即不能给原数组排序，又不能用多余空间，那么哈希表神马的也就不用考虑了，又说时间小于 O(n2)，也就不能用 brute force 的方法，那也就只能考虑用二分搜索法了，在区间 [1, n] 中搜索，首先求出中点 mid，然后遍历整个数组，统计所有小于等于 mid 的数的个数，如果个数小于等于 mid，则说明重复值在 [mid+1, n] 之间，反之，重复值应在 [1, mid-1] 之间，然后依次类推，直到搜索完成，此时的 low 就是我们要求的重复值，参见代码如下：

解法一：



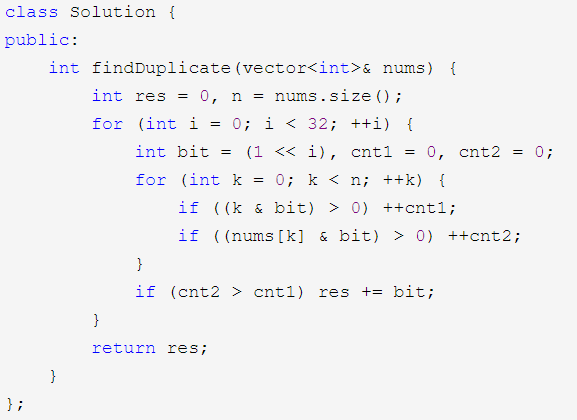
经过热心网友 [waruzhi](http://www.cnblogs.com/waruzhi/) 的留言提醒还有一种 O(n) 的解法，并给了[参考帖子](http://bookshadow.com/weblog/2015/09/28/leetcode-find-duplicate-number/)，发现真是一种不错的解法，其核心思想快慢指针在之前的题目 [Linked List Cycle II](http://www.cnblogs.com/grandyang/p/4137302.html) 中就有应用，这里应用的更加巧妙一些，由于题目限定了区间 [1,n]，所以可以巧妙的利用坐标和数值之间相互转换，而由于重复数字的存在，那么一定会形成环，用快慢指针可以找到环并确定环的起始位置，确实是太巧妙了！

解法二：



这道题还有一种位操作 Bit Manipulation 的解法，也十分的巧妙。思路是遍历每一位，然后对于 32 位中的每一个位 bit，都遍历一遍从0到 n-1，将0到 n-1 中的每一个数都跟 bit 相 ‘与’，若大于0，则计数器 cnt1 自增1。同时0到 n-1 也可以当作 nums 数组的下标，从而让 nums 数组中的每个数字也跟 bit 相 ‘与’，若大于0，则计数器 cnt2 自增1。最后比较若 cnt2 大于 cnt1，则将 bit 加入结果 res 中。这是为啥呢，因为对于每一位，0到 n-1 中所有数字中该位上的1的个数应该是固定的，如果 nums 数组中所有数字中该位上1的个数多了，说明重复数字在该位上一定是1，这样我们把重复数字的所有为1的位都累加起来，就可以还原出了这个重复数字，参见代码如下：

解法三：



313. Super Ugly Number

Write a program to find the nth super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of sizek. For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4.

Note:

(1) 1 is a super ugly number for any given primes.

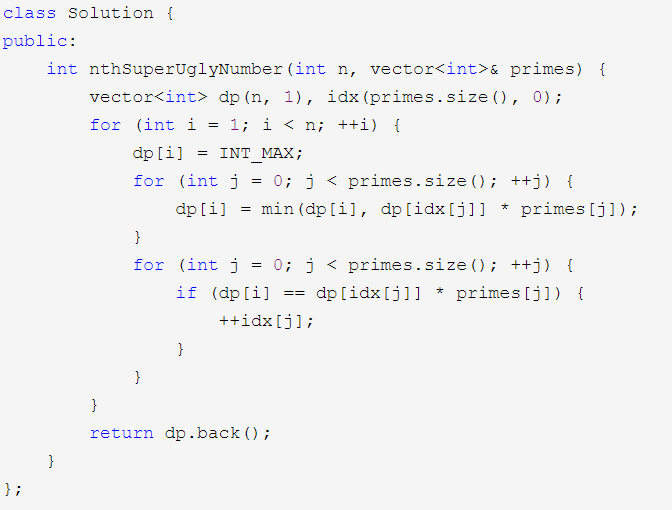
(2) The given numbers in primes are in ascending order.

(3) 0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000.

Credits:

Special thanks to [@dietpepsi](https://leetcode.com/discuss/user/dietpepsi) for adding this problem and creating all test cases.

这道题让我们求超级丑陋数，是之前那两道[Ugly Number 丑陋数](http://www.cnblogs.com/grandyang/p/4741934.html)和[Ugly Number II 丑陋数之二](http://www.cnblogs.com/grandyang/p/4743837.html)的延伸，质数集合可以任意给定，这就增加了难度。但是本质上和[Ugly Number II 丑陋数之二](http://www.cnblogs.com/grandyang/p/4743837.html)没有什么区别，由于我们不知道质数的个数，我们可以用一个idx数组来保存当前的位置，然后我们从每个子链中取出一个数，找出其中最小值，然后更新idx数组对应位置，注意有可能最小值不止一个，要更新所有最小值的位置，参见代码如下：



870. Advantage Shuffle

Given two arrays A and B of equal size, the *advantage of A with respect to B* is the number of indices i for which A[i] > B[i].

Return **any** permutation of A that maximizes its advantage with respect to B

Example 1:

Input: A = [2,7,11,15], B = [1,10,4,11]

Output: [2,11,7,15]

**Example 2:**

**Input:** A = [12,24,8,32], B = [13,25,32,11]

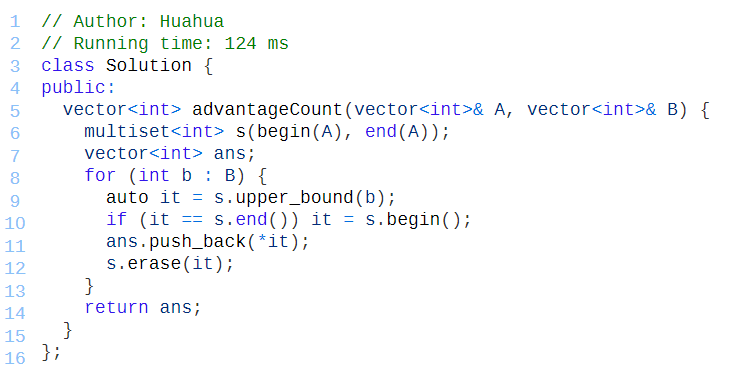
**Output:** [24,32,8,12]

Note:

* 1 <= A.length = B.length <= 10000
* 0 <= A[i] <= 10^9
* 0 <= B[i] <= 10^9

# **Solution: Greedy 田忌赛马**

Use the smallest unused number A[j] in A such that A[j] > B[i], if not possible, use the smallest number in A. ( Time complexity: O(nlogn) Space complexity: O(n) )



307. Range Sum Query - Mutable

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

The *update(i, val)* function modifies *nums* by updating the element at index *i* to *val*.

**Example:**

Given nums = [1, 3, 5]

sumRange(0, 2) -> 9

update(1, 2)

sumRange(0, 2) -> 8

**Note:**

1. The array is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRange* function is distributed evenly.

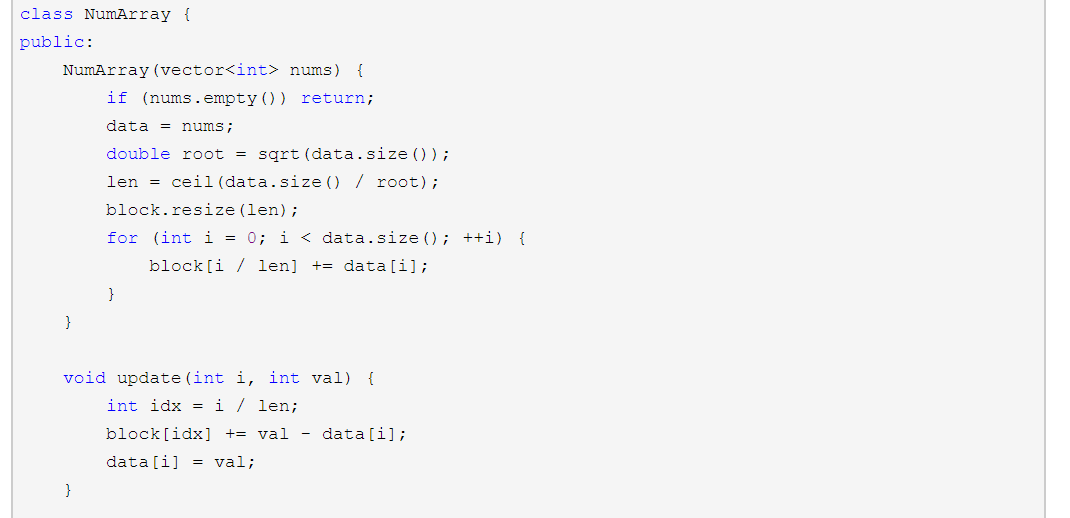
这道题是之前那道 [Range Sum Query - Immutable](http://www.cnblogs.com/grandyang/p/4952464.html) 的延伸，之前那道题由于数组的内容不会改变，所以我们只需要建立一个累计数组就可以支持快速的计算区间值了，而这道题说数组的内容会改变，如果我们还是用之前的方法建立累计和数组，那么每改变一个数字，之后所有位置的数字都要改变，这样如果有很多更新操作的话，就会十分不高效，估计很难通过吧。But，被 OJ 分分钟打脸， brute force 完全没有问题啊，这年头，装个比不容易啊。直接就用个数组 data 接住 nums，然后要更新就更新，要求区域和，就遍历求区域和，就这样 naive 的方法还能 beat 百分之二十多啊，这不科学啊，参见代码如下：

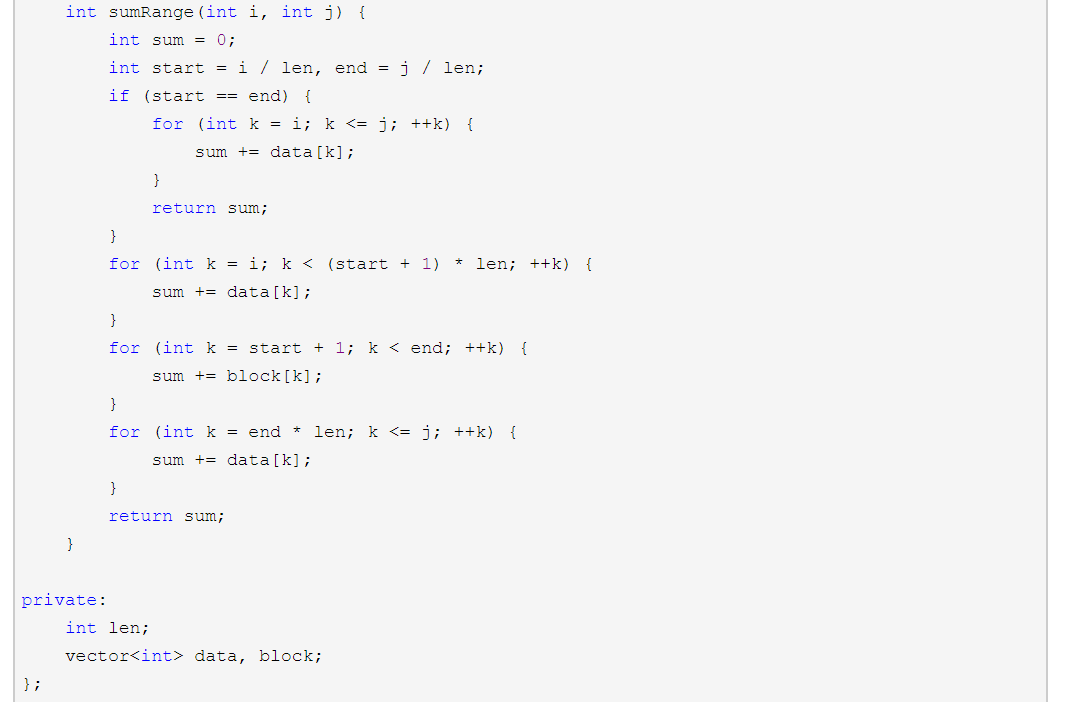
解法一：





咳咳，下面就开始闪亮的装比时间了，光芒必将盖过坂本大佬。上面的方法最大的问题，就是求区域和不高效，如果数组很大很大，每次求一个巨型的区间的和，都要一个一个的遍历去累加，累啊～但是一般的累加数组又无法应对这里的 update 操作，随便修改一个数字的话，那么其之后的所有累加和都会发生改变。所以解决方案就是二者折中一下，分块累加，各不干预。就是将原数组分为若干块，怎么分呢，这里就让每个 block 有 sqrt(n) 个数字就可以了，这个基本是让 block 的个数跟每个 blcok 中数字的个数尽可能相同的分割方法。然后我们就需要一个大小跟 block 个数相同的数组，来保存每个 block 的数字之和。在需要更新的时候，我们就先确定要更新的位置在哪个 block 里，然后只更新该 block 的和。而对于求区域和操作，我们还是要分别确定i和j分别属于哪个 block，若属于同一个 block，那么直接遍历累加即可，若属于不同的，则先从i累加到该 blcok 的末尾，然后中间横跨的那些 block 可以直接将和累加，对于j所在的 blcok，则从该 block 的开头遍历累加到j即可，参见代码如下：





…

…

…

還有一堆解法出來自己看

**Chapter 12 String**

409. Longest Palindrome

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

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

**Note:**

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

**Example:**

Input:

"abccccdd"

Output:

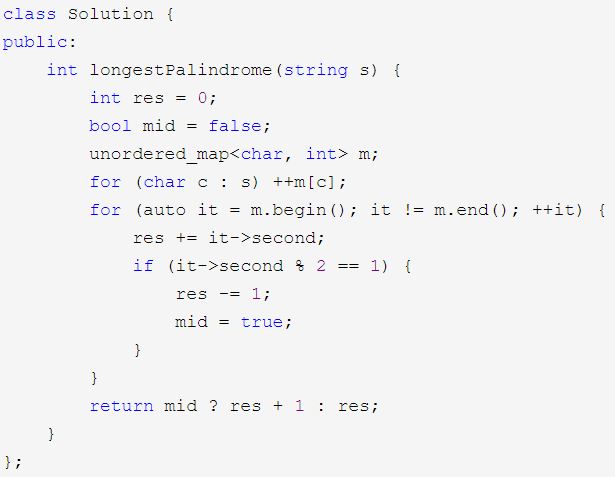
7

Explanation:

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

这又是一道关于回文字符串的问题，LeetCode上关于回文串的题有十来道呢，也算一个比较重要的知识点。但是这道题确实不算一道难题，给了我们一个字符串，让我们找出可以组成的最长的回文串的长度，由于字符顺序可以打乱，所以问题就转化为了求偶数个字符的个数，我们了解回文串的都知道，回文串主要有两种形式，一个是左右完全对称的，比如noon, 还有一种是以中间字符为中心，左右对称，比如bob，level等，那么我们统计出来所有偶数个字符的出现总和，然后如果有奇数个字符的话，我们取取出其最大偶数，然后最后结果加1即可，参见代码如下：

解法一：



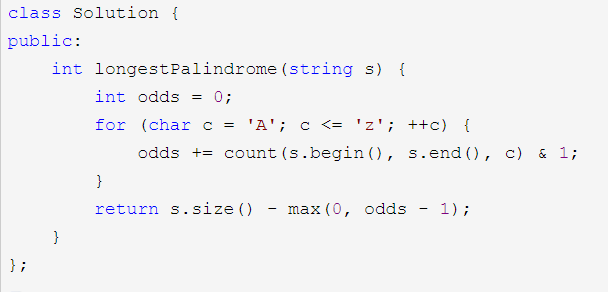
上面那种方法是通过哈希表来建立字符串和其出现次数的映射，这里我们可以换一种思路，来找出搜有奇数个的字符，我们采用的方法是使用一个set集合，如果遍历到的字符不在set中，那么就将其加入set，如果已经在set里了，就将其从set中删去，这样遍历完成后set中就是所有出现个数是奇数个的字符了，那么我们最后只要用s的长度减去0和set长度减一之间的较大值即可，为啥这样呢，我们想，如果没有出现个数是奇数个的字符，那么t的长度就是0，减1成了-1，那么s的长度只要减去0即可；如果有奇数个的字符，那么字符个数减1，就是不能组成回文串的字符，因为回文串最多允许一个不成对出现的字符，参见代码如下：

解法二：



最后这种方法利用到了STL中的count函数，就是找字符串中某个字符出现的个数，那么我们和1相与，就可以知道该个数是奇数还是偶数了，返回的写法和上面那种方法相同，参见代码如下：

解法三：



3. Longest Substring Without Repeating Characters

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

Example 1:

Input: "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: "bbbbb"

Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: "pwwkew"

Output: 3

Explanation: The answer is "wke", with the length of 3.

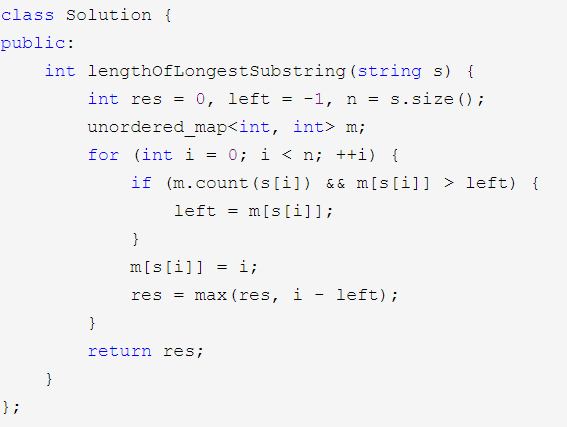
Note that the answer must be a substring, "pwke" is a *subsequence* and not a substring.

这道求最长无重复子串的题和之前那道 [Isomorphic Strings](http://www.cnblogs.com/grandyang/p/4465779.html) 很类似，属于 LeetCode 早期经典题目，博主认为是可以跟 [Two Sum](http://www.cnblogs.com/grandyang/p/4130379.html) 媲美的一道题。给了我们一个字符串，让求最长的无重复字符的子串，注意这里是子串，不是子序列，所以必须是连续的。先不考虑代码怎么实现，如果给一个例子中的例子 "abcabcbb"，让你手动找无重复字符的子串，该怎么找。博主会一个字符一个字符的遍历，比如 a，b，c，然后又出现了一个a，那么此时就应该去掉第一次出现的a，然后继续往后，又出现了一个b，则应该去掉一次出现的b，以此类推，最终发现最长的长度为3。所以说，需要记录之前出现过的字符，记录的方式有很多，最常见的是统计字符出现的个数，但是这道题字符出现的位置很重要，所以可以使用 HashMap 来建立字符和其出现位置之间的映射。进一步考虑，由于字符会重复出现，到底是保存所有出现的位置呢，还是只记录一个位置？我们之前手动推导的方法实际上是维护了一个滑动窗口，窗口内的都是没有重复的字符，需要尽可能的扩大窗口的大小。由于窗口在不停向右滑动，所以只关心每个字符最后出现的位置，并建立映射。窗口的右边界就是当前遍历到的字符的位置，为了求出窗口的大小，需要一个变量 left 来指向滑动窗口的左边界，这样，如果当前遍历到的字符从未出现过，那么直接扩大右边界，如果之前出现过，那么就分两种情况，在或不在滑动窗口内，如果不在滑动窗口内，那么就没事，当前字符可以加进来，如果在的话，就需要先在滑动窗口内去掉这个已经出现过的字符了，去掉的方法并不需要将左边界 left 一位一位向右遍历查找，由于 HashMap 已经保存了该重复字符最后出现的位置，所以直接移动 left 指针就可以了。维护一个结果 res，每次用出现过的窗口大小来更新结果 res，就可以得到最终结果啦。

这里可以建立一个 HashMap，建立每个字符和其最后出现位置之间的映射，然后需要定义两个变量 res 和 left，其中 res 用来记录最长无重复子串的长度，left 指向该无重复子串左边的起始位置的前一个，由于是前一个，所以初始化就是 -1，然后遍历整个字符串，对于每一个遍历到的字符，如果该字符已经在 HashMap 中存在了，并且如果其映射值大于 left 的话，那么更新 left 为当前映射值。然后映射值更新为当前坐标i，这样保证了 left 始终为当前边界的前一个位置，然后计算窗口长度的时候，直接用 i-left 即可，用来更新结果 res。

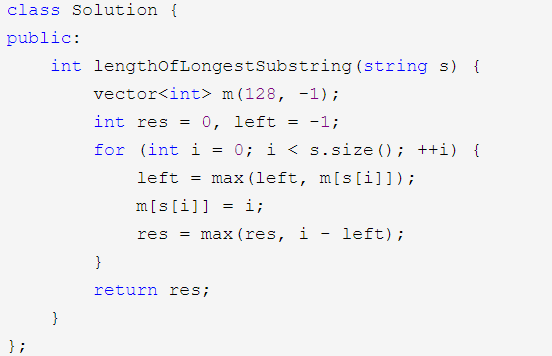
这里解释下程序中那个 if 条件语句中的两个条件 m.count(s[i]) && m[s[i]] > left，因为一旦当前字符 s[i] 在 HashMap 已经存在映射，说明当前的字符已经出现过了，而若 m[s[i]] > left 成立，说明之前出现过的字符在窗口内，那么如果要加上当前这个重复的字符，就要移除之前的那个，所以让 left 赋值为 m[s[i]]，由于 left 是窗口左边界的前一个位置（这也是 left 初始化为 -1 的原因，因为窗口左边界是从0开始遍历的），所以相当于已经移除出滑动窗口了。举一个最简单的例子 "aa"，当 i=0 时，建立了 a->0 的映射，并且此时结果 res 更新为1，那么当 i=1 的时候，发现a在 HashMap 中，并且映射值0大于 left 的 -1，所以此时 left 更新为0，映射对更新为 a->1，那么此时 i-left 还为1，不用更新结果 res，那么最终结果 res 还为1，正确，代码如下：

C++ 解法一：



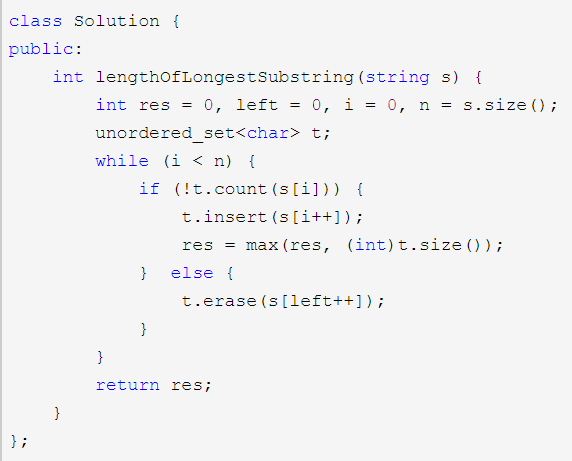
下面这种写法是上面解法的精简模式，这里我们可以建立一个 256 位大小的整型数组来代替 HashMap，这样做的原因是 ASCII 表共能表示 256 个字符，但是由于键盘只能表示 128 个字符，所以用 128 也行，然后全部初始化为 -1，这样的好处是不用像之前的 HashMap 一样要查找当前字符是否存在映射对了，对于每一个遍历到的字符，直接用其在数组中的值来更新 left，因为默认是 -1，而 left 初始化也是 -1，所以并不会产生错误，这样就省了 if 判断的步骤，其余思路都一样：

C++ 解法二：



下面这种解法使用了 HashSet，核心算法和上面的很类似，把出现过的字符都放入 HashSet 中，遇到 HashSet 中没有的字符就加入 HashSet 中并更新结果 res，如果遇到重复的，则从左边开始删字符，直到删到重复的字符停止：

C++ 解法三：



772. Basic Calculator III (Hard)

要付費

5. Longest Palindromic Substring

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

Example 1:

Input: "babad"

Output: "bab"

Note: "aba" is also a valid answer.

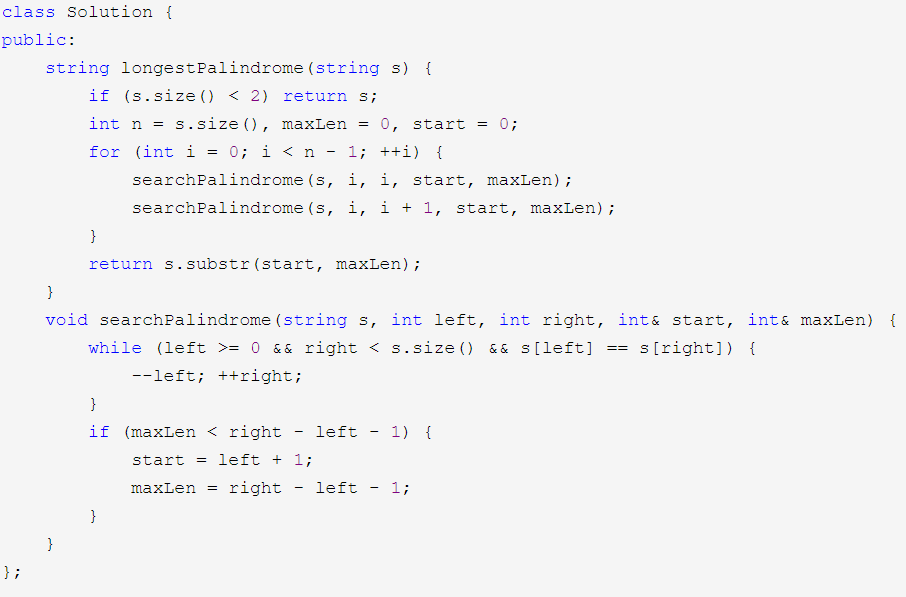
Example 2:

Input: "cbbd"

Output: "bb"

这道题让我们求[最长回文子串](http://en.wikipedia.org/wiki/Longest_palindromic_substring)，首先说下什么是回文串，就是正读反读都一样的字符串，比如 "bob", "level", "noon" 等等。那么最长回文子串就是在一个字符串中的那个最长的回文子串。LeetCode 中关于回文串的题共有五道，除了这道，其他的四道为 [Palindrome Number](http://www.cnblogs.com/grandyang/p/4125510.html)，[Validate Palindrome](http://www.cnblogs.com/grandyang/p/4030114.html)，[Palindrome Partitioning](http://www.cnblogs.com/grandyang/p/4270008.html)，[Palindrome Partitioning II](http://www.cnblogs.com/grandyang/p/4271456.html)，我们知道传统的验证回文串的方法就是两个两个的对称验证是否相等，那么对于找回文字串的问题，就要以每一个字符为中心，像两边扩散来寻找回文串，这个算法的时间复杂度是 O(n\*n)，可以通过 OJ，就是要注意奇偶情况，由于回文串的长度可奇可偶，比如 "bob" 是奇数形式的回文，"noon" 就是偶数形式的回文，两种形式的回文都要搜索，对于奇数形式的，我们就从遍历到的位置为中心，向两边进行扩散，对于偶数情况，我们就把当前位置和下一个位置当作偶数行回文的最中间两个字符，然后向两边进行搜索，参见代码如下：

解法一：



此题还可以用动态规划 Dynamic Programming 来解，根 [Palindrome Partitioning II](http://www.cnblogs.com/grandyang/p/4271456.html) 的解法很类似，我们维护一个二维数组 dp，其中 dp[i][j] 表示字符串区间 [i, j] 是否为回文串，当 i = j 时，只有一个字符，肯定是回文串，如果 i = j + 1，说明是相邻字符，此时需要判断 s[i] 是否等于 s[j]，如果i和j不相邻，即 i - j >= 2 时，除了判断 s[i] 和 s[j] 相等之外，dp[i + 1][j - 1] 若为真，就是回文串，通过以上分析，可以写出递推式如下：

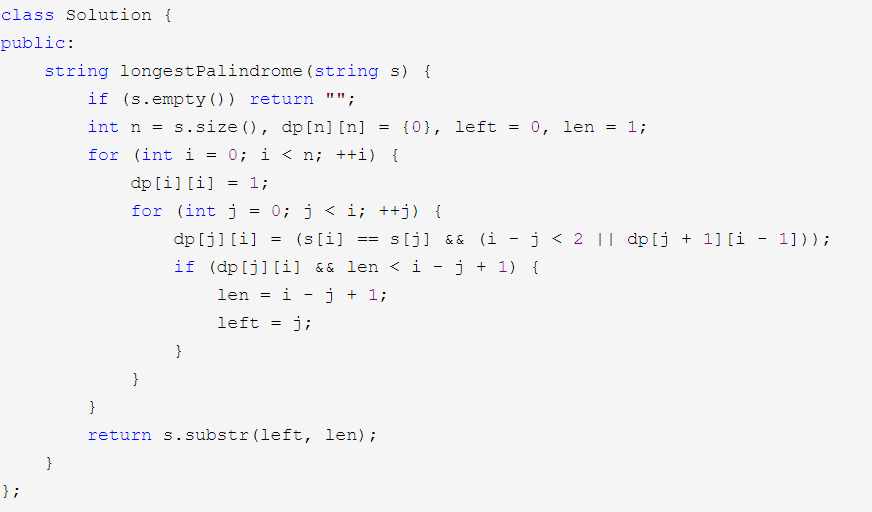
dp[i, j] = 1 if i == j

= s[i] == s[j] if j = i + 1

= s[i] == s[j] && dp[i + 1][j - 1] if j > i + 1

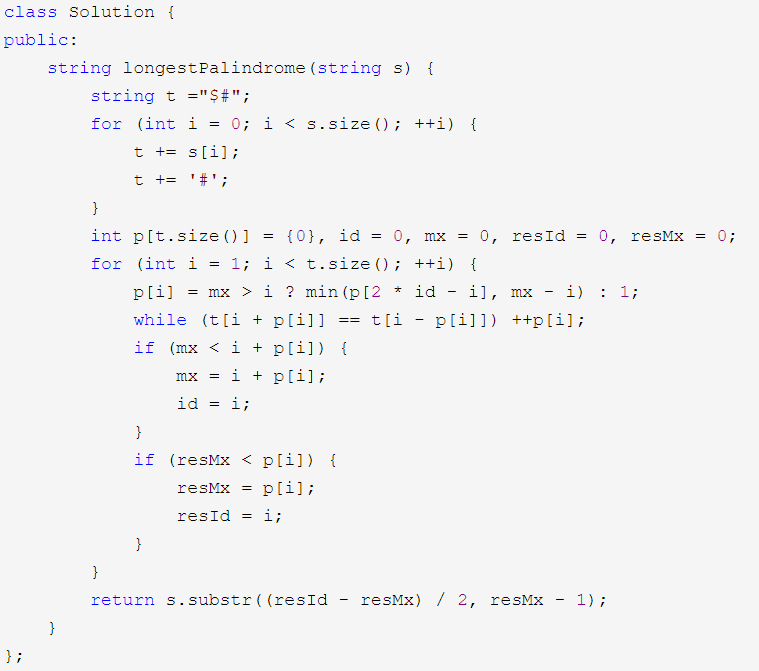
这里有个有趣的现象就是如果我把下面的代码中的二维数组由 int 改为 vector<vector<int>> 后，就会超时，这说明 int 型的二维数组访问执行速度完爆 std 的 vector 啊，所以以后尽可能的还是用最原始的数据类型吧。

解法二：



最后要来的就是大名鼎鼎的马拉车算法 Manacher's Algorithm，这个算法的神奇之处在于将时间复杂度提升到了 O(n) 这种逆天的地步，而算法本身也设计的很巧妙，很值得我们掌握，参见我另一篇专门介绍马拉车算法的博客 [Manacher's Algorithm 马拉车算法](http://www.cnblogs.com/grandyang/p/4475985.html)，代码实现如下：

解法三：



**Chapter 13 Linked-List**

83. Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only *once*.

Example 1:

Input: 1->1->2

Output: 1->2

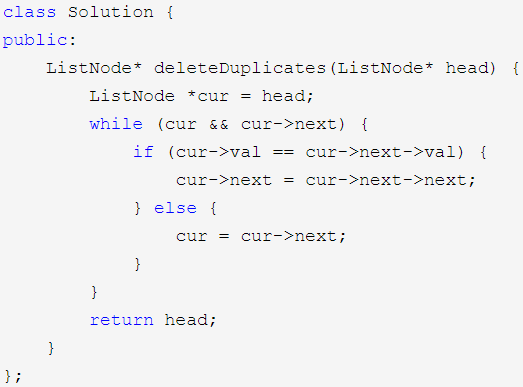
Example 2:

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

Output: 1->2->3

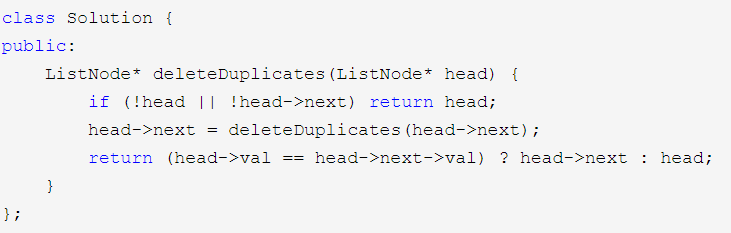
这道题让我们移除给定有序链表的重复项，那么可以遍历这个链表，每个结点和其后面的结点比较，如果结点值相同了，只要将前面结点的 next 指针跳过紧挨着的相同值的结点，指向后面一个结点。这样遍历下来，所有重复的结点都会被跳过，留下的链表就是没有重复项的了，代码如下：

解法一：



我们也可以使用递归的方法来做，首先判断是否至少有两个结点，若不是的话，直接返回 head。否则对 head->next 调用递归函数，并赋值给 head->next。这里可能比较晕，先看后面一句，返回的时候，head 结点先跟其身后的结点进行比较，如果值相同，那么返回后面的一个结点，当前的 head 结点就被跳过了，而如果不同的话，还是返回 head 结点。可以发现了，进行实质上的删除操作是在最后一句进行了，再来看第二句，对 head 后面的结点调用递归函数，那么就应该 suppose 返回来的链表就已经没有重复项了，此时接到 head 结点后面，在第三句的时候再来检查一下 head 是否又 duplicate 了，实际上递归一直走到了末尾结点，再不断的回溯回来，进行删除重复结点，参见代码如下：

解法二：



328. Odd Even Linked List

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example 1:

Input: 1->2->3->4->5->NULL

Output: 1->3->5->2->4->NULL

Example 2:

Input: 2->1->3->5->6->4->7->NULL

Output: 2->3->6->7->1->5->4->NULL

Constraints:

* The relative order inside both the even and odd groups should remain as it was in the input.
* The first node is considered odd, the second node even and so on ...
* The length of the linked list is between [0, 10^4].

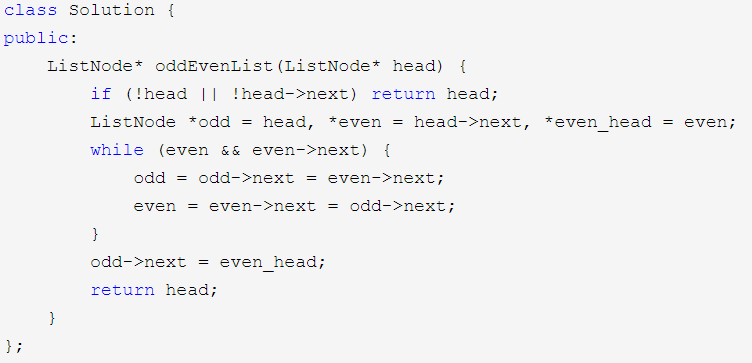
这道题给了我们一个链表，让我们分开奇偶节点，所有奇节点在前，偶节点在后。我们可以使用两个指针来做，pre指向奇节点，cur指向偶节点，然后把偶节点cur后面的那个奇节点提前到pre的后面，然后pre和cur各自前进一步，此时cur又指向偶节点，pre指向当前奇节点的末尾，以此类推直至把所有的偶节点都提前了即可，参见代码如下：

解法一：



还有一种解法，用两个奇偶指针分别指向奇偶节点的起始位置，另外需要一个单独的指针even\_head来保存偶节点的起点位置，然后把奇节点的指向偶节点的下一个(一定是奇节点)，此奇节点后移一步，再把偶节点指向下一个奇节点的下一个(一定是偶节点)，此偶节点后移一步，以此类推直至末尾，此时把分开的偶节点的链表连在奇节点的链表后即可，参见代码如下；

解法二：



19. Remove Nth Node From End of List

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

Example:

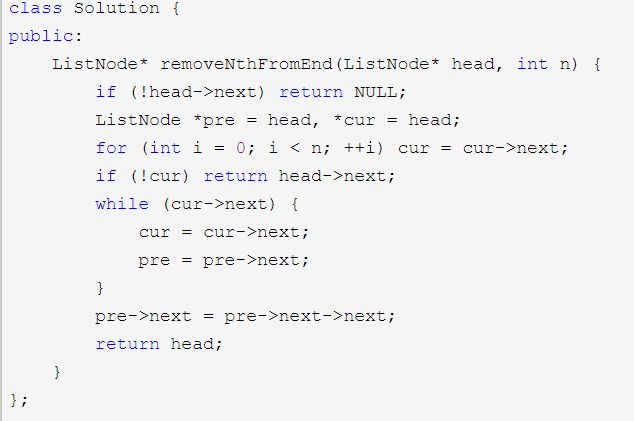
Given linked list: 1->2->3->4->5, and *n* = 2.

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

Note: Given *n* will always be valid.

Follow up: Could you do this in one pass?

这道题让我们移除链表倒数第N个节点，限定n一定是有效的，即n不会大于链表中的元素总数。还有题目要求一次遍历解决问题，那么就得想些比较巧妙的方法了。比如首先要考虑的时，如何找到倒数第N个节点，由于只允许一次遍历，所以不能用一次完整的遍历来统计链表中元素的个数，而是遍历到对应位置就应该移除了。那么就需要用两个指针来帮助解题，pre 和 cur 指针。首先 cur 指针先向前走N步，如果此时 cur 指向空，说明N为链表的长度，则需要移除的为首元素，那么此时返回 head->next 即可，如果 cur 存在，再继续往下走，此时 pre 指针也跟着走，直到 cur 为最后一个元素时停止，此时 pre 指向要移除元素的前一个元素，再修改指针跳过需要移除的元素即可，参见代码如下：



148. Sort List

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

Example 1:

Input: 4->2->1->3

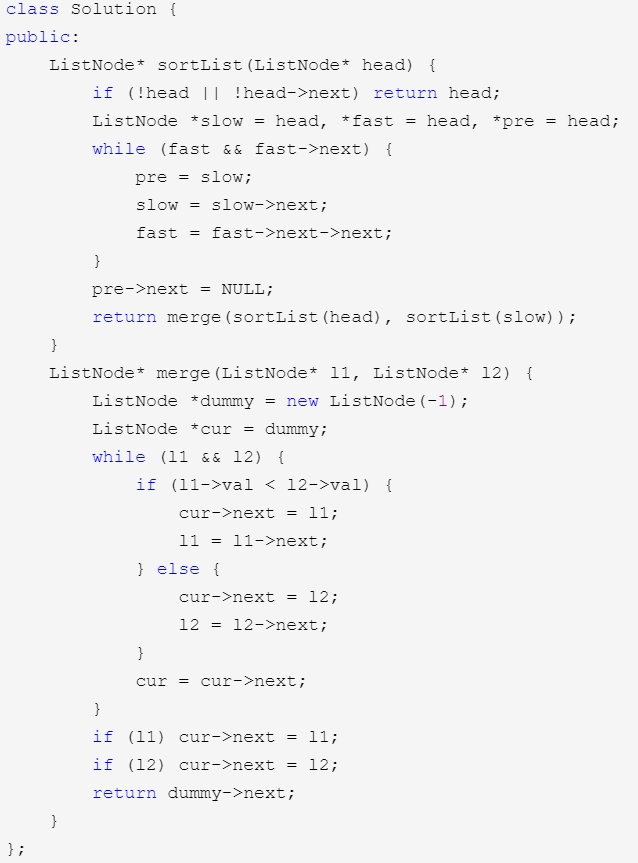
Output: 1->2->3->4

Example 2:

Input: -1->5->3->4->0

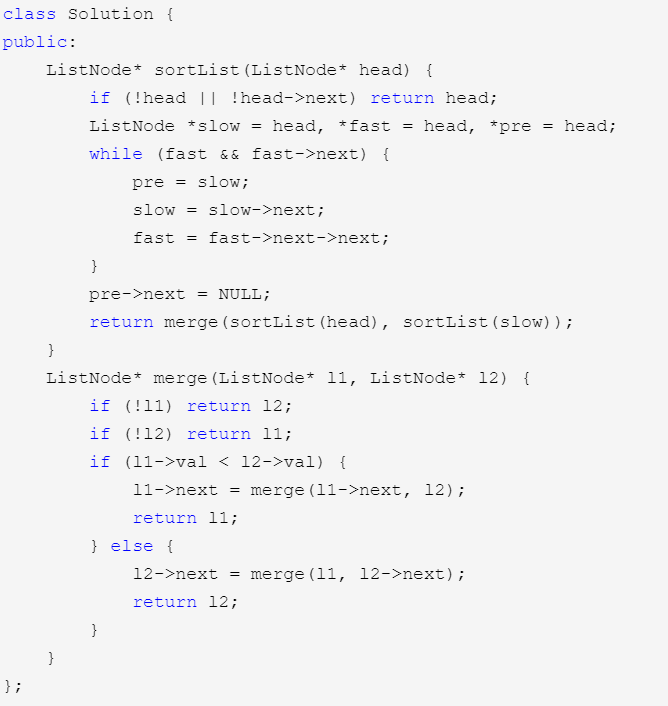
Output: -1->0->3->4->5

常见排序方法有很多，插入排序，选择排序，堆排序，快速排序，冒泡排序，归并排序，桶排序等等。。它们的时间复杂度不尽相同，而这里题目限定了时间必须为O(nlgn)，符合要求只有快速排序，归并排序，堆排序，而根据单链表的特点，最适于用归并排序。为啥呢？这是由于链表自身的特点决定的，由于不能通过坐标来直接访问元素，所以快排什么的可能不太容易实现（但是被评论区的大神们打脸，还是可以实现的），堆排序的话，如果让新建结点的话，还是可以考虑的，若只能交换结点，最好还是不要用。而归并排序（又称混合排序）因其可以利用递归来交换数字，天然适合链表这种结构。归并排序的核心是一个 merge() 函数，其主要是合并两个有序链表，这个在 LeetCode 中也有单独的题目 [Merge Two Sorted Lists](http://www.cnblogs.com/grandyang/p/4086297.html)。由于两个链表是要有序的才能比较容易 merge，那么对于一个无序的链表，如何才能拆分成有序的两个链表呢？我们从简单来想，什么时候两个链表一定都是有序的？就是当两个链表各只有一个结点的时候，一定是有序的。而归并排序的核心其实是分治法 Divide and Conquer，就是将链表从中间断开，分成两部分，左右两边再分别调用排序的递归函数 sortList()，得到各自有序的链表后，再进行 merge()，这样整体就是有序的了。因为子链表的递归函数中还是会再次拆成两半，当拆到链表只有一个结点时，无法继续拆分了，而这正好满足了前面所说的“一个结点的时候一定是有序的”，这样就可以进行 merge 了。然后再回溯回去，每次得到的都是有序的链表，然后进行 merge，直到还原整个长度。这里将链表从中间断开的方法，采用的就是快慢指针，大家可能对快慢指针找链表中的环比较熟悉，其实找链表中的中点同样好使，因为快指针每次走两步，慢指针每次走一步，当快指针到达链表末尾时，慢指针正好走到中间位置，参见代码如下：



下面这种方法也是归并排序，而且在merge函数中也使用了递归，这样使代码更加简洁啦～

C++ 解法二：



226. Invert Binary Tree

Invert a binary tree.

Example:

Input:



Output:



Trivia:

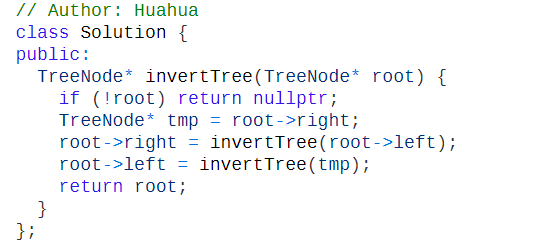
This problem was inspired by [this original tweet](https://twitter.com/mxcl/status/608682016205344768) by [Max Howell](https://twitter.com/mxcl):

**Google: 90% of our engineers use the software you wrote (Homebrew), but you can’t invert a binary tree on a whiteboard so f\*\*\* off.**

## Solution: Recursion

Recursive invert the left and right subtrees and swap them.

Time complexity: O(n) Space complexity: O(h)

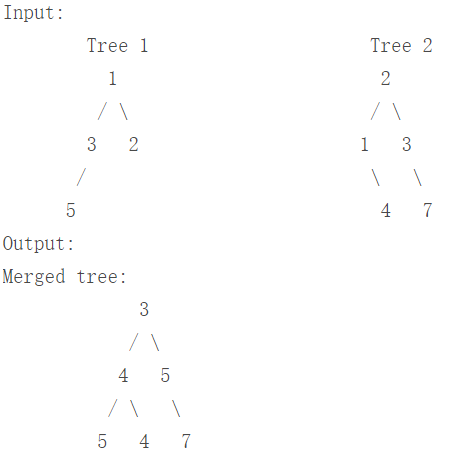


617. Merge Two Binary Trees

Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

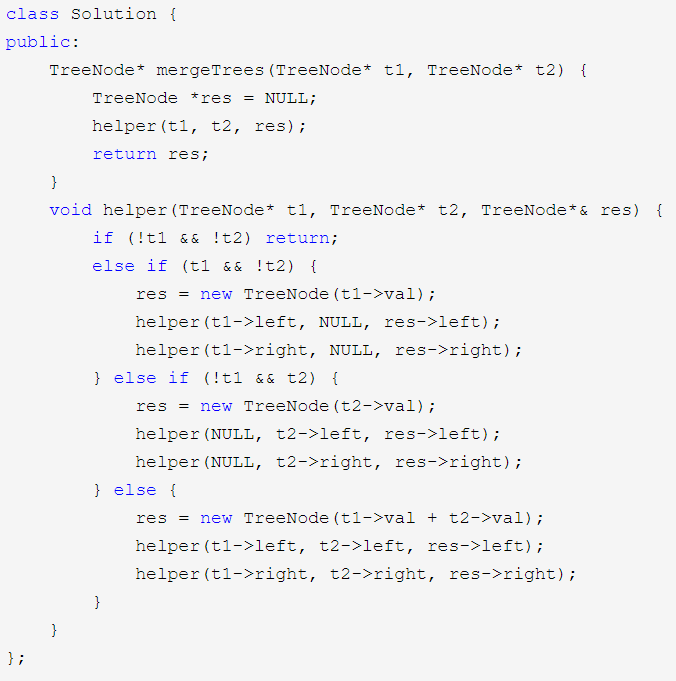
Example 1:



Note: The merging process must start from the root nodes of both trees.

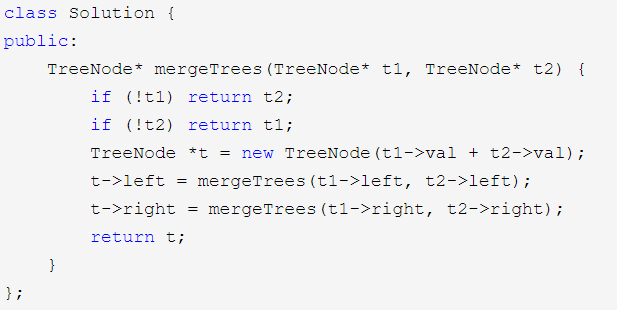
这道题给了两个二叉树，让我们合并成一个，规则是，都存在的结点，就将结点值加起来，否则空的位置就由另一个树的结点来代替。那么根据过往经验，处理二叉树问题的神器就是递归。根据题目中的规则，如果要处理的相同位置上的两个结点都不存在的话，直接返回即可，如果 t1 存在，t2 不存在，就以 t1 的结点值建立一个新结点，然后分别对 t1 的左右子结点和空结点调用递归函数，反之，如果 t1 不存在，t2 存在，就以 t2 的结点值建立一个新结点，然后分别对 t2 的左右子结点和空结点调用递归函数。如果 t1 和 t2 都存在，就以 t1 和 t2 的结点值之和建立一个新结点，然后分别对 t1 的左右子结点和 t2 的左右子结点调用递归函数，参见代码如下：

解法一：



其实远不用写的像上面那么复杂，连额外的函数都不用写，直接递归调用给定的函数即可，首先判断，如果 t1 不存在，则直接返回 t2，反之，如果 t2 不存在，则直接返回 t1。如果上面两种情况都不满足，那么以 t1 和 t2 的结点值之和建立新结点t，然后对 t1 和 t2 的左子结点调用递归并赋给t的左子结点，再对 t1 和 t2 的右子结点调用递归并赋给t的右子结点，返回t结点即可，参见代码如下：

解法二：



572. Subtree of Another Tree

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

Example 1:

Given tree s:

3

/ \

4 5

/ \

1 2

Given tree t:

4

/ \

1 2

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

Example 2:

Given tree s:

3

/ \

4 5

/ \

1 2

/

0

Given tree t:

4

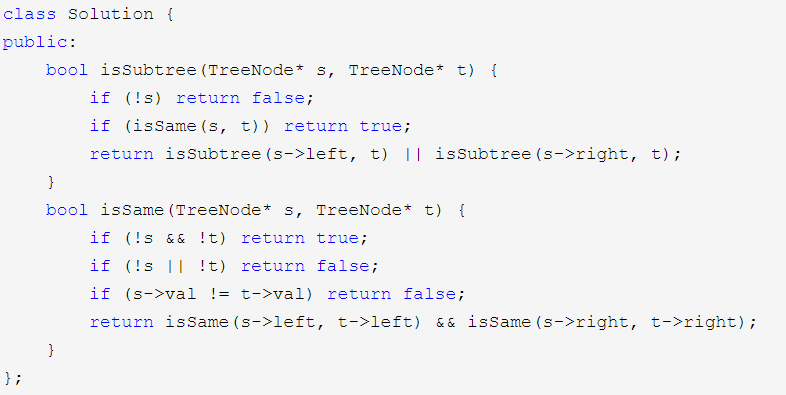
/ \

1 2

Return false.

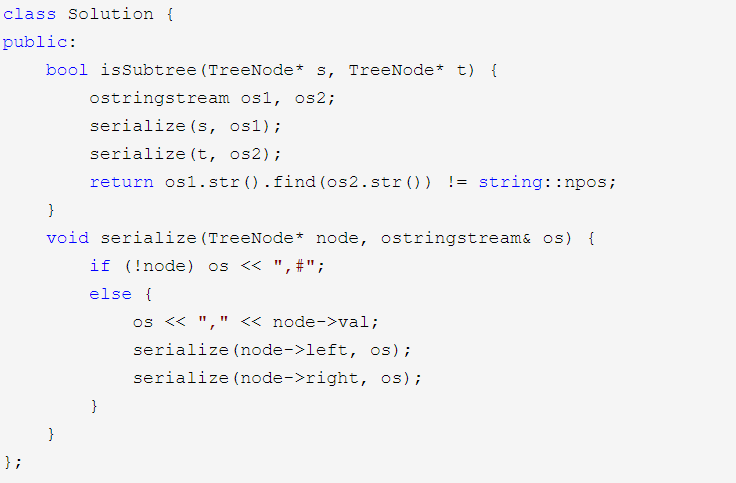
这道题让我们求一个数是否是另一个树的子树，从题目中的第二个例子中可以看出，子树必须是从叶结点开始的，中间某个部分的不能算是子树，那么我们转换一下思路，是不是从s的某个结点开始，跟t的所有结构都一样，那么问题就转换成了判断两棵树是否相同，也就是[Same Tree](http://www.cnblogs.com/grandyang/p/4053384.html)的问题了，这点想通了其实代码就很好写了，用递归来写十分的简洁，我们先从s的根结点开始，跟t比较，如果两棵树完全相同，那么返回true，否则就分别对s的左子结点和右子结点调用递归再次来判断是否相同，只要有一个返回true了，就表示可以找得到。

解法一：



下面这道题的解法用到了之前那道[Serialize and Deserialize Binary Tree](http://www.cnblogs.com/grandyang/p/4913869.html)的解法，思路是对s和t两棵树分别进行序列化，各生成一个字符串，如果t的字符串是s的子串的话，就说明t是s的子树，但是需要注意的是，为了避免出现[12], [2], 这种情况，虽然2也是12的子串，但是[2]却不是[12]的子树，所以我们再序列化的时候要特殊处理一下，就是在每个结点值前面都加上一个字符，比如','，来分隔开，那么[12]序列化后就是",12,#"，而[2]序列化之后就是",2,#"，这样就可以完美的解决之前的问题了，参见代码如下：

解法二：



404. Sum of Left Leaves

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

Example:

3

/ \

9 20

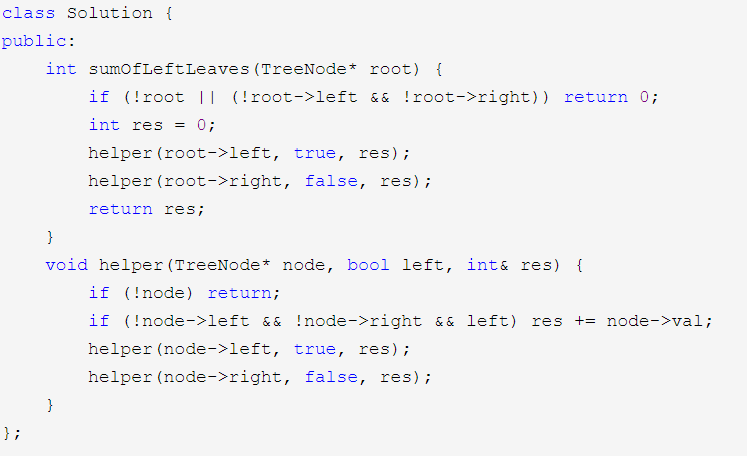
/ \

15 7

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

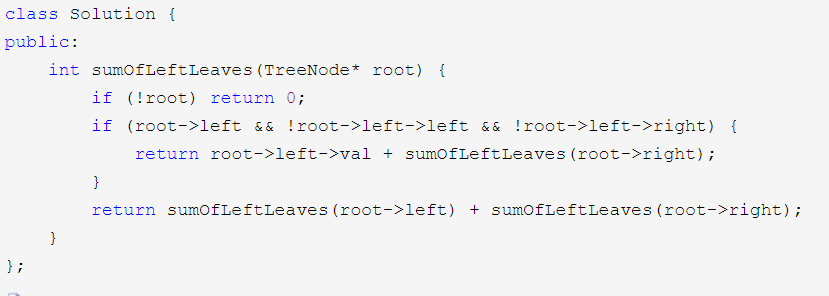
这道题让我们求一棵二叉树的所有左子叶的和，那么看到这道题我们知道这肯定是考二叉树的遍历问题，那么最简洁的写法肯定是用递归，由于我们只需要累加左子叶之和，那么我们在进入递归函数的时候需要知道当前结点是否是左子节点，如果是左子节点，而且该左子节点再没有子节点了说明其是左子叶，那么我们将其值加入结果res中，我们用一个bool型的变量，如果为true说明当前结点是左子节点，若为false则说明是右子节点，不做特殊处理，整个来说就是个递归的先序遍历的写法，参见代码如下：

解法一：



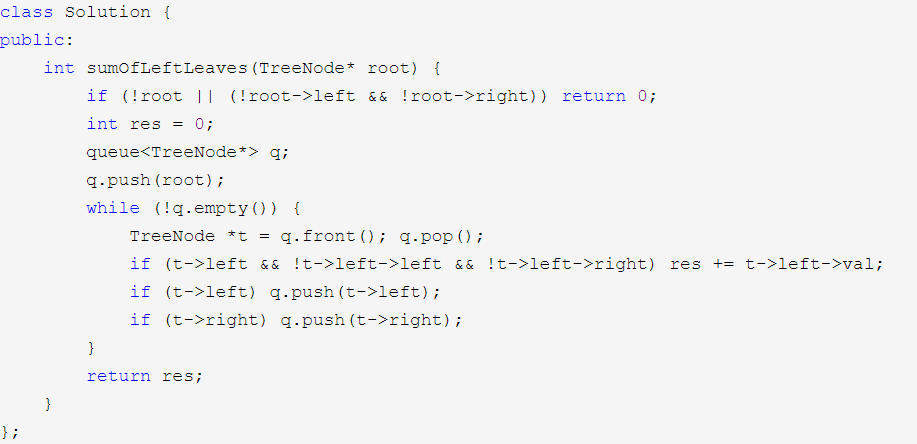
我们还可以写的更简洁一些，不需要写其他的函数，直接在原函数中检查当前节点的左子节点是否是左子叶，如果是的话，则返回左子叶的值加上对当前结点的右子节点调用递归的结果；如果不是的话，我们对左右子节点分别调用递归函数，返回二者之和，参见代码如下：

解法二：



我们也可以使用迭代来解，因为这道题的本质是遍历二叉树，所以我们可以用层序遍历的迭代写法，利用queue来辅助，注意对左子叶的判断和处理，参见代码如下：

解法三：



513. Find Bottom Left Tree Value

Given a binary tree, find the leftmost value in the last row of the tree.

Example 1:

Input:

2

/ \

1 3

Output:

1

Example 2:

Input:

1

/ \

2 3

/ / \

4 5 6

/

7

Output:

7

Note: You may assume the tree (i.e., the given root node) is not NULL.

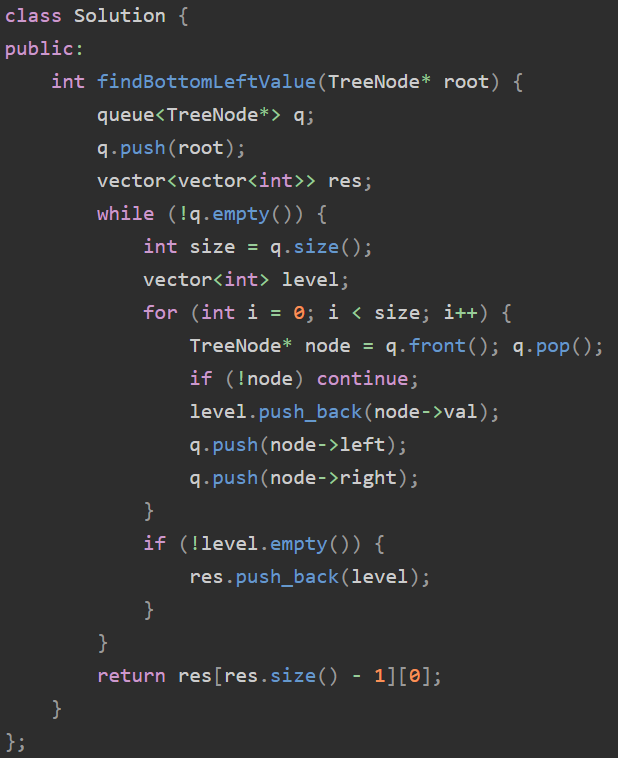
题目大意

求一个二叉树最下面一层的最左边节点。

解题方法

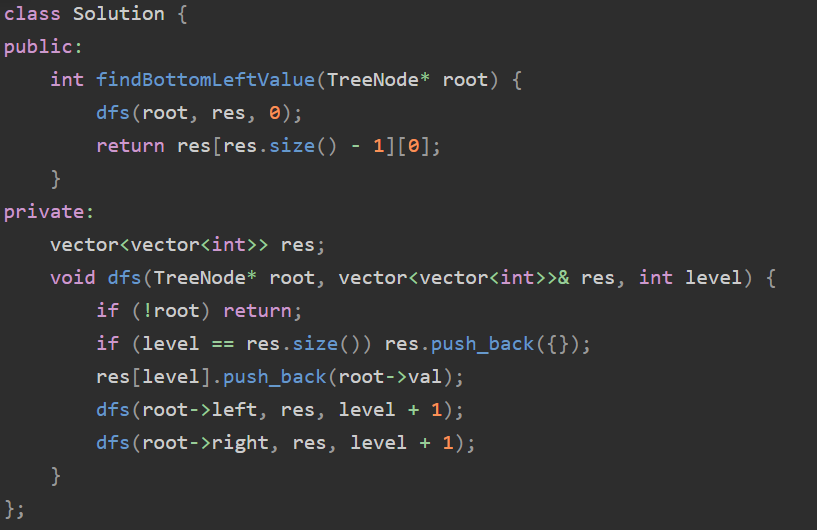
BFS

这就是所谓的BFS算法。广度优先搜索，但是搜索的顺序是有要求的，因为题目要最底层的叶子节点的最左边的叶子，那么进入队列的顺序就是先右节点再左节点，这样能把每层的节点都能从右到左过一遍，那么用一个int保存最后的节点值就可以了。



DFS

使用DFS很简单了，直接把每一层的元素放到list里面，然后取出最后层的第一个节点即可。至于子树的遍历顺序，需要保证先遍历左子树再遍历右子树，这样才能把最下面一层的最左边节点放到最左侧，至于根节点的值在哪个位置进行append是不重要的。



538. Convert BST to Greater Tree

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

**Example:**

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

5

/ \

2 13

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

18

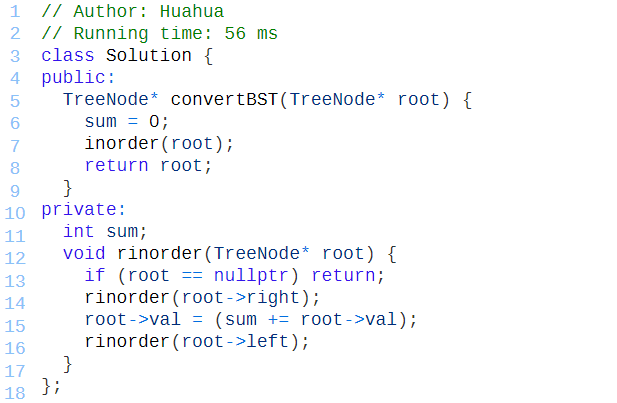
/ \

20 13

# Solution: reversed inorder traversal

in a BST, we can visit every node in the decreasing order. Using a member sum to track the sum of all visited nodes.

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

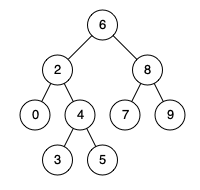


235. Lowest Common Ancestor of a Binary Search Tree

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

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

Given binary search tree: root = [6,2,8,0,4,7,9,null,null,3,5]



Example 1:

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

Output: 6

Explanation: The LCA of nodes 2 and 8 is 6.

Example 2:

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

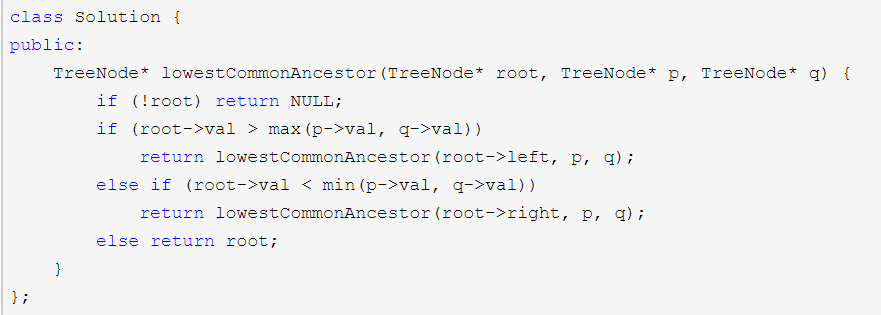
Output: 2

Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

Note:

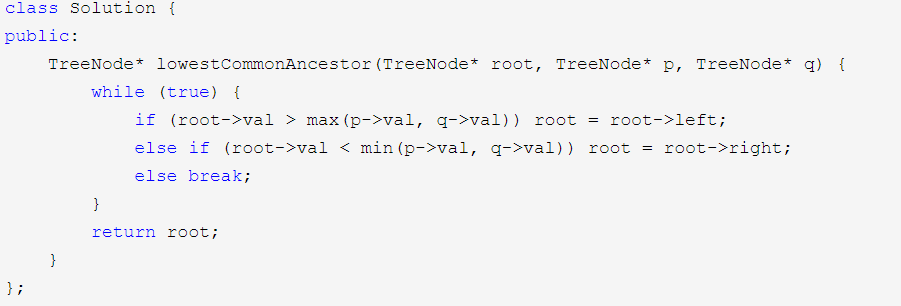
* All of the nodes' values will be unique.
* p and q are different and both values will exist in the BST.

这道题让我们求二叉搜索树的最小共同父节点, LeetCode中关于BST的题有 [Validate Binary Search Tree](http://www.cnblogs.com/grandyang/p/4298435.html)， [Recover Binary Search Tree](http://www.cnblogs.com/grandyang/p/4298069.html)， [Binary Search Tree Iterator](http://www.cnblogs.com/grandyang/p/4231455.html)， [Unique Binary Search Trees](http://www.cnblogs.com/grandyang/p/4299608.html)， [Unique Binary Search Trees II](http://www.cnblogs.com/grandyang/p/4301096.html)，[Convert Sorted Array to Binary Search Tree](http://www.cnblogs.com/grandyang/p/4295245.html) , [Convert Sorted List to Binary Search Tree](http://www.cnblogs.com/grandyang/p/4295618.html) 和 [Kth Smallest Element in a BST](http://www.cnblogs.com/grandyang/p/4620012.html)。这道题我们可以用递归来求解，我们首先来看题目中给的例子，由于二叉搜索树的特点是左<根<右，所以根节点的值一直都是中间值，大于左子树的所有节点值，小于右子树的所有节点值，那么我们可以做如下的判断，如果根节点的值大于p和q之间的较大值，说明p和q都在左子树中，那么此时我们就进入根节点的左子节点继续递归，如果根节点小于p和q之间的较小值，说明p和q都在右子树中，那么此时我们就进入根节点的右子节点继续递归，如果都不是，则说明当前根节点就是最小共同父节点，直接返回即可，参见代码如下：



当然，此题也有非递归的写法，用个 while 循环来代替递归调用即可，然后不停的更新当前的根节点，也能实现同样的效果，代码如下：

解法二：



530. Minimum Absolute Difference in BST

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

Example:

Input:

1

\

3

/

2

Output:

1

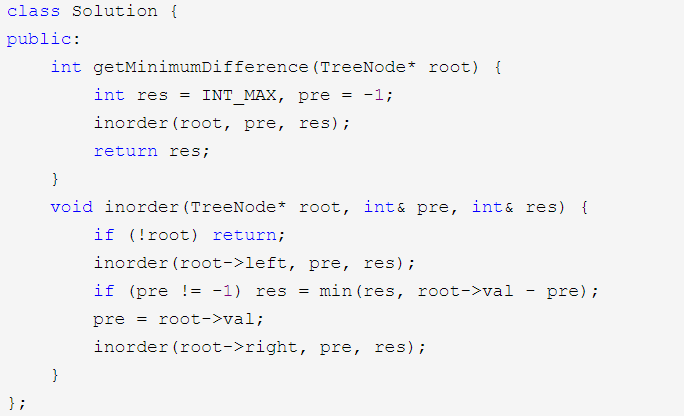
Explanation:

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

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

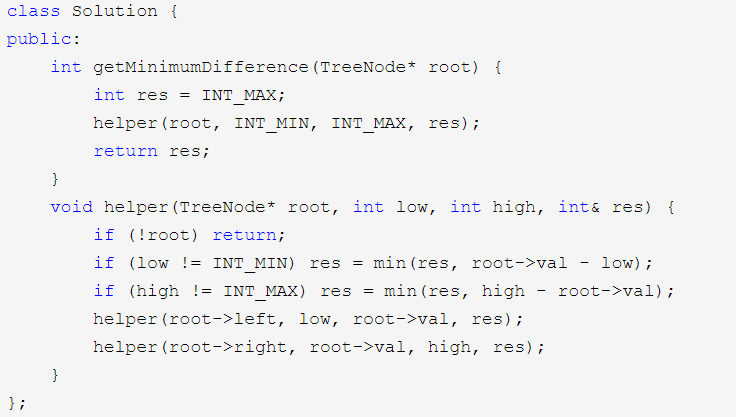
这道题给了我们一棵二叉搜索树，让我们求任意个节点值之间的最小绝对差。由于BST的左<根<右的性质可知，如果按照中序遍历会得到一个有序数组，那么最小绝对差肯定在相邻的两个节点值之间产生。所以我们的做法就是对BST进行中序遍历，然后当前节点值和之前节点值求绝对差并更新结果res。这里需要注意的就是在处理第一个节点值时，由于其没有前节点，所以不能求绝对差。这里我们用变量pre来表示前节点值，这里由于题目中说明了所以节点值不为负数，所以我们给pre初始化-1，这样我们就知道pre是否存在。如果没有题目中的这个非负条件，那么就不能用int变量来，必须要用指针，通过来判断是否为指向空来判断前结点是否存在。还好这里简化了问题，用-1就能搞定了，这里我们先来看中序遍历的递归写法，参见代码如下：

解法一：



其实我们也不必非要用中序遍历不可，用先序遍历同样可以利用到BST的性质，我们带两个变量low和high来分别表示上下界，初始化为int的极值，然后我们在递归函数中，分别用上下界和当前节点值的绝对差来更新结果res，参见代码如下：

解法二：



889. Construct Binary Tree from Preorder and Postorder Traversal

Return any binary tree that matches the given preorder and postorder traversals.

Values in the traversals pre and post are distinct positive integers.

Example 1:

Input: pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]

Output: [1,2,3,4,5,6,7]

Note:

* 1 <= pre.length == post.length <= 30
* pre[] and post[] are both permutations of 1, 2, ..., pre.length.
* It is guaranteed an answer exists. If there exists multiple answers, you can return any of them.

这道题给了一棵树的先序遍历和后序遍历的数组，让我们根据这两个数组来重建出原来的二叉树。之前也做过二叉树的先序遍历 [Binary Tree Preorder Traversal](http://www.cnblogs.com/grandyang/p/4146981.html) 和 后序遍历 [Binary Tree Postorder Traversal](http://www.cnblogs.com/grandyang/p/4251757.html)，所以应该对其遍历的顺序并不陌生。其实二叉树最常用的三种遍历方式，先序，中序，和后序遍历，只要知道其中的任意两种遍历得到的数组，就可以重建出原始的二叉树，而且正好都在 LeetCode 中有出现，其他两道分别是 [Construct Binary Tree from Inorder and Postorder Traversal](https://www.cnblogs.com/grandyang/p/4296193.html) 和 [Construct Binary Tree from Preorder and Inorder Traversal](https://www.cnblogs.com/grandyang/p/4296500.html)。如果做过之前两道题，那么这道题就没有什么难度了，若没有的话，可能还是有些 tricky 的，虽然这仅仅只是一道 Medium 的题。

我们知道，先序遍历的顺序是 根->左->右，而后序遍历的顺序是 左->右->根，既然要建立树，那么肯定要从根结点开始创建，然后再创建左右子结点，若你做过很多树相关的题目的话，就会知道大多数都是用递归才做，那么创建的时候也是对左右子结点调用递归来创建。心中有这么个概念就好，可以继续来找这个重复的 pattern。由于先序和后序各自的特点，根结点的位置是固定的，既是先序遍历数组的第一个，又是后序遍历数组的最后一个，而如果给我们的是中序遍历的数组，那么根结点的位置就只能从另一个先序或者后序的数组中来找了，但中序也有中序的好处，其根结点正好分割了左右子树，就不在这里细讲了，还是回到本题吧。知道了根结点的位置后，我们需要分隔左右子树的区间，先序和后序的各个区间表示如下：

preorder -> [root] [left subtree] [right subtree]

postorder -> [left subtree] [right substree] [root]

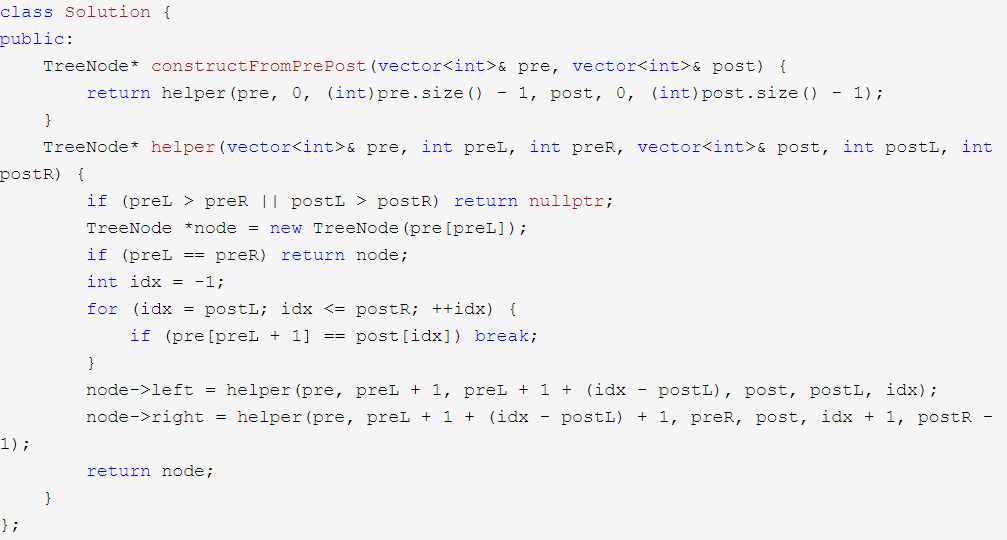
具体到题目中的例子就是：

preorder -> [1] [2,4,5] [3,6,7]

postorder -> [4,5,2] [6,7,3] [root]

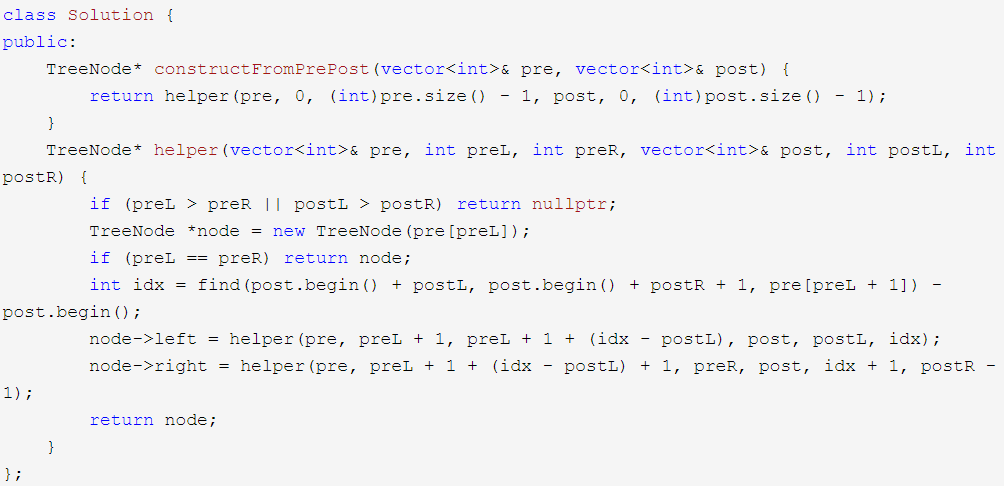
先序和后序中各自的左子树区间的长度肯定是相等的，但是其数字顺序可能是不同的，但是我们仔细观察的话，可以发现先序左子树区间的第一个数字2，在后序左右子树区间的最后一个位置，而且这个规律对右子树区间同样适用，这是为啥呢，这就要回到各自遍历的顺序了，先序遍历的顺序是 根->左->右，而后序遍历的顺序是 左->右->根，其实这个2就是左子树的根结点，当然会一个在开头，一个在末尾了。发现了这个规律，就可以根据其来定位左右子树区间的位置范围了。既然要拆分数组，那么就有两种方式，一种是真的拆分成小的子数组，另一种是用双指针来指向子区间的开头和末尾。前一种方法无疑会有大量的数组拷贝，不是很高效，所以我们这里采用第二种方法来做。用 preL 和 preR 分别表示左子树区间的开头和结尾位置，postL 和 postR 表示右子树区间的开头和结尾位置，那么若 preL 大于 preR 或者 postL 大于 postR 的时候，说明已经不存在子树区间，直接返回空指针。然后要先新建当前树的根结点，就通过 pre[preL] 取到即可，接下来要找左子树的根结点在 post 中的位置，最简单的方法就是遍历 post 中的区间 [postL, postR]，找到其位置 idx，然后根据这个 idx，就可以算出左子树区间长度为 len = (idx-postL)+1，那么 pre 数组中左子树区间为 [preL+1, preL+len]，右子树区间为 [preL+1+len, preR]，同理，post 数组中左子树区间为 [postL, idx]，右子树区间为 [idx+1, postR-1]。知道了这些信息，就可以分别调用递归函数了，参见代码如下：

解法一：



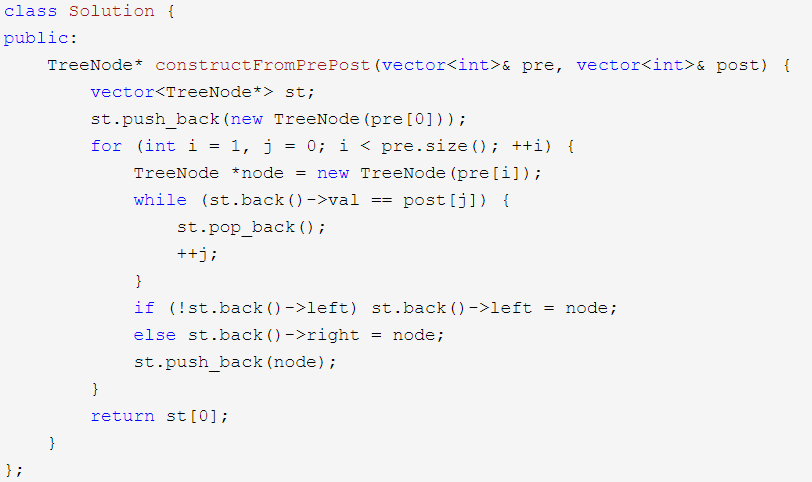
我们也可以使用 STL 内置的 find() 函数来查找左子树的根结点在 post 中的位置，其余的地方都跟上面的解法相同，参见代码如下：

解法二：



为了进一步优化时间复杂度，我们可以事先用一个 HashMap，来建立 post 数组中每个元素和其坐标之间的映射，这样在递归函数中，就不用进行查找了，直接在 HashMap 中将其位置取出来用即可，用空间换时间，也不失为一个好的方法，参见代码如下：

解法三：



106. Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

**Note:** You may assume that duplicates do not exist in the tree.

For example, given

inorder = [9,3,15,20,7]

postorder = [9,15,7,20,3]

Return the following binary tree:

3

/ \

9 20

/ \

15 7

这道题要求从中序和后序遍历的结果来重建原二叉树，我们知道中序的遍历顺序是左-根-右，后序的顺序是左-右-根，对于这种树的重建一般都是采用递归来做，可参见我之前的一篇博客[Convert Sorted Array to Binary Search Tree 将有序数组转为二叉搜索树](http://www.cnblogs.com/grandyang/p/4295245.html)。针对这道题，由于后序的顺序的最后一个肯定是根，所以原二叉树的根节点可以知道，题目中给了一个很关键的条件就是树中没有相同元素，有了这个条件我们就可以在中序遍历中也定位出根节点的位置，并以根节点的位置将中序遍历拆分为左右两个部分，分别对其递归调用原函数。代码如下：

上述代码中需要小心的地方就是递归是postorder的左右index很容易写错，比如 pLeft + i - iLeft - 1, 这个又长又不好记，首先我们要记住 i - iLeft 是计算inorder中根节点位置和左边起始点的距离，然后再加上postorder左边起始点然后再减1。我们可以这样分析，如果根节点就是左边起始点的话，那么拆分的话左边序列应该为空集，此时i - iLeft 为0， pLeft + 0 - 1 < pLeft, 那么再递归调用时就会返回NULL, 成立。如果根节点是左边起始点紧跟的一个，那么i - iLeft 为1， pLeft + 1 - 1 = pLeft，再递归调用时还会生成一个节点，就是pLeft位置上的节点，为原二叉树的一个叶节点。

我们下面来看一个例子, 某一二叉树的中序和后序遍历分别为：

Inorder:　　 　11　　4　　5　　13　　8　　9

Postorder:　　11　　4　　13　　9　　8　　5

11　　4　　5　　13　　8　　9　　　　　　=>　　　　　　　　　 5

11　　4　　13　　9　　8　　5　　　　　　　　　　　　　　　　/　　\

11　　4　　 　　13　　 8　　9　　　　　　=>　　　　　　　　　5

11　　4　　　　 13　　9　　8　　 　　　　　　　　　　　　　 /　　\

　　　　　　　　　　　　　　　　　　　　　　　　　　　　　4　　　8

11　　　　 　　13　　　　9　　　　　　　　=>　　　　　　　　　5

11　　　　　　 13　　　　9　　　　 　　　　　　　　　　　　 /　　\

　　　　　　　　　　　　　　　　　　　　　　　　　　　　　4　　　8

　　　　　　　　　　　　　　　　　　　　　　　　　　　　/　　　 / \

　　　　　　　　　　　　　　　　　　　　　　　　　　　11　　 13　　 9

94. Binary Tree Inorder Traversal

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

Example:

Input: [1,null,2,3]

1

\

2

/

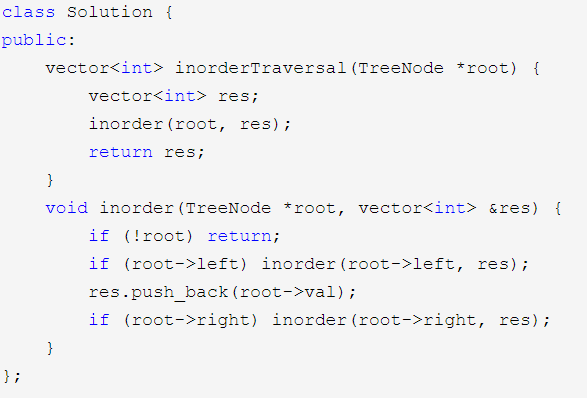
3

Output: [1,3,2]

Follow up: Recursive solution is trivial, could you do it iteratively?

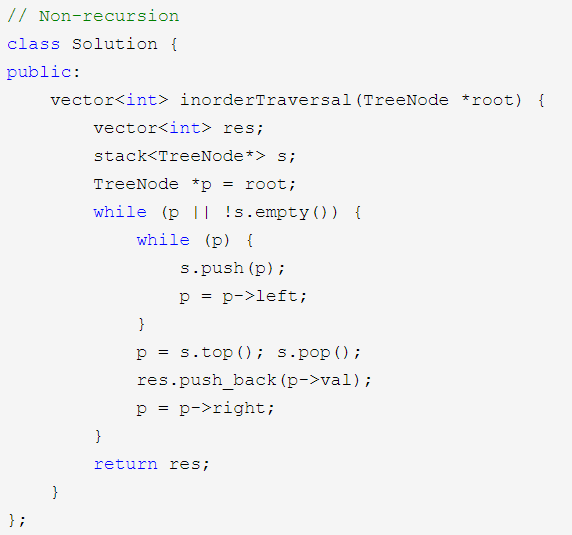
二叉树的中序遍历顺序为左-根-右，可以有递归和非递归来解，其中非递归解法又分为两种，一种是使用栈来接，另一种不需要使用栈。我们先来看递归方法，十分直接，对左子结点调用递归函数，根节点访问值，右子节点再调用递归函数，代码如下：

解法一：



下面再来看非递归使用栈的解法，也是符合本题要求使用的解法之一，需要用栈来做，思路是从根节点开始，先将根节点压入栈，然后再将其所有左子结点压入栈，然后取出栈顶节点，保存节点值，再将当前指针移到其右子节点上，若存在右子节点，则在下次循环时又可将其所有左子结点压入栈中。这样就保证了访问顺序为左-根-右，代码如下：

解法二：



下面来看另一种很巧妙的解法，这种方法不需要使用栈，所以空间复杂度为常量，这种非递归不用栈的遍历方法有个专门的名字，叫 Morris Traversal，在介绍这种方法之前，先来引入一种新型树，叫 [Threaded binary tree](http://en.wikipedia.org/wiki/Threaded_binary_tree)，这个还不太好翻译，第一眼看上去以为是叫线程二叉树，但是感觉好像又跟线程没啥关系，后来看到网上有人翻译为螺纹二叉树，但博主认为这翻译也不太敢直视，很容易让人联想到为计划生育做出突出贡献的某世界著名品牌，后经热心网友提醒，应该叫做线索二叉树。先来看看维基百科上关于它的英文定义：

A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node (**if** it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.

就是说线索二叉树实际上是把所有原本为空的右子节点指向了中序遍历顺序之后的那个节点，把所有原本为空的左子节点都指向了中序遍历之前的那个节点，具体例子可以点击[这里](http://en.wikipedia.org/wiki/Threaded_binary_tree)。那么这道题跟这个线索二叉树又有啥关系呢？由于既不能用递归，又不能用栈，那如何保证访问顺序是中序遍历的左-根-右呢。原来需要构建一个线索二叉树，需要将所有为空的右子节点指向中序遍历的下一个节点，这样中序遍历完左子结点后，就能顺利的回到其根节点继续遍历了。具体算法如下：

1. 初始化指针 cur 指向 root

2. 当 cur 不为空时

　 - 如果 cur 没有左子结点

　 a) 打印出 cur 的值

　　 b) 将 cur 指针指向其右子节点

　 - 反之

　 将 pre 指针指向 cur 的左子树中的最右子节点

　　　 \* 若 pre 不存在右子节点

　　　 a) 将其右子节点指回 cur

　　　　 b) cur 指向其左子节点

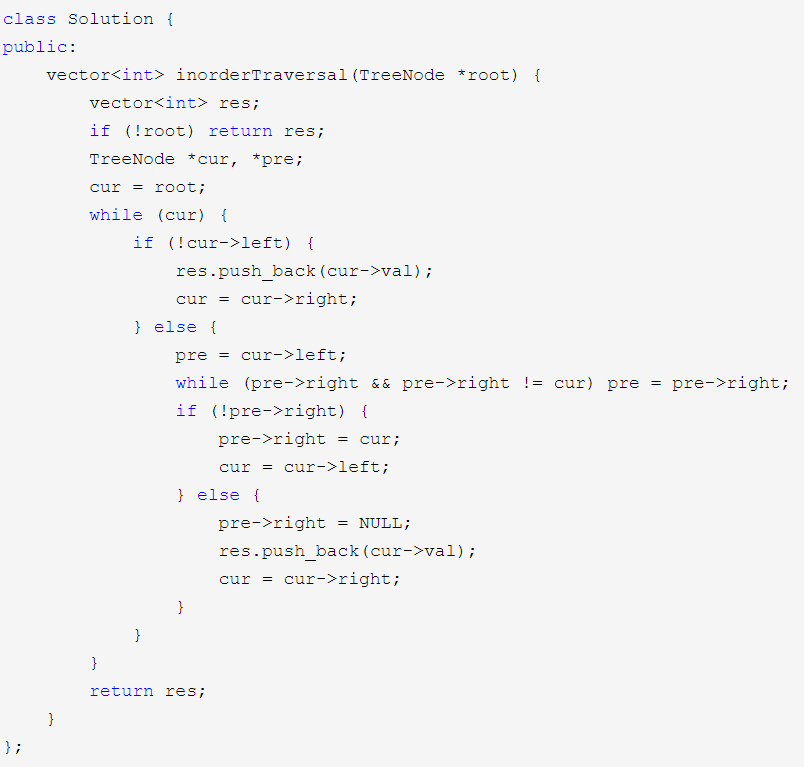
　　　 \* 反之

　　　　　 a) 将 pre 的右子节点置空

　　　　　 b) 打印 cur 的值

　　　　　 c) 将 cur 指针指向其右子节点

解法四：



145. Binary Tree Postorder Traversal

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

For example:

Given binary tree {1,#,2,3},

1

\

2

/

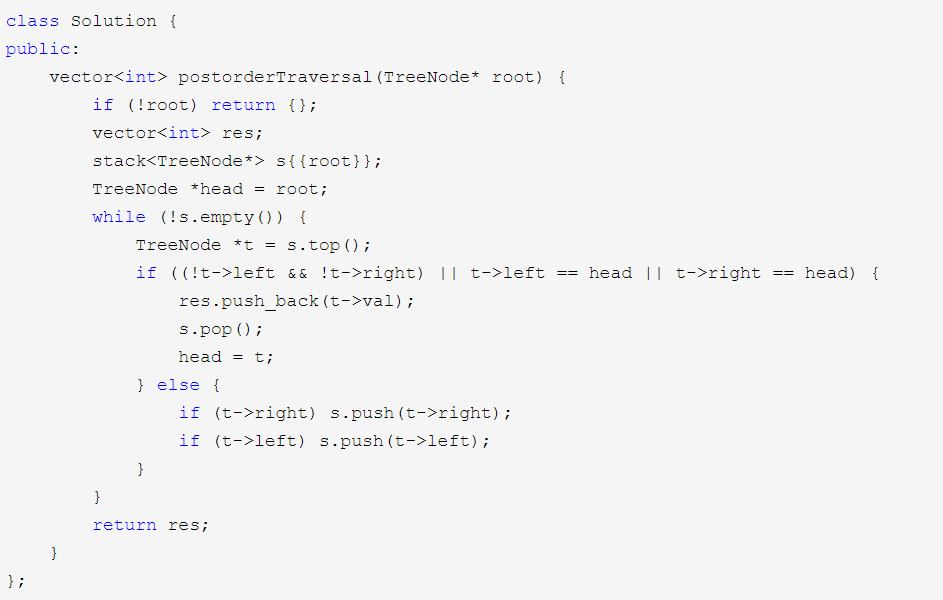
3

return [3,2,1].

**Note:** Recursive solution is trivial, could you do it iteratively?

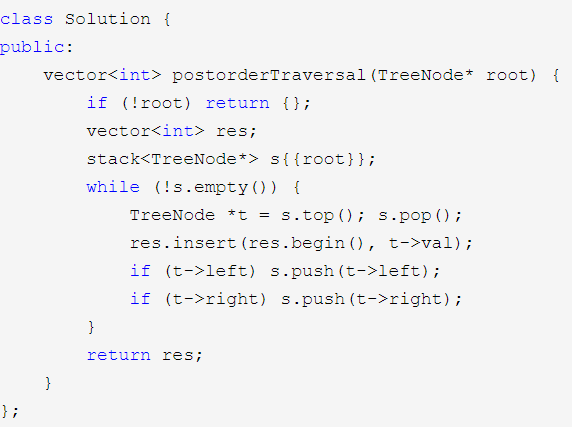
经典题目，求二叉树的后序遍历的非递归方法，跟前序，中序，层序一样都需要用到栈，后序的顺序是左-右-根，所以当一个结点值被取出来时，它的左右子结点要么不存在，要么已经被访问过了。先将根结点压入栈，然后定义一个辅助结点 head，while 循环的条件是栈不为空，在循环中，首先将栈顶结点t取出来，如果栈顶结点没有左右子结点，或者其左子结点是 head，或者其右子结点是 head 的情况下。将栈顶结点值加入结果 res 中，并将栈顶元素移出栈，然后将 head 指向栈顶元素；否则的话就看如果右子结点不为空，将其加入栈，再看左子结点不为空的话，就加入栈，注意这里先右后左的顺序是因为栈的后入先出的特点，可以使得左子结点先被处理。下面来看为什么是这三个条件呢，首先如果栈顶元素如果没有左右子结点的话，说明其是叶结点，而且入栈顺序保证了左子结点先被处理，所以此时的结点值就可以直接加入结果 res 了，然后移出栈，将 head 指向这个叶结点，这样的话 head 每次就是指向前一个处理过并且加入结果 res 的结点，那么如果栈顶结点的左子结点或者右子结点是 head 的话，说明其子结点已经加入结果 res 了，那么就可以处理当前结点了。

看到这里，大家可能对 head 的作用，以及为何要初始化为 root，还不是很清楚，这里再解释一下。head 是指向上一个被遍历完成的结点，由于后序遍历的顺序是左-右-根，所以一定会一直将结点压入栈，一直到把最左子结点（或是最左子结点的最右子结点）压入栈后，开始进行处理。一旦开始处理了，head 就会被重新赋值。所以 head 初始化值并没有太大的影响，唯一要注意的是不能初始化为空，因为在判断是否打印出当前结点时除了判断是否是叶结点，还要看 head 是否指向其左右子结点，如果 head 指向左子结点，那么右子结点一定为空，因为入栈顺序是根-右-左，不存在右子结点还没处理，就直接去处理根结点了的情况。若 head 指向右子结点，则是正常的左-右-根的处理顺序。那么回过头来在看，若 head 初始化为空，且此时正好左子结点不存在，那么在压入根结点时，head 和左子结点相等就成立了，此时就直接打印根结点了，明显是错的。所以 head 只要不初始化为空，一切都好说，甚至可以新建一个结点也没问题。将 head 初始化为 root，也可以，就算只有一个 root 结点，那么在判定叶结点时就将 root 打印了，然后就跳出 while 循环了，也不会出错。代码如下：



由于后序遍历的顺序是左-右-根，而先序遍历的顺序是根-左-右，二者其实还是很相近的，可以先在先序遍历的方法上做些小改动，使其遍历顺序变为根-右-左，然后翻转一下，就是左-右-根啦，翻转的方法我们使用反向Q，哦不，是反向加入结果 res，每次都在结果 res 的开头加入结点值，而改变先序遍历的顺序就只要该遍历一下入栈顺序，先左后右，这样出栈处理的时候就是先右后左啦，参见代码如下：

解法二：

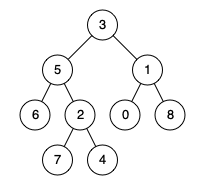


236. Lowest Common Ancestor of a Binary Tree

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](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Given the following binary tree: root = [3,5,1,6,2,0,8,null,null,7,4]



Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Note:

* All of the nodes' values will be unique.
* p and q are different and both values will exist in the binary tree.

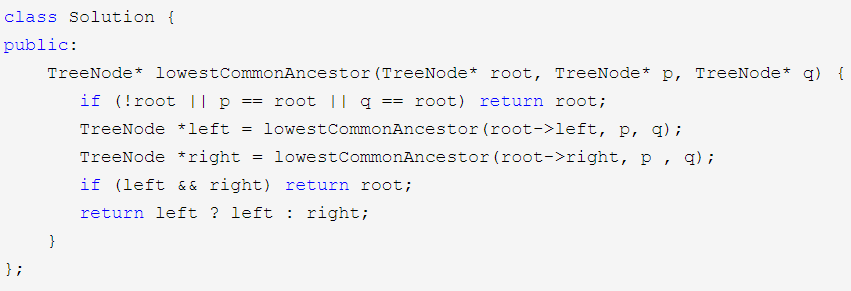
这道求二叉树的最小共同父节点的题是之前那道 [Lowest Common Ancestor of a Binary Search Tree](http://www.cnblogs.com/grandyang/p/4640572.html) 的 Follow Up。跟之前那题不同的地方是，这道题是普通是二叉树，不是二叉搜索树，所以就不能利用其特有的性质，我们只能在二叉树中来搜索p和q，然后从路径中找到最后一个相同的节点即为父节点，可以用递归来实现，在递归函数中，首先看当前结点是否为空，若为空则直接返回空，若为p或q中的任意一个，也直接返回当前结点。否则的话就对其左右子结点分别调用递归函数，由于这道题限制了p和q一定都在二叉树中存在，那么如果当前结点不等于p或q，p和q要么分别位于左右子树中，要么同时位于左子树，或者同时位于右子树，那么我们分别来讨论：

- 若p和q分别位于左右子树中，那么对左右子结点调用递归函数，会分别返回p和q结点的位置，而当前结点正好就是p和q的最小共同父结点，直接返回当前结点即可，这就是题目中的例子1的情况。

- 若p和q同时位于左子树，这里有两种情况，一种情况是 left 会返回p和q中较高的那个位置，而 right 会返回空，所以最终返回非空的 left 即可，这就是题目中的例子2的情况。还有一种情况是会返回p和q的最小父结点，就是说当前结点的左子树中的某个结点才是p和q的最小父结点，会被返回。

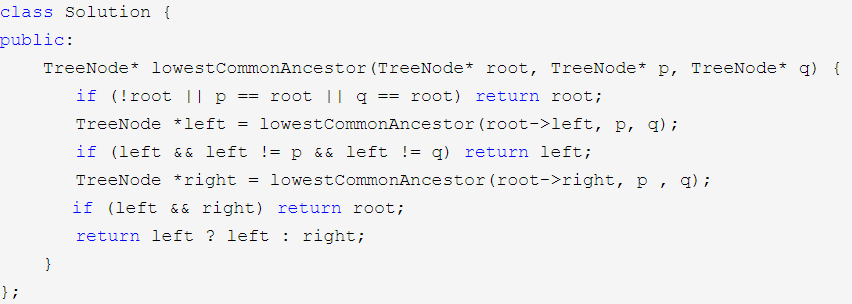
- 若p和q同时位于右子树，同样这里有两种情况，一种情况是 right 会返回p和q中较高的那个位置，而 left 会返回空，所以最终返回非空的 right 即可，还有一种情况是会返回p和q的最小父结点，就是说当前结点的右子树中的某个结点才是p和q的最小父结点，会被返回，写法很简洁，代码如下：

解法一：



上述代码可以进行优化一下，如果当前结点不为空，且既不是p也不是q，那么根据上面的分析，p和q的位置就有三种情况，p和q要么分别位于左右子树中，要么同时位于左子树，或者同时位于右子树。我们需要优化的情况就是当p和q同时为于左子树或右子树中，而且返回的结点并不是p或q，那么就是p和q的最小父结点了，已经求出来了，就不用再对右结点调用递归函数了，这是为啥呢？因为根本不会存在 left 既不是p也不是q，同时还有p或者q在 right 中。首先递归的第一句就限定了只要遇到了p或者q，就直接返回，之后又限定了只有当 left 和 right 同时存在的时候，才会返回当前结点，当前结点若不是p或q，则一定是最小父节点，否则 left 一定是p或者q。这里的逻辑比较绕，不太好想，多想想应该可以理清头绪吧，参见代码如下：

解法二：



109. Convert Sorted List to Binary Search Tree

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

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

Example:

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

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

0

/ \

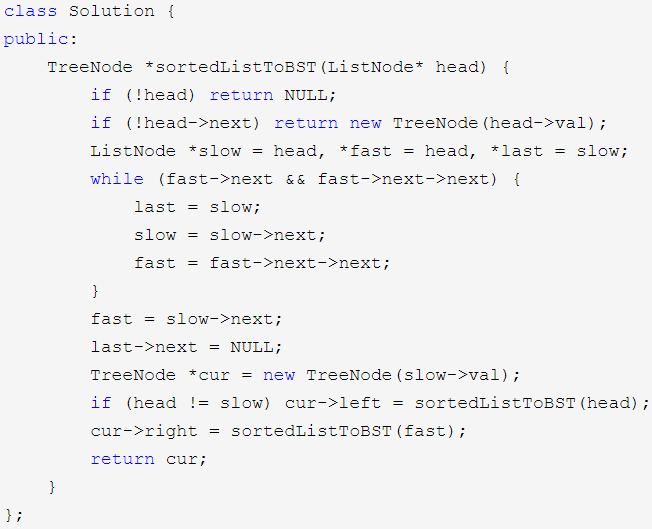
-3 9

/ /

-10 5

这道题是要求把有序链表转为二叉搜索树，和之前那道 [Convert Sorted Array to Binary Search Tree](http://www.cnblogs.com/grandyang/p/4295245.html) 思路完全一样，只不过是操作的数据类型有所差别，一个是数组，一个是链表。数组方便就方便在可以通过index直接访问任意一个元素，而链表不行。由于二分查找法每次需要找到中点，而链表的查找中间点可以通过快慢指针来操作，可参见之前的两篇博客 [Reorder List](http://www.cnblogs.com/grandyang/p/4254860.html) 和 [Linked List Cycle II](http://www.cnblogs.com/grandyang/p/4137302.html) 有关快慢指针的应用。找到中点后，要以中点的值建立一个数的根节点，然后需要把原链表断开，分为前后两个链表，都不能包含原中节点，然后再分别对这两个链表递归调用原函数，分别连上左右子节点即可。代码如下：

解法一：



我们也可以采用如下的递归方法，重写一个递归函数，有两个输入参数，子链表的起点和终点，因为知道了这两个点，链表的范围就可以确定了，而直接将中间部分转换为二叉搜索树即可，递归函数中的内容跟上面解法中的极其相似，参见代码如下：

解法二



897. Increasing Order Search Tree

Given a binary search tree, rearrange the tree in in-order so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only 1 right child.

Example 1:

Input: [5,3,6,2,4,null,8,1,null,null,null,7,9]

5

/ \

3 6

/ \ \

2 4 8

/ / \

1 7 9

Output: [1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]

1

\

2

\

3

\

4

\

5

\

6

\

7

\

8

\

9

**Note:**

1. The number of nodes in the given tree will be between 1 and 100.
2. Each node will have a unique integer value from 0 to 1000.

# **Solution: In-order traversal**

root = 5

inorder(root.left) 之后

self.prev = 4

（1-4）已经处理完了，这时候的树是很奇怪的一个形状，3即是2的右子树，又是5的左子树。

1

\

2 5

\ / \

3 6

\ \

**prev ->** **4** 8

/ \

7 9

—————————

5.left = None # 把5->3的链接断开

5

\

6

\

8

/ \

7 9

—————————–

self.prev.right = root <=> 4.right = 5

把5接到4的右子树

1

\

2

\

3

\

4

\

**5 <– prev**

\

6

\

8

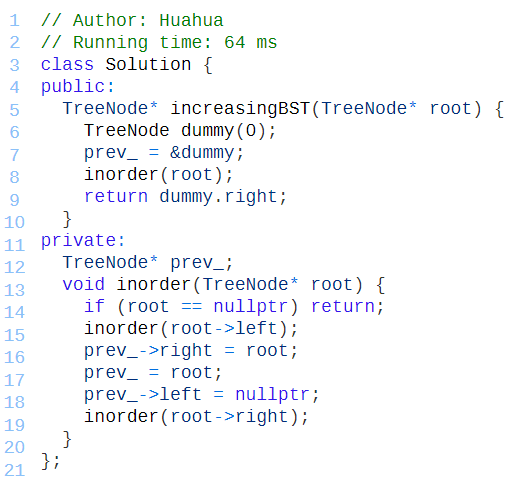
/ \

7 9

self.prev = root <=> prev = 5

inorder(5.right) <=> inorder(6) 然后再去递归处理6（及其子树）即可。

Time complexity: O(n) Space complexity: O(n)



653. Two Sum IV - Input is a BST

Given a Binary Search Tree and a target number, return true if there exist two elements in the BST such that their sum is equal to the given target.

Example 1:

Input:

5

/ \

3 6

/ \ \

2 4 7

Target = 9

Output: True

Example 2:

Input:

5

/ \

3 6

/ \ \

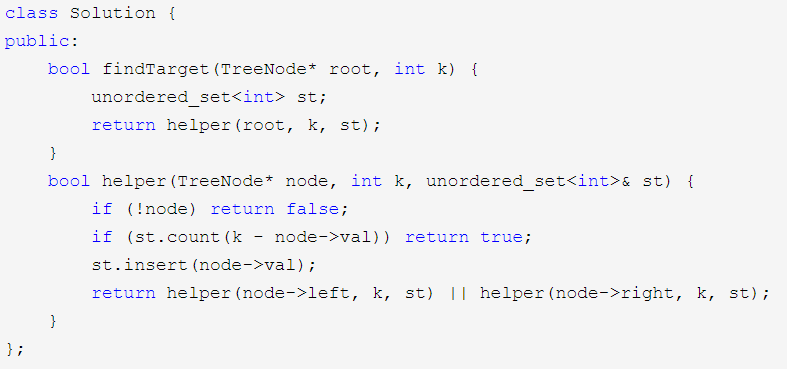
2 4 7

Target = 28

Output: False

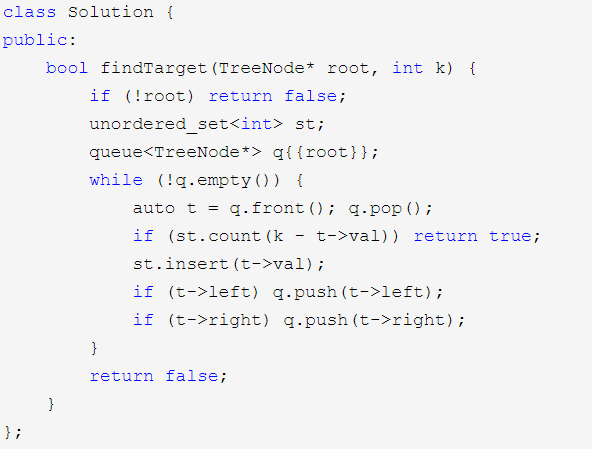
这道题又是一道2sum的变种题，博主一直强调，平生不识TwoSum，刷尽LeetCode也枉然！只要是两数之和的题，一定要记得先尝试用HashSet来做，这道题只不过是把数组变成了一棵二叉树而已，换汤不换药，我们遍历二叉树就行，然后用一个HashSet，在递归函数函数中，如果node为空，返回false。如果k减去当前结点值在HashSet中存在，直接返回true；否则就将当前结点值加入HashSet，然后对左右子结点分别调用递归函数并且或起来返回即可，参见代码如下：

解法一：



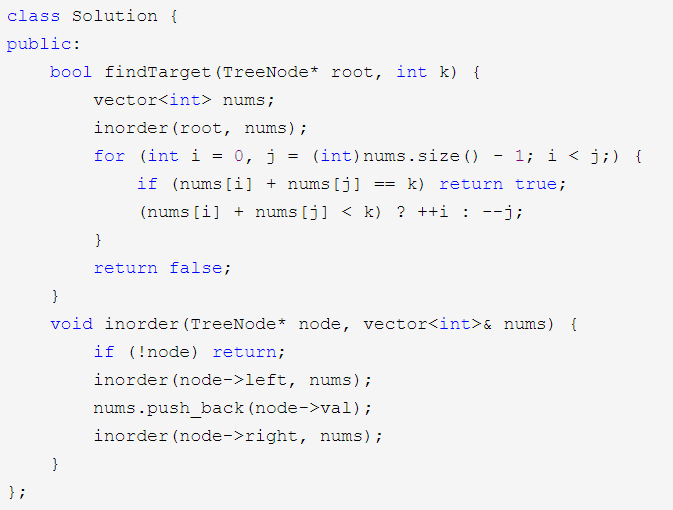
我们也可以用层序遍历来做，这样就是迭代的写法了，但是利用HashSet的精髓还是没变的，参见代码如下：

解法二：



由于输入是一棵二叉搜索树，那么我们可以先用中序遍历得到一个有序数组，然后在有序数组中找两数之和就很简单了，直接用双指针进行遍历即可，参见代码如下：

解法三：



450. Delete Node in a BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

Note: Time complexity should be O(height of tree).

Example:

root = [5,3,6,2,4,null,7]

key = 3

5

/ \

3 6

/ \ \

2 4 7

Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the following BST.

5

/ \

4 6

/ \

2 7

Another valid answer is [5,2,6,null,4,null,7].

5

/ \

2 6

\ \

4 7

这道题让我们删除二叉搜索树中的一个节点，难点在于删除完结点并补上那个结点的位置后还应该是一棵二叉搜索树。被删除掉的结点位置，不一定是由其的左右子结点补上，比如下面这棵树：

7

/ \

4 8

/ \

2 6

\ /

3 5

如果要删除结点4，那么应该将结点5补到4的位置，这样才能保证还是 BST，那么结果是如下这棵树：

7

/ \

5 8

/ \

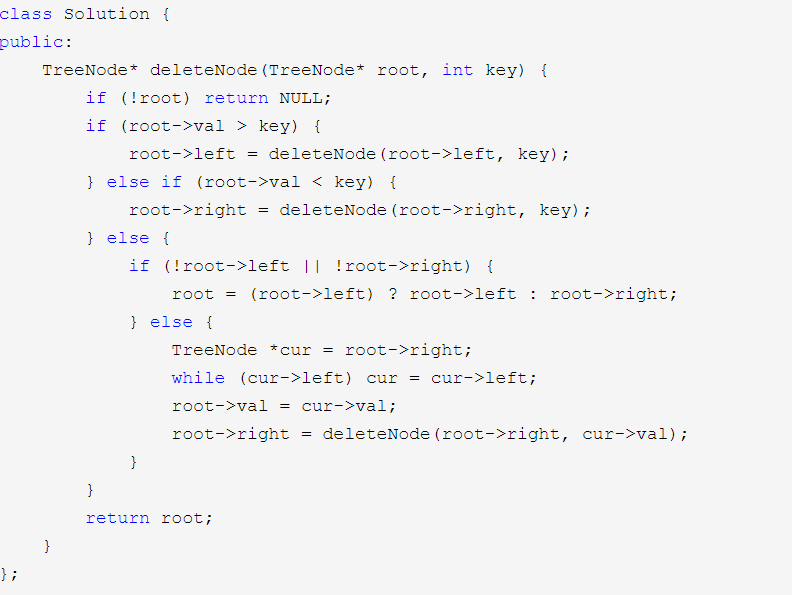
2 6

\

3

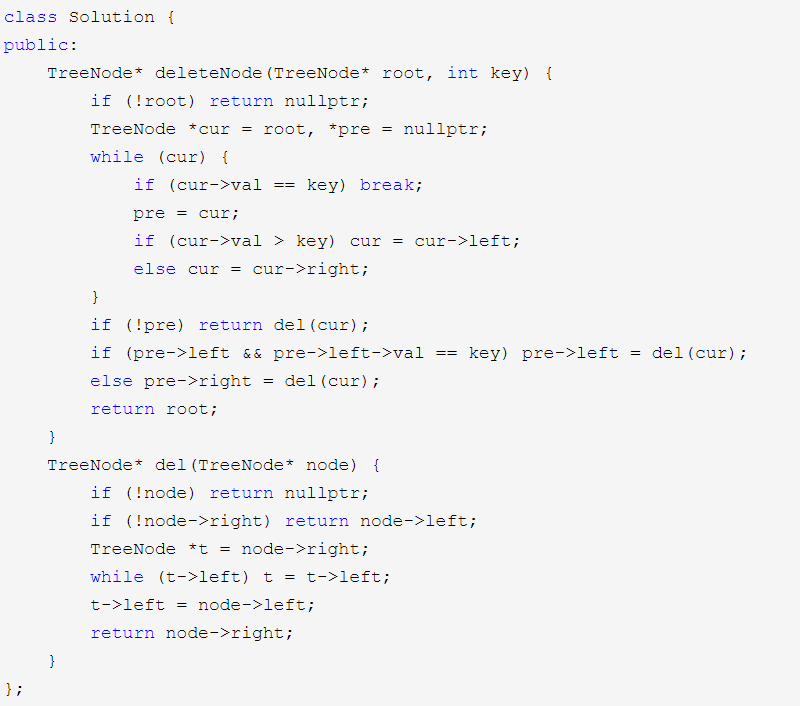
先来看一种递归的解法，首先判断根节点是否为空。由于 BST 的左<根<右的性质，使得可以快速定位到要删除的结点，对于当前结点值不等于 key 的情况，根据大小关系对其左右子结点分别调用递归函数。若当前结点就是要删除的结点，先判断若有一个子结点不存在，就将 root 指向另一个结点，如果左右子结点都不存在，那么 root 就赋值为空了，也正确。难点就在于处理左右子结点都存在的情况，需要在右子树找到最小值，即右子树中最左下方的结点，然后将该最小值赋值给 root，然后再在右子树中调用递归函数来删除这个值最小的结点，参见代码如下：

解法一：



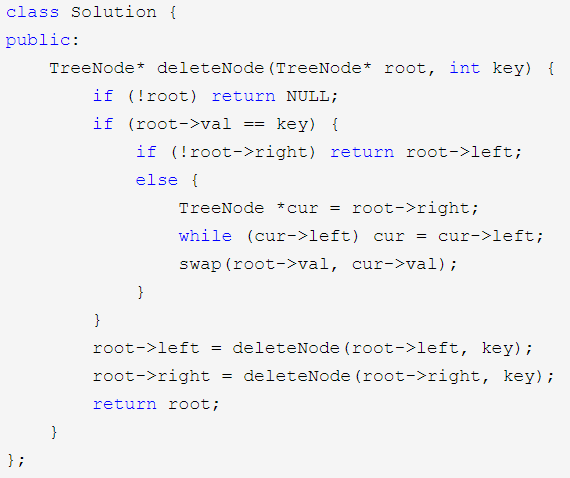
下面来看迭代的写法，还是通过 BST 的性质来快速定位要删除的结点，如果没找到直接返回空。遍历的过程要记录上一个位置的结点 pre，如果 pre 不存在，说明要删除的是根结点，如果要删除的结点在 pre 的左子树中，那么 pre 的左子结点连上删除后的结点，反之 pre 的右子结点连上删除后的结点。在删除函数中，首先判空，若为空，直接返回空指针；否则检测若右子结点不存在，直接返回左子结点即可，因为没有右子树就不会牵扯到调整树结构的问题；若右子结点存在，需要找到右子树中的最小值，即右子树中的最左子结点，用一个 while 循环找到即可，然后将要删除结点的左子结点连到右子树的最左子结点的左子结点上即可（说的有点绕，大家仔细体会一下），最后返回要删除结点的右子结点即可，文字表述确实比较绕，请大家自行带例子一步一步观察就会很清晰明了，参见代码如下：

解法二：



下面来看一种对于二叉树通用的解法，适用于所有二叉树，所以并没有利用 BST 的性质，而是遍历了所有的结点，然后删掉和 key 值相同的结点，参见代码如下：

解法三：



**Chapter 15 Graph**

1059. All Paths from Source Lead to Destination (Medium)

課金以解鎖題目

1135. Connecting Cities With Minimum Cost (Medium)

課金以解鎖題目

882. Reachable Nodes In Subdivided Graph (Hard)

Starting with an undirected graph (the "original graph") with nodes from `0` to `N-1`, subdivisions are made to some of the edges.

The graph is given as follows: edges[k] is a list of integer pairs (i, j, n) such that (i, j) is an edge of the original graph,

and n is the total number of new nodes on that edge.

Then, the edge (i, j) is deleted from the original graph, n new nodes (x\_1, x\_2, ..., x\_n) are added to the original graph,

and n+1 new edges (i, x\_1), (x\_1, x\_2), (x\_2, x\_3), ..., (x\_{n-1}, x\_n), (x\_n, j) are added to the original graph.

Now, you start at node 0 from the original graph, and in each move, you travel along one edge.

Return how many nodes you can reach in at most Mmoves.

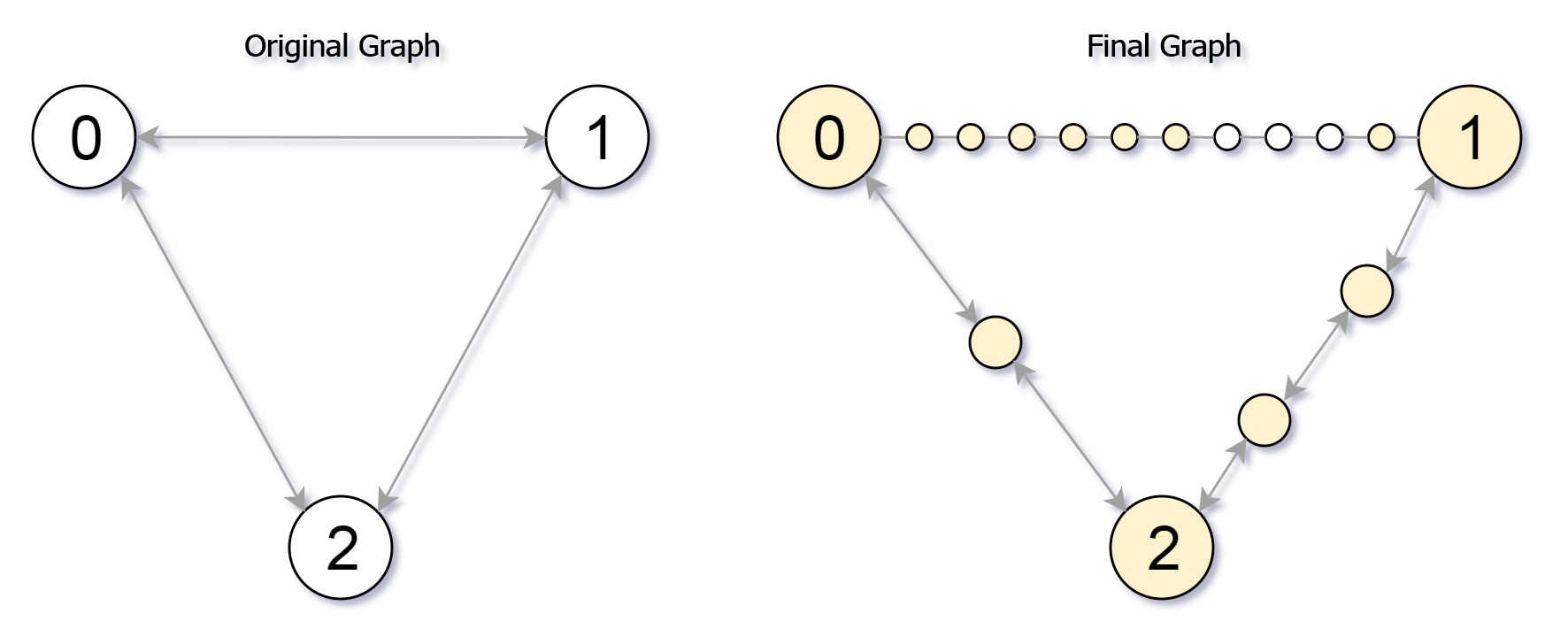
Example 1:

Input: `edges` = [[0,1,10],[0,2,1],[1,2,2]], M = 6, N = 3

Output: 13

Explanation:

The nodes that are reachable in the final graph after M = 6 moves are indicated below.



Example 2:

Input: `edges` = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], M = 10, N = 4

Output: 23

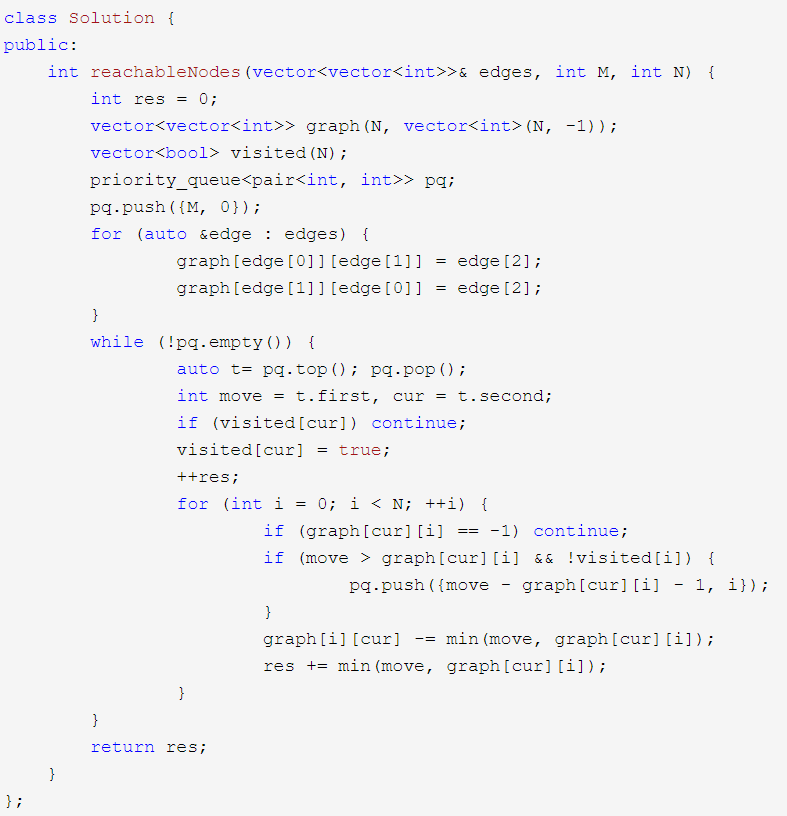
Note:

1. 0 <= edges.length <= 10000
2. 0 <= edges[i][0] < edges[i][1] < N
3. There does not exist any i != j for which edges[i][0] == edges[j][0] and edges[i][1] == edges[j][1].
4. The original graph has no parallel edges.
5. 0 <= edges[i][2] <= 10000
6. 0 <= M <= 10^9
7. 1 <= N <= 3000
8. A reachable node is a node that can be travelled to using at most M moves starting from node 0.

这道题给了我们一个无向图，里面有N个结点，但是每两个结点中间可能有多个不同的结点，假设每到达下一个相邻的结点需要消耗一步，现在我们有M步可以走，问我们在M步内最多可以到达多少个不同的结点。这里虽然有N个有编号的大结点，中间还有若干个没有编号的小结点，但是最后在统计的时候不分大小结点，全都算不同的结点。为了更好的理解这道题，实际上可以把N个有编号的结点当作N个大城市，比如省会城市，每两个省会城市中间有多个小城市，假设我们每次坐飞机只能飞到相邻的下一个城市，现在我们最多能坐M次飞机，问从省会大城市0出发的话，最多能到达多少个城市。由于省会城市是大型中转站，所以只有在这里才能有多个选择去往不同的城市，而在两个省会城市中的每个小城市，只有前后两种选择，所以这道题实际上还是一种图的遍历，只不过不保证每次都能到有编号的结点，只有到达了有编号的结点，才可以继续遍历下去。当到达了有编号的结点时，还要计算此时的剩余步数，就是用前一个有编号结点的剩余步数，减去当前路径上的所有小结点的个数。假如当前的剩余步数不够到达下一个大结点时，此时我们要想办法标记出来我们走过了多少个小结点，不然下次我们通过另一条路径到达相同的下一个大结点时，再往回走就有可能重复统计小结点的个数。由于小结点并没有标号，没法直接标记，只能通过离最近的大结点的个数来标记，所以虽然这道题是一道无向图的题，但是我们需要将其当作有向图来处理，比如两个大结点A和B，中间有10个小结点，此时在A结点时只有6步能走，那么我们走了中间的6个结点，此时就要标记从B出发往A方向的话只有4个小结点能走了。

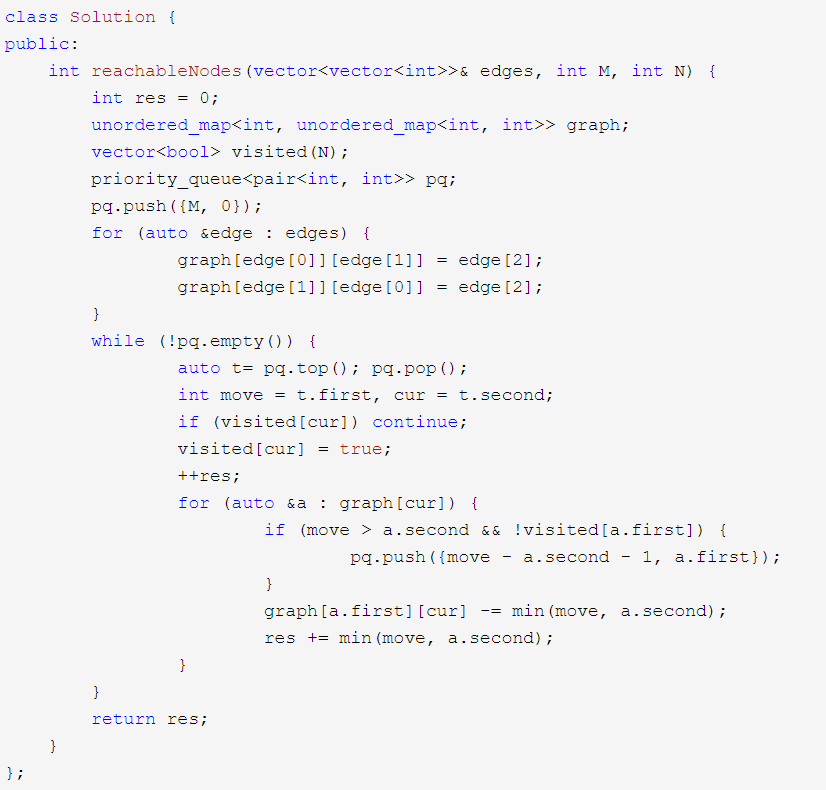
再进一步来分析，其实上对于每个结点来说（不论有没有编号），若我们能算出该结点离起始结点的最短距离，且该距离小于等于M的话，那这个结点就一定可以到达。这样来说，其实本质就是求单源点的最短距离，此时就要祭出神器迪杰斯特拉算法 Dijkstra Algorithm 了，LeetCode 中使用了该算法的题目还有 [Network Delay Time](https://www.cnblogs.com/grandyang/p/8278115.html) 和 [The Maze II](http://www.cnblogs.com/grandyang/p/6725380.html)。该算法的一般形式是用一个最小堆来保存到源点的最小距离，这里我们直接统计到源点的最小距离不是很方便，可以使用一个小 trick，即用一个最大堆来统计当前结点所剩的最大步数，因为剩的步数越多，说明距离源点距离越小。由于 Dijkstra 算法是以起点为中心，向外层层扩展，直到扩展到终点为止。根据这特性，用 BFS 来实现时再好不过了，首先来建立邻接链表，这里可以使用一个 NxN 的二维数组 graph，其中 graph[i][j] 表示从大结点i往大结点j方向会经过的小结点个数，建立邻接链表的时候对于每个 edge，要把两个方向都赋值，前面解释过了这里要当作有向图来做。然后使用一个最大堆，里面放剩余步数和结点编号组成的数对儿，把剩余步数放前面就可以默认按步数从大到小排序了，初始化时把 {M,0} 存入最大堆。还需要一个一维数组 visited 来记录某个结点是否访问过。在 while 循环中，首先取出堆顶元素数对儿，分别取出步数 move，和当前结点编号 cur，此时检查若该结点已经访问过了，直接跳过，否则就在 visited 数组中标记为 true。此时结果 res 自增1，因为当前大结点也是新遍历到的，需要累加个数。然后我们需要遍历所有跟 cur 相连的大结点，对于二维数组形式的邻接链表，我们只需要将i从0遍历到N，假如 graph[cur][i] 为 -1，表示结点 cur 和结点i不相连，直接跳过。否则相连的话，两个大结点中小结点的个数为 graph[cur][i]，此时要跟当前 cur 结点时剩余步数 move 比较，假如 move 较大，说明可以到达结点i，将此时到达结点i的剩余步数 move-graph[cur][i]-1（最后的减1是到达结点i需要的额外步数）和i一起组成数对儿，加入最大堆中。由于之前的分析，结点 cur 往结点i走过的所有结点，从结点i就不能再往结点 cur 走了，否则就累加了重复结点，所以 graph[i][cur] 要减去 move 和 graph[cur][i] 中的较小值，同时结果 res 要累加该较小值即可，参见代码如下：

解法一：



我们也可以使用 HashMap 来建立邻接链表，最后的运行速度果然要比二维数组形式的邻接链表要快一些，其他的地方都不变，参见代码如下：

解法二



**Chapter 16 Other Data Structures**

1135. Connecting Cities With Minimum Cost (Medium)

需課金

380. Insert Delete GetRandom O(1)

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

1. insert(val): Inserts an item val to the set if not already present.
2. remove(val): Removes an item val from the set if present.
3. getRandom: Returns a random element from current set of elements. Each element must have the same probability of being returned.

Example:

// Init an empty set.

RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomSet.insert(1);

// Returns false as 2 does not exist in the set.

randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].

randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.

randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].

randomSet.remove(1);

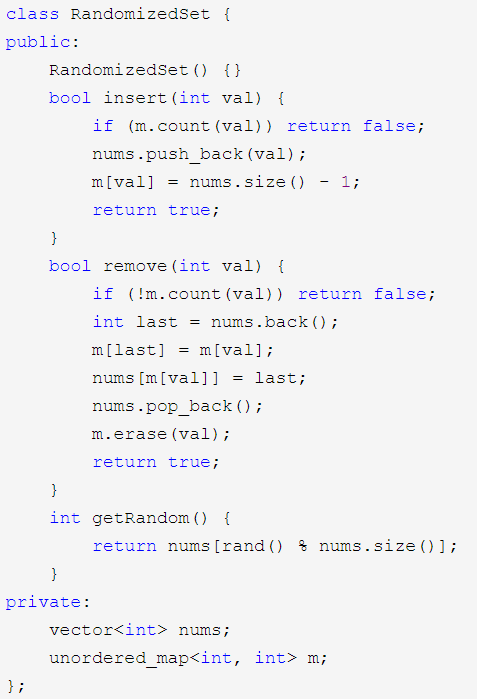
// 2 was already in the set, so return false.

randomSet.insert(2);

// Since 1 is the only number in the set, getRandom always return 1.

randomSet.getRandom();

这道题让我们在常数时间范围内实现插入删除和获得随机数操作，如果这道题没有常数时间的限制，那么将会是一道非常简单的题，直接用一个 HashSet 就可以搞定所有的操作。但是由于时间的限制，无法在常数时间内实现获取随机数，所以只能另辟蹊径。此题的正确解法是利用到了一个一维数组和一个 HashMap，其中数组用来保存数字，HashMap 用来建立每个数字和其在数组中的位置之间的映射，对于插入操作，先看这个数字是否已经在 HashMap 中存在，如果存在的话直接返回 false，不存在的话，将其插入到数组的末尾，然后建立数字和其位置的映射。删除操作是比较 tricky 的，还是要先判断其是否在 HashMap 里，如果没有，直接返回 false。由于 HashMap 的删除是常数时间的，而数组并不是，为了使数组删除也能常数级，实际上将要删除的数字和数组的最后一个数字调换个位置，然后修改对应的 HashMap 中的值，这样只需要删除数组的最后一个元素即可，保证了常数时间内的删除。而返回随机数对于数组来说就很简单了，只要随机生成一个位置，返回该位置上的数字即可，参见代码如下：



432. All O`one Data Structure

Implement a data structure supporting the following operations:

1. Inc(Key) - Inserts a new key with value 1. Or increments an existing key by 1. Key is guaranteed to be a non-empty string.
2. Dec(Key) - If Key's value is 1, remove it from the data structure. Otherwise decrements an existing key by 1. If the key does not exist, this function does nothing. Key is guaranteed to be a non-empty string.
3. GetMaxKey() - Returns one of the keys with maximal value. If no element exists, return an empty string "".
4. GetMinKey() - Returns one of the keys with minimal value. If no element exists, return an empty string "".

Challenge: Perform all these in O(1) time complexity.

这道题让我们实现一个全是O(1)复杂度的数据结构，包括了增加key，减少key，获取最大key，获取最小key，这几个函数。由于需要常数级的时间复杂度，我们首先第一反应就是要用哈希表来做，不仅如此，我们肯定还需要用list来保存所有的key，那么哈希表就是建立key和list中位置迭代器之间的映射，这不由得令人想到了之前那道[LRU Cache](http://www.cnblogs.com/grandyang/p/4587511.html)，也是用了类似的方法来解，但是感觉此题还要更加复杂一些。由于每个key还要对应一个次数，所以list中不能只放key，而且相同的次数可能会对应多个key值，所以我们用unordered\_set来保存次数相同的所有key值，我们建立一个Bucket的结构体来保存次数val，和保存key值的集合keys。解题思路主要参考了[网友ivancjw的帖子](https://discuss.leetcode.com/topic/63683/0ms-all-in-o-1-with-detailed-explantation)，数据结构参考了[史蒂芬大神的帖子](https://discuss.leetcode.com/topic/63827/c-solution-with-comments)，思路是，我们建立一个次数分层的结构，次数多的在顶层，每一层放相同次数的key值，例如下面这个例子：

"A": 4, "B": 4, "C": 2, "D": 1

那么用我们设计的结构保存出来就是：

row0: val = 4, keys = {"A", "B"}

row1: val = 2, keys = {"C"}

row2: val = 1, keys = {"D"}

好，我们现在来分析如何实现inc函数，我们来想，如果我们插入一个新的key，跟我们插入一个已经存在的key，情况是完全不一样的，那么我们就需要分情况来讨论:

- 如果我们插入一个新的key，那么由于该key没有出现过，所以加入后次数一定为1，那么就有两种情况了，如果list中没有val为1的这一行，那么我们需要插入该行，如果已经有了val为1的这行，我们直接将key加入集合keys中即可。

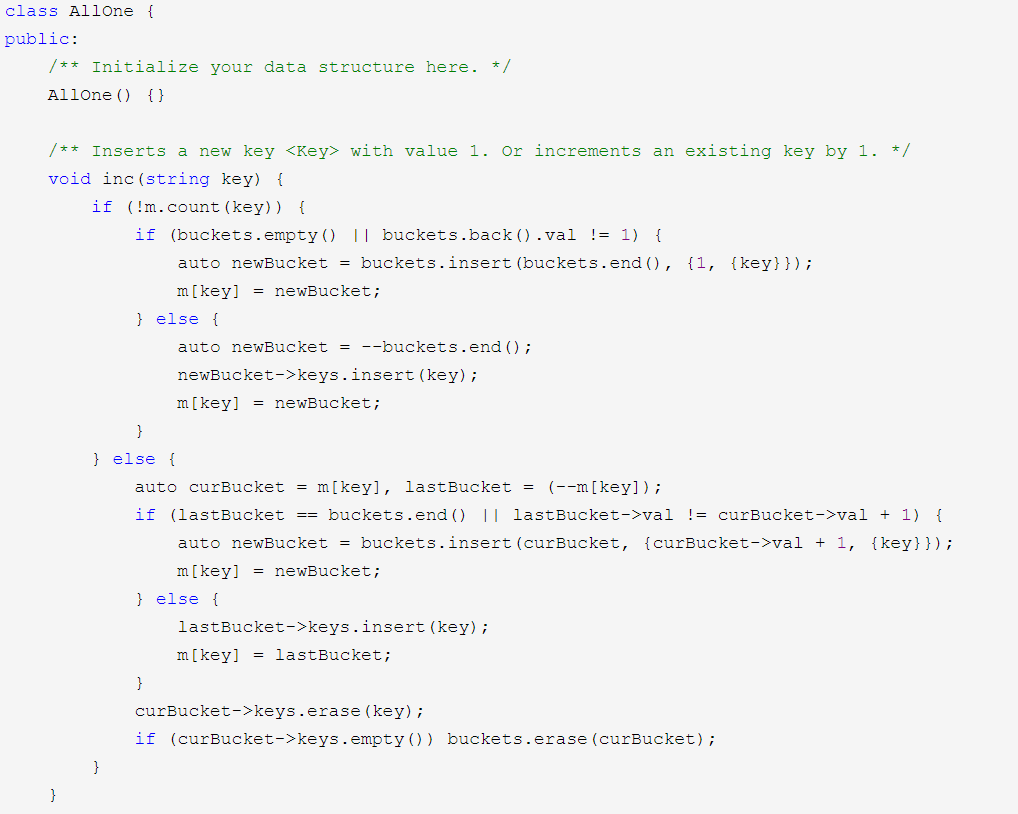
- 如果我们插入了一个已存在的key，那么由于个数增加了1个，所以该key值肯定不能在当前行继续待下去了，要往上升职啊，那么这里就有两种情况了，如果该key要升职到的那行不存在，我们需要手动添加那一行；如果那一行存在，我们之间将key加入集合keys中，记得都要将原来行中的key值删掉。

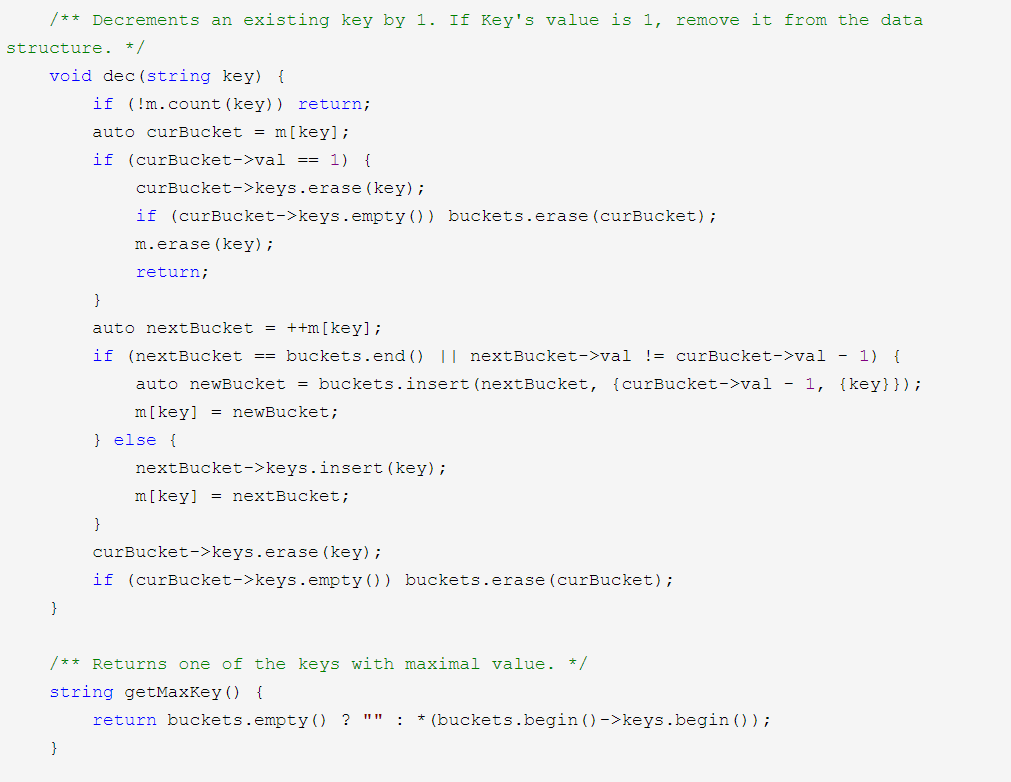
下面我们再来看dec函数如何实现，其实理解了上面的inc函数，那么dec函数也就没什么难度了：

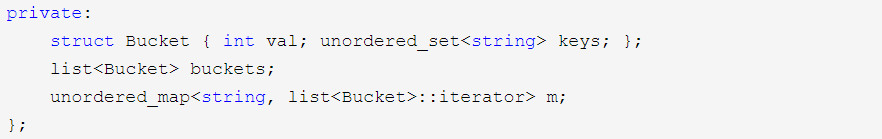
- 如果我们要删除的key不存在，那么直接返回即可。

- 如果我们要删除的key存在，那么我们看其val值是否为1，如果为1的话，那么直接在keys中删除该key即可，然后还需要判断如果该key是集合中的唯一一个，那么该行也需要删除。如果key的次数val不为1的话，我们要考虑降级问题，跟之前的升职很类似，如果要降级的行不存在，我们手动添加上，如果存在，则直接将key值添加到keys集合中即可。

当我们搞懂了inc和dec的实现方法，那么getMaxKey()和getMinKey()简直就是福利啊，不要太简单啊，直接返回首层和尾层的key值即可，参见代码如下：







716. Max Stack (Easy)課金