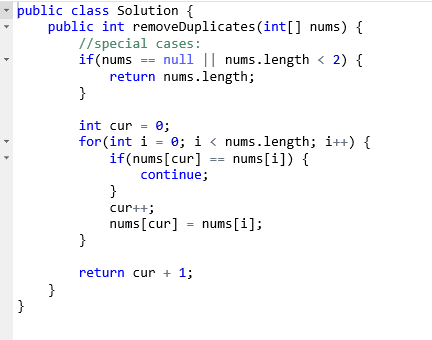
**刷题总结（2）**

**1 Remove duplicates from sorted array:**

这道题比较好的方法是采用“双指针”。首先将cur“指针”固定，移动i，直到第一个与cur所指元素不同的元素，之后cur加一，将i所指的元素与cur所指的元素交换：

To solve this problem in place and in O（n） time, I think we need two pointers. One pointer I, always moves move forward in a for loop to check whether the value pointed by it equals to the value pointed by cur. If the two value are the same, we just let I move. Once I encounter a value that does not equal to the value pointed by cur, because that at this time cur always points to the first duplicated element, we add one to cur, and then switch the values pointed by cur and I. Then with I and cur moving forward, all the duplicate values will be covered except the first one.



注意：

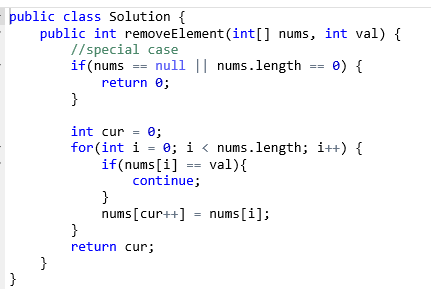
1 在交换cur和i所指的元素之前，cur要加1.

2 返回长度是cur值加1.

2 时间复杂度O(n)，空间O(1);

**2 Remove elements**

If we want to solve this problem in place and in O(n) time, we need to use two pointers. If we do not encounter the target value, the two pointers just move forward and they always point to the same position. (Thus it doesn’t matter if we switch their values). Once I points to a target value, we only let I moves forward until it reaches an element that does not equal to the target value. Because that at this time cur lacks behind and points to the first target value we encountered, we can switch the values pointed by cur and I, and then cur and I move forward. Because that the steps that cur lacks behind depends on how many continuing target values we found, all these values will be covered by new values when we switch the values pointed by cur and i:



注意：

1 本题中数组没有排序，与上题有区别。

2 在交换完之后cur值才加1，返回长度时就是cur的值。

3 时间O(n), 空间O(1).

**3 Pascal’s triangle**

Because that this problem requires us to return the whole triangle of n rows, we can just generate each row one by one. The elements in the current row can be computed by the elements of the last row. We can see that the first and the last element of each row are one, and for the rest elements, we can compute them as cur(i) = last(i - 1) + last(i). Thus, we can just use two for loops, one for computing each row and one for computing each element in a row.



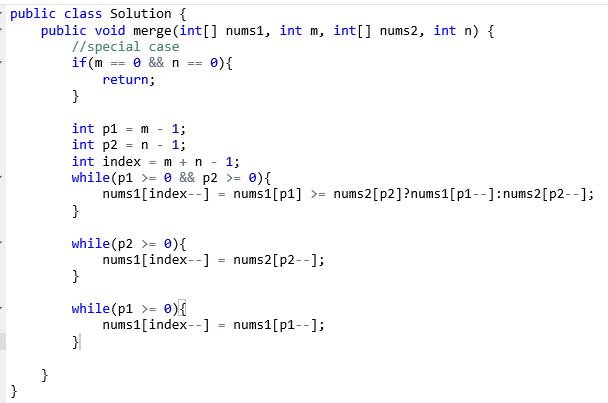
注意：

1 为了实现动态添加数组，我使用一个二维arraylist。

2 时间O(n^2), 空间也应该是O(n)级别?

**4 Merge sorted array**

Because that the two arrays are already sorted, we can use two pointers to scan the two arrays, at each time we compare the values pointed by the two pointers, choose the smaller one, store it in the new place, increment its pointer and then repeat this process. However, now we need to solve this problem in place with arrays, if we merge the two arrays from left to right, each time we want to store a new value from num2, we need to move all the elements beginning from the position of the pointer in num1, that’s time-consuming. I think we should merge from right to left because num1 has enough space for storing the two arrays. The pointers initially point to the last elements of the two array, we use another pointer to point to the current position for storing the new chosen value, which is at the end of num1 initially. We use a while loop and it stops when one of the pointer finishes scanning its array. Each time we choose the bigger element and store its value in the position in num1 pointing by index. After the while loop, it is possible that there are elements in an array left unmerged, we can just check whether p1 and p2 equal to 0, and if not, we move the rest elements into the rest places of num1.

注意：

1 由于是数组，从后往前归并可以避免随时挪动数组中的元素。

2 时间复杂度O(n), 空间O(1).

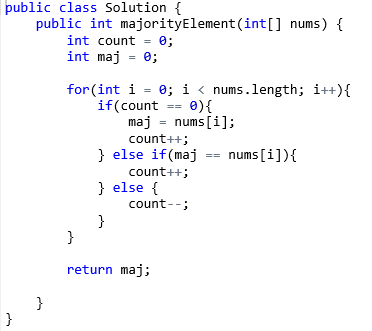
本题的测试集：

1. Nums1 = {0}, nums2 = {1}
2. Nums1 = {0}, nums2 = {1, 2, 3, 4, 5, 6}
3. Nums1 = {7}, nums2 = {1, 2, 3, 4, 5, 6}
4. Nums1 = {1,2,3,4,5,6}, nums2 = {7}
5. Nums1 = {2, 3, 4, 5, 6, 7}, nums2 = {1}
6. Nums1 = {1, 3, 5, 7, 9}, nums2 = {2, 4, 6, 8, 10}
7. Nums1 = {1, 2, 5, 7, 39}, nums2 = {0, 3, 4, 5 ,12, 17, 19, 20, 21, 33, 34, 45, 65, 72}

大数据量case无法给出

**5 Majority element**

The most straightforward way is to sort the array and then pick the one in the middle. However, there is one algorithm that can solve this problem in O(n) time. In this algorithm, we need a loop to scan all the elements in the array. We create two variables called count and major, initialized to 0. The algorithm works as follows: Each time we encounter a new element, we first check if count equals to zero, if it is, we set major to the value of the current element. If count is not 0, we check whether the value of major equals to the value of the current element, if it is, we increment count for 1, or we decrement count for 1. In this way, when finish scanning the array, the value of major must be the value of the majority element of the array.



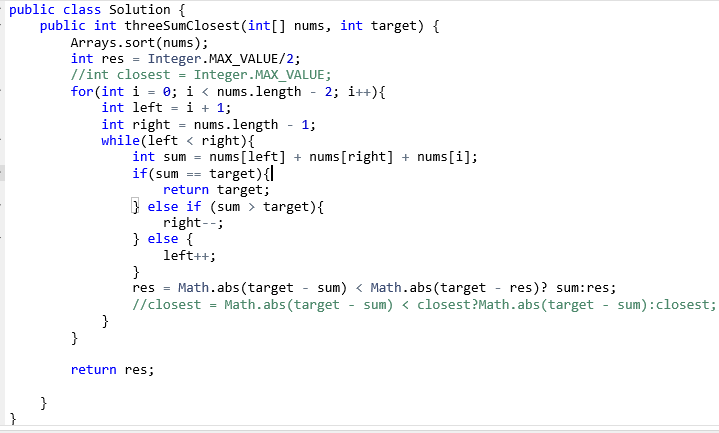
注意：

1 这个方法叫Moore’s voting algorithm.

2 时间复杂度O(n), 空间O(1).

**6 3 sum closest**

To this problem, if we use brute force algorithm, the time complexity is O(n^3). But I think there is a faster algorithm. We can at first fix one element, and use two pointers, one on the left and one on the right, to search for the closest value by moving them into the middle. First, we need to sort the array, this takes O(nlogn) time. Then, we use a pointer I to represent the position of the fixed element. We use a for loop and let I move from the left to right of the array, in each iteration, the position of I is fixed. In each iteration, we use two pointers to search the rest two elements. Left is initially equals to I + 1 and right is initially at the end of the array. When left < right, we keep searching. We first compute the sum of elements pointed by I, left and right, and compare it with the target value, if the sum is bigger than the target, considering that the array is sorted, we can only move the right pointer leftward to make the sum smaller, thus we decrement right for 1. Similarly, if the sum is smaller than the target, we increment left. Because that we need to find the sum value closest to the target, we create a global variable res to store the closest value that we have encountered and refresh it in each iteration. Of course, once we find a combination that equals to the target, we can just return. In this way, all the possible combination can be searched and the time complexity is O(n^2).



注意：

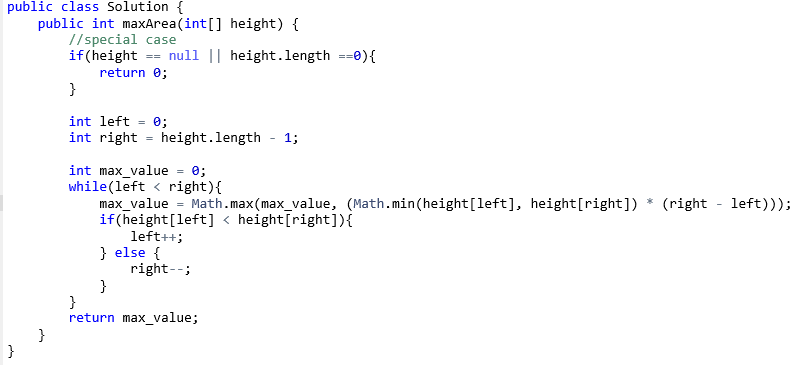
1 数组要先排序。

2 由于要找最接近的值，所以要有一个全局变量存储当前得到的最接近的值，每次得到一个组合，更新一下。

3 时间O(n^2), 空间O（1）

### **7 Container With Most Water**

For this problem, the brute force algorithm, that is, if we compute each pair of the vertical lines, the time complexity is O(n^2). However, in this problem one thing we need to pay attention is that the volume of water contained by two vertical lines has a lot to do with the shorter line of the two lines. Thus, we can use a greedy way to solve this problem in O(n) time. We use two pointers. One pointer, left, initialized to be at the leftmost (first) element of the array and the other pointer, right, initialized to point to the last element of array. When left < right, we keep searching. In each iteration, we compute the volume of water that can be contained by the left and right lines as: Math.min(height[left], height[right]) \* (right - left). Then, we compare the value of the height of line pointed by left and right. If the left element is smaller, this means that the volume of the container depends mostly on the height of the left line, if we move the right pointer to left, the volume will get smaller, thus, we can only move the right pointer to right. Similarly, if the right element is smaller, we can move it to left. We maintain a global variable to store the biggest volume we encountered. In this way, we can acquire the biggest volume in O(n) time.



注意

1 此题中，比较的是左右元素的相对大小，不需要先排序。

2 感觉这题有一些贪心的味道。

3 时间复杂度O(n), 空间O(1)。

**8 Minimum Path Sum**

This is a dynamic programming problem. If we use I and j to represent the element of the ith row and jth column, the minimum values of each position opt[i][j] can be computed by:

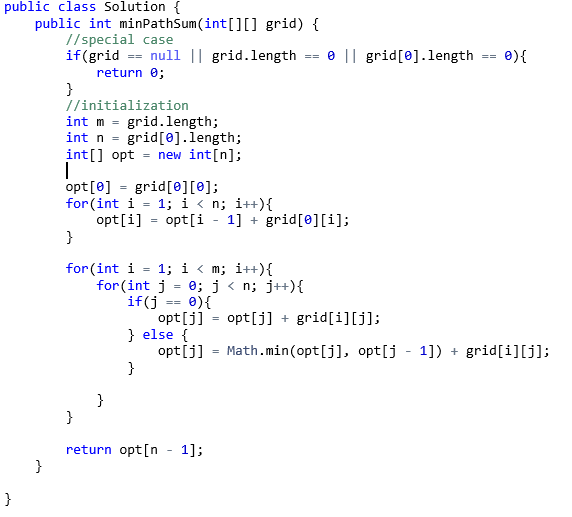
Opt[i][j] = min(opt[I - 1][j], opt[i][j - 1]) + grid[i][j] if I > 0 and j > 0

Opt[i][j - 1] + grid[i][j] if I = 0

Opt[I -1][j] + grid[i][j] if j = 0

Grid[i][j] if I = 0 and j = 0

Because that, when computing the opt value of element (I, j), we just need the opt values of the last row and the opt values of the position before the current position in the same row. Thus, we don’t need a m \* n matrix to store all the opt values, we just need an array of length n. When we compute the opt values of one row, we can just cover the existing values of the last row. The first is to initialize the array with the values in the first row. Then we use two for loops to iterate through all the positions of the grid, the result will be stored in the last element of the array.



注意：

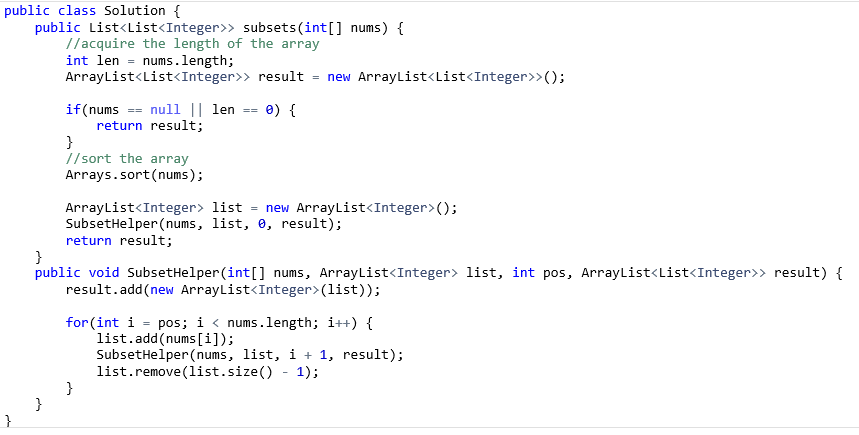
1 这道题使用一个n 长的array 记录opt values即可。

2 DP题目不要忘记初始化。

3 时间O(m \* n), 空间O(n)。

**9 Subsets**

This is an NP problem. We can use a recursive way to solve this problem. Because the element in a subset must be in non-decreasing order, we must sort the array first. We create another function helper to solve this problem. List is a temporary variable to hold the subset we have got. The input variable pos represents the index of the element in *num* that we begin to add to list. In the function, we first add the current list to the final result set because here list must be a valid subset. Then we use a for loop to iterate through all the values in *num* beginning from pos. In the for loop, we first add a new element into list to generate a new subset. Then we recursively call helper function, but variable pos must increment one. When the function helper returns, we must backtrack – delete the last element we just added to list, or some situations such as (1, 3) will never be added:



注意：

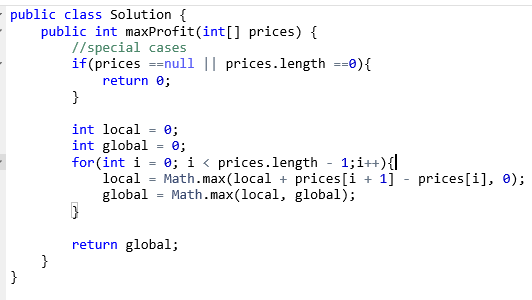
1要先进行排序。

2 注意递归调用返回后要回溯，即删掉最后刚加入的元素。

3 时间复杂度是O(2^n), 空间O(2^n).

**10 Best time to buy and sell stock**

We can use dynamic programming to solve this problem. We maintain two global variables, local and global. Local represents the profit we can get so far if we must sell the stock on the current position. Global stores the biggest profit we can get so far. We use a pointer to iterate through the array. And refresh local and global in each iteration:



注意：

1 指针i从0 到length – 1.

2 由于对于股票来说如果收益小于0，就可以不卖，所以计算local时用当前算出的local值与0比，小于0的话将local赋为0.

3 时间复杂度：O(n), 空间O（1）.

**数组题目总结1：**

数组的特点是可以在常数时间内取得任意位置的值，但缺点是长度固定，不容易加入或删除一个元素，因为要在中间加入或删除元素，即便空间够用，也需要挪动一部分元素。

个人感觉数组题目一般会结合其他比较有难度的知识点或技巧考察，比如：

**1 双指针：**

比如第1，2，6，7题。这种题目一般是要求一个由数组中多个元素组成的结果，需要将数组先排序，或者问题本身有规律，这样才能得到挪动指针的规则。这部分题目leetcode中有单独的专题。

**2 动态规划：**

动态规划个人感觉是常考类型，重中之重。一道题目，有如下特点的题目可能采用DP解：

1 求如下类型的结果：

a) Maximum/Minimum

b) Yes/No

c) Count all possbile solutions

2 原数据不能排序，元素位置不能调换。

**动态规划就是**解决了重复计算**的搜索**（九章算法）。个人感觉动态规划实际上将之前已经计算的结果存储，当之后的计算需要用到之前计算的结果时，直接调用结果，从而避免重复计算，减少时间复杂度。第8，10题是此类题目。我觉得咱们应该在这个专题里多花些时间。

**3 数组题目处理的小技巧：**

如第四题，为避免挪动其它的元素，采用从后往前merge的方法。第1，2题，用双指针解决数组删除元素时移动元素的问题。