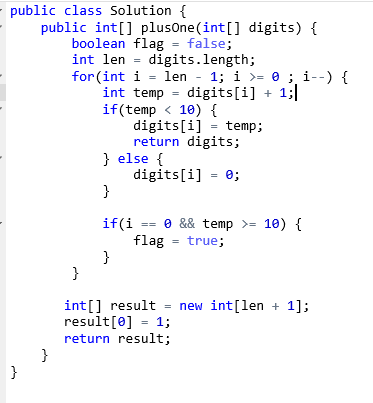
**刷题总结3**

**1 Plus one**

I think the major problem here is how to handle carry. Because that we only add 1 to the input number, there are two situations for each digit. First, the summation is smaller than 10 and the new value equals to digit[i] + 1, there is no carry. Second, the summation equals to 10, the new value is 0 and there is a carry of 1. Thus, we use a for loop to refresh each digit. We begin from the last value of the array, which is the most insignificant digit. In the for loop, we first add one to the current, if the summation is smaller than 10, we just refresh the value and return. If the summation equals to 10, we keep going in the for loop and add one to the next digit. Note that the leftmost digit may also need to carry and in this situation we need a new array. We can use a boolean variable flag, if the new value of the last element equals to 10, we set it to true. Then when the for loop exits, we create a new array and set its most significant digit to 1, the rest digits must be 0, so we do not need to assign new values to them:



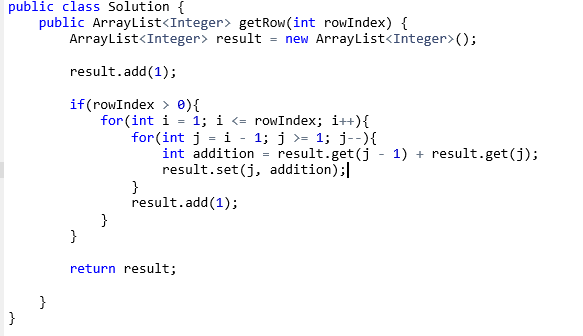
注意：

1 不要忘记最高位进位的问题，最高为进位需要重新开辟数组，把新最高位赋为1，其余为0，Java数组初始值就为0所以不用再改。

2 时间复杂度：O(n), 空间O(1)

**2 Pascal’s triangle II**

To get the values of one row of the Pascal’s triangle, we must make use of the last row. The regulation of the values in Pascal’s triangle is that, the first and last value is 1, and the ith value of one row can be acquired by adding up the (I – 1)th value and ith value of the last row. First, we create an arraylist to hold each row if the triangle, we first add 1 into the list as the first row. We need a for loop iterates from 1 to rowIndex to compute each row. Because that the new values in the current row can be computed by the values in the same and the last position of the last row, if we compute the values from right to left, we do not need to create new arraylists each time. We can just let new values cover the old values. Thus, we need another for loop iterates from I – 1 to 1 to compute the middle values of the current row, then set the values in the arraylist to new values. After the inner for loop, we should add the last ‘1’ to the arraylist:

注意：

1 为了计算每行时不用重新开辟数组，我们可以从右向左逐个计算。这样我们可以直接用新值覆盖上一行的值，而不影响下一位的计算。

2 每一行都比上一行多一个1，注意加上。

3 时间复杂度O(n^2), 空间O(n)

**3 Rotate Array**

I think, if we want to rotate in place, a good solution is as follows:

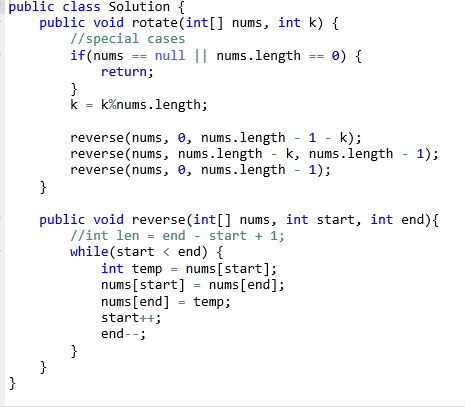
First, we reverse the 1th to (7 - k)th elements in the array

Then reverse the (7 – k + 1)th to the last element in the array

Finally, we reverse the elements in the whole array.

One thing we need to pay attention is that k may be bigger than the length of the array, thus, we first assign the value of k%num.length to k.

We can create a new function to reverse arrays. In this function, we reverse the elements in the array using two pointers. One initially points to the first element of the array and the other points to the end of the array. While ‘start’ is smaller than ‘end’, we switch the two elements pointed by the two pointers and then increment start and decrement end. After three reverse mentioned above, we can acquire the desired result:



注意：

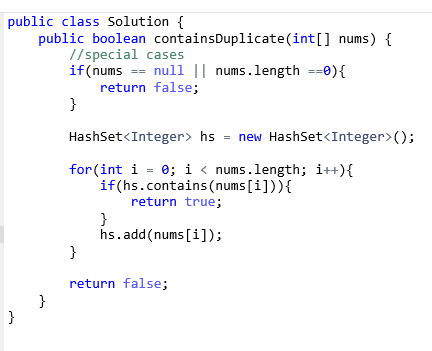
1 “三步翻转法”解决，不需要额外空间。

2 k值有可能比数组长度长，所以先将k与数组长度取模。

3 时间复杂度O(n), 空间复杂度O(1).

**4 Contains duplicate:**

For this problem, we can first sort the array, but this will take O(nlogn) time. If we want to solve the problem in O(n) time, we must use specific data structures. I think we can use a hashtable for that searching for an element in a hashtable only takes O(1) time. We use a for loop to scan the whole array, for each element, we first check whether it is already in the hashtable. If it is, it means that the value of the element appears before in the array and it is a duplicate, we return true. If we encounter a new value, we put it into the hashtable:



注意：

1 由于只需要存入已扫过的元素的值，不需要(key, value)值对，所以使用HashSet就可以了。

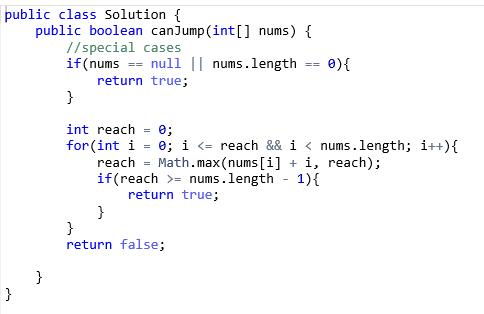
2 时间复杂度: O(n), 空间复杂度至少O(n)。

**5 Jump game:**

We should use dynamic programming to solve this problem. We maintain a variable called reach to hold the maximum position index we can reach until the current time. The equation of this problem is:

Reach = Max(nums[i] + I, reach) I >= 0

We use a for loop to scan the whole array. The initial value of reach is zero. Each time we encounter a new element, we refresh the value of reach. If reach is bigger than or equal to the position index of the last element of the array, it means that we can reach the last index and we can just return true. If the for loop exits, we return false:



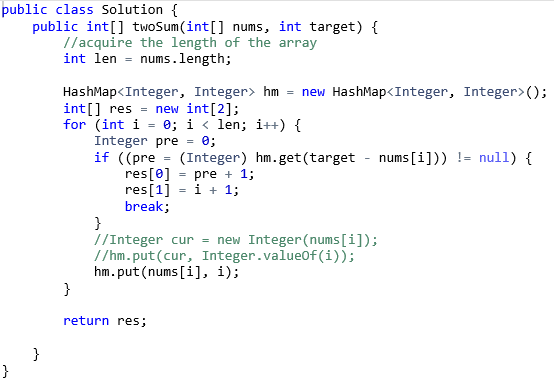
注意：

1 由于reach变量保存的是从数组起始位置开始能够到达的最远位置，随时在更新，不需要一个数组去保存每一个位置的reach。

2 时间复杂度：O(n), 空间O(1)

**6 Two sum**

Because that the problem requires us to return the indices of the two elements, if we sort the array, we also need extra space to store the original indices. Thus, I think we can make use of hashtable to solve this problem. Although this will use extra space, we can solve the problem in O(n) time. In the hashtable, we store the values of the elements we have encountered paired with their indices. We use a for loop to scan the whole array, every time we meet an element nums[i], we check whether the value (target – nums[i]) exists in the hashtable, if it does, we can easily get the indices of the two elements, and because that there is only one solution, we can break out of the for loop and return the needed values. If (target – nums[i]) does not exist in the array , we put it and its index into the hashtable and keep going in the for loop searching for solutions:



注意：

1 这道题使用了hashtable这种数据结构解决问题。实际上，还有一种做法是：首先对数组排序，之后是用双指针，一个在前一个在尾，使用“夹逼”方法寻找两个元素。但是这道题需要我们返回元素的index，如果采用排序夹逼的方法，也需要额外空间记录原本的index，所以相比用哈希表的方法没有优势。还有就是这道题说过结果只有一组，所以用哈希表解决比较好。如果可能存在相同的两个组合，那么哈希表的方法很难剔除重复。

2 时间O(n), 空间O(n).

### **7 Search Insert Position**

### In this problem, the array is sorted and we need to find an element in the array, I think binary search is a good way to solve this problem. We use two pointers which initially point to the first and last element of the array, when start + 1 is smaller than the value of end, we keep searching. In the while loop, we first compute the middle index between start and end, then we compare the value of nums[mid] and target. Depending on the relative value of them, we reassign the position of start and end pointer. When the while loop exits, there are five situations:

### 1 nums[start] equals to target, we return start.

### 2 nums[end] equals to target, we return end.

### 3 nums[start] is bigger than the target, we return start as the insert position.

### 4 nums[end] is smaller than target, we return end + 1

### 5 If none of above meets, target is bigger than nums[start] and smaller than nums[end], we return start + 1.

### 

注意：

1 二分法的while执行条件start + 1 < end，这样可以保证不会死循环。我觉的二分法可以在Leetcode binary search中单独总结一下模板。

2 如果target值在数组中找不到，需要返回应该插入的位置，这时会有5种情况，要考虑全面。

3 时间复杂度O(logn), 空间O(1).

**8 Unique Path**

This is a typical dynamic programming problem. If we use OPT[m][n] to represent the possible unique paths from the origin to the point [m][n], the general formula of computing OPT[m][n] is:

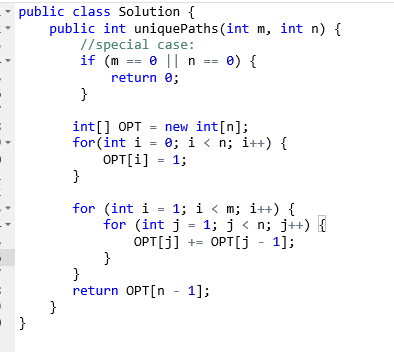
OPT[m][n] = OPT[m][ n - 1] m = 1, n > 1

OPT[m - 1][n] m > 1, n = 1

1 m, n = 1

OPT[m - 1][n] + OPT[m][n - 1] m, n > 1

Because that when computing OPT[m][n], we just need the OPT value of the same position of the last row and the OPT value of the last 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 to store the values line by line, when we compute a new OPT value, we can just cover the current one, which we do not need to use any more. In the program, we need to initialize the array first. All the OPT values are 1 for the first row. Then, we use two for loops, the outer for loop is used for computing each row and the inner for loop is used for computing the OPT values of the current row. The result is stored as the last element of the array when the for loops exit:



注意：

1. 使用一个长度为n的矩阵存储以往的计算结果即可，计算出新值后可以覆盖掉原值，因为之后不会再用。
2. 时间复杂度：O(m \* n), 空间O(n).

### **9 Search in Rotated Sorted Array**

### If there is no duplicate in the rotated array, we can make some modifications to the binary search algorithm to solve the problem. However, if there are duplicates in the array, we can do nothing but searching for the element linearly, because we can not locate the position of the target, we can not know whether the target is on the left part or the right part of the rotated array. Thus, for this problem, we just use a for loop to scan the array and search for the target element.

### 

### 注意：

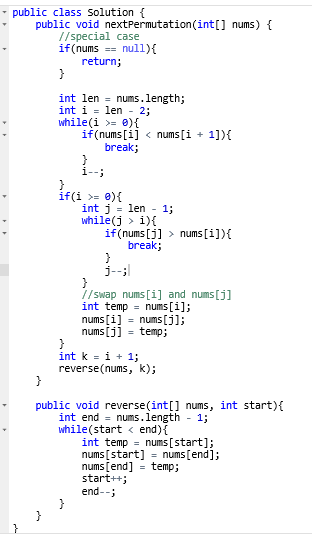
### 1 这道题和search in rotated array不同在于数组中可能有重复，举例:数组[1, 2, 3, 3 ,3, 3, 3]. Rotate之后是[3, 1, 2, 3, 3, 3, 3]，用二分法计算得到num[mid]后，其值和num[start]值一样，所以无法判断该target可能在哪一分支上，因此，只能用线性的算法查找。

### 时间复杂度O(n), 空间O(1)

**10 Next permutation:**

For this problem, there are three basic steps:

1. We scan the array from the end of the array to the beginning and we find the position of the first element that violate the increasing order of elements, we call it p.
2. After the first step, there are two kinds of situations. First, we can not find such an element in the array, this means that the array is totally lexicographically reversed, we just need to reverse the array again to rearrange it as the lowest possible order. If such an element exists, we should scan the array from the end of the array to find the first element that is bigger than p, we call it s, then we should swap p with the element which is before s.
3. Finally, we reverse the all the elements after p of the array:



注意：

1 这个算法，我觉得不容易想到，但是操作很简单，可以当规律记住。

2 时间复杂度O(n),空间O(1)