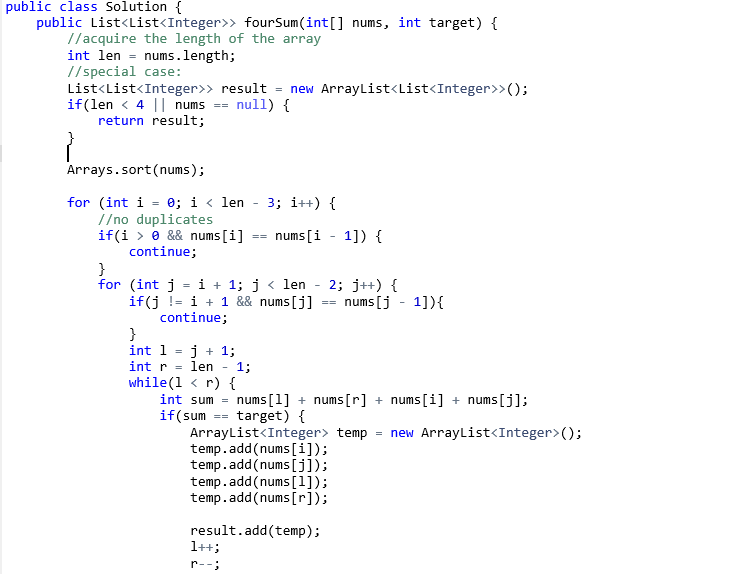
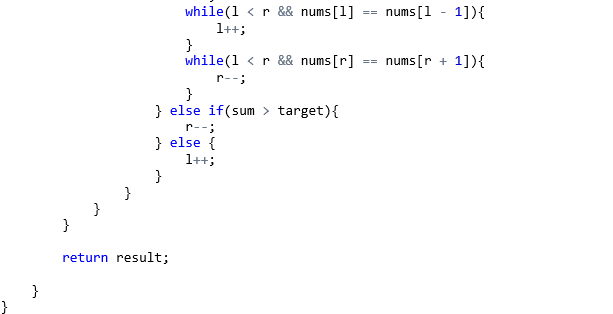
**刷题总结4**

**1 4Sum**

In this problem, we need to find 4 integers and the summation of them equals to a target value. Because that if we use two pointers to solve a 2Sum problem, we can solve it in O(n) time and we do not need to use extra space. I think we can simplify this problem into a 2Sum problem by fixing two integers in each iteration of searching. We first sort the array. In an outer for loop, we move a pointer I from the beginning of the array to right, and in this for loop, where I is fixed in each iteration, we use another for loop and move a pointer j from I + 1 to right, this is the second integer we are looking for. Because that we do not want duplicates in the results, when the element pointed by i or j equals to the last one, we move i or j one step further. In the second for loop, the first two integers are fixed, and we use two pointers to search for the rest two integers, this is a 2sum problem. We use two pointers, l and r, which initially pointing to j + 1 and the end of the array, and let them move to the centre. We use a while loop to search for these two integers, we first add up all the four integers, if the sum equals to the target, it means that we have found one combination, we create an arraylist and put it into the result. Then l increment 1 and r decrement 1, both of them move one step to the centre. To avoid duplicates, when the element pointed by I or r equals to the last one, we move I or r one step further. If the sum is bigger than the target, we must make the sum smaller and the only we to achieve this is to move the right pointer to left. Similarly, if the sum is smaller than the target, we can only move the left pointer to the right to make the sum bigger. In this way, we can solve a 2sum problem in O(n) time and with the two outer for loops, we can find all the combinations consisting of 4 integers.





注意：

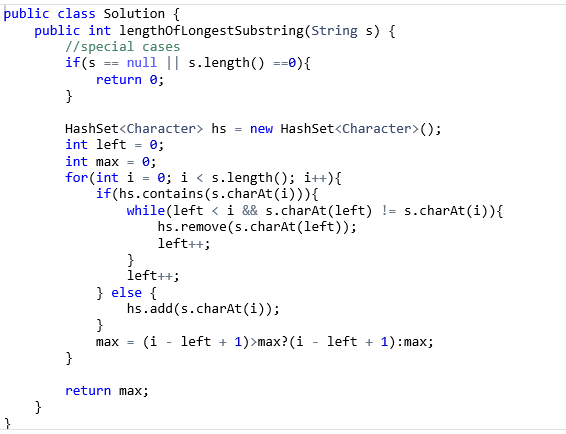
1 数组要先排序

2 为避免重复结果，当移动了指针后，如果当前元素与上一个元素相同，继续移动到下一个。不用担心会漏掉结果，because all the elements after this pointer has been searched by the pointers that moving after it.

3 时间O(n^3), 空间O(1).

**2 Longest Substring Without Repeating Characters**

For this problem, I think we can solve it in O(n) time making use of a hashtable. We can create a “window”, in this window, we maintain the substring without repeating characters that we are currently checking. We use a for loop and an index I, moving from the beginning of the string to the end, to represent the right edge of the “window”. We use another index, left, to represent the left edge of the window, it is initially pointing to the first character. We use a hashset to store the characters that we have encountered, in this way, once we meet a repeating character, we can be noticed immediately. In the for loop, we check whether the current character pointed by I already exists in the hashset or not. If there is a same character in the current substring in the “window”(between left and i), the current substring can not meet our requirement and we must move left pointer to the right until we pass through the same character. In this process, all the substrings we encountered can not meet our requirement, they all have repeating characters. We should remember to remove all the characters we meet in this process in the hashset, because they no longer exist in the current substring. If the character pointed by I does not exist in the hashset, we put it into the hashset and keep going in the iteration. In each iteration we compute the length of the current substring, we maintain the longest one that we have encountered.



注意：

1 这道题利用了“滑动窗”的方法。

2 时间复杂度O(2n) = O(n), 空间O(n).

**3 Valid Sudoku**

A sudoku is valid if it can meet three requirements:

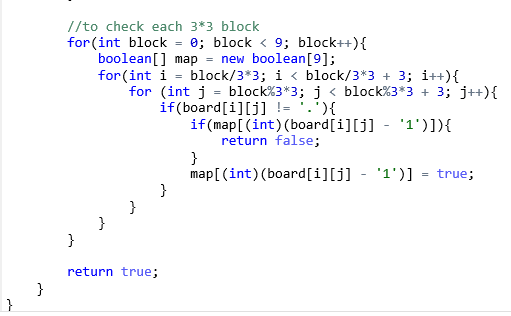
1 Each row must have 1 – 9 occuring just once.

2 Each column must have 1 – 9 occuring just once.

3 Each of the 3 \* 3 sub-boxes of the grid must have 1 – 9 occuring just once.

For this problem, I think we can just check each row, each column and each sub-box. First, we check each row. We use a for loop to check row one by one. When we are scanning one row, we create a boolean array with the length of 9 to store the numbers that we have met. Each time we check one element in the row, we first check if the corresponding position of the array is true, if it is, we must have encountered the same number in this row before, we return false. If it is false, we change the position into true. In this way, we only need an constant array to check each row. Then, we use the same method to check each column of the Sudoku. After that, we need to check each 3 \* 3 sub-box. There are 9 small sub-boxes, we first use a for loop to check each of them one by one. Then we can make use of the number of the sub-box to compute the index of each element, when checking each sub-box, we can use the same method mentioned before:





注意：

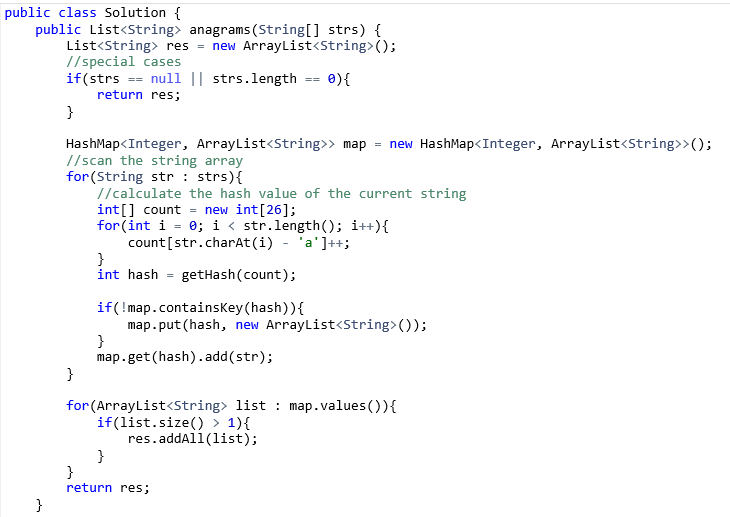
1 这个方法就是逐行，逐列以及对每个3 \* 3 小格进行扫描，看是否符合条件。 使用一个boolean数组，在相应的位置储存对应数字是否出现过，只要发现有重复，直接返回false。

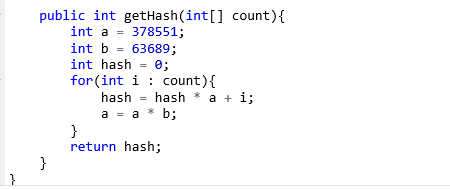
2 每行，每列以及每个小格都需要做9次检查，总共27 \* 9 次检测。

3 时间复杂度O(3 \* n^2), n = 9, 空间O(1)?

**4 Group Anagrams**

String anagrams have same length and same characters. The order of the characters can be different. I think an important thing of this problem is that how we recognize if several strings are anagrams or not. We can compute a kind of “hash value” of each string and guarantee that different anagrams have different “hash values”. For each string, we create a int array with length of 26, each position of the array store the numbers of corresponding letters in the string. Making use of the values in the array, we can compute a value for a given string, anagrams must have the same value because the this value have only to with the numbers of each letter appearing in the string. First we create a hashtable, the key is an integer (hashvalue) and the content is an arraylist for string all the anagrams of this hashvalue. We can scan the string array using a for loop, for each string, we compute the numbers of each letter and compute the hash value. If it is a new hash value, we put it into the hashtable. If the hashvalue already exists, we put the current string into the arraylist keyed by the hashvalue. Finally, we only need to iterate through the hashtable and put the anagrams in the arraylists into the final result.





注意：

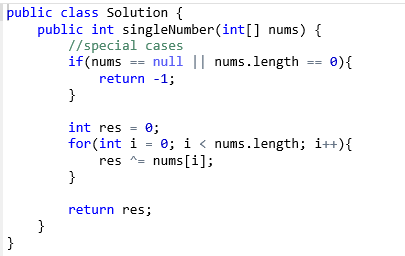
1 这道题的哈希值计算方法我也是参考了九章算法的方法，我觉得可以当作模板记住，不知哪位大神能具体解释一下这种方法的道理。

2 实际上，通过将string排序，也可以将anagrams聚到一起（anagrams按字母排序后都相同），但计算哈希值的时间复杂度是O(1),排序是O(nlogn).

3 假设有n个字符串，字符串最大长度为k，时间复杂度O(nk),空间O(nk).

**5 single number**

In this problem, we are required to solve the problem in O(n) time without extra space. Because that all elements appear twice except the one we are looking for, we can make use of a property of XOR operation. The XOR result of two identical integers must be 0. And XOR operation conforms to associativity. Thus, all we need to do is to XOR all the elements together and the final result must be the single number. In the program, we create a variable called res, it’s initial value is 0. Then we use a for loop to scan the array and compute res XOR the current integer. The value of res would be the single number.



注意：

1 这种方法利用了异或运算的性质：两个相同数字异或结果为0，以及其结合律。是种取巧的方法。

2 时间O(n),空间O(1).

**6 Happy Number:**

In the process of replacing the number, if we encounter a value that we have met before, the process would loop endlessly and this is not a happy number. Thus, we must use a data structure to store the numbers we acquire in this process. We can use a hashtable because the time for searching for a value in a hashtable is O(1). In the program, first we need to create a hashset. Then we use a while loop for the replacement process, the process continues if the replace number does not equal to 1. In the while loop, we first compute the new replace number. I create another function to compute the number. We just acquire each digit of the current number and sum up their squares. Then we check whether the new number already exists in the hashset, the process will loop endlessly and we can return false. If the new number does not exist in the hashset, we put it into the hashset and keep going in the process. If the while loop exits, we meet one and we return true;

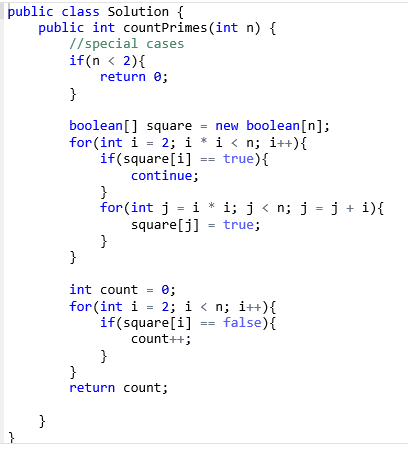


注意：

1 使用一个hashset记录每次新计算出的数值，一旦有重复，就会陷入死循环，返回false。

2 时间和空间复杂度怎么算我也不太清楚。

**7 Count Primes**

For this problem, we can make use of O(n) space to reduce the time complexity. We create a Boolean array of length n and each position in the array represents an integer. What we need to do is to mark off all the non-prime integers in this array by changing the corresponding positions into true. We first mark off all the integers that can be divided by 2, then mark off the integers that can be divided by 3. After that we consider 5 instead of 4, because all the integers that can be divided by 4 can also be divided by 2, and we have marked them off. We marking off the integers that can be divided by p when p < sqrt(n)(we don’t need to reach n) and the rest integers in the array are primes that are smaller than n. One thing to mention is that each time when we need to mark off the integers that can be divided by p, we do not need to begin with p, we can begin with the integer p^2, because all the non-primes smaller than p^2 has been marked off before. I the program, we use two for loops. The outer for loop iterates from 2 to sqrt(n), it represents the p mentioned before. In the for loop, we first check whether position p has been marked off, if it has not, we need t marked off all the integes that can be divided by p. We use a for loop to do this, beginning from p \* p. If position p has been marked off, then we continue with p + 1. Finally, we just need to count the number of positions in the array that are still flase.   


注意：

1 这个方法比较巧妙，可以记住。

2 leetcode上说时间复杂度是O(nLogLogn)，我不太清楚是怎么算的，空间应该是O(n).

**8 Isomorphic strings**

Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***. The mapping relations between the characters in the two strings must be consistent. If we scan the two strings at the same tme, we can acquire some mapping relations, but we need a data structure to store these relations for consistency checking in this process. I think we can make use of a hashtable. We first create a hashtable, both its key and value are character. Then we use a for loop to scan the two strings at the same time, let’s suppose that the current characters in the two strings are c1 and c2 respectively. In my program, I use c1 as the key of the hashtable. First we need to check whether c1 is already a key in the hashtable. If it is, we check whether the current c2 is same as the value of the key stored in the hashtable. If it is, then we keep checking the next two characters. If it isn’t, we return false because this means that the mapping relations are not consistent. If c1 is not yet a key value in the hashtable, we also have two situations. We need to check whether c2 is already stored in the hashtable as a value or not. If it is, this indicates that c2 maps to another character (not c1) before, this is inconsistent and we should return false. If c2 is not stored in the hashtable, c1 and c2 is new mappings, we put them into the hashtable. If the for loop exits, two strings are isomorphic:



注意：

1 使用一个hashtable记录已经得到的映射关系，注意映射关系必须是双向的，比如，a对应b，b只能和a对应。Java中可以直接利用containsValue方法检查反向的对应关系。

2 时间复杂度O(n), 空间O(n)。

**9 Repeated DNA sequences**

For this problem, we need a data structure to store the sequences that we have met. I think we can use a hashset. However, if we store character sequences in the hashset, it would be very space-consuming. Instead, because that there is 4 kinds of characters, we can encode each 10 – character sequence into integers. I use 0, 1, 2 and 3 to represent A, T, G and C respectively. Based on this mapping, we can encode each different 10-character sequence into a unique integer. I first create a hashmap to store the mapping relations between the characters and integers. Then I create two hashsets, one hashset, hs, is used to store the sequences we have encountered. Another one, unique, is used to store the sequences that appear more than once, I use a hashset to store them because same sequences may appear more than twice, but we only need one in the final result list. Same objects could not be added into the hashset twice. I use a for loop to scan the whole DNA sequence, and I make use of bit manipulations. Each time we meet a new character, we map it into an integer. In a binary point of view, it will be mapped into a two bit binary number. We don’t need to create a new variable to hold the encoded integer of each sequence, we just use one, called encode. Each time we finish encoding one character, we shift encode to left for 2 position and add the current encoded 2-bit binary to it. Because we only need to check each 10 – character sequence, we should maintain the last 10 encoded integers, which are 20 bits long. We can acquired the last 20 bits of variable encode by computing encode & 20 ‘1’s. Of course, when index I is smaller than 9, we can just add without cut. Then we only need to check if the current encoded value already exists in the hashset hs or not. If yes, we put the substring of last 10 characters into unique. If no, we continue. The sequences in unique are what we want.

注意：

1 这道题在每10个10个读取时，采用了二进制的一些操作，比如使用“与”操作截取前10个字符对应的编码串。

2 使用hashset存储已找到的重复字符串，这样即便相同字符串重现了第三次，也不会在结果中重复出现，因为同样的字串只能加入hashset一次。

3 时间：O(n), 空间O(n)

### 10 Fraction to Recurring Decimal

### When we divide two integers, we should do like this: first, we compute numerator / denominator, we only need the integer part. When computing the next digit, the numerator will change, we compute the new numerator as numerator % nominator \* 10. For this problem, it is possible that there is recurring decimal. At each time we compute the digits in the decimal, if the value of the past numerators appear again, there will be recurring decimal. Thus, I use a hashtable to store the values of numerators that we have met. Because we must remember the position that the recurring begins, we use values of the numerators as keys and its index in result string as the values of the hashtable. In the program, I use a stringbuffer to hold the result. We first use “XOR” to determine whether the result is negative or not. If it is, we add a ‘-’ first. To avoid overflow, before acquiring their absolute values, I map them into long. Next, we can acquire the integer part of the result by computing numerator / denominator. Then we compute the new numerator by numerator % denominator \* 10, if it is 0, we return result. If it is not 0, we create a hashmap add the point and use a while loop to compute the decimal. I use a variable cur to represent the numerators we compute from each digit. If cur is not 0, we keep computing. First we check whether the value of cur has appeared before or not. If it has, the recurring decimal begins from the index stored with the cur value that appear at the first time. We insert the left bracket in that position and add the right bracket at the end of the result string, then return. If the cur value is new, we add it into the hashmap, compute the current digit and add it into the result string, compute the new numerator and continue the process. Of course, if the value of cur is 0, the process stops and we return the value.

### 

### 

注意：

1 在对整形去绝对值时，为避免负转正溢出发生错误，可以先转换成long型。

2 时间复杂度 ? 空间复杂度?