快速排序算法

**版本一：数组的最后一位为主元**QUICKSORT(A, p, r)  
1 if p < r  
2    then q ← PARTITION(A, p, r)   //关键  
3         QUICKSORT(A, p, q - 1)  
4         QUICKSORT(A, q + 1, r)

数组划分  
快速排序算法的关键是PARTITION过程，它对A[p..r]进行就地重排：  
**PARTITION(A, p, r)**1  x ← A[r]  
2  i ← p - 1  
3  for j ← p to r - 1  
4       do if A[j] ≤ x  
5             then i ← i + 1  
6                  exchange A[i] <-> A[j]  
7  exchange A[i + 1] <-> A[r]  
8  return i + 1

求一个数组的中位数，复杂度为O(n)

可以采用快速排序的思路：不断地划分为左右两个区域，左边区域内的所有元素都小于某个主元，右边区域内的元素都大于某个主元。如果划分左右区域的分界j(这里的j和快速排序中的j的含义一致)刚好等于length/2，那么就找到了中位数，否则，如果下标 j<length/2，那么就继续在右边区域划分查找，如果下标j>length/2，那么就继续在左边区域划分查找，直到 j == length为止。

1. **private** **int** getMiddleNumber(**int**[] array,**int** left,**int** right){
2. **int** i = left;
3. **int** j = right+1;
4. **int** temp = array[left];//比较主元
6. **while**(**true**){
7. **while**(i<right && array[++i] < temp);
8. **while**(j>left && array[--j] >= temp);
10. **if**(i >= j)
11. **break**;
12. **else**
13. swap(array,i,j);  //交换，使得左边的都小于temp，右边的都大于temp
14. }
15. array[left]=array[j];
16. array[j] = temp;
18. //如果下标j等于length/2表示找到下标，此时数组坐标的都小于下标为j，右边都大于下标为j的元素
19. **if**(j == array.length/2){
20. **return** array[j];
21. }
22. //如果下标j小于length/2，就继续在右边找
23. **else** **if**(j < array.length/2)
24. **return** getMiddleNumber(array,j+1,right);
25. //如果下标j大于length/2则继续在左边划分寻找
26. **else**
27. **return** getMiddleNumber(array,left,j);
28. }

1.1有序链表中去除重复节点

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

For example,

Given 1->1->2, return 1->2.

Given 1->1->2->3->3, return 1->2->3.

***/\*\****

***\* Definition for singly-linked list.***

***\* public class ListNode {***

***\* int val;***

***\* ListNode next;***

***\* ListNode(int x) {***

***\* val = x;***

***\* next = null;***

***\* }***

***\* }***

***\*/*public** **class** Solution {

**public** ListNode deleteDuplicates(ListNode head) {

**if**(head == **null** || head.next == **null**)

**return** head;

ListNode prev = head;

ListNode p = head.next;

**while**(p != **null**){

**if**(p.val == prev.val){

prev.next = p.next;

p = p.next;

*//no change prev*

}**else**{

prev = p;

p = p.next;

}

}

**return** head;

}}

1.2有序数组中去除重复项

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example,  
Given input array A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

这道题要我们从有序数组中去除重复项，和之前那道[Remove Duplicates from Sorted List 移除有序链表中的重复项](http://www.cnblogs.com/grandyang/p/4066453.html) 的题很类似，但是要简单一些，因为毕竟数组的值可以通过下标直接访问，而链表不行。那么这道题的解题思路是，我们使用快慢指针来记录遍历的坐标，最开始时两个指针都指向第一个数字，如果两个指针指的数字相同，则快指针向前走一步，如果不同，则两个指针都向前走一步，这样当快指针走完整个数组后，慢指针当前的坐标加1就是数组中不同数组的个数，代码如下：

class Solution {

public:

int removeDuplicates(int A[], int n) {

if (n <= 1) return n;

int pre = 0, cur = 0;

while (cur < n) {

if (A[cur] == A[pre]) ++cur;

else A[++pre] = A[cur++];

}

return pre + 1;

}

};

2.Given a singly linked list L: L0→L1→ ... →Ln-1→Ln,  
reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→...

For example, given {1,2,3,4}, reorder it to {1,4,2,3}. You must do this in-place without altering the nodes' values.

**Analysis**

This problem is not straightforward, because it requires "in-place" operations. That means we can only change their pointers, not creating a new list.

**Java Solution**

This problem can be solved by doing the following:

1. Break list in the middle to two lists (use fast & slow pointers)，快慢指针找单链表的中间结点
2. Reverse the order of the second list，反转第二段链表
3. Merge two list back together，间隔的融合两个链表

**class** ListNode {

**int** val;

ListNode next;

ListNode(**int** x) {

val = x;

next = **null**;

}}

**public** **class** ReorderList {

**public** **static** **void** main(String[] args) {

ListNode n1 = **new** ListNode(1);

ListNode n2 = **new** ListNode(2);

ListNode n3 = **new** ListNode(3);

ListNode n4 = **new** ListNode(4);

n1.next = n2;

n2.next = n3;

n3.next = n4;

printList(n1);

reorderList(n1);

printList(n1);

}

**public** **static** **void** reorderList(ListNode head) {

**if** (head != **null** && head.next != **null**) {

ListNode slow = head;

ListNode fast = head;

*//use a fast and slow pointer to break the link to two parts.*

**while** (fast != **null** && fast.next != **null** && fast.next.next!= **null**) {

*//why need third/second condition?*

System.out.println("pre "+slow.val + " " + fast.val);

slow = slow.next;

fast = fast.next.next;

System.out.println("after " + slow.val + " " + fast.val);

}

ListNode second = slow.next;

slow.next = **null**;*// need to close first part*

*// now should have two lists: head and fast*

*// reverse order for second part*

second = reverseOrder(second);

ListNode p1 = head;

ListNode p2 = second;

*//merge two lists here*

**while** (p2 != **null**) {

ListNode temp1 = p1.next;

ListNode temp2 = p2.next;

p1.next = p2;

p2.next = temp1;

p1 = temp1;

p2 = temp2;

}

}

}

**public** **static** ListNode reverseOrder(ListNode head) {

**if** (head == **null** || head.next == **null**) {

**return** head;

}

ListNode pre = head;

ListNode curr = head.next;

**while** (curr != **null**) {

ListNode temp = curr.next;

curr.next = pre;

pre = curr;

curr = temp;

}

*// set head node's next*

head.next = **null**;

**return** pre;

}

**public** **static** **void** printList(ListNode n) {

System.out.println("------");

**while** (n != **null**) {

System.out.print(n.val);

n = n.next;

}

System.out.println();

}}

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

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

**public** **class** Solution {

**public** ListNode addTwoNumbers(ListNode l1, ListNode l2) {

**int** carry =0;

ListNode newHead = **new** ListNode(0);

ListNode p1 = l1, p2 = l2, p3=newHead;

**while**(p1 != **null** || p2 != **null**){

**if**(p1 != **null**){

carry += p1.val;

p1 = p1.next;

}

**if**(p2 != **null**){

carry += p2.val;

p2 = p2.next;

}

p3.next = **new** ListNode(carry%10);

p3 = p3.next;

carry /= 10;

}

**if**(carry==1)

p3.next=**new** ListNode(1);

**return** newHead.next;

}}

4.Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

**Analysis**

The key to solve the problem is defining a fake head. Then compare the first elements from each list. Add the smaller one to the merged list. Finally, when one of them is empty, simply append it to the merged list, since it is already sorted.

**public** ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode head = **new** ListNode(0);

ListNode p=head;

ListNode p1=l1;

ListNode p2=l2;

**while**(p1!=**null** && p2!=**null**){

**if**(p1.val < p2.val){

p.next = p1;

p1 = p1.next;

}**else**{

p.next = p2;

p2 = p2.next;

}

p=p.next;

}

**if**(p1!=**null**){

p.next = p1;

}

**if**(p2!=**null**){

p.next = p2;

}

**return** head.next;}

5.Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example, given 1->1->1->2->3, return 2->3.

**public** ListNode deleteDuplicates(ListNode head) {

ListNode t = **new** ListNode(0);

t.next = head;

ListNode p = t;

**while**(p.next!=null&&p.next.next!=**null**){

**if**(p.next.val == p.next.next.val){

**int** dup = p.next.val;

**while**(p.next!=null&&p.next.val==dup){

p.next = p.next.next;

}

}**else**{

p=p.next;

}

}

**return** t.next;

}

6.Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example, given 1->4->3->2->5->2 and x = 3, return 1->2->2->4->3->5.

**public** **class** Solution {

**public** ListNode partition(ListNode head, **int** x) {

**if**(head == **null**) **return** **null**;

ListNode fakeHead1 = **new** ListNode(0);

ListNode fakeHead2 = **new** ListNode(0);

fakeHead1.next = head;

ListNode p = head;

ListNode prev = fakeHead1;

ListNode p2 = fakeHead2;

**while**(p != **null**){

**if**(p.val < x){

p = p.next;

prev = prev.next;

}**else**{

p2.next = p;

prev.next = p.next;

p = p.next;

p2 = p2.next;

}

}

*// close the list*

p2.next = **null**;

//link two list

prev.next = fakeHead2.next;

**return** fakeHead1.next;

}}

# 7.[LeetCode – Find Peak Element](http://www.programcreek.com/2014/02/leetcode-find-peak-element/)

A peak element is an element that is greater than its neighbors. Given an input array where num[i] ≠ num[i+1], find a peak element and return its index. The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞. For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

**Thoughts**

This is a very simple problem. We can scan the array and find any element that is greater can its previous and next. The first and last element are handled separately.

**Java Solution**

|  |
| --- |
| **public** **class** Solution {  **public** **int** findPeakElement(**int**[] num) {  **int** max = num[0];  **int** index = 0;  **for**(**int** i=1; i<=num.length-2; i++){  **int** prev = num[i-1];  **int** curr = num[i];  **int** next = num[i+1];    **if**(curr > prev && curr > next && curr > max){  index = i;  max = curr;  }  }    **if**(num[num.length-1] > max){//最后一个元素特殊对待，当倒数第二个元素不是峰值的时候，最后  一个元素就需要与前面的峰值比较  **return** num.length-1;  }    **return** index;  }} |
|  |

8.1找出数组中出现次数超过一半的元素

1. **int** majorityElement(**const** vector<**int**>& nums) {
2. **int** m;
3. **int** count = 0;
4. **for** (auto num : nums) {
5. **if** (0 == count) {
6. m = num;
7. ++count;
8. }
9. **else** {
10. **if** (m == num)
11. ++count;
12. **else**
13. --count;
14. }
15. }
16. //若事先确定序列中存在超过一半的元素，则无需验证，m一定是目标元素
17. //若不确定，则需要遍历一遍这个数组，统计其出现的次数，验证是否超过一半
18. **return** m;
19. }

8.2、找出数组中出现次数超过三分之一的元素

1. pair<**int**, **int**> majorityElement(**const** vector<**int**>& nums) {
2. **int** m, n; //最多存在2个出现次数超过 1/3 的元素
3. **int** cm, cn; //对应 m 和 n 的统计
4. **for** (auto num : nums) {
5. **if** (cm == 0 || num == m) {
6. m = num;
7. ++cm;
8. }
9. **else** **if** (cn == 0 || num == n) {
10. n = num;
11. ++cn;
12. }
13. **else** {
14. --cm;
15. --cn;
16. }
17. }
19. //若事先确定序列中存在超过1/3的元素，则无需验证，m一定是目标元素
20. //若不确定，则需要遍历一遍这个数组，统计其出现的次数，验证是否超过1/3

     cm = cn = 0;

for (auto &a : nums) {

if (a == m) ++cm;

else if (a == n) ++cn;

}

if (cm > nums.size() / 3) res.push\_back(m);

if (cn > nums.size() / 3) res.push\_back(n);

return res;

1. }

9.Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example, given [3,2,1,5,6,4] and k = 2, return 5.

Note: You may assume k is always valid, 1 ≤ k ≤ array's length.

We can use a min heap to solve this problem. The heap stores the top k elements. Whenever the size is greater than k, delete the min. Time complexity is O(nlog(k)). Space complexity is O(k) for storing the top k numbers.

**public** **int** findKthLargest(**int**[] nums, **int** k) {

PriorityQueue<Integer> q = **new** PriorityQueue<Integer>(k);

**for**(**int** i: nums){

q.offer(i);

**if**(q.size()>k){

q.poll();

}

}

**return** q.peek();}

10.Given a sorted integer array without duplicates, return the summary of its ranges for consecutive numbers.

For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

**public** List<String> summaryRanges(**int**[] nums) {

List<String> result = **new** ArrayList<String>();

**if**(nums == **null** || nums.length==0)

**return** result;

**if**(nums.length==1){

result.add(nums[0]+"");

}

**int** pre = nums[0]; *// previous element*

**int** first = pre; *// first element of each range*

**for**(**int** i=1; i<nums.length; i++){

**if**(nums[i]==pre+1){

**if**(i==nums.length-1){

result.add(first+"->"+nums[i]);

}

}**else**{

**if**(first == pre){

result.add(first+"");

}**else**{

result.add(first + "->"+pre);

}

**if**(i==nums.length-1){

result.add(nums[i]+"");

}

first = nums[i];

}

pre = nums[i];

}

**return** result;

11.Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum ≥ s. If there isn't one, return 0 instead.

For example, given the array [2,3,1,2,4,3] and s = 7, the subarray [4,3] has the minimal length of 2 under the problem constraint.

**Analysis**

We can use 2 points to mark the left and right boundaries of the sliding window. When the sum is greater than the target, shift the left pointer; when the sum is less than the target, shift the right pointer.

|  |
| --- |
| **public** **int** minSubArrayLen(**int** s, **int**[] nums) {  **if**(nums==**null**||nums.length==0)  **return** 0;    **int** i=0;  **int** j=0;  **int** sum=0;    **int** minLen = Integer.MAX\_VALUE;    **while**(j<nums.length){  **if**(sum<s){  sum += nums[j];  j++;  }**else**{  minLen = Math.min(minLen, j-i);  **if**(i==j-1)  **return** 1;    sum -=nums[i];  i++;  }  }    **while**(sum>=s){//j已经到数组的尾部，此时的和仍大于s  minLen = Math.min(minLen, j-i);    sum -=nums[i++];  }    **return** minLen==Integer.MAX\_VALUE? 0: minLen; |

12.1Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

**Java Solution - Dynamic Programming**

This is similar to [maximum subarray](http://www.programcreek.com/2013/02/leetcode-maximum-subarray-java/" \o "LeetCode – Maximum Subarray (Java)). Instead of sum, the sign of number affect the product value.

When iterating the array, each element has two possibilities: positive number or negative number. We need to track a minimum value, so that when a negative number is given, it can also find the maximum value. We define two local variables, one tracks the maximum and the other tracks the minimum.

|  |
| --- |
| **public** **int** maxProduct(**int**[] nums) {  **int**[] max = **new** **int**[nums.length];  **int**[] min = **new** **int**[nums.length];    max[0] = min[0] = nums[0];  **int** result = nums[0];    **for**(**int** i=1; i<nums.length; i++){  **if**(nums[i]>0){  max[i]=Math.max(nums[i], max[i-1]\*nums[i]);  min[i]=Math.min(nums[i], min[i-1]\*nums[i]);  }**else**{  max[i]=Math.max(nums[i], min[i-1]\*nums[i]);  min[i]=Math.min(nums[i], max[i-1]\*nums[i]);  }    result = Math.max(result, max[i]);  }    **return** result;  } |

Time is O(n).

12.2Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4], the contiguous subarray [4,−1,2,1] has the largest sum = 6.

**1. Dynamic Programming Solution**

The changing condition for dynamic programming is "We should ignore the sum of the previous n-1 elements if nth element is greater than the sum."

|  |
| --- |
| **public** **class** Solution {  **public** **int** maxSubArray(**int**[] A) {  **int** max = A[0];  **int**[] sum = **new** **int**[A.length];  sum[0] = A[0];    **for** (**int** i = 1; i < A.length; i++) {  sum[i] = Math.max(A[i], sum[i - 1] + A[i]);  max = Math.max(max, sum[i]);  }    **return** max;  }} |

13.LeetCode - Reverse Integer:

*Reverse digits of an integer.  
Example1: x = 123, return 321  
Example2: x = -123, return -321*

**1. Naive Method**

We can convert the integer to a string/char array, reverse the order, and convert the string/char array back to an integer. However, this will require extra space for the string. It doesn't seem to be the right way, if you come with such a solution.

**2. Efficient Approach**

Actually, this can be done by using the following code.

|  |
| --- |
| **public** **int** reverse(**int** x) {  *//flag marks if x is negative*  **boolean** flag = **false**;  **if** (x < 0) {  x = 0 - x;  flag = **true**;  }  **int** res = 0;  **int** p = x;  **while** (p > 0) {  **int** mod = p % 10;  p = p / 10;  res = res \* 10 + mod;  }  **if** (flag) {  res = 0 - res;  }  **return** res;} |

1. 。二叉树最近公共父节点

在二叉树中找最近公共父节点。分为两种情况，一种是有父指针，一种没有父指针。

## 没有父指针

这种情况有点难。首先从根节点开始向下找，如果根节点等于其中一个子节点，那么根节点便是最近公共父结点。否则计算左子树和右子树中包含n1或n2的个数。如果左子树包含n1、n2那么最近公共父结点在左子树，如果右子树包含n1和n2，那么在右子树。如果左右子树各包含一个，那么最近公共父结点就是当前结点。如果二叉树是平衡的，那么算法复杂度为O(logN)。最坏情况就是树成了链表，算法时间负责度为O(N^2)。

思路清晰了，可以编写代码：

1. #include<iostream>
2. **struct** Node
3. {
4. **int** data;
5. Node\* left;
6. Node\* right;
7. Node() :left(NULL), right(NULL)
8. {}
9. };
10. //计算当前结点包含n1、n2个数
11. **int** countMatch(Node \*current, Node\* n1, Node\* n2)
12. {
13. **if** (current == NULL)
14. **return** 0;
15. **int** count = countMatch(current->left, n1, n2) + countMatch(current->right, n1, n2);
16. **if** (current == n1 || current == n2)
17. **return** 1 + count;
18. **return** count;
19. }
20. Node\* findLCA(Node\* root, Node\* n1, Node\* n2)
21. {
22. **if** (root == NULL)
23. **return** NULL;
24. **if** (root == n1 || root == n2)
25. **return** root;
26. **int** count = countMatch(root->left, n1, n2);//左子树包含n1和n2的个数
27. **if** (count == 1)
28. **return** root;//左子树一个，右子树肯定也有一个
29. **else** **if** (count == 2)//都在左子树
30. **return** findLCA(root->left, n1, n2);
31. **else**//都在右子树
32. **return** findLCA(root->right, n1, n2);
33. }
34. **int** main()
35. {
36. //测试
37. Node\* A[11];
38. **for** (**int** i = 0; i < 11; ++i)
39. {
40. A[i] = **new** Node();
41. A[i]->data = i;
42. }
44. **for** (**int** i = 0; i < 5; ++i)
45. {
46. A[i]->left = A[i \* 2 + 1];
48. A[i]->right = A[i \* 2 + 2];
50. }
52. Node\* Ancestor = findLCA(A[0],A[7], A[10]);

55. }

15.判断IP地址的合法性

#include <stdio.h>

int is\_valid\_ip(const char \*ip)

{

int section = 0; //每一节的十进制值

int dot = 0; //几个点分隔符

int last = -1; //每一节中上一个字符

while(\*ip)

{

if(\*ip == '.')

{

dot++;

if(dot > 3)

{

return 0;

}

if(section >= 0 && section <=255)

{

section = 0;

}else{

return 0;

}

}else if(\*ip >= '0' && \*ip <= '9')

{

section = section \* 10 + \*ip - '0';

if(last == '0')

{

return 0;

}

}else{

return 0;

}

last = \*ip;

ip++;

}

if(section >= 0 && section <=255)

{

if(3 == dot)

{

section = 0;

printf ("IP address success!\n");

//printf ("%d\n",dot);

return 0;

}

}

return 1;

}

int main()

{

is\_valid\_ip("192.249.23.123");

}

寻找重复数

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

**Note:**

1. You must not modify the array (assume the array is read only).
2. You must use only constant extra space.
3. Your runtime complexity should be less than O(n2).

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

这道题给了我们n+1个数，所有的数都在[1, n]区域内，首先让我们证明必定会有一个重复数，这不禁让我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" \t "http://www.cnblogs.com/grandyang/p/_blank)), 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。题目要求我们不能改变原数组，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，那我们也就只能考虑用二分搜索法了，我们在区别[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，如果个数大于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：

class Solution {

public:

int findDuplicate(vector<int>& nums) {

int low = 1, high = nums.size() - 1;

while (low < high) {

int mid = low + (high - low) \* 0.5;

int cnt = 0;

for (auto a : nums) {

if (a <= mid) ++cnt;

}

if (cnt <= mid) low = mid + 1;

else high = mid;

}

return low;

}

};

在一个有序矩阵中搜索一个数

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

[

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

Given target = 3, return true.

这道题要求搜索一个二维矩阵，由于给的矩阵是有序的，所以很自然的想到要用[二分查找法](http://zh.wikipedia.org/wiki/%E6%8A%98%E5%8D%8A%E6%90%9C%E7%B4%A2%E7%AE%97%E6%B3%95" \t "http://www.cnblogs.com/grandyang/p/_blank)，我们可以在第一列上先用一次二分查找法找到目标值所在的行的位置，然后在该行上再用一次二分查找法来找是否存在目标值，代码如下：

// Two binary search

class Solution {

public:

bool searchMatrix(vector<vector<int> > &matrix, int target) {

if (matrix.empty() || matrix[0].empty()) return false;

if (target < matrix[0][0] || target > matrix.back().back()) return false;

int left = 0, right = matrix.size() - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (matrix[mid][0] == target) return true;

else if (matrix[mid][0] < target) left = mid + 1;

else right = mid - 1;

}

int tmp = right;

left = 0;

right = matrix[tmp].size() - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (matrix[tmp][mid] == target) return true;

else if (matrix[tmp][mid] < target) left = mid + 1;

else right = mid - 1;

}

return false;

}

};

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

* Integers in each row are sorted in ascending from left to right.
* Integers in each column are sorted in ascending from top to bottom.

For example,

Consider the following matrix:

[

[1, 4, 7, 11, 15],

[2, 5, 8, 12, 19],

[3, 6, 9, 16, 22],

[10, 13, 14, 17, 24],

[18, 21, 23, 26, 30]

]

Given **target** = 5, return true.

Given **target** = 20, return false.

突然发现LeetCode很喜欢从LintCode上盗题，这是逼我去刷LintCode的节奏么?! 这道题让我们在一个二维数组中快速的搜索的一个数字，这个二维数组各行各列都是按递增顺序排列的，是之前那道[Search a 2D Matrix 搜索一个二维矩阵](http://www.cnblogs.com/grandyang/p/4323301.html)的延伸，那道题的不同在于每行的第一个数字比上一行的最后一个数字大，是一个整体蛇形递增的数组。所以那道题可以将二维数组展开成一个一位数组用一次二查搜索。而这道题没法那么做，这道题有它自己的特点。如果我们观察题目中给的那个例子，我们可以发现有两个位置的数字很有特点，左下角和右上角的数。左下角的18，往上所有的数变小，往右所有数增加，那么我们就可以和目标数相比较，如果目标数大，就往右搜，如果目标数小，就往左搜。这样就可以判断目标数是否存在。当然我们也可以把起始数放在右上角，往左和下搜，停止条件设置正确就行。代码如下：

class Solution {

public:

bool searchMatrix(vector<vector<int> > &matrix, int target) {

if (matrix.empty() || matrix[0].empty()) return false;

if (target < matrix[0][0] || target > matrix.back().back()) return false;

int x = matrix.size() - 1, y = 0;

while (true) {

if (matrix[x][y] > target) --x;

else if (matrix[x][y] < target) ++y;

else return true;

if (x < 0 || y >= matrix[0].size()) break;

}

return false;

}

};

三数之和小于指定的数

Given an array of n integers nums and a target, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition nums[i] + nums[j] + nums[k] < target.

For example, given nums = [-2, 0, 1, 3], and target = 2.

Return 2. Because there are two triplets which sums are less than 2:

[-2, 0, 1]

[-2, 0, 3]

Follow up:  
Could you solve it in O(n2) runtime?

这道题是3Sum问题的一个变形，让我们求三数之和小于一个目标值，那么最简单的方法就是穷举法，将所有的可能的三个数字的组合都遍历一遍，比较三数之和跟目标值之间的大小，小于的话则结果自增1

题目中的Follow up让我们在O(n^2)的时间复杂度内实现，那么我们借鉴之前那两道题[3Sum Closest](http://www.cnblogs.com/grandyang/p/4510984.html" \t "http://www.cnblogs.com/grandyang/p/_blank)和[3Sum](http://www.cnblogs.com/grandyang/p/4481576.html" \t "http://www.cnblogs.com/grandyang/p/_blank)中的方法，采用双指针来做，这里面有个trick就是当判断三个数之和小于目标值时，此时结果应该加上right-left，因为数组排序了以后，如果加上num[right]小于目标值的话，那么加上一个更小的数必定也会小于目标值，然后我们将左指针右移一位，否则我们将右指针左移一位，参见代码如下：

// O(n^2)

class Solution {

public:

int threeSumSmaller(vector<int>& nums, int target) {

if (nums.size() < 3) return 0;

int res = 0;

sort(nums.begin(), nums.end());

for (int i = 0; i < nums.size() - 2; ++i) {

int left = i + 1, right = nums.size() - 1;

while (left < right) {

if (nums[i] + nums[left] + nums[right] < target) {

res += right - left;

++left;

} else {

--right;

}

}

}

return res;

}

};

#### **Contains Duplicate III**

【题目】 Given an array of integers, find out whether there are two distinct indices  *i*  and  *j*  in the array such that the difference between  nums[i]  and  nums[j]  is at most  *t*  and the difference between  *i*  and  *j*  is at most  *k* .

【解答】终于有一道红黑树的题目了。首先在脑海里整理题意，相当于有一个窗口，窗口大小不得超过k，在窗口移动的过程中，寻找窗口内部元素不超过t的情况。

用一个TreeSet来存放窗口内的元素们，每来一个新元素，就：

* 使用floor方法去找是否已存在最大的小于等于新元素的元素，如果找到，就加上t看看是否能够等于或超过新元素，如果是，就说明这两个元素差值小于等于t；
* 使用ceiling方法去找是否已存在最小的大于等于新元素的元素，如果找到，就减掉t看看是否能够等于或者小于新元素，如果是，也说明这两个元素差值小于等于t。

**public** **class** **Solution** {

**public** boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

**if** (k <= 0 || t < 0)

**return** false;

TreeSet<Integer> **set** = new TreeSet<>();

**for** (int i = 0; i < nums.length; i++) {

int **val** = nums[i];

**if** (**set**.floor(**val**) != null && **val** <= t + **set**.floor(**val**))

**return** true;

**if** (**set**.ceiling(**val**) != null && **set**.ceiling(**val**) <= t + **val**)

**return** true;

**set**.add(**val**);

**if** (i >= k)

**set**.remove(nums[i - k]);

}

**return** false;

}

}

#### **Contains Duplicate II**

【题目】 Given an array of integers and an integer  *k* , find out whether there there are two distinct indices  *i*  and  *j*  in the array such that  nums[i] = nums[j]  and the difference between  *i* and *j*  is at most  *k* .

【解答】首先要理解题意，其实就是在一个长长的数组中，有一个大小可伸缩的滑动窗口，但是这个窗口的宽度不得超过j-i+1，在滑动这个窗口的过程中，如果发现两个数相等，那就返回true。因此使用一个map来存放滑动期间收集到的信息，key是该数，value是该数最后一次出现的位置。如果某次迭代发现该数再次出现，且两次出现的间距符合要求，就返回。

public **class** **Solution** {

public boolean containsNearbyDuplicate(int[] nums, int k) {

**if** (**null**==nums || nums.length<=1 || k<=0)

**return** **false**;

Map<Integer, Integer> map = **new** HashMap<>();

**for** (int i=0; i<nums.length; i++) {

int num = nums[i];

**if** (!map.containsKey(num)) {

map.put(num, i);

} **else** {

int diff = i - map.**get**(num);

**if** (diff<=k)

**return** **true**;

**else**

map.put(num, i);

}

}

**return** **false**;

}

}

四数之和

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

To make problem a bit easier, all A, B, C, D have same length of N where 0 ≤ N ≤ 500. All integers are in the range of -228 to 228 - 1 and the result is guaranteed to be at most 231 - 1.

Example:

Input:

A = [ 1, 2]

B = [-2,-1]

C = [-1, 2]

D = [ 0, 2]

Output:

2

Explanation:

The two tuples are:

1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0

2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0

这道题是之前那道[4Sum](http://www.cnblogs.com/grandyang/p/4515925.html" \t "http://www.cnblogs.com/grandyang/p/_blank)的延伸，让我们在四个数组中各取一个数字，使其和为0。那么最傻的方法就是遍历所有的情况，时间复杂度为O(n4)。但是我们想想既然[Two Sum](http://www.cnblogs.com/grandyang/p/4130379.html" \t "http://www.cnblogs.com/grandyang/p/_blank)那道都能将时间复杂度缩小一倍，那么这道题我们使用哈希表是否也能将时间复杂度降到O(n2)呢？答案是肯定的，我们如果把A和B的两两之和都求出来，在哈希表中建立两数之和跟其出现次数之间的映射，那么我们再遍历C和D中任意两个数之和，我们只要看哈希表存不存在这两数之和的相反数就行了，参见代码如下：

class Solution {

public:

int fourSumCount(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& D) {

int res = 0;

unordered\_map<int, int> m;

for (int i = 0; i < A.size(); ++i) {

for (int j = 0; j < B.size(); ++j) {

++m[A[i] + B[j]];

}

}

for (int i = 0; i < C.size(); ++i) {

for (int j = 0; j < D.size(); ++j) {

int target = -1 \* (C[i] + D[j]);

res += m[target];

}

}

return res;

}

};

最小移动次数使得数组各个元素相等

Given a non-empty integer array of size n, find the minimum number of moves required to make all array elements equal, where a move is incrementing n - 1 elements by 1.

Example:

Input:

[1,2,3]

Output:

3

Explanation:

Only three moves are needed (remember each move increments two elements):

[1,2,3] => [2,3,3] => [3,4,3] => [4,4,4]

这道题给了我们一个长度为n的数组，说是我们每次可以对n-1个数字同时加1，问最少需要多少次这样的操作才能让数组中所有的数字相等。那么我们想，为了快速的缩小差距，该选择哪些数字加1呢，不难看出每次需要给除了数组最大值的所有数字加1，这样能快速的到达平衡状态。但是这道题如果我们老老实实的每次找出最大值，然后给其他数字加1，再判断是否平衡，思路是正确，但是OJ不答应。正确的解法相当的巧妙，需要换一个角度来看问题，其实给n-1个数字加1，效果等同于给那个未被选中的数字减1，比如数组[1，2，3], 给除去最大值的其他数字加1，变为[2，3，3]，我们全体减1，并不影响数字间相对差异，变为[1，2，2]，这个结果其实就是原始数组的最大值3自减1，那么问题也可能转化为，将所有数字都减小到最小值，这样难度就大大降低了，我们只要先找到最小值，然后累加每个数跟最小值之间的差值即可，参见代码如下：

 class Solution {

public:

int minMoves(vector<int>& nums) {

int mn = INT\_MAX, res = 0;

for (int num : nums) mn = min(mn, num);

for (int num : nums) res += num - mn;

return res;

}

};

根据字符出现频率排序

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.

这道题让我们给一个字符串按照字符出现的频率来排序，那么毫无疑问肯定要先统计出每个字符出现的个数，那么之后怎么做呢？我们可以利用优先队列的自动排序的特点，把个数和字符组成pair放到优先队列里排好序后，再取出来组成结果res即可，参见代码如下：

class Solution {

public:

string frequencySort(string s) {

string res = "";

priority\_queue<pair<int, char>> q;

unordered\_map<char, int> m;

for (char c : s) ++m[c];

for (auto a : m) q.push({a.second, a.first});

while (!q.empty()) {

auto t = q.top(); q.pop();

res.append(t.first, t.second);

}

return res;

}

};

数组中所有缺失的数字

Given an array of integers where 1 ≤ a[i] ≤ n (n = size of array), some elements appear twice and others appear once.

Find all the elements of [1, n] inclusive that do not appear in this array.

Could you do it without extra space and in O(n) runtime? You may assume the returned list does not count as extra space.

Example:

Input:

[4,3,2,7,8,2,3,1]

Output:

[5,6]

将nums[i]置换到其对应的位置nums[nums[i]-1]上去，比如对于没有缺失项的正确的顺序应该是[1, 2, 3, 4, 5, 6, 7, 8]，而我们现在却是[4,3,2,7,8,2,3,1]，我们需要把数字移动到正确的位置上去，比如第一个4就应该和7先交换个位置，以此类推，最后得到的顺序应该是[1, 2, 3, 4, 3, 2, 7, 8]，我们最后在对应位置检验，如果nums[i]和i+1不等，那么我们将i+1存入结果res中即可，参见代码如下：

 class Solution {

public:

vector<int> findDisappearedNumbers(vector<int>& nums) {

vector<int> res;

for (int i = 0; i < nums.size(); ++i) {

if (nums[i] != nums[nums[i] - 1]) {

swap(nums[i], nums[nums[i] - 1]);

--i;

}

}

for (int i = 0; i < nums.size(); ++i) {

if (nums[i] != i + 1) {

res.push\_back(i + 1);

}

}

return res;

}

};

不等长链表数字相加之和

You are given two linked lists representing two non-negative numbers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Follow up:  
What if you cannot modify the input lists? In other words, reversing the lists is not allowed.

Example:

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

Output: 7 -> 8 -> 0 -> 7

这道题是之前那道[Add Two Numbers](http://www.cnblogs.com/grandyang/p/4129891.html" \t "http://www.cnblogs.com/grandyang/p/_blank)的拓展，我们可以看到这道题的最高位在链表首位置，如果我们给链表翻转一下的话就跟之前的题目一样了，这里我们来看一些不修改链表顺序的方法。由于加法需要从最低位开始运算，而最低位在链表末尾，链表只能从前往后遍历，没法取到前面的元素，那怎么办呢？我们可以利用栈来保存所有的元素，然后利用栈的后进先出的特点就可以从后往前取数字了，我们首先遍历两个链表，将所有数字分别压入两个栈s1和s2中，我们建立一个值为0的res节点，然后开始循环，如果栈不为空，则将栈顶数字加入sum中，然后将res节点值赋为sum%10，然后新建一个进位节点head，赋值为sum/10，如果没有进位，那么就是0，然后我们head后面连上res，将res指向head，这样循环退出后，我们只要看res的值是否为0，为0返回res->next，不为0则返回res即可，参见代码如下：

 class Solution {

public:

ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

stack<int> s1, s2;

while (l1) {

s1.push(l1->val);

l1 = l1->next;

}

while (l2) {

s2.push(l2->val);

l2 = l2->next;

}

int sum = 0;

ListNode \*res = new ListNode(0);

while (!s1.empty() || !s2.empty()) {

if (!s1.empty()) {sum += s1.top(); s1.pop();}

if (!s2.empty()) {sum += s2.top(); s2.pop();}

res->val = sum % 10;

ListNode \*head = new ListNode(sum / 10);

head->next = res;

res = head;

sum /= 10;

}

return res->val == 0 ? res->next : res;

}

};

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

**样例输入：**

5

1 2 3 4 5

**样例输出：**

1 3 5 2 4

对于普通的交换顺序，只要设两个指针分别从头跟尾扫描，当分别遇到不符合条件的位置时停止，然后交换位置，这样只要时间O(n)，空间O(1)，但是这样做会改变原来的顺序，要保持原来的位置，可以借用一个队列，先将一类数放到正确的位置上，再将队列里的数放到剩下的位置。时间仍为O(n)，但是要用额外空间，大小取决于输入。最坏情况下空间为O(n)。

int count = 0;

for (int i = 0; i < n; ++i)

{

if (v[i] % 2 == 1)

{

v[i - count] = v[i];

}

else

{

++count;

q.push(v[i]);}

}

for (int i = n - count; i < n; ++i)

{

v[i] = q.front();

q.pop();

}