题目和答案（Java 代码）

2016.06.03

- **二维数组中的查找**

在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

输入描述:

array： 待查找的二维数组

target：查找的数字

输出描述:

查找到返回 true，查找不到返回 false

代码：

```
class Solution {

    public:

    bool Find(vector<vector<int> > array,int target) {

        int row = array.size();//行数

        if(row==0) return false;

        int col = array[0].size();//列数

        if(col==0) return false;



        int i = 0, j = col - 1;//从右上角开始

        while(i<row && j>=0)

        {

            if(target==array[i][j])

                return true;//找到，立即退出
```

```
                else if(target>array[i][j]) i++;//向下一行
                else j--;//向前一列
            }
            if(i==row||j<0) return false;//没找到
            return true;//此句执行不到，只是为了编译没问题
        }
}
```

分析：时间复杂度 O(m+n)，m 和 n 分别是行数和列数。空间复杂度 O(1)。
思想是从右上角开始判断，每次判断可以排除一行或一列。

- **替换空格**

请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为 We Are Happy.则经过替换之后的字符串为 We%20Are%20Happy。
代码：

```
//length 为牛客系统规定字符串输出的最大长度，固定为一个常数
class Solution {
public:
    void replaceSpace(char *str,int length) {
        if(NULL==str) return;
        char *p = str;
        int size = 0;//记录str长度
        int blankNum = 0;//记录空格个数
        while(*p!='\0')//一直找到字符串尾部
        {
```

```
        if(*p==' ') blankNum++;
        size++;
        p++;
    }
    if(blankNum==0) return;//没有空格，直接返回

    int newSize = size - blankNum + blankNum*3;//计算新字符串大小
    //if(newSize>length) return;

    char *end = str + newSize;//
    *end = '\0';//最后一个字符置空
    end--;    //往前一步
    p = str + size - 1; //原字符串的最后一个字符处
    while(p>=str)
    {
        if(*p==' ')
        {//当前字符是空格，则替换
            *end-- = '0';
            *end-- = '2';
            *end-- = '%';
        }
        else
```

```
                {//否则，不替换

                    *end-- = *p;

                }

                p--;//依次向前

            }

        }

}
```

分析：时间复杂度 O(N)，N 为字符串长度。
思想是先遍历一次得到字符串的长度和空格个数，计算出替换后的字符串长度，这样会在原字符串后面空出一段；然后从后往前，碰到空格就替换。

鉴于义军的话：就在牛客网上的那个编辑器里写代码，不要在 VS 里面写，因为 VS 有自动补全功能，而面试的时候是没有自动补全的。
所以从现在起，不在 VS 中写代码了，可能会在 VS 里查看有哪些成员函数。

2016.06.04

- **从尾到头打印链表**

输入一个链表，从尾到头打印链表每个节点的值。

代码：

```
/**
*   struct ListNode {
*           int val;
*           struct ListNode *next;
*           ListNode(int x) :
*                   val(x), next(NULL) {
*           }
*   };
*/
class Solution {
public:
    vector<int> printListFromTailToHead(struct ListNode* head) {
        vector<int> vec;
        printTmp(head,vec);
        return vec;
    }
```

```
    void printTmp(ListNode *head, vector<int> &vec){
        if(NULL==head) return;
        printTmp(head->next,vec);
        vec.push_back(head->val);
    }
};
```

分析：递归，时间复杂度 O(N)，N 为链表长度。因为要返回一个 vector，所以必须另外写个函数，以便把 vector 作为参数。

- **重建二叉树**

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

代码：

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    struct TreeNode* reConstructBinaryTree(vector<int> pre,vector<int> in) {
        if(pre.size()==0 || in.size()==0) return NULL;
        int size = pre.size();
        return constructTmp(pre,in,0,size-1,0,size-1);
    }

    TreeNode * constructTmp(vector<int> &pre, vector<int> &in, int prei, int prej, int ini, int inj){
        if(prei>prej || ini>inj) return NULL;
        TreeNode * node = new TreeNode(pre.at(prei));
        int i = ini;
        while(in.at(i)!=pre.at(prei)) i++;
        int len = i - ini;
        node->left = constructTmp(pre,in,prei+1,prei+len,ini,i-1);
        node->right = constructTmp(pre,in,prei+len+1,prej,i+1,inj);
        return node;
```

```
    }
};
```

分析：时间复杂度 O(nlogn)，n 为节点个数。递归的思想。

2016.06.05

- **用两个栈实现队列**

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。 队列中的元素为 int 类型。

代码：

```
class Solution
{
public:
    void push(int node) {
        stack1.push(node);
    }

    int pop() {
        while(!stack1.empty()){
            stack2.push(stack1.top());
            stack1.pop();
        }
        int ret = stack2.top();
        stack2.pop();
        while(!stack2.empty()){
            stack1.push(stack2.top());
            stack2.pop();
        }
        return ret;
    }

private:
    stack<int> stack1;
    stack<int> stack2;
};
```

分析：push 时间复杂度为 O(1)，pop 时间复杂度为 O(n)。需要注意的是栈的操作 push、pop（并不返回元素）、top、empty。

- **旋转数组的最小数字**

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。
输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。

例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为 1。
NOTE：给出的所有元素都大于 0，若数组大小为 0，请返回 0。

代码：

```
class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        int size = rotateArray.size();
        if(size==0) return 0;
        int low = 0, high = size - 1, mid;
        while(low<high){
            if(rotateArray[low]<rotateArray[high]) break;
            else{
                mid = (low+high)/2;
                if(rotateArray[mid]>rotateArray[low]) low = mid;
                else if(rotateArray[mid]<rotateArray[mid-1]){
                    low = mid;
                    break;
                }else high = mid;

            }
        }
        return rotateArray[low];
    }
};
```

分析：线性扫描就不说了，此题的本意是利用二分查找，时间复杂度为 O(logn)。
思想主要是根据 arr[low]、arr[mid]、arr[high]的大小关系，判断最小元素在前半段还是后半段。需要注意的是红色代码处，这种情况发生在：5，6，1，2，3，4

此代码运行时间为 20ms，感觉还是有点慢哎！前面几题的运行时间都是 0 的。

下面用线性扫描试试，看看运行时间为多少：

```
class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        int size = rotateArray.size();
        if(size==0) return 0;
```

```
            int min = rotateArray[0];
            for(int i=1;i<size;i++)
                if(rotateArray[i]<min) min = rotateArray[i];
            return min;
        }
};
```

结果时间也是 20ms，呃呃呃。。。　不知什么情况了！ O(logn)和 O(n)的运行时间是一样的。

- **斐波那契数列**

大家都知道斐波那契数列，现在要求输入一个整数 n，请你输出斐波那契数列的第 n 项。

代码：

```
class Solution {
public:
    int Fibonacci(int n) {
        if(n==0) return 0;
        int tmp = 1;
        int ret = 0;
        for(int i=1;i<=n;i++){
            ret += tmp;
            tmp = ret - tmp;
        }
        return ret;
    }
};
```

分析：

1）拒绝用递归，因为会导致大量的重复计算。

2）初始条件为 f(0)=0,f(1)=1,f(2)=1，递推公式 f(n)=f(n-1)+f(n-2),n>2

3）为了少用一个变量，采用了红色代码部分。

- **跳台阶**

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

代码：

```
class Solution {
public:
    int jumpFloor(int n) {
        if(n<=0) return 0;
        int tmp = 1;
        int ret = 0;
```

```
            for(int i=1;i<=n;i++){
                ret += tmp;
                tmp = ret - tmp;
            }
            return ret;
        }
};
```

分析：递推（跟斐波那契数列一样）。设共有 f(n)种方法，则有：
1）如果第一次跳 1 级台阶，那么接下来有 f(n-1)种跳法；
2）如果第一次跳 2 级台阶，那么接下来有 f(n-2)种跳法；
第一次要么跳 1 级，要么跳 2 级，所以只用上面两种情况，故总跳法数 f(n)=f(n-1)+f(n-2)
初始条件：f(1)=1,f(2)=2

- **变态跳台阶**

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法

代码：

```
class Solution {
public:
    int jumpFloorII(int n) {
        if(n<=0) return 0;
        int tmp = 1;
        int ret = 0;
        for(int i=1;i<=n;i++){
            ret += tmp;
            tmp = ret;
        }
        return ret;
    }
};
```

分析：跟上一题一样道理，递推。
递推公式：f(n)=f(n-1)+f(n-2)+f(n-3)+…..+f(1)
初始条件：f(0)=1,f(1)=1,f(2)=2,f(3)=4…，这里加了一个 f(0)=1，使得更好的服从递推。

- **矩形覆盖**

我们可以用 2*1 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 2*1 的小矩形无重叠地覆盖一个 2*n 的大矩形，总共有多少种方法？

代码：

```
class Solution {
public:
```

```
    int rectCover(int n) {
        if(n<=0) return 0;
        int tmp = 1;
        int ret = 1;
        for(int i=2;i<=n;i++){
            ret += tmp;
            tmp = ret - tmp;
        }
        return ret;
    }
};
```

分析：递推公式 f(n)=f(n-1)+f(n-2)，f(1)=1,f(2)=2

- **二进制中 1 的个数**

输入一个整数，输出该数二进制表示中 1 的个数。其中负数用补码表示。

代码：

```
class Solution {
public:
     int   NumberOf1(int n) {
         int sum = 0;
         while(n){
             sum++;
             n &= (n-1);
         }
         return sum;
     }
};
```

分析：红色代码每执行一次，可以消去二进制中的一个 1。

- **数值的整数次方**

给定一个 double 类型的浮点数 base 和 int 类型的整数 exponent。求 base 的 exponent 次方。

代码：

```
class Solution {
public:
    double Power(double base, int exponent) {
        if((base<=0.000001 && base>=-0.000001) && exponent<0) return 0.0;
        if(exponent>0) return PowerTmp(base,exponent);
        else return 1.0/PowerTmp(base,-exponent);
    }
```

```
    double PowerTmp(double base, int ex){
        if(ex==0) return 1.0;
        if(ex==1) return base;
        double ret = Power(base,ex/2);
        if(ex&0x1) return ret*ret*base;
        else return ret*ret;
    }
};
```

分析：时间复杂度为 O(logn)，n 为指数。

1）首先判断 base 和 exponent 是否合法：0 的负数次幂不合法，0 的 0 次幂默认为 1.

2）考虑 exponent 正负，如果小于 0，返回倒数；

3）如果 exponent 为奇数，则 power(base,ex)=power(base,ex/2)*power(base,ex/2)*base;

如果 exponent 为偶数，则 power(base,ex)=power(base,ex/2)*power(base,ex/2)；

折半比循环快。

- **调整数组顺序使奇数位于偶数前面**

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。

代码：

```
class Solution {
public:
    void reOrderArray(vector<int> &array) {
        int size = array.size();
        if(size==0) return;
        int odd = 0, even = 0;//odd 为奇数下标，even 为第一个偶数下标

        while(array[even]&0x1) even++;//找到第一个偶数
        odd = even + 1;//从第一个偶数后面开始找到第一个奇数
        while(odd<size){
            while(odd<size && !(array[odd]&0x1)) odd++; //找到下一个奇数
            if(odd>=size) break;//如果没有奇数了，及时退出
            //将[even,odd-1]之间的偶数向后移一位
            int tmp = array[odd];
            for(int i=odd;i>even;i--)
                array[i] = array[i-1];
            array[even] = tmp;//把奇数放到前面
            even++;
            odd++;
```

```
            }
        }

};
```

分析：此题要是可以改变相对位置，则有时间复杂度为 O(n)、空间复杂度为 O(1)的方法。但是要保证相对位置不变，要么时间复杂度为 O(n)、空间复杂度为 O(n)，要么时间复杂度为 O(n*n)、空间复杂度为 O(1)。

本代码的时间复杂度为 O(n*n)，但运行时间为 0ms。。。那么此题的意义何在？

- **链表中倒数第 k 个结点**

输入一个链表，输出该链表中倒数第 k 个结点。

代码：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
    ListNode* FindKthToTail(ListNode* pListHead, unsigned int k) {
        if(NULL==pListHead || k==0) return NULL;

        ListNode *p = pListHead;
        int i = 1;
        while(i++<k){
            p = p->next;
            if(NULL==p) return NULL;//防止链表长度小于 k
        }
        ListNode *node = pListHead;
        while(p->next){//p->next 为空时，p 才指向最后一个节点
            node = node->next;
            p = p->next;
        }
        return node;
    }
};
```

分析：时间复杂度 O(n)，n 为节点个数。

思路：先让一个指针走 k-1 步，然后另一个指针从头开始，两个指针同步向后走，当第一个指针走到最后一个节点时，第二个指针正好是倒数第 k 个节点。
注意一下几个地方：
1）链表为空；
2）k 为 0
3）链表长度小于 k；
4）当第一个指针到达最后一个节点时，而不是指向空时，第二个指针才指向倒数第 k 个节点。

- **反转链表**

输入一个链表，反转链表后，输出链表的所有元素。

代码：

```
/*
struct ListNode {
	int val;
	struct ListNode *next;
	ListNode(int x) :
			val(x), next(NULL) {
	}
};*/
class Solution {
public:
	ListNode* ReverseList(ListNode* pHead) {
		if(NULL==pHead) return NULL;
		if(pHead->next==NULL) return pHead;
		ListNode *node = pHead->next;
		ListNode *head = ReverseList(node);
		node->next = pHead;
		pHead->next = NULL;
		return head;
	}
};
```

分析：递归实现。精髓在于如何返回头结点（原来的尾节点）：一旦找到尾节点，就一直返回它，不让它参与任何操作。
下面是进一步简化的代码：

```
class Solution {
public:
	ListNode* ReverseList(ListNode* pHead) {
		if(NULL==pHead) return NULL;
		if(pHead->next==NULL) return pHead;
		ListNode *head = ReverseList(pHead->next);
		pHead->next->next = pHead;
```

```
            pHead->next = NULL;
            return head;
        }
};
```

- **合并两个排序的链表**

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

代码：

```
/*
struct ListNode {
        int val;
        struct ListNode *next;
        ListNode(int x) :
                val(x), next(NULL) {
        }
};*/
class Solution {
public:
        ListNode* Merge(ListNode* pHead1, ListNode* pHead2){
            if(NULL==pHead1) return pHead2;
            if(NULL==pHead2) return pHead1;
            if(NULL==pHead1 && NULL==pHead2) return NULL;
            ListNode *p1 = pHead1, *p2 = pHead2, *pHead = NULL;//pHead 为归并后的头节点
            ListNode *p = new ListNode(-1);//申请一个节点，为了操作方便
            ListNode *q = p;//记住此申请节点，在最后要释放空间
            while(p1 && p2){
                if(p1->val<p2->val){
                    p->next = p1;
                    p1 = p1->next;
                }else{
                    p->next = p2;
                    p2 = p2->next;
                }
                p = p->next;
                if(NULL==pHead) pHead = p;//记住头节点
            }

            while(p1){
                p->next = p1;
```

```
            p1 = p1->next;
        }
        while(p2){
            p->next = p2;
            p2 = p2->next;
        }
        delete q;//释放申请的动态空间
        return pHead;
    }
};
```

分析：考查链表的归并。

思路：为了不单独选出头结点，特定义了一个节点 p，方便处理。指针 q 是为了记住申请空间的位置，便于之后释放。

- **树的子结构**

输入两颗二叉树 A，B，判断 B 是不是 A 的子结构。

代码：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class Solution {
public:
    bool HasSubtree(TreeNode* pRoot1, TreeNode* pRoot2){
        if(pRoot1==NULL || pRoot2==NULL) return false;
        return isSubtree(pRoot1,pRoot2);
    }

    bool isSubtree(TreeNode *pa, TreeNode *pb){
        if(pa==NULL) return false;//递归结束条件
        bool ret = false;//返回值
        if(pa->val==pb->val) ret = isSubtreeTmp(pa,pb);//节点值相同，继续判断左右子树
        if(ret) return true;//如果左右子树都相同，表示已找到，立即返回
        ret = isSubtree(pa->left,pb);//节点值不相同，在 A 树的左子树中寻找
        if(ret) return true;//如果找到，立即返回
        ret = isSubtree(pa->right,pb);//节点值不相同，在 B 树的右子树中寻找
```

```
            if(ret) return true;//如果找到，立即返回
            else return false;//上述都找不到时，返回 false
        }

        bool isSubtreeTmp(TreeNode *pa, TreeNode *pb){
            if(pb==NULL) return true;//为空，说明此子树已完全匹配
            if(pa==NULL && pb!=NULL) return false;//如果 A 树先空，而 B 树还没空，匹配失
败
            return pa->val==pb->val && isSubtreeTmp(pa->left,pb->left) &&
isSubtreeTmp(pa->right,pb->right);//如果节点值相同，继续匹配左子树和右子树
        }
};
```

分析：这题主要是搞清楚子结构的定义。

思路：先序遍历每一个节点，从 A 树根节点开始判断：

1）当前节点的值与 B 树根节点的值是否一样，若一样，执行 2)；否则，执行 3)；

2）继续判断 A 树左子树和 B 树左子树，以及 A 树右子树和 B 树右子树；

3）说明此节点一定不是子结构的根节点，则：

2.1）递归：令当前节点的左孩子为当前节点，执行 1)；

2.2）递归：令当前节点的右孩子为当前节点，执行 1)；

函数 isSubtree 中有很多 return 语句，主要是为了在找到子结构时能够立即返回，避免不必要的递归。

本题的“子结构”需要注意以下几点：

1）节点值可以重复；

2）树 B 只是树 A 的一部分，不一定非要到叶子节点，即下面的情况是满足子结构的：

```
      2                    2
    2   3                4   5
  4   5                    B 树
    6   7
    A 树
```

3）当树 A 不空、B 树空时，B 树也是 A 树的子结构。

- 二叉树的镜像

操作给定的二叉树，将其变换为源二叉树的镜像。

> 二叉树的镜像定义：源二叉树
>
> ```
>          8
>
>        /   \
>
>       6     10
> ```

```
        / \   / \

       5   7 9  11

       镜像二叉树

           8

         /   \

        10    6

       / \   / \

      11 9 7   5
```

代码：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/
class Solution {
public:
    void Mirror(TreeNode *pRoot) {
        if(pRoot==NULL) return;
        TreeNode *node = pRoot->left;
        pRoot->left = pRoot->right;
        pRoot->right = node;
        Mirror(pRoot->left);
        Mirror(pRoot->right);
    }
};
```

分析：递归地交换左右子树即可。

- **顺时针打印矩阵**

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字 1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

本题代码分成两步来说：

第一步：

```
class Solution {
public:
    vector<int> printMatrix(vector<vector<int> > matrix) {
        vector<int> vec;
        int row = matrix.size();//行数
        int col = matrix[0].size();//列数

        int k = 0;//控制循环输出的次数
        while(k<=row/2 && k<=col/2){
            int i = k, j = k;//每一次循环输出从 matrix[k][k]开始
            while(j<col-k) vec.push_back(matrix[i][j++]);//从“左上角向右上角”输出
            j--;i++;//j—因为上面 while 多加了 1；i++走向下一行
            while(i<row-k) vec.push_back(matrix[i++][j]);//从“右上角向右下角”输出
            i--;j--;//i—因为上面 while 多加了 1；j—走向左一列
            while(j>=k) vec.push_back(matrix[i][j--]);//从“右下角向左下角”输出
            j++;i--;//j++因为上面 while 多减了 1；i—走向上一行
            while(i>=k+1) vec.push_back(matrix[i--][j]);//从“左下角向左上角”输出
            k++;//准备进行下一轮
        }
        return vec;
    }
};
```

分析：一圈可以分成四步：

1）从左上角到右上角

2）从右上角到右下角

3）从右下角到左下角

4）从左下角到左上角

只要控制好每次输出的起点和终点即可：

1）从左上角到右上角的起点~终点：[k,k]~[k,col-k-1]

2）从右上角到右下角的起点~终点：[k+1,col-k-1]~[row-k-1,col-k-1]

3）从右下角到左下角的起点~终点：[row-k-1,col-k-2] ~[row-k-1,k]

4）从左下角到左上角的起点~终点：[row-k-2,k]~[k+2,k]

其中的 i，j 就是在起点和终点之间不断地变换。

来研究下 k 的取值范围，需要考虑多种特殊情况：

（1）row=0，col=0，即行数和列数都是 0；

（2）row=1 或者 col=1，即只有一行，或只有一列；
（3）row!=col，即矩阵不是方阵，这种情况比较普遍；
（4）每次 k 从主对角线依次开始，一圈之后，会输出两行、两列；
我们希望通过 k 的取值，把这些情况都包含进去：
（1）和（2）通过 “<=”来表达
（3）通过 “&&”来表达；
（4）通过“row/2”和“col/2”来表达；
所以综合起来，主循环条件就是：k<=row/2 && k<=col/2

上述代码能够保证不会少输出元素，但是很多情况会多输出元素，即某些元素被重复输出了。为了解决这个问题，需要加上下面几句：

第二步：

```
class Solution {
public:
    vector<int> printMatrix(vector<vector<int> > matrix) {
        vector<int> vec;
        int row = matrix.size();
        int col = matrix[0].size();

        int k = 0;
        while(k<=row/2 && k<=col/2){
            int i = k, j = k;
            if(i>=row-k) break;//(1)
            while(j<col-k) vec.push_back(matrix[i][j++]);
            j--;i++;
            if(j<k) break;//(2)
            while(i<row-k) vec.push_back(matrix[i++][j]);
            i--;j--;
            if(i<=k) break;//(3)
            while(j>=k) vec.push_back(matrix[i][j--]);
            j++;i--;
            if(j>=col-k-1) break;//(4)
            while(i>=k+1) vec.push_back(matrix[i--][j]);
            k++;
        }
        return vec;
    }
};
```

红色部分就是为了避免重复输出的，下面依次分析：
（1）这是为了防止重复上一轮循环中输出的元素。想想看，上一轮循环中，输出的最大一行（行数最大）是第 row-k-1 行（这里的 k 是上一轮的 k）。到了这一轮，我们要保证不能输出比第 row-k-1 还大的行，因为那些都是之前轮次输出过的。这一轮的 k 比上一轮的 k 大 1，所以此轮中只要 i>=row-k，说明此第 i 行已经上上一轮输出过了。一旦出现这种情况，立马跳出主循环。

（2）也是为了防止重复输出上一轮循环中输出的元素。上一轮中输出的最小列是第 k 列（上一轮的 k），这一轮输出的列不能小于上一轮的第 k 列。所以 j<k（此轮的 k）不能出现。

（3）防止这一轮刚才输出的行再次被输出。刚才输出的行（第 k 行）是从左上角到右上角，现在要输出的行（第 i 行）是右下角到左下角，那么这两行不能重复输出吧！所以不能 i<=k。

（4）防止这一轮刚才输出的列再次被输出。刚才输出的列（第 col-k-1 列）是从右上角到右下角，现在要输出的列（第 j 列）是从左下角到左上角，那么这两行也不能重复输出吧！所以不能 j>=col-k-1。

好了，经过上面四个“避免重复”，缺一不可，然后此题可以 AC 了。

上述讲解过程可能不太容易理解，但看的出来有很强的规律性，且代码简洁。主循环里面的四个小循环，每个小循环都是先计算好起点，然后判断是否会重复输出，最后再输出。一旦发现会重复输出，立马结束主循环。

- **包含 min 函数的栈**

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的 min 函数。

代码：

```
class Solution {
    vector<int> vec;
public:
    void push(int value) {
        vec.push_back(value);
    }
    void pop() {
        vec.pop_back();
    }
    int top() {
        return vec.at(vec.size()-1);
    }
    int min() {
        int size = vec.size();
        if(size==1) return vec.at(0);
        int min = vec.at(0);
        for(int i=1;i<size;i++)
            if(vec.at(i)<min) min = vec.at(i);
        return min;
    }
};
```

分析：此题本意在用两个栈，一个栈 m_data 用于存放数据，另一个栈 m_min 用于存放当

前最小值。比如：
1）插入 3 时，3 是当前最小值，m_data.push(3);m_min.push(3);　两者同时插入元素。只不过 m_min 插入的永远是当前最小值。
2）插入 4 时，3 还是当前的最小值，m_data.push(4);m_min.push(3);
3）插入 2 时，2 是当前最小值，m_data.push(2);m_min.push(2);
4）插入 1 时，1 是当前最小值，m_data.push(1);m_min.push(1);
5）获取当前最小值，则直接取 m_min 栈顶元素即可，即 m_min.top();
6）弹出 1 时，m_data.pop();m_min.pop();　两者同时弹出栈顶元素

《剑指 offer》上是这样写的，明显是以空间换时间的方法。而我没这样写，我的是用一个 vector 来保存元素，当获取最小值时，遍历 vector 获得最小值，时间复杂度为 O(N)，N 为栈中元素个数。

2016.06.09

- **栈的压入、弹出序列**

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列 1,2,3,4,5 是某栈的压入顺序，序列 4，5,3,2,1 是该压栈序列对应的一个弹出序列，但 4,3,5,1,2 就不可能是该压栈序列的弹出序列。

代码：

```
class Solution {
public:
    bool IsPopOrder(vector<int> pushV,vector<int> popV) {
        int pushSize = pushV.size();
        int popSize = popV.size();
        if(pushSize==0 || popSize==0 || pushSize!=popSize) return false;

        vector<int> stack;
        int top = 0;
        int i = 0, j = 0;
        while(i<pushSize){
            if(pushV[i]==popV[j]){
                j++;
                while(top>0 && j<popSize && stack[--top]==popV[j]) j++;
                if(j<popSize) top++;
            }else{
                stack.push_back(pushV[i]);
                top++;
            }
            i++;
        }
        if(top<=0) return true;
        else return false;
```

```
    }
};
```

分析：如果不允许改变输入的两个数组，那只有另外使用一个数组来当做栈，故空间复杂度为 O(n)。时间复杂度也为 O(n)。

- **从上往下打印二叉树**

从上往下打印出二叉树的每个节点，同层节点从左至右打印。

代码：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/
class Solution {
public:
    vector<int> PrintFromTopToBottom(TreeNode *root) {
        vector<int> vec;
        if(root==NULL) return vec;
        deque<TreeNode*> queue;
        queue.push_back(root);
        while(!queue.empty()){
            TreeNode * node = queue.front();
            queue.pop_front();
            vec.push_back(node->val);
            if(node->left!=NULL) queue.push_back(node->left);
            if(node->right!=NULL) queue.push_back(node->right);
        }
        return vec;
    }

};
```

分析：层次遍历。

- **二叉搜索树的后序遍历序列**

输入一个整数数组，判断该数组是不是某**二叉搜索树**的后序遍历的结果。如果是则输出 Yes，否则输出 No。假设输入的数组的任意两个数字都互不相同。

代码：

```
class Solution {
public:
    bool VerifySquenceOfBST(vector<int> sequence) {
        int size = sequence.size();
        if(size==0) return false;
        return isSequqenceOfBST(sequence,0,size-1);
    }

    bool isSequqenceOfBST(vector<int> vec, int s, int e){//s 为起点，e 为终点
        if(s>=e) return true;
        int tail = vec.at(e);
        int i = s;
        while(i<e && vec.at(i)<tail) i++;//找出左子树
        int j = i;
        while(j<e && vec.at(j)>tail) j++;//检验右子树是否都比根节点值大
        return j==e && isSequqenceOfBST(vec,s,i-1) && isSequqenceOfBST(vec,i,e-1);
//j==e 表示当前是否为 BST，isSequqenceOfBST(vec,s,i-1 判断左子树是否为 BST，
isSequqenceOfBST(vec,i,e-1)判断右子树是否为 BST。
    }
};
```

分析：因为是后序遍历，所以序列的最后一个元素是跟节点；然后再前面序列中找出左子树和右子树，分别判断当前序列、左子序列、右子序列分别是否为 BST。因为左子树的值都比根节点值小，右子树值都比根节点值大，所以很容易找出左子树和右子树。

- **二叉树中和为某一值的路径**

输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

代码：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/
class Solution {
public:
    vector<vector<int> > FindPath(TreeNode* root,int expectNumber) {
        vector<vector<int> > allPath;
        if(root==NULL) return allPath;
```

```
        vector<int> vec;
        findPathTmp(allPath,vec,root,expectNumber);
        return allPath;
    }

    void findPathTmp(vector<vector<int> > &allPath, vector<int> path, TreeNode *root,
int num){
        path.push_back(root->val);
        int sum = 0;
        for(int i=0;i<path.size();i++)
            sum += path[i];
        if(sum==num && root->left==NULL && root->right==NULL){//确保此节点是叶节
点
            vector<int> vec(path);
            allPath.push_back(vec);
        }

        if(root->left!=NULL) findPathTmp(allPath,path,root->left,num);//继续在左子树中
寻找
        if(root->right!=NULL) findPathTmp(allPath,path,root->right,num);//在右子树中寻
找
    }
};
```

分析：注意路径的终点必须是叶子节点，中间节点不行，比如：

```
      10
    5     12
  4   0
```

假设给定路径值为 15，则{10,15}并不满足要求，{10,15,0}才满足要求。

采用递归的思想，每到一个节点，就把此节点的值加入路径 path 中，然后判断 path 中的所有值的和是否等于给定值，若等于，则输出；否则继续遍历左子树和右子树。

2016.06.11

- **复杂链表的复制**

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点）。

代码：

```
/*
struct RandomListNode {
    int label;
    struct RandomListNode *next, *random;
```

```
    RandomListNode(int x) :
            label(x), next(NULL), random(NULL) {
    }
};
*/
class Solution {
public:
    RandomListNode* Clone(RandomListNode* pHead)
    {
        if(pHead==NULL) return NULL;
        RandomListNode *p = pHead;
        while(p!=NULL){//在原链表每个节点后面插入一个新节点
            RandomListNode *node = new RandomListNode(p->label);
            node->next = p->next;
            p->next = node;
            p = p->next->next;
        }

        p = pHead;
        while(p!=NULL){//设置新节点的 random 域
            if(p->random!=NULL)
                p->next->random = p->random->next;
            else p->next->random = NULL;
            p = p->next->next;
        }

        RandomListNode *newHead = pHead->next, *q;
        p = pHead;
        while(p!=NULL){//分离出新链表
            q = p->next;
            if(q==NULL) break;
            p->next = p->next->next;
            p = q;
        }
        return newHead;
    }
};
```

分析：时间复杂度为 O(n)，空间复杂度为 O(n)，n 为节点个数。分为三步：

1）在原链表中，在每个节点后面插入一个新节点，新节点的值为前面节点的值，即复制；

2）设置新节点的 random 域：对于每一个新节点 new，由其前驱节点 p 可以找到新节点的 random 域，即 new->random = p->random->next;

3）从链表中分离出新链表：新旧节点分来开来，即可得到新链表。

需注意两点：

（1）原链表的 random 域可能为空，故在设置新节点 random 域是要注意；
（2）分离出新链表时，要判断 q 是否为空。

- **二叉搜索树与双向链表**

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

代码：

```
/*
struct TreeNode {
	int val;
	struct TreeNode *left;
	struct TreeNode *right;
	TreeNode(int x) :
		val(x), left(NULL), right(NULL) {
	}
};*/
class Solution {
public:
	TreeNode* Convert(TreeNode* pRootOfTree){
		if(pRootOfTree==NULL) return NULL;
		TreeNode *pre = NULL;//先驱节点
		inOrderVisit(pRootOfTree,pre);//中序遍历，生成一个逆向的单链表
		//此时 pre 就是单链表的最后一个节点
		while(pre->left!=NULL){//逆向遍历单链表，并连接正序指针
			pre->left->right = pre;
			pre = pre->left;
		}
		return pre;//返回头节点
	}

	void inOrderVisit(TreeNode *root, TreeNode *&pre){
		if(root->left!=NULL) inOrderVisit(root->left,pre);//先左
		root->left = pre;
		pre = root;//及时更新前驱
		if(root->right!=NULL) inOrderVisit(root->right,pre);//后右
	}
};
```

分析：把二叉搜索树转变为有序双链表，利用中序遍历。在中序遍历过程中，记录前驱节点（即上一个访问的节点），这样就可以得到一个逆向的单链表。然后再逆向遍历单链表，将正序的指针也连上，就得到了一个双链表。

上面可以进一步简化：在遍历二叉树的过程中连接成双链表。其实在上面的代码中，已经知

道了前驱节点，那么前驱节点的后驱节点就是当前节点。所以代码简化如下：

```
class Solution {
public:
    TreeNode* Convert(TreeNode* pRootOfTree){
        if(pRootOfTree==NULL) return NULL;
        TreeNode *pre = NULL;
        inOrderVisit(pRootOfTree,pre);
        while(pre->left!=NULL) pre = pre->left;
        return pre;
    }

    void inOrderVisit(TreeNode *root, TreeNode *&pre){
        if(root->left!=NULL) inOrderVisit(root->left,pre);
        if(pre!=NULL) pre->right = root;//当前节点是前驱节点的后驱节点
        root->left = pre;//当前节点的前驱节点
        pre = root;
        if(root->right!=NULL) inOrderVisit(root->right,pre);
    }
};
```

两个代码的时间复杂度是一样的，不同在于：第一段代码在遍历二叉树过程中，没有连接正序链，而是在返回头结点时才连接；第二段代码在遍历二叉树的过程中就连接了正序链。

这里需要注意引用或指针的使用，比如下面的代码是有问题的：

```
class Solution {
public:
    TreeNode* Convert(TreeNode* pRootOfTree){
        if(pRootOfTree==NULL) return NULL;
        inOrderVisit(pRootOfTree,NULL);
        while(pre->left!=NULL) pre = pre->left;
        return pre;
    }

    void inOrderVisit(TreeNode *root, TreeNode **pre){
        if(root->left!=NULL) inOrderVisit(root->left,pre);
        if(pre!=NULL)(*pre)->right = root;
        root->left = *pre;
        *pre = root;
        if(root->right!=NULL) inOrderVisit(root->right,pre);
    }
};
```

即使使用了指针 pre，但因为其没有在主函数中定义，在返回到根节点时，其中的值已经不存在了，故而会导致错误。一定在主函数中定义 pre 变量（或者声明为全局变量）。

2016.06.12

- 字符串的排列

输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串 abc,则打印出由字符 a,b,c 所能排列出来的所有字符串 abc,acb,bac,bca,cab 和 cba。 结果请按字母顺序输出。字符串长度不超过 9(可能有字符重复),字符只包括大小写字母。

代码：

```
class Solution {
public:
    vector<string> Permutation(string str) {
        int size = str.size();
        vector<string> ret;
        if(size==0) return ret;
        set<string> perSet;//使用 set 来排序和去重
        for(int i=0;i<size;i++){
            swap(str,i,0);//选出第一个元素
            permutationTmp(str,perSet,1);//对剩下的 size-1 个元素全排列
            swap(str,i,0);//复原字符串
        }
        set<string>::iterator iter = perSet.begin();
        for(;iter!=perSet.end();iter++)//将 set 中元素复制给 vector，用于返回
            ret.push_back(*iter);
        return ret;
    }

    //perSet 要保存所用排列，所以引用传递；而 str 必须使用值传递，不能引用或指针
    void permutationTmp(string str, set<string> &perSet, int k){
        if(k==str.size()){
            perSet.insert(str);
            return;
        }
        for(int i=k;i<str.size();i++){
            swap(str,i,k);
            permutationTmp(str,perSet,k+1);
            swap(str,i,k);
        }
    }

    void swap(string &str, int i, int j){//交换元素
        char ch = str[i];
        str[i] = str[j];
        str[j] = ch;
```

```
    }
};
```

分析：简单的全排列问题。需要注意的是：字母可能重复，所以全排列中存在重复的字符串，要去重。本代码采用 set 容器来去重，而且因为 set 可以自动排序，所以也免去了自己去排序的过程。

- 数组中出现次数超过一半的数字

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为 9 的数组{1,2,3,2,2,2,5,4,2}。由于数字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2。如果不存在则输出 0。

代码：

```
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        int size = numbers.size();
        if(size==0) return 0;
        int count = 1;
        int num = numbers[0];
        for(int i=1;i<size;i++){
            if(num!=numbers[i]){
                count--;
                if(count==0){
                    num = numbers[i];
                    count = 1;
                }
            }else{
                count++;
            }
        }
        count = 0;
        for(int i=0;i<size;i++)
            if(num==numbers[i]) count++;
        if(count>size/2) return num;
        else return 0;
    }
};
```

分析：时间复杂度为 O(n)，空间复杂度为 O(1)，很不错的算法。设定两个变量：一个存放数字（设为 num），一个计数（设为 count），num 初始化为数组第一个元素，count 初始化为 1。从下一个元素 numbers[i]（0<i<size）开始遍历数组，如果 num!=numbers[i]，count 减一；若 count 变为 0，则令 num=numbers[i]，count=1。如果 num==numbers[i]，count 加一。那么最后 num 保存的值就是要找的值（如果存在的话）。

这样做的理由是：如果存在次数大于数组一半的数，则它出现的次数比其他所有元素出

现次数之和还要大。

需要注意的是：有可能不存在出现次数大于数组一半的数，所以最后还要判断 num 是否满足出现次数大于数组一半。

另一种方法是利用快速排序的划分思想：每次都能确定一个元素的最终位置，如果这个位置刚好是 size/2+1，那么它就是要找的数（如果存在的话）。这种算法的时间复杂度也是 O(n)，但相比于上面的算法，好理解些。如下：

代码（快排思想）：

```
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        int size = numbers.size();
        if(size==0) return 0;
        int num = partition(numbers);//找到有序数组中下标为 size/2+1 的元素
        int count = 0;
        for(int i=0;i<size;i++)//检查它出现的次数是否超过数组长度的一半
            if(num==numbers[i]) count++;
        if(count>size/2) return num;
        else return 0;
    }
    //非递归形式的划分
    int partition(vector<int> &numbers){
        int size = numbers.size();
        int low = 0, high = size - 1;
        while(low<high){
            int tmp = numbers[low];//以第一个元素为标兵
            while(low<high){
                while(low<high && numbers[high]>=tmp) high--;
                if(low>=high) break;//为了及时退出循环
                numbers[low] = numbers[high];
                low++;
                while(low<high && numbers[low]<tmp) low++;
                if(low>=high) break;//为了及时退出循环
                numbers[high] = numbers[low];
                high--;
            }
            numbers[low] = tmp;
            if(low==size/2+1) break;//找到有序数组中的下标为 size/2+1 的元素，结束划分
            else if(low>size/2+1){//继续划分左边
                high = low - 1;
                low = 0;
            }else{//继续划分右边
```

```
                low = high + 1;
                high = size - 1;
            }
        }
        return numbers[low];
    }
};
```

分析：为什么时间复杂度为 O(n)？因为如果数组元素排列是随机的，那么每次划分都能将数组均分为左右两半，这样下一次划分的规模就缩小了一半。n+n/2+n/4+n/8+….=n(1+1/2+1/4+1/8+….)=2n。所以平均时间复杂度为 O(n)。

- 最小的 K 个数

输入 n 个整数，找出其中最小的 K 个数。例如输入 4,5,1,6,2,7,3,8 这 8 个数字，则最小的 4 个数字是 1,2,3,4。

代码：

```
class Solution {
public:
    vector<int> GetLeastNumbers_Solution(vector<int> input, int k) {
        vector<int> vec;
        int size = input.size();
        if(size==0 || k<1 || k>size) return vec;
        partition(input,k);
        for(int i=0;i<k;i++)
            vec.push_back(input[i]);
        return vec;
    }

    void partition(vector<int> &numbers, int k){
        int size = numbers.size();
        int low = 0, high = size - 1;
        while(low<high){
            int tmp = numbers[low];//以第一个元素为标兵
            while(low<high){
                while(low<high && numbers[high]>=tmp) high--;
                if(low>=high) break;//为了及时退出循环
                numbers[low] = numbers[high];
                low++;
                while(low<high && numbers[low]<tmp) low++;
                if(low>=high) break;//为了及时退出循环
                numbers[high] = numbers[low];
                high--;
            }
```

```
                numbers[low] = tmp;
                if(low==k-1) break;//找到有序数组中的下标为 size/2+1 的元素，结束划分
                else if(low>k-1){//继续划分左边
                    high = low - 1;
                    low = 0;
                }else{//继续划分右边
                    low = high + 1;
                    high = size - 1;
                }
            }
        }
};
```

分析：还是快排的思想，平均时间复杂度为 O(n)。在划分的过程中，一旦确定了下标为 k-1 的元素位置，那么它之前的元素就是最小的 k 个元素。直接利用上一题的代码即可。

还可以用堆排序来解决，时间复杂度为 O(klogk+(n-k)logk)，因为 k 是个常数，所以时间复杂度为 O(nlogk)。如果 k 相对 n 来说很小的话，那么时间复杂度也是 O(n)。思路是这样的：顺序输入 k 个数，然后建立一个大顶堆；对于剩下的 n-k 个数，每输入一个数，如果这个数大于等于堆顶元素，则直接 pass；如果小于堆顶元素，则令它取代堆顶元素，然后向下调整堆。当遍历完数组后，堆中的 k 个数就是最小的 k 个数。代码就不写了。

- 连续子数组的最大和

HZ 偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为 8(从第 0 个开始,到第 3 个为止)。你会不会被他忽悠住？

代码：

```
class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array) {
        int size = array.size();
        if(size==0) return 0;
        int maxSum = array[0];
        int sum = array[0];
        for(int i=1;i<size;i++){
            if(sum<=0) sum = array[i];
            else sum += array[i];
            if(sum>maxSum) maxSum = sum;
        }
        return maxSum;
    }
```

```
};
```

分析：时间复杂度为 O(n)。思路是：定义两个变量 sum 和 maxSum，sum 用于记录连续子序列的和，maxSum 记录最大和。当遍历到一个元素 array[i]时，先判断 sum 是否小于 0，如果 sum<=0，说明前面的连续子序列的和为负数，那加上 array[i]一定比 array[i]小，所以抛弃 sum 的值，令其等于 array[i]；如果 sum>0，则加上 array[i]一定比 array[i]要大，所以 sum+=array[i]。
maxSum 在这个过程中，记录最大的 sum。

- 整数中 1 出现的次数（从 1 到 n 整数中 1 出现的次数）

求出 1~13 的整数中 1 出现的次数,并算出 100~1300 的整数中 1 出现的次数？为此他特别数了一下 1~13 中包含 1 的数字有 1、10、11、12、13，因此共出现 6 次,但是对于后面问题他就没辙了。ACMer 希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中 1 出现的次数。

代码：

```
class Solution {
public:
    int NumberOf1Between1AndN_Solution(int n){//假设 n=21345，方便解释
        if(n<1) return 0;
        if(n<10) return 1;//小于 10 时，只有一个 1
        string nStr = toString(n);//将 21345 转化为字符串
        int nSize = nStr.size();//长度为 5
        int po = pow(10,nSize-1);//计算最高位，10^4=10000
        int num1, num2;//num1 记录最高位含有 1 的个数，num2 记录剩下四位含 1 个数
        int first = nStr[0]-'0';//得到首位数字 2
        if(nStr[0]>'1') num1 = po;//首位大于 1 时，num1=10000
        else num1 = n - po*first + 1;//等于 1 时，num1=后四位数+1
        num2 = pow(10,nSize-2)*first*(nSize-1);//num2=排列组合
        return num1 + num2 + NumberOf1Between1AndN_Solution(n-po*first);//递归求剩下四位
    }

    string toString(int num){//转换 int 到字符串
        string str = "";
        while(num){
            str += num%10 + '0';
            num /= 10;
        }
        int size = str.size();
        for(int i=0;i<size/2;i++){
            char ch = str[i];
            str[i] = str[size-i-1];
            str[size-i-1] = ch;
```

```
        }
        return str;
    }
};
```

分析：找规律。假设 n=21345，把其分成两个部分：[1,1345]和[1346,21345]。

- 先看[1346,21345]这部分，最高位 2 是万位，万位是 1 的数有 10000~19999 共 10000 个；如果最高位不是大于 1 的数，而是等于 1（即 n=11345），则万位是 1 的个数为 1345+1=1346 个。 即程序中的 num1.
- [1346,21345]这 2 万个数分成[1346,11345]和[11346,21345]两部分。
- [1346,11345]这 10000 个数中，除去最高位，剩下四位中，选定一位为 1，其他三位可以从 0~9 随便选，故有 C(4,1) * 10^3=4000 个。
- 同理，[11346,21345]这 10000 个数中，除去最高位后，剩下四位中，也有 C(4,1) * 10^3=4000 个。

其实，[1346,21345]综合在一起的个数为：最高位*C(size-1,1)*10^(size-2) ，即程序中的 num2

- 还剩下[1,1345]这部分没计算，而这部分就可以递归地计算了。这就是为什么一上来就要这样划分的原因。

- **第一个只出现一次的字符位置**

在一个字符串(1<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符的位置。若为空串，返回-1。位置索引从 0 开始

代码：

```
class Solution {
public:
    int FirstNotRepeatingChar(string str) {
        int size = str.size();
        if(size==0) return -1;
        int num[52][2];
        for(int i=0;i<52;i++){//初始化
            num[i][0] = 0;
            num[i][1] = -1;//表示没出现
        }
        int tmp;
        for(int i=0;i<size;i++){//小写字母放在 num[26~51]，大写字母放在 num[0~25]
            if(str[i]>='a') tmp = 26 + str[i] - 'a';
            else tmp = str[i] - 'A';
            num[tmp][0]++;
            if(num[tmp][0]==1&&num[tmp][1]==-1) num[tmp][1] = i;
        }
        int min = size;//标记最早出现仅一次的字符下标
        bool isOne = false;
```

```
            for(tmp=0;tmp<52;tmp++)
                if(num[tmp][0]==1){
                    if(num[tmp][1]<min) min = num[tmp][1];
                    isOne = true;
                }
            if(isOne==false) return -1;
            return min;
        }
};
```

分析：主要思想是 hash。因为只有字母，所以可以大写或小写字母。定义数组大小为 52 即可。为了记录字符第一次出现的位置，定义一个二维数组 num[52][2]，num[i][0]表示此字母出现的次数，num[i][1]表示此字母第一次出现的次数。当遍历完字符数组后，就会得到每个字符出现的次数和第一次出现的位置。然后遍历 num 数组即可得到只出现一次且选出最先出现的字符位置。这里要防止字符串中没有只出现一次的字符，isOne 变量就是这个作用。

时间复杂度为 O(n)，n 为字符串长度。虽然题目说明了 n<=10000，其实 n 可以很大，只要不超过 int 的表示范围，本题的代码都是适用的。

- 把数组排成最小的数

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3，32，321}，则打印出这三个数字能排成的最小数字为 321323。

代码：

```
class Solution {
public:
    string PrintMinNumber(vector<int> numbers) {
        int size = numbers.size();
        if(size==0) return "";
        vector<string> numStr;
        for(int i=0;i<size;i++)
            numStr.push_back(toString(numbers[i]));
        sort(numStr.begin(),numStr.end(),cmp);
        string ret = "";
        for(int i=0;i<size;i++)
            ret += numStr[i];
        return ret;
    }

    static bool cmp(string str1, string str2){
        return str1<str2;                //str1+str2>str2+str1?0:1;
    }

    static string toString(int num){//int ----> string
        string str = "";
```

```
            while(num){
                str += num%10 + '0';
                num /= 10;
            }
            int size = str.size();
            for(int i=0;i<size/2;i++){
                char ch = str[i];
                str[i] = str[size-i-1];
                str[size-i-1] = ch;
            }
            return str;
        }
};
```

代码：其实就是字符串的大小比较。两个整数 n 和 m，转化为字符串后组合成 nm 和 mn，长度是一样的，直接按照字符串的比较规则来就行。

需要注意的是：

（1）sort 函数的比较函数的返回值问题：判断第一个参数是否应该移动到第二个参数的**前面**（The value returned indicates whether the element passed as first argument is considered to go before the second in the specific strict weak ordering it defines.）。如下面例子：

```
bool sortCmp(int a, int b){
    return a<b;
}
```

如果 a<b，则返回 true，表示 a 应该移动到 b 的前面，这就是从小到大排序。这也就是为什么网上说的“return a<b;就是从小到大排序；return a>b;就是从大到小排序”。

而 qsort 的比较函数于此不同，如下例子：

```
int qsortCmp(const void *a, const void *b){
    return *(int *)a - *(int *)b;
}
```

判断的是：第一个参数是否应该移动到第二个参数的**后面**。返回值大于 0，表示应该移动；小于 0，表示不移动。

综上，区别 sort 和 qsort 要点：

- 调用方式不同：sort(A,A+n,cmp);　　qsort(A,n,sizeof(A[0]),cmp);
- 比较函数形式不同：返回值、参数类型、默认比较规则

（2）sort 中的比较函数必须要声明为静态成员函数或全局函数，不能作为普通成员函数，否则会报错“ invalid use of non-static member function”。这是为什么？

因为：非静态成员函数是依赖于具体对象的，而 std::sort 这类函数是全局的，因此无法在 sort 中调用非静态成员函数。静态成员函数或者全局函数是不依赖于具体对象的，可以独立访问，无须创建任何对象实例就可以访问。同时静态成员函数不可以调用类的非静态成员。

此题时间复杂度 O(nlogn)，空间复杂度为 O(n)，n 为数组元素个数。

- 丑数

把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。例如 6、8 都是丑数，但 14 不是，因为它包含因子 7。 习惯上我们把 1 当做是第一个丑数。求按从小到大的顺序的第 N 个丑数。

代码：

```
class Solution {
public:
    int GetUglyNumber_Solution(int index) {
        if(index<1) return 0;
        int *ugly = new int[index];//存储丑数
        ugly[0] = 1;//第一个丑数是 1
        int i = 0, j = 0, k = 0;
        int count = 1;
        int min = 1;
        while(count<index){
            min = getMinist(2*ugly[i],3*ugly[j],5*ugly[k]);
            ugly[count] = min;
            while(2*ugly[i]<=min) i++;//确定下一个 i 值
            while(3*ugly[j]<=min) j++;
            while(5*ugly[k]<=min) k++;
            count++;
        }
        delete[] ugly;
        return min;
    }
    int getMinist(int a, int b, int c){//获取三者最小值
        int min = a > b ? b : a;
        min = min > c ? c : min;
        return min;
    }
};
```

分析：关键思想——后一个丑数是前面丑数乘以 2、3 或 5 得到的。然后就是要按顺序得到每一个丑数。比如说已经求出了 n 个丑数 ugly[0,n-1]，现在要求 ugly[n]，那么每个数 ugly[i] 乘以 2 都会得到一个数，只有大于 ugly[n-1]的数才可能是下一个丑数。在寻找丑数的过程中，可以记录 ugly[i]*2<=ugly[n-1]的最大的 i，这样就不用每个数都乘以 2 了。同样，对于 3,5 也是这样。从乘 2、乘 3、乘 5 的结果中找出一个最小的，它就是当前要找的丑数。【描述的不好理解，还是看剑指 offer 吧！】

时间复杂度 O(n)，空间复杂度 O(n)，n 为要求的第 n 个丑数。

- 数组中的逆序对

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

代码：

```
class Solution {
public:
    int InversePairs(vector<int> data) {
        int size = data.size();
        if(size<2) return 0;
        int *dataTmp = new int[size];//申请动态数组，辅助归并排序
        int count = 0;//记录逆序对的个数
        mergerSort(data,dataTmp,0,size-1,count);
        delete[] dataTmp;
        return count;
    }

    void mergerSort(vector<int> &data, int *dataTmp, int start, int end, int &count){
        if(start>=end) return;//start=end 的时候就可以返回了（只有一个元素）
        int mid = (start+end)/2;
        mergerSort(data,dataTmp,start,mid,count);//归并左边
        mergerSort(data,dataTmp,mid+1,end,count);//归并右边
        int i = start, j = mid + 1, k = start;
        while(i<=mid&&j<=end){
            if(data[i]>data[j]){
                dataTmp[k] = data[j++];
                count += mid - i + 1;//核心部位
            }else dataTmp[k] = data[i++];
            k++;
        }
        while(i<=mid) dataTmp[k++] = data[i++];
        while(j<=end) dataTmp[k++] = data[j++];
        for(i=start;i<=end;i++)
            data[i] = dataTmp[i];
    }
};
```

分析：O(n*n)复杂度的方法都能想到，但我们不能满足于此。以《剑指 offer》上的思想，用归并排序来解决。在归并排序的过程中，前一半 data[start,mid]和后一半 data[mid+1,end]，如果前一半的 data[i]大于后一半的 data[j]，那说明前一半在 data[i]之后的元素全大于 data[j]，这样关于 data[j]的逆序对就用 end-i+1 个了。

- 两个链表的第一个公共结点

输入两个链表，找出它们的第一个公共结点。

代码：

```
/*
struct ListNode {
	int val;
	struct ListNode *next;
	ListNode(int x) :
			val(x), next(NULL) {
	}
};*/
class Solution {
public:
	ListNode* FindFirstCommonNode( ListNode *pHead1, ListNode *pHead2) {
		if(pHead1==NULL || pHead2==NULL) return NULL;
		ListNode *p = pHead1, *q = pHead2;
		int len1 = 0, len2 = 0;
		while(p || q){
			if(p!=NULL){
				len1++;
				p = p->next;
			}
			if(q!=NULL){
				len2++;
				q = q->next;
			}
		}
		p = pHead1;
		q = pHead2;
		int dis = len1>len2?(len1-len2):(len2-len1);
		int i = 0;
		if(len1<len2)
			while(i++<dis) q = q->next;
		if(len1>len2)
			while(i++<dis) p = p->next;
		while(p&&q){
			if(p==q) break;
			p = p->next;
			q = q->next;
		}
		if(p==NULL || q==NULL) return NULL;
		else return q;
	}
};
```

分析：比较常见的算法就是求出两个链表的长度差 dis，然后令长链表指针先走 dis 步，然后两个链表指针同步走，当两个指针相等时，即为第一个公共节点。时间复杂度为 O(n)。

看到一个更屌的代码，不用计算长度差。虽然时间复杂度也是 O(n)，但代码简单了许多，也不好理解。

```
ListNode* FindFirstCommonNode( ListNode *pHead1, ListNode *pHead2) {
    ListNode *p1 = pHead1;
    ListNode *p2 = pHead2;
    while(p1!=p2){
        p1 = (p1==NULL ? pHead2 : p1->next);
        p2 = (p2==NULL ? pHead1 : p2->next);
    }
    return p1;
}
```

分析：其实还是长度差的原理，只是没有显式计算。这个代码的原理，画个图就好理解了。比如：长链表长度为 7，短链表长度为 5，公共长度为 3。此代码遍历的节点数跟上面代码是一样的，但代码长度减少了很多，是个不错的代码。

- **二叉树的深度**

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

代码：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/
class Solution {
public:
    int TreeDepth(TreeNode* pRoot)
    {
        if(pRoot==NULL) return 0;
        int left = TreeDepth(pRoot->left);
        int right = TreeDepth(pRoot->right);
        return (left>right?left:right) + 1;
    }
};
```

分析：递归。

- 平衡二叉树

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

代码：

```
class Solution {
public:
    bool IsBalanced_Solution(TreeNode* pRoot) {
        if(pRoot==NULL) return true;

        if(TreeDepth(pRoot)==-1) return false;
        else return true;
    }

    int TreeDepth(TreeNode* pRoot)
    {
        if(pRoot==NULL) return 0;
        int left = TreeDepth(pRoot->left);
        if(left==-1) return -1;
        int right = TreeDepth(pRoot->right);
        if(right==-1) return -1;
        if(left-right>1 || left-right<-1) return -1;
        else return (left>right?left:right) + 1;
    }
};
```

分析：利用上一题的获得树的深度思想，获得左右子树的高度，然后判断是否为平衡二叉树。

- 数组中只出现一次的数字

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

代码：

```
class Solution {
public:
    void FindNumsAppearOnce(vector<int> data,int* num1,int *num2) {
        int size = data.size();
        if(size<2 || num1==NULL || num2==NULL) return;

        int tmp = 0;
        for(int i=0;i<size;i++)//抑或，得到两个不同元素的抑或值
            tmp ^= data[i];
        int bit = 1;//记录最低位 1 所在位置
        while(tmp){//找出最低位 1
            if(tmp&0x1) break;
```

```
            bit = bit<<1;
            tmp = tmp>>1;
        }
        *num1 = 0;
        *num2 = 0;
        for(int i=0;i<size;i++)//bit 会将数组分成两半
            if(data[i]&bit) *num1 ^= data[i];
            else *num2 ^= data[i];
    }
};
```

分析：两个相同数的抑或为 0，所以将所有元素抑或到一起，结果就是那两个不同数的抑或值。找到这个值二进制中最低位 1 所在位置，然后用这一位来划分数组：此位为 1 的元素划分到一组；为 0 的元素划分到一组。这样，相同的元素肯定被划分到一组中，那两个不同的元素被划分到不同的组中。此时分别对两个数组抑或，即可得到两个值。

- 数字在排序数组中出现的次数

统计一个数字在排序数组中出现的次数。

代码：

```
class Solution {
public:
    int GetNumberOfK(vector<int> data ,int k) {
        int size = data.size();
        if(size==0) return 0;
        int first = findFirst(data,k,0,size-1);
        int last = findLast(data,k,0,size-1);
        if(low==-1||high==-1) return 0;//只要有一个返回-1，表示数组中没有此元素
        return high-low+1;
    }

    int findFirst(vector<int> &data, int &k, int low, int high){
        if(low>high) return -1;
        if(data[low]==k) return low;//既可以提前结束递归，又防止下面的 mid-1 越界
        int mid = (low+high)/2;
        if(data[mid]==k){
            if(data[mid-1]==k) high = mid - 1;
            else return mid; //前一个元素不等于 k，表示 mid 就是第一个值等于 k 的位置
        }else if(data[mid]>k) high = mid - 1;
        else low = mid + 1;
        return findFirst(data,k,low,high);
    }

    int findLast(vector<int> &data, int &k, int low, int high){
```

```
            if(low>high) return -1;
            if(data[high]==k) return high;
            int mid = (low+high)/2;
            if(data[mid]==k){
                if(data[mid+1]==k) low = mid + 1;
                else return mid;
            }else if(data[mid]<k) low = mid + 1;
            else high = mid - 1;
            return findLast(data,k,low,high);
        }
};
```

分析：数组有序，可以利用二分查找。找到值等于 k 的第一个元素位置 first 和最后一个元素位置 last，那么 last-first+1 就是其个数。不过，要把寻找 first 和 last 的过程分开。寻找 first 的过程中，每次都能排除一般的数据，所以时间复杂度为 O(logn)；同理，寻找 last 的时间复杂度也是 O(logn)。所以总的时间复杂度为 O(logn)。

2016.06.16

- 和为 S 的连续正数序列

小明很喜欢数学,有一天他在做数学作业时,要求计算出 9~16 的和,他马上就写出了正确答案是 100。但是他并不满足于此,他在想究竟有多少种连续的正数序列的和为 100(至少包括两个数)。没多久,他就得到另一组连续正数和为 100 的序列:18,19,20,21,22。现在把问题交给你,你能不能也很快的找出所有和为 S 的连续正数序列? Good Luck!

输入：一个正整数 S

输出：输出所有和为 S 的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序

代码 1：

```
class Solution {
public:
    vector<vector<int> > FindContinuousSequence(int sum) {
        vector<vector<int> > seq;
        if(sum<=1) return seq;
        for(int m=1;m<=sum/2+1;m++){
            int tmp = 2*sum + m*(m-1);
            int n = sqrt(tmp);
            if(n*(n+1)==tmp){//加入结果集
                vector<int> vec;
                for(int j=m;j<=n;j++)
                    vec.push_back(j);
                seq.push_back(vec);
            }
        }
        return seq;
```

```
    }
};
```

分析：利用数学公式。连续序列 m,m+1,m+2,m+3,…,n 累加和为(n+m)(n-m+1)/2，令其等于 S，即(n+m)(n-m+1)=2S，进一步 n(n+1) – m(m-1)=2S。此时，我们从 m=1 遍历到 m=sum/2+1，对于每一个 m，计算出 n=sqrt(2S+m(m-1))，然后检验 n(n+1)= 2S+m(m-1)是否成立，若成立，则输出。

代码 2：

```
class Solution {
public:
    vector<vector<int> > FindContinuousSequence(int sum) {
        vector<vector<int> > seq;
        if(sum<1) return seq;
        int low = 1, high = 2;
        int mid = sum/2 + 1;
        int curSum = low + high;
        while(high<=mid){
            if(curSum==sum)
                printSequence(seq,low,high);
            while(curSum>sum && low<high){
                curSum -= low;
                low++;
                if(curSum==sum) printSequence(seq,low,high);
            }
            high++;
            curSum += high;
        }
        return seq;
    }
    void printSequence(vector<vector<int> > &seq, int low, int high){
        vector<int> vec;
        for(int i=low;i<=high;i++)
            vec.push_back(i);
        seq.push_back(vec);
    }
};
```

分析：《剑指 offer》上的思想，不多说。

比较两个方法，虽然两种方法的时间复杂度为均为 O(sum)，但显然方法一的效率高一些，因为它只遍历到 sum/2 一次，而方法二中的 low 和 high 都要增加到 sum/2，相当于遍历到 sum/2 两次了。

- 和为 S 的两个数字

输入一个递增排序的数组和一个数字 S，在数组中查找两个数，使得他们的和正好是 S，如

果有多对数字的和等于 S，输出两个数的乘积最小的。

代码：

```
class Solution {
public:
    vector<int> FindNumbersWithSum(vector<int> array,int sum) {
        vector<int> ret;
        int size = array.size();
        if(size<2) return ret;
        int low = 0, high = size - 1;
        while(low<high){
            int tmp = array[low] + array[high];
            if(tmp==sum){
                ret.push_back(array[low]);
                ret.push_back(array[high]);
                break;
            }else if(tmp<sum) low++;
                else high--;
        }
        return ret;
    }
};
```

分析：两个指针，一个 low 从前往后，一个 high 从后往前，如果当前和等于 sum，直接输出；如果小于 sum，low++；如果大于 sum，high--。题目中要求输出积最小的一对，采用这样方法，第一次遇到的就是积最小的一对。所以，一旦遇到，立即返回。

- 左旋转字符串

汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的字符序列 S，请你把其循环左移 K 位后的序列输出。例如，字符序列 S=”abcXYZdef”,要求输出循环左移 3 位后的结果，即“XYZdefabc”。是不是很简单？OK，搞定它!

代码：

```
class Solution {
public:
    string LeftRotateString(string str, int n) {
        int size = str.size();
        if(size<2 || n<1) return str;
        if(n>=size) n = n%size;
        if(n==0) return str;
        reverseString(str,0,n-1);
        reverseString(str,n,size-1);
        reverseString(str,0,size-1);
```

```
            return str;
        }

        void reverseString(string &str, int s, int e){
            if(s==e) return;
            int mid = (s+e)/2, tmp;
            for(int i=s,j=0;i<=mid;i++,j++){
                tmp = str[i];
                str[i] = str[e-j];
                str[e-j] = tmp;
            }
        }
};
```

分析：不管是循环左移还是循环右移，只要旋转三遍即可。三次旋转依次是：[0,n-1]、[n,size-1]、[0,size-1]。此题需要注意的是 n 的取值，可能小于 0，可能大于 size。

- 翻转单词顺序列

牛客最近来了一个新员工 Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事 Cat 对 Fish 写的内容颇感兴趣，有一天他向 Fish 借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了，正确的句子应该是“I am a student.”。Cat 对一一的翻转这些单词顺序可不在行，你能帮助他么？

代码：

```
class Solution {
public:
    string ReverseSentence(string str) {
        int size = str.size();
        if(size<1) return str;
        reverseString(str,0,size-1);
        int s, e, i = 0;
        while(i<size){
            while(i<size&&str[i]==' ') i++;
            s = i;
            while(i<size&&str[i]!=' ') i++;
            e = i;
            reverseString(str,s,e-1);
        }
        return str;
    }

    void reverseString(string &str, int s, int e){
        if(s>=e) return;
        int mid = (s+e)/2, tmp;
```

```
            for(int i=s,j=0;i<=mid;i++,j++){
                tmp = str[i];
                str[i] = str[e-j];
                str[e-j] = tmp;
            }
        }
};
```

分析：跟上一题一样，关于字符串翻转的。要注意的是：只是句子翻转了，每个单词并没有翻转。所以，首先翻转整个句子，然后依次翻转每个单词即可。

- 扑克牌顺子

LL 今天心情特别好,因为他去买了一副扑克牌,发现里面居然有 2 个大王,2 个小王(一副牌原本是 54 张^_^)...他随机从中抽出了 5 张牌,想测测自己的手气,看看能不能抽到顺子,如果抽到的话,他决定去买体育彩票,嘿嘿!！“红心 A,黑桃 3,小王,大王,方片 5”,“Oh My God!”不是顺子.....LL 不高兴了,他想了想,决定大\小王可以看成任何数字,并且 A 看作 1,J 为 11,Q 为 12,K 为 13。上面的 5 张牌就可以变成“1,2,3,4,5”(大小王分别看作 2 和 4),“So Lucky!”。LL 决定去买体育彩票啦。 现在,要求你使用这幅牌模拟上面的过程,然后告诉我们 LL 的运气如何。为了方便起见,你可以认为大小王是 0。

输入：一个数组，里面有五个数，表示 LL 抽到的五张牌

输出：布尔值，为 true 表示可以买彩票，为 false 表示不能买

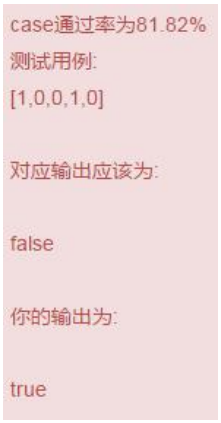
代码：

```
class Solution {
public:
    bool IsContinuous( vector<int> numbers ) {
        int size = numbers.size();
        if(size!=5) return false;
        sort(numbers.begin(),numbers.end());
        int zero = 0, i = 0;
        while(numbers[i]==0){
            zero++;
            i++;
        }
        int minus = 0;
        i++;
        while(i<size){
            if(numbers[i]==numbers[i-1]){//专门针对有重复的非 0 元素出现情况
                minus = 10;
                break;
            }
            minus += numbers[i] - numbers[i-1] - 1;
            i++;
        }
```

```
            if(minus<=zero) return true;
            else return false;
        }
};
```

分析：本题没有给定输入输出说明，虽然从给定的函数能够猜出，但给人题意不清的感觉。一开始以为要求概率呢！5 张牌要组成顺子，而且还有若干个 0（大小王），那就要求非 0 元素的差值（从小到大排序后的连续两个数的差值-1）不能大于 0 元素的个数。

题目中说了有 2 个大王和 2 个小王，好！尼玛，怎么测试用例还能出现 2 个 1，这能叫什么牌？坑爹的测试用例。



- 孩子们的游戏(圆圈中最后剩下的数)

每年六一儿童节,NowCoder 都会准备一些小礼物去看望孤儿院的小朋友,今年亦是如此。HF 作为 NowCoder 的资深元老,自然也准备了一些小游戏。其中,有个游戏是这样的:首先,让小朋友们围成一个大圈。然后,他随机指定一个数 m,让编号为 0 的小朋友开始报数。每次喊到 m 的那个小朋友要出列唱首歌,然后可以在礼品箱中任意的挑选礼物,并且不再回到圈中,从他的下一个小朋友开始,继续 0...m-1 报数....这样下去....直到剩下最后一个小朋友,可以不用表演,并且拿到 NowCoder 名贵的“名侦探柯南”典藏版(名额有限哦!!^_^)。请你试着想下,哪个小朋友会得到这份礼品呢?

输入：人数 n 和数 m （报数：0,1,2,。。。，m-1，报到 m-1 的人出队）

输出：最后剩下的人的位置（范围[0,n-1]）

代码 1：

```
struct ListNode{
    int val;
    ListNode *next;
    ListNode(int i){
        val = i;
        next = NULL;
    }
};

class Solution {
```

```
public:
    int LastRemaining_Solution(unsigned int n, unsigned int m){
        if(n==0 || m==0) return -1;
        ListNode *head, *p;
        for(int i=0;i<n;i++){//构造循环链表
            ListNode *node = new ListNode(i);
            node->next = NULL;
            if(i==0){//第一个节点
                head = node;
                p = node;
            }else {//其他节点
                p->next = node;
                p = node;
            }
        }
        p->next = head;//令尾指针指向头结点
        head = p;//令 head 指向尾节点，便于接下来的删除操作
        unsigned int step;
        unsigned int tmpM;
        unsigned int left = n;//链表中还剩下的人数
        while(true){
            step = 1;
            p = head->next;//p 为 head 的后驱节点，也是将要删除的节点
            if(p==head) break;//此时只剩一个节点了，立即退出
            if(m%left) tmpM = m%left;//为了避免不必要的走步
            else tmpM = left;//使得每次走的步数不超过剩下人数 left
            while(step<tmpM){//继续走
                head = p;
                p = p->next;
                step++;
            }
            head->next = p->next;//将节点 p 分离出来
            delete p;//删除 p，防止内存泄露
        }
        return head->val;
    }
};
```

分析：利用循环链表模拟游戏。首先申请含有 n 个节点的链表，令尾指针指向头结点，构成循环链表。然后根据 m，每次从链表中删除一个节点，直至只剩下一个节点。有个小技巧：令每次走步数不超过剩余人数，tmpM 就是这个作用。

时间复杂度为 O(n*m)，因为每次要走 m(m<n)步，要进行 n-1 次。空间复杂度为 O(n)。比较**奇葩**的是：我定义的 ListNode 结构，在编译时，提示重定义。原来在后台中，有个 Solution.h 头文件，里面已经定义了 ListNode。这表示用户还不能随便定义自己的结构，这个问题是系统不完善的地方。

另一种解法就是《剑指 offer》上说的，通过一段数学推导得出一个递推公式如下：
f(n,m)=0　　n=0
f(n,m)=(f(n-1,m)+m)%n　　n>0

代码很简洁，如下：

```
class Solution {
public:
    int LastRemaining_Solution(unsigned int n, unsigned int m){
        if(n<1 || m<1) return -1;
        int last = 0;
        for(int i=2;i<=n;i++)
            last = (last+m)%i;
        return last;
    }
};
```

- 求 1+2+3+...+n

求 1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case 等关键字及条件判断语句（A?B:C）。

代码 1（变相的递归）：

```
class Solution {
public:
    int Sum_Solution(int n) {
        int tmp = n;
        n && (tmp += Sum_Solution(n-1));
        return tmp;
    }
};
```

分析：如果用递归，而 if 又不能用，怎么办？利用逻辑运算的与、或、非。红色代码句还可以用!n || (tmp += Sum_Solution(n-1));来代替。

代码 2（变相的构造函数）：

```
class Temp{
public:
    static int sum;
    static int count;
    Temp(){
        count++;
        sum += count;
```

```
    }
};

int Temp::sum = 0;
int Temp::count = 0;

class Solution {
public:
    int Sum_Solution(int n) {
        Temp::count = 0;
        Temp::sum = 0;
        Temp * arr = new Temp[n];
        delete[] arr;
        return Temp::sum;
    }
};
```

分析：不能像书上那样写 main 函数，那么就再写一个类，让 Solution 类的 Sum_Solution 函数作为主函数。要注意的是，在 Sum_Solution 中要将 Temp::count 和 Temp::sum 置 0，否则就是通不过，不知道为什么，结果总是偏大。按说我只是在 Sum_Solution 函数中构造了 n 个对象，为什么结果会偏大呢？应该是后台系统的问题吧！因为在自己的 vs 编译器中是正确的。

代码 3（函数指针）：

```
class Solution {
public:
    typedef int (*fun)(int);
    static int Sum_Solution_Terminal(int n){
        return 0;
    }
    static int Sum_Solution(int n) {
        fun f[2] = {Sum_Solution_Terminal,Sum_Solution};
        return n+f[!!n](n-1);
    }
};
```

分析：通过!!n 来调用不同的函数。类似的方法还可用于虚函数。

注意：虽然题目中说了不能用乘除等，其实系统根本不检测，直接 return n*(n+1)/2;也能通过。看的出，系统很垃圾，或者说很随意。

- **不用加减乘除做加法**

写一个函数，求两个整数之和，要求在函数体内不得使用+、-、*、/四则运算符号。

代码（非递归，即循环形式）：

```
class Solution {
public:
    int Add(int a, int b){
        if(a==0) return b;
        if(b==0) return a;
        while(b){
            int tmp = a;
            a = a^b;
            b = tmp&b;
            b = b<<1;
        }
        return a;
    }
};
```

分析：不用加减乘除做加法运算，一般采用位运算，因为 CPU 所有运算都是二进制的运算。二进制运算：0+1=1,1+0=1,0+0=0,1+1=0（有个进位），这个运算就是抑或；进位的运算可以看做“与”，因为只有都是 1 的时候才会进位。举个例子：a=5=0101，b=3=0011，则 a^b=0110, a&b=0001， a^b 是结果，但 a&b 不等于 0，表示有进位，得将进位也加到结果上。进位应该加到左边一位上，即将 a&b 的结果左移一位，变成 0010，然后与 a^b 的结果再抑或，得到 0110^0010=0100=a(记为 a)，两者再与，0110&0010=0010=b(记为 b)，不为 0，继续移位、抑或、与，直到 b（与的结果）变成 0，此时的 a 就是要求的和。

代码 2（递归形式）：

```
class Solution {
public:
    int Add(int num1, int num2){
        if(num1==0) return num2;
        if(num2==0) return num1;
        return Add(num1^num2,(num1&num2)<<1);
    }
};
```

2016.06.18

- 把字符串转换成整数

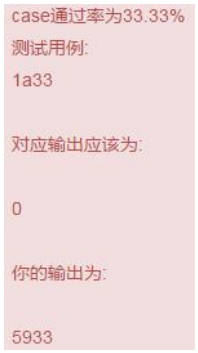
将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。

代码：

```
class Solution {
public:
    int StrToInt(string str) {
        int size = str.size();
        if(size==0) return 0;
```

```
            int i = 0;
            bool less0 = false;
            if(str[0]=='+' || str[0]=='-'){
                i++;//第一个字符是正负号
                if(str[0]=='-') less0 = true;//负数
            }
            int ret = 0;
            while(i<size && str[i]>='0' && str[i]<='9'){
                ret = ret*10 + str[i] - '0';
                i++;
            }
            if(i<size) return 0;//含有非数字字符
            if(less0) return -ret;//负数
            else return ret;//正数
        }
};
```

分析：没什么好说的，考点是考虑多种情况（正负号、含有非数字符号）。不用考虑前导空格和后导空格。



- 数组中重复的数字

在一个长度为 n 的数组里的所有数字都在 0 到 n-1 的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。 例如，如果输入长度为 7 的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字 2 或者 3。

代码：

```
class Solution {
public:
    // Parameters:
    //        numbers:     an array of integers
    //        length:      the length of array numbers
    //        duplication: (Output) the duplicated number in the array number
    // Return value:       true if the input is valid, and there are some duplications in the array number
```

```
    //                                  otherwise false
    bool duplicate(int numbers[], int length, int* duplication) {
        if(numbers==NULL || length<1) return false;
        int *arr = new int[length];
        int i;
        for(i=0;i<length;i++)
            arr[i] = 0;
        for(i=0;i<length;i++){
            arr[numbers[i]]++;
            if(arr[numbers[i]]>1) break;
        }
        delete[] arr;
        if(i==length) return false;
        *duplication = numbers[i];
        return true;
    }
};
```

分析：桶的应用。时间复杂度 O(n)，空间复杂度 O(n)。

注：题目跟以前不一样的是，有了注释。难道是换人了（这题是另外一个人出的）？

代码 2：

```
bool duplicate(int numbers[], int length, int* duplication) {
    if(numbers==NULL || length<1) return false;
    int i = 0;
    while(i<length){
        if(numbers[i]!=i){
            int tmp = numbers[i];
            if(numbers[i]==numbers[tmp]) break;
            numbers[i] = numbers[tmp];
            numbers[tmp] = tmp;
            continue;
        }
        i++;
    }
    if(i==length) return false;
    *duplication = numbers[i];
    return true;
}
```

分析：时间复杂度 O(n)，空间复杂度 O(1)。因为元素大小在[0,n-1]之间，过程如下：

1）在遍历到下标 i 时，判断 numbers[i]是否等于 i，如果相等，进入 2）；如果不等，进入 3）；

2）i++，返回 1）；

3）判断 numbers[i]和 numbers[numbers[i]]是否相等，如果相等，则找到重复元素，立即返回；如果不相等，进入 4）；

4）交换 numbers[i]和 numbers[numbers[i]]，返回 1）；

- 构建乘积数组

给定一个数组 A[0,1,...,n-1], 请构建一个数组 B[0,1,...,n-1], 其中 B 中的元素 B[i]=A[0]*A[1]*...*A[i-1]*A[i+1]*...*A[n-1]。不能使用除法。

代码：

```
class Solution {
public:
    vector<int> multiply(const vector<int>& A) {
        vector<int> ret;
        int size = A.size();
        if(size==0) return ret;
        int *vec = new int[size];
        int *ved = new int[size];
        vec[0] = 1;
        for(int i=1;i<size;i++)
            vec[i] = vec[i-1]*A[i-1];
        ved[size-1] = 1;
        for(int i=size-2;i>=0;i--)
            ved[i] = ved[i+1]*A[i+1];
        for(int i=0;i<size;i++)
            ret.push_back(vec[i]*ved[i]);
        delete[] vec;
        delete[] ved;
        return ret;
    }
};
```

分析：B[i]=A[0]*A[1]*...*A[i-1]*A[i+1]*...*A[n-1]的意思是 B[i]等于 A 中除 A[i]之外的所有元素之积。如果可以使用除法，则先得到 A 中所有元素的积 all，然后 B[i]=all/A[i]。【尽管可以使用除法，但这种方法也是不严谨的，因为 A[i]可能等于 0】

时间复杂度为 O(n^2)的方法是对于 B[i]，遍历数组 A。想要 O(n)时间复杂度的算法，那肯定需要 O(n)的空间复杂度。如下过程：

B[i]=A[0]*A[1]*...*A[i-1]*A[i+1]*...*A[n-1]=C[i]*D[i]，其中

C[i]= A[0]*A[1]*...*A[i-1],

D[i]= A[i+1]*...*A[n-1],

而 C[i]和 D[i]的递推公式：

C[i]=C[i-1]*A[i-1]

D[i]=D[i+1]*A[i+1]

这样，就可以通过数组 A 从前往后得到 C，从后往前得到 D；然后通过 C 和 D 得到 A。

简化后的代码如下：

```
class Solution {
public:
    vector<int> multiply(const vector<int>& A) {
```

```
        vector<int> ret;
        int size = A.size();
        if(size==0) return ret;
        ret.push_back(1);
        for(int i=1;i<size;i++)
            ret.push_back(ret[i-1]*A[i-1]);
        int tmp = 1;
        for(int i=size-2;i>=0;i--){
            tmp *= A[i+1];
            ret[i] *= tmp;
        }
        return ret;
    }
};
```

- 正则表达式匹配

请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（包含 0 次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab*a"均不匹配

代码：

```
class Solution {
public:
    bool matchTmp(char *str, char *pattern){
        if(*str=='\0' && *pattern=='\0') return true;
        if(*str!='\0' && *pattern=='\0') return false;
        if(*(pattern+1)=='*'){
            if(*pattern==*str || (*pattern=='.'&&*str!='\0'))
                return matchTmp(str+1,pattern+2)
                || matchTmp(str+1,pattern)
                || matchTmp(str,pattern+2);
            else return matchTmp(str,pattern+2);
        }
        if(*str==*pattern || (*pattern=='.'&&*str!='\0'))
            return matchTmp(str+1,pattern+1);
        return false;
    }

    bool match(char* str, char* pattern){
        if(str==NULL || pattern==NULL) return false;
        return matchTmp(str,pattern);
    }
```

```
};
```

分析：按照《剑指 offer》上的分析：

（1）当模式中的第二个字符不是‘*’时，如果字符串的第一个字符和模式中的第一个字符匹配，那么在字符串和模式上都向后移动一个字符，然后匹配剩余字符串和模式。如果字符串中的第一个字符和模式中的第一个字符不匹配，则直接返回 false。

（2）当模式中的第二个字符是‘*’时，可能有多重不同的匹配方式。一种是在模式上向后移动两个字符，这相当于‘*’和它前面的字符都被忽略掉了，因为‘*’可以匹配字符串中的 0 个字符。如果模式中的第一个字符和字符串中的第一个字符相匹配时，则・在字符串向后移动一个字符，而在模式上有两个选择：可以在模式上向后移动两个字符，也可以保持模式不变。

特殊测试用例：

输入：“”，“”

输出：true

输入：“”，“.*”

输出：true

输入：“”，“c*”

输出：true

输入：”a”，“.*”

输出：true

输入："aaba","ab*a*c*a"

输出：false

- 表示数值的字符串

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。

代码：

```
class Solution {
public:
    bool isNumeric(char* string){
        if(string==NULL) return false;
        int len = strlen(string);
        if(len==0) return false;
        //第一个字符只可能是数字、加号、减号
        if(string[0]=='-' || string[0]=='+' || isNumber(string[0]));
        else return false;
        int dotNum = 0, i;
        for(i=1;i<len;i++){
```

```
            if(isE(string[i])) break;//碰到 E 或 e，立即退出，方便之后字符判断
            if(string[i]=='.'){//遇到小数点
                if(dotNum==0){//是第一个小数点
                    dotNum = 1;
                    continue;
                }else break;//不是第一个小数点，即有多个小数点，出错
            }
            if(isNumber(string[i])==false) break;//非数字，出错
        }
        if(i<len){
            if(isE(string[i])){//E 或 e 之后，只可能是+、-、数字
                if(++i>=len) return false;//E 或 e 之后没字符了，出错!
                //E 或 e 之后之只可能是+、-、数字
                if(string[i]=='-' || string[i]=='+' || isNumber(string[i]));
                else return false;
                for(i++;i<len;i++){//接下来全是数字才对
                    if(isNumber(string[i])==false) break;
                }
            }
            if(i<len) return false;
        }
        return true;//以上检验都没出问题，则确实是一个数，返回 true
    }

    bool isNumber(char ch){//判断是否为数字
        if(ch>='0' && ch<='9') return true;
        else return false;
    }

    bool isE(char ch){//判断是否为大写 E 或小写 e
        if(ch=='E' || ch=='e') return true;
        else return false;
    }

};
```

分析：时间复杂度为 O(n)，n 为字符串长度。什么好说的，注意一下几点即可：

（1）第一位只可能是+、-、数字

（2）E 或 e 之后只能是+、-、数字，且接着后面必须全是数字

（3）小数点只能有一个

（4）不能出现非数字字符，比如：a~z，A~Z，和一些特殊字符

- 字符流中第一个不重复的字符

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前

两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。

代码：

```
class Solution
{
private:
	int indexArr[256];
	int index;

public:
	Solution():index(0){
		for(int i=0;i<256;i++)
			indexArr[i] = -1;
	}

  //Insert one char from stringstream
	void Insert(char ch)
	{
		if(indexArr[ch]==-1)
			indexArr[ch] = index;
		else if(indexArr[ch]>=0)
			indexArr[ch] = -2;
		index++;
	}
  //return the first appearence once char in current stringstream
	char FirstAppearingOnce()
	{
		int minIndex = 256;
		char ch = '\0';
		for(int i=0;i<256;i++){
			if(indexArr[i]>=0&&minIndex>indexArr[i]){
				minIndex = indexArr[i];
				ch = i;
			}
		}
		if(ch=='\0') return '#';
		else return ch;
	}

};
```

分析：这个题目很好理解，但给的接口很难理解，那个 insert 是干啥的？无奈去看了《剑指 offer》的代码，才知道啥意思!

2016.06.24

- 链表中环的入口结点

一个链表中包含环，请找出该链表的环的入口结点。

代码：

```
/*
struct ListNode {
	int val;
	struct ListNode *next;
	ListNode(int x) :
		val(x), next(NULL) {
	}
};
*/
class Solution {
public:
	ListNode* EntryNodeOfLoop(ListNode* pHead){
		if(pHead==NULL) return NULL;
		ListNode *fast = pHead, *slow = pHead;//慢指针一次走一步，快指针一次走两步
		slow = slow->next,fast = fast->next;
		if(fast!=NULL) fast = fast->next;
		while(slow&&fast&&slow!=fast){
			slow = slow->next,fast = fast->next;
			if(fast!=NULL) fast = fast->next;
		}
		if(slow==NULL || fast==NULL) return NULL;//如果没有环，则返回 NULL
		slow = pHead;//让慢指针回到头结点
		while(slow!=fast){//快、慢指针一次走一步
			slow = slow->next;
			fast = fast->next;
		}
		return slow;
	}
};
```

分析：《王道程序员求职宝典》上第 201 页有详细的推导过程。

- 删除链表中重复的结点

在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。 例如，链表 1->2->3->3->4->4->5 处理后为 1->2->5

代码：

```
/*
```

```
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
        val(x), next(NULL) {
    }
};
*/
class Solution {
public:
    ListNode* deleteDuplication(ListNode* pHead){
        if(pHead==NULL) return NULL;
        ListNode *pre = pHead, *p = pre->next;
        while(p&&p->val==pre->val){//从头一直找相同元素
            p = p->next;
        }
        if(p==NULL){//说明上面 while 循环因为 p=NULL 而结束
            if(pre->next==NULL) return pre;//链表只有一个节点
            else{//链表所有元素都相同
                while(pre){//遍历链表，删除每一个元素
                    p = pre;
                    pre = pre->next;
                    delete p;
                }
                return NULL;//返回空
            }
        }else{//说明上面 while 循环因为 p->val != pre->val 而结束
            if(p==pre->next){//链表第一个元素和第二个元素不同，则递归下去
                pre->next = deleteDuplication(p);
            }else{//前几个元素相同
                ListNode *q = pre;
                while(q!=p){//删除前几个相同的元素
                    ListNode *t = q;
                    q = q->next;
                    delete t;
                }
                return deleteDuplication(p);//递归剩下的元素
            }
        }
        return pHead;
    }
};
```

分析：逻辑题，没什么算法可言。考虑几种情况：

（1）只有一个节点，则不会出现重复元素；

（2）所有元素都相同，则删除后最后返回 NULL；
（3）前面几个元素相同，则删除这几个元素后，递归处理剩下元素；
（4）第一个元素跟第二个元素不同，则递归处理以第二个元素为头的链表。
总之一点，要特别注意前几个元素相同的情况，因为涉及到头结点的处理。

- **对称的二叉树**

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

代码（非递归）：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};
*/
class Solution {
public:
    bool isSymmetrical(TreeNode* pRoot){
        if(pRoot==NULL) return true;
        deque<TreeNode *> queue;//队列
        queue.push_back(pRoot);
        vector<int> vec;//保存每一层的所有节点值，包括空节点（值为：0x0FFFFFFF）
        while(!queue.empty()){//层次遍历
            int size = queue.size(), i = 0, j = 0;
            while(i++<size){
                TreeNode *p = queue.front();
                queue.pop_front();

                if(p->left!=NULL){
                    queue.push_back(p->left);
                    vec.push_back(p->left->val);
                }else vec.push_back(0x0FFFFFFF);
                if(p->right!=NULL){
                    queue.push_back(p->right);
                    vec.push_back(p->right->val);
                }else vec.push_back(0x0FFFFFFF);
            }
            int si = vec.size();
```

```
                while(j<si-1-j && vec[j]==vec[si-1-j]) j++;//判断每层的节点值是否对称
                if(j<si-1-j) return false;
                vec.clear();
            }
            return true;
        }
};
```

分析：利用层次遍历的性质，遍历每一层时，从左到右保存所有节点值，包括空节点。然后判断这些节点是否对称。

代码 2（递归）：

```
class Solution {
public:
    bool isSymmetrical(TreeNode* pRoot){
        if(pRoot==NULL) return true;
        return isSymmetricalTmp(pRoot,pRoot);
    }

    bool isSymmetricalTmp(TreeNode *root1, TreeNode *root2){
        if(root1==NULL&&root2==NULL) return true;
        if(root1==NULL || root2==NULL) return false;
        if(root1->val!=root2->val) return false;
        else                                                    return
isSymmetricalTmp(root1->left,root2->right)&&isSymmetricalTmp(root1->right,root2->left
);
    }
};
```

分析：代码看起来简单多了。主要思想：要判断两棵树 tree1 和 tree2 是否对称，则首先要判断根节点是否一样，如果一样，那么再递归判断 tree1 的左子树和 tree2 的右子树，以及 tree1 的右子树和 tree2 的左子树。如果不一样，直接返回 false。

- **按之字形顺序打印二叉树**

请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。

代码：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
```

```
    }
};
*/
class Solution {
public:
    vector<vector<int> > Print(TreeNode* pRoot) {
        vector<vector<int> > ret;
        if(pRoot==NULL) return ret;
        deque<TreeNode *> queue;//队列
        queue.push_back(pRoot);
        vector<int> vec;//记录每一层的节点
        bool leftToRight = true;//为 true 表示从左到右输出，false 表示从右到左输出
        while(!queue.empty()){//层次遍历
            int size = queue.size(), i = 0;
            while(i++<size){//获得当前层的所有元素
                TreeNode *p = queue.front();
                queue.pop_front();
                vec.push_back(p->val);
                if(p->left) queue.push_back(p->left);
                if(p->right) queue.push_back(p->right);
            }
            vector<int> vecTmp;
            if(leftToRight){//从左到右
                for(i=0,size = vec.size();i<size;i++)
                    vecTmp.push_back(vec[i]);
                leftToRight = false;
            }else{//从右到左
                for(i=vec.size()-1;i>=0;i--)
                    vecTmp.push_back(vec[i]);
                leftToRight = true;
            }
            ret.push_back(vecTmp);
            vec.clear();
        }
        return ret;
    }};
```

分析：层次遍历，没什么好说的。

- **二叉树的下一个结点**

给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

代码：

```
using namespace std;
/*
struct TreeLinkNode {
    int val;
    struct TreeLinkNode *left;
    struct TreeLinkNode *right;
    struct TreeLinkNode *next;
    TreeLinkNode(int x) :val(x), left(NULL), right(NULL), next(NULL) {

    }
};
*/
class Solution {
public:
    TreeLinkNode* GetNext(TreeLinkNode* pNode){
        if(pNode==NULL) return NULL;
        if(pNode->right!=NULL){//右孩子不为空，则找到右孩子的最左孩子
            TreeLinkNode *p = pNode->right;
            while(p->left) p = p->left;
            return p;
        }else if(pNode->next==NULL){//右孩子为空，且父节点为空，说明它是根节点
            return NULL;
        }else{//右孩子为空，父节点不空
            if(pNode->next->left==pNode)//是父节点的左孩子，则直接输出父节点
                return pNode->next;
            else{//是父节点的右孩子，则一直向上找，直到当前节点不是父节点的右孩子
                TreeLinkNode *p = pNode->next;
                while(p->next&&p->next->right==p){//父节点不空，且是父节点的右孩子
                    p = p->next;
                }
                if(p->next==NULL) return NULL;//如果父节点为空，则 pNode 为中序最后一个节点，返回 NULL
                else return p->next;//父节点不空，且当前节点 p 是其父节点的左孩子
            }
        }
    }
};
```

分析：此题画一个三层的满二叉树，对每个节点都找一次下一个节点，即可知道怎么做了。主要分以下几个方面：

（1）如果右孩子不空，则右孩子的最左节点就是下一个；

（2）如果右孩子为空，且父节点也为空，表示此节点是根节点，且只有左子树，说明此节点是中序遍历的最后一个节点，直接返回 NULL；

（3）右孩子为空，且父节点不空，则：

3.1）如果此节点是父节点的左孩子，则其父节点就是下一个；

3.2）如果是父节点的右孩子，则向上一直找，直到找到一个节点 p，它是其父节点的左孩子。如果找不到，则表明 pNode 是中序遍历的最后一个节点了，返回 NULL；如果找到了，则返回 p 的父节点。

- **把二叉树打印成多行**

从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

代码：

```
/*
struct TreeNode {
	int val;
	struct TreeNode *left;
	struct TreeNode *right;
	TreeNode(int x) :
			val(x), left(NULL), right(NULL) {
	}
};
*/
class Solution {
public:
		vector<vector<int> > Print(TreeNode* pRoot) {
			vector<vector<int> > ret;
			if(pRoot==NULL) return ret;
			deque<TreeNode *> qu;//队列
			qu.push_back(pRoot);//根节点入队
			vector<int> vec;//存放每层节点
			TreeNode *p;
			int num = 1, count = 0, below = 0;//num 是每层的节点数, count 为循环变量, 取值范围是[0,num-1]，below 记录着下一层的节点数
			while(!qu.empty()){
				count++;

				p = qu.front();//获得队首元素
				qu.pop_front();//队首元素出队

				vec.push_back(p->val);//记录此层的元素
				if(p->left!=NULL){//左孩子不空，入队
					qu.push_back(p->left);
					below++;//记录下一层节点数
				}
				if(p->right!=NULL){//右孩子不空，入队
					qu.push_back(p->right);
					below++;//记录下一层节点数
```

```
                    }
                    if(count==num){//当此层遍历完时，修改 num、below、count 的值
                        num = below;
                        below = 0;
                        count = 0;
                        ret.push_back(vec);//将此层元素放入结果集
                        vec.clear();//清空，为下一次记录做准备
                    }
                }
                return ret;
            }

};
```

分析：此题是简单的层次遍历，但是返回值是一个二维数组，每层放在一个数组中，所有层组成一个二维数组。不像以前的层次遍历直接输出，此时需要控制一下让每层放在一个数组中。用了三个变量 num、count、below 来完成：num 是每层的节点个数，初始值为 1，表示第一个层只有一个节点（根节点）。count 为循环变量，取值范围是[0,num-1]。below 用于当遍历上一层节点时，用来记录下一层节点的个数。这样就可以知道每层有多少个节点。当 count=num 时，表示当前层已经遍历完毕，需要更新 num、count、below 的值。

- **序列化二叉树**

请实现两个函数，分别用来序列化和反序列化二叉树

代码：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};
*/
class Solution {
public:
    void preVisit(TreeNode *root, string &str){//先序遍历，得到序列化
        char num[10];
        sprintf(num,"%d",root->val);
        str += num;
        str += ",";
        if(root->left){
            preVisit(root->left,str);
```

```
        }else str += "$,";
        if(root->right){
            preVisit(root->right,str);
        }else str += "$,";
    }

    char* Serialize(TreeNode *root) {//序列化
        if(root==NULL) return NULL;
        string str = "";
        preVisit(root,str);//获得序列化
        int size = str.size();
        char *retStr = new char[size];//申请动态内存，防止函数返回后空间释放，引起空指针异常
        strcpy(retStr,str.c_str());
        retStr[size-1] = '\0';//str 的最后一个字符是逗号，此时令其为'\0'
        return retStr;
    }

    TreeNode *preVisitCreateTree(char *str, int &len, int &start){//先序构造二叉树
        if(start>=len) return NULL;
        if(str[start]=='$'){
            start += 2;//str[start]='$',str[start+1]=',', 所以 start 要加 2
            return NULL;
        }
        //取出节点值
        int j = start;
        char num[10];
        while(j<len && str[j]>='0' && str[j]<='9'){
            num[j-start] = str[j];
            j++;
        }
        start = j + 1;//令 start 指向下一个节点值的首地址
        TreeNode *node = new TreeNode(atoi(num));//atoi 转化为 int 型
        node->left = preVisitCreateTree(str,len,start);//递归建立左子树
        node->right = preVisitCreateTree(str,len,start);//递归建立右子树
        return node;
    }

    TreeNode* Deserialize(char *str) {
        if(str==NULL) return NULL;
        int len = strlen(str);
        int start = 0;
        return preVisitCreateTree(str,len,start);
    }
```

```
};
```

分析：全程先序遍历即可。

- **二叉搜索树的第 k 个结点**

给定一颗二叉搜索树，请找出其中的第 k 大的结点。例如， 5 / \ 3 7 /\ /\ 2 4 6 8 中，按结点数值大小顺序第三个结点的值为 4。

代码 1：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};
*/
class Solution {
public:
    TreeNode* KthNode(TreeNode* pRoot, unsigned int k){
        if(pRoot==NULL || k==0) return NULL;
        vector<int> vec;
        inOrderVisit(pRoot,vec);
        if(k>vec.size()) return NULL;
        return preOrderVisit(pRoot,vec.at(k-1));
    }

    void inOrderVisit(TreeNode *root, vector<int> &vec){//中序遍历得到有序序列
        if(root->left) inOrderVisit(root->left,vec);
        vec.push_back(root->val);
        if(root->right) inOrderVisit(root->right,vec);
    }

    TreeNode *preOrderVisit(TreeNode *root, int val){//先序遍历找出第 k 大节点
        if(root==NULL) return NULL;
        if(root->val==val) return root;//找到
        TreeNode *left = preOrderVisit(root->left,val);//继续在左子树中寻找
        if(left) return left;//一旦找到，立马返回，避免不必要的递归
        TreeNode *right = preOrderVisit(root->right,val);//同上
        if(right) return right;
        return NULL;//只有找不到时，此句才会执行
    }
```

```
};
```

分析：不加思考的方案。先中序遍历搜索二叉树，得到一个有序序列，继而得到第 k 大的数 val，然后再次遍历（先序、中序、后序均可）二叉树，找到值等于 val 的节点。时间复杂度为 O(n)，n 为节点个数，但这种方法遍历了二叉树两次，且消耗了 O(n)的空间。

代码 2：

```
class Solution {
public:
    TreeNode* KthNode(TreeNode* pRoot, unsigned int k){
        if(pRoot==NULL || k==0) return NULL;
        int order = 1;//序号，标识当前访问的节点是第几大
        TreeNode *ret = inOrderVisit(pRoot,k,order);
        if(ret) return ret;
        return NULL;//k 的值大于节点个数
    }

    TreeNode* inOrderVisit(TreeNode *root, int k, int &order){
        if(root==NULL) return NULL;
        TreeNode * left = inOrderVisit(root->left,k,order);
        if(left) return left;//如果左孩子的返回值不为空，则已找到，立即返回
        if(order++==k) return root;//判断是否为第 k 大节点
        TreeNode * right = inOrderVisit(root->right,k,order);
        if(right) return right;//同 left
        return NULL;//只有 k 超出节点个数时，此句才会执行到
    }
};
```

分析：此方法在中序遍历的过程中找出第 k 大节点，时间复杂度 O(n)，且只遍历了一次。空间复杂度为 O(1)。相对于上一种方法，效率高了不少。

- 数据流中的中位数

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

代码：

```
class Solution {
    vector<int> data;
public:
    void Insert(int num)
    {
        data.push_back(num);
    }
```

```
    double GetMedian()
    {
        int size = data.size();
        sort(data.begin(),data.end());
        if(size&0x1) return data[size/2];
        else return (data[size/2-1]+data[size/2])/2.0;
    }
};
```

分析：直接的想法，读入每个数后，从小到大排序，取中值即可。时间复杂度为 O(nlogn)。

换一种时间复杂度为 O(n)的方法。快排的思想：每一次划分都能确定一个元素的最终位置 pos，如果 pos 正好是中点（size/2），则找到了中位数。如果 pos<size/2，则排除左边元素，在右边元素中寻找。如果 pos>size/2，则排除右边元素，在左边元素中寻找。

当然，size 为奇数和偶数时，还有点不一样。如果 size 为奇数，则只需一次寻找即可，即找到 size/2 的位置；如果 size 为偶数，则需要两次寻找，分别找到 size/2 和 size/2-1 两个位置，然后求均值。

代码就不写了。

2016.06.26

- 滑动窗口的最大值

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小 3，那么一共存在 6 个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下 6 个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

代码：

```
class Solution {
public:
    vector<int> maxInWindows(const vector<int>& num, unsigned int size){
        int n = num.size();
        vector<int> ret;
        if(n==0 || size==0 || size>n) return ret;
        deque<int> queue;
        int i = 0;
        for(;i<size;i++){//初始化队列
            while(!queue.empty()&&num[i]>num[queue.back()]) queue.pop_back();
            queue.push_back(i);
        }
        ret.push_back(num[queue.front()]);
        for(;i<n;i++){
            while(!queue.empty()&&num[i]>num[queue.back()]) queue.pop_back();
            if(!queue.empty()&&queue.front()+size<=i) queue.pop_front();
```

```
                queue.push_back(i);
                ret.push_back(num[queue.front()]);
            }
            return ret;
        }
};
```

分析：一般的想法是对每 size 个元素都遍历一遍，选出最大值。这样的复杂度为 O(kn)，如果 k 比较大的话，那么近似 O(n^2)了，显然效率不够高。

本代码采用了《剑指 offer》上的思想，详细讲解过程见 P291.

- 矩阵中的路径

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如

a b c e

s f c s

a d e e

矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符 b 占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

代码（非递归）：

```
class Solution {
public:
    bool hasPath(char* matrix, int rows, int cols, char* str){
        //输入正确性检查
        if(matrix==NULL || str==NULL || rows<=0 || cols<=0) return false;
        int size = strlen(matrix);
        int len = strlen(str);
        if(size!=rows*cols || len>size) return false;

        int *mark = new int[size];//标记矩阵，mark[i]=0 表示没走过，可以走
        for(int i=0;i<size;i++)//初始化为 0
            mark[i] = 0;
        int *stack = new int[len];//行走方向记录，是个栈，用于回溯。0：从下方走来；1：从左方走来；2：从上方走来；3：从右方走来
        int top = 0;//栈顶指针
        for(int i=0;i<rows;i++){//开始遍历字符矩阵，以每一个字符为第一步
            for(int j=0;j<cols;j++){
                top = 0;
                if(matrix[i*cols+j]==str[0])//此字符是路径的第一步
                    stack[top++] = -1;//第一步的方向标记为-1，表示无方向
                else continue;//此字符不是路径的第一步，则继续遍历字符矩阵下一字符
```

```
                int rowIndex = i, colIndex = j;//新行、列变量，为了不改变 i 和 j
                while(top&&top<len){//top!=0 表示不存在以 matrix[i][j] 为首的路径。top=len 表示已找到路径
                    if(mark[rowIndex*cols+colIndex]>3){//如果四个方向都走过了
                        mark[rowIndex*cols+colIndex] = 0;//消除记录，准备回溯
                        switch(stack[top-1]){//根据栈的记录
                            case 0: rowIndex++; break;//回溯到下方
                            case 1: colIndex--; break; //回溯到左方
                            case 2: rowIndex--; break; //回溯到上方
                            case 3: colIndex++; break; //回溯到右方
                            case -1: break;//这种情况表示已经回溯到路径的第一个字符了，就是说寻找路径失败
                        }
                        top--;//栈也回退一步
                        continue;//继续寻找
                    }
                    switch(mark[rowIndex*cols+colIndex]){//遍历当前字符的四个方向
                        //0：上方
                        case 0: mark[rowIndex*cols+colIndex]++;//下次向右走
                            if(rowIndex>0 && mark[(rowIndex-1)*cols+colIndex]==0 &&matrix[(rowIndex-1)*cols+colIndex]==str[top]){
                                stack[top++] = 0;//记录方向
                                rowIndex--;//向上走
                            }
                            break;
                        //1：右方
                        case 1: mark[rowIndex*cols+colIndex]++;//下次向下走
                            if(colIndex<cols-1 && mark[rowIndex*cols+colIndex+1]==0 && matrix[rowIndex*cols+colIndex+1]==str[top]){
                                stack[top++] = 1;
                                colIndex++;
                            }
                            break;
                        //2：下方
                        case 2: mark[rowIndex*cols+colIndex]++; //下次向左走
                            if(rowIndex<rows-1 && mark[(rowIndex+1)*cols+colIndex]==0 && matrix[(rowIndex+1)*cols+colIndex]==str[top]){
                                stack[top++] = 2;
                                rowIndex++;
                            }
                            break;
                        //3：左方
                        case 3: mark[rowIndex*cols+colIndex]++;//没有下次了
                            if(colIndex>0 &&
```

```
mark[rowIndex*cols+colIndex-1]==0 && matrix[rowIndex*cols+colIndex-1]==str[top]){
                                        stack[top++] = 3;
                                        colIndex--;
                                    }
                                    break;
                                }
                            }
                            if(top==len){//找到路径，释放空间，返回 true
                                delete[] mark;
                                delete[] stack;
                                return true;
                            }
                        }
                    }
                    delete[] mark;
                    delete[] stack;
                    return false;//没找到路径
                }
};
```

分析：类似于迷宫问题，典型的回溯法，时间复杂度为 O(kmn)，k 为路径 str 长度，m 为矩阵行数，n 为矩阵列数。没什么好的方法，但可以优化一下。这里将匹配子串 str 称为路径。本代码主要思想如下：

（1）分别以矩阵 matrix 的每一个元素 matrix[i][j]为路径 str 的第一个步 str[0]，如果 matrix[i][j]!=str[0]，则继续遍历 matrix 下一元素；如果相等，转向（2）；

（2）依次遍历 matrix[i][j]的四个方向，上：matrix[i-1][j]，右：matrix[i][j+1]；下：matrix[i+1][j]；左：matrix[i][j-1]。看看跟路径 str 的这一步 str[k]是否相等，相等的话，就往这个方向走一步；不相等的话，回溯到上一步。

前进需要一个标记矩阵 mark，它记录着行走的每一步，它有四个值：

0：表示还没走过，同时也表示应该向上走；

1：表示向右走；

2：表示向下走；

3：表示向左走；

代码中用了一个 switch 来根据 mark[i]的值判断该向哪里走。

回溯需要一个栈 stack，用于记录一路上的方向，即上一步到这一步是从哪个方向过来的，也有四个值：

0：表示从下方走来；

1：表示左方走来；

2：表示从上方走来；

3：表示从右方走来；

这四个值跟 mark 的四个值是对应的，也即栈 stack 和标记矩阵 mark 是对应的。

我们选择一个方向来说一下：

```
case 1: //右方
    mark[rowIndex*cols+colIndex]++;
    if(colIndex<cols-1 && mark[rowIndex*cols+colIndex+1]==0 &&
matrix[rowIndex*cols+colIndex+1]==str[top]){
        stack[top++] = 1;
        colIndex++;
    }
    break;
```

- mark[rowIndex*cols+colIndex]++;表示当前方向（右方）已经判断过了，让 mark 自加，表示下次得从下方开始判断了。
- 这个 if 语句中，
  - colIndex<cols-1 是防止越界
  - mark[rowIndex*cols+colIndex+1]==0 是检查这个位置是否已经走过了，等于 0 表示没走过，其他值表示已走过。这是题目中要求的，不能重复走。
  - matrix[rowIndex*cols+colIndex+1]==str[top]是判断下一个位置（右方）是否为路径的下一步，是的话，才有走的必要。
  - 如果 if 条件满足，则 stack[top++] = 1 记录是从左方走来的，用于之后的回溯。colIndex++表示向右方走一步
- 最后别忘了 break

注意到没有，不管走没走，一个方向一旦判断过，那就真的过去了。即不管这个 if 是否成立，mark[rowIndex*cols+colIndex]都是要递增的。所以 mark[rowIndex*cols+colIndex]必须要放在 if 的前面，否则当 if 成立时，colIndex 自加了一次，rowIndex*cols+colIndex 已不再是之前的了。

好了，就这样吧！有了思想（回溯），还得知道用什么数据结构、怎么用此结构去实现才行。最后注意调试，因为这样的代码很容易出错。

- 机器人的运动范围

地上有一个 m 行和 n 列的方格。一个机器人从坐标 0,0 的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于 k 的格子。 例如，当 k 为 18 时，机器人能够进入方格（35,37），因为 3+5+3+7 = 18。但是，它不能进入方格（35,38），因为 3+5+3+8 = 19。请问该机器人能够达到多少个格子？

代码：

```
class Solution {
public:
    int movingCount(int threshold, int rows, int cols){
        if(threshold<0 || rows<1 || cols<1) return 0;
        int size = rows*cols;
        int *mark = new int[size];//标记矩阵，用一维数组代替
        int errorNum = 0;//行坐标和列坐标位数之和大于 threshold 的单元格个数
        for(int i=0;i<size;i++)//初始化标记矩阵
            if(getSum(i/cols,i%cols)>threshold){//不能走的单元格用-1 标记
                mark[i] = -1;
```

```
                errorNum++;
            }else mark[i] = 0;//能走的用 0 标记
        int *stack = new int[size];//方向记录，栈的操作
        int top = 0;//栈顶指针
        stack[top++] = 0;//[0,0]进栈，值可以为任意数，因为[0,0]是起点
        int rowIndex = 0, colIndex = 0;//行、列变量
        set<int> trace;//记录走过的单元格，集合的性质是元素唯一
        trace.insert(0);//记录[0,0]单元格
        while(top){//当栈为空时，表示已回溯到起点[0,0]，结束
            if(mark[rowIndex*cols+colIndex]>3){//此单元格四个方向均已走过，准备回溯
                mark[rowIndex*cols+colIndex] = 0;//清楚标记
                switch(stack[top-1]){//看看从哪来
                    case 0: rowIndex++; break;//从下方来
                    case 1: colIndex--; break;//从左方来
                    case 2: rowIndex--; break;//从上方来
                    case 3: colIndex++; break;//从右方来
                }
                top--;//出栈
                if(trace.size()==size-errorNum) break;//如果此时已经走过了所有可以走
的单元格，立即退出，因为已经达到最大值
                continue;
            }

            switch(mark[rowIndex*cols+colIndex]){//看看往哪走
                //往上走
                case 0: mark[rowIndex*cols+colIndex]++;
                        if(rowIndex>0 && mark[(rowIndex-1)*cols+colIndex]==0){
                            trace.insert((rowIndex-1)*cols+colIndex);
                            stack[top++] = 0;
                            rowIndex--;
                        }
                        break;
                //往右走
                case 1: mark[rowIndex*cols+colIndex]++;
                        if(colIndex<cols-1 && mark[rowIndex*cols+colIndex+1]==0){
                            trace.insert(rowIndex*cols+colIndex+1);
                            stack[top++] = 1;
                            colIndex++;
                        }
                        break;
                //往下走
                case 2: mark[rowIndex*cols+colIndex]++;
                        if(rowIndex<rows-1                                    &&
mark[(rowIndex+1)*cols+colIndex]==0){
```

```
                                trace.insert((rowIndex+1)*cols+colIndex);
                                stack[top++] = 2;
                                rowIndex++;
                            }
                            break;
                    //往左走
                    case 3: mark[rowIndex*cols+colIndex]++;
                            if(colIndex>0 && mark[rowIndex*cols+colIndex-1]==0){
                                trace.insert(rowIndex*cols+colIndex-1);
                                stack[top++] = 3;
                                colIndex--;
                            }
                            break;
                }
            }
            delete[] mark;
            delete[] stack;
            return trace.size();
        }

        int getSum(int row, int col){//获取行、列位数之和
            int sum = 0;
            while(row){
                sum += row%10;
                row /= 10;
            }
            while(col){
                sum += col%10;
                col /= 10;
            }
            return sum;
        }
};
```

分析：本题与上一题虽然都是回溯法，但最大的区别在于，**此题的起点只能是[0,0]**，所以这就将时间复杂度从 O(n^2*m^2)降到了 O(mn)，即整个过程只需一次回溯即可。还有一个区别是，需要记录走过的单元格个数，即使回溯了，但走过的单元格都是要计数的。实现中，用了一个集合 trace 来记录走过的单元格编号，最后返回的也是 trace 的大小。

————微信扫描学习————