- 1.在一个二维数组中（每个一维数组的长度相同），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。
    - 方法一几乎美做任何优化，复杂度较高，要写出来需要细心；

```
class Solution {
public:
    bool Find(int target, vector<vector<int> > array) {
        int len = array.size();
        if(len == 0){return false;}
        for(int i=0; i<len; i++){
            if(findInArray(target, array[i])){
                return true;
            }
        }
        return false;
    }

    bool findInArray(int target, vector<int> array) {
        int len = array.size();
        if(len == 0){return false;}
        if(target < array[0] || target > array[len - 1]){
            return false;
        }
        int low = 0, high = len - 1;
        while(high - low > 1){
            int mid = (high + low) / 2;
            if(target == array[mid]){
                return true;
            }else if(target > array[mid]){
                low = mid;
            }else{
                high = mid;
            }
        }
        if(target == array[low] || target == array[high]){
            return true;
        }
        return false;
    }
};
```

```
//方法2， 利用矩阵已然排好序的特性，从左下到右上的探索
class Solution {
public:
    bool Find(int target, vector<vector<int> > array) {
        int row = array.size();
        int col = array[0].size();
        int i,j;
```

```
        for(i=row-1, j=0; i>=0 && j<col;){
            if(target == array[i][j]){
                return true;
            }else if(target > array[i][j]){
                j++;
            }else{
                i--;
            }
        }
        return false;
    }
};
```

- 2.请实现一个函数，将一个字符串中的每个空格替换成“%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。

```
class Solution {
public:
    void replaceSpace(char *str,int length) {
    string temp = "";
    for(int i=0;i<length; i++){
        if(str[i] == ' '){
            temp += "%20";
        }else{
            temp += str[i];
        }
    }
    strcpy(str, &temp[0]);
    }
};
```

- 3.输入一个链表，按链表从尾到头的顺序返回一个ArrayList。

```
/**
*  struct ListNode {
*      int val;
*      struct ListNode *next;
*      ListNode(int x) :
*          val(x), next(NULL) {
*      }
*  };
*/
class Solution {
public:
    vector<int> printListFromTailToHead(ListNode* head) {
        vector<int> temp;
        while(head){
            temp.insert(temp.begin(), head->val);
            head = head->next;
```

```
        }
        return temp;
    }
};
```

- 4.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。
    - 递归实现，前序遍历的开头是根结点，找到其在中序遍历中的位置就可以分开左右子树了
    - 注意，第一个for循环就找到根节点了，但是c++中的vector没法直接取其中的一段，所以还得慢慢赋值。

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

        int rootVal = pre[0], len = vin.size(), gen;
        for(gen=0; gen<len; gen++){
            if(vin[gen] == rootVal){
                break;
            }
        }

        vector<int> vin_left, pre_left;
        for(int i = 0; i < gen; i++){
            vin_left.push_back(vin[i]);
            pre_left.push_back(pre[i+1]);//先序第一个为根节点
        }

        // 右子树
        vector<int> vin_right, pre_right;
        for(int i = gen + 1; i < len; i++){
            vin_right.push_back(vin[i]);
            pre_right.push_back(pre[i]);
        }

        TreeNode* left = reConstructBinaryTree(pre_left, vin_left);
        TreeNode* right = reConstructBinaryTree(pre_right, vin_right);
        TreeNode *root = new TreeNode(rootVal);
        root->left = left;
```

```
        root->right = right;
        return root;
    }
};
```

- 5.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。

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

    int pop() {
        while(stack1.size() > 1){
            int node = stack1.top();
            stack1.pop();
            stack2.push(node);
        }
        int temp = stack1.top();
        stack1.pop();
        while(stack2.size() > 0){
            int node = stack2.top();
            stack2.pop();
            stack1.push(node);
        }
        return temp;
    }

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

- 6.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。输入一个非递减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。

```
class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        int len = rotateArray.size();
        if(len==0){return 0;}
        for(int i=len-1;i>=0;i--){
            if(rotateArray[i] < rotateArray[i-1]){
                return rotateArray[i];
            }
        }
```

```
    }
};
```

- 7.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（从0开始，第0项为0）。(n<=39)

```
class Solution {
public:
    int Fibonacci(int n) {
        if(n==0 || n==1){return n;}
        vector<int> temp;
        temp.push_back(0);
        temp.push_back(1);
        for(int i = 2; i <= n; i++){
            temp.push_back(temp[i - 1]+temp[i - 2]);
        }
        return temp[n];
    }
};
```

- 8.一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（先后次序不同算不同的结果）。

```
class Solution {
public:
    int jumpFloor(int number) {
        vector<int> out;
        out.push_back(1);
        out.push_back(2);
        if(number <= 2){
            return out[number - 1];
        }
        for(int i=2; i<number; i++){
            out.push_back(out[i-1]+out[i-2]);
        }
        return out[number - 1];
    }
};
```

- 9.一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。
    - 总结起来就是2的幂

```
class Solution {
public:
    int jumpFloorII(int number) {
        return 1 << (number - 1);
```

```
    }
};
```

- 10.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？
    - 比如n=3时，2*3的矩形块有3种覆盖方法

```
class Solution {
public:
    int rectCover(int number) {
        if(number <= 2){return number;}
        vector<int> out;
        out.push_back(1);
        out.push_back(2);
        for(int i=2;i<number;i++){
            out.push_back(out[i-1]+out[i-2]);
        }
        return out[number-1];
    }
};
```

- 11.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。
    - 独特思路：整数n，进行n&(n-1)运算，会把二进制表示中最右边的1变为0。

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

- 12.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。(保证base和exponent不同时为0)
    - 正数为logN，负数为N

```
class Solution {
public:
    double Power(double base, int exponent) {
        if(base == 0){return 0;}
        if(exponent == 0){return 1;}
        double sum = 1;
```

```
        if(exponent > 0){
            while(exponent != 0){
                int temp = exponent % 2;
                sum *= sum;
                exponent /= 2;
                if(temp == 1){
                    sum *= base;
                }
            }
        }
        else{
            while(exponent != 0){
                sum *= 1/base;
                exponent++;
            }
        }
        return sum;
    }
};
```

- 13.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。

```
class Solution {
public:
    void reOrderArray(vector<int> &array) {
        vector<int> even;
        int count = 0;
        for(int i=0;i<array.size();i++){
            if(array[i]%2==0){
                even.push_back(array[i]);
            }else{
                array[count] = array[i];
                count++;
            }
        }

        for(int i=count;i<array.size();i++){
            array[i] = even[i-count];
        }
    }
};
```

- 14.输入一个链表，输出该链表中倒数第k个结点。
    - 快慢指针，很有趣的方法

```
/*
struct ListNode {
    int val;
```

```
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
  ListNode* FindKthToTail(ListNode* pListHead, unsigned int k) {
    ListNode *fast = pListHead;
    ListNode *slow = pListHead;
    int count = 0;
    while(count < k && fast){
      fast = fast->next;
      count++;
    }
    if(count != k)
    {
      return NULL;
    }
    while(fast){
      fast = fast->next;
      slow = slow->next;
    }
    return slow;
  }
};
```

- 15.输入一个链表，反转链表后，输出新链表的表头。
  - 题目不难，多看看，注意开头的判断很重要。

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

```
        }
        return tail;
    }
};
```

- 16.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。
    - 这题都写烂了（汗

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
    ListNode* Merge(ListNode* pHead1, ListNode* pHead2)
    {
        ListNode *head = new ListNode(0);
        ListNode *tail = head;
        while(pHead1 && pHead2){
            ListNode *temp = new ListNode(0);
            temp->val = pHead1->val <= pHead2->val ? pHead1->val: pHead2->val;
            if(pHead1->val <= pHead2->val){
                pHead1 = pHead1->next;
            }else{
                pHead2 = pHead2->next;
            }
            tail->next = temp;
            tail = temp;
        }
        while(pHead1)
        {
            ListNode *temp = new ListNode(0);
            temp->val = pHead1->val;
            pHead1 = pHead1->next;
            tail->next = temp;
            tail = temp;
        }
        while(pHead2)
        {
            ListNode *temp = new ListNode(0);
            temp->val = pHead2->val;
            pHead2 = pHead2->next;
            tail->next = temp;
            tail = temp;
        }
        return head->next;
```

```
    }
};
```

- 19.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4 X 4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.
    - 利用4个变量记录轨迹，注意while、for循环的边界问题以及为了不重复而存在的if语句。

```
class Solution {
public:
    vector<int> printMatrix(vector<vector<int> > matrix) {
        int low = 0, high = matrix.size()-1;
        int left = 0, right = matrix[0].size()-1;
        vector<int> result;

        while(left<=right && low<=high){
            for(int i=left; i<=right; i++){
                result.push_back(matrix[low][i]);
            }

            for(int i=low+1; i<=high; i++){
                result.push_back(matrix[i][right]);
            }

            if(low < high){
                for(int i=right-1; i>=left; i--){
                    result.push_back(matrix[high][i]);
                }
            }

            if(left < right){
                for(int i=high-1; i>low; i--){
                    result.push_back(matrix[i][left]);
                }
            }
            low++;
            high--;
            left++;
            right--;
        }
        return result;
    }
};
```

- 20.定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。
    - 注意：保证测试中不会当栈为空的时候，对栈调用pop()或者min()或者top()方法。

```
class Solution {
public:
```

```
    vector<int> result;
    stack<int> temp;
    void push(int value) {
        result.push_back(value);
        if(result.size() == 1){
            temp.push(value);
        }else{
            temp.push(temp.top() > value ? value: temp.top());
        }
    }
    void pop() {
        temp.pop();
        result.erase(result.begin());
    }
    int top() {
        return result.back();
    }
    int min() {
        return temp.top();
    }
};
```

- 21.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可能为该栈的弹出顺序。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。（注意：这两个序列的长度是相等的）
    - 这道题的关键在于只要在第一个出栈的元素之前压进去的元素的出栈的相对顺序不变就可以了。

```
class Solution {
public:
    bool IsPopOrder(vector<int> pushV,vector<int> popV) {
        int first = popV[0];
        int count = popV.size()-1;
        for(int i=0; i<pushV.size(); i++){
            if(pushV[i]==first){
                return true;
            }
            for(; count>=0; count--){
                if(popV[count] == pushV[i]){
                    break;
                }
            }
            if(count < 1){
                return false;
            }
        }
        return true;
    }
};
```

- 28.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。

```
class Solution {
public:
  int MoreThanHalfNum_Solution(vector<int> numbers) {
    map<int, int> out;
    int maxCount=1, num=numbers[0];
    for(int i=0;i<numbers.size();i++){
      if(out.find(numbers[i]) == out.end()){
        out[numbers[i]] = 1;
      }
      else{
        out[numbers[i]]++;
      }
      if(out[numbers[i]] > maxCount){
        maxCount = out[numbers[i]];
        num = numbers[i];
      }
    }
    if(maxCount > numbers.size()/2){
      return num;
    }else{
      return 0;
    }
  }
};
```

- 29.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,。
  - 这题花了一点时间，原因在于，vector的语法（取部分元素），以及如何开始循环，元素合适需要被添加。

```
class Solution {
public:
  vector<int> GetLeastNumbers_Solution(vector<int> input, int k) {
    int len = input.size();
    vector<int> out;
    if(len == 0 || k > len){
      return out;
    }else{
      out.push_back(input[0]);
    }

    for(int i = 1; i< len; i++){
      int temp = out.size() >= k ? k : out.size();
      int j;
      for(j = 0; j < temp; j++){
        if(input[i] < out[j]){
```

```
                out.insert(out.begin()+j, input[i]);
                break;
            }
            if(j == temp-1 && temp < k){
                out.push_back(input[i]);
            }
        }
    }
    out.resize(k);
    return out;
  }
};
```

- 30.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)。给一个数组，返回它的最大连续子序列的和。(子向量的长度至少是1)

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

- 31.求出任意非负整数区间中1出现的次数（从1到n中1出现的次数）。
  - 可以尝试总结出个十百千万位出现1的次数然后简单求解。

```
class Solution {
public:
  int NumberOf1Between1AndN_Solution(int n)
  {
    int count = 0;
    for(int i=1;i<=n;i++){
```

```
            int temp = i;
            while(temp!=0){
                if(temp %10 == 1){
                    count++;
                }
                temp/=10;
            }
        }
        return count;
    }
};
```

- 32.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。

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

    static bool cmp(int a, int b)
    {
        string temp1 = to_string(a) + to_string(b);
        string temp2 = to_string(b) + to_string(a);
        return temp1 < temp2;
    }
};
```

- 33.把只包含质因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含质因子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。
    - 这题有意思，丑数的是个难点，该方法用的是三个指针指向不同的位置选择的，需牢记。

```
class Solution {
public:
    int GetUglyNumber_Solution(int index) {
        vector<int> out;
        out.push_back(1);
        int p2=0, p3=0,p5=0;
        while(out.size() < index){
            int temp = out[p3]*3 <= out[p5]*5 ? out[p3] * 3: out[p5] * 5;
            temp = out[p2]*2 < temp ? out[p2] * 2: temp;
```

```
            out.push_back(temp);
            if(temp == out[p2]*2){p2++;}
            if(temp == out[p3]*3){p3++;}
            if(temp == out[p5]*5){p5++;}
        }
        return out[index - 1];
    }
};
```

- 34.在一个字符串(0<=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,并返回它的位置, 如果没有则返回 -1（需要区分大小写）。
    - 用了map，某种程度上降低了难度；可以考虑用更简单的方法。

```
class Solution {
public:
    int FirstNotRepeatingChar(string str) {
        map<char, int> result;
        for(int i=0; i< str.size(); i++){
            if(result.find(str[i])==result.end()){
                result[str[i]]=1;
            }else{
                result[str[i]]++;
            }
        }
        for(int i=0; i< str.size(); i++){
            if(result[str[i]]==1){
                return i;
            }
        }
        return -1;
    }
};
```

- 36.输入两个链表，找出它们的第一个公共结点。
    - 这题有意思，因为链表可能不等长，但是各自拼接起来就等长了。
    - 如果不存在公共节点，跑完等长带后直接输出NULL.

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
    ListNode* FindFirstCommonNode( ListNode* pHead1, ListNode* pHead2) {
        ListNode *p1 = pHead1;
```

```
        ListNode *p2 = pHead2;
        if(!p1 || !p2){return NULL;}
        //if(p1==p2){return p1;}
        while(p1 != p2)
        {
            p1 = p1->next;
            p2 = p2->next;
            if(!p1 && !p2){return NULL;}
            if(!p1){p1 = pHead2;}
            if(!p2){p2 = pHead1;}
        }
        return p1;
    }
};
```

- 37.统计一个数字在排序数组中出现的次数。

```
class Solution {
public:
    int GetNumberOfK(vector<int> data ,int k) {
        bool exist = false;
        int count = 0;
        for(int i=0; i < data.size(); i++){
            if(data[i] == k){
                count++;
                exist = true;
            }else{
                if(exist){
                    return count;
                }
            }
        }
        return count;
    }
};
```

- 40.一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。
    - 套路还是map的套路，不过有个方法记一下；
    - 首先：位运算中异或的性质：两个相同数字异或=0，一个数和0异或还是它本身。
    - 当只有一个数出现一次时，我们把数组中所有的数，依次异或运算，最后剩下的就是落单的数，因为成对儿出现的都抵消了。

```
class Solution {
public:
    void FindNumsAppearOnce(vector<int> data,int* num1,int *num2) {
        map<int, int> result;
        for(int i=0; i<data.size(); i++){
            if(result.find(data[i]) == result.end()){
```

```
            result[data[i]] = 1;
        }else{
            result[data[i]]++;
        }
    }

    int count = 0;
    map<int, int>::iterator iter = result.begin();
    while(iter != result.end()){
        if(iter->second == 1){
            if(count == 0){
                *num1 = iter->first;
                count++;
            }else{
                *num2 = iter->first;
                return;
            }
        }
        iter++;
    }
  }
};
```

- 41.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!

```
class Solution {
public:
  vector<vector<int> > FindContinuousSequence(int sum) {
    vector<vector<int> > result;
    int low = 1, high = 2, current;
    while(high > low){
      current = (low+high)*(high-low+1)/2;
      if(sum == current){
        vector<int> temp;
        for(int i=low; i<=high; i++){
          temp.push_back(i);
        }
        result.push_back(temp);
        high++;
      }
      else if(sum > current){
        high++;
      }else{
        low++;
      }
    }
    return result;
```

```
    }
};
```

- 42.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。

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

- 43.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！

```
class Solution {
public:
    string LeftRotateString(string str, int n) {
        int len = str.size();
        if(len == 0){return str;}
        n %= len;
        return str.substr(n) + str.substr(0, n);
    }
};
```

- 44.翻转这些单词顺序("student. a am I"——>"I am a student.")

```
class Solution {
public:
    string ReverseSentence(string str) {
        int start = 0;
```

```
        string result="";
        for(int i=0; i< str.size(); i++){
            if(str[i] == ' '){
                result = ' ' + str.substr(start, i-start) + result;
                start = i+1;
            }
        }
        result = str.substr(start) + result;
        return result;
    }
};
```

- 45.请从输入的一个数组中判断能否构成5个一连的顺子，输入的数范围为0-13的整数，其中0可以表示1~13的任意整数。若能构成顺子，返回true，否则返回false。
    - 除0外，极差小于５；不能有重复。

```
class Solution {
public:
    bool IsContinuous( vector<int> numbers ) {
        if(numbers.size() < 5){return false;}

        map<int, int> result;
        int max = 1, min = 13;
        for(int i=0; i<numbers.size(); i++){
            if(result.find(numbers[i]) == result.end()){
                result[numbers[i]] = 1;
            }else{
                if(numbers[i] != 0){
                    return false;
                }else{
                    result[numbers[i]]++;
                }
            }
            if(numbers[i] == 0){
                continue;
            }
            max = max >= numbers[i] ? max : numbers[i];
            min = min <= numbers[i] ? min : numbers[i];
        }
        if(max - min < 5){
            return true;
        }else{
            return false;
        }
    }
};
```

- 46.每年六一儿童节,牛客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作为牛客的资深元老,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-1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,并且拿到牛客名贵的“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?(注:小朋友的编号是从0到n-1)

- 如果没有小朋友，请返回-1

```
class Solution {
public:
  int LastRemaining_Solution(int n, int m)
  {
    if(n==0){return -1;}

    int current = 0;
    for(int i=2; i<n+1; i++){
      current = (current + m) % i;
    }
    return current;
  }
};
```

- 47.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）。
  - 短路原理，sum等于0时，后面自动跳过，否则才需要看；

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

- 48.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。 +

```
/*
首先看十进制是如何做的： 5+7=12，三步走
第一步：相加各位的值，不算进位，得到2。
第二步：计算进位值，得到10. 如果这一步的进位值为0，那么第一步得到的值就是最终结果。
第三步：重复上述两步，只是相加的值变成上述两步的得到的结果2和10，得到12。

同样我们可以用三步走的方式计算二进制值相加： 5-101，7-111 第一步：相加各位的值，不算进位，
得到010，二进制每位相加就相当于各位做异或操作，101^111。
第二步：计算进位值，得到1010，相当于各位做与操作得到101，再向左移一位得到1010，(101&111)
<<1。
第三步重复上述两步， 各位相加 010^1010=1000，进位值为100=(010&1010)<<1。
  继续重复上述两步：1000^100 = 1100，进位值为0，跳出循环，1100为最终结果 */
```

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

- 49.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。 数值为0或者字符串不是一个合法的数值则返回0
    - 这道题除了最后返回之前的越界判断都是一遍过，代码量不小，细节需要注意。

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

        double sum = 0;
        for(int i=len-1; i>=1;i--){
            int num = charToInt(str[i]);
            if(num == -3){
                return 0;
            }
            sum += num * pow(10, len-i-1);
        }
        int num = charToInt(str[0]);
        if(num == -3){
            return 0;
        }else if(num == -2){
            sum *=-1;
        }else if(num != -1){
            sum += num * pow(10, len-1);
        }

        sum = (sum > INT32_MAX || sum < INT32_MIN)? 0 : sum;
        return sum;
    }

    int charToInt(char a){
        switch(a){
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
```

```
            return a - '0';
        case '+':
            return -1;
        case '-':
            return -2;
        default:
            return -3;
    }
  }
};
```

- 50.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}，那么对应的输出是第一个重复的数字2。

```
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) {
    map<int, int> result;
    for(int i=0; i<length; i++){
      if(result.find(numbers[i]) == result.end()){
        result[numbers[i]]=1;
      }else{
        *duplication = numbers[i];
        return true;
      }
    }
    return false;
  }
};
```

- 51.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。（注意：规定B[0] = A[1] * A[2] * ... * A[n-1]，B[n-1] = A[0] * A[1] * ... * A[n-2];）
    - 这题暴力法思路简单，但是复杂度大，利用下面的方法复杂度少了一级；不过好像乘法总数没少2333

```
/*
实现思路(非下图代码的方法，用到了数学归纳法)
既然不能用乘法，分析题目，我们可以将乘积拆为两项。即:
  C[i] = A[0] * A[1] *...*A[i-1]
```

```
    D[i] = A[i + 1] *...* A[n-1]
    B[i] = C[i] * D[i]
我们先来计算C[i]，使用数学归纳法：
    C[0] = 1
    C[1] = A[0]
    C[2] = A[0] * A[1]
    C[3] = A[0] * A[1] * A[2]
    ...
    C[i] = C[i-1] * A[i -1](i >=1)
我们继续用数学归纳法计算D[i]:
    D[n - 1] = 1
    D[n - 2] = A[n -1]
    D[n - 3] = A[n - 1] * A[n - 2]
    ...
    D[i] = D[i + 1]* A[i + 1](i <= n-2)
*/

class Solution {
public:
    vector<int> multiply(const vector<int>& A) {
        int len = A.size();
        vector<int> B(len, 1);
        for(int i=0; i<len; i++){
            for(int j=0; j<len; j++){
                if(i!=j){
                    B[j] *= A[i];
                }
            }
        }
        return B;
    }
};
```

- 52.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）。

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

```
        }
      }else{
        if(*str == *pattern || (*str != '\0' && *pattern == '.')){
          return match(str + 1, pattern + 1);
        }else{
          return false;
        }
      }
    }
};
```

- 53.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。
  - 思路不难，把e/E两边分别判断就好了，就是奇奇怪怪的边界条件有点烦。
  - strncpy(char* dst, const char* src, int maxlen)，这个函数好好记记，包括类似的。

```
class Solution
{
public:
  bool isNumeric(char *string)
  {
    int index = 0;
    while (string[index] != '\0')
    {
      if (string[index] == 'e' || string[index] == 'E')
      {
        char start[index + 1];
        strncpy(start, string, index);
        start[index] = '\0';
        return isDigit(start) >= 0 && isDigit(string + index + 1) == 1;
      }
      index++;
    }
    return isDigit(string) >= 0;
  }

  int isDigit(char *string)
  {
    //-1 for not num, 0 fot float, 1 fot int
    if (string[0] == '\0')
    {
      return -1;
    }
    int count = 0, point = 0, index = 0;
    while (string[index] != '\0')
    {
      if (index == 0 && (string[index] == '+' || string[index] == '-'))
      {
        if (count == 0)
        {
```

```
            count++;
        }
        else
        {
            return -1;
        }
      }
      else if (string[index] == '.')
      {
        if (point == 0)
        {
            point++;
        }
        else
        {
            return -1;
        }
      }
      else if (string[index] < '0' || string[index] > '9')
      {
        return -1;
      }
      index++;
    }
    if (point == 1)
    {
      return 0;
    }
    return 1;
  }
};
```

- 54.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"。

```
class Solution
{
public:
  map<char, int> result;
  queue<char> priority;
 //Insert one char from stringstream
  void Insert(char ch)
  {
    if(result.find(ch) == result.end()){
      result[ch] = 1;
      priority.push(ch);
    }else{
      result[ch]++;
    }
  }
```

```
  //return the first appearence once char in current stringstream
  char FirstAppearingOnce()
  {
    while(!priority.empty()){
      if(result[priority.front()] == 1){
        return priority.front();
      }else{
        priority.pop();
      }
    }
    return '#';
  }

};
```

- 55.给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。
  - 快慢指针赛跑

```
/*
struct ListNode {
  int val;
  struct ListNode *next;
  ListNode(int x) :
    val(x), next(NULL) {
  }
};
*/
class Solution {
public:
  ListNode* EntryNodeOfLoop(ListNode* pHead)
  {
    if(!pHead || !pHead->next || !pHead->next->next){return NULL;}
    ListNode *fast = pHead->next->next;
    ListNode *slow = pHead->next;
    while(slow != fast){
      if(!slow || !fast->next){return NULL;}
      slow = slow->next;
      fast = fast->next->next;
    }
    fast = pHead;
    while(fast != slow)
    {
      fast = fast->next;
      slow = slow->next;
    }
    return slow;
  }
};
```

- 56.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

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
        val(x), next(NULL) {
    }
};
*/
class Solution {
public:
    ListNode* deleteDuplication(ListNode* pHead)
    {
        if (!pHead || !pHead->next){return pHead;}
        ListNode *p = new ListNode(pHead->val-1);
        p->next = pHead;
        ListNode *pre  = p;
        ListNode *last = p->next;
        while (last){
            if(last->next && last->val == last->next->val){
                while (last->next && last->val == last->next->val){
                    last = last->next;
                }
                pre->next = last->next;
                last = last->next;
            }else{
                pre = pre->next;
                last = last->next;
            }
        }
        return p->next;
    }
};
```

- 65.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。

```
class Solution {
public:
    bool hasPath(char* matrix, int rows, int cols, char* str)
    {
        int *flag = new int[rows*cols];
        for(int i=0; i<rows*cols; i++){flag[i]=0;}
        for(int i=0; i<rows; i++){
            for(int j=0; j<cols; j++){
                if(subHasPath(matrix, rows, cols, i, j, 0, str, flag)){
```

```
                return true;
            }
          }
        }
        return false;
    }
    bool subHasPath(char* matrix, int rows, int cols, int row, int col, int len, char* str, int* flag){
        int index = row*cols+col;
        if(row < 0 || col < 0 || row >= rows || col >= cols || flag[index]==1 || matrix[index]!=str[len]){return
false;}
        if(len == strlen(str)-1){return true;}
        flag[index] = 1;
        if(subHasPath(matrix,rows,cols,row-1,col,len+1,str,flag)) return true;
        if(subHasPath(matrix,rows,cols,row+1,col,len+1,str,flag)) return true;
        if(subHasPath(matrix,rows,cols,row,col-1,len+1,str,flag)) return true;
        if(subHasPath(matrix,rows,cols,row,col+1,len+1,str,flag)) return true;
        flag[index] = 0;
        return false;
    }
};
```

- 66.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？

```
class Solution {
public:
    int movingCount(int threshold, int rows, int cols)
    {
        int *flag = new int[rows * cols];
        for(int i=0; i<rows*cols; i++){flag[i] = 0;}
        int count = 0;
        maxCount(threshold, rows, cols, 0, 0, flag, &count);
        return count;
    }
    void maxCount(int threshold, int rows, int cols, int i, int j, int *flag, int *count)
    {
        int index = i*cols+j;
        if(i<0 || j<0 || i>=rows || j>=cols || flag[index] == 1 || sum(i, j) > threshold){
            return;
        }
        (*count)++;
        flag[index] = 1;
        maxCount(threshold, rows, cols, i-1, j, flag, count);
        maxCount(threshold, rows, cols, i+1, j, flag, count);
        maxCount(threshold, rows, cols, i, j-1, flag, count);
        maxCount(threshold, rows, cols, i, j+1, flag, count);
    }

    int sum(int i, int j){
```

```
        int sum = 0;
        while(i!=0){
            sum+=i%10;
            i/=10;
        }
        while(j!=0){
            sum+=j%10;
            j/=10;
        }
        return sum;
    }
};
```

- 67.给你一根长度为n的绳子，请把绳子剪成整数长的m段（m、n都是整数，n>1并且m>1），每段绳子的长度记为k[0],k[1],...,k[m]。请问k[0]xk[1]x...xk[m]可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。
    - 我用的是动态规划，很神奇

```
class Solution {
public:
    int cutRope(int number) {
        if(number < 4){
            return number-1;
        }
        vector<int> result;
        result.push_back(0);
        result.push_back(1);
        result.push_back(2);
        result.push_back(3);
        for(int i=4; i<=number; i++){
            int max = 0;
            for(int j=1; j<=i/2; j++){
                max = max >= (result[j] * result[i-j])? max: result[j] * result[i-j];
            }
            result[i] = max;
        }
        return result[number];
    }
};
```

Tree

- 17.输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

```
/*
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 ||!pRoot2){return false;}
    if(pRoot1->val == pRoot2->val){
      if(isSub(pRoot1, pRoot2)){
        return true;
      }
    }
    return HasSubtree(pRoot1->left, pRoot2) || HasSubtree(pRoot1->right, pRoot2);
  }

  bool isSub(TreeNode *r1, TreeNode *r2)
  {
    if(!r2){return true;}
    if(!r1){return false;}
    if(r1->val != r2->val){return false;}
    return isSub(r1->left, r2->left) && isSub(r1->right, r2->right);
  }
};
```

- 18.操作给定的二叉树，将其变换为源二叉树的镜像。(即二叉树左右节点交换)

```
/*
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 *p) {
    if(!p || (!p->left && !p->right)){return;}
    TreeNode *temp = p->left;
    p->left = p->right;
    p->right = temp;
    Mirror(p->left);
    Mirror(p->right);
  }
};
```

- 22.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。

- 不难，就是用queue辅助广度遍历；

```
/*
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> result;
    if(!root){return result;}
    queue<TreeNode*> temp;
    temp.push(root);
    while(!temp.empty()){
      TreeNode *node = temp.front();
      temp.pop();

      result.push_back(node->val);
      if(node->left){
        temp.push(node->left);
      }
      if(node->right){
        temp.push(node->right);
      }
    }
    return result;
  }
};
```

- 23.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。

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

  bool ifSequence(vector<int> sequence, int start, int end){
    int root = sequence[end], index;
    bool side = false;
    for(int i=end; i>=start; i--){
```

```
            if(sequence[i] < root && !side){
                index = i;
                side = true;
            }
            if(side && sequence[i] > root){
                return false;
            }
        }
        return ifSequence(sequence, start, index) && ifSequence(sequence, index+1, end-1);
    }
};
```

- 24.输入一颗二叉树的根节点和一个整数，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。(注意: 在返回值的list中，数组长度大的数组靠前)

```
/*
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> > result;
    vector<int> tmp;
    vector<vector<int> > FindPath(TreeNode* root,int target) {
        if(root){
            find(root, target);
        }
        return result;
    }

    void find(TreeNode *root, int target){
        tmp.push_back(root->val);
        if(root->val == target && !root->left && !root->right){
            result.push_back(tmp);
        }
        if(root->left){
            find(root->left, target - root->val);
        }
        if(root->right){
            find(root->right, target - root->val);
        }
        tmp.pop_back();
    }
};
```

- 25.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），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

```
import java.util.Map;
import java.util.HashMap;
public class Solution {
    public RandomListNode Clone(RandomListNode pHead)
    {
        if(pHead == null)return null;
        RandomListNode newHead = null;
        RandomListNode p = pHead;
        RandomListNode q = null;
        Map<RandomListNode, RandomListNode> map = new HashMap<>();
        while(p != null){
            if(newHead == null){
                newHead = new RandomListNode(pHead.label);
                q = newHead;
                map.put(pHead, newHead);
            }else{
                if(p.next!= null && map.containsKey(p.next))
                    q.next = map.get(p.next);
                else{
                    if(p.next!= null){
                        RandomListNode temp = new RandomListNode(p.next.label);
                        map.put(p.next, temp);
                        q.next = temp;
                    }
                }
                if(p.random != null && map.containsKey(p.random))
                    q.random = map.get(p.random);
                else{
                    if(p.random != null){
                        RandomListNode temp = new RandomListNode(p.random.label);
                        map.put(p.random, temp);
                        q.random = temp;
                    }
                }
                p = p.next;
                q = q.next;
            }
        }
        return newHead;
    }
}
```

- 26.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。
    - 这个方法挺巧妙的，值得深思。

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
        val(x), left(NULL), right(NULL) {
    }
};*/
class Solution {
public:
  TreeNode *pre = NULL;
  TreeNode* Convert(TreeNode* root)
  {
    if(!root){return NULL;}
    Convert(root->right);
    if(!pre){
      pre = root;
    }else{
      root->right = pre;
      pre->left = root;
      pre = root;
    }
    Convert(root->left);
    return pre;
  }
};
```

- 27.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。

```
class Solution {
public:
  vector<string> Permutation(string str) {
    vector<string> result;
    if(str.size() == 0){return result;}
    if(str.size() == 1){
      result.push_back(str);
      return result;
    }
    for(int i=0; i<str.size(); i++){
      if(i!=0 && str[i] == str[0]){continue;}
      vector<string> temp = Permutation(str.substr(0, i) + str.substr(i+1));
      for(int j=0; j<temp.size(); j++){
        result.push_back(str[i]+temp[j]);
      }
    }
    return result;
  }
};
```

- 38.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。

```
/*
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){return 0;}
    int left = 1 + TreeDepth(pRoot->left);
    int right = 1 + TreeDepth(pRoot->right);
    return left < right ? right : left;
  }
};
```

- 39.输入一棵二叉树，判断该二叉树是否是平衡二叉树。
  - 即高度差是否大于1

```
class Solution {
public:
  bool IsBalanced_Solution(TreeNode* pRoot) {
    return depth(pRoot) != -1;
  }

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

- 57.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。

    - 分两种情况：如果有右节点，那就找到右节点往下的最左节点；如果没有右节点，那就找到父节点往上的某级节点使得自己成为其左节点，这时父节点就是下一个节点。

```
/*
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){
            return NULL;
        }
        TreeLinkNode *next = NULL;

        if(pNode->right != NULL){
            TreeLinkNode* temp = pNode->right;
            while(temp->left){
                temp = temp->left;
            }
            next = temp;
        }
        else{
            TreeLinkNode* currentNode = pNode;
            TreeLinkNode* parentNode = pNode->next;
            while(parentNode && currentNode == parentNode->right){
                currentNode = parentNode;
                parentNode = parentNode->next;
            }
            next = parentNode;
        }
        return next;
    }
};
```

- 58.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。
    - 递归比较简单，BFS应该也可以做，只是不太好

```
/*
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){return true;}
        return f(pRoot->left, pRoot->right);
    }

    bool f(TreeNode* r1, TreeNode* r2){
        if(!r1 && !r2){
            return true;
        }
        if(r1 && r2){
            return r1->val==r2->val && f(r1->left, r2->right) && f(r1->right, r2->left);
        }
        return false;
    }
};
```

- 59.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。
    - 与上题按行打印基本一致，唯一不同的就是需要奇偶行判断一下是否需要reverse(a.begin(), a.end());

```
/*
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> > output;
        if(!pRoot){return output;}

        queue<TreeNode*> a, b;
        int count = 1;
        vector<int> result;
        result.push_back(pRoot->val);
```

```
        output.push_back(result);
        result.resize(0);
        a.push(pRoot);
        while(!a.empty()){
            TreeNode* temp = a.front();
            a.pop();
            if(temp->left){
                result.push_back(temp->left->val);
                b.push(temp->left);
            }
            if(temp->right){
                result.push_back(temp->right->val);
                b.push(temp->right);
            }
            if(a.empty()){
                if(!b.empty()){
                    while(!b.empty()){
                        a.push(b.front());
                        b.pop();
                    }
                    if(count % 2 == 1){
                        reverse(result.begin(), result.end());
                        output.push_back(result);
                    }else{
                        output.push_back(result);
                    }
                    count++;
                    result.resize(0);
                }
            }
        }
        return output;
    }

};
```

- 60.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。
    - queue清空用clear()

```
/*
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> > output;
        if(!pRoot){return output;}
        vector<int> result;
        result.push_back(pRoot->val);
        output.push_back(result);
        result.resize(0);
        queue<TreeNode*> a, b;
        a.push(pRoot);
        while(!a.empty()){
            TreeNode* temp = a.front();
            a.pop();
            if(temp->left){
                result.push_back(temp->left->val);
                b.push(temp->left);
            }
            if(temp->right){
                result.push_back(temp->right->val);
                b.push(temp->right);
            }
            if(a.empty()){
                if(!b.empty()){
                    while(!b.empty()){
                        a.push(b.front());
                        b.pop();
                    }
                    output.push_back(result);
                    result.resize(0);
                }
            }
        }
        return output;
    }
};
```

- 61.请实现两个函数，分别用来序列化和反序列化二叉树
    - 二叉树的序列化是指：把一棵二叉树按照某种遍历方式的结果以某种格式保存为字符串，从而使得内存中建立起来的二叉树可以持久保存。序列化可以基于先序、中序、后序、层序的二叉树遍历方式来进行修改，序列化的结果是一个字符串，序列化时通过 某种符号表示空节点（#），以！ 表示一个结点值的结束（value!）。
    - 二叉树的反序列化是指：根据某种遍历顺序得到的序列化字符串结果str，重构二叉树。

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

```
*/
class Solution {
public:
    char* Serialize(TreeNode *root) {
        if(!root){
            return "#";
        }
        string val = to_string(root->val);
        val += '!';
        char* left = Serialize(root->left);
        char* right = Serialize(root->right);
        char* result = new char[val.size()+strlen(left)+strlen(right)];
        strcpy(result, val.c_str());
        strcat(result, left);
        strcat(result, right);
        return result;
    }
    TreeNode* Deserialize(char *str) {
        return decode(str);
    }

    TreeNode* decode(char* &str){
        if(*str == '#'){
            str++;
            return NULL;
        }
        int val = 0;
        while(*str != '!'){
            val = 10 * val + (*str - '0');
            str++;
        }
        str++;
        TreeNode* node = new TreeNode(val);
        node->left = decode(str);
        node->right = decode(str);
        return node;

    }
};
```

- 62.给定一棵二叉搜索树，请找出其中的第k小的结点。例如， （5，3，7，2，4，6，8） 中，按结点数值大小顺序第三小结点的值为4。
    - 这个要多看看，递归之后看着不难，但是还是要多记记

```
/*
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, int k)
    {
        TreeNode* res = NULL;
        int c = 0;
        inorder(pRoot, &res, k, &c);
        return res;
    }

    void inorder(TreeNode* root, TreeNode** res, int k, int *c){
        if(root){
            inorder(root->left, res, k, c);
            (*c)++;
            if(*c==k){*res = root;}
            inorder(root->right, res, k, c);
        }
    }
};
```

- 63.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。我们使用Insert()方法读取数据流，使用GetMedian()方法获取当前读取数据的中位数。

```
class Solution {
public:
    vector<int> max;
    vector<int> min;
    void Insert(int num)
    {
        int len = min.size() + max.size();
        if((len&1) == 0){
            if(max.size() > 0 && num < max[0]){
                max.push_back(num);
                push_heap(max.begin(), max.end(), less<int>());
                num = max[0];
                pop_heap(max.begin(), max.end(), less<int>());
                max.pop_back();
            }
            min.push_back(num);
            push_heap(min.begin(), min.end(), greater<int>());
        }else{
            if(min.size() > 0 && num > min[0]){
                min.push_back(num);
                push_heap(min.begin(), min.end(), greater<int>());
                num = min[0];
                pop_heap(min.begin(), min.end(), greater<int>());
                min.pop_back();
```

```
            }
            max.push_back(num);
            push_heap(max.begin(), max.end(), less<int>());
        }
    }

    double GetMedian()
    {
        int len = min.size()+max.size();
        if(len <= 0){
            return 0;
        }
        if((len&1) == 0){
            return (max[0] + min[0])/2.0;
        }else{
            return min[0];
        }
    }
};
```

- 64.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]}。

```
class Solution {
public:
    vector<int> maxInWindows(const vector<int>& num, unsigned int size)
    {
        vector<int> res;
        if(num.empty() || size>num.size() || size<1)
            return res;
        for(int i=0;i<=num.size()-size;i++){//两层循环，外层从数组的起点到数组末尾前size位
            int max=num[i];//将每一次循环的起始点设为最大值
            for(int j=i+1;j<i+size;j++){//内层循环遍历循环起始点后的数字，注意不越界
                if(max<num[j])//比较遍历的数字与之前最大值的大小
                    max=num[j];//给最大值重新赋值
            }
            res.push_back(max);
        }
        return res;
    }
};
```