企业要求 & 我们的目标

- 校招算法要求较高，尤其是名企(当然，企业考察不止考察代码思维，还会考察其他方面，我只想表述算法很重要)
- 有很多算法题目，作为应试者要能明白题目考察的核心点在哪里，核心点思路一定要明确，如果该思路明确了，哪怕代码最终没有写的很完美，甚至没有完成，也是有分数可以拿的，一定不要交白卷或者不回答
- 校招算法考察的内容比较多，核心主要是 1. 以特殊数据结构为载体的算法题，如：字符串，数组，链表，堆，栈，二叉树，hash等。 2. 以考察常见算法思想基础的算法题，如动归，贪心，分治，回溯，排序，查找等 3. 基于某种场景包装的1 or 2
- 所以我们的训练营内容设计主要包括，算法基础(解决1)，算法专题(解决2)，算法真题(熟悉3的方式)
- 目标1：不是会所有的算法题，这个也很难做到。我们的目标是基于算法练习达到，精于结构，敏于心智，熟于代码。对于已经会的，要能正确高效实现思想转化为代码，对于不会的，通过研究，能看懂别人的代码思想，从而以最短路径的方式，爬到巨人的肩膀。
- 目标2：熟悉使用参考文档，把使用参考文档作为一种习惯
- 目标3：深刻理解目前的题目，为后面打好基础

开始

1. 核心考点：数组相关，特性观察，时间复杂度把握 （day1）

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/abc3fe2ce8e146608e868a70efebf62e?tpId=13&tqId=11154&tPage=1&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//数组操作问题
//解决思路：
//如数组样式如下：
//   1 2 3 4
//   2 3 4 5
//   3 4 5 6
//正常查找的过程，本质就是排除的过程，如果双循环查找，本质是一次排除一个，效率过低
//根据题面要求，我们可以采取从右上角（或左下角）进行比较（想想为什么？），这样可以做到一次排除一行或者一列
```

C++代码

```
class Solution {
public:
    bool Find(int target, vector<vector<int> > array) {
        int i = 0;
        int j = array[0].size()-1;

        while( i < array.size() && j >= 0){
            if(target < array[i][j]){ //array[i][j]一定是当前行最大的，当前列最小的
```

```
                //target < array[i][j] 排除当前列
                j--;
            }
            else if(target > array[i][j]){
                //target > array[i][j] 排除当前行
                i++;
            }
            else{
                //找到
                return true;
            }
        }
        return false;
    }
};
```

Java代码

```
//语言切换时，OJ有时候会让同学们改类名，不要改，要不然会出现奇怪错误，应该是网站的bug
public class Solution {
        public boolean Find(int target, int [][] array) {
        if(array == null){
            return false;
        }
        int i = 0;
        int j = array[0].length - 1;
        while( i < array.length && j >= 0){
            if(target < array[i][j]){//array[i][j]一定是当前行最大的，当前列最小的
                //target < array[i][j] 排除当前列
                j--;
            }
            else if(target > array[i][j]){
                //target > array[i][j] 排除当前行
                i++;
            }
            else{
                //找到
                return true;
            }
        }
        return false;
    }
}
```

2. 核心考点：字符串相关，特性观察，临界条件处理

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/4060ac7e3e404ad1a894ef3e17650423?tpId=13&tqId=11155&tPage=1&rp=1&ru=%2Fta%2Fcoding-interviews&qru=%2Fta%2Fcoding-interviews%2Fquestion-ranking
//字符串操作问题
//解决思路：
//虽然是替换问题，但是生成的字符串整体变长了.
//因替换内容比被替换内容长，所以，一定涉及到字符串中字符的移动问题
//移动方向一定是向后移动，所以现在的问题无非是移动多少的问题
//因为是 ' ' -> "%20",是1换3，所以可以先统计原字符串中空格的个数(设为n)，然后可以计算出新字符串的长度
//所以：new_length = old_length + 2*n
//最后，定义新老索引（或者指针），各自指向新老空间的结尾，然后进行old->new的移动
//如果是空格，就连续放入“%20”,其他平移即可。
//当然，C++和Java都有很多容器，也可以从前往后通过开辟空间来进行解决。也就是使用空间来换取时间。
//但是，我们最好不要在当前场景下这么做
```

C++代码

```
class Solution {
public:
    void replaceSpace(char *str,int length) {
        int count = 0;
        char *start = str;
        while(*start){
            if(isspace(*start)){
                count++;
            }
            start++;
        }
        char *old_end = str+length; //C++要考虑'\0'
        char *new_end = str + length + 2*count;
        while(old_end>=str && new_end >= str){
            if(!isspace(*old_end)){
                *new_end = *old_end;
                new_end--,old_end--;
            }
            else{
                *new_end--='0';
                *new_end--='2';
                *new_end--='%';
                old_end--;
            }
        }
    }
};
```

Java代码

```
public class Solution {
    public String replaceSpace(StringBuffer str) {
        int count = 0;
        for(int i = 0; i < str.length(); i++){
            if(str.charAt(i) == ' '){
```

```
                count++;
            }
        }
        int new_length = str.length() + 2*count;
        int old_end = str.length() - 1; //索引老字符串最后一个有效位置
        int new_end = new_length - 1;   //索引新字符串最后一个有效位置
        str.setLength(new_length);      //设置字符串新大小，防止越界

        while(old_end >= 0 && new_end >= 0){
            if(str.charAt(old_end) == ' '){
                //当前位置是空格
                str.setCharAt(new_end--, '0');
                str.setCharAt(new_end--, '2');
                str.setCharAt(new_end--, '%');
                --old_end;
            }
            else{
                //当前位子不是空格，平移即可
                str.setCharAt(new_end--, str.charAt(old_end));
                --old_end;
            }
        }
        return str.toString();
    }
}
```

3. 核心考点：链表相关，多结构混合使用，递归

输入一个链表，按链表从尾到头的顺序返回一个ArrayList。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/d0267f7f55b3412ba93bd35cfa8e8035?
tpId=13&tqId=11156&tPage=1&rp=1&ru=%2Fta%2Fcoding-interviews&qru=%2Fta%2Fcoding-
interviews%2Fquestion-ranking
//链表问题
//解题思路：
//这道题整体解决思路很多，可以递归，也可以将数据保存数组，逆序数组
//我们可以三种方法都实现一下，具体可以酌情编写
```

C++代码

```
/**
*  struct ListNode {
*        int val;
*        struct ListNode *next;
*        ListNode(int x) :
*              val(x), next(NULL) {
*        }
*  };
*/
//方法一，stack方式，这种方式会有内存占用过多的问题
```

```
class Solution {

public:
    vector<int> printListFromTailToHead(ListNode* head) {
        stack<int> s;
        vector<int> v;
        while(head){
            s.push(head->val);
        }
        while(!s.empty()){
            v.push_back(s.top());
            s.pop();
        }
        return v;
    }
};

//方法二，逆置数组
class Solution {

public:
    vector<int> printListFromTailToHead(ListNode* head) {
        vector<int> v;
        while(head){
            v.push_back(head->val);
            head = head->next;
        }

        reverse(v.begin(), v.end());
        return v;
    }
};

//方法三，递归方式
class Solution {
public:
    void printListFromTailToHeadCore(ListNode* head, vector<int> &v){
        if(nullptr == head){
            return;
        }
        printListFromTailToHeadCore(head->next, v);
        v.push_back(head->val);
    }
    vector<int> printListFromTailToHead(ListNode* head) {
        vector<int> v;
        printListFromTailToHeadCore(head, v);
        return v;
    }
};
```

Java代码

```
/**
*    public class ListNode {
*        int val;
*        ListNode next = null;
*
*        ListNode(int val) {
*            this.val = val;
*        }
*    }
*
*/

import java.util.Stack;
import java.util.ArrayList;
//方法一，stack方式
public class Solution {
      public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        Stack<Integer> st = new Stack<>();
        while(listNode != null){
            st.push(listNode.val);
            listNode = listNode.next;
        }

        ArrayList<Integer> list = new ArrayList<>();
        while(!st.empty()){
            list.add(st.pop());
        }

        return list;
    }
}

//方法二，逆置数组
public class Solution {
      public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
          ArrayList<Integer> list = new ArrayList<>();
          while(listNode != null){
              list.add(listNode.val);
              listNode = listNode.next;
          }

          int i = 0;
          int j = list.size() - 1;
          while(i < j){
              Integer temp = list.get(i);
              list.set(i, list.get(j));
              list.set(j, temp);
              i++;
              j--;
          }
          return list;
    }
}
```

```
//方法三，递归方式
public class Solution {
      public void printListFromTailToHeadCore(ArrayList<Integer> list, ListNode listNode)
{
          if(listNode == null){
              return;
          }
          printListFromTailToHeadCore(list, listNode.next);
          list.add(listNode.val);
      }
      public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
          ArrayList<Integer> list = new ArrayList<>();
          printListFromTailToHeadCore(list, listNode);
          return list;
    }
}
```

4. 核心考点：二叉树重建，遍历理解，递归

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/8a19cbe657394eeaac2f6ea9b0f6fcf6?
tpId=13&tqId=11157&tPage=1&rp=1&ru=%2Fta%2Fcoding-interviews&qru=%2Fta%2Fcoding-
interviews%2Fquestion-ranking
//经典二叉树问题
//解题思路：
//根据root节点，将中序vector划分成vin_left，vin_right两部分中序子序列
//根据中序子序列长度，将前序vector划分成pre_left，pre_right对应的前序子序列
//root->left递归生成
//root->right递归生成
```

C++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:
    TreeNode* reConstructBinaryTreeCore(vector<int> pre,int preStart,int preEnd,vector<int>
vin, int vinStart, int vinEnd) {
        if(preStart > preEnd || vinStart > vinEnd){
```

```
            return nullptr;
        }
        TreeNode *root = new TreeNode(pre[preStart]);
        for(auto i = vinStart; i <= vinEnd; i++){//在中序序列中，找根节点，可以将数组划分为两部分
            if(pre[preStart] == vin[i]){
                //前序的第一个节点，是root，能将中序划分为两部分
                //一棵树，无论前，中，后怎么遍历，元素的个数是不变的
                //在实际遍历的时候，前，中，后序遍历，各种遍历方式左右子树的节点都是在一起的
                //所以这里重点是要想清楚下标问题
                //根据中序，我们能确认左子树的节点个数是：i - vinStart（没有从0开始哦）
                //所以，需要从preStart+1，连续i - vinStart个元素，就是左子树的前序序列
                root->left = reConstructBinaryTreeCore(pre,preStart+1, i-vinStart+preStart,
vin, vinStart, i-1);
                root->right = reConstructBinaryTreeCore(pre, i-vinStart+preStart+1,
preEnd,vin, i+1,vinEnd);
                break;

            }
        }
        return root;
    }
    TreeNode* reConstructBinaryTree(vector<int> pre,vector<int> vin) {
        if (pre.empty() || vin.empty()){
            return nullptr;
        }
        return reConstructBinaryTreeCore(pre, 0, pre.size()-1, vin, 0, vin.size()-1);
    }
};
```

Java代码

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode reConstructBinaryTreeCore(int [] pre,int preStart, int preEnd,int [] in,
int inStart, int inEnd) {
        if(preStart > preEnd || inStart > inEnd){
            return null;
        }

        TreeNode root = new TreeNode(pre[preStart]);
        int i = inStart;
        for(; i <= inEnd; i++){ //在中序序列中，找根节点，可以将数组划分为两部分
            if(in[i] == pre[preStart]){
                //前序的第一个节点，是root，能将中序划分为两部分
                //一棵树，无论前，中，后怎么遍历，元素的个数是不变的
```

```
                //在实际遍历的时候，前，中，后序遍历，各种遍历方式左右子树的节点都是在一起的
                //所以这里重点是要想清楚下标问题
                //根据中序，我们能确认左子树的节点个数是：i - inStart（没有从0开始哦）
                //所以，需要从preStart+1，连续i - inStart个元素，就是左子树的前序序列
                root.left = reConstructBinaryTreeCore(pre,preStart+1, i-inStart+preStart,
in, inStart, i-1);
                //右子树同理
                root.right = reConstructBinaryTreeCore(pre, i-inStart+preStart+1,preEnd, in,
i+1, inEnd);
                break;
            }
        }
        return root;
    }
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        if( pre.length == 0 || in.length == 0){
            return null;
        }
        //理论上，可以新建数组，保存前序，中序子序列，但是就需要花费额外空间
        //所以，我们采取在原数组内进行操作
        //使用闭区间限定数组范围
        return reConstructBinaryTreeCore(pre, 0, pre.length-1, in, 0, in.length-1);
    }
}
```

5. 核心考点：数组理解，二分查找，临界条件 （day1）

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/9f3231a991af4f55b95579b44b7a01ba?
tpId=13&tqId=11159&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//题目解析
//数组问题，本质其实是一个求最小值问题
//方法一：理论上，遍历一次即可，但是我们可以根据题面使用稍微高效且更简单一点的做法
//按照要求，要么是一个非递减排序的数组（最小值在最开始），要么是一个旋转(最小值在中间某个地方)
//而且，旋转之后有个特征，就是在遍历的时候，原始数组是非递减的，旋转之后，就有可能出现递减，引起递减的数字，就是最小值
//方法二：采用二分查找的方式，进行定位
//定义首尾下标，因为是非递减数组旋转，所以旋转最后可以看做成两部分，前半部分整体非递减，后半部分整体非递减，前半部分整体大于后半部分。
//所以，我们假设如下定义，left指向最左侧，right指向最右侧，mid为二分之后的中间位置。
//则，a[mid] >= a[left],说明mid位置在原数组前半部分，进一步说明，目标最小值，在mid的右侧，让left=mid
//a[mid] < a[left]，说明mid位置在原数组后半部分，进一步说明，目标最小值，在mid的左侧，让right=mid
//这个过程，会让[left, right]区间缩小
//这个过程中，left永远在原数组前半部分，right永远在原数组的后半部分，而范围会一直缩小
//当left和right相邻时，right指向的位置，就是最小元素的位置
//但是，因为题目说的是非递减，也就意味着数据允许重复，因为有重复发，就可能会有a[left] == a[mid] == a[right]的情况，我们就无法判定数据在mid左侧还是右侧。（注意，只要有两者不相等，我们就能判定应该如何缩小范围）
```

C++代码

```
//方法一
class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        if(rotateArray.empty()){
            return 0;
        }
        for(int i = 0; i < rotateArray.size()-1; i++){
            if(rotateArray[i] > rotateArray[i+1]){
                return rotateArray[i+1];
            }
        }
        return rotateArray[0];
    }
};
//方法二
class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        if(rotateArray.empty()){
            return 0;
        }

        int left = 0;
        int right = rotateArray.size() - 1;
        int mid = 0;
        //要一直满足该条件，以证明旋转特性
        while(rotateArray[left] >= rotateArray[right]){
            if(right - left == 1){
                //两个下标已经相邻了
                mid = right;
                break;
            }
            mid = left + ((right - left) >> 1); //注意操作符优先级
            if(rotateArray[mid] == rotateArray[left] && rotateArray[left] ==
rotateArray[right]){
                //无法判定目标数据在mid左侧，还是右侧我们采用线性遍历方式
                int result = rotateArray[left];
                for(int i = left+1; i < right; i++){
                    if(result > rotateArray[i]){
                        result = rotateArray[i];
                    }
                }
                return result;
            }
            if(rotateArray[mid] >= rotateArray[left]){ //试想两者相等， 隐含条件
rotateArray[left] >= rotateArray[right]
                //说明mid在前半部分
                left = mid;
            }
            else{
```

```
                //说明mid在后半部分
                right = mid;
            }
        }
        return rotateArray[mid];
    }
};
```

Java代码

```
//方法一
import java.util.ArrayList;
public class Solution {
    public int minNumberInRotateArray(int [] array) {
        if(array == null || array.length == 0){
            return 0;
        }

        for(int i = 0; i < array.length-1; i++){
            if(array[i] > array[i+1]){
                return array[i+1];
            }
        }

        return array[0];
    }
}
//方法二
import java.util.ArrayList;
public class Solution {
    public int minNumberInRotateArray(int [] array) {
        if(array == null || array.length == 0){
            return 0;
        }
        int left = 0;
        int right = array.length - 1;
        int mid = 0;
        while(array[left] >= array[right]){ left<right也可以
            if(right - left == 1){
                mid = right;
                break;
            }
            mid = left + ((right - left)>>1);
            if(array[left] == array[right] && array[mid] == array[left]){ //1
                int result = array[left];
                for(int i = left+1; i < right; i++){ //left和right值是相等的
                    if(array[i] < result){
                        result = array[i];
                    }
                }
                return result;
            }
            if(array[mid] >= array[left]){
```

```
                //说明mid在原数组的前半部分
                //如果array[mid] == array[left]， 上面1处的条件不满足且array[left] >=
array[right]，则，array[mid] > array[right]
                left = mid;
            }
            else{
                right = mid;
            }
        }
        return array[mid];
    }
}
```

6. 核心考点：空间复杂度，fib理解，剪枝重复计算

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，第1项是1）。

n<=39

解析：

```
//OJ链接：https://www.nowcoder.com/practice/c6c7742f5ba7442aada113136ddea0c3?
tpId=13&tqId=11160&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//题目解析
//斐波那契数列是 0 1 1 2 3 5 8 13 21 ...
//解题方式很多，有递归方式，也有动归(迭代)方式，但是都是最简单的方式
```

C++代码：

```
//迭代方案
class Solution {
public:
    int Fibonacci(int n) {
        if(n == 0)
            return 0;
        int first = 1;
        int second = 1;
        int third = 1;
        while(n > 2){
            third = first + second;
            first = second;
            second = third;
            n--;
        }

        return third;
    }
};
//递归方案
//直接用最简单的方式因为代码空间复杂度过高，过不了OJ，所以我们可以采用map进行“剪枝”
class Solution {
```

```
    private:
        unordered_map<int, int> filter;
    public:
        int Fibonacci(int n) {
            if(n == 0 || n == 1){
                return n;
            }
            int pre = 0;
            if(filter.find(n-1) == filter.end()){
                pre = Fibonacci(n-1);
                filter.insert({n-1, pre});
            }else{
                pre = filter[n-1];
            }
            int ppre = 0;
            if(filter.find(n-2) == filter.end()){
                ppre = Fibonacci(n-2);
                filter.insert({n-2, ppre});
            }else{
                ppre = filter[n-2];
            }

            return  pre + ppre;
        }
};
```

Java代码：

```
//迭代方案
public class Solution {
    public int Fibonacci(int n) {
        if(n == 0){
            return 0;
        }
        int first = 1;
        int second = 1;
        int third = 1; //因为从0开始，third等于n就不用判定了
        while(n > 2){
            third = first + second;
            first = second;
            second = third;
            --n;
        }
        return third;
    }
}

////递归方案
import java.util.HashMap;
import java.util.Map;
public class Solution {
    private Map<Integer, Integer> filter = new HashMap<>();
    public int Fibonacci(int n) {
```

```
        if(n == 0 || n == 1){
            return n;
        }
      //  filter = new HashMap<>();
        int pre = 0;
        if(filter.containsKey(n-1)){
            pre = filter.get(n-1);
        }
        else{
            pre = Fibonacci(n-1);
            filter.put(n-1, pre);
        }
        int ppre = 0;
        if(filter.containsKey(n-2)){
            ppre = filter.get(n-2);
        }
        else{
            ppre = Fibonacci(n-2);
            filter.put(n-2, ppre);
        }
        return pre + ppre;
    }
}
```

7. 核心考点：场景转化模型，模型提取解法，简单dp，fib

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）。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/8c82a5b80378478f9484d87d1c5f12a4?tpId=13&tqId=11161&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解析：
//解决方案很多，可以使用dp，也可以采用上题的思路
```

C++代码：

```
//方法一：简单动归方式-目前了解
//状态定义：f(i)：跳到i台阶的总跳法
//状态递推：f(i) = f(i-1)+f(i-2)
//初始状态：f(0) = 1（0台阶，就是起点，到达0台阶的方法有一种，就是不跳[这里可能有点奇怪，但是想想，如果方法次数为0，就说明不可能开始...]），f(1) = 1;
class Solution {
public:
    int jumpFloor(int number) {
        //dp[n] = dp[n-1]+dp[n-2];
        int *dp = new int[number+1];
        dp[0] = 1;
        dp[1] = 1;
        for(int i = 2; i<= number;i++){
            dp[i] = dp[i-1] + dp[i-2];
```

```
        }
        int num = dp[number]; //第number下标，就是第number阶台阶
        delete dp;
        return num;
    }
};
//方法二:
//在仔细看看这个代码，难道不像上题的斐波那契数列吗?
class Solution {
public:
    // 状态: f(n): 到第n阶台阶的总跳法，而到了n，只能是从n-1 or n-2跳过来的
    // 递推公式: 所以f(n) = f(n-1) + f(n-2)
    // 初始值: f(1) = 1, f(2) = 2,  可以看成，从平地(0)开始跳到1台阶，方法1种。或者跳到2台阶，有两种方式
    int jumpFloor(int n) {
        int first = 1; //第一个台阶
        int second = 2;//第二个台阶
        int third = n; //等于n直接就考虑了f(1) = 1 && f(2) = 2的情况
        while(n > 2){
            third = first + second;
            first = second;
            second = third;
            --n;
        }
        return third;
    }
};
```

Java代码:

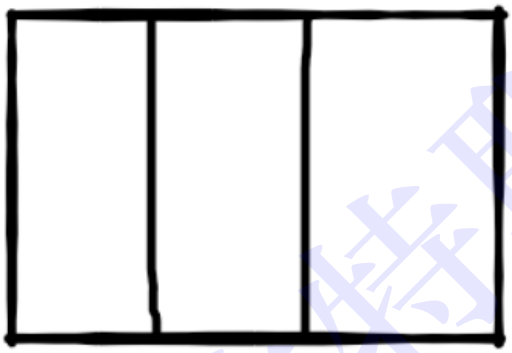
```
//方法一: 简单动归方式-目前了解
//状态定义: f(i): 跳到i台阶的总跳法
//状态递推: f(i) = f(i-1)+f(i-2)
//初始状态: f(0) = 1 (0台阶，就是起点，到达0台阶的方法有一种，就是不跳[这里可能有点奇怪，但是想想，如果方法次数为0，就说明不可能开始...]) , f(1) = 1;
public class Solution {
    public int JumpFloor(int target) {
        int [] dp = new int[target+1];
        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2; i <= target; i++){
            dp[i] = dp[i-1] + dp[i-2];
        }

        return dp[target];
    }
}
//方法二: 斐波那契数列
public class Solution {
    public int JumpFloor(int target) {
        int first = 1;
        int second = 2;
        int third = target;
```

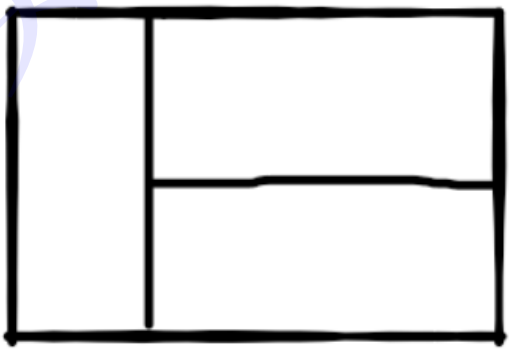
```
        while(target > 2){
            third = first + second;
            first =  second;
            second = third;
            --target;
        }

        return third;
    }
}

//后面的专题栏目，会针对性训练DP用法，此处仅仅为热身
```

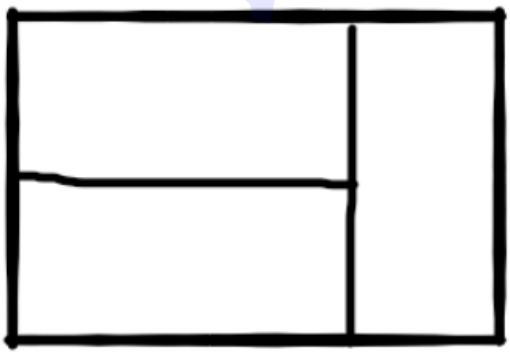
7. 核心考点：场景转化成模型，特殊情况分析，简单dp

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：







解析：

```
//OJ链接：https://www.nowcoder.com/practice/72a5a919508a4251859fb2cfb987a0e6?tpId=13&tqId=11163&tPage=1&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解析：
//用n个2*1的小矩形无重叠地覆盖一个2*n的大矩形，每次放置的时候，无非两种放法，横着放或竖着放
//其中，横着放一个之后，下一个的放法也就确定了，故虽然放置了两个矩形，但属于同一种放法
//其中，竖着放一个之后，本轮放置也就完成了，也属于一种方法
//所以，当2*n的大矩形被放满的时候，它无非就是从上面两种放置方法放置来的。
//我们继续使用dp来进行处理，当然后续会发现，斐波那契数列的方式也可以处理，因为之前已经讲过，就留给大家完成
//状态定义：f(n) ： 用n个2*1的小矩形无重叠地覆盖一个2*n的大矩形所用的总方法数
//状态递推：f(n) = f(n-1)【最后一个竖着放】 + f(n-2)【最后两个横着放】
//初始化：f(1) = 1,f(2) = 2,f(0)=1,注意f(0)我们这个可以不考虑，如果考虑值设为1，参考上题（这点确实有点蛋疼）
```

C++代码：

```
class Solution {
public:
    int rectCover(int number) {
        if(number < 2){ //这里要充分考虑number是[0,1]时的情况，OJ一般测试用例设计的比较全面，会有0，1传进来，这个时候，后续的dp[1] = 1;就可能报错
            return number;
        }
        //f(n) = f(n-1)+f(n-2)
        int *dp = new int[number+1];
        dp[1] = 1;
        dp[2] = 2;
        for( int i = 3; i <= number; i++){
            dp[i] = dp[i-1]+dp[i-2];
        }
        int num = dp[number];
        delete dp;
        return num;
    }
};

//这不就是菲波那切数列问题吗？我们反思一下，很多问题会包裹很多现实问题，解决问题的第一步往往是从实际问题中提炼出我们的解决问题的数学模型，然后在解决之
//这里也可以使用多种方法解决，不过我们这里重点用dp，倒不是说他是最优的，而是同学们平时在写代码的时候，这种思想用得少，我们就多用用
```

Java代码

```
public class Solution {
    public int RectCover(int target) {
        if(target < 2){
            return target;
        }
        int [] dp = new int[target+1];
        dp[0] = 1;
        dp[1] = 1;
        dp[2] = 2;
```

```
        for(int i = 3; i <= target; i++){
            dp[i] = dp[i-1] + dp[i-2];
        }

        return dp[target];
    }
}
```

9. 核心考点：二进制计算

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/8ee967e43c2c4ec193b040ea7fbb10b8?
tpId=13&tqId=11164&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//方式很多，我们直接用最优化的方案
```

C++代码：

```
class Solution {
public:
     int  NumberOf1(int n) {
         int count = 0;
         while(n){
             n &= (n-1); //可以避免无效检测
             count++;
         }
         return count;
     }
};
```

Java代码：

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

10. 核心考点：数组操作，排序思想的扩展使用 （day1）

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/beb5aa231adc45b2a5dcc5b62c93f593?
tpId=13&tqId=11166&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//这道题原题是不需要保证奇偶位置不变的。
//现在新增了需求，解决方法也比较多，我们用较优方式解决一下，借鉴一下插入排序的思想
```

C++代码

```
class Solution {
public:
    void reOrderArray(vector<int> &array) {
        int k = 0;
        for(int i = 0; i < array.size(); ++i){
            if(array[i] & 1){ //从左向右，每次遇到的，都是最前面的奇数，一定将来要被放在k下标处
                int temp = array[i]; //现将当前奇数保存起来
                int j = i;
                while(j > k){ //将该奇数之前的内容(偶数序列)，整体后移一个位置
                    array[j] = array[j-1];
                    j--;
                }
                array[k++] = temp; //将奇数保存在它将来改在的位置，因为我们是从左往右放的，没有跨越奇
数，所以一定是相对位置不变的
            }
        }
    }
};
```

Java代码　插入排序 将奇数之前的偶数向后移动，然后插入

```
public class Solution {
    public void reOrderArray(int [] array) {
        int k = 0;
        for(int i = 0; i < array.length; i++){
            if((array[i] & 1) == 1){//从左向右，每次遇到的，都是最前面的奇数，一定将来要被放在k下标处
                int temp = array[i];//现将当前奇数保存起来
                int j = i;
                while(j > k){//将该奇数之前的内容(偶数序列)，整体后移一个位置
                    array[j] = array[j-1];
                    j--;
                }
                array[k++] = temp;//将奇数保存在它将来改在的位置，因为我们是从左往右放的，没有跨越奇数，
所以一定是相对位置不变的
            }
        }
    }
}
```

11. 核心考点：链表，前后指针的使用，边界条件检测

解析：输入一个链表，输出该链表中倒数第k个结点。

```
//OJ链接：https://www.nowcoder.com/practice/529d3ae5a407492994ad2a246518148a?tpId=13&tqId=11167&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//较优解题思路：
//1. 题目中的链表是单链表，也就不能从后往前进行
//2. 可以定义两个指针，一个指针先走k步，再让另一个指针跟在后面，使用“前后指针”的方式，当前面的指针到达结尾，后面的指针，也就是倒数第k个节点
```

C++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(pListHead == nullptr){
            return nullptr;
        }

        //我们可以约定首个节点是第1个节点
        ListNode *front = pListHead;//前面的节点
        ListNode *rear  = pListHead;//后面的节点

        while(k > 0 && front){
            k--;
            front = front->next;
        }

        while(front){
            front = front->next;
            rear = rear->next;
        }
        return k > 0 ? nullptr:rear; //走到这里，front一定是nullptr
    }
};
```

Java代码

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
```

```
    }
}*/
public class Solution {
    public ListNode FindKthToTail(ListNode head,int k) {
        if(head == null || k < 0){
            return null;
        }

        ListNode front = head;
        ListNode rear  = head;

        while(k > 0 && front != null){
            k--;
            front = front.next;
        }

        while(front != null){
            front = front.next;
            rear  = rear.next;
        }
        return k > 0 ? null : rear;
    }
}
```

12.核心考点：链表操作，思维缜密程度

输入一个链表，反转链表后，输出新链表的表头

解析：

```
//OJ链接：https://www.nowcoder.com/practice/75e878df47f24fdc9dc3e400ec6058ca?
tpId=13&tqId=11168&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//也有较多解法，1. 定义三个指针，整体右移，边移动，边翻转，保证不会断链。
//2. 可以采用头插思想进行翻转
```

C++代码

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
//方法一
class Solution {
public:
    ListNode* ReverseList(ListNode* pHead) {
        if(pHead == nullptr || pHead->next == nullptr){
            return pHead;
```

```
        }
        //不带头结点，至少有2个节点
        ListNode *first  = pHead;        //指向第一个节点
        ListNode *second = first->next;  //指向第二个节点
        ListNode *third  = second->next; //指向第三个节点，可能为nullptr

        while(third){    //因为third在最前面，所以他一定先到nullptr，只要检测它是否合法即可
            //翻转
            second->next = first;
            //指针整体后移
            first = second;
            second = third;
            third = third->next;
        }
        second->next = first;  //当传入的链表只有两个节点 or 上述翻转结束时，最后一个节点并未翻转
        pHead->next = nullptr; //曾经的第一个节点，next并不是nullptr，设置一下
        pHead = second;        //头指针指向最后一个节点
        return pHead;
    }
};

//方法二，见java代码，比较简单，不在重复写了
```

Java代码

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/
//方法一
public class Solution {
    public ListNode ReverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        //不带头结点，至少有2个节点
        ListNode first  = head; //指向第一个节点
        ListNode second = first.next; //指向第二个节点
        ListNode third  = second.next;//指向第三个节点，可能为null

        while(third != null){
            //翻转
            second.next = first;
            //指针整体后移
            first = second;
            second = third;
            third = third.next;
        }
```

```
        second.next = first;//当传入的链表只有两个节点 or 上述翻转结束时，最后一个节点并未翻转
        head.next = null;//曾经的第一个节点，next并不是null，设置一下
        head = second;    //头指针指向最后一个节点
        return head;
    }
}

//方法二
//另外，这道题还可以采用头插的原则进行翻转，这里附上一份java代码，大家可以参考一下
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/
public class Solution {
    public ListNode ReverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode new_head = null;
        while(head != null){
            //先从原链表中去掉第一个节点
            ListNode p = head;
            head = head.next;

            //再将p标识的节点头查到新链表
            p.next = new_head;
            new_head = p;
        }
        head = new_head;
        return head;
    }
}
```

13. 核心考点：链表合并

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/d8b6b4358f774294a89de2a6ac4d9337?
tpId=13&tqId=11169&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//这道题，解题思路也很多。
//我们可以一个一个节点的归并，当然，也可以采用递归完成
```

C++代码

```
//方法一，比较好理解的
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
    ListNode* Merge(ListNode* pHead1, ListNode* pHead2)
    {
        //不考虑头结点(其实，考虑头结点简单，我们按照难的来)
        //合并前，先判定
        if(nullptr == pHead1){
            return pHead2;
        }
        if(nullptr == pHead2){
            return pHead1;
        }
        //合并中,无非是比较各自首节点谁小，就把该节点从原链表中删除，在尾插到新节点处，比较中，两个链表任何一个都不能为空
        ListNode *new_head = nullptr;
        ListNode *new_tail = nullptr;
        while(pHead1 && pHead2){
            //先判定拿那个节点
            ListNode *p = pHead1->val < pHead2->val ? pHead1 : pHead2;
            //再在指定链表中，删除目标节点
            if(p == pHead1){
                pHead1 = pHead1->next;
            }
            else{
                pHead2 = pHead2->next;
            }
            //尾插到新链表,这里要考虑第一次插入的情况
            if(nullptr == new_head){
                new_head = p;
                new_tail = p;
            }
            else{
                new_tail->next = p;
                new_tail = p;
            }
        }

        //合并后，可能会有：1. pHead1为空 2. pHead2为空 3. 都为空(合并完成)
        if(nullptr == pHead1){
            new_tail->next = pHead2;
        }
        else if(nullptr == pHead2){
            new_tail->next = pHead1;
        }
```

```
        else{
            //do nothing
        }

        return new_head;
    }
};

//方法二，上一道题是Java写的两种方法，我们这里来一个C++的
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
    ListNode* Merge(ListNode* pHead1, ListNode* pHead2)
    {
        //不考虑头结点(其实，考虑头结点简单，我们按照难的来)
        //合并前，先判定
        if(nullptr == pHead1){
            return pHead2;
        }
        if(nullptr == pHead2){
            return pHead1;
        }
        //合并中，找到第一个节点
        ListNode *new_head = nullptr;
        if(pHead1->val < pHead2->val){
            new_head = pHead1;
            pHead1 = pHead1->next;
        }
        else{
            new_head = pHead2;
            pHead2 = pHead2->next;
        }
        //合并剩下的节点
        new_head->next = Merge(pHead1, pHead2);

        return new_head;
    }
};
```

Java代码

```
/*
public class ListNode {
    int val;
    ListNode next = null;
```

```
    ListNode(int val) {
        this.val = val;
    }
}*/
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        //不考虑头结点(其实，考虑头结点简单，我们按照难的来)
        //合并前，先判定
        if(null == list1){
            return list2;
        }
        if(null == list2){
            return list1;
        }
        //合并中,无非是比较各自首节点谁小，就把该节点从原链表中删除，在尾插到新节点处，比较中，两个链表任何一个都不能为空
        ListNode new_head = null;
        ListNode new_tail = null;
        while(list1 != null && list2 != null){
            ListNode p = list1.val < list2.val ? list1 : list2;
            if(p == list1){
                list1 = list1.next;
            }
            else{
                list2 = list2.next;
            }
            if(null == new_head){
                new_head = p;
                new_tail = p;
            }
            else{
                new_tail.next = p;
                new_tail = p;
            }
        }

        //合并后，可能会有：1. pHead1为空 2. pHead2为空 3. 都为空(合并完成)
        if(null == list1){
            new_tail.next = list2;
        }
        else{
            new_tail.next = list1;
        }
        return new_head;
    }
}

//方法二，参加C++代码
```

14. 核心考点：二叉树理解，二叉树遍历

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）

解析：

```
//OJ链接：https://www.nowcoder.com/practice/6e196c44c7004d15b1610b9afca8bd88?
tpId=13&tqId=11170&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//二叉树都是递归定义的，所以递归操作是比较常见的做法
//首先明白：子结构怎么理解，可以理解成子结构是原树的子树(或者一部分)
//也就是说，B要是A的子结构，B的根节点+左子树+右子树，都在A中存在且构成树形结构
//比较的过程要分为两步
//1. 先确定起始位置
//2. 在确定从该位置开始，后续的左右子树的内容是否一致
```

C++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 IsSameFromBegin(TreeNode* begin, TreeNode* beginSub){
        //和下面的寻找代码不同的是，现在要执行的是第二步
        if(nullptr == beginSub){ //beginSub为nullptr，说明已经比较完了
            return true;
        }
        if(nullptr == begin){ //begin为空，说明beginSub不是你的子树
            return false;
        }
        if(begin->val != beginSub->val){
            return false; //说明，整树中，有不相等的节点
        }

        //在比较左右子树是否相等
        //这里大家深度想想递归是怎么返回的【整个递归的结果，由最深层调用的结果决定】
        return IsSameFromBegin(begin->left, beginSub->left) && IsSameFromBegin(begin->right,
beginSub->right);
    }
    //比较的过程要分为两步
    //1. 先确定起始位置
    //2. 在确定从该位置开始，后续的左右子树的内容是否一致
    bool HasSubtree(TreeNode* pRoot1, TreeNode* pRoot2)
    {
        //下面逻辑整体执行的是第一步
        //先判定两棵树有没有为空的情况
        if(pRoot1 == nullptr || pRoot2 == nullptr){
            return false;
        }
```

```
        bool result = false;
        //找到了起始位置
        if(pRoot1->val == pRoot2->val){
            //从该起始位置开始，查找是否一致
            result = IsSameFromBegin(pRoot1, pRoot2);
        }
        //如果pRoot1->val ! = pRoot2->val or 从起始位置开始，不一致
        if(!result){
            //在A树的左子树使用相同的方式找找
            result = HasSubtree(pRoot1->left, pRoot2);
        }
        //左子树没有，再在右子树找找
        if(!result){
            result = HasSubtree(pRoot1->right, pRoot2);
        }

        return result;
    }
};
```

Java代码

```
/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }
}
*/
public class Solution {
    public boolean IsSameFromBegin(TreeNode begin,TreeNode beginSub){
        if(beginSub == null){//beginSub为null，说明已经比较完了
            return true;
        }
        if(begin == null){ //begin为空，说明beginSub不是你的子树
            return false;
        }
        if(begin.val != beginSub.val){
            return false;//说明，整树中，有不相等的节点
        }
        //分别比较左右左右子树，必须都是相等的
        //这里大家深度想想递归是怎么返回的【整个递归的结果，由最深层调用的结果决定】
        return IsSameFromBegin(begin.left, beginSub.left) && IsSameFromBegin(begin.right,
beginSub.right);
    }
    public boolean HasSubtree(TreeNode root1,TreeNode root2) {
        //第一步，先确定起始比较位置
        if(root1 == null || root2 == null){
```

```
            return false;
        }
        boolean result = false;
        if(root1.val == root2.val){
            //确立起始比较位置,从该位置尝试比较
            result = IsSameFromBegin(root1, root2);
        }
        //说明root1.val != root2.val or 上面的起始位置不满足需求，换一个起始位置
        if(result != true){
            result = HasSubtree(root1.left, root2); //在左子树中找找
        }
        //同上
        if(result != true){
            result = HasSubtree(root1.right, root2); //在左子树中找找
        }

        return result;
    }
}
```

15. 核心考点：二叉树操作

操作给定的二叉树，将其变换为源二叉树的镜像。

输入描述:

```
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
```

解析：

```
//OJ链接：https://www.nowcoder.com/practice/564f4c26aa584921bc75623e48ca3011?
tpId=13&tqId=11171&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//仔细观察可以发现，所谓的二叉树镜像本质是自顶向下(or自底向上)进行左右子树交换的过程
//有了这个idea，我们会发现，代码很好写
```

C++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 == nullptr){
            return;
        }
        TreeNode *temp = pRoot->left;
        pRoot->left = pRoot->right;
        pRoot->right = temp;
        Mirror(pRoot->left);
        Mirror(pRoot->right);
        //TreeNode *temp = pRoot->left;
        //pRoot->left = pRoot->right;
        //pRoot->right = temp;
    }
};
```

Java代码

```
/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }
}
*/
public class Solution {
    public void Mirror(TreeNode root) {
        if(root == null){
            return;
        }

        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;
        Mirror(root.left);
        Mirror(root.right);
    }
}
```

16. 核心考点：链表操作，临界条件检查，特殊情况处理

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

解析：

```
//OJ链接：https://www.nowcoder.com/practice/fc533c45b73a41b0b44ccba763f866ef?tpId=13&tqId=11209&tPage=3&rp=1&ru=%2Fta%2Fcoding-interviews&qru=%2Fta%2Fcoding-interviews%2Fquestion-ranking
//解题思路：
//通过快慢指针的方式限定范围，进而达到去重的效果
//这里要考虑特别多的特殊情况，如：全部相同，全部不相同，部分相同等，为了方便解题我们定义头结点，主要是应对全部相同的情况
```

C++代码

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
        val(x), next(NULL) {
    }
};
*/
class Solution {
public:
    ListNode* deleteDuplication(ListNode* pHead)
    {
        if(nullptr == pHead){
            return pHead;
        }
        //考虑到与可能全部相同，带个头结点更简单
        ListNode *head = new ListNode(0);
        head->next = pHead;

        //prev 永远 在 last的前面
        ListNode *prev = head;
        ListNode *last = prev->next;
        while(last != nullptr){
            //1. 如果last和last->next不相等，就一直让prev和last往后走，先找到重复的开始
            while(last->next != nullptr && last->val != last->next->val){
                prev = prev->next;
                last = last->next;
            }
            //2. 如果如果last和last->next相等，就让last一直往后走，在找到重复的范围
            while(last->next != nullptr && last->val == last->next->val){
                last = last->next;
            }
            //走到这里结果一共有三种,注意：prev永远指向的是前驱有效节点：
            //1. last->next != nullptr  并且 (prev, last] 限定了一段重复范围，此时进行去重
```

```
            //2. last->next == nullptr && (prev, last] 限定了一段重复范围，此时进行去重，最后相当于
prev->next = nullptr
            //3. last->next == nullptr && prev->next == last,这说明，从本次循环开始，大家都不相同，
就不需要进行去重，这个是特殊情况
            if(prev->next != last){
                prev->next = last->next;
            }
            last = last->next;
        }
        return head->next;
    }
};
```

Java代码

```
/*
 public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
*/
public class Solution {
    public ListNode deleteDuplication(ListNode pHead){
        if(pHead == null){
            return pHead;
        }

        ListNode head = new ListNode(0);
        head.next = pHead;

        ListNode prev = head;
        ListNode last = prev.next;
        while(last != null){//last永远在前面
            //先找到重复的开始
            while(last.next != null && last.val != last.next.val){
                prev = prev.next;
                last = last.next;
            }
            //在找到重复的范围
            while(last.next != null && last.val == last.next.val){
                last = last.next;
            }
            //走到这里结果一共有三种,注意：prev永远指向的是前驱有效起始节点：
            //1. last.next != null  并且 (prev, last] 限定了一段重复范围，此时进行去重
            //2. last.next == null && (prev, last] 限定了一段重复范围，此时进行去重，最后相当于prev-
>next = nullptr
            //3. last.next == null && prev.next == last,这说明，从本次循环开始，大家都不相同，就不需
要进行去重，这个是特殊情况
            if(prev.next != last){
```

```
                //说明是一段范围，可以去重
                prev.next = last.next;
            }
            last = last.next; //走这一步，就是为了保证恢复的和最开始一致
        }
        return head.next;
    }
}
```

17. 核心考点：栈的规则性设计

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/4c776177d2c04c2494f2555c9fcc1e49?tpId=13&tqId=11173&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//很容易想到，在栈内部保存min变量，每次更新的时候，都对min变量进行更新。
//但是，面试官很容易就会问到：如果想拿出第二小，第三小的值怎么拿？
//用上面的办法就不行了
//为了满足通用，我们使用一个辅助栈，内部保存元素的个数和数据栈完全一样，不过，辅助栈内部永远保存本次入栈的数为所有数据的最小值（注意：辅助栈内部元素可能会出现“必要性”重复）
//我们这里是为了实现算法， 所以就不从0开始实现stack了
//题面说了，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，所以，后面的代码对空的检验可有可无
```

C++代码

```
class Solution {
    private:
        stack<int> data_stack; //数据栈
        stack<int> min_stack;  //辅助栈
    public:
    void push(int value) {
        data_stack.push(value);
        if(min_stack.size() == 0 || value < min_stack.top()){
            min_stack.push(value);
        }
        else{ //min_stack.size() != 0 && value >= min_stack.top()
            min_stack.push(min_stack.top());
        }
    }
    void pop() {
        if(data_stack.size() == 0 || min_stack.size() == 0){
            return;
        }
        data_stack.pop();
        min_stack.pop();
    }
    int top() {
```

```
        return data_stack.top();
    }
    int min() {
        return min_stack.top();
    }
};
```

Java代码

```
import java.util.Stack;

public class Solution {
    private Stack<Integer> data_stack = new Stack<>();
    private Stack<Integer> min_stack = new Stack<>();

    public void push(int node) {
        data_stack.push(node);
        if(min_stack.empty() || node < min_stack.peek()){
            min_stack.push(node);
        }
        else{
            //!min_stack.empty() & node  >=min_stack.peek()
            min_stack.push(min_stack.peek());
        }
    }

    public void pop() {
        data_stack.pop();
        min_stack.pop();
    }

    public int top() {
        return data_stack.peek();
    }

    public int min() {
        return min_stack.peek();
    }
}
```

18. 核心考点：栈的理解

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）

解析：

```
//OJ链接：https://www.nowcoder.com/practice/d77d11405cc7470d82554cb392585106?
tpId=13&tqId=11174&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
    //要判定第二个序列是否可能是该栈的弹出序列，就要使用指定的入栈顺序
    //模拟出来对应的弹栈序列，我们设入栈顺序序列式pushV， 可能出栈序列popV
    //popv的第一个元素，一定是最后入栈，最先弹栈的,而我们的入栈顺序是一定的
    //也就决定了，我们必须一直入栈，直到碰到popv的第一个元素，然后开始弹栈
    //最后在循环这个过程，如果符合要求，最后栈结构一定是空的
```

C++代码

```
class Solution {
public:
    bool IsPopOrder(vector<int> pushV,vector<int> popV) {
        if(pushV.size() == 0 || popV.size() == 0 || pushV.size() != popV.size()){
            return false;
        }
        stack<int> st; //辅助栈，帮我们进行入栈出栈模拟
        int i = 0;
        int j = 0;
        for(; i < pushV.size(); i++){ //遍历入栈序列，开始入栈
            st.push(pushV[i]);
            //如果该条件不满足，就要一直入栈
            //如果该条件满足，就要一直出栈
            //pushV代表对应的入栈逻辑，popV代表对应的出栈逻辑
            //两个元素相等，代表入栈逻辑结束，出栈逻辑开始，想想为什么?
            while(!st.empty() && st.top() == popV[j]){
                st.pop(); //去掉栈顶，在比较下一个
                j++;
            }
        }
        return st.empty();
    }
};
```

Java代码

```
import java.util.ArrayList;
import java.util.Stack;

public class Solution {
    public boolean IsPopOrder(int [] pushA,int [] popA) {
        if(pushA == null || popA == null || pushA.length == 0 || popA.length == 0 ||
pushA.length != popA.length){
            return false;
        }
        Stack<Integer> st = new Stack<>();
        int i = 0;
        int j = 0;
        for( ; i < pushA.length; i++){
            //如果该条件不满足，就要一直入栈
            //如果该条件满足，就要一直出栈
```

```
            //pushV代表对应的入栈逻辑，popV代表对应的出栈逻辑
            //两个元素相等，代表入栈逻辑结束，出栈逻辑开始，想想为什么?
            st.push(pushA[i]);
            while(!st.empty() && st.peek() == popA[j]){//入栈逻辑结束，开始出栈逻辑
                st.pop();
                j++;
            }
        }
        return st.empty();
    }
}
```

19. 核心考点：二叉树层序遍历

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

解析：

```
//OJ链接：https://www.nowcoder.com/practice/7fe2212963db4790b57431d9ed259701?
tpId=13&tqId=11175&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//本质是层序遍历二叉树，借助queue即可完成
```

C++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) {
        if(root == nullptr){
            return vector<int>();
        }
        vector<int> v;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            TreeNode *father = q.front();
            q.pop();
            v.push_back(father->val);
            if(father->left){
                //左子树不为空
                q.push(father->left);
            }
            if(father->right){
```

```
                //右子树不为空
                q.push(father->right);
            }
        }

        return v;
    }
};
```

Java代码

```
import java.util.Queue;
import java.util.LinkedList;
import java.util.ArrayList;

/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;
    }

}
*/
public class Solution {
    public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
        if(root == null){
            return new ArrayList<Integer>();
        }
        Queue<TreeNode> q = new LinkedList<>();
        ArrayList<Integer> result = new ArrayList<>();

        q.offer(root);
        while(!q.isEmpty()){
            TreeNode father = q.poll();
            result.add(father.val);
            if(father.left != null){
                q.offer(father.left);
            }
            if(father.right != null){
                q.offer(father.right);
            }
        }
        return result;
    }
}
```


20. 核心考点：BST特征的理解

输入一个非空整数数组，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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/a861533d45854474ac791d90e447bafd?tpId=13&tqId=11176&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//看清楚，是二叉搜索树 后序遍历
//二叉搜索树:它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
//后序遍历：先左后右再根
//BST的后序序列的合法序列是，对于一个序列S，最后一个元素是x （也就是root节点），如果去掉最后一个元素的序列为T，那么T满足：T可以分成两段，前一段（左子树）小于x，后一段（右子树）大于x，且这两段（子树）都是合法的后序序列
//验证思路就是：当前序列，及其子序列必须都满足上述定义
```

C++代码

```
class Solution {
public:
    bool VerifySquenceOfBSTCore(vector<int> &sequence, int start, int end){
        if(start >= end){
            //在不断查找过程中，区域不断缩小，为空时，证明之前的所有范围都满足检测条件
            //也就是是一个BST
            return true;
        }
        //拿到root节点的值
        int root = sequence[end];
        //先遍历左半部分，也就是整体都要比root小，拿到左子树序列
        int i = 0;
        while(i < end && sequence[i] < root){
            i++;
        }
        //在检测右子树是否符合大于root的条件，要从i开始，也就是右半部分的开始
        for(int j = i; j < end; j++){
            if(sequence[j] < root){
                //在合法位置处，当前值小于root，不满足BST定义
                return false;
            }
        }
        //走到这里，就说明，当前序列满足需求。但并不代表题目被解决了，还要在检测left和right各自是否也满足
        return VerifySquenceOfBSTCore(sequence, 0, i-1) &&  VerifySquenceOfBSTCore(sequence, i, end-1);
    }
    bool VerifySquenceOfBST(vector<int> sequence) {
        if(sequence.empty()){
            return false;
        }

        return VerifySquenceOfBSTCore(sequence, 0, sequence.size()-1);
    }
};
```

Java代码

```
public class Solution {
    public boolean VerifySquenceOfBSTCore(int [] sequence, int start, int end){
        if(start >= end){
            //在不断查找过程中，区域不断缩小，为空时，证明之前的所有范围都满足检测条件
            //也就是是一个BST
            return true;
        }
        //拿到root节点的值
        int root = sequence[end];
        //先遍历左半部分，也就是整体都要比root小，拿到左子树序列
        int i = 0;
        while(i < end && sequence[i] < root){
            i++;
        }
        //在检测右子树是否符合大于root的条件,要从i开始，也就是右半部分的开始
        for(int j = i; j < end; j++){
            if(sequence[j] < root){
                return false;
            }
        }
         //走到这里，就说明，当前序列满足需求。但并不代表题目被解决了，还要在检测left和right各自是否也满足
        return VerifySquenceOfBSTCore(sequence, 0, i-1) && VerifySquenceOfBSTCore(sequence,
i, end-1);
    }
    public boolean VerifySquenceOfBST(int [] sequence) {
        if(sequence.length == 0){
            return false;
        }

        return VerifySquenceOfBSTCore(sequence, 0, sequence.length-1);
    }
}
```

21. 核心考点：简单回溯法的使用

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/b736e784e3e34731af99065031301bca?
tpId=13&tqId=11177&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//这是一个典型的DFS回溯的算法
//回溯法本质是一个基于DFS的穷举的过程
//1. 先添加值
//2. 在判定现有结果是否满足条件
//3. DFS
//4. 回退
```

C++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 FindPathDFS(TreeNode* root, int expectNumber, vector<vector<int> > &result,
vector<int> &list){
        if(root == nullptr){
            return;
        }
        //先添加值
        list.push_back(root->val);
        expectNumber -= root->val;
        //检测条件是否满足题面需求
        if(root->left == nullptr && root->right == nullptr && expectNumber == 0){
            result.push_back(list);
        }
        //dfs进去
        FindPathDFS(root->left, expectNumber, result, list);
        FindPathDFS(root->right, expectNumber, result, list);

        list.pop_back(); //回退
    }
    vector<vector<int> > FindPath(TreeNode* root,int expectNumber) {
        vector<vector<int> > result;
        vector<int> list;
        FindPathDFS(root, expectNumber, result, list);
        return result;
    }
};
```

Java代码

```
import java.util.ArrayList;
/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }
```

```
}
*/
public class Solution {
    //基于DFS的回溯算法
    public void FindPathDFS(TreeNode root,int target, ArrayList<ArrayList<Integer>>
result,ArrayList<Integer> list){
        if(root == null){
            return;
        }
        //将当前值放入list待选结果集中
        list.add(root.val);
        //更新目标值
        target -= root.val;
        //1. 已经是叶子节点了
        //2. 从root到该叶子节点，之和是target
        //3. 是叶子节点，但是不满足节点，也不影响，程序会直接退出
        if(root.left == null && root.right == null && target == 0){ //回溯剪枝，去掉不满足条件的
结果
            result.add(new ArrayList<Integer>(list)); //注意深浅拷贝
        }

        FindPathDFS(root.left, target, result, list); //DFS递归统计
        FindPathDFS(root.right, target, result, list);

        list.remove(list.size()-1);
    }
    public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {
        ArrayList<ArrayList<Integer>> result = new  ArrayList<ArrayList<Integer>>();
        ArrayList<Integer> list = new ArrayList<>();
        FindPathDFS(root, target, result, list);
        return result;
    }
}
```

22. 核心考点：全排列问题，DFS

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`。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/fe6b651b66ae47d7acce78ffdd9a96c7?
tpId=13&tqId=11180&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//全排列问题，可以看做如下多叉树形态
//                 _________________ 开始 _________________
//               /                   |                   \
//         ____ a ____             ___ b ___             ___ c ___           ->    第一阶段
//        /           \           /         \           /         \
//       b             c         a           c         a           b        ->    第二阶段
//       |             |         |           |         |           |
//       c             b         c           a         b           a        ->    第三阶段
```

```
//
// 很明显，我们想要得到合适的排列组合，一定是深度优先的
//该问题可以把目标串理解成两部分：第一部分：以哪个字符开头，第二部分：剩下的是子问题
//所以，我们要让每个字符都要做一遍开头，然后在求解子问题
```

C++代码

```
class Solution {
public:
    void swap(string &str, int i, int j){
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
    bool IsExist(vector<string>& result, string &str){
        auto it = result.begin();
        for (; it != result.end(); ++it){
            if (*it == str){
                return true;
            }
        }
        return false;
    }
    void PermutationHelper(string &str, int start, vector<string>& result){
        if (start == str.length() - 1){
            if (!IsExist(result, str)){
                result.push_back(str);
            }
            return;
        }
        for (int i = start; i < (int)str.size(); i++){
            //start 和 i 的关系是：表示以谁开始
            swap(str, start, i);
            //当确定以哪个字符作为开始，就要在决定另一部分的排列组合种类
            //这里一定要深刻理解，i仅仅是决定以谁作为排列的开始，但是求sub字符串每次开始，都要从start+1开
始
            PermutationHelper(str, start + 1, result);
            swap(str, start, i);
        }
    }
    vector<string> Permutation(string str) {
        vector<string> result;
        if(str.length() > 0){
            PermutationHelper(str, 0, result);
            sort(result.begin(), result.end());
        }
        return result;
    }
};
```

Java代码

```
import java.util.ArrayList;
import java.util.Collections;

public class Solution {
    public void Swap(char[] str, int i, int j){
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
    public boolean IsExist(ArrayList<String> result, char[] str){
        return result.contains(String.valueOf(str));
    }
    public void PermutationHelper(char[] str, int start, ArrayList<String> result){
        if(start == str.length - 1){
            if(!IsExist(result, str)){
                result.add(new String(str));
            }
            return;
        }
        for(int i = start; i < str.length; i++){
            Swap(str, start, i);
            PermutationHelper(str, start+1, result);
            Swap(str, start, i);
        }
    }
    public ArrayList<String> Permutation(String str) {
        ArrayList<String> result = new ArrayList<>();
        if(str != null && str.length() > 0){
            PermutationHelper(str.toCharArray(), 0, result);
            Collections.sort(result);
        }
        return result;
    }
}
```

23. 核心考点: 数组使用，简单算法的设计 （day1）

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/e8a1b01a2df14cb2b228b30ee6a92163?tpId=13&tqId=11181&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//这道题整体思路比较明确
//思路一：定义map，使用<数字，次数>的映射关系，最后统计每个字符出现的次数
//思路二：排序，出现次数最多的数字，一定在中间位置。然后检测中间出现的数字出现的次数是否符合要求
//思路三：目标条件：目标数据超过数组长度的一半，那么对数组，我们同时去掉两个不同的数字，到最后剩下的一个数就是该数字。如果剩下两个，那么这两个也是一样的，就是结果），在其基础上把最后剩下的一个数字或者两个回到原来数组中，将数组遍历一遍统计一下数字出现次数进行最终判断。
```

C++代码

```
//思路一
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        unordered_map<int, int> map;
        int half = numbers.size()/2;
        for(int i = 0; i < numbers.size(); i++){
            auto it = map.find(numbers[i]);
            //如果已经在map中，进行自增，如果不在，插入，首次出现
            if( it != map.end() ){
                map[numbers[i]]++;
            }
            else{
                map.insert(make_pair(numbers[i], 1));
            }
            //自增或者插入一个，直接进行判定。注意，这里要考虑测试用例为{1}的情况
            //走到这里，对应的key val一定存在
            if(map[numbers[i]] > half){
                return numbers[i];
            }
        }
        //走到这里，说明没有找到
        return 0;
    }
};
//思路二
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        sort(numbers.begin(), numbers.end());
        int target = numbers[numbers.size()/2];
        int count = 0;
        for(int i = 0; i < numbers.size(); i++){
            if(target == numbers[i]){
                count++;
            }
        }
        if(count > numbers.size()/2){
            return target;
        }
        return 0;
    }
};
//思路三
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        if(numbers.size() == 0){
            return 0;
        }
        //重点写一下
```

```
        //可以采取不同的数量进行抵消的思路
        int number = numbers[0];
        int times  = 1;
        for(int i = 1; i < numbers.size(); i++){
            if(times == 0){ //如果当前times是0，说明之前的不同抵消完了
                number = numbers[i];
                times = 1;
            }
            else if(numbers[i] == number){
                times++;
            }
            else{
                times--;
            }
        }
        //如果输入本身满足条件，则times一定>0，并且number保存的就是准目标，但是还需要确认
        int count = 0;
        for(int i = 0; i < numbers.size(); i++){
            if(numbers[i] == number){
                count++;
            }
        }

        return count > numbers.size()/2 ? number : 0;
    }
};
```

Java代码

```
//思路一
import java.util.Map;
import java.util.HashMap;
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        if(array == null){
            return 0;
        }
        Map<Integer, Integer> map = new HashMap<>();
        for(int i = 0; i < array.length; i++){
            if(map.containsKey(array[i])){
                int count = map.get(array[i]);
                count++;
                map.put(array[i], count);
            }else{
                map.put(array[i], 1);
            }
            if(map.get(array[i]) > array.length/2){
                return array[i];
            }
        }
        return 0;
    }
}
```

```
//思路二
import java.util.Arrays;
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        if(array == null){
            return 0;
        }
        Arrays.sort(array);
        int target = array[array.length/2];
        int count = 0;
        for(int i = 0; i < array.length; i++){
            if(target == array[i]){
                count++;
            }
        }
        if(count > array.length/2){
            return target;
        }
        return 0;
    }
}
//思路三
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        if(array == null){
            return 0;
        }
        int target = array[0];
        int times = 1;

        for(int i = 1; i < array.length; i++){
            if(times == 0){
                target = array[i];
                times = 1;
            }
            else if(array[i] == target){
                times++;
            }
            else{
                times--;
            }
        }
        times  = 0;
        for(int i = 0; i < array.length; i++){
            if(target == array[i]){
                times++;
            }
        }

        return times > array.length/2 ? target : 0;
    }
}
```

24. 核心考点：topK问题

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,。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf?tpId=13&tqId=11182&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//top k问题
//解题思路：
//思路一：直接升序排序，取前n个，这个方法不考虑了
//思路二：可以采用最小堆，我们这里使用C+++ priority_queue or java PriorityQueue优先级队列进行处理（底层原理类似堆）.这里核心思路在于实现topk，我们使用现成的解决方案。如果需要了解堆实现，可以自行了解一下
```

C++代码

```
struct comp{
    bool operator()(const int &a, const int &b){
        return a < b; //我们需要最大堆，所以我们采用降序排序
    }
};

class Solution {
public:
    vector<int> GetLeastNumbers_Solution(vector<int> input, int k) {
        vector<int> list;
        if(input.size() == 0 || k <= 0 || k > input.size()){
            return list;
        }

        priority_queue<int, vector<int>, comp> queue; //采用指定容器实现最大堆
        for(int i = 0; i < input.size(); i++){
            if( i < k){
                //前k个元素，直接放入，priority_queue内部会降序排序
                queue.push(input[i]);
            }else{
                if(input[i] < queue.top()){
                    //如果新的数据，小于queue首部元素（最大值），进行更新
                    queue.pop();
                    queue.push(input[i]);
                }
            }
        }
        for(int i = 0; i < k; i++){
            list.push_back(queue.top());
            queue.pop();
        }
        return list;
    }
};
```

Java代码

```
import java.util.ArrayList;
import java.util.PriorityQueue;
import java.util.Collections;

public class Solution {
    public ArrayList<Integer> GetLeastNumbers_Solution(int [] input, int k) {
        ArrayList<Integer> list = new ArrayList<>();
        if(input == null || k <= 0 || k > input.length){
            return list;
        }
        //按照数值从大到小
        PriorityQueue<Integer> queue = new PriorityQueue<>(k, Collections.reverseOrder());
        for(int i = 0; i < input.length ; i++){
            if(i < k){
                queue.offer(input[i]);//前提前插入k个数据，queue会自动排序
            }else{
                //否则，就要展开淘汰的过程了，每次都淘汰最大的，剩下的最终就是最小的k个
                if(input[i] < queue.peek()){ //input[i]比最大的小，跟新之
                    queue.poll();
                    queue.offer(input[i]);
                }
            }
        }
        //返回对应的结果
        for(int i = 0 ; i < k; i++){
            list.add(queue.poll());
        }
        return list;
    }
}
```

25. 核心考点：简单动归问题

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)。给一个数组，返回它的最大连续子序列的和，你会不会被他忽悠住？(子向量的长度至少是1)

解析：

```
//OJ链接：https://www.nowcoder.com/practice/459bd355da1549fa8a49e350bf3df484?tpId=13&tqId=11183&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//方法一：我们可以使用dp完成
//定义状态# f(i)：以i下标结尾的最大连续子序列的和
//状态递推：f(i) = max(f(i-1)+array[i], array[i]) 【这里一定要注意连续关键字】
//状态初始化：f(0) = array[0], max = array[0]
//方法二：可以进行一定程度优化，具体写完基本版本，在考虑
```

C++代码

```
//方法一
class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array) {
        //经典且高频dp问题
        //定义状态# f(i): 以i下标结尾的最大连续子序列的和
        //状态递推: f(i) = max(f(i-1)+array[i], array[i]) 【这里一定要注意连续关键字】
        //状态初始化: f(0) = array[0], max = array[0]
        int max_value = array[0];
        int *dp = new int[array.size()];
        dp[0] = array[0];

        for(int i = 1; i < array.size(); i++){
            dp[i] = max(dp[i-1] + array[i], array[i]);
            if(max_value < dp[i]){
                max_value = dp[i];
            }
        }
        delete dp;
        return max_value;
    }
};

//方法二
//很明显，上面的代码，只会使用dp[i] 和 dp[i-1]，所以是有优化的可能的
class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array) {
        //经典且高频dp问题
        //定义状态# f(i): 以i下标结尾的最大连续子序列的和
        int max_value = array[0];
        int total = array[0]; //当前累计的和

        //for 循环，用来检测以i下标结尾的连续子序列的和
        for(int i = 1; i < array.size(); i++){
            if(total >= 0){
                //如果之前total累计的和>=0,说明当前数据+total，有利于整体增大
                total += array[i];
            }
            else{
                //如果之前累计的和<0,说明当前数据+total，不利于整体增大,丢弃之前的所有值
                //这里有一个基本事实，就是之前的连续数据和是确定的。
                //连续，是可以从以前到现在，也可以是从现在到以后。至于要不要加以前，就看以前对整体增大又没有
贡献度
                total = array[i];
            }
            //走到这，标示以i下标结尾的最大连续子序列的和已经算出，进行最大值统计
            if(max_value < total){
                max_value = total;
            }
        }
        return max_value;
```

```
    }
};
```

Java代码

```
//方法一
public class Solution {
    public int FindGreatestSumOfSubArray(int[] array) {
        //定义状态# f(i): 以i下标结尾的最大连续子序列的和
        //状态递推: f(i) = max(f(i-1)+array[i], array[i]) 【这里一定要注意连续关键字】
        //状态初始化: f(0) = array[0], max = array[0]
        //以[-2,-3,4,-1,-2,1,5,-3]为例,可以发现,
        //dp[0] = -2
        //dp[1] = -3
        //dp[2] = 4
        //dp[3] = 3
        //以此类推,会发现: dp[i] = max{dp[i-1]+array[i],array[i]}.

        int[] dp = new int[array.length];
        dp[0] = array[0];
        int max_value = array[0];
        for(int i = 1; i < array.length; i++){
            dp[i] = Math.max(dp[i-1] + array[i], array[i]);
            if(max_value < dp[i]){
                max_value = dp[i];
            }
        }
        return max_value;
    }
}
//方法二
public class Solution {
    public int FindGreatestSumOfSubArray(int[] array) {
        int total = array[0];
        int max_value = array[0];

        //for 循环，用来检测以i下标结尾的连续子序列的和
        for(int i = 1; i < array.length; i++){
            if(total >= 0){
                //如果之前total累计的和>=0,说明当前数据+total，有利于整体增大
                total += array[i];
            }else{
                //如果之前累计的和<0,说明当前数据+total，不利于整体增大,丢弃之前的所有值
                //这里有一个基本事实，就是之前的连续数据和是确定的。
                //连续，是可以从以前到现在，也可以是从现在到以后。至于要不要加以前，就看以前对整体增大又没有
贡献度
                total = array[i];
            }

            if(max_value < total){
                max_value = total;
            }
        }
```

```
        return max_value;
    }
}
```

26. 核心考点：hash思想

给定一个仅由小写字母组成的字符串。现在请找出一个位置，删掉那个字母之后，字符串变成回文。请放心总会有一个合法的解。如果给定的字符串已经是一个回文串，那么输出-1。

输入描述:

```
第一行包含T，测试数据的组数。后面跟有T行，每行包含一个字符串。
```

输出描述:

```
如果可以删去一个字母使它变成回文串，则输出任意一个满足条件的删去字母的位置（下标从0开始）。例如：

bcc

我们可以删掉位置0的b字符。
```

示例1

输入

复制

```
3
aaab
baa
aaa
```

输出

```
3
0
-1
```

解析：

```
//OJ链接：https://www.nowcoder.com/practice/b6edb5ca15d34b1eb42e4725a3c68eba?tpId=179&&tqId=34268&rp=1&ru=/activity/oj&qru=/ta/exam-other/question-ranking
//解题思路：
//可以从两侧进行统计，如果不同，则删除任意一个，在判定是否是回文，如果是，下标就是删除数据的下标，如果不是，就是另一个元素的下标
```

C++代码

```
#include <iostream>
#include <string>

using namespace std;

bool IsPalindrome(string &s, int *start, int *end){
    int i = 0;
    int j = s.size() - 1;
    bool result = true;
    while(i <= j){
        if(s[i] != s[j]){
            result = false;
            break;
        }
        i++,j--;
    }
    if(start != nullptr) *start = i;
    if(end != nullptr) *end = j;
    return result;
}

int main()
{
    int num = 0;
    cin >> num;
    while(num){
        string s;
        cin >> s;
        int start = 0;
        int end = s.size() - 1;
        if(IsPalindrome(s, &start, &end)){
            cout << -1 << endl; //已经是回文了
        }else{
            s.erase(end, 1);
            if(IsPalindrome(s, nullptr, nullptr)){
                cout << end << endl;
            }else{
                cout << start << endl;
            }
        }
        num--;
    }
}
```

Java代码

```
import java.util.Scanner;
public class Main{
    public static boolean IsPalindrome(StringBuffer sb, int[] start, int[] end){
        int i = 0;
        int j = sb.length() - 1;
        boolean result = true;
```

```
        while(i <= j){
            if(sb.charAt(i) != sb.charAt(j)){ //不是回文
                result = false;
                break;
            }
            i++;
            j--;
        }
        if(start != null) start[0] = i;
        if(end != null) end[0] = j;
        return result;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);

        int num = sc.nextInt();
        while(num > 0){
            StringBuffer sb = new StringBuffer(sc.next());

            //char[] arr = s.toCharArray();
            int[] start = new int[1];
            int[] end = new int[1];

            if(IsPalindrome(sb, start, end)){ //判定是否已经是回文了
                //说明本来就是回文
                System.out.println(-1);
            }
            else{
                sb.deleteCharAt(end[0]);
                if(IsPalindrome(sb, null, null)){
                    System.out.println(end[0]);
                }else{
                    System.out.println(start[0]);
                }
            }
            num--;
        }
    }
}
```

27. 核心考点：排序算法的特殊理解

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/8fecd3f8ba334add803bf2a06af1b993?tpId=13&tqId=11185&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//这道题很有意思，核心理解是我们对于排序算法的理解，通常我们所理解的排序是比较大小的
//如：升序排序的序列意思是：序列中任何一个数字，都比前面的小，比后面的大
//我们把说法换一下，对于本题，我们要的有效序列是：序列中任何一个元素y，和它前的任何一个元素x进行有序组合形成xy，比和他后面的任何一个元素z进行有效序列组合yz，满足条件xy < yz（采用字典序列排序）
//如{32，31}，有效组合是3132，所以我们拍完序列之后序列变成{31，32}
```

C++代码

```
class Solution {
public:
    static bool cmp(int x, int y){
        //要保证找到x，y构成的序列中，让小的放在前面
        string xs = to_string(x);
        string ys = to_string(y);
        string A  = xs;
        A += ys;
        string B  = ys;
        B += xs;
        return A < B;
    }
    string PrintMinNumber(vector<int> numbers) {
        sort(numbers.begin(), numbers.end(), cmp);
        string result = "";
        for(unsigned int i = 0; i < numbers.size(); i++){
            result += to_string(numbers[i]);
        }

        return result;
    }
};
```

Java代码

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Solution {
    public String PrintMinNumber(int [] numbers) {
        if(numbers == null){
            return new String();
        }
        ArrayList<Integer> list = new ArrayList<Integer>();
        for(int e : numbers){
            list.add(e);
        }
        Collections.sort(list, new Comparator<Integer>(){
            @Override
            public int compare(Integer x, Integer y){
```

```
                String xs = x + "" + y;
                String ys = y + "" + x;
                return xs.compareTo(ys);
            }
        });
        String result = new String();
        for(Integer e : list){
            result += e;
        }
        return result;
    }
}
```

28. 核心考点：单链表理解，临界条件判定

输入两个链表，找出它们的第一个公共结点。（注意因为传入数据是链表，所以错误测试数据的提示是用其他方式显示的，保证传入数据是正确的）

解析：

```
//OJ链接：https://www.nowcoder.com/practice/6ab1d9a29e88450685099d45c9e31e46?tpId=13&tqId=11189&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//题目要求是单链表，所以如果有交点，则最后一个链表的节点地址一定是相同的
//求第一公共节点，本质是让长的链表先走abs（length1-length2）步，后面大家的步调一致，往后找第一个地址相同的节点，就是题目要求的节点
//所以需要各自遍历两次链表
```

C++代码

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
    ListNode *GetListLength(ListNode *list, int &len){//len 是一个输出型参数
        if(list == nullptr){
            return list;
        }
        ListNode *end = list;
        while(list){
            end = list;
            list = list->next;
            len++;
        }

        return end;
```

```
    }
    //根据上面的题面，可以看出，是单链表
    ListNode* FindFirstCommonNode( ListNode* pHead1, ListNode* pHead2) {
        if(pHead1 == nullptr || pHead2 == nullptr){
            return nullptr;
        }
        int length1 = 0;
        int length2 = 0;
        ListNode *end1 = GetListLength(pHead1, length1);
        ListNode *end2 = GetListLength(pHead2, length2);
        if(end1 != end2){
            //说明两张链表根本就没有公共节点，其实这里不用判定也行，但是提前判定，后续就不用在忙活了
            //当然，你如果不想这样写，也可以，参加后面的java代码
            return nullptr;
        }
        int step = abs(length1 - length2);
        if(length1 > length2){
            while(step){
                pHead1 = pHead1->next;
                step--;
            }
        }
        else{
            while(step){
                pHead2 = pHead2->next;
                step--;
            }
        }
        while(pHead1 && pHead2){
            if(pHead1 == pHead2){
                return pHead1;
            }
            pHead1 = pHead1->next;
            pHead2 = pHead2->next;
        }
        return nullptr;
    }
};
```

Java代码

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/

public class Solution {
    public int GetListLength(ListNode list){
```

```
        if(list == null){
            return 0;
        }
        int len = 0;
        while(list != null){
            len++;
            list = list.next;
        }
        return len;
    }
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
        if(pHead1 == null || pHead2 == null){
            return null;
        }
        //java代码和C++代码略有不同，这里我们简单说明
        int length1 = GetListLength(pHead1);
        int length2 = GetListLength(pHead2);

        int step = Math.abs(length1 - length2);
        if(length1 > length2){
            while(step > 0){
                pHead1 = pHead1.next;
                step--;
            }
        }
        else{
            while(step > 0){
                pHead2 = pHead2.next;
                step--;
            }
        }
        while(pHead1 != null && pHead2 != null){
            if(pHead1 == pHead2){
                return pHead1;
            }
            pHead1 = pHead1.next;
            pHead2 = pHead2.next;
        }
        return null;
    }
}
```

29. 核心考点：二叉树深度的判定方法

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/435fb86331474282a3499955f0a41e8b?tpId=13&tqId=11191&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//1. 可以使用递归方式
//2. 可以层序遍历，统计层数，也就是深度or高度
```

C++代码

```
//方法一
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/
class Solution {
public:
    void TreeDepthHelper(TreeNode *pRoot, int curr, int &max){
        if(pRoot == nullptr){
            if(max < curr){
                max = curr;
            }
            return;
        }
        TreeDepthHelper(pRoot->left, curr+1, max);
        TreeDepthHelper(pRoot->right, curr+1, max);
    }
    int TreeDepth(TreeNode* pRoot)
    {
        if(pRoot == nullptr){
            return 0;
        }
        int depth = 0;
        int max = 0;
        TreeDepthHelper(pRoot, depth, max);
        return max;
    }
};
//方法二
class Solution {
public:
    int TreeDepth(TreeNode* pRoot)
    {
        if(pRoot == nullptr){
            return 0;
        }

        return 1 + max(TreeDepth(pRoot->left), TreeDepth(pRoot->right));
    }
```

```
};
//方法三，层序遍历
class Solution {
public:
    int TreeDepth(TreeNode* pRoot)
    {
        if(pRoot == nullptr){
            return 0;
        }
        //层序大法好
        queue<TreeNode*> q;
        q.push(pRoot);
        int depth = 0;
        while(!q.empty()){
            //每次把当前层全部处理完，切记
            int size = q.size();
            depth++; //每次处理一层，只要还在处理，就说明深度在递增
            for(int i = 0; i < size; i++){ //处理完本层，压人下一层
                TreeNode *curr = q.front();
                q.pop(); //去掉当前节点
                if(curr->left) q.push(curr->left);
                if(curr->right) q.push(curr->right);
            }
        }
        return depth;
    }
};
```

Java代码

```
//方法一
/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }

}
*/
public class Solution {
    private int max = 0;
    public void TreeDepthHelper(TreeNode root, int curr){
        if(root == null){
            if(max < curr){
                max = curr;
            }
            return;
        }
```

```
        TreeDepthHelper(root.left, curr+1);
        TreeDepthHelper(root.right, curr+1);
    }
    public int TreeDepth(TreeNode root) {
        if(root == null){
            return 0;
        }
        int depth = 0;
        TreeDepthHelper(root, depth);
        return max;
    }
}
//方法二
public class Solution {
    public int TreeDepth(TreeNode root) {
        if(root == null){
            return 0;
        }

        return Math.max(TreeDepth(root.left), TreeDepth(root.right)) + 1;
    }
}
//方法三
/**
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }

}
*/
import java.util.Queue;
import java.util.LinkedList;

public class Solution {
    public int TreeDepth(TreeNode root) {
        if(root == null){
            return 0;
        }
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);    //将root入队列
        int depth = 0;
        while(!q.isEmpty()){
            int size = q.size();  //获取队列中的元素个数
            depth++;
            for(int i = 0; i < size; i++){
                TreeNode curr = q.poll(); //获取队列头部元素,并在队列中删除
                if(curr.left != null) q.offer(curr.left);
```

```
                if(curr.right != null) q.offer(curr.right);
            }
        }
        return depth;
    }
}
```

30. 核心考点：异或理解，位运算

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

解析：

```
//OJ链接：https://www.nowcoder.com/practice/e02fdb54d7524710a7d664d082bb7811?tpId=13&tqId=11193&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//问题一：如果只有一个数据单独出现，直接整体异或得到结果
//但是这道题是两个不重复的数据,我们可以采取先整体异或，异或结果一定不为0，而其中为1的比特位，不同的两个数据该位置上的数据一定不同，所以我们可以用该比特位进行分组
//分组的结果一定是，相同数据被分到了同一组，不同数据一定被分到了不同的组
//问题就转化成了两个问题一
```

C++代码

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

        //第一步先整体异或
        int result = data[0];
        for(auto it = data.begin()+1; it != data.end(); it++){
            result ^= *it;
        }
        //第二步，分解找到第一个比特位为1,这里从低往高进行
        int flag = 1;
        int length = sizeof(int);
        for(int i = 0; i < length; i++){
            if((flag << i) & result){
                flag <<= i;
                break;
            }
        }
        //第三部，分组异或
        *num1 = 0;
        *num2 = 0;
        for(int i = 0; i < data.size(); i++){
            if(data[i] & flag){
                *num1 ^= data[i];
            }
```

```
            else{
                *num2 ^= data[i];
            }
        }
    }
};
```

Java代码

```
//num1,num2分别为长度为1的数组。传出参数
//将num1[0],num2[0]设置为返回结果
public class Solution {
    public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
        if(array == null || num1 == null || num2 == null){
            return;
        }
        num1[0] = 0;
        num2[0] = 0;
        int result = array[0];
        //第一步，将所有数据进行异或
        for(int i = 1; i < array.length; i++){
            result ^= array[i];
        }
        //第二步，根据题面，最终结果一定不为0，找到该数据第一个为1的比特位，从高向底
        int length = Integer.SIZE; //获取int有多少比特位
        int flag = 1;
        while(length >= 0){
            length -= 1;
            if(((flag<<length) & result) != 0){
                flag <<= length;
                break;
            }
        }
        //第三步，分组
        for(int i = 0; i < array.length; i++){
            if((array[i] & flag) == 0){
                num1[0] ^= array[i];
            }else{
                num2[0] ^= array[i];
            }
        }
    }
}
```

31. 核心考点：场景转化为模型，滑动窗口

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!

输出描述:

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

解析：

```
//OJ链接：https://www.nowcoder.com/practice/c451a3fd84b64cb19485dad758a55ebe?tpId=13&tqId=11194&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//注意，代码中只给你你目标和，其实隐含条件就是序列最大值不会大于该值
//可以采用滑动窗口的思想进行解决，这种方式可以说很nb了
```

C++代码

```
class Solution {
public:
    vector<vector<int> > FindContinuousSequence(int sum) {
        vector<vector<int> >result;
        //不考虑负数情况
        //定义两个位置，表示其实和结束，start和end之间就是一段连续递增的序列
        //两个起点，相当于动态窗口的两边，根据其窗口内的值的和来确定窗口的位置和大小
        //我们采用闭区间方式进行解决
        int low = 1;
        int high = 2;
        while(low < high){
            //由于是连续的，差为1的一个序列，那么求和公式是(a0+an)*n/2
            //这里的的low和high不光代表位置，也代表对应的值
            int total = (low + high)*(high - low + 1)/2; //注意，是闭区间哦
            if(sum == total){
                //说明该序列中的数据和符合要求
                vector<int> v;
                for(int i = low; i <= high; i++){
                    v.push_back(i);
                }
                result.push_back(v);
                //注意，这里要保证拿到一个序列之后，下一个序列只能从下一个数据开始
                //而high会自动调节
                low++;
            }
            else if(total < sum){
                //说明该序列区间中的数据和小于sum，应该扩大区间,以包含更多数据
                high++;
            }
            else{
                //说明该序列区间中的数据和大于sum，应该缩小区间,以包含较少数据
                low++;
            }
        }
        return result;
    }
};
```

Java代码

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<ArrayList<Integer> > FindContinuousSequence(int sum) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<>();
        //不考虑负数情况
        //定义两个位置，表示其实和结束，start和end之间就是一段连续递增的序列
        //两个起点，相当于动态窗口的两边，根据其窗口内的值的和来确定窗口的位置和大小
        //我们采用闭区间方式进行解决
        int low = 1;
        int high = 2;
        while(low < high){
            //由于是连续的，差为1的一个序列，那么求和公式是(a0+an)*n/2
            //这里的的low和high不光代表位置，也代表对应的值
            int total = (low + high)*(high-low+1)/2;
            if(total == sum){
                ArrayList<Integer> list = new ArrayList<>();
                for(int i = low; i <= high; i++){
                    list.add(i);
                }
                result.add(list);
                low++;
            }
            else if(total < sum){
                //说明该序列区间中的数据和小于sum，应该扩大区间,以包含更多数据
                high++;
            }
            else{
                //说明该序列区间中的数据和大于sum，应该缩小区间,以包含较少数据
                low++;
            }
        }
        return result;
    }
}
```

32. 核心考点：字符串逆置，循环次数去重

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！

解析：

```
//OJ链接：https://www.nowcoder.com/practice/12d959b108cb42b1ab72cef4d36af5ec?tpId=13&tqId=11196&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//思路一：保存第一个，剩下的整体前移一个，第一个放在最后，完成一次移动，一次能移动，多次也可以
//思路二：局部逆置，在整体逆置
```

C++代码

```
//方法一，一个一个迁移
class Solution {
public:
    void LeftRotateStringOne(string &str){
        char temp = str[0];
        int len = str.size();
        int i = 0;
        for(;i < len-1; i++){
            str[i] = str[i+1];
        }
        str[i] = temp;
    }
    string LeftRotateString(string str, int n) {
        if(n == 0 || str.empty()){
            return str;
        }
        n %= str.size();
        while(n){
            LeftRotateStringOne(str);
            n--;
        }
        return str;
    }
};
//方法二，先局部逆置，在整体逆置
class Solution {
public:
    void ReverseString(string &str, int start, int end){
        while(start < end){
            char temp = str[start];
            str[start]  = str[end];
            str[end]    = temp;
            start++, end--;
        }
    }
    string LeftRotateString(string str, int n) {
        if(n == 0 || str.empty()){
            return str;
        }
        n %= str.size();                    //去除重复移动
        ReverseString(str, 0, n-1);         //前半部分逆置
        ReverseString(str, n, str.size()-1); //后半部分逆置
        ReverseString(str, 0, str.size()-1); //整体逆置
        return str;
    }
};
```

Java代码

```
//很多在线oj上面的代码，都是不符合要求的，在正式面试的时候，面试官最看重的是思想设计

//方法一
```

```
public class Solution {
    public void LeftRotateStringHelper(char[] list){
        char temp = list[0];
        int i = 0;
        for(; i < list.length-1; i++){
            list[i] = list[i+1];
        }
        list[i] = temp;
    }
    public String LeftRotateString(String str,int n) {
        //java这里处理字符串方式特殊，我们这里重点特出算法设计，所以尽量不要用太多内置函数
        if(str.length() == 0 || n == 0){
            return str;
        }
        n %= str.length(); //有很多左移是重复的
        char[] list = str.toCharArray();
        while(n > 0){
            n--;
            LeftRotateStringHelper(list); //左移一次
        }
        return new String(list);
    }
}

//方法二
public class Solution {
    public void Reverse(char[] list, int start, int end){
        while(start < end){
            char temp = list[start];
            list[start] = list[end];
            list[end] = temp;
            start++;
            end--;
        }
    }
    public String LeftRotateString(String str,int n) {
        if(str.length() == 0 || n == 0){
            return str;
        }

        n %= str.length();
        char[] list = str.toCharArray();
        Reverse(list, 0, n-1);
        Reverse(list, n, str.length()-1);
        Reverse(list, 0, str.length()-1);

        return new String(list);
    }
}
```

33. 核心考点：子串划分，子串逆置

公司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?

解析：

```
//OJ链接：https://www.nowcoder.com/practice/3194a4f4cf814f63919d0790578d51f3?tpId=13&tqId=11197&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//又是一个字符串逆置的问题，我们可以参考上面的思路。不过这里要考虑自己划分子串的问题
```

C++代码

```
class Solution {
public:
    void Reverse(string &str, int start, int end){
        while(start < end){
            char temp = str[start];
            str[start] = str[end];
            str[end] = temp;
            start++;
            end--;
        }
    }
    string ReverseSentence(string str) {
        //先局部逆置，在整体逆置
        if(str.size() == 0){
            return str;
        }
        int i = 0;
        int j = i;
        int len = str.size();
        while(i < len){
            while( i < len && !isspace(str[i])) i++; //让i一直往后走，碰到第一个空格
            Reverse(str, j, i-1); //逆置当前有效子串，我们采用闭区间
            while( i < len && isspace(str[i])) i++; //过滤所有空格，指向下一个有效子串的开始
            j = i; //保存起始位置
        }
        //走到这里，一定是 i == str.size()了，但是最后一个有效子串并没有被逆置,逆置之
        Reverse(str, j, i-1);
        Reverse(str, 0, i-1); //最后在整体逆置
        return str;
    }
};
```

Java代码

```
public class Solution {
    public void Reverse(char[] list, int start, int end){
        while(start < end){
            char temp = list[start];
```

```
            list[start] = list[end];
            list[end] = temp;
            start++;
            end--;
        }
    }
    public String ReverseSentence(String str) {
        //java这里，我们不采用StringBuffer的做法，直接展示最原生态的算法思路
        if(str == null || str.length() == 0){
            return str;
        }

        char[] list = str.toCharArray();
        int len = list.length;
        int i = 0;
        int j = i;
        while(i < len){
            //让i一直往后走，碰到第一个空格
            while(i < len && !Character.isSpace(list[i])) i++;
            //逆置当前有效子串，我们采用闭区间
            Reverse(list, j, i-1);
            //过滤所有空格，指向下一个有效子串的开始
            while(i < len && Character.isSpace(list[i])) i++;
            j = i;//保存起始位置
        }
        //走到这里，一定是 i == str.size()了，但是最后一个有效子串并没有被逆置,逆置之
        Reverse(list, j, i-1);
        Reverse(list, 0, i-1);
        return new String(list);
    }
}
```

34. 核心考点：树遍历，stack，queue结合使用

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/91b69814117f4e8097390d107d2efbe0?
tpId=13&tqId=11212&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
//之字形打印，本质也是对树形结构的层序遍历，不过在遍历的过程中，需要更改遍历顺序
//我们可以采用stack和queue的方式来进行处理
//核心思路:当前层从左向右遍历，那么下层就从left到right入栈，当前层如果从右向左遍历，那么下层就从right到left
入栈
```

C++代码

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

```
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};
*/
//网上的方案都很写的很复杂，我们来一个简单的
class Solution {
public:
    vector<vector<int> > Print(TreeNode* pRoot) {
        vector<vector<int> > result;
        if(pRoot == nullptr){
            return result;
        }
        stack<TreeNode*> st; //保存要遍历的节点
        queue<TreeNode*> q;  //作为临时队列来进行辅助
        st.push(pRoot);
        int dir = 1; //1. 入栈顺序从left开始.  2.入栈顺序从right开始.
        vector<int> v;
        while(!st.empty()){
            int size = st.size();
            for(int i = 0; i < size; i++){ //清空本轮stack结构，并遍历，stack本身有逆序的功能
                TreeNode *curr = st.top();
                st.pop();
                v.push_back(curr->val);
                TreeNode *first = (dir == 1)?curr->left:curr->right;
                TreeNode *second = (dir == 1)?curr->right:curr->left;
                //将下一轮访问顺序放入q中
                if(first != nullptr) q.push(first);
                if(second != nullptr) q.push(second);
            }
            //将本层符合要求的所有节点，入result
            result.push_back(v);
            //将下一轮访问节点入栈，进行逆序
            while(!q.empty()){
                st.push(q.front());
                q.pop();
            }
            //一层遍历完毕，就要更改入栈顺序
            dir = (dir == 1)? 2 : 1;
            v.clear();
        }
        return result;
    }
};
```

Java代码

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Stack;
import java.util.Queue;
```

```
/*
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }

}
*/

public class Solution {
    public ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer> > result = new ArrayList<>();
        if(pRoot == null){
            return result;
        }
        Stack<TreeNode> st = new Stack<>();
        Queue<TreeNode> q = new LinkedList<>(); //临时区域
        st.push(pRoot);
        int dir = 1; //1: 代表left->right式入栈. 2: 代表right->left式入栈
        ArrayList<Integer> list = new ArrayList<>();//保存一层结果的临时变量
        while(!st.empty()){
            int size = st.size(); //清空本层所有节点，将下层节点按照要求入栈，栈具有天然的逆序的能力
            for(int i = 0; i < size; i++){
                TreeNode curr = st.pop();
                list.add(curr.val);
                TreeNode first = (dir == 1) ? curr.left : curr.right;
                TreeNode second = (dir == 1) ? curr.right : curr.left;
                if(first != null) q.offer(first);
                if(second != null) q.offer(second);
            }
            //本层遍历完毕，入结果集
            result.add(new ArrayList(list)); //一定要注意浅拷贝问题
            list.clear();
            //将所有节点入栈，进行逆序
            while(!q.isEmpty()){
                st.push(q.poll());
            }
            dir = (dir == 1) ? 2 : 1;
        }
        return result;
    }
}
```

35. 核心考点：BST有序性理解

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。

解析：

```
//OJ链接：https://www.nowcoder.com/practice/ef068f602dde4d28aab2b210e859150a?tpId=13&tqId=11215&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking
//解题思路：
////BST本身就是有序的，中序遍历即是升序
//要求第k小，即中序遍历时到达第k个元素(二叉搜索树，不存在两个相同的节点值)
//此处，我们不使用递归，我们采用循环中序遍历的方式
```

C++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* KthNode(TreeNode* pRoot, int k)
    {
        //我们认为，第k个，从k开始
        if(pRoot == nullptr || k <= 0){
            return nullptr;
        }
        stack<TreeNode*> st;
        TreeNode *node = pRoot;
        do{
            while(node != nullptr){ //将左子树全部入栈
                st.push(node);
                node = node->left;
            }
            if(!st.empty()){
                node = st.top();
                st.pop();
                k--;
                if(k == 0){
                    return node; //找到了该节点
                }
                node = node->right;
            }
        }while(node != nullptr || !st.empty());
        //node有可能为空，但是只要stack不为空，就要继续pop求下一个。有没有可能st为空？有可能，这个时候就要检测node，如果node不为空，就要整体检测右子树
        //走到这里，就说明没有找到
        return nullptr;
    }
};
```

Java代码

```
/*
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }

}
*/
import java.util.Stack;
//BST本身就是有序的，中序遍历即是升序
//要求第k小，即中序遍历时到达第k个元素(二叉搜索树，不存在两个相同的节点值)
//此处，我们不使用递归，我们采用循环中序遍历的方式
public class Solution {
    TreeNode KthNode(TreeNode pRoot, int k)
    {
        if(pRoot == null || k <= 0){
            return null;
        }
        Stack<TreeNode> st = new Stack<>();
        TreeNode node = pRoot;
        do{
            while(node != null){
                st.push(node);
                node = node.left;
            }
            if(!st.empty()){
                node = st.pop();
                //访问当前节点，中序
                k--;
                if(k == 0){
                    return node;//找到当前第k小节点
                }
                node = node.right;
            }
        }while(node != null || !st.empty()); //node有可能为空，但是只要stack不为空，就要继续pop求下一个。有没有可能st为空？有可能，这个时候就要检测node，如果node不为空，就要整体检测右子树
        //走到这里，就说明没有找到
        return null;
    }
}
```