出处：
https://github.com/ZYZMZM
https://github.com/IdiotXue/OfferCoding
打算结合，再梳理下剑指**offer**，以前没有刷过算法题目。

1. 赋值运算符函数
key:【返引用、传常引用、判同、局部变量换指针（RAII）】
2. 实现Singleton模式
key:【返引用，局部静态变量，编译器保证】
3. 数组中重复的数字
key:【判nullptr与越界、从0至len,if(A[i] != i){ if(A[i] == A[A[i]]) dup ; swap }】
4. 二维数组中的查找
key:【左上>右上>右下有序, target< 往左，>target往下】
5. 替换空格
key:【算str长度（一个空格+2），newLen != oldLen且不越界，逆序复制，遇空格则反向填充3字符】
6. 从尾到头打印链表
key:【vector+reverse、stack<ListNode *>+vectorListNode *node = head】
7. 重建二叉树
key:【中序左子树长度（前序（根）），】
8. 二叉树的下一个结点
key: [有右子树，右子树最左边的结点；无右子树，它的父节点向上回溯]
9. 用两个栈实现队列
key:【push1; cp1->2，2pop，cp2->1】
10. 斐波那契数列
key: [res = first + second;移动窗口]
11. 变态跳台阶
key:【台阶顶必存在，其他台阶可存在或不存在,故2^(n-1)】
12. 矩形覆盖
key:【斐波那契数列的变形res = m + n;】
13. 旋转数组的最小数字
key:【去右等值，判断有序，再二分】
14. 矩阵中的路径
key: [参数错误，路过标记，逐个比较，4个方向，恢复未途径]
15. 机器人的运动范围
key:【key: [参数错误，路过标记，逐个比较，4个方向】
16. 二进制中1的个数
key:【n & (n - 1)，最后一位1变成0，并更新n】
17.数值的整数次方
key:【A^B ; (B<0, 1/B)】
18. 删除链表中重复的结点
key: [虚拟头，相邻相等去除，不等则preNode后移，curNode始终后移]
19.正则表达式匹配
key:【下一位*，匹配或（.）=.=》str+1或pattern+2，否则pattern+2 ；下一位非*，匹配或（.）=.=》str+1且pattern+1，否则false】
20.表示数值的字符串
key:【排除反例，顺序E ; + ; . 】
21.调整数组顺序使奇数位于偶数前面
key: [冒泡修改法，或空间换时间]
22.链表中倒数第k个结点
key:【快慢指针，移动窗口法，快先N，慢再走，快到尾】
23.链表中环的入口结点
key:【快慢指针:1.相遇有环，2.遇后++求环长len; 3.快先跑一圈N，慢再走，快慢同速，快到尾{a+b = (n-1)*len}】
24.反转链表
key:【tmp=old->next，old-next=newHead，new=old，old=tmp】

25.合并两个排序的链表

key:【逐个比，遇nullptr,接other】

26.树的子结构

key:【1.判RootB在TreeA中；2.若在则递归左右子节点，B到达nullptr返回true,A为nullptr而B非nullptr则false】

27.二叉树的镜像

key【root非nullptrr则交换左右】

28.对称的二叉树

key:【根同nullptr => true;单nullptr=>false；不同value=>false;再递归左右】

29.顺时针打印矩阵

key:【while(up<=down && left<=right) { 左=》右，上=》下（up+1）；（判断是否最后一行/列 右=》左(right+1)；下到上(down - 1,> up)）}】

30.包含min函数的栈

key:【每次将value都push进stackVal；同时将Min.top与value比较push进stackMin 。同时pop出stackVal和stackMin】

31.栈的压入、弹出序列

key:【】

32.从上往下打印二叉树

key:【队列，取头，push队头进结果，push左右分支至队列】

33.把二叉树打印成多行

key:【】

34.按之字形顺序打印二叉树

key:

35.二叉搜索树的后序遍历序列

key:

36.二叉树中和为某一值的路径

key:

37.复杂链表的复制

key:

38.二叉搜索树与双向链表

key:

39.序列化二叉树

key:

40.字符串的排列

key:

41.数组中出现次数超过一半的数字

key:

42.最小的K个数

key:

43.数据流中的中位数

key:

44. 连续子数组的最大和

key:

45.1~n整数中1出现的次数

key:

46.把数组排成最小的数

key:

47.丑数

key:

48.第一个只出现一次的字符位置

key:

49.字符流中第一个不重复的字符

key:

50.数组中的逆序对

key:

51.两个链表的第一个公共结点

key:

52.数字在排序数组中出现的次数

1. 赋值运算符函数

key:【返引用、传常引用、判同、局部变量换指针（RAII）】

```
CMyString& operator=(const CMyString &str)
//考点1：返回引用，才能连续赋值str1=str2=str3
//考点2：传入参数类型为常量引用，避免调用复制构造和函数内改变rhs
//考点3：判断传入rhs和当前this是否为同一实例，避免释放自身空间
//考点4：创建临时实例，交换空间，出if后自动释放，指针tmp不会释放
```

> 题目描述
>
> 如下为类型**CMyString**的声明，请为该类型添加赋值运算符函数。

```
class CMyString {
public:
    CMyString(char *pData = nullptr);
    CMyString(const CMyString &str);
    ~CMyString(void);
private:
    char *m_pData;
};
```

> 解法一：经典解法

```
CMyString& CMyString::operator=(const CMyString &str){
    if(this == &str)
        return *this;

    delete []m_pData;
    m_pData = nullptr;

    m_pData = new char[strlen(str.m_pData) + 1]; //new失败时抛出异常
    strcpy(m_pData, str.m_pData);

    return *this;
}
```

> 解法二：考虑异常安全的版本

```
CMyString &CMyString::operator=(const CMyString &rhs){
    //考点1：返回引用，才能连续赋值str1=str2=str3
    //考点2：传入参数类型为常量引用，避免调用复制构造和函数内改变rhs
    //考点3：判断传入rhs和当前this是否为同一实例，避免释放自身空间
    if (this != &rhs){
        //考点4：创建临时实例，交换空间，出if后自动释放，指针tmp不会释放
        //这种做法是异常安全的(delete/new之间发送分配空间异常CMyString
        //无法保持有效状态)
        CMyString tmpString(rhs);
        char *tmp = m_pData;        //1.相当于备份原始A指针,C = A ,
        m_pData = tmpString.m_pData;//1.A = B ,B = C.交换
        tmpString.m_pData = tmp;
    }
    return *this;
}
```

> 完整`CMyString`

```
class CMyString
{
    friend ostream &operator<<(ostream &out, const CMyString &str){
        out << str.m_pData;
        return out;
    }

  public:
    CMyString(const char *pData = nullptr) : m_pData(nullptr){
        if (pData != nullptr){
            //strlen不包括'\0'
```

```
            m_pData = new char[strlen(pData) + 1];
            strcpy(m_pData, pData);
        }
    }
    CMyString(const CMyString &str) : m_pData(nullptr){
        if (str.m_pData != nullptr){
            //strlen不包括'\0'
            m_pData = new char[strlen(str.m_pData) + 1];
            strcpy(m_pData, str.m_pData);
        }
    }
    CMyString &operator=(const CMyString &);
    ~CMyString(){
        delete m_pData;
        m_pData = nullptr;
    }

  private:
    char *m_pData;
};
//最好再声明一次，外部函数才能调用
ostream &operator<<(ostream &out, const CMyString &str);
```

2. 实现Singleton模式

key:【返引用，局部静态变量，编译器保证】

> 题目描述
>
> 设计一个类，我们只能生成该类的一个实例。

这是最推荐的一种单例实现方式：

> 1. 通过局部静态变量的特性保证了线程安全 (C++11, GCC > 4.3, VS2015支持该特性);
> 2. 不需要使用共享指针，代码简洁；
> 3. 注意在使用的时候需要声明**单例的引用 `Single&` 才能获取对象**。

```
class Singleton{
public:
    ~Singleton()= default;
    Singleton(const Singleton&)=delete;             //delete
    Singleton& operator=(const Singleton&)=delete;  //delete

    static Singleton& get_instance(){  //return 引用
        static Singleton instance;     //C++11,安全
        return instance;
    }
};
//C++11，若当变量在初始化时，并发同时进入声明语句，并发线程将会阻塞等待初始化结束。所以具有线程安全性。
```

单例的模板基类

```
// brief: a singleton base class offering an easy way to create singleton
#include <iostream>
```

```cpp
template<typename T>
class Singleton{
public:
    static T& get_instance() noexcept(std::is_nothrow_constructible<T>::value){
        static T instance{token()};
        return instance;
    }
    virtual ~Singleton()  = default;
    Singleton(const Singleton&) = delete;
    Singleton& operator =(const Singleton&) = delete;
protected:
    struct token{}; // helper class
    Singleton() noexcept = default;
};

/*******************************************/
// Example:
// constructor should be public because protected `token` control the access

class DerivedSingle:public Singleton<DerivedSingle>{
public:
   DerivedSingle(token){//重点这个参数，只能继承过来
       std::cout<<"destructor called!"<<std::endl;
   }

   ~DerivedSingle(){    std::cout<<"constructor called!"<<std::endl;    }
   DerivedSingle(const DerivedSingle&)=delete;
   DerivedSingle& operator =(const DerivedSingle&)= delete;
};

int main(int argc, char* argv[]){
    DerivedSingle& instance1 = DerivedSingle::get_instance();
    DerivedSingle& instance2 = DerivedSingle::get_instance();
    return 0;
}
```

```cpp
/**
 * 懒汉式(线程安全):双重检测锁定（double checked locking）
 * 需要时创建实例，双重检测避免了创建后每次调用都要加锁，提高了效率
 */
#include <iostream>
#include <memory>    //std::shared_ptr
#include <mutex>     //std::mutex,std::lock_guard
//Singleton.h
class Singleton{
  public:
    static const std::shared_ptr<Singleton> GetInstance(){
        if (!sm_pInstance) {//避免创建了实例后，每次调用还要加锁
            std::lock_guard<std::mutex> guard(sm_mutex);
            if (!sm_pInstance)
                sm_pInstance.reset(new Singleton());
        }
        return sm_pInstance;
    }
    ~Singleton() { std::cout << "destruct singleton object" << std::endl; }

  private:
```

```
    static std::shared_ptr<Singleton> sm_pInstance;
    static std::mutex sm_mutex;
    Singleton() { std::cout << "construct singleton object" << std::endl; }
    Singleton(const Singleton &) = delete;
    Singleton &operator=(const Singleton &) = delete;
};
//Singleton.cpp
std::shared_ptr<Singleton> Singleton::sm_pInstance;
std::mutex Singleton::sm_mutex;
```

3. 数组中重复的数字

key: 【判nullptr与越界、从0至len,if(A[i] != i){ if(A[i] == A[A[i]]) dup ; swap }】

> 题目描述
>
> 在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。请找出数组中任意一个重复的数字。 例如，{2,3,1,0,2,5,3}，那么对应的输出是第一个重复的数字2。

> 思路:
>
> - 0、暴力 时间复杂度O(N*N) 空间复杂度O(1) `[i]^[j]`
> - 1、哈希表 时间复杂度O(N) 空间复杂度O(N)
> - 2、根据排序数组的特性，重排数组，原地算法
> - 3、不更改数组，借助O(N)空间，把原数组中值为m的数字复制到新数组中下标为m的位置，会重复。
> - 4、不更改数组， 时间复杂度O(N*logN) ,空间复杂度O(1)，二分查找的区间划分法。

```
class Solution {
public:
    /* 利用该数组的特性，若不存在重复的数字，0-i的数字都在相应的位置上 */
    bool duplicate(int numbers[], int length, int* duplication) {
        if(length < 2 || numbers == nullptr)
            return false;

        for(int i = 0; i < length; ++i)
        {
            if(numbers[i] < 0 || numbers[i] > length - 1)
                return false;//得先判断，否则下面可能越界
        }

        for(int i = 0; i < length; ++i)
        {
            if(numbers[i] != i)
            {
                if(numbers[i] == numbers[numbers[i]])
                {
                    *duplication = numbers[i];
                    //duplication[0] = numbers[i];
                    return true;
                }
                swap(numbers[i], numbers[numbers[i]]);
            }
        }
```

```
        return false;
    }
};
```

```
int duplicate(vector<int> & nums) { //有序情况
int L = 1 ,R = nums.size()-1;
while(L < R){
    int mid = (L + R) >> 1; //[L ,mid] [mid +1 , R ]
    int count = 0;

    for(int i = 0; i< nums.size() ;++i){
        if(nums[i] >=L && nums[i] <= R){
            count ++;
        }
    }//计数
    if(count > mid - L +1){//重复在[L ,mid]
        R = mid ;
    }
    else{//[mid +1 , R ]
        L = mid + 1;
    }
    return L;
}
```

4. 二维数组中的查找

key: 【左上>右上>右下有序, target< 往左，>target往下】

> 题目描述
>
> 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。
>
> 思路：
>
> 根据题意，是左到右递增，上到下递增，可从右上角开始，小于**target**，向下走，大于**target**，向左走。

```
class Solution {
public:
    bool Find(int target, vector<vector<int>> & array) {
        if(array.empty() || array[0].empty())
            return false;

        int rows = array.size();
        int cols = array.at(0).size();

        int i   = 0;
        int j   = cols -1;
        while(( i <= rows -1) &&( j >=0)){
```

```
            if(array [i][j] == target){
                return true;
            }

            if(array [i][j] > target){
                --j;//向左走
            }
             if(array [i][j] < target){
                 ++i;//向下走
            }
        }//end while
         return false;
    }
};
```

5. 替换空格

key:【算str长度（一个空格+2），newLen != oldLen且不越界，逆序复制，遇空格则反向填充3字符】

> 题目描述
>
> 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。
>
> 思路:
>
> 把空格替换成%20，共三个字符，那么我们对于每个空格要扩充两个字符的空间，因此我们首先计算空格的个数，然后对个数乘2加上原来的长度，便为新字符串的长度。然后两个变量，一个指向新串末尾，一个指向原串末尾，指向原串末尾的指针遇到空格就在新串末尾将替换字符串反序添加，否则，直接将原串字符添加到新串中。

```
class Solution {
public:
    void replaceSpace(char *str,int length) {
        if (str == nullptr || length <= 0)
            return;

        int newStrLen = length;
        for(int i =0 ; i < length -1;i++){
            if(str[i] == ' '){
                newStrLen += 2;
            }
        }

        int index = length-1;
        int decIndex = newStrLen-1;
        while(index >=0 && (decIndex != index)){
            if(str[index] !=' '){
                str[decIndex--] = str[index--];
            }
            else{
                str[decIndex--] ='0';
                str[decIndex--] ='2';
                str[decIndex--] ='%';
            }
```

```
        }
    }
};
```

6. 从尾到头打印链表

key:【vector+reverse、stack<ListNode *>+vectorListNode *node = head】

> 题目描述
>
> 输入一个链表，按链表从尾到头的顺序返回一个**ArrayList**。
>
> 思路:
>
> 可以采用栈来存储链表中的每个值，然后依次出栈并存入结果集中，返回即可。

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

        ListNode *node = head;//保存指针，拷贝指针比拷贝对象快，本例是int体现不出来
        stack<ListNode *> TempStack;
        while (node)
        {
            TempStack.push(node);
            node = node->next;
        }
        vector<int> vnTemp(TempStack.size());
        for (size_t i = 0; i < vnTemp.size(); ++i)
        {
            vnTemp[i] = TempStack.top()->val;
            TempStack.pop();
        }
        return vnTemp;
    }
    //解法2
    vector<int> printListFromTailToHead(ListNode* head) {
        vector<int> retVector;

        ListNode *node = head;//不应该修改head的
        while(node != nullptr){
            retVector.push_back(node->val);
            node =node->next;
```

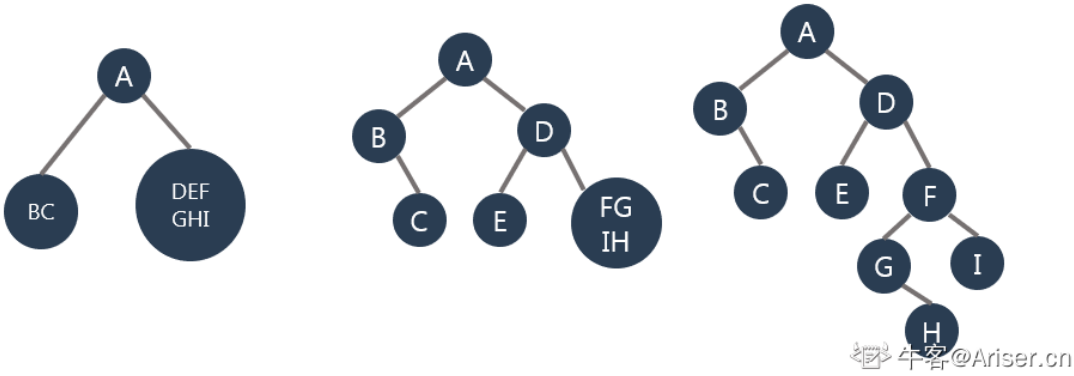
```
        }
        //return vector<int>(retVector.rbegin(), retVector.rend());

        reverse(retVector.begin(),retVector.end());//反转vector
        return retVector;
    }
};
```

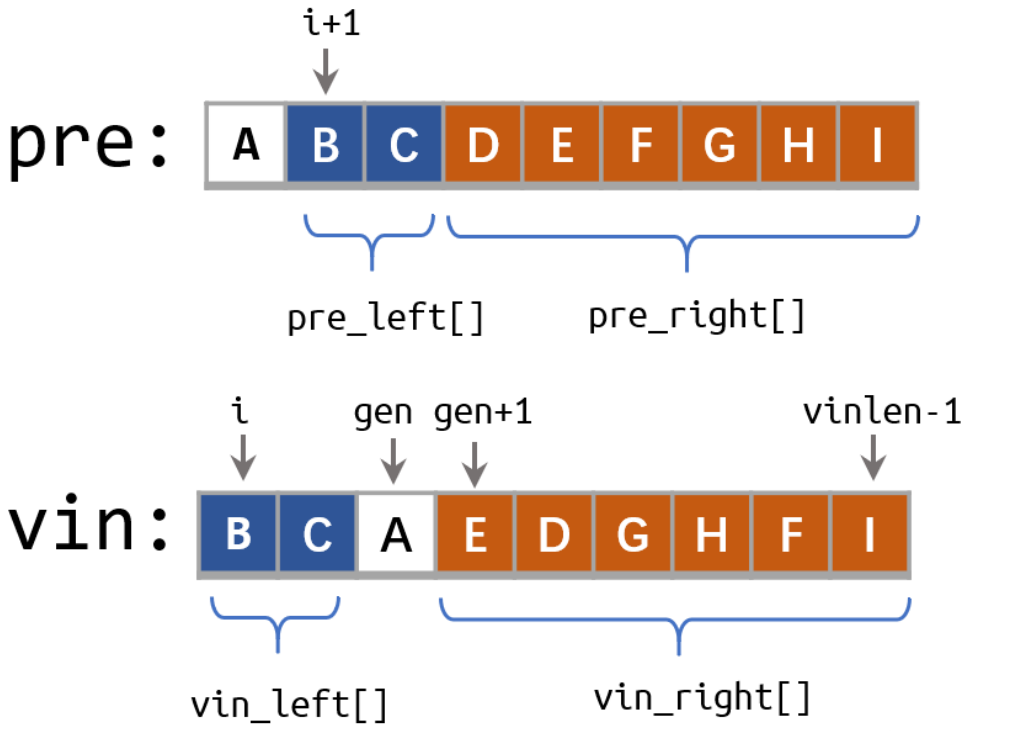
7. 重建二叉树

key:【中序左子树长度（前序（根）），】

> 题目描述
>
> 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。

> 思路：
> 前序遍历是**{**根，左子树，右子树**}**；中序是**{**左子树，根，右子树**}**
> 因此，按如下操作即可
> （**1**）在前序中拿出根（第一个元素），用于在中序中分出左右子树
> （**2**）根据中序得出的左右子树元素个数，在前序中分出左右子树
> （**3**）在（**1**）（**2**）中得到左右子树的前序和中序遍历，是原问题的子集
> 即：左右子树的根就是（**1**）中根的左右子结点
> （**4**）递归构建左右子树，直到叶子结点为止
> 特例：（**1**）前中序大小不同或大小为**0**（**2**）无法构成二叉树（**3**）部分无左子树或右子树
>
> 前序加中序序列，分解过程图示如下（王道数据结构**P120**）





思路：

1. 由先序序列第一个`pre[0]`在中序序列中找到根节点位置`gen`
2. 以`gen`为中心遍历
   - `0~gen`左子树
     - 子中序序列：`0~gen-1`，放入`vin_left[]`
     - 子先序序列：`1~gen`放入`pre_left[]`，`+1`可以看图，因为头部有根节点
   - `gen+1~vinlen`为右子树
     - 子中序序列：`gen+1 ~ vinlen-1`放入`vin_right[]`
     - 子先序序列：`gen+1 ~ vinlen-1`放入`pre_right[]`
3. 由先序序列`pre[0]`创建根节点
4. 连接左子树，按照左子树子序列递归（`pre_left[]`和`vin_left[]`）
5. 连接右子树，按照右子树子序列递归（`pre_right[]`和`vin_right[]`）
6. 返回根节点

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

```
class Solution {
public:
    unordered_map<int,int> pos;
    TreeNode* reConstructBinaryTree(vector<int> pre,vector<int> vin) {
        int n = pre.size();
        for(int i = 0; i < n; i ++){
            pos[vin[i]] = i;//中序遍历;各左子树的长度，根左边的长度。
        }
        return dfs(pre, vin, 0, n - 1, 0, n - 1);
    }
private:

    //前序遍历序列{3,9,20,15,7}
    //中序遍历序列{9,3,15,20,7}
    //后序遍历序列{9,15,7,20,3}
    //        3
    //       / \
    //     9    20
    //    / \   / \
    //   N   N 15  7

    TreeNode* dfs(vector<int> &pre, vector<int> &vin,
                            size_t nPreLeft, size_t nPreRight,
                            size_t nVinLeft, size_t nVinRight)
    {
        //前序 根左右  pre[pl]
        if(nPreLeft > nPreRight) return nullptr;
        TreeNode* root = new TreeNode(pre[nPreLeft]);  //pre[pl]根

        int leftLen = pos[pre[nPreLeft]] - nVinLeft;
        //gen = vl + leftLen

        //左子树 前序[(pl+1),(pl+len)]；   【1~gen】
        //左子树 中序[(vl),(vl+len-1)]；   【0~gen-1】
        root -> left = dfs(pre, vin, nPreLeft + 1, nPreLeft + leftLen,
                           nVinLeft, nVinLeft + leftLen - 1);
                           //中序起点，为左子树最右边界

        //右子树：【gen+1~vinlen】
        //前序[(pl+len)+1,pr]
        //中序[(vl+len-1),vr]
        root -> right = dfs(pre, vin,nPreLeft + leftLen + 1, nPreRight,
                            //前序终点，为右子树最左边界
                            nVinLeft + leftLen + 1, nVinRight);
        return root;
    }
```

```
};
```

递归遍历

```
void PreorderTraverse(TreeNode *pRoot) //前遍历
{
    if (pRoot == nullptr)
        return;
    std::cout << pRoot->val << " ";
    PreorderTraverse(pRoot->left);
    PreorderTraverse(pRoot->right);
}
void InorderTraverse(TreeNode *pRoot) //中序遍历
{
    if (pRoot == nullptr)
        return;
    InorderTraverse(pRoot->left);
    std::cout << pRoot->val << " ";
    InorderTraverse(pRoot->right);
}
void PostorderTraverse(TreeNode *pRoot) //后序遍历
{
    if (pRoot == nullptr)
        return;
    PostorderTraverse(pRoot->left);
    PostorderTraverse(pRoot->right);
    std::cout << pRoot->val << " ";
}
```

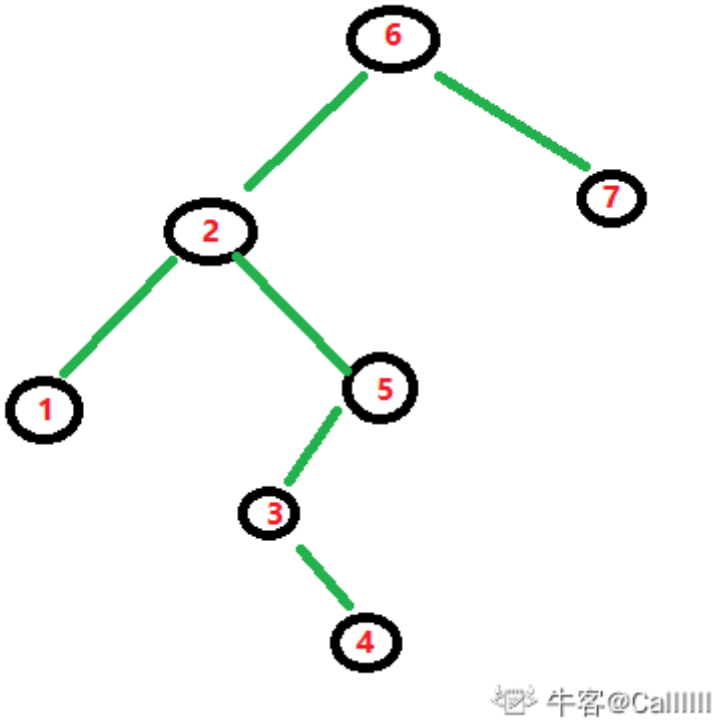
```
//        3
//       / \
//      9   20
//     / \  / \
//    N  N 15  7
//前序遍历序列{3,9,20,15,7}
//中序遍历序列{9,3,15,20,7}
//后续遍历序列{9,15,7,20,3}
//更简单的非递归前序遍历        根，左，右 用栈（反序）
void preorderTraversalNew(TreeNode *root, std::vector<int> &path)
{
    std::stack<std::pair<TreeNode *, bool> > tmpStack;
    bool isVisited = false;
    tmpStack.push(make_pair(root, isVisited));//false

    while(!tmpStack.empty())
    {
        root = tmpStack.top().first;
        isVisited = tmpStack.top().second;
        tmpStack.pop();
        if(root == nullptr)
            continue;
        if(isVisited)
        {
            path.push_back(root->val);
        }
```

```
        else//用栈（反序）
        {
//              //前序
//              tmpStack.push(make_pair(root->right, false));       //右
//              tmpStack.push(make_pair(root->left, false));        //左
//              tmpStack.push(make_pair(root, true));               //根

//              //中序
//              tmpStack.push(make_pair(root->right, false));       //右
//              tmpStack.push(make_pair(root, true));               //根
//              tmpStack.push(make_pair(root->left, false));        //左

            //后序
            tmpStack.push(make_pair(root, true));               //根
            tmpStack.push(make_pair(root->right, false));       //右
            tmpStack.push(make_pair(root->left, false));        //左
        }
    }
}
```

8. 二叉树的下一个结点

key: [有右子树，右子树最左边的结点；无右子树，它的父节点向上回溯]

> 题目描述
>
> 给定一个二叉树和其中的一个结点，请找出**中序遍历顺序的下一个结点并且返回**。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。
>
> 但是，如果在面试中，方法一肯定上不了台面。但是最优解法该怎么去想呢？想不出来就画图分析，举个中序遍历的图：如下：



红色数字是中序遍历的顺序。接下来，我们就假设，如果当前结点分别是**1,2 ... 7**，下一结点看有什么规律没?

复制代码

```
1 => 2 // 显然下一结点是 1 的父亲结点
2 => 3 // 下一节点是当前结点右孩子的左孩子结点，其实你也应该想到了，应该是一直到左孩子为空的那个结点
3 => 4 // 跟 2 的情况相似，当前结点右孩子结点的左孩子为空的那个结点
4 => 5 // 5 是父亲结点 3 的父亲结点，发现和1有点像，因为 1，3,同样是父亲结点的左孩子
5 => 6 // 跟 4=>5 一样的道理
6 => 7 // 跟 3=>4 一样的道理
7 => null // 因为属于最尾结点
```

此时，可以总结一下:
[1] 是一类：特点：当前结点是父亲结点的左孩子
[2 3 6] 是一类，特点：当前结点右孩子结点，那么下一节点就是：右孩子结点的最左孩子结点,如果右孩子结点没有左孩子就是自己
[4 5]是一类，特点：当前结点为父亲结点的右孩子结点，本质还是**[1]**那一类
[7]是一类，特点：最尾结点

```
/*
struct TreeLinkNode {
    int val;
    struct TreeLinkNode *left;
    struct TreeLinkNode *right;
    struct TreeLinkNode *next;//父结点的指针
    TreeLinkNode(int x) :val(x), left(NULL), right(NULL), next(NULL) {

    }
};
*/
class Solution {
public:
```

```
    TreeLinkNode* GetNext(TreeLinkNode* pNode)
    {
        if (!pNode) {
            return pNode;
        }

        // 属于[2 3 6]类
        if (pNode->right) {
            pNode = pNode->right;
            while (pNode->left) {
                pNode = pNode->left;
            }
            return pNode;
        }

        // 属于 [1] 和 [4 5]
        while (pNode->next) {
            TreeLinkNode *root = pNode->next;
            if (root->left == pNode) {
                return root;
            }
            pNode = pNode->next;
        }

        // 属于[7]
        return nullptr;
    }
};
```

9. 用两个栈实现队列

key: 【push1; cp1->2，2pop，cp2->1】

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

```
class Solution
{
public:
    void push(int node) {
        stack1.push(node);
    }
    void copy(stack<int> &inStack, stack<int> &outStack){
        while(inStack.size()){
            outStack.push(inStack.top());
            inStack.pop();
        }
    }

    int pop() {
        copy(stack1, stack2);
        int res = stack2.top();
        stack2.pop();
        copy(stack2, stack1);
        return res;
```

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

10. 斐波那契数列

key: [res = first + second;移动窗口]

> 题目描述
>
> 大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。n<=39

```
class Solution {
public:
    int Fibonacci(int n) {
        if(n == 0 || n == 1)
            return n;
        int first = 0, second = 1;
        int res = 0;
        for(int i = 2; i <= n; i ++)
        {
            res = first + second;
            first = second;
            second = res;
        }
        return res;
    }
};
```

11. 变态跳台阶

key:【台阶顶必存在，其他台阶可存在或不存在,故2^(n-1)】

> 题目描述
>
> 一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。
>
> 思路：可以采用数学归纳法得出，$f(n) = 2^{n-1}$

```
     //f(1) = 1
     //f(2) = f(1) + 1 = 2
     //f(3) = f(1) + f(2) + 1 (加1相当于从直接跳上n级)
class Solution {
public:
    int jumpFloorII(int number) {
        return 1<<(number-1);
    }
};
```

12. 矩形覆盖

key: 【斐波那契数列的变形res = m + n;】

> 题目描述
>
> 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?

```
class Solution {
    /* 动态规划题目
     * 竖着放，f(n-1)种方法填充右边
     * 横着放，最下角必须横着一个，需要f(n-2)种方法填充 边
     * 推广：使用 m*1 去 填充 m*n 的矩形，f(n) = f(n-1) + f(n-m)
     * 本题又是斐波那契数列的变形*/
    int rectCover(int number) {
        if(number == 1 || number == 2)
            return number;

        int m = 1,n = 2,res = 0;
        for(int i = 3; i <= number; ++i){
            res = m + n;
            m = n;
            n = res;
        }
        return res;
    }
};
```

13. 旋转数组的最小数字

key: 【去右等值，判断有序，再二分】

> 题目描述
>
> 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{2,3,4,5,1,2}为{1,2,2,3,4,5}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。
>
> 用中间值和高低位比较，看递增/递减，再缩小范围。

```
class Solution {
public:
    int minNumberInRotateArray(vector<int> &nums) {
        if (nums.empty()) //空数组
            return 0;

        // 2,4,1,2,2,2
        // 4,1,2,2,2,2
        size_t left = 0, right = nums.size() - 1;//二分
        while(nums[right] == nums[0] && right > 0)//移除右边相等元素
            right --;

        if(nums[left] <= nums[right])
```

```
            return nums[left];
        //else ... return nums[rigth]

        while(left < rigth){
            int mid = left + rigth >> 1; //[left, mid] [mid+1, rigth]
            if(nums[mid] < nums[0])
                rigth = mid;
            else
                left = mid + 1;
        }
        return nums[rigth];
    }
};
```

14. 矩阵中的路径

key: [参数错误，路过标记，逐个比较，4个方向，恢复未途径]

> 题目描述
>
> 请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如 **a b c e s f c s a d e e** 矩阵中包含一条字符串**"bccced"**的路径，但是矩阵中不包含**"abcb"**路径，因为字符串的第一个字符**b**占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。
>
> 思路：回溯法的典型应用，设置一个**flag**标记是否走过，递归结束再置回即可，对于所遍历的目标串，若在递归中走到**'\0'**，那么直接返回**true**；在递归中标志位为**true**或不匹配或边界问题，都返回**flase**，上下左右递归完，标志位置回，并返回**false**。

```
class Solution {
public:
    /* 回溯法的典型例题 */
    bool hasPath(char* matrix, int rows, int cols, char* str){
        if(matrix == nullptr || rows <= 0 || cols <= 0 || str == nullptr)
            return false;  //参数错误

        bool *flag = new bool[rows * cols];
        memset(flag, false, rows * cols); //标记是否走过~

        for(int i = 0; i < rows; ++i){
            for(int j = 0; j < cols; ++j){
                //0是str起点
                if(hasPath(matrix, rows, cols, i, j, 0, str, flag))
                    return true;
            }
        }
        delete []flag;
        return false;
    }

bool
hasPath(char* matrix,int rows,int cols,int i,int j,int k, char* str, bool *flag)
{
```

```
        int index = i * cols + j;

        if(i < 0 || i >= rows || j < 0 || j >= cols
                 || matrix[index] != str[k] || flag[index])
            return false;

        if(str[k + 1] == '\0')
            return true;


        flag[index] = true;

        if(        hasPath(matrix, rows, cols, i + 1, j, k + 1, str, flag)
                || hasPath(matrix, rows, cols, i - 1, j, k + 1, str, flag)
                || hasPath(matrix, rows, cols, i, j + 1, k + 1, str, flag)
                || hasPath(matrix, rows, cols, i, j - 1, k + 1, str, flag)){
            return true;
        }

        flag[index] = false; // 不通过 恢复
        return false;
    }
};
```

15. 机器人的运动范围

key: 【key: [参数错误，路过标记，逐个比较，4个方向】

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

```
class Solution {
public:
    /* 和上题同样的思路，只不过这里我们的flag标志不用在回溯了，若回溯会出现很多重复的计数 */
    int movingCount(int dthreshold, int rows, int cols)
    {
        bool *flag = new bool[rows * cols];
        memset(flag, false, rows * cols);
        return CalCount(dthreshold, rows, cols, flag, 0, 0);
    }

    int CalCount(int dthreshold, int rows, int cols, bool *flag, int i, int j)
    {
        int index = i * cols + j;
        if(i < 0 || i >= rows || j < 0 || j > cols || flag[index] ||
          GetSum(i) + GetSum(j) > dthreshold)
            return 0;

        flag[index] = true;
        /* 所有情况求和 */
        return 1 + (CalCount(dthreshold, rows, cols, flag, i + 1, j)
                  + CalCount(dthreshold, rows, cols, flag, i, j + 1)
```

```
                    + CalCount(dthreshold, rows, cols, flag, i - 1, j)
                    + CalCount(dthreshold, rows, cols, flag, i, j - 1));
    }


    int GetSum(int num){
        int res = 0;
        while(num){
            res += num % 10;
            num /= 10;
        }
        return res;
    }
};
```

16. 二进制中1的个数

key: 【n & (n - 1)，最后一位1变成0，并更新n】

> 题目描述
>
> 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。
>
> 思路：使用位运算，核心思路就是 根据 **n & (n - 1)** 把最后一位1变成0，并更新n，然后计数，直到 n 为0。

```
class Solution {
public:
     int NumberOf1(int n) {
         int count = 0;
         while (n){
             n = n & (n - 1); // 把最后一位1变成0
             ++count;
         }
         return count;
     }
};
```

17.数值的整数次方

key: 【A^B ; (B<0, 1/B)】

> 题目描述
>
> 给定一个**double**类型的浮点数**base**和**int**类型的整数**exponent**。求**base**的**exponent**次方。
>
> 保证**base**和**exponent**不同时为**0**
>
> 思路：使用快速幂算法，每轮循环基数都要自乘，指数为奇数，要给结果乘以基数，每轮循环指数右移1位，直到指数为0。

```
class Solution {
public:
    double Power(double base, int e) {
        double res = 1;
        for(int i = 0; i < abs(e); i ++){
            res *= base;
        }
        if(e < 0){
            res = 1 / res;
        }
        return res;
    }
};
```

18. 删除链表中重复的结点

key: [虚拟头，相邻相等去除，不等则preNode后移，curNode始终后移]

> 题目描述
>
> 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**
>
> 思路:
>
> 定义一个虚拟的头节点，让遍历的**pre**指针指向它，**pre**的**next**指向头节点，然后判断相同结点，**pre**指向相同结点之后的位置，若循环时不同，那么更新**pre**结点，最后返回虚拟结点的**next**指针。

```
class Solution {
public:
    /* 指针操作：需要一个虚拟头指针 */
    ListNode* deleteDuplication(ListNode* pHead)
    {
        if(pHead == nullptr)
            return nullptr;

        ListNode *dummyHead = new ListNode(0);//创建虚拟头节点，可能会删除头节点
        dummyHead->next = pHead;//保存头结点


        auto preNode = dummyHead;//preNode:虚拟头，无数据
        auto curNode = dummyHead->next;
        while(curNode)
        {
            if(curNode->next && curNode->val == curNode->next->val)
            {
                while(curNode->next && curNode->val == curNode->next->val){
                    curNode = curNode->next;//跳过重复
                }
                preNode->next = curNode->next; //curNode被删除了
            }
            else{
```

```
                preNode = curNode;
            }
            curNode = curNode->next;
        }

        return dummyHead->next;
    }
};
```

19.正则表达式匹配

key:【下一位*，匹配或（.）=.=》str+1或pattern+2，否则pattern+2；下一位非*，匹配或（.）=.=》str+1且pattern+1，否则false】

> 题目描述
>
> 请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（包含0次）。

> 思路：
>
> 若**模式**的下一个**字符为**`'*'`，：
>
> > 如果当前字符匹配，则**字符串向后移动一位**（`'*'`前面出现多次）或者**模式向后移动2位**（`'*'`前面出现1次）；
> >
> > 反之，如果当前**字符不匹配**，则**模式向后移动2位**（`'*'`前面出现0次）。
>
> 若**模式**的下一个字符**不是**`'*'`，
>
> > 如果当前**字符匹配**或者**模式为**`'.'`，则继续**匹配下一个字符**，**否则返回false**。

```
class Solution {
public:
    /* 分清楚情况 */
    bool match(char* str, char* pattern)
    {
        if(*str == '\0' && *pattern == '\0')//两者都为空 true
            return true;
        if(*str != '\0' && *pattern == '\0')//str不空 patten空
            return false;

        if(*(pattern + 1) != '*'){
            //匹配.下一位
            if(*str == *pattern || (*str != '\0' && *pattern == '.'))
                return match(str + 1, pattern + 1);
            else
                return false;
        }
        else {  //下一个字符为'*'
            if(*str == *pattern || (*str != '\0' && *pattern == '.'))
                return match(str, pattern + 2) //  .*代表0次时
                    || match(str + 1, pattern);// 跳过和匹配多个的情况
            else
                return match(str, pattern + 2);//字符不匹配，则模式向后移动2位
```

```
        }
    }
};
```

20.表示数值的字符串

key:【排除反例，顺序E；+；.】

> 题目描述
>
> 请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。
>
> 思路：排除掉所有错误情况

```
class Solution {
public:
    bool isNumeric(char* str)
    {
        bool hasM = false; //2次，e后+-或str[0]
        bool hasE = false; //E，1次，非末
        bool hasF = false; //小数点，1次，E后无.

        int size = strlen(str);
        for(int i = 0; i < size; ++i){
            if(str[i] == 'e' || str[i] == 'E'){
                if(i == size - 1 || hasE)
                    return false;
                hasE = true;
            }
            else if(str[i] == '+' || str[i] == '-'){
                // E或e之后正负号
                if(hasM && str[i - 1] != 'e' && str[i - 1] != 'E')
                    return false;
              // 首次，非首字符，E或e之后正负号
                if(!hasM && i != 0 && str[i - 1] != 'e' && str[i - 1] != 'E')
                    return false;
                hasM = true;
            }
            else if(str[i] == '.'){
                if(hasE || hasF) //E后无.
                    return false;
                 hasF = true;
            }
            else if(str[i] < '0' || str[i] > '9'){
                return false;
            }
        }
        return true;
    }

};
```

21.调整数组顺序使奇数位于偶数前面

key: [冒泡修改法，或空间换时间]

> 题目描述
>
> 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。
>
> 思路：
>
> 从后向前遍历，遇到**奇数在偶数后边**，就交换调整 **O(n^2)**

```
class Solution {
public:
    //冒泡修改法
    void reOrderArray(vector<int> &array) {
        int size = array.size();
        bool changed = true;
        for(int i = 0; i < size && changed;  ++i){
            changed  = false;
            for(int j = size - 1; j > i; --j){
              if((array[j-1]&1)==0 && (array[j]&1)){
                    swap(array[j], array[j - 1]);
                    changed = true;
                }
            }
        }
    }
    //以空间换时间
   void reOrderArray(vector<int> &array) {
        vector<int> OddArray,EvenArray;
        for(auto c:array){
            if(c&1)
                OddArray.push_back(c);
            else
                EvenArray.push_back(c);
        }
        array.erase(array.begin(),array.end());
        array.insert(array.end(),OddArray.begin(),OddArray.end());
        array.insert(array.end(),EvenArray.begin(),EvenArray.end());
    }
};
```

22.链表中倒数第k个结点

key: 【快慢指针，移动窗口法，快先N，慢再走，快到尾】

> 题目描述
>
> 输入一个链表，输出该链表中倒数第k个结点。
>
> 思路：
>
> 双指针，快慢指针法，一个指针走k步，一个指针从头开始，两个指针一起向后走，快指针到达链表末尾，慢指针所指位置就是倒数第k

```
class Solution {
public:
    /* 双指针 */
    ListNode* FindKthToTail(ListNode* pListHead, unsigned int k) {
        if(!pListHead)
            return nullptr;

        auto fast = pListHead;
        auto slow = pListHead;

        while(--k)
        {
            if(fast->next)
                fast = fast->next;
            else
                return nullptr;
        }//fast 指针先走完k步。
        while(fast->next)
        {
            slow = slow->next;
            fast = fast->next;
        }

        return slow;
    }
};
```

23.链表中环的入口结点

key: 【快慢指针:1.相遇有环，2.遇后++求环长len; 3.快先跑一圈N，慢再走，快慢同速，快到尾{a+b = (n-1)*len}】

题目描述

给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出**null**。

思路

确定链表中包含环。

判断环中有几个节点。

两个指针相遇的节点肯定是在环中。可以从这个节点出发，一边继续向前移动一边计数，当再次回到这个节点时，就可以得到环中节点数了。

第三步就是如何找到环的入口。（本题环中有四个节点）

我们还是用两个指针来解决这个问题。先定义两个指针**P1**（快指针）和**P2**（慢指针）指向链表的头节点。如果链表中的环有**n**个节点，则指针**P1**先在链表上向前移动**n**步，然后两个指针以相同的速度向前移动。当第一个指针指向环的入口节点时，第一个指针已经围绕着环走了一圈，又回到了入口节点。

```
class Solution {
public:
    /* 快慢指针跑，慢指针追上快指针，那么快指针当前继续走，慢指针从头走
     * 相遇结点处就是环的入口节点
     */
```

```
    ListNode* EntryNodeOfLoop(ListNode* pHead)
    {
        if(pHead == nullptr)
            return nullptr;

        auto pFast = pHead;
        auto pSlow = pHead;

        while(pFast && pFast->next)
        {
            pFast = pFast->next->next;
            pSlow = pSlow->next;
            if(pFast == pSlow)  //有环，pFast多跑n环
            {
                pSlow = pHead;  //慢回头
                while(pSlow != pFast)//依据：2(a+b) =a+b+(n-1)*(c+b) (len=c+b)
                {                    // (a+b) =(n-1)*(c+b)= n*len
                    pFast = pFast->next;
                    pSlow = pSlow->next;
                }
                return pFast;
            }
        }
        return nullptr;
    }

//hash表
    unordered_map<ListNode*,int> h;
    int flag = 1;
    for(auto p = head; p ; p = p -> next, id ++)
    {
        if(h[p] != 0){
            return p;
        }
        else
            h[p]  = flag;
    }
    return nullptr;
};
```

24.反转链表

key:【tmp=old->next，old-next=newHead，new=old，old=tmp】

> 题目描述
>
> 输入一个链表，反转链表后，输出新链表的表头。
>
> 思路：不断地交换头节点下一个结点与标记结点后面的结点的位置，直到链表末尾，交换完成
>
> 输入: **1->2->3->4->5->nullptr**
> 输出: **5->4->3->2->1->nullptr**

```
class Solution {
public:
```

```
    ListNode* ReverseList(ListNode* pHead) {
        ListNode* newHead = nullptr;
        while(pHead)
        {
            auto tempNode = pHead->next;
            pHead->next = newHead;

            newHead = pHead;
            pHead = tempNode;
        }
        return newHead;
    }
};
```

25.合并两个排序的链表

key:【逐个比，遇nullptr,接other】

> 题目描述
>
> 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**。
>
> 一般创建单链表，都会设一个**虚拟头结点**，也叫哨兵，因为这样每一个结点都有一个前驱结点。

```
class Solution {
public:
    ListNode* Merge(ListNode* pHead1, ListNode* pHead2)
    {
        ListNode *vhead = new ListNode(-1);
        ListNode *cur = vhead;
        while (pHead1 && pHead2) {
            if (pHead1->val <= pHead2->val) {
                cur->next = pHead1;
                pHead1 = pHead1->next;
            }
            else {
                cur->next = pHead2;
                pHead2 = pHead2->next;
            }
            cur = cur->next;
        }
        cur->next = pHead1 ? pHead1 : pHead2;
        return vhead->next;
    }
};
```

26.树的子结构

key:【1.判RootB在TreeA中；2.若在则递归左右子节点，B到达nullptr返回true,A为nullptr而B非nullptr则false】

> 题目描述

- 树的子结构：判断树**B**是树**A**的子结构

思路:

树操作一般用递归方便，树**A**和树**B**都需要递归

- 递归一：**树A递归判断各结点**与树**B的根节点**是否相等，若相等执行递归二
- 递归二：**结点相等，树同时递归判断左右子节点是否相等**，返回左子节点**&&**右子节点不相等返回**false**
- 递归一的终止条件：**A**到达**nullptr**；
- 递归二的终止条件：**B**到达**nullptr**返回**true**；**B**不为**nullptr**而**A**为**nullptr**返回**false**
- 特例：树**A**或**B**有一个为空

```
class Solution {
public:
    bool HasSubtree(TreeNode* pRoot1, TreeNode* pSubRoot2)
    {
        bool result = false;
        if(pRoot1  && pSubRoot2 )
        {
            if(pRoot1->val == pSubRoot2->val) //根一致，递归子树
                result = isSubtree(pRoot1, pSubRoot2);

            if(!result) //根不同，递归左子树
                result = HasSubtree(pRoot1->left, pSubRoot2);

            if(!result) //左子树不同，递归右子树
                result =HasSubtree(pRoot1->right, pSubRoot2);
        }
        return result;
    }

    bool isSubtree(TreeNode* pRoot1, TreeNode* pSubRoot2)
    {
        if(pSubRoot2 == nullptr) //若匹配到的子树nullptr
            return true;

        if(pRoot1 == nullptr)
            return false;

        if(pRoot1->val != pSubRoot2->val)
            return false;

        return isSubtree(pRoot1->left, pSubRoot2->left)
            && isSubtree(pRoot1->right, pSubRoot2->right);
    }
};
```

27.二叉树的镜像

key【root非nullptrr则交换左右】

题目描述

操作给定的二叉树，将其变换为源二叉树的镜像，左右子树互换。

```
class Solution {
public:
    //pRoot 不需要交换
    void Mirror(TreeNode *pRoot) {
        if(pRoot == nullptr)
            return;
        //交换节点
        swap(pRoot -> left, pRoot -> right);
        //分别完成 左右树
        Mirror(pRoot->left);
        Mirror(pRoot->right);
    }
};
```

28.对称的二叉树

key:【根同nullptr => true;单nullptr=>false；不同value=>false;再递归左右】

> 题目描述
>
> 请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。
>
> 思路
>
> 同时相等（**nullptr** ）是对称，相同分别递归左右树，不等**false**。

```
class Solution {
public:
    bool isSymmetrical(TreeNode* pRoot)
    {
        return isSymmetrical(pRoot, pRoot);
    }
    bool isSymmetrical(TreeNode* pRoot1, TreeNode* pRoot2)
    {
        if(pRoot1 == nullptr && pRoot2 == nullptr)
            return true;
        if(pRoot1 == nullptr || pRoot2 == nullptr)
            return false;

        if(pRoot1->val != pRoot2->val)
            return false;

        return isSymmetrical(pRoot1->left, pRoot2->right)
            && isSymmetrical(pRoot1->right, pRoot2->left);
    }
};
```

29.顺时针打印矩阵

key:【while(up<=down && left<=right){左=》右，上=》下（up+1）；（判断是否最后一行/列 右=》左(right+1)；下到上(down - 1,> up)）}】

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

```
class Solution
{
  public:
    vector<int> printMatrix(vector<vector<int>> &matrix)
    {
        vector<int> output;
        //矩阵为空，逻辑或判断前半部分为真则不执行后半部分
        if (matrix.empty() || matrix[0].empty())
            return output;

        int up = 0, down = matrix.size() - 1;
        int left = 0, right = matrix[0].size() - 1;

        output.reserve((down + 1) * (right + 1));//设置output大小
        while (up <= down && left <= right)
        {
            for (int i = left; i <= right; ++i)//左到右
                output.push_back(matrix[up][i]);
            for (int i = up + 1; i <= down; ++i)//上到下，去除一个上
                output.push_back(matrix[i][right]);
            if (up != down) //考虑最后一圈是一行
                for (int i = right - 1; i >= left; --i)//右到左，去除一个右
                    output.push_back(matrix[down][i]);
            if (left != right) //考虑最后一圈是一列
                for (int i = down - 1; i > up; --i)//下到上，去除上和下
                    output.push_back(matrix[i][left]);
            ++ up;
            -- down;
            ++ left;
            -- right;
        }
        return output;
    }
};
```

30.包含min函数的栈

key:【每次将value都push进stackVal；同时将Min.top与value比较push进stackMin 。同时pop出 stackVal和stackMin】

> 题目描述
>
> 定义栈的数据结构，得到栈中所含最小元素的`min`函数（时间复杂度应为`O（1）`）。

```
class Solution {
public:
    void push(int value) {
        stackVal.push(value);    //每次stackVal都push value
        int minValue = value;
        if(!stackMin.empty()){
            value = std::min(stackMin.top(),value);//存真实更小
        }
        stackMin.push(minValue);
    }
    void pop() {
        if(!stackVal.empty()){
            stackVal.pop();//弹出后还是stackMin
            stackMin.pop();
        }
    }
    int top() {
        return stackVal.top();
    }
    int min() {
        return stackMin.top();
    }
private:
    stack<int> stackVal;
    stack<int> stackMin;
};
```

31.栈的压入、弹出序列

key: 【】

> 题目描述
>
> - 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）

```
class Solution {
public:
    bool IsPopOrder(vector<int> pushV,vector<int> popV) {
        int size = pushV.size();
        int i = 0;
        int index = 0;
        stack<int> stack;

        while(i < size){
            stack.push(pushV[i++]);//每次压栈
            //出栈时 比较top元素和当前popV
            while(!stack.empty() && popV[index] == stack.top() && index++ <
popV.size()){
                stack.pop();
            }
        }
        return stack.empty();
```

```
    }
};
```

32.从上往下打印二叉树

key:【队列，取头，push队头进结果，push左右分支至队列】

> 题目描述
>
> - 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
>
> 思路
>
> - 一层层遍历相当于广度优先搜索，用队列
> - 特例：输入结点为空

```
class Solution {
public:
    vector<int> PrintFromTopToBottom(TreeNode* root) {
        vector<int> res;
        if(!root) return res;
        //非空
        queue<TreeNode*> q;
        q.push(root);

        //bfs 广度优先遍历
        while(q.size())
        {
            auto t = q.front();
            q.pop();
            res.push_back(t -> val);

            if(t -> left)
                q.push(t -> left);
            if(t -> right)
                q.push(t -> right);
        }

        return res;
    }
};
```

33.把二叉树打印成多行

key:【】

> 题目描述
>
> - 从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

```
class Solution {
public:
        vector<vector<int> > Print(TreeNode* root) {
            vector<vector<int> > res;
```

```
            if(!root) return res;

            queue<TreeNode*> q;
            q.push(root);
            q.push(nullptr);

            vector<int> layer;
            while(q.size())
            {
                auto t = q.front();
                q.pop();
                //读到nullptr表示分行 把layer push到 res
                if(!t)
                {
                    if(layer.empty())
                        break;
                    res.push_back(layer);
                    layer.clear();
                    q.push(nullptr);
                    continue;
                }
                layer.push_back(t -> val);
                if(t -> left)
                    q.push(t -> left);
                if(t -> right)
                    q.push(t -> right);
            }

            return res;
        }

};
```

34.按之字形顺序打印二叉树

key:

> 题目描述
>
> - 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。

```
class Solution {
public:
    vector<vector<int> > Print(TreeNode* root) {
        vector<vector<int> > res;
        if(!root) return res;
        queue<TreeNode*> q;
        q.push(root);
        q.push(nullptr);

        vector<int> layer;

        bool zigzag = false;
        while(q.size())
        {
```

```
            auto t = q.front();
            q.pop();
            if(!t)
            {
                if(layer.empty()) break;
                if(zigzag)
                    reverse(layer.begin(), layer.end());
                res.push_back(layer);
                layer.clear();
                zigzag = !zigzag;
                q.push(nullptr);
                continue;
            }
            layer.push_back(t -> val);
            if(t -> left)
                q.push(t -> left);
            if(t -> right)
                q.push(t -> right);
        }
        return res;
    }

};
```

```
struct TreeNode{
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int value):val(value),left(nullptr),right(nullptr){}
};
void LevelOrder(TreeNode *T)
{
    std::queue<TreeNode*> tmpQueue;

    if (T == nullptr){
        return;
    }
    tmpQueue.push(T);


    TreeNode *tmpNode = new TreeNode(-1);
    while (!tmpQueue.empty())
    {
        tmpNode = tmpQueue.front();
        cout << tmpNode->val<<" ";
        tmpQueue.pop();
        if (tmpNode->left){
            tmpQueue.push(tmpNode->left);
        }
        if (tmpNode->right){
            tmpQueue.push(tmpNode->right);
        }
    }
}
```

35.二叉搜索树的后序遍历序列

key:

> 题目描述
>
> - 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出**Yes**,否则输出**No**。假设输入的数组的任意两个数字都互不相同。

```
//左右根
//数组最后一个数字就是根节点/父亲结点
//左右子树分界点   与根节点相比较
class Solution {
public:
    bool VerifySquenceOfBST(vector<int> sequence) {
        if(sequence.empty())
            return false;
        return VerifySquenceOfBST(sequence, 0, sequence.size() - 1);
    }
    /* 后序遍历的最后一个是根，那么二叉搜索树左子树都小于根，右子树都大于根 */
    bool VerifySquenceOfBST(vector<int> sequence, int begin, int end) {
        if(begin >= end)
            return true;

        int root = sequence[end];
        int splitPos = begin;
        /* 在左子树中找到大于根的值，然后去右子树 */
        while(splitPos < end && sequence[splitPos] < root)
            splitPos ++;

        /* 来到右子树，若遍历到小于根的，那么直接返回false */
        for(int j = splitPos; j < end; ++j){
            if(sequence[j] < root){
                return false;
            }
        }

        return VerifySquenceOfBST(sequence, begin, splitPos - 1)
            && VerifySquenceOfBST(sequence, begin + splitPos, end - 1);
    }
};
```

36.二叉树中和为某一值的路径

key:

> 题目描述
>
> - 输入一颗二叉树的根节点和一个整数，打印出二叉树中结点值的和为输入整数的**所有路径**。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。(注意: 在返回值的**list**中，数组长度大的数组靠前)

```
class Solution {
public:
    vector<vector<int> > result;//所有路径
```

```
    vector<int> tmp;
    vector<vector<int> > FindPath(TreeNode* root,int sum) {
        dfs(root, sum);
        return result;
    }

    /* 回溯法，遇到结点压入vector中，把期望值减去该值，那么到叶子结点，判断期望值和叶子结点是否相同，若相同，那么匹配成功 */
    void dfs(TreeNode* root,int sum){
        if(root == nullptr)
            return ;

        tmp.push_back(root->val);

        if(sum == root->val && root->left == nullptr && root->right == nullptr){
            result.push_back(tmp);
        }//找到，把path存起来。

        sum -= root->val;

        /* 递归左子树与右子树 */
        dfs(root->left, sum);
        dfs(root->right, sum);

        /* 回溯时删除最后一个 */
        if(!tmp.empty())
            tmp.pop_back();

    }
};
```

37.复杂链表的复制

key:

> 题目描述
>
> 输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的**head**。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

```
class Solution {
public:
    /* 分三步实现：
     * 第一步：复制结点到原结点后
     * 第二步：复制random，指向为原结点random的next
     * 第三步：分离链表，奇结点是原链表，偶结点是复制的链表
     */
    RandomListNode* Clone(RandomListNode* pHead)
    {
        auto curNode = pHead;
        while(curNode)
        {
            RandomListNode* newNode = new RandomListNode(curNode->label);
            newNode->next = curNode->next;
            curNode->next = newNode;
```

```
            curNode = newNode->next;
        }

        curNode = pHead;
        while(curNode)
        {
            if(curNode->random)
                curNode->next->random = curNode->random->next;
            curNode = curNode->next->next;
        }


        auto pCloneHead = new RandomListNode(-1);
        auto cloneNode = pCloneHead;
        curNode = pHead;
        while(curNode)
        {
            cloneNode->next = curNode->next;
            cloneNode = cloneNode->next ;
            curNode->next = curNode->next->next;
            curNode = curNode->next;
        }
        return pCloneHead->next;
    }
};
```

38.二叉搜索树与双向链表

key:

> 题目描述
>
> 输入一棵二叉搜索树，将该**二叉搜索树转换**成一个**排序的双向链表**。不能创建新的结点，仅可调整树中结点指针的指向。

```
class Solution {
public:
    TreeNode* Convert(TreeNode* pRootOfTree){
        if(pRootOfTree == nullptr)
            return nullptr;
        TreeNode* node = nullptr;
        Convert(pRootOfTree, node);
        while(node  && node->left){
            node = node->left;
        }
        return node;
    }
    /* 中序遍历，遍历到一个结点修改指向 */
    void Convert(TreeNode* curNode, TreeNode* &lastNodeInList){
        if(curNode == nullptr)
            return;
        //左 <- 根 -> 右
        //
        if(curNode->left)//左分支,最右值
            Convert(curNode->left, lastNodeInList);
```

```
        curNode->left = lastNodeInList;//节点分割，左 <- 根

        if(lastNodeInList)//左分支，下一个肯定是根。
            lastNodeInList->right = curNode;
        lastNodeInList = curNode;//移动一位，

        if(curNode->right)//右分支
            Convert(curNode->right, lastNodeInList);
    }
};
```

39.序列化二叉树

key:

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

```
class Solution {
public:
    /* 解析和泛解析过程，自己定义好规则即可 */
    void Serialize2(TreeNode *root, string &str)
    {
        if(root == nullptr){
            str.push_back('#'),
            str.push_back(',');
            return;
        }

        str += to_string(root->val);  //根
        str.push_back(',');
        Serialize2(root->left, str);  //左树
        Serialize2(root->right, str); //右树
    }

    char* Serialize(TreeNode *root) {
       if(root == nullptr)
           return nullptr;
        string str = "";
        Serialize2(root, str);
        char *result = new char[str.length() + 1];
        strcpy(result, str.c_str());
        return result;
    }

    TreeNode* Deserialize2(string &str) {
        if(str.empty())
```

```
            return nullptr;
        if(str[0] == '#')
        {
            str = str.substr(2);//移除  '#'+','
            return nullptr;
        }
        TreeNode* node = new TreeNode(stoi(str));
        str = str.substr(str.find_first_of(',') + 1);
        node->left = Deserialize2(str);
        node->right = Deserialize2(str);
        return node;
    }
    TreeNode* Deserialize(char *s) {
        if(s == nullptr)
            return nullptr;
        string str(s);
        return Deserialize2(str);
    }
};
```

40.字符串的排列

key:

> 题目描述
>
> 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**。
>
> 输入描述:
>
> 输入一个字符串,长度不超过9(可能有字符重复),字符只包括大小写字母。

```
class Solution {
public:

    vector<string> result;

    /* 排列树的典型应用 */
    vector<string> Permutation(string str) {
        int length = str.length();
        if(length == 0)
            return result;
        Permutation(0, length, str);
        sort(result.begin(), result.end());//result 重排序
        return result;
    }

    void Permutation(int i, int length, string str) {
        if(i == length){
            result.push_back(str);
        }
        else{
            for(int j = i; j < length; ++j){
                if(j != i && str[j] == str[i])
                    continue;
```

```
                swap(str[i], str[j]);
                Permutation(i + 1, length, str);
                swap(str[i], str[j]);
            }
        }
    }
};
```

41.数组中出现次数超过一半的数字

key:

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

```
class Solution {
public:
    /* 可以使用两种方法，part维持一个次数，下个数与之前数字相同
     * 则次数加一，否则次数减一，次数减为零，那么保存新数字，并把次数置为1，
     * 所求数字，一定是最后一次保存的数字*/
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        int val, times = 1;
        for(auto x : numbers){
            if(val == x)
                times ++;
            else{
                times --;
                if(times == 0){
                    val = x;
                    times = 1;
                }
            }
        }
        times = 0;
        for(auto x : numbers){
            if(x == val)
                times ++;
        }
        if(times * 2 > numbers.size())
            return val;
        return 0;
    }
};
```

42.最小的K个数

key:

> 题目描述
>
> 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,。

```
sort(input.begin(),input.end());//1.暴力解答，不推荐
for(int i = 0; i < k; i ++)
 res.push_back(input[i]);
//2.大根堆
//3.partition函数，基于快排递归
//4.multiset
```

```
class Solution {
public:
    /*
     * 两种方法：第一种是基于partition函数的方法，使得比第K个数字小的都位于其左边。
     * 第二种方法不用修改输入的数组，可以使用multiset完成，容器中元素个数小于k
     * 加入，大于k，和最大的比较，小于最大的，那么就删除最大的，把它加入
     */
    vector<int> GetLeastNumbers_Solution(vector<int> input, int k) {
        vector<int> res;
        int size = input.size();
        if(size == 0 || k > size)
            return res;

        multiset<int, greater<int>> set;//带排序的set，大的在前面
        for(int val : input){
            if(set.size() < k){
                set.insert(val);
            }
            else{  //已经有k个元素的set
                if(val < *(set.begin())){
                    set.erase(set.begin());
                    set.insert(val);
                }
            }
        }

        for(int val : set)
            res.push_back(val);

        return res;
    }
};
```

43.数据流中的中位数

key:

> 题目描述
>
> 如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值**排序之后位于中间的数值**。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。我们使用`Insert()`方法读取数据流，使用`GetMedian()`方法获取当前读取数据的中位数。

```
class Solution {
public:
    vector<int> numbers;
    void Insert(int num){
```

```
        /* 返回大于num的第一个元素位置 */
        auto pos = upper_bound(numbers.begin(), numbers.end(), num);
        numbers.insert(pos, num);//有序插入
    }

    double GetMedian(){
        int size = numbers.size();
        if(size == 0)
            return 0.0;
        if(size % 2 != 0)
            return numbers[size / 2];
        return (numbers[size / 2] + numbers[size / 2 - 1]) / 2.0;
    }

};
```

44. 连续子数组的最大和

key:

> 题目描述
>
> 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)。(子向量的长度至少是1)

```
class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array) {
        int result = INT_MIN;
        int sum = 0;
        for(auto a : array){
            if(sum < 0){
                sum = 0 ;//舍去前面累计结果，负数累加只会让sum更小。
            }
            sum += a;
            result = max(sum,result);
        }
        return result;
    }
};
```

45.1~n整数中1出现的次数

key:

> 题目描述
>
> 求出1~13的整数中1出现的次数,并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、12、13因此共出现6次。

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

```
            int num = i;//不能直接修改i的值
            while(num){
                if(num % 10 == 1){//末位值
                    ++sum;
                }
                num /= 10;
            }
        }
        return sum;
    }
};
```

46.把数组排成最小的数

key:

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

```
class Solution {
public:
    static bool cmp(int a, int b){
        string as = to_string(a);
        string bs = to_string(b);
        return as + bs < bs + as; //字典序
    }

    string PrintMinNumber(vector<int> nums) {

        sort(nums.begin(), nums.end(),cmp); //自定义字典序比较
        string res;
        for(auto x : nums) {
            res += to_string(x);
        }
        return res;
    }
};
```

47.丑数

key:

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

```
class Solution {
public:
    /* 两种方法：
     * 第一种：直观暴力法，  如果一个数能被2整除，那么连续除2，如果能被3整除，那么连续除3，
     * 如果能被5整除，那么连续除5，最后结果为1，那么是丑数
```

```
     * 第二种：避免对非丑数的计算，以空间换时间的思想提高算法效率，如果数组中存在排序的丑数，那么
     * 下一个数他们中其乘2、乘3、乘5中的最小值
     */
    int GetUglyNumber_Solution(int index) {
        vector<int> vec;
        vec.push_back(1);

        int index2 = 0;
        int index3 = 0;
        int index5 = 0;

        for(int i = 1; i < index; ++i)
        {
            int val = min(min(vec[index2] * 2, vec[index3] * 3), vec[index5] * 5);
            vec.push_back(val);

            /* 不能使用else if，有可能出现最小值相同的情况  3*2  2*3 */
            if(val == vec[index2] * 2) ++index2;
            if(val == vec[index3] * 3) ++index3;
            if(val == vec[index5] * 5) ++index5;
        }

        return vec[index - 1];
    }

};
```

48.第一个只出现一次的字符位置

key:

> 题目描述
>
> 在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置, 如果没有则返回 -1（需要区分大小写）。

```
class Solution {
public:
    int FirstNotRepeatingChar(string str) {
        //针对第一次出现的字符/数字
        unordered_map<char,int> map;
        for(auto c : str){
            map[c] ++;
        }

        int res = -1;
        int index = 0;
        for(auto c : str){
            if(map[c] == 1){
                res = index;
                break;
            }
            index ++;
        }
```

```
        return res;
    }
};
```

49.字符流中第一个不重复的字符

key:

> 题目描述
>
> 请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"，如果当前字符流没有存在出现一次的字符，返回#字符。

```
class Solution
{
public:
    int hashMap[256];
    deque<char> deque;
    //Insert one char from stringstream
    void Insert(char ch)
    {
        ++hashMap[ch];
        if(hashMap[ch] == 1)
            deque.push_back(ch);
    }
  //return the first appearence once char in current stringstream
    char FirstAppearingOnce()
    {
        while(!deque.empty() && hashMap[deque.front()] > 1)
            deque.pop_front();
        if(deque.empty())
            return '#';
        return deque.front();
    }
};
```

50.数组中的逆序对

key:

> 题目描述
>
> 在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P。并将P对1000000007取模的结果输出。 即输出P%1000000007
>
> 输入描述:
>
> 题目保证输入的数组中没有的相同的数字数据范围： 对于%50的数据,size<=10^4 对于%75的数据,size<=10^5 对于%100的数据,size<=2*10^5

```
class Solution {
```

```
public:
    int countNum = 0;

    void MergeSort(vector<int>& vec, int gap)
    {
        int len = vec.size();
        vector<int> brr(len);

        int start1 = 0;
        int end1 = start1 + gap - 1;
        int start2 = end1 + 1;
        int end2 = start2 + gap - 1 > len - 1 ? len - 1 : start2 + gap - 1;

        int i = 0;
        while (start2 < len)
        {
            while (start1 <= end1 && start2 <= end2)
            {
                if (vec[start1] > vec[start2])//归并时候出现，逆序
                {
                    brr[i++] = vec[start2++];
                    countNum += end1 - start1 + 1; //则前部分，全为逆序
                    if (countNum >= 1000000007) {
                        countNum %= 1000000007;
                    }
                }
                else
                {
                    brr[i++] = vec[start1++];
                }
            }
            while (start1 <= end1)
            {
                brr[i++] = vec[start1++];
            }

            while (start2 <= end2)
            {
                brr[i++] = vec[start2++];
            }

            start1 = end2 + 1;
            end1 = start1 + gap - 1;
            start2 = end1 + 1;
            end2 = start2 + gap - 1 > len - 1 ? len - 1 : start2 + gap - 1;
        }

        while (start1 < len)
        {
            brr[i++] = vec[start1++];
        }

        for (int i = 0; i < len; i++)
        {
            vec[i] = brr[i];
        }
    }
```

```
    int InversePairs(vector<int> data) {
        int len = data.size();
        if(len == 0)
            return -1;
        for (int gap = 1; gap < len; gap *= 2)
        {
            MergeSort(data, gap);
        }

        return countNum % 1000000007;
    }

};
```

51.两个链表的第一个公共结点

key:

> 题目描述
>
> 输入两个链表，找出它们的第一个公共结点。
>
> 思路:
>
> 如果有公共结点肯定是在后面重叠，且后面部分都是共同的。
> 方法1：先计算出两个链表的长度，可以让比较长的先走两个链表长度之差的步数，两个再一起走。
> 方法2：不同部分为a， 和b，公共部分为c；a + c + b = b + c + a;让两个一起走，a走到头就转向b， b走到头转向a，则在公共部分相遇。

```
class Solution {
public:
    ListNode *findFirstCommonNode(ListNode *headA, ListNode *headB) {
        auto p = headA, q = headB;
        while(p != q) {
            if(p) p = p->next;
            else p = headB;
            if (q) q = q->next;
            else q = headA;
        }
        return p;
    }
};
```

52.数字在排序数组中出现的次数

key:

> 题目描述
>
> 统计一个数字在排序数组中出现的次数。

```
class Solution {
public:
    /* 二分查找 */
```

```
    int GetNumberOfK(vector<int> data ,int k) {
        int size = data.size();
        if(size == 0)
            return 0;
        int left = 0;
        int right = size - 1;
        int count = 0;
        while(left <= right)
        {
            int mid = (left + right) >> 1;
            if(data[mid] > k)
                right = mid - 1;
            else if(data[mid] < k)
                left = mid + 1;
            else{
                while(data[mid - 1] == data[mid])
                    --mid;

                while(data[mid++] == k)
                {
                    ++count;
                }
                break;
            }
        }
        return count;

    }
};
```

53.二叉搜索树的第k个结点

key:

> 题目描述
>
> 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。

```
class Solution {
public:
    TreeNode* res = nullptr;
    /* 根据中序遍历的有序规则，得到中序遍历的第k个值即可 */
    TreeNode* KthNode(TreeNode* root, int k)
    {
        dfs(root, k);
        return res;
    }

    void dfs(TreeNode* root = nullptr, int &k){//中序遍历 左根右
        if(!root)
            return;
        dfs(root -> left, k);//左
        k --;
        if(k == 0)
            res = root;
```

```
        if(k > 0)
            dfs(root -> right, k);//右
    }
};
```

54.二叉树的深度

key:

> 题目描述
>
> 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。

```
int TreeDepth(TreeNode* root)
{
    //递归
    //max（left, right） + 1
    if(!root)
        return 0;
    int left  = TreeDepth(root -> left);
    int right = TreeDepth(root -> right);
    return max(left,left) + 1;
}
```

55.平衡二叉树

key:

> 题目描述
>
> 输入一棵二叉树，判断该二叉树是否是平衡二叉树。

```
class Solution {
public:
    bool IsBalanced_Solution(TreeNode* pRoot) {
        if(pRoot == nullptr)
            return true;
        int left  = TreeDepth(root -> left);
        int right = TreeDepth(root -> right);
        if(abs(left - right) > 1)
            return false;
        return IsBalanced_Solution(pRoot->left)
                && IsBalanced_Solution(pRoot->right);
    }

    int TreeDepth(TreeNode* root) {
        //递归
        //max（left, right） + 1
        if(!root)
            return 0;
        int left  = TreeDepth(root -> left);
        int right = TreeDepth(root -> right);
        return max(left,left) + 1;
    }
```

```
};
```

56.数组中只出现一次的数字

key:

> 题目描述
>
> 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。

```
class Solution {
public:
    int findNumberAppearingOnce(vector<int>& nums) {
        int ans = 0;
        for (int i = 31; i >= 0; --i) {
            int cnt = 0;
            for (int x: nums) {
                if (x >> i & 1) {
                    cnt ++;
                }
            }
            if (cnt % 3 == 1) {
                ans = (ans * 2) + 1;
            }
            else {
                ans = ans * 2;
            }
        }
        return ans;
    }
};
```

57.和为S的两个数字

key:

> 题目描述
>
> 输入一个**递增排序**的数组和一个数字**S**，在数组中查找两个数，使得他们的和正好是**S**，如果有多对数字的和等于**S**，输出两个数的乘积最小的。

```
class Solution {
public:
    vector<int> FindNumbersWithSum(vector<int> array,int sum) { //二分

        vector<int> result;
        int length = array.size();
        int start = 0;
        int end = length - 1;
        while (start < end)
        {
            if (array[start] + array[end] == sum){
                result.push_back(array[start]);
                result.push_back(array[end]);
```

```
                break;
            }
            else if (array[start] + array[end] < sum)
                start++;
            else
                end--;
        }
        return result;
    }
};
```

58.和为S的连续正数序列

key:

> 题目描述
>
> 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

```
class Solution {
public:
    vector<vector<int> > findContinuousSequence(int sum) {
        vector<vector<int>> res;
        vector<int> path;
        for(int i = 1, j = 2; j < sum && i < j; j) {
            int ans = (i + j) * (j - i + 1) / 2; // 等差数列
            if ( ans == sum){//如果相同就加入。
                int k = i;
                while(k <= j)
                    path.push_back(k++);
                res.push_back(path);
                path.clear();
                i ++, j ++;//两个指针同时往后移。
            }
            if ( ans < sum) {//如果比较小，j就往后移动。
                j ++;
            }
            if ( ans > sum) {//如果比较小，j就往后移动。
                i ++;//否则i往后移动
            }
        }
        return res;

    }
};
```

59.翻转单词顺序列

key:

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

```
class Solution {
public:
    string ReverseSentence(string str) {
        //翻转所有字符
        reverse(str.begin(), str.end());
        //翻转单个单词
        for(int i = 0; i < str.size(); i ++)
        {
            int j = i;
            while(j < str.size() && str[j] != ' ')
                j ++;
            //翻转单个单词
            reverse(str.begin() + i, str.begin() + j);
            i = j;
        }

        return str;
    }
};
```

60.左旋转字符串

key:

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

```
class Solution {
public:
    /* 本题采用旋转法，左半部分旋转，右半部分旋转，整个串旋转 */
    void Reverse(string &str, int i, int j){
        while(i < j){
            swap(str[i++], str[j--]);
        }
    }
    string LeftRotateString(string str, int n) {
        int length = str.length();
        if(length == 0)
            return string("");

        int firstB = 0;
        int firstE = n - 1;
        int secondB = n;
        int secondE = length - 1;
```

```
        Reverse(str, firstB, firstE); // first
        Reverse(str, secondB, secondE);//second
        Reverse(str, firstB, secondE);//all = first + second

        return str;
    }
};
```

61.滑动窗口的最大值

key:

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

```
class Solution {
public:
    int Max(const vector<int>& num, int begin, int end, int &pos)
    {
        int max = INT_MIN;
        for(int i = begin; i <= end; ++i)
        {
            if(max < num[i])
            {
                max = num[i];
                pos = i;
            }
        }
        return max;
    }
    vector<int> maxInWindows(const vector<int>& num, unsigned int size)
    {
        int numSize = num.size();
        vector<int> result;
        int maxPos = 0;
        if(numSize < size || size == 0)
            return result;

        int begin = 0;
        int end = size - 1;

        int maxNum = Max(num, begin, end, maxPos);
        result.push_back(maxNum);
        while(end < numSize - 1)
        {
            ++end;
            ++begin;
            if(num[end] >= maxNum)
            {
                maxNum = num[end];
```

```
                result.push_back(maxNum);
                maxPos = end;
            }
            else
            {
                if(end - maxPos < size)
                {
                    result.push_back(maxNum);
                }
                else
                {
                    maxNum = Max(num, begin, end, maxPos);
                    result.push_back(maxNum);
                }
            }
        }
        return result;
    }
};
```

62.扑克牌顺子

key:

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

```
class Solution {
public:
    /*
     * 排序 + 计数
     * 0的个数大于等于空缺个数，那么连续
     * 有对子，那么不连续
     */
    bool IsContinuous( vector<int> numbers ) {
        int size = numbers.size();
        if(size == 0)
            return false;

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

        int zeroNum = 0;
        for(int i = 0; i < size; ++i)
        {
            if(numbers[i] == 0)
                ++zeroNum;
            if(i && numbers[i]&& numbers[i] == numbers[i - 1])
                return false; //排除连续2次为0，情况下还有相同值，肯定不能成为顺子
        }
```

```
        int i = zeroNum;
        int j = i + 1;
        int numGap = 0;
        while(j < size)
        {
            numGap += numbers[j] - numbers[i] - 1;
            ++i;
            ++j;
        }
        return numGap <= zeroNum ? true : false;
    }
};
```

63.圆圈中最后剩下的数

思路：本题就是有名的约瑟夫环问题。我们可以环形列表来模拟，每次从这个列表中删除第m个元素，一直到列表最后剩下一个元素为止。

```
* 两种方法：
* 约瑟夫环的经典的问题：环形链表（使用STL list来模拟）
* 数学方法（推导复杂但实现简单）
//考虑用STL中std::list来模拟这个环形列表，由于list并不是一个环形的结构，因此每次跌代器扫描到列表末尾的时候，要记得把跌代器移到列表的头部。这样就是按照一个圆圈的顺序来遍历这个列表了。
//由于我们需要一个有n个结点的环形列表来模拟这个删除的过程，因此内存开销为O(n)。而且这种方法每删除一个数字需要m步运算，总共有n个数字，因此总的时间复杂度是O(mn)。
```

```
class Solution {
public:
    int LastRemaining_Solution(int n, int m){ //O(n*m)
        if(n < 1 || m < 1)
            return -1;

        list<int> list;
        for(int i = 0; i < n; ++i){
            list.push_back(i);
        }

        auto it = list.begin();
        while(list.size() != 1){
            for(int i = 1; i < m; ++i){
                ++it;
                if(it == list.end()){
                    it =  list.begin();
                }
            }
            auto del = it;
            /* 保存下一个 */
            list.erase(del);
            ++it;
            if(it == list.end()){
                it =  list.begin();
            }
        }
```

```
        return list.back();
    }
    int LastRemaining_Solution(int n, int m) // Solution:  O(n)
    {
         if(n < 1 || m < 1)
             return -1;
         int last = 0;
         for(int i = 2; i <= n; ++i)
            last = (last + m) % i;
         return last;
    }
};
```

64.求1+2+3+...+n

key:

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

```
class Solution {
public:
 /* 可使用构造函数求解，也可以用虚函数求解，很多解法 */
 int Sum_Solution(int n) {
     int res = n;
     n && (res += Sum_Solution(n - 1));
     return res;
 }
};
```

65.不用加减乘除做加法

key:

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

```
class Solution {
public:
    int add(int num1, int num2){
        while(num2!=0){
            int sum = num1 ^ num2;//不进位的加法
            int carry = (num1 & num2)<<1;//进位
            num1 = sum;
            num2 = carry;
        }
        return num1;
    }
};
```

66.构建乘积数组

key:

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

```
class Solution {
public:
    /* 根据特点，我们将新数组展开，可分为上三角和下三角 */
    vector<int> multiply(const vector<int>& A) {
        int size = A.size();
        vector<int> result(size);

        if(size == 0)
            return result;
        result[0] = 1;
        for(int i = 1; i < size; ++i)
        {
            result[i] = result[i - 1] * A[i - 1];
        }

        int temp = 1;
        for(int i = size - 2; i >= 0; --i)
        {
            temp *= A[i + 1];
            result[i] *= temp;
        }
        return result;
    }
};
```

67.把字符串转换成整数

key:

> 题目描述
>
> 将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0。

```
class Solution {
public:
    int StrToInt(string str) {
        int k = 0;
        long long num = 0;
        int sign = 1;
        if(str[k] == '-') {
            k ++, sign = -1;
        }
        if(str[k] == '+'){
            k ++, sign = 1;
```

```
        }
        while(k < str.size())
        {
            if(str[k] >= '0' && str[k] <= '9')
            {
                num  = num * 10 + str[k] - '0';
                k ++;
            }
            else{
                num = 0;
                break;
            }
        }
        num *= sign;
        if(num > INT_MAX)   num = INT_MAX;
        if(num < INT_MIN)   num = INT_MIN;
        return static_cast<int>(num);
    }
};
```