2017年11月6日 10:56

11/6日开始刷算法模块：

这个牛客网直通bat算法视频的夹子下有：
1， leetcode的题库和答案
2，直通bat源码和视频
3，电子书下有2016名企题库和答案
4，数据库复习资料
5，剑指offer pdf（内含问题和代码）

# 一，直通bat算法课

http://tool.oschina.net/highlight 在线调亮

练习：

这里用ubuntu下的vim写：hello.cpp

编译：g++ hello.cpp -o hello

运行：./hello

第二天早上要回顾下昨天学习的内容

备份：到时候整个内容写完，就用pycharm打开，用git备份至github

解题：没有思路时记得画图，将思路直观化

## 1，二叉树打印，序列化和反序列化(11.6-11.7)

### 题1：

有一颗二叉树，请设计一个算法，按层次打印这颗二叉树。给定二叉树的根节点root，请返回打印结果。结果按每一层一个数组进行存贮，所有数组的顺序按照层数从上往下，而且每一层的数组内元素按照从左往右排列，保证节点数小于等于500.

解题思路：

难点是知道何时该换行，这里我们只需要两个变量，last和nlast。
last：表示正在打印的当前行的最右节点；nlast：表示下一行的最右节点。
遍历到last就知道该换行了，换行后，只需要last=nlast就可以继续下一行的打印过程。
这里nlast只需持续更新记录宽度优先队列中最新加入的节点即可。
重复这个过程，直到所有的节点都打印完。

### 题2：

二叉树（内存中）的序列化（存入文件）和反序列化：给定一颗二叉树的头节点head，并已知二叉树节点值类型为32位整型，请设计一种二叉树序列化和反序列化的方案，并用代码实现。

解题思路：

1. 先序遍历：先根结点，然后遍历左子树，最后遍历右子树。
2.中序遍历：先遍历左子树，然后遍历根结点，最后遍历右子树。
3.后序遍历：先遍历左子树，然后遍历右子树，最后遍历根节点。

以先序为例，给出序列化解题过程：
1，假设序列化结果为str，初始化为空字符串，
2，先序遍历二叉树时如果遇到空节点，在str末尾加上”#!“(#表示空，！表示一个值得结束。若不标识，则不同的二叉树可能得到相同的序列化结果，容易混淆)
3，遇到不为空的节点，假设节点值为3，则在字符串末尾加上“3！”

以先序为例，给出反序列化解题过程：
1，先将序列化的字符串，转为以“！”切分的对应的value数组。
2，然后以先序遍历的过程构建二叉树，以 12 3 # # #为例。先根节点12，然后左子树3，左子树的左孩子#，即NULL，没有建立左子树的过程了，然后右孩子#，即NULL，然后根节点的右子树#，即NULL。

用什么序 序列化，就用什么序 反序列化，一棵树序列化的结果是唯一的，反序列化后的二叉树也是唯一的

code：见algorithm_bat下的1_Binary_tree _printing的TreePrinter.cpp

```
#include<iostream>
#include<cstdlib>
#include<vector>
#include<queue>
using namespace std;
vector<char>::size_type n =0;
//定义数据结构，树节点
struct TreeNode{
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x):val(x),left(NULL),right(NULL)
        {
    }
};

//二叉树的序列化过程
class Serialize
{
public:
    //传入：二叉树的头结点和序列化后的向量，向量是引用的方式传递（可以直接修改其本身的值）
    void serialize(TreeNode* head, vector<char> &B)
    {
        //如果头节点为空
        if(!head)
        {
            B.push_back('#');
            B.push_back('!');
            return;
        }
        int value = head->val;
        //如果头结点值为空
        if(!value)
        {
            B.push_back('0');
            B.push_back('!');
            //递归求左子树和右子树
            serialize(head->left,B);
            serialize(head->right,B);
            return;
        }
        //处理当前根节点的值value
        int a = 0;
        //将value的值按位存，从个位到高位
        vector<int> A;
        while(value)
        {
            a = value%10;
            A.push_back(a);
            value /=10;
        }
        //A非空的话,将A中元素从高位到个位push进B
        while(!A.empty())
        {
            a = A.back();
            A.pop_back();
            B.push_back('0'+a);
        }
        //根节点的值按位处理完之后，加!
        B.push_back('!');
        //递归处理根节点的左右子树
        serialize(head->left,B);
        serialize(head->right,B);
        return;
    }
};

//二叉树的反序列化过程
class DeSerialize
{
public:
    //传入序列化向量，和待重建的二叉树头节点，头节点是引用的方式传递（可以直接修改其本身的值）
    void deserialize(vector<char> A,TreeNode* &head)
    {
        int i, j, value = 0;
        //代码最开始初始化了n，为0. 如果A.size = 0或者其第0个元素为#（即空）
        if(n>A.size()-1 || A[n] == '#')
            //是一颗空的二叉树
            return;
        i = j = n;
        //统计！之前的这个字符有多少位
        while(A[j] != '!')
            j++;
        //在这个字符长度之内
        while(j>i)
```

```
        {
            //从高位到地位，将该字符的每一位存入value
            value = (A[i]-'0')+value*10;
            i++;
        }
        //分配头结点内存,并给头结点对应元素赋值
        head = (TreeNode *)malloc(sizeof(TreeNode));
        (*head).val = value;
        (*head).right = (*head).left = NULL;
        //将索引往后挪
        n = i+1;
        //递归
        deserialize(A,(*head).left);
        deserialize(A,(*head).right);
    }
};

class TreePrinter
{
public:
    //输入：二叉树的根节点（传值的方式），返回：二维向量
    vector<vector<int> > printTree(TreeNode* root)
    {
        //返回的二维向量
        vector<vector<int> > res;
        //临时向量
        vector<int> temp;
        //存放树节点的队列
        queue<TreeNode*> que;
        que.push(root);
        //当前行的最右节点，下一行的最右节点（即队列中最新加入的元素），当前指向的节点
        TreeNode *last = root,*nLast = NULL,*Now = NULL;
        //如果队列非空
        while(!que.empty())
        {
            //当前队头元素，要打印的，放入临时向量
            Now = que.front();
            cout<<Now->val;
            temp.push_back(Now->val);
            //如果当前节点有左孩子,让其左孩子入队，并将nlast指向这个最新加入队的元素
            if(Now->left)
            {
                que.push(Now->left);
                nLast = Now->left;
            }
            //如果当前节点有右孩子，
            if(Now->right)
            {
                que.push(Now->right);
                nLast = Now->right;
            }
            //如果当前节点就是当前行的最右节点
            if(Now == last)
            {
                //将临时向量里的这一行内存push入二维向量，并清空临时向量
                res.push_back(temp);
                temp.clear();
                //打印换行，并开始下一行的操作，让last=nlast
                cout<<endl;
                last = nLast;
            }
            //弹出当前队头元素
            que.pop();
        }
        return res;
    }
};

//主函数
int main()
{
    //定义序列化的二叉树转为的字符数组
    char a[11] = {'1','2','!','3','!','#','!','#','!','#','!'};
    //将字符数组转为序列化向量
    vector<char> sor(a,a+11);
    //反序列化：将序列化向量转为二叉树
    DeSerialize d;
    TreeNode* T = NULL;
    d.deserialize(sor,T);

    //序列化:将二叉树转为序列化向量
    Serialize c;
    vector<char> res;
    c.serialize(T,res);
    //cout<<'1';
    //打印序列化后的内容
    for(int i = 0; i !=res.size();i++)
        cout<<res[i]<<" ";
    cout<<endl;
```

```
    //宽度优先打印二叉树
    vector<vector<int> > res1;
    TreePrinter t;
    res1=t.printTree(T);
    for(vector<vector<int> >::iterator iter1=res1.begin();iter1!=res1.end();iter1++)
    {
            for(vector<int>::iterator iter2=(*iter1).begin();iter2!=(*iter1).end();iter2++)
                    cout<<*iter2<<" ";
            cout<<endl;
    }
    return 0;
}
```

练习题：

如果对于一个字符串A，将A的前面任意一部分挪到后边去形成的字符串称为A的旋转词，比如A=“12345”，A的旋转词有“12345”，“23451”等，对于两个字符串A和B，请判断A和B是否互为旋转词

给定两个字符串A，B以及他们的长度lena，lenb，请返回一个bool值，代表他们是否互为旋转词。

测试样例：” cdab “，4，“abcd”，4

code：

```
#include<iostream>
#include<string>
using namespace std;
class Rotation
{
public:
    bool chkRotation(string A,int lena, string B,int lenb)
    {
        string sum = A + A;
        string::size_type pos = sum.find(B);
        if(pos != string::npos)
            return true;
        else
            return false;
    }
};

int main()
{
    string a = "abcd";
    int lena = 4;
    string b = "cdab";
    int lenb = 4;
    Rotation r;
    bool bl = r.chkRotation(a,lena,b,lenb);
    cout<<bl<<endl;
}
```

## 2，排序(11.7-11.10)

## A,从时间复杂度的角度介绍九个排序算法

### A1，（O(N^2)）时间复杂度：

#### 题1：冒泡排序

解题思路：

1，先对（0，N-1）即数组里的全部元素进行排序，先第0，第1比较，大的放后面，然后第1,2比较，大的放后面。这样依次下来，最大的值被放到了数组的最后面

2，再对（0，N-2）的元素进行排序，重复1里面的过程，将倒数第二大的元素放在了数组的倒数第二位

3，重复以上过程，直至只剩一个元素，这样冒泡排序得到的数组元素是从小到大排的

#### 题2：选择排序：

解题思路：

1，先对（0，N-1）即数组里的全部元素进行排序，挑选出其中的最小值，放在位置0上

2，然后对（1，N-1）的元素进行排序，同样重复1的过程，将最小的元素放在1上。

3，直到最后这个范围只包含一个元素时，整个数组就有序了，从小到大。

#### 题3，插入排序：

解题思路：

1，先对位置0和1上的元素进行比较，如果后者更小，则将其放到位置0上。

2，然后对位置2上的数和它前一个位置上的数进行比较，如果后者更小，再将其与位置0上的值进行比较，如果还小，则将其放在位置0上。

3，接下来，对位置k上的数，也需要依次与之前的k-1个值进行比较，直到之前的元素小于等于该值，然后将其插在此处。

4，这样直到最后，这个数组变得有序了，从小到大。

## A2，（O(N*logN) )时间复杂度：

### 题1：归并排序

解题思路：

1，先让数组中的每一个数成为单独的有序区间，

2，然后合并相邻的有序区间，将其变为最大长度为2的有序区间（注意：这个区间是有序的（从小到大）

3，然后再合并相邻的有序区间，得到最大长度为4的有序区间。

4，依次这样进行下去，8-》16-》...直到让数组中的所有数合并成一个有序区间

### 题2：快速排序

解题思路：

1，随机的在数组中选一个数，小于它的，放在该数的左边，大于它的，则统一放在该数的右边。这个过程是怎么实现的呢，

a，首先，将划分值放在数组的最右位置；

b,然后设置一个小于等于区间，最开始的时候，其长度为0，放在数组的左边，

c,接下来，从左到右遍历所有元素，如果当前元素大于划分值，则略过继续，如果当前元素小于等于划分值，我们就把当前值和小于等于区间的下一个值进行交换，并将小于等于区间往右扩一个位置。

e,在遍历完所有元素，直到最后一个元素时，将划分值与小于等于区间的下一个元素交换。

2，接下来分别对左右两个部分，递归这个过程。

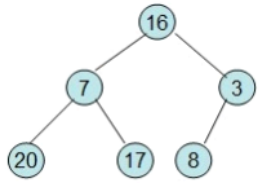
### 题3：堆排序

解题思路：

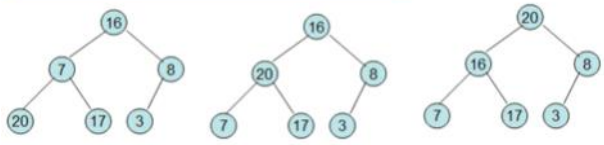
1，先将数组中的N个数，建成一个大小为N的大顶堆。我们知道堆顶是所有数中的最大值

a， 如：给定一个整形数组a[]={16,7,3,20,17,8}，对其进行堆排序。

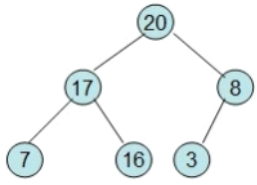
b，首先根据该数组元素构建一个完全二叉树，得到




c， 然后需要构造初始堆，则从最后一个非叶节点开始调整，调整过程如下：




d，20和16交换后导致16不满足堆的性质，因此需重新调整




e，这样就得到了初始堆。（即每次调整都是从父节点、左孩子节点、右孩子节点三者中选择最大者跟父节点进行交换(交换之后可能造成被交换的孩子节点不满足堆的性质，因此每次交换之后要重新对被交换的孩子节点进行调整)。有了初始堆之后就可以进行排序了。

2，把堆顶元素与堆的最后一个元素交换，然后把最大值脱离堆结构，放在数组最后的位置

3，再把N-1大小的堆进行调整，得到N-1大小的初始堆，同样的，将堆顶元素与堆得最后一个元素交换，然后同样的把最大值脱离堆结构，放在数组倒数第二的位置。

4，然后继续这样的过程，直至堆大小为1。然后将堆内元素放在数组的起始位置即可，即可得到一个从小到大的数组排列

### 题4：希尔排序

解题思路：

希尔排序是插入排序的改良，其步长是经过调整的，比如插入排序中，步长是1. 但希尔排序的步长却是从大到小调整的。

1, 6 5 3 1 8 7 2 4，这个序列，如果初始步长为3的话，6 5 3 作为前三个数是不需要调整的，

2，然后对于接下来的数1，往前跳三位，来到6这个数值，1和6进行比较，1比较小，因此1和6进行交换。此时1已经来到了位置0，再往前跳三步就越界了，因此调整停止-》1 5 3 6 8 7 2 4

2，接下里处理下一个数据，8，同样往前跳三位，来到5这个位置，8比较大，因此直接停止交换的过程，继续下一个数。

3，接下里的数据是7,7往前三位与3进行比较，同样比较大，停止交换的过程，继续下一个数

4，接下来是数据2,2需要交换-》1 5 3 2 8 7 6 4， 2接下来再往前跳三位，发现其需要与1比较，不交换

5，接下来是数据4，需要交换-》1 5 3 2 4 7 6 8， 4接下来再跳三位，发现需要交换-》1 4 3 2 5 7 6 8

6，进行到最后一位后，在进行步长为2，步长为1的调整，然后以步长为1的情况结束。

7，希尔排序的成败完全在于其步长的选择，如果最一开始，比较幸运的选择了合适的步长，很有可能第一轮就结束了。如果步长越恶劣，其时间复杂度越接近（O(N*logN)）

A3，（O(N)）时间复杂度趋近于这个：

这类算法有一个普遍的思想，其不是基于比较的排序模型， 而是基于桶排序

题1：计数排序

解题思路：

1，比如我们现在有三个员工，需要根据身高进行排序。李四：185，张三：167，王五：178

2，成年人的身高区间在（100,300）cm这个范围内。然后我们根据身高的厘米数来建立不同桶，比如身高为100，就建立100号桶，身高为300，就建立300号桶

3，然后我们将所有的员工按照他们对应的身高，放入不同桶中，

4，然后桶依次按从小到大的编号倒出员工，直到导出第300号桶的员工为止。然后员工倒出的顺序，就是其按身高排序的结果

题2，基数排序：

解题思路：

1，假设被排序的数都是10进制的，023,014,101,072,084,011

2，我们建立编号为0-9的桶，接下来我们按照每个数字的个位将其放入不同的桶中，

3，当所有元素都进桶之后，我们就按0-9的顺序依次倒出每号桶。

4，这样就组成一个序列，101,011,072,023,014，084然后我们按照这个序列的顺序，根据每个数字的十位数将其放入不同桶中

5，当所有元素都进桶之后，我们就按0-9的顺序依次倒出每号桶。

6，这样就组成一个序列，101,011,014,023,072,084.然后我们再按照这个序列的顺序，根据每个数字的百位数将其放入不同桶中

7，当所有元素都进桶之后，我们就按0-9的顺序依次倒出每号桶。最后一次倒出的序列，就是数组从小到大排序的序列了

## B，下面，我们在分析几个排序的空间复杂度：

B1，O（1）空间复杂度：

插入排序

冒泡排序

选择排序

堆排序

希尔排序

B2，O(log N) - O（N）空间复杂度:

快速排序

B3，O（N）空间复杂度：

归并排序

B4，O（M）空间复杂度：（M是桶的数量）

计数排序

基数排序

## C，最后，从稳定性的角度分析几个排序

什么是稳定性？在待排序的序列中，存在一些具有相同关键词的记录，如果经过排序，这些记录的相对顺序不变，称这种算法是稳定的，否则，是不稳定的。

C1，稳定的排序算法：

冒泡排序

插入排序

归并排序

计数排序

基数排序

插入排序

C2，不稳定的排序算法：

选择排序 ：如 2221, 1会先和第一个2交换，然后第一个2就变成了最后一个，这就破坏了2之间的相对位置。

快速排序：4,3,3,3.如随机选中了第二个三为随机数，那么排序完后，第二个三必定在最后一个位置，因此也破坏了其之间的相对位置。

希尔排序：5,1,1.步长为2的时候，1会与5交换，因此1之间的相对位置也发生了改变。

堆排序：如555，然后经过堆排序，变成了第三个五，第二个五，第一个五这样的顺序，也破坏了5之间的相对位置

## D，几点补充说明：

D1,排序算法无绝对优劣：

通常不能说那种排序算法好，这个要和排序的元素相关，比如对人的年龄和身高排序时，因为这种数据范围通常比较小，可以考虑用计数排序。

但对于均匀分布的整数，计数排序就不合适了。除非面试题特别说明，否则认为要排序的数据范围是均匀分布的。

D2，为什么叫快速排序：

快速排序之所以叫快速排序，并不代表其比堆排和归并排序优良。在最好的情况下，它的渐进复杂度和堆排与归并是相同的。只是快排的常量系数较小而已。

### D3，工程上的排序：

工程上的排序一般是综合排序，数组元素比较少时，会选择常量系数较低的插入排序。当数组元素较多时，则使用时间复杂度整体为（O(N*logN)）时间复杂度的快速排序或其他排序

### Code:九大排序算法

```
#include<iostream>
#include<cstdlib>
#include<math.h>
using namespace std;
#define N 10


//冒泡排序，时间:O(N^2),空间:O(1)，稳定
class BubbleSort
{
public:
    //传入指向数组的指针和数组长度
    void bubbleSort(int *A, int n)
    {
        int i = n,temp,j,flag;
        //从最开始的0-N-1，到0-N-2,直到0
        while(i)
        {
            //这里之所以是j<i-1,是因为后面有A[j+1]
            for(j = 0,flag=0;j<i-1;j++)
                //如果前面的数比后面的大，则换过去，一直换到最后
                if(A[j]>A[j+1])
                {
                    temp = A[j];
                    A[j] = A[j+1];
                    A[j+1] = temp;
                    flag = 1;
                }
            //flag为空，意味着整个数组已经不需要再排序了，直接break掉while即可。
            if(!flag)
                break;
            i--;
        }
    }
};

//选择排序，时间:O(N^2),空间:O(1),不稳定
class SelectionSort
{
public:
    //返回排序好的数组指针，传入指向待排序数组的指针和数组长度
    int * selectionSort(int *A, int n)
    {
        //选择排序的过程是首先从（0，N-1）中选出最小的，放在数组的最左边，然后从（1，N-1）中选出最小的，然后放在数组的左边第二个位置，依次下去
        int i = 0,j,min,temp;
        while(i < n-1)
        {
            //初始化最小值位置暂定为i，第0个元素
            min = i;
            //从第1个元素到最后一个元素，依次与第i（0）个元素比较,直到找到最小值所在的位置
            for( j = i+1; j < n; j++)
                if (A[j]<A[min])
                    min = j;
            //交换最小值和第i（0）个元素
            if(min != i)
            {
                temp = A[i];
                A[i] = A[min];
                A[min] = temp;
            }
            //继续从（1，N-1）的过程
            i++;
        }
        //返回排好序的数组
        return A;
    }
};

//插入排序，时间:O(N^2),空间:O(1)，稳定
class InsertSort
{
public:
    int* insertSort(int *A,int n)
    {
        int i = 1,j,temp;
```

```
        //首先先对位置0和1上的元素进行比较，如果后者更小，则将其放到位置0上，然后对位置2上的数和它前一个位置上的数进行比较，如果后者更小，再将其与位置0上的值进行比较，如果还小，则将其放在位置0上。接下来，对位置k上的数，也需要依次与之前的k-1个值进行比较，直到之前的元素小于等于该值，然后将其插在此处。
        while(i < n)
        {
            //现在要做的是，将A[i]处的元素与其之前的进行比较，如果小于之前的元素，则插入到合适的位置。
            temp = A[i];
            j = i -1;
            while(j>=0&&A[j]>temp)
            {
                //若元素大于A[i]处的元素，则将该处的元素右移
                A[j+1] = A[j];
                j--;
            }
            A[j+1]= temp;
            i++;
        }
        return A;
    }
};

//归并排序，时间:O(N*log N),空间:O(N)，稳定
class MergeSort
{
public:
    //归并排序
    int* mergeSort(int *A, int n)
    {
        //将A进行归并排序成新A
        Msort(A,A,0,n-1);
        return A;
    }
    //子排序,将A[s,...,t]归并排序为B[s,...,t]
    void Msort(int *A, int *B,int s,int t)
    {
        //数组长度为1
        if (s == t)
            B[s] = A[s];
        else
        {
            //数组中间位置
            int m = (s+t)/2;
            //空的数组
            int C[100] = {0};
            //递归：对A中的前半部分元素和空C进行排序，生成C
            Msort(A,C,s,m);
            //递归：排序A中的后半部分和排序后前半部分生成的C，生成总的排序好A中所有元素的C
            Msort(A,C,m+1,t);
            //合并C和空B，生成B
            Merge(C,B,s,m,t);
        }
    }

    //子归并,将有序的A[i,...,m]和A[m+1，...,n]归并为有序的B[i,...,n]
    void Merge(int *A, int *B,int i, int m, int n)
    {
        int j,k;
        for(j = m+1,k = i;i <=m &&j <= n;k++)
        {
            //判断相邻两个数组的第一个元素哪个小，小的放入B的起始位置，然后这个放入的数组索引右移一位
            if(A[i]<A[j])
                B[k] = A[i++];
            else
                B[k] = A[j++];
        }
        //处理剩余的最后一个元素，如果是前一个相邻数组的，用这个索引放入B，如果是另一个相邻数组的，则用另一个索引放入B
        if(i <=m)
            while(i <=m)
                B[k++]= A[i++];
        if(j <= n)
            while(j <=n)
                B[k++]= A[j++];
    }
};

//快速排序，时间复杂度O（N * log N),空间复杂度：O（log N）- O(N),不稳定
class QuickSort
{
public:
    int* quickSort(int *A,int n)
    {
        QSort(A,0,n-1);
        return A;
    }

    //对数组A[low,...,high]进行排序
```

```
    void QSort(int *A, int low, int high)
    {
        if(low < high)
        {
            //支点位置
            int n;
            //这一步之后，已将小于支点的放在了支点的左边，大于支点的放在了支点的右边
            n = Partition(A,low,high);
            //对支点左边的进行递归排序
            QSort(A,low,n-1);
            //对支点右边的部分进行递归排序
            QSort(A,n+1,high);
        }
    }
    //这里low，high是需要交换的数组的上下界。
    //交换数组A[low,...,high]的记录，支点记录到位，并返回其所在位置，此时在它之前（后）的记录均不大（小）于它
    int Partition(int *A,int low, int high)
    {
        //这里将A[low]处的值，作为随机选中的支点。
        int key = A[low];
        while(low < high)
        {
            //从右往左
            //如果下标大于上标，且下标的值大于支点的话，则下标位置的数不用变，下标往前挪一位。
            while(low < high && A[high]>=key)
                high--;
            //否则是小于支点值，故将此处的值换至上标处。
            A[low]= A[high];

            //从左往右
            //如果此时换到low处的值比key要小，ok，继续往前，上标往前挪一位
            while(low < high && A[low]<=key)
                low++;
            //否则low处的值大于支点的值的话，将此处的值换至下标处。
            A[high]=A[low];

        }
        //最后，low和high重合。需要交换的过程结束，最终将支点的值放至上标处。
        A[low] = key;
        //返回支点位置
        return low;
    }

};

//堆排序，时间复杂度O（N * log N),空间复杂度：O（1),不稳定
class HeapSort
{
public:
    int* heapSort(int *A,int n)
    {
        int i, temp;
        //从A中间位置，往左，构建大顶堆,循环结束时，其调整的是heapAdjust(A,0,n-1);
        for(i = n/2 -1;i>=0;--i)
            heapAdjust(A,i,n-1);
        //从A末尾位置开始
        for(i = n-1;i>0;i--)
        {
            //堆顶，最大值
            temp = A[0];
            //将A末尾的值放至堆顶
            A[0] = A[i];
            //将堆顶的值放至A的末尾
            A[i] = temp;
            //对重新设置过堆顶的堆进行大顶堆调整
            heapAdjust(A,0,i-1);
        }
        return A;
    }

    //调整至大顶堆。已知A[s,...,m]中记录的关键字除A[s]之外均满足堆的定义，本函数调整A[s]的关键字，使A[s,...,m]成为一个大顶堆.堆中索引是从0开始的。
    void heapAdjust(int *A,int s, int m)
    {
        //定义索引值和堆顶元素。
        int j, rc = A[s];
        for(j = 2*s+1;j<=m;j=2*j +1)
        {
            //这里判断左节点是否小于右节点，若是，则j++，直接判断右节点和根节点哪个大
            if(j < m && A[j]<A[j+1])
                j++;
            //判断根节点元素和右节点哪个大,如果堆顶元素大，则不用调整，直接break
            if(rc > A[j])
                break;
            //否则，右节点处值最大，换至堆顶
            A[s]= A[j];
```

```
            //将索引的位置换至j
            s = j;
        }
        //将原来索引处的位置换至原来的j处，即新的s索引处
        A[s]=rc;
    }
};

//希尔排序，时间复杂度O（N * log N),空间复杂度：O（1),不稳定
class ShellSort
{
public:
    int* shellSort(int *A, int n)
    {
        int gap,i,j,temp;
        if(n ==1)
            return A;
        //步长，从n/2开始，每次缩减一半
        for(gap = n/2;gap>0;gap /= 2)
        {
            //开始处理的位置，在步长大小之后开始
            i = gap;
            //从步长大小之后开始，每次进1，直到n。
            while(i < n)
            {
                //步长大小开始后的第一个值
                temp = A[i];
                //j为当前i往前跨gap大小的位置的索引
                j = i-gap;
                //如果距离大于0且j索引处的值，比i索引处的值大，则交换，将A[j]处的值放至j+步长gap之后的位置，也就是i的位置
                while(j >= 0&&A[j]>temp)
                {
                    A[j+gap]=A[j];
                    //继续往左
                    j = j -gap;
                }
                //将原来A[i]处的值，放至之前j的位置，不过while里面多减了一个gap，所以这里加上
                A[j+gap]=temp;
                i++;
            }
        }
        return A;
    }
};


//计数排序，时间复杂度O（N ),空间复杂度：O（M),稳定，只能处理小范围的int类型数据
class CountingSort
{
public:
    int* countingSort(int* A,int n)
    {
        //求数组A中的最大最小值
        int i,j,min,max;
        for (i =1,min = max=A[0];i<n;i++)
        {
            if(A[i]<=min)
                min = A[i];
            if(A[i]>max)
                max = A[i];
        }
        //定义桶空间，有为max-min+1个桶
        int *counts = (int *)calloc(max-min+1,sizeof(int));
        if(!counts)
            exit(-1);
        //对于A中的每个元素，将其放入对应的桶中，根据元素值减去最小值，来确定其该位于哪号桶。
        for(i = 0;i < n;i++)
            counts[A[i]-min]++;
        for(i = 0,j = 0;i<max-min+1;i++)
            //如果i号桶有数据
            while(counts[i])
            {
                //将其中数据：i+m，按顺序放入A中，从A[0]的位置开始放起
                A[j] = i + min;
                counts[i]--;
                j++;
            }
        free(counts);
        counts = NULL;
        return A;
    }
};
//基数排序，时间复杂度O（N ),空间复杂度：O（M),稳定
class RadixSort
{
```

```
public:
    //radix为关键词最高位数有几位。比如138,3位。
    int* radixSort(int* A,int n,int radix)
    {
        //定义桶,从0-9.N在代码的开头定义过了，是10.
        //其中temp[10][N],表示有10个桶，每个桶中可放N个数据。
        //order[x]表示其代表的x号桶内先已存入的数据个数。
        int temp[10][N]={0},order[10]={0};
        //pow(x,y):计算x的y次方。
        int m=(int)pow((double)10,radix-1);
        //int m = (int)pow((double)10,radix-1)
        int base=1;
        //1位及以上
        while(base <=m)
        {
            int i,k;
            for(i = 0;i <n;i++)
            {
                //依次对A中的每个数，除10得余数，然后根据个位放入桶
                int lsd = (A[i]/base)%10;
                //前面是几号桶，后面是该桶内的次序数组，初始化为0,此时，order[8]=0
                temp[lsd][order[lsd]]=A[i];
                //该桶的次序数组索引加一，现在order[8] = 1
                order[lsd]++;
            }
            //按桶的次序，取出数，放入A，从A[0]开始。
            for(i = 0,k = 0; i <10;i++)
            {
                //如果i号桶内有数据
                if(order[i])
                {
                    int j;
                    for(j = 0;j <order[i];j++,k++)
                        A[k] = temp[i][j];
                }
                //取完，将该号桶的次序数组清零。
                order[i]=0;
            }
            //base每次×10，那么下次就该是按十位来放了。直到base>m,即超高了最高位数对应的m
            base*=10;
        }
        return A;
    }
};

int main()
{
    //冒泡排序测试
    int arr[] = {2,4,7,9,8,6};
    BubbleSort b;
    b.bubbleSort(arr,6);
    cout<<"冒泡排序测试"<<":";
    for (int i = 0; i < 6; i++)
        cout<<arr[i]<<" ";
    cout<<endl;

    //选择排序测试
    SelectionSort s;
    s.selectionSort(arr,6);
    cout<<"选择排序测试"<<":";
    for (int i = 0; i < 6; i++)
        cout<<arr[i]<<" ";
    cout<<endl;

    //插入排序测试
    InsertSort i;
    i.insertSort(arr,6);
    cout<<"插入排序测试"<<":";
    for (int i = 0; i < 6; i++)
            cout<<arr[i]<<" ";
        cout<<endl;

    //归并排序测试
    int arr1[]={54,35,48,36,27,12,44,44,8,14,26,17,28};
    MergeSort m;
    m.mergeSort(arr1,13);
    cout<<"归并排序测试"<<":";
    for (int i = 0; i < 13; i++)
            cout<<arr1[i]<<" ";
        cout<<endl;

    //快速排序测试
    QuickSort q;
```

```
    q.quickSort(arr1,13);
    cout<<"快速排序测试"<<":";
    for (int i = 0; i < 13; i++)
                cout<<arr1[i]<<" ";
        cout<<endl;

    //堆排序测试
    HeapSort h;
    h.heapSort(arr1,13);
    cout<<"堆排序测试"<<":";
    for (int i = 0; i < 13; i++)
                cout<<arr1[i]<<" ";
        cout<<endl;

    //希尔排序测试
    ShellSort sh;
    sh.shellSort(arr1,13);
    cout<<"希尔排序测试"<<":";
    for (int i = 0; i < 13; i++)
                cout<<arr1[i]<<" ";
        cout<<endl;

    //计数排序测试
    CountingSort c;
    c.countingSort(arr1,13);
    cout<<"计数排序测试"<<":";
    for (int i = 0; i < 13; i++)
                cout<<arr1[i]<<" ";
        cout<<endl;

    //基数排序测试
    RadixSort r;
    r.radixSort(arr1,13,2);
    cout<<"基数排序测试"<<":";
    for (int i = 0; i < 13; i++)
                cout<<arr1[i]<<" ";
        cout<<endl;


    return 0;
}
```

## E，几道经典的笔试面试题：

E1：已知一个几乎有序的数组，几乎有序是指，如果把数组排好顺序的话，每个元素移动的距离不超过k，并且k相对于数组长度来说很小。请问选择什么方法对其排序比较好？

解题思路：

a, 首先时间复杂度O（N）的排序算法，其由于排序元素范围的问题，并不适合所有情况。因此这里我们不考虑计数和基数排序。

b, 然后是时间复杂度O（N^2）的排序算法，冒泡排序和选择排序，由于其排序不考虑原始数组的序列的，无论怎样其时间复杂度都是O（N^2），因此这里也不考虑

c, 接着时间复杂度O（N^2）的排序算法，插入排序，这个在本题效果是比较好的，因为其排序是考虑原始数据的序列的，并且不管怎么移动，这个移动范围是不超过k的。因此本题，如果使用插入排序的话，时间复杂度是O（N*k）

d, 然后是时间复杂度（O(N*logN)）的排序算法，快速排序，归并排序，其也不考虑原始数据的顺序。

e，这道题的最终答案是改进后的堆排序。排序之后每个元素的移动范围不超过k，因此整个数组的最小值，一定是在a[0]-a[k-1]这个区间的。

1，因此我们将a[0]-a[k-1]这个区间的元素，建立一个小根堆，然后堆顶元素就是这前k个值得最小值，弹出这个堆顶，放在数组的最左边。

2，接着，将a[k]这个元素，放入小顶堆的堆顶，然后对这个小顶堆进行调整， 然后整个数组第二小的元素就到了堆顶，然后弹出将其放在数组的第1个位置上。

3，接下来重复上述过程，每次调整都是在大小为k的堆里，所以成本为logk，然后我们要拍好n个元素，因此整个的时间复杂度为O（N*log k）

Code:

```
/*E1：已知一个几乎有序的数组，几乎有序是指，如果把数组排好顺序的话，每个元素移动的距离不超过k，并且k相对于数组长度来说很小。请问选择什么方法对其排序比较好？
解题思路：这道题的最终答案是改进后的堆排序。排序之后每个元素的移动范围不超过k，因此整个数组的最小值，一定是在a[0]-a[k-1]这个区间的。
        1，因此我们将a[0]-a[k-1]这个区间的元素，建立一个小顶堆，然后堆顶元素就是这前k个值得最小值，弹出这个堆顶，放在数组的最左边。
        2，接着，将a[k]这个元素，放入小根堆的堆顶，然后对这个小根堆进行调整， 然后整个数组第二小的元素就到了堆顶，然后弹出将其放在数组的第1个位置上。
        3，接下来重复上述过程，每次调整都是在大小为k的堆里，所以成本为logk，然后我们要拍好n个元素，因此整个的时间复杂度为O（N*log k）
*/
#include<iostream>
#include<vector>

using namespace std;

class SmallScaleSort
{
public:
    //传引用,这里k是每个元素最大移动的距离，这里我们设定为2
    vector<int> sortElement(vector<int> &A,int n, int k)
    {
        //创建一个容器B，其元素个数为n，且值均为0
        vector<int> B(n,0);
        int i,j;
        //将a[0]-a[k-1]这个区间的元素，建立一个小顶堆
```

```
        for (i = k/2-1;i >=0;--i)
            heapAdjust(A,i,k-1);
        //将小顶堆的元素放置数组B的初始位置
        B[0]=A[0];
        //将a[k]及往后的数据，依次放入小顶堆的堆顶，然后对这个小顶堆进行调整，然后生成新的小顶堆，获取其中第二小的元素
        for(i = 1,j = k;j<n;j++)
        {
            A[0] = A[j];
            heapAdjust(A,0,k-1);
            B[i++]= A[0];
        }
        //对A的前k个元素进行堆排序，得到的堆排结果是由大到小的。
        heapSort(A,k);
        //因此，在这里，将A[0]处的内容，重新存储成A[n-1]位置上。
        for(i = 0,j = n-1;i < k;j--,i++)
            A[j] = A[i];
        //0-k之外的部分，直接copy B里的
        for(i =0;i <n-k;i++)
            A[i]= B[i];
        return A;
    }

    //堆排序
    void heapSort(vector<int> &A,int n)
    {
        int i,temp;
        //将 (0, n-1)区间里的元素，建立一个小顶堆
        for(i = n/2-1;i>=0;--i)
            heapAdjust(A,i,n-1);
        //A中元素从后往前，
        for(i = n-1;i >0;i--)
        {
            //存贮小顶堆的堆顶
            temp = A[0];
            //将A中元素从后往前依次放入堆顶
            A[0] = A[i];
            //?
            A[i] = temp;
            //放入新的堆顶后，重新调整小顶堆
            heapAdjust(A,0,i-1);
        }
    }

    //已知A[s,...,m]中记录的关键字除A[s]之外均满足堆的定义，本函数调整A[s]的关键字，使A[s,...,m]成为一个小顶堆
    void heapAdjust(vector<int> &A,int s,int m)
    {
        //记录堆顶元素
        int j,rc = A[s];
        for(j = 2*s+1;j <=m;j = 2*j+1)
        {
            //如果堆顶的左节点比右节点大，则不用考虑左节点，j加一，指向右节点
            if(j < m && A[j]>A[j+1])
                j++;
            //如果堆顶元素小于A[j]的话，则不用执行，直接break
            if(rc < A[j])
                break;
            //否则，A[j]是最小的，将其放至堆顶
            A[s] = A[j];
            //将s的位置更新为原来j的位置
            s = j;
        }
        //将原来的堆顶元素，放至新的s处。
        A[s] = rc;
    }
};


int main()
{
    int b[]={2,1,4,3,6,5,8,7,10,9};
    //定义容器arr的值为b的第0个元素，到b的第九个元素，共10个
    vector<int> arr(b,b+10);
    SmallScaleSort s;
    s.sortElement(arr,10,2);
    for(int i = 0;i < 10;i++)
        cout<<arr[i]<<" ";
    cout<<endl;
    return 0;
}
```

E2，判断数组中是否有重复值，必须保证额外空间复杂度为O（1）：

解题思路：

这里如果没有空间复杂度为O（1）的限制，其实是应该用hash表的。在遍历数组的过程中，统计每个数出现的次数。但hash的时间，空间复杂度均为O（N）。

这里如果要求空间复杂度为O（1）的话，我们就需要先用O（1）空间复杂度的排序算法排序，然后相同的值会贴在一起。然后我们遍历一遍数组，就可以发现数组中是否有重复值。

因此，这里我们选择空间复杂度为O（1）的堆排序，如果利用递归的方式实现堆得话，时间复杂度就为O（log N）

所以，这里我们应该改出一个非递归版本的堆排序

Code:

```
/*
问题：
    判断数组中是否有重复值，必须保证额外空间复杂度为0（1）；
解题思路：
    这里如果没有空间复杂度为0（1）的限制，其实是应该用hash表的。在遍历数组的过程中，统计每个数出现的次数。但hash的时间，空间复杂度均为0（N）。
    这里如果要求空间复杂度为0（1）的话，我们就需要先用0（1）空间复杂度的排序算法排序，然后相同的值会贴在一起。然后我们遍历一遍数组，就可以发现数组中是否有重复值。
    因此，这里我们选择空间复杂度为0（1）的堆排序，如果利用递归的方式实现堆得话，时间复杂度就为0（log N）
    所以，这里我们应该改出一个非递归版本的堆排序
*/
#include<iostream>
#include<vector>

using namespace std;

class DuplicateChecker
{
public:
    //判断数组中是否存在重复值
    bool checkDuplicate(vector<int> &a,int n)
    {
        //堆排序
        heapSort(a,n);
        int i;
        //遍历排好序的数组，然后判断是否存在重复值
        for(i = 0; i < n-1;i++)
        {
            if(a[i] == a[i+1])
                return true;
        }
        return false;
    }
    //堆排序
    void heapSort(vector<int> &A,int n)
    {
        int i, temp;
        //调整至大顶堆
        for(i = n/2-1;i>=0;--i)
            heapAdjust(A,i,n-1);
        for(i = n-1;i>0;i--)
        {
            //得到大顶堆的堆顶元素，最大值
            temp = A[0];
            //将数组尾部元素放入堆顶
            A[0] = A[i];
            //将temp，即最大值，放入数组尾部
            A[i] = temp;
            //再调整至大顶堆
            heapAdjust(A,0,i-1);
        }
    }
    //已知A[s,...,m]中记录的关键字除A[s]之外均满足堆的定义，本函数调整A[s]的关键字，使A[s,...,m]成为一个大顶堆
    void heapAdjust(vector<int> &A,int s,int m)
    {
        //rc记录堆顶元素
        int j,rc =A[s];
        for(j = 2*s+1;j <=m;j=2*j+1)
        {
            //如果j<m且左节点的值小于右节点的值，则不考虑，索引加1
            if(j<m && A[j]<A[j+1])
                j++;
            //如果根节点的值大于右节点的值，则无需调整，break
            if(rc >A[j])
                break;
            //否则的话，将最大值放至堆顶
            A[s]= A[j];
            //将根节点换至原来j的位置
            s = j;
        }
        A[s] =rc;
    }
};


int main()
{
    int b[]={1,2,3,4,5,5,6};
    vector<int> arr(b,b+7);
    DuplicateChecker d;
    bool c=d.checkDuplicate(arr,7);
    cout<<c<<endl;
    return 0;
```

```
}
```

E3,把两个有序数组合并进第一个数组，然后第一个数组的空间，正好可以容纳两个数组的元素。假设有序数组A：2,4,6，和三个额外空间。B：1,3,5

解题思路：

a，首先，将第一个数组的最大元素即最后一个元素6和第二个数组的最后一个元素比较。然后6比较大，就将6拷贝到第一个数组的最后一个位置。

b,接下来，是4和5进行比较，发现5比较大，就把5拷贝到数组的倒数第二个位置。

c,再接着，将第二个数组的3和第一个数组的4进行比较，4比较大，就拷贝到第一个数组的倒数第三个空白位置

d,然后比较第二个数组的3和第一个数组的2，然后3比较大，就放在第一个数组的倒数第四个空白位置

e,再然后，将第二个数组的1和第一个数组的2进行比较，2比较大，就放在第一个数组的倒数第五个位置

f,再接着第二个数组的1，放在数组的初始位置

（从后往前覆盖数组A）

Code:

```
/*
问题：
    把两个有序数组合并进第一个数组，然后第一个数组的空间，正好可以容纳两个数组的元素。假设有序数组A：2,4,6，和三个额外空间。B：1,3,5
解题思路：
    a，首先，将第一个数组的最大元素即最后一个元素6和第二个数组的最后一个元素比较。然后6比较大，就将6拷贝到第一个数组的最后一个位置。
    b,接下来，是4和5进行比较，发现5比较大，就把5拷贝到数组的倒数第二个位置。
    c,再接着，将第二个数组的3和第一个数组的4进行比较，4比较大，就拷贝到第一个数组的倒数第三个空白位置
    d,然后比较第二个数组的3和第一个数组的2，然后3比较大，就放在第一个数组的倒数第四个空白位置
    e,再然后，将第二个数组的1和第一个数组的2进行比较，2比较大，就放在第一个数组的倒数第五个位置
    f,再接着第二个数组的1，放在数组的初始位置
    （从后往前覆盖数组A）
*/
#include<iostream>
#include<vector>
using namespace std;
class MergeOrderedArray
{
public:
    //传入数组A,B,及各自的长度
    int* mergeAB(int* A,int* B,int n,int m)
    {
        int i ,j ,newLength = m+n;
        for(i = n-1,j = m-1;i>=0&&j>=0;newLength--)
        {
            //如果A中的最后一个元素大，则在A的新长度末尾放入A，同时A的索引往左移一位
            if(B[j]<=A[i])
                A[newLength-1] = A[i--];
            //否则的话，就是B中的元素较大，则在A的新长度末尾放入B，同时B的索引往左移动一位
            else
                A[newLength-1] = B[j--];
        }
        //处理B最后剩下的一位
        while(j>=0)
        {
            A[newLength-1] = B[j--];
            newLength--;
        }
        return A;
    }
};


int main()
{
    int a[10]={1,3,7,8};
    int b[]={0,2,5,9};
    MergeOrderedArray m;
    m.mergeAB(a,b,4,4);
    for(int i=0;i<8;i++)
        cout<<a[i]<<" ";
    cout<<endl;
    return 0;
}
```

E4，经典荷兰国旗问题。只包含0,1,2的整数的数组进行排序。要求排序完的结果是0都在数组的左边，1都在数组的中间，2都在数组的右边。要求使用交换，原地排序，而不是利用计数进行排序。

解题思路：

其排序过程和快排是比较像的。最后能实现时间复杂度为O(N)，空间复杂度为O（1）。

a,在遍历整个数组之前，先在左侧设立一个0区{}，初始时长度为0，同时在右侧设立一个2区{}，初始长度也为0.

b,从左到右遍历整个数组，如果为1的话，则跳过进行下一个数，若为0，则和0区的下一个位置上的数进行交换，交换完0区得长度+1，若为2，则和2区的上一个位置的数进行交换，交换完2区的长度+1，但交换到这个位置的数是没有比较过得数，因此如果为2的话，做完这些操作仍停留在原地，进行比较操作。

c，当当前的位置和2区的位置重合时，过程停止。

Code:

```
/*
问题：
```

```
    经典荷兰国旗问题。只包含0,1,2的整数的数组进行排序。要求排序完的结果是0都在数组的左边，1都在数组的中间，2都在数组的右边。要求使用交换，原地排序，而不是利用计数进行排序。
解题思路：
    其排序过程和快排是比较像的。最后能实现时间复杂度为O(N)，空间复杂度为O（1）。
    a,在遍历整个数组之前，先在左侧设立一个0区{}，初始时长度为0，同时在右侧设立一个2区{}，初始长度也为0.
    b,从左到右遍历整个数组，如果为1的话，则跳过进行下一个数，若为0，则和0区的下一个位置上的数进行交换，交换完0区得长度+1，若为2，则和2区的上一个位置的数进行交换，交换完2区的长度+1，但交换到这个位置的数是没有比较过得数，因此如果为2的话，做完这些操作仍停留在原地，进行比较操作。
    c，当当前的位置和2区的位置重合时，过程停止。

*/
#include<iostream>
#include<vector>
using namespace std;
class ThreeColorSort
{
public:
    vector<int> sortThreeColor(vector<int> &A,int n)
    {
        int f,r,i,temp;
        //f为0区索引，r为2区索引。i从0-n-1,从左往右索引
        for(i = f =0,r = n-1; i<=r;i++)
        {
            //如果元素为0
            if(A[i] == 0)
            {
                //存贮原来0区下一个元素
                temp = A[f];
                //如果当前元素为0，就将其换至0区的下一个元素
                A[f] = A[i];
                //并把原来0区下一个元素挪到这个交换的位置
                A[i] =temp;
                //0区的索引往后挪一位
                f++;
            }
            //如果元素为2
            if(A[i] == 2)
            {
                //存贮2区的前一个元素，即数组的末尾
                temp = A[r];
                //如果当前元素为2，将其换至2区的前一个元素
                A[r]=A[i];
                //将原来2区的前一个元素挪到这个交换的位置
                A[i] = temp;
                //2区的索引往前挪一位
                r--;
                //2的话，换过来的元素还未经过判定，所以i--,再结合for循环里的i++，相当于保持原地不动
                i--;
            }
        }
        return A;
    }
};

int main()
{
    int a[6]={1,2,0,2};
    vector<int> b(a,a+4);
    ThreeColorSort t;
    t.sortThreeColor(b,4);
    for(int i=0;i<4;i++)
        cout<<b[i]<<" ";
    cout<<endl;
    return 0;
}
```

## E5，在行列都排好序的矩阵中找数，如果有的话，返回True，否则返回False。

0 1 2 5
2 3 4 7
4 4 4 8
5 7 7 9

## 解题思路：

假定二位数组大小为M*N的话，这个过程的时间复杂度可以做到O（M+N），空间复杂度可以做到O（1），比如我们现在找3在不在这个矩阵中。

1，从矩阵的右上角开始找，大于要找的数的话，由于行列都是有序的，这一列下面的数也一定大于这个数，因此向左移动。

2，如果向左移动到的2小于我们要找的3的话，其左边的也一定小于3，因此从2往下移动。

3，然后4又大于我们要找的3，只能往左移动。Find it。

根据每一位置上的数的比较，来判断是向左还是向下，如果这个过程中找到了，我们就直接返回True.如果直到越界了还没找到，则返回false

## Code:

```
/*
问题：
    在行列都排好序的矩阵中找数，如果有的话，返回True，否则返回False。
    0 1 2 5
    2 3 4 7
    4 4 4 8
```

```
    5 7 7 9
解题思路:
    假定二位数组大小为M*N的话，这个过程的时间复杂度可以做到O（M+N），空间复杂度可以做到O（1），比如我们现在找3在不在这个矩阵中。
    1，从矩阵的右上角开始找，大于要找的数的话，由于行列都是有序的，这一列下面的数也一定大于这个数，因此向左移动。
    2，如果向左移动到的2小于我们要找的3的话，其左边的也一定小于3，因此从2往下移动。
    3，然后4又大于我们要找的3，只能往左移动。Find it。
    根据每一位置上的数的比较，来判断是向左还是向下，如果这个过程中找到了，我们就直接返回True.如果直到越界了还没找到，则返回false

*/
#include<vector>
#include<iostream>
#include<string.h>
using namespace std;
class NumFinder
{
public:
    //n和m分别为行和列的值，x为待寻找的数
    bool findX(int mat[3][3],int n, int m, int x)
    {
        //定义行列索引，从右上角开始
        int hang = 0,lie = m-1;
        while(hang < n && lie >= 0)
        {
            //找到，return true
            if(mat[hang][lie] == x)
                return true;
            //如果当前值大于x，则往左，列减1
            if(mat[hang][lie]>x)
                lie--;
                //如果当前值小于x，则往下，行加1
            if(mat[hang][lie] <x)
                hang++;
        }
        return false;
    }
};
int main()
{
    //定义一个3*3的二维数组
    int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
    NumFinder n;
    bool d=n.findX(a,3,3,9);
    cout<<d<<endl;
    return 0;
}
```

E6，给定一个数组 1 5 4 3 2 6 7，返回数组中需要排序的最短子数组长度。比如这里只有 5 4 3 2 需要排序，因此返回4

解题思路：

这里的最优解时间复杂度为O(N)，空间复杂度为O（1）

a,首先，从左到右遍历数组，同时，用一个额外的变量记录下遍历过程中遇到的最大值，

b，然后我们只关注一种情况，就是记录的最大值大于当前遍历值得情况。然后记录下发生这种情况最右的位置。

c，接下来，从右往左遍历数组，同时，用一个额外的变量记录下遍历过程中遇到的最小值，

d，然后我们也只关注一种情况，就是记录的最小值小于当前遍历值得情况。然后记录下发生这种情况下最左的位置。

e，然后最左，左右区间的这个范围，就是数组中需要排序的最短子数组

Code:

```
/*
问题:
    给定一个数组 1 5 4 3 2 6 7，返回数组中需要排序的最短子数组长度。比如这里只有 5 4 3 2 需要排序，因此返回4
解题思路:
    这里的最优解时间复杂度为O(N)，空间复杂度为O（1）
    a,首先，从左到右遍历数组，同时，用一个额外的变量记录下遍历过程中遇到的最大值，
    b，然后我们只关注一种情况，就是记录的最大值大于当前遍历值得情况。然后记录下发生这种情况最右的位置。
    c，接下来，从右往左遍历数组，同时，用一个额外的变量记录下遍历过程中遇到的最小值，
    d，然后我们也只关注一种情况，就是记录的最小值小于当前遍历值得情况。然后记录下发生这种情况下最左的位置。
    e，然后最左，左右区间的这个范围，就是数组中需要排序的最短子数组
*/

#include<iostream>
#include<vector>
using namespace std;
class SubSequence
{
public:
    int shortSubSequence(vector<int> A,int n)
    {
        //初始化最大值为A[0],最小值为A[n-1],rd1为从左到右遍历，最大值大于当前值的最右位置
        //rd2为从右往左遍历，记录的最小值小于当前遍历值的情况的最左位置
        int max = A[0],min = A[n-1],i,rd1,rd2;
        //从位置1开始（0已经初始化为最大值），从左往后，找最大值，
        for(i = 1,rd1 =0;i<n;i++)
```

```
        {
            //找最大值
            if(A[i]>=max)
                max = A[i];
            //否则的话是，最大值大于当前值，最右位置索引挪至这里
            else
                rd1 = i;
        }
        //从位置n-2开始（n-1已经初始化为最小值），从右往左，找最小值
        for(i = n-2,rd2 = n-1;i>=0;i--)
        {
            //如果当前值比min小，则min替换为当前值
            if(A[i]<= min)
                min = A[i];
            //否则，min比当前小，最左索引挪至这里
            else
                rd2 =i;
        }
        //如果最右索引为0
        if(!rd1)
            return 0;
        else
            return rd1-rd2+1;
    }
};


int main()
{
    int a[6]={1,4,6,5,9,10};
    vector<int> b(a,a+6);
    SubSequence s;
    int d=s.shortSubSequence(b,6);
    cout<<d<<endl;
    return 0;
}
```

E7，给定一个整形数组，返回如果排序之后，相邻两数的最大差值。比如某数组排好序之后为1 2 3 4 7 8 9，那么最大差值来自7-4

解题思路：

这道题的最优解可以做到时间复杂度为O(N)，空间复杂度为O(N)，它的思想来自于桶排序的思想，但又不是真实的桶排序。如果数组长度为N的话，其空间复杂度可以做到O(N),而根数组中数值的范围没有关系。

a,首先我们遍历数组，找到其中的最小值和最大值。比如这里的1和9

b,在最小值和最大值之间，我们根据数组长度N，将其等量的分为N个区间。这里有7个数，因此区间分别为，（1-15/7）...(55/7,9),7个。

c,我们将每个数值放入对应范围的桶中，其中最大值放在第N+1号桶中。这样以来，1号桶中有最小值，第N+1号桶中有最大值。

d，桶有N+1个，而数值只有N个，因此，1号桶和N+1号桶之间必定有空桶；我们知道，桶内数据的差值，不会大于桶区间的大小。而空桶两侧的值，肯定会大于桶区间。

e,因此，我们完全不用考虑来自一个桶内元素的差值，只需要考虑不同桶之间元素的差值。也就是后一个桶的最小值减去前一个桶的最大值。然后我们记录下这个最大的差值即可。

Code:

```
/*
问题：
    给定一个整形数组，返回如果排序之后，相邻两数的最大差值。比如某数组排好序之后为 1 2 3 4 7 8 9，那么最大差值来自7-4
解题思路：
    这道题的最优解可以做到时间复杂度为O(N)，空间复杂度为O(N)，它的思想来自于桶排序的思想，但又不是真实的桶排序。如果数组长度为N的话，其空间复杂度可以做到O(N),而根数组中数值的范围没有关系。
    a,首先我们遍历数组，找到其中的最小值和最大值。比如这里的1和9
    b,在最小值和最大值之间，我们根据数组长度N，将其等量的分为N个区间。这里有7个数，因此区间分别为，（1-15/7）…(55/7,9),7个。
    c,我们将每个数值放入对应范围的桶中，其中最大值放在第N+1号桶中。这样以来，1号桶中有最小值，第N+1号桶中有最大值。
    d，桶有N+1个，而数值只有N个，因此，1号桶和N+1号桶之间必定有空桶；我们知道，桶内数据的差值，不会大于桶区间的大小。而空桶两侧的值，肯定会大于桶区间。
    e,因此，我们完全不用考虑来自一个桶内元素的差值，只需要考虑不同桶之间元素的差值。也就是后一个桶的最小值减去前一个桶的最大值。然后我们记录下这个最大的差值即可。
*/
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
class MaxGap
{
public:
    int maxGap(vector<int> A,int n)
    {
        //新建桶，n+1个
        vector<vector<int> > B(n+1);
        //最大最小值，索引，最大间隔
        int max,min,i,j,maxGap;
        for(i =1,max = min = A[0];i<n;i++)
        {
            if(A[i]>max)
                max = A[i];
            if(A[i]<min)
                min = A[i];
```

```
        }
        //桶宽度
        float gap = 1.0*(max-min)/n;
        //将元素放入对应范围的桶中，最大值放在N+1号桶。
        for(i =0;i < n;i++)
        {
            //元素A[i]应该放在第几号桶
            j = (int)((A[i]-min)/gap);
            //如果该号桶为空，随便放//？？
            if(B[j].empty())
            {
                //两个是为了等下替换掉一个值
                B[j].push_back(A[i]);
                B[j].push_back(A[i]);
            }
            //如果该桶非空，则需要判断下，将大的值放在桶顶，小的值放在桶底
            //注意，这里我们存贮的只是每个桶内元素的最大值和最小值，分别在位置1和0
            else
            {   //将小的值放在0位置
                if(B[j][0]>A[i])
                    B[j][0] = A[i];
                //将大的值放在1位置
                if(B[j][1]<A[i])
                    B[j][1] = A[i];
            }
        }
        //计算最大间隔,i为桶号
        for(i = j =0,maxGap = 0;i<n;i = j)
        {
            //如果空桶，则下一个
            while(B[i].empty())
                i++;
            //得到该桶的最大值，在桶顶的位置。
            max = B[i][1];
            //j为桶i接下来的桶的索引，之间可能有空桶
            j = i+1;
            if(j < n+1)
            {
                while(B[j].empty())
                    j++;
                if(j >= n+1)
                    break;
                //得到接下来一个桶的最小值
                min = B[j][0];
                if(min-max>maxGap)
                    maxGap = min - max;
            }
        }
        return maxGap;
    }
};
int main()
{
    int a[5]={1,2,5,4,6};
    vector<int> arr(a,a+5);
    MaxGap m;
    int c=m.maxGap(arr,5);
    cout<<c<<endl;
    return 0;
}
```

## 3，字符串（11.9-11.10，11.13写完5789的代码）

### A，字符串面试题的特点：

#### a,广泛性：

1，字符串可以看做字符类型的数组，可以跟数组的排序，查找有关。

2，很多其他类型都可以转化为字符串类型，进行处理

#### b，注意：

1.在java中字符串是不可更改的，此时可以使用StringBuffer，StringBuilder来进行处理。必要时，用toCharArray方法将字符串转为字符类型的数组。

#### c，一些概念：

1，回文

2，子串（连续）

3，子序列（不连续）

4，前缀树（trie树）

5，后缀树和后缀数组

6，匹配

7，字典序

#### d,一些操作：

1，与数组有关的操作，增删改查

2，字符的替换

3，字符串的旋转

## B，一些面试题：

### a，规则判断：

1，如判断一个字符串是否符合整数的规则，如果是，则将其转为整数返回

2，是不是浮点数，or回文规则

### b，数字运算：

1，int和long类型表达的整数范围有限，因此经常用string实现大整数，并进行大整数相关的加减乘除操作

### c，数组操作：

1，数组有关的调整排序等操作

2，快排的划分过程需要掌握和改写

### d，字符计数：

1，用hash表来统计字符出现的次数

2，用固定长度的数组，c++（255），java（65535）来代替hash表。

3，常见问题有：滑动窗口问题，寻找无重复字符子串问题，计算变位词问题。

### e,动态规划：

1，最长公共子串问题

2，最长公共子序列问题

3，最长回文子串

4，最长回文子序列

### f，搜索类型：

如有Str1和str2，str1每次移动1位，如何将str1->str2，打印变换轨迹，

1,这里采用宽度优先搜索

### g，高级算法和数据结构解决的问题：

1，manacher算法解决最长回文子串问题

2，KMP算法解决字符串匹配问题

3，前缀树结构

4，后缀树和后缀数组

5，面试中一般不出现，

## C1，问题：

给定彼此独立的两棵树，头节点分别为t1和t2，判断t1中是否有与t2树拓扑结构完全相同的子树。

### 解题思路：

普通解法是二叉树遍历与匹配问题，先遍历二叉树，找到与t2头节点相同的节点时，接下来看结构是否与t2对应，是的话返回true。这样的话，如果t1的大小为N，t2的大小为M的话，时间复杂度为O(N*M)

但这道题的最优解可以做到O（M+N），这个题表面来看是二叉树问题，实际上是字符串问题。先对t1，序列化成字符串s1，t2也序列化为字符串2，然后判断str1是否包含str2即可。

### code：

```
/*
问题：
    给定彼此独立的两棵树，头节点分别为t1和t2，判断t1中是否有与t2树拓扑结构完全相同的子树。
 解题思路：
    普通解法是二叉树遍历与匹配问题，先遍历二叉树，找到与t2头节点相同的节点时，接下来看结构是否与t2对应，是的话返回true。这样的话，如果t1的大小为N，t2的大小为M的话，时间复杂度为O(N*M)
    但这道题的最优解可以做到O（M+N），这个题表面来看是二叉树问题，实际上是字符串问题。先对t1，序列化成字符串s1，t2也序列化为字符串2，然后判断str1是否包含str2即可。


*/
#include<iostream>
#include<vector>
#include<string>
#include<map>
#include<cstdlib>
using namespace std;
vector<char>::size_type n =0;
struct TreeNode
{
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x):val(x),left(NULL),right(NULL){
    }
};
class IdenticalTree
{
public:
    //传入二叉树A，B的头节点，返回B是否在A中
    bool chktree(TreeNode *A,TreeNode *B)
    {
        //二叉树序列化
        string s1 = toString(A),s2 = toString(B);
        //字符串查找
```

```
        if(s1.find(s2)!=string::npos)
            return true;
        else
            return false;
    }
    //二叉树的序列化,传入二叉树的根节点，返回序列化后的字符串
    string toString(TreeNode* root)
    {
        string res;
        Serialize(root,res);
        return res;
    }
    //序列化过程,传引用
    void Serialize(TreeNode *root,string &res)
    {
        //如果根节点为空
        if(!root)
        {
            res+="#!";
            return;
        }
        //将根节点的值从int转为字符串
        res+=IntegertoStr((*root).val);
        //递归处理左右子树
        Serialize((*root).left,res);
        Serialize((*root).right,res);
    }

    //将整形转为字符串,用！做结束符
    string IntegertoStr(int m)
    {
        //如果m值为0
        if(!m)
            return "0!";
        string res;
        vector<int> temp;
        //将m逐位放入temp，从个位到高位
        while(m)
        {
            temp.push_back(m%10);
            m = m/10;
        }
        //将temp的内容从高位到低位放入res。
        for(vector<int>::reverse_iterator riter = temp.rbegin();riter!=temp.rend();riter++)
            res.push_back(*riter+48);
        res.push_back('!');
        return res;
    }
};

//用于检验，将定义的字符串转为二叉树
class DeSerialize
{
public:
        //传入序列化向量，和待重建的二叉树头节点，头节点是引用的方式传递（可以直接修改其本身的值）
        void deserialize(vector<char> A,TreeNode* &head)
        {
                int i, j, value = 0;
                //代码最开始初始化了n，为0. 如果A.size = 0或者其第0个元素为#（即空）
                if(n>A.size()-1 || A[n] == '#')
                        //是一颗空的二叉树
                        return;
                i = j = n;
                //统计！之前的这个字符有多少位
                while(A[j] != '!')
                        j++;
                //在这个字符长度之内
                while(j>i)
                {
                        //从高位到地位，将该字符的每一位存入value
                        value = (A[i]-'0')+value*10;
                        i++;
                }
                //分配头结点内存,并给头结点对应元素赋值
                head = (TreeNode *)malloc(sizeof(TreeNode));
                (*head).val = value;
                (*head).right = (*head).left = NULL;
                //将索引往后挪
                n = i+1;
                //递归
                deserialize(A,(*head).left);
                deserialize(A,(*head).right);
        }
};
```

```
int main()
{
    //定义序列化的二叉树转为的字符数组
        char a[11] = {'1','2','!','3','!','#','!','#','!','#','!'};
    char b[11] ={'1','2','!','3','!','#','!'};
        //将字符数组转为序列化向量
        vector<char> va(a,a+11);
    vector<char> vb(b,b+7);
        //反序列化：将序列化向量转为二叉树
        DeSerialize d;
        TreeNode* A = NULL;
    TreeNode* B = NULL;
    d.deserialize(va,A);
    d.deserialize(vb,B);
    //测试
    IdenticalTree i;
    bool bl = i.chktree(A,B);
    cout<<bl<<endl;
}
```

C2，

问题

给定两个字符串str1和str2，判断两者是否为变形词（出现的字符种类一样，且每种字符出现的次数也一样）。请实现函数判断两个字符串是否互为变形词。

解题思路：

使用hash表做词频统计即可。Str1建立hash表1，str2建立hash表2，比对两个hash表的记录即可。实现的时候，可用固定长度的数组来替代hash表，其时间复杂度为O(N),空间复杂度为O（N）。如字符ASCII范围为255的话，则生成256大小的数组，如果是65535的话，则生成65536大小的数组。可以更加省空间，同时速度也更快。

code：

```
/*
问题：
    给定两个字符串str1和str2，判断两者是否为变形词（出现的字符种类一样，且每种字符出现的次数也一样）。请实现函数判断两个字符串是否互为变形词。
解题思路：
    使用hash表做词频统计即可。Str1建立hash表1，str2建立hash表2，比对两个hash表的记录即可。实现的时候，可用固定长度的数组来替代hash表，其时间复杂度为O(N),空间复杂度为O（N）。如字符ASCII范围为255的话，则生成256大小的数组，如果是65535的话，则生成65536大小的数组。
    可以更加省空间，同时速度也更快。


*/
#include<iostream>
#include<string>
#include<vector>
using namespace std;
class ChkTransForm
{
public:
    bool chkTransForm(string A,int lena,string B,int lenb)
    {
        if(lena != lenb)
            return false;
        int hashTable1[256]={0},hashTable2[256]={0},i;
        //将字符串中的每一位，其ascii值为索引，统计次数
        for(i = 0;i< lena;i++)
        {
            hashTable1[A[i]]++;
            hashTable2[B[i]]++;
        }
        i = 0;
        //按ascii码顺序对两个数组进行相似性判定
        while((i<256)&&hashTable1[i] == hashTable2[i])
            i++;
        if(i>=256)
            return true;
        else
            return false;
    }
};

int main()
{
    string a("abcd"),b("rbca");
    ChkTransForm c;
    bool bl = c.chkTransForm(a,4,b,4);
    cout<<bl<<endl;
    return 0;
}
```

C3，

问题：

如果一个字符串str，将str前面任意部分挪到后面去形成的字符串叫做str的旋转词，比如str=1234，str的旋转词有1234,2341,3412,4123，给定两个字符串a，b，请判断a，b是否互为旋转词

解题思路：

最优解，时间复杂度为O（N），首先判断长度是否相等，否则false，如果长度相等的话，生成str1+str1的大字符串，然后用KMP算法判断大字符串中是否含有str2，如果包含，str1和str2是互为旋转词的。这是因为str1+str2这个大字符串，涵盖了str1的所有旋转词。

code：

见1中的练习题

## C4，

问题：

给定一个字符串，请在单词见做逆序调整。如，“pig loves dog",逆序成”dog loves pig“

解题思路：

1，首先，我们要实现一个将字符串局部所有字符逆序的函数f，如abcdef，先第一个和最后一个交换，然后交换第二个和倒数第二个，交换到中间停止。

2，接着，利用f将字符串中所有字符逆序，变成“god sevol gip”。

3，最后，找到逆序后的字符串中每个单词的区域，利用f将每个单词的区域逆序，变成“dog loves pig”

code：

```
/*
问题:
    给定一个字符串，请在单词见做逆序调整。如，“pig loves dog”,逆序成”dog loves pig“
解题思路:
    1，首先，我们要实现一个将字符串局部所有字符逆序的函数f，如abcdef，先第一个和最后一个交换，然后交换第二个和倒数第二个，交换到中间停止。
    2，接着，利用f将字符串中所有字符逆序，变成“god sevol gip”。
    3，最后，找到逆序后的字符串中每个单词的区域，利用f将每个单词的区域逆序，变成“dog loves pig”

*/
#include<iostream>
#include<string>
#include<vector>
using namespace std;
class SentenceReverse
{
public:
    //传入字符串和字符串长度
    string reverseSentence(string A, int n)
    {
        //逆序整个字符串
        reverseWord(A,0,n-1);
        int i = 0, j = 0;
        //逆序每个单词
        while(i < n)
        {
            //空格的话，索引往右挪
            while(i<n &&A[i] == ' ')
                i++;
            if(i>=n)
                return A;
            //j为每个单词的起始位置
            j = i;
            while(i <n &&A[i]!=' ')
                i++;
            //如果到最后的话，翻转完直接返回
            if(i >= n)
            {
                reverseWord(A,j,n-1);
                return A;
            }
            //翻转每个单词。上面while判定的时候，i多加了一次。
            reverseWord(A,j,i-1);
        }
        return A;
    }

    //逆序函数
    void reverseWord(string &A, int low, int high)
    {
        if(low>=high)
            return;
        char temp;
        while(low <high)
        {
            //交换最左最右位置的值
            temp = A[low];
            A[low]=A[high];
            A[high] = temp;
            //两者往中间移，直到low>high,即过了中点。
            low++;
            high--;

        }
```

```
    }
};

int main()
{
    string arr("dog loves pig");
    SentenceReverse s;
    string arr1 = s.reverseSentence(arr,13);
    cout<<arr1<<endl;
    return 0;
}
```

C5，

问题：

给定一个字符串str和一个整数i，i代表str中的位置，将str[0,i]移到右侧，str[I+1,n-1]移到左侧，举例：

str：“ABCDE”，i=2，将str调整为”DEABC“。要求，时间复杂度为O（N），额外空间复杂度为O（1）

解题思路：

1，先将str[0,i]上的部分逆序，ABCDE，变成CBADE

2，将str[i+1,n-1]上的部分逆序，变成CBAED

3，然后将str整体逆序，DEABC

Code:

```
/*
    问题：
        给定一个字符串str和一个整数i，i代表str中的位置，将str[0,i]移到右侧，str[I+1,n-1]移到左侧，举例：
        str：“ABCDE”，i=2，将str调整为”DEABC“。要求，时间复杂度为O（N），额外空间复杂度为O（1）
    解题思路：
        1，先将str[0,i]上的部分逆序，ABCDE，变成CBADE
        2，将str[i+1,n-1]上的部分逆序，变成CBAED
        3，然后将str整体逆序，DEABC
*/
#include<iostream>
#include<string>
#include<vector>
#include<stdlib.h>
using namespace std;
class StringShift
{
public:
    //返回交换过顺序后的字符串，传入：字符串，字符串长度，中间分割位置
    string stringShift(string A, int n,int len)
    {
        if(len > n)
            exit(-1);
        reverseWord(A,0,len-1);
        reverseWord(A,len,n-1);
        reverseWord(A,0,n-1);
        return A;
    }

    //局部逆序函数
    void reverseWord(string &A,int low,int high)
    {
        if(low >= high)
            return;
        char temp;
        while(low < high)
        {
            temp = A[low];
            A[low] = A[high];
            A[high] =temp;
            low++;
            high--;
        }
    }
};

int main()
{
    string s("ABCDE");
    StringShift ss;
    string res = ss.stringShift(s,5,3);
    cout<<res<<endl;
    return 0;
}
```

C6，

问题：

给定一个字符串类型的数组5strs，请找到一个拼接顺序，使得将所有字符串拼接起来组成的大字符串是所有可能性中字典顺序最小的，并返回这个大字符串。如strs = ["abc","de"]，可以拼成“abcde” ，也可以拼成“deabc” ，但前者字典序更小，所以返回“abcde”

解题思路：

如果数组长度为N的话，那么这道题的最优解为O（N*logN），其实质是一种排序的实现。排序的结果就是我们拼接的顺序。

一般的思路是：根据每个字符串的字典顺序排序：

如：["abc","de"] -》["abc","de"]

但这种思路是错误的，

如：["ba","b"] -》["b","ba"],这样拼接出来是bba，但对本题来说，最小的字典序是bba。所以这个思路是错误的。

正确的思路是：如果str1+str2 < str2+str1,则str1在前面，否则，str2在前面

code：

```
/*
问题:
        给定一个字符串类型的数组strs，请找到一个拼接顺序，使得将所有字符串拼接起来组成的大字符串是所有可能性中字典顺序最小的，并返回这个大字符串。
如strs = ["abc","de"]，可以拼成“abcde”，也可以拼成“deabc”，但前者字典序更小，所以返回“abcde”
解题思路:
        1,如果数组长度为N的话，那么这道题的最优解为O（N*logN），其实质是一种排序的实现。排序的结果就是我们拼接的顺序。
        2,一般的思路是：根据每个字符串的字典顺序排序:
                如：["abc","de"] -》["abc","de"]
                但这种思路是错误的，
                如：["ba","b"] -》["b","ba"],这样拼接出来是bba，但对本题来说，最小的字典序是bba。所以这个思路是错误的。
        3,正确的思路是：如果str1+str2 < str2+str1,则str1在前面，否则，str2在前面
*/
#include<iostream>
#include<string>
#include<vector>
using namespace std;
class MinDict
{
public:
    //找到最小字典序的数组排列。传入：strs为字符串数组，n为数组长度
    string findSmallest(vector<string> strs,int n)
    {
        //对字符串数组进行排序,排序的标准是：如果str1+str2 < str2+str1,则str1在前面，否则，str2在前面
        QuickSort(strs,0,n-1);
        string res;
        for(int i = 0; i < n;i++)
            res += strs[i];
        return res;
    }

    //对字符串数组里的元素按字典序排列
    void QuickSort(vector<string> &strs,int low,int high)
    {
        if(low < high)
        {
            //该操作后，res处的值处于它该处的位置，
            int res = Partition(strs,low,high);
            //迭代处理数组左边和右边
            QuickSort(strs,low,res-1);
            QuickSort(strs,res+1,high);
        }
    }

    //
    int Partition(vector<string> &strs,int low,int high)
    {
        //记录数组的初始元素
        string key = strs[low];
        while(low <high)
        {
            //当前的low和high，符合左边比较小，右边比较大的原则，所以不动，让high索引往前挪一位
            while(low<high && LT(key,strs[high]))
                high--;
            //否则的话，将high处的内容换过来
            strs[low] = strs[high];
            //将现在low处的值与原始的low值进行比较，如果比较小的话，则满足原则，low继续往右移动
            while(low <high&& LT(strs[low],key))
                low++;
            //将原来key处的值放在现在high的位置，然后继续处理
            strs[high] = key;

        }
        //将原始low处的元素放在它应该待的位置，新的low处。
        strs[low] = key;
        //此时low处的元素已被放在了合适的位置
        return low;
    }

    //判断s1+s2是大于还是小于s2+s1
    bool LT(string s1,string s2)
```

```
    {
        string temp1 = s1+s2,temp2 = s2+s1;
        if(temp1<= temp2)
            return true;
        else
            return false;
    }
};

int main()
{
    string a("abc"),b("de"),c("cab");
    vector<string> arr;
    arr.push_back(b);
    arr.push_back(a);
    arr.push_back(c);
    MinDict m;
    string res = m.findSmallest(arr,3);
    cout<<res<<endl;
    return 0;
}
```

C7，

问题：

给定一个字符串str，将其中所有空格字符替换成“%20”，假设str有足够空间

解题思路：

比如有 str=“a b c”

1，遍历str，发现空格数量为2，所以str在替换后，长度为3+2*3

2，从左往后遍历数组，从数组索引8的位置开始放元素，遇到非空格就直接放，如果有空格就依次放0,2，%

code：

```
/*
    问题：
        给定一个字符串str，将其中所有空格字符替换成“%20”，假设str有足够空间

    解题思路：
        比如有 str=“a b c”
        1，遍历str，发现空格数量为2，所以str在替换后，长度为3+2*3
        2，从左往后遍历数组，从数组索引8的位置开始放元素，遇到非空格就直接放，如果有空格就依次放0,2，%

*/
#include<iostream>
#include<string>
#include<vector>
using namespace std;
class SpaceReplace
{
public:
    string spaceReplace(string s,int len)
    {
        int spaceNum = 0,i,j;
        //原数组空格数
        for(i = 0; i<len;i++)
        {
            if(s[i] == ' ')
                spaceNum++;
        }
        //新数组长度
        int newLen = len+2*spaceNum;
        //cout<<newLen<<endl;
        //倒序定义数组
        string res(newLen,0);
        //对原来的数组倒序访问，放入新数组
        for(i = len-1,j= newLen-1;j>=0;)
        {
            if(s[i] == ' ')
            {
                res[j--] = '0';
                res[j--] = '2';
                res[j--] = '%';
                i--;
            }
            else
                res[j--] = s[i--];
        }
        return res;
    }
};
int main()
{
    string s("a b c");
    SpaceReplace S;
```

```
    string res = S.spaceReplace(s,5);
    cout<<res<<endl;
    return 0;
}
```

C8，

问题：

给定一个字符串str，判断是不是整体有效的括号字符串

举例，str=“（）”，true，

str =“（（）（））”，true

Str =（（））,true

Str =（（），false

Str =（）（，false

Str=（）a（），返回false

解题思路：

最优解时间复杂度，o(n), 空间复杂度O(1)

1，定义一个整形遍历num，代表（出现次数和）出现次数的差值

2，遍历过程中遇到（，则num++，

3，遍历过程中遇到），则num--。

4，遍历过程中如果num<0,则返回false

5，如果一直没有出现情况4，则一直遍历下去

6，遍历完成后，如果num==0，则返回true，否则返回false

code：

```
/*
    问题:
        给定一个字符串str，判断是不是整体有效的括号字符串
        举例，str=“（）”，true，
            str =“（（）（））”，true
            Str =（（））,true
            Str =（（），false
            Str =（），false
            Str=（）a（），返回false
    解题思路:
        最优解时间复杂度，O(n)，空间复杂度O(1)
        1，定义一个整形遍历num，代表（出现次数和）出现次数的差值
        2，遍历过程中遇到（，则num++，
        3，遍历过程中遇到），则num--。
        4，遍历过程中如果num<0,则返回false
        5，如果一直没有出现情况4，则一直遍历下去
        6，遍历完成后，如果num==0，则返回true，否则返回false

*/
#include<iostream>
#include<vector>
#include<string>
using namespace std;
class ParenthesesJudge
{
public:
    bool chkP(string s,int len)
    {
        int n = 0,i;
        for(i = 0;i<len;i++)
        {
            if(s[i] == '(')
                n++;
            if(s[i] == ')')
                n--;
            if(n<0)
                return false;
        }
        if(n == 0)
            return true;
        else
            return false;
    }
};
int main()
{
    string s("a()()");
    ParenthesesJudge p;
    bool b= p.chkP(s,5);
    cout<<b<<endl;
    return 0;
}
```

C9,

问题：

给定一个字符串str，返回str的最长无重复字符子串的长度。

举例：str="abdc",返回4

Str = “abcb”，则最长无重复字符子串为abc,返回3

解题思路：

最优解的时间复杂度为O(N),空间复杂度为O（N）

1，从左往后遍历，求出每个字符，往左，最长无重复子串的长度，并在其中找到最大值返回。

2，具体实施起来，我们需要两个变量，一个是hash表，map，其中统计了每种字符之前出现的位置，另一个整形变量pre，代表以当前字符的前一个字符结尾的情况下，最长无重复子串的长度。

3，然后查找当前字符在hash表中上一次出现的位置，标注其下一个位置为A，标注前一个字符往前最长无重复子串长度的位置为位置B，如果B比A近，则为B，如果B比A远，则为A。

4，根据这个位置来更新当前字符的pre变量，此外，hash表里的位置信息也需要更新，

5，然后获取pre的最大值，返回即可。

code：

```
/*
    问题：
        给定一个字符串str，返回str的最长无重复字符子串的长度。
        举例：str="abdc",返回4
            Str = “abcb”，则最长无重复字符子串为abc,返回3
    解题思路：
        最优解的时间复杂度为O(N),空间复杂度为O（N）
        1，从左往后遍历，求出每个字符，往左，最长无重复子串的长度，并在其中找到最大值返回。
        2，具体实施起来，我们需要两个变量，一个是hash表，map，其中统计了每种字符之前出现的位置，另一个整形变量pre，代表以当前字符的前一个字符结尾的情况下，最长无重复子串的长度。
        3，然后查找当前字符在hash表中上一次出现的位置，标注其下一个位置为A，标注前一个字符往前最长无重复子串长度的位置为位置B，如果B比A近，则为B，如果B比A远，则为A。
        4，根据这个位置来更新当前字符的pre变量，此外，hash表里的位置信息也需要更新，
        5，然后获取pre的最大值，返回即可。
*/
#include<iostream>
#include<string>
#include<map>
using namespace std;
class LongestSubstring
{
public:
    int longestSubstring(string s,int n)
    {
        //存贮每个字符和它出现的位置
        map<char,int> map;
        //上一个字符的最长无重复子串长度
        int pre = 0;
        //存贮每个字符的最长无重复子串长度
        int *dp = new int[n]();
        //计算每个字符
        for(int i = 0; i < n;i++)
        {
            //如果map中有key为s[i]的记录
            if(map.count(s[i]))
            {
                //如果第i个字符上次出现的位置比上一个字符的最长无重复子串位置往右,那么取i减去第i个字符上次出现的位置为最长无重复子串的位置
                if(map[s[i]] >= (i-pre))
                    dp[i] = i-map[s[i]];
                else
                    //最大无重复子串的长度：上一个字符的最大子串长度+1
                    dp[i]= pre+1;
            }
            //map中还没出现key为s[i]的记录
            else
                //更新pre和map
                dp[i] = pre+1;
                pre = dp[i];
                map[s[i]] = i;

        }
        //最长子串字符长度，默认为0
        int res = 0;
        for(int i = 1;i<n;i++)
        {
            if(dp[i]>res)
                res = dp[i];
        }
        //删除dp
        delete []dp;
        dp = NULL;
        return res;
    }
```

```
};
int main()
{
    string s("aabcd");
    LongestSubstring L;
    cout<<L.longestSubstring(s,5)<<endl;
    return 0;
}
```

## 4，队列和栈（11/14-11/15)

### A，栈和队列的基本性质：

1，栈是先进后出的，队是先进先出的。

2，栈和队列在实现结构上可以有数组和链表两种形式。

数组实现比较容易，用链表的话结构较复杂，因为牵扯到很多指针操作。

### B，栈的一些操作（时间复杂度，O（1））：

1，pop：从栈顶弹出一个元素

2，top | peek：只访问不弹出。

3，push：从栈顶压入一个元素

4，size：返回当前栈内元素个数

### C，队的一些操作（时间复杂度，O（1））：

大体与栈的一致，唯一不同的是以下两点，？？

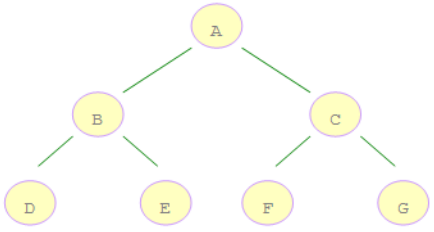
1，push：是从队尾压入一个元素

2，pop：从队头弹出一个元素。

3，此外，还要一种双端队列结构，其首尾都可以压入和弹出元素

4，还有一种优先级队列，其弹出的顺序可以根据队内的优先级，但优先级队列的结构是堆结构，而不是线性结构。

### D，与两种结构有关的图遍历操作




#### 1，深度优先遍历DFS，用栈实现

对于一颗二叉树，深度优先搜索(Depth First Search)是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。以上面二叉树为例，深度优先搜索的顺序为：ABDECFG。怎么实现这个顺序呢 ？

首先，深度优先搜索的顺序，就是元素入栈的顺序。假设我们这里有一个栈。

1，首先，我们将ABD入栈，此时栈顶元素D是最深的，弹出。

2，此时栈顶元素B，还有一个没有访问过得节点E，入栈，是最深的，再弹出

3，此时栈顶元素B，已经没有被访问过得节点，弹出。

4，此时栈顶元素A，还有一个没访问过得元素，CF入栈，此时栈顶F是最深元素，弹出

5，此时栈顶元素为C，还有一个没访问过得元素G，入栈，此时栈顶G是最深元素弹出

6，弹出此时栈顶元素C，再弹出新的栈顶元素A。

#### 2，宽度优先遍历BFS，用队实现

广度优先搜索(Breadth First Search),又叫宽度优先搜索或横向优先搜索，是从根结点开始沿着树的宽度搜索遍历，上面二叉树的遍历顺序为：ABCDEFG.可以利用队列实现广度优先搜索。

首先，所有元素入队列的顺序，就是我们宽度优先遍历的顺序。这里我们的步骤为:

1, A入队列，A出对列

2，把A的下级节点，B，C依次放入队列，然后B出队列，将B的下级节点D，E依次放入队列。

3，然后C出队列，将c的下级节点FG依次放入队列。

4，然后是DEFG依次出队列。

### E，递归与函数栈：

递归的 过程可以看做函数依次进入函数栈的过程。所有递归做的过程一定可以用非递归实现。

### F1，

#### 问题：

实现一个特殊的栈，在实现栈的基础功能的基础上，再实现返回栈中最小元素的操作getmin。

要求：

1，pop，push，getMin操作的时间复杂度都为O（1）

2，设计的栈类型可以使用现成的栈结构。

#### 解题思路：

这里我们使用两个栈，一个栈是stackData（保存当前栈中的元素），一个栈是stackMin（保存每一步的最小值）

方法1：

举个例子，这里我们要依次压入3 4 5 1 2 1，

1，对于stackData，每步正常压入即可。但只有当前数小于等于stackmin的栈顶时，才将元素压入stackMin。

2，弹出时，先在stackData中弹出栈顶元素，记为value，然后判断其与stackMin栈顶元素的大小比较，如果其等于stackmin的栈顶元素时，stackmin弹出栈顶元素。否则，stackmin不弹出

这样一来，每步操作后，stackmin仍能同步记录栈内元素的最小值

方法2：

1，对于stackData，每步正常压入即可，如果当前元素小于stackMin的栈顶，将元素压入stackMin，否则，就重复压入stackmin的栈顶。

2，弹出时两者同步弹出即可。

两者的时间复杂度都为O（1），空间复杂度都为O（N），区别在于方案1里面压入稍省空间，弹出略费时间。方案2压入时略费空间，弹出稍省时间。

code：

```
/*
        问题:
            实现一个特殊的栈，在实现栈的基础功能的基础上，再实现返回栈中最小元素的操作getmin。
            要求:
                1，pop，push，getMin操作的时间复杂度都为O（1）
                2，设计的栈类型可以使用现成的栈结构。
            解题思路:
            这里我们使用两个栈，一个栈是stackData（保存当前栈中的元素），一个栈是stackMin（保存每一步的最小值）
            方法1:
                举个例子，这里我们要依次压入3 4 5 1 2 1，
                1，对于stackData，每步正常压入即可。但只有当前数小于等于stackmin的栈顶时，才将元素压入stackMin。
                2，弹出时，先在stackData中弹出栈顶元素，记为value，然后判断其与stackMin栈顶元素的大小比较，如果其等于stackmin的栈顶元素时，
stackmin弹出栈顶元素。否则，stackmin不弹出
                这样一来，每步操作后，stackmin仍能同步记录栈内元素的最小值
            方法2:
                1，对于stackData，每步正常压入即可，如果当前元素小于stackMin的栈顶，将元素压入stackMin，否则，就重复压入stackmin的栈顶。
                2，弹出时两者同步弹出即可。
            两者的时间复杂度都为O（1），空间复杂度都为O（N），区别在于方案1里面压入稍省空间，弹出略费时间。方案2压入时略费空间，弹出稍省时
间。

*/
#include<iostream>
#include<string>
#include<stack>//pop,top,push
using namespace std;
class QueryMinStack
{
public:
    //定义两个栈，一个是源栈，一个是最小值栈
    stack<int> source_stack,min_stack;
    //重新定义push函数，传入：要入栈的元素
    void push(int value)
    {
        //直接调用栈自身带的函数
        source_stack.push(value);
        //最小值栈为空或者当前值小于最小值栈栈顶元素
        if(min_stack.empty()||min_stack.top()>=value)
            min_stack.push(value);
        //否则就放入最小值栈的栈顶元素
        else
            min_stack.push(min_stack.top());
    }

    //重新定义pop函数，源栈和最小值栈都要同步pop出栈顶,这里直接调用栈本身的pop方法
    void pop()
    {
        source_stack.pop();
        min_stack.pop();
    }
    //重新定义top函数,返回栈顶元素
    int top()
    {
        return source_stack.top();
    }
    //定义min函数，返回栈中最小元素
    int min()
    {
        return min_stack.top();
    }
};

int main()
{
    QueryMinStack q;
    q.push(3);
    q.push(4);
```

```
    q.push(5);
    int c = q.min();
    cout<<c<<endl;
    return 0;
}
```

F2，

问题：

编写一个类，只能用两个栈结构实现队列，支持队列的基本操作，add，poll，peek

解题思路：

队是先进先出，栈是先进后出。两个栈组合起来，刚好可以实现先进后出。

一个栈作为压入栈，stackPush，压入时只往这个栈压.还有一个栈作为弹出栈，stackPop，只从这个栈里拿数据。我们只需将stackPush中的内容，一次性的倒入stackPop中即可（如果stackPop不为空，则不能倒数据）。

code：

```
/*
        问题:
            编写一个类，只能用两个栈结构实现队列，支持队列的基本操作，add，poll，peek
        解题思路:
            队是先进先出，栈是先进后出。两个栈组合起来，刚好可以实现先进后出。
            一个栈作为压入栈，stackPush，压入时只往这个栈压.还有一个栈作为弹出栈，stackPop，只从这个栈里拿数据。我们只需将stackPush中的内容，一次性的倒入stackPop中即可（如果stackPop不为空，则不能倒数据）。

*/
#include<iostream>
#include<string>
#include<stack>
#include<vector>
#include<stdlib.h>
using namespace std;
class TwoStack2Queue
{
public:
    //定义两个栈，一个用来压入，一个用来弹出
    stack<int> stack_push,stack_pop;

    //传入一个向量和它的长度
    vector<int> twoStack(vector<int> ope,int n)
    {
        vector<int> res;
        int i;
        for(i = 0; i < n;i++)
        {
            //如果向量中第i个元素大于等于0，则直接push入stack_push栈，
            if(ope[i]>=0)
                push(ope[i]);
                //res.push_back(pop());
            if(ope[i]<0)
                exit(-1);
        }
        //将push栈内容一次性放入pop，并返回栈pop的栈顶元素，存入res
        res.push_back(pop());
        //将pop栈中的所有元素pop出，并存入res
        while(!stack_pop.empty())
        {
            res.push_back(pop());
        }
        //返回跟入栈顺序相同的向量
        return res;
    }

    //重定义push
    void push(int value)
    {
        stack_push.push(value);
    }

    //重定义pop
    int pop()
    {
        //倒入的前提，一次性倒入，pop栈为空
        if(stack_pop.empty())
            while(!stack_push.empty())
            {
                //这里用的是栈自带的函数
                stack_pop.push(stack_push.top());
                stack_push.pop();
            }
            //pop出栈顶元素
            int res = stack_pop.top();
            stack_pop.pop();
```

```
            return res;
        }

    };
    int main()
    {
        int a[6] = {1,2,3,0,4,0};
        vector<int> arr(a,a+6);
        TwoStack2Queue T;
        vector<int> res = T.twoStack(arr,6);
        for(vector<int>::iterator iter = res.begin();iter!=res.end();iter++)
            cout<<*iter<<" ";
        cout<<endl;
        return 0;
    }
```

F3，

问题：

实现一个栈的逆序，但只能用递归函数和这个栈本身的操作来实现，不能自己申请另外的数据结构。

解题思路：

首先实现一个递归函数，get().用来移除栈底的元素并返回。其思想是，

1,如果有元素1 ，2 ，3.第一层递归函数弹出栈顶元素1，然后将栈传入第二层递归函数。

2，第二层递归函数弹出栈顶2，然后将栈传给第三层递归函数

3，第三层递归函数弹出栈顶3，然后发现栈空了，就返回3做为第三层递归函数的结果

4，第二层递归函数，得到第三层递归函数的返回值，然后将自身弹出的值压入原来的栈，并将第三层的递归函数返回值再传给第一层

5，第一层同样的将自身的内容压入栈，并返回第三层的递归函数返回值作为整个函数的返回值。

然后再实现一个递归函数，reverse（ ）。用于整体的逆序

1，如果有元素1,2,3，第一层递归函数调用get方法，弹出并返回栈底元素3，此时栈变成【1,2】

2，第二层递归函数同样调用get，返回2，此时栈变成【1】

3，第三层函数调用get，返回1，此时栈空

4，然后第三层栈空之后，将返回值压入栈，并将栈传给第二层

5，第二层同样的将返回值压入栈，再将栈传给第一层

6，第一层同样的将返回值压入栈，这样就得到了 3 2 1的逆序栈。

code：

```
/*
        问题：
            实现一个栈的逆序，但只能用递归函数和这个栈本身的操作来实现，不能自己申请另外的数据结构。
        解题思路：
            首先实现一个递归函数，get(). 用来移除栈底的元素并返回。其思想是，
                1, 如果有元素1 ，2 ，3.第一层递归函数弹出栈顶元素1，然后将栈传入第二层递归函数。
                2, 第二层递归函数弹出栈顶2，然后将栈传给第三层递归函数
                3, 第三层递归函数弹出栈顶3，然后发现栈空了，就返回3做为第三层递归函数的结果
                4, 第二层递归函数，得到第三层递归函数的返回值，然后将自身弹出的值压入原来的栈，并将第三层的递归函数返回值再传给第一层
                5, 第一层同样的将自身的内容压入栈，并返回第三层的递归函数返回值作为整个函数的返回值。
            然后再实现一个递归函数，reverse（）。用于整体的逆序
                1, 如果有元素1,2,3，第一层递归函数调用get方法，弹出并返回栈底元素3，此时栈变成【1,2】
                2, 第二层递归函数同样调用get，返回2，此时栈变成【1】
                3, 第三层函数调用get，返回1，此时栈空
                4, 然后第三层栈空之后，将返回值压入栈，并将栈传给第二层
                5, 第二层同样的将返回值压入栈，再将栈传给第一层
                6, 第一层同样的将返回值压入栈，这样就得到了 3 2 1的逆序栈。
*/
#include<iostream>
#include<string>
#include<vector>
#include<stack>
#include<stdlib.h>
using namespace std;
class ReverseStack
{
public:
    vector<int> reverseStack(vector<int> A,int n)
    {
        stack<int> sta;
        int i;
        //将向量A中的元素逆序放入栈sta，向量123变为栈（顶）123（底）
        for(i = n-1;i>=0;i--)
            sta.push(A[i]);
        //对栈sta进行逆序，栈123变为321
        rStack(sta);
        vector<int> res;
        //将逆序后的sta，里面的元素，依次放入新的向量res,321
        while(!sta.empty())
        {
            res.push_back(sta.top());
            sta.pop();
        }
```

```
        //返回逆序的向量res
        return res;

    }

    //stack整体的逆序
    void rStack(stack<int> &A)
    {
        if(A.empty())
            return;
        //获取栈底元素3，同时栈变为12
        int res1 = Get(A);
        //递归
        rStack(A);
        //递归到栈空，就将最后一次的递归返回值压入栈
        A.push(res1);
    }

    //get函数：移除栈底元素并返回
    int Get(stack<int> &A)
    {
        if(A.empty())
            exit(-1);
        //获取栈顶元素
        int res1= A.top();
        A.pop();
        //如果获取后栈为空，直接返回
        if(A.empty())
            return res1;
        //否则的话，递归
        else
        {
            //将上一层递归的结果作为返回值返回
            int res2 = Get(A);
            //并将自身弹出的元素压回栈
            A.push(res1);
            //返回上一层递归的结果
            return res2;
        }
    }
};

int main()
{
    int a[4] = {4,3,2,1};
    vector<int> arr(a,a+4),res;
    ReverseStack R;
    res = R.reverseStack(arr,4);
    for(vector<int>::iterator iter = res.begin();iter!=res.end();iter++)
        cout<<*iter<<" ";
    cout<<endl;
    return 0;
}
```

F4，

问题：

一个栈中元素为整型，现想将该栈从顶到底按从大到小排序， 只许申请一个栈，除此之外可以申请新的变量，但不能申请额外的数据结构，如何完成排序。

解题思路：

将需要排序的栈记为stack，辅助的栈记为help。

1，在stack上执行pop操作，弹出的元素记为current，如果current《=当前help栈顶元素，则直接压入help中。如果current》当前help栈顶元素，则先将current放一边，先将help中的元素逐渐弹出，并且重新压回到stack中，直到current《=当前help的栈顶元素，此时放入current

2，最后，将help中的元素重新压回stack，即可完成从大到小的排序。

code：

```
/*      问题：
            一个栈中元素为整型，现想将该栈从顶到底按从大到小排序， 只许申请一个栈，除此之外可以申请新的变量，但不能申请额外的数据结构，如何完成排序。
        解题思路：
            将需要排序的栈记为stack，辅助的栈记为help。
            1，在stack上执行pop操作，弹出的元素记为current，如果current《=当前help栈顶元素，则直接压入help中。如果current》当前help栈顶元素，则先将current放一边，先将help中的元素逐渐弹出，并且重新压回到stack中，直到current《=当前help的栈顶元素，此时放入current
            2，最后，将help中的元素重新压回stack，即可完成从大到小的排序。



*/
#include<iostream>
#include<stack>
#include<string>
#include<vector>
```

```
using namespace std;
class StackSort
{
public:
    vector<int> stackSort(vector<int> v)
    {
        stack<int> s;
        //将传入的向量，逆序放入栈s
        for(vector<int>::reverse_iterator riter = v.rbegin();riter!=v.rend();riter++)
            s.push(*riter);
        //对栈进行排序
        stSort(s);
        //将排好序的内容存入向量res
        vector<int> res;
        while(!s.empty())
        {
            res.push_back(s.top());
            s.pop();
        }
        return res;
    }

    void stSort(stack<int> &s)
    {
        stack<int> help;
        // 在s上执行pop操作，弹出的元素记为current，如果current《=当前help栈顶元素，则直接压入help中。如果current》当前help栈顶元>素，则先将current放一边，先将help中的元素逐渐弹出，并且重新压回到stack中，直到current《=当前help的栈顶元素，此时放入current
        while(!s.empty())
        {
            int current = s.top();
            s.pop();
            if(help.empty()||current<=help.top())
                help.push(current);
            else
            {
                //如果help非空，且当前值大于help的栈顶元素时，就将help的栈顶元素压回s，
                while(!help.empty()&&current>help.top())
                {
                    s.push(help.top());
                    help.pop();
                }
                help.push(current);
            }
        }
        //将栈help里面的内容重新压回栈s
        while(!help.empty())
        {
            s.push(help.top());
            help.pop();
        }
    }
};
int main()
{
    int a[5] = {1,2,5,4,3};
    vector<int> v(a,a+5),res;
    StackSort S;
    res = S.stackSort(v);
    for(vector<int>::iterator iter = res.begin();iter != res.end();iter++)
        cout<<*iter<<" ";
    cout<<endl;
    return 0;
}
```

F5，

问题：

有一个整形数组arr和一个大小为W的窗口，从数组的最左边滑到最右边。窗口每次往右滑动一个位置。返回一个长度为n-w+1的数组res，res[i]表示每一种窗口状态下的最大值。如有数组 4 3 5 4 3 3 6 7，w=3，则第一个窗口最大值为5。

解题思路：

普通的解法时间复杂度为O（n*w),也就是每次对每个窗口遍历其中的w个数，选出最大值。

最优的解法，时间复杂度为O（n），本题的关键在于使用双端队列这样一个结构，设有一个双端队列qMax。双端队列中存贮的是数组中元素的下标。假设当前元素值为arr[i]，放入规则为：

1，如果qmax为空，则直接放入

2，如果qmax非空，取出当前qmax队尾存放的下标j。如果arr[j]>arr[i]，直接把下标i放入qmax的队尾（ai比较小，继续放即可）

3，如果arr[j]<=arr[i],则一直从qmax队尾弹出下标，直到某个下标的对应值大于arr[i],把i放入qmax队尾。（ai比较大，弹出之前较小的，放入大的。

取出规则：

1，如果qmax队头的下标等于i-w，（相当于已经过期了（窗口滑动时该元素已经过了窗口），弹出）弹出qmax当前队头下标

整体时间复杂度为O（n）

code：

```
/*
        问题：
            有一个整形数组arr和一个大小为W的窗口，从数组的最左边滑到最右边。窗口每次往右滑动一个位置。返回一个长度为n-w+1的数组res，res[i]表示每一种窗口状态下的最大值。如有数组 4 3 5 4 3 3 6 7，w=3，则第一个窗口最大值为5。


        解题思路：
            普通的解法时间复杂度为O（n*w),也就是每次对每个窗口遍历其中的w个数，选出最大值。
            最优的解法，时间复杂度为O（n），本题的关键在于使用双端队列这样一个结构，设有一个双端队列qMax。双端队列中存贮的是数组中元素的下标。假设当前元素值为arr[i]，放入规则为：
                1，如果qmax为空，则直接放入
                2，如果qmax非空，取出当前qmax队尾存放的下标j。如果arr[j]>arr[i]，直接把下标i放入qmax的队尾（ai比较小，继续放即可）
                3，如果arr[j]<=arr[i],则一直从qmax队尾弹出下标，直到某个下标的对应值大于arr[i],把i放入qmax队尾。（ai比较大，弹出之前较小的，放入大的。
                取出规则：
                1，如果qmax队头的下标等于i-w，（相当于已经过期了（窗口滑动时该元素已经过了窗口），弹出）弹出qmax当前队头下标
        整体时间复杂度为O（n）


*/


#include<iostream>
#include<vector>
#include<string>
#include<deque>
using namespace std;


class SlideWindowMax
{
public:
    //传入：向量，向量长度，窗口大小
    vector<int> slide(vector<int> v,int n, int w)
    {
        if(w == 1)
            return v;
        //定义双端队列,存贮向量v的下标
        deque<int> deq;
        //定义向量，存贮每个窗口的最大值
        vector<int> res;
        int i;
        for(i = 0;i < n;i++)
        {
            //如果双端队列为空，或者其队尾元素对应的值大于当前位置对应的元素值,直接放入当前位置即可
            if(deq.empty()||v[deq.back()]>v[i])
                deq.push_back(i);
            //
            else
            {
                //否则，如果队列非空且队尾元素对应的值小于当前位置对应的元素值，那么可以直接将队尾元素pop掉，因为没啥用了
                while(!deq.empty()&&v[deq.back()]<=v[i])
                    deq.pop_back();
                deq.push_back(i);
            }
            //如果队头元素过了窗口有效期，pop掉
            while((i-deq.front())>=w)
                deq.pop_front();
            if(i<w-1)
                continue;
            //将双端队列的队头元素对应的值push进最大值向量res，因为按照之前的规则，队头元素对应的值是大于后面元素对应的值的。
            res.push_back(v[deq.front()]);
        }
        //返回最大值向量
        return res;


    }
};


int main()
{
    int a[8] = {4,3,5,4,3,3,6,7};
    SlideWindowMax S;
    vector<int> arr(a,a+8),res;
    res = S.slide(arr,8,3);
    for(vector<int>::iterator iter = res.begin();iter!=res.end();iter++)
        cout<<*iter<<" ";
    cout<<endl;
    return 0;
}
```

F6，

问题：

给定一个没有重复元素的数组arr，写出生成这个数组的maxTree函数，要求如果数组长度为N，则时间复杂度为O（N），额外空间复杂度为

O（N），maxTree的概念如下：

1，maxtree是一颗二叉树，数组的每一个值对应一个二叉树节点

2，包括maxtree在内且在其中的每一颗子树上，值最大的节点都是树的头

解题思路：

假设数组是 3 4 5 1 2，对每个元素，我们先列出一个表，分别来统计每个元素，左边第一个大的数，和右边第一个大的数

| 左边第一个大的数 | | | | 右边第一个大的数 |
|---|---|---|---|---|
| 3->null | | | | 3->4 |
| 4->null | | | | 4->5 |
| 5->null | | | | 5->null |
| 1->5 | | | | 1->2 |
| 2->5 | | | | 2->null |

构建规则：

1，每个元素的父节点，是左边第一个比它大的数和右边第一个大的数中较小的那个。

2，如果一个元素是整个数组的最大值，则作为树的头结点出现

3，得到的二叉树为

```
          5
     4         2
3                   1
```

这里我们用栈来寻找每个元素左边第一个大的值和右边第一个大的值

如有元素3 1 2

3入栈时，栈空，可以直接入栈

1入栈时，1<3,直接入栈，此时，1左边第一个大的数为3

然后2入栈，1弹出，此时得到1右边第一个最大的数为2

这样类似的，得到每个元素的左边第一个大的数和右边第一个大的值。

code：

```
/*
    问题：
        给定一个没有重复元素的数组arr，写出生成这个数组的maxTree函数，要求如果数组长度为N，则时间复杂度为O（N），额外空间复杂度为
O（N），maxTree的概念如下：
            1，maxtree是一颗二叉树，数组的每一个值对应一个二叉树节点
            2，包括maxtree在内且在其中的每一颗子树上，值最大的节点都是树的头
    解题思路：
        假设数组是 3 4 5 1 2，对每个元素，我们先列出一个表，分别来统计每个元素，左边第一个大的数，和右边第一个大的数
            左边第一个大的数                右边第一个大的数
            3->null             3->4
            4->null             4->5
            5->null             5->null
            1->5            1->2
            2->5            2->null
        构建规则：
            1，每个元素的父节点，是左边第一个比它大的数和右边第一个大的数中较小的那个。
            2，如果一个元素是整个数组的最大值，则作为树的头结点出现
            3，得到的二叉树为
                        5
                    4                   2
                3                                   1
            这里我们用栈来寻找每个元素左边第一个大的值和右边第一个大的值
            如有元素3 1 2
            3入栈时，栈空，可以直接入栈
            1入栈时，1<3,直接入栈，此时，1左边第一个大的数为3
            然后2入栈，1弹出，此时得到1右边第一个最大的数为2
            这样类似的，得到每个元素的左边第一个大的数和右边第一个大的值。
*/
#include<iostream>
#include<string>
#include<stack>
#include<vector>

using namespace std;

class Array2MaxTree
{
public:
    vector<int> buildMaxTree(vector<int> v,int n)
    {
        stack<int> s;
        //左边第一个大的值
        vector<int> left,res;

        int i;
        //得到每个元素值的左边第一个大的数
        for(i = 0;i<n;i++)
        {
            //如果栈非空且当前元素大于栈顶，则弹出栈顶，直到当前元素的值小于栈顶，方可入栈
            while(!s.empty()&&v[s.top()]<=v[i])
                s.pop();
            //计算要入栈的i的左边第一个大的数，正常是当前的栈顶元素，否则的是-1
            if(s.empty())
                left.push_back(-1);
            else
```

```
                left.push_back(s.top());
            //i入栈，如果入栈前栈空，则左边第一个大的数为空，否则为入栈前的栈top
            s.push(i);
        }
        stack<int> st;
        vector<int> right(n,0);
        //得到每个元素值的右边第一个大的数，没有的话是-1
        for(i = n-1;i>=0;i--)
        {
            while(!st.empty()&&v[st.top()]<=v[i])
                st.pop();
            if(st.empty())
                right[i] = -1;
            else
                right[i] = st.top();
            st.push(i);
        }
        //res存放的是每个元素的父节点位置，取左右第一个最大值中较小的那个位置
        for(i = 0; i <n;i++)
        {
            //cout<<right[i]<<endl;
            //cout<<left[i]<<endl;
            //跟节点，
            if(right[i]==-1&&left[i] ==-1)
                res.push_back(-1);
            //选不为空的那个
            else if(right[i] == -1 || left[i] == -1)
                res.push_back(right[i]>left[i]?right[i]:left[i]);
            //选左右第一个最大值中较小的那个
            else
            {
                if(v[right[i]]>v[left[i]])
                    res.push_back(left[i]);
                else
                    res.push_back(right[i]);
            }
        }
        //res里存放的是传入向量元素的父节点坐标，比如原来向量为{3,1,4,2}，这里得到的res为2 0 -1 2，可以看到4的父节点为-1，即为根节点，3,2,对应的父节点都为2位置上的值，即4，然后1对应的父节点是0位置上的值，即3.
        return res;
    }
};
int main()
{
    int a[4] = {3,1,4,2};
    Array2MaxTree A;
    vector<int> v(a,a+4),res;
    res = A.buildMaxTree(v,4);
    for(vector<int> ::iterator iter = res.begin();iter!= res.end();iter++)
        cout<<*iter<<" ";
    cout<<endl;
    return 0;
}
```

## 5，链表（11/15-11/17）

### A,链表的知识点

1，在面试中，难度一般不高，主要考察指针等代码实现能力

2，与数组的区别：

a，数组在内存中是一块连续的存贮空间，可以直接用下标索引；

b，链表空间不一定连续，是临时分配的，需要用next指针找到下一个节点的位置，因此，不能直接访问第i个节点的位置，只能从头部开始，一个一个的来访问。

3，链表的分类：

a，单链表：只能通过next指针指向自己的下一个节点

b，双链表：除了有一个next指针外，还有一个pre指针，指向自己的上一个节点。

c，普通链表 | 循环链表：循环链表的的最后一个节点next指针指向头结点。对于循环双链表来说，除了最后一个元素的next指针指向头节点之外，头结点的pre指针也指向尾节点。

### B，链表问题代码实现的关键点。

1，链表调整函数的返回值类型，往往要求是节点类型

2，在调整链表的时候，需要注意哪些指针变化了，最好先采用画图的方式理清思路。同时注意在调整的时候要保存调整前的环境。

3，链表问题对边界问题处理严格，比如头节点，空节点，尾节点。

### C，链表的插入和删除的注意事项

1，注意特殊情况，链表为空，或长度为1.

2，插入的过程，要找到插入位置的前一个节点和后一个节点，要先保存前一个节点指向后一个节点的指针，然后将前一个节点的指针指向新的节点，将新插入节点的指针，指向后一个节点

3，删除的过程，同样找到要删除元素的前一个元素和后一个元素，先保存要删除的节点指向下一个节点的指针，然后删除该节点，将前一个元素的指针，指向下一个节点。

4，注意，如果在头和尾部插入或删除节点，在设计的时候，要考虑空节点的情况。双链表的插入和删除与单链表类似，只是除了next指针的操作外，还要考虑pre指针。

## D，单链表的翻转操作

1，注意特殊情况，链表为空，或者长度为1，

2，先找到要翻转节点的前后节点，首先保存指向后节点的指针，然后将next指向前一个节点，并将next设置为翻转部分新的头部。

3，设置完之后，再根据之前的记录，从下一个节点继续操作。

## E，一些注意事项：

1，大量链表问题可以使用额外的数据结构来简化调整过程，比如队列，栈，数组等。

2，但链表问题的最优解往往是不使用其他额外的结构。

## F1，

### 问题：

给定一个整数num，如何在节点值有序的环形链表中插入一个节点值为num的节点，并保证这个环形单链表仍然有序。

### 解题思路：

这里我们给出时间复杂度O（n），空间复杂度O（1）的操作，假定，节点值为num的新节点记为node。

1，如果环形单链表为空，node，自己形成环形链表，即next指针指向自己。然后返回node即可

2，如果链表非空，令变量p指向头结点，current指向下一个节点，然后让p和current同步移动下去。如果遇到p的节点值《=num，c的节点值》=num，则将node插入到这里。

3，插入node，然后返回head即可。

4，如果p和c转一圈都没发现合适的位置，那么，将node插入头结点的前面。这是因为，要么node的值比链表中所有节点的值都大，要么是都小。但这种情况下，如果插入的元素比链表中的都大时，直接返回头结点，但如果都小的话，返回新的节点

### code：

```
/*
        问题:
            给定一个整数num，如何在节点值有序的环形链表中插入一个节点值为num的节点，并保证这个环形单链表仍然有序。
        解题思路:
            这里我们给出时间复杂度O（n），空间复杂度O（1）的操作，假定，节点值为num的新节点记为node。
            1，如果环形单链表为空，node，自己形成环形链表，即next指针指向自己。然后返回node即可
            2，如果链表非空，令变量p指向头结点，current指向下一个节点，然后让p和current同步移动下去。如果遇到p的节点值《=num，c的节点值》=num，则将node插入到这里。
            3，插入node，然后返回head即可。
            4，如果p和c转一圈都没发现合适的位置，那么，将node插入头结点的前面。这是因为，要么node的值比链表中所有节点的值都大，要么是都小。但这种情况下，如果插入的元素比链表中的都大时，直接返回头结点，但如果都小的话，返回新的节点

*/
/*
int a = 10;
int *p = &a;
cout<<*p<<endl;
*/
#include<iostream>
#include<string>
#include<stack>
#include<vector>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};
class InsertOrderedRing
{
public:
    //传入，向量A，向量next和要插入的节点值,返回，链表的头节点
    ListNode* insert(vector<int> A, vector<int> nxt, int val)
    {
        //如果待插入的向量A为空，那么头结点设置为要插入的节点，返回即可
        if(A.empty())
        {
            ListNode *Head = new ListNode(val);
            return Head;
        }
        //设置头节点和指向头结点的指针
        ListNode *Head = new ListNode(A[0]),*p = Head;
        //nxt向量里存贮的是A元素中下一个节点的位置，通过这种方法，将A向量构造成一个环链表
        for(vector<int>::iterator iter = nxt.begin();iter!=nxt.end();iter++)
        {
            if(*iter==0)
            {
                (*p).next = Head;
                break;
            }
            (*p).next = new ListNode(A[*iter]);
            p = (*p).next;
        }
```

```
        //pre指针和now指针，用来遍历整个有序环形链表。
        ListNode *pre = Head,*now = (*Head).next;
        while(now != Head)
        {
            if((*pre).val<=val&&(*now).val>=val)
            {
                ListNode *p = new ListNode(val);
                (*p).next = now;
                (*pre).next = p;
                break;
            }
            //否则的话继续往后轮，直到now = head处，满了一圈，停止
            else
            {
                pre = now;
                now = (*now).next;
            }
        }
        //如果转了一圈后，还没有break，说明这个要插入的值，要么大于环内所有值，要么小于环内所有值，将其插入到head的前面
        if(now == Head)
        {
            ListNode *p = new ListNode(val);
            (*p).next = now;
            (*pre).next = p;
            //如果插入的值比原来的头节点值还小的话，则将新插入的点作为头结点
            if(val < (*Head).val)
                Head = p;
        }
        //用指针pre和指针p转一圈，直至p遍历至head，而pre遍历到head的前一个节点
        pre = Head,p = (*Head).next;
        while(p!=Head)
        {
            pre = p;
            p = (*p).next;
        }
        //设置尾节点
        (*pre).next = NULL;
        return Head;
    }
};
int main()
{
    int a[7]= {3,3,7,7,8,12,13},nxt[7] = {1,2,3,4,5,6,0};
    vector<int> arr(a,a+7),next(nxt,nxt+7);
    InsertOrderedRing I;
    ListNode* Head = I.insert(arr,next,11);
    ListNode *p = (*Head).next;
    //打印链表头节点
    cout<<(*Head).val<<" ";
    //依次打印链表中剩余的节点，到尾节点结束
    while(p != NULL)
    {
        cout<<(*p).val<< " ";
        p = (*p).next;
    }
    cout<<endl;
    return 0;
}
```

F2,

问题：

给定一个链表中的节点node，但不给定整个链表的头结点，如何在链表中删除node，请事先这个函数，要求时间复杂度为O（1）。

解题思路：

如果是双链表还是比较容易的，节点可以通过pre和next节点，找到想要的节点，删掉这个节点，只需前后节点连接一下即可。

对于单链表，比如有单向链表1-》2-》3，头节点时不可知的，只知道要删除的是节点2，这时我们只需将节点2的值变成节点3的值，然后删除节点3即可。但这种方法存在一个问题，就是没有办法删除最后一个节点，

code：

```
/*
        问题：
            给定一个链表中的节点node，但不给定整个链表的头结点，如何在链表中删除node，请事先这个函数，要求时间复杂度为O（1）。
        解题思路：
            如果是双链表还是比较容易的，节点可以通过pre和next节点，找到想要的节点，删掉这个节点，只需前后节点连接一下即可。
            对于单链表，比如有单向链表1-》2-》3，头节点时不可知的，只知道要删除的是节点2，这时我们只需将节点2的值变成节点3的值，然后删除节点3即可。但这种方法存在一个问题，就是没有办法删除最后一个节点，


*/

#include<iostream>
#include<string>
#include<stack>
#include<vector>
using namespace std;
struct ListNode
{
```

```
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL) {}
};
class RemoveFromList
{
public:
    bool removeNode(ListNode* pNode)
    {
        //无法删除最后一个节点
        if((*pNode).next == NULL)
            return false;
        //pre指针指向要删除的节点，p指针指向要删除节点的下一个节点
        ListNode *p = (*pNode).next,*pre = pNode;
        //p非尾节点
        while((*p).next != NULL)
        {
            //将要删除的节点的下一个节点的值赋给当前要删除的节点
            (*pre).val = (*p).val;
            //将pre挪到原来p的位置
            pre = p;
            //原来的p挪到p的后一个节点，这里用while循环，是保证被删除节点后的每个节点都能往前挪一位。
            p = (*p).next;
        }
        //上面结束时，还需要对结尾处做个处理
        (*pre).val = (*p).val;
        (*pre).next = NULL;
        delete p;
        p = NULL;
        return true;

    }

};
int main()
{
    return 0;
}
```

F3，

问题：

给定一个链表的头节点head，再给定一个数num，请把链表调整成节点值小于num的都放在链表的左边，值等于num的节点都放在链表的中间，值大于num的节点，都放在链表的右边。

解题思路：

简单的思路，是将链表中的值都放在一个数组里，然后对数据中的元素做类似快排划分的过程。然后将数组中的元素重新串联。

最优解：在遍历链表的过程中，将链表分为3个小链表，分别是值小于num的节点组成的链表，值等于num的节点组成的链表，和值大于num的节点组成的链表。最后再把这三条链表整体的连接起来

code：

```
/*
        问题：
            给定一个链表的头节点head，再给定一个数num，请把链表调整成节点值小于num的都放在链表的左边，值等于num的节点都放在链表的中间，值大于num的节点，都放在链表的右边。
        解题思路：
            简单的思路，是将链表中的值都放在一个数组里，然后对数据中的元素做类似快排划分的过程。然后将数组中的元素重新串联。
            最优解：在遍历链表的过程中，将链表分为3个小链表，分别是值小于num的节点组成的链表，值等于num的节点组成的链表，和值大于num的节点组成的链表。最后再把这三条链表整体的连接起来

*/
#include<iostream>
#include<string>
#include<stack>
#include<vector>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};
class DivideList
{
public:
    //传入，链表头节点，和要划分的节点值，返回链表头节点
    ListNode* listDivide(ListNode * head,int val)
    {
        ListNode *smallHead = NULL,*smallTail = NULL,*bigHead = NULL,*bigTail=NULL,*p = head,*q = head;
        while(p)
```

```
        {
            q = p;
            //p沿着next往下遍历，当p处的元素值小于等于划分值
            if((*p).val<=val)
            {
                //如果小链表为空，则将其链表头尾都设为p
                if(!smallHead)
                    smallHead = smallTail = p;
                //否则，将p加到小链表尾部，并将p设为链表尾部
                else
                {
                    (*smallTail).next = p;
                    smallTail = p;
                }
            }
            //p处的元素大于划分值
            else
            {
                //如果大链表为空，则将其链表头尾都设为p
                if(!bigHead)
                    bigHead = bigTail = p;
                //否则，将p加到链表尾部，并将p设为链表尾部
                else
                {
                    (*bigTail).next = p;
                    bigTail = p;
                }
            }
            //p沿着next继续往下走
            p = (*p).next;
            //? ?
            (*q).next = NULL;
        }
        //如果小链表非空，则将其链表头设为整合后的链表头，
        if(smallHead)
        {
            head = smallHead;
            //如果大链表非空，整合到小链表的后边
            if(bigHead)
                (*smallTail).next = bigHead;
        }
        else
            if(bigHead)
                head = bigHead;
        return head;
    }
};
class CreateListNode
{
public:
    //根据向量创建链表。
    ListNode* create(vector<int> A)
    {
        if(A.empty())
            return NULL;
        ListNode *Head = new ListNode(A[0]),*p = Head;
        for(vector<int>::iterator iter = A.begin()+1;iter!=A.end();iter++)
        {
            (*p).next = new ListNode(*iter);
            p = (*p).next;
        }
        return Head;
    }
    //打印链表内容
    void Print(ListNode* Head)
    {
        ListNode *p = Head;
        while(p)
        {
            cout<<(*p).val<<" ";
            p = (*p).next;
        }
        cout<<endl;
    }
};
int main()
{
    int a[5] = {1620,1134,861,563,1};
    vector<int> arr(a,a+5);
    CreateListNode C;
    ListNode *Head = C.create(arr);
    DivideList D;
    Head = D.listDivide(Head,1134);
    C.Print(Head);
    return 0;
}
```

F4，

问题：

给定两个有序链表的头结点head1和head2，打印两个有序链表的公共部分。

解题思路：

1，如果两个链表有一个为空，直接返回即可，不可能有公共部分

2，如果两个都非空，从两个链表的头节点开始遍历，如果list1的当前值小于list2的当前值，则继续遍历list1的下一个值。反之，如果list2的值小于list1的值，则继续遍历list2的下一个值。如果两者相等，就打印当前值，并都继续往下移动，

3，list1，list2有一个为空，整个过程就停止。

code：

```
/*
        问题:
            给定两个有序链表的头结点head1和head2，打印两个有序链表的公共部分。
        解题思路:
            1，如果两个链表有一个为空，直接返回即可，不可能有公共部分
            2，如果两个都非空，从两个链表的头节点开始遍历，如果list1的当前值小于list2的当前值，则继续遍历list1的下一个值。反之，如果list2的值小于list1的值，则继续遍历list2的下一个值。如果两者相等，就打印当前值，并都继续往下移动，
            3，list1，list2有一个为空，整个过程就停止。
*/
#include<iostream>
#include<string>
#include<stack>
#include<vector>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};

class Common
{
public:
    vector<int> findCommonParts(ListNode* headA,ListNode* headB)
    {
        vector<int> res;
        ListNode *p = headA,*q = headB;
        //沿着next遍历，任意一个指针指向null即可停止
        while(p&&q)
        {
            //如果碰到相同值，push进res向量，p，q均往后一步
            if((*p).val == (*q).val)
            {
                res.push_back((*p).val);
                p = (*p).next;
                q = (*q).next;
            }
            //p指向的值小，p往前一步，因为链表是有序的，是从小到大的
            else if((*p).val <(*q).val)
                p = (*p).next;
            else
                q = (*q).next;
        }
        return res;
    }
};

class CreateListNode
{
public:
    ListNode* Create(vector<int> A)
    {
        if(A.empty())
            return NULL;
        ListNode *Head=new ListNode(A[0]),*p=Head;
        for(vector<int>::iterator iter=A.begin()+1;iter!=A.end();iter++)
        {
            (*p).next=new ListNode(*iter);
            p=(*p).next;
        }
        return Head;
    }
    void Print(ListNode* Head)
    {
        ListNode *p=Head;
        while(p)
        {
            cout<<(*p).val<<" ";
            p=(*p).next;
        }
```

```
    }
};

int main()
{
    int a[7]={1,2,3,4,5,6,7},b[5] ={2,4,6,8,10};
    vector<int> arr(a,a+7),brr(b,b+5);
    CreateListNode C;
    ListNode *HeadA = C.Create(arr);
    ListNode *HeadB = C.Create(brr);
    Common Co;
    vector<int> res = Co.findCommonParts(HeadA,HeadB);
    for(vector<int>::iterator iter=res.begin();iter!=res.end();iter++)
        cout<<*iter<<" ";
    cout<<endl;
    return 0;
}
```

F5，

问题：

给定一个单链表的头结点head，实现一个调整单链表的函数，使得每K个节点之间逆序，如果最后不够K个节点一组，则不调整最后几个节点。

比如链表，1-》2-》3-》4-》5-》6-》7-》8-》null,K=3

调整后为3-》2-》1-》6-》5-》4-》7-》8-》null，

解题思路：

1，如果链表为空，长度为1，或小于等于2.这些情况都是不用进行处理的。

2，方法1，时间复杂度O（n），空间复杂度O（k），k为每k个节点。

a,用栈处理，让节点依次进栈，凑齐k个元素，就依次从栈中弹出，弹出顺序为 3 2 1，刚好逆序。

b,然后下一组元素做同样的处理，

c,如果凑不齐，则不进行栈操作

d,此外，我们还需要记录每个组的最后一个元素指向的指针，（第一组不用，需特殊处理）

e,最后，因为链表的头节点发生了改变，因此我们最好设计成返回值为节点的函数。

3，方法2，时间复杂度O（n），空间复杂度O（1）

a,方法2与方法1大体相似

b，收集到k个节点就做逆序操作，唯一不同的是不用栈来操作。

c,需要将每组的调整过得头结点和上一组的尾节点相连

code：

```
/*
        问题:
            给定一个单链表的头结点head，实现一个调整单链表的函数，使得每K个节点之间逆序，如果最后不够K个节点一组，则不调整最后几个节点。
            比如链表，1-》2-》3-》4-》5-》6-》7-》8-》null,K=3
            调整后为3-》2-》1-》6-》5-》4-》7-》8-》null,


        解题思路:
            1，如果链表为空，长度为1，或小于等于2.这些情况都是不用进行处理的。
            2，方法1，时间复杂度O（n），空间复杂度O（k），k为每k个节点。
                a,用栈处理，让节点依次进栈，凑齐k个元素，就依次从栈中弹出，弹出顺序为 3 2 1，刚好逆序。
                b,然后下一组元素做同样的处理，
                c,如果凑不齐，则不进行栈操作
                d,此外，我们还需要记录每个组的最后一个元素指向的指针，（第一组不用，需特殊处理）
                e,最后，因为链表的头节点发生了改变，因此我们最好设计成返回值为节点的函数。
            3，方法2，时间复杂度O（n），空间复杂度O（1）
                a,方法2与方法1大体相似
                b，收集到k个节点就做逆序操作，唯一不同的是不用栈来操作。
                c,需要将每组的调整过得头结点和上一组的尾节点相连


*/
//创建链表，或打印链表内容
#include<iostream>
#include<string>
#include<stack>
#include<vector>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next = NULL;
    ListNode(int x):val(x),next(NULL){}
};
class KReverseList
{
public:
    //传入链表头节点和逆序的间隔，返回k逆序过的链表
    ListNode* reverse(ListNode* head,int k)
    {
        //如果链表为空，或者间隔k的值小于等于1,直接返回即可
        if(!head || k<2)
```

```
            return head;
        //指针p指向一个间隔内第一个元素，指针q用来遍历整个链表。和上个间隔的尾节点。
        ListNode *p = head,*q=head,*LastTail = NULL;
        int i = 1,count = 0;
        while(q)
        {
            //每到一个间隔,对于复杂的问题，建议画图辅助
            if(i == k)
            {
                //如果count为0,即前面没有k逆序间隔,这是第一个k逆序，本间隔的k处节点逆序后成为链表的头，
                if(!count)
                    head = q;
                //否则,将上一个k逆序间隔的尾节点指针指向没逆序前的本k间隔的k处节点
                else
                    (*LastTail).next = q;
                //temp1记录当前节点的下一个节点，即下一个k间隔的开始
                ListNode *temp1 = (*q).next,*temp2 = NULL;
                //逆序本间隔的值。其中q指向本间隔最后一个元素，p指向本间隔第一个元素
                while(q!=p)
                {
                    //记录间隔第一个元素的下个节点
                    temp2= (*p).next;
                    //交换后，p变成了本间隔最后一个元素，设置LastTai为p
                    if((*q).next == temp1)
                        LastTail = p;


                    //将p换至间隔尾
                    (*p).next = (*q).next;
                    //q换到p的前面
                    (*q).next = p;
                    //p换为它原来的下一个元素，此时记住间隔的最后一个元素已经设定为p了
                    p = temp2;
                }
                //继续下一个间隔的第一个位置开始
                p = q = temp1;
                //i被重置1
                i = 1;
                //可以理解为间隔编号，第几个间隔
                count++;
            }
            //如果没到末尾，继续往下next
            if(q)
                q = (*q).next;
            //间隔内计数
            i++;
        }
        return head;
    }
};
class CreateListNode
{
public:
    ListNode* Create(vector<int> A)
    {
        if(A.empty())
            return NULL;
        ListNode *Head=new ListNode(A[0]),*p=Head;
        for(vector<int>::iterator iter=A.begin()+1;iter!=A.end();iter++)
        {
            (*p).next=new ListNode(*iter);
            p=(*p).next;
        }
        return Head;
    }
    void Print(ListNode* Head)
    {
        ListNode *p=Head;
        while(p)
        {
            cout<<(*p).val<<" ";
            p=(*p).next;
        }
        cout<<endl;
    }
};
int main()
{
    int a[8]={1,2,3,4,5,6,7,8};
    vector<int> arr(a,a+8);
    CreateListNode C;
    ListNode *HeadA=C.Create(arr);
    KReverseList K;
    ListNode *head = K.reverse(HeadA,3);
    C.Print(head);
    return 0;
}
```

F6，

问题：

给定一个单链表的头结点head，链表中的每个节点保存一个整数，再给定一个值val，把所有等于val的节点删掉

解题思路：

我们把整个过程看做一个构造链表的过程，假设之前的链表的头是head，尾是tail。

1，如果当前节点的值是val，就抛弃它，否则将其接到链表的末尾，如果有新的要接入，直接插到之前值和尾部之间即可，最后设置null为尾部

2，要注意特殊情况，和边界问题。

code：

```
/*
        问题:
            给定一个单链表的头结点head，链表中的每个节点保存一个整数，再给定一个值val，把所有等于val的节点删掉
        解题思路:
            我们把整个过程看做一个构造链表的过程，假设之前的链表的头是head，尾是tail。
            1，如果当前节点的值是val，就抛弃它，否则将其接到链表的末尾，如果有新的要接入，直接插到之前值和尾部之间即可，最后设置null为尾部
2，要注意特殊情况，和边界问题。
*/
#include<iostream>
#include<vector>
#include<string>
#include<stack>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};
class RemoveGivenValue
{
public:
    //传入链表，和要删除的值，返回，删除掉指定值后的链表
    ListNode* remove(ListNode* head,int val)
    {
        if(!head)
            return head;
        ListNode *Head= NULL,*Tail = NULL,*p = head,*temp = NULL;
        while(p)
        {
            temp = (*p).next;
            //如果当前节点的值等于要删除的值，那么删除该节点
            if((*p).val == val)
                delete p;
            //否则，将该节点链接到一个新的链表
            else
            {
                //如果新的链表为空，那么头节点设置为p
                if(!Head)
                    Head = p;
                else
                    //加到链表尾部
                    (*Tail).next = p;
                //将p设为新的Tail
                Tail = p;
                (*Tail).next = NULL;
            }
            //沿着next指针继续往下遍历链表
            p = temp;
        }
        return Head;
    }

};
//创建链表，打印链表
class CreateListNode
{
public:
    ListNode* Create(vector<int> A)
    {
        if(A.empty())
            return NULL;
        ListNode *Head=new ListNode(A[0]),*p=Head;
        for(vector<int>::iterator iter=A.begin()+1;iter!=A.end();iter++)
        {
            (*p).next=new ListNode(*iter);
            p=(*p).next;
        }
        return Head;
    }
    void Print(ListNode* Head)
    {
```

```
        ListNode *p=Head;
        while(p)
        {
            cout<<(*p).val<<" ";
            p=(*p).next;
        }
        cout<<endl;
    }
};
int main()
{
    int a[7]={1,2,3,4,3,2,1};
    vector<int> arr(a,a+7);
    CreateListNode C;
    ListNode *HeadA=C.Create(arr);
    RemoveGivenValue R;
    ListNode *head = R.remove(HeadA,2);
    C.Print(head);
    return 0;
}
```

## F7，

### 问题：

判断一个链表是否为回文结构，如12312，true，1231，false

### 解题思路：

时间复杂度都为O（N），但空间复杂度依次为O（N），O（N）/2， O(1)

1，方法1：

a，申请一个栈结构，在遍历链表的过程中，将节点依次压入栈中，此时，栈中从顶到底的元素顺序就是链表元素的逆序。

b,再次遍历链表，同时同步弹出栈中元素，每次都比对两者元素，如果都相同，则返回true，有一个不相同，返回false。

2，方法2：

a,同样申请一个栈结构，但用快慢两个指针同时遍历，快指针一次走两步，慢指针一次走一步，将慢指针每次遍历的内容压入栈，当快指针走完的时候，慢指针来到链表中间的位置，结束这个过程，此时，栈内有着链表的前半部分元素（中间位置的节点不放入）。

b，接下来，man指针继续往后遍历，栈也同步弹出元素，进行比对，如果每一步都相等，则返回true，否则，返回false

3，方法3：

a，先找到链表中间的位置，然后对其右半部分做逆序，接下来从链表的两头开始，依次对比值是否一样，

b，如果对比到中间位置，仍一样，则返回true。

c，方法3返回之前，需要将链表调整回来。

### code：

```
/*
        问题:
            判断一个链表是否为回文结构，如12312，是，1231，false


        解题思路:
            时间复杂度都为O（N），但空间复杂度依次为O（N），O（N）/2，   O(1)
            1，方法1:
                a，申请一个栈结构，在遍历链表的过程中，将节点依次压入栈中，此时，栈中从顶到底的元素顺序就是链表元素的逆序。
                b,再次遍历链表，同时同步弹出栈中元素，每次都比对两者元素，如果都相同，则返回true，有一个不相同，返回false。
            2，方法2:
                a,同样申请一个栈结构，但用快慢两个指针同时遍历，快指针一次走两步，慢指针一次走一步，将慢指针每次遍历的内容压入栈，当快指针走
完的时候，慢指针来到链表中间的位置，结束这个过程，此时，栈内有着链表的前半部分元素（中间位置的节点不放入）。
                b，接下来，man指针继续往后遍历，栈也同步弹出元素，进行比对，如果每一步都相等，则返回true，否则，返回false
            3，方法3:
                a，先找到链表中间的位置，然后对其右半部分做逆序，接下来从链表的两头开始，依次对比值是否一样，
                b，如果对比到中间位置，仍一样，则返回true。
                c，方法3返回之前，需要将链表调整回来。


*/
#include<iostream>
#include<string>
#include<vector>
#include<stack>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};


class PalindromeInList
{
public:
    //方法3
    bool isPalindrome(ListNode* head)
    {
```

```
        if(!head||!(*head).next)
            return true;
        //定义快慢指针
        ListNode *slow = head,*fast = head;
        //是否为奇数，默认不是
        int odd = 0;
        //指针移动，slow每次移1位，Fast每次移2位
        while(fast->next&&fast->next->next)
        {
            fast = (*(*fast).next).next;
            slow = (*slow).next;
        }
        //如果为奇数，则fast遍历到最后刚好是尾节点的。
        if(!fast->next)
            odd = 1;
        //用于链表下半部分的指针
        ListNode *pre = slow,*now = (*pre).next,*temp=NULL;
        //得到上半部分，此时slow刚好遍历到链表中间。
        (*slow).next = NULL;
        //遍历链表后半部分，对其做逆序
        while(now)
        {
            temp = (*now).next;
            (*now).next = pre;
            pre = now;
            now = temp;
        }
        //是否是回文结构
        bool res = true;
        //分别从头和尾遍历，此时pre指向的是逆序后的链表的头部,slow是尾部,判断左右两部分链表是否完全一致
        ListNode *fwd = head,*tail = pre;
        while(tail!=slow)
        {
            if((*fwd).val != (*tail).val)
            {
                res = false;
                break;
            }
            fwd=(*fwd).next;
            tail = (*tail).next;
        }
        //返回之前，将链表调整回来，此时pre指向逆序后的链表头部，
        now = pre;
        pre = (*pre).next;
        (*now).next = NULL;
        //将逆序过的链表逆序回来
        while(pre)
        {
            temp = (*pre).next;
            (*pre).next = now;
            now = pre;
            pre = temp;
        }
        return res;
    }
};
//创建链表，打印链表
class CreateListNode
{
public:
    ListNode* Create(vector<int> A)
    {
        if(A.empty())
            return NULL;
        ListNode *Head=new ListNode(A[0]),*p=Head;
        for(vector<int>::iterator iter=A.begin()+1;iter!=A.end();iter++)
        {
            (*p).next=new ListNode(*iter);
            p=(*p).next;
        }
        return Head;
    }
    void Print(ListNode* Head)
    {
        ListNode *p=Head;
        while(p)
        {
            cout<<(*p).val<<" ";
            p=(*p).next;
        }
        cout<<endl;
    }
};
int main()
{
    int a[5]={1,2,3,2,1};
    vector<int> arr(a,a+5);
```

```
    CreateListNode C;
    ListNode *HeadA=C.Create(arr);
    PalindromeInList P;
    bool res = P.isPalindrome(HeadA);
    cout<<res<<endl;
    C.Print(HeadA);
    return 0;
}
```

F8，

问题：

一个链表结构中，每个节点不仅含有一条指向下一个节点的next指针，同时还有一条rand指针，rand指针可能指向任何一个链表中的节点，请复制这种含有rand指针节点的链表。

解题思路：

这里我们介绍一种不适用其他数据结构的方法，

1，如果链表为空，或长度为0，则直接返回空

2，假设有链表1-》2-》3-》null，其中1的rand到3,2的rand1,3的ran2

3，首先从链表的头结点开始，根据next指针往下遍历，同时拷贝当前的节点，拷贝的节点rand指向null，拷贝的节点被放在当前节点和下一个节点之间。

4，再在新的链表中，遍历一下所有节点，但在遍历的过程中，同时拿到两个节点，比如先拿到1和1的复制，然后根据1的rand找到3，然后3的下一个节点就是3的拷贝，因此，可以成功的将1的拷贝的rand指针指向3的拷贝。

5，然后再拿到2和2的拷贝，通过2的rand指针找到1，同样的，2的拷贝的rand就可以指向1的拷贝。

6，继续遍历，同理，可以找到3的复制的rand指针指向2的拷贝。

7，然后将这个大的链表分流成123和1拷贝2拷贝3拷贝即可。

code：

```
/*
        问题：
            一个链表结构中，每个节点不仅含有一条指向下一个节点的next指针，同时还有一条rand指针，rand指针可能指向任何一个链表中的节点，请复制这种含有rand指针节点的链表。
        解题思路：
            这里我们介绍一种不适用其他数据结构的方法，
            1，如果链表为空，或长度为0，则直接返回空
            2，假设有链表1-》2-》3-》null，其中1的rand到3,2的rand1,3的ran2
            3，首先从链表的头结点开始，根据next指针往下遍历，同时拷贝当前的节点，拷贝的节点rand指向null，拷贝的节点被放在当前节点和下一个节点之间。
            4，再在新的链表中，遍历一下所有节点，但在遍历的过程中，同时拿到两个节点，比如先拿到1和1的复制，然后根据1的rand找到3，然后3的下一个节点就是3的拷贝，因此，可以成功的将1的拷贝的rand指针指向3的拷贝。
            5，然后再拿到2和2的拷贝，通过2的rand指针找到1，同样的，2的拷贝的rand就可以指向1的拷贝。
            6，继续遍历，同理，可以找到3的复制的rand指针指向2的拷贝。
            7，然后将这个大的链表分流成123和1拷贝2拷贝3拷贝即可。


*/
#include<iostream>
#include<string>
#include<vector>
#include<stack>
using namespace std;
struct RandomListNode
{
    int val;
    struct RandomListNode *next,*random;
    RandomListNode(int x):val(x),next(NULL),random(NULL){}
};
class CopyComplexList
{public:
    RandomListNode * Clone(RandomListNode* head)
    {
        //如果头结点为空
        if(!head)
            return head;
        //p用来遍历链表
        RandomListNode *p = head,*temp = NULL,*New = NULL;
        //复制每个节点，并插入（理不清就画图）
        while(p)
        {
            temp = (*p).next;
            //用p节点的值新建一个节点。p的复制
            New = new RandomListNode((*p).val);
            //将新copy的节点插入p和p的下一个节点之间
            (*New).next = (*p).next;
            (*p).next = New;
            //继续遍历p
            p = temp;
        }
        //根据原节点的random指针，获取copy节点的random指针
        p = head,New = (*p).next;
        while(p)
        {
            if((*p).random)
```

```
            (*New).random = (*(*p).random).next;
        //继续下一对元素和它的拷贝
        p = (*New).next;
        if(p)
            New = (*p).next;
    }
    //分离拷贝和源节点
    p = head,New = (*p).next;
    //拷贝链表的头节点
    RandomListNode* NHead = New;
    while((*New).next)
    {
        temp = (*New).next;
        (*p).next = (*New).next;
        (*New).next = (*temp).next;
        //继续往下走
        p = temp;
        New = (*p).next;
    }
    //设置尾节点
    (*p).next = NULL;
    return NHead;
  }
};
int main()
{
    return 0;
}
```

F9，

问题：

如何判断一个单链表是否有环，有环的话返回进入环的第一个节点，无环的话返回为空。 如果链表的长度为N，请做到时间复杂度O（n），额外空间复杂度O（1）。

解题思路：

1，如果没有额外空间复杂度限制，这里我们可以用hash表来做，从头节点开始遍历，每遍历一个节点，就在hash表中记录一下，如果一个链表无环，那么走到结尾处也不会出现重复的元素，此时，直接返回空。如果一个链表有环，则根据hash表，一定可以看到重复遍历到一个节点的情况。那么第一个重复遍历的节点，肯定就是入环的节点。直接返回即可。

2，但在额外空间复杂度为O（1）的限制下，

a，从头结点开始，用快慢两个指针进行遍历，快指针一次走两步，慢指针一次走一步。

b,如果一个链表无环，那么快指针则将快速发现尾，并返回空

c,如果一个链表有环，那么快指针和慢指针迟早会在环内的某个位置相遇，在他们相遇的时刻，让快指针从链表的头部开始，重新遍历，这次一次走1步，同时，慢指针也从相遇的位置开始，继续往下走，一次也走一步，当快慢指针再次相遇时，相遇到的节点即为入环的第一个节点。

code：

```
/*
        问题：
            如何判断一个单链表是否有环，有环的话返回进入环的第一个节点，无环的话返回为空。 如果链表的长度为N，请做到时间复杂度O（n），额外空间复杂度O（1）。
        解题思路：
            1，如果没有额外空间复杂度限制，这里我们可以用hash表来做，从头节点开始遍历，每遍历一个节点，就在hash表中记录一下，如果一个链表无环，那么走到结尾处也不会出现重复的元素，此时，直接返回空。如果一个链表有环，则根据hash表，一定可以看到重复遍历到一个节点的情况。那么第一个重复遍历的节点，肯定就是入环的节点。直接返回即可。
            2，但在额外空间复杂度为O（1）的限制下，
                a，从头结点开始，用快慢两个指针进行遍历，快指针一次走两步，慢指针一次走一步。
                b,如果一个链表无环，那么快指针则将快速发现尾，并返回空
                c,如果一个链表有环，那么快指针和慢指针迟早会在环内的某个位置相遇，在他们相遇的时刻，让快指针从链表的头部开始，重新遍历，这次一次走1步，同时，慢指针也从相遇的位置开始，继续往下走，一次也走一步，当快慢指针再次相遇时，相遇到的节点即为入环的第一个节点。

*/
#include<iostream>
#include<string>
#include<vector>
#include<stack>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};

class ChkRingInList
{
public:
    //返回入环节点的值
    int chkRing(ListNode* head)
    {
        if(!head)
            return -1;
```

```
        //定义快指针和慢指针
        ListNode *slow = head,*fast = head;
        //慢指针一次一步，快指针一次两步，如果相遇，则有环，就跳出
        while(((*fast).next&&(*(*fast).next).next))
        {
            slow = (*slow).next;
            fast = (*(*fast).next).next;
            if(slow == fast)
                break;
        }
        //如果快指针已经到头，说明在之前的过程中一直没有相遇，没有break，那么也是无环
        if(!(*fast).next || !(*(*fast).next).next)
            return -1;
        //这里应该是fast=head吧,找入环节点
        slow = head;
        while(slow != fast)
        {
            slow = (*slow).next;
            fast = (*fast).next;
        }
        //返回再次相遇节点的值
        return (*fast).val;
    }

};
int main()
{
    return 0;
}
```

F10，

问题：

如何判断两个无环单链表是否相交，相交的话返回第一个相交的节点，不相交的话返回空，如果两个链表长度分别为N和M，请做到时间复杂度O(N+M），额外空间复杂度为O（1）。

解题思路：

1，如果没有空间复杂度限制，我们可以用hash表，先遍历第一个链表，将第一个链表的节点都加入hash表中，然后开始遍历第二个链表，一旦发现第二个链表中的节点，在hash表中有记录，则表明该节点在第一个链表中也存在，是第一个相交的节点，如果遍历完了也没有发现如上的情况，则表明两个链表不相交。

2，遍历两个链表，得到两个链表的长度，比如第一个链表的长度,100，第二个链表的长度是50，那么让长度大的链表先走（len1-len2）步，然后再让两个链表一起往下走，如果两者相交的话，那么在他们同步走的时候一定会遍历到相同的节点，如果走到最后都不相交，则两个链表不相交。（不过我觉得这个思想有点问题，万一相交的部分在第一个链表已经走过的部分呢？）

code：

```
/*

    问题：
        如何判断两个无环单链表是否相交，相交的话返回第一个相交的节点，不相交的话返回空，如果两个链表长度分别为N和M，请做到时间复杂度O(N+M)，额外空间复杂度为O（1）。
    解题思路：
        1，如果没有空间复杂度限制，我们可以用hash表，先遍历第一个链表，将第一个链表的节点都加入hash表中，然后开始遍历第二个链表，一旦发现第二个链表中的节点，在hash表中有记录，则表明该节点在第一个链表中也存在，是第一个相交的节点，如果遍历完了也没有发现如上的情况，则表明两个链表不相交。
        2，遍历两个链表，得到两个链表的长度，比如第一个链表的长度,100，第二个链表的长度是50，那么让长度大的链表先走（len1-len2）步，然后再让两个链表一起往下走，如果两者相交的话，那么在他们同步走的时候一定会遍历到相同的节点，如果走到最后都不相交，则两个链表不相交。（不过我觉得这个思想有点问题，万一相交的部分在第一个链表已经走过的部分呢？）

*/
#include<iostream>
#include<string>
#include<vector>
#include<stack>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};
class Check2ListIntersect
{
public:
    bool chkIntersect(ListNode* headA,ListNode* headB)
    {
        //如果链表A，B有任何一方为空
        if(!headA || !headB)
            return false;
        //链表A和B的长度计数
        int NumofA = 0,NumofB=0,count=0;
        ListNode *p = headA;
        while(p)
```

```
        {
            NumofA++;
            p = (*p).next;
        }
        p = headB;
        while(p)
        {
            NumofB++;
            p = (*p).next;
        }
        //长的链表L，短的链表S
        ListNode *L = NULL,*S = NULL;
        if(NumofA > NumofB)
            L = headA,S=headB;
        else
            L= headB,S= headA;
        //长短链表的长度差值
        count = (NumofA > NumofB)?(NumofA-NumofB):(NumofB-NumofA);
        //先让长链表走count长度
        while(count--)
            L = (*L).next;
        while(L&&S)
        {
            //如果两个链表某处节点值相等
            if(L == S)
                return true;
            L = (*L).next;
            S = (*S).next;
        }
        return false;
    }
};

int main()
{
    return 0;
}
```

F11，

问题：

如何判断两个有环单链表是否相交？相交的话返回第一个相交的节点，不相交的话返回空，如果两个链表的长度分别为N和M，请做到时间复杂度O（N+M），空间复杂度O（1）。

解题思路：

1，先根据之前的有环单链表，找到入环节点的题目，找到各自的入环节点。

2，如果两个入环节点是同一个节点，则肯定两者是相交的。但两者在入环节点就相交了，肯定有更早的相交节点。做法跟找到无环单链表的相交节点类似。唯一不同的是，这里的终止位置为两者相同的入环节点。

3，如果两者的入环节点不是同一个节点，有两种情况，链表1和链表2都有独立的环，还有一种情况是，链表1和链表2共享一个大环，但入环节点不同。如何判定就是先从链表1的入环节点开始往下，如果一直遍历到整个环结束还没发现入环节点2，则是第一种情况，直接返回空即可，因为两者不相交。否则，是链表1，链表2共享一个大环，两者相交。返回任意一个链表的入环节点即可。

code：

```
/*
        问题：
            如何判断两个有环单链表是否相交？相交的话返回第一个相交的节点，不相交的话返回空，如果两个链表的长度分别为N和M，请做到时间复杂度O（N+M），空间复杂度O（1）。
        解题思路：
            1，先根据之前的有环单链表，找到入环节点的题目，找到各自的入环节点。
            2，如果两个入环节点是同一个节点，则肯定两者是相交的。但两者在入环节点就相交了，肯定有更早的相交节点。做法跟找到无环单链表的相交节点类似。唯一不同的是，这里的终止位置为两者相同的入环节点。
            3，如果两者的入环节点不是同一个节点，有两种情况，链表1和链表2都有独立的环，还有一种情况是，链表1和链表2共享一个大环，但入环节点不同。如何判定就是先从链表1的入环节点开始往下，如果一直遍历到整个环结束还没发现入环节点2，则是第一种情况，直接返回空即可，因为两者不相交。否则，是链表1，链表2共享一个大环，两者相交。返回任意一个链表的入环节点即可。

*/
#include<iostream>
#include<string>
#include<vector>
#include<stack>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};
class CheckRingListIntersect
{
public:
    bool chkRingListIntersect(ListNode* head1,ListNode* head2)
    {
        //获取链表1,2的入环节点
```

```
    ListNode *entry1 = chkLoop(head1),*entry2 = chkLoop(head2);
    //如果一个有环，一个无环，一定不相交
    if(entry1 == NULL || entry2 == NULL)
        return false;
    //如果入环节点是同一个节点，则肯定相交
    if(entry1 == entry2)
    {
        //获取第一个相交节点
        int Numof1 = 0,Numof2 = 0,count=0;
        //获取链表1到环之前的长度
        ListNode *p = head1;
        while(p!=entry1)
        {
            Numof1++;
            p = (*p).next;
        }
        //获取链表2到环之前的长度
        p = head2;
        while(p != entry1)
        {
            Numof2++;
            p = (*p).next;
        }

        //定义长链表，短链表
        ListNode *L = NULL,*S = NULL;
        if(Numof1>Numof2)
            L = head1,S = head2;
        else
            L = head2,S = head1;
        //获取长短链表的长度差值
        count = (Numof1>Numof2)?(Numof1-Numof2):(Numof2-Numof1);
        //先让长链表走一段
        while(count--)
            L = (*L).next;
        while(L!=entry1&&S!=entry2)
        {
            //相交
            if(L == S)
                break;
            L = (*L).next;
            S = (*S).next;
        }
        //返回相交节点
        //return L;
        return true;
    }
    //否则，两者的入环节点不同，需分情况讨论，看是独立环，还是公共环。
    else
    {
        //定义指针p指向链表1入环节点的下一个节点
        ListNode *p = (*entry1).next;
        //遍历环内节点
        while(p!=entry1)
        {
            //如果在环内找到了链表2的入环节点
            if(p == entry2)
                return true;
            p = (*p).next;
        }
        //否则就是独立环，不相交，返回false
        return false;
    }
}


//返回有环的链表入环节点
ListNode* chkLoop(ListNode* head)
{
    if(!head)
        return NULL;
    ListNode *slow = head,*fast = head;
    //快指针一次走两步，慢指针一次走一步，相遇时break
    while(((*fast).next&&(*(*fast).next).next))
    {
        slow = (*slow).next;
        fast = (*(*fast).next).next;
        if(slow == fast)
            break;
    }
    //如果fast走到结尾，说明之前没有因相遇而break，因此，无环
    if(!(*fast).next||!(*(*fast).next).next)
        return NULL;
    //第二次，slow从头出发，fast从原来的位置出发，再次相遇的位置，就是入环的位置，两者这次步长均为1
    slow = head;
    while(slow!=fast)
    {
        slow = (*slow).next;
        fast = (*fast).next;
```

```
        }
        //返回相遇时的指针
        return fast;
    }
};

int main()
{
    return 0;
}
```

F12，

问题：

给定两个单链表的头结点head1，和head2，如何判断两个链表是否相交？相交的话返回第一个相交节点，不相交的话返回空。

解题思路：

1，先判断两个链表是否有环，有的话，返回入环节点，

2，如果入环节点1个为空，1个不为空，则肯定不相交，

3，如果两者都无环，则直接套用无环是否相交的方法。

4，如果两者都有环，则套用有环是否相交的办法。

code：

```
/*
    问题:
        给定两个单链表的头结点head1，和head2，如何判断两个链表是否相交？相交的话返回第一个相交节点，不相交的话返回空。
    解题思路:
        1，先判断两个链表是否有环，有的话，返回入环节点，
        2，如果入环节点1个为空，1个不为空，则肯定不相交，
        3，如果两者都无环，则直接套用无环是否相交的方法。
        4，如果两者都有环，则套用有环是否相交的办法。

*/
#include<iostream>
#include<string>
#include<stack>
#include<vector>
using namespace std;
struct ListNode
{
    int val;
    struct ListNode *next;
    ListNode(int x):val(x),next(NULL){}
};

class CheckIntersectInRing
{
public:
    //判断两个单链表是否相交
    bool chkInter(ListNode* head1,ListNode* head2)
    {
        ListNode *entry1 = chkLoop(head1),*entry2 = chkLoop(head2);
        //都有环，调用有环是否相交的方法
        if(entry1!= NULL&&entry2!=NULL)
            return chkRingListInter(head1,head2);
        if(entry1 == NULL&&entry2 == NULL)
            return chkNoRingListInter(head1,head2);
        return false;
    }

    //无环单链表判相交
    bool chkNoRingListInter(ListNode* headA,ListNode* headB)
    {
        //如果链表A，B有任何一方为空
        if(!headA || !headB)
            return false;
        //链表A和B的长度计数
        int NumofA = 0,NumofB=0,count=0;
        ListNode *p = headA;
        while(p)
        {
            NumofA++;
            p = (*p).next;
        }
        p = headB;
        while(p)
        {
            NumofB++;
            p = (*p).next;
        }
```

```
    //长的链表L，短的链表S
    ListNode *L = NULL,*S = NULL;
    if(NumofA > NumofB)
        L = headA,S=headB;
    else
        L= headB,S= headA;
    //长短链表的长度差值
    count = (NumofA > NumofB)?(NumofA-NumofB):(NumofB-NumofA);
    //先让长链表走count长度
    while(count--)
        L = (*L).next;
    while(L&&S)
    {
        //如果两个链表某处节点值相等
        if(L == S)
            return true;
        L = (*L).next;
        S = (*S).next;
    }
    return false;
}

//有环单链表判相交
bool chkRingListInter(ListNode* head1,ListNode* head2)
{
    //获取链表1,2的入环节点
    ListNode *entry1 = chkLoop(head1),*entry2 = chkLoop(head2);
    //如果一个有环，一个无环，一定不相交
    if(entry1 == NULL || entry2 == NULL)
        return false;
    //如果入环节点是同一个节点，则肯定相交
    if(entry1 == entry2)
    {
        //获取第一个相交节点
        int Numof1 = 0,Numof2 = 0,count=0;
        //获取链表1到环之前的长度
        ListNode *p = head1;
        while(p!=entry1)
        {
            Numof1++;
            p = (*p).next;
        }
        //获取链表2到环之前的长度
        p = head2;
        while(p != entry1)
        {
            Numof2++;
            p = (*p).next;
        }

        //定义长链表，短链表
        ListNode *L = NULL,*S = NULL;
        if(Numof1>Numof2)
            L = head1,S = head2;
        else
            L = head2,S = head1;
        //获取长短链表的长度差值
        count = (Numof1>Numof2)?(Numof1-Numof2):(Numof2-Numof1);
        //先让长链表走一段
        while(count--)
            L = (*L).next;
        while(L!=entry1&&S!=entry2)
        {
            //相交
            if(L == S)
                break;
            L = (*L).next;
            S = (*S).next;
        }
        //返回相交节点
        //return L;
        return true;
    }
    //否则，两者的入环节点不同，需分情况讨论，看是独立环，还是公共环。
    else
    {
        //定义指针p指向链表1入环节点的下一个节点
        ListNode *p = (*entry1).next;
        //遍历环内节点
        while(p!=entry1)
        {
            //如果在环内找到了链表2的入环节点
            if(p == entry2)
                return true;
            p = (*p).next;
        }
        //否则就是独立环，不相交，返回false
        return false;
```

```
        }
    }

    //返回有环但链表入环节点
    ListNode* chkLoop(ListNode* head)
    {
        if(!head)
            return NULL;
        ListNode *slow = head,*fast = head;
        //快指针一次走两步，慢指针一次走一步，相遇时break
        while(((*fast).next&&(*(*fast).next).next))
        {
            slow = (*slow).next;
            fast = (*(*fast).next).next;
            if(slow == fast)
                break;
        }
        //如果fast走到结尾，说明之前没有因相遇而break，因此，无环
        if(!(*fast).next||!(*(*fast).next).next)
            return NULL;
        //第二次，slow从头出发，fast从原来的位置出发，再次相遇的位置，就是入环的位置，两者这次步长均为1
        slow = head;
        while(slow!=fast)
        {
            slow = (*slow).next;
            fast = (*fast).next;
        }
        //返回相遇时的指针
        return fast;
    }

};


int main()
{
    return 0;
}
```

## 6，二分搜索（11/17）

### A，二分搜索常见的应用场景

#### 1，在有序序列中查找一个数，m，看存不存在

做法：先判断序列中间的数是不是m，若是，直接返回即可。若小于m，则在右侧的序列中重复同样的步骤，否则，在左侧的序列中重复同样的步骤。如果到最后序列间隔到0都还没找到m，则返回false。时间复杂度为O（logN）

### B,面试中，二分搜索常见的考察点

1，对边界条件的考察和代码实现能力。比如终止循环的条件等。防止存在范围永远不缩小的情况。

2，常见变化：

给定条件不同：无重复或有重复的数组

判断条件不同：大于x还是小于

返回内容不同：是返回一个等于x的元素，或者所有

3，在有序循环数组（数组前面的任意一部分可以拿到后边来的数组）中做二分搜索

4，对半的时候，常用的方法是(left+right）/2,但这种写法，在数组长度过大的时候可能溢出，所以，更安全的写法是：left+（right-left）/2.

### C1,

#### 问题：

给定一个无序数组，已知任意相邻的两个元素，值都不重复，请返回任意一个局部最小的位置，

所谓的局部最小位置是指，如果arr[0]<arr[1],那么位置0就是一个局部最下的位置。如果位置i既不是最左也不是最右位置，那么只要满足位置i处的内容同时小于它左右两侧的值，那么i也是个局部最小的位置。

#### 解题思路：

这里解题的思路仍可以用二分搜索，其时间复杂度为O（logN），这里序列虽然不是有序的，但它相邻元素间有不重复的性质。

1，如果arr为空，或者长度为0，直接返回-1即可。这种情况下，局部最小元素是不存在的。

2，如果arr长度为1，那么直接返回位置0，因为0是局部最小位置。

3，如果arr大于1，那么先比较位置0和位置1，如果0较小，直接返回即可。同样的，如果位置n-1,小于n-2，也可以直接返回。

4，如果两头的都不是局部最小位置，则有arr0 》 arr1，arrn-1》arrn-2，所以从最左边的位置往右看，趋势是往下的，从最后边的位置往左看，趋势也是向下的。此时我们可以看下中间点mid的情况，如果mid即小于它左边的数，又小于它右边的数，直接返回即可

5，否则，如果mid的值比其右边的值小，比左边的值大，那么从右往左看mid处，它是向下的。那么在mid位置的左边，必然存在局部最小值，然后对左部分继续进行同样的二分搜索。同理。如果mid的值比左边的小，比右边的大，则从左往后看，它是向下的，那么在mid的右边，必然存在局部最小

6，如果mid处的值即大于左边位置，又大于右边位置，那么证明其左右两侧都有局部最小。任选一边即可。

7，这道题是为了说明，二分搜索不一定是要在有序的数组中进行，只要在进行搜索时，能淘汰一半，保留一半即可。

#### code：

```
/*

    问题：
```

```
        给定一个无序数组，已知任意相邻的两个元素，值都不重复，请返回任意一个局部最小的位置，
        所谓的局部最小位置是指，如果arr[0]<arr[1],那么位置0就是一个局部最下的位置。如果位置i既不是最左也不是最右位置，那么只要满足位置i处的内容同时小于它左右两侧的值，那么i也是个局部最小的位置。
    解题思路：
        这里解题的思路仍可以用二分搜索，其时间复杂度为O（logN），这里序列虽然不是有序的，但它相邻元素间有不重复的性质。
        1，如果arr为空，或者长度为0，直接返回-1即可。这种情况下，局部最小元素是不存在的。
        2，如果arr长度为1，那么直接返回位置0，因为0是局部最小位置。
        3，如果arr大于1，那么先比较位置0和位置1，如果0较小，直接返回即可。同样的，如果位置n-1,小于n-2，也可以直接返回。
        4，如果两头的都不是局部最小位置，则有arr0 》 arr1，arrn-1》arrn-2，所以从最左边的位置往右看，趋势是往下的，从最后边的位置往左看，趋势也是向下的。此时我们可以看下中间点mid的情况，如果mid即小于它左边的数，又小于它右边的数，直接返回即可
        5，否则，如果mid的值比其右边的值小，比左边的值大，那么从右往左看mid处，它是向下的。那么在mid位置的左边，必然存在局部最小值，然后对左部分继续进行同样的二分搜索。同理。如果mid的值比左边的小，比右边的大，则从左往后看，它是向下的，那么在mid的右边，必然存在局部最小
        6，如果mid处的值即大于左边位置，又大于右边位置，那么证明其左右两侧都有局部最小。任选一边即可。
7，这道题是为了说明，二分搜索不一定是要在有序的数组中进行，只要在进行搜索时，能淘汰一半，保留一半即可。
*/
#include<iostream>
#include<vector>
using namespace std;
class LocalMin
{
public:
    //返回任意一个局部最小值出现的位置
    int getLocalMin(vector<int> arr)
    {
        //如果向量为空，没有局部最小位置
        if(arr.empty())
            return -1;
        //如果有一个元素，或者0处元素小于1处的
        if(arr.size() == 1 || arr[0] < arr[1])
            return 0;
        //判断最后一个位置的元素是否小于倒数第二个，若是，返回倒数第一个位置
        if(arr[arr.size()-1] < arr[arr.size()-2])
            return arr.size()-1;
        //否则，定义二分搜索的起始位置：low：1，high：size-2，和mid
        int low = 1,high = arr.size()-2,mid;
        //终止条件
        while(low <= high)
        {
            mid = low +(high-low)/2;
            //如果mid处的值比mid-1处大，则根据下降趋势，low到mid-1处一定有局部最小
            if(arr[mid]>arr[mid-1])
                high = mid-1;
            //如果mid处的值比mid+1处大，根据趋势，mid+1到high之间一定有局部最小
            else if(arr[mid] > arr[mid+1])
                low = mid+1;
            //否则，即比左边小又比右边小，是局部最小位置
            else
                return mid;
        }
        return -1;
    }
};

int main()
{
    int a[30]={10,5,10,5,0,1,2,4,7,3,2,9,5,4,6,5,10,6,7,10,9,4,3,7,2,9,5,4,6,10};
    vector<int> arr(a,a+30);
    LocalMin L;
    cout<<L.getLocalMin(arr)<<endl;
    return 0;
}
```

## C2，

### 问题：

给定一个有序数组arr，再给定一个整数Num，请在arr中找到num这个数出现的最左边位置

### 解题思路：

假设数组为1 2 3 3 3 3 4 4 4 4 4 4 4 4 4，num为3

1，在进行二分搜索之前，先假设一个全局变量res，用来标识最后一个找到3的位置。初始时res=-1，如果在遍历的过程中，一直没找到，则最后返回-1即可、

2，首先找到中间的数，4，然后比较中间的数和num的大小，发现num是在数组的左边的，

3，再找到左边的中间位置，3，第一次找到3，将res更新为当前的位置，

4，但我们要找的是3出现最左边的位置，所以，当第一次出现3之后，对其左边的部分，继续二分，找到中间位置2

5，此时2左边的内容一定是小于3的，因此，处理右半部分，2 3，继续进行二分搜索，找到另一个3，位置更新为2，

6，此时，区间只剩一个数了，这是最后一次更新res

### code：

```
/*
        问题：
```

```
        给定一个有序数组arr，再给定一个整数Num，请在arr中找到num这个数出现的最左边位置
    解题思路：
        假设数组为1 2 3 3 3 3 4 4 4 4 4 4 4 4 4，num为3
        1，在进行二分搜索之前，先假设一个全局变量res，用来标识最后一个找到3的位置。初始时res=-1，如果在遍历的过程中，一直没找到，则最后返回-1即可、
        2，首先找到中间的数，4，然后比较中间的数和num的大小，发现num是在数组的左边的，
        3，再找到左边的中间位置，3，第一次找到3，将res更新为当前的位置，
        4，但我们要找的是3出现最左边的位置，所以，当第一次出现3之后，对其左边的部分，继续二分，找到中间位置2
        5，此时2左边的内容一定是小于3的，因此，处理右半部分，2 3，继续进行二分搜索，找到另一个3，位置更新为2，
        6，此时，区间只剩一个数了，这是最后一次更新res

*/
#include<iostream>
#include<vector>
using namespace std;
class LeftMostPosition
{
public:
    //传入向量，向量长度和要找的元素
    int findLeftPos(vector<int> arr, int n, int num)
    {
        if(!n)
            return -1;
        int low = 0,high = n-1,mid;
        int res = -1;
        while(low<=high)
        {
            mid = low+(high - low)/2;
            //数组是有序的，如果mid处的值小于待查找元素，则元素在右半区间
            if(arr[mid]<num)
                low = mid +1;
            //若大于，则在左边
            else if(arr[mid]>num)
                high = mid-1;
            //如果恰好等，那么因为要找的是最左位置，再去左边看看有没有更左的
            else
            {
                res = mid;
                high = mid-1;
            }
        }
        return res;
    }
};

int main()
{
    int a[5] = {1,2,3,3,4};
    vector<int> arr(a,a+5);
    LeftMostPosition L;
    cout<<L.findLeftPos(arr,5,3)<<endl;
    return 0;
}
```

C3，

问题：

给定一个有序循环数组arr，返回arr中的最小值。有序循环数组是指，有序数组左边任意长度的部分放到右边去，右边的部分拿到左边来。比如１２３３４，是有序循环数组。４１２３３也是。

解题思路：

元素位置从0-N-1，L初始为0，R初始为N-1，

1，如果L处的值小于R处的值，说明从L到R这一段是有序的，直接返回L即可，即是最小值

2，如果L处的值大于等于R处的值，说明这一段里面是有循环过的。此时，拿mid与L比较，如果L处的值》M处的值，那么最小值肯定在L，M这一段区间上。如果arr M > arr L,那么最小值只会出现在m，R这一段区间上。则在右半部分继续二分查找

3，有一种特殊情况，L，M，R处的值相等，这种情况下，只能用遍历的方法找到数组的最小值。

code：

```
/*
    问题：
        给定一个有序循环数组arr，返回arr中的最小值。有序循环数组是指，有序数组左边任意长度的部分放到右边去，右边的部分拿到左边来。比如 1 2 3 3 4，是有序循环数组。4 1 2 3 3也是。
    解题思路：
        元素位置从0-N-1，L初始为0，R初始为N-1，
        1，如果L处的值小于R处的值，说明从L到R这一段是有序的，直接返回L即可，即是最小值
        2，如果L处的值大于等于R处的值，说明这一段里面是有循环过的。此时，拿mid与L比较，如果L处的值》M处的值，那么最小值肯定在L，M这一段区间上。如果arr M > arr L,那么最小值只会出现在m，R这一段区间上。则在右半部分继续二分查找
        3，有一种特殊情况，L，M，R处的值相等，这种情况下，只能用遍历的方法找到数组的最小值。

*/
#include<iostream>
#include<vector>
#include<stdlib.h>
```

```
using namespace std;
class FindMinInOrderLoopArray
{
public:
    //传入向量和向量长度，返回最小值
    int getMin(vector<int> arr,int n)
    {
        //向量为空，没有最小值位置
        if(arr.empty())
            exit(-1);
        //如果长度为1，或者是没有循环过得有序数组，直接返回位置0处的元素
        if(n == 1 || arr[0] < arr[n-1])
            return arr[0];
        int low = 0,high = n-1,mid;
        while(low< high)
        {
            mid = low + (high - low)/2;
            //一定是循环过得，如果low处的值大于mid处的值，那么最小值在low，mid这一段内
            if(arr[low]>arr[mid])
                high = mid;
            //如果mid处的值大于high处的值，那么最小值一定出现在m，high,这段区间上
            else if(arr[mid]>arr[high])
                low = mid+1;
            //low = mid = high
            else
                break;
        }
        //最后区间low，high重合的情况，最小值在low或high处。
        if(low == high)
            return arr[low];
        //处理low == mid == high这种特殊情况
        int min = arr[low];
        while(low <= high)
        {
            if(arr[low]<min)
                min = arr[low];
            low++;
        }
        return min;


    }
};

int main()
{
    int a[3] = {2,1,2};
    vector<int> arr(a,a+3);
    FindMinInOrderLoopArray F;
    cout<<F.getMin(arr,3)<<endl;
    return 0;
}
```

## C4，

### 问题：

给定一个有序数组arr，其中不含重复元素，请找到满足arr[i] ==i条件的最左的位置，如果所有位置上的都不满足，返回-1.

### 解题思路：

1，首先，生成全局变量res，记录最后一次发生上述条件的情况。

2，如果arr[0]>N-1,那么后面的所有值都是大于N-1的，不可能出现条件那种情况，返回-1

3，如果arr[n-1]<0,同样返回-1

4，否则，判断M处的值是否大于M，如果大于M，那么M到N范围内的数都不可能满足条件，只能考虑0-M，反之，如果是小于M，则只能考虑M-N，如果arrm = m。首先记录下此处位置，然后由于记录的是最左的位置，则继续往左二分搜索

### code：

```
/*
        问题：
            给定一个有序数组arr，其中不含重复元素，请找到满足arr[i] ==i条件的最左的位置，如果所有位置上的都不满足，返回-1.
        解题思路：
            1，首先，生成全局变量res，记录最后一次发生上述条件的情况。
            2，如果arr[0]>N-1,那么后面的所有值都是大于N-1的，不可能出现条件那种情况，返回-1
            3，如果arr[n-1]<0,同样返回-1
            4，否则，判断M处的值是否大于M，如果大于M，那么M到N范围内的数都不可能满足条件，只能考虑0-M，反之，如果是小于M，则只能考虑M-N，如果arrm = m。首先记录下此处位置，然后由于记录的是最左的位置，则继续往左二分搜索

*/
#include<iostream>
#include<vector>
using namespace std;
class FindLeftPos
```

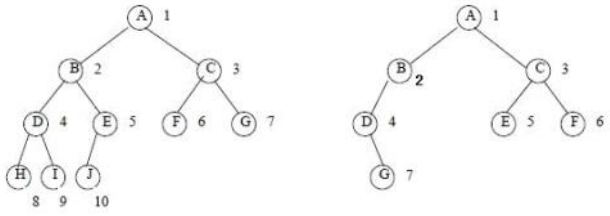
```
{
public:
    //传入向量arr和其长度，返回满足条件的最左位置
    int findPos(vector<int> arr,int n)
    {
        //数组为空，和后边两种情况，都是不可能满足条件的存在
        if(arr.empty()|| arr[0]> n-1||arr[n-1]<0)
            return -1;
        int res = -1;
        int low =0,high = n-1,mid;
        while(low<=high)
        {
            mid = low + (high - low)/2;
            //如果mid处的元素值小于mid,那么mid左侧的不可能满足条件，因此，low = mid+1
            if(arr[mid]<mid)
                low = mid+1;
            //如果mid处的元素大于mid。则其右侧的不可能，故high = mid-1
            else if(arr[mid]>mid)
                high = mid -1;
            //否则的话，mid处就满足条件，再往左，看是否有更左的满足条件的位置
            else
            {
                res = mid;
                high = mid-1;
            }
        }
        return res;
    }
};

int main()
{
    int a[5] = {-1,0,2,3};
    vector<int> arr(a,a+4);
    FindLeftPos F;
    cout<<F.findPos(arr,4)<<endl;
    return 0;
}
```

## C5，

### 问题：

给定一颗完全二叉树的头结点head，返回这颗树的节点数，如果完全二叉树的节点数为N，请事先时间复杂度低于O（N）的解法




(a) 一棵完全二叉树　　(b) 一棵非完全二叉树

图 6.3　完全二叉树和非完全二叉树示意图

### 解题思路：

最优解的思路来自二分搜索。完全二叉树，添加节点是从最后一层节点开始，从左往后添加，这一层满了，才开始下一层。删除过程也是同样的逆过程。

1，先找到二叉树最左的节点，（用来统计二叉树的高度）。其最左的节点，一定是在最后一层上的，

2，我们先找到二叉树的跟节点的右子树的最左节点，如果这个的最左节点能到最后一层，那么说明这颗二叉树根节点的左子树一定是棵满二叉树。可以根据满二叉树的节点公式，计算出左子树的节点个数，再加上根的，剩下的就是右子树的节点个数了。我们可以用递归的方式，在右子树上进行同样的过程。

3，但如果右子树没能到最后一层，说明右子树同样也是一颗满二叉树，用满二叉树的性质即可计算右子树的节点。左边的递归即可。

最优解的时间复杂度为O（logH^2),H为树高。

### code：

```
/*
        问题：
            给定一颗完全二叉树的头结点head，返回这颗树的节点数，如果完全二叉树的节点数为N，请事先时间复杂度低于O（N）的解法


        解题思路：
            最优解的思路来自二分搜索。完全二叉树，添加节点是从最后一层节点开始，从左往后添加，这一层满了，才开始下一层。删除过程也是同样的逆过程。
            1，先找到二叉树最左的节点，（用来统计二叉树的高度）。其最左的节点，一定是在最后一层上的，
            2，我们先找到二叉树的跟节点的右子树的最左节点，如果这个的最左节点能到最后一层，那么说明这颗二叉树根节点的左子树一定是棵满二叉树。可以根据满二叉树的节点公式，计算出左子树的节点个数，再加上根的，剩下的就是右子树的节点个数了。我们可以用递归的方式，在右子树上进行同样的过程。
            3，但如果右子树没能到最后一层，说明右子树同样也是一颗满二叉树，用满二叉树的性质即可计算右子树的节点。左边的递归即可。
            最优解的时间复杂度为O（logH^2),H为树高。
```

```
*/
#include<iostream>
#include<string>
#include<cmath>
using namespace std;
struct TreeNode
{
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x):val(x),left(NULL),right(NULL){}
};
class CountCompleteTree
{
public:
    //传入二叉树的根节点，返回树的节点数
    int count(TreeNode* root)
    {
        if(!root)
            return 0;
        //计算左子树高度
        int lenOfLeft = DepthOfTree((*root).left);
        //计算右子树高度
        int lenOfRight = DepthOfTree((*root).right);
        //如果左右子树等高，那么左子树一定是个满二叉树，套用节点公式2^树高-1，对于右子树则递归求解
        if(lenOfLeft == lenOfRight)
            //这里的-1刚好和根节点抵消
            return pow(2.0,lenOfLeft)+count((*root).right);
        //如果左右两边不等高，根据完全二叉树的性质，右边肯定满二叉树
        else
            return pow(2.0,lenOfRight)+count((*root).left);
    }

    //返回传入树的高度
    int DepthOfTree(TreeNode *root)
    {
        if(!root)
            return 0;
        TreeNode *p = root;
        //树高
        int len = 0;
        while(p)
        {
            len++;
            //一直沿着左孩子走
            p = (*p).left;
        }
        return len;
    }
};
int main()
{
    return 0;
}
```

C6，

问题：

如何更快的求一个整数k的N次方，如果两个整数相乘并得到结果的时间复杂度为O(1),得到整数k的N次方的过程请事先时间复杂度为O（logN）的方法。

解题思路：

1，普通的做法将k相乘N次，这种方法的时间复杂度是O（n）。

2，最优解，假设做10^75,

3，首先，我们将75拆解为二进制，1001011，= 10^64*10^8*10^2*10^1

4,我们先求10^1,然后根据10^1求10^2,再求得10^4,依次这样下去，得到10^64.

5,然后累乘该乘的值。

code：

```
/*
    问题：
        如何更快的求一个整数k的N次方，如果两个整数相乘并得到结果的时间复杂度为O(1),得到整数k的N次方的过程请事先时间复杂度为O（logN）的方法。
    解题思路：
        1，普通的做法将k相乘N次，这种方法的时间复杂度是O（n）。
        2，最优解，假设做10^75,
        3，首先，我们将75拆解为二进制，1001011，= 10^64*10^8*10^2*10^1
        4,我们先求10^1,然后根据10^1求10^2,再求得10^4,依次这样下去，得到10^64.
        5,然后累乘该乘的值。
```

```
*/
#include<iostream>
#include<cmath>
#include<bitset>
#include<string>
#include<vector>
using namespace std;
class QuickNPower
{
public:
    //计算k的N次方
    int getPower(int k, int N)
    {
        if(k == 0)
            return 0;
        if(N == 0)
            return 1;
        //防止溢出
        if(k>1000000007)
            k = k%1000000007;
        //存放k的2次放，4次放，8次方。。
        vector<long> arr;
        //存放N转换为二进制的形式
        vector<int> bit;
        //m为N次放，temp为k
        long long m = N,temp = k,res;
        while(m)
        {
            //在arr中依次放入，k，k^2,(k^2)^2,....，m可以转换为多少位二进制，这个k就要^2多少次
            arr.push_back(temp);
            //temp变为k^2
            temp*=temp;
            //防止溢出
            if(temp>1000000007)
                temp = temp%1000000007;
            //将m转为2进制，存入bit
            if(m%2)
                bit.push_back(1);
            else
                bit.push_back(0);
            m = m/2;
        }

        //根据bit里是0是1，判断要不要累乘对应位数的k次方
        res = 1;
        for(int i = 0;i<bit.size();i++)
        {
            //cout<< bit[i] << " "<< arr[i] << endl;
            if(bit[i])
            {
                res *= arr[i];
                //cout << res << endl;
                if(res > 1000000007)
                    res = res%1000000007;
            }
        }
        return res%1000000007;
    }
};
int main()
{
    QuickNPower Q;
    cout<<Q.getPower(2,14876069)<<endl;
    return 0;
}
```

## 7，二叉树

## 8，位运算

## 9，排列组合

## 10，概率

## 11，大数据

## 12，动态规划