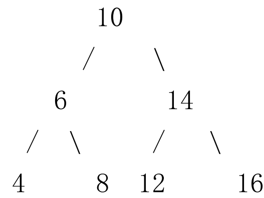
1. 把二元查找树转换成排序的双向链表

题目：输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。要求不能创建任何新的结点，只调整指针的指向。

比如将二元查找树



转换成双向链表

4=6=8=10=12=14=16。

分析：本题是微软的面试题。很多与树相关的题目都是用递归的思路来解决，本题也不例外。二元查找树 BST 的一个重要特性便是如果按照中序方式对其进行遍历，结果是一个有序的序列。

所以我们可以中序遍历整棵树。按照这个方式遍历树，比较小的结点总是最先被访问。如果我们每访问一个结点，假设之前访问过的结点已经调整成一个排序好的双向链表，我们再调整当前结点的指针，将其链接到链表的末尾。当所有结点都访问过之后，整棵树也就转换成一个排序双向链表了。

源码：

```
#include <iostream>
using namespace std;

typedef struct node
{  int data;
   struct node *left;
   struct node *right;
}BSTNode;

int InsertBST(BSTNode *&bt, int k)
{  //作为叶子结点插入 BST 树
   if(bt==NULL)
   {  bt=(BSTNode*)malloc(sizeof(BSTNode));
      bt->data=k;
      bt->left=bt->right=NULL;
      return 1;
   }
   if(bt->data==k)
     return 0;
   if(bt->data>k)
     return InsertBST(bt->left, k);
   if(bt->data<k)
     return InsertBST(bt->right, k);
}
```

```
void CreateBST(BSTNode *&bt, int r[], int n)
{  int i;
   for(i=0; i<n; i++)
   {  InsertBST(bt, r[i]);
   }
}

void ConvertToDLinkList(BSTNode *bt, BSTNode *&visited)
{  BSTNode *cur=bt;
   if(cur!=NULL)
   {  ConvertToDLinkList(cur->left, visited);
      cur->left=visited;
      if(visited!=NULL)
        visited->right=cur;
      visited=cur;
      ConvertToDLinkList(cur->right, visited);
   }
}

void DispDLinkList(BSTNode *bt)
{  while(bt!=NULL)
   {  printf("%d ", bt->data);
      bt=bt->right;
   }
}

int main()
{  int r[5]={3,88,9,42,16};
   BSTNode *bt=NULL;
   CreateBST(bt, r, 5);
   BSTNode *h=NULL;
   ConvertToDLinkList(bt, h);

   //查找双链表的开始结点
   while(h->left!=NULL)
     h=h->left;

   //逐个打印显示结点值
   DispDLinkList(h);
   system("pause");
   return 0;
}
```

2. 设计包含 min 函数的栈

题目：定义栈的数据结构，要求添加一个 min 函数，能够得到栈的最小元素。要求函数 min、push 以及 pop 的时间复杂度都是 O(1)。

分析：这是去年 google 的一道面试题。我们需要一个辅助栈。每次 push 一个新元素的时候，同时将最小元素（或最小元素的位置。考虑到栈元素的类型可能是复杂的数据结构，用最小元素的位置将能减少空间消耗）push 到辅助栈中；而每次 pop 一个元素出栈的时候，同时 pop 辅助栈。

源码：

```
#include <iostream>
using namespace std;

#define MaxSize 100

typedef struct
{   int data[MaxSize];
    int top;
}SqStack;

/**
 * s 作为主要存放数据的栈，t 作为辅助栈，存放当前栈中最小元素的下标
 */
void InitStack(SqStack *&s, SqStack *&t)
{   s=(SqStack*)malloc(sizeof(SqStack));
    t=(SqStack*)malloc(sizeof(SqStack));
    s->top=t->top=-1;
}

int Push(SqStack *s, SqStack *t, int data)
{   if(s->top==MaxSize-1)
      return 0;
    s->top++;
    s->data[s->top]=data;
    if(t->top==-1)
    {   t->top++;
        t->data[t->top]=0;
    }
    else
    {   t->top++;
        if(data<s->data[t->data[t->top-1]])
            t->data[t->top]=s->top;
        else
            t->data[t->top]=t->data[t->top-1];
    }
    return 1;
```

```
}

int Pop(SqStack *s, SqStack *t, int &data)
{  if(s->top==-1)
     return 0;
   data=s->data[s->top];
   s->top--;
   t->top--;
   return 1;
}

int min(SqStack *s, SqStack *t)
{   assert(s->top!=-1&&t->top!=-1);
    return s->data[t->data[t->top]];
}

int main()
{  SqStack *s, *t;
   InitStack(s,t);
   Push(s,t,12);
   Push(s,t,4);
   Push(s,t,-5);
   int m;
   printf("%d\n", min(s,t));
   Pop(s,t,m);
   printf("%d\n", min(s,t));
   Pop(s,t,m);
   printf("%d\n", min(s,t));
   system("pause");
   return 0;
}
```

3. 求子数组的最大和

题目：输入一个整形数组，数组里有正数也有负数。数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。求所有子数组的和的最大值。要求时间复杂度为O(n)。

例如输入的数组为1, -2, 3, 10, -4, 7, 2, -5，和最大的子数组为3, 10, -4, 7, 2，因此输出为该子数组的和18。

分析：本题最初为2005年浙江大学计算机系的考研题的最后一道程序设计题，在2006年里包括 google 在内的很多知名公司都把本题当作面试题。由于本题在网络中广为流传，本题也顺利成为2006年程序员面试题中经典中的经典。

如果不考虑时间复杂度，我们可以枚举出所有子数组并求出他们的和。不过非常遗憾的是，由于长度为 n 的数组有 $O(n^2)$个子数组；而且求一个长度为 n 的数组的和的时间复杂度为 O(n)。因此这种思路的时间是 $O(n^3)$。因为要求是 O(N)的复杂度，因此需采用的 DP 的思

想，记录下当前元素之和（为其最优状态，既最大），将其与目前所得的最大和比较，若大于则更新，否则继续。状态的累加遵循这个过程：如果当前和小于0，则放弃该状态，将其归零。

很容易理解，当我们加上一个正数时，和会增加；当我们加上一个负数时，和会减少。如果当前得到的和是个负数，那么这个和在接下来的累加中应该抛弃并重新清零，不然的话这个负数将会减少接下来的和。基于这样的思路，我们可以写出如下代码。

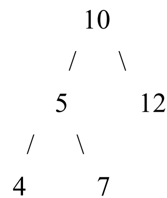
源码：

```
#include <iostream>
using namespace std;

#define MaxSize 100

int FindGreatestSubSum(int r[],int n,int &start,int &end)
{   int maxSum,curSum;
    maxSum=curSum=0;
    int curStart=0, curEnd=0;
    for (int i=0;i<n;i++)
    {
        curSum+=r[i];
        curEnd=i;
        if(curSum<0)
        {  curSum=0;
           curStart=i+1;
        }
        if(curSum>maxSum)
        {  maxSum=curSum;
           start=curStart;
           end=curEnd;
        }



    }
    if(maxSum==0){      //若是数组中的元素均为负数，则输出里面的最大元素
        maxSum=r[0];    //当然这步也可以写到上面一个循环里
        int k;
        for(int i=1;i<n;i++){
          if(maxSum<r[i]) {maxSum=r[i]; k=i;}
        }
        start=end=k;
    }
    return maxSum;
}
int main()
{  int a[8]={-51, -2, -3, 10, -4, -7, -2, -5};
```

```
    int start=0,end=0;
    cout<<FindGreatestSubSum(a,8,start,end)<<endl;
    cout<<start<<" "<<end;
    system("pause");
    return 0;
}
```

4. 在二元树中找出和为某一值的所有路径

题目：输入一个整数和一棵二元树。从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。打印出和与输入整数相等的所有路径。例如输入整数22和如下二元树



则打印出两条路径：10, 12和10, 5, 7。

二元树结点的数据结构定义为：

```
struct BinaryTreeNode          // a node in the binary tree
{
    int              m_nValue;   // value of node
    BinaryTreeNode *m_pLeft;   // left child of node
    BinaryTreeNode *m_pRight;  // right child of node
};
```

分析：这是百度的一道笔试题，考查对树这种基本数据结构以及递归函数的理解。当访问到某一结点时，把该结点添加到路径上，并累加当前结点的值。如果当前结点为叶结点并且当前路径的和刚好等于输入的整数，则当前的路径符合要求，我们把它打印出来。如果当前结点不是叶结点，则继续访问它的子结点。当前结点访问结束后，递归函数将自动回到父结点。我们不难看出保存路径的数据结构实际上是一个栈结构，因为路径要与递归调用状态一致，而递归调用本质就是一个压栈和出栈的过程。

源码：

```
#include <iostream>
using namespace std;

#define MaxSize 100

typedef struct node
{   int data;
    struct node *left, *right;
}BSTNode;

int InsertBST(BSTNode *&t, int d)
{   if(t==NULL)
```

```
    {   t=(BSTNode*)malloc(sizeof(BSTNode));
        t->data=d;
        t->left=t->right=NULL;
        return 1;
    }
    if(t->data==d)
      return 0;
    if(t->data>d)
      return InsertBST(t->left, d);
    if(t->data<d)
      return InsertBST(t->right, d);
}

void CreateBST(BSTNode *&t, int r[], int n)
{   int i;
    for(i=0; i<n; i++)
      InsertBST(t, r[i]);
}

void AllPath(BSTNode *b,int path[],int curp, int expSum)
{   int i,tmp=0;
    if (b!=NULL)
    { if(b->left==NULL&&b->right==NULL)
      { for(i=0; i<curp; i++)
            tmp+=path[i];
        if(tmp+b->data==expSum)
        {   printf("%d ",b->data);  //主要是最后这个点没压栈，故单独打印
            for(i=curp-1;i>=0;i--)
                printf("%d ",path[i]);
            printf("\n");
        }
      }
      else
      {   path[curp]=b->data;
          curp++;
          AllPath(b->left,path,curp, expSum);
          AllPath(b->right,path,curp, expSum);
          curp--;
      }
    }
}

int main()
{   int r[7]={6,3,4,7,1,9,10};
```

```
    BSTNode *t=NULL;
    CreateBST(t,r,7);
    int path[MaxSize];
    AllPath(t,path,0, 13);
    system("pause");
    return 0;
}
```

5.查找最小的 k 个元素

题目：输入 n 个整数，输出其中最小的 k 个。例如输入1，2，3，4，5，6，7和8这8个数字，则最小的4个数字为1，2，3和4。

分析：这道题最简单的思路莫过于把输入的 n 个整数排序，这样排在最前面的 k 个数就是最小的 k 个数。只是这种思路的时间复杂度为 O(n*log*n)。我们试着寻找更快的解决思路。

我们可以先创建一个大小为 k 的数据容器来存储最小的 k 个数字。接下来我们每次从输入的 n 个整数中读入一个数。如果容器中已有的数字少于 k 个，则直接把这次读入的整数放入容器之中；如果容器中已有 k 个数字了，也就是容器已满，此时我们不能再插入新的数字而只能替换已有的数字。我们找出这已有的 k 个数中最大值，然后拿这次待插入的整数和这个最大值进行比较。如果待插入的值比当前已有的最大值小，则用这个数替换替换当前已有的最大值；如果带插入的值比当前已有的最大值还要大，那么这个数不可能是最小的 k 个整数之一，因为我们容器内已经有 k 个数字比它小了，于是我们可以抛弃这个整数。

因此当容器满了之后，我们要做三件事情：一是在 k 个整数中找到最大数，二是有可能在这个容器中删除最大数，三是可能要插入一个新的数字，并保证 k 个整数依然是排序的。如果我们用一个二叉树来实现这个数据容器，那么我们能在 O(*log*k)时间内实现这三步操作。因此对于 n 个输入数字而言，总的时间效率就是 O(n*log*k)。

我们可以选择用不同的二叉树来实现这个数据容器。由于我们每次都需要找到 k 个整数中的最大数字，我们很容易想到用最大堆。在最大堆中，根结点的值总是大于它的子树中任意结点的值。于是我们每次可以在 O(1)得到已有的 k 个数字中的最大值，但需要 O(*log*k)时间完成删除以及插入操作。

源码：

```
#include <iostream>
using namespace std;

#define MaxSize 100

#define K 3

int heap[K];

void HeapAdjust(int r[], int low, int high)
{   int i=low, j=2*i+1, tmp;
    tmp=r[i];
    while(j<=high)
    {   if(j<high&&r[j]<r[j+1])
```

```
          j++;
        if(tmp<r[j])
        {  r[i]=r[j];
           i=j;
           j=2*i+1;
        }
        else
          break;
    }
    r[i]=tmp;
}

void CreateHeap(int r[], int n)
{  int i;
   for(i=n/2-1; i>=0; i--)
   {  HeapAdjust(r,i,n-1);
   }
}

void GetTopK(int r[], int n)
{  int i;
   for(i=0; i<K; i++)
     heap[i]=r[i];
   CreateHeap(heap,K);
   for(i=K; i<n; i++)
   {  if(heap[0]>r[i])
      {  heap[0]=r[i];
         HeapAdjust(heap, 0, K-1);
      }
   }
}

int main()
{  int r[12]={3,7,-24,60,17,0,36,788,224,40,27,99};
   GetTopK(r,12);

   int i;
   for(i=0; i<K; i++)
     printf("%d ", heap[i]);
   system("pause");
   return 0;
}
```

6. 判断整数序列是不是二元查找树的后序遍历结果

题目：输入一个整数数组，判断该数组是不是某二元查找树的后序遍历的结果。如果是返回 true，否则返回 false。

例如输入5、7、6、9、11、10、8，由于这一整数序列是如下树的后序遍历结果：

```
        8
      / \
     6     10
    / \      / \
  5   7     9    11
```

因此返回 true。

如果输入7、4、6、5，没有哪棵树的后序遍历的结果是这个序列，因此返回 false。

分析：这是一道 trilogy 的笔试题，主要考查对二元查找树的理解。

在后续遍历得到的序列中，最后一个元素为树的根结点。从头开始扫描这个序 列，比根结点小的元素都应该位于序列的左半部分；从第一个大于跟结点开始到跟结点前面的一个元素为止，所有元素都应该大于跟结点，因为这部分元素对应的是 树的右子树。根据这样的划分，把序列划分为左右两部分，我们递归地确认序列的左、右两部分是不是都是二元查找树。

源码：

```
#include <iostream>
using namespace std;

int VerifyBST(int r[], int n)
{    int i,j,root,left,right;
     if(n<=0)
        return 0;
     root=r[n-1];
     left=right=1;
     for(i=0; i<n-1; i++)
        if(r[i]>root)
           break;
     for(j=i; j<n-1; j++)
        if(r[j]<root)
           return 0;
     if(i>0)
        left=VerifyBST(r,i);
     if(j<n-1)
        right=VerifyBST(r+i,n-i-1);
     return (left&&right);
}


int main()
{   /*int arr1[] = {5,7,6,9,11,10,8};
    int arr2[] = {7,4,6,5};
```

```
    if(VerifyBST(arr1,7))
        printf("yes\n");
    else
        printf("no\n");*/

    system("pause");
    return 0;
}
```

7. 翻转句子中单词的顺序

题目：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。
例如输入“I am a student.”，则输出“student. a am I”。

分析：由于编写字符串相关代码能够反映程序员的编程能力和编程习惯，与字符串相关的问题一直是程序员笔试、面试题的热门题目。本题也曾多次受到包括微软在内的大量公司的青睐。

由于本题需要翻转句子，我们先颠倒句子中的所有字符。这时，不但翻转了句子中单词的顺序，而且单词内字符也被翻转了。我们再颠倒每个单词内的字符。由于单词内的字符被翻转两次，因此顺序仍然和输入时的顺序保持一致。

还是以上面的输入为例子。翻转“I am a student.”中所有字符得到“.tneduts a ma I”，再翻转每个单词中字符的顺序得到“students. a am I”，正是符合要求的输出。

源码：

```
#include <iostream>
using namespace std;

void reverse(char *begin, char *end)
{   char tmp;
    if(begin==NULL||end==NULL)
      return;
    while(begin<end)
    {   tmp=*begin;
        *begin=*end;
        *end=tmp;
        begin++; end--;
    }
}

//先逆转整个句子，然后从首字符开始扫描，
//每扫描到一个单词（遇到空白或结束字符），对这个单词进行逆转。
char *ReverseSentence(char *s)
{   if(s==NULL)
      return NULL;
    char *begin, *end;
```

```
    begin=end=s;
    while(*end!='\0')
      end++;
    end--;
    reverse(begin,end);
    while(*begin!='\0')
    {   end=begin;
        while(*end!=' '&&*end!='\0')
          end++;
        end--;
        reverse(begin,end);
        if(*(end+1)==' ')
          begin=end+2;
        else
          begin=end+1;
    }
    return s;
}

int main()
{   char a[100] = "I am a student.";
    ReverseSentence(a);
    cout<<a<<endl;
    system("pause");
    return 0;
}
```

8. 求 1+2+...+n

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

分析：这道题没有多少实际意义，因为在软件开发中不会有这么变态的限制。但这道题却能有效地考查发散思维能力，而发散思维能力能反映出对编程相关技术理解的深刻程度。

通常求1+2+…+n 除了用公式 n(n+1)/2之外，无外乎循环和递归两种思路。由于已经明确限制 for 和 while 的使用，循环已经不能再用了。同样，递归函数也需要用 if 语句或者条件判断语句来判断是继续递归下去还是终止递归，但现在题目已经不允许使用这两种语句了。

我们仍然围绕循环做文章。循环只是让相同的代码执行 n 遍而已，我们完全可以不用 for 和 while 达到这个效果。比如定义一个类，我们 new 一含有 n 个这种类型元素的数组，那么该类的构造函数将确定会被调用 n 次（运用静态变量）。

我们同样也可以围绕递归做文章。既然不能判断是不是应该终止递归，我们不妨定义两个函数。一个函数充当递归函数的角色，另一个函数处理终止递归的情况，我们需要做的就是在两个函数里二选一。从二选一我们很自然的想到布尔变量，比如 ture（1）的时候调用第一个函数，false（0）的时候调用第二个函数。那现在的问题是如和把数值变量 n 转换成布尔值。如果对 n 连续做两次反运算，即!!n，那么非零的 n 转换为 true，0转换为 false。

源码：

```
    #include <iostream>
using namespace std;

int (*ptr[2])(int n);  //函数指针数组

int fun(int n)
{  return ptr[!n](n);
}

int add(int n)
{  return (n+fun(n-1));
}

int end(int n)
{  return 0;
}

int func(int n)
{    int i=1;
     (n>1)&&(i=func(n-1)+n);
     return i;
}

int main()
{    int n, sum;
    printf("Input n: ");
    scanf("%d", &n);
    ptr[0] = &add;
    ptr[1] = &end;
   //sum = fun(n);
   sum=func(n);
    printf("sum: %d\n", sum);

    system("pause");
    return 0;
}
```

9. 查找链表中倒数第 K 个结点

题目：输入一个单向链表，输出该链表中倒数第 k 个结点。链表的倒数第 0 个结点为链表的尾指针。链表结点定义如下：

```
struct ListNode
{
```

```
    int      m_nKey;
    ListNode* m_pNext;
};
```

分析：为了得到倒数第 k 个结点，很自然的想法是先走到链表的尾端，再从尾端回溯 k 步。可是输入的是单向链表，只有从前往后的指针而没有从后往前的指针。因此我们需要打开我们的思路。

既然不能从尾结点开始遍历这个链表，我们还是把思路回到头结点上来。假设整个链表有 n 个结点，那么倒数第 k 个结点是从头结点开始的第 n-k-1个结点（从0开始计数）。如果我们能够得到链表中结点的个数 n，那我们只要从头结点开始往后走 n-k-1步就可以了。如何得到结点数 n？这个不难，只需要从头开始遍历链表，每经过一个结点，计数器加一就行了。这种思路的时间复杂度是 O(n)，但需要遍历链表两次。第一次得到链表中结点个数 n，第二次得到从头结点开始的第 n--k-1个结点即倒数第 k 个结点。

如果链表的结点数不多，这是一种很好的方法。但如果输入的链表的结点个数很多，有可能不能一次性把整个链表都从硬盘读入物理内存，那么遍历两遍意味着一个结点需要两次从硬盘读入到物理内存。我们知道把数据从硬盘读入到内存是非 常耗时间的操作。我们能不能把链表遍历的次数减少到1？如果可以，将能有效地提高代码执行的时间效率。

如果我们在遍历时维持两个指针，第一个指针从链表的头指针开始遍历，在第 k-1步之前，第二个指针保持不动；在第 k-1步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在 k-1，当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第 k 个结点。
这种思路只需要遍历链表一次。对于很长的链表，只需要把每个结点从硬盘导入到内存一次。因此这一方法的时间效率前面的方法要高。

讨论：这道题的代码有大量的指针操作。在软件开发中，错误的指针操作是大部分问题的根源。因此每个公司都希望程序员在操作指针时有良好的习惯，比如使用指针之前判断是不是空指针。这些都是编程的细节，但如果这些细节把握得不好，很有可能就会和心仪的公司失之交臂。

另外，这两种思路对应的代码都含有循环。含有循环的代码经常出的问题是在循环结束条件的判断。是该用小于还是小于等于？是该用 k 还是该用 k-1？由于题目要求的是从0开始计数，而我们的习惯思维是从1开始计数，因此首先要想好这些边界条件再开始编写代码，再者要在编写完代码之后再用边界值、边界值减1、边界值加1都运行一次（在纸上写代码就只能在心里运行了）。
扩展：和这道题类似的题目还有：输入一个单向链表。如果该链表的结点数为奇数，输出中间的结点；如果链表结点数为偶数，输出中间两个结点前面的一个。如果各位感兴趣，请自己分析并编写代码。

源码：

```
#include <iostream>
using namespace std;

typedef struct node
{  char data;
   struct node *next;
}LinkList;

void CreateList(LinkList *&t, char cstr[], int n)
```

```
{  LinkList *p, *r;
   int i;
   t=(LinkList*)malloc(sizeof(LinkList));
   t->next=NULL;
   r=t;
   for(i=0; i<n; i++)
   {  p=(LinkList*)malloc(sizeof(LinkList));
      p->data=cstr[i];
      r->next=p;
      r=p;
   }
   r->next=NULL;
}

void DispList(LinkList *t)
{  LinkList *p;
   if(t!=NULL)
   {  p=t->next;
      while(p!=NULL)
      {  printf("%c", p->data);
         p=p->next;
      }
      printf("\n");
   }
}

LinkList* GetKthFromTail(LinkList *t, int k)
{  if(t==NULL)
     return NULL;
   LinkList *before, *after;
   before=after=t->next;
   int i;
   for(i=0; i<k; i++)
   {  if(before->next!=NULL)
        before=before->next;
      else
        return NULL;
   }
   while(before->next!=NULL)  //注意循环结束条件，前一个结点没有后续结点了
   {   before=before->next;
       after=after->next;
   }
   return after;
}
```

```
int main()
{   char cstr[10]="huanggang";
    LinkList *t;
    CreateList(t, cstr, 9);
    cout<<GetKthFromTail(t,7)->data<<endl;

    system("pause");
    return 0;
}
```

10. 在排序数组中查找和为给定值得两个数字

题目：输入一个已经按升序排序过的数组和一个数字，在数组中查找两个数，使得它们的和正好是输入的那个数字。要求时间复杂度是 O(n)。如果有多对数字的和等于输入的数字，输出任意一对即可。例如输入数组1、2、4、7、11、15和数字15。由于4+11=15，因此输出4和11。

分析：如果我们不考虑时间复杂度，最简单想法的莫过去先在数组中固定一个数字，再依次判断数组中剩下的 n-1个数字与它的和是不是等于输入的数字。可惜这种思路需要的时间复杂度是 $O(n^2)$。

我们假设现在随便在数组中找到两个数。如果它们的和等于输入的数字，那太好了，我们找到了要找的两个数字；如果小于输入的数字呢？我们希望两个数字的和再大一点。由于数组已经排好序了，我们是不是可以把较小的数字的往后面移动一个数字？因为排在后面的数字要大一些，那么两个数字的和也要大一些，就有可能等于输入的数字了；同样，当两个数字的和大于输入的数字的时候，我们把较大 的数字往前移动，因为排在数组前面的数字要小一些，它们的和就有可能等于输入的数字了。

我们把前面的思路整理一下：最初我们找到数组的第一个数字和最后一个数字。当两个数字的和大于输入的数字时，把较大的数字往前移动；当两个数字的和小于数字时，把较小的数字往后移动；当相等时，打完收工。这样扫描的顺序是从数组的两端向数组的中间扫描。

如果输入的数组是没有排序的，但知道里面数字的范围，其他条件不变，如和在 O(n)时间里找到这两个数字？假设数组为 A，输入的数字为 n。low<=A[i]<=high。那么可以生成一个这样的数组 B，该数组的元素是一个结构体，结构体由一个 int 变量和一个 bool 变量组成。 结构体中的 int 变量取值从 low 到 high，bool 变量一开始全部置为0. 数组中的元素按照 int 变量进行升序排列。

然后进行一下操作：

1. 从头到尾扫描 A，根据 A[i]的取值，找到 B 中 int 值等于 A[i]的那个元素，并将 bool 变量置为1.
2. 这样一来，问题又转换成了最初的问题：在排序数组中查找和为给定值的两个数字这个问题，只不过，在找到两个元素的 int 值相加等于 n 后，还需要判断其 bool 值是否都为1，是才能输出。

怎么样，思路很巧妙吧！有了上述的分析，写出代码就不是很难了。对于如何优化数组 B，可以有不同的手段。一种思路是申请空间是(high-low)/8，以 bit 来表示 bool 类型。此种方法需要些转换来定位表示的[low,high]中的哪一个数字。但是总体来说，还是不难的。

源码：

```
  #include <iostream>
using namespace std;

int find(int r[], int n, int sum, int &a, int &b)
{   if(n<=0)
      return 0;
    int head=0, tail=n-1;
    while(head<tail)
    {  if((r[head]+r[tail])<sum)  //三种情况
          head++;
       else if((r[head]+r[tail])>sum)
          tail--;
       else
       {  a=head;
          b=tail;
          return 1;
       }
    }
    return 0;
}


int main()
{    int data[]={1,2,4,7,11,15};
     int a, b;
     a=b=0;
     if(find(data,6,15,a,b)){
        cout<<a<<" "<<b<<endl;
     }
     else cout<<"no answer"<<endl;

     system("pause");
     return 0;
}
```

11. 求二元查找树的镜像

题目： 输入一颗二元查找树，将该树转换为它的镜像，即在转换后的二元查找树中，左子树的结点都大于右子树的结点。用递归和循环两种方法完成树的镜像转换。

例如输入：

```
    8
   / \
 6      10
 /\      /\
5 7    9   11
```

输出：

```
     8
   / \
 10     6
 /\      /\
11   9   7 5
```

定义二元查找树的结点为：

```
struct BSTreeNode // a node in the binary search tree (BST)
{
    int          m_nValue; // value of node
    BSTreeNode *m_pLeft; // left child of node
    BSTreeNode *m_pRight; // right child of node
};
```

分析： 尽管我们可能一下子不能理解镜像是什么意思，但上面的例子给我们的直观感觉，就是交换结点的左右子树。我们试着在遍历例子中的二元查找树的同时来交换每个结点的左右子树。遍历时首先访问头结点8，我们交换它的左右子树得到：

```
      8
   / \
 10    6
 /\      /\
9   11 5 7
```

我们发现两个结点6和10的左右子树仍然是左结点的值小于右结点的值，我们再试着交换他们的左右子树，得到：

```
     8
   / \
 10     6
 /\      /\
11   9   7   5
```

刚好就是要求的输出。

上面的分析印证了我们的直觉：在遍历二元查找树时每访问到一个结点，交换它的左右子树。这种思路用递归不难实现，将遍历二元查找树的代码稍作修改就可以了。

源码：

```
#include <iostream>
using namespace std;

typedef struct node
{  int data;
   struct node *left;
   struct node *right;
}BSTNode;

int InsertBST(BSTNode *&bt, int k)
{  if(bt==NULL)
   {  bt=(BSTNode*)malloc(sizeof(BSTNode));
      bt->data=k;
      bt->left=bt->right=NULL;
      return 1;
   }
   if(bt->data==k)
   {  return 0;
   }
   if(bt->data<k)
   {  return InsertBST(bt->right, k);
   }
   if(bt->data>k)
   {  return InsertBST(bt->left, k);
   }
}

void InOrder(BSTNode *bt)
{  if(bt!=NULL)
   {  InOrder(bt->left);
      printf("%d ", bt->data);
      InOrder(bt->right);
   }
}

void MirrorBST(BSTNode *bt)
{  BSTNode *tmp;
   if(bt!=NULL)
   {  tmp=bt->left;
      bt->left=bt->right;
      bt->right=tmp;
      MirrorBST(bt->left);
      MirrorBST(bt->right);
   }
```

```
}

void CreateBST(BSTNode *&bt, int r[], int n)
{  int i;
   bt=NULL;
   for(i=0; i<n; i++)
     InsertBST(bt, r[i]);
}

int main()
{    int r[6]={6,3,4,7,1,9};
     BSTNode *bt;
     CreateBST(bt, r, 6);
     MirrorBST(bt);
     InOrder(bt);
     system("pause");
     return 0;
}
```

12. 从上往下遍历二元树

题目：输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印。

例如输入

```
     8
   / \
  6    10
 /\     /\
5 7   9 11
```

输出8　6　10　5　7　9　11。

分析：这曾是微软的一道面试题。这道题实质上是要求遍历一棵二元树，只不过不是我们熟悉的前序、中序或者后序遍历。

我们从树的根结点开始分析。自然先应该打印根结点8，同时为了下次能够打印8的两个子结点，我们应该在遍历到8时把子结点6和10保存到一个数据容器中。现在数据容器中就有两个元素6 和10了。按照从左往右的要求，我们先取出6访问。打印6的同时要把6的两个子结点5和7放入数据容器中，此时数据容器中有三个元素10、5和7。接下来我们应该从数据容器中取出结点10访问了。注意10比5和7先放入容器，此时又比5和7先取出，就是我们通常说的先入先出。因此不难看出这个数据容器的类型应该是个队列。

既然已经确定数据容器是一个队列，现在的问题变成怎么实现队列了。实际上我们无需自己动手实现一个，因为STL已经为我们实现了一个很好的deque（两端都可以进出的队列），我们只需要拿过来用就可以了。

我们知道树是图的一种特殊退化形式。同时如果对图的深度优先遍历和广度优先遍历有比较深刻的理解，将不难看出这种遍历方式实际上是一种广度优先遍历。因此这道题的本质是在二元树上实现广度优先遍历。

源码：

```
#include <iostream>
using namespace std;

#define MaxSize 100

typedef struct node
{  char data;
   struct node *lchild;
   struct node *rchild;
}BTNode;

void CreateBTNode(BTNode *&t, char *cstr)
{  BTNode *st[MaxSize], *p=NULL;
   int j=0,k,top=-1;
   t=NULL;
   char ch=cstr[j];
   while(ch!='\0')
   {  switch(ch)
      {  case '(':
              top++;
              st[top]=p;
              k=1;
              break;
        case ',':
             k=2;
             break;
        case ')':
             top--;
             break;
        default:
             p=(BTNode*)malloc(sizeof(BTNode));
             p->data=ch;
             p->lchild=p->rchild=NULL;
             if(t==NULL)
               t=p;
             else
             { switch(k)
               {  case 1: st[top]->lchild=p; break;
                  case 2: st[top]->rchild=p; break;
               }
             }
```

```
        }
        j++;
        ch=cstr[j];
    }
}

void LevelOrder(BTNode *t)
{  BTNode * qu[MaxSize], *p;
   int front, rear;
   front=rear=-1;
   if(t!=NULL)
   {  rear++;
      qu[rear]=t;
      while(rear!=front)
      {  front++;
         p=qu[front];
         printf("%c ", p->data);
         if(p->lchild!=NULL)
         {  rear++;
            qu[rear]=p->lchild;
         }
         if(p->rchild!=NULL)
         {  rear++;
            qu[rear]=p->rchild;
         }
      }
   }
}

int main()
{    char cstr[MaxSize]="D(B(A,C),F(E,G))";
     BTNode *t;
     CreateBTNode(t,cstr);
     LevelOrder(t);

     system("pause");
     return 0;
}
```

13. 第一个只出现一次的字符

题目：在一个字符串中找到第一个只出现一次的字符。如输入 abaccdeff，则输出 b。

分析：这道题是 2006 年 google 的一道笔试题。看到这道题时，最直观的想法是从头开始扫描这个字符串中的每个字符。当访问到某字符时拿这个字符和后面的每个字符相比较，如果在后面没有发现重复的字符，则该字符就是只出现一次的字符。如果字符串有 n 个字符，每个字符可能与后面的 O(n)个字符相比较，因此这种思路时间复杂度是 $O(n^2)$。我们试着去找一个更快的方法。

由于题目与字符出现的次数相关，我们是不是可以统计每个字符在该字符串中出现的次数？要达到这个目的，我们需要一个数据容器来存放每个字符的出现次数。在这个数据容器中可以根据字符来查找它出现的次数，也就是说这个容器的作用是把一个字符映射成一个数字。在常用的数据容器中，哈希表正是这个用途。

哈希表是一种比较复杂的数据结构。由于比较复杂，STL 中没有实现哈希表，因此需要我们自己实现一个。但由于本题的特殊性，我们只需要一个非常简单的哈希表就能满足要求。由于字符（char）是一个长度为 8 的数据类型，因此总共有可能 256 种可能。于是我们创建一个长度为 256 的数组，每个字母根据其 ASCII 码值作为数组的下标对应数组的对应项，而数组中存储的是每个字符对应的次数。这样我们就创建了一个大小为 256，以字符 ASCII 码为键值的哈希表。

我们第一遍扫描这个数组时，每碰到一个字符，在哈希表中找到对应的项并把出现的次数增加一次。这样在进行第二次扫描时，就能直接从哈希表中得到每个字符出现的次数了。

源码：

```
#include <iostream>
using namespace std;

#define MaxSize 256

char GetFirstNoRepeatChar(char* s)
{  int hash[256]={0}; // get a hash table, and initialize it
   if (s==NULL)
     return 0; // invalid input
   char *p=s;
   while(*p!='\0')
   { hash[*p]++; // get the how many times each char appears in the string
     p++;
   }
   p=s;
   while (*p!='\0') {
     if (hash[*p]==1)
        return *p; // find the first char which appears only once
     p++;
   }
   // if the string is empty
    // or every char in it appears at least twice
   return 0;
```

```
}
int main()
{   char cstr[6]="abbaw";
    cout<<GetFirstNoRepeatChar(cstr)<<endl;

     system("pause");
     return 0;
}
```

14. 圆圈中最后剩下的数字

题目：n 个数字（0,1,…,n-1）形成一个圆圈，从数字 0 开始，每次从这个圆圈中删除第 m 个数字（第一个为当前数字本身，第二个为当前数字的下一个数字）。当一个数字删除后，从被删除数字的下一个继续删除第 m 个数字。求出在这个圆圈中剩下的最后一个数字。

分析：本题就是有名的约瑟夫环问题。既然题目有一个数字圆圈，很自然的想法是我们用一个数据结构来模拟这个圆圈。在常用的数据结构中，我们很容易想到用环形列表。我们可以创建一个总共有 m 个数字的环形列表，然后每次从这个列表中删除第 m 个元素。

源码：

```
#include <iostream>
using namespace std;

int josephus(int n, int m)
{  int i, j, t=0, *p=new int[n];
   for (i=0; i<n; i++)
     p[i]=i+1;
   for (i=n; i>=2; i--)
   {  t=(t+m-1)%i;
      cout<<p[t]<<" ";
      for(j=t+1; j<i; j++)
        p[j-1]=p[j];
   }
   t=p[0];
   delete []p;
   return t;
}
int main()
{    cout<<josephus(6,2)<<endl;
     system("pause");
     return 0;
}
```

15. 含有指针成员的类的拷贝

题目： 下面是一个数组类的声明与实现。请分析这个类有什么问题，并针对存在的问题提出几种解决方案。

```
template<typename T> class Array
{
public:
	Array(unsigned arraySize):data(0), size(arraySize)
	{
		if(size > 0)
			data = new T[size];
	}
	~Array()
	{
		if(data) delete[] data;
	}
	void setValue(unsigned index, const T& value)
	{
		if(index < size)
			data[index] = value;
	}
	T getValue(unsigned index) const
	{
		if(index < size)
			return data[index];
		else
			return T();
	}
private:
	T* data;
	unsigned size;
};
```

分析： 我们注意在类的内部封装了用来存储数组数据的指针。软件存在的大部分问题通常都可以归结指针的不正确处理。这个类只提供了一个构造函数，而没有定义构造拷贝函数和重载拷贝运算符函数。当这个类的用户按照下面的方式声明并实例化该类的一个实例

```
Array A(10);
Array B(A);
```

或者按照下面的方式把该类的一个实例赋值给另外一个实例

```
Array A(10);
Array B(10);
B=A;
```

编译器将调用其自动生成的构造拷贝函数或者拷贝运算符的重载函数。在编译器生成的缺省的构造拷贝函数和拷贝运算符的重载函数，对指针实行的是按位拷贝，仅仅只是拷贝指针的地址，而不会拷贝指针的内容。因此在执行完前面的代码之后，A.data 和 B.data 指向的同一地址。当 A 或者 B 中任意一个结束其生命周期调用析构函数时，会删除 data。由于他们的 data 指向的是同一个地方，两个实例的 data 都被删除了。但另外一个实例并不知道它的 data 已经被删除了，当企图再次用它的 data 的时候，程序就会不可避免地崩溃。

由于问题出现的根源是调用了编译器生成的缺省构造拷贝函数和拷贝运算符的重载函数。一个最简单的办法就是禁止使用这两个函数。于是我们可以把这两个函数声明为私有函数，如果类的用户企图调用这两个函数，将不能通过编译。实现的代码如下：

```
private:
      Array(const Array& copy);
      const Array& operator = (const Array& copy);
```

最初的代码存在问题是因为不同实例的 data 指向的同一地址，删除一个实例的 data 会把另外一个实例的 data 也同时删除。因此我们还可以让构造拷贝函数或者拷贝运算符的重载函数拷贝的不只是地址，而是数据。由于我们重新存储了一份数据，这样一个实例删除的时候，对另外一个实例没有影响。这种思路我们称之为深度拷贝。实现的代码如下：

```
public:
      Array(const Array& copy):data(0), size(copy.size)
      {
            if(size > 0)
            {
                  data = new T[size];
                  for(int i = 0; i < size; ++ i)
                        setValue(i, copy.getValue(i));
            }
      }
      const Array& operator = (const Array& copy)
      {
            if(this == &copy)
                  return *this;
            if(data != NULL)
            {
                  delete []data;
                  data = NULL;
            }
            size = copy.size;
            if(size > 0)
            {
                  data = new T[size];
```

```
                for(int i = 0; i < size; ++ i)
                    setValue(i, copy.getValue(i));
            }
        }
```

为了防止有多个指针指向的数据被多次删除，我们还可以保存究竟有多少个指针指向该数据。只有当没有任何指针指向该数据的时候才可以被删除。这种思路通常被称之为引用计数技术。在构造函数中，引用计数初始化为 1；每当把这个实例赋值给其他实例或者以参数传给其他实例的构造拷贝函数的时候，引用计数加 1，因为这意味着又多了一个实例指向它的 data；每次需要调用析构函数或者需要把 data 赋值为其他数据的时候，引用计数要减 1，因为这意味着指向它的 data 的指针少了一个。当引用计数减少到 0 的时候，data 已经没有任何实例指向它了，这个时候就可以安全地删除。实现的代码如下：

```
public:
      Array(unsigned arraySize)
            :data(0), size(arraySize), count(new unsigned int)
      {
            *count = 1;
            if(size > 0)
                  data = new T[size];
      }
      Array(const Array& copy)
            : size(copy.size), data(copy.data), count(copy.count)
      {
            ++ (*count);
      }
      ~Array()
      {
            Release();
      }
      const Array& operator = (const Array& copy)
      {
            if(data == copy.data)
                  return *this;
            Release();
            data = copy.data;
            size = copy.size;
            count = copy.count;
            ++(*count);
      }
 private:
      void Release()
      {
            --(*count);
```

```
            if(*count == 0)
            {
                  if(data)
                  {
                        delete []data;
                        data = NULL;
                  }
                  delete count;
                  count = 0;
            }
      }
      unsigned int *count;
```

16. O(logn)求 Fibonacci 数列

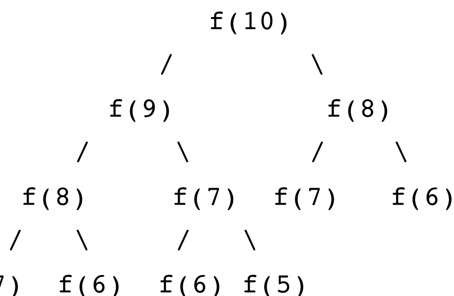
题目：定义 Fibonacci 数列如下：

```
         /  0                        n=0
f(n)=       1                        n=1
         \  f(n-1)+f(n-2)            n=2
```

输入 n，用最快的方法求该数列的第 n 项。

分析：在很多 C 语言教科书中讲到递归函数的时候，都会用 Fibonacci 作为例子。因此很多程序员对这道题的递归解法非常熟悉。

但是，教科书上反复用这个题目来讲解递归函数，并不能说明递归解法最适合这道题目。我们以求解 f(10)作为例子来分析递归求解的过程。要求得 f(10)，需要求得 f(9)和 f(8)。同样，要求得 f(9)，要先求得 f(8)和 f(7)……我们用树形结构来表示这种依赖关系



我们不难发现在这棵树中有很多结点会重复的，而且重复的结点数会随着 n 的增大而急剧增加。这意味这计算量会随着 n 的增大而急剧增大。事实上，用递归方法计算的时间复杂度是以 n 的指数的方式递增的。大家可以求 Fibonacci 的第 100 项试试，感受一下这样递归会慢到什么程度。在我的机器上，连续运行了一个多小时也没有出来结果。

其实改进的方法并不复杂。上述方法之所以慢是因为重复的计算太多，只要避免重复计算就行了。比如我们可以把已经得到的数列中间项保存起来，如果下次需要计算的时候我们先查找一下，如果前面已经计算过了就不用再次计算了。

更简单的办法是从下往上计算，首先根据 f(0)和 f(1)算出 f(2)，在根据 f(1)和 f(2)算出 f(3)……依此类推就可以算出第 n 项了。很容易理解，这种思路的时间复杂度是 O(n)。

这还不是最快的方法。下面介绍一种时间复杂度是O(logn)的方法。在介绍这种方法之前，先介绍一个数学公式：

$\{f(n), f(n-1), f(n-1), f(n-2)\} =\{1, 1, 1,0\}^{n-1}$

(注：{f(n+1), f(n), f(n), f(n-1)}表示一个矩阵。在矩阵中第一行第一列是f(n+1)，第一行第二列是f(n)，第二行第一列是f(n)，第二行第二列是f(n-1)。)

有了这个公式，要求得f(n)，我们只需要求得矩阵{1, 1, 1,0}的n-1次方，因为矩阵{1, 1, 1,0}的n-1次方的结果的第一行第一列就是f(n)。这个数学公式用数学归纳法不难证明。感兴趣的朋友不妨自己证明一下。

现在的问题转换为求矩阵{1, 1, 1, 0}的乘方。如果简单第从0开始循环，n次方将需要n次运算，并不比前面的方法要快。但我们可以考虑乘方的如下性质：

/ $a^{n/2}*a^{n/2}$ n为偶数时

a^n=

\ $a^{(n-1)/2}*a^{(n-1)/2}$ n为奇数时

要求得n次方，我们先求得n/2次方，再把n/2的结果平方一下。如果把求n次方的问题看成一个大问题，把求n/2看成一个较小的问题。这种把大问题分解成一个或多个小问题的思路我们称之为分治法。这样求n次方就只需要logn次运算了。

实现这种方式时，首先需要定义一个2×2的矩阵，并且定义好矩阵的乘法以及乘方运算。当这些运算定义好了之后，剩下的事情就变得非常简单。

源码：

```
/*2*2 矩阵类*/
class Matrix {
public:
    /*构造函数*/
    Matrix(__int64 a00 = 0, __int64 a01 = 0, __int64 a10 = 0, __int64 a11 = 0)
{
        this->a00 = a00;
        this->a01 = a01;
        this->a10 = a10;
        this->a11 = a11;
    }

    __int64 a00;
    __int64 a01;
    __int64 a10;
    __int64 a11;
};

Matrix multiply(const Matrix &a, const Matrix &b) {
    Matrix ans;
    /*根据题意要求，只需取最后四位数字即可，故取模 10000*/
    ans.a00 = (a.a00 * b.a00 + a.a01 * b.a10) % 10000;
    ans.a01 = (a.a00 * b.a01 + a.a01 * b.a11) % 10000;
```

```
    ans.a10 = (a.a10 * b.a00 + a.a11 * b.a10) % 10000;
    ans.a11 = (a.a10 * b.a01 + a.a11 * b.a11) % 10000;

    return ans;
}

/*求 2*2 矩阵[1 1 1 0]的 n（n >= 1）次幂。分治法求矩阵的幂 A[n]。
 *先求 A[n/2]，当然要分 n 的奇偶进行对应处理。
 */
Matrix pow(int n) {
    Matrix ans;
    if(n == 1) {
        ans.a00 = 1; ans.a01 = 1; ans.a10 = 1; ans.a11 = 0;
    } else if(n % 2 == 0) {
        ans = pow(n / 2);
        ans = multiply(ans, ans);
    } else if(n % 2 == 1) {
        ans = pow((n - 1) / 2);
        ans = multiply(ans, ans);
        Matrix one(1, 1, 1, 0);
        ans = multiply(ans, one);
    }

    return ans;
}

__int64 fac(int n) {
    if(n == 0)
        return 0;
    if(n == 1)
        return 1;

    Matrix ans = pow(n - 1);
    return ans.a00; //矩阵第一行第一列元素即为 fac[n]
}

int main()
{
    int n;
    while(cin >> n, n != -1) {
        int output;
        output = fac(n) % 10000;
```

```
        cout << output << endl;
    }

    system("pause");
     return 0;
}
```

17. 把字符串转换为整数

题目：输入一个表示整数的字符串，把该字符串转换成整数并输出。例如输入字符串"345"，则输出整数 345。

分析：这道题尽管不是很难，学过 C/C++语言一般都能实现基本功能，但不同程序员就这道题写出的代码有很大区别，可以说这道题能够很好地反应出程序员的思维和编程习惯，因此已经被包括微软在内的多家公司用作面试题。建议读者在往下看之前自己先编写代码，再比较自己写的代码和下面的参考代码有哪些不同。

首先我们分析如何完成基本功能，即如何把表示整数的字符串正确地转换成整数。还是以"345"作为例子。当我们扫描到字符串的第一个字符'3'时，我们不知道后面还有多少位，仅仅知道这是第一位，因此此时得到的数字是 3。当扫描到第二个数字'4'时，此时我们已经知道前面已经一个 3 了，再在后面加上一个数字 4，那前面的 3 相当于 30，因此得到的数字是 3*10+4=34。接着我们又扫描到字符'5'，我们已经知道了'5'的前面已经有了 34，由于后面要加上一个 5，前面的 34 就相当于 340 了，因此得到的数字就是 34*10+5=345。分析到这里，我们不能得出一个转换的思路：每扫描到一个字符，我们把在之前得到的数字乘以 10 再加上当前字符表示的数字。这个思路用循环不难实现。

由于整数可能不仅仅之含有数字，还有可能以'+'或者'-'开头，表示整数的正负。因此我们需要把这个字符串的第一个字符做特殊处理。如果第一个字符是'+'号，则不需要做任何操作；如果第一个字符是'-'号，则表明这个整数是个负数，在最后的时候我们要把得到的数值变成负数。

接着我们试着处理非法输入。由于输入的是指针，在使用指针之前，我们要做的第一件是判断这个指针是不是为空。如果试着去访问空指针，将不可避免地导致程序崩溃。另外，输入的字符串中可能含有不是数字的字符。每当碰到这些非法的字符，我们就没有必要再继续转换。最后一个需要考虑的问题是溢出问题。由于输入的数字是以字符串的形式输入，因此有可能输入一个很大的数字转换之后会超过能够表示的最大的整数而溢出。

现在已经分析的差不多了，开始考虑编写代码。首先我们考虑如何声明这个函数。由于是把字符串转换成整数，很自然我们想到：

```
int StrToInt(const char* str);
```

这样声明看起来没有问题。但当输入的字符串是一个空指针或者含有非法的字符时，应该返回什么值呢？0 怎么样？那怎么区分非法输入和字符串本身就是"0"这两种情况呢？接下来我们考虑另外一种思路。我们可以返回一个布尔值来指示输入是否有效，而把转换后的整数放到参数列表中以引用或者指针的形式传入。于是我们就可以声明如下：

```
bool StrToInt(const char *str, int& num);
```

这种思路解决了前面的问题。但是这个函数的用户使用这个函数的时候会觉得不是很方便，因为他不能直接把得到的整数赋值给其他整形变量，显得不够直观。

前面的第一种声明就很直观。如何在保证直观的前提下当碰到非法输入的时候通知用户呢？一种解决方案就是定义一个全局变量，每当碰到非法输入的时候，就标记该全局变量。用户在调用这个函数之后，就可以检验该全局变量来判断转换是不是成功。

源码：

```
#include <iostream>
using namespace std;

int ParseInt(char *s, int &num)
{   int sign=0;
    num=0;
    if(s==NULL)
      return 0;
    if(*s=='+')   // the first char in the string maybe '+' or '-'
    { sign=0; s++; }
    if(*s=='-')
    { sign=1; s++; }

    // the remaining chars in the string
    while(*s!='\0')
    {   if(*s>='0'&&*s<='9')
            num=num*10+*s-'0';
        else
          {num=0; return 0;}   // if the char is not a digit, invalid input
        s++;
    }
    if(sign==1)
      num=0-num;
    return 1;
}

int main()
{      char s[20]="-45678";
       int num;
       if(ParseInt(s, num))
          cout<<num<<endl;
       else
          cout<<"Invalid Param!"<<endl;

       system("pause");
       return 0;
}
```

18. 用两个栈实现一个队列

题目：某队列的声明如下：

```
template<typename T> class CQueue
{
public:
       CQueue() {}
       ~CQueue() {}
       void appendTail(const T& node);  // append a element to tail
       void deleteHead();               // remove a element from head
private:
      T> m_stack1;
      T> m_stack2;
};
```

分析：从上面的类的声明中，我们发现在队列中有两个栈。因此这道题实质上是要求我们用两个栈来实现一个队列。相信大家对栈和队列的基本性质都非常了解：栈是一种后入先出的数据容器，因此对队列进行的插入和删除操作都是在栈顶上进行；队列是一种先入先出的数据容器，我们总是把新元素插入到队列的尾部，而从队列的头部删除元素。

我们通过一个具体的例子来分析往该队列插入和删除元素的过程。首先插入一个元素 a，不妨把先它插入到 m_stack1。这个时候 m_stack1 中的元素有{a}，m_stack2 为空。再插入两个元素 b 和 c，还是插入到 m_stack1 中，此时 m_stack1 中的元素有{a,b,c}，m_stack2 中仍然是空的。

这个时候我们试着从队列中删除一个元素。按照队列先入先出的规则，由于 a 比 b、c 先插入到队列中，这次被删除的元素应该是 a。元素 a 存储在 m_stack1 中，但并不在栈顶上，因此不能直接进行删除。注意到 m_stack2 我们还一直没有使用过，现在是让 m_stack2 起作用的时候了。如果我们把 m_stack1 中的元素逐个 pop 出来并 push 进入 m_stack2，元素在 m_stack2 中的顺序正好和原来在 m_stack1 中的顺序相反。因此经过两次 pop 和 push 之后，m_stack1 为空，而 m_stack2 中的元素是{c,b,a}。这个时候就可以 pop 出 m_stack2 的栈顶 a 了。pop 之后的 m_stack1 为空，而 m_stack2 的元素为{c,b}，其中 b 在栈顶。

这个时候如果我们还想继续删除应该怎么办呢？在剩下的两个元素中 b 和 c，b 比 c 先进入队列，因此 b 应该先删除。而此时 b 恰好又在栈顶上，因此可以直接 pop 出去。这次 pop 之后，m_stack1 中仍然为空，而 m_stack2 为{c}。

从上面的分析我们可以总结出删除一个元素的步骤：当 m_stack2 中不为空时，在 m_stack2 中的栈顶元素是最先进入队列的元素，可以 pop 出去。如果 m_stack2 为空时，我们把 m_stack1 中的元素逐个 pop 出来并 push 进入 m_stack2。由于先进入队列的元素被压到 m_stack1 的底端，经过 pop 和 push 之后就处于 m_stack2 的顶端了，又可以直接 pop 出去。

接下来我们再插入一个元素 d。我们是不是还可以把它 push 进 m_stack1？这样会不会有问题呢？我们说不会有问题。因为在删除元素的时候，如果 m_stack2 中不为空，处于 m_stack2 中的栈顶元素是最先进入队列的，可以直接 pop；如果 m_stack2 为空，我们把 m_stack1 中的元素 pop 出来并 push 进入 m_stack2。由于 m_stack2 中元素的顺序和 m_stack1 相反，最先进入队列的元素还是处于 m_stack2 的栈顶，仍然可以直接 pop。不会出现任何矛盾。

源码：

```
#include <iostream>
using namespace std;

#define M 100

typedef struct
{   int data[M];
    int top;
}SqStack;

SqStack *in, *out;

void InitStack(SqStack *&s)
{   s=(SqStack*)malloc(sizeof(SqStack));
    s->top=-1;
}

int Push(SqStack *&s, int data)
{   if(s->top>=M-1)
        return 0;
    s->top++;
    s->data[s->top]=data;
    return 1;
}

int Pop(SqStack *&s, int &data)
{   if(s->top<0)
        return 0;
    data=s->data[s->top];
    s->top--;
    return 1;
}

int StackEmpty(SqStack *s)
{   return (s->top==-1);
}

void InitQueue()
{   InitStack(in);
    InitStack(out);
}
```

```
int QueueEmpty()
{   if(StackEmpty(in)&&StackEmpty(out))
        return 1;
    else
        return 0;
}

int enQueue(int data)
{   return Push(in, data);
}

int deQueue(int &data)
{   int tmp;
    if(QueueEmpty())
      return 0;
    else if(StackEmpty(out))
    { while(!StackEmpty(in))
      {   Pop(in ,tmp);
          printf("pop from in->%d\n", tmp);
          Push(out, tmp);
      }
    }
    Pop(out,data);
    return 1;
}



int main()
{
     InitQueue();
     int data;
     enQueue(5);
     enQueue(25);
     enQueue(88);
     deQueue(data);
     printf("%d\n",data);
     system("pause");
    return 0;
}
```

19. 反转链表

题目： 输入一个链表的头结点，反转该链表，并返回反转后链表的头结点。链表结点定义如下：

```
struct ListNode
{
      int        m_nKey;
      ListNode* m_pNext;
};
```

分析： 这是一道广为流传的微软面试题。由于这道题能够很好的反应出程序员思维是否严密，在微软之后已经有很多公司在面试时采用了这道题。

为了正确地反转一个链表，需要调整指针的指向。与指针操作相关代码总是容易出错的，因此最好在动手写程序之前作全面的分析。在面试的时候不急于动手而是一开始做仔细的分析和设计，将会给面试官留下很好的印象，因为在实际的软件开发中，设计的时间总是比写代码的时间长。与其很快地写出一段漏洞百出的代码，远不如用较多的时间写出一段健壮的代码。

为了将调整指针这个复杂的过程分析清楚，我们可以借助图形来直观地分析。假设下图中 l、m 和 n 是三个相邻的结点：

a✍b✍…✍l m→n→…

假设经过若干操作，我们已经把结点 l 之前的指针调整完毕，这些结点的 m_pNext 指针都指向前面一个结点。现在我们遍历到结点 m。当然，我们需要把调整结点的 m_pNext 指针让它指向结点 l。但注意一旦调整了指针的指向，链表就断开了，如下图所示：

a✍b✍…l✍m n→…

因为已经没有指针指向结点 n，我们没有办法再遍历到结点 n 了。因此为了避免链表断开，我们需要在调整 m 的 m_pNext 之前要把 n 保存下来。

接下来我们试着找到反转后链表的头结点。不难分析出反转后链表的头结点是原始链表的尾位结点。什么结点是尾结点？就是 m_pNext 为空指针的结点。

源码：

```
#include <iostream>
using namespace std;

typedef struct node
{   int data;
    struct node *next;
}LinkList;

void CreateList(LinkList *&L, int arr[], int n)
{   LinkList *p,*r;
    L=(LinkList*)malloc(sizeof(LinkList));
    L->next=NULL;
    r=L;
```

```
    int i;
    for(i=0; i<n; i++)
    {   p=(LinkList*)malloc(sizeof(LinkList));
        p->data=arr[i];
        p->next=r->next;
        r->next=p;
        r=p;
    }
    r->next=NULL;
}

void DispList(LinkList *L)
{   LinkList *p;
    if(L==NULL)
      return;
    p=L->next;
    while(p!=NULL)
    {   cout<<p->data<<" ";
        p=p->next;
    }
    cout<<endl;
}

LinkList *ReverseList(LinkList *L)
{   LinkList *p, *q;
    if(L==NULL)
      return NULL;
    p=L->next;
    L->next=NULL;   //断开头结点
    while(p!=NULL)   //头插法，逆序
    {   q=p->next;
        p->next=L->next;
        L->next=p;
        p=q;
    }
    return L;
}

int main()
{    LinkList *L;
     int r[4]={1,2,3,4};
     CreateList(L, r, 4);
```

```
    DispList(L);
    L=ReverseList(L);
    DispList(L);

    system("pause");
  return 0;
}
```

20. 最长公共子串

题目：如果字符串一的所有字符按其在字符串中的顺序出现在另外一个字符串二中，则字符串一称之为字符串二的子串。注意，并不要求子串（字符串一）的字符必须连续出现在字符串二中。请编写一个函数，输入两个字符串，求它们的最长公共子串，并打印出最长公共子串。

例如：输入两个字符串 BDCABA 和 ABCBDAB，字符串 BCBA 和 BDAB 都是是它们的最长公共子串，则输出它们的长度 4，并打印任意一个子串。

分析：求最长公共子串（Longest Common Subsequence, LCS）是一道非常经典的动态规划题，因此一些重视算法的公司像 MicroStrategy 都把它当作面试题。

完整介绍动态规划将需要很长的篇幅，因此我不打算在此全面讨论动态规划相关的概念，只集中对 LCS 直接相关内容作讨论。如果对动态规划不是很熟悉，请参考相关算法书比如算法讨论。

先介绍 LCS 问题的性质：记 $X_m=\{x_0, x_1,\ldots x_{m-1}\}$和 $Y_n=\{y_0,y_1,\ldots,y_{n-1}\}$为两个字符串，而 $Z_k=\{z_0,z_1,\ldots z_{k-1}\}$是它们的 LCS，则：

1. 如果 $x_{m-1}=y_{n-1}$，那么 $z_{k-1}=x_{m-1}=y_{n-1}$，并且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的 LCS；
2. 如果 $x_{m-1}\neq y_{n-1}$，那么当 $z_{k-1}\neq x_{m-1}$ 时 Z 是 X_{m-1} 和 Y 的 LCS；
3. 如果 $x_{m-1}\neq y_{n-1}$，那么当 $z_{k-1}\neq y_{n-1}$ 时 Z 是 Y_{n-1} 和 X 的 LCS；

下面简单证明一下这些性质：

1. 如果 $z_{k-1}\neq x_{m-1}$，那么我们可以把 x_{m-1}（y_{n-1}）加到 Z 中得到 Z'，这样就得到 X 和 Y 的一个长度为 k+1 的公共子串 Z'。这就与长度为 k 的 Z 是 X 和 Y 的 LCS 相矛盾了。因此一定有 $z_{k-1}=x_{m-1}=y_{n-1}$。

既然 $z_{k-1}=x_{m-1}=y_{n-1}$，那如果我们删除 z_{k-1}（x_{m-1}、y_{n-1}）得到的 Z_{k-1}，X_{m-1} 和 Y_{n-1}，显然 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个公共子串，现在我们证明 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的 LCS。用反证法不难证明。假设有 X_{m-1} 和 Y_{n-1} 有一个长度超过 k-1 的公共子串 W，那么我们把加到 W 中得到 W'，那 W'就是 X 和 Y 的公共子串，并且长度超过 k，这就和已知条件相矛盾了。

2. 还是用反证法证明。假设 Z 不是 X_{m-1} 和 Y 的 LCS，则存在一个长度超过 k 的 W 是 X_{m-1} 和 Y 的 LCS，那 W 肯定也 X 和 Y 的公共子串，而已知条件中 X 和 Y 的公共子串的最大长度为 k。矛盾。

3. 证明同 2。

有了上面的性质，我们可以得出如下的思路：求两字符串 $X_m=\{x_0, x_1,\ldots x_{m-1}\}$和 $Y_n=\{y_0,y_1,\ldots,y_{n-1}\}$的 LCS，如果 $x_{m-1}=y_{n-1}$，那么只需求得 X_{m-1} 和 Y_{n-1} 的 LCS，并在其后添加 x_{m-1}（y_{n-1}）即可；如果 $x_{m-1}\neq y_{n-1}$，我们分别求得 X_{m-1} 和 Y 的 LCS 和 Y_{n-1} 和 X 的 LCS，并且这两个 LCS 中较长的一个为 X 和 Y 的 LCS。

如果我们记字符串 X_i 和 Y_j 的 LCS 的长度为 c[i,j]，我们可以递归地求 c[i,j]：

```
          /        0                                    if i<0 or j<0
c[i,j]=            c[i-1,j-1]+1                         if i,j>=0 and xi=xj
          \        max(c[i,j-1],c[i-1,j]                if i,j>=0 and xi≠xj
```

上面的公式用递归函数不难求得。但从前面求 Fibonacci 第 n 项(本面试题系列第 16 题）的分析中我们知道直接递归会有很多重复计算，我们用从底向上循环求解的思路效率更高。为了能够采用循环求解的思路，我们用一个矩阵（参考代码中的 LCS_length）保存下来当前已经计算好了的 c[i,j]，当后面的计算需要这些数据时就可以直接从矩阵读取。另外，求取 c[i,j]可以从 c[i-1,j-1] 、c[i,j-1]或者 c[i-1,j]三个方向计算得到，相当于在矩阵 LCS_length 中是从 c[i-1,j-1]，c[i,j-1]或者 c[i-1,j]的某一个各自移动到 c[i,j]，因此在矩阵中有三种不同的移动方向：向左、向上和向左上方，其中只有向左上方移动时才表明找到 LCS 中的一个字符。于是我们需要用另外一个矩阵（参考代码中的 LCS_direction）保存移动的方向。

21. 左旋字符串

题目：定义字符串的左旋转操作：把字符串前面的若干个字符移动到字符串的尾部。如把字符串 abcdef 左旋转 2 位得到字符串 cdefab。请实现字符串左旋转的函数。要求时间对长度为 n 的字符串操作的复杂度为 O(n)，辅助内存为 O(1)。

分析：如果不考虑时间和空间复杂度的限制，最简单的方法莫过于把这道题看成是把字符串分成前后两部分，通过旋转操作把这两个部分交换位置。于是我们可以新开辟一块长度为 n+1 的辅助空间，把原字符串后半部分拷贝到新空间的前半部分，在把原字符串的前半部分拷贝到新空间的后半部分。不难看出，这种思路的时间复杂度是 O(n)，需要的辅助空间也是 O(n)。

接下来的一种思路可能要稍微麻烦一点。我们假设把字符串左旋转 m 位。于是我们先把第 0 个字符保存起来，把第 m 个字符放到第 0 个的位置，在把第 2m 个字符放到第 m 个的位置…依次类推，一直移动到最后一个可以移动字符，最后在把原来的第 0 个字符放到刚才移动的位置上。接着把第 1 个字符保存起来，把第 m+1 个元素移动到第 1 个位置…重复前面处理第 0 个字符的步骤，直到处理完前面的 m 个字符。

该思路还是比较容易理解，但当字符串的长度 n 不是 m 的整数倍的时候，写程序会有些麻烦，感兴趣的朋友可以自己试一下。由于下面还要介绍更好的方法，这种思路的代码我就不提供了。

我们还是把字符串看成有两段组成的，记位 XY。左旋转相当于要把字符串 XY 变成 YX。我们先在字符串上定义一种翻转的操作，就是翻转字符串中字符的先后顺序。把 X 翻转后记为 X^T。显然有$(X^T)^T=X$。

我们首先对 X 和 Y 两段分别进行翻转操作，这样就能得到 X^TY^T。接着再对 X^TY^T 进行翻转操作，得到$(X^TY^T)^T=(Y^T)^T(X^T)^T=YX$。正好是我们期待的结果。

分析到这里我们再回到原来的题目。我们要做的仅仅是把字符串分成两段，第一段为前面 m 个字符，其余的字符分到第二段。再定义一个翻转字符串的函数，按照前面的步骤翻转三次就行了。时间复杂度和空间复杂度都合乎要求。

源码：

22. 整数的二进制表示中 1 的个数

题目：输入一个整数，求该整数的二进制表达中有多少个 1。例如输入 10，由于其二进制表示为 1010，有两个 1，因此输出 2。

分析：这是一道很基本的考查位运算的面试题。包括微软在内的很多公司都曾采用过这道题。一个很基本的想法是，我们先判断整数的最右边一位是不是 1。接着把整数右移一位，原来处于右边第二位的数字现在被移到第一位了，再判断是不是 1。这样每次移动一位，直到这个整数变成 0 为止。现在的问题变成怎样判断一个整数的最右边一位是不是 1 了。很简单，如果它和整数 1 作与运算。由于 1 除了最右边一位以外，其他所有位都为 0。因此如果与运算的结果为 1，表示整数的最右边一位是 1，否则是 0。

可能有读者会问，整数右移一位在数学上是和除以 2 是等价的。那可不可以把上面的代码中的右移运算符换成除以 2 呢？答案是最好不要换成除法。因为除法的效率比移位运算要低的多，在实际编程中如果可以应尽可能地用移位运算符代替乘除法。

这个思路当输入 i 是正数时没有问题，但当输入的 i 是一个负数时，不但不能得到正确的 1 的个数，还将导致死循环。以负数 0x80000000 为例，右移一位的时候，并不是简单地把最高位的 1 移到第二位变成 0x40000000，而是 0xC0000000。这是因为移位前是个负数，仍然要保证移位后是个负数，因此移位后的最高位会设为 1。如果一直做右移运算，最终这个数字就会变成 0xFFFFFFFF 而陷入死循环。

为了避免死循环，我们可以不右移输入的数字 i。首先 i 和 1 做与运算，判断 i 的最低位是不是为 1。接着把 1 左移一位得到 2，再和 i 做与运算，就能判断 i 的次高位是不是 1……这样反复左移，每次都能判断 i 的其中一位是不是 1。
另外一种思路是如果一个整数不为 0，那么这个整数至少有一位是 1。如果我们把这个整数减去 1，那么原来处在整数最右边的 1 就会变成 0，原来在 1 后面的所有的 0 都会变成 1。其余的所有位将不受到影响。举个例子：一个二进制数 1100，从右边数起的第三位是处于最右边的一个 1。减去 1 后，第三位变成 0，它后面的两位 0 变成 1，而前面的 1 保持不变，因此得到结果是 1011。

我们发现减 1 的结果是把从最右边一个 1 开始的所有位都取反了。这个时候如果我们再把原来的整数和减去 1 之后的结果做与运算，从原来整数最右边一个 1 那一位开始所有位都会变成 0。如 1100&1011=1000。也就是说，把一个整数减去 1，再和原整数做与运算，会把该整数最右边一个 1 变成 0。那么一个整数的二进制有多少个 1，就可以进行多少次这样的操作。

源码：

```
#include <iostream>
using namespace std;

int CountsOneNumber(int n)
{   int cnt=0;
    while(n)
    {   n=n&(n-1);
        cnt++;
    }
    return cnt;
}
```

```
int main()
{     int n=9;
      cout<<CountsOneNumber(n)<<endl;
      system("pause");
      return 0;
}
```

23. 跳台阶问题

题目：一个台阶总共有 n 级，如果一次可以跳 1 级，也可以跳 2 级。求总共有多少总跳法，并分析算法的时间复杂度。

分析：这道题最近经常出现，包括 MicroStrategy 等比较重视算法的公司都曾先后选用过个这道题作为面试题或者笔试题。

首先我们考虑最简单的情况。如果只有 1 级台阶，那显然只有一种跳法。如果有 2 级台阶，那就有两种跳的方法了：一种是分两次跳，每次跳 1 级；另外一种就是一次跳 2 级。现在我们再来讨论一般情况。我们把 n 级台阶时的跳法看成是 n 的函数，记为 f(n)。当 n>2 时，第一次跳的时候就有两种不同的选择：一是第一次只跳 1 级，此时跳法数目等于后面剩下的 n-1 级台阶的跳法数目，即为 f(n-1)；另外一种选择是第一次跳 2 级，此时跳法数目等于后面剩下的 n-2 级台阶的跳法数目，即为 f(n-2)。因此 n 级台阶时的不同跳法的总数 f(n)=f(n-1)+(f-2)。

我们把上面的分析用一个公式总结如下：

```
        /   1                         n=1
f(n)=       2                         n=2
        \   f(n-1)+(f-2)              n>2
```

分析到这里，相信很多人都能看出这就是我们熟悉的 Fibonacci 序列。至于怎么求这个序列的第 n 项，请参考本面试题系列第 16 题，这里就不在赘述了。

24. 栈的 push 和 pop 序列

题目：输入两个整数序列。其中一个序列表示栈的 push 顺序，判断另一个序列有没有可能是对应的 pop 顺序。为了简单起见，我们假设 push 序列的任意两个整数都是不相等的。比如输入的 push 序列是 1、2、3、4、5，那么 4、5、3、2、1 就有可能是一个 pop 系列。因为可以有如下的 push 和 pop 序列：push 1，push 2，push 3，push 4，pop，push 5，pop，pop，pop，pop，这样得到的 pop 序列就是 4、5、3、2、1。但序列 4、3、5、1、2 就不可能是 push 序列 1、2、3、4、5 的 pop 序列。

分析：这道题除了考查对栈这一基本数据结构的理解，还能考查我们的分析能力。这道题的一个很直观的想法就是建立一个辅助栈，每次 push 的时候就把一个整数 push 进入这个辅助栈，同样需要 pop 的时候就把该栈的栈顶整数 pop 出来。判断栈顶元素与 pop 元素是否相等，不等则 push，相等则出栈，继续比较栈顶元素与 pop 元素。

来看看序列 4、3、5、1、2。第一个希望被 pop 出来的数字是 4，因此 4 需要先 push 到栈里面。由于 push 的顺序已经由 push 序列确定了，也就是在把 4 push 进栈之前，数字 1，2，3 都需要 push 到栈里面。此时栈里的包含 4 个数字，分别是 1，2，3，4，其中 4

位于栈顶。把 4 pop 出栈后，剩下三个数字 1，2，3。把 4 pop 出来之后，3 位于栈顶，直接 pop。接下来希望 pop 的数字是 5，由于 5 不是栈顶数字，我们到 push 序列中没有被 push 进栈的数字中去搜索该数字，幸运的时候能够找到 5，于是把 5 push 进入栈。此时 pop 5 之后，栈内包含两个数字 1、2，其中 2 位于栈顶。这个时候希望 pop 的数字是 1，由于不是栈顶数字，我们需要到 push 序列中还没有被 push 进栈的数字中去搜索该数字。但此时 push 序列中所有数字都已被 push 进入栈，因此该序列不可能是一个 pop 序列。也就是说，如果我们希望 pop 的数字正好是栈顶数字，直接 pop 出栈即可；如果希望 pop 的数字目前不在栈顶，我们就到 push 序列中还没有被 push 到栈里的数字中去搜索这个数字，并把在它之前的所有数字都 push 进栈。如果所有的数字都被 push 进栈仍然没有找到这个数字，表明该序列不可能是一个 pop 序列。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

#define SIZE 5

int judge(int spush[], int spop[])
{   int PushCnt=0, PopCnt=0;
    stack<int> st;
    while(PushCnt<SIZE)
    {   st.push(spush[PushCnt]);
        while(!st.empty()&&PopCnt!=SIZE&&st.top()==spop[PopCnt])
        {   PopCnt++;
            st.pop();
        }
        PushCnt++;
    }
    if(PopCnt==SIZE)
       return 1;
    else
       return 0;
}

int main()
{    int n=9;
     int Spush[SIZE],Spop[SIZE];
     for(int i=0;i<SIZE;i++)
         cin>>Spush[i];
     for(int i=0;i<SIZE;i++)
         cin>>Spop[i];
     if(judge(Spush,Spop)) cout<<"Yes"<<endl;
```

```
    else cout<<"No"<<endl;

    system("pause");
    return 0;
}
```

25. 在 1 到 n 的正数中输出 1 出现的次数

题目：输入一个整数 n，求从 1 到 n 这 n 个整数的十进制表示中 1 出现的次数。例如输入 12，从 1 到 12 这些整数中包含 1 的数字有 1，10，11 和 12，1 一共出现了 5 次。

分析：这是一道广为流传的 google 面试题。用最直观的方法求解并不是很难，但遗憾的是效率不是很高；而要得出一个效率较高的算法，需要比较强的分析能力，并不是件很容易的事情。当然，google 的面试题中简单的也没有几道。

我们用一个稍微大一点的数字 21345 作为例子来分析。我们把从 1 到 21345 的所有数字分成两段，即 1-1345 和 1346-21345。先来看 1346-21345 中 1 出现的次数。1 的出现分为两种情况：一种情况是 1 出现在最高位（万位）。从 1 到 21345 的数字中，1 出现在 10000-19999 这 10000 个数字的万位中，一共出现了 10000（10^4）次；另外一种情况是 1 出现在除了最高位之外的其他位中。例子中 1346-21345，这 20000 个数字中后面四位中 1 出现的次数是 2000 次（$2*10^3$，其中 2 是第一位的数值，10^3 是因为数字的后四位数字其中一位为 1，其余的三位数字可以在 0 到 9 这 10 个数字任意选择，由排列组合可以得出总次数是 $2*10^3$）。

至于从 1 到 1345 的所有数字中 1 出现的次数，我们就可以用递归地求得了。这也是我们为什么要把 1-21345 分为 1-1235 和 1346-21345 两段的原因。因为把 21345 的最高位去掉就得到 1345，便于我们采用递归的思路。

分析到这里还有一种特殊情况需要注意：前面我们举例子是最高位是一个比 1 大的数字，此时最高位 1 出现的次数 10^4（对五位数而言）。但如果最高位是 1 呢？比如输入 12345，从 10000 到 12345 这些数字中，1 在万位出现的次数就不是 10^4 次，而是 2346 次了，也就是除去最高位数字之后剩下的数字再加上 1。

源码：

这道题，我认为是总结一个递推公式，然后用递推法实现，比较好。可以推出下列递推公式:

```
/*
 * f(n)=(a>1?s:n-s*a+1)+a*f(s-1)+f(n-s*a)当 n>9 时;
 * L 是 n 的位数
 * a 是 n 的第一位数字
 * s 是 10 的 L－1 次方
 * n-s*a 求的是 a 后面的数.
 * 公式说明：
 * 求 0-n 由多少个数字 1,分三部分,一是所有数中第一位有多少个 1,对应(a>1?s:n-s*a+1)
 * 当 a 大于 1 是,应该有 a 的 L1 次， a 小于 1 是有 n-s*a+1。
 * 如 n 是 223 所有数中第一位有 1 是 100;n 是 123 所有数中第一位是 1 的有 24
 * 二是 对应 a*f(s-1） 如 n 是 223 应该有 2*f(99)个 1
 * 三是 对应 f(n-s*a) 如 n 是 223 应该有 f(23)个 1。
```

```
 */

long f(long n)
{   if(n<9)
      return (n>0)?1:0;
   long a,s;
   int L;
   L=(int)(log10(n*1.0)+1);
   s=(long)pow(10, (L-1)*1.0);
   a=(long)n/s;
   return (a>1?s:n-s*a+1)+a*f(s-1)+f(n-s*a);
}
```

26. 和为 n 的连续正数序列

题目：输入一个正数 n，输出所有和为 n 连续正数序列。例如输入 15，由于 1+2+3+4+5=4+5+6=7+8=15，所以输出 3 个连续序列 1-5、4-6 和 7-8。

分析：这是网易的一道面试题。这道题和本面试题系列的第 10 题有些类似。我们用两个数 small 和 big 分别表示序列的最小值和最大值。首先把 small 初始化为 1，big 初始化为 2。如果从 small 到 big 的序列的和大于 n 的话，我们向右移动 small，相当于从序列中去掉较小的数字。如果从 small 到 big 的序列的和小于 n 的话，我们向右移动 big，相当于向序列中添加 big 的下一个数字。一直到 small 等于(1+n)/2，因为序列至少要有两个数字。

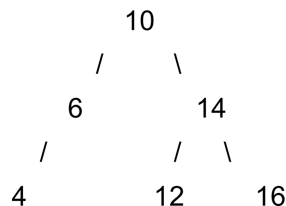
源码：

```
#include <iostream>
#include <stack>
using namespace std;

void PrintContinuousSequence(int small, int big);

void FindContinuousSequence(int n)
{         if(n < 3)
                  return;
         int small = 1;
         int big = 2;
         int middle = (1 + n) / 2;
         int sum = small + big;
         while(small < middle)
         {        // we are lucky and find the sequence
                  if(sum == n)
                           PrintContinuousSequence(small, big);
                  // if the current sum is greater than n,
```

```
            // move small forward
            while(sum > n)
            {
                sum -= small;
                small ++;
                // we are lucky and find the sequence
                if(sum == n)
                    PrintContinuousSequence(small, big);
            }
            // move big forward
            big ++;
            sum += big;
        }
}

void PrintContinuousSequence(int small, int big)
{   for(int i = small; i <= big; ++ i)
        printf("%d ", i);
    printf("\n");
}

int main()
{   FindContinuousSequence(15);
    system("pause");
    return 0;
}
```

27. 二元树的深度

题目：输入一棵二元树的根结点，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。例如：输入二元树：



输出该树的深度 3。

二元树的结点定义如下：

```
struct SBinaryTreeNode // a node of the binary tree
{       int       m_nValue; // value of node
        SBinaryTreeNode    *m_pLeft;    // left child of node
        SBinaryTreeNode    *m_pRight; // right child of node
};
```

分析：这道题本质上还是考查二元树的遍历。题目给出了一种树的深度的定义。当然，我们可以按照这种定义去得到树的所有路径，也就能得到最长路径以及它的长度。只是这种思路用来写程序有点麻烦。

我们还可以从另外一个角度来理解树的深度。如果一棵树只有一个结点，它的深度为 1。如果根结点只有左子树而没有右子树，那么树的深度应该是其左子树的深度加 1；同样如果根结点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加 1。如果既有右子树又有左子树呢？那该树的深度就是其左、右子树深度的较大值再加 1。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

#define MaxSize 100

typedef struct node
{   char data;
    struct node *lchild, *rchild;
}BTNode;

void CreateBTNode(BTNode *&b, char *s)
{   BTNode *st[MaxSize], *p=NULL;
    int top=-1,k,j=0;
    char ch=s[j];
    b=NULL;
    while(ch!='\0')
    {   switch(ch)
        {   case '(': top++; st[top]=p; k=1; break;
            case ',':k=2; break;
            case ')':top--; break;
            default: p=(BTNode*)malloc(sizeof(BTNode));
                        p->data=ch;
                        p->lchild=p->rchild=NULL;
                        if(b==NULL)
                          b=p;
                        else
                        {   switch(k)
                            {   case 1: st[top]->lchild=p; break;
                                case 2: st[top]->rchild=p; break;
                            }
                        }
        }
        j++;
```

```
            ch=s[j];
        }
}

int Depth(BTNode *b)
{   int lh, rh;
    if(b==NULL)
        return 0;
    else
    {   lh=Depth(b->lchild);
        rh=Depth(b->rchild);
    }
    return (lh>rh)?(lh+1):(rh+1);
}

int main()
{     char s[100]="A(B(C,D),G(,E))";
      BTNode *b;
      CreateBTNode(b, s);
      cout<<Depth(b)<<endl;

      system("pause");
      return 0;
}
```

28. 字符串的排列

题目：输入一个字符串，打印出该字符串中字符的所有排列。例如输入字符串 abc，则输出由字符 a、b、c 所能排列出来的所有字符串 abc、acb、bac、bca、cab 和 cba。

分析：这是一道很好的考查对递归理解的编程题，因此在过去一年中频繁出现在各大公司的面试、笔试题中。

我们以三个字符abc为例来分析一下求字符串排列的过程。首先我们固定第一个字符a，求后面两个字符 bc 的排列。当两个字符 bc 的排列求好之后，我们把第一个字符 a 和后面的 b 交换，得到 bac，接着我们固定第一个字符 b，求后面两个字符 ac 的排列。现在是把 c 放到第一位置的时候了。记住前面我们已经把原先的第一个字符 a 和后面的 b 做了交换，为了保证这次 c 仍然是和原先处在第一位置的 a 交换，我们在拿 c 和第一个字符交换之前，先要把 b 和 a 交换回来。在交换 b 和 a 之后，再拿 c 和处在第一位置的 a 进行交换，得到 cba。我们再次固定第一个字符 c，求后面两个字符 b、a 的排列。

既然我们已经知道怎么求三个字符的排列，那么固定第一个字符之后求后面两个字符的排列，就是典型的递归思路了。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

#define MaxSize 100

void Permutation(char *cstr, char *begin)
{   if(cstr==NULL||begin==NULL)
        return;
     if(*begin=='\0')
        printf("%s\n", cstr);
     char *p, tmp;
     for(p=begin; *p!='\0'; p++)
     {   tmp=*p;
         *p=*begin;
         *begin=tmp;
         Permutation(cstr, begin+1);
         tmp=*p;
         *p=*begin;
         *begin=tmp;
     }
}

int main()
{    char s[100]="abcd";
     Permutation(s, s);

     system("pause");
     return 0;
}
```

29. 调整数组顺序使得奇数位于偶数前面

题目：输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。要求时间复杂度为O(n)。

分析：如果不考虑时间复杂度，最简单的思路应该是从头扫描这个数组，每碰到一个偶数时，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，这时把该偶数放入这个空位。由于碰到一个偶数，需要移动 O(n)个数字，因此总的时间复杂度是$O(n^2)$。

要求的是把奇数放在数组的前半部分，偶数放在数组的后半部分，因此所有的奇数应该位于偶数的前面。也就是说我们在扫描这个数组的时候，如果发现有偶数出现在奇数的前面，我们可以交换他们的顺序，交换之后就符合要求了。

因此我们可以维护两个指针，第一个指针初始化为数组的第一个数字，它只向后移动；第二个指针初始化为数组的最后一个数字，它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是偶数而第二个指针指向的数字是奇数，我们就交换这两个数字。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

#define MaxSize 100

void OddEvenSplit(int r[], int n)
{   if(r==NULL||n<=0)
        Return;
    int i,j,tmp;
    i=0;
    j=n-1;
    while(i!=j)
    {   while(j>i&&r[j]%2==0)
            j--;
        while(i<j&&r[i]%2==1)
            i++;
        if(i<=j)
        {   tmp=r[i];
            r[i]=r[j];
            r[j]=tmp;
        }
    }
}

int main()
{    int a[] = {3, 4, 7, 9, 10, 8, 4 , 1 ,2, 11};
     OddEvenSplit(a, 10);
     for (int i = 0; i < 10; i++)
        cout << a[i] << ' ';
     cout << endl;
     system("pause");
     return 0;
}
```

30. 赋值运算符重载函数

问题：给出如下 CMyString 的声明，要求为该类型添加赋值运算符函数。

```
class CMyString
{
public:
CMyString(char* pData = NULL);
CMyString(const CMyString& str);
~CMyString(void);

private:
char* m_pData;
};
```

当面试官要求应聘者定义一个复制运算符函数时，他会关注如下几点：

- 是否把返回值的类型声明为该类型的引用，并在函数结束前返回实例自身（即*this）的引用？只有返回一个引用，才可以允许连续赋值。否则如果函数的返回值是 void，假设有三个 CMyString 的对象，str1、str2 和 str3，在程序中语句 str1=str2=str3 将不能通过编译。
- 是否把传入的参数的类型声明为常量引用？如果传入的参数不是引用而是实例，那么从形参到实参会调用一次构造拷贝函数。把参数申明为引用可以避免这样的无谓消耗，能提高代码的效率。同时，我们在赋值运算符函数内是不会改变传入的实例的状态的，因此应该为传入的引用参数加上 const 关键字。
- 是否记得释放实例自身已有的内存？如果忘了在分配新内存之前释放自身已有的空间，将出现内存泄露。
- 是否判断传入的参数是不是和当前的实例（*this）是不是同一个实例？如果是同一个，则不进行赋值操作，直接返回。如果事先不判断，就进行赋值，那么在释放实例自身的内存的时候就会导致严重的问题：当*this 和传入的参数是同一个实例时，那么一旦释放了自身的内存，传入的参数的内存也同时被释放了，因此再也找不到需要赋值的内容了。

当我们完整地考虑了上述几方面之后，我们可以写出如下的代码：

```
CMyString& CMyString::operator =(const CMyString &str)
{
if(this == &str)
return *this;
delete []m_pData;
m_pData = NULL;
m_pData = new char[strlen(str.m_pData) + 1];
strcpy(m_pData, str.m_pData);
return *this;
}
```

这是一般 C++教材上提供的参考代码。如果是面试的是应届毕业生或者 C++初级程序员，如果能全面地考虑到前面四点并完整地写出代码，面试官可能会让他通过这轮面试。但如果面试的是 C++的高级程序员，面试官可能会提出更高的要求。

面试官会提醒我们在前面的函数中，显示地用 delete 释放自身 m_pData 的内存。同时我们也会在析构函数中用 delete 释放自身 m_pData 的内存。如果这个类型中添加新的指

针成员变量，那么我们至少需要做两处修改，即同时在析构函数和这个赋值运算符函数里添加一条 delete 语句来释放新指针所指向的内存。一个改动需要在代码中多个地方修改代码，通常是有安全隐患的。通常我们会记得在析构函数里用 delete 释放指针成员变量，但未必每次都记得到赋值运算符函数来添加代码释放内存。

更好的办法在复制运算符函数中利用析构函数自动释放实例已有的内存。下面是这种思路的参考代码：

```
CMyString& CMyString::operator =(const CMyString &str)
{
if(this != &str)
{
CMyString strTemp(str);

char* pTemp = strTemp.m_pData;
strTemp.m_pData = m_pData;
m_pData = pTemp;
}

return *this;
}
```

在这个函数中，我们定义一个临时实例 strTemp，并把 strTemp 的 m_pData 指向当前实例（*this）的 m_pData。由于 strTemp 是个局部变量，但程序员运行到 if 的外面是也就出了的该变量的域，就会自动调用 strTemp 的析构函数，就会把 strTemp.m_pData 所指向的内存释放掉。由于 strTemp.m_pData 指向的内存就是当前实例之前 m_pData 的内存。这就相当于自动调用析构函数释放当前实例的内存。如果新增加指针成员变量，我们只需要在析构函数里正确地释放，而不需要对赋值运算符函数做任何修改。

31. 从尾到头输出链表

题目：输入一个链表的头结点，从尾到头反过来输出每个结点的值。链表结点定义如下：

```
struct ListNode
{
    int m_nKey;
    ListNode* m_pNext;
};
```

分析：这是一道很有意思的面试题。该题以及它的变体经常出现在各大公司的面试、笔试题中。看到这道题后，第一反应是从头到尾输出比较简单。于是很自然地想到把链表中链接结点的指针反转过来，改变链表的方向。然后就可以从头到尾输出了。反转链表的算法详见本人面试题精选系列的第 19 题，在此不再细述。但该方法需要额外的操作，应该还有更好的方法。

接下来的想法是从头到尾遍历链表，每经过一个结点的时候，把该结点放到一个栈中。当遍历完整个链表后，再从栈顶开始输出结点的值，此时输出的结点的顺序已经反转过来了。该方法需要维护一个额外的栈，实现起来比较麻烦。

既然想到了栈来实现这个函数，而递归本质上就是一个栈结构。于是很自然的又想到了

用递归来实现。要实现反过来输出链表，我们每访问到一个结点的时候，先递归输出它后面的结点，再输出该结点自身，这样链表的输出结果就反过来了。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

typedef struct node
{   int data;
    struct node *next;
}LinkList;

void CreateList(LinkList *&L, int arr[], int n)
{   L=(LinkList*)malloc(sizeof(LinkList));
    L->next=NULL;
    if(arr==NULL||n<=0)
      return;
    int i;
    LinkList *p, *r;
    r=L;
    for(i=0; i<n; i++)
    {   p=(LinkList*)malloc(sizeof(LinkList));
        p->data=arr[i];
        r->next=p;
        r=p;
    }
    r->next=NULL;
}

void ReverseDisp(LinkList *L)
{   if(L!=NULL)
    {   if(L->next!=NULL)
            ReverseDisp(L->next);
        cout<<L->data<<" ";
    }
}

int main()
{    int r[4]={1,2,3,4};
     LinkList *L;
     CreateList(L, r, 4);
     ReverseDisp(L->next);   //指向第一个数据结点
```

```
    system("pause");
    return 0;
}
```

32. 不能被继承的类

题目： 用 C++设计一个不能被继承的类。

分析： 这是 Adobe 公司 2007 年校园招聘的最新笔试题。这道题除了考察应聘者的 C++基本功底外，还能考察反应能力，是一道很好的题目。

在 Java 中定义了关键字 final，被 final 修饰的类不能被继承。但在 C++中没有 final 这个关键字，要实现这个要求还是需要花费一些精力。

首先想到的是在 C++ 中，子类的构造函数会自动调用父类的构造函数。同样，子类的析构函数也会自动调用父类的析构函数。要想一个类不能被继承，我们只要把它的构造函数和析构函数都定义为私有函数。那么当一个类试图从它那继承的时候，必然会由于试图调用构造函数、析构函数而导致编译错误。

可是这个类的构造函数和析构函数都是私有函数了，我们怎样才能得到该类的实例呢？这难不倒我们，我们可以通过定义静态来创建和释放类的实例。

基于这个思路，我们可以写出如下的代码：

```
class FinalClass1
{
public:
    static FinalClass1* GetInstance()
    {
        return new FinalClass1;
    }
    static void DeleteInstance( FinalClass1* pInstance)
    {
        delete pInstance;
        pInstance = 0;
    }
private:
    FinalClass1() {}
    ~FinalClass1() {}
};
```

这个类是不能被继承，但在总觉得它和一般的类有些不一样，使用起来也有点不方便。比如，我们只能得到位于堆上的实例，而得不到位于栈上实例。

能不能实现一个和一般类除了不能被继承之外其他用法都一样的类呢？办法总是有的，不过需要一些技巧。请看如下代码：

```
template <typename T> class MakeFinal
{
    friend T;
private:
    MakeFinal() {}
    ~MakeFinal() {}
};
```

```
class FinalClass2 : virtual public MakeFinal<FinalClass2>
{
public:
    FinalClass2() {}
    ~FinalClass2() {}
};
```

这个类使用起来和一般的类没有区别，可以在栈上、也可以在堆上创建实例。尽管类 MakeFinal<FinalClass2>的构造函数和析构函数都是私有的，但由于类 FinalClass2 是它的友元函数，因此在 FinalClass2 中调用 MakeFinal<FinalClass2>的构造函数和析构函数都不会造成编译错误。

但当我们试图从 FinalClass2 继承一个类并创建它的实例时，却不同通过编译。

```
class Try : public FinalClass2
{
public:
    Try() {}
    ~Try() {}
};
Try temp;
```

由于类 FinalClass2 是从类 MakeFinal<FinalClass2>虚继承过来的，在调用 Try 的构造函数的时候，会直接跳过 FinalClass2 而直接调用 MakeFinal<FinalClass2> 的构造函数。非常遗憾的是，Try 不是 MakeFinal<FinalClass2>的友元，因此不能调用其私有的构造函数。

基于上面的分析，试图从 FinalClass2 继承的类，一旦实例化，都会导致编译错误，因此是 FinalClass2 不能被继承。这就满足了我们设计要求。

33. 在 O(1)时间内删除链表结点

题目：给定链表的头指针和一个结点指针，在 O(1)时间删除该结点。链表结点的定义如下：

```
struct ListNode
{
    int m_nKey;
    ListNode* m_pNext;
};
```

函数的声明如下：

```
void DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted);
```

分析：这是一道广为流传的 Google 面试题，能有效考察我们的编程基本功，还能考察我们的反应速度，更重要的是，还能考察我们对时间复杂度的理解。

在链表中删除一个结点，最常规的做法是从链表的头结点开始，顺序查找要删除的结点，找到之后再删除。由于需要顺序查找，时间复杂度自然就是 O(n) 了。

我们之所以需要从头结点开始查找要删除的结点，是因为我们需要得到要删除的结点的前面一个结点。我们试着换一种思路。我们可以从给定的结点得到它的下一个结点。这个时候我们实际删除的是它的下一个结点，由于我们已经得到实际删除的结点的前面一个结点，

因此完全是可以实现的。当然，在删除之前，我们需要需要把给定的结点的下一个结点的数据拷贝到给定的结点中。此时，时间复杂度为 O(1)。

上面的思路还有一个问题：如果删除的结点位于链表的尾部，没有下一个结点，怎么办？我们仍然从链表的头结点开始，顺便遍历得到给定结点的前序结点，并完成删除操作。这个时候时间复杂度是 O(n)。

那题目要求我们需要在 O(1)时间完成删除操作，我们的算法是不是不符合要求？实际上，假设链表总共有 n 个结点，我们的算法在 n-1 总情况下时间复杂度是 O(1)，只有当给定的结点处于链表末尾的时候，时间复杂度为 O(n)。那么平均时间复杂度[(n-1)*O(1)+O(n)]/n，仍然为 O(1)。

值得注意的是，为了让代码看起来简洁一些，上面的代码基于两个假设：（1）给定的结点的确在链表中；（2）给定的要删除的结点不是链表的头结点。不考虑第一个假设对代码的鲁棒性是有影响的。至于第二个假设，当整个列表只有一个结点时，代码会有问题。但这个假设不算很过分，因为在有些链表的实现中，会创建一个虚拟的链表头，并不是一个实际的链表结点。这样要删除的结点就不可能是链表的头结点了。当然，在面试中，我们可以把这些假设和面试官交流。这样，面试官还是会觉得我们考虑问题很周到的。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

typedef struct node
{   int data;
    struct node *next;
}LinkList;

void CreateList(LinkList *&L, int arr[], int n)
{   L=(LinkList*)malloc(sizeof(LinkList));
    L->next=NULL;
    if(arr==NULL||n<=0)
       return;
    int i;
    LinkList *p, *r;
    r=L;
    for(i=0; i<n; i++)
    {   p=(LinkList*)malloc(sizeof(LinkList));
        p->data=arr[i];
        r->next=p;
        r=p;
    }
    r->next=NULL;
}
```

```
void DeleteNode(LinkList *L, LinkList *toBeDeleted)
{   if(L==NULL||toBeDeleted==NULL)
        return;
    if(toBeDeleted->next!=NULL)
    {   LinkList *tmp=toBeDeleted->next;
        toBeDeleted->data=tmp->data;
        toBeDeleted->next=tmp->next;
        free(tmp);
    }
    else
    {   LinkList *tmp=L->next;
        while(tmp!=NULL&&tmp->next!=toBeDeleted)
            tmp=tmp->next;
        tmp->next=NULL;
        free(toBeDeleted);
    }
}

LinkList *FindNode(LinkList *L, int k)
{   LinkList *p=L->next;
    while(p!=NULL&&p->data!=k)
        p=p->next;
    return p;
}

void Disp(LinkList *L)
{   if(L!=NULL)
    {   LinkList *p=L->next;
        while(p!=NULL)
        {   cout<<" "<<p->data;
            p=p->next;
        }
        cout<<endl;
    }
}

int main()
{    int r[4]={1,2,3,4};
     LinkList *L, *p;
     CreateList(L, r, 4);
     p=FindNode(L, 4);
     DeleteNode(L, p);
```

```
    Disp(L);
    system("pause");
    return 0;
}
```

34. 找出数组中两个只出现一次的数字

题目：一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是 O(n)，空间复杂度是 O(1)。

分析：这是一道很新颖的关于位运算的面试题。

首先我们考虑这个问题的一个简单版本：一个数组里除了一个数字之外，其他的数字都出现了两次。请写程序找出这个只出现一次的数字。

这个题目的突破口在哪里？题目为什么要强调有一个数字出现一次，其他的出现两次？我们想到了异或运算的性质：任何一个数字异或它自己都等于 0。也就是说，如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现依次的数字，因为那些出现两次的数字全部在异或中抵消掉了。

有了上面简单问题的解决方案之后，我们回到原始的问题。如果能够把原数组分为两个子数组。在每个子数组中，包含一个只出现一次的数字，而其他数字都出现两次。如果能够这样拆分原数组，按照前面的办法就是分别求出这两个只出现一次的数字了。

我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其他数字都出现了两次，在异或中全部抵消掉了。由于这两个数字肯定不一样，那么这个异或结果肯定不为 0，也就是说在这个结果数字的二进制表示中至少就有一位为 1。我们在结果数字中找到第一个为 1 的位的位置，记为第 N 位。现在我们以第 N 位是不是 1 为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第 N 位都为 1，而第二个子数组的每个数字的第 N 位都为 0。

现在我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其他数字都出现了两次。因此到此为止，所有的问题我们都已经解决。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

int FindFirstOne(int value);
bool TestBit(int value,int pos);
int FindNums(int date[],int length,int &num1,int &num2){
    if(length<2){return 0;}
    int ansXor=0;
    for(int i=0;i<length;i++){
        ansXor^=date[i];                    //异或
    }
    int pos=FindFirstOne(ansXor);
    num1=num2=0;
    for(int i=0;i<length;i++){
        if(TestBit(date[i],pos))
```

```
                num1^=date[i];
            else
                num2^=date[i];
        }
        return 1;
}
int FindFirstOne(int value){        //取二进制中首个为 1 的位置
        int pos=1;
        while((value&1)!=1){
            value=value>>1;
            pos++;
        }
        return pos;
}
bool TestBit(int value,int pos){   //测试某位置是否为 1
        return ((value>>pos)&1);
}
int main(void){
        int date[10]={1,2,3,4,5,6,4,3,2,1};
        int ans1,ans2;
        if(FindNums(date,10,ans1,ans2)==1)
            cout<<ans1<<" "<<ans2<<endl;
        else
            cout<<"error"<<endl;

        system("pause");
        return 0;
}
```

35. 找出两个链表的第一个公共结点

题目：两个单向链表，找出它们的第一个公共结点。

链表的结点定义为：

```
struct ListNode
{
    int  m_nKey;
    ListNode* m_pNext;
};
```

分析：这是一道微软的面试题。微软非常喜欢与链表相关的题目，因此在微软的面试题中，链表出现的概率相当高。

如果两个单向链表有公共的结点，也就是说两个链表从某一结点开始，它们的 m_pNext 都指向同一个结点。但由于是单向链表的结点，每个结点只有一个 m_pNext，因此从第一个公共结点开始，之后它们所有结点都是重合的，不可能再出现分叉。所以，两个有公共结点而部分重合的链表，拓扑形状看起来像一个 Y，而不可能像 X。

看到这个题目，第一反应就是蛮力法：在第一链表上顺序遍历每个结点。每遍历一个结点的时候，在第二个链表上顺序遍历每个结点。如果此时两个链表上的结点是一样的，说明此时两个链表重合，于是找到了它们的公共结点。如果第一个链表的长度为 m，第二个链表的长度为 n，显然，该方法的时间复杂度为 O(mn)。

接下来我们试着去寻找一个线性时间复杂度的算法。我们先把问题简化：如何判断两个单向链表有没有公共结点？前面已经提到，如果两个链表有一个公共结点，那么该公共结点之后的所有结点都是重合的。那么，它们的最后一个结点必然是重合的。因此，我们判断两个链表是不是有重合的部分，只要分别遍历两个链表到最后一个结点。如果两个尾结点是一样的，说明它们用重合；否则两个链表没有公共的结点。

在上面的思路中，顺序遍历两个链表到尾结点的时候，我们不能保证在两个链表上同时到达尾结点。这是因为两个链表不一定长度一样。但如果假设一个链表比另一个长 l 个结点，我们先在长的链表上遍历 l 个结点，之后再同步遍历，这个时候我们就能保证同时到达最后一个结点了。由于两个链表从第一个公共结点考试到链表的尾结点，这一部分是重合的。因此，它们肯定也是同时到达第一公共结点的。于是在遍历中，第一个相同的结点就是第一个公共的结点。

在这个思路中，我们先要分别遍历两个链表得到它们的长度，并求出两个长度之差。在长的链表上先遍历若干次之后，再同步遍历两个链表，知道找到相同的结点，或者一直到链表结束。此时，如果第一个链表的长度为 m，第二个链表的长度为 n，该方法的时间复杂度为 O(m+n)。

源码：

```
int length(node *list){ //求长度
    int len=0;
    while(list){
        len++;
        list=list->next;
    }
    return len;
}
node *find_node(node *&listA,node *&listB){
  int lenA=length(listA),lenB=length(listB);
  node *lA=listA,*lB=listB;
  if(lenA>lenB){
   for(int i=1;i<=lenA-lenB;i++)
    lA=lA->next;
  }else{
   for(int i=1;i<=lenB-lenA;i++)
   lB=lB->next;
  }
while(lA&&lB){
  if(lA==lB){
  return lA;
}else{
  lA=lA->next;
```

```
  lB=lB->next;
 }
}
return NULL;
}
```

36. 删除字符串中指定字符

题目：输入两个字符串，从第一字符串中删除第二个字符串中所有的字符。例如，输入”They are students.”和”aeiou”，则删除之后的第一个字符串变成”Thy r stdnts.”。

分析：这是一道微软面试题。在微软的常见面试题中，与字符串相关的题目占了很大的一部分，因为写程序操作字符串能很好的反映我们的编程基本功。

要编程完成这道题要求的功能可能并不难。毕竟，这道题的基本思路就是在第一个字符串中拿到一个字符，在第二个字符串中查找一下，看它是不是在第二个字符串中。如果在的话，就从第一个字符串中删除。但如何能够把效率优化到让人满意的程度，却也不是一件容易的事情。也就是说，如何在第一个字符串中删除一个字符，以及如何在第二字符串中查找一个字符，都是需要一些小技巧的。

首先我们考虑如何在字符串中删除一个字符。由于字符串的内存分配方式是连续分配的。我们从字符串当中删除一个字符，需要把后面所有的字符往前移动一个字节的位置。但如果每次删除都需要移动字符串后面的字符的话，对于一个长度为 n 的字符串而言，删除一个字符的时间复杂度为 O(n)。而对于本题而言，有可能要删除的字符的个数是 n，因此该方法就删除而言的时间复杂度为 $O(n^2)$。

事实上，我们并不需要在每次删除一个字符的时候都去移动后面所有的字符。我们可以设想，当一个字符需要被删除的时候，我们把它所占的位置让它后面的字符来填补，也就相当于这个字符被删除了。在具体实现中，我们可以定义两个指针(pFast 和 pSlow)，初始的时候都指向第一字符的起始位置。当 pFast 指向的字符是需要删除的字符，则 pFast 直接跳过，指向下一个字符。如果 pFast 指向的字符是不需要删除的字符，那么把 pFast 指向的字符赋值给 pSlow 指向的字符，并且 pFast 和 pStart 同时向后移动指向下一个字符。这样，前面被 pFast 跳过的字符相当于被删除了。用这种方法，整个删除在 O(n)时间内就可以完成。

接下来我们考虑如何在一个字符串中查找一个字符。当然，最简单的办法就是从头到尾扫描整个字符串。显然，这种方法需要一个循环，对于一个长度为 n 的字符串，时间复杂度是 O(n)。

由于字符的总数是有限的。对于八位的 char 型字符而言，总共只有 2^8=256 个字符。我们可以新建一个大小为 256 的数组，把所有元素都初始化为 0。然后对于字符串中每一个字符，把它的 ASCII 码映射成索引，把数组中该索引对应的元素设为 1 。这个时候，要查找一个字符就变得很快了：根据这个字符的 ASCII 码，在数组中对应的下标找到该元素，如果为 0，表示字符串中没有该字符，否则字符串中包含该字符。此时，查找一个字符的时间复杂度是 O(1)。其实，这个数组就是一个 hash 表。这种思路的详细说明，详见本面试题系列的第 13 题。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

#define M 256
```

```
char *DelSpecChars(char *src, char *dst)
{   char *begin=src;
    char *end=src;
    char hashtable[256];
    int i=0;
    memset(hashtable,0,sizeof(hashtable));
    while(*dst)
        ++hashtable[*dst++];
    while(*end)
    {
        if(!hashtable[*end])//del character not detected
        {
            *begin = *end ;
            ++begin;
        }
        ++end;

    }

   *begin= '\0';
   return src;
}

int main(void){
    char src[] = "They are students.";
    char del[] = "aeiou";
    DelSpecChars(src, del);
    printf("Output : %s\n",src);

    system("pause");
    return 0;
}

char *DelSpecChars(char *src, char *dst)
{   char *p=src;
    char hashtable[256];
    int i=0;
    memset(hashtable,0,sizeof(hashtable));
    while(*dst)
        ++hashtable[*dst++];
    int k=0;
```

```
    while(*p!='\0')
    {
        if(!hashtable[*p])//del character not detected
        {
            *(p-k) = *p ;
        }
        else
        {   ++k;
        }
        ++p;

    }
    *(p-k)='\0';

   return src;
}
```

37. 寻找丑数

题目：我们把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。例如 6、8 都是丑数，但 14 不是，因为它包含因子 7。习惯上我们把 1 当做是第一个丑数。求按从小到大的顺序的第 1500 个丑数。

分析：这是一道在网络上广为流传的面试题，据说 google 曾经采用过这道题。所谓一个数 m 是另一个数 n 的因子，是指 n 能被 m 整除，也就是 n % m == 0。根据丑数的定义，丑数只能被 2、3 和 5 整除。也就是说如果一个数如果它能被 2 整除，我们把它连续除以 2；如果能被 3 整除，就连续除以 3；如果能被 5 整除，就除以连续 5。如果最后我们得到的是 1，那么这个数就是丑数，否则不是。

我们只需要在函数 GetUglyNumber_Solution1 中传入参数 1500，就能得到第 1500 个丑数。该算法非常直观，代码也非常简洁，但最大的问题我们每个整数都需要计算。即使一个数字不是丑数，我们还是需要对它做求余数和除法操作。因此该算法的时间效率不是很高。

接下来我们换一种思路来分析这个问题，试图只计算丑数，而不在非丑数的整数上花费时间。根据丑数的定义，丑数应该是另一个丑数乘以 2、3 或者 5 的结果（1 除外）。因此我们可以创建一个数组，里面的数字是排好序的丑数。里面的每一个丑数是前面的丑数乘以 2、3 或者 5 得到的。

这种思路的关键在于怎样确保数组里面的丑数是排好序的。我们假设数组中已经有若干个丑数，排好序后存在数组中。我们把现有的最大丑数记做 M。现在我们来生成下一个丑数，该丑数肯定是前面某一个丑数乘以 2、3 或者 5 的结果。我们首先考虑把已有的每个丑数乘以 2。在乘以 2 的时候，能得到若干个结果小于或等于 M 的。由于我们是按照顺序生成的，小于或者等于 M 肯定已经在数组中了，我们不需再次考虑；我们还会得到若干个大于 M 的结果，但我们只需要第一个大于 M 的结果，因为我们希望丑数是按从小到大顺序生成的，其他更大

的结果我们以后再说。我们把得到的第一个乘以 2 后大于 M 的结果，记为 M_2。同样我们把已有的每一个丑数乘以 3 和 5，能得到第一个大于 M 的结果 M_3 和 M_5。那么下一个丑数应该是 M_2、M_3 和 M_5 三个数的最小者。

前面我们分析的时候，提到把已有的每个丑数分别都乘以 2、3 和 5，事实上是不需要的，因为已有的丑数是按顺序存在数组中的。对乘以 2 而言，肯定存在某一个丑数 T_2，排在它之前的每一个丑数乘以 2 得到的结果都会小于已有最大的丑数，在它之后的每一个丑数乘以 2 得到的结果都会太大。我们只需要记下这个丑数的位置，同时每次生成新的丑数的时候，去更新这个 T_2。对乘以 3 和 5 而言，存在着同样的 T_3 和 T_5。

和第一种思路相比，这种算法不需要在非丑数的整数上做任何计算，因此时间复杂度要低很多。感兴趣的读者可以分别统计两个函数 GetUglyNumber_Solution1(1500)和 GetUglyNumber_Solution2(1500)的运行时间。当然我们也要指出，第二种算法由于要保存已经生成的丑数，因此需要一个数组，从而需要额外的内存。第一种算法是没有这样的内存开销的。

源码：

```
#include <iostream>
#include <stack>
using namespace std;

int Min(int number1, int number2, int number3)
{
    int min = (number1 < number2) ? number1 : number2;
    min = (min < number3) ? min : number3;
    return min;
}

int GetUglyNumber(int index)
{  if(index <= 0)
        return 0;
    int *pUglyNumbers = new int[index];
    pUglyNumbers[0] = 1;
    int nextUglyIndex = 1;
    int *pMultiply2 = pUglyNumbers;
    int *pMultiply3 = pUglyNumbers;
    int *pMultiply5 = pUglyNumbers;

    while(nextUglyIndex < index)
    {
        int min = Min(*pMultiply2 * 2, *pMultiply3 * 3, *pMultiply5 * 5);
        pUglyNumbers[nextUglyIndex] = min;
        while(*pMultiply2 * 2 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply2;
```

```
        while(*pMultiply3 * 3 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply3;
        while(*pMultiply5 * 5 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply5;
        ++nextUglyIndex;
    }
    int ugly = pUglyNumbers[nextUglyIndex - 1];
    delete[] pUglyNumbers;
    return ugly;
}

int main(void){
    int data=GetUglyNumber(150);
    cout<<data<<endl;
    system("pause");
    return 0;
}
```

38. 输出 1 到最大 N 的 N 位数

题目：输入数字 n，按顺序输出从 1 最大的 n 位 10 进制数。比如输入 3，则输出 1、2、3 一直到最大的 3 位数即 999。

分析：这是一道很有意思的题目。看起来很简单，其实里面却有不少的玄机。应聘者在解决这个问题的时候，最容易想到的方法是先求出最大的 n 位数是什么，然后用一个循环从 1 开始逐个输出。

初看之下，好像没有问题。但如果我们仔细分析这个问题，就能注意到这里没有规定 n 的范围，当我们求最大的 n 位数的时候，是不是有可能用整型甚至长整型都会溢出？

分析到这里，我们很自然的就想到我们需要表达一个大数。最常用的也是最容易实现的表达大数的方法是用字符串或者整型数组（当然不一定是最有效的）。用字符串表达数字的时候，最直观的方法就是字符串里每个字符都是’0’到’9’之间的某一个字符，表示数字中的某一位。因为数字最大是 n 位的，因此我们需要一个 n+1 位字符串（最后一位为结束符号’\0’）。当实际数字不够 n 位的时候，在字符串的前半部分补零。这样，数字的个位永远都在字符串的末尾（除去结尾符号）。

首先我们把字符串中每一位数字都初始化为’0’。然后每一次对字符串表达的数字加上 1，再输出。因此我们只需要做两件事：一是在字符串表达的数字上模拟加法。另外我们要把字符串表达的数字输出。值得注意的是，当数字不够 n 位的时候，我们在数字的前面补零。输出的时候这些补位的 0 不应该输出。比如输入 3 的时候，那么数字 98 以 098 的形式输出，就不符合我们的习惯了。

第二种思路基本上和第一种思路相对应，只是把一个整型数值换成了字符串的表示形式。同时，值得提出的是，判断打印是否应该结束时，我没有调用函数 strcmp 比较字符串 number 和”99…999”（n 个 9）。这是因为 strcmp 的时间复杂度是 O(n)，而判断是否溢出的平均时间复杂度是 O(1)。

第二种思路虽然比较直观，但由于模拟了整数的加法，代码有点长。要在面试短短几十分钟时间里完整正确写出这么长代码，不是件容易的事情。接下来我们换一种思路来考虑这

个问题。如果我们在数字前面补 0 的话，就会发现 n 位所有 10 进制数其实就是 n 个从 0 到 9 的全排列。也就是说，我们把数字的每一位都从 0 到 9 排列一遍，就得到了所有的 10 进制数。只是我们在输出的时候，数字排在前面的 0 我们不输出罢了。

全排列用递归很容易表达，数字的每一位都可能是 0 到 9 中的一个数，然后设置下一位。递归结束的条件是我们已经设置了数字的最后一位。。对比这两种思路，我们可以发现，递归能够用很简洁的代码来解决问题。

源码：

39. 颠倒栈

题目：用递归颠倒一个栈。例如输入栈{1, 2, 3, 4, 5}，1 在栈顶。颠倒之后的栈为{5, 4, 3, 2, 1}，5 处在栈顶。

分析：乍一看到这道题目，第一反应是把栈里的所有元素逐一 pop 出来，放到一个数组里，然后在数组里颠倒所有元素，最后把数组中的所有元素逐一 push 进入栈。这时栈也就颠倒过来了。颠倒一个数组是一件很容易的事情。不过这种思路需要显示分配一个长度为 O(n)的数组，而且也没有充分利用递归的特性。

我们再来考虑怎么递归。我们把栈{1, 2, 3, 4, 5}看成由两部分组成：栈顶元素 1 和剩下的部分{2, 3, 4, 5}。如果我们能把{2, 3, 4, 5}颠倒过来，变成{5, 4, 3, 2}，然后在把原来的栈顶元素 1 放到底部，那么就整个栈就颠倒过来了，变成{5, 4, 3, 2, 1}。

接下来我们需要考虑两件事情：一是如何把{2, 3, 4, 5}颠倒过来变成{5, 4, 3, 2}。我们只要把{2, 3, 4, 5}看成由两部分组成：栈顶元素 2 和剩下的部分{3, 4, 5}。我们只要把{3, 4, 5}先颠倒过来变成{5, 4, 3}，然后再把之前的栈顶元素 2 放到最底部，也就变成了{5, 4, 3, 2}。

至于怎么把{3, 4, 5}颠倒过来……很多读者可能都想到这就是递归。也就是每一次试图颠倒一个栈的时候，现在栈顶元素 pop 出来，再颠倒剩下的元素组成的栈，最后把之前的栈顶元素放到剩下元素组成的栈的底部。递归结束的条件是剩下的栈已经空了。

我们需要考虑的另外一件事情是如何把一个元素 *e* 放到一个栈的底部，也就是如何实现 `AddToStackBottom`。这件事情不难，只需要把栈里原有的元素逐一 pop 出来。当栈为空的时候，push 元素 *e* 进栈，此时它就位于栈的底部了。然后再把栈里原有的元素按照 pop 相反的顺序逐一 push 进栈。

注意到我们在 push 元素 *e* 之前，我们已经把栈里原有的所有元素都 pop 出来了，我们需要把它们保存起来，以便之后能把他们再 push 回去。我们当然可以开辟一个数组来做，但这没有必要。由于我们可以用递归来做这件事情，而递归本身就是一个栈结构。我们可以用递归的栈来保存这些元素。

源码：

40. 扑克牌的顺子

题目：从扑克牌中随机抽 5 张牌，判断是不是一个顺子，即这 5 张牌是不是连续的。2-10 为数字本身，A 为 1，J 为 11，Q 为 12，K 为 13，而大小王可以看成任意数字。

分析：这题目很有意思，是一个典型的寓教于乐的题目。

我们需要把扑克牌的背景抽象成计算机语言。不难想象，我们可以把 5 张牌看成由 5 个数字组成的数组。大小王是特殊的数字，我们不妨把它们都当成 0，这样和其他扑克牌代表的数字就不重复了。

接下来我们来分析怎样判断 5 个数字是不是连续的。最直观的是，我们把数组排序。但值得注意的是，由于 0 可以当成任意数字，我们可以用 0 去补满数组中的空缺。也就是排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但如果我们有足够的 0 可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0，1，3，4，5}。在 1 和 3 之间空缺了一个 2，刚好我们有一个 0，也就是我们可以它当成 2 去填补这个空缺。

于是我们需要做三件事情：把数组排序，统计数组中 0 的个数，统计排序之后的数组相邻数字之间的空缺总数。如果空缺的总数小于或者等于 0 的个数，那么这个数组就是连续的；反之则不连续。最后，我们还需要注意的是，如果数组中的非 0 数字重复出现，则该数组不是连续的。换成扑克牌的描述方式，就是如果一副牌里含有对子，则不可能是顺子。

本文为了让代码显得比较简洁，上述代码用 C++的标准模板库中的 vector 来表达数组，同时用函数 sort 排序。当然我们可以自己写排序算法。为了有更好的通用性，上述代码没有限定数组的长度和允许出现的最大数字。要解答原题，我们只需要确保传入的数组的长度是 5，并且 maxNumber 为 13 即可。

源码：

41. 把数组排成最小的数

题目：输入一个正整数数组，将它们连接起来排成一个数，输出能排出的所有数字中最小的一个。例如输入数组{32, 321}，则输出这两个能排成的最小数字 32132。请给出解决问题的算法，并证明该算法。

分析：这是 09 年 6 月份百度新鲜出炉的一道面试题，从这道题我们可以看出百度对应聘者在算法方面有很高的要求。

这道题其实是希望我们能找到一个排序规则，根据这个规则排出来的数组能排成一个最小的数字。要确定排序规则，就得比较两个数字，也就是给出两个数字 m 和 n，我们需要确定一个规则 m 和 n 哪个更大，而不是仅仅只是比较这两个数字的数值哪个更大。

根据题目的要求，两个数字 m 和 n 排成的数字 mn 和 nm，如果 mn<nm，那么我们应该输出 mn，也就是 m 应该排在 n 的前面，也就是 m 小于 n；反之，如果 nm<mn，n 小于 m。如果 mn==mn，m 等于 n。

接下来我们考虑怎么去拼接数字，即给出数字 m 和 n，怎么得到数字 mn 和 nm 并比较它们的大小。直接用数值去计算不难办到，但需要考虑到的一个潜在问题是 m 和 n 都在 int 能表达的范围内，但把它们拼起来的数字 mn 和 nm 就不一定能用 int 表示了。所以我们需要解决大数问题。一个非常直观的方法就是把数字转换成字符串。

另外，由于把数字 m 和 n 拼接起来得到的 mn 和 nm，它们所含有的数字的个数肯定是相同的。因此比较它们的大小只需要按照字符串大小的比较规则就可以了。

上述代码中，我们在函数 compare 中定义比较规则，并根据该规则用库函数 qsort 排序。最后把排好序的数组输出，就得到了根据数组排成的最小的数字。

找到一个算法解决这个问题，不是一件容易的事情。但更困难的是我们需要证明这个算法是正确的。接下来我们来试着证明。

首先我们需要证明之前定义的比较两个数字大小的规则是有效的。一个有效的比较需要三个条件：1.自反性，即 a 等于 a；2.对称性，即如果 a 大于 b，则 b 小于 a；3.传递性，即如果 a 小于 b，b 小于 c，则 a 小于 c。现在分别予以证明。

1. 自反性。显然有 $aa=aa$，所以 $a=a$。
2. 对称性。如果 a 小于 b，则 $ab<ba$，所以 $ba>ab$。因此 b 大于 a。
3. 传递性。如果 a 小于 b，则 $ab<ba$。当 a 和 b 用十进制表示的时候分别为 l 位和 m 位时，$ab=a\times10^{m}+b$，$ba=b\times10^{l}+a$。所以 $a\times10^{m}+b<b\times10^{l}+a$。于是有 $a\times10^{m}-a< b\times10^{l}-b$，即 $a(10^{m}-1)<b(10^{l}-1)$。所以 $a/(10^{l}-1)<b/(10^{m}-1)$。

   如果 b 小于 c，则 $bc<cb$。当 c 表示成十进制时为 m 位。和前面证明过程一样，可以得到 $b/(10^{m}-1)<c/(10^{n}-1)$。

   所以 $a/(10^{l}-1)< c/(10^{n}-1)$。于是 $a(10^{n}-1)<c(10^{l}-1)$，所以 $a\times10^{n}+c<c\times10^{l}+a$，即 $ac<ca$。所以 a 小于 c。

在证明了我们排序规则的有效性之后，我们接着证明算法的正确性。我们用反证法来证明。

我们把 n 个数按照前面的排序规则排好顺序之后，表示为 $A_1A_2A_3\cdots A_n$。我们假设这样排出来的两个数并不是最小的。即至少存在两个 x 和 y（$0<x<y<n$），交换第 x 个数和地 y 个数后，$A_1A_2\cdots A_y\cdots A_x\cdots A_n<A_1A_2\cdots A_x\cdots A_y\cdots A_n$。

由于 $A_1A_2\cdots A_x\cdots A_y\cdots A_n$ 是按照前面的规则排好的序列，所以有 $A_x<A_{x+1}<A_{x+2}<\cdots<A_{y-2}<A_{y-1}<A_y$。

由于 A_{y-1} 小于 A_y，所以 $A_{y-1}A_y<A_yA_{y-1}$。我们在序列 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n$ 交换 A_{y-1} 和 A_y，有 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n<A_1A_2\cdots A_x\cdots A_yA_{y-1}\cdots A_n$（这个实际上也需要证明。感兴趣的读者可以自己试着证明）。我们就这样一直把 A_y 和前面的数字交换，直到和 A_x 交换为止。于是就有 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n<A_1A_2\cdots A_x\cdots A_yA_{y-1}\cdots A_n< A_1A_2\cdots A_x\cdots A_yA_{y-2}A_{y-1}\cdots A_n<\cdots< A_1A_2\cdots A_yA_x\cdots A_{y-2}A_{y-1}\cdots A_n$。

同理由于 A_x 小于 A_{x+1}，所以 $A_xA_{x+1}<A_{x+1}A_x$。我们在序列 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n$ 仅仅只交换 A_x 和 A_{x+1}，有 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n<A_1A_2\cdots A_yA_{x+1}A_x\cdots A_{y-2}A_{y-1}\cdots A_n$。我们接下来一直拿 A_x 和它后面的数字交换，直到和 A_{y-1} 交换为止。于是就有 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n<A_1A_2\cdots A_yA_{x+1}A_x\cdots A_{y-2}A_{y-1}\cdots A_n<\cdots< A_1A_2\cdots A_yA_{x+1}A_{x+2}\cdots A_{y-2}A_{y-1}A_x\cdots A_n$。

所以 $A_1A_2\cdots A_x\cdots A_y\cdots A_n< A_1A_2\cdots A_y\cdots A_x\cdots A_n$。这和我们的假设的 $A_1A_2\cdots A_y\cdots A_x\cdots A_n <A_1A_2\cdots A_x\cdots A_y\cdots A_n$ 相矛盾。

所以假设不成立，我们的算法是正确的。

源码：

42. 旋转数组的最小元素

题目：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个排好序的数组的一个旋转，输出旋转数组的最小元素。例如数组{3, 4, 5, 1, 2}为{1, 2, 3, 4, 5}的一个旋转，该数组的最小值为 1。

分析：这道题最直观的解法并不难。从头到尾遍历数组一次，就能找出最小的元素，时间复杂度显然是 O(*N*)。但这个思路没有利用输入数组的特性，我们应该能找到更好的解法。

我们注意到旋转之后的数组实际上可以划分为两个排序的子数组，而且前面的子数组的元素都大于或者等于后面子数组的元素。我们还可以注意到最小的元素刚好是这两个子数组的分界线。我们试着用二元查找法的思路在寻找这个最小的元素。

首先我们用两个指针，分别指向数组的第一个元素和最后一个元素。按照题目旋转的规则，第一个元素应该是大于或者等于最后一个元素的（这其实不完全对，还有特例。后面再讨论特例）。

接着我们得到处在数组中间的元素。如果该中间元素位于前面的递增子数组，那么它应该大于或者等于第一个指针指向的元素。此时数组中最小的元素应该位于该中间元素的后面。我们可以把第一指针指向该中间元素，这样可以缩小寻找的范围。同样，如果中间元素位于后面的递增子数组，那么它应该小于或者等于第二个指针指向的元素。此时该数组中最小的元素应该位于该中间元素的前面。我们可以把第二个指针指向该中间元素，这样同样可以缩小寻找的范围。我们接着再用更新之后的两个指针，去得到和比较新的中间元素，循环下去。

按照上述的思路，我们的第一个指针总是指向前面递增数组的元素，而第二个指针总是指向后面递增数组的元素。最后第一个指针将指向前面子数组的最后一个元素，而第二个指针会指向后面子数组的第一个元素。也就是它们最终会指向两个相邻的元素，而第二个指针指向的刚好是最小的元素。这就是循环结束的条件。

由于我们每次都把寻找的范围缩小一半，该算法的时间复杂度是 O(log*N*)

值得注意的是，如果在面试现场写代码，通常我们需要用一些测试用例来验证代码是不是正确的，我们在验证的时候尽量能考虑全面些。像这道题，我们出来最简单测试用例之外，我们至少还需要考虑如下的情况：

1. 把数组前面的 0 个元素从最前面搬到最后面，也就是原数组不做改动，根据题目的规则这也是一个旋转，此时数组的第一个元素是大于小于或者等于最后一个元素的；
2. 排好序的数组中有可能有相等的元素，我们特别需要注意两种情况。一是旋转之后的数组中，第一个元素和最后一个元素是相等的；另外一种情况是最小的元素在数组中重复出现
3. 在前面的代码中，如果输入的数组不是一个排序数组的旋转，那将陷入死循环。因此我们需要跟面试官讨论是不是需要判断数组的有效性。在面试的时候，面试官讨论如何验证输入的有效性，能显示我们思维的严密性。本文假设在调用函数 Min 之前，已经验证过输入的有效性了。

最后需要指出的是，如果输入的数组指针是非法指针，我们是用异常来做错误处理。这是因为在这种情况下，如果我们用 return 来结束该函数，返回任何数字都不是正确的。