**程序员面试题精选100题   [折叠]**

**前言**

　　随着高校的持续扩张，每年应届毕业生的数目都在不断增长，伴随而来的是应届毕业生的就业压力也越来越大。

　　在这样的背景下，就业变成一个买方市场的趋势越来越明显。为了找到一个称心的工作，绝大多数应届毕业生都必须反复经历简历筛选、电话面试、笔试、面试等环节。在这些环节中，面试无疑起到最为重要的作用，因为通过面试公司能够最直观的了解学生的能力。

　　为了有效地准备面试，面经这个新兴概念应运而生。笔者在当初找工作阶段也从面经中获益匪浅并最终找到满意的工作。为了方便后来者，笔者花费大量时间收集并整理散落在茫茫网络中的面经。不同行业的面经全然不同，笔者从自身专业出发，着重关注程序员面试的面经，并从精选出若干具有代表性的技术类的面试题展开讨论，希望能给读者带来一些启发。

　　由于笔者水平有限，给各面试题提供的思路和代码难免会有错误，还请读者批评指正。另外，热忱欢迎读者能够提供更多、更好的面试题，本人将感激不尽。

---------------------------------------------------------------------------------------------------------------------------------------------

**(01)把二元查找树转变成排序的双向链表**

　　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  
                                            10  
                                          /    \  
                                        6       14  
                                      /  \     /　 \  
                                   4     8  12 　  16  
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。下面我们用两种不同的递归思路来分析。

　　思路一：当我们到达某一结点准备调整以该结点为根结点的子树时，先调整其左子树将左子树转换成一个排好序的左子链表，再调整其右子树转换右子链表。最近链接左子链表的最右结点（左子树的最大结点）、当前结点和右子链表的最左结点（右子树的最小结点）。从树的根结点开始递归调整所有结点。

　　思路二：我们可以中序遍历整棵树。按照这个方式遍历树，比较小的结点先访问。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双向链表，我们再把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。

参考代码：

首先我们定义二元查找树结点的数据结构如下：  
    struct BSTreeNode // a node in the binary search tree  
    {  
     int          m\_nValue; // value of node：当前节点的值  
    BSTreeNode  \*m\_pLeft;  // left child of node：指向左子节点的指针  
    BSTreeNode  \*m\_pRight; // right child of node：指向右子节点的指针  
    };

思路一对应的代码：当我们到达某一结点准备调整以该结点为根结点的子树时，先调整其左子树将左子树转换成一个排好序的左子链表，再调整其右子树转换右子链表。最近链接左子链表的最右结点（左子树的最大结点）、当前结点和右子链表的最左结点（右子树的最小结点）。从树的根结点开始递归调整所有结点。  
///////////////////////////////////////////////////////////////////////  
// Covert a sub binary-search-tree into a sorted double-linked list  
// Input: pNode - the head of the sub tree  
//        asRight - whether pNode is the right child of its parent  
// Output: if asRight is true, return the least node in the sub-tree  如果asRight为true，则返回子树中的最小节点  
//         else return the greatest node in the sub-tree                    否则，返回子树中最大的节点  
///////////////////////////////////////////////////////////////////////  
BSTreeNode\* ConvertNode(BSTreeNode\* pNode, bool asRight)  
{  
    if(!pNode)  
       return NULL;  
  
**//定义左子树和右子树的指针**  
    BSTreeNode \*pLeft = NULL;  
    BSTreeNode \*pRight = NULL;  
  
    // Convert the left sub-tree，如果当前节点pNode的左节点m\_pLeft不为空，则递归左子树，并false返回左子树中最小值  
    if(pNode->m\_pLeft)  
       pLeft = ConvertNode(pNode->m\_pLeft, false);  
  
    // Connect the greatest node in the left sub-tree to the current node将左子树当中的最大节点赋值给当前节点进行调整  
    if(pLeft)  
    {  
       pLeft->m\_pRight = pNode;  
       pNode->m\_pLeft = pLeft;  
    }  
  
    // Convert the right sub-tree，如果当前节点pNode的右节点m\_pRight不为空，则递归右子树，并true返回右子树中的最小值  
    if(pNode->m\_pRight)  
       pRight = ConvertNode(pNode->m\_pRight, true);  
  
    // Connect the least node in the right sub-tree to the current node将右子树当中的最小节点赋值给当前节点进行调整  
    if(pRight)  
    {  
       pNode->m\_pRight = pRight;  
       pRight->m\_pLeft = pNode;  
    }  
  
    BSTreeNode \*pTemp = pNode;  
  
    // If the current node is the right child of its parent, 如果当前节点的右孩子是它的父亲节点  
    // return the least node in the tree whose root is the current node。则返回当前节点树中的最小节点  
    if(asRight)  
    {  
       while(pTemp->m\_pLeft)  
           pTemp = pTemp->m\_pLeft;  
    }  
    // If the current node is the left child of its parent, 如果当前节点的左孩子是它的父亲节点  
    // return the greatest node in the tree whose root is the current node。则返回当前节点的最大节点  
    else  
    {  
       while(pTemp->m\_pRight)  
           pTemp = pTemp->m\_pRight;  
    }  
   
    return pTemp;  
}  
  
///////////////////////////////////////////////////////////////////////  
// Covert a binary search tree into a sorted double-linked list  
// Input: the head of tree  
// Output: the head of sorted double-linked list  
///////////////////////////////////////////////////////////////////////  
BSTreeNode\* Convert(BSTreeNode\* pHeadOfTree)  
{  
    // As we want to return the head of the sorted double-linked list,  
    // we set the second parameter to be true。初始调用点  
    return ConvertNode(pHeadOfTree, true);  
}

思路二对应的代码：**我们可以中序遍历整棵树。按照这个方式遍历树，比较小的结点先访问。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双向链表，我们再把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。**  
///////////////////////////////////////////////////////////////////////  
// Covert a sub binary-search-tree into a sorted double-linked list  
// Input: pNode -           the head of the sub tree  
//        pLastNodeInList - the tail of the double-linked list  
///////////////////////////////////////////////////////////////////////  
void ConvertNode(BSTreeNode\* pNode, BSTreeNode\*& pLastNodeInList)  
{  
    if(pNode == NULL)  
       return;  
  
    BSTreeNode \*pCurrent = pNode;  
  
    // Convert the left sub-tree，如果当前节点的左子树不为空，则递归调用左子树  
    if (pCurrent->m\_pLeft != NULL)  
       ConvertNode(pCurrent->m\_pLeft, pLastNodeInList);  
  
    // Put the current node into the double-linked list，将当前节点赋值给pLastNodeInList链表  
    pCurrent->m\_pLeft = pLastNodeInList;   
    if(pLastNodeInList != NULL)  
       pLastNodeInList->m\_pRight = pCurrent;  
  
    pLastNodeInList = pCurrent;  
  
    // Convert the right sub-tree，如果右子树不为空，则递归调用右子树  
    if (pCurrent->m\_pRight != NULL)  
       ConvertNode(pCurrent->m\_pRight, pLastNodeInList);  
}  
  
///////////////////////////////////////////////////////////////////////  
// Covert a binary search tree into a sorted double-linked list，按中序遍历方式的调用情况  
// Input: pHeadOfTree - the head of tree  
// Output: the head of sorted double-linked list  
///////////////////////////////////////////////////////////////////////  
BSTreeNode\* Convert\_Solution1(BSTreeNode\* pHeadOfTree)  
{  
    BSTreeNode \*pLastNodeInList = NULL;  
    ConvertNode(pHeadOfTree, pLastNodeInList);  
  
    // Get the head of the double-linked list  
    BSTreeNode \*pHeadOfList = pLastNodeInList;  
    while(pHeadOfList && pHeadOfList->m\_pLeft)  
       pHeadOfList = pHeadOfList->m\_pLeft;  
  
    return pHeadOfList;

}

-----------------------------------------------------------------------------------------------------------

**(02)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。

我看到这道题目时，第一反应就是每次push一个新元素时，将栈里所有逆序元素排序。这样栈顶元素将是最小元素。但由于不能保证最后push进栈的元素最先出栈，这种思路设计的数据结构已经不是一个栈了。

在栈里添加一个成员变量存放最小元素（或最小元素的位置）。每次push一个新元素进栈的时候，如果该元素比当前的最小元素还要小，则更新最小元素。

乍一看这样思路挺好的。但仔细一想，该思路存在一个重要的问题：如果当前最小元素被pop出去，如何才能得到下一个最小元素？

**因此仅仅只添加一个成员变量存放最小元素（或最小元素的位置）是不够的。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；每次pop一个元素出栈的时候，同时pop辅助栈。**

参考代码：

#include <deque>  
#include <assert.h>  
  
template <typename T> class CStackWithMin  
{  
public:  
    CStackWithMin(void) {}  
    virtual ~CStackWithMin(void) {}  
  
    T& top(void);  
    const T& top(void) const;  
  
    void push(const T& value);  
    void pop(void);  
  
    const T& min(void) const;  
  
private:  
    T>m\_data;// theelements of stack  
    size\_t>m\_minIndex;// the indicesof minimum elements  
};  
  
// get the last element of mutable stack  
template <typename T> T& CStackWithMin<T>::top()  
{  
    return m\_data.back();  
}  
  
// get the last element of non-mutable stack  
template <typename T> const T& CStackWithMin<T>::top() const  
{  
    return m\_data.back();  
}  
  
// insert an elment at the end of stack  
template <typename T> void CStackWithMin<T>::push(const T& value)  
{  
    // append the data into the end of m\_data  
    m\_data.push\_back(value);  
  
    // set the index of minimum elment in m\_data at the end of m\_minIndex  
    if(m\_minIndex.size() == 0)  
       m\_minIndex.push\_back(0);  
    else  
    {  
       if(value < m\_data[m\_minIndex.back()])  
           m\_minIndex.push\_back(m\_data.size() - 1);  
       else  
           m\_minIndex.push\_back(m\_minIndex.back());  
    }  
}  
  
// erease the element at the end of stack  
template <typename T> void CStackWithMin<T>::pop()  
{  
    // pop m\_data  
    m\_data.pop\_back();  
  
    // pop m\_minIndex  
    m\_minIndex.pop\_back();  
}  
  
// get the minimum element of stack  
template <typename T> const T& CStackWithMin<T>::min() const  
{  
    assert(m\_data.size() > 0);  
    assert(m\_minIndex.size() > 0);  
  
    return m\_data[m\_minIndex.back()];  
}

举个例子演示上述代码的运行过程：

  步骤              数据栈            辅助栈                最小值  
1.push 3    3          0             3  
2.push 4    3,4        0,0           3  
3.push 2    3,4,2      0,0,2         2  
4.push 1    3,4,2,1    0,0,2,3       1  
5.pop       3,4,2      0,0,2         2  
6.pop       3,4        0,0           3  
7.push 0    3,4,0      0,0,2         0

讨论：如果思路正确，编写上述代码不是一件很难的事情。但如果能注意一些细节无疑能在面试中加分。比如我在上面的代码中做了如下的工作：

         用模板类实现。如果别人的元素类型只是int类型，模板将能给面试官带来好印象；

         两个版本的top函数。在很多类中，都需要提供const和非const版本的成员访问函数；

         min函数中assert。把代码写的尽量安全是每个软件公司对程序员的要求；

         添加一些注释。注释既能提高代码的可读性，又能增加代码量，何乐而不为？

总之，在面试时如果时间允许，尽量把代码写的漂亮一些。说不定代码中的几个小亮点就能让自己轻松拿到心仪的Offer。

------------------------------------------------------------------------------------------------------------

**(03)－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(n2)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(n3)。

**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。**

参考代码：

/////////////////////////////////////////////////////////////////////////////  
// Find the greatest sum of all sub-arrays  
// Return value: if the input is valid, return true, otherwise return false  
/////////////////////////////////////////////////////////////////////////////  
bool FindGreatestSumOfSubArray  
(  
    int \*pData,           // an array  
    unsigned int nLength, // the length of array  
    int &nGreatestSum     // the greatest sum of all sub-arrays  
)  
{  
    // if the input is invalid, return false  
    if((pData == NULL) || (nLength == 0))  
       return false;  
  
    int nCurSum = nGreatestSum = 0;  
    for(unsigned int i = 0; i < nLength; ++i)  
    {  
       nCurSum += pData[i];  
  
       // if the current sum is negative, discard it  
       if(nCurSum < 0)  
           nCurSum = 0;  
  
       // if a greater sum is found, update the greatest sum  
       if(nCurSum > nGreatestSum)  
           nGreatestSum = nCurSum;

    }

    // if all data are negative, find the greatest element in the array  
    if(nGreatestSum == 0)  
    {  
       nGreatestSum = pData[0];  
       for(unsigned int i = 1; i < nLength; ++i)  
       {  
           if(pData[i] > nGreatestSum)  
              nGreatestSum = pData[i];  
       }  
    }  
  
    return true;  
}

讨论：上述代码中有两点值得和大家讨论一下：

         函数的返回值不是子数组和的最大值，而是一个判断输入是否有效的标志。如果函数返回值的是子数组和的最大值，那么当输入一个空指针是应该返回什么呢？返回0？那这个函数的用户怎么区分输入无效和子数组和的最大值刚好是0这两中情况呢？基于这个考虑，本人认为把子数组和的最大值以引用的方式放到参数列表中，同时让函数返回一个函数是否正常执行的标志。

         输入有一类特殊情况需要特殊处理。当输入数组中所有整数都是负数时，子数组和的最大值就是数组中的最大元素。

**--------------------------------------------------------------------------------------------------------------------------------**

**(04)－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

                                            10  
                                           /   \  
                                          5     12  
                                        /   \     
                                     4      7

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。因此我们在函数退出之前要在路径上删除当前结点并减去当前结点的值，以确保返回父结点时路径刚好是根结点到父结点的路径。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。**

参考代码：

///////////////////////////////////////////////////////////////////////  
// Find paths whose sum equal to expected sum  
///////////////////////////////////////////////////////////////////////  
void FindPath  
(  
    BinaryTreeNode\*   pTreeNode,    // a node of binary tree  
    int               expectedSum,  // the expected sum  
    std::vector<int>&path,        // a pathfrom root to current node  
    int&              currentSum    // the sum of path  
)  
{  
    if(!pTreeNode)  
       return;  
  
    currentSum += pTreeNode->m\_nValue;  
    path.push\_back(pTreeNode->m\_nValue);  
  
    // if the node is a leaf, and the sum is same as pre-defined,   
    // the path is what we want. print the path  
    bool isLeaf = (!pTreeNode->m\_pLeft && !pTreeNode->m\_pRight);  
    if(currentSum == expectedSum && isLeaf)  
    {        
std::vector<int>::iterator iter =path.begin();  
for(; iter != path.end(); ++ iter)  
           std::cout<<\*iter<<'\t';  
       std::cout<<std::endl;  
    }  
  
    // if the node is not a leaf, goto its children  
    if(pTreeNode->m\_pLeft)  
       FindPath(pTreeNode->m\_pLeft, expectedSum, path, currentSum);  
    if(pTreeNode->m\_pRight)  
       FindPath(pTreeNode->m\_pRight, expectedSum, path, currentSum);  
  
    // when we finish visiting a node and return to its parent node,  
    // we should delete this node from the path and   
    // minus the node's value from the current sum  
    currentSum -= pTreeNode->m\_nValue; //!!I think here is no use  
    path.pop\_back();  
}

---------------------------------------------------------------------------------------------------------------------------

**(05)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。

**我们可以开辟一个长度为k的数组。每次从输入的n个整数中读入一个数。如果数组中已经插入的元素少于k个，则将读入的整数直接放到数组中。否则长度为k的数组已经满了，不能再往数组里插入元素，只能替换了。如果读入的这个整数比数组中已有k个整数的最大值要小，则用读入的这个整数替换这个最大值；如果读入的整数比数组中已有k个整数的最大值还要大，则读入的这个整数不可能是最小的k个整数之一，抛弃这个整数。这种思路相当于只要排序k个整数，因此时间复杂可以降到O(n+n*log*k)。通常情况下k要远小于n，所以这种办法要优于前面的思路。**

这是我能够想出来的最快的解决方案。不过从给面试官留下更好印象的角度出发，我们可以进一步把代码写得更漂亮一些。从上面的分析，当长度为k的数组已经满了之后，如果需要替换，每次替换的都是数组中的最大值。**在常用的数据结构中，能够在O(1)时间里得到最大值的数据结构为最大堆。因此我们可以用堆（heap）来代替数组。**

另外，自己重头开始写一个最大堆需要一定量的代码。我们现在不需要重新去发明车轮，因为前人早就发明出来了。同样，STL中的set和multiset为我们做了很好的堆的实现，我们可以拿过来用。既偷了懒，又给面试官留下熟悉STL的好印象，何乐而不为之？

参考代码：

#include <set>  
#include <vector>  
#include <iostream>  
  
using namespace std;  
  
typedef multiset<int, greater<int> >  IntHeap;  
  
///////////////////////////////////////////////////////////////////////  
// find k least numbers in a vector  
///////////////////////////////////////////////////////////////////////  
void FindKLeastNumbers  
(  
    const vector<int>& data,               // a vector of data  
    IntHeap& leastNumbers,                 // k least numbers, output  
    unsigned int k                                
)  
{  
    leastNumbers.clear();  
  
    if(k == 0 || data.size() < k)  
       return;  
  
    vector<int>::const\_iterator iter = data.begin();  
    for(; iter != data.end(); ++ iter)  
    {  
       // if less than k numbers was inserted into leastNumbers  
       if((leastNumbers.size()) < k)  
           leastNumbers.insert(\*iter);  
  
       // leastNumbers contains k numbers and it's full now  
       else  
       {  
           // first number in leastNumbers is the greatest one  
           IntHeap::iterator iterFirst = leastNumbers.begin();  
  
           // if is less than the previous greatest number   
           if(\*iter < \*(leastNumbers.begin()))  
           {  
              // replace the previous greatest number  
              leastNumbers.erase(iterFirst);  
              leastNumbers.insert(\*iter);  
           }  
       }  
    }  
}

**(06)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。**

参考代码：

using namespace std;  
///////////////////////////////////////////////////////////////////////  
// Verify whether a squence of integers are the post order traversal  
// of a binary search tree (BST)  
// Input: squence - the squence of integers  
//        length  - the length of squence  
// Return: return ture if the squence is traversal result of a BST,  
//         otherwise, return false  
///////////////////////////////////////////////////////////////////////  
bool verifySquenceOfBST(int squence[], int length)  
{  
    if(squence == NULL || length <= 0)  
       return false;  
  
    // root of a BST is at the end of post order traversal squence  
    int root = squence[length - 1];  
  
    // the nodes in left sub-tree are less than the root  
    int i = 0;  
    for(; i < length - 1; ++ i)  
    {  
       if(squence[i] > root)  
           break;  
    }  
  
    // the nodes in the right sub-tree are greater than the root  
    int j = i;  
    for(; j < length - 1; ++ j)  
    {  
       if(squence[j] < root)  
           return false;  
    }  
    // verify whether the left sub-tree is a BST  
    bool left = true;  
    if(i > 0)  
       left = verifySquenceOfBST(squence, i);  
  
    // verify whether the right sub-tree is a BST  
    bool right = true;  
    if(i < length - 1)  
       right = verifySquenceOfBST(squence + i, length - i - 1);  
  
    return (left && right);  
}

**(07)－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。

参考代码：

///////////////////////////////////////////////////////////////////////  
// Reverse a string between two pointers  
// Input: pBegin - the begin pointer in a string  
//        pEnd   - the end pointer in a string  
///////////////////////////////////////////////////////////////////////  
void Reverse(char \*pBegin, char \*pEnd)  
{  
    if(pBegin == NULL || pEnd == NULL)  
       return;  
  
    while(pBegin < pEnd)  
    {  
       char temp = \*pBegin;  
       \*pBegin = \*pEnd;  
       \*pEnd = temp;  
  
       pBegin ++, pEnd --;  
    }  
}  
  
///////////////////////////////////////////////////////////////////////  
// Reverse the word order in a sentence, but maintain the character  
// order inside a word  
// Input: pData - the sentence to be reversed  
///////////////////////////////////////////////////////////////////////  
char\* ReverseSentence(char \*pData)  
{  
    if(pData == NULL)  
       return NULL;  
  
    char \*pBegin = pData;  
    char \*pEnd = pData;  
  
    while(\*pEnd != '\0')  
       pEnd ++;  
    pEnd--;  
  
    // Reverse the whole sentence  
    Reverse(pBegin, pEnd);  
  
    // Reverse every word in the sentence  
    pBegin = pEnd = pData;  
    while(\*pBegin != '\0')  
    {  
       if(\*pBegin == ' ')  
       {  
           pBegin ++;  
           pEnd ++;  
           continue;  
       }  
       // A word is between with pBegin and pEnd, reverse it  
       else if(\*pEnd == ' ' || \*pEnd == '\0')  
       {  
           Reverse(pBegin, --pEnd);  
           pBegin = ++pEnd;  
       }  
       else  
       {  
           pEnd ++;  
       }  
    }  
  
    return pData;  
}

**(08)－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。我们可以将需要执行的代码放到构造函数里。**如下代码正是基于这个思路：

class Temp  
{  
public:  
    Temp() { ++ N; Sum += N; }  
  
    static void Reset() { N = 0; Sum = 0; }  
    static int GetSum() { return Sum; }  
  
private:  
    static int N;  
    static int Sum;  
};  
  
int Temp::N = 0;  
int Temp::Sum = 0;  
  
int solution1\_Sum(int n)  
{  
      Temp::Reset();  
  
    Temp \*a = new Temp[n];  
    delete []a;  
    a = 0;  
  
      return Temp::GetSum();  
}

**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。**有了上述分析，我们再来看下面的代码：

class A;  
A\* Array[2];  
  
class A  
{  
public:  
    virtual int Sum (int n) { return 0; }  
};  
  
class B: public A  
{  
public:  
    virtual int Sum (int n) { return Array[!!n]->Sum(n-1)+n; }  
};  
  
int solution2\_Sum(int n)  
{  
    A a;  
    B b;  
    Array[0] = &a;  
    Array[1] = &b;  
  
    int value = Array[1]->Sum(n);  
  
    return value;  
}

这种方法是用虚函数来实现函数的选择。当n不为零时，执行函数B::Sum；当n为0时，执行A::Sum。我们也可以直接用函数指针数组，这样可能还更直接一些：

typedef int (\*fun)(int);  
  
int solution3\_f1(int i)   
{  
    return 0;  
}  
  
int solution3\_f2(int i)  
{  
    fun f[2]={solution3\_f1, solution3\_f2};   
    return i+f[!!i](i-1);  
}

另外我们还可以让编译器帮我们来完成类似于递归的运算，比如如下代码：

template <int n> struct solution4\_Sum  
{  
    enum Value { N = solution4\_Sum<n - 1>::N + n};  
};

template <> struct solution4\_Sum<1>  
{  
    enum Value { N = 1};  
};

solution4\_Sum<100>::N就是1+2+...+100的结果。当编译器看到solution4\_Sum<100>时，就是为模板类solution4\_Sum以参数100生成该类型的代码。但以100为参数的类型需要得到以99为参数的类型，因为solution4\_Sum<100>::N=solution4\_Sum<99>::N+100。这个过程会递归一直到参数为1的类型，由于该类型已经显式定义，编译器无需生成，递归编译到此结束。由于这个过程是在编译过程中完成的，因此要求输入n必须是在编译期间就能确定，不能动态输入。这是该方法最大的缺点。而且编译器对递归编译代码的递归深度是有限制的，也就是要求n不能太大。

大家还有更多、更巧妙的思路吗？欢迎讨论^\_^

**(09)－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。

思路一的参考代码：

///////////////////////////////////////////////////////////////////////  
// Find the kth node from the tail of a list  
// Input: pListHead - the head of list  
//        k         - the distance to the tail  
// Output: the kth node from the tail of a list  
///////////////////////////////////////////////////////////////////////  
ListNode\* FindKthToTail\_Solution1(ListNode\* pListHead, unsigned int k)  
{  
    if(pListHead == NULL)  
       return NULL;  
  
    // count the nodes number in the list  
    ListNode \*pCur = pListHead;  
    unsigned int nNum = 0;  
    while(pCur->m\_pNext != NULL)  
    {  
       pCur = pCur->m\_pNext;  
       nNum ++;  
    }  
  
    // if the number of nodes in the list is less than k  
    // do nothing  
    if(nNum < k)  
       return NULL;  
  
    // the kth node from the tail of a list   
    // is the (n - k)th node from the head  
    pCur = pListHead;  
    for(unsigned int i = 0; i < nNum - k; ++ i)  
       pCur = pCur->m\_pNext;  
  
    return pCur;  
}

思路二的参考代码：

///////////////////////////////////////////////////////////////////////  
// Find the kth node from the tail of a list  
// Input: pListHead - the head of list  
//        k         - the distance to the tail  
// Output: the kth node from the tail of a list  
///////////////////////////////////////////////////////////////////////  
ListNode\* FindKthToTail\_Solution2(ListNode\* pListHead, unsigned int k)  
{  
    if(pListHead == NULL)  
       return NULL;  
  
    ListNode \*pAhead = pListHead;  
    ListNode \*pBehind = NULL;  
  
    for(unsigned int i = 0; i < k; ++ i)  
    {  
       if(pAhead->m\_pNext != NULL)  
           pAhead = pAhead->m\_pNext;  
       else  
       {  
           // if the number of nodes in the list is less than k,   
           // do nothing  
           return NULL;  
       }  
    }  
  
    pBehind = pListHead;  
  
    // the distance between pAhead and pBehind is k  
    // when pAhead arrives at the tail, p  
    // Behind is at the kth node from the tail  
    while(pAhead->m\_pNext != NULL)  
    {  
       pAhead = pAhead->m\_pNext;  
       pBehind = pBehind->m\_pNext;  
    }  
  
    return pBehind;  
}

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。

**(10)－在排序数组中查找和为给定值的两个数字**

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(n2)。

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。**

问题是这样的思路是不是正确的呢？这需要严格的数学证明。感兴趣的读者可以自行证明一下。

参考代码：

///////////////////////////////////////////////////////////////////////  
// Find two numbers with a sum in a sorted array  
// Output: ture is found such two numbers, otherwise false  
///////////////////////////////////////////////////////////////////////  
bool FindTwoNumbersWithSum  
(  
    int data[],           // a sorted array  
    unsigned int length,  // the length of the sorted array       
    int sum,              // the sum  
    int& num1,            // the first number, output  
    int& num2             // the second number, output  
)  
{

    bool found = false;  
    if(length < 1)  
       return found;  
  
    int ahead = length - 1;  
    int behind = 0;  
  
    while(ahead > behind)  
    {  
       long long curSum = data[ahead] + data[behind];  
  
       // if the sum of two numbers is equal to the input  
       // we have found them  
       if(curSum == sum)  
       {  
           num1 = data[behind];  
           num2 = data[ahead];  
           found = true;  
           break;  
       }  
       // if the sum of two numbers is greater than the input  
       // decrease the greater number  
       else if(curSum > sum)  
           ahead --;  
       // if the sum of two numbers is less than the input  
       // increase the less number  
       else  
           behind ++;  
    }  
  
    return found;  
}

扩展：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？

**(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。**参考代码如下：

///////////////////////////////////////////////////////////////////////  
// Mirror a BST (swap the left right child of each node) recursively  
// the head of BST in initial call  
///////////////////////////////////////////////////////////////////////  
void MirrorRecursively(BSTreeNode \*pNode)  
{  
    if(!pNode)  
       return;  
  
    // swap the right and left child sub-tree  
    BSTreeNode \*pTemp = pNode->m\_pLeft;  
    pNode->m\_pLeft = pNode->m\_pRight;  
    pNode->m\_pRight = pTemp;  
      
// mirror left child sub-tree if not null  
    if(pNode->m\_pLeft)  
       MirrorRecursively(pNode->m\_pLeft);    
  
    // mirror right child sub-tree if not null  
    if(pNode->m\_pRight)  
       MirrorRecursively(pNode->m\_pRight);   
}

由于递归的本质是编译器生成了一个函数调用的栈，因此用循环来完成同样任务时最简单的办法就是用一个辅助栈来模拟递归。首先我们把树的头结点放入栈中。在循环中，只要栈不为空，弹出栈的栈顶结点，交换它的左右子树。如果它有左子树，把它的左子树压入栈中；如果它有右子树，把它的右子树压入栈中。这样在下次循环中就能交换它儿子结点的左右子树了。参考代码如下：

///////////////////////////////////////////////////////////////////////  
// Mirror a BST (swap the left right child of each node) Iteratively  
// Input: pTreeHead: the head of BST  
///////////////////////////////////////////////////////////////////////  
void MirrorIteratively(BSTreeNode \*pTreeHead)  
{  
    if(!pTreeHead)  
       return;  
  
    std::stack<BSTreeNode\*>stackTreeNode;  
    stackTreeNode.push(pTreeHead);  
  
while(stackTreeNode.size())  
    {  
       BSTreeNode \*pNode = stackTreeNode.top();  
       stackTreeNode.pop();  
  
       // swap the right and left child sub-tree  
       BSTreeNode \*pTemp = pNode->m\_pLeft;  
       pNode->m\_pLeft = pNode->m\_pRight;  
       pNode->m\_pRight = pTemp;  
  
       // push left child sub-tree into stack if not null  
       if(pNode->m\_pLeft)  
           stackTreeNode.push(pNode->m\_pLeft);  
  
       // push right child sub-tree into stack if not null  
       if(pNode->m\_pRight)  
           stackTreeNode.push(pNode->m\_pRight);  
    }  
}

**(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。

参考代码：

#include <deque>  
#include <iostream>  
using namespace std;  
  
struct BTreeNode // a node in the binary tree  
{  
    int         m\_nValue; // value of node  
    BTreeNode  \*m\_pLeft;  // left child of node  
    BTreeNode  \*m\_pRight; // right child of node  
};  
  
///////////////////////////////////////////////////////////////////////  
// Print a binary tree from top level to bottom level  
// Input: pTreeRoot - the root of binary tree  
///////////////////////////////////////////////////////////////////////  
void PrintFromTopToBottom(BTreeNode \*pTreeRoot)  
{  
    if(!pTreeRoot)  
       return;  
  
    // get a empty queue  
    deque<BTreeNode \*> dequeTreeNode;  
  
    // insert the root at the tail of queue  
    dequeTreeNode.push\_back(pTreeRoot);  
  
    while(dequeTreeNode.size())  
    {  
       // get a node from the head of queue  
       BTreeNode \*pNode = dequeTreeNode.front();  
       dequeTreeNode.pop\_front();  
  
       // print the node  
       cout << pNode->m\_nValue << ' ';  
  
       // print its left child sub-tree if it has  
       if(pNode->m\_pLeft)  
           dequeTreeNode.push\_back(pNode->m\_pLeft);  
       // print its right child sub-tree if it has  
       if(pNode->m\_pRight)  
           dequeTreeNode.push\_back(pNode->m\_pRight);  
    }  
}

**(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(n2)。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。

参考代码如下：

///////////////////////////////////////////////////////////////////////  
// Find the first char which appears only once in a string  
// Input: pString - the string  
// Output: the first not repeating char if the string has, otherwise 0  
///////////////////////////////////////////////////////////////////////  
char FirstNotRepeatingChar(char\* pString)  
{  
    // invalid input  
    if(!pString)  
       return 0;  
  
    // get a hash table, and initialize it   
       constinttableSize =256;  
unsignedinthashTable[tableSize];  
for(unsignedinti = 0; i<tableSize; ++ i)  
hashTable[i] = 0;  
  
    // get the how many times each char appears in the string  
    char\* pHashKey = pString;  
    while(\*(pHashKey) != '\0')  
       hashTable[\*(pHashKey++)] ++;  
  
    // find the first char which appears only once in a string  
    pHashKey = pString;  
    while(\*pHashKey != '\0')  
    {  
       if(hashTable[\*pHashKey] == 1)  
           return \*pHashKey;  
  
       pHashKey++;  
    }  
  
    // if the string is empty   
    // or every char in the string appears at least twice  
    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：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。

在参考代码中，我们用STL中std::list来模拟这个环形列表。由于list并不是一个环形的结构，因此每次跌代器扫描到列表末尾的时候，要记得把跌代器移到列表的头部。这样就是按照一个圆圈的顺序来遍历这个列表了。

这种思路需要一个有n个结点的环形列表来模拟这个删除的过程，因此内存开销为O(n)。而且这种方法每删除一个数字需要m步运算，总共有n个数字，因此总的时间复杂度是O(mn)。当m和n都很大的时候，这种方法是很慢的。

接下来我们试着从数学上分析出一些规律。首先定义最初的n个数字（0,1,…,n-1）中最后剩下的数字是关于n和m的方程为f(n,m)。

在这n个数字中，第一个被删除的数字是m%n-1，为简单起见记为k。那么删除k之后的剩下n-1的数字为0,1,…,k-1,k+1,…,n-1，并且下一个开始计数的数字是k+1。相当于在剩下的序列中，k+1排到最前面，从而形成序列k+1,…,n-1,0,…k-1。该序列最后剩下的数字也应该是关于n和m的函数。由于这个序列的规律和前面最初的序列不一样（最初的序列是从0开始的连续序列），因此该函数不同于前面函数，记为f’(n-1,m)。最初序列最后剩下的数字f(n,m)一定是剩下序列的最后剩下数字f’(n-1,m)，所以f(n,m)=f’(n-1,m)。

接下来我们把剩下的的这n-1个数字的序列k+1,…,n-1,0,…k-1作一个映射，映射的结果是形成一个从0到n-2的序列：

k+1    ->    0  
k+2    ->    1  
…  
n-1    ->    n-k-2  
0   ->    n-k-1  
…  
k-1   ->   n-2

把映射定义为p，则p(x)= (x-k-1)%n，即如果映射前的数字是x，则映射后的数字是(x-k-1)%n。对应的逆映射是p-1(x)=(x+k+1)%n。

由于映射之后的序列和最初的序列有同样的形式，都是从0开始的连续序列，因此仍然可以用函数f来表示，记为f(n-1,m)。根据我们的映射规则，映射之前的序列最后剩下的数字f’(n-1,m)= p-1 [f(n-1,m)]=[f(n-1,m)+k+1]%n。把k=m%n-1代入得到f(n,m)=f’(n-1,m)=[f(n-1,m)+m]%n。

**经过上面复杂的分析，我们终于找到一个递归的公式。要得到n个数字的序列的最后剩下的数字，只需要得到n-1个数字的序列的最后剩下的数字，并可以依此类推。当n=1时，也就是序列中开始只有一个数字0，那么很显然最后剩下的数字就是0。**我们把这种关系表示为：

**0                  n=1  
f(n,m)={  
                 [f(n-1,m)+m]%n     n>1**

尽管得到这个公式的分析过程非常复杂，但它用递归或者循环都很容易实现。最重要的是，这是一种时间复杂度为O(n)，空间复杂度为O(1)的方法，因此无论在时间上还是空间上都优于前面的思路。

思路一的参考代码：

///////////////////////////////////////////////////////////////////////  
// n integers (0, 1, ... n - 1) form a circle. Remove the mth from   
// the circle at every time. Find the last number remaining   
// Input: n - the number of integers in the circle initially  
//        m - remove the mth number at every time  
// Output: the last number remaining when the input is valid,  
//         otherwise -1  
///////////////////////////////////////////////////////////////////////  
int LastRemaining\_Solution1(unsigned int n, unsigned int m)  
{  
    // invalid input  
    if(n < 1 || m < 1)  
       return -1;  
  
    unsigned int i = 0;  
  
    // initiate a list with n integers (0, 1, ... n - 1)  
    list<int> integers;  
    for(i = 0; i < n; ++ i)  
       integers.push\_back(i);  
  
    list<int>::iterator curinteger = integers.begin();  
    while(integers.size() > 1)  
    {  
       // find the mth integer. Note that std::list is not a circle  
       // so we should handle it manually  
       for(int i = 1; i < m; ++ i)  
       {  
           curinteger ++;  
           if(curinteger == integers.end())  
              curinteger = integers.begin();  
       }  
  
       // remove the mth integer. Note that std::list is not a circle  
       // so we should handle it manually  
       list<int>::iterator nextinteger = ++ curinteger;  
       if(nextinteger == integers.end())  
           nextinteger = integers.begin();  
  
       -- curinteger;  
       integers.erase(curinteger);  
       curinteger = nextinteger;  
    }  
  
    return \*(curinteger);  
}

思路二的参考代码：

///////////////////////////////////////////////////////////////////////  
// n integers (0, 1, ... n - 1) form a circle. Remove the mth from   
// the circle at every time. Find the last number remaining   
// Input: n - the number of integers in the circle initially  
//        m - remove the mth number at every time  
// Output: the last number remaining when the input is valid,  
//         otherwise -1  
///////////////////////////////////////////////////////////////////////  
int LastRemaining\_Solution2(int n, unsigned int m)  
{  
    // invalid input  
    if(n <= 0 || m < 0)  
       return -1;  
  
    // if there are only one integer in the circle initially,  
    // of course the last remaining one is 0  
    int lastinteger = 0;  
  
    // find the last remaining one in the circle with n integers  
    for (int i = 2; i <= n; i ++)   
       lastinteger = (lastinteger + m) % i;  
  
    return lastinteger;  
}

如果对两种思路的时间复杂度感兴趣的读者可以把n和m的值设的稍微大一点，比如十万这个数量级的数字，运行的时候就能明显感觉出这两种思路写出来的代码时间效率大不一样。

**(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，看到题目就能写出如下的递归求解的代码。

///////////////////////////////////////////////////////////////////////  
// Calculate the nth item of Fibonacci Series recursively  
///////////////////////////////////////////////////////////////////////  
long long Fibonacci\_Solution1(unsigned int n)  
{  
    int result[2] = {0, 1};  
    if(n < 2)  
       return result[n];  
  
    return Fibonacci\_Solution1(n - 1) + Fibonacci\_Solution1(n - 2);  
}

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

                  f(10)  
               /        \  
            f(9)         f(8)  
          /     \       /    \  
       f(8)     f(7)  f(6)   f(5)  
      /   \     /   \   
   f(7)  f(6)  f(6) f(5)

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)。

///////////////////////////////////////////////////////////////////////  
// Calculate the nth item of Fibonacci Series iteratively  
///////////////////////////////////////////////////////////////////////  
long long Fibonacci\_Solution2(unsigned n)  
{  
    int result[2] = {0, 1};  
    if(n < 2)  
       return result[n];  
  
    long long  fibNMinusOne = 1;  
    long long  fibNMinusTwo = 0;  
    long long  fibN = 0;  
    for(unsigned int i = 2; i <= n; ++ i)  
    {  
       fibN = fibNMinusOne + fibNMinusTwo;  
  
       fibNMinusTwo = fibNMinusOne;  
       fibNMinusOne = fibN;  
    }  
  
    return fibN;  
}

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：

        /  an/2\*an/2                      n为偶数时  
an=  
        \  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。完整的实现代码如下所示。

#include <cassert>  
  
///////////////////////////////////////////////////////////////////////  
// A 2 by 2 matrix  
///////////////////////////////////////////////////////////////////////  
struct Matrix2By2  
{  
    Matrix2By2  
    (  
       long long m00 = 0,   
       long long m01 = 0,   
       long long m10 = 0,   
       long long m11 = 0  
    )  
    :m\_00(m00), m\_01(m01), m\_10(m10), m\_11(m11)   
    {  
    }  
  
    long long m\_00;  
    long long m\_01;  
    long long m\_10;  
    long long m\_11;  
};  
  
///////////////////////////////////////////////////////////////////////  
// Multiply two matrices  
// Input: matrix1 - the first matrix  
//        matrix2 - the second matrix  
//Output: the production of two matrices  
///////////////////////////////////////////////////////////////////////  
Matrix2By2 MatrixMultiply  
(  
    const Matrix2By2& matrix1,   
    const Matrix2By2& matrix2  
)  
{  
    return Matrix2By2(  
       matrix1.m\_00 \* matrix2.m\_00 + matrix1.m\_01 \* matrix2.m\_10,  
       matrix1.m\_00 \* matrix2.m\_01 + matrix1.m\_01 \* matrix2.m\_11,  
       matrix1.m\_10 \* matrix2.m\_00 + matrix1.m\_11 \* matrix2.m\_10,  
       matrix1.m\_10 \* matrix2.m\_01 + matrix1.m\_11 \* matrix2.m\_11);  
}  
  
///////////////////////////////////////////////////////////////////////  
// The nth power of matrix   
// 1  1  
// 1  0  
///////////////////////////////////////////////////////////////////////  
Matrix2By2 MatrixPower(unsigned int n)  
{  
    assert(n > 0);  
  
    Matrix2By2 matrix;  
    if(n == 1)  
    {  
       matrix = Matrix2By2(1, 1, 1, 0);  
    }  
    else if(n % 2 == 0)  
    {  
       matrix = MatrixPower(n / 2);  
       matrix = MatrixMultiply(matrix, matrix);  
    }  
    else if(n % 2 == 1)  
    {  
       matrix = MatrixPower((n - 1) / 2);  
       matrix = MatrixMultiply(matrix, matrix);  
       matrix = MatrixMultiply(matrix, Matrix2By2(1, 1, 1, 0));  
    }  
  
    return matrix;  
}  
  
///////////////////////////////////////////////////////////////////////  
// Calculate the nth item of Fibonacci Series using devide and conquer  
///////////////////////////////////////////////////////////////////////  
long long Fibonacci\_Solution3(unsigned int n)  
{  
    int result[2] = {0, 1};  
    if(n < 2)  
       return result[n];  
  
    Matrix2By2 PowerNMinus2 = MatrixPower(n - 1);  
    return PowerNMinus2.m\_00;  
}

**(17)－把字符串转换成整数**

 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脸，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。

下面我们写出完整的实现代码。参考代码：

enum Status {kValid = 0, kInvalid};  
int g\_nStatus = kValid;  
  
///////////////////////////////////////////////////////////////////////  
// Convert a string into an integer  
///////////////////////////////////////////////////////////////////////  
int StrToInt(const char\* str)  
{  
    g\_nStatus = kInvalid;  
      longlongnum = 0;  
  
    if(str != NULL)  
    {  
       const char\* digit = str;  
  
       // the first char in the string maybe '+' or '-'  
       bool minus = false;  
       if(\*digit == '+')  
           digit ++;  
       else if(\*digit == '-')  
       {  
           digit ++;  
           minus = true;  
       }  
  
       // the remaining chars in the string  
       while(\*digit != '\0')  
       {  
           if(\*digit >= '0' && \*digit <= '9')  
           {  
              num = num \* 10 + (\*digit - '0');  
  
                        // overflow    
                                if(num>std::numeric\_limits<int>::max())  
{  
 num = 0;  
break;  
}  
  
digit++;  
}  
           // if the char is not a digit, invalid input  
           else  
           {  
              num = 0;  
              break;  
           }  
       }  
  
       if(\*digit == '\0')  
       {  
           g\_nStatus = kValid;  
           if(minus)  
              num = 0 - num;  
       }  
    }  
  
      return static\_cast<int>(num);  
}

讨论：在参考代码中，我选用的是第一种声明方式。不过在面试时，我们可以选用任意一种声明方式进行实现。但当面试官问我们选择的理由时，我们要对两者的优缺点进行评价。第一种声明方式对用户而言非常直观，但使用了全局变量，不够优雅；而第二种思路是用返回值来表明输入是否合法，在很多API中都用这种方法，但该方法声明的函数使用起来不够直观。

最后值得一提的是，在C语言提供的库函数中，函数atoi能够把字符串转换整数。它的声明是int atoi(const char \*str)。该函数就是用一个全局变量来标志输入是否合法的。

**(18)－用两个栈实现队列**

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了：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。

我们用一个表来总结一下前面的例子执行的步骤：

|  |  |  |
| --- | --- | --- |
| **操作** | **m\_stack1** | **m\_stack2** |
| **append a** | {a} | {} |
| **append b** | {a,b} | {} |
| **append c** | {a,b,c} | {} |
| **delete head** | {} | {b,c} |
| **delete head** | {} | {c} |
| **append d** | {d} | {c} |
| **delete head** | {d} | {} |

总结完push和pop对应的过程之后，我们可以开始动手写代码了。参考代码如下：

///////////////////////////////////////////////////////////////////////  
// Append a element at the tail of the queue  
///////////////////////////////////////////////////////////////////////  
template<typename T> void CQueue<T>::appendTail(const T& element)  
{  
    // push the new element into m\_stack1  
    m\_stack1.push(element);  
}

///////////////////////////////////////////////////////////////////////  
// Delete the head from the queue  
///////////////////////////////////////////////////////////////////////  
template<typename T> void CQueue<T>::deleteHead()  
{  
    // if m\_stack2is empty,and there are some  
//elements inm\_stack1, push them in m\_stack2  
if(m\_stack2.size()<= 0)  
{  
while(m\_stack1.size()>0)  
{  
T&data =m\_stack1.top();  
m\_stack1.pop();  
m\_stack2.push(data);  
}  
}  
  
// push theelement into m\_stack2  
assert(m\_stack2.size()>0);  
m\_stack2.pop();  
}

扩展：这道题是用两个栈实现一个队列。反过来能不能用两个队列实现一个栈？如果可以，该如何实现？

**(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。

基于上述分析，我们不难写出如下代码：

///////////////////////////////////////////////////////////////////////  
// Reverse a list iteratively  
// Input: pHead - the head of the original list  
// Output: the head of the reversed head  
///////////////////////////////////////////////////////////////////////  
ListNode\* ReverseIteratively(ListNode\* pHead)  
{  
    ListNode\* pReversedHead = NULL;  
    ListNode\* pNode = pHead;  
    ListNode\* pPrev = NULL;  
    while(pNode != NULL)  
    {  
       // get the next node, and save it at pNext  
       ListNode\* pNext = pNode->m\_pNext;  
  
       // if the next node is null, the currect is the end of original   
       // list, and it's the head of the reversed list  
       if(pNext == NULL)  
           pReversedHead = pNode;  
  
       // reverse the linkage between nodes  
       pNode->m\_pNext = pPrev;  
  
       // move forward on the the list  
       pPrev = pNode;  
       pNode = pNext;  
    }  
  
    return pReversedHead;  
}

扩展：本题也可以递归实现。感兴趣的读者请自己编写递归代码。

**(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Xm={x0, x1,…xm-1}和Yn={y0,y1,…,yn-1}为两个字符串，而Zk={z0,z1,…zk-1}是它们的LCS，则：

1.       如果xm-1=yn-1，那么zk-1=xm-1=yn-1，并且Zk-1是Xm-1和Yn-1的LCS；  
2.       如果xm-1≠yn-1，那么当zk-1≠xm-1时Z是Xm-1和Y的LCS；  
3.       如果xm-1≠yn-1，那么当zk-1≠yn-1时Z是Yn-1和X的LCS；

下面简单证明一下这些性质：

1.       如果zk-1≠xm-1，那么我们可以把xm-1（yn-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zk-1=xm-1=yn-1。

既然zk-1=xm-1=yn-1，那如果我们删除zk-1（xm-1、yn-1）得到的Zk-1，Xm-1和Yn-1，显然Zk-1是Xm-1和Yn-1的一个公共子串，现在我们证明Zk-1是Xm-1和Yn-1的LCS。用反证法不难证明。假设有Xm-1和Yn-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Xm-1和Y的LCS，则存在一个长度超过k的W是Xm-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Xm={x0, x1,…xm-1}和Yn={y0,y1,…,yn-1}的LCS，如果xm-1=yn-1，那么只需求得Xm-1和Yn-1的LCS，并在其后添加xm-1（yn-1）即可；如果xm-1≠yn-1，我们分别求得Xm-1和Y的LCS和Yn-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Xi和Yj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）](http://zhedahht.blog.163.com/blog/static/25411174200722991933440/)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。

参考代码如下：

#include "string.h"  
  
// directions of LCS generation  
enum decreaseDir {kInit = 0, kLeft, kUp, kLeftUp};  
  
/////////////////////////////////////////////////////////////////////////////  
// Get the length of two strings' LCSs, and print one of the LCSs  
// Input: pStr1         - the first string  
//        pStr2         - the second string  
// Output: the length of two strings' LCSs  
/////////////////////////////////////////////////////////////////////////////  
int LCS(char\* pStr1, char\* pStr2)  
{  
    if(!pStr1 || !pStr2)  
       return 0;  
      
size\_t length1 = strlen(pStr1);  
    size\_t length2 = strlen(pStr2);  
    if(!length1 || !length2)  
       return 0;  
      
size\_t i, j;  
      
// initiate the length matrix  
    int \*\*LCS\_length;  
    LCS\_length = (int\*\*)(new int[length1]);  
    for(i = 0; i < length1; ++ i)  
       LCS\_length[i] = (int\*)new int[length2];  
  
    for(i = 0; i < length1; ++ i)  
       for(j = 0; j < length2; ++ j)  
           LCS\_length[i][j] = 0;

    // initiate the direction matrix  
    int \*\*LCS\_direction;  
    LCS\_direction = (int\*\*)(new int[length1]);  
    for( i = 0; i < length1; ++ i)  
       LCS\_direction[i] = (int\*)new int[length2];  
  
    for(i = 0; i < length1; ++ i)  
       for(j = 0; j < length2; ++ j)  
           LCS\_direction[i][j] = kInit;  
  
    for(i = 0; i < length1; ++ i)  
    {  
       for(j = 0; j < length2; ++ j)  
       {  
           if(i == 0 || j == 0)  
           {  
              if(pStr1[i] == pStr2[j])  
              {  
                  LCS\_length[i][j] = 1;  
                  LCS\_direction[i][j] = kLeftUp;  
              }  
              else  
                  LCS\_length[i][j] = 0;  
           }  
           // a char of LCS is found,   
           // it comes from the left up entry in the direction matrix  
           else if(pStr1[i] == pStr2[j])  
           {  
              LCS\_length[i][j] = LCS\_length[i - 1][j - 1] + 1;  
              LCS\_direction[i][j] = kLeftUp;  
           }  
           // it comes from the up entry in the direction matrix  
           else if(LCS\_length[i - 1][j] > LCS\_length[i][j - 1])  
           {  
              LCS\_length[i][j] = LCS\_length[i - 1][j];  
              LCS\_direction[i][j] = kUp;  
           }  
           // it comes from the left entry in the direction matrix  
           else  
           {  
              LCS\_length[i][j] = LCS\_length[i][j - 1];  
              LCS\_direction[i][j] = kLeft;  
           }  
       }  
    }  
    LCS\_Print(LCS\_direction, pStr1, pStr2, length1 - 1, length2 - 1);  
  
    return LCS\_length[length1 - 1][length2 - 1];  
}

/////////////////////////////////////////////////////////////////////////////  
// Print a LCS for two strings  
// Input: LCS\_direction - a 2d matrix which records the direction of   
//                        LCS generation  
//        pStr1         - the first string  
//        pStr2         - the second string  
//        row           - the row index in the matrix LCS\_direction  
//        col           - the column index in the matrix LCS\_direction  
/////////////////////////////////////////////////////////////////////////////  
void LCS\_Print(int \*\*LCS\_direction,   
             char\* pStr1, char\* pStr2,   
             size\_t row, size\_t col)  
{  
    if(pStr1 == NULL || pStr2 == NULL)  
       return;  
  
    size\_t length1 = strlen(pStr1);  
    size\_t length2 = strlen(pStr2);  
      
if(length1 == 0 || length2 == 0 || !(row < length1 && col < length2))  
       return;  
  
    // kLeftUp implies a char in the LCS is found  
    if(LCS\_direction[row][col] == kLeftUp)  
    {  
       if(row > 0 && col > 0)  
           LCS\_Print(LCS\_direction, pStr1, pStr2, row - 1, col - 1);  
         
// print the char  
       printf("%c", pStr1[row]);  
    }  
    else if(LCS\_direction[row][col] == kLeft)  
    {  
       // move to the left entry in the direction matrix  
       if(col > 0)  
           LCS\_Print(LCS\_direction, pStr1, pStr2, row, col - 1);  
    }  
    else if(LCS\_direction[row][col] == kUp)  
    {  
       // move to the up entry in the direction matrix  
       if(row > 0)  
           LCS\_Print(LCS\_direction, pStr1, pStr2, row - 1, col);  
    }  
}

扩展：如果题目改成求两个字符串的最长公共子字符串，应该怎么求？子字符串的定义和子串的定义类似，但要求是连续分布在其他字符串中。比如输入两个字符串BDCABA和ABCBDAB的最长公共字符串有BD和AB，它们的长度都是2。

**(21)－左旋转字符串**

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XT。显然有(XT)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XTYT。接着再对XTYT进行翻转操作，得到(XTYT)T=(YT)T(XT)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。

参考代码如下：

#include "string.h"  
  
///////////////////////////////////////////////////////////////////////  
// Move the first n chars in a string to its end   
///////////////////////////////////////////////////////////////////////  
char\* LeftRotateString(char\* pStr, unsigned int n)  
{  
    if(pStr != NULL)  
    {  
       int nLength = static\_cast<int>(strlen(pStr));  
       if(nLength > 0 || n == 0 || n > nLength)  
       {  
           char\* pFirstStart = pStr;  
           char\* pFirstEnd = pStr + n - 1;  
           char\* pSecondStart = pStr + n;  
           char\* pSecondEnd = pStr + nLength - 1;  
  
           // reverse the first part of the string  
           ReverseString(pFirstStart, pFirstEnd);  
           // reverse the second part of the strint  
           ReverseString(pSecondStart, pSecondEnd);  
           // reverse the whole string  
           ReverseString(pFirstStart, pSecondEnd);  
       }  
    }  
  
    return pStr;  
}  
  
///////////////////////////////////////////////////////////////////////  
// Reverse the string between pStart and pEnd  
///////////////////////////////////////////////////////////////////////  
void ReverseString(char\* pStart, char\* pEnd)  
{  
    if(pStart == NULL || pEnd == NULL)  
    {  
       while(pStart <= pEnd)  
       {  
           char temp = \*pStart;  
           \*pStart = \*pEnd;  
           \*pEnd = temp;  
  
           pStart ++;  
           pEnd --;  
       }  
    }  
}

**(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。

得到的代码如下：

///////////////////////////////////////////////////////////////////////  
// Get how many 1s in an integer's binary expression  
///////////////////////////////////////////////////////////////////////  
int NumberOf1\_Solution1(int i)  
{  
    int count = 0;  
    while(i)  
    {  
       if(i & 1)  
           count ++;  
  
       i = i >> 1;  
    }  
  
    return count;  
}

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。基于此，我们得到如下代码：

///////////////////////////////////////////////////////////////////////  
// Get how many 1s in an integer's binary expression  
///////////////////////////////////////////////////////////////////////  
int NumberOf1\_Solution2(int i)  
{  
    int count = 0;  
    unsigned int flag = 1;  
    while(flag)  
    {  
       if(i & flag)  
           count ++;  
  
       flag = flag << 1;  
    }  
  
    return count;  
}

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。

这种思路对应的代码如下：

///////////////////////////////////////////////////////////////////////  
// Get how many 1s in an integer's binary expression  
///////////////////////////////////////////////////////////////////////  
int NumberOf1\_Solution3(int i)  
{  
    int count = 0;  
  
    while (i)  
    {  
       ++ count;  
       i = (i - 1) & i;  
    }  
  
    return count;  
}

扩展：如何用一个语句判断一个整数是不是二的整数次幂？

**(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](http://zhedahht.blog.163.com/blog/static/25411174200722991933440/)，这里就不在赘述了。

**(24)－栈的push、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：这到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。

我们以前面的序列4、5、3、2、1为例。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。接下来希望被pop的是5，由于仍然不是栈顶数字，我们接着在push序列中4以后的数字中寻找。找到数字5后再一次push进栈，这个时候5就是位于栈顶，可以被pop出来。接下来希望被pop的三个数字是3，2，1。每次操作前都位于栈顶，直接pop即可。

再来看序列4、3、5、1、2。pop数字4的情况和前面一样。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。

基于前面的分析，我们可以写出如下的参考代码：

#include <stack>  
  
/////////////////////////////////////////////////////////////////////////////  
// Given a push order of a stack, determine whether an array is possible to   
// be its corresponding pop order  
// Input: pPush   - an array of integers, the push order  
//        pPop    - an array of integers, the pop order  
//        nLength - the length of pPush and pPop  
// Output: If pPop is possible to be the pop order of pPush, return true.  
//         Otherwise return false  
/////////////////////////////////////////////////////////////////////////////  
bool IsPossiblePopOrder(const int\* pPush, const int\* pPop, int nLength)  
{  
    bool bPossible = false;  
  
    if(pPush && pPop && nLength > 0)  
    {  
       const int \*pNextPush = pPush;  
       const int \*pNextPop = pPop;  
  
       // ancillary stack  
       std::stack<int>stackData;  
  
       // check every integers in pPop  
       while(pNextPop - pPop < nLength)  
       {  
           // while the top of the ancillary stack is not the integer   
           // to be poped, try to push some integers into the stack  
           while(stackData.empty() || stackData.top() != \*pNextPop)  
           {  
              // pNextPush == NULL means all integers have been   
              // pushed into the stack, can't push any longer  
              if(!pNextPush)  
                  break;  
  
              stackData.push(\*pNextPush);  
  
              // if there are integers left in pPush, move   
              // pNextPush forward, otherwise set it to be NULL  
              if(pNextPush - pPush < nLength - 1)  
                  pNextPush ++;  
              else  
                  pNextPush = NULL;  
           }  
  
           // After pushing, the top of stack is still not same as   
           // pPextPop, pPextPop is not in a pop sequence  
           // corresponding to pPush  
           if(stackData.top() != \*pNextPop)  
              break;  
  
           // Check the next integer in pPop  
           stackData.pop();  
           pNextPop ++;  
       }  
  
       // if all integers in pPop have been check successfully,   
       // pPop is a pop sequence corresponding to pPush   
       if(stackData.empty() && pNextPop - pPop == nLength)  
           bPossible = true;  
    }  
  
    return bPossible;  
}

**(25)-在从1到n的正数中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。

首先我们来看最直观的方法，分别求得1到n中每个整数中1出现的次数。而求一个整数的十进制表示中1出现的次数，就和[本面试题系列的第22题](http://zhedahht.blog.163.com/blog/static/2541117420073118945734/)很相像了。我们每次判断整数的个位数字是不是1。如果这个数字大于10，除以10之后再判断个位数字是不是1。基于这个思路，不难写出如下的代码：

int NumberOf1(unsigned int n);  
  
/////////////////////////////////////////////////////////////////////////////  
// Find the number of 1 in the integers between 1 and n  
// Input: n - an integer  
// Output: the number of 1 in the integers between 1 and n  
/////////////////////////////////////////////////////////////////////////////  
int NumberOf1BeforeBetween1AndN\_Solution1(unsigned int n)  
{  
    int number = 0;  
  
    // Find the number of 1 in each integer between 1 and n  
    for(unsigned int i = 1; i <= n; ++ i)  
       number += NumberOf1(i);  
  
    return number;  
}  
  
/////////////////////////////////////////////////////////////////////////////  
// Find the number of 1 in an integer with radix 10  
// Input: n - an integer  
// Output: the number of 1 in n with radix  
/////////////////////////////////////////////////////////////////////////////  
int NumberOf1(unsigned int n)  
{  
    int number = 0;  
    while(n)  
    {  
       if(n % 10 == 1)  
           number ++;  
  
       n = n / 10;  
    }  
  
    return number;  
}

这个思路有一个非常明显的缺点就是每个数字都要计算1在该数字中出现的次数，因此时间复杂度是O(n)。当输入的n非常大的时候，需要大量的计算，运算效率很低。我们试着找出一些规律，来避免不必要的计算。

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1-1235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（104）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\*103，其中2的第一位的数值，103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\*103）。

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104（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104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。

基于前面的分析，我们可以写出以下的代码。在参考代码中，为了编程方便，我把数字转换成字符串了。

#include "string.h"  
#include "stdlib.h"  
  
int NumberOf1(const char\* strN);  
int PowerBase10(unsigned int n);  
  
/////////////////////////////////////////////////////////////////////////////  
// Find the number of 1 in an integer with radix 10  
// Input: n - an integer  
// Output: the number of 1 in n with radix  
/////////////////////////////////////////////////////////////////////////////  
int NumberOf1BeforeBetween1AndN\_Solution2(int n)  
{  
    if(n <= 0)  
       return 0;  
  
    // convert the integer into a string  
    char strN[50];  
    sprintf(strN, "%d", n);  
  
    return NumberOf1(strN);  
}

/////////////////////////////////////////////////////////////////////////////  
// Find the number of 1 in an integer with radix 10  
// Input: strN - a string, which represents an integer  
// Output: the number of 1 in n with radix  
/////////////////////////////////////////////////////////////////////////////  
int NumberOf1(const char\* strN)  
{  
    if(!strN || \*strN < '0' || \*strN > '9' || \*strN == '\0')  
       return 0;  
  
    int firstDigit = \*strN - '0';  
    unsigned int length = static\_cast<unsigned int>(strlen(strN));  
  
    // the integer contains only one digit  
    if(length == 1 && firstDigit == 0)  
       return 0;  
  
    if(length == 1 && firstDigit > 0)  
       return 1;  
  
    // suppose the integer is 21345  
    // numFirstDigit is the number of 1 of 10000-19999 due to the first digit  
    int numFirstDigit = 0;  
    // numOtherDigits is the number of 1 01346-21345 due to all digits  
    // except the first one  
    int numOtherDigits = firstDigit \* (length - 1) \* PowerBase10(length - 2);  
    // numRecursive is the number of 1 of integer 1345  
    int numRecursive = NumberOf1(strN + 1);  
  
    // if the first digit is greater than 1, suppose in integer 21345  
    // number of 1 due to the first digit is 10^4. It's 10000-19999  
    if(firstDigit > 1)  
       numFirstDigit = PowerBase10(length - 1);  
  
    // if the first digit equals to 1, suppose in integer 12345  
    // number of 1 due to the first digit is 2346. It's 10000-12345  
    else if(firstDigit == 1)  
       numFirstDigit = atoi(strN + 1) + 1;  
  
    return numFirstDigit + numOtherDigits + numRecursive;  
}  
  
/////////////////////////////////////////////////////////////////////////////  
// Calculate 10^n  
/////////////////////////////////////////////////////////////////////////////  
int PowerBase10(unsigned int n)  
{  
    int result = 1;  
    for(unsigned int i = 0; i < n; ++ i)  
       result \*= 10;  
  
    return result;  
}

**(26)-和为n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](http://zhedahht.blog.163.com/blog/static/2541117420072143251809/)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。

基于这个思路，我们可以写出如下代码：

void PrintContinuousSequence(int small, int big);  
  
/////////////////////////////////////////////////////////////////////////  
// Find continuous sequence, whose sum is n  
/////////////////////////////////////////////////////////////////////////  
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;  
    }  
}  
  
/////////////////////////////////////////////////////////////////////////  
// Print continuous sequence between small and big  
/////////////////////////////////////////////////////////////////////////  
void PrintContinuousSequence(int small, int big)  
{  
    for(int i = small; i <= big; ++ i)  
       printf("%d ", i);  
  
    printf("\n");  
}

**(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：

                                            10  
                                          /     \  
                                        6        14  
                                      /         /   \  
                                    4         12     16

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。

上面的这个思路用递归的方法很容易实现，只需要对遍历的代码稍作修改即可。参考代码如下：

///////////////////////////////////////////////////////////////////////  
// Get depth of a binary tree  
// Input: pTreeNode - the head of a binary tree  
// Output: the depth of a binary tree  
///////////////////////////////////////////////////////////////////////  
int TreeDepth(SBinaryTreeNode \*pTreeNode)  
{  
    // the depth of a empty tree is 0  
    if(!pTreeNode)  
       return 0;  
  
    // the depth of left sub-tree  
    int nLeft = TreeDepth(pTreeNode->m\_pLeft);  
    // the depth of right sub-tree  
    int nRight = TreeDepth(pTreeNode->m\_pRight);  
  
    // depth is the binary tree  
    return (nLeft > nRight) ? (nLeft + 1) : (nRight + 1);  
}

**(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。

基于前面的分析，我们可以得到如下的参考代码：

void Permutation(char\* pStr, char\* pBegin);  
  
/////////////////////////////////////////////////////////////////////////  
// Get the permutation of a string,   
// for example, input string abc, its permutation is   
// abc acb bac bca cba cab  
/////////////////////////////////////////////////////////////////////////  
void Permutation(char\* pStr)  
{  
    Permutation(pStr, pStr);  
}  
  
/////////////////////////////////////////////////////////////////////////  
// Print the permutation of a string,   
// Input: pStr   - input string  
//        pBegin - points to the begin char of string   
//                 which we want to permutate in this recursion  
/////////////////////////////////////////////////////////////////////////  
void Permutation(char\* pStr, char\* pBegin)  
{  
    if(!pStr || !pBegin)  
       return;  
  
    // if pBegin points to the end of string,  
    // this round of permutation is finished,   
    // print the permuted string  
    if(\*pBegin == '\0')  
    {  
       printf("%s\n", pStr);  
    }  
    // otherwise, permute string  
    else  
    {  
       for(char\* pCh = pBegin; \*pCh != '\0'; ++ pCh)  
       {  
           // swap pCh and pBegin  
           char temp = \*pCh;  
           \*pCh = \*pBegin;  
           \*pBegin = temp;  
  
           Permutation(pStr, pBegin + 1);  
  
           // restore pCh and pBegin  
           temp = \*pCh;  
           \*pCh = \*pBegin;  
           \*pBegin = temp;  
       }  
    }  
}

扩展1：如果不是求字符的所有排列，而是求字符的所有组合，应该怎么办呢？当输入的字符串中含有相同的字符串时，相同的字符交换位置是不同的排列，但是同一个组合。举个例子，如果输入aaa，那么它的排列是6个aaa，但对应的组合只有一个。

扩展2：输入一个含有8个数字的数组，判断有没有可能把这8个数字分别放到正方体的8个顶点上，使得正方体上三组相对的面上的4个顶点的和相等。

**(29)-调整数组顺序使奇数位于偶数前面**

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(n２)。

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。

基于这个思路，我们可以写出如下的代码：

void Reorder(int \*pData, unsigned int length, bool (\*func)(int));  
bool isEven(int n);  
  
/////////////////////////////////////////////////////////////////////////  
// Devide an array of integers into two parts, odd in the first part,  
// and even in the second part  
// Input: pData  - an array of integers  
//        length - the length of array  
/////////////////////////////////////////////////////////////////////////  
void ReorderOddEven(int \*pData, unsigned int length)  
{  
    if(pData == NULL || length == 0)  
       return;  
  
    Reorder(pData, length, isEven);  
}  
  
/////////////////////////////////////////////////////////////////////////  
// Devide an array of integers into two parts, the intergers which   
// satisfy func in the first part, otherwise in the second part  
// Input: pData  - an array of integers  
//        length - the length of array  
//        func   - a function  
/////////////////////////////////////////////////////////////////////////  
void Reorder(int \*pData, unsigned int length, bool (\*func)(int))  
{  
    if(pData == NULL || length == 0)  
       return;  
  
    int \*pBegin = pData;  
    int \*pEnd = pData + length - 1;  
  
    while(pBegin < pEnd)  
    {  
       // if \*pBegin does not satisfy func, move forward  
       if(!func(\*pBegin))  
       {  
           pBegin ++;  
           continue;  
       }  
  
       // if \*pEnd does not satisfy func, move backward  
       if(func(\*pEnd))  
       {  
           pEnd --;  
           continue;  
       }  
  
       // if \*pBegin satisfy func while \*pEnd does not,  
       // swap these integers  
       int temp = \*pBegin;  
       \*pBegin = \*pEnd;  
       \*pEnd = temp;  
    }  
}  
  
/////////////////////////////////////////////////////////////////////////  
// Determine whether an integer is even or not  
// Input: an integer  
// otherwise return false  
/////////////////////////////////////////////////////////////////////////  
bool isEven(int n)  
{  
    return (n & 1) == 0;  
}

讨论：

上面的代码有三点值得提出来和大家讨论：

１．函数isEven判断一个数字是不是偶数并没有用%运算符而是用&。理由是通常情况下位运算符比%要快一些；

２．这道题有很多变种。这里要求是把奇数放在偶数的前面，如果把要求改成：把负数放在非负数的前面等，思路都是都一样的。

３．在函数Reorder中，用函数指针func指向的函数来判断一个数字是不是符合给定的条件，而不是用在代码直接判断（hard code）。这样的好处是把调整顺序的算法和调整的标准分开了（即解耦，decouple）。当调整的标准改变时，Reorder的代码不需要修改，只需要提供一个新的确定调整标准的函数即可，提高了代码的可维护性。例如要求把负数放在非负数的前面，我们不需要修改Reorder的代码，只需添加一个函数来判断整数是不是非负数。这样的思路在很多库中都有广泛的应用，比如在ＳＴＬ的很多算法函数中都有一个仿函数（functor）的参数（当然仿函数不是函数指针，但其思想是一样的）。如果在面试中能够想到这一层，无疑能给面试官留下很好的印象。

**(30)-异常安全的赋值运算符重载函数**

题目：类CMyString的声明如下：

class CMyString  
{  
public:  
    CMyString(char\* pData = NULL);  
    CMyString(const CMyString& str);  
    ~CMyString(void);  
    CMyString& operator = (const CMyString& str);  
  
private:  
    char\* m\_pData;  
};

请实现其赋值运算符的重载函数，要求异常安全，即当对一个对象进行赋值时发生异常，对象的状态不能改变。

分析：首先我们来看一般C++教科书上给出的赋值运算符的重载函数：

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;  
}

我们知道，在分配内存时有可能发生异常。当执行语句new char[strlen(str.m\_pData) + 1]发生异常时，程序将从该赋值运算符的重载函数退出不再执行。注意到这个时候语句delete []m\_pData已经执行了。也就是说赋值操作没有完成，但原来对象的状态已经改变。也就是说不满足题目的异常安全的要求。

为了满足异常安全这个要求，一个简单的办法是掉换new、delete的顺序。先把内存new出来用一个临时指针保存起来，只有这个语句正常执行完成之后再执行delete。这样就能够保证异常安全了。

下面给出的是一个更加优雅的实现方案：

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;  
}

该方案通过调用构造拷贝函数创建一个临时对象来分配内存。此时即使发生异常，对原来对象的状态没有影响。交换临时对象和需要赋值的对象的字符串指针之后，由于临时对象的生命周期结束，自动调用其析构函数释放需赋值对象的原来的字符串空间。整个函数不需要显式用到new、delete，内存的分配和释放都自动完成，因此代码显得比较优雅。

**(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](http://zhedahht.blog.163.com/blog/static/2541117420073471124487/)，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。

基于这样的思路，不难写出如下代码：

///////////////////////////////////////////////////////////////////////

// Print a list from end to beginning

// Input: pListHead - the head of list

///////////////////////////////////////////////////////////////////////

void PrintListReversely(ListNode\* pListHead)

{

    if(pListHead != NULL)

    {

       // Print the next node first

       if (pListHead->m\_pNext != NULL)

       {

           PrintListReversely(pListHead->m\_pNext);

       }

       // Print this node

       printf("%d", pListHead->m\_nKey);

    }

}

扩展：该题还有两个常见的变体：

1.       从尾到头输出一个字符串；

2.       定义一个函数求字符串的长度，要求该函数体内不能声明任何变量。

**(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：

///////////////////////////////////////////////////////////////////////

// Define a class which can't be derived from

///////////////////////////////////////////////////////////////////////

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：

///////////////////////////////////////////////////////////////////////

// Define a class which can't be derived from

///////////////////////////////////////////////////////////////////////

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)。

基于前面的分析，我们不难写出下面的代码。

参考代码：

///////////////////////////////////////////////////////////////////////

// Delete a node in a list

// Input: pListHead - the head of list

//        pToBeDeleted - the node to be deleted

///////////////////////////////////////////////////////////////////////

void DeleteNode(ListNode\* pListHead, ListNode\* pToBeDeleted)

{

    if(!pListHead || !pToBeDeleted)

       return;

    // if pToBeDeleted is not the last node in the list

    if(pToBeDeleted->m\_pNext != NULL)

    {

       // copy data from the node next to pToBeDeleted

       ListNode\* pNext = pToBeDeleted->m\_pNext;

       pToBeDeleted->m\_nKey = pNext->m\_nKey;

       pToBeDeleted->m\_pNext = pNext->m\_pNext;

       // delete the node next to the pToBeDeleted

       delete pNext;

       pNext = NULL;

    }

    // if pToBeDeleted is the last node in the list

    else

    {

       // get the node prior to pToBeDeleted

       ListNode\* pNode = pListHead;

       while(pNode->m\_pNext != pToBeDeleted)

       {

           pNode = pNode->m\_pNext;

       }

       // deleted pToBeDeleted

       pNode->m\_pNext = NULL;

       delete pToBeDeleted;

       pToBeDeleted = NULL;

    }

}

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。

**(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。

基于上述思路，我们不难写出如下代码：

///////////////////////////////////////////////////////////////////////

// Find two numbers which only appear once in an array

// Input: data - an array contains two number appearing exactly once,

//               while others appearing exactly twice

//        length - the length of data

// Output: num1 - the first number appearing once in data

//         num2 - the second number appearing once in data

///////////////////////////////////////////////////////////////////////

void FindNumsAppearOnce(int data[], int length, int &num1, int &num2)

{

    if (length < 2)

       return;

    // get num1 ^ num2

    int resultExclusiveOR = 0;

    for (int i = 0; i < length; ++ i)

       resultExclusiveOR ^= data[i];

    // get index of the first bit, which is 1 in resultExclusiveOR

    unsigned int indexOf1 = FindFirstBitIs1(resultExclusiveOR);

    num1 = num2 = 0;

    for (int j = 0; j < length; ++ j)

    {

       // divide the numbers in data into two groups,

       // the indexOf1 bit of numbers in the first group is 1,

       // while in the second group is 0

       if(IsBit1(data[j], indexOf1))

           num1 ^= data[j];

       else

           num2 ^= data[j];

    }

}

///////////////////////////////////////////////////////////////////////

// Find the index of first bit which is 1 in num (assuming not 0)

///////////////////////////////////////////////////////////////////////

unsigned int FindFirstBitIs1(int num)

{

    int indexBit = 0;

    while (((num & 1) == 0) && (indexBit < 32))

    {

       num = num >> 1;

       ++ indexBit;

    }

    return indexBit;

}

///////////////////////////////////////////////////////////////////////

// Is the indexBit bit of num 1?

///////////////////////////////////////////////////////////////////////

bool IsBit1(int num, unsigned int indexBit)

{

    num = num >> indexBit;

    return (num & 1);

}

**(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)。

基于这个思路，我们不难写出如下的代码：

///////////////////////////////////////////////////////////////////////

// Find the first common node in the list with head pHead1 and

// the list with head pHead2

// Input: pHead1 - the head of the first list

//        pHead2 - the head of the second list

// Return: the first common node in two list. If there is no common

//         nodes, return NULL

///////////////////////////////////////////////////////////////////////

ListNode\* FindFirstCommonNode( ListNode \*pHead1, ListNode \*pHead2)

{

    // Get the length of two lists

    unsigned int nLength1 = ListLength(pHead1);

    unsigned int nLength2 = ListLength(pHead2);

    int nLengthDif = nLength1 - nLength2;

    // Get the longer list

    ListNode \*pListHeadLong = pHead1;

    ListNode \*pListHeadShort = pHead2;

    if(nLength2 > nLength1)

    {

       pListHeadLong = pHead2;

       pListHeadShort = pHead1;

       nLengthDif = nLength2 - nLength1;

    }

    // Move on the longer list

    for(int i = 0; i < nLengthDif; ++ i)

       pListHeadLong = pListHeadLong->m\_pNext;

    // Move on both lists

    while((pListHeadLong != NULL) &&

       (pListHeadShort != NULL) &&

       (pListHeadLong != pListHeadShort))

    {

       pListHeadLong = pListHeadLong->m\_pNext;

       pListHeadShort = pListHeadShort->m\_pNext;

    }

    // Get the first common node in two lists

    ListNode \*pFisrtCommonNode = NULL;

    if(pListHeadLong == pListHeadShort)

       pFisrtCommonNode = pListHeadLong;

    return pFisrtCommonNode;

}

///////////////////////////////////////////////////////////////////////

// Get the length of list with head pHead

// Input: pHead - the head of list

// Return: the length of list

///////////////////////////////////////////////////////////////////////

unsigned int ListLength(ListNode\* pHead)

{

    unsigned int nLength = 0;

    ListNode\* pNode = pHead;

    while(pNode != NULL)

    {

       ++ nLength;

       pNode = pNode->m\_pNext;

    }

    return nLength;

}

(36)-在字符串中删除特定的字符

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(n2) 。

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28=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](http://zhedahht.blog.163.com/blog/static/25411174200722191722430/) 。

基于上述分析，我们可以写出如下代码：

///////////////////////////////////////////////////////////////////////

// Delete all characters in pStrDelete from pStrSource

///////////////////////////////////////////////////////////////////////

void DeleteChars(char\* pStrSource, const char\* pStrDelete)

{

     if(NULL == pStrSource || NULL == pStrDelete)

        return;

     // Initialize an array, the index in this array is ASCII value.

     // All entries in the array, whose index is ASCII value of a

     // character in the pStrDelete, will be set as 1.

     // Otherwise, they will be set as 0.

     const unsigned int nTableSize = 256;

     int hashTable[nTableSize];

     memset(hashTable, 0, sizeof(hashTable));

     const char\* pTemp = pStrDelete;

     while ('\0' != \*pTemp)

     {

        hashTable[\*pTemp] = 1;

        ++ pTemp;

     }

     char\* pSlow = pStrSource;

     char\* pFast = pStrSource;

     while ('\0' != \*pFast)

     {

        // if the character is in pStrDelete, move both pStart and

        // pEnd forward, and copy pEnd to pStart.

        // Otherwise, move only pEnd forward, and the character

        // pointed by pEnd is deleted

        if(1 != hashTable[\*pFast])

        {

            \*pSlow = \*pFast;

            ++ pSlow;

        }

        ++pFast;

     }

     \*pSlow = '\0';

}

**IT公司笔试题算法部分**

**1、将一整数逆序后放入一数组中（要求递归实现）**

void convert(int \*result, int n)

{

    if(n>=10)

        convert(result+1, n/10);

    \*result = n%10;

}

int main(int argc, char\* argv[])

{

    int n = 123456789, result[20]={};

    convert(result, n);

    printf("%d:", n);

    for(int i=0; i<9; i++)

        printf("%d", result[i]);

    return getchar();

}

**2、求高于平均分的学生学号及成绩（学号和成绩人工输入）**

double find(int total, int n)

{

    int number, score,  average;

    scanf("%d", &number);

    if(number != 0){

        scanf("%d", &score);

        average = find(total+score, n+1);

        if(score >= average)

            printf("%d:%d\n", number, score);

        return average;

    }else{

        printf("Average=%d\n", total/n);

        return total/n;

    }

}

int main(int argc, char\* argv[])

{

    find(0, 0);

    return getchar();

}

**3、递归实现回文判断（如：abcdedbca就是回文）**

int find(char \*str, int n)

{

    if(n<=1)    return 1;

    else if(str[0]==str[n-1])   return find(str+1, n-2);

    else        return 0;

}

int main(int argc, char\* argv[])

{

    char \*str = "abcdedcba";

    printf("%s: %s\n", str, find(str,

 strlen(str)) ? "Yes" : "No");

    return getchar();

}

**4、组合问题（从M个不同字符中任取N个字符的所有组合）**

void find(char \*source, char \*result, int n)

{

    if(n==1){

        while(\*source)

           printf("%s%c\n", result, \*source++);

    }else{

        int i, j;

        for(i=0; source[i] != 0; i++);

        for(j=0; result[j] != 0; j++);

        for(; i>=n; i--)

        {

            result[j] = \*source++;

            result[j+1] = '\0';

            find(source, result, n-1);

        }

    }

}

int main(int argc, char\* argv[])

{

    int const n = 3;

    char \*source = "ABCDE", result[n+1] = {0};

    if(n>0 && strlen(source)>0 && n<=strlen(source))

        find(source, result, 3);

    return getchar();

}

**5、分解成质因数(如435234=251\*17\*17\*3\*2)**

void prim(int m, int n)

{

    if(m>n){

        while(m%n != 0) n++;

        m /= n;

        prim(m, n);

        printf("%d\*", n);

    }

}

int main(int argc, char\* argv[])

{

    int n = 435234;

    printf("%d=", n);

    prim(n, 2);

    return getchar();

}

**6、寻找迷宫的一条出路（o：通路； X障碍）**

#define MAX\_SIZE  8

int H[4] = {0, 1, 0, -1};

int V[4] = {-1, 0, 1, 0};

char Maze[MAX\_SIZE][MAX\_SIZE] = {{'X','X','X','X','X','X','X','X'},

                                 {'o','o','o','o','o','X','X','X'},

                                 {'X','o','X','X','o','o','o','X'},

                                {'X','o','X','X','o','X','X','o'},

                                {'X','o','X','X','X','X','X','X'},

{'X','o','X','X','o','o','o','X'},

                            {'X','o','o','o','o','X','o','o'},

                                 {'X','X','X','X','X','X','X','X'}};

void FindPath(int X, int Y)

{

    if(X == MAX\_SIZE || Y == MAX\_SIZE){

        for(int i = 0; i < MAX\_SIZE; i++)

for(int j = 0; j < MAX\_SIZE; j++)

                  printf("%c%c", Maze[i][j], j < MAX\_SIZE-1 ? ' ' : '\n');

}else for(int k = 0; k < 4; k++)

if(X >= 0 && Y >= 0 && Y < MAX\_SIZE && X < MAX\_SIZE && 'o' == Maze[X][Y]){

                    Maze[X][Y] = ' ';

                    FindPath(X+V[k], Y+H[k]);

                    Maze[X][Y] ='o';

}

}

int main(int argc, char\* argv[])

{

    FindPath(1,0);

    return getchar();

}

**7、随机分配座位，共50个学生，使学号相邻的同学座位不能相邻(早些时候用C#写的，没有用C改写）。**

static void Main(string[] args)

{

    int Tmp = 0, Count = 50;

    int[] Seats = new int[Count];

    bool[] Students = new bool[Count];

    System.Random RandStudent=new System.Random();

    Students[Seats[0]=RandStudent.Next(0,Count)]=true;

    for(int i = 1; i < Count; )

{

        Tmp=(int)RandStudent.Next(0,Count);

        if((!Students[Tmp])&&(Seats[i-1]-Tmp!=1) && (Seats[i-1] - Tmp) != -1){

            Seats[i++] = Tmp;

Students[Tmp] = true;

        }

    }

    foreach(int Student in Seats)

        System.Console.Write(Student + " ");

    System.Console.Read();

}

**8、求网格中的黑点分布（有6\*7的网格，在某些格子中有黑点，已知各行与各列中有黑点的点数之和）**

#define ROWS 6

#define COLS 7

int iPointsR[ROWS] = {2, 0, 4, 3, 4, 0};           // 各行黑点数和的情况

int iPointsC[COLS] = {4, 1, 2, 2, 1, 2, 1};        // 各列黑点数和的情况

int iCount, iFound;

int iSumR[ROWS], iSumC[COLS], Grid[ROWS][COLS];

int Set(int iRowNo)

{

if(iRowNo == ROWS){

        for(int iColNo=0; iColNo < COLS && iSumC[iColNo]==iPointsC[iColNo]; iColNo++)

           if(iColNo == COLS-1){

               printf("\nNo.%d:\n", ++iCount);

               for(int i=0; i < ROWS; i++)

                  for(int j=0; j < COLS; j++)

                      printf("%d%c", Grid[i][j], (j+1) % COLS ? ' ' : '\n');

               iFound = 1;                         // iFound = 1，有解

           }

    }else{

        for(int iColNo=0; iColNo < COLS; iColNo++)

        {

            if(iPointsR[iRowNo] == 0){

                Set(iRowNo + 1);

   }else if(Grid[iRowNo][iColNo]==0){

Grid[iRowNo][iColNo] = 1;

iSumR[iRowNo]++; iSumC[iColNo]++;                                  if(iSumR[iRowNo]<iPointsR[iRowNo] && iSumC[iColNo]<=iPointsC[iColNo])

                     Set(iRowNo);

else if(iSumR[iRowNo]==iPointsR[iRowNo] && iRowNo < ROWS)

                     Set(iRowNo + 1);

                Grid[iRowNo][iColNo] = 0;

                iSumR[iRowNo]--; iSumC[iColNo]--;

            }

        }

    }

return iFound;   // 用于判断是否有解

}

int main(int argc, char\* argv[])

{

    if(!Set(0))

        printf("Failure!");

    return getchar();

}

**9、有4种面值（面值为1, 4, 12, 21）的邮票很多枚，从中最多任取5张进行组合，求邮票最大连续组合值**

#define N 5

#define M 5

int k, Found, Flag[N];

int Stamp[M] = {0, 1, 4, 12, 21};

// 在剩余张数n中组合出面值和Value

int Combine(int n, int Value)

{

       if(n >= 0 && Value == 0){

              Found = 1;

              int Sum = 0;

              for(int i=0; i<N && Flag[i] != 0; i++){

                     Sum += Stamp[Flag[i]];

                     printf("%d ", Stamp[Flag[i]]);

              }

              printf("\tSum=%d\n\n", Sum);

       }else for(int i=1; i<M && !Found && n>0; i++)

              if(Value-Stamp[i] >= 0){

                     Flag[k++] = i;

                     Combine(n-1, Value-Stamp[i]);

                     Flag[--k] = 0;

              }

       return Found;

}

int main(int argc, char\* argv[])

{

       for(int i=1; Combine(N, i); i++, Found=0);

       return getchar();

}

**10、大整数数相乘的问题。**

void Multiple(char A[], char B[], char C[])

{

    int TMP, In=0, LenA=-1, LenB=-1;

    while(A[++LenA] != '\0');

    while(B[++LenB] != '\0');

    int Index, Start = LenA + LenB - 1;

    for(int i=LenB-1; i>=0; i--)

    {

        Index = Start--;

        if(B[i] != '0'){

            for(int In=0, j=LenA-1; j>=0; j--)

            {

                TMP = (C[Index]-'0') + (A[j]-'0') \* (B[i] - '0') + In;

                C[Index--] = TMP % 10 + '0';

                In = TMP / 10;

            }

            C[Index] = In + '0';

        }

    }

}

int main(int argc, char\* argv[])

{

    char A[] = "21839244444444448880088888889";

    char B[] = "38888888888899999999999999988";

char C[sizeof(A) + sizeof(B) - 1];

    for(int k=0; k<sizeof(C); k++)

        C[k] = '0';

    C[sizeof(C)-1] = '\0';

    Multiple(A, B, C);

    for(int i=0; C[i] != '\0'; i++)

        printf("%c", C[i]);

    return getchar();

}

**11、求最大连续递增数字串（如“ads3sl456789DF3456ld345AA”中的“456789”）**

int GetSubString(char \*strSource, char \*strResult)

{

    int iTmp=0, iHead=0, iMax=0;

    for(int Index=0, iLen=0; strSource[Index]; Index++)

    {

        if(strSource[Index] >= '0' && strSource[Index] <= '9'

&& strSource[Index-1] > '0' && strSource[Index] == strSource[Index-1]+1)

{

            iLen++;                     // 连续数字的长度增1

        }else

{                          // 出现字符或不连续数字

            if(iLen > iMax)

            {

            iMax = iLen;

iHead = iTmp;

            }

            // 该字符是数字，但数字不连续

            if(strSource[Index] >= '0' && strSource[Index] <= '9'){

                iTmp = Index;

iLen = 1;

            }

        }

    }

    for(iTmp=0 ; iTmp < iMax; iTmp++)   // 将原字符串中最长的连续数字串赋值给结果串

        strResult[iTmp] = strSource[iHead++];

    strResult[iTmp]='\0';

    return iMax;                               // 返回连续数字的最大长度

}

int main(int argc, char\* argv[])

{

    char strSource[]="ads3sl456789DF3456ld345AA", char strResult[sizeof(strSource)];

printf("Len=%d, strResult=%s \nstrSource=%s\n", GetSubString(strSource, strResult),

strResult, strSource);

    return getchar();

}

**12、四个工人，四个任务，每个人做不同的任务需要的时间不同，求任务分配的最优方案。（2005年5月29日全国计算机软件资格水平考试——软件设计师的算法题）。**

#include "stdafx.h"

#define N 4

int   Cost[N][N] = { {2, 12, 5, 32},            // 行号：任务序号，列号：工人序号

                    {8, 15, 7, 11},              // 每行元素值表示这个任务由不同工人完成所需要的时间

                    {24, 18, 9, 6},

                    {21, 1, 8, 28}};

int MinCost=1000;

int Task[N], TempTask[N], Worker[N];

void Assign(int k, int cost)

{

       if(k==N)

       {

              MinCost = cost;

              for(int i=0; i<N; i++)

                     TempTask[i] = Task[i];

       }else{

              for(int i=0; i<N; i++){

                     if(Worker[i]==0 && cost+Cost[k][i] < MinCost)

                     {

                            Worker[i] = 1;       Task[k] = i;

                            Assign(k+1, cost+Cost[k][i]);

                            Worker[i] = 0; Task[k] = 0;

                     }

              }

       }

}

int main(int argc, char\* argv[])

{

       Assign(0, 0);

       printf("最佳方案总费用=%d\n", MinCost);

       for(int i=0; i<N; i++)           /\* 输出最佳方案 \*/

              printf("\t任务%d由工人%d来做：%d\n", i, TempTask[i], Cost[i][TempTask[i]]);

       return getchar();

}

**13、八皇后问题（输出所有情况，不过有些结果只是旋转了90度而已）。哈哈：）回溯算法的典型例题**

#define N 8

int Board[N][N];

int Valid(int i, int j)              // 所下棋子有效性的严正

{

       int k = 1;

       for(k=1; i>=k && j>=k;k++)

              if(Board[i-k][j-k])  return 0;

       for(k=1; i>=k;k++)

              if(Board[i-k][j])            return 0;

       for(k=1; i>=k && j+k<N;k++)

              if(Board[i-k][j+k]) return 0;

       return 1;

}

void Trial(int i, int n)

{

       if(i==n){

              for(int k=0; k<n; k++){

                     for(int m=0; m<n; m++)

                            printf("%d ", Board[k][m]);

                     printf("\n");

              }

              printf("\n");

       }else{

              for(int j=0; j<n; j++){

                     Board[i][j] = 1;

                     if(Valid(i,j))

                            Trial(i+1, n);

                     Board[i][j] = 0;

              }

       }

}

int main(int argc, char\* argv[])

{

       Trial(0, N);

       return getchar();

}

**14、实现strstr功能（寻找子串在父串中首次出现的位置）**

char \* strstring(char \*ParentString, char \*SubString)

{

       char \*pSubString, \*pPareString;

       for(char \*pTmp=ParentString; \*pTmp; pTmp++)

       {

              pSubString = SubString;

              pPareString = pTmp;

              while(\*pSubString == \*pPareString && \*pSubString != '\0')

              {

                     pSubString++;

                     pPareString++;

              }

              if(\*pSubString == '\0') return pTmp;

       }

       return NULL;

}

int main(int argc, char\* argv[])

{

       char \*ParentString = "happy birthday to you!";

       char \*SubString = "birthday";

       printf("%s",strstring(ParentString, SubString));

       return getchar();

}}

**15、现在小明一家过一座桥，过桥的时候是黑夜，所以必须有灯。现在小明过桥要1秒，小明的弟弟要3秒，小明的爸爸要6秒，小明的妈妈要八秒，小明的爷爷要12秒。每次此桥最多可过两人，而过桥的速度依过桥最慢者而定，而且灯在点燃后30秒就会熄灭。问小明一家如何过桥？（原本是个智力题，这里用程序来求解）**

#include "stdafx.h"

#define N    5

#define SIZE 64

// 将人员编号：小明-0，弟弟-1，爸爸-2，妈妈-3，爷爷-4

// 每个人的当前位置：0--在桥左边， 1--在桥右边

int Position[N];

// 过桥临时方案的数组下标； 临时方案； 最小时间方案；

int Index, TmpScheme[SIZE], Scheme[SIZE];

// 最小过桥时间总和，初始值100；每个人过桥所需要的时间

int MinTime=100, Time[N]={1, 3, 6, 8, 12};

// 寻找最佳过桥方案。Remnant:未过桥人数; CurTime:当前已用时间;

// Direction:过桥方向,1--向右,0--向左

void Find(int Remnant, int CurTime, int Direction)

{

    if(Remnant==0){                               // 所有人已经过桥，更新最少时间及方案

        MinTime=CurTime;

        for(int i=0; i<SIZE && TmpScheme[i]>=0; i++)

        {

            Scheme[i]=TmpScheme[i];

        }

    }else if(Direction==1){                        // 过桥方向向右，从桥左侧选出两人过桥

        for(int i=0; i<N; i++)

        {

            if(Position[i]==0 && CurTime+Time[i]<MinTime){

                TmpScheme[Index++] = i;

                Position[i] = 1;

                for(int j=0; j<N; j++)

                {

                    int TmpMax = (Time[i]>Time[j] ? Time[i] : Time[j]);

                    if(Position[j]==0 && CurTime+TmpMax<MinTime)

                    {

                        TmpScheme[Index++] = j;

                        Position[j] = 1;

                        Find(Remnant-2, CurTime+TmpMax, !Direction);

                        Position[j] = 0;

                        TmpScheme[--Index] = -1;

                    }

                }

                Position[i] = 0;

                TmpScheme[--Index] = -1;

            }

        }

    }else{        // 过桥方向向左，从桥右侧选出一个人回来送灯

        for(int j=0; j<N; j++)

        {

            if(Position[j]==1 && CurTime+Time[j] < MinTime)

            {

                TmpScheme[Index++] = j;

                Position[j] = 0;

                Find(Remnant+1, CurTime+Time[j], !Direction);

                Position[j] = 1;

                TmpScheme[--Index] = -1;

            }

        }

    }

}

int main(int argc, char\* argv[])

{

    for(int i=0; i<SIZE; i++)             // 初始方案内容为负值，避免和人员标号冲突

        Scheme[i] = TmpScheme[i] = -1;

Find(N, 0, 1);                          // 查找最佳方案

    printf("MinTime=%d:", MinTime);      // 输出最佳方案

    for(int i=0; i<SIZE && Scheme[i]>=0; i+=3)

        printf("  %d-%d  %d", Scheme[i], Scheme[i+1], Scheme[i+2]);

    printf("\b\b  ");

    return getchar();

}

**16、2005年11月金山笔试题。编码完成下面的处理函数。函数将字符串中的字符'\*'移到串的前部分，前面的非'\*'字符后移，但不能改变非'\*'字符的先后顺序，函数返回串中字符'\*'的数量。如原始串为：ab\*\*cd\*\*e\*12，处理后为\*\*\*\*\*abcde12，函数并返回值为5。（要求使用尽量少的时间和辅助空间）**

int change(char \*str)                                  /\* 这个算法并不高效，从后向前搜索效率要高些 \*/

{

       int count = 0;                              /\* 记录串中字符'\*'的个数 \*/

       for(int i=0, j=0; str[i]; i++)          /\* 重串首开始遍历 \*/

       {

              if(str[i]=='\*'){                      /\* 遇到字符'\*' \*/

                     for(j=i-1; str[j]!='\*'&&j>=0; j--) /\* 采用类似插入排序的思想，将\*前面 \*/

                            str[j+1]=str[j];                       /\* 的非\*字符逐个后移，直到遇到\*字符 \*/

                     str[j+1] = '\*';

                     count++;

              }

       }

       return count;

}

int main(int argc, char\* argv[])

{

       char str[] = "ab\*\*cd\*\*e\*12";

       printf("str1=%s\n", str);

       printf("str2=%s, count=%d", str, change(str));

       return getchar();

}

// 终于得到一个比较高效的算法，一个网友提供，应该和金山面试官的想法一致。算法如下：

int change(char \*str)

{

       int i,j=strlen(str)-1;

       for(i=j; j>=0; j--)

       {

              if(str[i]!='\*'){

                     i--;

              }else if(str[j]!='\*'){

                     str[i] = str[j];

                     str[j] = '\*';

                     i--;

              }

       }

       return i+1;

}

**17、2005年11月15日华为软件研发笔试题。实现一单链表的逆转。**

#include "stdafx.h"

typedef char eleType;            // 定义链表中的数据类型

typedef struct listnode           // 定义单链表结构

{

       eleType data;

       struct listnode \*next;

}node;

node \*create(int n)        // 创建单链表，n为节点个数

{

       node \*p = (node \*)malloc(sizeof(node));

       node \*head = p;     head->data = 'A';

       for(int i='B'; i<'A'+n; i++)

       {

              p = (p->next = (node \*)malloc(sizeof(node)));

              p->data = i;

              p->next = NULL;

       }

       return head;

}

void print(node \*head)          // 按链表顺序输出链表中元素

{

       for(; head; head = head->next)

              printf("%c ", head->data);

       printf("\n");

}

node \*reverse(node \*head, node \*pre)  // 逆转单链表函数。这是笔试时需要写的最主要函数

{

       node \*p=head->next;

       head->next = pre;

       if(p)

              return reverse(p, head);

       else

              return head;

}

int main(int argc, char\* argv[])

{

       node \*head = create(6);

       print(head);

       head = reverse(head, NULL);

       print(head);

       return getchar();

}

**18、编码实现字符串转整型的函数（实现函数atoi的功能），据说是神州数码笔试题。如将字符串 ”+123”123, ”-0123”-123, “123CS45”123, “123.45CS”123, “CS123.45”0**

#include "stdafx.h"

int str2int(const char \*str)            // 字符串转整型函数

{

       int i=0, sign=1, value = 0;

       if(str==NULL)       return NULL;         // 空串直接返回 NULL

       if(str[0]=='-' || str[0]=='+'){   // 判断是否存在符号位

              i = 1;

              sign = (str[0]=='-' ? -1 : 1);

       }

       for(; str[i]>='0' && str[i]<='9'; i++)     // 如果是数字，则继续转换

              value = value \* 10 + (str[i] - '0');

       return sign \* value;

}

int main(int argc, char \*argv[])

{

       char \*str = "-123.45CS67";

       int  val  = str2int(str);

       printf("str=%s\tval=%d\n", str, val);

       return getchar();

}

**19、歌德巴赫猜想。任何一个偶数都可以分解为两个素数之和。**

#include "stdafx.h"

#include "math.h"

int main(int argc, char\* argv[])

{

       int Even=78, Prime1, Prime2, Tmp1, Tmp2;

       for(Prime1=3; Prime1<=Even/2; Prime1+=2)

       {

              for(Tmp1=2,Tmp2=sqrt(float(Prime1)); Tmp1<=Tmp2 && Prime1%Tmp1 != 0; Tmp1++);

              if(Tmp1<=Tmp2) continue;

              Prime2 = Even-Prime1;

              for(Tmp1=2,Tmp2=sqrt(float(Prime2)); Tmp1<=Tmp2 && Prime2%Tmp1 != 0; Tmp1++);

              if(Tmp1<=Tmp2) continue;

              printf("%d=%d+%d\n", Even, Prime1, Prime2);

       }

       return getchar();

}

**20、快速排序（东软喜欢考类似的算法填空题，又如堆排序的算法等）**

#include "stdafx.h"

#define N 10

int part(int list[], int low, int high)   // 一趟排序，返回分割点位置

{

     int tmp = list[low];

     while(low<high){

         while(low<high && list[high]>=tmp) --high;

         list[low] = list[high];

         while(low<high && list[low]<=tmp) ++low;

         list[high] = list[low];

     }

     list[low] = tmp;

     return low;

}

void QSort(int list[], int low, int high) // 应用递归进行快速排序

{

     if(low<high){

         int mid = part(list, low, high);

         QSort(list, low, mid-1);

         QSort(list, mid+1, high);

     }

}

void show(int list[], int n)              // 输出列表中元素

{

     for(int i=0; i<n; i++)

         printf("%d ", list[i]);

     printf("\n");

}

int main(int argc, char\* argv[])

{

     int list[N] = {23, 65, 26, 1, 6, 89, 3, 12, 33, 8};

     show(list, N);                       // 输出排序前序列

     QSort(list, 0, N-1);                 // 快速排序

     show(list, N);                       // 输出排序后序列

     return getchar();

}

**21、2005年11月23日慧通笔试题：写一函数判断某个整数是否为回文数，如12321为回文数。可以用判断入栈和出栈是否相同来实现（略微复杂些），这里是将整数逆序后形成另一整数，判断两个整数是否相等来实现的。**

#include "stdafx.h"

int IsEchoNum(int num)

{

     int m = 0;

     for(int n = num; n; n/=10)

         m = m\*10 + n%10;

     return m==num;

}

int main(int argc, char\* argv[])

{

     int num = 12321;

     printf("%d %d\n", num, IsEchoNum(num));

     return getchar();

}

**22、删除字符串中的数字并压缩字符串（神州数码以前笔试题），如字符串”abc123de4fg56”处理后变为”abcdefg”。注意空间和效率。（下面的算法只需要一次遍历，不需要开辟新空间，时间复杂度为O(N)）**

#include "stdafx.h"

void delNum(char \*str)

{

     int i, j=0;

     for(i=j=0; str[i] && (str[i]<'0' || str[i]>'9'); j=++i);// 找到串中第一个数字的位子

     for(; str[i]; i++)     // 从串中第一个数字的位置开始，逐个放入后面的非数字字符

          if(str[i]<'0' || str[i]>'9')

              str[j++] = str[i];

     str[j] = '\0';

}

int main(int argc, char\* argv[])

{

     char str[] = "abc123ef4g4h5";

     printf("%s\n", str);

     delNum(str);

     printf("%s\n", str);

     return getchar();

}

**23、求两个串中的第一个最长子串（神州数码以前试题）。如"abractyeyt","dgdsaeactyey"的最大子串为"actyet"。**

#include "stdafx.h"

char \*MaxSubString(char \*str1, char \*str2)

{

     int i, j, k, index, max=0;

     for(i=0; str1[i]; i++)

         for(j=0; str2[j]; j++)

         {

              for(k=0; str1[i+k]==str2[j+k] && (str2[i+k] || str1[i+k]); k++);

              if(k>max){         // 出现大于当前子串长度的子串，则替换子串位置和程度

                   index = j;    max = k;

              }

         }

     char \*strResult = (char \*)calloc(sizeof(char), max+1);

     for(i=0; i<max; i++)

         strResult[i] = str2[index++];

     return strResult;

}

int main(int argc, char\* argv[])

{

     char str1[] = "abractyeyt", str2[] = "dgdsaeactyey";

     char \*strResult = MaxSubString(str1, str2);

     printf("str1=%s\nstr2=%s\nMaxSubString=%s\n", str1, str2, strResult);

     return getchar();

}

**24、不开辟新空间完成字符串的逆序**

#include "stdafx.h"

void change(char \*str)

{

     for(int i=0,j=strlen(str)-1; i<j; i++, j--)

{

         str[i] ^= str[j] ^= str[i] ^= str[j];

     }

}

int main(int argc, char\* argv[])

{

     char str[] = "abcdefg";

     printf("strSource=%s\n", str);

     change(str);

     printf("strResult=%s\n", str);

     return getchar();

}

**25、删除串中指定的字符**

#include "stdafx.h"

void delChar(char \*str, char c)

{

     int i, j=0;

     for(i=0; str[i]; i++)

         if(str[i]!=c) str[j++]=str[i];

     str[j] = '\0';

}

int main(int argc, char\* argv[])

{

     char str[] = "abcdefgh";    // 注意，此处不能写成char \*str = "abcdefgh";

     printf("%s\n", str);

     delChar(str, 'c');

     printf("%s\n", str);

     return getchar();

}

**26、判断单链表中是否存在环（网上说的笔试题）**

#include "stdafx.h"

typedef char eleType;       // 定义链表中的数据类型

typedef struct listnode     // 定义单链表结构

{

     eleType data;

     struct listnode \*next;

}node;

node \*create(int n)              // 创建单链表，n为节点个数

{

     node \*p = (node \*)malloc(sizeof(node));

     node \*head = p;    head->data = 'A';

     for(int i='B'; i<'A'+n; i++)

     {

         p = (p->next = (node \*)malloc(sizeof(node)));

         p->data = i;

         p->next = NULL;

     }

     return head;

}

void addCircle(node \*head, int n)    // 增加环，将链尾指向链中第n个节点

{

     node \*q, \*p = head;

     for(int i=1; p->next; i++)

     {

         if(i==n) q = p;

          p = p->next;

     }

     p->next = q;

}

int isCircle(node \*head)    // 这是笔试时需要写的最主要函数，其他函数可以不写

{

     node \*p=head,\*q=head;

     while( p->next && q->next)

     {

         p = p->next;

         if (NULL == (q=q->next->next)) return 0;

         if (p == q)

              return 1;

     }

     return 0;

}

int main(int argc, char\* argv[])

{

     node \*head = create(12);

     addCircle(head, 8);         // 注释掉此行，连表就没有环了

     printf("%d\n", isCircle(head));

     return getchar();

}

**难题**

**喝酒问题**

由n元组(x1,x2,……，xn)组成的一个状态空间 E={(x1,x2,……，xn) | xi ∈Si, i=1,2,..,n}，给定关于n元组中的分量的一个约束集D，要求E中满足D的全部约束的所有n元组。其中Si是分量xi的定义域且|Si|有限，i=1,2,...n。我们称E中满足D的全部约束条件的任一n元组为问题P的一个解。  
  
对于n元组(x1,x2,……，xn)中分量的约束，一般分为两类，一类是显约束，它给出对于n元组中分量的显式限制，比如当i≠j时xi≠xj；另一类是隐约束，它给出对于n元组中分量的隐式限制，比如：f(x1,x2,……，xn)≠ 0，其中f是隐函数。不过隐式显式并不绝对，两者可以相互转换。  
  
解问题P的最朴素的方法是穷举法，即对E中的所有n元组，逐一地检测其是否满足D的全部约束。全部满足，才是问题p的解;只要有一个不满足，就不是问题P的解。显然，如果记m(i)=|S(i+1)|,i=0,1,...n-1，那么，穷举法需要对m=m(0)\*m(1)\*...\*m(n-1)个n元组一个不漏地加以检测。可想而知，其计算量是非常之大的。  
  
我们发现，对于许多问题，所给定的约束集D具有完备性，即i元组(x1,x2,……，xi)满足D中仅涉及到x1,x2,……，xi的所有约束意味着j(j<i)元组(x1,x2,……，xj)一定也满足D中仅涉及到x1,x2,……，xj的所有约束，i=1,2,……,n。换句话说，只要存在O≤j≤n-1，使得(x1,x2,……，xj)违反D中仅涉及到x1,x2,……，xj的约束之一，以(x1,x2,……，xj)为前缀的任何n元组(x1,x2,……,xj,……,xn)一定也违反D中仅涉及到又x1,x2,……，xi的一个约束，其中n≥i≥j。  
  
这个发现告诉我们，对于约束集D具有完备性的问题P，一旦检测断定某个j元组(x1,x2,……,xj)违反D中仅涉及x1,x2,……,xj的一个约束，就可以肯定，以(x1,x2,……,xj)为前缀的任何n元组(x1,x2,……,xj,……,xn)都不会是问题的解，因而就不必去搜索它们、检测它们。回溯法正是针对这类问题，利用这类问题的上述性质而提出来的比穷举法效率高得多的算法。  
  
回溯法首先将问题P的n元组的状态空间E表示成一棵高为n的带权有序树T，把在E中求问题P的所有解转化为在T中搜索问题P的所有解。树T类似于检索树。它可这样构造：设Si中的元素可排成x(i,1),x(i,2),……,x(i,m(i-1)),i=1,2,……,n。从根开始，让T的第i层的每一个结点都有m(i)个儿子。这m(i)个儿子到它们的共同父亲的边，按从左到右的次序分别带权x(i+1,1),x(i+1,2),……,x(i+1,m(i)),i=0,1,2,……,n-1。照这种构造方式，E中的一个n元组(x1,x2,……，xn)对应于T中的一个叶结点，T的根到这个叶结点的路上依次的n条边分别以x1,x2,……，xn为其权，反之亦然。另外，对于任意的0≤i≤n-1，E中n元组(x1,x2,……，xn)的一个前缀i元组(x1,x2,……，xi)对应于T中的一个非叶结点，T的根到这个非叶结点的路上依次的i条边分别以了x1,x2,……，xi为其权，反之亦然。特别，E中的任意一个n元组的空前缀()，对应于T的根。  
  
因而，在E中寻找问题P的一个解等价于在T中搜索一个叶结点，要求从T的根到该叶结点的路上依次的n条边相应带的n个权x1,x2,……，xn满足约束集D的全部约束。在T中搜索所要求的叶结点，很自然的一种方式是从根出发逐步深入，让路逐步延伸，即依次搜索满足约柬条件的前缀1元组(xl)，前缀2元组(xl,x2)，前缀i元组(x1,x2,……，xi)，……，直到i=n为止。注意，在这里，我们把(x1,x2,……，xi)应该满足的D中仅涉及x1,x2,……，xi的所有约束当做判断(x1,x2,……，xi)是问题p的解的必要条件，只有当这个必要条件加上条件i=n才是充要条件。为了区别，我们称使积累的判别条件成为充要条件的那个条件(如条件i=n)为终结条件。  
  
在回溯法中，上面引入的树T被称为问题P的状态空间树;树T上的任意一个结点被称为问题p的状态结点;树T上的任意一个叶结点被称为问题P的一个解状态结点;树T上满足约束集D的全部约柬的任意一个叶结点被称为问题P的一个回答状态结点，简称为回答结点或回答状态，它对应于问题P的一个解。  
  
例如8皇后问题，就是要确定一个8元组（x1,x2,..,x8)，xi表示第i行的皇后所在的列，这样的问题很容易应用上面的搜索树模型；然而，有些问题的解无法表示成一个n元组，因为事先无法确定这个n是多少，比如这个喝酒问题，问题的解就是一系列的倒酒喝酒策略，但是事先无法确定究竟需要进行多少步；还有著名的8数码问题（文曲星上的那个9x9方格中移数字的游戏），那个问题也是预先不知道需要移动多少步才能达到目标。不过这并不影响回溯法的使用，只要该问题有解，一定可以将解用有限的变元来表示，我们可以假设n就是问题的一个解的变元的个数，这样就可以继续利用上面的搜索树模型了。事实上，这棵搜索树并非预先生成的，而是在搜索的过程中逐步生成的，所以不知道树的深度n并不影响在树中搜索叶子节点。但是有一点很重要，如果问题根本不存在有限的解，或者问题的状态空间无穷大，那么沿着某条道路从根出发搜索叶节点，可能永远无法达到叶结点，因为搜索树会不断地扩展，然而叶结点也许是确实存在的，只要换一条道路就可以找到，只不过一开始就走错了路，而这条错路是永远无法终止的。为了避免这种情况我们一般都规定状态空间是有限的，这样即使搜索整个状态空间的每个状态也可以在有限时间内完成，否则的话回溯法很可能不适用。  
  
搜索树的每一个节点表示一个状态，节点i要生成节点j必须满足约束集D中的约束条件，我们也可以将这个约束条件称为“状态转移规则”或者“产生规则”（意指从节点i产生节点j的规则，这是从“产生式系统”理论的角度来解释回溯法）。因此回溯法的实质是在一个状态空间中，从起始状态（搜索树的根）搜索到一条到达目标状态（搜索树的叶结点）的路径（就和走迷宫差不多，这是从图论的角度来解释回溯法）。一般来说，为了防止搜索的过程中出现回路，必须记录已经走过的节点（状态），在同一条路径中不能重复走过的节点（状态），这样只要状态空间是有限的，回溯法总是可以终止的。  
  
===========================================================================================  
  
下面我们就根据回溯法来解决这个喝酒问题  
  
（1）状态的表示  
一个状态用一个7元组表示 X=（x1,x2,x3,x4,x5,x6,x7)；，其中x1~x3分别表示a,b,c三个酒瓶中的酒，x4~x7分别表示A,B,C，D四个人已经喝的酒；  
  
（2）约束条件  
1。每个人喝的酒不能超过4两；   
2。每个瓶中容纳的酒不能超过该瓶的容量；   
为了方便设第k个人喝的酒不超过C[k], 第i个酒瓶的容量为C*， 则  
C[1]=C[2]=8, C[3]=3, C[4]=C[5]=C[6]=C[7]=4;  
约束条件为  
 0<= X <= C;  
  
（3）状态的转移规则（状态产生规则）  
从某个状态X转移到另一个状态Y有以下几种情况：  
1。i瓶中的酒倒入j瓶中，并将j瓶装满:   Y = X - (C[j]-X[j]) ,  Y[j] = C[j],  i,j∈[1,3]  
2。i瓶中的酒倒入j瓶中，并将i瓶倒空:   Y = 0 ,   Y[j] = X[j] + X   ,  i,j∈[1,3]        
3。某个人j喝光了i瓶中的酒: Y = 0; Y[j] = X[j] +  X,    i∈[1,3], j∈[4,7]  
当然状态Y必须满足（2）中的约束条件；  
  
（4）初始状态  
a,b两个瓶中装满酒，c中为空：  X0[1]=C[1], X0[2]=C[2], X0[3]=C[3], X0[4]=X0[5]=X0[6]=X0[7]=0；  
  
（5）目标状态  
所有的瓶中的酒为空，每个人都喝饱了酒：    Xn[1]=Xn[2]=Xn[3]=0 , Xn[4]=C[4]，  Xn[5]=C[5]， Xn[6]=C[6]， Xn[7]=C[7];  
  
  
下面给出一个通用的回溯法伪代码：  
  
void DFS\_TRY( s )  
{  
  if (状态s是目标状态) {  
    打印结果；  
    退出；       // 如果要求输出所有的解，这里退出函数，如果只要求输出一组解，这里退出整个程序  
  }  
  for 从状态s根据产生规则产生的每个状态t   
    if (t不在堆栈中) {        
      状态t压入堆栈；  
      DFS\_TRY(t);  
      状态t弹出堆栈；  
    }  
}  
  
主程序为：  
  
初始状态s0压入堆栈；  
DFS\_TRY(s0);  
  
  
然而，对于这个问题，如果单纯地用上面的回溯法解决效率非常的低，几乎无法忍受。所以要改进一下。我们注意到每个状态是一个7元组，而且根据约束条件，所有的合法的状态的个数是8\*8\*3\*4\*4\*4\*4 =49152个，完全可以将所有的状态记录下来，即使穷举所有的状态也是可以忍受的。所以在上面的DFS\_TRY中，我们不是在堆栈中寻找已经搜索过的状态，而是在一个状态表中找已经搜索过的状态，如果某个状态在状态表中的标志表明该状态已经搜索过了，就没有必要再搜索一遍。比如，单纯的回溯法搜索出来的搜索树如下所示：  
  
             a  
     / \  
           /   \  
          b     c  
    \   /  
     \ /  
             d  
     / \  
           /   \  
  
从a出发，搜索 a - b - d - ... 然后回溯到a, 又搜索到 a - c - d - ..., 因为d在搜索的路径上并没有重复，所以在堆栈中是发现不了d节点被重复搜索的，这样就重复搜索了d和它的子树；如果用一个表格纪录每个节点是否被搜索过了，这样搜索 a - b - d - ...回溯到a, 又搜索到 a - c - d ，这时候查表发现d已经搜索过了，就可以不用再搜索d和它的子树了。  
  
这种用一个表格来记录状态的搜索策略叫做“备忘录法”，是动态规划的一种变形，关于动态规划和备忘录法，请参见:  
http://algorithm.myrice.com/algorithm/technique/dynamic\_programming/index.htm  
  
  
备忘录法的伪代码：  
  
bool Memoire\_TRY( s )  
{  
  if (状态s是目标状态) {  
    记录状态s；  
    return true;       // 这里假设只要求输出一组解  
  }  
  for 从状态s根据产生规则产生的每个状态t   
    if (状态t没有被搜索过) { // 注意这里的改变  
        标记状态t被访问过；  
 if (DFS\_TRY(t)) {  
     记录状态s；  
     return true;  
 }           
    }  
  return false;  
}  
  
主程序为：  
  
初始化设置状态表中的所有状态未被访问过  
初始状态设为s0；  
if (Memoire\_TRY(s0))  
  打印记录下来的解;  
  
这样就不需要自己设置堆栈了，但是需要维护一个状态访问表。  
  
下面是按照这种思路写的程序，注意，求出来的不是最优解，但是很容易修改该程序求出最优解。  
  
#include <iostream.h>  
#include <string.h>  
  
const int CUP\_COUNT   = 3;  // 酒杯的数目  
const int STATE\_COUNT = 7;  // 状态变量的维数  
typedef int State[STATE\_COUNT];     // 记录状态的类型  
const State CONSTR = {8, 8, 3, 4, 4, 4, 4}; // 约束条件  
const State START = {8, 8, 0, 0, 0, 0, 0}; // 初始状态  
const State GOAL = {0, 0, 0, 4, 4, 4, 4}; // 目标状态  
const int MAX\_STATE\_COUNT = 10\*10\*10\*10\*10\*10\*10; //态空间的状态数目  
const MAX\_STEP = 50;  // 假设最多需要50步就可以找到目标  
  
const State key = {3, 5, 3, 3, 2, 0, 0};  
  
bool visited[MAX\_STATE\_COUNT]; // 用来标记访问过的状态  
  
State result[MAX\_STEP];  // 记录结果；  
int step\_count = 0;  // 达到目标所用的步数  
  
  
// 计算状态s在状态表中的位置  
int pos(const State &s)  
{  
 int p = 0;  
 for (int i=0; i<STATE\_COUNT; i++) {  
  p = p\*10 + s;  
 }  
 return p;  
}  
  
// 判断状态a,b是否相等  
bool equal(const State &a, const State &b) {  
 for (int i=0; i<STATE\_COUNT; i++)  
  if (a!=b) return false;  
 return true;  
}  
  
void printState(const State &s) {  
 for (int i=0; i<STATE\_COUNT; i++)  
  cout << s << " ";  
 cout << endl;  
}  
  
// 备忘录法搜索  
bool Memoire\_TRY(const State &s, int step)  
{  
 if (memcmp(s,GOAL,sizeof(s))==0) {  // 如果是目标状态  
  step\_count = step;   
  memcpy(result[step-1],s, sizeof(s)); //  记录状态s  
  return true;  
 }  
   
 int i, j;  
  
 // 第一种规则，第i个人喝光杯子j中的酒  
 for (i=CUP\_COUNT; i<STATE\_COUNT; i++)   
  if (s < CONSTR)    // 如果第i个人还可以喝  
   for (j=0; j<CUP\_COUNT; j++)   
    if (s[j]>0 && s + s[j] <= CONSTR) {  // 如果第i个人可以喝光第j杯中的酒  
     State t;  
     memcpy(t, s, sizeof(s));  
     t += t[j];     // 第i个人喝光第j杯的酒  
     t[j] = 0;  
     int tmp = pos(t);  
     if (!visited[pos(t)]) {   // 如果状态t没有访问过  
      visited[pos(t)] =true;  // 标记状态t访问过了  
      if (Memoire\_TRY(t, step+1)) { // 从状态t出发搜索  
       memcpy(result[step-1],s, sizeof(s)); //  记录状态s  
       return true;  
      } // end of if (Memoire\_TRY(t, step+1))  
     } // end of if (!visited[pos(t)])  
    } // end of if (s + s[j] <= CONSTR)  
  
 // 第二种规则，将杯子i中的酒倒入到杯子j中去  
 for (i=0; i<CUP\_COUNT; i++)  
  for (j=0; j<CUP\_COUNT; j++)  
   if (i != j) {     
    int k = (CONSTR[j] - s[j] < s ? CONSTR[j] - s[j] : s ); // 计算出可以从i中倒入j中的酒的数量  
    if (k > 0) {  // 如果可以倒           
     State t;  // 生成新的状态t  
     memcpy(t, s, sizeof(s));  
     t -= k;  
     t[j] += k;   
     int tmp = pos(t);  
     if (!visited[pos(t)]) {  // 如果状态t没有访问过  
      visited[pos(t)] =true; // 标记状态t访问过了  
      if (Memoire\_TRY(t, step+1)) { // 从状态t出发搜索  
       memcpy(result[step-1],s, sizeof(s)); //  记录状态s  
       return true;  
      } // end of if (Memoire\_TRY(t, step+1))  
     } // end of if (!visited[pos(t)])   
    } // end of if (k > 0)  
   } // end of if (i != j)  
  
 return false;  
} // end of Memoire\_TRY  
  
   
  
void main()  
{  
 memset(visited, false, sizeof(visited));  
 if (Memoire\_TRY(START,1)) {  
  cout << "find a solution: " << endl;  
  for (int i=0; i<step\_count; i++) {  
   for (int j=0; j<STATE\_COUNT; j++)  
    cout << result[j] << " ";  
   cout << endl;  
  }  
 } else  
  cout << "no solution." << endl;  
}*

**鹰蛋**