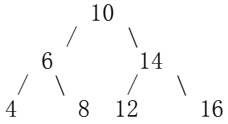
程序员面试题精选(01)－把二元查找树转变成排序的双向链表

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



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
        BSTreeNode  *m_pLeft;  // left child of node
        BSTreeNode  *m_pRight; // right child of node
    };
```

思路一对应的代码：

```
//////////////////////////////////////////////////////////////////////
// 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
//         else return the greatest node in the sub-tree
//////////////////////////////////////////////////////////////////////
BSTreeNode* ConvertNode(BSTreeNode* pNode, bool asRight)
{
      if(!pNode)
            return NULL;
      BSTreeNode *pLeft = NULL;
      BSTreeNode *pRight = NULL;

      // Convert the left sub-tree
      if(pNode->m_pLeft)
            pLeft = ConvertNode(pNode->m_pLeft, false);

      // Connect the greatest node in the left sub-tree to the current node
      if(pLeft)
      {
            pLeft->m_pRight = pNode;
            pNode->m_pLeft = pLeft;
      }
      // Convert the right sub-tree
      if(pNode->m_pRight)
            pRight = ConvertNode(pNode->m_pRight, true);
      // Connect the least node in the right sub-tree to the current node
      if(pRight)
      {
            pNode->m_pRight = pRight;
            pRight->m_pLeft = pNode;
      }

      BSTreeNode *pTemp = pNode;
      // If the current node is the right child of its parent,
      // return the least node in the tree whose root is the current node
      if(asRight)
      {
            while(pTemp->m_pLeft)
                  pTemp = pTemp->m_pLeft;
      }
      // If the current node is the left child of its parent,
      // return the greatest node in the tree whose root is the current node
      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
    return ConvertNode(pHeadOfTree, true);
}
```

思路二对应的代码：

```
//////////////////////////////////////////////////////////////////////
// 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
    if (pCurrent->m_pLeft != NULL)
        ConvertNode(pCurrent->m_pLeft, pLastNodeInList);

    // Put the current node into the double-linked list
    pCurrent->m_pLeft = pLastNodeInList;
    if(pLastNodeInList != NULL)
        pLastNodeInList->m_pRight = pCurrent;

    pLastNodeInList = pCurrent;

    // Convert the right sub-tree
    if (pCurrent->m_pRight != NULL)
        ConvertNode(pCurrent->m_pRight, pLastNodeInList);
}

//////////////////////////////////////////////////////////////////////
// Covert a binary search tree into a sorted double-linked list
// 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;
}
```

程序员面试题精选(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;               // the elements of stack
     size_t> m_minIndex;      // the indices of 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。

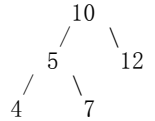
程序员面试题精选(03)－求子数组的最大和
题目：输入一个整形数组，数组里有正数也有负数。数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。求所有子数组的和的最大值。要求时间复杂度为O(n)。
例如输入的数组为1, -2, 3, 10, -4, 7, 2, -5，和最大的子数组为3, 10, -4, 7, 2，因此输出为该子数组的和18。
分析：本题最初为2005年浙江大学计算机系的考研题的最后一道程序设计题，在2006年里包括google在内的很多知名公司都把本题当作面试题。由于本题在网络中广为流传，本题也顺利成为2006年程序员面试题中经典中的经典。
如果不考虑时间复杂度，我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;
            // 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 > nGreatestSum)
                        nGreatestSum = pData;
            }
      }

      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。

程序员面试题精选(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



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 path from 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;
      path.pop_back();
}
```

程序员面试题精选(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(nlogn)。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+nlogk)。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);
                  }
            }
      }
}
```

程序员面试题精选(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 > 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);
}
```

程序员面试题精选(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;
}
```

程序员面试题精选(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^_^

程序员面试题精选(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。

程序员面试题精选(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？

程序员面试题精选(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);
	}
}
```

程序员面试题精选(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);
      }
}
```

程序员面试题精选(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
      const int tableSize = 256;
      unsigned int hashTable[tableSize];
      for(unsigned int i = 0; i < tableSize; ++ i)
            hashTable = 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;
}
```

程序员面试题精选(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。

程序员面试题精选(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 == ?)
                  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;
```

程序员面试题精选(16)－O(logn)求Fibonacci数列

题目：定义Fibonacci数列如下：

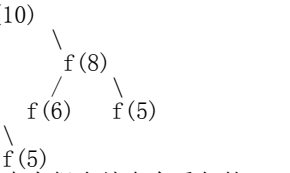
$$f(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ f(n-1)+f(n-2) & n=2\end{cases}$$

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



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;
}
```

程序员面试题精选(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;
	long long num = 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。

程序员面试题精选(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_stack2 is empty,
      // and there are some elements in m_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 the element 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？

程序员面试题精选(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。

程序员面试题精选(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）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 = (int*)new int[length2];

      for(i = 0; i < length1; ++ i)
            for(j = 0; j < length2; ++ j)
                  LCS_length[j] = 0;

      // initiate the direction matrix
      int **LCS_direction;
      LCS_direction = (int**)(new int[length1]);
      for( i = 0; i < length1; ++ i)
            LCS_direction = (int*)new int[length2];

      for(i = 0; i < length1; ++ i)
            for(j = 0; j < length2; ++ j)
                  LCS_direction[j] = kInit;

      for(i = 0; i < length1; ++ i)
```

```
	{
		for(j = 0; j < length2; ++ j)
		{
			if(i == 0 || j == 0)
			{
				if(pStr1 == pStr2[j])
				{
					LCS_length[j] = 1;
					LCS_direction[j] = kLeftUp;
				}
				else
					LCS_length[j] = 0;
			}
			// a char of LCS is found,
			// it comes from the left up entry in the direction matrix
			else if(pStr1 == pStr2[j])
			{
				LCS_length[j] = LCS_length[i - 1][j - 1] + 1;
				LCS_direction[j] = kLeftUp;
			}
			// it comes from the up entry in the direction matrix
			else if(LCS_length[i - 1][j] > LCS_length[j - 1])
			{
				LCS_length[j] = LCS_length[i - 1][j];
				LCS_direction[j] = kUp;
			}
			// it comes from the left entry in the direction matrix
			else
			{
				LCS_length[j] = LCS_length[j - 1];
				LCS_direction[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。

程序员面试题精选(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 --;
            }
        }
}
```

程序员面试题精选(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？

程序员面试题精选(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

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;
}
```

? 依次检查pop序列，当前栈顶不对就压栈，直到满足为止。如果push序列空了，就返回false
不知道我写的对不对。。。。

```
int good_order(int push[], int pop[], int size)
{
        int *tmp = (int *)malloc(size * sizeof(int));
        int top = 0, cur_push = 0, cur_pop = 0;
        tmp[top] = push[cur_push++];

        for(; cur_pop < size; cur_pop++) {
                while(cur_push < size && tmp[top] != pop[cur_pop])
                        tmp[++top] = push[cur_push++];
                if(tmp[top] == pop[cur_pop])
                        top--;
                else{
                        free(tmp);
                        tmp = NULL;
                        return 0;
                }
        }
        free(tmp);
        return 1;
}
```

程序员面试题精选100题(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;
```
程序员面试题精选100题(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;
}
```

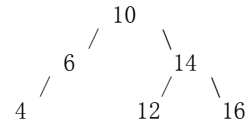
程序员面试题精选100题(26)-和为n连续正数序列
题目：输入一个正数n，输出所有和为n连续正数序列。
例如输入15，由于1+2+3+4+5=4+5+6=7+8=15，所以输出3个连续序列1-5、4-6和7-8。
分析：这是网易的一道面试题。
这道题和本面试题系列的第10题有些类似。我们用两个数small和big分别表示序列的最小值和最大值。首先把small初始化为1，big初始化为2。如果从small到big的序列的和大于n的话，我们向右移动small，相当于从序列中去掉较小的数字。如果从small到big的序列的和小于n的话，我们向右移动big，相当于向序列中添加big的下一个数字。一直到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");
}
```

程序员面试题精选100题(27)-二元树的深度

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



输出该树的深度3。
二元树的结点定义如下：

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

分析：这道题本质上还是考查二元树的遍历。
题目给出了一种树的深度的定义。当然，我们可以按照这种定义去得到树的所有路径，也就能得到最长路径以及它的长度。只是这种思路用来写程序有点麻烦。
我们还可以从另外一个角度来理解树的深度。如果一棵树只有一个结点，它的深度为1。如果根结点只有左子树而没有右子树，那么树的深度应该是其左子树的深度加1；同样如果根结点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加1。如果既有右子树又有左子树呢？那该树的深度就是其左、右子树深度的较大值再加1。
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);
```

程序员面试题精选100题(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。

程序员面试题精选100题(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。

程序员面试题精选100题(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。

程序员面试题精选100题(31)-从尾到头输出链表
题目：输入一个链表的头结点，从尾到头反过来输出每个结点的值。链表结点定义如下：

```
struct ListNode

{

      int       m_nKey;

      ListNode* m_pNext;

};
```

分析：这是一道很有意思的面试题。该题以及它的变体经常出现在各大公司的面试、笔试题中。

看到这道题后，第一反应是从头到尾输出比较简单。于是很自然地想到把链表中链接结点的指针反转过来，改变链表的方向。然后就可以从头到尾输出了。反转链表的算法详见本人面试题精选系列的第19题，在此不再细述。但该方法需要额外的操作，应该还有更好的方法。

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

既然想到了栈来实现这个函数，而递归本质上就是一个栈结构。于是很自然的又想到了用递归来实现。要实现反过来输出链表，我们每访问到一个结点的时候，先递归输出它后面的结点，再输出该结点自身，这样链表的输出结果就反过来了。

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。

程序员面试题精选100题(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。
程序员面试题精选100题(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。

程序员面试题精选100题(35)-找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;
}
```

程序员面试题精选（35）：一次遍历链表求中间节点位置
思路：声明两指针p和q，p每往后移动两节点，q往后移动一个节点，当p->next==NULL时，q便是中间节点的位置。
知道思路后，实现就比较简单了，在此不给出代码。

程序员面试题精选100题(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 。
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';
}
```

程序员面试题精选（36）：找出数组中唯一的重复元素
1-1000放在含有1001个元素的数组中，只有唯一的一个元素值重复，其它均只出现一次。每个数组元素只能访问一次，设计一个算法，将它找出来；不用辅助存储空间，能否设计一个算法实现？
将1001个元素相加减去1,2,3,……1000数列的和，得到的差即为重复的元素。

```
  int   Find(int   *   a)
  {
  int   i;//变量
  for   (i   =   0   ;i<=1000;i++)
  {
  a[1000]   +=   a[i];
  }
  a[1000]   -=   (i*(i-1))/2       //i的值为1001
  return   a[1000];
  }
```

利用下标与单元中所存储的内容之间的特殊关系，进行遍历访问单元，一旦访问过的单元赋予一个标记，利用标记作为发现重复数字的关键。代码如下：

```
void FindRepeat(int array[], int length)
{
    int index=array[length-1]-1;
    while ( true )
    {
       if ( array[index]<0 )
           break;
       array[index]*=-1;
       index=array[index]*(-1)-1;
    }

    cout<<"The repeat number is "<<index+1<<endl;
}
```

此种方法不非常的不错，而且它具有可扩展性。在坛子上有人提出：
对于一个既定的自然数 N ，有一个 N + M 个元素的数组，其中存放了小于等于 N 的所有自然数，求重复出现的自然数序列{X} 。

对于这个扩展需要，自己在A_B_C_ABC(黄瓜儿才起蒂蒂)的算法的基础上得到了自己的算法
代码：
按照A_B_C_ABC(黄瓜儿才起蒂蒂)的算法，易经标记过的单元在后面一定不会再访问到，除非它是重复的数字，也就是说只要每次将重复数字中的一个改为靠近N+M的自然数，让遍历能访问到数组后面的单元，就能将整个数组遍历完。
代码：

```
*/
void FindRepeat(int array[], int length, int num)
{
 int index=array[length-1]-1;
 cout<<"The repeat number is ";
 while ( true )
 {
 if ( array[index]<0 )
 {
   num--;
   array[index]=length-num;
   cout<<index+1<<'t';
 }

 if ( num==0 )
 {
   cout<<endl;
  return;
 }
 array[index]*=-1;
 index=array[index]*(-1)-1;
 }
}
```

程序员面试题精选（37）：判断字符串是否是回文字符串或者是否含有回文字符子串
题目来自BMY BBS算法版，原题如下：
不仅能判断规则的中心对称,如123454321,还要能判断如123456547890中的45654的不规则部分中心对称
算法思想
从第一个字符开始，逐个扫描，对每一个字符，查找下一个相同字符，判断这两个字符之间的字符串是否回文。时间复杂度O(n^3)，所以说是笨笨解，师弟说可以做到O(n^2)...
算法实现

```
/*==========================================================================
功能：判断字符串是否是回文字符串或者是否含有回文字符子串
作者：sunnyrain
日期：2008-09-11
编译环境：VC++6.0
```

```
==========================================================================
#include<iostream>
using namespace std;
int find(char ch,char *str,size_t length) //查找str开始的字符串中第一个ch出现的位置,没找到返回-1
{
        for(int i=0;i<length;i++)
        {
                if(ch == str[i])
                        return i;
        }
        return -1;
}
int isSym(char *str,size_t length)  //判断一个字符串是否全对称,0对称，-1不对称
{
        for(int i=0;i<length/2;i++)
        {
                if(str[i] != str[length-i-1])
                        return -1;
        }
        return 0;
}
int isSymmetry(char *str,size_t length)//返回0表示全部对称，-1表示无对称，其他值表示局部对称起始位置
{
        int begin=0,end;
        char ch;
        for(int i=0;i<length;i++)
        {
                ch = str[i];
                begin = end = i;
                while((begin = find(ch,str+end+1,length-end-1)) != -1)
                {
                        end = begin+end+1;
                        if(isSym(str+i,end-i+1) == 0)
                                return i;
                }
        }
        return -1;
}
int main()
{
        char aa1[] = "123454321";
        char aa2[] = "123456547890";
        char aa3[] = "781234327891";
        char aa4[] = "954612313217891";
        cout<<isSymmetry(aa1,sizeof(aa1)/sizeof(char)-1)<<endl;
        cout<<isSymmetry(aa2,sizeof(aa2)/sizeof(char)-1)<<endl;
        cout<<isSymmetry(aa3,sizeof(aa3)/sizeof(char)-1)<<endl;
        cout<<isSymmetry(aa4,sizeof(aa4)/sizeof(char)-1)<<endl;
        return 0;
}
```

程序员面试题精选（38）：2008百度校园招聘的一道笔试题

题目大意如下：

一排N（最大1M）个正整数+1递增，乱序排列，第一个不是最小的，把它换成-1，最小数为a且未知求第一个被-1替换掉的数原来的值，并分析算法复杂度。

解题思路：

一般稍微有点算法知识的人想想就会很容易给出以下解法：

设 Sn = a + (a+1) + (a+2) + ...+ (a+n-1) = na +n(n-1)/2

扫一次数组即可找到最小值a，时间复杂度O(n)

设 S = 修改第一项后所有数组项之和, 求和复杂度为O(n)

则被替换掉的第一项为 a1=Sn-S-1

总的时间复杂度为 O(1)+O(n)+O(n) = O(n)

根据该算法写出程序很简单，就不写了

主要是解题过程中没有太考虑题目中给的1M这个数字，一面的时候被问到求和溢出怎么办？

当时我一想，如果要考虑溢出，必然是要处理大数问题，以前没有看到大数就头疼……所以立马想了个绕过大数加法的方法，如下：

设定另外一个数组b[N]

用 a, a+1,a+2....a+n-1依次分别减去原数组，得到的差放在该数组里，此求差过程复杂度为O(n)

对该数组各项求和即可得到Sn-S

面试官让证明一下我的设想，当时还没有给我纸和笔，用手在桌子上比划了一下没想出来，回来躺在床上想了一会就想出来了，也没什么难度：

相减求和后的数组，最差情况下应该是连续n/2个负数或者正数相加，如果不溢出，后面正负混合相加的话肯定不会溢出；这种情况下的最差特殊情况就 是，原数列按照降序排列（除了第一项被替换掉了），而我们减时所用数列是增序排列。所得结果将是1个正数，n/2-1个负数，n/2个正数；而且我们相当 于用最大的n/2个数减去最小的n/2个数，差值之和最大，取到了最差情况，我们只考虑后面一半求和的情况即可（前面有个-1不方便处理）：

S(n/2) = (n-1) + (n-3) + (n-5)+ .....+ 1　　(n为奇数时最后一项是0，不影响我们讨论数量级计算溢出)
　　= [(n-1)+1] * n/4 = n^2/4
题目中给定n最大为1M = 1024*1024
那么S(n/2)的最大量级为1024^4 = 2^40
而long long类型为64位，可以存放下该和，成功避免大数问题。
直接求和办法，一是和可能溢出，二是面试官要求把原始数组改称long long的话(即a可以也可能很大，求和时稍微加一下就会溢出)就得考虑大数求解了；而这种差值办法可以直接消掉a，求和只和n相关，和a无关。

程序员面试题精选（39）：一道autodesk笔试题求解

去年5月参加Autodesk实习生招聘时的算法题，当时时间紧，题量相当大，没有来得及做这道题，之后也因为懒一直没有做，小崔的答案发给我也没看，今天突然想起来，就做了一下，比想像的要简单一些：）
```
/*==========================================================================
题目：给定一个M×N矩阵，从左上角元素开始，按照顺时针螺旋状打印所有矩阵元素
关键点：遍历过程的方向控制和左右界控制、上下界控制
作者：sunnyrain
日期：2007.9.3下午
运行环境：vc++ 6.0
==========================================================================*/
#include<iostream>
#include<vector>
using namespace std;
//定义方向
enum Direction{ToRight,Down,ToLeft,Up};
//改变方向函数
void turnDirect(Direction& cur)
{
 switch(cur)
 {
 case ToRight:
  cur = Down;
  break;
 case Down:
  cur = ToLeft;
  break;
 case ToLeft:
  cur = Up;
  break;
 case Up:
  cur = ToRight;
  break;
 }
}
//遍历矩阵
void traverse(vector<vector<int> >& vv)
{
 int m = vv[0].size();
 int n = vv.size();
 int up=0,down=n,left=0,right=m;
 int i=0,j=0;
 Direction curD = ToRight;
 for(;up !=down || left != right;)
 {
  switch(curD)
  {
  case ToRight: //从左向右遍历
   for(j=left;j<right;j++)
    cout<<vv[i][j]<<" ";
   cout<<endl;
   --j;     //列坐标出界回退
   up++;    //遍历完一行上界下移
   turnDirect(curD);  //改变遍历方向
   break;
  case Down:   //从上向下遍历
   for(i=up;i<down;i++)
    cout<<vv[i][j]<<" ";
   cout<<endl;
   --i;     //出界回退
   right--; //遍历完一列右界左移
   turnDirect(curD);
   break;
  case ToLeft:  //从右向左遍历
   for(j=right-1;j>=left;--j)
    cout<<vv[i][j]<<" ";
   cout<<endl;
```

```
   ++j;      //出界回退
   down--;   //下界上移
   turnDirect(curD);
   break;
  case Up:      //从下向上遍历
   for(i=down-1;i>=up;--i)
    cout<<vv[i][j]<<" ";
   cout<<endl;
   ++i;      //出界回退
   left++;   //左界右移
   turnDirect(curD);
   break;
  }
 }
}
int main()
{
 int m,n,i,j,k=0;
 cout<<"please input the m and n of the matrix(m*n):"<<endl;
 cin>>m>>n;
 vector<vector<int> > vec(n);

 //初始化矩阵
 for(i=0;i<n;i++)
  for(j=0;j<m;j++)
   vec[i].push_back(++k);
    //正常遍历矩阵
 for(i=0;i<n;i++)
 {
  for(j=0;j<m;j++)
   cout<<vec[i][j]<<"\t";
  cout<<endl;
 }
 //顺时针螺旋遍历矩阵
 traverse(vec);
 return 0;
}
```

程序员面试题精选（40）：一道SPSS笔试题求解

```
/*==============================================================================
题目：输入四个点的坐标，求证四个点是不是一个矩形
关键点：
1.相邻两边斜率之积等于-1，
2.矩形边与坐标系平行的情况下，斜率无穷大不能用积判断。
3.输入四点可能不按顺序，需要对四点排序
作者：sunnyrain
日期：2007.9.2 & 2007.9.3
运行环境：vc++ 6.0
==============================================================================*/
#include<iostream>
#include<limits>
using namespace std;
const double MAX = numeric_limits<double>::max();  //斜率最大值
class Point
{
 float x;
 float y;
public:
 Point(float _x = 0, float _y = 0):x(_x),y(_y){}
 float getX() const
 {
  return x;
 }
 float getY() const
 {
  return y;
 }
 //为点重新设置坐标
 void set(float _x,float _y)
 {
  x = _x;
  y = _y;
 }
 //重载 == 成员操作符
```

```
 bool operator == (Point& p)
 {
  return (this->x == p.x && this->y == p.y);
 }
 //重载流插入操作符
 friend istream & operator >> (istream& is, Point & p)
 {
  return is>>p.x>>p.y;
 }
};
class Line  //两点形成一条直线/线段
{
 Point start;
 Point end;
 double k;  //斜率
public:
 Line(){}
 Line(Point s,Point e):start(s),end(e)
 {
  if(start.getX() - end.getX() != 0)
   k = (start.getY() - end.getY())/(start.getX() - end.getX()) ;
  else
   k = MAX;  //两点x坐标相等则斜率无穷大
 }
 double getK()
 {
  return k;
 }
};
//查找数组pp中是否存在点p，是返回数组序号，否返回-1
int findPoint(Point *pp,int size,Point &p)
{
 for(int i=0;i<size;i++)
 {
  if(pp[i] == p)
   return i;
 }
 return -1;
}
//主函数
int main()
{
 Point p[4];
 Point *s[4];
 int i;
 for(i=0; i<4; i++)
 {
  cout<<"Please input the coordinates of the "<<i+1<<" point in format \"x.xx y.yy\""<<endl;
  cin>>p[i];
 }

/* p[0].set(0,0);
 p[1].set(1,1);
 p[2].set(2,0);
 p[3].set(1,-1);*/
 float left,up,right,down;

 //获取四点x坐标最大值和最小值
 left = p[0].getX(); //left为四点x坐标最小值
 right = p[0].getX(); //right为四点x坐标最大值
 for(i=1;i<4;i++)
 {
  if(left>p[i].getX())
   left = p[i].getX();
  if(right<p[i].getX())
   right = p[i].getX();
 }
 //获取四点y坐标最大值和最小值
 up = p[0].getY(); //up为四点y坐标最大值
 down = p[0].getY(); //四点y坐标最小值
 for(i=1;i<4;i++)
 {
  if(up<p[i].getY())
   up = p[i].getY();
  if(down > p[i].getY())
```

```
   down = p[i].getY();
 }
 //判断矩形与坐标系平行情况
 Point P1(left,up),P2(right,up),P3(right,down),P4(left,down);
 if(findPoint(p,4,P1) != -1 && findPoint(p,4,P2) != -1 && findPoint(p,4,P3) != -1 && findPoint(p,4,P4) !=
-1)
 {
  cout<<"是矩形"<<endl;
  return 0;
 }
 //按照顺时针方向对四点排序
 for(i=0;i<4;i++)
 {
  if(p[i].getX() == left)
   s[0] = &p[i];
  else if(p[i].getY() == up)
   s[1] = &p[i];
  else if(p[i].getX() == right)
   s[2] = &p[i];
  else if(p[i].getY() == down)
   s[3] = &p[i];
 }
 //排序后的四点顺时针相连组成矩形边
 Line one(*s[0],*s[1]),two(*s[1],*s[2]),three(*s[2],*s[3]),four(*s[3],*s[0]);
 cout<<"k1 = "<<one.getK()<<endl;
 cout<<"k2 = "<<two.getK()<<endl;
 cout<<"k3 = "<<three.getK()<<endl;
 cout<<"k4 = "<<four.getK()<<endl;
 //判断相邻边斜率之积是否都等于0
 if(one.getK()*two.getK() == -1 || (one.getK() == 0 && two.getK() == MAX) || (one.getK() == MAX &&
two.getK() == 0) )
  if(two.getK()*three.getK() == -1 || (two.getK() == 0 && three.getK() == MAX) || (two.getK() == MAX &&
three.getK() == 0) )
   if(three.getK()*four.getK() == -1 || (three.getK() == 0 && four.getK() == MAX) || (three.getK() == MAX
&& four.getK() == 0) )
    if(four.getK()*one.getK() == -1 || (four.getK() == 0 && one.getK() == MAX) || (four.getK() == MAX &&
one.getK() == 0) )
    {
     cout<<"是矩形!"<<endl;
     return 0;
    }
 cout<<"不是矩形"<<endl;
 return 0;
}
```

程序员面试题精选（41）：编译器对内存填充长度之误解

看了《C++ 对像模型》的人，往往会误以为编译器填充是按照计算机字长填充的，如下：

```
class A
{
   double a;
   char b;
};
sizeof(A) == ?
```

不了解填充的人会以为是9,看了c++对象模型的（像我）往往会以为是12，昨晚看《程序员面试宝典》一道类似题，开始以为答案给错了。。今天一试才知道，原来我错了。。上题答案（在编译器默认情况下）是 16，VC6.0、MinGW、VS.net均如此。。

《程序员面试宝典》上如是说：CPU的优化原则大致是这样的：对于n字节的元素（n=2、4、8……）它的首地址能被n整除，才能获得最好的性能。设计编译器的时候可以遵循这个原则。也就是说，默认情况下，编译器往往以最大的变量的长度为填充长度，而不是按字节长度。当然也可以通过 #pragma pack(n) 指定编译器的填充长度。这时候应该不是cpu的效率最高的情况了。

另外有个网友讨论说道如果一个类中含有另一个类对象，是否按照包含类的长度填充呢？试验了一下，不是这样，而是按照语言中的基本类型的最大长度填充。没想到，面试题中也会考到这么bt的题目，长见识了。

程序员面试题精选（42）：约瑟夫问题的数学方法

问题描述：N个人围成圆圈，从1开始报数，到第M个人令其出列，然后下一个人继续从1开始报数，到第M个人令其出列，如此下去，直到只剩一个人为止。显示最后一个人为剩者。

无论是用链表实现还是用数组实现都有一个共同点：要模拟整个游戏过程，不仅程序写起来比较烦，而且时间复杂
度高达O(nm)，当n，m非常大(例如上百万，上千万)的时候，几乎是没有办法在短时间内出结果的。

为了讨论方便，先把问题稍微改变一下，并不影响原意：

问题描述：n个人（编号0~(n-1)），从0开始报数，报到(m-1)的退出，剩下的人继续从0开始报数。求胜利者的编号。

我们知道第一个人(编号一定是m%n-1) 出列之后，剩下的n-1个人组成了一个新的约瑟夫环（以编号为k=m%n的人开
始）：
  k k+1 k+2 ... n-2, n-1, 0, 1, 2, ... k-2
并且从k开始报0。

现在我们把他们的编号做一下转换：
k   --> 0
k+1   --> 1
k+2   --> 2
...
...
k-2   --> n-2
k-1   --> n-1

变换后就完完全全成为了(n-1)个人报数的子问题，假如我们知道这个子问题的解：例如x是最终的胜利者，那么根
据上面这个表把这个x变回去不刚好就是n个人情况的解吗？！！变回去的公式很简单，相信大家都可以推出来：x'
=(x+k)%n

如何知道(n-1)个人报数的问题的解？对，只要知道(n-2)个人的解就行了。(n-2)个人的解呢？当然是先求(n-3)的
情况 ---- 这显然就是一个倒推问题！好了，思路出来了，下面写递推公式：

令f[i]表示i个人玩游戏报m退出最后胜利者的编号，最后的结果自然是f[n]

递推公式
f[1]=0;
f[i]=(f[i-1]+m)%i; (i>1)

有了这个公式，我们要做的就是从1-n顺序算出f[i]的数值，最后结果是f[n]。因为实际生活中编号总是从1开始，
我们输出f[n]+1

由于是逐级递推，不需要保存每个f[i]，程序也是异常简单：

```
#include <stdio.h>

main()
{
  int n, m, i, s=0;
  printf ("N M = "); scanf("%d%d", &n, &m);
  for (i=2; i<=n; i++) s=(s+m)%i;
  printf ("The winner is %d\n", s+1);
}
```

这个算法的时间复杂度为O(n)，相对于模拟算法已经有了很大的提高。算n，m等于一百万，一千万的情况不是问题
了。

程序员面试题精选（43）：数组中连续元素相加和最小的元素序列
题目描述： 有一个集合{14,56,53,4,-9,34,...n}里面共n个数
  里面可以有负数也可以没有
  用一个时间复杂度为o(n)的算法找出其中的一个连续串象(53,4,-9) 这样(串里的数字个数任意)
  使得这个连续串为所有这样连续串里各个数字相加和最小的一个
代码实现如下（程序没有考虑有多组解的情况）

```
#include <iostream>
using namespace std;
template<typename T>
int getMinSum(T* a,int n,T* pbegin,T* pend)
{
 T   min   =   a[0];
 T   sum   =   a[0];
 T   tempbegin   =   0;
 *pbegin   =   0;
 *pend   =   0;
 for   (int   i   =   1;   i   <   n;   i++)
 {
  if(sum   <   0)
   sum   =   sum   +   a[i];
  else
  {
   tempbegin   =   i;
   sum   =   a[i];
```

```
  }
  if   (sum   <   min)
  {
   min   =   sum;
   *pbegin   =   tempbegin;
   *pend   =   i;
  }
 }
 return   min;
}
int main()
{
 int   a[]   =   {8,   9,   -3   ,-10   ,7   ,0   ,8   ,-12,   9,   8   ,-1   ,-2   ,9};

 int   begin;
 int   end;
 int   sum;
 int   k   =   sizeof(a)   /   sizeof(int);
 sum=getMinSum(a,k,&begin,&end);
 cout<<"The   min   sum   is "<<sum<<endl;
 cout<<"And   the   begin   is   "<<begin<<",and   the   end   is   "<<end<<endl;

 return   0;
}
```

程序员面试题精选（44）：整数分割（即求一个数N由小于等于N的数相加所得的所有组合）

题目描述：比如给定一整数4，其有如下情况：4=4；

4=3+1；

4=2+2；

4=2+1+1；

4=1+1+1+1；

下面便是两种版本的分割实现代码。

```
#include "stdio.h"
 int  Compute( int  number,  int  maximum)
  {
     if  (number  ==   1   ||  maximum  ==   1 )
         return   1 ;
     else   if  (number  <  maximum)
         return  Compute(number, number);
     else   if  (number  ==  maximum)
         return   1   +  Compute(number, maximum  -   1 );
     else
         return  Compute(number, maximum  -   1 )  +  Compute(number  -  maximum, maximum);
 }

 int  IntPartionNo( int  n)//////////////////求组合总数版本；
 {
     return  Compute(n, n);
}
 int  IntegerPartition( int  n)///////////////求组合总数并打印出所有情况版本；
 {
     int   * partition  =   new   int [n]();
     int   * repeat  =   new   int [n]();

  partition[ 1 ]  =  n;
  repeat[ 1 ]  =   1 ;
     int  count  =   1 ;
     int  part  =   1 ;

     int  last, smaller, remainder;

  printf( " %3d " , partition[ 1 ]);
     do
  {
   last  =  (partition[part]  ==   1 )  ?  (repeat[part -- ]  +   1 ) :  1 ;
   smaller  =  partition[part]  -   1 ;
         if  (repeat[part]  !=   1 )
             -- repeat[part ++ ];
   partition[part]  =  smaller;
   repeat[part]  =   1   +  last  /  smaller;

         if  ((remainder  =  last  %  smaller)  !=   0 )
   {
    partition[ ++ part]  =  remainder;
    repeat[part]  =   1 ;
   }
```

```
        ++ count;

   printf( " \n " );
        for  ( int  i  =   1 ; i  <=  part;  ++ i)
            for  ( int  j  =   1 ; j  <=  repeat[i];  ++ j)
     printf( " %3d " , partition[i]);

  }   while (repeat[part]  !=  n);

     if  (partition)
  {
   delete [] partition;
   partition  =   0 ;
  }
     if  (repeat)
  {
   delete [] repeat;
   repeat  =   0 ;
  }

     return  count;
}
int main()
{
 printf("%d\n",IntPartionNo(4));
 IntegerPartition(4);
 getchar();
}
```

程序员面试题精选（45）：求给定整数其二进制形式含1的个数

题目描述：求给定整数其二进制形式含1的个数，比如255含8个1，因其二进制表示为11111111；

下面给出了两种求解代码实现。

```
#include <iostream>
using namespace std;
int  CountOne( int  n)
{
 int  count  =   0 ;
 while  (n)
 {
  ++ count;
       n  &=  n  -   1 ;
    }

 return  count;
}
int  CountOnesUsingTable(unsigned  int  i)
{
 static   int  BIT_TABLES[ 256 ]  =
 {
  0 , 1 , 1 , 2 , 1 , 2 , 2 , 3 , 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4
   , 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5
   , 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5
   , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6
   , 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5
   , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6
   , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6
   , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7
   , 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5
   , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6
   , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6
   , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7
   , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6
   , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7
   , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7
   , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7 , 5 , 6 , 6 , 7 , 6 , 7 , 7 , 8
    } ;

 return  BIT_TABLES[i  &   255 ]  +  BIT_TABLES[i >> 8   &   255 ]  +

        BIT_TABLES[i >> 16   &   255 ]  +  BIT_TABLES[i >> 24   &   255 ];
}
int main()
{
 cout<<"158 has : "<<CountOne(158)<<endl;
 cout<<"158 has : "<<CountOnesUsingTable(158)<<endl;
```

```
 getchar();
}
```
打印内存小技巧收藏
新一篇: 程序员面试题精选（46）：矩阵式螺旋输出 | 旧一篇: 程序员面试题精选（45）：求给定整数其二进制形式含1的个数
在学习C++的时候，由于编译器背着我们干了太多的事儿，所以看看那些高级数据结构在汇编级别是怎么样的，在内存中是如何的，对编写高效代码很有帮助。
下边是一个小函数，帮你打印内存中的内容。如果使用微软的编译器，各个内存对象之间可能会有byte guard，即编译器会在分配的数据对象之间插入空白bytes，便于检测破坏邻接对象，所以和教科书上的连续分配内存有些差异，注意一下就行了
以下是实现加测试代码：
```
#include  < cstdio >
struct Test{
 int a;
 char b;
};
void  ShowBytes( void *  s,  int  n)
{
    unsigned  char *  start  =  (unsigned  char * )s;

    printf( " [OFFSET] ADDRESS: VALUE\n\n " );

 for  ( int  i  =  0 ; i  <  n; i ++ )
 {
       printf( "  [%.4d] %.8X: %.2X\n " , i, start  +  i,  * (start  +  i));

  if  ((i  +  1 )  %  4  ==  0 )
  {
           printf( " ----------------------\n " );
       }
   } // for
 }
int main()
{
 Test *t;
 Test a={12,'A'};
 t=&a;
 ShowBytes(t,8);
 getchar();
 return 0;
}
```
程序员面试题精选（53）：删除链表结点（时间复杂度为0（1））
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。

程序员面试题精选（54）：找出数组中两个只出现一次的数字收藏

新一篇: 部分it公司的笔试小算法题精选 | 旧一篇: 程序员面试题精选（53）：删除链表结点（时间复杂度为0（1））

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。

代码如下：

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

}
//////////////////////////////////////////////////////////////////////
// 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];
      }
}

int main()
{
 int a[8]={2,3,6,8,3,2,7,7};
 int x,y;
 FindNumsAppearOnce(a,8,x,y);
 cout<<x<<"\t"<<y;
 getchar();
}
```

一道百度面试题题解

问： A厂有1万个工人，编号0-9999，(EE[10000])，1个厂长(GG)分派任务，1个监工(MM)管理工人.厂子忙的时间不确定,可能突然很忙,1天接到任务5000多个,1个任务只能分配给1个工人做，也可能好几十天没新任务.厂长分配任务给这1万个工人干，按工人编号一个一个来，到最后一个工人就又从头开始，任务完成时间各不相同，可能一个工人在分配任务的时候手里还有任务， 就得换下一个。

但是这1万个工人都很懒，领到了任务先不做，需要监工1个1个去问，如果工人有任务，就 做，如果工人没任务，则不做。厂长只管分任务，1个1个来，可能几天也没新任务，不累; 但是监工很累，监工每天都要看所有工人的情况，即使这些工人都没有任务，实际上每天工人(80%左右)是没任务的，请问，怎么让监工的工作轻松下来. 比如说每天只问1小半工人.

Peak Wong:

分析如下：

因为“任务完成时间各不相同” ，所以有可能a，b，c某天都有任务，但b的任务最先完成，那么当b的任务完成后，有任务的人的工号可能是不连续的;

用一个数组表示1万个工人是否有任务，并保存最后被分配任务的人的工号;

1)从前一天“最后被分配任务的人的工号”开始，依次问下一个工号的人，置对应的工作状态，直到碰到前一天无工作，且当天也无工作的人; 并更新当步最后有工作的人的工号为当天的“最后被分配任务的人的工号”;

2)从前一天“最后被分配任务的人的工号”开始，依次问上一个工号且前一天有工作的人;

问题是监工可以知道那些信息，否则还不是一个一个接着去问。
还有就是tailzhou的步骤1消耗的时间T1， 工人完成的时间T2,如果T2

所以很多条件都没有限制。

可是仔细想想，就是监工要记录所有工人的工作状态，然后每天只查询在工作的工人就可以了。(并且记录谁还在工作中)

其实最根本的解决之道是在每次厂长分配任务时，监工也被通知谁被分配了任务。而现在题目的假设是厂长太忙了，来不及通知监工(其实让监工分配就行了)。

解决这个问题很简单，监工只要记录(也可能是他猜测)上次厂长分配任务最后分配到那个人就可以了。

然后每天查询时，除了监督前一天来在工作的人外，还要查看从上次分配到任务的编号的下一个编号开始的工人，向后依次查询，知道遇到一个没有被分配工作的 人(这个工人后面不会再有人被分配任务了).同时监工还要记录或猜测哪个工人是当天最后被分配任务的。在“查看从上次分配到任务的编号的下一个编号开始的 工人，向后依次查询”过程中，如果工人的话可信，询问他们就可以了。如果不可信，那么可以猜测为最后一个昨天空闲，今天忙的工人就可以了。

其实，监工既然可以监督，他总得有渠道知道当天那个工人还有活要做(不然所有工人都可以说今天我没有任务呀)，所以没有这么复杂的问题。

解决问题：

有一问：监工是不是问了以后，工人就会一直把工作做完?

如果这样的话，最简单的一个考虑，他第一次问的时候记住这个工人今天能否做完，不能做完的话，哪天才能做完。

监工实际上只需要在工人做完了以后的第二天去问就可以了。因为不做完，厂长不会给他分新任务。

但问题说：但是监工很累，监工每天都要看所有工人的情况，即使这些工人都没有任务那又不一样了，就是说监工必须每天问，不然工人不会开始工作。

这时候就是没有工作的工人不需要问。

首先工作队列，每个人问一遍，今天能做完的移到空闲队列。

空闲队列，二分查找，找到非空闲的，往前处理完，今天能做完的不动，不能做完的移到工作队列。

很简单的问题阿。二分查找需要问的人非常少的。

绝对能满足楼主说的：比如说每天只问1小半工人，用我的方法厂长分任务的时候不需要通知监工。

实际上通知也没有意义，因为是按顺序分配，监工实际上需要知道的是今天有多少新任务。

就算不知道，用二分查找10000人也最多需要10几次。

这10几次，再根据实际上每天工人(80%左右)是没任务的，所以实际上每天问的人只有20%左右。

1.问题分析

现在的情况是监工每天要查看所有的工人，催他们工作，因为不催他们不开工，即要访问EE[10000]的每个元素一次。目标是每天只问一小半的工人，实 际上没有工作的工人是不需要问的，最理想的情况就是监工只问有工作的工人，或者尽可能少地问没有工作的工人，即要尽可能少地访问EE[10000]的元 素。

怎么办呢?监工想了一个办法，他做了一万张卡片，每张卡片上写着工人的编号，从0-9999，恰好和数组EE[10000]的下标对应。

监工拿着他的秘密武器上阵了，0号，有工作没?没有。好，放右边口袋。1号，有工作没?有。今天能干完吗?能。好，放右边口袋(并且放在0号后面)。2 号，有工作没?有，今天能干完吗?不能。嗯，放左边。3号，没有。放右边(1号后面)。4号，今天干不完，放左边。……第二天，先看右边(昨天没事的)， 0号，有工作没，有，今天能干完吗，能，好，不动。1号有工作没，有，干不完，好，放左边(接着昨天后面放)。3号，没有，哦，厂长GG还没有分配到这里 阿，

那明天检查空的从这里开始就可以了(记住)，但今天仍然轻松不了，因为可能是从后面的号码过来，并且分配到前面来了。右边全部查完了。再查左边。2 号，今天能干完，放右边(并且放在0号后面)。阿，又碰到了1号了，今天的检查结束。第三天，总算可以轻松了。从3号开始，先查右边。今天做得完，不动， 做不完，移到左边后面，碰到没有任务的，或者碰到3号，右边检查完。再看左边，做得完的，移到右边，并且按顺序插入其它卡片中间，做不完的，不动，直到碰 到今天新加入的做不完的，或者整个左边的卡片都检查完。

说了这么多，实际上就是左边的是工作的队列，右边的是没工作的队列，左边和右 边的区别就是右边的要保持按编号排序(因为厂长GG分配任务是按顺序分的)。再拿支铅笔，在有的卡片上做做记号。工作轻松不少。以上都是假设工人必须每天 催才会工作，并且监工每天都是在厂长分配任务之后才去催。如果工人催一次就会一直工作，那么简单，监工只需要在卡片上记下还要几天才去问就行了。如果监工 是每天早上去催，厂长去可能在清早(问之前)或下午(问之后，但工人还没工作完)去分任务。

如果是前者，当然没问题，如果是后者问题来了，因为3号是今天才能完工的，但也是放在右边，如果厂长刚好分了2号和4号，那么按上面的逻辑，4号就催不到了。

所以为了避免这种情况，当天能完的还不能放在空闲里。嗯，只要再准备一个口袋就行了。好在监工的衣服上面有两个口袋，下面也有两个，用了下面两个，上面还有两个没有被使用。拿一个来用就行了。

检查下面的左右口袋时，凡是当天能做完的，都放在左上。OK，先右下，再左下，当天能做完的这会儿放右上，再把左上的按顺序插入右下。应该没问题了吧， 不管厂长何时分任务，监工只需要看自己的口袋就可以了。右下需要访问的是从昨天访问的没工作的开始，再到一个新的没工作的。左下都要访问。右上或左上的卡 片只需要整理。因为实际上每天工人(80%左右)是没任务的，都在右下，并且也可能好几十天没新任务，这下轻松了，只用问小部分工人就能保证工作正常进行 了。

好了，如果想先模拟一遍怎么办，用大脑模拟，10000人太多想不过来，累。写个程序吧。要写程序得先有算法。

2.算法

以下算法模拟最一般的情况，即监工不知道厂长何时分任务，厂长在一天的任何时候都可以分任务，工人每天都要问才工作，监工只在早上去催(为了简化工时的计算，即工时以天为单位)。

完全的随机起点模拟，即此方法可从任何监工想要采用此方法的时间点开始。

假设一个任务完成的时间是1-N天，厂子一天接到的新任务数是0-M个。用T分钟模拟一天。定时器是精度毫秒级。

A.用0-N之间的数初始化EE[10000]，模拟当前工人的工作状态。EE[i]表示工人还要多少天完成这个任务。EE[i]=0，表示没任务。

B.设置定时器，厂长分任务定时器为1-T*60*1000毫秒之间某个时间，监工定时器设为T*60*1000毫秒。

C.厂长定时器到，厂长分任务。用c记录厂长从哪里开始。第一次时有个随机初始化的工程，随机一个0-9999之间的数，然后找到第一个EE[i]=0的i，从这个c=i开始分配。

随机产生0-M，如果=0，则c=i不变，如果是1-M之间的值，则一个个查，碰到EE[i]=0的，给他随机一个1-N的值，直到分完这些任务，并且c=最后分到的+1。这个题目是研究监工的问题是，厂长比较轻松，下次让他继续从这里找下去。

重新设定厂长定时器，定时为T*60*1000-上次定的时，再加上1-T*60*1000毫秒之间某个时间,因为厂长也只会一天分一次，所以先要把今 天的时间用完，再加上下一天的某个时间，(从前面可以看出，厂长的定时器设成T也是一样的，只要考虑一下访问共享数据的问题，这里先不考虑这个问题)。

D.监工定时器到，监工问工人。

新建四个链表。a(今天还不能做完的),b(没有工作的),c(今天可以做完的),d(今天可以做完的)，初始化为空。但b为有序链表。c和d轮流使用。

第一次定时到，访问EE[10000]，今天不能做完的(EE[i]＞ 1)接到a的尾巴上，能做完的(EE[i]=1)接到c的尾巴上，没有工作的(EE[i]=0)接b，d空。

第二次定时到，访问b，今天不能做完的接到a的尾巴上，能做完的接到d的尾巴上，并且记录出现有工作的人后再出现没有工作的人的结点指针p。如果没有这样的人，那就是链表第一个人或者为空(大家都有工作)。但不管怎样必须把整个链表b访问一遍。

访问a,不能做完的不动，能做完的接到d的尾巴上，最后将c中的元素插入b。注意，链表中元素唯一，也就是说移到另一个链表的时候，也意味着从原链表删除。

第三次定时到，从p开始访问b中的元素，今天不能做完的接到a的尾巴上，能做完的接到c的尾巴上，直到找到一个没有工作的或者已经全部找了一遍(找 的时候调整p到新的位置)。到链表最后一个后，可能要从头找。

访问a,不能做完的不动，能做完的接到c的尾巴上，最后将d中的元素插入b。

第四次定时到，和第3次类似，只是c和d的位置对调了一下。到这时，监工的工作已经轻松了，整个系统将按这个新的方式一直运行下去。

监工的定时器不需要重新设置。

a,b,c,d中的元素内容为工人编号，访问时语法类似if(EE[p-＞ index]＞ 1)。

腾讯部分笔试面试题
1、请定义一个宏，比较两个数a、b的大小，不能使用大于、小于、if语句
2、如何输出源文件的标题和目前执行行的行数
3、两个数相乘，小数点后位数没有限制，请写一个高精度算法
4、写一个病毒
5、有A、B、C、D四个人，要在夜里过一座桥。他们通过这座桥分别需要耗时1、2、5、10分钟，只有一支手电，并且同时最多只能两个人一起过桥。请问，如何安排，能够在17分钟内这四个人都过桥？

2008年腾讯招聘
选择题(60)
c/c++ os linux 方面的基础知识 c的Sizeof函数有好几个!
程序填空(40)
1.(20) 4空x5
不使用额外空间,将 A,B两链表的元素交叉归并
2.(20) 4空x5
MFC 将树序列化 转存在数组或 链表中!

1, 计算 a^b << 2 (运算符优先级问题)

2 根据先序中序求后序

3 a[3][4]哪个不能表示 a[1][1]: *(&a[0][0]) *(*(a+1)+1) *(&a[1]+1) *(&a[0][0]+4)

4 for(int i...)
for(int j...)
printf(i,j);
printf(j)
会出现什么问题

5 for(i=0;i<10;++i,sum+=i);的运行结果

6 10个数顺序插入查找二叉树，元素62的比较次数

7 10个数放入模10hash链表，最大长度是多少

8 fun((exp1,exp2),(exp3,exp4,exp5))有几个实参

9 希尔 冒泡 快速 插入 哪个平均速度最快

10 二分查找是 顺序存储 链存储 按value有序中的哪些

11 顺序查找的平均时间

12 *p=NULL *p=new char[100] sizeof(p)各为多少

13 频繁的插入删除操作使用什么结构比较合适，链表还是数组

14 enum的声明方式

15 1-20的两个数把和告诉A,积告诉B，A说不知道是多少，

B也说不知道，这时A说我知道了，B接着说我也知道了，问这两个数是多少

大题：

1 把字符串转换为小写，不成功返回NULL,成功返回新串

```
char* toLower(char* sSrcStr)
{
char* sDest= NULL;
if( __1___)
{
int j;
sLen = strlen(sSrcStr);
sDest = new [_______2_____];
if(*sDest == NULL)
return NULL;
sDest[sLen] = '\0';
while(_____3____)
sDest[sLen] = toLowerChar(sSrcStr[sLen]);
```

```
}
return sDest;
}
```

2 把字符串转换为整数 例如："-123" -> -123

```
main()
{
.....
if( *string == '-' )
n = ____1_____;
else
n = num(string);
.....
}

int num(char* string)
{
for(;!(*string==0);string++)
{
int k;
k = __2_____;
j = --sLen;
while( __3__)
k = k * 10;
num = num + k;
}
return num;
}
```

附加题：

1 linux下调试core的命令，察看堆栈状态命令
2 写出socks套接字 服务端 客户端 通讯程序
3 填空补全程序，按照我的理解是添入：win32调入dll的函数名
查找函数入口的函数名 找到函数的调用形式
把formView加到singledoc的声明 将singledoc加到app的声明

4 有关系 s(sno,sname) c(cno,cname) sc(sno,cno,grade)
1 问上课程 "db"的学生no
2 成绩最高的学生号
3 每科大于90分的人数

主要是c/c++、数据结构、操作系统等方面的基础知识。好像有sizeof、树等选择题。填空题是补充完整程序。附加题有写算法的、编程的、数据库sql语句查询的。还有一张开放性问题。
请定义一个宏，比较两个数a、b的大小，不能使用大于、小于、if语句
#define Max(a,b) (a/b)?a:b
如何输出源文件的标题和目前执行行的行数
int line = __LINE__;
char *file = __FILE__;
cout<<"file name is "<<(file)<<",line is "<<line<<endl;
两个数相乘，小数点后位数没有限制，请写一个高精度算法

写一个病毒

```
while (1)
        {
                int *p = new int[10000000];
        }
```

不使用额外空间,将 A,B两链表的元素交叉归并
将树序列化 转存在数组或 链表中

```
struct st{
 int i;
 short s;
 char c;
};
sizeof(struct st);
```

8

```
   char * p1;
   void * p2;
   int p3;
   char p4[10];
   sizeof(p1...p4) =?
```

4，4，4，10

二分查找
快速排序
双向链表的删除结点

有12个小球,外形相同,其中一个小球的质量与其他11个不同
给一个天平,问如何用3次把这个小球找出来
并且求出这个小球是比其他的轻还是重
解答:
哈哈，据说这是微软前几年的一个面试题。很经典滴啊！三次一定能求出来，而且能确定是重还是轻。
数据结构的知识还没怎么学透，不过这个题我到是自己研究过，可以分析下。
将12个球分别编号为a1,a2,a3.....a10,a11,a12.
第一步：将12球分开3拨，每拨4个，a1~a4第一拨，记为b1， a5~a6第2拨，记为b2，其余第3拨，记为b3；
第二步：将b1和b2放到天平两盘上，记左盘为c1，右为c2；这时候分两中情况：

1.c1和c2平衡，此时可以确定从a1到a8都是常球；然后把c2拿空，并从c1上拿下a4，从a9到a12四球里随便取三球，假设为a9到a11，放到c2上。此时c1上是a1到a3，c2上是a9到a11。从这里又分三种情况：
A：天平平衡，很简单，说明没有放上去的a12就是异球，而到此步一共称了两次，所以将a12随便跟11个常球再称一次，也就是第三次，马上就可以确定a12是重还是轻；
B： 若c1上升，则这次称说明异球为a9到a11三球中的一个，而且是比常球重。取下c1所有的球，并将a8放到c1上，将a9取下，比较a8和a11（第三 次称），如果平衡则说明从c2上取下的a9是偏重异球，如果不平衡，则偏向哪盘则哪盘里放的就是偏重异球；
C：若c1下降，说明a9到a11里有一个是偏轻异球。次种情况和B类似，所以接下来的步骤照搬B就是；

2.c1和c2不平衡，这时候又分两种情况，c1上升和c1下降，但是不管哪种情况都能说明a9到a12是常球。这步是解题的关键。也是这个题最妙的地方。
A：c1上升，此时不能判断异球在哪盘也不能判断是轻还是重。取下c1中的a2到a4三球放一边，将c2中的a5和a6放到c1上，然后将常球a9放到c2上。至此，c1上是a1，a5和a6，c2上是a7，a8和a9。此时又分三中情况：
1） 如果平衡，说明天平上所有的球都是常球，异球在从c1上取下a2到a4中。而且可以断定异球轻重。因为a5到a8都是常球，而第2次称的时候c1是上升 的，所以a2到a4里必然有一个轻球。那么第三次称就用来从a2到a4中找到轻球。这很简单，随便拿两球放到c1和c2，平衡则剩余的为要找球，不平衡则 哪边低则哪个为要找球；
2）c1仍然保持上升，则说明要么a1是要找的轻球， 要么a7和a8两球中有一个是重球（这步懂吧？好好想想，很简单的。因为a9是常球，而取下的a2到a4肯定也是常球，还可以推出换盘放置的a5和a6也 是常球。所以要么a1轻，要么a7或a8重）。至此，还剩一次称的机会。只需把a7和a8放上两盘，平衡则说明a1是要找的偏轻异球，如果不平衡，则哪边 高说明哪个是偏重异球；
3）如果换球称第2次后天平平衡打破，并且c1降低了，这说明异球肯定在换过来的a5和a6两求中，并且异球偏重，否则天平要么平衡要么保持c1上升。确定要找球是偏重之后，将a5和a6放到两盘上称第3次根据哪边高可以判定a5和a6哪个是重球；
B： 第1次称后c1是下降的，此时可以将c1看成c2，其实以后的步骤都同A，所以就不必要再重复叙述了。至此，不管情况如何，用且只用三次就能称出12个外 观手感一模一样的小球中有质量不同于其他11球的偏常的球。而且在称的过程中可以判定其是偏轻还是偏重。

给一个奇数阶N幻方，填入数字1，2，3...N*N,使得横竖斜方向上的和都相同
答案:
```
#include<iostream>
#include<iomanip>
#include<cmath>
usingnamespace std;
int main()
{
 int n;
 cin>>n;
 int i;
 int **Matr=newint*[n];//动态分配二维数组
 for(i=0;i<n;++i)
      Matr[ i ]=newint[n];//动态分配二维数组
 //j=n/2代表首行中间数作为起点，即1所在位置
 int j=n/2,num=1;//初始值
 i=0;
 while(num!=n*n+1)
 {
//往右上角延升，若超出则用%转移到左下角
      Matr[(i%n+n)%n][(j%n+n)%n]=num;
    //斜行的长度和n是相等的，超出则转至下一斜行
    if(num%n==0)
          i++;
   else
      {
          i--;
          j++;
      }
      num++;
 }
 for(i=0;i<n;i++)
 {
```

```
      for(j=0;j<n;++j)
         cout<<setw((int)log10(n*n)+4)<<Matr[ i][ j ];//格式控制
      cout<<endl<<endl;//格式控制
 }
for(i=0;i<n;++i)
      delete [ ]Matr[ i ];
return1;
}
```

腾讯的一道面试题:(与百度相似,可惜昨天百度死在这方面了)////
在一个文件中有 10G 个整数，乱序排列，要求找出中位数。内存限制为 2G。只写出思路即可。
答案:
1， 把整数分成256M段，每段可以用64位整数保存该段数据个数，256M*8 = 2G内存，先清0
2，读10G整数，把整数映射到256M段中，增加相应段的记数
3，扫描256M段的记数，找到中位数的段和中位数的段前面所有段的记数，可以把其他段的内存释放
4，因中位数段的可能整数取值已经比较小（如果是32bit整数，当然如果是64bit整数的话，可以再次分段），对每个整数做一个记数，再读一次10G整数，只读取中位数段对应的整数，并设置记数。
5，对新的记数扫描一次，即可找到中位数。
如果是32bit整数，读10G整数2次，扫描256M记数一次，后一次记数因数量很小，可以忽略不记
(设是32bit整数，按无符号整数处理
整数分成256M段？ 整数范围是0 - 2^32 - 1 一共有4G种取值，4G/256M = 16，每16个数算一段 0-15是1段，16-31是一段，...
整数映射到256M段中？ 如果整数是0-15，则增加第一段记数，如果整数是16-31，则增加第二段记数，...

其实可以不用分256M段，可以分的段数少一写，这样在扫描记数段时会快一些，还能节省一些内存)

腾讯题二:
一个文件中有40亿个整数，每个整数为四个字节，内存为1GB，写出一个算法：求出这个文件里的整数里不包含的一个整数
答:
方法一: 4个字节表示的整数，总共只有2^32约等于4G个可能。
为了简单起见，可以假设都是无符号整数。
分配500MB内存，每一bit代表一个整数，刚好可以表示完4个字节的整数，初始值为0。基本思想每读入一个数，就把它对应的bit位置为 1 ，处理完40G个数后，对500M的内存遍历，找出一个bit为0的位，输出对应的整数就是未出现的。
算法流程：
1）分配 5 0 0 M B 内存buf，初始化为 0
2）unsigned int x=0x1;
   for each int j in file
   buf=buf |x < <j;
   end
(3) for(unsigned int i=0; i <= 0xffffffff; i++)
       if (!(buf & x < <i))
       {
           output(i);
           break;
       }
以上只是针对无符号的，有符号的整数可以依此类推。

方法二:
文件可以分段读啊，这个是O（2n）算法，应该是很快的了，而且空间也允许的。
不过还可以构造更快的方法的，更快的方法主要是针对定位输出的整数优化算法。
思路大概是这样的，把值空间等分成若干个值段，比如值为无符号数，则
00000000H-00000FFFH
00001000H-00001FFFH
.....
0000F000H-0000FFFFH
.....
FFFFF000H-FFFFFFFFH
这样可以订立一个规则，在一个值段范围内的数第一次出现时，对应值段指示值Xn=Xn+1，如果该值段的所有整数都出现过，则Xn=1000H，这样后面输出定位时就可以直接跳过这个值段了，因为题目仅仅要求输出一个，这样可以大大减少后面对标志数值的遍历步骤。
理论上值段的划分有一定的算法可以快速的实现，比如利用位运算直接定位值段对应值进行计算。
腾讯面试题:
有1到10w这10w个数，去除2个并打乱次序，如何找出那两个数。（不准用位图！！）
位图解决:
  位图的方法如下
假设待处理数组为A[10w-2]
定义一个数组B[10w],这里假设B中每个元素占用1比特,并初始化为全0

```
for(i=0;i <10w-2;i++)
{
 B[ A[i] ]=1
}
```

那么B中不为零的元素即为缺少的数据
这种方法的效率非常高，是计算机中最常用的算法之一
其它方法:

求和以及平方和可以得到结果，不过可能求平方和运算量比较大（用64位int不会溢出）

腾讯面试题:
腾讯服务器每秒有2w个QQ号同时上线，找出5min内重新登入的qq号并打印出来。

解答: 第二题如果空间足够大,可以定义一个大的数组
a[qq号],初始为零,然后这个qq号登陆了就a[qq号]++
最后统计大于等于2的QQ号
这个用空间来代替时间

第二个题目，有不成熟的想法。
2w x 300s
所以用 6,000,000 个桶。删除超时的算法后面说，所以平均桶的大小是 1 。
假设 qq 号码一共有 10^10 个，所以每个桶装的 q 号码是 10^10 / (6 * 10^6) 个，这个是插入时候的最坏效率（插入同一个桶的时候是顺序查找插入位置的）。
qq的节点结构和上面大家讨论的基本一样，增加一个指针指向输出列表，后面说。
```
struct QQstruct {
  num_type   qqnum;
  timestamp  last_logon_time;
  QQstruct   *pre;
  QQstruct   *next;
  OutPutList *out;    // 用于 free 节点的时候，顺便更新一下输出列表。
}
```

另外增加两个指针列表。
第一个大小 300 的循环链表，自带一个指向 QQStruct 的域，循环存 300 秒内的qq指针。时间一过就 free 掉， 所以保证所有桶占用的空间在 2w X 300 以内。
第二个是 输出列表， 就是存放题目需要输出的节点。
如果登陆的用户，5分钟内完全没有重复的话，每秒 free 掉 2w 个节点。
不过在 free 的时候，要判断一下时间是不是真的超时，因为把节点入桶的时候，遇到重复的，会更新一下最后登陆的时间。当然啦，这个时候，要把这个 qq 号码放到需要输出的列表里面。
程序员面试题精选（56）：找出两个链表的第一个公共结点收藏
新一篇: 1到n之间1的个数 | 旧一篇: vs下三个比较实用方便的小技巧

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1个结点，我们先在长的链表上遍历1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;
}
```

求大数据量数组中不重复元素的个数收藏

新一篇: 几道面试题的求解 | 旧一篇: 求n!末尾0的个数

有2.5亿个整数(这2.5亿个整数存储在一个数组里面，至于数组是放在外存还是内存，没有进一步具体说明)；

要求找出这2.5亿个数字里面，不重复的数字的个数（那些只出现一次的数字的数目）；

另外，可用的内存限定为600M;

要求算法尽量高效，最优；

1. caoxic的算法

```
BYTE    marks[2^29];//512M   // BYTE marks[2^32/8]; //用这个就更清楚了,标志所有整数(2^32)出现的可能
BYTE    repmarks[2^25];//32M 32M*8>2.5亿  //标志2.5亿个数字数组里面的数字是否重复过
const BYTE bitmarks[8]={1,2,4,8,16,32,64,128};
DWORD    CalcDifNum(DWORD *pBuf,DWORD bufcount)
{
    DWORD dw ;
    DWORD count = 0 ;// 不重复的数字（包括出现多次的数字，只算一个）的个数,例子：1 2 2 3 5 3 4 算5个
    DWORD count2 = 0 ;//重复出现的数字的个数，例子：1 2 2 3 5 3 4 算2个
    memset(marks,0,sizeof(marks));
    memset(repmarks,0,sizeof(repmarks));
    ASSERT(sizeof(repmarks)*8>=bufcount);//断言repmarks数组够用
    for(dw=0;dw<bufcount;dw++)
    {
        if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])
        {
            repmarks[dw>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复
        }
        else
        {
            count ++;
```

```
            marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现
        }
    }

    memset(marks,0,sizeof(marks));
    for(dw=0;dw<bufcount;dw++)
    {
        if(repmarks[dw>>3] & bitmarks[dw&7])//
        {
            if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])
            {
            }
            else
            {
                count2 ++; //重复的数字的数量
                marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];
            }
        }
    }
    return count-count2;
}
```

2. 改一下，应该也可以：

```
BYTE    marks[2^29];//512M   // BYTE marks[2^32/8]; //用这个就更清楚了,标志所有整数(2^32)出现的可能
BYTE    repmarks[2^25];//32M 32M*8>2.5亿  //标志2.5亿个数字数组里面的数字是否重复过
const BYTE bitmarks[8]={1,2,4,8,16,32,64,128};
DWORD    CalcDifNum(DWORD *pBuf,DWORD bufcount)
{
    DWORD dw ;
    DWORD count = 0 ;// 不重复的数字（包括出现多次的数字，只算一个）的个数,例子：1 2 2 3 5 3 4 算5个
    DWORD count2 = 0 ;//只出现一次数字的个数，例子：1 2 2 3 5 3 4 算3个
    memset(marks,0,sizeof(marks));
    memset(repmarks,0,sizeof(repmarks));
    ASSERT(sizeof(repmarks)*8>=bufcount);//断言repmarks数组够用
    for(dw=0;dw<bufcount;dw++)
    {
        if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])
        {
            repmarks[dw>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复
        }
        else
        {
            count ++;
            marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现
        }
    }

    for(dw=0;dw<bufcount;dw++)
    {
        if(!(repmarks[dw>>3] & bitmarks[dw&7]))//非重复的位置
        {
                count2 ++; //只出现一次的数字的数量
        }
    }
    return count2;
}
```

3. 把数组里面的数字分两类，正数负数，marks数组分成两部分标志,1.该数已经标志[0 --> 2^28-1] 2.标识该数已经重复[2^28 -->2^29-1]。

```
BYTE    marks[2^29];//512M   // BYTE marks[2^32/8]; //用这个就更清楚了,标志所有整数(2^32)出现的可能
const BYTE bitmarks[8]={1,2,4,8,16,32,64,128};
DWORD    CalcDifNum(DWORD *pBuf,DWORD bufcount)
{
    DWORD dw ;
    DWORD count = 0 ;// 不重复的数字（包括出现多次的数字，只算一个）的个数,例子：1 2 2 3 5 3 4 算5个
    DWORD count2 = 0 ;//重复出现的数字的个数，例子：1 2 2 3 5 3 4 算2个
    memset(marks,0,sizeof(marks));
    for(dw=0;dw<bufcount;dw++)
    {
        if(pBuf[dw]>=0)
        {
         if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])
         {
            if(marks[2^28+pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])
            {
```

```
            }else
            {
               marks[2^28+pBuf[dw]>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复
               count2 ++;
            }
         }
         else
         {
             count ++;
             marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现
         }
        }
    }

    memset(marks,0,sizeof(marks));
    for(dw=0;dw<bufcount;dw++)
    {
        if(pBuf[dw]<0)
        {
         pBuf[dw] = -pBuf[dw];
         if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])
         {
            if(marks[2^28+pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])
            {

            }else
            {
               marks[2^28+pBuf[dw]>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复
               count2 ++;
            }
         }
         else
         {
             count ++;
             marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现
         }
        }
    }

    return count-count2;
}
```

程序员面试题（60）：不要被阶乘吓倒收藏

新一篇: 用fstream对二进制文件的读写 | 旧一篇: static_cast、dynamic_cast、reinterpret_cast、和const_cast

阶乘（Factorial）是个很有意思的函数，但是不少人都比较怕它，我们来看看两个与阶乘相关的问题：

1. 给定一个整数N，那么N的阶乘N！末尾有多少个0呢？例如：N＝10，N！＝3 628 800，N！的末尾有两个0。

2. 求N！的二进制表示中最低位1的位置。

请点击"我要发言"，提交您的解法或者问题。

我要看答案

有些人碰到这样的题目会想：是不是要完整计算出N！的值？如果溢出怎么办？事实上，如果我们从"哪些数相乘能得到10"这个角度来考虑，问题就变得简单了。

首 先考虑，如果N！= K×10M，且K不能被10整除，那么N！末尾有M个0。再考虑对N！进行质因数分解，N！=（2x）×（3y）×（5z）…，由于10 = 2×5，所以M只跟X和Z相关，每一对2和5相乘可以得到一个10，于是M = min（X, Z）。不难看出X大于等于Z，因为能被2整除的数出现的频率比能被5整除的数高得多，所以把公式简化为M = Z。

根据上面的分析，只要计算出Z的值，就可以得到N！末尾0的个数。

【问题1的解法一】

要计算Z，最直接的方法，就是计算i（i =1, 2, …, N）的因式分解中5的指数，然后求和：

代码清单2-6

```
--------------------------------------------------------------------------------
ret = 0;
for(i = 1; i <= N; i++)
{
j = i;
while(j % 5 ==0)
{
ret++;
j /= 5;
}
}
```

```
--------------------------------------------------------------------------------
```

【问题1的解法二】

公式：Z = [N/5] +[N/52] +[N/53] + …（不用担心这会是一个无穷的运算，因为总存在一个K，使得5K > N，[N/5K]=0

。)
公式中，[N/5]表示不大于N的数中5的倍数贡献一个5，[N/52]表示不大于N的数中52的倍数再贡献一个5，……代码如下：
```
ret = 0;
while(N)
{
ret += N / 5;
N /= 5;
}
```
问题2要求的是N！的二进制表示中最低位1的位置。给定一个整数N，求N！二进制表示的最低位1在第几位？例如：给定N = 3，N！= 6，那么N！的二进制表示（1 010）的最低位1在第二位。
为了得到更好的解法，首先要对题目进行一下转化。
首先来看一下一个二进制数除以2的计算过程和结果是怎样的。
把一个二进制数除以2，实际过程如下：
判断最后一个二进制位是否为0，若为0，则将此二进制数右移一位，即为商值（为什么）；反之，若为1，则说明这个二进制数是奇数，无法被2整除（这又是为什么）。
所以，这个问题实际上等同于求N！含有质因数2的个数。即答案等于N！含有质因数2的个数加1。
【问题2的解法一】
由于N! 中含有质因数2的个数，等于 N/2 + N/4 + N/8 + N/16 + …[1]，
根据上述分析，得到具体算法，如下所示：
代码清单2-7

```
int lowestOne(int N)
{
int Ret = 0;
while(N)
{
N >>= 1;
Ret += N;
}
return Ret;
}
```

【问题2的解法二】
N! 含有质因数2的个数，还等于N减去N的二进制表示中1的数目。我们还可以通过这个规律来求解。
下面对这个规律进行举例说明，假设 N = 11011，那么N!中含有质因数2的个数为 N/2 + N/4 + N/8 + N/16 + …
即： 1101 + 110 + 11 + 1
=（1000 + 100 + 1）
+（100 + 10）
+（10 + 1）
+ 1
=（1000 + 100+ 10 + 1）+（100 + 10 + 1）+ 1
= 1111 + 111 + 1
=（10000 -1）+（1000 - 1）+（10-1）+（1-1）
= 11011-N二进制表示中1的个数
小结
任 意一个长度为m的二进制数N可以表示为N = b[1] + b[2] * 2 + b[3] * 22 + … + b[m] * 2(m-1)，其中b [i]表示此二进制数第i位上的数字（1或0）。所以，若最低位b[1]为1，则说明N为奇数；反之为偶数，将其除以2，即等于将整个二进制数向低位移一位。
相关题目
给定整数n，判断它是否为2的方幂（解答提示：n>0&&（（n&（n-1））==0））。
[1] 这个规律请读者自己证明（提示N/k，等于1, 2, 3, …, N中能被k整除的数的个数）。
一道经典的面试题：如何从N个数中选出最大（小）的n个数？收藏
新一篇: c/c++基本文件读写 | 旧一篇: 变态比赛规则
一道经典的面试题：如何从N个数中选出最大（小）的n个数？
北京交大LuoBin
这 个问题我前前后后考虑了有快一年了，也和不少人讨论过。据我得到的消息，Google和微软都面过这道题。这道题可能很多人都听说过，或者知道答案（所谓 的“堆”），不过我想把我的答案写出来。我的分析也许存有漏洞，以交流为目的。但这是一个满复杂的问题，蛮有趣的。看完本文，也许会启发你一些没有想过的 解决方案（我一直认为堆也许不是最高效的算法）。在本文中，将会一直以寻找n个最“大”的数为分析例子，以便统一。注：本文写得会比较细节一些，以便于绝 大多数人都能看懂，别嫌我罗嗦:) 我很不确定多少人有耐心看完本文！
Naive 方法：
首先，我们假设n和N都是内存可容纳的，也就是说N个数可以一次load到内存里存放在数组里（如果非要存在链表估计又是另一个challenging的 问题了）。从最简单的情况开始，如果n=1，那么没有任何疑惑，必须要进行N-1次的比较才能得到最大的那个数，直接遍历N个数就可以了。如果n=2呢？ 当然，可以直接遍历2遍N数组，第一遍得到最大数max1，但是在遍历第二遍求第二大数max2的时候，每次都要判断从N所取的元素的下标不等于max1 的下标，这样会大大增加比较次数。对此有一个解决办法，可以以max1为分割点将N数组分成前后两部分，然后分别遍历这两部分得到两个“最大数”，然后二 者取一得到max2。

也可以遍历一遍就解决此问题，首先维护两个元素max1，max2（max1>=max2），取到N中的一个数以后，先和max1比，如果比 max1大（则肯定比max2大），直接替换max1，否则再和max2比较确定是否替换max2。采用类似的方法，对于n=2，3，4……一样可以处 理。这样的算法时间复杂度为O（nN）。当n越来越大的时候（不可能超过N/2，否则可以变成

是找N-n个最小的数的对偶问题），这个算法的效率会越来越 差。但是在n比较小的时候（具体多小不好说），这个算法由于简单，不存在递归调用等系统损耗，实际效率应该很不错.
堆：
当n较大的时候采用什么算法呢？首先我们分析上面的算法，当从N中取出一个新的数m的时候，它需要依次和max1，max2，max3……max n比较，一直找到一个比m小的max x，就用m来替换max x，平均比较次数是n/2。可不可以用更少的比较次数来实现替换呢？最直观的方法是，也就是网上文章比较推崇的堆。堆有这么一些好处：1.它是一个完全二 叉树，树的深度是相同节点的二叉树中最少的，维护效率较高；2.它可以通过数组来实现，而且父节点p与左右子节1，r点的数组下标的关系是s[l] = 2*s[p]+1和s[r] = 2*s[p]+2。在计算机中2*s[p]这样的运算可以用一个左移1位操作来实现，十分高效。再加上数组可以随机存取，效率也很高。3.堆的 Extract操作，也就是将堆顶拿走并重新维护堆的时间复杂度是O（logn），这里n是堆的大小。
具体到我们的问题，如何具体实现呢？首先开辟一个大小为n的数组区A，从N中读入n个数填入到A中，然后将A维护成一个小顶堆（即堆顶A[0]中存放的是 A中最小的数）。然后从N中取出下一个数，即第n+1个数m，将m与堆顶A[0]比较，如果m<=A[0]，直接丢弃m。否则应该用m替换 A[0]。但此时A的堆特性可能已被破坏，应该重新维护堆：从A[0]开始，将A[0]与左右子节点分别比较（特别注意，这里需要比较“两次”才能确定最 大数，在后面我会根据这个来和“败者树”比较），如果A[0]比左右子节点都小，则堆特性能够保证，勿需继续，否则如左（右）节点最大，则将A[0]与左（右）节点交换，并继续维护左（右）子树。依次执行，直到遍历完N，堆中保留的n个数就是N中最大的n个数。这都是堆排序的基本知识， 唯一的trick就是维护一个小顶堆，而不是大顶堆。不明白的稍微想一下。维护一次堆的时间复杂度为O（logn），总体的复杂度是O（Nlogn）这样 一来，比起上面的O（nN），当n足够大时，堆的效率肯定是要高一些的。当然，直接对N数组建堆，然后提取n次堆顶就能得到结果，而且其复杂度是 O（nlogN）,当n不是特别小的时候这样会快很多。但是对于online数据就没办法了，比如N不能一次load进内存，甚至是一个流，根本不知道N 是多少。
败者树：
有没有别的算法呢？我先来说一说败者树（loser tree）。也许有些人对loser tree不是很了解，其实它是一个比较经典的外部排序方法，也就是有x个已经排序好的文件，将其归并为一个有序序列。败者树的思想咋一看有些绕，其实是为 了减小比较次数。首先简单介绍一下败者树：败者树的叶子节点是数据节点，然后两两分组（如果节点总数不是2的幂，可以用类似完全树的结构构成树），内部节 点用来记录左右子树的优胜者中的“败者”（注意记录的是输的那一方），而优胜者则往上传递继续比较，一直到根节点。如果我们的优胜者是两个数中较小的数， 则根节点记录的是最后一次比较中的“败者”，也就是所有叶子节点中第二小的那个数，而最小的那个数记录在一个独立的变量中。这里要注意，内部节点不但要记 录败者的数值，还要记录对应的叶子节点。如果是用链表构成的树，则内部节点需要有指针指向叶子节点。这里可以有一个trick，就是内部节点只记录“败者 ”对应的叶子节点，具体的数值可以在需要的时候间接访问（这一方法在用数组来实现败者树时十分有用，后面我会讲到）。关键的来了，当把最小值输出后，最小 值所对应的叶子节点需要变成一个新的数（或者改为无穷大，在文件归并的时候表示文件已读完）。接下来维护败者树，从更新的叶子节点网上，依次与内部节点比 较，将“败者”更新，胜者往上继续比较。由于更新节点占用的是之前的最小值的叶子节点，它往上一直到根节点的路径与之前的最小值的路径是完全相同的。内部 节点记录的“败者”虽然称为“败者”，但却是其所在子树中最小的数。也就是说，只要与“败者”比较得到的胜者，就是该子树中最小的那个数（这里讲得有点绕 了，看不明白的还是找本书看吧，对照着图比较容易理解）。
注：也可以直接对N构建败者树，但是败者树用数组实现时不能像堆一样进行增量维护，当叶子节点的个数变动时需要完全重新构建整棵树。为了方便比较堆和败者树的性能，后面的分析都是对n个数构建的堆和败者树来分析的。
总而言之，败者树在进行维护的时候，比较次数是logn+1。与堆不同的是，败者树是从下往上维护，每上一层，只需要和败者节点比较“一次”即可。而堆在 维护的时候是从上往下，每下一层，需要和左右子节点都比较，需要比较两次。从这个角度，败者树比堆更优一些。但是，请注意但是，败者树每一次维护必定需要 从叶子节点一直走到根节点，不可能中间停止；而堆维护时，“有可能”会在中间的某个层停止，不需要继续往下。这样一来，虽然每一层败者树需要的比较次数比 堆少一倍，但是走的层数堆会比败者树少。具体少多少，从平均意义上到底哪一个的效率会更好一些？那我就不知道了，这个分析起来有点麻烦。感兴趣的人可以尝 试一下，讨论讨论。但是至少说明了，也许堆并非是最优的。
具体到我们的问题。类似的方法，先构建一棵有n个叶子节点的败者树，胜出者w是n个中最小的那一个。从N中读入一个新的数m后，和w比较，如果比w小，直 接丢弃，否则用m替换w所在的叶子节点的值，然后维护该败者树。依次执行，直到遍历完N，败者树中保留的n个数就是N中最大的n个数。时间复杂度也是 O（Nlogn）

类快速排序方法：
快速排序大家大家都不陌生了。主要思想是找一个“轴”节点，将数列交换变成两部分，一部分全都小于等于“轴”，另一部分全都大于等于“轴”，然后对两部分 递归处理。其平均时间复杂度是O（NlogN）。从中可以受到启发，如果我们选择的轴使得交换完的“较大”那一部分的数的个数j正好是n，不也就完成了在 N个数中寻找n个最大的数的任务吗？当然，轴也许不能选得这么恰好。可以这么分析，如果j>n，则最大的n个数肯定在这j个数中，则问题变成在这j个数中找出n个最大的数；否则如果j<n，则这j个数肯定是n个最大的数的一部分，而剩下的j-n个数在小于等于轴的那一部分中，同样可递归处理。
令人愉悦的是，这个算法的平均复杂度是O（N）的。怎么样？比堆的O（Nlogn）可能会好一些吧？！（n如果比较大肯定会好）
需要注意的是，这里的时间复杂度是平均意义上的，在最坏情况下，每次分割都分割成1：N-2，这种情况下的时间复杂度为O（n）。但是我们还有杀手锏，可 以有一个在最坏情况下时间复杂度为O（N）的算法，这个算法是在分割数列的时候保证会按照比较均匀的比例分割，at least 3n/10-6。具体细节我就不再说了，感兴趣的人参考算法导论（Introduction to Algorithms 第二版第九章 “Medians and Orders Statistics”）。
还是那个结论，堆不见得会是最优的。
本文快要 结束了，但是还有一个问题：如果N非常大，存放在磁盘上，不能一次装载进内存呢？怎么办？对于介绍的Naive方法，堆，败者树等等，依然适用，需要注意 的就是每次从磁盘上尽量多读一些数到内存区，然后处理完之后再读入一批。减少IO次数，自然能够提高效率。而对于类快速排序方法，稍微要麻烦一些：分批读 入，假设是M个数，然后从这M个数中选出n个最大的数缓存起来，直到所有的N个数都分批处理完之后，再将各批次缓存的n个数合并起来再进行一次类快速排序 得到最终的n个最大的数就可以了。在运行过程中，如果缓存数太多，可以不断地将多个缓存合并，保留这些缓存中最大的n个数即可。由于类快速排序的时间复杂 度是O（N），这样分批处理再合并的办法，依然有极大的可能会比堆和败者树更优。当然，在空间上会占用较多的内存。
总结：对于这个问题，我 想了很多，但是觉得还有一些地方可以继续深挖：1. 堆和败者树到底哪一个更优？可以通过理论分析，也可以通过实验来比较。也许会有人觉得这个很无聊；2. 有没有近似的算法或者概率算法来解决这个问题？我对这方面实在不熟悉，如果有人有想法的话可以一块交流。如果有分析错误或遗漏的地方，请告知，我不怕丢 人，呵呵！最后请时刻谨记，时间复杂度不等于实际的运行时间，一个常数因子很大的O（logN）算法也许会比常数因子小的O（N）算法慢很多。所以说，n 和N的具体值，以及编程实现的质量，都会影响到实际效率。我看过一篇论文，给出的算法在进行字符串查找时，比hash还要快，是不是难以想象？

1到n之间1的个数收藏

新一篇: 程序员面试题精选{57）：求n的加法组合 | 旧一篇: 程序员面试题精选（56）：找出两个链表的第一个公共结点

Consider a function which, for a given whole number n, returns the number of ones required when writing out all numbers between 0 and n.
For example, f(13)=6. Notice that f(1)=1. What is the next largest n such that f(n)=n?

算法思想：

循环求出每个数中1的个数，累计之，若满足f(n)=n，则退出，否则继续。
代码如下：

```
 /**************************************************************************
* 0~n之间1的个数,如f(13)=6
* 1,2,3,4,5,6,7,8,9,10,11,12,13.1的个数为6
* 要求:输入一个正整数n,求出f(n),及求解f(n)=n
**************************************************************************/

#include <stdio.h>
#include <string.h>
#include <Windows.h>

class CalculationTimes
 {
public:
     CalculationTimes(){}
     ~CalculationTimes(){}

    int GetTimes(int n);
};

//计算正整数n中1的个数
int CalculationTimes::GetTimes(int n)
 {
    int count=0;
    while(n)
     {
        if(n%10==1)
            count++;
        n/=10;
    }
    return count;
}

//显示菜单
void show_menu()
 {
    printf("------------------------------------------- ");
    printf("input command to test the program ");
    printf("   i or I : input n to test ");
    printf("   g or G : get n to enable f(n)=n ");
    printf("   q or Q : quit ");
    printf("------------------------------------------- ");
    printf("$ input command >");
}

void main()
 {
    char sinput[10];
    int n;

    show_menu();

    scanf("%s",sinput);
    while(stricmp(sinput,"q")!=0)
     {
        int t=0,count=0;
        if(stricmp(sinput,"i")==0)
         {
            printf("  please input an integer:");
            scanf("%d",&n);

            count=0;
            CalculationTimes obj;
            t=GetTickCount();
            for(int i=1;i<=n;i++)
```

```
                count+=obj.GetTimes(i);
            t=GetTickCount()-t;
            printf("   count=%d    time=%d ",count,t);
        }
        else if(stricmp(sinput,"g")==0)
         {
            CalculationTimes obj;
            n=2;
            count=1;
            t=GetTickCount();
            while(1)
             {
                count+=obj.GetTimes(n);
                if(count==n)
                    break;
                n++;
            }
            t=GetTickCount()-t;
            printf("   f(n)=n=%d    time=%d ",n,t);
        }


        //输入命令
        printf("$ input command >");
        scanf("%s",sinput);
    }
}
```

程序员面试题精选{57）：求n的加法组合收藏
新一篇: 程序员面试题精选{57）：求n的加法组合 | 旧一篇: 1到n之间1的个数
一个正整数n可以写为几个正整数的和，如：
4=4
4=3+1
4=2+2
4=2+1+1
4=1+1+1+1
输入一个正整数,找出符合这种要求的所有正整数序列(序列不重复)
算法思想：

以n=6为例，将数继序列暂时存于a[MAXCOL]中，且初始时值全为1。
对数组中从jcol列开始的newn个元素进行操作f(6,0,0) ——函数GetCombinations(newn,newj,col)
col记录调用该函数时是在第col列。

初始化
count=6
j=0,newn=0
count=n
1. 如果count>1，令a[jcol]=count;
        若count>a[col]，表明该尝试不满足条件，count=count-1，重复1;
        否则将该行第jcol+1列到jcol+count-1列的值改为0;
   否则，退出;
2. newn=n-count
3. 如果new>1，则对从该行开始从jcol+count开始的newn个元素进行类似操作，并返回该新的newn对应的序列个数;
   否则，count=count-1，返回1。
算法比较：
“acm题目及我的程序（3）——正整数n的加法组合”——使用二维数组存放加法序列
        #define MAXROW 12000
        #define MAXCOL 20
        a[MAXROW][MAXCOL]
        算法效率低，空间浪费严重
“acm题目及我的程序（3）——正整数n的加法组合（改进）”——使用二维数组存放加法序列
        a[MAXROW][MAXCOL]
        #define MAXROW 6000
        #define MAXCOL 30
        算法效率高，空间浪费不严重

“acm题目及我的程序（3）——正整数n的加法组合（改进2）”——使用动态二维数组存放加法序列
        vector<vector<int> > m_venline
        算法效率高，空间浪费很少

  本文算法——使用一维数组存放加法序列，且计算每个n的加法序列的个数
        a[MAXCOL]
        算法效率最高，空间根本不浪费
代码如下：

```
  /*************************************************************************
```

```
 * 一个正整数n可以写为几个正整数的和
 * 4=4
 * 4=3+1
 * 4=2+2
 * 4=2+1+1
 * 4=1+1+1+1
 * 要求:输入一个正整数,找出符合这种要求的所有正整数序列(序列不重复)
 ******************************************************************************/

#include <stdio.h>
#include <string.h>
#include <CONIO.H>
#include <vector>
using namespace std;

#define MAXCOL 80
#define FILENAMELENGTH 100

class AdditionCombination
 {
public:
    int a[MAXCOL];
    int m_number;    //条用GetCombinations函数时count的值

public:
    AdditionCombination(int number)
     {
        m_number=number;
        for(int j=0;j<MAXCOL;j++)
            a[j]=1;
    }

     ~AdditionCombination(){}

    void Initialize()
     {
        for(int j=0;j<m_number;j++)
            a[j]=1;
    }

    void Initialize(int jcol)
     {
        for(int j=jcol;j<m_number;j++)
            a[j]=1;
    }

    int GetCombinations(int n,int jcol,int col);
    void display(int n);
};

//在数组中从irow行,jcol列开始对n阶子矩阵进行
//col记录调用该函数前jcol的值
int AdditionCombination::GetCombinations(int n,int jcol,int col)
 {
    int nTotalCount=0;
    int j=0,newn=0;
    int count=n;

    while(count>1)
     {
        if(jcol==0)
            Initialize();
        else
            Initialize(jcol);
        a[jcol]=count;

        //算法优化,删除不满足的序列
        if(a[jcol]>a[col])
         {
            count--;
            continue;
        }

        for(j=jcol+1;j<jcol+count;j++)
            a[j]=0;
```

```
        newn=n-count;
        if(newn>1)
         {
            int newj=jcol+count;
            int newcount=GetCombinations(newn,newj,jcol);
            nTotalCount+=newcount;
        }

        count--;
        //display(m_number);
        nTotalCount++;
    }

    if(jcol==0)
     {
        Initialize();
        //display(m_number);
        nTotalCount++;
    }
    else
        Initialize(jcol);

    return nTotalCount;
}

void AdditionCombination::display(int n)
 {
    printf("   %d=%d",n,a[0]);
    for(int j=1;j<n;j++)
     {
        if(a[j])
            printf("+%d",a[j]);
    }
}

//将结果写入文件
void WriteToFile(vector<vector<int> > info)
 {
    char filename[FILENAMELENGTH];

    int size=info.size();
    //构造文件名
    sprintf(filename,"%d-%d result.txt",info[0][0],info[size-1][0]);
    FILE *fp=fopen(filename,"w");
    if(fp==NULL)
     {
        printf("can not wirte file!");
        exit(0);
    }

    //写入个数
    for(int i=0;i<size;i++)
        fprintf(fp,"n=%d    count=%d ",info[i][0],info[i][1]);

    fclose(fp);
}

//显示菜单
void show_menu()
 {
    printf("-------------------------------------------- ");
    printf("input command to test the program ");
    printf("   i or I : input n to test ");
    printf("   t or T : get count from n1 to n2 ");
    printf("   q or Q : quit ");
    printf("-------------------------------------------- ");
    printf("$ input command >");
}

void main()
 {
    char sinput[10];
    int n;
```

```
    show_menu();

    scanf("%s",sinput);
    while(stricmp(sinput,"q")!=0)
     {
        if(stricmp(sinput,"i")==0)
         {
            printf("  please input an integer:");
            scanf("%d",&n);

            AdditionCombination obj(n);
            int count=obj.GetCombinations(n,0,0);
            printf("   count = %d ",count);
        }
        else if(stricmp(sinput,"t")==0)
         {
            int n1,n2;
            printf("  please input the begin number:");
            scanf("%d",&n1);
            printf("  please input the  end  number:");
            scanf("%d",&n2);

            printf("  press any key to start ... ");
            getch();

            vector<vector<int> > info;
            vector<int> line;

            AdditionCombination obj(n1);
            for(int i=n1;i<=n2;i++)
             {
                obj.Initialize();
                obj.m_number=i;
                int count=obj.GetCombinations(i,0,0);
                printf("  n=%d    count=%d ",i,count);
                line.clear();
                line.push_back(i);
                line.push_back(count);
                info.push_back(line);
            }
            printf(" ");

            //写入文件
            printf("$ write the numbers to file(Y,N)? >");
            scanf("%s",sinput);
            if(stricmp(sinput,"y")==0)        //写入文件
             {
                WriteToFile(info);
                printf("  write successfully! ");
            }
            printf(" ");
        }

        //输入命令
        printf("$ input command >");
        scanf("%s",sinput);
    }
 }
```

google的一道面试题收藏

新一篇: 程序员面试题精选{58}：字符串插入字符个数 | 旧一篇: 程序员面试题精选{57}：求n的加法组合

题目：

输入a1,a2,...,an,b1,b2,...,bn, 在O(n)的时间,O(1)的空间将这个序列顺序改为a1,b1,a2,b2,a3,b3,...,an,bn

不需要移动，通过交换完成，只需一个交换空间

例如，N=9时，第2步执行后，实际上中间位置的两边对称的4个元素基本配对，只需交换中间的两个元素即可，如下表所示。颜色表示每次要交换的元素,左边向右交换,右边向左交换。交换过程如下表所示。

	1	2	3	4	5	6	7	8	9	n+1	n+2	n+3
n+4	n+5	n+6	n+7	n+8	n+9							
	n-8	n-7	n-6	n-5	n-4	n-3	n-2	n-1	N	2n-8	2n-7	2n-6
2n-5	2n-4	2n-3	2n-2	2n-1	2n	交换开始位置		交换个数				
	a1	a2	a3	a4	a5	a6	a7	a8	a9	b1	b2	b3
b4	b5	b6	b7	b8	b9	2?n+1	2n-1?n	1				
1	a1	b1	a3	a4	a5	a6	a7	a8	b8	a2	b2	b3
b4	b5	b6	b7	a9	b9	3?n+1	2n-2?n	2				

2 a1 B1 a2 b2 a5 a6 a7 b6 b7 a3 a4 b3
b4 b5 a8 b8 a9 b9 5?n+1 2n-4?n 4
3 a1 B1 a2 b2 X1 Y1=(a6 a7 b7) X2 X3 Y2=(a4 b3 b4) X4
a8 b8 a9 b9 对称交换
a1 B1 a2 b2 X3 Y2 X4 X1 Y1 X2 a8 b8
a9 b9
a1 B1 a2 b2 X3 Y2 X1 X4 Y1 X2 a8 b8
a9 b9
4 a1 B1 a2 b2 A3 A4 B3 B4 A5 B5 A6 A7
B6 B7 a8 b8 a9 b9
5 a1 B1 a2 b2 A3 B3 A4 B4 A5 B5 A6 B6
A7 B7 a8 b8 a9 b9
交换x1,x3；交换x2,x4；再交换中间的x1,x4；交换y1,y2；
算法思想：
以N=9为例(中间的竖表示中间位置)：
a1 a2 a3 a4 a5 a6 a7 a8 a9 | b1 b2 b3 b4 b5 b6 b7 b8 b9
头尾的元素不需任何操作
1. 左边从位置left=2开始，右边从位置n+1开始，向右交换count=1个元素，即a2,b1交换
右边从位置right=2n-1开始，左边从位置n开始，向左交换count=1个元素，即b8,a9交换
序列变为：
a1 b1 a3 a4 a5 a6 a7 a8 b8 | a2 b2 b3 b4 b5 b6 b7 a9 b9
故已经成功放好位置的有(a1,b1),(a9,b9)
其中(a8,b8),(a2,b2)也配对，只需将其交换到相应的位置即可
2. 左边从位置left=3开始，右边从位置n+1开始，向右交换count=2个元素，即a3 a4 和 a2 b2交换
右边从位置right=2n-2开始，左边从位置n开始，向左交换count=2个元素，即b6 b7 和 a8 b8交换
序列变为：
a1 b1 a2 b2 a5 a6 a7 b6 b7 | a3 a4 b3 b4 b5 a8 b8 a9 b9
故又成功放好位置的有(a2,b2),(a8,b9)
3. 左边从位置left=5开始，右边从位置n开始，已不能满足交换count=4个元素的要求，故退出循环
4. 序列缩小为a5 a6 a7 b6 b7 | a3 a4 b3 b4 b5, 对序列中没有放好的数据按快处理
将序列看作：x1=(a5) y1=(a6 a7 b6) x2=(b7) | x3=(a3) y2=(a4 b3 b4) x4=(b5)
交换x1,x3；交换x2,x4；再交换中间的x1,x4；交换y1,y2；
此时序列变为S1：a3 a4 b3 b4 a5 b5 a6 a7 b6 b7
5. 若交换到左边的b有配对的a，则中间的序列S2=(a5 b6)作为新的序列；否则将S1作为新的序列；对该序列进行上述操作，直到所有元素都放到正确的位置
此例中，(a6,a7,b6,b7),(a3,a4,b3,b4)以配对，只需交换中间的两个元素即可，序列缩小a5 b5

==========
为方便比较，序列变化如下：
a1 a2 a3 a4 a5 a6 a7 a8 a9 | b1 b2 b3 b4 b5 b6 b7 b8 b9
a1 b1 a3 a4 a5 a6 a7 a8 b8 | a2 b2 b3 b4 b5 b6 b7 a9 b9
a1 b1 a2 b2 a5 a6 a7 b6 b7 | a3 a4 b3 b4 b5 a8 b8 a9 b9
a1 b1 a2 b2 a3 a4 b3 b4 a5 | b5 a6 b6 a7 b7 a8 b8 a9 b9
a1 b1 a2 b2 a3 b3 a4 b4 a5 | b5 a6 b6 a7 b7 a8 b8 a9 b9
源代码如下：

```
/*********************************************************************
* 输入a1,a2,...,an,b1,b2,...,bn
* 在O(n)的时间,O(1)的空间
* 将这个序列顺序改为a1,b1,a2,b2,a3,b3,...,an,bn
*********************************************************************/

#include <stdio.h>
#include <string.h>
#include <CONIO.H>

#define MAXSIZE 2*100000

int exchangetimes=0;    //交换次数

//交换两个数据
void swap(int *x,int *y)
 {
    int t;
    t=*x;
    *x=*y;
    *y=t;

    exchangetimes++;
}

//按要求交换序列(假设元素从下标1开始存放)
void exchange(int a[],int m)
 {
    int n=m/2;
```

```
if(n==1)         //a1,b1 ==>不需交换
    return;
else if(n==2)     //a1,a2,b1,b2 ==>只需交换中间的两个数据
 {
    swap(a+1,a+2);
    return;
}

int done;         //已经处理的数据个数
int left;         //左边开始交换的位置
int right;         //右边开始交换的位置
int count;         //每次交换的数据个数
int lefta;         //左边未处理的a个数
int notmatch;     //左边未匹配的a个数

//初始化
done=1;
left=1;             //左边从位置1开始向右交换
right=2*n-2;     //右边从位置2*n-2开始向左交换
count=1;         //每次交换count个数据
lefta=notmatch=n-1;

while(1)
 {
    //左边从left开始和右边从n开始向右交换count个数据
    for(int j=0;j<count;j++)
        swap(a+left+j,a+n+j);

    //右边从right开始和左边从n-1开始向左交换count个数据
    for(j=0;j<count;j++)
        swap(a+right-j,a+n-1-j);

    //交换后将其调整为要求的序列
    if(count>=4)
     {
        exchange(a+left,count);
        exchange(a+right-count+1,count);
    }

    //重新调整各变量
    done+=count;
    lefta=n-done-count;
    notmatch=lefta-count;
    left=left+count;
    right=right-count;
    count*=2;

    if(notmatch<count)
        break;
}

int x,y;

//左边剩下的a不能与从右边交换过来的b配对
//如n=13时,上面的循环结束后变为: a9 (b6 b7 b8) b9 | a5 (a6 a7 a8) b5
//此时,lefta=1,notmatch<0,分块交换,各个块如下
//x1=(a9),  y1=(b6 b7 b8),  x2=(b9),  x3=(a5),  y2=(a6 a7 a8),  x4=(b5)
//上述序列变为 x1 y1 x2 x3 y2 x4, x的长度均为1,y的长度均为3
//x1 x3交换,x2 x4交换==>x3 x4 x1 x2,然后中间的x4 x1交换==>x3 x1 x4 x2
//y1 y2交换==>y2 y1
//上述6块经4次交换后变为 x3 y2 x1 x4 y1 x2
//n=13时,经上述交换后变为 a5 a6 a7 a8 a9 b5 b6 b7 b8 b9
if(notmatch<0)
 {
    count/=2;     //if n=13, then here count=4
    x=lefta;     //x块的长度,if n=13, then heare x=1
}
else
 {
    //递归调用中间对称的count个数据
    exchange(a+n-count,count);
    exchange(a+n,count);

    x=notmatch;
}
```

```
     ////////////////////////////////////////////////////////////////////////
    y=count-x;     //y块的长度,if n=13, then heare y=3
    //左边从left开始和右边从n开始向右交换x个数据,即x1 x3交换
    //右边从right开始和左边从n-1开始向左交换x个数据,即x2 x4交换
    for(int j=0;j<x;j++)
     {
        swap(a+left+j,a+n+j);         //左边向左交换
        swap(a+right-j,a+n-1-j);     //右边向左交换
    }

    //交换到中间的x1 x4交换
    for(j=0;j<x;j++)
        swap(a+n-x+j,a+n+j);

    //交换y1 y2数据块
    for(j=0;j<y;j++)
        swap(a+left+x+j,a+n+x+j);
     ////////////////////////////////////////////////////////////////////////

    //处理余下的序列
    if(notmatch<0)
     {
        int newm=2*(n-left);     //或者size=x+count
        exchange(a+left,newm);
    }
    else if(notmatch>0)
     {
        int newm=2*notmatch;
        exchange(a+left+count,newm);
    }
}

//显示菜单
void show_menu()
 {
    printf("-------------------------------------------- ");
    printf("input command to test the program ");
    printf("   i or I : input n to test ");
    printf("   t or T : test program ");
    printf("   q or Q : quit ");
    printf("-------------------------------------------- ");
    printf("$ input command >");
}

//显示数据
void display(int a[],int n)
 {
    for(int i=0; i<n;i++)
        printf("%3d",a[i]);
    printf(" ");
}

//检查交换是否正确
bool check(int a[],int n)
 {
    int i;

    for(i=0;i<n-2;i+=2)
     {
        if(a[i]!=i/2+1)
            return false;
    }

    for(i=1;i<n-1;i+=2)
     {
        if(a[i]!=(n+i+1)/2)
            return false;
    }

    return true;
}

void main()
 {
```

```
    int a[MAXSIZE];
    int n;
    char sinput[10];

    show_menu();

    scanf("%s",sinput);
    while(stricmp(sinput,"q")!=0)
     {
        if(stricmp(sinput,"i")==0)
         {
            printf("  please input n:");
            scanf("%d",&n);

            //且假设数组中的数据为1,2,3,...,n,n+1,n+2,...,2n
            for(int i=0; i<2*n;i++)    //初始化
                a[i]=i+1;

            display(a,2*n);

            exchangetimes=0;

            //交换
            exchange(a,2*n);
            display(a,2*n);

            printf("   exchange times: %d ",exchangetimes);
        }
        else if(stricmp(sinput,"t")==0)
         {
            int n1,n2;
            printf("  please input the begin number:");
            scanf("%d",&n1);
            printf("  please input the  end  number:");
            scanf("%d",&n2);

            printf("  press any key to start ... ");
            getch();

            for(int i=n1;i<=n2;i++)
             {
                //初始化
                for(int j=0; j<2*i;j++)
                    a[j]=j+1;

                exchangetimes=0;
                exchange(a,2*i);

                if(check(a,2*i))
                    printf("  n=%d ... ok!    exchange times: %d ",i,exchangetimes);
                else
                    printf("  n=%d ... wrong! ",i);
            }

            printf(" ");
        }

        //输入命令
        printf("$ input command >");
        scanf("%s",sinput);
    }
}
```