精选微软等公司数据结构+算法面试100题

---- 答案(第21-40题)V0.3版

作者前言：

1. 首先恭喜你，你确实得到了一份不错的资源。
送给你一句话，饮水当思源。
2. 本来可以把这份第21-40的答案弄成word形式的，
那样的话，大家可以直接复制黏贴其中的源码，直接上机验证。
//虽然其中大部分源码，我已经编译调试了，能保证基本的精准。
3. 但我实在实在无法忍受大量大量的人，随意任意转载，
却不注明作者本人我 和出处，有的甚至私自据为己有。所以..见谅。

还是这句话，展示自己的思考结果，是一种骄傲。

其它资源，下载地址：
[第1题-60题汇总]微软等数据结构+算法面试100题
http://download.csdn.net/source/2826690

[答案V0.2版]精选微软数据结构+算法面试100题[前20题]//修正
http://download.csdn.net/source/2813890
//此份答案是针对最初的V0.1版本，进行的校正与修正。
[答案V0.1版]精选微软数据结构+算法面试100题[前25题]
http://download.csdn.net/source/2796735

[第二部分]精选微软等公司结构+算法面试100题[前41-60题]:
http://download.csdn.net/source/2811703
[第一部分]精选微软等公司数据结构+算法经典面试100题[1-40题]
http://download.csdn.net/source/2778852

更多资源，下载地址：
http://v_july_v.download.csdn.net/

若你对以下答案，有任何问题，欢迎联系我：
My E-mail:
zhoulei0907@yahoo.cn
或者，直接回复于此帖上：
[推荐] [整理]算法面试：精选微软经典的算法面试100题[前40-60题]
http://topic.csdn.net/u/20101023/20/5652ccd7-d510-4c10-9671-307a56006e6d.html

July、2010年11月14日，晚，于东华理工。

第 21 题

2010 年中兴面试题

编程求解：

输入两个整数 n 和 m，从数列 1，2，3.....n 中 随意取几个数,

使其和等于 m ,要求将其中所有的可能组合列出来.

//此题与第 14 题差不多，再次不做过多解释。

//July、2010/10/22。

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

void qiujie(int sum, int n)
{
    static list<int> ilist;
    if(n < 1 || sum < 1)
        return;
    if(sum > n)
    {
        ilist.push_front(n);
        qiujie(sum-n, n-1);
        ilist.pop_front();
        qiujie(sum, n-1);
    }
    else
    {
        cout<<sum;
        for(list<int>::iterator it = ilist.begin(); it != ilist.end(); ++it)
            cout << " " << *it;
        cout << endl;
    }
}

int main()
{
    int m, n;
    cout<<"请输入你要等于多少的数值 m:"<<endl;
    cin>>m;
    cout<<"请输入你要从 1.....n 数列中取值的 n："<<endl;
    cin>>n;

    qiujie(m,n);
    return 0;
}
```

//
请输入你要等于多少的数值 m:
10
请输入你要从 1.....n 数列中取值的 n：
8
2 8
3 7
4 6
1 4 5
2 3 5
1 2 3 4
Press any key to continue
//////////////////////////////////////

第 22 题：
有 4 张红色的牌和 4 张蓝色的牌，主持人先拿任意两张，
再分别在 A、B、C 三人额头上贴任意两张牌，

A、B、C 三人都可以看见其余两人额头上的牌，
看完后让他们猜自己额头上是什么颜色的牌，
A 说不知道，B 说不知道，C 说不知道，然后 A 说知道了。

请教如何推理，A 是怎么知道的。如果用程序，又怎么实现呢？

//July、2010/10/22
//今是老妈生日，祝福老妈，生日快乐。!:).

4 张 r 4 张 b
有以下 3 种组合：
rr bb rb

1.B，C 全是一种颜色
B C A
bb.rr bb.rr

2.
B C A
bb rr bb/RR/BR,=>A:BR
rr bb =>A:BR

3.
B C A
BR BB RR/BR, =>A:BR
//推出 A:BR 的原因，
//如果 A 是 RR，
//那么，当 ABC 都说不知道后，B 最后应该知道自己是 BR 了。
//因为 B 不可能 是 RR 或 BB。

4.
B C A
BR BR BB/RR/BR
//推出 A:BR 的原因
//如果，A 是 BB，那么 B=>BR/RR，
//如果 B=>RR,那么一开始，C 就该知道自己是 BR 了。
//如果 B=>BR,

//最后，也还是推出=>A:BR
//至于程序，暂等高人。

第 23 题：
用最简单， 最快速的方法计算出下面这个圆形是否和正方形相交。"
3D 坐标系 原点(0.0,0.0,0.0)
圆形:
半径 r = 3.0
圆心 o = (*.*, 0.0, *.*)

正方形:
4 个角坐标;
1:(*.*, 0.0, *.*)
2:(*.*, 0.0, *.*)
3:(*.*, 0.0, *.*)
4:(*.*, 0.0, *.*)
*/

参考：
http://www.cssdn.net/thread-10486-1-2.html

第 24 题：

1.反转链表

2.合并链表。

pPrev<-pNode<-pNext

```
ListNode* ReverseIteratively(ListNode* pHead)
{
        ListNode* pReversedHead = NULL;
        ListNode* pNode = pHead;
        ListNode* pPrev = NULL;
        while(pNode != NULL)                //pNode<=>m
        {
                ListNode* pNext = pNode->m_pNext;           //n 保存在 pNext 下

                //如果 pNext 指为空，则当前结点 pNode 设为头。
                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;
}
```

或者，这样写：

head->next -> p-> q

```
template<class T>
Node<T>* Reverse(Node<T>* head)
{
  p=head->next;
  while(p)
  {
      q=p->next;                //p->next 先保存在 q 下
       p->next=head->next;   //p 掉头指向 head->next
      head->next=p;             //p 赋给 head->next
      p=q;                       //q 赋给 p。
                                  //上俩行即，指针前移嘛...
  }
  return head;
```

第 25 题：
写一个函数,它的原形是 int continumax(char *outputstr,char *intputstr)
功能：
在字符串中找出连续最长的数字串，并把这个串的长度返回，
并把这个最长数字串付给其中一个函数参数 outputstr 所指内存。
例如："abcd12345ed125ss123456789"的首地址传给 intputstr 后，函数将返回 9，
outputstr 所指的值为 123456789

//leeyunce
这个相对比较简单，思路不用多说，跟在序列中求最小值差不多。未经测试。有错误欢迎指出。

```
int continumax(char *outputstr, char *intputstr)
{
    int i, maxlen = 0;
    char * maxstr = 0;

    while (true)
    {
        while (intputstr && (*intputstr<'0' || *intputstr>'9')) //skip all non-digit characters
        {
            intputstr++;
        }

        if (!intputstr) break;

        int count = 0;
        char * tempstr = intputstr;
        while (intputstr && (*intputstr>='0' && *intputstr<='9')) //OK, these characters are
digits
        {
            count++;
            intputstr++;
        }
        if (count > maxlen)
        {
            maxlen = count;
            maxstr = tempstr;
        }
    }

    for (i=0; i<maxlen; i++)
    {
        outputstr[i] = maxstr[i];
```

```
    }

    outputstr[i] = 0;

    return maxlen;
}
```

26.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)。

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。

```
#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 --;
        }
    }
}
```

27.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。

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。

```
int jump_sum(int n)    //递归版本
{
    assert(n>0);
    if (n == 1 || n == 2) return n;
    return jump_sum(n-1)+jump_sum(n-2);
}

int jump_sum(int n) //迭代版本
{
    assert(n>0);
    if (n == 1 || n == 2) return n;

    int an, an_1=2, an_2=1;
    for (; n>=3; n--)
    {
        an = an_2 + an_1;
        an_2 = an_1;
        an_1 = an;
    }
    return an;
}
```

28.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……

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：

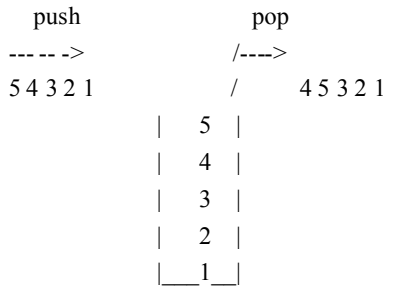
```
////////////////////////////////////////////////////////////////////////
// Get how many 1s in an integer's binary expression
////////////////////////////////////////////////////////////////////////
int NumberOf1_Solution2(int i)
{
	int count = 0;
	unsigned int flag = 1;
	while(flag)
	{
		if(i & flag)
			count ++;

		flag = flag << 1;
	}

	return count;
}
```

29.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。

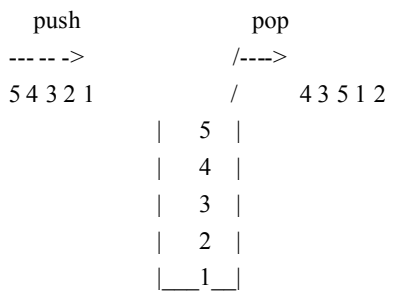
如果我们希望 pop 的数字正好是栈顶数字，直接 pop 出栈即可；
如果希望 pop 的数字目前不在栈顶，我们就到
push 序列中还没有被 push 到栈里的数字中去搜索这个数字，
并把在它之前的所有数字都 push 进栈。
如果所有的数字都被 push 进栈仍然没有找到这个数字，表明该序列不可能是一个 pop 序列。

//////////////////////////////////////
我们来着重分析下此题：
push 序列已经固定，



1.要得到 4 5 3 2 1 的 pop 序列，即 pop 的顺序为 4->5->3->2->1
首先，要 pop4，可让 push5 之前，pop4，然后 push5，pop5
然后发现 3 在栈顶，直接 pop 3..2..1

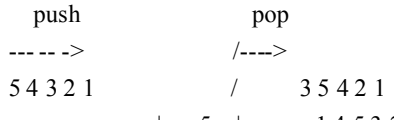
2.再看一序列，



想得到 4 3 5 1 2 的 pop 序列，是否可能?　　4->3->5->1->2
同样在 push5 之前，push 了 4 3 2 1，pop4，pop 3，然后再 push 5，pop5
　　2
再看栈中的从底至上是 1 ，由于 1 2 已经在栈中，所以只能先 pop2，才能 pop1。
所以，很显然，不可能有 4 3 5 1 2 的 pop 序列。

所以上述那段注释的话的意思，即是，
如果，一个元素在栈顶，直接 pop 即可。如果它不在栈顶，那么从 push 序列中找这个元素找到，那么 push 它，然后再 pop 它。否则，无法在 那个顺序中 pop。
//////////////////////////////////////

//////////////////////////////////////
push 序列已经固定，



3 5 4 2 1　//可行
1 4 5 3 2　//亦可行，不知各位，是否已明了题意?:)..

| 4 |
| 3 |
| 2 |
|__1__|

//
今早我也来了，呵。
昨晚，后来，自个又想了想，要怎么才能 pop 想要的一个数列?
push 序列已经固定，
  push pop
--- -- -> /---->
5 4 3 2 1 / 5 4 3 2 1
| 5 |
| 4 |
| 3 |
| 2 |
|__1__|

比如，当栈中已有数列 2
1
而现在我随机 要 pop4，一看，4 不在栈中，再从 push 序列中，
正好，4 在 push 队列中，push4 进栈之前，还得把 4 前面的数，即 3 先 push 进来，。
好，现在，push 3, push 4，然后便是想要的结果：pop 4。

所以，当我要 pop 一个数时，先看这个数 在不在已经 push 的 栈顶，如果，在，好，直接 pop 它。
如果，不在，那么，从 push 序列中，去找这个数，找到后，push 它进栈，
如果 push 队列中它的前面还有数，那么 还得把它前面数，先 push 进栈。
如果铺设队列中没有这个数，那当然 就不是存在这个 pop 结果了。

不知，我说明白了没?:）.接下来，给一段，参考程序：

```
//有误之处，恳请指正。July、2010/11/09。:)。
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;
}
```

30.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。

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);

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;
}

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。

各位，不妨讨论下，更好的解决办法。:)...

31.华为面试题：

一类似于蜂窝的结构的图，进行搜索最短路径（要求 5 分钟）

基于虚信道路由算法

为了在六角形蜂窝网络设计最短路径的算法，我们使用虚信道技术来避免死锁。

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

算法 2（Double XY 算法）：

输入：当前节点坐标（Xcurrent，Ycurrent，Zcurrent），目的节点坐标（Xdest，Ydest，Zdest），节点类型 NodeFlag 以及源节点 Z 向坐标 Zsource。

输出：选择的输出通道 Channel。

过程：

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

```
Xoffset := Xdest - Xcurrent;
Yoffset := Ydest - Ycurrent;
Zoffset := Zdest - Zcurrent;
Zsrcoffset := Zsource - Zdest;
if (Zoffset ≥ 0 and
( Zsrcoffset > 0 or Zsrcoffset = 0 )) then
if (NodeFlag = Black) then

if (Xoffset > 0) then
   Channel := X0+;
if (Yoffset > 0) then
   Channel := Y0+;
if (Zoffset > 0) then
   Channel := Z+;
if (NodeFlag = White) then

if (Xoffset < 0) then
   Channel := X0－;
if (Yoffset < 0) then
   Channel := Y0－;
if (Zoffset ≤ 0 and Zsrcoffset < 0) then
if (NodeFlag = White) then

if (Xoffset < 0) then
   Channel := X1－;
if (Yoffset < 0) then
   Channel := Y1－;
if (Zoffset < 0) then
   Channel := Z－;
```

```
if (NodeFlag = Black) then
if (Xoffset > 0) then
   Channel := X1+;
if (Yoffset > 0) then
   Channel := Y1+;
if (Xoffset = 0 and Yoffset = 0 and Zoffet = 0) then
   Channel := internal;
```

第 31 题，完。

32.

有两个序列 a,b，大小都为 n,序列元素的值任意整数，无序；
要求：通过交换 a,b 中的元素，使[序列 a 元素的和]与[序列 b 元素的和]之间的差最小。
例如:

```
var a=[100,99,98,1,2, 3];
var b=[1, 2, 3, 4,5,40];
```

求解思路：

当前数组 a 和数组 b 的和之差为

A = sum(a) - sum(b)

a 的第 i 个元素和 b 的第 j 个元素交换后，a 和 b 的和之差为

A' = sum(a) - a[i] + b[j] - （sum(b) - b[j] + a[i])

= sum(a) - sum(b) - 2 (a[i] - b[j])

= A - 2 (a[i] - b[j])

设 x = a[i] - b[j]

|A| - |A'| = |A| - |A-2x|

假设 A > 0,

当 x 在 (0,A)之间时，做这样的交换才能使得交换后的 a 和 b 的和之差变小，
x 越接近 A/2 效果越好,

如果找不到在(0,A)之间的 x，则当前的 a 和 b 就是答案。

所以算法大概如下：

在 a 和 b 中寻找使得 x 在(0,A)之间并且最接近 A/2 的 i 和 j，交换相应的 i 和 j 元素，
重新计算 A 后，重复前面的步骤直至找不到(0,A)之间的 x 为止。

//

算法

1. 将两序列合并为一个序列，并排序，为序列 Source
2. 拿出最大元素 Big，次大的元素 Small
3. 在余下的序列 S[:-2]进行平分，得到序列 max，min
4. 将 Small 加到 max 序列，将 Big 加大 min 序列，重新计算新序列和，和大的为 max，小的为 min。

//

```
def mean( sorted_list ):
    if not sorted_list:
        return (([],[]))

    big = sorted_list[-1]
    small = sorted_list[-2]
    big_list, small_list = mean(sorted_list[:-2])

    big_list.append(small)
    small_list.append(big)

    big_list_sum = sum(big_list)
    small_list_sum = sum(small_list)

    if big_list_sum > small_list_sum:
        return ( (big_list, small_list))
    else:
        return (( small_list, big_list))

tests = [   [1,2,3,4,5,6,700,800],
            [10001,10000,100,90,50,1],
            range(1, 11),
            [12312, 12311, 232, 210, 30, 29, 3, 2, 1, 1]
            ]
for l in tests:
    l.sort()
    print
    print "Source List:\t", l
    l1,l2 = mean(l)
    print "Result List:\t", l1, l2
    print "Distance:\t", abs(sum(l1)-sum(l2))
    print '-*'*40
```

输出结果
Source List: [1, 2, 3, 4, 5, 6, 700, 800]
Result List: [1, 4, 5, 800] [2, 3, 6, 700]
Distance: 99
-*

Source List: [1, 50, 90, 100, 10000, 10001]
Result List: [50, 90, 10000] [1, 100, 10001]
Distance: 38
-*

Source List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Result List: [2, 3, 6, 7, 10] [1, 4, 5, 8, 9]
Distance: 1
-*

Source List: [1, 1, 2, 3, 29, 30, 210, 232, 12311, 12312]
Result List: [1, 3, 29, 232, 12311] [1, 2, 30, 210, 12312]
Distance: 21

33.

实现一个挺高级的字符匹配算法：
给一串很长字符串，要求找到符合要求的字符串，例如目的串：123
1******3***2 ,12*****3 这些都要找出来
其实就是类似一些和谐系统。。。。。

分析：
自然匹配就是对待匹配的每个字符挨个匹配
设你的待匹配字串长度位 n,模式字符串长度位 m.
对于待匹配字符串中的任意一个字符最坏情况下要匹配 m 次，也就是说这个字符不在模式字符串中。
所以最坏情况下总共是 m*n 此匹配，时间复杂度就是 O(m*n)

倘若使用 hash 表对待字符串进行 hash 处理 O(n)的时间复杂度，那么对于模式字符串中的任意字符，
仅需一次 hash 判断就可以得知是否在待匹配字符串中出现。
最坏仅需 m 次就可以得到结果。时间复杂度为 O(m)或者 O(n);

与此题相类似：
就是给一个很长的字符串 str 还有一个字符集比如{a,b,c} 找出 str 里包含{a,b,c}的最短子串。
要求 O(n)?

比如，字符集是 a,b,c，字符串是 abdcaabcx，则最短子串为 abc。

用两个变量 front rear 指向一个的子串区间的头和尾
用一个 int cnt[255]={0}记录当前这个子串里 字符集 a,b,c 各自的个数，
一个变量 sum 记录字符集里有多少个了

rear 一直加，更新 cnt[]和 sum 的值，直到 sum 等于字符集个数
然后 front++,直到 cnt[]里某个字符个数为 0，这样就找到一个符合条件的字串了

继续前面的操作，就可以找到最短的了。
//还有没有人，对此题了解的比较深的？望 也多阐述下...:)。

34.实现一个队列。
队列的应用场景为：
一个生产者线程将 int 类型的数入列，一个消费者线程将 int 类型的数出列

生产者消费者线程演示
一个生产者线程将 int 类型的数入列，一个消费者线程将 int 类型的数出列

```
#include <windows.h>
#include <stdio.h>
#include <process.h>
#include <iostream>
#include <queue>
using namespace std;
HANDLE ghSemaphore;     //信号量
const int gMax = 100; //生产(消费)总数
std::queue<int> q;          //生产入队,消费出队
//生产者线程
unsigned int __stdcall producerThread(void* pParam)
{
    int n = 0;
    while(++n <= gMax)
    {
        //生产
        q.push(n);
        cout<<"produce "<<n<<endl;
        ReleaseSemaphore(ghSemaphore, 1, NULL); //增加信号量
        Sleep(300);//生产间隔的时间,可以和消费间隔时间一起调节
    }
    _endthread(); //生产结束
```

```
    return 0;
}
//消费者线程
unsigned int __stdcall customerThread(void* pParam)
{
    int n = gMax;
    while(n--)
    {
        WaitForSingleObject(ghSemaphore, 10000);
        //消费
        q.pop();
        cout<<"custom    "<<q.front()<<endl;
        Sleep(500);//消费间隔的时间,可以和生产间隔时间一起调节
    }
    //消费结束
    CloseHandle(ghSemaphore);
    cout<<"working end."<<endl;
    _endthread();
    return 0;
}
void threadWorking()
{
    ghSemaphore = CreateSemaphore(NULL, 0, gMax, NULL); //信号量来维护线程同步

    cout<<"working start."<<endl;
    unsigned threadID;
    HANDLE handles[2];
    handles[0] = (HANDLE)_beginthreadex(
                        NULL,
                        0,
                        producerThread,
                        nullptr,
                        0,
                        &threadID);
    handles[1] = (HANDLE)_beginthreadex(
                        NULL,
                        0,
                        customerThread,
                        nullptr,
                        0,
                        &threadID);
    WaitForMultipleObjects(2, handles, TRUE, INFINITE);
}
int main()
```

```
{
    threadWorking();
    getchar();
    return 0;
}
```

35.

求一个矩阵中最大的二维矩阵(元素和最大).如:

1 2 0 3 4

2 3 4 5 1

1 1 5 3 0

中最大的是:

4 5

5 3

要求:(1)写出算法;(2)分析时间复杂度;(3)用 C 写出关键代码

此第 35 题与第 3 题相类似，一个是求最大子数组和，一个是求最大子矩阵和。

3.求子数组的最大和

题目：

输入一个整形数组，数组里有正数也有负数。

数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。

求所有子数组的和的最大值。要求时间复杂度为 O(n)。

例如输入的数组为 1, -2, 3, 10, -4, 7, 2, -5，和最大的子数组为 3, 10, -4, 7, 2，

```
int maxSum(int* a, int n)
{
  int sum=0;
  int b=0;

  for(int i=0; i<n; i++)
  {
    if(b<=0)                //此处修正下，把 b<0 改为 b<=0
      b=a[i];
    else
      b+=a[i];
    if(sum<b)
      sum=b;
  }
  return sum;
}
```

//
解释下：
例如输入的数组为 1, -2, 3, 10, -4, 7, 2, -5，
那么最大的子数组为 3, 10, -4, 7, 2，
因此输出为该子数组的和 18

所有的东西都在以下俩行，
即：
  b：0 1 -1 3 13 9 16 18 7
sum：0 1 1 3 13 13 16 18 18

其实算法很简单，当前面的几个数，加起来后，b<0 后，
把 b 重新赋值，置为下一个元素，b=a[i]。
当 b>sum，则更新 sum=b;
若 b<sum，则 sum 保持原值，不更新。:)。July、10/31。
//

现在回到我们的最初的最大子矩阵的问题,
 假设最大子矩阵的结果为从第 r 行到 k 行、从第 i 列到 j 列的子矩阵，
如下所示(ari 表示 a[r][i],假设数组下标从 1 开始)：

| a11 …… a1i ……a1j ……a1n |
| a21 …… a2i ……a2j ……a2n |
…..

| ar1 …… ari ……arj ……arn | 第 r 行 . . .
…….. |
V
| ak1 …… aki ……akj ……akn | 第 k 行 . . .

…..
| an1 …… ani ……anj ……ann |

 那么我们将从第 r 行到第 k 行的每一行中相同列的加起来，可以得到一个一维数组如下：
 (ar1+……+ak1, ar2+……+ak2, ……,arn+……+akn)
 由此我们可以看出最后所求的就是此一维数组的最大子断和问题，
到此我们已经将问题转化为上面的已经解决了的问题了。

ar1
..
ak1
注，是竖直方向，相加

```
//有误之处，肯定指正。:)。
    #include <iostream>
 2 using namespace std;
 3
 4 int  ** a;
 5 int  **sum;

 6 int  max_array(int *a,int n)
 7 {
 8              int *c = new int [n];
 9              int i =0;
10              c[0] = a[0];
11              for(i=1;i<n;i++)
12              {
13                              if(c[i-1]<0)
14                                              c[i] = a[i];
15                              else
16                                              c[i] = c[i-1]+a[i];
17              }
18              int max_sum = -65536;
19              for(i=0;i<n;i++)
20                              if(c[i]>max_sum)
21                                              max_sum = c[i];
22              delete []c;
23              return max_sum;
24
25 }

26 int  max_matrix(int n)
27 {
28              int i =0;
29              int j = 0;
30              int max_sum = -65535;
31              int * b = new int [n];
32
33              for(i=0;i<n;i++)
34              {
35                              for(j=0;j<n;j++)
36                                              b[j]= 0;
37                              for(j=i;j<n;j++)
//把数组从第 i 行到第 j 行相加起来保存在 b 中，在加时，自底向上，首先计算行间隔(j-i)等
于 1 的情况，
//然后计算 j-i 等于 2 的情况，一次类推,在小间隔的基础上一次累加，避免重复计算
```

```
38                      {
39                              for(int k =0;k<=n;k++)
40                                      b[k] += a[j][k];
41                              int sum = max_array(b,n);
42                              if(sum > max_sum)
3                                       max_sum = sum;
44                      }
45              }
46              delete []b;
47              return max_sum;
48 }

49 int main()
50 {
51              int n;
52              cin >> n;
53
54              a = new int *[n];
55              sum = new int *[n];
56              int i =0;
57              int j =0;
58              for(i=0;i<n;i++)
59              {
60                      sum[i] = new int[n];
61                      a[i] = new int[n];
62                      for(j=0;j<n;j++)
63                      {
64                              cin>>a[i][j];
65                              sum[i][j] =0 ;
                                //sum[r][k]表示起始和结尾横坐标分别为 r,k 时的最大子矩阵
66                              //sum[r][k] = max{sum (a[i][j]):r<=i<=k}:0<=k<=n-1
67                      }
68              }
69              /*
70              int b[10]={31,-41,59,26,-53,58,97,-93,-23,84};
71              cout << max_array(b,10) << endl;
72              */
73              cout << max_matrix(n);
74 }
```

我们再来分析下这段，代码，为了让你真正弄透它。:)。
//July，11.14.
求最大子矩阵，我们先按给的代码的思路来：
1.求最大子矩阵，我们把矩阵中，每一竖直方向的排列，看做一个元素。
所以，矩阵就转化成了我们熟悉的一维数组。

即以上矩阵，相当于：
a[1->n][1] a[1->n][2] ... a[1->n][i] .. a[1->n][j] .. a[1->n][n]
1->n 表示竖直方向，同一列的元素相加。

那么，假设最大子矩阵，是在第 r 行->第 k 行，所有元素的和。
| ar1 …… ari ……arj ……arn |
| . . . . |
| . . . . |
| ak1 …… aki ……akj ……akn |
所以题目就转化成了类似第 3 题的思路。

2.先把这第 r 行->k 行的列的元素，分别相加。
即这段代码：

```
26 int  max_matrix(int n)
27 {
28              int i =0;
29              int j = 0;
30              int max_sum = -65535;
31              int * b = new int  [n];
32
33              for(i=0;i<n;i++)
34              {
35                      for(j=0;j<n;j++)
36                              b[j]= 0;
37                      for(j=i;j<n;j++)
//把数组从第 i 行到第 j 行相加起来保存在 b 中，在加时，自底向上，
//首先计算行间隔(j-i)等于 1 的情况，然后计算 j-i 等于  2 的情况，
//一次类推,在小间隔的基础上一次累加，避免重复计算
38                      {
39                              for(int k =0;k<=n;k++)
40                                      b[k] += a[j][k];
41                              int sum = max_array(b,n);
42                              if(sum > max_sum)
3                                       max_sum = sum;
44                      }
45              }
46              delete []b;
47              return max_sum;
```

```
48 }
```

咱们，来稍微分析下，
即，求这段矩阵的和

```
i 行  a[i][1] a[r][2] ... a[r][k] .. a[r][n]
 | a[i+1][1]
...
 v a[j-1][1]
j 行  a[j][1] a[j][2] ... a[j][k] .. a[j][n]
```

```
for(i=0;i<n;i++)   //第 i 行
 {
   for(j=0;j<n;j++)   //第 j 行
      b[j]=0;              //先把 b[j]初始化为 0
   for(j=i;j<n;j++)   //第 i 行->第 j 行     固定行
    {
      for(int k=0;k<=n;k++)     //从上而下，列元素相加
          b[k] += a[j][k];
          //相加之后，调用上述的求和函数 max_array(b,n)即可。
          int sum=max_array(b,n);
          if(sum>max_sum)
              max_sum=sum;     //sum->b 的结果
    }
 }
  delete []b;
  return max_sum;
```

至于求和 max_array(int* a,int n)函数，

```
 6 int  max_array(int *a,int n)
 7 {
 8              int *c = new int [n];
 9              int i =0;
10              c[0] = a[0];
11              for(i=1;i<n;i++)
12              {
13                        if(c[i-1]<0)
14                                  c[i] = a[i];
15                        else
16                                  c[i] = c[i-1]+a[i];
17              }
```

```
18              int max_sum = -65536;
19              for(i=0;i<n;i++)
20                      if(c[i]>max_sum)
21                              max_sum = c[i];
22              delete []c;
23              return max_sum;
24
25 }
```

代码，则与这个差不多：

```
int maxSum(int* a, int n)
{
  int sum=0;
  int b=0;

  for(int i=0; i<n; i++)
  {
    if(b<0)                 //其实，此处 b<0，亦可。无需 b<=0.
      b=a[i];
    else
      b+=a[i];
    if(sum<b)
      sum=b;
  }
  return sum;
}
```

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

所有的东西都在以下俩行，
即：

```
  b：0   1   -1   3   13    9   16   18   7
sum：0   1    1   3   13   13   16   18   18
```

最后，矩阵之和，在 main 函数里，调用这个函数 cout << max_matrix(n);输出即可。
有误之处，欢迎指正。

另外，调换俩个 for 循环的顺序，是否更精准?

```
for(i=0;i<n;i++)   //第 i 行
 {
```

```
    for(j=0;j<n;j++)   //第 j 行
        b[j]=0;              //先把 b[j]初始化为 0
    for(int k=0;k<=n;k++)    //固定一列，然后 0 列->k 列—>n 列，逐级+。
     {
        for(j=i;j<n;j++)   //第 i 行->第 j 行->第 n 行+ +++
         //调换俩个 for 循环的顺序，是否更精准?

            b[k] += a[j][k];
            //相加之后，调用上述的求和函数 max_array(b,n)即可。
         int sum=max_array(b,n);
        if(sum>max_sum)
        max_sum=sum;     //sum->b 的结果
      }
  }
 delete []b;
 return max_sum;
```

完。:)

36.引用自网友：longzuo

谷歌笔试：
n 支队伍比赛，分别编号为 0，1，2。。。。n-1，已知它们之间的实力对比关系，
存储在一个二维数组 w[n][n]中，w[i][j] 的值代表编号为 i，j 的队伍中更强的一支。

所以 w[i][j]=i 或者 j，现在给出它们的出场顺序，并存储在数组 order[n]中，
比如 order[n] = {4,3,5,8,1.....}，那么第一轮比赛就是 4 对 3， 5 对 8。.....

胜者晋级，败者淘汰，同一轮淘汰的所有队伍排名不再细分，即可以随便排，
下一轮由上一轮的胜者按照顺序，再依次两两比，比如可能是 4 对 5,直至出现第一名

编程实现，给出二维数组 w，一维数组 order 和 用于输出比赛名次的数组 result[n]，求出 result。

```
#include <stdio.h>
#include <list>
#include <iostream>

void raceResult(int** w, int* order, int* result, int n)
{
    std::list<int> winer;
```

```
int count = n;
while(n)
{
	winer.push_front(order[--n]);
}

int resultNum = count - 1;
int nFirst, nSecond;
int round = 1;
while(winer.size() > 1)
{
	//一轮开始
	std::cout<<std::endl<<"round "<<round++<<std::endl;
	std::list<int>::iterator it = winer.begin();
	while (it != winer.end())
	{
		nFirst = *it;
		if (++it == winer.end())
		{
			//轮空
			std::cout<<nFirst<<" rest this round"<<std::endl;
		}
		else
		{
			nSecond = *it;
			int nWiner = *((int*)w + count * nFirst + nSecond);
			if (nWiner  == nFirst)
			{
				it = winer.erase(it);
				result[resultNum--] = nSecond;
				std::cout<<nFirst<<" kick out "<<nSecond<<std::endl;
			}
			else
			{
				it = winer.erase(--it);
				result[resultNum--] = nFirst;
				++it;
				std::cout<<nSecond<<" kick out "<<nFirst<<std::endl;
			}
		}
	}
}
if (winer.size() == 1)
{
```

```
            result[0] = winer.front();
        }
        std::cout<<std::endl<<"final result: ";
        int nPlace = 0;
        while(nPlace < count)
        {
            std::cout<<std::endl<<result[nPlace++];
        }
}


void test()
{
        //team 2>team 1>team 3>team 0>team 4>team 5
        int w[6][6] = {
            0,1,2,3,0,0,
            1,1,2,1,1,1,
            2,2,2,2,2,2,
            3,1,2,3,3,3,
            0,1,2,3,4,5
        };
        int order[6] = {1,3,4,2,0,5};
        int result[6] = {-1};
        raceResult((int**)w, order, result, 6);
        getchar();
}
//自己加上主函数，测试了下，结果竟正确..

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

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

round 1
1 kick out 3
2 kick out 4
0 kick out 5

round 2
2 kick out 1
0 rest this round
```

round 3
2 kick out 0

final result:
2
0
1
5
4
3
//

37.

有 n 个长为 m+1 的字符串，
如果某个字符串的最后 m 个字符与某个字符串的前 m 个字符匹配，则两个字符串可以联接，
问这 n 个字符串最多可以连成一个多长的字符串，如果出现循环，则返回错误。

恩，好办法
引用 5 楼 hblac 的回复:
37. 把每个字符串看成一个图的顶点，两个字符串匹配就连一条有向边。相当于判断一个有向图
是否有环以及求它的直径

38.

百度面试：
1.用天平（只能比较，不能称重）从一堆小球中找出其中唯一一个较轻的，
使用 x 次天平，最多可以从 y 个小球中找出较轻的那个，求 y 与 x 的关系式

2.有一个很大很大的输入流，大到没有存储器可以将其存储下来，而且只输入一次，如何从这个输入
流中随机取得 m 个记录
3.大量的 URL 字符串，如何从中去除重复的，优化时间空间复杂度

38,1. y=3^x
38,2. 每次输入一个记录时，随机产生一个 0 到 1 之间的随机数，
用这些随机数维护一个大小为 m 的堆，即可。
38,3.大量的 URL 字符串，如何从中去除重复的，优化时间空间复杂度
这道题见过了，解法是构造一个 hash 函数，把 url 适当散列到若干个，
比如 1000 个小文件中，然后在每个小文件中去除重复的 url，再把他们合并。

原理是相同的 url，hash 之后的散列值仍然是相同的。
38,1
samsho2
我对天平称重这道题的理解是：
每次将球分成三堆，尽可能让三堆球一样多或者让其中一堆多或者少一个。
球数 y 和沉重次数 x 的关系是：
y=1 =》x=0；（显然）
y=2 =》x=1；（显然）
y=3 =》x=2；（显然）
如果 y>3，也是将球分为三堆，记为 A 堆、B 堆、C 堆
if y=3k（k>1）
　称重一次，就可以判断瑕疵球在哪堆，从而使得球数降为 k 个；
if y=3k+1（k>=1） 假设 C 堆多放 1 球，A 堆和 B 堆进行称重
　称重一次，就可以判断瑕疵球在哪堆，
　if A == B ，瑕疵球在 C 堆，从而使得球数降为 k+1 个；
　else 瑕疵球在轻的堆，从而使得球数降为 k 个；
if y=3k-1（k>=2） 假设 C 堆少放 1 球，A 堆和 B 堆进行称重
　称重一次，就可以判断瑕疵球在哪堆，
　if A == B ，瑕疵球在 C 堆，从而使得球数降为 k 个；
　else 瑕疵球在轻的堆，从而使得球数降为 k-1 个；

利用以上过程反复，可得结果。
最后的次数 x = log3（y），可能在 y 哪里还需要做点什么修正。但复杂度就是 log y

38，1.
用天平（只能比较，不能称重）从一堆小球中找出其中唯一一个较轻的，
使用 x 次天平 最多可以从 y 个小球中找出较轻的那个，求 y 与 x 的关系式

hengchun11
用天平比较二边放一样的球数：有三种可能性
第一种 左边重 说明较轻的在右边；
第二种 右边重 说明较轻的在左边；
第三种 一样重 说明较轻的不在这里面；
以上有三种可能性，在称 x=1 的情况下，说明 y=2 是可以称出来的 y=3，也是可以的；y=4 就不行

了
所以 我觉得 分成三部分来称 就可以称出最多的球
x=1 y=3
x=2 y=9
x=3 y=27
可以得出 y=3 的 x 次方

39.

网易有道笔试：

(1).

求一个二叉树中任意两个节点间的最大距离，

两个节点的距离的定义是 这两个节点间边的个数，

比如某个孩子节点和父节点间的距离是 1，和相邻兄弟节点间的距离是 2，优化时间空间复杂度。

(2).

求一个有向连通图的割点，割点的定义是，如果除去此节点和与其相关的边，

有向图不再连通，描述算法。

先看第 39 题的第 1 小问，

求一个二叉树中任意俩个结点之间的距离。

以前自个，写的，求二叉树中节点的最大距离...

```
void traversal_MaxLen(NODE* pRoot)
{
    if(pRoot == NULL)
    {
        return 0;
    };

    if(pRoot->pLeft == NULL)
    {
        pRoot->MaxLeft = 0;
    }
    else                                        //若左子树不为空
    {
        int TempLen = 0;
        if(pRoot->pLeft->MaxLeft > pRoot->pLeft->MaxRight)
          //左子树上的，某一节点，往左边大，还是往右边大
        {
            TempLen+=pRoot->pLeft->MaxLeft;
        }
        else
        {
            TempLen+=pRoot->pLeft->MaxRight;
        }
        pRoot->nMaxLeft = TempLen + 1;
        traversal_MaxLen(NODE* pRoot->pLeft);
        //此处，加上递归
    }
```

```
    if(pRoot->pRigth == NULL)
    {
        pRoot->MaxRight = 0;
    }
    else                                        //若右子树不为空
    {
        int TempLen = 0;
        if(pRoot->pRight->MaxLeft > pRoot->pRight->MaxRight)
        //右子树上的，某一节点，往左边大，还是往右边大
        {
            TempLen+=pRoot->pRight->MaxLeft;
        }
        else
        {
            TempLen+=pRoot->pRight->MaxRight;
        }
        pRoot->MaxRight = TempLen + 1;
        traversal_MaxLen(NODE* pRoot->pRight);
        //此处，加上递归
    }

   if(pRoot->MaxLeft + pRoot->MaxRight > 0)
    {
        MaxLength=pRoot->nMaxLeft + pRoot->MaxRight;
    }
}


// 数据结构定义
    struct NODE
    {
        NODE* pLeft;                // 左子树
        NODE* pRight;               // 右子树
        int nMaxLeft;               // 左子树中的最长距离
        int nMaxRight;              // 右子树中的最长距离
        char chValue;               // 该节点的值
    };

    int nMaxLen = 0;

    // 寻找树中最长的两段距离
    void FindMaxLen(NODE* pRoot)
    {
        // 遍历到叶子节点，返回
```

```
if(pRoot == NULL)
{
	return;
}

// 如果左子树为空，那么该节点的左边最长距离为 0
if(pRoot -> pLeft == NULL)
{
	pRoot -> nMaxLeft = 0;
}

// 如果右子树为空，那么该节点的右边最长距离为 0
if(pRoot -> pRight == NULL)
{
	pRoot -> nMaxRight = 0;
}

// 如果左子树不为空，递归寻找左子树最长距离
if(pRoot -> pLeft != NULL)
{
	FindMaxLen(pRoot -> pLeft);
}

// 如果右子树不为空，递归寻找右子树最长距离
if(pRoot -> pRight != NULL)
{
	FindMaxLen(pRoot -> pRight);
}

// 计算左子树最长节点距离
if(pRoot -> pLeft != NULL)
{
	int nTempMax = 0;
	if(pRoot -> pLeft -> nMaxLeft > pRoot -> pLeft -> nMaxRight)
	{
		nTempMax = pRoot -> pLeft -> nMaxLeft;
	}
	else
	{
		nTempMax = pRoot -> pLeft -> nMaxRight;
	}
	pRoot -> nMaxLeft = nTempMax + 1;
}
```

```
        // 计算右子树最长节点距离
        if(pRoot -> pRight != NULL)
        {
            int nTempMax = 0;
            if(pRoot -> pRight -> nMaxLeft > pRoot -> pRight -> nMaxRight)
            {
                nTempMax = pRoot -> pRight -> nMaxLeft;
            }
            else
            {
                nTempMax = pRoot -> pRight -> nMaxRight;
            }
            pRoot -> nMaxRight = nTempMax + 1;
        }

        // 更新最长距离
        if(pRoot -> nMaxLeft + pRoot -> nMaxRight > nMaxLen)
        {
            nMaxLen = pRoot -> nMaxLeft + pRoot -> nMaxRight;
        }
}
//很明显，思路完全一样，但书上 给的这段代码 更规范！:)。
```

zhoulei0907

```
/*
 * return the depth of the tree
 */
int get_depth(Tree *tree) {
    int depth = 0;
    if ( tree ) {
        int a = get_depth(tree->left);
        int b = get_depth(tree->right);
        depth = ( a > b ) ? a : b;
        depth++;
    }
    return depth;
}

/*
 * return the max distance of the tree
 */
int get_max_distance(Tree *tree) {
```

```
    int distance = 0;
    if ( tree ) {
        // get the max distance connected to the current node
        distance = get_depth(tree->left) + get_depth(tree->right);

        // compare the value with it's sub trees
        int l_distance = get_max_distance(tree->left);
        int r_distance = get_max_distance(tree->right);
        distance = ( l_distance > distance ) ? l_distance : distance;
        distance = ( r_distance > distance ) ? r_distance : distance;
    }
    return distance;
}
```

解释一下，get_depth 函数是求二叉树的深度，用的是递归算法：
一棵二叉树的深度就是它的左子树的深度和右子树的深度，两者的最大值加一。

get_max_distance 函数是求二叉树的最大距离，也是用递归算法：
首先算出经过根节点的最大路径的距离，其实就是左右子树的深度和；
然后分别算出左子树和右子树的最大距离，三者比较，最大值就是当前二叉树的最大距离了。

这个算法不是效率最高的，因为在计算二叉树的深度的时候存在重复计算。
但应该是可读性比较好的，同时也没有改变原有二叉树的结构和使用额外的全局变量。

July:
很好。那么，咱们再来 探讨下这个二叉树的最大距离问题。
计算一个二叉树的最大距离有两个情况:
  情况 A: 路径经过左子树的最深节点，通过根节点，再到右子树的最深节点。
  情况 B: 路径不穿过根节点，而是左子树或右子树的最大距离路径，取其大者。
只需要计算这两个情况的路径距离，并取其大者，就是该二叉树的最大距离。

简单的写下算法。
1.如果根结点，为空，当然 return 0；
2.如果左子树不为空，
    寻找左子树上最深的那个点（左深度）。
  否则，左子树为空
    不寻找。
    //即最大距离不通过根结点。
    //即最大距离为 maxLeft =左深度+1
3.如果右子树不为空
    寻找右子树上最深的那个点（右深度）。
  否则，右子树为空
    不寻找。
    //即最大距离不通过根结点。

//即最大距离为 maxRight =右深度+1
所以，最大的距离，即为
当有左，无右时，则最大距离 maxLen=maxLeft（左深度）+1　　//不过根结点
当有右，无左时，则最大距离 maxLen=maxRight（右深度）+1　　//不过根结点
当有左，也有右时，则最大距离 maxLen=maxLeft（左深度）+1 + maxRight（右深度）+1　　//过根结点

三者，比较，即得，最终的 maxLen。

然后么最后的问题就只剩，求左子树 maxLeft 或者右子树 maxRight 的深度问题。
求一个子树，如左子树的 maxLeft，即深度问题，
我们可以这么考虑，
左子树不为空，左子树上的，某一节点，往左边大，还是往右边大
往左边大，那么 maxLen 加上 往左边的距离，　即相当于搜索往深的那一边 左边 搜索
往右边大，那么 maxLen 加上 往右边的距离。　即相当于搜索往深的那一边 左边 搜索

好比 凿井一样，总要往更深的方向凿。
凿到某一个深度后，想下，是往左边一点凿，更好列，还是往右边一点点凿更好列。
总之，目的就是为了，凿到更大 的深度。
就是这个道理了。:)。

经过上述，我一番苦口婆心之后，再来看以下这段代码，是不是更加容易懂了。
:)....

```
void FindMaxLen(NODE* pRoot)
    {
        // 遍历到叶子节点，返回
        if(pRoot == NULL)
        {
            return;
        }

        // 如果左子树为空，那么该节点的左边最长距离为 0
        if(pRoot -> pLeft == NULL)
        {
            pRoot -> nMaxLeft = 0;
        }

        // 如果右子树为空，那么该节点的右边最长距离为 0
        if(pRoot -> pRight == NULL)
        {
            pRoot -> nMaxRight = 0;
        }
```

```
// 如果左子树不为空，递归寻找左子树最长距离
if(pRoot -> pLeft != NULL)
{
	FindMaxLen(pRoot -> pLeft);
}

// 如果右子树不为空，递归寻找右子树最长距离
if(pRoot -> pRight != NULL)
{
	FindMaxLen(pRoot -> pRight);
}

// 计算左子树最长节点距离
if(pRoot -> pLeft != NULL)
{
	int nTempMax = 0;
	if(pRoot -> pLeft -> nMaxLeft > pRoot -> pLeft -> nMaxRight)
	{
		nTempMax = pRoot -> pLeft -> nMaxLeft;
	}
	else
	{
		nTempMax = pRoot -> pLeft -> nMaxRight;
	}
	pRoot -> nMaxLeft = nTempMax + 1;
}

// 计算右子树最长节点距离
if(pRoot -> pRight != NULL)
{
	int nTempMax = 0;
	if(pRoot -> pRight -> nMaxLeft > pRoot -> pRight -> nMaxRight)
	{
		nTempMax = pRoot -> pRight -> nMaxLeft;
	}
	else
	{
		nTempMax = pRoot -> pRight -> nMaxRight;
	}
	pRoot -> nMaxRight = nTempMax + 1;
}

// 更新最长距离
if(pRoot -> nMaxLeft + pRoot -> nMaxRight > nMaxLen)
```

```
        {
            nMaxLen = pRoot -> nMaxLeft + pRoot -> nMaxRight;
        }
}
```

求左子树 maxLeft 或者右子树 maxRight 的深度问题，就涉及一个递归问题了。
即我们搜索 这个树的深度时，不一直就用着递归往下搜索么。

好比凿井，当我们试探性的是往左，还是往右，更深一点，
如果，能往右，那么递归 往右凿， //即只要右子树存在，那么不断的递归右子树，找最大深度。
如果，能往左，那么递归 往左凿。 //即只要左子树存在，那么不断的递归左子树，找最大深度。
这样，井是不是 已经凿 的很深了。
很享受，这种凿井的过程，
希望，我能与更多的人，一起来凿井，越凿越要往深处凿，凿的越深越好。

同时，把每一道题目，解释的越简单易懂，则是我的目标之一。
谢谢。:)

39.
网易有道笔试：
(2).
求一个有向连通图的割点，割点的定义是，如果除去此节点和与其相关的边，
有向图不再连通，描述算法。

网友回复，有误，指正：
求无向连通图的割点集
mysword
最简单的，删掉一个点然后判断连通性，不就可以了？ //这句话，道出了割点的定义。
BlueSky2008
可以更简单一些：
在深度优先树中，根结点为割点，当且仅当他有两个或两个以上的子树。
其余结点 v 为割点，当且仅当存在一个 v 的后代结点 s，s 到 v 的祖先结点之间没有反向边。

记发现时刻 dfn(v)为一个节点 v 在深度优先搜索过程中第一次遇到的时刻。
记标号函数 low(v) = min(dfn(v), low(s), dfn(w))
s 是 v 的儿子，(v,w)是反向边。

low(v) 表示从 v 或 v 的后代能追溯到的标号最小的节点。

则非根节点 v 是割点，当且仅当存在 v 的一个儿子 s，low(s) > = dfn(v)

40.百度研发笔试题

引用自：zp155334877

1)设计一个栈结构，满足一下条件：min，push，pop 操作的时间复杂度为 O(1)。

2)一串首尾相连的珠子(m 个)，有 N 种颜色(N<=10)，
设计一个算法，取出其中一段，要求包含所有 N 中颜色，并使长度最短。
并分析时间复杂度与空间复杂度。

3)设计一个系统处理词语搭配问题，比如说 中国 和人民可以搭配，
则中国人民 人民中国都有效。要求：

*系统每秒的查询数量可能上千次；
*词语的数量级为 10W；
*每个词至多可以与 1W 个词搭配

当用户输入中国人民的时候，要求返回与这个搭配词组相关的信息。

40.百度研发笔试题

引用自：zp155334877

1)设计一个栈结构，满足一下条件：min，push，pop 操作的时间复杂度为 O(1)。……

所以，此题的第 1 小题，即是借助辅助栈，保存最小值，
且随时更新辅助栈中的元素。
如先后，push 2 6 4 1 5
stack A stack B（辅助栈）

4: 5 1 //push 5,min=p->[3]=1 ^
3: 1 1 //push 1,min=p->[3]=1 | //此刻 push 进 A 的元素 1 小于 B 中栈顶元素 2
2: 4 2 //push 4,min=p->[0]=2 |
1: 6 2 //push 6,min=p->[0]=2 |
0: 2 2 //push 2,min=p->[0]=2 |

push 第一个元素进 A，也把它 push 进 B，
当向 Apush 的元素比 B 中的元素小， 则也 push 进 B，即更新 B。否则，不动 B，保存原值。
向栈 A push 元素时，顺序由下至上。
辅助栈 B 中，始终保存着最小的元素。

然后，pop 栈 A 中元素，5 1 4 6 2
A B ->更新
4: 5 1 1 //pop 5,min=p->[3]=1 |

3: 1 1 2 //pop 1,min=p->[3]=2 |
2: 4 2 2 //pop 4,min=p->[0]=2 |
1: 6 2 2 //pop 6,min=p->[0]=2 |
0: 2 2 NULL //pop 2,min=p->[0]=NULL v

当 pop A 中的元素小于 B 中栈顶元素时，则也要 pop B 中栈顶元素。

index 貌似错了，修正下，

所以，此题的第 1 小题，即是借助辅助栈，保存最小值，
且随时更新辅助栈中的元素。
如先后，push 2 6 4 1 5
 stack A stack B（辅助栈）

4: 5 1 //push 5,min=p->[3]=1 ^
3: 1 1 //push 1,min=p->[3]=1 | //此刻 push 进 A 的元素 1 小于 B 中栈顶元素 2
2: 4 2 //push 4,min=p->[0]=2 |
1: 6 2 //push 6,min=p->[0]=2 |
0: 2 2 //push 2,min=p->[0]=2 |

push 第一个元素进 A，也把它 push 进 B，
当向 Apush 的元素比 B 中的元素小， 则也 push 进 B，即更新 B。否则，不动 B，保存原值。
向栈 A push 元素时，顺序由下至上。
辅助栈 B 中，始终保存着最小的元素。

然后，pop 栈 A 中元素，5 1 4 6 2
  A B ->更新
4: 5 1 1 //pop 5,min=p->[3]=1 |
3: 1 1 2 //pop 1,min=p->[0]=2 |
2: 4 2 2 //pop 4,min=p->[0]=2 |
1: 6 2 2 //pop 6,min=p->[0]=2 |
0: 2 2 NULL //pop 2,min=NULL v

当 pop A 中的元素小于 B 中栈顶元素时，则也要 pop B 中栈顶元素。

2)一串首尾相连的珠子(m 个)，有 N 种颜色(N<=10)，
设计一个算法，取出其中一段……
2.就是给一个很长的字符串 str 还有一个字符集比如{a,b,c} 找出 str 里包含{a,b,c}的最短子串。
要求 O(n)?

比如，字符集是 a,b,c，字符串是 abdcaabcx，则最短子串为 abc

用两个变量 front，rear 指向一个的子串区间的头和尾
用一个 int cnt[255]={0}记录当前这个子串里 字符集 a,b,c 各自的个数，
一个变量 sum 记录字符集里有多少个了。

rear 一直加，更新 cnt[]和 sum 的值，直到 sum 等于字符集个数
然后 front++,直到 cnt[]里某个字符个数为 0，这样就找到一个符合条件的字串了

继续前面的操作，就可以找到最短的了。
http://www.gevgb.com/bbs/viewthread.php?tid=21&extra=page%3D1

3.
可以建立一个 RDF 文件，利用 protege 4.0。或者自己写的 RDF/XML 接口也行。
在其中建立两个“描述对象”一个是国家，一个是人民。 国家之下建立一个中国，
人民。并且在关系中建立一个关系：对每个存在的国家，都 have 人民。
这样，每当用户输入这两个词的时候，
就利用语义框架/接口来判断一下这两个词汇的关系，返回一个值即可。

--贝叶斯分类--
其实贝叶斯分类更实用一些。 可以用模式识别的贝叶斯算法，
写一个类并且建立一个词汇-模式的表。
这个表中每个模式，也就是每个词汇都设一个域，可以叫做 most-fitted word，
然后对这个分类器进行训练。

这个训练可以在初期设定一些关联词汇；也可以在用户每次正确输入查询的时候来训练。通过训练，每个单词对应出现概率最高的单词设到最适合域里面。
然后对于每个词，都返回最适合的单词。
（答案 V0.3 版完）

若你对以上答案，有任何问题，欢迎联系我：
My E-mail:
zhoulei0907@yahoo.cn
或者，直接回复于此帖上：
[推荐] [整理]算法面试：精选微软经典的算法面试 100 题[前 60 题]
http://topic.csdn.net/u/20101023/20/5652ccd7-d510-4c10-9671-307a56006e6d.html

作者声明：

向你的厚道致敬。谢谢。

July、2010 年 11 月 14 日，晚，于东华理工。