

仅作参考，请勿外传，谢谢

A. 数组与递归过程

习题

【A-1】猴子吃桃

有一天，一只猴子摘了一些桃子。它吃掉了一半，又吃了一个，第二天也是这样，吃了所有桃子的一半，又多吃了一个 ... 到了第十天，小猴子一看，只有1个桃子了。求最早的时候，小猴子摘了几个桃子？

$$a = 1$$

$$a = (a+1) \times 2$$

第几天	桃子数
10	1
9	4
8	10
7	22
6	46
5	94
4	190
3	382
2	766
1	1534

```
void monkey()
{
    int a = 1;
    int i;
    for (i=0;i<9;i++)
        a = (a + 1) * 2;
    printf("%d \n",a);
}
```



【A-2】约瑟夫（Josephus）问题

在Jewish-Roman战争时期，著名的历史学家Josephus（37岁）与一些犹太叛乱者被罗马的军队围困在一个山洞里，这些叛乱者宁为玉碎不为瓦全，决定以自杀作为最后的抗争手段，但是Josephus不想死呵，所以非常有数学天赋的他就提出了一个方案让自杀变得有“趣”些：让我们围成一个圆圈，然后从第一个人开始计数，每次杀掉第 2 个人，直到剩下最后一个，他再自杀。毫无察觉的众人轻率的答应了，而万万没有想到阴险的Josephus早就计算好了自己所要站在的位置，使得最后剩下的人就是他自己。当然至于后来Josephus去投降来保全自己的生命已经是另外一个故事了，那么问题就是，对于 n 个人，最后剩下的是谁呢？

例如：对于5个人的情况：1,2,3,4,5

依此杀掉 2,4,1,5，后剩下的是3

[问题2-1]

17世纪的法国数学家加斯帕在《数目的游戏问题》中讲的一个故事：15个教徒和 15 个非教徒在深海上遇险，必须将一半的人投入海中，其余的人才能幸免于难，于是想了一个办法：30个人围成一圆圈，从第一个人开始依次报数，每数到第九个人就将他扔入大海，如此循环进行直到仅余15个人为止。问怎样排法，才能使每次投入大海的都是非教徒。

答案： $+++@ @+@+@ @ @+@+++@ @+@ @ @+++@+@ @+$
(+: 表示被扔下海的非教徒, @: 留在船上活命的教徒)

[问题2-2]

古代判处死刑有一种变态的法律：将n个犯人围成一圈，从第 s 个人开始数起，每到第d个人就拉出来处决，然后再数 d 个…直到剩下最后一个可以赦免。对于这个问题，我们可以用很多方法去解决 一：我们可以建立一个链表，然后再首尾相连，形成一个环，...

[问题2-3]

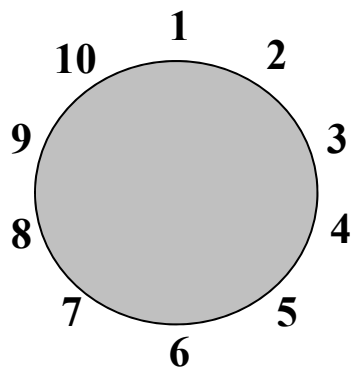
围绕着山顶有 10 个洞，狐狸要吃兔子，兔子说：“可以，但必须找到我，我就藏身于这十个洞中，你从 1 0 号洞出发，先到 1 号洞找，第二次隔 1 个洞找，第三次隔 2 个洞找，以后如此类推，次数不限。”但狐狸从早到晚进进出出了1000次，仍没有找到兔子。问兔子究竟藏在哪个洞里？

[问题2-4]

编号为1, 2,, n 的 n 个人按照顺时针方向围坐一圈，每个人有且只有一个密码（正整数）。一开始任选一个正整数作为报数上限值，从第一个人开始顺时针方向自1开始报数，报到m时停止报数。报m的人出列，将他的密码作为新的 m 值，从他在顺时针方向的下一个人开始重新报数，如此下去，直到所有人全部出列为止。设计一个程序来求出出列顺序。

简化的约瑟夫（Josephus）问题

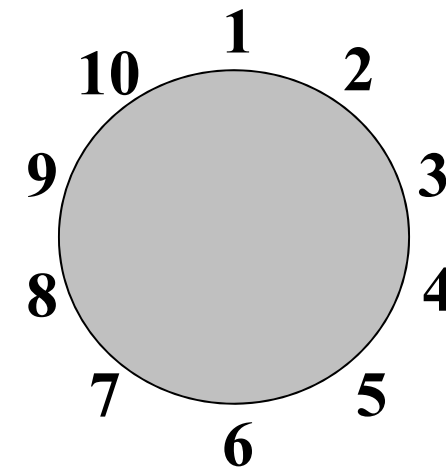
n 个骑士编号 $1, 2, \dots, n$ ，围坐在圆桌旁，编号为 k 的骑士从 1 开始报数，报到 m 的骑士出列，然后下一个位置再从 1 开始报数，找出最后留在圆桌旁的骑士编号。



假定自然数 $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ 是环形排列的，试编写一程序，从 1 开始按顺时针方向跳过 3 个数取出一个数 4，再跳过三个数取出 7，以此类推，直到将数取完为止。

约瑟夫问题—算法

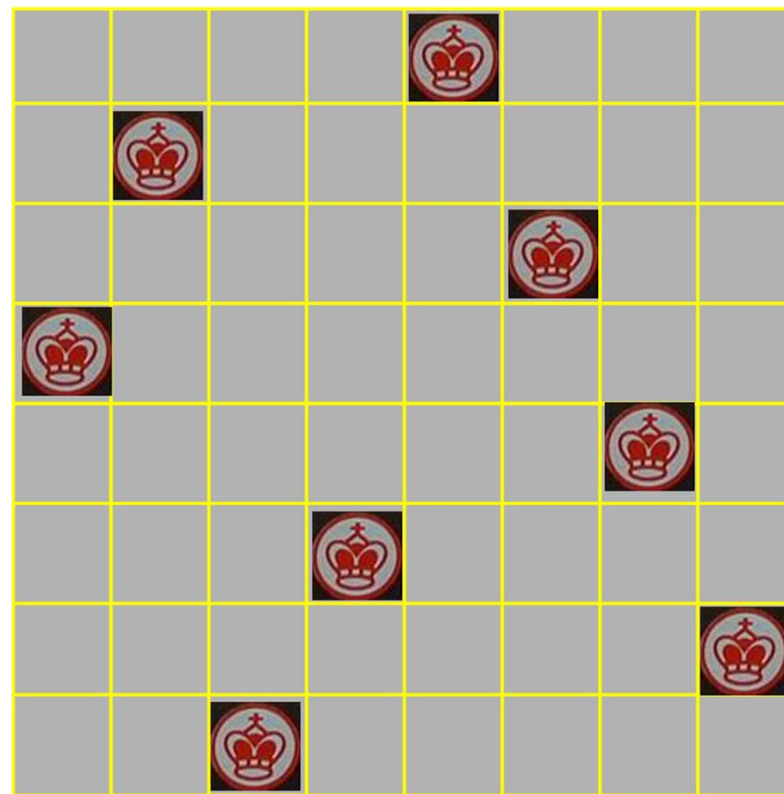
```
void Josephus()
{
    int Data[] = {1,2,3,4,5,6,7,8,9,10};
    int nLen = 10;
    int nCounter = 0;
    while (nLen > 0)
    {
        nCounter += 3;
        nCounter = nCounter % nLen;
        printf("%d ", Data[nCounter]);
        Move(Data, nCounter, nLen);
        nLen--;
    }
}
```



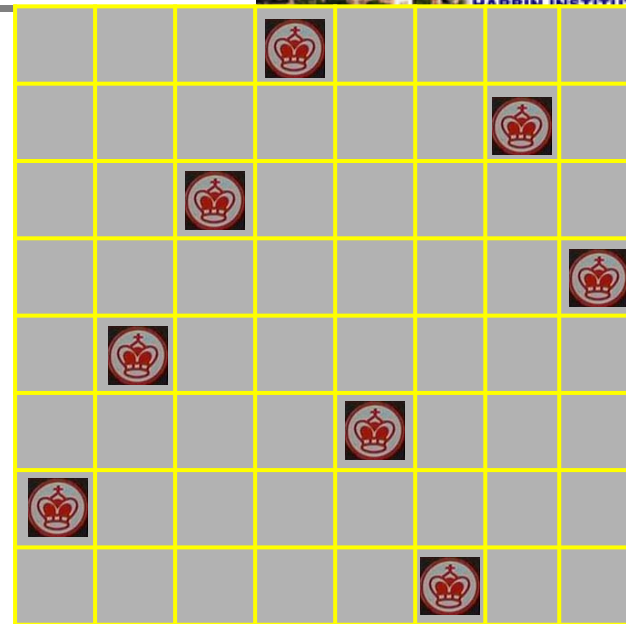
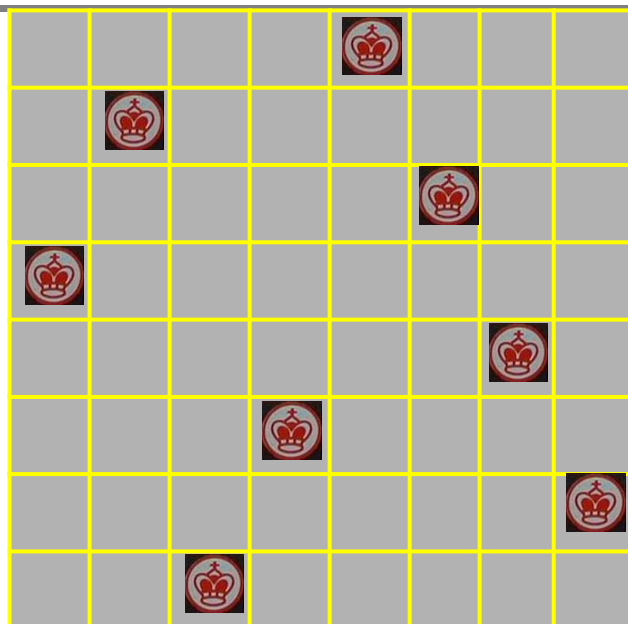
```
void Move(int *Data, int nInd, int nLen)
{
    int i;
    for (i = nInd; i < nLen - 1; i++)
        Data[i] = Data[i + 1];
}
```

【A-3】八皇后问题

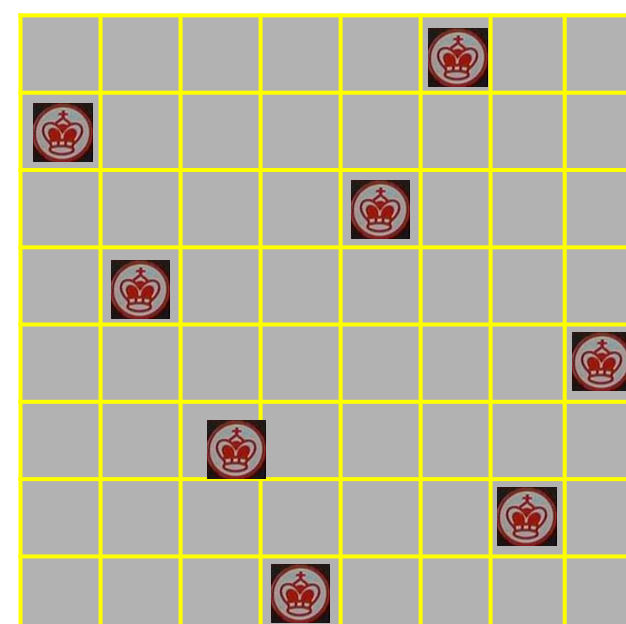
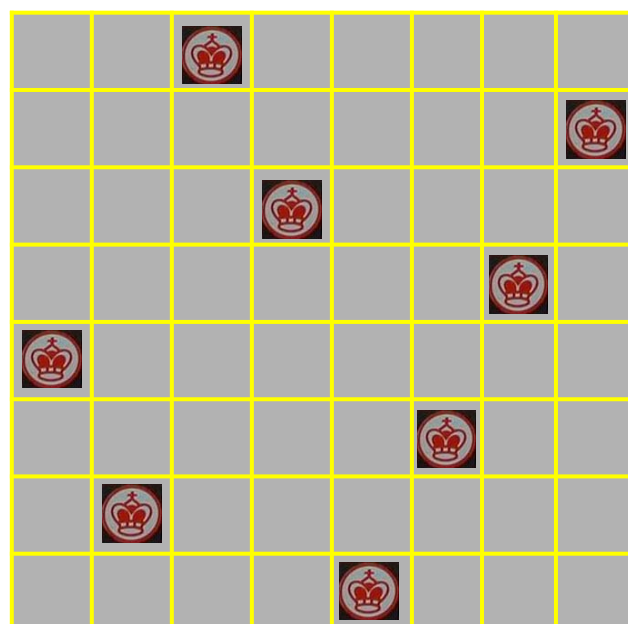
八皇后问题是大数学家高斯于1850年提出来的。该问题是在 8×8 的国际象棋棋盘上放置8个皇后，使得没有一个皇后能“吃掉”任何其他一个皇后，即没有任何两个皇后被放置在棋盘的同一行、同一列或同一斜线上。要求编一个程序求出该问题的所有解。



高斯认为有76种方案。1854年在柏林的象棋杂志上不同的作者发表了40种不同的解，后来有人用图论的方法解出92种结果。



a	b
c	d



■回溯

在解空间中有条不紊地搜索问题的解的方法。问题的解表示成解向量或 n 元式 $(x_1, x_2, x_3, \dots, x_n)$ 。对给定的问题的解空间就是解向量： $(x_1, x_2, x_3, \dots, x_n)$ 的定义域。

回溯算法的基本思想是：从一条路往前走，能进则进，不能进则退回来，换一条路再试。

用回溯算法解决问题的一般步骤为：

- (1) 定义一个解空间，它包含问题的解；
- (2) 利用适于搜索的方法组织解空间；
- (3) 利用深度优先法搜索解空间；
- (4) 利用限界函数避免移动到不可能产生解的子空间。

问题的解空间通常是在搜索问题的解的过程中动态产生的，这是回溯算法的一个重要特性。

顾名思义，回溯算法是一种很易理解的“**知难而退**”的算法，即当发现一个解不可用时返回上一层解的算法思想，最好的例子就是解迷宫，从入口进入，逢岔口便走最左边一个（右边也是一样，不妨设为左），走到尽头便退回上一个岔口，仍走最左边一个（也就是从最初的入口看，封掉刚走过的那条路的最左边一条），如此下去，即刻走出迷宫。

约束：

求得的解要满足一组约束。

显约束：指每个 x_i 从给定集合取值的规则

隐约束：描述 $x_i(1 \leq i \leq n)$ 之间应有的相互关系。



回溯算法的实现

为解决这个问题，我们把棋盘的横坐标定为 i ，纵坐标定为 j ， i 和 j 的取值范围是从 1 到 8。当某个皇后占了位置 (i,j) 时，在这个位置的垂直方向、水平方向和斜线方向都不能再放其它皇后了。用语句实现，可定义如下三个整型数组：

$a[8]$, $b[15]$, $c[24]$ 。其中：

$a[j-1]=1$ 第 j 列上无皇后；

$a[j-1]=0$ 第 j 列上有皇后；

$b[i+j-2]=1$ (i,j) 的对角线（左上至右下）无皇后；

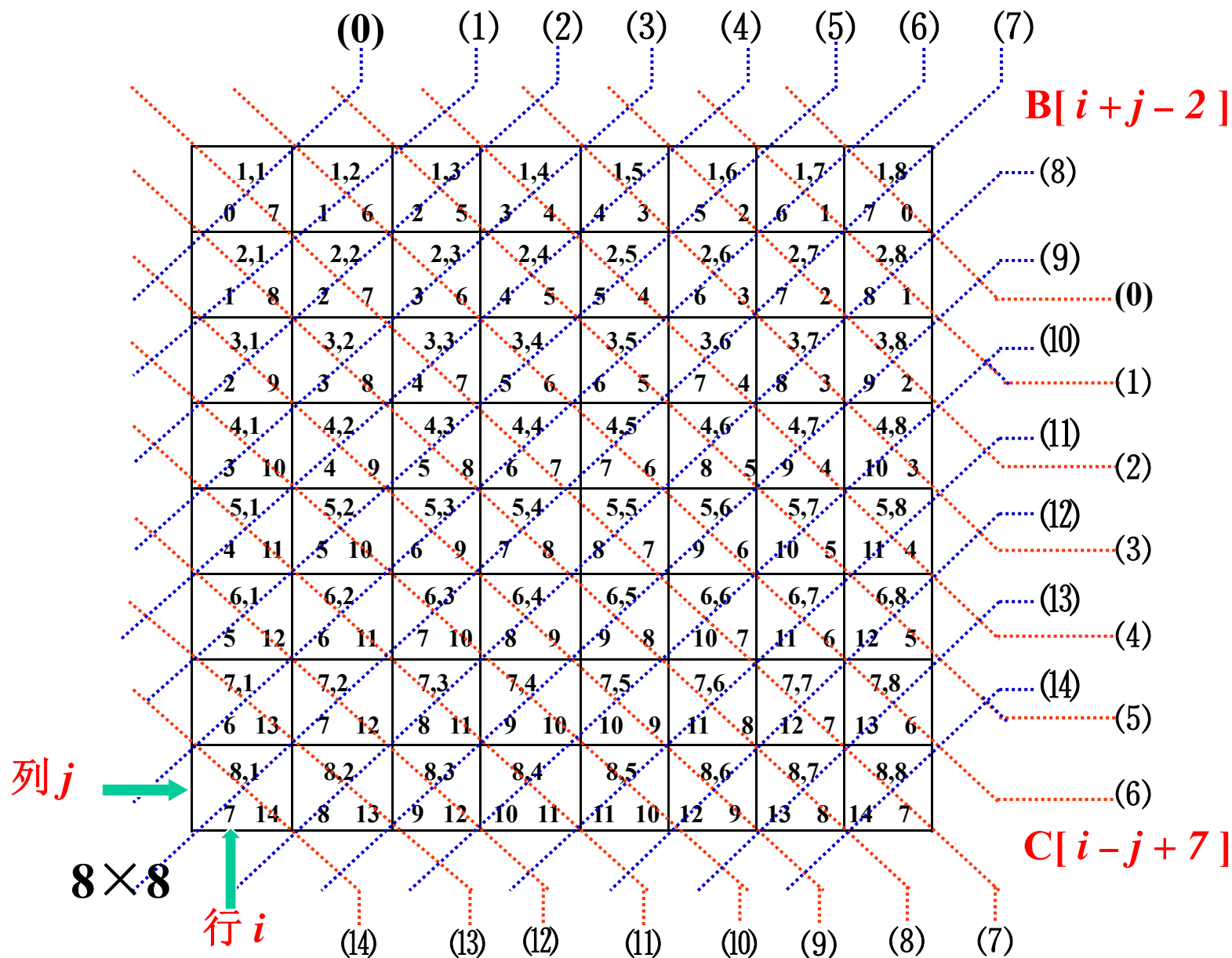
$b[i+j-2]=0$ (i,j) 的对角线（左上至右下）有皇后；

$c[i-j+7]=1$ (i,j) 的对角线（右上至左下）无皇后；

$c[i-j+7]=0$ (i,j) 的对角线（右上至左下）有皇后；

为第 i 个皇后选择位置的算法如下：

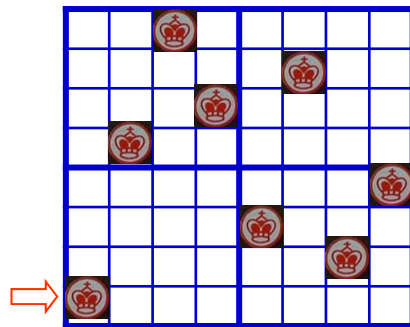
```
for(j=1;j<=8;j++)           /*第i个皇后在第j行*/
    if ((i,j)位置为空)      /*即相应的三个数组的对应元素值为1*/
    { 占用位置 (i,j)        /*置相应的三个数组对应的元素值为0*/
        if (i<8) 为i+1个皇后选择合适的位置；
        else 输出一个解    }
```



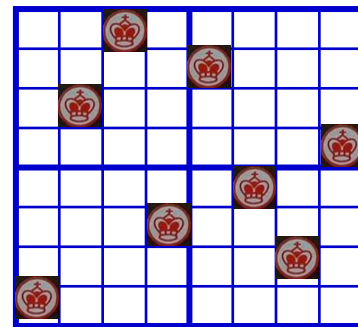
```

void try( int i )
{ int j;
  for (j=1;j<=8;j++)
    if (a[j-1]+b[i+j-2]+c[i-j+7]==3)      /*如果第i列第j行为空*/
      { a[j-1]=0; b[i+j-2]=0; c[i-j+7]=0; /*占用第i列第j行*/
        q[i-1]=j;
        if(i<8)
          try(i+1);
        else                                /*输出一组解*/
          { for(k=0;k<8;k++) printf("%d,",q[k]);
            printf("\n"); }                /*某列行号
          a[j-1]=1;b[i+j-2]=1;c[i-j+7]=1;
          q[i-1]=0;
        }
      }
}
    
```

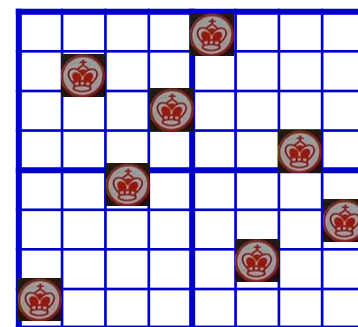




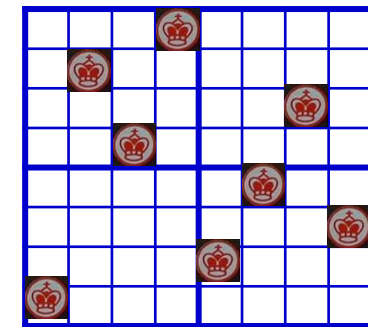
(1) 1 5 8 6 3 7 2 4



(2) 1 6 8 3 7 4 2 5

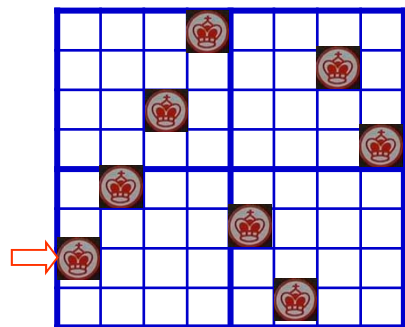


(3) 1 7 4 6 8 2 5 3

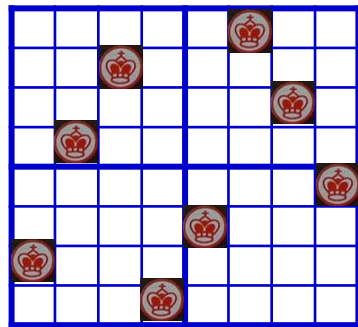


(4) 1 7 5 8 2 4 6 4

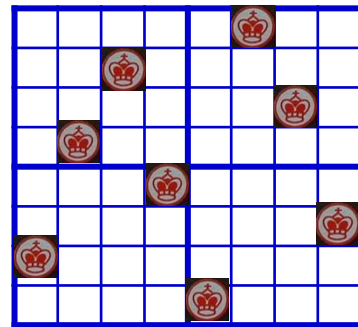
4种



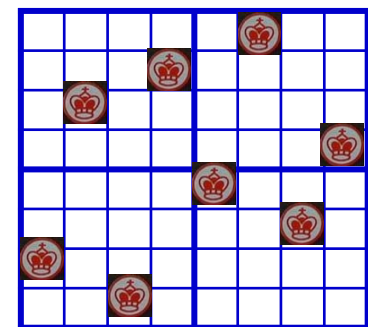
(5) 2 4 6 8 3 1 7 5



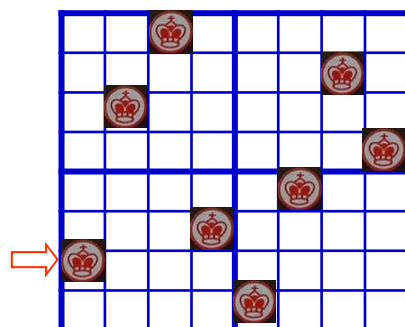
(6) 2 5 7 1 3 8 6 4



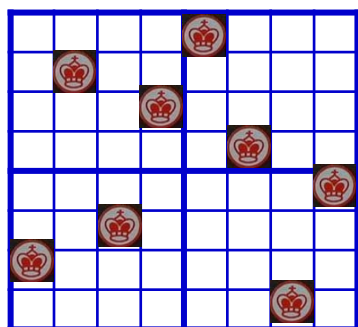
(7) 2 5 7 4 1 8 6 3



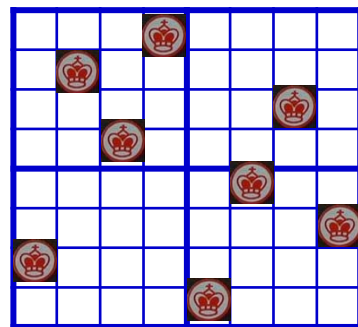
(8) 2 6 1 7 4 8 3 5



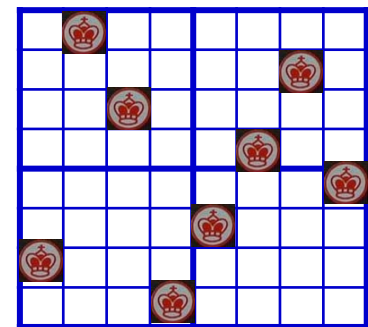
(9) 2 6 8 3 1 4 7 5



(10) 2 7 3 6 8 5 1 4

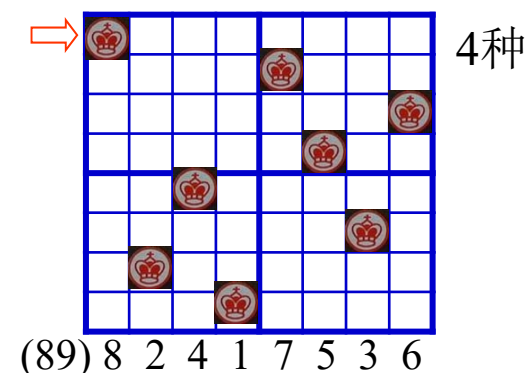
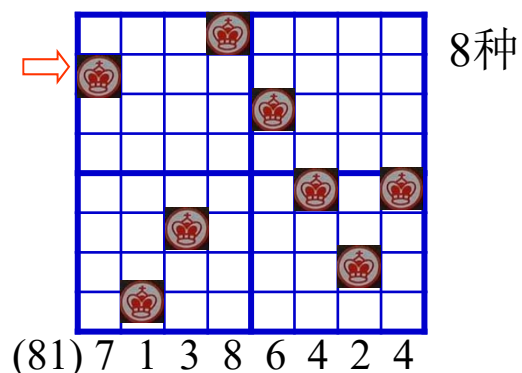
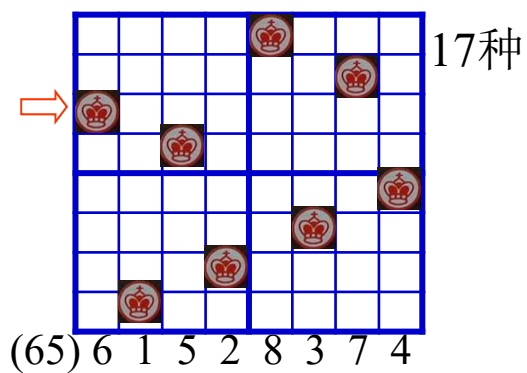
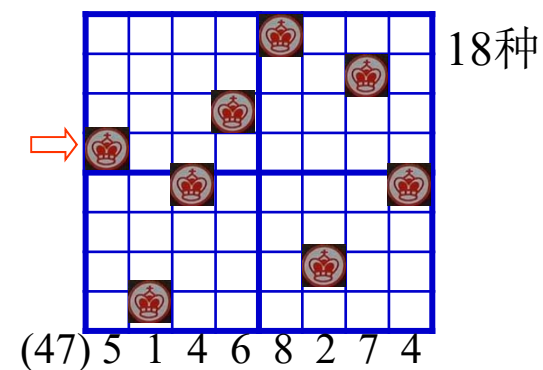
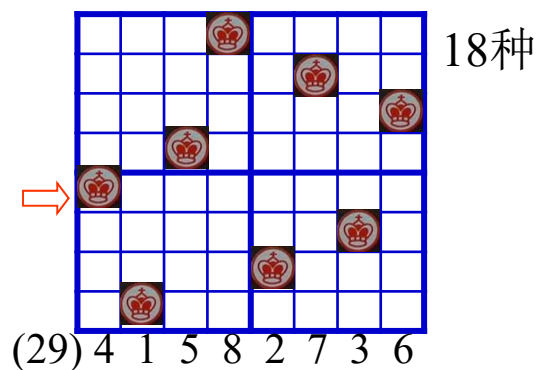
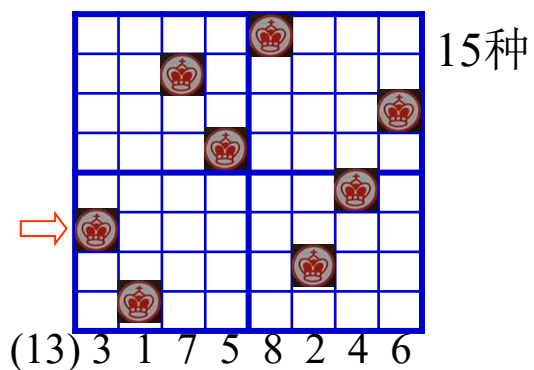


(11) 2 7 5 8 1 4 6 3



(12) 2 8 6 1 3 5 7 4

8种



共 $4+8+15+18+18+17+8+4=92$ 种

"E:\我的课件\Datastructure'

```
1:1, 5, 8, 6, 3, 7, 2, 4  
2:1, 6, 8, 3, 7, 4, 2, 5  
3:1, 7, 4, 6, 8, 2, 5, 3  
4:1, 7, 5, 8, 2, 4, 6, 3  
5:2, 4, 6, 8, 3, 1, 7, 5  
6:2, 5, 7, 1, 3, 8, 6, 4  
7:2, 5, 7, 4, 1, 8, 6, 3  
8:2, 6, 1, 7, 4, 8, 3, 5  
9:2, 6, 8, 3, 1, 4, 7, 5
```

```
80:6, 8, 2, 4, 1, 7, 5, 3  
81:7, 1, 3, 8, 6, 4, 2, 5  
82:7, 2, 4, 1, 8, 5, 3, 6  
83:7, 2, 6, 3, 1, 4, 8, 5  
84:7, 3, 1, 6, 8, 5, 2, 4  
85:7, 3, 8, 2, 5, 1, 6, 4  
86:7, 4, 2, 5, 8, 1, 3, 6  
87:7, 4, 2, 8, 6, 1, 3, 5  
88:7, 5, 3, 1, 6, 8, 2, 4  
89:8, 2, 4, 1, 7, 5, 3, 6  
90:8, 2, 5, 3, 1, 7, 4, 6  
91:8, 3, 1, 6, 2, 5, 7, 4  
92:8, 4, 1, 3, 6, 2, 7, 5  
Press any key to continue
```

【A-4】背包问题

1978年由Merkel和Hellman提出的。它的主要思路是假定某人拥有大量物品，重量各不同。此人通过秘密地选择一部分物品并将它们放到背包中来加密消息。背包中的物品中重量是公开的，所有可能的物品也是公开的，但背包中的物品是保密的。附加一定的限制条件，给出重量，而要列出可能的物品，在计算上是不可实现的。背包问题是熟知的不可计算问题，背包体制以其加密，解密速度快而引人注目。但是，大多数一次背包体制均被破译了，因此现在很少有人使用它。

随着时间发展，背包问题演变成很多类似的问题。

背包分两种，加法背包和乘法背包！以加法背包为例！

如： $1 < 2, 1+2 < 4, 1+2+4 < 8, 1+2+4+8 < 16, \dots$ ，

如果我们选择这样一些数，这些数从小到大排列，如果前面所有的数加起来的值总小于后面的数，那么这些数就可以构成一个背包，我们给一个这个背包里面的某些数的和，这个数就是被加密的数，由这个背包组成这个数只有一种组合方式，这个方式就是秘密了。

如给大家一个背包：(2, 3, 6, 12, 24, 48)，
由这个背包里的某些数构成的数：86，你知道86怎么来的吗？当然，你看着背包里面的内容，可以知道是由 $2+12+24+48$ 得到的，如果你没有这个背包，而是直接得到这个86，你知道组成这个86的最小的数是多少吗？你无法知道，因为加起来等于86的数非常多： $85+1=86$ ， $82+2=86$ 等等，你是无法知道的，所以，背包加密非常难破。

如果游戏里面，用户名和密码只能有字母和数字组成，那么总共就那么36个元素，我们利用一个包含36个元素的背包，背包的一个元素对应一个字母或者数字，当玩家设好用户名后，我们把这个用户名翻译成背包里面的元素，然后把这些用户转化而来的数字加起来，得到一个数，这个数就是用户名。

有个密码是：511，这个密码是经过加法背包加密的，如果你没有背包，你就无法知道密码！

如果我告诉你背包是：(1,2,4,8,16,32,64,128,256)，且告诉你这个是加法背包，而且有如下对应关系：

1→a, 2→b, 4→c, 8→d, 16→e, 32→f, 64→g, 128→h, 256→i,

那么你也也许知道密码就是：abcdefghi

4-1 01背包问题

[题目]

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c , 价值是 w 。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量, 且价值总和最大。

[基本思路]

这是最基础的背包问题, 特点是: 每种物品仅有一件, 可以选择放或不放。

用子问题定义状态: 即 $f[v]$ 表示前 i 件物品恰放入一个容量为 v 的背包可以获得的**最大价值**。则其状态转移方程便是:

$$f[v] = \max \{ f[v], f[v-c] + w \}$$

4-2 完全背包问题

[题目]

有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的费用是c，价值是w。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

[基本思路]

这个问题非常类似于01背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取0件、取1件、取2件……等很多种。如果仍然按照解01背包时的思路，令 $f[v]$ 表示前i种物品恰放入一个容量为v的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：

$$f[v] = \max \{ f[v - k \times c] + k \times w \mid 0 \leq k \times c \leq v \}。$$

这跟01背包问题一样有 $O(NV)$ 个状态需要求解，但求解每个状态的时间则不是常数了，求解状态 $f[v]$ 的时间是 $O(v/c)$ ，总的复杂度是超过 $O(VN)$ 的。

4-3 多重背包问题

[题目]

有N种物品和一个容量为V的背包。第i种物品最多有n件可用，每件费用是c，价值是w。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

[基本算法]

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第i种物品有n+1种策略：

取0件，取1件……取 n件。

令f[v]表示前i种物品恰放入一个容量为v的背包的最大权值，则：

$$f[v] = \max \{ f[v - k \times c] + k \times w \mid 0 \leq k \leq n \}。$$

复杂度是 $O(V \times \sum n)$ 。

4-4 有依赖的背包问题

这种背包问题的物品间存在某种“依赖”的关系。也就是说， i 依赖于 j ，表示若选物品 i ，则必须选物品 j 。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；

另外，没有某件物品同时依赖多件物品。

4-5 背包问题的演变

以上涉及的各种背包问题都是要求在背包容量（费用）的限制下求可以取到的最大价值，但背包问题还有很多种灵活的问法，在这里值得提一下。但是我认为，只要深入理解了求背包问题最大价值的方法，即使问法变化了，也是不难想出算法的。

例如，求解最多可以放多少件物品或者最多可以装满多少背包的空间。这都可以根据具体问题利用前面的方程求出所有状态的值（ f 数组）之后得到。

还有，如果要求的是“总价值最小”“总件数最小”，只需简单的将上面的状态转移方程中的 max 改成 min 即可。

4-6 常见的背包问题

设有一个背包可以放入的物品的重量为 s ，现有 n 件物品，重量分别为 $w[1], w[2], \dots, w[n]$ 。问能否从这 n 件物品中选择若干件放入此背包中，使得放入的重量之和正好为 S 。如果存在一种符合上述要求的选择，则称此背包问题有解(或称其解为真)；否则称此背包问题无解(或称其解为假)。试用递归方法设计求解背包问题的算法。

运用局部最佳策略以求达到全局最佳结果。

给定输入数据为 $A[1], A[2], \dots, A[n]$ ，对解附有某些约束条件和表示最佳结果的目标函数，欲求满足约束条件的子集 $A[i_k]$ ，使目标函数值达到最大（或最小）。

满足约束条件的任意子集叫做可用解，使给定的目标值达到最大（或最小）的可用解叫做最佳解。

最终目的是求全局最佳解。


```
Dataset Greedy(A, n)
LIST A; int n;
{
    solution =  $\emptyset$ ;
    for( i = 1; i <= n; i++ )
        {
            x = select(A);
            if ( feasible( solution, x ) )
                solution = union ( solution , x );
        }
    return solution;
}
```

select 对 A 确定一种取值办法；每步对一个 $x=A[i]$ 作出判断：x 是否可以包含在最佳解中？若 x 加入局部最佳解时就产生不可用解，放弃 x，另作选择。

union 将 x 和 solution 结合成新的解向量，并修改目标函数的值。



在**贪心算法**（greedy method）中采用逐步构造最优解的方法。在每个阶段，都作出一个看上去最优的决策（在一定的标准下）。决策一旦作出，就不可再更改。作出贪婪决策的依据称为贪婪准则（greedy criterion）。

[例找零钱] 一个小孩买了价值少于 1 美元的糖，并将 1 美元的钱交给售货员。售货员希望用数目最少的硬币找给小孩。假设提供了数目不限的面值为 25 美分、10 美分、5 美分、及 1 美分的硬币。售货员分步骤组成要找的零钱数，每次加入一个硬币。选择硬币时所采用的贪婪准则如下：每一次选择应使零钱数尽量增大。为保证解法的可行性（即：所给的零钱等于要找的零钱数），所选择的硬币不应使零钱总数超过最终所需的数目。

假设需要找给小孩 67 美分，首先入选的是两枚 25 美分的硬币，第三枚入选的不能是 25 美分的硬币，否则硬币的选择将不可行（零钱总数超过 67 美分），第三枚应选择 10 美分的硬币，然后是 5 美分的，最后加入两个 1 美分的硬币。

采用递归算法求解背包问题

$KNAP(s, n) = \begin{cases}$	True	$s = 0$	此时背包问题一定有解
	False	$s < 0$	总重量不能为负数
	False	$s > 0$ 且 $n < 1$	物品件数不能为负数
	$KNAP(s, n-1)$ 或 $KNAP(s - w[n], n-1)$	$s > 0$ 且 $n \geq 1$	所选物品中不包括 $w[n]$ 时 所选物品中包括 $w[n]$ 时

```
enum boolean { False, True }
boolean Kanp(int *w, int s, int n)
{
    if ( s == 0 ) return True;
    if ( s < 0 || s > 0 && n < 1 ) return False;
    if ( Kanp ( s - W[n], n-1 ) == True )
        { cout << W[n] << ' , ' ; return True; }
    return Kanp( s, n-1 );
}
```



【A-5】 设 $m \times n$ 阶稀疏矩阵A有 t 个非零元素，其三元组表表示为LTMA[1:(t+1), 1:3]，试问：非零元素的个数 t 达到什么时候时用LTMA表示A才有意义？

解答：

A的最后一个元素是 a_{mn} ，其余元素的下标积必小于 mn

即有： $3(t+1) \leq mn$

因而：当 $t \leq mn/3 - 1$ 时，用LTMA表示A才有意义。

【A-6】 假设按低下标优先存储整数数组A(-3:8, 3:5, -4:0, 0:7)时，第一个元素的字节存储地址是100，每个整数占4个字节。
问：A(0, 4, -2, 5)的存储地址是什么？

坐标平移： A(-3:8,3:5,-4:0,0:7)对应规范化的数组为：
A(0:11,0:2,0:4,0:7)
元素A(0,4,-2,5) 相当于 A(3,1,2,5)

所以有：

$$\begin{aligned} L(3,1,2,5) &= 100 + 4(3 \times 2 \times 4 \times 7 + 1 \times 4 \times 7 + 2 \times 7 + 5) \\ &= 1164 \end{aligned}$$

【A-7】 求一个数组的所有有序子数组中最长的一个。

如：在数组[1, 8, 1, 2, 5, 0, 11, 9]中，这个最长的有序子数组为：
[1, 2, 5]，其长度为3。

```
....  
length=1;  
for (i = 0; i<n-1; i++)  
{  j1 = j2 = k = i;  
  while( ( k < n-1) && (a[k] < a[k+1]) )  
  {  k++;  
    j2++; }  
  if (length < j2 -j1 +1)  length = j2 -j1 + 1;  
}  
....
```



```
for (i = 0, length=1; i<n-1; i++)  
{  for (j1 = j2 = k = i; k < n-1 && a[k] < a[k+1]; k++, j2++);  
  if (length < j2-j1+1)  length = j2 -j1 + 1;  
}
```

【A-8】花盆排序问题

```
#include<iostream.h>
#include<string.h>
void main()
{ char flower[100],*r,*b,*w,temp;
  Int count;
  cin>>flower;
  count=strlen(flower);
  r=flower;
  b=w=&flower[count-1];
  while(r-w<0)
  { switch (*w)
    { case 'r': { temp=*r; *r=*w; *w=temp; r++; break; }
      case 'w': { w--; break; }
      case 'b': { temp=*w; *w=*b; *b=temp; w--; b--; break; }
    }
  }
  cout<<flower;
}
```

假设红、白、蓝每种颜色的花盆至少有一盆，亦即 $n \geq 3$ ，杂乱的排在一起，编写一个高效算法使花盆按红、蓝、白的顺序排序。

假设每组输入为：一个整数 N ，表示有 N 个花盆。然后 N 个整数，每个整数为1（若为红花盆）、2（若为白花盆）或3（若为蓝花盆），每个整数用空格分开。输入 $N = 0$ 时程序结束。

【A-9】找第一次重复出现的元素

当我们遇到这样的数组
 $a=\{1, 2, 3, 3, 5, 7, 8\}$ ，存在重复的元素，需要从中找出3第一次出现的位置，这里3第一次出现的位置是2，《编程珠玑》里给出了很好的分析，二分搜索主要的精髓在于不变式的设计（就是上面的while循环条件式）。



```
int binsearch_first(int arr[], int k, int n)
{
    int l = -1, h = n;
    while(l + 1 != h)
    {
        int m = (l + h) / 2;
        if(k > arr[m])
            l = m;
        else
            h = m;
    }
    int p = h;
    if(p >= n || arr[p] != k)
        return -1;
    return h;
}
```


算法分析:

设定两个不存在的元素 $a[-1]$ 和 $a[n]$, 使得 $a[-1] < t \leq a[n]$, 但是我们并不会去访问这两个元素, 因为 $(l+u)/2 > l=-1$, $(l+u)/2 < u=n$ 。

循环不变式为 $l < u$ && $t > a[l]$ && $t \leq a[u]$ 。

循环退出时必然有 $l+1=u$, 而且 $a[l] < t \leq a[u]$ 。循环退出后 u 的值为 t 可能出现的位置, 其范围为 $[0, n]$, 如果 t 在数组中, 则第一个出现的位置 $p=u$, 如果不在, 则设置 $p=-1$ 返回。

该算法的效率虽然解决了更为复杂的问题, 但是其效率比初始版本的二分查找还要高, 因为它在每次循环中只需要比较一次, 前一程序则通常需要比较两次。

例如:

对于数组 $a=\{1, 2, 3, 3, 5, 7, 8\}$, 我们如果查找 $t=3$, 则可以得到 $p=u=2$, 如果查找 $t=4$, $a[3]<t\leq a[4]$, 所以 $p=u=4$, 判断 $a[4] \neq t$, 所以设置 $p=-1$ 。一种例外情况是 $u \geq n$, 比如 $t=9$, 则 $u=7$, 此时也是设置 $p=-1$ 。特别注意的是, $l=-1$, $u=n$ 这两个值不能写成 $l=0$, $u=n-1$ 。虽然这两个值不会访问到, 但是如果改成后面的那样, 就会导致二分查找失败, 那样就访问不到第一个数字。如在 $a=\{1, 2, 3, 4, 5\}$ 中查找1, 如果初始设置 $l=0$, $u=n-1$, 则会导致查找失败。

同理: 查找数组中一个元素最后出现的位置如下:

```
int binsearch_last(int arr[], int n, int k)
{
    int l = -1, h = n;
    while(l + 1 != h)
    {
        int m = (l + h) / 2;
        if(k >= arr[m])
            l = m;
        else
            h = m;
    }
    int p = l;
    if(p <= -1 || arr[p] != k)
        return -1;
    return p;
}
```



【A-10】旋转数组中查找指定的元素。

将一个有序数组以一个点为中心进行旋转（前后颠倒）；
例如{4, 5, 1, 2, 3}是{1,2,3,4,5}以3为中心的旋转。
这样数组就变得整体无序了，但是还是部分有序的。

问题：查找给定元素。

方案：可以找到旋转的中点，然后对两个部分有序的数组分别进行二分查找



```

int bsearch_rotate(int a[], int n, int t)  //2次二分查找
{
    int p = split(a, n); //找到分割位置
    if (p == -1)
        return bsearch_first(a, n, t); //如果原数组有序，则直接二分查找即可
    else
    {
        int left = bsearch_first(a, p+1, t); //查找左半部分
        if (left == -1)
        { //左半部分没有找到，则查找右半部分
            int right = bsearch_first(a+p+1, n-p-1, t); //
            if (right != -1)
                return right+p+1; //返回位置，注意要
            return -1;
        }
        return left; //左半部分找到，则直接返回
    }
}
    
```

```

int split(int a[], int n)
{
    for (int i=0; i<n-1; i++)
    {   if (a[i+1] < a[i])
        return i;
    }
    return -1;
}
    
```

```
int bsearch_rotate(int a[], int n, int t)    //1次二分查找
{
    int low = 0, high = n-1;
    while (low <= high)
    {
        int mid = low + (high-low) / 2;
        if (t == a[mid])
            return mid;
        if (a[mid] >= a[low]) //数组左半有序
        {
            if (t >= a[low] && t < a[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
        else //数组右半段有序
        {
            if (t > a[mid] && t <= a[high])
                low = mid + 1;
            else
                high = mid - 1;
        }
    }
    return -1;
}
```



【A-11】 已知由n个整数构成序列的数据分布为先下降再上升，即一开始数据是严格递减的，后来数据是严格递增的，设计尽可能高效算法，找到序列中的最小值。

```
int FindMin(int Arr[], int Left, int Right)
{
    int Mid, Result;
    while (Left < Right)
    {
        Mid = (Left + Right) / 2;
        if (Arr[Mid] < Arr[Mid-1] && Arr[Mid] < Arr[Mid+1])
            return Mid;
        if (Arr[Mid] < Arr[Mid-1] && Arr[Mid] > Arr[Mid+1])
            Left = Mid + 1;
        if (Arr[Mid] > Arr[Mid-1] && Arr[Mid] < Arr[Mid+1])
            Right = Mid - 1;
    }
    if (Left == Right)
        Result = Left;
    if (Left < Right)
        Result = Arr[Left] < Arr[Right] ? Left : Right;
    return Result;
}
```



另：已知由n个整数构成序列的数据分布为先上升再下降，即一开始数据是严格递增的，后来数据是严格递减的，设计尽可能高效算法，找到序列中的最大值。

```
int FindMax(int Arr[], int Left, int Right)
{
    int Mid, Result;
    while (Left < Right)
    {
        Mid = (Left + Right) / 2;
        if (Arr[Mid] > Arr[Mid-1] && Arr[Mid] > Arr[Mid+1])
            return Mid;
        if (Arr[Mid] > Arr[Mid-1] && Arr[Mid] < Arr[Mid+1])
            Left = Mid + 1;
        if (Arr[Mid] < Arr[Mid-1] && Arr[Mid] > Arr[Mid+1])
            Right = Mid - 1;
    }
    if (Left == Right)    Result = Left;
    if (Left < Right)    Result = Arr[Left] > Arr[Right] ? Left : Right;
    return Result;
}
```

调用：p=FindMax(A, 0, n-1);



【A-12】 在一个有序数组中只有一个元素出现一次，其余元素均出现2次，找出这个元素。

利用异或运算的这两个法则：

$$(1) \quad a \oplus b = b \oplus a$$

$$(2) \quad a \oplus b \oplus c = a \oplus (b \oplus c) = (a \oplus b) \oplus c;$$



```
int singleNumber(int A[], int n)
{
    int num=0;
    for(int i=0;i<n;i++)
        num=num^A[i];
    return num;
}
```


【A-13】 一个数组有除了两个元素只出现一次，其他元素全部都出现了两次，请找出只出现一次的两个元素，并输出。

因为这两个只出现一次的元素一定是不相同的，所以这两个元素的二进制形式肯定至少有某一位是不同的，即一个为0，另一个为1，找到这一位。

可以根据前面异或得到的数字找到这一位，怎么找呢？稍加分析就可以知道，异或得到这个数字二进制形式中任意一个为1的位都是我们要找的那一位，找到这一位就可以了(这很容易)。

再然后，以这一位是1还是0为标准，将数组的 n 个元素分成了两部分，将这一位为0的所有元素做异或，得出的数就是只出现一次的数中的一个;将这一位为1的所有元素做异或，得出的数就是只出现一次的数中的另一个。从而解出题目。忽略寻找不同位的过程，总共遍历数组两次，时间复杂度为 $O(n)$ 。

```
int main()
{
    int a[N] = {1,2,7,99,3,3,4,1,2,4}, flat[N] = {0};
    int i, j, sum = a[0], last_bit = 0, sum0 = 0, sum1 = 0;
    for(i = 1; i < N; i++)
        sum = sum ^ (a[i]);           // 注释1
    for(i = 0; last_bit != 1; i++)
    {
        last_bit = sum & 1;
        sum = sum >> 1;    }         // 注释2
    for(j = 0; j < N; j++)
        flat[j] = (a[j] >> (i-1)) & 1; //注释3

    for(j = 0; j < N; j++)
        (flat[j] == 1) ? (sum1 = sum1 ^ (a[j])) : (sum0 = sum0 ^ (a[j]));
    printf("%d %d\n", sum0, sum1);
    return 0;
}
```



注释1：如果当数组中有两个元素只出现一次时，则将上述异或的结果转化成二进制，并从右到左进行判断，将第一个为1的位置记录为i

注释2：接着对所有的元素的二进制形式的第i位进行扫描，将结果记录到flat数组flat[j]=1表明第j元素的第i位为1，否则为0

注释3：根据上述循环得到的flat数组，对所有元素进行分组结果为1的在一起进行异或操作，为0的在一起进行异或操作，两组异或的结果就是两个只出现一次的元素

【A-14】有序数组的插入问题可以通过二分搜索实现，只是返回值不同而已，在没找到元素的时候返回 l （最左值）值就是元素应该插入的位置。



```
int binsearch1(int arr[], int k, int l, int h)
{
    if(l > h) return -1;
    int mid;
    while(l <= h)
    {
        mid = (l + h) / 2;
        if(k == arr[mid])
            return mid;
        else if(k > arr[mid])
            l = mid + 1;
        else
            h = mid - 1;
    }
    return l;
}
```

【A-15】 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。

思路：

- (1) 让指针指向数组的头部和尾部，相加，如果小于M，则增大头指针，如果大于则减小尾指针
- (2) 退出的条件，相等或者头部=尾部



算法：


```
void function(int a[],int n,int M)
{
    int i=0,j=n-1;
    while(i!=j)
    {
        if(a[i]+a[j]==M)
        {
            printf("%d,%d",a[i],a[j]);
            break;
        }
        a[i]+a[j]>M?j--:i++;
    }
}
```

【A-16】数组求和

给定一个含有 n 个元素的整型数组 a ，求 a 中所有元素的和。可能您会觉得很简单，是的，的确简单，但是为什么还要说呢，原因是这道题要求用递归法，只用一行代码。

【分析】①如果数组元素个数为0，那么和为0。②如果数组元素个数为 n ，那么先求出前 $n - 1$ 个元素之和，再加上 $a[n - 1]$ 即可。

```
int sum(int *a, int n)
{   return n == 0 ? 0 : sum(a, n - 1) + a[n - 1]; }
```



【A-17】求数组中的最大值与最小值

给定一个含有 n 个元素的整型数组 a ，找出其中的最大值和最小值。

【分析】常规的做法是遍历一次，分别求出最大值和最小值。

现考虑分治法(Divide and conquer)，将数组分成左右两部分，先求出左半部份的最大值和最小值，再求出右半部份的最大值和最小值，然后综合起来求总体的最大值及最小值。这是一个递归过程，对于划分后的左右两部分，重复这个过程，直到划分区间内只剩一个元素或者两个元素。

```
void MaxandMin(int *a, int l, int r, int *maxValue, int *minValue)
{
    int m, lmax, lmin, rmax, rmin;
    if(l == r) // l与r之间只有一个元素
    {
        *maxValue = a[l];
        *minValue = a[l];
        return;
    }
    if(l + 1 == r) // l与r之间只有两个元素
    {
        if(a[l] >= a[r])
        {
            *maxValue = a[l];
            *minValue = a[r];
        }
        else
        {
            *maxValue = a[r];
            *minValue = a[l];
        }
        return;
    }
    m = (l + r) / 2; // 求中点
    MaxandMin(a, l, m, &lmax, &lmin); // 递归计算左半部份
    MaxandMin(a, m + 1, r, &rmax, &rmin); // 递归计算右半部份
    *maxValue = max(lmax, rmax); // 总的最大值
    *minValue = min(lmin, rmin); // 总的最小值
}
```



【A-18】求数组的最大值与次大值

给定一个含有 n 个元素的整型数组，求其最大值和次大值。

【分析】同样是用分治法，先求出左边的最大值 leftmax 和次大值 leftsecond ，再求出右边的最大值 rightmax 和次大值 rightsecond ，然后合并。

分情况考虑：

- ① 如果 $\text{leftmax} > \text{rightmax}$ ，那么可以肯定 leftmax 是最大值，但次大值不一定是 rightmax ，但肯定不是 rightsecond ，只需将 leftsecond 与 rightmax 做一次比较即可。
- ② 如果 $\text{rightmax} > \text{leftmax}$ ，那么可以肯定 rightmax 是最大值，但次大值不一定是 leftmax ，但肯定不是 leftsecond ，所以只需将 leftmax 与 rightsecond 做一次比较即可。

```
void MaxandMin(int a[], int left, int right, int *max, int *second)
{ int leftmax , leftsecond, rightmax, rightsecond , mid;
  if(left == right)
  { *max = a[left] ;
    *second = -1000; }
  else if(left + 1 == right)
  { *max = a[left] > a[right] ? a[left] : a[right] ;
    *second = a[left] < a[right] ? a[left] : a[right] ; }
  else
  { mid = left + (right - left) / 2 ;
    MaxandMin(a, left, mid, &leftmax, &leftsecond) ;
    MaxandMin(a, mid + 1, right, &rightmax, &rightsecond) ;
    if (leftmax > rightmax)
    { *max = leftmax ;
      *second = leftsecond > rightmax ? leftsecond : rightmax ; }
    else
    { *max = rightmax ;
      *second = leftmax < rightsecond ? rightsecond : leftmax ; }
  }
}
```



【A-19】求数组中出现次数超过一半的元素

给定一个 n 个整型元素的数组 a ，其中有一个元素出现次数超过 $n / 2$ ，求这个元素。

【分析】设置一个当前值和当前值的计数器，初始化当前值为数组首元素，计数器值为1，然后从第二个元素开始遍历整个数组，对于每个被遍历到的值 $a[i]$ ①如果 $a[i] == \text{currentValue}$ ，则计数器值加1；②如果 $a[i] != \text{currentValue}$ ，则计数器值减1，如果计数器值小于0，则更新当前值为 $a[i]$ ，并将计数器值重置为1。



```
int Find(int* a, int n)
{
    int curValue = a[0];
    int count = 1;
    for (int i = 1; i < n; ++i)
    {
        if (a[i] == curValue)        count++;
        else
        {
            count--;
            if (count < 0)
            {
                curValue = a[i]; count = 1;
            }
        }
    }
    return curValue;
}
```

【A-20】求数组中元素最短的距离

给定一个含有 n 个元素的整型数组，找出数组中的两个元素 x 和 y 使得 $\text{abs}(x - y)$ 值最小

【分析】先对数组排序，然后遍历一次即可。

```
int compare(const void* a, const void* b)
{   return *(int*)a - *(int*)b ;
} // 求数组中元素的最短距离

void MinimumDistance(int* a, int n)
{   // Sort
    qsort(a, n, sizeof(int), compare) ;
    int i ; // Index of number 1
    int j ; // Index of number 2
    int minDistance = numeric_limits<int>::max() ;
    for (int k = 0; k < n - 1; ++k)
    {   if (a[k + 1] - a[k] < minDistance)
        {   minDistance = a[k + 1] - a[k] ;
            i = a[k] ;
            j = a[k + 1] ;
        }
    }
    cout << "Minimum distance is: " << minDistance
<< endl ;
    cout << "i = " << i << " j = " << j << endl ;
}
```

【A-21】求两个有序数组中的共同元素

给定两个含有n个元素的有序（非降序）整型数组a和b，求出其共同元素，比如a = 0, 1, 2, 3, 4, b = 1, 3, 5, 7, 9输出 1, 3。

【分析】利用数组有序的性质，用两个指针i和j分别指向a和b，比较a[i]和b[j]，根据比较结果移动指针，则有如下三种情况：

- ①a[i]< b[j]，则i++，继续比较；
- ②a[i]=b[j]，则i++,j++，继续比较；
- ③a[i]>b[j]，则j++，继续比较重复以上过程直到i或j到达数组末尾。

```
void FindCommon(int* a, int* b, int n)
{
    int i = 0;
    int j = 0;
    while (i < n && j < n)
    {
        if (a[i] < b[j])    ++i;
        else if(a[i] == b[j])
        {
            cout << a[i] << endl;
            ++i;
            ++j;
        }
        else// a[i] > b[j]
            ++j;
    }
}
```

【A-22】合并两个数组

给定含有 n 个元素的两个有序（非降序）整型数组 a 和 b 。合并两个数组中的元素到整型数组 c ，要求去除重复元素并保持 c 有序（非降序）。

。例子如下

$a = 1, 2, 4, 8$

$b = 1, 3, 5, 8$

$c = 1, 2, 3, 4, 5, 8$

【分析】利用合并排序的思想，两个指针 i, j 和 k 分别指向数组 a 和 b ，然后比较两个指针对应元素的大小，有以下三种情况

1. $a[i] < b[j]$ ，则 $c[k] = a[i]$ 。
2. $a[i] = b[j]$ ，则 $c[k]$ 等于 $a[i]$ 或 $b[j]$ 皆可。
3. $a[i] > b[j]$ ，则 $c[k] = b[j]$ 。

重复以上过程，直到 i 或者 j 到达数组末尾，然后将剩下的元素直接copy到数组 c 中即可。

```
int combine_array(int *a, int an, int *b, int bn, int *c)
{
    int ai=0, bi=0, ci=0;
    while(ai<an && bi<bn)
    {
        if(a[ai]==b[bi])
        {
            c[ci++]=a[ai++];
            bi++;
        }
        else if(a[ai]<b[bi])
            c[ci++]=a[ai++];
        else
            c[ci++]=b[bi++];
    }
    while(ai<an)
        c[ci++]=a[ai++];
    while(bi<bn)
        c[ci++]=b[bi++];
    return ci;
}
```



【A-23】找到数组中的唯一元素

给定含有1001个元素的数组，其中存放了1-1000之内的整数，只有一个整数是重复的，请找出这个数。

【分析】求出整个数组的和，再减去1-1000的和。

【A-24】找出现奇数次的元素

给定一个含有n个元素的整型数组a，其中只有一个元素出现奇数次，找出这个元素。

【分析】因为对于任意一个数k，有 $k \oplus k = 0$ ， $k \oplus 0 = k$ ，所以将a中所有元素进行异或，那么个数为偶数的元素异或后都变成了0，只留下了个数为奇数的那个元素。

【A-25】求数组中满足给定和的数对

给定两个有序整型数组a和b，各有n个元素，求两个数组中满足给定和的数对，即对a中元素i和b中元素j，满足 $i + j = d$ (d已知)。

【分析】两个指针i和j分别指向数组的首尾，然后从两端同时向中间遍历，直到两个指针交叉。

【A-26】最大和子段

给定一个整型数组a，求出最大连续子段之和，如果和为负数，则按0计算，比如1， 2， -5， 6， 8则输出 $6 + 8 = 14$ 。

```
int Sum(int* a, int n)
{
    int curSum = 0;
    int maxSum = 0;
    for (int i = 0; i < n; i++)
    {
        if (curSum + a[i] < 0)
            curSum = 0;
        else
        {
            curSum += a[i];
            maxSum = max(maxSum, curSum);
        }
    }
    return maxSum;
}
```

【A-27】最大子段积

给定一个整型数组a，求出最大连续子段的乘积，

如 1,2,-8,12,7
输出 $12 * 7 = 84$

【分析】与最大子段和类似，注意处理负数的情况。

```
int MaxProduct(int *a, int n)
{
    int maxProduct = 1; // max positive product at current position
    int minProduct = 1; // min negative product at current position
    int r = 1; // result, max multiplication totally
    for (int i = 0; i < n; i++)
    {
        if (a[i] > 0)
        {
            maxProduct *= a[i];
            minProduct = min(minProduct * a[i], 1);
        }
        else if (a[i] == 0)
        {
            maxProduct = 1;
            minProduct = 1;
        }
        else // a[i] < 0
        {
            int temp = maxProduct;
            maxProduct = max(minProduct * a[i], 1);
            minProduct = temp * a[i];
        }
        r = max(r, maxProduct);
    }
    return r;
}
```


【A-28】数组循环移位

将一个含有 n 个元素的数组向右循环移动 k 位，要求时间复杂度是 $O(n)$ ，且只能使用两个额外的变量。

【分析】比如数组 1 2 3 4 循环右移1位 将变成 4 1 2 3，观察可知1 2 3 的顺序在移位前后没有改变，只是和4的位置交换了一下，所以等同于1 2 3 4 先划分为两部分1 2 3 | 4，然后将1 2 3逆序，再将4 逆序 得到 3 2 1 4，最后整体逆序 得到 4 1 2 3。

```
void swap(int *x,int *y)
{
    *x=*x+*y;
    *y=*x-*y;
    *x=*x-*y;
}
```



```
void right_loop_k(int *a, int n,int k)
{
    int i;
    for(i=0;i<(n-k)/2;i++)
        swap(a+i,a+(n-k-1)-i);
    for (i=0;i<k/2;i++)
        swap(a+i+n-k,a+n-1-i);
    for (i=0;i<n/2;i++)
        swap(a+i,a+n-1-i);
}
```

问：向左循环移动 k 位？

【A-29】字符串逆序

给定一个含有 n 个元素的字符数组 a ，将其原地逆序。

【分析】题目要求的是原地逆序，也就是不允许额外分配空间，那么参数肯定是字符数组形式，因为字符串是不能被修改的，用两个指针分别指向字符数组的首尾，交换其对应的字符，然后两个指针分别向数组中央移动，直到交叉为止。

【A-30】组合问题

给定一个含有n个元素的整型数组a，从中任取m个元素，求所有组合。
比如：a = 1, 2, 3, 4, 5, m = 3
输出：1 2 3, 1 2 4, 1 2 5, 1 3 4, 1 3 5, 1 4 5, 2 3 4, 2 3 5, 2 4 5, 3 4 5

【分析】排列组合问题，首选回溯法，为了简化问题，我们将a中n个元素值分别设置为1-n

```
int IsValid(int*buffer,int lastIndex, int value)
{   for (int i = 0; i < lastIndex; i++)
        if (buffer[i] >= value)
            return 0;
    return 1;
}
```

```
void PrintArray(int *a, int n)
{   for (int i = 0; i < n; ++i)
        printf("%d,",a[i]);
    printf("\n");
}
```

```
void Select(int *buffer,int t, int n, int m)
{   if (t == m)
        PrintArray(buffer, m);
    else
    {   for (int i = 1; i <= n; i++)
            {   buffer[t] = i;
                if (IsValid(buffer,t, i))
                    Select(buffer,t + 1, n, m);
            }
    }
```

```
int buffer[]={1,2,3,4,5} ;
Select(buffer,0,5,3);
```



【A-31】重排问题

给定含有 n 个元素的整型数组 a ，其中包括0元素和非0元素，对数组进行排序，要求：

(1) 排序后所有0元素在前，所有非零元素在后，且非零元素排序前后相对位置不变；

(2) 不能使用额外存储空间。

例如输入： 0, 3, 0, 2, 1, 0, 0

输出： 0, 0, 0, 0, 3, 2, 1

【分析】此排序非传统意义上的排序，因为它要求排序前后非0元素的相对位置不变，或许叫做整理会更恰当一些。我们可以从后向前遍历整个数组，遇到某个位置 i 上的元素是非0元素时，如果 $a[k]$ 为0，则将 $a[i]$ 赋值给 $a[k]$ ， $a[k]$ 赋值为0。实际上 i 是非0元素的下标，而 k 是0元素的下标

```
void Arrange(int* a, int n)
{
    int k = n - 1 ;
    for (int i = n - 1; i >= 0; --i)
        if (a[i] != 0)
        {
            if (a[k] == 0)
            {
                a[k] = a[i] ;
                a[i] = 0 ;
            }
            --k ;
        }
}
```



【A-32】 求出绝对值最小的元素

给定一个有序整数序列（非递减序），可能包含负数，找出其中绝对值最小的元素，比如给定序列 -5, -3, -1, 2, 8 则返回1。

【分析】由于给定序列是有序的，而这又是搜索问题，所以首先想到二分搜索法。可以分为下面几种情况：

①如果给定的序列中所有的数都是正数，那么数组的第一个元素即是结果；

②如果给定的序列中所有的数都是负数，那么数组的最后一个元素即是结果；

③如果给定的序列中既有正数又有负数，那么绝对值得最小值一定出现在正数和负数的连接处。

为什么？因为对于负数序列来说，右侧的数字比左侧的数字绝对值小，如上面的-5, -3, -1，而对于正整数来说，左边的数字绝对值小，比如上面的2, 8，将这个思想用于二分搜索，可先判断中间元素和两侧元素的符号，然后根据符号决定搜索区间，逐步缩小搜索区间，直到只剩下两个元素。

```

int absMin(int *a, int size)
{ if(size == 1)    return a[0];
  if(a[0] * a[size-1] >= 0)
    return (a[0] >= 0) ? a[0] : a[size-1];
  else
  { int low = 0, high = size-1, mid;
    while(low < high)
    {
      if(low + 1 == high)
        return abs(a[low]) < abs(a[high]) ? a[low] : a[high];
      mid = low + (high - low) / 2;
      if(a[low]*a[mid]*a[high]==0)  return(0);
      if(a[low] * a[mid] > 0)
        low = mid;
      if(a[high] * a[mid] > 0)
        high = mid;
    }
  }
}
    
```



有序（自小到达）绝对值最大呢？

【A-33】 在一个长度为 n 整数序列中，奇数元素和偶数元素各占一半，存放在数组 $A[n]$ 中。

设计一个时间和空间尽可能高效的算法：

`NewSequ(int A[], int n)`

重新排列这些整数，使奇数元素存放在奇数单元，偶数元素存放在偶数单元。说明你所设计算法的时间和空间复杂度。

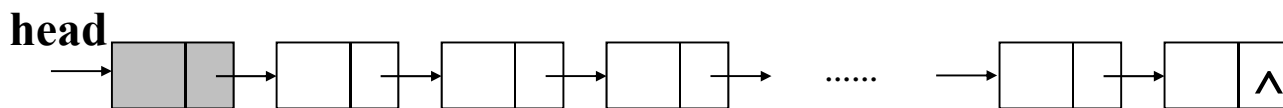
B. 单向链表有关算法

给出结构:

```
typedef struct node {  
    char *data;  
    struct node *next;  
} node_t;
```

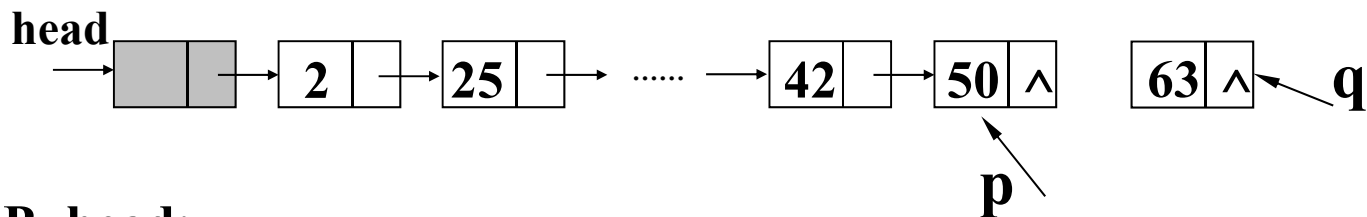
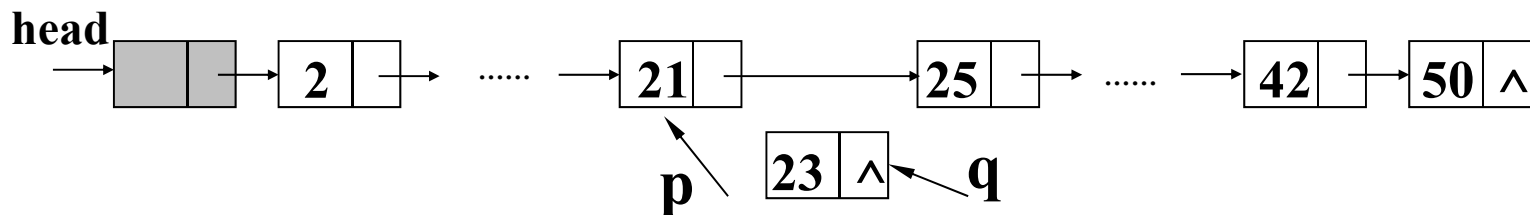
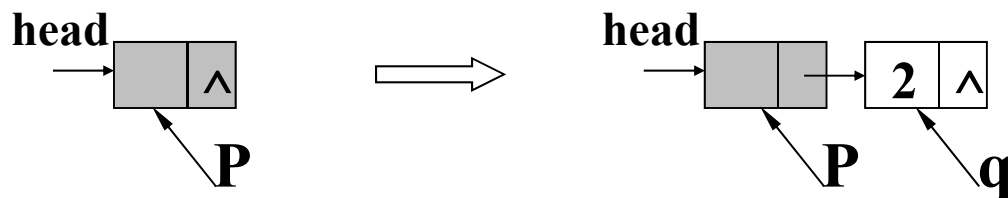
打印链表的函数:

```
void list_display(node_t *head)  
{  
    for (; head; head = head->next)  
        printf("%s ", head->data);  
    printf("\n");  
}
```



【B-1】输入系列整数，建立按升序排列的有序链表

分析：



P=head;

while((p->next!=NULL)&&(p->next->data<=x)) p=p->next;

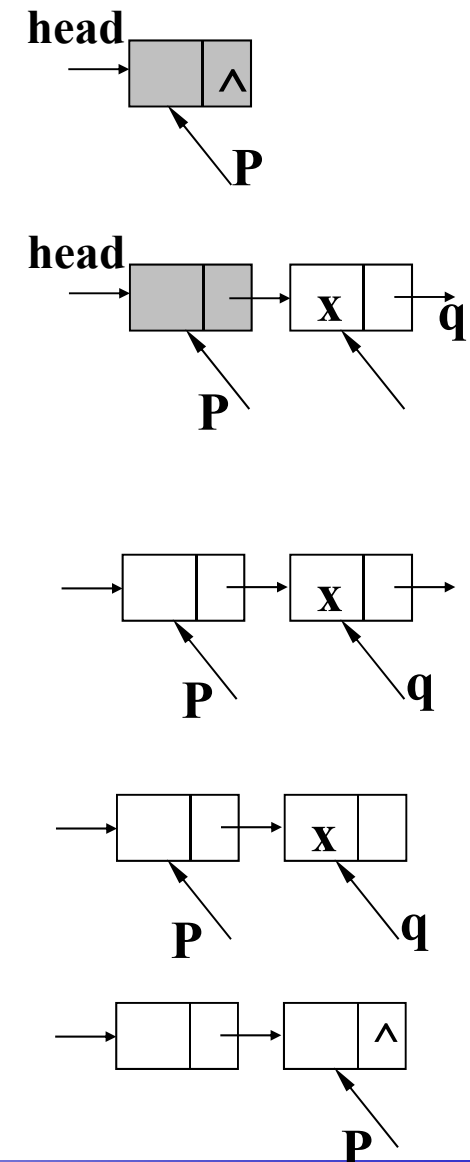
算法

```

NODE *creat_link()
{
    NODE *head,*p,*q;
    int x;
    head=(NODE *)malloc(sizeof(NODE *));
    head->next=NULL;
    scanf("%d",&x);
    while(x!=-999)
    {
        q=(NODE *)malloc(sizeof(NODE *));
        q->data=x;
        q->next=NULL;
        p=head;
        while((p->next!=NULL)&&(p->next->data<=x))
            p=p->next;
        q->next=p->next;
        p->next=q;
        scanf("%d",&x);
    }
    return(head);
}
    
```

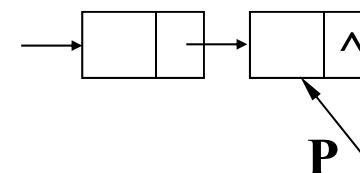
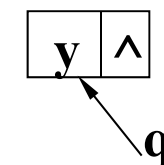
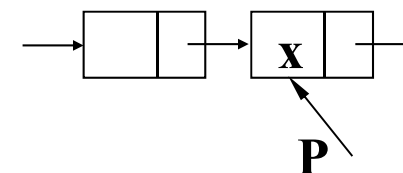
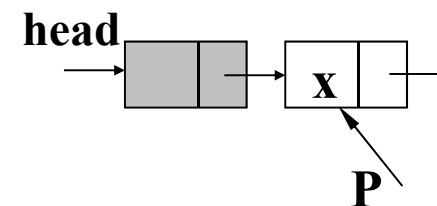
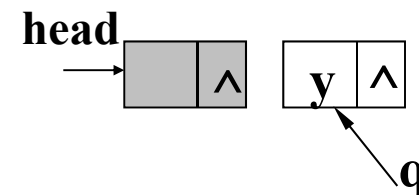
【B-2】删除值为x的结点

```
Void delete_x(NODE *head,int x)
{
    NODE *p,*q;
    p=head;
    while((p->next!=NULL)&&(p->next->data!=x))
        p=p->next;
    If(p->next)
    {
        q=p->next;
        p->next=q->next;
        free(q);
    }
}
```



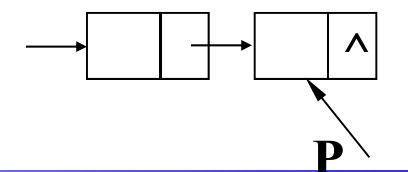
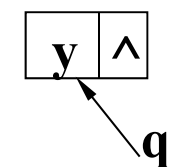
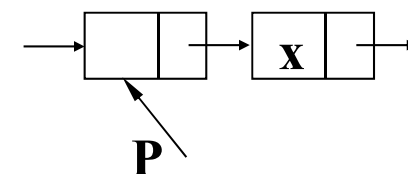
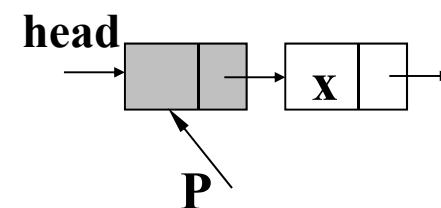
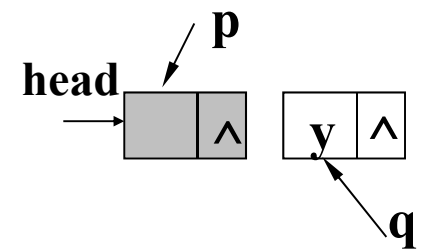
【B-3】 在值为x的结点后插入一个新的结点y,
若x不存在, 则插入在表尾

```
Void insert_after_x(NODE *head,int x,int y)
{
    NODE *p,*q;
    q=(NODE *)malloc(sizeof(NODE *));
    q->data=y ;   q->next=NULL;
    if(head->next) { head->next=q; return;}
    p=head->next;
    while((p->next!=NULL)&&(p->data!=x))
        p=p->next;
    q->next=p->next;
    p->next=q;
}
```



【B-4】 在值为x的结点前插入一个新的结点y,
若x不存在, 则插入在表尾

```
Void insert_after_x(NODE *head,int x,int y)
{
    NODE *p,*q;
    q=(NODE *)malloc(sizeof(NODE *));
    q->data=y;    q->next=NULL;
    p=head;
    while((p->next!=NULL)&&(p->next->data!=x))
        p=p->next;
    q->next=p->next;
    p->next=q;
}
```



【B-5】 删除链表heada自第i个元素起共len个元素，然后将heada插入到链表headb中第j个元素位置

```
NODE *delete_i_len(NODE *heada,int i,int len)
{
    NODE *p,*q;
    int k;
    if (i==1)
        for(k=1;k<=len;k++)
        {
            q=heada;
            heada=heada->next;
            free(q);
        }
    else
    {
        p=heada;
        for(k=1;k<=i-2;k++) p=p->next;
        for(k=1;k<=len;k++)
        {
            q=p->next;
            p->next=q->next;
            free(q);
        }
    }
    return(heada);
}
```

```

NODE *insert_j (NODE *heada,NODE *headb,int j)
{  NODE *p,*q;
   int k;
   p=heada;
   while(p!=NULL) p=p->next ;
   if(j==1)
   { p->next=headb;
     headb=heada;    }
   else
   { q=headb;
     for(k=1;k<=j-2;k++) q=q->next;
     p->next=q->next;
     q->next=heada;          }
   return(headb);
}
    
```

调用:

```

p=delete_i_len(head,i,len);
q=insert_j(p,head,j);
    
```


【B-6】 计算链表长度($O(n)$)

```
int list_len(node_t *head)
{
    int i;
    for (i = 0; head; head = head->next, i++);
    return i;
}
```

【B-7】 反转链表, $O(n)$ 完成反转

算法:

t遍历链表, q记录t的上一个结点, p是一个临时变量用来缓存t的值。

```
void reverse(node_t *head)
{
    node_t *p = 0, *q = 0, *t = 0;
    for (t = head; t; p = t, t = t->next, p->next = q, q = p);
}
```

【B-8】查找倒数第k个元素(尾结点记为倒数第0个)

算法：2个指针p, q初始化指向头结点。p先跑到k结点处, 然后q再开始跑, 当p跑到最后跑到尾巴时, q正好到达倒数第k个. 复杂度O(n)。

```
node_t *_kth(node_t *head, int k)
{
    int i = 0;
    node_t *p = head, *q = head;
    for (; p && i < k; p = p->next, i++);
    if (i < k) return 0;
    for (; p->next; p = p->next, q = q->next);
    return q;
}
```



【B-9】查找中间结点

算法：设两个初始化指向头结点的指针p, q。p每次前进两个结点, q每次前进一个结点, 这样当p到达链表尾巴的时候, q到达了中间. 复杂度 $O(n)$ 。

```
node_t *middle(node_t *head)
{
    node_t *p, *q;
    for (p = q = head; p->next; p = p->next, q = q->next)
    {
        p = p->next;
        if (!(p->next)) break;
    }
    return q;
}
```



【B-10】逆序打印链表

已知链表的头结点, 逆序打印这个链表.使用递归(即让系统使用栈), 时间复杂度 $O(n)$

```
void r_display(node_t *t)
{
    if (t)
    {
        r_display(t->next);
        printf("%s", t->data);
    }
}
```



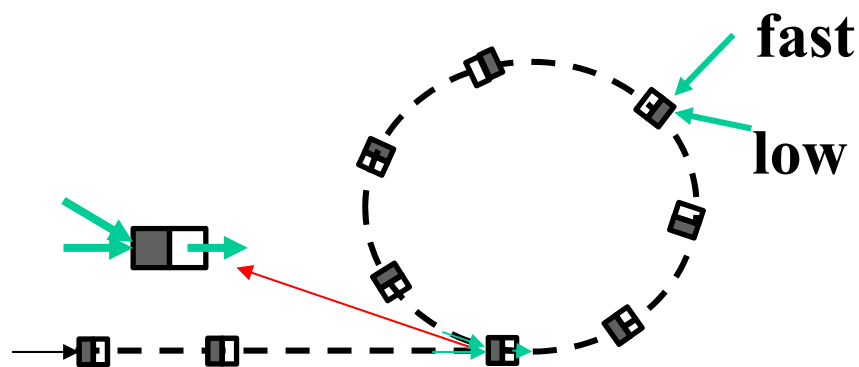
【B-11】线性链表环路问题

问题1：如何判断单链表中是否存在环？

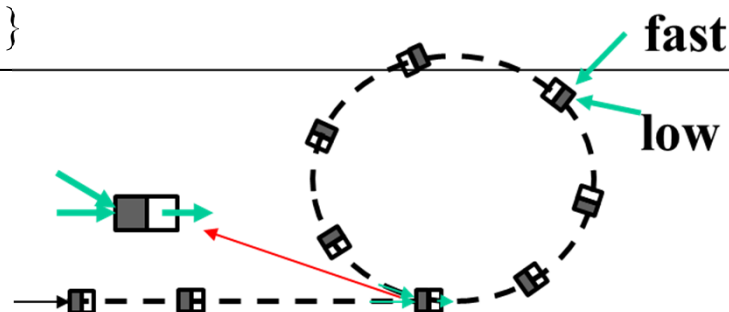
设一快一慢两个指针（Node *fast, *low）同时从链表起点开始遍历，快指针每次移动长度为2，慢指针则为1。

若无环，开始遍历之后fast不可能与low重合，fast或fast->next领先于low到达表尾；

若有环，则fast必然不迟于low先进入环，且由于fast移动步长为2，low移动步长为1，则在low进入环后继续绕环遍历一周之前fast必然能与low重合（且必然是第一次重合）。



```
bool IfCircle(Node* head, Node* &encounter)
{
    Node *fast = head, *slow = head;
    while(fast && fast->next)
    {
        fast = fast->next->next;
        slow = slow->next;
        if(fast == slow)
        {
            encounter = fast;
            return true;
        }
    }
    encounter = NULL;
    return false;
}
```



算法2: 设两个指针p, q, 初始化指向头。p以步长2的速度向前跑, q的步长是1。这样, 如果链表不存在环, p和q肯定不会相遇。如果存在环, p和q一定会相遇。(就像两个速度不同的汽车在一个环上跑绝对会相遇)。复杂度 $O(n)$

```
int any_ring(node_t *head)
{
    node_t *p, *q;
    for (p = q = head; p; p = p->next, q = q->next)
    {
        p = p->next;
        if (!p) break;
        if (p == q) return 1; //yes
    }
    return 0; //fail find
}
```



问题2：若存在环，如何找到环的入口点？

设链起点到环入口点间的距离为 x ，环入口点到 $fast$ 与 low 重合点的距离为 y ，又设在 $fast$ 与 low 重合时 $fast$ 已绕环 n 周 ($n > 0$)，且此时 low 移动总长度为 s ， $fast$ 移动总长度为 $2s$ ，设环的长度为 r 。则：

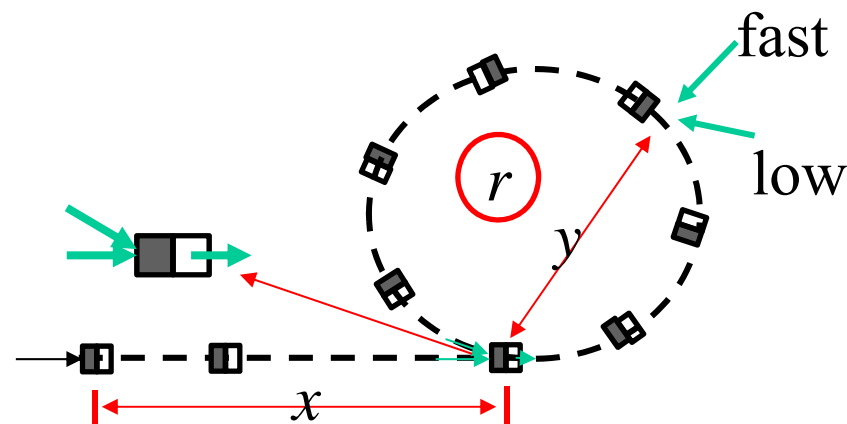
$$s = x + y \quad (1)$$

$$s + nr = 2s, \quad n > 0 \quad (2)$$

由(2)式得 $s = nr$

代入(1)式得

$$x = nr - y \quad (3)$$




现设一指针 $p1$ 从链表起点处开始遍历，指针 $p2$ 从 $encounter$ 处开始遍历， $p1$ 和 $p2$ 移动步长均为 1。当 $p1$ 移动 x 步即到达环的入口点，由(3)式可知，此时 $p2$ 也已移动 x 步即 $nr - y$ 步。由于 $p2$ 是从 $encounter$ 处开始移动，故 $p2$ 移动 nr 步是移回到了 $encounter$ 处，再退 y 步则是到了环的入口点。也即，当 $p1$ 移动 x 步第一次到达环的入口点时， $p2$ 也恰好到达了该入口点。

算法1


```
Node* findEntry1(Node* head, Node* encounter)
{
    Node *p1 = head, *p2 = encounter;
    while(p1 != p2)
    {
        p1 = p1->next;
        p2 = p2->next;
    }
    return p1;
}
```

p扫描的步长为1, q扫描的步长为2。它们的相遇点为图中meet处(环上)。设头指针head到入口点entry之间的距离是K. 当q入环的时候, p已经领先了q为: $d=K\%n$ (n为环的周长)。

meet相对entry的距离(行进方向)为x:
 $(n-d)+x = 2x$ (p行进的路程是q的两倍), 解得 $x=n-d$, 那么当p和q在meet处相遇时, 从head处再发出一个步长为1的指针r, r和q会在entry处相遇! 如算法2。



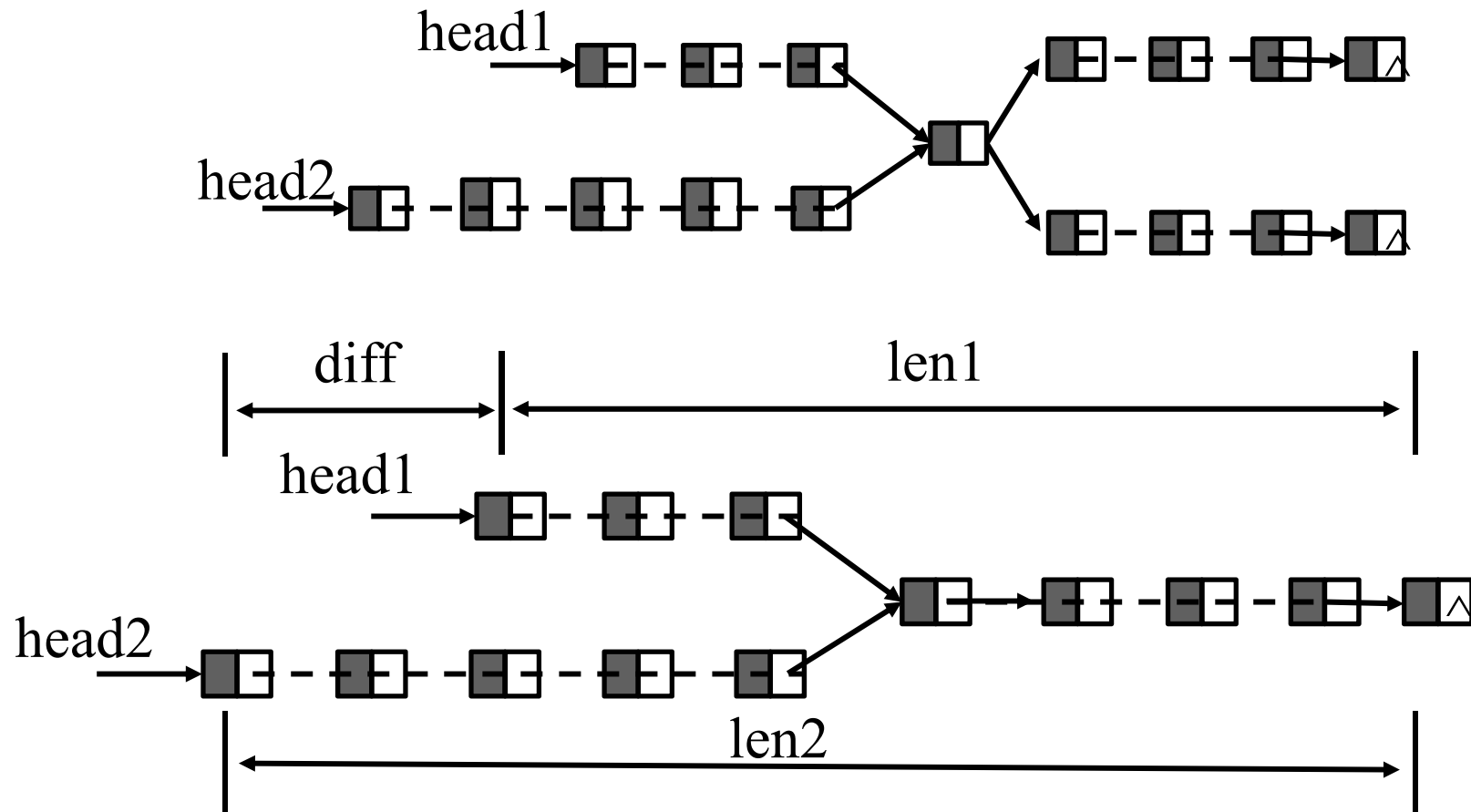
```
node_t *find_entry(node_t *head)
{
    node_t *p, *q, *r;
    for (p = q = head; p; p = p->next, q = q->next)
    {
        p = p->next;
        if (!p) break;
        if (p == q) break;
    }
    if (!p) return 0; //no ring in list
    for (r = head, q = q->next; q != r; r = r->next, q = q->next);
    return r;
}
```



算法2:

初始化三个指针p, q, r全部指向head。然后p以2的速度行进, q以1的速度行进。当p和q相遇的时候, 发出r指针并以1的速度行进, 当p和r相遇返回这个结点。复杂度O(n)

【B-12】线性链表交叉问题



算法1:



```
Node *IfCross(node *head1, node *head2)
{
    node *p1, *p2; int len1=0;int len2=0;int diff = 0;
    if(NULL == head1 || NULL == head2)
        return NULL;    //有为空的链表, 不相交
    p1 = head1;
    p2 = head2;
    while(NULL != p1->next)
    {
        p1 = p1->next;
        len1++;
    }
    while(NULL != p2->next)
    {
        p2 = p2->next;
        len2++;
    }

    if(p1 != p2) //最后一个节点不相同,返回NULL
        return NULL;
```

```
diff = abs(len1 - len2);
if(len1 > len2)
{
    p1 = head1;
    p2 = head2;
}
else
{
    p1 = head2;
    p2 = head1;
}
for(int i=0; i<diff; i++)
    p1 = p1->next;
while(p1 != p2)
{
    p1 = p1->next;
    p2 = p2->next;
}
return p1;    //相交入口点
}
```

算法2：判断两个单链表是否相交

两个指针遍历这两个链表,如果他们的尾结点相同,则必定相交.复杂度 $O(m+n)$

```
int is_intersect(node_t *a, node_t *b)
{
    if (!a || !b) return -1;    //a or b is NULL
    for (; a->next; a = a->next);
    for (; b->next; b = b->next);
    return a == b?1: 0; //return 1 for yes, 0 for no
}
```



假设两个链表a,b.a比b长k个结点($k \geq 0$), 那么当a_ptr,b_ptr两个指针同时分别遍历a,b的时候, 必然b_ptr先到达结尾(NULL),而此时a_ptr落后a的尾巴k个结点。

如果此时再从a的头发出一个指针t,继续和a_ptr一起走,当a_ptr达到结尾(NULL)时,t恰好走了k个结点.此时从b的头发出一个指针s, s和t一起走,因为a比b长k个结点,所以,t和s会一起到达交点。

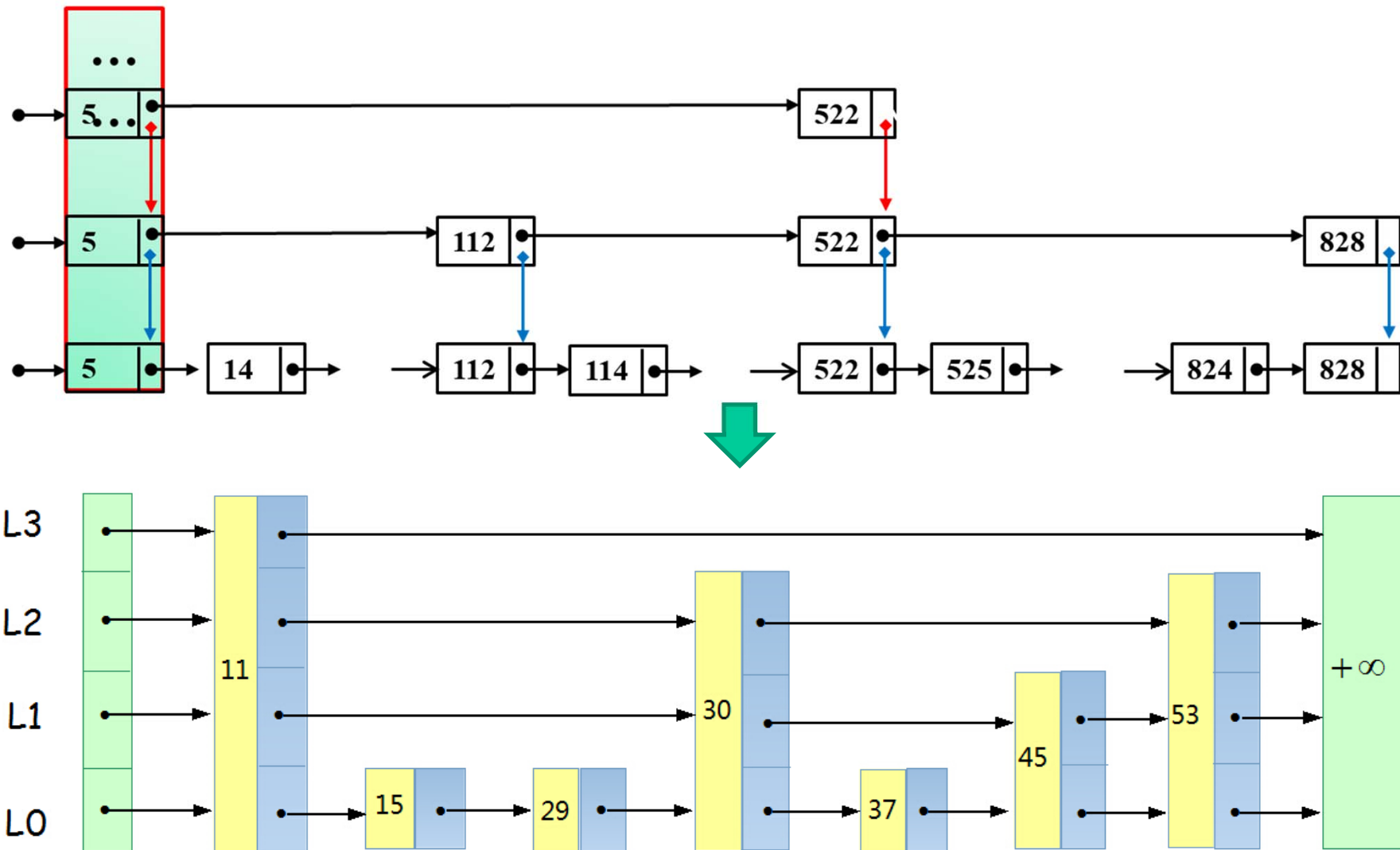
算法3:

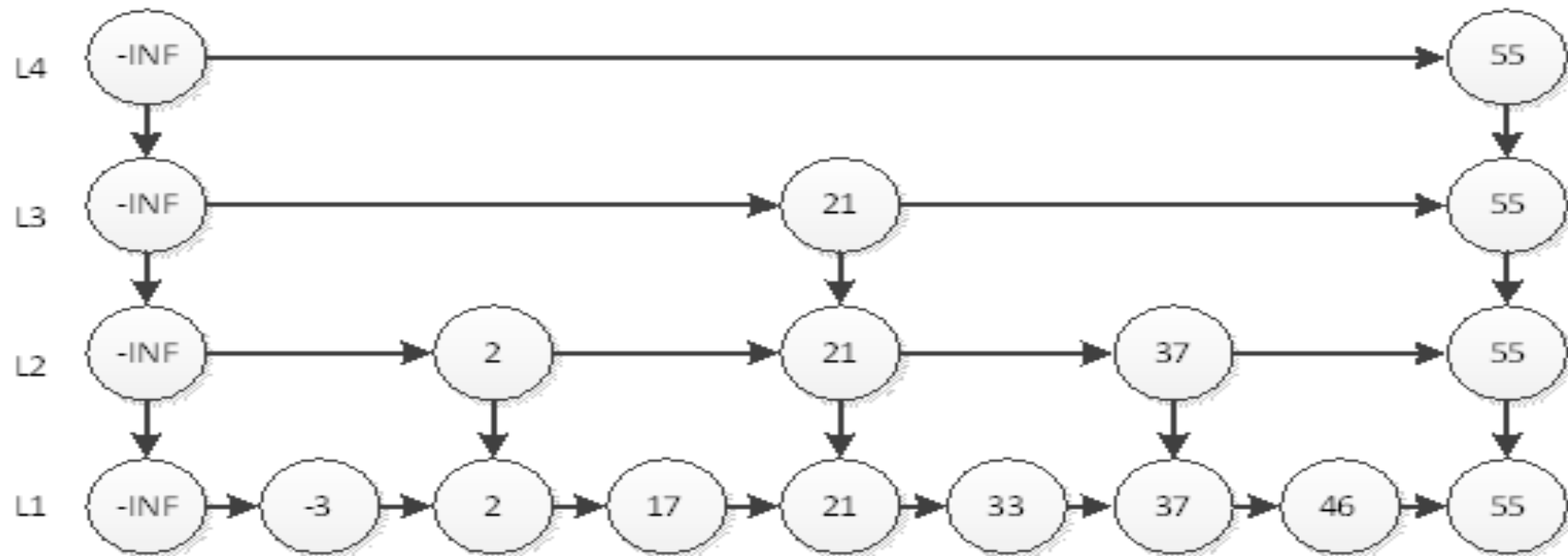
p,q分别遍历链表a,b,假设q先到达NULL,此时从a的头发出一个指针t,当p到达NULL时,从b的头发出s,当s==t的时候即交点。

```
node_t *intersect_point(node_t *a, node_t *b)
{
    //当a,b不相交,函数返回0,否则返回相交结点指针
    node_t *p, *q, *k, *t, *s;
    for (p = a, q = b; p && q; p = p->next, q = q->next);
    k = (p == 0)?q: p;    //k record the pointer not NULL
    t = (p == 0)?b: a;    //if p arrive at tail first, t = b ; else p = a
    s = (p == 0)?a: b;
    for (; k; k = k->next, t = t->next);
    for (; t != s; t = t->next, s = s->next);
    return t;
}
```



【B-13】跳表





Skip List
 又称跳跃表
 简称跳表

- redis和levelDB都是用了它；
- 他在有序链表的基础上进行扩展；
- 解决了有序链表结构查找特定值困难的问题；
- 查找特定值的时间复杂度为 $O(\log n)$ ；
- 一种可以代替平衡树的数据结构；

部分数据库的索引类型：

数据库	索引方式
Oracle	B-树
Mysql	B-tree 和 hash 数据结构
DB2	B+树
Redis（阿里云）	跳表
levelDB（google）	跳表
Sybase	<p>聚集索引（clustered indexes）把表中的记录的物理存储重新排序。因此，每张表只能有一个聚集索引。聚集索引的叶节点包含数据页。</p> <p>非聚集索引中（nonclustered indexes），索引的逻辑顺序与物理存储顺序不同。非聚集索引的叶节点中不包含数据页，而是包含索引行。</p>

C. 二叉树相关算法

【C-1】证明任一棵满二叉树T中的分支数 B 满足：

$$B=2(n_0-1) \quad , \quad \text{其中 } n_0 \text{ 为叶子结点数}$$

证明：

满二叉树中不存在度为1的节点，设度为2的结点数为 n_2

则： $n=n_0+n_2$

又： $n=B+1$

所以有： $B=n_0+n_2-1 \quad , \quad \text{而 } n_0=n_2+1, n_2=n_0-1$

$$B=n_0+n_0-1-1=2(n_0-1)$$

【C-2】 结点总数统计

(1) 求任意二叉树结点总数

$\left\{ \begin{array}{l} f(b)=0 \\ f(b)=1 \\ f(b)=f(b->lchild)+f(b->rchild)+1 \end{array} \right.$	若 $b=NULL$
	若 $b->lchild=b->rchild=NULL$
	其它

```
int nodes( BTREE *b)
{
    int num1,num2 ;
    if ( b==NULL ) return(0);
    else if (( b->lchild==NULL) && ( b->rchild==NULL)) return(1);
    else {
        num1=nodes(b->lchild);
        num2=nodes(b->rchild);
        return(num1+num2+1);
    }
}
```

(2) 求任意二叉树叶子结点数

$f(b)=0$	若 $b=NULL$
$f(b)=1$	若 $b->lchild=b->rchild=NULL$
$f(b)=f(b->lchild)+f(b->rchild)$	其它

```
int leafs( BTREE *b)
{ int num1,num2 ;
  if ( b==NULL ) return(0);
  else if (( b->lchild==NULL) && ( b->rchild==NULL)) return(1);
  else { num1=leafs(b->lchild);
        num2=leafs(b->rchild);
        return(num1+num2);  }
}
```

(3)求任意二叉树单孩子结点（度为1）数

$$\begin{cases} f(b)=0 & \text{若 } b=NULL \\ f(b)=1 & \text{若 } b \rightarrow lchild, b \rightarrow rchild \text{ 其中一个为 } NULL \\ f(b)=f(b \rightarrow lchild)+f(b \rightarrow rchild) & \text{其它} \end{cases}$$

```
int onechild( BTREE *b)
{   int num1,num2 ;
    if ( b==NULL ) return(0);
    else if ( (( b->lchild==NULL) && ( b->rchild<>NULL)) ||
              (( b->rchild==NULL) && ( b->lchild<>NULL)) )
        return(1);
    else { num1=onechild(b->lchild);
          num2=onechild(b->rchild);
          return(num1+num2);   }
}
```

(4) 求任意二叉树双孩子结点（度为2）数

$$\begin{cases} f(b)=0 & \text{若 } b=NULL \\ f(b)=1 & \text{若 } b \rightarrow lchild \neq NULL, \text{ 且 } b \rightarrow rchild \neq NULL \\ f(b)=f(b \rightarrow lchild)+f(b \rightarrow rchild) & \text{其它} \end{cases}$$

```
int twochild( BTREE *b)
{
    int num1,num2 ;
    if ( b==NULL ) return(0);
    else if (( b->lchild!=NULL) && ( b->rchild!=NULL)) return(1);
    else {
        num1=twochild(b->lchild);
        num2=twochild(b->rchild);
        return(num1+num2);
    }
}
```

【C-3】按层序遍历二叉树.

```
Void Level_list_Btree( BTREE T)
{
    QUEUE Q;
    MAKENULL(Q);
    ENQUEUE(T,Q)
    while(!EMPTY(Q))
    {
        P=FRONT(Q);
        DEQUEUE(Q);
        visite(P->data);
        if(P->lchild) ENQUEUE(P->lchild,Q)
        if(P->rchild) ENQUEUE(P->rchild,Q);
    }
}
```

下标	0	1	2	3	...
结点↑					
层号					
度					
双亲					
...					

同时可得到：
宽度、高度等信息
判断：
满二叉树
完全二叉树等

【C-4】在左右链表示的二叉树中，查找值为x的结点，并求x结点所在的层数。

```

BinTree SearchBTree(BinTree *T,DataType x)
{ if(T){ if(T->data==x) return(T);
        SearchBTree(T->lchild,x);
        SearchBTree(T->rchild,x); }
}

Int Inlevel(BinTree *T,DataType x)
{ int Static l=0;
  if(T){ if(l==0) { l++;
                if(T->data==x) return(l);
                if(T->lchild!!T->rchild) l++; }
        else { if(T->data==x) return(l);
                if(T->lchild!!T->rchild) l++;
                else return(0); }
        Inlevel(T->lchild,x);
        Inlevel(T->rchild,x);
  }
}
    
```

【C-5】设计算法，以（**Key**，**LT**，**RT**）的形式打印二叉树，其中**Key**为结点的值，**LT**和**RT**是以括号形式表示的二叉树。

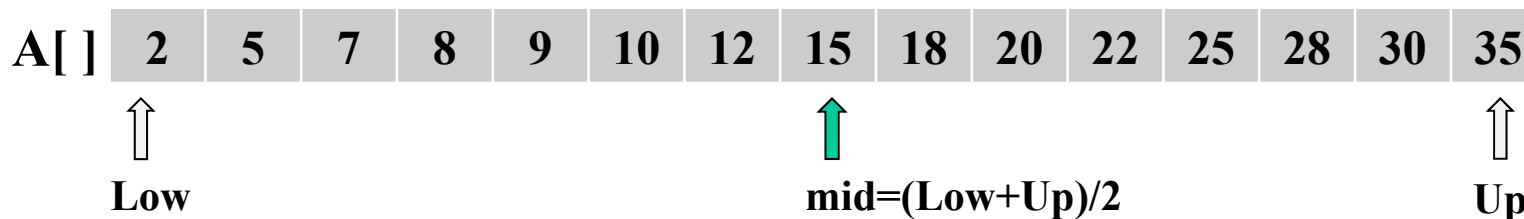
```
Void printTree(BinTree T)
{ if(T) { printf("(%c",T->data);
          printTree(T->lchild);
          if(T->lchild&&T->rchild)
              printf(",");
          printTree(T->rchild);
          printf(")");
        }
}
```


【C-6】对于给定的一个排好序的整数序列，设计一个算法构造一棵二叉树，使得在该二叉树中，以任意结点为根的子树的高度之差的绝对值不大于1。

分析：

- 1) 已知条件“排好序”的整数序列？
- 2) 构造二叉树？二叉排序树/二叉查找树？
- 3) 任意结点为根的子树，左右子树的高度差不超过 $|D_l - D_r| \leq 1$ ，平衡

例如：



$T = \text{CreateTreeFromArray}(A, \text{Low}, \text{Up})$

```

T = new BNODE;
T = a[mid]; root->rchild = root->lchild = NULL;
T->lchild = CreateTreeFromArray(a, Low, mid-1);
T->rchild = CreateTreeFromArray(a, mid + 1, Up);
    
```

```
BTREE *CreateTreeFromArray(int a[], int Low, int Up)
{
    BTREE *root;
    int mid;
    if (Low > Up)
        return NULL;
    else
    {
        mid = (Low + Up) / 2;
        root = new BNODE;
        root->data = a[mid];
        root->rchild = NULL; root->lchild = NULL;
        root->lchild = CreateTreeFromArray(a, Low, mid - 1);
        root->rchild = CreateTreeFromArray(a, mid + 1, Up);
        return root;
    }
}
```

【C-7】设计算法，以 (*Key*, *LT*, *RT*) 的形式打印二叉树，其中 *Key* 为结点的值，*LT* 和 *RT* 是以括号形式表示的二叉树。

```
void DispBTNode(BTREE *B)
{
    if ( B!=NULL)
    {
        printf("%c",B->data);
        if(B->lchild!=NULL || B->rchild!=NULL)
        {
            printf("(");
            DispBTNode(B->lchild);
            if(B->rchild!=NULL)
                printf(",");
            DispBTNode(B->rchild);
            printf(")");
        }
    }
}
```

例如：A (B , C (D , E (F)))

【C-8】以 (*Key*, 层号) 的形式输出二叉树的各结点。

```
void PrintTree(BTREE *T, int n)
{
    if(T)
    {
        printf("(%c,%d)",T->data,n);

        PrintTree(T->lchild,n+1);
        PrintTree(T->rchild,n+1);
    }
}
```

```
Void Layer(BTREE &root)
{   BTREE *que[MAXN];    //循环队列
    int hp , tp , lc , level ;
    if (root!=NULL)
        {   hp=0;tp=1;lc=1; que[0]=root; level=1;
            do{   hp=(hp%MAXN)+1;
                root=que[hp];
                if (root->lchild!=NULL)
                    {   tp=(tp%MAXN)+1;
                        que[tp]=root->lchild;   }
                if (root->rchild!=NULL)
                    {   tp=(tp%MAXN)+1;
                        que[tp]=root->rchild;   }
                printf("Level(“,root->data:2,”)=“,level);
                if (hp==lc) {   level++; lc=tp;}
            }while(hp==tp);
        }
}
```

【C-9】 给定一棵用链表表示的二叉树，其根节点为**Root**，试写出求各结点的层数的算法。

```
BinTree SearchBTree(BTREEe *T, ElementType x)
{ if(T){ if(T->data==x) return(T);
        SearchBTree(T->lchild,x);
        SearchBTree(T->rchild,x); }
}
```

```
Int Inlevel(BTREE *T, ElementType x)
{ int Static l=0;
  if(T){ if(l==0) { l++;
                if(T->data==x) return(l);
                if(T->lchild!!T->rchild) l++; }
        else { if(T->data==x) return(l);
                if(T->lchild!!T->rchild) l++;
                else return(0); }
        Inlevel(T->lchild,x);
        Inlevel(T->rchild,x);
  }
}
```

【C-10】在左右链表示的二叉树中，查找值为x的结点，并求x结点所在的层数。

【C-11】设计算法判断给定的二叉树是否为满二叉树。

{	f(b)=TRUE	若 b->lchild=NULL 且 b->rchild=NULL
	f(b)=FALSE	若 b->lchild, b->rchild 其中的一个为 NULL
	f(b)=f(b->lchild)&&f(b->rchild)	若 b->lchild<>NULL 且 b->rchild<>NULL

```
int isfulltree( BTREE *b)
{
    if ( b!=NULL )
        if (( b->lchild==NULL) && ( b->rchild==NULL)) return(1);
        else if (( b->lchild==NULL) || ( b->rchild==NULL)) return(0);
        else return(isfulltree(b->lchild)&&isfulltree(b->rchild));
}
```

【C-12】设计算法判断给定的二叉树是否为完全二叉树.

算法思想:

根据完全二叉树的定义, 对完全二叉树按照从上到下、从左到右的层次遍历, 应该满足一下两条要求:

- ✓ 某节点没有左孩子, 则一定无右孩子
- ✓ 若某节点缺左或右孩子, 则其所有后继一定无孩子

若不满足上述任何一条, 均不为完全二叉树。

算法思路: 采用层序遍历算法, 用`cm`变量值表示迄今为止二叉树为完全二叉树 (其初值为1, 一旦发现不满足上述条件之一, 则置`cm`为0), `bj`变量值表示迄今为止所有节点均有左右孩子 (其初值为1), 一旦发现一个节点没有左孩子或没有右孩子时置`bj`为0), 在遍历完毕后返回`cm`的值。

```

int IsCompleteBTree1(BTREE *b)
{
    BTREE *Qu[MaxSize],*p;
    int front=0,rear=0, cm=1, bj=1;
    if(b!=NULL) return 1;
    Qu[++rear]=b;
    while(front!=rear)
    {
        p=Qu[++front];
        if(p->lchild==NULL)
        {
            bj=0;
            if(p->rchild!=NULL)
                cm=0;
        }
        else
        {
            if(bj==1)
            {
                Qu[++rear]=p->lchild;
                if(p->rchild==NULL)
                    bj=0;
                else
                    Qu[++rear]=p->rchild;
            }
            else
                cm=0;
        }
    }
    return cm;
}
    
```

//定义一个队列，用于层次遍历

//根节点进队

//*p节点没有左孩子

//没有左孩子但有右孩子
//则不是完全二叉树

//*p节点有左子树
//迄今为止，所有节点均有左右孩子
//左孩子进队
//*p有左孩子但没有右孩子

//右孩子进队

//bj=0:迄今为止，已有节点缺孩子
//而此时*p节点有左孩子，违反（2）

思路：在层序遍历的过程中，找到第一个非满节点。满节点指的是同时拥有左右孩子的节点。在找到第一个非满节点之后，剩下的节点不应该有孩子节点；如果有，那么该二叉树就不是完全二叉树。

```

Bool IsCompleteTree2(BTREE *root)
{
    QUEUE *Q; BTREE *temp; bool flag = false;
    EnQueue(root, Q);
    while(!EMPTY(Q))
    {
        temp = DeQueue(Q);
        if(flag)
        {
            if(temp->lchild || temp->rchild) //
                return false;
        }
        else
        {
            if(temp->lchild && temp->rchild) //左右孩子同时存在，则压入队列
            {
                EnQueue(temp->lchild, Q);
                EnQueue(temp->rchild, Q);
            }
            else if(temp->rchild) //如果只存在右孩子，则，一定不满足满二叉树
                return false;
            else if(temp->lchild)
            {
                EnQueue(temp->lchild); //改变标志位
                flag = true;
            }
            else flag = true;
        }
    }
    return true;
}
    
```

任意的一个二叉树，都可以补成一个满二叉树。这样中间就会有很多空洞。在广度优先遍历的时候，如果是满二叉树，或者完全二叉树，这些空洞是在广度优先的遍历的末尾，所以，但我们遍历到空洞的时候，整个二叉树就已经遍历完成了。而如果，是非完全二叉树，我们遍历到空洞的时候，就会发现，空洞后面还有没有遍历到的值。这样，只要根据是否遍历到空洞，整个树的遍历是否结束来判断是否是完全的二叉树。

算法如下：

```
bool IsCompleteBTree3(BTREE *root)
{
    QUEUE *Q;
    BTREE *ptr;
    EnQueue(root, Q);          // 进行广度优先遍历（层次遍历），并把NULL节点也放入队列
    while ((ptr = DeQueue(Q)) != NULL)
    {
        EnQueue(ptr->left, Q);
        EnQueue(ptr->right, Q);
    }
    while (! EMPTY(Q))        // 判断是否还有未被访问到的节点
    {
        ptr = DeQueue(Q);
        // 有未访问到的的非NULL节点，则树存在空洞，为非完全二叉树
        if (ptr !=NULL) return false;
    }
    return true;
}
```

【C-13】 设二叉树BT采用二叉链表结构存储。试设计一个算法输出二叉树中所有非叶子结点，并求出非叶子结点的个数。

```
int count=0;
void No_Leaf ( BTreeNode *BT )
{   if ( BT )
    {
        if ( BT->lchild || BT->rchild )
        {
            printf("%c,",BT->data);
            count++;
        }
        No_Leaf ( BT->lchild );
        No_Leaf ( BT->rchild );
    }
}
```



```
int No_Leaf(BTREE *BT)
{
    if(BT)
    { static int count=0;
      if ( BT->lchild || BT->rchild )
      { printf("%c,",BT->data);
        count++;
      }
      No_Leaf ( BT->lchild );
      No_Leaf ( BT->rchild );
      return count;
    }
}
```

【C-14】 设二叉树T采用二叉链表结构存储，试设计算法求出二叉树中离根最近的第一个叶子结点。

```
BTNode * Firstleaf( BTNode *bt )
{
    InitQueue( Q ); //初始化队列Q
    if( bt )
    {
        EnQueue( Q , bt );
        while( !EmptyQueue( Q ) )
        {
            DeQueue( Q , p );
            if( !p->lchild && !p->rchild )
                return p;
            if( p->lchild ) EnQueue( Q,p->lchild );
            if( p->rchild ) EnQueue( Q,p->rchild );
        }
    }
}
```

【C-15】输出二叉树所有叶子结点。

```
void DispLeaf(BTREE *B)
{
    if(B!=NULL)
    {
        if(B->lchild==NULL && B->rchild==NULL)
            printf("%c ",B->data);
        else
        {
            DispLeaf(B->lchild);
            DispLeaf(B->rchild);
        }
    }
}
```


【C-16】查找后序遍历的第一个被访问的结点.

已知二叉树BT采用左右链表示法（亦称二叉链表）作为其存储结构，并设计一个非递归算法FirstNode (BTree BT)，直接返回BT的后序遍历的第一个被访问的结点。分析算法的时间复杂度。

“直接”的含义是，不能通过后序遍历得到二叉树的后序序列，然后返回后序序列的第一个结点。

```
BTREE *First_Node_Postlist(BTREE *T)
{
    BTREE *p,*q;
    p=T;
    while(1)
    {
        while(p->lchild)
            p=p->lchild;
        if(!p->rchild)
            return(p);
        else
            p=p->rchild;
    }
}
```



```
BTREE * (BTREE *T)
{
    BTREE *p;
    p=T;
    while(1)
    {
        while(p->rchild)
            p=p->rchild;
        if(!p->lchild)
            return(p);
        else
            p=p->lchild;
    }
}
```



【C-17】 输出根结点到叶子结点的路径。

0

```
void Allpath2(BTNode *b, ElemType path[], int pathlen)
{ //先序遍历方法输出每个叶子结点到根结点的路径序列
  if(b!=NULL)
  {   if (b->lchild==NULL && b->rchild==NULL) //b为叶子
      { //输出栈中所有结点值
        printf("%c->", b->data);
        for (int i=pathlen-1; i>0; i--)
            printf("%c->", path[i]);
        printf("%c\n", path[0]);
      }
      else
      {   path[pathlen++] = b->data;
          Allpath2(b->lchild, path, pathlen);
          Allpath2(b->rchild, path, pathlen);
      }
  }
}
```



【C-18】输出一条最长路径-方法1



```
void MaxPath(BTNode *b, ElemType path[], int pathlen,
             ElemType maxpath[], int *maxpathlen)
{ //先序遍历方法输出一条最长路径
  if(b==NULL) //pathlen和maxpathlen的初值均为0
  { if(pathlen > *maxpathlen) //通过比较求最长路径
    { for(int i=pathlen-1; i>=0; i--) maxpath[i]=path[i];
      *maxpathlen=pathlen;
    }
  }
  else
  { path[pathlen]=b->data; //将当前节点放入路径中
    pathlen++; //路径长度增1
    MaxPath(b->lchild, path, pathlen, maxpath, maxpathlen); //递归扫描左子树
    MaxPath(b->rchild, path, pathlen, maxpath, maxpathlen); //递归扫描右子树
  }
}
```

思考题：如何求一条最短路径？

输出一条最长路径-方法2

```
int Depth(BTREE *T) // 求二叉树T的深度
{
    if(T==NULL)
        return(0);        //空树深度为0
    return 1+(Depth(T->lchild)>Depth(T->rchild)? Depth(T->lchild):Depth(T->rchild));
    //选择左右孩子深度高的然后加上根节点这一层就是深度了
}

void LongestPath(BTREE *T)
{
    if(T!=NULL) //非空二叉树
    {
        printf("%c,",T->data);    //输出根节点
        if(Depth(T->lchild)>Depth(T->rchild))    //判断左右子树的深度
            LongestPath(T->lchild);
        else
            LongestPath(T->rchild);
    }
}
```

【C-19】求最短路径长度.

```
int ShortestDepth(BTREE *root)
{  int rdepth,ldepth;
   if(!root)  //空树
       return 0;
   else if(!root->rchild && !root->lchild)    //只有根节点
       return 1;
   else if(root->lchild && !root->rchild)      //只有左子树
       return ShortestDepth(root->lchild) + 1;
   else if(!root->lchild && root->rchild)      //只有右子树
       return ShortestDepth(root->rchild) + 1;
   ldepth=ShortestDepth(root->lchild);
   rdepth=ShortestDepth(root->rchild);
   if(ldepth<rdepth)
       return ldepth+1;
   else
       return rdepth+1;
}
```

```
int Get_Node_Path(BTREE *root,char x)
```

```
{ BTREE *stack[100],*p;  int top=0,b;
```

```
do{ while(root!=NULL)
```

```
{ if(root->data==x)
```

```
{ printf("%c->",root->data);
```

```
while(top>1) printf("%c->",stack[top--]->data);
```

```
printf("%c.\n",stack[top]->data); return(1); } //找到该结点，输出路径
```

```
top++;
```

```
stack[top]=root;
```

```
root=root->lchild;
```

```
}
```

```
p=NULL; b=1;
```

```
while((top!=0)&& b) //右子树不存在或已访问
```

```
{ root=stack[top];
```

```
if(root->rchild==p)
```

```
{ top--; p=root; } //p指向刚访问
```

```
else
```

```
{ root=root->rchild; b=0; }
```

```
}
```

```
}while(top!=0);
```

```
return(0) //没有找到该结点
```

```
}
```

【C-20】 给定结点到根的路径

算法1：采用后序非递归遍历



```

void Ancestors(BTREE *root,char x)
{
    int b,top=0;
    BTREE *stack[MAX], *p,*q;
    p=root;
    if(p->data==x)    printf("%c 根节点",p->data);
    else
        do{
            while((p->lchild!=NULL)&&(p->data!=x))
                { stack[++top]=p;  p=p->lchild;      }
            if(p->data==x)
                while(top!=0)
                    {  p=stack[top--];
                      printf("%c ",p->data);      }
            q=NULL; b=1;
            while((top!=0)&&b)
                {  p=stack[top];
                  if(p->rchild==q)
                      { top--;  q=p;  }
                  else
                      { p=p->rchild; b=0;  }
                }
            }while(top!=0);
        }
    }
    
```

输出位置的变化

算法2：采用递归遍历

```
void XToRoot( BTREE *B , char path[] , int pathlen , char x)
{ //递归查找给定结点的路径
  int i;
  if(B!=NULL)
  {
    if(B->data == x) //找到结点x
    { printf("%c 到根的路径: %c ", x, x);
      for(i=pathlen-1;i>=0;i--)
        printf("%c ",path[i]);
      printf("\n");
    }
    else
    { path[pathlen]=B->data; //当前结点放入路径中
      pathlen++;
      XToRoot(B->lchild, path , pathlen, x); //递归扫描左子树
      XToRoot(B->rchild, path , pathlen, x); //递归扫描右子数
      pathlen--;
    }
  }
}
```



【C-22】给定两个结点的最近双亲.

可以分别找到两个结点到根的路径

从根开始，两条路径上最后一个相同的结点为二者的共同双亲。

三叉链表


二叉树的三叉链表存储表示

```
typedef struct BiTPNode
{
    TElemType data;
    struct BiTPNode *parent, *lchild, *rchild;
} BiTPNode, *BiPTree;
```

【C-23】构造三叉链表表示的二叉树T.

C22

```
void CreateBiTree(BiPTree *T)
{
    TElemType ch;
    scanf("%c",&ch);
    if(ch=='#') /* 空 */
        *T=NULL;
    else
    {
        *T=(BiPTree)malloc(sizeof(BiTPNode)); /* 动态生成根结点 */
        if(!*T) exit(OVERFLOW);
        (*T)->data=ch; /* 给根结点赋值 */
        (*T)->parent=NULL; /* 根结点无双亲 */
        CreateBiTree(&(*T)->lchild); /* 构造左子树 */
        if((*T)->lchild) /* 有左孩子 */
            (*T)->lchild->parent=*T; /* 给左孩子的双亲域赋值 */
        CreateBiTree(&(*T)->rchild); /* 构造右子树 */
        if((*T)->rchild) /* 有右孩子 */
            (*T)->rchild->parent=*T; /* 给右孩子的双亲域赋值 */
    }
}
```



【C-24】 销毁二叉树T.

```
void DestroyBiTree(BiPTree *T)
{
    if(*T) /* 非空树 */
    {
        if((*T)->lchild) /* 有左孩子 */
            DestroyBiTree(&(*T)->lchild); /* 销毁左孩子子树 */
        if((*T)->rchild) /* 有右孩子 */
            DestroyBiTree(&(*T)->rchild); /* 销毁右孩子子树 */
        free(*T); /* 释放根结点 */
        *T=NULL; /* 空指针赋0 */
    }
}
```

```

void InOrder_NoStatic1(BiPTree PT)
{ BiPTree p,pr;
  if(PT !=NULL)
  { p=PT;
    while(p!=NULL)
    { while(p->lchild!=NULL)    p=p->lchild;
      printf("%c ",p->data);//访问最左孩子
      if(p->rchild!=NULL)
        p=p->rchild; continue; //如果最左孩子存在有孩子,返回第一步操作:找右孩子的最左孩子
                                //访问完左孩子,及左孩子的子孩子。应该访问根结点

      while(p->parent)
      { pr=p->parent;
        if(pr->rchild) //双亲存在右孩子时
          if(pr->rchild==p)
            { p=pr; continue; } //当双亲只有右孩子时
          else
            { //当是由左孩子指向双亲,又存在右孩子时
              printf("%c ",p->data);//先访问根节点
              p=pr->rchild; //再访问右孩子,重新当成一颗树;回到第一步:break的原因
              break;        }
            else
            { p=pr; //不存在右孩子时,直接访问该根结点,p指向上一层;继续循环 visit(pr->data);
              printf("%c ",p->data);        }
            }//while end
          if(!p->parent) break;
        } //while
      } //if
    }
  }

```

【C-25】二叉树三叉链表 非递归中序遍历1.



```

void InOrder_NoStatic2 (TriTree PT, void (*visit)(TElemType))
{   TriTree p=PT, pr;
    while(p)
    {   if (p->lchild)
        p = p->lchild;    //寻找最左下结点
        else {   visit(p->data); //找到最左下结点并访问
                if (p->rchild)
                    p = p->rchild;    //若有右子树，转到该子树，继续寻找最左下结点
                else {   pr = p;    //否则返回其父亲
                        p = p->parent;
                        while (p && (p->lchild != pr || !p->rchild))
                        {   //若其不是从左子树回溯来的，或左结点的父亲并没有右孩子
                            if (p->lchild == pr) //若最左结点的父亲并没有右孩子
                                visit(p->data); //直接访问父亲（不用转到右孩子）
                            pr = p; //父亲已被访问，故返回上一级
                            p = p->parent; //该while循环沿双亲链一直查找，若无右孩子则访问，直至
                                找到第一个有右孩子的结点为止（但不访问该结点，留给下
                                步if语句访问）
                        }
                        if (p)
                        {   //访问父亲，并转到右孩子（经上步while处理，可以确定此时p有右孩子）
                            visit(p->data);
                            p = p->rchild;
                        }
                    }
                }
        }
    }
}
    
```

**【C-26】二叉树三叉链表
非递中序归遍历2.**

```

void InOrder_NoStatic3 (TriTree PT, void (*visit)(TElemType))
{
    TriTree p=PT, pr;
    while(p)
    {
        if (p->lchild) p = p->lchild;
        else {
            visit(p->data);
            if (p->rchild) p = p->rchild;
            else {
                pr = p;
                p = p->parent;
                while (p && (p->lchild != pr || !p->rchild))
                {
                    if (p->lchild == pr) visit(p->data);
                    pr = p;
                    p = p->parent;
                }
                if (p) {
                    visit(p->data);
                    p = p->rchild;
                }
            }
        }
    }
}
    
```

**【C-27】二叉树三叉链表
非递归中序遍历3.**

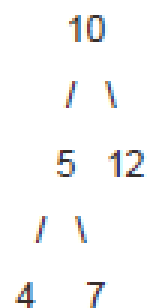


【C-28】二叉树三叉链表 非递归后续遍历.

```
void PostByParent(TriTree T)
{
    BTREE *pre, *cur;
    cur=T;
    while(cur)
    {
        while(cur->lchild) cur = cur->lchild; //遇到左子结点, 则一直向左
        if(cur->rchild) //如果结点有右子树进入右子树
            cur = cur->rchild;
        else //否则, 输出并向上回溯
        {
            printf("1%c",cur->data);
            pre = cur;    cur = cur->parent;
            while(cur)
            {
                if(cur->lchild == pre) //如果回溯时, 是从左子树回溯的
                {
                    if(cur->rchild)
                    {
                        cur = cur->rchild;
                        break; //break后回到第6行
                    }
                }
                else //从右子树回溯的, 则继续向上多回溯一个
                {
                    printf("2%c",cur->data); //从右子树回溯,要输出父结点
                    pre = cur;    cur = cur->parent;
                }
            }
        }
    }
    //while(cur!=T);
}
```

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。

【C-29】打印二叉树中与给定整数相等的所有路径。

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，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。

```
void findpath(Node* root,vector<int>& nodes,int sum)
{
    if(root == NULL) return;
    nodes.push_back(root->value);
    if(root->left == NULL && root->right == NULL)
    {

        if(root->value == sum)
        {
            for(int i=0;i<nodes.size();i++)
                cout<<nodes[i]<<" ";
            cout<<endl;
        }
    }
    else
    {
        if(root->left != NULL)
        {
            findpath(root->left,nodes,sum-root->value);
        }
        if(root->right != NULL)
        {
            findpath(root->right,nodes,sum-root->value);
        }
    }
    nodes.pop_back();
}
```

```
struct Node
{
    int value;
    Node* left;
    Node* right;
    Node(){left=NULL;right=NULL;}
    Node(int v){value=v;left=NULL;right=NULL;}
};
```

```
int main()
{
    Node *tmp ;
    Node* root = new Node(10);
    tmp = new Node(5);
    root->left = tmp ;
    tmp = new Node(12);
    root->right = tmp;
    tmp = new Node(4);
    root->left->left = tmp;
    tmp = new Node(7);
    root->left->right = tmp;
    vector<int> v;
    findpath(root,v,22);
    return 0;
}
```

【C-30】将二叉链表存储的二叉树转换到按照完全二叉树存储的数组中。

方法一：

```
char array[100];
int Num=0;
```

```
void btree2array1(BTREE *T, int i)
{
    if(T == NULL)
        array[i] = '-';
    else
    {
        array[i] = T->data;
        Num=i;
        btree2array1(T->lchild, 2*i);
        btree2array1(T->rchild, 2*i+1);
    }
}
```

调用：

```
btree2array1(T,1);
for(i=1;i<=Num;i++)
    printf("%c ",array[i]);
printf("\n");
```


方法二:

```
char array[100];
int btree2array2(BTREE *T,int i)
{
    static int n=0;
    if(T == NULL)
        array[i] = '-';
    else
    { array[i] = T->data;
      n=i;
      btree2array2(T->lchild, 2*i);
      btree2array2(T->rchild, 2*i+1);
    }
    return(n);
}
```

调用:

```
for(i=1;i<=btree2array2(T,1);i++)
    printf("%c ",array[i]);
printf("\n");
```

方法三:

```
int btree2array3(BTREE *T,char *array,int i)
{
    static n=0;
    if(T == NULL)
        array[i] = '-';
    else
    { array[i] = T->data;
      n=i;
      btree2array3(T->lchild, array,2*i);
      btree2array3(T->rchild, array,2*i+1);
    }
    return(n);
}
```

调用:

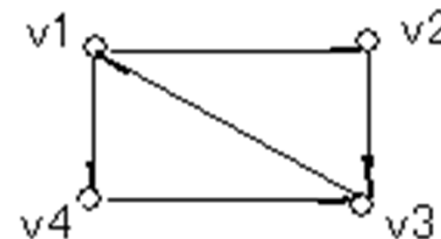
```
char p[100];
for(i=0;i<100;i++) p[i]='-';
for(i=1;i<=btree2array3(T,p,1);i++)
    printf("%c ",p[i]);
printf("\n");
```

- 01. 二叉树的问题
- 02. 1. 二叉树三种周游 (traversal) 方式:
- 03. 2. 怎样从顶部开始逐层打印二叉树结点数据
- 04. 3. 如何判断一棵二叉树是否是平衡二叉树
- 05. 4. 设计一个算法, 找出二叉树上任意两个节点的最近共同父结点, 复杂度如果是 $O(n^2)$ 则不得
- 06.
- 07. 分。
- 08. 5. 如何不用递归实现二叉树的前序/后序/中序遍历?
- 09. 6. 在二叉树中找出和为某一值的所有路径
- 10. 7. 怎样编写一个程序, 把一个有序整数数组放到二叉树中?
- 11. 8. 判断整数序列是不是二叉搜索树的后序遍历结果
- 12. 9. 求二叉树的镜像
- 13. 10. 一棵排序二叉树 (即二叉搜索树BST), 令 $f = (\text{最大值} + \text{最小值}) / 2$, 设计一个算法, 找出距
- 14.
- 15. 离 f 值最近、大于 f 值的结点。复杂度如果是 $O(n^2)$ 则不得分。
- 16. 11. 把二叉搜索树转变成排序的双向链表
- 17. 12. 打印二叉树中的所有路径 (与题目5很相似)

D. 图相关题目

【D-8】在右图所示的有向图中：

- (1) 该图是强连通的吗？若不是，则给出其强连通分量。
- (2) 请给出所有的简单路径及有向环。
- (3) 请给出每个顶点的度，入度和出度。
- (4) 请给出其邻接表、邻接矩阵及逆邻接表。



(1) 该图是强连通的，所谓强连通是指有向图中任意顶点都存在到其他各顶点的路径。

(2) 简单路径是指在一条路径上只有起点和终点可以相同的路径：

有 v_1v_2 、 v_2v_3 、 v_3v_1 、 v_1v_4 、 v_4v_3 、 $v_1v_2v_3$ 、 $v_2v_3v_1$ 、 $v_3v_1v_2$ 、 $v_1v_4v_3$ 、 $v_4v_3v_1$ 、 $v_3v_1v_4$ 、

另包括所有有向环，有向环如下：

$v_1v_2v_3v_1$ 、 $v_1v_4v_3v_1$ (这两个有向环可以任一顶点作为起点和终点)

(3) 每个顶点的度、入度和出度：

$D(v_1)=3$ $ID(v_1)=1$ $OD(v_1)=2$

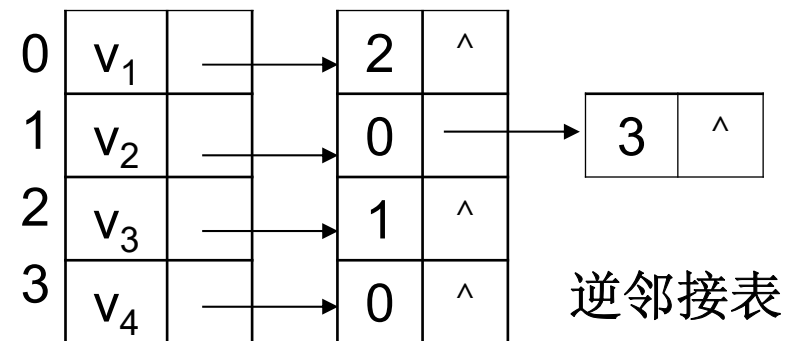
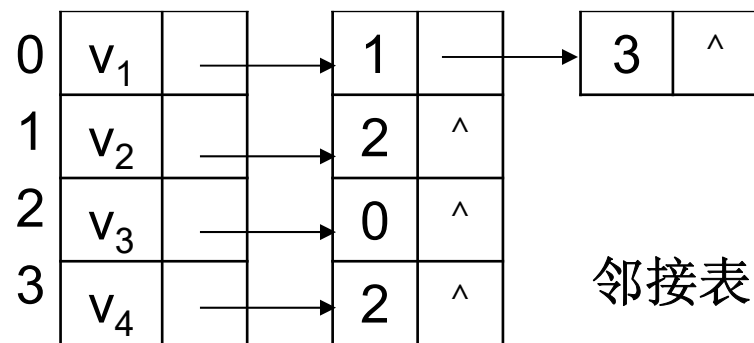
$D(v_2)=2$ $ID(v_2)=1$ $OD(v_2)=1$

$D(v_3)=3$ $ID(v_3)=2$ $OD(v_3)=1$

$D(v_4)=2$ $ID(v_4)=1$ $OD(v_4)=1$

(4)邻接表:

vertex firstedge next



邻接矩阵:

0	1	0	1
0	0	1	0
1	0	0	0
0	0	1	0

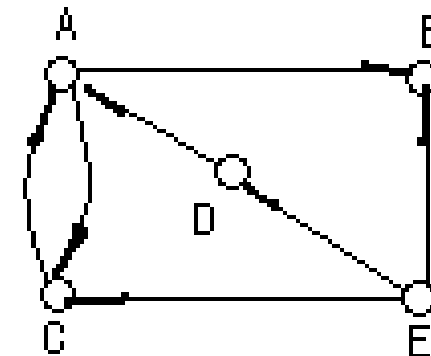
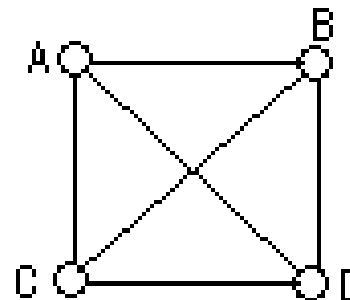
【D-9】 假设图的顶点是A, B...,
请根据下述的邻接矩阵画出相应的无向图或有向图。

0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

(a)

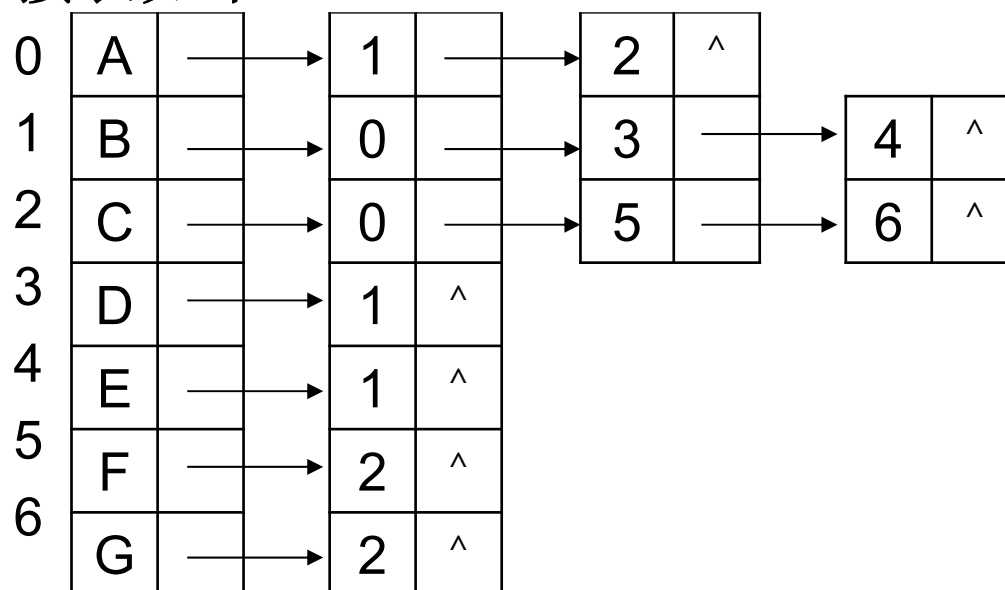
0	1	1	0	0
0	0	0	1	0
0	0	0	1	0
1	0	0	0	1
0	1	0	1	0

(b)



【D-10】 假设一棵完全二叉树包括A,B,C...等七个结点，写出其邻接表和邻接矩阵。

邻接表如下：



邻接矩阵如下：

0	1	1	0	0	0	0
1	0	0	1	1	0	0
1	0	0	0	0	1	1
0	1	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	1	0	0	0	0

【D-11】对 n 个顶点的无向图和有向图，采用邻接矩阵和邻接表表示时，如何判别下列有关问题？

- (1) 图中有多少条边？
- (2) 任意两个顶点 i 和 j 是否有边相连？
- (3) 任意一个顶点的度是多少？

对于 n 个顶点的无向图和有向图，用邻接矩阵表示时：

- (1) 设 m 为矩阵中非零元素的个数

无向图的边数 = $m/2$

有向图的边数 = m

- (2) 无论是有向图还是无向图，在矩阵中第 i 行,第 j 列的元素若为非零值，则该两顶点有边相连。

- (3) 对于无向图，任一顶点 i 的度为第 i 行中非零元素的个数。

对于有向图，任一顶点 i 的入度为第 i 列中非零元素的个数，出度为第 i 行中非零元素的个数，度为入度出度之和。

当用邻接表表示时:

(1)对于无向图, 图中的边数=边表中结点总数的一半。

对于有向图, 图中的边数=边表中结点总数。

(2)对于无向图, 任意两顶点间是否有边相连, 可看其中一个顶点的邻接表, 若表中的**adjvex**域有另一顶点位置的结点, 则表示有边相连。

对于有向图, 则表示有出边相连。

(3)对于无向图, 任意一个顶点的度则由该顶点的边表中结点的个数来决。

对于有向图, 任意一个顶点的出度由该顶点的边表中结点的个数来决定, 入度则需遍历各顶点的边表。

(用逆邻接表可容易地得到其入度。)

```

void NonSuccFirstTopSort(ALGraph G)
{ //优先输出无后继的顶点,此处用逆邻接表存储
    int outdegree[MaxVertexNum]; //出度向量, 此处MaxVertexNum>=G.n
    SeqStack S; //将栈中data向量的基类型改为int
    int i,j,count=0; //count对输出的顶点数目计数, 初值为0
    EdgeNode *p;
    for(i=0;i<G.n;i++)
    { outdegree[i]=0;
      visited[i]=FALSE; //标志向量初始化
    }
    for(i=0;i<G.n;i++)
    { for(p=G.adjlist[i].firstedge;p;p=p->next) //扫描i的入边表
      outdegree[p->adjvex]++; //设p->adjvex=j, 则将<j,i>的起点j出度加1
    }
    InitStack(&s);
}
    
```

```

for(i=0;i<G.n;i++)
    if (outdegree[i]==0)
        Push(&S, i);
while(!StackEmpty(S))
{ i=pop(&s);visited[i]=TRUE;
  count++;
  for(p=G.adjlist[i].firstedge;p;p=p->next)
  { j=p->adjvex;
    outdegree[j]--;
    if (outdegree[j]==0)
        Push(&S, j);
  }
}
if (count<G.n)
{ printf("G中存在有向环，排序失败!");
  for(i=0;i<G.n;i++)
    if (visited[i]==FALSE)
        printf("%c",G.adjlist[i].vertex);
}
else printf("G中无有向环!");
}
    
```

//出度为0的顶点i入栈
 //栈非空，有出度为0的顶点
 //顶点计数加1
 //修改以i为弧头的弧的弧尾顶点的出度
 //将新生成的出度为0的顶点入栈
 //出度为0的顶点j入栈
 //end of for
 //end of while
 //输出顶点数小于n

【D-19】 写一算法求有向图的所有根(若存在)，分析算法的时间复杂度。

```
typedef enum{FALSE, TRUE} Boolean; //FALSE为0, TRUE为1
Boolean visited[MaxVertexNum]; //访问标志向量是全局量
void DFSTraverse(ALGraph *G)
{
    //对以邻接表表示的有向图G, 求所有根
    int i,j;
    for (j=0;j<G->n;j++)
    {
        for(i=0;i<G->n;i++)
            visited[i]=FALSE; //标志向量初始化
        DFS(G, j); //以vj为源点开始DFS搜索, 也可用BFS(G,j)
        i=0;
        while(i<G->n)&&(visited[i]==TRUE)
            i++;
        if (i==G->n) printf("root:%c",G->adjlist[j].vertex);
    }
} //DFSTraverse
```

该算法的为二重循环，若调用的DFS算法的复杂度为 $O(n+e)$ ，该算法的时间复杂度为 $O(n(n+e))$ ，调用的DFSM算法的复杂度为 $O(n*n)$ ，所以该算法的时间复杂度为 $O(n^3)$