第三章 树

树是一种非线性的数据结构，用它能很好地描述有分支和层次特性的数据集合。树型结构在现实世界中广泛存在，如社会组织机构的组织关系图就可以用树型结构来表示。树在计算机领域中也有广泛应用，如在编译系统中，用树表示源程序的语法结构。在数据库系统中，树型结构是数据库层次模型的基础，也是各种索引和目录的主要组织形式。在许多算法中，常用树型结构描述问题的求解过程、所有解的状态和求解的对策等。在这些年的国内、国际信息学奥赛、大学生程序设计比赛等竞赛中，树型结构成为参赛者必备的知识之一，尤其是建立在树型结构基础之上的搜索算法。

在树型结构中，二叉树是最常用的结构，它的分支个数确定、又可以为空、并有良好的递归特性，特别适宜于程序设计，因此也常常将一般树转换成二叉树进行处理。

第一节 树的概念

一、树的基本概念

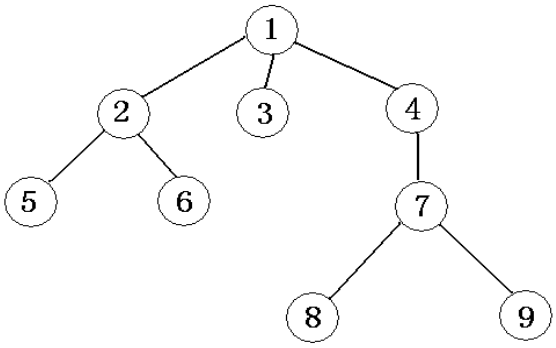
一棵树是由 n（n>0）个元素组成的有限集合，其中：

（1）每个元素称为结点(node)；

（2）有一个特定的结点，称为根结点或树根（root）；

（3）除根结点外，其余结点能分成 m（m>=0）个互不相交的有限集合 $T_0,T_1,T_2,\cdots\cdots T_{m-1}$。其中的每个子集又都是一棵树，这些集合称为这棵树的子树。

如下图是一棵典型的树：



二、树的基本概念

A． 树是递归定义的；

B． 一棵树中至少有 1 个结点。这个结点就是根结点，它没有前驱，其余每个结点都有唯一的一个前驱结点。每个结点可以有 0 或多个后继结点。因此树虽然是非线性结构，但也是有序结构。至于前驱后继结点是哪个，还要看树的遍历方法，我们将在后面讨论；

C． 一个结点的子树个数，称为这个结点的度（degree，结点 1 的度为 3，结点 3 的度为 0）；度为 0 的结点称为叶结点（树叶 leaf，如结点 3、5、6、8、9）；度不为 0 的结点称为分支结点（如结点 1、2、4、7）；根以外的分支结点又称为内部结点（结点 2、4、7）；树中各结点的度的最大值称为这棵树的度（这棵树的度为 3）。

D． 在用图形表示的树型结构中，对两个用线段（称为树枝）连接的相关联的结点，称上端的结点为下端结点的父结点，称下端的结点为上端结点的子结点。称同一个父结点的多个子结点为兄弟结点。如结点 1 是结点 2、3、4 的父结点，结点 2、3、4 是结点 1 的子结点，它们又是兄弟结点，同时结点 2 又是结点 5、6 的父结点。称从根结点到某个子结点所经过的所有结点为这个子结点

的祖先。如结点 1、4、7 是结点 8 的祖先。称以某个结点为根的子树中的任一结点都是该结点的子孙。如结点 7、8、9 都是结点 4 的子孙。

E．定义一棵树的根结点的层次（level）为 0，其它结点的层次等于它的父结点层次加 1。如结点 2、3、4 的层次为 1，结点 5、6、7 的层次为 2，结点 8、9 的层次为 3。一棵树中所有的结点的层次的最大值称为树的深度（depth）。如这棵树的深度为 3。

F．对于树中任意两个不同的结点，如果从一个结点出发，自上而下沿着树中连着结点的线段能到达另一结点，称它们之间存在着一条路径。可用路径所经过的结点序列表示路径，路径的长度等于路径上的结点个数减 1。如上图中，结点 1 和结点 8 之间存在着一条路径，并可用（1、4、7、8）表示这条路径，该条路径的长度为 3。注意，不同子树上的结点之间不存在路径，从根结点出发，到树中的其余结点一定存在着一条路径。

G．森林（forest）是 m(m>=0)棵互不相交的树的集合。

三、树的存储结构

方法 1：数组，称为“父亲表示法”。

```
const m=10;                    //树的结点数
type node=record
       data:integer;           //数据域
       parent:integer          //指针域
     end;
var tree:array[1..m] of node;
```

优缺点：利用了树中除根结点外每个结点都有唯一的父结点这个性质。很容易找到树根，但找孩子时需要遍历整个线性表。

方法 2：树型单链表结构，称为“孩子表示法”。每个结点包括一个数据域和一个指针域（指向若干子结点）。称为“孩子表示法”。假设树的度为 10，树的结点仅存放字符，则这棵树的数据结构定义如下：

```
const m=10;                    //树的度
type tree=^node;
     node=record
       data:char;              //数据域
       child:array[1..m] of tree    //指针域，指向若干孩子结点
     end;
var t:tree;
```

缺陷：只能从根（父）结点遍历到子结点，不能从某个子结点返回到它的父结点。但程序中确实需要从某个结点返回到它的父结点时，就需要在结点中多定义一个指针变量存放其父结点的信息。这种结构又叫带逆序的树型结构。

方法 3：树型双链表结构，称为“父亲孩子表示法”。每个结点包括一个数据域和二个指针域（一个指向若干子结点，一个指向父结点）。假设树的度为 10，树的结点仅存放字符，则这棵树的数据结构定义如下：

```
const m=10;                    //树的度
type tree=^node;
     node=record
       data:char;              //数据域
```

```
        child:array[1..m] of tree;  //指针域，指向若干孩子结点
        father:tree                 //指针域，指向父亲结点
    end;
var t:tree;
```

方法 4：二叉树型表示法，称为“孩子兄弟表示法”。也是一种双链表结构，但每个结点包括一个数据域和二个指针域（一个指向该结点的第一个孩子结点，一个指向该结点的下一个兄弟结点）。称为“孩子兄弟表示法”。假设树的度为 10，树的结点仅存放字符，则这棵树的数据结构定义如下：

```
type tree=^node;
     node=record
       data:char;
       firstchild,next: tree;
    end;
    var t:tree;
```

四、树的遍历

在应用树结构解决问题时，往往要求按照某种次序获得树中全部结点的信息，这种操作叫作树的遍历。遍历的方法有多种，常用的有：

A、先序（根）遍历：先访问根结点，再从左到右按照前序思想遍历各棵子树。
如上图前序遍历的结果为：125634789；

B、后序（根）遍历：先从左到右遍历各棵子树，再访问根结点。如上图后序遍历的结果为：562389741；

C、层次遍历：按层次从小到大逐个访问，同一层次按照从左到右的次序。
如上图层次遍历的结果为：123456789；

D、叶结点遍历：有时把所有的数据信息都存放在叶结点中，而其余结点都是用来表示数据之间的某种分支或层次关系，这种情况就用这种方法。如上图按照这个思想访问的结果为：56389；

大家可以看出，AB 两种方法的定义是递归的，所以在程序实现时往往也是采用递归的思想。既通常所说的“深度优先搜索”。如用前序遍历编写的过程如下：

```
procedure tra1(t,m)                          //递归
begin
    if t <>nil then begin
                    write(t^.data,’ ’);      //访问根结点
                    for I:=1 to m do         //前序遍历各子树
                        tra1(t^.child[I],m);
                   end;
end;
```

C 这种方法应用也较多，实际上是我们将的“广度优先搜索”。思想如下：若某个结点被访问，则该结点的子结点应登录，等待被访问。顺序访问各层次上结点，直至不再有未访问过的结点。为此，引入一个队列来存储等待访问的子结点，设一个队首和队尾指针分别表示出队、进队的下标。程序框架如下：

```
Const n=100;
Var hend,tail,I:integer;
    Q:array[1..n] of tree;
```

```
    P:tree;
Begin
  tail:=1;head:=1;
  Q[tail]:=t;                          //t 进队
  Tail:=tail+1;
  While ( head<tail) do                //队列非空
    Begin
      P:=q[head];                      //取出队首结点
      head:=head+1;
      Write(p^.data,‘    ‘);           //访问某结点
      For I:=1 to m do                 //该结点的所有子结点按顺序进队
        If p^.child[I]<>nil   then  begin
                                        q[tail]:=p^.child[I];
                                        tail:=tail+1;
                                      end;
    End;
End;
```

【例 1】单词查找树

【问题描述】

在进行文法分析的时候，通常需要检测一个单词是否在我们的单词列表里。为了提高查找和定位的速度，通常都画出与单词列表所对应的单词查找树，其特点如下：

1．根结点不包含字母，除根结点外每一个结点都仅包含一个大写英文字母；

2．从根结点到某一结点，路径上经过的字母依次连起来所构成的字母序列，称为该结点对应的单词。单词列表中的每个单词，都是该单词查找树某个结点所对应的单词；

3．在满足上述条件下，该单词查找树的结点数最少。

4．例如图 2 左边的单词列表就对应于右边的单词查找树。注意，对一个确定的单词列表，请统计对应的单词查找树的结点数（包含根结点）。

【问题输入】

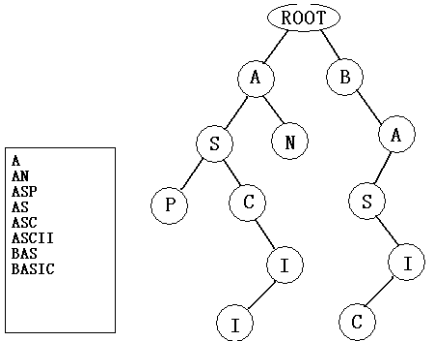
输入文件名为 word.in，该文件为一个单词列表，每一行仅包含一个单词和一个换行/回车符。每个单词仅由大写的英文字母组成，长度不超过 63 个字母 。文件总长度不超过 32K，至少有一行数据。

【问题输出】

输出文件名为 word.out，该文件中仅包含一个整数，该整数为单词列表对应的单词查找树的结点数。

【样例输入】

```
A
AN
ASP
AS
ASC
ASCII
BAS
BASIC
```



【样例输出】

13　　图 2

【算法分析】

首先要对建树的过程有一个了解。对于当前被处理的单词和当前树：在根结点的子结点中找单词的第一位字母，若存在则进而在该结点的子结点中寻找第二位……如此下去直到单词结束，即不需要在该树中添加结点；或单词的第 n 位不能被找到，即将单词的第 n 位及其后的字母依次加入单词查找树中去。但，本问题只是问你结点总数，而非建树方案，且有 32K 文件，所以应该考虑能不能通过不建树就直接算出结点数？为了说明问题的本质，我们给出一个定义：一个单词相对于另一个单词的差：设单词 1 的长度为 L，且与单词 2 从第 N 位开始不一致，则说单词 1 相对于单词 2 的差为 L-N+1，这是描述单词相似程度的量。可见，将一个单词加入单词树的时候，须加入的结点数等于该单词树中已有单词的差的最小值。

单词的字典顺序排列后的序列则具有类似的特性，即在一个字典顺序序列中，第 m 个单词相对于第 m-1 个单词的差必定是它对于前 m-1 个单词的差中最小的。于是，得出建树的等效算法：

① 读入文件；

② 对单词列表进行字典顺序排序；

③ 依次计算每个单词对前一单词的差，并把差累加起来。注意：第一个单词相对于“空”的差为该单词的长度；

④ 累加和再加上 1（根结点），输出结果。

就给定的样例，按照这个算法求结点数的过程如下表：

表 6_1

原单词列表	排序后的列表	差值	总计	输出
A	A	1	12	13
AN	AN	1		
ASP	AS	1		
AS	ASC	1		
ASC	ASCII	2		
ASCII	ASP	1		
BAS	BAS	3		
BASIC	BASIC	2		

【数据结构】

先确定 32K（32*1024=32768 字节）的文件最多有多少单词和字母。当然应该尽可能地存放较短的单词。因为单词不重复，所以长度为 1 的单词（单个字母）最多 26 个；长度为 2 的单词最多为 26*26=676 个；因为每个单词后都要一个换行符（换行符在计算机中占 2 个字节），所以总共已经占用的空间为：（1+2）*26+（2+2）*676=2782 字节；剩余字节（32768-2782=29986 字节）分配给长度为 3 的单词（长度为 3 的单词最多有 26*26*26=17576 个）有 29986/（3+2）≈5997。所以单词数量最多为 26+676+5997=6699。

定义一个数组：a：array[1..32767] of string；把所有单词连续存放起来，用选择排序或快排对单词进行排序。

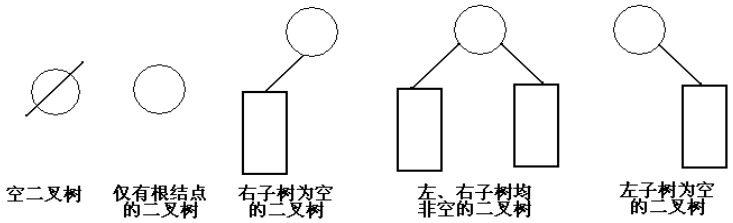
【参考程序】

```
Program p3_1;
var a:array[0..8000]of string;          //数组可以到 32768
      i,j,n,t,k:longint;
      s:string;

{procedure qsort(l,r:longint);          //快排，本程序用选择排序， 未选用快排。
var i,j:longint;mid,t:string;
 begin
  i:=l;j:=r;mid:=a[(l+r)div 2];
  repeat
   while a[i]<mid do inc(i);
   while a[j]>mid do dec(j);
    if i<=j then
      begin
        t:=a[i];a[i]:=a[j];a[j]:=t;
        inc(i);dec(j);
      end;
  until i>j;
  if i<r then qsort(i,r);
  if l<j then qsort(l,j);
 end; }

Begin
  assign(input,'word.in');reset(input);
  assign(output,'word.out');rewrite(output);
  while not eof do                  //读入文件中的单词并且存储到数组中
  begin
    inc(n);
    readln(a[n]);
  end;
  for i:=1 to n-1 do                //单词从小到大排序，选择排序可改为快排 qsort(1,n)
   for j:=i+1 to n do
    if (a[i]>a[j]) then             //两个单词进行交换
      begin
         s:=a[i];
         a[i]:=a[j];
         a[j]:=s;
      end;
  t:= length(a[1]);                 //先累加第 1 个单词的长度
  for i:=2 to n do                  //依次计算每个单词对前一单词的差
   begin
    j:=1;
    while (a[i,j]=a[i-1,j]) and (j<=length(a[i-1]))do inc(j);   //求两个单词相同部分的长度
    t:=t+length(a[i])-j+1;                                      //累加两个单词的差 length(a[i])-j+1
   end;
  writeln(t+1);                     //输出单词查找树的结点数
  close(input);close(output);
End.
```

第二节　二叉树

一、二叉树基本概念

二叉树（binary tree,简写成 BT）是一种特殊的树型结构，它的度数为 2 的树。即二叉树的每个结点最多有两个子结点。每个结点的子结点分别称为左孩子、右孩子，它的两棵子树分别称为左子树、右子树。二叉树有 5 中基本形态：



前面引入的树的术语也基本适用于二叉树，但二叉树与树也有很多不同，如：首先二叉树的每个结点至多只能有两个结点，二叉树可以为空，二叉树一定是有序的，通过它的左、右子树关系体现出来。

二、二叉树的性质

【性质 1】在二叉树的第 i 层上最多有 2^{i-1} 个结点（i>=1）。

证明：很简单，用归纳法：当 i=1 时，$2^{i-1}=1$ 显然成立；现在假设第 i-1 层时命题成立，即第 i-1 层上最多有 2^{i-2} 个结点。由于二叉树的每个结点的度最多为 2，故在第 i 层上的最大结点数为第 i-1 层的 2 倍，即 $2*2^{i-2}=2^{i-1}$。

【性质 2】深度为 k 的二叉树至多有 2^k-1 个结点（k>=1）。

证明：在具有相同深度的二叉树中，仅当每一层都含有最大结点数时，其树中结点数最多。因此利用性质 1 可得，深度为 k 的二叉树的结点数至多为：

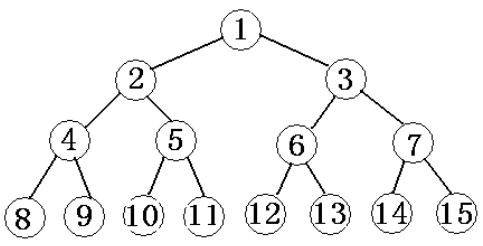
$$2^0+2^1+\cdots+2^{k-1}=2^k-1$$

故命题正确。

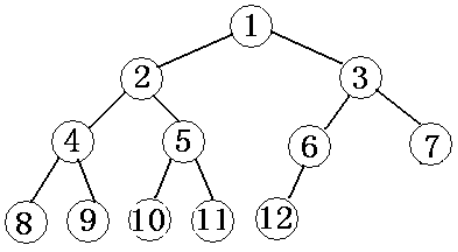
特别：一棵深度为 k 且有 2^k-1 个结点的二叉树称为满二叉树。如下图 A 为深度为 4 的满二叉树，这种树的特点是每层上的结点数都是最大结点树。

可以对满二叉树的结点进行连续编号，约定编号从根结点起，自上而下，从左到右，由此引出完全二叉树的定义，深度为 k，有 n 个结点的二叉树当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 到 n 的结点一一对应时，称为完全二叉树。

下图 B 就是一个深度为 4，结点数为 12 的完全二叉树。它有如下特征：叶结点只可能在层次最大的两层上出现；对任一结点，若其由分支下的子孙的最大层次为 m，则在其左分支下的子孙的最大层次必为 m 或 m+1。下图 C、D 不是完全二叉树，Q 请大家思考为什么？



图A



图B

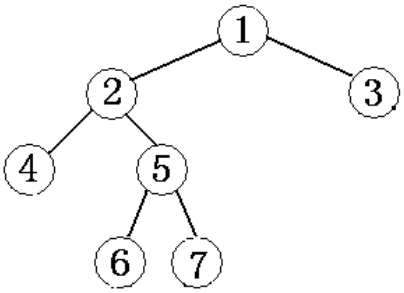


图 C

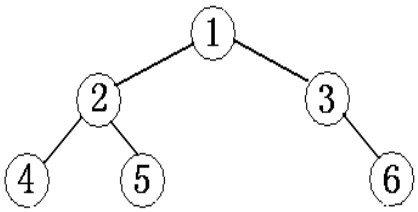


图 D

【性质 3】对任何一棵二叉树，如果其叶结点数为 n_0，度为 2 的结点数为 n_2，则一定满足：$n_0=n_2+1$。

证明：因为二叉树中所有结点的度数均不大于 2，所以结点总数(记为 n)应等于 0 度结点数 n_0、1 度结点 n_1 和 2 度结点数 n_2 之和：

$$n=n_o+n_1+n_2 \quad \cdots\cdots(式子\ 1)$$

另一方面，1 度结点有一个孩子，2 度结点有两个孩子，故二叉树中孩子结点总数是：

$$n_1+2n_2$$

树中只有根结点不是任何结点的孩子，故二叉树中的结点总数又可表示为：

$$n=n_1+2n_2+1 \quad \cdots\cdots(式子\ 2)$$

由式子 1 和式子 2 得到：

$$n_o=n_2+1$$

【性质 4】具有 n 个结点的完全二叉树的深度为 $trunc(LOG_2^n)+1$

证明：假设深度为 k，则根据完全二叉树的定义，前面 k-1 层一定是满的，所以 $n>2^{k-1}-1$。但 n 又要满足 $n<=2^k-1$。所以，$2^{k-1}-1<n<=2^k-1$。变换一下为 $2^{k-1}<=n<2^k$。

以 2 为底取对数得到：$k-1<=LOG_2^n<k$。而 k 是整数，所以 $k= trunc(LOG_2^n)+1$。

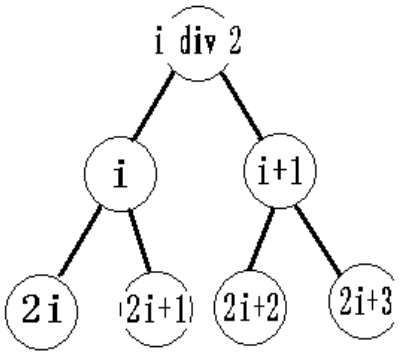
【性质 5】对于一棵 n 个结点的完全二叉树，对任一个结点（编号为 i），有：

①如果 i=1,则结点 i 为根，无父结点；如果 i>1,则其父结点编号为 trunc(i/2)。

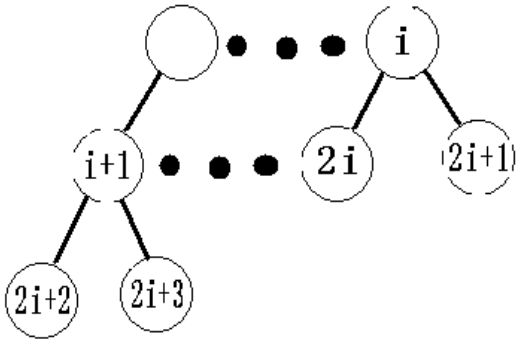
如果 2*i>n，则结点 i 无左孩子（当然也无右孩子，为什么？即结点 i 为叶结点）；否则左孩子编号为 2*i。

②如果 2*i+1>n，则结点 i 无右孩子；否则右孩子编号为 2*i+1。

证明：略，我们只要验证一下即可。总结如下图：



结点i和i+1在同一层上



结点i和i+1不在同一层上

三、二叉树的存储结构

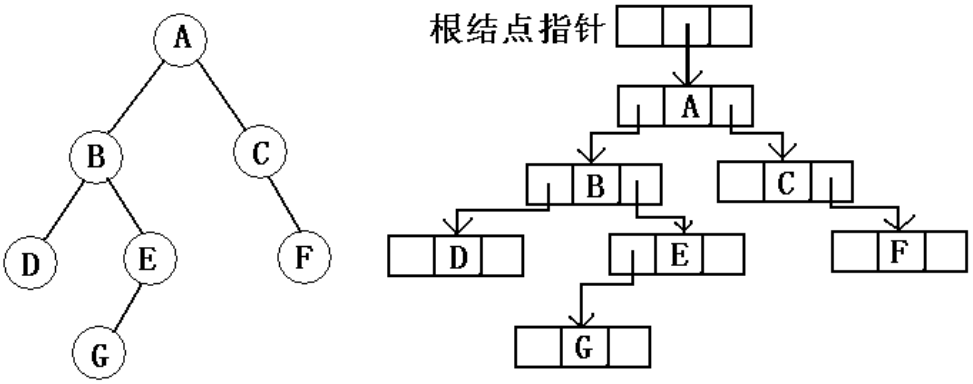
①链式存储结构，即单链表结构或双链表结构（同树）。数据结构修改如下：

```
type tree=^node;
     node=record
       data:char;
       lchild,rchild:tree
     end;
var bt:tree;
```

或：

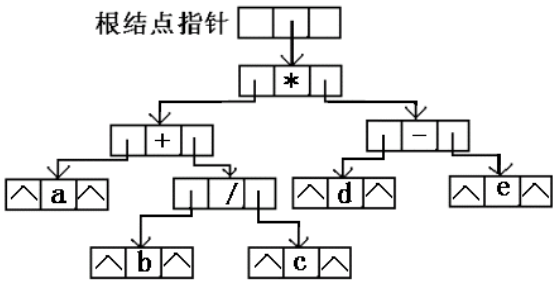
```
type tree=^node;
     node=record
       data:char;
       lchild,rchild,father:tree
     end;
var bt:tree;
```

如左图的一棵二叉树用单链表就可以表示成右边的形式：



②顺序存储结构，即几个数组加一个指针变量。数据结构修改如下：

```
const n=10;
var data:array[1..n] of char;
    lchild:array[1..n] of integer;
    rchild:array[1..n] of integer;
    bt:integer;  {根结点指针}
```



二叉树在处理表达式时经常用到，一般用叶结点表示运算元，分支结点表示运算符。这样的二插树称为表达式树。如现在有一个表达式：(a+b/c) *(d-e)。可以用以右图表示：

数据结构定义如下：

按表达式的书写顺序逐个编号，分别为 1..9，注意表达式中的所有括号在树中是不出现的，因为表达式树本身就是有序的。叶结点的左右子树均为空（用 0 表示）。

```
const n=9;
var data:array[1..n] of char=( ‘a’ , ‘+’ , ‘b’, ‘/’, ‘c’, ‘*’, ‘d’, ‘-’, ‘e’) ;
    lchild:array[1..n] of integer=(0,1,0,3,0,2,0,7,0);
    rchild:array[1..n] of integer=(0,4,0,5,0,8,0,9,0);
    bt:integer;                    //根结点指针,初值=6，指向‘*’
```

二叉树的操作：最重要的是遍历二叉树，但基础是建一棵二叉树、插入一个结点到二叉树中、删除结点或子树等。

【例 2】医院设置

【问题描述】

设有一棵二叉树（如图 3，其中圈中的数字表示结点中居民的人口，圈边上数字表示结点编号。现在要求在某个结点上建立一个医院，使所有居民所走的路程之和为最小，同时约定，相邻接点之间的距离为 1。就本图而言，若医院建在 1 处，则距离和=4+12+2*20+2*40=136；若医院建在 3 处，则距离和=4*2+13+20+40=81……

【输入格式】

输入文件名为 hospital.in，其中第一行一个整数 n，表示树的结点数（n<=100）。接下来的 n 行每行描述了一个结点的状况，包含三个整数，整数之间用空格（一个或多个）分隔，其中：第一个数为居民人口数；第二个数为左链接，为 0 表示无链接；第三个数为右链接，为 0 表示无链接。

【输出格式】

输出文件名为 hospital.out，该文件只有一个整数，表示最小距离和。

【样例输入】

```
5
13 2 3
4   0 0
12 4 5
20 0 0
40 0 0
```

【样例输出】

```
81
```

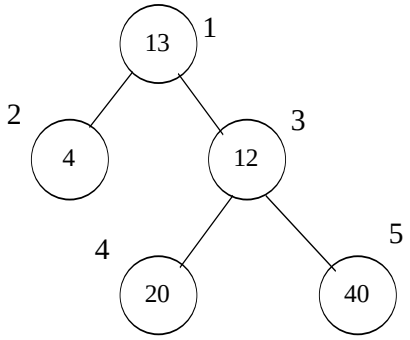


图 3

【算法分析】

这是一道简单的二叉树应用问题，问题中的结点数并不多，数据规模也不大，采用邻接矩阵存储，用 Floyed 法求出任意两结点之间的最短路径长，然后穷举医院可能建立的 n 个结点位置，找出一个最小距离的位置即可。当然也可以用双链表结构或带父结点信息的数组存储结构来解决，但实际操作稍微麻烦了一点。

【参考程序】

```
Program p3_2;
Var a : Array [1..100] Of Longint;
    g : Array [1..100, 1..100] Of Longint;
    n, i, j, k, l, r, min, total : Longint;
```

```
Begin
  Assign(Input, 'hospital.in');  Reset(Input);
  Assign(Output, 'hospital.in');  Rewrite(Output);
  Readln(n);
  For i := 1 To n Do
    For j := 1 To n Do
      g[i][j] := 1000000;
  For i := 1 To n Do              //读入、初始化
    Begin
      g[i][i] := 0;
      Readln(a[i], l, r);
      If l > 0 Then Begin g[i][l] := 1; g[l][i] := 1 End;
      If r > 0 Then Begin g[i][r] := 1; g[r][i] := 1 End;
    End;
  For k := 1 To n Do              //用Floyed法求任意两结点之间的最短路径长
    For i := 1 To n Do
      If i <> k Then
        For j := 1 To n Do
          If (i <> j) And (k <> j) And (g[i][k] + g[k][j] < g[i][j])
            Then g[i][j] := g[i][k] + g[k][j];
  min := Maxlongint;
  For i := 1 To n Do              //穷举医院建在N个结点，找出最短距离
    Begin
      total := 0;
      For j := 1 To n Do
        Inc(total, g[i][j] * a[j]);
      If total < min Then min := total;
    End;
  Writeln(min);
  Close(Input); Close(Output);
End.
```

【后记】

在各种竞赛中经常遇到这样的问题：N-1 条公路连接着 N 个城市，从每个城市出发都可以通过公路到达其它任意的城市。每个城市里面都有一定数量的居民，但是数量并不一定相等，每条公路的长度也不一定相等。X 公司（或者是政府）决定在某一个城市建立一个医院/酒厂/游乐场……，问：将它建在哪里，可以使得所有的居民移动到那里的总耗费最小？这种题目都是本题的“变型”，一般称为“树的中心点问题”。除了简单的穷举法外，还有更好的时间复杂度为 O(n)的算法，我们讲在后面的章节中继续讨论。

四、遍历二叉树

在二叉树的应用中，常常要求在树中查找具有某种特征的结点，或者对全部结点逐一进行某种处理。这就是二叉树的遍历问题。所谓二叉树的遍历是指按一定的规律和次序访问树中的各个结点，而且每个结点仅被访问一次。“访问”的含义很广，可以是对结点作各种处理，如输出结点的信息等。遍历一般按照从左到右的顺序，共有 3 种遍历方法，先（根）序遍历，中（根）序遍历，后（根）序遍历。

㈠先序遍历的操作定义如下：

若二叉树为空，则空操作，否则

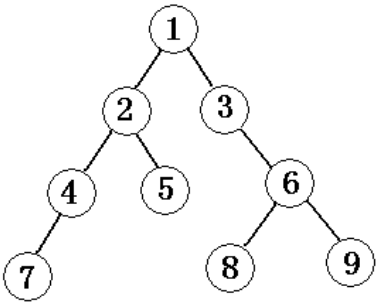
①访问根结点

②先序遍历左子树

③先序遍历右子树

先序遍历右图结果为：124753689

```
procedure preorder(bt:tree);      //先序遍历根结点为 bt 的二叉树的递归算法
    begin
      if bt<>nil then begin
                        write(bt^.data);
                        preorder(bt^.lchild);
                        preorder(bt^.rchild);
                      end;
    end;
```



㈡中序遍历的操作定义如下：

若二叉树为空，则空操作，否则

①中序遍历左子树

②访问根结点

③中序遍历右子树

中序遍历右图结果为：742513869

```
procedure inorder(bt:tree);      //中序遍历根结点为 bt 的二叉树的递归算法
    begin
      if bt<>nil then begin
                        preorder(bt^.lchild);
                        write(bt^.data);
                        preorder(bt^.rchild);
                      end;
    end;
```

㈢后序遍历的操作定义如下：

若二叉树为空，则空操作，否则

①后序遍历左子树

②后序遍历右子树

③访问根结点

后序遍历右图结果为：745289631

```
procedure postorder(bt:tree);      //后序遍历根结点为 bt 的二叉树的递归算法
  begin
    if bt<>nil then begin
                      preorder(bt^.lchild);
                      preorder(bt^.rchild);
                      write(bt^.data);
                    end;
  end;
```

当然，我们还可以把递归过程改成用栈实现的非递归过程，以先序遍历为例，其它的留给作者完成。

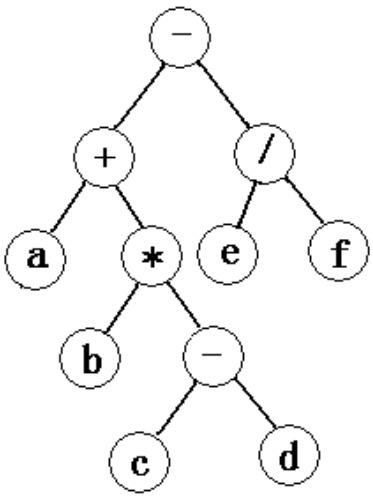
```
Procedure preorder(bt:tree);          //先序遍历 bt 所指的二叉树
 Var   stack:array[1..n] of tree;     //栈
       top:integer;                   //栈顶指针
       P:tree;
 begin
    top:=0;
    While not ((bt=nil)and(top=0)) do
    Begin
      While bt<>nil do begin                  //非叶结点
                         Write(bt^.data);   //访问根
                         top:=top+1;        //右子树压栈
                         stack[top]:=bt^.rchild
                         bt:=bt^.lchild;    //遍历左子树
                       end;
      If top<>0 then begin                    //栈中所有元素出栈，遍历完毕
                       bt:=stack[top];
                       top:=top-1;
                     End;
    End;
  End;
```

关于前面讲的表达式树，我们可以分别用先序、中序、后序的遍历方法得出完全不同的遍历结果，如对于右图 3 种遍历结果如下，它们正好对应着表达式的 3 种表示方法。

-+a*b-cd/ef (前缀表示、波兰式)

a+b*c-d-e/f (中缀表示)

abcd-*+ef/- (后缀表示、逆波兰式)



表达式（a+b*(c-d)-e/f）的二叉树

五、二叉树的其它重要操作

除了“遍历”以外，二叉树的其它重要操作还有：建立一棵二叉树、插入一个结点到二叉树中、删除结点或子树等。

1、建立一棵二叉树

```
Procedure pre_crt(var bt:tree);          //按先序次序输入二叉树中结点的值，生成
  begin          //二叉树的单链表存储结构，bt 为指向根结点的指针,’’表示空树
      Read(ch);
      If ch=’’ then bt:=nil
      Else begin
            New(bt);                    //建根结点
            Bt^.data:=ch;
            Pre_crt(bt^.lchild);        //建左子树
            Pre_crt(bt^.rchild);        //建右子树
          End;
    End;
```

2、删除二叉树

```
Procedure dis(var bt:tree);                 //删除二叉树
    Begin
      If bt<>nil then begin
                        Dis(bt^.lchild);  //删左子树
                        Dis(bt^.rchild);  //删右子树
                        Dispose(bt);      //释放父结点
                      End;
    End;
```

3．插入一个结点到二叉树中

```
Procedure insert(var bt:tree;n:integer);     //插入一个结点到二叉树中
    Begin
      If bt=nil then begin
                       new(bt);
                       bt^.data:=n;
                       bt^.lchild:=nil;
                       bt^.rchild:=nil;
                     End
       Else  if n<bt^.data then insert(bt^.lchild,n)
                         else if n>bt^.data then insert(bt^.rchild,n);
    End;
```

4．在二叉树中查找一个数，找到返回该结点,否则返回 nil

```
Function find(bt:tree;n:integer):tree;  //在二叉树中查找一个数，找到返回该结点,否则返回 nil。
  Begin
```

```
        if bt=nil then find:=nil
                else if n<bt^.data then find(bt^.lchild,n)
                                else if n>bt^.data then find(bt^.rchild,n)
                                                else find:=bt;
    end;
```

5．用嵌套括号表示法输出二叉树

```
    Procedure print(bt:tree);                    //用嵌套括号表示法输出二叉树
      Begin
        If bt<>nil   then   begin
                              Write(bt^.data);
                              If (bt^.lchild<>nil)or(bt^.rchild<>nil)   then
                                  Begin
                                      Write( '(' );
                                      Print(bt^.lchild);
                                      If bt^.rchild<>nil then write( ',' );
                                      Print(bt^.rchild);
                                      Write( ')' );
                                   End;
                            End;
      End;
```

下面我们换个角度考虑这个问题，从二叉树的遍历已经知道，任意一棵二叉树结点的前序序列和中序序列是唯一的。反过来，给定结点的前序序列和中序序列，能否确定一棵二叉树呢？又是否唯一呢？

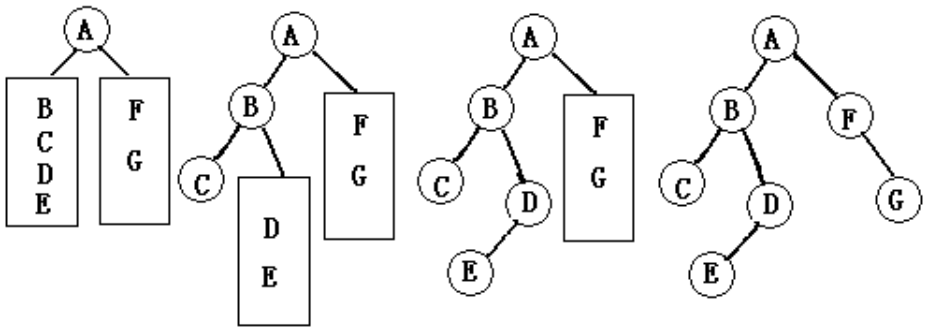
由定义，二叉树的前序遍历是先访问根结点，再遍历左子树，最后遍历右子树。即在结点的前序序列中，第一个结点必是根，假设为 root。再结合中序遍历，因为中序遍历是先遍历左子树，再访问根，最后遍历右子树。所以结点 root 正好把中序序列分成了两部分，root 之前的应该是左子树上的结点，root 之后的应该是右子树上的结点。依次类推，便可递归得到整棵二叉树。

结论：已知前序序列和中序序列可以确定出二叉树；

已知中序序列和后序序列也可以确定出二叉树；

但，已知前序序列和后序序列却不可以确定出二叉树；为什么？举个 3 个结点的反例。

例如：已知结点的前序序列为 ABCDEFG，中序序列为 CBEDAFG。构造出二叉树。过程见下图：



【例 3】二叉树的遍历问题

【问题描述】

输入一棵二叉树的先序和中序遍历序列，输出其后序遍历序列。

【输入格式】

输入文件为 tree.in，共两行，第一行一个字符串，表示树的先序遍历，第二行一个字符串，表示树的中序遍历。树的结点一律用小写字母表示。

【输出格式】

输出文件为 tree.out，仅一行，表示树的后序遍历序列。

【样例输入】

abdec

dbeac

【样例输出】

debca

【参考程序】

```
Program p3_3;
Var s1, s2 : String;
Procedure try(l1, r1, l2, r2 : Integer);                    //递归、后序
  Var m : Integer;
  Begin
    m := pos(s1[l1], s2);                                   //求l1的第一个字符在l2中的位置
    If m>l2 Then try(l1 + 1, l1 + m - l2, l2, m - 1);       //遍历第m个的左半边
    If m<r2 Then try(l1 + m - l2 + 1, r1, m + 1, r2);       //遍历第m个的右半边
    Write(s1[l1])
  End;
Begin                                                       //main
    Assign(Input，’tree.in’); Reset(Input);
    Assign(Onput，’tree.out’); Rewrite(Onput);
    Readln(s1); Readln(s2);
    try(1, Length(s1), 1, Length(s2));
    Writeln
    Close(Input); Close(Output);
End.
```

六、普通树转换成二叉树

由于二叉树是有序的，而且操作和应用更广泛，所以在实际使用时，我们经常把普通树转换成二叉树进行操作。如何转换呢？一般方法如下：

1．将树中每个结点除了最左边的一个分支保留外，其余分支都去掉；

2．从最左边结点开始画一条线，把同一层上的兄弟结点都连起来；

3．以整棵树的根结点为轴心，将整棵树顺时针大致旋转 45 度。

第一节中的图 A 所示的普通树转换成二叉树的过程如图 B 所示：

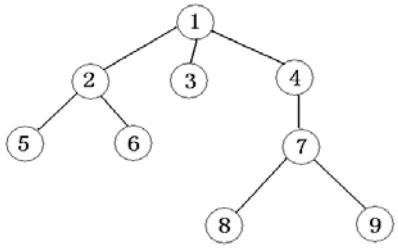


图 A

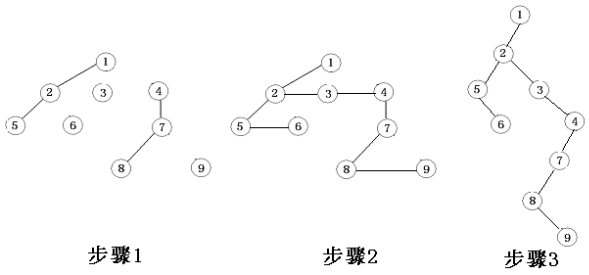


图 B

同样我们可以把森林也转换成二叉树处理，假设 F={T1，T2，…，Tm}是森林，则可按如下规则转换成一棵二叉树 B=(root,lb,rb)。

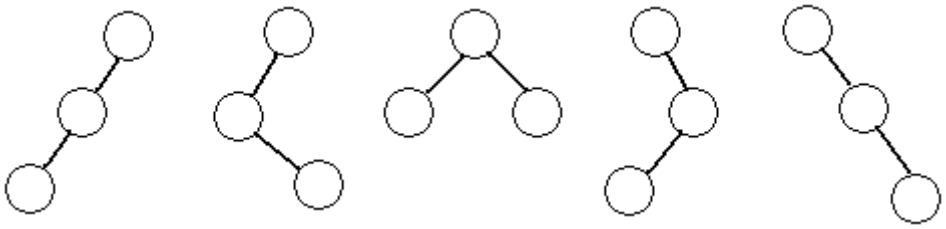
1．若 F 为空，即 m=0，则 B 为空树；

2．若 F 非空，即 m<>0，则 B 的根 root 即为森林中第一棵树的根 root(T1)；B 的左子树 lb 是从 T1 中根结点的子树森林 F1={T11，T12，…，T1m1}转换而成的二叉树；其右子树 rb 是从森林 F’={T2，T3，…，Tm}转换而成的二叉树。

七、树的计数问题

具有 n 个结点的不同形态的二叉树有多少棵？具有 n 个结点的不同形态的树有多少棵？首先了解两个概念，“相似二叉树”是指两者都为空树或者两者均不为空树，且它们的左右子树分别相似。“等价二叉树”是指两者不仅相似，而且所有对应结点上的数据元素均相同。二叉树的计数问题就是讨论具有 n 个结点、互不相似的二叉树的数目 Bn。

在 n 很小时，很容易得出，B0=1，B1=1，B2=2，B3=5（见下图）。



一般情况，一棵具有 n(n>1)个结点的二叉树可以看成是由一个根结点、一棵具有 i 个结点的左子树、和一棵具有 n-i-1 个结点的右子树组成，其中 0<=i<=n-1，

由此不难得出下列递归公式：

$$B0 = 1$$

$$Bn = \sum_{i=0}^{n-1} B_i B_{n-i-1} \quad (n>=1)$$

我们可以利用生成函数讨论这个递归公式，得出：$Bn=C^n_{2n}/(n+1)$。

由于树可以转换成二叉树，所以，可以推出：$Tn=B_{n-1}$。

【课堂练习】

1、一棵完全二叉树的结点总数为 18，其叶结点数为_____。

A.7 个　　B.8 个　　C.9 个　　D.10 个

2、 二叉树第 10 层的结点数的最大数目为_____。

A.10　　B.100　　C.512　　D.1024

3、一棵深度为 K 的满二叉树有（　　）个结点。

A.2^K-1　　B.2^K　　C.2*K　　D.2*K-1

4、对任何一棵二叉树 T，设 n_0、n_1、n_2 分别是度数为 0、1、2 的顶点数，则下列判断中正确的是_____。

A.$n_0=n_2+1$　　B.$n_1=n_0+1$　　C. $n_2=n_0+1$　　D.$n_2=n_0+1$

5、一棵 n 个节点的完全二叉树，则该二叉树的高度 h 为(　)。

A.n/2　　B.log(n)　　C.log(n)/2　　D. $\lfloor \log(n) \rfloor + 1$

6、一棵完全二叉树上有 1001 个结点，其中叶子结点的个数是_____。

A.250　　B.500　　C.254　　D.501

7、如果一棵二叉树有 N 个度为 2 的节点，M 个度为 1 的节点，则该树的叶子个数为_____。

A. N+1　　B. 2 * N-1　　C.N-1　　D. M+N-1

8、一棵非空二叉树的先序遍历序列和后序遍历序列正好相反，则该二叉树一定满足_____。

A.所有结点均无左孩子　　B.所有的结点均无右孩子

C.只有一个叶子结点　　D.是任意一棵二叉树

9、将有关二叉树的概念推广到三叉树，则一棵有 244 个结点的完全三叉树的高度是_____。

A.4　　B.5　　C.6　　D.7

10、在一棵具有 K 层的满三叉树中，结点总数为_____。

A.$(3^k-1)/2$　　B.3^k-1　　C.$(3^k-1)/3$　　D.3^k

11、设树 T 的高度为 4，其中度为 1、2、3 和 4 的结点个数分别为 4、2、1、1，则 T 中的叶子数为_____。

A.5　　B.6　　C.7　　D.8

12、一棵树 T 有 2 个度数为 2 的结点、有 1 个度数为 3 的结点、有 3 个度数为 4 的结点，那么树 T 有(　　)个树叶。

A.14　　B.6　　C.18　　D.7

13、某二叉树中序序列为 abcdefg，后序序列为 bdcafge，则前序序列是_____。

A.egfacbd　　B.eacbdgf　　C.eagcfbd　　D.以上答案都不对

14、已知某二叉树的中序遍历序列是 debac，后序遍历序列是 dabec，则它的前序遍历序列是_____。

A.a c b e d　　B.d e c a b　　C.d e a b c　　D.c e d b a

15、一颗二叉树的中序遍历序列为 DGBAECHF，后序遍历序列为 GDBEHFCA，则前序遍历序列是_____。

A. ABCDFGHE　　B. ABDGCEFH　　C. ACBGDHEF　　D. ACEFHBGD

16、已知一棵二叉树的前序序列为 ABDEGCFH，中序序列为 DBGEACHF，则该二叉树的层次序列为___。

A.GEDHFBCA　　B.DGEBHFCA　　C.ABCDEFGH　　D.ACBFEDHG

17、已知一棵二叉树的前序遍历结果为 ABDECFHJIG，中序遍历的结果为 DBEAJHFICG，则这棵二叉树的深度为_____。

A.3　　B.4　　C.5　　D.6

18、二叉树的先序遍历和中序遍历如下：先序遍历:EFHIGJK，中序遍历:HFIEJKG。

该二叉树根的右子树的根是_____。

A. E　　B. F　　C. G　　D.H

19、中缀表达式 A-（B+C/D）*E 的后缀形式是_____。

A.AB-C+D/E*　　B.ABC+D/-E*　　C.ABCD/E*+-　　D.ABCD/+E*-

20、设森林 F 对应的二叉树为 B，它有 M 个结点，B 的根为 P，P 的右子树结点个数为 N，森林 F 中第一棵树的结点个数是_____。

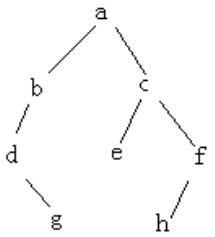
A.M-N　　B.M-N-1　　C.N+1　　D.条件不充分，无法确定

【练习答案】

1.C 2.C 3.A 4.A 5.D 6.D 7.A 8.A 9.B 10.A
11.D 12.A 13.B 14.D 15.B 16.C 17.C 18.C 19.D 20.A

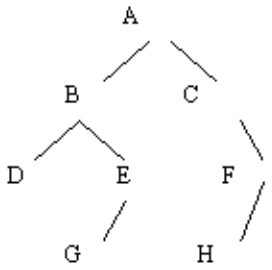
7、此树的边数为 2N+M，故而总共有 2N+M+1 个顶点。除去度为 1、2 的顶点，度为 0 的节点（即叶子节点）有(2N+M+1)-(N+M)=N+1。答案：A

12、设树 T 有 n 个结点，m 条边。边数为结点的度数之和，即 m=2×2+1×3+3×4=19，n=m+1=20。n 个结点中有 1+2+3=6 个分支结点，有叶结点 20-6=14 个。答案：A



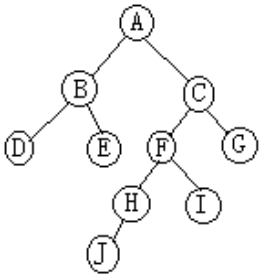
15、中序遍历 DGBAECHF 和后序遍历 GDBEHFCA 唯一对应一棵二叉树前序遍历为 ABDGCEFH。答案：B

16、由前序序列和中序序列可以唯一地确定一棵二叉树，根据题设中的前序序列和中序序列确定的二叉树为：



由此可见，其层次序列为 ABCDEFGH。答案：C

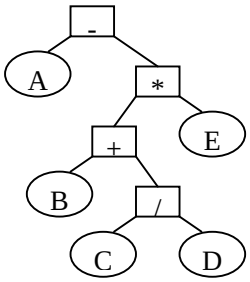
17、由题目给出二叉树先序遍历结点的顺序：ABDECFHJIG 可知结点 A 为二叉树的根结点。再由中序遍历的结点顺序：DBEAJHFICG 可知结点 A 前的 DBE 为根结点 A 的左子树的结点，而 JHFICG 为根结点 A 的右子树的结点。



先来看 A 结点的左子树的结点 DBE。在先序遍历顺序中，B 结点在 A 结点之后，说明 B 结点是左子树的根结点，D 结点又在 B 结点之后，则 D 是 B 的左子树的根结点。结点 E 由中序遍历的顺序可知是 B 的右子树的根结点。同样求出右子树 JHFICG，如下图。

由图可知，二叉树的深度为 5。答案：C。

19、该题答案是（D），本题主要考察学生怎样将中缀表达转换为后缀表达式。可以先画出该二叉树：



对其进行后根遍历即为答案。

第三节　二叉排序树

我们都已经很熟悉各种排序的方法了，现在我们可以利用二叉树的有序性进行快速排序和插入、查找。这样的树就称为二叉排序（查找）树。它有这样的性质，任何结点的值都大于它左子树上结点的值，小于右子树上结点的值，然后采用中序遍历就正好生成一个有序序列。

如下面的排序二叉树按中序遍历结果为：5 6 8 9 10 11 13 14 15 17

如何生成这样的一棵二叉树呢?

【例 4】编程输入一组的整数（约定大于等于 0，输入负数表示结束），用二叉排序树排序后按从小到大输出。

【输入样例】treesort.in

10 8 6 13 9 5 11 15 14 17 -1

【输出样例】treesort.out

5 6 8 9 10 11 13 14 15 17

【算法分析】

先生成一个结点，再根据大小决定这个结点是插在左子树还是右子树上，如此重复直到输入一个负数。

【参考程序】

```
type tree=^node;
     node=record
        data:integer;
        lchild,rchild:tree;
     end;
var
      bt:tree;
      n:integer;
```

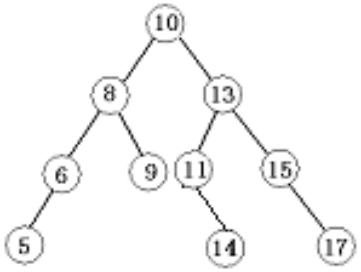


图 4

```
procedure creat_order_tree(var btx:tree;nx:integer);  //插入一个数到一棵排序二叉树中去
  var p,s,f:tree;
      flag:boolean;     //用来标识要插入的数是否在树中出现过，防止同一个数重复出现在树中
  begin
    new(s);
    s^.data:=nx;
    s^.lchild:=nil;
    s^.rchild:=nil;                  //以上 4 个语句为新建一个结点 s
    flag:=true;                      //假设没出现过
    p:=btx;                          //P 指向根结点
    while (p<>nil)and flag do        //为结点 S 找插入位置、同时判断是否出现过
      begin
        f:=p;
        if s^.data=p^.data then flag:=false;          //出现过做标记
```

```
        if s^.data<p^.data then p:=p^.lchild;          //沿左子树方向找
        if s^.data>p^.data then p:=p^.rchild;          //沿由子树方向找
      end;
    if flag then begin   //没出现过且 p=nil 说明已找到叶结点了，那么插入到叶结点的左右孩子中
                  if btx=nil then btx:=s;                          //作为根结点
                    if s^.data<f^.data then f^.lchild:=s;    //插入左孩子
                    if s^.data>f^.data then f^.rchild:=s;    //右孩子
                  end;
  end;
procedure inorder_print(btx:tree);          //递归中序输出
  begin
    if btx<>nil then
        begin
          inorder_print(btx^.lchild);    //从小到大输出，如果要求从大到小
          write(btx^.data,'  ');          //只要先右再左就可以了
          inorder_print(btx^.rchild);
        end;
  end;
BEGIN                                     //主程序
  bt:=nil;                                //根结点初始化，不指向任何结点
  writeln('input data(if <0 then over!):');
  repeat                                  //不断输入正数，不断插入
    read(n);
    if n>=0 then creat_order_tree(bt,n);
  until n<0;
  write('output sorted data:');
  inorder_print(bt);
  writeln;readln;
END.
```

注：procedure creat_order_tree 也可改成递归过程，如下：

```
procedure insert(var btx:tree;nx:integer);
begin
    new(s); s^.data:=nx; s^.lchild:=nil; s^.rchild:=nil;
    if btx=nil then btx:=s;                                  //作为根结点
          else if s^.data<btx^.data
                  then insert(btx^.lchild,nx);              //插到左子树}
                   else if s^.data>btx^.data then insert(btx^.rchild,nx);
                                          //插到右子树；再否则，即相等则什么也不做，跳过
end;
```

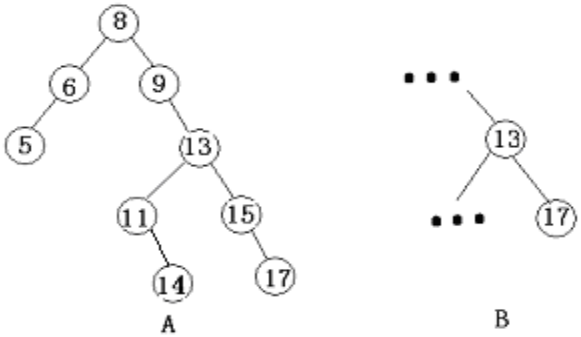
注意：二叉排序数的删除与一般意义上的树及普通二叉树有所不同，因为它要保证所有结点按中序遍历后按序输出。所以，删除二叉树 bt 中一个数据域为 x 的结点的过程如下：

① 首先查找到数据域为 X 的结点 P；

② 若结点 P 没有左子树，则用右子树的根代替被删除的结点；

③ 若结点 P 有左子树，则在其左子树中找到最右结点 R，将 P 的右子树置为 R 的右子树，再将 P 的左子树的根结点代替被删除的结点 P。

如图 4 中，删除数据域为 10 的结点变成下图左边的 A 图，删除数据域为 15 的结点变成下图右边的 B 图。



删除操作的过程如下：

```
procedure delnodex(var bt:tree;x:integer);
var p,q,r,t:tree;
begin
  p:=bt;  q:=nil;
  while (p<>nil)and(q^.data<>x) do
      if (x<p^.data) then   begin   q:=p;p:=p^.lchild; end
                    else   begin   q:=p;p:=p^.rchild; end;
  if (p=nil) then writeln('not found')
       else if (p^.lchild=nil)
            then if (q=nil) then t:=p^.rchild
                       else if (q^.lchild=p) then q^.lchild:=p^.rchild
                                             else q^.rchild:=p^.rchild
            else begin
                    r:=p^.lchild;
                    while (p^.rchild<>nil) do r:=r^.rchild;
                    r^.rchild:=p^.rchild;
                    if (q=nil) then t:=p^.lchild
                          else if (q^.lchild=p) then q^.lchild:=p^.lchild
                                                else q^.rchild:=p^.lchild;
                 end;
end;
```

第四节　哈夫曼树

一、问题的引入

将树结构用于实际，常常要考虑一个问题，即如何设计一棵二叉树，使得执行路径最短，即算法的效率最高。

例如：现有一批球磨机上的铁球，需要将它分成四类：直径不大于 20 的属于第一类;直径大于 20 而不大于 50 的属于第二类;直径大于 50 而不大于 100 的属于第三类;其余的属于第四类;假定这批球中属于第一、二、三、四类铁球的个数之比例是 1:2:3:4。

我们可以把这个判断过程表示为下图中的两种方法:

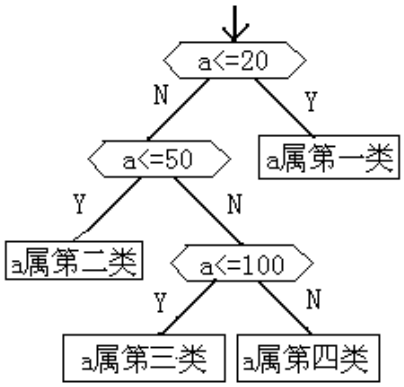


图1

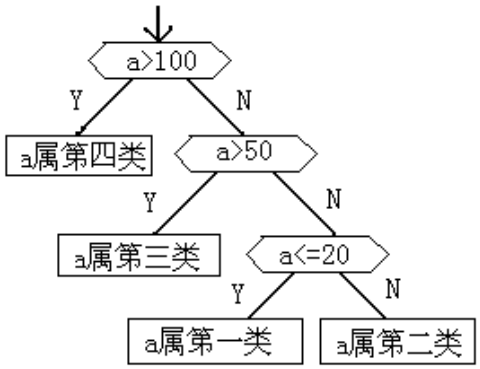


图2

那么究竟将这个判断过程表示成哪一个判断框，才能使其执行时间最短呢？让我们对上述判断框做一具体的分析。

假设有 1000 个铁球，则各类铁球的个数分别为：100、200、300、400；

对于图 1 和 图 2 比较的次数分别如下表所示:

图 1		
序号	比较式	比较次数
1	a<=20	1000
2	a<=50	900
3	a<=100	700
合计		2600

图 2		
序号	比较式	比较次数
1	a>100	1000
2	a>50	600
3	a<=20	300
合计		1900

过上述分析可知，图 2 所示的判断框的比较次数远远小于图 1 所示的判断框的比较次数。为了找出比较次数最少的判断框，将涉及到树的路径长度问题。

二、哈夫曼树的基本术语

1. 树的路径和路径长度

若在一棵树中存在着结点序列 $k_1,k_2,\ldots,k_j$,使得 k_i 是 k_{i+1} 的双亲(1<=i<j)，则称此结点序列是从 k_1 到 k_j 的**路径**，因树中每个结点只有一个双亲结点，所以它也是这两个结点之间的唯一路径。从 k_1 到 k_j 所经过的分支数称为这两点之间的**路径长度**，它等于路径上的结点数减 1。

2.**结点的权和带权路径长度**

在许多应用中，常常将树中的结点赋上一个有着某种意义的实数，我们称此实数为该结点的**权**。**结点的带权路径长度**规定为从根结点到该结点之间的路径长度与该结点上权的乘积。

3.**树的带权路径长度**

树的带权路径长度定义为树中所有叶子结点的带权路径长度之和，通常记为:

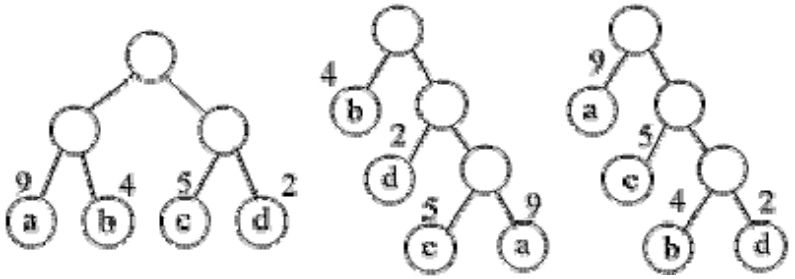
$$WPL=\sum_{i=1}^{n} w_i l_i$$

其中 n 表示叶子结点的数目，w_i 和 l_i 分别表示叶子结点 k_i 的权值和根到 k_i 之间的路径长度。

三、哈夫曼树

哈夫曼(Huffman)树又称最优二叉树。它是 n 个带权叶子结点构成的二叉树中，带权路径长度 WPL 最小的二叉树。因为构造这种树的算法是最早由哈夫曼于 1952 年提出的，所以被称之为哈夫曼树。

例如，由四个叶子结点 a,b,c,d,分别带权为 9、4、5、2，由它们构成的三棵不同的二叉树如下图 3 所示。



(a) (b) (c)

图 3 由四个叶子结点构成的三棵不同的带权二叉树

每一棵二叉树的带权路径长度 WPL 分别为:

(a)WPL=9×2+4×2+5×2+2×2=40

(b)WPL=4×1+2×2+5×3+9×3=50

(c)WPL=9×1+5×2+4×3+2×3=37

其中(c)树的 WPL 最小，此树就是哈夫曼树。

由上面可以看出，由 n 个带权叶子结点所构成的二叉树中，满二叉树或完全二叉树不一定就是最优二叉树，权值越大的结点离根越近的二叉树才是最优二叉树。

在给定一组具有确定权值的叶结点，可以构造出不同的带权二叉树。例如，给出 4 个叶结点，设其权值分别为 1，3，5，7，我们可以构造出形状不同的多个二叉树。这些形状不同的二叉树的带权路径长度将各不相同。图 4 给出了其中 5 个不同形状的二叉树。

这五棵树的带权路径长度分别为:

(a) WPL＝1×2＋3×2＋5×2＋7×2＝32

(b) WPL＝1×3＋3×3＋5×2＋7×1＝29

(c) WPL＝1×2＋3×3＋5×3＋7×1＝33

(d) WPL＝7×3＋5×3＋3×2＋1×1＝43

(e) WPL＝7×1＋5×2＋3×3＋1×3＝29

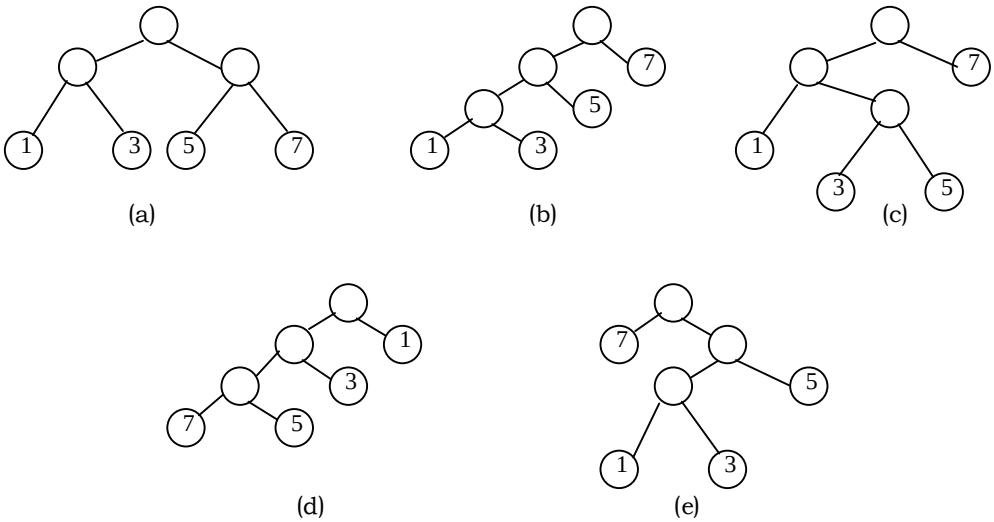


图 4　具有相同叶子结点和不同带权路径长度的二叉树

由此可见，由相同权值的一组叶子结点所构成的二叉树有不同的形态和不同的带权路径长度，那么如何找到带权路径长度最小的二叉树（即哈夫曼树）呢？根据哈夫曼树的定义，一棵二叉树要使其 WPL 值最小，必须使权值越大的叶结点越靠近根结点，而权值越小的叶结点越远离根结点。哈夫曼树的最主要的特点是带权的二叉树的路径长度最小的是权大的叶子离根最近的二叉树，因此哈夫曼树又称为最优叶子二叉树。

注意：

①叶子上的权值均相同时，完全二叉树一定是最优二叉树，否则完全二叉树不一定是最优二叉树。

②最优二叉树中，权越大的叶子离根越近。

③最优二叉树的形态不唯一。

如何构造哈夫曼二叉树呢？D · A · Huffman 给出了一个简单而又漂亮的算法，这个算法称为哈夫曼算法，它的基本思想就是让权大的叶子离根最近，具体做法是：

（1）根据给定的 n 个权值 $\{w_1,w_2,\cdots,w_n\}$，构造 n 棵二叉树的集合 F＝ $\{T_1,T_2,\cdots,T_n\}$，其中每棵二叉树中均只含一个带权值为 w_i 的根结点，其左、右子树为空树；

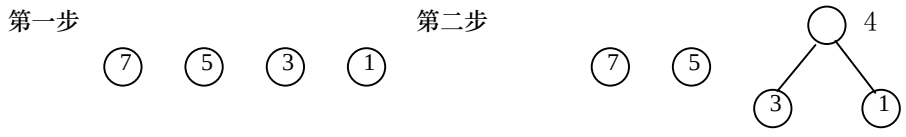
（2）在 F 中选取其根结点的权值为最小的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和；

（3）从 F 中删去这两棵树，同时加入刚生成的新树；

（4）重复（2）和（3）两步，直到 F 中只含一棵树为止。

从上述算法中可以看出，F 实际上是森林，算法的目的是不断地对森林中的二叉树进行“合并”，最终得到哈夫曼二叉树。

图 5 给出了前面提到的叶结点权值集合为 W＝{1，3，5，7}的哈夫曼树的构造过程。可以计算出其带权路径长度为 29，由此可见，对于同一组给定叶结点所构造的哈夫曼树，树的形状可能不同，但带权路径长度值是相同的，一定是最小的。



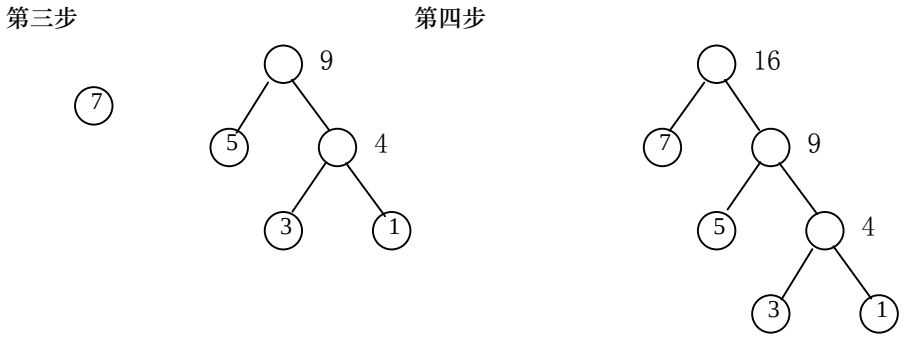


图 5　哈夫曼树的建立过程

四、哈夫曼树的构造算法

从上述算法中可以看出，F 实际上是森林，该算法的思想是不断地进行森林 F 中的二叉树的“合并”，最终得到哈夫曼树。

实际上，哈夫曼算法的实现与实际问题所采用的存储结构有关。现假设用数组 F 来存储哈夫曼树，其中第 i 个数组元素 F[i]是哈夫曼树中的一个结点，其地址为 i，有 3 个域，Data 域存放该结点的权值，lChild 域和 rChild 域分别存放该结点左、右子树的根结点的地址（下标）。

data	lchild	rchild

在初始状态下：F[i].Data=W_i，F[i].lChild=F[i].rChild=0，i=1,2,…,n。即先构造了 n 个叶子。在以后每步构造一棵新二叉树时，都需要对森林中所有二叉树的根结点进行排序，因此可用数组 a 作为排序暂存空间，其中第 i 个数组元素 a[i]是森林 F 中第 i 棵二叉树的根结点，有 2 个域，Data 是根结点所对应的权值，Addr 是根结点在 F 中的地址（下标）。

在初始状态下：a[i].Data=W_i，a[i].Addr=i，i=1,2,…n。

下面给出建立哈夫曼二叉树的过程：

```
PROCEDURE CreateHfm(F,t,a,n)      //已知 n 个权值 a[i]，构造哈夫曼树 F，且根结点地址(下标)为 t
VAR i:Integer;
BEGIN
    FOR i:=1 TO n DO BEGIN        //初始化
        F[i].Data:=a[i].Data;
        F[i].lChild:=0;
        F[i].rChild:=0;
        a[i].Addr:=i
    END;
    t:=n+1;                       //t 指向下一个可利用单元
    i:=n;                         //当前森林中的二叉树是 i
    WHILE i>=2 DO   BEGIN
        Insert(a,i);              //对 a 的前 i 个元素按 Data 域进行排序
        F[t].Data:=a[1].Data+a[2].Data;    //生成新的二叉树
        F[t].lChild:=a[1].Addr;
```

```
            F[t].rChild:=a[2].Addr;
            a[1].Data:=F[t].Data;          //修改森林
            a[1].Addr:=t;
            a[2].Data:=a[i].Data;          //修改森林
            a[2].Addr:=a[i].Addr;
            i:=i-1;                        //二叉树数目减一
            t:=t+1;
        END;
END;
```

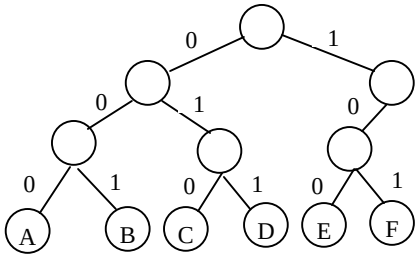
该算法是使用数组下标来建立二叉链表的，很容易将该段程序代码改为使用指针链表来建立二叉链表的形式，留给大家完成。

五、哈夫曼树的应用

哈夫曼树的应用很广，哈夫曼编码就是其中的一种，下面给予简要介绍。

在电报通讯中，电文是以二进制的 0、1 序列传送的。在发送端需要将电文中的字符序列转换成二进制的 0、1 序列（即编码），在接收端又需要把接收到的 0、1 序列转换成对应的字符序列（即译码）。

最简单的二进制编码方式是等长编码，例如，假定电文中只使用 A、B、C、D、E、F 这六种字符，若进行等长编码，它们分别需要三位二进制字符，可以依次编码为 000、001、010、011、100、101。若用这六个字符作为六个叶子结点，来生成一棵二叉树，让该二叉树中每个分支结点的左、右分支分别用 0 和 1 编码，从树根结点到每个叶子结点的路径上所经过的分支的 0、1 编码序列应等于该叶子结点的二进制编码，则对应的编码二叉树如下图所示。



由常识可知，电文中每个字符出现的频率一般是不同的。假定在一份电文中，这六个字符出现的频率依次为 4、2、6、8、3、2，则电文经过编码后的总长度 L 可以由下式计算出来:

$$L=\sum_{i=1}^{n} c_i l_i$$

其中 n 表示电文中使用的字符种数，c_i 和 l_i 分别表示对应字符 k_i 在电文中的出现频率和编码长度。结合我们的例子，可以求出 L 为:

$$L=\sum_{i=1}^{6}(c_i\times 3)=3\times(4+2+6+8+3+2)=75$$

可知，采用等长编码时，传送电文的总长度为 75。

如何通过缩短传送电文的总长度，以节省传送时间呢？如果采用不等长编码，让出现频率高的字符具有较短的编码，让出现频率低的字符具有较长的编码，这样有可能缩短传送电文的总长度。采用不等长编码要避免译码的二义性或多义性。假设用 0 表示字符 D，用 01 表示字符 C，则当接收到编码串…01…，并译字符 0 时，是立即译出对应的字符 D，还是与下一个字符 1 一起译为对应的字符 C 呢？这种情况下就会产生二义性了。因此，如果对某一字符集进行不等长编码，则要求字符集中任一字符的编码都不能是其他字符编码的前缀。符合此要求的编码叫做前缀编码。显然，等长编码是前缀编码，这从等长编码所对应的编码二叉树也可以直观地看出，任一叶子结点都不可能是其他叶子结点的前驱，也就是说，只有当一个结点是另一个结点的前驱时，该结点的字符编码才会是另一个结点的字符编码的前缀。

为了使不等长编码为前缀编码，可以用该字符集中的每个字符作为叶子结点生成一棵编码二叉树。为了获得传送电文的最短长度，可以将每个字符的出现频率作为字符结点的权值赋予该结点上，求出此树的最小带权路径长度就等于求出传送电文的最短长度。因此，求传送电文的最短长度问题就转化为求由字符集中的所有字符作为叶子结点，由字符的出现频率作为其权值所产生的哈夫曼树的问题。

根据上面所讨论的例子，生成的哈夫曼树如图 6 所示，由编码哈夫曼树得到的字符编码称为哈夫曼编码。在图 6 中，A、B、C、D、E、F 这六个字符的哈夫曼编码依次为：00、1110、01、10、110、1111。电文的最短传送长度为：

$$L = WPL = \sum_{i=1}^{6} w_i l_i$$

$$= 4\times 2 + 2\times 4 + 6\times 2 + 8\times 2 + 3\times 3 + 2\times 4$$

$$= 61$$

显然，这比等长编码所得到的传送总长度 75 要小得多。

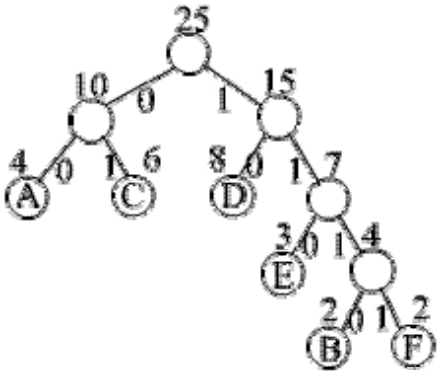


图 6　编码哈夫曼树

第四章 图

第一节 图的基本概念及存储结构

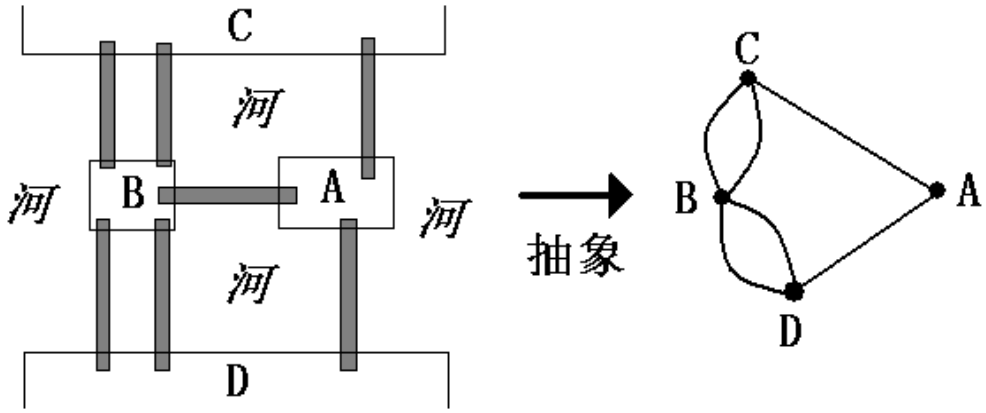
一、图的基本概念

线性表：数据间关系是线性的，每个元素只有一个前趋，一个后续，1：1；

树：有着明显的层次关系，每个元素只有一个前趋，但有多个后续，1：N；

图：数据之间的关系是任意，每个元素的前趋和后续个数是不定的，M：N；

引入：柯尼斯堡七桥问题，能否从 A 地发出，各座桥恰好通过一次，最后回到出发地 A?



结论：1736 年，数学家欧拉 首先解决了这个问题，由此开创了图论研究。这事实上是欧拉图的“一笔画问题”。答案是否定的，因为，对于每一个顶点，不论如何经过，必须有一条进路和一条出路，与每一个顶点相邻的线（关联边）必须是偶数条（除起点和终点外），而此图中所有点都只有奇数条关联边。在后面的应用中，我们将专门讨论这个问题。

定义：简单讲，一个图是由一些点和这些点之间的连线组成的。严格意义讲，图是一种数据结构，定义为：graph=（V，E）。V 是一个非空有限集合，代表顶点（结点），E 代表边的集合，一般用（V_x，V_y）表示，其中，V_x，V_y属于 V。

分类：如果边是没有方向的，称为“无向图”。表示时用一队圆括号表示，如：（V_x，V_y），（V_y，V_x），当然这两者是等价的。并且说边（V_x，V_y）依附于（相关联）顶点 V_x和 V_y。

如果边是带箭头的，则称为“有向图”，表示时用一队尖括号表示，此时<V_x，V_y>与<V_y，V_x>是不同的，如<V_x，V_y>的起点为 V_x，终点为 V_y。有向图中的边又称为弧。起点称为弧头、终点称为弧尾。

相邻：若两个结点 U、V 之间有一条边连接，则称这两个结点 U、V 是关联的。

带权图：两点间不仅有连接关系，还标明了数量关系（距离、费用、时间等）。

图的阶：图中结点的个数。

结点的度：图中与结点 A 关联的边的数目，称为结点 A 的度。

入度：在有向图中，把以结点 V 为终点的边的数目称为 V 的入度；

出度：在有向图中，把以结点 U 为起点的边的数目称为 U 的出度；

奇点：度数为奇数的结点；

偶点：度数为偶数的结点；

终端结点：在有向图中，出度为 0 的结点；

定理 1：图中所有结点的度数之和等于边数的 2 倍；

定理 2：任意一个图一定有偶数个奇点；

连通：如果图中结点 U，V 之间存在一条从 U 通过若干条边、点到达 V 的通路，则称 U、V 是连通的。

路(径)：从一个结点出发，沿着某些边连续地移动而到达另一个指定的结点，这种依次由结点和边组成的序列，叫“路”或者“路径”。

路径长度：路径上边的数目。

简单路径：在一个路径上，各个结点都不相同，则称为简单路径。下边的左图中 1—2—3 是一条简单路径，长度为 2，而 1—3—4—1—3 就不是简单路径。

回路：起点和终点相同的路径，称为回路，或“环”。下边的右图中 1-2-1 为一个回路。

完全图：一个 n 阶的完全无向图含有 n*(n-1)/2 条边；

一个 n 阶的完全有向图含有 n*(n-1)条边；

稠密图：当一个图的边数接近完全图时；

稀疏图：当一个图的边数远远少于完全图时；

在具体使用时，要根据是稠密图、还是稀疏图而选用不同的存储结构；

子图：从一个图中取出若干顶点、若干边构成的一个新图；

有根图：在一个图中，如果从顶点 U 到顶点 V 有路径，则称 U 和 V 是连通的；在一个图中，若存在一个顶点 W，它与其它顶点都是连通的，则称此图为有根图，顶点 W 即为它的根，下面的两个图都是有根图，其中，下边的左图的 1、2、3、4 都可以作为根；而下边的右图的 1、2 才可以作为根。

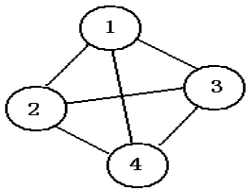
连通图：如果一个无向图中，任意两个顶点之间都是连通的，则称该无向图为连通图。否则称为非连通图；下边的左图为一个连通图。

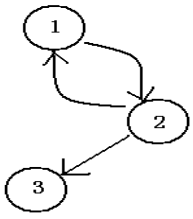
强连通图：在一个有向图中，对于任意两个顶点 U 和 V，都存在着一条从 U 到 V 的有向路径，同时也存在着一条从 V 到 U 的有向路径，则称该有向图为强连通图；下边的右图不是一个强连通图。

连通分支：一个无向图的连通分支定义为该图的最大连通子图，下边的左图的连通分支是它本身。

强连通分支：一个有向图的强连通分支定义为该图的最大的强连通子图，下边的右图含有两个强连通分支，一个是 1 和 2 构成的一个子图，一个是 3 独立构成的一个子图。

网络：带权的连通图。



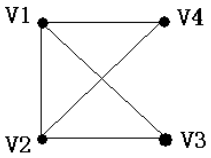


二、图的存储结构

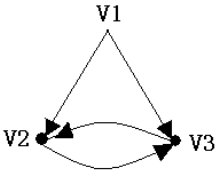
1、邻接矩阵表示法（顺序存储）

邻接矩阵是表示顶点之间相邻关系的矩阵，设 G={V,E}是一个度为 n 的图(顶点序号分别用 1,2,……，n 表示)，则 G 的邻接矩阵是一个 n 阶方阵，G[i,j]的值定义如下：

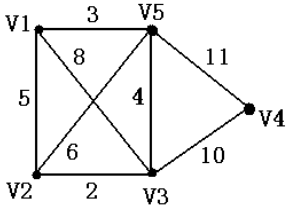
$$G[i,j]=\begin{cases}1\text{ 或权值} & \text{当 } v_i \text{ 与 } v_j \text{ 之间有边或弧时，取值为 1 或权值}\\ 0\text{ 或}\infty & \text{当 } v_i \text{ 与 } v_j \text{ 之间无边或弧时，取值为 0 或}\infty\text{（无穷大）}\end{cases}$$



图（A）



图（B）



图（C）

上图中的 3 个图对应的邻接矩阵分别如下：

$$G(A)=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}\quad G(B)=\begin{bmatrix}0&1&1\\0&0&1\\0&1&0\end{bmatrix}\quad G(C)=\begin{bmatrix}\infty&5&8&\infty&3\\5&\infty&2&\infty&6\\8&2&\infty&10&4\\\infty&\infty&10&\infty&11\\3&6&4&11&\infty\end{bmatrix}$$

我们发现：无向图的邻接矩阵是对称的。相应的数据结构可以定义为：

```
Const n=20；max=10000;              //对于带权图把 max 看成∞
Type g=Array[1..n,1..n] Of 0..1;    //对于带权图可以用 Integer、Real 等
```

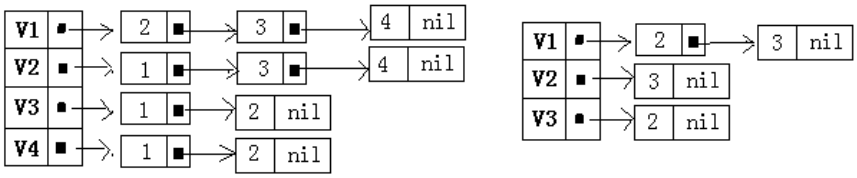
下面给出带权无向图的邻接矩阵建立过程：

```
const max=1e5;n=10;
type graph=array[1..n,1..n ] of real;
var i,j,k,e:integer;
    g:graph;   w:real;
begin
  For i:=1 To n Do              //初试化
     For j:=1 To n Do
        g[i,j]: =max;           //对于不带权的图 g[i,j]: =0
  Readln(e);                    //读入边的数目
  For k:= 1 To e Do
     Begin
        Read(i,j,w);            //读入两个顶点序号及权值
        g[i,j]:=w;              //对于不带权的图 g[i,j]: =1
        g[j,i]:=w;              //无向图的对称性
     End;
  for I:=1 to n do
```

```
    begin
      for j:=1   to   n do write(g[i,j]);   writeln;
    end;
END.
```

2、邻接表表示法（链式存储法）

以上图（A)、图（B）的邻接表分别如下，请大家自己画出图（C）的邻接表。



相应的数据结构描述为：

```
type grapglist=^enode;                  //边表
     enode=record
           adj:1..n;                    //边的起点
           next:grapglist;              //指向的下条边
     end;
     vnode=record                       //顶点表
           v:vtype;                     //顶点数据类型
           link:grapglist;              //指向的下个顶点
     end;
var adjlist=array[1..n] of vnode;       //n 个顶点的邻接表
```

下面给出有向图结构的邻接表的建立过程：

```
procedure createlist(var g:adjlist);
  var s:node;
  begin
      read(n,e);                        //n 为顶点数，e 为边数
      for I:=1 to n do                  //建立顶点表
        begin
          read(g[I].v);
          g[I].link:=nil;
        end;
      for k:= 1   to   e   do           //建立边表
        begin
           read(I,j); new(s);
           s^.adj：=j;                  //新结点的序号为读入的终点
           s^.next:=g[I].link;          //把新结点插入到当前结点 I 之后
           g[I].link:=s;
         end;
   end.
```

第二节　图的遍历

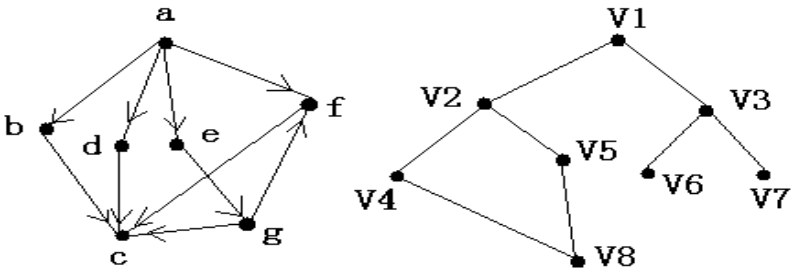
一、概念

从图中某一顶点出发系统地访问图中所有顶点，使每个顶点恰好被访问一次，这种运算操作被称为图的遍历。为了避免重复访问某个顶点，可以设一个标志数组 visited[i]，未访问时值为 false，访问一次后就改为 true。

图的遍历分为深度优先遍历和广度（宽度）优先遍历两种方法。

二、深度优先遍历

图的深度优先搜索类似于树的先序遍历。从图中某个顶点 V_i 出发，访问此顶点并作已访问标记，然后从 V_i 的一个未被访问过的邻接点 V_j 出发再进行深度优先遍历，当 V_i 的所有邻接点都被访问过时，则退回到上一个顶点 V_k，再从 V_k 的另一个未被访问过的邻接点出发进行深度优先遍历，直至图中所有顶点都被访问到为止。如下面两个图的深度优先遍历结果分别为：a，b，c，d，e，g，f 和 V1,V2,V4,V8,V5,V3,V6,V7。



对于一个连通图，深度优先遍历的递归过程如下(对于非连通图，要多次调用本过程)

```
Procedure   dfs(i: integer);                    //图用邻接表存储，其它存储方式只要对本过程稍作修改
  Begin
      Write(g[i].v);                            //输出——最简单的访问方式
      Visited[i]:=True;  p:=g[i].link;
      While p<>Nil   Do                         //按深度优先搜索的顺序遍历与 i 相关联的所有顶点
        Begin
          j:  =p^.adj;                          //j 为 i 的一个后继
          If Not Visited[j]   Then dfs(j);      //深度、递归
          p:=p^.next;                           //p 为 i 的下一个后继
        End;
  End;
Procedure   dfs(i:integer);                     //图用邻接矩阵存储
  Begin
      访问顶点 i;
      Visited[i]:=True;
      For j:=1   to   n do                      //按深度优先搜索的顺序遍历与 i 相关联的所有顶点
        Begin
          If (Not Visited[j]) and (a[i,j]=1)   Then   dfs(j);
        End;
  End;
```

以上 dfs(i)的时间复杂度为 O（n*n)。对于一个非连通图，调用一次本过程 dfs(i)，即按深度优先顺序依次访问了顶点 i 所在的（强）连通分支。主程序如下：

```
procedure work;                          //邻接矩阵
begin
  fillchar(Visited,sizeof(Visited),0);   //初始化
  for i:=1 to n do                       //深度优先搜索每一个为访问的顶点
     if not Visited then dfs(i);
end;
```

例 1、液晶数字显示

如图是液晶屏显示的十个阿拉伯数字（7 笔划），这里把横和竖的一个短划都称为一笔，如 7 有 3 笔，8 有 7 笔。

编一个程序，重新排列十个数，使其相邻数字都可以由另一个数字加上几笔或减去几笔组成（如：4107395682），但不能又加又减。

打印所有可能的排列。

【算法分析】

显然，对于任意一个数字，我们可以加上一笔（或减去一笔），去判断能否转变成其它数字，这样不断深度优先搜索下去，程序如下：

```
program ex1;
const a:array[0..9,0..9] of integer=((0,1,0,0,0,0,0,1,1,0),
                                     (1,0,0,1,1,0,0,1,1,1),
                                     (0,0,0,0,0,0,0,0,1,0),
                                     (0,1,0,0,0,0,0,1,1,1),
                                     (0,1,0,0,0,0,0,0,1,1),
                                     (0,0,0,0,0,0,1,0,1,1),
                                     (0,0,0,0,0,1,0,0,1,0),
                                     (1,1,0,1,0,0,0,0,1,1),
                                     (1,1,1,1,1,1,1,1,0,1),
                                     (0,1,0,1,1,1,0,1,1,0)); //0,1 分别表示两数字间是否可以转换
var
   b:array[1..10]of integer;
   c:array[0..9]of 0..1;
   i,s:integer;

procedure try(k:integer);
var j,m:integer;
begin
```

```
      for j:=0 to 9 do
      begin
        if (a[b[k-1],j]=1) and (c[j]=1) then      //如果满足条件，则搜索下一位数
        begin
          b[k]:=j;   c[j]:=0;
          if k<10 then try(k+1)
                  else begin
                         for m:=1 to 10 do write(b[m]);
                         writeln;
                         s:=s+1;
                       end;
          c[j]:=1;
        end;
    end;
end;
begin
      for i:=0 to 9 do c[i]:=1;           //一开始所有数都没有出现过
      for i:=0 to 9 do
      begin
           b[1]:=i;
           c[i]:=0;
           try(2);
           c[i]:=1;
      end;
      writeln(s);
end.
```

【程序输出】

```
0731495682
2807314956
2837014956
2841073956
2865937014
2865941073
2865941370
3701495682
4107395682
6593701482
6594107382
6594137082
total=12
```

例 2、有趣的四色问题

人人都熟悉地图，可是绘制一张普通的政区图，至少需要几种颜色，才能把相邻的政区或区域通过不同的颜色区分开来，就未必是一个简单的问题了。

这个地图着色问题，是一个著名的数学难题。大家不妨用一张中国政区图来试一试，无论从哪里开始着色，至少都要用上四种颜色，才能把所有省份都区别开来。所以，很早的时候就有数学家猜想："任何地图的着色，只需四种颜色就足够了。"这就是"四色问题"这个名称的由来。

数学史上正式提出"四色问题"的时间是在 1852 年。当时伦敦的大学的一名学生法朗西斯向他的老师、著名数学家、伦敦大学数学教授莫根提出了这个问题，可是莫根无法解答，求助于其它数学家，也没有得到答案。于是从那时起，这个问题便成为数学界的一个"悬案"。

一直到二十年前的 1976 年 9 月，《美国数学会通告》正式宣布了一件震撼全球数学界的消息：美国伊利诺斯大学的两位教授阿贝尔和哈根，利用电子计算机证明了"四色问题"这个猜想是完全正确的！他们将普通地图的四色问题转化为 2000 个特殊图的四色问题，然后在电子计算机上计算了足足 1200 个小时，最后成功地证明了四色问题。现在，就让我们也来当一当解决“悬案”的高手。

【问题描述】

设有如下图所示的地图，每个区域代表一个省，区域中的数字代表省的编号，将每个省涂上红（R），蓝（B），白（W），黄（Y）四种颜色之一，使相邻的省份有不同的颜色。

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td colspan="3">5</td><td>6</td></tr>
</table>

输入数据格式:
6
0 1 0 0 1 0
1 0 1 0 1 0
0 1 0 1 1 0
0 0 1 0 0 1
1 1 1 0 0 1
0 0 0 1 1 0
输出结果:
336　//方案数

【输入格式】

用邻接矩阵表示地图。从文件中读入，文件格式如下:

N （有 N 个省）

N 行用空格隔开的 0/1 串 （1 表示相邻，0 表示不相邻）

【输出格式】

RBWY 串

【算法分析】

这是一道非常典型的图的深搜。在填写每一个省的颜色时检查与相邻已填省份的颜色是否相同。如果不同，则填上；如果相同（冲突），则另选一种；如果已没有颜色可供选择，则回溯到上一省份。重复这一过程，直到所有省的颜色都已填上。

最主要的问题在于如何解决相邻省的颜色冲突。对每一个省份，可供选择的颜色一共有四种；对省份 I 来说颜色 X 可填的条件是编号为 1~(I-1)且与省 I 相邻的省份的颜色都不是 X。

【数据结构】

1、解决方案：用数组 S 存储。1-4 表示四种颜色，输出时再转化成字符；

2、地图：用 N X N 数组 A 存储。a[i,j]=0 则表示省 I 和省 J 不相邻，a[i,j]=1 则表示相邻。

【参考程序】

```
const num=20;
var a:array [1..num,1..num] of 0..1;
    s:array [1..num] of 0..4;        //用 1-4 分别代表 RBWY 四种颜色；0 代表未填进任何颜色
    k1,k2,n:integer;
```

```
function pd(i,j:integer):boolean;          //判断可行性：第 I 个省填上第 J 种颜色
var k:integer;
  begin
     for k:=1 to i-1 do
        if (a[i,k]=1) and (j=s[k]) then   //省 I 和省 J 相邻且将填进的颜色和已有的颜色相同
            begin pd:=false; exit; end;
     pd:=true;
  end;
procedure print;                           //打印结果
var k:integer;
  begin
      for k:=1 to n do                     //将数字转为 RBWY 串
          case s[k] of
            1:write('R':4);
            2:write('B':4);
            3:write('W':4);
            4:write('Y':4);
          end;
       writeln;
  end;
procedure try(i:integer);
var j:integer;
  begin
    for j:=1 to 4 do
       if pd(i,j) then begin
                          s[i]:=j;
                          if i=n then print else try(i+1);
                          s[i]:=0;
                       end;
  end;
BEGIN
   readln(n);
   for k1:=1 to n do
     begin
       for k2:=1 to n do read(a[k1,k2]);
       readln;
     end;
   for k1:=1 to n do s[k1]:=0;
   try(1);
END.
```

三、广度（宽度）优先遍历

从图中某个顶点 V_1 出发，访问此顶点，然后依次访问与 V_1 邻接的、未被访问过的所有顶点，然后再分别从这些顶点出发进行广度优先遍历，直到图中所有被访问过的顶点的相邻顶点都被访问到。若此时图中还有顶点尚未被访问，则另选图中一个未被访问过的顶点作为起点，重复上述过程，直到图中所有顶点都被访问到为止。

如上面两个图的广度优先遍历结果分别为：a，b，d，e，f，c，g。

V1，V2，V3，V4，V5，V6，V7，V8。

两种遍历方法相比，深度优先遍历实际上是尽可能地走“顶点表”；而广度优先遍历是尽可能沿顶点的“边表”进行访问，然后再沿边表对应顶点的边表进行访问，因此，有关边表的顶点需要保存(用队列，先进先出)，以便进一步进行广度优先遍历。下面是广度优先遍历的过程：

```
Procedure bfs(i:integer);                    //图用邻接矩阵表示
  Begin
    访问顶点 i;
    Visited[i]:=true;
    顶点 i 入队 q;
    while 队列 q 非空 do
      begin
        从队列 q 中取出队首元素 v;
        for j:=1 to n do
            begin
                if (not Visited[j]) and (a[v,j]=1) then
                    begin
                        访问顶点 j;
                        Visited[j]:=true;
                        顶点 j 入队 q
                    end;
            end;
      end;
   End;
```

以上 bfs(i)的时间复杂度仍为 O（n*n)。对于一个非连通图，调用一次本过程 bfs(i)，即按广度优先顺序依次访问了顶点 i 所在的（强）连通分支。主程序如下：

```
  procedure work;                            //邻接矩阵
    begin
      fillchar(Visited,sizeof(Visited),0);
      for i:=1 to n do                       //广度优先搜索每一个为访问的顶点
          if not Visited[i] then bfs(i);
    end;
```

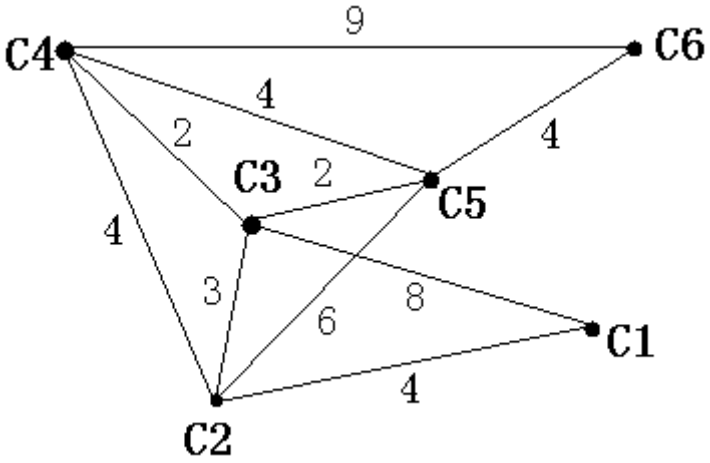
四、图的最短路径算法

在带权图 G=（V，E）中，若顶点 Vp,Vq 是图 G 的两个顶点，从顶点 Vp 到 Vq 的路径长度定义为路径上各条边的权值之和。从顶点 Vp 到 Vq 可能有多条路径，其中路径长度最短的一条路径称为顶点 Vp 到 Vq 的最短路径。

有两类最短路径问题：一是求从某个顶点（源结点）到其它顶点（目的结点）的最短路径问题。二是求图中每一对顶点间的最短路径。

例 3、最短路径问题

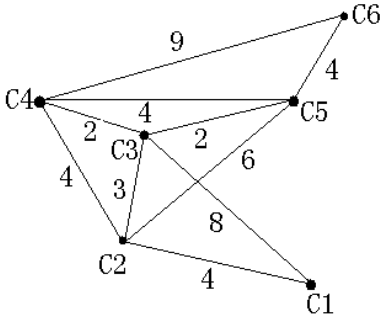
下图是六个城市之间道路联系的示意图，连线表示两城市间有道路相通，连线旁的数字表示路程。请编写一程序，由计算机找出从 C1 城到 C6 城之间路长最短的一条路径，输出路径序列及总长度。



link[i,j]:=0 表示城市 i 与城市 j 之间没有通路，否则为权值。基本思想为：假设 K 为当前城市编号，R 为下一城市编号（2<=R<=6），按下列规则进行遍历：

if link(k,r)>0 且 Cr 没有被访问过 面 then 记录该城市编号，k:=r。

可以用深度优先遍历，也可用广度优先遍历。下面的程序采用后者，前者留给大家完成。注意，广度优先遍历时，最先找到的路径未必是最短路径，而只是走过城市最少的路径而已。所以，找到一条路径后应采用“打擂台”的思想保存最小值。另外，按照广度优先遍历的要求，设一个 pnt 数组和 open、closed 用于队列存放之间结点。



【参考程序】

```
const max=maxint;
      link:array[1..5,1..6] of integer=((0,4,8,0,0,0),
                                        (4,0,3,4,6,0),
                                        (8,3,0,2,2,0),
                                        (0,4,2,0,4,9),
                                        (0,6,2,4,0,4));
type fg=set of 1..6;
var mincost,step,open,closed,i,k,n,r:integer;
    path:array[1..7] of 1..6;              //存放最终结果
    flag:array[1..100] of fg;              //标志数组，队列，头尾指针分别为 open,closed
```

```
    city,pnt:array[1..100] of byte;        //数值数组，pnt 用于队列
procedure exam;                            //判断最短路径和求和
  var n,i,y,cost:integer;
      s:array[1..7]   of 1..6;
  begin
    y:=open;
    n:=0; cost:=0;
    while y> 0 do begin inc(n);s[n]:=y;y:=pnt[y]; end; //计算步长（深度 ）
    for i:=n-1   downto 1 do                          //计算路径
      cost:=cost+link[city[s[i+1]],city[s[i]]];
    if cost<mincost then begin step:=n-1;             //记忆最短路径
                               mincost:=cost;
                               for i:=1   to   step   do path[i]:=city[s[i]];
                         end;
  end;
procedure print;                                      //输出 path 数组和 mincost
  begin
    writeln('path:');
    write('1');
    for i:=step downto 1 do write('->',path[i]);
    writeln;
    writeln('mincost=',mincost);
  end;
BEGIN
  mincost:=max;                            //打擂台，赋初值
  flag[1]:=[1];
  city[1]:=1;                              //初始化，从第一个结点 C1 开始遍历
  n:=0;
  pnt[1]:=0;                               //队列初始化
  closed:=0;
  open:=1;
  repeat                                   //直到队列空，结束程序
    inc(closed);
    k:=city[closed];
    if k<>6 then begin                     //判断有没有到达目的结点 C6
                   for r:=2   to 6 do   //广度遍历
                     if (not(r in flag[closed])) and (link[k,r]>0)   then
                       begin          //没访问过，则进队
                         inc(open);
                         city[open]:=r;
```

```
                    flag[open]:=flag[closed]+[r];
                    pnt[open]:=closed;
                    if r=6 then exam;
                  end;
              end;
  until closed>=open;
  print;
  readln;
END.
```

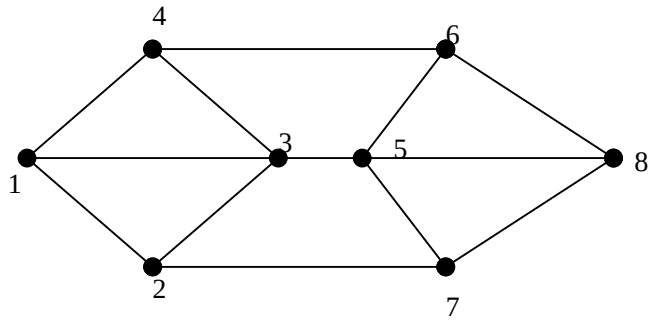
【运行结果】

path:1-→2-→3-→5-→6

mincost = 13

例 4：最短路径问题

求从任意一个顶点 V_i 出发，对给出的图，求到达任意顶点 V_j（i<>j）的所有最短路径。



【输入样例 1】

start:1

end:8

【输出样例 1】

1－→2－→7－→8

1－→3－→5－→8

1－→4－→6－→8

1－→8

total=3　step=3

【输入样例 2】

start:2

end:6

【输出样例 2】

2－→1－→4－→6

2－→3－→4－→6

2－→3－→5－→6

2－→7－→5－→6

2－→7－→8－→6

2－→6

total=5　step=3

【算法分析】

1、首先可以用如下的邻接表表示此图各端点的邻接关系：

顶点	接邻顶点				个数
1	2	3	4	-	3
2	1	3	7	-	3
3	1	2	4	5	4
4	1	3	6	-	3
5	3	6	7	8	4
6	4	5	8	-	3
7	2	5	8	-	3
8	5	6	7	-	3

即：const d=array[1..8,1..4] of byte =

((2,3,4,0),(1,3,7,0),(1,2,4,5),(1,3,6,0),(3,6,7,8),(4,5,8,0),(2,5,8,0),(5,6,7,0))

```
var   a:array[1..50000] of integer;        //队列
      head,tail:integer;                   //f 队头指针，r 队尾指针
      b:array[1..50000] of integer;        //链接表，表示某一个结点的前趋结点
      c:array[1..50000] of integer;        //存储到达目标结点后各最短路径的线路
```

2、算法描述：

⑴ 设立初值，并令起始结点进队：

head:=0;tail:=1;a[1]:=start; b[1]:=0; c[1]:=1;

⑵ 将顶点的顺序出队，并访问与该层各顶点相邻接但又没有访问过的顶点，同时将这些结点进队列，且设立前趋链接指针，若此时的结点为目标结点，输出最短路径。

⑶ 打印搜索到的各条最短路径的各结点编号，并结束程序。

【参考程序】

```
Program ex4;
const d:array[1..8,1..4] of
          byte=((2,3,4,0),(1,3,7,0),(1,2,4,5),(1,3,6,0),(3,6,7,8),(4,5,8,0),(2,5,8,0),(5,6,7,0));
        n:array[1..8] of byte=(3,3,4,3,4,3,3,3);
var
   a,b,c:array[0..50000] of   integer;
   i,j,k,l,start,ed,total,head,tail,max:longint;

procedure seach(tail:integer);                  //调用递归过程，一步一步返回起始点
var   i:integer;
begin
    if b[tail]=0 then i:=1                      //到起始点
                 else seach(b[tail]);           //调用前趋结点
   if a[tail]<>ed   then   write(a[tail],'-->')  //输出最短路径的线路
                    else writeln(a[tail]);
end;
```

```
begin
   write('start:');readln(start);                  //输入起点
   write('end:');readln(ed);                       //输入终点
   head:=0;   tail:=1;                             //队头、队尾指针
   a[1]:=start;                                    //第 1 个结点入队
   b[1]:=0;                                        //第 1 个结点的前趋结点 0
   c[1]:=1;                                        //第 1 个结点路径的长度
   max:=maxint;
   repeat
     inc(head);                                    //队头结点出队
     if c[head]<max then begin                     //算出最短路径的长度
       for i:=1 to 4 do
         if (d[a[head],i]>0) then                  //相邻接的结点
         begin
           inc(tail);                              //队尾指针加 1
           a[tail]:=d[a[head],i];                  //相邻接的结点的入队
           b[tail]:=head;                          //保存前一个结点
           c[tail]:=c[head]+1;                     //本结点的长度为 head 结点长度加 1
           if a[tail]=ed then                      //此时的结点为目标结点
           begin
             seach(tail);                          //调用递归过程，输出最短路径的线路
             if c[tail]<max then max:=c[tail]; //保存最短路径的长度
             total:=total+1;
           end;
         end;
     end;
   until head>=tail;
   writeln(start,'-->',ed);                        //按样例的格式输出
   write('total=',total,'      ');
   writeln('step=',max-1);
end.
```

【上机练习】

1、液晶数字显示（题目见例 1，文件名 Ex1.pas）
2、有趣的四色问题（题目见例 2，文件名 Ex2.pas）
3、最短路径问题（题目见例 3，文件名 Ex3.pas）
4、最短路径问题（题目见例 4，文件名 Ex4.pas）

第三节　无向图的传递闭包问题

无向图的传递闭包主要用于判断图的连通性和图中满足条件的连通分支，具有很高的实用价值。而且，借鉴无向图的传递闭包思想，可以计算图中每一对顶点之间的最短路径（实际上就是 Floyed 算法的思想）。

一、 判断任两个顶点之间是否有路

例 1、输入一张无向图，指出该图中哪些顶点对之间有路。

输入：n（顶点数，1<=n<=20）
　　e（边数，1<=e<=210）
　　以下 e 行，每行为有边连接的一对顶点。
输出：k 行，每行两个数，为存在通路的顶点对序号 i,j，输出时要求 i<j。

输入: 3 4
　　1 3
　　3 4
　　4 2
　　2 1

输出: 1 2
　　1 3
　　2 3

【算法分析】

很容易想到，可以用宽度优先或深度优先遍历来解决。因为从任意一个顶点出发，进行一次遍历，就可以求出此顶点和其它各个顶点的连通状况。所以只要把每个顶点作为出发点都进行一次遍历，就能知道任意两个顶点之间是否有路存在。

一次遍历的时间复杂度为 O（n），要穷举每个顶点，所以总的时间复杂度为 O（n*n）。

【参考程序 1】

```
program ex1_1;
const   maxn=20;
var
   link,longlink:array[1..maxn,1..maxn] of boolean;
   visit:array[1..maxn] of boolean;
   n,e,i,j,x,y:longint;

procedure dfs(i:longint);
var j:longint;
begin
```

```
    for j:=1 to n do
        if (not visit[j])and(link[i,j]) then
            begin
                visit[j]:=true;
                dfs(j);
            end;
end;

begin
    read(n,e);
    fillchar(link,sizeof(link),false);
    for i:=1 to e do
      begin
        read(x,y);
        link[x,y]:=true;
        link[y,x]:=true;
      end;

    fillchar(longlink,sizeof(longlink),false);
    for i:=1 to n do
      begin
        fillchar(visit,sizeof(visit),false);
        visit[i]:=true;
        dfs(i);
        for j:=1 to n do longlink[i,j]:=visit[j];
      end;

    for i:=1 to n-1 do
        for j:=i+1 to n do
            if longlink[i,j] then
                writeln(i,' ',j);
end.
```

2、设 link,longlink:array[1..20,1..20] of Boolean；分别存放无向图和它的传递闭包。若 longlink[i,j]=true，表示顶点对 i,j 之间有路；否则无路。

我们采用递推（迭代）的方法不断对 longlink 进行运算（产生 $longlink^{(0)}$, $longlink^{(1)}$,……, $longlink^{(n)}$）。在递推的过程中，路径长度的“+”运算和比较大小的运算用相应的逻辑运算符“and”和“or”代替。对于 i，j 和 k=1,……,n，如果图中顶点 i 至顶点 j 间存在通路且通路上所有顶点的序号均属于{1,2,……,k}，则定义 $longlink_{ij}^{(k)}$=true；否则值为 false。有：

$longlink_{ij}^{(k)}= longlink_{ij}^{(k-1)}$ or ($longlink_{ik}^{(k-1)}$ and $longlink_{kj}^{(k-1)}$)

由于布尔型的存储量少于整数，且位逻辑运算的执行速度快于算术运算，所以空间和时间效率都很好。传递闭包的计算过程如下：

```
longlink 的初值赋为 link;
for k:=1 to n do
  for i:=1 to n do
    for j:=1 to n do
      longlink[i,j]= longlink[i,j] or (longlink[i,k] and longlink[k,j]);
```

显然计算的时间复杂度为 O（n*n*n）。了解 Floyed 算法求最短路径问题的学生，一眼就应该看出这个程序段和思想与 Floyed 算法完全一致，程序如下：

【参考程序 2】

```
program ex1_2;
const   maxn=20;
var
   link,longlink:array[1..maxn,1..maxn] of boolean;
   n,e,i,j,k,x,y:longint;
begin
      read(n,e);
      fillchar(link,sizeof(link),false);
      for i:=1 to e do
      begin
           read(x,y);
           link[x,y]:=true;
           link[y,x]:=true;
      end;
       longlink:=link;

      for k:=1 to n do
          for i:=1 to n do
              for j:=1 to n do
                   longlink[i,j]:=longlink[i,j] or longlink[i,k] and longlink[k,j];

      for i:=1 to n-1 do
          for j:=i+1 to n do
              if longlink[i,j] then
                  writeln(i,' ',j);
end.
```

二、 寻找满足条件的连通分支

例 2、输入一张顶点带权的无向图，分别计算含顶点数最多的一个连通分支和顶点的权之和最大的一个连通分支。

输入：n（顶点数，1<=n<=20）

以下 n 行，依次表示顶点 1 ~ 顶点 n 上的权；

e（边数，1<=e<=210）

以下 e 行，每行为有边连接的一对顶点。

输出：两行，一行为含顶点数最多的一个连通分支，一行为顶点的权之和最大的一个连通分支，输出时按顶点编号从小到大输出。

输入:5

3

4

5

8

10

3

1 2

1 3

4 5

输出: 1 2 3

4 5

【算法分析】

我们可以先通过例 1 的 longlink 计算出每个顶点所在的连通分支，然后在所有可能的连通分支中找出满足条件的解即可。至于计算连通分支的顶点方案，只要分别从连通分支中任选一个代表顶点，由此出发，通过深度优先搜索即可得到顶点方案。设:

best,besti 分别存放含顶点数最多的连通分支中的顶点数和代表顶点；

max,maxk 分别存放顶点的权之和最大的连通分支的顶点权之和和代表顶点；

计算 best,besti,max,maxk 的过程如下:

1、读入无向图的信息；

2、计算传递闭包 longlink；

3、穷举每一个顶点

```
for I:=1 to n do
begin
   k:=0;s:=0
   for j:=1 to n do        //计算顶点 i 所在连通分支中的顶点总数 k 和顶点的权之和 s
     if longlink[i,j] then begin
                             inc(k);
                             inc(s,顶点 j 的权)
                           end;
   if k>best then begin best:=k;besti:=I end;
```

```
                                    //若 k 为目前最大，则记入 best,i 作为代表顶点记入 besti
            if s>max then begin max:=s;maxk:=I end;
                                    //若 s 为目前最大，则记入 max,i 作为代表顶点记入 maxk
            if k=n then break;      //若整个图是连通图，则退出
        end;
```

4、dfs(besti); {从代表顶点 besti 出发，深度优先搜索含顶点数最多的连通分支}

5、dfs(maxk); {从代表顶点 maxk 出发，深度优先搜索顶点的权之和最大的连通分支}

显然，以上算法的时间复杂度为 O（n*n)。

【参考程序】

```
program ex2;
const   maxn=20;
var
   w:array[1..maxn] of longint;
   link,longlink:array[1..maxn,1..maxn] of boolean;
   out:array[1..maxn] of boolean;
   n,e,i,j,k,s,x,y,best,besti,max,maxk:longint;

procedure dfs(k:longint);
var i:longint;
begin
      for i:=1 to n do
          if (longlink[k,i])and(not out[i]) then
             begin
                  out[i]:=true;
                  dfs(i);
             end;
end;

begin
      read(n);
      for i:=1 to n do read(w[i]);
      read(e);
      fillchar(link,sizeof(link),false);
      for i:=1 to e do
      begin
           read(x,y);
           link[x,y]:=true;
           link[x,y]:=true;
      end;
```

```
    longlink:=link;
    for k:=1 to n do
        for i:=1 to n do
            for j:=1 to n do
                longlink[i,j]:=longlink[i,j] or longlink[i,k] and longlink[k,j];
    best:=1; besti:=1;
    max:=w[1]; maxk:=1;
    for i:=1 to n do
    begin
        k:=0;s:=0;
        for j:=1 to n do
            if longlink[i,j] then
                begin
                    inc(k);
                    inc(s,w[i])
                end;
        if k>best then begin best:=k;besti:=i; end;
        if s>max then begin max:=s;maxk:=i; end;
        if k=n then break;
    end;

    fillchar(out,sizeof(out),false);
    out[besti]:=true;
    dfs(besti);
    for i:=1 to n do
        if out[i] then write(i,' ');
    writeln;

    fillchar(out,sizeof(out),false);
    out[maxk]:=true;
    dfs(maxk);
    for i:=1 to n do
        if out[i] then write(i,' ');
    writeln;
end.
```

三、欧拉回路

1、欧拉路：在无孤立顶点的图中，若存在一条路，经过图中每条边一次且仅一次，则称此路为欧拉路。如下图 1（左）中存在一条从顶点 1 到顶点 6 的欧拉路。后面的例题（一笔画问题）本质上就是判断一个图是否存在欧拉路。

2、欧拉回路：在无孤立顶点的图中，若存在一条路，经过图中每条边一次且仅一次，且回到原来位置，则称此路为欧拉回路。如下图 1（右）中任意两个顶点之间都存在欧拉回路。著名的柯尼斯堡七桥问题（图论起源）本质上就是讨论一个图的欧拉回路问题。

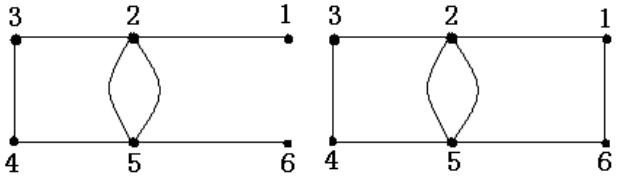


图 1

3、欧拉图：存在欧拉回路的图，称为欧拉图，上图 1（右）所示的图就是一个欧拉图。

4、定理 1：存在欧拉路的条件：图是连通的，且存在 0 个或 2 个奇点。如果存在 2 个奇点，则欧拉路一定是从一个奇点出发，以另一个奇点结束。

5、定理 2：存在欧拉回路的条件：图是连通的，且不存在奇点。

6、哈密尔顿图：在无孤立顶点的连通图中，若存在一条路，经过图中每个顶点一次且仅一次，则称此图为哈密尔顿图。

7、哈密尔顿环：是一条沿着图的 n 条边环行的路径，它访问每一个顶点一次且仅一次，并且返回到它的开始位置。

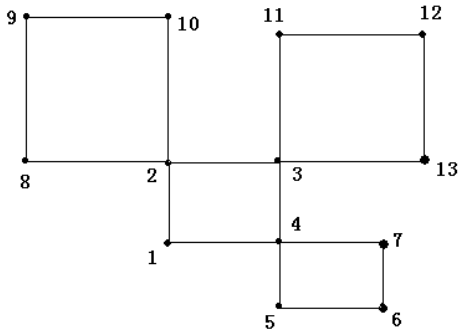


图 3

8、寻找欧拉回路的算法

寻找欧拉回路的算法有多种，但有些算法不能很好地解决如上图 3 的情形，下面介绍一种基于递归的经典算法框架：

```
find_circuit(结点 i);
    当结点 i 有邻居时
        {  选择任意一个邻居 j;
           删除边(i,j);
           find_circuit（结点 j);
        }
    circuit[circuitpos]=结点 i;
    circuitpos:=circuitpos+1;
```

如果寻找欧拉回路，对任意一个点执行 find_circuit。如果是寻找欧拉路径，对一个奇点执行 find_circuit。算法的时间复杂度为 O(m+n)。

例 3、寻找一个图的欧拉路的算法实现

输入:

5 5

1 2

2 3

3 4

4 5

5 1

输出:

1 5 4 3 2 1

【参考程序】(Euler.pas)

```
program Euler;
const   maxn=100;
var
    g:array[1..maxn,1..maxn] of longint;
    du:array[1..maxn] of longint;
    circuit:array[1..maxn] of longint;
    n,e,circuitpos,i,j,x,y,start:longint;

procedure find_circuit(i:longint);
var j:longint;
begin
      for j:=1 to n do
          if g[i,j]=1 then
              begin
                    g[i,j]:=0;
                    g[j,i]:=0;
                    find_circuit(j);
              end;
      circuitpos:=circuitpos+1;
      circuit[circuitpos]:=i;
end;

begin
      fillchar(g,sizeof(g),0);
      read(n,e);
      for i:=1 to e do
      begin
            read(x,y);
            g[x,y]:=1;
```

```
            g[y,x]:=1;
            du[x]:=du[x]+1;
            du[y]:=du[y]+1;
        end;

        start:=1;
        for i:=1 to n do
            if du[i] mod 2 =1 then
                start:=i;

        circuitpos:=0;
        find_circuit(start);

        for i:=1 to circuitpos do write(circuit[i],' ');
        writeln;
end.
```

9、寻找哈密尔顿环的算法

到现在为止寻找哈密尔顿环并没有一种有效的算法，一般只能用搜索解决。

与判断欧拉图、找出欧拉回路相对应，如何判断一个图是否是哈密尔顿图、又如何找出一个图中的所有哈密尔顿环呢？假设用 X[1..k]表示求得的解，其中 X[i]是找到的环中第 i 个被访问的顶点，我们先用递归算法求出从 X[1..k-1]生成 X[k]的过程：

```
Procedure Next(k);
Var x:array[1..n] of integer;
    graph:array [1..n,1..n] of boolean;        //布尔型邻接矩阵
    k,j:integer;
Begin
  x[k]:=(x[k]+1) mod  (n+1);
  If x[k]=0 Then Return;
  If graph[x[k-1],x[k]] Then                   //有边相连
     For  j:=1 To k-1 Do                       //查与前 k-1 个顶点是否相同
       If  x[j]=x[k] Then Exit;                //有则出循环
  If  j=k  Then                                //有一个不同顶点
  If  (k<n) or ((k=n) and (graph[x[n],1])) Then Return;
End;
```

再利用下面的回溯法就可以求出所有的哈密尔顿环了：

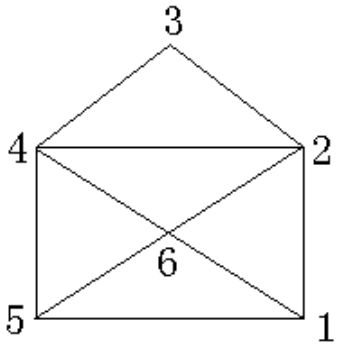
```
Procedure Hamilton(k);
Var  x:array[1..n] of integer;
     k,n:integer;
Begin
  Next(k);
  If x[k]=0 Then Return;
  If k=n Then Print(x)                         //Print 为输出一个方案的过程
         Else Hamilton(k+1);
End;
```

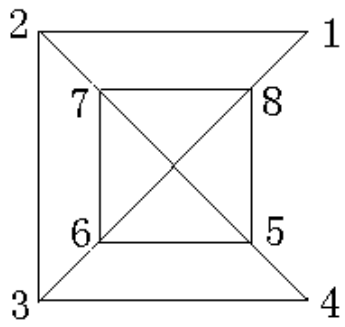
四、应用举例

例 4、一笔画问题（one.pas）

【问题描述】

编程对给定的一个图，判断能否一笔画出，若能请输出一笔画的先后顺序，否则输出“No Solution!”。





【输入格式】

输入文件名 one.in，共 n+1 行，第 1 行为图的顶点数 n，接下来的 n 行（每行 n 个数据）为图的邻接矩阵，G[i,j]=1 表示顶点 i 和顶点 j 有边相连，G[i,j]=0 表示顶点 i 和顶点 j 无边相连。

【输出格式】

输出文件名 one.out，若能一笔画出，输出一笔画出的顶点先后顺序，否则输出“No Solution!”。

【输入样例】

```
6
0 1 0 0 1 1
1 0 1 1 0 1
0 1 0 1 0 0
0 1 1 0 1 1
1 0 0 1 0 1
1 1 0 1 1 0
```

【输出样例】

```
5--->1--->2--->3--->4--->2--->6--->4--->5--->6--->1
```

【算法分析】

由数学知识可知：当一个图的顶点全是偶点或仅有两个奇点时才能一笔画出，而且此图必须是连通图。

【参考程序】

```
program one;
const   maxn=100;
var
   g:array[1..maxn,1..maxn] of longint;
   du:array[1..maxn] of longint;
   circuit:array[1..maxn] of longint;
   n,circuitpos,i,j,start,oddnumber:longint;
```

```
procedure find_circuit(i:longint);
var j:longint;
begin
    for j:=1 to n do
      if g[i,j]=1 then
        begin
          g[i,j]:=0;
          g[j,i]:=0;
          find_circuit(j);
        end;
    circuitpos:=circuitpos+1;
    circuit[circuitpos]:=i;
end;

begin
    assign(input,'one.in');   reset(input);
    assign(output,'one.out');rewrite(output);

    read(n);
    for i:=1 to n do
    begin
      du[i]:=0;
      for j:=1 to n do
      begin
        read(g[i,j]);
        du[i]:=du[i]+g[i,j];
      end;
    end;

    start:=1; oddnumber:=0;
    for i:=1 to n do
      if du[i] mod 2 =1 then
        begin
          start:=i;
          oddnumber:=oddnumber+1;
        end;

    if (oddnumber>2)or(oddnumber=1)
      then writeln('No Solution!')
      else begin
```

```
                    circuitpos:=0;
                    find_circuit(start);
                    for i:=1 to circuitpos-1 do write(circuit[i],'--->');
                    writeln(circuit[circuitpos]);
                end;
    close(input);close(output);
end.
```

例 5、铲雪车问题(snow.pas)

【问题描述】

随着白天越来越短夜晚越来越长，我们不得不考虑铲雪问题了。整个城市所有的道路都是双车道，因为城市预算的削减，整个城市只有 1 辆铲雪车。铲雪车只能把它开过的地方（车道）的雪铲干净，无论哪儿有雪，铲雪车都得从停放的地方出发，游历整个城市的街道。现在的问题是：最少要花多少时间去铲掉所有道路上的雪呢?

【输入格式】

输入数据的第 1 行表示铲雪车的停放坐标（x,y)，x，y 为整数，单位为米。下面最多有 100 行，每行给出了一条街道的起点坐标和终点坐标，所有街道都是笔直的，且都是双向一个车道。铲雪车可以在任意交叉口、或任何街道的末尾任意转向，包括转 U 型弯。铲雪车铲雪时前进速度为 20 km/h，不铲雪时前进速度为 50 km/h。

保证：铲雪车从起点一定可以到达任何街道。

【输出格式】

铲掉所有街道上的雪并且返回出发点的最短时间，精确到分种。

【输入样例】snow.in

```
0 0
0 0 10000 10000
5000 -10000 5000 10000
5000 10000 10000 10000
```

【输出样例】snow.out

```
3:55
```

注解：3 小时 55 分钟

【算法分析】

把一条路拆分成两条路，每一个点都是偶点了，所以这个图一定存在一条欧拉回路，这不正是题目所要求的吗?

【参考程序】

```
program snow;
var   ppp,pp,h,m:longint;
      x1,y1,x2,y2,ans:real;
begin
  assign(input,'snow.in');   reset(input);
  assign(output,'snow.out');rewrite(output);
```

```
  readln(x1,y1);
  ans:=0;
  while not(eoln) do
    begin
      readln(x1,y1,x2,y2);
      ans:=ans+sqrt(sqr(x1-x2)+sqr(y1-y2)); //距离公式累加
    end;
  readln;
  ans:=ans*2/1000/20;                          //单位转化
  h:=trunc(ans); m:=round(60*(ans-h));
  if m=60 then begin m:=0; inc(h) end;
  write(h,':');
  if m<10 then write(0); writeln(m);
  close(output);   close(input);
end.
```

【上机练习】

1、输入一张无向图，指出该图中哪些顶点对之间有路。(题目见例 1，文件名 Ex1.pas)

2、输入一张顶点带权的无向图，分别计算含顶点数最多的一个连通分支和顶点的权之和最大的一个连通分支。(题目见例 2，文件名 Ex2.pas)

3、寻找一个图的欧拉路的算法实现。(题目见例 3，文件名 Euler.pas)

4、一笔画问题。(题目见例 4，文件名 one.pas)

5、铲雪车问题。(题目见例 5，文件名 snow.pas)

6、刻录光盘

【问题描述】

在 FJOI2009 夏令营快要结束的时候，很多营员提出来要把整个夏令营期间的资料刻录成一张光盘给大家，以便大家回去后继续学习。组委会觉得这个主意不错！可是组委会一时没有足够的空光盘，没法保证每个人都能拿到刻录上资料的光盘，又来不及去买了，怎么办呢？！

组委会把这个难题交给了 DYJ，DYJ 分析了一下所有营员的地域关系，发现有些营员是一个城市的，其实他们只需要一张就可以了，因为一个人拿到光盘后，其他人可以带着 U 盘之类的东西去拷贝啊！

可是，DYJ 调查后发现，有些营员并不是那么的合作，他们愿意某一些人到他那儿拷贝资料，当然也可能不愿意让另外一些人到他那儿拷贝资料，这与我们 JSOI 宣扬的团队合作精神格格不入!!!

现在假设总共有 N 个营员（2<=N<=200），每个营员的编号为 1~N。DYJ 给每个人发了一张调查表，让每个营员填上自己愿意让哪些人到他那儿拷贝资料。当然，如果 A 愿意把资料拷贝给 B，而 B 又愿意把资料拷贝给 C，则一旦 A 获得了资料，则 B，C 都会获得资料。

现在，请你编写一个程序，根据回收上来的调查表，帮助 DYJ 计算出组委会至少要刻录多少张光盘，才能保证所有营员回去后都能得到夏令营资料?

【输入格式】cdrom.in

先是一个数 N，接下来的 N 行，分别表示各个营员愿意把自己获得的资料拷贝给其他哪些营员。即输

入数据的第 i+1 行表示第 i 个营员愿意把资料拷贝给那些营员的编号，以一个 0 结束。如果一个营员不愿意拷贝资料给任何人，则相应的行只有 1 个 0，一行中的若干数之间用一个空格隔开。

【输出格式】cdrom.out

一个正整数，表示最少要刻录的光盘数。

【输入样例】	【输出样例】
5 2 4 3 0 4 5 0 0 0 1 0	1

7、骑马修栅栏

【问题描述】

农民 John 每年有很多栅栏要修理。他总是骑着马穿过每一个栅栏并修复它破损的地方。

John 是一个与其他农民一样懒的人。他讨厌骑马，因此从来不两次经过一个一个栅栏。你必须编一个程序，读入栅栏网络的描述，并计算出一条修栅栏的路径，使每个栅栏都恰好被经过一次。John 能从任何一个顶点(即两个栅栏的交点)开始骑马，在任意一个顶点结束。

每一个栅栏连接两个顶点，顶点用 1 到 500 标号(虽然有的农场并没有 500 个顶点)。一个顶点上可连接任意多(>=1)个栅栏。所有栅栏都是连通的(也就是你可以从任意一个栅栏到达另外的所有栅栏)。

你的程序必须输出骑马的路径(用路上依次经过的顶点号码表示)。我们如果把输出的路径看成是一个 500 进制的数，那么当存在多组解的情况下，输出 500 进制表示法中最小的一个 (也就是输出第一个数较小的，如果还有多组解，输出第二个数较小的，等等)。 输入数据保证至少有一个解。

【输入格式】fence.in

第 1 行: 一个整数 F(1 <= F <= 1024)，表示栅栏的数目

第 2 到 F+1 行: 每行两个整数 i, j(1 <= i,j <= 500)表示这条栅栏连接 i 与 j 号顶点。

【输出格式】fence.out

输出应当有 F+1 行，每行一个整数，依次表示路径经过的顶点号。注意数据可能有多组解，但是只有上面题目要求的那一组解是认为正确的。

【输入样例】	【输出样例】
9 1 2 2 3 3 4 4 2 4 5 2 5 5 6 5 7 4 6	1 2 3 4 2 5 4 6 5 7

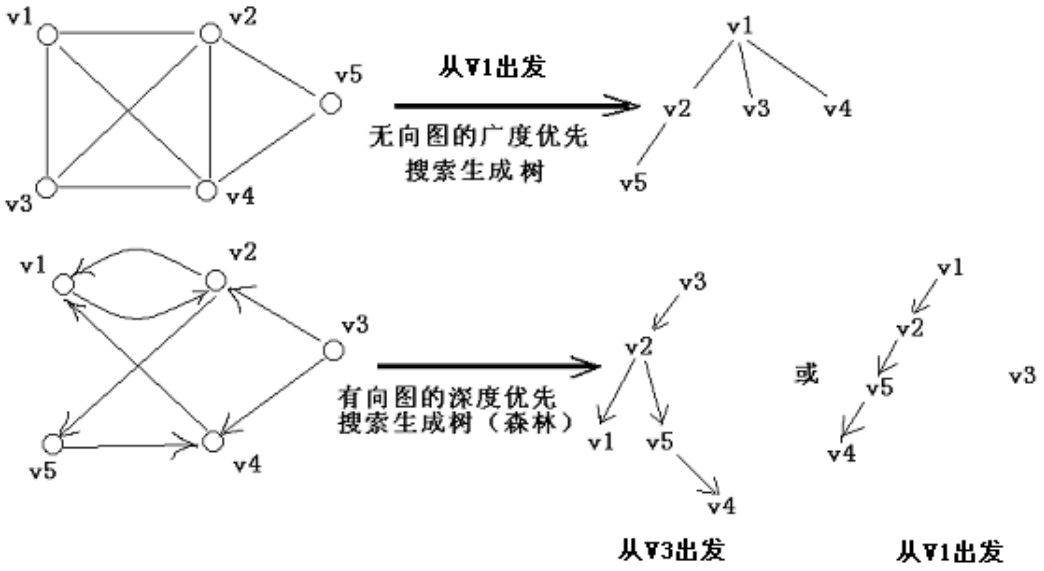
第四节　最小生成树算法

一、生成树的概念

若图是连通的无向图或强连通的有向图，则从其中任一个顶点出发调用一次 bfs 或 dfs 后便可以系统地访问图中所有顶点；若图是有根的有向图，则从根出发通过调用一次 dfs 或 bfs 亦可系统地访问所有顶点。在这种情况下，图中所有顶点加上遍历过程中经过的边所构成的子图称为原图的生成树。

对于不连通的无向图和不是强连通的有向图，若有根或者从根外的任意顶点出发，调用一次 bfs 或 dfs 后不能系统地访问所有顶点，而只能得到以出发点为根的连通分支（或强连通分支）的生成树。要访问其它顶点则还需要从没有访问过的顶点中找一个顶点作为起始点，再次调用 bfs 或 dfs，这样得到的是生成森林。

由此可以看出，一个图的生成树是不唯一的，不同的搜索方法可以得到不同的生成树，即使是同一种搜索方法，出发点不同亦可导致不同的生成树。如下图：



但不管如何，我们都可以证明：具有 n 个顶点的带权连通图，其对应的生成树有 n-1 条边。

二、求图的最小生成树算法

严格来说，如果图 G=（V，E）是一个连通的无向图，则把它的全部顶点 V 和一部分边 E’构成一个子图 G’，即 G’=（V， E’），且边集 E’能将图中所有顶点连通又不形成回路，则称子图 G’是图 G 的一棵生成树。

对于加权连通图，生成树的权即为生成树中所有边上的权值总和，权值最小的生成树称为图的最小生成树。

求图的最小生成树具有很高的实际应用价值，比如下面的这个例题。

例 1、城市公交网

[问题描述]

有一张城市地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权为在这两个城市之间修建高速公路的造价，研究后发现，这个地图有一个特点，即任一对城市都是连通的。现在的问题是，要修建若干高速公路把所有城市联系起来，问如何设计可使得工程的总造价最少。

【输入格式】

n（城市数，1<=n<=100），e（边数）

以下 e 行，每行 3 个数 i,j,w_{ij}，表示在城市 i,j 之间修建高速公路的造价。

【输出格式】

n-1 行，每行为两个城市的序号，表明这两个城市间建一条高速公路。

【输入样例】

5 8

1 2 2

2 5 9

5 4 7

4 1 10

1 3 12

4 3 6

5 3 3

2 3 8

【输出样例】

1　2

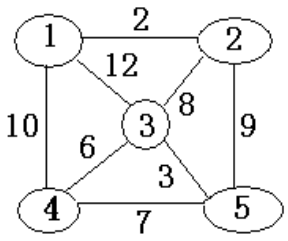
2　3

3　4

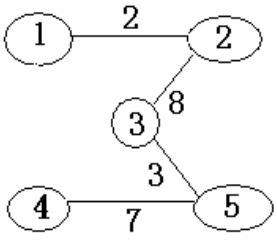
3　5

【算法分析】

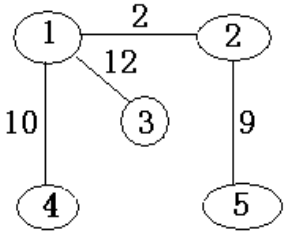
下面的图（A）表示一个 5 个城市的地图，图（B)、(C）是对图（A）分别进行深度优先遍历和广度优先遍历得到的一棵生成树，其权和分别为 20 和 33，前者比后者好一些，但并不是最小生成树，最小生成树的权和为 19。



图（A）



图（B）



图（C）

出发点：具有 n 个顶点的带权连通图，其对应的生成树有 n-1 条边。

那么选哪 n-1 条边呢？设图 G 的度为 n，G= (V，E)，我们介绍两种基于贪心的算法，Prim(普里姆)算法和 Kruskal(克鲁斯卡尔)算法。

1、用 Prim(普里姆)算法求最小生成树的思想如下：

①设置一个顶点的集合 S 和一个边的集合 TE，S 和 TE 的初始状态均为空集；

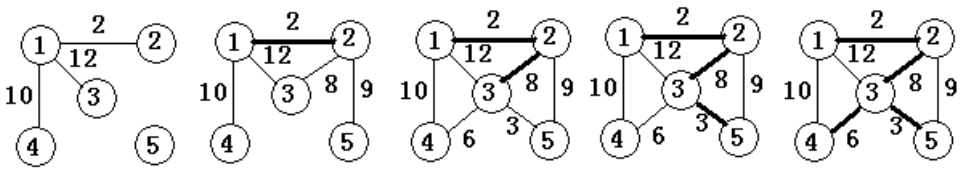
②选定图中的一个顶点 K，从 K 开始生成最小生成树，将 K 加入到集合 S；

③重复下列操作，直到选取了 n-1 条边：

选取一条权值最小的边（X，Y），其中 X∈S，not (Y∈S)；

将顶点 Y 加入集合 S，边（X，Y）加入集合 TE；

④得到最小生成树 T =（S，TE）



图(A) 图(B) 图(C) 图(D) 图(E)

上图是按照 Prim 算法，给出了例题中的图（A）最小生成树的生成过程（从顶点 1 开始）。其中图（E）中的 4 条粗线将 5 个顶点连通成了一棵最小生成树。Prim 算法的正确性可以通过反证法证明。

因为操作是沿着边进行的，所以数据结构采用邻接矩阵表示法，下面给出 Prim 算法构造图的最小生成树的具体算法框架。

①初始化；

```
readln(n,e);                          //顶点数 n 和边数 e
s:=[k];                               //将一个顶点 k 放入集合 S
fillchar(w,sizeof(w),$7f);
for i:=1 to e do                      //读入连通的边和权值
  begin
    readln(x,y,w[x,y]);
    w[y,x]:=w[x,y];
  end;
```

② 求出最小生成树的 n-1 条边；

```
for i:=1 to n-1 do                    //求出最小生成树的 n-1 条边
begin
  t:=maxlongint;
  for j:=1 to n do
    for k:=1 to n do                  //查找权值最小的一条边
      if (j in s) and not(k in s) and (w[j,k]<t) then
        begin                         //其中 X∈S，not (Y∈S)；
          t:=w[j,k];
          x:=j; y:=k;
        end;
    s:=s+[y];                         //y 为新加入的顶点
    te[x,y]:=true;                    //设定 x,y 边连通
```

```
            te[y,x]:=true;
        end;
```

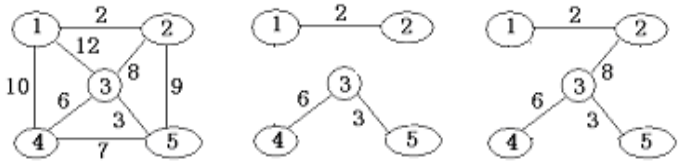
③ 输出；

```
    for i:=1 to n-1 do
      for j:=i+1 to n do
        if te[i,j] then writeln(i,'  ',j);
```

2、用 Kruskal(克鲁斯卡尔)算法求最小生成树的思想如下：

设最小生成树为 T=（V，TE），设置边的集合 TE 的初始状态为空集。将图 G 中的边按权值从小到大排好序，然后从小的开始依次选取，若选取的边使生成树 T 不形成回路，则把它并入 TE 中，保留作为 T 的一条边；若选取的边使生成树形成回路，则将其舍弃；如此进行下去，直到 TE 中包含 n-1 条边为止。最后的 T 即为最小生成树。如何证明呢?

下图是按照 Kruskal 算法给出了例题中图（A）最小生成树的生成过程：



Kruskal 算法在实现过程中的关键和难点在于：如何判断欲加入的一条边是否与生成树中已保留的边形成回路？我们可以将顶点划分到不同的集合中，每个集合中的顶点表示一个无回路的连通分量，很明显算法开始时，把所有 n 个顶点划分到 n 个集合中，每个集合只有一个顶点，表明顶点之间互不相通。当选取一条边时，若它的两个顶点分属于不同的集合，则表明此边连通了两个不同的连通分量，因每个连通分量无回路，所以连通后得到的连通分量仍不会产生回路，因此这条边应该保留，且把它们作为一个连通分量，即把它的两个顶点所在集合合并成一个集合。如果选取的一条边的两个顶点属于同一个集合，则此边应该舍弃，因为同一个集合中的顶点是连通无回路的，若再加入一条边则必然产生回路。

下面给出利用 Kruskal 算法构造图的最小生成树的具体算法框架。

① 初始化，并按照权值从小到大排好序；

```
  readln(n,e);                          //顶点数 n 和边数 e
  for i:=1 to e do                      //输入连通的边和权值
    readln(a[i],b[i],c[i]);
  for i:=1 to n do s[i]:=[i];           //S[i]都是集合，初始时 S[i]=[i]
  for i:=1 to e-1 do                    //按边的权值，从小到大排序
    for j:=i+1 to e do
      if c[i]>c[j] then
        begin  交换 a[i]和 a[j]，b[i]和 b[j]，c[i]和 c[j]的值  end;
```

② 建立最小生成树的所有边

```
  for i:=1 to n do                      //当选取一条边时，计算它的两个顶点所在的集合编号
    begin
      for j:=1 to n do
```

```
        begin
          if a[i] in s[j] then t:=j;
          if b[i] in s[j] then p:=j;
        end;
        if t<>p then                    //它的两个顶点分属于不同的集合，此边可连通
        begin
          inc(k);
          d[k]:=i;                      //保存此边起始点的编号
          s[t]:=s[t]+s[p];              //两个顶点所在集合合并成一个集合
          s[p]:=[];                     //另一集合置空
        end;
      end;
```

③ 输出最小生成树的各边：

```
  for i:=1 to k do
    writeln(a[d[i]],' ',b[d[i]]);
```

3、总结

以上两个算法的时间复杂度均为 O（n*n）。**参考程序见 Prim.pas 和 Kruskal.pas。**

三、应用举例

例 2、最优布线问题（wire.pas）

【问题描述】

学校有 n 台计算机，为了方便数据传输，现要将它们用数据线连接起来。两台计算机被连接是指它们时间有数据线连接。由于计算机所处的位置不同，因此不同的两台计算机的连接费用往往是不同的。

当然，如果将任意两台计算机都用数据线连接，费用将是相当庞大的。为了节省费用，我们采用数据的间接传输手段，即一台计算机可以间接的通过若干台计算机（作为中转）来实现与另一台计算机的连接。

现在由你负责连接这些计算机，任务是使任意两台计算机都连通（不管是直接的或间接的）。

【输入格式】

输入文件 wire.in，第一行为整数 n（2<=n<=100），表示计算机的数目。此后的 n 行，每行 n 个整数。第 x+1 行 y 列的整数表示直接连接第 x 台计算机和第 y 台计算机的费用。

【输出格式】

输出文件 wire.out，一个整数，表示最小的连接费用。

【输入样例】

```
3
0 1 2
1 0 1
2 1 0
```

【输出样例】

2（注：表示连接 1 和 2，2 和 3，费用为 2）

【算法分析】

本题是典型的求图的最小生成树问题，可以用 Prim 算法或者 Kruskal 算法求出，下面的程序在数据结构上对 Kruskal 算法做了一点修改，具体细节请看程序及注解。

【参考程序】

```
Program wire;
var g : Array [1..100, 1..100] Of Integer; //邻接矩阵
    l : Array [0..100] Of Integer;          //l[i]存放顶点i到当前已建成的生成树中，
                                            //任意一顶点j的权值g[i,j]的最小值
    u : Array [0..100] Of Boolean;          //u[i]=True，表示顶点i还未加入到生成树中；
                                            //u[i]=False，表示顶点i已加入到生成树中
    n, i, j, k, total : Integer;
Begin
  Assign(Input, 'wire.in');  Reset(Input);
  Assign(Output, 'wire.out'); Rewrite(Output);
  Readln(n);
  For i := 1 To n Do
    Begin
      For j := 1 To n Do Read(g[i，j]);
      Readln;
    End;
  Fillchar(l, sizeof(l), $7F);       //初始化为maxint
  l[1] := 0;                         //开始时生成树中只有第1个顶点
  Fillchar(u, sizeof(u), 1);         //初始化为True，表示所有顶点均未加入
  For i := 1 To n Do
    Begin
      k := 0;
      For j := 1 To n Do             //找一个未加入到生成树中的顶点，记为k，
                                     要求k到当前生成树中所有顶点的代价最小
        If u[j] And (l[j] < l[k])  Then k := j;
      u[k] := False;                 //顶点k加入生成树
      For j := 1 To n Do             //找到生成树中的顶点j，要求g[k,j]最小
        If u[j] And (g[k，j] < l[j]) Then l[j] := g[k，j];
    End;
  total := 0;
  For i := 1 To n Do  Inc(total, l[i]);   //累加
  Writeln(total);
  Close(Input); Close(Output);
End.
```

【上机练习】

1、用 Prim 和 Kruskal 两种算法分别完成例 1。

2、用自已编写 Prim 和 Kruskal 程序分别完成《最优布线问题》。

3、局域网

【问题描述】

某个局域网内有 n(n<=100)台计算机，由于搭建局域网时工作人员的疏忽，现在局域网内的连接形成了回路，我们知道如果局域网形成回路那么数据将不停的在回路内传输，造成网络卡的现象。因为连接计算机的网线本身不同，所以有一些连线不是很畅通，我们用 f(i,j)表示 i,j 之间连接的畅通程度(f(i,j)<=1000)，f(i,j)值越小表示 i,j 之间连接越通畅，f(i,j)为 0 表示 i,j 之间无网线连接。现在我们需要解决回路问题，我们将除去一些连线，使得网络中没有回路，并且被除去网线的Σf(i,j)最大，请求出这个最大值。

【输入格式】

第一行两个正整数 n k

接下来的 k 行每行三个正整数 i j m 表示 i,j 两台计算机之间有网线联通，通畅程度为 m

【输出格式】

一个正整数，Σf(i,j)的最大值

【输入输出样例】

【输入样例】		【输出样例】
5 5 1 2 8 1 3 1	1 5 3 2 4 5 3 4 2	8

4、繁忙的都市

【问题描述】

城市 C 是一个非常繁忙的大都市，城市中的道路十分的拥挤，于是市长决定对其中的道路进行改造。城市 C 的道路是这样分布的：城市中有 n 个交叉路口，有些交叉路口之间有道路相连，两个交叉路口之间最多有一条道路相连接。这些道路是双向的，**且把所有的交叉路口直接或间接的连接起来了**。每条道路都有一个分值，分值越小表示这个道路越繁忙，越需要进行改造。但是市政府的资金有限，市长希望进行改造的道路越少越好，于是他提出下面的要求：

1． 改造的那些道路能够把所有的交叉路口直接或间接的连通起来。

2． 在满足要求 1 的情况下，改造的道路尽量少。

3． 在满足要求 1、2 的情况下，改造的那些道路中分值最大的道路分值尽量小。

【编程任务】

作为市规划局的你，应当作出最佳的决策，选择那些道路应当被修建。

【输入格式】city.in

第一行有两个整数 n,m 表示城市有 n 个交叉路口，m 条道路。接下来 m 行是对每条道路的描述，u, v, c 表示交叉路口 u 和 v 之间有道路相连，分值为 c。(1≤n≤300，1≤c≤10000)

【输出格式】city.out

两个整数 s, max，表示你选出了几条道路，分值最大的那条道路的分值是多少。

【输入输出样例】

【输入样例】		【输出样例】
4 5 1 2 3 1 4 5 2 4 7	2 3 6 3 4 8	3 6

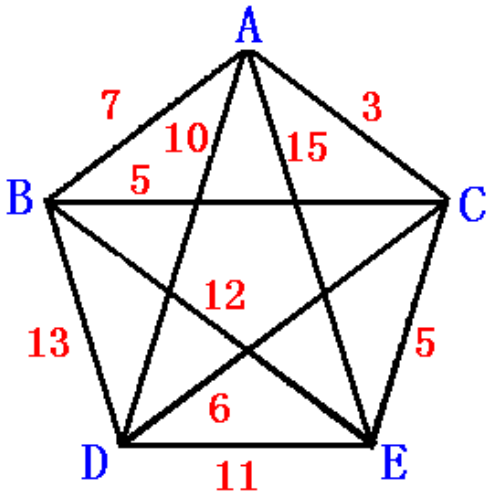
第五节　最短路径算法

最短路径是图论中的一个重要问题，具有很高的实用价值，也是信息学竞赛中常见的一类中等难度的题目，这类问题很能联系实际，考察学生的建模能力，反映出学生的创造性思维，因为有些看似跟最短路径毫无关系的问题也可以归结为最短路径问题来求解。本文就简要分析一下此类问题的模型、特点和常用算法。

在带权图 G=（V，E）中，若顶点 Vi,Vj 是图 G 的两个顶点，从顶点 Vi 到 Vj 的路径长度定义为路径上各条边的权值之和。从顶点 Vi 到 Vj 可能有多条路径，其中路径长度最小的一条路径称为顶点 Vi 到 Vj 的最短路径。一般有两类最短路径问题：一类是求从某个顶点（源点）到其它顶点（终点）的最短路径；另一类是求图中每一对顶点间的最短路径。

对于不带权的图，只要人为的把每条边加上权值 1，即可当作带权图一样处理了。

例 1、假设 A、B、C、D、E 各个城市之间旅费如下图所示。某人想从城市 A 出发游览各城市一遍，而所用旅费最少，试编程输出结果。



【问题分析】

解这类问题时，很多同学往往不得要领，采用穷举法把所有可能的情况全部列出，再找出其中旅费最少的那条路径；或者采用递归（深搜）找出所有路径，再找出旅费最少的那条。但这两种方法都是费时非常多的解法，如果城市数目多的话则很可能要超时了。

实际上我们知道，递归（深搜）之类的算法一般用于求所有解问题（例如求从 A 出发每个城市都要走一遍一共有哪几种走法？），所以这些算法对于求最短路径这类最优解问题显然是不合适的。

首先，对于这类图，我们都应该先建立一个邻接矩阵，存放任意两点间的数据（距离、费用、时间等），以便在程序中方便调用，上图的邻接矩阵如下：

```
const dis:array[1..5,1..5] of integer =( ( 0, 7, 3,10,15),
                                         ( 7, 0, 5,13,12),
                                         ( 3, 5, 0, 6, 5),
                                         (10,13, 6, 0,11),
                                         (15,12, 5,11, 0));
```

以下介绍几种常见的、更好的算法。

一、 宽度优先搜索

宽搜也并不是解决这类问题的优秀算法，这里只是简单介绍一下算法思路，为后面的优秀算法做个铺垫。具体如下：

1、从 A 点开始依次展开得到 AB、AC、AD、AE 四个新结点（第二层结点），当然每个新结点要记录下其旅费；

2、再次由 AB 展开得到 ABC、ABD、ABE 三个新结点（第三层结点），而由 AC 结点可展开得到 ACB、ACD、ACE 三个新结点，自然由 AD 可以展开得到 ADB、ADC、ADE，由 AE 可以展开得到 AEB、AEC、AED 等新结点，对于每个结点也须记录下其旅费；

3、再把第三层结点全部展开，得到所有的第四层结点：ABCD、ABCE、ABDC、ABDE、ABEC、ABED、……、AEDB、AEDC，每个结点也需记录下其旅费；

4、再把第四层结点全部展开，得到所有的第五层结点：ABCDE、ABCED、……、AEDBC、AEDCB，每个结点也需记录下其旅费；

5、到此，所有可能的结点均已展开，而第五层结点中旅费最少的那个就是题目的解了。

由上可见，这种算法也是把所有的可能路径都列出来，再从中找出旅费最少的那条，显而易见也是一种很费时的算法。

二、A＊算法

A＊算法是在宽度优先搜索算法的基础上，每次并不是把所有可展开的结点展开，而是对所有没有展开的结点，利用一个自己确定的估价函数对所有没展开的结点进行估价，从而找出最应该被展开的结点（也就是说我们要找的答案最有可能是从该结点展开），而把该结点展开，直到找到目标结点为止。

这种算法最关键的问题就是如何确定估价函数，估价函数越准，则能越快找到答案。A＊算法实现起来并不难，只不过难在找准估价函数，大家可以自己找相关资料学习 A*算法。

三、等代价搜索法

等代价搜索法也是在宽度优先搜索的基础上进行了部分优化的一种算法，它与 A＊算法的相似之处都是每次只展开某一个结点（不是展开所有结点），不同之处在于：它不需要去另找专门的估价函数，而是以该结点到 A 点的距离作为估价值，也就是说，等代价搜索法是 A＊算法的一种简化版本。它的大体思路是：

1、从 A 点开始依次展开得到 AB（7）、AC（3）、AD（10）、AE（15）四个新结点，把第一层结点 A 标记为已展开，并且每个新结点要记录下其旅费（括号中的数字）；

2、把未展开过的 AB、AC、AD、AE 四个结点中距离最小的一个展开，即展开 AC（3）结点，得到 ACB（8）、ACD（16）、ACE（13）三个结点，并把结点 AC 标记为已展开；

3、再从未展开的所有结点中找出距离最小的一个展开，即展开 AB（7）结点，得到 ABC（12）、ABD（20）、ABE（19）三个结点，并把结点 AB 标记为已展开；

4、再次从未展开的所有结点中找出距离最小的一个展开，即展开 ACB（8）结点，……；

5、每次展开所有未展开的结点中距离最小的那个结点，直到展开的新结点中出现目标情况（结点含有 5 个字母）时，即得到了结果。

由上可见，A＊算法和等代价搜索法并没有象宽度优先搜索一样展开所有结点，只是根据某一原则（或某一估价函数值）每次展开距离 A 点最近的那个结点（或是估价函数计算出的最可能的那个结点），反复下去即可最终得到答案。虽然中途有时也展开了一些并不是答案的结点，但这种展开并不是大规模的，不是

全部展开，因而耗时要比宽度优先搜索小得多。

例 2、题目基本同例 1，现在把权定义成距离，现在要求 A 点到 E 点的最短路径，但并不要求每个城市都要走一遍。

【问题分析】

既然不要求每个点都要走一遍，只要距离最短即可，那么普通的宽度优先搜索已经没有什么意义了，实际上就是穷举。那么等代价搜索能不能再用在这题上呢？答案是肯定的，但到底搜索到什么时候才能得到答案呢？这可是个很荆手的问题。

是不是搜索到一个结点是以 E 结束时就停止呢？显然不对。

那么是不是要把所有以 E 为结束的结点全部搜索出来呢？这简直就是宽度优先搜索了，显然不对。

实际上，应该是搜索到：当我们确定将要展开的某个结点（即所有未展开的结点中距离最小的那个点）的最后一个字母是 E 时，这个结点就是我们所要求的答案！因为比这个结点大的点再展开得到的解显然不可能比这个结点优！

那么，除了等代价搜索外，有没有其它办法了呢？下面就介绍这种求最短路径问题的其它几种成熟算法。

四、宽度优先搜索+剪枝

搜索之所以低效，是因为在搜索过程中存在着大量的重复和不必要的搜索。因此，提高搜索效率的关键在于减少无意义的搜索。假如在搜索时已经搜出从起点 A 到点 B 的某一条路径的长度是 X，那么我们就可以知道，从 A 到 B 的最短路径长度必定≤X，因此，其他从 A 到 B 的长度大于或等于 X 的路径可以一律剔除。具体实现时，可以开一个数组 h[1..n]，n 是结点总数，h[i]表示从起点到结点 i 的最短路径长度。算法流程如下：

1、 初始化：

将起点 start 入队，h[start]:=0，h[k]:=maxlongint(1<=k<=n，且 k≠start)。

2、repeat

取出队头结点赋给 t；

while t 有相邻的结点没被扩展 do

begin

t 扩展出新的结点 newp；

如果 h[t]+w[t,newp] <h[newp]，

则将 newp 入队，把 h[newp]的值更新为 h[t]+w[t,newp]；

end

until 队列空；

以上算法实现的程序如下：

```
const   maxn=100;
        maxint=maxlongint div 4;
        maxq=10000;
var     h:array[1..maxn] of longint;
        g:array[1..maxn,1..maxn] of longint;
        n,i,j:longint;
```

```
procedure bfs;
var head,tail,i,t:longint;
    q:array[1..maxq] of longint;
begin
      for i:=1 to n do h[i]:=maxint;
      h[1]:=0;
      q[1]:=1;
      head:=0;tail:=1;
      repeat
            head:=head+1;
            t:=q[head];
            for i:=1 to n do
                if (g[t,i]<>maxint)and(h[t]+g[t,i]<h[i]) then
                   begin
                         tail:=tail+1;
                         q[tail]:=i;
                         h[i]:=h[t]+g[t,i];
                   end;
      until head=tail;
end;

begin
      assign(input,'data.in');   reset(input);
      read(n);
      for i:=1 to n do
          for j:=1 to n do
          begin
               read(g[i,j]);
               if (g[i,j]<=0)and(i<>j) then g[i,j]:=maxint;
          end;

      bfs;

      for i:=2 to n do
          writeln('From 1 To ',i,' Weigh ',h[i]);
      close(input);
end.
```

五、迭代法

该算法的中心思想是：任意两点 i,j 间的最短距离（记为 Dij）会等于从 i 点出发到达 j 点的以任一点为中转点的所有可能的方案中，距离最短的一个。即：

Dij = min { Dij , Dik+Dkj }，1<=k<=n。

这样，我们就找到了一个类似动态规划的表达式，只不过这里我们不把它当作动态规划去处理，而是做一个二维数组用以存放任意两点间的最短距离，利用上述公式不断地对数组中的数据进行处理，直到各数据不再变化为止，这时即可得到 A 到 E 的最短路径。

算法流程如下：

D[i]表示从起点到 i 的最短路的长度，g 是邻接矩阵，s 表示起点；

1、D[i]:=g[s,i] (1<=i<=n);

2、repeat

c:=false; {用以判断某一步是否有某个 Dij 值被修改过}

for j:=1 to n do

for k:=1 to n do

if D[j]>D[k]+g[k,j] then

begin D[j]:= D[k]+g[k,j]; c:=true; end;

Until not c;

这种算法是产生这样一个过程：不断地求一个数字最短距离矩阵中的数据的值，而当所有数据都已经不能再变化时，就已经达到了目标的平衡状态，这时最短距离矩阵中的值就是对应的两点间的最短距离。

这个算法实现的程序如下：

```
const   maxn=100;
        maxint=maxlongint div 4;
var     D:array[1..maxn] of longint;
        g:array[1..maxn,1..maxn] of longint;
        n,i,j,k:longint;
        c:boolean;
begin
    assign(input,'data.in'); reset(input);
    read(n);
    for i:=1 to n do
        for j:=1 to n do
        begin
            read(g[i,j]);
            if (g[i,j]<=0)and(i<>j) then g[i,j]:=maxint;
        end;

    for i:=1 to n do D[i]:=g[1,i];
    repeat
        c:=false;
```

```
            for j:=1 to n do
                for k:=1 to n do                {k 是中转点}
                    if D[j]>D[k]+g[k,j] then
                        begin
                            D[j]:=D[k]+g[k,j];
                            c:=true;
                        end;
    until not c;

    for i:=2 to n do
        writeln('From 1 To ',i,' Weigh ',D[i]);
    close(input);
end.
```

六、动态规划

动态规划算法已经成为了许多难题的首选算法。某些最短路径问题也可以用动态规划来解决，通常这类最短路径问题所对应的图必须是有向无回路图。因为如果存在回路，动态规划的无后效性就无法满足。

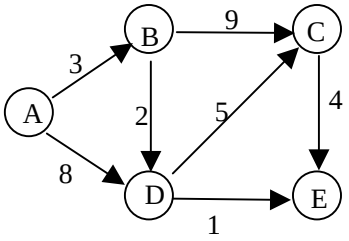
我们知道，动态规划算法与递归算法的不同之处在于它们的算法表达式：

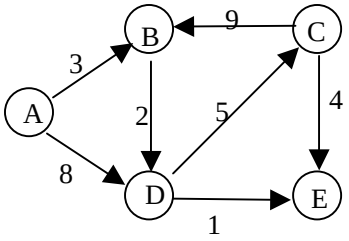
递归：类似 f(n)=x1*f(n-1)+x2*f(n-2)……，即可以找到一个确定的关系的表达式；

动态规划：类似 f(n)=min(f(n-1)+x1,f(n-2)+x2……)，即我们无法找到确定关系的表达式，只能找到这样一个不确定关系的表达式，f(n)的值是动态的，随着 f(n-1),f(n-2)等值的改变而确定跟谁相关。

为了给问题划分阶段，必须对图进行一次拓扑排序（见下一节内容），然后按照拓扑排序的结果来动态规划。

譬如，有如下两个有向图：





右图因为存在回路而不能用动态规划。而左图是无回路的，所以可以用动态规划解决。

对左图拓扑排序，得到的序列是 A、B、D、C、E。

设 F(E)表示从 A 到 E 的最短路径长度，然后按照拓扑序列的先后顺序进行动态规划：

F(A)=0

F(B)=min{ F(A) }+3=3

F(D)=min{ F(A)+8, F(B)+2 }=5

F(C)=min{ F(B)+9, F(D)+5 }=10

F(E)=min{ F(D)+1, F(C)+4 }=6

总的式子是：F(i)=min{ F(k)+dis(i,k) }，k 与 i 必须相连，且在拓扑序列中，k 在 i 之前。

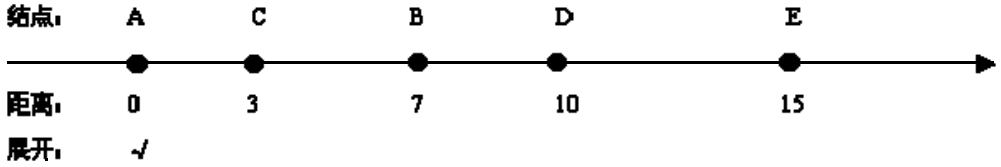
这个算法的参考程序如下：

```
const   maxn=100;
        maxint=maxlongint div 4;
var     g:array[1..maxn,1..maxn] of longint;   //有向图的邻接矩阵
        pre:array[1..maxn] of longint;          //pre[i]记录结点 i 的入度
        tp:array[1..maxn] of longint;           //拓扑排序得到的序列
        s:array[1..maxn] of longint;            //记录最短路径长度
        n,i,j,k:longint;

Begin
    assign(input,'data.in');reset(input);
    read(n);
    fillchar(pre,sizeof(pre),0);
    for i:=1 to n do
      for j:=1 to n do
        begin
          read(g[i,j]);
          if g[i,j]>0 then
              pre[j]:=pre[j]+1;            //如果存在一条有向边 i→j，就把 j 的入度加 1
        end;
    for i:=1 to n do                       //拓扑排序
      begin
        j:=1;
        while (pre[j]<>0) do j:=j+1;       //找入度为 0 的结点
        pre[j]:=-1;
        tp[i]:=j;
        for k:=1 to n do
            if g[j,k]>0 then   pre[k]:=pre[k]-1;
      end;
    filldword(s,sizeof(s)div 4,maxint);     //s 数组中的单元初始化为 maxint
    s[1]:=0;                                //默认起点是 1 号结点
    for i:=2 to n do                        //动态规划
      for j:=1 to i do
        if (g[tp[j],tp[i]]>0)and
          (s[tp[j]]+g[tp[j],tp[i]]<s[tp[i]])then
            s[tp[i]]:=s[tp[j]]+g[tp[j],tp[i]];
    for i:=2 to n do
        writeln('From 1 To ',i,' Weigh ',s[i]);
    close(input);
end.
```

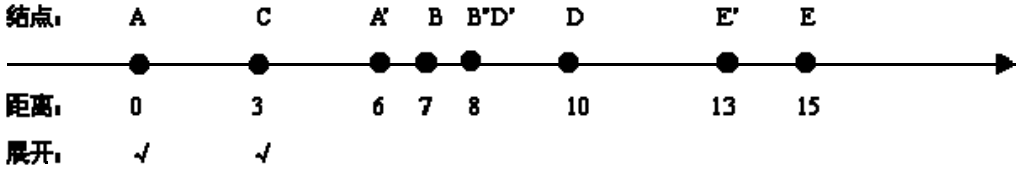
七、标号法

标号法是一种非常直观的求最短路径的算法，单从分析过程来看，我们可以用一个数轴简单地表示这种算法：

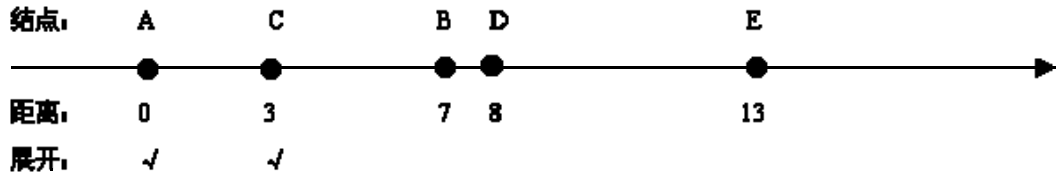
1、以 A 点为 0 点，展开与其相邻的点，并在数轴中标出。



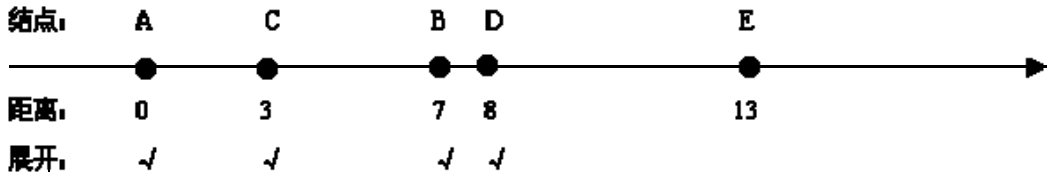
2、因为 C 点离起点 A 最近，因此可以断定 C 点一定是由 A 直接到 C 点这条路径是最短的（因为 A、C 两点间没有其它的点，所以 C 点可以确定是由 A 点直接到达为最短路径）。因而就可以以已经确定的 C 点为当前展开点，展开与 C 点想连的所有点 A'、B'、D'、E'。



3、由数轴可见，A 与 A'点相比，A 点离原点近，因而保留 A 点，删除 A'点，相应的，B、B'点保留 B 点，D、D'保留 D'，E、E'保留 E'，得到下图：



4、此时再以离原点最近的未展开的点 B 联接的所有点，处理后，再展开离原点最近未展开的 D 点，处理后得到如下图的最终结果：



5、由上图可以得出结论：点 C、B、D、E 就是点 A 到它们的最短路径（注意：这些路径并不是经过了所有点，而是只经过了其中的若干个点，而且到每一个点的那条路径不一定相同）。因而 A 到 E 的最短距离就是 13。至于它经过了哪几个点大家可在上述过程中加以记录即可。

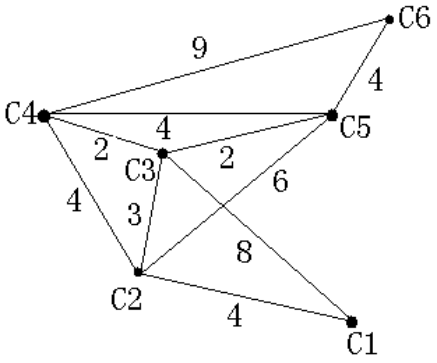
标号法的参考程序如下：

```
const   maxn=100;
        maxint=maxlongint div 4;
var     g:array[1..maxn,1..maxn] of longint; //邻接矩阵
        mark:array[1..maxn] of boolean;      //用来标志某个点是否已被扩展
        s:array[1..maxn] of longint;         //存储最短路径长度
```

```
    n,i,j,k:longint;

begin
    assign(input,'data.in');
    reset(input);
    read(n);
    for i:=1 to n do
        for j:=1 to n do
        begin
            read(g[i,j]);
            if (i<>j)and(g[i,j]=0) then g[i,j]:=maxint;
        end;

    fillchar(mark,sizeof(mark),false);      //mark 初始化为 false
    mark[1]:=true;                          //将起点标志为已扩展
    for i:=1 to n do s[i]:=g[1,i];          //s 数组初始化
    repeat
        k:=0;
        for j:=1 to n do                    //挑选离原点最近的点
            if (not mark[j])and((k=0)or(s[k]>s[j])) then
                k:=j;
        if k<>0 then
            begin
                mark[k]:=true;
                for j:=1 to n do            //扩展结点 k
                    if (not mark[j])and(s[k]+g[k,j]<s[j]) then
                        s[j]:=s[k]+g[k,j];
            end;
    until k=0;

    for i:=2 to n do
        writeln('From 1 To ',i,' Weigh ',s[i]);
    close(input);
end.
```

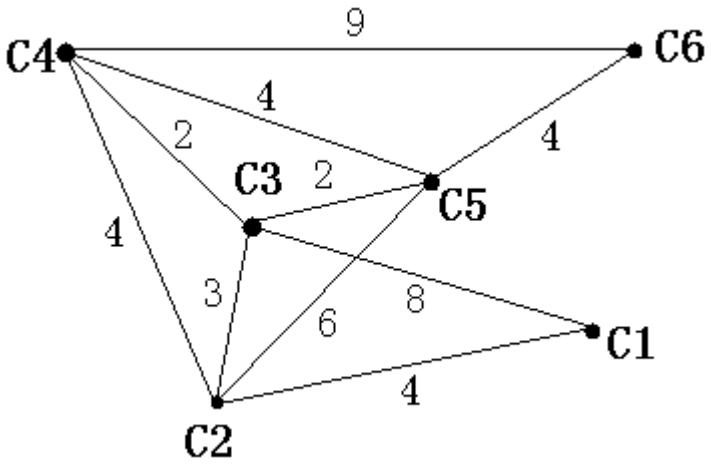
八、Dijkstra(迪杰斯特拉)算法（从一个顶点到其余各顶点的最短路径，单源最短路径）

例 3、如下图，假设 C_1,C_2,C_3,C_4,C_5,C_6 是六座城市，他们之间的连线表示两城市间有道路相通，连线旁的数字表示路程。请编写一程序，找出 C_1 到 C_i 的最短路径($2 \leqslant i \leqslant 6$)，输出路径序列及最短路径的路程长度。



【问题分析 1】

对于一个含有 n 个顶点和 e 条边的图来说，从某一个顶点 Vi 到其余任一顶点 Vj 的最短路径，可能是它们之间的边（Vi，Vj），也可能是经过 k 个中间顶点和 k+1 条边所形成的路径(1≤k≤n-2)。下面给出解



决这个问题的 Dijkstra 算法思想

。

设图 G 用邻接矩阵的方式存储在 GA 中，GA[i,j]=maxint 表示 Vi，Vj 是不关联的，否则为权值（大于 0 的实数)。设集合 S 用来保存已求得最短路径的终点序号，初始时 S=[Vi]表示只有源点，以后每求出一个终点 Vj，就把它加入到集合中并作为新考虑的中间顶点。设数组 dist[1..n]用来存储当前求得的最短路径，初始时 Vi，Vj 如果是关联的，则 dist[j]等于权值，否则等于 maxint，以后随着新考虑的中间顶点越来越多，dist[j]可能越来越小。再设一个与 dist 对应的数组 path[1..n]用来存放当前最短路径的边，初始时为 Vi 到 Vj 的边，如果不存在边则为空。

执行时，先从 S 以外的顶点（即待求出最短路径的终点）所对应的 dist 数组元素中，找出其值最小的元素（假设为 dist[m]），该元素值就是从源点 Vi 到终点 Vm 的最短路径长度，对应的 path[m]中的顶点或边的序列即为最短路径。接着把 Vm 并入集合 S 中，然后以 Vm 作为新考虑的中间顶点，对 S 以外的每个顶点 Vj，比较 dist[m]+GA[m,j]的 dist[j]的大小，若前者小，表明加入了新的中间顶点后可以得到更好的方案，即可求得更短的路径，则用它代替 dist[j]，同时把 Vj 或边（Vm，Vj）并入到 path[j]中。重复以上过程 n-2 次，即可在 dist 数组中得到从源点到其余各终点的最段路径长度，对应的 path 数组中保存着相应的最段路径。

对于上图，采用 Dijkstra 算法找出 C_1 到 C_i 之间的最短路径(2≤i≤6)的过程如下：

初始时：选择 m=1，初始与 C_1 点相联的结点，并赋值。

	1	2	3	4	5	6
Dist	0	4	8	maxint	maxint	maxint
Path	C_1	C_1,C_2	C_1,C_3			

第一次：选择 m=2，则 S=[C_1,C_2]，计算比较 dist[2]+GA[2,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	10	maxint
Path	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_5	

第二次：选择 m=3，则 S=[C_1,C_2,C_3]，计算比较 dist[3]+GA[3,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	9	maxint
Path	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	

第三次：选择 m=4，S=[C_1,C_2,C_3,C_4]，计算比较 dist[4]+GA[4,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	9	17
Path	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	C_1,C_2,C_4,C_6

第四次：选择 m=5，则 S=[C_1,C_2,C_3,C_4,C_5]，计算比较 dist[5]+GA[5,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	9	13
Path	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	C_1,C_2,C_3,C_5,C_6

因为该图的度 n=6，所以执行 n-2=4 次后结束，此时通过 dist 和 path 数组可以看出：

C_1 到 C_2 的最短路径为：C_1——C_2，长度为：4；

C_1 到 C_3 的最短路径为：C_1——C_2——C_3，长度为：7；

C_1 到 C_4 的最短路径为：C_1——C_2——C_4，长度为：8；

C_1 到 C_5 的最短路径为：C_1——C_2——C_3——C_5，长度为：9；

C_1 到 C_6 的最短路径为：C_1——C_2——C_3——C_5——C_6，长度为：13；

下面给出具体的 Dijkstra 算法框架（注：为了实现上的方便，我们用一个一维数组 s[1..n]代替集合 S，用来保存已求得最短路径的终点集合，即如果 s[j]=0 表示顶点 Vj 不在集合中，反之，s[j]=1 表示顶点 Vj 已在集合中）。

```
Procedure Dijkstra(GA,dist,path,i);//求 Vi 到各点的最短路径，参数 GA,dist,path 可省略，
        //为全量变量，GA 为邻接矩阵，dist 和 path 为变量型参数，其中 path 的基类型为集合。
  Begin
    For j:=1 To n Do Begin                    //初始化
                    If j<>i Then s[j]:=0
                            Else s[j]:=1;
                    dist[j]:=GA[i,j];
                    If dist[j]<maxint   //maxint 为假设的一个足够大的数
                      Then path[j]:=[i]+[j]   Else path[j]:=[ ];
                  End;
```

```
    For k:=1 To n-2 Do
      Begin
        min:=maxint；m:=i;
        For j:=1 To n Do                          //求出第 k 个终点 Vm
          If (s[j]=0) and (dist[j]<min) Then   Begin m:=j；min:=dist[j]；  End;
        If m<>i Then s[m]:=1 else exit;      //若条件成立，则把 Vm 加入到 S 中，否则退出循环，
                  因为剩余的终点，其最短路径长度均为 maxint，无需再计算下去
        For j:=1 To n Do                          //对 s[j]=0 的更优元素作必要修改
          If (s[j]=0) and (dist[m]+GA[m,j]<dist[j])
            Then Begin
                   Dist[j]:=dist[m]+GA[m,j];
                  path[j]:=path[m]+[j];
                End;
      End;
End;
```

【问题分析 2】

源		C2 点	C3 点	C4 点	C5 点	C6 点		各点前缀结点
c=1		dist[2]=4	dist[3]=8	dist[4]=∞	dist[5]=∞	dist[6]=∞		prev[2]=1,prev[3]=1
c=2		dist[2]=4	dist[3]=7	dist[4]=8	dist[5]=10	dist[6]=∞		prev[3]prev[4]prev[5]=2
c=3		dist[2]=4	dist[3]=7	dist[4]=8	dist[5]=9	dist[6]=∞		prev[5]=3
c=4		dist[2]=4	dist[3]=7	dist[4]=8	dist[5]=9	dist[6]=17		prev[6]=4
c=5		dist[2]=4	dist[3]=7	dist[4]=8	dist[5]=9	dist[6]=13		prev[6]=5

【运行结果】

```
1--6 : 13
6<--5<--3<--2<--1
```

【参考程序】

```
const
   n = 6;
cost : array[1..n,1..n] of integer = ((0,4,8,maxint,maxint,maxint),
                                      (4,0,3,4,6,maxint),
                                      (8,3,0,2,2,maxint),
                                      (maxint,4,2,0,4,9),
                                      (maxint,6,2,4,0,4),
                                      (maxint,maxint,maxint,9,4,0));
var
  dist : array[1..n] of integer;
  mark : array[1..n] of boolean;
  prev : array[1..n] of integer;
  i,v0 : integer;
```

```
procedure dijkstra;
var i,j,min,c: integer;
begin
  fillchar(mark,sizeof(mark),false);
  for i := 1 to n do dist[i] := maxint; //dist 初始化值为最大
  dist[v0] := 0;                        //源点到本身的距离为 0，v0 值可为变量形参数
  for i := 1 to n-1 do
  begin
    min := maxint;
    for j := 1 to n do                  //找距 v0 最短距离的点设为 c
      if not mark[j] and (dist[j]<min) then
        begin
          min := dist[j];
          c := j;
        end;
    mark[c] := true;
    for j := 1 to n do                  //重新计算起始点 v0 到各点的最短距离
      if not mark[j] and (dist[c]+cost[c,j]<dist[j]) then
      begin
        dist[j] := dist[c]+cost[c,j];   //更新 dist[j]的值，使 dist[j]的值最小
        prev[j] := c;                   //保存前继的结点号
      end;
  end;
end;

begin
  v0:=1;                                //源点的起始点，v0 可改为其他值
  write('Input i = ');                  //输入要到达的终点
  readln(i);
  dijkstra;
  writeln(v0,'--',i,' : ',dist[i]);     //输出最短路径的值
  while i<>v0 do                        //输出最短路径的所有结点
    begin
write(i,'<--');
i:=prev[i];
    end;
  writeln(v0);
  readln;
end.
```

```
{begin                                  //本程序段做为主程序，可求 v0 到所有点的最短路径
  write('v0=');                         //输入源点 v0 的编号
  readln(v0);
  dijkstra;
  for i := 1 to n do                    //通过循环求出 v0 到所有点 i 的最短路径
   if i <> v0 then                      //起判断作用，不输出 v0 到本身的最短路径
     if dist[i]<>maxint then writeln(v0,'--',i,':',dist[i]) //输出最短路径及值
                    else writeln(v0,'--',i,':Impossible!');    //表示该路径不通
  readln;
end.}
```

九、Floyed(弗洛伊德)算法

例 4、求任意一对顶点之间的最短路径。

【问题分析】

这个问题的解法有两种：一是分别以图中的每个顶点为源点共调用 n 次 Dijkstra 算法，这种算法的时间复杂度为 O（n^3）；另外还有一种算法：Floyed 算法，它的思路简单，但时间复杂度仍然为 O（n^3）。

下面介绍 Floyed 算法，设具有 n 个顶点的一个带权图 G 的邻接矩阵用 GA 表示，再设一个与 GA 同类型的表示每对顶点之间最短路径长度的二维数组 A，A 的初值等于 GA。Floyed 算法需要在 A 上进行 n 次运算，每次以 V_k（1≤k≤n）作为新考虑的中间点，求出每对顶点之间的当前最短路径长度，最后依次运算后，A 中的每个元素 A[i，j]就是图 G 中从顶点 V_i 到顶点 V_j 的最短路径长度。再设一个二维数组 P[1..n,1..n]，记录最短路径，其元素类型为集合类型 set of 1..n。

Floyed 算法的具体描述如下：

```
Procedure Floyed(GA，A，P);
  Begin
    For i:=1 To n Do                          //最短路径长度数组和最短路径数组初始化
      For j:=1 To n Do
        Begin
          A[i,j]:=GA[i,j];
          If A[i,j]<maxint Then p[i,j]:=[i]+[j]    Else p[i,j]:=[ ];
        End;
    For k:=1 To n Do                              //n 次运算
      For i:=1 To n Do
        For j:=1 To n Do
          Begin
            If (i=k)or(j=k)or(i=j) Then Continue;      //无需计算，直接进入下一轮循环
            If A[i,k]+A[k,j]<A[i,j] Then Begin         //找到更短路径、保存
                                          A[i,j]: = A[i,k]+A[k,j];
                                          P[i,j]: = P[i,k]+P[k,j];
                                      End;
          End;
```

End;

对于上图，大家可以运用 Floyed 算法，手工或编程试着找出任意一对顶点之间的最短路径及其长度。

十、总结与思考

最短路径问题的求解还不止这几种算法，比如分枝定界等等，而且可以创造出各种各样的新算法来。不同的最短路径问题到底用哪种算法，以及还需要对该种算法作什么改动，是非常重要的，这种能力往往是很多同学所欠缺的，这需要大家在平常的训练中多做这类题目，还要多总结，以达到熟能生巧的境界。

在学习完最短路径后，有没有人想到：能不能修改这些算法，实现求最长路径的问题呢？这种发散性的思维是值得称赞的，对于不存在回路的有向图，这种算法是可行的。但需要提醒的是：如果有向图出现了回路，按照最长路径的思想和判断要求，则计算可能沿着回路无限制的循环下去。如何判断一个有向图中是否存在回路？可以用 bfs 或 dfs 在搜的过程检查这个点是否在前面出现过；也可以用拓扑排序算法。

【上机练习】

1、Internet 消息发布

【问题描述】

设 Internet 上有 N 个站点，通常从一个站点发送消息给其他 N－1 个站点，需依次发送 N－1 次。这样从一个站点发布消息传遍 N 个站点时，可能要较长时间。而当一个站点发布消息给另一个站点后，已获得消息的这两个站点就可以发布消息给另外两个站点，此后就有四个站点可以同时发布消息，这种发布消息方法应该会缩短消息传遍 N 个站点的时间。

请您编一个程序，设从每一个站点都可以向其他 N－1 个站点同时发送消息，编程求出从第一个站点开始发布消息传遍 N 个站点的最短时间。

【输入格式】

由文件 internet.in 输入数据，文件的第一行是 Internet 上的站点数 N（1<=N<=100），第二行起是邻接矩阵严格的下三角部分，各行是整数或字符 X。A(I, J)表示从 I 站点发送消息给 J 站点所需要的时间。假设网络是无方向的，故 A(I, J)＝A(J, I)，当 A(I, J)＝-1 时，表示从站点 I 不能直接向站点 J 发送消息；A(I,I)＝0 表示没有必要自己给自己送消息（1<=I<=N），严格的下三角阵表示如下：

A(2, 1)

A(3, 1), A(3, 2)

A(4, 1), A(4, 2), A(4, 3)

⋮

A(N, 1), A(N, 2)……A(N, N-1)

【输出格式】

输出只有一行，它是一个非负整数，若为 0 表示无解，非 0 表示合符题意的最小整数。

【输入输出样例】

输入样例：	输出样例：
5 50 30　　5 100　20　50 10　-1　-1　10	35

2、最短路径问题

【问题描述】

平面上有 n 个点（n<=100），每个点的坐标均在-10000~10000 之间。其中的一些点之间有连线。若有连线，则表示可从一个点到达另一个点，即两点间有通路，通路的距离为两点间的直线距离。现在的任务是找出从一点到另一点之间的最短路径。

【输入格式】

输入文件为 short.in，共 n+m+3 行，其中:

第一行为整数 n。

第 2 行到第 n+1 行（共 n 行），每行两个整数 x 和 y，描述了一个点的坐标。

第 n+2 行为一个整数 m，表示图中连线的个数。

此后的 m 行，每行描述一条连线，由两个整数 i 和 j 组成，表示第 i 个点和第 j 个点之间有连线。

最后一行：两个整数 s 和 t，分别表示源点和目标点。

【输出格式】

输出文件为 short.out，仅一行，一个实数（保留两位小数），表示从 s 到 t 的最短路径长度。

【输入样例】

5
0 0
2 0
2 2
0 2
3 1
5
1 2
1 3
1 4
2 5
3 5
1 5

【输出样例】

3.41

3、最优乘车

【问题描述】

H 城是一个旅游胜地，每年都有成千上万的人前来观光。为方便游客，巴士公司在各个旅游景点及宾馆，饭店等地都设置了巴士站并开通了一些单程巴上线路。每条单程巴士线路从某个巴士站出发，依次途经若干个巴士站，最终到达终点巴士站。

一名旅客最近到 H 城旅游，他很想去 S 公园游玩，但如果从他所在的饭店没有一路已士可以直接到达 S 公园，则他可能要先乘某一路巴士坐几站，再下来换乘同一站台的另一路巴士，这样换乘几次后到达 S 公园。

现在用整数 1,2,…N 给 H 城的所有的巴士站编号，约定这名旅客所在饭店的巴士站编号为 1，S 公园巴士站的编号为 N。

写一个程序，帮助这名旅客寻找一个最优乘车方案,使他在从饭店乘车到 S 公园的过程中换车的次数最少。

【输入格式】

输入文件是 Travel.in。文件的第一行有两个数字 M 和 N(1<=M<=100 1<N<=500)，表示开通了 M 条单程巴士线路，总共有 N 个车站。从第二行到第 M 刊行依次给出了第 1 条到第 M 条巴士线路的信息。其中第 i+1 行给出的是第 i 条巴士线路的信息，从左至右按运行顺序依次给出了该线路上的所有站号相邻两个站号之间用一个空格隔开。

【输出格式】

输出文件是 Travel.out，文件只有一行。如果无法乘巴士从饭店到达 S 公园，则输出"N0"，否则输出你的程序所找到的最少换车次数，换车次数为 0 表示不需换车即可到达。

【输入样例】Travel.in

3 7
6 7
4 7 3 6
2 1 3 5

【输出样例】Travel.out

2

4、珍珠

【问题描述】

有 n 颗形状和大小都一致的珍珠，它们的重量都不相同。n 为整数，所有的珍珠从 1 到 n 编号。你的任务是发现哪颗珍珠的重量刚好处于正中间，即在所有珍珠的重量中，该珍珠的重量列(n+1)/2 位。下面给出将一对珍珠进行比较的办法：

给你一架天平用来比较珍珠的重量，我们可以比出两个珍珠哪个更重一些，在作出一系列的比较后，我们可以将某些肯定不具备中间重量的珍珠拿走。

例如，下列给出对 5 颗珍珠进行四次比较的情况：

1、 珍珠 2 比珍珠 1 重
2、 珍珠 4 比珍珠 3 重
3、 珍珠 5 比珍珠 1 重
4、 珍珠 4 比珍珠 2 重

根据以上结果，虽然我们不能精确地找出哪个珍珠具有中间重量，但我们可以肯定珍珠 1 和珍珠 4 不可能具有中间重量，因为珍珠 2、4、5 比珍珠 1 重，而珍珠 1、2、3 比珍珠 4 轻，所以我们可以移走这两颗珍珠。

写一个程序统计出共有多少颗珍珠肯定不会是中间重量。

【输入格式】

输入文件第一行包含两个用空格隔开的整数 N 和 M，其中 1<=N<=99，且 N 为奇数，M 表示对珍珠进行的比较次数，接下来的 M 行每行包含两个用空格隔开的整数 x 和 y，表示珍珠 x 比珍珠 y 重。

【输出格式】

输出文件仅一行包含一个整数，表示不可能是中间重量的珍珠的总数。

【输入样例】BEAD.IN

5 4
2 1
4 3
5 1
4 2

【输出样例】BEAD.OUT

2

5、信使

【问题描述】

战争时期，前线有 n 个哨所，每个哨所可能会与其他若干个哨所之间有通信联系。信使负责在哨所之间传递信息，当然，这是要花费一定时间的（以天为单位）。指挥部设在第一个哨所。当指挥部下达一个命令后，指挥部就派出若干个信使向与指挥部相连的哨所送信。当一个哨所接到信后，这个哨所内的信使们也以同样的方式向其他哨所送信。直至所有 n 个哨所全部接到命令后，送信才算成功。因为准备充足，每个哨所内都安排了足够的信使（如果一个哨所与其他 k 个哨所有通信联系的话，这个哨所内至少会配备 k 个信使）。

现在总指挥请你编一个程序，计算出完成整个送信过程最短需要多少时间。

【输入格式】

输入文件 msner.in，第 1 行有两个整数 n 和 m，中间用 1 个空格隔开，分别表示有 n 个哨所和 m 条通信线路。1<=n<=100。

第 2 至 m+1 行：每行三个整数 i、j、k，中间用 1 个空格隔开，表示第 i 个和第 j 个哨所之间存在通信线路，且这条线路要花费 k 天。

【输出格式】

输出文件 msner.out，仅一个整数，表示完成整个送信过程的最短时间。如果不是所有的哨所都能收到信，就输出-1。

【输入样例】

4 4
1 2 4
2 3 7
2 4 1
3 4 6

【输出样例】

11

第六节 拓扑排序算法

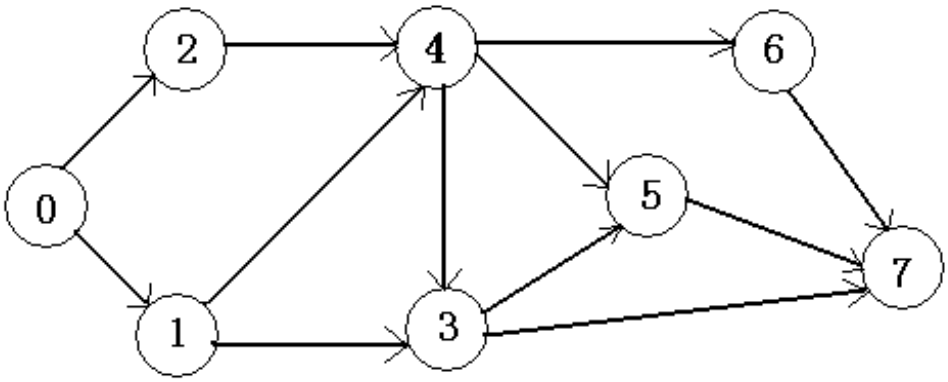
在日常生活中，一项大的工程可以看作是由若干个子工程（这些子工程称为“活动”）组成的集合，这些子工程（活动）之间必定存在一些先后关系，即某些子工程（活动）必须在其它一些子工程（活动）完成之后才能开始，我们可以用有向图来形象地表示这些子工程（活动）之间的先后关系，子工程（活动）为顶点，子工程（活动）之间的先后关系为有向边，这种有向图称为“顶点活动网络”，又称“AOV 网”。

在 AOV 网中，有向边代表子工程（活动）的先后关系，即有向边的起点活动是终点活动的前驱活动，只有当起点活动完成之后终点活动才能进行。如果有一条从顶点 Vi 到 Vj 的路径，则说 Vi 是 Vj 的前驱，Vj 是 Vi 的后继。如果有弧<Vi，Vj>，则称 Vi 是 Vj 的直接前驱，Vj 是 Vi 的直接后继。

一个 AOV 网应该是一个有向无环图，即不应该带有回路，否则必定会有一些活动互相牵制，造成环中的活动都无法进行。

把不带回路的 AOV 网中的所有活动排成一个线性序列，使得每个活动的所有前驱活动都排在该活动的前面，这个过程称为“拓扑排序”，所得到的活动序列称为“拓扑序列”。

需要注意的是 AOV 网的拓扑序列是不唯一的，如对下图进行拓扑排序至少可以得到如下几种拓扑序列：02143567、01243657、02143657、01243567。



在上图所示的 AOV 网中，工程 1 和过程 2 显然可以同时进行，先后无所谓；但工程 4 却要等工程 1 和工程 2 都完成以后才可进行；工程 3 要等到工程 1 和工程 4 完成以后才可进行；工程 5 又要等到工程 3、工程 4 完成以后才可进行；工程 6 则要等到工程 4 完成后才能进行；工程 7 要等到工程 3、工程 5、过程 6 都完成后才能进行。可见由 AOV 网构造拓扑序列具有很高的实际应用价值。

其实，构造拓扑序列的拓扑排序算法思想很简单：**只要选择一个入度为 0 的顶点并输出，然后从 AOV 网中删除此顶点及以此顶点为起点的所有关联边；重复上述两步，直到不存在入度为 0 的顶点为止，若输出的顶点数小于 AOV 网中的顶点数，则输出“有回路信息”，否则输出的顶点序列就是一种拓扑序列。**

对上图进行拓扑排序的过程如下：

1、 选择顶点 0（唯一），　　输出 0，　删除边<0，1>，<0，2>；
2、 选择顶点 1（不唯一，可选顶点 2），输出 1，　删除边<1，3>，<1，4>；
3、 选择顶点 2（唯一），　　输出 2，　删除边<2，4>；
4、 选择顶点 4（唯一），　　输出 4，　删除边<4，3>，<4，5>，<4,6>；
5、 选择顶点 3（不唯一，可选顶点 6），输出 3，　删除边<3，5>，<3，7>；
6、 选择顶点 5（不唯一，可选顶点 6），输出 5，　删除边<5，7>；

7、 选择顶点 6（唯一），　　输出 6，　删除边<6，7>；

8、 选择顶点 7（唯一），　　输出 7，　结束。

输出的顶点数 m=8，与 AOV 网中的顶点数 n=8 相等，所以输出一种拓扑序列：01243567。

为了算法实现上的方便，我们采用邻接表存储 AOV 网，不过稍做修改，在顶点表中增加一个记录顶点入度的域 id，**具体的拓扑排序算法描述如下：**

```
Procedure TopSort(dig:graphlist);          //用邻接表 dig 存储图 G
Var n,top,i,j,k,m: Integer;
    P:graphlist;
Begin
  n:=dig.adjv;                             //取顶点总个数
  top:=0;                                  //堆栈、初始化
  For i:=1 To n Do                         //对入度为 0 的所有顶点进行访问，序号压栈
    If dig.adj[i].id = 0 Then Begin
                                 dig.adj[i].id:=top;
                                 top:=i;
                               End;
  m:=0;                                    //记录输出的顶点个数
  While top<>0  Do                         //栈不空
    Begin
      j:=top;                              //取一个入度为 0 的顶点序号
      top:=dig.adj[top].id;                //出栈、删除当前处理的顶点、指向下个入度为 0 的顶点
      Write(dig.adj[top].v);               //输出顶点序号
      m:=m+1;
      p:=dig.adj[j].link;                  //指向 Vj 邻接表的第一个邻接点
      While p<>nil  Do                     //删除所有与 Vj 相关边
        Begin
          k:=p^.adjv;                      //下一个邻接点
          dig.adj[k].id:= dig.adj[k].id - 1;  //修正相应点的入度
          If dig.adj[k].id = 0 Then Begin  //入度为 0 的顶点入栈
                                      dig.adj[k].id:=top;
                                      top:=k;
                                    End;
          p:=p^.next;                      //沿边表找下一个邻接点
        End;
    End;
    If m<n Then Writeln('no solution!'); //有回路
End;
```

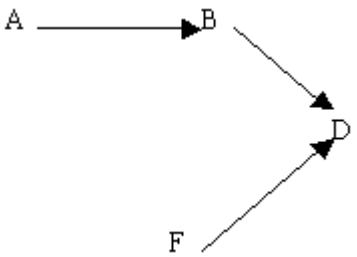
思考：拓扑排序一般用在哪些场合呢？

解答：如判回路或图的动态规划过程中分阶段。

例 1、士兵排队（soldier）

【问题描述】

有 n 个士兵（1≤n≤26），编号依次为 A、B、C，…… 队列训练时，指挥官要把一些士兵从高到矮依次排成一行。但现在指挥官不能直接获得每个人的身高信息，只能获得“p1 比 p2 高”这样的比较结果（p1，p2∈{'A'，…，'Z'}），记为 p1>p2。例如 A>B，B>D，F>D。士兵的身高关系如图所示：



对应的排队方案有三个：AFBD、FABD、ABFD。

【输入格式】soldier.in

第一行：一个整数 k

第二至第 k+1 行：每行两个大写字母（中间和末尾都没有空格），代表两个士兵，且第一个士兵高度大于第二个士兵。

【输出格式】soldier.out

一个只包含大写字母的字符序列，表示排队方案（只要一种方案即可）。

【输入样例】

3
AB
BD
FD

【输出样例】

AFBD

【算法分析】

士兵的身高关系对应一张有向图，图中的顶点对应一个士兵，有向边<v_i，v_j>表示士兵 i 高于士兵 j。我们按照从高到矮将士兵排出一个线形的顺序关系，即为对有向图的顶点进行拓扑排序。

【参考程序】

```
program soldier;
var   g:array['A'..'Z','A'..'Z'] of 0..1; //图的邻接矩阵
      d:array['A'..'Z'] of longint;          //记录各顶点的入度
      s:array['A'..'Z'] of boolean;          //用来记录出现过的士兵名
      ans:string;
      ch,i:char;
```

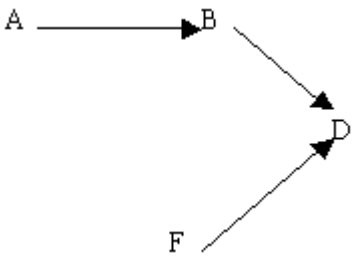
```
procedure readata;                          //读入，构图
var i,j,k:longint;
    a,b:char;
begin
    fillchar(g,sizeof(g),0);                //g、d、s 初始化
    fillchar(d,sizeof(d),0);
    fillchar(s,sizeof(s),false);
    readln(k);                              //读入边的条数
    for i:=1 to k do
    begin
        readln(a,b);
        g[a,b]:=1;                          //构造有向边 a→b
        d[b]:=d[b]+1;                       //将 b 的入度加 1
        s[a]:=true;                         //将 a、b 标记为出现过
        s[b]:=true;
    end;
end;

begin  {main}
    assign(input,'soldier.in');  reset(input);
    assign(output,'soldier.out');rewrite(output);
    readata;
    ans:='';                                //拓扑排序
    repeat
        ch:='A';
        while (ch<='Z') and not( (s[ch])and(d[ch]=0) ) do ch:=chr(ord(ch)+1);
        if ch<='Z' then
          begin
            s[ch]:=false;
            ans:=ans+ch;
            for i:='A' to 'Z' do
              if (s[i])and(g[ch,i]=1) then d[i]:=d[i]-1;
          end;
    until ch>'Z';
    writeln(ans);
    close(input);close(output);
end.
```

【上机练习】

1、士兵排队（soldier）

【问题描述】

有 n 个士兵（1≤n≤26），编号依次为 A、B、C，…… 队列训练时，指挥官要把一些士兵从高到矮依次排成一行。但现在指挥官不能直接获得每个人的身高信息，只能获得“p1 比 p2 高”这样的比较结果（p1，p2∈{'A'，…，'Z'}），记为 p1>p2。例如 A>B，B>D，F>D。士兵的身高关系如图所示：



对应的排队方案有三个：AFBD、FABD、ABFD。

【输入格式】soldier.in

第一行：一个整数 k

第二至第 k+1 行：每行两个大写字母（中间和末尾都没有空格），代表两个士兵，且第一个士兵高度大于第二个士兵。

【输出格式】soldier.out

一个只包含大写字母的字符序列，表示排队方案（只要一种方案即可）。

【输入样例】

3

AB

BD

FD

【输出样例】

AFBD

2、病毒

【问题描述】

有一天，小 y 突然发现自己的计算机感染了一种病毒！还好，小 y 发现这种病毒很弱，只是会把文档中的所有字母替换成其它字母，但并不改变顺序，也不会增加和删除字母。

现在怎么恢复原来的文档呢！小 y 很聪明，他在其他没有感染病毒的机器上，生成了一个由若干单词构成的字典，字典中的单词是按照字母顺序排列的，他把这个文件拷贝到自己的机器里，故意让它感染上病毒，他想利用这个字典文件原来的有序性，找到病毒替换字母的规律，再用来恢复其它文档。

现在你的任务是：告诉你被病毒感染了的字典，要你恢复一个字母串。

【输入格式】virus.in

第一行为整数 K（≤50000），表示字典中的单词个数。

以下 K 行，是被病毒感染了的字典，每行一个单词。

最后一行是需要你恢复的一串字母。

所有字母均为小写。

【输出格式】virus.out

输出仅一行，为恢复后的一串字母。当然也有可能出现字典不完整、甚至字典是错的情况，这时请输出一个 0。

【输入样例】

6

cebdbac

cac

ecd

dca

aba

bac

cedab

【输出样例】

abcde

3、家谱树

【问题描述】

有个人的家族很大，辈分关系很混乱，请你帮整理一下这种关系。

【编程任务】

给出每个人的孩子的信息。

输出一个序列，使得每个人的后辈都比那个人后列出。

【输入格式】gentree.in

第 1 行一个整数 N（1<=N<=100），表示家族的人数。

接下来 N 行，第 I 行描述第 I 个人的儿子。

每行最后是 0 表示描述完毕。

【输出格式】gentree.out

输出一个序列，使得每个人的后辈都比那个人后列出。

如果有多解输出任意一解。

【输入样例】

5

0

4 5 1 0

1 0

5 3 0

3 0

【输出样例】

2 4 5 3 1

4、烦人的幻灯片

【问题描述】

李教授将于今天下午作一次非常重要的演讲。不信的事他不是一个非常爱整洁的人，他把自己演讲要用的幻灯片随便堆在了一起。因此，演讲之前他不得不去整理这些幻灯片。作为一个讲求效率的学者，他希望尽可能简单地完成它。教授这次演讲一共要用 n 张幻灯片（n<=26），这 n 张幻灯片按照演讲要使用的顺序已经用数字 1~n 编了号。因为幻灯片是透明的，所以我们不能一下子看清每一个数字所对应的幻灯片。

现在我们用大写字母 A,B,C……再次把幻灯片依次编号。你的任务是编写一个程序，把幻灯片的数字编号和字母编号对应起来，显然这种对应应该是唯一的；若出现多种对应的情况或是某些数字编号和字母编号对应不起来，我们称对应是无法实现的。

【输入格式】slides.in

文件的第一行只有一个整数 n，表示有 n 张幻灯片，接下来的 n 行每行包括 4 个整数 xmin,xmax,ymin,ymax（整数之间用空格分开）为幻灯片的坐标，这 n 张幻灯片按其在文件中出现的顺序从前到后依次编号为 A,B,C……，再接下来的 n 行依次为 n 个数字编号的坐标 x,y，显然在幻灯片之外是不会有数字的。

【输出格式】slides.out

若是对应可以实现，输出文件应该包括 n 行，每一行为一个字母和一个数字，中间以一个空格隔开，并且每行以字母的生序排列，注意输出的字母要大写并且定格；反之，若是对应无法实现，在文件的第一行顶格输出 None 即可。首行末无多余的空格。

【输入样例】

```
4
6 22 10 20
4 18 6 16
8 20 2 18
10 24 4 8
9 15
19 17
11 7
21 11
```

【输出样例】

```
A 4
B 1
C 2
D 3
```

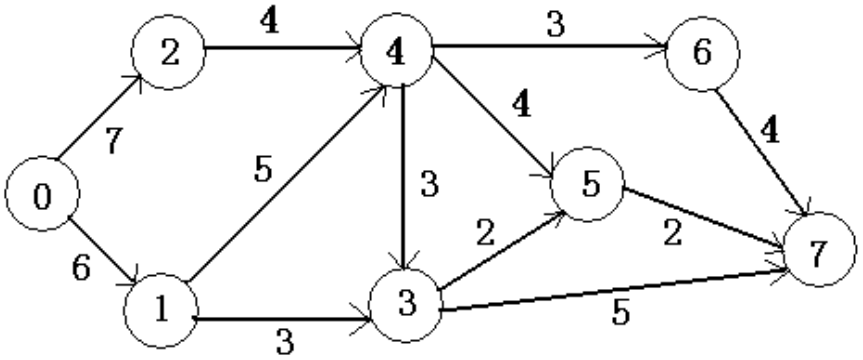
第七节　关键路径算法

利用 AOV 网络，对其进行拓扑排序能对工程中活动的先后顺序作出安排。但一个活动的完成总需要一定的时间，为了能估算出某个活动的开始时间，找出那些影响工程完成时间的最主要的活动，我们可以利用带权的有向网，图中的边表示活动，边上的权表示完成该活动所需要的时间，一条边的两个顶点分别表示活动的开始事件和结束事件，这种用边表示活动的网络，称为“AOE 网”。

其中，有两个特殊的顶点（事件），分别称为源点和汇点，源点表示整个工程的开始，通常令第一个事件（事件 1）作为源点，它只有出边没有入边；汇点表示整个工程的结束，通常令最后一个事件（事件 n）作为汇点，它只有入边没有出边；其余事件的编号为 2 到 n-1。

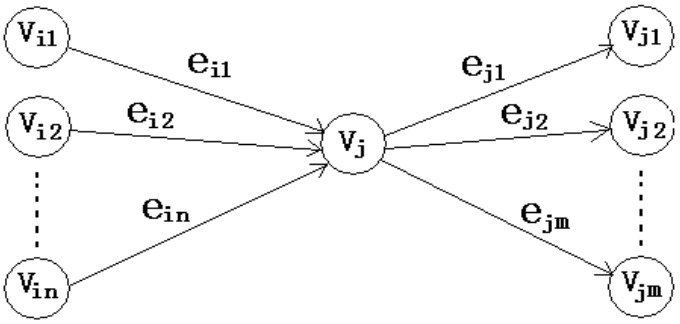
在实际应用中，AOE 网应该是没有回路的，并且存在唯一的入度为 0 的源点和唯一的出度为 0 的汇点。

下图表示一个具有 12 个活动的 AOE 网。图中有 8 个顶点，分别表示事件 0 到 7，其中，0 表示开始事件，7 表示结束事件，边上的权表示完成该活动所需要的时间。



AOE 网络要研究的问题是完成整个工程至少需要多少时间？哪些活动是影响工程进度的关键？

下面先讨论一个事件的最早发生时间和一个活动的最早开始时间。如下图，事件 V_j 必须在它的所有入边活动 e_{ik}（1≤k≤n）都完成后才能发生。活动 e_{ik}（1≤k≤n）的最早开始时间是与它对应的起点事件 V_{ik} 的最早发生时间。所有以事件 V_j 为起点事件的出边活动 e_{jk}（1≤k≤m）的最早开始时间都等于事件 V_j 的最早发生时间。所以，我们可以从源点出发按照上述方法，求出所有事件的最早发生时间。



设数组 earliest[1..n]表示所有事件的最早发生时间，则我们可以按照拓扑顺序依次计算出 earliest[k]：

earliest[1]=0

earliest[k]=max{earliest[j]+dut[j,k]}

（其中，事件 j 是事件 k 的直接前驱事件，dut[j,k]表示边<j，k>上的权）

对于上图，用上述方法求 earliest[0..7]的过程如下：

earliest[0]=0

earliest[1]=earliest[0]+dut[0,1]=0+6=6

earliest[2]=earliest[0]+dut[0,2]=0+7=7

earliest[4]=max{earliest[1]+dut[1,4]，earliest[2]+dut[2,4]}

=max{6+5,7+4}

=11

earliest[3]=max{earliest[1]+dut[1,3]，earliest[4]+dut[4,3]}

=max{6+3,11+3}

=14

earliest[5]=max{earliest[3]+dut[3,5]，earliest[4]+dut[4,5]}

=max{14+2,11+4}

=16

earliest[6]=earliest[4]+dut[4,6]=11+3=14

earliest[7]=max{earliest[3]+dut[3,7]，earliest[5]+dut[5,7], earliest[6]+dut[6,7]}

=max{14+5,16+2,14+4}

=19

最后得到的 earliest[7]就是汇点的最早发生时间，从而可知整个工程至少需要 19 天完成。

但是，在不影响整个工程按时完成的前提下，一些事件可以不在最早发生时间发生，而向后推迟一段时间，我们把事件最晚必须发生的时间称为该事件的最迟发生时间。同样，有些活动也可以推迟一段时间完成而不影响整个工程的完成，我们把活动最晚必须开始的时间称为该活动的最迟开始时间。一个事件在最迟发生时间内仍没发生，或一个活动在最迟开始时间内仍没开始，则必然会影响整个工程的按时完成。事件 V_j 的最迟发生时间应该为：它的所有直接后继事件 V_{jk}（$1 \leqslant k \leqslant m$）的最迟发生时间减去相应边<$V_j$，$V_{jk}$>上的权（活动 e_{jk} 需要时间），取其中的最小值。且汇点的最迟发生时间就是它的最早发生时间，再按照逆拓扑顺序依次计算出所有事件的最迟发生时间，设用数组 lastest[1..n]表示，即：

lastest[n]=earliest[n]

lastest[j]=min{lastest[k]-dut[j,k]}

(其中，事件 k 是事件 j 的直接后继事件，dut[j,k]表示边<j，k>上的权)

对于上图，用上述方法求 lastest [0..7]的过程如下：

lastest[7]=earliest[7]=19

lastest[6]=lastest[7]-dut[6,7]=19-4=15

lastest[5]=lastest[7]-dut[5,7]=19-2=17

lastest[3]=min{lastest[5]-dut[3,5]，lastest[7]-dut[3,7]}

=min{17-2,19-5}

=14

lastest[4]=min{lastest[3]-dut[4,3]，lastest[5]-dut[4,5], lastest[6]-dut[4,6]}

=min{14-3,17-4,15-3}

=11

lastest[2]=lastest[4]-dut[2,4]=11-4=7

lastest[1]=min{lastest[3]-dut[1,3]，lastest[4]-dut[1,4]}

=min{14-3,11-5}

=6

lastest[0]=min{lastest[1]-dut[0,1]，lastest[2]-dut[0,2]}

=min{6-6,7-7}

=0

计算好每个事件的最早和最迟发生时间后，我们可以很容易地算出每个活动的最早和最迟开始时间，假设分别用 actearliest 和 actlastest 数组表示，设活动 i 的两端事件分别为事件 j 和事件 k,如下所示：

活动 i

事件 j ——————————> 事件 k

则：actearliest[i]=earliest[j]

actlastest[i]=lastest[k]-dut[j,k]

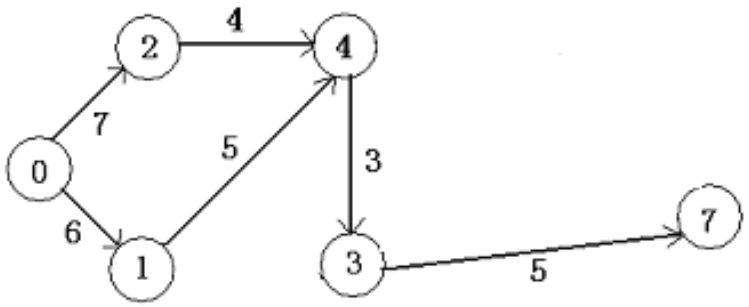
对于上图，用上述方法求得所有活动的最早和最迟开始时间如下表：

活动	<0,1>	<0,2>	<1,3>	<1,4>	<2,4>	<3,5>	<3,7>	<4,3>	<4,5>	<4,6>	<5,7>	<6,7>
最早	0	0	6	6	7	14	14	11	11	11	16	14
最迟	0	0	11	6	7	15	14	11	13	12	17	15
余量	0	0	5	0	0	1	0	0	2	1	1	1

上表中的余量（称为开始时间余量）是该活动的最迟开始时间减去最早开始时间，余量不等于 0 的活动表示该活动不一定要在最早开始时间时就进行，可以拖延一定的余量时间再进行，也不会影响整个工程的完成。而余量等于 0 的活动必须在最早开始时间时进行，而且在规定的工期内完成，否则将影响整个工程的完成。

我们把开始时间余量为 0 的活动称为“关键活动”，由关键活动所形成的从源点到汇点的每一条路径称为“关键路径”。

上图所示的 AOE 网的关键路径如下图所示。



细心的读者可能已经发现，其实关键路径就是从源点到汇点具有最大路径长度的那些路径。这很容易理解，因为整个工程的工期就是按照最长路径计算出来的。很显然，要想缩短整个工程的工期，就应该想法设法去缩短关键活动的持续时间。读者可以根据上面的思想编程求出 AOE 网的关键路径。