**算法与数据结构综合训练**

**基本算法演示程序实习报告**

|  |  |  |
| --- | --- | --- |
| **作者** | **底云飞（20142983）** | |
| **指导** | **软件工程系** | **刘立嘉** |

**文档说明**

本文档为2016年暑期算法与数据结构综合训练第二阶段项目综合应用算法训练的实习报告。

**文档控制**

文档作者：底云飞

创建日期：2016年06月01日

确认日期：2016年06月01日

控制编码：ZHYY\_01

当前版本：1.0

**更改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 作者 | 版本 | 更改参考 |
|  |  |  |  |

**文件归档:**

|  |  |  |
| --- | --- | --- |
| 归档编号 | 归档时间 | 发给用户时间 |
|  |  |  |

**目录**

1 由遍历序列恢复二叉树（作者） 8

1.1算法程序及注释 8

1.2数据输入/输出说明 8

1.3算法流程图 8

1.4算法测试截图 8

2 Floyd算法（作者） 9

2.1算法程序及注释 10

2.2数据输入/输出说明 10

2.3算法流程图 10

2.4算法测试截图 10

3 拓扑排序（作者） 10

3.1算法程序及注释 11

3.2数据输入/输出说明 11

3.3算法流程图 11

3.4算法测试截图 11

4 矩阵运算算法（作者） 16

4.1算法程序及注释 16

4.2数据输入/输出说明 16

4.3算法流程图 16

4.4算法测试截图 16

## 1 由遍历序列恢复二叉树（底云飞）

功能要求：输出二叉树形态或输出二叉树的三种遍历序列

### 1.1算法程序及注释

/\*

前序：a b c d e x

中序：b c a e d x

后序：c b e x d a

\*/

#include<iostream>

using namespace std;

#define TREELEN 100

//数据结构定义

struct NODE

{

NODE \*pLeft; //左子树

NODE \*pRight; //右子树

char chValue; //该节点的值

};

void ReBuild(char \*pPreOrder, char \*pInOrder, int nTreeLen, NODE \*\*pRoot)

{

// 传过来的数据：ReBuild(szPreOrder, szInOrder, number, &pRoot);

if(pPreOrder == NULL || pInOrder == NULL) //检查边界条件，检查前序或者中序是否为空

return ;

NODE \*pTemp = new NODE; //获得前序遍历的第一个节点

pTemp->chValue = \*pPreOrder; //下面三句是将第一个节点初始化

pTemp->pLeft = NULL;

pTemp->pRight = NULL;

if(\*pRoot == NULL) //如果节点为空，把当前节点复制到根节点

\*pRoot = pTemp;

if(nTreeLen == 1) //如果当前树长度为1，那么已经是最后一个节点，则二叉树只有一个节点

return;

//寻找子树长度

char \*pOrgInOrder = pInOrder;

char \*pLeftEnd = pInOrder;

int nTempLen = 0; //为了记录每递归一次，子树的长度不要超出序列长度

while(\*pPreOrder != \*pLeftEnd) //找到左子树的结尾

{

/\* if(pPreOrder == NULL || pLeftEnd == NULL) //同19行的判断重复，可能不需要再写

return;\*/

nTempLen++;

//记录临时长度，以免溢出

if(nTempLen > nTreeLen)

break;

pLeftEnd++;

}

int nLeftLen = (int)(pLeftEnd - pOrgInOrder); //寻找左子树长度

int nRightLen = nTreeLen - nLeftLen - 1; //寻找右子树长度，总长度减去左子树，减去根结点

if(nLeftLen > 0) //重建左子树

ReBuild(pPreOrder + 1, pInOrder, nLeftLen, &((\*pRoot)->pLeft)); //&((\*pRoot)->pLeft)递归到左子树的下一个节点

if(nRightLen > 0) //重建右子树

ReBuild(pPreOrder + nLeftLen + 1, pInOrder + nLeftLen + 1, nRightLen, &((\*pRoot)->pRight));

}

////前序遍历结果

//void PrePrint(NODE \*pRoot)

//{

// if(pRoot == NULL)

// return;

// cout << pRoot->chValue << " ";

// PrePrint(pRoot->pLeft);

// PrePrint(pRoot->pRight);

//}

////中序遍历结果

//void InPrint(NODE \*pRoot)

//{

// if(pRoot == NULL)

// return;

// InPrint(pRoot->pLeft);

// cout << pRoot->chValue << " ";

// InPrint(pRoot->pRight);

//}

//后序遍历结果

void LastPrint(NODE \*pRoot) //递归输出序列节点的值

{

if(pRoot == NULL)

return;

LastPrint(pRoot->pLeft);

LastPrint(pRoot->pRight);

cout << pRoot->chValue << " ";

}

void main()

{

char szPreOrder[TREELEN]; //前序序列个数

char szInOrder[TREELEN]; //中序序列个数

int number = 1; //number为序列元素个数，初始化为1

cout << "请输入序列元素个数：";

cin >> number;

if(number <= 0)

{

cout << "没有二叉树" << endl;

exit(1);

}

cout << "请输入前序序列：" << endl;

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

{

cin >> szPreOrder[i];

if(number == 1)

{

cout << "后序序列为：" << endl << szPreOrder[i] << endl;

exit(1);

}

}

cout << "请输入中序序列：" << endl;

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

cin >> szInOrder[i];

NODE \*pRoot = NULL;

ReBuild(szPreOrder, szInOrder, number, &pRoot);

/\*cout<<"前序遍历结果："<<endl;

PrePrint(pRoot);

cout << endl << endl;;

cout<<"中序遍历结果："<<endl;

InPrint(pRoot);

cout << endl;\*/

cout << endl << "后序遍历结果：" << endl;

LastPrint(pRoot);

cout << endl;

}

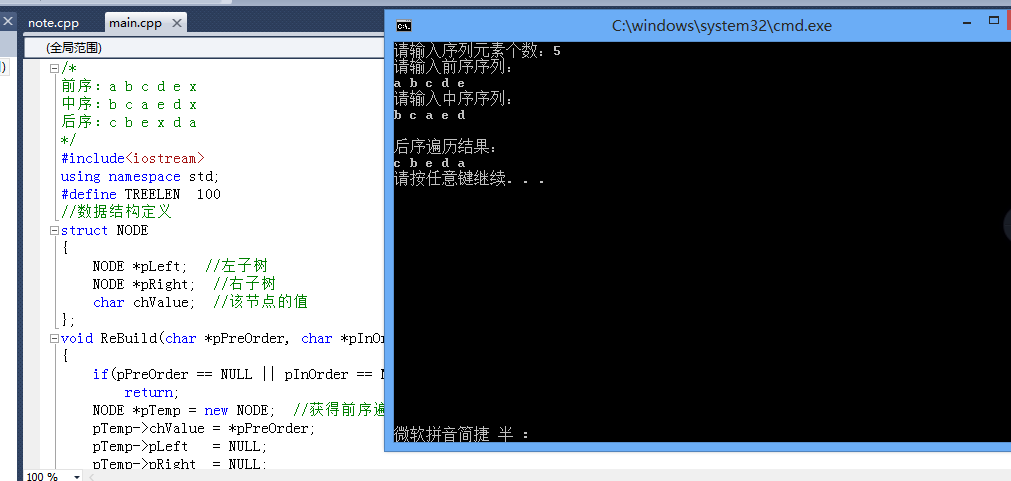
### 1.2数据输入/输出说明

输入：遍历序列。即输入前序和中序序列。输出为二叉树的后序序列。

当输入前序（a b c d e）和中序(b c a e d)序列之后，可以输出后序序列(c b e d a)。。

### 1.3算法流程图

### 1.4算法测试截图



## 2 Floyd算法（底云飞）

功能要求：输出各顶点对间最短路径和路径长度。

### 2.1算法程序及注释

//算法6.11　弗洛伊德算法

#include <iostream>

using namespace std;

#define MaxInt 32767 //表示边的权值的极大值，即∞

#define MVNum 100 //最大顶点数

typedef char VerTexType; //假设顶点的数据类型为字符型

typedef int ArcType; //假设边的权值类型为整型

int Path[MVNum][MVNum]; //最短路径上顶点vj的前一顶点的序号

int D[MVNum][MVNum]; //记录顶点vi和vj之间的最短路径长度

//图的邻接矩阵

typedef struct

{

VerTexType vexs[MVNum]; //顶点表

ArcType arcs[MVNum][MVNum]; //邻接矩阵

int vexnum, arcnum; //图的当前点数和边数

} AMGraph;

//确定点v在G中的位置

int LocateVex(AMGraph G , VerTexType v)

{

for(int i = 0; i < G.vexnum; ++i)

if(G.vexs[i] == v) //如果在顶点表中有V这个顶点，那么就把顶点的位置i传回去

return i;

return -1; //如果没有这个顶点，那么邻接矩阵中的距离则变为MaxInt

}//LocateVex

//采用邻接矩阵表示法，创建有向网G

void CreateUDN(AMGraph &G)

{

int i , j , k;

cout << "请输入总顶点数，总边数，以空格隔开:";

cin >> G.vexnum >> G.arcnum; //输入总顶点数，总边数

cout << endl << "输入点的名称，如a" << endl;

for(i = 0; i < G.vexnum; ++i)

{

cout << "请输入第" << (i + 1) << "个点的名称:";

cin >> G.vexs[i]; //依次输入点的信息 。char vexs[MVNum]; //顶点表

}

for(i = 0; i < G.vexnum; ++i) //初始化邻接矩阵，边的权值均置为极大值MaxInt。vexnum总顶点数

{

for(j = 0; j < G.vexnum; ++j)

{

if(j == i) //相等则表示同一个点，距离为0

G.arcs[i][j] = 0; //int arcs[MVNum][MVNum]; //邻接矩阵

else

G.arcs[i][j] = MaxInt;

}//for(i)

}//for(i)

cout << endl << "输入边依附的顶点及权值，如a b 3" << endl;

for(k = 0; k < G.arcnum; ++k) //构造邻接矩阵.arcnum边的总数

{

VerTexType v1 , v2; //VerTexType相当于char，字符型

ArcType w; //ArcType相当于int，整型

cout << "请输入第" << (k + 1) << "条边依附的顶点及权值:";

cin >> v1 >> v2 >> w; //输入一条边依附的顶点及权值

i = LocateVex(G, v1);

j = LocateVex(G, v2); //确定v1和v2在G中的位置，即顶点数组的下标

G.arcs[i][j] = w; //顶点为<v1, v2>的边的权值置为w

}//for

}//CreateUDN

//用Floyd算法求有向网G中各对顶点i和j之间的最短路径

void ShortestPath\_Floyed(AMGraph G)

{

int i , j , k ;

for (i = 0; i < G.vexnum; ++i) //各对结点之间初始已知路径及距离

for(j = 0; j < G.vexnum; ++j)

{

D[i][j] = G.arcs[i][j]; //int D[MVNum][MVNum]; 记录顶点vi和vj之间的最短路径长度

if(D[i][j] < MaxInt && i != j)

Path[i][j] = i; //如果i和j之间有弧，则将j的前驱置为i

//int Path[MVNum][MVNum];最短路径上顶点vj的前一顶点的序号

else

Path [i][j] = -1; //如果i和j之间无弧，则将j的前驱置为-1

}//for

for(k = 0; k < G.vexnum; ++k)

for(i = 0; i < G.vexnum; ++i)

for(j = 0; j < G.vexnum; ++j)

if(D[i][k] + D[k][j] < D[i][j]) //如果从i经k到j的一条路径更短

{

D[i][j] = D[i][k] + D[k][j]; //如果经过k路径更短，那么更新D[i][j]，i和j之间的最短路径

Path[i][j] = Path[k][j]; //更改j的前驱为k

}//if

}//ShortestPath\_Floyed

//显示最短路径

void DisplayPath(AMGraph G , int begin , int temp ) //temp终点

{

if(Path[begin][temp] != -1) //Path [i][j] = -1; //如果i和j之间无弧，则将j的前驱置为-1

{

DisplayPath(G , begin , Path[begin][temp]); //递归输出路径

cout << G.vexs[Path[begin][temp]] << "--->";

}

}//DisplayPath

void main()

{

cout << "\*\*\*\*\*\*\*\*\*\*\*\*算法6.11　弗洛伊德算法求最短路径\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

AMGraph G;

char start , destination; //起点和终点

int num\_start , num\_destination; //用于表示起点和终点的位置

CreateUDN(G); //采用邻接矩阵表示法，创建有向网G

//cout << endl << "有向网G创建完成！" << endl;

ShortestPath\_Floyed(G); //用Floyd算法求有向网G中各对顶点i和j之间的最短路径

//cout << "请依次输入路径的起点与终点的名称：";

for(int i = 0; i < G.vexnum; i++) //两个循环是为了输出每两个有路径的点的距离

{

for(int j = 0; j < G.vexnum; j++)

{

start = G.vexs[i];

destination = G.vexs[j];

num\_start = LocateVex(G , start);

num\_destination = LocateVex(G , destination);

DisplayPath(G , num\_start , num\_destination); //显示出最短路径，参数为图G，起点num\_start，终点num\_destination

if(D[num\_start][num\_destination] == 0 || D[num\_start][num\_destination] == 32767) //如果两个点之间的距离为0或者32767，则不输出这个路径

continue;

else

{

cout << G.vexs[num\_destination] << endl;

cout << "最短路径的长度为：" << D[num\_start][num\_destination] << endl;

cout << endl;

}

}

}

}//main

### 2.2数据输入/输出说明

输入：有向图（顶点序列和有向边序列）。

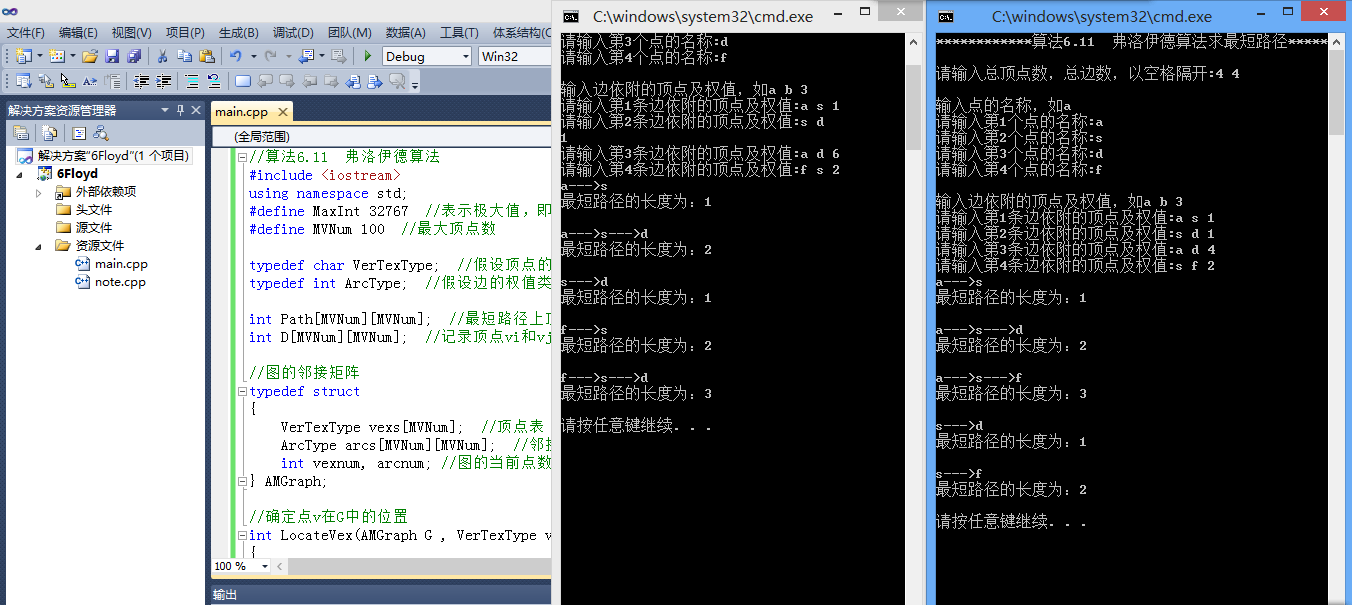
输出为指定点到其他所有可以到达点的最短路径和路径长度。

### 2.3算法流程图

给出该算法程序的流程图（可以以图片形式）。

### 2.4算法测试截图

两次输入不一样的地方在于第四条路径，前面的是从s到f的，相当于正着走，后面的是f到s的，相当于返回来走，由此产生的路径就是不同的。



## 3 拓扑排序（底云飞）

功能要求：能判断是否是 AOV 网；输出有效的拓扑排序序列。

### 3.1算法程序及注释

//算法6.12　拓扑排序

#include <iostream>

using namespace std;

#define MVNum 100 //最大顶点数

#define OK 1

#define ERROR 0

typedef char VerTexType;

//图的邻接表存储表示

typedef struct ArcNode //边结点

{

int adjvex; //该边所指向的顶点的位置

struct ArcNode \*nextarc; //指向下一条边的指针

} ArcNode;

typedef struct VNode

{

VerTexType data; //顶点信息

ArcNode \*firstarc; //指向第一条依附该顶点的边的指针

} VNode, AdjList[MVNum]; //AdjList表示邻接表类型

typedef struct

{

AdjList vertices; //邻接表

AdjList converse\_vertices; //逆邻接表

int vexnum, arcnum; //图的当前顶点数和边数

} ALGraph;

//- - - - -顺序栈的定义- - - - -

typedef struct

{

int \*base;

int \*top;

int stacksize;

} spStack;

int indegree[MVNum]; //数组indegree存放个顶点的入度

spStack S;

//栈的操作

void InitStack(spStack &S)

{

S.base = new int[MVNum]; //初始化栈

if(!S.base)

exit(1);

S.top = S.base;

S.stacksize = MVNum;

}//InitStack

//进栈

void Push(spStack &S , int i)

{

if(S.top - S.base == S.stacksize)

return;

\*S.top++ = i;

}//Push

//出栈

void Pop(spStack &S , int &i)

{

if(S.top == S.base)

return;

i = \*--S.top;

}//Pop

//判断栈是否为空

bool StackEmpty(spStack S)

{

if(S.top == S.base)

return true;

return false;

}//StackEmpty

//确定点v在G中的位置

int LocateVex(ALGraph G , VerTexType v)

{

for(int i = 0; i < G.vexnum; ++i)

if(G.vertices[i].data == v)

return i;

return -1;

}//LocateVex

//创建有向图G的邻接表、逆邻接表

int CreateUDG(ALGraph &G)

{

cout << "请输入总顶点数，总边数，以空格隔开:";

cin >> G.vexnum >> G.arcnum; //输入总顶点数，总边数

cout << endl << "输入点的名称，如a" << endl;

for(int i = 0; i < G.vexnum; ++i) //输入各点，构造表头结点表

{

cout << "请输入第" << (i + 1) << "个点的名称:";

cin >> G.vertices[i].data; //输入顶点值

G.converse\_vertices[i].data = G.vertices[i].data;

G.vertices[i].firstarc = NULL; //初始化表头结点的指针域为NULL

G.converse\_vertices[i].firstarc = NULL;

}

cout << endl << "输入边依附的顶点，如a b" << endl;

for(int k = 0; k < G.arcnum; ++k) //输入各边，构造邻接表

{

VerTexType v1 , v2;

int i , j;

cout << "请输入第" << (k + 1) << "条边依附的顶点:";

cin >> v1 >> v2; //输入一条边依附的两个顶点

i = LocateVex(G, v1);

j = LocateVex(G, v2);

//确定v1和v2在G中位置，即顶点在G.vertices中的序号

ArcNode \*p1 = new ArcNode; //生成一个新的边结点\*p1

p1->adjvex = j; //邻接点序号为j

p1->nextarc = G.vertices[i].firstarc;

G.vertices[i].firstarc = p1;

//将新结点\*p1插入顶点vi的边表头部

ArcNode \*p2 = new ArcNode; //生成一个新的边结点\*p1

p2->adjvex = i; //逆邻接点序号为i

p2->nextarc = G.converse\_vertices[j].firstarc;

G.converse\_vertices[j].firstarc = p2;

//将新结点\*p1插入顶点vi的边表头部

}

return OK;

}//CreateUDG

//求出各顶点的入度存入数组indegree中

void FindInDegree(ALGraph G)

{

int i, count;

for(i = 0 ; i < G.vexnum ; i++)

{

count = 0;

ArcNode \*p = G.converse\_vertices[i].firstarc;

if(p)

{

while(p)

{

p = p->nextarc;

count++;

}

}

indegree[i] = count;

}

}//FindInDegree

//有向图G采用邻接表存储结构.若G无回路，则生成G的一个拓扑序列topo[]并返回OK，否则ERROR

int TopologicalSort(ALGraph G , int topo[])

{

int i , m;

FindInDegree(G); //求出各顶点的入度存入数组indegree中

InitStack(S); //栈S初始化为空

for(i = 0; i < G.vexnum; ++i)

if(!indegree[i]) Push(S, i); //入度为0者进栈

m = 0; //对输出顶点计数，初始为0

while(!StackEmpty(S)) //栈S非空

{

Pop(S, i); //将栈顶顶点vi出栈

topo[m] = i; //将vi保存在拓扑序列数组topo中

++m; //对输出顶点计数

ArcNode \*p = G.vertices[i].firstarc; //p指向vi的第一个邻接点

while(p)

{

int k = p->adjvex; //vk为vi的邻接点

--indegree[k]; //vi的每个邻接点的入度减1

if(indegree[k] == 0) Push(S, k); //若入度减为0，则入栈

p = p->nextarc; //p指向顶点vi下一个邻接结点

}//while(p)

}//while(!StackEmpty(S))

if(m < G.vexnum) return ERROR; //该有向图有回路

else return OK;

}//TopologicalSort

int main()

{

//cout << "\*\*\*\*\*\*\*\*\*\*\*\*算法6.12　拓扑排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

ALGraph G;

CreateUDG(G);

int \*topo = new int [G.vexnum];

cout << endl;

//cout << "有向图的邻接表、逆邻接表创建完成!" << endl << endl;

if(TopologicalSort(G , topo))

{

cout << "是AOV网，该有向图的拓扑排序序列为：";

for(int j = 0 ; j < G.vexnum; j++)

{

if(j != G.vexnum - 1)

cout << G.vertices[topo[j]].data << " , ";

else

cout << G.vertices[topo[j]].data << endl << endl;

}//for

}

else

cout << "网中存在环，不是AOV网，无法进行拓扑排序！" << endl << endl;

return OK;

}//main

### 3.2数据输入/输出说明

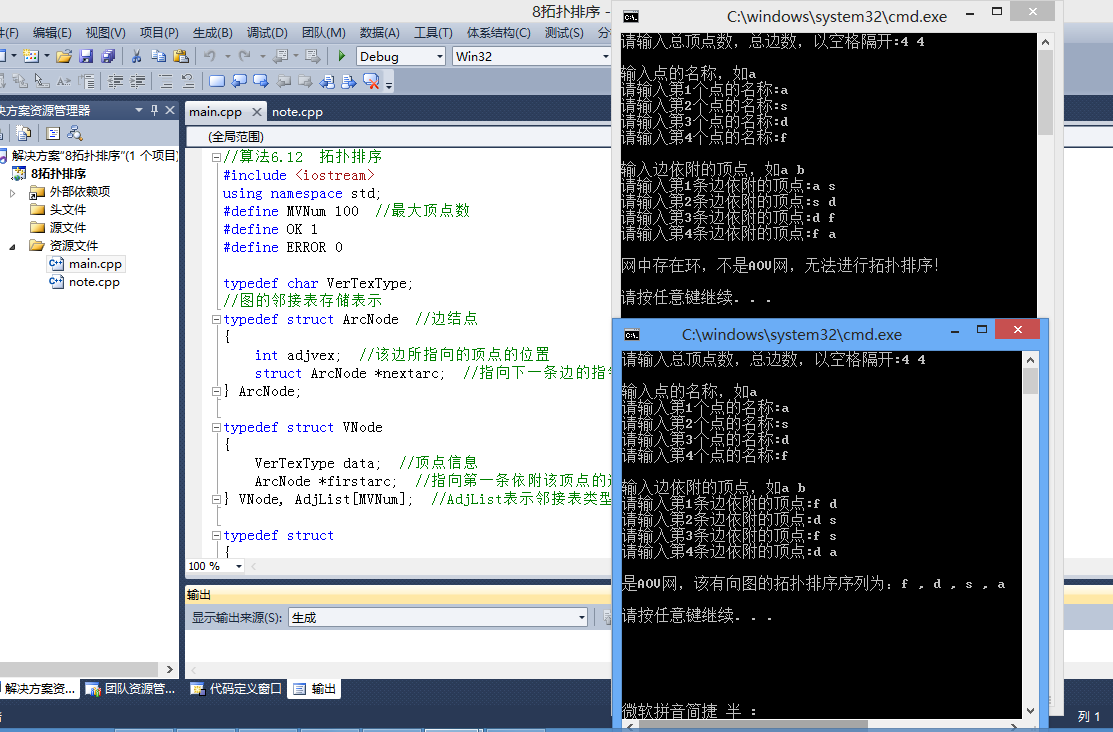
输入：有向图（顶点序列，有向边序列）

输出：如果不是AOV网，则显示不是AOV网；如果是，显示是AOV网，并且输出拓扑排序序列。

### 3.3算法流程图

给出该算法程序的流程图（可以以图片形式）。

### 3.4算法测试截图



## 4 矩阵运算算法（底云飞）

功能要求：输出矩阵加法、减法和乘法的运算结果

### 4.1算法程序及注释

#include <iostream>

#define N 10

using namespace std;

//矩阵加法

int jia(int a[N][N], int b[N][N], int n)

{

int c[N][N];

cout << endl << "矩阵加法为：" << endl;

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

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

{

c[i][j] = a[i][j] + b[i][j];

cout << c[i][j] << " ";

if(j == (n - 1)) cout << endl;

}

return c[N][N];

}

//矩阵减法

int jian(int a[N][N], int b[N][N], int n)

{

int d[N][N];

cout << endl << "矩阵减法为：" << endl;

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

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

{

d[i][j] = a[i][j] - b[i][j];

cout << d[i][j] << " ";

if(j == (n - 1)) cout << endl;

}

return d[N][N];

}

//矩阵乘法

int cheng(int a[N][N], int b[N][N], int n)

{

int e[N][N];

cout << endl << "矩阵乘法为：" << endl;

for(int i = 0; i < n; i++) //横行乘以竖列能够得出当前行列的值

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

{

int s = 0;

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

s = s + a[i][k] \* b[k][j]; //需要三个数乘之后再加

e[i][j] = s;

cout << s << " ";

if(j == (n - 1)) cout << endl; //计算完三个数之后 要换行

}

cout << endl;

return e[N][N];

}

//打印出矩阵

void display(int a[N][N], int n)

{

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

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

{

cout << a[i][j] << " ";

if(j == n - 1)

cout << endl;

}

}

void main()

{

int n = 1; //矩阵阶数

int a[N][N], b[N][N], c[N][N], d[N][N], e[N][N]; //abcde分别表示第一，第二，加法，减法，乘法矩阵

cout << "请输入矩阵的阶数：";

cin >> n;

cout << "请输入第一个矩阵的值：" << endl;

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

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

cin >> a[i][j];

cout << "输入的第一个矩阵为：" << endl;

display(a, n);

cout << "请输入第二个矩阵的值：" << endl;

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

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

cin >> b[i][j];

cout << "输入的第二个矩阵为：" << endl;

display(b, n);

c[N][N] = jia(a, b, n);

d[N][N] = jian(a, b, n);

e[N][N] = cheng(a, b, n);

}

### 4.2数据输入/输出说明

输入待运算的矩阵，输出矩阵的加法，减法，乘方的运算结果矩阵。

### 4.3算法流程图

给出该算法程序的流程图（可以以图片形式）。

### 4.4算法测试截图

