图及相关算法

一、 图的基本概念

1. 图的的定义

图是由一个顶点的集合V和一个顶点间关系的集合E组成：

记 G=（V，E）

V：顶点的有限非空集合。

E：顶点间关系的有限集合（边集）。

存在一个结点v，可能含有多个前驱结点和后继结点。

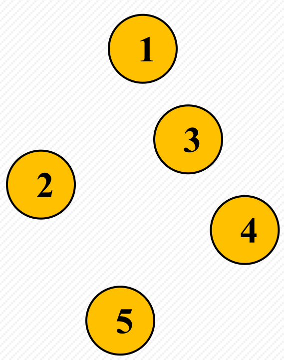


图1

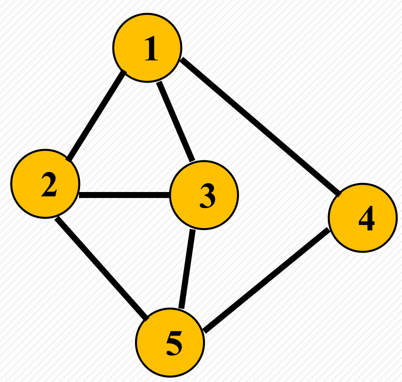


图2

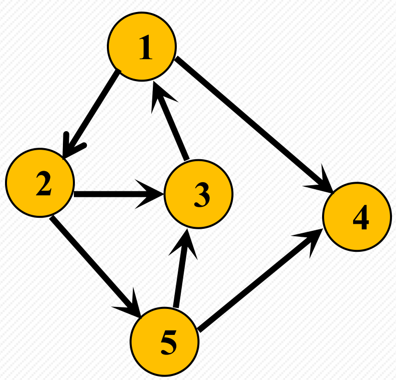


图3

2. 无向图和有向图

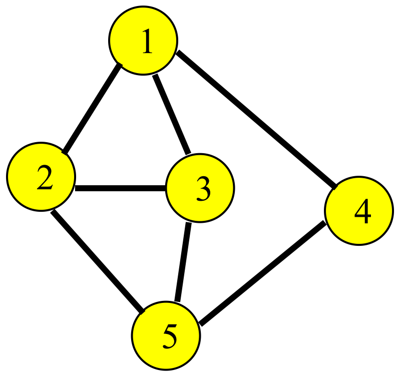
无向图：

在图G=（V，E）中，如果对于任意的顶点a，b∈V，当(a，b)∈E时，必有（b，a）∈E（即关系R对称），此图称为无向图。

无向图中用不带箭头的边表示顶点的关系。

V={1, 2, 3, 4, 5}

E={(1, 2),(1, 3),(1, 4),(2,3),(2,5),(3, 5),(4,5)}

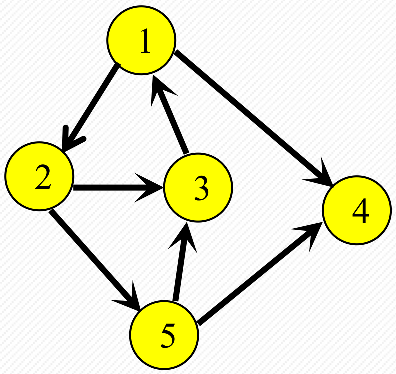


有向图:

如果对于任意的顶点a，b∈V，当(a，b)∈E时 ，(b，a)∈E未必成立，则称此图为有向图。

在有向图中，通常用带箭头的边连接两个有关联的结点。

V={1, 2, 3, 4,5}
E={<1, 2> , <1 , 4> , <2 , 3> , < 2 , 5> , <3 , 1>,<5 , 3>, < 5 , 4 >}



3. 顶点的度、入度和出度

在无向图中：顶点v的度是指与顶点v相连的边的数目D(v)。D(2)=3

在有向图中：

入度——以该顶点为终点的边的数目和。ID(3)=2

出度——以该顶点为起点的边的数目和。OD(3)=1

度数为奇数的顶点叫做奇点，度数为偶数的点叫做偶点。

度：等于该顶点的入度与出度之和。

D(5)=ID(5)+OD(5)=1+2=3

结论：图中所有顶点的度=边数的两倍

$$\sum_{i=1}^{n} D(v_i) = 2 * e$$

无向完全图 $e = \dfrac{n*(n-1)}{2}$

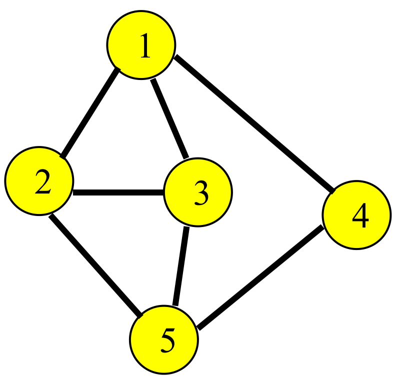


图1 无向图

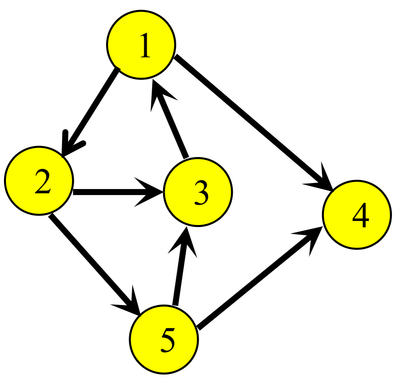


图2 有向图

4. 路径、简单路径、回路

在图G=（V，E）中，如果对于顶点a，b，存在满足下述条件的顶点序列$x_1 \ldots\ldots x_k (k>1)$

(1) $x_1=a$，$x_k=b$ (2) $(x_i，x_{i+1}) \in E$ $i=1 \cdot\cdot k-1$

则称顶点序列$x_1=a$，x_2，…，$x_k=b$为顶点a到顶点b的一条路径，而路径上边的数目（k-1）称为该路径的长度。

图1: 1. (1,2,3,5) 长度=3

2. (1,2,3,5,2) 长度=4

3. (1,2,5,4,1) 长度=4

图2: (1,2,5,4) 长度=3

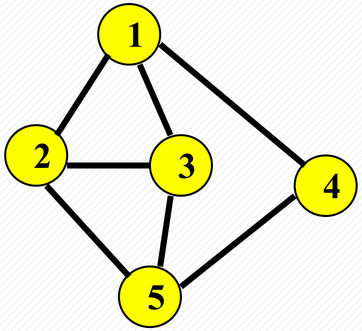


图1

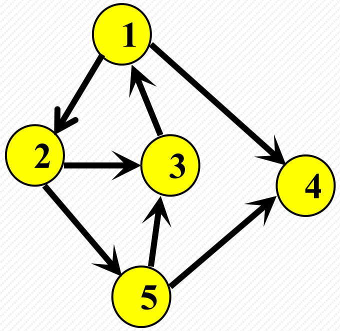


图2

(1) 如果一条路径上的顶点除起点x_1和终点x_k可以相同外，其它顶点均不相同，则称此路径为一条**简单路径**。

(2) $x_1=x_k$的简单路径称为回路（也称为环）。

5. 连通、连通图、连通分量（无向图）

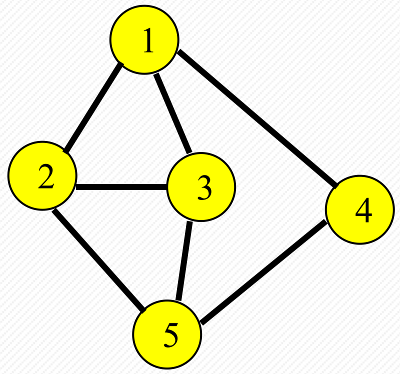


图1 连通图

连通：“连成一片”。

连通图：“能连成一片的图”。

定义：
连通：如果存在一条从顶点u到v的路径，则称u和v是连通的。
连通图：图中任意的两个顶点u和v都是连通的，称为连通图。
否则称为非连通图。

连通分量：无向图中的极大连通子图。

图2中有3个连通分量：

{1 2 4 5} {3 6} {7 8}

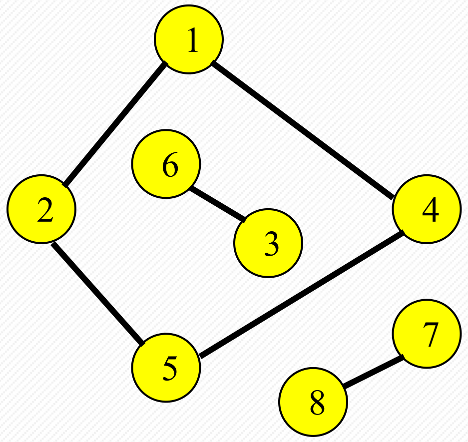
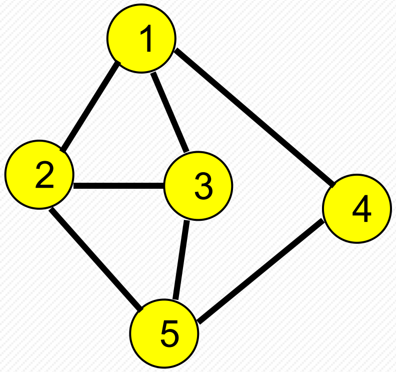


图2 非连通图

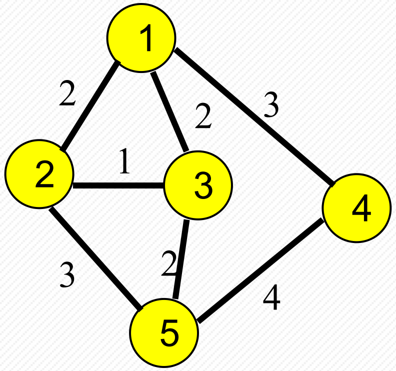
求连通分量数，最大连通分量等。

有向图：强连通、强连通图、强连通分量

6. 带权图



一般的图边上没有数字，边仅表示两个顶点的相连接关系。



图中的边可以加上表示某种含义的数值，数值称为边的权，此图称为带权图。也称为网。

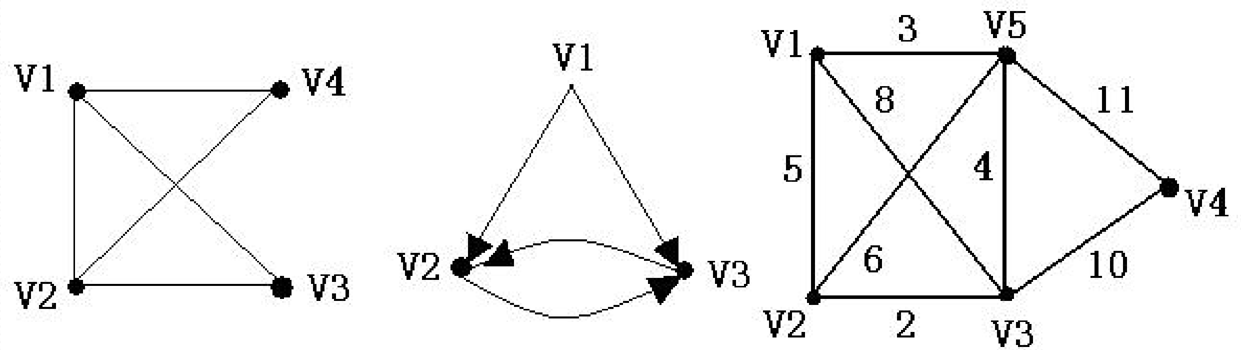
图的存储

图型结构的存储分为静态存储和动态存储。我们介绍下面三种：邻接矩阵、邻接表（用数组模拟）、边集数组。

1. 邻接矩阵

邻接矩阵是表示顶点间相邻关系的矩阵。若**G=（V，E）**是一个具有**n**个顶点的图，则**G**的邻接矩阵是如下定义的二维数组**a**，其规模为**n*n**。

$$a[i,j]=\begin{cases}1(\text{或权}) & (v_i,v_j)\in E\\ 0(\pm\infty) & (v_i,v_j)\notin E\end{cases}$$



图（A）　　图（B）　　图（C）

上图中的3个图对应的邻接矩阵分别如下：

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

下面是建立图的邻接矩阵的参考程序段：

```
int i,j,k,e,n;double g[101][101];
double w;
int main()
{   int i,j;
    for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
        g[i][j] = 0x7fffffff;  //初始化，对于不带权的图g[i][j]=0，
                               //表示没有边连通。这里用0x7fffffff代替无穷大。
    cin >> e;
    for (k = 1; k <= e; k++)
    {    cin >> i >> j >> w;
         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,如果是有向图则不要有这句！
    }
    return 0;
}
```

邻接矩阵存储的特点

1. 占用的存储单元数只与顶点数有关而与边数无关，n*n的二维数组。

2. 方便度数的计算。

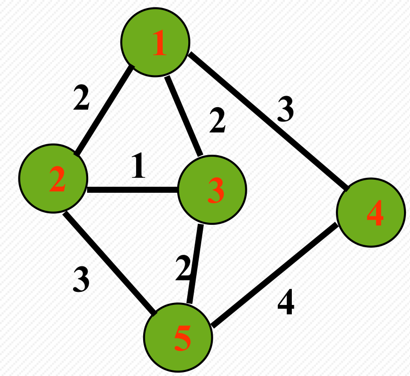
3. 容易判断两点之间是否有边相连。

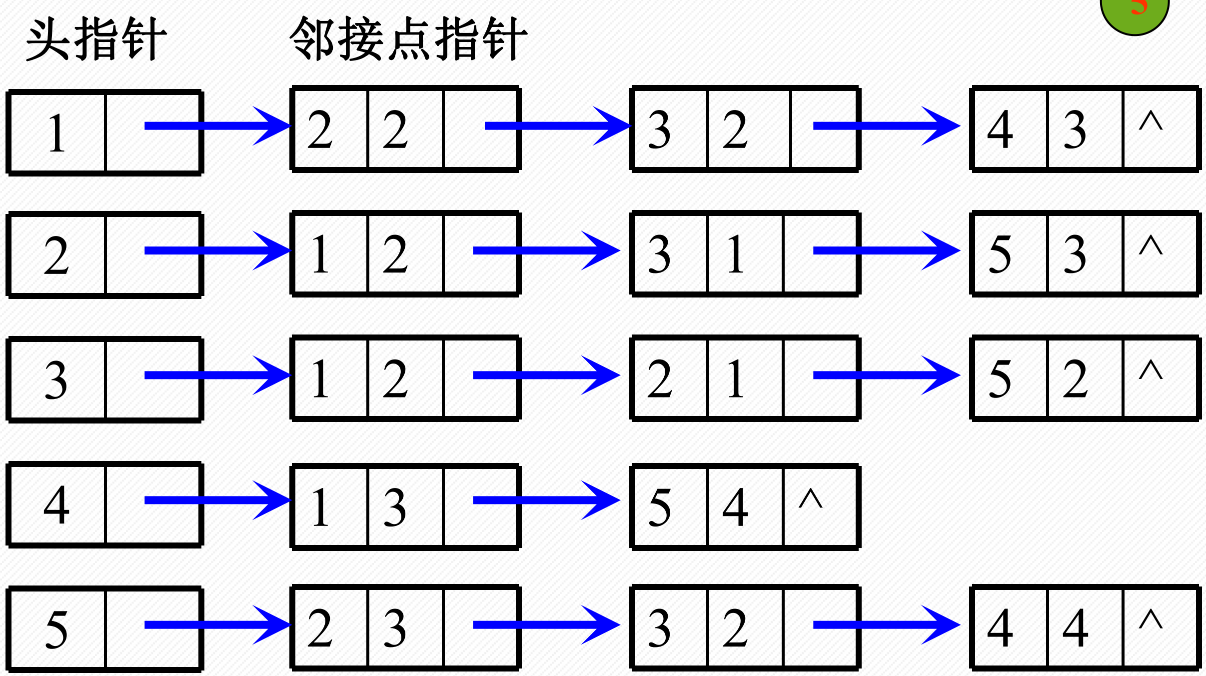
4. 寻找一个点相连的所有边需要一个1到n的循环。

2、邻接表

顶点	邻接点指针

邻接点	边权值	下一个邻接点指针



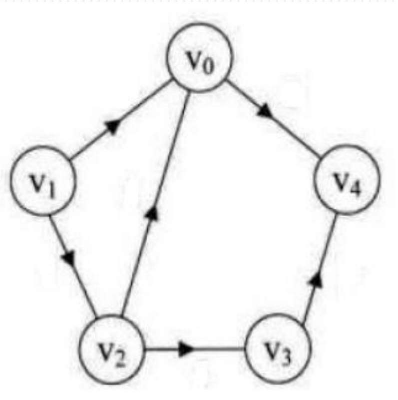


数组模拟邻接表方法一

- int g[101][101];
- 寻找顶点i有边相连的点可以这样来做，g[i][0]表示i发出的边的数量，g[i][j]表示i发出的第j条边是通向哪个顶点的。
- for (int j = 1; j <= g[i][0]; j++)
- ......g[i][j].....
- 这样就可以处理i发出的每条边，也就能找到顶点i有边相连的顶点。

数组模拟邻接表方法二

数组模拟邻接表存储：　大多数情况下只要用数组模拟即可



5 6
0 4
1 0
1 2
2 0
2 3
3 4

参考程序段：

```
#include <iostream>
using namespace std;
const int maxn=1001,maxm=100001;
struct Edge
{
        int next; //下一条边的编号
        int to; //这条边到达的点
}edge[maxm];

int head[maxn],num_edge,n,m,u,v;

void add_edge(int from,int to) //加入一条从from到to的单向边
{
        edge[++num_edge].next=head[from];
        edge[num_edge].to=to;
        head[from]=num_edge;
}
```

head[maxn]

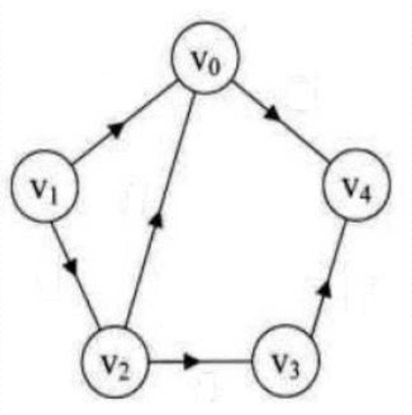
0	1	2	3	4	
1	3	5	6		

edge[maxm]

	0	1	2	3	4	5	6	
next		0	0	2	0	4	0	
to		4	0	2	0	3	4	

计算每个点的出度

```
int main(){
        num_edge=0;
        scanf("%d %d",&n,&m);//读入点数和边数
        for(int i=1;i<=m;i++){
            scanf("%d %d",&u,&v);
            //u、v之间有一条边
            add_edge(u,v);
        }
        int j,chudu[maxn];
        for(int i=0;i<n;i++){  //求出每一个顶点的出度
            int tot=0;
            j=head[i];
            while (j!=0){
             tot++;j=edge[j].next;
             }
            chudu[i]=tot;
        }
        for(int i=0;i<n;i++) cout<<chudu[i]<<" ";
        return 0;
}
```



head[maxn]

0	1	2	3	4	
1	3	5	6		

edge[maxm]

	0	1	2	3	4	5	6	
next		0	0	2	0	4	0	
to		4	0	2	0	3	4	

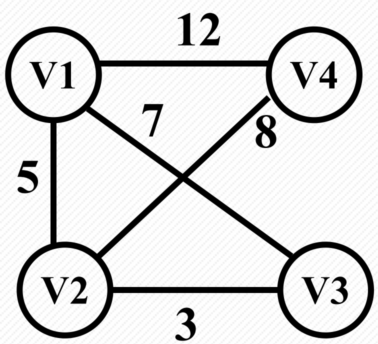
两种方法各有用武之地，需按具体情况，具体选用。

邻接表存储特点

1. 对于稀疏图，用链表实现的邻接表，可以节省内存。
2. 可以快速找到与当前顶点相连的点。
3. 判断两点是否相连不如邻接矩阵快速。

3. 边集数组

边集数组：是利用一维数组存储图中所有边的一种图的表示方法。



起点	1	1	1	2	2
终点	2	3	4	3	4
权	5	7	12	3	8

无向带权图的边集数组表示法

图的遍历

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。

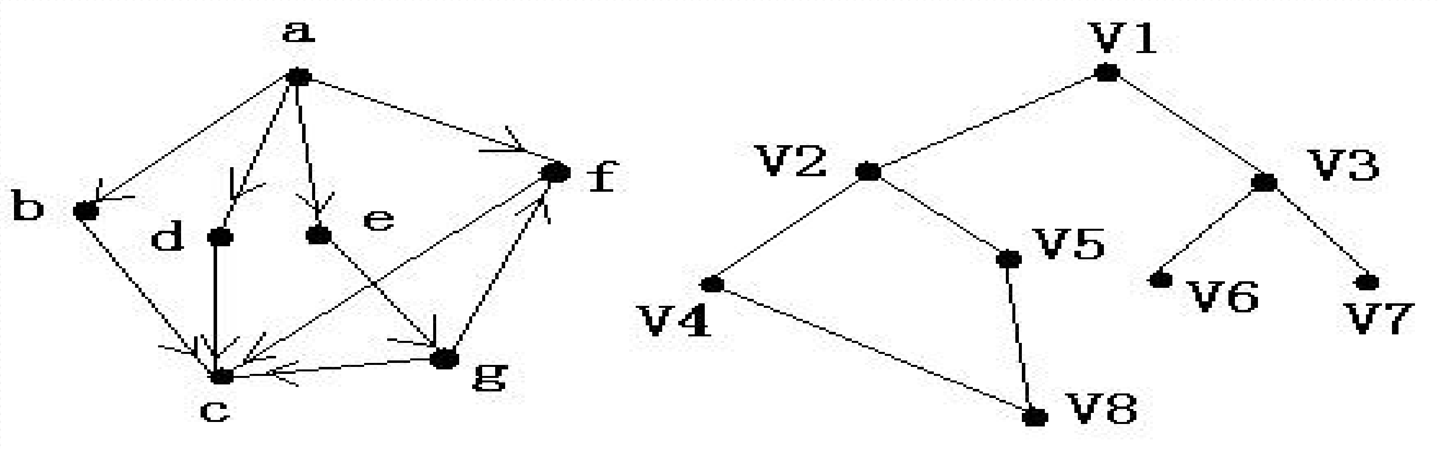
遍历可以采取两种方法进行：

深度优先遍历

广度优先遍历

1. 图的深度优先遍历

对下面两个图分别进行深度优先遍历，写出遍历结果。
注意：分别从a和V1出发。



左图从顶点a出发，进行深度优先遍历的结果为：
a，b，c，d，e，g，f
右图从V_1出发进行深度优先遍历的结果为：
V_1，V_2，V_4，V_8，V_5，V_3，V_6，V_7

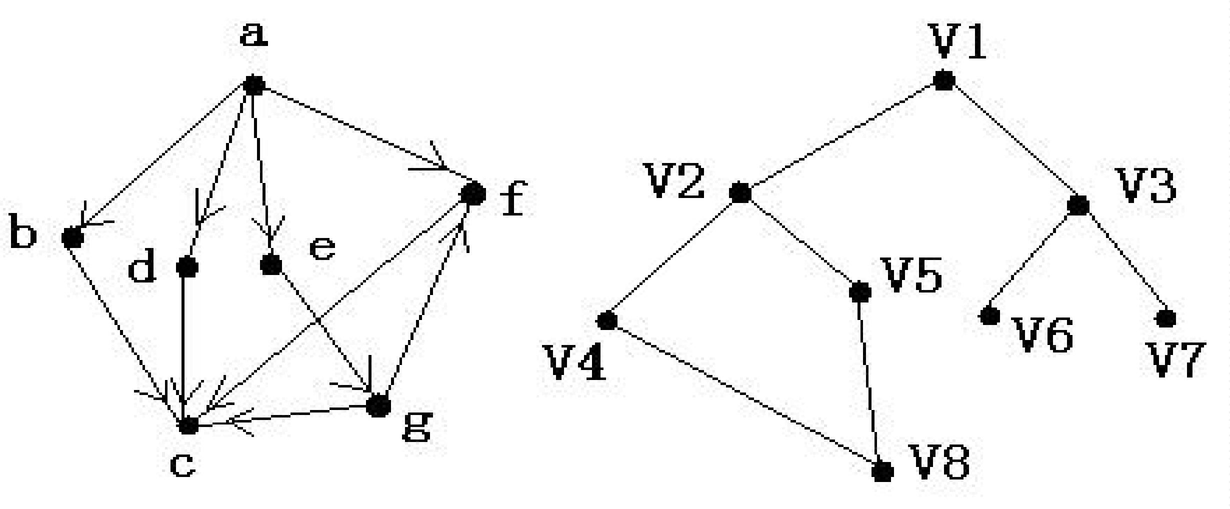
//DFS参考代码

```
#include <cstdio>
const int maxn=1010;
int a[maxn][maxn];
int vis[maxn];
int n,m;
void dfs(int u){
        printf("%d\n",u);
        vis[u]=1;
        for(int i=1;i<=n;i++)
                if(a[u][i]==1&&vis[i]==0) dfs(i);
}
int main(){
        scanf("%d%d",&n,&m);
        for(int i=0;i<m;i++){
                int x,y;
                scanf("%d%d",&x,&y);
                a[x][y]=a[y][x]=1;
        }
        dfs(1);
        return 0;
}
```

为了避免重复访问某个顶点，可以设一个标志数组vis[i]，未访问时值为0，访问一次后就改为1。

2. 图的广度优先遍历

对下面两个图分别从a和V1出发进行广度优先遍历，写出遍历结果。



左图从顶点a出发，进行广度优先遍历的结果为：

a，b，d，e，f，c，g

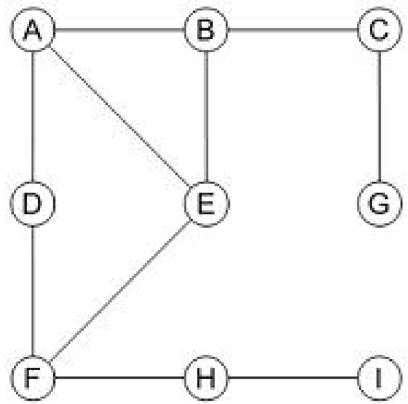
右图从V1出发进行广度优先遍历的结果为：

V_1，V_2，V_3，V_4，V_5，V_6，V_7，V_8

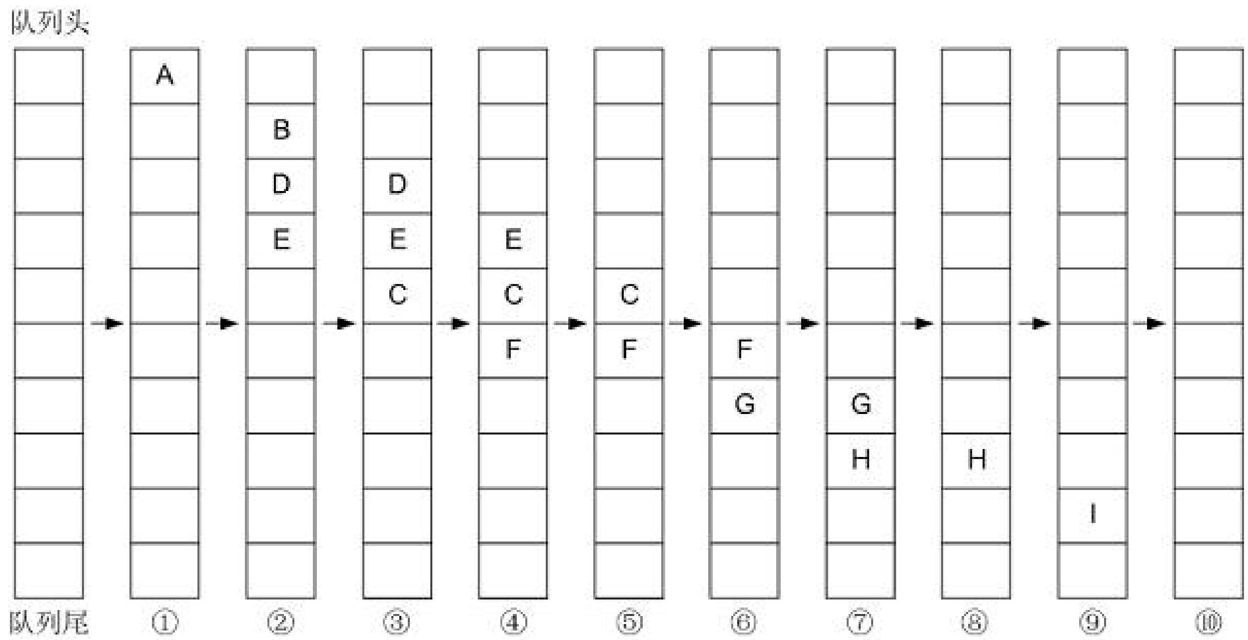
广度优先遍历的实现：

为避免重复访问，也需要一个状态数组 vis[n]，用来存储各顶点的访问状态。如果 vis[i] = 1，则表示顶点 i 已经访问过；如果 vis[i] = 0，则表示顶点 i 还未访问过。初始时，各顶点的访问状态均为 0。

为了实现逐层访问， BFS 算法在实现时需要使用一个队列。



(a) 无向图*G*



```
//BFS参考代码
#include <cstdio>
#include <iostream>
using namespace std;
const int maxn=1010;
int q[maxn];
int a[maxn][maxn];
int vis[maxn];
int n,m;
void bfs(int u){
		int head=0,tail=1;
		q[0]=u;
		vis[u]=1;
		while(head<tail){
				int p=q[head++];
				cout<<p<<endl;
				for(int i=1;i<=n;i++){
						if(a[p][i]==1&&vis[i]==0){
								q[tail++]=i;
								vis[i]=1;
						}
				}
		}
}
```

```
int main(){
        cin>>n>>m;
        for(int i=0;i<m;i++){
                int x,y;
                cin>>x>>y;
                a[x][y]=a[y][x]=1;
        }

       bfs(1);
        return 0;
}
```

计算无向图的连通分量数量：

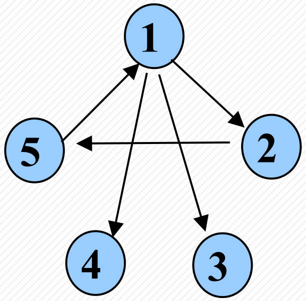
如果一个图不是连通图，那么一次DFS或BFS只能遍历一个连通分量。

求连通分量数量的方法很简单，每次找一个没有遍历的顶点i作为起点dfs(i)或bfs(i)即可，调用的次数就是连通分量数量。

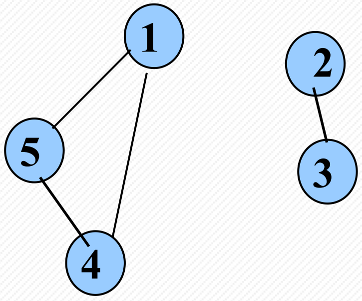
算法：

```
for(int i=1;i<=n;i++)
        if(!vis[i]){
                dfs(i);//或bfs(i);
                cnt++;//连通分量个数
        }
```

图的遍历是图的算法的基础，掌握了图的遍历后，在此基础上稍加变化，能解决很多图的问题。



以3为起点根本不能遍历整个图



这个非连通无向图任何一个点为起点都不能遍历整个图

例1. 公司数量

【问题描述】

在某个城市里住着n个人，现在给定关于 n个人的m条信息（即某2个人认识），假设所有认识（直接或间接认识都算认识）的人一定属于同一个公司。

若是某两人不在给出的信息里，那么他们不认识，属于两个不同的公司。

已知人的编号从1至n。

请计算该城市最多有多少公司。

【输入】

第一行：n（<=10000,人数），

第二行：m（<=100000，信息）

以下若干行：每行两个数：i 和j，中间一个空格隔开，表示i和j相互认识。

【输出】

公司的数量。

city.in	city.out
11 9 1 2 4 5 3 4 1 3 5 6 7 10 5 10 6 10 8 9	3

【数据规模】

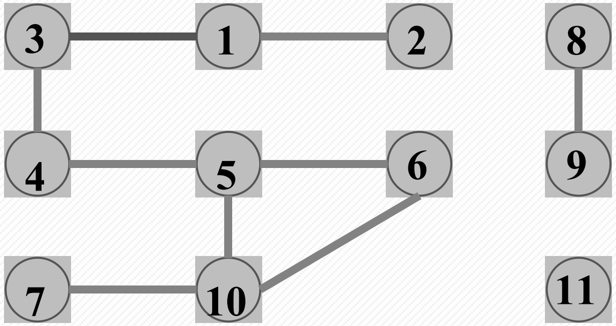
100%的数据：

n<=10000；

m<=100000。

分析：

以人为图的顶点，相互认识的建立无向边，求无向图的连通分量数。



```
#include <cstdio>        //深度优先参考代码
const int maxn=10010;
int a[maxn][maxn];
int vis[maxn];
int n,m,cnt;
void dfs(int k){
		vis[k]=1;
		for(int i=1;i<=n;i++)
			if(!vis[i]&&a[k][i]) dfs(i);
}
int main(){
		scanf("%d%d",&n,&m);
		int x,y;
		for(int i=1;i<=m;i++){
			scanf("%d%d",&x,&y);
			a[x][y]=a[y][x]=1;
		}
		for(int i=1;i<=n;i++)if(!vis[i]){
			dfs(i);
			cnt++;
		}
		printf("%d\n",cnt);
		return 0;
}
```

//优先参考代码

```
#include <cstdio>
const int maxn=10010;
int a[maxn][maxn];
int vis[maxn];
int q[maxn];
int n,m,cnt;
void bfs(int k){
        int head=0,tail=1;
        q[0]=k;
        vis[k]=1;
        while(head<tail){
                int p=q[head++];
                for(int i=1;i<=n;i++){

        if(!vis[i]&&a[p][i]){
        q[tail++]=i;
                                vis[i]=1;
                        }
                }
        }
}
```

```
int main(){
        scanf("%d%d",&n,&m);
        int x,y;
        for(int i=1;i<=m;i++){

        scanf("%d%d",&x,&y);
                a[x][y]=a[y][x]=1;
        }
        for(int i=1;i<=n;i++)if(!vis[i]){
                bfs(i);
                cnt++;
        }
        printf("%d\n",cnt);
        return 0;
}
```

例2. 油田(zoj1709 poj1562)

【题目描述】

GeoSurvComp 地质探测公司负责探测地下油田。每次 GeoSurvComp 公司都是在一块长方形的土地上来探测油田。在探测时，他们把这块土地用网格分成若干个小方块，然后逐个分析每块土地，用探测设备探测地下是否有油田。方块土地底下有油田则称为 pocket，如果两个pocket相邻，则认为是同一块油田，油田可能覆盖多个 pocket。

你的工作是计算长方形的土地上有多少个不同的油田。

【输入描述】

输入文件中包含多个测试数据，每个测试数据描述了一个网格。

每个网格数据的第一行为两个整数：m n，分别表示网格的行和列；如果m = 0，则表示输入结束，否则 1≤m≤100，1 ≤n≤100。

接下来有m 行数据，每行数据有 n 个字符（不包括行结束符）。每个字符代表一个小方块，如果为"*"，则代表没有石油，如果为"@"，则代表有石油，是一个 pocket。

【输出描述】

对输入文件中的每个网格，输出网格中不同的油田数目。如果两块不同的 pocket 在水平、垂直、或者对角线方向上相邻，则被认为属于同一块油田。每块油田所包含的 pocket 数目不会超过 100。

样例输入：	样例输出：
3 5 *@*@* **@** *@*@* 5 5 ****@ *@@*@ *@**@ @@@*@ @@**@ 0 0	1 2

分析：

从网格中某个“@”字符位置开始进行 DFS 搜索，可以搜索到跟该“@”字符位置同属一块油田的所有“@”字符位置。

遍历整个图，求图的连通分量。注意是8连通。

```
//DFS:油田
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
const int maxn=110;
const int dx[8]={-1,1,0,0,-1,1,-1,1};
const int dy[8]={0,0,-1,1,-1,1,1,-1};
int g[maxn][maxn];
int n,m,cnt;
void dfs(int x,int y){
		g[x][y]=0;
		for(int i=0;i<8;i++){
			int xx=x+dx[i],yy=y+dy[i];
			if(xx>=1&&xx<=n&&yy>=1&&yy<=m&&g[xx][yy]==1){
				dfs(xx,yy);
			}
		}
}
```

```
int main(){
        while((cin>>n>>m)&&m){
                char ch;
                memset(g,0,sizeof(g));
                for(int i=1;i<=n;i++)
                for(int j=1;j<=m;j++){
                        cin>>ch;
                        g[i][j]=(ch=='@'?1:0);
                }
                int cnt=0;
                for(int i=1;i<=n;i++)
                for(int j=1;j<=m;j++)
                        if(g[i][j]==1){
                                dfs(i,j);
                                cnt++;
                        }
                cout<<cnt<<endl;
        }
 return 0;
}
```

```
//BFS:油田
#include <cstdio>  #include <cstring>  #include <iostream>
using namespace std;
const int maxn=110;
const int dx[8]={-1,1,0,0,-1,1,-1,1};
const int dy[8]={0,0,-1,1,-1,1,1,-1};
int g[maxn][maxn], n,m,cnt;
struct node{int x,y;}cur,nxt;
node q[maxn*maxn];
void bfs(int x,int y){
        int head=0; tail=1;
        g[x][y]=0; q[0].x=x;  q[0].y=y;
        while(head<tail){
                cur=q[head++];
                for(int i=0;i<8;i++){
                        int xx=cur.x+dx[i]; int yy=cur.y+dy[i];

        if(xx>=1&&xx<=n&&yy>=1&&yy<=m&&g[xx][yy]==1){
                                g[xx][yy]=0;       nxt.x=xx; nxt.y=yy;
                                q[tail++]=nxt;
                        }
                }
        }
}
```

```
int main(){
	while((cin>>n>>m)&&m){
		char ch;
		memset(g,0,sizeof(g));
		for(int i=1;i<=n;i++)
		for(int j=1;j<=m;j++){
			cin>>ch;
			g[i][j]=(ch=='@'?1:0);
		}
		int cnt=0;
		for(int i=1;i<=n;i++)
		for(int j=1;j<=m;j++)
			if(g[i][j]==1){
				bfs(i,j);
				cnt++;
			}
		cout<<cnt<<endl;
	}
	return 0;
}
```

例3. 细胞cell

【题目描述】

一矩形阵（n*m）列由数字0到9组成,数字1到9代表细胞,细胞的定义为沿细胞数字上下左右还是细胞数字则为同一细胞,求给定矩形阵列的细胞个数。（细胞数字指1到9）

【输入】

第一行输入n和m(n,m<20)

第二行开始为该矩阵

【输出】

一共有的细胞个数。

样例输入：

```
4 10
0234500067
1034560500
2045600671
0000000089
```

样例输出：

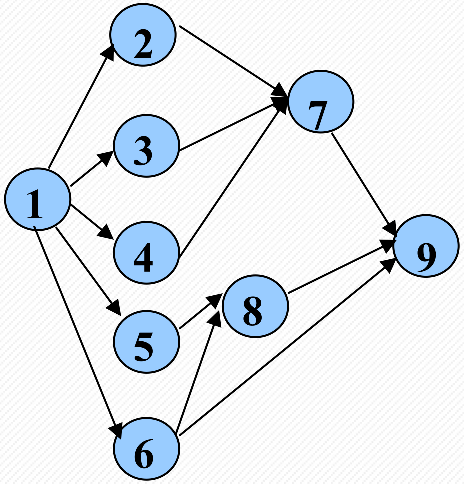
```
4
```

```
#include<string.h>
#define MaxN 1000000
const int gx[4]={0,0,1,-1};const int gy[4]={1,-1,0,0};int BFS_IAA_(int,int);
char map[1000][1000];
int qx[MaxN],qy[MaxN];
int main(void)
{
        int n,m,ans=0;
        scanf("%d%d\n",&n,&m); memset(map,'0',sizeof(map));
        for(int i=1;i<=n;i++)
                {
                        for(int j=1;j<=m;j++)
                                map[i][j]=getchar();
                        getchar();
                }
        for(int i=1;i<=n;i++)
                for(int j=1;j<=m;j++)
                        if (map[i][j]!='0')
                                {
                                        BFS_IAA_(i,j);
                                        ans++;
                                }
        printf("%d\n",ans);
        return 0;
}
```

```
int BFS_IAA_(int x,int y)
{
	int head=1,tail=0,tx,ty,count=0;
	map[x][y]='0';	qx[head]=x;	qy[head]=y;
	do
	{
		count++;		tail++;
		tx=qx[tail];
		ty=qy[tail];
		for (int i=0;i<4;i++)
		{
				x=tx+gx[i];	y=ty+gy[i];
				if ('0'==map[x][y]) continue;
				map[x][y]='0';
				head++;
				qx[head]=x;	qy[head]=y;
		}
	}
	while(head!=tail);
	return count;
}
```

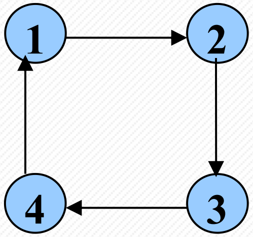
AOV网

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，我们把一条有向边起点的活动成为终点活动的前驱活动，同理终点的活动称为起点活动的后继活动。而只有当一个活动全部的前驱全部都完成之后，这个活动才能进行。例如在上图中，只有当工程1完成之后，工程2、3、4、5、6才能开始进行。只有当2、3、4全部完成之后，7才能开始进行。

一个AOV网必定是一个有向无环图，即不应该带有回路。否则，会出现先后关系的自相矛盾。



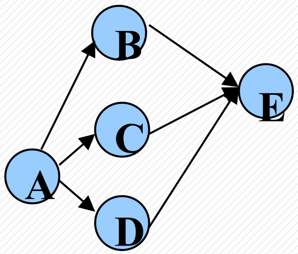
上图就是一个出现环产生自相矛盾的情况。4是1的前驱，想完成1，必须先完成4。3是4的前驱，而2是3的前驱，1又是2的前驱。最后造成想完成1，必须先完成1本身，这显然出现了矛盾。

拓扑排序算法

拓扑排序算法，只适用于AOV网（有向无环图）。

把AOV网中的所有活动排成一个序列， 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”。

一个AOV网的拓扑序列是不唯一的，例如下面的这张图，它的拓扑序列可以是：ABCDE，也可以是ACBDE，或是ADBCE。在下图所示的AOV网中，工程B和工程C显然可以同时进行，先后无所谓；但工程E却要等工程B、C、D都完成以后才能进行。



构造拓扑序列可以帮助我们合理安排一个工程的进度，由AOV网构造拓扑序列具有很高的实际应用价值。

算法思想：

构造拓扑序列的拓扑排序算法思想很简单：

(1) 选择一个入度为0的顶点并输出。

(2) 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。

(3) 重复上述两步，直到不存在入度为0的顶点为止。

(4) 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。

从第四步可以看出，拓扑排序可以用来判断一个有向图是否有环。只有有向无环图才存在拓扑序列。

算法实现：

(a) 数据结构：indgr[i]: 顶点i的入度;

stack[ ]: 栈

(b) 初始化:top=0 (栈顶指针置零)

(c) 将初始状态所有入度为0的顶点压栈

(d) i=0 (计数器)

(e) while 栈非空(top>0)

i. 栈顶的顶点v出栈；top-1; 输出v；i++;

ii. for v的每一个后继顶点u

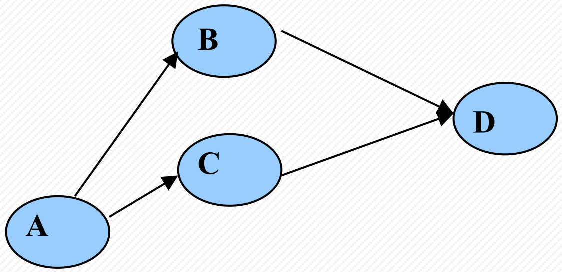
1. indgr[u]--; u的入度减1

2. if（u的入度变为0） 顶点u入栈

(f) 算法结束

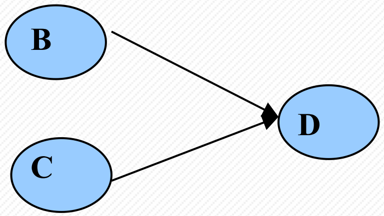
这个程序采用栈来找出入度为0的点，栈里的顶点，都是入度为0的点。

我们结合下图详细讲解：



开始时，只有A入度为0，A入栈。

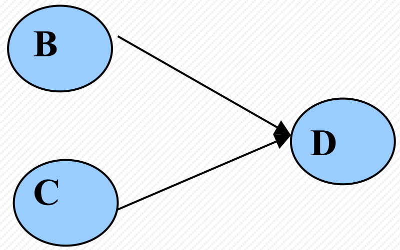
栈：A



栈顶元素A出栈并输出A，A的后继B、C入度减1，相当于删除A的所有关联边

拓扑序列：A

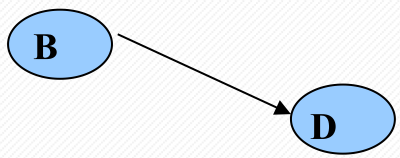
栈：空



B、C的入度都变成0，依次将B、C入栈。

拓扑序列：A

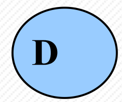
栈：BC（入栈顺序不唯一）



栈顶元素C出栈并输出C，C的后继D入度减1

拓扑序列：AC

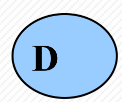
栈：B



栈顶元素B出栈并输出B，B的后继D入度再减1。这时D入度为0，入栈。

拓扑序列：ACB

栈：D



栈顶元素D出栈并输出D。栈空，结束。

拓扑序列：ACBD（不唯一）

栈：空

简单&高效&实用的算法。上述实现方法复杂度O(V+E)。

例4. 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。

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】

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】

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
```

```
#include<cstdio>
#include<iostream>
using namespace std;
int a[101][101],c[101],r[101],ans[101];
int i,j,tot,temp,num,n,m;
int main()
{
    cin >> n;
    for (i = 1; i <= n; i++)
    {
      do
      {
        cin >> j;
        if (j !=0 )
        {
          c[i]++;//c[i]用来存i的出度
          a[i][c[i]] = j;
          r[j]++;//r[j]用来存j的入度
        }
      } while (j != 0);
    }
```

```
for (i = 1; i <= n; i++)
  if (r[i] == 0)
    ans[++tot] = i;//把图中所有入度为0的点入栈，栈用一维数组ans[]表示
do
{
    temp = ans[tot];
    cout << temp << " ";
    tot--;num++;//栈顶元素出栈输出
    for (i = 1; i <= c[temp]; i++)
    {
       r[a[temp][i]]--;
       if (r[a[temp][i]] == 0) //如果入度减1后变成0，则将这个后继点入栈
       ans[++tot] = a[temp][i];
    }
}while (num != n);//如果输出的点的数目num等于n，说明算法结束
return 0;
}
```

最短路径

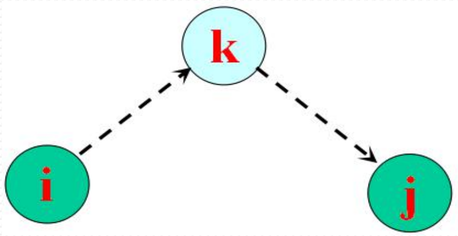
最短路问题（short-path problem）是网络理论解决的典型问题之一，可用来解决管路铺设、线路安装、厂区布局和设备更新等实际问题。基本内容是：若网络中的每条边都有一个数值（长度、成本、时间等），则找出两节点（通常是源节点和阱节点）之间总权和最小的路径就是最短路问题。

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（dijkstra算法、 Bellman-ford 算法、SPFA算法）；另一类是求图中每一对顶点间的最短路径（弗洛伊德算法f loyed）。

Floyd算法

求任意一对顶点之间的最短路径。时间复杂度为**O(n³)**，适用于负边权的情况。

原理：如果我们已经知道了图中任意两点间只允许以编号<=k-1的点作为中转时的最短路，能不能以此推出任意两点间只允许以编号<=k的点作为中转时的最短路呢？



if (d[i][k]+d[k][j]<d[i][j]) d[i][j]=d[i][k]+d[k][j]

算法实现：

```
for (int k=1;k<=n;k++)
    for (int i=1;i<=n;i++)
        for (int j=1;j<=n;j++)
            if (d[i][k]+d[k][j]<d[i][j])
                d[i][j]=d[i][k]+d[k][j];
```

初始化条件：

d[i][i]=0; //自己到自己为0；对角线为0；

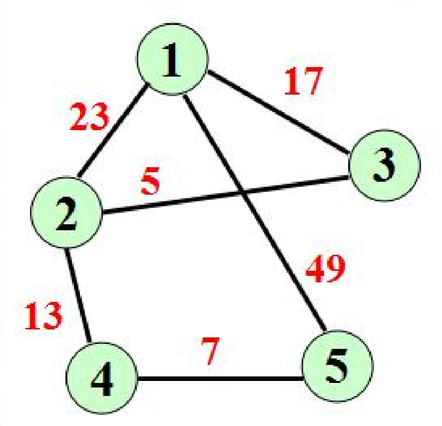
d[i][j]=边权，i与j有直接相连的边

d[i][j]= +∞，i与j无直接相连的边。

// 如果是int数组，采用memset(d, 0x7f, sizeof(d))可全部初始化为一个很大的数

例5. 已知下图中给定的关系，顶点个数n<=100，m条边，两点之间的距离w<=1000,求给定两点p、q之间的最短距离。输入数据，第一行4个整数n m p q，接下来有m行，每行三个数依次描述了一条边的起点、终点和权值。

```
5 6 1  5
1  2  23
1  3  17
1  5  49
2  3  5
2  4  13
4  5  7
```



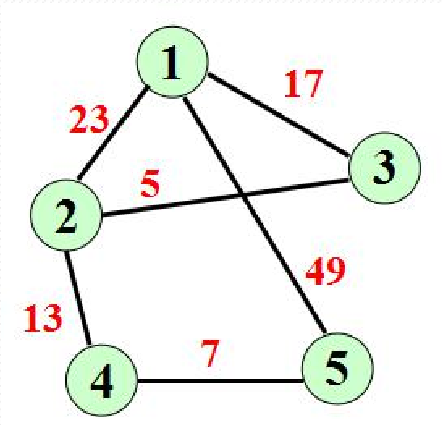
要求：输出最短距离d[p][q]。

分析：D[i][j]表示顶点i到顶点j之间的最短距离。初始化如下：

0	23	17	∞	49
23	0	5	13	∞
17	5	0	∞	∞
∞	13	∞	0	7
49	∞	∞	7	0

floyed →

0	22	17	35	42
22	0	5	13	20
17	5	0	18	25
35	13	18	0	7
42	20	25	7	0



初始状态:

0	23	17	∞	49
23	0	5	13	∞
17	5	0	∞	∞
∞	13	∞	0	7
49	∞	∞	7	0

k=1:

0	23	17	∞	49
23	0	5	13	72
17	5	0	∞	66
∞	13	∞	0	7
49	72	66	7	0

k=2:

0	23	17	36	49
23	0	5	13	72
17	5	0	18	66
36	13	18	0	7
49	72	66	7	0

k=3:

0	22	17	35	49
22	0	5	13	71
17	5	0	18	66
35	13	18	0	7
49	71	66	7	0

k=4:

0	22	17	35	42
22	0	5	13	20
17	5	0	18	25
35	13	18	0	7
42	20	25	7	0

终止状态:

0	22	17	35	42
22	0	5	13	20
17	5	0	18	25
35	13	18	0	7
42	20	25	7	0

参考代码:

```
#include<iostream>
#include<cstring>
using namespace std;
const int maxn=101; const int maxw=100001;
int d[maxn][maxn];  int n,m,p,q;
void init(){
        int i,j,k;
        cin>>n>>m>>p>>q;
        for(i=1;i<=n;i++)
          for(j=1;j<=n;j++)
            if  (i==j) d[i][j]=0;
            else d[i][j]=maxw;
        for(int t=1;t<=m;t++){
          cin>>i>>j>>k;  d[i][j]=k;d[j][i]=k;
        }
}
```

```
void floyed(){
  for(int k=1;k<=n;k++)
   for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
       if (d[i][k]+d[k][j]<d[i][j])
          d[i][j]=d[i][k]+d[k][j];
}
void print(){
        cout<<d[p][q]<<endl;
}
int main(){
        init();
        floyed();
        print();
        return 0;
}
```

判断图中的两点是否连通

算法实现：

把相连的两点间的距离设为dis[i][j]=true,不相连的两点设为dis[i][j]=false，用Floyed算法的变形：

```
for (k = 1; k <= n; k++)
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      dis[i][j] = dis[i][j] || (dis[i][k] && dis[k][j]);
```

最后如果dis[i][j]=true的话，那么就说明两点之间有路径连通。

有向图与无向图都适用。

Dijkstra算法

- **Dijkstra算法思想：如果图是不带负权的有向图或者无向图，从起点v0每次新扩展一个距离最短的点，再以这个点为中间点，更新起点到其他所有点的距离。**
- **由于所有边权都为正，故不会存在一个距离更短的没被扩展过的点，所以这个点的距离永远不会再被改变，因而保证了算法的正确性。**
- **Dijkstra算法步骤：**
- **①初始化d[v0]=0，源点到其他点的距离值d[i]=∞。**
- **②经过n次如下步骤操作，最后得到v0到n个顶点的最短距离：**
- **A.选择一个未标记的点k并且d[k]的值是最小的；**
- **B.标记点k，即f[k]=1；**
- **C.以k为中间点，修改源点v0到其他未标记点j的距离值d[j]**
- **算法的时间复杂度为O(n^2)，因为每次寻找未标记的结点需要耗时O(n)，可以用堆优化此处，时间复杂度降为O((n+m)logm)。**

数据结构：

f[i]：值为true，已求得最短距离，在集合1中；

值为false，在集合2中。

开始点（源点）：start

d[i]：顶点start到 i的最短距离。

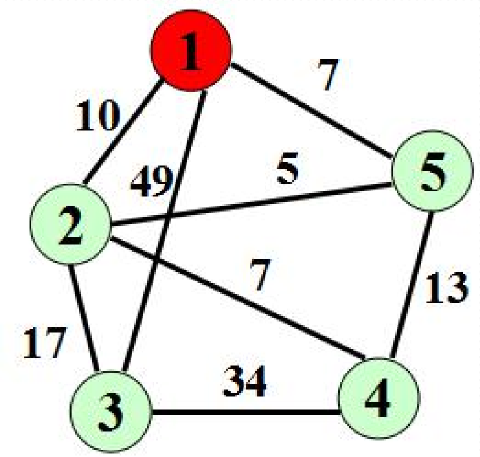
初始：

d[start]=0；

d[i]=a[start][i]；

path[i]：i的前驱结点。

初始表：



顶点	1	2	3	4	5
f[i]	T	F	F	F	F
d[i]	0	10	49	∞	7
path[i]	1	1	1		1

Dijkstra算法：

将顶点分为两个集合：已求得最短距离的点集合1，待求点集合2。

1. 在集合2中找一个到start距离最近的顶点k : min{d[k]}

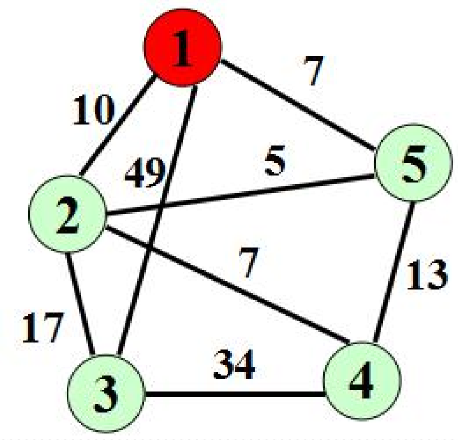
2. 把顶点k加到集合1中，同时修改集合2 中的剩余顶点j的d[j]是否经过k后变短。如果变短修改d[j]

if（d[k]+a[k][j]<d[j]） d[j]=d[k]+a[k][j]

3. 重复1，直至集合2空为止。

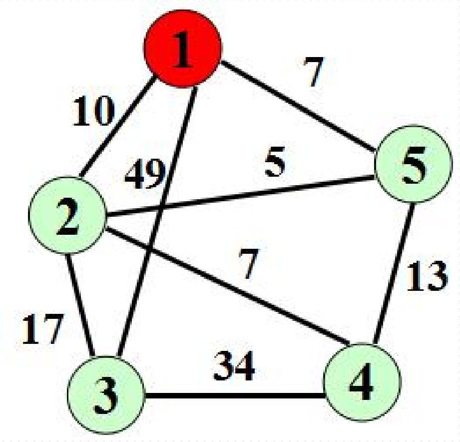
顶点	1	2	3	4	5
f[i]	T	F	F	F	F
d[i]	0	10	49	∞	7
path[i]	1	1	1		1

顶点	1	2	3	4	5
f[i]	T	F	F	F	T
d[i]	0	10	49	20	7
path[i]	1	1	1	5	1



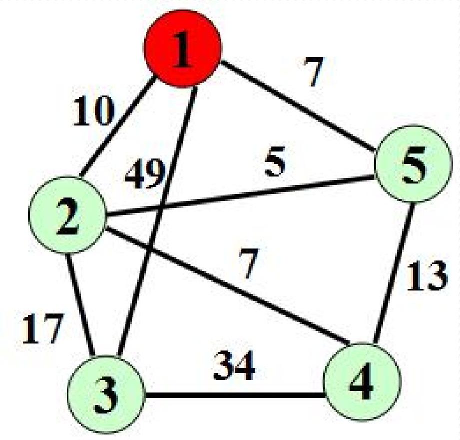
顶点	1	2	3	4	5
f[i]	T	F	F	F	T
d[i]	0	10	49	20	7
path[i]	1	1	1	5	1

顶点	1	2	3	4	5
f[i]	T	T	F	F	T
d[i]	0	10	27	17	7
path[i]	1	1	2	2	1

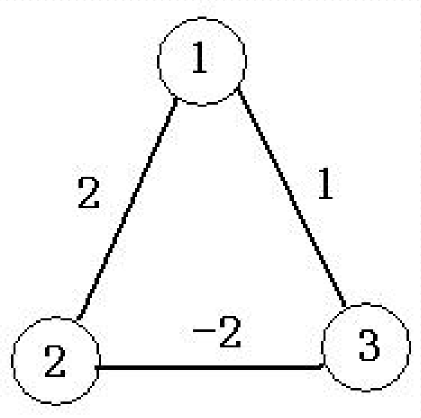


顶点	1	2	3	4	5
f[i]	T	T	F	T	T
d[i]	0	10	27	17	7
path[i]	1	1	2	2	1

顶点	1	2	3	4	5
f[i]	T	T	T	T	T
d[i]	0	10	27	17	7
path[i]	1	1	2	2	1



Dijkstra使用条件：边上的权值不能为负。

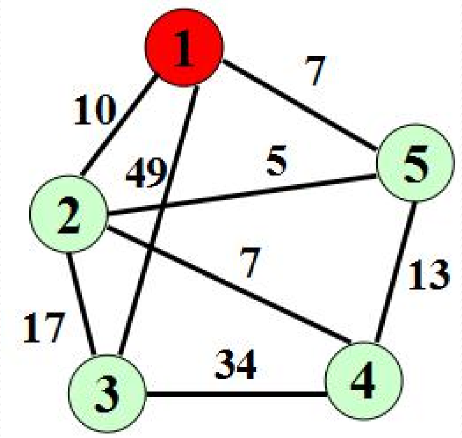


使用Dijkstra算法，得到1--->3的最短路径为1，而实际上是0。

例6. 已知下图中给定的关系，顶点个数n<=100,m条边，两点之间的距离w<=1000,求给定点p到其他点的最短距离。输入数据，第一行3个整数n m p,接下来有m行，每行三个数依次描述了一条边的起点、终点和权值。

【数据输入】

5 8 1
1 2 10
1 5 7
1 3 49
2 3 17
2 4 7
2 5 5
3 4 34
4 5 13



参考代码：

```
#include<iostream>
#include<cstring>
using namespace std;
const int maxn=101;
int f[maxn],a[maxn][maxn],d[maxn],path[maxn];  int n,m,start;
const int inf=99999;
void init(){
		cin>>n>>m>>start;
		int x,y,w;
		for(int i=1;i<=n;i++)
		  for(int j=1;j<=n;j++){
				if(j==i) a[i][j]=0;
				else a[i][j]=inf;
		  }
		for(int i=1;i<=m;i++)
		{	cin>>x>>y>>w;
			a[x][y]=w;
			a[y][x]=w;
		}
}
```

```
void dijkstra(int s){
        for (int i=1;i<=n;i++)  {  d[i]=inf; f[i]=false;  }
        d[s]=0;
        for (int i=1;i<=n;i++)
        {
                int mind=inf;
                int k;//用来记录准备放入集合1的点
                for (int j=1;j<=n;j++)  //查找集合2中d[]最小的点
                   if  ( (!f[j]) &&(d[j]<mind) )
                    { mind=d[j]; k=j; };
                if  (mind==inf)  break; //更新结点求完了
                f[k]=true;   // 加入集合1
                for (int j=1;j<=n;j++)  //修改集合2中的d[j]
                    if ((!f[j]) && (d[k]+a[k][j]<d[j]))
                     { d[j]=d[k]+a[k][j];path[j]=k;}
         }
}
```

```
void dfs(int i){
        if(i!=start) dfs(path[i]);
        cout<<i<<' ';
}
void write(){
    for (int i=1;i<=n;i++)
        if(i!=start) {
                dfs(i);
                cout<<d[i]<<endl;
          }
}
int main(){
        init();
        dijkstra(start);
        write();
        return 0;
}
```

Bellman-Ford算法O(NE)

- 简称Ford（福特）算法，同样是用来计算从一个点到其他所有点的最短路径的算法，也是一种单源最短路径算法。
- 能够处理存在负边权的情况，但无法处理存在负权回路的情况（下文会有详细说明）。
- 算法时间复杂度：O(NE)，N是顶点数，E是边数。
- 算法实现：

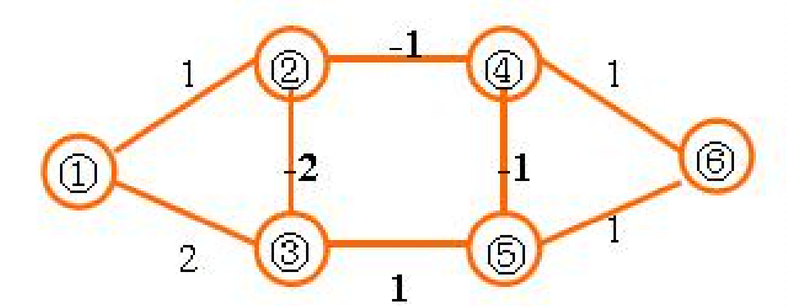
设s为起点，dis[v]即为s到v的最短距离，pre[v]为v前驱。w[j]是边j的长度，且j连接u、v。

初始化：dis[s]=0,dis[v]=∞（v≠s），pre[s]=0

```
for (i = 1; i <= n-1; i++)
for (j = 1; j <= E; j++)        //注意要枚举所有边，不能枚举点。
  if (dis[u]+w[j]<dis[v])          //u、v分别是这条边连接的两个点。
   {
      dis[v] =dis[u] + w[j];
      pre[v] = u;
   }
```

负权回路：

虽然Bellman-Ford算法可以求出存在负边权情况下的最短路径，却无法解决存在负权回路的情况。



存在负权回路的图无法求出最短路径，Bellman-Ford算法可以在有负权回路的情况下输出错误提示。

如果在Bellman-Ford算法的两重循环完成后，还是存在某条边使得：dis[u]+w<dis[v]，则存在负权回路：

for每条边(u,v)

If (dis[u]+w<dis[v]) return false

例5用Bellman-Ford算法实现

```
#include<iostream> #include<cstring> using namespace std;
const int maxn=101; const int maxw=100001;
int d[maxn],f[maxn][3],w[maxn*maxn];  int n,m,p,q;
void init(){
   cin>>n>>m>>p>>q;
   for(int i=1;i<=n;i++)d[i]=maxw;
   for(int i=1;i<=m;i++) cin>>f[i][1]>>f[i][2]>>w[i];
}
void ford(){
   d[p]=0;
   for(int i=1;i<=n;i++)
            for(int j=1;j<=m;j++){
               if (d[f[j][1]]+w[j]<d[f[j][2]]) d[f[j][2]]=d[f[j][1]]+w[j];
               if (d[f[j][2]]+w[j]<d[f[j][1]]) d[f[j][1]]=d[f[j][2]]+w[j];
            }
}
int main(){
   init();
   ford();
   cout<<d[q]<<endl;
   return 0;
}
```

SPFA算法

SPFA是Bellman-Ford算法的一种队列实现，减少了不必要的冗余计算。

主要思想是：

初始时将起点加入队列。每次从队列中取出一个元素，并对所有与它相邻的点进行修改，若某个相邻的点修改成功，则将其入队。直到队列为空时算法结束。

这个算法，简单的说就是队列优化的**bellman-ford**，利用了每个点不会更新次数太多的特点发明的此算法。

SPFA 在形式上和广度优先搜索非常类似，不同的是广度优先搜索中一个点出了队列就不可能重新进入队列，但是**SPFA**中一个点可能在出队列之后再次被放入队列，也就是说一个点修改过其它的点之后，过了一段时间可能会获得更短的路径，于是再次用来修改其它的点，这样反复进行下去。

算法时间复杂度：O(kE)，E是边数。k是常数，平均值为2。

算法实现：

dis[i]记录从起点s到i的最短路径，w[i][j]记录连接i，j的边的长度。pre[v]记录前趋。

team[1..n]为队列，头指针head，尾指针tail。

布尔数组exist[1..n]记录一个点是否现在存在在队列中。

初始化：dis[s]=0,dis[v]=∞（v≠s），memset(exist,false,sizeof(exist));

起点入队team[1]=s; head=0; tail=1;exist[s]=true;

```
do
{
1. 头指针向下移一位，取出指向的点u。
2. exist[u]=false;已被取出了队列
3. for与u相连的所有点v   //注意不要去枚举所有点，用数组模拟邻接表存储
        if (dis[v]>dis[u]+w[u][v])
          {
             dis[v]=dis[u]+w[u][v];
             pre[v]=u;
            if (!exist[v]) //队列中不存在v点，v入队。
              {
                   尾指针下移一位，v入队;
                   exist[v]=true;
              }
          }
}   while (head < tail);
```

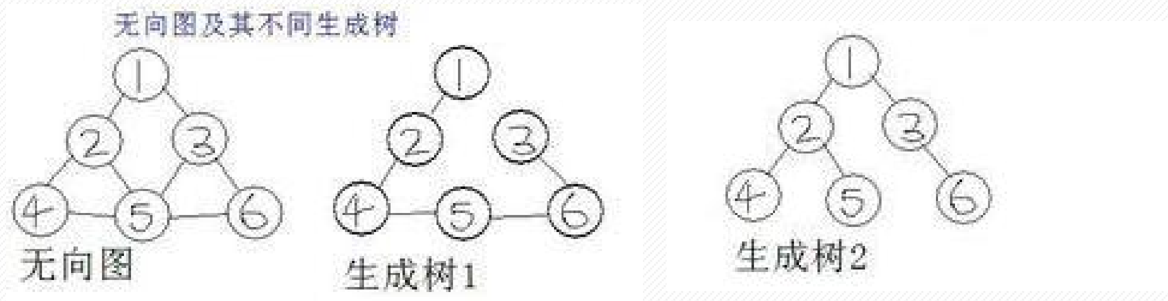
例6用SPFA算法实现

```
#include <iostream>
#include <cstdio>
using namespace std;
#define INF 0x3f3f3f3f
const int maxn=1001,maxm=100001;
int a[maxn][maxn],n,m,s;
int
dis[maxn],pre[maxn],team[maxn],head,tail;
bool f[maxn];
void init(){
        scanf("%d %d %d",&n,&m,&s);
        //读入点数和边数
        int x,y,w;
        for(int i=1;i<=n;i++)
          for(int j=1;j<=n;j++){
                if(j==i) a[i][j]=0;
                else a[i][j]=INF;
          }
        for(int i=1;i<=m;i++){
                cin>>x>>y>>w;
                a[x][y]=w;
                a[y][x]=w;
        }
}
```

```
void dfs(int i){
        if(i!=s) dfs(pre[i]);
        cout<<i<<' ';
}
void write(){
    for (int i=1;i<=n;i++)
        if(i!=s) {
            dfs(i);
            cout<<dis[i]<<endl;
        }
}
```

```
void SPFA(){
        for (int i=1;i<=n;i++) dis[i]=INF;
        dis[s]=0;
        head=0;
        tail=1;
        team[0]=s;
        f[s]=true;
        while(head<tail){
                int u=team[head++];
                f[u]=false;
                for (int v=1;v<=n;v++){
                        if (dis[v]>dis[u]+a[u][v]){
                                dis[v]=dis[u]+a[u][v];
                                pre[v]=u;
                                if (!f[v]){ //队列中不存在v点，v入队
                                        team[tail++]=v; f[v]=true;
                                }
                        }
                }
        }
}
int main(){
   init();
   SPFA();
   write();
   return 0;
}
```

图的最小生成树（MST）



不知道大家还记不记得树的一个定理：**N**个点用**N-1**条边连接成一个连通块，形成的图形只可能是树，没有别的可能。

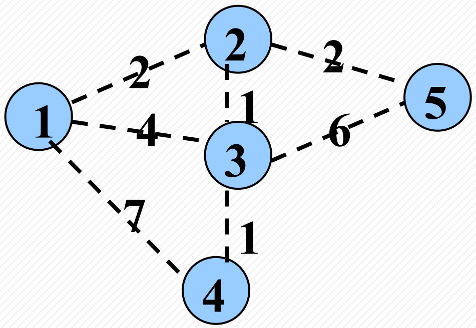
一个有**N**个点的图，边一定是大于等于**N-1**条的。图的最小生成树，就是在这些边中选择**N-1**条出来，连接所有的**N**个点。这**N-1**条边的边权之和是所有方案中最小的。

最小生成树用来解决什么问题？

就是用来解决如何用最小的“代价”用N-1条边连接N个点的问题。

【引例】

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？



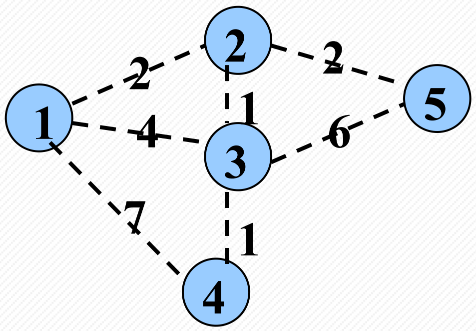
Prim算法

算法分析&思想讲解：

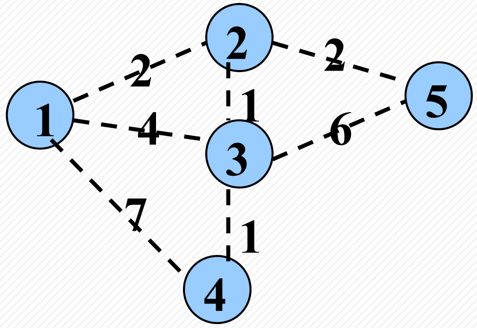
Prim算法采用“蓝白点”思想：白点代表已经进入最小生成树的点，蓝点代表未进入最小生成树的点。

Prim算法每次循环都将一个蓝点u变为白点，并且此蓝点u与白点相连的最小边权min[u]还是当前所有蓝点中最小的。这样相当于向生成树中添加了n-1次最小的边，最后得到的一定是最小生成树。

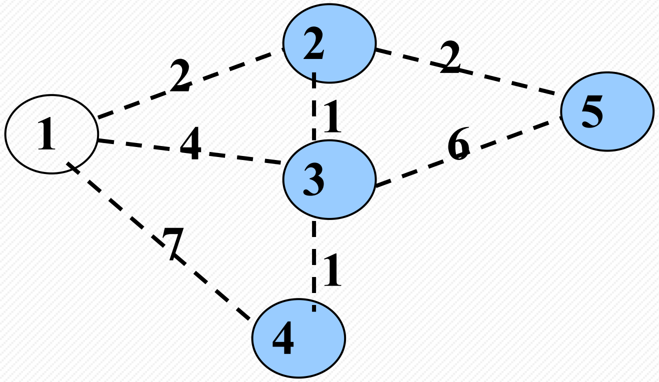
我们通过对右图最小生成树的求解模拟来理解上面的思想。蓝点和虚线代表未进入最小生成树的点、边；白点和实线代表已进入最小生成树的点、边。



初始时所有点都是蓝点，min[1]=0,min[2、3、4、5]=∞。权值之和MST=0。

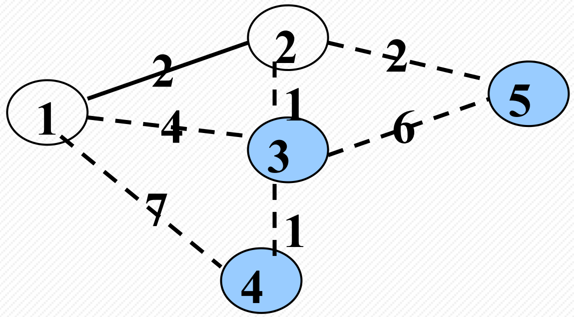


第一次循环自然是找到min[1]=0最小的蓝点1。将1变为白点，接着枚举与1相连的所有蓝点2、3、4，修改它们与白点相连的最小边权。



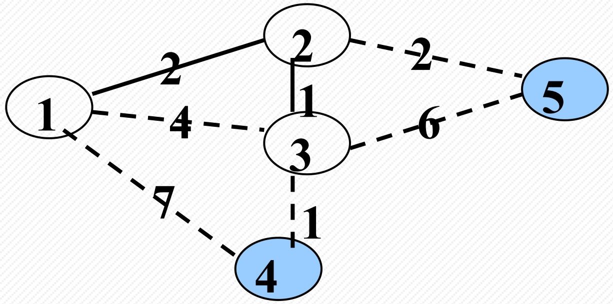
min[2]=w[1][2]=2;
min[3]=w[1][3]=4;
min[4]=w[1][4]=7;

第二次循环是找到min[2]最小的蓝点2。将2变为白点，接着枚举与2相连的所有蓝点3、5，修改它们与白点相连的最小边权。



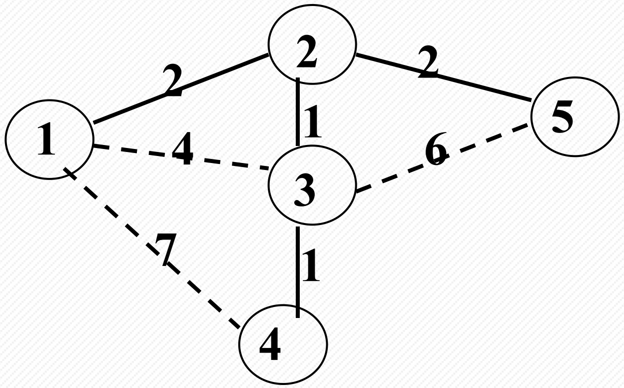
min[3]=w[2][3]=1;
min[5]=w[2][5]=2;

第三次循环是找到min[3]最小的蓝点3。将3变为白点，接着枚举与3相连的所有蓝点4、5，修改它们与白点相连的最小边权。



min[4]=w[3][4]=1;
由于min[5]=2 < w[3][5]=6;所以不修改min[5]的值。

最后两轮循环将点4、5以及边w[2][5],w[3][4]添加进最小生成树。



最后权值之和MST=6。

这n次循环，每次循环我们都能让一个新的点加入生成树，n次循环就能把所有点囊括到其中；每次循环我们都能让一条新的边加入生成树，n-1次循环就能生成一棵含有n个点的树；每次循环我们都取一条最小的边加入生成树，n-1次循环结束后，我们得到的就是一棵最小的生成树。

这就是Prim采取贪心法生成一棵最小生成树的原理。
算法时间复杂度：O (N^2)。

算法描述：
以1为起点生成最小生成树，min[v]表示蓝点v与白点相连的最小边权。
MST表示最小生成树的权值之和。
(a) 初始化：min[v]= ∞(v≠1); min[1]=0;MST=0;
(b) for (i = 1; i<= n; i++)
1. 寻找min[u]最小的蓝点u。
2. 将u标记为白点
3. MST+=min[u]
4. for 与白点u相连的所有蓝点v

```
if (w[u][v]<min[v])
    min[v]=w[u][v];
```

(c)算法结束： MST即为最小生成树的权值之和

例7. 最优布线问题 http://www.codevs.cn/problem/1231/

【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）

【参考程序】

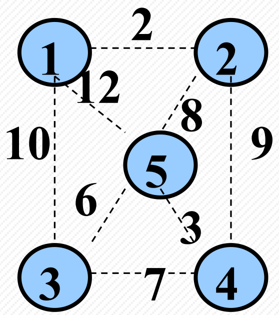
```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int g[101][101];        //邻接矩阵
int minn[101];          //minn[i]存放蓝点i与白点相连的最小边权
bool u[101];            //u[i]=true，表示顶点i还未加入到生成树中
                        //u[i]=false，表示顶点i已加入到生成树中
int n,i,j;
int main()
{
    cin >> n;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            cin >> g[i][j];
    memset(minn,0x7f,sizeof(minn));   //初始化为maxint
    minn[1] = 0;
    memset(u,1,sizeof(u));   //初始化为true，表示所有顶点为蓝点
```

```
for (i = 1; i <= n; i++)
  {
     int k = 0;
     for (j = 1; j <= n; j++)  //找一个与白点相连的权值最小的蓝点k
        if (u[j] && (minn[j] < minn[k]))
           k = j;
     u[k] = false;            //蓝点k加入生成树，标记为白点
     for (j = 1; j <= n; j++)        //修改与k相连的所有蓝点
        if (u[j] && (g[k][j] < minn[j]))
            minn[j] = g[k][j];
  }
  int total = 0;
  for (i = 1; i <= n; i++)           //累加权值
     total += minn[i];
  cout << total << endl;
  return 0;
}
```

知识扩展：本算法在移动通信、智能交通、移动物流、生产调度等物联网相关领域都有十分现实的意义，采用好的算法，就能节省成本提高效率。

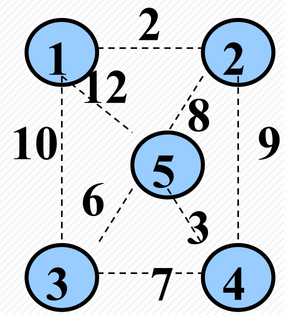
【引例】

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？



Kruskal算法

Kruskal（克鲁斯卡尔）算法是一种巧妙利用并查集来求最小生成树的算法。



Kruskal算法将一个连通块当做一个集合。**Kruskal**首先将所有的边按从小到大顺序排序（一般使用快排），并认为每一个点都是孤立的，分属于**n**个独立的集合。然后按顺序枚举每一条边。如果这条边连接着两个不同的集合，那么就把这条边加入最小生成树，这两个不同的集合就合并成了一个集合；如果这条边连接的两个点属于同一集合，就跳过。直到选取了**n-1**条边为止。

最终求得最小生成树权值为**19**。

通过上面的模拟能够看到，Kruskal算法每次都选择一条最小的，且能合并两个不同集合的边，一张n个点的图总共选取n-1次边。因为每次我们选的都是最小的边，所以最后的生成树一定是最小生成树。每次我们选的边都能够合并两个集合，最后n个点一定会合并成一个集合。通过这样的贪心策略，Kruskal算法就能得到一棵有n-1条边，连接着n个点的最小生成树。

Kruskal算法的时间复杂度为O(E*logE)，E为边数。

算法描述:

1. 初始化并查集。father[x]=x。

2. tot=0;

3. 将所有边用快排从小到大排序。

4. 计数器 k=0;

5. for (i=1; i<=M; i++)　//循环所有已从小到大排序的边

if 这是一条u,v不属于同一集合的边(u,v)(因为已经排序，所以必为最小)

begin

①合并u,v所在的集合，相当于把边(u,v)加入最小生成树。

②tot=tot+W(u,v);

③k++;

④如果k=n-1,说明最小生成树已经生成，则break;

end;

6. 结束，tot即为最小生成树的总权值之和。

例8.最短网络Agri-Net http://www.luogu.org/problem/show?pid=1546

【问题描述】

农民约翰被选为他们镇的镇长！他其中一个竞选承诺就是在镇上建立起互联网，并连接到所有的农场。当然，他需要你的帮助。约翰已经给他的农场安排了一条高速的网络线路，他想把这条线路共享给其他农场。为了用最小的消费，他想铺设最短的光纤去连接所有的农场。你将得到一份各农场之间连接费用的列表，你必须找出能连接所有农场并所用光纤最短的方案。每两个农场间的距离不会超过100000。

【输入格式】

第一行：	农场的个数，N（3<=N<=100）。
第二行..结尾	后来的行包含了一个N*N的矩阵,表示每个农场之间的距离。理论上，他们是N行，每行由N个用空格分隔的数组成，实际上，他们限制在80个字符，因此，某些行会紧接着另一些行。当然，对角线将会是0，因为不会有线路从第i个农场到它本身。

【输出格式】

只有一个输出，其中包含连接到每个农场的光纤的最小长度。

【输入样例】agrinet.in

```
4
0 4 9 21
4 0 8 17
9 8 0 16
21 17 16 0
```

【输出样例】agrinet.out

```
28
```

【参考程序】

```
#include<cstdio>
#include<iostream>
#include<algorithm>                    //sort()需要用到<algorithm>库
using namespace std;
struct point
    {     int x;    int y;     int v;  };   //定义结构类型，表示边
point a[9901];                         //存边
int fat[101];
int n,i,j,x,m,tot,k;
int father(int x)
{
   if (fat[x] != x) fat[x] = father(fat[x]);
   return fat[x];
}
void unionn(int x,int y)
{
   int fa = father(x);
   int fb = father(y);
   if (fa != fb) fat[fa] = fb;
}
```

```
int cmp(const point &a,const point &b)      //sort()自定义的比较函数
{
     if (a.v < b.v) return 1;
       else return 0;
}
int main()
{
   cin >> n;
   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
      {
         cin >> x;
         if (x != 0)
         {
            m++;
            a[m].x = i; a[m].y = j; a[m].v = x;
         }
      }
```

```
for (i = 1; i <= n; i++) fat[i] = i;
    sort(a+1,a+m+1,cmp);        //C++标准库中自带的快排
        //cmp为自定义的比较函数。表示a数组的1-m按规则cmp排序

for (i = 1; i <= m; i++)
    {
        if (father(a[i].x) != father(a[i].y))
        {
            unionn(a[i].x,a[i].y);
            tot += a[i].v;
            k++;
        }
        if (k == n-1) break;
    }
    cout << tot;
    return 0;
}
```

Thanks for listening！