**王道上机总结**

在vs环境下，在开始部分插入#pragma warning (disable:4996)可以直接使用scanf

复杂度问题估计：

acm中每秒运行不超过千万次，

所以n\*n的问题不能够超过3000

需要开辟较大的数据时，使用全局变量，或者new，malloc

codeblocks的使用时，建工程一定要建成英文，否则无法调试

快排的使用：

#include<algorithm>

using namespace std;

普通数组

bool cmp(int a,int b){

    return a>b;//降序

}

sort(buf,buf+n);//升序

sort(buf,buf+n,cmp);

结构体数组：

方法一：

typedef struct Stu{

  char name[1000];

  int age;

  int grade;

  bool operator <(const Stu&b)const{

   if(grade!=b.grade) return grade<b.grade;

  int temp=strcmp(name,b.name);

  if(temp!=0) return temp<0;

  else

      return age<b.age;

  }

}Stu;

bool cmp(Stu a,Stu b){

  if(a.grade!=b.grade) return a.grade<b.grade;

  int temp=strcmp(a.name,b.name);

  if(temp!=0) return temp<0;

  else

      return a.age<b.age;

}

日期求差值问题：

同一按某一基准0算

   void nextday(){

           day++;

        if(day>monthdays[month][flag])

        {

            day=1;

            month++;

            if(month>12)

            {

                month=1;

                year++;

                flag=isYeap(year);

            }

        }

       };

计算从1年1月1日开始计算

牺牲内存换取时间

贪心算法：局部最优

括号匹配：入栈，出栈（有时间思考复杂的四则运算加括号合法性判定）

#include<stack>

using namespace std;

stack<类型名，可以结构体> 名称A

A.pop();

A.top();

A.push();

A.size();

简单计算器问题（有时间思考复杂计算器）：

运算栈，数据栈

难点在于定义运算符号等级

串的开始和末尾各加#区别

推荐方法：先将中缀表达式转换为后缀表达式，再进行计算

//获得后缀表达式

string getPost(string str)

{

    string post;

    stack <char> oper;

    int i = 0;

    int length = str.size();

    while (!oper.empty())

        oper.pop();

    for (i = 0; i<length; i++)

    {

      //数字直接入字符串里，不过末尾记得加空格来区分

        if (('0' <= str[i] && str[i] <= '9') || str[i] == '.')

        {

            while (('0' <= str[i] && str[i] <= '9') || str[i] == '.')

            {

                post += str[i++];

            }

            post += ' ';

            i--;//这里要进行还原，防止和for循环里的i++冲突

        }

        if (Isoperator(str[i]))

        {

         //这里当符号栈非空且栈顶优先级大于当前运算符则出栈，加入到字符串里

            while (!oper.empty() && Isoperator(oper.top()) && (priorit(oper.top()) >= priorit(str[i])))

            {

                post += (oper.top());

                post += (' ');

                oper.pop();

            }

            oper.push(str[i]);

        }

     //左括号直接入栈

        if (str[i] == '(')

            oper.push(str[i]);

   //右括号则出栈直到遇到左括号

        if (str[i] == ')')

        {

            while (!oper.empty() && oper.top() != '(')

            {

                post += (oper.top());

                post += (' ');

                oper.pop();

            }

            oper.pop();//删除左括号

        }

    }

   最后将剩下的符号栈内容全出栈

    while (oper.size()>1)

    {

        post += (oper.top());

        post += (' ');

        oper.pop();

    }

    post += (oper.top());

    oper.pop();

    return post;

}

//通过后缀表达式来计算

double getValue(string str, bool &flag)

{

    stack <double> num;

    double ans = 0;

    double a, b;

    int i;

    int length = str.size();

    while (!num.empty())

        num.pop();

    for (i = 0; i<length; i++)

    {

        if (('0' <= str[i] && str[i] <= '9') || str[i] == '.')

        {

            string temp;

            while (('0' <= str[i] && str[i] <= '9') || str[i] == '.')

            {

                temp += str[i++];

            }

            i--;//防止越界

            num.push(strToDouble(temp));

        }

        if (str[i] == ' ')

            continue;

//  数字出栈的时候先出b，再出的a

        if (Isoperator(str[i]))

        {

            if (num.size() >= 2)

            {

                b = num.top();

                num.pop();

                a = num.top();

                num.pop();

                if (str[i] == '+')

                {

                    num.push(a + b);

                }

                if (str[i] == '-')

                {

                    num.push(a - b);

                }

                if (str[i] == '\*')

                {

                    num.push(a\*b);

                }

                if (str[i] == '/')

                {

                    if (b != 0)

                        num.push(a / b);

                    else

                    {

                        flag = false;

                        return ans;

                    }

                }

            }

                else

                {

                    flag = false;

                    return ans;

                }

        }

    }

    if (1!=num.size())

    {

        flag = false;

        return ans;

    }

    else

    {

        flag = true;

        ans = num.top();

        num.pop();

        return ans;

    }

}

哈夫曼树

#include<queue>

using namespace std;

queue<类型>  名称

优先队列

priority\_queue<类型名称> 名称 //大顶堆

priority\_queue<int,vector<int>，greater<int>这个有个空格，非常重要，不然就成了“》操作符号”> 名称//小顶堆

priority\_queue<int>默认大顶堆

对于结构体，要重载操作符<

typedef struct Int{

   int i;

   bool operator <(const Int &b)const{

   return i>b.i;  //注意这里，要改为大于，不然又是默认的大顶堆

   }

}Int;

操作：

记得之前要清空堆

   while(Huffman.empty()!=true){

        Huffman.pop();

    a=Huffman.top();.//取堆顶元素

        Huffman.pop();//删除堆顶元素

        b=Huffman.top();

        Huffman.pop();

        result+=a+b;

求最小带权路径和

//不要建树

ans+=a+b；

queue.push(a+b);

二叉树遍历问题：

在使用结构体时，用new/malloc分配内存，不要直接声明结构体，否则函数结束后，结构体作为临时变量被释放后

然后就GG了

由前序中序求后序：

后序遍历

void travel(Node \* T){

   if(T->lchild!=NULL)

       travel(T->lchild);

   if(T->rchild!=NULL)

       travel(T->rchild);

    printf("%c",T->value);

}

由前序中序建树的过程：

Node \* createTree(char first[],char middle[],int s1,int e1,int s2,int e2){

    Node\*  pnode=new Node;

    int i;

    pnode->value = first[s1];

    int rootId;

    for(i=s2;i<=e2;i++){

        if(pnode->value==middle[i]){

             rootId=i;

             break;

          }

    }

    if(rootId!=s2){

        pnode->left=createTree(first,middle,s1+1,s1+rootId-s2,s2,rootId-1);

    }

    else

        pnode->left=NULL;

    if(rootId!=e2){

        pnode->right=createTree(first,middle,s1+rootId-s2+1,e1,rootId+1,e2);

    }

    else

        pnode->right=NULL;

    return pnode;

}

二叉排序树又叫二叉搜索树，左子树的值小于根节点，根节点小于右子树的值

二叉排序树的中序遍历得到一个升序的有序序列

创建二叉排序树：

Node\* Insert(Node\* tree,int value){

   if(tree==NULL){

    Node\* pnode=NULL;

     tree= pnode= new Node;

     pnode->value=value;

     pnode->left=NULL;

     pnode->right=NULL;

     return tree;

   }

   else{

        if(value<tree->value){

          tree->left = Insert(tree->left,value);

        }

        else if(value>tree->value){

           tree->right=Insert(tree->right,value);

        }

        return tree;

   }

}

一个调试技巧：

当断点调试遇到报错，无法进入函数内部，那么可以考虑参数的值是否合法

取模问题：

a%b等于a的绝对值取余，符号与a相同

如果担心余数是负数，可以在加余数加上一个b再取余

int最大值。约等于4后面9个0，大概10位数可以解决

进制转换问题

do{

   ans[size++]=a%m;

  a/=m;

}while(a!=0)

do while，这样a==0时也可以处理，

a为要转换的十进制，

m为几进制

几进制转几进制：

先转换为10进制，再转换为需要的进制

  lengthN=strlen(n);//n为字符数组，类似7Ab3,num为其对应的十进制数

        num=0;

        c=1;

        size=0;

        for(i=lengthN-1; i>=0; i--)

        {

            temp=0;

            if('0'<=n[i]&&n[i]<='9')

                temp=n[i]-'0';

            if('a'<=n[i]&&n[i]<='f')

                temp=n[i]-'a'+10;//这里注意要加上10

            if('A'<=n[i]&&n[i]<='F')

                temp=n[i]-'A'+10;

              num+=temp\*c;

               c\*=a;

        }

             do

        {

            x=num%b;

            num/=b;

          ans[size++]=x>9?x+'A'-10:x+'0';

        }while(num!=0);//将对应的十进制转换为相应的进制。

求最大公约数gcd：

欧几里得算法：

对a、b求最大公约数，若a、b均为0，则最大公约数不存在，若其中一个为0，则返回另外一个非0值

两个非0，则等价于b,a%b求最大公约数

int gcd(int a,int b){

 if(b==0)

  return a;

  return  gcd(b,a%b);

}

求最小公倍数lcm：

结论：

     两者乘积除以最大公倍数

int gcd(int a,int b)

{

  if(b==0)

    return a;

  else

    return gcd(b,a%b);

}

int lcm(int a,int b)

{

  return a\*b/gcd(a,b);

 }

素数判定：

bool isPrime(int x){

    int i,temp;

 if(x<=1)

    return false;

 temp=(int)sqrt(x)+1;

   for(i=2;i<temp;i++)

 {

    if(x%i==0)

        return false;

 }

  return true;

}

素数筛选问题：

输出从1到n的素数

如果一个一个判定，太花时间：

筛选方法，从2开始，删除2的倍数，mark标记为true；

然后找到下个mark为false的数i，删除i的倍数

直到n-1；

  for(i=0;i<10000+1;i++)

            mark[i]=false;

        for(i=2;i<n;i++)

        {

            if(mark[i]==true)

            {

                continue;

            }

            mark[i]=true;

            prime[count++]=i;

            for(j=2\*i;j<n;j=j+i)

                mark[j]=true;

        }

输出格式要求技巧：

要求最后一个不带空格：

那么第一个直接输出，以后每个数据输出为空格加数据：

first初始为true；

 if(first)

              {

                   first=false;

               cout<<prime[i];

               }

            else{

                cout<<" "<<prime[i];

            }

分解素数：

将120分成2\*2\*2\*3\*5

求n

思路先求出小于sqrt（n）的素数

然后用n对这些素数求模，一个数可以不断的求，知道化为1

如果n到最后都没化为1，那么，一定存在一个大因子为化简后的n

算法如下：

  ansSize=0;

     count=0;

        for(i=0; i<primeSize; i++)//primeSize为素数个数

        {

            if(n%prime[i]==0)

            {

                ansPrime[ansSize]=prime[i];

                ansNum[ansSize]=0;

                while(n%prime[i]==0)

                {

                    ansNum[ansSize]++;

                    n/=prime[i];

                }//相同素数处理

                ansSize++;

            }

            if(n==1)

                break;

        }

        if(n!=1)

        {

            ansPrime[ansSize]=n;//ansNum[ansSize]为结果的素数，ansNum为该素数的幂指数

            ansNum[ansSize++]=1;

        }

        //错误做法，例如200006其因子为2和100003，这个时候n!=0;ansSize=1;

        /\*if(n!=1)

        {

            ansPrime[0]=n;

             ansNum[0]=1;

             ansSize++;

        }

        \*/

整除的概念

A/B=整数

B能整除A

A能被B整除

二分求幂

int pow1(int a,int b)

{

   int ans=1;

   if(a==0&&b==0)

    return 0;

   while(b!=0)

    {

      if(b%2==1)

       {

        ans\*=a;

       }

       a\*=a;

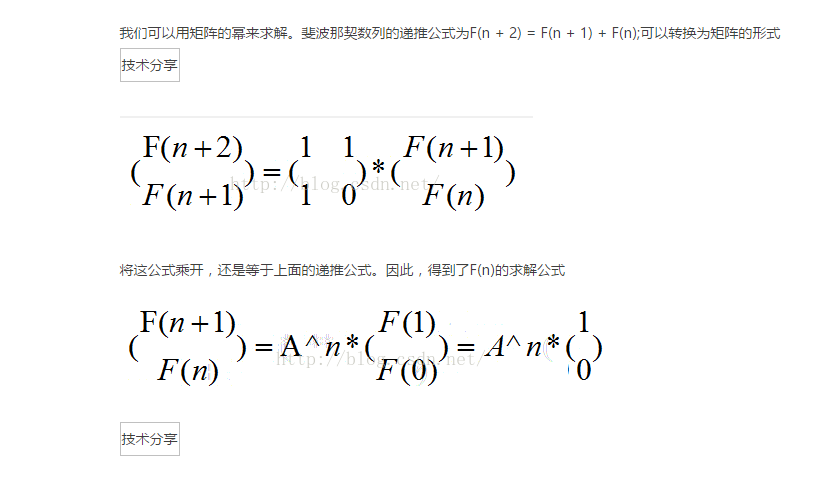
       b/=2;

   }

   return ans;

}

矩阵快速幂及利用矩阵快速幂求斐波那契数



F（n）=A^n次后A【1】【0】的值

A[0][0] = 1; A[0][1] = 1;

A[1][0] = 1; A[1][1] = 0;

A = pows(A, n);

printf("%d\n", A[1][0]);

矩阵快速幂乘法：

Matrix expo(Matrix p,int k)

{

    if(k==1)

    return p;

    Matrix e;

    memset(e.data,0,sizeof(e.data));

    for(int i=0;i<N;++i)

     {e.data[i][i]=1;}

    if(k==0)

    return e;//单位矩阵

    while(k)

    {

        if(k%2==1)

        e=mul(p,e);

        p=mul(p,p);

        k/=2;

    }

    return e;

}

高精度加法、乘法

高精度乘法一般会出现一个大整数与一个小数相乘的情况：

typedef struct bigInteger{

  int size;//长度

  int num[1000];//数据,每4位存放一个int值

  void init()

  {

      int i;

      size=0;

   for(i=0;i<1000;i++)

    num[i]=0;

  }

   void set(char\* str)

   {

       init();

       int length=strlen(str);

       int temp=0;

       int c=1;

       int j=0;

       int i;

       for(i=length-1;i>=0;i--)

       {

           temp = temp+(str[i]-'0')\*c;

           c\*=10;

           j++;

           if(j==4)

           {

            num[size++]=temp;

            temp=0;

            c=1;

            j=0;

           }

       }

       if(0<j&&j<4)

        num[size++]=temp;

   }

   void output()

   {

       int tempSize=size,i;

       for(i=tempSize-1;i>=0;i--)

       {

           if(i==tempSize-1)

            printf("%d",num[i]);

          else

            printf("%04d",num[i]);

       }

       cout<<endl;

   }

   bigInteger operator+(const bigInteger &a)const

   {

       int i,temp;

       bigInteger ret;

       int carry=0;//进位

       ret.init();

       for(i=0;i<size||i<a.size;i++)

          {

             temp=num[i]+a.num[i]+carry;

             carry=temp/10000;

              temp%=10000;

             ret.num[ret.size++]=temp;

          }

        if(carry!=0)

               ret.num[ret.size++]=carry;

        return ret;

   }

 bigInteger operator\*(int x)

   {

       int i,temp;

       bigInteger ret;

       int carry=0;//进位

       ret.init();

       for(i=0;i<size;i++)

          {

             temp=num[i]\*x+carry;

             carry=temp/10000;

              temp%=10000;

             ret.num[ret.size++]=temp;

          }

        if(carry!=0)

               ret.num[ret.size++]=carry;

        return ret;

   }

}bigInteger;

图论相关知识：

常用的两种表达图的数据结构：邻接矩阵和邻接表

邻接矩阵：适合于稠密图，和经常需要判定特定节点是否相连，时间复杂度o（n\*n）

邻接表：适合于稀疏图，和某节点相邻的节点判定，时间复杂度o（n+e）

邻接表使用：

struct Edge{

  int nextNode;

 int cost;//权重

};

#include<vector>

using namespace std;

vector<Edge> edge[N];//n个节点

初始化：

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

 edge[i].clear();

添加:

Edge tmp;

   tmp.cost=38;

   tmp.nextEdge=3;

   edge[2].push\_back(tmp);

//遍历

   for(i=0;i<edge[2].size();i++)

   {

        int nextNode=edge[2][i].nextEdge;

        int cost=edge[2][i].cost;

   }

   //删除edge[1][i]；

    // 删除的第元素的地址   删除最后一个元素地址+1

   edge[1].erase(edge[1].begin()+i,edge[1].begin()+i+1);

并查集：

适合于集合的合并和查找

int Tree[N];

//初始化

void initTree(int Tree[],int n)

{

    int i;

  for(i=1;i<=n;i++)

    Tree[i]=-1;

}

//找根

int findRoot(int Tree[],int x)

{

    int tmp;

    if(Tree[x]==-1)

        return x;

    else{

        tmp=findRoot(Tree,Tree[x]);

        Tree[x]=tmp;

        return tmp;

    }

}

//合并

void mergeTree(int Tree[],int a,int b)

{

    int rootA=findRoot(Tree,a);

    int rootB=findRoot(Tree,b);

    if(rootA!=rootB)

        Tree[rootB]=rootA;

}

畅通问题：

N个城市，有M条路

问还需要修多少条路

看成连通分量的问题

然后求出连通分量个数，利用并查集

最低要修的路的条数就是连通分量个数减一

  while(cin>>n&&n!=0){

        cin>>m;

        initTree(Tree,n);

        while(m--)

        {

            cin>>a>>b;

            mergeTree(Tree,a,b);//合并

        }

        ans=0;

        for(i=1;i<=n;i++)

        {

            if(Tree[i]==-1)

            ans++;

        }

        cout<<ans-1<<endl;

    }

最小生成树问题：

Kruskal算法：

每个节点都是单独的连通分量

先将边排序

然后从小到大，取每条边是否在同一个连通分量，如果不在同一连通分量，则将两个连通分量合并

直到找到n-1条边

时间复杂度为O（eloge）。

prim算法为O（v\*v）

代码如下：

//定义边，重载<符号方便排序

typedef struct Edge{

 int strat;

 int end;

 int cost;

 bool operator<(const Edge &b)const

 {

     return cost<b.cost;

 }

}Edge;

int findRoot(int Tree[],int x)

{

    if(Tree[x]==-1)

        return x;

    else

        {

            int tmp=findRoot(Tree,Tree[x]);

            Tree[x]=tmp;

            return tmp;

        }

}

  for(i=1;i<=n;i++)

            Tree[i]=-1;

        numEdge=n\*(n-1)/2;

        for(i=1;i<=numEdge;i++)

        {

            cin>>start>>end>>cost;

            edge[i].strat=start;

            edge[i].end=end;

            edge[i].cost=cost;

        }

        sort(edge+1,edge+1+numEdge);

        minCost=validEdge=0;

        for(i=1;i<=numEdge;i++)

        {

            int rootA=findRoot(Tree,edge[i].strat);

            int rootB=findRoot(Tree,edge[i].end);

            if(rootA!=rootB)

                {

                    Tree[rootB]=rootA;

                     minCost+=edge[i].cost;

                     validEdge++;

                }

            if(validEdge==n-1)

                break;

        }

        cout<<minCost<<endl;

求连通子图中个数最大的子图的节点数

只需要在加入边的时候更新以每个节点为根的节点个数

  int rootA=findRoot(Tree,a);

        int rootB=findRoot(Tree,b);

        if(rootA!=rootB)

        {

            Tree[rootB]=rootA;

            count[rootA]+=count[rootB];

        }

最短路径：

Floyd算法：

floyd算法是一个经典的动态规划算法。用通俗的语言来描述的话，首先我们的目标是寻找从点i到点j的最短路径。从动态规划的角度看问题，我们需要为这个目标重新做一个诠释（这个诠释正是动态规划最富创造力的精华所在），floyd算法加入了这个概念  Ak(i,j)：表示从i到j中途不经过索引比k大的点的最短路径。

就是从只经过编号为1（或者不经过）的点，更新最短路径，再更新最多经过1,2的最短路径，最后直到所有的路都被

更新

动态规划思想：

动态规划算法**与分治法类似**，其基本思想也是将待求解问题分解成若干个子问题。  
但是经分解得到的子问题**往往不是互相独立**的。不同子问题的数目常常**只有多项式量级**。在用分治法求解时，有些子问题被重复计算了许多次。如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。  
**用一个表来记录所有已经解决的子问题的答案**。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划的基本思想。

时间复杂度：O（n\*n\*n）空间复杂度O（n\*n）

当点数不超过200时，即可使用该算法。

#define Infinite 123123123

//初始化

 for(i=1;i<=n;i++)

        for(j=1;j<=n;j++)

      {

          if(i==j)

            map[i][j]=0;

          else

            map[i][j]=Infinite;

      }

//更新数据

      for(i=1;i<=m;i++)

        {

            cin>>a>>b>>c;

//无向图需要双向更新

if(map[a][b]>c)

                {

                    map[a][b]=c;

                    map[b][a]=c;

                }

        }

//floyd算法

        for(k=1;k<=n;k++)

        for(i=1;i<=n;i++)

        for(j=1;j<=n;j++)

        {

            if(map[i][k]+map[k][j]<map[i][j])

                map[i][j]=map[i][k]+map[k][j];

        }

//结果输出

        cout<<map[1][n]<<endl;

dijistra算法：

O（n\*n）复杂度，用链表vector实现，空间复杂度O（n+e）

贪心的思想，局部最优

算法思想

最短路径的最优子结构性质

   该性质描述为：如果P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，k和s是这条路径上的一个中间顶点，那么P(k,s)必定是从k到s的最短路径。下面证明该性质的正确性。

   假设P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，则有P(i,j)=P(i,k)+P(k,s)+P(s,j)。而P(k,s)不是从k到s的最短距离，那么必定存在另一条从k到s的最短路径P'(k,s)，那么P'(i,j)=P(i,k)+P'(k,s)+P(s,j)<P(i,j)。则与P(i,j)是从i到j的最短路径相矛盾。因此该性质得证。

Dijkstra算法

   由上述性质可知，如果存在一条从i到j的最短路径(Vi.....Vk,Vj)，Vk是Vj前面的一顶点。那么(Vi...Vk)也必定是从i到k的最短路径。为了求出最短路径**，Dijkstra就提出了以最短路径长度递增，逐次生成最短路径的算法**。譬如对于源顶点V0，首先选择其直接相邻的顶点中长度最短的顶点Vi，那么当前已知可得从V0到达Vj顶点的最短距离dist[j]=min{dist[j],dist[i]+matrix[i][j]}。根据这种思路，

假设存在G=<V,E>，源顶点为V0，U={V0},dist[i]记录V0到i的最短距离，path[i]记录从V0到i路径上的i前面的一个顶点。

1.从V-U中选择使dist[i]值最小的顶点i，将i加入到U中；

2.更新与i直接相邻顶点的dist值。(dist[j]=min{dist[j],dist[i]+matrix[i][j]})

3.知道U=V，停止。

注意：

Floyd和dijstra算法均不能处理负权重，

要处理，可以考虑bellman算法

代码实现：

//边的实现

typedef struct Edge{

    int nextNode;

    int dis;

}Edge;

int main(){

   vector<Edge> edge[101];

   int n,m;

   int i,j;

   int a,b,c;

   int Dis[101];

   bool mark[101];

   int start;

   int end;

   int nextNode;

    while(cin>>n>>m&&!(n==0&&m==0))

    {

        start=1;

        end=n;

//初始化

        for(i=1;i<=n;i++)

            {

                edge[i].clear();

                mark[i]=false;

                Dis[i]=-1;

            }

//加入边的信息

        while(m--)

        {

            cin>>a>>b>>c;

            Edge tmp;

            tmp.nextNode=b;

            tmp.dis=c;

            edge[a].push\_back(tmp);

            tmp.nextNode=a;

            edge[b].push\_back(tmp);

        }

        Dis[start]=0;

        mark[start]=true;

        nextNode=start;

//dijistra算法

        for(i=1;i<n;i++)

        {

            for(j=0;j<(int)edge[nextNode].size();j++)

            {

                int tmpNextNode=edge[nextNode][j].nextNode;

                int tmpDis=edge[nextNode][j].dis;

                if(mark[tmpNextNode]==true)

                    continue;

                if(Dis[tmpNextNode]==-1||Dis[tmpNextNode]>Dis[nextNode]+tmpDis)

                    Dis[tmpNextNode]=Dis[nextNode]+tmpDis;

            }

    //找出最小的点然后更新

            int min=123123123;

            for(j=1;j<=n;j++)

            {

                if(mark[j]==true)

                    continue;

                if(Dis[j]==-1)

                    continue;

                if(Dis[j]<min)

                {

                    min=Dis[j];

                    nextNode=j;

                }

            }

            mark[nextNode]=true;

            if(mark[end]==true)

                break;

        }

       cout<<Dis[end]<<endl;

    }

 return 0;

}