克鲁斯卡尔算法（最小生成树）

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
typedef struct node
{
	int x;
	int y;
	int value;
	struct node *next;
}LNode,*linklist;
int cmp(const struct node *p1,const struct node *p2)
{
	return (int)(p1->value-p2->value);
}
LNode edgelist[600000];
int ei=0;
int n,m;
long long int sum=0;
int father[500000];
void make()//初始化，每个节点父节点是他自己 ， 每个父节点是一个集合
{
	int i;
	for(i=0;i<n;i++) father[i]=i;
}

int find_father(int x)//找父节点
{
	if(x!=father[x])
	{
		x=father[x];
		x=find_father(x);
	}
	return x;
}
/*int find_father(int x)//找父节点  并查集路径压缩的小优化（都只要 1 步找到祖先结点）
{
	if(x==father[x])
		return x;
	else
	{
		father[x]=find_father(father[x]);   //父节点设为根节点
		return father[x];                   //返回父节点
	}
```

```
}*/
void Union(int x, int y)//将 x,y 合并到同一个集合
{
    x=find_father(x);
    father[x]=find_father(y);
}
void Kruskal()
{
    int i;
    make();
    for(i=0;i<ei;i++)//将边的顺序按从小到大取出来
    {
        if(find_father(edgelist[i].x)!=find_father(edgelist[i].y))
         //如果两个顶点不在一个集合中，就并到一个集合里，生成树的长度加上这条边
的长度
        {
            Union(edgelist[i].x,edgelist[i].y);  //合并两个顶点到一个集合
            sum=sum+edgelist[i].value;
        }
    }
    return;
}
int main()
{
    int i,j,x,y,z;
    scanf("%d%d",&n,&m);
    for(i=0;i<m;i++)//输入边
    {
        scanf("%d%d%d",&x,&y,&z);
        edgelist[ei].x=x;edgelist[ei].y=y;edgelist[ei].value=z;ei++;
    }
    qsort(edgelist,ei,sizeof(struct node),cmp);//按边的长度排序
    Kruskal();
    printf("%lld",sum);
    return 0;
}
```

拓扑排序 P.S. 对于拓扑排序不唯一的情况，先输出序号大的点，再输出序号小的点。
优先队列和链式向前星

```
#include<iostream>
```

```
#include<algorithm>
#include<queue>
using namespace std;
priority_queue<int> q;
struct node//链式向前星
{
	int from;
	int to;
	int before;//同起点上一条边编号 ， -1 为无
	int value;
}edge[500000];
int head[500000];//以 i 为起点上一条
int in[500000];//入度
int n,m;
void tuopu()
{
	int i,j;
	for(i=n;i>=1;i--)
	{
		if(in[i]==0)
		{
			q.push(i);
			in[i]=-1;
		}
	}
	while(q.empty()!=1)
	{
		int x=q.top();
		q.pop();
		cout<<x<<' ';
		for(j=head[x];j!=-1;j=edge[j].before)//链式向前星访问套路，需要记住
		{
			in[edge[j].to]--;
			if(in[edge[j].to]==0)
			{
				in[edge[j].to]=-1;
				q.push(edge[j].to);
			}
		}
	}
}
int main()
{
	int i,j,x,y;
```

```
    cin>>n>>m;
    for(i=0;i<=n;i++) head[i]=-1;
    for(i=0;i<m;i++)
    {
        cin>>x>>y;
        edge[i].from=x;edge[i].to=y;edge[i].before=head[x];head[x]=i;in[y]++;
    }
    tuopu();
    cout<<"\n";
    return 0;
}
```

floyd 算法：可以有负权边，不能有负权回路（求的是点点之间最小值）

```
#include<stdio.h>
long long int n,m,p;
long long int a[510][510];
int main()
{
    long long int i,j,x,y,z,k;
    for(i=0;i<510;i++)
    {
        for(j=0;j<510;j++)
        {
            if(i==j) a[i][j]=0;
            else a[i][j]=1e9;
        }
    }
    scanf("%lld%lld%lld",&n,&m,&p);
    for(i=0;i<m;i++)
    {
        scanf("%lld%lld%lld",&x,&y,&z);
        if(a[x][y]>z) a[x][y]=z;//可能有重复边
    }
    for(k=1;k<=n;k++)//核心代码
    {
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
            {
                if(a[i][j]>a[i][k]+a[k][j]) a[i][j]=a[i][k]+a[k][j];
            }
        }
```

```
    }
    for(i=0;i<p;i++)
    {
        scanf("%lld%lld",&x,&y);
        if(a[x][y]==1e9) printf("-1\n");
        else printf("%lld\n",a[x][y]);
    }
}
```

最大流问题

```
//网络流 Dinic 算法
#include<iostream>
#include<algorithm>
#include<queue>
#include<cstdio>
#include<cstring>
using namespace std;
struct node//链式向前星
{
    int to;int value;int before;
}edge[10020];
int head[10020],dep[10020];
int inque[10020];
int cnt=1;
int n,m,s,t;
int min(int a,int b)
{
    if(a<b) return a;
    else return b;
}
long long int dfs(int pos,long long int low)//当前位置和最小残量，dfs 用于寻找增广路
{
    long long int out=0;
    if(pos==t)//到达汇点
    {
        return low;//返回最小残量（当它为 0 时说明没有增广路了）
    }
    for(int i=head[pos];i!=0&&low!=0;i=edge[i].before)
    {
        int v=edge[i].to;
        if(edge[i].value!=0&&dep[v]==dep[pos]+1)//小优化：仅当 v 在当前位置的下一层
```

```
中才进行查找是否有增广路
            {
                long long int rlow=dfs(v,min(low,edge[i].value));
                low-=rlow;//该点使用的流量增加
                edge[i].value-=rlow;//过去的边是剩余可支配的量
                edge[i^1].value+=rlow;//反向边+流量
                out+=rlow;
            }
        }
        if(out==0) dep[pos]=0x3f3f3f3f;
        return out;//返回该点已使用流量
}
bool bfs()//给增广路上的点分层
{
    std::queue<int>q;
    memset(dep,0x3f3f3f3f,sizeof(dep));
    memset(inque,0,sizeof(inque));
    dep[s]=0;//源点深度当然为 0
    q.push(s);
    while(!q.empty())
    {
        int u=q.front();
        q.pop();
        inque[u]=0;//不在队伍中了
        for(int i=head[u];i!=0;i=edge[i].before)
        {
            int v=edge[i].to;//通向的点
            if(edge[i].value!=0&&dep[v]>dep[u]+1)//如果容量不为 0 且在 u 点之前还没有
被搜到
            {
                dep[v]=dep[u]+1;
                if(inque[v]==0)//如果点 v 不在当前队伍中
                {
                    q.push(v);
                    inque[v]=1;
                }
            }
        }
    }
    if (dep[t]!=0x3f3f3f3f)//只要汇点被搜到了，就还有增广路
        return 1;
    return 0;
}
int main()
```

```
{
    long long int maxflow=0;
    cin>>n>>m>>s>>t;
    int i,a,b,c;
    for(i=0;i<m;i++)
    {
        cin>>a>>b>>c;
        cnt++;edge[cnt].to=b;edge[cnt].value=c;edge[cnt].before=head[a];head[a]=cnt;//
正边为偶数，从 2 开始
        cnt++;edge[cnt].to=a;edge[cnt].value=0;edge[cnt].before=head[b];head[b]=cnt;//
反边为奇数，从 3 开始，初始流量为 0
    }
    while(bfs())//如果还有增广路就继续 dfs
        maxflow=maxflow+dfs(s, 1e9);
    cout<<maxflow<<endl;
    return 0;
}
```

LCA 最近公共祖先（链式向前星）

该题找到祖先后算出两节点之间最短距离

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
struct node
{
    int to;
    int before;
}edge[200000];
int lg[200000];
int head[200000],depth[200000],fa[200000][22];//fa[u][i]表示 u 的第 2 的 i 次方个祖先（fa[u][0]
就是 u 的父亲）
int n,cnt=0;
void dfs(int now,int father)
{
    int i;
    fa[now][0]=father;depth[now]=depth[father]+1;
    for(i=1;i<=lg[depth[now]];i++)
        fa[now][i]=fa[fa[now][i-1]][i-1]; //这个转移可以说是算法的核心之一
                                          //意思是 now 的 2^i 祖先等于 now 的 2^(i-1)祖先
```

的 2^(i-1)祖先

```
                                                    //2^i = 2^(i-1) + 2^(i-1)
    for(i=head[now];i;i=edge[i].before)
        if(edge[i].to!=father) dfs(edge[i].to,now);
}
int LCA(int x,int y)
{
    int k;
    if(depth[x]<depth[y])
    {
        int tmp=x;x=y;y=tmp;
    }
    while(depth[x]>depth[y])
        x=fa[x][lg[depth[x]-depth[y]]-1]; //先跳到同一深度
    if(x==y) //如果 x 是 y 的祖先，那他们的 LCA 肯定就是 x 了
    {
        return x;
    }
    for(k=lg[depth[x]]-1;k>=0;k--) //不断向上跳（lg 就是之前说的常数优化）
    {
        if(fa[x][k]!=fa[y][k])  //因为我们要跳到它们 LCA 的下面一层，所以它们肯定不相等，
如果不相等就跳过去。
        {
            x=fa[x][k];
            y=fa[y][k];
        }
    }
    return fa[x][0];  //返回父节点
}
int main()
{
    int fu,sum;
    int a,b,i,j,q;
    cin>>n;
    for(i=1;i<=n-1;i++)
    {
        cin>>a>>b;
        cnt++;edge[cnt].to=b;edge[cnt].before=head[a];head[a]=cnt;
        cnt++;edge[cnt].to=a;edge[cnt].before=head[b];head[b]=cnt;
    }
    for(i=1;i<=n;i++) lg[i]=lg[i-1]+(1<<lg[i-1]==i);
    dfs(1,0);
    cin>>q;
    for(i=0;i<q;i++)
```

```
    {
        cin>>a>>b;
        fu=LCA(a,b);
        sum=depth[a]+depth[b]-2*depth[fu];
        if(sum%2==0) cout<<"YE5"<<endl;
        else cout<<"N0"<<endl;
    }
    return 0;
}
```

迪杰斯特拉算法

```
#include<iostream>
#include<algorithm>
#include<queue>
using namespace std;
struct node
{
    int to,value,before;
}edge[300000];
int head[300000],cnt=0,n,m,vis[300000],dis[300000];
#define P pair<long long int,int>
priority_queue<P,vector<P>,greater<P> >q;//为格式模板
//把最小的元素放在队首的优先队列,这是一个写法，优先队列是以 pair 组中第一个元素（greater 与默认相反，是小顶堆）排序
void dijkstra(int);
void add(int u,int v,int w)
{
    cnt++;
    edge[cnt].to=v;
    edge[cnt].value=w;
    edge[cnt].before=head[u];
    head[u]=cnt;
}
int main()
{
    cin>>n>>m;
    int i,x,y,z,from,to;
    cin>>from>>to;
    for(i=1;i<=m;i++)
    {
        cin>>x>>y>>z;
```

```
            add(x,y,z);
            add(y,x,z);
        }
        dijkstra(from);
        cout<<dis[to]<<endl;
        return 0;
}
void dijkstra(int from)
{
        for(int i=1;i<=n;i++)
        {
            dis[i]=1e9;
        }
        dis[from]=0;
        q.push(make_pair(0,from));//pair 入队需要用 make_pair
        while(!q.empty())
        //堆为空即为所有点都更新
        {
            int x=q.top().second;
            q.pop();
            //记录堆顶并将其弹出
            if(!vis[x])
            //没有遍历过才需要遍历
            {
                vis[x]=1;
                for(int i=head[x];i;i=edge[i].before)
                //搜索堆顶所有连边
                {
                    int v=edge[i].to;
                    dis[v]=min(dis[v],dis[x]+edge[i].value);
                    //松弛操作
                    q.push(make_pair(dis[v],v));
                }
            }
        }
}
```

两个点是否连通
现在有两种操作:
第一种: 在两个城市间新建一条路。(路双向互通, 不保证两个城市间只会有一条直接的路)

第二种：询问两个城市间是否存在互通路径。

```
#include<iostream>
#include<algorithm>
#include<queue>
using namespace std;
int father[500000];
int tmp[500000];
int find_father(int x)//找父节点
{
	if(x!=father[x])
	{
		x=father[x];
		x=find_father(x);
	}
	return x;
}
int main()
{
	int i,j,x,y,z,cnt=0,n,m;
	cin>>n>>m;
	for(i=0;i<=n;i++) father[i]=i;
	for(i=0;i<m;i++)
	{
		int a,b;
		cin>>z>>x>>y;
		if(z==1)
		{
			a=find_father(x);
			b=find_father(y);
			father[x]=a;
			father[y]=b;
			if(a!=b)
			{
				father[b]=a;
			}
		}
		else if(z==2)
		{
			a=find_father(x);
			b=find_father(y);
			if(a!=b) printf("NO\n");
			else printf("YES\n");
		}
	}
```

```
    return 0;
}
```

二分图，间谍躲安全屋

```
//经典二分图算法。题意：n 个间谍 n 个安全屋，都有一个坐标，间谍最多可以移动距离 d
//问最少有多少人躲不进安全屋
#include<iostream>
#include<cmath>
using namespace std;
int map[300][300],visit[300];
int n,d;
int man[300][2];//存间谍坐标
int house[300];//记录这个房子有没有人住进去了
bool dfs(int x)
{
    int i;
    for(i=1;i<=n;i++)
    {
        if(map[x][i]==1&&visit[i]==0)//有线且本次没被找过
        {
            visit[i]=1;
            if(house[i]==0||dfs(house[i]))//没人住进去或者可以换一个人连线
            {
                house[i]=x;
                return true;
            }
        }
    }
    return false;
}
int main()
{
    int i,j;
    cin>>n>>d;//n 指安全屋与间谍数量
    for(i=1;i<=n;i++) cin>>man[i][0]>>man[i][1];
    for(i=1;i<=n;i++)
    {
        int x,y;
        cin>>x>>y;
```

```
            for(j=1;j<=n;j++)
            {
                if(d*d>=(man[j][0]-x)*(man[j][0]-x)+(man[j][1]-y)*(man[j][1]-y))    map[i][j]=1;//
能不能进入
            }
        }
        int num=0;
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++) visit[j]=0;
            if(dfs(i)) num++;
        }
        cout<<num<<endl;
        return 0;
}
```