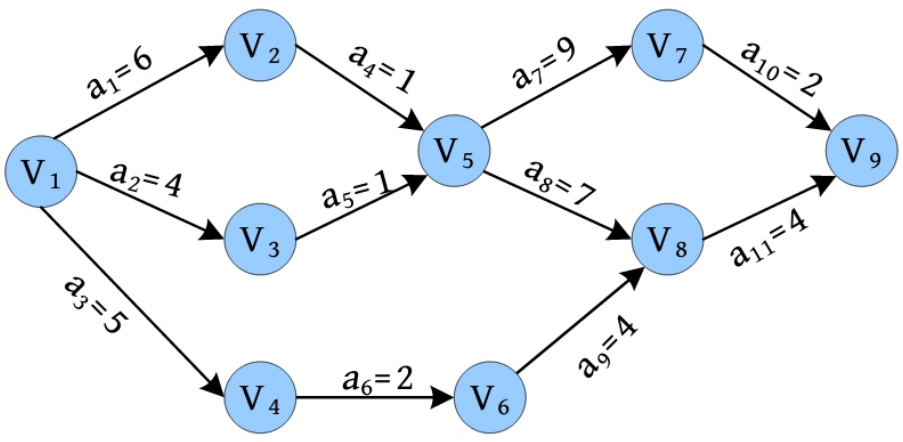
SDUTOJ2498

题目描述（SDUTOJ2498）：一个无环的有向图被称为有向无环图（Directed Acyclic Graph，之后简称 DAG）。AOE（Activity On Edge）网是指以边表示活动的网，如下图所示。



在上图中共有 11 个活动、9 个事件。整个工程只有一个开始点和一个完成点，即只有一个入度为零的点（源点）和一个出度为零的点（汇点）。关键路径指从开始点到完成点的最长路径。路径的长度是边上活动耗费的时间。如上图所示，1-2-5-7-9 是关键路径（关键路径不止一条，输出字典序最小的），权值之和为 18。

输入：输入包含多组数据，不超过 10 组。第 1 行包含节点数 n（2≤n≤10000）和边数 m（1≤m≤50000），接下来的 m 行，包含每条边的起点 s 和终点 e，权值 w（1≤s,e≤n，s!=e，1≤w≤20）。数据保证图连通，且只有一个源点和汇点。

输出：单行输出关键路径的权值和，并且从源点输出关键路径上的路径（如果有多条，则输出字典序最小的）。

输入样例	输出样例
9 11 1 2 6 1 3 4 1 4 5 2 5 1 3 5 1 4 6 2 5 7 9 5 8 7 6 8 4 8 9 4 7 9 2	18 1 2 2 5 5 7 7 9

题解：本题求解关键路径**实际上就是求解最长路径。求解最长路径时可以将权值加负号求解最短路径，也可以改变松弛条件，若距离较大则更新。**

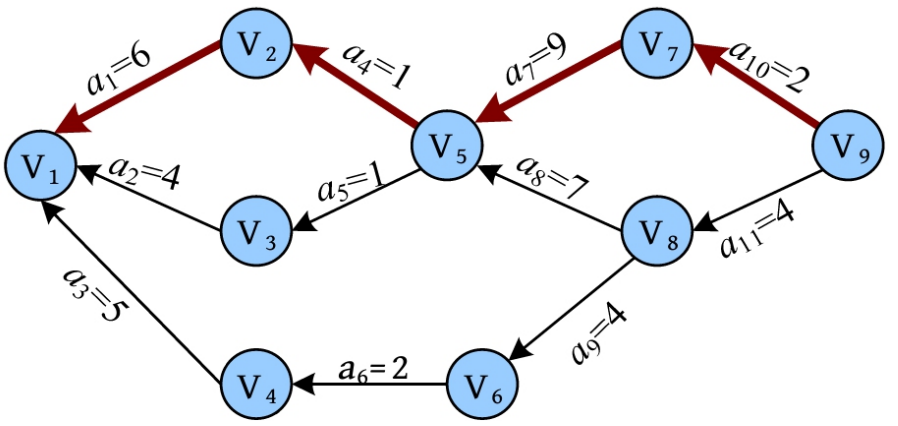
- 对有向无环图，可以按拓扑序列松弛求解最长路径，也可以用 Bellman 或 SPFA 算法权值加负号求解最短路径，或者改变松弛条件求解最长路径。
- 对有向有环图，可以用 Bellman 或 SPFA 算法判断环，若有正环，则不存在最长路径。

需要注意的是，Dijkstra 算法不可以用于处理负权边，也无法通过改变松弛条件得到最长路径。Bellman 算法的时间复杂度为 O(n×m)，可能会超时，所以可以采用 SPFA 算法，该算法的时间复杂度为 O(k×m)，k 是一个较小的常数，最多为 O(n×m)。其次，该题需要输出路径，而且该路径需要按字典序选取，所以**反向建图会更便于记录路径。**

路径的字典序最小就是走到一个点，继续向下一步走时，选择编号最小的，这就是字典序，但是在最短路径的更新过程中，如果 dis[y]==dis[x]+w&&x<pre[y]，路径长度相等但是 x 比 y 的前驱节点编号更小，则更新 y 的前驱节点为 x，即 pre[y]=x。

在本题的 AOE 网中，V_5-V_7-V_9和 V_5-V_8-V_9的路径长度是一样的，按字典序应该走前者。如果逆向走，从 V_9到 V_7，则 dis[7]=2；从 V_9到 V_8，则 dis[8]=4；从 V_8到 V_5，则 dis[5]=11，pre[5]=8；从 V_7到 V_5，则 dis[7]+9=11=dis[5]，但是 7 比 8 的字典序小，更新 5 的前驱为 7，pre[5]=7。

在原图的逆向图上，从后向前走一条最长路径，然后根据前驱数组，1 的前驱为 2，输出 1 2；2 的前驱为 5，输出 2 5；5 的前驱为 7，输出 5 7；7 的前驱为 9，输出 7 9。



1. 算法设计

（1）建立原图的逆向图。检查入度为 0 的节点 s 和出度为 0 的节点 t。

（2）使用 SPFA 算法求最长路径。如果 dis[y]<dis[x]+e[i].w||(dis[y]==dis[x]+e[i].w&&x<pre[y]))，则更新 dis[y]=dis[x]+e[i].w; pre[y]=x。

2. 算法实现

```
void spfa(int u){
    queue<int>q;
    q.push(u);
    inq[u]=1;
    while(!q.empty()){
        int x=q.front();
        q.pop();
        inq[x]=0;
        for(int i=head[x];i;i=e[i].next){
            int y=e[i].to;
            if(dis[y]<dis[x]+e[i].w||(dis[y]==dis[x]+e[i].w&&x<pre[y])){
                dis[y]=dis[x]+e[i].w;
                pre[y]=x;
                if(!inq[y]){
                    q.push(y);
                    inq[y]=1;
                }
            }
        }
    }
}
```

HDU4109

题目描述（HDU4109）：阿里本学期开设了计算机组成原理课程。他了解到指令之间可能存在依赖关系，例如 WAR（写入后读取）、WAW、RAW。

如果两个指令之间的距离小于安全距离，则会导致危险，这可能导致错误的结果。所以需要设计特殊的电路以消除危险。然而，解决此问题的最简单方法是添加气泡（无用操作），这意味着浪费时间以确保两条指令之间的距离不小于安全距离。对两条指令之间距离的定义是它们的开始时间之间的差。

现在有很多指令，已知指令之间的依赖关系和安全距离，可以根据需要同时运行多个指令，并且 CPU 速度非常快，只需花费 1ns 即可完成任何指令。你的工作是重新排列指令，以便 CPU 用最短的时间完成所有指令。

输入：输入包含几个测试用例。每个测试用例的第 1 行都包含两个整数 N 和 M（N≤1000,M ≤10000），表示 N 个指令和 M 个依赖关系。以下 M 行，每行都包含 3 个整数 X、Y、Z，表示 X 和 Y 之间的安全距离为 Z，Y 在 X 之后运行。指令编号为 0～N-1。

输出：单行输出一个整数，即 CPU 运行所需的最短时间。

输入样例	输出样例
5 2 1 2 1 3 4 1	2

题解：根据测试用例，构建的图形结构如下图所示。在第 1ns 中，执行指令 0、1 和 3；在第 2ns 中，执行指令 2 和 4。答案是 2。

按照拓扑排序求每个节点的最长距离，然后求各个节点最长距离的最大值。

算法代码：

```
void TopoSort(){//按拓扑排序求最长距离
   int cnt=0;
   for(int i=0;i<n;i++)
      if(in[i]==0){
         s.push(i);
         d[i]=1;
      }
   while(!s.empty()){
      int u=s.top();
      s.pop();
      topo[cnt++]=u;
      for(int v=0;v<n;v++){
          if(map[u][v]){
             d[v]=max(d[v],d[u]+map[u][v]);
             if(--in[v]==0)
                s.push(v);
          }
      }
   }
}

int main(){
   int u,v,w;
   while(cin>>n>>m){
      memset(map,0,sizeof(map));
      memset(in,0,sizeof(in));
      memset(d,0,sizeof(d));
      for(int i=1;i<=m;i++){
          cin>>u>>v>>w;
          map[u][v]=w;
          in[v]++;
      }
      TopoSort();
      int ans=0;
      for(int i=0;i<n;i++)
         ans=max(ans,d[i]);
      cout<<ans<<endl;
   }
   return 0;
}
```

POJ1949

题目描述（POJ1949）：约翰有一份必须完成的 N（3≤N≤10 000）个家务的清单。每个家务都需要一个整数时间 T（1≤T≤100）才能完成，并且可能还有其他家务必须在这个家务开始之前完成。至少有一个家务没有先决条件：第 1 号。家务 K（K>1）只能以家务 1～K-1 作为先决条件。计算完成所有 N 个家务所需的最少时间。当然，可以同时进行彼此不依赖的家务。

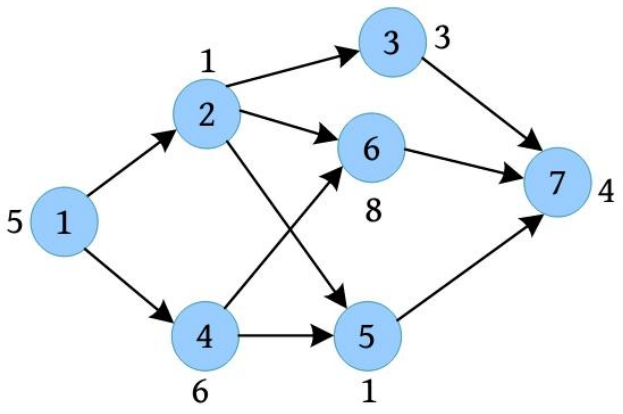
输入：第 1 行包含一个整数 N。第 2～N+1 行描述每个家务，第 2 行包含家务 1；第 3 行包含家务 2，以此类推。每行都包含完成家务的时间、先决条件的数量 P_i（0≤P_i≤100）和 P_i 个先决条件。

输出：单行输出完成所有家务所需的最少时间。

输入样例	输出样例
7 5 0 1 1 1 3 1 2 6 1 1 1 2 2 4 8 2 2 4 4 3 3 5 6	23

题解：根据输入样例 1，构建的图形结构如下图所示。

```
7    //家务数
5 0  //第 1 个家务，时间是 5，没有先决条件
1 1 1 //第 2 个家务，时间是 1，有 1 个先决条件为 1
3 1 2
6 1 1
1 2 2 4 //第 5 个家务，时间是 1，有两个先决条件为 2、4
8 2 2 4
4 3 3 5 6
```



分析：

- 家务 1 在时间 0 开始，在时间 5 结束；
- 家务 2 在时间 5 开始，在时间 6 结束；
- 家务 3 在时间 6 开始，在时间 9 结束；
- 家务 4 在时间 5 开始，在时间 11 结束；
- 家务 5 在时间 11 开始，在时间 12 结束；
- 家务 6 在时间 11 开始，在时间 19 结束；
- 家务 7 在时间 19 开始，在时间 23 结束。

本题的关键在于，**家务 K（K>1）只能以家务 1～K-1 作为先决条件**。也就是说，输入第 K 个家务时，它的先决条件均已确定什么时间结束。因此在输入过程中直接求最长距离即可。如果没有先决条件限制，则不可以这样计算。

算法代码：

```
int main(){
    int ans=0,w,num,y;
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%d%d",&w,&num);//本题数据量大，使用 cin 易超时
        d[i]=w;
        for(int j=1;j<=num;j++){
            scanf("%d",&y);
            d[i]=max(d[i],d[y]+w);
        }
        ans=max(ans,d[i]);
    }
    printf("%d\n",ans);
    return 0;
}
```

HDU1224

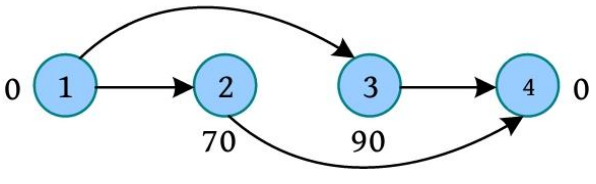
题目描述（HDU1224）：旅游公司展示了一种新型 DIY 线路。各线路都包含一些可由游客自己选择的城市。根据该公司的统计数据，每个城市都有自己的评分，评分越高越有趣。例如，巴黎的评分是 90，纽约的评分是 70，等等。世界上不是任何两个城市之间都可以直飞的，因此旅游公司提供了一张地图，告诉游客是否可以在地图上任意两个城市之间直飞。在地图上用一个数字标记每个城市，一个数字较大的城市不能直接飞往数字较小的城市。薇薇从杭州出发（杭州是第 1 个城市，也是最后 1 个城市，所以杭州被标记为 1 和 N+1），它的评分为 0。薇薇希望尽可能地让游览变得有趣。

输入：第 1 行是整数 T，表示测试用例数。每个测试用例的第 1 行都是一个整数 N（2≤N≤100），表示城市数。然后是 N 个整数，表示城市的评分。接着是整数 M，后跟 M 对整数 A_i、B_i（1≤i≤M），表示从城市 A_i 可以直飞到城市 B_i。

输出：对于每个测试用例，都单行输出评分之和的最大值和最佳 DIY 线路。在测试用例之间都输出一个空行。

输入样例	输出样例
2 3 0 70 90 4 1 2 1 3 2 4 3 4 3 0 90 70 4 1 2 1 3 2 4 3 4	CASE 1# points : 90 circuit : 1->3->1 CASE 2# points : 90 circuit : 1->2->1

题解：本题其实是**求解 1～N+1 的最长路径**。根据输入样例 1，构建的图如下图所示。



起点和终点的评分为 0，终点 4 实际上也是起点 1，因为起点编号为 1 和 N+1。1→3→1 这条路径的评分之和最大，因此答案为 90。

1. 算法设计

可以使用邻接矩阵存储，使用两个 for 语句更新。也可以使用 SPFA 算法求最长路径。

（1）读入每个节点的评分，将第 N+1 个节点的评分设置为 0。

（2）读入可以直飞的城市编号，采用邻接矩阵存储。

（3）枚举 j=1...n+1，i=1...j-1，如果 map[i][j]&&dis[j]<dis[i]+qd[j]，则

```
dis[j]=dis[i]+qd[j]; pre[j]=i
```

（4）递归输出最长的回路。

2. 算法实现

```
void printpath(int i){//输出最长的回路
    if(i==-1)
        return;
    printpath(pre[i]);
    cout<<i<<"->";
}
```

```
int main(){
    int T,u,v,cas=0;
    cin>>T;
    while(T--){
        memset(pre,-1,sizeof(pre));
        memset(dis,0,sizeof(dis));
        memset(map,0,sizeof(map));
        cin>>n;
        for(int i=1;i<=n;i++)
            cin>>qd[i];
        qd[n+1]=0;
        cin>>m;
        for(int i=1;i<=m;i++){
            cin>>u>>v;
            map[u][v]=1;
        }
        for(int j=1;j<=n+1;j++)
            for(int i=1;i<j;i++)
                if(map[i][j]&&dis[j]<dis[i]+qd[j]){
                    dis[j]=dis[i]+qd[j];
                    pre[j]=i;
                }
        if(cas)
            cout<<endl;
        cout<<"CASE "<<++cas<<"#"<<endl;
        cout<<"points : "<<dis[n+1]<<endl;
        cout<<"circuit : ";
        printpath(pre[n+1]);//最后一个节点，手工输出1，所以从pre[n+1]开始
        cout<<"1"<<endl;
    }
    return 0;
}
```

HDU1317

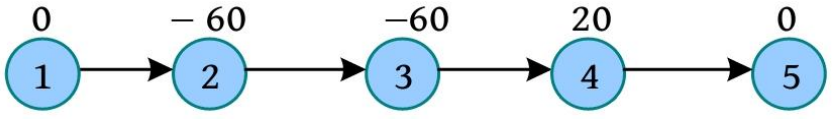
题目描述（HDU1317）：有 n（n≤100）个房间，每个房间都有一个能量值（范围是-100～+100）。以单向门连接两个房间，可以通过任何连接所在房间的门到达另一个房间，从而进入另一个房间，到达该房间时会自动获得该房间的能量。可以多次进入同一个房间，每次都能获得能量。初始能量值为 100，初始位置是 1 号房间，要走到 n 号房间。1 号房间和 n 号房间的能量值均为 0。到达 n 号房间可获胜，如果中途能量值小于或等于 0，则会因能量耗尽而死亡。

输入：输入包含几个测试用例。每个测试用例的第 1 行都为 n，表示房间数。接下来是 n 个房间的信息。每个房间的信息都包括：房间 i 的能量值、离开房间 i 的门数量、房间 i 可以通过门到达的房间列表。在最后一个测试用例之后是包含-1 的行。

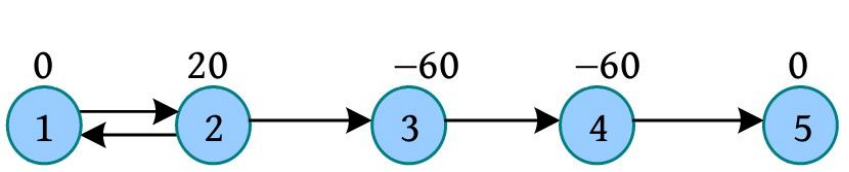
输出：如果玩家有可能获胜，则输出 winnable，否则输出 hopeless。

输入样例	输出样例
5 0 1 2 -60 1 3 -60 1 4 20 1 5 0 0 5 0 1 2 20 1 3 -60 1 4 -60 1 5 0 0 5 0 1 2 21 1 3 -60 1 4 -60 1 5 0 0 5 0 1 2 20 2 1 3 -60 1 4 -60 1 5 0 0 -1	hopeless hopeless winnable winnable

题解：根据输入样例 1，构建的图如下图所示，到不了 5 号房间能量就耗尽了，输出“hopeless”。



根据输入样例 4，构建的图如下图所示，有正环且可以到达终点，输出“winnable”。



如果从 1 号房间到 n 号房间不连通，则必然不能获胜。如果有正环，则环上的点到 n 号房间连通即可获胜。如果没有环，则到达终点的最长路径的能量值大于 0 即可获胜。

1. 算法设计

（1）用 Floyd 算法判断连通性，判断能否从 1 号房间走到 n 号房间，如果不连通则结束。

（2）用 SPFA 算法判断有没有正环，在 cnt[v]≥n 时有正环，判断环上一点到终点是否连通。如果没有正环，则判断到达终点的最长路径的能量值是否大于 0 即可。

注意：由于该题给出的数据是每个节点的能量值，而不是边的能量值，需要用 Floyd 算法判断连通性，因此用邻接矩阵来存储图。

2. 算法实现

```
void floyd(){//判断连通性
   for(int k=1;k<=n;k++)
      for(int i=1; i<=n;i++)
         for(int j=1;j<=n;j++)
            if(reach[i][j]||(reach[i][k]&&reach[k][j]))
               reach[i][j]=true;
}
bool spfa(int s){//判断有没有正环，并求最大能量值
   queue<int> q;
   memset(power,0,sizeof(power));
   memset(vis,0,sizeof(vis));
   memset(cnt,0,sizeof(cnt));
   power[s]=100;
   q.push(s);
   vis[s]=1;
   cnt[s]++;
   while(!q.empty()){
      int v=q.front();
      q.pop();
      vis[v]=0;
      for(int i=1;i<=n;i++){
         if(g[v][i]&&power[i]<power[v]+energy[i]&&power[v]+energy[i]>0){
            power[i]=power[v]+energy[i];
            if(!vis[i]){
               if(++cnt[i]>=n) //有正环
                  return reach[v][n]; //返回 v 到 n 是否连通
               vis[i]=1;
               q.push(i);
            }
         }
      }
   }
   return power[n]>0;
}
void solve(){
   int k,door;
   while(cin>>n&&n!=-1){
      memset(g,false,sizeof(g));
      memset(reach,false,sizeof(reach));
      for(int i=1;i<=n;i++){
         cin>>energy[i]>>k;
         for(int j=1;j<=k;j++){
            cin>>door;
            g[i][door]=true;
            reach[i][door]=true;
         }
      }
      floyd();
      if(!reach[1][n]){
         cout<<"hopeless"<<endl;
         continue;
      }
      if(spfa(1))
         cout<<"winnable"<<endl;
      else
         cout<<"hopeless"<<endl;
   }
}
```