0.1 二分图匹配

0.1.1 概念

点覆盖、最小点覆盖
点覆盖集即一个点集，使得所有边至少有一个端点在集合里。
极小点覆盖：本身为点覆盖，其真子集都不是。
最小点覆盖：假如选了一个点就相当于覆盖以它为端点的所有边，选择最少的点来覆盖所有的边。
点覆盖数：最小点覆盖的点数。

边覆盖、极小边覆盖
边覆盖集即一个边集，使得所有点都与集合里的边邻接。
极小边覆盖：本身是边覆盖，其真子集都不是。
最小边覆盖：边最少的边覆盖。
边覆盖数：最小边覆盖的边数。

独立集、极大独立集
独立集即一个点集，集合中任两个结点不相邻。
极大独立集：本身为独立集，再加入任何点都不是。
最大独立集：点最多的独立集。
独立数：最大独立集的点。

团
团即一个点集，集合中任两个结点相邻。
极大团：本身为团，再加入任何点都不是。
最大团：点最多的团。
团数：最大团的点数。

边独立集、极大边独立集
边独立集即一个边集，满足边集中的任两边不邻接。（一个顶点最多只有一条边经过）
极大边独立集：本身为边独立集，再加入任何边都不是。
最大边独立集：边最多的边独立集。
边独立数：最大边独立集的边数。
边独立集又称匹配，相应的有极大匹配，最大匹配，匹配数)。

支配集、极小支配集
支配集即一个点集，使得所有其他点至少有一个相邻点在集合里。
极小支配集：本身为支配集，其真子集都不是。
最小支配集：点最少的支配集。
支配数：最小支配集的点数。

边支配集、极小边支配集
边支配集即一个边集，使得所有边至少有一条邻接边在集合里。
极小边支配集：本身是边支配集，其真子集都不是。
最小边支配集：边最少的边支配集。
边支配数：最小边支配集的边数。

最小路径覆盖
最小路径覆盖：是“路径”覆盖“点”，即用尽量少的不相交简单路径覆盖有向无环图 G 的所有顶点，即每个顶点严格属于一条路径。路径的长度可能为 0(单个点)。
最小路径覆盖数＝ G 的点数－最小路径覆盖中的边数。应该使得最小路径覆盖中的边数尽量多，但是又不能让两条边在同一个顶点相交。

拆点
将每一个顶点 i 拆成两个顶点 Xi 和 Yi。然后根据原图中边的信息，从 X 部往 Y 部引边。所有边的方向都是由 X 部到 Y 部。因此，所转化出的二分图的最大匹配数则是原图 G 中最小路径覆盖上的边数。因此由最小路径覆盖数＝原图 G 的顶点数－二分图的最大匹配数。
如果原图允许路径交叉，即同一顶点多条路径经过【例如 POJ 2594】，那么可以用 Floyd 算法将所有间接连通的点对变为直接连通，这样一来如果原图种的最小路径覆盖在一个顶点有多条路径，新图可以使得一些路径直接跳过这个顶点而直接到达终点，这样就可以使用匈牙利算法求出最大匹配数。

匹配
匹配是一个边集，满足边集中的边两两不邻接。匹配又称边独立集。
在匹配中的点称为匹配点或饱和点；反之，称为未匹配点或未饱和点。
交错轨是图的一条简单路径，满足任意相邻的两条边，一条在匹配内，一条不在匹配内。
增广轨：是一个始点与终点都为未匹配点的交错轨。
最大匹配是具有最多边的匹配。
匹配数是最大匹配的大小。
完美匹配是匹配了所有点的匹配。(判断完美匹配可以求出最大匹配数然后和顶点数比较）
完备匹配是匹配了二分图较小集合（二分图 X，Y 中小的那个）的所有点的匹配。

增广轨定理：一个匹配是最大匹配当且仅当没有增广轨。
所有匹配算法都是基于增广轨定理：一个匹配是最大匹配当且仅当没有增广轨。这个定理适用于任意图。

0.1.2 二分图的性质

二分图中，点覆盖数是匹配数。

1. 二分图的最小点覆盖数等于最大匹配数：最少的点使得每条边都至少和其中的一个点相关联。并且最小点覆盖的顶点一定在最大匹配的边的端点中产生. 原因如下. 只需要考虑边不是最大匹配边能否被最大匹配边的端点覆盖. 如果这条边的两个端点都不是最大匹配边端点集中的点, 那么这条边就是增广路, 这和最大匹配性质相违背 (求完最大匹配后不应该还有增广路), 所以这条边一定有一个端点是最大匹配边端点集中的点, 那么只需要选择这个点就能覆盖这条边了.

2. 二分图的独立数（最大独立集的点数）等于顶点数减去最大匹配数。把最大匹配两端的点都从顶点集中去掉这个时候剩余的点是独立集，这是 $|V|-2*|M|$，同时必然可以从每条匹配边的两端取一个点加入独立集并且保持其独立集性质，所以一共是 $|V|-|M|$。

3. 最小边覆盖 = 图中点的个数 -最大匹配数 = 最大独立集

0.1.3 二分图的判定

二分图：有两顶点集且图中每条边的的两个顶点分别位于两个顶点集中，每个顶点集中没有边直接相连接！
无向图 G 为二分图的充分必要条件是，G 至少有两个顶点, 且其所有回路的长度均为偶数。

判断二分图的常见方法是染色法：
开始对任意一未染色的顶点染色，之后判断其相邻的顶点中，若未染色则将其染上和相邻顶点不同的颜色，若已经染色且颜色和相邻顶点的颜色相同则说明不是二分图，若颜色不同则继续判断，bfs 和 dfs 可以搞定！
易知：任何无回路的的图均是二分图
用 BFS+ 链式前向星判断二分图的代码：

```
const int MAX_N=210;
const int MAX_M=40010;

int n,m,total;
int head[MAX_N],color[MAX_N];

struct Edge{
    int to,next;

    Edge() {}
    Edge(int to,int next) : to(to),next(next) {
    }
}edge[MAX_M];

inline void AddEdge(int from,int to)
{
    edge[total].to=to;
    edge[total].next=head[from];
    head[from]=total++;
}
```

```
21
22 inline bool IsBipartiteGraph()
23 {
24     int st=1;
25     memset(color,-1,sizeof(color));
26     while(1){
27         int find=0;
28         for(int i=st;i<=n;i++){
29             if(color[i]==-1){
30                 st=i;
31                 find=1;
32                 break;
33             }
34         }
35         if(find==0) break;
36         queue<int> que;
37         color[st]=1;
38         // color[i]=0 和 color[j]=0 是不同的两组
39         que.push(st);
40         st++;
41         while(!que.empty()){
42             int cur=que.front();
43             que.pop();
44             for(int i=head[cur];i!=-1;i=edge[i].next){
45                 int v=edge[i].to;
46                 if(color[v]==-1){
47                     color[v]=1-color[cur];
48                     que.push(v);
49                 }else if(color[v]==color[cur]){
50                     return false;
51                 }
52             }
53         }
54     }
55     return true;
56 }
57
58 int main()
59 {
60     while(~scanf("%d%d",&n,&m)){
61         total=0;
62         memset(head,-1,sizeof(head));
63         for(int i=0;i<m;i++){
64             int from,to;
65             scanf("%d%d",&from,&to);
66             AddEdge(from,to);
67         }
68         if(IsBipartiteGraph()==false){
69             printf("No\n");
70         }else {
71            printf("Yes\n");
72        }
73     }
74     return 0;
75 }
```

0.1.4 匈牙利算法 (Hungary Algorithm)

根据一个匹配是最大匹配当且仅当没有增广路, 求最大匹配就是找增广轨，直到找不到增广轨，就找到了最大匹配。遍历每个点，查找增广路，若找到增广路，则修改匹配集和匹配数，否则，终止算法，返回最大匹配数。

时间复杂度是：$O(n*m)$

```
const int MAX_N=510;

int T,n,m,total;
int vis[MAX_N],match[MAX_N],head[MAX_N];

struct Edge{
        int to,next;

        Edge () {}
        Edge(int to,int next) : to(to),next(next) {
        }
}edge[MAX_N*MAX_N];

inline void AddEdge(int from,int to)
{
        edge[total].to=to;
        edge[total].next=head[from];
        head[from]=total++;
}

inline bool dfs(int u)
{
        for(int i=head[u];i!=-1;i=edge[i].next){
                int v=edge[i].to;
                if(!vis[v]){
                        vis[v]=1;
                        if(match[v]==-1 || dfs(match[v])){
                                match[v]=u;
                                return true;
                        }
                }
        }
        return false;
}

inline void solve()
{
        memset(match,-1,sizeof(match));
        int res=0;
        for(int i=1;i<=n;i++){
                memset(vis,0,sizeof(vis));
                if(dfs(i)){
                        res++;
                }
        }
        printf("%d\n",res); // res 即是最大匹配数
}

int main()
{
        scanf("%d",&T);
        while(T--){
                total=0;
                memset(head,-1,sizeof(head));
                scanf("%d%d",&n,&m);
                for(int i=1;i<=m;i++){
                        int from,to;
                        scanf("%d%d",&from,&to);
                        AddEdge(from,to);
                }
                solve();
        }
        return 0;
}
```

0.1.5 输出最小点覆盖的点

选择最少的点, 使得每条边至少有一个端点被选中. 最小覆盖数等于最大匹配数。
先求出最大匹配数, 并将每个 x 和 y 对应的匹配记录下来. 然后从 X 中的所有未匹配点出发扩展匈牙利树, 标记树中的所有点, 则 X 中的未标记点和 Y 中的已标记点组成了所求的最小覆盖.

在一个 $R*C(R,C\leq 1000)$ 的网格上放了 $n\leq 10^5$ 目标. 可以在网格外发射子弹, 子弹会沿着垂直或者水平方向飞行, 并打掉飞行路径上的所有目标. 计算最少需要多少子弹, 各从哪些位置发射, 才能把所有目标全部打掉.

建图: 将每一行看作一个 X 结点, 每一列看作一个 Y 结点, 每个目标对应一条边. 这样, 子弹打掉所有目标意味着每条边至少有一个结点被选中. 需要特别注意的是本题: 各行从上到下编号为 $1\sim R$, 各列从左到右编号为 $1\sim C$!

```
const int MAX_N = 1010;

int n, m, k, total;
int head[MAX_N], visx[MAX_N], visy[MAX_N];
int  matchx[MAX_N], matchy[MAX_N];

struct Edge{
    int to, next;
}edge[MAX_N*MAX_N];


inline void AddEdge(int from, int to)
{
    edge[total].to = to;
    edge[total].next = head[from];
    head[from] = total++;
}

inline bool dfs(int u)
{
    visx[u] = 1;
    for(int i = head[u]; i != -1; i = edge[i].next){
        int v = edge[i].to;
        if(visy[v]) continue;
        visy[v] = 1;
        if(matchy[v] == -1 || dfs(matchy[v])){
            matchx[u] = v;
            matchy[v] = u;
            return true;
        }
    }
    return false;
}

inline int Hungary()
{// 匈牙利算法求最大匹配
    int res = 0;
    memset(matchy, -1, sizeof(matchy));
    memset(matchx, -1, sizeof(matchx));
    for(int i = 1; i <= n; i++){
        memset(visx, 0, sizeof(visx));
        memset(visy, 0, sizeof(visy));
        if(dfs(i)) res++;
    }
    return res;
}

int main()
{
    while(cin >> n >> m >> k && (n || m || k)){
        total = 0;
```

```
        memset(head, -1, sizeof(head));
        for(int i = 0; i < k; i++){
            int tmpx, tmpy;
            cin >> tmpx >> tmpy;
            AddEdge(tmpx, tmpy);
        }
        int ans = Hungary();
        // 将所有的 X 顶点和 Y 顶点标记状态清0
        memset(visx, 0, sizeof(visx));
        memset(visy, 0, sizeof(visy));
        for(int i = 1; i <= n; i++){
            if(matchx[i] == -1){
            //对所有不在最大匹配中的 X 顶点扩展匈牙利树标记树中顶点,
                dfs(i);
            }
        }
        cout << ans ;
        for(int i = 1; i <= n; i++){ // X 顶点中所有没被标记的顶点
            if(visx[i] == 0) cout << " r" << i ;
        }
        for(int i = 1; i <= m; i++){ // Y 顶点中所有被标记的顶点
            if(visy[i] == 1) cout << " c" << i ;
        }
        cout << endl;
    }
    return 0;
}
```

0.1.6 霍普克洛夫特 -卡普算法

Hopcroft-Carp 算法先使用 BFS 查找多条增广路，然后使用 DFS 遍历增广路（累加匹配数，修改匹配点集），循环执行，直到没有增广路为止。
BFS 遍历只对点进行分层（不标记是匹配点和未匹配点），然后用 DFS 遍历看上面的层次哪些是增广路径（最后一个点是未匹配的）。
BFS 过程可以看做是图像树结构一样逐层向下遍历，还要防止出现相交的增广路径。
每次使用调用 BFS 查找到多条增广路的路径长度都是相等的，而且都以第一次得到的 dis 为该次查找增广路径的最大长度。
时间复杂度:$O(n*\sqrt{(}m))$

```
int matchx[MAX_N], matchy[MAX_N], head[MAX_N];
// matchx[] 记录 x 点集的匹配
int disx[MAX_N],disy[MAX_N], vis[MAX_N];
// disx[] 记录 x 点集的点的层次

struct Edge{
    int to, next;

    Edge () {}
    Edge(int _to, int _next) : to(_to), next(_next) {
    }
}edge[MAX_N*MAX_N];

inline void init()
{
    total=0;
    memset(matchx,-1,sizeof(matchx));
    memset(matchy,-1,sizeof(matchy));
    memset(head,-1,sizeof(head));
}

inline void AddEdge(int from, int to)
{
    edge[total].to = to;
    edge[total].next = head[from];
```

```
26      head[from] = total++;
27  }
28
29  inline bool bfs()
30  {
31      dis = INT_MAX;
32      memset(disx, -1, sizeof(disx));
33      memset(disy, -1, sizeof(disy));
34      queue<int> que;
35      //从 x 中找到所有未匹配点，组成第 0 层
36      for(int i = 0; i < n; i++){
37          if(matchx[i] == -1){
38              que.push(i);
39              disx[i] = 0;
40          }
41      }
42      while(!que.empty()){
43          int u = que.front();
44          que.pop();
45          if( disx[u] > dis) break;
46          //新的 x 点集中的点增广路径长度大于 dis
47          for(int i = head[u]; i != -1; i = edge[i].next){
48              int v = edge[i].to;
49              if( disy[v] == -1){  // v 是未匹配点
50                  disy[v] = disx[u] + 1;
51
52                  if(matchy[v] == -1) {//得到本次 BFS 遍历的最大层
53                      dis = disy[v];
54                  } else {
55                      disx[matchy[v]] = disy[v] + 1;
56                      // v 是匹配点，继续延伸
57                      que.push(matchy[v]);
58                  }
59              }
60          }
61      }
62      return dis != INT_MAX;
63  }
64
65  inline bool dfs(int u)
66  {
67      for(int i = head[u]; i != -1; i = edge[i].next){
68          int v = edge[i].to;
69          if(!vis[v]){
70              vis[v] = 1;
71              if(matchy[v] != -1 && disy[v] == dis) continue;
72              if(matchy[v] == -1 || dfs(matchy[v]) ){
73                  matchy[v] = u;
74                  matchx[u] = v;
75                  return true;
76              }
77          }
78      }
79      return false;
80  }
81
82  //返回最大匹配数
83  inline int Hopcroft_Carp()
84  {
85      int ans = 0;
86      while( bfs() ){
87          memset(vis, 0, sizeof(vis));
88          for(int i = 0; i < n; i++){
89              if(matchx[i] == -1 && dfs(i)) ans++;
90          }
```

```
91      }
92      return ans;
93  }
```

匈牙利算法和 Hopcroft-Carp 算法细节的对比:
匈牙利算法每次都以一个点查找增广路径，Hopcroft-Carp 算法是每次都查找多条增广路径；
匈牙利算法每次查找的增广路径的长度是随机的，Hopcroft-Carp 算法每趟查找的增广路径的长度只会在原来查找到增广路径的长度增加偶数倍（除了第一趟，第一趟得到增广路径长度都是 1）。

0.1.7 二分图带权匹配

Kuhn-Munkers 算法用来解决最大权匹配问题：在一个二分图内，左顶点为 X，右顶点为 Y，现对于每组左右连接 $X[i],Y[j]$ 有权 $w[i][j]$，求一种匹配使得所有 $w[i][j]$ 的和最大。
最大权匹配一定是完备匹配。如果两边的点数相等则是完美匹配。
如果点数不相等，其实可以虚拟一些点，使得点数相等，也成为了完美匹配。

算法描述：
Kuhn-Munkers 算法是通过给每个顶点一个标号（叫做顶标）来把求最大权匹配的问题转化为求完备匹配的问题的。设顶点 X_i 的顶标为 $A[i]$，顶点 Yi 的顶标为 $B[i]$，顶点 Xi 与 Yj 之间的边权为 $w[i,j]$。在算法执行过程中的任一时刻，对于任一条边 (i,j)，$A[i]+B[j]>=w[i,j]$ 始终成立，初始 $A[i]$ 为与 xi 相连的边的最大边权，$B[j]=0$。

Kuhn-Munkers 算法的正确性基于以下定理：
若由二分图中所有满足 $A[i]+B[j]=w[i,j]$ 的边 (i,j) 构成的子图（称做相等子图）有完备匹配，那么这个完备匹配就是二分图的最大权匹配。因为对于二分图的任意一个匹配，如果它包含于相等子图，那么它的边权和等于所有顶点的顶标和；如果它有的边不包含于相等子图，那么它的边权和小于所有顶点的顶标和。所以相等子图的完备匹配一定是二分图的最大权匹配。

Kuhn-Munkers 算法思路：
初始时为了使 $A[i]+B[j]>=w[i,j]$ 恒成立，令 $A[i]$ 为所有与顶点 Xi 关联的边的最大权，$B[j]=0$。如果当前的相等子图没有完备匹配，就按下面的方法修改顶标以使扩大相等子图，直到相等子图具有完备匹配为止。
我们求当前相等子图的完备匹配失败了，是因为对于某个 X 顶点，我们找不到一条从它出发的交错路。这时我们获得了一棵交错树，它的叶子结点全部是 X 顶点。现在我们把交错树中 X 顶点的顶标全都减小某个值 d，Y 顶点的顶标全都增加同一个值 d，那么我们会发现：

1. 两端都在交错树中的边 (i,j)，$A[i]+B[j]$ 的值没有变化。也就是说，它原来属于相等子图，现在仍属于相等子图。

2. 两端都不在交错树中的边 (i,j)，$A[i]$ 和 $B[j]$ 都没有变化。也就是说，它原来属于（或不属于）相等子图，现在仍属于（或不属于）相等子图。

3. X 端不在交错树中，Y 端在交错树中的边 (i,j)，它的 $A[i]+B[j]$ 的值有所增大。它原来不属于相等子图，现在仍不属于相等子图。

4. X 端在交错树中，Y 端不在交错树中的边 (i,j)，它的 $A[i]+B[j]$ 的值有所减小。也就说，它原来不属于相等子图，现在可能进入了相等子图，因而使相等子图得到了扩大。

现在的问题就是求 d 值了。
为了使 $A[i]+B[j]\geq w[i,j]$ 始终成立，且至少有一条边进入相等子图，d 应该等于：$Min(A[i]+B[j]-w[i,j]),Xi$ 在交错树中，Yi 不在交错树中。

Kuhn-Munkers 算法实现：
朴素的实现方法，时间复杂度为 $O(n^4)$: 需要找 $O(n)$ 次增广路，每次增广最多需要修改 $O(n)$ 次顶标，每次修改顶标时由于要枚举边来求 d 值，复杂度为 $O(n^2)$，总的复杂度为:$O(n^4)$。
实际上 KM 算法的复杂度是可以做到 $O(n^3)$ 的。我们给每个 Y 顶点一个“松弛量”函数 slack，每次开始找增广路时初始化为无穷大。在寻找增广路的过程中，检查边 (i,j) 时，如果它不在相等子图中，则让 $slack[j]$ 变成原值与 $A[i]+B[j]-w[i,j]$ 的较小值。这样，在修改顶标时，取所有不在交错树中的 Y 顶点的 $slack$ 值中的最小值作为 d 值即可。但还要注意一点：修改顶标后，要把所有的不在交错树中的 Y 顶点的 $slack$ 值都减去 d。

Kuhn — Munkras 算法流程：

1. 初始化可行顶标的值
2. 用匈牙利算法寻找完备匹配
3. 若未找到完备匹配则修改可行顶标的值
4. 重复 (2)(3) 直到找到相等子图的完备匹配为止

一些技巧：

- 最小权完备匹配：将所有的边权值取其相反数，求最大权完备匹配，匹配的值再取相反数即可。
- KM 算法的运行要求是必须存在一个完备匹配，求一个最大权匹配 (不一定完备)：把不存在的边权值赋为 0。
- 边权之积最大：每条边权取自然对数，然后求最大和权匹配，求得的结果 a 再算出 e^a 就是最大积匹配。需要注意精度问题。
- 当算法结束之后, 所有顶标之和最小. 即 $\sum(lx[i]+ly[i])(1 \le i \le n)$ 最小:
  例如 [UVALive 11383]: 有一个 $n*n$ 的矩阵, 需要定义行值 $row[i]$ 和列值 $col[j]$ 使得对于矩阵中的任意元素 $data[i][j] \le row[i]+col[j]$, 求最小的 $\sum(row[i]+col[i])$. 即所有行列值和最小. 那么就可以定义 $lx[i]=max(w[i][j])$,$ly[i]=0$ 分别代表 i 行的行值和 i 列的列值, 其中 $w[i][j]$ 是矩阵中的元素值. 显然这样取一定可以满足 $w[i][j] \le lx[i]+ly[j]$, 但是这样不能保证所有行列值和最小, 需要用 KM() 算法对行列值进行松弛. 当跑完 KM() 算法后的顶点坐标值 (即行列值)$lx[i]$ 和 $ly[i]$ 就是最优的了.

最小权匹配。以 POJ 2195 为例。

给出一个 $r*c$ 的矩阵，字母 H 代表房屋，字母 m 代表客人，房屋的数量和客人的数量相同。每间房只能住一个人。求这些客人全部住进客房的最少移动步数？

```
const int MAX_N = 400;

//求最大小权匹配时图的级别一般为() 10^2 ，所以用邻接矩阵存图
int r, c, n, m;
char s[MAX_N][MAX_N];
int match[MAX_N], visx[MAX_N], visy[MAX_N];
int lx[MAX_N], ly[MAX_N], w[MAX_N][MAX_N], slack[MAX_N];

struct Pos{
    int x,y;
}house[MAX_N * MAX_N],host[MAX_N * MAX_N];

inline bool dfs(int x)
{
    visx[x] = 1;
    for(int y = 0; y < m; y++){
        if(visy[y]) continue;
        int tmp = lx[x] + ly[y] - w[x][y];
        if( tmp == 0 ){
            visy[y] = 1;
            if(match[y] == -1 || dfs(match[y])){
                match[y] = x;
                return true; // 找到增广轨
            }
        }else {
            slack[y] = min(slack[y], tmp);
        }
    }
    return false;
    // 没有找到增广轨，说明顶点 X 没有对应的匹配
    // 与完备匹配（相等子图的完备匹配）不符
}

inline int KM()
{
    memset(match, -1, sizeof(match));
```

```
    memset(ly, 0, sizeof(ly));
    for(int i = 0; i < n; i++){
        lx[i] = INT_MIN;
        for(int j = 0; j < m; j++){
            lx[i] = max(lx[i], w[i][j]);
        }
    }
    for(int i = 0; i < n; i++){
        //初始边的松弛值为最大
        for(int j = 0; j < m; j++){ slack[j] = INT_MAX; }
        while(1){
            memset(visx, 0, sizeof(visx));
            memset(visy, 0, sizeof(visy));
            if(dfs(i)) break; //找到增广轨，则该点增广完成，进入下一点增广
            //没有找到增广轨需要改变顶标使图中可行边数量增加
            int d = INT_MAX;
            for(int j = 0; j < m; j++){
                if( !visy[j] ) d = min(d, slack[j]);
            }
            //增广轨（增广过程中遍历到）中 X 方顶标全部减去常数d
            for(int j = 0; j < n; j++) { if(visx[j]) lx[j] -= d; }
            //增广轨中 Y 方顶标全部增加d
            for(int j = 0; j < m; j++) {
                if(visy[j]) ly[j] += d;
                else slack[j] -= d; //不在增广轨中的顶点Y
            }
        }
    }
    int res = 0;
    for(int j = 0; j < m; j++) {
        if( match[j] != -1) res += w[match[j]][j];
    }
    return res;
}

int main()
{
    while(~scanf("%d%d",&r, &c) && (r || c)){
        n = m = 0;
        for(int i = 0; i < r; i++) {
            scanf("%s",s[i]);
            for(int j = 0; j < c; j++){
                if(s[i][j] == 'H'){
                    house[n].x = i;
                    house[n++].y = j;
                }else if(s[i][j] == 'm') {
                    host[m].x = i;
                    host[m++].y = j;
                }
            }
        }
        for(int i = 0; i < n; i++){
            for(int j = 0; j < m; j++){
                int tmp = abs(house[i].x - host[j].x) + abs(house[i].y - host[j].y);
                w[i][j] = -tmp; // 求最小权匹配，将边权取反，然后求最大权匹配
            }
        }
        int ans = KM();
        printf("%d\n",-ans); // 结果取相反数
    }
    return 0;
}
```

0.1.8 二分图带权匹配拆点

有 m 个作坊和 n 件物品，给出每件商品在每个作坊加工完成的时间，求出加工完 n 件商品的最少平均时间。每件商品只能在一个作坊完成，每个作坊同一时间只能加工一件商品，每件商品的完成时间需要加上它的等待时间。

假设一个作坊最终加工 k 件商品，加工顺序依次是从 1 到 k，则在该作坊加工的商品的总耗时是：

$$cost[1] + (cost[1] + cost[2]) + (cost[1] + cost[2] + cost[3]) + .. + (cost[1] + ... + cost[k])$$

则第 i 个商品对总耗时的贡献是:$cost[i] * (k - i + 1)$, 那么倒数第 i 个商品对总耗时的贡献是:$cost[i] * i$。

需要将 m 个作坊拆成 $n * m$ 个作坊，用 $w[i][j * n + p]$ 表示第 i 个商品在第 j 个作坊倒数第 p 个加工完成的权值 (对总耗时的贡献)。剩下的就是常规的将最小权匹配转换成最大权匹配做法。

```
for(int i = 0; i < n; i++){
    for(int j = 0; j < m; j++){
        int tmp;
        cin >> tmp;
        for(int k = 0; k < n; k++){
            w[i][j * n + k] = -(k + 1) * tmp;
        }
    }
}
m = n * m;
```

0.1.9 稳定婚姻匹配

GaleShapley Algorithm: 男子将一轮一轮地去追求他中意的女子，女子可以选择接受或者拒绝他的追求者。第一轮，每个男子都选择自己名单上排在首位的女子, 并向她表白。此时，一个女子可能面对的情况有三种：没有人跟她表白，只有一个人跟她表白，有不止一个人跟她表白。在第一种情况下，这个女子什么都不用做，只需要继续等待；在第二种情况下，接受那个人的表白，答应暂时和他匹配；在第三种情况下，从所有追求者中选择自己最中意的那一位，答应和他暂时匹配，并拒绝所有其他追求者。
第一轮结束后，有些男子已经匹配，有些男子仍然是单身。在第二轮追女行动中，每个单身男子都从所有还没拒绝过他的女子中选出自己最中意的那一个，并向她表白，不管她现在是否是单身。和第一轮一样，女子们需要从表白者中选择最中意的一位，拒绝其他追求者。注意，如果这个女子已经有匹配了，当她遇到了更好的追求者时，她必须放弃现有匹配，和更好的追求者形成新的匹配。这样，一些单身男子将会得到匹配，那些已经有了匹配的男子也可能重新变为单身。在以后的每一轮中，单身男子继续追求列表中的下一个女子，女子则从包括现男友在内的所有追求者中选择最好的一个，并对其他人说不。这样一轮一轮地进行下去，直到某个时候所有人都不再单身，下一轮将不会有任何新的表白发生，整个过程自动结束。此时的婚姻搭配就一定是稳定的了。

判断算法有穷性:
随着轮数的增加，总有一个时候所有人都能配对。由于在每一轮中，至少会有一个男子向某个女子告白，因此总的告白次数将随着轮数的增加而增加。倘若整个流程一直没有因所有人都配上对了而结束，最终必然会出现某个男子追遍了所有女子的情况。而一个女子只要被人追过一次，以后就不可能再单身了。既然所有女子都被这个男子追过，就说明所有女子现在都不是单身，也就是说此时所有人都已配对。

判断匹配稳定性
首先注意到，随着轮数的增加，一个男子追求的对象总是越来越糟，而一个女子的男友只可能变得越来越好。假设男 A 和女 1 各自有各自的对象，但比起现在的对象，男 A 更喜欢女 1。因此，男 A 之前肯定已经跟女 1 表白过。既然女 1 最后没有跟男 A 在一起，说明女 1 拒绝了男 A，也就是说她有了比男 A 更好的男孩儿。这就证明了，两个人虽然不是一对，但都觉得对方比自己现在的伴侣好，这样的情况绝不可能发生。

性质
这种男追女, 女拒男的方案对男性更有利。事实上，稳定婚姻搭配往往不止一种，然而上述算法的结果可以保证，每一位男性得到的伴侣都是所有可能的稳定婚姻搭配方案中最理想的，同时每一位女性得到的伴侣都是所有可能的稳定婚姻搭配方案中最差的。

时间复杂度:$O(n^2)$

有 n 对男女, 先给出每个女生对 n 位男生的选择意向, 排在前面的优先选择, 然后给出 n 位男生的选择意向, 排在前面的优先选择. 输出每位女生的匹配, 使得每位女生都是稳定的最佳选择.

下面算法中存储女生的选择意向时有两种选择: 队列和数组. 把注释部分去掉//即是队列写法.

```
const int MAX_N = 1010;

int T, n;
int x[MAX_N][MAX_N], y[MAX_N][MAX_N]; //x means girls, y means boys
//x[i][j] = k means ith girl's kth preference is jth boy
int matchx[MAX_N], matchy[MAX_N];
//matchx[i] is used to store ith girl's final match
int order[MAX_N];
//order[i] means the ith girl's choose order; initialized as 1
queue<int> girl[MAX_N]; // girl[i] is used to store ith girl's preference order

void GaleShapley()
{
    memset(matchy, -1, sizeof(matchy));
    // all boy' states are initialized as -1 to imply not match
    queue<int> single; //store all single girls' index
    for(int i = 1; i <= n; i++) { single.push(i); }
    while(!single.empty()){
        int tmpx = single.front(); //single girl
        single.pop();
        //int tmpy = girl[tmpx].front();
        // tmpx's current priority preference
        //girl[tmpx].pop();
        int tmpy = x[tmpx][order[tmpx]++];
        int cur = matchy[tmpy]; //The boy tmpy's current match
        if(cur == -1) {
            matchx[tmpx] = tmpy;
            matchy[tmpy] = tmpx;
        }else if(y[tmpy][tmpx] < y[tmpy][cur]){
        //The girl tmpx's preference is priority to the girl cur
            matchx[tmpx] = tmpy;
            matchy[tmpy] = tmpx;
            single.push(cur);
        }else {  //The girl tmpx doen't find match
            single.push(tmpx);
        }
    }
}

int main()
{
    cin >> T;
    while(T--> 0){
        cin >> n;
        for(int i = 1; i <= n; i++) { order[i] = 1; }
        for(int i = 1; i <= n; i++){
            //while(!girl[i].empty()) { girl[i].pop(); }
            for(int j = 1; j <= n; j++){
                cin >> x[i][j];
                //int t;
                //cin >> t;
                //girl[i].push(t);
            }
        }
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                int t;
                cin >> t;
                y[i][t] = j;
            }
        }
```

```
62          GaleShapley();
63          for(int i = 1; i <= n; i++){
64          //output each girl's matching result
65              cout << matchx[i] << endl;
66          }
67          if(T > 0) cout << endl;
68      }
69      return 0;
70  }
```