**// ======= 拓扑排序 =======**

for(int i=0; i<cnt; i++)//入度为0的点入栈

if(!indeg[i]) s.push(i);

while(!s.empty()){

int u=s.top(); s.pop();

T.push\_back(u); //T保存拓扑序

for(int v:G[u]){ //G为邻接表

indeg[v]--;

if(!indeg[v]) s.push(v);

}

}

**// ======= SCC强连通分量 =======**

int dfs\_clock = 0, pre[MaxN],low[MaxN];

int scc\_cnt, sccno[MaxN], size[MaxN];

stack<int> S; vector<int> G[MaxN];

void Add(int a,int b) {G[a].push\_back(b);}

void dfs(int u) {

pre[u] = low[u] = ++dfs\_clock;

S.push(u);

for (int i = 0; i < G[u].size(); i++) {

int v = G[u][i];

if (!pre[v]) {

dfs(v);

low[u] = min(low[u], low[v]);

}

else if (!sccno[v])

low[u] = min(low[u],pre[v]);

}

if (low[u] == pre[u]) {

scc\_cnt++;

int original\_size = S.size(), tmp = 0;

do {

tmp = S.top(); S.pop();

sccno[tmp] = scc\_cnt;

} while (tmp != u);

size[scc\_cnt] = original\_size - S.size();

} } // end of dfs

int main()

{ for (int i=0; i<n; i++) if (!pre[i]) dfs(i); }

**// ========= BCC 桥 割点 =========**

struct Edge{ int u,v; };

vector<int>G[MAXN],bcc[MAXN];

vector<Edge>bridge;

int low[MAXN],pre[MAXN],dfs\_clock=0,

iscut[MAXN],bccno[MAXN],bcc\_cnt=0;

//bccno是每个点在哪块 bcc是第i块有哪些点

//点编号0~n-1

stack<Edge>s;//保存在当前bcc中的边, 割顶的bccno无意义, 因为存在于多个bcc中

void dfs(int u, int fa)

{

low[u]=pre[u]=++dfs\_clock;

int len=G[u].size(),i,child=0;

for(i=0;i<len;i++)

{

int v=G[u][i];

Edge e = {u,v};

if(!pre[v]) //not accessed yet

{

s.push(e);//store cut edge

dfs(v,u);

child++;

low[u]=min(low[u],low[v]);

if(low[v] >= pre[u]) //if cut, bcc find

{

if(low[v] > pre[u])

bridge.push\_back(e);

iscut[u]=1;

bcc\_cnt++;//bcc start from 1

bcc[bcc\_cnt].clear();

while(1)

{

Edge x = s.top();

s.pop();

if(bccno[x.u]!=bcc\_cnt)

{

bcc[bcc\_cnt].push\_back(x.u);

bccno[x.u]=bcc\_cnt;

}

if(bccno[x.v]!=bcc\_cnt)

{

bcc[bcc\_cnt].push\_back(x.v);

bccno[x.v]=bcc\_cnt;

}

if(x.u==u && x.v==v) break;

}

}

}

else if( pre[u] > pre[v] && v!=fa )

//early than father

{

s.push(e);

low[u]=min(low[u],pre[v]);

}

}

if(child<=1 && fa<0)

iscut[u]=0;

}

int main() {

scanf("%d%d",&n,&m);

for(i=1;i<=m;i++) {

scanf("%d%d",&a,&b);

G[a].push\_back(b);

G[b].push\_back(a);

}

for(i=0;i<n;i++)

if(!pre[i])

dfs(i,-1);

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

if (iscut[i])

printf("node %d is cut\n", i);

for (auto e: bridge)

printf("bridge %d -- %d\n",e.u,e.v);

}

**// ========= 2-SAT =========**

// node indexed from 0~n-1

#include <cstring>

int n, cnt=0, sol[MAXN\*2];

vector<int> G[MAXN\*2];

bool mark[MAXN\*2];

void Clear() {

for (int i = 0; i < MAXN\*2; i++)

G[i].clear();

memset(mark, 0, sizeof(mark));

cnt = 0;

}

//x=xval or y = yval

void add\_constrain(int x,int xval,int y,int yval) {

//x is xval OR y is yval

x=2\*x+xval;

y=2\*y+yval;

G[x^1].push\_back(y); //!x->y

G[y^1].push\_back(x); //!y->x

}

bool dfs(int u) {

if(mark[u^1]) return false;

if(mark[u]) return true;

mark[u] = true;

sol[cnt++]=u; // stack

for(int i = 0;i<G[u].size();i++)

if(!dfs(G[u][i]))

return false;

return true;

}

bool twosat() {

for(int i=0;i<2\*n;i+=2)

if(!mark[i] && !mark[i+1]) { //未涂色

cnt=0;

if(!dfs(i)) { //出现contradiction

while(cnt) mark[sol[--cnt]]=0;

//i的结果全部不要

if(!dfs(i+1)) return false;

//always contradict

}

}

return true;

}

**// ========= 2-SAT SCC =========**

// node indexed from 0~n-1

#include <cstring>

int n, dfs\_clock=0, scc\_cnt=0;

int pre[MAXN\*2],low[MAXN\*2];

int sccno[MAXN\*2];

stack<int> S;

vector<int> G[MAXN\*2];

void Clear() {

for (int i = 0; i < n\*2; i++)

G[i].clear();

dfs\_clock = scc\_cnt = 0;

memset(pre, 0, sizeof(pre));

memset(low, 0, sizeof(low));

memset(sccno, 0, sizeof(sccno));

}

void add\_constrain(int x,int xval,int y,int yval)

{见2-sat普通版本的add\_constrain;}

void dfs(int u) { 间SCC部分的dfs; }

bool twosat() {

for (int i=0; i<n\*2; i++)

if (!pre[i])

dfs(i);

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

if (sccno[i\*2] == sccno[i\*2+1])

return false;

return true;

}

**// ========== SPFA 最短路==========**

vector<int> G[MaxN]; int tot=0,

v[MaxM],NEXT[MaxM],w[MaxM],head[MaxN]; //memset

int Dis[MaxN],Que[MaxQ]; bool vis[MaxN];

void AddEdge(int a,int b,int c)

{v[++tot]=b;w[tot]=c;NEXT[tot]=head[a];

head[a]=tot;}

void Spfa(int Source) {

memset(Dis,63,sizeof(Dis));

memset(vis, 0, sizeof(vis));

int Qhead=1, Qtail=1;

Dis[Que[1]=Source]=0;

while(Qhead<=Qtail) {

for(int i=head[Que[Qhead]];i;i=NEXT[i])

if(Dis[Que[Qhead]]+w[i]<Dis[v[i]]) {

Dis[v[i]]=Dis[Que[Qhead]]+w[i];

if(!vis[v[i]]) {

vis[v[i]]=1;

Que[++Qtail]=v[i];

}

} vis[Que[Qhead++]]=0;

}

} // main: S = ???; // Spfa();

**// ============ LCA ============**

struct Edge{int u,v,w;};

vector<Edge> edges;

vector<int> G[maxn];

int dep[maxn]; //在dfs树上的深度

int f[maxn][maxlog], g[maxn][maxlog];

//点index: 1~N (不能为0)

//init: f, g = 0

//f[i,j]记录i结点向上走2^j步后所到达的祖先

//g[i,j]记i结点向上走2^j步路途中边权最小值

void dfs(int u) {

for (int i=1;i<maxlog;i++) {

f[u][i] = f[f[u][i-1]][i-1];

g[u][i] = min(g[u][i-1], g[f[u][i-1]][i-1]);

}

for (int i=0; i<G[u].size(); i++) {

Edge& e = edges[G[u][i]];

int v = e.v, w = e.w;

if (!dep[v]) {

f[v][0]=u;

g[v][0]=w;

dep[v] = dep[u] + 1;

dfs(v);

}

}

}

int LCA(int a,int b) {

if (dep[a]>dep[b]) swap(a,b); //保证b更深

int Ans=INF;

for (int i=maxlog-1;i>=0;i--)

if (dep[f[b][i]]>=dep[a]){

Ans=min(Ans,g[b][i]);

b=f[b][i];

} //将b移动至与a同一深度

if (a==b) return a; //LCA=a=b

for (int i=maxlog-1;i>=0;i--)

if (f[a][i]!=f[b][i]){

Ans=min(Ans,min(g[a][i],g[b][i]));

a=f[a][i];b=f[b][i];

}//向上找到LCA

Ans=min(Ans,min(g[a][0],g[b][0]));

return f[a][0]; //LCA=f[a][0]-f[b][0]

}

// int main()

// 不连通的森林:

for (int i=1; i<=n; i++)

if (!dep[i]) { dep[i] = 1; dfs(i); }

// 牢记 dep[root] == 1 != 0

**// ========= dijkstra =========**

struct State {

int u, d; State (int u = 0, int d = 0): u(u), d(d){}

bool operator < (const State& another) const { return d > another.d; }

};

void dijkstra (int s) {

memset(vis, 0, sizeof(int) \* n);

memset(dis, inf, sizeof(int) \* n);

dis[s] = 0;

priority\_queue<State> que;

que.push(State(s, dis[s]));

while (!que.empty()) {

int u = que.top().u; que.pop();

if (vis[u]) continue; vis[u] = 1;

for (int i = 0; i < G[u].size(); i++)

{

Edge& edge = edges[G[u][i]];

int v = edge.v, w = edge.w;

if (dis[v] > dis[u] + w) {

dis[v] = dis[u] + w;

que.push(State(v, dis[v]));

}

}

}

}

**// ========= 二分图匹配 =========**

// O(V\*E)

// 点index不得为0

vector<int> G[maxn];

int link[maxn];

bool vis[maxn];

int ans = 0;

int dfs(int u) {

for (auto v : G[u])

if (!vis[v]) {

vis[v]=1;

if (!link[v] || dfs(link[v]))

{ link[v] = u; return 1; }

} return 0; }

int main() {

// 左半张图为1~m 右半张图为m+1~n

// 由左半张图向右半张图连单向边

for (int i=1;i<=M;i++)

{memset(vis,0,sizeof(vis)); if(dfs(i)) ans++;}

printf("%d\n", ans);

for (int i=M+1;i<=N;i++)

if (link[i]) printf("%d %d\n",link[i],i);

}

**// ========= 网络流相关笔记 =========**

========== 若带权值 ==========

最小点权覆盖 = 最大流

最大点权独立集合 = 总权值 - 最小点权覆盖

========== 不带权值 ==========

二分图最小顶点覆盖 = 二分图最大匹配；

二分图最大独立集 = 节点总数数（n）- 最大匹配数

求DAG图的最小路径覆盖:

(在图中找尽量少的路径，使得每个节点恰好在一条路径上)

1. 把原图中所有节点i拆成i与i’

2. 如果原图存在有向边i->j, 则在二分图中引入边i->j’

3. DAG最小路径覆盖 = 节点数（n）- 最大匹配数；

============================

网络流解法的构图：

超级源点与左边集合的每一点相连，若是求最小点覆盖，权值为1，若是求最小点权覆盖集，权值为该点的点权

超级汇点与右边集合的每一点相连，权值同上

左右集合之间练得边容量均为INF

**// ============= DINIC() =============**

// O(n^2\*m) // 二分图匹配 O(n^(0.5) \* m)

// all con[]==1 O(min(n^(2/3),m(1/2))\*m)

int m0 = 1, S, T; int head[MaxNode],u[MaxEdge],v[MaxEdge],

NEXT[MaxEdge],con[MaxEdge]; int Q[MaxQue],dis[MaxNode],cur[MaxNode],vis[MaxNode];

void Clear() {

// cstring memset(head u v NEXT con Q cur) to 0}

void Add(int a,int b,int c) {

v[++m0]=b;u[m0]=a;NEXT[m0]=head[a];head[a]=m0;con[m0]=c;

v[++m0]=a;u[m0]=b;NEXT[m0]=head[b];head[b]=m0;con[m0]=0;

}

int bfs() {

int Qhead=0,Qtail=0;

memset(vis,0,sizeof(vis));

memset(dis,127,sizeof(dis));

dis[S]=0;vis[S]=1;Q[++Qtail]=S;

while(Qhead<Qtail) {

++Qhead;

for(int i=head[Q[Qhead]];i;i=NEXT[i])

if(!vis[v[i]] && con[i]) {

vis[v[i]]=1;

dis[v[i]]=dis[Q[Qhead]]+1;

Q[++Qtail]=v[i];

}

}

return vis[T];

}

int dfs(int now,int lim) {

if (now==T || !lim) return lim;

int flow=0, f;

for (int& i=cur[now];i;i=NEXT[i]) {

if (dis[v[i]]>dis[now] && con[i])

if ((f=dfs(v[i],Min(lim-flow,con[i]))) > 0){

flow+=f;

con[i]-=f;

con[i^1]+=f;

if (flow==lim) break;

}

}

return flow;

}

int DINIC() {

int flow=0;

while(bfs()) {

memcpy(cur,head,sizeof(head));

flow+=dfs(S,INF);

}

return flow;

}

// Ans = DINIC();

**// ========== 费用流 ==========**

最小费用最大流m0=1;

void AddEdge(int a,int b,int c,int d)

{ v[++m0]=b;u[m0]=a;con[m0]=c;cost[m0]=d;prep[m0]=head[a];head[a]=m0;

v[++m0]=a;u[m0]=b;con[m0]=0;cost[m0]=-d;prep[m0]=head[b];head[b]=m0;}

bool spfa()

{ memset(Dis,127,sizeof(Dis)); memset(vis,0,sizeof(vis));

Dis[S]=0; vis[S]=1; Que[Qhead=Qtail=1]=S;

while(Qhead<=Qtail){

for(int

i=head[Que[Qhead]];i;i=prep[i]) if(con[i]&&Dis[v[i]]>Dis[Que[Qhead]]+cost[i]){

Dis[v[i]]=Dis[Que[Qhead]]+cost[i];

path[v[i]]=i;

if(!vis[v[i]])

vis[Que[++Qtail]=v[i]] = 1;

}

vis[Que[Qhead]]=0; ++Qhead;

} return Dis[T]<2100000000; }

void CostFlow() {

int x; Ans=0;

memset(path,0,sizeof(path));

while(spfa())

{

int f=INF;

for(x=T;x!=S;x=u[path[x]])

f=Min(f,con[path[x]]);

for(x=T;x!=S;x=u[path[x]])

{con[path[x]]-=f; con[path[x]^1]+=f; }

Ans+=Dis[T]\*(long long)f;

}}

S=1;T=2; CostFlow(); printf("%d\n",Ans);

**// ============= ISAP =============**

int source, sink, p[max\_nodes], num[max\_nodes], cur[max\_nodes], d[max\_nodes];

bool visited[max\_nodes];

struct Edge{

int from, to, cap, flow;

Edge(){}

Edge(int a, int b, int c, int d):from(a), to(b), cap(c), flow(d){}

};

int num\_nodes, num\_edges;

vector<Edge> edges;

vector<int> G[max\_nodes]; // 每个节点出发的边编号 正向边

// 预处理, 反向 BFS 构造 d 数组

void bfs(){

memset(visited, 0, sizeof(visited));

queue<int> Q; Q.push(sink);

visited[sink]=true; d[sink]=0;//距离为0

while(!Q.empty()){

int u=Q.front(); Q.pop();

for(auto ix=G[u].begin(); ix!=G[u].end(); ++ix){

Edge& e=edges[(\*ix)];//因为从汇点开始 用反向边

if(!visited[e.to])

visited[e.to]=true, d[e.to]=d[u]+1, Q.push(e.to);

}

}

}

int augment(){// 找到一条增广路 增广

int u=sink, rsd=0x7fffffff;

// 从汇点到源点通过 p 追踪增广路径, rsd 为一路上最小的残量

while(u!=source){

Edge& e=edges[p[u]];

rsd=min(rsd, e.cap-e.flow);

u=edges[p[u]].from;

}

u=sink;

while(u!=source){// 从汇点到源点更新流量

edges[p[u]].flow+=rsd; //正向边+流量 反向边-流量

edges[p[u]^1].flow-=rsd;

u=edges[p[u]].from;

}

return rsd;

}

int max\_flow(){

int flow=0;

bfs();

memset(num, 0, sizeof(num));

for(int i=0; i<num\_nodes; i++)

num[d[i]]++; //和t距离为i的节点

int u=source; //当前节点

memset(cur, 0, sizeof(cur));

while(d[source]<num\_nodes){//s到t的距离不能超过点数

if(u==sink) flow+=augment(), u=source;

bool advanced=false;

for(int i=cur[u]; i<G[u].size(); i++){

//当前弧优化 从第cur个点开始做 因为前cur-1个点都已经用干净了

Edge& e=edges[G[u][i]]; //正边

if(e.cap>e.flow&&d[u]==d[e.to]+1){//边上有残量 并且是一条 允许弧

advanced=true; p[e.to]=G[u][i]; //下一个点的 上一条弧

cur[u]=i; //更新u点的当前弧到u的第i个点

u=e.to; break;

}

}

if(!advanced){

// 当前(过时的)剩余网络下 u不能允许弧连接到t了

// retreat 更新分层图

// remark: u的邻接边不一定都是允许弧

// 所以更新到u邻接边的距离+1是新的剩余网络中的允许弧

int m=num\_nodes-1; //默认u的距离是最大值(从剩余网络中排除)

for(auto ix=G[u].begin(); ix!=G[u].end(); ++ix)

if(edges[\*ix].cap>edges[\*ix].flow) m=min(m, d[edges[\*ix].to]);

if(--num[d[u]]==0) break;//gap 优化, 如果和t距离d[u]的所有点都没了 s和t一定断开了 直接退出

d[u]=m+1, num[d[u]]++;

cur[u]=0;

if(u!=source) u=edges[p[u]].from; //retreat to 到u的前一个点

}

}

return flow;

}

**// ========= 次小生成树 =========**

// 增量最小MST: (m次加边求MST)回路性质 加边后删除生成树以外的所有边

// 最小瓶颈MST/路: (最大边最小) 原图MST满足瓶颈性质

// 次小生成树: 边uv和点uv之间的最小瓶颈(最大边权)边交换

int n, m, fa[MAXN], maxcost[MAXN][MAXN], pre[MAXN];

bool vis[MAXN];

struct Edge{

int u, v, w, inMST;

Edge(int u=0, int v=0, int dist=0):u(u), v(v), w(dist){inMST=0;}

}edge[MAXM];

vector<Edge> vec[MAXN];//MST

bool cmp(const Edge& a, const Edge& b){return a.w<b.w;}

int root(int x){return fa[x]==x?x:fa[x]=root(fa[x]);}

void kruskal(){

sort(edge, edge+m, cmp);

for(int i=1; i<=n; i++) fa[i]=i;

int cnt=0;

for(int i=0; i<m; i++){

int x=root(edge[i].u), y=root(edge[i].v);

if(x!=y){

fa[y]=x;

vec[edge[i].u].push\_back(Edge(edge[i].u, edge[i].v, edge[i].w));

vec[edge[i].v].push\_back(Edge(edge[i].v, edge[i].u, edge[i].w));

edge[i].inMST=1;

if(++cnt==n-1) break;

}

}

}

void dfs(int u){

vis[u]=1;

for(int i=0; i<vec[u].size(); i++){

int v=vec[u][i].v;

if(!vis[v]){ //access a new node

//u是v的父亲 在有根树中

for(int j=1; j<=n; j++) if(vis[j])//relax from all node visited

maxcost[j][v]=maxcost[v][j]=max(maxcost[j][u], vec[u][i].w);//j->v = max(j->u,u->v)

dfs(v);

}

}

}

int nextMST(){

int i, ans=0x3f3f3f3f;

for(i=0; i<m; i++){

Edge e=edge[i];

if(!e.inMST) ans=min(ans, e.w-maxcost[e.u][e.v]); //ans是边权增大了多少

}

return ans;

}

**// ========= 曼哈顿最小生成树 =========**

struct BIT {

int min\_val,pos;

void init()

{min\_val=INF;pos=-1;}

} bit[maxn];

void update(int x,int val,int pos){

for(int i=x;i>=1;i-=lowbit(i))

if(val<bit[i].min\_val)

bit[i].min\_val=val,bit[i].pos=pos;

}

int ask(int x,int m){

int min\_val=INF, pos=-1;

for(int i=x;i<=m;i+=lowbit(i))

if(bit[i].min\_val<min\_val)

min\_val=bit[i].min\_val,pos=bit[i].pos;

return pos;

}

int Manhattan\_minimum\_spanning\_tree(int n,Point \*p){

int a[maxn],b[maxn]; // tmp

for(int dir=0;dir<4;dir++){

//4种坐标变换

if(dir==1||dir==3){

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

swap(p[i].x,p[i].y);

}

else if(dir==2){

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

p[i].x=-p[i].x;

}

// 我们将坐标按X排序(Y为第二关键字)，将Y-X离散化

// 用BIT来维护，查询对于某一个(X0,Y0)

// 查询比(Y0-X0)大的中X1+Y1最小的点

sort(p, p + n);

for(int i=0;i<n;i++){

a[i]=b[i]=p[i].y-p[i].x;

}

sort(b, b + n);

int m = unique(b,b+n)-b;

for(int i=1;i<=m;i++)

bit[i].init();

//对于四种坐标变换 每次新建一个树状数组

for(int i=n-1;i>=0;i--){ // x从大到小 保证X1>=X0

int pos=lower\_bound(b,b+m,a[i])-b+1;

//BIT中从1开始

int ans=ask(pos,m);

if(ans!=-1) // dist函数计算的是曼哈顿距离

addedge(p[i].id,p[ans].id,dist(i,ans));

update(pos,p[i].x+p[i].y,i);

}

}

}

**// ========= 最小树形图 （朱刘）=========**

//如果没有root, 加虚拟根，和每个点权值是所有边权值和+1，最后答案减去(和+1)

struct edge{

int u, v, w;

edge(int u=0, int v=0, int w=0):u(u), v(v), w(w){}

};

int n, m, ans;

vector<edge> g;

int id[maxn], inw[maxn], v[maxn], pre[maxn];

//inw:最小入边, v:一个点属于哪个环, id: 重新建图的点编号

bool zhuLiuAlg(int root){

ans=0;

while(true){

for(int i=0; i<n; i++) inw[i]=INF, id[i]=-1, v[i]=-1, pre[i]=-1;

for(int i=0; i<g.size(); i++)

if(g[i].w<inw[g[i].v]&&g[i].v!=g[i].u)

inw[g[i].v]=g[i].w, pre[g[i].v]=g[i].u;

pre[root]=root, inw[root]=0;

//判断是否可能, 因为后边修改了权值, 可以直接加到答案里

for(int i=0; i<n; i++){

if(inw[i]==INF) return false;

ans+=inw[i];

}

//找圈 && 缩点，缩成的点编号，判断是否还有环

int idx=0;

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

if(v[i]==-1){

int t=i;

while(v[t]==-1) v[t]=i, t=pre[t];

if(v[t]!=i||t==root) continue;

id[t]=idx++;

for(int j=pre[t]; j!=t; j=pre[j]) id[j]=idx-1;

}

if(idx==0) return true; // 没有环了

for(int i=0; i<n; i++) if(id[i]==-1) id[i]=idx++;

// 重新建图

for(int i=0; i<g.size(); i++)

g[i].w-=inw[g[i].v], g[i].u=id[g[i].u], g[i].v=id[g[i].v];//减权值避免删边

n=idx; root=id[root];

}

}

**// =========== 稳定婚姻 ===========**

int n, m[maxn][maxn], wife[maxn], cur[maxn], w[maxn][maxn], hus[maxn];

queue<int> Q;

void solve(){

while(!Q.empty()){

int man=Q.front(); Q.pop();

int woman=m[man][cur[man]++];

if(!hus[woman]) hus[woman]=man, wife[man]=woman;//直接配对

else if(w[woman][man]<w[woman][hus[woman]]){//如果当前男生的更好, 抛弃现在的舞伴，重新配对

wife[hus[woman]]=0; //被抛弃的男生重新回到单身状态

Q.push(hus[woman]);

hus[woman]=man, wife[man]=woman;//新的一对

}else Q.push(man); //男生没人要

}

for(int i=1; i<=n; i++) printf("%d\n", wife[i]);

}

int main(){

int T; scanf("%d", &T);

while(T--){

while(!Q.empty()) Q.pop();

scanf("%d", &n);

for(int i=1; i<=n; i++){

for(int j=1; j<=n; j++) scanf("%d", &m[i][j]); //编号为i的男生第j喜欢的女生

cur[i]=1, wife[i]=0, Q.push(i);//男生i下一个要邀请对象, 男生i的舞伴编号

}

int x;

for(int i=1; i<=n; i++){

for(int j=1; j<=n; j++) scanf("%d", &x), w[i][x]=j; //女生i心目中，男生x的排名

hus[i]=0; //编号为i的女生的舞伴编号

}

solve(); if(T) puts("");

}

return 0;

}

**// =========== KM 最大权完全匹配 ===========**

//最小点覆盖 = 最大匹配

//最大独立集 = 最小边覆盖 = 点数 - 最大匹配

//最大团 = 补图的最大独立集

//最小路径覆盖: 原图拆点 = 点数 - 拆点图最大匹配

//求最小权完备匹配: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

//KM算法的运行要求是必须存在一个完备匹配，如果求一个最大权匹配(不一定完备):把不存在的边权值赋为0。

//求边权之积最大: 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

int G[MAXN][MAXN], ex\_girl[MAXN], ex\_boy[MAXN], match[MAXN], slack[MAXN], N, n, m;

bool vis\_girl[MAXN], vis\_boy[MAXN];

bool dfs(int girl){

vis\_girl[girl]=true;

for(int boy=0; boy<N; ++boy){

if(vis\_boy[boy]) continue; // 每一轮匹配 每个男生只尝试一次

int gap=ex\_girl[girl]+ex\_boy[boy]-G[girl][boy];

if(gap==0){ // 如果符合要求

vis\_boy[boy]=true;

if(match[boy]==-1||dfs(match[boy])){match[boy]=girl;return true;}// 找到一个没有匹配的男生 或者该男生的妹子可以找到其他人

}else slack[boy]=min(slack[boy], gap); // slack 可以理解为该男生要得到女生的倾心 还需多少期望值 取最小值 备胎的样子

}

return false;

}

int KM(){

memset(match, -1, sizeof match); memset(ex\_boy, 0, sizeof ex\_boy);

for(int i=0; i<N; ++i){// 每个女生的初始期望值是与她相连的男生最大的好感度

ex\_girl[i]=G[i][0];

for(int j=1; j<N; ++j)

ex\_girl[i]=max(ex\_girl[i], G[i][j]);

}

for(int i=0; i<N; ++i){// 尝试为每一个女生解决归宿问题

fill(slack, slack+N, INF);// 因为要取最小值 初始化为无穷大

while(1){

// 为每个女生解决归宿问题的方法是 ：如果找不到就降低期望值，直到找到为止

// 记录每轮匹配中男生女生是否被尝试匹配过

memset(vis\_girl, false, sizeof vis\_girl);

memset(vis\_boy, false, sizeof vis\_boy);

if(dfs(i)) break; // 找到归宿 退出

// 如果不能找到 就降低期望值

// 最小可降低的期望值

int d=INF;

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

if(!vis\_boy[j]) d=min(d, slack[j]);

for(int j=0; j<N; ++j){

if(vis\_girl[j]) ex\_girl[j]-=d;

// 所有访问过的女生降低期望值

if(vis\_boy[j]) ex\_boy[j]+=d;

// 所有访问过的男生增加期望值

else slack[j]-=d;

// 没有访问过的boy 因为girl们的期望值降低，距离得到女生倾心又进了一步！

}

}

}

int res=0;// 匹配完成 求出所有配对的好感度的和

for(int i=0; i<N; ++i) res+=G[match[i]][i];

return res;

}

**// ========= sublime 配置 =========**

{

"cmd": ["g++", "-std=c++11", "${file}", "-o", "${file\_base\_name}"],

"file\_regex": "^(..[^:]\*):([0-9]+):?([0-9]+)?:? (.\*)$",

"working\_dir": "${file\_path}",

"selector": "source.c, source.c++",

"variants":

[

{

"name": "Run",

"cmd": ["bash", "-c", "g++ -std=c++11 '${file}' -o '${file\_base\_name}' && open -a terminal '${file\_path}/${file\_base\_name}'"]

}

]

}