**Pick’s Theorem:**

#define SIZE 1005

typedef long long ll;

struct point{

ll x, y;

}para[SIZE];

int n;

ll areaOfPolygon2(){ //2\*Area of simple polygon para[], which has n points

para[n]=para[0];

ll area=0;

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

area+=(para[i].x)\*(para[i+1].y);

area-=(para[i+1].x)\*(para[i].y);

}

if(area<0LL) area\*=-1; //If points are in clockwise order

return area;

}

ll numberOfLatticePoints(point a, point b){ //Number of lattice points in line segment connecting a, b

ll xd=a.x>b.x ? a.x-b.x : b.x-a.x;

ll yd=a.y>b.y ? a.y-b.y : b.y-a.y;

return \_\_gcd(xd, yd); //All lattice points except one vertex

}

ll latticePointsInsidePoly(){ //Number of lattice points strictly inside polygon

ll area=areaOfPolygon2();

ll b=0;

for(int i=0; i<n; i++) b+=numberOfLatticePoints(para[i], para[i+1]);

return (area-b)/2+1;

}

**Primes For Hashing:**

500000003

500002597

1000002307

1000019903

**Diameter of a Tree:**

#define NODES 100105  
*/\**  
 *\* Assumes 0 based indexing of tree*  
 *\* Use clearAll for multiple test cases*  
 *\* Call proc() to calculate the tree's diametre,*  
 *\* nodes at the ends of the diameter,*   
 *\* and nodes at the centre of the diameter*   
*\*/*  
**struct** Tree{  
 vector <**int**> adj[NODES], centre; *//Adjacency List and nodes at centre of diameter*  
 vector <**int**> adjw[NODES]; *//Adjacence List of edge weights*  
 **int** dis[3][NODES];  
 **int** far[NODES]; *//Distance of farthest node from each node*  
 **int** end1, end2, n, dia; *//n=number of vertices of tree*  
 *//end1 and end2 are the nodes at the diameter;*  
 *//dia is the diametre of the tree*

**void** clearAll(){  
 centre.clear();  
 **for**(**int** i=0; i<NODES; i++){  
 adj[i].clear();  
 adjw[i].clear();  
 }  
 }  
 **void** addEdge(**int** u, **int** v, **int** w){ *//Adds edge between nodes u, v with weight w*  
 adj[u].push\_back(v);  
 adjw[u].push\_back(w);  
 adj[v].push\_back(u);  
 adjw[v].push\_back(w);  
 }  
 **void** dfs(**int** nd, **int** ind){  
 **int** v;  
 **for**(**int** i=0; i<adj[nd].size(); i++){  
 v=adj[nd][i];  
 **if**(dis[ind][v]==-1){  
 dis[ind][v]=adjw[nd][i]+dis[ind][nd];  
 dfs(v, ind);  
 }  
 }  
 }  
 **int** findmax(**int** ind){  
 **int** mx=0;  
 **for**(**int** i=1; i<n; i++){  
 **if**(dis[ind][mx]<dis[ind][i]) mx=i;  
 }  
 **return** mx;  
 }  
 **void** findDia(){  
 memset(dis, -1, **sizeof**(dis));  
 dis[0][0]=0;  
 dfs(0, 0);  
 end1=findmax(0);  
 dis[1][end1]=0;  
 dfs(end1, 1);  
 end2=findmax(1);  
 dis[2][end2]=0;  
 dfs(end2, 2);  
 dia=dis[1][end2];  
 }  
 **void** findFarthestNodes(){  
 **for**(**int** i=0; i<n; i++){  
 far[i]=max(dis[1][i], dis[2][i]);  
 }  
 }  
 **int** findMinFar(){  
 **int** mn=far[0];  
 **for**(**int** i=1; i<n; i++){  
 mn=min(mn, far[i]);  
 }  
 **return** mn;  
 }

**void** findCentres(){  
 **int** mn=findMinFar();  
 **for**(**int** i=0; i<n; i++){  
 **if**(far[i]==mn){  
 centre.push\_back(i);  
 }  
 }  
 }  
 **void** proc(){  
 findDia();  
 findFarthestNodes();  
 findCentres();  
 }  
};

**Min Cost Max Flow (Bellman Ford):**

**int** src, snk, nNode, nEdge;  
**int** fin[MAXN], pre[MAXN], dist[MAXN];  
**int** cap[MAXE], cost[MAXE], nxt[MAXE], to[MAXE], from[MAXE];  
  
**inline** **void** init(**int** \_src, **int** \_snk, **int** nodes){  
 memset(fin, -1, **sizeof**(fin));  
 nNode=nodes, nEdge=0;  
 src=\_src, snk=\_snk;  
}  
  
**inline** **void** addEdge(**int** u, **int** v, **int** \_cap, **int** \_cost){  
 from[nEdge]=u, to[nEdge]=v, cap[nEdge]=\_cap, cost[nEdge]=\_cost;  
 nxt[nEdge]=fin[u], fin[u]=nEdge++;  
 from[nEdge]=v, to[nEdge]=u, cap[nEdge]=0, cost[nEdge]=-(\_cost);  
 nxt[nEdge]=fin[v], fin[v]=nEdge++;  
}  
  
**bool** bellman(){  
 **int** iter, u, v, i;  
 **bool** flag=**true**;  
 memset(dist, 0x7f, **sizeof**(dist));  
 memset(pre, -1, **sizeof**(pre));  
 dist[src]=0;  
 **for**(iter=1; iter<nNode && flag; iter++){  
 flag=**false**;  
 **for**(u=0; u<nNode; u++){  
 **for**(i=fin[u]; i>=0; i=nxt[i]){  
 v=to[i];  
 **if**(cap[i] && dist[v]>dist[u]+cost[i]){  
 dist[v]=dist[u]+cost[i];  
 pre[v]=i;  
 flag=**true**;  
 }  
 }  
 }  
 }  
 **return** (dist[snk]<INF);  
}  
  
**int** mcmf(**int** &fcost){  
 **int** netflow, i, bot, u;  
 netflow=fcost=0;  
 **while**(bellman()){  
 bot=INF;  
 **for**(u=pre[snk]; u>=0; u=pre[from[u]]) bot=min(bot, cap[u]);  
 **for**(u=pre[snk]; u>=0; u=pre[from[u]]){  
 cap[u]-=bot;  
 cap[u^1]+=bot;  
 fcost+=bot\*cost[u];  
 }  
 netflow+=bot;  
 }  
 **return** netflow;  
}

**Max Flow (Dinitz):**

**int** res[NODES][NODES], nxt[NODES][NODES];  
**bool** adj[NODES][NODES];  
**int** q[NODES], lvl[NODES];  
**int** src, sink;  
**inline** **void** init(){  
 memset(res, 0, **sizeof**(res));  
 memset(nxt, -1, **sizeof**(nxt));  
 memset(adj, **false**, **sizeof**(adj));  
}  
  
**inline** **void** process\_graph(){  
 **for**(**int** i=0; i<NODES; i++){  
 **for**(**int** j=NODES-2; j>=0; j--){  
 nxt[i][j]=(adj[i][j+1]) ? j+1 : nxt[i][j+1];  
 }  
 }  
}  
  
**bool** bfs(){  
 memset(lvl, -1, **sizeof**(lvl));  
 lvl[src]=0;  
 **int** st=0, ed=1;  
 q[st]=src;  
 **int** u, v;  
 **while**(st!=ed){  
 u=q[st];  
 st++;  
 v=(adj[u][0]) ? 0:nxt[u][0];  
 **for**(; v!=-1; v=nxt[u][v]){  
 **if**(lvl[v]==-1 && res[u][v]){  
 q[ed]=v;  
 lvl[v]=lvl[u]+1;  
 ed++;  
 }  
 }  
 }  
 **return** (lvl[sink]!=-1);  
}  
  
**int** dfs(**int** u, **int** flow){  
 **if**(u==sink) **return** flow;  
 **int** v=(adj[u][0])? 0:nxt[u][0];  
 **int** tmp;  
 **for**(; v!=-1; v=nxt[u][v]){  
 **if**(res[u][v] && lvl[v]==lvl[u]+1){  
 tmp=dfs(v, min(flow, res[u][v]));  
 **if**(tmp){  
 res[u][v]-=tmp;  
 res[v][u]+=tmp;  
 **return** tmp;  
 }  
 }  
 }  
 **return** 0;  
}  
  
**int** max\_flow(){  
 process\_graph();  
 **int** res=0, tmp;  
 **while**(bfs()){  
 **while**((tmp=dfs(src, INF))){  
 res+=tmp;  
 }  
 }  
 **return** res;  
}