**IR E VIR**

  public static void main([String](http://www.google.com/search?q=allinurl%3AString+java.sun.com&bntl=1)[] args) throws [Exception](http://www.google.com/search?q=allinurl%3AException+java.sun.com&bntl=1) {

 //Reader r = new Reader("test/t");

 [Reader](http://www.google.com/search?q=allinurl%3AReader+java.sun.com&bntl=1) r = new [Reader](http://www.google.com/search?q=allinurl%3AReader+java.sun.com&bntl=1)();

 [BufferedWriter](http://www.google.com/search?q=allinurl%3ABufferedWriter+java.sun.com&bntl=1) w = new [BufferedWriter](http://www.google.com/search?q=allinurl%3ABufferedWriter+java.sun.com&bntl=1)(new [OutputStreamWriter](http://www.google.com/search?q=allinurl%3AOutputStreamWriter+java.sun.com&bntl=1)([System](http://www.google.com/search?q=allinurl%3ASystem+java.sun.com&bntl=1).out));



 int i = 0;

 for (;;) {

 int n = r.readInt(), m = r.readInt();

 if (n == 0 && m == 0) {

 break;

 }



 LinkedList<Integer>[] v = new [LinkedList](http://www.google.com/search?q=allinurl%3ALinkedList+java.sun.com&bntl=1)[n];

 for (int j = 0; j < v.length; j++) {

 v[j] = new LinkedList<Integer>();

 }

 for (int j = 0; j < m; j++) {

 int a = r.readInt() - 1;

 int b = r.readInt() - 1;

 v[a].add(b);

 if (r.readInt() == 2) {

 v[b].add(a);

 }

 }



 boolean f = false;

 if (dfs(v)) {

 transpose(v);

 f = dfs(v);

 }



 w.write(f ? '1' : '0');

 w.write('\n');

 w.flush();

 }



 }



 private static void transpose(LinkedList<Integer>[] v) {

 // for (int i = 0; i < v.length; i++) {

 // System.out.println(v[i]);

 // }



 LinkedList<Integer>[] t = new [LinkedList](http://www.google.com/search?q=allinurl%3ALinkedList+java.sun.com&bntl=1)[v.length];

 for (int j = 0; j < v.length; j++) {

 t[j] = new LinkedList<Integer>();

 }



 for (int i = 0; i < v.length; i++) {

 for ([Integer](http://www.google.com/search?q=allinurl%3AInteger+java.sun.com&bntl=1) j : v[i]) {

 t[j].add(i);

 }

 }

 for (int i = 0; i < t.length; i++) {

 v[i] = t[i];

 }



 // System.out.println("-------");

 // for (int i = 0; i < v.length; i++) {

 // System.out.println(v[i]);

 // }

 }



 private static boolean dfs(LinkedList<Integer>[] v) {

 // DFS(G)

 int[] color = new int[v.length];

 int[] predecessor = new int[v.length];

 int[] distancia = new int[v.length];

 int time = 0;



 int i = 0;

 //for (int i = 0; i < v.length; i++) {

 for ([Integer](http://www.google.com/search?q=allinurl%3AInteger+java.sun.com&bntl=1) j : v[i]) {

 if (color[j] == 0) {

 dfs\_visit(v, color, j, distancia, time, predecessor);

 }

 }

 //}



 // System.out.println(Arrays.toString(color));

 for (int j = 0; j < color.length; j++) {

 if (color[j] != 2)

 return false;

 }

 return true;

 // for each vertex u E V[G]

 // color[u] <- white

 // pi[u] <- NIL

 // time <- 0

 // for each u E V[G]

 // if color[u] = white

 // DFS visit(u)

 }



 private static void dfs\_visit(LinkedList<Integer>[] v, int[] color, int j, int[] distancia, int time, int[] predecessor) {

 // DFS visit(u)

 // color[u] <- gray

 // d[u] <- time <- time + 1

 // for each v E Adj[u]

 color[j] = 1;

 distancia[j] = (time = time + 1);



 for ([Integer](http://www.google.com/search?q=allinurl%3AInteger+java.sun.com&bntl=1) i : v[j]) {

 if (color[i] == 0) {

 predecessor[i] = j;

 dfs\_visit(v, color, i, distancia, time, predecessor);

 color[j] = 2;

 }

 }

 color[j] = 2;

 // if color[u] = white

 // pi[v] <- u

 // DFS visit(v)

 // color[u] <- black

 // f [u] <- time <- time + 1

 }

**Estruturas**

O número de vértices será sempre n e o de arestas m. Os números máximos de vértices e arestas são MAXN e MAXM, respectivamente. O maior grau possível é MAXD.

**Matriz de Adjacências**

Representação: (G[][], n)

int G[MAXN][MAXN];  
int n;

**Listas de Adjacências**

Representação: (G[][], grau[], n, m)

int G[MAXN][MAXD];  
int grau[MAXN];  
int n, m;

**Lista de Arestas**

Representação: (E[], n, m)

int E[MAXM][2];  
int n, m;

**Matriz de Adjacências Bipartida**

Representação: (G[][], n, m)

int G[MAXN][MAXM];  
int n, m;

**Buscas**

**Busca em Largura (BFS)**

*Entrada*:

* Grafo por Listas de Adjacências (G[][], grau[], n, m).
* Nó inicial no\_inicial.

*Complexidade*: O(n + m)

Global:

int fila[MAXN];  
int visitado[MAXN];

main:

int ini, fim;  
  
for(int i=0; i<n; i++)  
 visitado[i] = 0;  
visitado[no\_inicial] = 1;  
  
ini = fim = 0;  
fila[fim++] = no\_inicial;  
  
while(ini != fim) {  
 int no = fila[ini++];  
  
 Processa\_No(no);  
  
 for(int i=0; i<grau[no]; i++) {  
 int viz = G[no][i];  
 if(!visitado[viz]) {  
 fila[fim++] = viz;  
 visitado[viz] = 1;  
 }  
}

**Busca em Profundidade (DFS)**

*Entrada*:

* Grafo por Listas de Adjacências (G[][], grau[], n, m).
* Nó inicial no\_inicial.

*Complexidade*: O(n + m)

Global:

int visitado[MAXN];  
  
void DFS(int no) {  
 visitado[no] = 1;  
  
 PreProcessa(no);  
   
 for(int i=0; i<grau[no]; i++)  
 if(!visitado[ G[no][i] ])  
 DFS( G[no][i] );  
  
 PosProcessa(no);  
}

main:

for(int i=0; i<n; i++)  
 visitado[i] = 0;  
  
DFS(no\_inicial);

**Árvore Geradora Mínima**

**Algoritmo de Prim**

*Entrada*:

* Grafo por Matriz de Adjacências (G[][], n).

*Saída*:

* Custo da Árvore Geradora Mínima em total.

*Complexidade*: O(n^2)

Global:

#include <values.h>  
const int INF = MAXINT/2;  
  
int fixo[MAXN];  
int custo[MAXN];

main:

int total = 0;  
  
for(int i=0; i<n; i++) {  
 fixo[i] = 0;  
 custo[i] = INF;  
}  
custo[0] = 0;  
  
for(int faltam = n; faltam>0; faltam--) {  
 int no = -1;  
 for(int i=0; i<n; i++)  
 if(!fixo[i] && (no==-1 || custo[i] < custo[no]))  
 no = i;  
 fixo[no] = 1;  
  
 if(custo[no] == INF) {  
 total = INF;  
 break;  
 }  
 total += custo[no];  
  
 for(int i=0; i<n; i++)  
 if(custo[i] > G[no][i])  
 custo[i] = G[no][i];  
}

**Caminhos Mínimos**

**Busca em Largura (BFS)**

*Entrada*:

* Grafo por Listas de Adjacências (G[][], grau[], n, m) sem peso nas arestas.
* Vértice de origem orig.

*Saída*:

* Vetor de distâncias dist[] do vértice orig a todos os outros.

*Complexidade*: O(n + m)

Global:

#include <values.h>  
const int INF = MAXINT/2;  
  
int fila[MAXN];  
int dist[MAXN];

main:

int ini, fim;  
  
for(int i=0; i<n; i++)  
 dist[i] = INF;  
dist[orig] = 0;  
  
ini = fim = 0;  
fila[fim++] = orig;  
  
while(ini != fim) {  
 int no = fila[ini++];  
  
 for(int i=0; i<grau[no]; i++) {  
 int viz = G[no][i];  
 if(dist[viz] == INF) {  
 fila[fim++] = viz;  
 dist[viz] = dist[no] + 1;  
 }  
 }  
}

**Algoritmo de Dijkstra**

*Entrada*:

* Grafo por Matriz de Adjacências (G[][], n) com peso nas arestas.
* Vértice de origem orig.

*Saída*:

* Vetor de distâncias dist[] do vértice orig a todos os outros.

*Complexidade*: O(n^2)

Global:

#include <values.h>  
const int INF = MAXINT/2;  
  
int fixo[MAXN];  
int dist[MAXN];

main:

for(int i=0; i<n; i++) {  
 fixo[i] = 0;  
 dist[i] = INF;  
}  
dist[0] = 0;  
  
for(int faltam = n; faltam>0; faltam--) {  
 int no = -1;  
 for(int i=0; i<n; i++)  
 if(!fixo[i] && (no==-1 || dist[i] < dist[no]))  
 no = i;  
 fixo[no] = 1;  
  
 if(dist[no] == INF)  
 break;  
  
 for(int i=0; i<n; i++)  
 if(dist[i] > dist[no]+G[no][i])  
 dist[i] = dist[no]+G[no][i];  
}

**Algoritmo de Floyd-Warshall**

*Entrada*:

* Grafo por Matriz de Adjacências (G[][], n) com peso nas arestas.

*Saída*:

* Matriz de distâncias G[][] entre todos os pares de vértices.

*Complexidade*: O(n.m + n^2)

Global:

#include <values.h>  
const int INF = MAXINT/2;

main:

for(int k=0; k<n; k++)  
 for(int i=0; i<n; i++)  
 if( i!=k && G[i][k]<INF )  
 for(int j=0; j<n; j++)  
 if( G[i][j] > G[i][k]+G[k][j] )  
 G[i][j] = G[i][k]+G[k][j];

**Bipartite Matching**

*Entrada*:

* Grafo por Matriz de Adjacências Bipartida (G[][], n, m).

*Saída*:

* Número máximo de pares casados como valor de retorno.
* Vetores conjuge0[] e conjuge1[] que indicam um possível *matching* que alcança o tamanho máximo.

*Complexidade*: O(n^3)

Global:

int conjuge0[MAXN];  
int conjuge1[MAXM];  
int visitado[MAXM];  
  
int Aumenta(int no) {  
 int i;  
 for(i=0; i<m; i++)  
 if(G[no][i]==1 && !visitado[i]) {  
 visitado[i]=1;  
 if(conjuge1[i]==-1 || Aumenta(conjuge1[i])) {  
 conjuge0[no] = i;  
 conjuge1[i] = no;  
 return 1;  
 }  
 }  
 return 0;  
}  
  
int Matching() {  
 int i;  
 int aumentou;  
 int ncasados;  
  
 for(i=0; i<n; i++)  
 conjuge0[i] = -1;  
 for(i=0; i<m; i++)  
 conjuge1[i] = -1;  
  
 ncasados=0;  
 do {  
 aumentou=0;  
 for(i=0; i<m; i++)  
 visitado[i] = 0;  
 for(i=0; i<n; i++)  
 if(conjuge0[i]==-1)  
 if(Aumenta(i)) {  
 aumentou = 1;  
 ncasados++;  
 }  
 } while(aumentou);  
  
 return ncasados;  
}

**Produto Escalar**

Calcula o produto escalar entre dois pontos.

*Entrada*:

* Pontos pta e ptb e origem pto.

*Saída*:

* Valor do produto escalar (pta - pto) . (ptb - pto).

*Complexidade*: O( 1 )

Global:

long long prodesc(int pto[2], int pta[2], int ptb[2]) {  
 long long v1 = ((long long) pta[0]-pto[0])\*(ptb[0]-pto[0]);  
 long long v2 = ((long long) pta[1]-pto[1])\*(ptb[1]-pto[1]);  
   
 return v1+v2;  
}

**Produto Vetorial**

Calcula o produto vetorial entre dois pontos.

*Entrada*:

* Pontos pta e ptb e origem pto.

*Saída*:

* Valor do produto vetorial (pta - pto) X (ptb - pto).

*Complexidade*: O( 1 )

Global:

long long prodvet(int pto[2], int pta[2], int ptb[2]) {  
 long long v1 = ((long long) pta[0]-pto[0])\*(ptb[1]-pto[1]);  
 long long v2 = ((long long) pta[1]-pto[1])\*(ptb[0]-pto[0]);  
   
 return v1-v2;  
}

**Produto Vetorial (Sinal)**

Calcula o sinal do produto vetorial entre dois pontos de forma "segura", evitando overflow.

*Entrada*:

* Pontos pta e ptb e origem pto.

*Saída*:

* Sinal do produto vetorial (pta - pto) X (ptb - pto).

*Complexidade*: O( 1 )

Global:

int prodvetsn(int pto[2], int pta[2], int ptb[2]) {  
 long long v1 = ((long long) pta[0]-pto[0])\*(ptb[1]-pto[1]);  
 long long v2 = ((long long) pta[1]-pto[1])\*(ptb[0]-pto[0]);  
  
 if( v1 < v2 ) return -1;  
 else if( v1 > v2 ) return +1;  
 else return 0;  
}

**Teste de Pertinência de Ponto em Segmento**

Testa se um ponto pertence a um segmento. Depende de prodesc e prodvet.

*Entrada*:

* Ponto p e segmento (a,b).

*Saída*:

* Valor booleano que indica se o segmento contém o ponto ou não.

*Complexidade*: O( 1 )

Global:

bool interPtSeg(int p[2], int a[2], int b[2]) {  
 if(prodesc(a,b,b)==0) // B == A.  
 return prodesc(a,p,p)==0;  
 else  
 return prodvet(p,a,b)==0 && prodesc(a,p,b)>=0 && prodesc(b,p,a)>=0;  
}

**Distância de Ponto a Segmento**

Retorna o quadrado da distância de um ponto a um segmento.  A computação é exata, não estando sujeita a erros de precisão. Depende de prodesc e prodvet.

*Entrada*:

* Ponto p e segmento (a,b).

*Saída*:

* num e den inteiros tais que num/den == dist(P, AB)^2.

*Complexidade*: O( 1 )

Global:

void dist2PtSeg(int p[2], int a[2], int b[2], long long \*num, long long \*den) {  
 if(prodesc(a,b,b)>0 && prodesc(a,p,b)>=0 && prodesc(b,p,a)>=0) {  
 \*num = prodvet(p,a,b); \*num \*= \*num;  
 \*den = prodesc(a,b,b);  
 } else {  
 \*num = min(prodesc(a,p,p), prodesc(b,p,p));  
 \*den = 1L;  
 }  
}

**Teste de Pertinência de Ponto em Polígono**

Testa se um ponto pertence a um polígono. Ponto na borda é considerado contido. Depende de interPtSeg.

*Entrada*:

* Ponto p e polígono (poly[][2], n), onde n é o numero de pontos em poly.

*Saída*:

* Valor booleano que indica se o polígono contém o ponto ou não.

*Complexidade*: O( n )

Global:

bool interPtPoly(int p[2], int (\*poly)[2], int n) {  
 // Testa borda.  
 for(int i=0; i<n; i++)  
 if(interPtSeg(p, poly[i], poly[(i+1)%n]))  
 return true;  
  
 // Testa ponto estritament dentro do polígono.  
 double anguloTotal = 0;  
 for(int i=0; i<n; i++) {  
 int \*a, \*b;  
 a = poly[i];   
 b = poly[(i+1)%n];  
  
 double angulo = atan2(b[0]-p[0],b[1]-p[1]) - atan2(a[0]-p[0],a[1]-p[1]);  
 if(angulo>PI)  
 angulo -= 2\*PI;  
 else if(angulo<-PI)  
 angulo += 2\*PI;  
 anguloTotal += angulo;  
 }  
   
 return fabs(anguloTotal) > PI; // fora: 0, dentro: 2\*PI  
}

**Teste de Interseção de Segmentos**

Testa se dois segmentos possuem interseção não vazia. Depende de prodvetsn, MIN e MAX.

*Entrada*:

* Segmentos (a,b) e (c,d).

*Saída*:

* Valor booleano que indica se os segmentos se intersectam ou não.

*Complexidade*: O( 1 )

Global:

bool interSegSeg(int a[2], int b[2], int c[2], int d[2]) {  
 int i, r1, r2;  
 for(i=0; i<2; i++)  
 if( MIN(a[i],b[i]) > MAX(c[i],d[i]) || MAX(a[i],b[i]) < MIN(c[i],d[i]) )  
 return 0;  
  
 r1 = prodvetsn(a, c, b) \* prodvetsn(a, d, b);  
 r2 = prodvetsn(c, a, d) \* prodvetsn(c, b, d);  
 return r1<=0 && r2<=0;  
}

**Módulo**

*Entrada*:

* Inteiros a (podendo ser negativo!) e n > 0.

*Saída*:

* Valor de a módulo n.

*Complexidade*: O( 1 )

Global:

int mod(int a, int n) {  
 return (a%n + n)%n;  
}

**Exponenciação Modular Rápida**

*Entrada*:

* Inteiros a, b e n.

*Saída*:

* Valor de a^b mod n.

*Complexidade*: O( log(b) )

Global:

int expmod(int a, int b, int n) {  
 if(b == 0)  
 return 1;  
 else {  
 long long res = expmod(a, b/2, n);  
 res = (res\*res) % n;  
 if(b%2 == 1)  
 res = (res\*a) % n;  
 return (int) res;  
 }  
}

**Máximo Divisor Comum**

Utiliza o algoritmo de Euclides.

*Entrada*:

* Inteiros a e b.

*Saída*:

* Maior inteiro que divide a e b.

*Complexidade*: O( log(a) + log(b) )

Global:

int mdc(int a, int b) {  
 if(a<0) a = -a;  
 if(b<0) b = -b;  
  
 if(b == 0)  
 return a;  
 else  
 return mdc(b, a%b);  
}

**Máximo Divisor Comum Estendido**

Utiliza o algoritmo de Euclides estendido.

*Entrada*:

* Inteiros positivos a e b.

*Saída*:

* Maior inteiro que divide a e b como retorno.
* Variáveis inteiras x e y tais que a.x + b.y = mdc(a,b).

*Complexidade*: O( log(a) + log(b) )

Global:

int mdc(int a, int b, int \*x, int \*y) {  
 int xx, yy, d;  
 if(b==0) {  
 \*x=1; \*y=0;  
 return a;  
 }  
  
 d = mdc(b, a%b, &xx, &yy);  
 \*x = yy;  
 \*y = xx - a/b\*yy;  
 return d;  
}

**Teste de Primalidade**

*Entrada*:

* Inteiro n.

*Saída*:

* Valor booleano que indica se n é primo ou não.

*Complexidade*: O( sqrt(n) )

Global:

int ehPrimo(int n) {  
 if(n==2) return 1;  
 if(n<=1 || n%2 == 0) return 0;  
  
 for(int i=3; i\*i<=n; i+=2)  
 if(n%i == 0)  
 return 0;  
  
 return 1;  
}

**Crivo de Eratóstenes**

*Entrada*:

* Inteiro n <= MAXN.

*Saída*:

* Vetor ehprimo[] que indica se os números menores ou iguais a n são primos ou não.

*Complexidade*: O( n.log(n) )

Global:

int ehprimo[MAXN+1];  
  
void achaPrimos(int n) {  
 ehprimo[0] = ehprimo[1] = 0;  
 ehprimo[2] = 1;  
  
 for(int i=3; i<=n; i++)  
 ehprimo[i] = i%2;  
  
 for(int i=3; i\*i<=n; i+=2)  
 if(ehprimo[i])  
 for(int j=i\*i; j<=n; j+=i)  
 ehprimo[j] = 0;  
}

**Função 'phi' de Euler**

*Entrada*:

* Inteiro n > 0.

*Saída*:

* Valor ϕ(n), onde ϕ é a função *phi* de Euler, equivalente ao número de primos relativos a n (mdc=1) menores o iguais a n.

*Complexidade*: O( n )

*Dependências*:

* Crivo de Eratóstenes: ehprimo[i], para i <= n/2.

Global:

int phi(int n) {  
 int res = n;  
 for(int p=2; 2\*p<=n; p++)   
 if(ehprimo[p] && n%p==0)  
 res = res/p\*(p-1);  
 return res;  
}

**Resolvedor de Equação Modular Linear**

*Entrada*: Inteiro a e b quaiquer e n positivo.

*Saída*: Menor valor positivo x tal que a.x == b (mod n) ou -1 se não houver solução.

*Complexidade*: O( n )

*Dependências*:

* Módulo: mod(int a, int n).
* MDC Estendido: mdc(int a,int b,int \*x, int\*y).

Global:

int solve(int a, int b, int n) {  
 int d, x, y;  
   
 a = mod(a, n);  
 b = mod(b, n);  
 d = mdc(a, n, &x, &y);  
 if(b%d==0)  
 return mod( ((long long)x%(n/d)) \* ((b/d)%(n/d)) , n/d );  
 else  
 return -1;  
}