**// Grafos.**

**// DFS y BFS.**

**int** parent[MAX]; seen[MAX];

**bool** BFS(**int** s, **int** t) {

**queue** <int> q;

memset(seen, 0, sizeof(seen));

parent[s] = -1; seen[s] = 1;

q.**push**(s);

**while**(!q.**empty**()) {

s = q.**front**();

q.**pop**();

**if** (s == t) **break**;

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

**if** (!seen[i] && C[s][i] > 0)

parent[i] = s, q.**push**(i);

}

**return** seen[t] != 0;

}

**// Articulation Points**

**int** tim;

**bool** artic[MV];

**int** d[MV] , low[MV], seen[MV], parent[MV];

**void** dfs(**int** x) {

seen[x]=1;

low[x]=d[x]=tim++;

**for**(**int** i=0; i<deg[x]; i++)

**if**(!seen[ady[x][i]]) {

parent[ady[x][i]]=x;

dfs(ady[x][i]);

**if**(low[x]>low[ady[x][i]])

low[x]=low[ady[x][i]];

**if**(low[ady[x][i]]>=d[x])

artic[x]=**true**;

}

**else** **if**(ady[x][i]!=parent[x])

low[x]<?=d[ady[x][i]];

}

**void** dfs\_f(**int** n) {

memset(artic,0,sizeof(artic));

memset(seen,0,sizeof(seen));

memset(parent,-1,sizeof(parent));

tim=0;

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

**if**(!seen[i]) {

seen[i]=1; low[i]=d[i]=tim++; **int** nh=0;

**for**(**int** j=0; j<deg[i]; j++)

**if**(!seen[ady[i][j]]) {

nh++; parent[ady[i][j]]=i;

dfs(ady[i][j]);

}

**if**(nh>=2) artic[i]=**true**;

} }

**// Deteccion de puentes (Bridges).**

**void** dfs(**int** x) {

seen[x]=1; low[x]=d[x]=tim++;

**for**(**int** i=0; i<deg[x]; i++)

**if**(!seen[ady[x][i]]) {

parent[ady[x][i]]=x;

dfs(ady[x][i]);

**if**(low[x]>low[ady[x][i]])

low[x]=low[ady[x][i]];

**if**(low[ady[x][i]]==d[ady[x][i]]) {

//x - ady[x][i] es un Puente

}

**else** **if**(ady[x][i]!=parent[x])

low[x]<?=d[ady[x][i]];

} }

**void** dfs\_f(**int** n) {

memset(seen,0,sizeof(seen));

memset(parent,-1,sizeof(parent));

tim=0;

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

**if**(!seen[i]) dfs(i);

}

**// Ciclo de Euler**

Existencia del ciclo de Euler. Un grafo tiene un ciclo de Euler si y solo si (i) está conectado y (2) todos sus vértices tienen grado par. Existencia de un camino de Euler. Un grafo tiene un camino de Euler si y solo si (i) está conectado y (2) exactamente 2 de sus vértices tienen grado impar, los cuales constituyen el inicio y el n del camino. Algoritmo para encontrar el camino de Euler. Encontrar ciclos de vértices disjuntos e irlos uniendo.

**int** tour(**int** x) {

**int** w,v=x;

**bool** stilledges=**true**;

**while**(stilledges) {

stilledges=**false**;

**for**(**int** i=0; i<MB; i++)

**if**(mat[v][i]) {

w=i; stilledges=**true**;

**break**;

}

**if**(stilledges) {

S.**push**(v);

mat[v][w]--; mat[w][v]--;

v=w;

} }

**return** v;

}

//x contiene el vertice por el cual empieza el ciclo

**void** find\_euler(**int** x) {

**int** v=x;

**bool** f=**true**; //solo se usa para imprimir bien

**while**(tour(v)==v&&!S.**empty**()) {

v=S.**top**(); S.**pop**();

**if**(f) {

**cout**<<x+1<<' '<<v+1<<**endl**;

**cout**<<v+1<<' ';

}

**else** {

**cout**<<v+1<<**endl**;

**if**(!S.**empty**()) **cout**<<v+1<<' ';

}

f=**false**;

} }

**// Grafo cactus**

**struct** eje { **int** t,i; };

**typedef** **vector**<eje> cycle;

**int** n,m,us[MAXM],pa[MAXN],epa[MAXN],tr[MAXM];

**vector**<eje> ady[MAXN];

**void** iniG(**int** nn) {

n=nn; m=0;

**fill**(ady,ady+n,**vector**<eje>()); **fill**(pa,pa+n,-1);

}

//f:from t:to d:0 si no es dirigido y 1 si es dirigido

**void** addE(**int** f, **int** t, **int** d) {

ady[f].**push\_back**((eje) {t,m});

**if**(!d) ady[t].**push\_back**((eje){f,m}), tr[m]=0;

us[m++]=0;

}

//devuelve false si algun eje esta en mas de un ciclo

**bool** cycles(**vector**<cycle>& vr,**int** f=0,**int** a=-2,**int** ai=-2) {

**int** t; pa[f]=a; epa[f]=ai;

**for**(**int** i=0; i<ady[f].**size**(); i++)

**if**(!tr[ady[f][i].i]++)

if(pa[t=ady[f][i].t]!=-1) {

cycle c(1,ady[f][i]); **int** ef=f;

**do** {

**if**(!ef) **return** 0;

eje e=ady[pa[ef]][epa[ef]];

**if**(us[e.i]++) **return** 0;

c.**push\_back**(e);

} **while**((ef=pa[ef])!=t);

vr.**push\_back**(c);

} **else** **if**(!cycles(vr,t,f,i)) **return** 0;

**return** 1;

};

**// Determinar si un grafo es bipartito**

Determinar si un grafo es bipartito es equivalente a determinar si el grafo puede ser coloreado con 2 colores, de tal forma que no haya dos vértices compartiendo el mismo color, lo cual a su vez es equivalente a determinar si el grafo no tiene un ciclo de longitud impar.

memset(col,-1,sizeof(col));

**if**(dfs(0,0)) //entonces es bipartite

**bool** dfs(**int** x,**int** color) {

col[x]=(color+1)%2;

**for**(**int** i=0; i<deg[x]; i++)

**if**(col[adj[x][i]]==-1) {

**if**(!dfs(adj[x][i],col[x])) **return false**;

}

**else** **if**(col[adj[x][i]]==col[x]) **return false**;

**return true**;

}

**// Ordenamiento Topologico.**

**int** indeg[MAXV];

//contiene el numero de aristas que entran al vértice

**queue**<int> q;

**for**(**int** i=0; i<n; i++) **if**(!indeg[i]) q.**push**(i);

**while**(!q.**empty**()) {

**int** x=q.**front**(); q.**pop**();

//Aqui poner codigo para procesar el vertice (x)

**for**(**int** i=0; i<deg[x]; i++) {

indeg[adj[x][i]]--;

**if**(!indeg[adj[x][i]]) q.**push**(adj[x][i]);

} }

**// Flujos en Redes (Network Flow).**

**// Bipartite Matching.**

//Numero de nodos a la izquierda

#define M 50

//Numero de nodos a la derecha

#define N 50

//graph[i][j]=1,si hay una arista de i a j

**bool** graph[M][N];

**bool** seen[N];

//Contienen -1 si no hay matching

**int** matchL[M], matchR[N];

**int** n,m;

**bool** bpm( **int** u ) {

**for**(**int** v=0; v<n; v++)

**if**(graph[u][v]) {

**if**(seen[v]) **continue**;

seen[v] = **true**;

**if**(matchR[v]<0||bpm(matchR[v])) {

matchL[u] = v; matchR[v] = u;

**return true**;

} }

**return false**;

}

//Ejemplo de uso

**int** main() {

**while**(**cin**> >m> >n) {

memset(graph,0,sizeof(graph));

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

**for**(**int** j=0; j<n; j++)

**cin**>>graph[i][j];

memset( matchL, -1, sizeof( matchL ) );

memset( matchR, -1, sizeof( matchR ) );

**int** cnt = 0;

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

memset( seen, 0, sizeof( seen ) );

**if**( bpm( i ) ) cnt++;

}

**cout**<<cnt<<**endl**;

}

**return** 0;

}

**// MaxFlow (Edmond-Karps).**

#define MV 100

//Numero maximo de aristas sali**end**o de un vertice

#define INF 0x3f3f3f3f

**int** c[MV][MV], f[MV][MV], adj[MV][MV];

**int** deg[MV], parent[MV], seen[MV];

**int** bfs\_edmond(**int** s,**int** t) {

//inicializar busqueda

memset(seen,0,sizeof(seen));

parent[s]=-1; seen[s]=1;

//Hacer BFS

**queue**<int> q; q.**push**(s);

**int** x;

**bool** found=**false**;

**while**(!q.**empty**()&&!found) {

x=q.**front**(); q.**pop**();

**for**(**int** i=0; i<deg[x]&&!found; i++)

**if**(!seen[adj[x][i]]&&

(c[x][adj[x][i]]-f[x][adj[x][i]])>0) {

parent[adj[x][i]]=x; seen[adj[x][i]]=1;

**if**(adj[x][i]==t)

found=**true**;

q.**push**(adj[x][i]);

} }

**if**(!found) {

**return** 0;

}

//Obtener el maximo volumen que puede ser enviado

//a traves del camino encontrado

**int** res=INF; x=t;

**while**(parent[x]!=-1) {

res<?=(c[parent[x]][x]-f[parent[x]][x]);

x=parent[x];

}

**return** res;

}

**void** augment\_path(**int** s,**int** t,**int** v) {

**int** x=t;

**while**(x!=s) {

f[parent[x]][x]+=v;

f[x][parent[x]]-=v;

x=parent[x];

} }

Uso: llenar matriz de ady, establecer flujo a cero, y llenar capacidades (soporta grafos no dirigidos), y a continuación:

**int** v, tot=0;

**while**((v=bfs\_edmond(s,t))) {

tot+=v;

augment\_path(s,t,v);

}

en tot, queda el flujo máximo que se puede enviar de s a t

**// S-T Minimum Cut. Max Flow Min Cut Teorema**

// El valor de un flujo máximo es igual a la capacidad de un corte mínimo.

All pairs-Minimum Cut. Se puede encontrar el mínimo corte de un grafo fijando un vértice y corriendo n-1 flujos a partir de ese vértice a los demás, y tomando el mínimo de estos. Otra forma sin aplicar flujos, es mediante el algoritmo de Stoer-Wagner, el cual es el siguiente:

// Maximum number of vertices in the graph

#define NN 256

// Maximum edge weight

//(MAXW \* NN \* NN must fit into an int)

#define MAXW 1000

// Adjacency matrix and some internal arrays

**int** g[NN][NN], v[NN], w[NN], na[NN];

**bool** a[NN];

**int** minCut( **int** n ) {

// init the remaining vertex set

**for**( **int** i = 0; i < n; i++ ) v[i] = i;

// run Stoer-Wagner

**int** best = MAXW \* n \* n;

**while**( n > 1 ) {

// initialize the set A and vertex weights

a[v[0]] = **true**;

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

a[v[i]] = **false**;

na[i - 1] = i; w[i] = g[v[0]][v[i]];

}

// add the other vertices

**int** prev = v[0];

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

// find the most tightly connected non-A vertex

**int** zj = -1;

**for**( **int** j = 1; j < n; j++ )

**if**(!a[v[j]]&&(zj<0||w[j]>w[zj]))

zj = j;

// add it to A

a[v[zj]] = **true**;

// last vertex?

**if**( i == n - 1 ) {

// remember the cut weight

best <?= w[zj];

// merge prev and v[zj]

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

g[v[i]][prev] = g[prev][v[i]] += g[v[zj]][v[i]];

v[zj] = v[--n];

**break**;

}

prev = v[zj];

// update the weights of its neighbours

**for**(**int** j=1; j<n; j++) **if**(!a[v[j]])

w[j] += g[v[zj]][v[j]];

} }

**return** best;

}

Forma de uso: llenar matriz de adyacencia g y establecer n al número de vértices del grafo.

**// MinCost MaxFlow (Tambien MaxCost MaxFlow).**

using namespace std;

#define MV 250 //Numero de vertices de la red

**int** adj[MV][MV]; //Lista de adyacencia

**int** deg[MV]; //Grado de cada vertice

**int** f[MV][MV]; //flujos de las aristas

**int** cap[MV][MV]; //capacidad de las aristas

**double** cost[MV][MV]; //costos de las aristas

**double** d[MV]; //Vector distancia (Dijkstra)

**int** par[MV]; //Vector padre (Dijkstra)

**int** seen[MV]; //Vector seen(Djikstra)

**double** pi[MV]; //funcion de etiquetado para los nodos

//#define INF 1000000000000000000LL //(long long)

#define INFD 1e9 //(double)

#define INF 0x3f3f3f3f //(int)

**bool** djikstra(**int** s,**int** t,**int** n) {

**for**(**int** i=0; i<n; i++) d[i]=INFD;

memset(par,-1,sizeof(par));

memset(seen,0,sizeof(seen));

par[s]=s; d[s]=0;

**while**(1) {

**int** u=-1; **double** mmin=INFD;

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

**if**(!seen[i]&&d[i]<mmin) {

mmin=d[i]; u=i;

}

**if**(u==-1) **break**;

seen[u]=1;

**for**(**int** i=0; i<deg[u]; i++) {

**int** v=adj[u][i];

**if**(seen[v]) **continue**;

//checar si hay flujo de u a v

**if**(f[u][v]<cap[u][v]&&d[v]>d[u] +

(pi[u]+cost[u][v]-pi[v])) {

d[v]=d[u]+(pi[u]+cost[u][v]-pi[v]);

par[v]=u;

} } }

**for**(**int** i=0; i<n; i++) **if**(pi[i]<INFD) pi[i]+=d[i];

**return** par[t]>=0;

}

**void** init\_pi(**int** s,**int** n) //Bellman-Ford para maxcost-maxflow

{

**for**(**int** i=0; i<n; i++) pi[i]=INFD;

pi[s]=0;

**for**(**int** i=0; i<n-1; i++)

**for**(**int** j=0; j<n; j++)

**for**(**int** k=0; k<deg[j]; k++)

**if**((f[j][adj[j][k]]<cap[j][adj[j][k]]) &&

(pi[adj[j][k]]>pi[j]+cost[j][adj[j][k]]))

pi[adj[j][k]]=pi[j]+cost[j][adj[j][k]];

}

**int** mcmf(**int** s,**int** t,**int** n, **double** &fcost,**bool** mincost) {

memset(f,0,sizeof(f));

**if**(mincost) //Si es mincost entonces funcion de etiquetado=0

**for**(**int** i=0; i<n; i++) pi[i]=0;

**else** //Si es maxcost entonces inicializar con Bellman-Ford

init\_pi(s,n);

**int** flow=0;

fcost=0;

**while**(djikstra(s,t,n)) {

//Obtener el cuello de botella

**int** bot=INF, v=t, u;

**while**(v!=s) {

u=par[v];

bot<?=(cap[u][v]-f[u][v]);

v=u;

}

//Actualizar el flujo y el costo

v=t;

**while**(v!=s) {

u=par[v];

f[u][v]+=bot; f[v][u]-=bot;

fcost+=(bot\*cost[u][v]);

v=u;

}

flow+=bot;

}

**return** flow;

}

//Ejemplo de uso

**int** main() {

memset(deg,0,sizeof(deg)); //establecer el grado a 0

**int** source=0,sink=n;

//Leer el grafo y almacenar valores para capacidad y costo

adj[source][deg[source]++]=i;

cap[source][nodo]=cca;

cost[source][nodo]=20;

// Asi como tambien crear la arista que va al reves

adj[nodo][deg[nodo]++]=source;

cap[nodo][source]=0;

cost[nodo][source]=-20;

**bool** bmincost=**true**; //si se quiere maxcost, establecer false

**double** fcost; //Valor del costo

**int** flow=mcmf(source,sink,n+1,fcost,bmincost);

}

Si se quiere obtener el costo maximo, entonces antes de realizar el algoritmo se deben negar los costos.

El algoritmo anterior asume que si (u, v) pertenece a E, entonces (v, u) no pertenece a E, por lo que si se quieren representar grafos no dirigidos, se debe dividir cada vértice en dos nuevos vértices, el primero al que se conectaran todas las aristas que entran al vértice, al segundo se conectaran todas las aristas que salen del vértice, y además se unen estos dos nuevos vértices con una arista dirigida del primero al segundo, con costo de 0 y capacidad infinita.

**// Teoria de Grafos. Formula de Euler.**

V − E + F = 2

Numero de arboles diferentes etiquetados.

, s numero de componentes conexas.

**// Programacion Dinamica.**

**// LCS**

#define MATCH 1

#define L 2

#define U 3

#define M 500

**int** len[M][M],p[M][M];

// Obtener longitud de LCS

**int** lcs(**char** X[],**char** Y[]) {

**int** m=strlen(X); **int** n=strlen(Y);

**for** (**int** i=1; i<=m; i++) len[i][0]=0;

**for** (**int** j=0; j<=n; j++) len[0][j]=0;

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

**for** (**int** j=1; j<=n; j++) {

**if** (X[i-1]==Y[j-1]) {

len[i][j]=len[i-1][j-1]+1;

p[i][j]=MATCH; /\* match, incrementar \*/

}

**else** **if** (c[i-1][j]>=c[i][j-1]) {

len[i][j]=len[i-1][j];

p[i][j]=R; /\* de arriba \*/

}

**else** {

len[i][j]=len[i][j-1];

p[i][j]=L; /\* de la izquierda \*/

}

}

**return** len[m][n];

}

// Imprimir LCS

**void** print\_lcs(**int** m,**int** n) {

**if**(m==0||n==0) return;

**if**(p[m][n]==MATCH) {

print\_lcs(m-1,n-1);

**cout**<<arrm[m]<<' ';

}

**else** **if**(p[m][n]==L) print\_lcs(m,n-1);

**else** print\_lcs(m-1,n); }

**// Edit Distance.**

#define MATCH 0

#define SUBST 1

#define DELETE 2

#define INSERT 3

#define ML 85 //Maxima longitud de las cadenas

**int** parent[ML][ML], cost[ML][ML];

**char** s[85],t[85];

//Edit distance para transformar s a t.

**int** ed\_distance(void) {

**int** n=strlen(s), m=strlen(t);

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

cost[0][i]=i;

parent[0][i]=DELETE;

}

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

cost[i][0]=i;

parent[i][0]=INSERT;

}

parent[0][0]=-1;

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

**for**(**int** j=1; j<=n; j++) {

**if**(s[j-1]==t[i-1]) {

cost[i][j]=cost[i-1][j-1];

parent[i][j]=MATCH;

}

**else** {

cost[i][j]=cost[i-1][j-1]+1;

parent[i][j]=SUBST;

}

**if**(cost[i][j-1]+1<cost[i][j]) {

cost[i][j]=cost[i][j-1]+1;

parent[i][j]=DELETE;

}

**if**(cost[i-1][j]+1<cost[i][j]) {

cost[i][j]=cost[i-1][j]+1;

parent[i][j]=INSERT;

}

}

**return** cost[m][n];

}

//Inicializar nin=1,off=0,m=strlen(t),n=strlen(s)

**void** print\_edit(**int** m,**int** n) {

**if**(parent[m][n]!=-1) {

switch(parent[m][n]) {

case MATCH:

print\_edit(m-1,n-1);

**break**;

case SUBST:

print\_edit(m-1,n-1);

printf("%d Replace %d,%c\n", nin++,n+off,t[m-1]);

**break**;

case DELETE:

print\_edit(m,n-1);

printf("%d Delete %d\n",nin++, n+off);

off--;

**break**;

case INSERT:

print\_edit(m-1,n);

printf("%d Insert %d,%c\n",nin++, n+off+1,t[m-1]);

off++;

**break**;

}

}

}

**// Problema del cartero chino (Chinese Postman Problem).**

**int** best[1<<14], floyd[25][25];

**int** lodd[20], deg[25], ngen;

**int** solve(**int** x) {

**if**(best[x]==-1) {

best[x]=INF;

**for**(**int** i=0; i<ngen; i++)

**for**(**int** j=i+1; j<ngen; j++)

**if**((x> >i)%2&&(x> >j)%2)

best[x]<?=(floyd[lodd[i]][lodd[j]] +

solve(x-(1<<i)-(1<<j)));

}

**return** best[x];

}

**int** main() {

**int** n,e;

**while**(scanf("%d",&n)==1&&n) {

memset(best,-1,sizeof(best));

memset(deg,0,sizeof(deg));

best[0]=0;

scanf("%d",&e);

memset(floyd,0x3f,sizeof(floyd));

**int** res=0;

**for**(**int** i=0; i<e; i++) {

**int** a,b,c;

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

a--; b--; res+=c;

deg[a]++; deg[b]++;

floyd[a][b]<?=c; floyd[b][a]<?=c;

}

**for**(**int** k=0; k<n; k++)

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

**for**(**int** j=0; j<n; j++)

floyd[i][j]<?=(floyd[i][k]+floyd[k][j]);

**int** n2=0;

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

**if**(deg[i]%2) lodd[n2++]=i;

ngen=n2; printf("%d\n",res+solve((1<<n2)-1));

}

**return** 0;

}

**// Geometria Clasica.**

**// Punto de Interseccion (Seg-Seg, Seg-Linea, Linea-Linea).**

**bool** SegSegInt(point a,point b,point c,point d,point p) {

**double** s,t,num,denom;

denom=a[X]\*double(d[Y]-c[Y])+b[X]\*double(c[Y]-d[Y]) +

d[X]\*double(b[Y]-a[Y])+c[X]\*double(a[Y]-b[Y]);

//Paralelos

**if**(denom==0.0) **return false**;

num=a[X]\*double(d[Y]-c[Y])+c[X]\*double(a[Y]-d[Y]) +

d[X]\*double(c[Y]-a[Y]);

s=num/denom;

num=-(a[X]\*double(c[Y]-b[Y])+b[X]\*double(a[Y]-c[Y]) +

c[X]\*double(b[Y]-a[Y]));

t=num/denom;

p[X]=a[X]+s\*(b[X]-a[X]); p[Y]=a[Y]+s\*(b[Y]-a[Y]);

**return** (0.0<=s&&s<=1.0&&0.0<=t&&t<=1.0);

}

El codigo anterior funciona para la interseccion de dos segmentos (a,b) y (c,d), para interseccion de segmento

linea modificar la ultima condicion por:

**return** (0.0<=s&&s<=1.0);

Y para la interseccion linea-linea, simplemente regresar **true**.

**// Interseccion de Rectangulos.**

// left lower(xi,yi), right upper(xf,yf)

struct rect {

**int** xi,xf,yi,yf;

};

**bool** inter\_rect(rect &a,rect &b,rect &c) {

c.xi=max(a.xi,b.xi); c.xf=min(a.xf,b.xf);

c.yi=max(a.yi,b.yi); c.yf=min(a.yf,b.yf);

**if**(c.xi<=c.xf&&c.yi<=c.yf) **return true**;

**return false**;

}

// Calculo del area total de un conjunto de rectangulos.

**double** get\_Area\_Rect(**int** n) {

set<double> sx; set<double> sy;

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

sx.insert(R[i].xi); sx.insert(R[i].xf);

sy.insert(R[i].yi); sy.insert(R[i].yf);

}

**vector**<double> vx(sx.**begin**(),sx.**end**());

**vector**<double> vy(sy.**begin**(),sy.**end**());

**double** res=0.0;

**for**(**int** i=0; i<nx-1; i++) {

**for**(**int** j=0; j<ny-1; j++) {

**bool** inrect=**false**;

**for**(**int** k=0; k<n&&!inrect; k++)

**if**(R[k].xi<=vx[i]&&vx[i+1]<=R[k].xf&&

R[k].yi<=vy[j]&&vy[j+1]<=R[k].yf)

inrect=**true**;

**if**(inrect) res+=(vx[i+1]-vx[i])\*(vy[j+1]-vy[j]);

} }

**return** res;

}

**// Punto en Poligono.**

**bool** InPoly(point &q,polygon P,**int** n) {

**int** rcross,lcross,i1;

rcross=lcross=0;

**bool** rstrad,lstrad; **double** x;

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

//El punto es un vertice

**if**(P[i][X]==q[X]&&P[i][Y]==q[Y])

**return true**;

i1=(i-1+n)%n;

rstrad=(P[i][Y]>q[Y])!=(P[i1][Y]>q[Y]);

lstrad=(P[i][Y]<q[Y])!=(P[i1][Y]<q[Y]);

**if**(rstrad||lstrad) {

x=(q[Y]\*(P[i][X]-P[i1][X])–P[i1][Y]\*P[i][X]+

P[i1][X]\*P[i][Y])/(P[i][Y]-P[i1][Y]);

**if**(rstrad&&x>q[X]) rcross++;

**if**(rstrad&&x<q[X]) lcross++;

} }

// El punto esta en una arista

**if**((rcross%2)!=(lcross%2))

**return true**;

// Estrictamente interior

**if**((rcross%2)==1) **return true**;

**else** **return false**;

}

**// Poligonos Lattice y Teorema de Pick.**

// A(P) = I(P) + B(P)/2 - 1

// tot contiene el número de puntos en la frontera del poligono

**for**(**int** i = 0; i < verts.size(); i++) {

j = (i+1)%verts.size();

dx = abs(verts[j].first-verts[i].first);

dy = abs(verts[j].second-verts[i].second);

tot += gcd(dy,dx);

}

**// Mínimo rectángulo encapsulador.**

// Primero se tiene que calcular el ConvexHull.

// Y se usará los datos de la pila resultante.

**double** smallestBoundingRectangle (**int** n) {

// Se toma cada elemento de la pila

**double** mmin = 0;

**double** flag2 = **true**;

**double** inc = 1;

**double** a = 0, b = 90; // Topes para la primera iteración

**while** (inc>=1e-12) {

**double** area = 0;

**double** ta,tb;

**int** flag = **true**;

**for** (**double** teta=a; teta<=b; teta+=inc) {

**double** angle = (PI\*teta)/180.0;

point temp;

**double** x1,x2,y1,y2;

**bool** first=**true**;

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

temp[X] = pila[i]->v[X]\*cos(angle) –

pila[i]->v[Y]\*sin(angle);

temp[Y] = pila[i]->v[X]\*sin(angle) +

pila[i]->v[Y]\*cos(angle);

**if** (first) {

x1 = temp[X]; x2 = temp[X];

y1 = temp[Y]; y2 = temp[Y];

first = !first;

}

x1<?=temp[X]; x2>?=temp[X];

y1<?=temp[Y]; y2>?=temp[Y];

}

**if** (flag) {

area = (x2-x1) \* (y2-y1);

ta = teta - inc; tb = teta + inc;

flag = !flag;

}

**if** ((x2-x1) \* (y2-y1) < area) {

area = (x2-x1) \* (y2-y1);

ta = teta - inc; tb = teta + inc;

}

}

a = ta; b = tb;

**if** (flag2) {

mmin = area; flag2= !flag2;

}

mmin<?=area; inc/=10;

}

**return** mmin;

}

**// Distancia más cercana entre 2 polígonos.**

#define MAXP 105

#define X 0

#define Y 1

#define DIM 2

#define INF 1E18;

#define MAXV 30

typedef **double** Ti**pop**unto;

typedef Tipopunto point[DIM];

const **double** PI = 2\*acos(0);

struct poly {

**int** vnum;

point v;

**bool** del;

};

poly P[MAXV][MAXP];

**int** nP[MAXV];

**double** adj[MAXV][MAXV];

Tipopunto dist(point a,point b) {

Tipopunto dx=a[X]-b[X];

Tipopunto dy=a[Y]-b[Y];

**return** sqrt(dx\*dx+dy\*dy);

}

Tipopunto Dot(point a,point b) {

**return** a[X]\*b[X]+a[Y]\*b[Y];

}

**double** dist\_pnt\_to\_seg(point p,point a,point b,point pclose){

Tipopunto v[2]= {b[X]-a[X],b[Y]-a[Y]};

Tipopunto w[2]= {p[X]-a[X],p[Y]-a[Y]};

Tipopunto c1 = Dot(w,v);

**if** ( c1 <= 0 ) {

pclose[X]=a[X]; pclose[Y]=a[Y];

**return** dist(p,a);

}

**double** c2 = Dot(v,v);

**if** ( c2 <= c1 ) {

pclose[X]=b[X]; pclose[Y]=b[Y];

**return** dist(p,b);

}

**double** t = c1 / c2;

pclose[X]=a[X]+t\*v[X];

pclose[Y]=a[Y]+t\*v[Y];

**return** dist(p,pclose);

}

// i y j son los polígonos del arreglo de polígonos P

// el arreglo nP lleva el número de puntos

**double** minimumDistancePolygons(**int** a, **int** b) {

**double** minima = INF;

point temp;

**for** (**int** i=0; i<nP[a]; i++) {

**for** (**int** j=0; j<nP[b]; j++) {

// Condicion acorde al problema

**if** ((b==0||b==1) && j==nP[b]-1)

**break**;

minima<?=dist\_pnt\_to\_seg(P[a][i].v,P[b][j].v,

P[b][(j+1)%nP[b]].v,temp);

} }

**for** (**int** i=0; i<nP[b]; i++) {

**for** (**int** j=0; j<nP[a]; j++) {

// Condicion acorde al problema

**if** ((a==0||a==1) && j==nP[a]-1) **break**;

minima<?=dist\_pnt\_to\_seg(P[b][i].v,P[a][j].v,

P[a][(j+1)%nP[a]].v,temp);

} }

**return** minima;

}

**// Criba de Eratostenes.**

**// En un rango.**

**void** sieve(**int** L,**int** U) {

**int** i,j,d; d=U-L+1;

**bool** \*flag=**new** bool[d];

**for** (i=0; i<d; i++)

flag[i]=**true**;

**for** (i=(L%2!=0); i<d; i+=2)

flag[i]=**false**;

**for** (i=3; i<=sqrt(U); i+=2) {

**if** (i>L && !flag[i-L])

**continue**;

j=L/i\*i;

**if** (j<L) j+=i; **if** (j==i) j+=i;

j-=L;

**for** (; j<d; j+=i) flag[j]=**false**;

}

**if** (L<=1) flag[1-L]=**false**;

**if** (L<=2) flag[2-L]=**true**;

/\* output the result \*/

**for** (i=0; i<d; i++) **if**(flag[i])

**cout** << (L+i) << " ";

**cout** << **endl**;

}

**// Función Phi Euler (Numero de primos relativos a un número)**

**// De un solo numero.**

**int** phi\_euler(**int** x) {

map<int,int> m=fact\_primo(x);

map<int,int>::**iterator** it;

**int** res=x;

**for**(it=m.**begin**(); it!=m.**end**(); it++){

res/=(it->first); res\*=(it->first-1);

}

**return** res;

}

**// En un rango determinado.**

**int** euler[M]; **char** prime[M];

**int** phi\_euler2(**int** x) {

memset(prime,-1,sizeof(prime));

criba[0]=criba[1]=1;

**for**(**int** i=0; i<M; i++) euler[i]=i;

**for**(**int** i=0; i<M/2; i++)

**if**(prime[i])

**for**(**int** j=i+i; j<M; j+=i) {

prime[j]=0; euler[j]/=i;

euler[j]\*=(i-1);

} }

**// Combinaciones. C(n,k)**

**void** div\_by\_gcd(ll &a, ll &b) {

ll g = \_\_gcd(a,b);

a /= g; b /= g;

}

ll C(**int** n, **int** k) {

ll num = 1,den = 1,tomult,todiv;

**if**(k > n/2) k = n-k;

**for**(**int** i = k; i; i--) {

tomult = n-k+i; todiv = i;

div\_by\_gcd(tomult,todiv);

div\_by\_gcd(num,todiv); div\_by\_gcd(tomult,den);

num \*= tomult; den \* = todiv;

}

**return** num/den;

}

**// Triangulo de Pascal (DP)**

**void** pascal(**int** m) {

C[0][0]=1;

**for**(**int** i=1; i<=m; i++) { C[i][0]=C[i][i]=1;

**for**(**int** j=1; j<i; j++)

C[i][j]=C[i-1][j-1]+C[i-1][j];

} }

**// Modular Multiplication of big numbers**

**ll** mulmod(**ll** a, **ll** b, **ll** m) {

**ll** x = 0, y = a % m;

**while** (b > 0) {

**if** (b % 2 == 1) x = (x + y) % m;

y = (y \* 2) % m; b /= 2;

} **return** x;

}

**// Hashing una base**

ull h\_text[MAXN], pot[MAXN];

ull calc\_hash(**int** i, **int** j) {

**return** h\_text[f] - h\_text[i-1] \* pot[j-i];

}

**int** main() {

h\_text[0] = 0ULL;

**for**(**int** i = 1; j <= size\_text; i++)

h\_text[i] = h\_text[i-1] \* BASE + text[i];

pot[0] = 1;

**for**(**int** i = 1; i < MAXN; i++)

pot[i] = pot[i-1] \* BASE;

}

**// Hashing dos bases**

**int** ta, tb, pos;

**char** A[1000005], B[1000005], C[1000005];

ull pot[3][1000005], Dp[3][1000005], Hb[3];

ull calc\_hash( **int** ptr, **int** i, **int** f ) {

**return** Dp[ptr][f] - Dp[ptr][i-1]\*pot[ptr][f-i+1];

}

**int** main() {

scanf("%s%s", A + 1, B + 1);

ta = strlen( A + 1 ); tb = strlen( B + 1 );

**if**( ta < tb ) {

printf("%s", A+1); **return** 0;

}

pot[0][0] = 1, pot[1][0] = 1;

**for**( **int** i = 1; i <= ta; i ++ )

pot[0][i] = pot[0][i-1]\*33LL,

pot[1][i] = pot[1][i-1]\*41LL;

**for**( **int** i = 1; i <= tb; i ++ ) {

Hb[0] = Hb[0]\*33LL + ( B[i] - 'a' );

Hb[1] = Hb[1]\*41LL + ( B[i] - 'a' );

}

**for**( **int** i = 1; i <= ta; i ++ ) {

pos ++;

Dp[0][pos] = Dp[0][pos-1]\*33LL + (A[i]-'a');

Dp[1][pos] = Dp[1][pos-1]\*41LL + (A[i]-'a');

C[pos] = A[i];

**if**(pos>=tb && calc\_hash(0,pos-tb+1,pos)==Hb[0]

&& calc\_hash( 1, pos-tb+1,pos) == Hb[1] )

pos -= tb;

}

C[pos + 1] = '\0'; printf("%s", C + 1);

**return** 0;

}

**// FFT polynomial multiply**

**typedef** complex<**double**> base;

**void** fft (**vector**<base> & a, **bool** invert) {

**int** n = (**int**) a.size();

**for** (**int** i=1, j=0; i<n; ++i) {

**int** bit = n >> 1;

**for** (; j>=bit; bit>>=1) j -= bit;

j += bit;

**if** (i < j) **swap** (a[i], a[j]);

}

**for** (**int** len=2; len<=n; len<<=1) {

**double** ang = 2\*PI/len \* (invert ? -1 : 1);

base wlen (cos(ang), sin(ang));

**for** (**int** i=0; i<n; i+=len) { base w (1);

**for** (**int** j=0; j<len/2; ++j) {

base u = a[i+j], v = a[i+j+len/2] \* w;

a[i+j] = u + v;

a[i+j+len/2] = u - v;

w \*= wlen;

} } }

**if** (invert)

**for** (**int** i=0; i<n; ++i) a[i] /= n;

}

**void** multiply (**const vector**<**int**> & a, **const vector**<**int**> & b,

**vector**<**int**> & res) {

**vector**<base> fa(a.**begin**(),a.**end**()), fb(b.**begin**(),b.**end**());

size\_t n = 1;

**while** (n < max (a.size(), b.size())) n <<= 1;

n <<= 1;

fa.resize (n), fb.resize (n);

fft (fa, **false**), fft (fb, **false**);

**for** (size\_t i=0; i<n; ++i) fa[i] \*= fb[i];

fft (fa, **true**); res.resize (n);

**for** (size\_t i=0; i<n; ++i)

res[i] = **int** (fa[i].real() + 0.5);

}

**// Aritmética de precisión arbitraria implementada con cadenas**

**string** convertir(lln) {

**string** c("");

do {

c += (char)(n%10+’0’); n /= 10;

} **while**(n);

reverse(c.**begin**(),c.**end**());

**return** c;

}

**string** borrar\_ceros(**string** a) {

**int** i=0;

**while**(a[i]==’0’&&i<a.size()-1) i++;

**return** a.substr(i,a.size()-i);

}

**bool** menor(**string** a, **string** b) {

a=borrar\_ceros(a); b=borrar\_ceros(b);

**if**(a.size()<b.size()) **return true**;

**else** **if**(a.size()>b.size())

**return false**;

**else** **return** a<b;

}

**string** suma(**string** a, **string** b) {

**string** ans(""); **int** k=0;

**if**(a.size()<b.size()) swap(a,b);

**int** j = a.size()-1, i = b.size()-1;

**for**(; j>=0; j--,i--) {

**int** u = a[j]-’0’;

**if**(i>=0) {

ans += (u+(b[i]-’0’)+k)%10+’0’;

k = (u+(b[i]-’0’)+k)>=10;

}

// ans += (u-(b[i]-’0’)+k+10)%10+’0’;

// k = 0-((u-(b[i]-’0’)+k)<0);}

**else** {

ans += (u+k)%10+’0’;

k = (u+k)>=10;

} }

/\* ans += (u+k+10)%10+’0’;

k = 0-((u+k)<0); } } \*/

**if**(k) ans += ’1’;

reverse(ans.**begin**(),ans.**end**());

**return** ans;

}

**string** mult(**string** a, **string** b) {

**int** n = a.size(), m = b.size();

**int** t,k,i;

**string** ans(m+n,’0’);

**for**(**int** j = m; j>0; j--) {

**for**(i = n,k=0; i>0; i--) {

t = ((a[i-1]-’0’)\*(b[j-1]-’0’));

t += (ans[i+j-1]-’0’) + k;

ans[i+j-1] = (t%10)+’0’; k = t/10;

}

ans[j-1]=k+’0’;

}

**return** borrar\_ceros(ans);

}

**string** divide\_d(**string** a, **int** d) {

**string** temp("");

**int** N,i,res=0;

N = a.size();

temp+=((a[0]-’0’)/d)+’0’;

res = ((int)(a[0]-’0’))%d;

**for**(i=1; i<N; i++) {

res =(res\*10)+(a[i]-’0’);

temp +=(res/d+’0’); res = res%d;

}

**return** borrar\_ceros(temp);

}

**string** divide(**string** u, **string** v) {

**string** d(""),ans(""),parcial("");

**string** temp1(""),temp2("");

**vector** <string> mul;

mul.clear();

**if**(v.size()==1)

**return** divide\_d(u,v[0]-’0’);

**int** m,q=0,a1,a2,a3,a4,a5,j,inc;

d += (10/((v[0]-’0’)+1)+’0’);

u = mult(d,u); v = mult(d,v);

u.insert(u.**begin**(),’0’); j = 0;

mul.**push**\_back("0"); mul.**push**\_back(v);

**for**(**int** k=2; k<10; k++)

mul.**push**\_back(suma(mul[k-1],v));

m = u.size()-v.size()-1;

**while**(j<=m) {

a1 = u[j]-’0’; a2 = u[j+1]-’0’; a3 = u[j+2]-’0’;

a4 = v[0]-’0’; a5 = v[1]-’0’;

**if**(a1==a4) q = 9;

**else** q = (a1\*10+a2)/a4;

**while**(q\*a5>((a1\*10+a2-q\*a4)\*10+a3)) q--;

parcial.erase();

**for**(**int** l=j; l<j+v.size()+1; l++)

parcial += u[l];

**if**(menor(parcial,mul[q])) q--;

temp2 = resta(parcial,mul[q]);

**for**(**int** l=j; l<j+v.size()+1; l++)

u[l] = temp2[l-j];

ans += (char)(q+’0’); j++;

}

**return** borrar\_ceros(ans);

}

**// Misceláneas**

**// Extracción de datos listados en una sola fila cuando no se especifica su número.**

**cin**.getline(conjuntos, 1000);

ptr = strtok(conjuntos," ");

**while**(ptr!=NULL) {

numero=atoi(ptr);

B.insert(numero);

ptr = strtok(NULL, " ");

}

**// Probar si un año es bisiesto**

**bool** leap(**int** y) {

**return** y % 4 == 0 && (y % 100 != 0 || y % 400 == 0);

}

**// Primeras cifras de n a la k.**

// para obtener las 3 más significativas

x = k \* log10(n)

signif = pow(10.0, x - floor(x)) \* 100

**// Inverso modular de inv(mod M)**

ll inverso(ll inv, ll M){  
for(ll i = 1; i <= 1e9+7; i++)  
 if( (i\*inv)%M == 1 ) return i;  
}

**// Cantidad números fibonacci hasta n**

floor((log10(n)+ (log10(5)/2))/log10(1.6180));

numero áureo = (1+sqrt(5))/2 = 1.6180339887498948482

**// TRABAJO CON BITS**

Set\_union Set\_intersection Set\_subtraction Set\_negation

A | B A & B A & ~B ALL\_BITS ^ A

Set\_bit Clear\_bit Test\_bit

A |= 1 << bit A &= ~(1 << bit) (A & 1 << bit) != 0

**// Longitud de los números de 1 a N**

LL sumDig(LL n, LL m) { // resultado modulo m

LL b=10, d=1, r=0;

**while**(b<=n){

r = (r + (b-b/10LL)\*(d++)) %m; b\*=10LL;

}

**return** (r + (n-b/10LL+1LL)\*d) %m;

}

**// Ternas pitagóricas**

Las soluciones primitivas positivas de con y par son , , donde r y s son enteros arbitrarios de paridad opuesta con y .

**// Lectura en Java**

class test {

public static void main (String [] args) throws IOException {

// Use BufferedReader rather than RandomAccessFile

BufferedReader f = new BufferedReader(

new FileReader("test.in"));

// input file name goes above

PrintWriter out = new PrintWriter(new BufferedWriter(

new FileWriter("test.out")));

// Use StringTokenizer vs. readLine/split - lots faster

StringTokenizer st = new StringTokenizer(f.readLine());

// Get line, break into tokens

int i1 = Integer.parseInt(st.nextToken());

int i2 = Integer.parseInt(st.nextToken());

out.println(i1+i2); out.close();

System.exit(0); // don't omit this!

}

}

// Para leer de la entrada estandar usar

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

Scanner cin = new Scanner(System.in);

BigInteger a = cin.nextBigInteger();

int b = cin.nextInt(); cin.close();