**Convex Hull**

point **resta**(point p1, point p2) {

**return** (point ) { p1.x - p2.x, p1.y - p2.y } ;

}

**double** **crossProd**(point p1, point p2) {

**return** p1.x \* p2.y - p1.y \* p2.x;

}

**int** N;

point vertex[5000];

**void** **convexHull**(point ps[]) {

**int** n = N, k = 0;

**for** (**int** i = 0; i < n; vertex[k++] = ps[i++])

**while** (k >= 2 && crossProd(resta(vertex[k - 1], vertex[k - 2]), resta(ps[i], vertex[k - 2])) <= 0)

--k;

**for** (**int** i = n - 2, t = k + 1; i >= 0; vertex[k++] = ps[i--])

**while** (k >= t && crossProd(resta(vertex[k - 1], vertex[k - 2]), resta(ps[i], vertex[k - 2])) <= 0)

--k;

}

**LCA**

//L es el nivel del nodo, T es el padre del nodo, P es donde se va a computar el SPT

int L[MAXN], T[MAXN], P[MAXN][LOGMAXN];

void SPT(int N) {

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

for (int j = 0; 1 << j < N; ++j)

P[i][j] = -1;

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

P[i][0] = T[i];

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

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

if (P[i][j - 1] != -1)

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

}

int lca(int p, int q) {

int log, lc;

if (L[p] < L[q])

swap(p, q);

for (log = 1; 1 << log <= L[p]; log++);

log--;

for (int i = log; i >= 0; i--)

if (L[p] - (1 << i) >= L[q])

p = P[p][i];

if (p == q)

lc = p;

else {

for (int i = log; i >= 0; i--)

if (P[p][i] != -1 && P[p][i] != P[q][i]) {

p = P[p][i];

q = P[q][i];

}

lc = T[p];

}

return lc;

}

**SPT LMQ**

int M[MAXN][LOGMAXN];

void createSPT(int N) {

int i, j;

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

M[i][0] = i;

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

for (i = 0; i + (1 << j) - 1 < N; i++)

if (notes[M[i][j - 1]] < notes[M[i + (1 << (j - 1))][j - 1]])

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

else

M[i][j] = M[i + (1 << (j - 1))][j - 1];

}

int get\_minor(int i, int j) {

int k = log2(j - i + 1);

return (notes[M[i][k]] <= notes[M[j - (1 << k) + 1][k]]) ?

notes[M[i][k]] : notes[M[j - (1 << k) + 1][k]];

}

**Articulation vertex**

#define MAXV 50005

#define MAXE 100005

struct node {

int v, next;

} L[MAXE];

int V, E, cnt, top, subtrees;int x, y;

int ptr[MAXV];int dfsnum[MAXV], low[MAXV];

bool mk[MAXV], ap[MAXV];

void DFS( int u ) {

mk[u] = true;

dfsnum[u] = low[u] = ++cnt;

for ( int i = ptr[u]; i >= 0; i = L[i].next ) {

int v = L[i].v;

if ( !mk[v] ) {

if ( !u ) subtrees++;

DFS( v );

if ( low[v] >= dfsnum[u] ) ap[u] = true;

low[u] <?= low[v];

} else low[u] <?= dfsnum[v]; }}

int main() {

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

scanf( "%d %d", &V, &E );

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

scanf( "%d %d", &x, &y );

x--; y--;

L[i] = ( node ) { y, ptr[x] };

ptr[x] = i;

L[ i + E ] = ( node ) { x, ptr[y] };

ptr[y] = i + E;

}

// Find Articulation Points

DFS( 0 );

if ( subtrees < 2 ) ap[0] = false;

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

if ( ap[i] ) printf( "%d\n", i + 1 );}

**Bridge**

#define MAXV 10000

#define MAXE 30000

#define BACK( x ) ( ( x < E ) ? x + E : x - E )

struct edge {

int v, next;

} edges[ 2 \* MAXE ];

int V, E;int u, v;int \_time;

int p[MAXV];int dfsn[MAXV], low[MAXV];

bool cut[ 2 \* MAXE ];vector< pair< int, int > > bridges;

void dfs( int u ) {

dfsn[u] = low[u] = ++\_time;

for ( int i = p[u]; i != -1; i = edges[i].next ) {

int v = edges[i].v;

if ( !dfsn[v] ) {

cut[ BACK( i ) ] = 1;

dfs( v );

low[u] <?= low[v];

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

bridges.push\_back( make\_pair( u, v ) );

} else if ( !cut[i] ) low[u] <?= dfsn[v]; }}

int main() {

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

scanf( "%d %d", &V, &E );

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

scanf( "%d %d", &u, &v );

u--; v--;

edges[i] = ( edge ) { v, p[u] };

p[u] = i;

edges[ i + E ] = ( edge ) { u, p[v] };

p[v] = i + E; }

dfs( 0 );

int size = bridges.size();

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

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

printf( "(%d,%d)\n", bridges[i].first + 1, bridges[i].second + 1 );}

**Chequear si un grafo es bipartito**

#define MAXN 10001

vector<int> g[MAXN];

int part[MAXN];

bool possible, multiples;int p;

void check(int nod) {

if (!possible)

return;

if (part[nod] == -1) {

if (p > 1)

multiples = true;

part[nod] = p;

p += 2;}

for (int i = 0; i < (int) g[nod].size(); ++i) {

int v = g[nod][i];

if(v == nod)

continue;

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

part[v] = (part[nod] % 2) ? part[nod] - 1 : part[nod] + 1;

check(v);

} else if (part[v] == part[nod]) {

possible = false;

break;}}}

**Floyd warshall**

void Floyd(int V, int[][] c) {

int w;

for (int k = 0; k < V; k++)

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

if (c[i][k] < oo)

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

if (c[k][j] < oo) {

w = c[i][k] + c[k][j];

if (w < c[i][j]) {

c[i][j] = w; }}}}

**Disjoint set**

**int** p[2000005], cnt[2000005];

**inline** **void** **initialize**(**int** n) {

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

p[i] = i;

cnt[i] = 1;

}

}

**inline** **int** **find\_set**(**int** x) {

**return** p[x] = p[x] != x ? find\_set(p[x]) : x;

}

**inline** **void** **merge**(**int** set1, **int** set2) {

p[set2] = set1;

cnt[set1] += cnt[set2];

}

**MST Kruskal**

// Se hace el disjoint set primero

**struct** edge {

**int** u, v, w;

**bool** **operator<**(**const** edge& e) **const** {

**return** w < e.w;

}

};

vector<edge> e;

sort(e.begin(), e.end());

**int** cnt1 = 0, max\_v = 0, mst = 0;

**for** (size\_t i = 0; i < e.size() && cnt1 < n - 1; ++i) {

**int** u = e[i].u, v = e[i].v, w = e[i].w;

**int** set1 = find\_set(u);

**int** set2 = find\_set(v);

**if** (set1 != set2) {

merge(set1, set2);

cnt1++;

max\_v = max(max\_v, w); // lo que se quiere

mst += w;

}

}

**Segment Tree**

**int** array[n];

**int** tree[4 \* n], prop[4 \* n];

**void** **initialize**(**int** nod, **int** lo, **int** hi) {

//llegada a una hoja del tree

**if** (lo == hi) {

tree[nod] = array[lo];

**return**;

}

**int** mid = (hi + lo) / 2;

initialize(2 \* nod, lo, mid);

initialize(2 \* nod + 1, mid + 1, hi);

tree[nod] = tree[2 \* nod] + tree[2 \* nod + 1];

}

**void** **update**(**int** nod, **int** lo, **int** hi, **int** slo, **int** shi, **int** v) {

**if** (slo > hi || lo > shi)

**return**;

**int** tmpR = max(lo, slo), tmpL = min(hi, shi);

tree[nod] += v \* (tmpL - tmpR + 1);

**int** mid = (hi + lo) / 2, r = 2 \* nod, l = r + 1;

**if** (prop[nod]) {

prop[r] += prop[nod];

prop[l] += prop[nod];

tree[r] += prop[nod] \* (mid - lo + 1);

tree[l] += prop[nod] \* (hi - (mid + 1) + 1);

prop[nod] = 0;

}

**if** (slo <= lo && hi <= shi) {

prop[nod] += v;

**return**;

}

update(r, lo, mid, slo, shi, v);

update(l, mid + 1, hi, slo, shi, v);

}

**int** **query**(**int** nod, **int** lo, **int** hi, **int** slo, **int** shi) {

**if** (slo > hi || lo > shi)

**return** 0;

**int** mid = (hi + lo) / 2, r = 2 \* nod, l = r + 1;

**if** (prop[nod]) {

prop[r] += prop[nod];

prop[l] += prop[nod];

tree[r] += prop[nod] \* (mid - lo + 1);

tree[l] += prop[nod] \* (hi - (mid + 1) + 1);

prop[nod] = 0;

}

**if** (slo <= lo && hi <= shi) {

**return** tree[nod];

}

**int** rr = query(r, lo, mid, slo, shi);

**int** ll = query(l, mid + 1, hi, slo, shi);

**return** rr + ll;

}

**Triangulo de Pascal**

//combinaciones de n en k C(n,k)

**void** **crearTrianguloPascal**(**int** n) {

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

dp[i][0] = 1;

}

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

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

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

dp[i][j] %= MOD;

}

}

}

**Calcular las raices de un polinomio de cualquier grado**

ll divisor[1000000];

ll ndiv;

//N es el termino independiente

void find\_div(ll N) {

ll i, p, old;

if (N < 0)

N = -N;

ndiv = 0;

divisor[ndiv++] = 1;

for (p = 2; p <= 46337 && p \* p <= N; p++) {

old = ndiv;

while (N % p == 0) {

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

ll tmp = divisor[ndiv - old] \* p;

divisor[ndiv] = tmp;

ndiv++;

}

N /= p;

}

}

if (N > 1) {

old = ndiv;

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

divisor[ndiv] = divisor[ndiv - old] \* N;

ndiv++;

}

}

//sort(divisor, divisor + ndiv);

}

**Rotar un punto counterclockwise**

inline double to\_radian(double ang) {

return PI \* ang / 180;

}

inline void rotate(double xx, double yy, double& nx, double& ny, double ang) {

nx = xx \* cos(to\_radian(ang)) - yy \* sin(to\_radian(ang));

ny = xx \* sin(to\_radian(ang)) + yy \* cos(to\_radian(ang));

}

**Joseph (eliminación en círculo)**

//para un k mayor O(n)

int joseph(int n, int k) {

int res = 0;

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

res = (res + k) % i;

return res + 1;

}

//para un k pequeño k log n

ll joseph1(ll n, ll k) {

if (n == 1)

return 0;

if (k == 1)

return n - 1;

if (k > n)

return (joseph1(n - 1, k) + k) % n;

ll cnt = n / k;

ll res = joseph1(n - cnt, k);

res -= n % k;

if (res < 0)

res += n;

else

res += res / (k - 1);

return res;

}

**Complex**

**inline** P **operator\***(**const** P b)

{**return** P(x\*b.x-y\*b.y , x\*b.y + y\*b.x );}

**inline** P **operator/**(**const** P b){

P s = P( x\*b.x + y\*b.y , y\*b.x-x\*b.y);

s.x /= ( b.x\*b.x + b.y\*b.y);

s.y /= ( b.x\*b.x + b.y\*b.y);

**return** s;

}