Contents

[KMP 1](#__RefHeading___Toc946_642709658)

[Interval Tree 1](#__RefHeading___Toc948_642709658)

[Interval Tree (lazy) 2](#__RefHeading___Toc950_642709658)

[Tree of primitive Pythagorean triples 3](#__RefHeading___Toc952_642709658)

[Miller-Rabin Primality Test 3](#__RefHeading___Toc954_642709658)

[Hopcroft-Karp - Max matching for bipartite graph - O(E\*sqrt(V)) 4](#__RefHeading___Toc956_642709658)

[Geometry convex hull - O(n logn) 6](#__RefHeading___Toc958_642709658)

[Union-Find Disjoint Sets 6](#__RefHeading___Toc960_642709658)

[Z Function - O(n) 7](#__RefHeading___Toc962_642709658)

[EGG - VNSPOJ 7](#__RefHeading___Toc964_642709658)

[Fermat’s Little Theorem 8](#__RefHeading___Toc966_642709658)

[LCA - <O(nlogn), O(logn)> 8](#__RefHeading___Toc968_642709658)

[Extended Euclidean algorithm 8](#__RefHeading___Toc970_642709658)

[Phi-Euler function 9](#__RefHeading___Toc972_642709658)

# KMP

char T[MAX\_N], P[MAX\_N]; // T = text, P = pattern

int b[MAX\_N], n, m; // b = back table, n = length of T, m = length of P

void kmpPreprocess() {

n = strlen(T);

m = strlen(P);

int i = 0, j = -1; b[0] = -1;

while (i < m) {

while (j >= 0 && P[i] != P[j]) j = b[j];

i++; j++;

b[i] = j;

} }

void kmpSearch() {

int i = 0, j = 0;

while (i < n) {

while (j >= 0 && T[i] != P[j]) j = b[j];

i++; j++;

if (j == m) {

printf("P is found at index %d in T\n", i - j);

j = b[j];

} } }

# Interval Tree

int n, a[100001], t[400001];

void ITinit(int k, int l, int r) {

if (l == r) t[k] = l;

else {

ITinit(k<<1, l, (l+r)>>1);

ITinit((k<<1)+1, ((l+r)>>1)+1, r);

int p1 = t[k<<1], p2 = t[(k<<1)+1];

t[k] = (a[p1] <= a[p2]) ? p1 : p2;

} }

int ITupdate(int k, int l, int r, int i, int v) { // a[i] <- v

if (l > i || r < i) return t[k];

if (l == r) { a[i] = v; return t[k] = l; }

int p1 = ITupdate(k<<1, l, (l+r)>>1, i, v);

int p2 = ITupdate((k<<1)+1, ((l+r)>>1)+1, r, i, v);

return t[k] = (a[p1] <= a[p2]) ? p1 : p2;

}

int ITrmq(int k, int l, int r, int i, int j) { // return index

if (l > j || r < i) return -1;

if (l >= i && r <= j) return t[k];

int p1 = ITrmq(k<<1, l, (l+r)>>1, i, j);

int p2 = ITrmq((k<<1)+1, ((l+r)>>1)+1, r, i, j);

if (p1 == -1) return p2;

if (p2 == -1) return p1;

return (a[p1] <= a[p2]) ? p1 : p2;

}

# Interval Tree (lazy)

int arr[N];

int tree[MAX];

int lazy[MAX];

void build\_tree(int node, int a, int b) { // build\_tree(1, 0, n-1)

if(a > b) return;

if(a == b) {

tree[node] = arr[a];

return;

}

build\_tree(node << 1, a, (a+b)/2);

build\_tree((node << 1) + 1, 1 + ((a+b) >> 1), b);

tree[node] = max(tree[node << 1], tree[(node << 1) + 1]);

}

void update\_tree(int node, int a, int b, int i, int j, int value) {

if (lazy[node]) {

tree[node] += lazy[node];

if(a != b) {

lazy[node << 1] += lazy[node];

lazy[(node << 1) + 1] += lazy[node];

}

lazy[node] = 0;

}

if(a > b || a > j || b < i)

return;

if(a >= i && b <= j) {

tree[node] += value;

if(a != b) {

lazy[node << 1] += value;

lazy[(node << 1) + 1] += value;

}

return;

}

update\_tree(node << 1, a, (a+b)/2, i, j, value);

update\_tree((node << 1) + 1, ((a+b) >> 1) + 1, b, i, j, value);

tree[node] = max(tree[node << 1], tree[(node << 1) + 1]);

}

int query\_tree(int node, int a, int b, int i, int j) {

if(a > b || a > j || b < i) return -inf;

if(lazy[node]) {

tree[node] += lazy[node];

if(a != b) {

lazy[node << 1] += lazy[node];

lazy[(node << 1) + 1] += lazy[node];

}

lazy[node] = 0;

}

if(a >= i && b <= j)

return tree[node];

int q1 = query\_tree(node << 1, a, (a+b) >> 1, i, j);

int q2 = query\_tree((node << 1) + 1, ((a+b) >> 1) + 1, b, i, j);

int res = max(q1, q2);

return res;

}

# Tree of primitive Pythagorean triples


\begin{array}{lcr}
A = \begin{bmatrix} 1 & -2 & 2 \\ 2 & -1 & 2 \\ 2 & -2 & 3 \end{bmatrix} &
B = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 3 \end{bmatrix} &
C = \begin{bmatrix} -1 & 2 & 2 \\ -2 & 1 & 2 \\ -2 & 2 & 3 \end{bmatrix}
\end{array}


# Miller-Rabin Primality Test

typedef long long ll;

int prime[] = {2,3,5,7,11,13,17,19,23,29,31,37};

ll multiply(ll a, ll b, ll mod) // O(1) for (a\*b)%m

{

a %= mod; b %= mod;

long double res = a;

res \*= b;

ll c = (ll)(res / mod);

a = (a\*b - c\*mod) % mod;

if (a < 0) a += mod;

return a;

}

ll power(ll a, ll b, ll mod)

{

ll ans = 1;

while(b)

{

if (b&1) ans = multiply(ans, a, mod);

a = multiply(a, a, mod);

b >>= 1;

}

return ans;

}

bool Miller(ll p)

{

if (p < 2) return false;

if (p != 2 && !(p&1)) return false;

int cnt = 0;

ll s = p - 1;

while (!(s&1))

{

s >>= 1;

++cnt;

}

int sz = sizeof(prime) / sizeof(int);

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

{

ll a = prime[i];

if (a >= p) break;

ll x = power(a, s, p);

if (x == 1 || x == p-1) continue;

int flag = 0;

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

{

x = multiply(x, x, p);

if(x == 1) return false;

if(x == p-1)

{

flag = 1;

break;

}

}

if (!flag) return false;

}

return true;

}

# Hopcroft-Karp - Max matching for bipartite graph - O(E\*sqrt(V))

const int MAXN = 50000;

const int MAXM = 150000;

int n1, n2, m, pairU[MAXN+5], pairV[MAXN+5], dist[MAXN+5];

vector<int> adj[MAXN+5];

bool bfs() {

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

queue<int> q;

for (int u = 1; u <= n1; ++u)

if (pairU[u] == -1) q.push(u), dist[u] = 0;

int mx = -1;

while (!q.empty()) {

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

if (mx != -1 && dist[u] >= mx) continue;

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

int v = adj[u][i];

if (pairV[v] == -1) mx = dist[u];

else if (dist[pairV[v]] == -1)

dist[pairV[v]] = dist[u] + 1, q.push(pairV[v]);

}

}

return (mx != -1);

}

bool dfs(int u) {

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

int v = adj[u][i];

if (pairV[v] == -1 || (dist[pairV[v]] == dist[u] + 1 && dfs(pairV[v]))) {

pairV[v] = u;

pairU[u] = v;

return true;

}

}

dist[u] = -1;

return false;

}

int maxMatching() {

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

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

int res = 0;

while (bfs()) {

for (int u = 1; u <= n1; ++u)

if (pairU[u] == -1 && dfs(u)) ++res;

}

return res;

}

int main() {

scanf("%d %d %d", &n1, &n2, &m);

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

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

adj[u].push\_back(v);

}

printf("%d", maxMatching());

return 0;

}

# Geometry convex hull - O(n logn)

typedef pair<double, double> point;  
  
bool cw(const point &a, const point &b, const point &c) {  
 return (b.first - a.first) \* (c.second - a.second) - (b.second - a.second) \* (c.first - a.first) < 0;  
}  
  
vector<point> convexHull(vector<point> p) {  
 int n = p.size();  
 if (n <= 1)  
 return p;  
 int k = 0;  
 sort(p.begin(), p.end());  
 vector<point> q(n \* 2);  
 for (int i = 0; i < n; q[k++] = p[i++])  
 for (; k >= 2 && !cw(q[k - 2], q[k - 1], p[i]); --k);

for (int i = n - 2, t = k; i >= 0; q[k++] = p[i--])  
 for (; k > t && !cw(q[k - 2], q[k - 1], p[i]); --k);

q.resize(k - 1 - (q[0] == q[1]));  
 return q;  
}  
  
int main() {  
 vector<point> points(4);  
 points[0] = point(0, 0);  
 points[1] = point(3, 0);  
 points[2] = point(0, 3);  
 points[3] = point(1, 1);  
 vector<point> hull = convexHull(points);  
 cout << (3 == hull.size()) << endl;  
}

# Union-Find Disjoint Sets

class UnionFind {

private:

vi p, rank, setSize;

int numSets;

public:

UnionFind(int N) {

setSize.assign(N, 1); numSets = N; rank.assign(N, 0);

p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }

int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }

bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }

void unionSet(int i, int j) {

if (!isSameSet(i, j)) { numSets--;

int x = findSet(i), y = findSet(j);

if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }

else { p[x] = y; setSize[y] += setSize[x];

if (rank[x] == rank[y]) rank[y]++; } } }

int numDisjointSets() { return numSets; }

int sizeOfSet(int i) { return setSize[findSet(i)]; }

};

# Z Function - O(n)

int L = 0, R = 0;

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

if (i > R) {

L = R = i;

while (R < n && s[R-L] == s[R]) ++R;

z[i] = R-L; R--;

}

else {

int k = i-L;

if (z[k] < R-i+1) z[i] = z[k];

else {

L = i;

while (R < n && s[R-L] == s[R]) ++R;

z[i] = R-L; --R;

}

}

}

# EGG - VNSPOJ

const int N = 5009, z = 5000;

int t, n, m, f[N][N], llog[N];

int Log(int x) {

int cnt = 0;

while (x) cnt++, x /= 2;

return cnt;

}

void init() {

for (int i=1;i<=z;i++) llog[i] = Log(i);

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

for (int j=1;j<=z;j++) f[i][j] = z;

for (int i=1;i<=z;i++) f[1][i] = i;

for (int i=2;i<=z;i++) {

for (int j=1;j<=z;j++) {

if (i > llog[j]) f[i][j] = f[i-1][j];

else

for (int k=2;k<=j;k++)

f[i][j] = min(f[i][j], max(f[i-1][k-1], f[i][j-k])+1);

}

}

}

int main() {

init();

scanf("%d", &t);

while (t--) {

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

printf("%d\n", f[n][m]);

}

return 0;

}

# Fermat’s Little Theorem

# LCA - <O(nlogn), O(logn)>

int par[MAXN + 1][LOGMAXN + 1]; // initially all -1

void dfs(int v, int p = -1) {

par[v][0] = p;

if (p != -1)

h[v] = h[p] + 1;

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

if (par[v][i-1] != -1)

par[v][i] = par[par[v][i-1]][i-1];

for (int i = 0; i < adj[v].size(); ++i)

if (adj[v][i] != p)

dfs(adj[v][i], v);

}

int LCA(int v, int u) {

if (h[v] < h[u])

swap(u, v);

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

if (par[v][i] != -1 && h[par[v][i]] >= h[u])

v = par[v][i];

// now h[v] == h[u]

if (v == u)

return v;

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

if (par[v][i] != par[u][i])

v = par[v][i], u = par[u][i];

return par[v][0];

}

# Extended Euclidean algorithm

ll ExtendedGcd(ll a, ll b, ll &x, ll &y) {

if (a % b == 0) {

x = 0;

y = 1;

return b;

}

ll newx, newy;

ll ret = ExtendedGcd(b, a % b, newx, newy);

x = newy;

y = newx - newy \* (a / b);

return ret;

}

Trả về một nghiệm của pt với , , . Khi đó nghiệm tổng quát của pt là , .

# Phi-Euler function

ll f(ll n) {

if (n == 1)

return 1;

ll res = n;

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

if (n % i == 0) {

res = res / i \* (i - 1);

while (n % i == 0) {

n /= i;

}

}

if (n > 1) {

res = res / n \* (n - 1);

}

return res;

}