```
  ll get(int in, int l, int r, int s, int e){
    pushdown(in, l, r);
    if(l > e || r < s)return 0;
    if(l >= s && r <= e){
      return tree[in];
    }
    int mid = (l + r) / 2;
    ll le, ri;
    le = get(2 * in, l, mid, s, e);
    ri = get(2 * in + 1, mid + 1, r, s, e);
    return le + ri;
  }
  void add(int s, int e, ll v){
    add(1, 0, n - 1, s, e, v);
  }
  void add(int in, int l, int r, int s, int e, ll v){
    pushdown(in, l, r);
    if(l > e || r < s)return;
    if(l >= s && r <= e){
      lazy[in] += v;
      pushdown(in, l, r);
      return;
    }
    int mid = (l + r) / 2;
    add(2 * in, l, mid, s, e, v);
    add(2 * in + 1, mid + 1, r, s, e, v);
    tree[in] = tree[2 * in] + tree[2 * in + 1];
  }
};
```

```
// segment tree with lazy propagation

struct Tr{
  vector<ll> tree;
  vector<ll> arr;
  vector<ll> lazy;
  int n;
  Tr(vector<ll> _arr){
    arr = _arr;
    n = arr.size();
    lazy.resize(4 * n);
    tree.resize(4 * n);
    build(1, 0, n - 1);
  }

  void build(int in, int l, int r){
    if(l == r){
      tree[in] = arr[l];
      return;
    }
    int mid = (l + r) / 2;
    build(2 * in, l, mid);
    build(2 * in + 1, mid + 1, r);
    tree[in] = tree[in * 2] + tree[in * 2 + 1];
  }
  void pushdown(int in, int l, int r){
    tree[in] += lazy[in] * (r - l + 1);
    if(l != r){
      lazy[2 * in] += lazy[in];
      lazy[2 * in + 1] += lazy[in];
    }
    lazy[in] = 0;
  }
  void update(int in, ll v){
    update(1, 0, n - 1, in, v);
  }
  void update(int in, int l, int r, int i, ll v){
    pushdown(in, l, r);
    if(l > i ||  r < i)return;
    if(l == r){
      tree[in] += v;
      return;
    }
    int mid = (l + r) / 2;
    update(2 * in, l, mid, i, v);
    update(2 * in + 1, mid + 1, r, i, v);
    tree[in] = tree[in * 2] + tree[in * 2 + 1];
  }
  ll get(int s, int e){
    return get(1, 0, n - 1, s, e);
  }
```

```
// segment tree
struct Tr{
  vector<ll> tree;
  vector<ll> arr;
  int n;
  Tr(vector<ll> _arr){
    arr = _arr;
    n = arr.size();
    tree.resize(4 * n);
    build(1, 0, n - 1);
  }
  void build(int in, int l, int r){
    if(l == r){
      tree[in] = arr[l];
      return;
    }
    int mid = (l + r) / 2;
    build(2 * in, l, mid);
    build(2 * in + 1, mid + 1, r);
    tree[in] = max(tree[in * 2], tree[in * 2 + 1]);
  }
  void update(int in, int v){
    update(1, 0, n - 1, in, v);
  }
  void update(int in, int l, int r, int i, int v){
    if(l > i || r < i)return;
    if(l == r){
      tree[in] = v;
      return;
    }
    int mid = (l + r) / 2;
    update(2 * in, l, mid, i, v);
    update(2 * in + 1, mid + 1, r, i, v);
    tree[in] = max(tree[in * 2], tree[in * 2 + 1]);
  }
  ll get(int s, int e){
    return get(1, 0, n - 1, s, e);
  }
  ll get(int in, int l, int r, int s, int e){
    if(l > e || r < s)return -1e9;
    if(l >= s && r <= e){
      return tree[in];
    }
    int mid = (l + r) / 2;
    ll le, ri;
    le = get(2 * in, l, mid, s, e);
    ri = get(2 * in + 1, mid + 1, r, s, e);
    return max(le, ri);
  }
};
```

```
// persistent segment tree
const int N = (1<<17);
struct Node;
Node* empty;
struct Node{
	int sum;
	Node * lft, * rit;
	Node(){lft = rit = this; sum = 0;}
	Node(int s, Node* l = empty, Node* r = 
empty){
		lft = l, rit = r;
		sum = s;
	}
};
Node* insert(Node* cur, int ns, int ne, int val){
	if(val < ns || val > ne) return cur;
	if(ns==ne)	return new Node(cur-
>sum+1);
	int med = ns+((ne-ns)>>1);
	Node* lf = insert(cur->lft, ns, med, val);
	Node* rt = insert(cur->rit, med+1, ne, 
val);
	return new Node(lf->sum+rt->sum, lf, rt);
}

int query(Node* e, Node* s, int k, int ns, int ne){
	if(ns == ne)	return ns;
	int lsz = e->lft->sum - s->lft->sum;
	int med = ns+((ne-ns)>>1);
	if(k<=lsz)
		return query(e->lft, s->lft, k, ns, 
med);
	return query(e->rit, s->rit, k-lsz, med+1, 
ne);
}
Node* roots[N];
int m, x, q, l, r, k;
int main() {
	empty = new Node;
	roots[0] = empty;
	scanf("%d %d", &m, &q);
	for(int i = 1 ; i <= m ; ++i){
		scanf("%d", &x);
		roots[i] = insert(roots[i-1], -1e9, 
1e9, x);
	}
	while(q--){
		scanf("%d %d %d", &l, &r, &k);
		printf("%d\n", query(roots[r], 
roots[l-1], k, -1e9, 1e9));
	}
}
```

```
//dot product
double operator*(const Point &a, const Point &b)
{
  return a.x*b.x + a.y*b.y;
}

//cross product
double operator^(const Point &a, const Point &b)
{
  return a.x*b.y - a.y*b.x;
}
//multiplication by a factor
Point operator*(const double factor, const Point
& p){
  return Point(factor * p.x, factor * p.y);
}
Point operator*(const Point & p, const double
factor){
  return Point(factor * p.x, factor * p.y);
}
//comparisons (precision error)
bool operator==(const Point & a, const Point &
b){
  return a.x == b.x && a.y == b.y;
}
bool operator!=(const Point & a, const Point & b)
{
  return a.x != b.x || a.y != b.y;
}
//----------------------functions----------------------
// angle [-pi, pi]
double angle(const Point& p){
  return atan2(p.y, p.x);
}
double angle(const Point& a, const Point& b){
  return atan2(a^b, a*b);
}
Point rotate(const Point &p, double an){
  return Point(p.x * cos(an) - p.y * sin(an), p.x *
sin(an) + p.y * cos(an));
}
Point rotate(const Point &p, double an, Point&
around){
  return rotate(p - around, an) + around;
}
double norm(const Point &p){
  return sqrt(p*p);
}
Point perp(const Point &p){
  return Point(-p.y, p.x);
}
```

```
// GEOMTRY
struct Point{
  double x, y;
  Point(double x, double y):x(x), y(y){}

  Point():x(0), y(0){}

  //operators
  Point& operator=(const Point& o){
    x = o.x;
    y = o.y;
    return *this;
  }

  Point& operator+=(const Point& o){
    x += o.x;
    y += o.y;
    return *this;
  }

  Point& operator-=(const Point& o){
    x -= o.x;
    y -= o.y;
    return *this;
  }
  Point& operator*=(double fact){
    x -= fact;
    y -= fact;
    return *this;
  }
  Point& operator/=(double fact){
    x /= fact;
    y /= fact;
    return *this;
  }
};

Point any;

//--------------------operators----------------------
//minus
Point operator-(const Point &a){
  return Point(-a.x, -a.y);
}
//addition
Point operator+(const Point &a, const Point &b){
  return Point(a.x+b.x, a.y+b.y);
}
//subtraction
Point operator-(const Point &a, const Point &b){
  return Point(a.x-b.x, a.y-b.y);
}
```

```
}
void reflection(const Point &p, const Line &l,
Point &res){
  Point pr;
  proj(p, l, pr);
  res = p + 2 * (pr - p);
}
//----------------------segment----------------------
struct Segment{
  Point a, ab;
  Segment(const Point &a, const Point &b):a(a),
ab(b-a){}
  Segment():a(), ab(){}
  Point b () const {
    return a + ab;
  }
};
bool onsegment(const Segment& l, const Point&
p){
  return ((p - l.a) ^ l.ab) == 0 && ((p - l.a) * (p -
l.b())) <= 0;
}

double dist(const Segment& r, const Point& p,
Point& res = any){
  if((p - r.a) * (r.ab) <= 0){res = r.a;return norm(p
- r.a);}
  if((p - r.b()) * (-r.ab) <= 0){res = r.b();return
norm(p - r.b());};
  res = r.a + proj((p - r.a), r.ab) / norm(r.ab) * r.ab;
  return abs(proj(p-r.a, perp(r.ab)));
}

bool inter(const Segment& s1, const Segment
&s2, Point& res = any){
  if((s1.ab ^ s2.ab) == 0)return (onsegment(s1,
s2.a)
                                  || onsegment(s1,
s2.b())
                                  || onsegment(s2, s1.a)
                                  || onsegment(s2,
s1.b())
                                  ); // parallel
  double t1 = ((s2.a - s1.a) ^ s2.ab) / (s1.ab ^
s2.ab);
  double t2 = ((s1.a - s2.a) ^ s1.ab) / (s2.ab ^
s1.ab);
  if(t1 < 0 || t1 > 1 || t2 < 0 || t2 > 1)
        return 0;// does not intersect
  res = s1.a + s1.ab * t1;
  return 1;
}
Point bisector(const Point &a, const Point &b){
  return a * norm(b) + b * norm(a);
}
//projection
double proj(const Point &a, const Point &b){
  return a * b / norm(b);
}

//ccw , 0 -> collinear, > 0 counter clockwise, < 0
clockwise
double ccw(const Point &a, const Point &b, const
Point &orig){
  return (a - orig) ^ (b - orig);
}

double ccw(const Point &a, const Point &b){
  return (a) ^ (b);
}
//-------------------------lines-------------------------
struct Line{
  Point a, ab;
  Line(const Point &a, const Point &b):a(a),
ab(b-a){}
  Line():a(), ab(){}
  Point b(){
    return a + ab;
  }
};
// precision
bool online(const Line& l, Point& p){
  return ((p - l.a) ^ l.ab) == 0;
}
double dist(const Line& r, Point& p){
  return abs(proj(p-r.a, perp(r.ab)));
}
bool inter(const Line& s1, const Line &s2,
Point& res){
  if((s1.ab ^ s2.ab) == 0){
    if(online(s1, s2.a)); // coincident
    else; // parallel
    return 0;
  }
  double t = ((s2.a - s1.a) ^ s2.ab) / (s1.ab ^
s2.ab);
  res = s1.a + s1.ab * t;
  return 1;
}
double proj(const Point &p, const Line &l, Point
&res){
  double t = (p - l.a) * l.ab;
  res = l.a + t * l.ab;
  return t;
```

```
// bellman ford
#include <bits/stdc++.h>
using namespace std;
const int N = 2005;
const int M = 20000;
const int OO = 1000000000;
struct Edge{
  int u, v, w;
  Edge(){}
  Edge(int _u, int _v, int _w): u{_u}, v{_v},
w{_w}{}
  int other(int x){
    return x ^ u ^ v;
  }
} e[M];

int dist[N];
int par[N];
// directed weighted graph
bool bellman_ford(int s, int n = N, int m = M){
  for(int i = 0;i < n;i++)
    dist[i] = OO;
  dist[s] = 0;
  auto relax = [dist](int u, int edge){
    int v = e[edge].other(u);
    if(dist[u] + e[edge].w < dist[v])
      par[v] = u, dist[v] = dist[u] + e[edge].w;
  };
  for(int i = 0;i < n - 1;i++)
    for(int j = 0;j < m;j++)
      relax(e[j].u, j);
  for(int i = 0;i < m;i++)
    if(dist[e[i].u] + e[i].w < dist[e[i].v])
      return false;
  return true;
}
```

```
// articulation points
const int N = 200001;
const int M = 200001;
struct Edge{
  int u, v;
  int other(int i){
    return i ^ u ^ v;
  }
};
vector<int> adj[N];
Edge e[M];
bool is_artic[N];
int tt = 1;
int tin[N];
int dfs1(int u, int p = -1){
  tin[u] = tt++;
  int low = tin[u];
  int cnt = 0;
  for(auto el : adj[u]){
    int v = e[el].other(u);
    if(tin[v] == 0){// not yet visited
      int ch = dfs1(v, u);
      low = min(low, ch);
      if(p != -1 && ch >= tin[u])
        is_artic[u] = 1;
      cnt++;
    }else
      low = min(low, tin[v]);
  }
  if(cnt > 1 && p == -1)
    is_artic[u] = 1;
  return low;
}

int main(int argc, char** argv) {
//  freopen("in.txt", "r", stdin);
//  freopen("out.txt", "w", stdout);
  int n, m;
  cin >> n >> m;
  for(int i = 0;i < m;i++){
    scanf("%d %d", &e[i].u, &e[i].v);
    e[i].u--;
    e[i].v--;
    adj[e[i].u].push_back(i);
    adj[e[i].v].push_back(i);
  }
  for(int i = 0;i < n;i++)
    if(!tin[i])
      dfs1(i);

  return 0;
}
```

```
    e[i].u--;
    e[i].v--;
    adj[e[i].u].push_back(i);
    adj[e[i].v].push_back(i);
  }
  dfs1(0);
  build(0, com++);
}


// ------------------------------------------
// Convex hull optimization
struct Line {
	mutable ll k, m, p;
	bool operator<(const Line& o) const
{ return k < o.k; }
	bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
	// (for doubles, use inf = 1/.0, div(a,b) =
a/b)
	const ll inf = LLONG_MAX;
	ll div(ll a, ll b) { // floored division
		return a / b - ((a ^ b) < 0 && a %
b); }
	bool isect(iterator x, iterator y) {
		if (y == end()) { x->p = inf; return
false; }
		if (x->k == y->k) x->p = x->m >
y->m ? inf : -inf;
		else x->p = div(y->m - x->m, x->k
- y->k);
		return x->p >= y->p;
	}
	void add(ll k, ll m) {
		auto z = insert({k, m, 0}), y = z++,
x = y;
		while (isect(y, z)) z = erase(z);
		if (x != begin() && isect(--x, y))
isect(x, y = erase(y));
		while ((y = x) != begin() && (--x)-
>p >= y->p)
			isect(x, erase(y));
	}
	ll query(ll x) {
		assert(!empty());
		auto l = *lower_bound(x);
		return l.k * x + l.m;
	}
};
```

```
// bridge tree
const int N = 200001;
const int M = 200001;
struct Edge{
  int u, v;
  int other(int i){
    return i ^ u ^ v;
  }
};
vector<int> adj[N];
Edge e[M];
bool is_bridge[M];
int tt = 1;
int tin[N];
int dfs1(int u, int edge = -1){
  tin[u] = tt++;
  int res = tin[u];
  for(auto el : adj[u]){
    int v = e[el].other(u);
    if(tin[v] == 0)// not yet visited
      res = min(res, dfs1(v, el));
    else if(el != edge)// not the parent edge
      res = min(res, tin[v]);
  }
  if(edge != -1 && res == tin[u])// no back edges
from u's subtree
    is_bridge[edge] = true;
  return res;
}
vector<int> tree[N];
int com = 0;
bool vis[N];
int rep[N];
void build(int u, int cur){
  vis[u] = 1;
  rep[u] = cur;
  for(auto el : adj[u]){
    int v = e[el].other(u);
    if(vis[v])continue;
    if(is_bridge[el]){
      tree[cur].push_back(com);
      tree[com].push_back(cur);
      build(v, com++);
    }else
      build(v, cur);
  }
}
int main(int argc, char** argv) {
  int n, m;
  cin >> n >> m;
  for(int i = 0;i < m;i++){
    scanf("%d %d", &e[i].u, &e[i].v);
```

```
    e[par[u]].c     -= res;
    e[par[u] ^ 1].c += res;
    return res;
  }
  int augment(int s, int t){
    queue<int> q;
    q.push(s);
    vis[s] = tt;
    while(q.size()){
      int u = q.front();q.pop();
      if(u == t)break;
      for(auto el : adj[u]){
        Edge& ee = e[el];
        if(vis[ee.v] != tt && ee.c != 0){
          vis[ee.v] = tt;
          par[ee.v] = el;
          q.push(ee.v);
        }
      }
    }
    if(vis[t] != tt)
      return 0;
    return
      fix_path(s, t, OO);
  }
  void add_directed(int u, int v, int c){
    e.push_back(Edge(u, v, c));
    adj[u].push_back(e.size() - 1);
    e.push_back(Edge(v, u, 0));
    adj[v].push_back(e.size() - 1);
  }
};

int main(){
//  freopen("in.txt", "r", stdin);
//  freopen("out.txt", "w", stdout);
  int n, m;
  scanf("%d %d", &n, &m);
  int u, v, c;
  Edmond mf(n, m, false);
  for(int i = 0;i < m;i++){
    scanf("%d %d %d", &u, &v, &c);
    u--, v--;
    mf.add_edge(u, v, c);
  }
  printf("%lld", mf.max_flow(0, n - 1));
  return 0;
}
```

```
// Edmond
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int OO = 1e9;
class Edmond{
  public:
    bool directed;
    int n;
    int add;
    Edmond(int _n):directed{1}, n{_n}, add{0}{}
    Edmond(int _n, bool _directed):n{_n},
directed{_directed}, add{0}{
      par.resize(n);
      vis.resize(n);
      adj.resize(n);
      vis.resize(n);
    }
    void add_edge(int u, int v, int c){
      add_directed(u, v, c);
      if(!directed)
        add_directed(v, u, c);
    }
    ll max_flow(int s, int t){
      if(adj.size() <= s)return 0;
      ll res = 0;
      int aug = 0;
      while(aug = augment(s, t)){
        tt++;
        res += aug;
      }
      return res;
    }
    struct Edge{
      int oc;
      int u, v, c;
      Edge(int _u, int _v, int _c):u{_u}, v{_v},
c{_c}{oc = c;}
    };
    vector<vector<int>> adj;
    vector<Edge> e;
  private:
    const int OO = 1e9;
    int tt = 1;
    vector<int> vis;
    vector<int> par;
    int fix_path(int s, int u, int flow){
      if(u == s)
        return flow;
      int res = fix_path(s, e[par[u]].u, min(flow,
e[par[u]].c));
```

```cpp
// Chinese remainder theorem
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long ull;
typedef long long ll;
ll euclid(ll a, ll b, ll& x, ll& y){
  if(b == 0){
    x = 1;
    y = 0;
    return a;
  }
  ll x1, y1;
  ll g = euclid(b, a % b, x1, y1);
  x = y1;
  y = (x1 - a / b * y1);
  return g;
}
ll inv(ll n, int mod){
  ll x, y;
  euclid(n, mod, x, y);
  return x;
}
ll solve(vector<pair<int, int>> v){
  ll M = 1;
  for(auto &el : v)
    M *= el.second;
  ll res = 0;
  for(int i = 0;i < v.size();i++){
    res += v[i].first * (M / v[i].second) % M *
inv(M / v[i].second, v[i].second) % M;
    res %= M;
  }
  return (res + M) % M;
}
int main()
{
  //  freopen("in.txt", "r", stdin);
  //  freopen("out.txt", "w", stdout);
  cout << solve({{2, 3}, {3, 8}});
  return 0;
}
```

```cpp
// shortest path matrix exponentiation
#include <bits/stdc++.h>
using namespace std;
const int OO = 1000000000;
typedef long long ll;
typedef vector<vector<int>> Mat;
Mat sqr(const Mat& A){
  Mat C(A.size(), vector<int>(A.size(), OO));
  for(int i = 0;i < A.size();i++)
    for(int j = 0;j < A.size();j++)
      for(int k = 0;k < A.size();k++)
        C[i][j] = min(C[i][j], A[i][k] + A[k][j]);
  return C;
}
Mat shortest(Mat adj){
  int n = adj.size();
  int x = 1;
  while(x < n - 1){
    x *= 2;
    adj = sqr(adj);
  }
  return adj;
}
int main(int argc, char** argv) {
  int n, m;
  cin >> n >> m;

  Mat adj(n, vector<int>(n, OO));
  for(int i = 0;i < n;i++)
    adj[i][i] = 0;
  int u, v, w;
  for(int i = 0;i < m;i++){
    cin >> u >> v >> w;
    u--, v--;
    adj[u][v] = adj[v][u] = w;
  }
  adj = shortest(adj);
  for(int i = 0;i < n;i++){
    for(int j = 0;j < n;j++)
      cout << adj[i][j] << ' ';
    cout << endl;
  }
  return 0;
}
```

```
// HLD
using namespace std;
const int N = 100001;
const int LG = 18;
vector<int> adj[N];
int sz[N];
int sc[N];
int par[N];
int depth[N];
void dfs(int nd, int p, int d){
  par[nd] = p;
  depth[nd] = d;
  sz[nd] = 1;
  for(auto el : adj[nd]){
    if(el == p)continue;
    dfs(el, nd, d + 1);
    sz[nd] += sz[el];
    if(sc[nd] == 0 || sz[el] > sz[sc[nd]])sc[nd] = el;
  }
}
// tree is decomposed into paths. Each path is
stored in contiguous subarray in arr.
int pos_in_arr[N];
int arr[N*2];
int chain[N];
int chain_num = 0;
int head[N];
int pos = 0;
void hld(int nd){
  pos_in_arr[nd] = pos++;
  assert(pos <= N);
  arr[pos - 1] = -1;
  chain[nd] = chain_num;
  if(sc[nd] != 0){
    head[sc[nd]] = head[nd];
    hld(sc[nd]);
  }
  for(auto el : adj[nd]){
    if(sz[el] > sz[nd])continue;
    if(sc[nd] == el)continue;
    chain_num++;
    head[el] = el;
    hld(el);
  }
}
int main()
{
//  freopen("in.txt", "r", stdin);
//  freopen("out.txt", "w", stdout);
    dfs(0, -1, 0);
    hld(0);
}
```

```
// rabin miller primality test
#define EPS 1e-38
#define OO 1e9
#define ll long long
#define forn(i, n) for(int i = 0;i < n;i++)
const ll MOD = 1000000007;
using namespace std;
ll pow1(ll n, int p, int MOD){
  ll cur = n, res = 1;
  while(p){
    if(p & 1)
      (res *= cur) %= MOD;
    p /= 2;
    (cur *= cur) %= MOD;
  }
  return res;
}
bool is_prime(ll n){
  if(n == 2 || n == 3)
    return true;
  if(n < 2)
    return false;
  int s = 0;
  int m = n - 1;
  while(m % 2 == 0){
    m /= 2;
    s++;
  }
  for(int i = 0;i < 30;i++){
    ll x = rand() % (n - 3) + 2;
    ll p1 = pow1(x, m, n);

    if(p1 == 1 || p1 == n - 1)
      continue;
    bool ok = 0;
    for(int j = 0;j < s;j++){
      (p1 *= p1) %= n;
      if(p1 == n - 1){
        ok = 1;
        break;
      }
    }
    if(!ok)
      return false;
  }
  return true;
}

int main(){
  srand(time(0));
  cout << is_prime(1000000009) << endl;
}
```

```
  for(int i = 0;i < N;i++)
    st[i][0] = par[i];
  memset(st, -1, sizeof(st));
  for(int i = 1;i < LG;i++)
    for(int j = 0;j < N;j++)
      if(st[j][i - 1] != -1)
        st[j][i] = st[st[j][i-1]][i-1];
}
int lca(int u, int v){
  if(depth[u] < depth[v])swap(u, v);
  int dif = depth[u] - depth[v];
  for(int i = LG - 1;i >= 0;i--){
    if(dif & (1 << LG))
      u = st[u][i];
  }
  if(u == v)
    return u;
  for(int i = LG - 1;i >= 0;i--){
    if(st[u][i] != -1 && st[u][i] != st[v][i])
      u = st[u][i], v = st[v][i];
  }
  u = st[u][0];
  return u;
}

int dist(int u, int v){
  int lc = lca(u, v);
  return depth[u] + depth[v] - 2 * depth[lc];
}

int main()
{
//    freopen("in.txt", "r", stdin);
//    freopen("out.txt", "w", stdout);
    int n, q;
    scanf("%d %d", &n, &q);
    int m = n-1;
    int x, y;
    for(int i = 0;i < m;i++){
        scanf("%d %d", &x, &y);
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    int root = decompose(1);
    return 0;
}
```

```
// Centroid decomposition

#include <bits/stdc++.h>
#define ODD(n) (2*n+1)
#define OO 1e9
#define EVEN(n) (2*n)
using namespace std;
const int N = 100001;
vector<int> adj[N];
vector<int> dec_adj[N];
int dec_par[N];
int par[N];
int depth[N];
bool done[N];
int sz[N];
int dfs(int u, int p){
    if(done[u])return 0;
    if(p != -1)
      depth[u] = depth[p] + 1;
    par[u] = p;
    int res = 1;
    for(auto el : adj[u])
        if(p != el)
            res += dfs(el, u);
    return sz[u] = res;
}
int get_centroid(int u, int p, int tree_size){
    for(auto el : adj[u])
        if(p != el && !done[el] && sz[el] >
tree_size/2)
            return get_centroid(el, u, tree_size);
    return u;

}
void  add_edge(int a, int b){
    dec_adj[a].push_back(b);
    par[b] = a;
}
int decompose(int u){
    int tree_size = dfs(u, -1);
    int cent = get_centroid(u, -1, tree_size);
    done[cent] = 1;
    for(auto el : adj[cent])
        if(!done[el])
            add_edge(cent, decompose(el));
    return cent;
}

//lca
const int LG = 18;
int st[N][LG];
void init(){
```

```
sort_index[in][L[i-1].second];
        else
          sort_index[in][L[i].second] = i;
      }
    }
    void assign_suff(int in){
      suff.resize(n);
      for(int i = 0;i < n;i++){
        suff[sort_index[in][i]] = i;
      }
    }
    void compute_lcp(){
      lcp.resize(n);
      vector<int> rank(n);
      int k = 0;
      for(int i = 0;i < n;i++)
        rank[suff[i]] = i;
      for(int i = 0;i < n;i++, k && k--){
        if(rank[i] == n - 1){
          k = 0;
          continue;
        }
        int j = suff[rank[i] + 1];
        while(i + k < n && j + k < n && str[i+k]
== str[j+k])k++;
        lcp[rank[i]] = k;
      }
    }
};

// prefix function
vector<int> prefix_function(string str){
  int n = str.size();
  vector<int> pre(n);
  for(int i = 1;i < str.size();i++){
    int k = pre[i - 1];
    while(k && str[i] != str[k])
      k = pre[k - 1];

    if(str[k] == str[i])
      k++;

    pre[i] = k;
  }
  return pre;
}
```

```
// suffix array
#include <bits/stdc++.h>

using namespace std;
// N should be at least 1.5 * (2 ^ LGN)
const int N = 2 * (1 << 10);
// LGN is max power of 2 >= max size
const int LGN = 10;
class Suffix_array{
  public:
    int sort_index[LGN + 1][N];
    int n;
    string str;
    vector<int> suff;
    vector<int> lcp;
    Suffix_array(string _str){
      str = _str;
      memset(sort_index, -1, sizeof(sort_index));
      vector<pair<pair<int, int>, int>> L; // pair of
previous sorted index, index
      n = str.size();
      init_L(L);
      assign_sort_index(L, 0);
      for(int i = 1;i <= LGN;i++){
        assign_L(L, i);
        assign_sort_index(L, i);
      }
      assign_suff(LGN);
      compute_lcp();
    }

  private:
    void init_L(vector<pair<pair<int, int>,
int>>& L){
      L.resize(n);
      for(int i = 0;i < n;i++)
        L[i] = {{str[i], -1}, i};
      sort(L.begin(), L.end());
    }

    void assign_L(vector<pair<pair<int, int>,
int>>& L, int in){
      for(int i = 0;i < n;i++)
        L[i] = {{sort_index[in-1][i], sort_index[in-
1][i + (1 << (in - 1))]}, i};
      sort(L.begin(), L.end());
    }
    void assign_sort_index(vector<pair<pair<int,
int>, int>>& L, int in){
      for(int i = 0;i < n;i++){
        if(i && L[i].first == L[i-1].first)
          sort_index[in][L[i].second] =
```

```
            tree[new_in].c = c;
            tree[new_in].in = -1;
            tree[new_in].suffix_link =
tree[new_in].dict_link = -1;
            return new_in;
        }
        int go(int state, char c){
            if(tree[state].go[c-'a'] == -1){
                if(tree[state].next[c-'a'] != -1)
                    tree[state].go[c-'a'] =
tree[state].next[c-'a'];
                else
                    tree[state].go[c-'a'] = state == 0 ?
0 : go(get_suffix_link(state), c);
            }
            return tree[state].go[c-'a'];
        }
        int get_suffix_link(int state){
            if(tree[state].suffix_link == -1){
                if(state == 0 || tree[state].p == 0){
                    tree[state].suffix_link = 0;
                }
                else
                    tree[state].suffix_link =
go(get_suffix_link(tree[state].p), tree[state].c);
            }
            return tree[state].suffix_link;
        }
        int get_dict_link(int state){
            if(tree[state].dict_link == -1){
                if(state == 0)
                    tree[state].dict_link = 0;
                else{
                    int suf = get_suffix_link(state);
                    if(tree[suf].in != -1)
                        tree[state].dict_link = suf;
                    else
                        tree[state].dict_link =
get_dict_link(suf);
                }
            }
            return tree[state].dict_link;
        }
        void out(int state, int pos){
            if(tree[state].in == -1)
                state = get_dict_link(state);
            while(state != root){
                state = get_dict_link(state);
            }
        }
};
class Aho{
    public:
        Aho(){
            memset(tree[root].next, -1,
sizeof(tree[root].next));
            memset(tree[root].go, -1,
sizeof(tree[root].go));
            tree[root].suffix_link =
tree[root].dict_link = -1;
            tree[root].p = -1;
            tree[root].c = '$';
            tree[root].in = -1;
        }
        void add_string(string str){
            patterns.push_back(str);
            int state = root;
            for(int i = 0;i < str.size();i++){
                char c = str[i];
                if(tree[state].next[c-'a'] == -1)
                    tree[state].next[c-'a'] =
add_node(c, state);
                state = tree[state].next[c-'a'];
            }
            tree[state].in = patterns.size()-1;
        }
        void match(string str){
            int state = 0;
            for(int i = 0;i < str.size();i++){
                state = go(state, str[i]);
                out(state, i);
            }
        }
        struct node{
            int go[26];
            int next[26];
            char c;
            int p;
            int suffix_link;
            int dict_link;
            int in;
        } tree[10001];
        int sz = 1;
        int root =  0;
        vector<string> patterns;
        int add_node(char c, int p){
            int new_in = sz++;
            memset(tree[new_in].next, -1,
sizeof(tree[new_in].next));
            memset(tree[new_in].go, -1,
sizeof(tree[new_in].go));
            tree[new_in].p = p;
```

```
  for(int i = 1;i < n;i *= 2){
    for(int j = 0;j < n;j += 2 * i){
      for(int k = j;k < i + j;k++){
        Complex x = root[i + k - j] * perm[k + i];
        perm[k + i] = perm[k] - x;
        perm[k] = perm[k] + x;
      }
    }
  }
  return perm;
}
vector<Complex> multiply(vector<Complex> a,
vector<Complex> b){
  a = FFT(a);
  b = FFT(b);
  vector<Complex> r(a.size());
  for(int i = 0;i < r.size();i++){
    r[i] = a[i] * b[i];
    r[i].y *= -1;
  }
  r = FFT(r);
  for(auto& el : r){
    el = el / (double)N;
    el.y *= -1;
  }
  return r;
}
```

```
struct Complex{
  double x, y;
  Complex(){
    x = y = 0;
  }
  Complex(double _x, double _y) : x {_x}, y
{_y}{}
  Complex(double _x) : x {_x}, y {0} {}
};
Complex operator*(const Complex& a, const
Complex& b){
  return Complex(a.x * b.x - a.y * b.y, a.x * b.y +
a.y * b.x);
}
Complex operator/(const Complex& a, const
double d){
  return Complex(a.x / d, a.y / d);
}
Complex operator+(const Complex& a, const
Complex& b){
  return Complex(a.x + b.x, a.y + b.y);
}
Complex operator-(const Complex& a, const
Complex& b){
  return Complex(a.x - b.x, a.y - b.y);
}

const int LG = 19;
const int N = (1 << LG);
int rev[N];
Complex root[N];
void init(){
  rev[0] = 0;
  for(int i = 1;i < N;i++)
    rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (LG -
1));
  root[1].x = 1;
  for(int i = 1;i < LG;i++){
    double th = 2 * M_PI / (1 << (i + 1));
    Complex z(cos(th), sin(th));
    for(int j = (1 << (i - 1));j < (1 << i);j++){
      root[2 * j] = root[j];
      root[2 * j + 1] = root[j] * z;
    }
  }
}
vector<Complex> FFT(const
vector<Complex>& arr){
  int n = arr.size();
  vector<Complex> perm(n);
  for(int i = 0;i < n;i++)
    perm[i] = arr[rev[i]];
```

```
const int MAXN = 51;
int cap[MAXN][MAXN], path[MAXN],
prv[MAXN], ds[MAXN];

bool visited[MAXN];
int n, src, sink, pathLength;

int min(int a, int b) {
        return a < b ? a : b;
}
/* Dijkstra's varient[MaxiMin] to augmenting
path */
int getPath(int StartNode, int TargetNode) {
        int i, maxd, maxi, cur;
        memset(visited, 0, sizeof(bool) * n);
        memcpy(ds, &cap[StartNode][0],
sizeof(int) * n);

        cur = StartNode, visited[cur] = true;

        for (i = 0; i < n; i++)
                prv[i] = cur;

        while (1) {
                maxd = 0, maxi = -1;

                for (i = 0; i < n; i++) {
                        if (!visited[i] && ds[i] >
maxd)
                                maxd = ds[i], maxi
= i;
                }

                if (maxd == 0)
                        break;
                if (maxi == TargetNode)
                        break;
                cur = maxi, visited[cur] = true;

                for (i = 0; i < n; i++) {
                        if (min(ds[cur], cap[cur][i])
> ds[i]) /* MaxiMin */
                                ds[i] = min(ds[cur],
cap[cur][i]), prv[i] = cur;
                }
        }
        int pi = TargetNode;
        pathLength = 0;

        while (1) {
                path[pathLength++] = pi;
                if (pi == StartNode)
```

```
struct point {
  int X, Y;
      point operator - (const point & o) const {
            return point({ X - o.X, Y - o.Y });
     }
};
int n, q;

long long cross(const point& a, const point& b) {
 return 1LL * a.X * b.Y - 1LL * a.Y * b.X;
}

long long dot(const point  & a, const point& b) {
   return 1LL * a.X * b.X + 1LL * a.Y * b.Y;
}

int winding(const vector<point>& v, point p) {
       int wn = 0;
     for (int i = 0, len = v.size(); i < len; ++i) {
        int j = i == (len - 1) ? 0 : (i + 1);
          if (v[i].Y <= p.Y)
                  wn += (v[j].Y > p.Y) && cross(v[j]
- v[i], p - v[i]) > 0;
              else
                  wn -= (v[j].Y <= p.Y) && cross(v[j]
- v[i], p - v[i]) < 0;
      }
      return wn;
}

bool pointOnSegment(point a, point b, point p) {
    return !cross(a - p, b - p) && dot(p - a, b - a)
>= 0
                && dot(p - b, a - b) >= 0;
}
```

```
                int a, b, c;
                cin >> a >> b >> c;
                cap[a - 1][b - 1] = cap[b - 1][a - 1] = c;

                e[i] = edge(a - 1, b - 1);
            }

            int max_ = maxFlow();
            //cout<<max_<<"\n";

            memset(visited, 0, sizeof(bool) * n);

            /* find all nodes reachable from src */
            flood_fill(src);

            for (i = 0; i < links; i++) { /* any edge that is reachable from u but not v is mincut edge */
                if (visited[e[i].from] && !visited[e[i].to])
                    cout << e[i].from + 1 << " " << e[i].to + 1 << "\n";
                else if (visited[e[i].to] && !visited[e[i].from])
                    cout << e[i].from + 1 << " " << e[i].to + 1 << "\n";
            }
            cout << "\n";
        }
        return 0;
}
```

```
            break;
        pi = prv[pi];
    }
    //reverse(path,path+pathLength);

    return ds[TargetNode];
}

int maxFlow() {
    int newflow, m, n, tf = 0;

    while (newflow = getPath(src, sink)) {
        for (int i = pathLength - 1; i > 0; i--) {
            m = path[i], n = path[i - 1];
            cap[m][n] -= newflow;
            cap[n][m] += newflow;
        }
        tf += newflow;
    }
    return tf;
}

void flood_fill(int src) {
    visited[src] = 1;
    for (int i = 0; i < n; i++)
        if (!visited[i] && cap[src][i] > 0)
            flood_fill(i);
}

struct edge {
    int from, to;
    edge() {
    }
    edge(int f, int t) {
        from = f, to = t;
    }
};
edge e[501];

int main() {
    int i, j, links;

    while (cin >> n >> links && (n || links)) {
        src = 0;
        sink = 1;

        for (i = 0; i < n + 2; i++)
            for (j = 0; j < n + 2; j++)
                cap[i][j] = 0;

        for (i = 0; i < links; i++) {
```

```
	scanf("%d", &n);
	for (int i = 1; i <= n; i++)
		scanf("%lld", &a[i]);
	for (int i = 2; i <= n; i++) {
		scanf("%d%lld", &u, &c);
		man s;
		s.co = c;
		s.son = i;
		edg[u].push_back(s);
	}
	dfs(1);
	for (int i = 1; i <= n; i++) {
		int f = Find(i);
		ans[fa[0][f]]--;
		ans[fa[0][i]]++;
	}
	sum_dfs(1);
	for (int i = 1; i <= n; i++)
		printf("%lld ", ans[i]);
	printf("\n");
	return 0;
}
```

```
#define inf 0x3f3f3f3f
#define met(a,b) memset(a,b,sizeof a)
#define pb push_back
typedef long long ll;
using namespace std;
const int N = 2e5 + 50;
int n, m, k, u;
int fa[20][N];
ll cost[20][N];
ll a[N], ans[N];
struct man {
	int son;
	ll co;
};
vector<man> edg[N];
void dfs(int x) {
	for (int i = 1; fa[i - 1][fa[i - 1][x]]; i++) {
		fa[i][x] = fa[i - 1][fa[i - 1][x]];
		cost[i][x] = cost[i - 1][x] + cost[i -
1][fa[i - 1][x]];
	}
	for (int i = 0; i < edg[x].size(); i++) {
		man e = edg[x][i];
		int v = e.son;
		ll c = e.co;
		fa[0][v] = x;
		cost[0][v] = c;
		dfs(v);
	}
}
int Find(int x) {
	ll c = a[x];
	for (int i = 19; i >= 0; i--) {
		if (cost[i][x] <= c && fa[i][x]) {
			c -= cost[i][x];
			x = fa[i][x];
		}
	}
	return x;
}
void sum_dfs(int x) {
	for (int i = 0; i < edg[x].size(); i++) {
		man e = edg[x][i];
		int v = e.son;
		ll c = e.co;
		sum_dfs(v);
		ans[x] += ans[v];
	}
}
int main() {
	ll c;
	met(ans, 0);
```

```
		comps.back().push_back(x);
				comp[x] = sz(comps) - 1;
			}
		}
}

void scc() {
		int n = sz(adjList);

		inStack.clear();		inStack.resize(n);
		lowLink.clear();		lowLink.resize(n);
		dfn.clear();			dfn.resize(n, -1);
		ndfn = 0;

		comp.clear(), comp.resize(n);
		comps.clear();

		lp(i, n) if (dfn[i] == -1)
			tarjan(i);
}

void computeCompGraph() {
		int csz = comps.size(), cntSrc = csz,
cntSnk = csz;

		outDeg.clear();
		outDeg.resize(csz);
		inDeg.clear();
		inDeg.resize(comps.size());
		dagList.clear();
		dagList.resize(csz);	//will contain
duplicates

		for (int i = 0; i < sz(adjList); i++)
			for (int j = 0; j < sz(adjList[i]); j+
+) {
				int k = adjList[i][j];
				if (comp[k] != comp[i]) {

		dagList[comp[i]].push_back(comp[k]);
		//reverse
					if (!
(inDeg[comp[k]]++))		cntSrc--;
					if (!
(outDeg[comp[i]]++))			cntSnk--;
				} else
					;// this edge is for a
component comp[k]
			}

		/* Min edges to convert DAG to one cycle
```

```
Euler -O(E)
- Euler path 2nodes(start&end) odd degress, all
other even
- Euler cycle all even
//use all edges only once
void euler(vector< vi > & adjMax, vi & ret, int n,
int i, bool isDirected = false) {
		lp(j, n) {
			if(adjMax[i][j]) {
				adjMax[i][j]--;
				if(!isDirected)
		adjMax[j][i]--;
				euler( adjMax, ret, n, j,
isDirected );
			}
		}
		ret.push_back( i );
}

----------------
Tarjan -O(E + V)
//strongly connected components - component
graph - bridges - art points - biconnected
component
vector< vector<int> > adjList, comps, dagList;
vector<int> inStack, lowLink, dfn, comp, inDeg,
outDeg;
stack<int> stk;
int ndfn, cntSrc, cntSnk;

void tarjan(int node) {
		lowLink[node] = dfn[node] = ndfn++,
inStack[node] = 1;
		stk.push(node);

		rep(i, adjList[node]) {
			int ch = adjList[node][i];
			if (dfn[ch] == -1) {
				tarjan(ch);
				lowLink[node] =
min(lowLink[node], lowLink[ch]);
			} else if (inStack[ch])
				lowLink[node] =
min(lowLink[node], dfn[ch]);
		}

		if (lowLink[node] == dfn[node]) {
			comps.push_back(vector<int> ());
			int x = -1;
			while (x != node) {
				x = stk.top(), stk.pop(),
inStack[x] = 0;
```

```
                        low[u] = min(low[u],
low[w]);
                }
                else if(w != v)
                        low[u] = min(low[u],
dfn[w]);
        }
}

bool root = false;
void art_points(int i, int p) {
        low[i] = dfn[i] = ++num;

        for (int j = 0; j < n; ++j) if (adjMat[i][j]
&& j != p) {
                if (dfn[j] == 0) {
                        art_points(j, i);
                        low[i] = min(low[i],
low[j]);
                        if (low[j] >= dfn[i]){
                                if(dfn[i] == 0 &&
root == false)
                                        root = true;
                                else

        artpoints.insert(i);
                        }

                } else if (i != j)
                        low[i] = min(low[i],
dfn[j]);
        }
}

stack<pair<int,int>> component;
void find_biconnected_component(int node,int
par){
        lowLink[node] = dfn[node] = ndfn++;

        rep(i, adjList[node][i]){
                int ch = adjList[node][i];
                if(node != ch && dfn[ch] <
dfn[node])

        component.push(make_pair(node,ch));

                if(dfn[ch] == -1){

        find_biconnected_component(ch,node);
                        lowLink[node] =
min(lowLink[node],lowLink[ch]);
        if (comps.size() == 1)
                cout << "0\n";
        else {
                cout << max(cntSrc, cntSnk) <<
"\n";
        }
        */
}

void art_bridges(int i, int p) {
        low[i] = dfn[i] = ++num;

        for (int j = 0; j < n; ++j) if (adjMat[i][j]
&& j != p) {
                if (dfn[j] == 0) {
                        art_bridges(j, i);
                        low[i] = min(low[i],
low[j]);
                        if (low[j] == dfn[j])

        bridges.push_back(make_pair(min(i, j),
max(i, j)));
                } else
                        low[i] = min(low[i],
dfn[j]);
        }
}

void run_art_bridges() {
        lp(i, n) low[i] = -1, dfn[i] = 0;

        bridges.clear();

        lp(i, n) if (!dfn[i])
                art_bridges(i, -1);

        sort(all(bridges));

}

void dfnlow(int u, int v)       //Just for calcing Tw
tabels
{
        int i, w;

        dfn[u] = low[u] = num++;
        for(i=0;i<graph[u].size();i++)
        {
                w = graph[u][i];
                if(dfn[w] <= NOT_VISITED)
                {
                        dfnlow(w, u);
```

```
                        comp[x] = sz(comps) - 1;
                }
                cmp_root_node[ comp[node] ] = node;
        }
}

void scc() {
        int n = sz(adjList);

        inStack.clear();
        inStack.resize(n);
        lowLink.clear();
        lowLink.resize(n);
        assigned_val.clear();
        assigned_val.resize(n);
        cmp_root_node.clear();
        cmp_root_node.resize(n);
        dfn.clear();                    dfn.resize(n, -1);
        ndfn = 0;

        comp.clear(), comp.resize(n);
        comps.clear();

        lp(i, n) if (dfn[i] == -1)
                tarjan(i);
}

void add_or(int a, int b){
        adjList[NOT(a)].push_back(b);
        adjList[NOT(b)].push_back(a);
}

void add_or_not_both(int a, int b){
        add_or(a, b);
        add_or(NOT(a), NOT(b));
}

void force_value(int i, bool b){
        if(b)
                adjList[NOT(i)].push_back(i);
        else
                adjList[i].push_back(NOT(i));
}

bool is_solvable(){
   for(int i = 0 ; i < n ; i += 2)
      if( comp[i] == comp[NOT(i)] )
          return false;
   return true;
}
```

```
                        if(lowLink[ch] >= dfn[node]){
                                //Get all edges till reach (node,ch) edge
                                do {
                                        edge = component.top(); component.pop();
                                        cout << edge.first+1 << " " << edge.second+1 << "\n";
                                }while(edge.first != node || edge.second != ch);
                        }
                }else if(node != ch)
                        lowLink[node] = min(lowLink[node], dfn[ch]);
        }
}
----------
2 Satisfiability -O(n+m)
//(x1 or x2) and (x1- or x3) = ?
#define NOT(x)    (1^(x))

int n, m, ndfn;
vector< vector<int> > adjList, comps, dagList;
vector<int> inStack, lowLink, dfn, comp, assigned_val, cmp_root_node;
stack<int> stk;

void tarjan(int node) {
        lowLink[node] = dfn[node] = ndfn++, inStack[node] = 1;
        stk.push(node);

        rep(i, adjList[node]) {
                int ch = adjList[node][i];
                if (dfn[ch] == -1) {
                        tarjan(ch);
                        lowLink[node] = min(lowLink[node], lowLink[ch]);
                } else if (inStack[ch])
                        lowLink[node] = min(lowLink[node], dfn[ch]);
        }

        if (lowLink[node] == dfn[node]) {
                comps.push_back(vector<int> ());
                int x = -1;
                while (x != node) {
                        x = stk.top(), stk.pop(), inStack[x] = 0;

                comps.back().push_back(x);
```

```
		if(!visited[i] || cap[StartNode]
[i]<=0) continue;
		ret =
getPath(i,TargetNode,ourLen+1,min(maxcap,cap[
StartNode][i]));
		if(ret>0) break;
	}
	return ret;
}
int maxFlow (int src,int sink){
	int total_flow = 0;
	while(1){
		memset(visited,0,sizeof visited);
		int newflow =
getPath(src,sink,0,INT_MAX);
		if(!newflow) break;
		for(int i=1; i<pathLength; i++){
			int m = path[i-1],n=path[i];
			cap[m][n]-=newflow;
			cap[n][m]+=newflow;
		}
		total_flow+=newflow;
	}
	return total_flow;
}
//max edge-disjoint path
-set all weights on edges to 1 then max flow
//Capacities on vertices
-vertex splitting (node to edge) then max flow
//Max independent path ^^
-set all weights on edges&nodes to 1 then max
flow
//Multi-source Multi-sink
-add super source & super sink(with edge to OO)
then max flow
//Max bipartite matching
-multi-source with max edge-disjoint then max
flow (not only 1)
//Min path Coverage
-n-m
//Find MinCut Edges
-flood fill mark sources and sinks (every zero
between S&T)
----------
Maximum Bipartite Matching -O(EV)
bool canMatch(int i) // O(E)
	{
		rep(j, adjMax[i]) if(adjMax[i][j]
&& !vis[j]) {
			vis[j] = 1;
			if( colAssign[j] < 0 ||
canMatch(colAssign[j]) ) {
```

```
void assign_values(){
	vector<int>
comp_assigned_value(comps.size(), -1);
	lp(i, comps.size()) {
		if (comp_assigned_value[i] == -1)
{
			comp_assigned_value[i] =
1;
			int not_ithcomp =
comp[ NOT(cmp_root_node[i]) ];

	comp_assigned_value[ not_ithcomp ] = 0;
		}
	}
	lp(i, n)
		assigned_val[i] =
comp_assigned_value[ comp[i] ];
}

int main() {
	while(cin>>n>>m && (n+m)){
		adjList.clear();
		adjList.resize(n);

		lp(i, m){
			cin>>a>>b;
			add_or(a, b);
		}
		scc();
		if (!is_solvable()){
			cout<<"no solution\n";
			continue;
		}
		assign_values();
	}
	return 0;
}

----------
Maximum Flow -O(max_flow * E)
//max flow - min cut
int getPath(int StartNode, int TargetNode,int
ourLen, int maxcap){
	path[ourLen] = StartNode;
	if(StartNode == TargetNode){
		pathLength = ourLen+1;
		return maxcap;
	}
	int ret = 0;
	vistited[StartNode]=true;
	for(int i=0; i<n; i++){
```

```
		}
		rep(i, mnPathCvs)
		{
			rep(j, mnPathCvs[i])

	cout<<mnPathCvs[i][j]<<" ";
			cout<<"\n";
		}
		*/

		return matches;
	}
	----------
Lowest Common Ancestor -O(N) -Q(logN)
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const int MAX = 5001;
const int LOGMAX = 13;

int P[MAX][LOGMAX], parent[MAX],
depth[MAX];
vector<vector<int> > tree;

void tree_dfs(int i, int p = -1, int d = 0) {
	parent[i] = p, depth[i] = d;
	for (int j = 0; j < tree[i].size(); j++)
		if (tree[i][j] != p)
			tree_dfs(tree[i][j], i, d + 1);
}

void preprocessing(int n) // O(nlog(n)) // P[i][j] is
2^j ancestor of i
		{
	memset(parent, -1, n * sizeof(int));
	for (int i = 0; i < n; i++)
		if (parent[i] == -1)
			tree_dfs(i, i);

	for (int i = 0; i < n; i++) {
		for (int j = 0; j < (1 << j); j++)
			P[i][j] = -1;
		P[i][0] = parent[i]; // the first
ancestor of node
	}

	for (int j = 1; (1 << j) < n; j++)
		for (int i = 0; i < n; i++)
			if (P[i][j - 1] != -1)
```

```
				colAssign[j] = i,
rowAssign[i] = j;
				return true;
			}
		}
		return false;
	}

	typedef vector<int>	vi;
	vector<vi> adjMax;
	vi vis, colAssign, rowAssign;

	vector< pair<int, int> >
bipartiteMatching()	// O(EV)
	{	// In case spares graph, use adjList
		vector< pair<int, int> > matches;

		if(sz(adjMax) == 0)
			return matches;

		int maxFlow = 0, rows =
sz(adjMax), cols = sz(adjMax[0]);
		colAssign = vi(cols, -1),
rowAssign = vi(rows, -1);

		lp(i, rows) {
			vis = vi(cols, 0);
			if( canMatch(i)  )
				maxFlow++;
		}

		lp(j, cols) if(colAssign[j] != -1)


matches.push_back( make_pair(colAssign[j], j) );
		// this is col sorted list...u can use
rowAssign for reverse
		// as you see, rowAssign was not
important now

		/*
		vector< vector<int> > mnPathCvs;

		lp(j, n) if(colAssign[j] == -1) {
			vector<int> v(1, j+1);
			int t = rowAssign[j];
			while(t != -1) {
				v.push_back(t+1);
				t = sz(v)%2 ?
rowAssign[j] : colAssign[j] ;
			}
			mnPathCvs.push_back(v);
```

```
int main() {
    //freopen("input.txt","r",stdin);
    //freopen("output.txt","w",stdout);
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            scanf("%d", &a[i][j]);

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < MAXN; j++) {
            used[j] = false;
            minval[j] = INF;
        }
        int j_cur = 0;
        par[j_cur] = i;
        do {
            used[j_cur] = true;
            int j_next, delta = INF, i_cur = par[j_cur];
            for (int j = 0; j <= n; j++)
                if (!used[j]) {
                    int cur = a[i_cur][j] - u[i_cur] - v[j];
                    if (cur < minval[j]) {
                        minval[j] = cur; link[j] = j_cur;
                    }
                    if (minval[j] < delta) {
                        delta = minval[j]; j_next = j;
                    }
                }
            for (int j = 0; j <= n; j++)
                if (used[j]) {
                    u[par[j]] += delta; v[j] -= delta;
                }
                else {
                    minval[j] -= delta;
                }
            j_cur = j_next;
        } while (par[j_cur]);
        do {
            int j_prev = link[j_cur];
            par[j_cur] = par[j_prev];
            j_cur = j_prev;
        } while (j_cur > 0);
    }

    printf("%d", -v[0]);
    return 0;
}
```

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

//call pre-processing before using it
int LCA(int n, int p, int q) //O(log(n))
        {
    if (depth[p] < depth[q])
        swap(p, q);
    int log;
    for (log = 1; 1 << log <= depth[p]; log++)
        ;
    log--;

    for (int i = log; i >= 0; i--)
        //find ancestor of p on the same level with q
        if (depth[p] - (1 << i) >= depth[q])
            p = P[p][i];

    if (p == q)
        return p; //one of them parent of another

    for (int i = log; i >= 0; i--)
        //we compute LCA(p, q) using the values in P
        if (P[p][i] != -1 && P[p][i] != P[q][i])
            p = P[p][i], q = P[q][i];

    return parent[p];
}

int shortestPath(int n, int p, int q) {
    return depth[p] + depth[q] - 2 * depth[LCA(n, p, q)] + 1;
}

int main() {

    return 0;
}
```

Hungarian matching algorithm - O(N ^ 3)

```
const int MAXN = 105;
const int INF = 1000 * 1000 * 1000;

int n;
int a[MAXN][MAXN];
int u[MAXN], v[MAXN], link[MAXN], par[MAXN], used[MAXN], minval[MAXN];
```

```
	for(int i=n-1; i>0; i--)
		freq[pi[i-1]]+=freq[i];

	freq.erase(freq.begin());
}
```

num of unique pre : freq of prefix at #1
num of uniqe substring : For each prefix P
-O(n^2)

```
	cnt += CountUniquePrefixes(reverse(P))
```

Aho-Corasick -O(m+n+k) # k total occurrences

```
//match many patterns in 1 string
const int MAX = 26 + 1;			//+1 to assign starting from 1
//TODO: Input limits, to be modified
const int MAX_PAT_LEN = 15;
const int MAX_PATS = 30;
char pats[MAX_PATS][MAX_PAT_LEN];
char arr[100005];


int nextNodeId = 0;

int cIdx(char c) {
	return 1 + c - 'a';
	//return c;		//in case any problems
}

//you can either use Large MAX (e.g. 260) or reassign string to smalled indices
struct node {
	node* fail;
	node* child[MAX];
	vector<int> patIdx;
	vector<char> chars;
	int id;

	node() {
		memset(child, 0, sizeof child);
		id = nextNodeId++;
	}

	void insert(char* str, int idx, bool firstCall = true) {
		// You can remove next part, it just reassign string to indices to keep small mem
		if (firstCall) {
			char *t = str;
			while (*t)
				*t = cIdx(*t), t++;
		}
```

KMP -O(n+m)

```
//pattern matching- longest suffix or prefix palindrome
// - add chars to get palindrome - repetition
void KMP(string str,string pat){
	int n = sz(str);
	int m = sz(pat);
	vector<int> longestPrefix = computePrefix(pat);

	for(int i =0,k=0; i<n; i++){
		while(k>0 && pat[k] != str[i])
			k = longestPrefix[k-1];

		if(pat[k] == str[i])
			k++;
		if(k==m){
			cout << i-m+1 << "\n";
			k = longestPrefix[k-1];
		}
	}
}

vector<int> computePrefix(string pat){
	int m = sz(pat);
	vector<int> longestPrefix(m);

	for(int i =1,k=0; i<m; ++i){
		while(k>0 && pat[k] != pat[i])
			k = longestPrefix[k-1];

		if(pat[k] == pat[i])
			longestPrefix[i] = ++k;
		else
			longestPrefix[i] = k;
	}
	return longestPrefix;
}
```

longest suffix : rts@str use failure func - ans f[size-1]
add char to pal : str + reverse of str before longest suffix
repetition :if Len%(Len-F[Len-1]) = 0 - ans Len-F[Len-1] -(s=p+p 2nd match)
freqency of all prefixes:

```
void fp (){
	vector<int> pi = computePrefix(pat);
	in n = sz(pi);
	vector<int> freq(n+1);
	for(int i=0; i<n; i++)
		++freq[pi[i]];
```

```
                        move(k, ch);

                        cur->child[ch]->fail = k;

                        cur->child[ch]->patIdx.insert(cur->child[ch]->patIdx.end(),
                                                      k->patIdx.begin(), k->patIdx.end());
                }
        }
        return root;
}

void match(const char* str, node* root, vector<vector<int> > & res) {
        node* k = root;

        for (int i = 0; str[i]; i++) {
                move(k, str[i]);
                for (int j = 0; j < k->patIdx.size(); j++)
                        res[k->patIdx[j]].push_back(i);
        }
}

int main() {

        scanf("%s", arr);

        int k;
        scanf("%d", &k);

        for (int i = 0; i < k; i++)
                scanf("%s", &pats[i]);

        node* root = buildAhoTree(pats, k);

        vector<vector<int> > res(k);
        match(arr, root, res);

        for (int i = 0; i < k; i++) {
                if (sz(res[i]))
                        printf("y\n");
                else
                        printf("n\n");
        }

        return 0;
}
```

```
                ////////////////////////////////

                if (!*str) {
                        patIdx.push_back(idx);
                } else {
                        if (!child[*str]) {
                                child[*str] = new node();

        chars.push_back(*str);
                        }
                        child[*str]->insert(str + 1, idx, false);
                }
        }
};

void move(node* &k, char c) {
        while (!k->child[c])
                k = k->fail;
        k = k->child[c];
}

node* buildAhoTree(char pat[][MAX_PAT_LEN], int len) {
        node * root = new node();

        for (int i = 0; i < len; i++) {
                root->insert(pat[i], i);
        }

        queue<node*> q;

        for (int i = 0; i < MAX; i++)
                if (root->child[i])
                        q.push(root->child[i]), root->child[i]->fail = root;
                else
                        root->child[i] = root;
//Sentinel

        node* cur;
        while (!q.empty()) {
                cur = q.front();
                q.pop();

                for (int i = 0; i < cur->chars.size(); i++) {
                        int ch = cur->chars[i];
                        q.push(cur->child[ch]);
                        node* k = cur->fail;
```

```
vector<point> cutPolygon(point a, point b, const
vector<point> &Q)
{
   vector<point> P;
   for (int i = 0; i < (int)Q.size(); i++)
   {
      double left1 = cross(toVec(a, b), toVec(a,
Q[i])), left2 = 0;
      if (i != (int)Q.size()-1)
         left2 = cross(toVec(a, b), toVec(a,
Q[i+1]));
      if (left1 > -EPS) P.push_back(Q[i]); // Q[i] is
on the left of ab
      if (left1 * left2 < -EPS) // edge (Q[i],
Q[i+1]) crosses line ab
         P.push_back(lineIntersectSeg(Q[i],
Q[i+1], a, b));
   }
   if (!P.empty() && !(P.back() == P.front()))
      P.push_back(P.front()); // make P's first
point = P's last point
   return P;
}

//check if the polygon isconvex
bool isConvex(const vector<point> &P)   //
returns true if all three
{
   int sz = (int)P.size(); // consecutive vertices of
P form the same turns
   if (sz <= 3) return false; // a point/sz=2 or a
line/sz=3 is not convex
   bool isLeft = ccw(P[0], P[1], P[2]); //
remember one result
   for (int i = 1; i < sz-1; i++) // then compare
with the others
      if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2])
!= isLeft)
         return false; // different sign -> this
polygon is concave
   return true;
}
```

```
// A C++ program that implements Z algorithm
for pattern searching
void getZarr(string str, int Z[]);
void search(string text, string pattern)
{
   string concat = pattern + "$" + text;
   int l = concat.length();
   int Z[l];
   getZarr(concat, Z);
   for (int i = 0; i < l; ++i)
   {
      if (Z[i] == pattern.length())
         cout << "Pattern found at index "
            << i - pattern.length() -1 << endl;
   }
}
void getZarr(string str, int Z[])
{
   int n = str.length();
   int L, R, k;

   L = R = 0;
   for (int i = 1; i < n; ++i)
   {
      if (i > R)
      {
         L = R = i;
         while (R<n && str[R-L] == str[R])
            R++;
         Z[i] = R-L;
         R--;
      }
      else
      {
         k = i-L;
         if (Z[k] < R-i+1)
            Z[i] = Z[k];
         else
         {
            L = i;
            while (R<n && str[R-L] == str[R])
               R++;
            Z[i] = R-L;
            R--;
         }
      }
   }
}
```