ICPC Team Reference Material

Contents

1 Setup

1.1 Vimrc

```
let mapleader = "\"
syntax on
filetype plugin on
set nocompatible
set autoread
set foldmethod=marker
set autoindent
set clipboard+=unnamedplus
set number relativenumber
set shiftwidth=2 softtabstop=2 expandtab
map <leader>c :w! && !compile %:p:r<CR>
vmap < <gv
vmap > >gv
```

1.2 Capslock as Escape

```
setxkbmap -layout us
xmodmap -e 'clear Lock'
xmodmap -e 'keycode 66 = Escape'
```

1.3 Compilation

```
#!/bin/bash
# put this file in .local/bin or add its dir to the PATH variable
compile() {
  g++ -Wall -Wextra -Wshadow -Ofast -std=c++17 -pedantic -Wformat=2 -Wconversion -Wlogical-op -Wshift-
      overflow=2 -Wduplicated-cond -Wfloat-equal -fno-sanitize-recover -fstack-protector -fsanitize=
      address,undefined -fmax-errors=2 -o "$1"{,.cpp}
}
compile "$1"
```

2 Graph algorithms

2.1 Adjacency list representation

```
template <class T>
class Graph {
public:
  vector <int> _head, _next, _to;
  vector <T> _cost;
  int edge_number;
  bool isDirected;

  Graph() = default;
  Graph(int V, int E, bool isDirec) {
    isDirected = isDirec;
    _head.assign(V + 9, 0);
    _next.assign(isDirected ? E + 9 : E * 2 + 9, 0);
    _to.assign(isDirected ? E + 9 : E * 2 + 9, 0);
    // _cost.assign(isDirected ? E + 9 : E * 2 + 9, 0);
    edge_number = 0;
  }

  void addEdge(int u, int v, T w = 0) {
    _next[++edge_number] = _head[u];
    _to[edge_number] = v;
    // _cost[edge_number] = w;
    _head[u] = edge_number;
  }

  void addBiEdge(int u, int v, int w = 0) {
    addEdge(u, v, w);
    addEdge(v, u, w);
  }

  void dfs(int node) {
    vis[node] = true;
    for(int i = _head[node]; i; i = _next[i]) if(!vis[_to[i]]) {
  dfs(_to[i]);
      }
  }
};
```

2.2 Articulation points and bridges

```
const int N = 1e5 + 9, M = 2e6 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Next[M], To[M], Cost[M];
int Par[N], dfs_num[N], dfs_low[N];
int ne, n, m, u, v, w;
int root, rootChildren, dfs_timer, bridgeInx;
bool Art[N];
vector < pair <int, int> > bridges(M);

void addEdge(int from, int to, int cost = 0) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head,    0, sizeof(Head[0])    * (n + 2));
  memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
  memset(Par,    -1, sizeof(Par[0])     * (n + 2));
  memset(Art,     0, sizeof(Art[0])     * (n + 2));
  ne = dfs_timer = bridgeInx = 0;
}

void Tarjan(int node) {
  dfs_num[node] = dfs_low[node] = ++dfs_timer;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0)
      {
  if(node == root) ++rootChildren;

  Par[To[i]] = node;
  Tarjan(To[i]);
  dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);

  if(dfs_low[To[i]] >= dfs_num[node])
    Art[node] = true;

  if(dfs_low[To[i]] > dfs_num[node])
    bridges[bridgeInx++] = make_pair(node, To[i]);
      }
    else if(To[i] != Par[node])
      dfs_low[node] = Min(dfs_low[node], dfs_num[To[i]]);
  }
}

int main() {
  cin >> n >> m;
  _clear();

  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0) {
      root = i;
      rootChildren = 0;
      Tarjan(i);
      Art[root] = (rootChildren > 1);
    }

  cout << "Art Points :\n";
  for(int i = 1; i <= n; ++i) if(Art[i])
        cout << i << " ";

  cout << "\nBridges :\n";
  for(int i = 0; i < bridgeInx; ++i)
    cout << bridges[i].first << " - " << bridges[i].second << endl;
}
```

2.3 Bellman ford

```
//  Bellman-Ford Algorithm
//  In programming contests, the slowness of Bellman Ford and its negative cycle detection feature
//  causes it to be used only to solve the SSSP problem on small graph
//  which is not guaranteed to be free from negative weight cycle

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

bool hasNC() {
  for(int i = 1; i <= n; ++i)
    for(int j = Head[i]; j; j = Next[j])
      if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]])
  return true;

  return false;
}

bool Bellman_Ford(int src) {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  dis[src] = 0;
  bool newRelaxation = true;

  for(int i = 2; i <= n && newRelaxation; ++i) {
    newRelaxation = false;
    for(int i = 1; i <= n; ++i)
      for(int j = Head[i]; j; j = Next[j])
  if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]]) {
    dis[To[j]] = dis[i] + Cost[j];
    Par[To[j]] = i;
    newRelaxation = true;
  }
  }
```

```
  return hasNC();
}
```

2.4 Bi-connected components

```
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Next[M], To[M];
int Par[N], dfs_num[N], dfs_low[N];
int ne, n, m, u, v;
int root, rootChildren, dfs_timer;
int Stack[N], top, ID;
bool Art[N];
vector < vector <int> > BiCCs(N), BiCCIDs(N);

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void _clear() {
  memset(Head,       0, sizeof(Head[0])       * (n + 2));
  memset(dfs_num,    0, sizeof(dfs_num[0])    * (n + 2));
  memset(Par,       -1, sizeof(Par[0])        * (n + 2));
  memset(Art,        0, sizeof(Art[0])        * (n + 2));
  ne = dfs_timer = top = ID = 0;
  BiCCs = BiCCIDs = vector < vector <int> > (N);
}

void Tarjan(int node) {
  dfs_num[node] = dfs_low[node] = ++dfs_timer;
  Stack[top++] = node;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0) {
      if(node == root) ++rootChildren;

      Par[To[i]] = node;
      Tarjan(To[i]);

      dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);

      if(dfs_low[To[i]] >= dfs_num[node]) {
  Art[node] = true;
  ++ID;
  for(int x = -1; x ^ To[i];) {
    x = Stack[--top];
    BiCCIDs[x].emplace_back(ID);
    BiCCs[ID].emplace_back(x);
  }
  BiCCIDs[node].emplace_back(ID);
  BiCCs[ID].emplace_back(node);
      }
    }
    else if(To[i] != Par[node])
      dfs_low[node] = Min(dfs_low[node], dfs_num[To[i]]);
  }
}

int main() {
  cin >> n >> m;
  _clear();

  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0) { // O(n + m)
      root = i;
      rootChildren = 0;
      Tarjan(i);
      Art[root] = (rootChildren > 1);
    }

  for(int i = 1; i <= ID; ++i) {
    cout << "Component : " << i << " contains : ";
    for(int j = 0; j < (int)BiCCs[i].size(); ++j)
      cout << BiCCs[i][j] << " \n"[j == BiCCs[i].size() - 1];
  }
}
```

2.5 Bi-partite graph

```
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];
bool color[N], vis[N];

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

bool checkBiPartite(int node, int par = 0) {
  if(vis[node])
    return color[par] != color[node];

  color[node] = color[par] ^ 1;

  vis[node] = true;
  bool ok = true;
  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != par)
      ok &= checkBiPartite(To[i], node);

  return ok;
}

int main() {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  bool isBiPartite = true;
  for(int i = 1; i <= n; ++i) if(!vis[i])
        isBiPartite &= checkBiPartite(i);

  cout << (isBiPartite ? "YES" : "NO") << endl;
}
```

2.6 Breadth first search (BFS)

```
#include <bits/stdc++.h>
using namespace std;
typedef int64_t  ll;

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void BFS(int src)  {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  queue <int> q;
  q.push(src);
  dis[src] = 0;

  int u;
  while(q.size())  {
    u = q.front(); q.pop();
    for(int i = Head[u]; i; i = Next[i])
      if(dis[To[i]] == oo) {
  dis[To[i]] = dis[u] + 1;
```

```cpp
    Par[To[i]] = u;
    q.push(To[i]);
      }
    }
}

int main()
{
  cin >> n >> m >> st >> tr;
  while(m--) {
    cin >> u >> v >> tax;
    addEdge(u, v, tax);
    addEdge(v, u, tax);
  }

  BFS(st);
  cout << dis[tr] << endl;
}
```

2.7 Connected components

```cpp
const int N = 1e5 + 9, M = 1e6 + 9;

int Head[N], Next[M], To[M], ne, u, v, n, m, CCs;
bool visited[N];

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void DFS(int node) {
  visited[node] = true;
  for(int e = Head[node]; e; e = Next[e])
    if(!visited[To[e]])
      DFS(To[e]);
}

int main() {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  for(int node = 1; node <= n; ++node) if(!visited[node])
          ++CCs, DFS(node);

  cout << CCs << endl;
}
```

2.8 Cycle detection (directed graph)

```cpp
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];
bool hasCycle;
char visited[N];

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void DFS(int node) {
  if(hasCycle |= visited[node] == 1)
    return; /** Oops, revisiting active node **/
  visited[node] = 1; /** current node legend mode has been activated **/

  for(int i = Head[node]; i; i = Next[i])
    if(visited[To[i]] != 2)
      DFS(To[i]);

  visited[node] = 2; /** done with this node and mark it as visited **/
}

int main() {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
  }

  for(int i = 1; i <= n; ++i)
    if(!visited[i])
      DFS(i);

  cout << (hasCycle ? "YES" : "NO") << endl;
}
```

2.9 Cycle detection (undirected graph)

```cpp
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];
bool visited[N], hasCycle;

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void DFS(int node, int parent = -1) {
  if(hasCycle |= visited[node])
    return;
  visited[node] = true;

  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != parent)
      DFS(To[i], node);
}

int main() {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  for(int i = 1; i <= n; ++i)
    if(!visited[i])
      DFS(i);

  cout << (hasCycle ? "YES" : "NO") << endl;
}
```

2.10 Depth first search (DFS)

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef int64_t  ll;

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];
bool vis[N];

void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void DFS(int node)  {
  vis[node] = true;
  for(int i = Head[node]; i; i = Next[i])
    if(!vis[To[i]])
```

```
      DFS(To[i]);
}

int main()  {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  for(int i = 1; i <= n; ++i)
    if(!vis[i])
      DFS(i);
}
```

2.11 Dijkstra (dense graph)

```
/** Dijkstra on dense graphs
    complexity : O(n^2 + m)
**/

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void Dijkstra(int src, int V) {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  vector <bool> mark(V + 1, false);

  dis[src] = 0;
  for(int i = 1; i <= V; ++i) {
    int node = 0;
    for(int j = 1; j <= V; ++j)
      if(!mark[j] && dis[j] < dis[node])
  node = j;

    if(dis[node] == INF) break;

    mark[node] = true;
    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]]) {
  dis[To[i]] = dis[node] + Cost[i];
  Par[To[i]] = node;
      }
  }
}
```

2.12 Dijkstra (grid)

```
const int dr[]   = { 1, -1, 0, 0, 1, 1, -1, -1 };
const int dc[]   = { 0, 0, 1, -1, 1, -1, 1, -1 };
const char dir[] = {'D', 'U', 'R', 'L'};

const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;

int grid[N][N], dis[N][N], n, m;

bool valid(int r, int c) {
  return r >= 1 && r <= n && c >= 1 && c <= m;
}

void Dijkstra(int sr, int sc) {
  memset(dis, 0x3f, sizeof (dis)); // memset(dis, 0x3f, n * m) we don't do that here

  priority_queue <tuple <int, int, int> > Q;

  dis[sr][sc] = grid[sr][sc];
  Q.push({-grid[sr][sc], sr, sc});

  int cost, r, c, nr, nc;
  while(Q.size()) {
    tie(cost, r, c) = Q.top(); Q.pop();
    if((-cost) > dis[r][c]) continue; // lazy deletion

    for(int i = 0; i < 4; ++i) {
      nr = r + dr[i];
      nc = c + dc[i];

      if(!valid(nr, nc)) continue;

      if(dis[r][c] + grid[nr][nc] < dis[nr][nc]) {
  dis[nr][nc] = dis[r][c] + grid[nr][nc];
  Q.push({-dis[nr][nc], nr, nc});
      }
    }
  }
}
```

2.13 Dijkstra (negative weighted graph)

```
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void Dijkstra(int src) {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  priority_queue <pair <ll, int> > Q;

  dis[src] = 0;
  Q.push({-dis[src], src});

  int node;
  ll cost;
  while(Q.size()) {
    tie(cost, node) = Q.top(); Q.pop();
    if((-cost) > dis[node]) continue;

    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]]) {
  dis[To[i]] = dis[node] + Cost[i];
  Q.push({-dis[To[i]], To[i]});
  Par[To[i]] = node;
      }
  }
}
```

2.14 Dijkstra (sparse graph)

```
/** Dijkstra on sparse graphs
    - complexity : O(n + m)logn -> O(nlogn + m)
    - Single Source Single Destination Shortest Path Problem
    - Positive Weight Edges only
    * Subpaths of shortest paths from u to v are shortest paths!
    **/

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];
```

```cpp
void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void Dijkstra(int src, int trg) {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  priority_queue <pair <ll, int> > Q;

  dis[src] = 0;
  Q.push({-dis[src], src});

  int node;
  ll cost;
  while(Q.size()) {
    tie(cost, node) = Q.top(); Q.pop();

    if((-cost) > dis[node]) continue; // lazy deletion
    if(node == trg) return;           // cheapest cost in case of positive weight edges

    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]]) {
  dis[To[i]] = dis[node] + Cost[i];
  Q.push({-dis[To[i]], To[i]});
  Par[To[i]] = node;
      }
  }
}
```

2.15 Directed cyclic graph into acyclic

```cpp
const int N = 1e5 + 9, M = 2e6 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], To[M], Next[M], Cost[M];
int dfs_num[N], dfs_low[N], out[N];
int Stack[N], compID[N], compSize[N];
int ne, n, m, u, v, w;
int dfs_timer, top, ID;
bool in_stack[N];

int HeadDAG[N], ToDAG[M], NextDAG[M], CostDAG[M], neDAG;

void addEdge(int from, int to, int cost = 0) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void addEdgeDAG(int from, int to, int cost = 0) {
  NextDAG[++neDAG] = HeadDAG[from];
  HeadDAG[from] = neDAG;
  CostDAG[ne] = cost;
  ToDAG[neDAG] = to;
  ++out[from];
}

void _clear() {
  memset(Head,     0, sizeof(Head[0])     * (n + 2));
  memset(dfs_num,  0, sizeof(dfs_num[0])  * (n + 2));
  memset(compID,   0, sizeof(compID[0])   * (n + 2));
  memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
  memset(HeadDAG,  0, sizeof(HeadDAG[0])  * (n + 2));
  memset(out,      0, sizeof(out[0])      * (n + 2));
  ne = dfs_timer = top = neDAG = ID = 0;
}

void Tarjan(int node) {
  dfs_num[node] = dfs_low[node] = ++dfs_timer;
  in_stack[Stack[top++] = node] = true;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0)
      Tarjan(To[i]);

    if(in_stack[To[i]])
      dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
  }

  if(dfs_num[node] == dfs_low[node]) {
    ++ID;
    for(int cur = -1; cur ^ node;) {
      in_stack[cur = Stack[--top]] = false;
      compID[cur] = ID;
      ++compSize[ID];
    }
  }
}

void Tarjan() {
  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0)
      Tarjan(i);
}

void DFS(int node) {
  dfs_num[node] = 1;
  for(int i = Head[node]; i; i = Next[i]) {
    if(compID[node] != compID[To[i]])
      addEdgeDAG(compID[node], compID[To[i]]);

    if(dfs_num[To[i]] == 0)
      DFS(To[i]);
  }
}

void construct_dag() {
  memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));

  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0)
      DFS(i);
}
```

2.16 Edge classification

```cpp
#pragma GCC optimize ("Ofast")

#include <bits/stdc++.h>

#define UNVISITED  0
#define EXPLORED   1
#define VISITED    2

using namespace std;

typedef int64_t  ll;

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Next[M], To[M], Par[N], in_time[N], ne, n, m, u, v, dfs_timer;
char dfs_num[N];

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void edgeClassification(int node) {
  dfs_num[node] = EXPLORED;
  in_time[node] = ++dfs_timer;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == UNVISITED) {
      cout << "Tree Edge : " << node << " -> " << To[i] << endl;

      Par[To[i]] = node;
      edgeClassification(To[i]);
    }
    else if(dfs_num[To[i]] == VISITED) {
      /** Cross Edges only occur in directed graph */
      if(in_time[To[i]] < in_time[node])
  cout << "Cross Edge : " << node << " -> " << To[i] << endl;
      else
  cout << "Forward Edge : " << node << " -> " << To[i] << endl;
    }
    else if(dfs_num[To[i]] == EXPLORED) {
      if(Par[node] == To[i])
  cout << "Bi-Directional Edge : " << node << " -> " << To[i] << endl;
      else
  cout << "Backward Edge : " << node << " -> " << To[i] << " (Cycle)" << endl;
    }
```

```
  }

  dfs_num[node] = VISITED;
}

int main() {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
  }

  for(int i = 1; i <= n; ++i)
    if(!dfs_num[i])
      edgeClassification(i);
}
```

2.17 Eulerian tour tree

```
int Head[N], To[M], Next[M], Cost[M];
int ne, n, m, u, v, w;

int Last[N], First[N], euler_tour[1 + N << 1];
ll  Height[1 + N << 1];
int euler_timer;

void addEdge(int from, int to, int cost = 0) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head,       0, sizeof(Head[0])       * (n + 2));
  memset(Last,       0, sizeof(Last[0])       * (n + 2));
  memset(First,      0, sizeof(First[0])      * (n + 2));
  ne = euler_timer = 0;
}

/**
   euler_tour[1 .. n * 2 - 1] = which records the sequence of visited nodes
   Height[1 .. n * 2 - 1]     = which records the depth of each visited node

   First[1 .. n]      = records the index of the first occurrence of node i in euler_tour
   Last[1 .. n]       = records the index of the last  occurrence of node i in euler_tour
**/

void EulerianTour(int node, ll depth = 0) {
  euler_tour[++euler_timer] = node;
  Height[euler_timer] = depth;
  First[node] = euler_timer;

  for(int i = Head[node]; i; i = Next[i])
    if(First[To[i]] == 0) {
      EulerianTour(To[i], depth + Cost[i]);

      euler_tour[++euler_timer] = node;
      Height[euler_timer] = depth;
    }

  Last[node] = euler_timer;
}

void show() {
  for(int i = 1; i < (n << 1); ++i) cout << euler_tour[i] << " ";cout << endl;
  for(int i = 1; i < (n << 1); ++i) cout << Height[i] << " ";    cout << endl;
  for(int i = 1; i <= n; ++i)       cout << First[i] << " ";     cout << endl;
  for(int i = 1; i <= n; ++i)       cout << Last[i] << " ";      cout << endl;
}

int main() {
  cin >> n >> m;
  _clear();

  while(m--) {
    cin >> u >> v >> w;
    addEdge(u, v, w);
    addEdge(v, u, w);
  }

  EulerianTour(1);
  show();
}
```

2.18 Flood fill

```
/** check if there is a path from (0, 0) to (n - 1, m - 1) using ’.’ only **/

int dr[4] = {1, -1, 0, 0};
int dc[4] = {0, 0, 1, -1};
char grid[N][M];
int n, m;

bool valid(int r, int c) {
  return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] == ’.’;
}

bool isDis(int r, int c) {
  return r == n - 1 && c == m - 1;
}

bool FloodFill(int r, int c) {
  if(!valid(r, c)) return false;
  if(isDis(r, c)) return true;

  grid[r][c] = ’#’;
  for(int i = 0; i < 4; ++i)
    if(FloodFill(r + dr[i], c + dc[i]))
      return true;
  return false;
}

int main() {
  cin >> n >> m;
  for(int i = 0; i < n; ++i)
    for(int j = 0; j < m; ++j)
      cin >> grid[i][j];

  cout << (FloodFill(0, 0) ? "YES" : "NO") << endl;
}
```

2.19 Floyd warshall (all-pairs shortest path)

```
/** -The graph has a ’negative cycle’ if at the end of the algorithm,
  the distance from a vertex v to itself is negative.

  - before k-th phase the value of d[i][j] is equal to the length of
  the shortest path from vertex i to the vertex j,
  if this path is allowed to enter only the vertex with numbers smaller than k
  (the beginning and end of the path are not restricted by this property).
**/

const int N = 500 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
const i64 INF = 0x3f3f3f3f3f3f3f3f;

int Par[N][N], n, m, u, v, tax;
i64 adj[N][N], dis[N][N];

vector <int> restorePath(int st, int tr) {
  vector <int> path;
  if(dis[st][tr] == INF) return path;

  for(int i = tr; st ^ i; i = Par[st][i])
    path.push_back(i);

  path.push_back(st);
  reverse(path.begin(), path.end());
  return path;
}

void Floyd_Warshall() {
  for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= n; ++j)
      Par[i][j] = i;

  for(int k = 1; k <= n; ++k)
    for(int i = 1; i <= n; ++i)
      for(int j = 1; j <= n; ++j)
  if(dis[i][k] + dis[k][j] < dis[i][j]) {
    dis[i][j] = dis[i][k] + dis[k][j];
    Par[i][j] = Par[k][j];
  }
}
```

2.20 Minimum spanning tree (Kruskal)

```cpp
class UnionFind {
  vector <int> par;
  vector <int> siz;
  int num_sets;
  size_t sz;

public:
  UnionFind() : par(1, -1), siz(1, 1), num_sets(0), sz(0) {}
  UnionFind(int n) : par(n + 1, -1), siz(n + 1, 1), num_sets(n), sz(n) {}

  int find_set(int u) {
    assert(u <= sz);

    int leader;
    for(leader = u; ~par[leader]; leader = par[leader]);

    for(int next = par[u]; u != leader; next = par[next]) {
      par[u] = leader;
      u = next;
    }
    return leader;
  }

  bool same_set(int u, int v) {
    return find_set(u) == find_set(v);
  }

  bool union_set(int u, int v) {
    if(same_set(u, v)) return false;

    int x = find_set(u);
    int y = find_set(v);

    if(siz[x] < siz[y]) swap(x, y);

    par[y] = x;
    siz[x] += siz[y];

    --num_sets;
    return true;
  }

  int number_of_sets() {
    return num_sets;
  }

  int size_of_set(int u) {
    return siz[find_set(u)];
  }

  size_t size() {
    return sz;
  }

  void clear() {
    par.clear();
    siz.clear();
    sz = num_sets = 0;
  }

  void assign(size_t n) {
    par.assign(n + 1, -1);
    siz.assign(n + 1,  1);
    sz = num_sets = n;
  }

  map < int, vector <int> > groups(int st) {
    map < int, vector <int> > ret;

    for(size_t i = st; i < sz + st; ++i)
      ret[find_set(i)].push_back(i);

    return ret;
  }
};

int n, m, u, v, w;
vector < tuple <int, int, int> > edges;
UnionFind uf;

pair < ll, vector < pair <int, int> > > Kruskal() {
  sort(edges.begin(), edges.end());

  vector < pair <int, int> > mstEdges;
  int from, to, cost;
  ll minWieght = 0;

  for(tuple <int, int, int> edge : edges) {
    tie(cost, from, to) = edge;
    if(uf.union_set(from, to)) {
      minWieght += cost;
      mstEdges.push_back(make_pair(from, to));
    }
  }

  if(mstEdges.size() == n - 1)
    return make_pair(minWieght, mstEdges);

  return make_pair(-1, vector < pair <int, int> > ());
}
```

2.21 Kth ancestor and lowest common ancestor (binary lifting)

```cpp
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const int LOG = 20;

int Head[N], To[M], Next[M], Par[N];
int up[N][LOG + 1];
int Log[N], Level[N];
int ne, n, u, v, q;

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void _clear() {
  memset(Head,  0, sizeof(Head[0])   * (n + 2));
  memset(Par,   0, sizeof(Par[0])    * (n + 2));
  memset(Level, 0, sizeof(Level[0])  * (n + 2));
  ne = 0;
}

int lastBit(int a) {
  return (a & -a);
}

void logCalc() {
  Log[1] = 0;
  for(int i = 2; i < N; ++i)
    Log[i] = Log[i >> 1] + 1;
}

void DFS(int node, int depth = 0) {
  Level[node] = depth;
  up[node][0] = Par[node];  // Par[root] = root

  for(int i = 1; i <= LOG; ++i) {
    up[node][i] = up[up[node][i - 1]][i - 1];
  }

  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != Par[node]) {
      Par[To[i]] = node;
      DFS(To[i], depth + 1);
    }
}

int KthAncestor(int u, int k) {
  if(k > Level[u]) return -1;

  for(int i = lastBit(k); k; k -= lastBit(k), i = lastBit(k))
    u = up[u][Log[i]];

  return u;
}

int LCA(int u, int v) {
  if(Level[u] < Level[v]) swap(u, v);
  int k = Level[u] - Level[v];

  u = KthAncestor(u, k);
  if(u == v) return u;

  for(int i = LOG; i >= 0; --i)
    if(up[u][i] ^ up[v][i])
      {
  u = up[u][i];
  v = up[v][i];
      }

  return up[u][0];
}

int main() {
  cin >> n;
```

```
  _clear();

  for(int i = 1; i < n; ++i) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  logCalc();
  for(int i = 1; i <= n; ++i) if(Par[i] == 0) {
      Par[i] = i;
      DFS(i);
    }

  cin >> q;
  while(q--) {
    cin >> u >> v;
    cout << LCA(u, v) << endl;
  }
}
```

2.22 Lowest common ancestor (euler tour)

```
template <class T, class F = function <T(const T&, const T&)> >
class SparseTable {
  int _N;
  int _LOG;
  vector <T> _A;
  vector < vector <T> > ST;
  vector <int> Log;
  F func;

public :
  SparseTable() = default;

  template <class iter>
  SparseTable(iter _begin, iter _end, const F _func = less <T>()) : func(_func) {
    _N = distance(_begin, _end);
    Log.assign(_N + 1, 0);
    for(int i = 2; i <= _N; ++i)
      Log[i] = Log[i >> 1] + 1;

    _LOG = Log[_N];

    _A.assign(_N + 1, 0);
    ST.assign(_N + 1, vector <T> (_LOG + 1, 0));

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; ++i, ++j)
      _A[j] = *i;

    build();
  }

  void build() {
    for(int i = 1; i <= _N; ++i)
      ST[i][0] = i;

    for(int j = 1, k, d; j <= _LOG; ++j) {
      k = (1 << j);
      d = (k >> 1);

      for(int i = 1; i + k - 1 <= _N; ++i) {
  T const & x = ST[i][j - 1];
  T const & y = ST[i + d][j - 1];

  ST[i][j] = func(_A[x], _A[y]) ? x : y;
      }
    }
  }

  T query(int l, int r) {
    int d = r - l + 1;
    T const & x = ST[l][Log[d]];
    T const & y = ST[l + d - (1 << Log[d])][Log[d]];

    return func(_A[x], _A[y]) ? x : y;
  }
};

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], To[M], Next[M], Cost[M];
int ne, n, m, u, v, w, q;

int Last[N], First[N], euler_tour[N << 1];
int  Height[N << 1];
int euler_timer;

void addEdge(int from, int to, int cost = 1) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head,  0, sizeof(Head[0])  * (n + 2));
  memset(Last,  0, sizeof(Last[0])  * (n + 2));
  memset(First, 0, sizeof(First[0]) * (n + 2));
  ne = euler_timer = 0;
}

void EulerianTour(int node, int depth = 0) {
  euler_tour[++euler_timer] = node;
  Height[euler_timer] = depth;
  First[node] = euler_timer;

  for(int i = Head[node]; i; i = Next[i])
    if(First[To[i]] == 0) {
      EulerianTour(To[i], depth + Cost[i]);

      euler_tour[++euler_timer] = node;
      Height[euler_timer] = depth;
    }

  Last[node] = euler_timer;
}

int main() {
  cin >> n >> m;
  _clear();

  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  EulerianTour(1);

  SparseTable <int> st(Height + 1, Height + euler_timer + 1, [&](int a, int b) { return a <= b; });

  int l, r; cin >> q;
  while(q--) {
    cin >> l >> r;

    int left = Last[l];
    int right = Last[r];
    if(left > right) swap(left, right);

    cout << euler_tour[ st.query(left, right) ] << endl;
  }
}
```

2.23 Minimum vertex cover

```
const int N = 1e5 + 9;

int Head[N], Next[N << 1], To[N << 1], ne, u, v, n, MVC;

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

bool DFS(int node, int par = -1) {
  bool black = false;
  for(int e = Head[node]; e; e = Next[e])
    if(To[e] != par)
      black |= DFS(To[e], node);

  MVC += black;
  return !black;
}

int main() {
  cin >> n;
  while(--n) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
```

```
  }

  DFS(1);
  cout << MVC << endl;
}
```

2.24 Restoring the path

```
const int dr []     = {-1, 0, 1, 0};
const int dc []     = {0, 1, 0, -1};
const char dir []   = {'U', 'R', 'D', 'L'};
map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};

/** Implicit Graphs

    - in BFS, Dijkstra or Bellman-Ford function write -> Par[nr][nc] = dir[i ^ 2]
    - char Par[N][N] initialize with -1
    - si start i
    - sj strat j
    - fi target i
    - fj target j
    - char dir and its map inv
    - dr, dc
**/

string restorePath(int si, int sj, int fi, int fj) {
  string s;
  if(Par[ei][ej] == -1) return s;

  int ei = fi, ej = fj;
  for(char i = Par[fi][fj]; (si ^ fi) || (sj ^ fj); i = Par[fi][fj]) {
    s  += dir[inv[i] ^ 2];
    fi += dr[inv[i]];
    fj += dc[inv[i]];
  }

  reverse(s.begin(), s.end());
  return s;
}

/** Explicit Graphs (BFS, Dijkstra or Bellman-Ford)

    - int Par[N] initialize with  -1
    - ll dis[N]  initialize with  0x3f
    - ll INF = 0x3f3f3f3f3f3f3f3f
**/

vector <int> restorePath(int dest) {
  vector <int> path;
  if(dis[dest] == INF) return path;

  for(int i = dest; ~i; i = Par[i])
    path.push_back(i);

  reverse(path.begin(), path.end());
  return path;
}

/** in case of Floyd-Warshall:

    - ll dis[N][N] initialize with 0x3f
    - ll INF = 0x3f3f3f3f3f3f3f3f
    - int Par[N][N] initialize with       Par[i][j] = i;
    - in Floyd-Warshall function write -> Par[i][j] = Par[k][j];
**/

vector <int> restorePath(int st, int tr) {
  vector <int> path;
  if(dis[st][tr] == INF) return path;

  for(int i = tr; st ^ i; i = Par[st][i])
    path.push_back(i);

  path.push_back(st);
  reverse(path.begin(), path.end());
  return path;
}
```

2.25 Shortest path faster algorithml (SPFA)

```
/** Shortest Path Faster Algorithm :
    - This algorithm runs in O(kE) where k is a number depending on the graph.
    - The maximum k can be V (which is the same as the time complexity of Bellman Fords).
    - However, in practice SPFA (which uses a queue) is as fast as Dijkstras (which uses a priority
          queue).
    - SPFA can deal with negative weight edge. If the graph has no negative cycle, SPFA runs well on
          it.
    - If the graph has negative cycle(s), SPFA can also detect it as there must be some vertex (those
          on the negative cycle)
    that enters the queue for over V , 1 times.
**/

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;
ll mINF = 0xc0c0c0c0c0c0c0c0;

int Head[N], Par[N], Next[M], To[M], Cost[M], Cnt[N], ne, n, m, u, v, st, tax;
ll dis[N];
bool Inq[N];

void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void _set() {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));
  memset(Cnt,    0, sizeof(Cnt[0]) * (n + 2));
  memset(Inq,    0, sizeof(Inq[0]) * (n + 2));
}

bool SPFA(int src) {
  _set();

  deque <int> Q;
  Q.push_front(src);

  dis[src] = 0;
  Cnt[src] = 1;
  Inq[src] = 1;

  int node;
  while(Q.size()) {
    node = Q.front(); Q.pop_front(); Inq[node] = 0;

    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]]) {
  dis[To[i]] = dis[node] + Cost[i];
  Par[To[i]] = node;

  if(!Inq[To[i]]) {
    if(++Cnt[To[i]] == n)
      return true; // graph has a negative weight cycle

    if(Q.size() && dis[To[i]] > dis[Q.front()])
      Q.push_back(To[i]);
    else
      Q.push_front(To[i]);

    Inq[To[i]] = true;
  }
      }
  }
  return false;
}
```

2.26 Single source shortest path

```
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], ne, n, m, u, v, st, tr;
ll dis[N];

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void _clear() {
```

```cpp
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void BFS(int src) {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  queue <int> Q;
  Q.push(src);
  dis[src] = 0;

  int node;
  while(Q.size()) {
    node = Q.front(); Q.pop();
    for(int i = Head[node]; i; i = Next[i]) if(dis[To[i]] == INF) {
  dis[To[i]] = dis[node] + 1;
  Par[To[i]] = node;
  Q.push(To[i]);
      }
  }
}
```

2.27 Single source shortest path (grid)

```cpp
const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

const int dr []     = {-1, 0, 1, 0};
const int dc []     = {0, 1, 0, -1};
const char dir []   = {'U', 'R', 'D', 'L'};
map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};

int dis[N][N], n, m;
char Par[N][N];

bool valid(int r, int c) {
  return r >= 1 && r <= n && c >= 1 && c <= m && dis[r][c] == oo;
}

void BFS(int sr, int sc) {
  memset(dis, 0x3f, sizeof(dis));
  memset(Par,   -1, sizeof(Par));

  queue < pair <int, int> > Q;
  dis[sr][sc] = 0;
  Q.push({sr, sc});

  int r, c, nr, nc;
  while(Q.size()) {
    tie(r, c) = Q.front(); Q.pop();

    for(int i = 0; i < 4; ++i) {
      nr = r + dr[i];
      nc = c + dc[i];

      if(!valid(nr, nc)) continue;

      dis[nr][nc] = dis[r][c] + 1;
      Par[nr][nc] = dir[i ^ 2];
      Q.push({nr, nc});
    }
  }
}
```

2.28 Tarjan (strongly connected components)

```cpp
const int N = 1e5 + 9, M = 2e6 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], To[M], Next[M], Cost[M];
int dfs_num[N], dfs_low[N];
int Stack[N], compID[N], compSize[N];
int ne, n, m, u, v, w;
int dfs_timer, top, ID;
bool in_stack[N];

void addEdge(int from, int to, int cost = 0) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head,     0, sizeof(Head[0])     * (n + 2));
  memset(dfs_num,  0, sizeof(dfs_num[0])  * (n + 2));
  memset(compID,   0, sizeof(compID[0])   * (n + 2));
  memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
  ne = dfs_timer = top = ID = 0;
}

void Tarjan(int node) {
  dfs_num[node] = dfs_low[node] = ++dfs_timer;
  in_stack[Stack[top++] = node] = true;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0)
      Tarjan(To[i]);

    if(in_stack[To[i]])
      dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
  }

  if(dfs_num[node] == dfs_low[node]) {
    ++ID;
    for(int cur = -1; cur ^ node;) {
      in_stack[cur = Stack[--top]] = false;
      compID[cur] = ID;
      ++compSize[ID];
    }
  }
}

void Tarjan() {
  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0)
      Tarjan(i);
}
```

2.29 Topological sort (DFS)

```cpp
int Head[N], Next[M], To[M], ne, n, m, u, v;
bool vis[N];
vector <int> t_sort;

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void DFS(int node) {
  vis[node] = true;
  for(int i = Head[node]; i; i = Next[i])
    if(!vis[To[i]])
      DFS(To[i]);

  t_sort.push_back(node);
}

vector <int> topological_sort(int n) {

  t_sort.clear();
  for(int i = 1; i <= n; ++i) if(!vis[i])
        DFS(i);

  reverse(t_sort.begin(), t_sort.end());
  return t_sort;
}

int main() {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
  }

  vector <int> v = topological_sort(n);
  for(int i : v)
    cout << i << ' ';
}
```

2.30 Topological sort (kahns algorithm)

```
int Head[N], Next[M], To[M], in[N], ne, n, m, u, v;

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

vector <int> kahn(int n) {
  vector <int> ready, ret;

  for(int i = 1; i <= n; ++i)
    if(!in[i])
      ready.push_back(i);

  int node;
  while(!ready.empty()) {
    node = ready.back(); ready.pop_back();
    ret.push_back(node);

    for(int i = Head[node]; i; i = Next[i])
      if(--in[To[i]] == 0)
  ready.push_back(To[i]);
  }
  return ret;
}

int main() {
  cin >> n >> m;
  while(m--) {
    cin >> u >> v;
    addEdge(u, v);
    ++in[v];
  }

  vector <int> v = kahn(n);
  if((int)v.size() == n) for(int i : v)
         cout << i << ' ';
  else
    cout << "not a DAG!" << endl;
}
```

2.31 Tree diameter

```
const int N = 3e5 + 9, M = 6e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Next[M], To[M], Par[N], toLeaf[N], maxLength[N], ne, n, m, u, v, w;

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  memset(Par, -1, sizeof(Par[0])  * (n + 2));
  ne = 0;
}

void dfs_toLeaf(int node, int par = -1)
{
  toLeaf[node] = 0;
  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != par) {
      dfs_toLeaf(To[i], node);
      if(toLeaf[To[i]] + 1 > toLeaf[node])
  toLeaf[node] = toLeaf[To[i]] + 1;
    }
}

void dfs_maxLength(int node, int par = -1)
{
  int firstMax = -1;
  int secondMax = -1;
  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != par) {
      dfs_maxLength(To[i], node);

      if(toLeaf[To[i]] > firstMax) {
  if(firstMax > secondMax)
    secondMax = firstMax;
  firstMax = toLeaf[To[i]];
      } else if(toLeaf[To[i]] > secondMax)
  secondMax = toLeaf[To[i]];
    }
  maxLength[node] = firstMax + secondMax + 2;
}

void Solve()
{
  cin >> n;
  _clear();

  for(int i = 1; i < n; ++i) {
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }

  dfs_toLeaf(1);
  dfs_maxLength(1);

  int diameter = 0;
  for(int i = 1; i <= n; ++i)
    if(maxLength[i] > diameter)
      diameter = maxLength[i];

  cout << diameter << endl;
}
```

2.32 0-1 BFS

```
const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void addEdge(int from, int to, int cost) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  ne = 0;
}

void BFS(int src, int trg) {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  deque <int> Q;
  Q.push_front(src);
  dis[src] = 0;

  int node;
  while(Q.size()) {
    node = Q.front(); Q.pop_front();
    if(node == trg) return;

    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]]) {
  dis[To[i]] = dis[node] + Cost[i];
  if(Cost[i])
    Q.push_back(To[i]);
  else
    Q.push_front(To[i]);
      }
  }
}
```

2.33 0-1 BFS (grid)

```
const int dr[]   = { -1, -1, 0, 1, 1,  1,  0, -1 };
const int dc[]   = {  0,  1, 1, 1, 0, -1, -1, -1 };
const char dir[] = {'D', 'U', 'R', 'L'};

const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;

int dis[N][N], n, m, si, sj, ti, tj;
char grid[N][N];

bool valid(int r, int c) {
  return r >= 1 && r <= n && c >= 1 && c <= m;
```

```cpp
}

int ZBFS(int sr, int sc, int tr, int tc) {
  memset(dis, 0x3f, sizeof (dis)); // memset(dis, 0x3f, n * m) we don't do that here

  deque <pair <int, int> > Q;

  dis[sr][sc] = 0;
  Q.push_front({sr, sc});

  int r, c, nr, nc, ncost;
  while(Q.size()) {
    tie(r, c) = Q.front(); Q.pop_front();
    if(r == tr && c == tc) return dis[r][c];

    for(int i = 0; i < 8; ++i) {
      nr = r + dr[i];
      nc = c + dc[i];

      if(!valid(nr, nc)) continue;
      ncost = (i != grid[r][c]);

      if(dis[r][c] + ncost < dis[nr][nc]) {
  dis[nr][nc] = dis[r][c] + ncost;

  if(ncost)
    Q.push_back({nr, nc});
  else
    Q.push_front({nr, nc});
      }
    }
  }
  return oo;
}
```

3 Data structures

3.1 Union find disjoint sets

```cpp
/**
   Maintain a set of elements partitioned into non-overlapping subsets. Each
   partition is assigned a unique representative known as the parent, or root. The
   following implements two well-known optimizations known as union-by-size and
   path compression. This version is simplified to only work on integer elements.

   - find_set(u) returns the unique representative of the partition containing u.
   - same_set(u, v) returns whether elements u and v belong to the same partition.
   - union_set(u, v) replaces the partitions containing u and v with a single new
   partition consisting of the union of elements in the original partitions.

   Time Complexity:
   - O(a(n)) per call to find_set(), same_set(), and union_set(), where n is the
   number of elements, and a(n) is the extremely slow growing inverse of the Ackermann function
   (effectively a very small constant for all practical values of n).

   Space Complexity:
   - O(n) for storage of the disjoint set forest elements.
   - O(1) auxiliary for all operations.
**/

class UnionFind {
  vector <int> par;
  vector <int> siz;
  int num_sets;
  size_t sz;

public:
  UnionFind() : par(1, -1), siz(1, 1), num_sets(0), sz(0) {}
  UnionFind(int n) : par(n + 1, -1), siz(n + 1, 1), num_sets(n), sz(n) {}

  int find_set(int u) {
    assert(u <= sz);

    int leader;
    for(leader = u; ~par[leader]; leader = par[leader]);

    for(int next = par[u]; u != leader; next = par[next]) {
      par[u] = leader;
      u = next;
    }
    return leader;
  }

  bool same_set(int u, int v) {
    return find_set(u) == find_set(v);
  }

  bool union_set(int u, int v) {
    if(same_set(u, v)) return false;

    int x = find_set(u);
    int y = find_set(v);

    if(siz[x] < siz[y]) swap(x, y);

    par[y] = x;
    siz[x] += siz[y];

    --num_sets;
    return true;
  }

  int number_of_sets() {
    return num_sets;
  }

  int size_of_set(int u) {
    return siz[find_set(u)];
  }

  size_t size() {
    return sz;
  }

  void clear() {
    par.clear();
    siz.clear();
    sz = num_sets = 0;
  }

  void assign(size_t n) {
    par.assign(n + 1, -1);
    siz.assign(n + 1,  1);
    sz = num_sets = n;
  }

  map < int, vector <int> > groups(int st) {
    map < int, vector <int> > ret;

    for(size_t i = st; i < sz + st; ++i)
      ret[find_set(i)].push_back(i);

    return ret;
  }
};
```

3.2 Segment tree (RMQ)

```cpp
/** resources:
    1. https://codeforces.com/blog/entry/15729
    2. https://codeforces.com/blog/entry/15890
    3. https://codeforces.com/blog/entry/18051
    4. https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/practice-problems/algorithm/range-minimum-query/description/
**/

template <class T, class F = function <T(const T &, const T &)> >
class SegmentTree {
  vector <T> _A;
  vector <T> ST;
  vector <T> LT;
  F func;
  int _N;

public :
  template <class iter>
  SegmentTree(iter _begin, iter _end, const F _func = [](T a, T b) {return a <= b ? a : b;}) : func(_func) {
    _N = distance(_begin, _end);
    _N = (1 << (int)ceil(log2(_N)));

    _A.assign(_N + 1,  0);
    ST.assign(_N << 1, 0);
    LT.assign(_N << 1, 0);

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; ++i, ++j)
      _A[j] = *i;

    build(1, 1, _N);
  }
```

```

  void build(int p, int l, int r) {
    if(l == r) {
      ST[p] = _A[l];
      return;
    }

    int mid = (l + r) >> 1;

    build(p + p, l, mid);
    build(p + p + 1, mid + 1, r);

    const T & x = ST[p + p];
    const T & y = ST[p + p + 1];

    ST[p] = func(x, y);
  }

  void update_range(int ul, int ur, int delta) {
    update_range(ul, ur, delta, 1, 1, _N);
  }

  T query(int ql, int qr) {
    return query(ql, qr, 1, 1, _N);
  }

  void update_point(int inx, int delta) {
    inx += _N - 1;
    ST[inx] = delta;

    while(inx > 1) {
      inx >>= 1;

      const T & x = ST[inx + inx];
      const T & y = ST[inx + inx + 1];

      ST[inx] = func(x, y);
    }
  }

private :
  void update_range(int ul, int ur, int delta, int p, int l, int r) {
    if(r < ul || ur < l)
      return;

    if(ul <= l && r <= ur) {
      ST[p] += delta;
      LT[p] += delta;
      return;
    }

    propagate(p);

    int mid = (l + r) >> 1;

    update_range(ul, ur, delta, p + p, l, mid);
    update_range(ul, ur, delta, p + p + 1, mid + 1, r);

    const T & x = ST[p + p];
    const T & y = ST[p + p + 1];

    ST[p] = func(x, y);
  }

  T query(int ql, int qr, int p, int l, int r) {
    if(r < ql || qr < l)
      return INT_MAX;

    if(ql <= l && r <= qr)
      return ST[p];

    propagate(p);

    int mid = (l + r) >> 1;

    const T & x = query(ql, qr, p + p, l, mid);
    const T & y = query(ql, qr, p + p + 1, mid + 1, r);

    return  func(x, y);
  }

  void propagate(int p) {
    if(LT[p]) {
      ST[p + p]     += LT[p];
      ST[p + p + 1] += LT[p];
      LT[p + p]     += LT[p];
      LT[p + p + 1] += LT[p];
      LT[p] = 0;
    }
  }
};
```

3.3 Merge sort tree

```
/** https://www.spoj.com/problems/KQUERY/
 **/

class SegmentTree {
  vector <vector <int> > sTree;
  vector <int> localArr;
  int NP2, oo = 0x3f3f3f3f;

public :
  template <class T>
  SegmentTree(T _begin, T _end) {
    NP2 = 1;
    int n = _end - _begin;
    while(NP2 < n) NP2 <<= 1;

    sTree.assign(NP2 << 1, vector <int> ());
    localArr.assign(NP2 + 1, 0);

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; i++, ++j)
      localArr[j] = *i;

    build(1, 1, NP2);
  }

  void build(int p, int l, int r) {
    if(l == r) {
      sTree[p].push_back(localArr[l]);
      return;
    }

    build(left(p),  l,     mid(l, r));
    build(right(p), mid(l, r) + 1, r);

    merge(p);
  }

  int query(int ql, int qr, int k) {
    return query(ql, qr, k, 1, 1, NP2);
  }

private :
  int query(int ql, int qr, int k, int p, int l, int r) {
    if(isOutside(ql, qr, l, r))
      return 0;

    if(isInside(ql, qr, l, r)) {
      return sTree[p].end() - upper_bound(sTree[p].begin(), sTree[p].end(), k);
    }

    return query(ql, qr, k, left(p),  l,     mid(l, r)) + query(ql, qr, k, right(p), mid(l, r) + 1, r);
  }

  void merge(int p) {
    vector <int> & L = sTree[left(p)];
    vector <int> & R = sTree[right(p)];

    int l_size = L.size();
    int r_size = R.size();
    int p_size = l_size + r_size;

    L.push_back(INT_MAX);
    R.push_back(INT_MAX);

    sTree[p].resize(p_size);

    for(int k = 0, i = 0, j = 0; k < p_size; ++k)
      if(L[i] <= R[j])
  sTree[p][k] = L[i], i += (L[i] != INT_MAX);
      else
  sTree[p][k] = R[j], j += (R[j] != INT_MAX);

    L.pop_back();
    R.pop_back();
  }

  inline bool isInside(int ql, int qr, int sl, int sr) {
    return (ql <= sl && sr <= qr);
  }

  inline bool isOutside(int ql, int qr, int sl, int sr) {
    return (sr < ql || qr < sl);
  }

  inline int mid (int l, int r) {
```

```
    return ((l + r) >> 1);
  }

  inline int left(int p) {
    return (p << 1);
  }

  inline int right(int p) {
    return ((p << 1) | 1);
  }
};
```

3.4 Sparse table (RMQ)

```
template <class T, class F = function <T(const T&, const T&)> >
class SparseTable {
  int _N;
  int _LOG;
  vector <T> _A;
  vector < vector <T> > ST;
  vector <int> Log;
  F func;

public :
  SparseTable() = default;

  template <class iter>
  SparseTable(iter _begin, iter _end, const F _func = less <T> ()) : func(_func) {
    _N = distance(_begin, _end);

    Log.assign(_N + 1, 0);
    for(int i = 2; i <= _N; ++i)
      Log[i] = Log[i >> 1] + 1;

    _LOG = Log[_N];

    _A.assign(_N + 1, 0);
    ST.assign(_N + 1, vector <T> (_LOG + 1, 0));

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; ++i, ++j)
      _A[j] = *i;

    build();
  }

  void build() {
    for(int i = 1; i <= _N; ++i)
      ST[i][0] = i;

    for(int j = 1, k, d; j <= _LOG; ++j) {  // the two nested loops below have overall time complexity
          = O(n log n)
      k = (1 << j);
      d = (k >> 1);

      for(int i = 1; i + k - 1 <= _N; ++i) {
  T const & x = ST[i][j - 1];     // starting subarray at index = i     with length = 2^{j - 1}
  T const & y = ST[i + d][j - 1]; // starting subarray at index = i + d with length = 2^{j - 1}

  ST[i][j] = func(_A[x], _A[y]) ? x : y;
      }
    }
  }

  T query(int l, int r) { // this query is O(1)
    int d = r - l + 1;
    T const & x = ST[l][Log[d]];
    T const & y = ST[l + d - (1 << Log[d])][Log[d]];

    return func(_A[x], _A[y]) ? x : y;
  }
};
```

3.5 Sparse table (RSQ)

```
template <class T, class F = function <T(const T &, const T &)> >
class SparseTable {
  int _N;
  int _LOG;
  vector <T> _A;
  vector < vector <T> > ST;
  vector <int> Log;
  F func;

public :
  SparseTable() = default;

  template <class iter>
  SparseTable(iter _begin, iter _end, F _func = [](T a, T b) { return a + b; }) : func(_func) {
    _N = distance(_begin, _end);

    Log.assign(_N + 1, 0);
    for(int i = 2; i <= _N; ++i)
      Log[i] = Log[i >> 1] + 1;

    _LOG = Log[_N];

    _A.assign(_N + 1, 0);
    ST.assign(_N + 1, vector <T> (_LOG + 1, 0));

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; ++i, ++j)
      _A[j] = *i;

    build();
  }

  void build() {
    for(int i = 1; i <= _N; ++i)
      ST[i][0] = _A[i];

    for(int j = 1, k, d; j <= _LOG; ++j) {
      k = (1 << j);
      d = (k >> 1);

      for(int i = 1; i + k - 1 <= _N; ++i) {
  T const & x = ST[i][j - 1];     // starting subarray at index = i     with length = 2^{j - 1}
  T const & y = ST[i + d][j - 1]; // starting subarray at index = i + d with length = 2^{j - 1}

  ST[i][j] = func(x, y);
      }
    }
  }

  T query(int l, int r) {
    int d = r - l + 1;
    T ret = 0;

    for(int i = l; d; i += lastBit(d), d -= lastBit(d))
      ret = func(ret, ST[i][Log[lastBit(d)]]);

    return ret;
  }

  int lastBit(int a) {
    return (a & -a);
  }
};
```

3.6 Merge sort

```
/**
   Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time
        complexity can be expressed as following recurrence relation.

  T(n) = 2T(n/2) + theta(n)

   The above recurrence can be solved either using the Recurrence Tree method or the Master method. It
        falls in case II of Master Method and the solution of the recurrence is theta(nLogn). Time
       complexity of Merge Sort is  theta(nLogn) in all 3 cases (worst, average and best) as merge
       sort always divides the array into two halves and takes linear time to merge two halves.
   Auxiliary Space: O(n)
   Algorithmic Paradigm: Divide and Conquer
   Sorting In Place: No in a typical implementation
   Count Inversion of array: yes
   https://discuss.codechef.com/t/iiti15-editorial/4427
   Stable: Yes
**/

ll inversions;

template <class T>
void merge(T localArr [], int l, int mid, int r) {
  int l_size = mid - l + 1;
  int r_size = r - mid;

  T L[l_size + 1];
  T R[r_size + 1];
```

```cpp
  for(int i = 0; i < l_size; ++i) L[i] = localArr[i + l];
  for(int i = 0; i < r_size; ++i) R[i] = localArr[i + mid + 1];

  T Mx;
  if(sizeof(T) == 4) Mx = INT_MAX;
  else Mx = LONG_MAX;

  L[l_size] = R[r_size] = Mx;

  for(int k = l, i = 0, j = 0; k <= r; ++k)
    if(L[i] <= R[j])
      localArr[k] = L[i], i += (L[i] != Mx);
    else
      localArr[k] = R[j], j += (R[j] != Mx), inversions += l_size - i;
}

template <class T>
void merge_sort(T localArr [], int l, int r) {
  if(r - l)
    {
      int mid = (l + r) >> 1;
      merge_sort(localArr, l,     mid);
      merge_sort(localArr, mid + 1, r);
      merge(localArr,      l,  mid, r);
    }
}

template <class T>
void merge_sort(T _begin, T _end) {
  const int sz = _end - _begin;
  __typeof(*_begin) localArray[sz];

  __typeof(_begin) k = _begin;
  for(int i = 0; k != _end; ++i, ++k)
    localArray[i] = *k;

  merge_sort(localArray, 0, sz - 1);

  k = _begin;
  for(int i = 0; k != _end; ++i, ++k)
    *k = localArray[i];
}
```

3.7 Selection sort

```cpp
template <class T>
void selection_sort(T _begin, T _end, int round) {
  const int sz = _end - _begin;
  int localArray[sz];

  __typeof(_begin) k = _begin;
  for(int i = 0; k != _end; ++i, ++k)
    localArray[i] = *k;

  int MnInx;
  round = min(sz, round);
  for(int i = 0; i < round; ++i) {
    MnInx = i;
    for(int j = i + 1; j < sz; ++j)
      if(localArray[j] < localArray[MnInx])
  MnInx = j;
    swap(localArray[MnInx], localArray[i]);
  }

  k = _begin;
  for(int i = 0; k != _end; ++i, ++k)
    *k = localArray[i];
}
```

3.8 Bubble sort

```cpp
/**
   Bubble sort consists of n rounds. On each round, the algorithm iterates
   through the elements of the array. Whenever two consecutive elements are found
   that are not in correct order, the algorithm swaps them. The algorithm can be
   implemented as follows:
**/

template <class T>
void bubble_sort(T _begin, T _end, int round) {
  const int sz = _end - _begin;
  int localArray[sz];

  __typeof(_begin) k = _begin;
  for(int j = 0; k != _end; ++k, ++j)
    localArray[j] = *k;

  round = min(round, sz);
  for(int i = 0; i < round; ++i) /* n rounds -> n_th element **/
    for(int j = 0; j < sz - 1; ++j) if(localArray[j] > localArray[j + 1])
          swap(localArray[j], localArray[j + 1]);

  k = _begin;
  for(int j = 0; k != _end; ++k, ++j)
    *k = localArray[j];
}

/**
   After the first round of the algorithm, the largest element will be in the correct
   position, and in general, after k rounds, the k largest elements will be in the
   correct positions.
**/
```

4 Mathematics

4.1 Euler totient function

```cpp
/**
    Constraints:
    1 <= n <= 1e7
    2 <= a <= 10^{14}

    Time Complexity:
    linear_sieve takes O(n)
    Phi takes O(n / (ln(n) - 1.08))

    Space Complexity:
    O(MaxN + n / (ln(n) - 1.08))

    Explanation:
    Phi(n) = n * ((p1 - 1) / p1) * ((p2 - 1) / p2) *...* ((pk - 1) / pk)
    Phi(n) = n * (1 - (1 / p1)) * (1 - (1 / p2)) *...* (1 - (1 / pk))

    Applications:
    Eulers theorem:
    a^phi(m) cong 1 (mod m) if a and m are relatively prime.

    Fermats little theorem:
    when m is a prime:
    a^{m minus 1} cong 1 (mod m)

    As immediate consequence we also get the equivalence:
    a^n cong a^{n mod phi(m)} (mod m)
    This allows computing x^n mod m for very big n, especially if n is the result of another
          computation,
    as it allows to compute n under a modulo.
**/

int lp[N], Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

ll Phi(ll a) { // for Queries
  ll ret = a, p;
  for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
    if (p * p > a) break;
    if (a % p) continue;
    ret -= ret / p;
    while (a % p == 0) a /= p;
  }
  if (a > 1) ret -= ret / a;
  return ret;
}
```

4.2 Euler phi sieve

```
/*
  Constraints:
  1 <= n <= 1e7

  Time Complexity:
  Phi_sieve takes O(n * ln(ln(n)))

  Space Complexity:
  MaxN
*/

int EulerPhi[N];

void Phi_sieve(int n) {
  for (int i = 1; i <= n; ++i) {
    EulerPhi[i] = i;
  }
  for (int i = 2; i <= n; ++i) {
    if (EulerPhi[i] == i)
      for (int j = i; j <= n; j += i) {
  EulerPhi[j] -= EulerPhi[j] / i;
      }
  }
}
```

4.3 Extended wheel factorization

```
/*
  Constraints:
  1 <= n <= 1e7
  2 <= a <= 1e{14}

  Time Complexity:
  linear_sieve takes O(n)
  Factorization takes O(n / (ln(n) - 1.08))

  Space Complexity:
  O(MaxN + n / (ln(n) - 1.08)
*/

int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

vector<pair<ll, int>> Factorization(ll a) {
  vector<pair<ll, int> > ret;
  ll p;
  for (int i = 0, cnt; i < pnx && (p = Primes[i], true) && p * p <= a; ++i) {
    if (a % p) continue;
    cnt = 0;
    while (a % p == 0) a /= p, ++cnt;
    ret.emplace_back(p, cnt);
  }
  if (a > 1) ret.emplace_back(a, 1);
  return ret;
}
```

4.4 Least prime factorization

```
/*
  Constraints:
  1 <= n <= 1e7

  Time Complexity:
  linear_sieve takes O(n)
  Factorization takes O(log(n))

  Space Complexity:
  O(MaxN + n / (ln(n) - 1.08)
*/

int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

vector<pair<int, int>> Factorization(int n) {
  vector<pair<int, int>> ret;
  while (n > 1) {
    int p = leastPrime[n], cnt = 0;
    while (n % p == 0) n /= p, ++cnt;
    ret.emplace_back(p, cnt);
  }
  return ret;
}
```

4.5 Linear sieve

```
/**
    Constraints:
    1 <= n <= 1e7

    Time Complexity:
    linear_sieve takes O(n)

    Space Complexity:
    O(MaxN + n / (ln(n) - 1.08))
**/

int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}
```

4.6 Miller Rabin test

```
ll ModExp(ll base, ll e, ll mod) {
  ll result;
  base %= mod;

  for(result = 1; e; e >>= 1ll) {
    if(e & 1ll)
      result = ((i128)result * base) % mod;
    base = ((i128)base * base) % mod;
  }
  return result;
}

bool CheckComposite(ll n, ll p, ll d, int r) {
  ll a = ModExp(p, d, n);
  if(a == 1 || a == n - 1)
    return false;

  for(int i = 1; i < r; ++i) {
    a = ((i128)a * a) % n;
    if(a == n - 1)
      return false;
  }
  return true;
}

bool Miller(ll n) {
```

```cpp
  if(n < 2) return false;

  int r; ll d;
  for(r = 0, d = n - 1; (d & 1ll) == 0; d >>= 1ll, ++r);

  for(int p : {2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
    if(n == p)
      return true;
    if(CheckComposite(n, p, d, r))
      return false;
  }
  return true;
}
```

4.7 Mobius function

```cpp
/**
    Constraints:
    1 <= x <= 1e7
    2 <= n <= 10^{14}

    Time Complexity:
    linear_sieve takes O(x)
    mobius takes O(n / (ln(n) - 1.08))

    Space Complexity:
    O(MaxN + n / (ln(n) - 1.08))
*/

int lp[N], Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int x) {
  for (int i = 2; i <= x; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }

    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= x; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

int mobius(ll n) {
  ll p, pp;
  char mob = 1;
  for (int i = 0; i < pnx && (p = Primes[i], pp = p * p, true); ++i) {
    if (pp > n) break;
    if (n % p) continue;
    if (n % pp == 0) return 0;
    n /= p;
    mob = -mob;
  }
  if (n > 1) mob = -mob;
  return mob;
}
```

4.8 Mobius sieve

```cpp
/*
  Constraints:
  1 <= n <= 1e7

  Time Complexity:
  mu_sieve takes O(n)

  Space Complexity:
  O(MaxN)
*/

int mu[N], lp[N], Primes[78522], pnx;

void mu_sieve(int n) {
  mu[1] = 1;
  fill(mu, mu + N, 1);
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
      mu[i] = -1;
    }
    for (int j = 0, nxt; j < pnx && Primes[j] <= lp[i] && (nxt = i * Primes[j]) <= n; ++j) {
      lp[nxt] = Primes[j];
      mu[nxt] = (lp[i] == Primes[j] ? 0 : -mu[i]);
    }
  }
}
```

4.9 Phi factorial

```cpp
/**
    Constraints:
    1 <= x <= 1e7
    2 <= n <= 1e7

    Time Complexity:
    linear_sieve takes O(x)
    phi_factorial takes O(n)

    Space Complexity:
    O(MaxN + n / (ln(n) - 1.08))
*/

int lp[N], Primes[664580], pnx; /** number of primes = n / (ln(n) - 1.08) **/

void linear_sieve(int x) {
  for (int i = 2; i <= x; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }

    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= x; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

ll phi_factorial(int n) {
  ll ret = 1;
  for (int i = 2; i <= n; ++i) {
    ret = ret * (lp[i] == i ? i - 1 : i);
  }
  return ret;
}
```

4.10 Pisano periodic sequence

```cpp
/** Algorithm 1: **/

/** Constraints :
    1 <= n <= CR
    Where CR stands for your computing resources. In my case,
    CR = 100,000,000
    ----------------
    The algorithm is constructed around the ideas that a Pisano sequence always starts with 0 and 1, and that this sequence of Fibonacci
    numbers taken modulo n can be constructed for each number by adding the previous remainders and taking into account the modulo n.
    ----------------

    definition:
    The sequence of Fibonacci numbers {F_n} is periodic modulo any modulus m (Wall 1960), and the period (mod m)
    is the known as the Pisano period pi(m) (Wrench 1969). For m=1, 2, ..., the values of pi(m) are
    1, 3, 8, 6, 20, 24, 16, 12, 24, 60, 10, ... (OEIS A001175).

    Since pi(10)=60, the last digit of F_n repeats with period 60, as first noted by Lagrange in 1774 (Livio 2002, p. 105).
    The last two digits repeat with a period of 300, and the last three with a period of 1500.
    In 1963, Geller found that the last four digits have a period of 15000 and the last five a period of 150000.
    Jarden subsequently showed that for d>=3, the last d digits have a period of 15*10^(d minus 1) ( Livio 2002, pp. 105-106).
    The sequence of Pisano periods for n=1, 10, 100, 1000, ... are therefore 60, 300, 1500, 15000, 150000, 1500000, ... (OEIS A096363).
    pi(m) is even if m>2 (Wall 1960). pi(m)=m iff m=24*5^(k-1) for some integer k>1 (Fulton and Morris 1969, Wrench 1969).

    resources :
    1. https://webbox.lafayette.edu/~reiterc/nt/qr_fib_ec_preprint.pdf
    2. https://www.youtube.com/watch?v=Nu-lW-Ifyec&ab_channel=Numberphile
    4. http://webspace.ship.edu/msrenault/fibonacci/fib.htm
    5. https://www.theoremoftheday.org/Binomial/PeriodicFib/TotDPeriodic.pdf
    7. http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibmaths.html#fibmod
    8. http://webspace.ship.edu/msrenault/fibonacci/FibThesis.pdf
```

```cpp
    9. https://www.fq.math.ca/Scanned/1-2/vinson.pdf
**/

vector <int> pisano_periodic_sequence(int n) {
  vector <int> period;

  int current = 0, next = 1;
  period.push_back(current);

  if(n < 2) return period;
  current = (next += current) - current;

  while(current != 0 || next != 1) {
    period.push_back(current);
    current = current + next >= n ? (next += current - n) + (n - current) : (next += current) - current;
  }
  return period;
}

/** ***************************************************************************************** **/
/** Algorithm 2: **/

/** constraints:
  1 <= n <= 10^{18}
  --------------
  problem statement (https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=4479):
  For any integer n, the sequence of Fibonacci numbers F_i taken mod n is periodic.
  define K(n) = the length of the period of the Fibonacci sequence reduced mod n.
  The task is to print the length of the period of this sequence K(n).
  --------------
  Definition:
  The Pisano period  pi(n)  is a multiplicative function, that is if  a  and  b  are coprime than  pi(ab) = pi(a) * pi(b).
  So we need only concern ourselves with the value of  pi(p^k) for prime  p.
  (Factoring even a large number is still better than brute force periodicity search.)

  It is hypothesized that  p(p^k) = p^{k - 1} * pi(p) and since no counterexamples are known to exist,
  you might as well use that in your algorithm.

  So, how to calculate  pi(p)  efficiently? There are two special cases and two general cases

  pi(2^k) = 3*2^{k - 1}
  pi(5^k) = 4*5^k

  If  p cong 1  or  p cong 9 (mod10)  then  pi(p) | p - 1
  If  p cong 3  or  p cong 7 (mod10)  then  pi(p) | 2 * (p + 1) , and by an odd divisor too.

  The last two statements give us a relatively small number of cases to try (after factoring  p - 1  or  2*(p + 1) .)
  Now use your favorite formula to calculate large values of the Fibonacci numbers  F(x) (mod p).
  [See Michal answer to What is a fast algorithm to find the remainder of the division of a huge Fibonacci number by some big integer?](https://www.quora.com/Whats-a-fast-algorithm-to-find-the-remainder-of-the-division-of-a-huge-Fibonacci-number-by-some-big-integer/answer/Michal-Fori%C5%A1ek) .
  To test a candidate period  R , calculate  F(R) (mod p)  and  F(R + 1) (mod p).
  If these are equal to  F(0) = 0  and  F(1) = 1 , then  pi(p) | R.

  It might be that  p - 1  or  2*(p + 1)  have a lot of divisors, but we dont need to try them all.
  Suppose  q^k | R  for some prime q.
  Then test  R/q . If that doesnt produce a cycle, then  pi(p)  must have factor  q^k ,
  and we can leave it in and go on to other factors.
  Otherwise, we can use  R/q  as our new starting point and repeat the process.
  Thus we have to do a number of checks proportional to  Omega(2*(p + 1)) , not  d(2*(p + 1)).

  ------------
  Donald Wall proved several other properties, some of which you may find interesting:
   If m > 2, k(m) is even.
   For any even integer n > 2, there exists m such that k(m) = n.
   k(m) <= (m^2) - 1
   k(2^n) = 3 * 2^{n - 1}
   k(5^n) = 4 * 5^n
   If n > 2, k(10^n) = 15 * 10^{n - 1}
   k(2 * 5^n) = 6n
**/

#pragma  GCC optimize ("Ofast")

#include <bits/stdc++.h>

#define endl        '\n'

using namespace std;

typedef long long     ll;
typedef __int128    i128;
typedef __int128_t ui128;

template <class T>
using matrix = vector < vector <T> >;

template <class T> string to_string(T x) {
  int sn = 1;
  if(x < 0) sn = -1, x *= sn;
  string s = "";
  do {
    s = "0123456789"[x % 10] + s, x /= 10;
  } while(x);
  return (sn == -1 ? "-" : "") + s;
}

auto str_to_int(string x) {
  ui128 ret = (x[0] == '-' ? 0 : x[0] - '0');
  for(int i = 1; i < (int)x.size(); ++i) ret = ret * 10 + (x[i] - '0');
  return (x[0] == '-' ? -1 * (i128)ret : ret);
}

istream & operator >> (istream & in, i128 & i) noexcept {
  string s;
  in >> s;
  i = str_to_int(s);
  return in;
}

ostream & operator << (ostream & os, const i128 i) noexcept {
  os << to_string(i);
  return os;
}

void Fast() {
  cin.sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);
}

ll n;
vector <int> primes;
matrix <ll> fibMatrix = {{1, 1},
       {1, 0}
};

i128 gcd(i128 a, i128 b) {
  while (a && b)
    a > b ? a %= b : b %= a;
  return a + b;
}

i128 lcm(i128 a, i128 b) {
  return a / gcd(a, b) * b;
}

vector < array <ll, 2> > factorize(ll x) {
  vector < array <ll, 2> > ret;
  for(int i = 0; 1ll * primes[i] * primes[i] <= x; ++i) {
    if(x % primes[i]) continue;

    int cnt = 0;
    while (x % primes[i] == 0) {
      cnt++;
      x /= primes[i];
    }
    ret.push_back({primes[i], cnt});
  }

  if(x > 1) ret.push_back({x, 1});
  return ret;
}

matrix <ll> MatMul(matrix <ll> A, matrix <ll> B, ll mod) {
  int ra = A.size(), cb = B[0].size(), ca = A[0].size();
  matrix <i128> C(ra, vector <i128> (cb));

  for(int i = 0; i < ra; ++i)
    for (int j = 0; j < cb; ++j) {
      C[i][j] = 0;
      for(int k = 0; k < ca; ++k)
  C[i][j] = (C[i][j] + (i128)A[i][k] * B[k][j]);
    }

  matrix <ll> ret(ra, vector <ll> (cb));
  for(int i = 0; i < ra; ++i)
    for (int j = 0; j < cb; ++j)
      ret[i][j] = C[i][j] % mod;

  return ret;
}

matrix <ll> MatPow(matrix <ll> A, ll p, ll mod) {
  int r = A.size(), c = A[0].size();
  assert(r == c && p);
  matrix <ll> result = A;
  p--;

```

```cpp
  while(p) {
    if(p & 1ll) result = MatMul(result, A, mod);
    A = MatMul(A, A, mod);
    p >>= 1ll;
  }
  return result;
}

i128 ModExp(i128 a, ll p) {
  i128 result = 1;
  while(p) {
    if(p & 1ll) result = result * a;
    a *= a;
    p >>= 1ll;
  }
  return result;
}

ll nthFib(ll n, ll mod) {
  return MatPow(fibMatrix, n, mod)[0][1];
}

bool is_period(ll n, ll mod) {
  return nthFib(n, mod) == 0 && nthFib(n + 1, mod) == 1;
}

ll solver(ll x, ll mod) {
  vector < array <ll, 2> > factors = factorize(x);
  for(int i = 0; i < (int)factors.size(); ++i) {
    while(x % factors[i][0] == 0 && is_period(x / factors[i][0], mod))
      x /= factors[i][0];
  }
  return x;
}

ll pisano_prime(ll val) {
  if(val == 2) return 3;
  if(val == 5) return 20;
  if(val % 10 == 1 || val % 10 == 9)
    return solver(val - 1, val);

  return solver(2 * (val + 1), val);
}

const int N = 1e7 + 9;
bitset <N> isPrime;

void Precomputation_Sieve() {
  isPrime.set();
  int _sqrt = sqrtl(N);

  for(int i = 5; i <= _sqrt; i += 6) {
    if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
    i += 2;
    if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
    i -= 2;
  }
}

vector <int> Primes(int n) {
  vector <int> _Primes;

  if(n >= 2) _Primes.push_back(2);
  if(n >= 3) _Primes.push_back(3);

  for (int i = 5; i <= n; i += 6) {
    if(isPrime[i]) _Primes.push_back(i);
    i += 2;
    if(isPrime[i]) _Primes.push_back(i);
    i -= 2;
  }
  return _Primes;
}

void initialize()
{
  Precomputation_Sieve();
  primes = Primes(N);
}

void Solve() {
  initialize();
  cin >> n;
  vector < array <ll, 2> > factors = factorize(n);

  i128 ans = 1;
  for (int i = 0; i < (int)factors.size(); ++i) {
    ans = lcm(ans, (i128)pisano_prime(factors[i][0]) * ModExp(factors[i][0], factors[i][1] - 1));
  }
  cout << ans << endl;
}
```

4.11 Segmented sieve

```cpp
/**
   constraints:
   1 <= l, r <= 1e{14}
   1 <= r - l + 1 <= 1e7
   2 <= x <= 1e7

   Time complexity:
   segmented_sieve takes O((r - l + 1) * ln(ln(r)))
   linear_sieve takes O(n)

   Space Complexity:
   O(2 * MaxN + n / (ln(n) - 1.08))
**/

int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
bool isPrime[N];

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

vector<ll> segmented_sieve(ll l, ll r) {
  l += l == 1;
  int limit = r - l + 1;
  vector<ll> ret;
  memset(isPrime, true, sizeof(isPrime));

  ll p;
  for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
    for (ll j = max(p * p, (l + p - 1) / p * p); j <= r; j += p)
      isPrime[j - l] = false;
  }

  for (int i = 0; i < limit; ++i)
    if (isPrime[i])
      ret.emplace_back(i + l);
  return ret;
}
```

4.12 Simple sieve

```cpp
/**
   constraints:
   1 <= n <= 1e8

   Time complexity:
   O(n * ln(ln(sqrt(n))) + n)

   Space Complexity:
   O(MaxN +  n / (ln(n) - 1.08))
**/

bool isPrime[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void sieve(int x) {
  int basis[3] = {2, 3, 5};
  int wheel[8] = {7, 11, 13, 17, 19, 23, 29, 1};
  int inc[8] = {4, 2, 4, 2, 4, 6, 2, 6};
  int inx[31];

  memset(inx, 0, sizeof(inx));
  memset(isPrime, true, sizeof(isPrime));

  for (int p : basis) if (x > p) Primes[pnx++] = p;
  for (int i = 0; i < 8; ++i) inx[wheel[i]] = i;

  int c = 0;
  for (int i = 7; i <= x; i += inc[c++]) {
    if (isPrime[i]) {
      Primes[pnx++] = i;
      int d = inx[i % 30];
      for (ll j = i * 1ll * i; j <= x; j += i * inc[d++]) {
  isPrime[j] = false;
```

```
  if (d == 8) d = 0;
      }
    }
    if (c == 8) c = 0;
  }
}
```

4.13 The stable marriage problem

```
const int N = 1e3+ 9, M = 1e3 + 9, oo = 0x3f3f3f3f;
queue <int> Q;

int husband[N], wife[N], Next[N], order[N][N], pref[N][N], n, v;

void _clear() {
  memset(wife,    0, sizeof(wife[0])    * (n + 2));
  memset(husband, 0, sizeof(husband[0]) * (n + 2));
  memset(Next,    0, sizeof(Next[0])    * (n + 2));
}

void engage(int man, int woman) {
  int exWife = wife[man];
  wife[man] = woman;
  husband[woman] = man;

  if(exWife)
    Q.push(exWife);
}

void Solve() {
  _clear();
  cin >> n;

  for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= n; ++j)
      cin >> pref[i][j];

  for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= n; ++j) {
      cin >> v;
      order[i][v] = j;
    }

  for(int i = 1; i <= n; ++i)
    Q.push(i);

  int man, woman;
  while(Q.size()) {
    woman = Q.front(); Q.pop();
    man = pref[woman][++Next[woman]];

    if(!wife[man] || order[man][woman] < order[man][wife[man]])
      engage(man, woman);
    else
      Q.push(woman);
  }

  for(int i = 1; i <= n; ++i)
    cout << husband[i] << endl;
}
```

4.14 Wheel sieve

```
/**
    Constraints:
    1 <= n <= 1e9
    2 <= x <= 9700000

    Time Complexity:
    wheel_sieve takes O(n / ln(ln(n)))
    coPrimes takes O(x * ln(ln(x)))

    Space Complexity:
    O(MaxN / 32 + n / (ln(n) - 1.08) + x)
**/

bitset<N> isPrime;
int inx[30100];
int Primes[50908031], pnx; /** size of Primes = n / (ln(n) - 1.08) */

vector<int> coPrimes(int x) {
  int basis[5] = {3, 5, 7, 11, 13};

  vector<int> ret;
  bitset<30100> isCoprime;
  isCoprime.set();

  for (int b : basis)
    for (int d = b * b; d <= x; d += b << 1)
      isCoprime.reset(d);

  for (int i = 17; i <= x; i += 2)
    if (isCoprime[i]) ret.push_back(i);

  ret.push_back(x + 1);
  ret.push_back(x + 17);
  return ret;
}

void wheel_sieve(int n) {
  int basis[6] = {2, 3, 5, 7, 11, 13};
  vector<int> wheel = coPrimes(2 * 3 * 5 * 7 * 11 * 13);
  int sz = wheel.size();

  for (int k = 0; k < sz; ++k)
    inx[wheel[k]] = k;

  isPrime.set();
  inx[1] = sz - 2;
  int inc[sz - 1];

  for (int i = 1; i < sz; ++i)
    inc[i - 1] = wheel[i] - wheel[i - 1];

  for (int p : basis) {
    if (n >= p)
      Primes[pnx++] = p;
  }

  int c = 0;
  for (ll i = 17; i <= n; i += inc[c++]) {
    if (isPrime[i]) {
      Primes[pnx++] = i;
      int d = inx[i % 30030];
      for (ll j = i * i; j <= n; j += i * inc[d++]) {
  isPrime.reset(j);
  if (d == sz - 1) d = 0;
      }
    }
    if (c == sz - 1) c = 0;
  }
}
```

5 String Processing

5.1 Trie

```
class Trie {
private:
  Trie* children[26]; // Pointer = 8 Byte; 8*26 = 208 Byte
  int prefixs, words; // 8 Byte
  bool iseow; // 1 Byte
  char cur_letter; // 1 Byte
  vector <string> lex;
  priority_queue <pair <int, string>, vector <pair <int, string>>, greater <pair <int, string>>>
        occurrence; // small at top

public:
  Trie(char lett = '\0') {
    memset(children, 0, sizeof(children));
    prefixs = words = 0;
    iseow = false;
    cur_letter = lett;
  }

  void insert(string &str) { // O(l)
    Trie* cur = this;
    int inx, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(cur->children[inx] == nullptr)
  cur->children[inx] = new Trie(str[i]);

      cur = cur->children[inx];
      cur->prefixs++;
    }
```

```cpp
    cur->iseow = true;
    cur->words++;
  }

  int search_word(string &str) { // O(1)
    Trie* cur = this;
    int inx, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(cur->children[inx] == nullptr) {
  return 0;
      }
      cur = cur->children[inx];
    }
    return cur->words;
  }

  int search_prefix(string &str) { // O(1)
    Trie* cur = this;
    int inx = 0, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(cur->children[inx] == nullptr) {
  return 0;
      }
      cur = cur->children[inx];
    }
    return cur->prefixs;
  }

  bool erase(string &str) {
    if(!search_word(str))
      return false;

    Trie* cur = this;
    int inx, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(--cur->children[inx]->prefixs == 0) {
  cur->children[inx] = nullptr;
  return true;
      }
      cur = cur->children[inx];
    }
    if(--cur->words == 0) {
      cur->iseow = false;
    }
    return true;
  }

private:
  void dfs(Trie* node, string s) { // lex order dfs -> traverse all the strings starting from root
    if(node->iseow) {
      lex.emplace_back(s);
    }

    for(int j = 0; j < 26; ++j)
      if(node->children[j] != nullptr) {
  dfs(node->children[j], s + string(1, node->children[j]->cur_letter));
      }
  }

  void dfs2(Trie* node, string s) { // autocomplete dfs -> traverse all the strings starting from the
        end of the given prefix
    if(node->iseow) {
      if(occurrence.size() < 10) {
  occurrence.push(make_pair(node->words, s));
      } else {
  if(node->words > occurrence.top().first) {
    occurrence.pop();
    occurrence.push(make_pair(node->words, s));
  }
      }
    }

    for(int i = 0; i < 26; ++i) if(node->children[i] != nullptr) {
  dfs2(node->children[i], s + string(1, node->children[i]->cur_letter));
      }
  }

public:
  vector <string> lex_order() { // all strings in lexicographical order
    lex.clear();
    Trie* cur = this;
    for(int i = 0; i < 26; ++i) if(cur->children[i] != nullptr) {
  dfs(cur->children[i], string(1, cur->children[i]->cur_letter));
      }
    return lex;
  }

  void autocomplete(string &pref) { // suggest top ten words with max frequency
    if(!search_prefix(pref))
      return;

    Trie* cur = this;
    int inx, presz = pref.size();
    for(int i = 0; i < presz; ++i) {
      inx = pref[i] - 'a';
      cur = cur->children[inx];
    }

    for(int i = 0; i < 26; ++i) if(cur->children[i] != nullptr) {
  dfs2(cur->children[i], string(1, cur->children[i]->cur_letter));
      }

    vector <string> st;
    while(!occurrence.empty()) {
      st.emplace_back(pref + occurrence.top().second);
      occurrence.pop();
    }
    if(cur->iseow) {
      st.emplace_back(pref);
    }
    while(!st.empty()) {
      cout << st.back() << endl;
      st.pop_back();
    }
  }
};
```

5.2 Knuth Morris Pratt (KMP)

```cpp
/**
 * KMP(Knuth-Morris-Pratt) Algorithm
 ** Longest Prefix
 *** proper prefix = all prefixes except the whole string
 *** propre suffix = all suffixes except the whole string
 ** Prefix Function = Failure Function
 *** Given String P of len m, Find F[m];
 *** let t = P[0....i]
 *** f[i] = length of the longest proper prefix of t that is suffix of t
 *** calculating i different ways
 *** match the pattern against itself
 *** O(m) for failure function
 *** O(n) for KMP
 **/

vector <int> LongestPrefix(string &p) {
  int psz = p.size();
  vector <int> longest_prefix(psz, 0);

  for(int i = 1, k = 0; i < psz; ++i) {
    while(k && p[k] != p[i]) k = longest_prefix[k - 1];
    longest_prefix[i] = (p[k] == p[i] ? ++k : k);
  }
  return longest_prefix;
}

vector <int> KMP(string &s, string &p) {
  int ssz = s.size(), psz = p.size();

  vector <int> longest_prefix = LongestPrefix(p), matches;

  for(int i = 0, k = 0; i < ssz; ++i) {
    while(k && p[k] != s[i]) k = longest_prefix[k - 1]; // Fail go back
    k += (p[k] == s[i]);

    if(k == psz) {
      matches.emplace_back(i - psz + 1);
      k = longest_prefix[k - 1]; // faill safe and find another pattern
    }
  }
  return matches;
}
```

6 Geometry

6.1 Point

```cpp
/**
   notes:
```

```
  EPS = 1e-9
  -----------------------------------
  Integers   |       Doubles        |
  -----------------------------------
  a == b    | fabs(a - b) < EPS     |
  a <= b    | a < b + EPS           |
  a >= b    | a + EPS > b           |
  a < b     | a + EPS < b           |
  a > b     | a > b + EPS           |
  x >= 0.0   | x > -EPS             |
  x <= 0.0   | x < EPS              |
  ----------------------------------
**/

class point {
public :
  ld x, y;

  point() = default;
  point(ld _x, ld _y) : x(_x), y(_y) {}

  bool operator < (point other) const {
    if(fabs(x - other.x) > EPS) // if(x != other.x)
      return x < other.x;
    return y < other.y;
  }

  bool operator == (point other) const {
    return ((fabs(x - other.x) < EPS) && (fabs(y - other.y) < EPS)); // " < EPS " equal to " == zero "
  }

  bool operator > (point other) const {
    if(fabs(x - other.x) > EPS)
      return x > other.x;
    return y > other.y;
  }

  ld dist(point other) { // Euclidean distance
    ld dx = this->x - other.x;
    ld dy = this->y - other.y;
    return sqrtl(dx * dx + dy * dy);
  }

  ld DEG_to_RAD(ld theta) {
    return theta * PI / 180.0;
  }

  ld RAD_to_DEG(ld theta) {
    return theta * 180.0 / PI;
  }

  point rotate(ld theta) {
    ld rad = DEG_to_RAD(theta);
    return point(cos(theta) * x - sin(theta) * y,
     sin(theta) * x + cos(theta) * y);
  }
};
```

7 More advanced topics

7.1 A* Algorithm

```
const int dr []     = {-1, 0, 1, 0};
const int dc []     = {0, 1, 0, -1};
const char dir []   = {'U', 'R', 'D', 'L'};
map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};

const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
const ll INF = 0x3f3f3f3f3f3f3f3f;

char grid[N][N], Par[N][N];
int dis[N][N], n, m, si, sj, ti, tj;

vector < pair <int, int> > restorePath(int sr, int sc, int tr, int tc) {
  vector < pair <int, int> > ret;
  if(dis[tr][tc] == oo) return ret;

  for(char i = Par[tr][tc]; (sr ^ tr) || (sc ^ tc); i = Par[tr][tc]) {
    ret.push_back({tr, tc});
    tr += dr[inv[i]];
    tc += dc[inv[i]];
  }

  ret.push_back({sr, sc});
  reverse(ret.begin(), ret.end());
  return ret;
}

bool valid(int r, int c) {
  return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] != '%';
}

/** admissible heuristic **/
int manhattanDistance(int x1, int y1, int x2, int y2) {
  return (abs(x1 - x2) + abs(y1 - y2));
}

int Astar(int sr, int sc, int tr, int tc) {
  memset(dis, 0x3f, sizeof (dis));
  memset(Par,   -1, sizeof (Par));

  priority_queue <tuple <int, int, int> > Q;

  dis[sr][sc] = 0;
  Q.push({-manhattanDistance(sr, sc, tr, tc), sr, sc});

  int hcost, r, c, nr, nc;
  while(Q.size()) {
    tie(hcost, r, c) = Q.top(); Q.pop();
    if(r == tr && c == tc) return dis[r][c];

    for(int i = 0; i < 4; ++i) {
      nr = r + dr[i];
      nc = c + dc[i];

      if(!valid(nr, nc)) continue;

      if(dis[r][c] + 1 < dis[nr][nc]) {
  dis[nr][nc] = dis[r][c] + 1;
  Par[nr][nc] = dir[i ^ 2];
  Q.push({-dis[nr][nc] -manhattanDistance(nr, nc, tr, tc), nr, nc});
      }
    }
  }
  return -1;
}

void Solve() {
  cin >> si >> sj >> ti >> tj >> n >> m;
  for(int i = 0; i < n; ++i)
    for(int j = 0; j < m; ++j)
      cin >> grid[i][j];

  cout << Astar(si, sj, ti, tj) << endl;
  vector < pair <int, int> > path = restorePath(si, sj, ti, tj);

  for(auto point : path)
    cout << point.first << " " << point.second << endl;
}
```

7.2 Mo's algorithm

```
const int N = 3e4 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const ll INF = 0x3f3f3f3f3f3f3f3f;
const int BLK = 256;

struct query {
  int l, r, id, blk;

  query() = default;
  query(int _l, int _r, int _id) {
    l = _l;
    r = _r;
    id = _id;
    blk = l / BLK;
  }

  bool operator < (const query other) const {
    if(blk ^ other.blk)
      return blk < other.blk;
    return (blk & 1) ? r < other.r : r > other.r;
  }
} queries[M];

int res[M], freq[M << 3], cur;

void add(int id) {
  cur += (++freq[id] == 1);
}

void remove(int id) {
```

```cpp
  cur -= (--freq[id] == 0);
}

int get_res() {
  return cur;
}

int cur_l, cur_r, l, r, n, q, a[N];

void Solve() {
  cin >> n;
  for(int i = 1; i <= n; ++i) cin >> a[i];

  cin >> q;
  for(int i = 1; i <= q; ++i) {
    cin >> l >> r;
    queries[i] = query(l, r, i);
  }

  sort(queries + 1, queries + 1 + q);

  cur_l = 1, cur_r = 0; // assign to right invalid index
  for(int i = 1; i <= q; ++i) {
    int ql = queries[i].l;
    int qr = queries[i].r;

    // Add right
    while(cur_r < qr) add(a[++cur_r]);
    // Add left
    while(cur_l > ql) add(a[--cur_l]);
    // Remove right
    while(cur_r > qr) remove(a[cur_r--]);
    // Remove left
    while(cur_l < ql) remove(a[cur_l++]);

    res[queries[i].id] = get_res();
  }

  for(int i = 1; i <= q; ++i)
    cout << res[i] << "\n";
}
```

7.3 Square root decomposition

```cpp
const int N = 5e5 + 9, M = 1e3 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const ll INF = 0x3f3f3f3f3f3f3f3f;
const int BLK = 256;

int n, q, a[N], type, x, y, z;
vector <int> bs[M];

int query(int l, int r, int val) {
  int cur_l = l / BLK;
  int cur_r = r / BLK;
  int ans = 0;

  if(cur_l == cur_r) {
    for (int i = l; i <= r; ++i)
      ans += (a[i] >= val);
  } else {
    for(int i = l, _end = (cur_l + 1) * BLK; i < _end; ++i)
      ans += (a[i] >= val);
    for(int i = cur_l + 1; i <= cur_r - 1; ++i)
      ans += bs[i].end() - lower_bound(bs[i].begin(), bs[i].end(), val);
    for(int i = cur_r * BLK; i <= r; ++i)
      ans += (a[i] >= val);
  }
  return ans;
}

void build() {
  for(int i = 0; i < n; ++i)
    bs[i / BLK].emplace_back(a[i]);

  for(int i = 0; i < M; ++i)
    sort(bs[i].begin(), bs[i].end());
}

void update(int id, int delta) {
  int pos = lower_bound(bs[id / BLK].begin(), bs[id / BLK].end(), a[id]) - bs[id / BLK].begin();
  bs[id / BLK][pos] = delta;
  sort(bs[id / BLK].begin(), bs[id / BLK].end());
  a[id] = delta;
}

void Solve() {
  cin >> n;
  for(int i = 1; i <= n; ++i) cin >> a[i];

  build();
  cin >> q;
  while(q--) {
    cin >> type >> x >> y;
    if(type == 0) {
      cin >> z;
      cout << query(x, y, z) << endl;
    }
    else
      update(x, y);
  }
}
```

8 Miscellaneous

8.1 C++ template

```cpp
#pragma GCC optimize("Ofast")

#include <bits/stdc++.h>

#define endl '\n'
#define read(a, n)  for(int i = 0; i < n; cin >> a[i++]);

#define debug(args ...) {     \
    string _s = #args;        \
    replace(_s.begin(), _s.end(), ',', ' ');  \
    stringstream _ss(_s);     \
    istream_iterator<string> _it(_ss);    \
    err(_it, args);       \
  }

using namespace std;

using i128 = __int128_t;
using i64 = int64_t;
using i32 = int32_t;

using u128 = __uint128_t;
using u64 = uint64_t;
using u32 = uint32_t;
using ld = long double;

const int N = 2e5 + 9, M = 3e7 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const ld eps = 1e-9;

void err(istream_iterator<string> it) {}
template<typename T, typename ... Args>
void err(istream_iterator<string> it, T a, Args ... args) {
  cerr << *it << " = " << a << endl;
  err(++it, args ...);
}

void fast() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
}

void file() {
  freopen("input.in", "r", stdin);
  freopen("output.out", "w", stdout);
}

void Solve() {

}

int main() {
  fast();
  // file();

  int t = 1; // cin >> t;
  for(int i = 1; i <= t; ++i) {
    Solve();
  }
}
```

8.2 Double comparison

```
bool approximatelyEqual(double a, double b, double epsilon) {
  return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool essentiallyEqual(double a, double b, double epsilon) {
  return fabs(a - b) <= ((fabs(a) > fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool definitelyGreaterThan(double a, double b, double epsilon) {
  return (a - b) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool definitelyLessThan(double a, double b, double epsilon) {
  return (b - a) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

8.3 Fast input/output

```
/**
   Fast Input/Output method for C++:
   1. cin(with sync_with_stdio(false) & cin.tie(nullptr)):
   - int:
   - |n = 5e6| => 420ms
   - |n = 1e7| => 742ms
   - ll:
   - |n = 5e6| => 895ms

   2. read (using getchar()):
   - int:
   - |n = 5e6| => 173ms
   - |n = 1e7| => 172ms
   - ll:
   - |n = 5e6| => 340ms
**/

ll readll () {
  bool minus = false;
  unsigned long long result = 0;
  char ch;
  ch = getchar();

  while (true) {
    if (ch == '-') break;
    if (ch >= '0' && ch <= '9') break;
    ch = getchar();
  }

  if (ch == '-') minus = true;
  else result = ch - '0';

  while (true) {
    ch = getchar();
    if (ch < '0' || ch > '9') break;
    result = result * 10 + (ch - '0');
  }

  if (minus) return -(ll)result;
  return result;
}

int readi () {
  bool minus = false;
  unsigned int result = 0;
  char ch;
  ch = getchar();

  while (true) {
    if (ch == '-') break;
    if (ch >= '0' && ch <= '9') break;
    ch = getchar();
  }

  if (ch == '-') minus = true;
  else result = ch - '0';

  while (true) {
    ch = getchar();
    if (ch < '0' || ch > '9') break;
    result = result * 10 + (ch - '0');
  }

  if (minus) return -(int)result;
  return result;
}
```

8.4 Gcd & Lcm

```
i64 gcd(i64 a, i64 b) { // binary GCD uses about 60% fewer bit operations
  if (!a) return b;

  u64 shift = __builtin_ctzll(a | b);
  a >>= __builtin_ctzll(a);

  while (b) {
    b >>= __builtin_ctzll(b);

    if (a > b)
      swap(a, b);
    b -= a;
  }
  return a << shift;
}

i64 lcm(i64 a, i64 b) {
  return a / gcd(a, b) * b;
}
```

8.5 Modular calculations

```
/*
  - It also has important applications in many tasks unrelated to arithmetic, since it can be used
        with any operations that have the property of associativity:
*/

// 1. Modular Exponentiation

i64 binExp(i64 a, i64 b, i64 p) {
  i64 res;
  for (res = 1; b; b >>= 1) {
    if (b & 1ll)
      res = res * a % p;
    a = a * a % p;
  }
  return res;
}

// 2. Modular Multiplication

i64 binMul(i64 a, i64 b, i64 p) {
  i64 res;
  a %= p;
  for (res = 0; b; b >>= 1) {
    if (b & 1ll)
      res = (res + a) % p;
    a = (a + a) % p;
  }
  return res;
}

// 3. Modular Multiplicative Inverse

i64 modInv(i64 b, i64 p) {
  return binExp(b, p - 2, p); // Guaranteed that p is a Prime Number
}
```

8.6 Debugging tools

```
#define  rforeach(_it, c)    for(__typeof((c).rbegin()) _it = (c).rbegin(); _it != (c).rend(); ++_it)
#define  foreach(_it, c)     for(__typeof((c).begin()) _it = (c).begin(); _it != (c).end(); ++_it)
#define  all(a)              (a).begin(), (a).end()
#define  sz(a)               (int)a.size()
#define  endl                '\n'

typedef int64_t  ll;

#if __cplusplus >= 201402L

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
template <class T> T rnd_in_rng(T a, T b)
{ return uniform_int_distribution <ll> (a, b)(rng); /** [a, b] **/}

template <typename F, typename S>
ostream & operator << (ostream & os, const pair <F, S> & p)
```

```cpp
{ return os << "(" << p.first << ", " << p.second << ")"; }

template <typename F, typename S>
ostream & operator << (ostream & os, const map <F, S> & _mp)
{ os << "["; foreach(it, _mp) { if(it != _mp.begin()) os << ", "; os << it->first << " = " << it->second; } return os << "]"; }

template <typename T>
ostream & operator << (ostream & os, const vector <T> & _v)
{ os << "["; foreach(it, _v) { if(it != _v.begin()) os << ", "; os << *it; } return os << "]"; }

template <typename T>
ostream & operator << (ostream & os, const set <T> & _st)
{ os << "["; foreach(it, _st) { if(it != _st.begin() ) os << ", "; os << *it; } return os << "]"; }

template <typename T, size_t S>
ostream & operator << (ostream & os, const array <T, S> & _ar)
{ os << "["; foreach(it, _ar) { if(it != _ar.begin() ) os << ", "; os << *it; } return os << "]"; }

template <typename T> void write(T _begin, T _end)
{ for(auto i = _begin; i != _end; ++i) cout << (*i) << ' '; cout << endl; }

template <typename T> void read(T _begin, T _end)
{ for(auto i = _begin; i != _end; ++i) cin >> (*i); }

#endif

clock_t start_time;
string run_time()
{ return to_string((clock() - (double)start_time) / CLOCKS_PER_SEC) + " sec"; }

#ifndef ONLINE_JUDGE
#include <sys/resource.h>
void resize_stack()
{
  rlimit rlim;
  getrlimit(RLIMIT_STACK, &rlim);
  rlim.rlim_cur = (1 << 28);
  setrlimit(RLIMIT_STACK, &rlim);
}
#else
void resize_stack() {};
#endif
```

8.7 Overloaded operators to accept 128 Bit integer

```cpp
typedef __uint128_t       ui128;
typedef __int128           i128;

template <class T> string to_string(T x) {
  int sn = 1; if(x < 0) sn = -1, x *= sn; string s = "";
  do { s = "0123456789"[x % 10] + s, x /= 10; } while(x);
  return (sn == -1 ? "-" : "") + s;
}

auto str_to_int(string x) {
  ui128 ret = (x[0] == '-' ? 0 : x[0] - '0');
  for(int i = 1; i < x.size(); ++i) ret = ret * 10 + (x[i] - '0');
  return (x[0] == '-' ? -1 * (i128)ret : ret);
}

istream & operator >> (istream & in, i128 & i) noexcept { string s; in >> s; i = str_to_int(s); return in; }
ostream & operator << (ostream & os, const i128 i) noexcept { os << to_string(i); return os; }
istream & operator >> (istream & in, ui128 & i) noexcept { string s; in >> s; i = str_to_int(s); return in; }
ostream & operator << (ostream & os, const ui128 i) noexcept { os << to_string(i); return os; }
```

8.8 Policy based data structures

```cpp
#if      __cplusplus >= 201402L
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#endif

#if    __cplusplus >= 201402L
using namespace __gnu_cxx;
using namespace __gnu_pbds;
#endif

template <class T, typename Comp = less <T> >
using indexed_set = tree <T, null_type, Comp, rb_tree_tag, tree_order_statistics_node_update>;

template <typename K, typename V, typename Comp = less <K>>
using indexed_map = tree <K, V, Comp, rb_tree_tag, tree_order_statistics_node_update>;
```

8.9 Pseudo random number generator

```cpp
/** pseudo-random number generator | C++xx >= C++11 **/

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

T myRand(T a, T b) {
  return uniform_int_distribution <T> (a, b)(rng);
}
```

8.10 Stress test

```cpp
#include <bits/stdc++.h>

using namespace std;

#define endl '\n'

using i128 = __int128_t;
using i64 = int64_t;
using i32 = int32_t;
using i16 = int16_t;
using i8 = int8_t;

using u128 = __uint128_t;
using u64 = uint64_t;
using u32 = uint32_t;
using u16 = uint16_t;
using u8 = uint8_t;

void fast() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
}

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

/** 64-bit signed int Generator
 **/
i64 int64(i64 a, i64 b) {
  return uniform_int_distribution <i64> (a, b)(rng);
}

/** Customize your Generator depending on the input
 **/
void gen () {
  ofstream cout("input.in");
  i32 t = 2;
  cout << t << endl;

  while (t--) {
    i32 n = int64(1, 100), m = int64(1, 100);
    cout << n << " " << m << endl;

    while (m--) {
      i32 u = int64(1, n), v = int64(1, n), c = int64(1, 4);
      cout << u << " " << v << " " << c << endl;
    }
  }
}

i32 main (i32 arg, char* args[]) {
  fast();

  i32 tc = 0;
  i32 limit = 100;
  if(arg != 3) return 0;

  string flags = "g++ -Wall -Wextra -Wshadow -Og -g -Ofast -std=c++17 -D_GLIBCXX_ASSERTIONS -DDEBUG -ggdb3 -fsanitize=address,undefined -fmax-errors=2 -o ";
  string ex = ".cpp", bf, oz, pr;

  bf = flags + args[1] + " " + args[1] + ex;
  oz = flags + args[2] + " " + args[2] + ex;
  char bff[bf.size() + 1];
  char ozz[oz.size() + 1];
  strcpy(bff, bf.c_str());
  strcpy(ozz, oz.c_str());
```

```cpp

  // compile command
  system(bff);
  system(ozz);

  ex = ".out";
  pr = "./";
  bf = pr + args[1] + " < input.in > " + args[1] + ex;
  oz = pr + args[2] + " < input.in > " + args[2] + ex;
  strcpy(bff, bf.c_str());
  strcpy(ozz, oz.c_str());

  while (++tc <= limit) {
    gen();
    cerr << tc << endl;
    // run command
    system(bff);
    system(ozz);

    ifstream brute_forces("brute_force.out");
    ifstream optimizes("optimized.out");

    string brute_force, optimized;
    getline(brute_forces, brute_force, (char)EOF);
    getline(optimizes, optimized, (char)EOF);

    if(brute_force != optimized) {
      cerr << "Wrong Answer" << endl;
      break;
    } else if (tc == limit) {
      cout << "Accepted insha'a Allah" << endl;
    }
  }
}
```