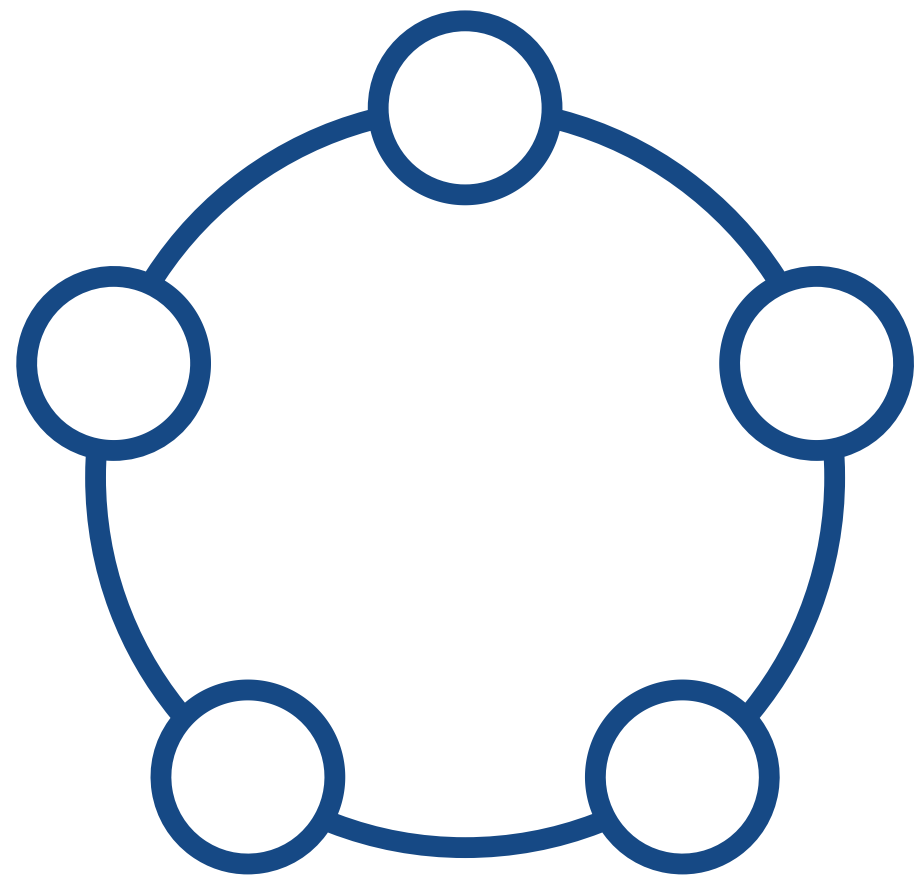


# Semana 4



## Teoria de Grafos



Ana Luiza Sticca, Gabriel  
Antônio, Guilherme Cabral e  
João Pedro Marques

# Problemas



**09** Loopy Cab  
Drivers

**10** Reversing Roads





Loopy Cab Drivers

# Loopy Cab Drivers

Você é motorista de táxi e regularmente leva passageiros ao centro para comprar algo em uma loja. Muitas vezes eles pedem para você esperar por eles, mas pode ser impossível encontrar estacionamento. Quando você não consegue encontrar estacionamento, você tem que dirigir pela cidade, voltando periodicamente ao local onde deixou seu passageiro, para que quando ele voltar da loja você possa buscá-lo novamente. Infelizmente, a cidade está cheia de ruas de mão única e, além disso, nem sempre é possível ir de um determinado ponto da cidade a qualquer outro ponto, porque as ruas de mão única podem não passar entre duas partes diferentes.

Como nem sempre é possível voltar ao ponto de partida, quando se tem que esperar um passageiro dirigindo pela cidade é preciso permanecer nas regiões que permitem voltar ao local onde o deixou. Portanto, seu trabalho é fazer uma lista das regiões onde você pode ficar dirigindo. Você não precisa se preocupar em conseguir chegar a um lugar inicialmente.

# Loopy Cab Drivers

**Entrada:** A entrada consiste na descrição de uma cidade. A descrição começa com uma linha contendo um número inteiro  $0 \leq n \leq 10000$ . A seguir estão  $n$  linhas, cada uma contendo um par de nomes de lojas  $A$  e  $B$ , onde  $A \neq B$ . O par  $A$  seguido por  $B$  indica que existe alguma maneira de dirigir de  $A$  para  $B$  (nessa direção). Nenhum par de lojas é listado mais de uma vez, mas é possível que um par e seu reverso sejam listados. Cada nome é uma sequência entre 1 e 20 letras (a – z, A – Z).


# Loopy Cab Drivers

**Saída:** Imprima cada grupo de duas ou mais lojas que podem ser visitadas juntas (indefinidamente), um grupo por linha. Comece cada grupo com a palavra 'ok'. Liste as lojas dentro de cada grupo em ordem lexicográfica. Ordene os grupos lexicograficamente também. Finalmente, se houver alguma loja onde você NÃO deve dirigir (você não pode visitá-la em conjunto com pelo menos uma outra loja indefinidamente), liste-as na última linha (em ordem lexicográfica). Comece essa lista com a palavra 'evitar'.

O resultado do exemplo indica que você pode dirigir para sempre entre BahamaBucks e CVS, ou entre HEB e Starbucks, mas se estiver no OfficeDepot, não poderá ir a nenhum outro lugar e voltar novamente.


## Sample Input 1

```
6
Starbucks HEB
BahamaBucks CVS
CVS HEB
CVS BahamaBucks
HEB Starbucks
BahamaBucks OfficeDepot
```



## Sample Output 1

```
okay BahamaBucks CVS
okay HEB Starbucks
avoid OfficeDepot
```



# Loopy Cab Drivers

**Estrutura de dados:** Listas de Adjacência

**Vértices:** Lojas da cidade

**Arestas:** rotas entre as lojas



# Loopy Cab Drivers (versão 1)

```
vector< int > grafo[NMAX];
int comp[NMAX];

int tempo = 1;
int pre[NMAX];
int low[NMAX];

int newId = 0;
map< string, int > id;
string iid[NMAX];

int qtdSCC = 0;
vector< string > SCC[NMAX];

stack< int > pilha;
```

```
void Tarjan(int u)
{

    pilha.push(u);

    pre[u] = low[u] = tempo++;

    for(auto v : grafo[u])
    {

        if(pre[v] == 0)
        {

            Tarjan(v);

            low[u] = min(low[u], low[v]);

        }else if(comp[v] == 0)
        {

            low[u] = min(low[u], pre[v]);

        }
```

```

if(pre[u] == low[u])
{
    qtdSCC++;

    int v;

    do
    {
        v = pilha.top();
        pilha.pop();

        comp[v] = qtdSCC;
        SCC[qtdSCC].push_back(iid[v]);

    }while(v != u);

    sort(SCC[qtdSCC].begin(), SCC[qtdSCC].end());
}

```

```

bool cmp(vector< string > a, vector< string > b)
{

    return a[0] < b[0];

}

```

```
int main()
{

    int n, u, v, i;

    string s;

    vector< string > avoid;

    cin >> n;

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

        cin >> s;

        if(id[s] == 0)
        {

            id[s] = ++newId;
            iid[newId] = s;

        }

    }

}
```

```
u = id[s];

cin >> s;

if(id[s] == 0)
{

    id[s] = ++newId;
    iid[newId] = s;

}

v = id[s];

grafo[u].push_back(v);
```

```

for(i = 1; i <= newId; i++)
{
    if(comp[i] == 0) Tarjan(i);

}

sort(SCC + 1, SCC + qtdSCC + 1, cmp);

for(i = 1; i <= qtdSCC; i++)
{
    if(SCC[i].size() == 1)
    {

        avoid.push_back(SCC[i][0]);

        continue;
    }
}

```

```

        cout << "okay";
        for(auto cur : SCC[i]) cout << " " << cur;
        cout << endl;

    }

    if(avoid.size() != 0)
    {

        cout << "avoid";
        for(auto cur : avoid) cout << " " << cur;
        cout << endl;

    }
    return 0;
}

```

# Loopy Cab Drivers (versão 2)

```
// Executar primeiro DFS para obter os tempos de 'leave'
Stack<Integer> s = new Stack<>();
boolean[] visitado = new boolean[unico];
for (int i = 0; i < unico; i++) {
    if (!visitado[i]) {
        dfs1(adj1, visitado, s, i);
    }
}

// Executar segundo DFS para obter componentes
List<Integer> unicoComponente = new ArrayList<>();
List<List<Integer>> outrosComponentes = new ArrayList<>();
Arrays.fill(visitado, false);
while (!s.isEmpty()) {
    int i = s.pop();

    if (!visitado[i]) {
        List<Integer> componente = new ArrayList<>();
        dfs2(adj2, visitado, componente, i);
        if (componente.size() == 1) {
            unicoComponente.add(componente.get(0));
        } else {
            outrosComponentes.add(componente);
        }
    }
}
```

```
// Ordenar tudo
for (List<Integer> componente : outrosComponentes) {
    Collections.sort(componente);
}
outrosComponentes.sort(Comparator.comparing(list -> list.get(0)));
Collections.sort(unicoComponente);

// Imprimir componentes conectados
for (List<Integer> componente : outrosComponentes) {
    System.out.print("ok ");
    for (int j : componente) {
        System.out.print(inverso.get(j) + " ");
    }
    System.out.println();
}

// Imprimir componentes solitários
if (!unicoComponente.isEmpty()) {
    System.out.print("avoid ");
    for (int i : unicoComponente) {
        System.out.print(inverso.get(i) + " ");
    }
    System.out.println();
}
```

```
static void dfs1(List<Integer>[] adj, boolean[] visitado, Stack<Integer> s, int atual) {  
    visitado[atual] = true;  
    for (int i : adj[atual]) {  
        if (!visitado[i]) {  
            dfs1(adj, visitado, s, i);  
        }  
    }  
    s.push(atual);  
}  
  
static void dfs2(List<Integer>[] adj, boolean[] visitado, List<Integer> comp, int atual) {  
    visitado[atual] = true;  
    comp.add(atual);  
  
    for (int i : adj[atual]) {  
        if (!visitado[i]) {  
            dfs2(adj, visitado, comp, i);  
        }  
    }  
}
```

# Exemplos rodados

```
6
Starbucks HEB
BahamaBucks CVS
CVS HEB
CVS BahamaBucks
HEB Starbucks
BahamaBucks OfficeDepot
okay BahamaBucks CVS
okay HEB Starbucks
avoid OfficeDepot
```

```
6
Starbucks HEB
BahamaBucks CVS
CVS HEB
CVS BahamaBucks
HEB Starbucks
BahamaBucks OfficeDepot
ok CVS BahamaBucks
ok HEB Starbucks
avoid OfficeDepot

Process finished with exit code 0
|
```

10

Reversing Roads



# Reversing Roads

Você trabalha para a cidade de One-Direction-Ville. A cidade determina que todas as estradas em seus limites tenham apenas uma direção. Você está avaliando propostas para um novo loteamento e sua rede viária. Um problema que você observou em algumas propostas iniciais é que é impossível chegar a determinados locais a partir de outros ao longo das estradas propostas. Para acelerar a avaliação de propostas subsequentes, você deseja escrever um programa para determinar se é possível chegar a qualquer local a partir de qualquer outro local; você chama isso de proposta válida. E se uma proposta não for válida, então o seu programa deverá descobrir se existe uma maneira fácil de corrigi-la invertendo a direção de uma das estradas.

# Reversing Roads

**Entrada:** A entrada consiste em vários casos de teste, no máximo 5. Cada caso de teste começa com uma linha contendo dois inteiros,  $1 \leq m \leq 50$  e  $0 \leq n \leq m(m-1)/2$ .  $m$  indica o número de locais na proposta, e  $n$  indica o número de estradas que conectam esses locais. Seguindo isso estão  $n$  linhas. Cada linha contém dois inteiros separados por espaço  $a$  e  $b$ , onde  $0 \leq a, b \leq m$  e  $a \neq b$ . Isto indica que existe uma estrada a partir do local  $a$  para localização  $b$ . Se houver uma estrada de  $a$  para  $b$ , então não haverá estrada de  $b$  para  $a$ . Além disso, nunca haverá mais de uma estrada entre dois locais. O último caso de teste é seguido pelo fim do arquivo.

# Reversing Roads

**Saída:** Para cada caso, exibir o número do caso seguido da indicação se a proposta é válida ou não. Se a proposta for válida, saída válida. Se não for válido, mas ao inverter o sentido de uma estrada pode tornar-se válido, imprima os dois locais que descrevem a estrada existente que deve ser invertida. Se mais de uma inversão de estrada puder criar uma proposta válida, imprima a primeira que aparecer na entrada. Se a proposta não for válida e impossível de ser válida invertendo uma estrada, imprima inválida. Siga o formato da saída de amostra.

### Sample Input 1

```
3 3
0 1
1 2
2 0
3 3
0 1
1 2
0 2
3 2
1 2
0 2
4 4
0 1
1 2
2 3
0 3
```

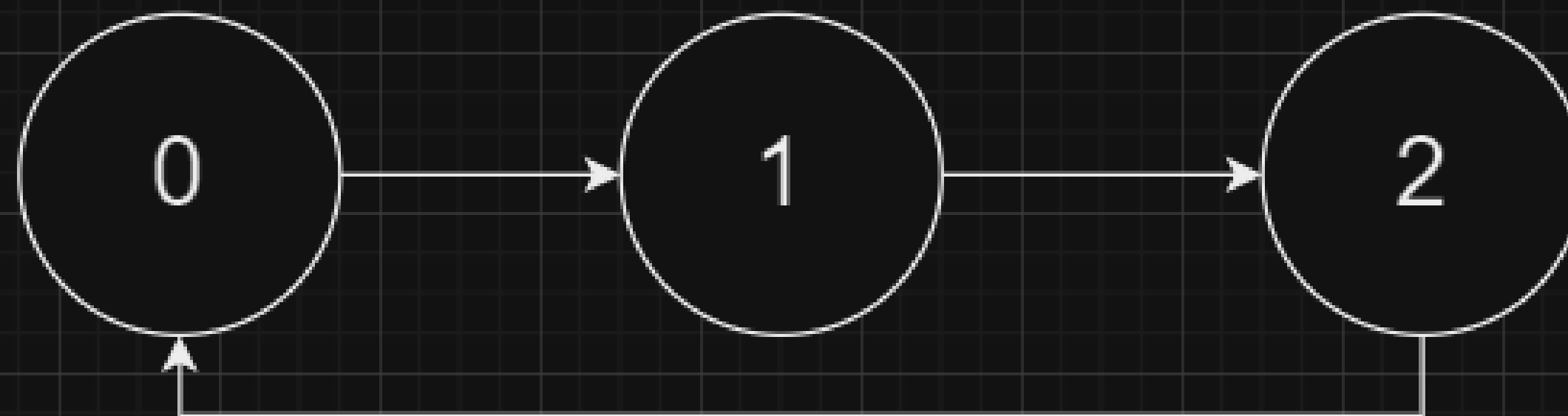


### Sample Output 1

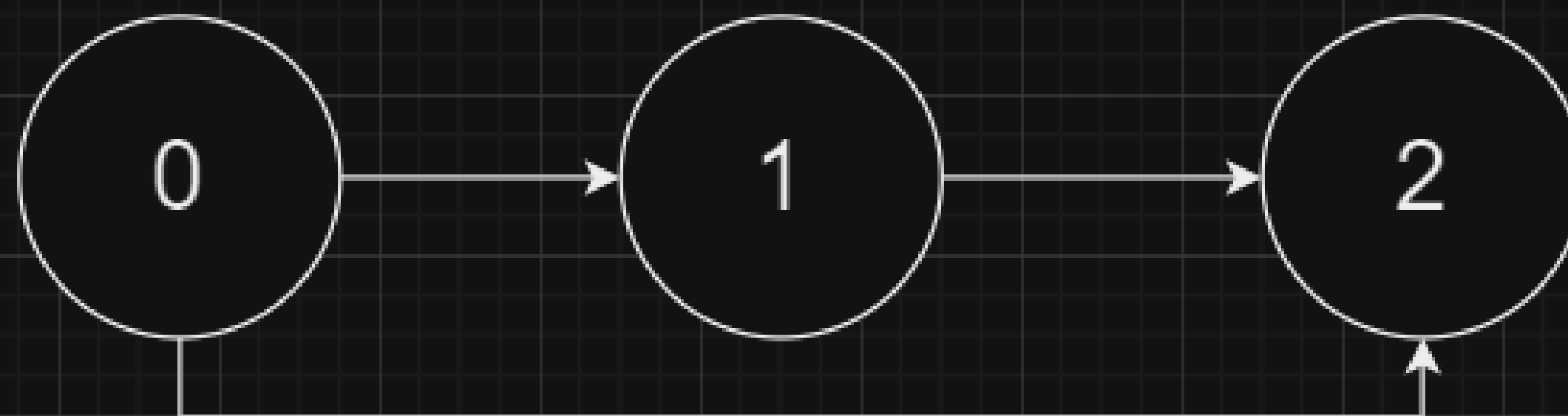
```
Case 1: valid
Case 2: 0 2
Case 3: invalid
Case 4: 0 3
```



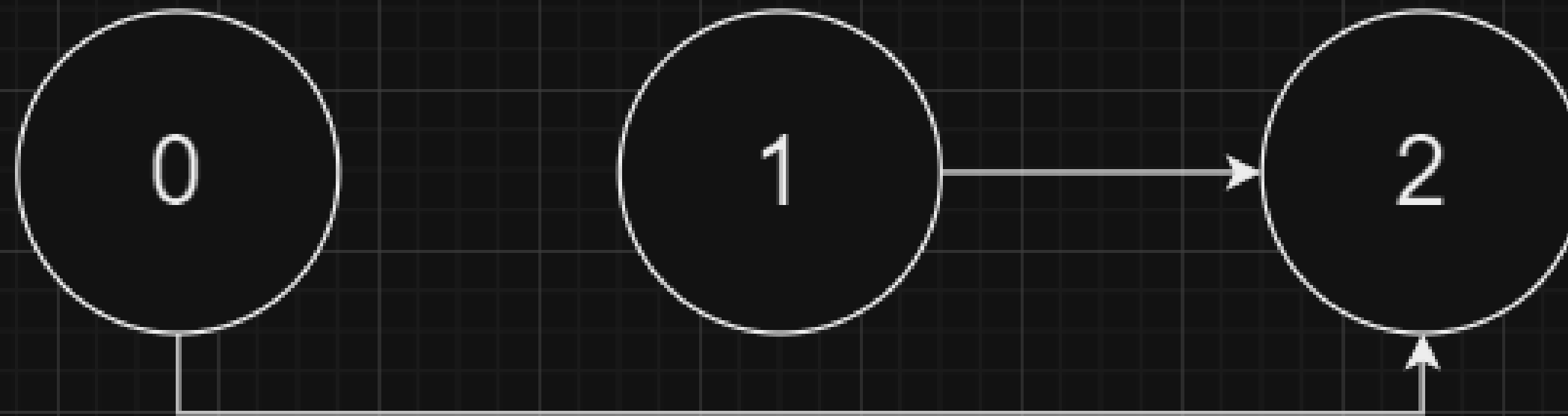
1 caso: válido



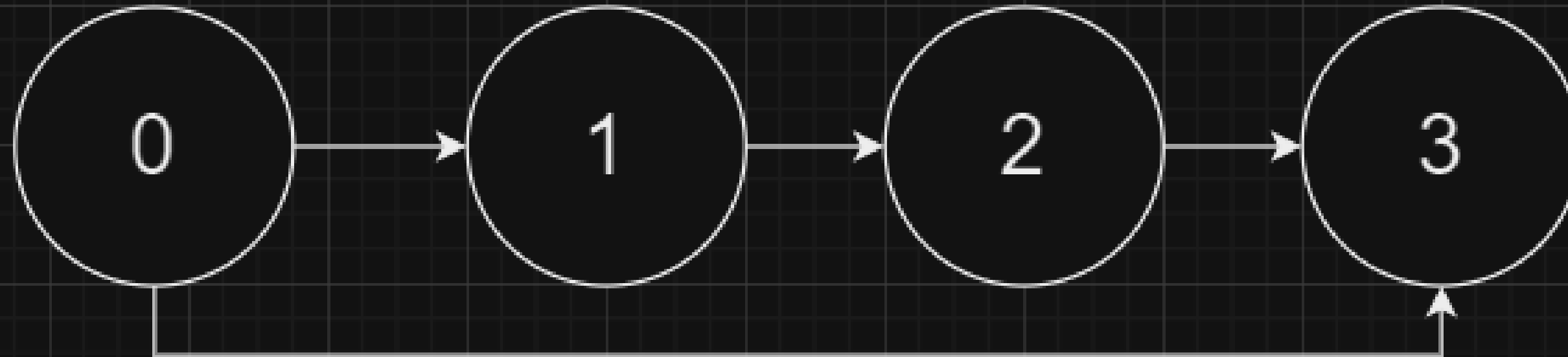
2 caso: trocar a direção da aresta 0-2



3 caso: inválido



4 caso: trocar a direção da aresta 0-3



# Reversing Roads

**Estrutura de dados:** Listas de Adjacência

**Vértices:** Locais do loteamento

**Arestas:** Estradas que conectam os locais

# Reversing Roads (versão 1)

```
bool grafo[NMAX][NMAX];
int comp[NMAX];

vector< pair< int, int > > edges;

int tempo;
int pre[NMAX];
int low[NMAX];

int qtdSCC;
vector< int > SCC;

stack< int > pilha;
```

```
pilha.push(u);

pre[u] = low[u] = tempo++;

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

    if(grafo[u][v] == false) continue;

    if(pre[v] == 0)
    {

        Tarjan(v);

        low[u] = min(low[u], low[v]);

    }else if(comp[v] == 0)
    {

        low[u] = min(low[u], pre[v]);
```



```

int main()
{

    int u, v, caso, i, j;

    pair< int, int > invert;

    caso = 1;

    while(cin >> n >> m)
    {

        for(i = 0;i < n;i++)
            for(j = 0;j < n;j++)
                grafo[i][j] = false;

        edges.clear();

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

            cin >> u >> v;

            grafo[u][v] = true;

            edges.push_back({u, v});

```

```

tempo = 1;
for(i = 0;i < n;i++) pre[i] = 0;
for(i = 0;i < n;i++) comp[i] = 0;

qtdSCC = 0;
SCC.clear();
Tarjan(0);

cout << "Case " << caso++ << ": ";

if(SCC.size() == n)
{

    cout << "valid" << endl;

}else
{

    invert = {-1, -1};

    for(j = 0;j < m;j++)
    {

```

```

for(j = 0; j < m; j++)
{
    u = edges[j].first;
    v = edges[j].second;

    grafo[u][v] = false;
    grafo[v][u] = true;

    tempo = 1;
    for(i = 0; i < n; i++) pre[i] = 0;
    for(i = 0; i < n; i++) comp[i] = 0;

    qtdSCC = 0;
    SCC.clear();
    Tarjan(0);

    if(SCC.size() == n)
    {

        invert = {u, v};

        break;
    }
}

```

```

        grafo[u][v] = true;
        grafo[v][u] = false;
    }

    if(invert == make_pair(-1, -1))
        cout << "invalid" << endl;
    else
        cout << invert.first << " " << invert.second << endl;
}

```

# Reversing Roads (versão 2)

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int u, v, caso, i, j;
    Pair<Integer, Integer> invertido;

    caso = 1;

    while (scanner.hasNextInt()) {
        n = scanner.nextInt(); // Número de vértices do grafo
        m = scanner.nextInt(); // Número de arestas do grafo

        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                grafo[i][j] = false; // Inicializa a matriz de adjacência como falsa (sem arestas)
            }
        }

        arestas.clear(); // Limpa a lista de arestas

        for (i = 0; i < m; i++) {
            u = scanner.nextInt();
            v = scanner.nextInt();

            grafo[u][v] = true; // Adiciona a aresta de u para v no grafo
            arestas.add(new Pair<>(u, v)); // Adiciona a aresta à lista de arestas
        }
    }
}
```

```

tempo = 1;
Arrays.fill(pre, val: 0); // Inicializa o vetor de tempos de descoberta como 0
Arrays.fill(comp, val: 0); // Inicializa o vetor de componentes fortemente conectados como 0

qtdSCC = 0; // Inicializa o contador de componentes fortemente conectados como 0
SCC.clear(); // Limpa a lista de vértices do componente fortemente conectado
Tarjan(u: 0); // Executa o algoritmo de Tarjan a partir do vértice 0

System.out.print("Caso " + caso++ + ": ");

if (SCC.size() == n) {
    System.out.println("válido"); // Se todos os vértices estão em um componente fortemente conectado, o grafo é válido
} else {
    invertido = new Pair<>(-1, -1);

    for (j = 0; j < m; j++) {
        u = arestas.get(j).first;
        v = arestas.get(j).second;

        grafo[u][v] = false; // Remove a aresta de u para v
        grafo[v][u] = true; // Adiciona a aresta de v para u (invertida)

        tempo = 1;
        Arrays.fill(pre, val: 0);
        Arrays.fill(comp, val: 0);

        qtdSCC = 0;
        SCC.clear();
        Tarjan(u: 0);

        if (SCC.size() == n) {
            invertido = new Pair<>(u, v); // Se o grafo invertido é válido, armazena as arestas invertidas
            break;
        }

        grafo[u][v] = true; // Restaura a aresta de u para v
        grafo[v][u] = false; // Remove a aresta de v para u
    }
}

```

```

public static void Tarjan(int u) {
    pilha.push(u);
    pre[u] = baixo[u] = tempo++;

    for (int v = 0; v < n; v++) {
        if (!grafo[u][v]) continue; // Se não há aresta de u para v, continue para o próximo vértice

        if (pre[v] == 0) {
            Tarjan(v);
            baixo[u] = Math.min(baixo[u], baixo[v]); // Atualiza o tempo de descoberta mais baixo
        } else if (comp[v] == 0) {
            baixo[u] = Math.min(baixo[u], pre[v]); // Atualiza o tempo de descoberta mais baixo
        }
    }

    if (pre[u] == baixo[u]) {
        qtdSCC++;
        int v;

        do {
            v = pilha.pop();
            comp[v] = qtdSCC;
            if (qtdSCC == 1) SCC.add(v); // Adiciona o vértice ao componente fortemente conectado
        } while (v != u);
    }
}
}

```

# Exemplos rodados

```
3 3
0 1
1 2
2 0
Case 1: valid
3 3
0 1
1 2
0 2
Case 2: 0 2
3 2
1 2
0 2
Case 3: invalid
4 4
0 1
1 2
2 3
0 3
Case 4: 0 3
```

```
3 3
0 1
1 2
2 0
3 3
0 1
1 2
0 2
3 2
1 2
0 2
4 4
0 1
1 2
2 3
0 3
Caso 1: válido
Caso 2: 0 2
Caso 3: -1 -1
Caso 4: 0 3
```