CLUB DE PROGRAMACIóN COMPETITIVA
UADY

7

Introducción a árboles y grafos

CLUB DE PROGRAMACIóN COMPETITIVA UADY

Grafos

Es una estructura que consta de aristas o nodos conectados entre si por vértices, que pueden ser dirigidos, no dirigidos y pueden o no tener peso.

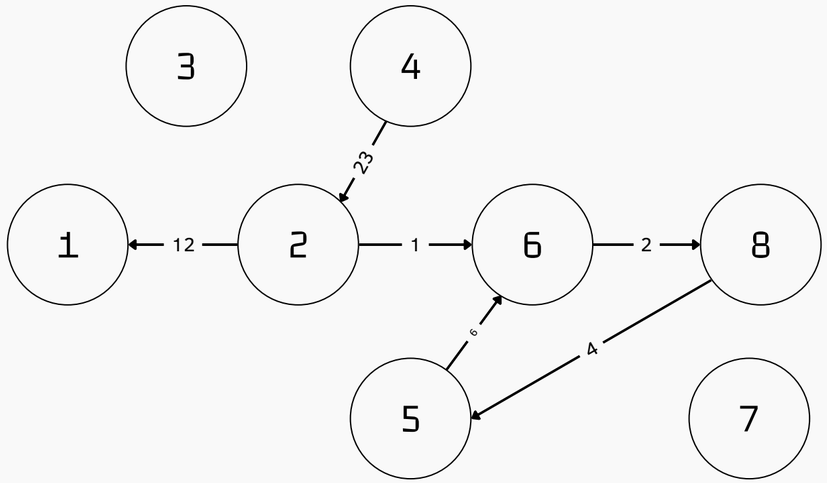
Arboles

Son grafos donde si existen **n** aristas y **n-1** vértices. Se cumple que para cada par de aristas existe camino único entre ambas. Puede ser descrito como un grafo aciclico diirigido con la restricción de que cada nodo solo es apuntado por un único nodo

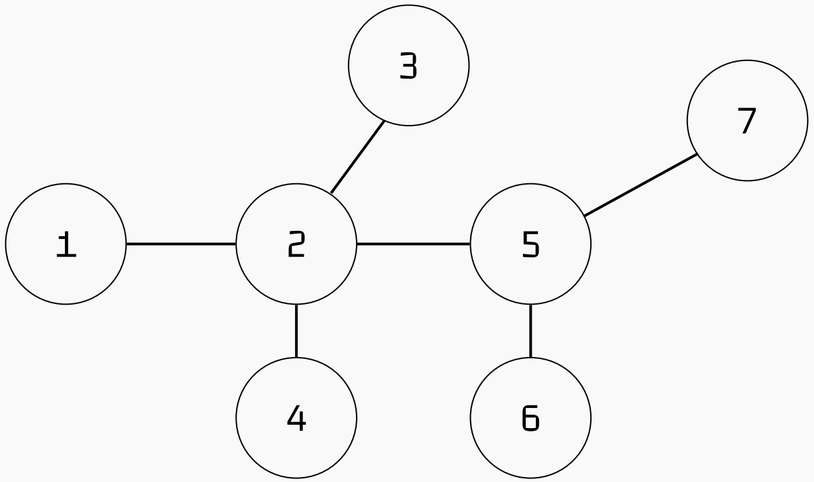
Tambien puede ser definido recursivamente si seleccionamos una arista como “raíz”: Un arbol consta de una raíz que está conectada a algúna cantidad de (sub)arboles.

CLUB DE PROGRAMACIóN COMPETITIVA UADY

Grafo dirigido con pesos



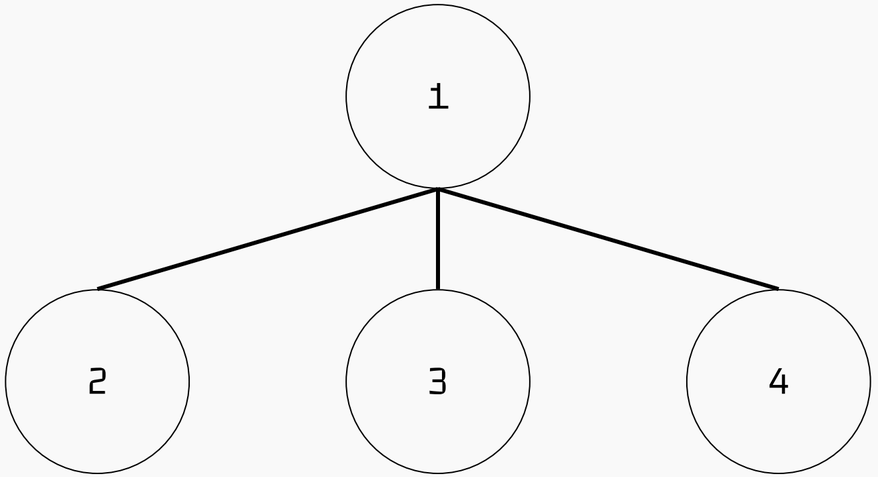
Grafo no dirigido sin pesos



Supongamos que tenemos un conjunto de **n** ciudades conectadas por **n-1** caminos y sabemos que es posible alcanzar cualquier ciudad utilizando algúna cantidad de caminos a partir de cualquier otra ciudad. Partiendo de una ciudad **u**, ¿Cual es la mayor distancia que puede ser recorrida para llegar a cualquier otra ciudad?

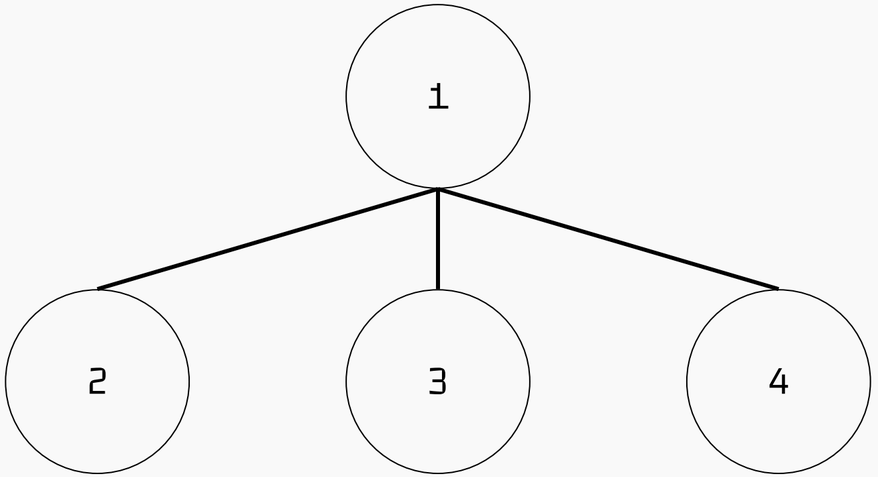
Supongamos que tenemos un conjunto de **n** ciudades conectadas por **n-1** caminos y sabemos que es posible alcanzar cualquier ciudad utilizando algúna cantidad de caminos a partir de cualquier otra ciudad. Partiendo de una ciudad **u**, ¿Cual es la mayor distancia que puede ser recorrida para llegar a cualquier otra ciudad?

Esta situación puede ser modelada como un arbol.
Cada ciudad puede ser vista como un arista y los caminos entre ellas como vértices, siendo **u** la raíz.



Supongamos que tenemos un conjunto de **n** ciudades conectadas por **n-1** caminos y sabemos que es posible alcanzar cualquier ciudad utilizando algúna cantidad de caminos a partir de cualquier otra ciudad. Partiendo de una ciudad **u**, ¿Cual es la mayor distancia que puede ser recorrida para llegar a cualquier otra ciudad?

Esta situación puede ser modelada como un arbol.
Cada ciudad puede ser vista como un arista y los caminos entre ellas como vértices, siendo **u** la raíz.
Ahora solo necesitamos una manera de recorrer el arbol y encontrar la distancia de la ciudad más lejana a partir de la raíz.



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

Para recorrer un grafo o arbol primero necesitamos una estructura que nos indique para cada nodo todos los que son adyacentes a el.

Para un grafo de **n** nodos usualmente se nos proporciona una lista de **m** parejas **(u, v)**, que indica que existe una conexión desde **u** hasta **v**, bidireccional en caso de que el problema indique que se trata de un grafo no dirigido. Construir una lista de adyacencia a partir de esta entrada es sencillo.

```
cin >> n >> m;
vector<int> adj[n];
for[int i = 0; i < m; i++]{
    ll u, v;
    cin >> u >> v;
    adj[u].push_back[v];
    adj[v].push_back[u];
}
```

adj es un arreglo de vectores donde **adj[i]** contiene un vector con todos los nodos que son adyacentes a **i**

adj también podría ser un mapa de vectores o un mapa de sets.

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

Lista de adyacencia

```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```
Entrada:

9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

Lista de adyacencia

```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

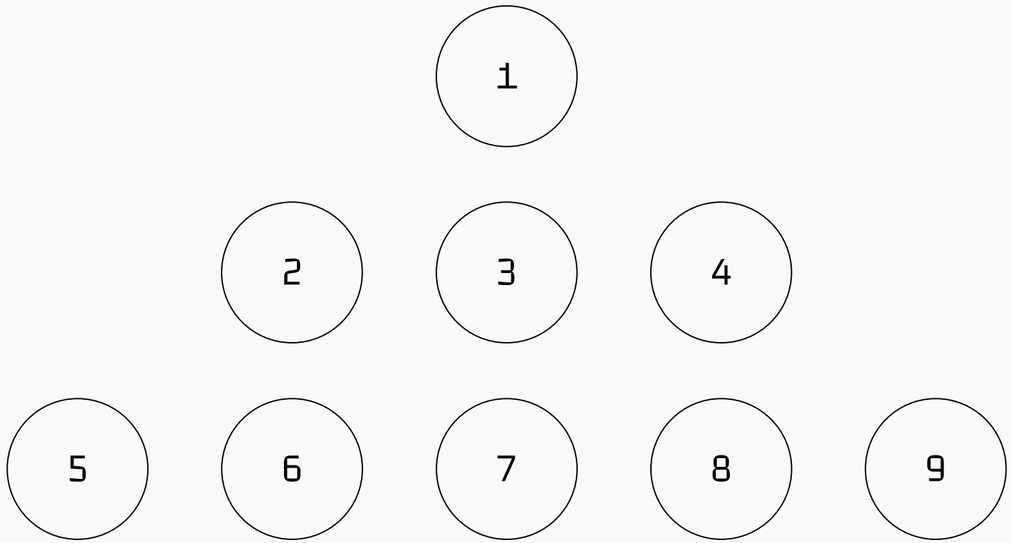
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {}
2: {}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

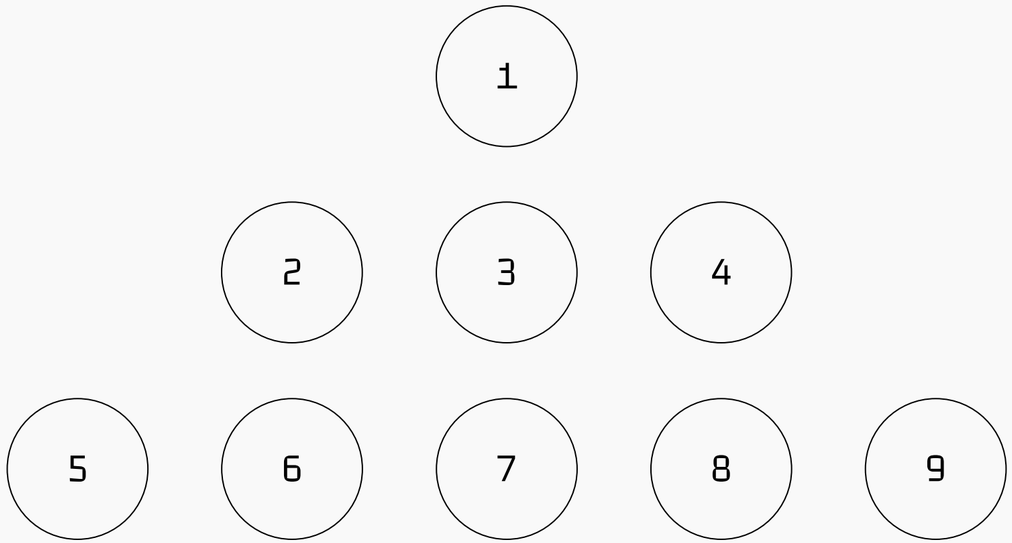
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {}
2: {}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

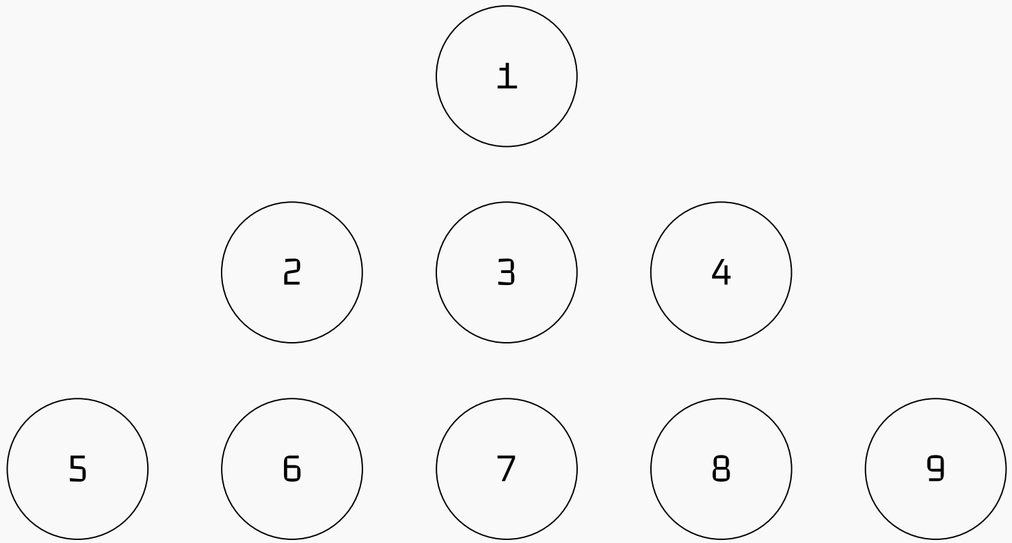
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {}
2: {}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

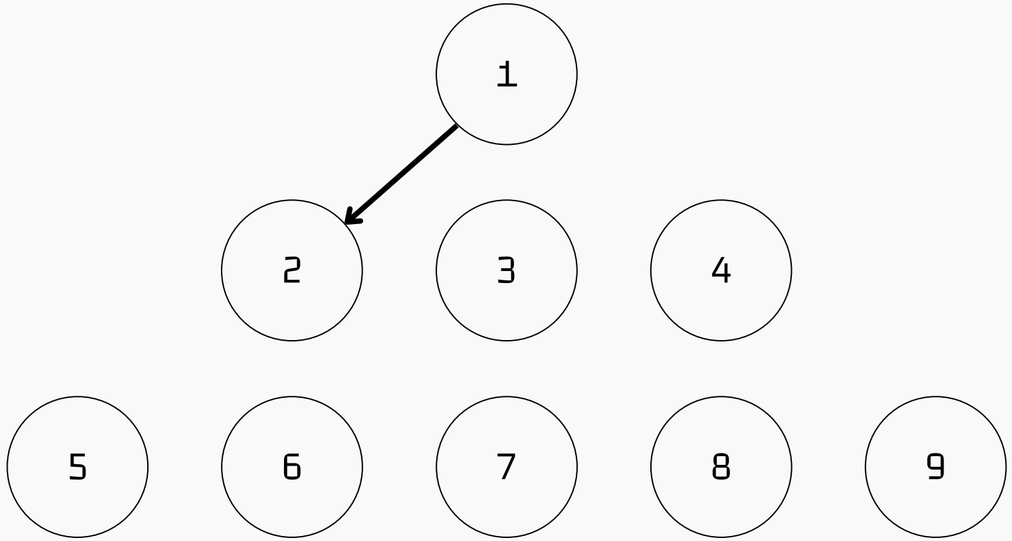
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2}
2: {}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

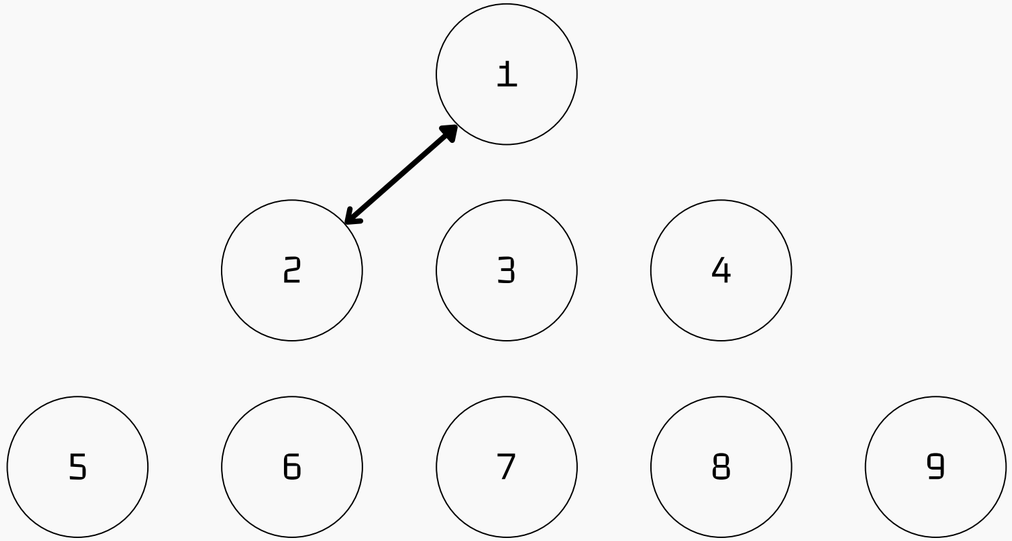
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2}
2: {1}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

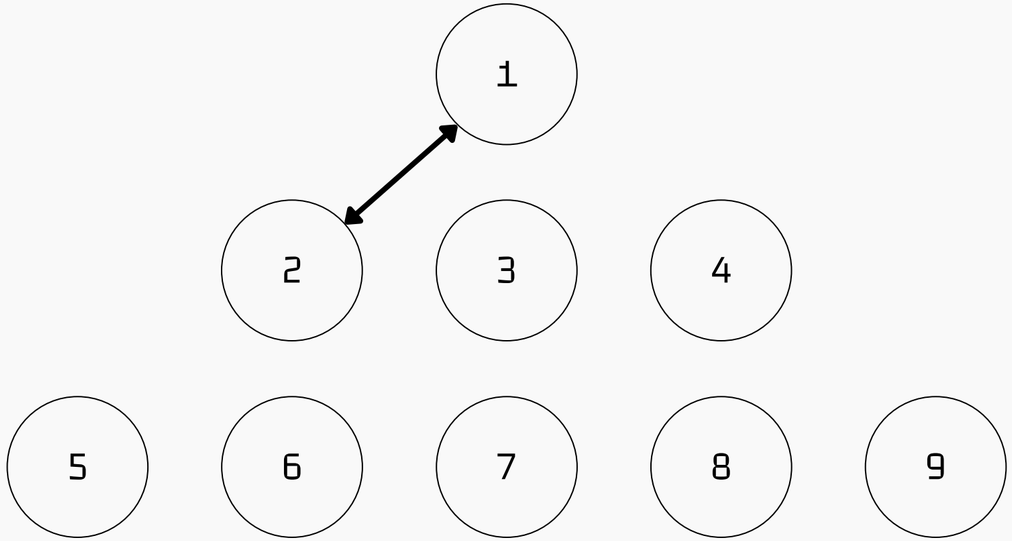
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2}
2: {1}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

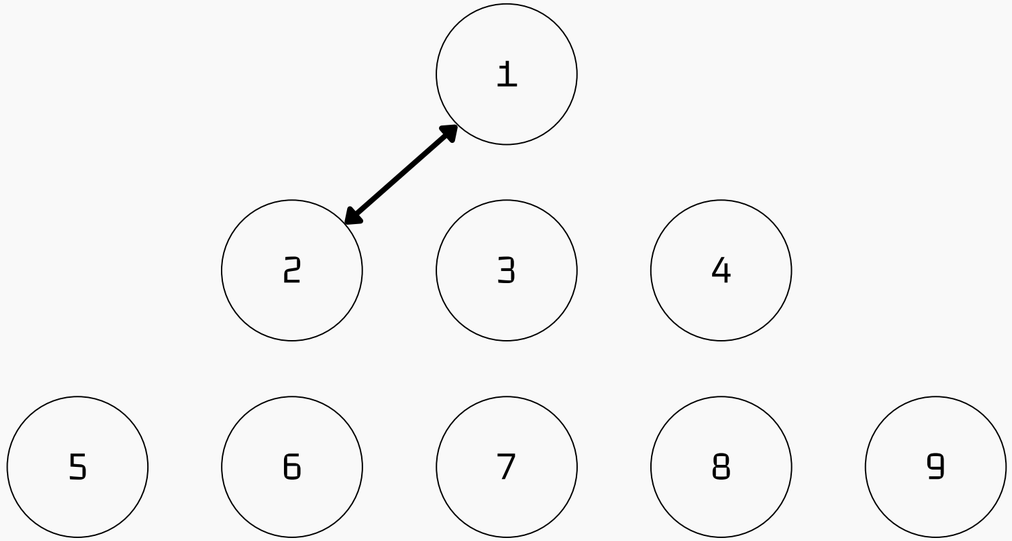
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2}
2: {1}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

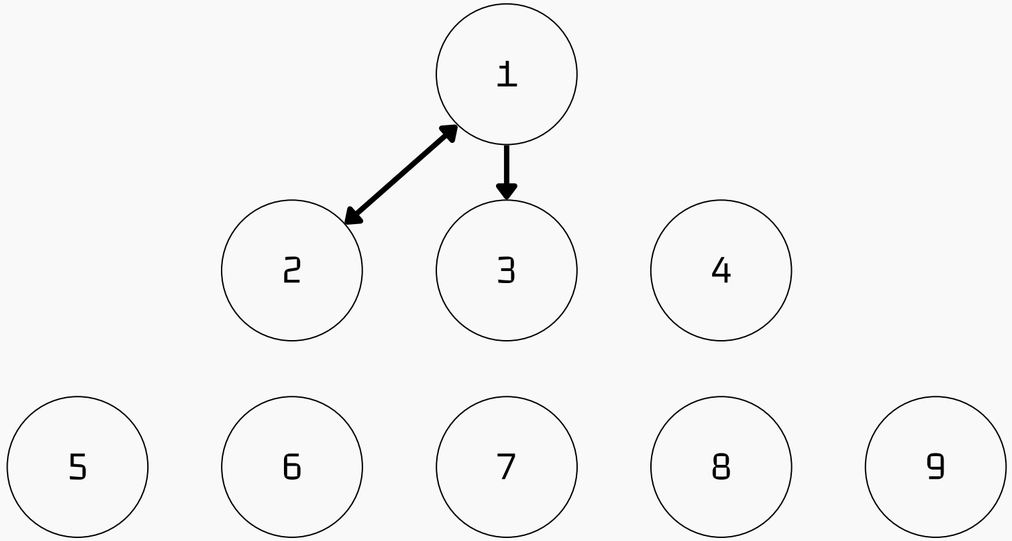
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3}
2: {1}
3: {}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

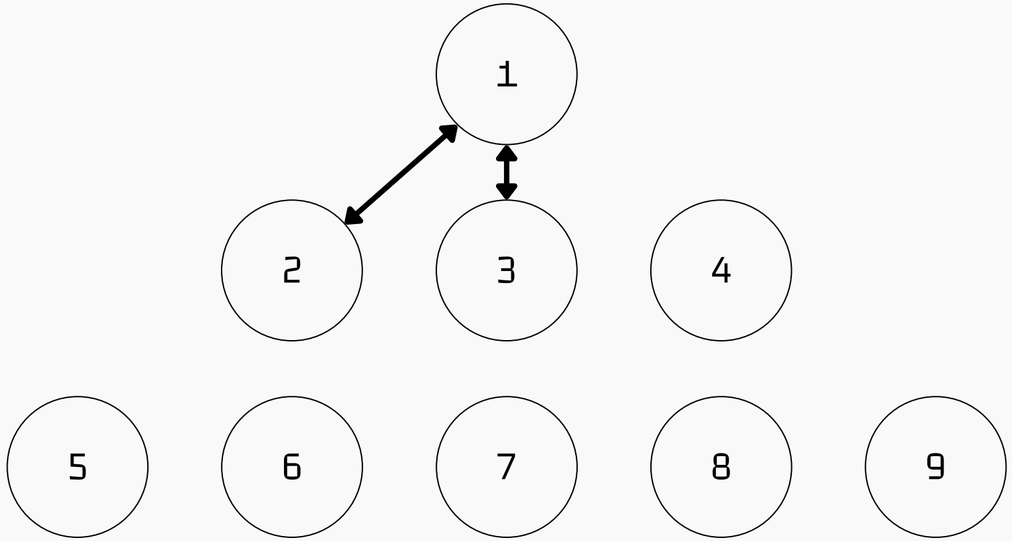
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3}
2: {1}
3: {1}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

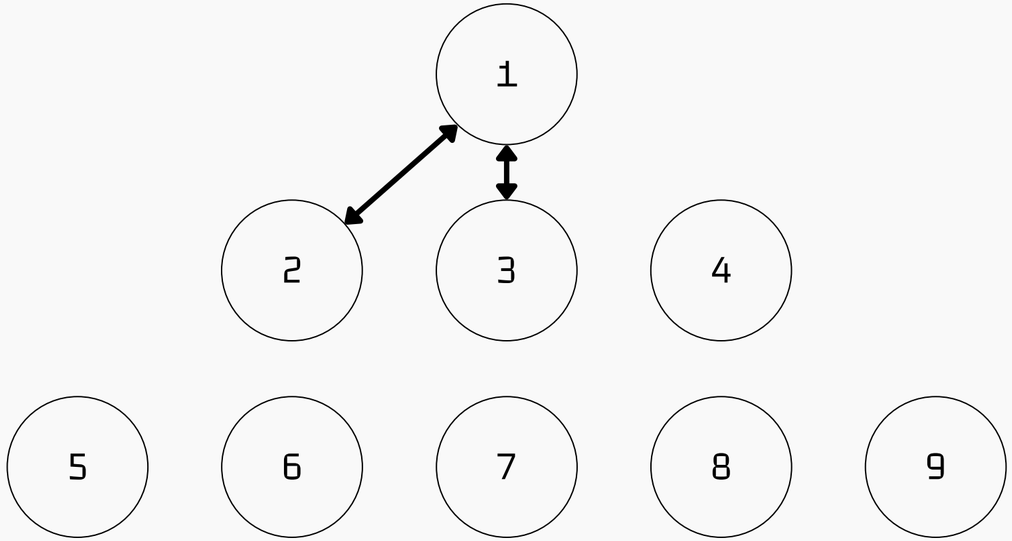
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3}
2: {1}
3: {1}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

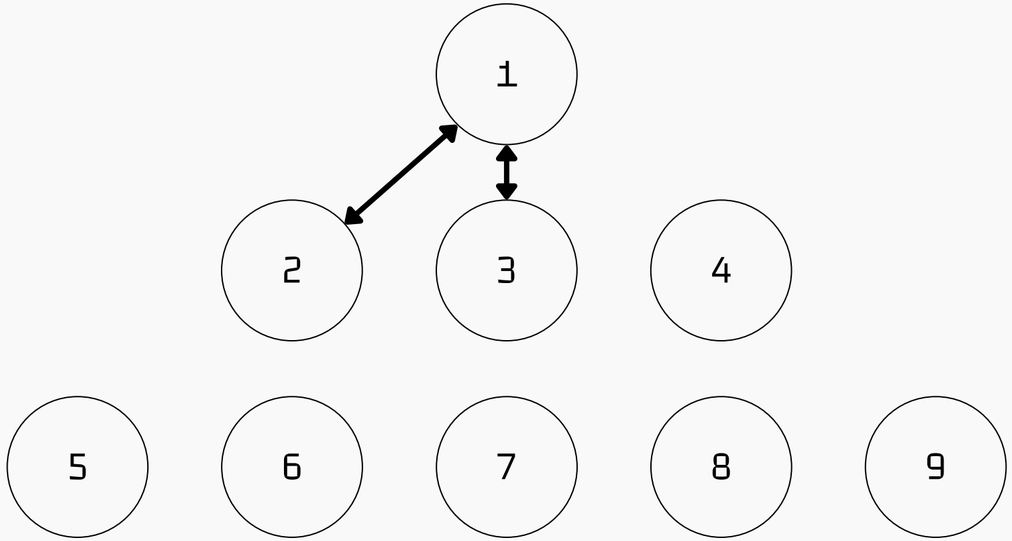
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3}
2: {1}
3: {1}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

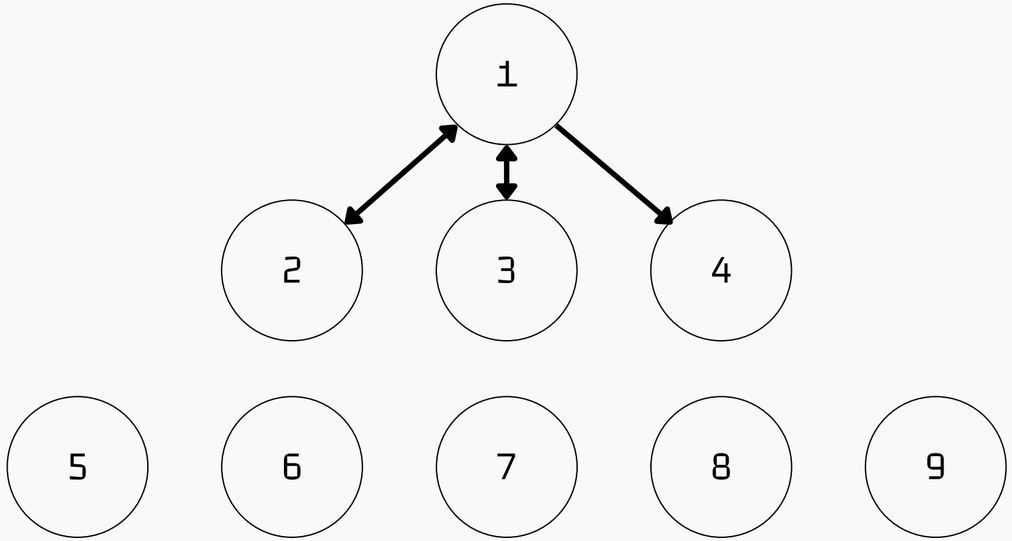
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1}
3: {1}
4: {}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

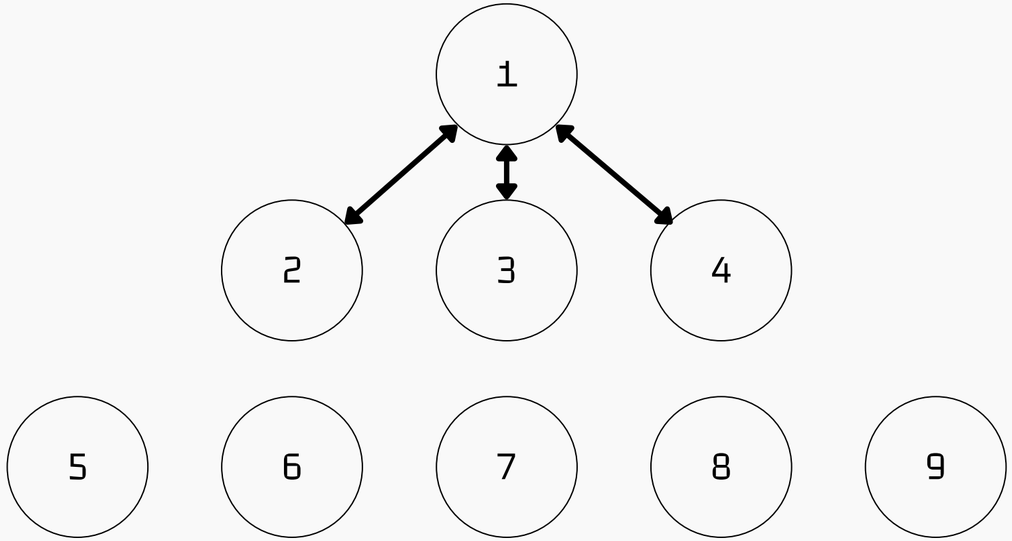
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1}
3: {1}
4: {1}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

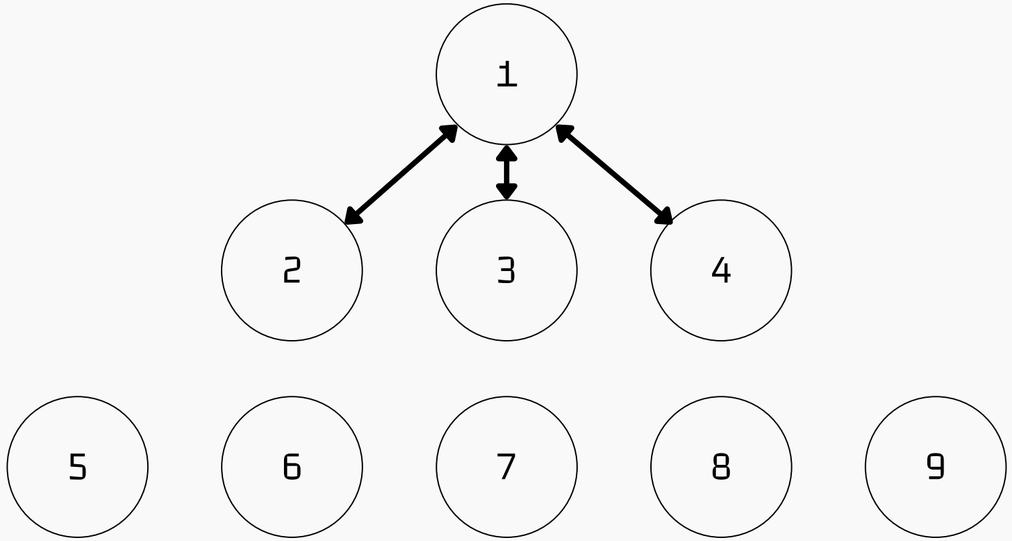
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1}
3: {1}
4: {1}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

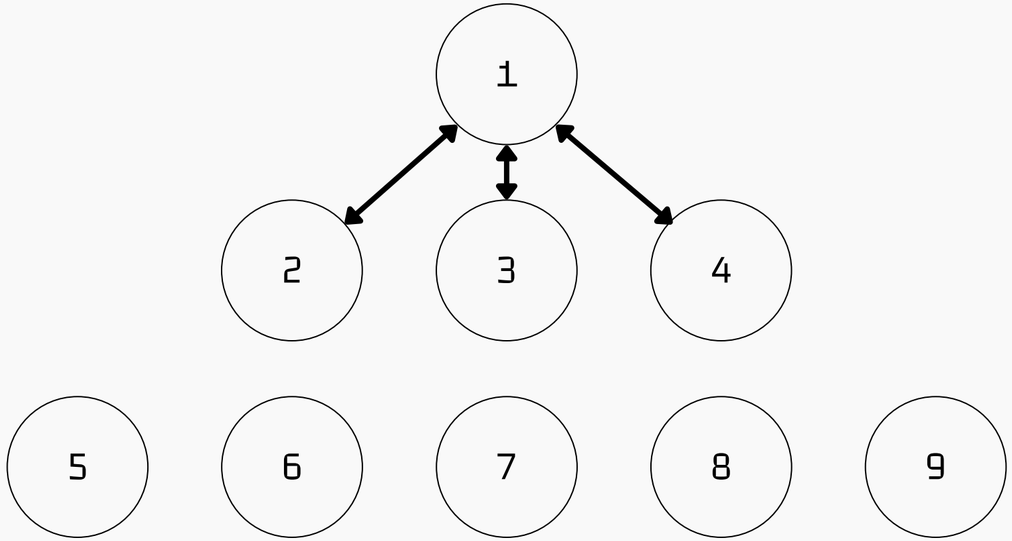
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1}
3: {1}
4: {1}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

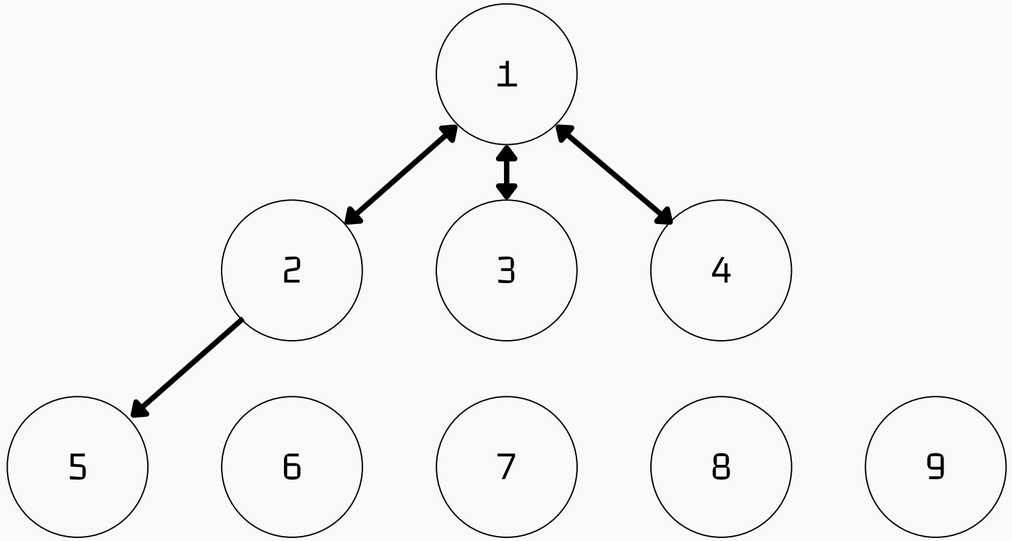
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5}
3: {1}
4: {1}
5: {}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

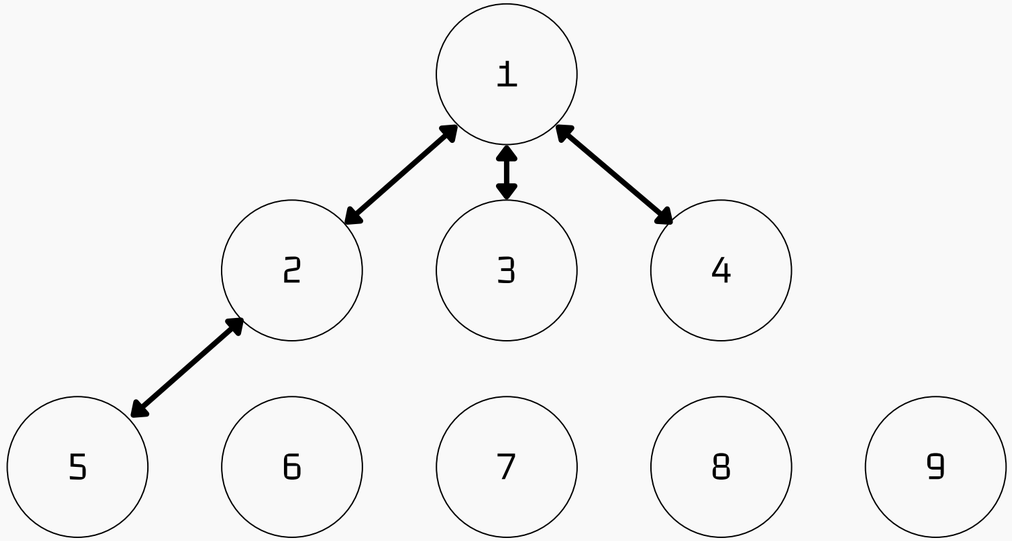
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5}
3: {1}
4: {1}
5: {2}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

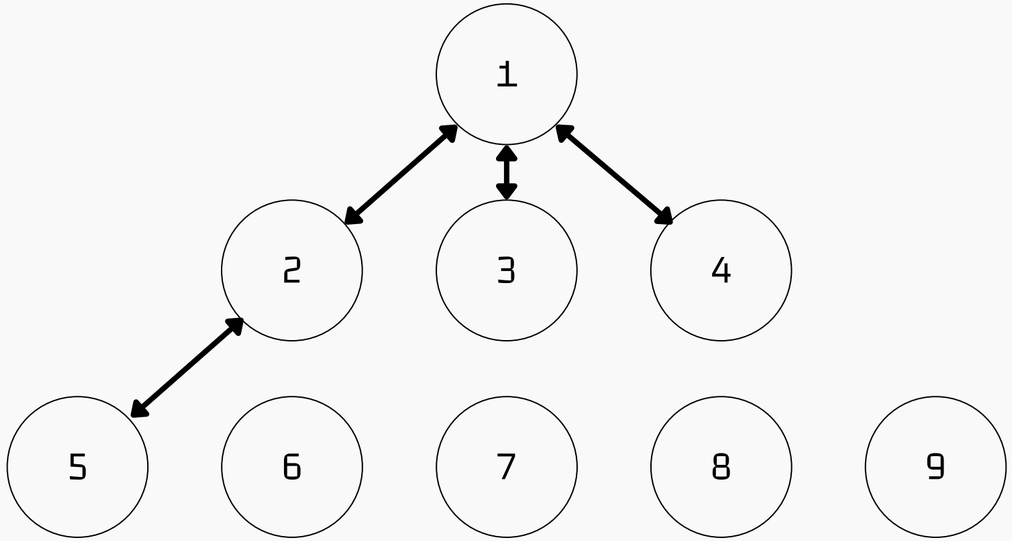
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5}
3: {1}
4: {1}
5: {2}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

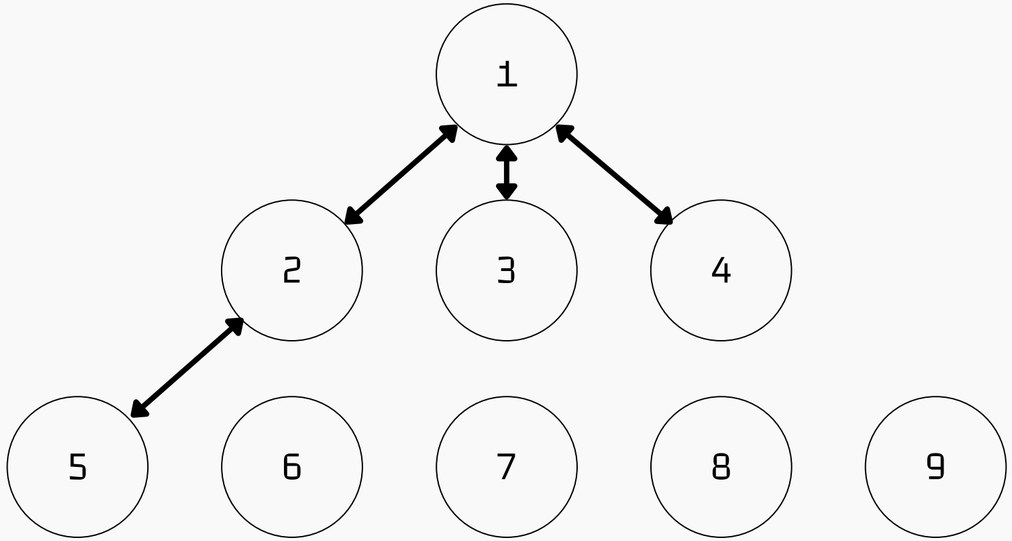
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5}
3: {1}
4: {1}
5: {2}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

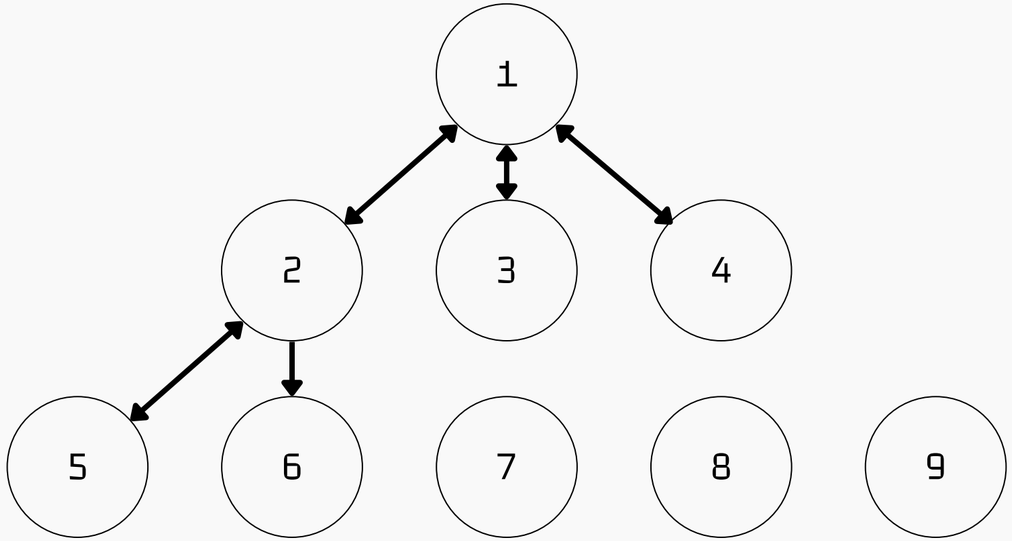
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1}
4: {1}
5: {2}
6: {}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

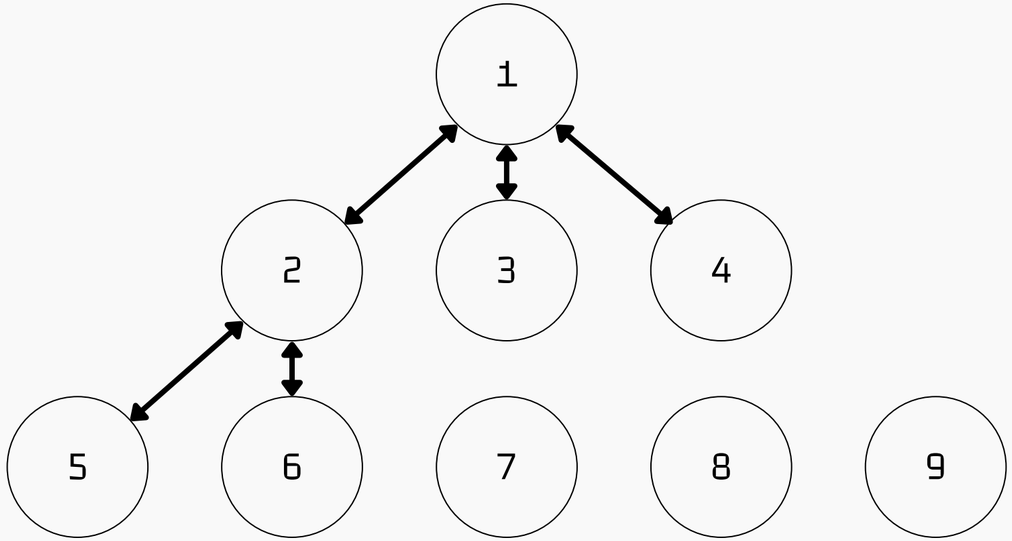
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1}
4: {1}
5: {2}
6: {2}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

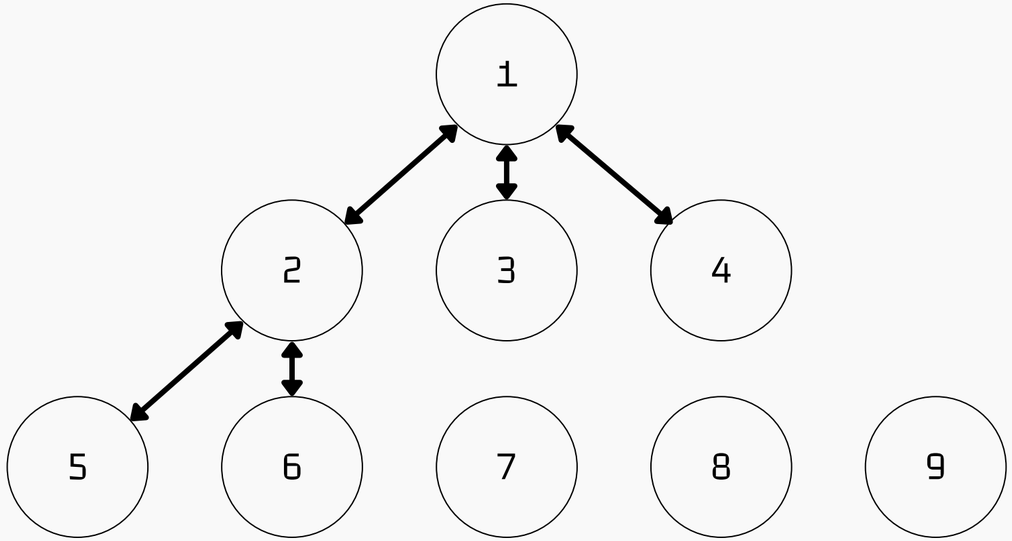
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1}
4: {1}
5: {2}
6: {2}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

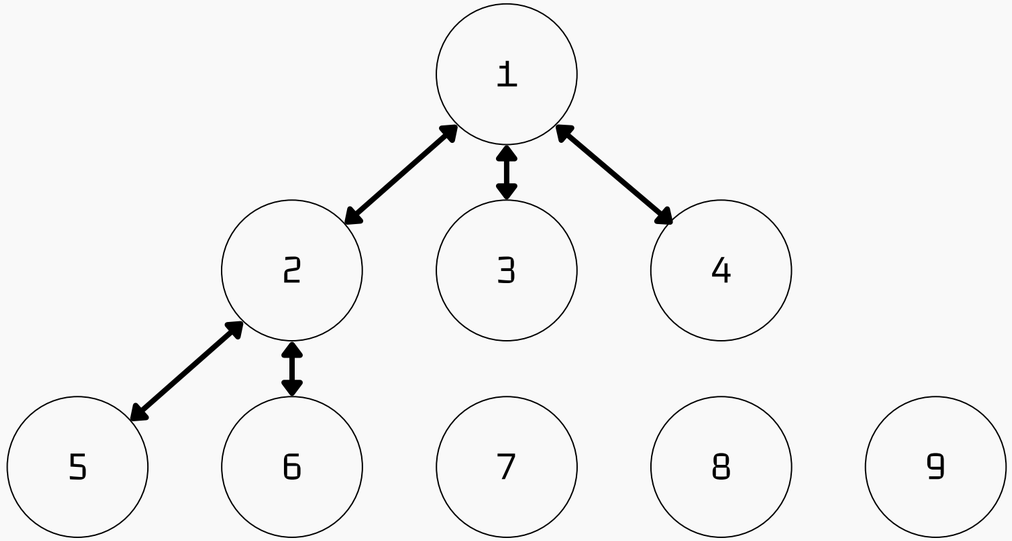
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1}
4: {1}
5: {2}
6: {2}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

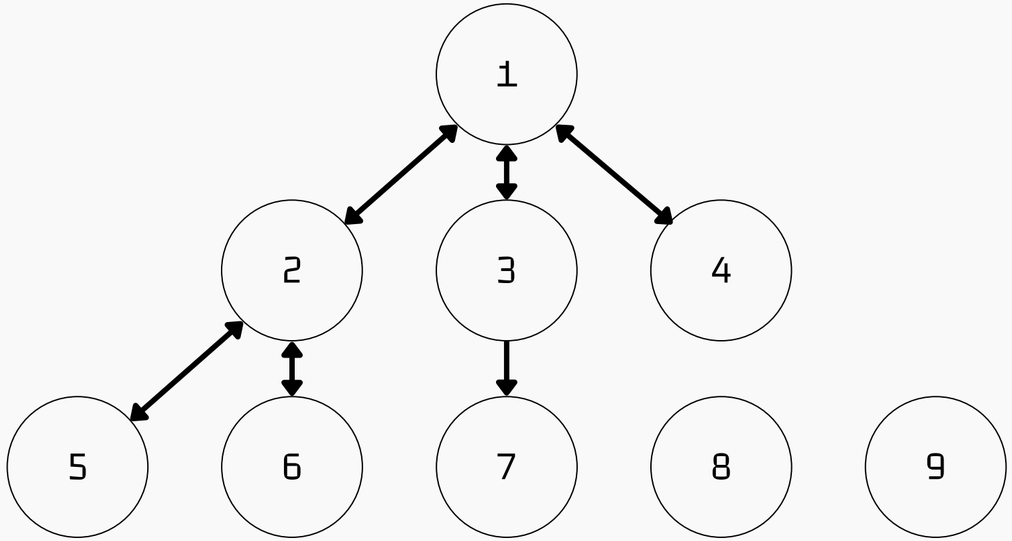
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1}
5: {2}
6: {2}
7: {}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

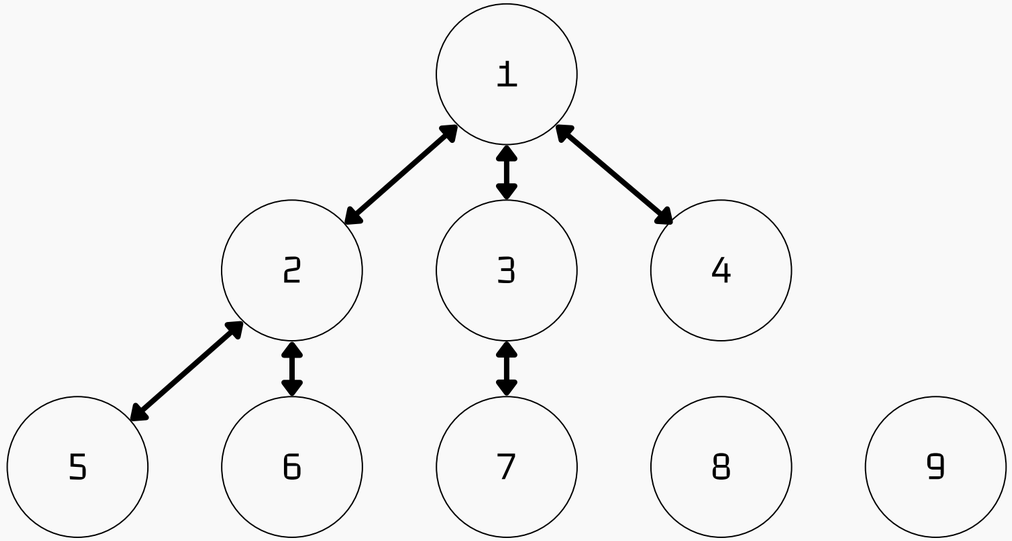
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1}
5: {2}
6: {2}
7: {3}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

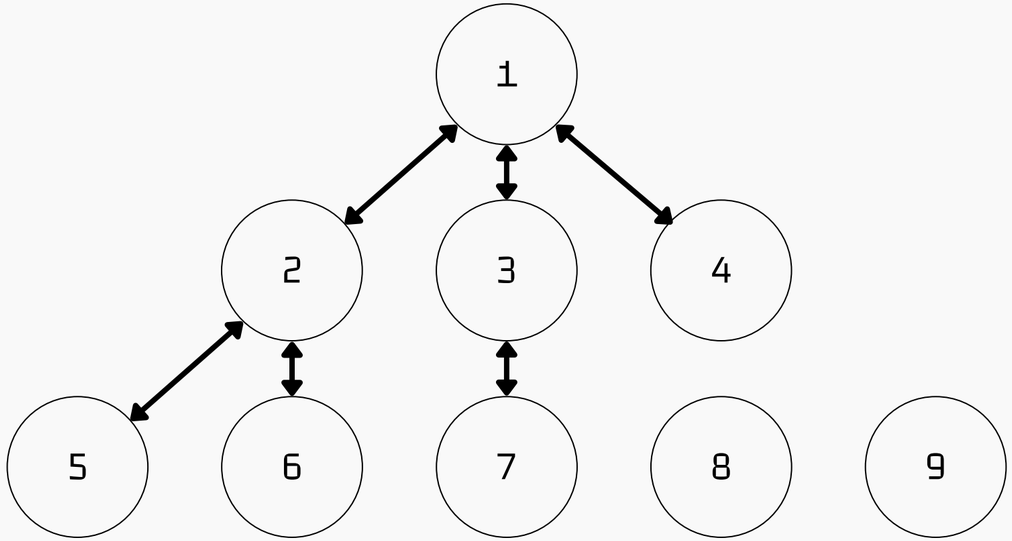
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1}
5: {2}
6: {2}
7: {3}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

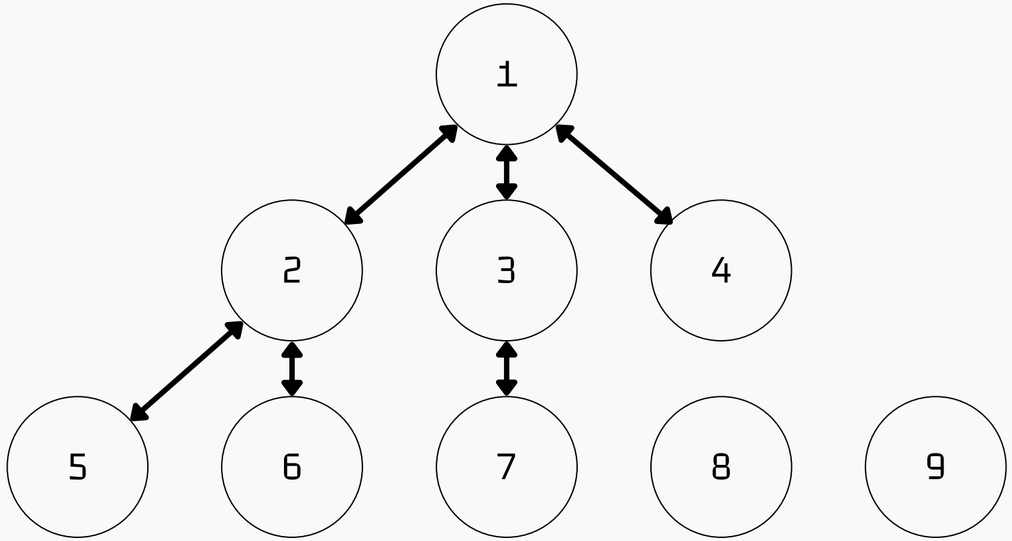
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1}
5: {2}
6: {2}
7: {3}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

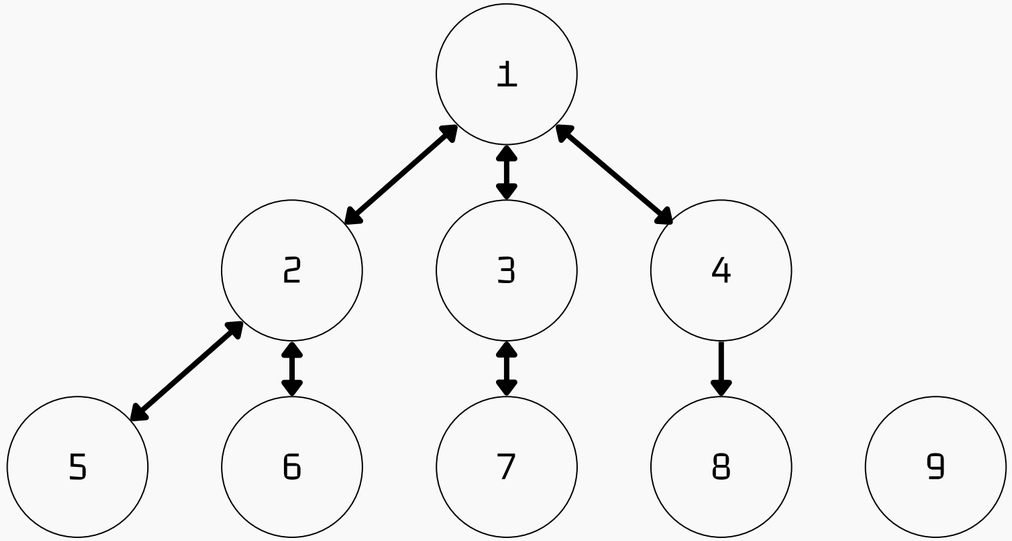
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8}
5: {2}
6: {2}
7: {3}
8: {}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

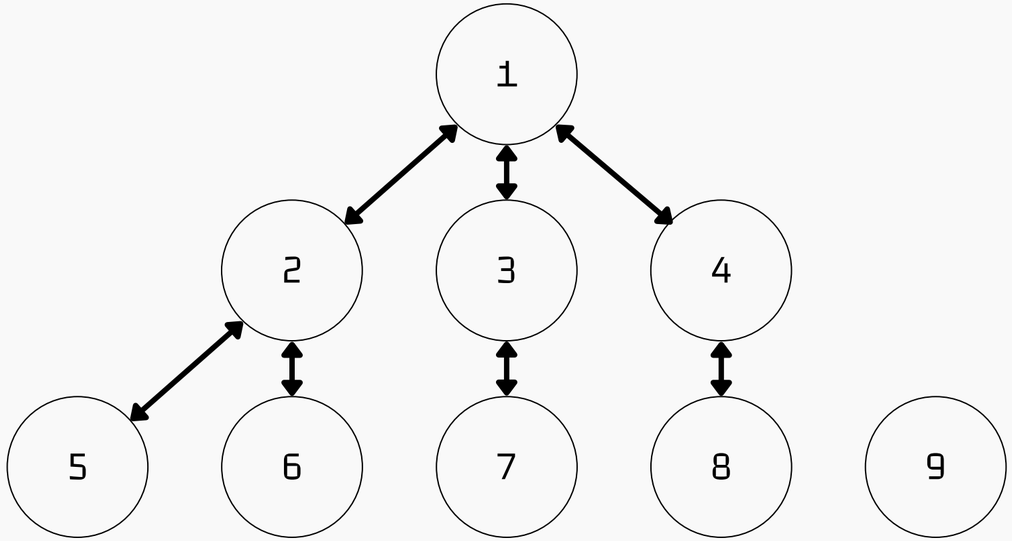
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8}
5: {2}
6: {2}
7: {3}
8: {4}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

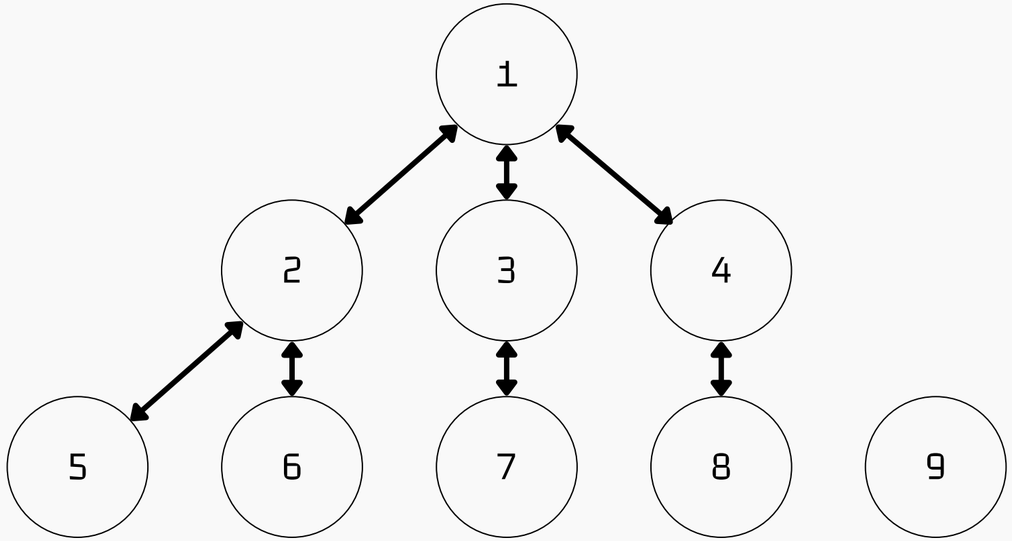
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8}
5: {2}
6: {2}
7: {3}
8: {4}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

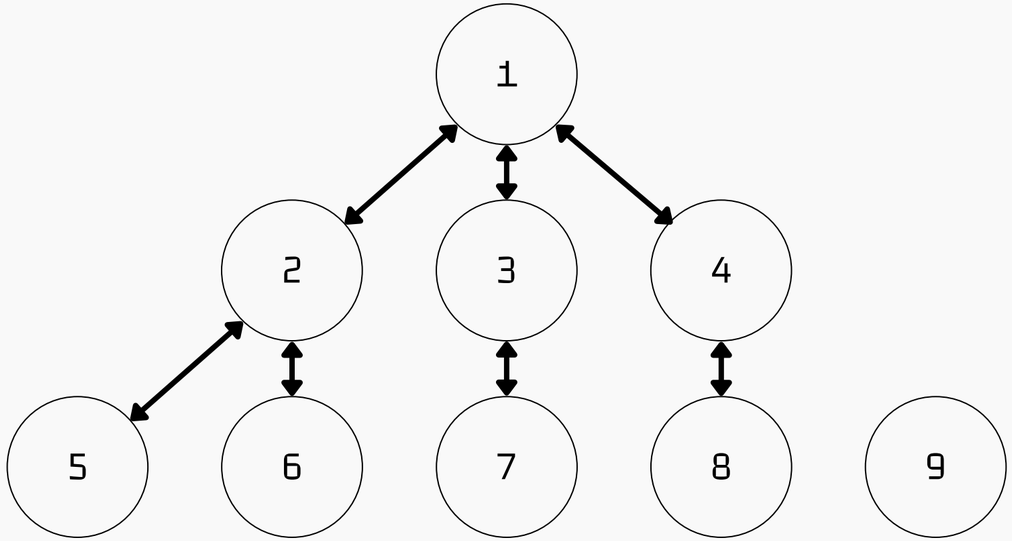
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8}
5: {2}
6: {2}
7: {3}
8: {4}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

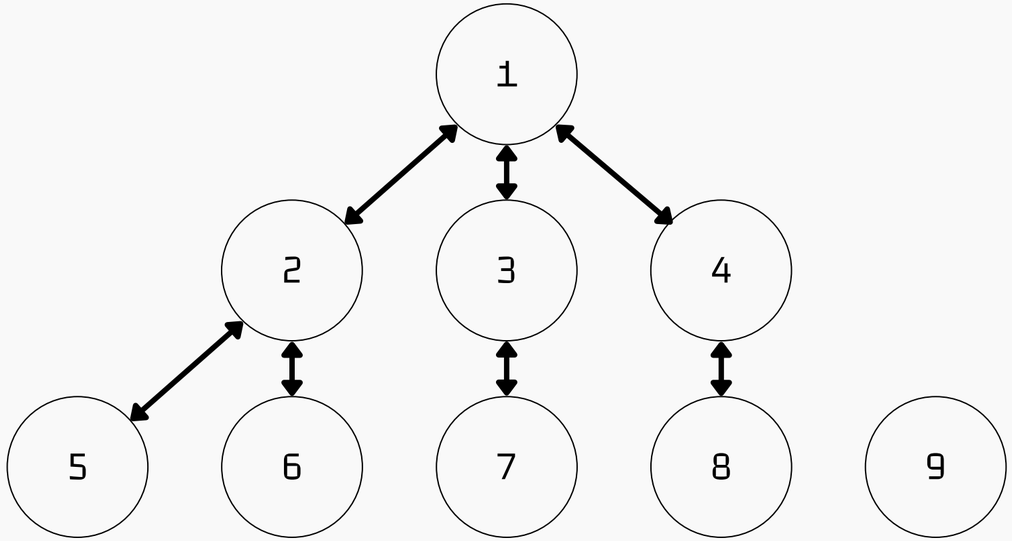
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8}
5: {2}
6: {2}
7: {3}
8: {4}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

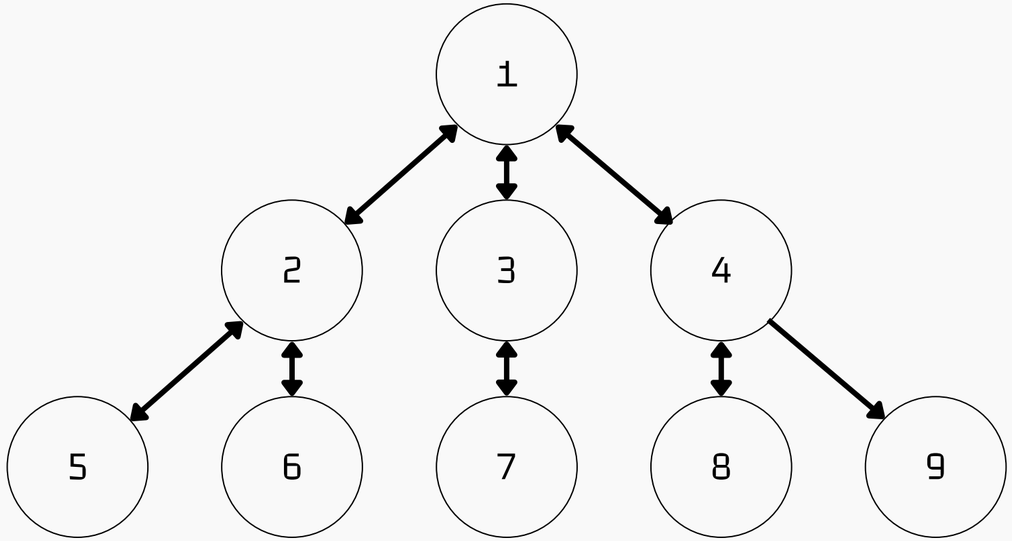
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

Lista de adyacencia

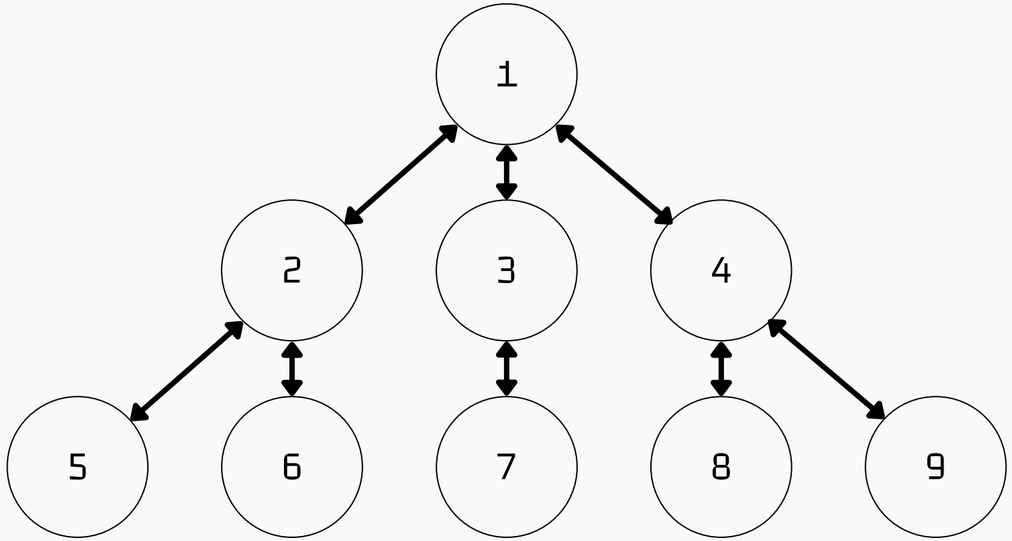
```
cin >> n >> m;
vector<int> adj[n+1];
for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

Entrada:

```
9 8
1 2
1 3
1 4
2 5
2 6
3 7
4 8
4 9
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

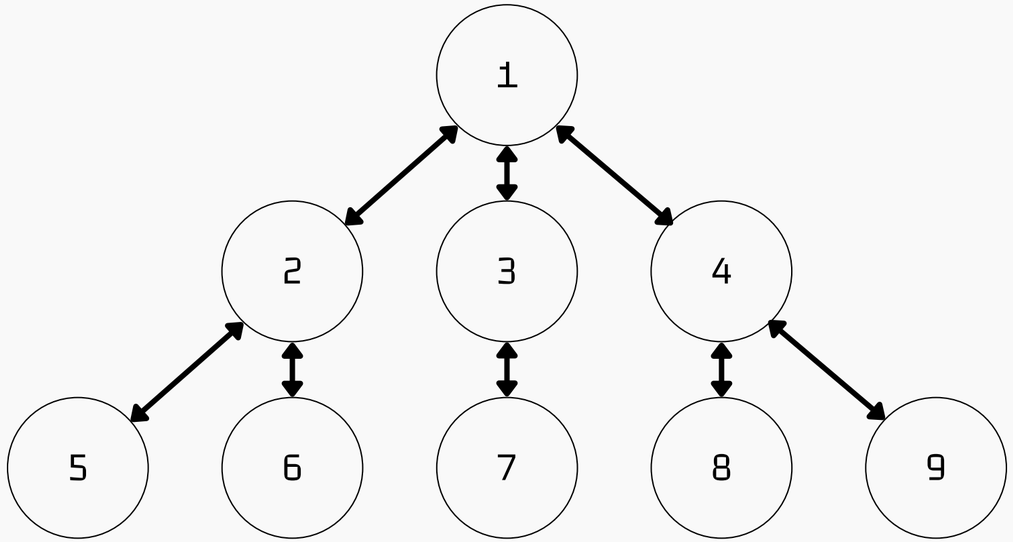
```cpp
int ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: false
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

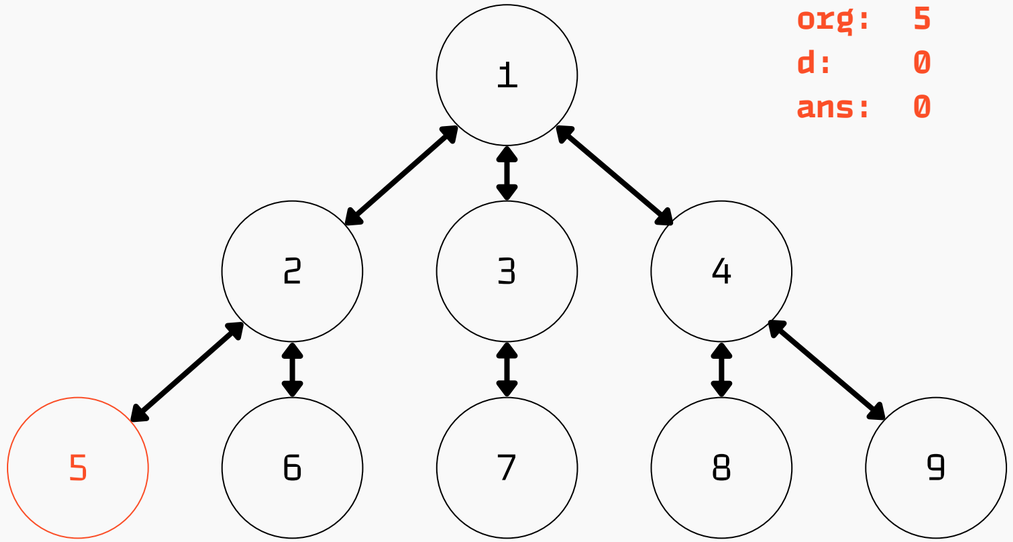
```
int ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: false
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

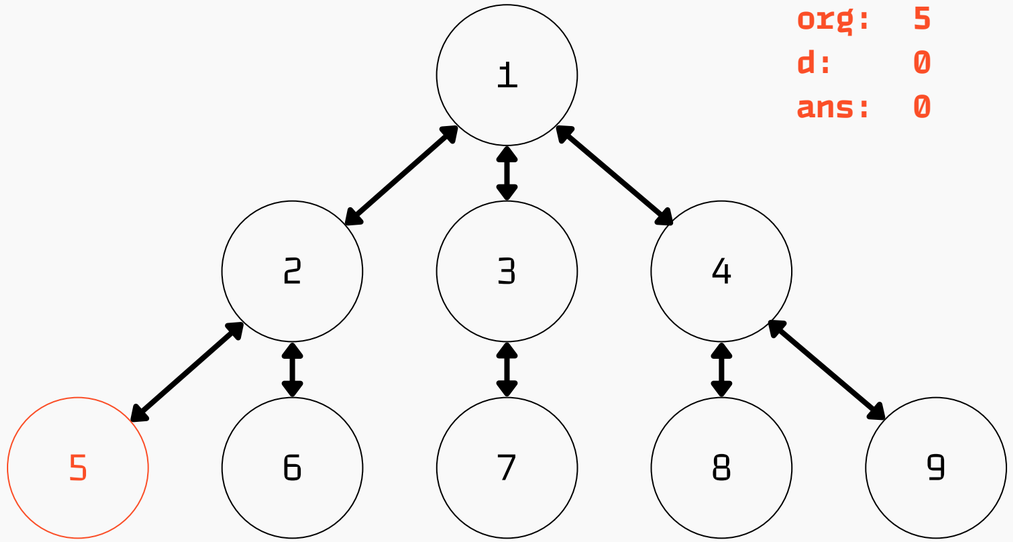
```
ans = 0;

void dfs[int org, int d]{
    ans = max[ans, d];
    vis[org] = true;
    for[auto node: adj[org]]{
        if[!vis[node]]{
            dfs[node, d+1];
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: false
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs[5, 0]`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

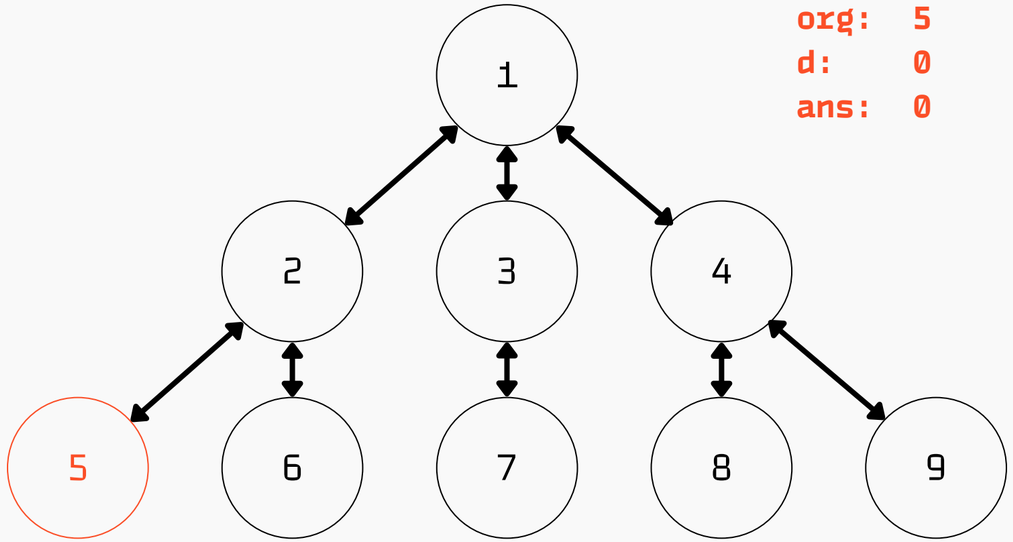
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

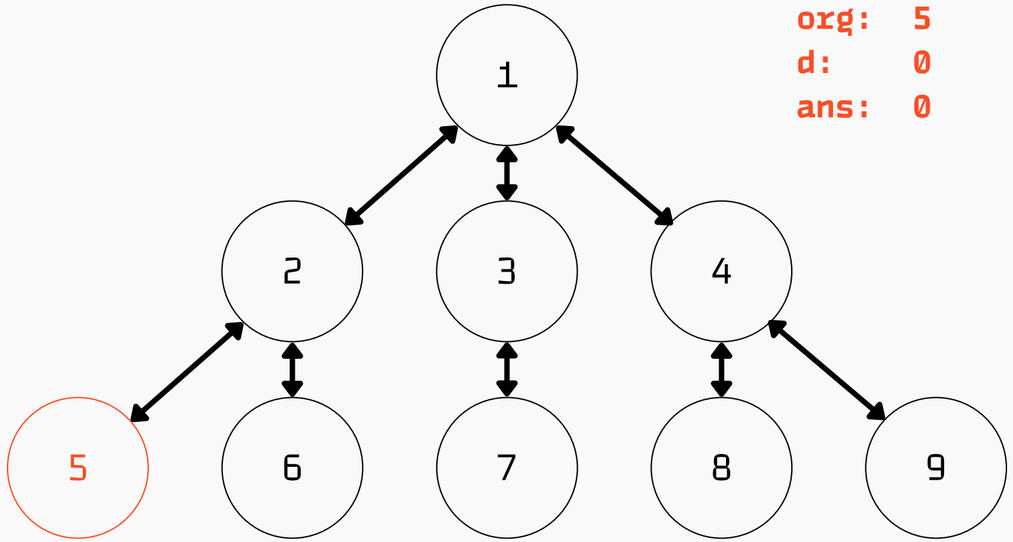
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

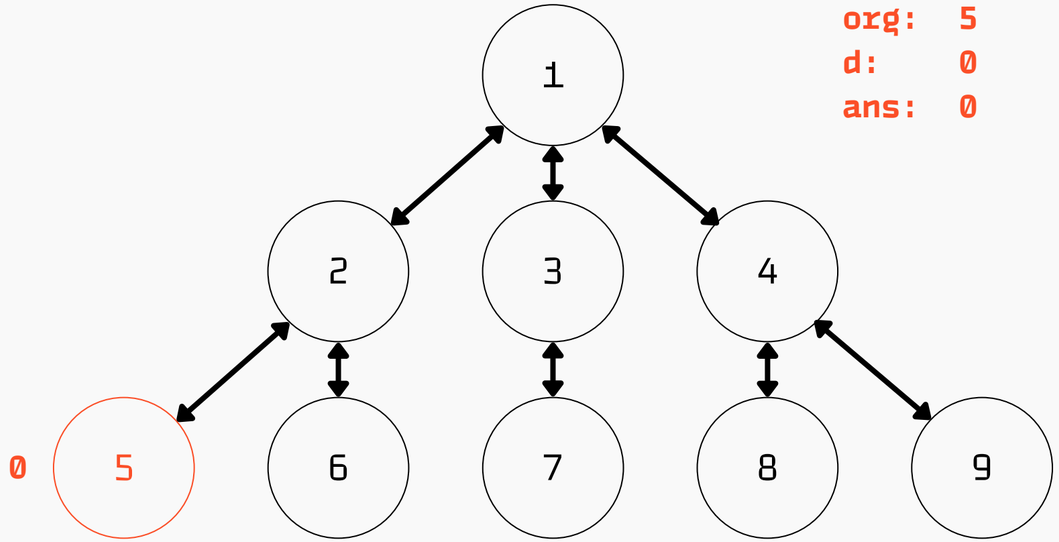
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

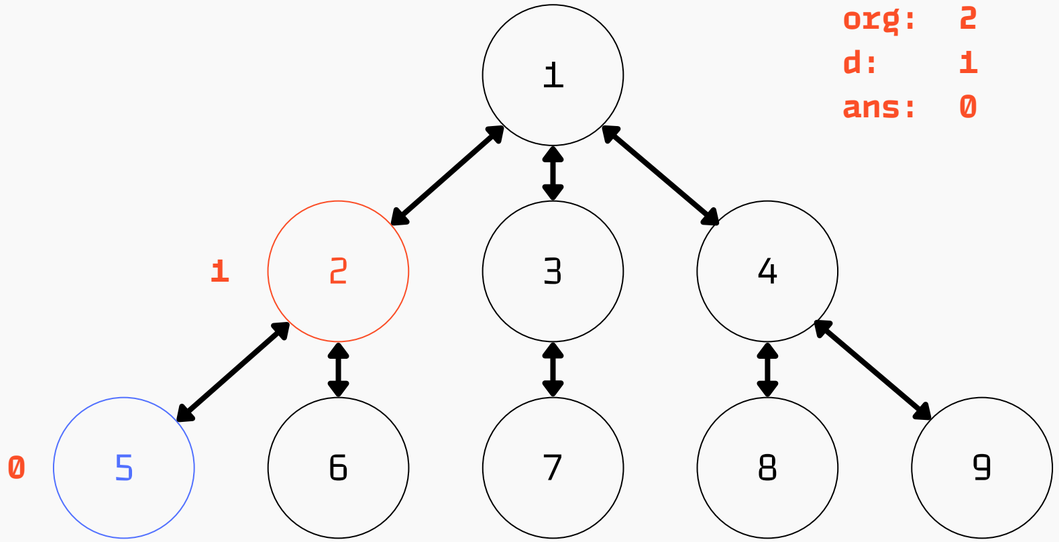
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

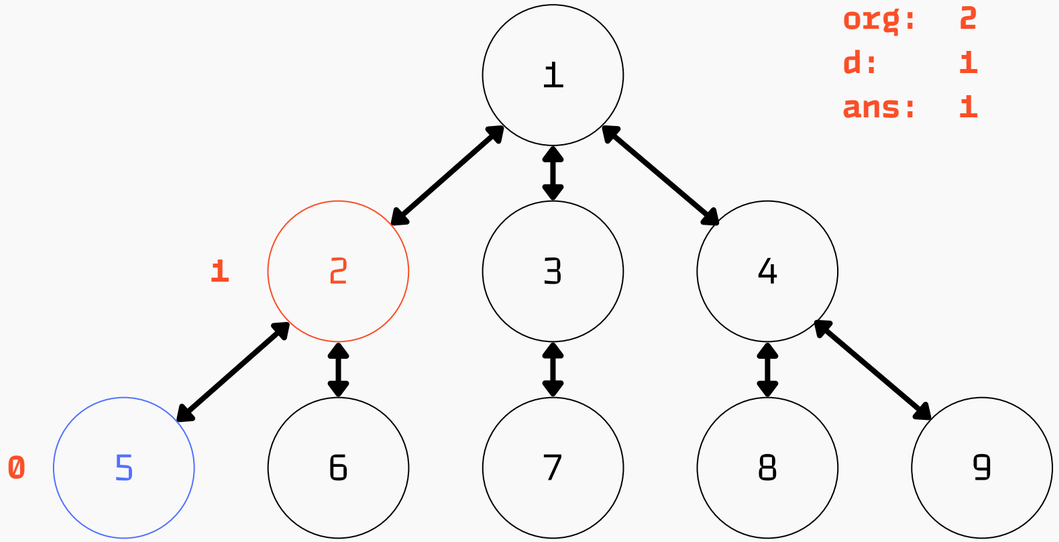
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: false
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

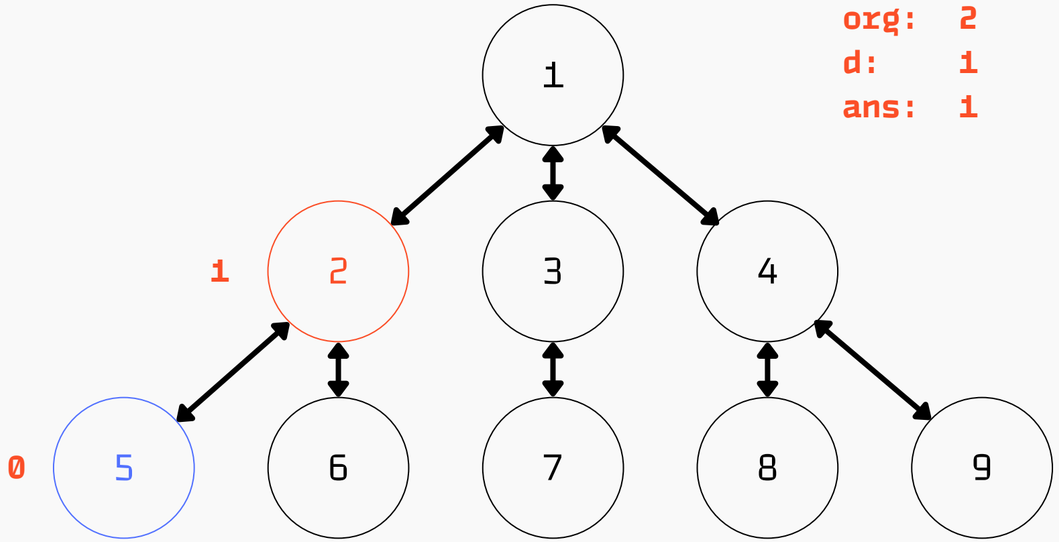
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

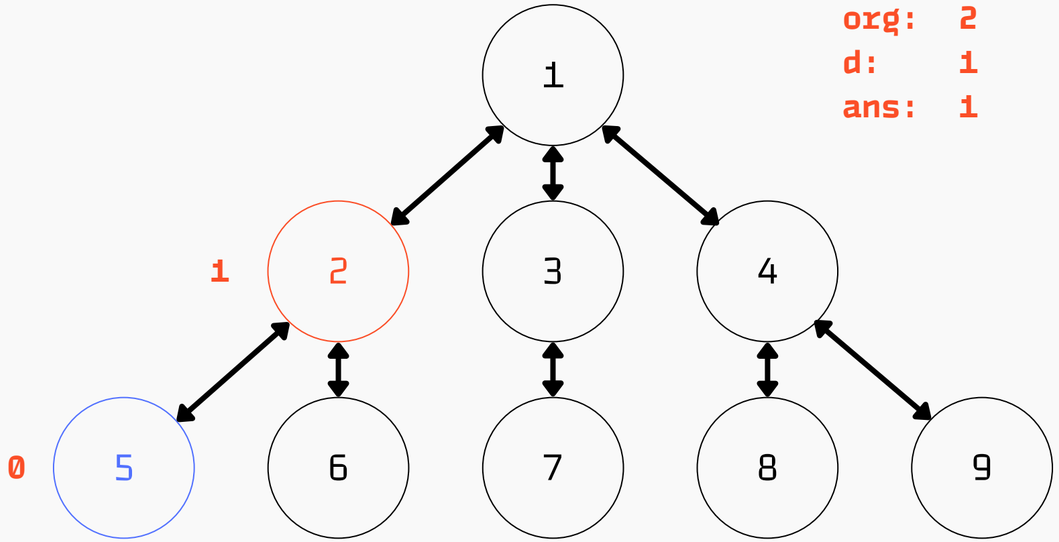
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

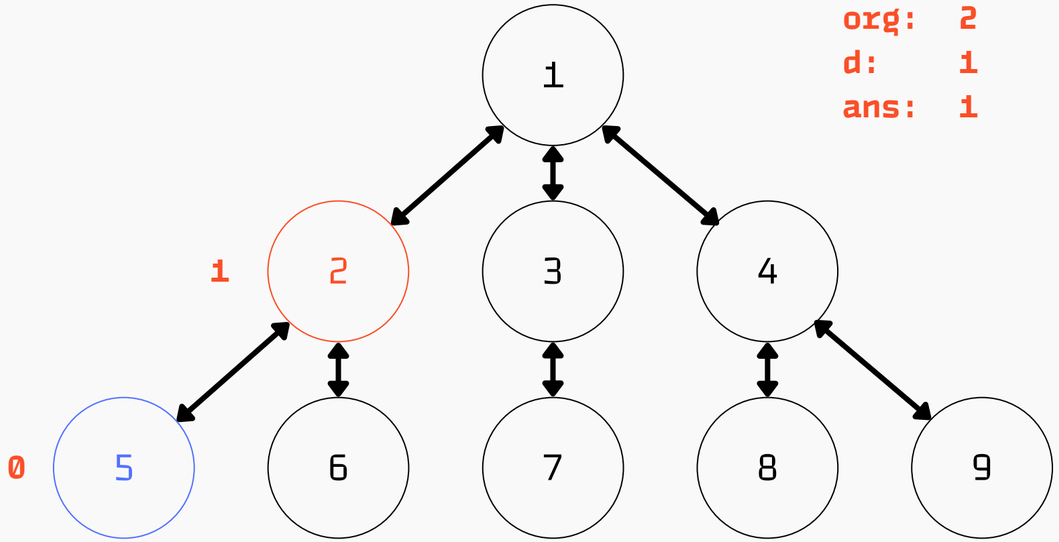
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

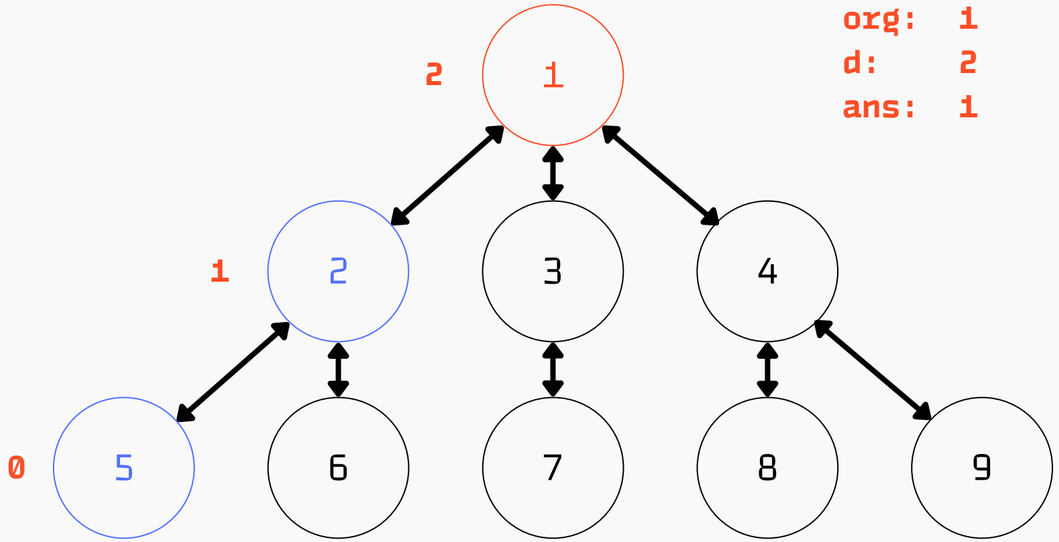
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

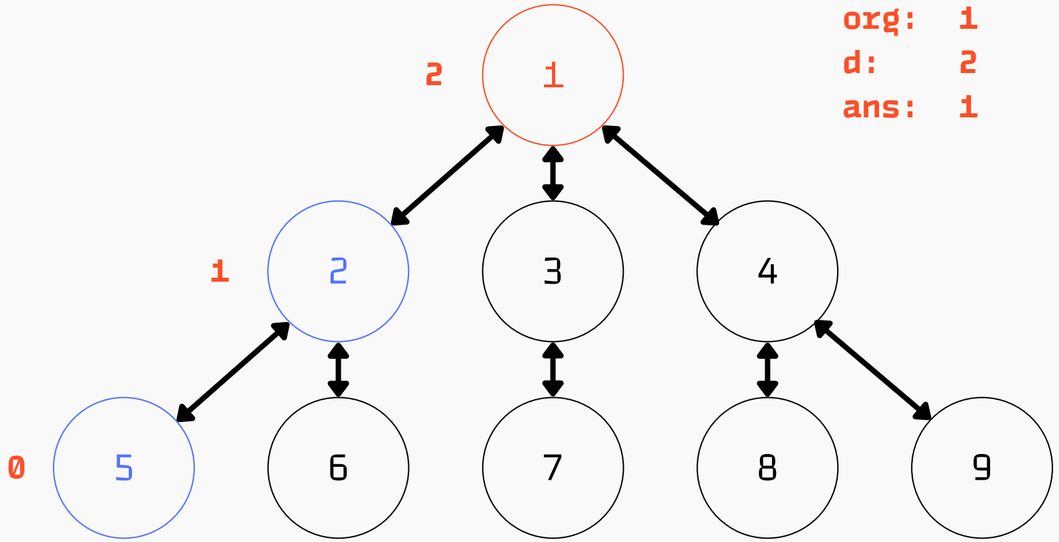
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: false
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

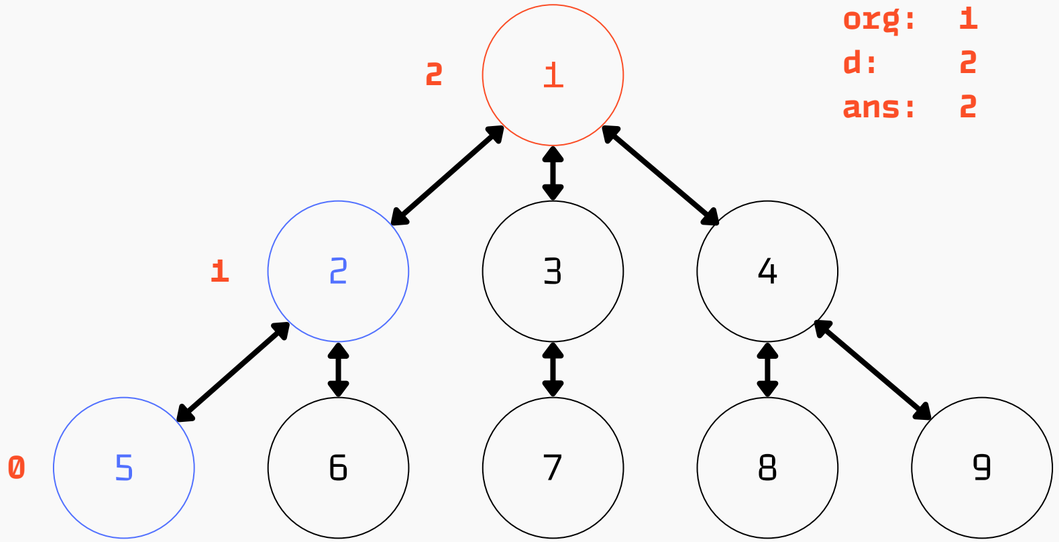
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: true
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

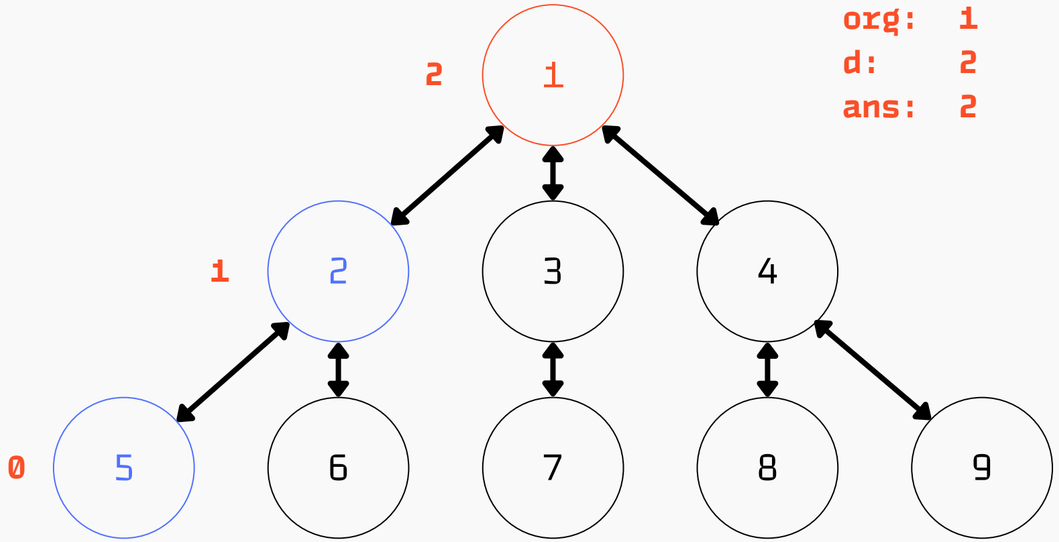
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: true
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

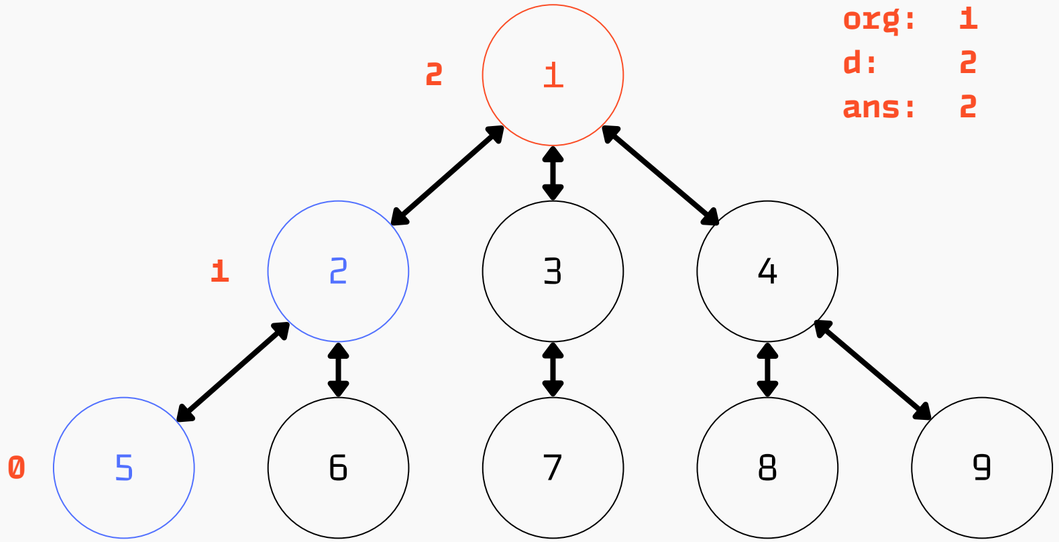
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: true
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

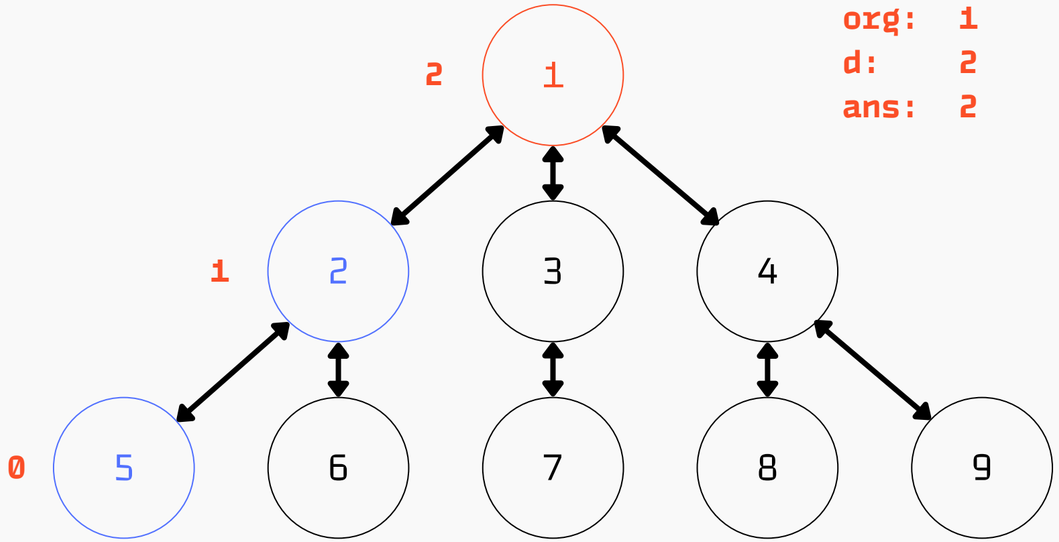
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: true
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

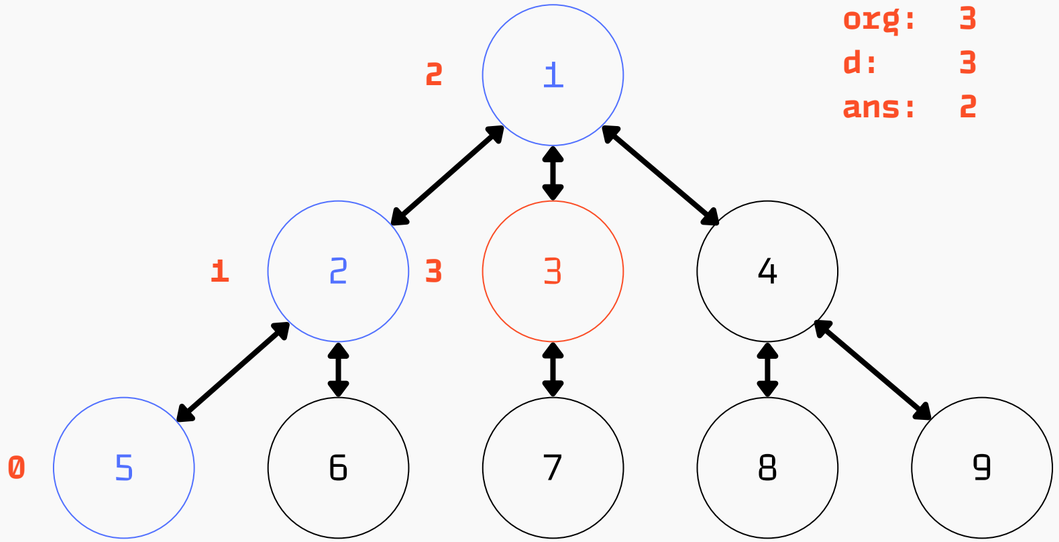
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: true
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

DFS: Busqueda en profundidad

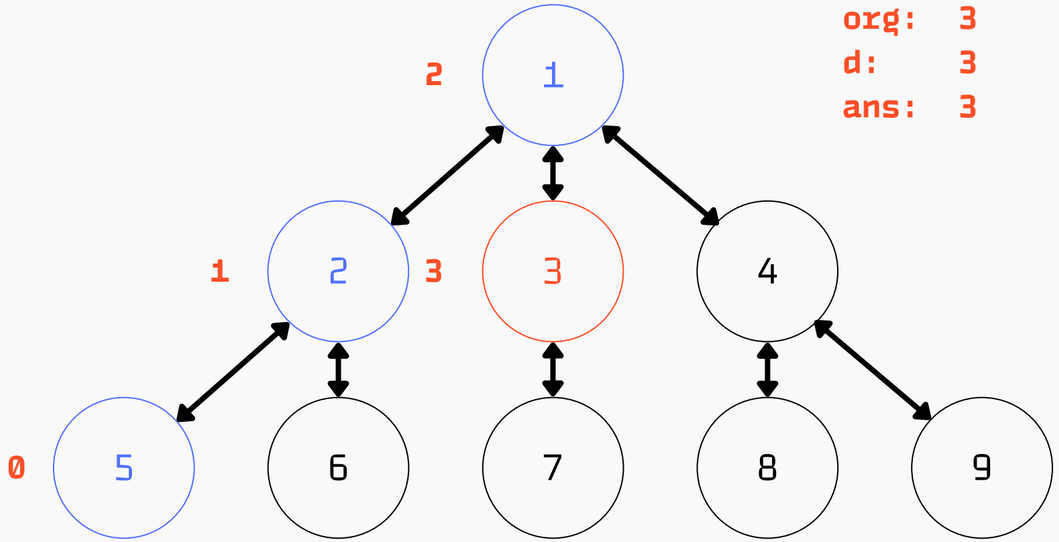
```
ans = 0;

void dfs(int org, int d){
    ans = max(ans, d);
    vis[org] = true;
    for(auto node: adj[org]){
        if(!vis[node]){
            dfs(node, d+1);
        }
    }
}
```

vis

```
0: false
1: true
2: true
3: false
4: false
5: true
6: false
7: false
8: false
9: false
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```

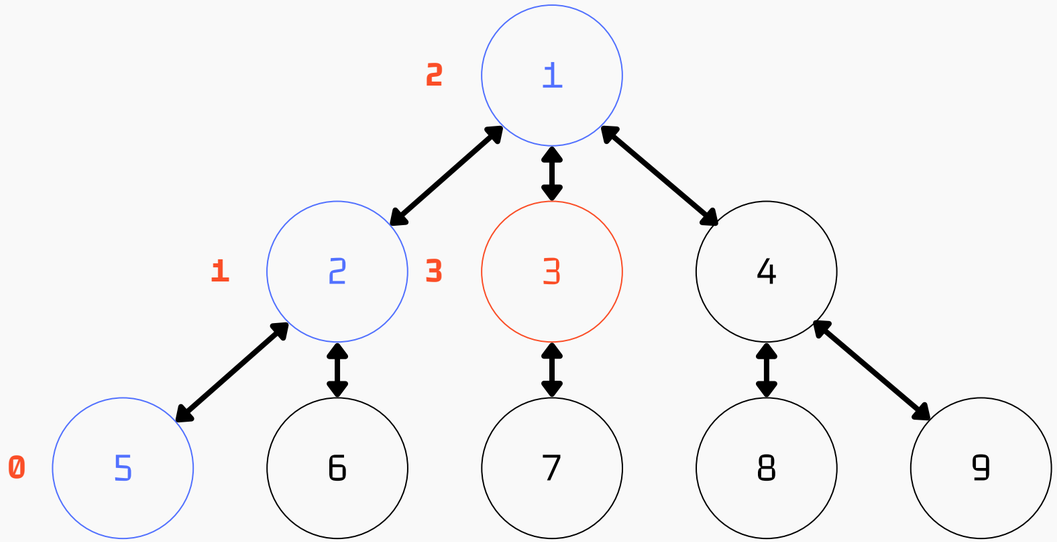


Con el grafo construido solo nos queda recorrerlo. La DFS es un recorrido recursivo que visita el primer nodo adyacente hasta que no es posible y luego retorna. Necesitamos llevar cuenta de los nodos visitados para no causar un ciclo infinito.

Supongamos que deseamos saber la distancia de la ciudad mas lejana a 5, llamaremos **`dfs(5, 0)`**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

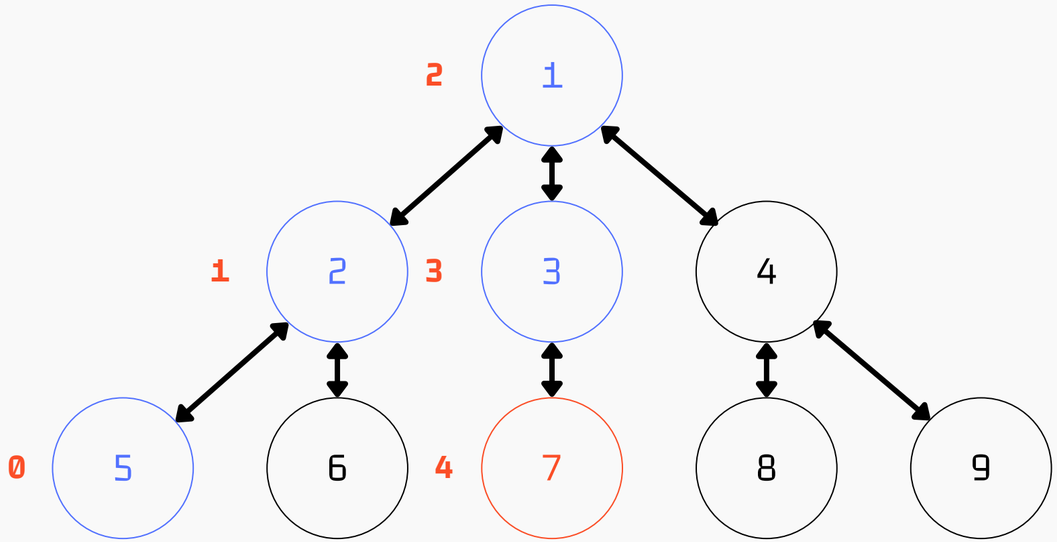
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

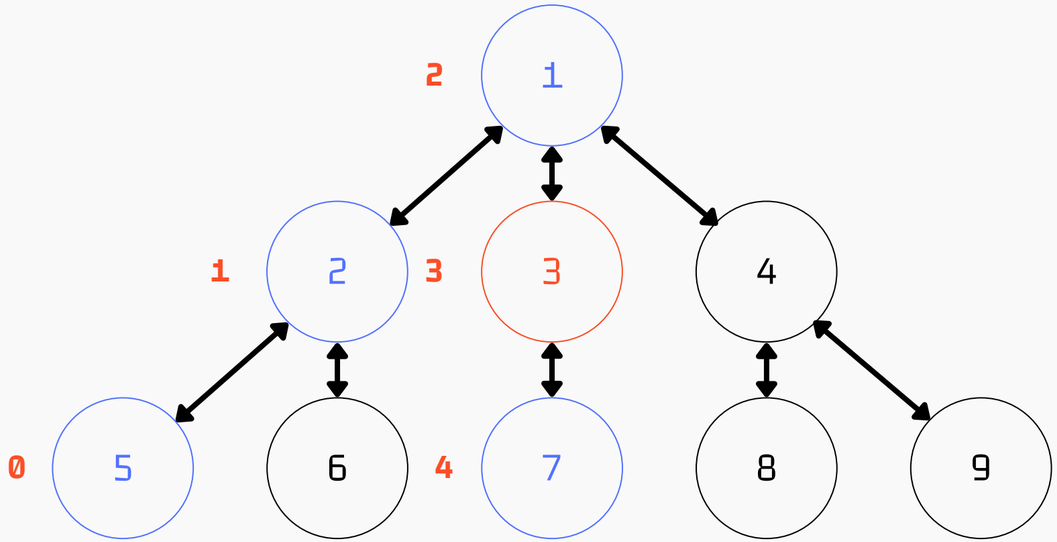
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

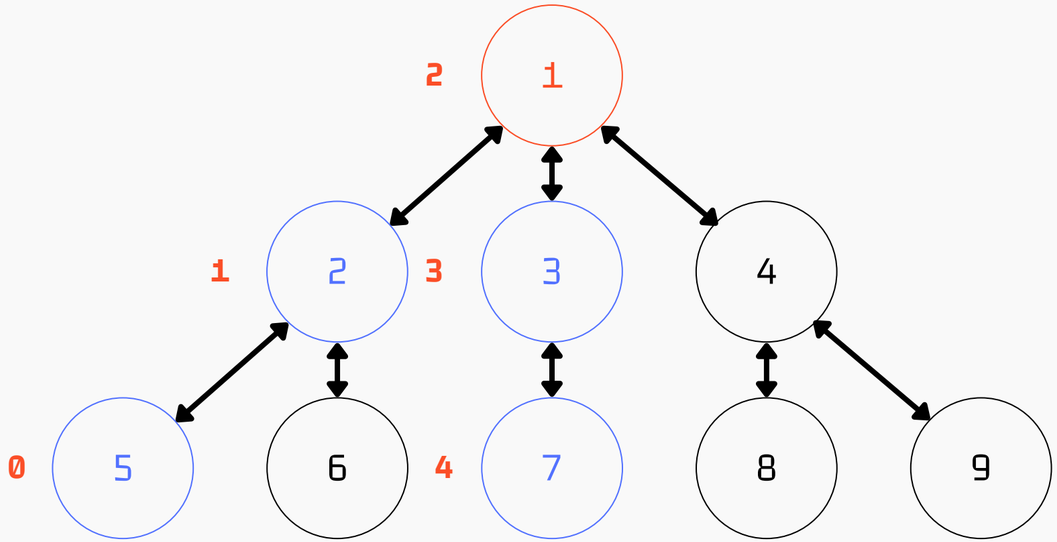
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

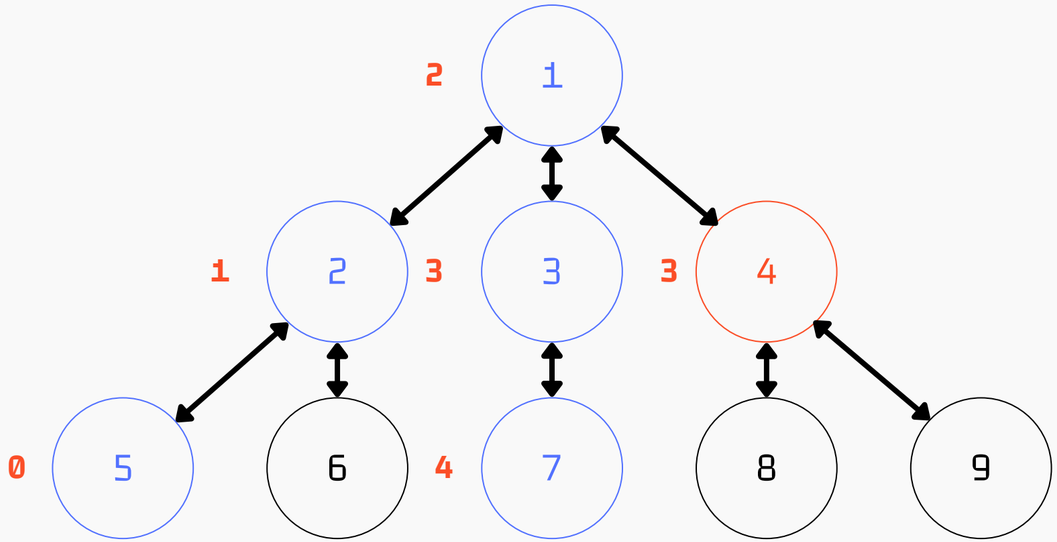
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

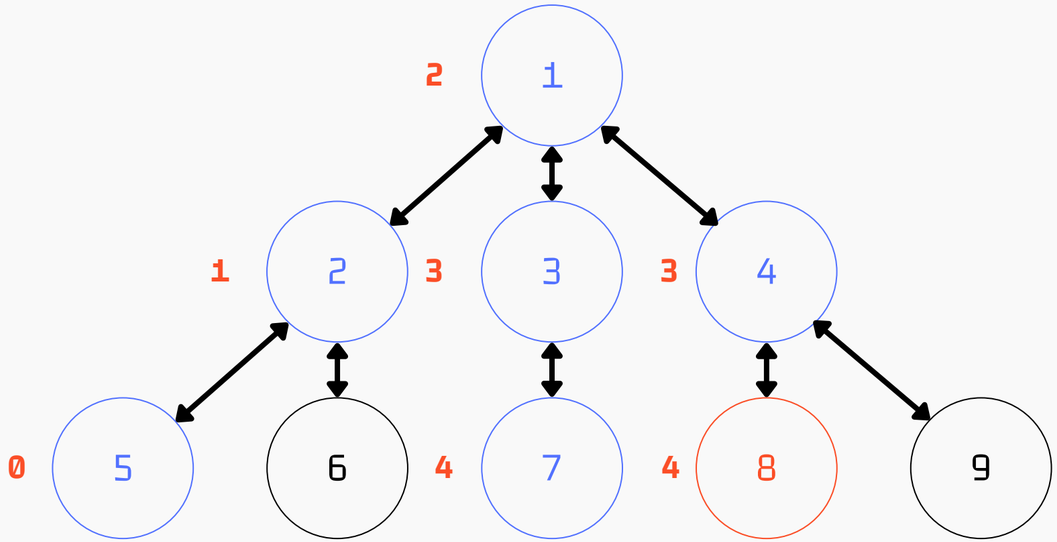
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

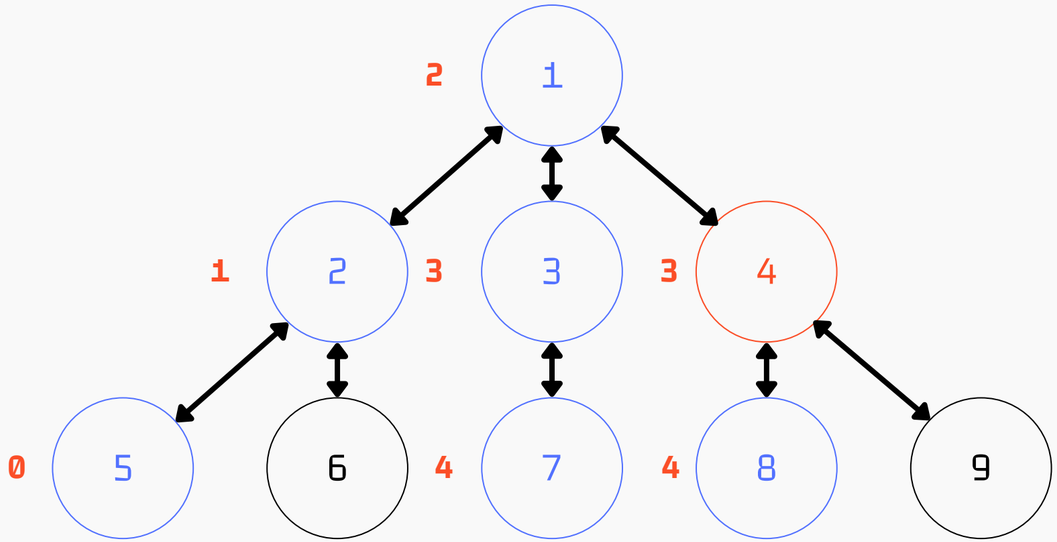
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

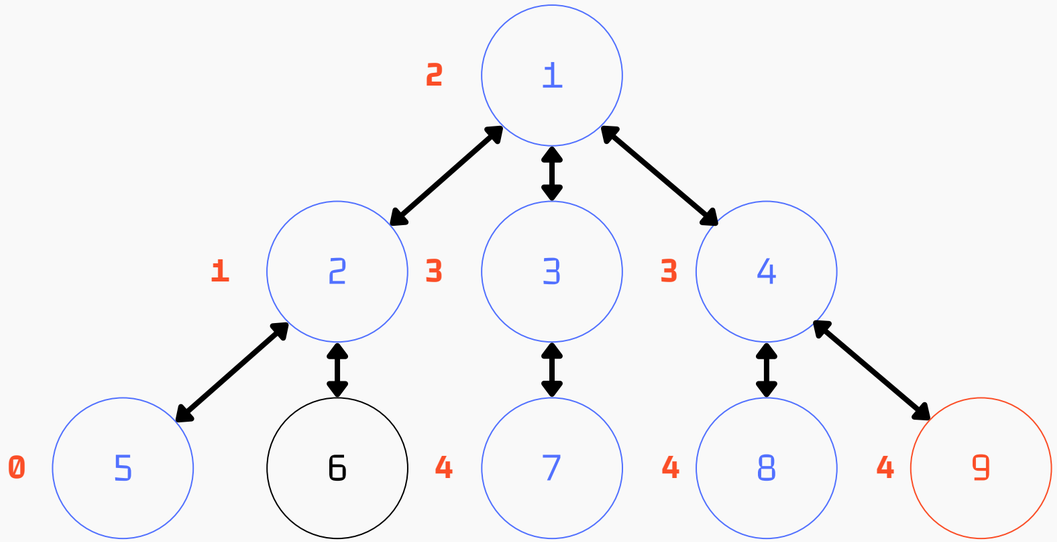
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

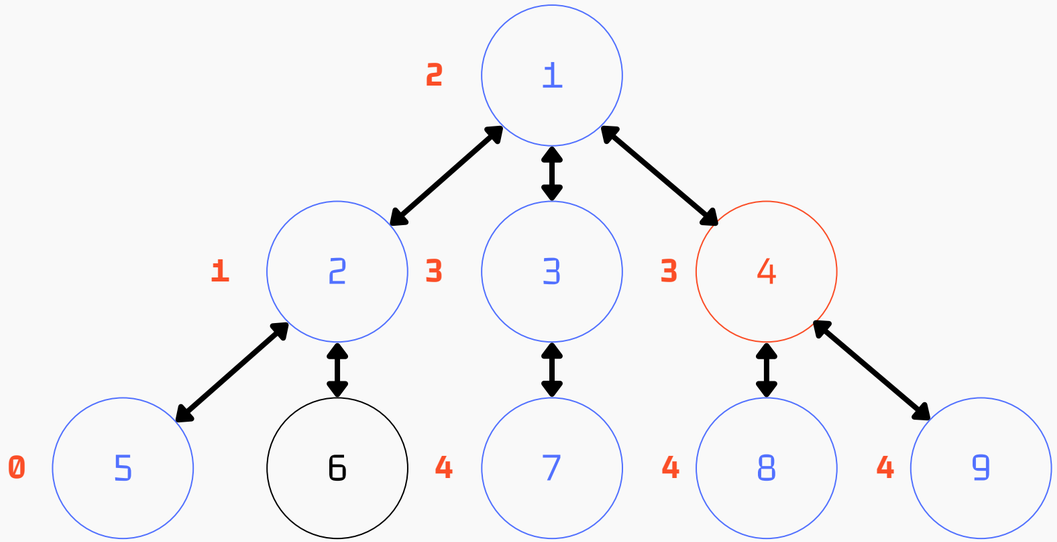
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

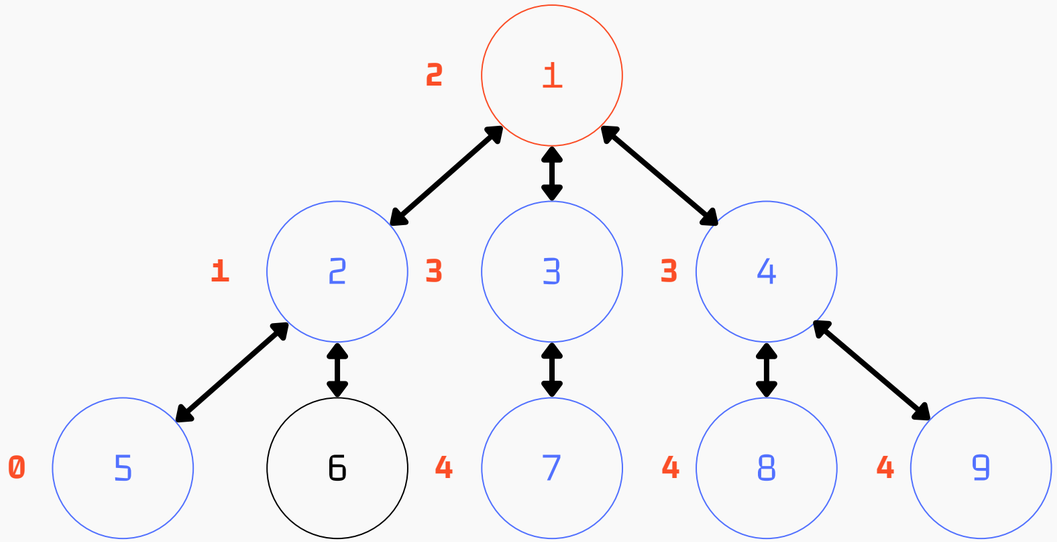
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

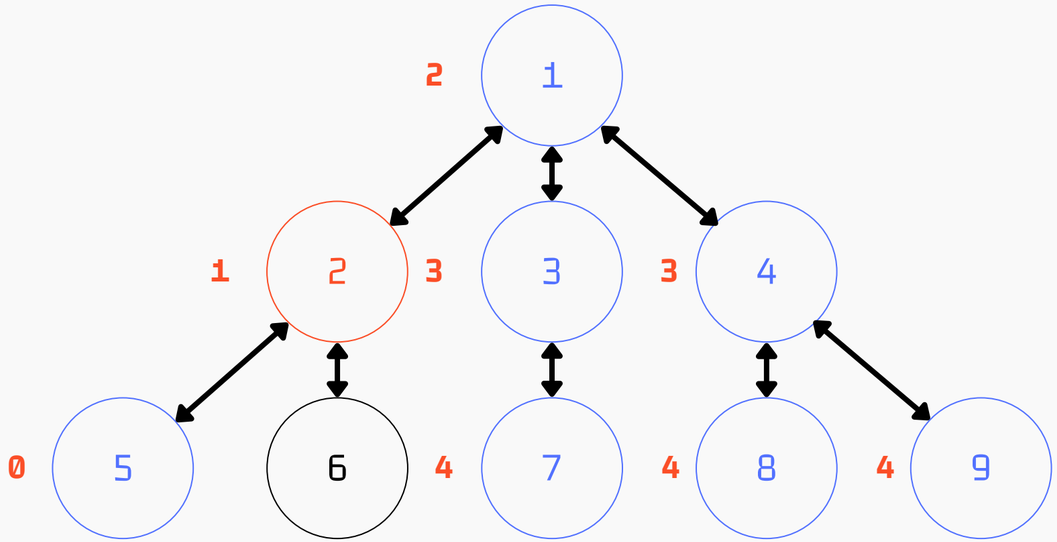
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

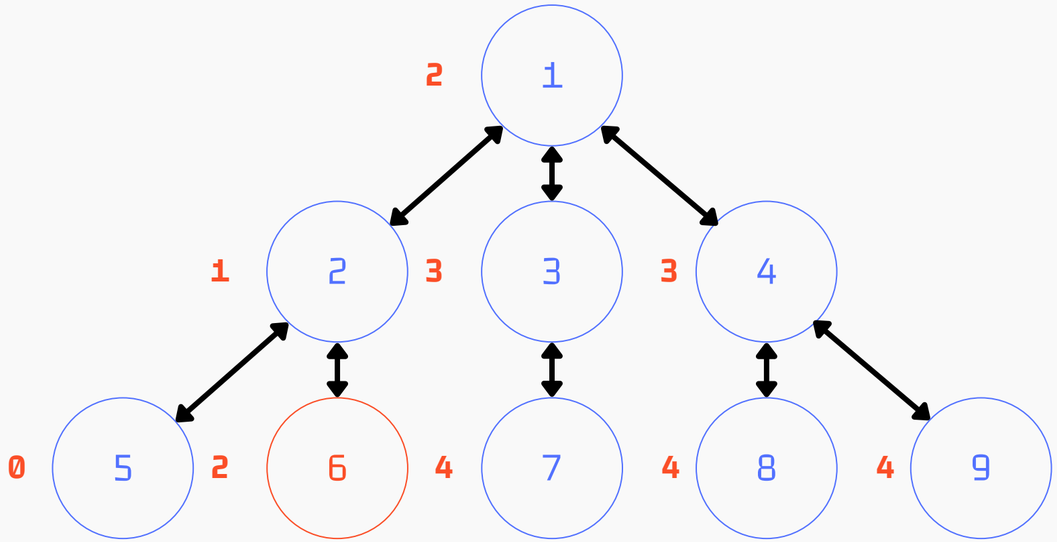
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

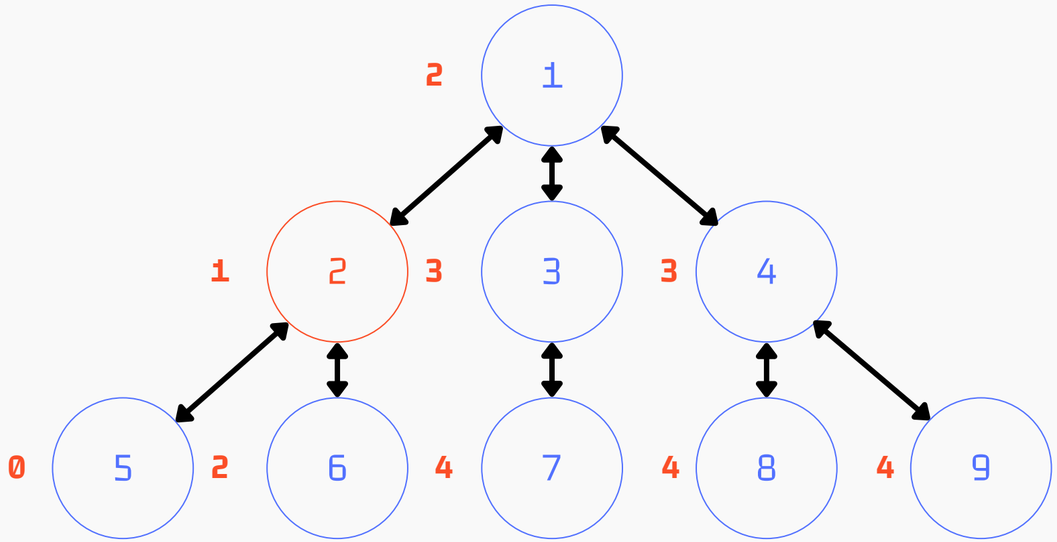
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

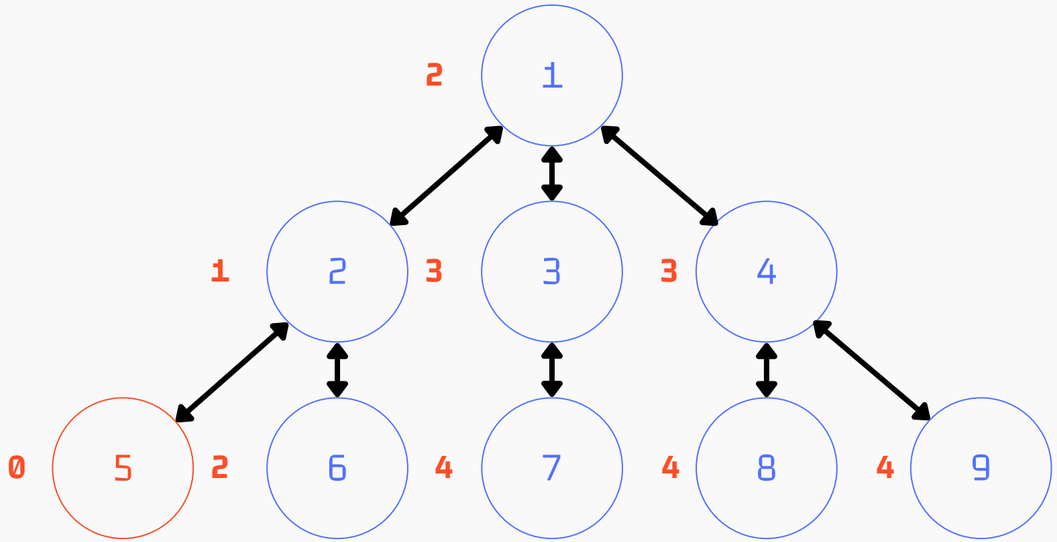
DFS: Busqueda en profundidad



Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

DFS: Busqueda en profundidad



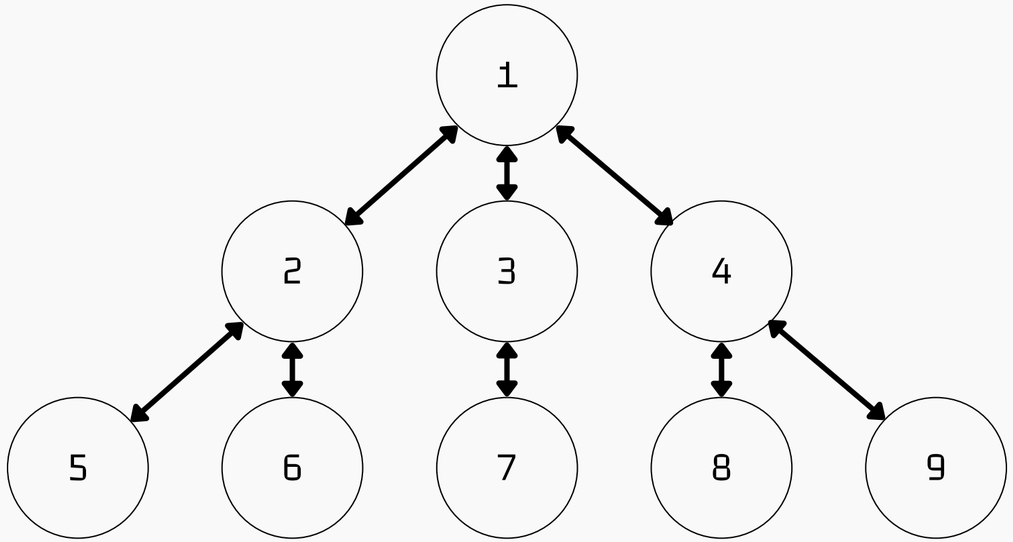
Eventualmente la DFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

```
ans = 0;

void bfs[int org]{
    queue<int> ord;
    vis[org] = 0;
    ord.push[org];

    while[!ord.emtpy[]]{
        int act = ord.front[];
        ord.pop[];
        for[auto node: adj[act]]{
            if[vis[node] == -1]{
                vis[node] = vis[act] + 1;
                ans = max[ans, vis[node]];
                ord.push[node];
            }
        }
    }
}
```

vis	adj
0: -1	0: {}
1: -1	1: {2, 3, 4}
2: -1	2: {1, 5, 6}
3: -1	3: {1, 7}
4: -1	4: {1, 8, 9}
5: -1	5: {2}
6: -1	6: {2}
7: -1	7: {3}
8: -1	8: {4}
9: -1	9: {4}



A diferencia de la DFS, la BFS visita primero todos los nodos adyacentes al actual y los coloca en una fila para recorrerlos en ese orden en la siguiente iteración. Podemos usar el arreglo de visitados para guardar la distancia de cada nodo, por lo que debemos inicializarlo en –1.

CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

```
ans = 0;

void bfs(int org){
    queue<int> ord;
    vis[org] = 0;
    ord.push(org);

    while(!ord.emtpy()){
        int act = ord.front();
        ord.pop();
        for(auto node: adj[act]){
            if(vis[node] == -1){
                vis[node] = vis[act] + 1;
                ans = max(ans, vis[node]);
                ord.push(node);
            }
        }
    }
}
```

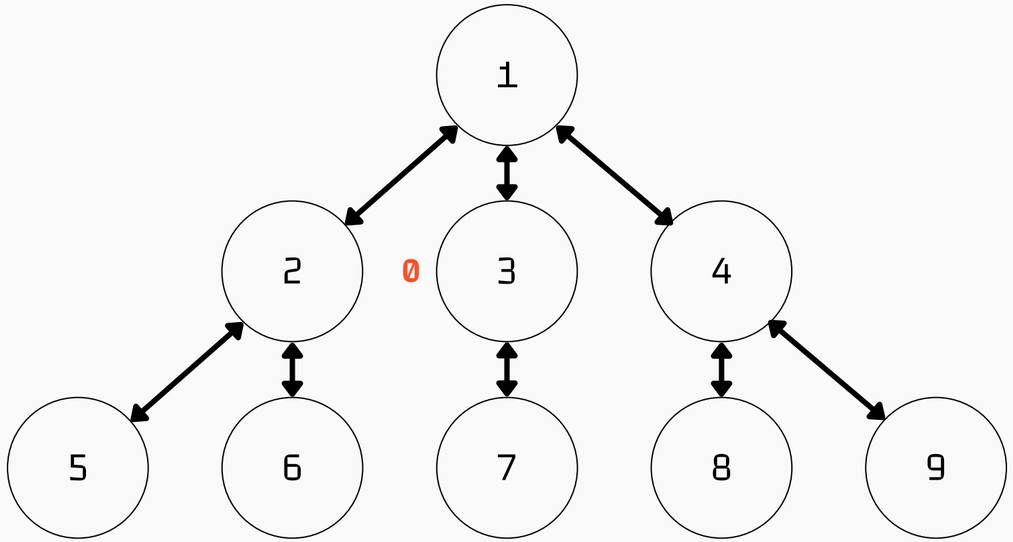
ord

vis

```
0: -1
1: -1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

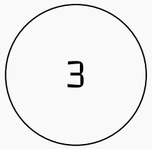
```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

ord



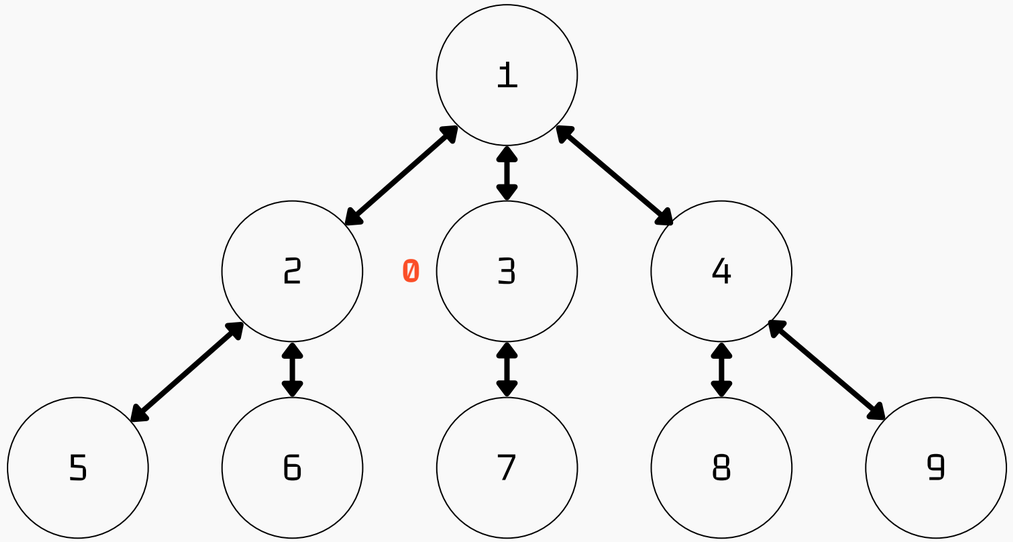

```
ans = 0;

void bfs(int org){
    queue<int> ord;
    vis[org] = 0;
    ord.push(org);

    while(!ord.emtpy()){
        int act = ord.front();
        ord.pop();
        for(auto node: adj[act]){
            if(vis[node] == -1){
                vis[node] = vis[act] + 1;
                ans = max(ans, vis[node]);
                ord.push(node);
            }
        }
    }
}
```

vis

```
0: -1
1: -1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

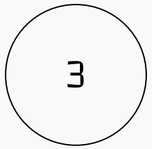
```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

ord



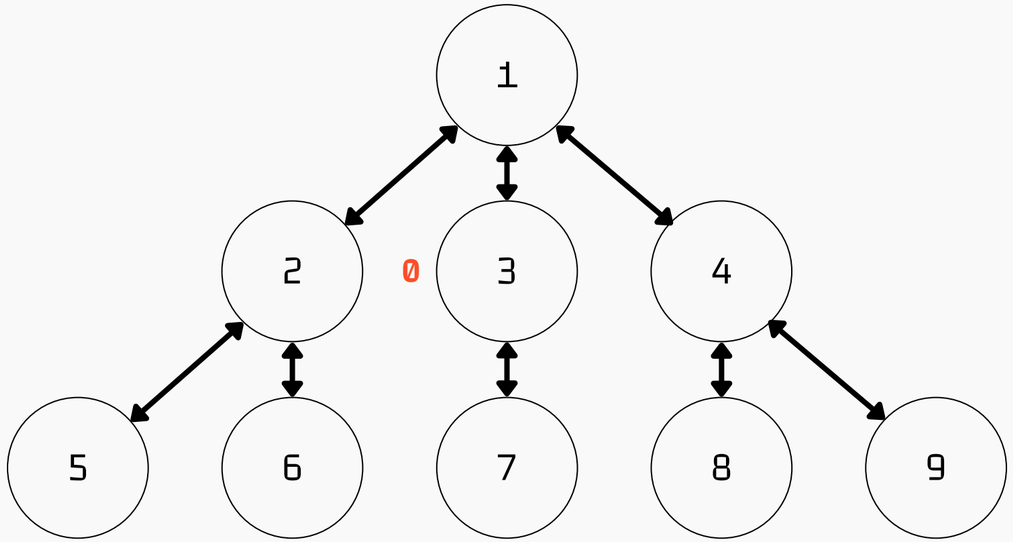

```
ans = 0;

void bfs(int org){
    queue<int> ord;
    vis[org] = 0;
    ord.push(org);

    while(!ord.emtpy()){
        int act = ord.front();
        ord.pop();
        for(auto node: adj[act]){
            if(vis[node] == -1){
                vis[node] = vis[act] + 1;
                ans = max(ans, vis[node]);
                ord.push(node);
            }
        }
    }
}
```

vis

```
0: -1
1: -1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

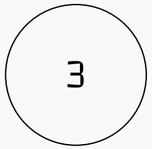
```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

ord



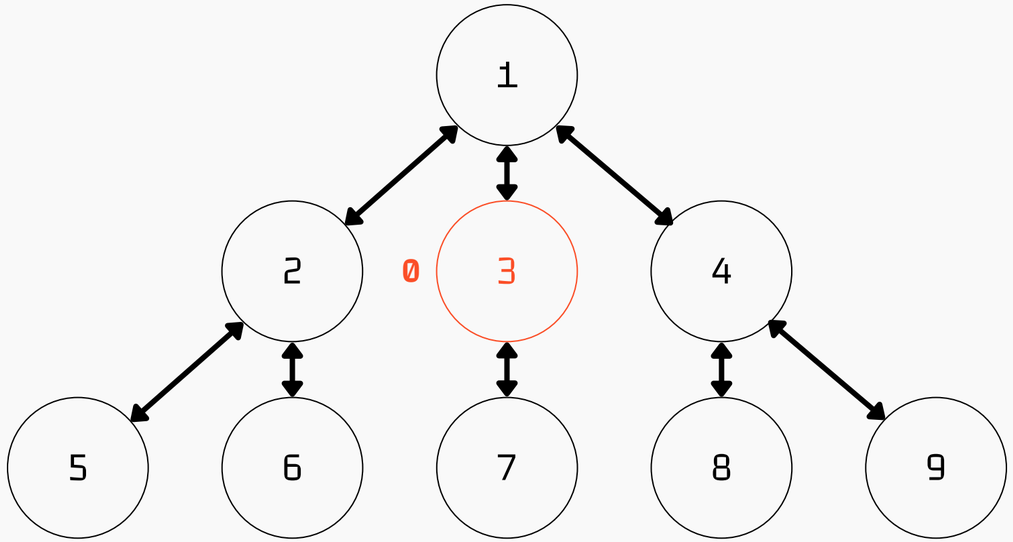

```
ans = 0;

void bfs[int org]{
    queue<int> ord;
    vis[org] = 0;
    ord.push[org];

    while[!ord.emtpy[]]{
        int act = ord.front[];
        ord.pop[];
        for[auto node: adj[act]]{
            if[vis[node] == -1]{
                vis[node] = vis[act] + 1;
                ans = max[ans, vis[node]];
                ord.push[node];
            }
        }
    }
}
```

vis

```
0: -1
1: -1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

```
ans = 0;

void bfs(int org){
    queue<int> ord;
    vis[org] = 0;
    ord.push(org);

    while(!ord.emtpy()){
        int act = ord.front();
        ord.pop();
        for(auto node: adj[act]){
            if(vis[node] == -1){
                vis[node] = vis[act] + 1;
                ans = max(ans, vis[node]);
                ord.push(node);
            }
        }
    }
}
```

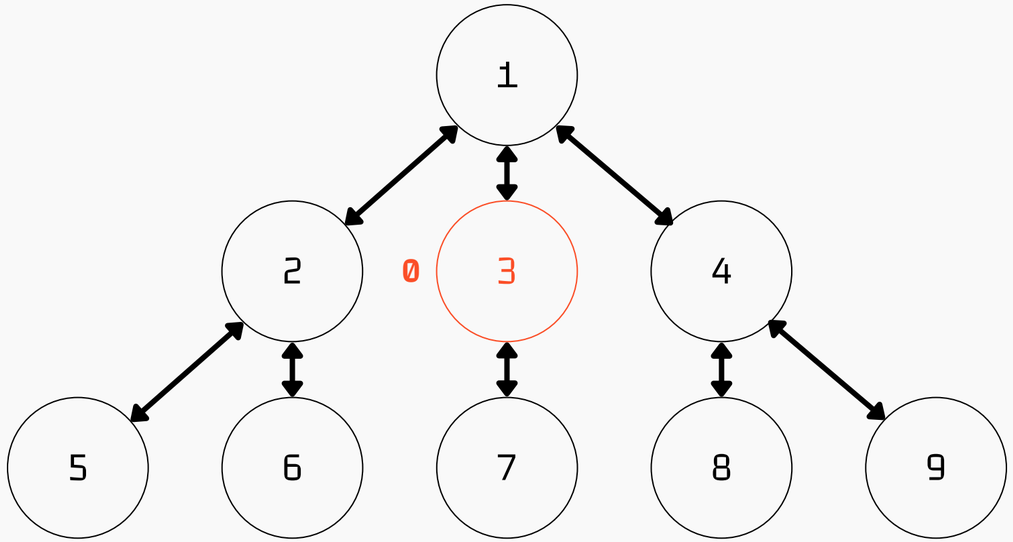
ord

vis

```
0: -1
1: -1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

```
ans = 0;

void bfs(int org){
    queue<int> ord;
    vis[org] = 0;
    ord.push(org);

    while(!ord.emtpy()){
        int act = ord.front();
        ord.pop();
        for(auto node: adj[act]){
            if(vis[node] == -1){
                vis[node] = vis[act] + 1;
                ans = max(ans, vis[node]);
                ord.push(node);
            }
        }
    }
}
```

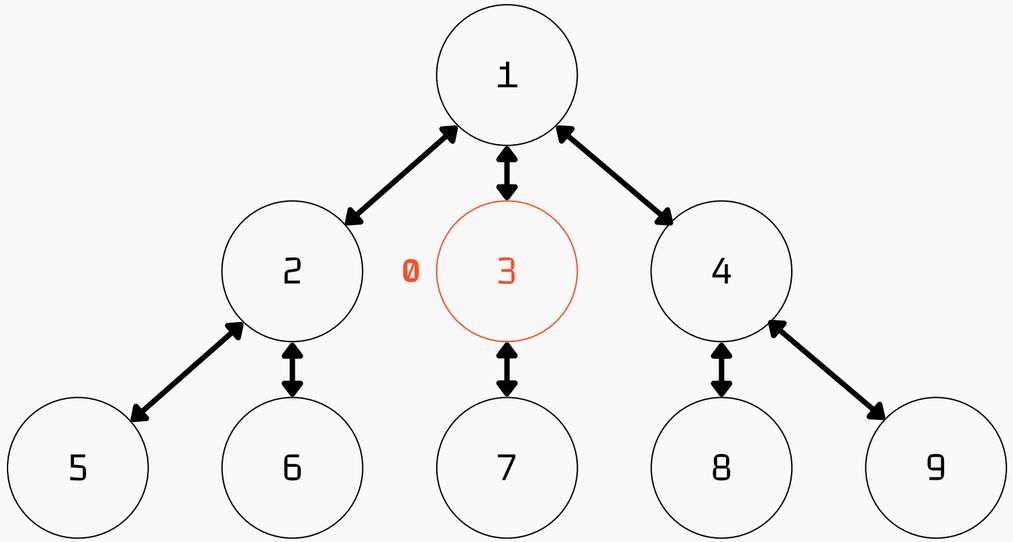
ord

vis

```
0: -1
1: -1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

```
ans = 0;

void bfs[int org]{
    queue<int> ord;
    vis[org] = 0;
    ord.push[org];

    while[!ord.emtpy[]]{
        int act = ord.front[];
        ord.pop[];
        for[auto node: adj[act]]{
            if[vis[node] == -1]{
                vis[node] = vis[act] + 1;
                ans = max[ans, vis[node]];
                ord.push[node];
            }
        }
    }
}
```

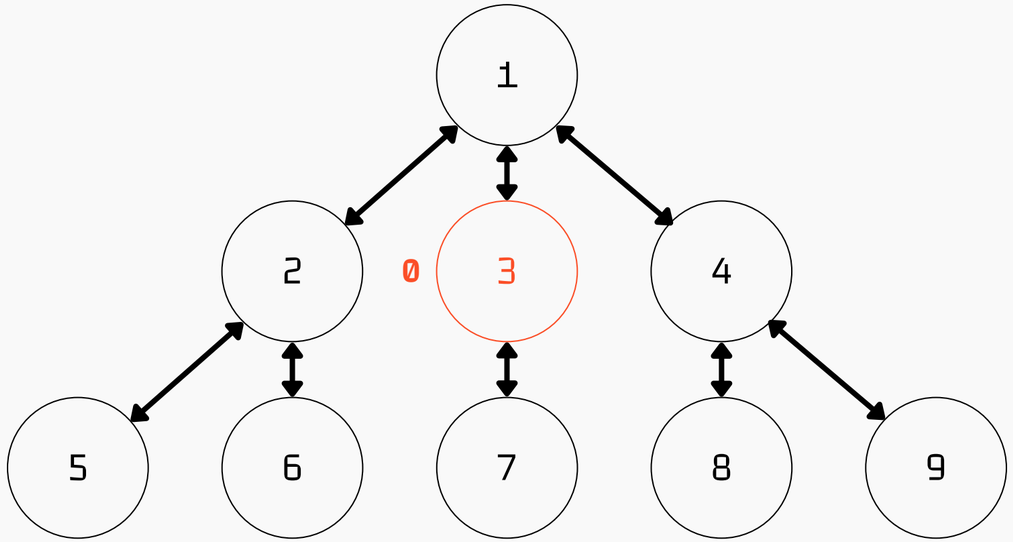
ord

vis

```
0: -1
1: -1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

```
ans = 0;

void bfs(int org){
    queue<int> ord;
    vis[org] = 0;
    ord.push(org);

    while(!ord.emtpy()){
        int act = ord.front();
        ord.pop();
        for(auto node: adj[act]){
            if(vis[node] == -1){
                vis[node] = vis[act] + 1;
                ans = max(ans, vis[node]);
                ord.push(node);
            }
        }
    }
}
```

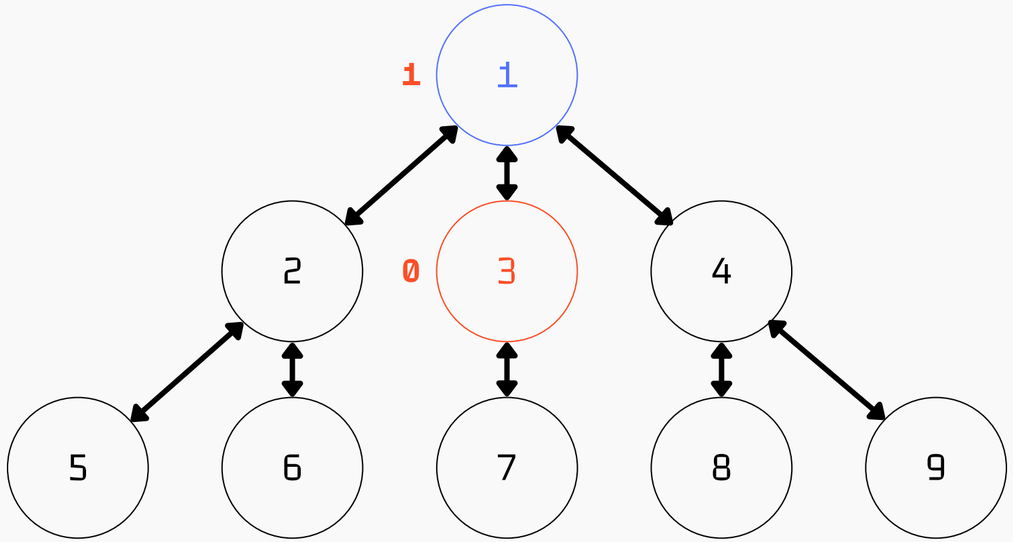
ord

vis

```
0: -1
1: 1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



CLUB DE PROGRAMACIóN COMPETITIVA UADY

BFS: Busqueda en anchura

```
ans = 0;

void bfs[int org]{
    queue<int> ord;
    vis[org] = 0;
    ord.push[org];

    while[!ord.emtpy[]]{
        int act = ord.front[];
        ord.pop[];
        for[auto node: adj[act]]{
            if[vis[node] == -1]{
                vis[node] = vis[act] + 1;
                ans = max[ans, vis[node]];
                ord.push[node];
            }
        }
    }
}
```

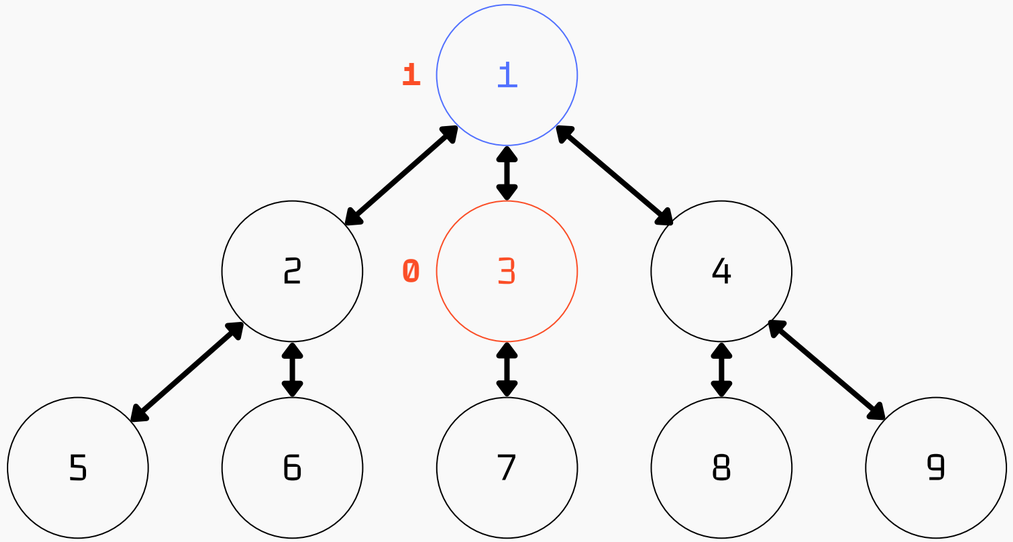
ord

vis

```
0: -1
1: 1
2: -1
3: 0
4: -1
5: -1
6: -1
7: -1
8: -1
9: -1
```

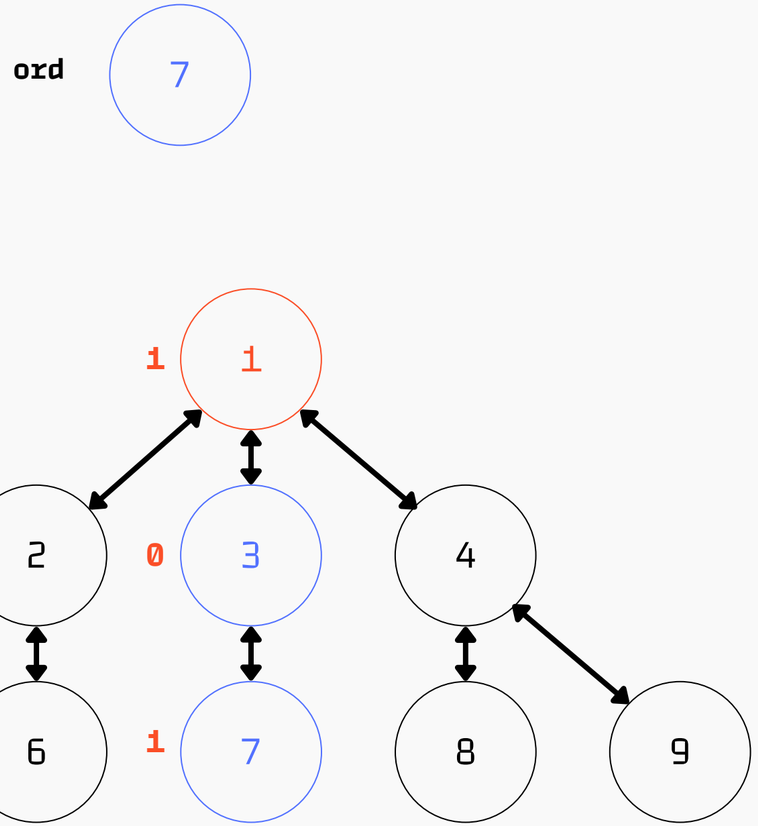
adj

```
0: {}
1: {2, 3, 4}
2: {1, 5, 6}
3: {1, 7}
4: {1, 8, 9}
5: {2}
6: {2}
7: {3}
8: {4}
9: {4}
```



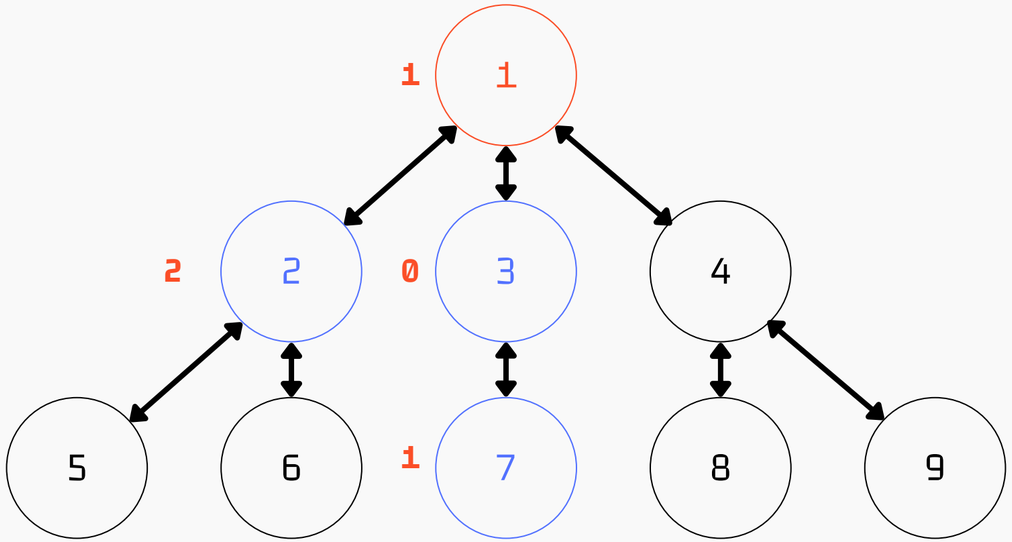
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



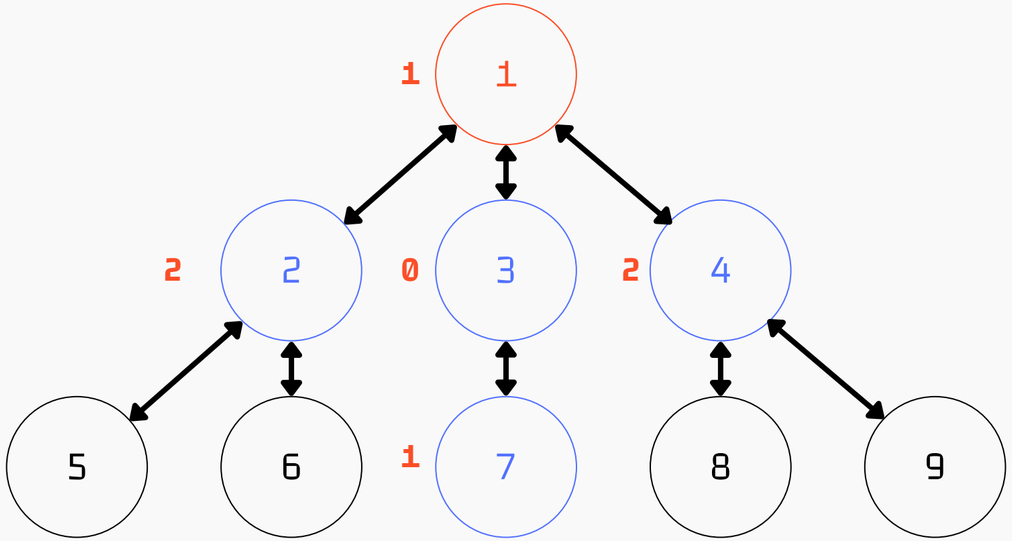
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



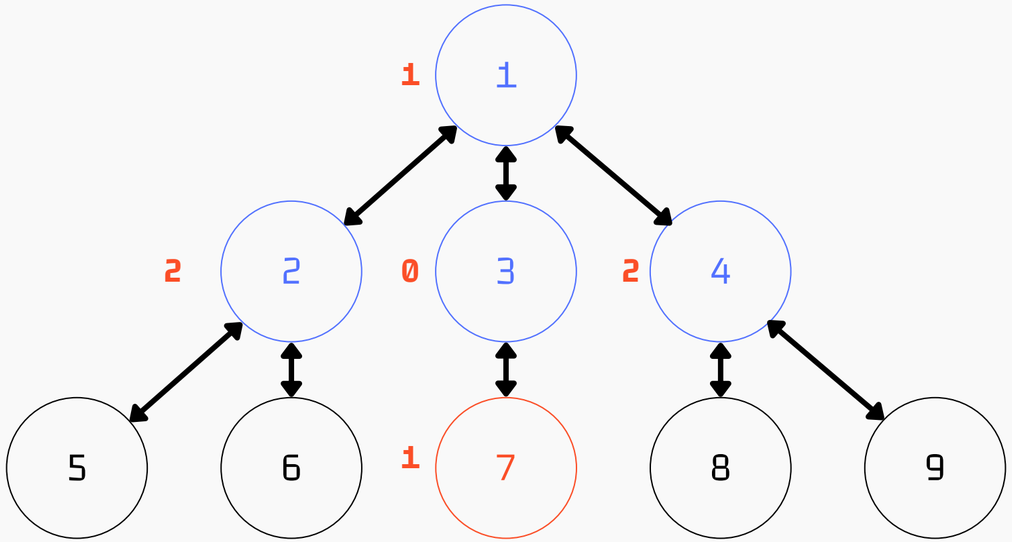
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



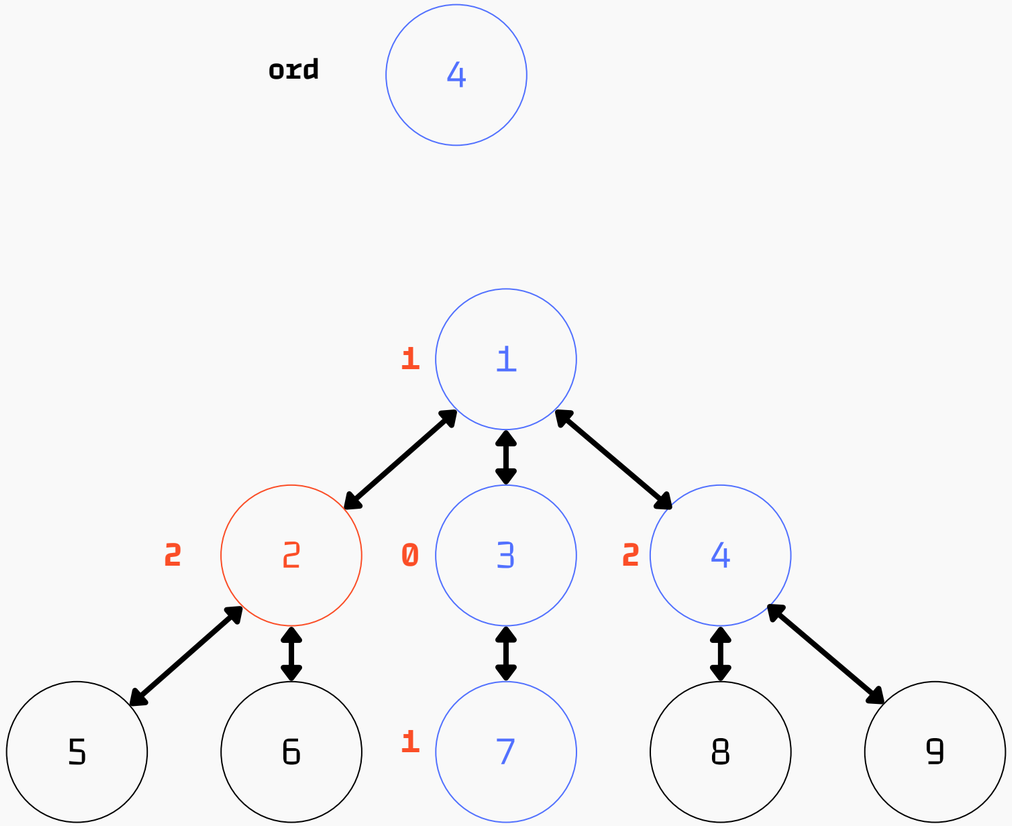
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



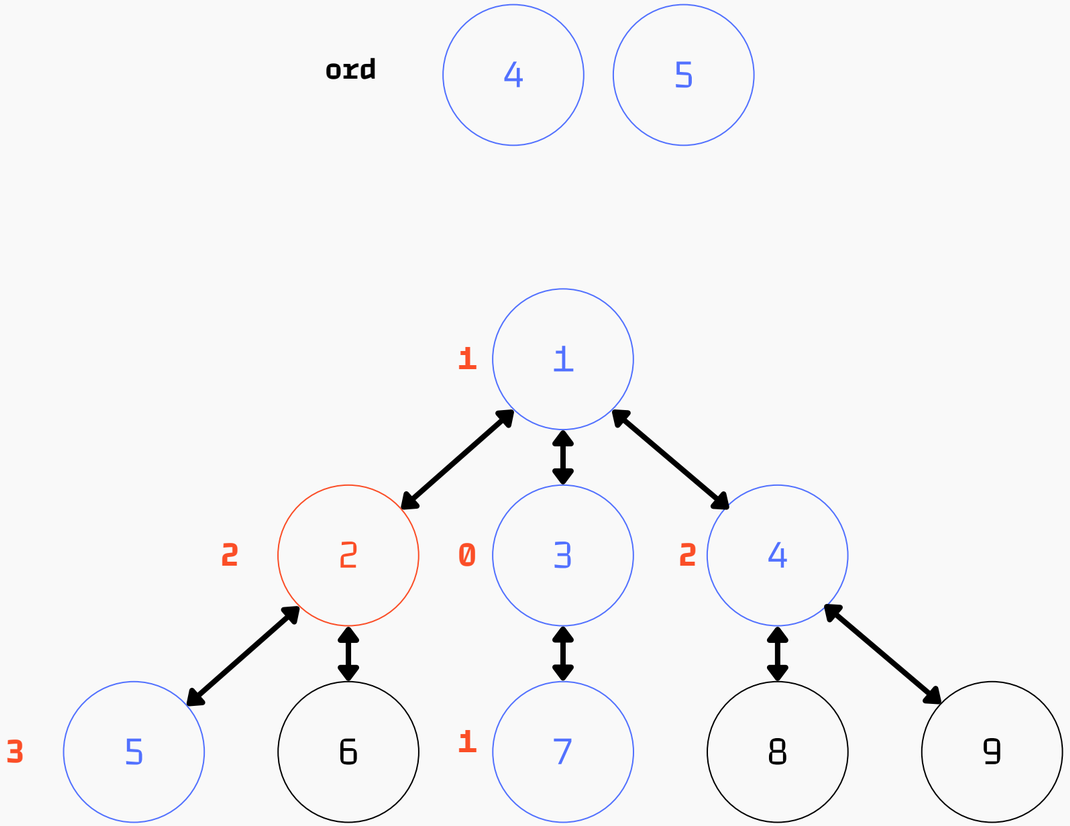
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



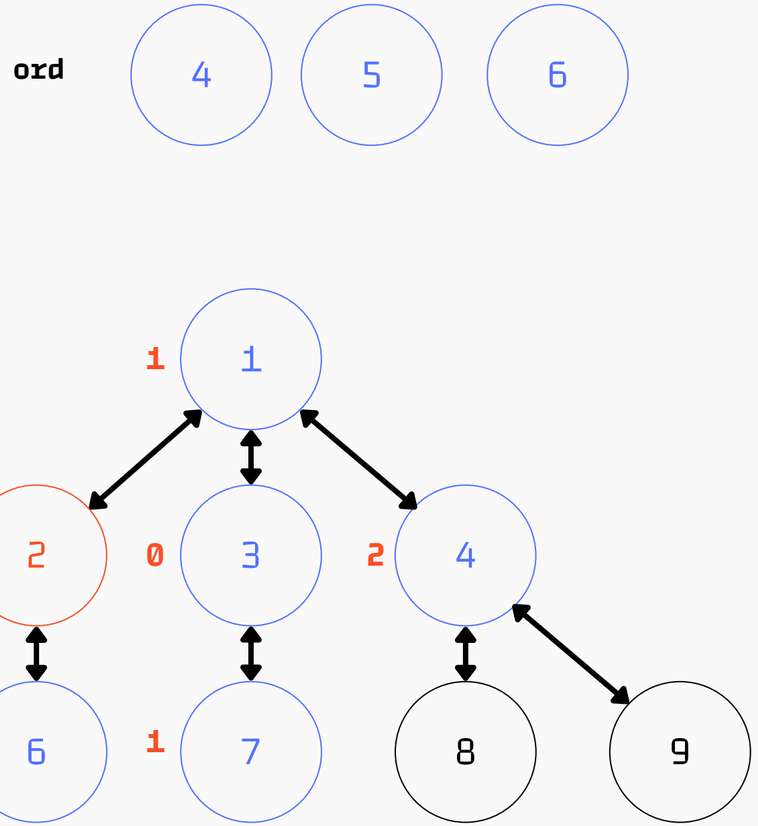
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



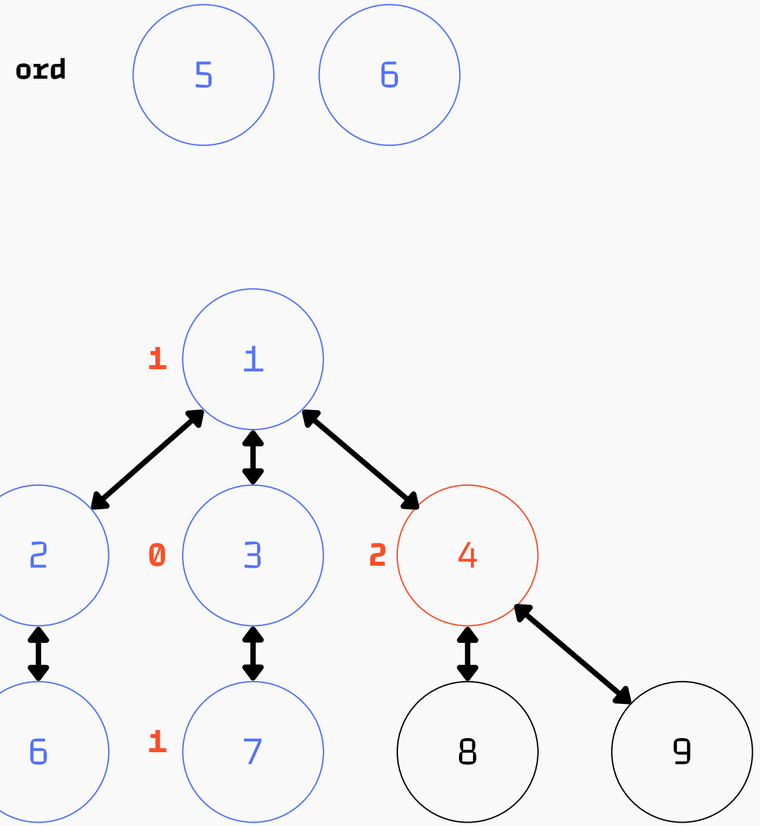
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



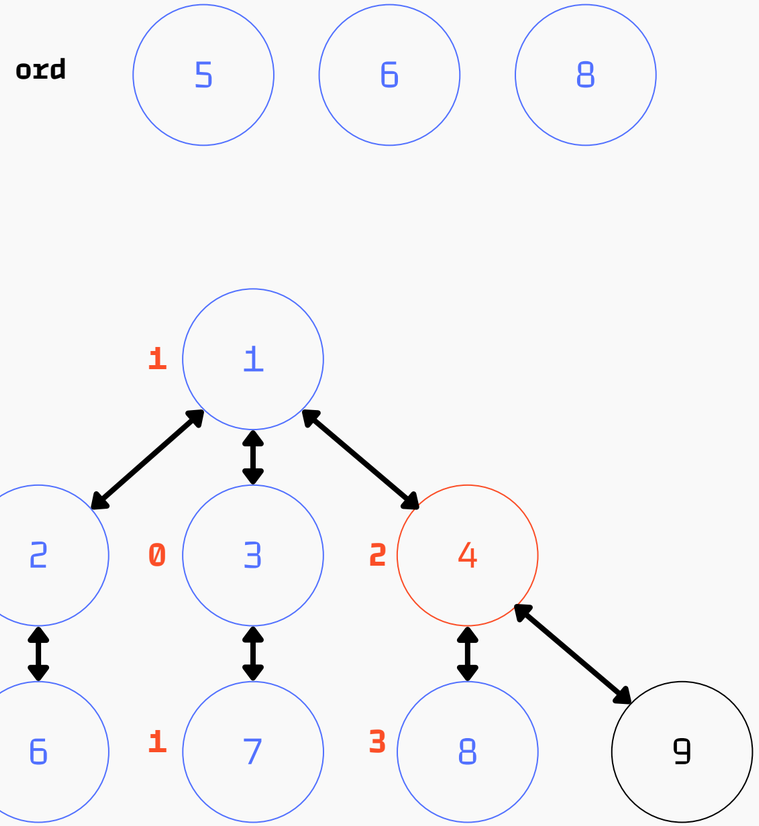
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



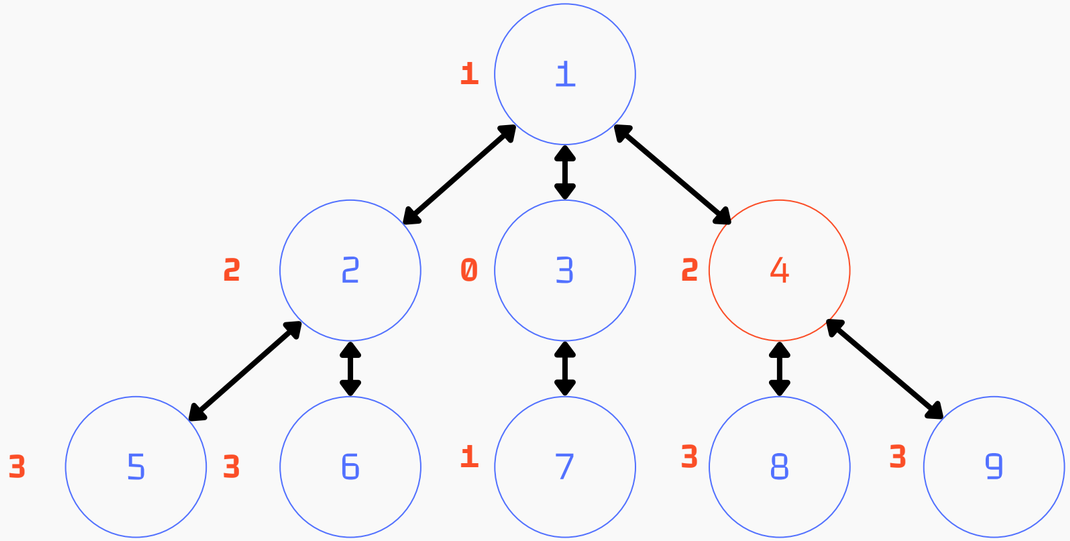
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



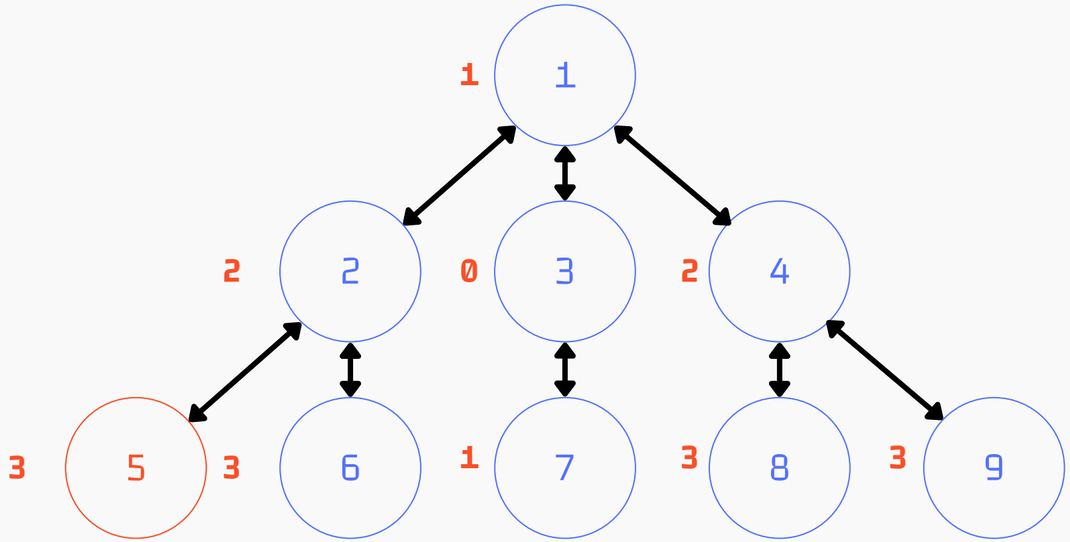
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



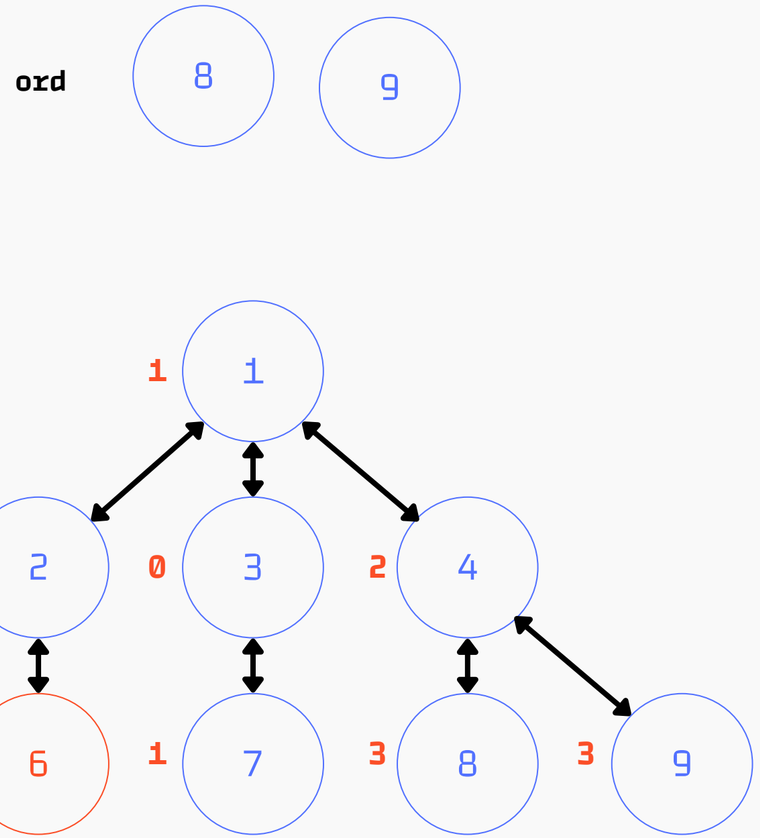
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



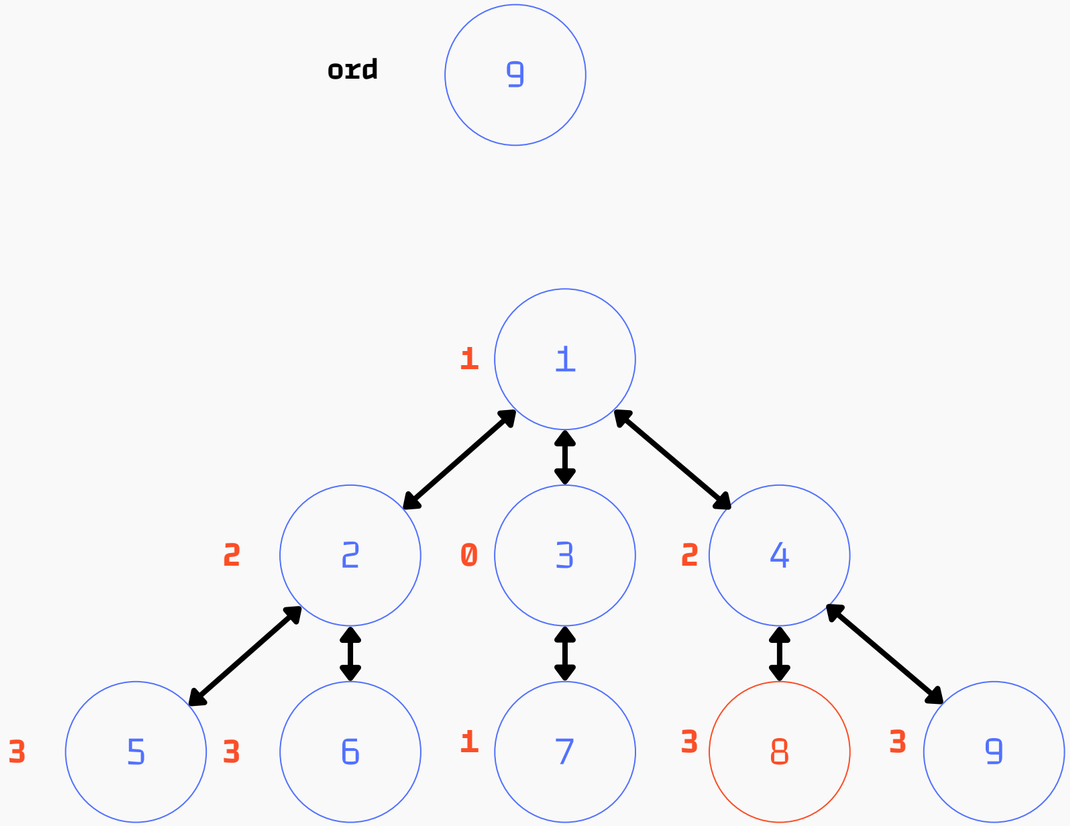
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



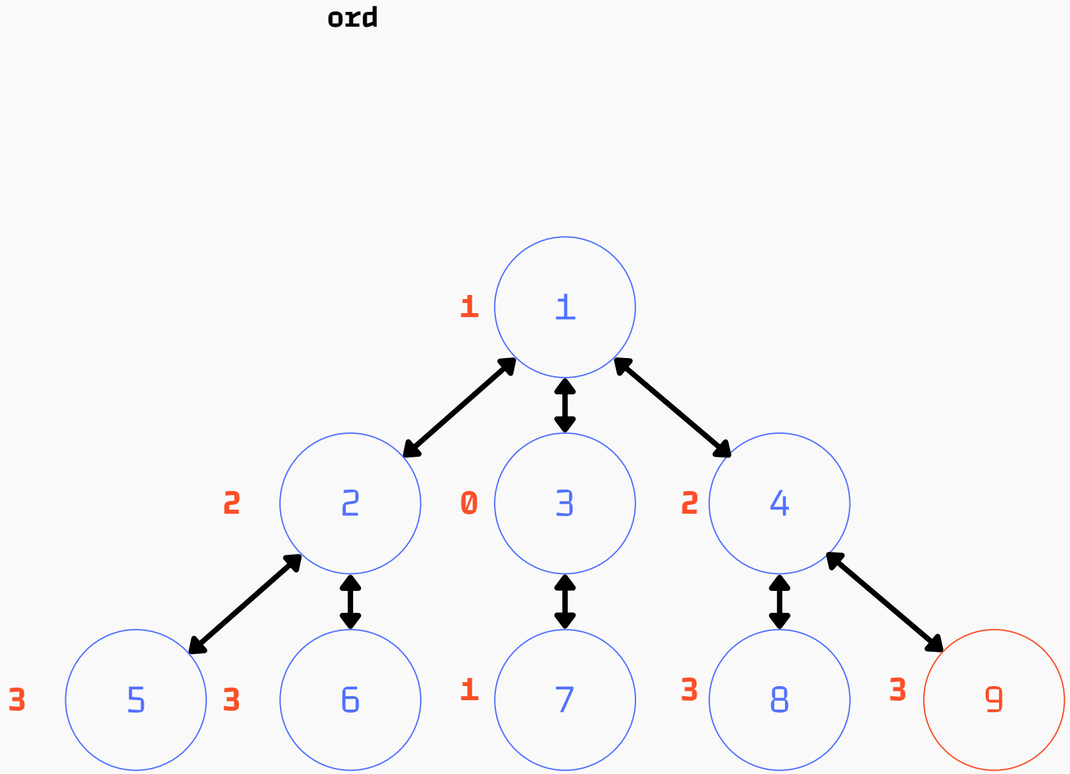
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



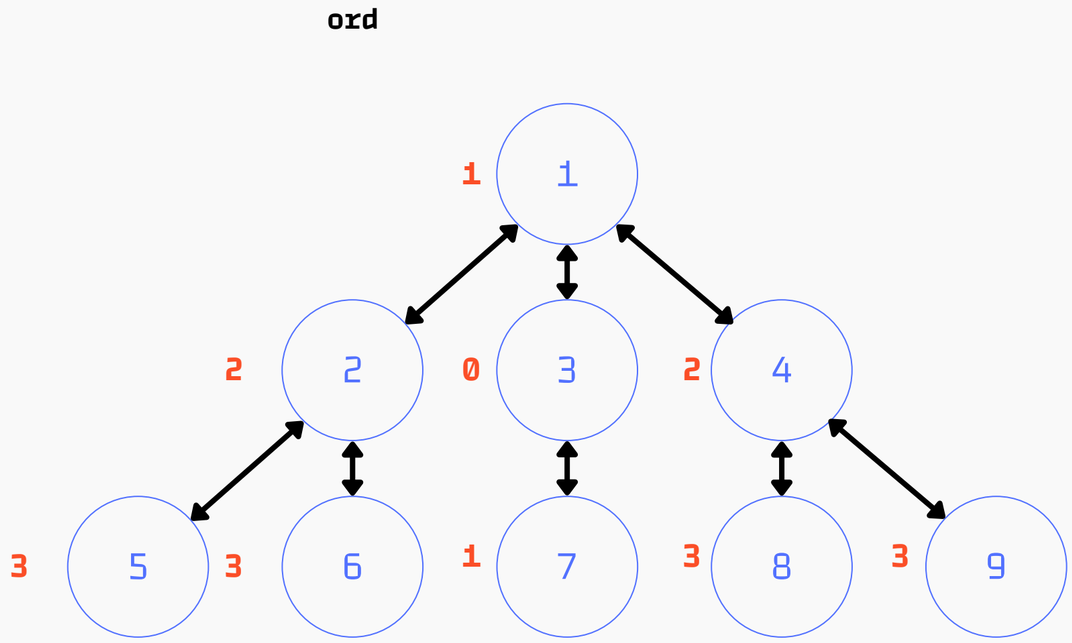
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

BFS: Busqueda en anchura



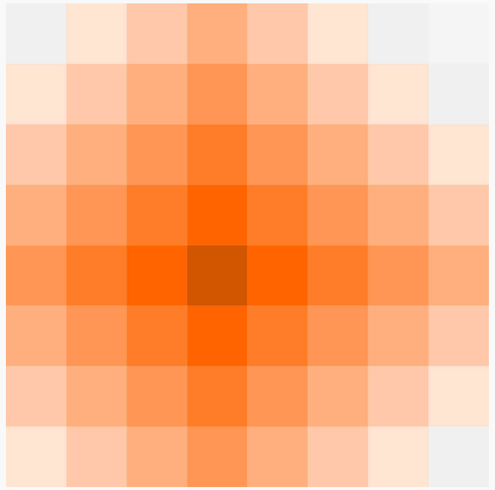
CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

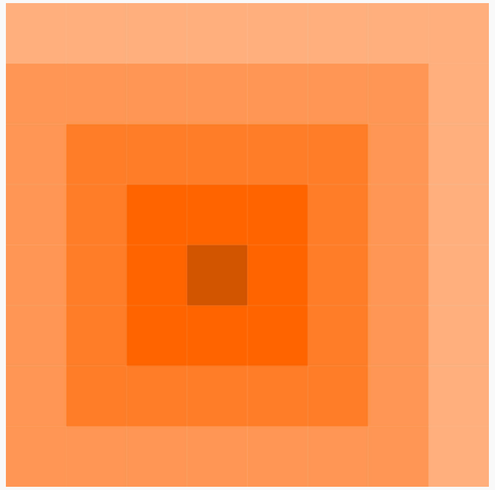
BFS: Busqueda en anchura



Eventualmente la BFS visitará todos los nodos y pasará por cada vértice, por lo que su complejidad es **O(n + m)**

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY





Matrices o tableros como grafos

Existen problemas donde se nos proporciona una matríz o tablero como entrada y debemos resolver un problema relacionado con algúna secuencia de movimientos sobre la cuadricula.

Es posible representar una matríz de estas características como un grafo, donde cada casilla es un nodo y es adyacente a las 4 u 8 casillas circundantes, segun si el problema cuenta o no las diagonales como aledañas.

En este tipo de problemas no necesitamos una lista de adyacencias, dado que para cada nodo o casilla **(x, y)** podemos computar sus adyacencias con **(x±1, y±1)**

En las imagenes podemos observar la misma matríz y casilla de origen. En la imagen superior las casillas diagonales no se consideran como adyacentes, en la inferior si.


CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY