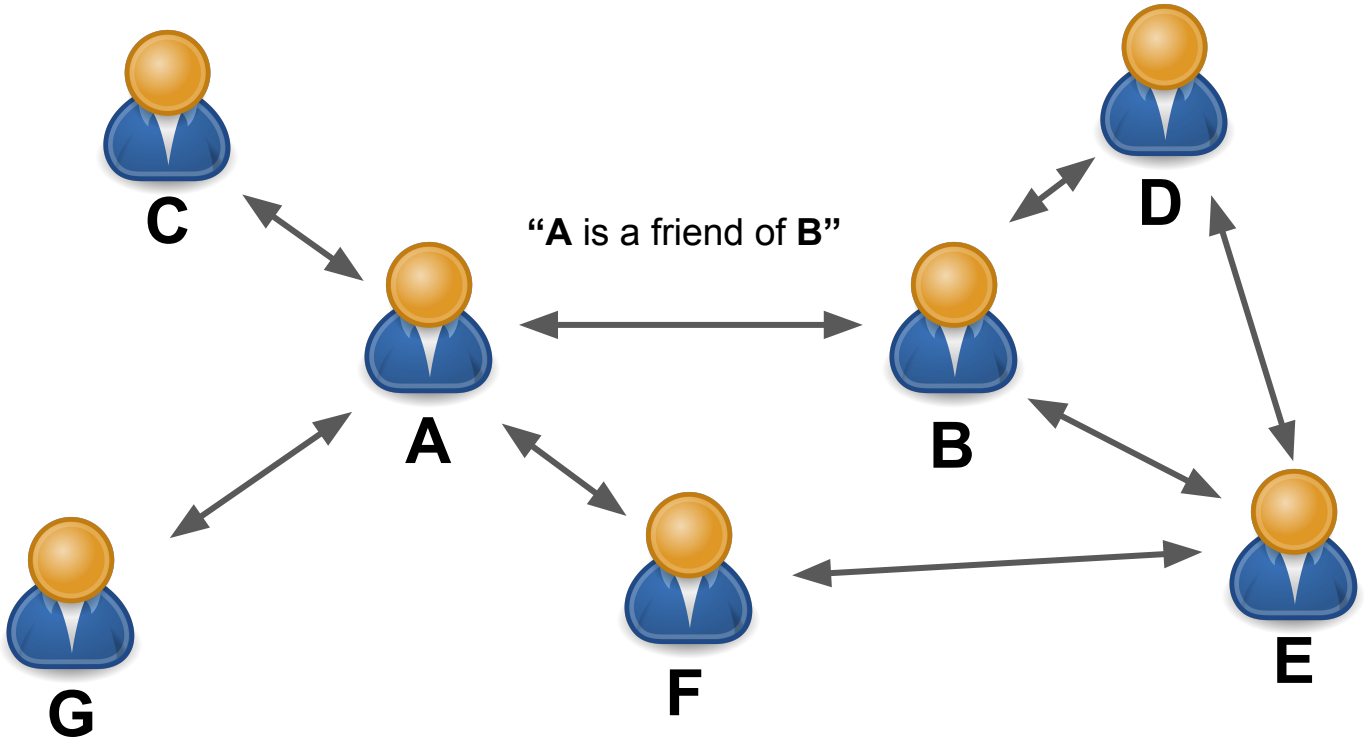
Graph Theory

Swiss Olympiad in Informatics 2019

What is a Graph?



Abstract Modelling of a Graph

Definition:

A graph G consists of:

a set of vertices (v_1, v_2, ... v_n)

a set of edges

a weight function (optional)

Properties of Graphs

Graphs can be

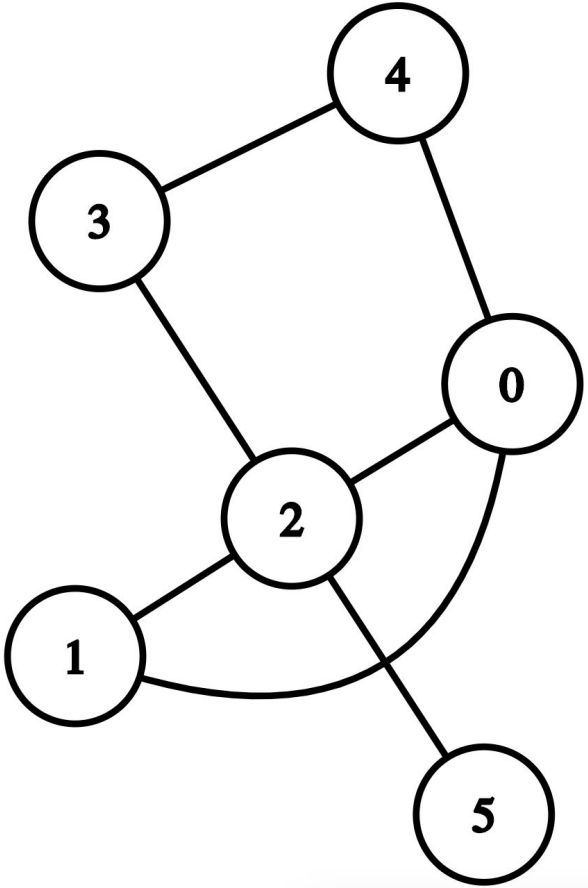
weighted, unweighted

directed, undirected

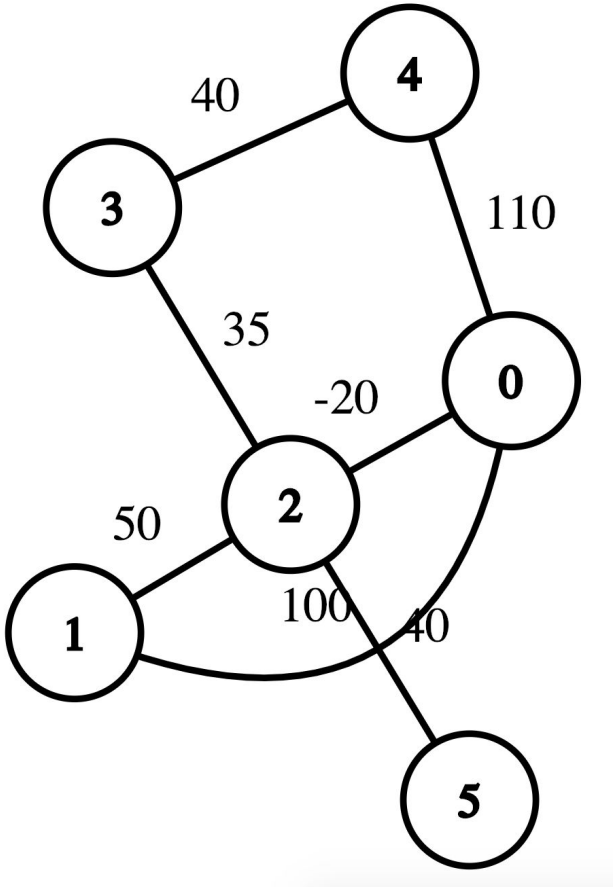
connected, disconnected

...

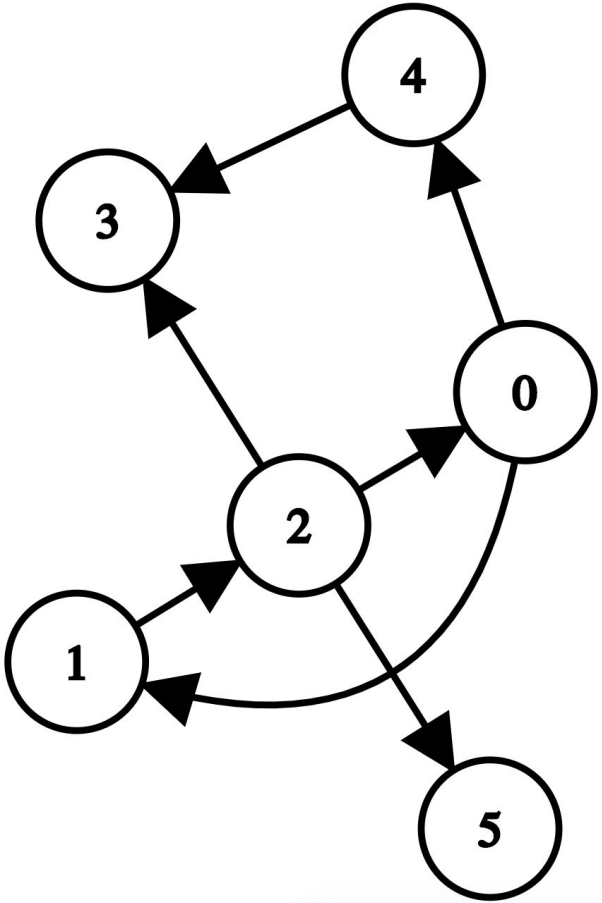
Simple



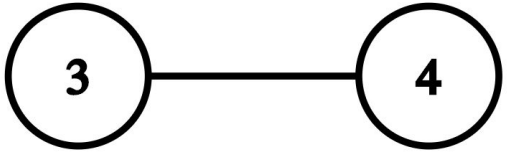
Weighted

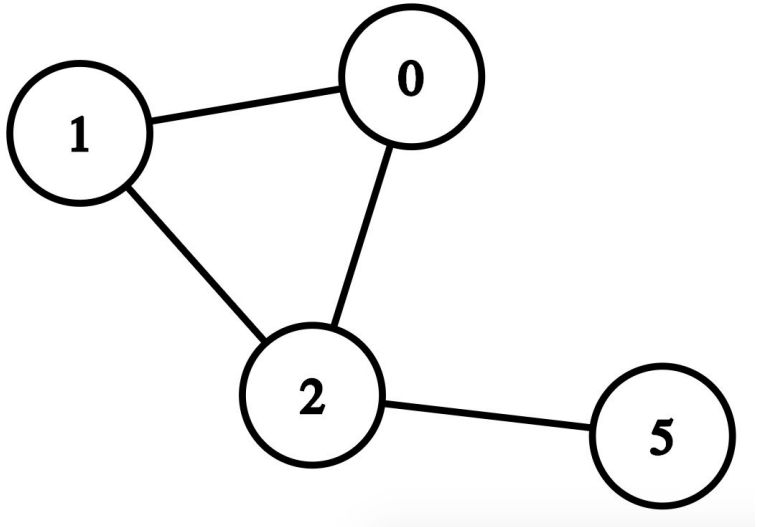


Directed



Disconnected



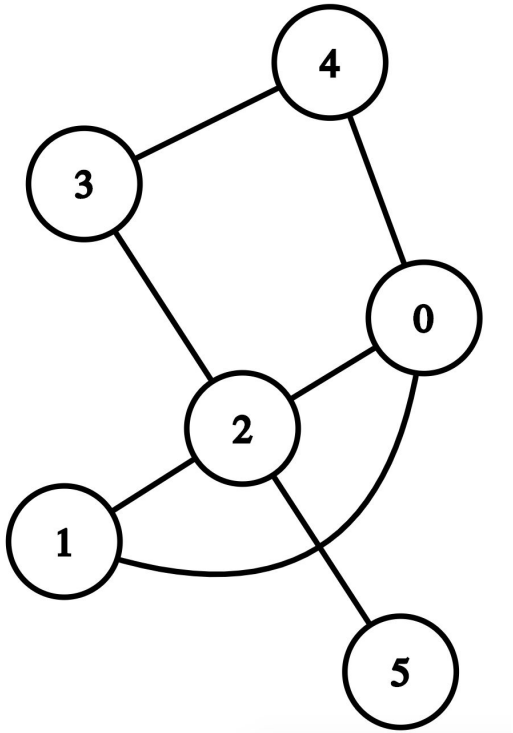


Graphs on Computers

How can we represent graphs in a program?

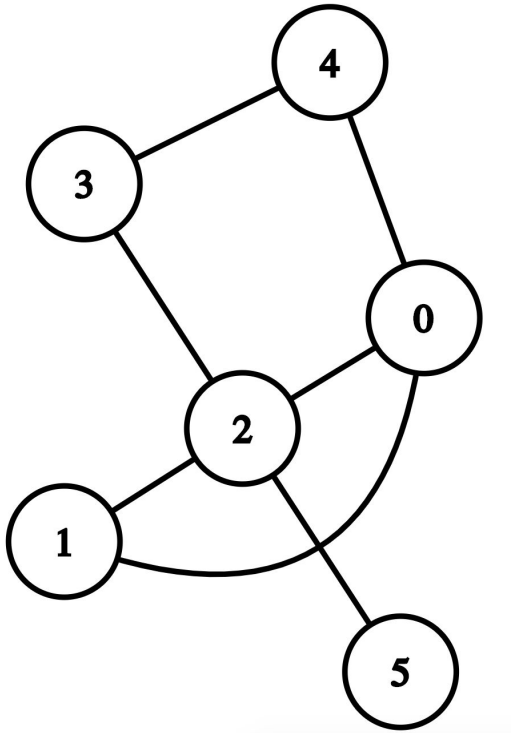
Adjacency Matrix

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

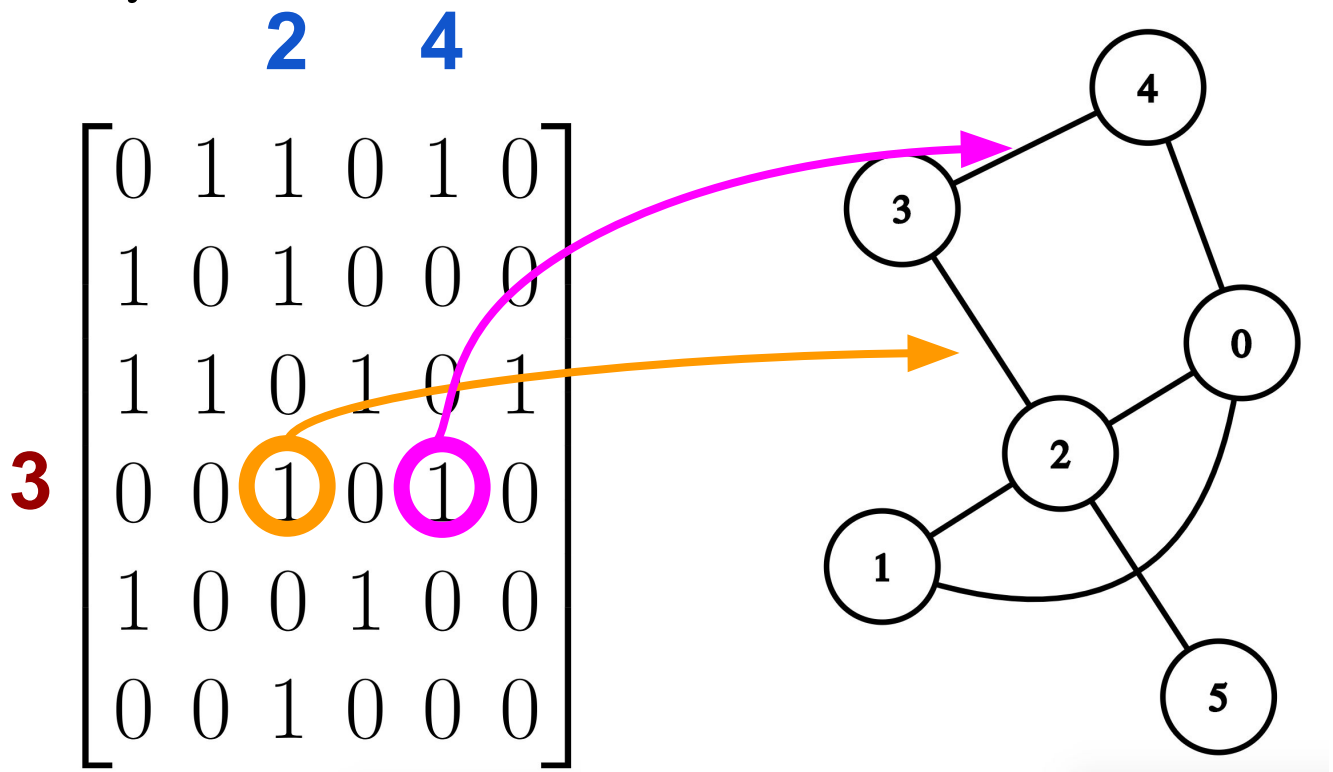


Adjacency Matrix

$$
\begin{array}{c}
\\
\mathbf{3}
\end{array}
\begin{bmatrix}
0 & 1 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
$$

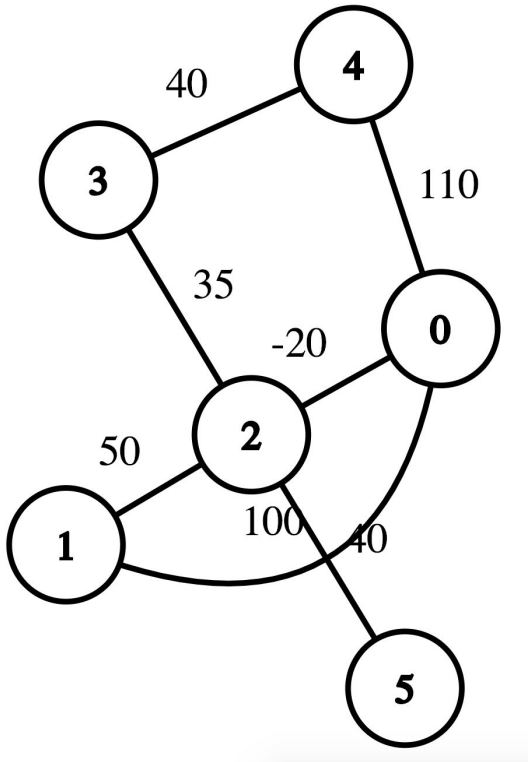


Adjacency Matrix



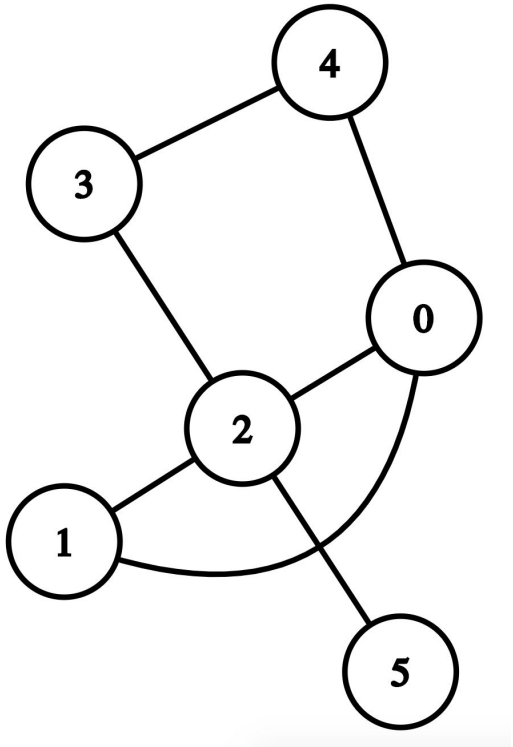
Adjacency Matrix in a Weighted Graph

$$\begin{bmatrix} 0 & 40 & -20 & 0 & 110 & 0 \\ 40 & 0 & 50 & 0 & 0 & 0 \\ -20 & 50 & 0 & 35 & 0 & 100 \\ 0 & 0 & 35 & 0 & 40 & 0 \\ 110 & 0 & 0 & 40 & 0 & 0 \\ 0 & 0 & 100 & 0 & 0 & 0 \end{bmatrix}$$



Adjacency List

```
0: 1, 2, 4
1: 0, 2
2: 0, 1, 3, 5
3: 2, 4
4: 0, 3
5: 2
```



Adjacency List Implementation

```
vector<vector<int>> graph;
```

Depth First Search

Stack

- Last-In First-Out
- `stack<int>` is a stack of integers
- `push(x)` pushes x to the top
- `top()` returns element at the top
- `pop()` pops element at the top

Stack

```
stack<int> numbers;
numbers.push(1);
numbers.push(2);
numbers.push(3);
cout << numbers.top() << " ";
numbers.pop();
numbers.pop();
cout << numbers.top() << " ";
```

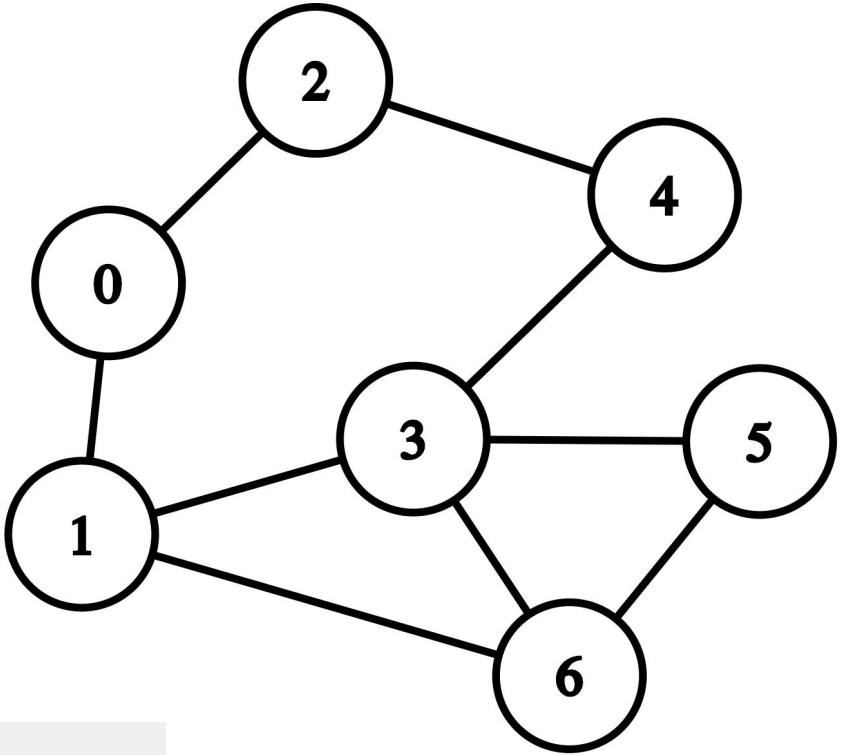
3 1

Depth First Search (DFS)

- pop unvisited vertex **v** from stack
- mark **v** as visited
- push all adjacent vertices to the stack
- repeat

DFS

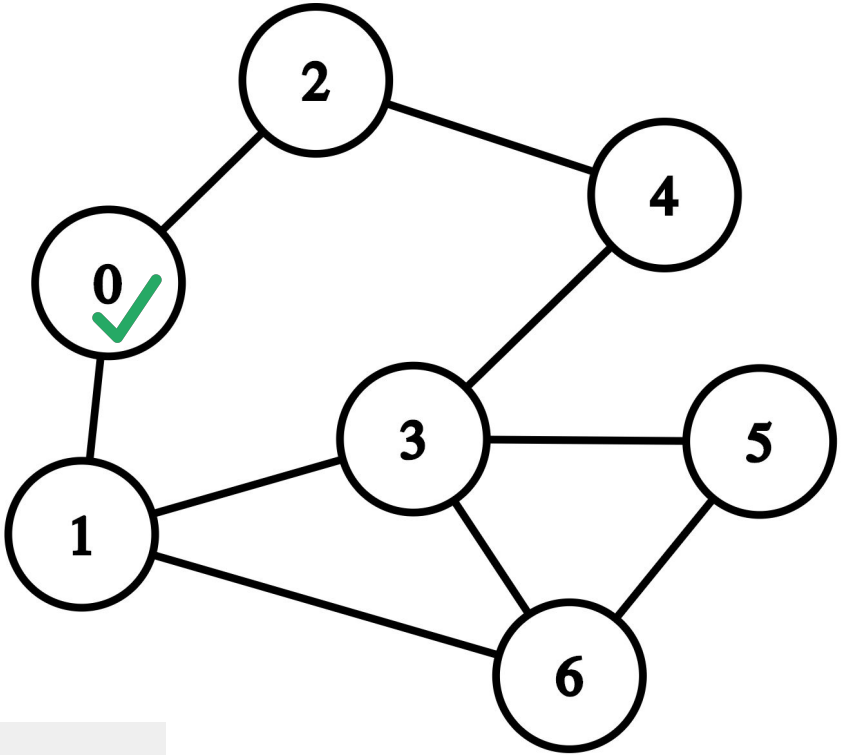
>




```
0
```

DFS

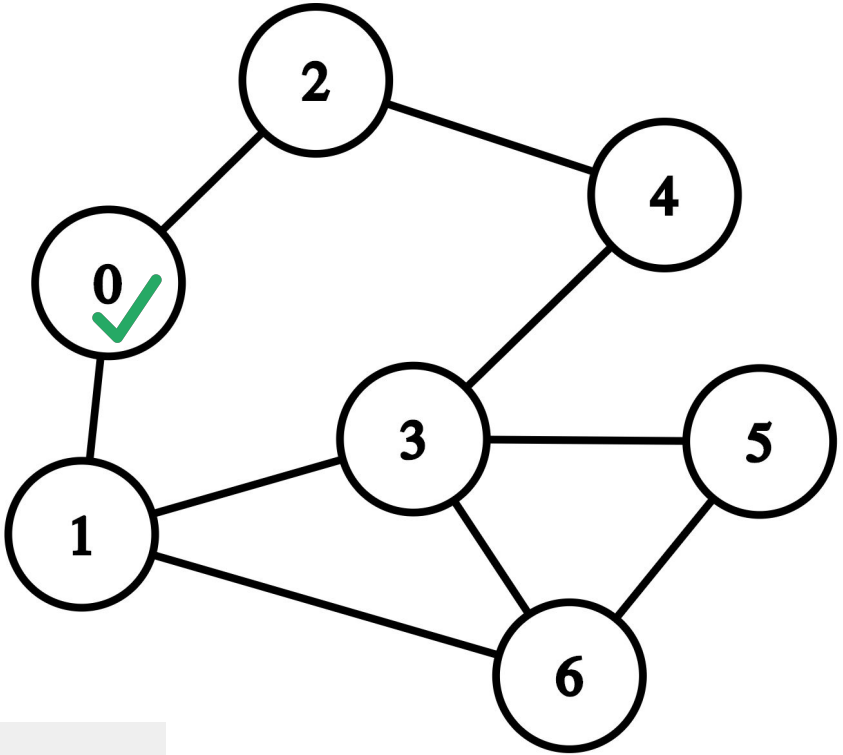
>0



`CurrentNode=`**0**

DFS

>0

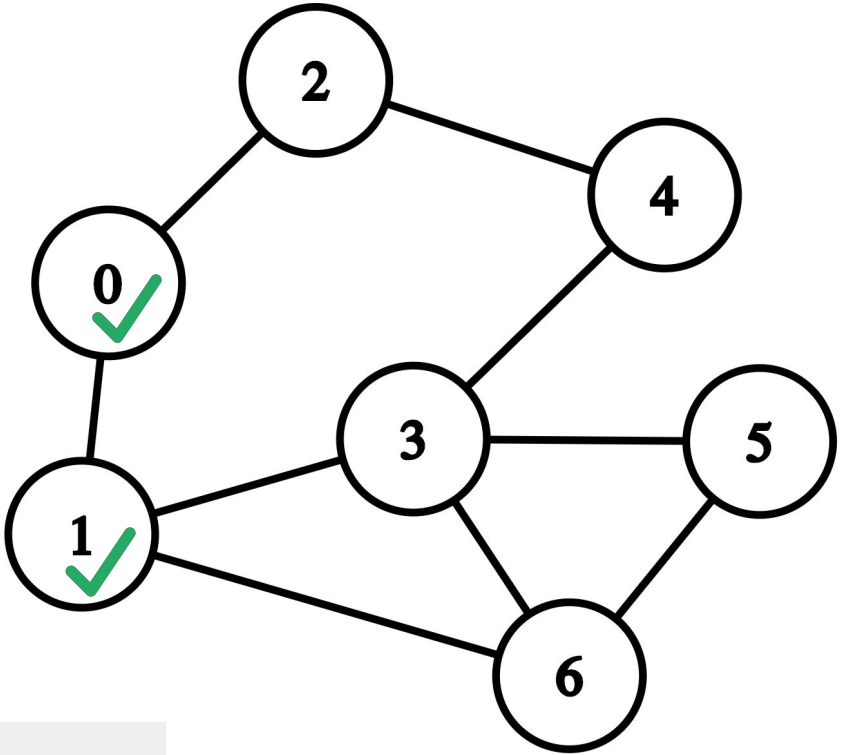


`CurrentNode=`**0**

```
2,1
```

DFS

```
>0  1
```

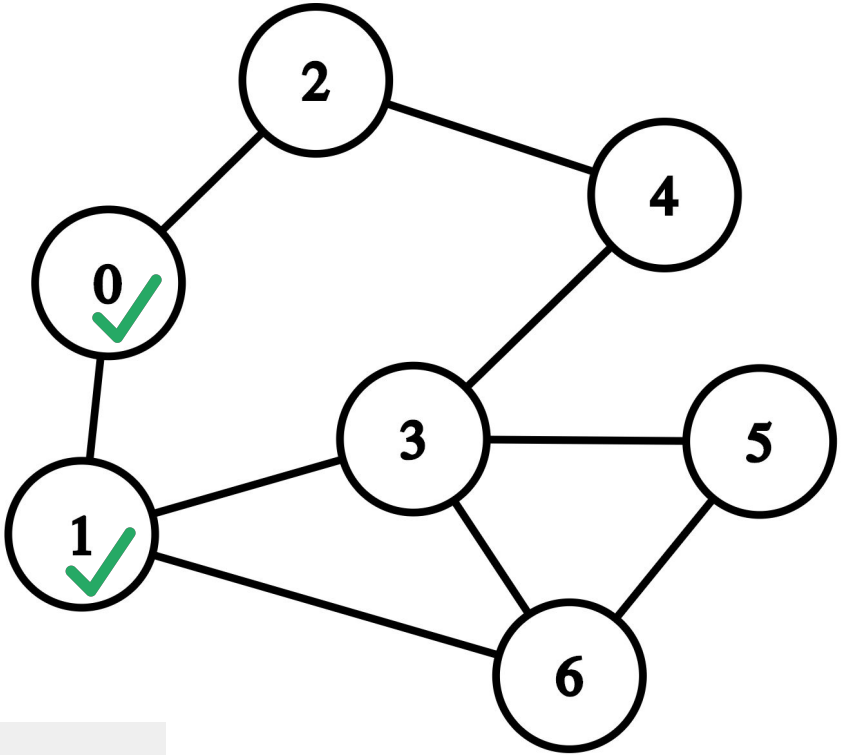



```
CurrentNode=1
```

2

DFS

```
>0 1
```

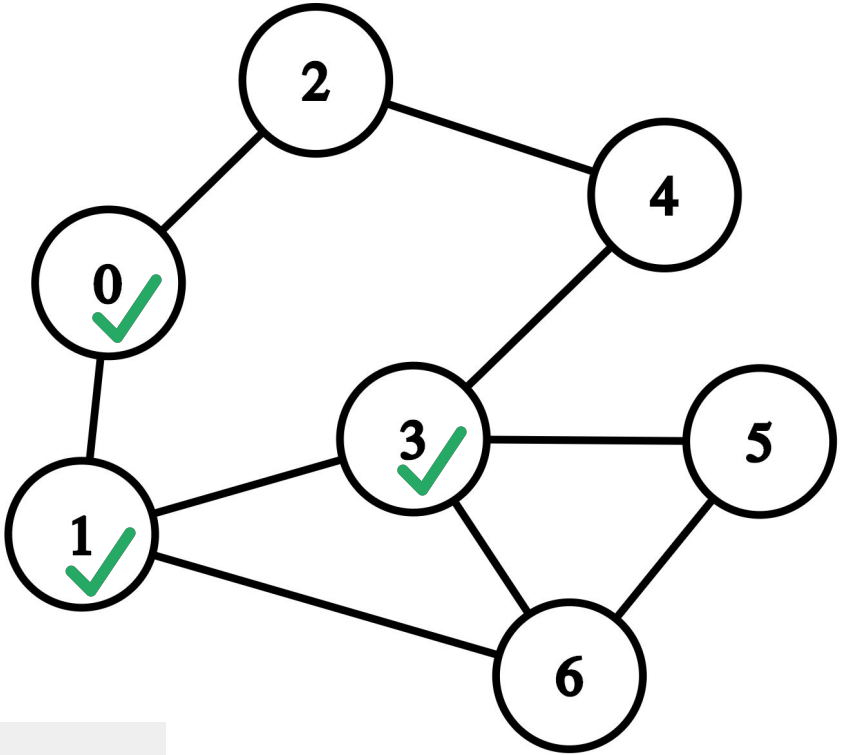



```
CurrentNode=1
```

```
2,6,3
```

DFS

```
>0 1 3
```

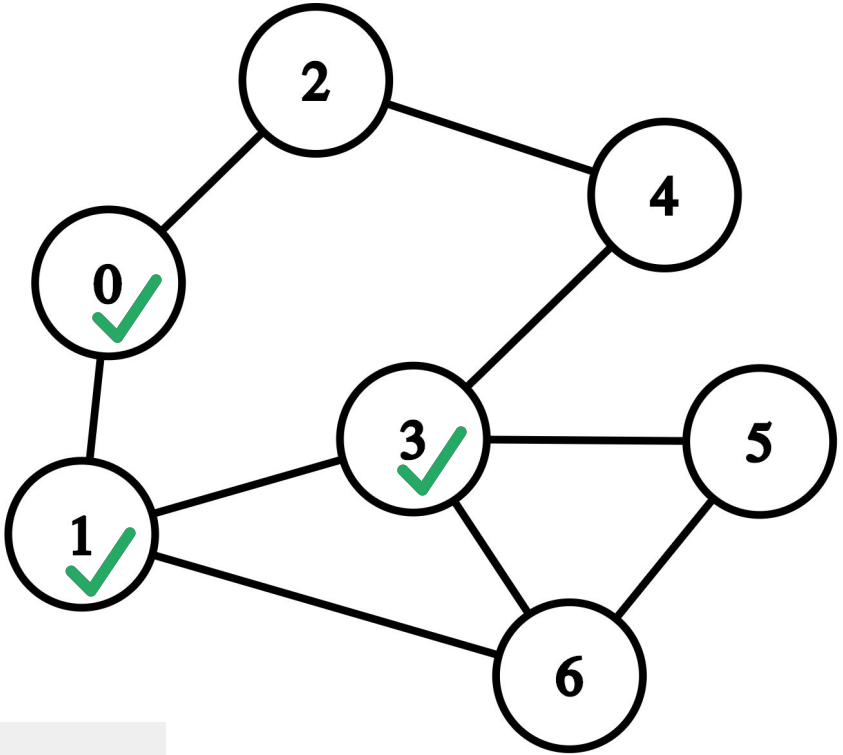



```
CurrentNode=3
```

2,6

DFS

```
>0  1  3
```

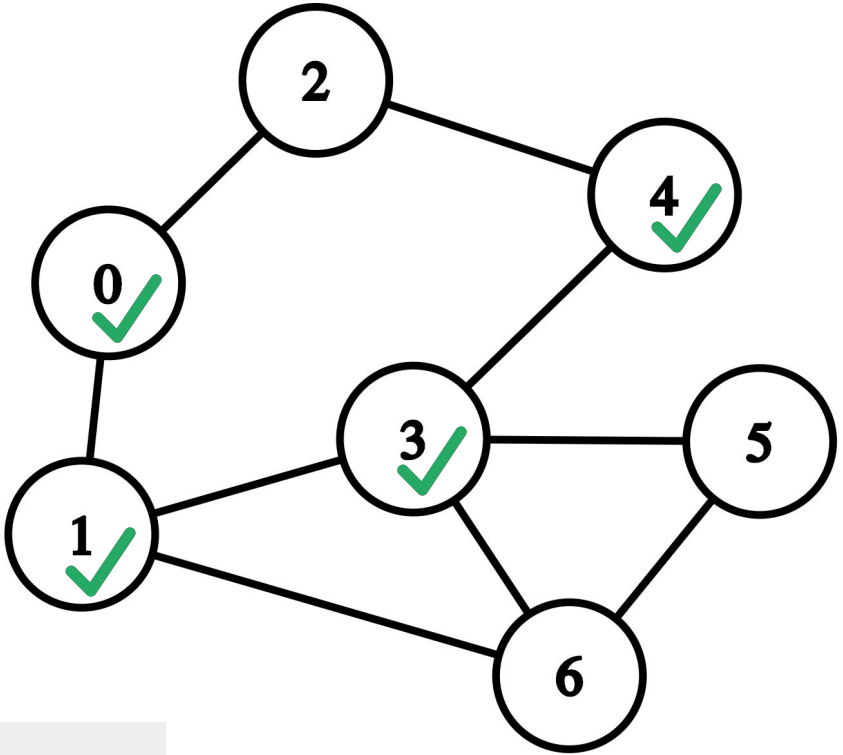



```
CurrentNode=3
```

2,6,6,5,4

DFS

```
>0 1 3 4
```

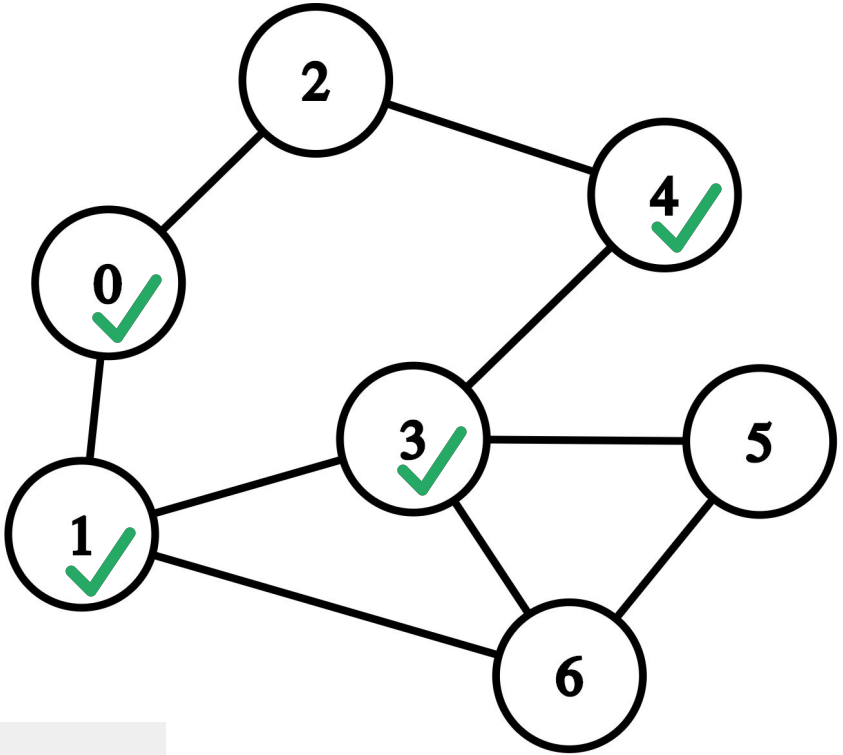


`CurrentNode=`**`4`**

`2,6,6,5`

DFS

```
>0 1 3 4
```

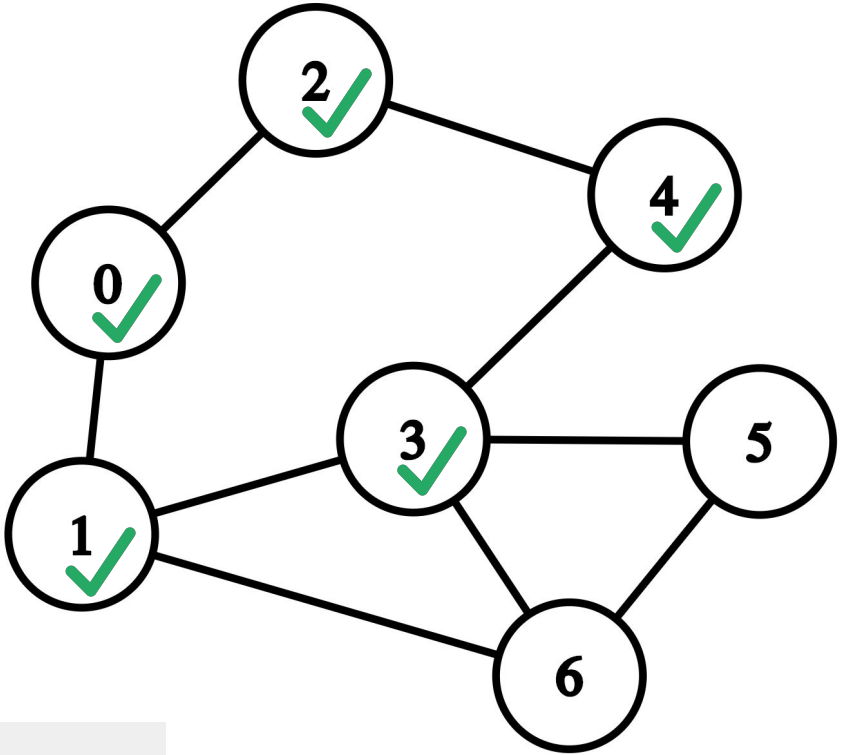


`CurrentNode=`**`4`**

```
2,6,6,5,2
```

DFS

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

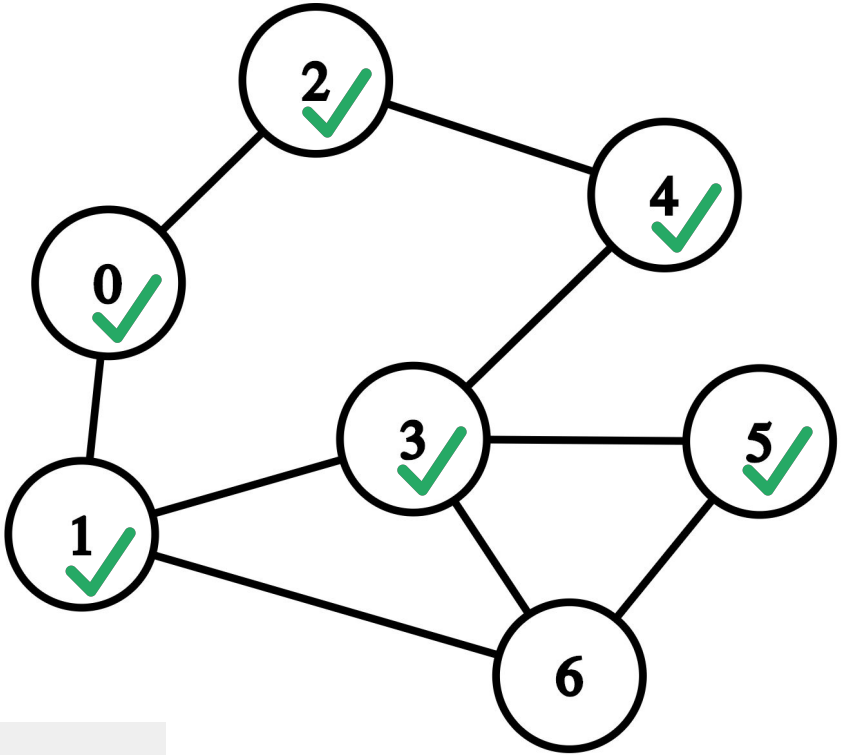



```
CurrentNode=2
```

```
2,6,6,5
```

DFS

```
>0 1 3 4 2 5
```

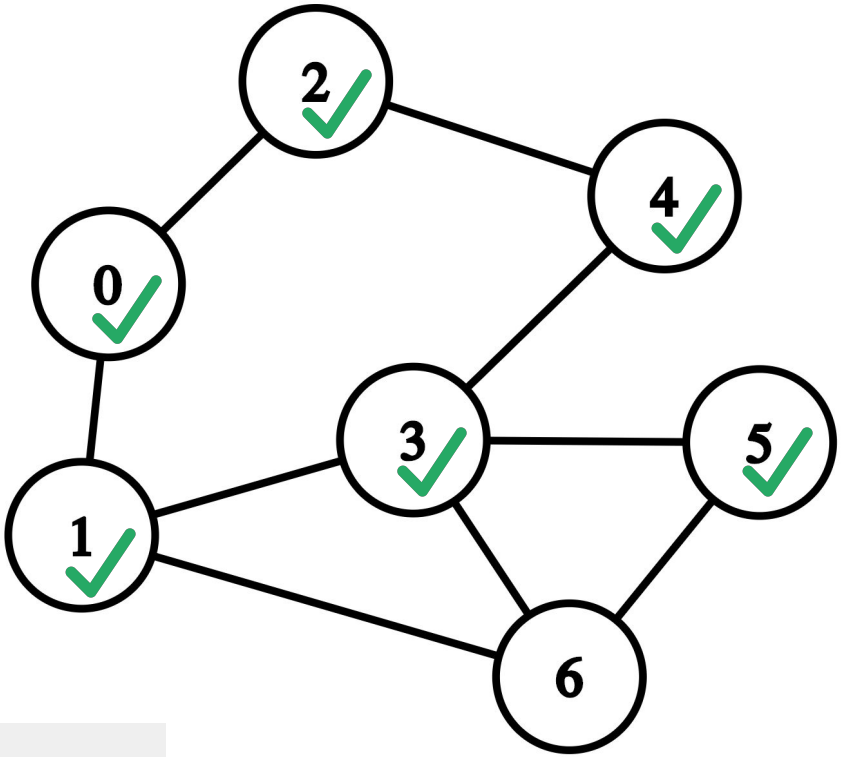



```
CurrentNode=5
```

2,6,6

DFS

```
>0 1 3 4 2 5
```

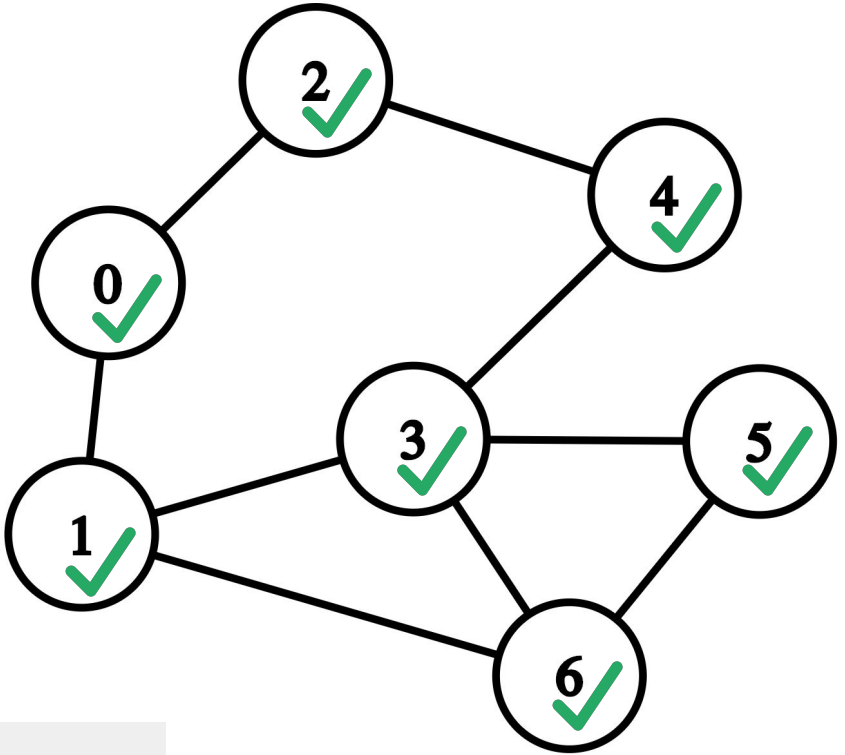



```
CurrentNode=5
```

2,6,6,6

DFS

```
>0 1 3 4 2 5 6
```

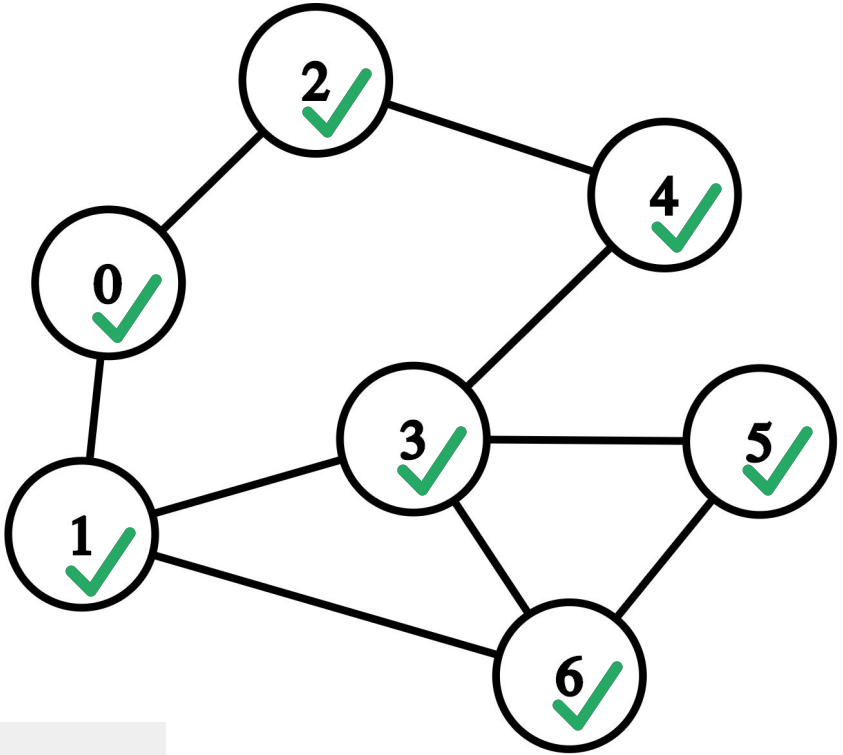



```
CurrentNode=6
```

```
2,6,6
```

DFS

```
>0 1 3 4 2 5 6
```

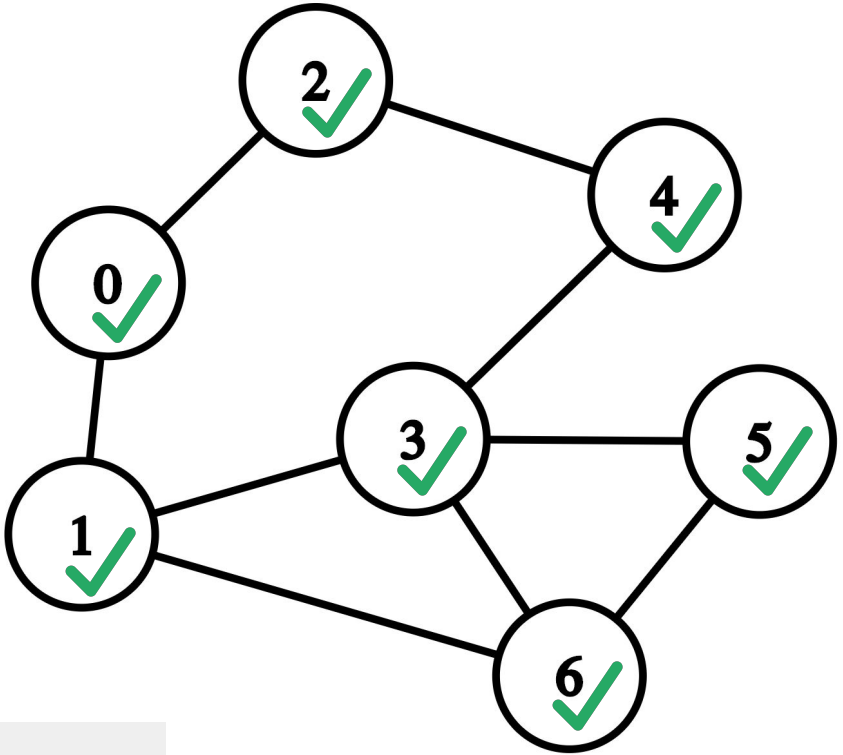



```
CurrentNode=6
```

```
2,6
```

DFS

```
>0 1 3 4 2 5 6
```

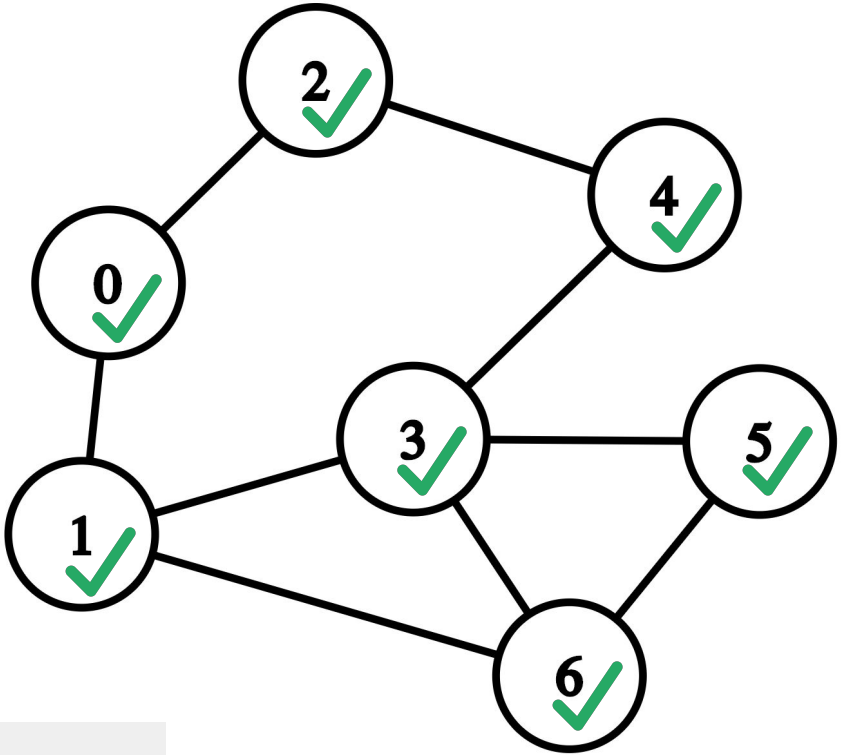



```
CurrentNode=6
```

2

DFS

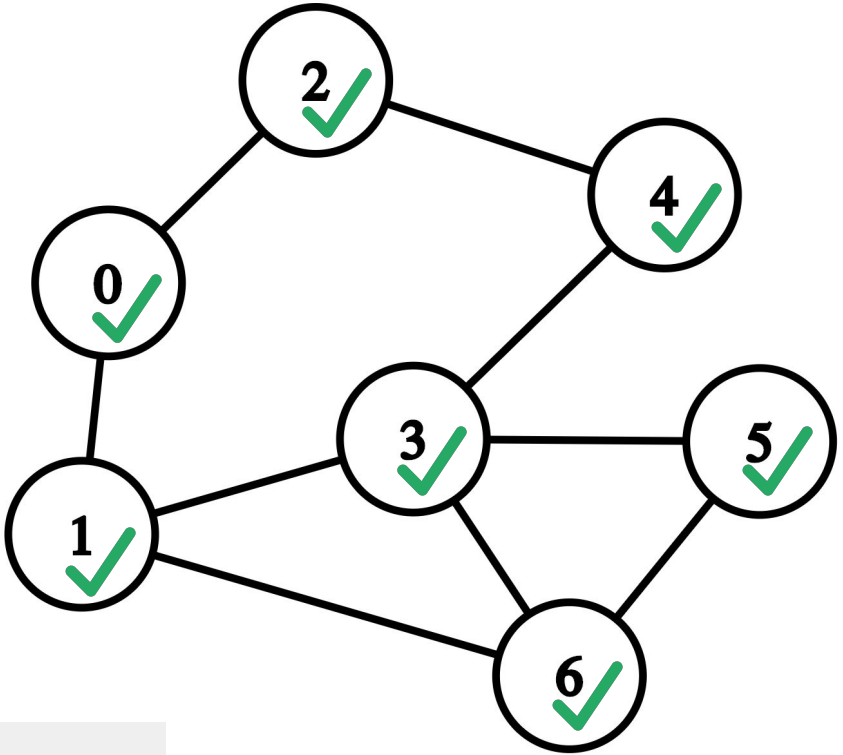
```
>0 1 3 4 2 5 6
```




```
CurrentNode=2
```

DFS

```
>0 1 3 4 2 5 6
```




```
Finished
```

empty

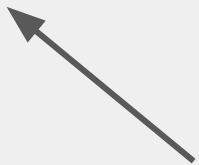
```
void dfs() {
    stack<int> vertices;
    vertices.push(0);
    while (!vertices.empty()) {
        int current = vertices.top();
        vertices.pop();
        if (!visited[current]) {
            visited[current] = true;
            cout << current << " ";
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                vertices.push(next);
            }
        }
    }
}
```

```
void dfs() {
    stack<int> vertices;
    vertices.push(0);
    while (!vertices.empty()) {
        int current = vertices.top();
        vertices.pop();
        if (!visited[current]) {
            visited[current] = true;
            cout << current << " ";
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                vertices.push(next);
            }
        }
    }
}
```

Push root to stack

```
void dfs() {
    stack<int> vertices;
    vertices.push(0);
    while (!vertices.empty()) {
        int current = vertices.top();
        vertices.pop();
        if (!visited[current]) {
            visited[current] = true;
            cout << current << " ";
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                vertices.push(next);
            }
        }
    }
}
```

```
void dfs() {
    stack<int> vertices;
    vertices.push(0);
    while (!vertices.empty()) {
        int current = vertices.top();
        vertices.pop();
        if (!visited[current]) {
            visited[current] = true;
            cout << current << " ";
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                vertices.push(next);
            }
        }
    }
}
```



Push adjacent vertices

Recursive DFS

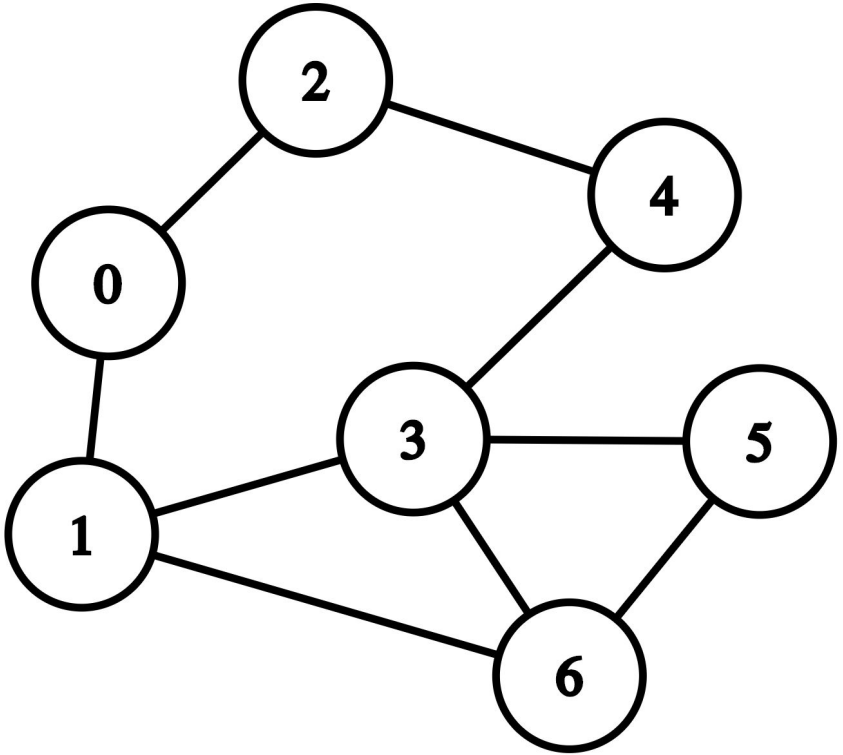
Same idea

Implemented recursively

```
void dfs(int current) {
    if (!visited[current]) {
        visited[current] = true;
        cout << current << " ";
        for (int i = 0; i < graph[current].size(); i++) {
            int next = graph[current][i];
            dfs(next);
        }
    }
}
```

Recursive DFS

>

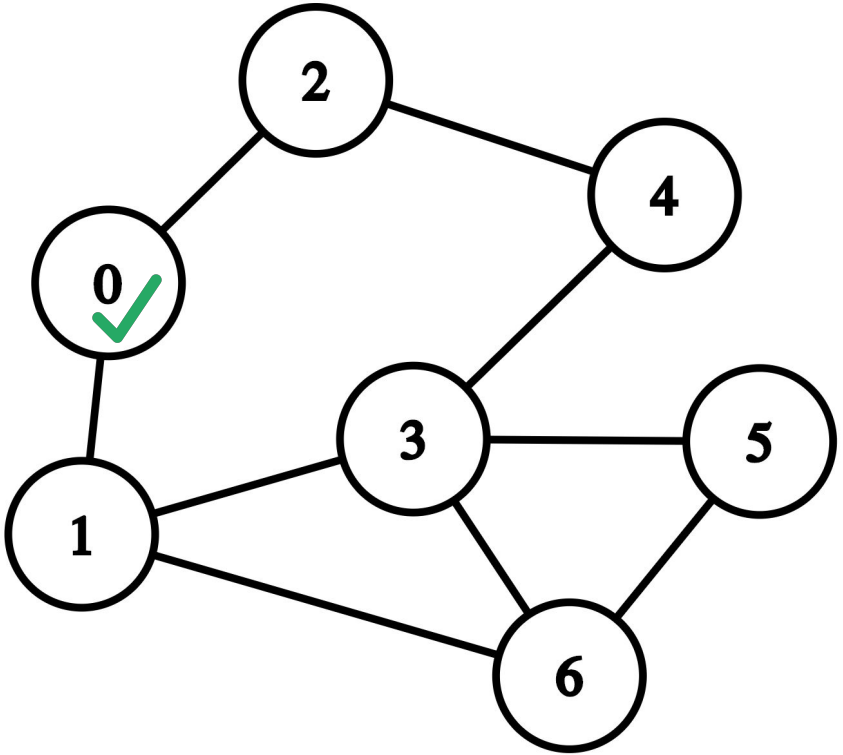



```
dfs(0)
```

DFS

>0

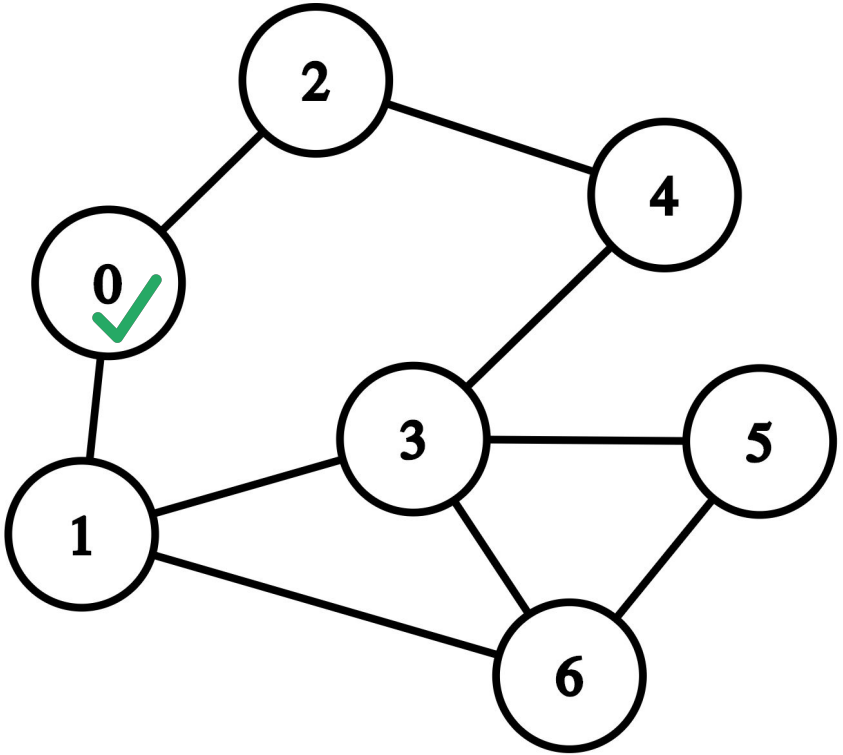
CurrentNode=**0**




```
dfs(0)
```

DFS

>0

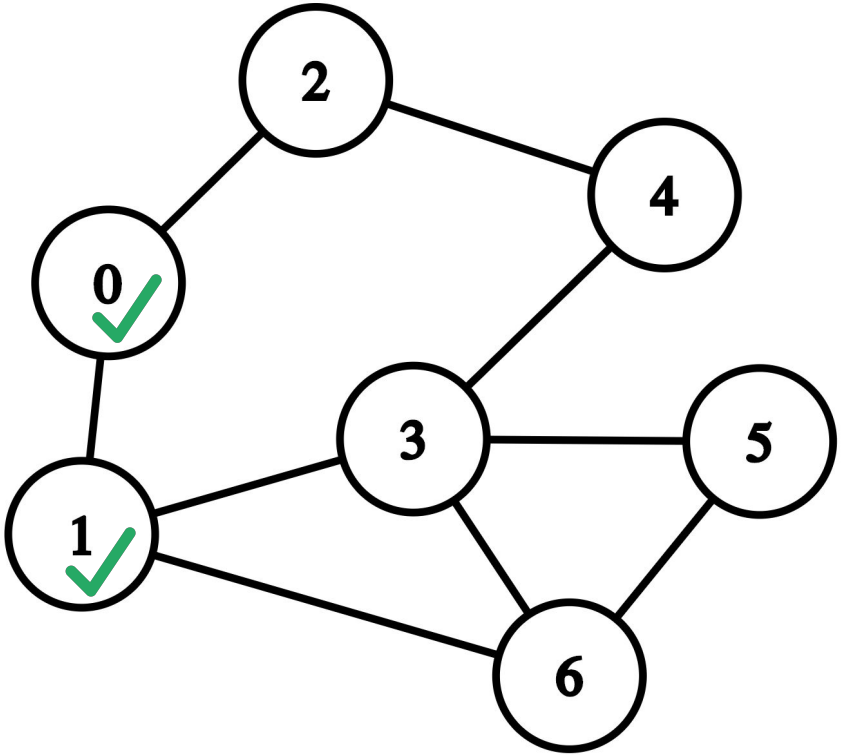


CurrentNode=**0**

dfs(0) → dfs(1)

DFS

```
>0  1
```

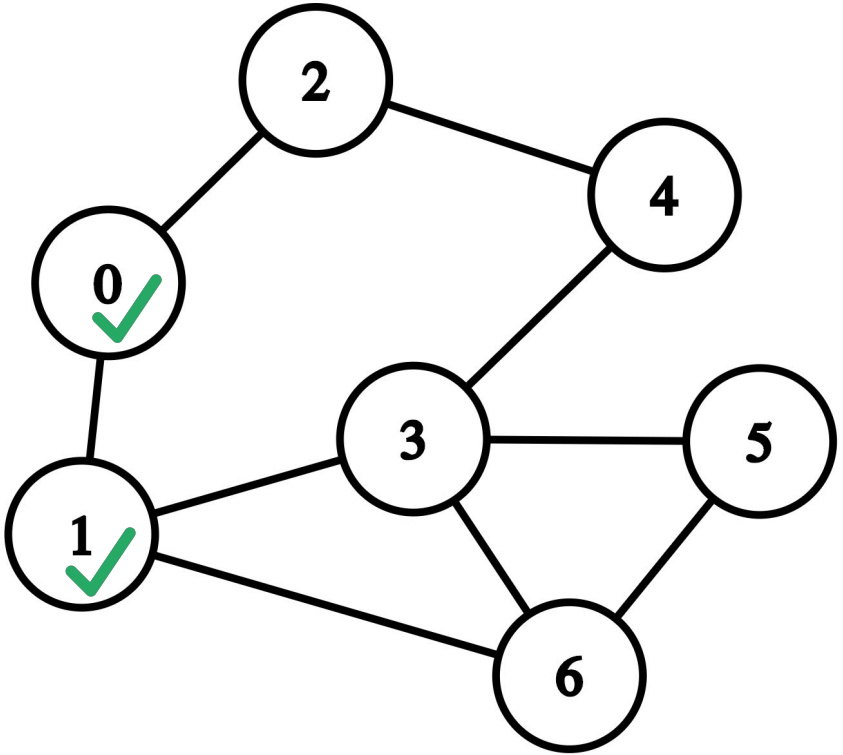



```
CurrentNode=1
```

dfs(0) → **dfs(1)**

DFS

```
>0 1
```

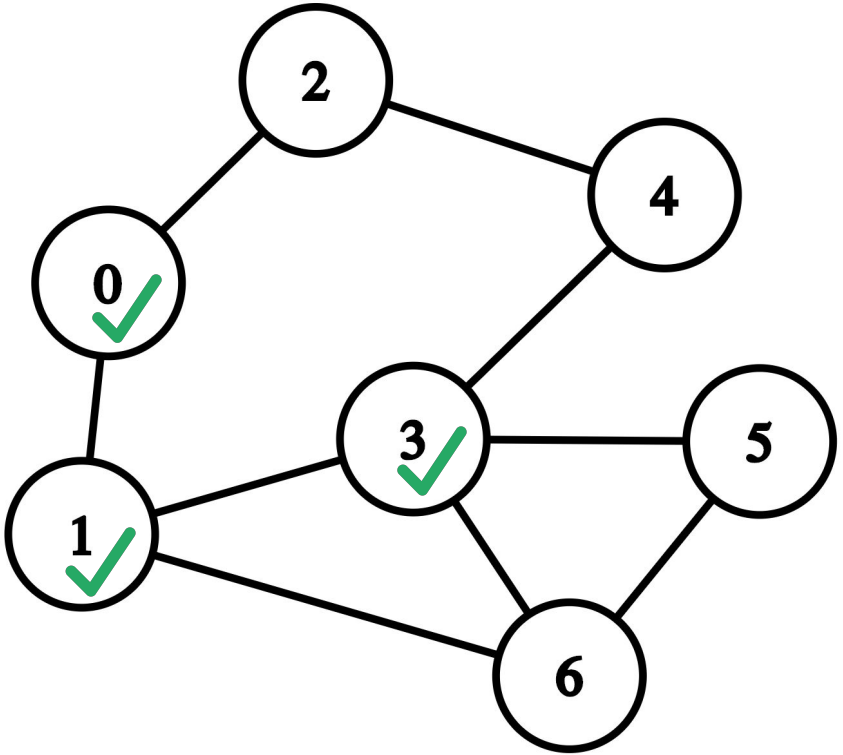



```
CurrentNode=1
```

dfs(0) → **dfs(1)** → dfs(3)

DFS

```
>0  1  3
```

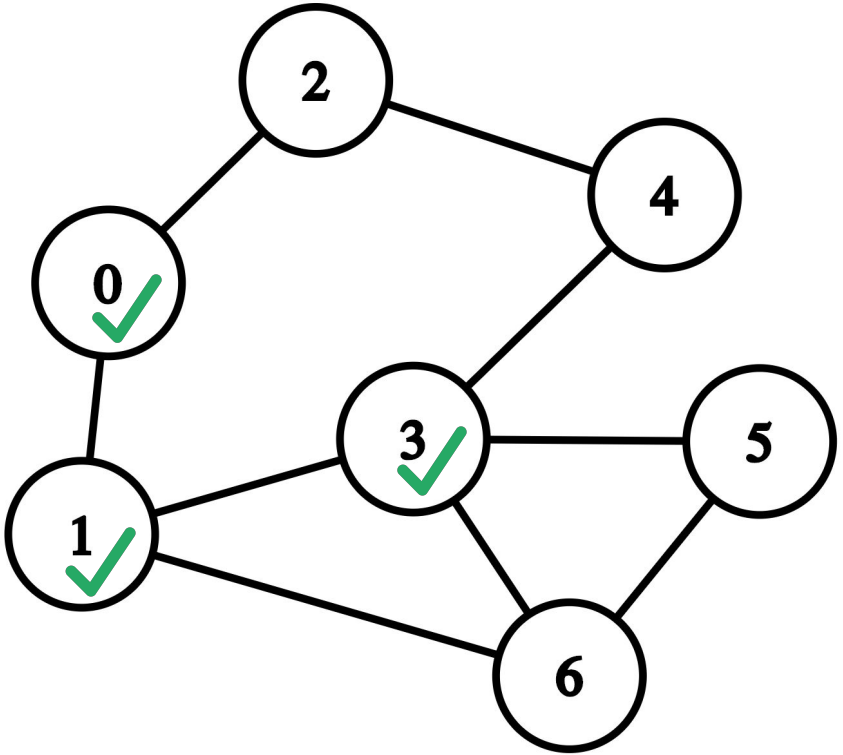



```
CurrentNode=3
```

dfs(0) → **dfs(1)** → **dfs(3)**

DFS

```
>0  1  3
```

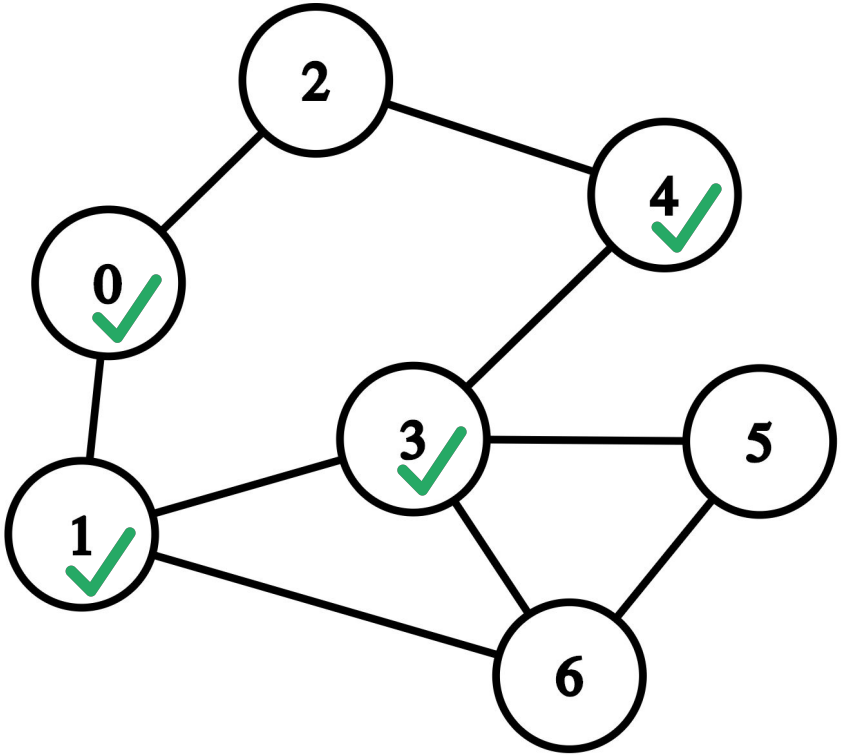


`CurrentNode=`**`3`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → `dfs(4)`

DFS

```
>0  1  3  4
```

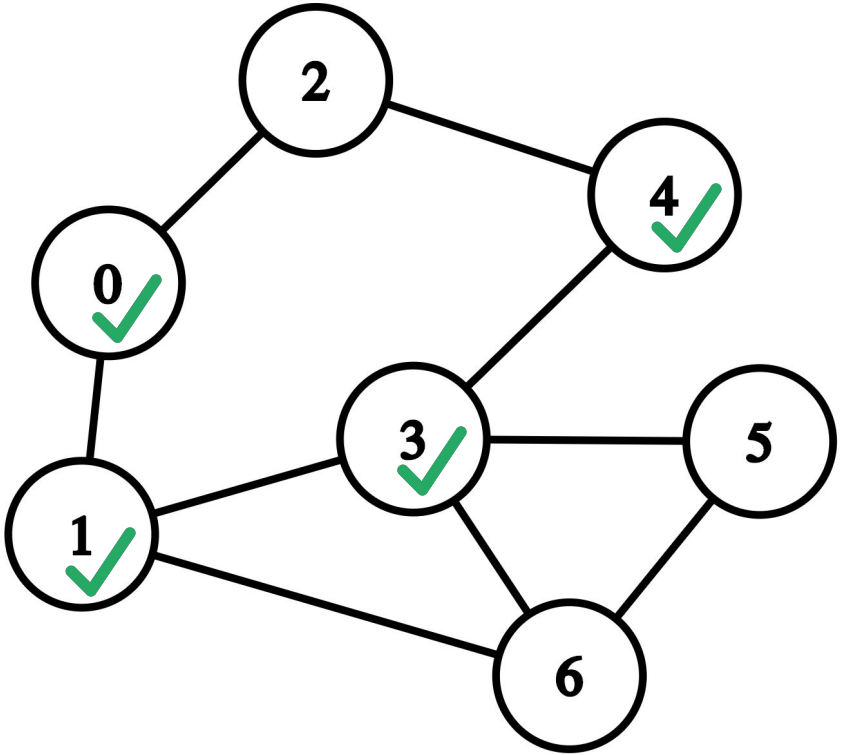


CurrentNode=**4**

dfs(0) → **dfs(1)** → **dfs(3)** → **dfs(4)**

DFS

```
>0 1 3 4
```

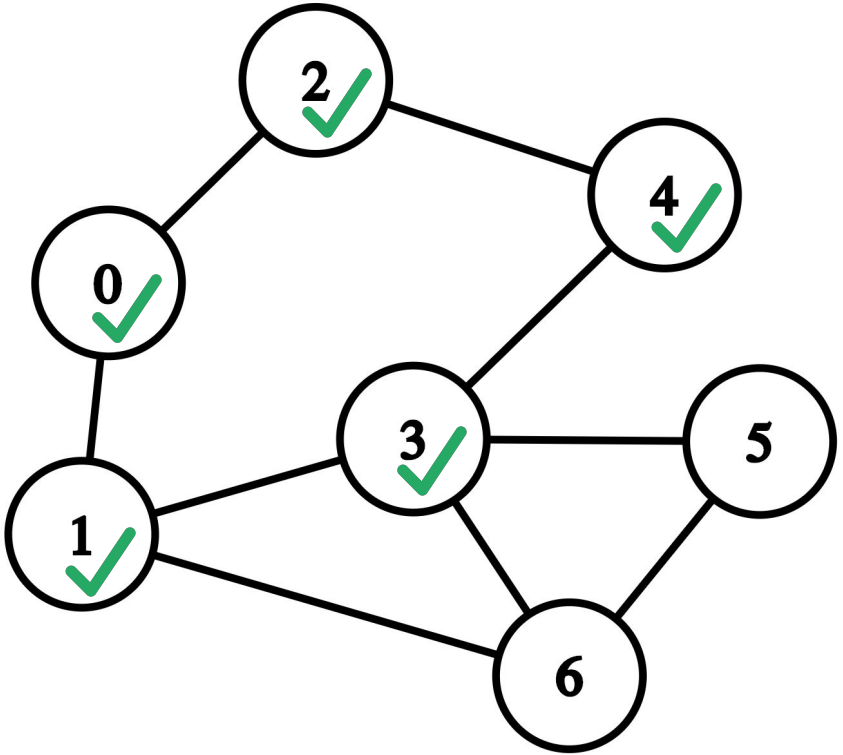


`CurrentNode=`**4**

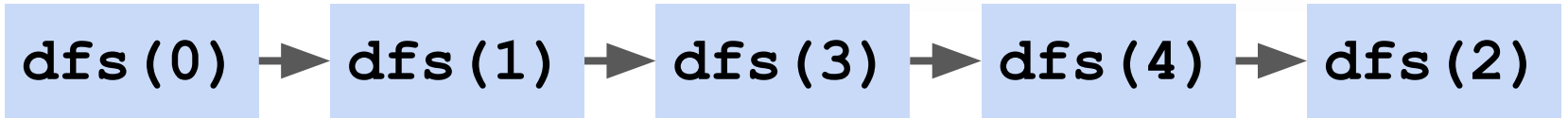
`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(4)`** → `dfs(2)`

DFS

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

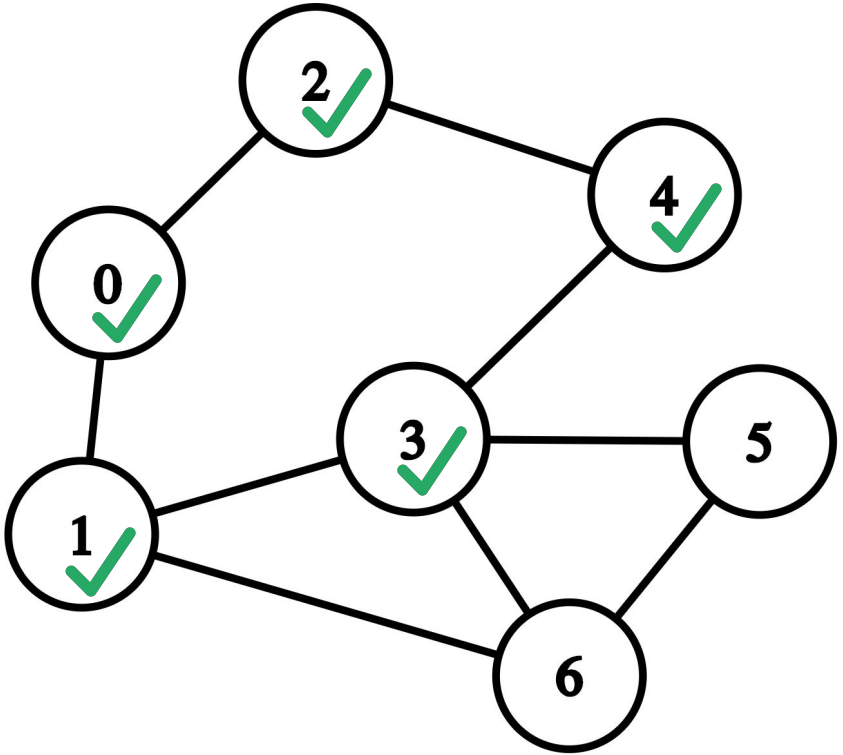


`CurrentNode=`**2**



DFS

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

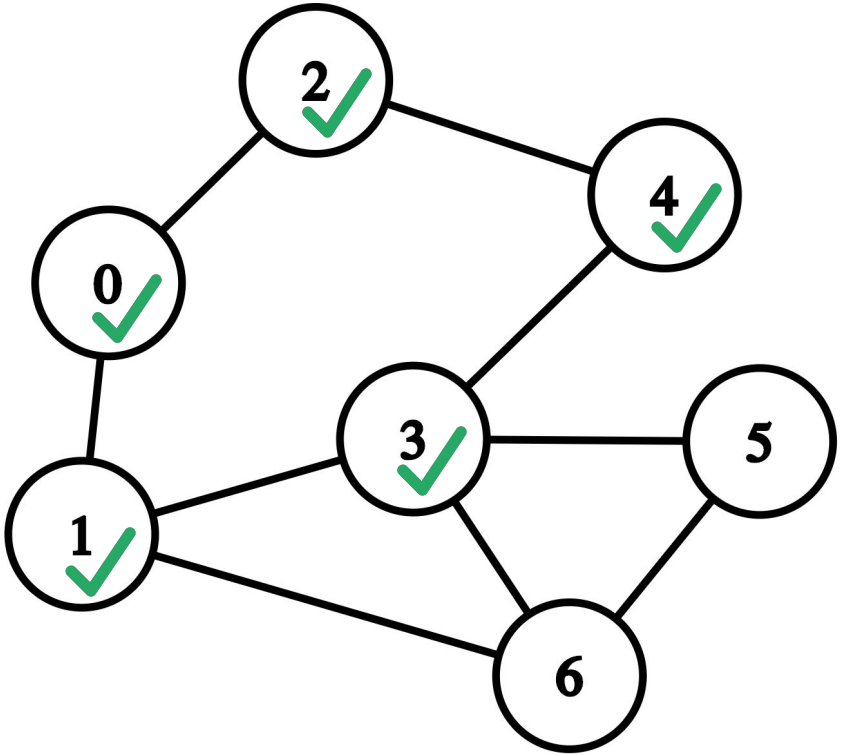


`CurrentNode=`**`4`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(4)`**

DFS

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

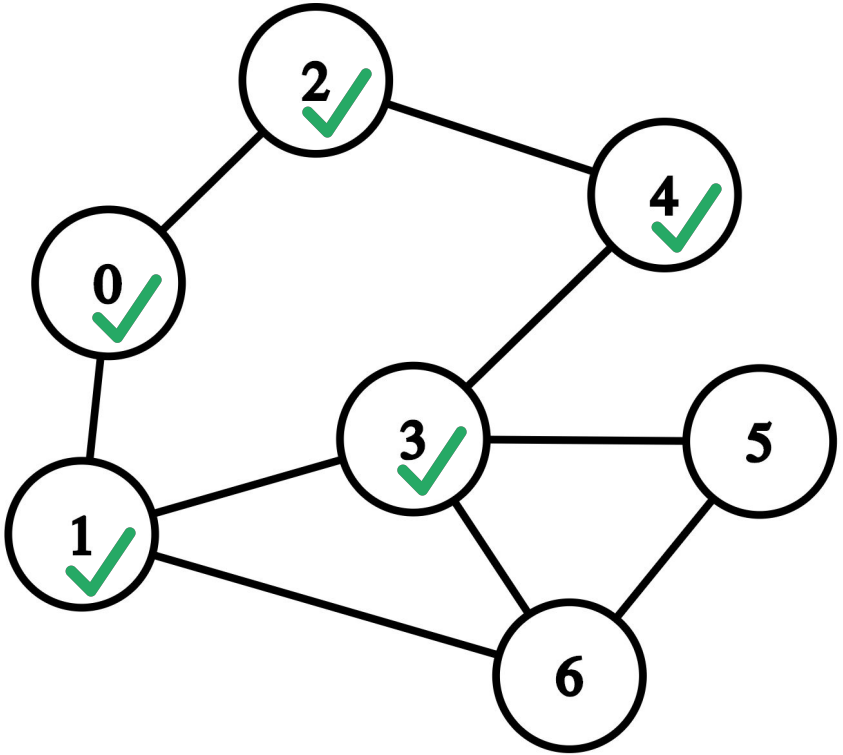



```
CurrentNode=3
```

dfs(0) → **dfs(1)** → **dfs(3)**

DFS

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

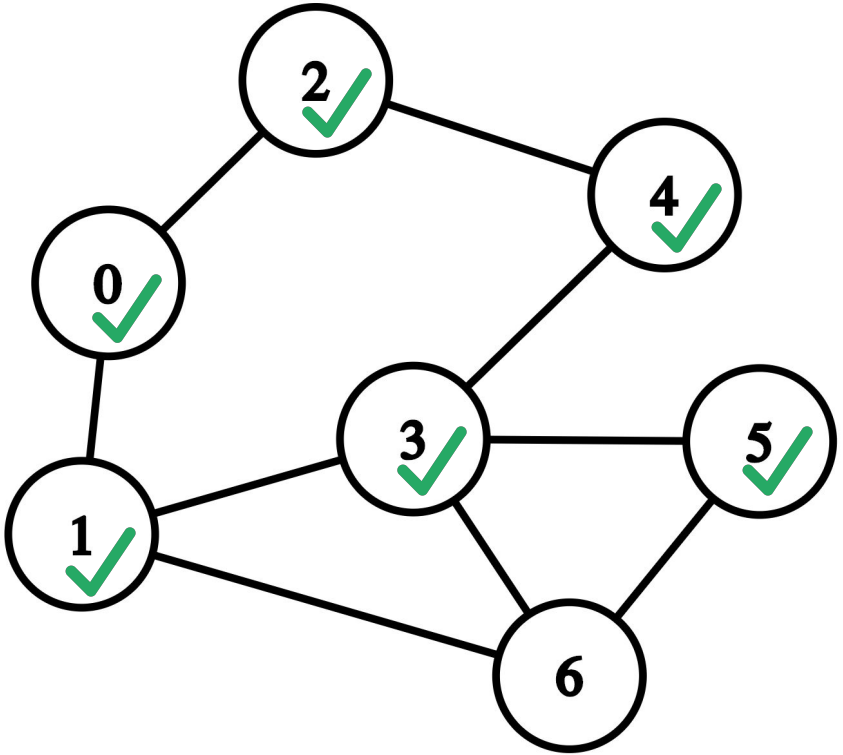


`CurrentNode=`**`3`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → `dfs(5)`

DFS

```
>0 1 3 4 2 5
```

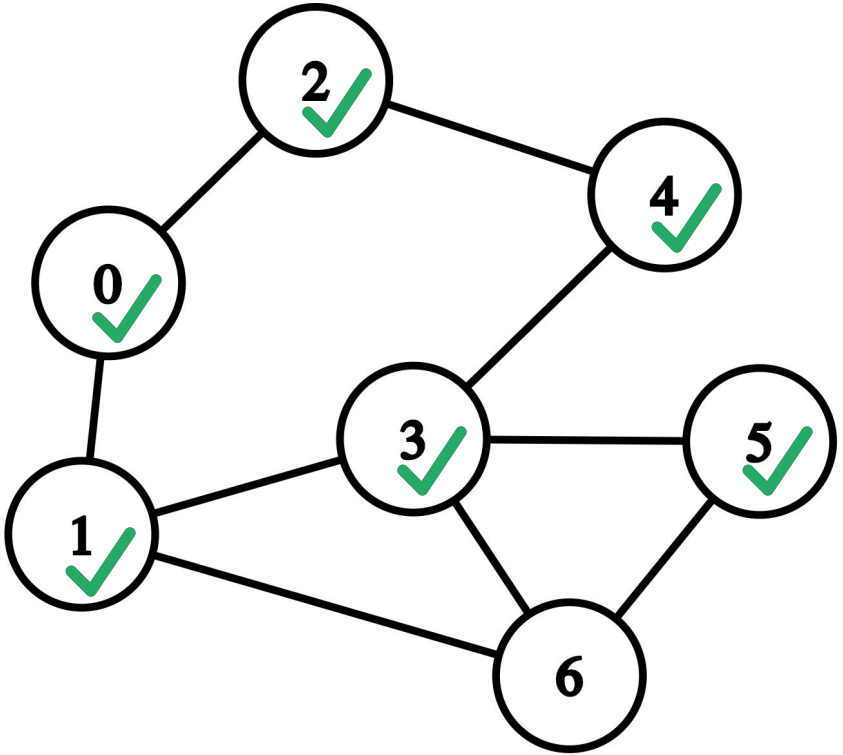


`CurrentNode=`**`5`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(5)`**

DFS

```
>0 1 3 4 2 5
```

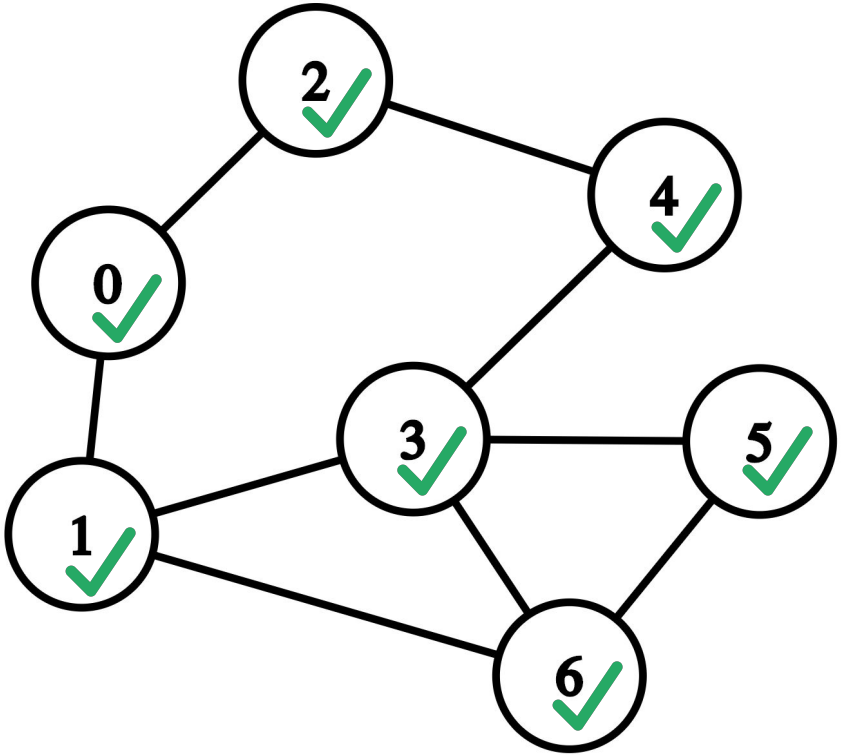


`CurrentNode=`**`5`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(5)`** → `dfs(6)`

DFS

```
>0 1 3 4 2 5 6
```

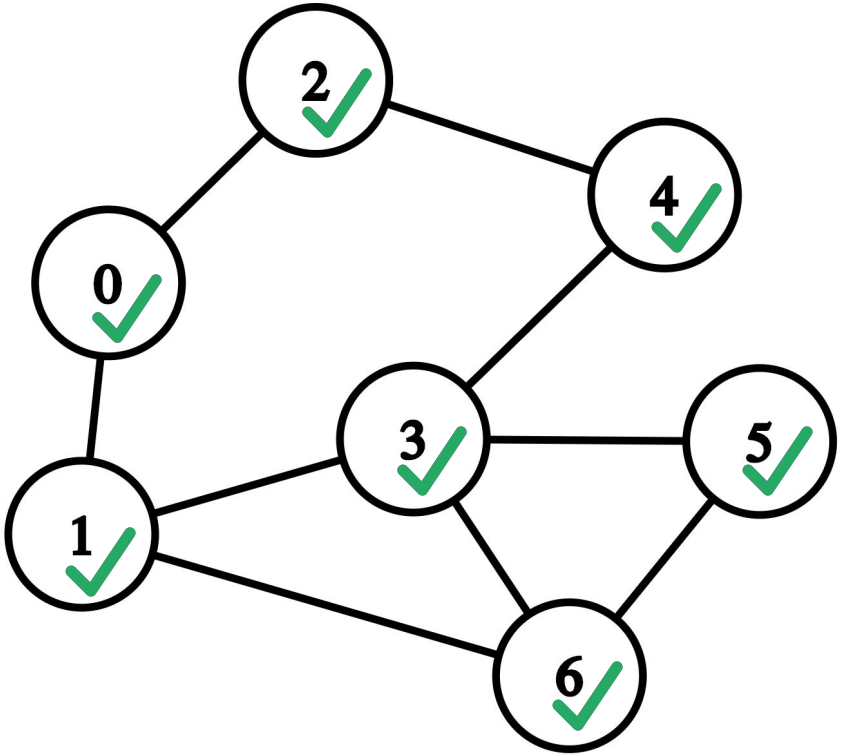


CurrentNode=**6**

dfs(0) → **dfs(1)** → **dfs(3)** → **dfs(5)** → **dfs(6)**

DFS

```
>0 1 3 4 2 5 6
```

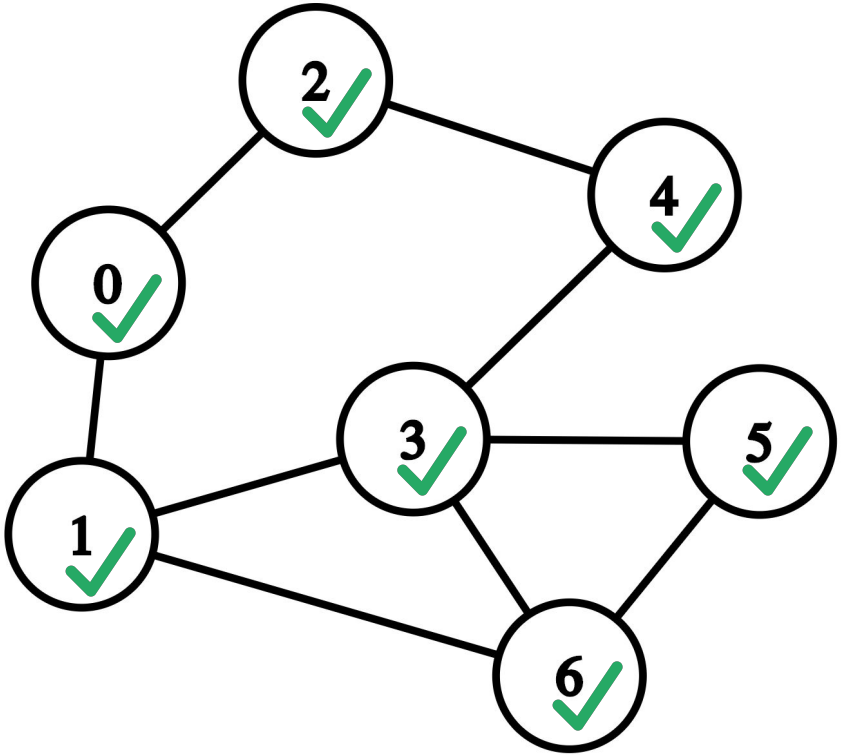


`CurrentNode=`**`5`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(5)`**

DFS

```
>0 1 3 4 2 5 6
```

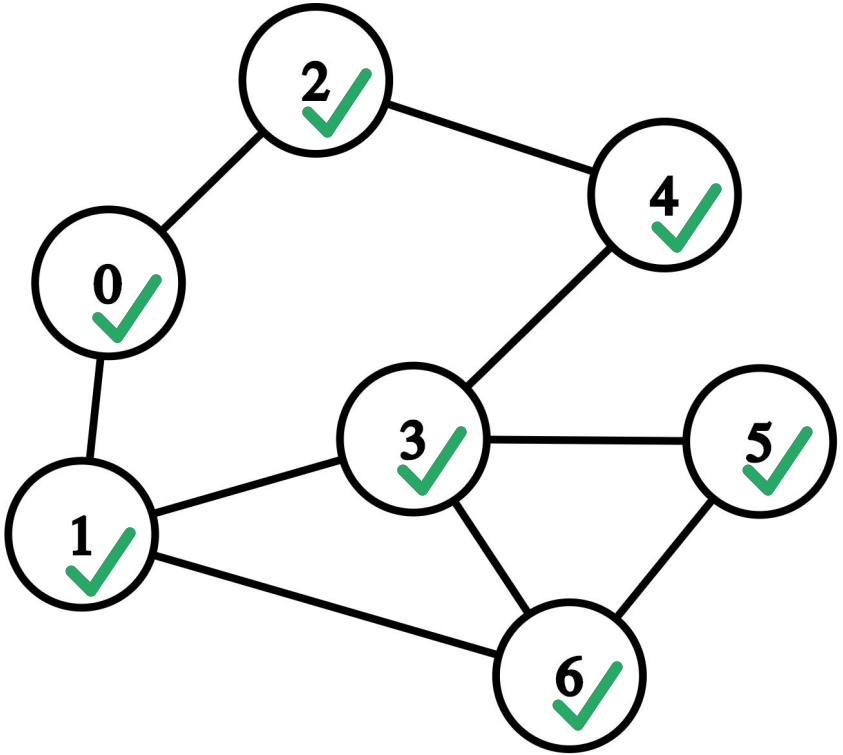



```
CurrentNode=3
```

dfs(0) → **dfs(1)** → **dfs(3)**

DFS

```
>0 1 3 4 2 5 6
```

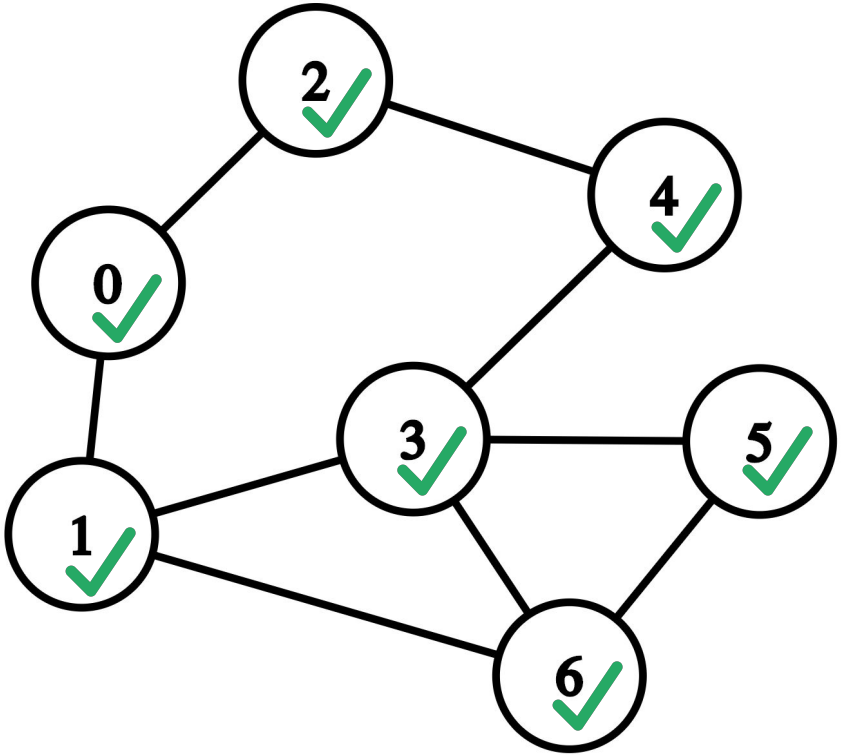



```
CurrentNode=1
```

dfs(0) → **dfs(1)**

DFS

```
>0 1 3 4 2 5 6
```

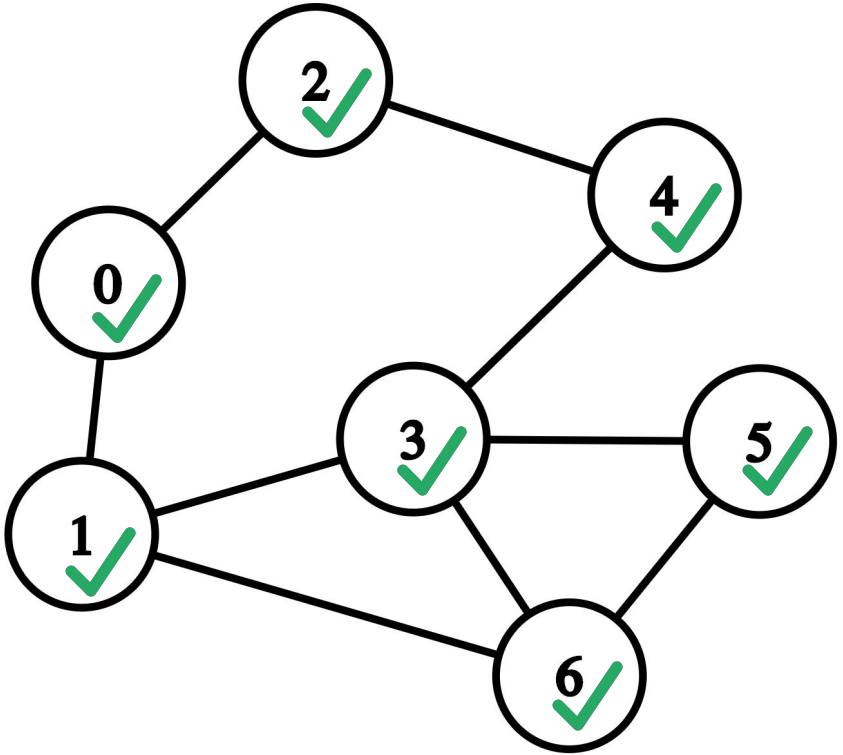



```
CurrentNode=0
```

```
dfs(0)
```

DFS

```
>0 1 3 4 2 5 6
```

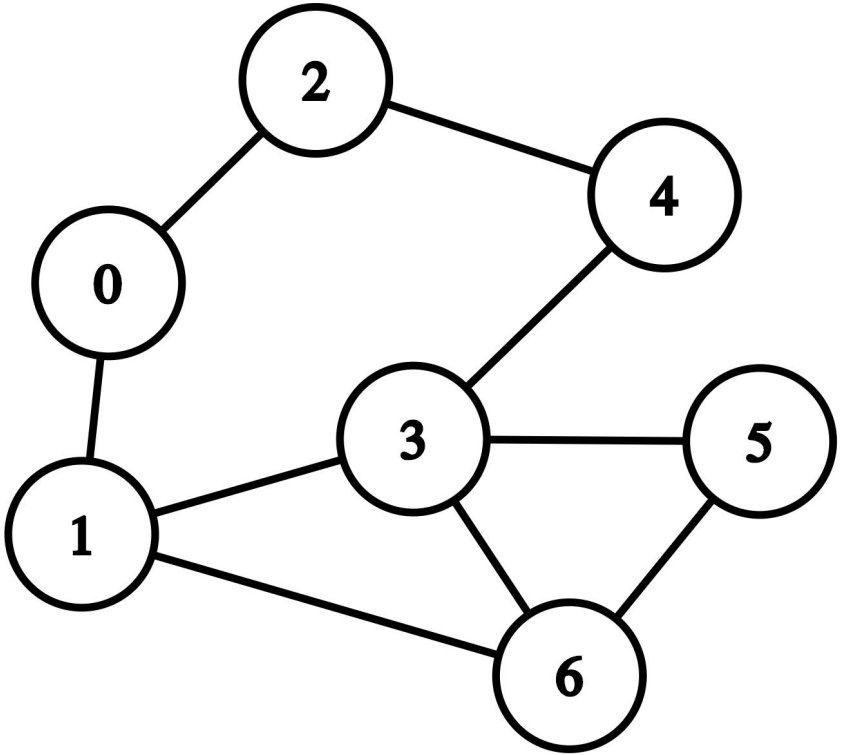



```
done
```

DFS-Tree

DFS-Tree

>

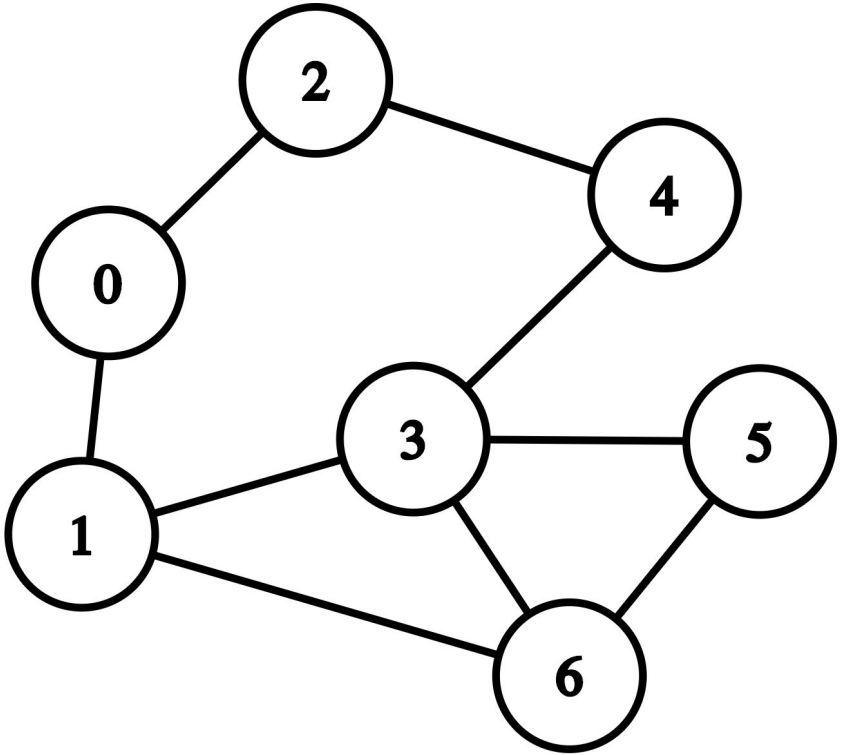



```
dfs(0)
```

DFS-Tree

>0

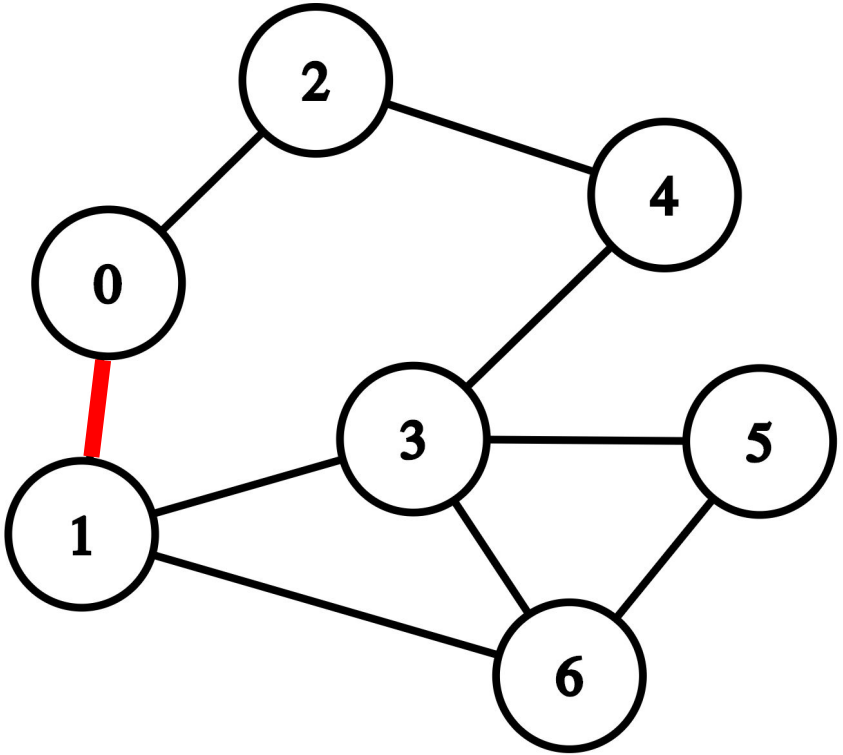
CurrentNode=**0**




```
dfs(0)
```

DFS-Tree

>0

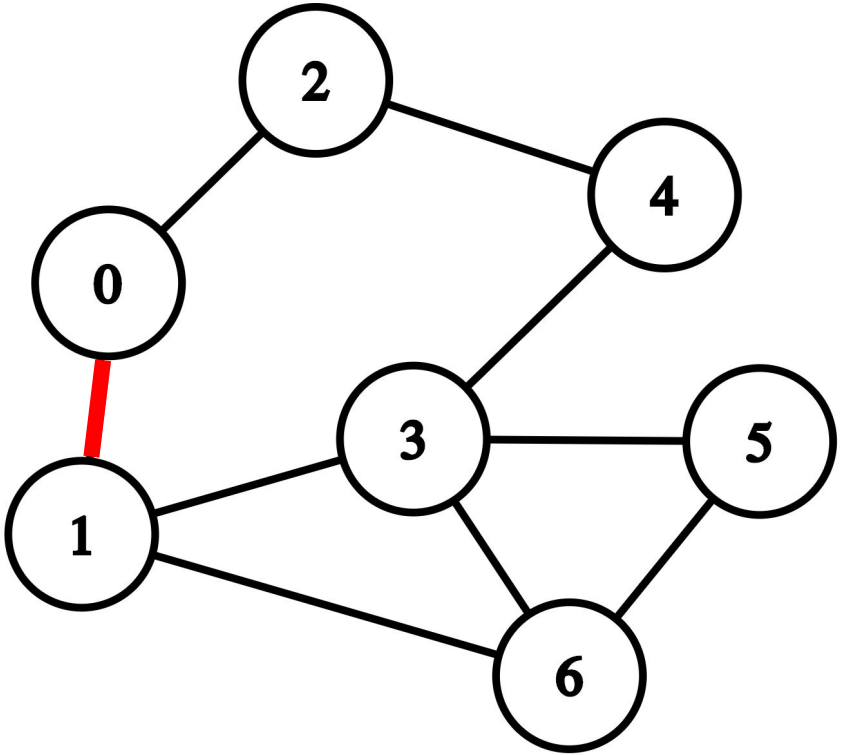


CurrentNode=**0**

dfs(0) → dfs(1)

DFS-Tree

```
>0  1
```

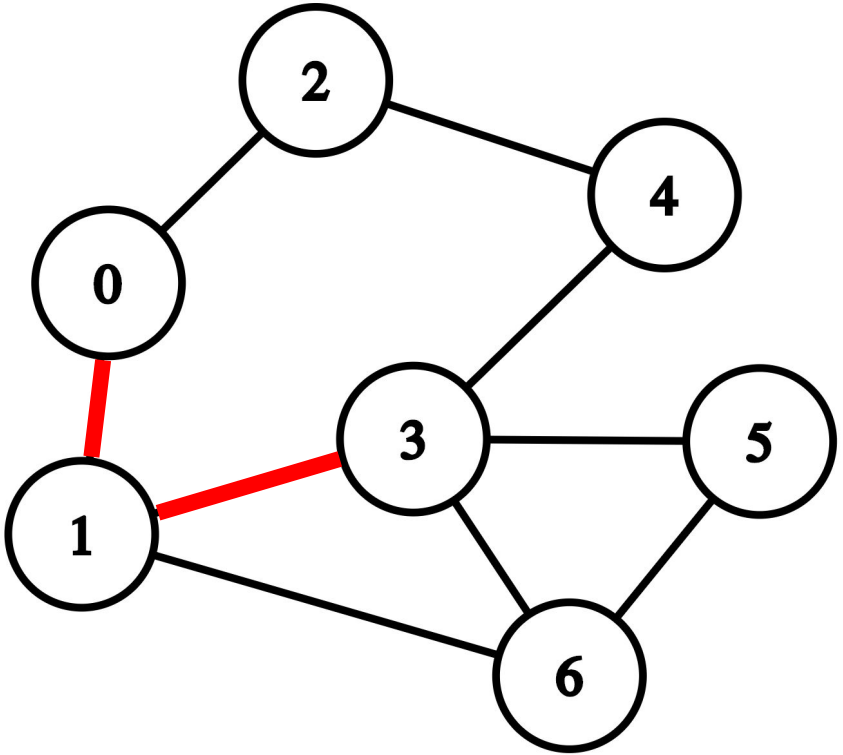



```
CurrentNode=1
```

dfs(0) → **dfs(1)**

DFS-Tree

```
>0 1
```

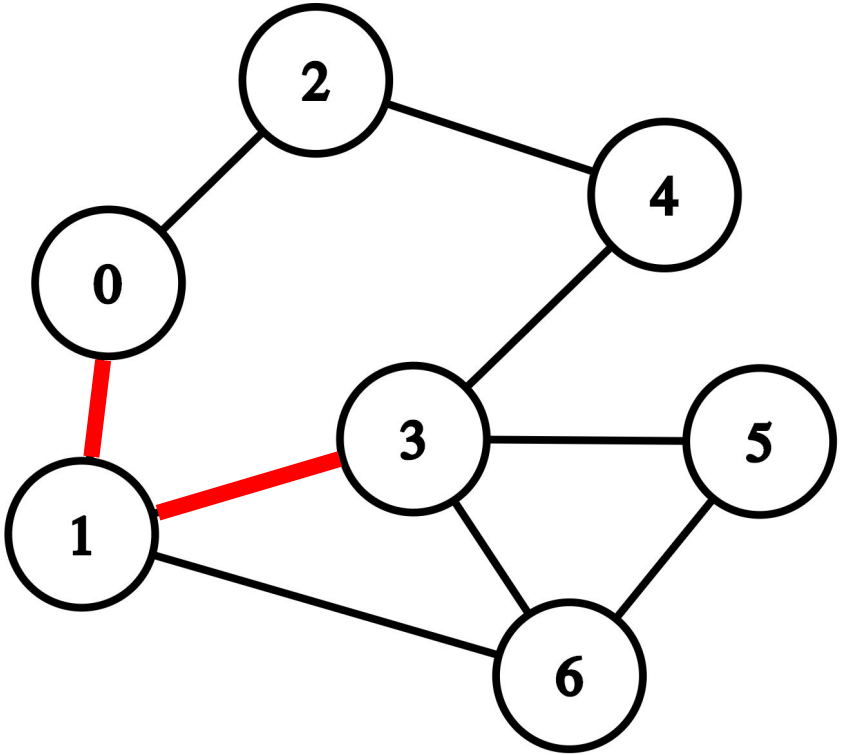



```
CurrentNode=1
```

dfs(0) → **dfs(1)** → dfs(3)

DFS-Tree

```
>0  1  3
```

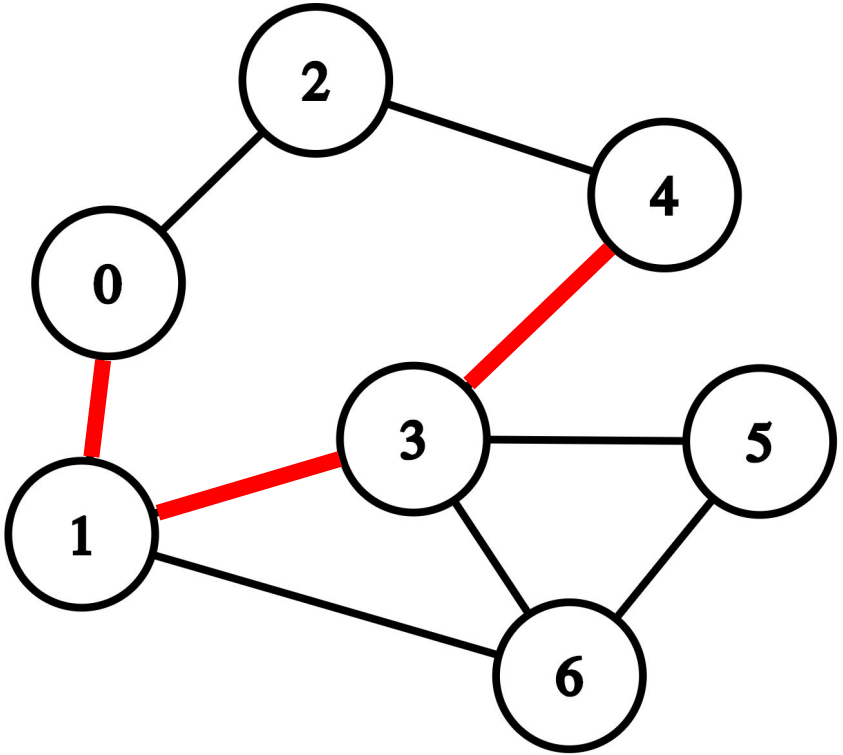


`CurrentNode=`**`3`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`**

DFS-Tree

```
>0 1 3
```

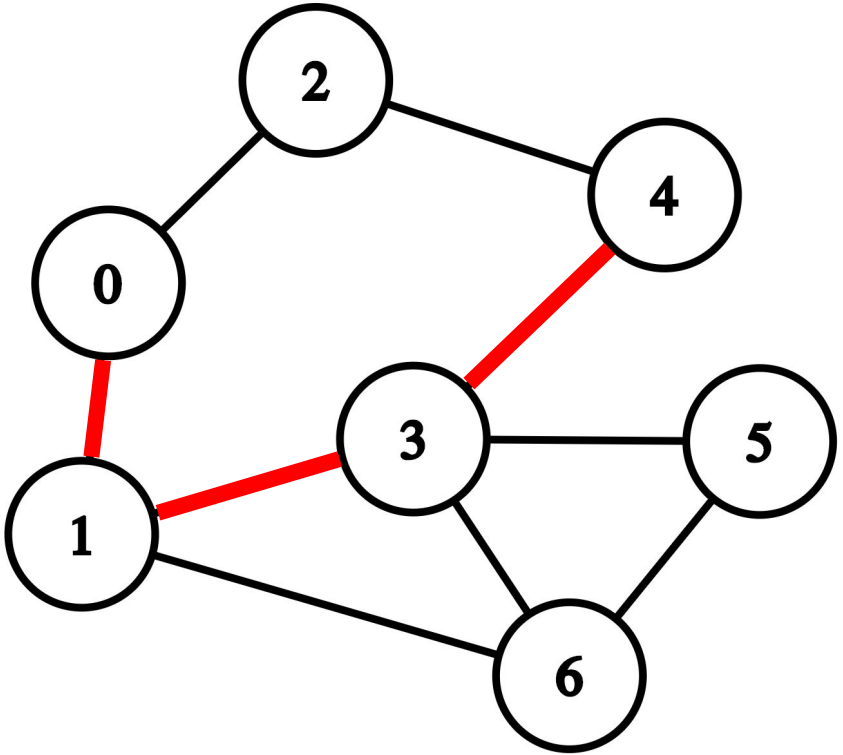


`CurrentNode=`**`3`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → `dfs(4)`

DFS-Tree

```
>0  1  3  4
```

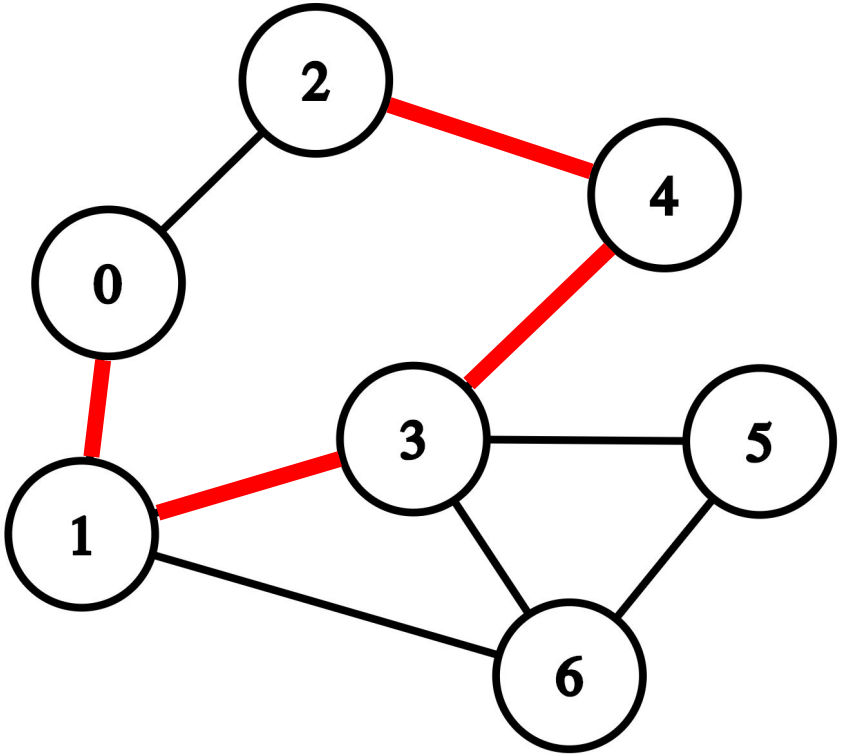


`CurrentNode=`**`4`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(4)`**

DFS-Tree

```
>0  1  3  4
```

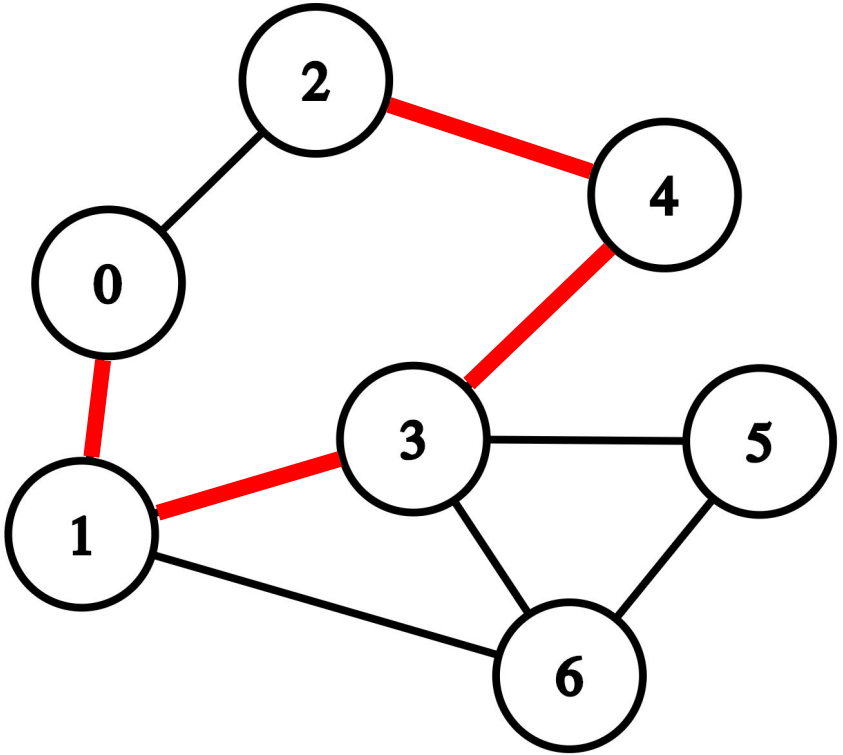


`CurrentNode=`**4**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(4)`** → `dfs(2)`

DFS-Tree

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

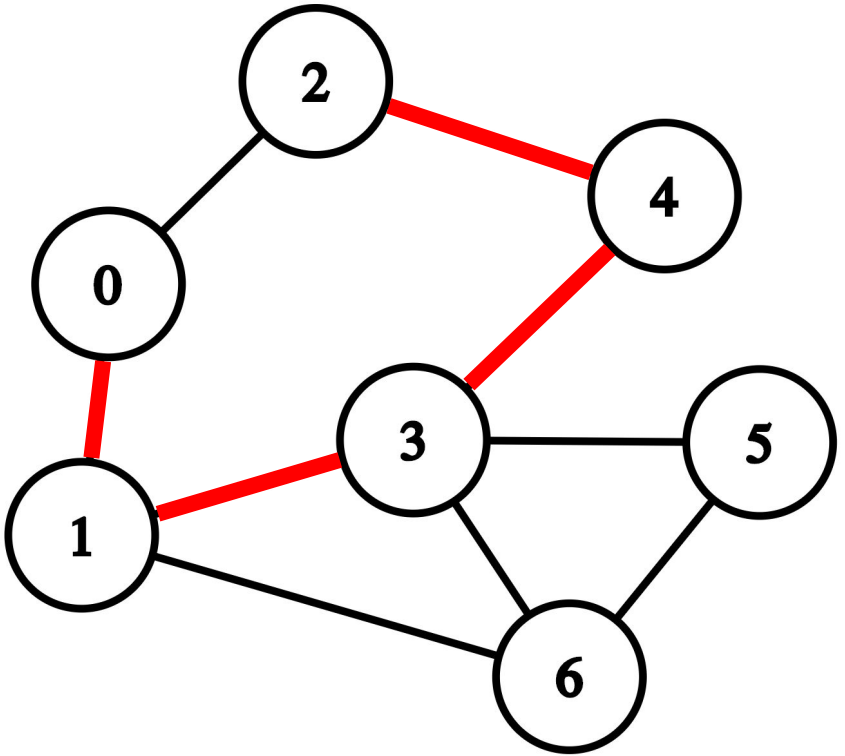


CurrentNode=**2**

dfs(0) → **dfs(1)** → **dfs(3)** → **dfs(4)** → **dfs(2)**

DFS-Tree

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

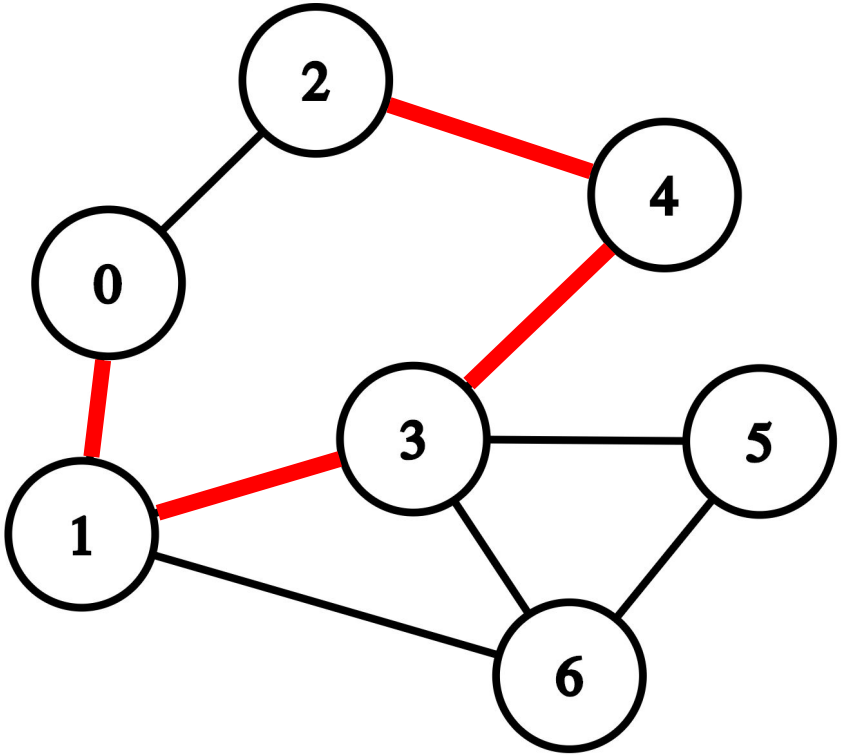


`CurrentNode=`**4**

dfs(0) → **dfs(1)** → **dfs(3)** → **dfs(4)**

DFS-Tree

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

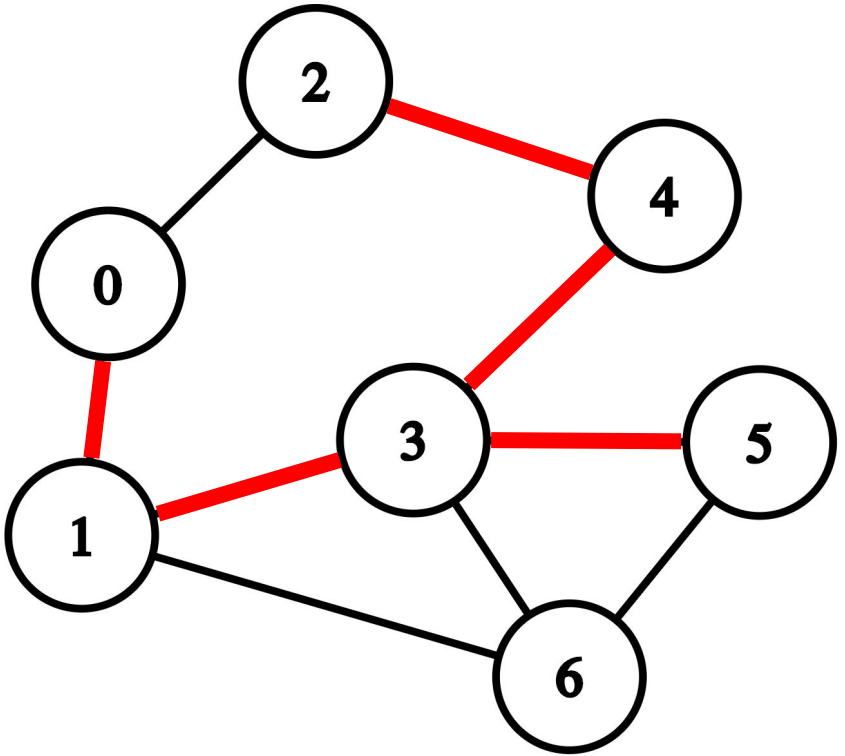



```
CurrentNode=3
```

dfs(0) → **dfs(1)** → **dfs(3)**

DFS-Tree

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

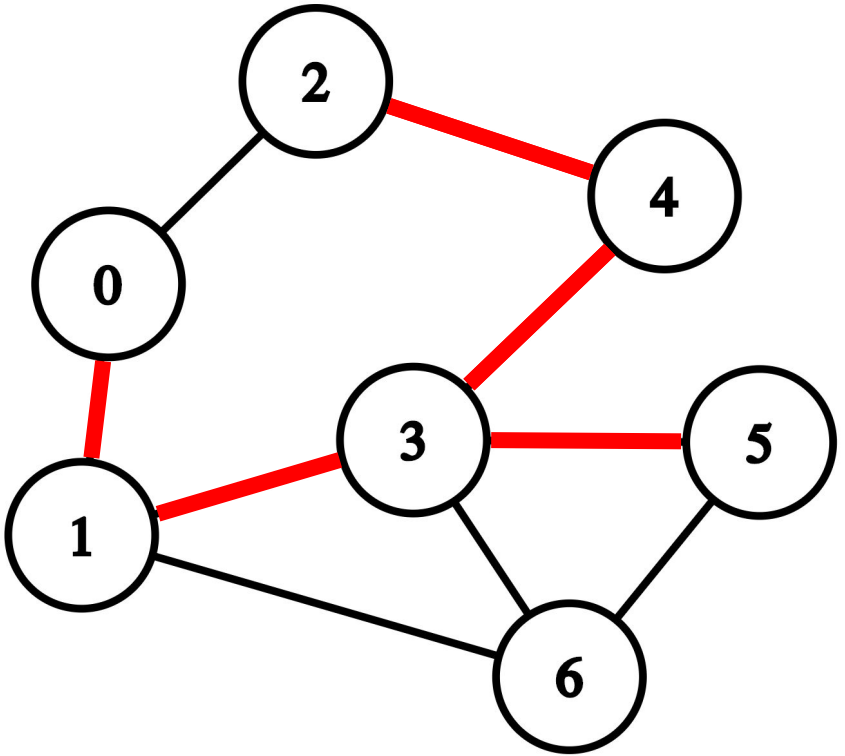


`CurrentNode=`**`3`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → `dfs(5)`

DFS-Tree

```
>0 1 3 4 2 5
```

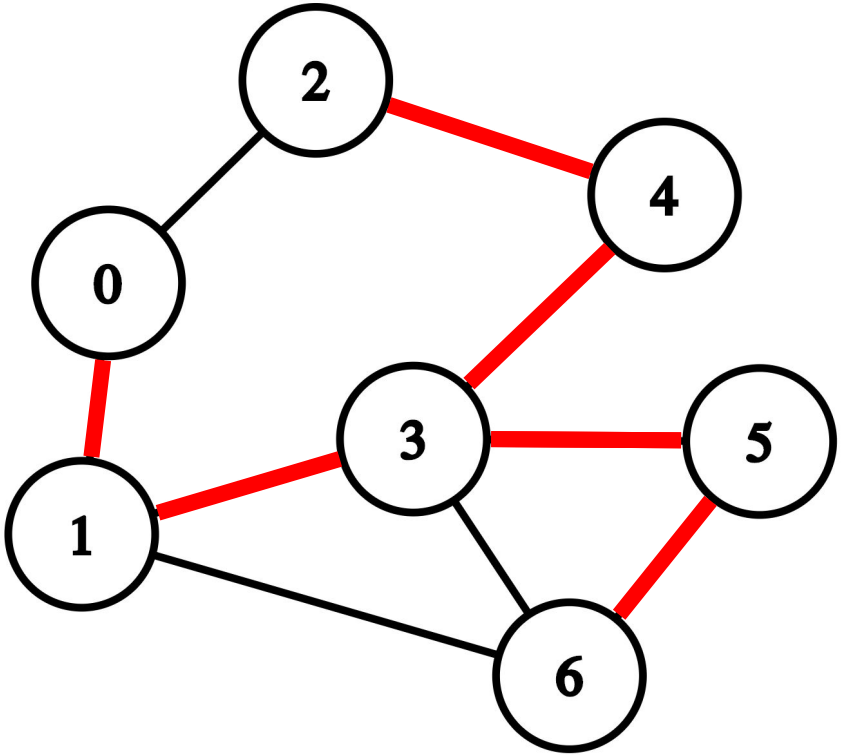


`CurrentNode=`**`5`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(5)`**

DFS-Tree

```
>0 1 3 4 2 5
```

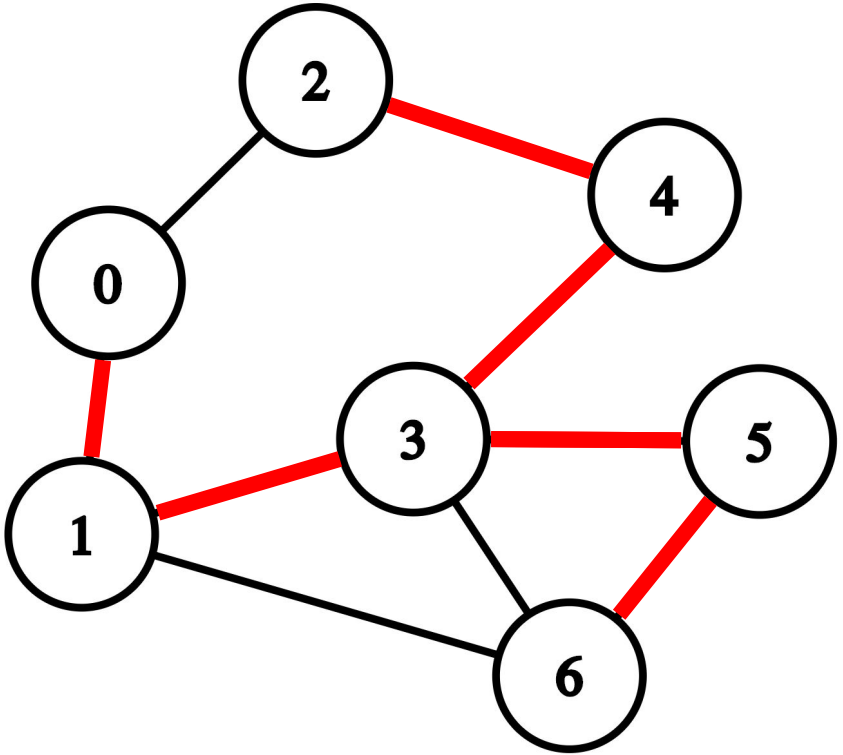


`CurrentNode=`**`5`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(5)`** → `dfs(6)`

DFS-Tree

```
>0 1 3 4 2 5 6
```

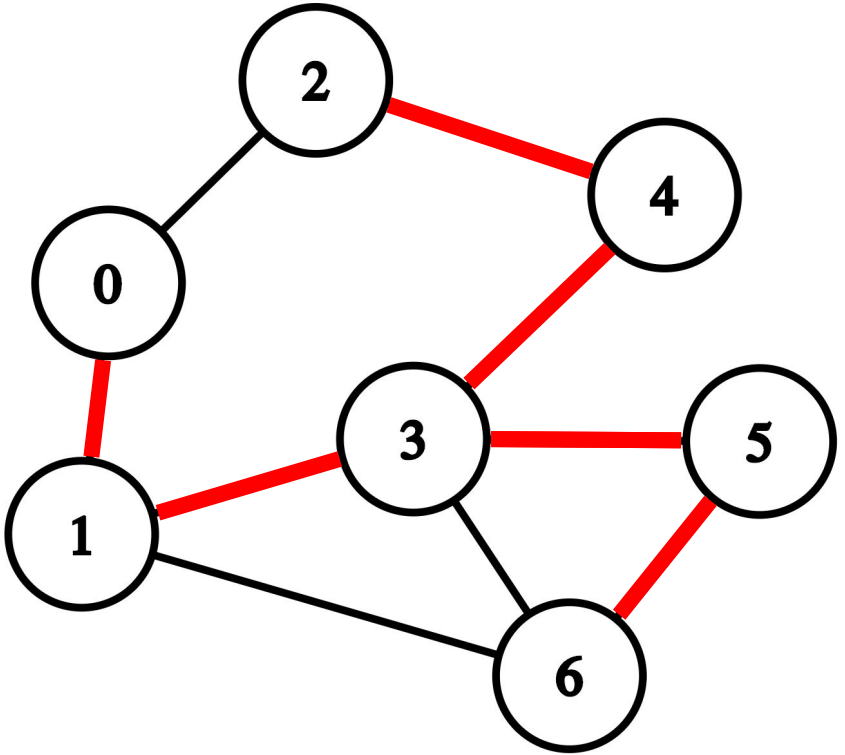


CurrentNode=**6**

dfs(0) → **dfs(1)** → **dfs(3)** → **dfs(5)** → **dfs(6)**

DFS-Tree

```
>0 1 3 4 2 5 6
```

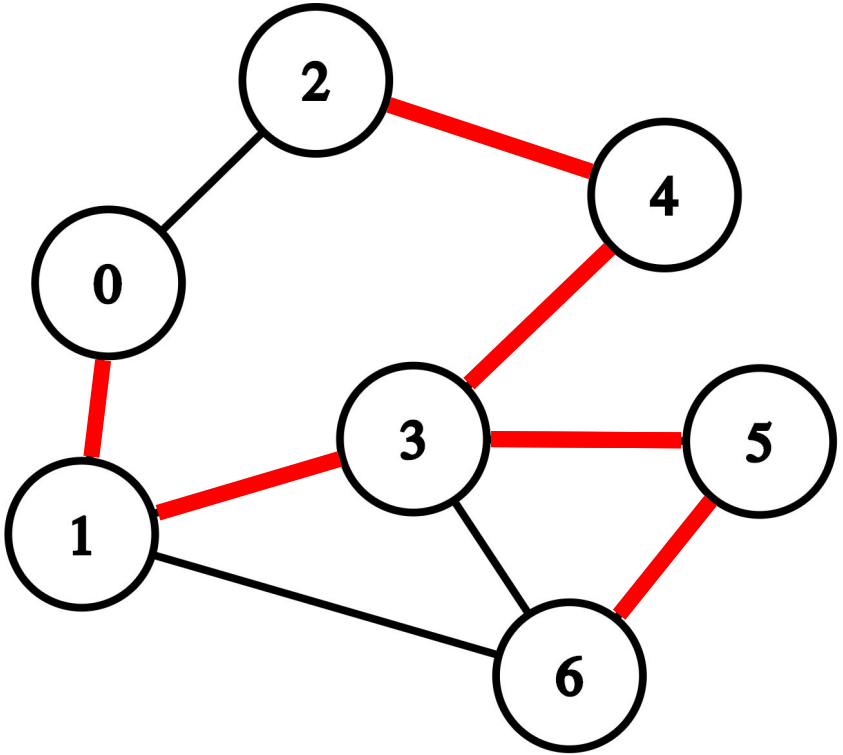


`CurrentNode=`**`5`**

`dfs(0)` → **`dfs(1)`** → **`dfs(3)`** → **`dfs(5)`**

DFS-Tree

```
>0 1 3 4 2 5 6
```

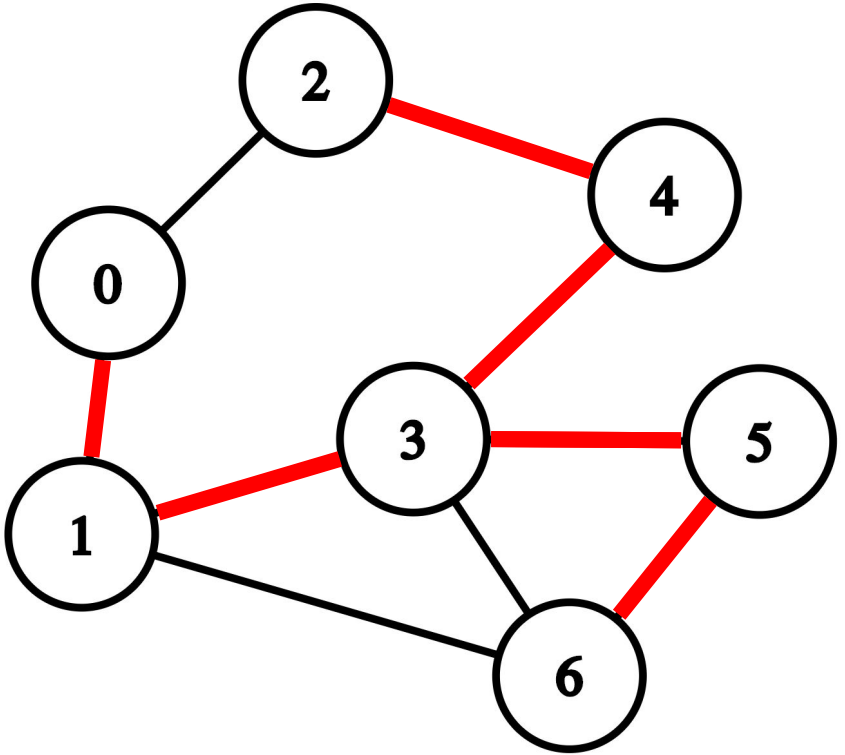



```
CurrentNode=3
```

dfs(0) → **dfs(1)** → **dfs(3)**

DFS-Tree

```
>0 1 3 4 2 5 6
```

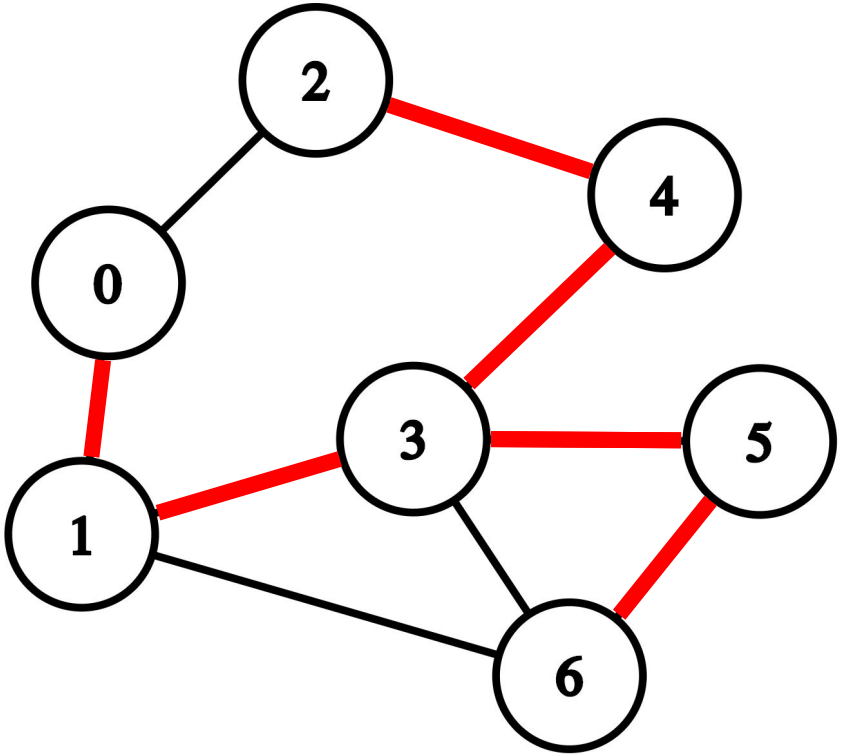


CurrentNode=**1**

dfs(0) → **dfs(1)**

DFS-Tree

```
>0 1 3 4 2 5 6
```

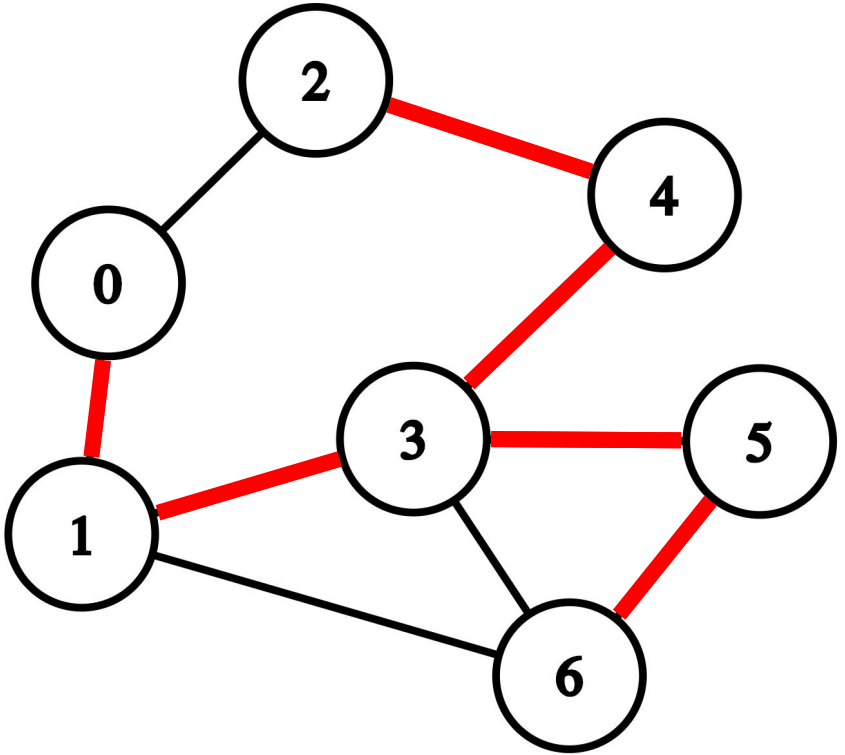



```
CurrentNode=0
```

```
dfs(0)
```

DFS-Tree

```
>0 1 3 4 2 5 6
```



we have found the DFS-Tree

Further Applications of DFS

Further Applications of DFS

Is a graph connected?

Check if every vertex is visited after DFS from any vertex

Is vertex **B** reachable from **A**?

Check if **B** is visited after DFS from **A**

Finding Cycles

Finding Cycles

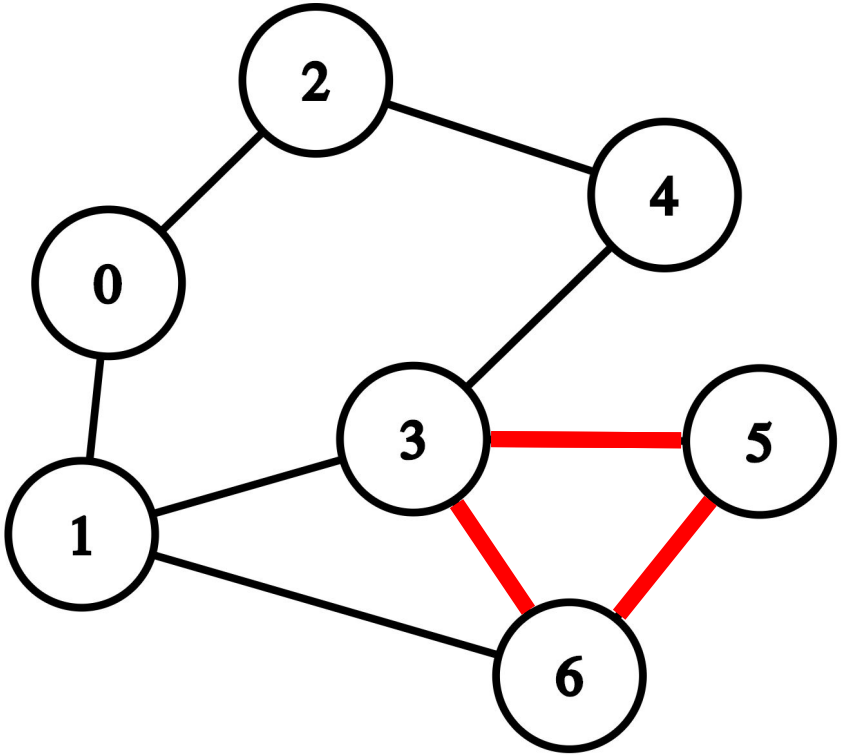
We can use DFS to check if a graph contains cycles

If we reach a visited vertex, we have found a cycle

But: Be careful, single edges do not count as cycles

Keep track of parent-vertex

Finding Cycles



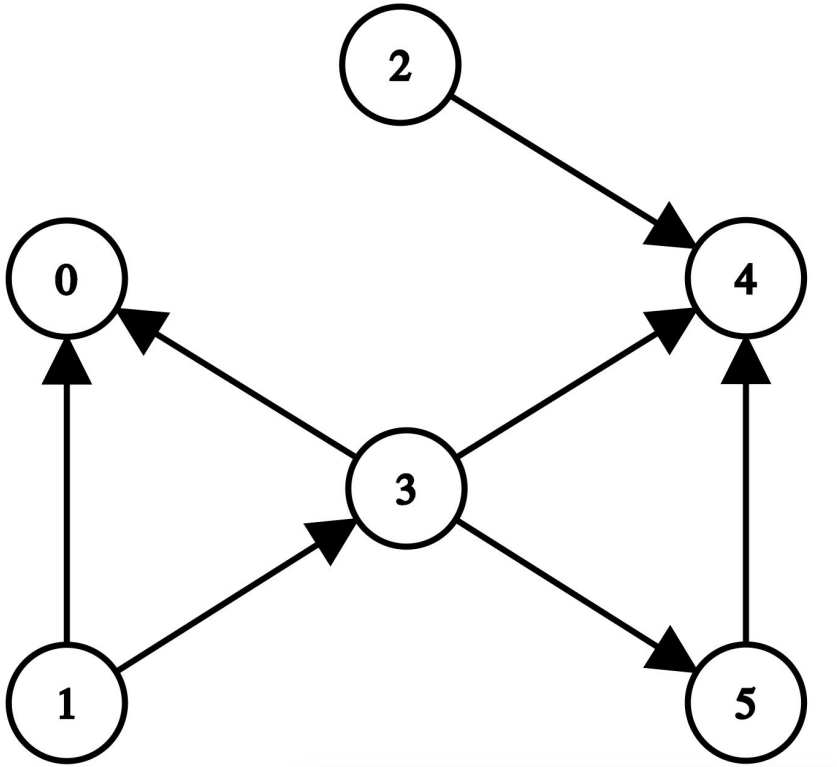
Finding Cycles in a Directed Graph

Finding Cycles in a Directed Graph

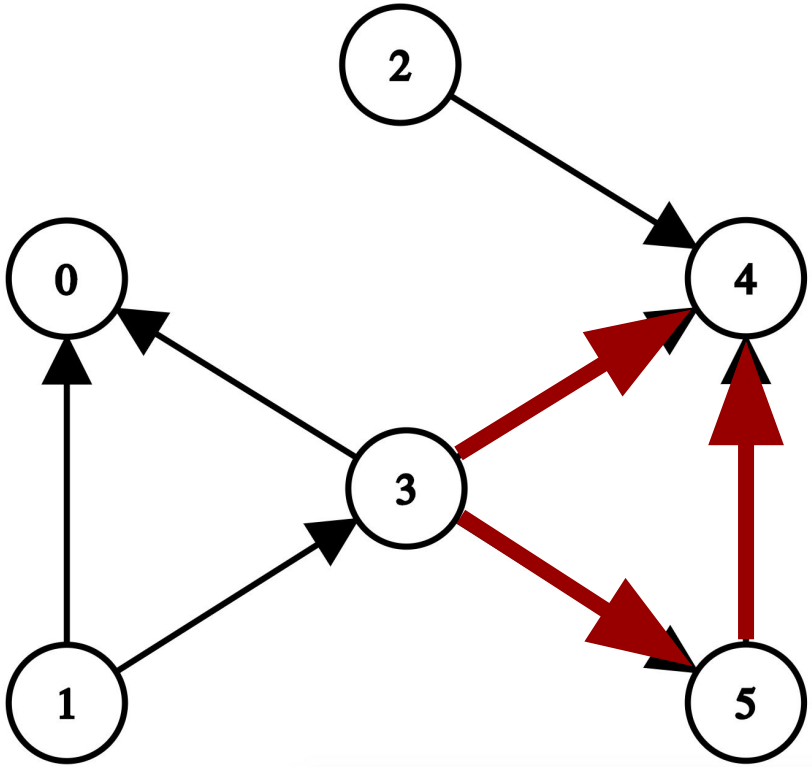
Same strategy as before?

Sounds good, does not work!

Finding Cycles in a Directed Graph



Finding Cycles in a Directed Graph



This is not a cycle, but our algorithm would identify it as one

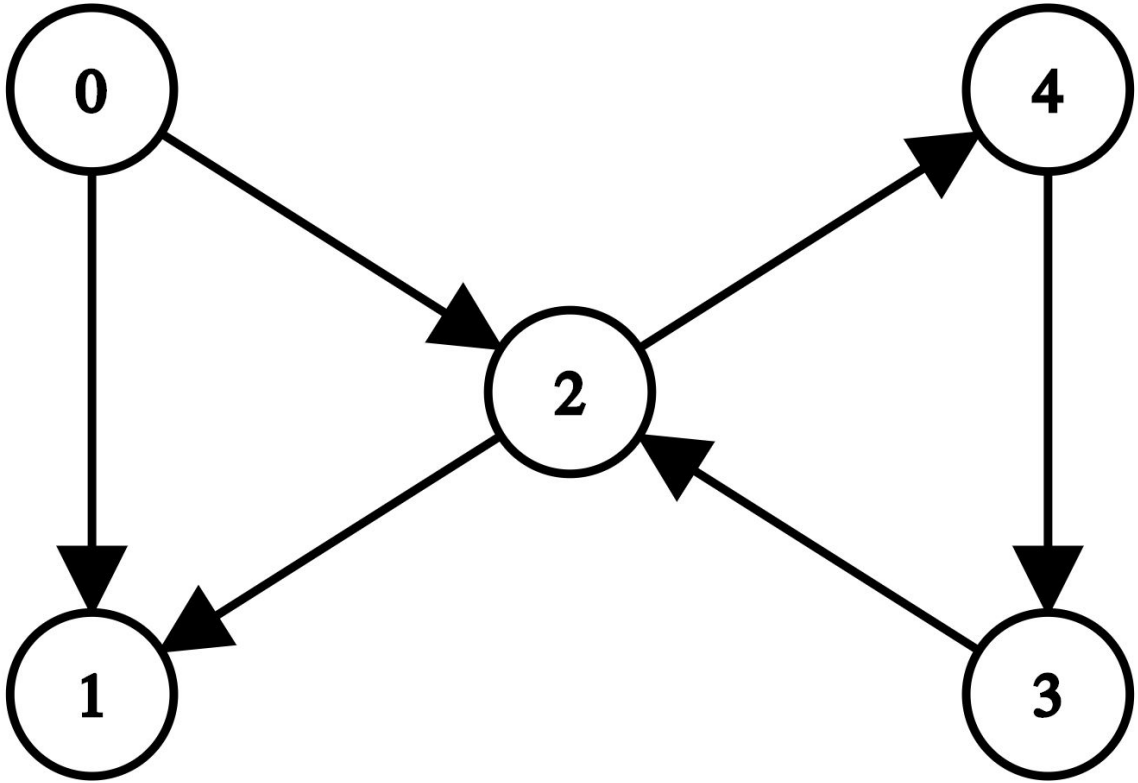
3 to 4
3 to 5
5 to 4, but 4 is visited

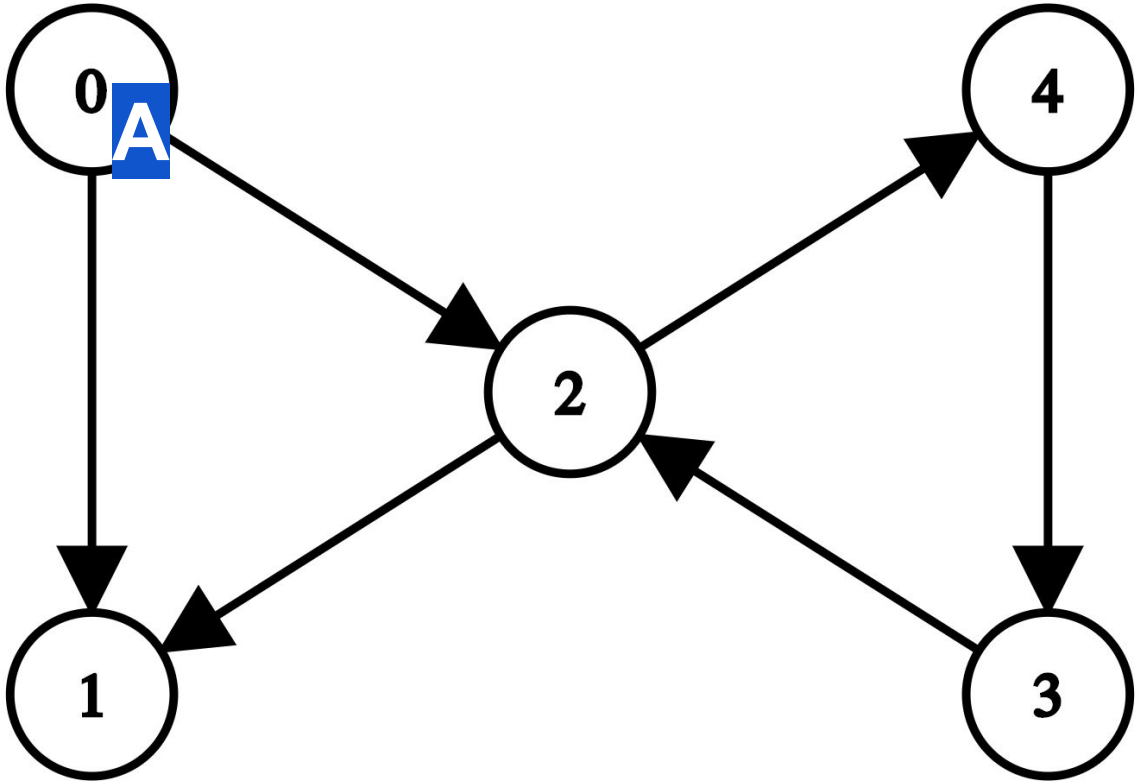
Finding Cycles in a Directed Graph

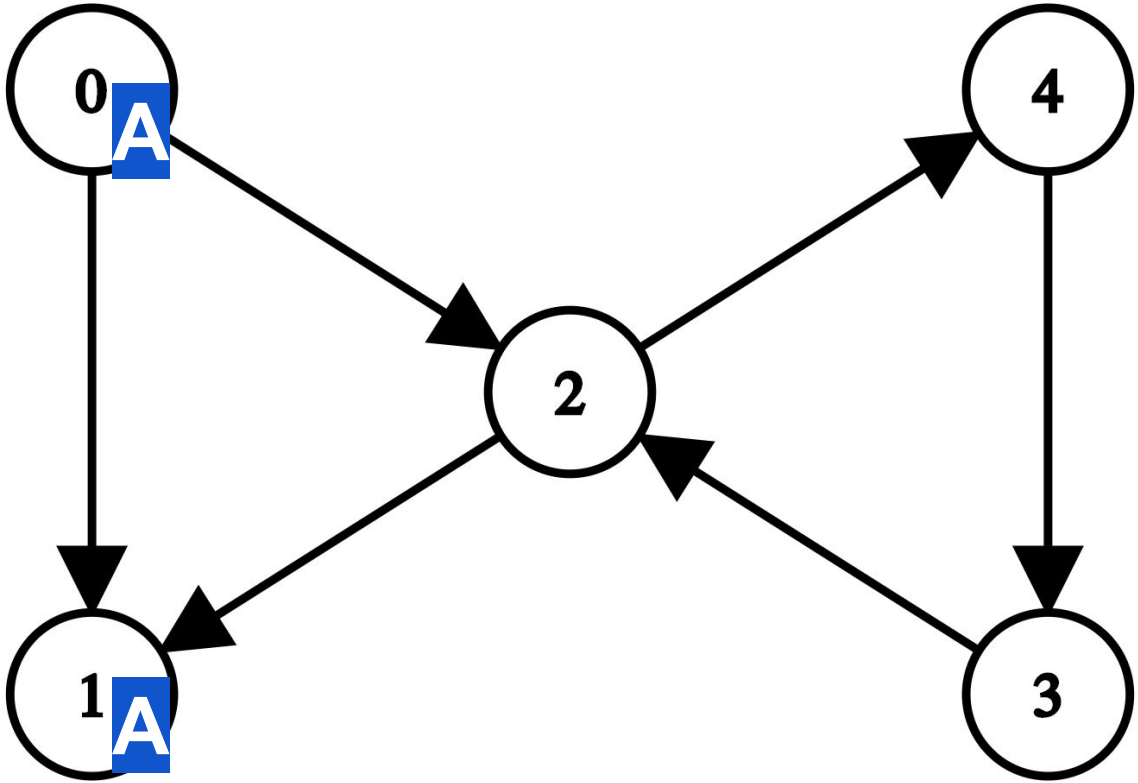
Instead of just marking a vertex as visited, keep track of whether it is in our current path

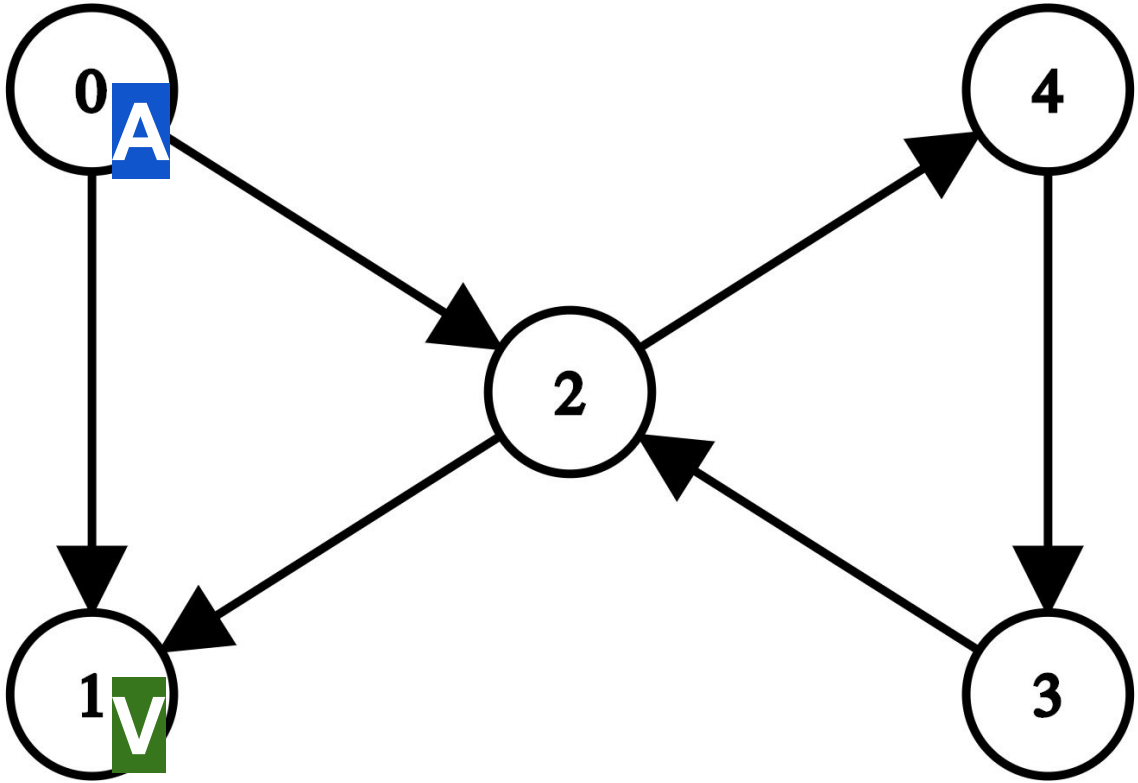
Thus, every vertex has three states (**unvisited, active, visited**)

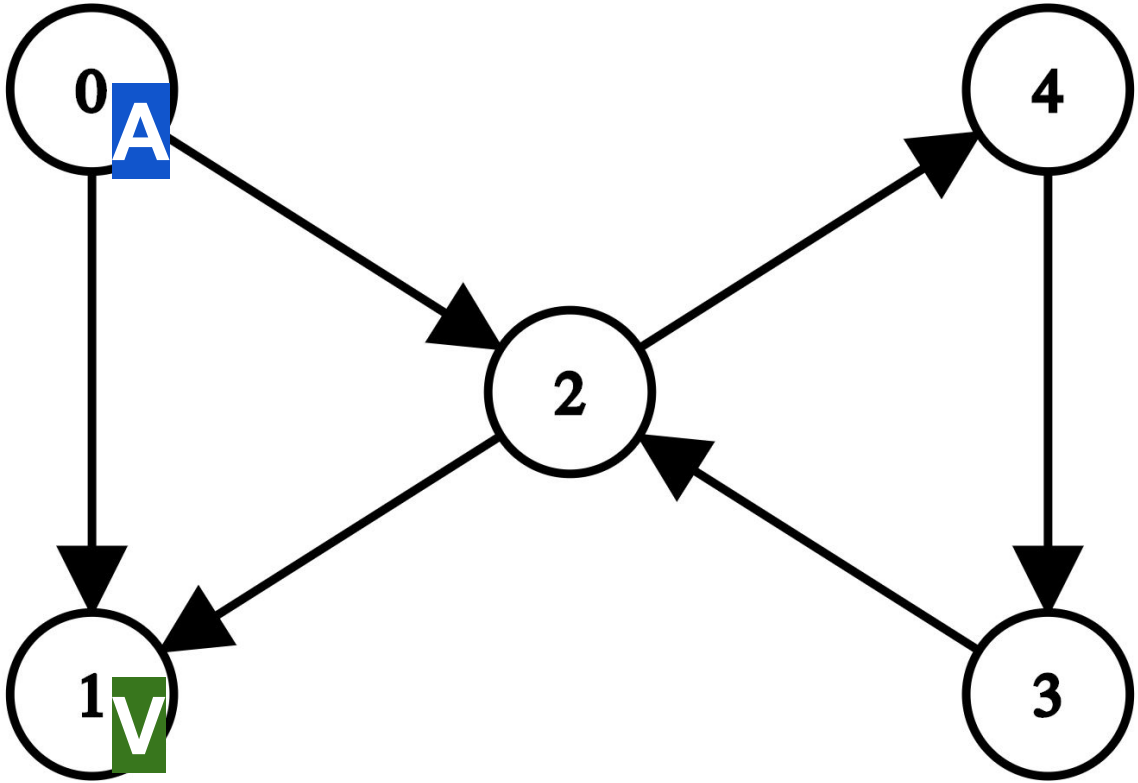
When we reach an active vertex, it is a cycle

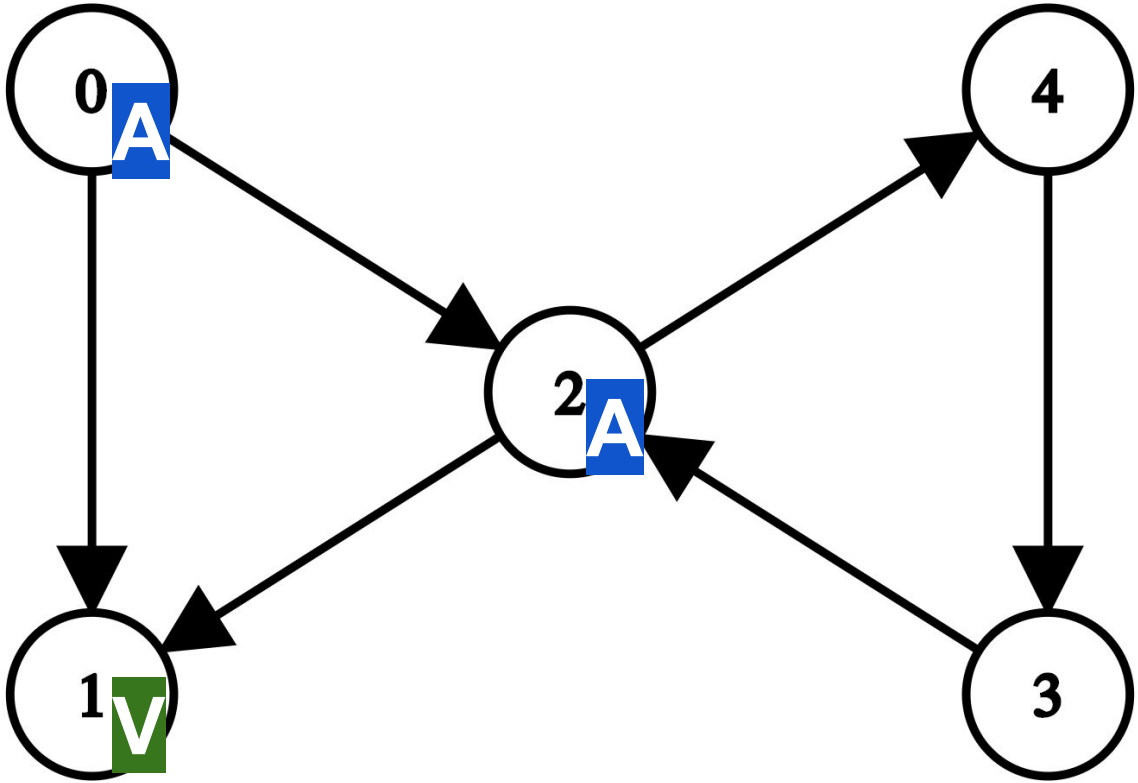

0
4
2
1
3

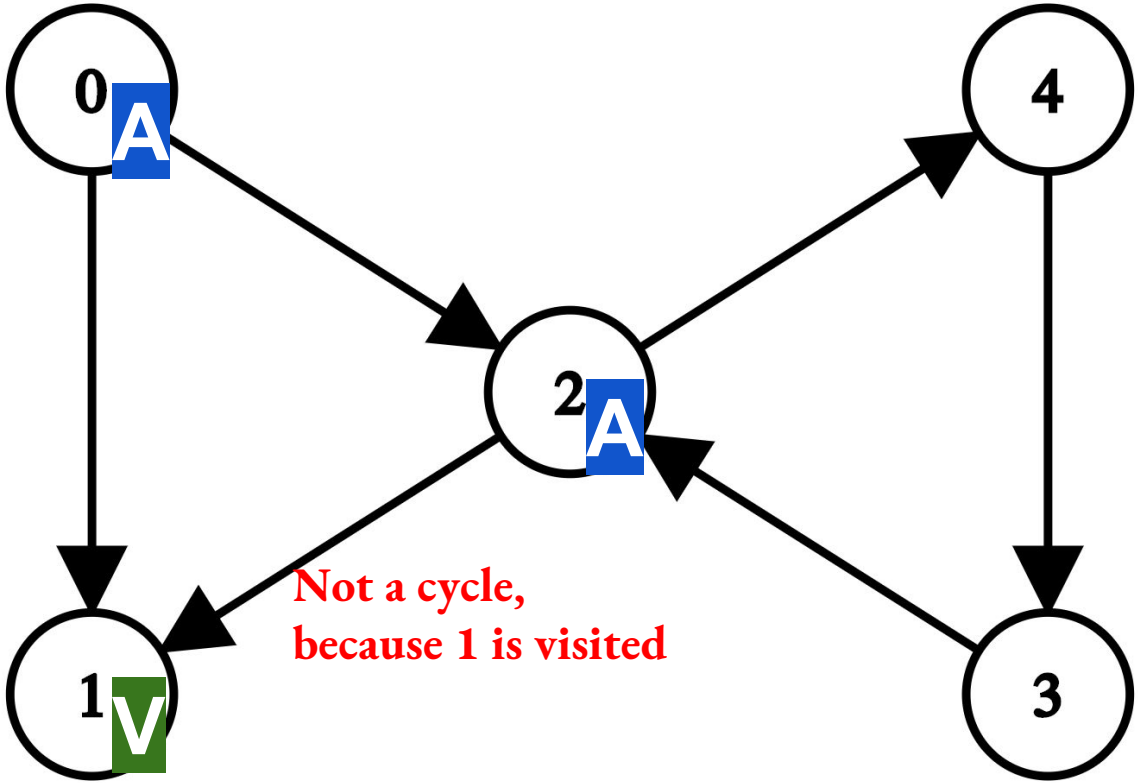

0
A
4
2
1
3

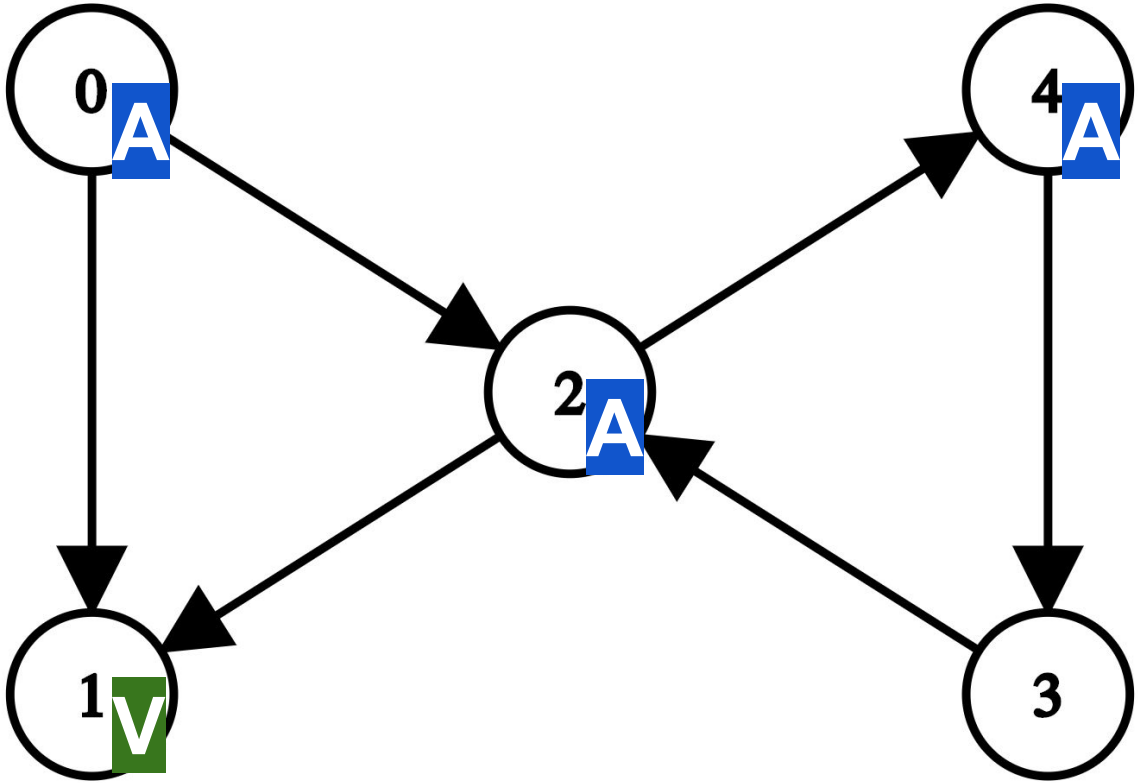

0
A
4
2
1
A
3

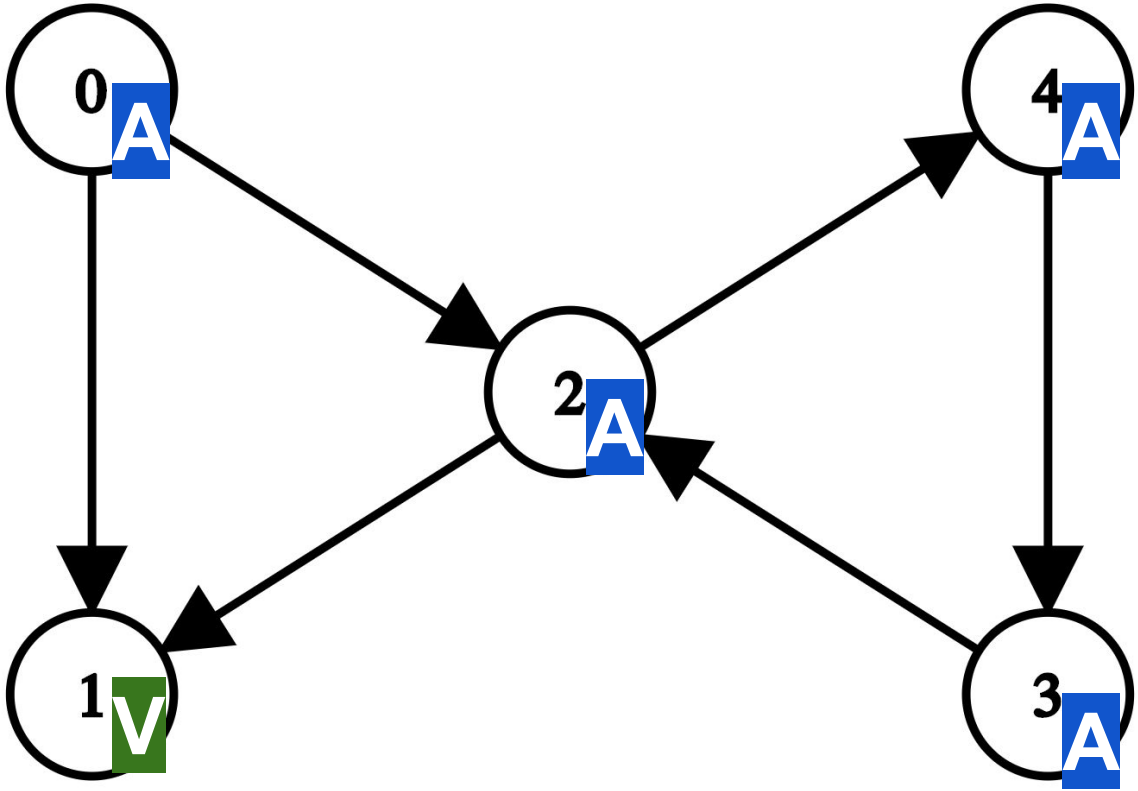

0
A
4
2
1
V
3

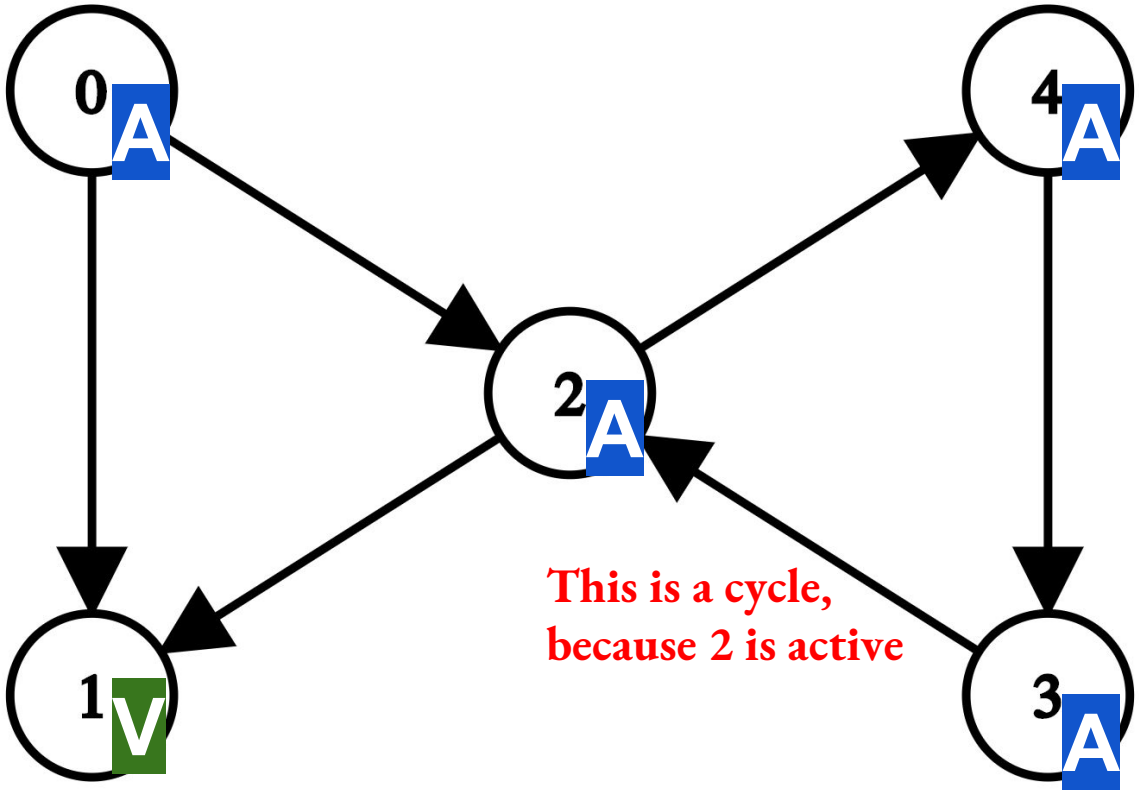

0
A
4
2
1
V
3

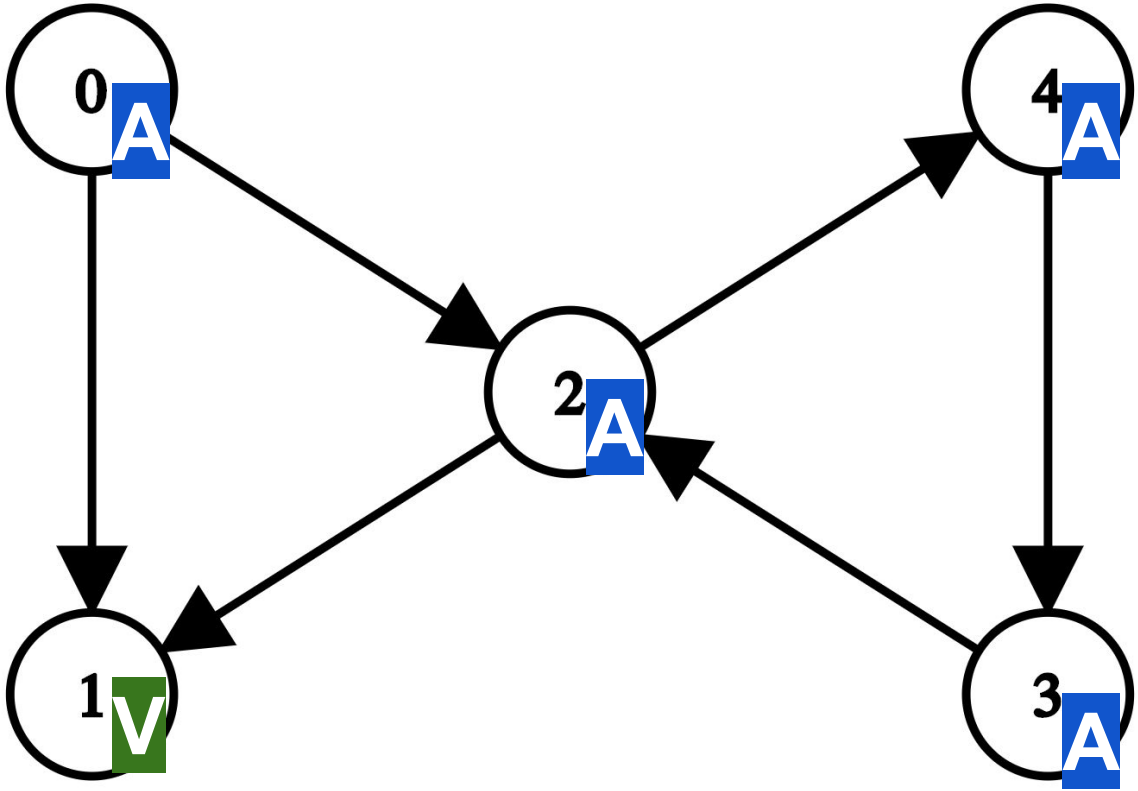

0
A
4
2
A
1
V
3

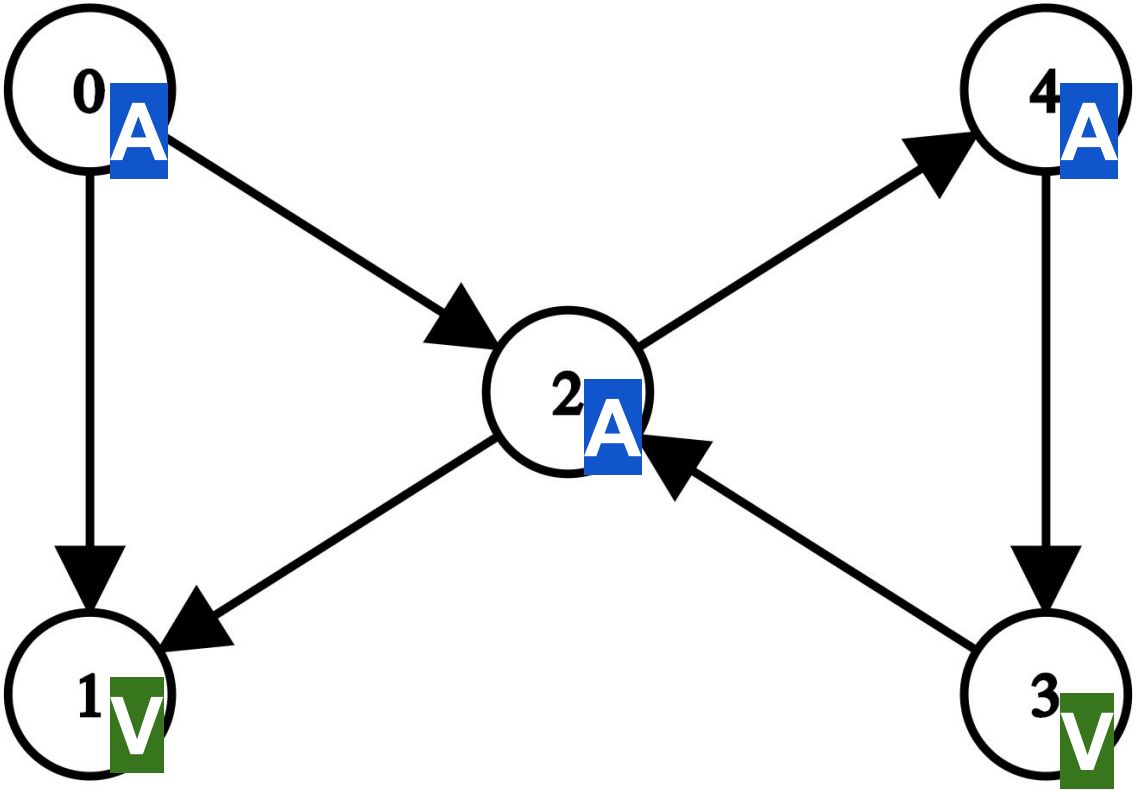

0
A
4
2
A
Not a cycle,
because 1 is visited
1
V
3

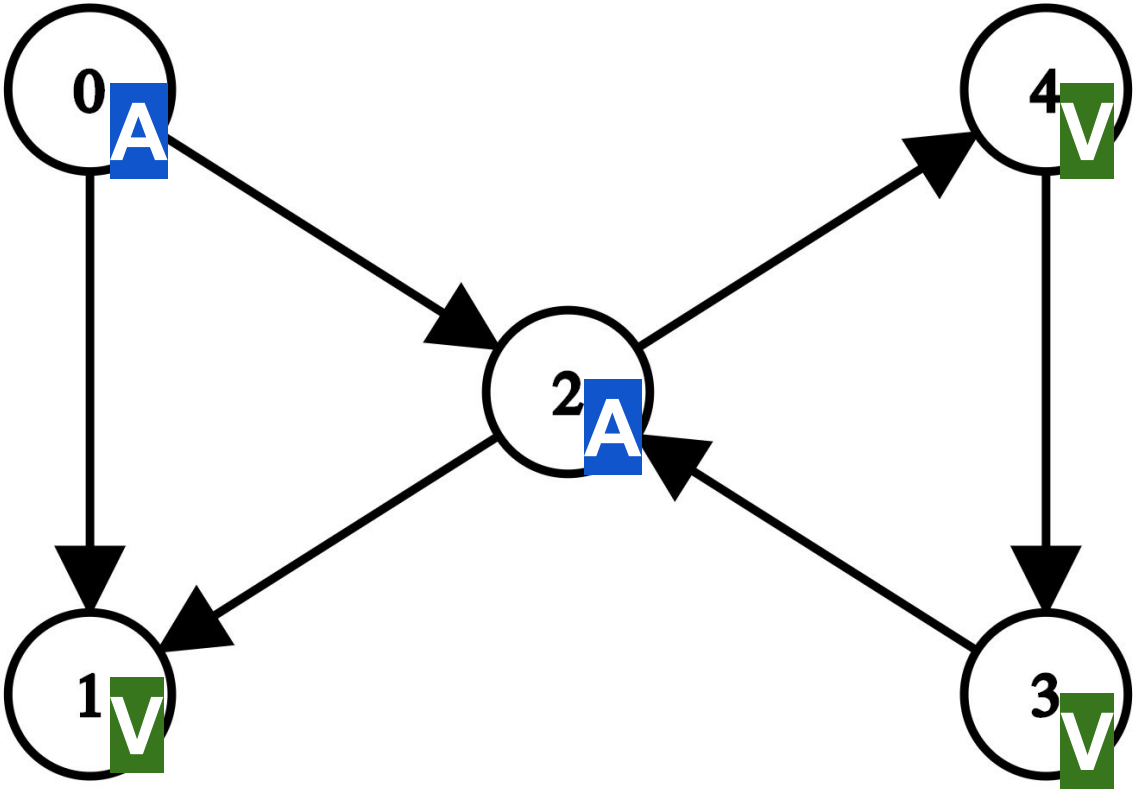

0
A
4
A
2
A
1
V
3

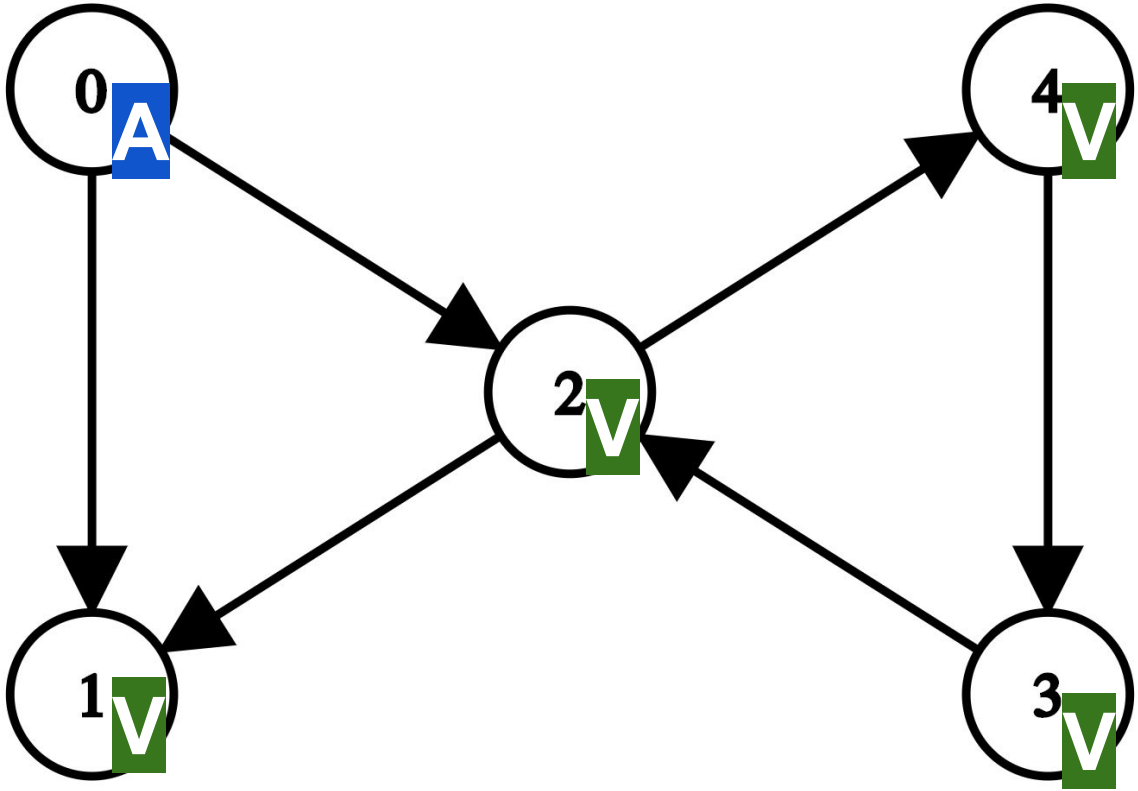

0
A
4
A
2
A
1
V
3
A

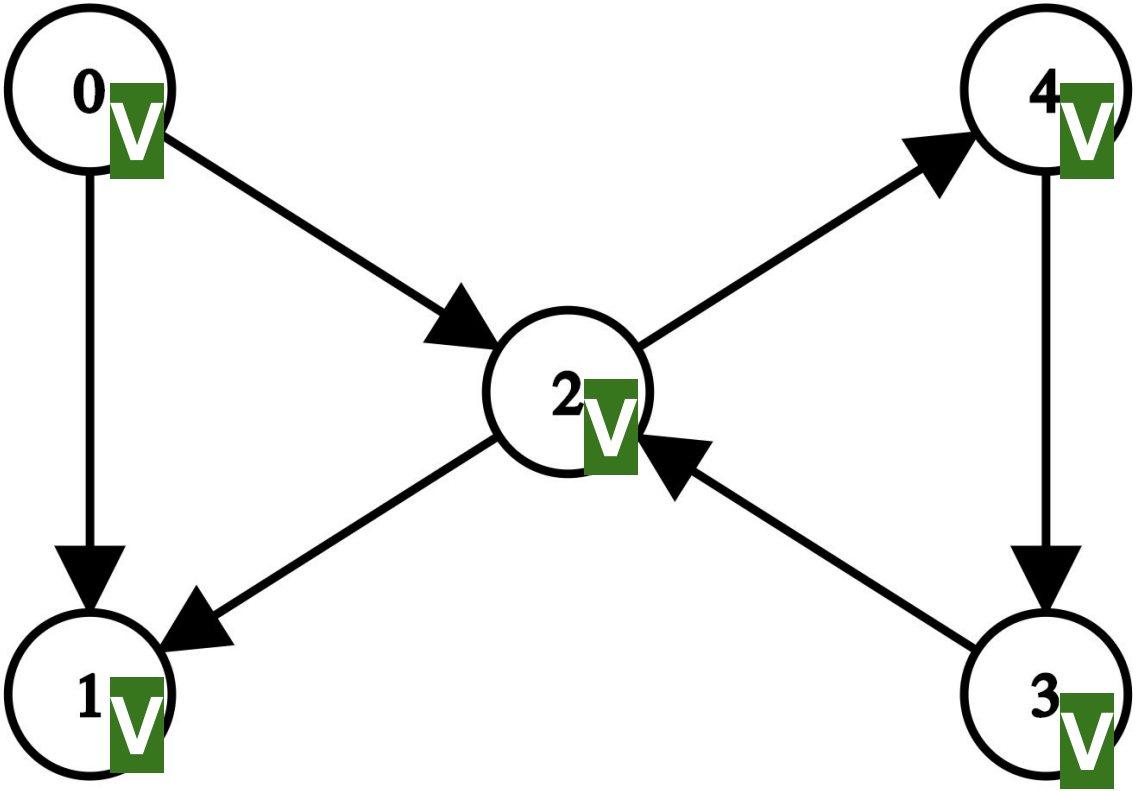

0
A
4
A
2
A
This is a cycle,
because 2 is active
1
V
3
A


0
A
4
A
2
A
1
V
3
A


0
A
4
A
2
A
1
V
3
V


0
A
4
V
2
A
1
V
3
V


0
A
4
V
2
V
1
V
3
V

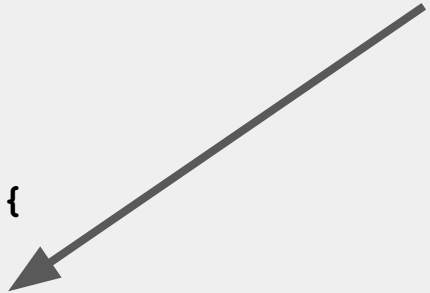

0
V
4
V
2
V
1
V
3
V

```
const int UNVISITED = 0;
const int ACTIVE = 1;
const int VISITED = 2;

void findcycle(int current) {
    cout << current << " ";
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size(); i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            findcycle(next);
        } else if (state[next] == ACTIVE) {
            cout << "BACKEDGE FOUND";
        }
    }
    state[current] = VISITED;
}
```

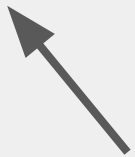
```
const int UNVISITED = 0;
const int ACTIVE = 1;
const int VISITED = 2;

void findcycle(int current) {
    cout << current << " ";
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size(); i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            findcycle(next);
        } else if (state[next] == ACTIVE) {
            cout << "BACKEDGE FOUND";
        }
    }
    state[current] = VISITED;
}
```



```
const int UNVISITED = 0;
const int ACTIVE = 1;
const int VISITED = 2;

void findcycle(int current) {
    cout << current << " ";
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size(); i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            findcycle(next);
        } else if (state[next] == ACTIVE) {
            cout << "BACKEDGE FOUND";
        }
    }
    state[current] = VISITED;
}
```



Traverse all neighbours

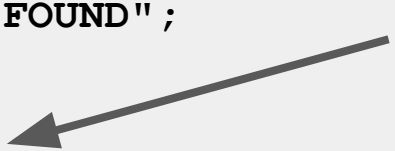
```
const int UNVISITED = 0;
const int ACTIVE = 1;
const int VISITED = 2;

void findcycle(int current) {
    cout << current << " ";
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size(); i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            findcycle(next);
        } else if (state[next] == ACTIVE) {
            cout << "BACKEDGE FOUND";
        }
    }
    state[current] = VISITED;
}
```

if we reach an active vertex, then we have found a cycle

```
const int UNVISITED = 0;
const int ACTIVE = 1;
const int VISITED = 2;

void findcycle(int current) {
    cout << current << " ";
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size(); i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            findcycle(next);
        } else if (state[next] == ACTIVE) {
            cout << "BACKEDGE FOUND";
        }
    }
    state[current] = VISITED;
}
```



Topological Sorting

Topological Sorting

Consider a directed acyclic graph

An arc from **w** to **v** means that **v** < **w**

Topological Sorting: If **v** < **w**, then **v** has to appear earlier than **w** in list

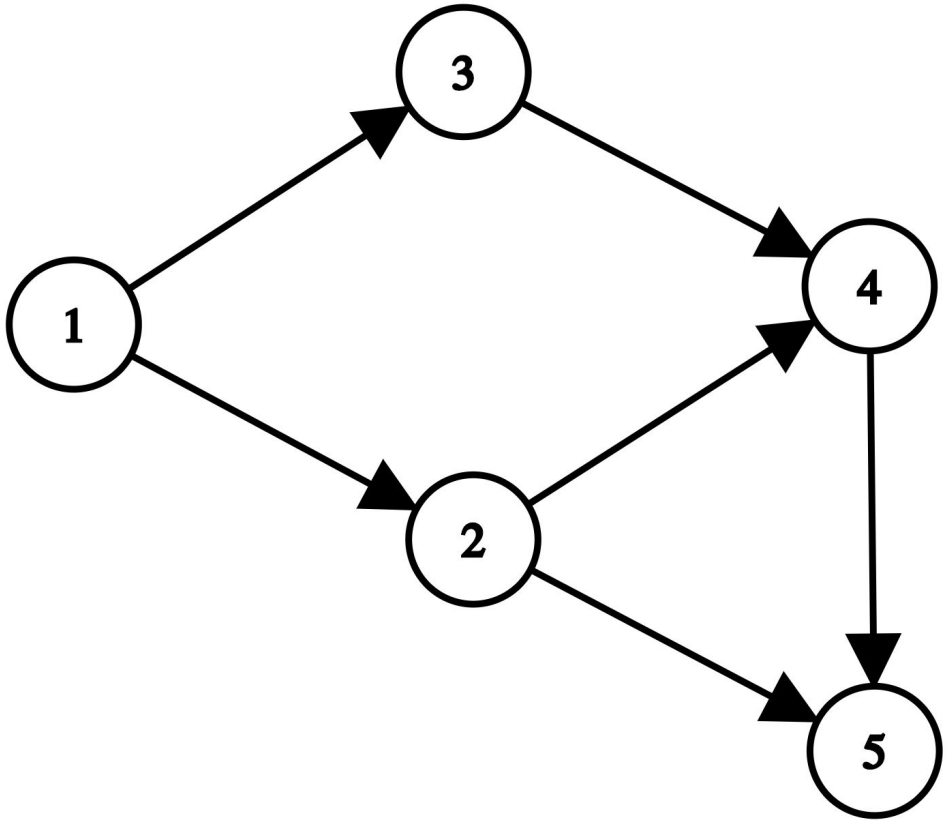
Topological Sorting

$5 < 4 < 3 < 2 < 1$

$5 < 4 < 2 < 3 < 1$

but not

$5 < 2 < 4 < 3 < 1$



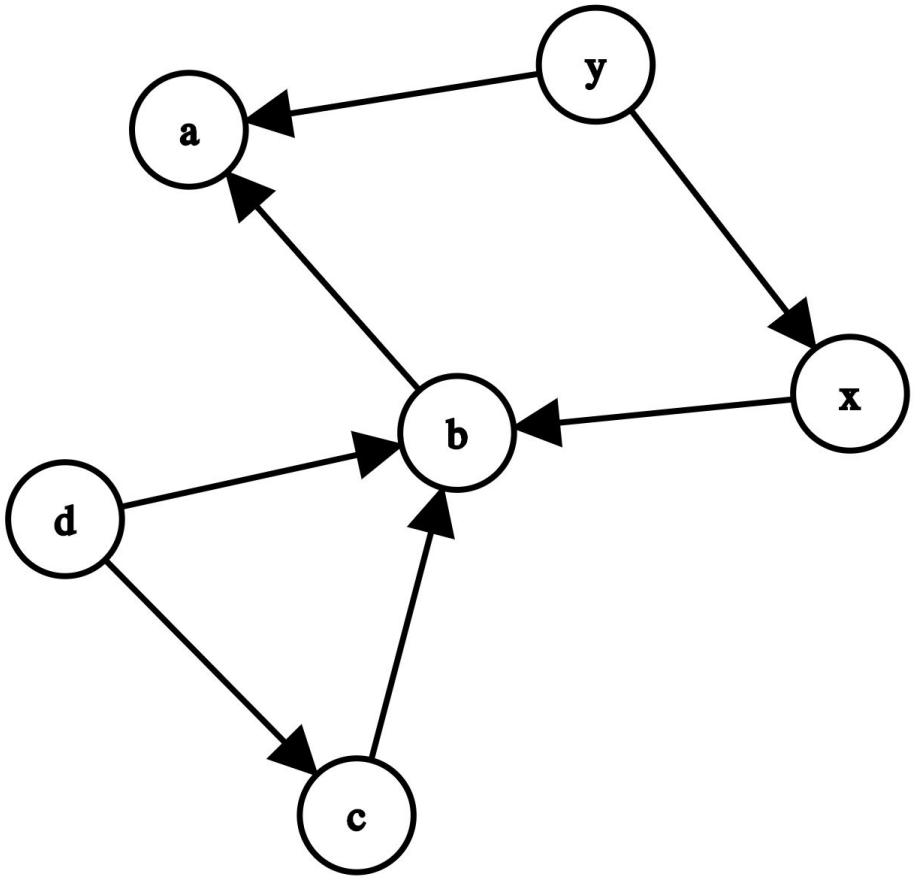
Topological Sorting

a < b < x < y < c < d

a < b < c < x < d < y

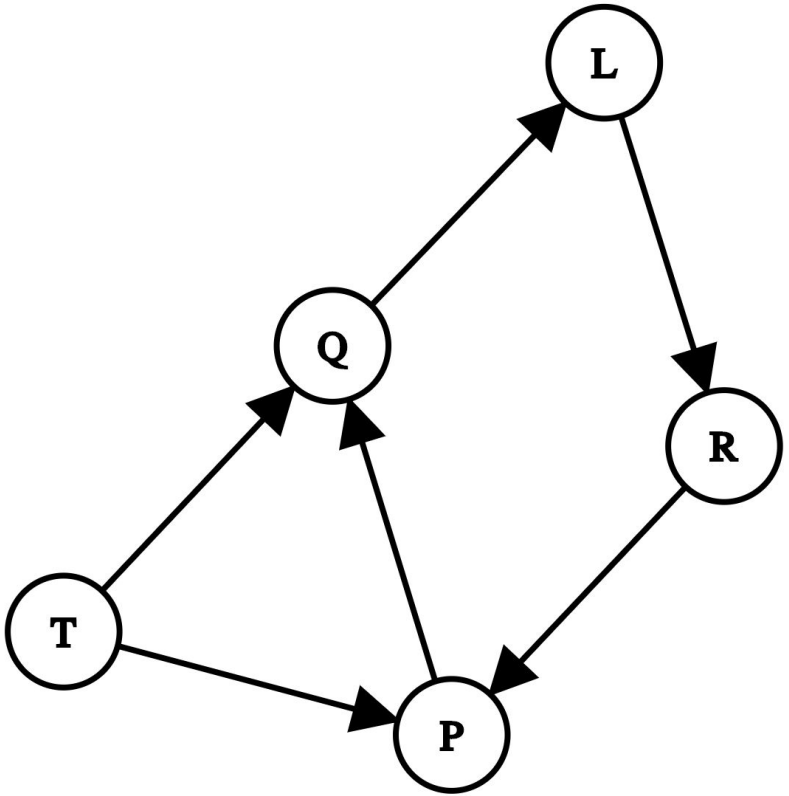
a < b < x < c < y < d

...



Topological Sorting

None, the graph is not acyclic (P, Q, L, R)



Topological Sorting

Idea for Algorithm

After recursive DFS all smaller vertices have been visited

Then push `current` to back of stack

```
void toposort(int current) {
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size; i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            toposort(next);
        } else if (state[next] == ACTIVE) {
            cout << "CYCLIC DEPENDENCY";
        }
    }
    state[current] = VISITED;
    topo.push(current);
}
```

```
void toposort(int current) {
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size; i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            toposort(next);
        } else if (state[next] == ACTIVE) {
            cout << "CYCLIC DEPENDENCY";
        }
    }
    state[current] = VISITED;
    topo.push(current);
}
```

Similar to finding cycles

```
void toposort(int current) {
    state[current] = ACTIVE;
    for (int i = 0; i < graph[current].size; i++) {
        int next = graph[current][i];
        if (state[next] == UNVISITED) {
            toposort(next);
        } else if (state[next] == ACTIVE) {
            cout << "CYCLIC DEPENDENCY";
        }
    }
    state[current] = VISITED;
    topo.push(current);
}
```

Push to result after all smaller vertices were visited

Breadth-First-Search

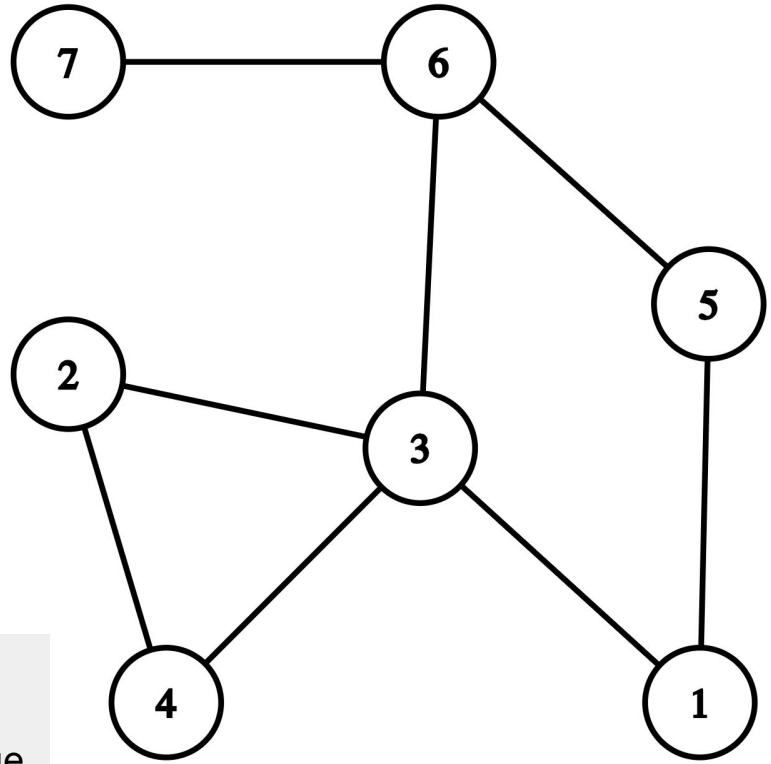
Breadth-First-Search

Vertices are visited in order of discovery

Instead of using a stack we use a queue

BFS

>



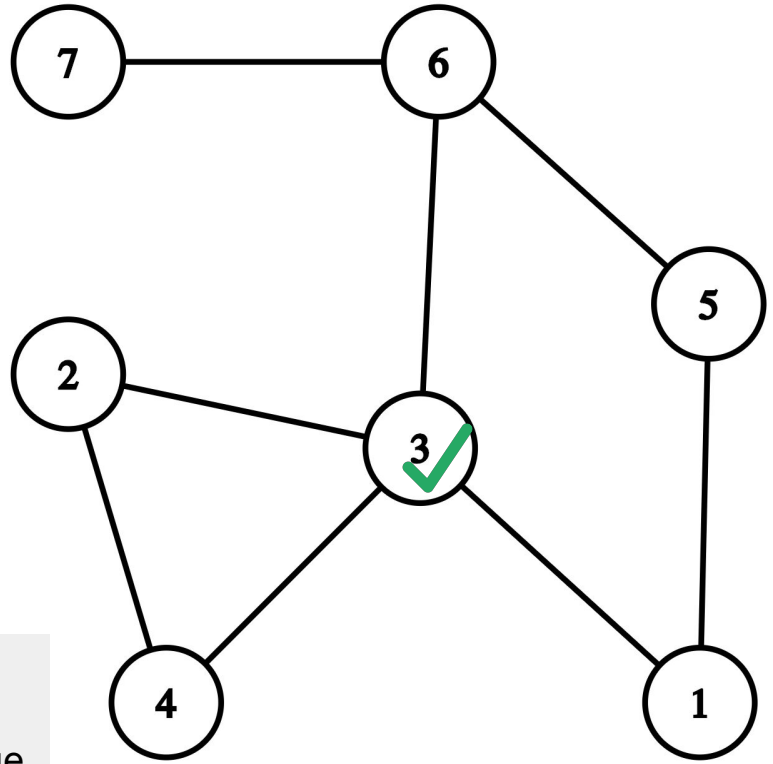
3

Queue

BFS

>3

CurrentNode=**3**



Queue

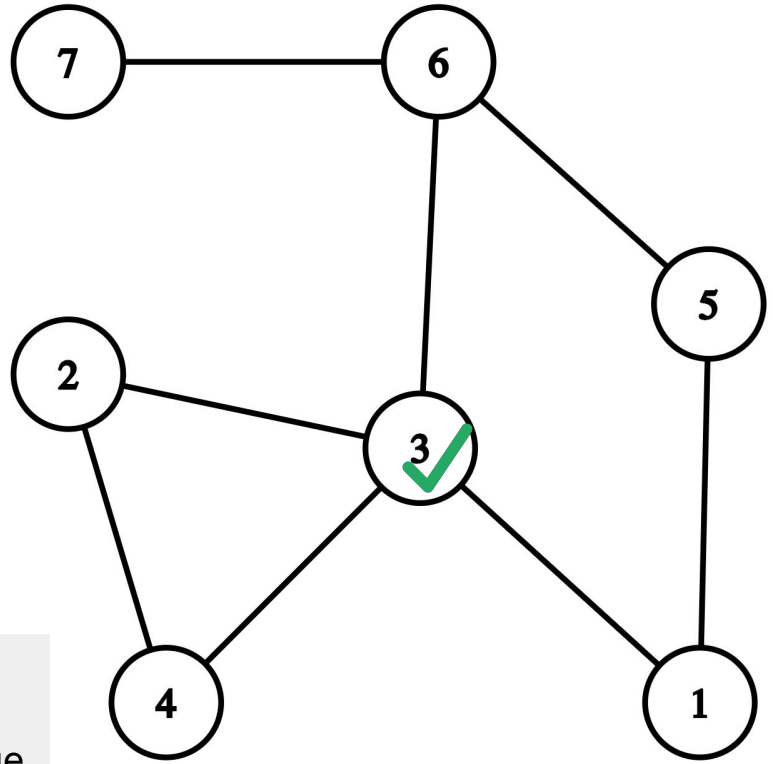
BFS

>3

CurrentNode=**3**

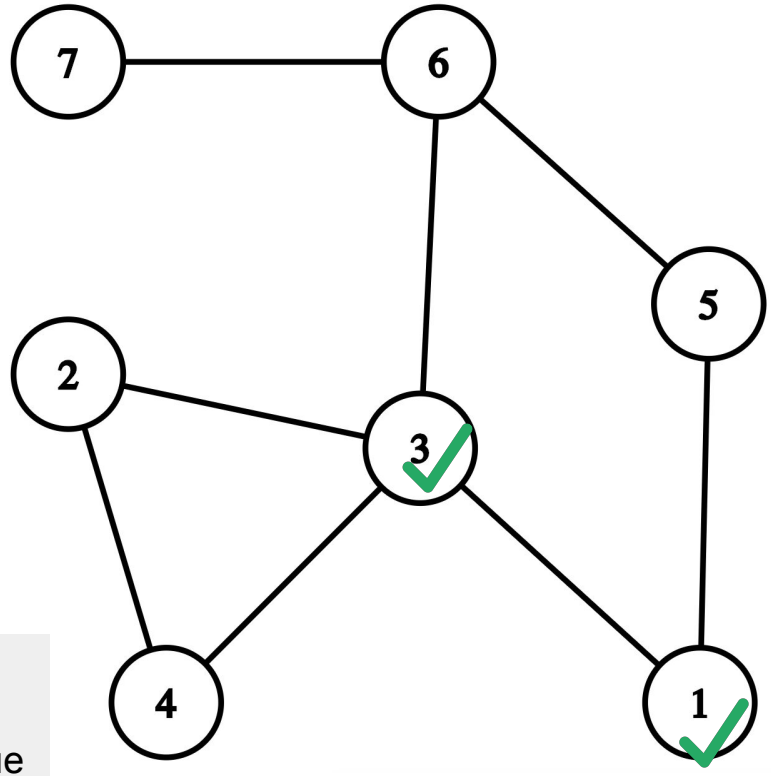
1 2 4 6

Queue



BFS

```
>3 1
```



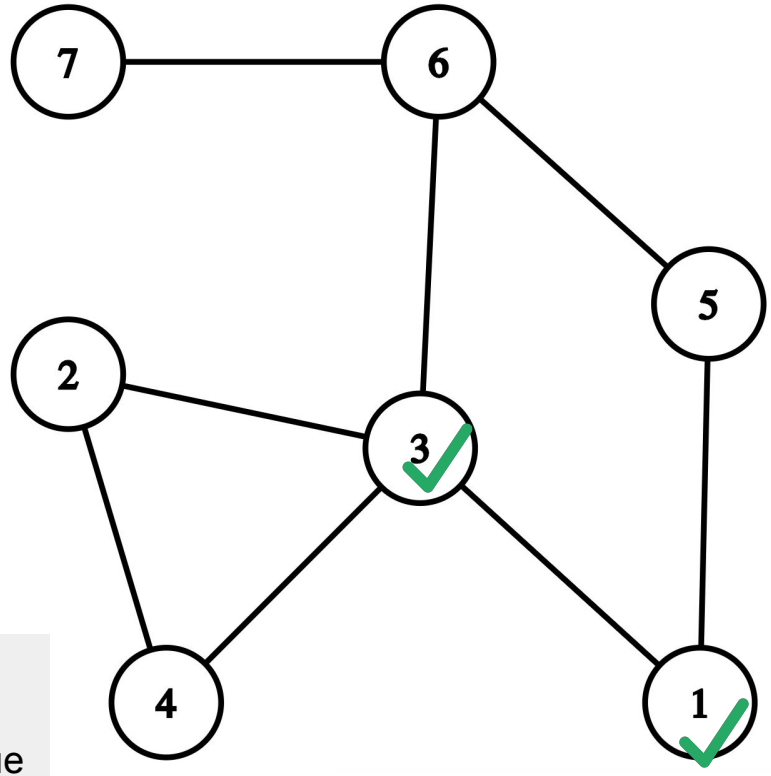
`CurrentNode=`**1**

2 4 6

Queue

BFS

```
>3 1
```




```
CurrentNode=1
```

2 4 6 5

Queue

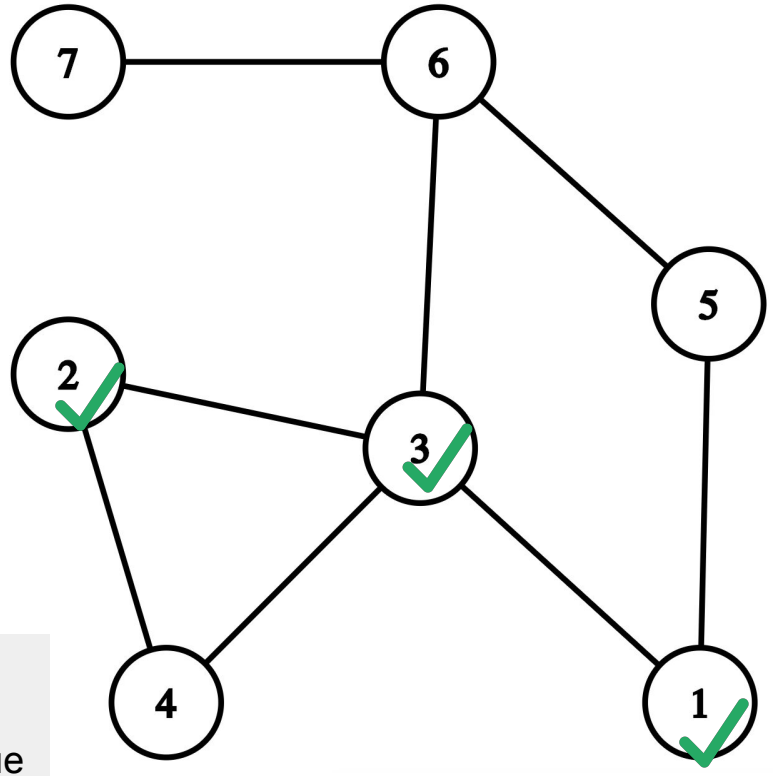
BFS

```
>3 1 2
```

```
CurrentNode=2
```

4 6 5

Queue



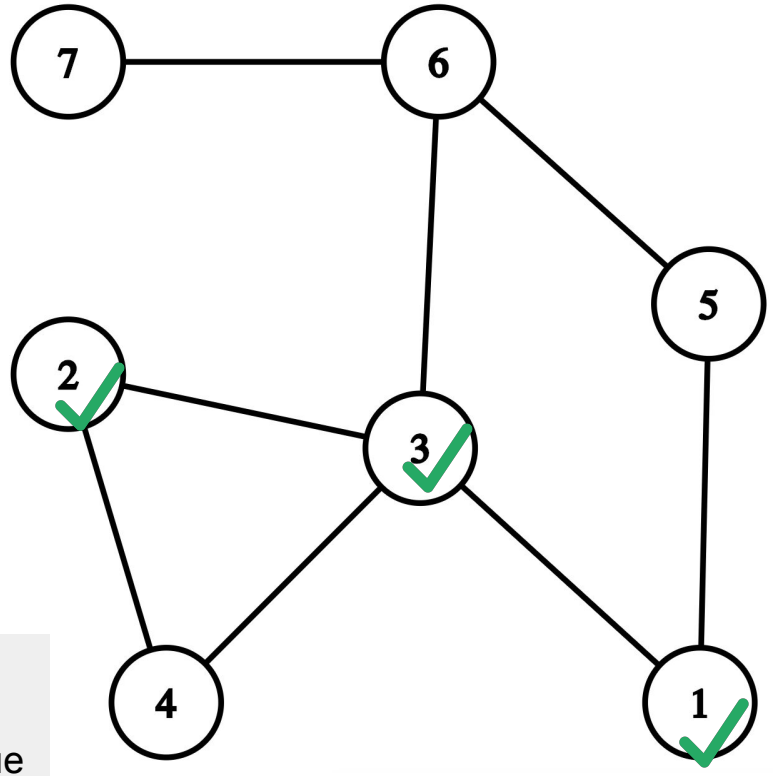
BFS

```
>3 1 2
```

```
CurrentNode=2
```

4 6 5 4

Queue



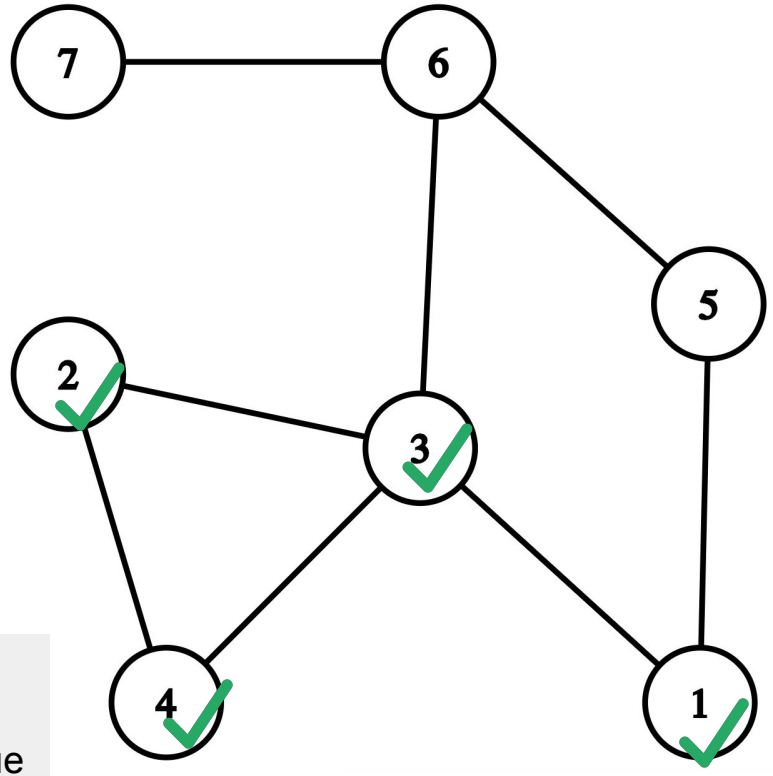
BFS

```
>3 1 2 4
```

CurrentNode=**4**

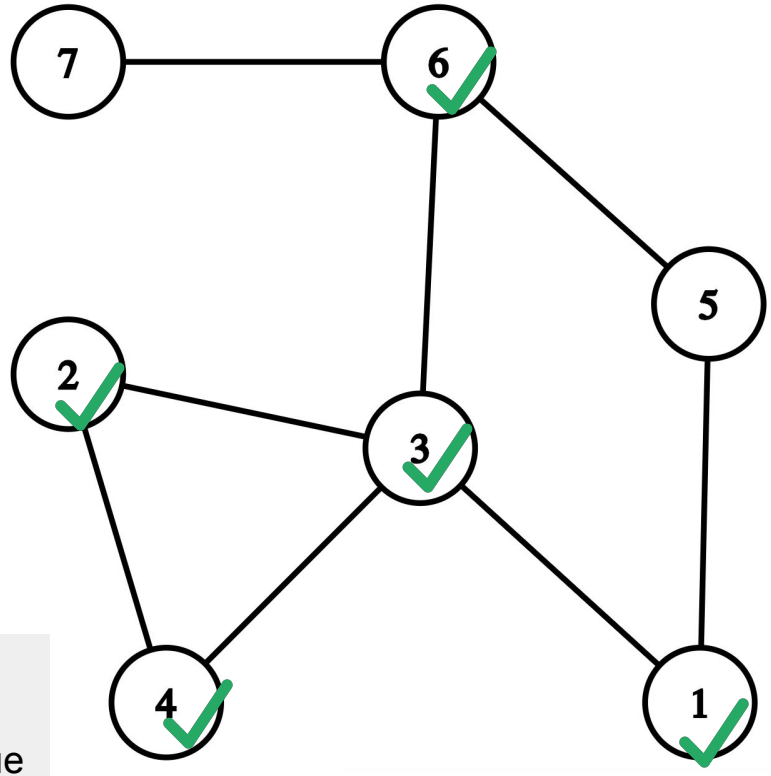
6 5 4

Queue



BFS

```
>3 1 2 4 6
```



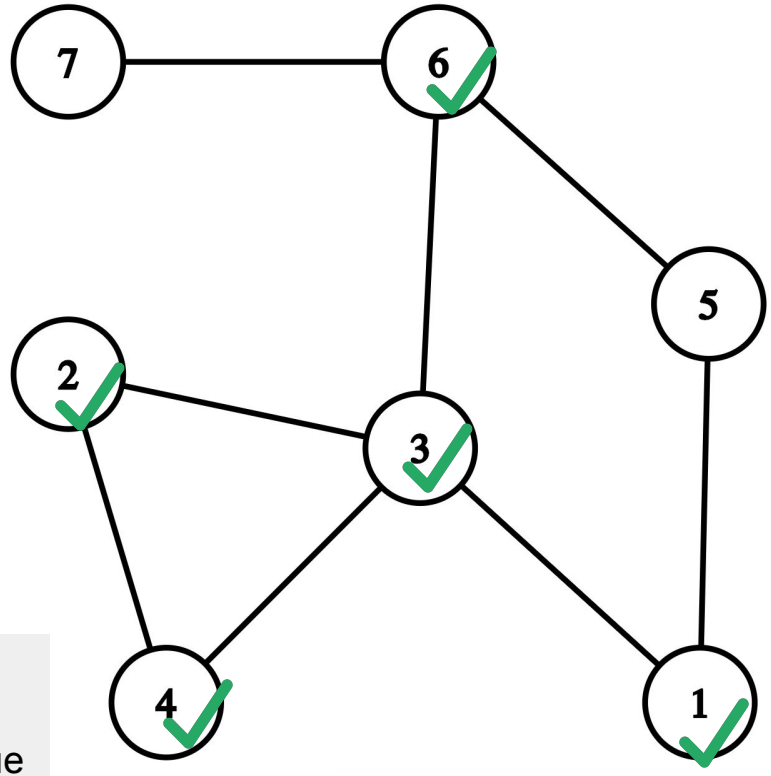

```
CurrentNode=6
```

5 4

Queue

BFS

```
>3 1 2 4 6
```



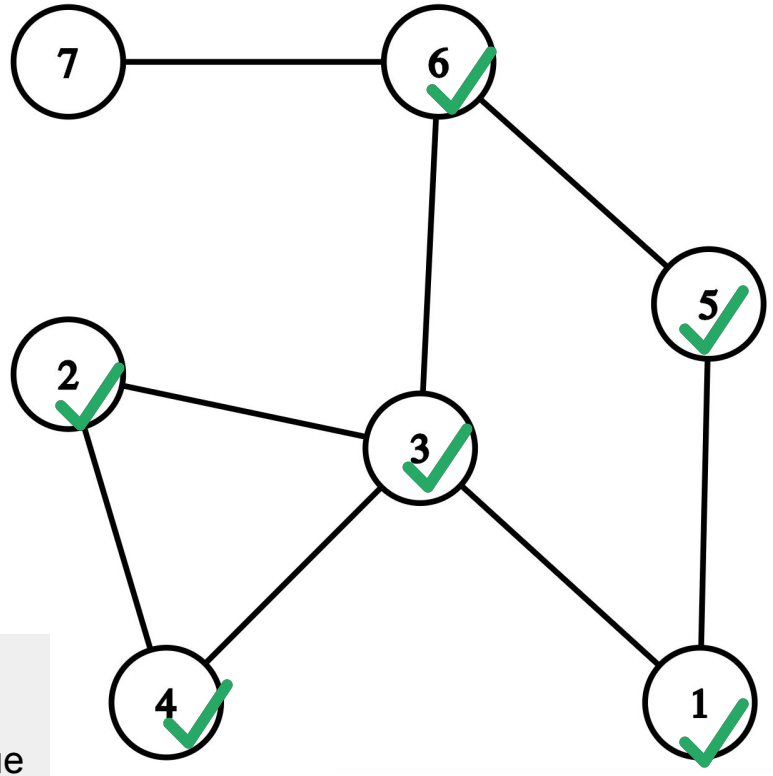

```
CurrentNode=6
```

5 4 5 7

Queue

BFS

```
>3 1 2 4 6
```




```
CurrentNode=5
```

4 5 7

Queue

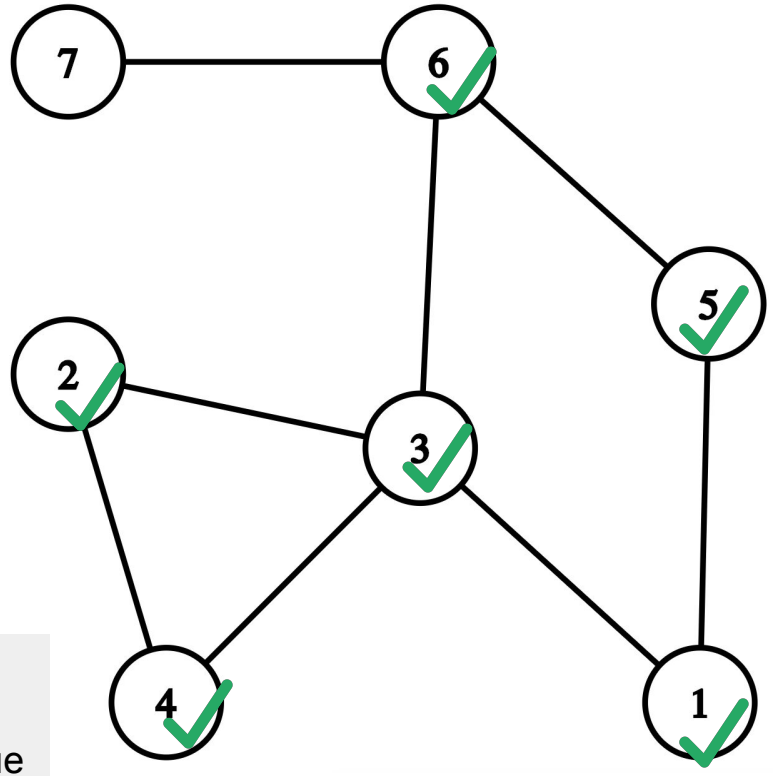
BFS

```
>3 1 2 4 6
```

CurrentNode=**4**

5 7

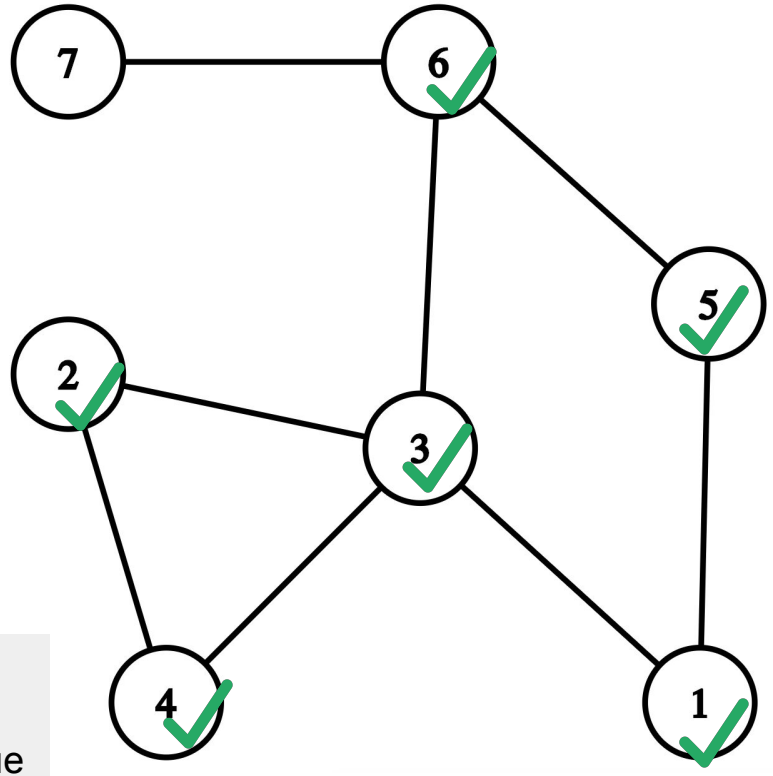
Queue



BFS

```
>3 1 2 4 6
```

```
CurrentNode=5
```



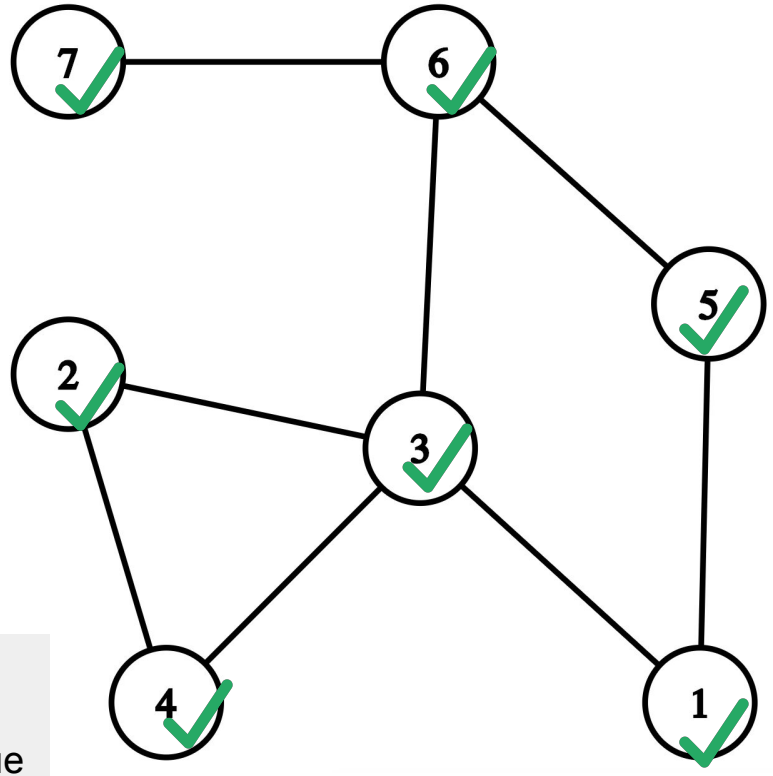
7

Queue

BFS

```
>3 1 2 4 6 7
```

```
CurrentNode=7
```



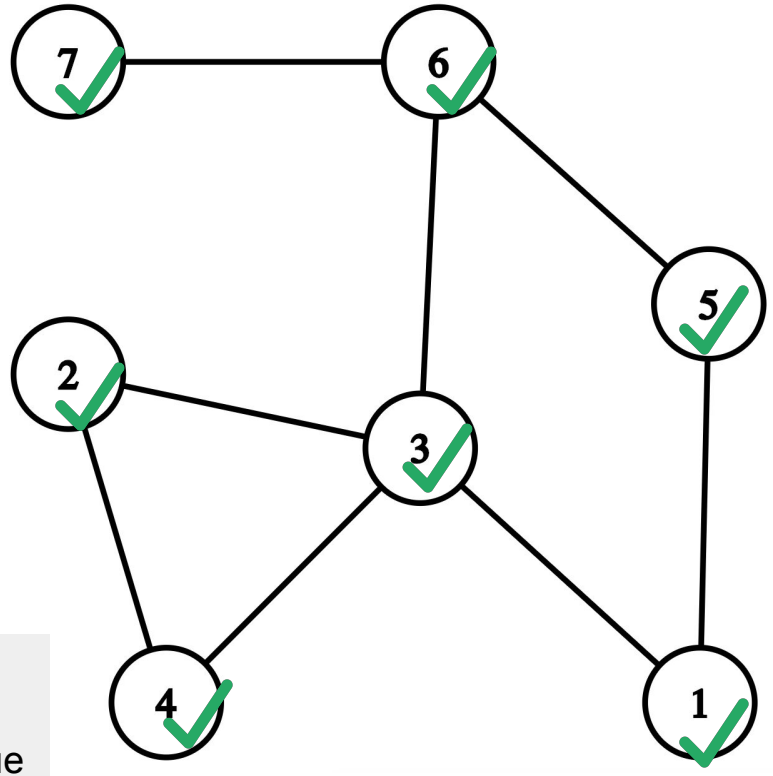
Queue

BFS

```
>3 1 2 4 6 7
```

```
Finished
```

Queue



```
void bfs(int root) {
    queue<int> q;
    q.push(root);
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        if (!visited[current]) {
            cout << current << " ";
            visited[current] = true;
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                q.push(next);
            }
        }
    }
}
```

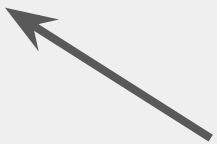
```
void bfs(int root) {
    queue<int> q;
    q.push(root);
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        if (!visited[current]) {
            cout << current << " ";
            visited[current] = true;
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                q.push(next);
            }
        }
    }
}
```

Pop as long as there are elements in the queue

```
void bfs(int root) {
    queue<int> q;
    q.push(root);
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        if (!visited[current]) {
            cout << current << " ";
            visited[current] = true;
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                q.push(next);
            }
        }
    }
}
```

Only proceed, if not visited

```
void bfs(int root) {
    queue<int> q;
    q.push(root);
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        if (!visited[current]) {
            cout << current << " ";
            visited[current] = true;
            for (int i = 0; i < graph[current].size(); i++) {
                int next = graph[current][i];
                q.push(next);
            }
        }
    }
}
```



Push adjacent vertices

Summary

What is a graph

Depth-First-Search

Finding Cycles

Topological Sorting

Breadth-First-Search

Dijkstra

Dijkstra's Algorithm

Given a weighted graph, find the shortest path from vertex A to B

Dijkstra's Algorithm

For each vertex v set **`dist[v]`** = ∞ and **`known[v] = false`**

Set **`dist[A] = 0`**

While there are unknown vertices in the graph

    Select unknown vertex p with lowest **`dist[p]`**

    Set **`known[p] = true`**

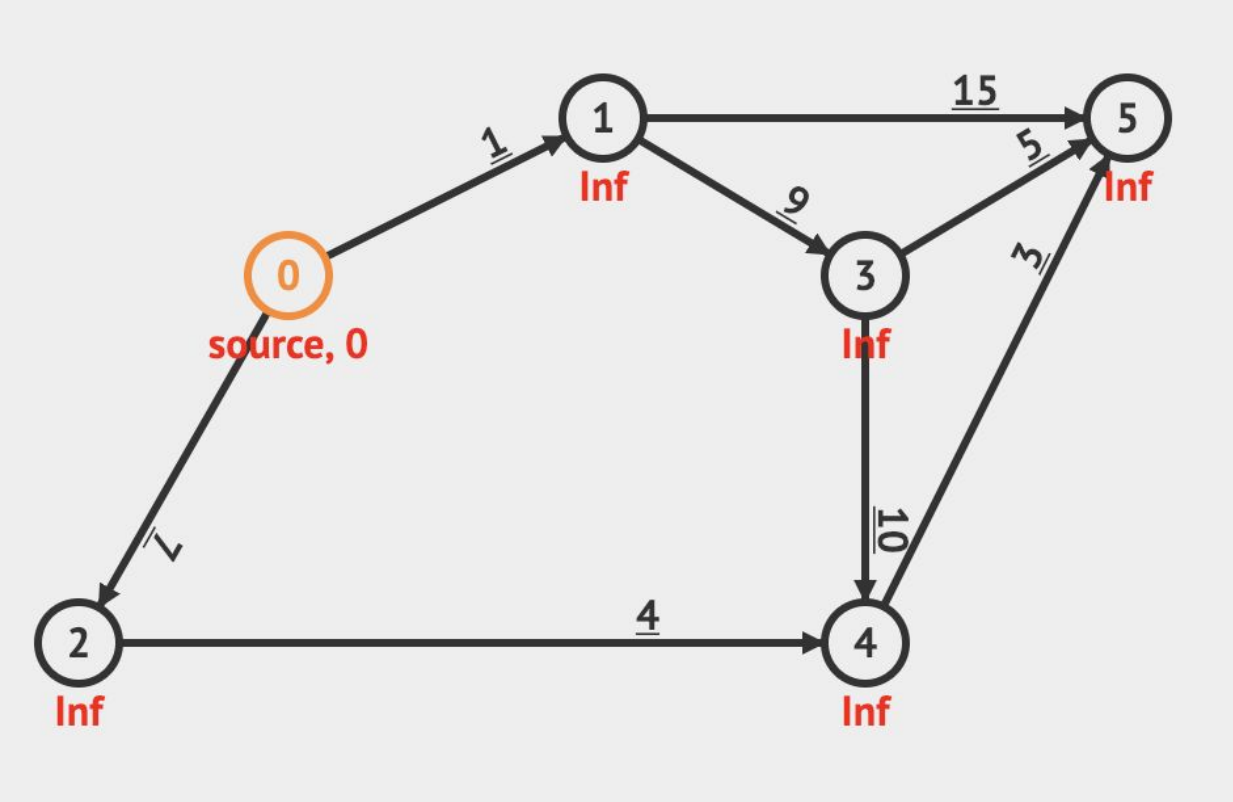
    For each edge (p, q) with weight w

        if **`dist[p] + w < dist[q]`**

            `dist[q] = dist[p] + w`

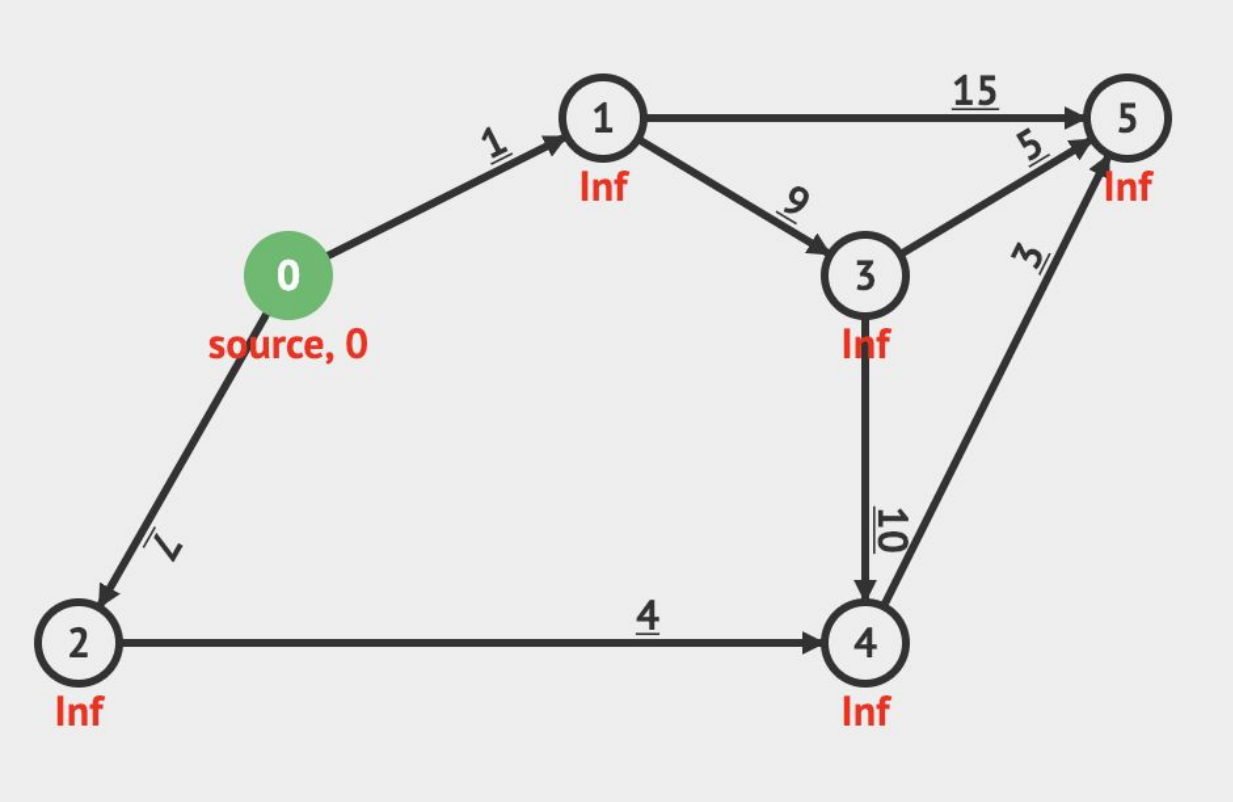
            `path[q] = w`

Dijkstra's Algorithm



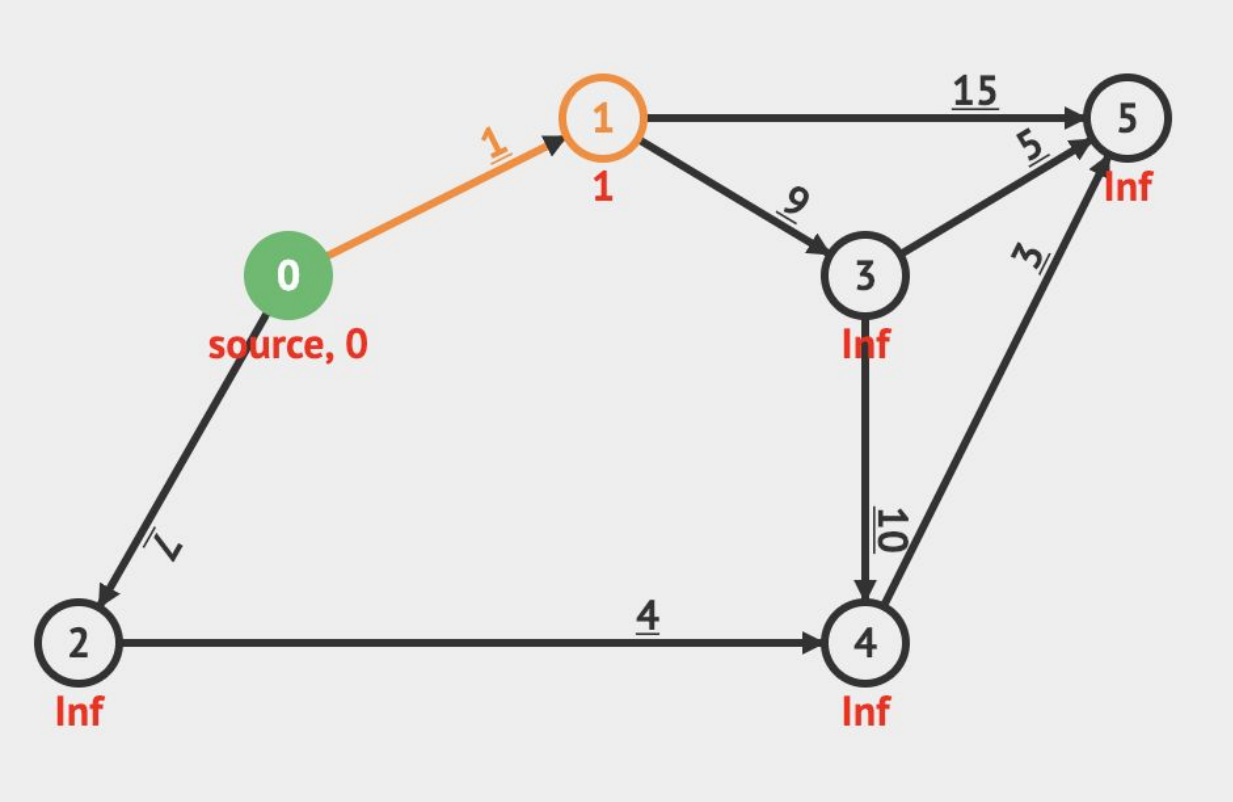
source: visualgo.net

Dijkstra's Algorithm



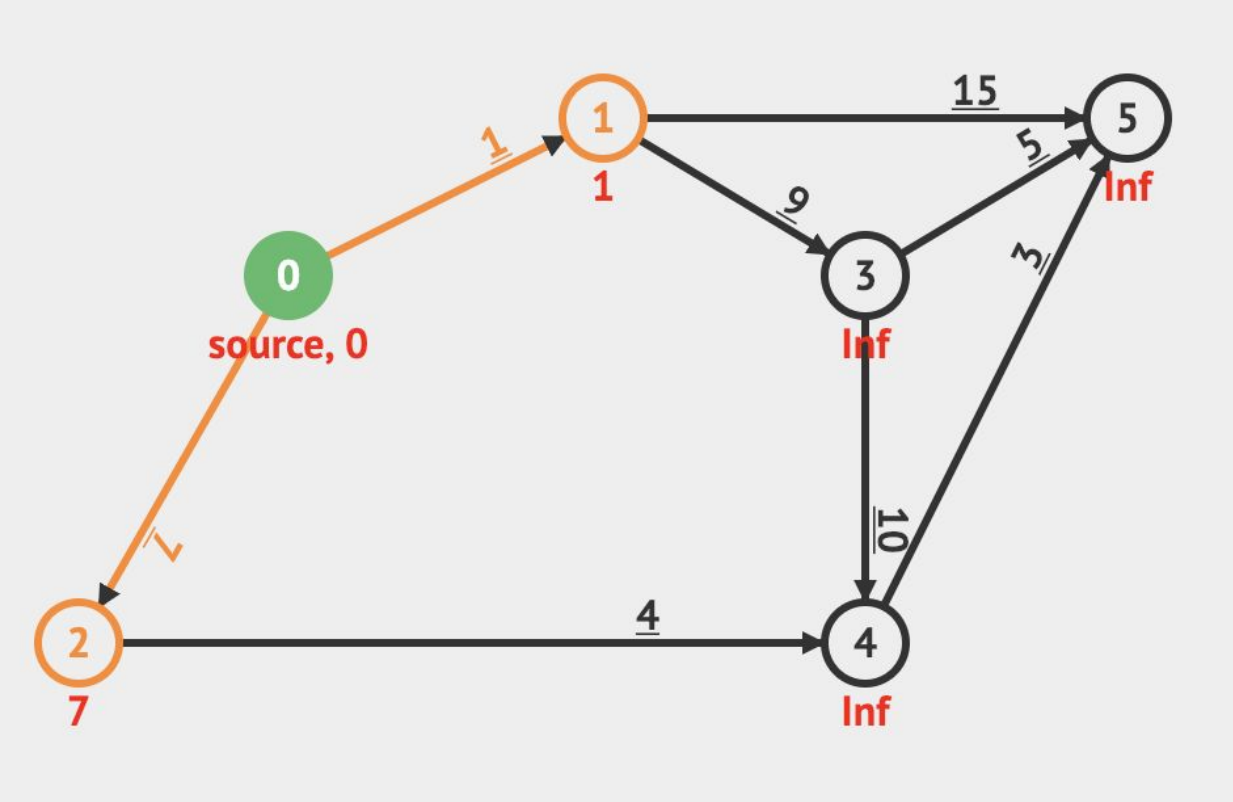
source: visualgo.net

Dijkstra's Algorithm



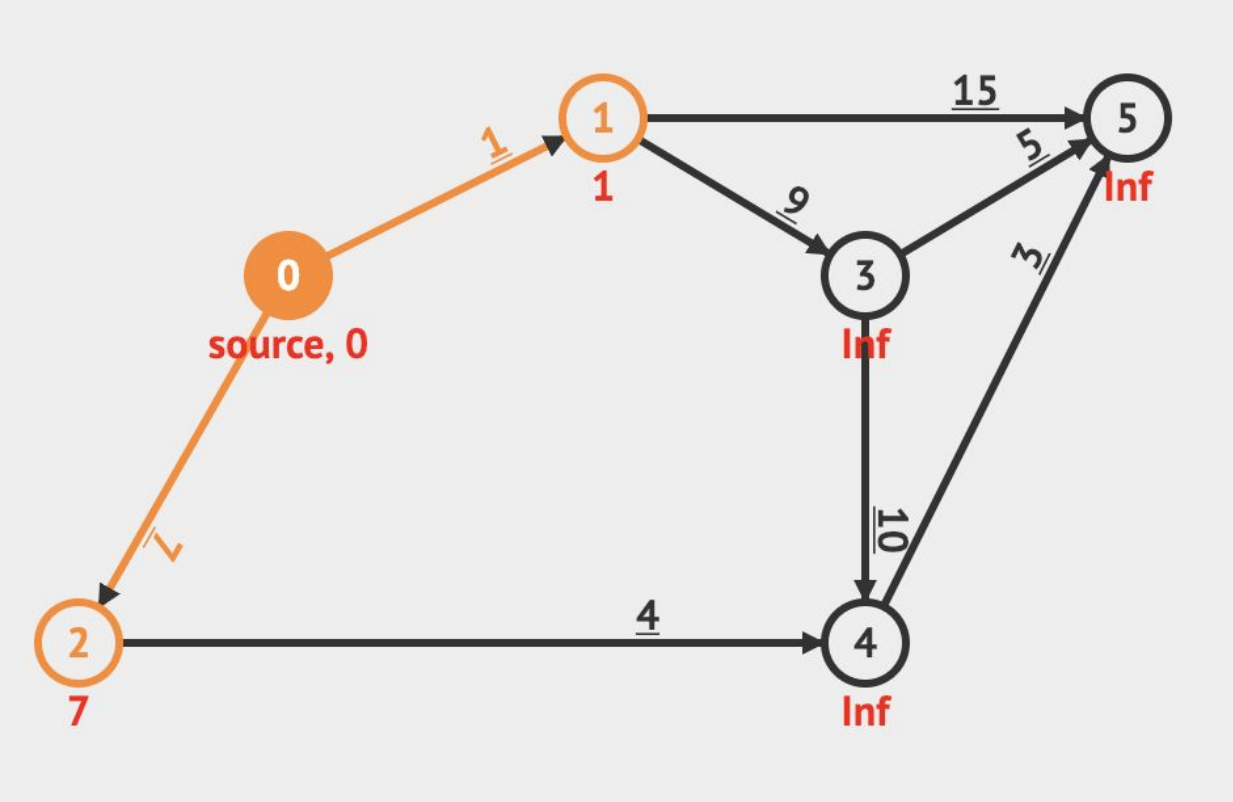
source: visualgo.net

Dijkstra's Algorithm



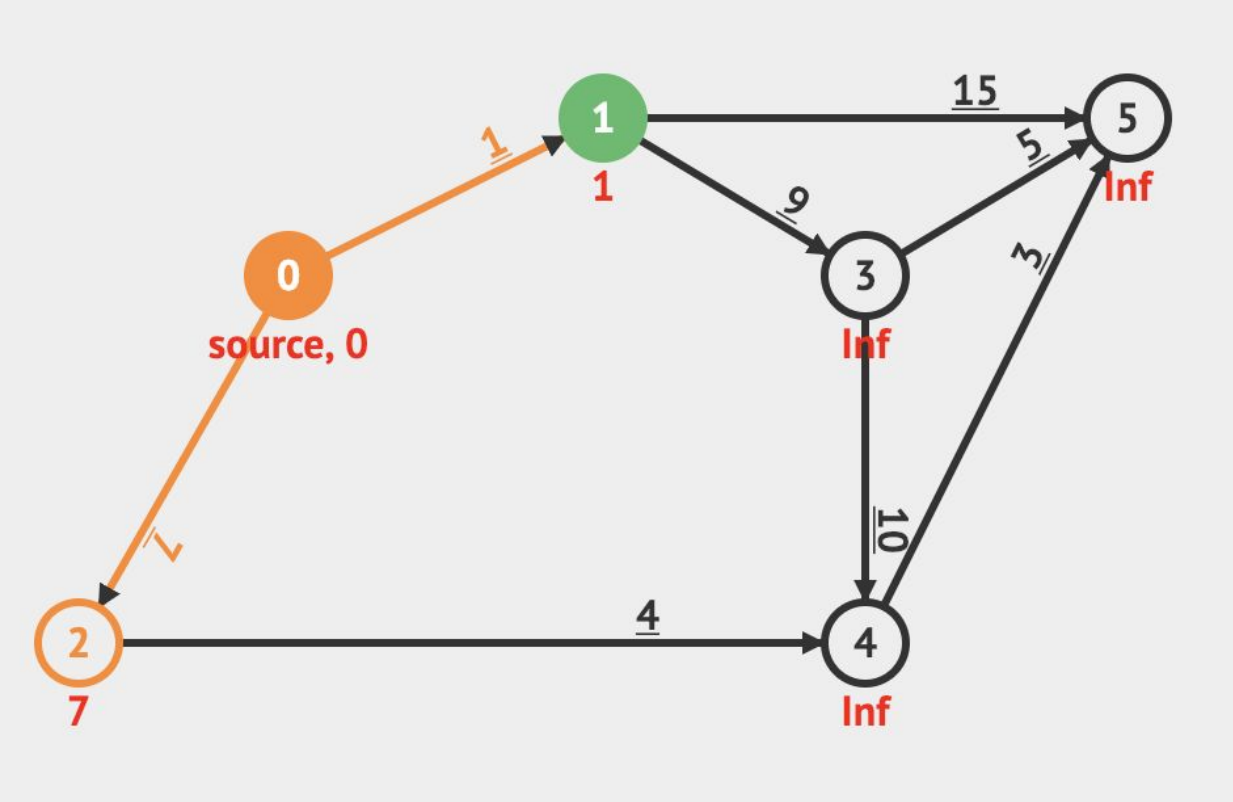
source: visualgo.net

Dijkstra's Algorithm



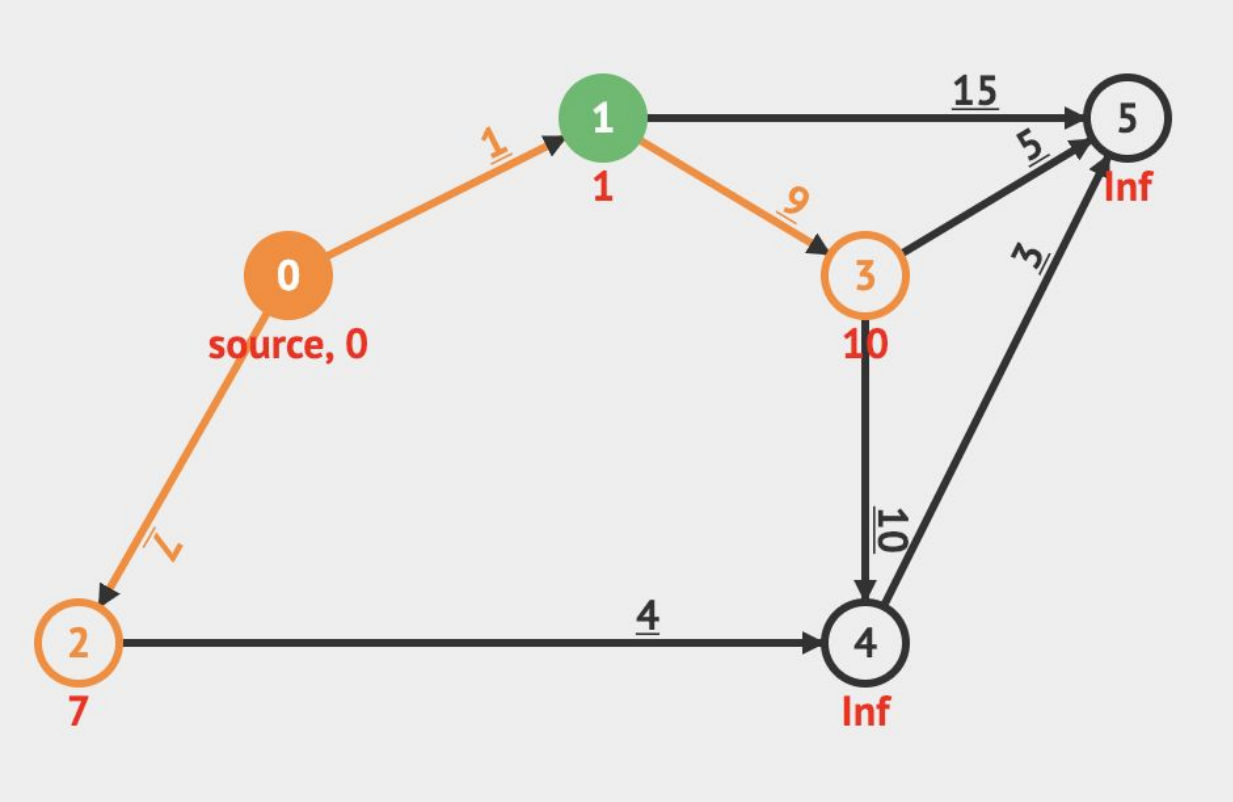
source: visualgo.net

Dijkstra's Algorithm



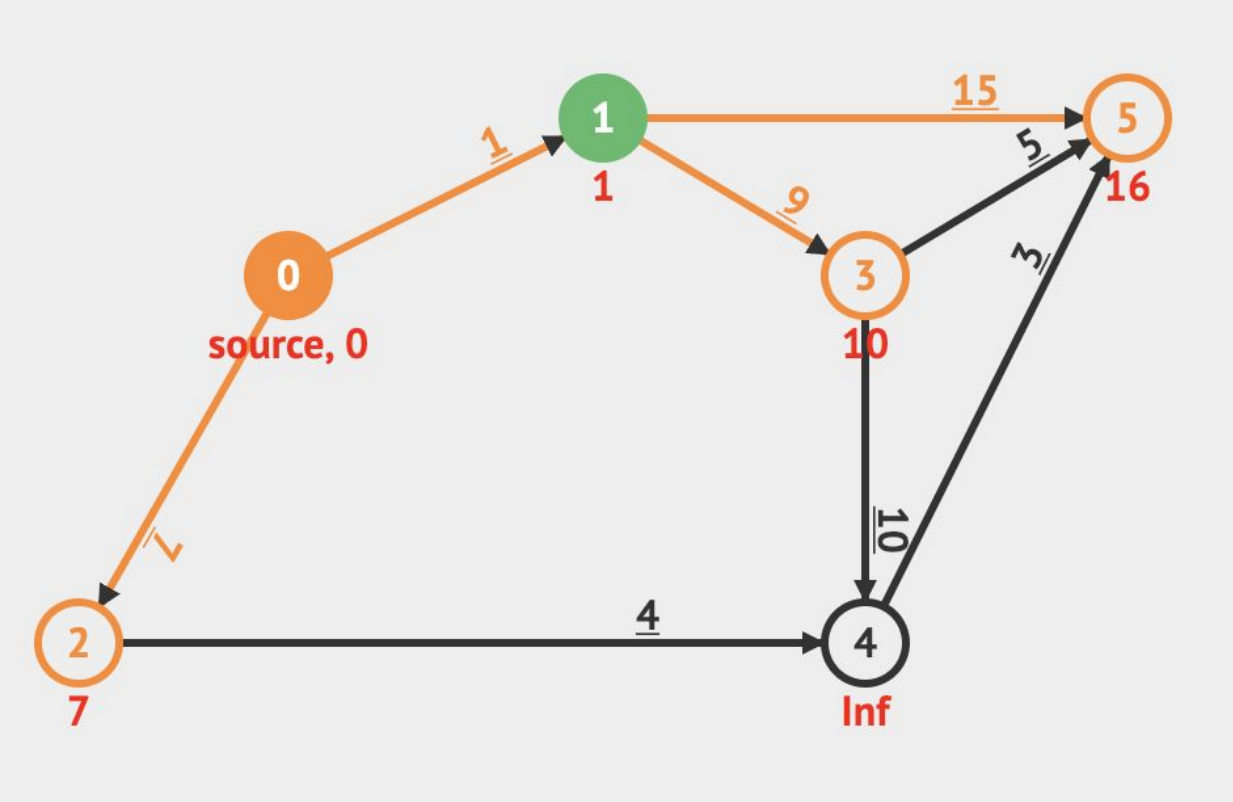
source: visualgo.net

Dijkstra's Algorithm



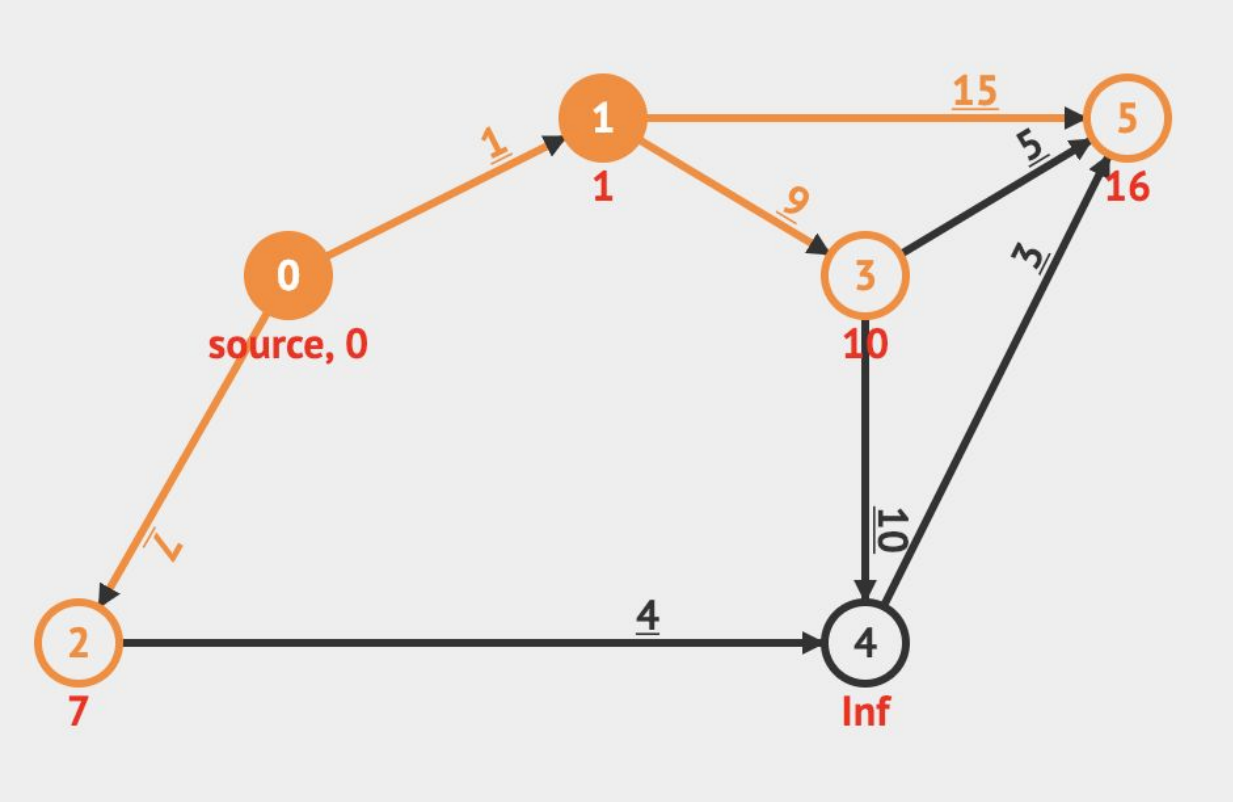
source: visualgo.net

Dijkstra's Algorithm



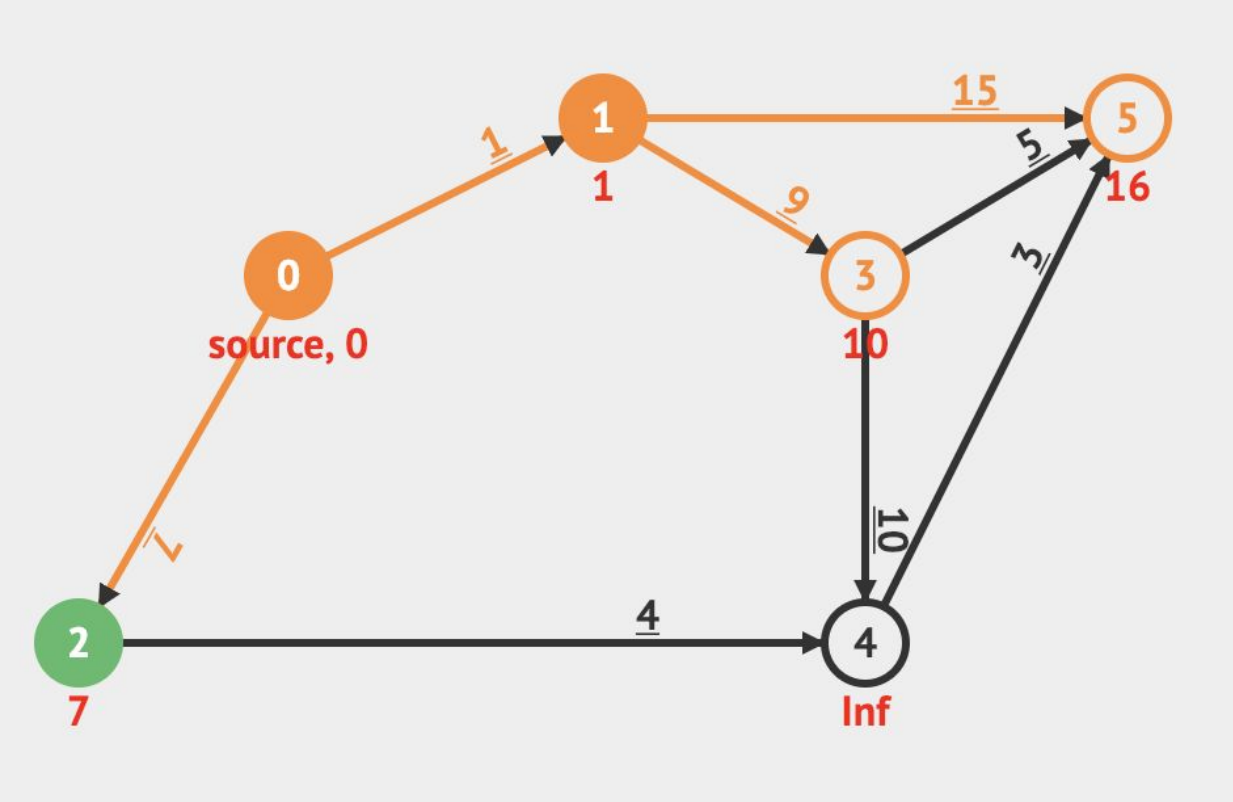
source: visualgo.net

Dijkstra's Algorithm



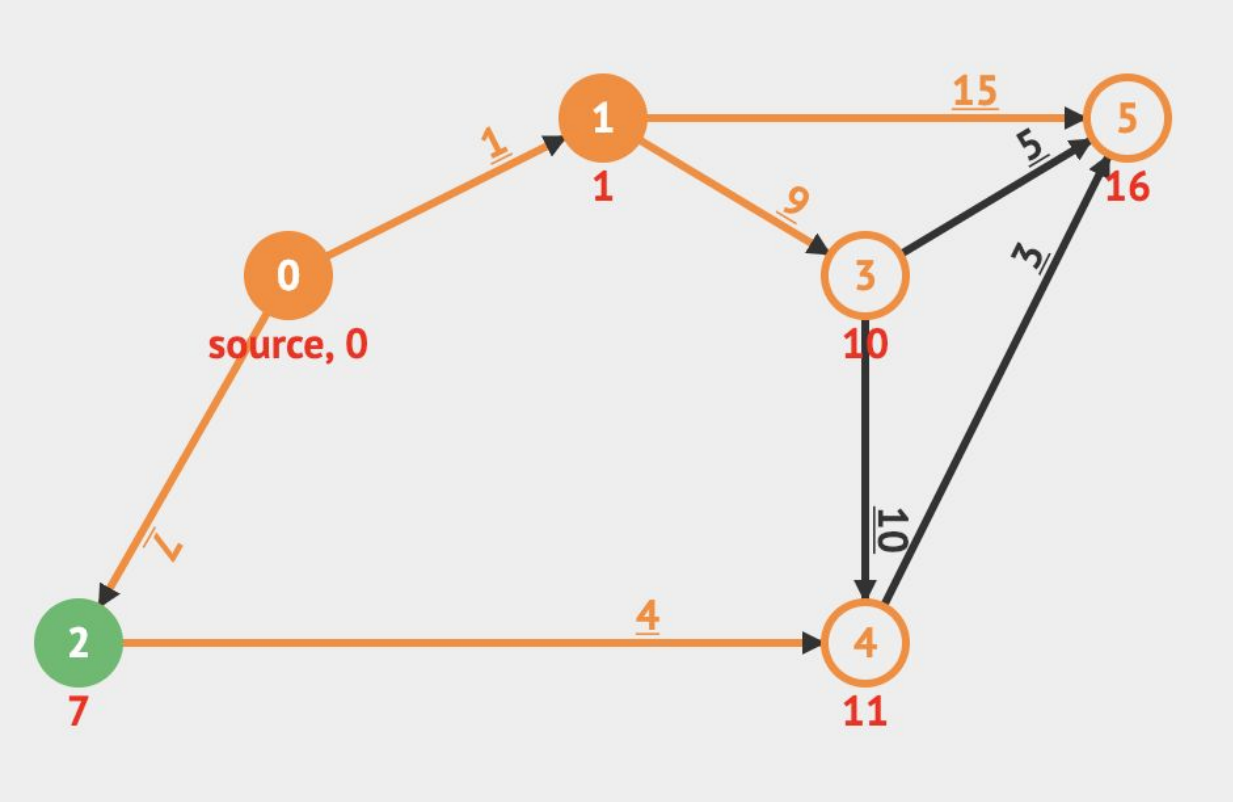
source: visualgo.net

Dijkstra's Algorithm



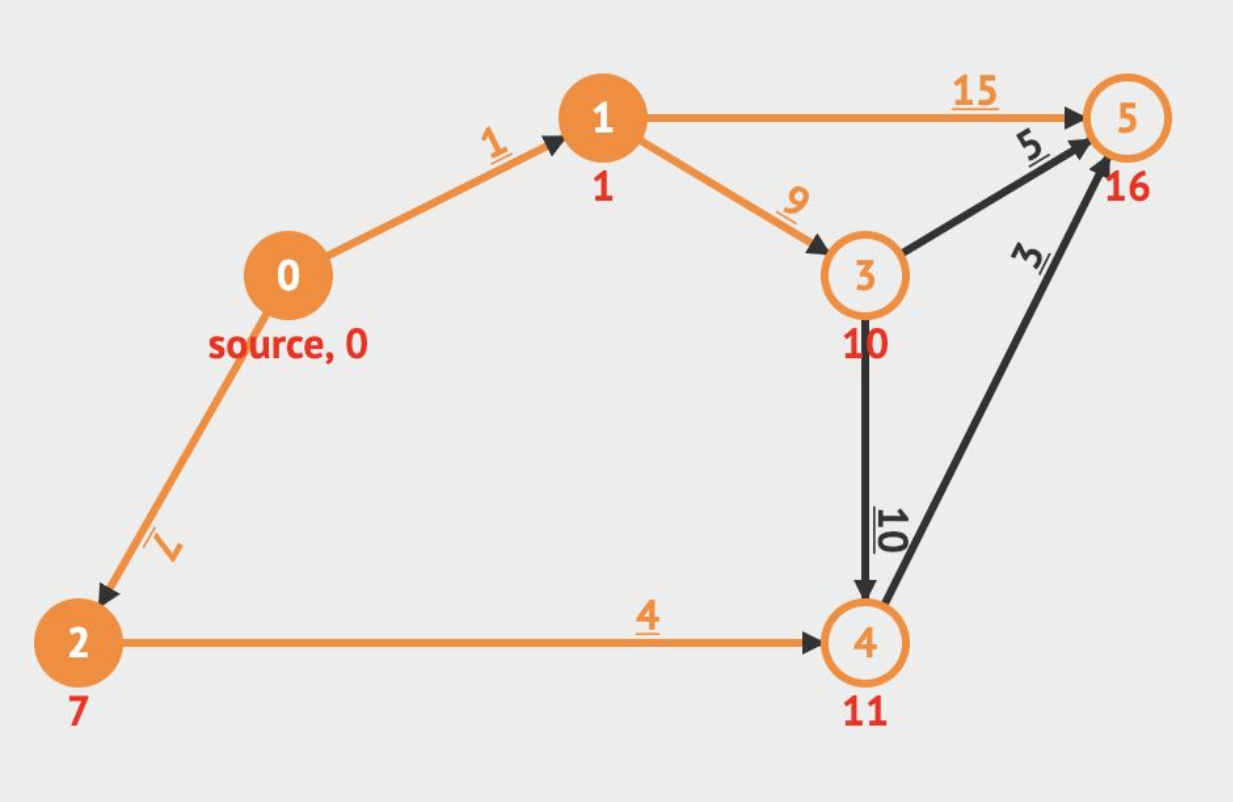
source: visualgo.net

Dijkstra's Algorithm



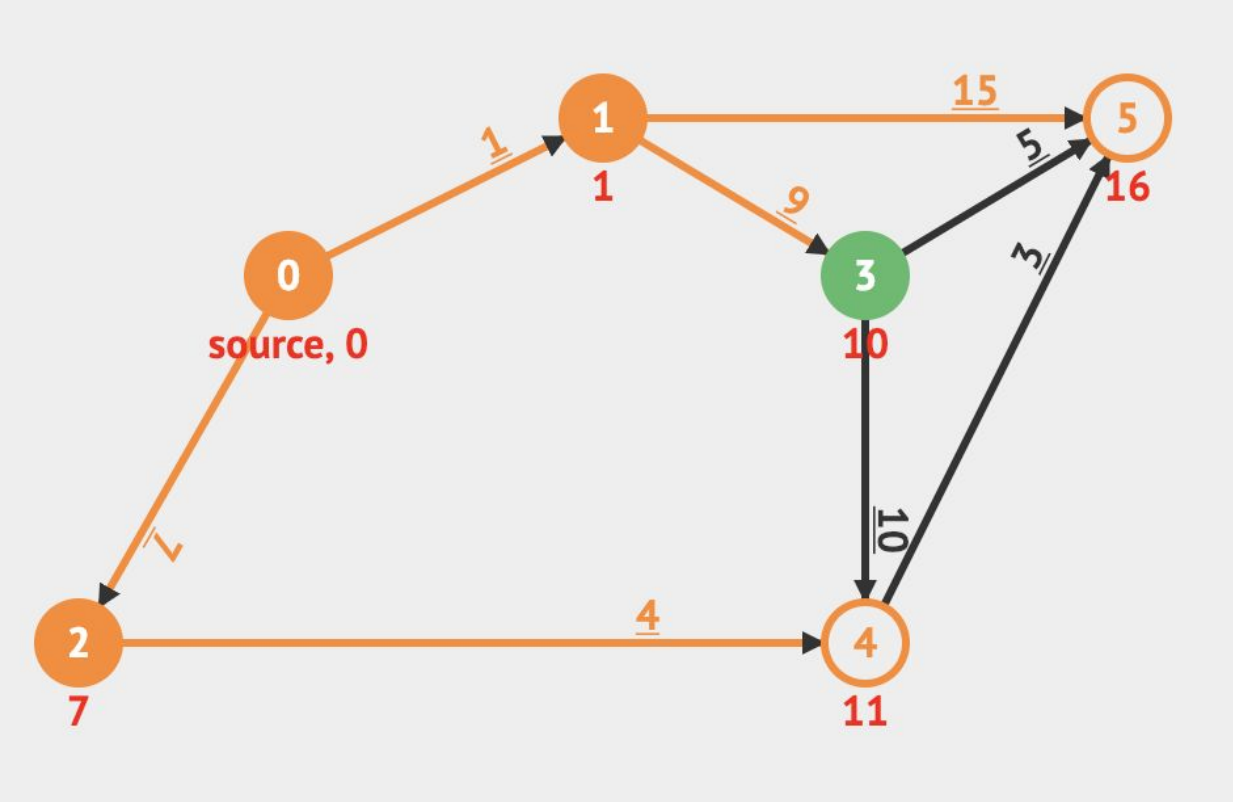
source: visualgo.net

Dijkstra's Algorithm



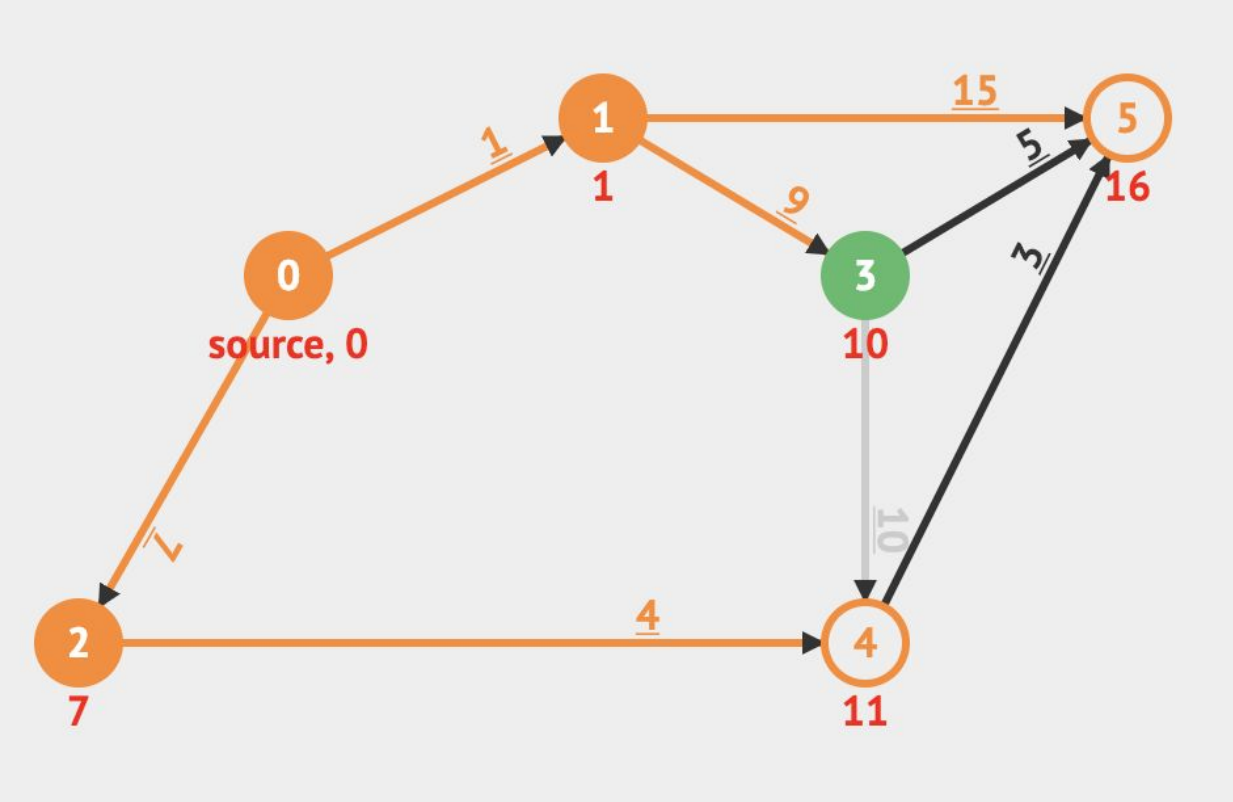
source: visualgo.net

Dijkstra's Algorithm



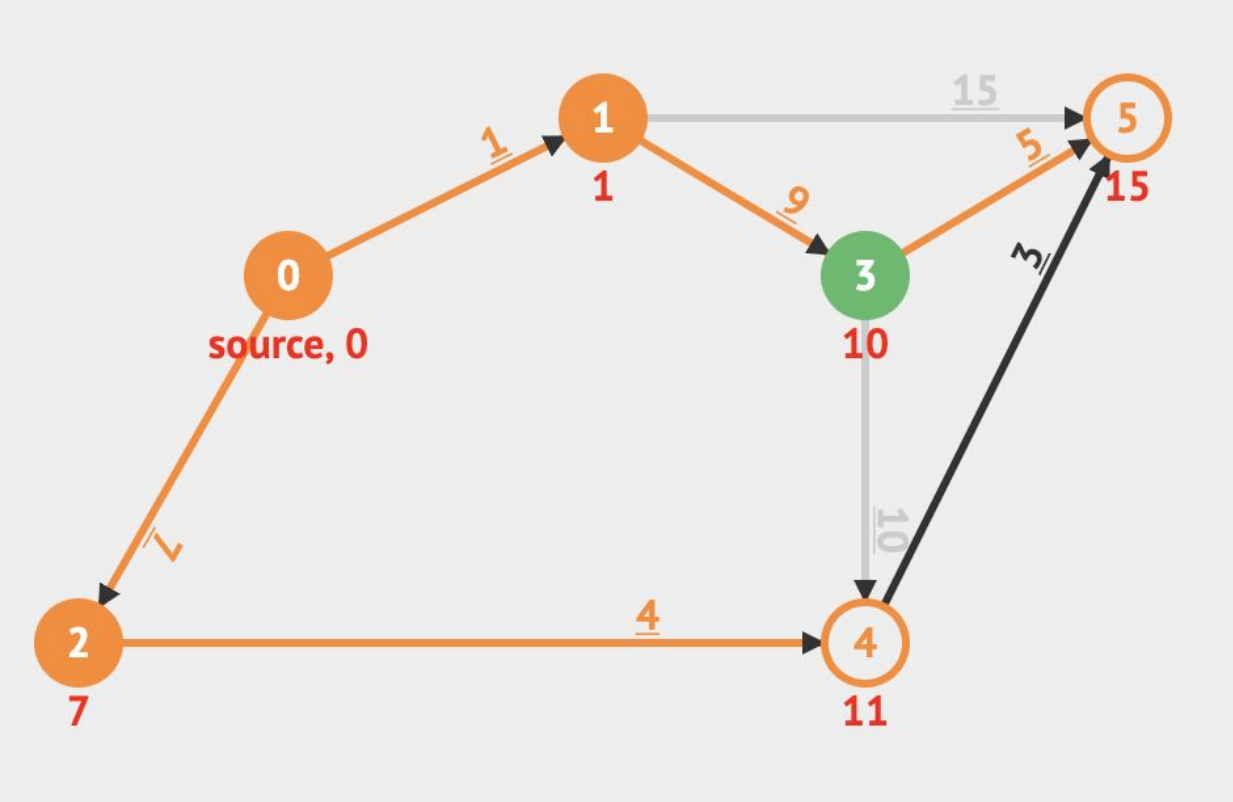
source: visualgo.net

Dijkstra's Algorithm



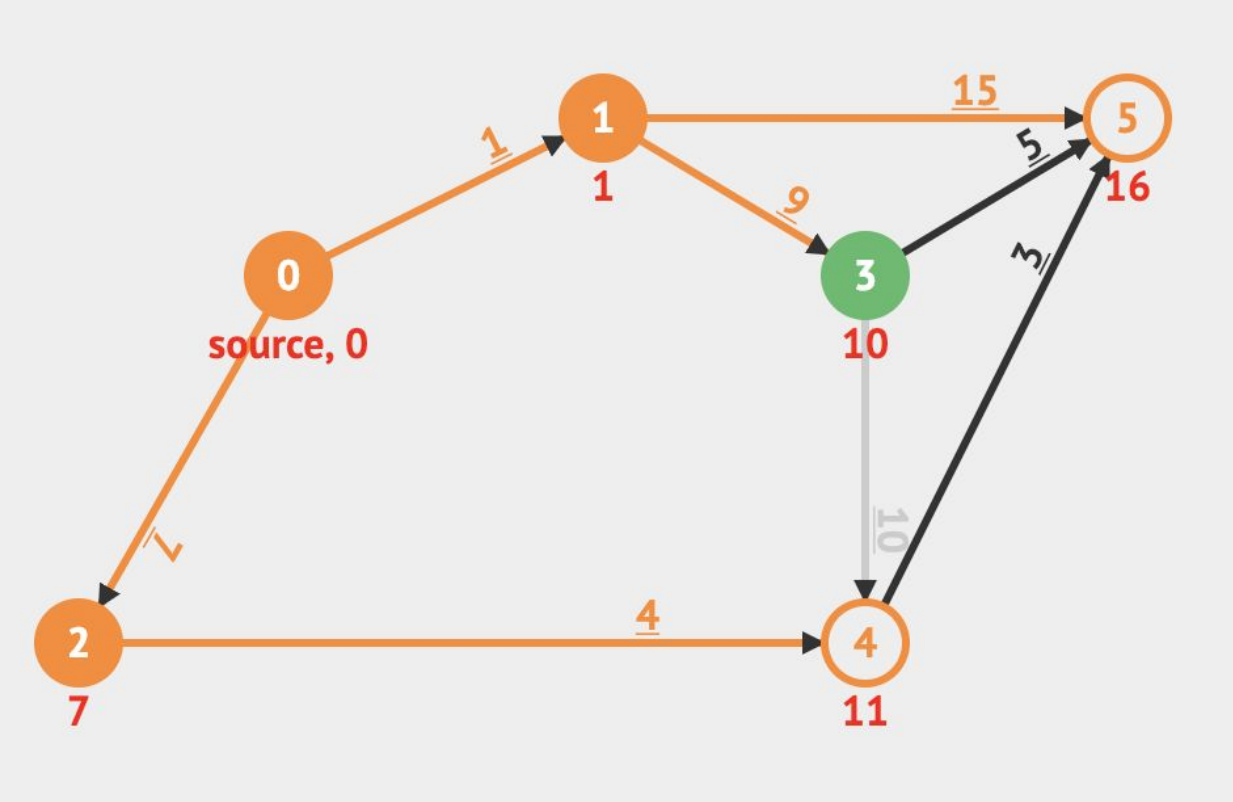
source: visualgo.net

Dijkstra's Algorithm



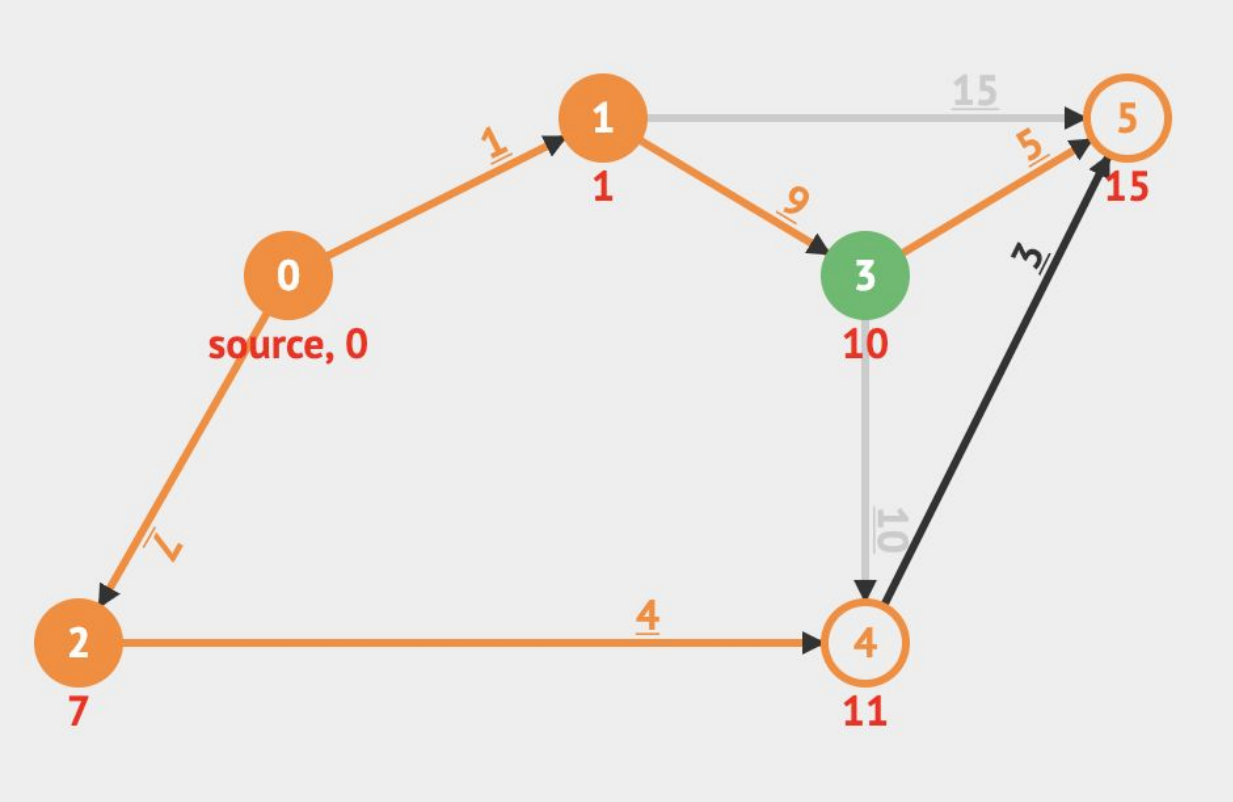
source: visualgo.net

Dijkstra's Algorithm



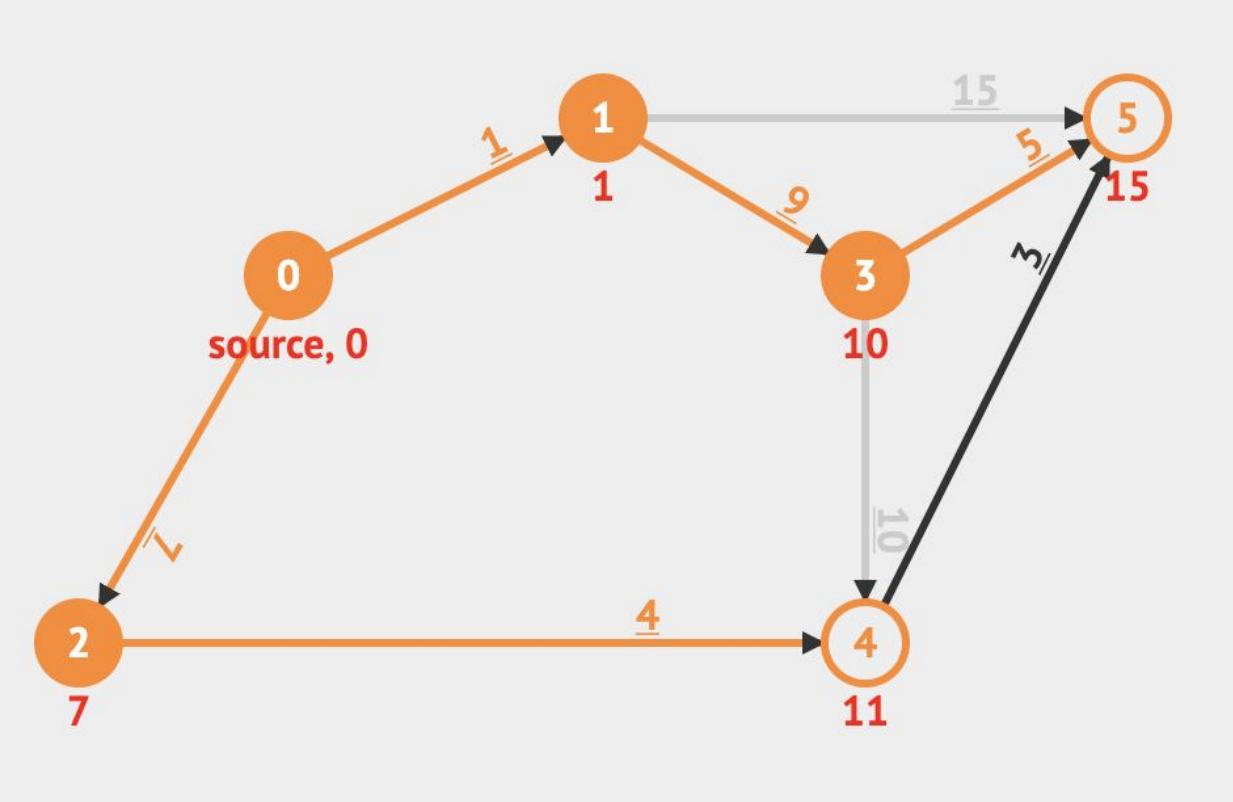
source: visualgo.net

Dijkstra's Algorithm



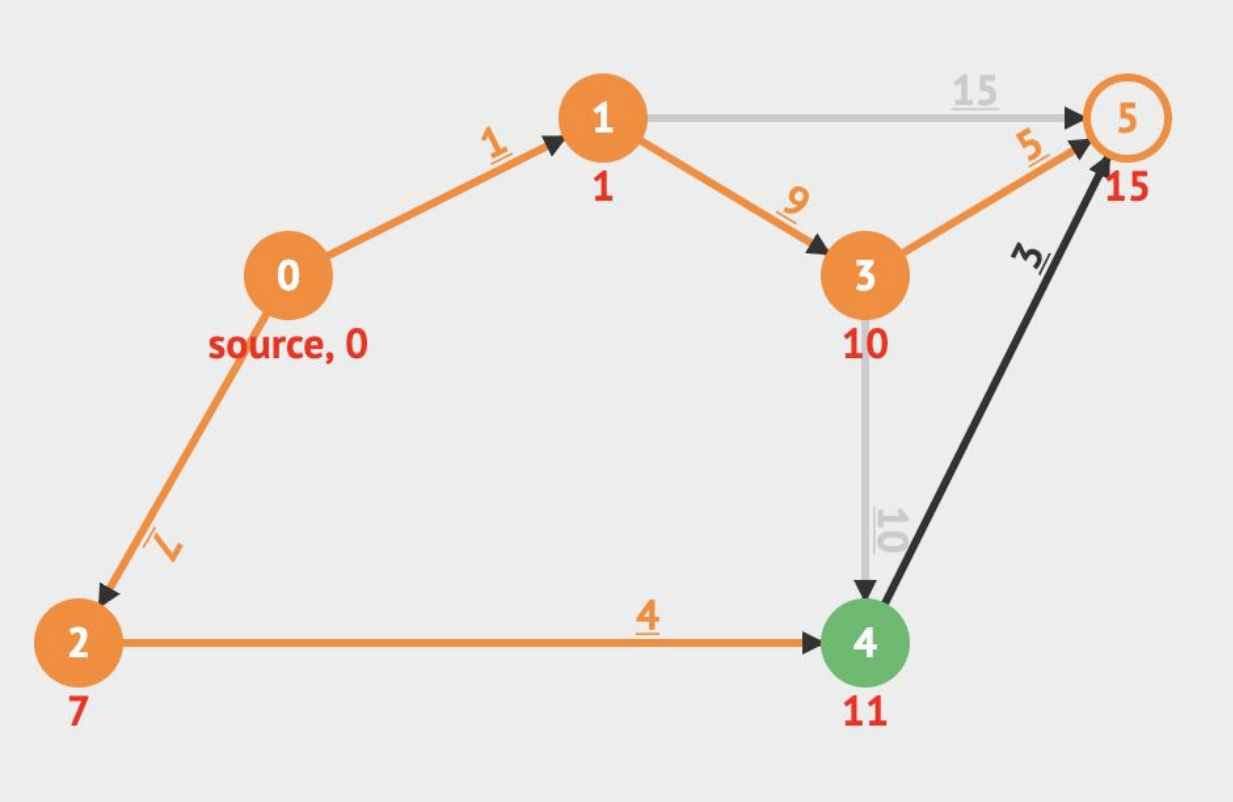
source: visualgo.net

Dijkstra's Algorithm



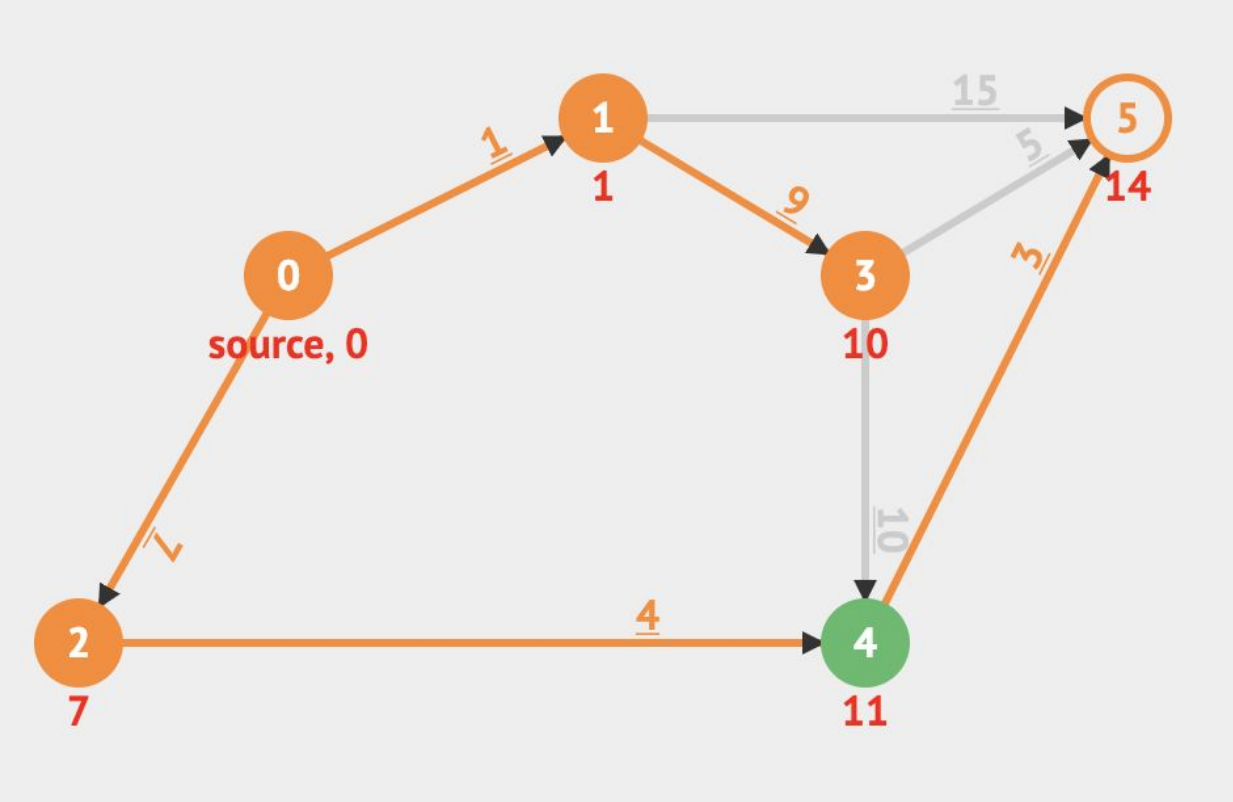
source: visualgo.net

Dijkstra's Algorithm



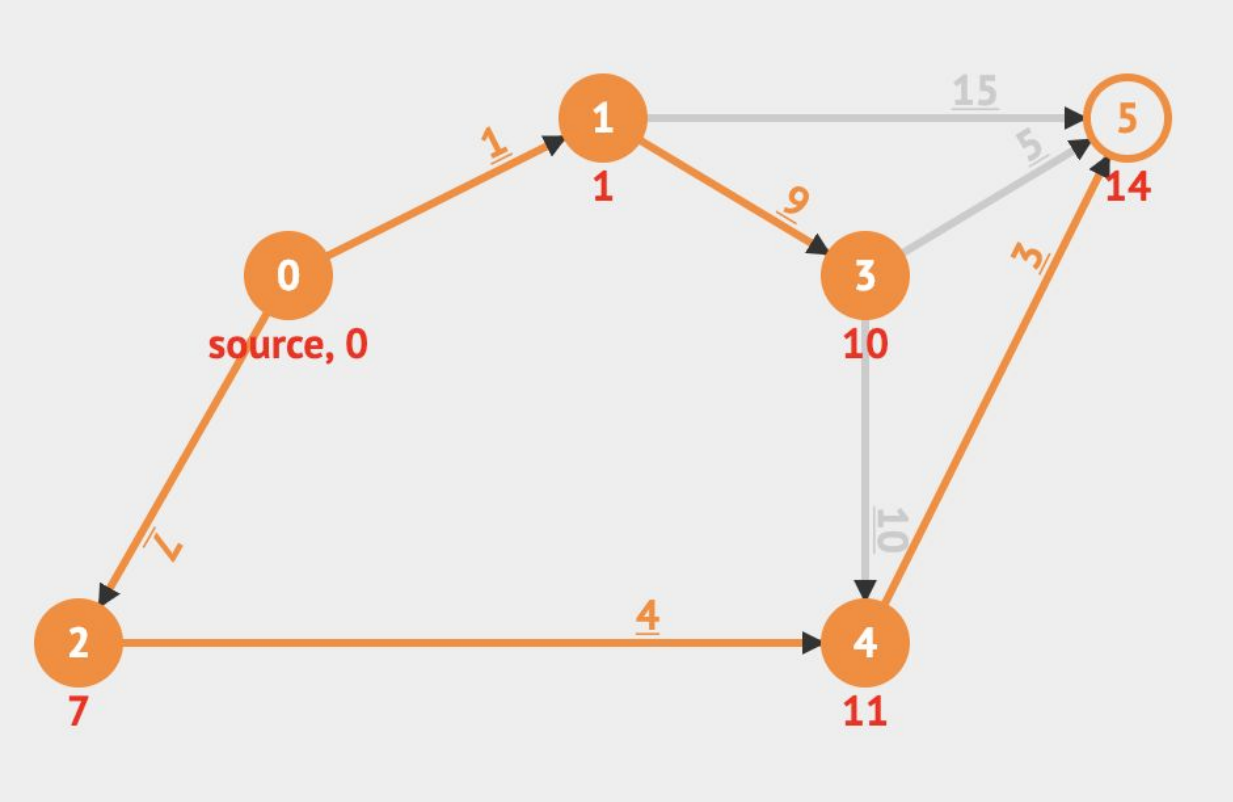
source: visualgo.net

Dijkstra's Algorithm



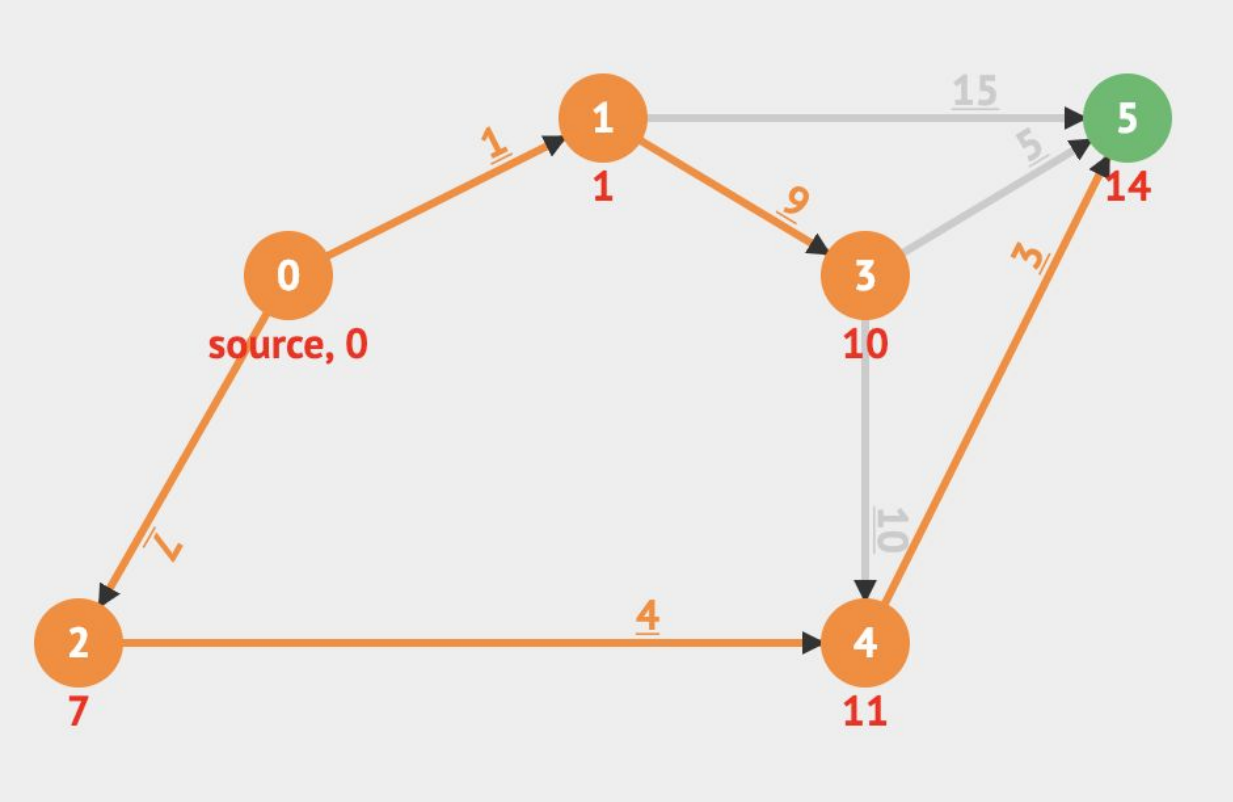
source: visualgo.net

Dijkstra's Algorithm



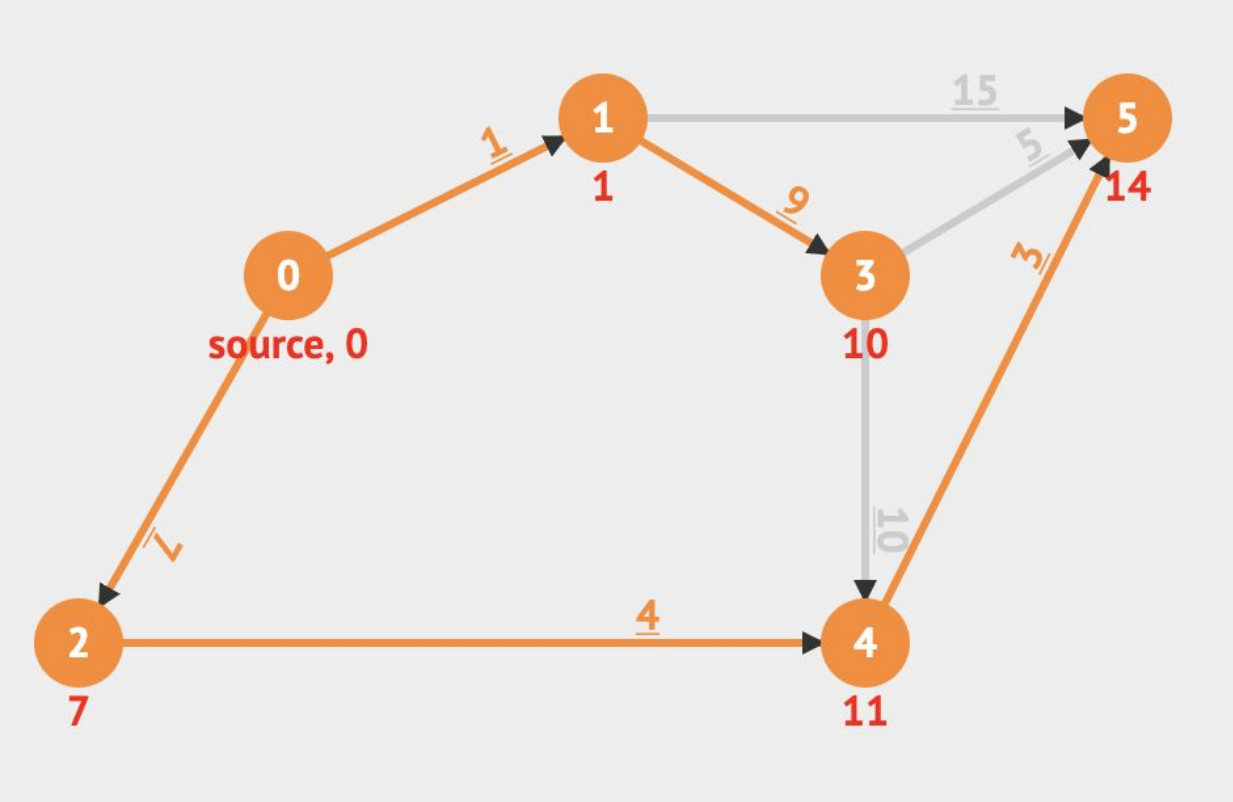
source: visualgo.net

Dijkstra's Algorithm



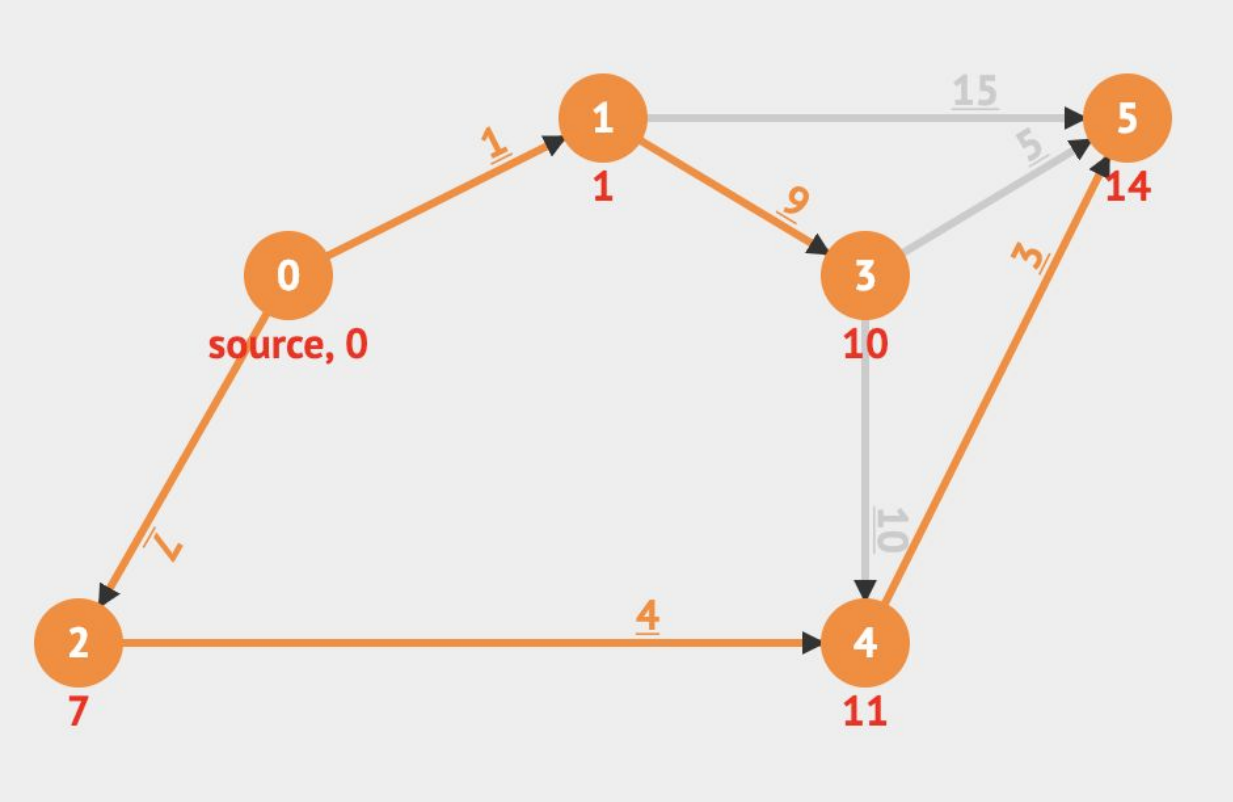
source: visualgo.net

Dijkstra's Algorithm



source: visualgo.net

Dijkstra's Algorithm



source: visualgo.net

Coding Tasks

Coding Tasks

Casinos

Hint: use `int dfs(...)`

FindCycle

Toposort22