Chapter 5: Graph Traversal

Algorithms

Kinds of graphs | Representation | Depth-first search (DFS) | Breadth-first search (BFS) | Mazes and shortest paths | Tarjan SCC | Topological sort | Complexity

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/algorithms/slides/Chapter_05_Graph_Traversal.html

Learning Outcomes

Graph basics

Tell apart directed, undirected, weighted, bipartite and complete graphs and trees; store a graph as an adjacency list or matrix.

DFS

Write DFS recursively and with a stack, trace the visit order step by step, and explain backtracking.

BFS

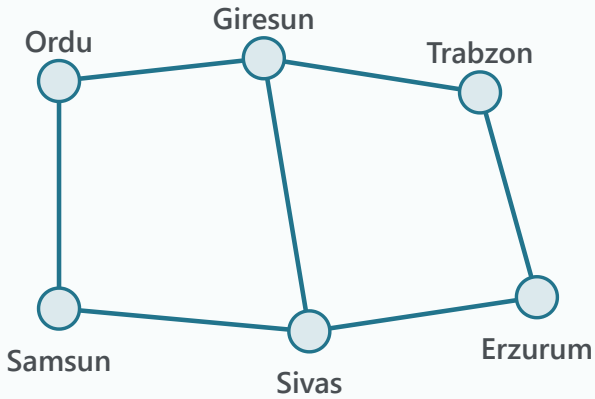
Write BFS with a queue; apply levels, parents and unweighted shortest paths to a maze.

Tarjan and analysis

Find strongly connected components with index / lowlink values; justify the O(V + E) running time of every traversal.

Graphs Are Everywhere

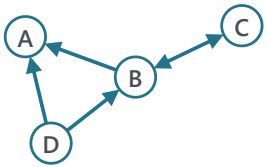
- The world is a **network**: cities are linked by roads, people by relationships, computers by cables.
- A **graph** G = (V, E) consists of **vertices** V and the **edges** E that join them.
- Graph algorithms **explore and make sense of** these networks: shortest paths, clustering, connectivity, flow…
- Uses: social networks, maps and navigation, network routing, compilers, games.



Cities are vertices, roads are edges.

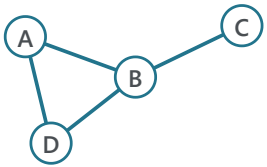
V = number of vertices, E = number of edges. Running times are stated in terms of these two: $O(V + E)$, $O(V^2)$…

Kinds of Graphs



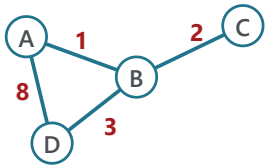
Directed

Edges are traversed only in the direction of the arrow.



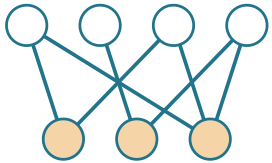
Undirected

Edges can be traversed both ways.



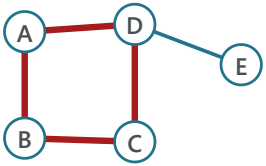
Weighted

Each edge carries a weight (cost, distance).



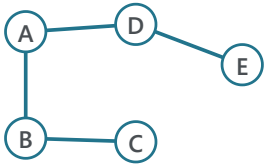
Bipartite

Edges only join vertices in different sets.



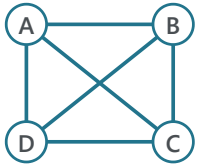
Cyclic

Some path returns to its start, such as A–D–C–B–A.



Tree

Connected and acyclic; exactly one path between any two vertices.



Complete

Every pair of vertices is joined: $n(n-1)/2$ edges.



Not simple

Has parallel edges and self-loops; a **simple graph** has neither.

INTRODUCTION

Families of Graph Algorithms

1 · Graph traversal

DFS goes deep from a vertex; **BFS** sweeps the network level by level.

2 · Shortest paths

Dijkstra: shortest paths from a source to every other vertex.

3 · Minimum spanning tree (MST)

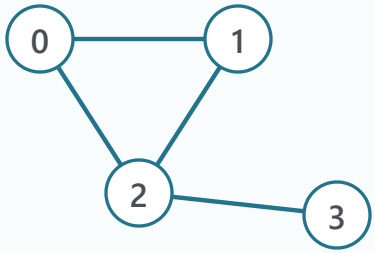
Kruskal: picks the edges that connect the network at minimum total cost.

4 · Network flow

The maximum flow that can be sent from source to sink in a capacitated network.

Algorithm	Everyday example
DFS	Escaping a maze: follow one corridor to its end, then back up
BFS	News spreading through a city: first close friends, then their friends
Dijkstra	Finding the fastest delivery route for a courier

Storing a Graph



V = 4, E = 4

Adjacency list

```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```

Adjacency matrix

	0	1	2	3
0	0	**1**	**1**	0
1	**1**	0	**1**	0
2	**1**	**1**	0	**1**
3	0	0	**1**	0

	Adjacency list	Adjacency matrix
Memory	O(V + E)	$O(V^2)$
All neighbors of u	O(deg(u))	O(V)
Is (u, v) an edge?	O(deg(u))	O(1)
Traversal (DFS / BFS)	**O(V + E)**	**$O(V^2)$**

The Java code in this chapter uses adjacency lists: `List<Integer>[] adj`.

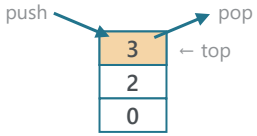
TRAVERSAL

Graph Traversal

- A traversal visits every (reachable) vertex **exactly once** to explore the graph's structure **systematically**.
- There are two main strategies; they differ in **which data structure** supplies the next vertex:

Depth-First Search (DFS)

Uses a **stack** (LIFO). It starts at a vertex and goes as **deep** as it can; when the neighbors run out it **backtracks**.



Breadth-First Search (BFS)

Uses a **queue** (FIFO). It visits vertices at distance 1, then distance 2, … **level by level**.



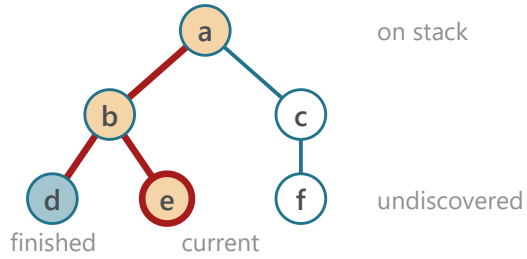
A visited array is essential: unlike a tree, a graph can have cycles. Without marking `visited[]`, the algorithm circles the same cycle forever.

DFS: The Idea

- In a maze, follow a corridor **until it dead-ends**; then **back up** to the last fork and try a corridor you have not taken yet.
- In the recursive version the **call stack** remembers where to back up to; the iterative version uses an explicit **stack**.
- The visited vertices, with the edges that first reached them, form a **DFS tree**.
- The visit order depends on the **order of neighbors**; one graph can have several valid DFS orders.

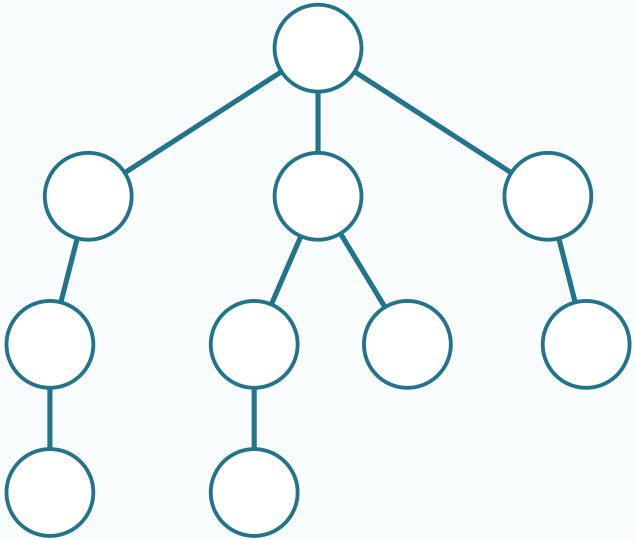
Vertex states

white: not yet discovered
yellow: discovered, still in progress (on the stack)
dark: finished, all neighbors examined



DFS answers: "Is it reachable?", "Is there a cycle?", "How many components?", "In what order?" (topological sort), "What are the strongly connected components?" (Tarjan).

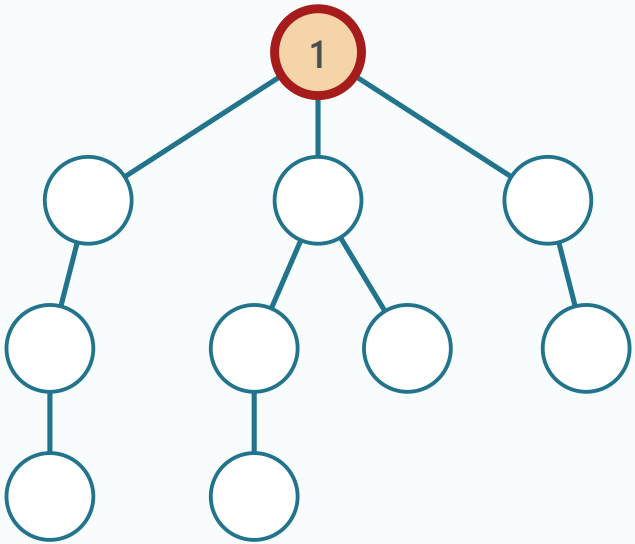
DFS Step by Step: Visit Order



Stack (recursion): empty

Start: no vertex is visited yet. DFS starts at the root; each vertex gets its **visit number**.

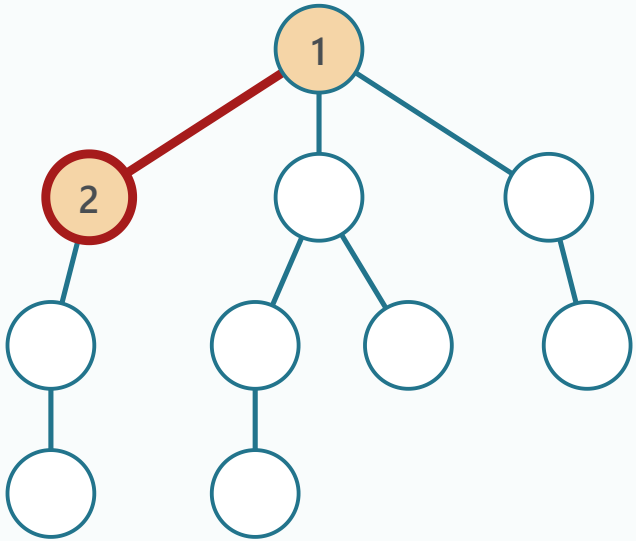
DFS Step by Step: Visit Order



Stack (recursion): 1

Visit the root (1). Pick its first neighbor and go **deeper**.

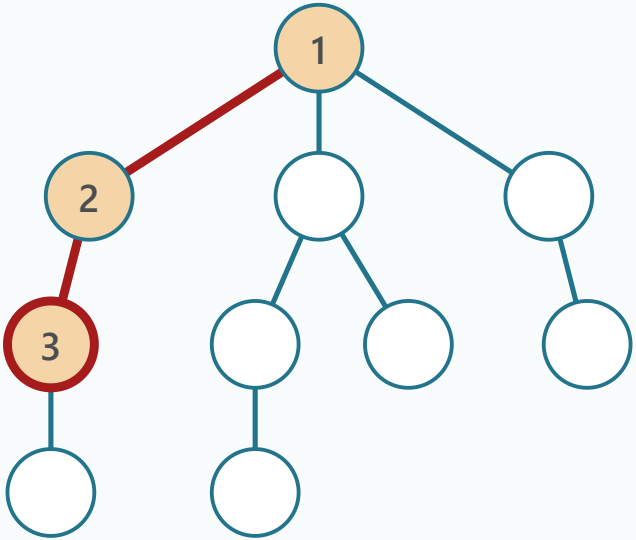
DFS Step by Step: Visit Order



Stack (recursion): 1 → 2

Descend to the first undiscovered neighbor of 1: **2**. The stack grows.

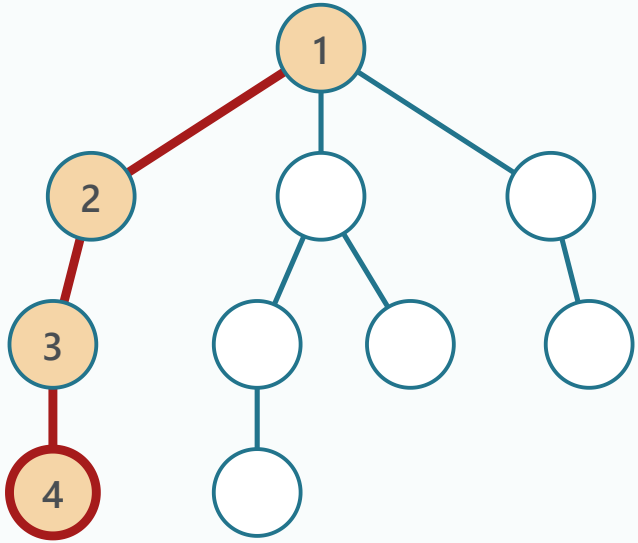
DFS Step by Step: Visit Order



Stack (recursion): 1 → 2 → 3

Descend to the first undiscovered neighbor of 2: **3**. The stack grows.

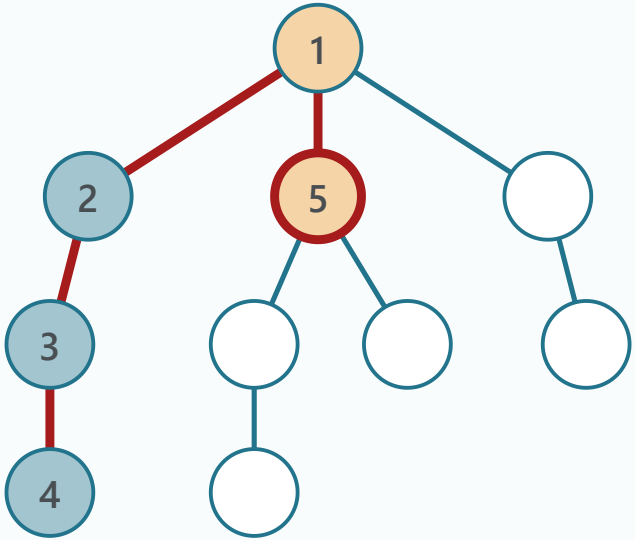
DFS Step by Step: Visit Order



Stack (recursion): 1 → 2 → 3 → 4

Descend to the first undiscovered neighbor of 3: **4**. The stack grows.

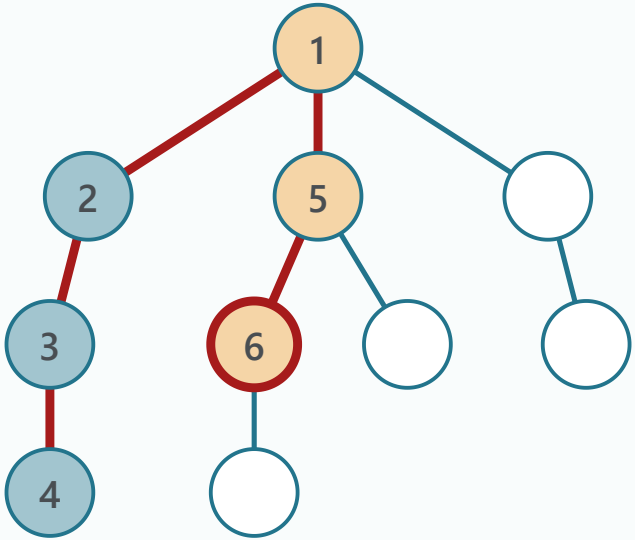
DFS Step by Step: Visit Order



Stack (recursion): 1 → 5

4, 3 and 2 have no undiscovered neighbors left → **backtrack**. Visit the next child of 1: **5**.

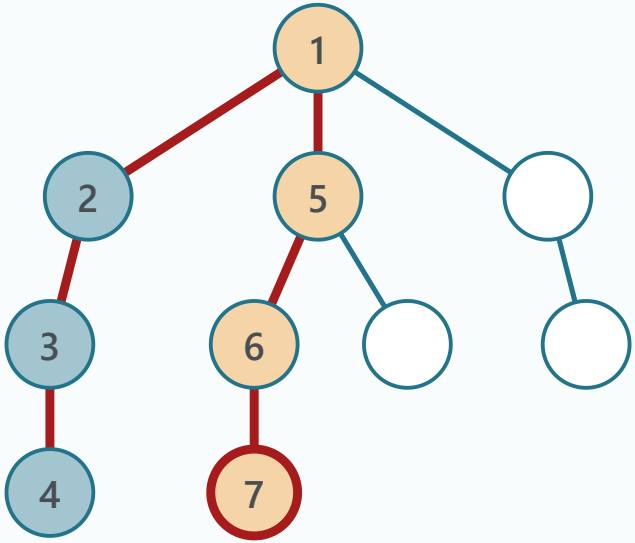
DFS Step by Step: Visit Order



Stack (recursion): 1 → 5 → 6

Descend to the first undiscovered neighbor of 5: **6**. The stack grows.

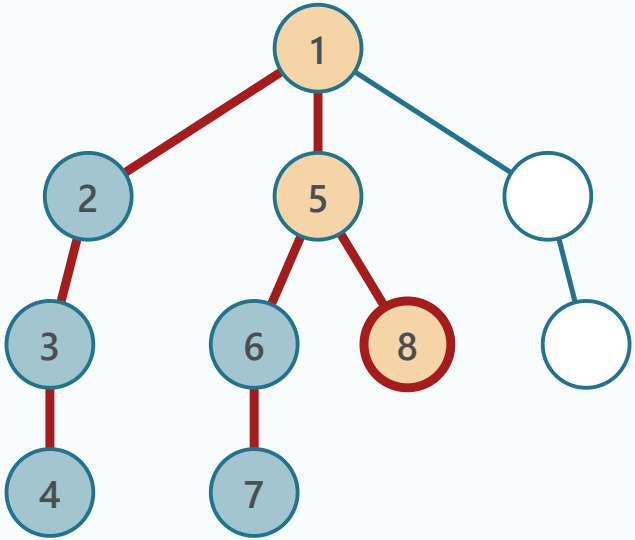
DFS Step by Step: Visit Order



Stack (recursion): 1 → 5 → 6 → 7

Descend to the first undiscovered neighbor of 6: **7**. The stack grows.

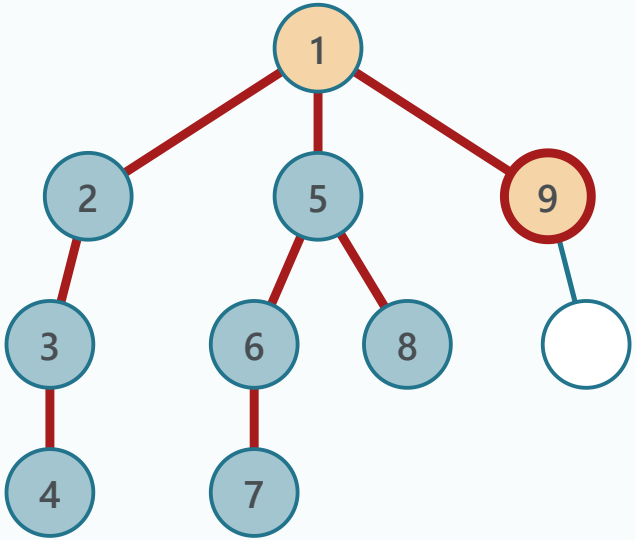
DFS Step by Step: Visit Order



Stack (recursion): 1 → 5 → 8

7 and 6 have no undiscovered neighbors left → **backtrack**. Visit the next child of 5: **8**.

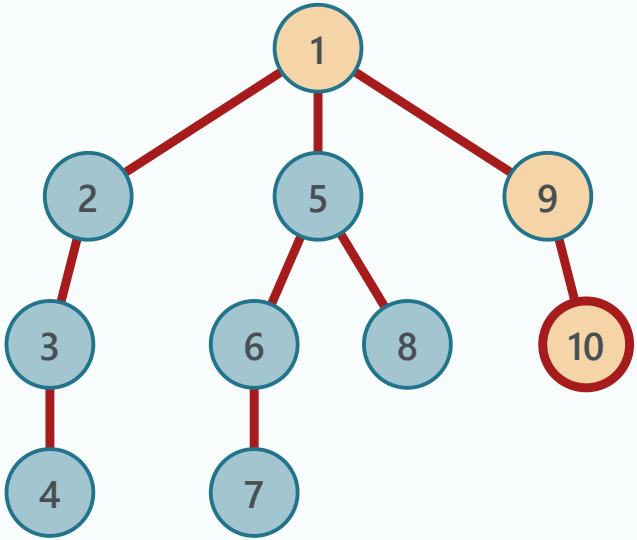
DFS Step by Step: Visit Order



Stack (recursion): 1 → 9

8 and 5 have no undiscovered neighbors left → **backtrack**. Visit the next child of 1: **9**.

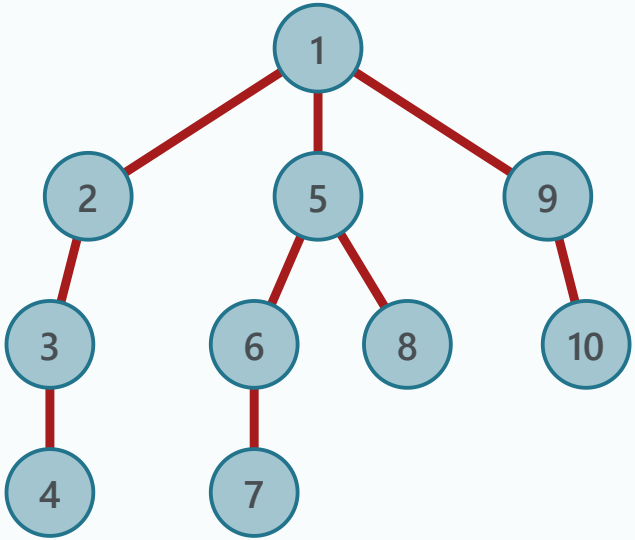
DFS Step by Step: Visit Order



Stack (recursion): 1 → 9 → 10

Descend to the first undiscovered neighbor of 9: **10**. The stack grows.

DFS Step by Step: Visit Order



Stack (recursion): empty

All vertices visited; the stack is empty. Visit order: **1, 2, 3, …, 10**: each branch was followed to its end before backtracking.

Recursive DFS: Pseudocode

Standard pseudocode

```
procedure DFS(G, v) is
  label v as discovered
  for all directed edges from v to w
      that are in G.adjacentEdges(v) do
    if vertex w is not labeled
        as discovered then
      call DFS(G, w)
```

Plain version

```
DFS(node, visited):
  if node is not
        in visited:
    add node to visited
    process node (e.g., print it)
    for each neighbor of node:
      DFS(neighbor, visited)
```

- The left version checks a neighbor **before calling**; the right one checks **at the start** of the call. Both give the same order; the right one makes extra calls that return at once.
- To cover every vertex of a disconnected graph, restart DFS from **each unvisited vertex** (each restart finds one connected component).

Iterative DFS: Pseudocode

Standard pseudocode

```
procedure DFS_iterative(G, v) is
  let S be a stack
  S.push(v)
  while S is not empty do
    v = S.pop()
    if v is not labeled as discovered then
      label v as discovered
      for all edges from v to w
          in G.adjacentEdges(v) do
        S.push(w)
```

Plain version

```
DFS(start):
  create an empty stack
  push start onto the stack
  while the stack is not empty:
    current = pop the stack
    if current is not
          in visited:
      add current to visited
      for each neighbor of current:
        push neighbor onto the stack
```

- An **explicit stack** instead of recursion: no risk of `StackOverflowError` on very deep graphs.
- Marking happens **on pop**, so a vertex can enter the stack several times (stack size O(E)).
- Since neighbors are pushed, the last one pushed comes out first: the order is the **reverse** of the recursive version. To match it, push the neighbors in reverse.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

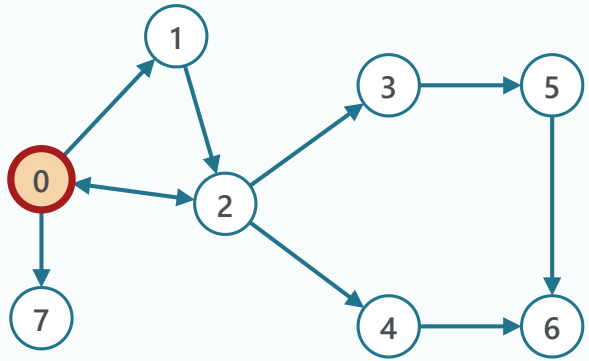
`v = 0` `w = –`

CALL STACK

`dfs(0)`

OUTPUT

```
0
```



Directed example: 0→1, 0→2, 0→7, 1→2, 2→0, 2→3, 2→4, 3→5, 4→6, 5→6. `dfs(0)` : mark 0 and print it.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

`v = 1` `w = –`

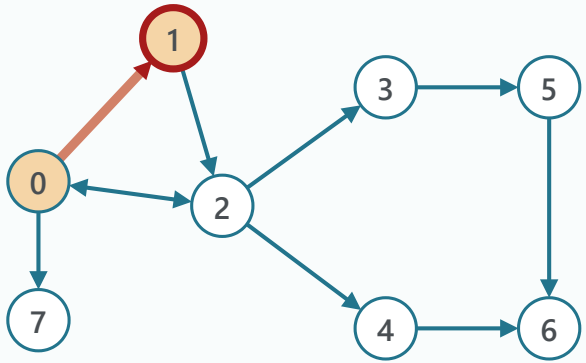
CALL STACK

`dfs(1)`

`dfs(0)`

OUTPUT

```
0 1
```



Neighbor 1 of 0 is unvisited → `dfs(1)`: mark it, print it. A new frame is pushed.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

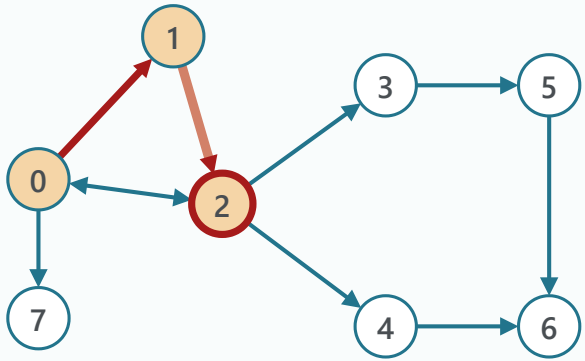
`v = 2` `w = –`

CALL STACK

- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

```
0 1 2
```



Neighbor 2 of 1 is unvisited → `dfs(2)`: mark it, print it. A new frame is pushed.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int w : adj[v])
▶     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

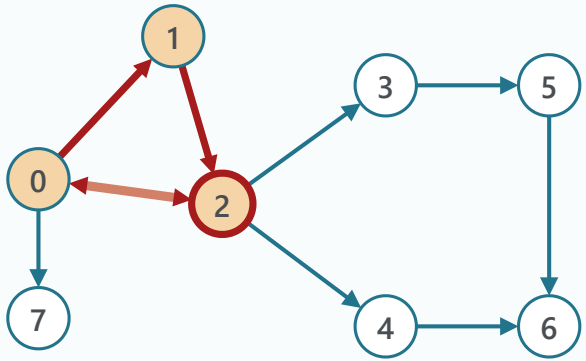
`v = 2` `w = 0`

CALL STACK

- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

```
0 1 2
```



Neighbor 0 of 2 is **already visited** → skip it (no going around the cycle).

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

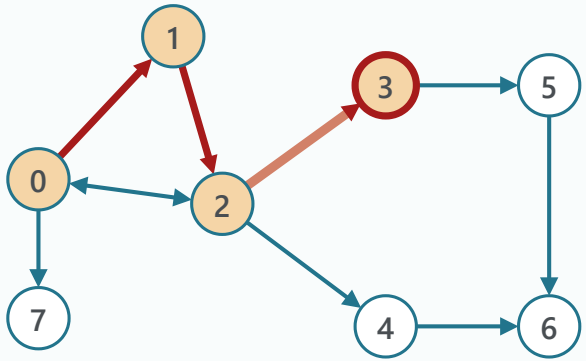
`v = 3` `w = –`

CALL STACK

- `dfs(3)`
- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

```
0 1 2 3
```



Neighbor 3 of 2 is unvisited → `dfs(3)` : mark it, print it. A new frame is pushed.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

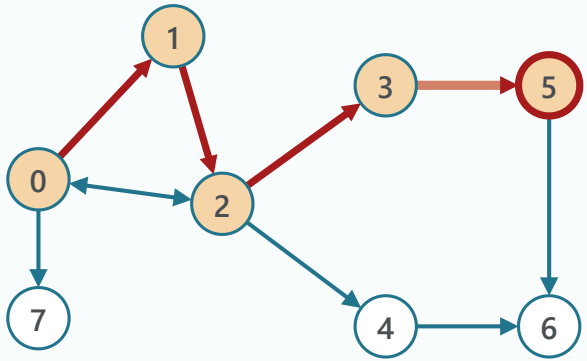
`v = 5` `w = –`

CALL STACK

- `dfs(5)`
- `dfs(3)`
- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

```
0 1 2 3 5
```



Neighbor 5 of 3 is unvisited → `dfs(5)`: mark it, print it. A new frame is pushed.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

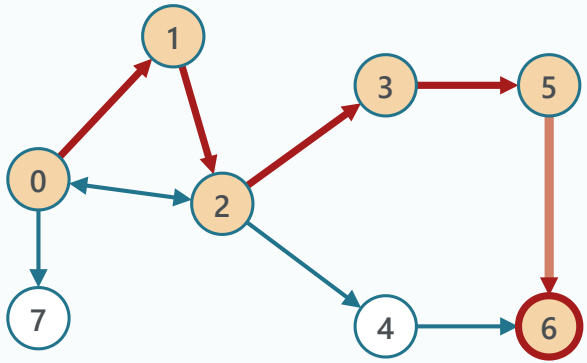
`v = 6` `w = –`

CALL STACK

- `dfs(6)`
- `dfs(5)`
- `dfs(3)`
- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

```
0 1 2 3 5 6
```



Neighbor 6 of 5 is unvisited → `dfs(6)` : mark it, print it. A new frame is pushed.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
▶       dfs(w, visited);
7 }
```

VARIABLES

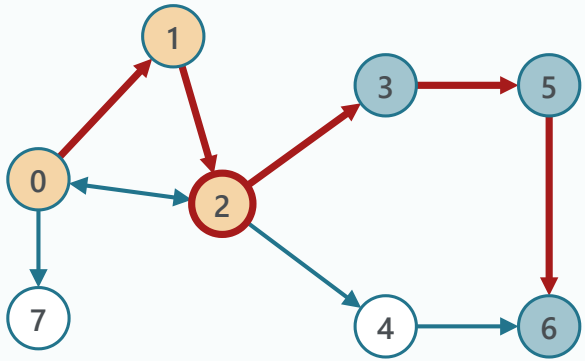
`v = 2` `w = 3`

CALL STACK

- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

```
0 1 2 3 5 6
```



6, 5 and 3 are finished (no neighbors left) → their calls return and control comes back to **2**: backtracking.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

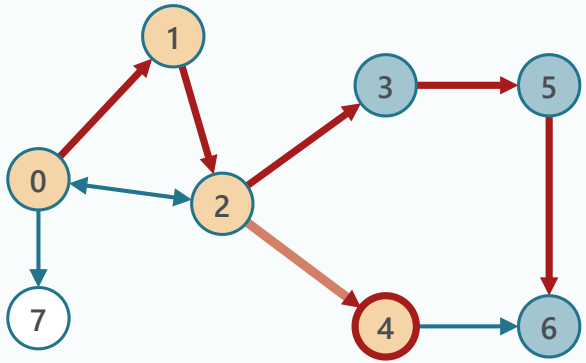
`v = 4` `w = –`

CALL STACK

- `dfs(4)`
- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

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



Neighbor 4 of 2 is unvisited → `dfs(4)` : mark it, print it. A new frame is pushed.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int w : adj[v])
▶     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

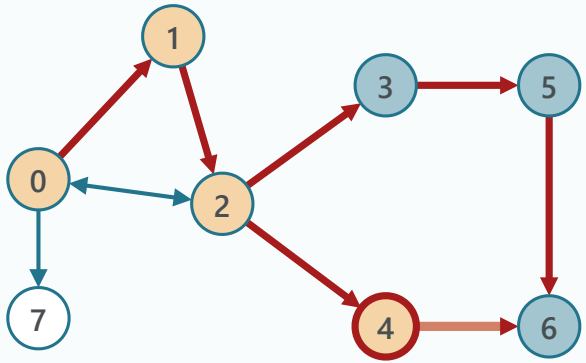
`v = 4` `w = 6`

CALL STACK

- `dfs(4)`
- `dfs(2)`
- `dfs(1)`
- `dfs(0)`

OUTPUT

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



Neighbor 6 of 4 is **already visited** → skip it (no going around the cycle).

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
▶       dfs(w, visited);
7 }
```

VARIABLES

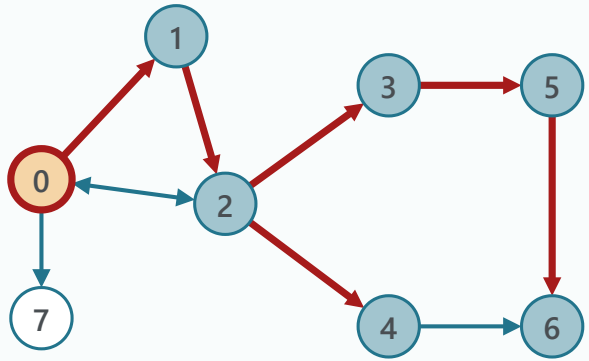
`v = 0` `w = 1`

CALL STACK

`dfs(0)`

OUTPUT

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



4, 2 and 1 are finished (no neighbors left) → their calls return and control comes back to **0**: backtracking.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int w : adj[v])
▶     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

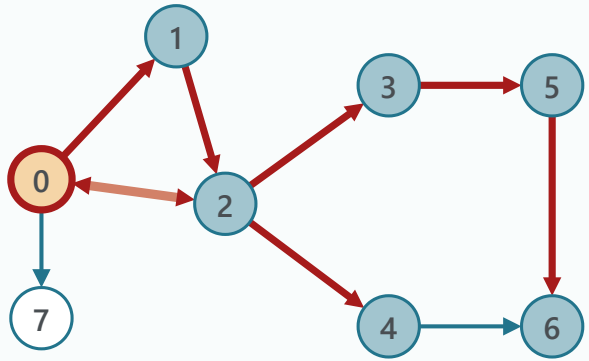
`v = 0` `w = 2`

CALL STACK

`dfs(0)`

OUTPUT

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



Neighbor 2 of 0 is **already visited** → skip it (no going around the cycle).

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
7 }
```

VARIABLES

`v = 7` `w = –`

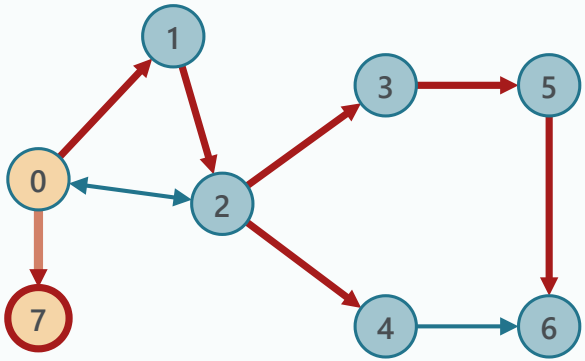
CALL STACK

`dfs(7)`

`dfs(0)`

OUTPUT

```
0 1 2 3 5 6 4 7
```



Neighbor 7 of 0 is unvisited → `dfs(7)`: mark it, print it. A new frame is pushed.

Recursive DFS: Tracing `dfs(0)`

```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int w : adj[v])
5     if (!visited[w])
6       dfs(w, visited);
  }
```

VARIABLES

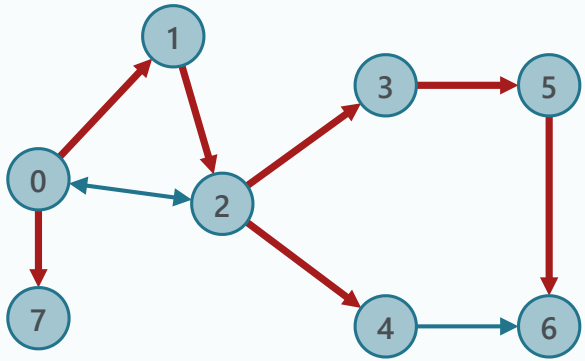
`v = –` `w = –`

CALL STACK

empty

OUTPUT

```
0 1 2 3 5 6 4 7
```



7 and 0 are finished; `dfs(0)` returns. Output: **0 1 2 3 5 6 4 7**. The bold edges form the **DFS tree**.

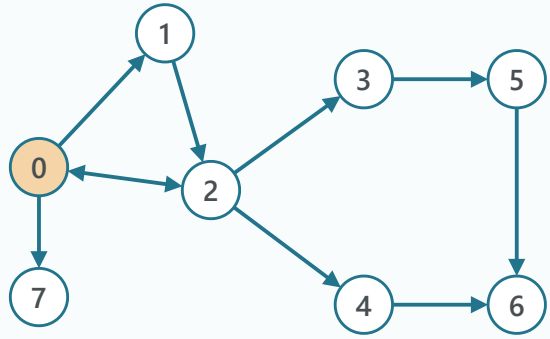
Stack DFS: Tracing `dfsStack(0)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
▶    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
8        visited[v] = true;
9        System.out.print(v + " ");
10       for (int w : adj[v])
11         stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = –` `stack = [0]`

OUTPUT



stack (bottom → top)

0

Push the start vertex 0. No vertex is marked yet.

Stack DFS: Tracing `dfsStack(0)`

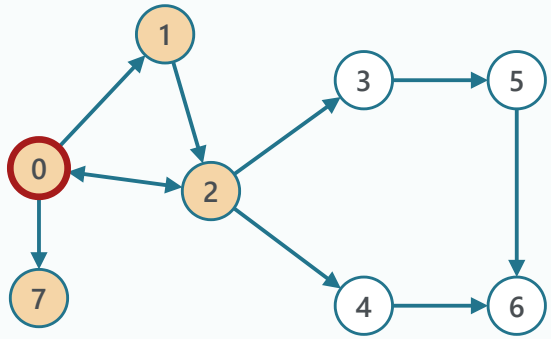
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
▶        for (int w : adj[v])
▶          stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 0` `stack = [1, 2, 7]`

OUTPUT

```
0
```



stack (bottom → top)

1	2	7

Pop 0: mark and print; push 1, 2, 7 → the last one pushed, **7**, is on top.

Stack DFS: Tracing `dfsStack(0)`

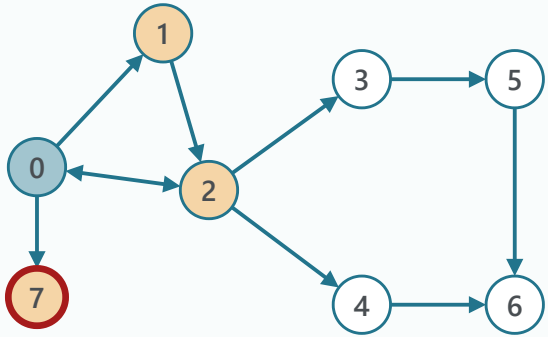
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
10       for (int w : adj[v])
11         stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 7` `stack = [1, 2]`

OUTPUT

```
0 7
```



stack (bottom → top)



Pop 7: mark and print; no neighbors, nothing is pushed.

Stack DFS: Tracing `dfsStack(0)`

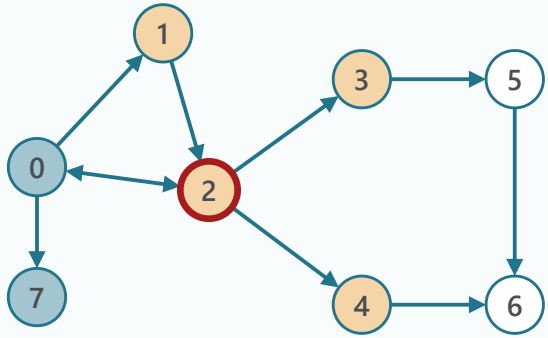
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
▶        for (int w : adj[v])
▶          stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 2` `stack = [1, 0, 3, 4]`

OUTPUT

```
0 7 2
```



stack (bottom → top)

1	0	3	4

Pop 2: mark and print; push 0, 3, 4 → the last one pushed, **4**, is on top.

Stack DFS: Tracing `dfsStack(0)`

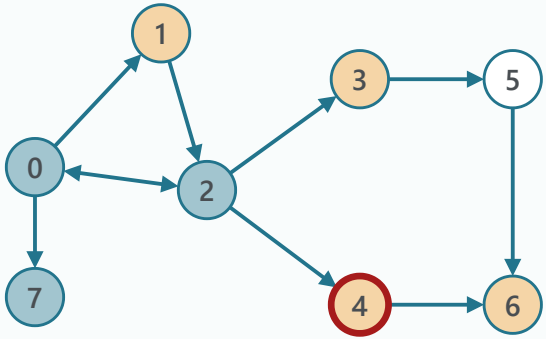
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
▶        for (int w : adj[v])
▶          stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 4` `stack = [1, 0, 3, 6]`

OUTPUT

```
0 7 2 4
```



stack (bottom → top)

1	0	3	6

Pop 4: mark and print; push 6 → the last one pushed, **6**, is on top.

Stack DFS: Tracing `dfsStack(0)`

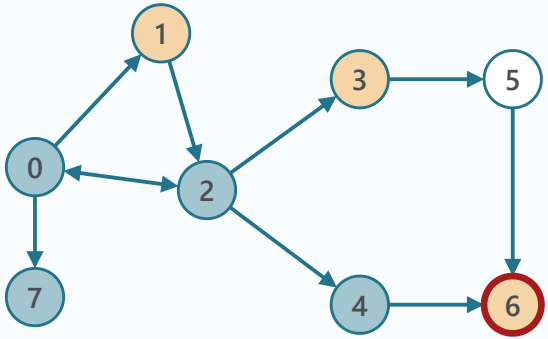
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
10       for (int w : adj[v])
11         stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 6` `stack = [1, 0, 3]`

OUTPUT

```
0 7 2 4 6
```



stack (bottom → top)

1	0	3

Pop 6: mark and print; no neighbors, nothing is pushed.

Stack DFS: Tracing `dfsStack(0)`

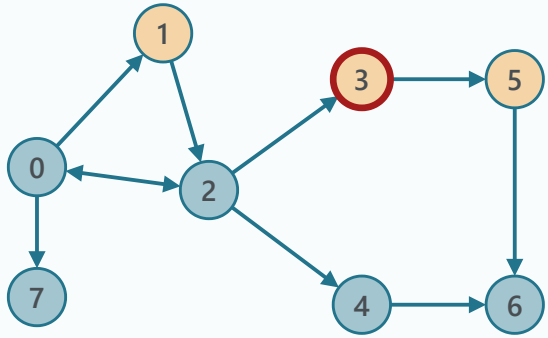
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
▶        for (int w : adj[v])
▶          stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 3` `stack = [1, 0, 5]`

OUTPUT

```
0 7 2 4 6 3
```



stack (bottom → top)

1	0	5

Pop 3: mark and print; push 5 → the last one pushed, **5**, is on top.

Stack DFS: Tracing `dfsStack(0)`

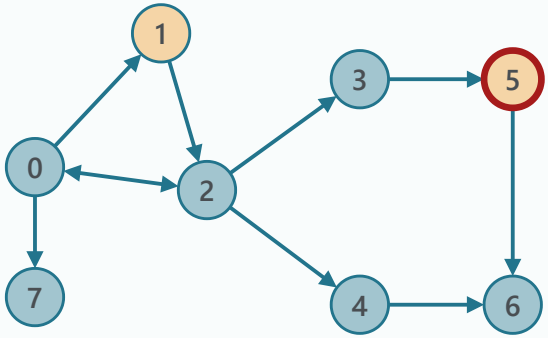
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
▶        for (int w : adj[v])
▶          stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 5` `stack = [1, 0, 6]`

OUTPUT

```
0 7 2 4 6 3 5
```



stack (bottom → top)

1	0	6

Pop 5: mark and print; push 6 → the last one pushed, **6**, is on top.

Stack DFS: Tracing `dfsStack(0)`

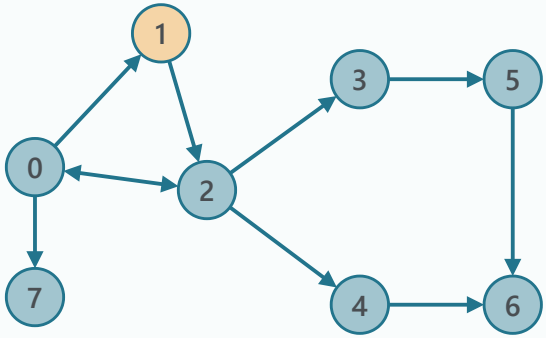
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {
8        visited[v] = true;
9        System.out.print(v + " ");
10       for (int w : adj[v])
11         stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 6` `stack = [1, 0]`

OUTPUT

```
0 7 2 4 6 3 5
```



stack (bottom → top)

1	0

Pop 6, but it is **already visited** → do nothing. (A vertex can be pushed more than once.)

Stack DFS: Tracing `dfsStack(0)`

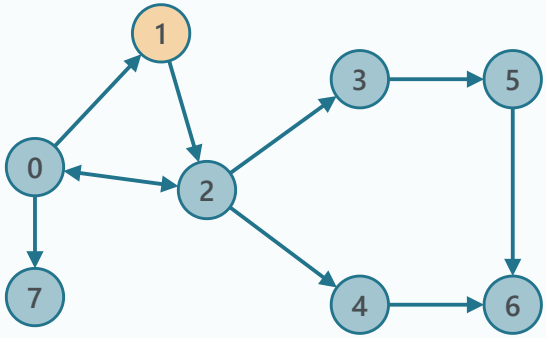
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {
8        visited[v] = true;
9        System.out.print(v + " ");
10       for (int w : adj[v])
11         stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 0` `stack = [1]`

OUTPUT

```
0 7 2 4 6 3 5
```



stack (bottom → top)

1

Pop 0, but it is **already visited** → do nothing. (A vertex can be pushed more than once.)

Stack DFS: Tracing `dfsStack(0)`

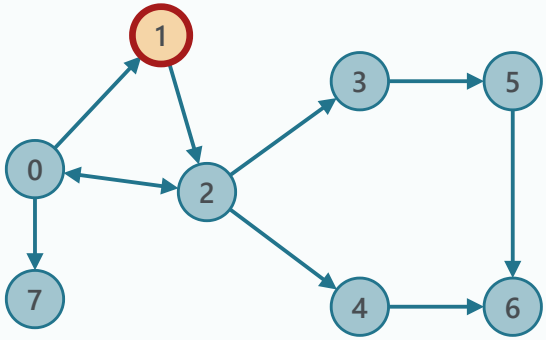
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
▶        visited[v] = true;
▶        System.out.print(v + " ");
▶        for (int w : adj[v])
▶          stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 1` `stack = [2]`

OUTPUT

```
0 7 2 4 6 3 5 1
```



stack (bottom → top)

2

Pop 1: mark and print; push 2 → the last one pushed, **2**, is on top.

Stack DFS: Tracing `dfsStack(0)`

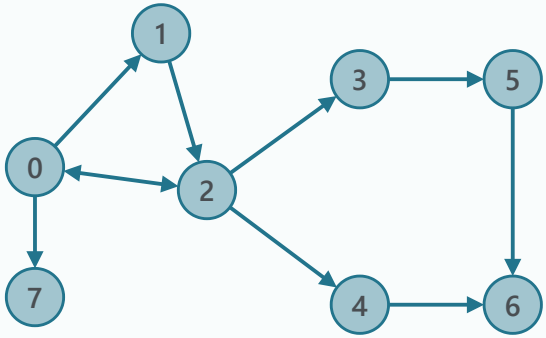
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
5    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {
8        visited[v] = true;
9        System.out.print(v + " ");
10       for (int w : adj[v])
11         stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = 2` `stack = []`

OUTPUT

```
0 7 2 4 6 3 5 1
```



stack (bottom → top)

Pop 2, but it is **already visited** → do nothing. (A vertex can be pushed more than once.)

Stack DFS: Tracing `dfsStack(0)`

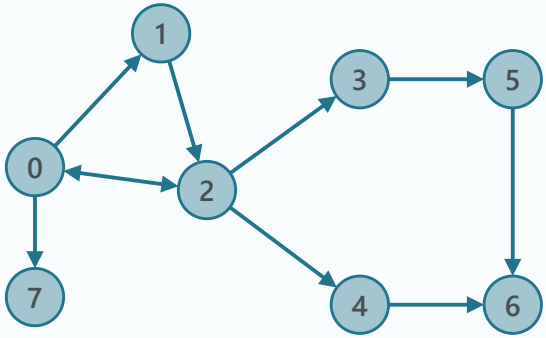
```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>
   ();
4    stack.push(s);
▶    while (!stack.isEmpty()) {
6      int v = stack.pop();
7      if (!visited[v]) {
8        visited[v] = true;
9        System.out.print(v + " ");
10       for (int w : adj[v])
11         stack.push(w);
12     }
13   }
14 }
```

VARIABLES

`v = –` `stack = []`

OUTPUT

```
0 7 2 4 6 3 5 1
```



stack (bottom → top)

The stack is empty and the loop ends. Output: **0 7 2 4 6 3 5 1**, unlike the recursive order (0 1 2 3 5 6 4 7), because neighbors come out in reverse.

DFS: Analysis and Uses

Complexity

- Each vertex is marked **once**: O(V).
- Each adjacency list is scanned once; the lists have total length E (2E if undirected): O(E).
- Total: **O(V + E)** with adjacency lists, **$O(V^2)$** with a matrix (a row of V entries per vertex).
- Extra memory: `visited[]` O(V) plus the stack: O(V) for recursion (the longest path); O(E) for the "mark on pop" version.

Where is it used?

- Reachability, counting **connected components**
- **Cycle detection**: an edge to a vertex on the stack (yellow) = a back edge
- **Topological sort**: reverse finishing order
- **Strongly connected components** (Tarjan, Kosaraju)
- Bridges and articulation points, solving mazes and puzzles, backtracking

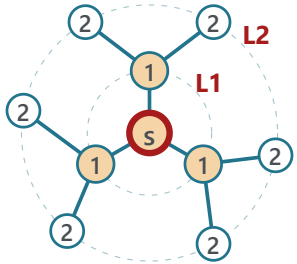
DFS does **not** find shortest paths: the first path it finds to a target may be a long detour. For the fewest edges you need BFS.

BFS: The Idea

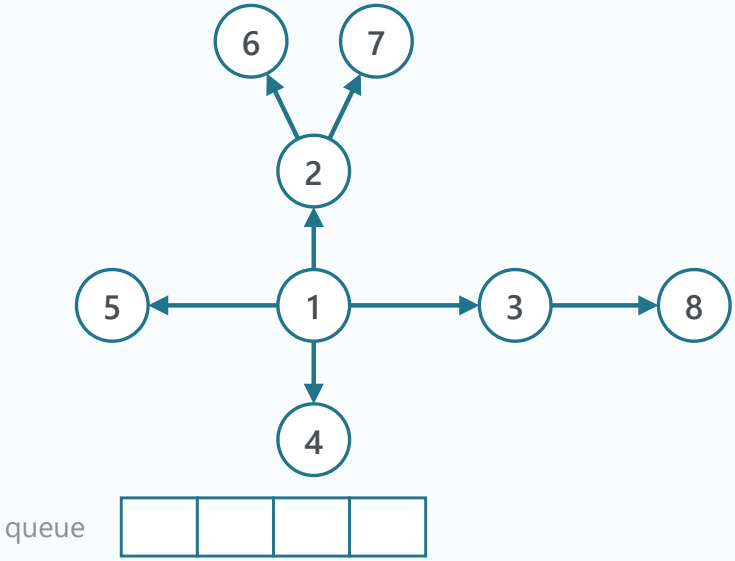
- Like ripples from a stone in water: first everything **1 edge** away, then everything **2 edges** away…
- It models news spreading: a person tells all their friends, who tell their own friends.
- The **queue** (FIFO) keeps the order: first discovered, first processed.
- A vertex is marked **when it is enqueued**, so each vertex enters the queue at most once.
- In an unweighted graph it finds the **path with the fewest edges** from the start to every vertex.

Levels

L0 = {s}
L1 = neighbors of s
L2 = new neighbors of L1
…
Every vertex in Lk is at distance exactly **k** from s.



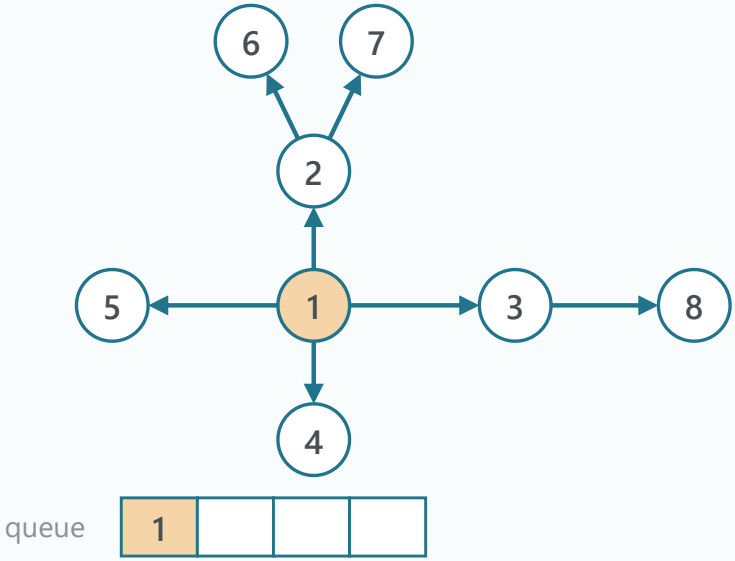
BFS Step by Step: Level by Level



processed: —

Start: all vertices are **white** (undiscovered). Yellow = discovered, waiting in the queue · dark = processed.

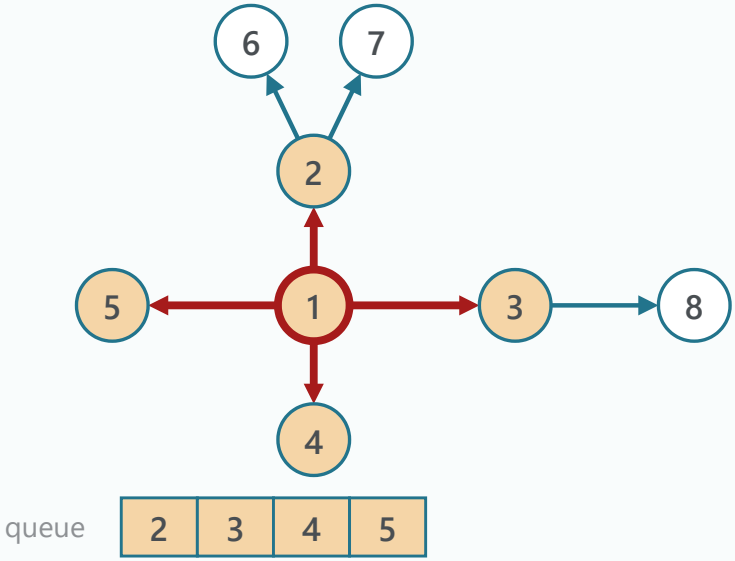
BFS Step by Step: Level by Level



processed: —

The start vertex 1 is marked **discovered** and enqueued.

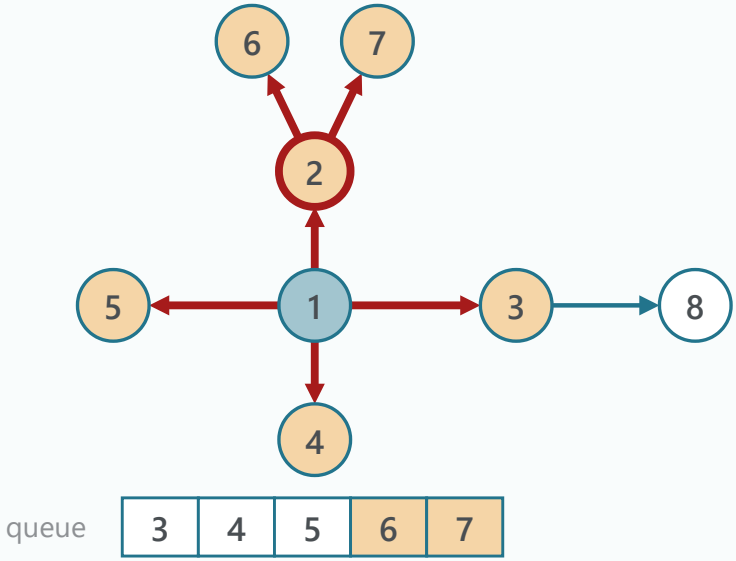
BFS Step by Step: Level by Level



processed: 1

Dequeue **1**; its undiscovered neighbors 2, 3, 4, 5 are marked and enqueued (level 1).

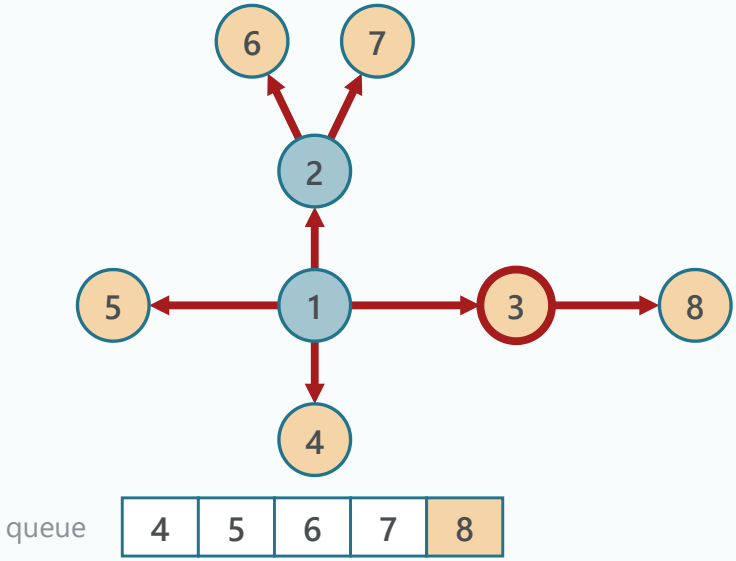
BFS Step by Step: Level by Level



processed: 1 2

Dequeue **2**; its undiscovered neighbors 6, 7 are marked and enqueued (level 2).

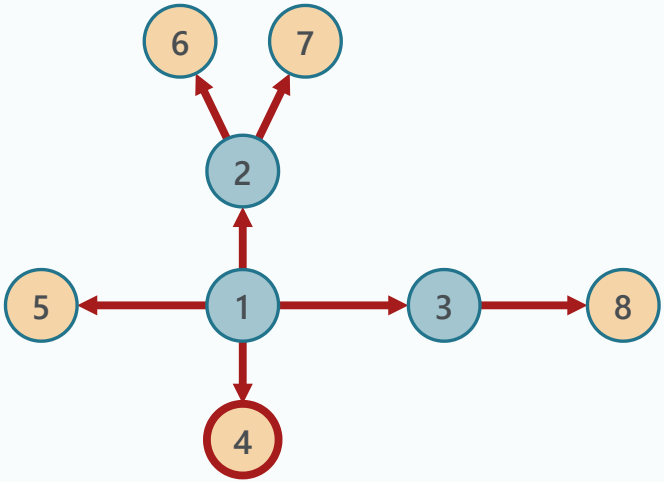
BFS Step by Step: Level by Level



processed: 1 2 3

Dequeue **3**; its undiscovered neighbor 8 is marked and enqueued (level 2).

BFS Step by Step: Level by Level

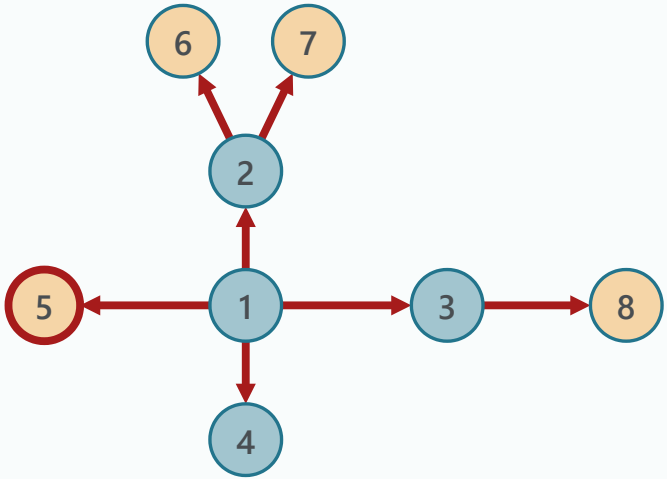


queue | 5 | 6 | 7 | 8 |

processed: 1 2 3 4

Dequeue **4**; no undiscovered neighbors.

BFS Step by Step: Level by Level



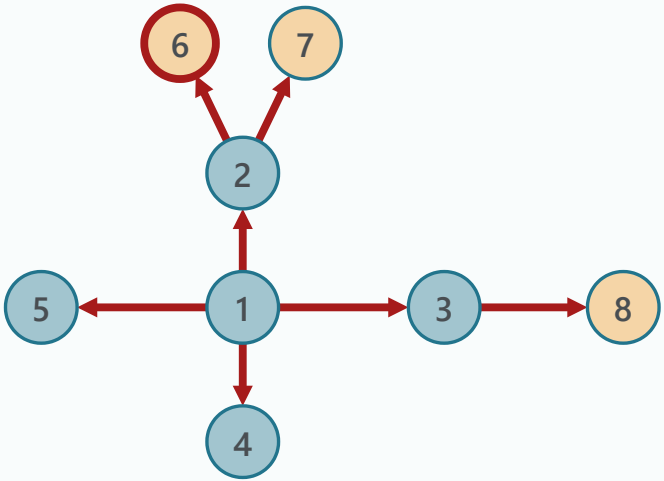
queue

6	7	8	

processed: 1 2 3 4 5

Dequeue **5**; no undiscovered neighbors.

BFS Step by Step: Level by Level

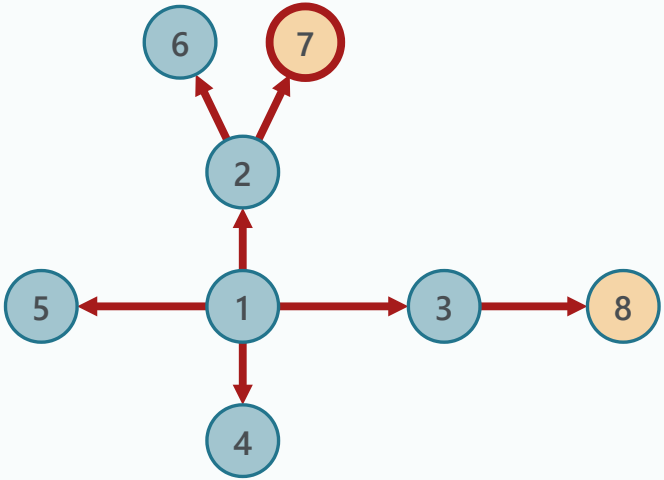


queue | 7 | 8 | | |

processed: 1 2 3 4 5 6

Dequeue **6**; no undiscovered neighbors.

BFS Step by Step: Level by Level



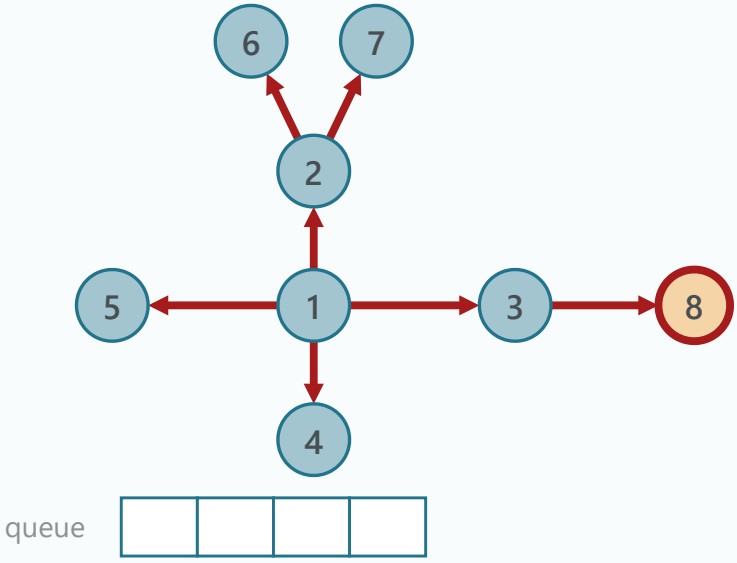
queue



processed: 1 2 3 4 5 6 7

Dequeue **7**; no undiscovered neighbors.

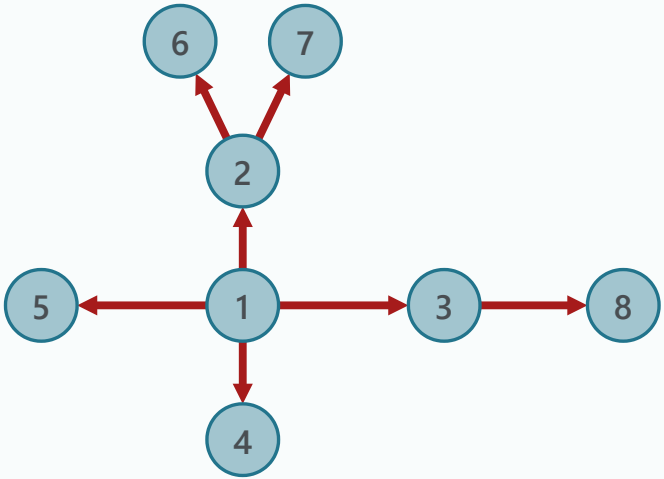
BFS Step by Step: Level by Level



processed: 1 2 3 4 5 6 7 8

Dequeue **8**; no undiscovered neighbors → the queue is empty and BFS ends.

BFS Step by Step: Level by Level



processed: 1 2 3 4 5 6 7 8

Result: visit order **1, 2, 3, 4, 5, 6, 7, 8**. First the vertices at distance 1 (2, 3, 4, 5), then those at distance 2 (6, 7, 8): **level by level**.

BFS: Pseudocode

Standard pseudocode

```
procedure BFS(G, root) is
  let Q be a queue
  label root as explored
  Q.enqueue(root)
  while Q is not empty do
    v := Q.dequeue()
    for all edges from v to w
        in G.adjacentEdges(v) do
      if w is not labeled as explored then
        label w as explored
        w.parent := v
        Q.enqueue(w)
```

Plain version

```
BFS(start):
  create an empty queue
  mark start as discovered
  enqueue start
  while the queue is not empty:
    current = dequeue
    for each neighbor of current
      if neighbor is undiscovered
        mark neighbor as discovered
        neighbor.parent = current
        enqueue neighbor
```

- The `w.parent := v` links form the BFS tree; following them from the target back to the root gives the **shortest path**.
- Mark a vertex when it is **enqueued**, not when it is dequeued; otherwise the same vertex enters the queue many times.

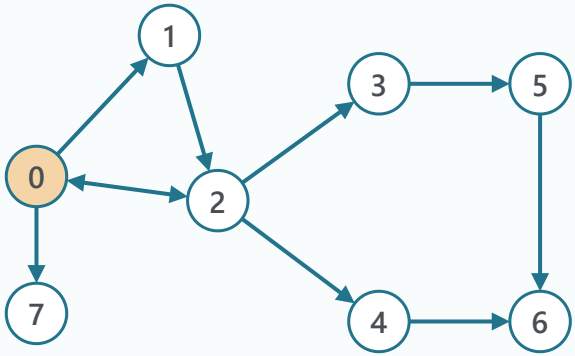
BFS: Tracing `bfs(0)`

```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
▶    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = –` `queue = [0]`

OUTPUT



queue (front → back)

0

Mark 0 and enqueue it. The queue holds only 0.

Step 1 / 16

BFS: Tracing `bfs(0)`

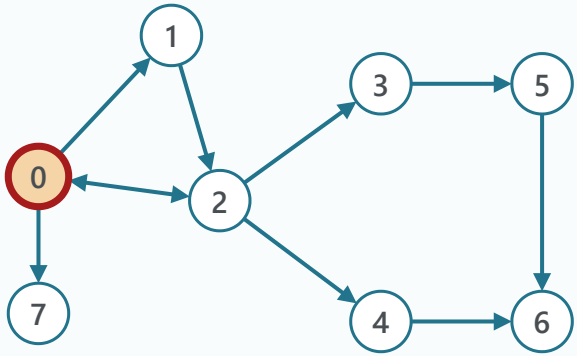
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 0` `queue = []`

OUTPUT

```
0
```



queue (front → back)

Dequeue **0** from the front and print it.

BFS: Tracing `bfs(0)`

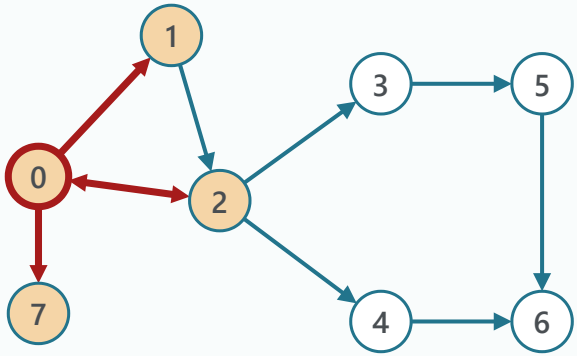
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (!visited[w]) {
▶          visited[w] = true;
▶          queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 0` `queue = [1, 2, 7]`

OUTPUT

```
0
```



queue (front → back)

1	2	7

Neighbors of 0: undiscovered 1, 2, 7 are marked and added to the **back** of the queue.

BFS: Tracing `bfs(0)`

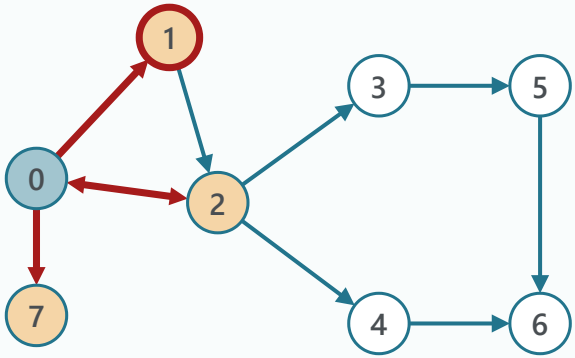
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 1` `queue = [2, 7]`

OUTPUT

```
0 1
```



queue (front → back)

2	7

Dequeue **1** from the front and print it.

BFS: Tracing `bfs(0)`

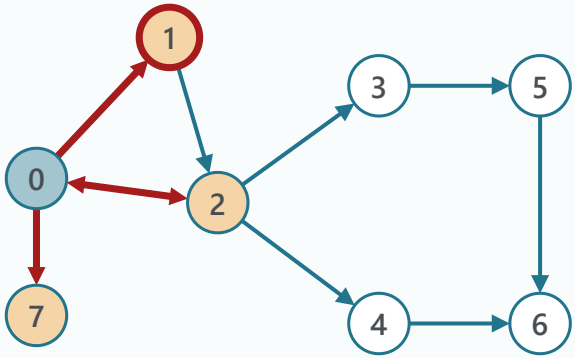
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 1` `queue = [2, 7]`

OUTPUT

```
0 1
```



queue (front → back)

2	7

Neighbors of 1: 2 is already marked → skip.

BFS: Tracing `bfs(0)`

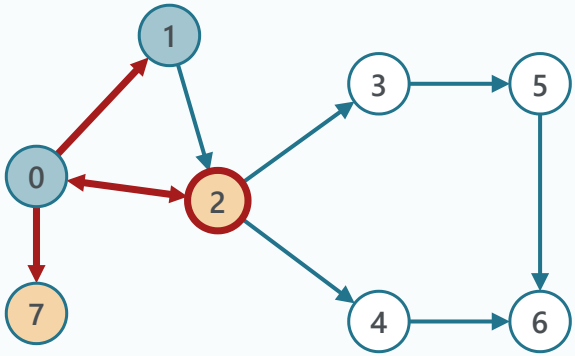
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 2` `queue = [7]`

OUTPUT

```
0 1 2
```



queue (front → back)

7

Dequeue **2** from the front and print it.

BFS: Tracing `bfs(0)`

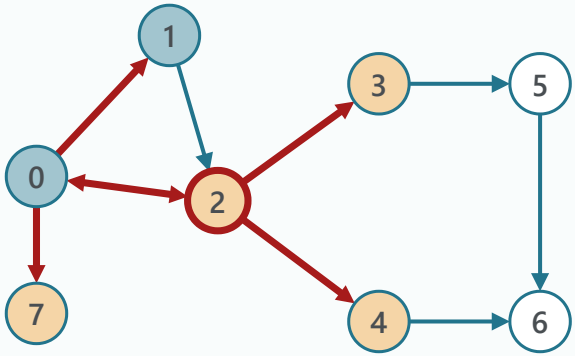
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (!visited[w]) {
▶          visited[w] = true;
▶          queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 2` `queue = [7, 3, 4]`

OUTPUT

```
0 1 2
```



queue (front → back)

7	3	4

Neighbors of 2: undiscovered 3, 4 are marked and added to the **back** of the queue; 0 is already marked → skip.

BFS: Tracing `bfs(0)`

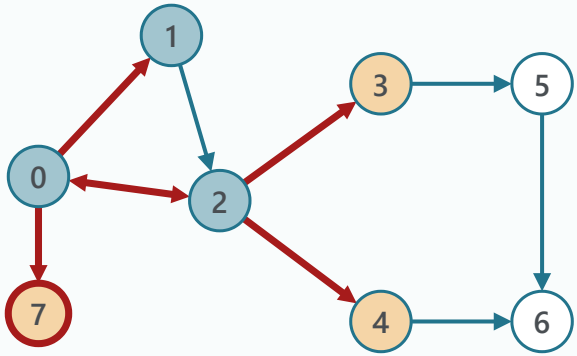
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 7` `queue = [3, 4]`

OUTPUT

```
0 1 2 7
```



queue (front → back)

3	4

Dequeue **7** from the front and print it.

BFS: Tracing `bfs(0)`

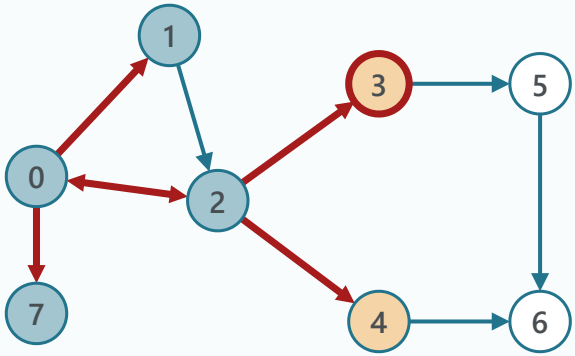
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 3` `queue = [4]`

OUTPUT

```
0 1 2 7 3
```



queue (front → back)

4

Dequeue **3** from the front and print it.

BFS: Tracing `bfs(0)`

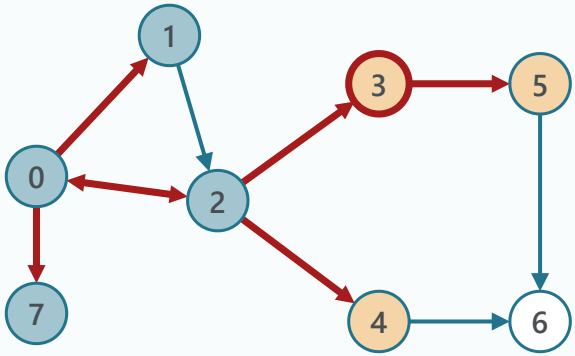
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (!visited[w]) {
▶          visited[w] = true;
▶          queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 3` `queue = [4, 5]`

OUTPUT

```
0 1 2 7 3
```



queue (front → back)

4	5

Neighbors of 3: undiscovered 5 is marked and added to the **back** of the queue.

BFS: Tracing `bfs(0)`

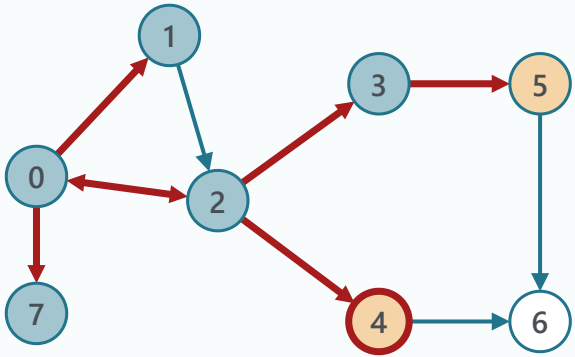
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 4` `queue = [5]`

OUTPUT

```
0 1 2 7 3 4
```



queue (front → back)

5

Dequeue **4** from the front and print it.

BFS: Tracing `bfs(0)`

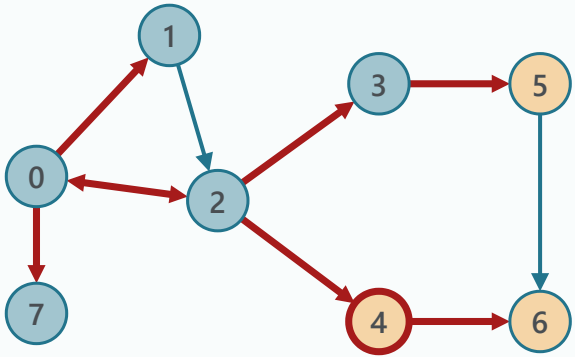
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (!visited[w]) {
▶          visited[w] = true;
▶          queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 4` `queue = [5, 6]`

OUTPUT

```
0 1 2 7 3 4
```



queue (front → back)

5	6

Neighbors of 4: undiscovered 6 is marked and added to the **back** of the queue.

BFS: Tracing `bfs(0)`

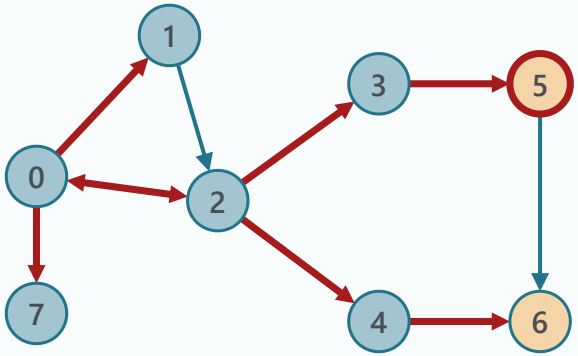
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 5` `queue = [6]`

OUTPUT

```
0 1 2 7 3 4 5
```



queue (front → back)

6

Dequeue **5** from the front and print it.

BFS: Tracing `bfs(0)`

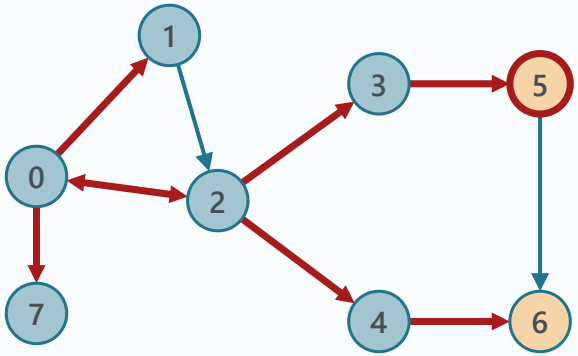
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 5` `queue = [6]`

OUTPUT

```
0 1 2 7 3 4 5
```



queue (front → back)

6

Neighbors of 5: 6 is already marked → skip.

BFS: Tracing `bfs(0)`

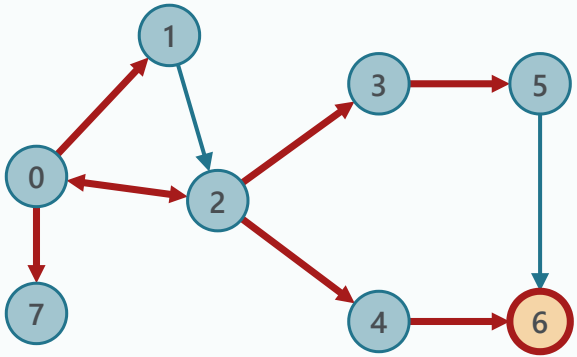
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
5    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = 6` `queue = []`

OUTPUT

```
0 1 2 7 3 4 5 6
```



queue (front → back)

Dequeue **6** from the front and print it.

BFS: Tracing `bfs(0)`

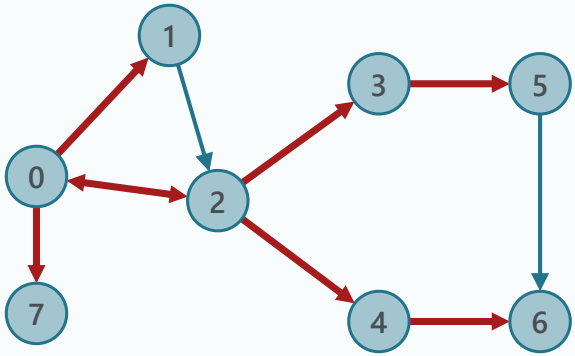
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> queue = new LinkedList<>();
4    visited[s] = true; queue.add(s);
▶    while (!queue.isEmpty()) {
6      int v = queue.poll();
7      System.out.print(v + " ");
8      for (int w : adj[v])
9        if (!visited[w]) {
10         visited[w] = true;
11         queue.add(w);
12       }
13   }
14 }
```

VARIABLES

`v = –` `queue = []`

OUTPUT

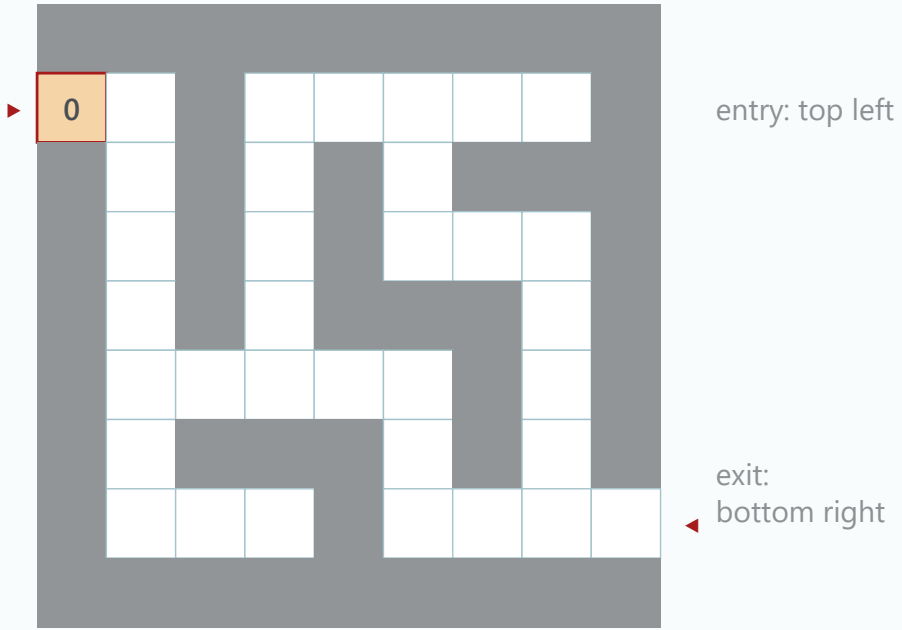
```
0 1 2 7 3 4 5 6
```



queue (front → back)

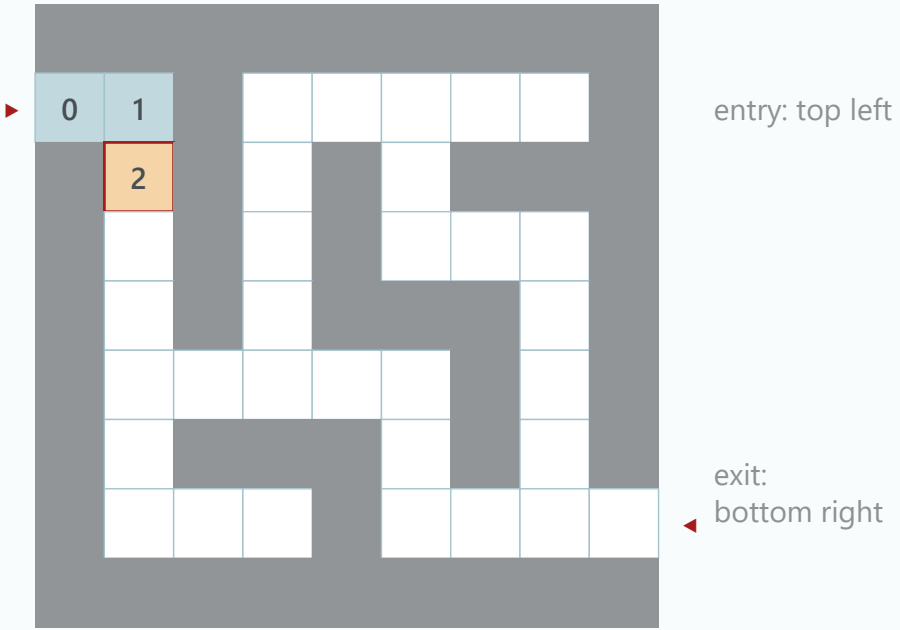
The queue is empty. Output: **0 1 2 7 3 4 5 6**, level by level: {0}, {1, 2, 7}, {3, 4}, {5, 6}. The bold edges form the BFS tree.

BFS in a Maze: Try Every Path at Once



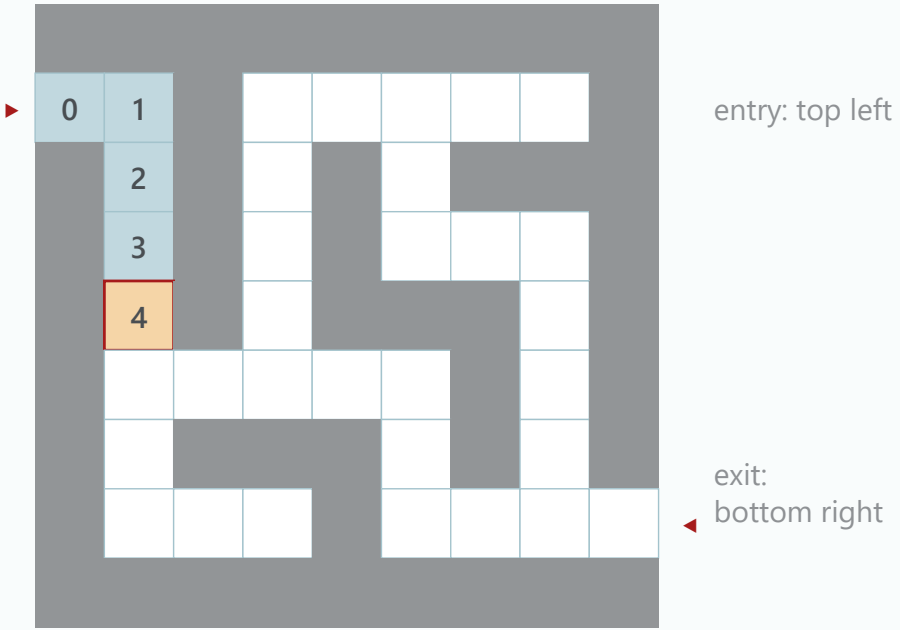
The entry cell is enqueued with distance 0. Walls (dark) are not vertices; every open cell is a vertex and adjacent open cells share an edge.

BFS in a Maze: Try Every Path at Once



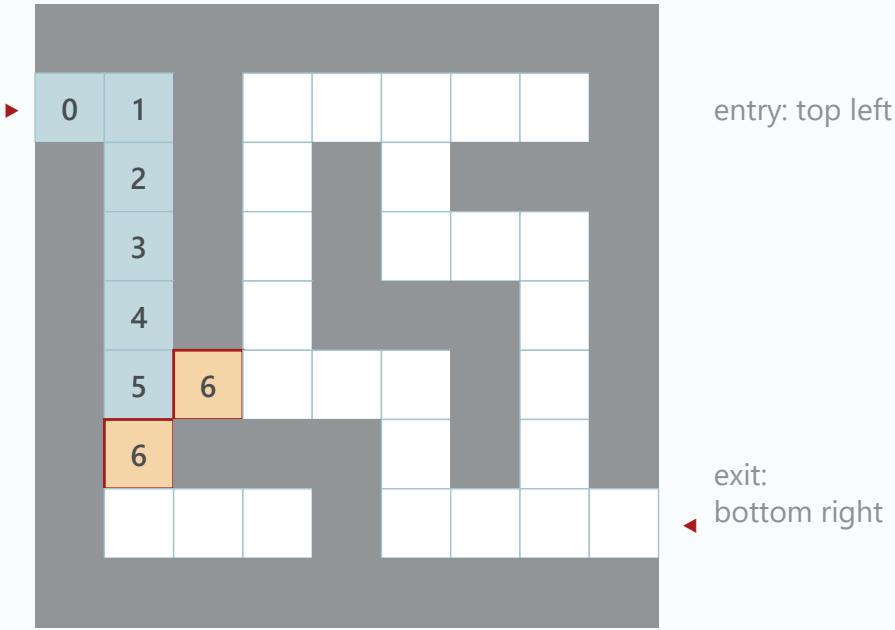
At each step the wave advances one cell in every direction: cells at distance 1 and 2. Yellow = the current **frontier**.

BFS in a Maze: Try Every Path at Once



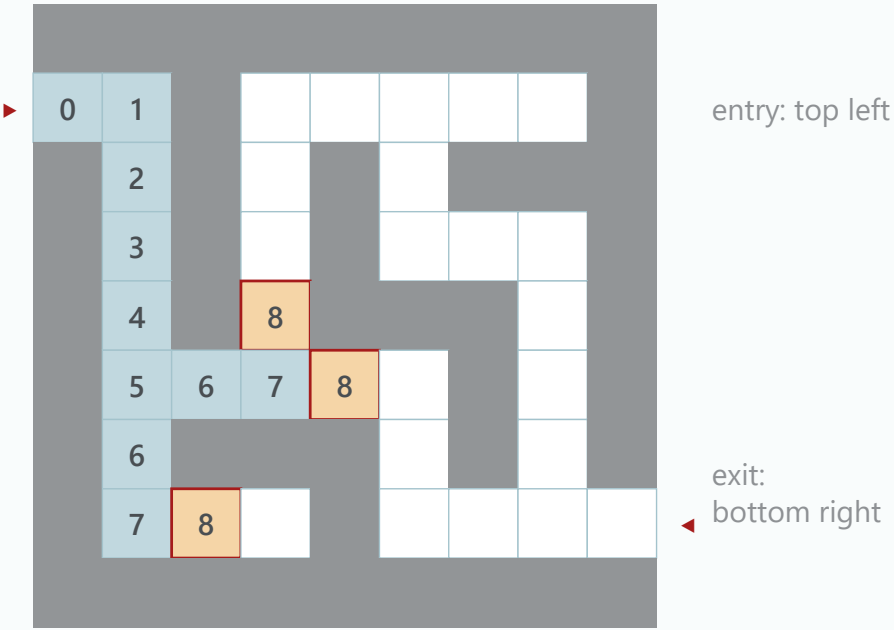
A single corridor: the wave moves down. The number in each cell is the **fewest steps** from the entry.

BFS in a Maze: Try Every Path at Once



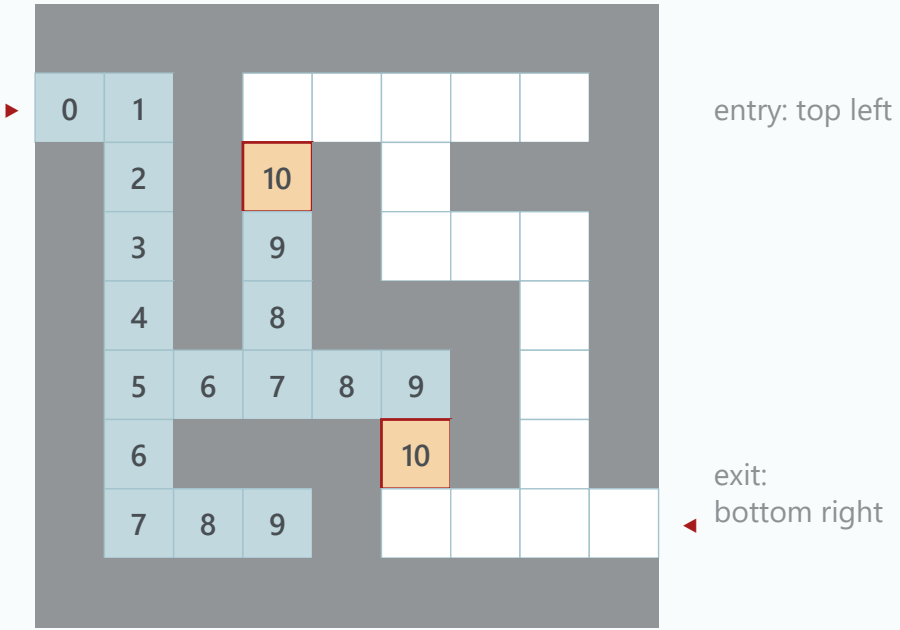
At distance 5 the path **forks**: the wave spreads right and down. BFS "tries every path at once".

BFS in a Maze: Try Every Path at Once



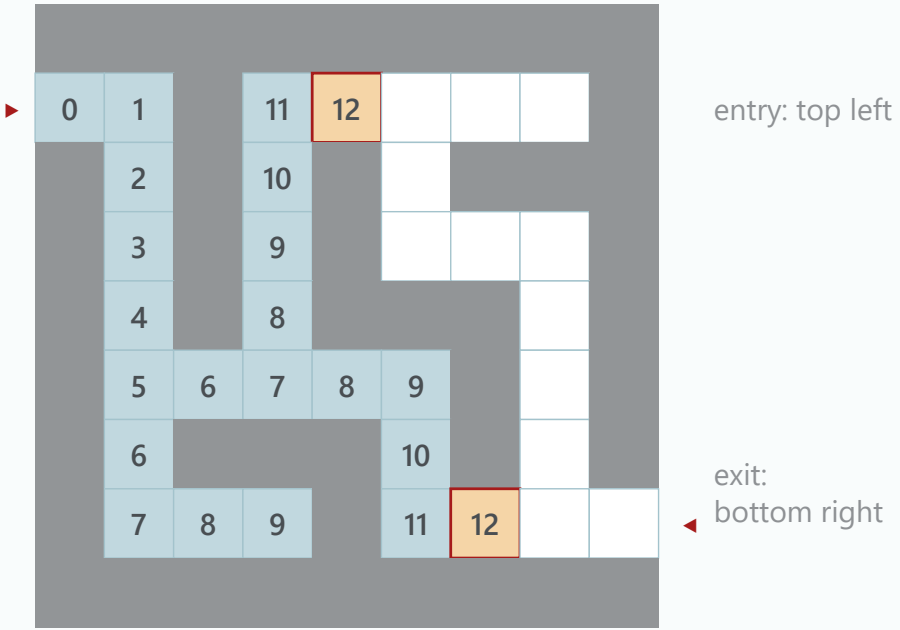
Both branches advance together: one toward the upper room, one downward.

BFS in a Maze: Try Every Path at Once



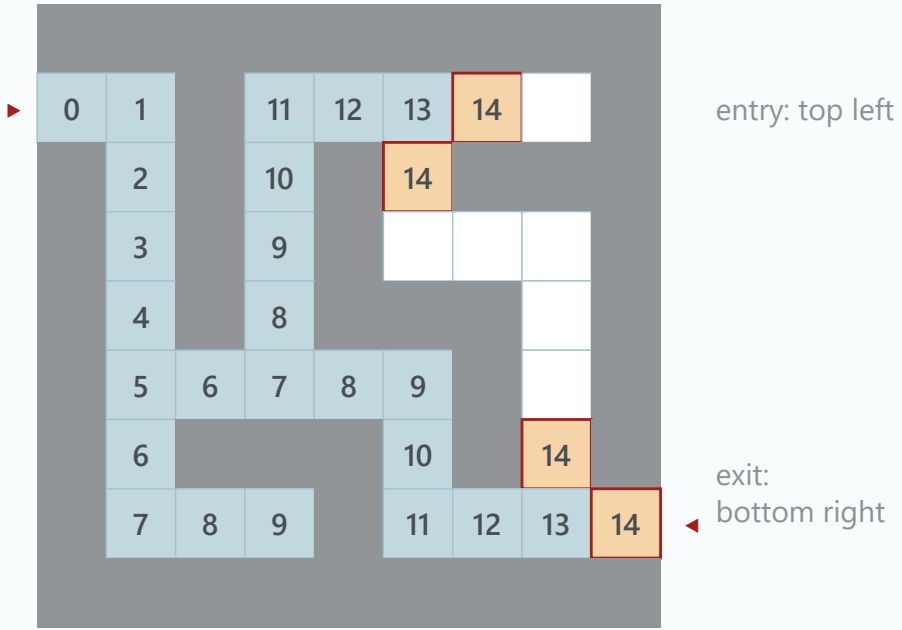
Distance 10: the wave has reached the upper corridor and, below, the bottom corridor.

BFS in a Maze: Try Every Path at Once



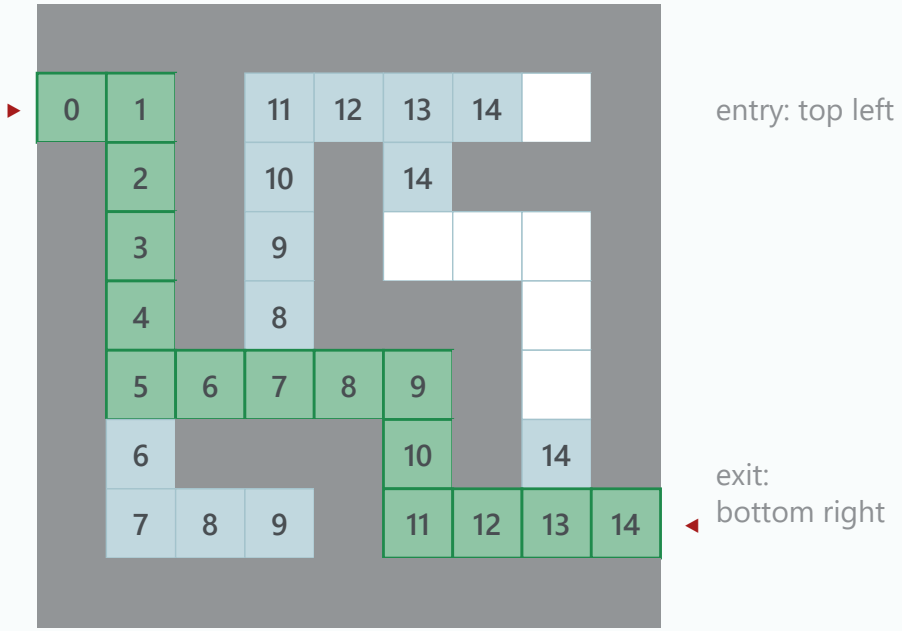
The bottom corridor nears the exit; the upper branch wanders into dead ends.

BFS in a Maze: Try Every Path at Once



The exit is reached in **14** steps. When BFS first reaches a cell, that distance is guaranteed to be the shortest.

BFS in a Maze: Try Every Path at Once



Following parent links back from the exit gives the **shortest path** (green, 15 cells, 14 steps). The remaining cells (distance 15–17) were not needed for this path.

BFS Tree and Shortest Paths

vertex	1	2	3	4	5	6	7	8
dist	0	1	1	1	1	2	2	2
parent	—	1	1	1	1	2	2	3

The "BFS Step by Step" graph with source 1. Path(8): 8 → parent 3 → parent 1; reversed, **1** → **3** → **8** (2 edges).

- On discovery: `dist[w] = dist[v] + 1;` `parent[w] = v;`
- Distances leave the queue in **nondecreasing** order (0, 1, 1, 1, 1, 2, 2, 2), so the first path to reach a vertex is a shortest one.

Rebuilding the path

```
List<Integer> path(int[] parent, int s, int t) {
  LinkedList<Integer> path = new LinkedList<>();
  for (int v = t; v != -1; v = parent[v])
    path.addFirst(v);   // walk back from t
  if (path.getFirst() != s)
    return null;        // t unreachable
  return path;
}
```

Valid only for **unweighted** graphs (or graphs with all weights equal). Weighted graphs need Dijkstra.

BFS: Filling `dist[]` and `parent[]`

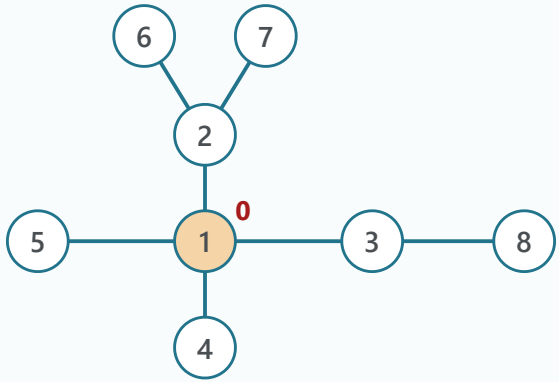
```
1  void bfsDist(int s) {
▶    Arrays.fill(dist, -1);
▶    Arrays.fill(parent, -1); dist[s] = 0;
▶    Queue<Integer> queue = new LinkedList<>();
▶    queue.add(s);
6    while (!queue.isEmpty()) {
7      int v = queue.poll();
8      for (int w : adj[v])
9        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = –` `queue = [1]`

`dist = [0, -1, -1, -1, -1, -1, -1, -1]`

`parent = [-1, -1, -1, -1, -1, -1, -1, -1]`





Undirected graph, source 1 (arrays shown with indices 1–8). dist[] = −1 means "undiscovered"; parent[] also starts at −1 (no parent). dist of 1 is 0, and only 1 is in the queue.

BFS: Filling `dist[]` and `parent[]`

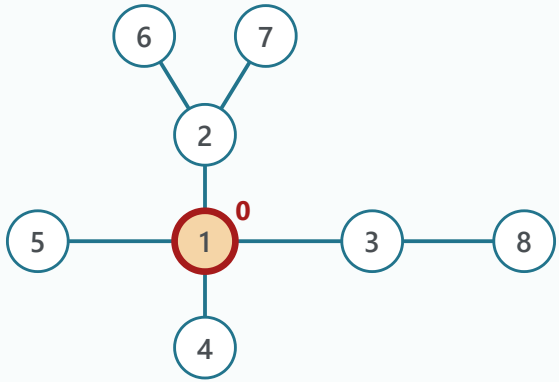
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
8      for (int w : adj[v])
9        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 1` `queue = []`

`dist = [0, -1, -1, -1, -1, -1, -1, -1]`

`parent = [-1, -1, -1, -1, -1, -1, -1, -1]`



queue number: distance

Dequeue 1 from the front; now examine its neighbors 2, 3, 4, 5 in order.

BFS: Filling `dist[]` and `parent[]`

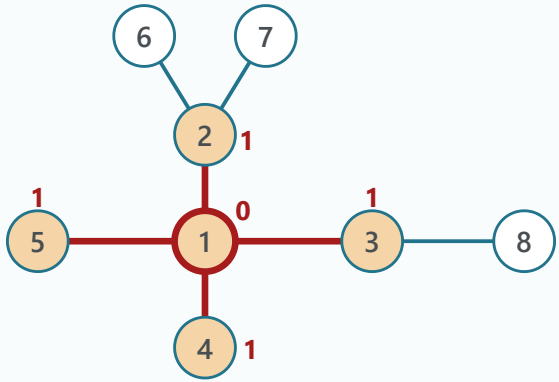
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
7      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
▶          dist[w] = dist[v] + 1;
▶          parent[w] = v;
▶          queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 1` `queue = [2, 3, 4, 5]`

`dist = [0, 1, 1, 1, 1, -1, -1, -1]`

`parent = [-1, 1, 1, 1, 1, -1, -1, -1]`



queue | 2 | 3 | 4 | 5 | | number: distance

All four neighbors are undiscovered: their distance becomes dist[1] + 1 = 1, their parent 1, and they are enqueued.

BFS: Filling `dist[]` and `parent[]`

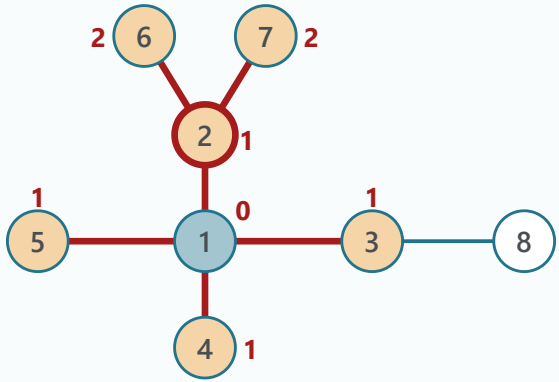
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
▶          dist[w] = dist[v] + 1;
▶          parent[w] = v;
▶          queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 2` `queue = [3, 4, 5, 6, 7]`

`dist = [0, 1, 1, 1, 1, 2, 2, -1]`

`parent = [-1, 1, 1, 1, 1, 2, 2, -1]`



queue | 3 | 4 | 5 | 6 | 7 | number: distance

Dequeue 2. Neighbor 1 is already discovered, skip it; 6 and 7 are discovered: distance 1 + 1 = 2, parent 2.

BFS: Filling `dist[]` and `parent[]`

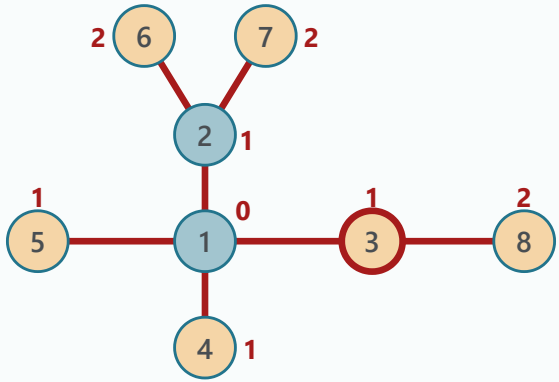
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
▶          dist[w] = dist[v] + 1;
▶          parent[w] = v;
▶          queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 3` `queue = [4, 5, 6, 7, 8]`

`dist = [0, 1, 1, 1, 1, 2, 2, 2]`

`parent = [-1, 1, 1, 1, 1, 2, 2, 3]`



queue | 4 | 5 | 6 | 7 | 8 | number: distance

Dequeue 3. Neighbor 1 is already discovered, skip it; 8 is discovered: distance 1 + 1 = 2, parent 3.

BFS: Filling `dist[]` and `parent[]`

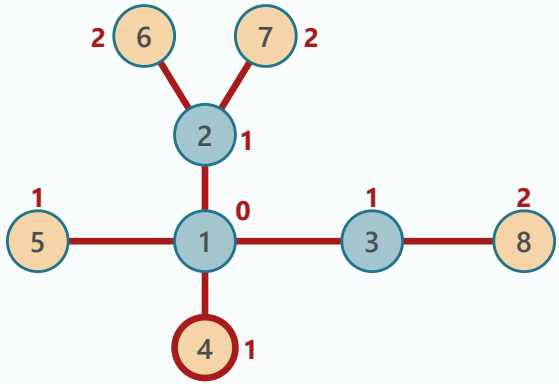
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 4` `queue = [5, 6, 7, 8]`

`dist = [0, 1, 1, 1, 1, 2, 2, 2]`

`parent = [-1, 1, 1, 1, 1, 2, 2, 3]`



queue | 5 | 6 | 7 | 8 | | number: distance

Dequeue 4 (distance 1). Its only neighbor, 1, is already discovered: nothing new is enqueued.

BFS: Filling `dist[]` and `parent[]`

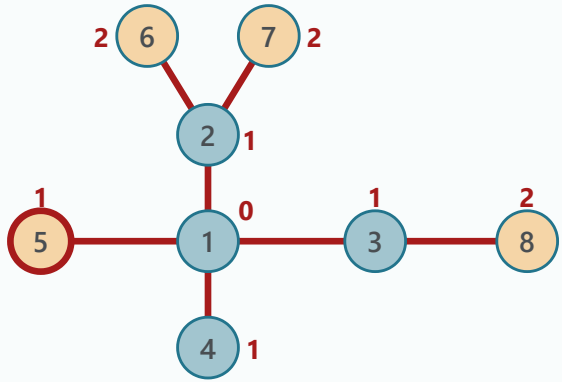
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 5` `queue = [6, 7, 8]`

`dist = [0, 1, 1, 1, 1, 2, 2, 2]`

`parent = [-1, 1, 1, 1, 1, 2, 2, 3]`



queue | 6 | 7 | 8 | | | number: distance

Dequeue 5 (distance 1). Its only neighbor, 1, is already discovered: nothing new is enqueued.

BFS: Filling `dist[]` and `parent[]`

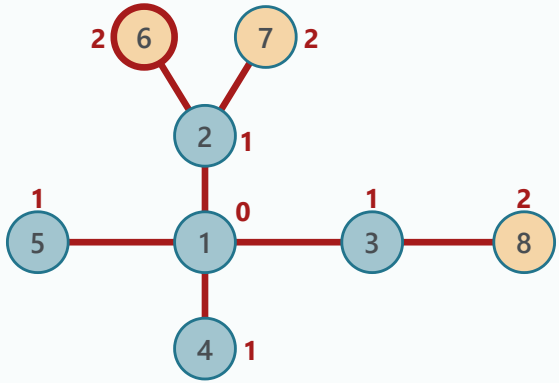
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 6` `queue = [7, 8]`

`dist = [0, 1, 1, 1, 1, 2, 2, 2]`

`parent = [-1, 1, 1, 1, 1, 2, 2, 3]`



queue | 7 | 8 | | | | number: distance

Dequeue 6 (distance 2). Its only neighbor, 2, is already discovered: nothing new is enqueued.

BFS: Filling `dist[]` and `parent[]`

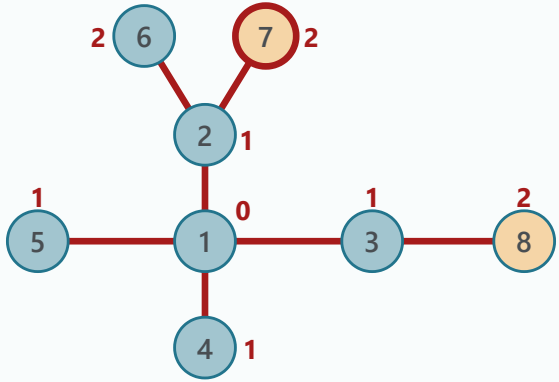
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 7` `queue = [8]`

`dist = [0, 1, 1, 1, 1, 2, 2, 2]`

`parent = [-1, 1, 1, 1, 1, 2, 2, 3]`



queue | 8 | | | | | number: distance

Dequeue 7 (distance 2). Its only neighbor, 2, is already discovered: nothing new is enqueued.

BFS: Filling `dist[]` and `parent[]`

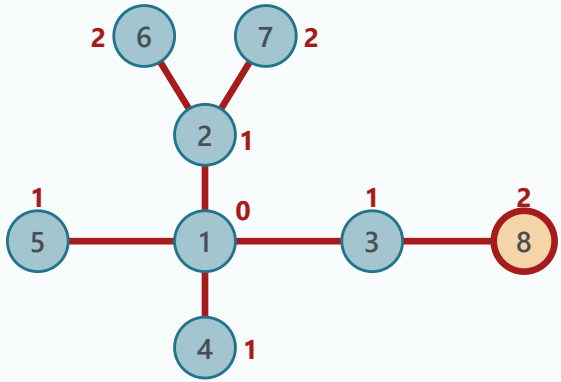
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
6    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = 8` `queue = []`

`dist = [0, 1, 1, 1, 1, 2, 2, 2]`

`parent = [-1, 1, 1, 1, 1, 2, 2, 3]`



queue | | | | | | number: distance

Dequeue 8 (distance 2). Its only neighbor, 3, is already discovered: nothing new is enqueued.

BFS: Filling `dist[]` and `parent[]`

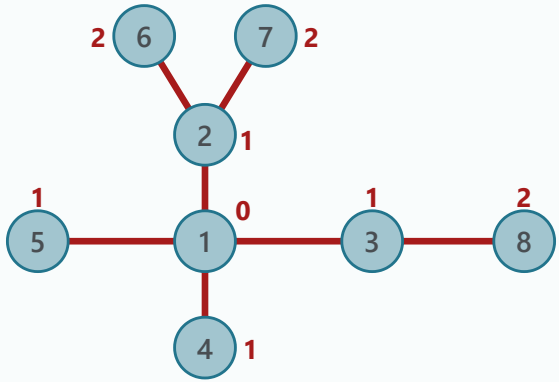
```
1  void bfsDist(int s) {
2    Arrays.fill(dist, -1);
3    Arrays.fill(parent, -1); dist[s] = 0;
4    Queue<Integer> queue = new LinkedList<>();
5    queue.add(s);
▶    while (!queue.isEmpty()) {
7      int v = queue.poll();
8      for (int w : adj[v])
9        if (dist[w] == -1) {  // new
10         dist[w] = dist[v] + 1;
11         parent[w] = v;
12         queue.add(w);
13       }
14   }
15 }
```

VARIABLES

`v = –` `queue = []`

`dist = [0, 1, 1, 1, 1, 2, 2, 2]`

`parent = [-1, 1, 1, 1, 1, 2, 2, 3]`



queue [ ] [ ] [ ] [ ] [ ] number: distance

The queue is empty. The bold edges (parent links) form the BFS tree; e.g. 8 → parent 3 → parent 1: the shortest path is 1 → 3 → 8, of length dist[8] = 2.

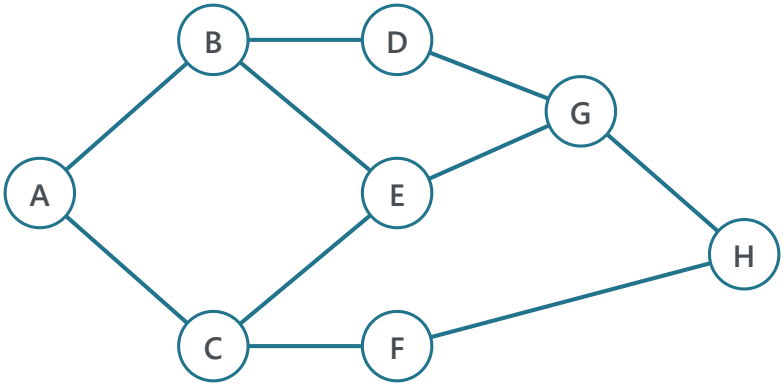
BFS vs. DFS

	DFS	BFS
Helper structure	Stack (LIFO) / recursion	Queue (FIFO)
Search pattern	one branch to its end, then backtrack	level by level (in order of distance)
Time	$O(V + E)$ list · $O(V^2)$ matrix	$O(V + E)$ list · $O(V^2)$ matrix
Extra memory	$O(V)$: the longest path is on the stack	$O(V)$: the widest level is in the queue
Shortest paths	no	yes (unweighted graphs)
Tree produced	deep, narrow DFS tree	shallow, wide BFS tree
Typical uses	cycles, components, topological order, SCCs, backtracking	fewest steps, levels, bipartiteness test, spreading

On the same graph (8-vertex directed graph, source 0): DFS **0 1 2 3 5 6 4 7**, BFS **0 1 2 7 3 4 5 6**.

Try It: BFS and DFS

Start A BFS DFS step run to end reset

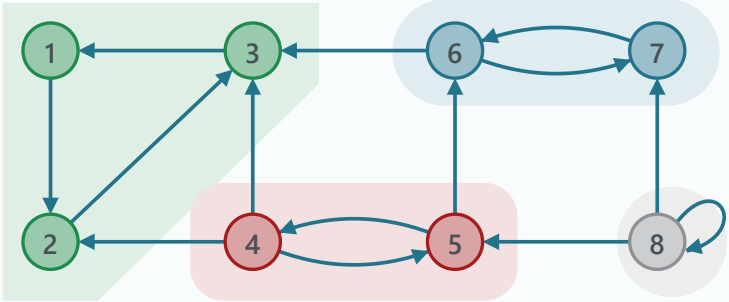


Pick a start vertex, press **BFS** or **DFS**, then advance with **step**.

Neighbors in alphabetical order; DFS in recursive order. Yellow: queued / on stack · orange ring: current · dark: done.

Strongly Connected Components

- In a directed graph, u and v are **mutually reachable** if there is a path from u to v **and** from v to u.
- A **strongly connected component** (SCC) is a **maximal** set of vertices in which every vertex can reach every other.
- Every vertex belongs to exactly one SCC; a single vertex can be an SCC on its own ({8}).
- "A vertex plus everything reachable from it" is **not** an SCC: 8 reaches every other vertex, yet none of them reaches 8.
- Tarjan's algorithm (1972) finds all SCCs with **a single DFS**.



SCCs: {1, 2, 3}, {4, 5}, {6, 7}, {8}

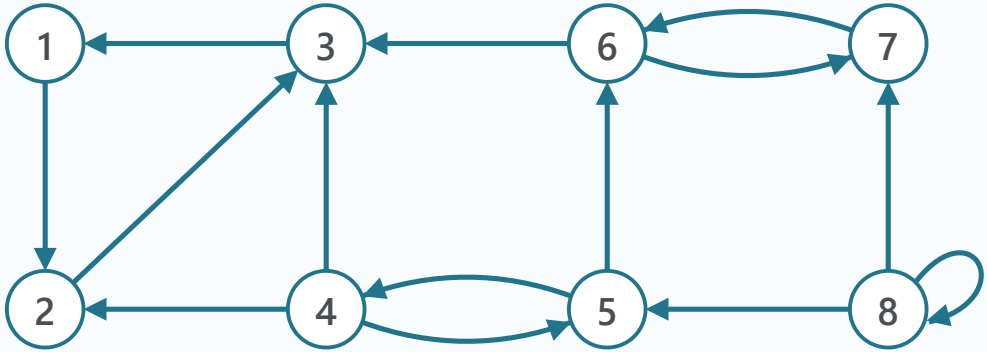
Shrinking each SCC to one vertex gives the **component graph**, which is acyclic (a DAG): communities in social networks, mutually dependent modules in compilers, solving 2-SAT.

Tarjan: Principles and Steps

1. **It is DFS-based**: try the vertices in turn and start a DFS from each unvisited one.
2. During the DFS each visited vertex gets an **index**: its **discovery order** (1, 2, 3, ...). Vertices on the current DFS path whose SCC is not yet known are kept on a **stack**.
3. For each vertex we compute its **lowlink**: the smallest index reachable from its DFS subtree using at most one edge to a vertex **still on the stack**.
   - Tree edge v → w: on return, `lowlink(v) = min(lowlink(v), lowlink(w))`
   - Edge to a w on the stack: `lowlink(v) = min(lowlink(v), index(w))`
   - Edge to a w in a finished SCC: ignored.
4. When v is finished, if **lowlink(v) = index(v)**, v is the **root** of an SCC: popping the stack down to v (inclusive) yields that SCC.

If lowlink(v) < index(v), v's subtree can reach an older stack vertex; v is in that vertex's SCC and is not a root.

Tarjan Step by Step: index, lowlink

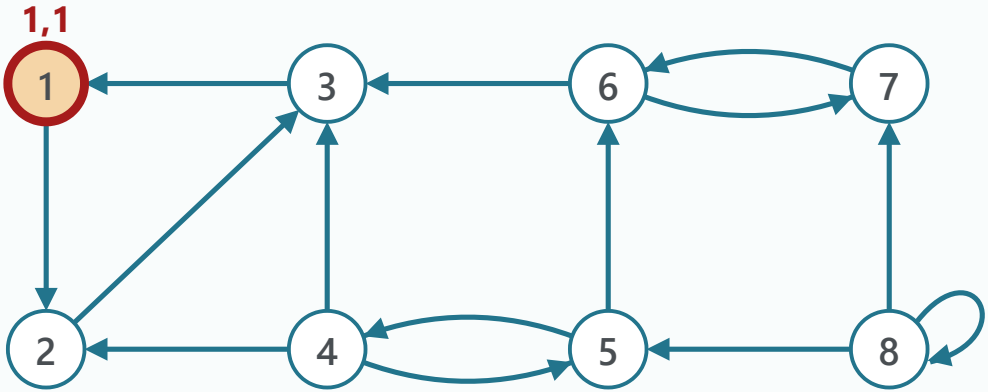


stack

SCCs: —

Directed graph with 8 vertices. Next to each vertex we write **index,lowlink**. White = unvisited, yellow = on the stack, colored = SCC found.

Tarjan Step by Step: index, lowlink

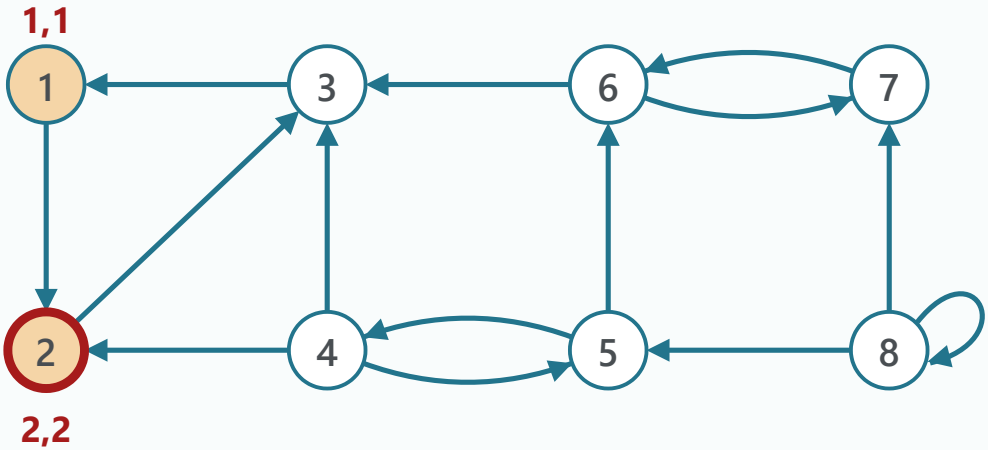


stack 1

SCCs: —

`SCC(1)` : index = lowlink = **1**; push 1.

Tarjan Step by Step: index, lowlink

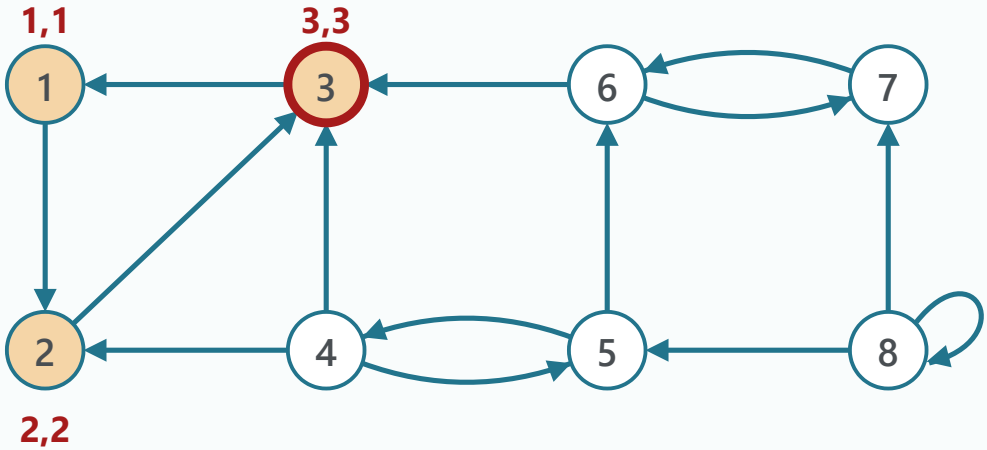


stack | 1 | 2 |

SCCs: —

`SCC(2)` : index = lowlink = **2**; push 2.

Tarjan Step by Step: index, lowlink

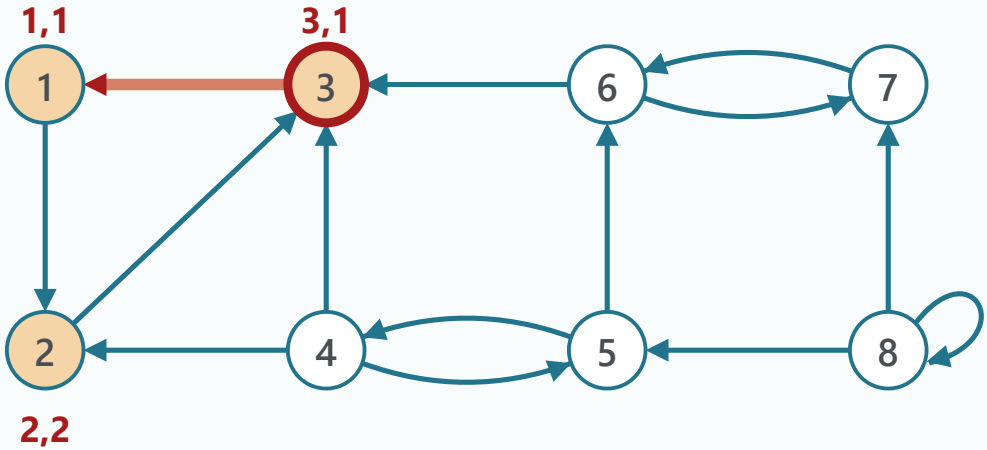


stack | 1 | 2 | 3 |

SCCs: —

`SCC(3)` : index = lowlink = **3**; push 3.

Tarjan Step by Step: index, lowlink

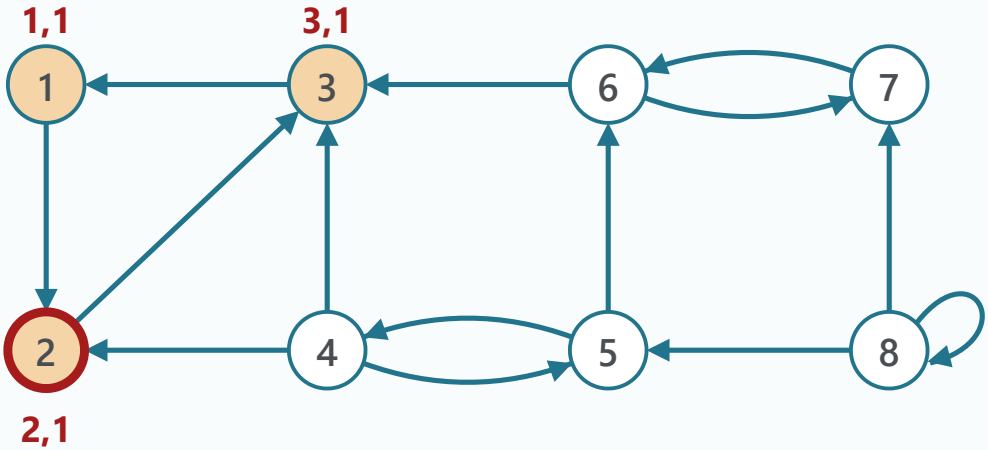


stack | 1 | 2 | 3 |

SCCs: —

Edge 3 → 1: 1 is **on the stack** → lowlink(3) = min(3, index(1) = 1) = **1**.

Tarjan Step by Step: index, lowlink

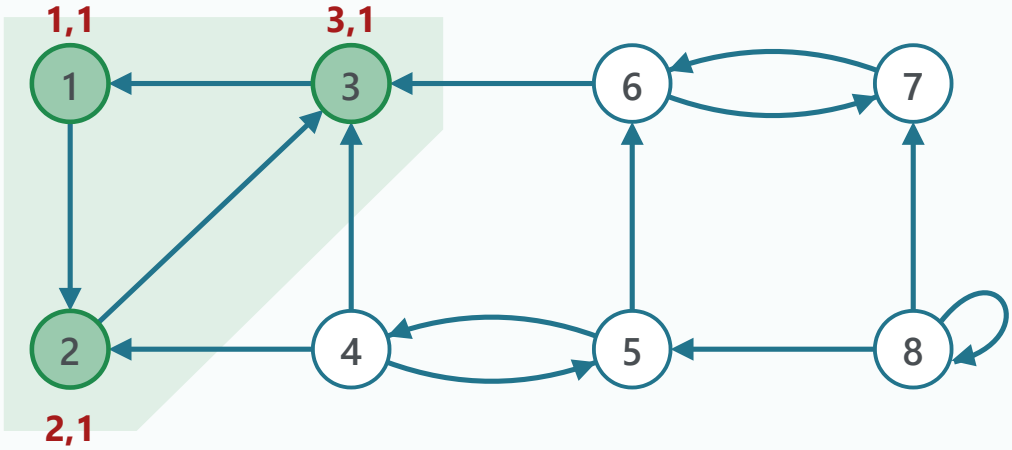


stack | 1 | 2 | 3 |

SCCs: —

3 is finished; lowlink 1 < index 3 → not a root, it stays on the stack. Back at 2: lowlink(2) = min(2, lowlink(3) = 1) = **1**.

Tarjan Step by Step: index, lowlink

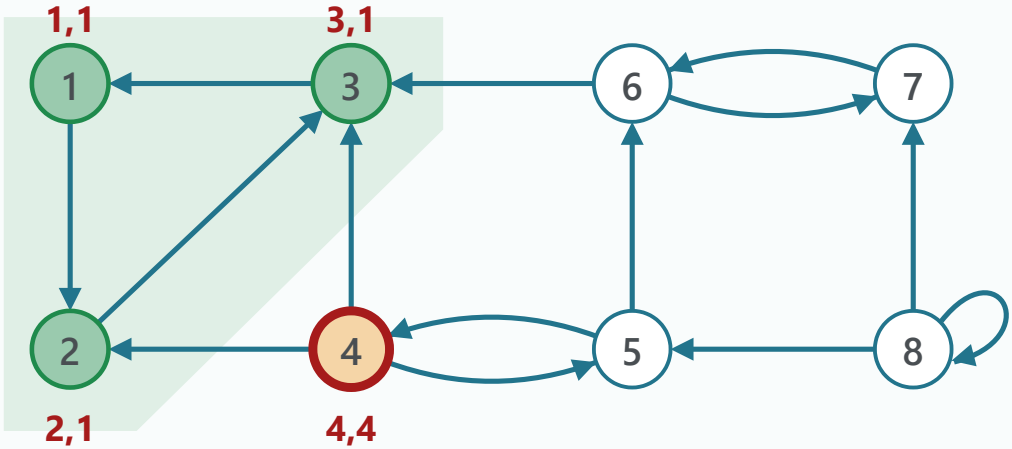


stack

SCCs: {1, 2, 3}

2 is finished; lowlink 1 < index 2 → not a root, it stays on the stack. Back at 1: lowlink(1) = 1 is unchanged. At 1, lowlink = index = 1 → 1 **is a root**: popping down to 1 gives an SCC: **{1, 2, 3}**.

Tarjan Step by Step: index, lowlink

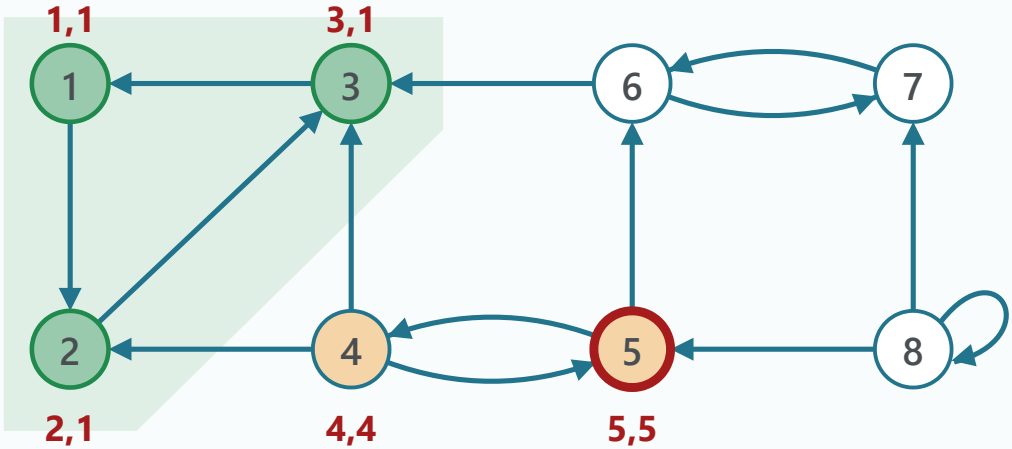


stack [4]

SCCs: {1, 2, 3}

`SCC(4)` : index = lowlink = **4**; push 4.

Tarjan Step by Step: index, lowlink

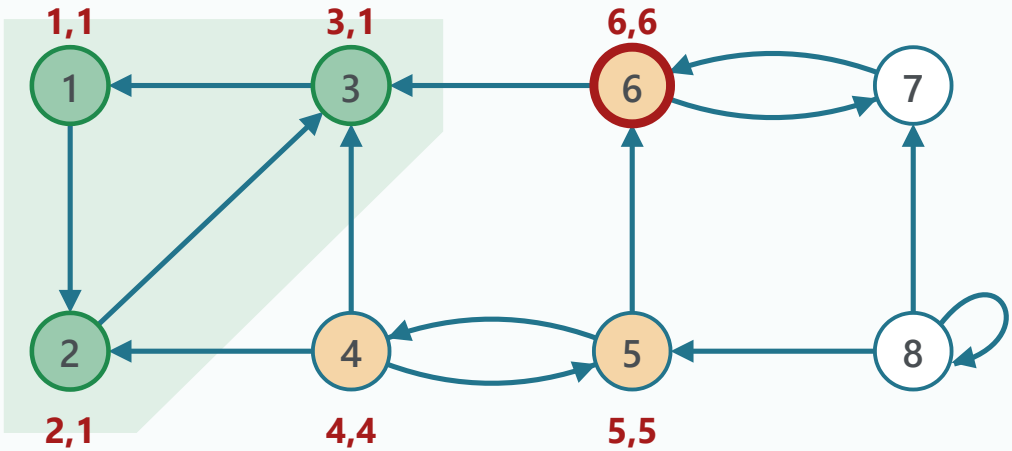


stack | 4 | 5 |

SCCs: {1, 2, 3}

4 → 2 is ignored (2 is in a finished SCC, not on the stack). 4 → 3 is ignored (3 is in a finished SCC, not on the stack). `SCC(5)` : index = lowlink = **5**; push 5.

Tarjan Step by Step: index, lowlink

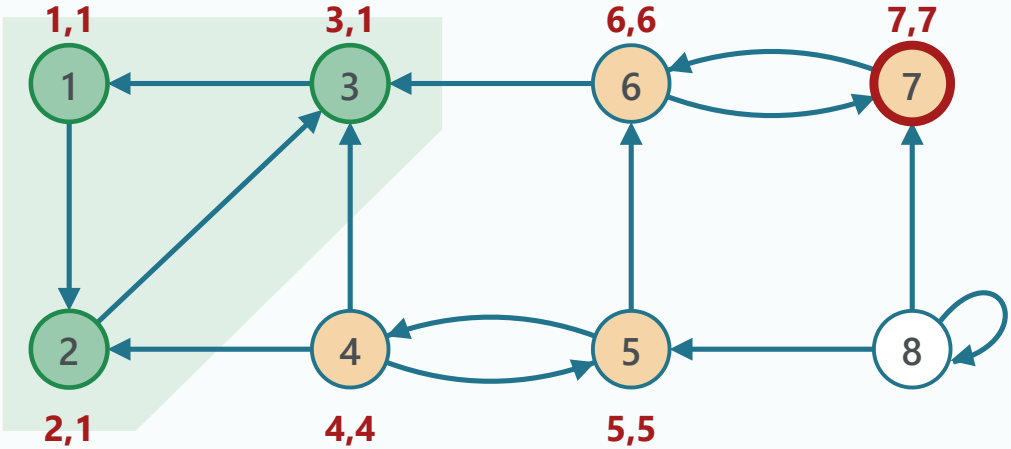


stack | 4 | 5 | 6 |

SCCs: {1, 2, 3}

`SCC(6)` : index = lowlink = **6**; push 6.

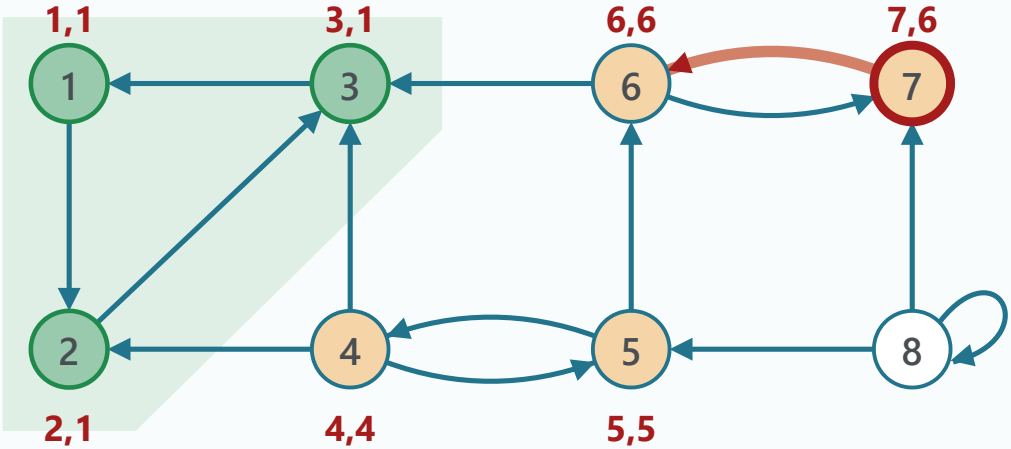
Tarjan Step by Step: index, lowlink



stack | 4 | 5 | 6 | 7 | SCCs: {1, 2, 3}

6 → 3 is ignored (3 is in a finished SCC, not on the stack). `SCC(7)` : index = lowlink = **7**; push 7.

Tarjan Step by Step: index, lowlink

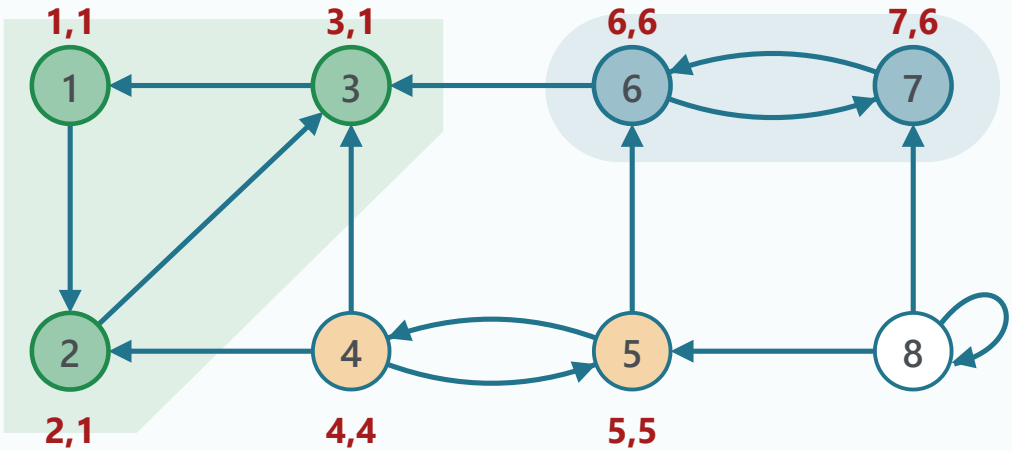


stack | 4 | 5 | 6 | 7 |

SCCs: {1, 2, 3}

Edge 7 → 6: 6 is **on the stack** → lowlink(7) = min(7, index(6) = 6) = **6**.

Tarjan Step by Step: index, lowlink

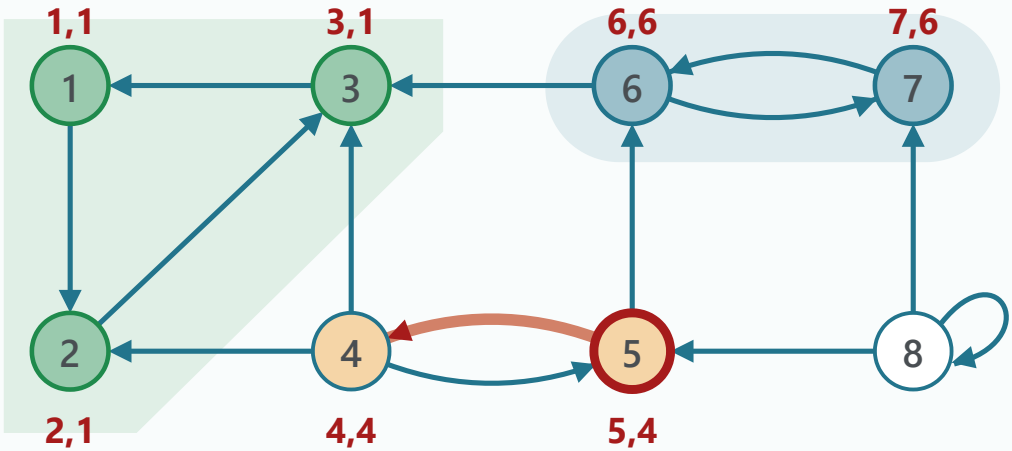


stack | 4 | 5 |

SCCs: {1, 2, 3} · {6, 7}

7 is finished; lowlink 6 < index 7 → not a root, it stays on the stack. Back at 6: lowlink(6) = 6 is unchanged. At 6, lowlink = index = 6 → 6 **is a root**: popping down to 6 gives an SCC: **{6, 7}**.

Tarjan Step by Step: index, lowlink

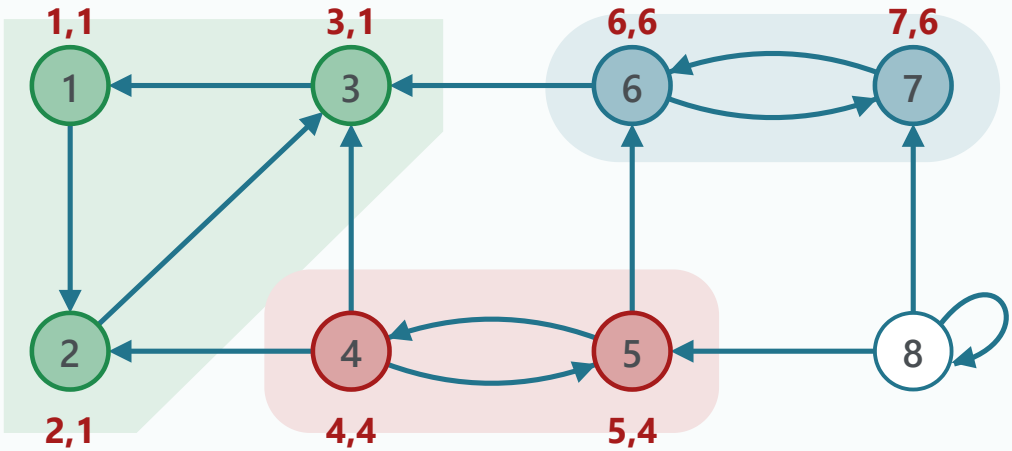


stack | 4 | 5 |

SCCs: {1, 2, 3} · {6, 7}

Back at 5: lowlink(5) = 5 is unchanged. Edge 5 → 4: 4 is **on the stack** → lowlink(5) = min(5, index(4) = 4) = **4**.

Tarjan Step by Step: index, lowlink

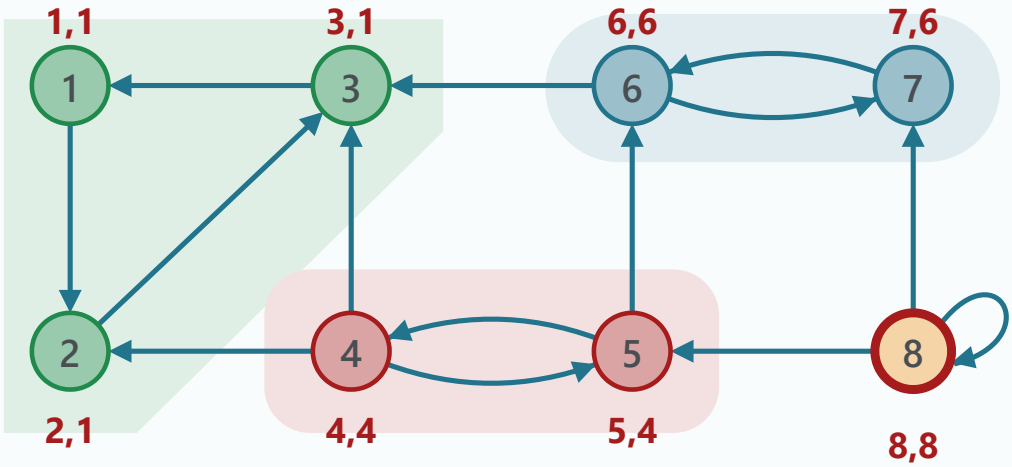


stack

SCCs: {1, 2, 3} · {6, 7} · {4, 5}

5 is finished; lowlink 4 < index 5 → not a root, it stays on the stack. Back at 4: lowlink(4) = 4 is unchanged. At 4, lowlink = index = 4 → 4 **is a root**: popping down to 4 gives an SCC: **{4, 5}**.

Tarjan Step by Step: index, lowlink

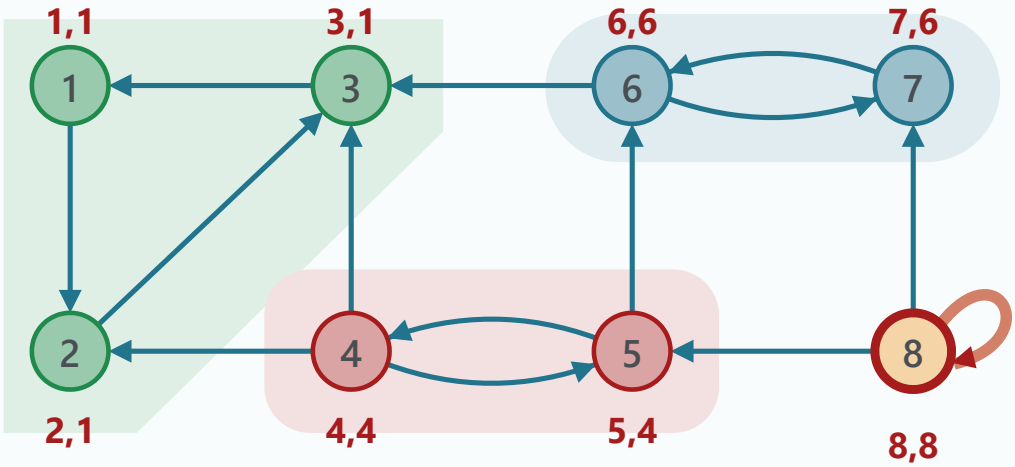


stack 8

SCCs: {1, 2, 3} · {6, 7} · {4, 5}

`SCC(8)` : index = lowlink = **8**; push 8.

Tarjan Step by Step: index, lowlink

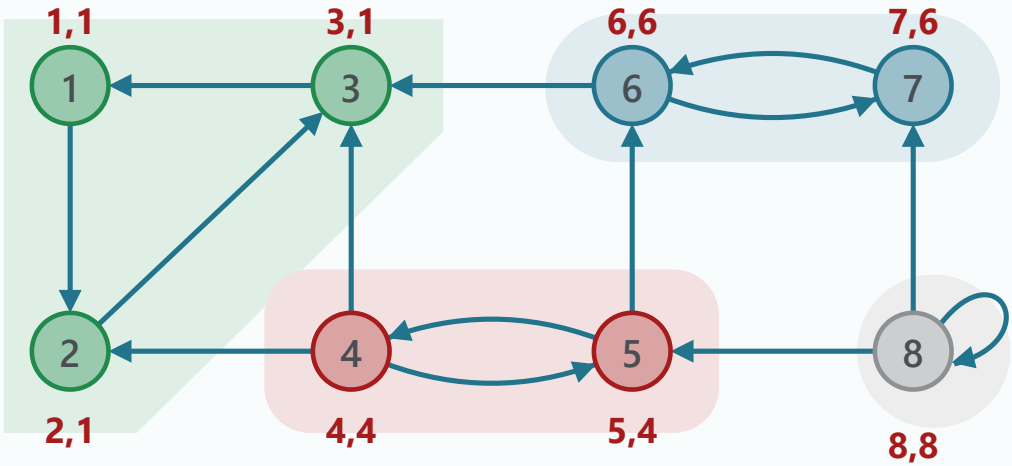


stack [8]

SCCs: {1, 2, 3} · {6, 7} · {4, 5}

8 → 5 is ignored (5 is in a finished SCC, not on the stack). 8 → 7 is ignored (7 is in a finished SCC, not on the stack). Self-loop 8 → 8: 8 is on the stack, but lowlink = min(8, 8) = 8 stays the same.

Tarjan Step by Step: index, lowlink

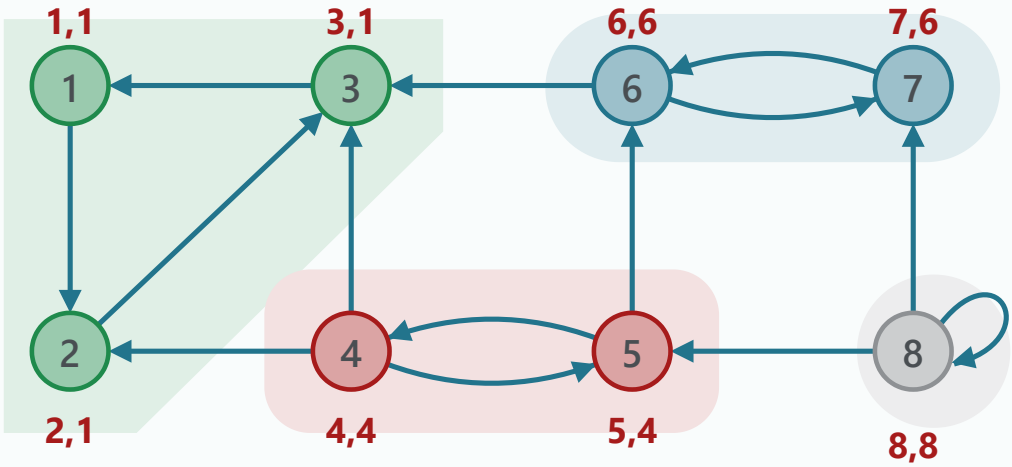


stack

SCCs: {1, 2, 3} · {6, 7} · {4, 5} · {8}

At 8, lowlink = index = 8 → 8 **is a root**: popping down to 8 gives an SCC: **{8}**.

Tarjan Step by Step: index, lowlink



stack empty

SCCs: {1, 2, 3} · {6, 7} · {4, 5} · {8}

Result: **4 SCCs**. Each SCC was found the moment its root (the vertex with index = lowlink: 1, 6, 4, 8) finished; they appear in reverse topological order.

Tarjan: Pseudocode

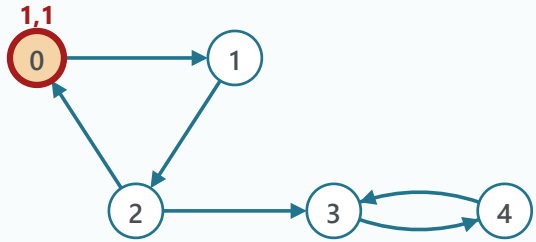
```
TARJAN_SCC(G):
  index = 1
  stack = [ ]
  SCCs = [ ]
  for each v in V:
    if v.index is undefined:
      SCC(v)
  return SCCs
```

```
SCC(v):
  v.index = v.lowlink = index++
  stack.push(v)
  for each (v, w) in E:
    if w.index is undefined:
      SCC(w)
      v.lowlink = min(v.lowlink, w.lowlink)
    else if w is on the stack:
      v.lowlink = min(v.lowlink, w.index)
  if v.lowlink == v.index:
    scc = [ ]
    repeat until w == v:  // at least once
      w = stack.pop()
      scc.add(w)
    SCCs.add(scc)
```

- In the figure the index starts at 1 (`index = 1`); starting at 0 gives the same SCCs.
- "Is it on the stack?" is answered in O(1) time by an `onStack[]` array kept alongside the stack.

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



stack `0`

Graph: 0→1, 1→2, 2→0, 2→3, 3→4, 4→3;
`index[w] == 0` means unvisited.
`scc(0)`: index and low = 1; push 0.

Step 1 / 20

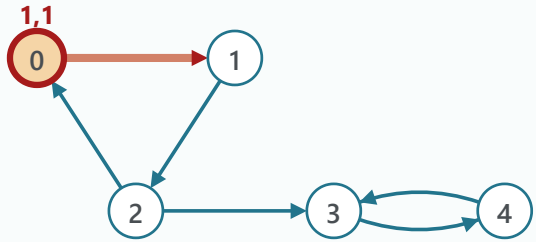
VARIABLES

`v = 0` `w = –` `low = [1, 0, 0, 0, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
▶      if (index[w] == 0) {
▶        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



stack [0]

0 → 1: 1 is unvisited → recursive call.

Step 2 / 20

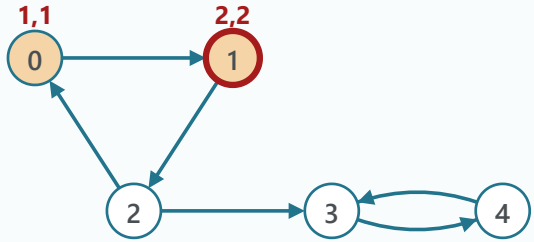
VARIABLES

`v = 0` `w = 1` `low = [1, 0, 0, 0, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
▶    index[v] = low[v] = ++counter;
▶    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



stack | 0 | 1 |

`scc(1)` : index and low = 2; push 1.

Step 3 / 20

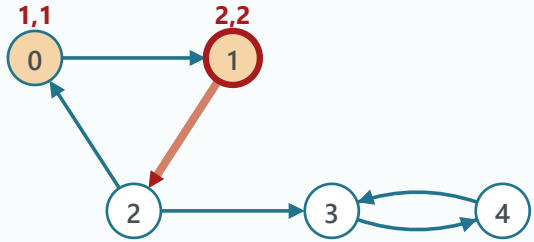
VARIABLES

`v = 1` `w = –` `low = [1, 2, 0, 0, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
▶      if (index[w] == 0) {
▶        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



stack | 0 | 1 |

1 → 2: 2 is unvisited → recursive call.

Step 4 / 20

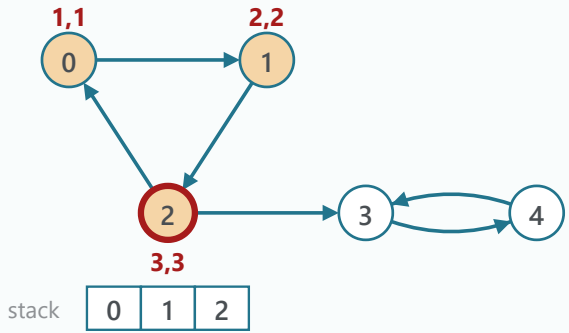
VARIABLES

`v = 1` `w = 2` `low = [1, 2, 0, 0, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



`scc(2)` : index and low = 3; push 2.

Step 5 / 20

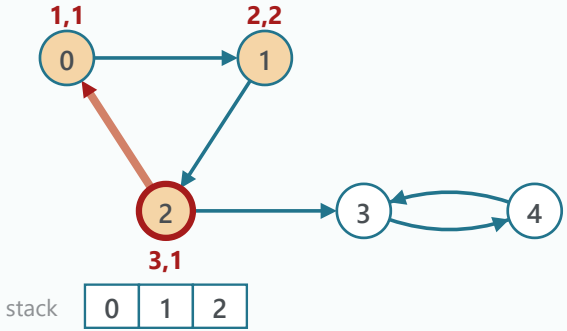
VARIABLES

`v = 2` `w = –` `low = [1, 2, 3, 0, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
▶      } else if (onStack[w])
▶        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



2 → 0: 0 is **on the stack** (back edge) → low[2] = min(3, index[0] = 1) = **1**.

Step 6 / 20

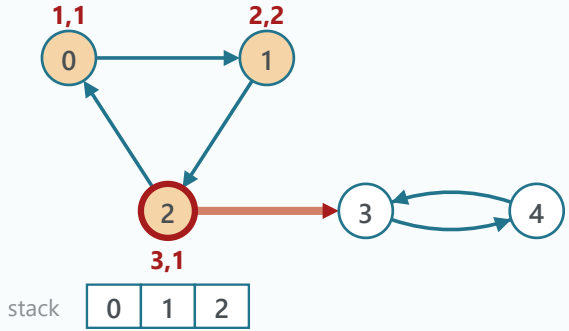
VARIABLES

`v = 2` `w = 0` `low = [1, 2, 1, 0, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
▶      if (index[w] == 0) {
▶        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



2 → 3: 3 is unvisited → recursive call.

Step 7 / 20

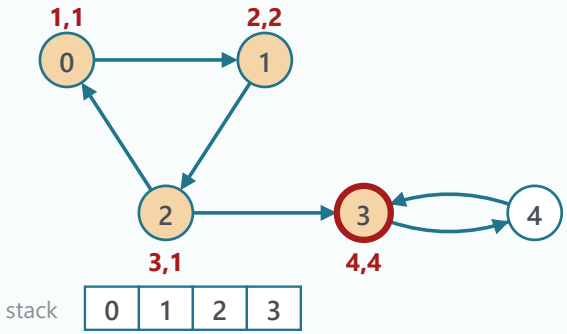
VARIABLES

`v = 2` `w = 3` `low = [1, 2, 1, 0, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
▶    index[v] = low[v] = ++counter;
▶    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



`scc(3)` : index and low = 4; push 3.

Step 8 / 20

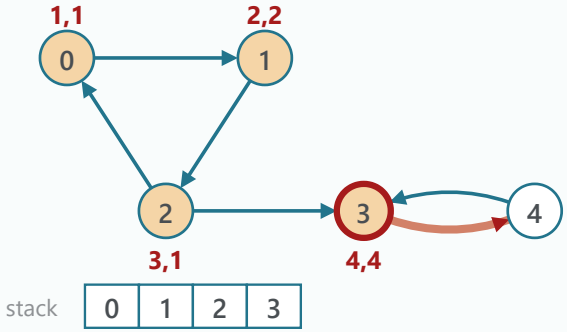
VARIABLES

`v = 3` `w = –` `low = [1, 2, 1, 4, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
▶      if (index[w] == 0) {
▶        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



3 → 4: 4 is unvisited → recursive call.

Step 9 / 20

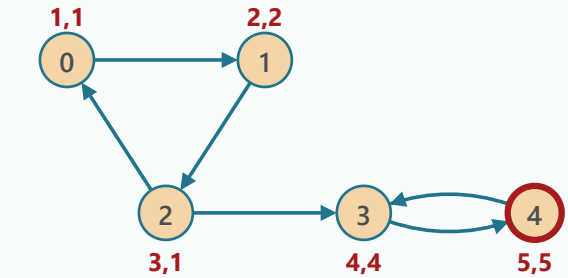
VARIABLES

`v = 3` `w = 4` `low = [1, 2, 1, 4, 0]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
▶    index[v] = low[v] = ++counter;
▶    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



`scc(4)` : index and low = 5; push 4.

Step 10 / 20

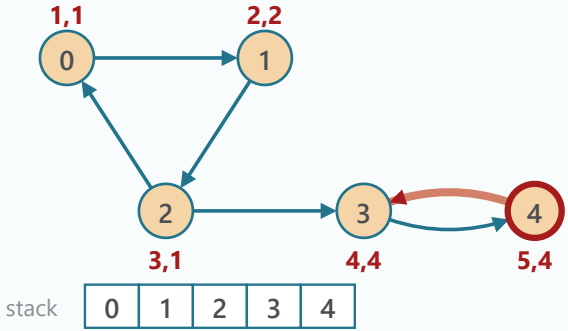
VARIABLES

`v = 4` `w = –` `low = [1, 2, 1, 4, 5]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
▶      } else if (onStack[w])
▶        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



4 → 3: 3 is **on the stack** (back edge) → low[4] = min(5, index[3] = 4) = **4**.

Step 11 / 20

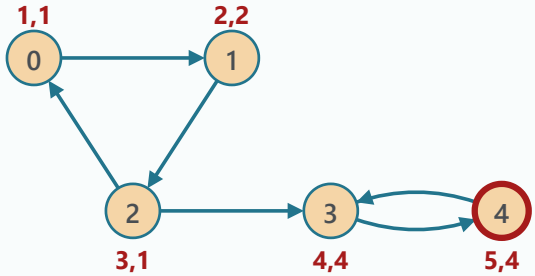
VARIABLES

`v = 4` `w = 3` `low = [1, 2, 1, 4, 4]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
▶    if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



low[4] = 4 ≠ index[4] = 5 → 4 is not a root and stays on the stack; return.

Step 12 / 20

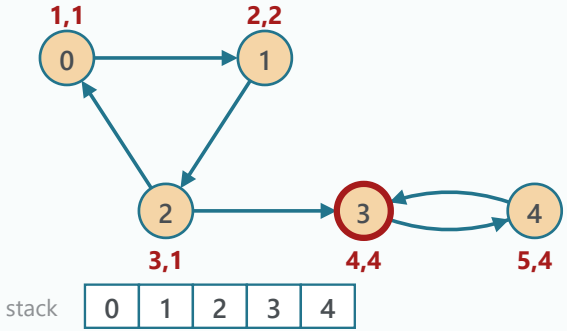
VARIABLES

`v = 4` `w = –` `low = [1, 2, 1, 4, 4]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
▶        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



`scc(4)` returned: low[3] = min(4, low[4] = 4) = **4**.

Step 13 / 20

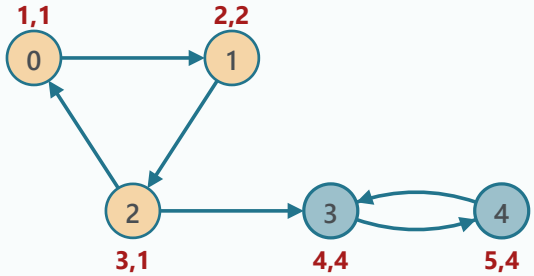
VARIABLES

`v = 3` `w = 4` `low = [1, 2, 1, 4, 4]`

OUTPUT

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
▶    if (low[v] == index[v]) {
▶      int w;
▶      do {
▶        w = stack.pop(); onStack[w] = false;
▶        System.out.print(w + " ");
▶      } while (w != v);
▶      System.out.println();
18   }
19 }
```



low[3] == index[3] = 4 → 3 is a root; pop and print 4, 3: SCC **{3, 4}**.

Step 14 / 20

VARIABLES

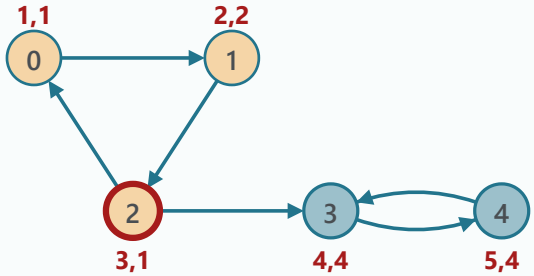
`v = 3` `w = –` `low = [1, 2, 1, 4, 4]`

OUTPUT

```
4 3
```

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
▶        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



`scc(3)` returned: low[2] = min(1, low[3] = 4) = **1**.

Step 15 / 20

VARIABLES

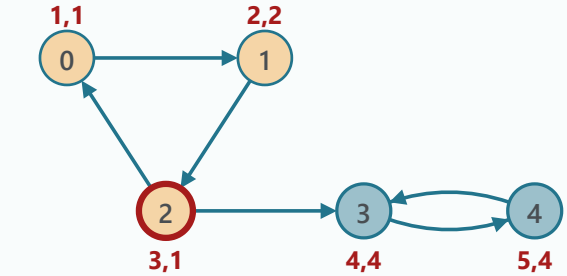
`v = 2` `w = 3` `low = [1, 2, 1, 4, 4]`

OUTPUT

```
4 3
```

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
▶    if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



low[2] = 1 ≠ index[2] = 3 → 2 is not a root and stays on the stack; return.

Step 16 / 20

VARIABLES

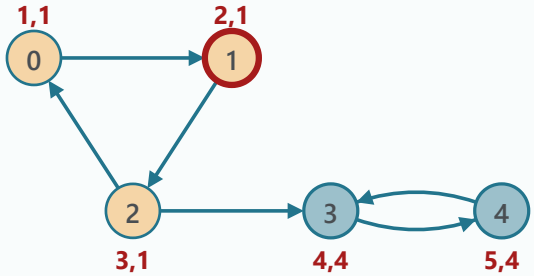
`v = 2` `w = –` `low = [1, 2, 1, 4, 4]`

OUTPUT

```
4 3
```

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
▶        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



`scc(2)` returned: low[1] = min(2, low[2] = 1) = **1**.

Step 17 / 20

VARIABLES

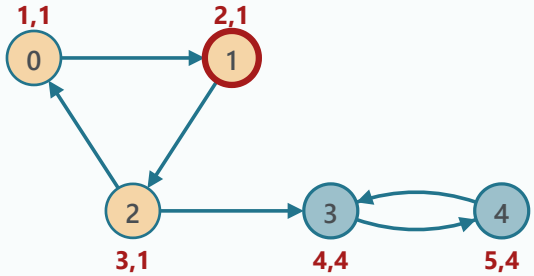
`v = 1` `w = 2` `low = [1, 1, 1, 4, 4]`

OUTPUT

```
4 3
```

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
▶    if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



low[1] = 1 ≠ index[1] = 2 → 1 is not a root and stays on the stack; return.

Step 18 / 20

VARIABLES

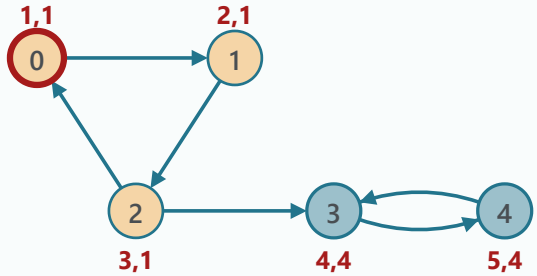
`v = 1` `w = –` `low = [1, 1, 1, 4, 4]`

OUTPUT

```
4 3
```

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
▶        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
11   if (low[v] == index[v]) {
12     int w;
13     do {
14       w = stack.pop(); onStack[w] = false;
15       System.out.print(w + " ");
16     } while (w != v);
17     System.out.println();
18   }
19 }
```



`scc(1)` returned: low[0] = min(1, low[1] = 1) = **1**.

Step 19 / 20

VARIABLES

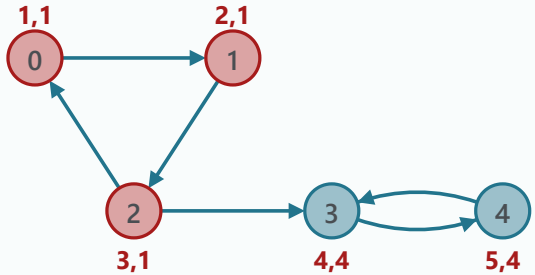
`v = 0` `w = 1` `low = [1, 1, 1, 4, 4]`

OUTPUT

```
4 3
```

Tarjan: Tracing `scc(0)`

```
1  void scc(int v) {
2    index[v] = low[v] = ++counter;
3    stack.push(v); onStack[v] = true;
4    for (int w : adj[v]) {
5      if (index[w] == 0) {
6        scc(w);
7        low[v] = Math.min(low[v], low[w]);
8      } else if (onStack[w])
9        low[v] = Math.min(low[v], index[w]);
10   }
▶    if (low[v] == index[v]) {
▶      int w;
▶      do {
▶        w = stack.pop(); onStack[w] = false;
▶        System.out.print(w + " ");
▶      } while (w != v);
▶      System.out.println();
18   }
19 }
```



stack

low[0] == index[0] = 1 → 0 is a root; pop and print 2, 1, 0: SCC **{0, 1, 2}**.

Step 20 / 20

VARIABLES

`v = 0` `w = –` `low = [1, 1, 1, 4, 4]`

OUTPUT

```
4 3
2 1 0
```

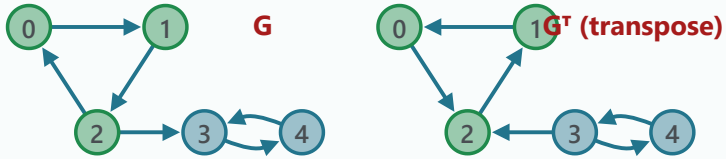
Complexity and Kosaraju

Tarjan's running time: O(V + E)

- V vertices, E edges.
- Each vertex is visited, pushed and popped once: O(V).
- Each edge is examined once: O(E).
- Extra memory: index, lowlink, onStack and the stack: O(V).

Kosaraju's algorithm

1. DFS pushes vertices in **finishing order**.
2. Build the **transpose** of the graph.
3. DFS on the transpose in pop order; each DFS tree is an SCC.



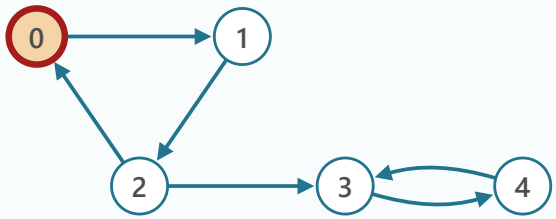
On the "Tarjan Step by Step" graph Kosaraju finds the same SCCs: {1, 2, 3}, {4, 5}, {6, 7}, {8}.

	Tarjan	Kosaraju
DFS runs	1	2 (one on the transpose)
Extra	index / lowlink, stack	transpose (O(V + E) memory)
Time	O(V + E)	O(V + E)

Kosaraju: SCCs with Two DFS Passes

```
 1 void kosaraju() {
 2   for (int v = 0; v < n; v++)  // pass 1
 3     if (!visited[v]) dfs1(v);
 4   Arrays.fill(visited, false); // pass 2
 5   while (!stack.isEmpty()) {
 6     int v = stack.pop();
 7     if (!visited[v]) {   // new SCC
 8       dfs2(v); System.out.println(); }
 9   }
10 }
11 void dfs1(int v) {
▶    visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```

original graph



stack ← top on the right

calls: dfs1(0)

Pass 1: DFS on the original graph. `dfs1(0)` : mark 0. The stack (top on the right) is empty; a vertex is pushed only **when it finishes**.

Step 1 / 12

VARIABLES

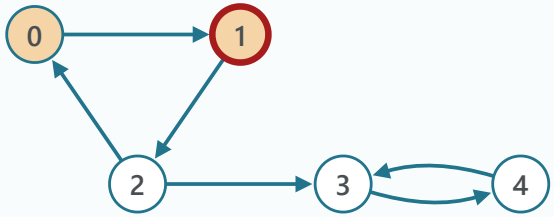
`v = 0` `stack = []`

OUTPUT

Kosaraju: SCCs with Two DFS Passes

```
 1 void kosaraju() {
 2   for (int v = 0; v < n; v++)  // pass 1
 3     if (!visited[v]) dfs1(v);
 4   Arrays.fill(visited, false); // pass 2
 5   while (!stack.isEmpty()) {
 6     int v = stack.pop();
 7     if (!visited[v]) {   // new SCC
 8       dfs2(v); System.out.println(); }
 9   }
10 }
11 void dfs1(int v) {
▶    visited[v] = true;
13   for (int w : adj[v])
▶      if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```

original graph



stack ← top on the right

calls: dfs1(0) → dfs1(1)

Neighbor 1 is unvisited → `dfs1(1)` : mark 1.

Step 2 / 12

VARIABLES

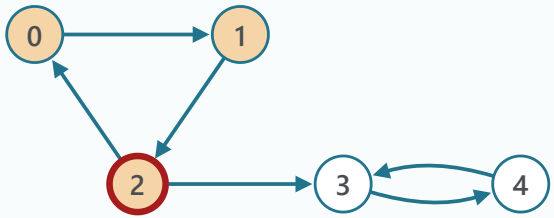
`v = 1` `stack = []`

OUTPUT

Kosaraju: SCCs with Two DFS Passes

```
 1 void kosaraju() {
 2   for (int v = 0; v < n; v++)  // pass 1
 3     if (!visited[v]) dfs1(v);
 4   Arrays.fill(visited, false); // pass 2
 5   while (!stack.isEmpty()) {
 6     int v = stack.pop();
 7     if (!visited[v]) {   // new SCC
 8       dfs2(v); System.out.println(); }
 9   }
10 }
11 void dfs1(int v) {
▶    visited[v] = true;
13   for (int w : adj[v])
▶      if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```

original graph



stack  ← top on the right

calls: dfs1(0) → dfs1(1) → dfs1(2)

Neighbor 2 is unvisited → `dfs1(2)` : mark 2.

Step 3 / 12

VARIABLES

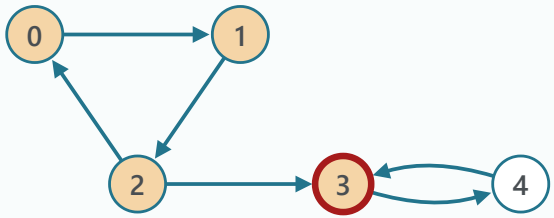
`v = 2` `stack = []`

OUTPUT

Kosaraju: SCCs with Two DFS Passes

```
 1 void kosaraju() {
 2   for (int v = 0; v < n; v++)  // pass 1
 3     if (!visited[v]) dfs1(v);
 4   Arrays.fill(visited, false); // pass 2
 5   while (!stack.isEmpty()) {
 6     int v = stack.pop();
 7     if (!visited[v]) {   // new SCC
 8       dfs2(v); System.out.println(); }
 9   }
10 }
11 void dfs1(int v) {
▶    visited[v] = true;
13   for (int w : adj[v])
▶      if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```

original graph



stack ← top on the right

calls: dfs1(0) → dfs1(1) → dfs1(2) → dfs1(3)

Neighbor 3 is unvisited → `dfs1(3)` : mark 3. (2's first neighbor, 0, is already visited and was skipped.)

Step 4 / 12

VARIABLES

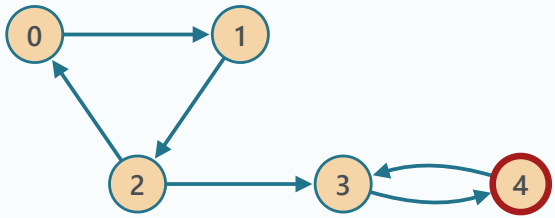
`v = 3` `stack = []`

OUTPUT

Kosaraju: SCCs with Two DFS Passes

```
 1 void kosaraju() {
 2   for (int v = 0; v < n; v++)  // pass 1
 3     if (!visited[v]) dfs1(v);
 4   Arrays.fill(visited, false); // pass 2
 5   while (!stack.isEmpty()) {
 6     int v = stack.pop();
 7     if (!visited[v]) {   // new SCC
 8       dfs2(v); System.out.println(); }
 9   }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
▶    stack.push(v);        // v finished
16 }
```

original graph



stack  ← top on the right

calls: dfs1(0) → dfs1(1) → dfs1(2) → dfs1(3) → dfs1(4)

4's only neighbor, 3, is already visited: 4 finishes → **push it**. The stack accumulates the finishing order.

VARIABLES

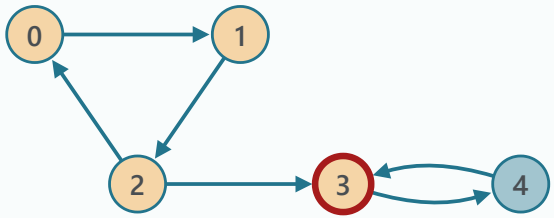
`v = 4` `stack = [4]`

OUTPUT

Kosaraju: SCCs with Two DFS Passes

```
 1 void kosaraju() {
 2   for (int v = 0; v < n; v++)  // pass 1
 3     if (!visited[v]) dfs1(v);
 4   Arrays.fill(visited, false); // pass 2
 5   while (!stack.isEmpty()) {
 6     int v = stack.pop();
 7     if (!visited[v]) {   // new SCC
 8       dfs2(v); System.out.println(); }
 9   }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
▶    stack.push(v);        // v finished
16 }
```

original graph



stack | 4 | 3 | | | | ← top on the right

calls: dfs1(0) → dfs1(1) → dfs1(2) → dfs1(3)

`dfs1(4)` returned; 3 has no other neighbor: 3 finishes and is pushed.

Step 6 / 12

VARIABLES

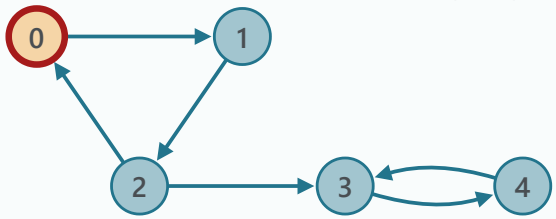
`v = 3` `stack = [4, 3]`

OUTPUT

Kosaraju: SCCs with Two DFS Passes

```
 1 void kosaraju() {
 2   for (int v = 0; v < n; v++)  // pass 1
 3     if (!visited[v]) dfs1(v);
 4   Arrays.fill(visited, false); // pass 2
 5   while (!stack.isEmpty()) {
 6     int v = stack.pop();
 7     if (!visited[v]) {   // new SCC
 8       dfs2(v); System.out.println(); }
 9   }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
▶    stack.push(v);        // v finished
16 }
```

original graph



stack | 4 | 3 | 2 | 1 | 0 | ← top on the right

calls: dfs1(0)

2, 1 and 0 finish in turn and are pushed. 0 is on top: the **last to finish**, it lies in a source component.

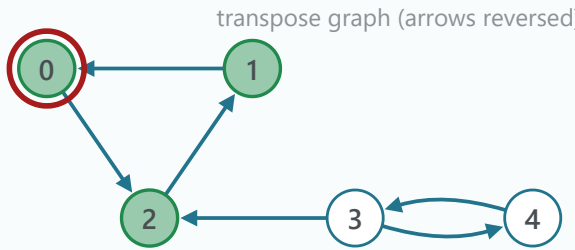
VARIABLES

`v = 0` `stack = [4, 3, 2, 1, 0]`

OUTPUT

Kosaraju: SCCs with Two DFS Passes

```
1  void kosaraju() {
2    for (int v = 0; v < n; v++)  // pass 1
3      if (!visited[v]) dfs1(v);
▶    Arrays.fill(visited, false); // pass 2
▶    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {   // new SCC
▶        dfs2(v); System.out.println(); }
9    }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```



stack | 4 | 3 | 2 | 1 | | ← top on the right

calls: —

Pass 2: `visited` is cleared and the arrows are **reversed**. Pop 0; on the transpose `dfs2(0)` reaches 0 → 2 → 1 and prints them: SCC {0, 1, 2}.

Step 8 / 12

VARIABLES

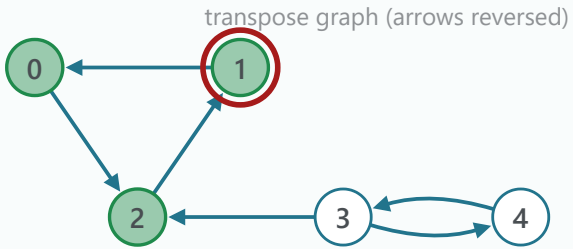
`v = 0` `stack = [4, 3, 2, 1]`

OUTPUT

```
0 2 1
```

Kosaraju: SCCs with Two DFS Passes

```
1  void kosaraju() {
2    for (int v = 0; v < n; v++)  // pass 1
3      if (!visited[v]) dfs1(v);
4    Arrays.fill(visited, false); // pass 2
▶    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {   // new SCC
8        dfs2(v); System.out.println(); }
9    }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```



stack | 4 | 3 | 2 | | | ← top on the right

calls: —

Pop 1, but it is already visited (printed with 0's component): skip. The same happens to 2 next.

Step 9 / 12

VARIABLES

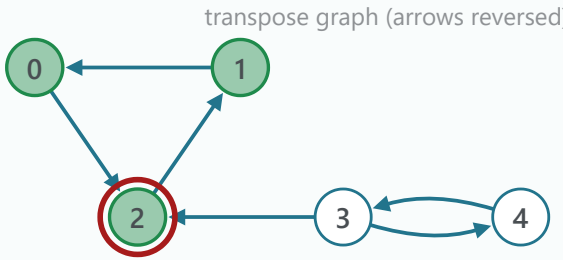
`v = 1` `stack = [4, 3, 2]`

OUTPUT

```
0 2 1
```

Kosaraju: SCCs with Two DFS Passes

```
1  void kosaraju() {
2    for (int v = 0; v < n; v++)  // pass 1
3      if (!visited[v]) dfs1(v);
4    Arrays.fill(visited, false); // pass 2
▶    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {   // new SCC
8        dfs2(v); System.out.println(); }
9    }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```



stack | 4 | 3 | | | | ← top on the right

calls: —

2 is visited too: skip.

Step 10 / 12

VARIABLES

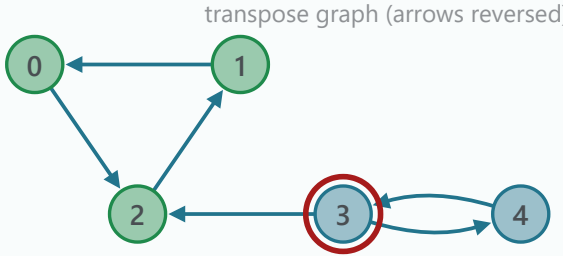
`v = 2` `stack = [4, 3]`

OUTPUT

```
0 2 1
```

Kosaraju: SCCs with Two DFS Passes

```
1  void kosaraju() {
2    for (int v = 0; v < n; v++)  // pass 1
3      if (!visited[v]) dfs1(v);
4    Arrays.fill(visited, false); // pass 2
5    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {   // new SCC
▶        dfs2(v); System.out.println(); }
9    }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```



Pop 3, unvisited: on the transpose `dfs2(3)` reaches 3 → 4 (the reversed edge 3 → 2 leads to visited 2 and is skipped). SCC {3, 4}.

VARIABLES

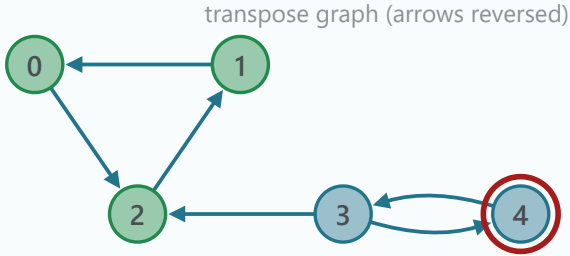
`v = 3` `stack = [4]`

OUTPUT

```
0 2 1
3 4
```

Kosaraju: SCCs with Two DFS Passes

```
1  void kosaraju() {
2    for (int v = 0; v < n; v++)  // pass 1
3      if (!visited[v]) dfs1(v);
4    Arrays.fill(visited, false); // pass 2
▶    while (!stack.isEmpty()) {
▶      int v = stack.pop();
▶      if (!visited[v]) {   // new SCC
8        dfs2(v); System.out.println(); }
9    }
10 }
11 void dfs1(int v) {
12   visited[v] = true;
13   for (int w : adj[v])
14     if (!visited[w]) dfs1(w);
15   stack.push(v);        // v finished
16 }
```



Pop 4, visited: skip. The stack is empty; two SCCs found: {0, 1, 2} and {3, 4}.

Step 12 / 12

VARIABLES

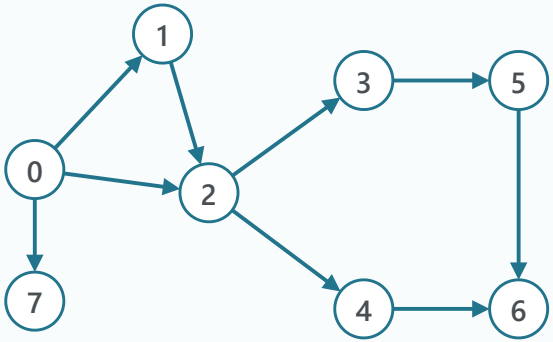
`v = 4` `stack = []`

OUTPUT

```
0 2 1
3 4
```

APPLICATIONS

Topological Sort



In-degrees: 0:0, 1:1, 2:2, 3:1, 4:1, 5:1, 6:2, 7:1

- In a directed acyclic graph (DAG), an ordering where u comes **before** v for every edge u → v: course prerequisites, build order, task scheduling.
- **Kahn** (BFS-based): enqueue the vertices of in-degree 0; when a vertex is removed, delete its edges and enqueue any vertex whose in-degree drops to 0.
- **DFS-based**: list vertices in finishing order, then reverse the list.

Kahn	**0 1 7 2 3 4 5 6**
DFS (reverse finish)	**0 7 1 2 4 3 5 6**

Both are valid: a topological order is not unique. If fewer than V vertices come out, the graph has a **cycle** (add the edge 2 → 0 from the DFS example and no ordering exists). Time O(V + E).

Topological Sort: Tracing Kahn's Algorithm

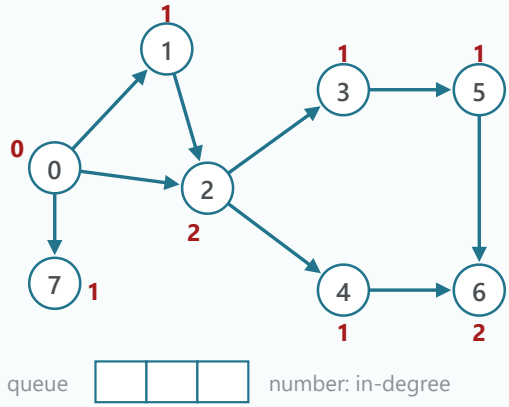
```
1  void kahn() {
▶    int[] indeg = new int[n];
▶    for (int v = 0; v < n; v++)
▶      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
9      int v = queue.poll();
10     System.out.print(v + " ");
11     for (int w : adj[v])
12       if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

VARIABLES

`v = –` `queue = []`

`indeg = [0, 1, 2, 1, 1, 1, 2, 1]`

OUTPUT



A directed acyclic graph (DAG). First count each vertex's **in-degree**: how many edges enter it? The small numbers next to the vertices are the `indeg[]` values.

Step 1 / 10

Topological Sort: Tracing Kahn's Algorithm

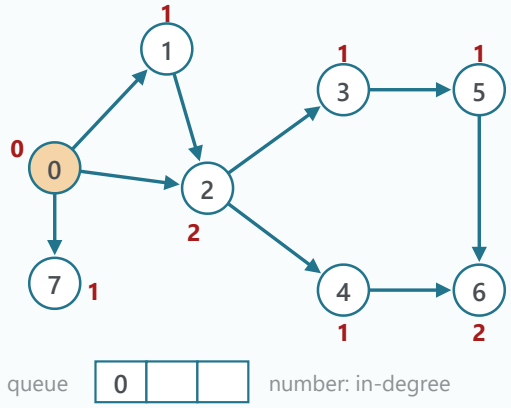
```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
▶    for (int v = 0; v < n; v++)
▶      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
9      int v = queue.poll();
10     System.out.print(v + " ");
11     for (int w : adj[v])
12       if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

VARIABLES

`v = –` `queue = [0]`

`indeg = [0, 1, 2, 1, 1, 1, 2, 1]`

OUTPUT



Vertex 0 is the only one with in-degree 0: it has no prerequisites, so it is enqueued.

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

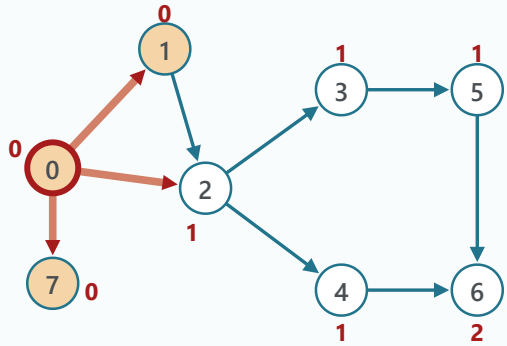
VARIABLES

`v = 0` `queue = [1, 7]`

`indeg = [0, 0, 1, 1, 1, 1, 2, 0]`

OUTPUT

```
0
```



Dequeue and print 0; its edges count as deleted, so the in-degrees of 1, 2 and 7 drop by 1. 1 and 7 reach zero and are enqueued.

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

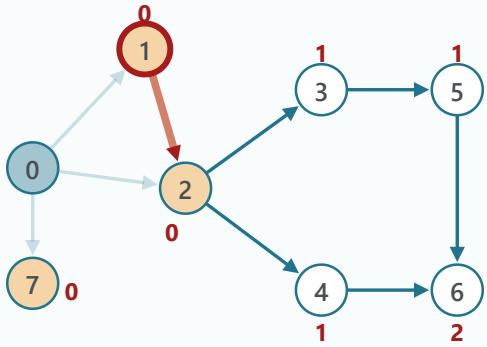
VARIABLES

`v = 1` `queue = [7, 2]`

`indeg = [0, 0, 0, 1, 1, 1, 2, 0]`

OUTPUT

```
0 1
```



Dequeue and print 1; its edges count as deleted, so the in-degree of 2 drops by 1. 2 reaches zero and is enqueued.

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

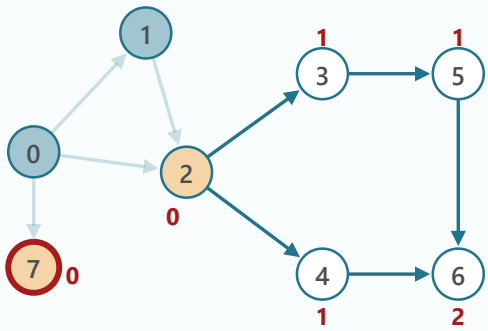
VARIABLES

`v = 7` `queue = [2]`

`indeg = [0, 0, 0, 1, 1, 1, 2, 0]`

OUTPUT

```
0 1 7
```



Dequeue and print 7. It has no outgoing edges; no in-degree changes.

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

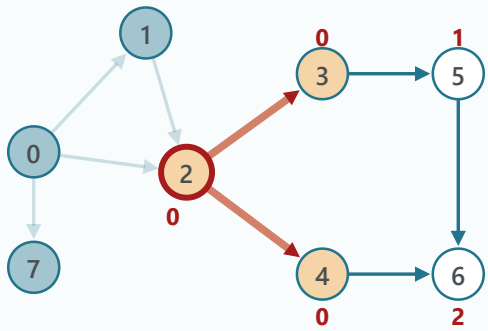
VARIABLES

`v = 2` `queue = [3, 4]`

`indeg = [0, 0, 0, 0, 0, 1, 2, 0]`

OUTPUT

```
0 1 7 2
```



Dequeue and print 2; its edges count as deleted, so the in-degrees of 3 and 4 drop by 1. 3 and 4 reach zero and are enqueued.

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

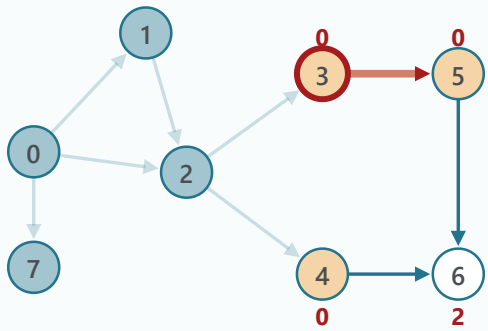
VARIABLES

`v = 3` `queue = [4, 5]`

`indeg = [0, 0, 0, 0, 0, 0, 2, 0]`

OUTPUT

```
0 1 7 2 3
```



Dequeue and print 3; its edges count as deleted, so the in-degree of 5 drops by 1. 5 reaches zero and is enqueued.

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

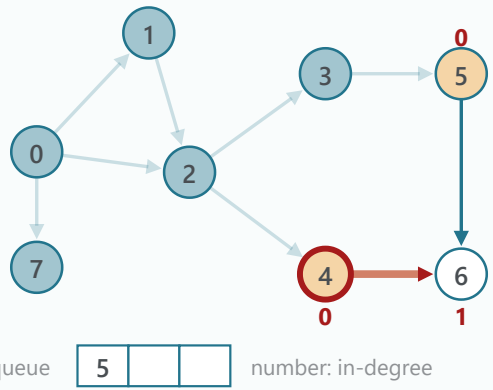
VARIABLES

`v = 4` `queue = [5]`

`indeg = [0, 0, 0, 0, 0, 0, 1, 0]`

OUTPUT

```
0 1 7 2 3 4
```



Dequeue and print 4; its edges count as deleted, so the in-degree of 6 drops by 1. Nothing reaches zero (6 still has 1 prerequisite).

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

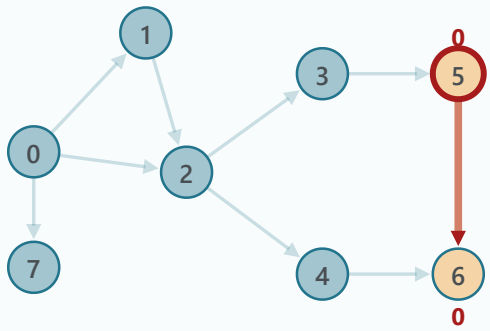
VARIABLES

`v = 5` `queue = [6]`

`indeg = [0, 0, 0, 0, 0, 0, 0, 0]`

OUTPUT

```
0 1 7 2 3 4 5
```



Dequeue and print 5; its edges count as deleted, so the in-degree of 6 drops by 1. 6 reaches zero and is enqueued.

Topological Sort: Tracing Kahn's Algorithm

```
1  void kahn() {
2    int[] indeg = new int[n];
3    for (int v = 0; v < n; v++)
4      for (int w : adj[v]) indeg[w]++;
5    Queue<Integer> queue = new LinkedList<>();
6    for (int v = 0; v < n; v++)
7      if (indeg[v] == 0) queue.add(v);
8    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      System.out.print(v + " ");
▶      for (int w : adj[v])
▶        if (--indeg[w] == 0) queue.add(w);
13   }
14 }
```

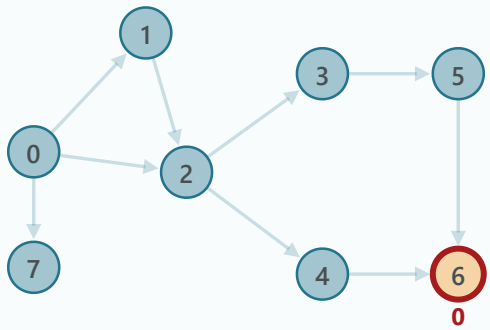
VARIABLES

`v = 6` `queue = []`

`indeg = [0, 0, 0, 0, 0, 0, 0, 0]`

OUTPUT

```
0 1 7 2 3 4 5 6
```



Dequeue and print 6. It has no outgoing edges; no in-degree changes. The queue is empty and all 8 vertices came out, so there is no cycle. Order: **0 1 7 2 3 4 5 6**.

APPLICATIONS

More Uses of Traversal

Connected components

Start a new DFS/BFS from each unvisited vertex; the number of starts = the number of components.

Bipartiteness test

Color the BFS levels with two colors; if two neighbors share a color there is an odd cycle and the graph is not bipartite.

Cycle detection

Undirected: a visited neighbor that is not the parent.
Directed: a back edge to a vertex on the stack (yellow).

Bridges and articulation points

Tarjan's lowlink idea: if lowlink(w) > disc(v), then v–w is a bridge.

Iterative deepening (IDDFS)

DFS runs with depth limits 0, 1, 2…: the memory of DFS with BFS's guarantee of the shallowest solution.

Random walk

Move to a random neighbor at each step; PageRank and sampling large networks.

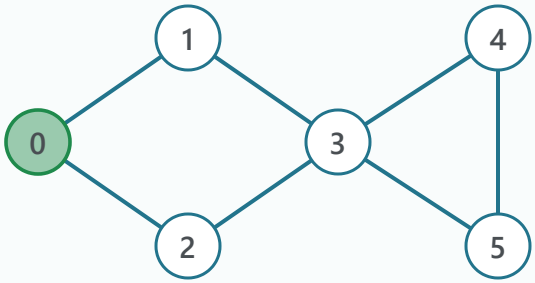
Bipartiteness Test with BFS

```
1  boolean isBipartite(int s) {
▶    Arrays.fill(color, -1);
▶    color[s] = 0; queue.add(s);
4    while (!queue.isEmpty()) {
5      int v = queue.poll();
6      for (int w : adj[v])
7        if (color[w] == -1) {
8          color[w] = 1 - color[v];
9          queue.add(w);
10       } else if (color[w] == color[v])
11         return false;  // same color
12   }
13   return true;
14 }
```

VARIABLES

`v = –` `w = –` `queue = [0]`

`color = [0, -1, -1, -1, -1, -1]`





Undirected graph: the cycle 0-1-3-2 and the triangle 3-4-5. color = −1 means uncolored; source 0 gets color 0 (green) and is enqueued.

Step 1 / 6

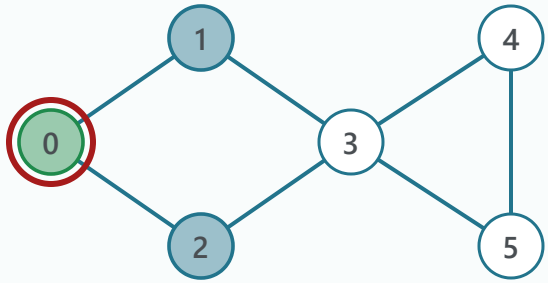
Bipartiteness Test with BFS

```
1  boolean isBipartite(int s) {
2    Arrays.fill(color, -1);
3    color[s] = 0; queue.add(s);
4    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (color[w] == -1) {
▶          color[w] = 1 - color[v];
▶          queue.add(w);
10       } else if (color[w] == color[v])
11         return false;  // same color
12   }
13   return true;
14 }
```

VARIABLES

`v = 0` `w = –` `queue = [1, 2]`

`color = [0, 1, 1, -1, -1, -1]`



queue | 1 | 2 | | color 0 | color 1

Dequeue 0 (color 0). Its uncolored neighbors 1 and 2 get the opposite color (1) and are enqueued.

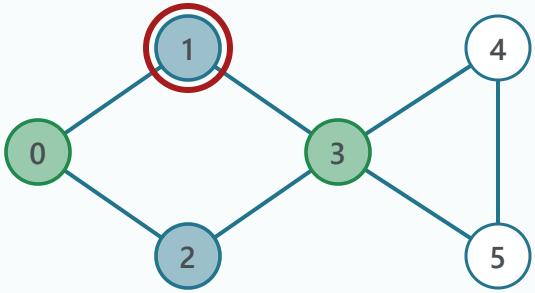
Bipartiteness Test with BFS

```
1  boolean isBipartite(int s) {
2    Arrays.fill(color, -1);
3    color[s] = 0; queue.add(s);
4    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (color[w] == -1) {
▶          color[w] = 1 - color[v];
▶          queue.add(w);
10       } else if (color[w] == color[v])
11         return false;  // same color
12   }
13   return true;
14 }
```

VARIABLES

`v = 1` `w = –` `queue = [2, 3]`

`color = [0, 1, 1, 0, –1, –1]`





Dequeue 1 (color 1). Uncolored neighbor 3 gets the opposite color (0) and is enqueued. Neighbor 0 already has the opposite color: the edge joins the two sets, fine.

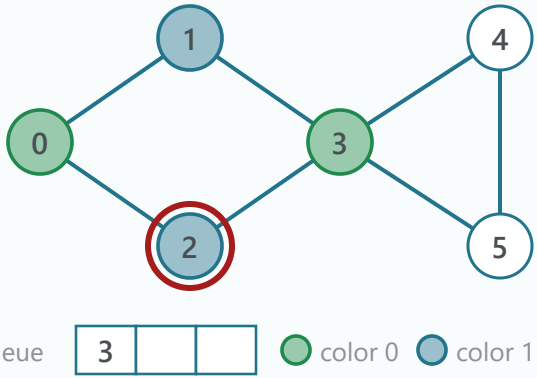
Bipartiteness Test with BFS

```
1  boolean isBipartite(int s) {
2    Arrays.fill(color, -1);
3    color[s] = 0; queue.add(s);
4    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (color[w] == -1) {
8          color[w] = 1 - color[v];
9          queue.add(w);
▶        } else if (color[w] == color[v])
11         return false;  // same color
12   }
13   return true;
14 }
```

VARIABLES

`v = 2` `w = –` `queue = [3]`

`color = [0, 1, 1, 0, –1, –1]`



Dequeue 2 (color 1). Neighbors 0 and 3 already have the opposite color: the edges join the two sets, fine.

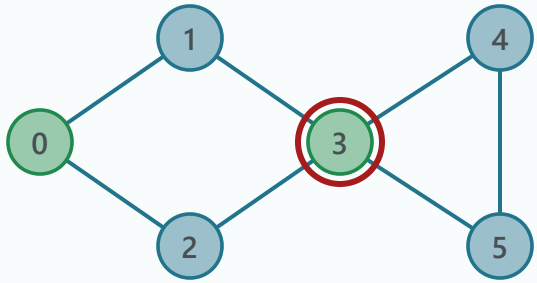
Bipartiteness Test with BFS

```
1  boolean isBipartite(int s) {
2    Arrays.fill(color, -1);
3    color[s] = 0; queue.add(s);
4    while (!queue.isEmpty()) {
▶      int v = queue.poll();
▶      for (int w : adj[v])
▶        if (color[w] == -1) {
▶          color[w] = 1 - color[v];
▶          queue.add(w);
10       } else if (color[w] == color[v])
11         return false;  // same color
12   }
13   return true;
14 }
```

VARIABLES

`v = 3` `w = –` `queue = [4, 5]`

`color = [0, 1, 1, 0, 1, 1]`



queue | 4 | 5 | | color 0 | color 1

Dequeue 3 (color 0). Uncolored neighbors 4 and 5 get the opposite color (1) and are enqueued. Neighbors 1 and 2 already have the opposite color: the edges join the two sets, fine.

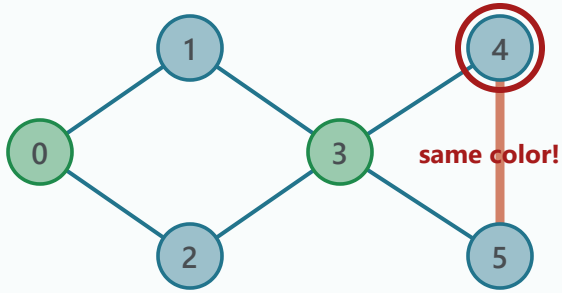
Bipartiteness Test with BFS

```
1  boolean isBipartite(int s) {
2    Arrays.fill(color, -1);
3    color[s] = 0; queue.add(s);
4    while (!queue.isEmpty()) {
5      int v = queue.poll();
6      for (int w : adj[v])
7        if (color[w] == -1) {
8          color[w] = 1 - color[v];
9          queue.add(w);
▶        } else if (color[w] == color[v])
▶          return false;  // same color
12   }
13   return true;
14 }
```

VARIABLES

`v = 4` `w = 5` `queue = [5]`

`color = [0, 1, 1, 0, 1, 1]` `return = false`



queue 5 color 0 color 1

Dequeue 4; neighbor 3 has a different color (fine), but neighbor 5 has the **same color** (1). Both ends of an edge fell into one set: there is an **odd cycle**, like the triangle 3-4-5 → `false`.

Summary

Topic	Key points
Representation	list O(V + E) memory, matrix O(V^2); traversal O(V + E) with a list, O(V^2) with a matrix
DFS	stack / recursion; go deep, backtrack; a visited array guards against cycles
Stack DFS	mark on pop; a vertex may be pushed several times; order is the reverse of the neighbor order
BFS	queue; mark on enqueue; level by level; unweighted shortest paths via the parent array
Maze	cell = vertex; BFS tries all paths at once, and the first path to reach the exit is shortest
SCC	maximal mutually reachable set; the component graph is a DAG
Tarjan	index = discovery order; lowlink; lowlink = index → root, pop the SCC off the stack; O(V + E)
Applications	topological order (Kahn / reverse finish), components, bipartiteness, cycles, bridges

SUMMARY

Common Mistakes

Forgetting the visited array

On a cyclic graph DFS recurses forever and BFS's queue never empties. That is where the myth "BFS fails on cyclic graphs" comes from; with marking it works fine.

Marking on dequeue in BFS

The same vertex gets enqueued by many neighbors; the result may still be right, but the queue grows to O(E). Mark on enqueue instead.

Expecting shortest paths from DFS

The first path DFS finds to a target is not the shortest. Use BFS when unweighted, Dijkstra when weighted.

The order of stack DFS

Pushing neighbors in order reverses the result (0 7 2 … ≠ 0 1 2 …). To get the recursive order, push them in reverse.

Skipping the "on the stack?" check in Tarjan

Only a w on the stack gets `min(…, index(w))`; a w whose SCC is already found is ignored. Otherwise roots are lost.

Disconnected graphs

A traversal from one source sees only its component. To cover the whole graph, restart from every unvisited vertex.

SUMMARY

Worked Problems

Undirected graph, neighbors in alphabetical order: A–B, A–C, B–D, C–D, C–E, D–F, E–F, F–G.

1. Give the BFS visit order from A and the distance of each vertex.
2. What is the recursive DFS visit order from A?
3. What is the shortest path from A to G (using the BFS parent array)?
4. Is the graph bipartite? Give the two sets.
5. Directed graph 0→1, 1→2, 2→0, 1→3, 3→4, 4→5, 5→3, 5→6: list the SCCs in the order Tarjan finds them (start at 0, index from 1).
6. What is Kahn's order for A→C, B→C, B→D, C→E, D→E, E→F (queue starts in alphabetical order)?

▶ **Show answers**

Exercises

1. On the worked-problems graph, what order does **stack DFS** from A give (neighbors pushed alphabetically)? How many pushes in total?
2. With V = 1000 and E = 3000, compare the work of BFS on an adjacency list and on a matrix.
3. Give a small weighted graph where BFS misses the shortest path.
4. How do you count the connected components of an undirected graph? Running time?
5. In "Tarjan Step by Step", what if a w in a finished SCC also triggered `min(v.lowlink, w.index)`?

▶ **Show answers**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 20: Elementary Graph Algorithms).
- Sedgewick and Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Sections 4.1–4.2).
- R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms", *SIAM Journal on Computing* 1(2), 1972.
- A. B. Kahn, "Topological Sorting of Large Networks", *Communications of the ACM* 5(11), 1962.
- Kleinberg and Tardos, *Algorithm Design*, Pearson, 2006 (Chapter 3: Graphs).

The End

Chapter 5: Graph Traversal

Previous: Chapter 4: Searching Algorithms · Next: Chapter 6: Shortest Paths