EDA 软件设计 I

Lecture 10

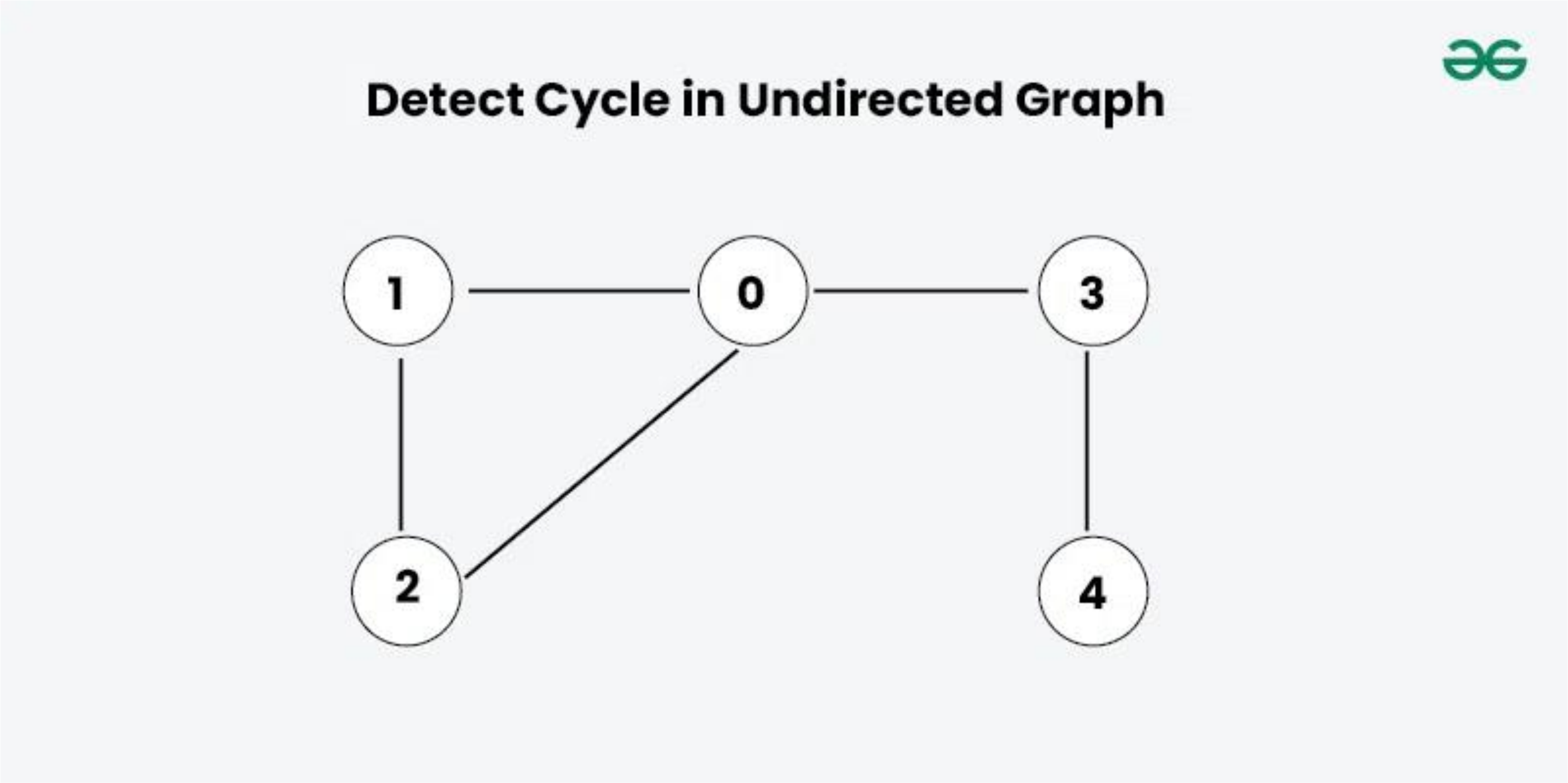
Cycle Detection

- 分为无向图的环检测和有向图的环检测
- **无向图的环检测**
  - BFS环检测：
    1. 从图中每个未访问的节点开始进行广度优先遍历（BFS）
    2. 在遍历过程中，标记每个访问过的节点
    3. 如果遇到已经标记为已访问的节点，则表示存在环
    4. 继续进行BFS遍历，直到所有节点都被访问或检测到环

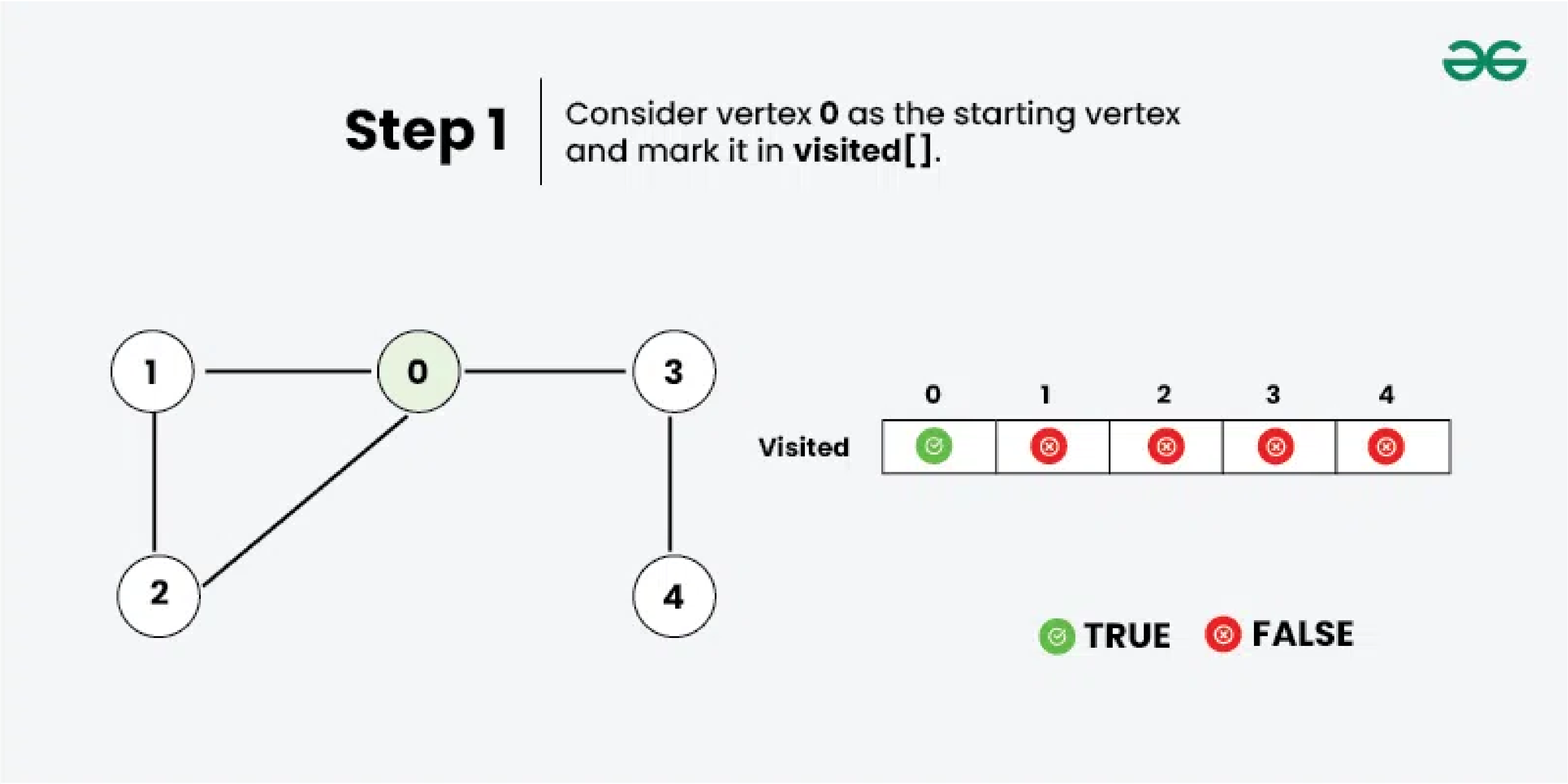
Cycle Detection

- 分为无向图的环检测和有向图的环检测
- **无向图的环检测**
  - BFS环检测：
    1. 从图中每个未访问的节点开始进行广度优先遍历（BFS）
    2. 在遍历过程中，标记每个访问过的节点
    3. 如果遇到已经标记为已访问的节点，则表示存在环？
    4. 继续进行BFS遍历，直到所有节点都被访问或检测到环

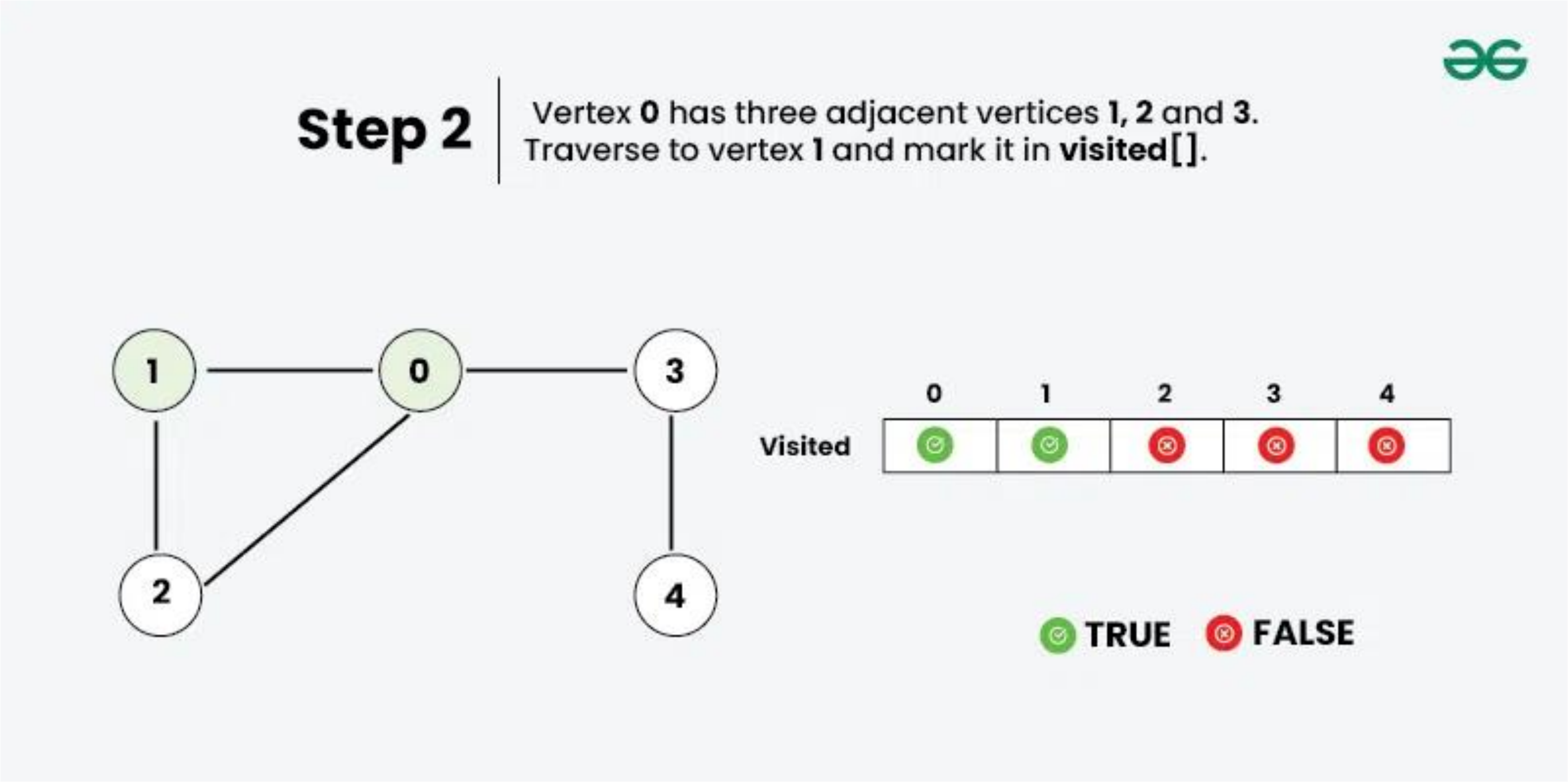
无向图BFS环检测



无向图BFS环检测

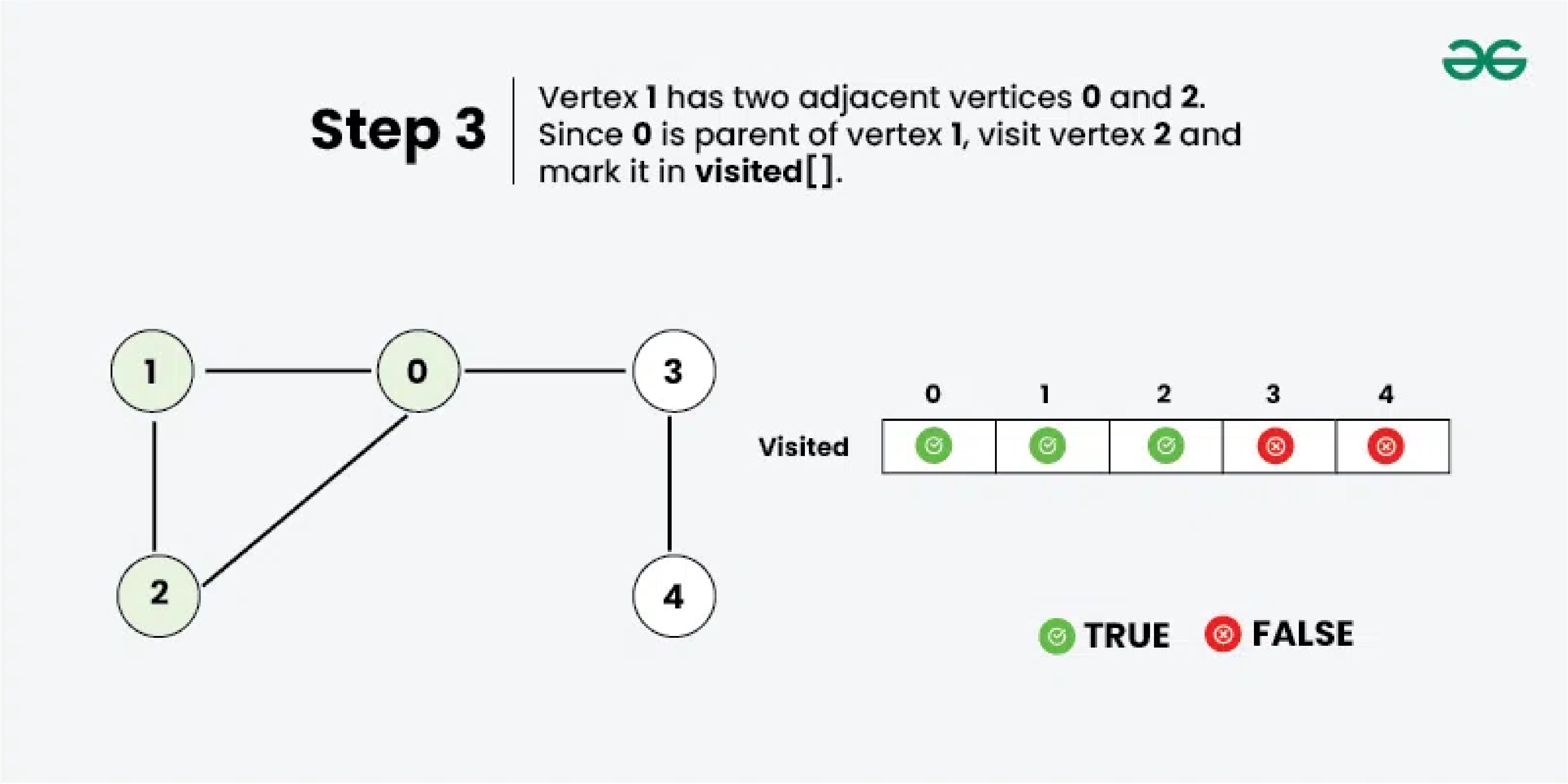

Step 1
Consider vertex 0 as the starting vertex
and mark it in visited[].
1
0
3
2
4
Visited
0
1
2
3
4
TRUE
FALSE

无向图BFS环检测

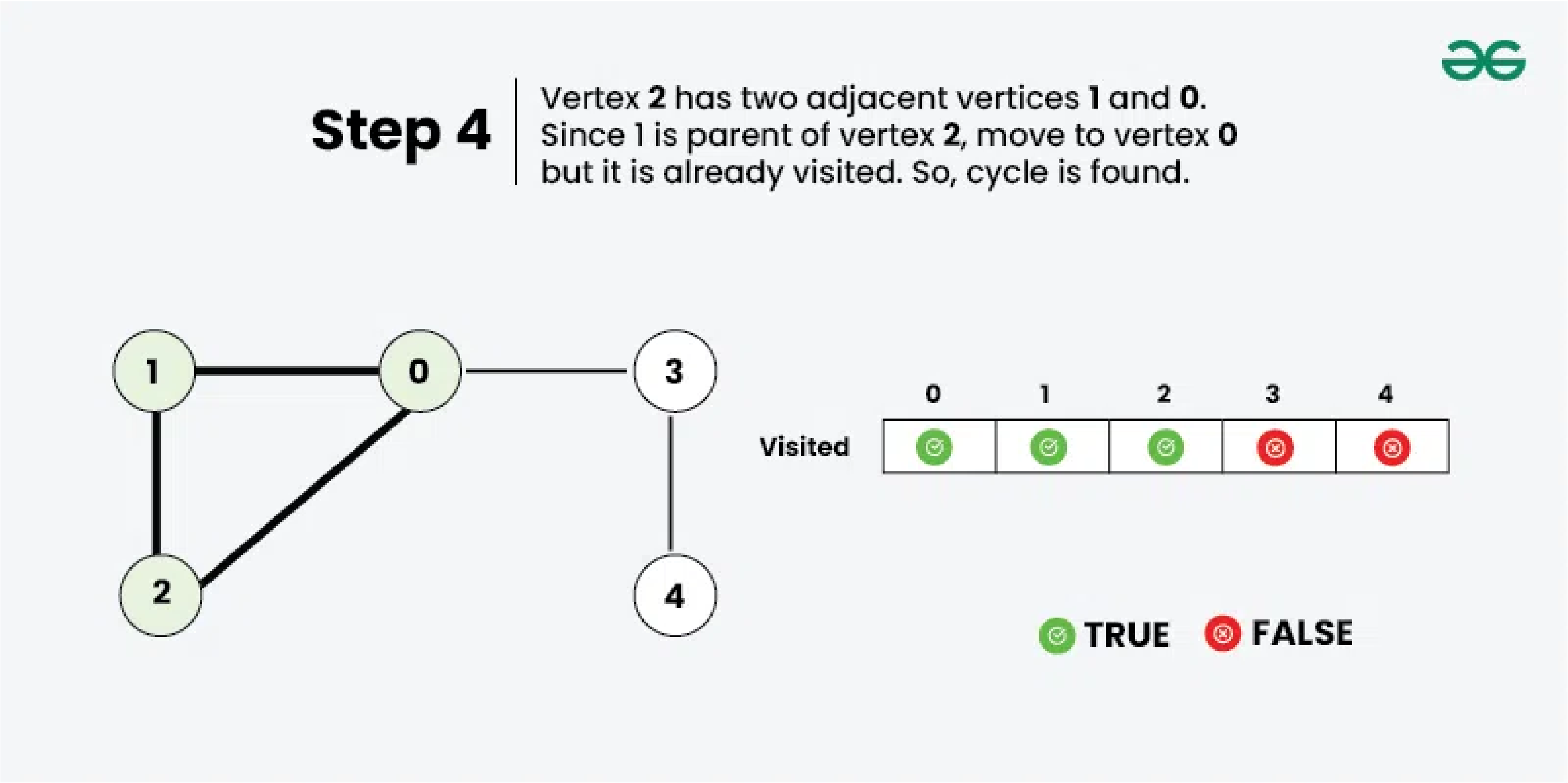


Step 2
Vertex 0 has three adjacent vertices 1, 2 and 3.
Traverse to vertex 1 and mark it in visited[].
1
0
3
2
4
0
1
2
3
4
Visited
TRUE
FALSE

无向图BFS环检测


Step 3
Vertex 1 has two adjacent vertices 0 and 2.
Since 0 is parent of vertex 1, visit vertex 2 and
mark it in visited[].
1
0
3
2
4
Visited
0
1
2
3
4
TRUE
FALSE

无向图BFS环检测


Step 4
Vertex 2 has two adjacent vertices 1 and 0.
Since 1 is parent of vertex 2, move to vertex 0
but it is already visited. So, cycle is found.
1
0
3
2
4
Visited
0
1
2
3
4
TRUE
FALSE

拓扑排序算法

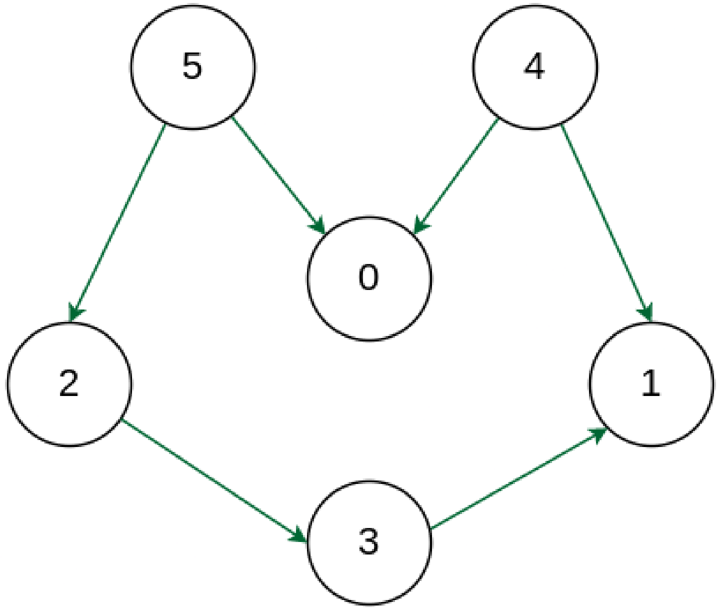
- What for? —— 拓扑排序是专门应用于**X**场景的，可以用来解决其中**Y**问题的一种算法
  - **X**: 涉及事物之间依赖关系（**DAG**）
  - **Y**: 执行顺序排序
- **拓扑**：就是代表图结构（input）
- **排序**：根据拓扑结构，找到一种**符合依赖关系的节点排列顺序**(output)

拓扑排序算法目标

- 在**图论**中，拓扑排序可以被形式化地定义为：
  - 给定一个**有向无环图** $G = (V, E)$，拓扑排序是对图中顶点的线性排序 $[v_1\ v_2, \cdots, v_n]$，使得对于每一条有向边（v_i , v_j）$\in E$，节点 v_i 都出现在节点 v_j 之前（即 v_i 在 v_j 之前被排列）
  - 数学表达式为：对于任意（u, v）$\in E$，有 $u \prec v$，其中 $\prec$ 表示顺序关系
- 拓扑排序的**目标**：为DAG中的所有节点找到一种线性排序，使得每个节点都出现在它所有前置依赖的节点之后

Your idea?

- 拓扑排序的**目标**：为DAG中的所有节点找到一种线性排序，使得**每个节点都出现在它所有前置依赖的节点之后**
- **How to achieve this goal? What is your first attempt?**
- **直觉**: 先找入度低（被依赖得少）的点，使其排序在前



算法原理 @ 拓扑排序（Kahn）

Kahn (卡恩) Algorithm Review: **通过不断找到入度为零的节点并处理它们**

① 入度为零的节点：首先在有向无环图（DAG）中找到所有入度为零的节点，这些节点没有依赖，可以优先处理

② 移除节点及更新入度：处理一个入度为零的节点，将其从图中移除，并将与之相邻节点的入度减1，如果某个相邻节点的入度变为零，则将其加入下一轮处理

③ 重复直到处理完所有节点：不断重复上述过程，直到所有节点都被处理。如果剩余节点无法被处理，说明图中存在环

算法原理 @ 拓扑排序（DFS）

- **Why（可利用DFS实现的基础）：**DFS的深入探索+回溯的特点非常适合处理依赖关系
  - 因为它可以先处理所有依赖的节点，然后再处理当前节点
- **原理 (short):** 递归地访问所有依赖的节点，并**在回溯时将节点加入排序结果中**

算法原理 @ 拓扑排序（DFS） in detail

① **DFS 遍历：**对图中的每个节点进行深度优先搜索，如果一个节点还没有被访问过，则开始访问该节点

② **递归处理节点：**在访问某个节点时，先递归地访问它所有的邻居节点（即被该节点指向的节点）

- 这一步确保我们在访问当前节点之前，已经访问了它依赖的所有节点

③ **回溯是记录节点：**当所有邻居节点都已经被处理完毕时，回溯到当前节点。在回溯的过程中，将该节点加入结果列表中

- (注意：加入列表的顺序是回溯时的顺序，而不是递归进入时的顺序。这确保了依赖的节点被先处理)

④ **生成拓扑顺序：**完成整个 DFS 遍历后，最终的拓扑排序是按照节点回溯的顺序排列的

算法原理 @ 拓扑排序（DFS）

- 简单归纳:

  ① Run **DFS**

  ② 回溯时记录节点顺序（区别于DFS遍历，在深入探索时便记录节点顺序

  ③ Reverse节点顺序 → 拓扑排序顺序

带着问题看算法

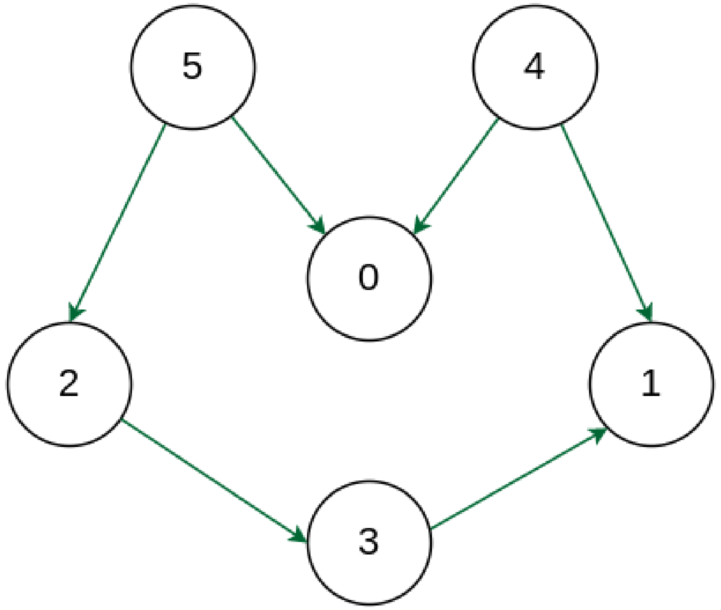
- 为什么拓扑排序不能在**无向图**上实现?
- 为什么拓扑排序不能对**有环的有向图**实现?
- 拓扑排序的输出是「**唯一确定**」的吗?

拓扑排序过程可视化

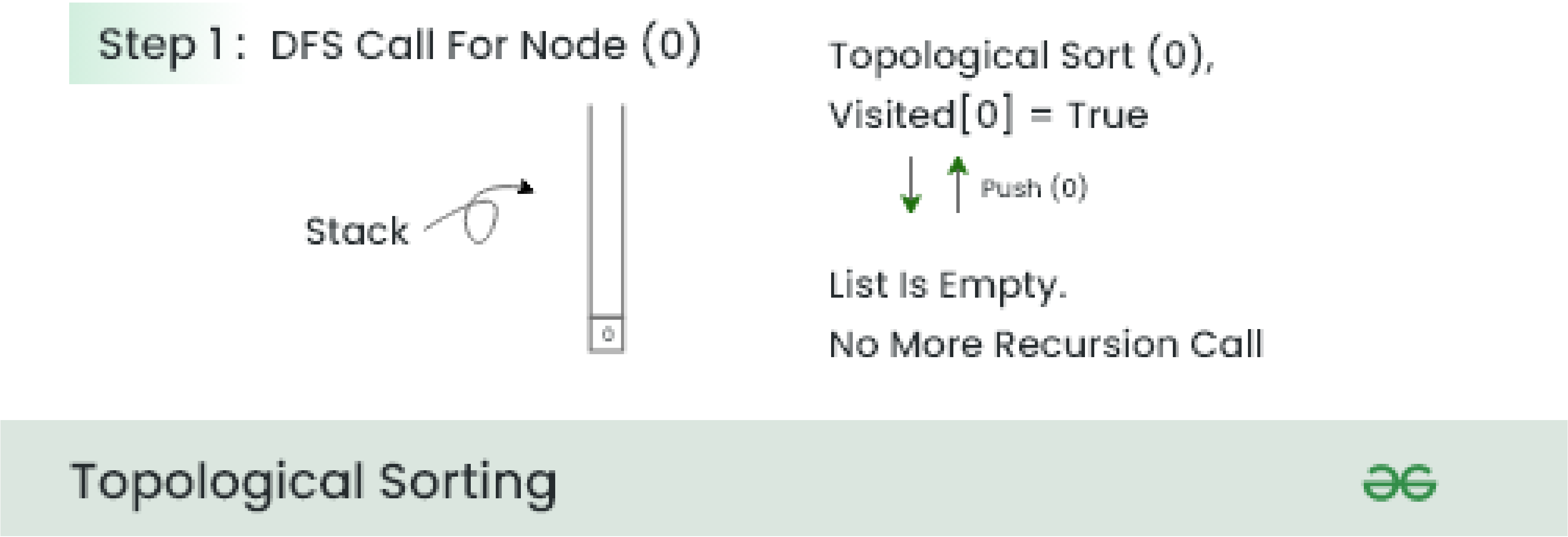
Adjacency List (G)

0	{}
1	{}
2	{3}
3	{1}
4	{0,1}
5	{2,0}

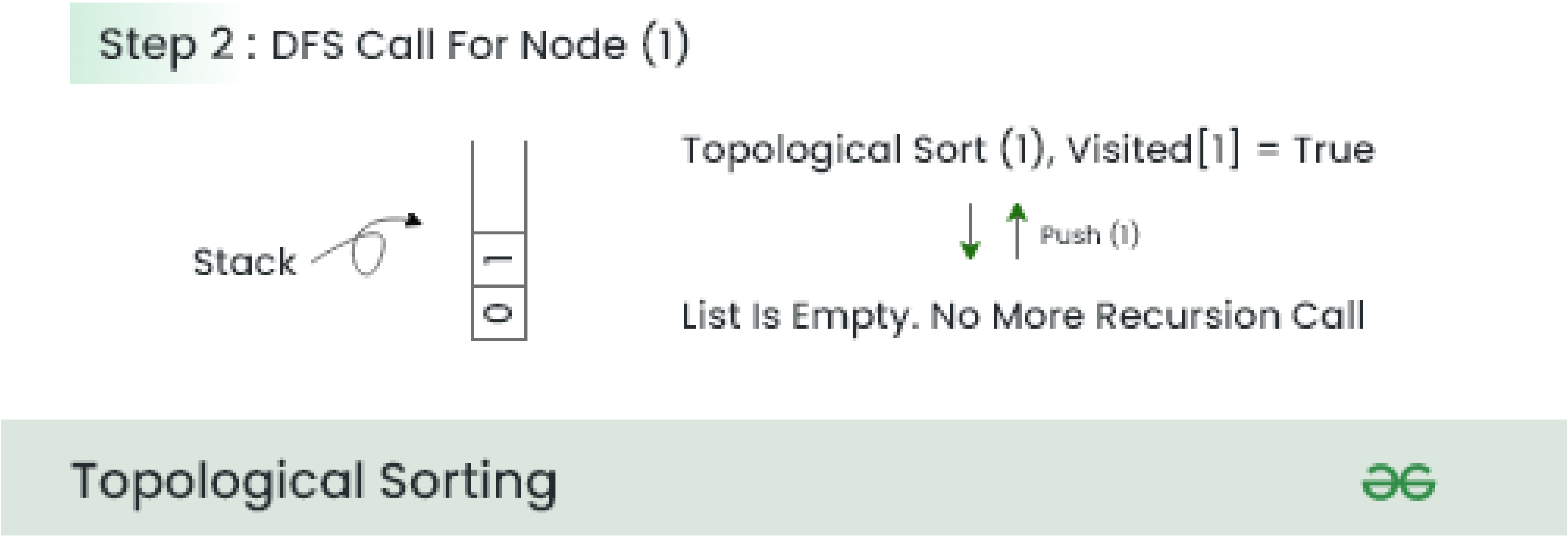
	0	1	2	3	4	5
Visited	False	False	False	False	False	False



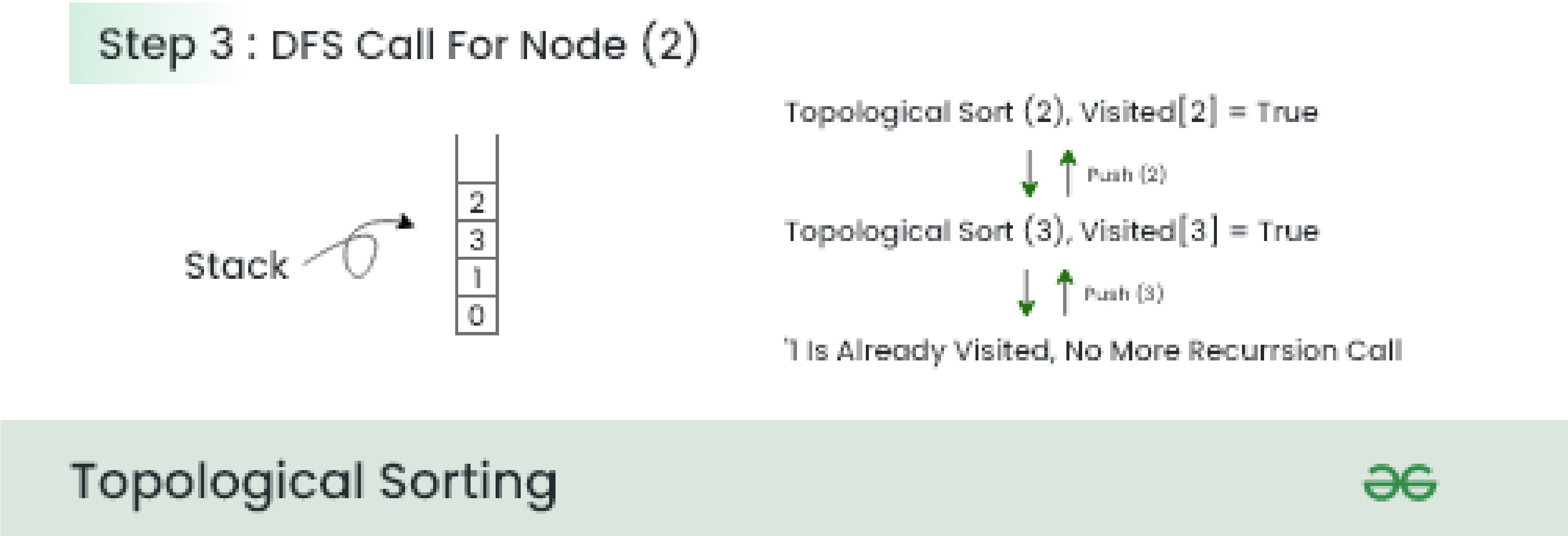
- 对于图中每个未访问的顶点，执行以下操作：
  ① 调用DFS函数，并将该顶点作为参数传递
  ② 在DFS函数中，将顶点标记为已访问，并递归调用DFS函数，处理该顶点所有未访问的邻居
  ③ 一旦所有邻居都已被访问，将该顶点压入栈中
  ④ 当所有顶点都被访问完后，从栈中弹出元素并将其依次添加到输出列表中，直到栈为空——最终得到的列表为输出



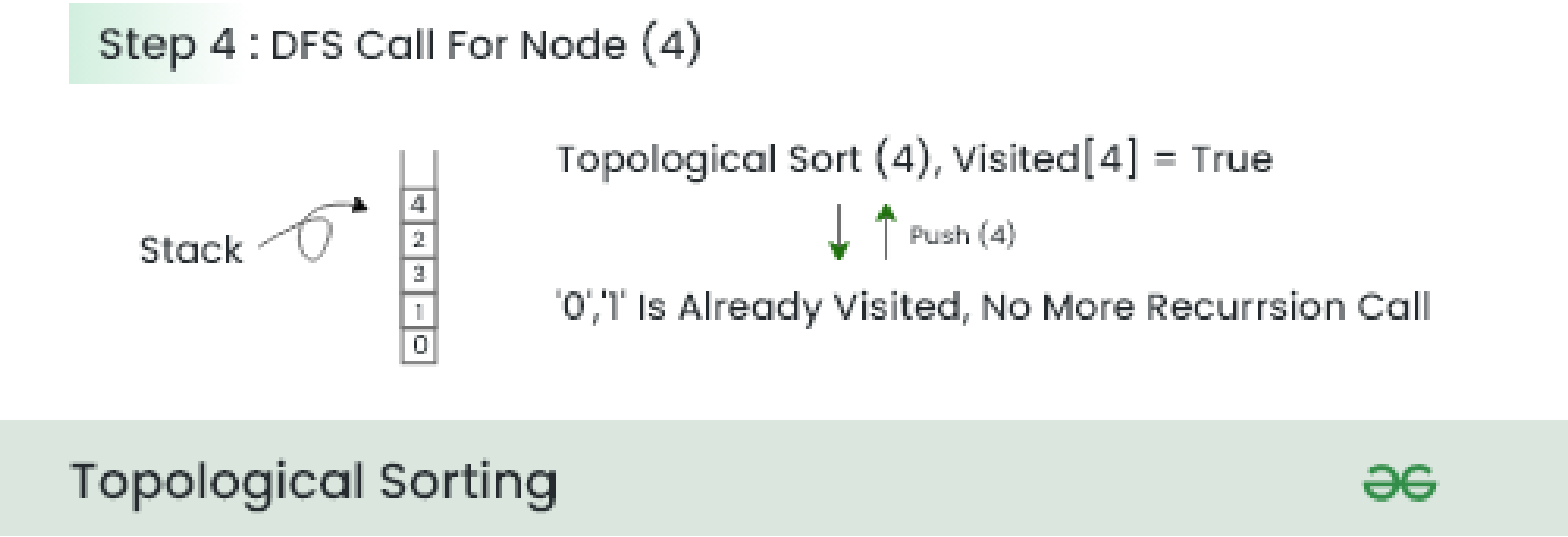
- 对于图中每个未访问的顶点，执行以下操作：
  ① 调用DFS函数，并将该顶点作为参数传递
  ② 在DFS函数中，将顶点标记为已访问，并递归调用DFS函数，处理该顶点所有未访问的邻居
  ③ 一旦所有邻居都已被访问，将该顶点压入栈中
  ④ 当所有顶点都被访问完后，从栈中弹出元素并将其依次添加到输出列表中，直到栈为空——最终得到的列表为输出



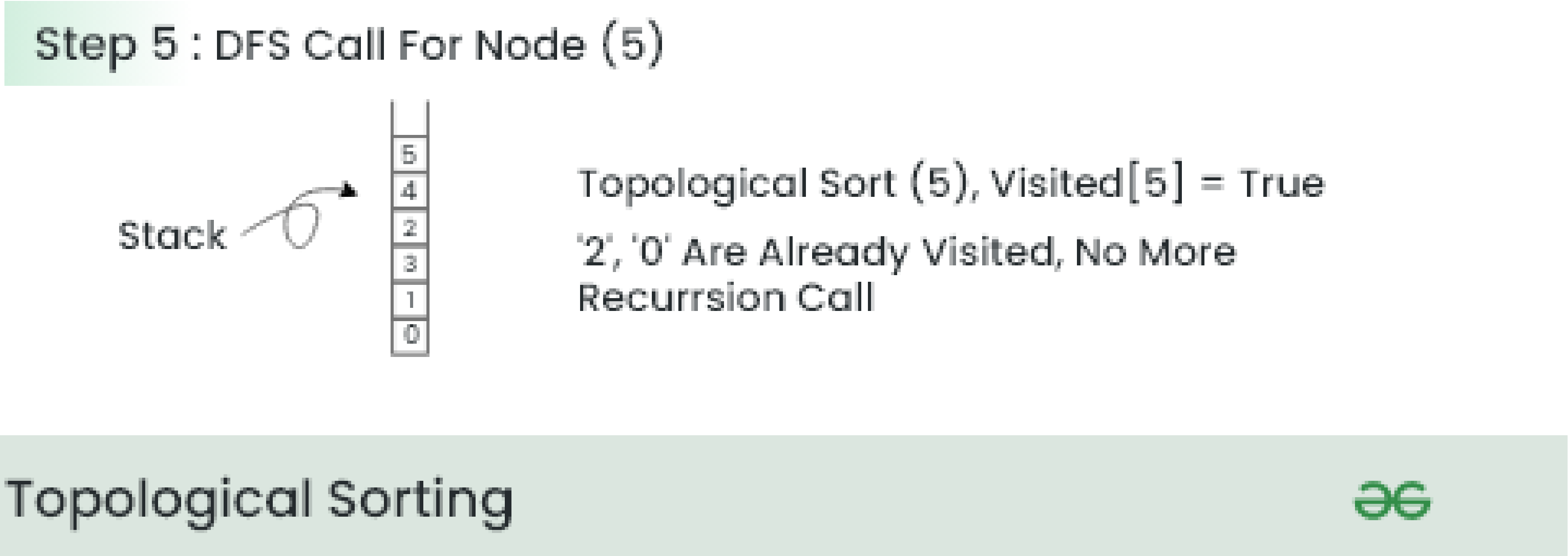
- 对于图中每个未访问的顶点，执行以下操作：
  ① 调用DFS函数，并将该顶点作为参数传递
  ② 在DFS函数中，将顶点标记为已访问，并递归调用DFS函数，处理该顶点所有未访问的邻居
  ③ 一旦所有邻居都已被访问，将该顶点压入栈中
  ④ 当所有顶点都被访问完后，从栈中弹出元素并将其依次添加到输出列表中，直到栈为空——最终得到的列表为输出



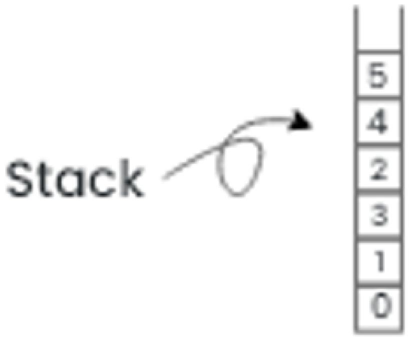
- 对于图中每个未访问的顶点，执行以下操作：
  ① 调用DFS函数，并将该顶点作为参数传递
  ② 在DFS函数中，将顶点标记为已访问，并递归调用DFS函数，处理该顶点所有未访问的邻居
  ③ 一旦所有邻居都已被访问，将该顶点压入栈中
  ④ 当所有顶点都被访问完后，从栈中弹出元素并将其依次添加到输出列表中，直到栈为空——最终得到的列表为输出



- 对于图中每个未访问的顶点，执行以下操作：
  ① 调用DFS函数，并将该顶点作为参数传递
  ② 在DFS函数中，将顶点标记为已访问，并递归调用DFS函数，处理该顶点所有未访问的邻居
  ③ 一旦所有邻居都已被访问，将该顶点压入栈中
  ④ 当所有顶点都被访问完后，从栈中弹出元素并将其依次添加到输出列表中，直到栈为空——最终得到的列表为输出

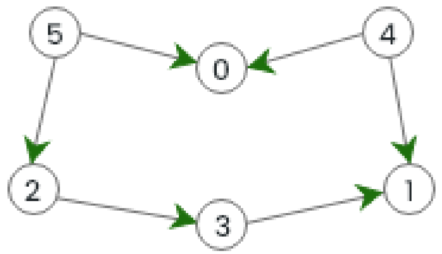


- 对于图中每个未访问的顶点，执行以下操作：
    ① 调用DFS函数，并将该顶点作为参数传递
    ② 在DFS函数中，将顶点标记为已访问，并递归调用DFS函数，处理该顶点所有未访问的邻居
    ③ 一旦所有邻居都已被访问，将该顶点压入栈中
    ④ 当所有顶点都被访问完后，从栈中弹出元素并将其依次添加到输出列表中，直到栈为空——最终得到的列表为输出



拓扑排序的最后一步，我们将**栈中的所有元素弹出**

最终output：5,4,2,3,1,0



Adjacency List (G)

0	{}
1	{}
2	{3}
3	{1}
4	{0,1}
5	{2,0}

	0	1	2	3	4	5
Visited	False	False	False	False	False	False

Step 1 : Topological Sort (0), Visited[0] = True

↓

List Is Empty. No More Recursion Call

Stack | 0 |

Step 2 : Topological Sort (1), Visited[1] = True

↓

List Is Empty. No More Recursion Call

Stack | 0 | 1 |

Step 3 : Topological Sort (2), Visited[2] = True

↓

Topological Sort (3), Visited[3] = True

↓

'1 Is Already Visited, No More Recurrsion Call

Stack | 0 | 1 | 3 | 2 |

Step 4 : Topological Sort (4), Visited[4] = True

↓

'0','1' Is Already Visited, No More Recurrsion Call

Stack | 0 | 1 | 2 | 3 | 4 |

Step 5 : Topological Sort (5), Visited[5] = True

'2', '0' Are Already Visited, No More Recurrsion Call

Stack | 0 | 1 | 2 | 3 | 4 | 5 |

Step 6 : Print All Elements Of Stack From Top To Bottom

Topological Sorting

拓扑排序Animation