

Topological Sort

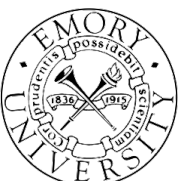
Data Structures and Algorithms

Emory University

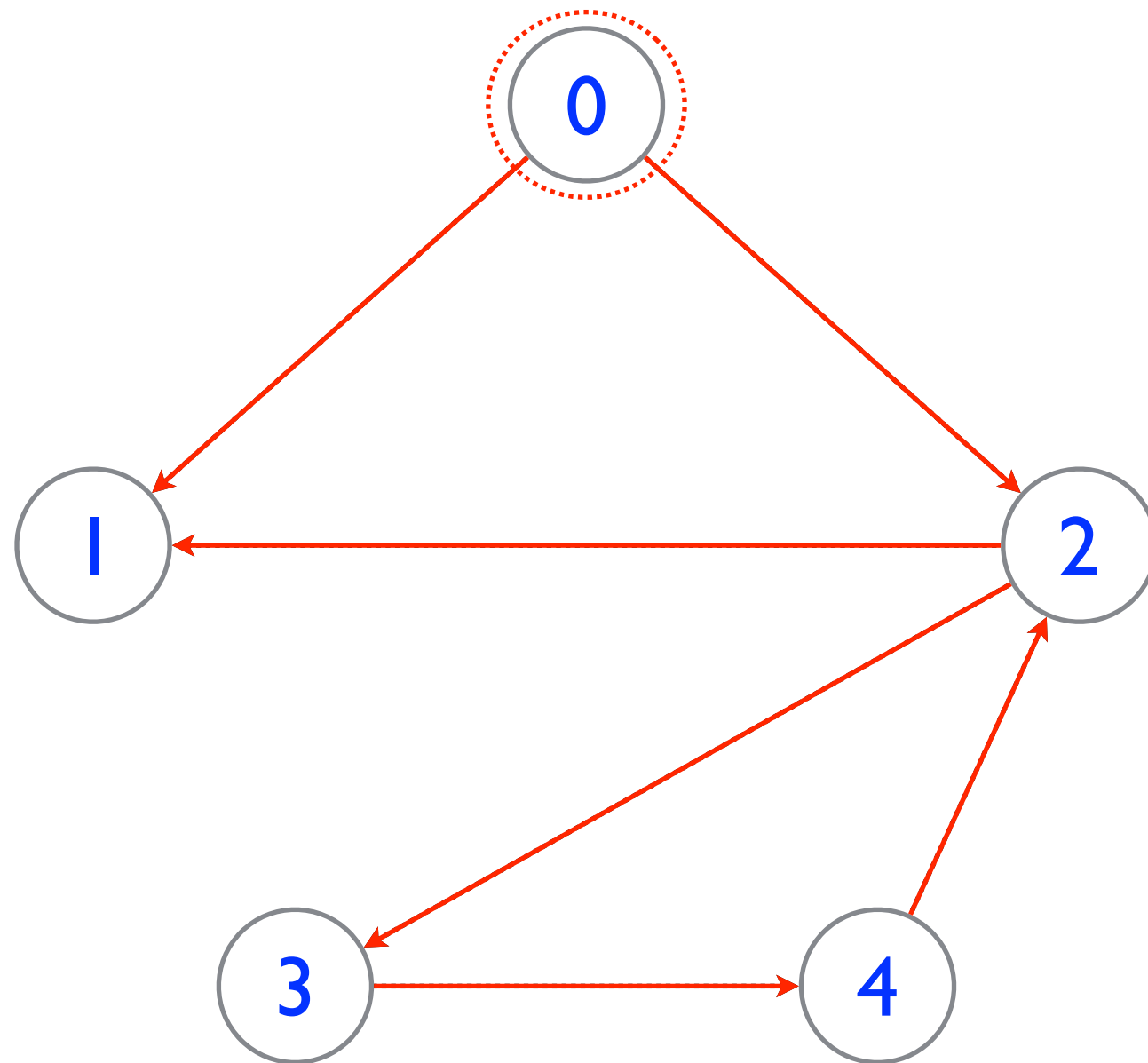
Jinho D. Choi



EMORY
UNIVERSITY



Cycle Detection



No incoming edge

No cycle

No cycle

Cycle!


Cycle Detection

```
public boolean containsCycle()
{
    Deque<Integer> notVisited = new ArrayDeque<>();
    for (int i=0; i<size(); i++) notVisited.add(i);

    while (!notVisited.isEmpty())
    {
        if (containsCycleAux(notVisited.poll(), notVisited, new HashSet<>()))
            return true;
    }

    return false;
}
```

`boolean containsCycleAux(int target,
Deque<Integer> notVisited,
Set<Integer> visited)`



Cycle Detection

```
private boolean containsCycleAux(int target, Deque<Integer> notVisited,
                                Set<Integer> visited)
{
    notVisited.remove(target);
    visited.add(target);

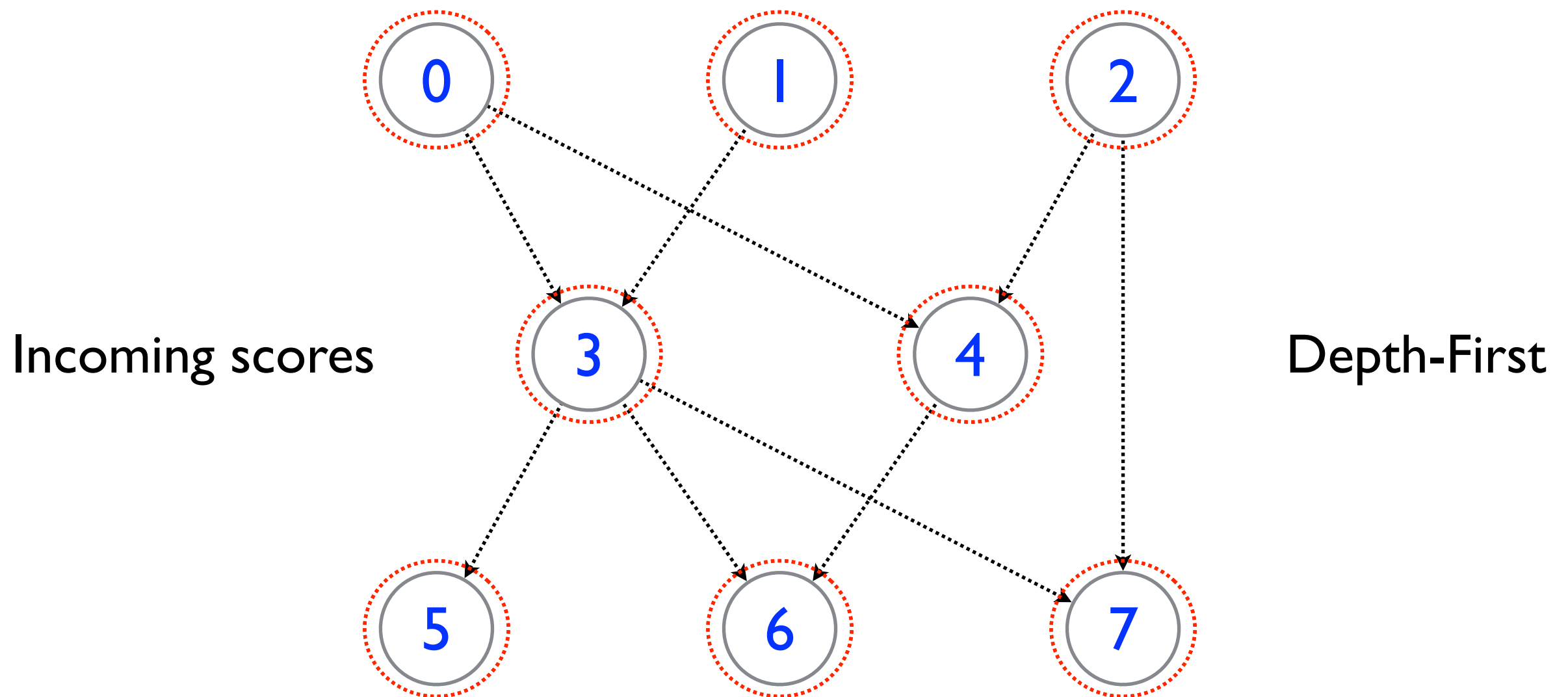
    for (Edge edge : getIncomingEdges(target))
    {
        if (visited.contains(edge.getSource()))
            return true;

        if (containsCycleAux(edge.getSource(), notVisited, new HashSet<>(visited)))
            return true;
    }

    return false;
}
```

Topological Sort

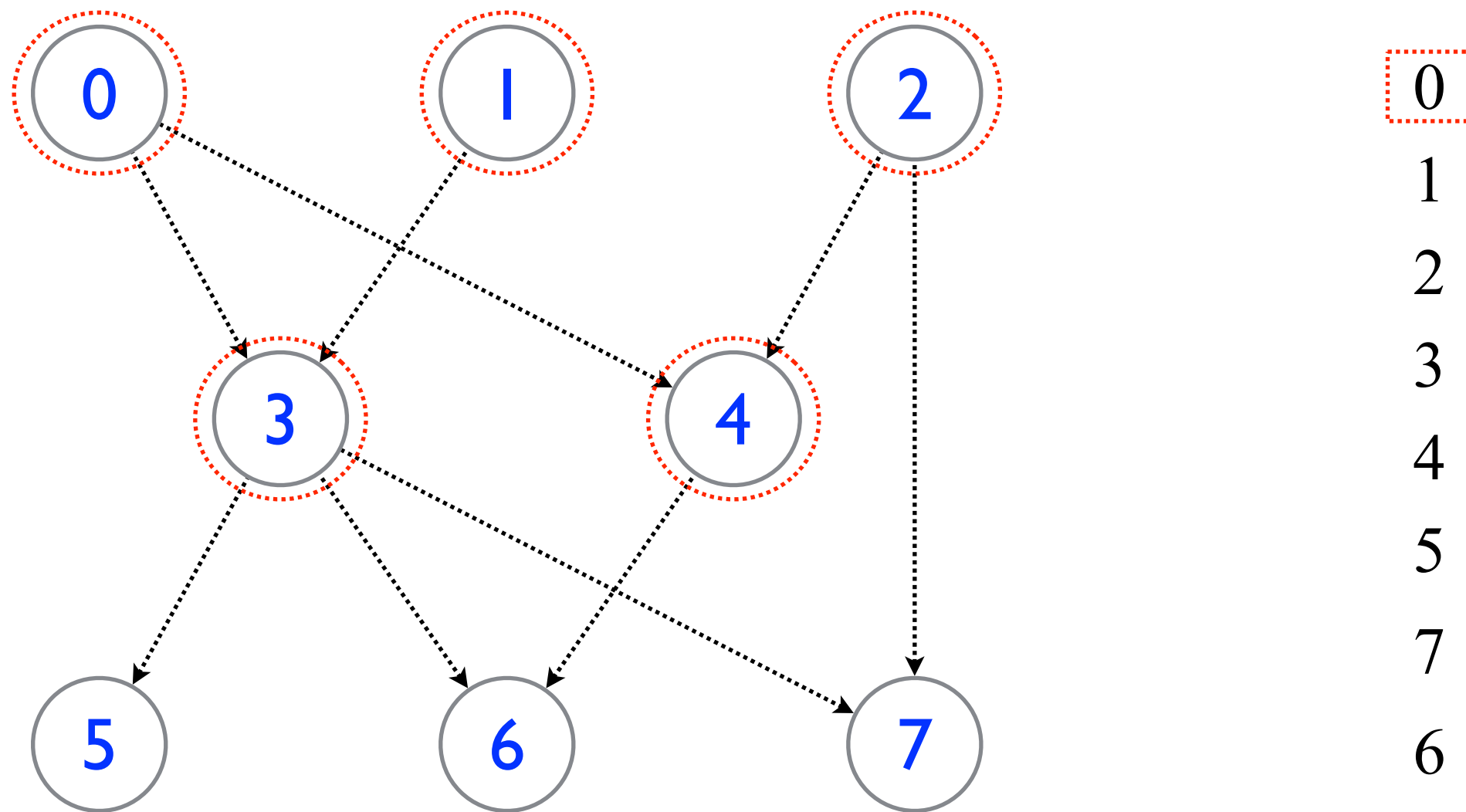
Sort vertices in a linear order such that for each vertex, all vertices associated with its incoming edges must appear first.



Sort by Incoming Scores

Keep adding vertices with no incoming edge.

Incoming score of each vertex = Σ of all source vertices



```

public List<Integer> sort(Graph graph)
{
    Deque<Integer> global = graph.getVerticesWithNoIncomingEdges();
    Deque<Edge>[] outgoingEdgesAll = graph.getOutgoingEdges();
    List<Integer> order = new ArrayList<Integer>();

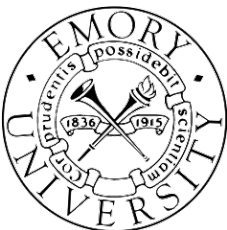
    while (!global.isEmpty())
    {
        Deque<Integer> local = new ArrayDeque<>();
        int vertex = global.poll(); order.add(vertex);
        Deque<Edge> outgoingEdges = outgoingEdgesAll[vertex];

        while (!outgoingEdges.isEmpty())
        {
            Edge edge = outgoingEdges.poll();
            List<Edge> incomingEdges = graph.getIncomingEdges(edge.getTarget());
            incomingEdges.remove(edge);
            if (incomingEdges.isEmpty()) local.add(edge.getTarget());
        }

        while (!local.isEmpty()) global.addLast(local.removeFirst());
    }

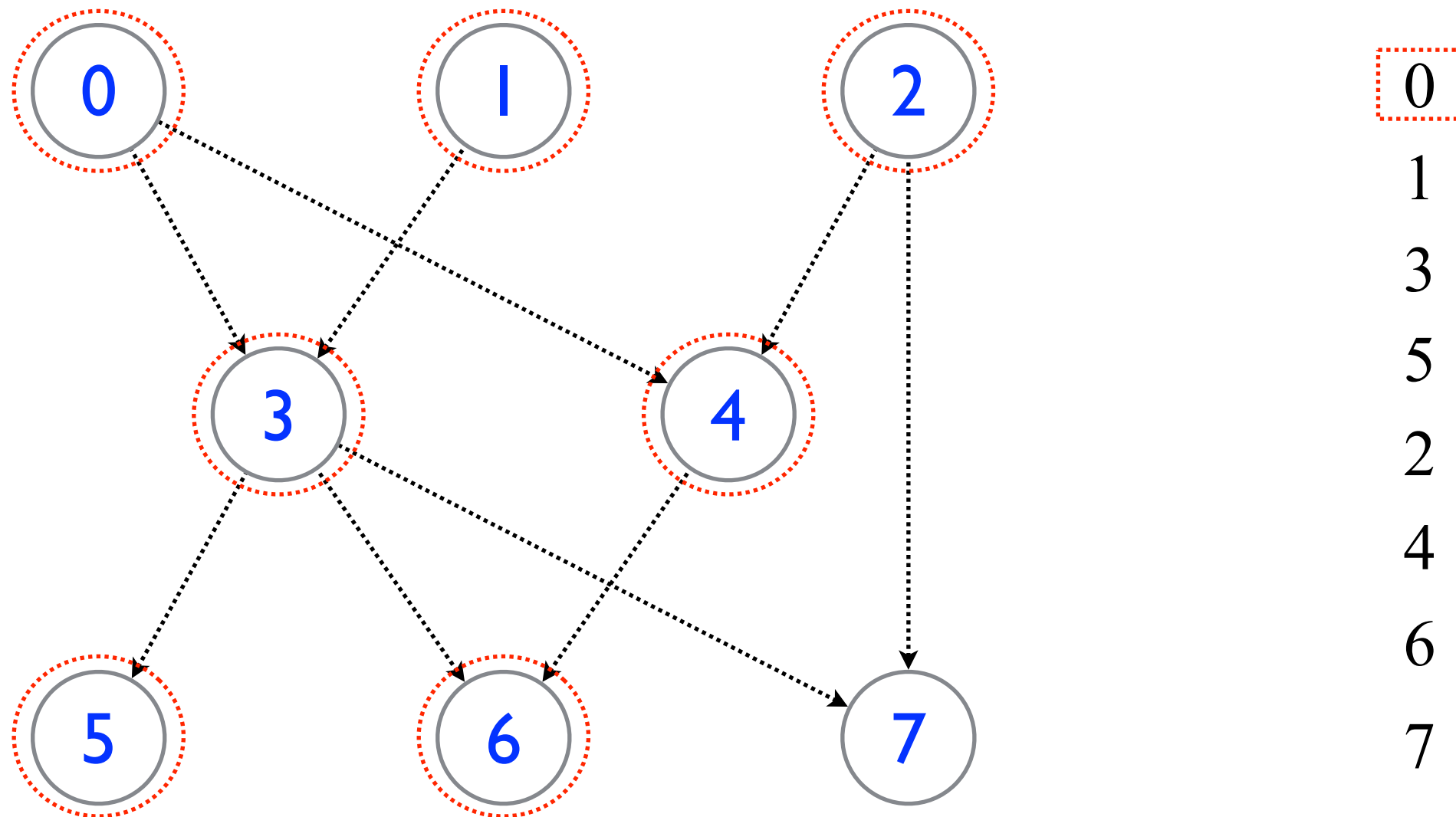
    if (!graph.isEmpty()) throw new IllegalArgumentException("Cyclic graph.");
    return order;
}

```



Sort by Depth-First

Keep adding vertices that are illegible.



0

1

3

5

2

4

6

7


```

public List<Integer> sort(Graph graph)
{
    Deque<Integer> global = graph.getVerticesWithNoIncomingEdges();
    Deque<Edge>[] outgoingEdgesAll = graph.getOutgoingEdges();
    List<Integer> order = new ArrayList<Integer>();

    while (!global.isEmpty())
    {
        Deque<Integer> local = new ArrayDeque<>();
        int vertex = global.poll(); order.add(vertex);
        Deque<Edge> outgoingEdges = outgoingEdgesAll[vertex];

        while (!outgoingEdges.isEmpty())
        {
            Edge edge = outgoingEdges.poll();
            List<Edge> incomingEdges = graph.getIncomingEdges(edge.getTarget());
            incomingEdges.remove(edge);
            if (incomingEdges.isEmpty()) local.add(edge.getTarget());
        }

        while (!local.isEmpty()) global.addFirst(local.removeLast());
    }

    if (!graph.isEmpty()) throw new IllegalArgumentException("Cyclic graph.");
    return order;
}

```

