1. Apply the strong components algorithm to the digraph that has the following adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Your solution should show the depth-first spanning forest and vertex numbering generated by the first depth-first search. It should also show the depth-first spanning forest generated by the second DFS.

Figure 1 shows one possible diagram for the graph. From the diagram it’s clear that the graph has four strong components: \( \{0, 1, 2, 3\}, \{4\}, \{5\}, \{6\} \).

![Figure 1: The Directed Graph](image)

Figure 2 shows a depth-first spanning forest. Outside each vertex is the number assigned by the algorithm. There are other possibilities. The figure only shows tree edges.

Figure 3 shows the second depth-first spanning forest. Once again there are other possibilities, although any spanning forest must have the same vertex sets in each tree of the forest.

2. Show the shortest paths generated by running Dijkstra’s algorithm on the graph of Figure 7.25 [in the text], beginning at Vertex 4. Show the D values as each vertex is processed as in Figure 7.17 [in the text].
Figure 2: The First Depth-First Spanning Forest

Figure 3: The Second Depth-First Spanning Forest

Figure 4 shows the graph from Figure 7.25 (although it’s laid out somewhat differently). The edges in the shortest paths are solid lines. Other edges are dashed lines.

Tables 1 and 2 show the changes in the $d$ values and the $p$ values. A dash indicates that the vertex has been marked.
3. Write an algorithm to determine whether an undirected graph of $|V|$ vertices contains a cycle. Your algorithm should run in $\Theta(|V|)$ time.

The excerpts from a Graph class on the following pages contain a public wrapper method for finding a cycle. The private method returns an int instead of a boolean. This makes it relatively easy to print the actual cycle. When the return value is a valid vertex, the current vertex is part of the cycle and can print itself before returning. When the return value is either NO_CYCLE_FOUND or CYCLE_FOUND, the calling vertex is not part of the cycle.
public class Graph {
    private int vertexCount;
    private int[][] adjMatrix;
    private boolean[] visited;
    private int how_many_visited;
    private static int NO_PARENT = -1;
    private static int NO_CYCLE_FOUND = -1;
    private static int CYCLE_FOUND = -2;

    // Method
    // containsCycle
    // Purpose
    // Wrapper for recursive containsCycle
    public boolean containsCycle() {
        for (int v = 0; v < vertexCount; v++)
            visited[v] = false;
        how_many_visited = 0;
        int rv;
        for (int v = 0; v < vertexCount; v++)
            if (!visited[v]) {
                rv = containsCycle(NO_PARENT, v);
                if (rv == CYCLE_FOUND) {
                    System.out.println();
                    return true;
                } else if (how_many_visited == vertexCount) {
                    return false;
                }
            }
        return false; // never executed
    }

    // Method
    // containsCycle
    // Purpose
    // Execute DFS looking for a back edge
    // Return value
    // NO_CYCLE_FOUND: if no cycle was found;
    // w: a vertex is the first vertex in the spanning tree on
    // the cycle
    // CYCLE_FOUND: current vertex is a parent or ancestor
    // of the first vertex in the spanning tree.
    private int containsCycle(int parent, int v) {
        int rv;
        visited[v] = true;
        how_many_visited++;
        if (how_many_visited == vertexCount)
return NO_CYCLE_FOUND;
for (int w = 0; w < vertexCount; w++)
    if (adjMatrix[v][w] != 0 && w != v && w != parent) {
        if (visited[w]) { // We found a back edge
            System.out.print(w + " ");
            return w;
        } else { // Make a recursive call
            rv = containsCycle(v, w);
            if (rv >= 0) {
                System.out.print(v + " ");
                if (rv == v)
                    return CYCLE_FOUND;
                else
                    return rv;
            } else if (rv == CYCLE_FOUND) {
                return CYCLE_FOUND;
            } else if (how_many_visited == vertexCount) {
                return NO_CYCLE_FOUND;
            } else {
                // NO_CYCLE_FOUND, but vertices remain to be visited
            }
        }
    }
return NO_CYCLE_FOUND;
} // containsCycle