1. Write and analyze a method, $\text{components}(G)$, for an undirected graph $G$, that returns a dictionary mapping each vertex to an integer that serves as an identifier for its connected component. That is, two vertices should be mapped to the same identifier if and only if they are in the same connected component.

**Algorithm.**

The algorithm (Algorithm 1) essentially performs a breadth first search over the entire graph. When a new component is encountered a label is given to the first vertex and then the rest of the component is labelled in a breadth-first-search manner (Algorithm 2).

### Algorithm 1 $\text{components}(G)$

**Input:** Undirected graph $G = (V, E)$

**Output:** Map between each vertex to its connected component label

1. Label $cc \leftarrow 0$
2. Map $ccs \leftarrow \emptyset$
3. for all $v \in V$ do
4. if $ccs.get(v) = \emptyset$ then
5. $\text{labelComponent}(G, v, ccs, cc)$
6. $cc \leftarrow cc + 1$
7. return $ccs$

### Algorithm 2 $\text{labelComponents}(G, s, ccs, cc)$

**Input:** Undirected graph $G$, Vertex $s$, Map between vertices and labels $ccs$, Label $cc$

1. Queue $Q \leftarrow \emptyset$
2. $Q.enqueue(s)$
3. while $\neg Q.isEmpty()$ do
4. $v \leftarrow Q.dequeue()$
5. for all $e \in G.outgoingEdges(v)$ do
6. $u \leftarrow G.opposite(e, v)$
7. if $ccs.get(u) = \emptyset$ then
8. $ccs.put(u, cc)$
9. $q.enqueue(u)$

**Time Complexity.**

**Theorem 1.** Algorithm 1 runs in $O(n + m)$ time on an undirected graph.
Proof. Let the map data structure be implemented as a resizable hash map and the queue data structure to be implemented as a linked list. All labeling and queue operations can be completed in $O(1)$ time. The overall algorithm will only visit each vertex and label it one time, totalling $O(n)$. The algorithm must visit each incident edge of all vertices, totalling $O(\Sigma_{v \in V} \text{deg}(v)) = O(m)$. Thus in total the algorithm takes $O(n + m)$ time.

Memory Complexity.

Theorem 2. Algorithm 1 uses $O(n)$ additional memory.

Proof. The queue and map data structures can contain a maximum of $O(n)$ elements in them. There is no other cause for extra memory usage, as the algorithm is non-recursive.

Note: Depth-first-search may also be used.
2. Show that, if $T$ is a BFS tree produced for a connected graph $G$, then, for each vertex $v$ at level $i$, the path of $T$ between $s$ and $v$ has $i$ edges, and any other path of $G$ between $s$ and $v$ has at least $i$ edges.

Solution.

**Theorem 3.** If $T$ is a BFS tree produced for a connected graph $G$, then, for each vertex $v$ at level $i$, the path of $T$ between $s$ and $v$ has $i$ edges, and any other path of $G$ between $s$ and $v$ has at least $i$ edges.

**Proof.** Assume the opposite – (1) the path of each vertex between $s$ and $v$ does not contain $i$ edges, and (2) any other path contains $i$ or fewer edges. (1) Essentially, when BFS discovers a vertex, that vertex is on level $i$. However, if a vertex on level $i$ had $j < i$ edges, then it would have had to have been discovered on level $j$. (2) On the other side, all other paths to $v$ having $j \leq i$ would imply that they would be the discovery path to $v$ in a BFS and thus been in $T$. Both arguments arrive at a contradiction. □
3. An Euler tour of a directed graph $G$ with $n$ vertices and $m$ edges is a cycle that traverses each edge of $G$ exactly once according to its direction. Such a tour always exists if $G$ is connected and the in-degree equals the out-degree of each vertex in $G$. Describe and analyze an $O(n + m)$-time algorithm for finding an Euler tour of such a directed graph $G$.

**Algorithm.**

The algorithm is described in Algorithm 3. Its basis is a depth first search. Because of the input, the depth first search can start from any vertex and be completed in one execution of the main loop. The modification is tracking a stack of a current cycle and adding it to the overall tour at the appropriate points (when a cycle is finished). The input is assumed to be a graph where in-degree is the same as out-degree for all nodes.

**Algorithm 3 Euler Tour**

**Input:** Directed Graph $G = (V, E)$

**Output:** Euler tour $T$ of vertices in cycle

1. $T \leftarrow \emptyset$
2. $G' = (V', E') \leftarrow G$
3. $v \leftarrow G'.\text{anyVertex}()$
4. Stack $S \leftarrow \emptyset$
5. repeat
6. if $G'.\text{outDegree}(v) = 0$ then
7. $T \leftarrow T \cup \{v\}$
8. $v \leftarrow S.\text{pop}()$
9. else
10. $S.\text{push}(v)$
11. $e \leftarrow G'.\text{anyOutgoingEdge}(v)$
12. $v \leftarrow G'.\text{opposite}(e)$
13. $E' \leftarrow E' \setminus \{e\}$
14. until $G'.\text{outDegree}(v) = 0 \land S.\text{isEmpty}()$
15. return $T$

**Time Complexity.**

**Theorem 4.** Algorithm 3 runs in $O(n + m)$ time on a directed graph.

**Proof.** The overall algorithm will only visit each edges once. The removal of edges can actually be implemented as labeling or in a hash set. Each vertex will be visited at least once because the graph is connected – $O(n)$. The algorithm must visit each incident edge of all vertices, totalling $O(\sum_{v \in V}\text{deg}(v)) = O(m)$. Thus in total the algorithm takes $O(n + m)$ time. \hfill \Box

**Memory Complexity.**

**Theorem 5.** Algorithm 3 uses $O(n + m)$ additional memory.

**Proof.** The stack will contain at most $O(n)$ nodes and the tour will contain $O(m)$ vertices because of the reasoning in the above proof. Thus in total there is $O(n + m)$ memory usage. \hfill \Box

**Note:** There are many ways to define and write this algorithm.
4. **Bonus.** An independent set of an undirected graph \( G = (V, E) \) is a subset \( I \) of \( V \) such that no two vertices in \( I \) are adjacent. That is, if \( u \) and \( v \) are in \( I \), then \( (u, v) \) is not in \( E \). A maximal independent set \( M \) is an independent set such that, if we were to add any additional vertex to \( M \), then it would not be independent any more. Every graph has a maximal independent set. (Can you see this? This question is not part of the exercise, but it is worth thinking about.) Give and analyze an efficient algorithm that computes a maximal independent set for a graph \( G \).

**Algorithm.**

The algorithm is quite simplistic and is described in Algorithm 4. Essentially, pick a vertex and remove all adjacent vertices. Repeat this until there are no vertices left. Every time a vertex is chosen add it to the independent set.

<table>
<thead>
<tr>
<th>Algorithm 4: Maximal Independent Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Graph ( G = (V, E) )</td>
</tr>
<tr>
<td><strong>Output:</strong> Maximal Independent Set ( I \subseteq V )</td>
</tr>
<tr>
<td>1: ( I \leftarrow \emptyset )</td>
</tr>
<tr>
<td>2: ( V' \leftarrow V )</td>
</tr>
<tr>
<td>3: while (<del>V'.isEmpty()</del>) do</td>
</tr>
<tr>
<td>4: ( v \leftarrow \text{anyVertex}(V') )</td>
</tr>
<tr>
<td>5: ( I \leftarrow I \cup {v} )</td>
</tr>
<tr>
<td>6: ( V' \leftarrow V' \setminus {v} )</td>
</tr>
<tr>
<td>7: for all ( u \in G.\text{outgoingEdges}(v) ) do</td>
</tr>
<tr>
<td>8: ( V' \leftarrow V' \setminus {u} )</td>
</tr>
<tr>
<td>9: return ( I )</td>
</tr>
</tbody>
</table>

**Time Complexity.**

**Theorem 6.** Algorithm 4 runs in \( O(n) \) time, where \( n \) is the number of vertices of \( G \).

*Proof. *All vertices will be visited once. Let the removal procedure actually be implemented by labeling or a hash set mechanism for constant time operations. Total is \( O(n) \). ∎

**Memory Complexity.**

**Theorem 7.** Algorithm 4 uses \( O(n) \) additional memory.

*Proof. *At most every node would contain a label and possibly be placed in \( I \). No other additional memory is used. Total is thus \( O(n) \). ∎