Solution 1. In each step, the breadth-first algorithm proceeds from a vertex (fetched from the front of Q) to the direct neighbours. It needs to keep track of which vertices are still to consider, stored in a queue Q, and the distance vector storing the current shortest distance to vertex 2. We can visualize its behavior, by showing the content of Q and the distance vector at each time step:

<table>
<thead>
<tr>
<th>Step</th>
<th>Queue</th>
<th>Distance vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2)</td>
<td>(∞,0,∞,∞,∞,∞)</td>
</tr>
<tr>
<td>2</td>
<td>(1,4)</td>
<td>(1,0,∞,1,∞,∞)</td>
</tr>
<tr>
<td>3</td>
<td>(4,3)</td>
<td>(1,0,2,1,∞,∞)</td>
</tr>
<tr>
<td>4</td>
<td>(3,5,6)</td>
<td>(1,0,2,1,2,2)</td>
</tr>
<tr>
<td>5</td>
<td>(5,6)</td>
<td>(1,0,2,1,2,2)</td>
</tr>
<tr>
<td>6</td>
<td>(6)</td>
<td>(1,0,2,1,2,2)</td>
</tr>
<tr>
<td>7</td>
<td>∅</td>
<td>(1,0,2,1,2,2)</td>
</tr>
</tbody>
</table>

The vertices are visited in the order (2,1,4,3,5,6)

Solution 2.
Consider the above graph:

- It has two weakly connected components, namely \{1, \ldots , 6\} and \{7, 8, 9\}.
- It has three strongly connected components, namely \{1, 2, 3\}, \{4, 5, 6\} and \{7, 8, 9\}.

The SCC \{1, 2, 3\} has no in-component, but the vertices 4, 5, 6 in its out-component.
The SCC \{4, 5, 6\} has the vertices 1, 2, 3 in its in-component, but no out-component.
The SCC \{7, 8, 9\} has empty in- and out-components.

Solution 3. Topological sorting: Kahn’s algorithm from Wikipedia

L := Empty list that will contain the sorted elements
S := Set of all nodes with no incoming edges
while S is non-empty do
remove a node \( n \) from \( S \)
add \( n \) to tail of \( L \)
for each node \( m \) with an edge \( e \) from \( n \) to \( m \) do
remove edge \( e \) from the graph
if \( m \) has no other incoming edges then
insert \( m \) into \( S \)
if graph has edges then
return error (graph has at least one cycle)
else
return \( L \) (a topologically sorted order)

Implementation in R:

- Graph given as adjacency matrix \( A \)

```r
# Graph given as adjacency matrix A
topo_sort <- function (g) {
  A <- t(as.matrix(as_adjacency_matrix(g)))

  count.incoming <- function (i) {
    # Convention: \( A_{ij} = 1 \) denotes edge \( j \rightarrow i \)
    sum(A[i,])
  }

  N <- nrow(A);
  L <- c();
  S <- Filter( function(i) { count.incoming(i) == 0 }, 1:N );
  while (length(S) > 0) {
    n <- S[1]; S <- S[-1];
    L <- c(L, n)
    for (m in 1:N) {
      if (A[m,n] > 0) {
        A[m,n] <- 0;
        if (count.incoming(m) == 0) {
          S <- c(m, S);
        }
      }
    }
  }
  if (sum(A) > 0) {
    stop('Not a DAG!');
  }
}
```
Analysis of runtime:

- count.incoming: $O(N)$
- Initialization of $S$: $O(N^2)$
- While loop runs $N$ times:
  - Appending to $L$: $O(N)$
  - For loop runs $N$ times and uses count.incoming: $O(N^2)$

Total runtime: $O(N^3)$

- Alternative algorithm: Based on depth-first search
- Best possible runtime: $O(N + M)$, i.e. linear

Finally, we should test our algorithm on some examples:

```r
library(igraph)
g_1 <- tribble(~from, ~to,  
               1, 2,  
               1, 3,  
               2, 3) %>%
               graph_from_data_frame()
g_2 <- tribble(~from, ~to,  
               1, c(2, 3, 6, 10),  
               2, c(3, 12),  
               3, c(4),  
               4, c(5, 6),  
               6, c(7, 8, 9, 10),  
               9, c(10),  
               10, c(11),  
               11, c(12, 13)) %>%
               unnest() %>%
               graph_from_data_frame()
plot(g_1, vertex.color = "lightblue", vertex.size = 25)
```
```r
topo_sort(g_1)
## + 3/3 vertices, named, from 36c4d43:
## [1] 1 2 3

plot(g_2, vertex.color = "lightblue", vertex.size = 25)
```
Solution 4. First, we install load some libraries and then read in the data:

```r
library(tidyverse)
library(igraph)

g <- "amazon0302.txt" %>%
```

```r
topo_sort(g_2)
## + 13/13 vertices, named, from f682f0:
## [1] 1 2 3 4 5 6 8 7 9 10 11 13 12
```

```r
```
• Then, we look at the empirical degree distribution:

```r
deg_df <- tibble(prob=degree_distribution(g)) %>%
  mutate(k=seq.int(from=0, along.with=prob))
```

```
## Warning: Transformation introduced infinite values in continuous x-axis
## Warning: Transformation introduced infinite values in continuous y-axis
```

```r
deg_df %>%
ggplot(aes(k, prob)) +
geom_point() +
scale_x_log10() +
scale_y_log10()
```
As explained in the lecture to investigate its tail it is usually better to plot the cumulative empirical degree distribution:

```r
deg_df <- deg_df %>%
  mutate(ecdf = 1 - cumsum(prob))

deg_df %>%
  ggplot(aes(k, ecdf)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()
```

```
## Warning: Transformation introduced infinite values in continuous x-axis
```
• Now we are ready to load the poweRlaw library and fit a (discrete) power law distribution to the observed degrees:

```r
library(poweRlaw)

deg_seq <- g %>%
  degree(mode = "all")

dpl <- displ$new(deg_seq)

fit <- estimate_xmin(dpl)
fit
```
Thus, the library would choose $x_{\text{min}} = 35$ for the following fit:

```r
dpl$xmin = fit$xmin
dpl$pars = fit$pars

plot(dpl)
lines(dpl, col = "blue", lwd = 2)
```
where we have used the built-in methods for plotting the cumulative distribution of the data and fitted power law.

To investigate if this cutoff is a reasonable choice, we can visualize how the fitted tail exponent $\alpha$ changes with $x_{\text{min}}$:

```r
tibble(cutoff = seq(min(deg_seq), 0.9 * max(deg_seq), by = 1),
alpha = map_dbl(cutoff,
  function(x_cut) {
    dpl$xmin = x_cut
    estimate_pars(dpl)$pars
  }))
```
Finally, we fit exponential and log-normal distributions and compare the results with our power law:

```
dpl$xmin = fit$xmin
dpl$pars = fit$pars

### Fit discrete exponential distribution
dexp <- disexp$new(deg_seq)
dexp$xmin <- fit$xmin
dexp$pars <- estimate_pars(dexp)

### Fit discrete log-normal
dln <- dislnorm$new(deg_seq)
dln$xmin <- fit$xmin
dln$pars <- estimate_pars(dln)
```
The exponential fit is obviously not good, but the discrete power law and discrete log-normal are visually indistinguishable. Indeed, a likelihood ratio test reveals that we cannot reject the null hypothesis that both fits are equally good.

```r
comp <- compare_distributions(dpl, dln)
## p-value for dpl being better
comp$p_one_sided
```
## 0.5020309

## p-value for any model being better
comp$p_two_sided

## [1] 0.9959382

## Detailed comparison of likelihood ratio per data point
comp$ratio %>%
group_by(x, ratio) %>%
summarize(cnt = n()) %>%
ggplot(aes(x, ratio,
    color = ratio >= 0,
    size = cnt)) +
geom_point(alpha = 0.6) +
scale_color_discrete(name = "Test result",
    labels = c("log-normal", "power law")) +
labs(x = "k", y = "Likelihood ratio", size = "Count") +
scale_x_log10()