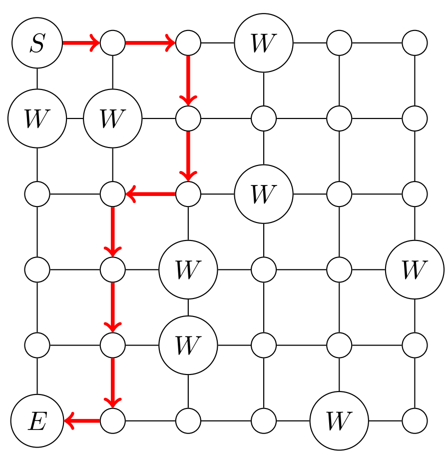
Problem 1

(a) *Explored* array at the beginning and at the end of each call of the DFS function is shown below:

	Beginning of the call	end of the call
DFS(1)	[0,0,0,0,0]	[1,1,1,1,1]
DFS(2)	[1,0,0,0,0]	[1,1,1,0,1]
DFS(3)	[1,1,0,0,0]	[1,1,1,0,0]
DFS(5)	[1,1,1,0,0]	[1,1,1,0,1]
DFS(4)	[1,1,1,0,1]	[1,1,1,1,1]

(b)
Explored $\leftarrow$ array of all zeros
Function DFS *(starting vertex s, parent p)*
Explored[s] $\leftarrow 1$
foreach (s, v) in E **do**
  if v is not explored **then**
    $cycle$ =DFS(v, s)
    if $cycle$ is $None$ **then**
      return $cycle$.append(s)
    end if
  else
    if v is not equal to p **then**
      return (s, v)
    end if
  end if
end foreach
return $None$

Problem 2

(a)


(b) Breadth-first search is best for finding a path of the shortest length, as it explores all neighbors at a depth uniformly. Therefore, as soon as it reaches the exit, it will be the shortest path.

$Unexplored \leftarrow$ queue of unexplored nodes
$Visited \leftarrow$ set of explored nodes
$Parents \leftarrow$ dictionary of node parents
Function BFS *(starting vertex s, graph $G = (V, E)$)*
$Unexplored$.enqueue(s)
//nodes must be added to $Visited$ at the same time they are enqueued,
//so that they don't get enqueued again later
$Visited$.add(s)
$Parents[s] = None$
while $Unexplored$ is not empty **do**
    $neighbor \leftarrow$ dequeue $Unexplored$
    if $neighbor$ is exit **then**
        $path = []$
        while $neighbor$ is not $None$ **do**
            $path$.append($neighbor$)
            $neighbor = Parents[neighbor]$
        end while
        Return reverse($path$)
    end if
    foreach $(neighbor, v)$ in E **do**
        if v is not in $Visited$ and not a wall **then**
            $Unexplored$.enqueue(v)
            $Visited$.add(v)
            $Parents[v] = neighbor$
        end if
    end foreach
end while
Return $None$

(c) This algorithm is modified from the previous example because the algorithm was returning as soon as it reached an exit. Instead, continue iterating through every node updating the furthest exit. After every node has been visited, return the path to the exit that is the furthest away from the start.

$Unexplored \leftarrow$ queue of unexplored nodes
$Visited \leftarrow$ set of explored nodes
$Parents \leftarrow$ dictionary of node parents
$Distance \leftarrow$ dictionary of node distance
$furthestExit \leftarrow$ None
$maxDistance \leftarrow 0$
Function BFS *(starting vertex s, graph $G = (V, E)$)*
$Unexplored$.enqueue(s)
$Visited$.add(s)
$Parents[s] \leftarrow None$
$Distance[s] \leftarrow 0$
while $Unexplored$ is not empty **do**
    $neighbor \leftarrow$ dequeue $Unexplored$
    if $neighbor$ is exit **then**
        if $Distance[neighbor] > maxDistance$ **then**

$maxDistance \leftarrow Distance[neighbor]$
$furthestExit \leftarrow neighbor$
end if
end if
foreach $(neighbor, v)$ in E **do**
if v is not in $Visited$ and not a wall **then**
$Unexplored$.enqueue(v)
$Visited$.add(v)
$Parents[v] \leftarrow neighbor$
$Distance[v] \leftarrow Distance[neighbor] + 1$
end if
end foreach
end while
if $furthestExit$ is not $None$ **then**
$path \leftarrow []$
$neighbor \leftarrow furthestExit$
while $neighbor$ is not $None$ **do**
$path$.append($neighbor$)
$neighbor \leftarrow Parents[neighbor]$
end while
Return reverse($path$)
end if
Return $None$

Problem 3

We can find the optimal community by using BFS to determine the size of each community and choosing the smallest community consisting of at least 1500 people:

$Unexplored \leftarrow$ Queue of unexplored nodes (initially empty)
$Visited \leftarrow$ Set of all visited nodes (initially empty)
$The\_chosen\_community\_size = Infinity$
$The\_chosen\_community = None$
Function BFS ($starting\ vertex\ s$, $Graph\ G = (V, E)$)
$Unexplored$.enqueue(s)
$Visited$.add(s)
$size = 0$
while $Unexplored$ is not empty **do**
$current\_node \leftarrow Unexplored$.pop(0)
$size += 1$
foreach $neighbor$ in $Graph(current\_node)$ **do**
if $neighbor$ not in $Visited$ **then**
$Visited$.add($neighbor$)
$Unexplored$.append($neighbor$)
end if
end foreach
end while
return $size$

Problem 3 continued on next page. . .

foreach $person$ in $Graph$ **do**
  if $person$ not in $Visited$ **then**
    $community\_size$ = BFS ($person$)
    if $community\_size > 1499$ **AND** $community\_size < The\_chosen\_community\_size$ **then**
      $The\_chosen\_community\_size = community\_size$
      $The\_chosen\_community$ = community which contains $person$
    end if
  end if
end foreach
return $The\_chosen\_community$

Problem 4

(a) **Input:** $G = (V, E)$ represented as adjacency list
Output: $inDegree \leftarrow$ array of in-degrees for each vertex
$inDegree \leftarrow$ array of size V initialized to all zeros
foreach each vertex v in V **do**
  foreach each neighbor u of v **do**
    $inDegree[u] \leftarrow inDegree[u] + 1$
  end foreach
end foreach
return $inDegree$

Runtime complexity: $O(V + E)$

(b) **Input:** $G = (V, E)$ represented as adjacency list
Output: $outDegree \leftarrow$ array of out-degrees for each vertex
$outDegree \leftarrow$ array of size V initialized to all zeros
foreach each vertex v in V **do**
  foreach each neighbor u of v **do**
    $outDegree[v] \leftarrow outDegree[v] + 1$
  end foreach
end foreach
return $outDegree$

Runtime complexity: $O(V + E)$

(c) **Input:** $G = (V, E)$ represented as adjacency matrix
Output: $inDegree \leftarrow$ array of in-degrees for each vertex
$inDegree \leftarrow$ array of size V initialized to 0
foreach each column j from 0 to $|V| - 1$ **do**
  foreach each row i from 0 to $|V| - 1$ **do**
    if $G[i][j] = 1$ **then**
      $inDegree[j] \leftarrow inDegree[j] + 1$
    end if
  end foreach
end foreach
return $inDegree$

Problem 4 continued on next page. . .

Runtime complexity: $O(V^2)$

(d) **Input:** $G = (V, E)$ represented as adjacency matrix
Output: $outDegree \leftarrow$ array of out-degrees for each vertex
$outDegree \leftarrow$ array of size V initialized to 0
foreach each row i from 0 to $|V| - 1$ **do**
    foreach each column j from 0 to $|V| - 1$ **do**
        if $G[i][j] = 1$ **then**
            $outDegree[i] \leftarrow outDegree[i] + 1$
        end if
    end foreach
end foreach
return $outDegree$

Runtime complexity: $O(V^2)$

Problem 5

We can use the idea of backtracking. First, we count the in-degree of each vertex, then we use recursive calls of the backtracking function to find all the sortings. The pseudocode is described below:

$V \leftarrow$All vertices
$visited \leftarrow$Empty list
$all_sorts \leftarrow$Empty list
$current_sort \leftarrow$Empty list
Function backtrack($visited$)
if $visited$.size= V.size **then**
    all_sorts.append($current\_sort$)
    return
end if
foreach v in V **do**
    if v.in_degree= 0 **AND** v is not in $visited$ **then**
        $current\_sort.append(v)$
        $visited.append(v)$
        foreach neighbor n of v **do**
            n.in_degree= n.in_degree-1
        end foreach
        backtrack($visited$)
        Remove v from $current\_sort$
        Remove v from $visited$
        foreach neighbor n of v **do**
            n.in_degree= n.in_degree$+1$
        end foreach
    end if
end foreach
return all_sorts