Name: - Olohi Goodness John
Peers: - Catherine Weeks, Sanjana Yasna
References: - https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-breadth-first-search-bfs/

Part 6: Properties of BFS

First, write out BFS in formal pseudocode (see Part 2: BFS for details).

If you completed the first step correctly, then the following property should hold. For every undirected or directed graph G = (V,E) in adjacency-list representation and for every starting vertex s in V:

- At the conclusion of BFS, a vertex v in V is marked as explored if and only if there is a path from s to v in G.
- The running time of BFS is O(m + n), where m = $\|E\|$ and $n = \|V\|$.

For each property above, write an argument in English (or informal proof) as to why (or why not) the property holds. You must give reference to specific line numbers in your pseudocode algorithm. If you are unsure how to start, try explaining the property to a friend or stuffy.

```
1. def findBFSPath(graph, start_node, end_node)
2.    Let queue be a Queue
3.    Let visited be a Hashmap
4.    queue.append(start_node)
5.    while queue:
6.        popped = queue.popleft()
7.        if popped == end_node:
8.            return path(visited, popped, start_node)
9.        else:
10.            for value in graph[popped]:
11.                if value not in visited:
12.                    visited[value] = popped
13.                    queue.append(value)
14.    return None

15. def path(visited, popped, start)
16.    Let result be a new queue
```

```
17.     current = popped
18.     while current != start:
19.         result.append(current)
20.         current = visited[current]
21.     result.append(start)
22.     # reverse the list and return
23.     left = 1
24.     right = result.length
25.     while left <= right:
26.         temp = result[left]
27.         result[left] = result[right]
28.         result[right] = temp
29.         left += 1
30.         right -= 1
31.     return result
```

1. At the conclusion of BFS, a vertex v in V is marked as explored if and only if there is a path from s to v in G.

   The algorithm, shown in the pseudocode lines $1 - 31$, portrays that a node, v, on the path of s can be gotten to by performing a breadth-first search starting from s. This implies that for a node to be on the path of s, it must either be a neighbor of s or connected to s through a neighbor (or neighbor of neighbors of ...) of s. For our purpose, we shorten this concept to: g is connected to s through a neighbor relationship.

   As seen on lines $3, 19, 20$, nodes are entered into the set tracker, **visited**, only if they are connected to s through a neighbor relationship. This means that if s and v are connected, v is entered into the visited tracker. Bidirectionally, if v is visited, then it must be connected to s through a neighbor relationship, which is the definition of a path, a connection between two nodes. Therefore, a path exists between s and v if and only if it is entered into the visited tracker (marked as visited).

2. The running time of BFS is $O(m + n)$, where m = $\|E\|$ and $n = \|V\|$.

   The BFS involves first, iterating through every vertex connected to the start node (lines $2 - 13$). Therefore in the case of connected graphs, all vertices are explored once, constituting, $O(n)$ time. If the target or end node is found, the path() function on line 15 is called, whose time complexity we consider trivial because it's only called once.

For every node, the BFS then iterates through its neighbors, exploring all edges in the graph (in the worst case scenario of a connected graph). This gives an added time complexity is $O(m)$.
As such, the total time complexity is $O(m + n)$.

Part 7: Properties of ISCONNECTED

Consider the isConnected function your wrote above. If you implemented it correctly, then the following property should hold. For every undirected graph G = (V,E) in adjacency-list representation:

- The function returns true if and only if for every pair u, v of vertices in V, there is a path from u to v and from v to u (i.e., u and v are connected components).
- The running time of your algorithm is O(m + n), where $m = \|E\|$ and $n = \|V\|$.

For each property above, write an argument in English (or informal proof) as to why (or why not) the property holds. You must give reference to specific line numbers in your code. To make it easier, we recommend copying relevant code blocks from the programming part into your Pseudocode file (use verbatim blocks if you are using LaTeX) AND adding line numbers.

```
1.   def isConnected(graph):
2.       start = list(graph)[0]
3.        Let queue be a deque()
4.        Let set be a Set()
5.        queue.append(start)
6.        while (queue):
7.            popped = queue.popleft()
8.            for n in graph[popped]:
9.                if n not in set:
10.                   queue.append(n)
11.                   set.add(n)
12.        return len(set) == len(graph)
```

1. The function returns true if and only if for every pair u, v of vertices in V, there is a path from u to v and from v to u (i.e., u and v are connected components).

   The algorithm shown involves checking, from a start vertex (line 2), *key*, if a path exists from *key* to every other vertex in the graph.

It does this by traversing through all nodes reachable from the first element in the graph and comparing the count of all nodes reached with the length of the graph (lines $6-12$). If a graph is disconnected, then some vertices in its adjacency list won't be reached from key. If, however, all vertices are reached (implied by the length of the set tracker, set, being equal to the length of graph) from the first element, then we can be sure there's no "break" in the graph. Therefore, all elements are connected and it returns True. Otherwise it returns False. Bidirectionally, we say that if the function returns True, then the length of the set tracker and graph are equal ($line12$), which implies that every node in the graph was visited, which further implies that the graph is connected.
Therefore,

"The function returns true if and only if for every pair u, v of vertices in V, there is a path from u to v and from v to u (i.e., u and v are connected components)."

2. The runtime of this algorithm is $O(m+n)$ because, in the worst case scenario of a connected graph, it enqueues every vertex exactly once, accounting for $O(n)$ time. For every node, it then checks its neighbors, which is same as checking every edge connected to the node. This implies that it checks every edge exactly once for directed graphs and twice for undirected graphs if the graph is connected. This accounts for $O(m)$ time. Therefore, the total time complexity is $O(m+n)$.

   *** Note: I consulted GeekforGeeks to better understand how/why we can say the edge is processed exactly once or twice. See link in the file header ***

Part 8: Properties of TOPOSORT

Consider the topoSort function you wrote above. If you implemented the topoSort function correctly, then the following property should hold. For every directed acyclic graph G = (V,E) in adjacency-list representation: (a) At the conclusion of topoSort, every vertex v has been assigned a position in the final list and this list constitutes a topological ordering of G. (b) The running time of topoSort is O(m + n), where $m = |E|$ and $n = |V|$.

For each property above, write an argument in English (or informal proof) as to why (or why not) the property holds. You must give reference to specific line numbers in your code. To make it easier, we recommend copying relevant code blocks from the programming part into your Pseudocode file (use verbatim blocks if you are using LaTeX, see above) AND adding line numbers.

```
1. def topoSort(graph):
2.     Let sorted be a Queue
3.     Let visited be a Set
4.     for key in graph:
5.         if key not in visited:
6.             visited.add(key)
7.             helperTopo(graph, key, sorted, visited)
8.     return sorted

9. def helperTopo(graph, key, visited) -> None:
10.     neighbors = graph[key] if key in graph else []
11.     for n in neighbors:
12.         if n not in visited:
13.             visited.add(n)
14.             helperTopo(graph, n, sorted, visited)
15.     sorted.appendleft(key)
```

1. At the conclusion of topoSort, every vertex v has been assigned a position in the final list and this list constitutes a topological ordering of G.
   The algorithm for topological sorting, pseudocode shown above, involves performing a depth-first search on **every** unvisited vertex in the graph, and its neighbors. The depth-first search ends with the node under iteration being added to the final list. This ensures that every vertex in the graph, connected or not, is visited and added to the list.
   A topological ordering occurs when all child nodes come after their parent nodes. The algorithm visits in reverse order: it explores all the child nodes of a key, before adding the key to the list (lines $11 - 14$). However, because keys are added to the front of the list (line 15), the algorithm returns the list in reverse order to how they were visited. As such, all child nodes appear after their parents nodes in the returned list.
   Therefore, at the conclusion of the sort, "every vertex v has been assigned a position in the final list and this list constitutes a topological ordering of G."

2. The running time of topoSort is O(m + n), where $m = \|E\|$ and $n = \|V\|$.
   ** Note: For the purposes of this proof, we do not consider the extra time complexity caused by the "appendleft()" function. ***
   In general, the algorithm iterates through every vertex in the array, which accounts for $O(n)$ runtime complexity, where n is the number of vertices in the graph, G. It also explores the edges for every vertex, exactly once, since the graph is directed. This

accounts for an additional $O(m)$ complexity, where m is the number of edges in G. Therefore, in total, the running time of topoSort is $O(m + n)$ where $m = \|E\|$ and $n = \|V\|$.