Name: - Olohi Goodness John
Peers: - Catherine Weeks
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.

```
# Algorithm implemented in part 2
1. def isConnected(graph) -> bool:
2.     for key in graph:
3.         for sec_key in graph:
4.             if key == sec_key:
5.                 continue
6.             if findPathHelper(graph, key, sec_key) is False:
7.                 return False
8.     return True

9. def findPathHelper(graph, node1, node2) -> bool:
10.     Let path_queue be a Queue
11.     Let hash_set be a Set
12.     path_queue.append(node1)
13.     hash_set.add(node1)
14.     while path_queue:
15.         popped = path_queue.popleft()
```

```
16.         if popped == node2:
17.             return True
18.         for value in graph[popped]:
19.             if value not in hash_set:
20.                 path_queue.append(value)
21.                 hash_set.add(value)

    return False
```

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).

   Since the above statement is bidirectional, we prove the two parts of the statement independently, namely:

   - If for every pair, u, v, there's a path from u to v in v to u, then the function returns true
   - If the function returns true, then for every pair, u, v, there's a path from u to v

   Starting with the first part, the algorithm shown above involves checking, for each vertex, *key*, if a path exists between *key* and every other vertex, *sec_key* (lines $2 - 7$). It returns False if a disconnection is detected between *key* and *sec_key* (lines $6 - 7$). If, however, it iterates through **all** pairs and finds no disconnection, then it returns True (line 8).

   Therefore, it holds true that "if for every pair of vertices, u, v, there's a path from v to u, the function returns True."

   Proof for the second part: "If the function returns true, then for every pair, u, v, there's a path from u to v."
   The key to proving this statement is noting that the function returns true **ONLY** after iterating through all possible pairs of vertices and confirming that there's no disconnection between **ANY** pairs. As such, if the function returns True, we can be assured that no disconnection was found between any pair of vertices, which implies that:
   "If the function returns true, then for every pair, u, v, there is a path from u to v."

   Since the algorithm holds true for both parts of the statement, we can conclude that:
   "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 running time of your algorithm is O(m+n), where $m = \|E\|$ and $n = \|V\|$.

   In the case of my algorithm, the above statement doesn't hold true. This is because my algorithm, in the worst case where the graph is connected, iterates through every vertex n times (lines $2 - 3$). As such, the running time is $O(n^2)$, assuming the findPathHelper function is a given (i.e if we don't consider its complexity). If we consider the complexity of the findPathHelper function, however, the time complexity is $O(n^2)(m + n)$. I don't explain this further because it's irrelevant to the proof.

   That said, it is possible to implement an algorithm that runs in $O(m + n)$ time. For the purpose of this proof, I discuss the algorithm, but **do not implement it as my solution to part 2**, because I love the beauty of both algorithms. Let's call the "ideal" algorithm, Y. Y works by picking a random node, say the first node in the adjacency list and performing a BFS on the graph starting from the first node. Y keeps track of nodes visited using a set. It then compares the length of the set with the length of the graph. and returns True if they're equal or False otherwise. See algorithm below.

   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 $O(m + n)$.

```
def isConnected(graph):
    start = graph.keys()[0]
    Let queue be a deque()
    Let set be a Set()
    queue.append(start)
    while (queue):
        popped = queue.popleft()
        for n in graph[popped]:
            if n not in set:
                queue.append(n)
                set.add(n)
    return len(set) == len(graph)
```

   *** 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 ***