4. Directed or Undirected

```
public class IsDirected {
   public static boolean isDirected(int[][] matrix) {
       // check that is a valid graph
       O(1) time and space, nothing being initialized and constant time
       operations
       if(matrix == null || matrix.length <= 1 || matrix.length !=
matrix[0].length)
           return false;

       Space O(1), nothing being initialized
       Outer Looper Time: O(n) where n is the number of rows in an nxn matrix
       for(int row = 0; row < matrix.length; row++) {
           Inner Looper Time: O(n) where n is the number of cols which is
           equal to the number of rows in an nxn matrix
           for(int col = 0; col < matrix[row].length; col++) {
               // check for symmetry across diagonal
               if(matrix[row][col] != matrix[col][row]) {
                   return true;
               }
           }
       }
       return false;
   }
}
```

Time Complexity: O(n^2) where is the amount of rows (matrix.length) in the input matrix.
Space Complexity: O(1).

5. Every Path

```
public static void findEveryPath(int[][] matrix, int u, int w) {
    O(1) time and space
    Queue<ArrayList<Integer>> frontier = new LinkedList<>();
    ArrayList<Integer> visited = new ArrayList<>();
    ArrayList<Integer> path = new ArrayList<>();
    path.add(u);
    frontier.add(path);

    We visit each node only once, so this loop iterates V time where V is
    the number of vertices in the adjacency matrix
    while(!frontier.isEmpty()) {
        path = frontier.poll();
        int vertex = path.getLast();

        if(vertex == w && path.size() == 5)
            System.out.println(path);

        if(!visited.contains(vertex)) {
            visited.add(vertex); We add a vertex to the visited list

            The loop will iterate E times or the number of
            outgoing edges of that node
            for (int adjacentVertex = 0; adjacentVertex < matrix[0].length;
adjacentVertex++) {
                if (matrix[vertex][adjacentVertex] != 0) {
                    Copy the list of the path taken so far, O(V) time
                    O(V) space
                    ArrayList<Integer> tempPath = new ArrayList<>(path);
                    Add to the new list the next adjacent node you step
                    towards
                    tempPath.add(adjacentVertex);
                    Add current path to the list of all paths taken, the
                    last node of the path defines what nodes should be
                    explored
                    frontier.offer(tempPath);
                }
            }
        }
    }
}
```

Time Complexity: O(V+E*V) where V is the number of vertices in the input matrix which is also equal to matrix.length, and E is the maximum amount of outgoing edges from one node which is also less than or equal to matrix.length. It is O(V+E*V) instead of O(V+E) because we must copy the array of all of the previously visited nodes for every outgoing edge.

Space Complexity: O(V^2) where V is the number of vertices in the input matrix which is also equal to matrix.length. Not O(V) because the frontier stack does not only store the nodes that will be explored, it stores all of the nodes to be explored AND all the paths to get that node.

5. Draw Graph

```
input = [('I', 1), ('A', 5), ('E', 4), ('F', 1), ('T', 2), ('S', 3)]

Initialize DiGraph Object O(1) time and space
https://github.com/networkx/networkx/blob/main/networkx/classes/digraph.py#L21
G = nx.DiGraph()

Iterates the same amount of times as vertices in the input which is also equal
to the length of the input array
for vertex, adjacent in input:
   O(1) space and time, add node hashmap of successor nodes, hashmap of
   predecessor nodes, and hashmap of all nodes and their attributes
   G.add_node(vertex)

Iterates the same amount of times as vertices in the input which is also equal
to the length of the input array
for index, vertex_pair in enumerate(input):
   vertex, adjacent_index = vertex_pair
   left_ajdacent_index = (index - adjacent_index) % len(input)
   right_ajdacent_index = (index + adjacent_index) % len(input)
   left_ajdacent = input[left_ajdacent_index][0]
   right_ajdacent = input[right_ajdacent_index][0]
   Each node has a left and right neighbor which will be added to the internal
   hashmaps of successor, predecessor, and node attributes
   G.add_edge(vertex, left_ajdacent)
   G.add_edge(vertex, right_ajdacent)

O(V+E) => O(V) time because each node and edge must be drawn, where V is the
amount of vertices in the input and E is the amount of edges, but every vertix
has at most 2 edges so E <= 2*V. O(V+2*V) => O(V)

https://gist.github.com/koorukuroo/5efb58a86c5396f13650
nx.draw_networkx(G, arrows=True)

https://github.com/matplotlib/matplotlib/blob/main/lib/matplotlib/pyplot.py
Time O(1)
plt.show()
```

Time Complexity: O(V) where V is the number of vertices in the input equal to the array's length. Every vertex and edge must be added to the DiGraph object and drawn on the plot, but it is not O(V+E) because there are at most 2*V outgoing edges. If E is less than or equal to 2*V, the worst case is O(V+2*V), which simplifies to O(V).

Space Complexity: O(V), where V is the number of vertices in the input equal to the length of the array. Every vertex and edge is stored internally in the DiGraph object, but it is not O(V+E) because there are at most 2*V outgoing edges, so E is less than or equal to 2*V. So, the new worst case is O(V+2*V), which simplifies to O(V).