adjMatrix Inputs:

```
// testing functions

adjMatrix adj(6);
adj.displayMatrix();
adj.addVertex();
adj.displayMatrix();
adj.addEdge(1, 0);
adj.addEdge(2, 4);
adj.addEdge(1, 2);
adj.addEdge(3, 3);
adj.addEdge(4, 4);
adj.addEdge(1, 4);
adj.addEdge(4, 5);
adj.displayMatrix();
adj.addVertex();
adj.displayMatrix();
adj.removeVertex(1);
adj.displayMatrix();
adj.removeVertex(2);
adj.displayMatrix();

cout << "Edge at (1, 2)? " << adj.testEdge(1, 2) << endl;
cout << "Edge at (1, 3)? " << adj.testEdge(1, 3) << endl;

int n = 1;
```

```
adj.addEdge(1, 3);
adj.addEdge(1, 6);

vector<int> neightborsExample = adj.outgoingNeighbors(n);
cout << n << ": ";
for (int i = 0; i < neightborsExample.size(); i++)
  cout << neightborsExample[i] << " ";
cout << endl << endl;

adj.displayMatrix();

adj.removeEdge(1,2);
adj.removeEdge(1,3);
adj.removeEdge(2,2);
adj.removeEdge(2,3);

adj.addVertex();
adj.addEdge(0,1);
adj.addEdge(1,4);
adj.addEdge(4,6);
adj.addEdge(6,5);
adj.addEdge(2,5);
adj.addEdge(2,0);
adj.addEdge(0,3);
adj.addEdge(1,3);
adj.addEdge(3,5);
adj.addEdge(3,6);
adj.addEdge(3,2);
```

```
adj.addEdge(3,4);

adj.displayMatrix();

vector<int> neighbors3 = adj.outgoingNeighbors(3);
cout << 3 << ": ";
for (int i = 0; i < neighbors3.size(); i++)
  cout << neighbors3[i] << " ";
cout << endl;
```

adjMatrix Outputs:

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

1 0 1 0 1 0 0

0 0 0 0 1 0 0

0 0 0 1 0 0 0

0 0 0 0 1 1 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

1 0 1 0 1 0 0 0

0 0 0 0 1 0 0 0

0 0 0 1 0 0 0 0

0 0 0 0 1 1 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 1 0 0 0

0 0 1 0 0 0 0

0 0 0 1 1 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0

0 0 1 0 0 0

0 0 1 1 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

Edge at (1, 2)? 1

Edge at (1, 3)? 0

Vertex does not exist!

1: 2 3

0 0 0 0 0 0

0 0 1 1 0 0

0 0 1 1 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 1 0 1 0 0 0

0 0 0 1 1 0 0

1 0 0 0 0 1 0

0 0 1 0 1 1 1

0 0 0 0 0 0 1

0 0 0 0 0 0 0

0 0 0 0 0 1 0

3: 2 4 5 6

adjList Inputs:

```
// testing functions

adjList adj(6);
adj.addEdge(1, 0);
adj.addEdge(2, 4);
adj.addEdge(1, 2);
adj.addEdge(3, 3);
adj.addEdge(4, 4);
adj.addEdge(1, 4);
adj.addEdge(4, 5);
adj.displayMatrix();
adj.removeVertex(2);
adj.displayMatrix();

cout << "Edge at (1, 0)? " << adj.testEdge(1, 0) << endl;
cout << "Edge at (1, 3)? " << adj.testEdge(1, 3) << endl;

cout << endl;

cout << "Outgoing neighbors of Vertex 3: ";
vector<int> neighbors_3 = adj.outgoingNeighbors(3);
for (int i = 0; i < neighbors_3.size(); i++)
  cout << neighbors_3[i] << " ";
cout << endl;

cout << endl;
```

```
adj.displayMatrix();
adj.addVertex();
adj.displayMatrix();
adj.addEdge(1, 0);
adj.addEdge(2, 4);
adj.addEdge(1, 2);
adj.addEdge(3, 3);
adj.addEdge(4, 4);
adj.addEdge(1, 4);
adj.addEdge(4, 5);
adj.displayMatrix();
adj.addVertex();
adj.displayMatrix();
adj.removeVertex(1);
adj.displayMatrix();
adj.removeVertex(2);
adj.displayMatrix();

cout << "Edge at (1, 2)? " << adj.testEdge(1, 2) << endl;
cout << "Edge at (1, 3)? " << adj.testEdge(1, 3) << endl;

int n = 1;

adj.addEdge(1, 3);
adj.addEdge(1, 6);

vector<int> neightborsExample = adj.outgoingNeighbors(n);
```

```
cout << n << ": ";
for (int i = 0; i < neightborsExample.size(); i++)
  cout << neightborsExample[i] << " ";
cout << endl << endl;

adj.displayMatrix();

adj.removeEdge(1,2);
adj.removeEdge(1,3);
adj.removeEdge(2,2);
adj.removeEdge(2,3);

adj.addVertex();
adj.addEdge(0,1);
adj.addEdge(1,4);
adj.addEdge(4,6);
adj.addEdge(6,5);
adj.addEdge(2,5);
adj.addEdge(2,0);
adj.addEdge(0,3);
adj.addEdge(1,3);
adj.addEdge(3,5);
adj.addEdge(3,6);
adj.addEdge(3,2);
adj.addEdge(3,4);

adj.displayMatrix();
```

```
vector<int> neighbors3 = adj.outgoingNeighbors(3);
cout << 3 << ": ";
for (int i = 0; i < neighbors3.size(); i++)
  cout << neighbors3[i] << " ";
cout << endl;
```

adjList Outputs:

0: n

1: 4->2->0->n

2: 4->n

3: 3->n

4: 5->4->n

5: n

0: n

1: 4->0->n

2: 2->n

3: 4->3->n

4: n

Edge at (1, 0)? 1

Edge at (1, 3)? 0

Outgoing neighbors of Vertex 3: 4 3

0: n

1: 4->0->n

2: 2->n

3: 4->3->n

4: n

0: n

1: 4->0->n

2: 2->n

3: 4->3->n

4: n

5: n

0: n

1: 4->2->0->4->0->n

2: 4->2->n

3: 3->4->3->n

4: 5->4->n

5: n

0: n

1: 4->2->0->4->0->n

2: 4->2->n

3: 3->4->3->n

4: 5->4->n

5: n

6: n

0: n

1: 3->1->n

2: 2->3->2->n

3: 4->3->n

4: n

5: n

0: n

1: 3->1->n

2: 3->2->n

3: n

4: n

Edge at (1, 2)? 0

Edge at (1, 3)? 1

1: 6 3 3 1

0: n

1: 6->3->3->1->n

2: 3->2->n

3: n

4: n

0: 3->1->n

1: 3->4->6->3->1->n

2: 0->5->n

3: 4->2->6->5->n

4: 6->n

5: n

3: 4 2 6 5

I chose the above test cases for my code because I wanted to ensure that the program could handle any inputs, ordinary or not, to prove its robustness. Both the adjacency matrix and adjacency list could receive out-of-bounds inputs for each and every function and return an error message (such as those seen in the outputs: “Vertex does not exist!”). The test cases used above checked edge cases, such as removing a vertex that doesn’t exist in either implementation or attempting to BFS or DFS from a source in which there were no neighbors. The result of this extensive testing is robust code that can handle any input.