Challenge: Implement breadth-first search

Implement BFS

In this step, you'll finish implementing the doBFS function, which performs a breadth-first search on a graph and returns an array of objects describing each vertex.

For each vertex v, the object's distance property should be vertex v's distance from the source, and the predecessor property should be vertex v's predecessor on a shortest path from the source. If there is no path from the source to vertex v, then v's distance and predecessor should both be *null*. The source's predecessor should also be *null*.

In the starter code, the function initializes the distance and predecessor values to null, and then enqueues the source vertex. It is up to you to implement the rest of the algorithm, as described in the pseudocode.

Java | Python | C++ | JS

```java
import java.util.LinkedList;
import java.util.Queue;

class BFSInfo {
  public BFSInfo() {
    this.distance = -1;
    this.predecessor = -1;
  }

  public BFSInfo(int distance, int predecessor) {
    this.distance = distance;
    this.predecessor = predecessor;
  }

  public int distance;
  public int predecessor;
};

class Solution {
  public static BFSInfo[] doBFS(int[][] graph, int source) {
    System.out.println(graph.length);
    BFSInfo[] bfsInfo = new BFSInfo[graph.length];
    BFSInfo[] bfsInfo = new BFSInfo[graph.length];

    bfsInfo[source] = new BFSInfo();
    bfsInfo[source].distance = 0;

    Queue<Integer> q = new LinkedList<Integer>();
    q.add(source);

    // Traverse the graph

    // As long as the queue is not empty:
    //  Repeatedly dequeue a vertex u from the queue.
    //
    //  For each neighbor v of u that has not been visited:
    //     Set distance to 1 greater than u's distance
    //     Set predecessor to u
    //     Enqueue v
    //
    //  Hint:
    //  use graph to get the neighbors,
    //  use bfsInfo for distances and predecessors

    return bfsInfo;
  }
}
```