Input: matrix = [
[1,0,0,1,0]
[1,0,1,0,0]
[0,0,1,0,1]
[1,0,1,0,1]
[1,0,1,1,0]
]

2D array of potentially unequal height and width containing only 0s and 1s
0 represents land
1 represents part of a river
A river consist of any number of 1s that are either horizontally or vertically adjacent (but not diagonally adjacent). The # of adjacent 1s forming a river determine its size
Note: A river can twist. In other words, it doesn't have to be a straight vertical line or a straight horizontal line; it can be L-shaped, for example.

Output: [1,2,2,2,5]

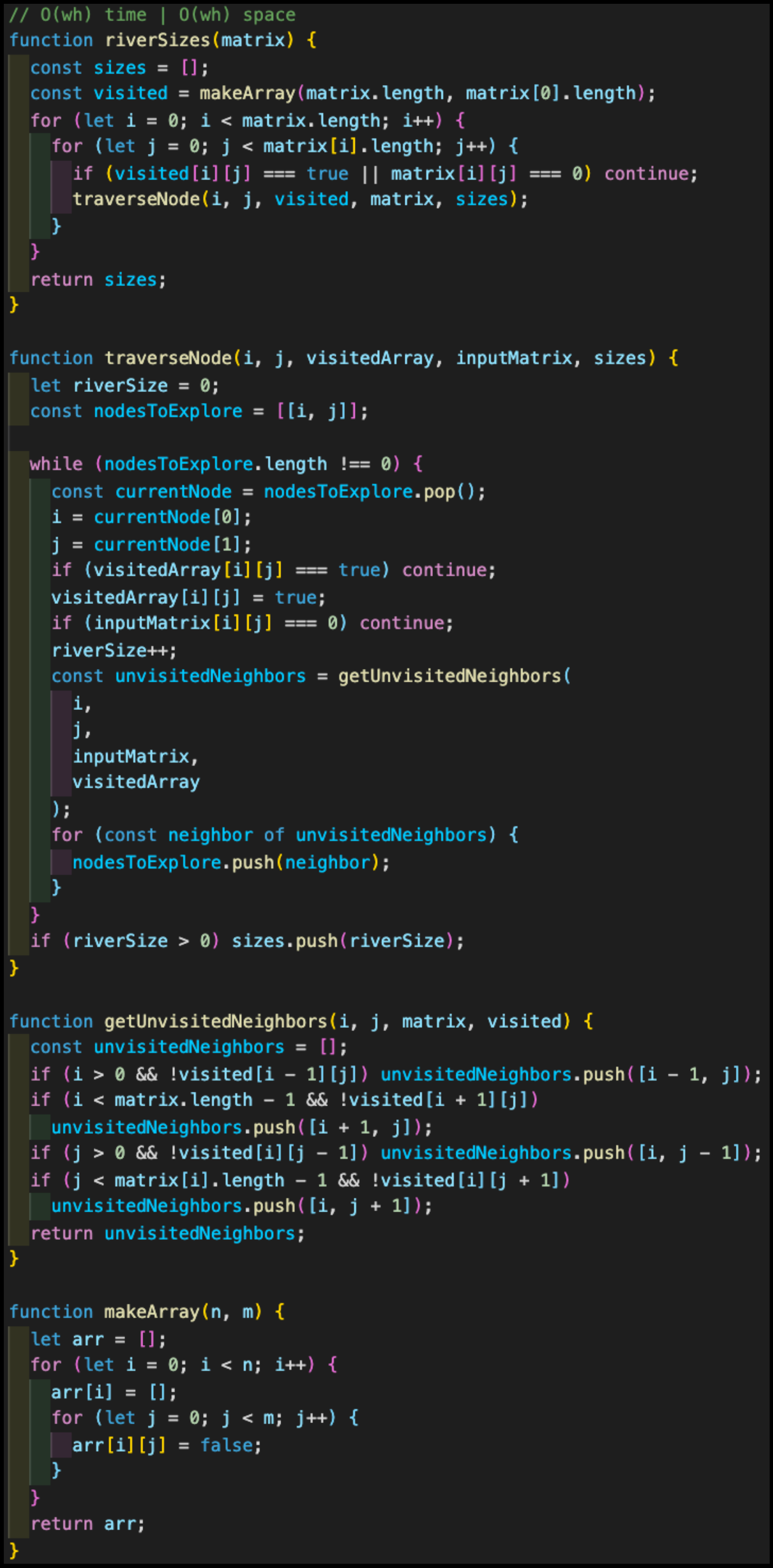
returns an array of the sizes of all rivers represented in the input matrix (order of sizes is not important)

---

1) Traverse through the matrix and at each node:
   if the node is marked as visited or it is zero then continue.
   Otherwise we traverse the Node (the node's value is a 1 so it is the start of a river)

2) Traversing the node means:
   We are at a node with the value 1 and we want to see what other nodes around it have the value 1
   If there is another node with the value 1, we traverse it and increment our river size from 1 to 2
   We use a DFS approach using a Stack
   We add the current Node to the stack and while the stack is not empty we:
   1. Pop off the next node
   2. If the node is visited, we continue to the next loop iteration. If it is not we mark it as visited
   3. If the node is of value 0 we continue to the next iteration. If it is not, we increment our river size
   4. We then get the unvisited neighbours of our node and add them to the stack
   5. Once the stack is empty and we exit out of the loop, we check if the riverSize > 0, if it is then we add it to our output array

3) To get unvisited neighbors we create a new array that will hold the unvisited, non-zero neighbours of our node
   1. If the above node exists ($i>0$) and it is unvisited, we push it onto our unvisited array
   2. If the below node exists ($i <$ matrix.length $- 1$) and it is unvisited, we push it onto our unvisited array
   3. If the left node exists ($j>0$) and it is unvisited, we push it onto our unvisited array
   4. If the right node exists ($j <$ matrix[i].length $- 1$) and it is unvisited, we push it onto our unvisited array
   We then return the unvisited array which is then added to the stack



```javascript
// O(wh) time | O(wh) space
function riverSizes(matrix) {
  const sizes = [];
  const visited = makeArray(matrix.length, matrix[0].length);
  for (let i = 0; i < matrix.length; i++) {
    for (let j = 0; j < matrix[i].length; j++) {
      if (visited[i][j] === true || matrix[i][j] === 0) continue;
      traverseNode(i, j, visited, matrix, sizes);
    }
  }
  return sizes;
}

function traverseNode(i, j, visitedArray, inputMatrix, sizes) {
  let riverSize = 0;
  const nodesToExplore = [[i, j]];

  while (nodesToExplore.length !== 0) {
    const currentNode = nodesToExplore.pop();
    i = currentNode[0];
    j = currentNode[1];
    if (visitedArray[i][j] === true) continue;
    visitedArray[i][j] = true;
    if (inputMatrix[i][j] === 0) continue;
    riverSize++;
    const unvisitedNeighbors = getUnvisitedNeighbors(
      i,
      j,
      inputMatrix,
      visitedArray
    );
    for (const neighbor of unvisitedNeighbors) {
      nodesToExplore.push(neighbor);
    }
  }
  if (riverSize > 0) sizes.push(riverSize);
}

function getUnvisitedNeighbors(i, j, matrix, visited) {
  const unvisitedNeighbors = [];
  if (i > 0 && !visited[i - 1][j]) unvisitedNeighbors.push([i - 1, j]);
  if (i < matrix.length - 1 && !visited[i + 1][j])
    unvisitedNeighbors.push([i + 1, j]);
  if (j > 0 && !visited[i][j - 1]) unvisitedNeighbors.push([i, j - 1]);
  if (j < matrix[i].length - 1 && !visited[i][j + 1])
    unvisitedNeighbors.push([i, j + 1]);
  return unvisitedNeighbors;
}

function makeArray(n, m) {
  let arr = [];
  for (let i = 0; i < n; i++) {
    arr[i] = [];
    for (let j = 0; j < m; j++) {
      arr[i][j] = false;
    }
  }
  return arr;
}
```