Rising Tides

Pseudocode for breadth-first, and the accompanying description, adapted from wording by the great Julie Zelenski.

Global sea levels have been rising, and the most recent data suggest that the rate at which sea levels are rising is increasing. This means that city planners in coastal areas need to start designing developments so that an extra meter of water doesn't flood people out of their homes.

Your task in this part of the assignment is to build a tool that models flooding due to sea level rise. To do so, we're going to model terrains as grids of `double`s, where each `double` represents the altitude of a particular square region on Earth. Higher values indicate higher elevations, while lower values indicate lower elevations. For example, take a look at the three grids to the right. Before moving on, take a minute to think over the following questions, which you don't need to submit. Which picture represents a small hill? Which one represents a long, sloping incline? Which one represents a lowland area surrounded by levees?

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

We can model the flow of water as follows. We'll imagine that there's a water source somewhere in the world and that we have a known height for the water. Water will then flow anywhere it can reach by moving in the four cardinal directions (up/down/left/right) without moving to a location at a higher elevation than the initial water height. For example, suppose that the upper-left corner of each of the three above worlds is the water source. Here's what would be underwater given several different water heights:

Water source at top-left corner

Height: 0m

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

Height: 1m

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

Height: 2m

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

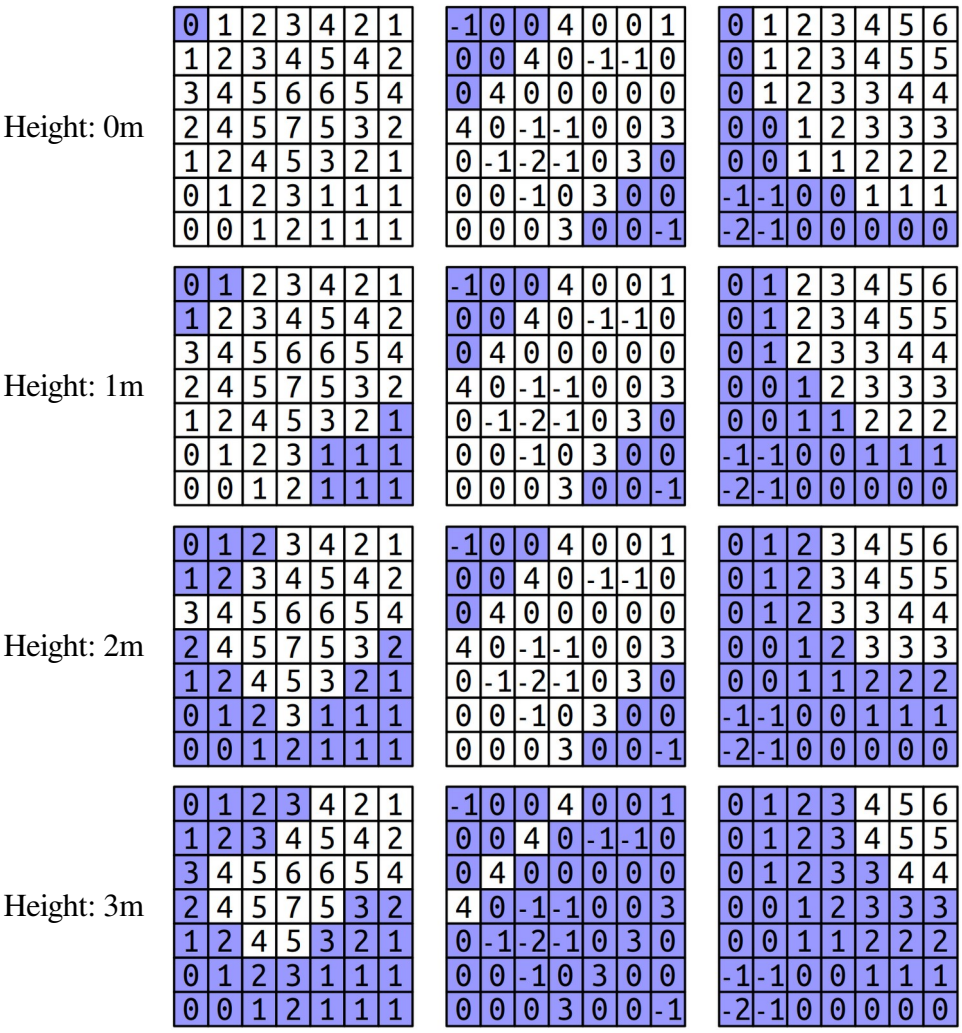
-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

A few things to notice here. First, notice that the water height is independent of the height of the terrain at its starting point. For example, in the bottom row, the water height is always two meters, even though the terrain height of the upper-left corner is either 0m or -1m, depending on the world. Second, in the terrain used in the middle column, notice that the water stays above the upper diagonal line of 4's, since we assume water can only move up, down, left, and right and therefore can't move diagonally through the gaps. Although there's a lot of terrain below the water height, it doesn't end up under water until the height reaches that of the barrier.

It's possible for a grid to have multiple water sources. This might happen, for example, if we were looking at a zoomed-in region of the San Francisco Peninsula, we might have water to both the east and west of the region of land in the middle, and so we'd need to account for the possibility that the water level is rising on both sides. Here's another set of images, this time showing where the water would be in the sample worlds above assume that both the top-left and bottom-right corner are water sources. (We'll assume each water source has the same height.)

Water sources at top-left and bottom-right corners



Notice that the water overtops the levees in the central world, completely flooding the area, as soon as the water height reaches three meters. The water line never changes, regardless of the current elevation. As such, water will never flood across cells at a higher elevation than the water line, but will flood across cells at the same height or below the water line

Your task is to implement a function

```
public static boolean[][] floodedRegionsIn(double[][] terrain,
                                           GridLocation[] sources,
                                           double height);
```

that takes as input a terrain (given as a `double[][]`), a list of locations of water sources (represented as a `GridLocation[]`; more on `GridLocation` later), and the height of the water level, then returns a `boolean[][]` indicating, for each spot in the terrain, whether it's under water (`true`) or above the water (`false`).

You may have noticed that we're making use of the `GridLocation` type. This is a type representing a position in a 2D. You can create a `GridLocation` and access its row and column using this syntax:

```
GridLocation location = new GridLocation(137, 42);
System.out.println(location.row);
System.out.println(location.col);
```

Now that we've talked about the types involved here, let's address how to solve this problem. How, exactly, do you determine what's going to be underwater? Doing so requires you to determine which grid locations are both (1) below the water level and (2) places water can flow to from one of the sources.

Fortunately, there's a beautiful algorithm you can use to solve this problem called ***breadth-first search***. The idea is to simulate having the water flow out from each of the sources at greater and greater distances. First, you consider the water sources themselves. Then, you visit each location one step away from the water sources. Then, you visit each location two steps away from the water sources, then three steps, four steps, etc. In that way, the algorithm ends up eventually finding all places the water can flow to, and it does so fairly quickly!

Breadth-first search is typically implemented by using a ***queue*** that will process every flooded location. The idea is the following: we begin by enqueuing each water source, or at least the sources that aren't above the water line. That means that the queue ends up holding all the flooded locations zero steps away from the sources. We'll then enqueue a next group of elements, corresponding to all the flooded locations one step away from the sources. Then we'll enqueue all flooded locations two steps away from the sources, then three steps away, etc. Eventually, this process will end up visiting every flooded location in the map.

Let's make this a bit more concrete. To get the process started, you add to the queue each of the individual water sources that happen to be at least as high as the grid cell they're located in. From then on, the algorithm operates by dequeuing a location from the front of the queue. Once you have that location, you look at each of the location's neighbors in the four cardinal directions. For each of those neighbors, if that neighbor is already flooded, you don't need to do anything because the search has already considered that square. Similarly, if that neighbor is above the water line, there's nothing to do because that square shouldn't end up under water. However, if neither of those conditions hold, you then add the neighbor to the queue, meaning "I'm going to process this one later on." By delaying processing that neighbor, you get to incrementally explore at greater and greater distances from the water sources. The process stops once the queue is empty; when that happens, every location that needs to be considered will have been visited.

At each step in this process, you're removing the location from the queue that's been sitting there the longest. Take a minute to convince yourself that this means that you'll first visit everything zero steps from a water source, then one step from a water source, then two steps, etc.

To spell out the individual steps of the algorithm, here's some ***pseudocode*** for breadth-first search:

```
create an empty queue;
for (each water source at or below the water level) {
    flood that square;
    add that square to the queue;
}

while (the queue is not empty) {
   dequeue a position from the front of the queue;

   for (each square adjacent to the position in a cardinal direction) {
      if (that square is at or below the water level and isn't yet flooded) {
         flood that square;
         add that square to the queue;
      }
   }
}
```

As is generally the case with pseudocode, several of the operations that are expressed in English need to be fleshed out a bit. For example, the loop that reads

```
for (each square adjacent to the position in a cardinal direction)
```

is a conceptual description of the code that belongs there. It's up to you to determine how to code this up; this might be a loop, or it might be several loops, or it may be something totally different. The basic idea, however, should still make sense. What you need to do is iterate over all the locations that are one position away from the current position. How you do that is up to you.

To summarize, here's what you need to do for this assignment:

> Implement the `floodedRegionsIn` function in `RisingTides.java`. Your code should use the breadth-first search algorithm outlined above in pseudocode to determine which regions are under water. Water flows up, down, left, and right and will submerge any region whose height is less than or equal to the global water level passed in as a parameter. Test as you go.

Some notes on this problem:

- The initial height of the water at a source may be below the level of the terrain there. If that happens, that water source doesn't flood anything, including its initial terrain cell. (This is an edge case where both the "flood it" and "don't flood it" options end up being weird, and for consistency we decided to tiebreak in the "don't flood it" direction.)
- The heights of adjacent cells in the grid may have no relation to one another. For example, if you have the topography of Preikestolen in Norway, you might have a cell of altitude 0m immediately adjacent to a cell of altitude 604m. If you have a low-resolution scan of Mount Whitney and Death Valley, you might have a cell of altitude 4,421m next to a cell of altitude -85m.

The demo app we've included will let you simulate changes to the sea levels for several geographical regions of the United States. The data here is mostly taken from the National Oceanographic and Atmospheric Administration (NOAA), which provides detailed topological maps for the US. Play around with the data sets – what do you find?

We also have some maps from Mars, supplied by a former student as an extension to their assignment! The Martian terrain includes many regions way above and *way* below 0 elevation, so try out some large positive and large negative altitudes to get the full experience here.