**EXPLANATION**

**Description of algorithm**

Throughout the course of my time working on this project, my algorithm changed many times. The main thing I struggled with was getting my algorithm to run fast enough on large documents. The data structure that ultimately helped me speed up my run time immensely was the Python dictionary.

1. First, I wrote a class called *location* that mimics the commonly used Node structure. Each locationis meant to represent a single grid space in the input maze. Each locationstores the following variables:
   1. *Children* -- a dictionary that stores a list of locations that are accessible from the current location.
   2. *Row* -- an integer that represents the row of the current location in relation to the input maze.
   3. *Column* -- an integer that represents the column of the current location in relation to the input maze.
   4. *Data* -- an integer that represents the number of spaces the current location can reach (e.g. a location’s data is 5, so it is connected to any location around it that is (+/-) 5 column or row spaces away, if that location is within the bounds of the maze).
   5. *Active* -- a Boolean that indicates whether or not a location has been visited. This will come in handy for the next step.
2. After opening the file and interpreting the lines, I created a 2-D list, or matrix, of locations.
   1. I accomplished this with a for loop nested within a while loop.
   2. Within the for loop, I created a location for each grid space *matrix*[i][j], starting with (i = 0, j = 0) and ending with (i = n-1, j = n-1).
3. Next, using my matrix of locations, I created an adjacency list representation of the matrix that would later come into play for a breadth-first search.
   1. Using a nested for loop, I assigned children, or reachable locations from the current location, for each spot in the matrix.
   2. To do so, I used the data of each location to check whether it was safe to move left, right, up, or down – “safe” meaning that the coordinates of the next location had to be in bounds of the matrix, and that location had to be set to Active = True (it has not yet been visited).
4. Now, all of the locations in my matrix have been assigned children. My next step was to create an “adjacency dictionary” with a key for each location, and the value of each key being the children of that location.
   1. This “adjacency dictionary” is similar in setup to the adjacency matrix; however, by using this dictionary structure, I made it much easier to look up the children of any given location without having to waste time iterating through a whole list.
   2. The adjacency dictionary improved my run-time immensely.
5. Finally, using the adjacency dictionary, I constructed a “BFS dictionary”, where each key is an integer that represents a layer of the BFS tree (beginning at 0 and ending at the shortest path to the destination location), and each value is a dictionary of all locations belonging to that layer.
   1. This part of my algorithm takes the longest.
   2. It begins with a dictionary that has one entry, {0: { matrix[0][0] } }.
   3. For each layer *i* beginning with 0, I add another layer (*i*+1) to the dictionary that consists of the children of each location in layer *i*.
   4. With large files, this process takes a long time because I am iterating through every location in layer *i*, which, gets to be quite a large number if the shortest path to the destination location is in a much later layer.
   5. To speed up the process a bit, I put a “break” signal in my loop so it breaks at the first detection of the destination location. This way, if the first occurrence of the destination location is in layer 4, but the BFS tree goes all the way to layer 20, I can exit at layer 4 before unnecessarily going through further layers.
   6. The “break” condition speeds up the run time of my program immensely.
   7. Storing the children of each location in a dictionary rather than a list speeds up the time as well since I could simply combine dictionaries instead of iterating through each element of a list.
6. In the last step, I check to see if the destination location is in the last constructed BFS layer.
   1. If it is, this is the shortest path to the location, and I print a message accordingly.
   2. However, if the destination location is not in the last layer, it is not reachable from the start location. Thus, there is no path, and I print a message saying so.

**Running time in milliseconds on the sample inputs with n = 100 and n = 1000**

* 1. For “testMediumA.txt” the run time is 195.65296173095703 ms.
  2. For “testMediumB.txt” the run time is 172.4560260772705 ms.
  3. For “testLargeA.txt” the run time is 637633.1069469452 ms
  4. For “testLargeB.txt” the run time is 28636.504888534546 ms.

**Instructions for running the program**

1. To run the program, call function hw3(fileName) where “hw3” is the name of the function, and “fileName” is the name of the file in string form.
2. Example: hw3(“testLargeA.txt”)