Langton's Ant Algorithm

Design, Test Plan, Test Results:

Problems to be solved:

1) Main: Get user input for x and y coordinates and translate that to matrix[x][y] in 2D array. Make sure the user input make sense. The x and y values that the user enters should be decremented by one internally to account for rows and columns starting at 0 for 2D arrays, which is not intuitive for a user. Another reason to decrement by one is because otherwise a x value of 80 and y value of 80 would be out of bounds on an 80x80 matrix.
2) Main: Get user input for how many rows and columns should be in the matrix.
3) Main: Get user input for total number of steps ant should move
4) Ant Class: Make an antMovement function that will move the ant based on the color of the cell it is on and its current orientation. This function should rotate the ant, swap the cell color that the ant has left, increment the number of steps, and place a "@" where the ant should land. This function should also be able to handle when an ant is about to go out of the array bounds. In the latter case, the ant should rotate 180 degrees and then move forward.
    a. If an ant lands on a white square it should turn ninety degrees, flips the color of the square to black, and moves forward one unit. If an ant lands on a black square it should turn ninety degrees, flips the color of the square to white, and move forward one unit.
    b. Make a dummy x and dummy y variable which will hold the x and y values the ant most recently occupied. The actual x, y, variables could then be changed based on what the ant's next move should be and operate independently from the dummy x and dummy y values.
    c. Create a direction variable that will be assigned a value of 1 at the onset of the game by design. Whenever the ant would move ninety degrees to the right, increment the direction variable. When the ant moves ninety degrees to the left, decrement the direction variable. The direction variable should possess a value of one to four, where one represents north, two represents east, three represents south, and four represents west.
        i. Add additional logical that will reset the direction variable to one, if the variable moves past four. If the direction variable is less than one, reset it to four.

5) Board Class: designateColor function should establish the proper cell symbol (""for white, # for black) given the current color at the onset of the ant movement function. The function should also swap the color of a cell after the ant leaves it. If it's white it should turn black. If it's black it should turn white.
6) Board Class: getColorAt function should retrieve the color of the cell that the ant is about to move to.

    a. Retrieve the color of the cell that the ant will move to before an "@" is placed on the cell and return white or black to the color variable in my ant movement function.
7) Board Class: setAntPosition function should set an ant at matrix[x][y] given the new x, y coordinates.
8) printfunction should print the current state of the board.
9) inputValidation function should receive user input as a string using getline. It should check the contents of the string. If all the contents are a digit and the number is within bounds (number >0 and number <81), it should convert the data type to int and return the number.
    a. If it's not within bounds or not a digit, user should get an error message and get prompted to enter a valid number.
10) Menu Class: addMember function should add members to the menu vector.
11) Menu Class: displayMenu should display the contents of the vector
12) Menu Class verifyAndReturn should prompt users to make a valid selection based on the menu option. If the selection is not valid, the user should be prompted to enter a valid selection.

Menu Test Case

Make two menu options, "start" and "exit". Start can be accessed by entering 1. Exit can be accessed by entering 2.

Test Case	Input	Desired Output
Invalid menu option	5	Please make a selection from the options above.
Exit Menu	2	You have exited.
Start	1	Starts game

Ant Test Case

Test Case	Input: # of Columns	Input: # of Rows	Input: # of Steps	Input: X Coordinate	Input: Y Coordinate	Desired Output
Input is an invalid x coordinate	25	20	3	0		Invalid response. Please enter a number greater than 0
Test ant movement	5	5	3	1	1	All white board appears by design with ant at matrix[0][0] position. Ant

						turns right 90 degrees, move one cell over. Old cell turns black. Ant turns right again, move one cell over. Turns white again moves one cell over. Old cell turns black.
Test Ant Movement	4	4	7	1	1	Default all white board. Ant moves according to algorithm. When the ant lands on a black space and is about to step out of bounds, in the next step the cell changes color and the ant reorients 180 degrees. Then, the ant moves to the next cell.
Input too many rows	20	100				Input is invalid. Please enter a valid number.

Reflection:

Langton's ant algorithm is based on two rules; if an ant lands on a white square, it turns ninety degrees, flips the color of the square to black, and moves forward one unit. If an ant lands on a black square it turn ninety degrees, flips the color of the square to white, and moves forward one unit. There were many factors to consider to move the ant with respect to the algorithm. I needed to know where the ant was currently, where the ant would move next, how the ant was

oriented, and what color cell the ant was currently on. I also to account for when the ant collided with a wall.

My plans for this project was to have a user provide the x and y coordinates values for the dynamically created 2D array, the number of rows and column for the 2D matrix, and the number of steps the ant would move. The default design features were to have a white board with the ant facing north at the start of the game. The ant would be situated on the board based on the user's specification. The user would also specify how many steps the ant should move. From there, the ant would move *x* number of steps with respect to Langton's ant algorithm. If the ant encountered a wall, it would reorient 180 degrees and move forward.

My first design attempted to account for every scenario the ant would face using a series of if statements, where each if statement represented a single scenario. If my ant matched the scenario depicted in any of the if statements, the statement would execute. My program design was initially flawed because it was not scalable and was contingent on a cell being either blank (white) or containing an "#" (black), but did not account for a situation when the ant would occupy a cell. Other design flaws in my original plan included failing to adequately remember both the cell the ant was previously on and the new cell the ant would move to. This problem prevented me from successfully swapping the cell color that the ant moved away from. I also conceptualized orientation in a convoluted way. I created variables North, South, East, and West, assigning them each a value. North would be assigned to one, East would be assigned to two, South would be assigned to three, and West would be assigned to four. Conceptualizing orientation in this way forced me to remember too many variables and made my code less readable. My initial design was also not at all modularized. My ant movement function pseudocode aimed to include all the scenarios the ant would face, with each scenario in its own if statement. I had not made a separate board class yet, so the entire ant movement would be responsible for swapping cell colors, reorienting the ant, and repositioning and reorienting the ant if it almost went out of bounds. I also encountered an issue with my input validation design that would be used for user input. At first I added in logic that would allow a user to only enter an integer that was <81. This was because the maximum width and height of the board was 80. However, this logic presented an issue since a user could type 80 for the x or y coordinate when establishing matrix[x][y], thereby setting a value out of bounds because internally rows and columns start at 0 for 2D arrays.

I had to change my initial design for my program to work. Although I borrowed many facets of my original pseudocode, I made some drastic changes as well. The first thing I did was create a board class. My board class included several new functions. I created a designateColor function to manage the color of the cell, a getColorAt function to determine the color of the current cell, a setAntPosition function to place the ant in a cell, and a print function to print the current state of the board. I extracted and compartmentalized pieces of the original ant movement function into several small functions and put them in my newly made board class to make my program more versatile.

My original plan failed to adequately remember both the cell the ant was previously on and the new cell the ant would move to. I resolved this by making a dummy x and dummy y

variable which would hold the x and y values the ant most recently occupied. The actual x, y, variables could then be changed based on what the ant's next move would be and not be affected by the dummy x and dummy y values.

My original plan was also flawed because it failed to keep track of color accurately. My previous logic kept track of color by determining if a cell contained a blank space or octothorpe. However, this logic failed to account for when the cell was occupied by an ant. My new get color function retrieved the color of the cell that the ant would move to before an "@" was placed on the cell and returned white or black to the color variable in my ant movement function. Afterwards, the color was used to determine the ants new orientation and movement. The if statement that updated the direction variable would now execute if the color == 'w' or color =='b' which made my program no longer reliant on what was contained in an array cell. The ant movement function no longer had the issue of having to ascertain what color a cell was if an ant was on top of it because the color of the cell was assessed before the ant moved to that cell.

In my final design, I also simplified how I would account for changing the ant orientation. Previously, I created four variables, North, East, South, and West and assigned them the values one to four, respectively. This design was convoluted since there were so many variables. I addressed the ant's orientation in a simplified manner by creating just one direction variable, which would be assigned a value of 1 at the onset of the game. The game was designed to make the ant direction face north by default. Whenever the ant would move ninety degrees to the right, I would increment the direction variable. When the ant moved ninety degrees to the left, I decremented the direction variable. The direction variable could possess a value of one to four, where one represented north, two represented east, three represented south, and four represented west. This was a more elegant solution to the orientation problem as it was based on the fact that if the ant was on a white cell, it would move clockwise and increment the current direction by one, and if the ant was on a black cell, it would move counterclockwise and decrement the current direction by one. I also added additional logical that would reset the direction variable to one, if the direction variable incremented past four. If the direction variable was less than one, it would be reset to four.

I made changes to my input validation function in my final design as well. I needed to gather matrix[x][y] input in a user friendly way. For example, although internally matrix[0][0] is referred to as the first row and first column coordinate, this is not intuitive for the user. To accommodate this scenario, I asked the user to specify the row and column they wanted their ant to begin. Internally, I made x =x -1 for the row value entered and y =y-1 for the column value entered. I did so because I wanted my program to operate under the assumption that the user did not know how 2D arrays work. Adding this logic also allowed my inputValidation function to make more sense. A user could now type in 80 for the rows, column, x coordinate, and y coordinate, and the x and y coordinate would not be out of bounds since the x and y coordinate would de decremented by one internally.

This project taught me the importance of creating modular code and anticipating problems as early as possible in the design stage. I initially created incomplete and inaccurate pseudocode, and then started coding promptly after. I quickly ran into problems and became

frustrated because I did not think my design through. This forced me to revisit my design. After I invested a lot of time into my design and addressed all the problems I could foresee, I started being able to write usable code quickly and efficiently. This project has taught me that it is important to do as much thinking upfront so that coding becomes a translating task.