Matthew Morris
Professor Kontothanassis
DS210 HW
6 March 2025

Collaborators: Liam Durcan

Question 1:

```
use std::ops::Neg;

#[derive(Debug, Copy, Clone, PartialEq)]
```

Imported the necessary packages as a part of the code: negative operations, as well as debug, copy, clone and partialeq: all things needed to perform the commands here

Debug: for println!()
Copy/Clone: let p2= p without needing clone functionality
Partialeq: used for the test casing with assert_eq

```
struct Point<T> {
   x: T,
   y: T,
}
```

Created the main struct “point”, assigning x and t to point t. T is a generic, thus referenced with <>.

```
impl<T: Copy + Neg<Output = T>> Point<T> {
```

Needed to specify how the points comply, before putting the functions within this code.

```
fn clockwise(self) -> Point<T> {
       Point {
           x: self.y,
           y: -self.x,
       }
   }
```

Defined the functionality for clockwise. Assigning x to y, and y to -x. This is because you can observe this exact functionality with the utilization of the function matrix a @ [cos(theta) -sin(theta); sin(theta) cos(theta) with 90 degrees. I played around and noticed a general trend resulting in this case with a graph on desmos as well, but an array made it easier to visualize vs one point.

```
    fn counterclockwise(self) -> Point<T> {
        Point {
            x: -self.y,
            y: self.x,
        }
    }
}
```

Similar functionality with counterclockwise.

```
fn main() {
    let p = Point {x: 1.0, y: 2.0};
    let clockwise = p.clockwise();
    let counterclockwise = p.counterclockwise();
    let x = Point {x: 1, y: 2};
    let counterclockwise1 = x.counterclockwise();
    let clockwise1 = x.clockwise();

    println!("Original: {:?}", p);
    println!("Clockwise: {:?}", clockwise);
    println!("Counterclockwise: {:?}", counterclockwise);
    println!("Original: {:?}", x);
    println!("Clockwise: {:?}", clockwise1);
    println!("Counterclockwise: {:?}", counterclockwise1);
}
```

The main function essential takes p as the point (1, 2) and plugs it into the function. From there you do p.clockwise or p.counterclockwise, which performs the calculations as described earlier.

Println! Prints the transformations with {:?} -> needed to print out the function versus normal brackets. This is a part of the debug package. I included both the counterclockwise and clockwise functionality with the generic type, to prove that both an integer and a float work. It also has no specification for integer capacity, thus an i32 or a f64 work just fine.

Test cases:

```
#[test]
fn test_clockwise_rotation() {
    let point = Point {x: 1, y: 2};
    let expected = Point {x: 2, y: -1};
    assert_eq!(point.clockwise(), expected);
}
```

Here is an example of the test code, where I give the start point, and an expected point, where it applies the clockwise functionality to the test case.

Terminal Output (Question 1):
Original: Point { x: 1.0, y: 2.0 }
Clockwise: Point { x: 2.0, y: -1.0 }
Counterclockwise: Point { x: -2.0, y: 1.0 }
Original: Point { x: 1, y: 2 }
Clockwise: Point { x: 2, y: -1 }
Counterclockwise: Point { x: -2, y: 1 }

Question 2:

```
use std::fmt;
const SIZE: usize = 24;
const ITERATIONS: usize = 24;
```

Imported standard format from library
Defined SIZE and ITERATIONS as 24, so that I can use these as references if I wanted to change the iterations or square matrix size. They are both unsigned because iterations and matrix size must be > 0.

```
fn count_neighbors(board: &Vec<Vec<i32>>, i: usize, j: usize) -> i32 {
   let mut count = 0;
   let size = SIZE as isize;
```

Initialized the count_neigbhors function, which is a vector. Defined it as taking a board that is a i32 vector inside of a vector (so that it can be n*n matrix). Defined i and j as usize, so that it doesn't matter what size they are. Additionally assigned mutable count = 0, to count neighbors as it iterates through all the points in the matrix. Assigned size as SIZE (previously defined constant).

```
for x in -1..=1 {
       for y in -1..=1 {
           if x == 0 && y == 0 {
               continue;
           }

           let neighbor_x = match i as isize + x {
               n if n < 0 => (size - 1) as usize,
               n if n >= size => 0,
               n => n as usize,
           };
```

```
            let neighbor_y = match j as isize + y {
                n if n < 0 => (size - 1) as usize,
                n if n >= size => 0,
                n => n as usize,
            };

            count += board[neighbor_x][neighbor_y];
        }
    }
    count
```

This is the main counting neighbors function. I wanted it to have the functionality of wrapping, rather than just omitting counting the neighbors if the point was on an edge. I did through a match statement (which addresses every possible case). Essentially it takes points from x and y from -1 to 1 which means the points surrounding the point. For the mapping, it essentially takes the a value, and if is at 0 or size (the other edge of the matrix), then it assigns it to size or 0, swapping it essentially.

Then, with all of this in mind, it evaluates the neighbors and adds a count to the board based on neighbor x and y.

```
fn calculate_liveness(current_state: i32, live_neighbors: i32) -> i32 {
    if live_neighbors == 3 || (live_neighbors == 2 && current_state == 1) {
        1
    } else {
        0
    }
}
```

new code

This calculates liveness based on the following characteristics: if live_neighbors ==3 or if current state==1 or live_neighbors is equal to 2.

```
fn next_generation(current: &Vec<Vec<i32>>) -> Vec<Vec<i32>> {
    let mut next = vec![vec![0; SIZE]; SIZE];
    for x in 0..SIZE {
        for y in 0..SIZE {
            let live_neighbors = count_neighbors(current, x, y);
            next[x][y] = if live_neighbors == 3 || (live_neighbors == 2 &&
current[x][y] == 1) {
                1
            } else {
```

```
                0
            };
        }
    }
    next
}
```

For this part of the function, we need to assign the successive iteration of the function. We iterate through 0..SIZE for x and y to get through all the points in the matrix. From there, we evaluate the live neighboards (where they equal count_neigbors), and evaluate based on the categories given. If live neighbors == 3 or live neighbors == 2 and current value == 1, then it returns 1, whereas if it doesn't fulfill that then the cell dies.

```
fn display_gameboard(gameboard: &Vec<Vec<i32>>) {
    for row in gameboard {
        for &cell in row {
            print!("{}", if cell == 1 { "x  " } else { "·  " });
        }
        println!();
    }
}
```

The next important function is the gameboard display. We have all the necessary functions so far, but no way to display them. For this, we essentially take the vector of gameboard, and assign it to x if the value in the vector given is == 1, else ·, which gives us a simplistic board of x's and ·'s. It then goes on to print the function once iterating through all possible rows and cells.

```
fn main() {
    let mut board = vec![vec![0; SIZE]; SIZE];
    let initial_points = vec![(0,1), (1,2), (2,0), (2,1), (2,2)];
    for &(x, y) in &initial_points {
        board[x][y] = 1;
    }

    for i in 0..ITERATIONS {
        println!("Iteration: {}", i + 1);
        display_gameboard(&board);
        board = next_generation(&board);
    }
}
```

For the main function, we put everything together, assigning the board to an arbitrary size (filled in by what we defined as size), and then defined the initial points that we wanted as 1. We

assigned initial points, but never mentioned what they would be defined them as, so we need to iterate through inital_points to assign them on the board as 1.

Then it iterates through ITERATIONS which counts the # of iterations, and then i+1, where it prints the successive gameboard.

```
#[test]
fn test_newgeneration() {
    let mut board = vec![vec![0; SIZE]; SIZE];
    board[0][1] = 1;
    board[1][2] = 1;
    board[2][0] = 1;
    board[2][1] = 1;
    board[2][2] = 1;
    let new_board = next_generation(&board);
    let mut expected_board = new_board.clone();
    expected_board[1][0] = 1;
    expected_board[0][1] = 0;
    expected_board[2][0] = 0;
    expected_board[3][1] = 1;

    assert_eq!(new_board, expected_board);
}
```

Test function #1: Takes a set of points, and applies the transformation on them (next_generation) and then compares it against expected_board values to see if they work. I used the initially given points and compared them against what I manually calculated as the neighboring values to see if the function works.

```
#[test]
fn test_blinker_oscillator() {
    let mut board = vec![vec![0; SIZE]; SIZE];

    board[1][2] = 1;
    board[1][3] = 1;
    board[1][4] = 1;

    let new_board = next_generation(&board);

    let mut expected_board = vec![vec![0; SIZE]; SIZE];
    expected_board[0][3] = 1;
    expected_board[1][3] = 1;
    expected_board[2][3] = 1;
```

```
    assert_eq!(new_board, expected_board);
}
```

This test I was slightly confused on, so I used ChatGPT to come up with another test. For this, it came up with a blinker test, where you can have a distinct points set, versus the ones given initially for the test. This plots points along a line and predicts based on that, which allows for a different set of points, as opposed to basing the fact that my function works on my inital test points given. This results in a predictable result, which ChatGPT gave me as the answer.

Below is the ChatGPT Query:

how can i get another test

To create another test for your Conway's Game of Life implementation, follow these steps:

1. Define a New Test Case

You need a different initial board configuration and an expected outcome based on Conway's rules.

2. Steps to Create a Test

- **Choose a new initial configuration** (e.g., a "Blinker" oscillator).
- **Compute the expected next state manually**.
- **Use assert_eq! to compare next_generation() output with expected results.**

Example: Testing a "Blinker" Oscillator

A Blinker is a common pattern in Conway's Game of Life that alternates between two states:

Initial State:

Copy

```
. . . . .
. x x x .
. . . . .
. . . . .
```

Next Generation:

```
. . x . .
. . x . .
. . x . .
. . . . .
```

This pattern should repeat indefinitely.

Code for New Test

rust

```rust
#[test]
fn test_blinker_oscillator() {
    let mut board = vec![vec![0; SIZE]; SIZE];

    // Initial state of the blinker (horizontal)
    board[1][2] = 1;
    board[1][3] = 1;
    board[1][4] = 1;

    let new_board = next_generation(&board);

    // Expected state of the blinker (vertical)
    let mut expected_board = vec![vec![0; SIZE]; SIZE];
    expected_board[0][3] = 1;
    expected_board[1][3] = 1;
    expected_board[2][3] = 1;

    assert_eq!(new_board, expected_board);
}
```

Why This Works

- It sets up an **initial** `board` with a horizontal blinker.
- It computes the **next state** using `next_generation(&board)`.
- It **manually defines the expected next state** (`expected_board`).
- It checks whether `next_generation()` correctly transformed the board.

Next Steps

- Try running `cargo test` to verify the test.
- Add more test cases for **still lifes** (e.g., a **Block** pattern that remains unchanged) or **gliders** (moving patterns).

Terminal Output (Question 2):

```
   Compiling question2 v0.1.0
(/Users/matthewmorris/Downloads/BU/DS210_HW4/question2)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.79s
     Running `target/debug/question2`
Iteration: 1
. x . . . . . . . . . . . . . . . . . . . . . . . .
. . x . . . . . . . . . . . . . . . . . . . . . . .
x x x . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
Iteration: 2
. . . . . . . . . . . . . . . . . . . . . . . . . .
x . x . . . . . . . . . . . . . . . . . . . . . . . .
. x x . . . . . . . . . . . . . . . . . . . . . . . .
. x . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
```

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 3

. . . . . .
. . **x** . . . . . .
x . **x** . . . . . .
. **x** **x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 4

. . . . . .
. **x** . . . . . .
. . **x** **x** . . . . . .
. **x** **x** . . . . . .
. . . . . .
. . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 5

. . . . . .
. . **x** . . . . . .
. . . **x** . . . . . .
. **x x x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 6

. . . . . .
. . . . . .
. x . x . . . . . .
. . x x . . . . . .
. . x . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 7

. . . . . .
. . . . . .
. . . x . . . . . .
. x . x . . . . . .
. . x x . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 8

. . . . . .
. . . . . .
. . **x** . . . . . .
. . . **x x** . . . . . .
. . **x x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 9

. . . . . .
. . . . . .
. . . **x** . . . . . .
. . . . **x** . . . . . .
. . **x x x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 10

. . . . . .
. . . . . .
. . . . . .
. . **x** . **x** . . . . . .
. . . **x** **x** . . . . . .
. . . **x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 11

. . . . . .
. . . . . .
. . . . . .
. . . . **x** . . . . . .
. . **x** . **x** . . . . . .
. . . **x** **x** . . . . . .
. . . . . .

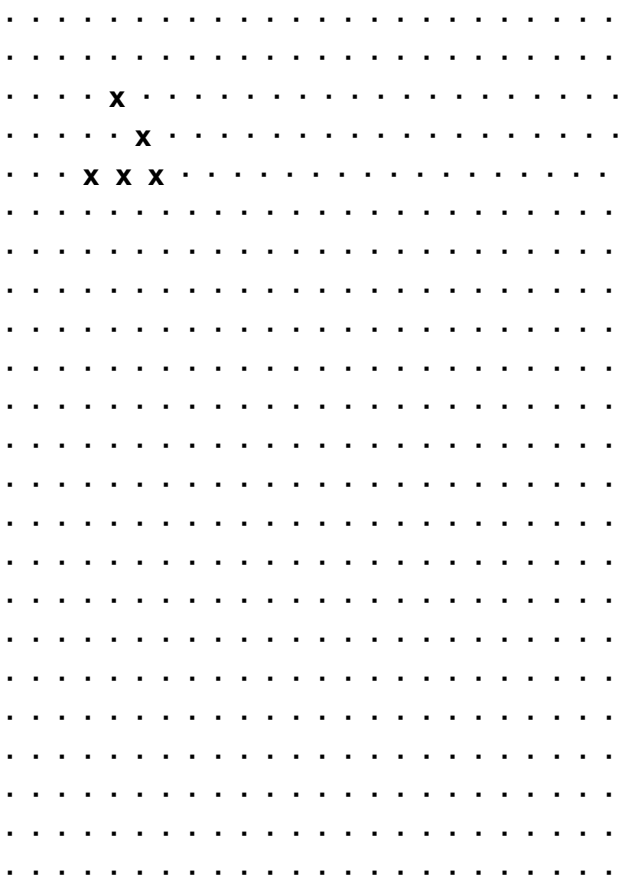
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 12

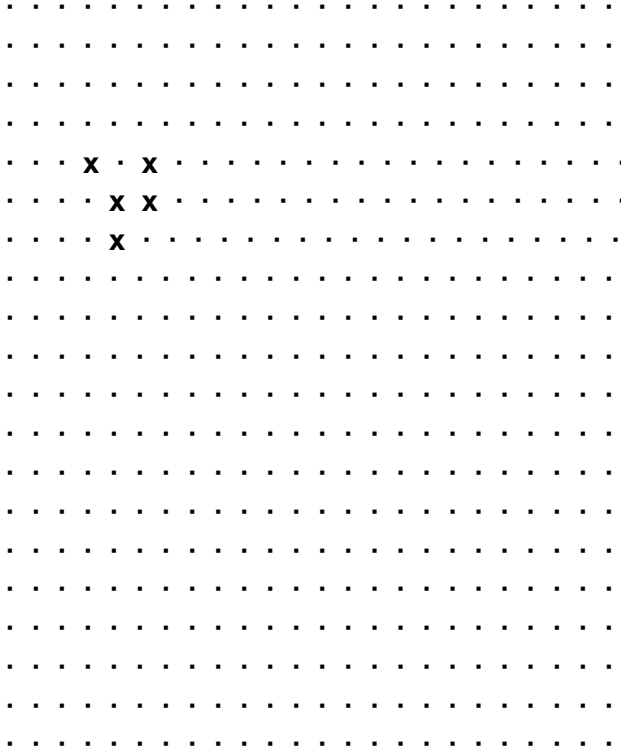
. . . . . .
. . . . . .
. . . . . .
. . . **x** . . . . . .
. . . . **x x** . . . . . .
. . . **x x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 13

. . . . . .



Iteration: 14



. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 15

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . **x** . . . . . .
. . . **x** . **x** . . . . . .
. . . . **x** **x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 16

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . **x** . . . . . .
. . . . . **x** **x** . . . . . .
. . . . **x** **x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

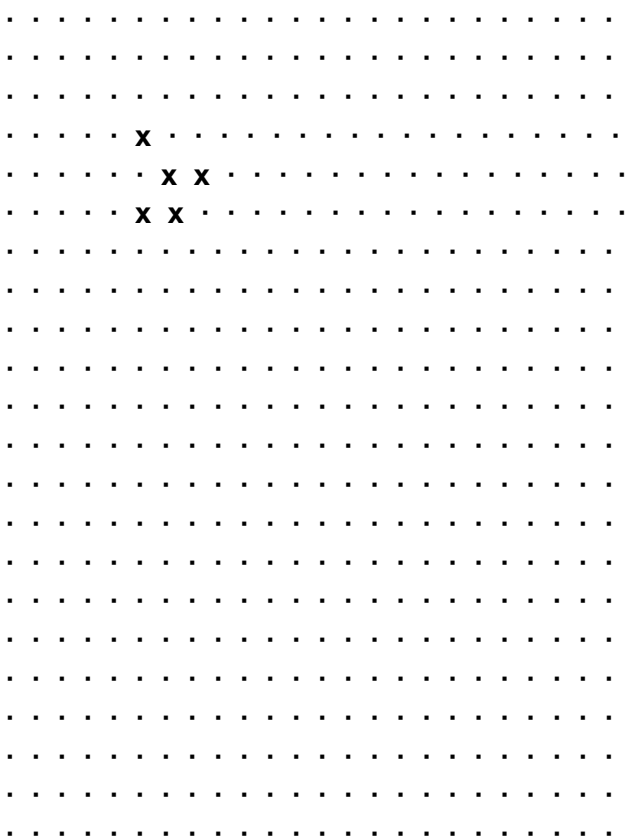
Iteration: 17

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . **x** . . . . . .
. . . . . **x** . . . . . .
. . . . **x x x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

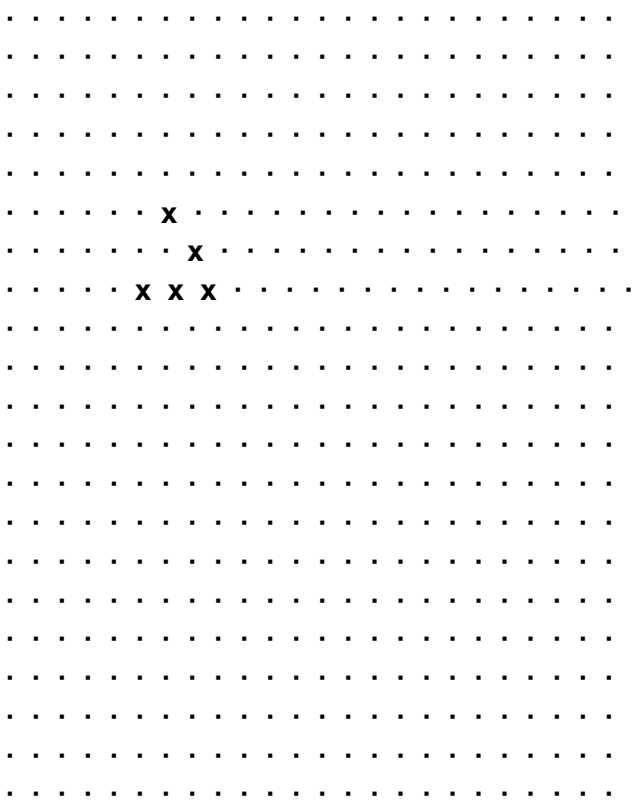
Iteration: 18

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . **x . x** . . . . . .
. . . . . **x x** . . . . . .
. . . . . **x** . . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
Iteration: 19
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . **x** . . . . . .
. . . . **x** . **x** . . . . . .
. . . . . **x** **x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
Iteration: 20
. . . . . .
. . . . . .



Iteration: 21



. . . . . .
. . . . . .
. . . . . .

Iteration: 22

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . x . x . . . . . .
. . . . . x x . . . . . .
. . . . . x . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 23

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . x . . . . . .
. . . . . x . x . . . . . .
. . . . . x x . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Iteration: 24

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . **x** . . . . . .
. . . . . **x x** . . . . . .
. . . . . **x x** . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Corrections:

```rust
fn calculate_liveness(current_state: i32, live_neighbors: i32) -> i32 {
    if live_neighbors == 3 || (live_neighbors == 2 && current_state == 1) {
        1
    } else {
        0
    }
}
```

Explanation of Mistake:

In my original code, I did not create an individual function for the calculation of liveness. This is important because we need an individual check to verify the values of the board. In this new code, I implement a check, verifying each of the variables defined previously, seeing if the requirements (which are given as a part of the instructions) are true or not, to define if the new value is either 0 or 1.

How I learned from this mistake:

This was a simple mistake, because I now know how important it is to have individualized functions to make code more efficient and easy to understand and use. This mistake is also partly missing that final instruction in the homework assignment which is my fault for not reading thoroughly enough. Going forward I should be more thorough with the read-through and check each assignment.