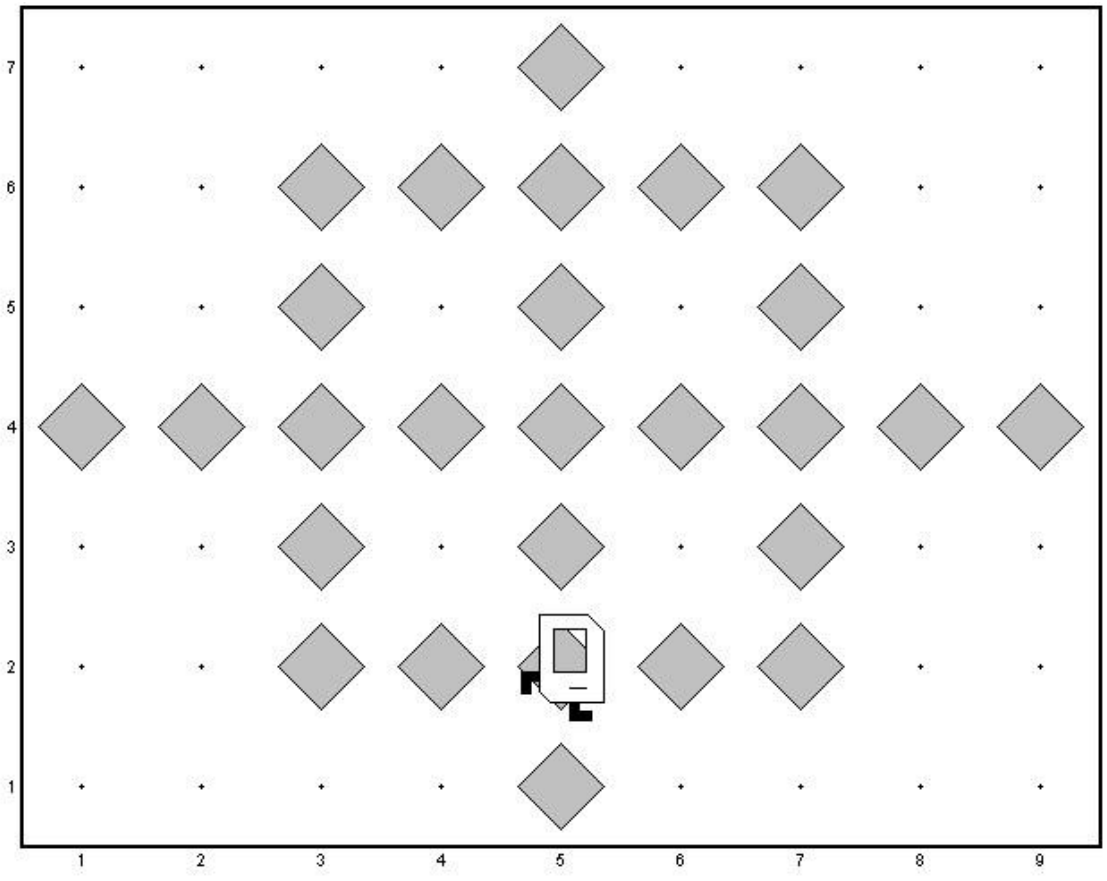
Karel the Robot Solution Report



Abdallah Alkayyal

Abstract

This assignment aimed to create an optimized Java algorithm to automate Karel, a robot, in dividing a given map into 4 + 4 equal areas.

The goal was to generate the largest possible equal squares for the inner chambers and form outer chambers in the shape of L. Special cases, including maps that were too small for the desired shape, were considered.

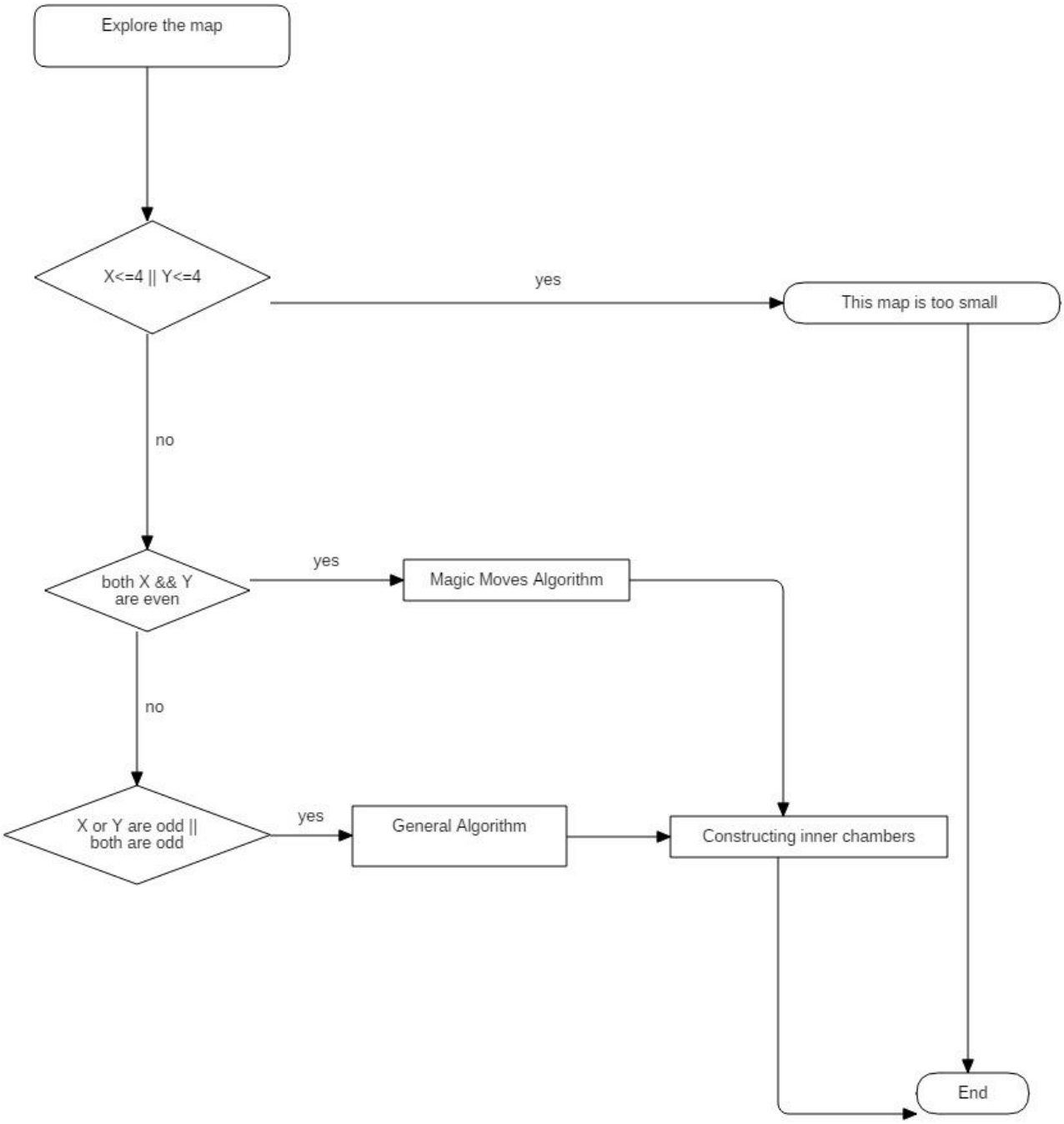
The approach involved developing two distinct algorithms to handle different scenarios and achieve an efficient solution with minimal beepers and steps.

One algorithm was designed specifically for maps with even dimensions (both rows and columns), addressing the even-even case.

Another algorithm was crafted to handle maps with odd dimensions (both rows and columns), covering the even-even case.

A similar algorithm was also used to adapt to maps with even rows and odd columns, referred to as the even-odd case. Each algorithm was meticulously designed to optimize the map division process, ensuring accurate placement of inner squares and L-shaped outer chambers in each scenario.

Flow Charts


Explore the map
X<=4 || Y<=4
yes
This map is too small
no
both X && Y are even
yes
Magic Moves Algorithm
no
X or Y are odd || both are odd
yes
General Algorithm
Constructing inner chambers
End

Attributes and Methods

```
private int steps = 0, x = 1, y = 1, beepers = 0;
```

```
public void exploreMap() {...}
1 usage
public void outerChambers() {...}
1 usage
public void magicMoves() {...}
2 usages
public void putMagicLineOfBeepers(int x, int y) {...}
17 usages
public void goToMiddle(int temp) {...}
2 usages
public void putBeeperAndIncrement() {...}
2 usages
public void directCorrect(){...}
13 usages
public void putOneLineOfBeepers(int temp) {...}
11 usages
public void putDoubleLineOfBeepers(int temp) {...}
4 usages
public void goToClosestCorner(String s) {...}
1 usage
public void innerChambers() {...}
13 usages
public void directKarelToDown() {...}
10 usages
public void directKarelToUp() {...}
10 usages
public void directKarelToRight() {...}
12 usages
public void directKarelToLeft() {...}
```

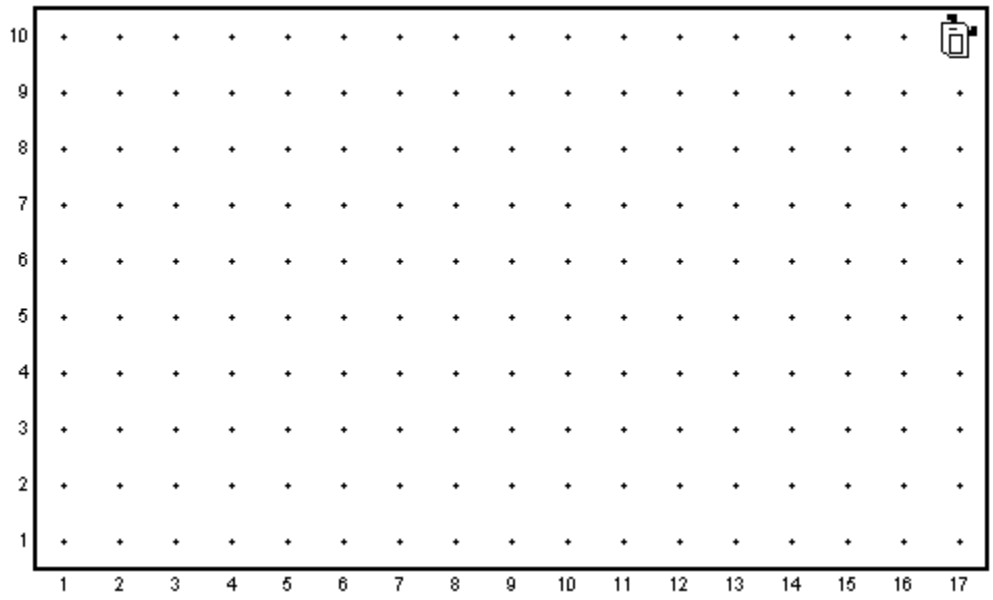
Implementation and Analysis

The program is structured into three primary functions:

First function: `public void exploreMap() {...}`

Function is to enable Karel to explore the map by moving horizontally to determine the number of rows (x-coordinate) until it reaches a barrier. Once a barrier is encountered, Karel changes its direction and moves vertically to explore the number of columns (y-coordinate) until it faces another barrier. This sequential exploration allows Karel to systematically gather information about the map's dimensions.

After executing the exploreMap() function, Karel consistently positions itself in the top-right corner of the map.



Second function: `public void outerChambers() {...}`

The function divides the map into four equal areas by initially splitting along the smaller dimension (columns or rows) and then progressing towards the larger dimension. This strategy optimizes the process, reducing unnecessary steps in the subsequent inner chambers.

Here's a breakdown of the function's workflow:

- First, the function checks if either the **x** or **y** of Karel's position is less than or equal to 4. If this condition is true, it indicates that the map is too small to be divided into 4+4 equal areas, and the program exists. This check ensures that the map has sufficient dimensions for the division process. (Magic Moves Algorithm)

- Next, the function checks if both the x and y values are even. If they are, it invokes the **magicMoves**() function, which offers a more efficient approach for splitting the map when both dimensions are even. Further details about the **magicMoves**() function will be provided later in the report.

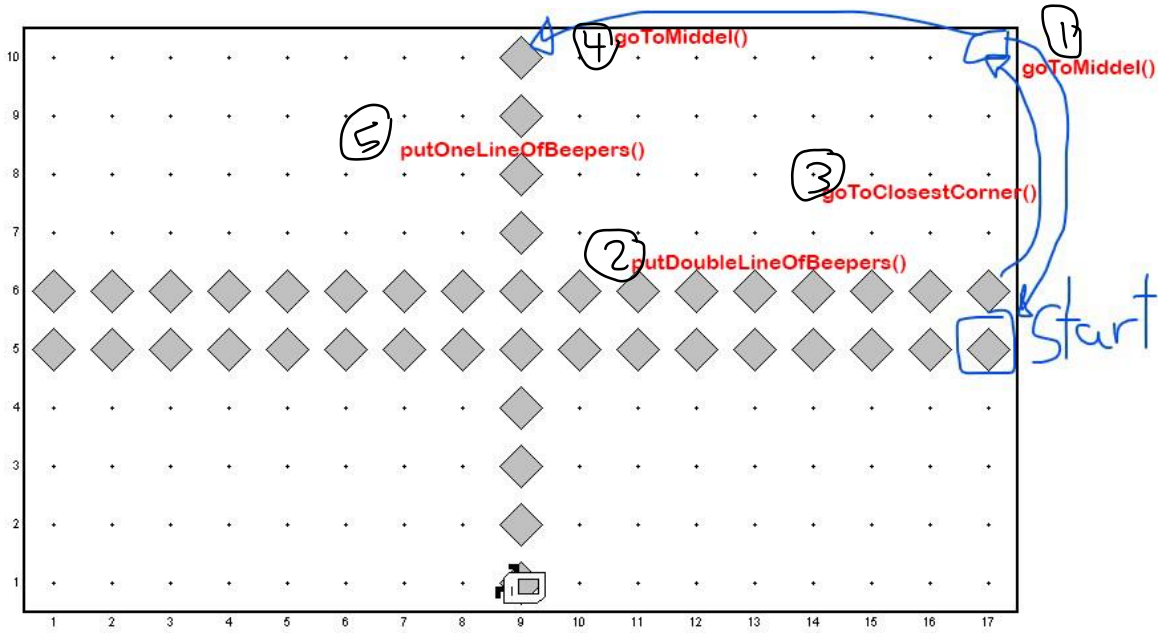
- If either the x or y value is odd, the function proceeds with the division process. It determines which dimension (x or y) has a smaller value and orients Karel towards the left or down accordingly. This positioning ensures that Karel is in the correct starting position for further division.

- The function then calls the **goToMiddle**() function to move Karel to the middle of the smaller dimension. This step ensures that Karel is properly positioned for placing the division lines of beepers.

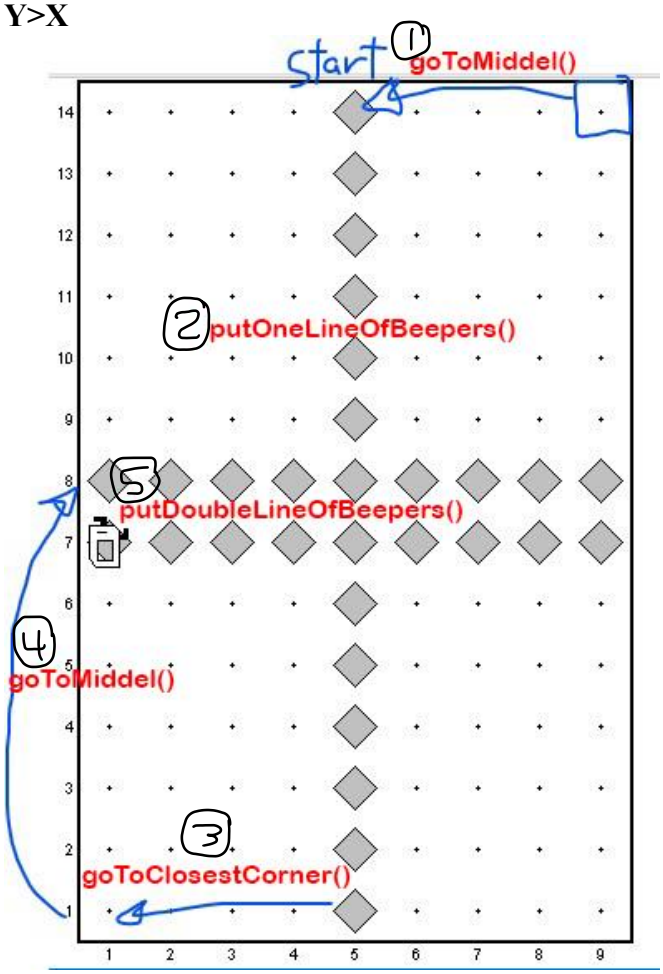
- Next, the function checks if the smaller dimension is odd or even. Based on this determination, it selects the appropriate function to use for dividing the map. If the dimension is odd, the **putOneLineOfBeepers**() function is called.
  If the dimension is even, the **putDoubleLineOfBeepers**() function is used. These functions place the desired pattern of beepers to create the division lines.

- Finally, the function calls the **goToClosestCorner**() function to split the remaining section of the map in same way. This optimization strategy aims to avoid unnecessary steps and position Karel in the closest location for efficient division into inner chambers.

X>Y


goToMiddel()
goToMiddel()
putOneLineOfBeepers()
goToClosestCorner()
putDoubleLineOfBeepers()
Start

Y>X


Start
goToMiddel()
putOneLineOfBeepers()
putDoubleLineOfBeepers()
goToMiddel()
goToClosestCorner()

After positioning Karel in an efficient location to start constructing the inner chambers, the **innerChambers**() function is called.

Third function: `public void innerChambers() {...}`

This function is responsible for dividing the map into the largest possible equal squares Here's how it works:

- First, the function finds the smallest dimension (columns or rows) by comparing **x** and **y** values. This is done to determine which dimension Karel will proceed along to build the inner chambers.

- Once the smaller dimension is determined, the function calculates the number of steps, denoted as 'n', required to traverse that dimension. The formula used is **(dimension - 2) / 2**, which represents the length of each inner chamber side. The result is rounded up to the nearest integer using the **Math.ceil()** function to ensure that Karel covers the maximum possible area within each quadrant.

- After calculating 'n', Karel takes a backstep to position itself correctly for constructing the inner chambers.

- the function begins a loop that iterates through eight possible paths, representing each quadrant and the spaces in between. Karel checks its looking orientation using conditional expressions and aligns itself properly for each loop.

  During each iteration, Karel walks and draws the edges of the square for the corresponding quadrant. Here's how it progresses:

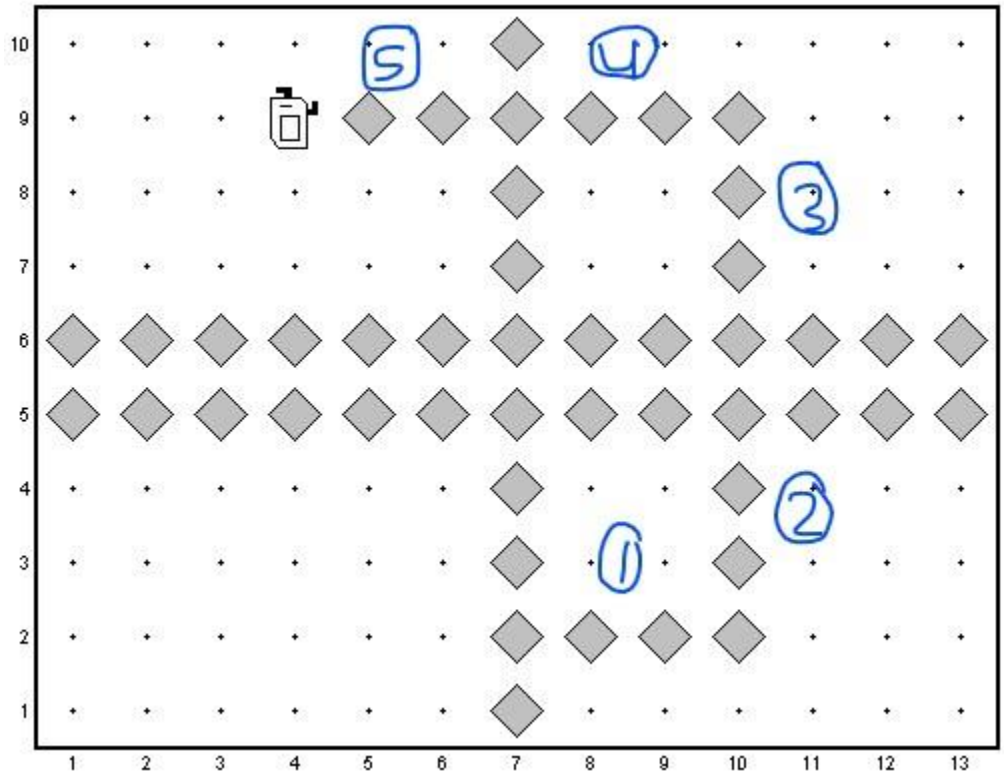
  1. If Karel is facing north (**facingNorth**() is true), it calls the **directKarelToLeft**() function to align itself towards the left.
  2. If Karel is facing west (**facingWest**() is true), it calls the **directKarelToDown**() function to align itself downwards.
  3. If Karel is facing south (**facingSouth**() is true), it calls the **directKarelToRight**() function to align itself towards the right.
  4. If Karel is facing east (**facingEast**() is true), it calls the **directKarelToUp**() function to align itself upwards.

Once Karel is aligned in the appropriate direction, it proceeds to draw the edges of the square for that quadrant. The specific drawing method depends on the parity of the dimensions and is handled by the **putOneLineOfBeepers**(**n**) and **putDoubleLineOfBeepers**(**n**) functions. However, these functions are used in a different manner to reduce code repetition.

- The function includes additional logic. When Karel is facing a double line of beepers, it skips one step to avoid overlapping with the existing line and ensure precise positioning within the inner chambers.

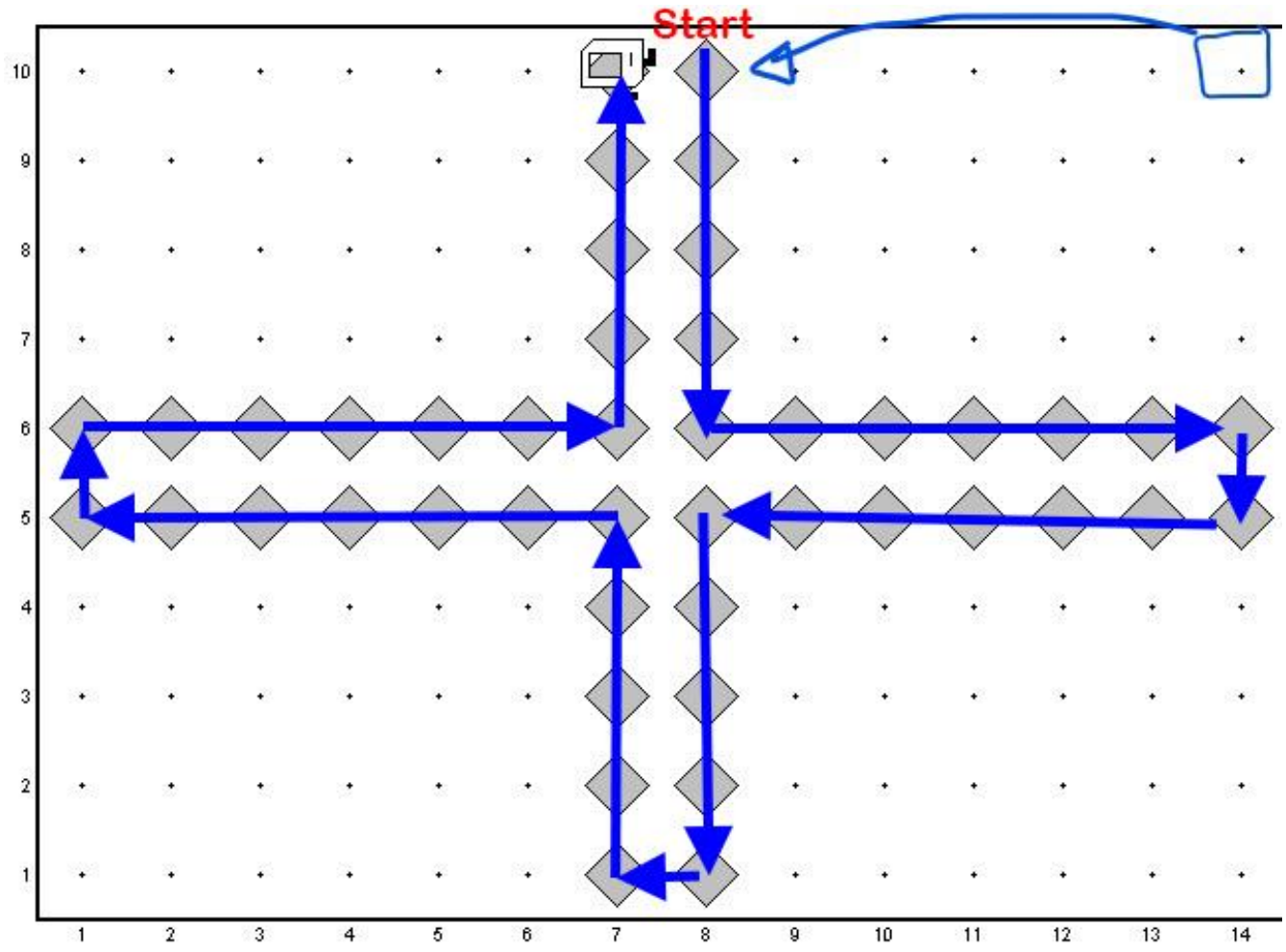
- The loop continues until all eight directions are covered, effectively creating the largest possible equal squares in each quadrant.

First 5 iterations:



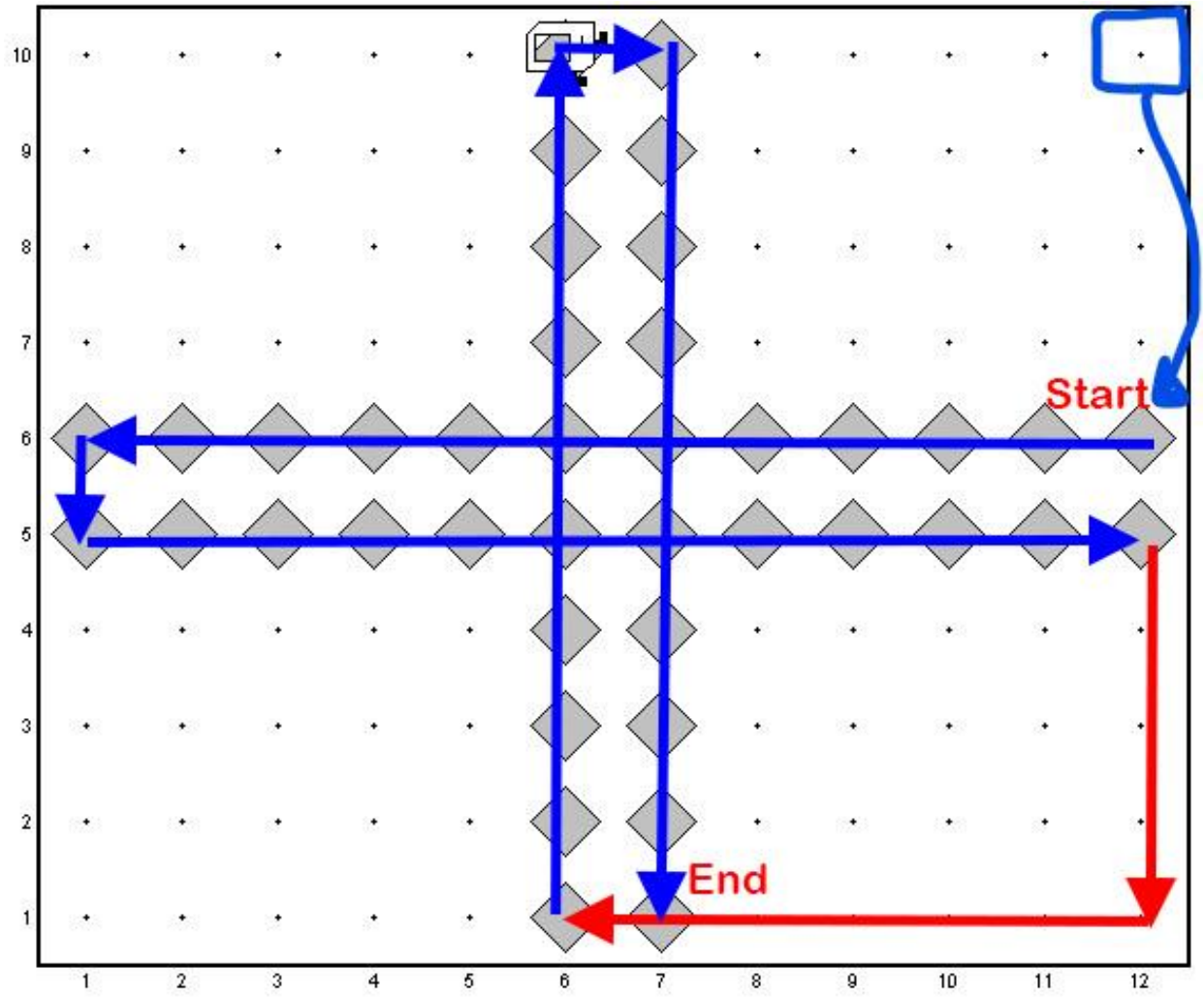
Magic Moves Algorithm

In the scenario where both the x and y values are even, I utilize a distinctive approach that differs from the previously described method. This alternative strategy incorporates the utilization of "**magicMoves**()" in conjunction with the "**putMagicLineOfBeepers**()" function.
I have found that this approach is notably more effective and efficient, particularly in reducing unnecessary steps when dealing with the even-even case.



There is no step that is unnecessary.

General algorithm:



The approach of **magicMoves**() avoids the use of red lines.

```
public void magicMoves() {...}
2 usages
public void putMagicLineOfBeepers(int x, int y) {...}
```

The **magicMoves**() function is responsible for efficiently dividing the map into four equal areas when both the x and y values are even.

If the **X** value is greater than or equal to the **Y** value, Karel goes to the middle of the top edge using the **goToMiddle**() function, passing the x value as the parameter. Then, it calls the **putMagicLineOfBeepers**() function, passing the x and y values to place a specific pattern of beepers.

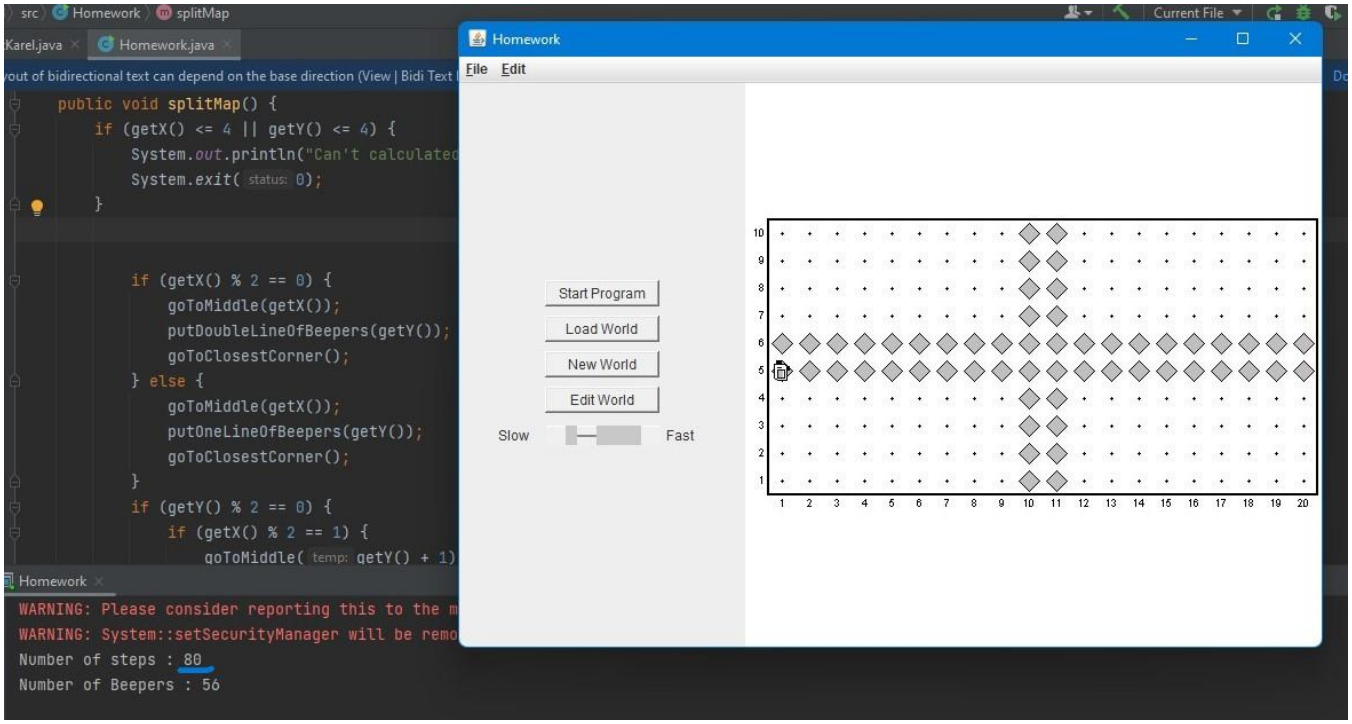
If the **Y** value is greater than the **X** value, Karel first directs itself downward using the **directKarelToDown**() function. Then, it goes to the middle of the right edge by calling the **goToMiddle**() function, passing the y+1 value as the parameter. After that, it calls the **putMagicLineOfBeepers**() function, passing the y and x values to place the desired pattern of beepers.

The **putMagicLineOfBeepers**() function is responsible for placing beepers in a specific pattern based on the given x and y values. It starts by placing a line of beepers using the **putOneLineOfBeepers**() function with a length of y/2.
Then, depending on Karel's facing direction and the availability of a clear path, it changes the direction using the **directKarelToRight**(), **directKarelToDown**(), **directKarelToLeft**(), or **directKarelToUp**() functions.

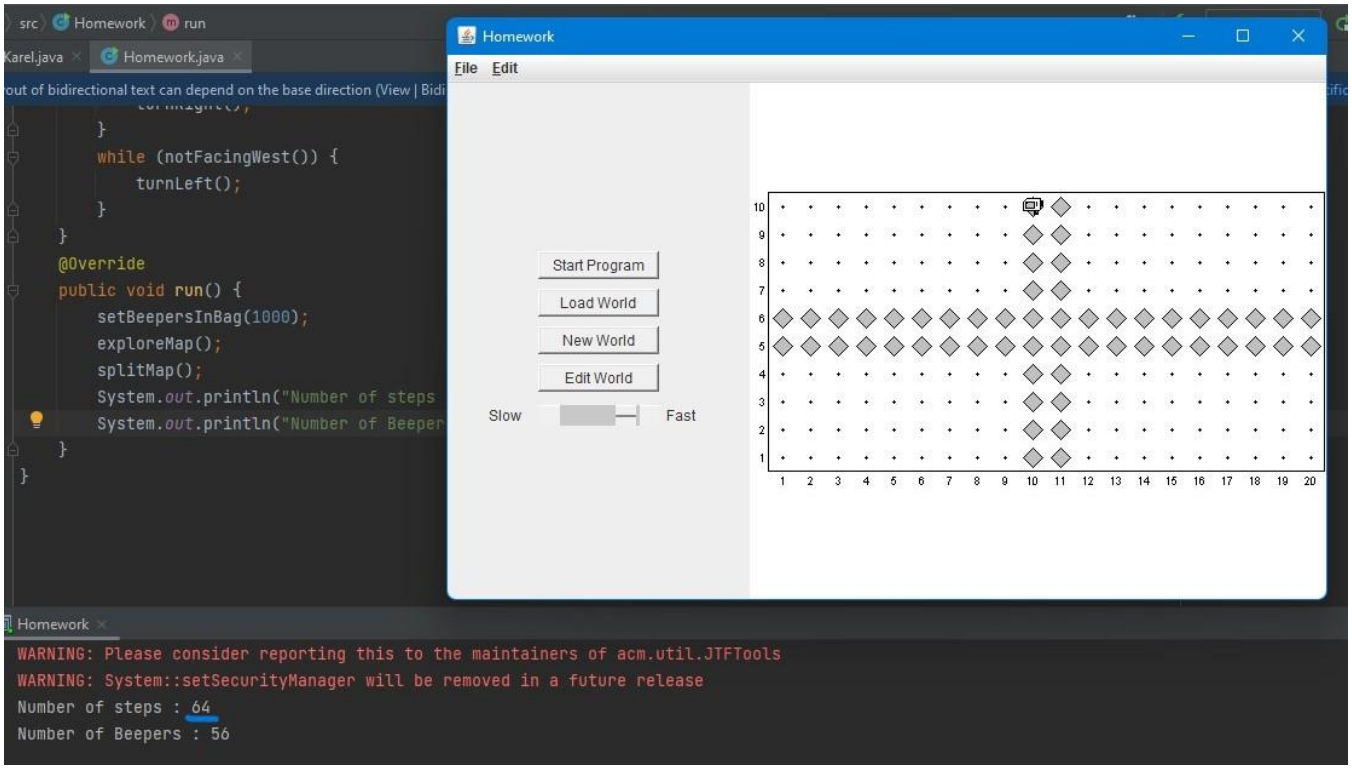
Next, it places a double line of beepers using the **putDoubleLineOfBeepers**() function with a length of x/2. Again, it adjusts the direction based on Karel's facing direction and available clear path. This process is repeated for the remaining sides of the square, placing double lines of beepers, and adjusting the direction accordingly.

Finally, it places another line of beepers using the putOneLineOfBeepers() function with a length of y/2.

And this picture shows number of steps before optimization:



After optimization :



Helper Functions

```
public void correctDirection(){...}
```

Function was implemented to reduce redundancy in the code by consolidating the logic for adjusting Karel's orientation based on its current facing direction. This function is utilized in both the **goToMiddle**() and **goToClosestCorner**() functions.

```
public void goToMiddle(int temp) {...}
```

Function moves Karel to the middle of a given dimension. It takes an integer argument **temp** representing the dimension number.

If the dimension is even, Karel walks **temp / 2 - 1** steps. If the dimension is odd, Karel walks **Math.ceil(temp / 2)** steps.

At the end of the function, Karel's direction is oriented towards the center of the map using the **correctDirection()** function, preparing for the subsequent steps required for map division.

```
public void goToClosestCorner(String s) {...}
```

Function moves Karel to the nearest corner of the map, considering the specified direction **S** (up, down, right, or left). By using appropriate orientation commands like directKarelToLeft() or directKarelToRight(), Karel is aligned towards the intended corner. Karel then proceeds to move until encountering a barrier, ensuring successful arrival at the closest corner.

The function also incorporates the correctDirection() step to properly orient Karel's direction for subsequent map division tasks.

```
public void putOneLineOfBeepers(int temp) {...}
```

function is used to place a line of beepers in Karel's path. It iterates temp - 1 times, where temp represents the desired number of steps. With each iteration, Karel puts a beeper and increments the step count. If Karel encounters a barrier and cannot move forward, it still places a beeper before exiting the loop.

```
public void putDoubleLineOfBeepers(int temp) {...}
```

function places a double line of beepers in Karel's path. It first calls putOneLineOfBeepers to place a single line of beepers. Then, based on Karel's direction and barriers, it adjusts Karel's position. Finally, it calls putOneLineOfBeepers again to place the second line of beepers.

```
public void setDefault() {...}
```

function resets the variables steps, x, y, and beepers to their default values. It sets steps to 0 and x and y to 1, indicating Karel's starting position. It also sets beepers to 0.

```
public void directKarelToDown() {...}
```
orients Karel downwards.

```
public void directKarelToUp() {...}
```
orients Karel upwards.

```
public void directKarelToRight() {...}
```
orients Karel to the right.

```
public void directKarelToLeft() {...}
```
orients Karel to the left.

```
public void moveWithIncrement() {...}
```

function moves Karel one step forward and increments the steps counter.

```
public void putBeeperAndIncrement() {...}
```

function checks if there is no beeper present in Karel's current location. If there is no beeper, it places one and increments the beepers counter.

Finally, here's what happens in the run method:

```java
public void run() {
    setBeepersInBag(1000);
    exploreMap();
    outerChambers();
    innerChambers();
    System.out.println("Number of steps : "  +getSteps());
    System.out.println("Number of Beepers : "  +getBeepers());
}
```

Test Cases

X	Y	Steps	Beepers	Algorithm	Equality of X and Y
16	10	106	68	MagicMoves	X>Y
10	16	107	68	MagicMoves	Y>X
20	20	190	16e	MagicMoves	Equals
9	18	102	54	General	Y>X
21	18	181	110	General	X>Y
14	27	173	102	General	Y>X
25	15	151	83	General	X>Y
10	10	85	56	MagicMoves	Equals
20	10	120	76	MagicMoves	Y>X
6	9	54	26	General	Y>X
7	7	50	25	General	Equals
7	9	55	27	General	Y>X
10	35	173	98	General	Y>X
35	10	174	98	General	X>Y
47	47	450	265	General	Equals
50	50	505	376	MagicMoves	Equals