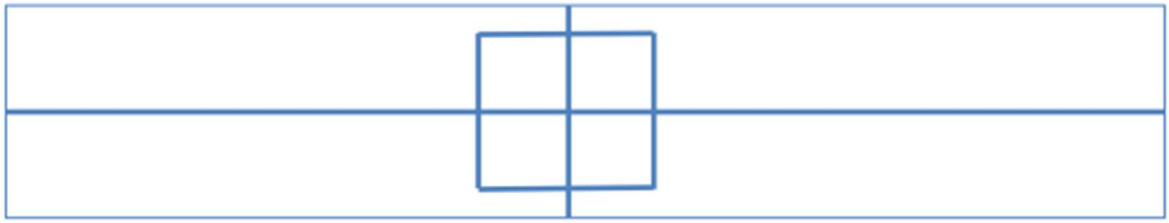
Karel Assignment

Rawan Afaneh

What are we required to do in this assignment:

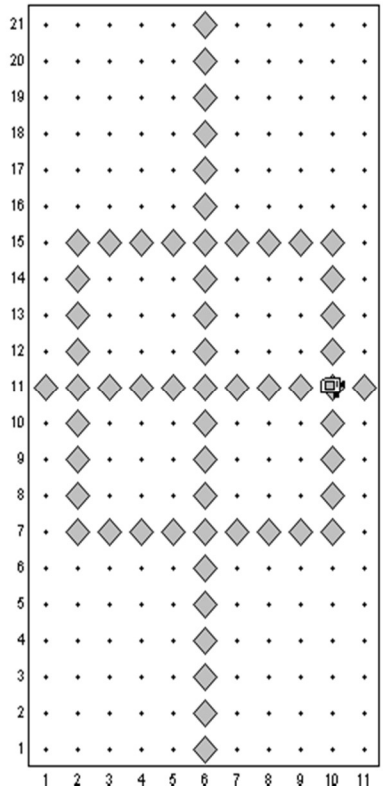
Divide a given map into 4 + 4 as shown in the diagram – Do all the analysis to handle the special cases. i.e. maps that can't be divided into the required shape because it's too small.

Notes:
- The inner chambers should be the biggest possible equal's squares.
- The outer chambers should be equal in size, and they should be L-shaped (they shouldn't be rectangles or squares).
- You are allowed to use double lines of beepers if you need to, however, you need to observe that beepers use should be optimized.

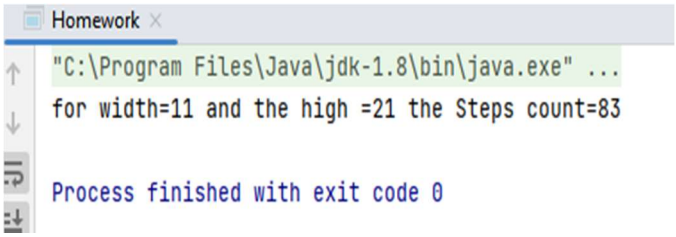


To initiate the report, I will provide a comprehensive overview of the code's solution, encompasses the consideration of various scenarios during the coding process:

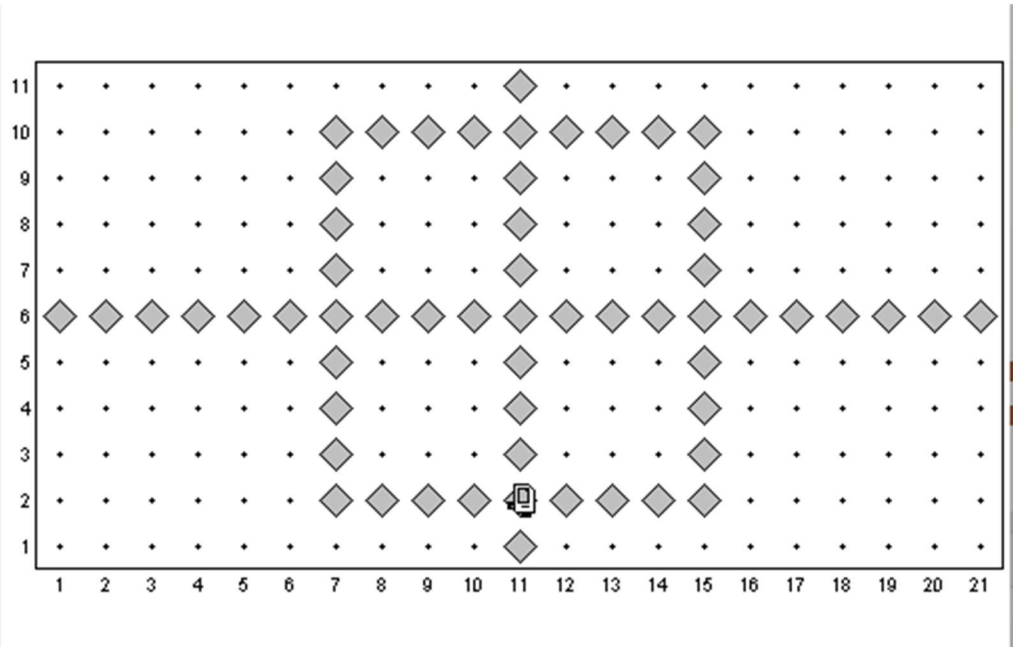
In the scenario where the width is less than the height and both are odd:



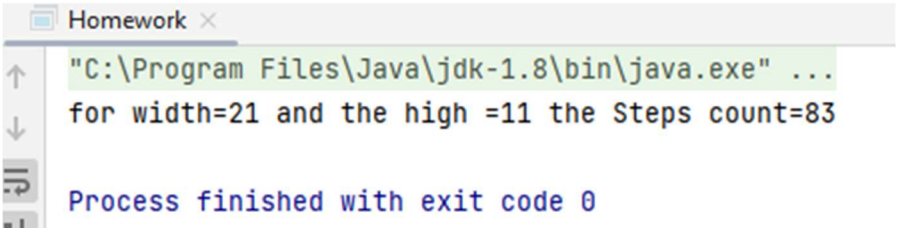
The steps that Karel takes to draw it are as follows:



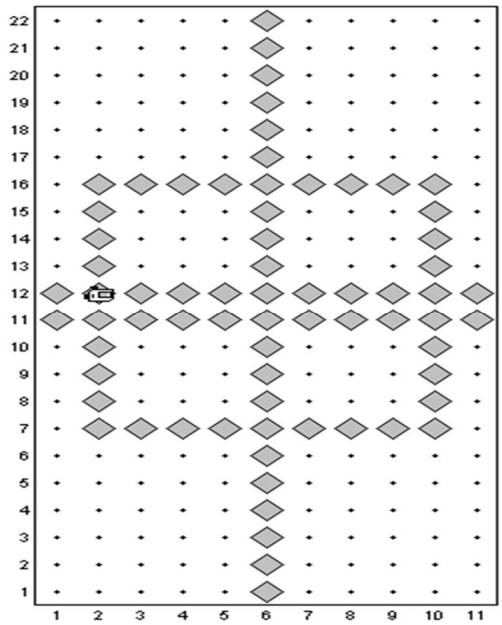
Here is another scenario when both the width and height are odd, but the width is greater than the height:



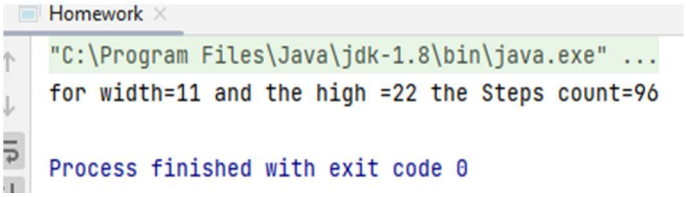
The steps that Karel takes to draw it are as follows:



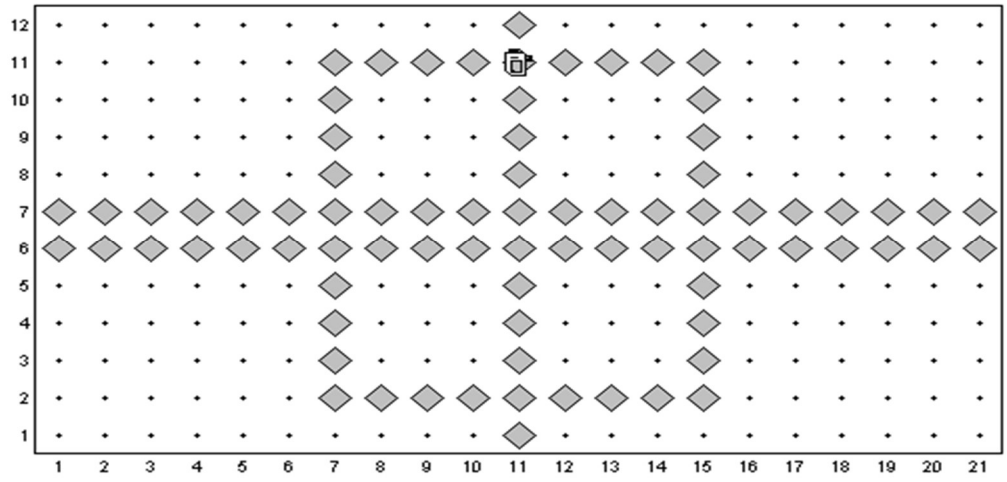
In this specific scenario where the width is less than the height, and the width is odd while the height is even:



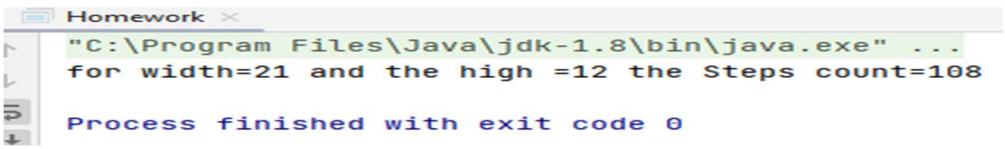
The steps that Karel takes to draw it are as follows:



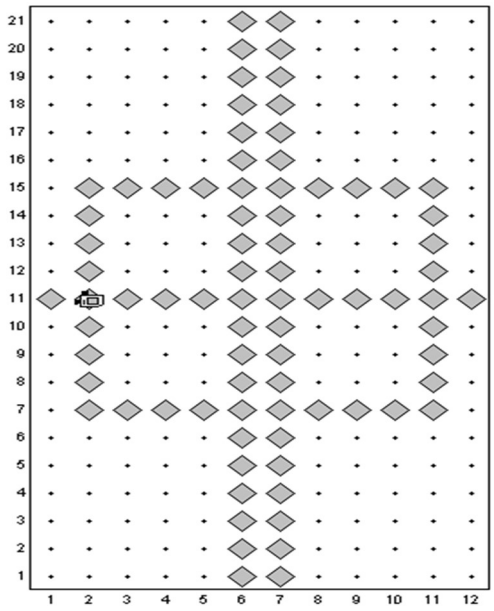
In this scenario where the width greater than the height, and the width is odd while the height is even:



The steps that Karel takes to draw it are as follows:



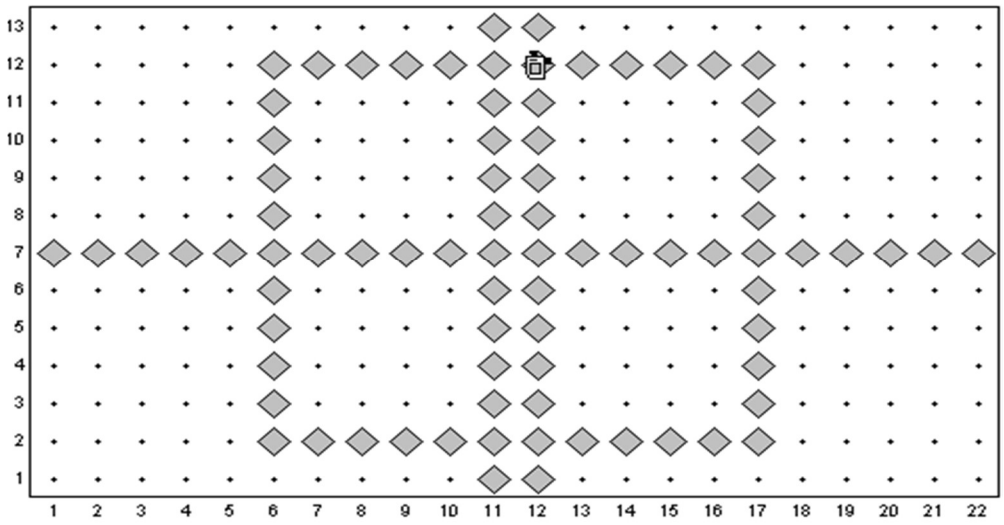
In this scenario where the width smaller than the height, and the width is even while the height is odd:



The steps that Karel takes to draw it are as follows:

```
Homework
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
for width=12 and the high =21 the Steps count= 107

Process finished with exit code 0
```

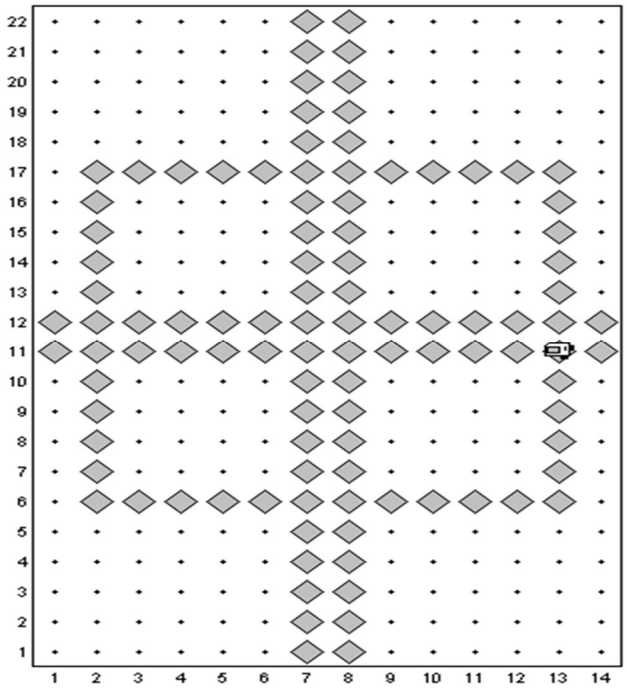
In this scenario where the width greater than the height, and the width is even while the height is odd:



The steps that Karel takes to draw it are as follows:

```
Homework
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
for width=22 and the high =13 the Steps count= 112

Process finished with exit code 0
```

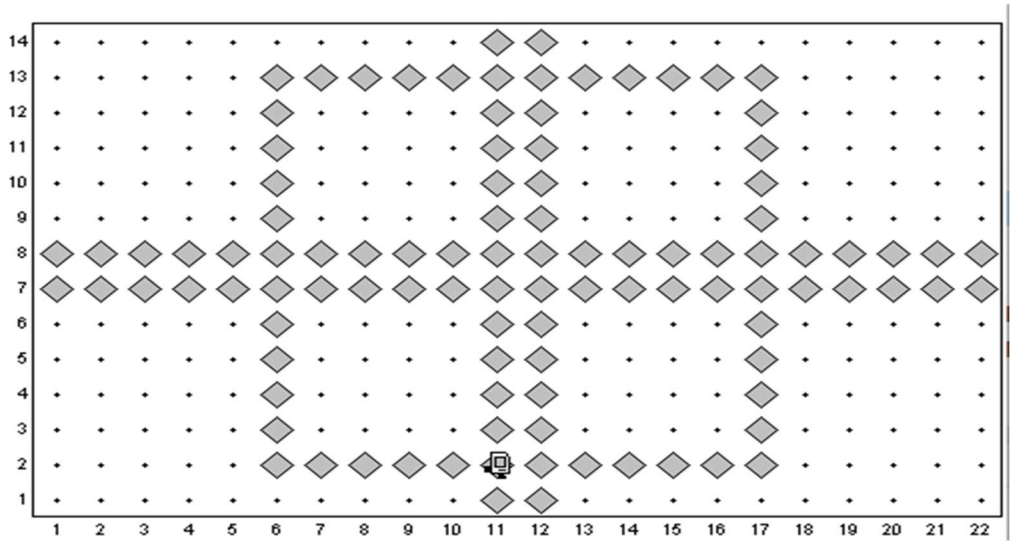
In this scenario where the width smaller than the height, and both of them are even:



The steps that Karel takes to draw it are as follows:

```
Homework
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
for width=14 and the high =22 the Steps count= 136

Process finished with exit code 0
```

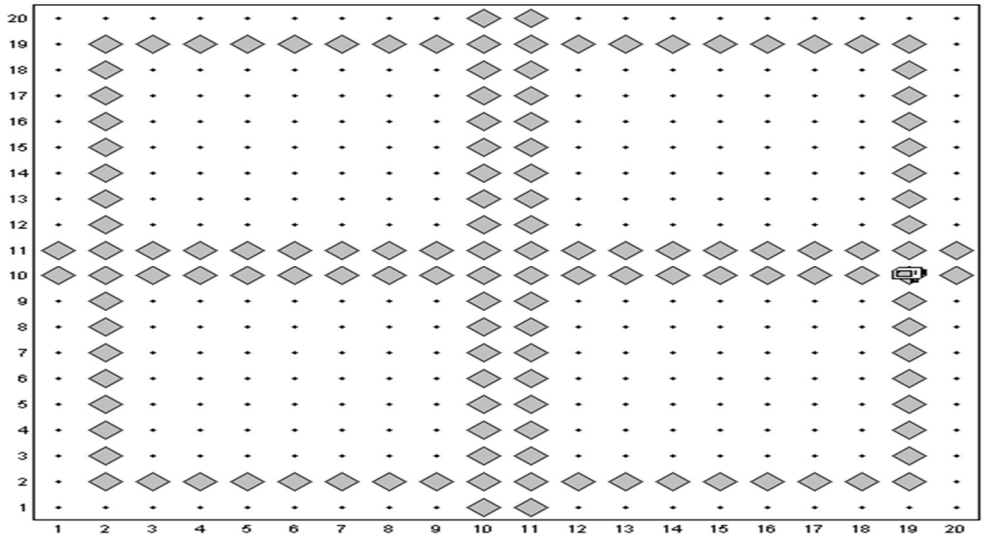
In this scenario where the width greater than the height, and both of them are even:



The steps that Karel takes to draw it are as follows:

```
Homework
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
for width=22 and the high =14 the Steps count= 139

Process finished with exit code 0
```

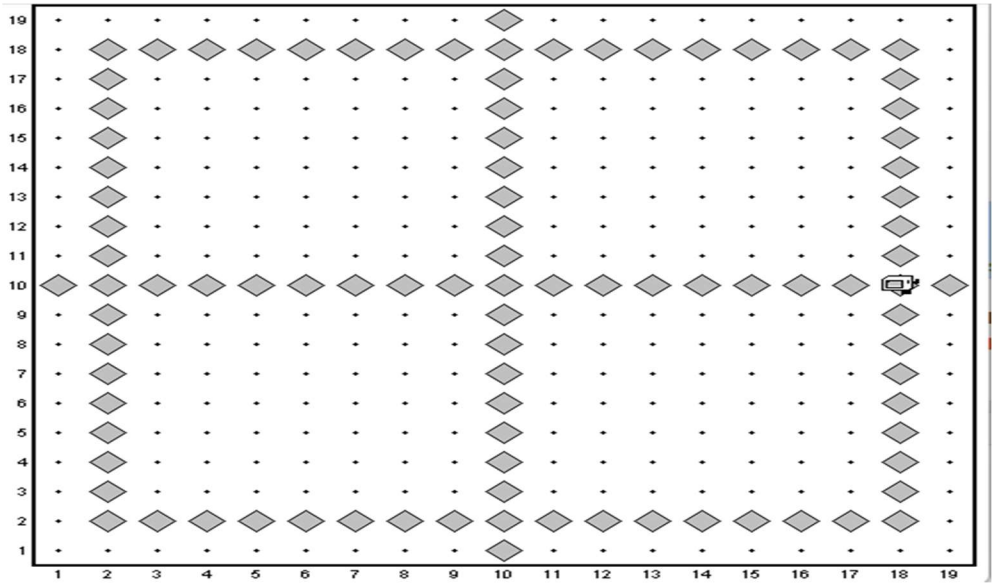
In this scenario, where both the width and height have equal sizes and are even:



The steps that Karel takes to draw it are as follows:

```
Homework
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
for width=14 and the high =22 the Steps count= 136

Process finished with exit code 0
```

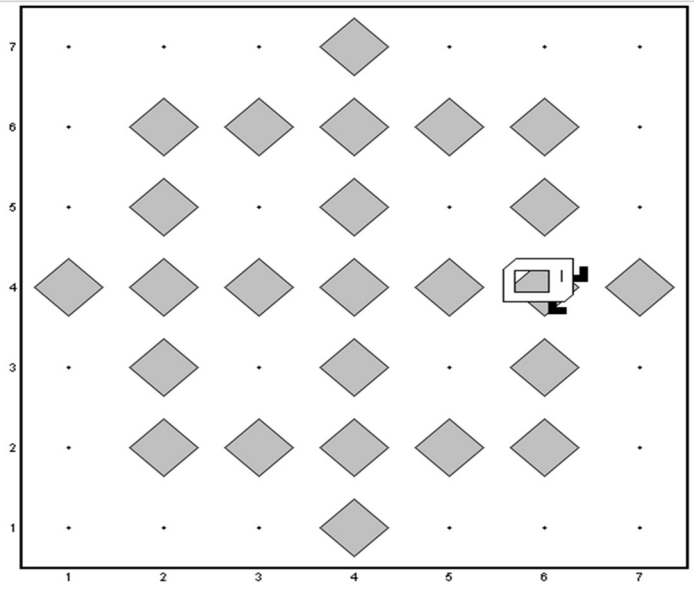
In this scenario, where both the width and height have equal sizes and are odd:



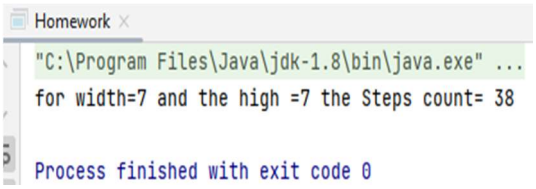
The steps that Karel takes to draw it are as follows:

```
Homework
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
for width=19 and the high =19 the Steps count= 128

Process finished with exit code 0
```

This refers to the smallest squares that Karel, the robot, will perform:



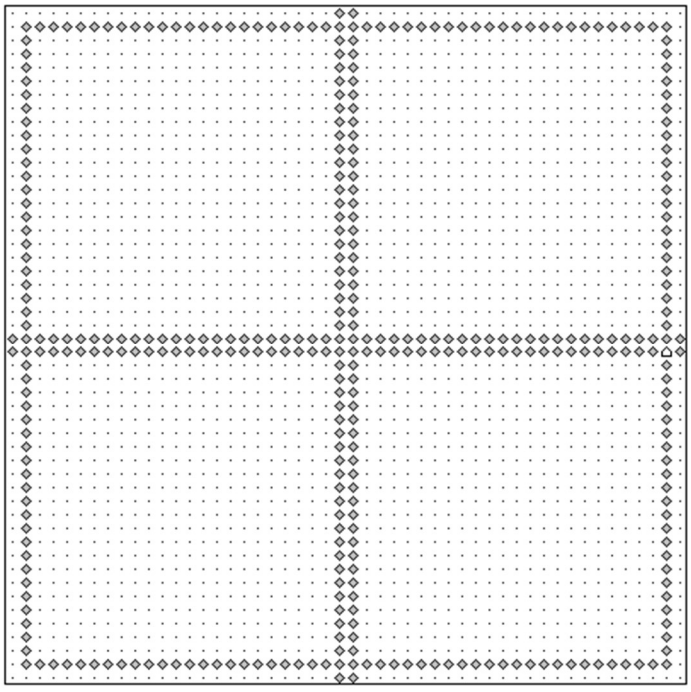
The steps that Karel takes to draw it are as follows:



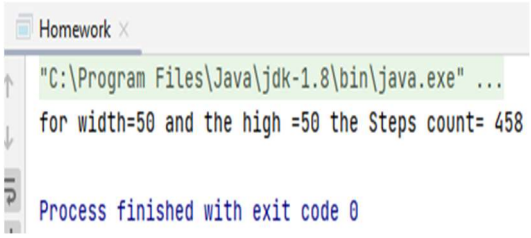
Smaller than that squares will not be accepted, and Karel will not execute any actions

---

This refers to the biggest squares that Karel, the robot, will perform:



The steps that Karel takes to draw it are as follows:



First, I will present the code implementation of the solution. After presenting the code, I will proceed to explain each method in detail:

```
import stanford.karel.SuperKarel;
import java.lang.Math;
public class Homework extends SuperKarel {
    private int countstep;

    public void run() {
        countstep=0;
        int width = getWorld().getColumns();
        int height  = getWorld().getRows();
        if(width<7||height <7){return;}
        if (height  < width) {
            drawthestrightline(height , width,
height , width);
        } else drawthestrightline(height , width,
width, height );

        int step = step(width, height );
        if (width % 2 == 0 && height  < width) {
            move();
            countstep++;
        }
        motion(Math.min(height, width), step);
        if ((height  % 2 == 0 && ((height  >
width) || height  == width)) || (height  % 2 == 0
&& width % 2 == 0) || (width % 2 == 0 && height
< width)) {
            move();
            countstep++;
        }
        motion(Math.min(height, width), step);
        System.out.println("for width=" +width +"
and the high ="+height +" the Steps count= "+
countstep);
    }
    private void putdouble(int high, int width) {

        int count1 = 0;
        while (count1 != 2) {
            move();
            countstep++;
            if (!beepersPresent()) {
                putBeeper();
            }
            if (frontIsBlocked() && count1 == 0)
{
                if (high > width || high ==
width) {
                    turnRight();
                } else turnLeft();
                move();
                countstep++;
                if (high > width || high ==
width) {
                    turnRight();
                } else turnLeft();
                if (!beepersPresent()) {
                    putBeeper();
                    count1++;
                }
            }
            if (frontIsBlocked() && count1 == 1)
break;
        }

    }
```

```
private void gotowall() {
    while (!frontIsBlocked()) {
        move();
        countstep++;
    }
    turnLeft();
}

private void gotomiddle(int x) {
    int count = 0;
    while (count != (x - 1) / 2) {
        move();
        countstep++;
        count++;
    }
}

private void putone() {

    while (!frontIsBlocked()) {
        if (!beepersPresent()) {
            putBeeper();
        }
        move();
        countstep++;

    }
    putBeeper();
}

private int stepsstright(int steps) {
    int step = steps;
    while (steps != 0) {
        move();
        countstep++;
        if (!beepersPresent()) {
            putBeeper();
        }
        steps--;
    }
    return step;

}

private int step(int w, int h) {
    int stepw = (w / 2);
    int steph = (h / 2);
    int step;
    if (stepw < steph) {
        step = stepw - 1;
        if (w % 2 == 0) {
            step = stepw - 2;
        }
    } else {
        step = steph - 1;
        if (h % 2 == 0) step = steph - 2;

    }

        move();
        countstep++;

    turnRight();
    return step;

}
```

```
  private void motion(int h, int step) {
        stepsstright(step);
        turnLeft();
        stepsstright(step);
        if (h % 2 == 0) {
            move();
            countstep++;
        }
        stepsstright(step);
        turnLeft();
        stepsstright(step);
    }

    private void drawthestrightline(int height , int width, int x, int y) {
        if (height < width) {
            turnLeft();
        }
        gotomiddle(x);
        if (x % 2 == 0) {
            putBeeper();
            if (height  < width) {
                turnRight();
            } else turnLeft();
            putdouble(height , width);
        } else {
            if (height  < width) {
                turnRight();
            } else turnLeft();
            putone();
        }
        turnLeft();
        gotowall();

        gotomiddle(y);

        if (y % 2 == 0) {
            if (height  < width) {
                move();
                countstep++;
            }
            putBeeper();
            turnLeft();
            putdouble(height , width);

        } else {
            turnLeft();
            putone();
        }
        turnAround();

    }
}
```

To ensure a clear and systematic presentation, I will start by explaining each method employed in the code before moving on to the run method:

1- `putdouble(int high, int width)` method :

```
private void putdouble(int high, int width) {

 int count1 = 0;
 while (count1 != 2) {
     move();
     countstep++;

     if (!beepersPresent()) {
         putBeeper();
     }
     if (frontIsBlocked() && count1 == 0) {
         if (high > width || high == width) {
             turnRight();
         } else turnLeft();
         move();
         countstep++;
         if (high > width || high == width) {
             turnRight();
         } else turnLeft();
         if (!beepersPresent()) {
             putBeeper();
             count1++;
         }
     }
     if (frontIsBlocked() && count1 == 1) break;
 }

}
```

This method will be invoked when Karel needs to draw two lines beside each other, specifically in the case where the number is even.

To track if Karel reaches the wall, a variable named **count1** is initialized and incremented each time Karel encounters a wall during its movement.

A while loop is utilized to break when **count1** equals 2, indicating that Karel has reached the second wall. The movement process involves initially moving straight towards the first wall and then turning back in the next column to reach the second wall.

During Karel's movement, a check is performed to determine if a beeper is present by **!beepersPresent()** If no beeper is present, Karel will place a beeper at the current position. However, if a beeper is already present, Karel will skip placing another beeper and move over it without adding an additional one. This logic ensures that only one beeper is placed at each position, preventing unnecessary duplication.

If the width is greater than the height, the movement strategy of Karel is indeed adjusted to optimize for fewer steps. The code is designed to prioritize efficient movement patterns in this particular scenario. To minimize the number of steps taken by Karel, the code determines the most optimal direction and path for movement. It considers factors such as starting position, turning angles, and path planning to ensure Karel moves efficiently towards the desired goal. By adjusting the movement strategy, the code

aims to reduce unnecessary backtracking or redundant movements, leading to a more efficient traversal of the grid.
By taking into account the difference in width and height, the code optimizes Karel's movement to achieve the task while minimizing the total number of steps required. This optimization enhances the overall efficiency and performance of Karel's movement in cases where the width is greater than the height.

2- `putone()` method :

```
private void putone() {

   while (!frontIsBlocked()) {
       if (!beepersPresent()) {
           putBeeper();
       }
    move();
    countstep++;

     }
   putBeeper();
 }
```

This method is invoked when Karel needs to draw a single line from where the Karel is and go straight to the wall, specifically in the case where the number is odd.

A while loop is used to break when Karel's front is blocked, indicating that it has reached the wall. This is checked using the **!frontIsBlocked()** method.

Within the loop, there is a check to determine if there is a beeper present at Karel's current position. If a beeper is present, Karel will skip over it and move to the next position without adding an additional beeper. However, if there is no beeper at the current position, Karel will place a beeper and then move to the next position. This ensures that only one beeper is placed at each position, preventing duplicates. By following this logic, Karel can effectively draw a single line while skipping over any existing beepers and placing a beeper at empty positions.

3- `gotowall()` method:

```
private void gotowall() {
    while (!frontIsBlocked()) {
        move();
        countstep++;
    }
    turnLeft();
}
```

This method is invoked to make Karel move towards the wall. It utilizes a while loop to continuously move Karel until it reaches the wall.

4- `gotomiddle (int x)` method:

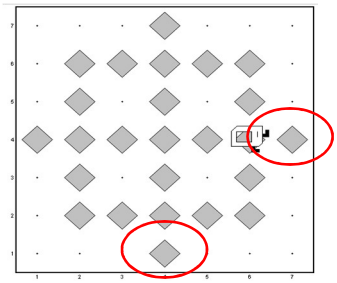
```
private void gotomiddle(int x) {
    int count = 0;
    while (count != (x - 1) / 2) {
        move();
        countstep++;
        count++;
    }
}
```

This method takes a variable **x** as input, representing the value that Karel will go halfway towards. The primary objective of this method is to move Karel towards the midpoint of **x** using a while loop and keep track of the count until it reaches the halfway point.

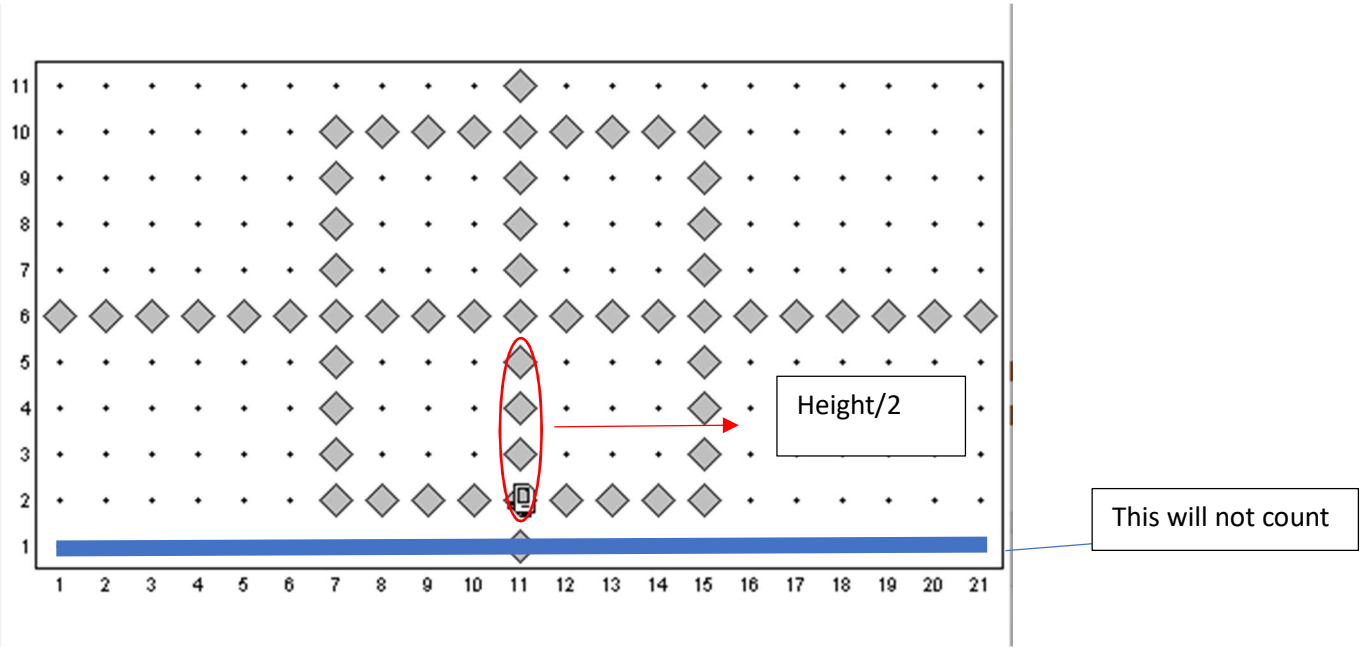
5- `step (int w, int h)` method:

```
private int step(int w, int h) {
    int stepw = (w / 2);
    int steph = (h / 2);
    int step;
    if (stepw < steph) {
        step = stepw - 1;
        if (w % 2 == 0) {
            step = stepw - 2;
        }
    } else {
        step = steph - 1;
        if (h % 2 == 0) step = steph - 2;

    }

    move();
    countstep++;

    turnRight();
    return step;

}
```

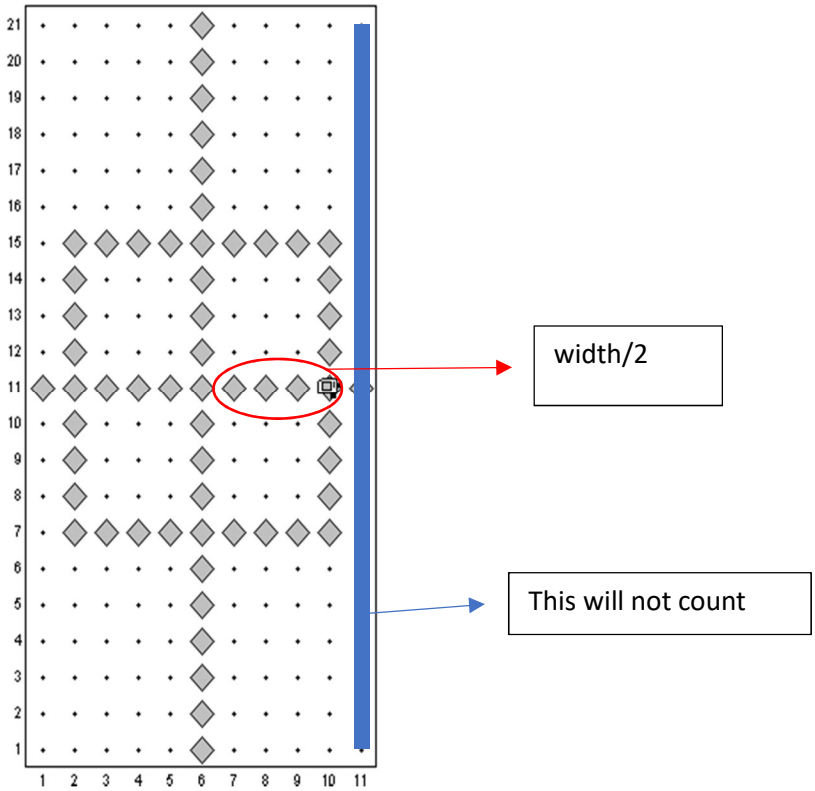
This method calculates the dimensions of a square and determines the number of steps that Karel needs to move to create the square's side length.
To calculate the length of the square, I pass the width and height as parameters to the method. Inside the method, I assign the value of half the width **(w/2)** to a variable called **stepw**, and you assign the value of half the height **(h/2)** to a variable called **steph**.
the move present as this step:



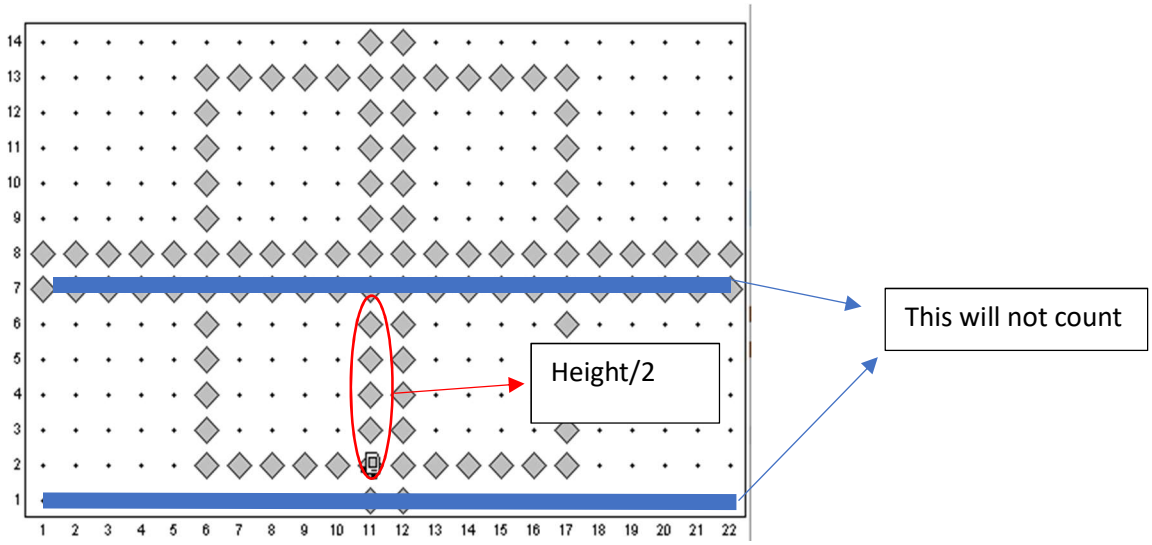
To calculate the steps if width greater than the height and odd:
I get the half of the height and minus from it the m



To calculate the steps if width smallest than the height and odd:
I get the half of the width and minus from it the m



To calculate the steps if width greater than the height and even:
I get the half of the height and minus from it the m and one of the double line



And so on.
Then the Karel will move 1 that present the step
And the method will return the number of the step.

7- `stepsstright (int steps)` method:

```
private int stepsstright(int steps) {
    int step = steps;
    while (steps != 0) {
        move();
        countstep++;
        if (!beepersPresent()) {
            putBeeper();
        }
        steps--;
    }
    return step;

}
```

This method takes the number of steps as a parameter, representing the number of steps that Karel needs to move.

Inside the method, a while loop is used to ensure that Karel continues moving until the number of steps reach 0. The loop condition checks if the steps variable is greater than 0, indicating that there are still steps remaining.

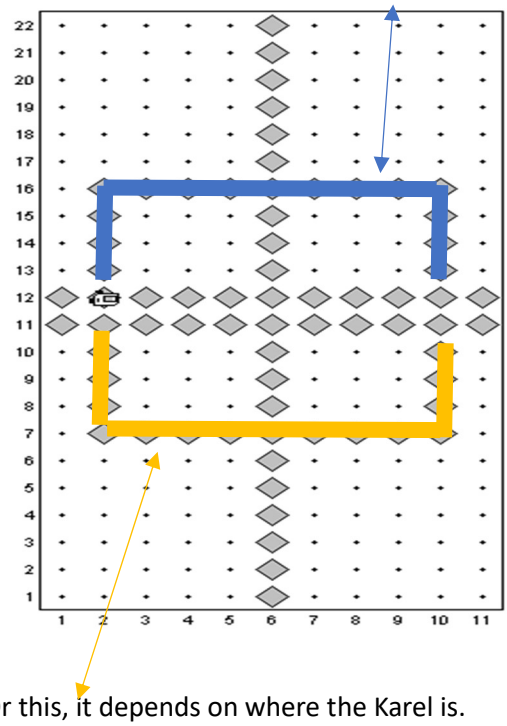
During this movement, Karel checks if there is a beeper present in the current position. If there is no beeper, Karel places a beeper in that position. However, if a beeper is already present, Karel skips placing a new beeper and proceeds to the next position.

By checking for the presence of a beeper and making decisions based on it, the code ensures that Karel only places a beeper where needed, avoiding duplications.

8- `motion (int h, int step)` method:

```
private void motion(int h, int step) {
        stepsstright(step);
        turnLeft();
        stepsstright(step);
        if (h % 2 == 0) {
            move();
            countstep++;
        }
        stepsstright(step);
        turnLeft();
        stepsstright(step);
    }
```

this method make the Karel do this :



Or this, it depends on where the Karel is.

And we pass to this method “h” that present either height or width and the number of the steps

```
if (height  < width) {
        motion(height , step);
    } else motion(width, step);
```

and in this condition

```
if (h % 2 == 0) {
            move();
            countstep++;
        }
```

it will see if the parameter that I pass is even or not, if it’s even the Karel will skip on line of the double lines. Then continue to draw the next straight line.

9- `drawthestrightline (int height , int width, int x, int y)` method:

```
private void drawthestrightline(int height , int width, int x, int y) {
    if (height < width) {
        turnLeft();
    }
    gotomiddle(x);
    if (x % 2 == 0) {
        putBeeper();
        if (height  < width) {
            turnRight();
        } else turnLeft();
        putdouble(height , width);
    } else {
        if (height  < width) {
            turnRight();
        } else turnLeft();
        putone(height , width);
    }
    turnLeft();
    gotowall();

    gotomiddle(y);

    if (y % 2 == 0) {
        if (height  < width) {
            move();
            countstep++;
        }
        putBeeper();
        turnLeft();
        putdouble(height , width);

    } else {
        turnLeft();
        putone(height , width);
    }
    turnAround();

}
```

This method will pass to it the height and the width and where the Karel will go first and where Karel will go second.

```
if (height < width) {
            turnLeft();
        }
```

The if statement checks if the height is smaller than the width. If this condition is true, it means that Karel needs to move in the height direction first. To accomplish this, the code instructs Karel to turn left. By turning left in this scenario, Karel adjusts its orientation to align with the height direction. This ensures that Karel moves along the shorter dimension first, as it results in fewer steps compared to moving along the longer dimension.

the code proceeds to make Karel move to the middle of the specified dimension by **gotomiddle(x)**. This ensures that Karel is positioned correctly to start drawing the double line or single line.

Once Karel is in the middle position, the appropriate method (depending on whether the dimension is even or odd) is called to draw the desired line by **putone**, **putdouble**. The previously mentioned methods

are used to accomplish this task efficiently, with beepers being placed or skipped based on their presence in each position.
After completing the line drawing, the code calls the **gotowall** method to make Karel move to the next wall. The **gotowall** method ensures that Karel reaches the wall and positions itself in the middle of it by **gotomiddle(y).** Finally, based on whether the dimension is even or odd, the code determines whether to draw a double line or a single line using the corresponding method. Then the Karel will turn around by **turnAround()** to be prepared to do the steps.

10- `run()` method:

```
            private int countstep;
                public void run() {
                countstep=0;
                int width = getWorld().getColumns();
                int height  = getWorld().getRows();
                if(width<7||height <7){return;}
                if (height  < width) {
                    drawthestrightline(height , width, height , width);
                } else drawthestrightline(height , width, width, height );

                int step = step(width, height );
                if (width % 2 == 0 && height  < width) {
                    move();
                    countstep++;
                }
                motion(Math.min(height, width), step);
                if ((height  % 2 == 0 && ((height  > width) || height  == width)) || (height
        % 2 == 0 && width % 2 == 0) || (width % 2 == 0 && height  < width)) {
                    move();
                    countstep++;
                }
                motion(Math.min(height, width), step);
                System.out.println("for width=" +width +" and the high ="+height +" the Steps
        count= "+ countstep);
            }
```

To keep track of Karel's movements, a parameter called `countstep` is created. This parameter serves as a counter to keep a tally of each move made by Karel throughout the program's execution. By incrementing `countstep` with each movement.
In the first of the code we obtaining the number of columns and rows, assigning them to variables named **height** and **width**, respectively. These variables store the dimensions of the grid or layout within which Karel will operate.
Next, a conditional check is performed to verify if both the **width** and **height** are smaller than 7. If this condition evaluates to **True**, it implies that the dimensions are smaller than the required minimum threshold for the squares to be accepted. In such a case, Karel will not execute any actions as per the code's design. This check ensures that the code operates within the specified constraints and prevents Karel from performing actions in scenarios where the dimensions do not meet the minimum requirement.

```
if (height  < width) {
            drawthestrightline(height , width, height , width);
```

```
} else drawthestrightline(height , width, width, height );
```

The code snippet I provided demonstrates a conditional statement that checks whether the height is smaller than the width. If this condition evaluates to true, the code executes the `drawthestrightline(height, width, height, width)`. This means that Karel will first draw the line along the height and then move to draw the line along the width.
The rationale behind this logic is to minimize the number of movements and steps taken by Karel. By starting with the shorter dimension (height) and then moving to the longer dimension (width), Karel can complete the task with fewer movements. This approach optimizes Karel's path to draw the lines efficiently.
On the other hand, if the height is not smaller than the width, indicating that the height is equal to or larger than the width, the code executes the height `drawthestrightline(height, width, width, height)`. In this case, Karel will first draw the line along the width and then move to draw the line along the height. The same principle applies here, aiming to minimize the number of movements and steps.
By dynamically selecting the order of dimensions to draw the lines, the code ensures optimal movement and efficiency for Karel in different scenarios.

After determining the appropriate sequence for drawing the lines based on the dimensions, the code proceeds to invoke the **step(width, height)** method and stores the returned value in the **step** variable. This step calculates the number of steps required to complete the drawing.
Following that, the code executes the necessary movements and drawing actions based on each scenario. This includes invoking the **motion** method, which is responsible for drawing the inner square based on the specific conditions of each scenario.
Throughout the execution, the `countstep` variable is incremented with each movement of Karel. This allows for tracking the number of steps taken during the drawing process.
Finally, the code prints out the value of the `countstep` variable, which represents the total number of steps Karel has taken to complete the drawing. This provides insight into the efficiency and complexity of the code, giving a measure of the overall movement required by Karel to accomplish the task.

Finally, I present my code and the solution I derived from it. After extensive testing and experimentation, I arrived at the following moves that minimize the overall movement and result in the least number of steps (countstep):

1. First, I analyze the dimensions of the square, considering the width and height.
2. Based on the dimensions, I determine the sequence and direction of movements to optimize for efficiency.
3. I utilize various methods, such as `run`, `drawthestrightline`, `motion`, `stepsstright`, `step`, `gotomiddle`, `gotowall`, `putone` and `putdouble` to control Karel's movements and drawing actions.
4. I implement conditionals and loops to handle different scenarios and ensure the correct execution of the drawing process.
5. Throughout the code, I carefully consider the presence of beepers and adjust Karel's actions accordingly.
6. I keep track of Karel's movement using the **countstep** variable, which increments with each step taken.
7. Finally, I print the value of **countstep** to display the total number of steps required to complete the drawing.

By optimizing the movements and considering various scenarios, I achieved a solution that minimizes Karel's movement and achieves the desired outcome efficiently. The resulting **countstep** value represents the steps required to draw the square.