Karel The Robot
Dividing World Assignment

Yazan Abdullah

Contents

List of Figures

Chapter 1

Notes

Throughout this Report, I will present my approach to solving "Karel the Robot" assignment.

This chapter contains miscellaneous notes that do not have a particular theme. If you want to get started with the solution, please refer to Chapter 2.

My focus was on writing clearly what I aimed to achieve at each point and how I attempted to achieve it. These details can sometimes get lost in the implementation, so I have made a conscious effort to keep them clear.

In this assignment we were instructed to make our code as short as possible. I adhered to this requirement; but at the same time, tried my best to not sacrifice readability.

Currently my code is 309 lines long; however, it should be noted that a lot of these lines are blank – 47 blank lines to be precise. Hence actual number of code lines is **262**.
Note: I obtained the number of blank lines by running the following shell command:

```
shell$ grep -c '^[[:space:]]*$' src/Homework.java
```

Chapter 2

Overview of my Approach

2.1 The `run()` Method

In this section, I will present the `run()` method, which outlines the flow of my code. The goal of this section is to provide an understanding of how I structured the problem. The finer details will be presented later in the document, for a visual aid Figure 2.1 shows the flowchart of the `run` method.

First, I reset Karel's attributes to ensure that different runs do not affect each other. Then I made Karel figure out the dimensions of the world, by moving to the top-right corner, and calculated some values that will be used throughout the code, such as the parity of the number of streets.

After that, I managed to reduce the amount of code significantly, by always assuming that the number of streets is no less than that of the avenues. When This is not the case, I altered Karel's facing direction, and swapped his left and right, so now the same algorithm will mirror its normal behavior and still work, (for further details on this, see Section 2.2).

Based on the dimensions of the world, I dispatched the problem to the proper handler: `handleSpecialWorlds()` for handling special cases, and to the more cost-effective method between `solveUsingParallelCuts()` and `solveUsingperpendicularCuts()`.

Note here, I compare the cost according to the following equation:

```
cost = costOfBeeper * requiredBeepers + costOfMove * requiredMoves
```

Since I don't have any way of knowing the values of `costOfBeeper` and `costOfMove`, I arbitrarily chose to assign them 1000 and 10 respectively. These values can be easily adjusted to suit the specific scenario we are working with.

Lastly, After the world is divided, I send Karel to the starting position at

corner $(1, 1)$ and output the relevant statistics to the console.

You can see Figure 2.1 for the flowchart of the method. Also, you can see my code below:

```
public void run() {
    resetKarel();
    findDimensionsOfWorld();
    calculateAttributes();
    if (streets < avenues)
        standardizeProblem();
    if (isSpecialWorld()) {
        handleSpecialWorlds();
    } else if (costOfPerpendicularCuts() <= costOfParallelCuts()) {
        solveUsingPerpendicularCuts();
    } else {
        solveUsingParallelCuts();
    }
    goToStartingPosition();
    System.out.printf("The number of used Beepers is %d\n",
        usedBeepers);
    System.out.printf("The number of moves made is %d\n",
        numOfMoves);
}
```

2.2 Standardizing the Problem

In this section, I will explain how I simplified my solution by standardizing the assigned problem.

The standard problem: Throughout my code, I assumed two things:

1. Karel stands the top-right corner, facing up.
2. The number of streets is no less than that of the avenues.

For an example of the standard problem see Figure 2.2.

The first assumption occurs naturally after calling `findDimensionsOfWorld()` method.

So, if the avenues are not greater than the streets, all the algorithms will execute normally.

However, in nearly half the cases, this is not true, meaning the avenues are greater than the streets. Here, my goal was to make a few modifications so that

the same algorithms for the normal case, would still execute correctly.

The first step was to make Karel turn right and then swap the values of streets and avenues. Now, Karel's position is similar to the normal one, except that it is in the top-left corner – imagine the world rotated. See Figure 2.3.

One straightforward solution is to move Karel to the opposite corner. While this solves the problem, it requires many unnecessary moves, so I had to look for something else.

The approach I settled on involves swapping Karel's left and right directions, the resulted behavior mirrors that of normal execution.

Following are the relevant pieces of code:

```
private Direction left = Direction.LEFT;
private Direction right = Direction.RIGHT;

private void standardizeProblem() {
    swap_dimensions();
    turnToDirection(right);
    swap_directions();
}

private void swap_directions() {
    var tmp = left; left = right; right = tmp;
}

private void swap_dimensions() {
    var tmp = streets; streets = avenues; avenues = tmp;
    calculateAttributes();
}

public void calculateAttributes() {
    oddNumOfAvenues = avenues%2 == 1;
    oddNumOfStreets = streets%2 == 1;
    halfAvenues = avenues/2;
    halfStreets = streets/2;
}
```

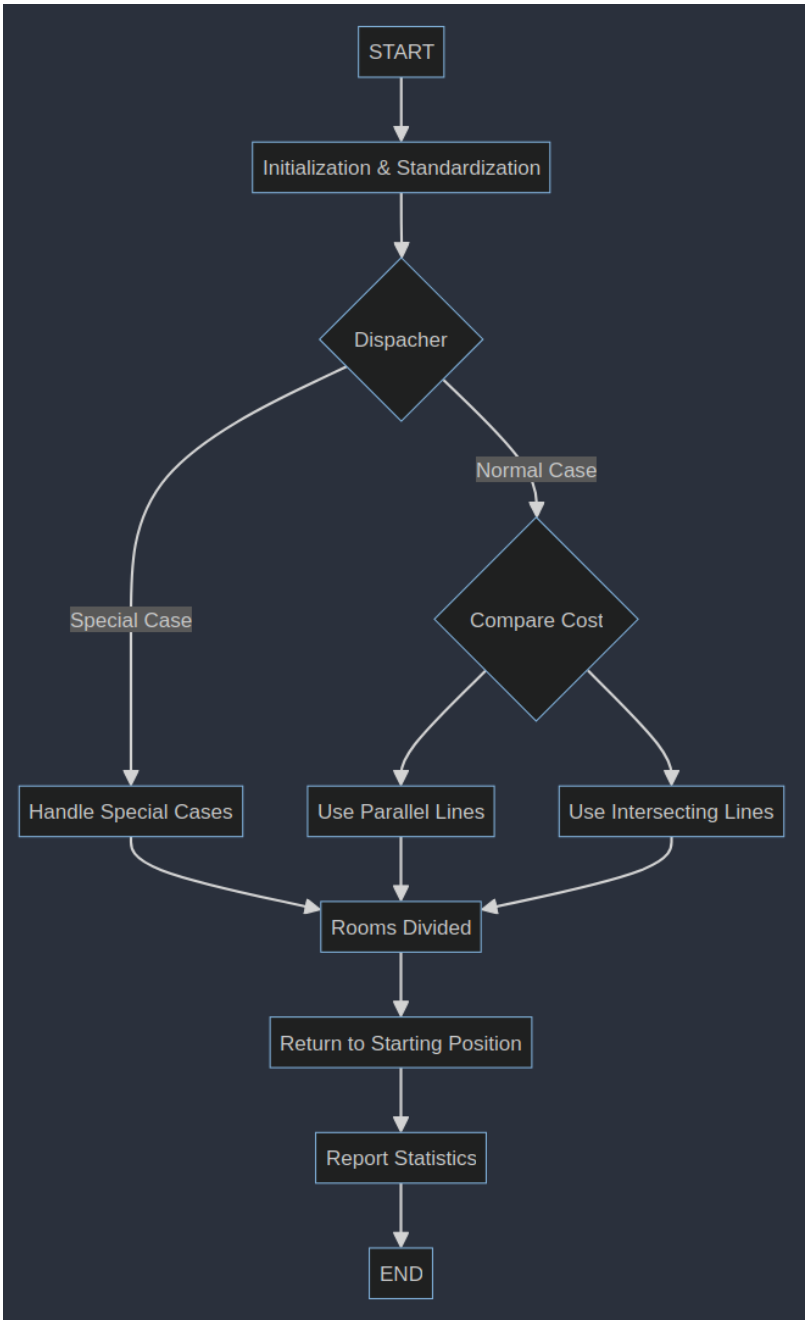

Figure 2.1: Flowchart of the run() Method

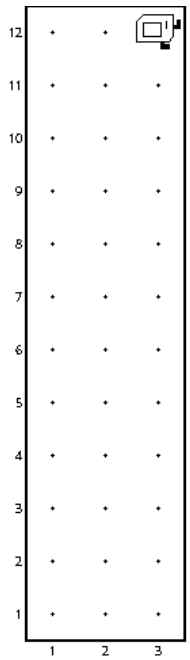

Figure 2.2: The Standard Problem

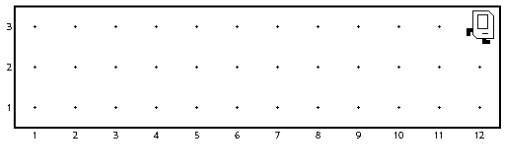

Figure 2.3: After turning to the right, it's like Karel's on the top left corner.

Chapter 3

Super Duper Karel: Extending Super Karel

In this chapter, I will explain what the `SuperDuperKarel` class is, and why I designed it. First, I will address the reasons behind its creation, then I will present details of the class.

For the given problem `SuperKarel` had rather limited functionality, so extending them and adding more methods was necessary. I chose to implement these additional methods, like move n steps forward, in a new class called `SuperDuperKarel`. This approach helped to declutter my solution and enhance its readability, even though this class added a few extra lines of code, see Figure 3.1.

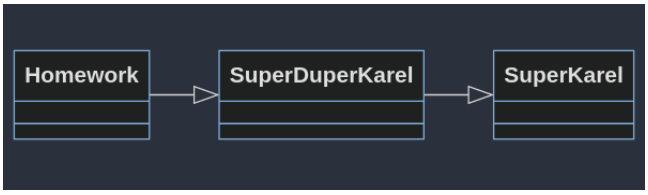

Figure 3.1: Class Heirarchy in My Code

`SuperDuperKarel` extends `SuperKarel` and adds several features to it, such as:

- Tracking the number of moves Karel makes by overriding the `move()` method.
- Tracking the number of beepers Karel puts or picks by overriding the `putBeeper()` and `pickBeeper()` methods.

- Has `putBeepersIfNecessary()` method, that checks whether the corner already has a beeper on it, so as to not waste unnecessary beepers.
- Has `moveIfYouCan()` method that will protect Karel from crashing into walls, by checking whether the `frontIsClear()` first.
- Has multiple `move*()` methods, that will significantly simplify the implementation any solution afterwards.

The following code block shows all the methods `SuperDuperKarel` has. *The implementation of all the methods are very simple and straightforward, so I won't include them in this report.*

```
abstract class SuperDuperKarel extends SuperKarel {
    protected int numOfMoves, usedBeepers;

    protected void resetStatistics();
    @Override public void move();
    @Override public void putBeeper();
    @Override public void pickBeeper();
    public void putBeeperIfNecessary();
    public void moveIfYouCan();
    public int moveTillWall();
    public void move(int moves, boolean putBeepersOnTheWay);
    public void turnToDirection(Direction direction);
    public void move(Direction direction, int moves, boolean
        putBeepersOnTheWay);
}
```

Note I used an `enum` class called `Direction` to indicate the direction Karel should turn to. The following code block contains the implementation of `Direction`:

```
enum Direction {
    LEFT, RIGHT, BACKWARDS, FORWARDS;
    public Direction opposite() {
        return switch (this) {
            case LEFT -> RIGHT;
            case RIGHT -> LEFT;
            case FORWARDS -> BACKWARDS;
            case BACKWARDS -> FORWARDS;
        };
    }
}
```

This `enum` allowed me to parameterize the direction in my algorithm, which made it possible for me to simplify and standardize the world I am trying to divide. For more details on this, see Section 2.2.

Chapter 4

First Method: Parallel Cuts

This chapter explains the first of two methods I used to divide Karel's worlds; this one is most effective when the difference between the number of streets and avenues is big, that is the world looks like a narrow rectangle, see figure 2.2.

Note: This method doesn't work on arbitrary worlds, so before using it, you need to make sure the world is not a special world, (use `isSpecialWorld()` method), and that at least one of the dimensions is greater than or equal to 7.

This chapter is divided in two sections, the first explains the algorithm for dividing the world, and the second presents an analysis of cost.

4.1 The Algorithm

In this section, I will present the `solveUsingParallelCuts()` method, and explain how it works.

The algorithms starts by computing the number of cells of each room can have, It does so by subtracting the number beepers necessary to divide the world from the number of all corners available, and then dividing the result by 4.

Each cut, will mark the boundaries of a new room, and will require `Math.min(streets, avenues)` beepers. Sometimes, the remaining cells are not multiples of 4, so some more cells will need to be covered with beepers to make the rooms equal, the number of such cells is stored in the variable `taken`.

After that, I wrote two `for` loops, the first is to cover the `taken` cells that I talked about in the previous paragraph, and the second will build the boundaries of each room. Note that it runs for three iterations, as we need three cuts to divide four rooms.

After the first `for` loop is finished, `taken` will be used in a **different** way. At every iteration, it is assumed that Karel will stand on the first row of the next room, and is expected to put beepers in a way that this room will have `roomSize` corners in it; but, sometimes, not all the row will be empty, in such cases the number of covered corners in the first row of the crrent room will be stored in `taken`, and to handle this case I would simply ask Karel to fill this room with `roomSize + taken` corners, so after it finishes, I would have the right number of corners in the room.

To reduce the number of movements Karel makes, I made Karel move in a way similar to *snakes*, that is going from right to left and then from left to right and so on. To achieve this I declared the local variable `dir`, which denoted the direction of movements in this iteration, and inverted its direction at the end of every iteration.

The following code block, shows the `solveUsingParallelCuts()` method:

```
private void solveUsingParallelCuts() {
    int roomSize = (streets*avenues - costOfParallelCutsBeepers()) /
        4;
    int taken = (streets*avenues - 4 * roomSize - 3 *
        Math.min(streets, avenues));

    turnToDirection(left);

    Direction dir = left;

    for (int i = 0, rows = taken/avenues; taken > 0 && i <= rows;
        ++i, dir = dir.opposite()) {
        if (avenues <= taken) {
            move(avenues - 1, true);
            taken -= avenues;
        } else {
            move(taken - 1, true);
            move(avenues - taken, false);
        }
        move(dir, (beepersPresent() ? 1 : 0), false);
        turnToDirection(dir);
    }

    for (int i = 0; i < 3; ++i) {
        int shifted = (roomSize+taken)%avenues;

        move(dir, (roomSize+taken) / avenues, false);
        move(dir.opposite(), avenues - 1 - (shifted), true);
        moveDiagonally(dir);
        if (shifted > 0) {
            move(forwards, shifted-1, true);
```

```
        }
        turnAround();
        taken = shifted;
        dir = dir.opposite();
    }
}
```

4.2 Cost Analysis

In this section I will present, the cost analysis for Parallel cuts method.

In this method, Karel will put beepers for two different reasons: first, to divide the world into four rooms, and second, to ensure all the rooms are equal in size.

To divide the world into four rooms, Karel would need to make 3 cuts, see Figure 4.1 To reduce the cost, these cuts will be across the smaller dimension.

Sometimes, after subtracting the beepers used to make the cuts, the number of remaining corners is not divisible by four, hence more beepers will be needed to cover these additional corners, See Figure 4.2.

See the code block below, for the implementation:

```
private int costOfParallelBeepers() {
    if (Math.max(streets, avenues) < 7)
        return streets * avenues; // infinity

    int cuttingBeepers = 3 * Math.min(streets, avenues);
    return cuttingBeepers + (streets*avenues - cuttingBeepers) % 4;
}
```

Following is the analysis of the number of moves made by Karel:

- `streets` + `avenues` - 2: to find the dimensions of the world.
- 3 * (`avenues-1`): to make the three cuts.
- `streets` - 1: this sum is accumulated across the entire algorithm; basically the reasoning for it, is that Karel would need this number of movements to get back to the initial row, and it doesn't move upwards, so over the algorithm, it will move this number of steps downwards.
- 2 * (`avenues-1`): this only happens when there are "left-over" corners that need to be covered.

Therefore, the following equation calculates the approximate number of movements:

```
private int costOfParallelMoves() {
    return 2 * (streets-1) + 4 * (avenues-1) + 2 *
        ((streets*avenues-3*avenues)%4 > 0 ? avenues-1 : 0);
}
```

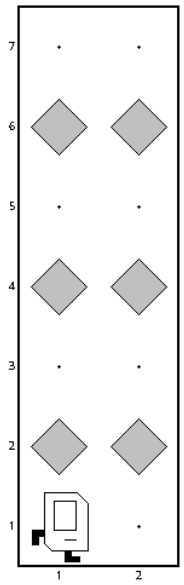

Figure 4.1: We need three cuts, to divide the world into 4 rooms

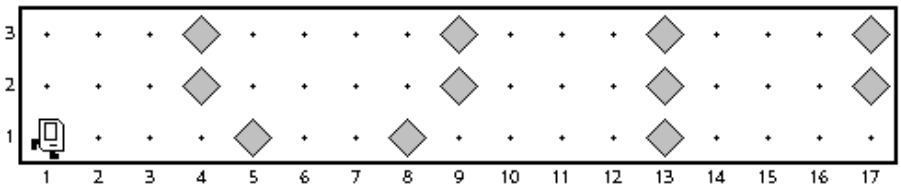

Figure 4.2: Sometimes, more beepers are needed to equalize the rooms

Chapter 5

Second Method: Perpendicular Cuts

This chapter explains the second of two methods I used to divide Karel's worlds; this one is best suited when the difference between the number of streets and avenues is small, i.e., when the world looks similar to a square.

Note: This method doesn't work on arbitrary worlds, so before using it, you need to ensure the world is not a special world, (use `isSpecialWorld()` method), and that both streets and avenues be greater than 1.

This chapter is divided into two sections, the first explains the algorithm for dividing the world, and the second presents an analysis of cost.

5.1 The Algorithm

In this section, I will present the `solveUsingperpendicularCuts()` method, and explain how it works.

The algorithms classifies the worlds into three categories, based on the parity of the dimensions:

1. Odd number of streets and odd number of avenues.
2. Either dimension is even but not both.
3. Even number of streets and even number of avenues.

Each of these categories will be explained in its own subsection.

5.1.1 Both Dimensions are Odd

In this subsection, I will explain how I divide worlds with odd dimensions using perpendicular cuts.

This case is the simplest. When both dimensions are odd, there will be a middle row and a middle column, such that filling either of them with beepers separates the world into two equally-sized rooms. By splitting both, we get four equally-sized rooms.

Here is the code for this part:

```
if (oddNumOfStreets && oddNumOfAvenues) {
    move(left, halfAvenues, false);
    move(left, streets-1, true);
    move(left, halfAvenues, false);
    move(left, halfStreets, false);
    move(left, avenues-1, true);
}
```

5.1.2 Dimensions with Different Parity

In this subsection, I will explain how I divide worlds with dimensions having different parity using perpendicular cuts.

This case is trickier than the previous one. My approach was to find the odd dimension and split the world into two equal parts from its middle cell. Then, I would split each of the empty parts in two, using an S-turn. Sometimes, one additional beeper (for each part) would be used to cover an empty corner when the split yielded an areas that cannot be divided in two.

To reduce the number of lines, I wrote the code assuming the number streets is odd. And used `standardizeProblem()` method from Section 2.2 to make the same piece of code work for the other case.

Following is the code for this case:

```
} else if (oddNumOfStreets != oddNumOfAvenues) {
    if (oddNumOfAvenues) standardizeProblem();

    move(backwards, halfStreets, false);
    move(right, avenues-1, true);
    move(right, halfStreets, false);
    move(right, halfAvenues-1, false);
    move(right, (halfStreets+1)/2 -1, true);
    move(left, 1, false);
    move(right, 1, halfStreets%2 == 1);
```

```
        move(forwards, halfStreets, true);
        move(right, 1, false);
        move(left, 1, halfStreets%2 == 1);
        move(forwards, halfStreets/2-1, true);
    }
```

5.1.3 Both Dimensions are Even

In this subsection, I will explain how I divide worlds with even dimensions using perpendicular cuts.

This is the trickiest of all three cases. It is very similar to the previous subsection, but since neither dimension is odd, after splitting either dimension, the resulting areas will be irregular, requiring more care.

First, I assumed the splitting cut would be as shown in Figure 5.1
Then I calculated the area of each of the resulting rooms if I split the rooms naively. After that, I calculated how many of these splitting beepers need to be shifted to make the rooms equal.

Finally, I took care of the case where where an additional beeper is needed, (same as the previous section).

Here is the code for this part:

```
    } else {
        int area1 = halfStreets * (halfAvenues - 1);
        int area2 = halfAvenues * (halfStreets - 1);
        int toDel = (area2 - area1) / 2;

     // The first half cut
        move(backwards, halfStreets-1, false);
        move(right, halfAvenues-1, true);
        moveDiagonally(left);
        move(forwards, halfAvenues-1, true);
        move(right, halfStreets, false);

     // The first additional Beeper
        if (0 == ((area1&1)^(area2&1)))
            move(0,true);

     // the second cut - part 1
        move(right, halfAvenues-1, false);
        move(right, halfStreets - 2 - toDel, true);
        moveDiagonally(left);
        move(forwards, toDel, true);
```

```
        // The second additional Beeper
           move();
           if (0 == ((area1&1)^(area2&1)))
               move(0,true);

        // The second cut - part 2
           if (toDel > 0) {
               moveDiagonally(right);
               move(forwards, toDel - 1, true);
               moveDiagonally(left);
           } else {
               move();
           }
           move(forwards, halfStreets - 2 - toDel, true);
       }
   }
```

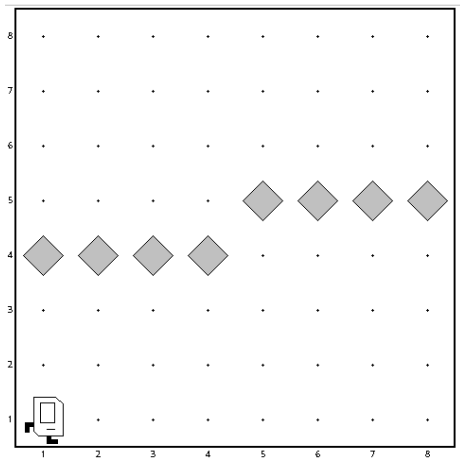

Figure 5.1: Splitting an Even dimension in half

5.2 Cost Analysis

In this section, I will present the cost analysis of the perpendicular cuts method.

Naturally, splitting the algorithm into three cases will yield three different analyses of cost, one for each method. However, one commonality among them is the required number of moves Karel needs to make, which is:

- `(streets-1) + (avenues-1)`: to find the dimensions of the world.
- `2 * (streets-1) + 2 * (avenues-1)`: to split the world.

- The number of needed S turns: case1 $\rightarrow$ 0, case2 $\rightarrow$ 1, case3 $\rightarrow$ 2

```
private int costOfPerpendicularMoves() {
    return 3*(streets-1 + avenues-1) + 1-(streets%2) + 1-avenues%2;
}
```

Note: The symmetry between the number of streets and avenues in the previous equation, results in the irrelevance of which dimension we start splitting from and makes the cost of movements the same. This also simplifies the analysis of standardization, which doesn't change the cost.

Following is the analysis for calculating the required number of beepers:

case 1: Both dimensions are odd: in this case Karel would need `streets + avenues` beepers to make the vertical and horizontal cuts; however, since these two cuts intersect at one corner we need to subtract 1 from the equation.

case 2: Dimensions with different parity: similarly in this case Karel needs `streets + avenues - 1` beepers; however, sometimes it is possible to need 2 additional beepers to make rooms of equal size (odd areas).

case 3: Both dimensions are even: this case is a little different; we need `streets + avenues - 2` this time we need -2 because the first cut isn't a straight line, but an S-turn; same as before, we might need 2 additional beepers to make rooms of equal size (odd areas).

And here is the code for this part:

```
private int costOfPerpendicularCuts() {
    if (Math.min(streets, avenues) < 2)
        return streets * avenues; // infinity

    if (oddNumOfStreets && oddNumOfAvenues) {
        return streets + avenues - 1;
    } else if (oddNumOfStreets || oddNumOfAvenues) {
        int theOddOne = (oddNumOfAvenues ? avenues : streets);
        return streets + avenues + -1
        + (theOddOne/2 % 2 == 0 ? 0 : 2);
    } else {
        return avenues + streets + -2
        + (streets/2*avenues - avenues/2 - streets/2 - 1) % 2 * 2;
    }
}
```

Chapter 6

Handling Special Cases

In this chapter, I dedicated a section for each class of cases, each section will start by describing the worlds it handles, and then present how I divides these worlds.

6.1 Case 1: 1x{1..6}.w

The section handles 6 different special worlds, namely: (1x1.w), (1x2.w), (1x3.w), (1x4.w), (1x5.w) and (1x6.w), see Figure 6.1.

Note: This is after standardization, therefore this section actually handles 12 worlds, see Section 2.2.

One observation that simplified the solution greatly, was that each case can be reduced to the one smaller than it. Here is how:

- Case (1x6.w): can be solved by putting down a beeper then moving downwards. Now the problem is reduced to solving (1x5.w).
- Case (1x5.w): can be solved by moving downwards one move. Now the problem is reduced to solving (1x4.w).
- can be solved by putting down a beeper then moving downwards. Now the problem is reduced to solving (1x3.w).
- Case (1x3.w): can be solved by moving downwards one move and then putting down one beeper.
- Cases (1x1.w) and (1x2.w): do nothing, as the world is not divisible.

And here is my code:
Note: I used the *fall through* behavior of the `switch` statement, to automatically reduce the problem to the one smaller than it.

```
    } else if (avenues == 1) {
        switch (streets) {
            case 6:
                putBeeperIfNecessary();
                move();
            case 5:
                move();
            case 4:
                putBeeperIfNecessary();
                move();
            case 3:
                move();
                putBeeperIfNecessary();
        }
    }
```

6.2 Case 2: 2x{2..6}.w

The section handles 5 different special worlds, namely: (2x2.w), (2x3.w), (2x4.w), (2x5.w) and (2x6.w), see Figure 6.2.

Note: This is after standardization, therefore this section actually handles 10 worlds, see Section 2.2.

Similar to the previous Section 6.1, I can reduce the solution for any world here to doing some steps then solving a smaller world. Smaller worlds can be solved by placing beeper diagonally, whereas larger ones requires putting beepers in the top row(s), to reduced the problem to solving a smaller world.

See the following code block:

```
    } else if (avenues == 2) {
        int idx = streets;
        Direction dir = right;
        for (; idx >= 5; --idx, dir = dir.opposite()) { // filling
            top row(s)
            move(dir, 1, true);
            move(dir.opposite(), 1, true);
        }
        for (; idx > 1; --idx, dir = dir.opposite()) { // placing
            beepers diagonally
            putBeeperIfNecessary();
            moveDiagonally(dir);
        }
        putBeeperIfNecessary();
    }
```

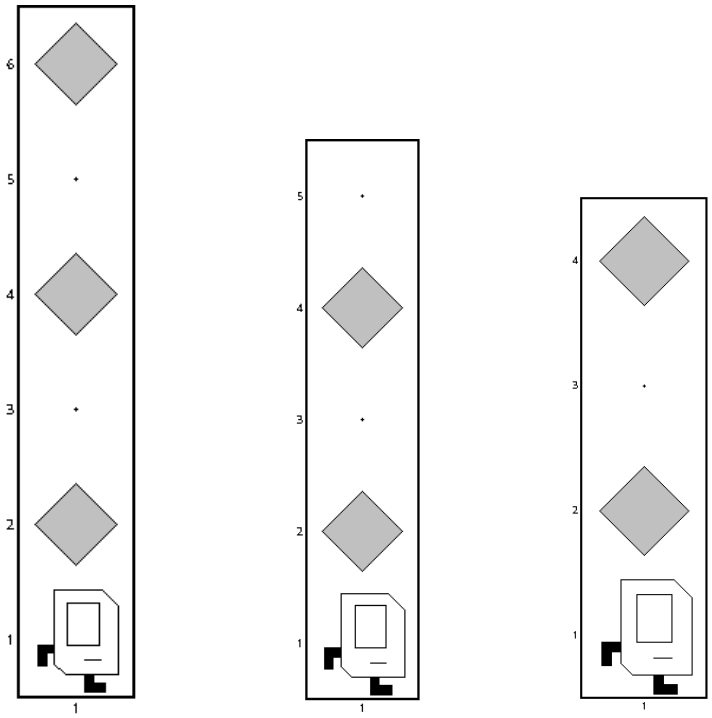

Figure 6.1: Special Worlds (1x6.w) & (1x5.w) & (1x4.w)

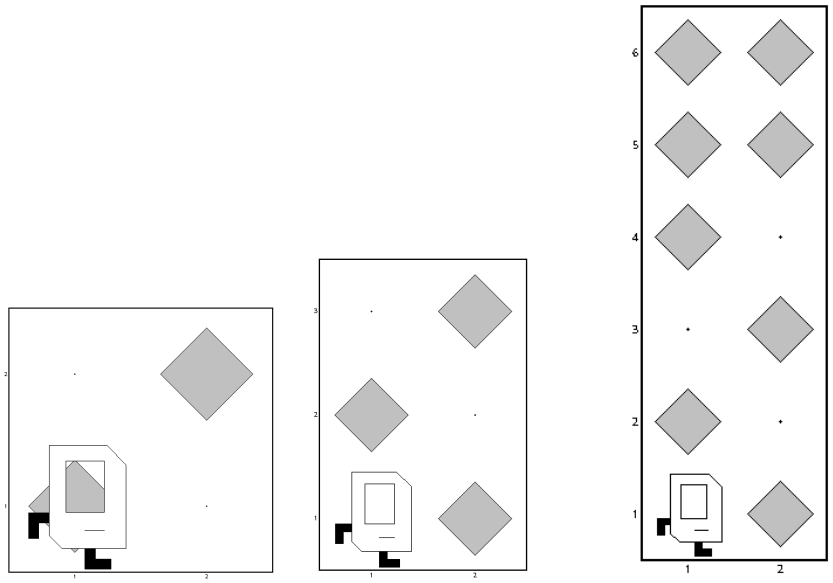

Figure 6.2: Special Worlds (2x2.w) & (2x3.w) & (2x6.w)

Assignment Instructions

Karel Assignment

Divide a given map into 4 equal chambers – Do all the analysis to handle the special cases. i.e. maps that can't be divided into 4 equal chambers should be divided into the biggest possible number of equal chambers (3, 2, or 1).

- You are allowed to use double lines of beepers if you need to, however, you need to observe that beepers use should be optimized.

Notes:

- Assume having enough number of beepers (say 1000) in Karel's bag. You can use the API to setup an initial value of the beepers.
- You can't use the classes API to solve the assignment, and you should be using only the functions given in Karel reference card. Karel is a black box that came out of the factory with certain capabilities according to it's reference card. The only exception to that is initializing Karel's bag with beepers.

Optimize your solution as follows:

- Karel should achieve his task with the lowest number of moves. Add a moves counter to your code and print it while Karel is moving.
- You should minimize the number of lines in your code to the lowest possible number of lines by writing reusable functions.
- Use the lowest possible number of beepers to achieve your task

Deliverables:

- Homework.java
- A report in pdf format that shows how you solved the problem and your optimizations.
- A video that explains your solution with the optimizations that is no longer that 8 mins. Start your video by running your code and showing how your solution works. Upload your video on Youtube as non-public listing and send the link only.
- In summary the deliverables are: Homework.java, report.pdf, and a link to Youtube video.