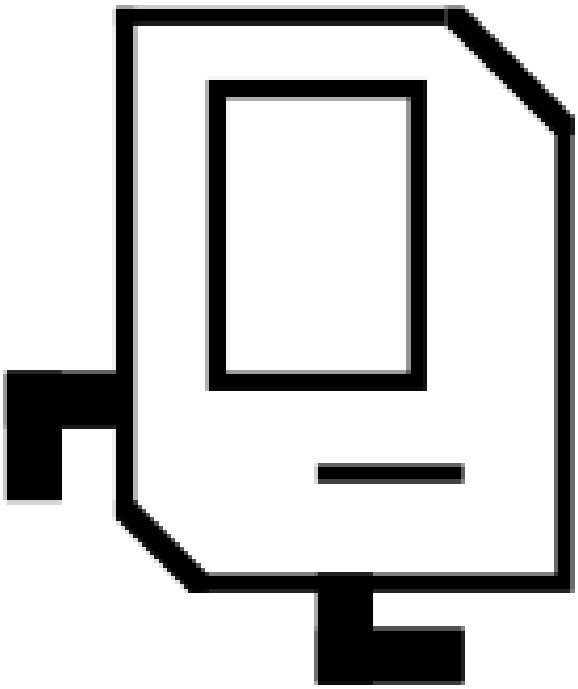


Karel Assignment

By: Osama Amireh

Contents

Introduction to Karel the Robot

Karel the Robot is a simple, yet powerful teaching environment designed to introduce students to imperative programming concepts. Developed by Richard Pattis at Stanford University, Karel provides an engaging way to learn programming fundamentals. The name “Karel” pays homage to the Czech playwright Karel Čapek, who popularized the term “robot” in his 1923 play “R.U.R.” (Rossum’s Universal Robots).

In Karel’s world, there are several methods like:

- move: Moves Karel one step forward in the current direction.
- turnLeft: Rotates Karel 90 degrees counterclockwise.
- turnRight: Rotates Karel 90 degrees clockwise.
- turnAround: Rotates Karel 180 degrees.
- putBeeper: Place a beeper at Karel’s current location.
- pickBeeper: Picks up a beeper from Karel’s current location.

By combining these commands, I have declared some new methods to simplify the solution:

- **put**: to check if there isn’t any beeper present so we can put beeper, it aims to prevent putting beepers in the same location.
- **moveWithCalculation**: to calculate the steps while moving.
- **drawLine**: to put beepers from the current location to the end of the map, using while the front is clear.
- **PutDouble**: to put double beepers in the location where Karel stands and the location next to him on the right.
- We will talk about the other methods below when needed.

Problem definition

This documentation addresses the assignment of solving Karel the robot problem. The objective is partitioning a provided map into four sections using the minimum number of movements and beepers. If the map cannot be segmented into four sections, then it should be split into three; if that's not possible, then into two sections.

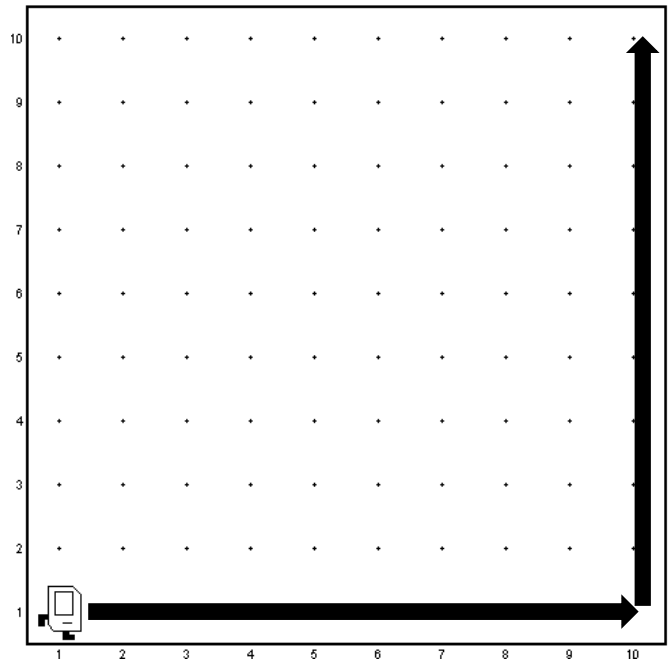
Karel always starts at the bottom left corner of the map and must return to the same corner. Double lines of beepers are necessary when the width or height of the map is even or in some special cases.

Solution definition

I analyzed the requirements and experimented with different approaches. I went through various solutions, prioritizing simplicity, efficiency, and minimizing the required steps. The following will explain each method, and the cases which encountered me:

Calculate the width and the height:

Calculate the width and the height of the map because it is the key to the solution, so I decided to create a method, called **calculate**, to calculate the length where Karel stands to the edge of the map. I use this method to calculate the width and the height of the map by declaring two private integer variables called **height** and **width** with their setters.



Small Maps:

If the map size was smaller than 2x2, then it can't be divided.

All One Cases:

If the **width** or the **height** is equal to one, then I declared a method called: **oneWithMoreThanThree** to do the following:

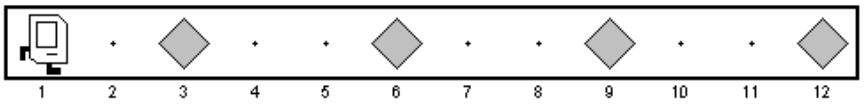
If the other one is equal to 3 or 5 then we have a static code to do this because it doesn't follow a certain rule.

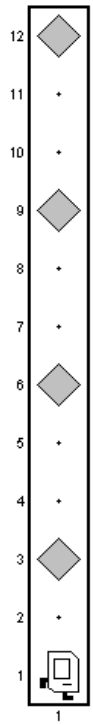
Else if it is 4 or 6 and more then there is a method called **OneOrTwoWithMoreThanFourCases**, that takes two integers: n and x as parameters, x displays if we have the case of one or the case of two, so here x should be 1, and if the width is 1 then n is height, and vice versa:

After calculating the width and the height Karel will be at the top right corner of the map, now he put a beeper, and then we have multiple cases:

1. If mod n (which is the height or the width as we said before) on 4 is 0 then:

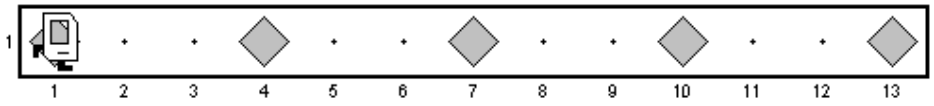
   Karel after the calculations on the right corner (on col 12), he put beeper and the count starts with 1, then he moved forward while the front was clear then if (count % (n / 4) == 0) he put a beeper, either way, the count will increase by 1. In the second step, the count now is 2 and the condition is false so he just moves till he reaches row 9 (cause the count will be 4) the condition is true so he will put a beeper, and it's just like that.

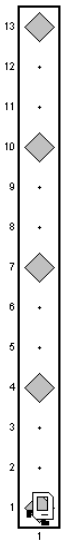




2. If mod n (which is the height or the width as we said before) on 4 is 1 then:

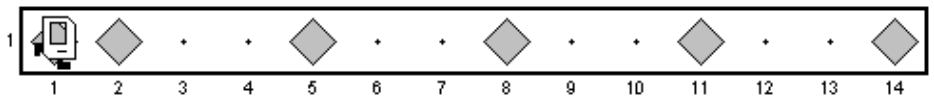
Same as the previous case but here automatically will put a beeper at the end.

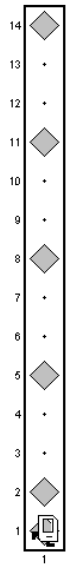




3. If mod n (which is the height or the width as we said before) on 4 is 2 then:

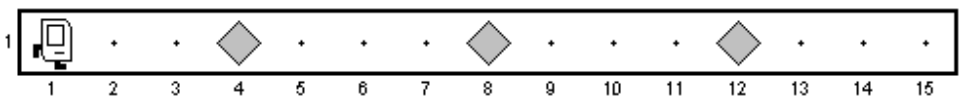
   Also, the Same as the first case but here he should put an extra beeper at the end. (remember that after the calculation Karel always starts from the end)

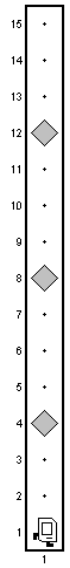




4. If mod n (which is the height or the width as we said before) on 4 is 2 then:

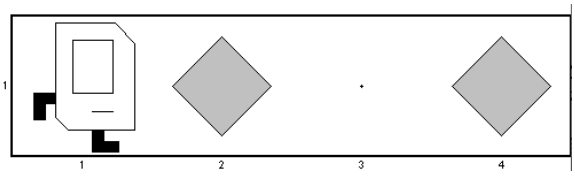
   Here we have a special treatment, first, we don't need to put the first beeper so we pick the first beeper then we should increase n by 1, to treat this as it is (count % (n / 4) == 0) but should set the count to 2 as we are the second col (from the right).

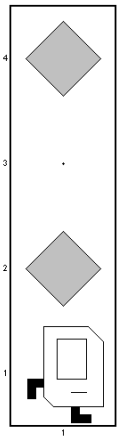




5. What if n was 4 then, we have a special case:

   Here we must put beepers not just if it is dividable by 4, but also by 2.

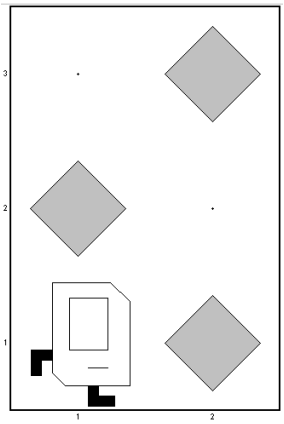


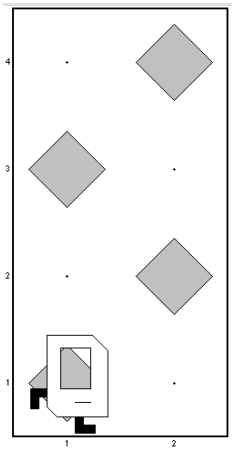


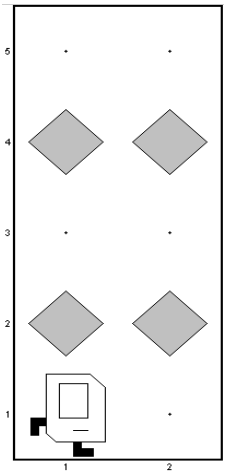
All Two Cases:

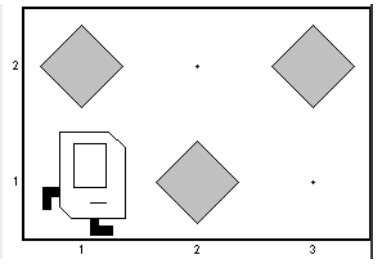
If the width or the height is equal to two, then:

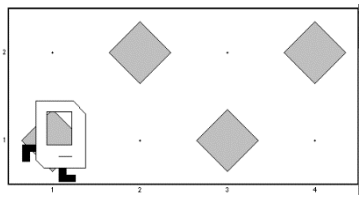
If n is 3, 4, or 5, it didn't follow the rule, so it has its code in a method called **twoWithLessThanFiveCases**:

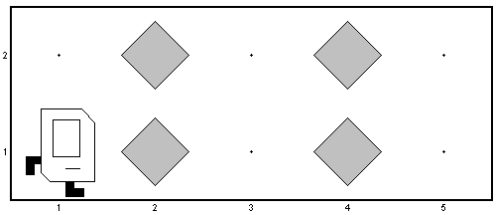






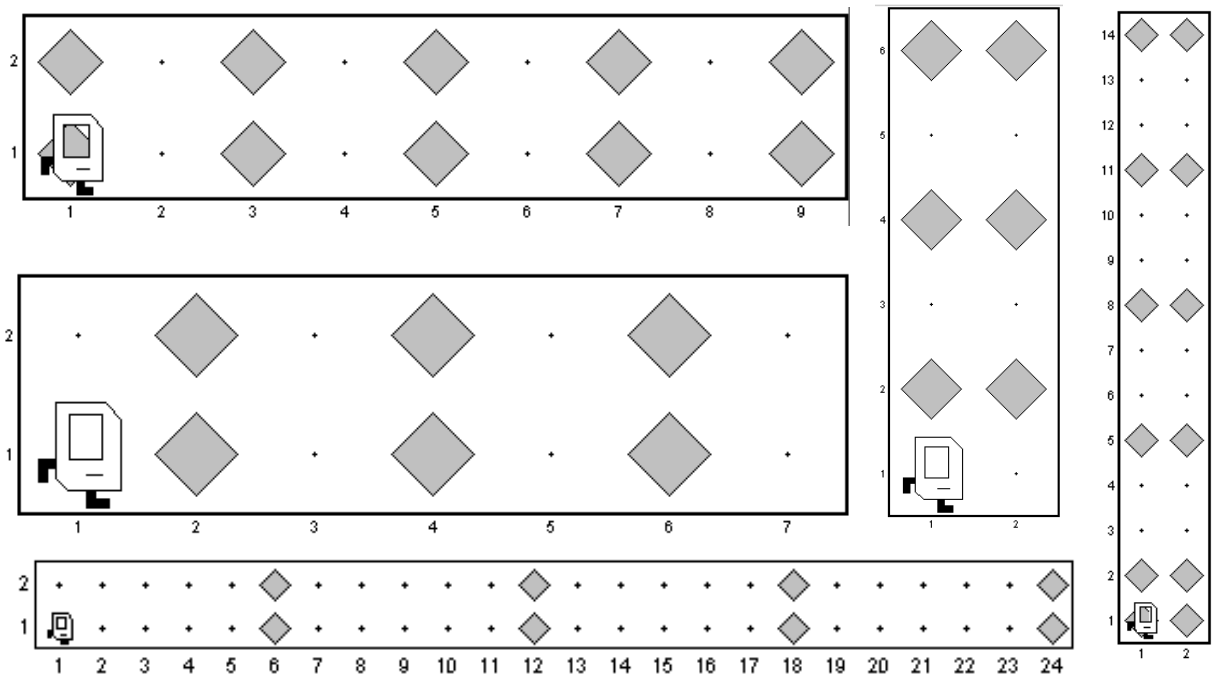






Else if n is bigger than 5 then it's like All one cases except that we should put double beepers next to each other instead of one beeper, so I declared a method called **putDouble**, so first we put 2 beepers and then follow the same role of all one cases but using **OneOrTwoWithMoreThanFourCases** with x equals to two not one, and the n as we said before, it is the height or the width.

Some examples:



Bigger Than Two by Two Cases:

I have declared 4 methods for these cases:

drawHorizontalWhenOdd, **drawHorizontalWhenEven**, **drawVerticalWhenOdd**, and **drawVerticalWhenEven** are implemented to draw horizontal and vertical lines based on whether the grid dimensions are odd or even.

Here we have 4 cases, the four cases make Karel move half of the width or the height and then turn left or right based on his position using **drawLine**, the odd cases draw one and the even cases draw double lines:

1. If the width and the height are both odd:
   Then, we should call: **drawHorizontalWhenOdd** and **drawVerticalWhenOdd**.
2. If the width and the height are both even:
   Then, we should call **drawHorizontalWhenEven** and **drawVerticalWhenEven**.
3. If the width is odd and the height is even:
   Then, we should call **drawVerticalWhenOdd** and **drawHorizontalWhenEven**.
4. If the width is even and the height is odd:
   Then, we should call **drawVerticalWhenEven** and **drawHorizontalWhenOdd**.