Karel the Robot
and Algorithms (Karel3)

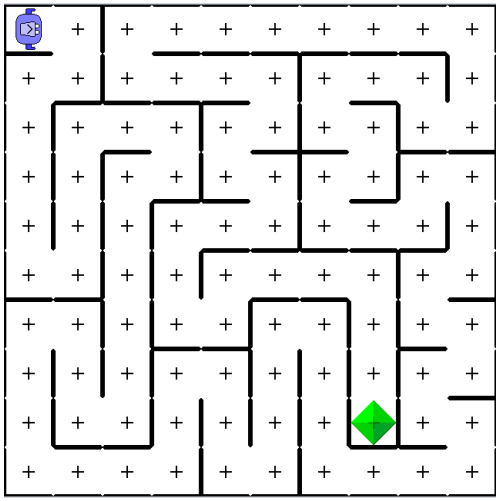
Objective: To learn algorithmic design using Karel the Robot and solve problems.

Background:
Figuring out how to solve a particular problem by computer generally requires considerable creativity. The process of designing a solution strategy is traditionally called **algorithmic design**.

The word algorithm comes from the name of a ninth-century Persian mathematician, Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî, who wrote an influential treatise on mathematics. Today, the notion of an algorithm has been formalized so that it refers to a solution strategy that meets the following conditions:

1. The strategy is expressed in a form that is clear and unambiguous.
2. The steps in the strategy can be carried out. (No wishful magical incantations!)
3. The strategy always terminates after a finite number of steps. (converges to a solution)

Solving a labyrinth:
As an example of algorithmic design, suppose that you wanted to teach Karel to navigate a labyrinth. A labyrinth is just a long hallway with lots of twists and turns. The exit is marked by a beeper. There are no crossroads or dead ends to confuse the path to the beeper.



To make this more challenging, the Karel environment has many different labyrinths preprogrammed inside its interface. You can click the button on the left of the **goal** button to see the other configurations. Your code must be general enough to solve all of them. This is much more complex than the simple problems you have handled thus far.

Put yourself in Karel's place and imagine walking the labyrinth. Think about what decisions you must make as you walk each hallway. Turn your decisions into code.

Assignment:

1. Solve the problem **`walkTheLabyrinth`** (1.4.3). After Karel completes one labyrinth, then put your code to the test. Press the button to the right of the **start** button to test all configurations. You will know you succeeded when the interface gives you a message like:

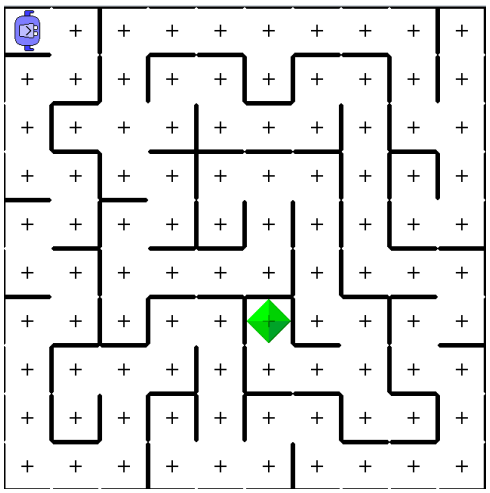
```
OK: checked 17500 random worlds
```

   Again, save your file by moving and renaming the **karel.txt** file containing your solution.

Solving a maze:

A maze is different from a labyrinth. A maze can have crossroads with paths that lead to dead ends. If you use the labyrinth algorithm on a maze you may get stuck in an infinite loop inside a dead end. Solving a maze must be approached differently than other problems.

Suppose that you wanted to teach Karel to escape from a maze. In Karel's world, a maze might look like this:



The exit to the maze is marked by a beeper. It's Karel's job is to navigate the corridors of the maze until it finds the beeper indicating the exit. The program, however, must be general enough to solve any maze, and not just the one pictured here.

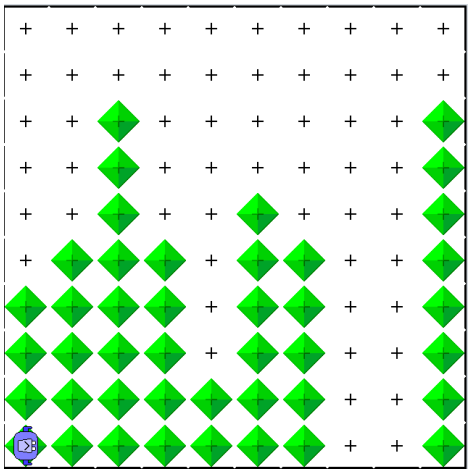
For most mazes, however, you can use a simpler strategy called the **left-hand rule**. This is where you begin by putting your left hand on the adjacent wall and then go through the maze without ever taking your hand off the wall. Another way to express this strategy is to proceed through the maze one step at a time, always taking the leftmost available path. This is called a "brute-force" method because it is effective but not efficient. It may take Karel down paths that make the journey longer, but it always guarantees a solution with no infinite loops.

Assignment:

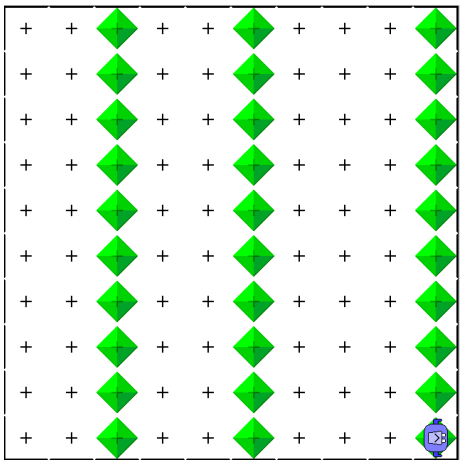
2. Solve the problem **`solveTheMaze`** (2.4.1). After Karel completes one maze, test all other configurations with the button to the right of the **start** button. Prove to yourself it works with an "OK" from the interface.

Counting without variables:
Karel has a lot of features, like iteration and conditionals, but he is missing the ability to store information in containers (like variables). If you try to declare or use variables Karel will reject them and refuse to operate. Instead, your algorithm must determine the count (state) of the operation you are trying to compute.

Suppose you are given a world like below:



Each vertical stack of beepers represents a bit in a message transmission. If there are 6 or more beepers, then it represents a "1". If there are 0 to 5 beepers, then it represents a "0". Karel will "count" the beepers in one vertical stack. If it represents a "1", then it fills in beepers to the top of the vertical. If it represents a "0", then it removes all of the beepers in the stack. The result of the above world is given below:



Assignment:

3. Solve the problem **`quantizeBits`** (2.4.2). After Karel completes one quantization, test all other configurations with the button to the right of the **start** button. Prove to yourself it works with an "OK" from the interface.