1. Example of how hard problems can get, once you involve loops
   a. Write a program that reads in an integer $N$ from the keyboard, and displays a diamond
      shape on the screen with width $2N$ and height $2N$. For example, if $N=5$, it should display
      the following figure on the screen:

      ![Diamond Shape]

      $N=5$ lines high

      $N=5$ characters wide

   b. Write a program that reads in an integer $N$ from the keyboard, and displays whether $N$ is a
      prime number or not. A number is "prime" if its only factors are 1 and itself. A "factor"
      is a number that divides another number evenly.

      Modify your program by adding a loop to find the first prime number larger than 1000.

   c. Write a program that reads in an integer $N$ from the keyboard, and displays whether $N$ is a
      "perfect number" or not. A number is "perfect" if it is equal to the sum of all of its
      factors (not including itself as a factor, but including 1 as a factor). 6 is the first perfect
      number, because its factors are 1, 2, and 3, and $1+2+3 = 6$.

      Add a loop to your program to find the next perfect number after 6.
2. **Design Strategy 1: Use recipes for basic computational problems**

   a. **The Repeat-X Algorithm**: Repeat some set of Java commands $X$ times.

      Here is the recipe, written as an algorithm:

      ```java
      i := 1
      while i < X repeat:
          execute set of Java commands, which might depend on i and X
          i := i + 1
      ```

      Suppose the set of Java commands that you had to repeat was just the single command:
      ```java
      System.out.print("*");
      ```
      and suppose that the number of times to repeat was stored in a variable called `numStars`. Write the Java code to implement this algorithm.

   b. **The Sum Algorithm**: Take some set of Java commands that computes a number. Repeat these commands $X$ times, and compute the sum of the results from each repetition.

      Here is the recipe, written as an algorithm:

      ```java
      j := 1
      sum := 0
      while j < X repeat:
          currentVal := execute set of Java commands, which might depend on j and X
          sum := sum + currentVal
          j := j + 1
      ```

      Use this algorithm to compute the sum of the squares of the integers between 1 and 10.

   c. **The Accumulate Algorithm**: This is a slightly more general version of the Sum Algorithm. Let $f$ be a function (like sum, product, min, max) that takes a set of numbers and returns a single value. We'll call $f$ the accumulator function. This algorithm takes a set of Java commands that computes a number, and repeats this set of commands $X$ times, and computes $f(\{\text{result}_1, \ldots, \text{result}_X\})$.

      Here is the algorithm:

      ```java
      j := 1
      finalResult := f({}) // the accumulator function applied to the empty set
      while j < X repeat:
          currentVal := execute set of Java commands, which might depend on j and X
          finalResult := f((finalResult, currentVal))
          j := j + 1
      ```

      Use this algorithm to read in 10 numbers from the keyboard, and find the largest one.
3. Design Strategy 2: Breaking problems down into manageable parts

a. Problem 1a (drawing the diamond) can be solved using only these parts:
   - variables and assignment statements
   - System.out.println() and System.out.print()
   - the Repeat-X algorithm

   See if you can determine how to break the problem down into these parts. Specifically, list all of the ways you need to use the Repeat-X algorithm (what is the stuff being repeated, and how many times is it repeated?).

   Once you have that figured out, then you can start writing the Java code to solve this problem.

b. Problem 1b (testing for prime numbers) can be solved using these parts:
   - System.out.println and assignment statements
   - the Accumulate Algorithm (what set of commands gets repeated? what is the accumulator function?)
   - The last line of the problem can be solved with the Repeat-X algorithm.

c. Problem 1c (testing for perfect numbers) can also be solved with our algorithms. List the parts needed to solve this problem, and then write an algorithm to solve the whole problem.