Solution to Practice Problems:
Basic Design Strategies

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

   ```java
   import java.util.Scanner;
   public class Diamond {
       public static void main(String [] args) {
           Scanner kb = new Scanner(System.in);
           int N = kb.nextInt();

           // a Repeat-X loop that goes N times, to write the first N lines
           int numSpacesBefore = N;
           int numSpacesMiddle = 0;
           for(int line=0; line<N; line++) {
               // A Repeat-X loop that goes numSpacesBefore times
               for(int space=0; space<numSpacesBefore; space++) {
                   System.out.print(" ");
               }
               System.out.print("*");
               // A Repeat-X loop that goes numSpacesMiddle times
               for(int space=0; space<numSpacesMiddle; space++) {
                   System.out.print(" ");
               }
               System.out.print("*");
               // Update how many spaces to print for next line
               numSpacesBefore--;
               numSpacesMiddle += 2;
               System.out.println(); // end of the line
           }
       }
   }
   ```

// a Repeat-X loop that goes N times, to write the second N lines
// This is basically a repeat of the loop above, except now
// the numSpacesBefore starts at 0 and goes up, and
// the numSpacesMiddle starts at 2*(N-1) and goes down.
for(int line=0; line<N; line++) {
    // A Repeat-X loop that goes numSpacesBefore times
    for(int space=0; space<numSpacesBefore; space++) {
        System.out.print(" ");
    }
    System.out.print("*");

    // A Repeat-X loop that goes numSpacesMiddle times
    for(int space=0; space<numSpacesMiddle; space++) {
        System.out.print(" ");
    }
    System.out.print("*");

    // Update how many spaces to print for next line
    numSpacesBefore++;
    numSpacesMiddle -= 2;
    System.out.println(); // end of the line
}
}

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.

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

public class PrimeTester {
    public static void main(String [] args) {
        // This loop is like a Repeat-X loop, except it keeps going until
        // foundPrime is true, instead of going a predetermined number of times.
        boolean foundPrime = false;
        int test = 1001;
        while(!foundPrime) {

            // an accumulator loop, where the accumulator function is true
            // if all the individual results are false
            boolean testIsPrime = true;
            for(int factor=2; factor<test; factor++) {
                if(test % factor == 0) {
                    testIsPrime = false;
                    break;
                }
            }

            foundPrime = testIsPrime;
            test++;
        }
    }
}
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.

```java
public class PerfectTester {
    public static void main(String[] args) {
        // This loop is like a Repeat-X loop, except it keeps going until
        // foundPerfect is true, instead of going a set number of times.
        boolean foundPerfect = false;
        int test = 7;
        while(!foundPerfect) {
            // a sum algorithm, where we only add an element
            // if it's actually a factor
            int sumFactors = 1;
            for(int factor=2; factor<test; factor++) {
                if(test % factor == 0) {
                    sumFactors += factor;
                }
            }
            foundPerfect = (test == sumFactors);
            test++;
        }
        System.out.println("found a perfect number larger than 6: "+(test-1));
    }
}
```
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:

```
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.

```java
int j = 1;
while(j<=numStars) {
    System.out.print("*");
    j += 1;
}
```

OR

```java
for(int j=0; j<numStars; j++) {
    System.out.print("*");
}
```

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.

```java
int sum = 0;
for(int j=1; j<=10; j++) {
    sum += j * j;
}
```

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.

```java
import java.util.Scanner;
public class MaxOf10 {
    public static void main(String[] args) {
        Scanner kb = new Scanner(System.in);
        double max = Double.NEGATIVE_INFINITY;
        for(int j=0; j<10; j++) {
            double val = kb.nextDouble();
            if(val > max) {
                max = val;
            }
        }
    }
}
```
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?).

   1 repeat-X loop to print the spaces before the first * on each line
      - the number of spaces (X) depends on which line you're on
        - for the first 5 lines, it starts with 5, and goes down by 1 each line
        - for the second 5 lines, it starts with 0, and goes up by 1 each line

   1 repeat-X loop to print the spaces between the *'s on each line
      - the number of spaces (X) depends on which line you're on
        - for the first 5 lines, it starts with 0, and goes up by 2 each line
        - for the second 5 lines, it starts with 2*(N-1) and goes down by 2 each line

   1 repeat-X loop to print the first N lines
      - this repeats N times, and the commands that get repeated are the two repeat-X loops above, plus commands to print the two stars

   1 repeat-X loop to print the second N lines
      - this repeats N times, and the commands that get repeated are the two repeat-X loops above, plus commands to print the two stars

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

   (See solution above)

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 accumulate algorithm to test a number N for primality will run N times. Each time through, it executes a set of Java commands that test whether the number j is a factor of N (true or false). The finalResult starts as true, and f(finalResult, currentVal) is true if both finalResult and currentVal are true, and false otherwise.

   - 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.

Parts needed:
- a set of Java commands to test whether a number \( k \) is a factor of another number \( n \)
- a sum algorithm to compute the sum of the factors of a number \( n \)
- a repeat-X algorithm to keep computing the sum of the factors of different numbers \( n \), until the sum of the factors matches the number \( n \) itself (meaning \( n \) is a perfect number)

The whole algorithm:

```java
// this outer loop is the repeat-X algorithm
test = 7 // starting with the first possible number after 6
isPerfect = false
while not isPerfect repeat:
    // here is the sum algorithm
    sum = 0
    j = 2
    while j < test repeat:
        add = 0
        // here is the algorithm to determine whether j is a factor of test
        if test modulus j is zero:
            add = j
            sum += add
    if sum == test:
        isPerfect = true
    test = test + 1

print test-1 // subtract one because the loop adds one before it stops
```