Functions Practice

For each of the functions below, write code to implement that function. You may use the `FunctionPractice.java` file in today's IntelliJ section, or you may start a new project. You should test each function in `main()`.

1. Write a function that calculates and returns the Euclidean distance between two points in the Cartesian plane. The declaration should be

   ```
   public static double distance(int x1, int y1, int x2, int y2)
   ```

   For instance, `distance(1, 2, 3, -4)` should return the Euclidean distance between (1, 2) and (3, -4), which is approximately 6.3245. Hint: use `Math.sqrt()`.

2. Write a function that sums up a range of numbers, given an upper and lower bound. Both bounds should be included in the sum. The function declaration should be

   ```
   public static int sumRange(int lower, int upper)
   ```

   Example: calling `sumRange(1, 9)` should return 45.

3. Write a function called `totalDistance` that takes an integer array of ordered pairs, and computes the total distance from the first point to the second, plus the second to the third, etc. The points array will contain (x, y) ordered pairs in this format: {x1, y1, x2, y2, x3, y3, etc}. Do not rewrite the distance formula; instead, make calls to your `distance()` function from earlier. The function declaration should be:

   ```
   public static double totalDistance(int[] points)
   ```

   Example: calling `totalDistance(new int[] {1, 2, 3, -4, -3, 0})` should add up the distance from (1, 2) to (3, -4), and then the distance from (3, -4) to (-3, 0), which in total is approximately 13.5356.

4. Recall that in a right triangle, the square of the hypotenuse *c* is equal to the sum of the squares of the other two sides *a* and *b*, such that $a^2 + b^2 = c^2$. When *a*, *b*, and *c* are all integers, these numbers are known as a *Pythagorean triple*. An example is (3, 4, 5). Write a function to find and print all Pythagorean triples where each number may be no larger than 500. Use a triply-nested `for` loop that tries all possibilities for *a*, *b*, and *c* and prints only the combinations that fit the equation. This is an example of *brute-force computation*, a technique where you just try all the possibilities until something works. For many problems, there are better algorithmic techniques than brute-force, but a brute-force algorithm is often very simple to write.

   The declaration should be:

   ```
   public static void printTriples()
   ```

   Hint: Use manual multiplication to calculate the square of a number, rather than `Math.pow()`, and do not use the square root function (`Math.sqrt`) either. The reason for this is Java cannot represent some floating-point numbers exactly in binary (just like we can't write down 1/3 exactly as a decimal), so if we compare two floating-point numbers for equality in Java, it's possible they will come back as "not equal" even if they should be equal. In other words, floating-point math sometimes produces *round-off errors*. Integer math (assuming the integers can fit in the data type you use) is always exact, so this doesn't happen.

5. Write a function to determine if a number is prime or not. Recall that an integer is prime if it has no other factors other than itself and 1. Return `true` if the number is prime, `false` if not. Hint: use a loop. The declaration should be:

```
public static boolean isPrime(int n)
```

6. Write a function called `genRandom` that works like the Python random number generator, with an upper and lower bound. The number generated should be in that range, inclusive.

```
public static int genRandom(int lower, int upper)
```

7. Write a function called `sumDigits` that takes a single integer argument and returns the sum of its digits (as an integer). For instance, the sum of the digits of the number 324 is 3+2+4=9. The function declaration should be:

```
public static int sumDigits(long num)
```

Note: the `long` data type in Java can hold bigger numbers than an `int` can. In Java, an `int` can hold positive or negative numbers up to approximately two billion (technically between -2^{31} and $2^{31}-1$) whereas a `long` can hold integers up to almost 20 digits (technically between -2^{63} and $2^{63}-1$).

Hint: Use the % operator (remainder) to extract the right-most digit from the number (take the remainder mod 10). Use division by 10 to eliminate that right-most digit entirely and continue until you run out of digits.

8. Write a function that takes an integer parameter. This function should print out the "pseudo Roman numeral" equivalent of the number. I say “pseudo” because we will simplify Roman numerals a bit by getting rid of the subtraction rules. For example, normally 9 is written as IX = 10 – 1, but your program can print VIIII.

In Roman numerals, M = 1000, D = 500, C = 100, L = 50, X = 10, V = 5, and I = 1.

The declaration line should be:

```
public static void roman(int number)
```

Hint: Use a loop that runs until the user’s number becomes equal to zero. Inside the loop, write `if` statements that test how big the number is. If the number is bigger than or equal to one of the exact Roman numerals above, print that numeral, subtract the value from the user’s number, and loop again.

Challenge: make this print out “true” Roman numerals; e.g., for 9 it should print IX, not VIIII. Try to find an algorithm for this on your own, but I have a hint if you really want it.