FUNCTIONS

The purpose of this assignment is to give you practice writing programs with Java functions (static methods). The first exercise involves real-valued functions; the second exercise focuses on integer-valued functions; the third exercise considers functions on arrays.

1. **Activation functions.** Write a program `ActivationFunction.java` to compute various activation functions that arise in neural networks. An *activation function* is a function that maps real numbers into a desired range, such as between 0 and 1 or between –1 and +1.

   - The *Heaviside step* function is given by

   $$H(x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{1}{2} & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

   - The *sigmoid* function is given by

   $$\sigma(x) = \frac{1}{1 + e^{-x}}$$

   - The *hyperbolic tangent* function is given by

   $$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

   - The *softsign* function is given by

   $$f(x) = \frac{x}{1 + |x|}$$

   - The *square nonlinearity* function is given by

   $$\text{SQNL}(x) = \begin{cases} -1 & \text{if } x \le -2 \\ x + \frac{x^2}{4} & \text{if } -2 < x < 0 \\ x - \frac{x^2}{4} & \text{if } 0 \le x < 2 \\ 1 & \text{if } x \ge 2 \end{cases}$$

   All activation functions should return NaN (not a number) if the argument is NaN.

   To do so, organize your program according to the following public API:

```
public class ActivationFunction {

    // Returns the Heaviside function of x.
    public static double heaviside(double x)

    // Returns the sigmoid function of x.
    public static double sigmoid(double x)

    // Returns the hyperbolic tangent of x.
    public static double tanh(double x)

    // Returns the softsign function of x.
    public static double softsign(double x)

    // Returns the square nonlinearity function of x.
    public static double sqnl(double x)

    // Takes a double command-line argument x and prints each activation
    // function, evaluated, in the format (and order) given below.
    public static void main(String[] args)
}
```

Here are some sample executions:

```
~/Desktop/functions> java-introcs ActivationFunction 0.0
heaviside(0.0) = 0.5
  sigmoid(0.0) = 0.5
     tanh(0.0) = 0.0
 softsign(0.0) = 0.0
     sqnl(0.0) = 0.0

~/Desktop/functions> java-introcs ActivationFunction 1.0
heaviside(1.0) = 1.0
  sigmoid(1.0) = 0.7310585786300049
     tanh(1.0) = 0.7615941559557649
 softsign(1.0) = 0.5
     sqnl(1.0) = 0.75

~/Desktop/functions> java-introcs ActivationFunction -0.5
heaviside(-0.5) = 0.0
  sigmoid(-0.5) = 0.3775406687981454
     tanh(-0.5) = -0.4621171572600098
 softsign(-0.5) = -0.3333333333333333
     sqnl(-0.5) = -0.4375
```

2. **Greatest common divisors.** Write a program `Divisors.java` to compute the greatest common divisor and related functions on integers:

   - The *greatest common divisor (gcd)* of two integers a and b is the largest positive integer that is a divisor of both a and b. For example, $gcd(1440, 408) = 24$ because 24 is a divisor of both 1440 and 408 ($1440 = 24 \cdot 60,\ 408 = 24 \cdot 17$) but no larger integer is a divisor of both. By convention, $gcd(0, 0) = 0$.

   - The *least common multiple (lcm)* of two integers a and b is the smallest positive integer that is a multiple of both a and b. For example, $lcm(56, 96) = 672$ because 672 is a multiple of both 56 and 96 ($672 = 56 \cdot 7 = 96 \cdot 12$) but no smaller positive number is a multiple of both. By convention, if either a or b is 0, then $lcm(a, b) = 0$.

   - Two integers are *relatively prime* if they share no positive common divisors (other than 1). For example, 221 and 384 are *not* relatively prime because 17 is a common divisor.

- *Euler's totient function* $\phi(n)$ is the number of integers between 1 and n that are relatively prime with n. For example, $\phi(9) = 6$ because the six numbers $1, 2, 4, 5, 7$, and 8 are relatively prime with 9. Note that if $n \leq 0$, then $\phi(n) = 0$.

To do so, organize your program according to the following public API:

```
public class Divisors {

    // Returns the greatest common divisor of a and b.
    public static int gcd(int a, int b)

    // Returns the least common multiple of a and b.
    public static int lcm(int a, int b)

    // Returns true if a and b are relatively prime; false otherwise.
    public static boolean areRelativelyPrime(int a, int b)

    // Returns the number of integers between 1 and n that are
    // relatively prime with n.
    public static int totient(int n)

    // Takes two integer command-line arguments a and b and prints
    // each function, evaluated in the format (and order) given below.
    public static void main(String[] args)
}
```

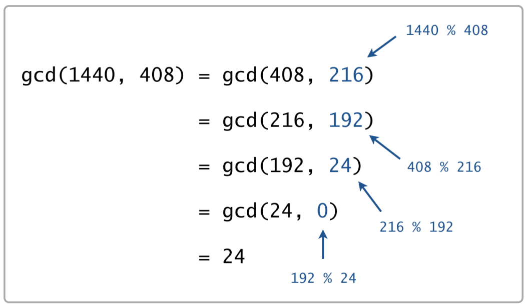
Here are some sample executions:

```
~/Desktop/functions> java-introcs Divisors 1440 408
gcd(1440, 408) = 24
lcm(1440, 408) = 24480
areRelativelyPrime(1440, 408) = false
totient(1440) = 384
totient(408) = 128

~/Desktop/functions> java-introcs Divisors 987 610
gcd(987, 610) = 1
lcm(987, 610) = 602070
areRelativelyPrime(987, 610) = true
totient(987) = 552
totient(610) = 240
```

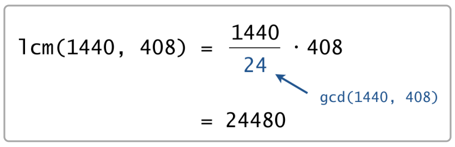
Use the following algorithms to implement the corresponding functions.

- *Greatest common divisor*. Implement an iterative version of *Euclid's algorithm*. To compute the greatest common divisor of a and b:
  - Replace (a, b) with $(|a|, |b|)$.
  - Repeatedly replace (a, b) with $(b, a \,\%\, b)$ until the second integer in the pair is zero.
  - Return the first integer in the pair as the *gcd*.



- *Least common multiple*. Use the following formula, which relates the *gcd* and *lcm* functions:

$$lcm(a, b) = \frac{|a| \cdot |b|}{gcd(a, b)}$$



To avoid integer overflow, perform the division *before* the multiplication. Recall that $lcm(0, 0) = 0$.

- *Relatively prime*. Two integers a and b are relatively prime if and only if $gcd(a, b) = 1$.
- *Euler's totient function*. Use the definition and call `areRelativelyPrime()` for each positive integer between 1 and n.

The greatest common divisor and least common multiple functions arise in a variety of applications, including reducing fractions, modular arithmetic, and cryptography. Euler's totient function plays an important role in number theory, including Euler's theorem and cyclotomic polynomials.

3. **Audio collage.** Create a library to manipulate digital audio and use that library to create an audio collage. As in lecture, we will represent sound as an array of real numbers between –1 and +1, with 44,100 samples per second. You will write a library of functions to produce audio effects by manipulating such arrays.

- *Amplify*. Crate a new sound that contains the same samples as an existing sound, but with each sample multiplied by a constant α. This increases the volume when $\alpha > 1$ and decreases it when $0 < \alpha < 1$.
- *Reverse*. Create a new sound that contains the same samples as an existing sound, but in reverse order. This can lead to unexpected and entertaining results.
- *Merge/join*. Create a new sound that combines two existing sounds by appending the second one after the first. If the two sounds have m and n samples, then the resulting sound has $m + n$ samples. This enables you to play two sounds sequentially.
- *Mix/overlay*. Create a new sound that combines two existing sounds by summing the values of the corresponding samples. If one sound is longer than the other, append 0s to the shorter sound before summing. This enables you to play two sounds simultaneously.
- *Change speed*. Create a new sound that changes the duration of an existing sound via *resampling*. If the existing sound has n samples, then the new sound has $\lfloor n / \alpha \rfloor$ samples for some constant $\alpha > 0$, with sample i of the new sound having the same amplitude as sample $\lfloor i \alpha \rfloor$ of the existing sound.

To do so, organize your program according to the following public API:

```
public class AudioCollage {

    // Returns a new array that rescales a[] by a multiplicative factor of alpha.
    public static double[] amplify(double[] a, double alpha)

    // Returns a new array that is the reverse of a[].
    public static double[] reverse(double[] a)

    // Returns a new array that is the concatenation of a[] and b[].
    public static double[] merge(double[] a, double[] b)

    // Returns a new array that is the sum of a[] and b[],
    // padding the shorter arrays with trailing 0s if necessary.
    public static double[] mix(double[] a, double[] b)

    // Returns a new array that changes the speed by the given factor.
    public static double[] changeSpeed(double[] a, double alpha)

    // Creates an audio collage and plays it on standard audio.
    // See below for the requirements.
    public static void main(String[] args)
}
```

Here is some more information about the required behavior:

- The functions must not mutate the argument array(s).
- You may assume that the array arguments are not `null` and that $\alpha > 0$ in `changeSpeed()`.

- The `main()` method must create an audio collage and play it using `StdAudio.play()`.
  - Do not use any command-line arguments.
  - The duration must be between 10 and 60 seconds (441,000 to 2,646,000 samples).
  - All samples sent to standard audio must be between –1 and +1.
  - It must use at least five differen WAV files. Several WAV files are provided: beatbox.wav, buzzer.wav, chimes.wav, cow.wav, dialup.wav, exposure.wav, harp.wav, piano.wav, scratch.wav, silence.wav, and singer.wav. You may also supply or create your own.
  - Use `StdAudio.read()` to read a WAV file and extract its samples as an array of floating-point numbers between –1 and +1.
  - It must use all of the audio effects specified in the API (*amplify*, *reverse*, *merge*, *mix*, and *change speed*). Feel free to add additional audio effects (see the FAQ for some ideas).
  - Be creative!

This problem was inspired by an audio effects programming assignment by Keith Vertanen.

Submission. Submit a `.zip` file containing `ActivationFunction.java`, `Divisors.java`, and `AudioCollage.java`. You may also submit supplementary .wav files—put them in the same directory as your .java files. (There is no need to submit the provided .wav files.) You may not call library functions except those in `java.lang`. Use only Java features that have already been introduced in the course (e.g., loops, arrays, and functions but not objects).