See the Assessment Guide for information on how to interpret this report.

ASSESSMENT SUMMARY

```
Compilation:  PASSED
API:          PASSED

SpotBugs:     PASSED
PMD:          PASSED
Checkstyle:   PASSED

Correctness:  63/64 tests passed
Memory:       No tests available for autograding.
Timing:       No tests available for autograding.

Aggregate score: 98.59%
[ Compilation: 5%, API: 5%, Style: 0%, Correctness: 90% ]
```

ASSESSMENT DETAILS

```
The following files were submitted:
----------------------------------
2.0K May  9 16:30 ActivationFunction.java
2.0K May  9 16:30 AudioCollage.java
1.5K May  9 16:30 Divisors.java


*********************************************************************************
*  COMPILING
*********************************************************************************


% javac ActivationFunction.java
*-----------------------------------------------------------

% javac Divisors.java
*-----------------------------------------------------------

% javac AudioCollage.java
*-----------------------------------------------------------


================================================================


Checking the APIs of your programs.
*-----------------------------------------------------------
ActivationFunction:

Divisors:

AudioCollage:

================================================================
```

```
*****************************************************************************
*  CHECKING STYLE AND COMMON BUG PATTERNS
*****************************************************************************

% spotbugs *.class
*-----------------------------------------------------------

============================================================

% pmd .
*-----------------------------------------------------------

============================================================

% checkstyle *.java
*-----------------------------------------------------------

% custom checkstyle checks for ActivationFunction.java
*-----------------------------------------------------------

% custom checkstyle checks for Divisors.java
*-----------------------------------------------------------

% custom checkstyle checks for AudioCollage.java
*-----------------------------------------------------------

============================================================

*****************************************************************************
*  TESTING CORRECTNESS
*****************************************************************************

Testing correctness of ActivationFunction
*-----------------------------------------------------------
Running 17 total tests.

Test 1: check output format of main() for inputs from assignment specification
  % java 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

  % java 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

  % java 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

  ==> passed

  Test 2: check correctness of main() for inputs from assignment specification
    * x = 0.0
    * x = 1.0
    * x = -0.5
  ==> passed

  Test 3: check correctness of heaviside() for given x
    * heaviside(0.0)
    * heaviside(0.5)
    * heaviside(1.0)
    * heaviside(2.0)
    * heaviside(-1.0)
    * heaviside(0.4999999999999999944488848768742172978818416595458984375)
    * heaviside(0.5000000000000000111022302462515654042363166809082031 25)
    * heaviside(0.9999999999999999888977697537484345957636833190917968 75)
    * heaviside(1.0000000000000000222044604925031308084726333618164062 5)
  ==> passed

  Test 4: check heaviside() for special values of x
    * heaviside(Double.POSITIVE_INFINITY)
    * heaviside(Double.NEGATIVE_INFINITY)
    * heaviside(Double.NaN)
    * heaviside(-0.0)
    * heaviside(Double.MIN_NORMAL)
    * heaviside(Double.MAX_VALUE)
    * heaviside(Double.MIN_VALUE)
    * heaviside(-Double.MAX_VALUE)
    * heaviside(-Double.MIN_VALUE)
  ==> passed

  Test 5: check correctness of heaviside() for random x
    * 10000 trials with x between 0.0 and 1.0
    * 10000 trials with x between -1.0 and 0.0
    * 10000 trials with x between 1.0 and 2.0
    * 10000 trials with x between 2.0 and 10.0
    * 10000 trials with x between -2.0 and 1.0
    * 10000 trials with x between -10.0 and 10.0
  ==> passed

  Test 6: check correctness of sigmoid() for given x
    * sigmoid(0.0)
    * sigmoid(0.5)
    * sigmoid(1.0)
    * sigmoid(2.0)
    * sigmoid(-1.0)
    * sigmoid(-2.0)
  ==> passed

  Test 7: check sigmoid() for special values of x
    * sigmoid(Double.POSITIVE_INFINITY)
    * sigmoid(Double.NEGATIVE_INFINITY)
    * sigmoid(Double.NaN)
    * sigmoid(-0.0)
    * sigmoid(Double.MIN_NORMAL)
    * sigmoid(Double.MAX_VALUE)
    * sigmoid(Double.MIN_VALUE)
    * sigmoid(-Double.MAX_VALUE)
    * sigmoid(-Double.MIN_VALUE)
  ==> passed

  Test 8: check correctness of sigmoid() for random x
```

```
    * 10000 trials with x between 0.0 and 1.0
    * 10000 trials with x between -1.0 and 0.0
    * 10000 trials with x between 1.0 and 2.0
    * 10000 trials with x between 2.0 and 10.0
    * 10000 trials with x between -2.0 and 1.0
    * 10000 trials with x between -10.0 and 10.0
  ==> passed

  Test 9: check correctness of tanh() for given x
    * tanh(0.0)
    * tanh(0.5)
    * tanh(1.0)
    * tanh(2.0)
    * tanh(-1.0)
    * tanh(-2.0)
  ==> passed

  Test 10: check tanh() for special values of x
    * tanh(Double.POSITIVE_INFINITY)
      - tanh() returns wrong value
      - student   tanh(x) = NaN
      - reference tanh(x) = 1.0

    * tanh(Double.NEGATIVE_INFINITY)
      - tanh() returns wrong value
      - student   tanh(x) = NaN
      - reference tanh(x) = -1.0

    * tanh(Double.NaN)
    * tanh(-0.0)
    * tanh(Double.MIN_NORMAL)
    * tanh(Double.MAX_VALUE)
      - tanh() returns wrong value
      - student   tanh(x) = NaN
      - reference tanh(x) = 1.0

    * tanh(Double.MIN_VALUE)
    * tanh(-Double.MAX_VALUE)
      - tanh() returns wrong value
      - student   tanh(x) = NaN
      - reference tanh(x) = -1.0

    * tanh(-Double.MIN_VALUE)
  ==> FAILED

  Test 11: check correctness of tanh() for random x
    * 10000 trials with x between 0.0 and 1.0
    * 10000 trials with x between -1.0 and 0.0
    * 10000 trials with x between 1.0 and 2.0
    * 10000 trials with x between 2.0 and 10.0
    * 10000 trials with x between -2.0 and 1.0
    * 10000 trials with x between -10.0 and 10.0
  ==> passed

  Test 12: check correctness of softsign() for given x
    * softsign(0.0)
    * softsign(0.5)
    * softsign(1.0)
    * softsign(2.0)
    * softsign(-1.0)
    * softsign(-2.0)
  ==> passed

  Test 13: check softsign() for special values of x
    * softsign(Double.POSITIVE_INFINITY)
```

```
    * softsign(Double.NEGATIVE_INFINITY)
    * softsign(Double.NaN)
    * softsign(-0.0)
    * softsign(Double.MIN_NORMAL)
    * softsign(Double.MAX_VALUE)
    * softsign(Double.MIN_VALUE)
    * softsign(-Double.MAX_VALUE)
    * softsign(-Double.MIN_VALUE)
  ==> passed

  Test 14: check correctness of softsign() for random x
    * 10000 trials with x between 0.0 and 1.0
    * 10000 trials with x between -1.0 and 0.0
    * 10000 trials with x between 1.0 and 2.0
    * 10000 trials with x between 2.0 and 10.0
    * 10000 trials with x between -2.0 and 1.0
    * 10000 trials with x between -10.0 and 10.0
  ==> passed

  Test 15: check correctness of sqnl() for given x
    * sqnl(0.0)
    * sqnl(0.5)
    * sqnl(1.0)
    * sqnl(2.0)
    * sqnl(-1.0)
    * sqnl(-2.0)
    * sqnl(1.9999999999999997779553950749686919152736663818359375)
    * sqnl(2.0000000000000004440892098500626161694526672363281 25)
    * sqnl(-2.0000000000000004440892098500626161694526672363281 25)
    * sqnl(-1.9999999999999997779553950749686919152736663818359375)
  ==> passed

  Test 16: check sqnl() for special values of x
    * sqnl(Double.POSITIVE_INFINITY)
    * sqnl(Double.NEGATIVE_INFINITY)
    * sqnl(Double.NaN)
    * sqnl(-0.0)
    * sqnl(Double.MIN_NORMAL)
    * sqnl(Double.MAX_VALUE)
    * sqnl(Double.MIN_VALUE)
    * sqnl(-Double.MAX_VALUE)
    * sqnl(-Double.MIN_VALUE)
  ==> passed

  Test 17: check correctness of sqnl() for random x
    * 10000 trials with x between 0.0 and 1.0
    * 10000 trials with x between -1.0 and 0.0
    * 10000 trials with x between 1.0 and 2.0
    * 10000 trials with x between 2.0 and 10.0
    * 10000 trials with x between -2.0 and 1.0
    * 10000 trials with x between -10.0 and 10.0
  ==> passed


  ActivationFunction Total: 16/17 tests passed!


  ================================================================
  Testing correctness of Divisors
  *----------------------------------------------------------
  Running 24 total tests.

  Test 1: check output format of main() for given command-line arguments
    % java Divisors 1440 408
    gcd(1440, 408) = 24
```

```
    lcm(1440, 408) = 24480
    areRelativelyPrime(1440, 408) = false
    totient(1440) = 384
    totient(408) = 128

    % java Divisors 987 610
    gcd(987, 610) = 1
    lcm(987, 610) = 602070
    areRelativelyPrime(987, 610) = true
    totient(987) = 552
    totient(610) = 240

  ==> passed

  Test 2: check that main() prints correct values for given command-line arguments
    * a = 1440, b = 408
    * a = 987, b = 610
  ==> passed

  Test 3: check correctness of gcd() for given a and b
    * gcd(1440, 408)
    * gcd(408, 1440)
    * gcd(210, 45)
    * gcd(3571, 60707)
    * gcd(196418, 317811)
    * gcd(2147483600, 1857573314)
  ==> passed

  Test 4: check correctness of gcd() with a or b negative
    * gcd(-1440, 408)
    * gcd(1440, -408)
    * gcd(-1440, -408)
    * gcd(-408, 1440)
    * gcd(408, -1440)
    * gcd(-408, -1440)
  ==> passed

  Test 5: check correctness of gcd() with |p| or |q| close to 2^31
    * gcd(-2147483647, 2147483647)
    * gcd(-2147483647, -2147483647)
    * gcd(2147483647, -2147483647)
    * gcd(2147483647, 2147483647)
    * gcd(-2147483600, 1857573314)
    * gcd(2147483600, -1857573314)
  ==> passed

  Test 6: check correctness of gcd() with a = 0 or b = 0
    * gcd(0, 1)
    * gcd(0, 2)
    * gcd(0, 5)
    * gcd(0, -10)
    * gcd(0, 123456789)
    * gcd(1, 0)
    * gcd(2, 0)
    * gcd(5, 0)
    * gcd(-10, 0)
    * gcd(123456789, 0)
    * gcd(0, 0)
  ==> passed

  Test 7: check that gcd(a, b) = gcd(b, a) for random a and b
    * 1000 trials with a and b between 1 and 10
    * 1000 trials with a and b between 1 and 1000
    * 1000 trials with a and b between 1 and 1000000
    * 1000 trials with a and b between 1 and 100000000
```

```
    * 1000 trials with a and b between -10 and 10
    * 1000 trials with a and b between -1000 and 1000
    * 1000 trials with a and b between -100000 and 100000
    * 1000 trials with a and b between -10000000 and 10000000
  ==> passed

  Test 8: check correctness of gcd() for random a and b
    * 1000 trials with a and b between 1 and 10
    * 1000 trials with a and b between 1 and 1000
    * 1000 trials with a and b between 1 and 1000000
    * 1000 trials with a and b between 1 and 100000000
    * 1000 trials with a and b between -10 and 10
    * 1000 trials with a and b between -1000 and 1000
    * 1000 trials with a and b between -100000 and 100000
    * 1000 trials with a and b between -10000000 and 10000000
  ==> passed

  Test 9: check correctness of lcm() for given a and b
    * lcm(96, 56)
    * lcm(56, 96)
    * lcm(210, 45)
    * lcm(3571, 60707)
  ==> passed

  Test 10: check correctness of lcm() for a and b that might cause overflow
    * lcm(5772000, 2652000)
    * lcm(2652000, 5772000)
    * lcm(1640957, 1653787)
    * lcm(1653787, 1640957)
    * lcm(2137049094, 15485863)
    * lcm(15485863, 2137049094)
  ==> passed

  Test 11: check correctness of lcm() with a or b negative
    * lcm(96, -56)
    * lcm(-96, 56)
    * lcm(-96, -56)
    * lcm(56, -96)
    * lcm(-56, 96)
    * lcm(-56, -96)
  ==> passed

  Test 12: check correctness of lcm() with a = 0 or b = 0
    * lcm(0, 1)
    * lcm(0, 2)
    * lcm(0, 5)
    * lcm(0, -10)
    * lcm(0, 123456789)
    * lcm(1, 0)
    * lcm(2, 0)
    * lcm(5, 0)
    * lcm(-10, 0)
    * lcm(123456789, 0)
    * lcm(0, 0)
  ==> passed

  Test 13: check that lcm(a, b) = lcm(b, a) for random a and b
    * 1000 trials with a and b between 1 and 10
    * 1000 trials with a and b between 1 and 100
    * 1000 trials with a and b between 1 and 10000
    * 1000 trials with a and b between 1 and 100000
    * 1000 trials with a and b between -10 and 10
    * 1000 trials with a and b between -100 and 100
    * 1000 trials with a and b between -1000 and 1000
    * 1000 trials with a and b between -10000 and 10000
```

```
==> passed

Test 14: check correctness of lcm() for random a and b
  * 1000 trials with a and b between 1 and 10
  * 1000 trials with a and b between 1 and 100
  * 1000 trials with a and b between 1 and 1000
  * 1000 trials with a and b between 1 and 10000
  * 1000 trials with a and b between -10 and 10
  * 1000 trials with a and b between -100 and 100
  * 1000 trials with a and b between -1000 and 1000
  * 1000 trials with a and b between -10000 and 10000
==> passed

Test 15: check correctness of areRelativelyPrime() for given a and b
  * areRelativelyPrime(1440, 408)
  * areRelativelyPrime(408, 1440)
  * areRelativelyPrime(210, 45)
  * areRelativelyPrime(3571, 60707)
  * areRelativelyPrime(196418, 317811)
  * areRelativelyPrime(2147483600, 1857573314)
==> passed

Test 16: check correctness of areRelativelyPrime() with a = 0 or b = 0
  * areRelativelyPrime(0, 1)
  * areRelativelyPrime(0, 2)
  * areRelativelyPrime(0, 5)
  * areRelativelyPrime(0, -10)
  * areRelativelyPrime(0, 123456789)
  * areRelativelyPrime(1, 0)
  * areRelativelyPrime(2, 0)
  * areRelativelyPrime(5, 0)
  * areRelativelyPrime(-10, 0)
  * areRelativelyPrime(123456789, 0)
  * areRelativelyPrime(0, 0)
==> passed

Test 17: check that areRelativelyPrime(a, b) = areRelativelyPrime(b, a) for random a and b
  * 1000 trials with a and b between 1 and 10
  * 1000 trials with a and b between 1 and 100
  * 1000 trials with a and b between 1 and 10000
  * 1000 trials with a and b between 1 and 100000
  * 1000 trials with a and b between -10 and 10
  * 1000 trials with a and b between -100 and 100
  * 1000 trials with a and b between -1000 and 1000
  * 1000 trials with a and b between -10000 and 10000
==> passed

Test 18: check that areRelativelyPrime() is consistent with gcd() for random a and b
  * 1000 trials with a and b between 0 and 10
  * 1000 trials with a and b between 0 and 1000
  * 1000 trials with a and b between 0 and 1000000
  * 1000 trials with a and b between 0 and 100000000
  * 1000 trials with a and b between -10 and 10
  * 1000 trials with a and b between -1000 and 1000
  * 1000 trials with a and b between -100000 and 100000
  * 1000 trials with a and b between -10000000 and 10000000
==> passed

Test 19: check correctness of areRelativelyPrime() for random a and b
  * 1000 trials with a and b between 1 and 10
  * 1000 trials with a and b between 1 and 1000
  * 1000 trials with a and b between 1 and 1000000
  * 1000 trials with a and b between 1 and 100000000
  * 1000 trials with a and b between -10 and 10
  * 1000 trials with a and b between -1000 and 1000
```

```
    * 1000 trials with a and b between -100000 and 100000
    * 1000 trials with a and b between -10000000 and 10000000
  ==> passed

  Test 20: check correctness of totient() for given n
    * totient(9)
    * totient(56)
    * totient(96)
    * totient(408)
    * totient(1440)
    * totient(42473)
  ==> passed

  Test 21: check correctness of totient() for small n
    * totient(0)
    * totient(1)
    * totient(2)
    * totient(3)
    * totient(-1)
    * totient(-2)
    * totient(-3)
  ==> passed

  Test 22: check that totient(n) = n-1 when n is prime
    * 1000 trials with n between 2 and 1000
    * 1000 trials with n between 2 and 10000
    * 100 trials with n between 2 and 100000
  ==> passed

  Test 23: check that totient() is consistent with areRelativelyPrime()
    * 100 trials with n between 1 and 10
    * 100 trials with n between 1 and 100
    * 100 trials with n between 1 and 1000
    * 100 trials with n between 1 and 10000
    * 10 trials with n between 1 and 100000
  ==> passed

  Test 24: check correctness of totient() for random n
    * 1000 trials with n between 1 and 10
    * 1000 trials with n between 1 and 100
    * 1000 trials with n between 1 and 1000
    * 1000 trials with n between 1 and 10000
    * 100 trials with n between 1 and 100000
    * 100 trials with n between -100000 and -1
  ==> passed


  Divisors Total: 24/24 tests passed!


  ================================================================
  Testing correctness of AudioCollage
  *----------------------------------------------------------
  Running 23 total tests.

  Test 1: check correctness of amplify() for random a[] and given alpha
    * 1000 trials, n =  2, alpha = 1.0
    * 1000 trials, n =  3, alpha = 2.0
    * 1000 trials, n =  4, alpha = 0.5
    * 1000 trials, n =  5, alpha = 1.0
    * 1000 trials, n =  6, alpha = 2.0
    * 1000 trials, n = 10, alpha = 0.5
    * 1000 trials, n = 20, alpha = 0.25
  ==> passed
```

```
Test 2: check correctness of amplify() for given WAV file and alpha
  * file = silence.wav, alpha = 1.0
  * file = silence.wav, alpha = 0.5
  * file = silence.wav, alpha = 2.0
  * file = buzzer.wav, alpha = 1.0
  * file = buzzer.wav, alpha = 0.5
  * file = buzzer.wav, alpha = 2.0
==> passed

Test 3: check correctness of reverse() for random a[]
  * 1000 trials, n = 2
  * 1000 trials, n = 3
  * 1000 trials, n = 4
  * 1000 trials, n = 5
  * 1000 trials, n = 6
  * 1000 trials, n = 10
  * 1000 trials, n = 20
==> passed

Test 4: check correctness of reverse() for given WAV file
  * file = silence.wav
  * file = buzzer.wav
  * file = cow.wav
  * file = harp.wav
  * file = scratch.wav
==> passed

Test 5: check correctness of merge() for random a[] and b[]
  * 1000 trials, m =  2, n =  2
  * 1000 trials, m =  3, n =  3
  * 1000 trials, m =  4, n =  5
  * 1000 trials, m =  5, n =  4
  * 1000 trials, m =  6, n =  8
  * 1000 trials, m = 10, n =  3
  * 1000 trials, m = 20, n = 30
==> passed

Test 6: check correctness of merge() for given WAV files
  * file1 = silence.wav, file2 = silence.wav
  * file1 =  buzzer.wav, file2 =  buzzer.wav
  * file1 =     cow.wav, file2 =     cow.wav
  * file1 =  buzzer.wav, file2 =     cow.wav
  * file1 =     cow.wav, file2 =  buzzer.wav
  * file1 =    harp.wav, file2 = scratch.wav
  * file1 = scratch.wav, file2 =    harp.wav
==> passed

Test 7: check correctness of mix() for random a[] and b[] with m = n
  * 1000 trials, m =  2, n =  2
  * 1000 trials, m =  3, n =  3
  * 1000 trials, m =  4, n =  4
  * 1000 trials, m =  5, n =  5
  * 1000 trials, m =  6, n =  6
  * 1000 trials, m = 10, n = 10
  * 1000 trials, m = 20, n = 20
==> passed

Test 8: check correctness of mix() for random a[] and b[] with m < n
  * 1000 trials, m =  2, n =  3
  * 1000 trials, m =  3, n =  4
  * 1000 trials, m =  4, n =  6
  * 1000 trials, m =  4, n =  8
  * 1000 trials, m = 10, n = 11
  * 1000 trials, m = 20, n = 30
==> passed
```

```
Test 9: check correctness of mix() for random a[] and b[] with m > n
  * 1000 trials, m =  3, n =  2
  * 1000 trials, m =  4, n =  3
  * 1000 trials, m =  6, n =  4
  * 1000 trials, m =  8, n =  4
  * 1000 trials, m = 11, n = 10
  * 1000 trials, m = 30, n = 20
==> passed

Test 10: check correctness of mix() for given WAV files
  * file1 = silence.wav, file2 = silence.wav
  * file1 =  buzzer.wav, file2 =  buzzer.wav
  * file1 =     cow.wav, file2 =     cow.wav
  * file1 =  buzzer.wav, file2 =     cow.wav
  * file1 =     cow.wav, file2 =  buzzer.wav
  * file1 =    harp.wav, file2 = scratch.wav
  * file1 = scratch.wav, file2 =    harp.wav
==> passed

Test 11: check correctness of changeSpeed() for random a[] and given alpha
         with alpha * n integral
  * 1000 trials, n =  2, alpha = 1.0
  * 1000 trials, n =  3, alpha = 2.0
  * 1000 trials, n =  4, alpha = 0.5
  * 1000 trials, n =  5, alpha = 1.0
  * 1000 trials, n =  6, alpha = 2.0
  * 1000 trials, n = 10, alpha = 0.5
  * 1000 trials, n = 20, alpha = 0.25
==> passed

Test 12: check correctness of changeSpeed() for random a[] and given alpha,
         with alpha * n fractional
  * 1000 trials, n =  2, alpha = 2.75
  * 1000 trials, n =  3, alpha = 3.5
  * 1000 trials, n =  4, alpha = 9.875
  * 1000 trials, n =  5, alpha = 0.5
  * 1000 trials, n =  6, alpha = 0.25
  * 1000 trials, n = 10, alpha = 0.375
  * 1000 trials, n = 20, alpha = 0.9375
==> passed

Test 13: check correctness of changeSpeed() for random a[] and random alpha
  * 1000 trials, n =  2
  * 1000 trials, n =  3
  * 1000 trials, n =  4
  * 1000 trials, n =  5
  * 1000 trials, n =  6
  * 1000 trials, n = 10
  * 1000 trials, n = 20
==> passed

Test 14: check correctness of changeSpeed() for given WAV file and alpha
  * file = silence.wav, alpha = 1.0
  * file = silence.wav, alpha = 0.5
  * file = silence.wav, alpha = 2.0
  * file = buzzer.wav, alpha = 1.0
  * file = buzzer.wav, alpha = 0.5
  * file = buzzer.wav, alpha = 2.0
==> passed

Test 15: check that amplify() does not mutate a[]
  * trials = 10, n =  3, alpha = 0.5
  * trials = 10, n =  5, alpha = 0.75
  * trials = 10, n =  7, alpha = 2.0
```

```
    * trials = 10, n = 10, alpha = 1.5
  ==> passed

  Test 16: check that reverse() does not mutate a[]
    * trials = 10, n =  3
    * trials = 10, n =  5
    * trials = 10, n =  7
    * trials = 10, n = 10
  ==> passed

  Test 17: check that merge() does not mutate either a[] or b[]
    * trials = 10, m =  3, n =  3
    * trials = 10, m =  4, n =  5
    * trials = 10, m =  5, n =  4
    * trials = 10, m = 10, n =  3
    * trials = 10, m = 20, n = 30
  ==> passed

  Test 18: check that mix() does not mutate either a[] or b[]
    * trials = 10, m =  3, n =  3
    * trials = 10, m =  4, n =  5
    * trials = 10, m =  5, n =  4
    * trials = 10, m = 10, n =  3
    * trials = 10, m = 20, n = 30
  ==> passed

  Test 19: check that changeSpeed() does not mutate a[]
    * trials = 10, n =  3, alpha = 1.0
    * trials = 10, n =  5, alpha = 2.0
    * trials = 10, n =  8, alpha = 0.5
    * trials = 10, n = 10, alpha = 0.75
  ==> passed

  Test 20: check output format of main()
    % java AudioCollage
    [no output]

  ==> passed

  Test 21: check that main() reads at least 5 distinct WAV files
  ==> passed

  Test 22: check that main() produces valid sound
    * sound is of correct duration
    * samples are in specified range
  ==> passed

  Test 23: check that the autograder can save the samples to a WAV file
  ==> passed


  AudioCollage Total: 23/23 tests passed!


  ================================================================
  ********************************************************************************
  *  TIMING
  ********************************************************************************
```