COS125 - Precept 10 (Functions)

1 A `Polynomial` library

Download `precept10.zip` from the precepts webpage, unzip and open the project folder. Open `Polynomial.java` and implement as many of the following functions as you can. (Splitting the work across groups is absolutely fine – you can work on a method before it's implemented, and plug it in later!) Remember to include the tests below (and others you come up with) in `main()`.

- `print(int[] a)` prints the abstract polynomial represented by the integer coefficients in array `a`.

  For example, it should print `1 * X^2 + 2 * X + 1` if `a = [1, 2, 1]`; if `a = [3, 0, 2, 0]`, then it should print `3 * X^3 + 0 * X^2 + 2 * X + 0`.

  Bonus: can you omit the coefficient `1` and zero monomials, so the the former is printed as `X^2 + 2 * X + 1` and the latter as `3 * X^3 + 2 * X`?

- `print(double[] a)` prints the abstract polynomial represented by `a`.

  For example, it should print `1.0 * X^2 + 2.0 * X + 1.0` when `a = [1.0, 2.0, 1.0]`; and `3.0 * X^3 + 0.0 * X^2 + 2.0 * X + 0.0` if `a = [3.0, 0.0, 2.0, 0.0]`.

- `linearRoot(double[] a)` should return the root of the linear polynomial given by `a`. If it is not linear (i.e., `a.length` is not 2), it should return `NaN`.

  For example, `linearRoot([1.0, 0.0])` should return `0.0`, `linearRoot([2.0, 0.0])` should return `0.5` and `linearRoot([2.0, 0.0, 0.0])` should return `NaN`.

- `derivative(double[] a)` should return the derivative of the polynomial described by `a` (as a `double[]` array). The derivative of

  $$a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$$

  is

  $$n a_n X^{n-1} + (n-1) a_{n-1} X^{n-2} + \cdots + 2 a_2 X + a_1.$$

  For example, it should return `[2.0, 2.0]` when `a = [1.0, 2.0, 1.0]`; and `[9.0, 0.0, 2.0]` if `a = [3.0, 0.0, 2.0, 0.0]`.

- `nearestRoot(double[] a, double start, double precision)` should return the approximate root found via the Newton-Raphson method with starting point x_0 = `start`. The Newton-Raphson method proceeds by setting the $(k+1)^{\text{th}}$ approximation of a root of the function f as $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ (where f' is the derivative of f) when $|f(x_k)| \geq$ `precision`. The approximate root is the first x_k that violates the inequality.

  For example, if `a = [1.0, -1.0, 0]`, `nearestRoot(a, -1.0, 1.0E-10)` should return (approximately) `0.0`; `nearestRoot(a, 2.0, 1.0E-10)` should return (approximately) `1.0`; and `nearestRoot(a, 0.1, 0.2)` should return (approximately) `0.1`.

- `double[] quadraticRoots(double[] a)` should return both roots of the quadratic polynomial described by `a`, as given by the quadratic formula:

$$\frac{-a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}{2a_0}.$$

  If it is not quadratic (i.e., `a.length` is not `3`), it should return `null`. (Bonus: handle the case where $a_1^2 - 4a_2a_0 < 0$ too.)

  For example, `quadraticRoots(a)` should return `[1.0, 1.0]` if `a = [1.0, -2.0, 1.0]`; `[-1.0, 2.0]` if `a = [1.0, -1.0, 2.0]`; and `null` if `a = [1.0, 1.0]`.

- **Bonus** `cubicRoots(double[] a)` and `quarticRoots(double[] a)` should return all roots of the cubic/quartic polynomial described by `a`. You can use the general-case cubic and quartic formulas (or any other technique).

  Notice that you may need to deal with complex numbers!

2 Modularizing `ColorContrast.java`

Copy your submission of `ColorContrast.java` into the folder and modularize your code by implementing the methods

- `boolean isAboveLuminanceThreshold(int colorLevel)`
- `double colorLuminance(double colorLevel, boolean aboveThreshold)`
- `double totalLuminance(double luminanceRed,`
  `double luminanceGreen, double luminanceBlue)`
- `double colorContrast(double luminanceText, double luminanceBackground)`

and using them in `main()`.

Then, count the number of lines of code of the new version and compare to the first!