Programming Exam: Polynomials

In this exam, your task is to implement a data structure to store, manipulate and evaluate polynomials. The focus of the exam will be on implementing a *sparse* version of this data structure: in a sparse data structure, coefficients which are equal to zero are not stored.

This leads to very efficient memory usage and operations when we are manipulating polynomials which contain many zero coefficients.

Introduction

Recall that a polynomial is a sum of terms.

For instance, for the polynomial $7.0x^2 + 2.0x^5 + 4.5x^{10}$, the terms are $7.0x^2$, $2.0x^5$ and $4.5x^{10}$. Each term is composed of a *coefficient* and an *exponent*. For the term $7.0x^2$, the coefficient is the `Double` 7.0 and the exponent is the `Integer` 2. (Note that in this exam, exponents are always non-negative.)

As such, a polynomial can alternatively be thought of as a *symbol table* which maps integers (the exponents) to doubles (the coefficients):

```
{ 2 -> 7.0, 5 -> 2.0, 10 -> 4.5 }
```

or as a *list* of pairs of a double and integer:

```
[ (7.0, 2), (2.0, 5), (4.5, 10) ].
```

Furthermore, polynomials can be added and multiplied. Here are a few examples.

Let $P_1 = 7.0x^2 + 2.0x^5 + 4.5x^{10}$ and $P_2 = 3.0 + 1.0x^5$. The addition of P_1 and P_2 is

$$P_1 + P_2 = (7.0x^2 + 2.0x^5 + 4.5x^{10}) + (3.0 + 1.0x^5) = 3.0 + 7.0x^2 + 3.0x^5 + 4.5x^{10}$$

where, when polynomials have terms of the same exponent, the coefficients of these terms are added together. For instance, here, both P_1 and P_2 have a term of exponent 5, so the corresponding coefficients 2.0 and 1.0 are added.

Likewise, we have the product of P_1 and P_2,

$$P_1 \cdot P_2 = (7.0x^2 + 2.0x^5 + 4.5x^{10}) \cdot (3.0 + 1.0x^5) = 21.0x^2 + 6.0x^5 + 7.0x^7 + 15.5x^{10} + 4.5x^{15}$$

following the standard properties of distributivity of multiplication over addition.

Below is another, longer example of polynomial addition and multiplication:

$$(2x^2 + 5x^3 + 3x^4) + (6x^1 + 7x^2 + 4x^3)$$

$$\begin{aligned} &= 2x^2 + 5x^3 + 3x^4 \\ &\quad + 6x^1 + 7x^2 + 4x^3 \\ &= 6x^1 + 9x^2 + 9x^3 + 3x^4 \end{aligned}$$

$$(2x^2 + 5x^3 + 3x^4) \cdot (6x^1 + 7x^2 + 4x^3)$$

$$\begin{aligned} &= 2x^2 \cdot (6x^1 + 7x^2 + 4x^3) \\ &\quad + 5x^3 \cdot (6x^1 + 7x^2 + 4x^3) \\ &\quad + 3x^4 \cdot (6x^1 + 7x^2 + 4x^3) \\ &= 12x^3 + 14x^4 + 8x^5 \\ &\qquad + 30x^4 + 35x^5 + 20x^6 \\ &\qquad + 18x^5 + 21x^6 + 12x^7 \\ &= 12x^3 + 44x^4 + 61x^5 + 41x^6 + 12x^7 \end{aligned}$$

In case it may be helpful, these operations are also defined a bit more formally in the *Formal Description of Operations* appendix.

Finally, polynomials can also be evaluated: to this end, every occurrence of the formal variable x is replaced by an actual value. For instance, to evaluate P_1 at $x = 2.0$, we replace every occurrence of x in P_1 by 2.0 and then we compute the result:

$$P_1(2.0) = 7.0 \cdot (2.0)^2 + 2.0 \cdot (2.0)^5 + 4.5 \cdot (2.0)^{10} = 4700.0$$

API and Specification

The API of the class you must implement is provided below:

`public class`	`Polynomial`	immutable polynomial class
	`Polynomial(double[] coeffs)`	create a polynomial from an array `coeffs` in which `coeffs[i]` is coefficient of x^i
`double`	`coeff(int expo)`	return the value of the coefficient of the term with exponent `expo` (and `0.0` if such a coefficient doesn't exist)
`Polynomial`	`add(Polynomial that)`	create a new immutable polynomial resulting from the addition of the instance polynomial to `that`
`Polynomial`	`multiply(Polynomial that)`	create a new immutable polynomial resulting from the product of the instance polynomial and `that`
`double`	`evaluate(double x)`	evaluate the polynomial at some value `x`
`String`	`toString()`	return a string representation of the polynomial *(provided)*

In addition, the `Polynomial` class contains a *public* static constant `MAX_EXPONENT` which indicates the largest possible exponent of any term contained by a `Polynomial` object.

String Format

For convenience, we provide a static class called `PolynomialTools` which already contains a method `toString(Polynomial obj)` able to nicely format a `Polynomial` object using only its `coeff(int expo)` method and the constant `MAX_EXPONENT`. You need only make sure that you compile the `PolynomialTools` class separately before running your program.

The pretty printer represents polynomials as a sequence of non-zero terms, concatenated by `" + "` and sorted from lowest to highest exponent—with the final requirement that no two terms have the same exponent and that `x^0` is omitted. The special case of the zero polynomial is displayed as `0.0`.

As examples, `3.0*x^1 + 5.0*x^10 + 14.0*x^15` and `4.5 + 1.0*x^10` are string representations of two polynomials.

The goal of the pretty printer is to help you print some information about your class which might be helpful for both tests (such as those provided in the `main` method) and debugging.

Exceptions

You should throw a `RuntimeException` in three situations:

- in the constructor `Polynomial(double[] cs)` if the length of `cs` is equal or larger to `MAX_EXPONENT`;
- in the method `coeff` when `expo` is negative, or when it is equal or larger than `MAX_EXPONENT`;
- finally, in the method `multiply` if ever the exponent of a term, while computing the the product of two polynomials, is equal or larger than `MAX_EXPONENT`.

Submission Requirements

You must submit a file `Polynomial.java` which implements the constructor and the methods `add`, `multiply` and `evaluate`, as well as the helper method `coeff`, from the template code.

Your code will be evaluated based on both correctness and style. Furthermore, it is much more important to **submit code that compiles** rather than unfinished code that does not.

You must also submit a file `readme.txt` that contains the order of growth of the run time of your `add` and `multiply` methods.

For full credit, the *run time* of your methods `add` and `multiply` must depend only on the number of non-zero coefficients of the `Polynomial` object. For example, if P_a and P_b are two `Polynomials` each with N non-zero coefficients, then $N^2 \log N$ is an acceptable run time for the multiplication of P_a and P_b (note that it is one out of many acceptable run times).

There are no run time requirements for methods `coeff` and `evaluate` (besides returning the correct result!).

Possible Progress Steps

Note that a `Polynomial` may, at most, have only a unique coefficient associated to a given exponent.

For this reason, one solution (out of several posibilities) involves using a symbol table, such as `ST.java`, with `Integer` exponents as keys and `Double` coefficients as values. In this case, it may be useful to note that `ST` iterates over keys in sorted order.

An instance variable of type `ST<Integer, Double>` has been defined for your convenience. Your next progress steps may be:

- Define the constructor, `coeff` and `add`, then **submit**.
- Test your data structure to make sure it compiles and that `add` works properly. (If the tests all fail, this could also mean that your `coeff` method does not work properly: if so you may have to test it manually.)
- Implement `multiply`, test and **submit**.
- Implement `evaluate`, test and **submit**. Remember that $7.8^{5.0}$ can be computed in Java with `Math.pow(7.8, 5.0)`.
- Complete `readme.txt` and **submit**.

Do not forget to also compile `PolynomialTools.java` before running your class, we suggest you use the command line in the folder in which you are working:

```
javac-algs4 *.java && java-algs4 -ea Polynomial
```

Appendix: Formal Description of Operations

The examples provided in the main description of the exam, combined with your existing knowledge of mathematical operations, should be sufficient to understand how the operations on polynomials work. But if needed, below is a more formal description of both addition and multiplication.

More formally, a polynomial P is the sum of (possibly zero) terms M_i, which themselves are the product of a *coefficient* c_i and the variable x raised to the i-th power — and i is called the *exponent* of M_i.

Let P_a be a polynomial with coefficients a_i and P_b be a polynomial with coefficients b_i.

- The polynomial $P_c = P_a + P_b$ which results from adding P_a and P_b, has coefficients c_i defined by:

$$c_i := a_i + b_i.$$

- The polynomial $P_d = P_a \cdot P_b$ which results from multiplying P_a and P_b, has coefficients d_i defined by the following sum, with k iterating from 0 to i:

$$d_i := \sum_{k=0}^{i} a_k \cdot b_{i-k}.$$

$$P_j = 2x^2 + 5x^3 + 3x^4 \cdot P_k = 6x^1 + 7x^2 + 4x^3$$

$$\begin{array}{lllll}
i = 0 & & & & \\
i = 1 & & & & \\
i = 2 & & & & \\
i = 3 & 2x^2 \cdot 6x^1 & & & = 12x^3 \\
i = 4 & 2x^2 \cdot 7x^2 & + \; 5x^3 \cdot 6x^1 & & = 44x^4 \\
i = 5 & 2x^2 \cdot 4x^3 & + \; 5x^3 \cdot 7x^2 & + \; 3x^4 \cdot 6x^1 & = 61x^5 \\
i = 6 & 5x^3 \cdot 4x^3 & + \; 3x^4 \cdot 7x^2 & & = 41x^6 \\
i = 7 & 3x^4 \cdot 4x^3 & & & = 12x^7 \\
\hline
 & 12x^3 + 44x^4 + 61x^5 + 41x^6 + 12x^7 & & &
\end{array}$$

Figure 1: Example illustrating the process of multiplying two polynomials: notice how the coefficient for x^5 in the final product $P_j \cdot P_k$ is the sum of the coefficient of all pair of terms, one of which is from P_j and the other of which is from P_k, such that their exponents sum to 5. Indeed, $61 = 2 \cdot 4 + 5 \cdot 7 + 3 \cdot 6 = 8 + 35 + 18 = 61$.