CSCI 271 Algorithm Analysis & Data Structures
Spring 2026 — Assignment II

Due by: Friday Feb. 6, 2026 11:59 AM

PLESE DO NOT include your WID in any submission

The Problem

The idea behind this assignment is to design a data structure that offers exact arithmetic calculations. Due to physical memory limits, computers struggle with precision when performing a division of two numbers; for example, the value on the fraction $\frac{1}{3}$ is reduced to $0.\dot{3}$ which is limited by the size of memory available to represent extended number precision. A possible solution is to provide a data type that implements the "Fraction" data structure so that we keep the values in question as fractions. The operations defined on the Fraction data type should offer an implementation of arithmetic operations needed to manipulate them. To illustrate the concept, consider the following evaluation of the fractions:

$$\frac{16}{\frac{3}{5}+7}\times\frac{6}{7}=\frac{16}{\frac{3}{5}+\frac{35}{5}}\times\frac{6}{7}=\frac{16\times5}{38}\times\frac{6}{7}=\frac{16\times5\times6}{38\times7}=\frac{8\times5\times6}{19\times7}=\frac{240}{133}$$

Basically, we plan to avoid float number division by keeping the values as fractions, which means that we need several operations to implement the data type and implement the operations needed to perform the desired arithmetic manipulations. To help you understand how to use the class **`Fraction`**, consider the operations above and the corresponding sample code below:

```
Fraction a = new Fraction(16);
Fraction b = new Fraction(3,5).add(new fraction(7));
Fraction c = new Fraction(6,7);
Fraction results = c.multiply(a.divide(b));
System.out.println(results)       // this calls toString() to print "240/133"
```

The output would be:

```
240/133
```

This is exactly what each of the tasks below will do step-by-step.

Task 1: [20 marks]

Fractions should be created in a reduced, standard form, which means that the denominator is greater than 0 and both the numerator and denominator do not have a common factors as follows:

```
new Fraction( 6, -24)  // creates a fraction of numerator = -1 and denominator = 4
new Fraction( 0, 8 )   // creates a fraction of numerator = 0, denominator = 1 (not 8)
```

The constructor method needs to find the greatest common divisor (GCD) and divide both numerator and denominator by it. Your task is to implement the class **`Fraction`** and the constructor methods needed as follows:

```
Fraction(long a, long b) // creates a new fraction of a/b
Fraction(long a)         // creates a new fraction of a/1
```

Task 2: [10 marks]

Implement the method **`toString()`** which returns a String value of the fraction such as "1/3" with no spaces. If the denominator = 1, the method returns the numerator, i.,e., it returns "5" instead of "5/1". The method also returns the appropriate string value for "**`Infinity`**", "**`-Infinity`**", or "`NaN`" for ∞, $-\infty$, and *NaN* respectively. It should work as follows:

```
fr = new Fraction(8, -6)  // fr.toString() is "-4/3"
fr = new Fraction(23, 0)  // fr.toString() is "Infinity"
fr = new Fraction(-6, 0)  // fr.toString() is "-Infinity"
fr = new Fraction(7, 1)   // fr.toString() is "7" (not "7/1")
fr = new Fraction(0, 0)   // fr.toString() is "NaN"
```

Task 3: [60 marks — 10 for each method]

Implement methods to manipulate instances of **`Fraction`** class as follows:

`Fraction divide(Fraction f)` // returns a new fraction of this fraction divided by f
`Fraction multiply(Fraction f)` // returns a new fraction of this fraction multiplied by f
`Fraction subtract(Fraction f)` // returns a new fraction of this fraction minus f.
`Fraction add(Fraction f)` //returns a new fraction of this fraction plus f

`Fraction negate( )` // returns a new fraction that is the *negative* of this Fraction.
// negate of **`Infinity`** is **`-Infinity`**. negate of **`NaN`** is **`NaN`**

`Fraction pow(int n)` // returns a new fraction of this fraction raised to the n power.
// n may be zero or negative.

Note: none of the above methods modify the value of their argument f.

Task 4: [10 marks]

Write an application program function **`main()`** which uses your implementation for the class fraction, creates instance objects of it, and uses them to test and demonstrate that it works for all the conditions specified above. You need to show that your implementation works not only for the above examples but also for all conditions listed above. Your choices, organisation and test cases will be evaluated accordingly. Please provide sufficient comments in your source code to

explain the code that examines and tests each condition. You need to list the conditions and develop a test case for each one and indicate it clearly in your source code.

Helpful Information & Hints:

- Make sure constructors methods initialise new fractions in normalised, reduced forms.
- The fraction a/b is normalised when *b >= 0* (denominator is not negative) and when *a* and *b* are relatively prime (they do not have common factors). For this, use the GCD algorithm below to divide by their common factor).
- Wikipedia has a good explanation of the Euclidean algorithm for GCD, read it!
- To reduce a fraction into a normalised form, a method needs to divide both, the numerator and denominator, by the greatest common divisor (GCD). The GCD of two values should always be > 0, even if both values are zero. To help you, use Euclid's algorithm for GCD as the pseudocode below:

```
gcd( a, b ):
      if ( a < 0 ) then a = -a // to avoid sign problems
      while b != 0:
            remainder = a % b
            a = b
            b = remainder
      if a == 0
            a = 1
return a
```

- Develop your code and test it incrementally. First write the code for tasks 1 & 2 then test them. After that, add the code for methods in task 3, one by one testing each of them incrementally.
- Use a "truth table" to check if they return the correct result for the correct numbers.
- No special code or "if" tests are needed for most operations. You should not need to write a lot of "`if ( denominator == 0 ) ...`" or other logical expressions.
- After the **`add`** and **`subtract`** methods are completed correctly, then implement the remaining methods like **`multiply`** and **`divide`**.
- **What to Submit**:
    1. Upload your **source code** to your 271 assignment repository on GitHub and submit a link to it on blackboard.
    2. Capture **the output of your test program** into a text file (using the "script" command you learned in csci210) to show that your program works correctly! **Submit this output file on blackboard. Screen shots are not acceptable!**

Evaluation:

- Marks are allocated for each of the methods
- For each method, the **correctness** of the code will gain 80% of the marks allocated to it. For instance, correct code for the method **`add()`** will earn 8 out of the allocated 10 marks.
- Clarity of **comments and description** of methods and code segments will be graded at 10%
- **Test cases in the main program** will be graded with another 10% — please document these clearly. For each test case, please explain the condition it tests and how it does so briefly.