CISS245: Advanced Programming
Assignment 7

Name: _____

Objectives

- Write constructors.
- Use default values for parameters.
- Use initializer list for constructor.
- Write get and set methods.
- Overloaded operators.
- Pass objects to methods by reference.
- Pass objects to methods by constant reference.
- Write constant methods.
- Use `this` pointer to return a copy of an object.
- Write functions with object parameters.

You are given the following (possibly incomplete files):

- `Rational.h`
- `Rational.cpp`
- `main.cpp` (the test code)

Important Warning: Again, the files are meant to be skeleton file and might not be complete and might have deliberate missing details or even errors.

Create directory `ciss245/a/a07/a07q01`. Keep all your files in this directory.

If you're doing a copy-and-paste of the given code, note that some character might be changed by PDF to other characters. In particular the - character might actually not be the dash character. Looking at the compiler error message will help you find these minor annoying issues so that you can correct them.

Study the given test code. Add tests if necessary to test all methods and functions. Such low level function/method tests are called **unit tests**.

Observe the following very carefully:

- All methods must be constant whenever possible.
- All parameters which are objects (or struct variables) must be pass by reference or pass by constant reference as much as possible.
- Reuse code as much as possible. For instance `operator!=()` should use `operator==()`.

Let me know ASAP if you see a typo.

Q1. This assignment builds a complete library for working with fractions. Refer to your work on `Fraction` struct. We'll be using a class (and not a struct). Also, instead of call the class `Fraction`, I will call it `Rational`. In Computer Science and Math, a rational number is the same as a fraction – they mean the same thing.

You will need to write three files:

- `main.cpp`: This program tests the `Rational` class and it's supporting functions and operators.
- `Rational.h`: This is the header function containing the `Rational` class.
- `Rational.cpp`: This file contains the definitions for the methods in `Rational.h`.

Read the following carefully before diving into coding.

The following is `Rational.h`:

```
/*****************************************************************************
 File  : Rational.h
 Author: Y. Liow

 Rational class
 [WRITE A COMPLETE DOCUMENTATION FOR THIS LIBRARY]
 *****************************************************************************/

#include <iostream>

#ifndef RATIONAL_H
#define RATIONAL_H

int GCD(int, int);
int sign(int);

class Rational
{
public:
    Rational(int a = 0, int b = 1) : n(a), d(b) {}
    Rational(const Rational & r) : n(r.n), d(r.d) {}
    int get_n() const { return n; }
    int get_d() const { return d; }

    void reduce();

    bool operator==(const Rational &) const;
    bool operator!=(const Rational &) const;
    bool operator> (const Rational &) const;
```

```
    bool operator>=(const Rational &) const;
    bool operator< (const Rational &) const;
    bool operator<=(const Rational &) const;

    Rational & operator+=(const Rational &);
    Rational & operator-=(const Rational &);
    Rational & operator*=(const Rational &);
    Rational & operator/=(const Rational &);

    Rational operator+() const;
    Rational operator-() const;

    Rational operator+(const Rational &) const;
    Rational operator-(const Rational &) const;
    Rational operator*(const Rational &) const;
    Rational operator/(const Rational &) const;

    Rational abs() const;
    Rational pow(int) const;

    int get_int() const;
    double get_double() const;

    //operator int() const;     // You need not implement this.
                                // SEE COMMENTS BELOW
    //operator double() const;  // You need not implement this.
                                // SEE COMMENTS BELOW

private:
    int n; // numerator
    int d; // denominator
};

std::ostream & operator<<(std::ostream &, const Rational &);
std::istream & operator>>(std::istream &, Rational &);

void reduce(Rational &);
Rational abs(const Rational &);
Rational pow(const Rational &, int);

// Operations with int on the left
bool operator==(int, const Rational &);
bool operator!=(int, const Rational &);
bool operator> (int, const Rational &);
bool operator>=(int, const Rational &);
```

```
bool operator< (int, const Rational &);
bool operator<=(int, const Rational &);

int & operator+=(int &, const Rational &);
int & operator-=(int &, const Rational &);
int & operator*=(int &, const Rational &);
int & operator/=(int &, const Rational &);

Rational operator+(int, const Rational &);
Rational operator-(int, const Rational &);
Rational operator*(int, const Rational &);
Rational operator/(int, const Rational &);

// Operations with double on the left
// (This section is OPTIONAL. Once you implement the corresponding
// functions for int, should should be able to do it very quickly dor
// double.
/*
bool operator==(double, const Rational &);
bool operator!=(double, const Rational &);
bool operator> (double, const Rational &);
bool operator>=(double, const Rational &);
bool operator< (double, const Rational &);
bool operator<=(double, const Rational &);

double & operator+=(double &, const Rational &);
double & operator-=(double &, const Rational &);
double & operator*=(double &, const Rational &);
double & operator/=(double &, const Rational &);

double operator+(double, const Rational &);
double operator-(double, const Rational &);
double operator*(double, const Rational &);
double operator/(double, const Rational &);
*/

#endif
```

The following is `Rational.cpp`:

```
#include <iostream>
#include "Rational.h"

std::ostream & operator<<(std::ostream & cout, const Rational & r)
```

```
{
    // TODO
    return cout;
}


std::istream & operator>>(std::istream & cin, Rational & r)
{
    // TODO
    return cin;
}
```

(Note that a constructor, copy constructor, `get_n()`, and `set_n()` have already been completed for you in `Rational.h`. The rest of the methods/functions should be implemented in `Rational.cpp`)

For `main.cpp`, see the section on Test Code below.

You must implement all the functions/operators defined in the header file. The implementation should be kept in `Rational.cpp`. You must also include test cases for all functions/operators/methods in your `main()`.

See notes below.

Advice.

Read the whole document carefully before diving into the code. As you read, don't worry about the whole program and don't worry about the details in the sections on notes. After reading the whole pdf, look at the section on Test Code again and work on one function/method at a time. This means building and testing one method/function at a time. You can't run the given skeleton code since some of the methods/functions are not completed yet. So when you create the source files from the skeleton code, you want to comment out most of the code first. After that you want to slowly uncomment and add more test code in `main.cpp` as you implement one method/function as a time in the `Rational` class.

Again develop and test in tiny steps – *one* method/function at a time.

1. Read the whole pdf quickly and carefully. Understand as much as you can. Don't worry if you don't remember everything you read because you can re-read it again later when you need to.
2. Create `main.cpp`, `Rational.h`, and `Rational.cpp` from the provided skeleton code.
3. Comment out the methods/functions which are not completed yet in `main.cpp`, `Rational.h`, and `Rational.cpp`.
4. Look at the section on Test Code and implement one method/function at at time, uncommenting and adding test code as you go along. Re-read the notes in this pdf when you need to.

The `Rational` class

It should be clear what the methods and the functions should compute. The following is a short explanation. Let me know if you have questions.

The class must support the following:

1. The constructor allows calls of the form `Rational(a,b)` or `Rational(a)` or `Rational()`. For instance `Rational(5)` will construct a `Rational` object that models `5/1`. `Rational()` will construct a `Rational` object that models `0/1`.
2. Arithmetic operators `+=`, `-=`, `*=`, `+`, `-`, `*`. These operators do the obvious things with one caveat. The assignment operators besides modifying the object invoking the operator will also return a copy of the object. So for instance:

```
Rational a(1, 2);  // a models 1/2
Rational b(1, 4);  // b models 1/4
Rational c;        // c models 0/1
c = (a += b);      // a becomes 3/4 and c becomes 3/4
```

   This means that the prototype for `+=` in the `Rational` class can be

```
Rational operator+=(const Rational &);
```

   but it wouldn't work for

```
Rational a(1, 2);  // a models 1/2
Rational b(1, 4);  // b models 1/4
(a += b) += b;     // a should model 1/2 + 1/4 + 1/4 = 1/1
```

   So `+=` should return a reference to the object invoking the method:
   `Rational::operator+=` should look like this:

```
Rational & Rational::operator+=(const Rational & b)
{
    // modify *this;
    return *this;
}
```

   Note that you must implement both the binary `{` and the unary `{` as well as the binary `+` and the unary `+`. In other words you can do `a - b`, `-b`, `a + b`, `+a` if `a` and `b` are `Rational` objects. Once you are done with the above, then `-=`, `-`, etc. are similar and will be very easy.
3. Boolean operators `==`, `!=`, `<`, `<=`, `>`, `>=`. Note that `a/b` is the same as `c/d` exactly when `a*d` is `b*c`. Your code should be minimal. For instance `!=` should be in terms of `==`, i.e., `operator!=` should call `operator==`. If `a`, `b`, `c`, `d` are all positive integers, then `a/b` is less then (i.e., `<`) `c/d` exactly when `a*d` is less than `b*c`. Note that `<=` must be implemented in terms of `<` and `==`. Etc.
4. `reduce()`: This will reduce the rational so that it's in the simplest form. For

instance if the fraction modeled by your object is `2/4` and the object calls `reduce()`, then the fraction modeled by your object becomes 1/2. Note that if the object modeled is `1/-2`, on calling `reduce()`, it becomes `-1/2`. In other words, not only does calling `reduce()` will remove common divisors from the numerator and denominator, it must make sure that the denominator becomes positive.

5. If `r` is a `Rational` object, then `r.get_d()` returns the denominator of `r` (as an integer).
6. If `r` is a `Rational` object, then `r.get_n()` returns the numerator of `r` (as an integer).
7. The method `get_int()` will return the integer part of the fraction. In other words if `Rational` object `r` models `5/3`, then `get_int(r)` returns integer 1. Note that `get_int()` does not round up. [The `int()` conversion operator does the same thing but is commented out. You need not implement it. See comments below.]
8. The method `get_double()` will return the double of the fraction. In other words if `Rational` object `r` models `5/3`, then `r.get_double()` return the double `5.0/3.0`. [The `double()` conversion operator is commented out. You need not implement it. See comments below.]
9. `operator<<`. If your object models `2/4`, then `operator<<` prints `"2/4"`.(without the double quotes). If your object models `2/1`, the print method prints `"2"` (note: there is no 1). If the object modeled is `2/-4`, then `operator<<` prints `"-2/4"`. If the object modeled is `0/4`, then `operator<<` prints `0`. If the object modeled is `0/-4`, then `operator<<` prints 0.
10. `operator>>`. If your object is `r` then executing `std::cin >> r` and you enter `"2 4"` (without double quotes) and press the enter key, then `r` models `2/4`.
11. The `abs()` method and the `abs()` function returns the absolute value of the `Rational`. Note that there is an `abs()` method and an `abs()` function.

```
Rational r(-2, 3);
Rational x = r.abs(); // calls the method
Rational y = abs(r);  // calls the (non-method) function
```

The `abs()` method performs the computation. To avoid code duplication, the `abs()` function calls the `abs()` method. See notes below.

12. The `pow()` method and the `pow()` function returns the power of the `Rational` for the given integer exponent. Note that there is an `pow()` method and a `pow()` function. The `pow()` method performs the computation. To avoid code duplication, `pow()` function calls the `pow()` method. See notes below.
Write several tests for every method/function to ensure that your code is working correctly.
More details are below.

Notes on `reduce()`

The `reduce()` method will reduce your fraction (in the usual sense of arithmetic from your pre-college days). Recall that the correct way to reduce `a/b` is to divide `a` and `b` by the greatest common divisor of `a`, `b`. In other words, if the greatest common divisor of `a` and `b` is `g`, then `a/b` becomes `(a/g)/(b/g)`. For instance `18/24 = (18/6)/(24/6) = 3/4` because the greatest common divisor of `18` and `24` is `6`. The greatest common divisor function can be found in my notes on Recursive functions.

What about the case where the numerator and/or the denominator is not positive?

Besides dividing out by the greatest common divisor, your method must also remove redundant negative signs and ensure the that denominator is positive. For instance is the `Rational` object `r` models `-2/-3`, then after calling `r.reduce()`, `r` models `2/3`. If `r` models `2/-3`, after calling `r.reduce()`, `r` models `-2/3`.

Finally, for the case of `0`. If your `Rational` object `r` models `0/5`, after calling `r.reduce()`, `r` models `0/1`. If `r` models `0/-5`, after calling `r.reduce()`, `r` models `0/1`. In other words if the `Rational` object models the real world concept of `0` (such as `0/1`, `0/5`, `0/-1000`), then after calling `r.reduce()`, r will always model `0/1`.

Note that there is a `reduce()` method in `Rational` and there is also a `reduce()` non-member function. If `r` is a `Rational` object,

```
r.reduce();
```

has the same effect as

```
reduce(r);
```

Computations Involving Rationals and Integers

You also must have the additional functions (outside the class, also known as non-member functions). Note that, as mentioned in class, an expression like `1+x` where `x` is a `Rational` object is translated to either

(a) `1.operator+(x)`, or
(b) `operator+(1,x)`.

Now obviously (a) is not possible since that implies that `1` is an object. So (b) is the only possibility. Therefore you need to implement a function (not a method). It's probably best to bundle this function with the `Rational` class (i.e. the function's prototype is in `Rational.h` and the definition of the function should be in `Rational.cpp`). In other words you need function

```
Rational operator+(int a, const Rational & b);
```

that returns a `Rational` object modeling the sum of `a` and `b`. For instance if `a`=`1` and `b` models the `1/2`, then `operator+(a, b)` must return a `Rational` object modeling `3/2`. Note that your class already knows how to add two `Rational` objects. To avoid code duplication, your `operator+(a, b)` should call the `operator+` in the class.

To be concrete let me repeat myself with some examples.

The `Rational` header, once implemented, must be able to work with `Rational` objects and integers in the same expression. For instance

```
std::cout << Rational(1, 2) + 1 << std::endl;
std::cout << 2 + Rational(1, 3) << std::endl;
```

should produce this output:

```
3/2
7/3
```

In general we want to be able to evaluate

```
a [op] b
```

where `a`,`b` are either `Rational` objects or integers and `[op]` is any operator in the header file.

Note that in fact you don't have to do much to make

```
Rational(1, 2) + 1
```

work ... C++ will convert `1` to `Rational(1)` for you!!! This is automatic type conversion. You have actually seen that before. Recall that in CISS240 when you do

```
1.234 + 1
```

C++ will convert `1` to `1.0` (as a `double`) for you

```
1.234 + 1.0
```

and then use `double` addition to carry out the computation. It's the same situation for

```
Rational(1, 2) + 1
```

C++ will attempt to convert `1` to a `Rational` object. This can be done with the constructor, i.e. `Rational(1)`. And C++ will do it for you automatically, i.e., C++ will execute

```
Rational(1, 2).operator+(Rational(1))
```

when it sees

```
Rational(1, 2) + 1
```

So there's nothing to do at all! (Of course C++ will also try to match `Rational(1, 2) + 1` with other `+` operators. For instance there's the usual `+` for (int, int) from CISS240. But that does not apply since there's no way to automatically convert `Rational(1, 2)` to an `int` value.)

However for the expression

```
2 + Rational(1, 3)
```

we will need to define

```
Rational operator+(int, const Rational &)
```

This is the function that C++ will use for the evaluation of

```
2 + Rational(1, 3)
```

Likewise you need to implement the following in `Rational.h`:

```
bool operator==(int, const Rational &);
bool operator!=(int, const Rational &);
bool operator> (int, const Rational &);
bool operator>=(int, const Rational &);
bool operator< (int, const Rational &);
bool operator<=(int, const Rational &);

int & operator+=(int &, const Rational &);
int & operator-=(int &, const Rational &);
int & operator*=(int &, const Rational &);
int & operator/=(int &, const Rational &);
```

```
Rational operator+(int, const Rational &);
Rational operator-(int, const Rational &);
Rational operator*(int, const Rational &);
Rational operator/(int, const Rational &);
```

Note that these prototypes are non-member function prototypes outside the `Rational` class definition, i.e., these are not methods of the class.

You must have the least amount of code duplication in your implementation. [HINT: For instance `1 + Rational(1, 2)` is the same as `Rational(1, 2) + 1`.]

Note added: Here's an important note on binary operators involving values of different types. Look at for instance an expression of the form

```
int i = 5;
Rational r(3, 2); // i.e., r = 3/2
i /= r;
```

The intended effect is to set `i` to the value of `i / r`. But how do you compute `i / r`? Note that `i` is an integer while `r` is a fraction. Should `i / r` be `(5/1) / (3/2)` which would give you `10/3` and then you give `3` to `i`, or should `i / r` be `5/1` which would give `5` to `i`. The first *promotes* `5` to `5/1` while the second *demotes* `3/2` to `1`. You should adopt the "automatic type promotion" approach from CISS240 because for C++ that's the general expectation. For instance try this:

```
int i = 5;
double r = 3.0 / 2;
i /= r;
std::cout << i << '\n';
```

and you'll see that the output is `3` (and not `1`).

Besides the binary operators, you need to also implement the unary operators:

```
Rational Rational::operator+() const; // positive operator (not addition)
Rational Rational::operator-() const; // negative operator (not subtraction)
```

For instance

```
Rational a(1, 2);
Rational b = -a; // same as a.operator-(), so b models -1/2
Rational c = +a; // same as a.operator+(), so c models 1/2
```

Make sure you see the difference between the following:

`a - b`	same as	`a.operator-(b)`
`-a`	same as	`a.operator-()`
`a + b`	same as	`a.operator+(b)`
`+a`	same as	`a.operator+()`

`Rational::pow()`, `Rational::abs()` and Related Functions

In addition to basic operators there is another method, `pow()`, which returns the power of the `Rational` object invoking the call to the power of the integer passed in. You only need to handle the case of positive integer powers. For instance

```
std::cout << Rational(-2, 3).pow(0) << std::endl;
std::cout << Rational(-2, 3).pow(3) << std::endl;
std::cout << Rational(-2, 3).pow(-3) << std::endl;
```

produce the following output:

```
1
-8/27
-27/8
```

because mathematically

$$(-2/3)^0 = 1$$
$$(-2/3)^3 = (-2/3)(-2/3)(-2/3) = -8/27$$
$$(-2/3)^{-3} = ((-2/3)3)^{-1} = ((-2/3)(-2/3)(-2/3))^{-1} = (-8/27)^{-1} = (-27/8)$$

Instead of calling methods, you can achieve the same thing using the `pow()` function:

```
std::cout << pow(Rational(-2, 3), 0) << std::endl;
std::cout << pow(Rational(-2, 3), 3) << std::endl;
std::cout << pow(Rational(-2, 3), -3) << std::endl;
```

[You should know from MATH104 that $a^0 = 1$ (if a is not zero) and $(a/b)^{-1} = b/a$.]

A function (this is not a method – see the header file) that you need to implement is `abs()` which returns the absolute value (as a `Rational` object) of the `Rational` object passed in. For instance

```
std::cout << abs(Rational(-15, 7)) << std::endl;
```

produces the following output:

```
15/7
```

Besides the above, your `Rational` library also allows you to do this:

```
std::cout << Rational(-15, 7).abs() << std::endl;
```

Type Conversion Operators

There are two methods that act as type conversion. Note in particular that

- `get_int()` returns the integer value of the `Rational` object. For instance if `x = Rational(7,2)`, i.e. `x` models the (real-world) fraction `7/2` which is 3.5, then `x.get_int()` return 3, i.e., the integer part of 3.5.
- `get_double()` returns the `double` value of the `Rational` object. For instance if `x = Rational(7,2)`, i.e. `x` models the fraction 7/2 which is 3.5, then `x.get_double()` return 3.5 as a `double`.

Note that I've included (but commented out) the following prototypes

```
//operator int() const;
//operator double() const;
```

These operators are called type conversion operators. Their prototypes look like this:

```
operator [type name]() const;
```

(No return type.) If these were implemented you can do the following:

```
Rational r(7, 2);
int i = (int) r;       // or i = int(r)
double d = (double) r; // or d = double(r)
```

i.e.,

```
    (int) r       is the same as    r.operator int()
    (double) r    is the same as    r.operator double()
```

Of course `operator int()` should do the same thing as `get_int()` and `operator double()` should do the same thing as `get_double()`. However there's a vital difference between `operator int()` and `get_int()`. C++ will automatically use type conversion operators when trying to find a suitable function/method. For instance if you have `operator int()` implemented the following

```
Rational(1, 2) + 1
```

will become one of the following:

```
Rational(1, 2) + Rational(1)       -- 1 is auto converted to Rational(1)
(int) Rational(1, 2) + 1           -- Rational(1,2) is auto converted to 0
(int) Rational(1, 2) + Rational(1) -- both are auto converted
```

which are all valid since we have the following prototypes:

```
Rational::operator+(const Rational &) const
operator+(int, int)
operator+(int, const Rational &);
```

At this point C++ will have problems compiling your program because C++ does not know which function/method to call!!! This situation is called **ambiguous invocation**. In fact it's even worse if you have `operator double()` implemented since C++ will include

```
(double) Rational(1, 2) + 1
```

Note that C++ will only apply an automatic type conversion *once for each parameter*. So for instance C++ will *not* type convert the above to this:

```
Rational(1, 2) +
```

(It's possible to prevent certain cases of ambiguous invocation by preventing automatic type conversion.)

That's why the `int()` and `double()` type conversion operators have been commented out. It *is* possible to implement `int()` and `double()` operators in such a way as to avoid ambiguous invocation but it will be too much work.

By the way

`Rational(1)`

is also considered a type conversion, from `int` to `Rational`.

Test Code

The following test code is incomplete. Add tests to ensure that your Rational library is sufficiently tested.

```
// File: main.cpp

#include <iostream>
#include "Rational.h"


void test_input()
{
    Rational r;
    std::cin >> r;
    std::cout << r.get_n() << ' ' << r.get_d() << std::endl;
}


void test_output()
{
    int n = 0, d = 0;
    std::cin >> n >> d;
    std::cout << Rational(n, d) << std::endl;
}


void test_reduce_method()
{
    Rational r;
    std::cin >> r;
    r.reduce();
    std::cout << r << std::endl;
}


void test_reduce_function()
{
    Rational r;
    std::cin >> r;
    reduce(r);
    std::cout << r << std::endl;
}
```

```
void test_eq()
{
    Rational r, s;
    std::cin >> r >> s;
    std::cout << (r == s) << std::endl;
}


int main()
{
    int option = 0;
    std::cin >> option;

    switch (option)
    {
        case 1:
            test_input();
            break;
        case 2:
            test_output();
            break;
        case 3:
            test_reduce_method();
            break;
        case 4:
            test_reduce_function();
            break;
    }

    return 0;
}
```

Here's the test plan. The numbering refers to the test option.

1. Test input
2. Test output
3. Test reduce method
4. Test reduce nonmember function
5. Test abs method
6. Test abs nonmember function
7. Test `==`
8. Test `!=`
9. Test `>`
10. Test `>=`
11. Test `<`
12. Test `<=`
13. Test `+=`
14. Test `-=`
15. Test `*=`
16. Test `/=`
17. Test `+` (unary)
18. Test `-` (unary)
19. Test `+` (binary)
20. Test `-` (binary)
21. Test `*`
22. Test `/`
23. Test `get_int`
24. Test `get_double`
25. Test `int == Rational`
26. Test `int != Rational`
27. Test `int > Rational`
28. Test `int >= Rational`
29. Test `int < Rational`
30. Test `int <= Rational`
31. Test `int += Rational`
32. Test `int -= Rational`
33. Test `int *= Rational`
34. Test `int /= Rational`
35. Test `int + Rational`
36. Test `int - Rational`
37. Test `int * Rational`
38. Test `int / Rational`
39. Test `pow` non-member function
40. Test `pow` member function of `Rational`

Asides

With the `Rational` class, you can of course perform computations involving fractions.

For instance you know how to find integer solutions to say $x^2 - 4y^2 = 1$ in some range range (say $-10..10$ for both x and y):

```
for x = -10, -9, ..., 9, 10:
    for y = -10, -9, ..., 9, 10:
        if x * x - 4 * y * y == 1, then print x, y
```

For fractions, you can do this:

```
for n0 = -10, -9, ..., 9, 10:
    for d0 = -10, -9, ..., 9, 10:
        for n1 in -10, ..., 10:
            for d1 in -10, ..., 10:
                r = Rational(n0,d0)
                s = Rational(n1,d1)
                if r * r - 4 * s * s == 1, then print r, s
```

In this case, you'll only find solutions with the numerators and denominators in the given range. Note that there is an infinite number of fractions in any closed interval with more than one point. For instance in [0,1], you have 1/2 and 1/3 and 1/4 and ... Another thing to note is that there are many repeats in the above example. For instance the loops will run through 1/2 several times because 3/6, 4/8, 5/10 are all the same as 1/2.

You probably know that

$$1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + \cdots = 1/(1 - 1/2) = 2$$

Now you can use your `Rational` class like the following and see what happens when you compute the left-hand side up to 1/1024:

$$1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + \cdots + 1/1024$$

You can also use your `Rational` to give an approximation to π using this "stack" of fractions expression for π:

$$\pi = \cfrac{4}{1 + \cfrac{1^2}{2 + \cfrac{3^2}{2 + \cfrac{5^2}{2 + \cfrac{7^2}{2 + \cfrac{9^2}{2 + \cfrac{11^2}{\ddots}}}}}}}$$

Obviously if you want to do this, you have to stop at some point, say up to 1001^2. (The expression on the right is called a "continued fraction".)

*** SPOILER ON INEQUALITIES ***

Note that if a, b, c, d are integers, then

$$\frac{a}{b} < \frac{c}{d}$$

is the same as

$$ad < cb$$

if $b > 0$ and $d > 0$. You have to switch the direction of inequality if b is negative, and you have to switch it twice if both b and d are negative. This is from your elementary algebra course (MATH104). This means that

$$\frac{a}{b} < \frac{c}{d} \text{ is the same as } \begin{cases} ad < bc \text{ if } b > 0,\ d > 0 \\ ad > bc \text{ if } b > 0,\ d < 0 \\ ad > bc \text{ if } b < 0,\ d > 0 \\ ad < bc \text{ if } b < 0,\ d < 0 \end{cases}$$

Likewise

$$\frac{a}{b} \leq \frac{c}{d} \text{ is the same as } \begin{cases} ad \leq bc \text{ if } b > 0,\ d > 0 \\ ad \geq bc \text{ if } b > 0,\ d < 0 \\ ad \geq bc \text{ if } b < 0,\ d > 0 \\ ad \leq bc \text{ if } b < 0,\ d < 0 \end{cases}$$