CISS245: Advanced Programming
Assignment 2 (Review)

Objectives

1. Review of CISS240.

For this assignment, you will implement a library for modeling fractions. Each fraction is modeled (of course) with two integer variables (to model the numerator and denominator). The ideas present in this assignment will be used again in a later assignment when you "combine" two integer variable into a single variable.

The questions build on top of each other until the last one. Therefore you should start with Q1, when you are done with Q1, copy the code to Q2, etc. The first 8 questions are very simple. In fact the first few questions already appear in the review set. You will then use the library to factorize polynomials of degree 3 (the last two questions). The only one that really requires more work is Q10.

Q1. The follow skeleton code is given. There are three files. Note that the files are (of course) incomplete not just because the functions are not implemented. The preprocessor directives (i.e., `ifndef`, ...) are not included: you must complete them too.

```
// File  : main.cpp
// Author: smaug

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

void test_Fraction_print()
{
    int xn = 0, xd = 0; // numerator and denominator of a fraction
    std::cin >> n >> d;
    Fraction_print(xn, xd);
    std::cout << std::endl;
}


int main()
{
    int option;
    std::cin >> option:
    switch (option)
    {
        case 1:
            test_Fraction_print();
            break;
    }

    return 0;
}
```

```
// File: Fraction.h
// Author: smaug

//-------------------------------------------------------------------------------
// Print fraction modeled by numerator n and denominator d.
//-------------------------------------------------------------------------------
void Fraction_print(int, int);
```

```
// File: Fraction.cpp
// Author: smaug

#include <iostream>


void Fraction_print(int n, int d)
{
}
```

The following are test cases. Note that underlined texts denotes user input. It should be clear from the test cases that the output is a reduced fraction, i.e., common factors between the numerator and denominator are removed.

Test 1

```
1 1 3
1/3
```

Test 2

```
1 -1 3
-1/3
```

Test 3

```
1 1 -3
-1/3
```

Test 4

```
1 -1 -3
1/3
```

Test 5

```
1 0 1
0
```

Test 6

```
1 0 5
0
```

Test 7

```
1 0 -6
0
```

Test 8

```
1 10 3
10/3
```

Test 9

```
1 10 2
5
```

Test 10

```
1 10 15
2/3
```

Test 11

```
1 60 45
4/3
```

Test 12

```
1 10 0
undefined
```

Test 13

```
1 -5 0
undefined
```

Q2. This is a continuation of Q1. (In other words, after you finished Q1, copy the files from Q1 to the directory/folder for Q2.)

Now include a function to add fractions in your library. Test it throughly. I will not supply test cases since you should know how to add fractions!

See the skeleton code below for the prototype.

Note that, as stated in the comments for the function to add fractions, the add function does *not* simplify the fraction, i.e., it does *not* perform reduction of the fraction by divide the numerator and denominator by the greatest common divisor.

```
// File  : main.cpp
// Author: smaug

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

void test_Fraction_print()
{
    int xn = 0, xd = 0; // numerator and denominator of a fraction

    std::cin >> xn >> xd;
    Fraction_print(xn, xd);
    std::cout << std::endl;
}


void test_Fraction_add()
{
    int xn = 0, xd = 0; // Fraction xn/xd
    int yn = 0, yd = 0; // Fraction yn/yd
    int zn = 0, zd = 0; // Fraction zn/zd

    std::cin >> yn >> yd >> zn >> zd;
    Fraction_add(xn, xd, yn, yd, zn, zd);
    Fraction_print(xn, xd);
    std::cout << std::endl;
}


int main()
{
    int option;
    std::cin >> option:
    switch (option)
    {
        case 1:
            test_Fraction_print();
            break;
        case 2:
            test_Fraction_add();
            break;
    }

    return 0;
}
```

```
// File: Fraction.h

//--------------------------------------------------------------------------------
// Print fraction modeled by numerator n and denominator d.
//--------------------------------------------------------------------------------
void Fraction_print(int n, int d);

//--------------------------------------------------------------------------------
// The fraction modeled by xn and xd as numerator and denominator, i.e., xn/xd,
// is set to the sum of the fractions modeled by yn/yd and zn/zd, i.e.,
// xn/xd = yn/yd + zn/zd
//--------------------------------------------------------------------------------
void Fraction_add(int & xn, int & xd,
                  int yn, int yd,
                  int zn, int zd);
```

```
// File: Fraction.cpp

void Fraction_print(int n, int d)
{
}


void Fraction_add(int & xn, int & xd,
                  int yn, int yd,
                  int zn, int zd)
{
}
```

Q3. This is a continuation of Q2. (In other words, after you finished Q2, copy the files from Q2 to the directory/folder for Q3.)

Now include a function to subtract fractions in your library. Test it throughly.

You must add a test function to the main C++ file and allow user to test your function when the user enters option 3. (Refer to Q2.) Previous code for testing with option 1 and 2 must be kept.

```
// File: Fraction.h

//-----------------------------------------------------------------------------
// Print fraction modeled by numerator n and denominator d.
//-----------------------------------------------------------------------------
void Fraction_print(int n, int d);

//-----------------------------------------------------------------------------
// The fraction modeled by xn and xd as numerator and denominator, i.e., xn/xd,
// is set to the sum of the fractions modeled by yn/yd and zn/zd, i.e.,
// xn/xd = yn/yd + zn/zd
//-----------------------------------------------------------------------------
void Fraction_add(int & xn, int & xd,
                  int yn, int yd,
                  int zn, int zd);

//-----------------------------------------------------------------------------
// The fraction modeled by xn and xd as numerator and denominator, i.e., xn/xd,
// is set to the difference of the fractions modeled by yn/yd and zn/zd, i.e.,
// xn/xd = yn/yd - zn/zd
//-----------------------------------------------------------------------------
void Fraction_sub(int & xn, int & xd,
                  int yn, int yd,
                  int zn, int zd);
```

```
// File: Fraction.cpp

void Fraction_print(int n, int d)
{
}


void Fraction_add(int & xn, int & xd,
                  int yn, int yd,
                  int zn, int zd)
{
}


void Fraction_sub(int & xn, int & xd,
                  int yn, int yd,
                  int zn, int zd)
{
}
```

Q4. This is a continuation of Q3. (In other words, after you finished Q3, copy the files from Q3 to the directory/folder for Q4.)

Now include a function to multiply fractions in your library. Test it throughly. The name of the function must be `Fraction_mult`. (There's only one reasonable prototype of this function.)

You must add a test function to the main C++ file and allow user to test your function when the user enters option 4. (Refer to Q2.) Previous code for testing with option 1, 2, and 3 must be kept.

Q5. This is a continuation of Q4.

Now include a function to divide fractions in your library. Test it throughly. The name of the function must be `Fraction_div`. (There's only one reasonable prototype of this function.)

You must add a test function to the main C++ file and allow user to test your function when the user enters option 5. (Refer to Q2.) Previous code for testing with option 1, 2, 3, and 4 must be kept.

Q6. This is a continuation of Q5.

It's reasonable to also have augmented operations on fractions so that we can perform something like `a/b += c/d`, i.e., a function called `Fraction_addeq`. For instance if `a/b` is 1/2 and `c/d` is 1/4, after `a/b += c/d`, a/b becomes 3/4 (and `c/d` is unchanged.) I'll leave it to you to try that on your own just for practice. These and other operators will appear in a future assignment.

For this question, we start writing comparison operations.

Now include a function to check when two fractions are the same. Test it throughly. The name of the function must be `Fraction_eq`. The function returns a boolean value. (There's only one reasonable prototype of this function.)

You must add a test function to the main C++ file and allow user to test. your function when the user enters option 6. (Refer to Q2.) Previous code for testing with option 1, 2, 3, 4, and 5 must be kept.

WARNING: 1/2 is equal to $-2/-4$, $-1/2$ is equal to $1/(-2)$.

I'm sure you know how to check if two fractions are the same. But I'll give you some test cases anyway, so that we can agree on the output format. Basically, you print the boolean value returned by the function you are implementing.

Test 1

```
6 1 2 -2 -4
1
```

Test 2

```
6 -1 2 1 -2
1
```

Test 3

```
6 1 2 2 1
0
```

Q7. This is a continuation of Q6.

Now include a function to check when two fractions are not the same. Test it throughly. The name of the function must be `Fraction_neq`. The function returns a boolean value. (There's only one reasonable prototype of this function.)

Note that you must use the `Fraction_eq` function, i.e., you must call the `Fraction_eq` function. In fact this function should be very easy.

You must add a test function to the main C++ file and allow user to test your function when the user enters option 7. (Refer to Q2.) Previous code for testing with option 1, 2, 3, 4, 5, and 6 must be kept.

Q8. This is a continuation of Q7.

Now include a function to check when a fraction is less than another. Test it throughly. The name of the function must be `Fraction_lt`. The function returns a boolean value. (There's only one reasonable prototype of this function.)

You must add a test function to the main C++ file and allow user to test your function when the user enters option 8. (Refer to Q2.) Previous code for testing with option 1, 2, 3, 4, 5, 6, and 7 must be kept.

WARNING: Make sure you test negative fractions.

Test 1

```
8 1 2 1 2
0
```

Test 2

```
8 1 2 1 -2
0
```

Test 3

```
8 3 8 1 2
1
```

Test 4

```
8 -3 8 1 2
1
```

Test 5

```
8 -3 8 -1 2
0
```

Test 6

```
8 2 4 -3 8
0
```

Test 7

```
8 2 4 -3 -8
0
```

Q9. [Brute force factorization of rational polynomial]

This question uses the Fraction library you built in Q1-Q8.

The goal is to write a program that attempts to factorize a polynomial of degree 3. The polynomial has fractional coefficients. We will assume the coefficient of the highest degree term is 1. Here's an example:

$$x^3 - \frac{13}{12}x^2 + \frac{3}{8}x - \frac{1}{24}$$

Now it turns out that the above polynomial is the same as

$$\left(x - \frac{1}{2}\right)\left(x - \frac{1}{3}\right)\left(x - \frac{1}{4}\right)$$

We also say that 1/2, 1/3, 1/4 are roots of the above given polynomial. (This is recorded as Test 1 below.)

Here's another example:

$$x^3 - \frac{25}{12}x^2 + \frac{13}{9}x - \frac{1}{3} = \left(x - \frac{2}{3}\right)\left(x - \frac{2}{3}\right)\left(x - \frac{3}{4}\right)$$

Our goal is this: If the user gives you fractions a, b, c

$$x^3 + ax^2 + bx + c$$

you need to compute fraction (if possible) d, e, f such that

$$x^3 + ax^2 + bx + c = (x - d)(x - e)(x - f)$$

We will do this by brute force: We will let d run over a range of fractions, e run over a range of fractions, and f run over a range of fractions. Now find d, e, f are fractions, they are made up of numerators and denominators. I'll let d be `dn/dd`, e be `en/ed`, f be `fn/fd`. Likewise let a be `an/ad`, b be `bn/bd`, c be `cn/cd`. Altogether we have 6 inputs and 6 outputs. We will let the numerators `dn`, `en`, `fn` run through the range $[-50, 50]$ and `dd`, `ed`, `fd` denominators run through the same $[1, 50]$. So your code structure to enumerate fraction `d` in the above ranges is:

```
for dn = -50, ..., 50:
    for dd = 1, ..., 50:
        ...
```

Of course the enumeration of all fractions `d` and `f` would look like:

```
for dn = -50, ..., 50:
    for dd = 1, ..., 50:
        for en = -50, ..., 50:
            for ed = 1, ..., 50:
                ...
```

The enumeration of each fraction requires two for-loops. So what do you do in the innermost body of the 6 for-loops:

```
for loops over d, e, f: (6 loops)
         if (x-d)(x-e)(x-f) is x^3 + ax^2 + bx + d:
             print (x-d)(x-e)(x-f)
```

But how do you check `(x-d)(x-e)(x-f) is x^3 + ax^2 + bx + d` since `C++` cannot do such comparison. `C++` don't even understand polynomials since there's no such thing as a polynomial type.

Well it's not that bad. Since you can only work with integers, and now fractions since you have a small fraction library, you have to work directly with coefficients. Note that, using college algebra

$$\begin{aligned}(x-d)(x-e)(x-f) &= (x^2-(d+e)x+de)(x-f)\\ &= x^3-(d+e+f)x^2+(de+df+ef)-(def)\end{aligned}$$

Therefore to say

$$(x-d)(x-e)(x-f) = x^3+ax^2+bx+c$$

is the same as saying

$$x^3-(d+e+f)x^2+(de+df+ef)-(def) = x^3+ax^2+bx+c$$

which is the same as saying

$$\begin{aligned}-(d+e+f) &= a\\ de+df+ef &= b\\ -(def) &= c\end{aligned}$$

[OPTIMIZATION: From

$$\begin{aligned}-(d+e+f) &= a\\ de+df+ef &= b\\ -(def) &= c\end{aligned}$$

once d, e are given, f is determined:

$$
\begin{aligned}
f &= -a - d - e \\
f &= \frac{b - de}{d + e} \\
f &= -\frac{c}{de}
\end{aligned}
$$

]

These are three boolean equality comparisons of fractions (and you have the `Fraction_eq` function) and the fractions might involve fraction operations (mainly addition and multiplication). The only operator you need which is not implemented in the previous questions is the "negative of". So you might want to add a `Fraction_neg` function. The general idea is this:

```
for loops over d, e, f: (6 loops)
        compute fraction x = -(d + e + f)
        compute fraction y = d * e + d * f + e * f
        compute fraction z = -(d * e * f)
        if x is a and y is b and z is c,
            print (x-d)(x-e)(x-f)
```

One last thing: you must make sure you organize `d,e,f` so that they are in ascending order: `d` $\leq$ `e` $\leq$ `f`.

If no factorization is found (either the polynomial cannot be factored or the the roots are outside our range), you print `not found`.

Test 1

```
-13 12 3 8 -1 24
x^3 + (-13/12)x^2 + (3/8)x + (-1/24) = (x - (1/4))(x - (1/3))(x - (1/2))
```

Note: The roots on the right must be listed in ascending order.

Test 2

```
-26 15 8 15 0 1
x^3 + (-26/15)x^2 + (8/15)x + (0) = (x - (0))(x - (2/5))(x - (4/3))
```

Test 3

```
-6 11 0 1 0 1
x^3 + (-6/11)x^2 + (0)x + (0) = (x - (0))(x - (0))(x - (6/11))
```

Test 4

```
-19 2 27 1 -45 2
x^3 + (-19/2)x^2 + (27)x + (-45/2) = (x - (3/2))(x - (3))(x - (5))
```

Test 5

```
-244 105 188 105 -16 35
x^3 + (-244/105)x^2 + (188/105)x + (-16/35) = (x - (2/3))(x - (4/5))(x - (6/7))
```

Test 6

```
-217 30 44 3 -91 10
x^3 + (-217/30)x^2 + (44/3)x + (-91/10) = (x - (7/5))(x - (3/2))(x - (13/3))
```

Test 7

```
-95 42 61 42 -2 7
x^3 + (-95/42)x^2 + (61/42)x + (-2/7) = (x - (3/7))(x - (1/2))(x - (4/3))
```

Test 8

```
-9 1 87 4 -35 4
x^3 + (-9)x^2 + (87/4)x + (-35/4) = (x - (1/2))(x - (7/2))(x - (5))
```

Test 9

```
-25 12 13 9 -1 3
x^3 + (-25/12)x^2 + (13/9)x + (-1/3) = (x - (2/3))(x - (2/3))(x - (3/4))
```

Test 10

```
-60 47 1200 2209 -8000 103823
x^3 + (-60/47)x^2 + (1200/2209)x + (-8000/103823) = (x - (20/47))(x - (20/47))(x - (20/47))
```

Test 11

```
-143 23 217 23 -97 23
not found
```

Q10. [Brute force factorization of rational polynomial (with some improvement)]

Notice that your program from Q9 is *really* slow.

Now speed up your program from Q9 in the following way: Note that if n runs through [-50, 50] (integer values) and d runs through $[1, 50]$, you will get $101 \times 50 = 5050$ fractions. But many of these fractions are repeats. For instance

$$1/1 = 2/2 = 3/3 = 4/4 = ... = 50/50$$

which gives you 50 fractions which are the same. Also, $-1/1 = -2/2 = -3/3 = -4/4 = ... = -50/50$. You also have

$$1/2 = 2/4 = 3/6 = 4/8 = ... = 25/50$$

which gives you 25 fractions which are the same. Etc.

So there's no hope in factorizing for instance

$$x^3 - 1973/816 * x^2 + 1103/1224 * x - 505/7344$$

since the factorization is

$$\left(x - \frac{101}{51}\right)\left(x - \frac{1}{3}\right)\left(x - \frac{5}{48}\right)$$

which contains roots with numerators and/or denominators greater than 50. You can try to expand your range to 100 and see how long it takes. Try this if you don't believe me:

```
for (int i = -100; i <= 100; ++i)
{
    std::cout << i << '\n';
    for (int j = 1; j <= 100; ++j)
    {
        for (int k = -100; k <= 100; ++k)
        {
            for (int l = 1; l <= 100; ++l)
            {
                for (int m = -100; m <= 100; ++m)
                {
                    for (int n = 1; n <= 100; ++n)
                    {
                        int a;
                    }
                }
            }
        }
    }
}
```

How would you not check the redundant cases?

Here's the challenge. Factorize this one:

$$x^3 - 79477/10212 * x^2 + 172871/10212 * x - 6790/851$$

You may assume the numerator and denominator range in absolute value is at most 100.

SPOILER ALERT!!! ... SPOILER ALERT!!! ... SPOILER ALERT!!!

As a pre-processing step, create a collection of unique fractions in your range, i.e., with numerators in [-100, 100] and denominators in [1, 100].

Then run through this collection of unique fractions with the same idea as in Q9.

At this point, you don't know how to create an array of fractions. You do know how to create an array of integers (or doubles or bools or chars). So your collection of fractions has to be an array of numerators and an array of denominators. (You have seen this before in CISS240.) So if the numerator array is `n` and the numerator array is `d`, then `n[5]`, `d[5]` would be the fifth unique fraction in your collection.

You can also sort this collection of fractions. Why would you do that? Before in your loops over the fractions, your second loop can start with the same fraction value as the first, i.e., the second loop should NOT start with the first fraction in your collection of fractions. Therefore the structure of code should look like this:

```
for i = 0, 1, 2, ..., n - 1: // assuming you have n unique fractions
    for j = i, ..., n - 1:
        for k = j, ..., n - 1:
            do something with fraction n[i]/d[i] and n[j]/d[j] and n[k]/d[k]
```

Get it?

Another thing you should be aware of (which might be useful in the future). Besides the `int` type, there's also something called the `long long int` type which can represent larger integers not covered by `int`. There's also something called `long double`. These are very useful for number crunching operations in cryptography, stocks, physics, etc.