COMP 322: Assignment 1 - Winter 2014

Due Feb 5th 2015, at 11:30pm

1 Introduction

For this assignment, we will play with the concept of continued fractions to experiment with C-style pointers/references and with basic C++ syntax. As much as people have fun memorizing the random sequence of digits in the decimal representation of π (67890 is no joke), there are other representations that express the importance of this tasty number.

$$\pi = \frac{3}{10^0} + \frac{1}{10^1} + \frac{4}{10^2} + \frac{1}{10^3} + \frac{5}{10^4} + \frac{9}{10^5} + \frac{2}{10^6} + \frac{6}{10^7} + \frac{5}{10^8} + \frac{3}{10^2} + \cdots (?)$$

$$= \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} - \frac{4}{15} + \frac{4}{17} - \frac{4}{19} + \cdots$$

$$= 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{9 \times 10 \times 11} + \frac{4}{11 \times 12 \times 13} - \frac{4}{13 \times 14 \times 15} + \cdots$$

$$= \cfrac{4}{1 + \cfrac{1^2}{2 + \cfrac{3^2}{2 + \cfrac{5^2}{2 + \cfrac{7^2}{2 + \cfrac{9^2}{2 + \ddots}}}}}} = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5 + \cfrac{3^2}{7 + \cfrac{4^2}{9 + \ddots}}}}} = 3 + \cfrac{1^2}{6 + \cfrac{3^2}{6 + \cfrac{5^2}{6 + \cfrac{7^2}{6 + \cfrac{9^2}{6 + \ddots}}}}}$$

$$= 2 + \cfrac{2}{1 + \cfrac{1}{1/2 + \cfrac{1}{1/3 + \cfrac{1}{1/4 + \ddots}}}} = 2 + \cfrac{2}{1 + \cfrac{1 \cdot 2}{1 + \cfrac{2 \cdot 3}{1 + \cfrac{3 \cdot 4}{1 + \ddots}}}}$$

I am sure many beautiful algorithms run through your mind as you look at all representations (except the first one, which we'll leave to the pi-ranking-hungry people, while we enjoy our pie and write pretty programs to compute π).

Notes on the assignment:

- Feel free to submit any solution for the 0-credit question. Do not worry about style or commenting, unless you want the instructor or the TAs to help you with it.
- Collaboration is strongly encouraged for all questions, but make sure your graded submission reflects individual work. The Discussion Board, me and the TAs are there for you to use as well.
- It might happen that some topics in this homework won't be covered in class. This is normal as we only have one hour of lecture per week. You should not have trouble finding out answers in C++ online tutorials, and as always, don't forget to e-mail us or discuss with other students.
- We provide a `Makefile` and a header file `continued.h`. Your submission should compile using the `make` command on Trottier (3rd floor) machines when run on the **original** `Makefile`, **original** `continued.h` and your submission `continuedStudent.cpp`. **ONLY submit the** `.cpp` **file.**

2 Simple continued fractions

For this first homework we will only look at *simple continued fractions* of the following standard form:

$$x = x_0 + \cfrac{1}{x_1 + \cfrac{1}{x_2 + \cfrac{1}{x_3 + \cfrac{1}{x_4 + \cfrac{1}{x_5 + \cfrac{1}{x_6 + \ddots}}}}}}$$

where $x_0, x_1, x_2, x_3, \cdots$ are all positive integers. We will use the shorthand $[x_0; x_1, x_2, x_3, \cdots]$. Unfortunately π does not have a nice regular continued fractions representation, but other special numbers do. For example.

- All rational numbers have a finite such representation
- All quadratic irrational numbers (i.e. solutions to quadratic equations $ax^2 + bx + c = 0$) have periodic continued fractions
- The golden ratio $\varphi = [1; 1, 1, 1, 1, \ldots]$
- The Euler constant $e = [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, \ldots]$

3 Assignment Requirements

Question 1 (0 credits): The Euclidean algorithm can be used compute the greatest common divisor of two integers a and b (assume $0 < a < b$). It is based on a very simple rule: if q and r are the quotient and remainder of b divided by a ($b = a * q + r$ and $r < a$), then $gcd(a, b) = gcd(r, a)$. Write an algorithm to compute the GCD of two numbers based on this rule (the base case should be $gcd(0, x) = x$). Follow the following declaration:

```
unsigned int gcd(unsigned int, unsigned int);
```

Question 2 (0 credits): A much quicker version of the algorithm in Question 1 is based on rules that do not depend on integer division:

- $gcd(x, y) = gcd(y, x)$
- $gcd(2x, 2y) = 2 \times gcd(x, y)$
- $gcd(2x, 2y + 1) = gcd(x, 2y + 1)$
- if $x > y$, $gcd(2x + 1, 2y + 1) = gcd(x - y, 2y + 1)$

Write an algorithm that uses **only** division by 2 and subtraction to compute the GCD of two numbers. For fun, try to avoid division altogether by using only bitwise operations (no division!).

```
unsigned int gcdFaster(unsigned int, unsigned int);
```

Question 3 (35 credits): Based on Question 1, compute the continued fraction representation of b/a for two integers b and a, with $b > a$. See below how the GCD algorithm in Question 1 gives us this representation:

$$\frac{\mathbf{75}}{\mathbf{33}} = 2 + \frac{1}{\frac{\mathbf{33}}{\mathbf{9}}} = 2 + \frac{1}{3 + \frac{1}{\frac{\mathbf{9}}{\mathbf{6}}}} = 2 + \frac{1}{3 + \frac{1}{1 + \frac{1}{\frac{\mathbf{6}}{\mathbf{3}}}}} = 2 + \frac{1}{3 + \frac{1}{1 + \frac{1}{2}}}$$

What is the GCD of 75 and 33? What are 2 and 9 when we divide 75 by 33? The continued fraction of 75/33 is $[2; 3, 1, 2]$.

We will represent continued fractions using the following `struct`:

```
struct ContinuedFraction {
   int head;
   ContinuedFraction *tail;
};
```

For example, 75/33 is a `ContinuedFraction` for which `head` is equal to `2` and `tail` is a pointer to a `ContinuedFraction` representing 33/9. Another example is 2, which is a `ContinuedFraction` for which `head` is equal to `2` and `tail` is `nullptr`.

Write a method with signature

```
ContinuedFraction *getCFlargerThanOne(unsigned int b, unsigned int a);
```

that returns the continued fraction of b/a (NOT of a/b), assuming that b is larger than a.

Question 4 (10 credits): Assume that $x > 1$ and that the continued fraction of x looks as below:

$$x = x_0 + \frac{1}{x'}$$

for $x_0 > 0$ and $x' > 1$.

Then:

$$\frac{1}{x} = 0 + \frac{1}{x} = 0 + \cfrac{1}{x_0 + \cfrac{1}{x'}}$$

If $x > 1$ and the continued fraction of x is $[x_0; x_1, x_2, ...]$, what is the continued fraction of $1/x$? Write a method with signature

```
ContinuedFraction *getCF(unsigned int b, unsigned int a);
```

which computes the continued fraction of any positive fraction b/a (using Question 3).

Question 5 (30 credits) How can we represent a periodic continued fraction, like that of the golden ratio $\varphi = [1; 1, 1, 1, \cdots]$? Since `tail` is a pointer to a `ContinuedFraction`, there is nothing stopping us from having the pointer point to the same `struct`. After all, the golden ratio has the property that

$$\varphi = 1 + \frac{1}{\varphi}$$

This is how one can build the golden ratio:

```
ContinuedFraction goldenRatio;
goldenRatio.head = 1;
goldenRatio.tail = &goldenRatio;
```

Write a method that builds any periodic continued fraction:

```
ContinuedFraction *getCF(unsigned int head, ContinuedFraction *fixedPart,
                         ContinuedFraction *period);
```

which takes three parameters: the head of the representation, the sequence of integers that are not part of the period in the representation, and the period. That is, for input 3, $[4, 1, 3]$ and $[1; 2]$ should return $[3; 4, 1, 3, 1, 2, 1, 2, 1, 2, \dots]$

In order to avoid a slightly more tedious solution that involves copying data, you may feel very tempted to write something like this:

```
...
ContinuedFraction *toRet = new ContinuedFraction;
...
toRet->tail = &fixed;
...
```

and subsequently let the tail of the last continued fraction in `fixed` point to the first such fraction in `period`.

Do not do it.

It is an insidious mistake that will come back to haunt you.

By saying `toRet->tail = &fixed`, you give both `fixed` and `toRet->tail` the same address in memory. As a result, should the method caller decide to change the contents of `fixed`, or if the variable goes out of scope, the value of `toRet->tail` will be compromised. In a short assignment, this might not always cause problems, but it is still a very bad idea; you wouldn't want to get into a bad habit, right? (Note that we've provided the full example of what NOT to do in `continued.h`. Seriously, don't do it.)

A safer alternative is to copy the desired values from the method parameters into newly created ContinuedFraction `struct`s.

Question 6 (25 credits) We will now get approximations to a continued fraction. For this, we will use a `struct` to represent a fraction:

```
struct Fraction {
    int numerator;
    int denominator;
}
```

Write a method with signature

```
Fraction getApproximation(ContinuedFraction *fr, unsigned int n);
```

which returns the fraction representing a truncation of the input continued fraction `fr`, where the truncation has size at most `n`. For input $tr = [3; 4, 1, 3, 1, 2, 1, 2, 1, 2, \dots]$ and $n = 3$ you should return $16/5 = [3; 4, 1]$.

Hint: think recursive! given $x = [x_0; x_1, x_2, x_3, \dots]$,

$$x = 1 + \frac{1}{x'}$$

where $x' = [x_1; x_2, x_3, \dots]$.