Number Theory and Abstract Algebra for Programmers

Noel Niles

December 2, 2017

Contents

1 Preface

During my first discrete math class I became interested in number theory. I was interested because it seemed complex, but it was based on the same rules I had learned in elementary school. The more I learned the more connections I saw. I started thinking "Why wasn't this explained earlier?". Later, I took some linear algebra, calculus, abstract algebra and read as much as I could in between my normal computer science classes. The more I learned about these abstract theories the more I saw how they all connected and could be useful (to the bane of my pure math friends). I still think much of this material could be used as a mathematical motivator in elementary level classes.

I am writing this book to solidify my learning, to give it shape. I am a computer scientist, software engineer, not a mathematician, but my interests always bring me back to number theory. I will explain some important concepts related to number theory and abstract algebra and I will show how to apply these concepts using the language I'm familiar with, code.

All of the code in this book is written in go. I made a choice. I think it's a good one. Helpful references for running the code examples can be found in Appendix A. All of the source code is in a git repo somwhere that will be made available at some time.

2 Introduction

Not long ago, it was thought that number theory and abstract algebra were the domain of pure mathemeticians with no applications to the real world. Now, they are major subects of interest with applications in cryptograhy, electronic currency, coding theory, and wireless communication.

This text will not follow any linear format. It won't go from top to bottom or inside to out. It will be a meandering journey just as math is supposed to be. New concepts will be introduced without notice or explanation. Wherever possible I will try to reference more expository material in future chapters but maybe I won't. If your confused about a certain topic then that's a good start.

3 In the begining there was Euclid

3.1 Greatest Common Divisor

How many times does a smaller measure go into a larger measure? How many liters are in a gallon? How many meters in a kilometer? How many π in a day? How many radii in a circumference? These kinds of questions were asked thousands of years ago and have motivated all of fundamental mathematics.

It is just these sorts of questions that fueled the creation of Euclid's elements centuries ago. And this is why every number theory book that I have ever read begins with Euclid's Greatest Common Divisor algorithm. I have often thought that it would be nicer to learn about sets and groups first (after I had learned about sets and groups of course), but when talking about sets and groups the GCD plays an important role in describing their structure and properties, so I will stick with tradtion and begin with Euclid's GCD algorithm.

First let us state the GCD algorithm the way Euclid did. Euclid thought of numbers as measures or lengths. So, we will use the same terminology. A number therefore is a measure between two points. For example if A and B are points AB is a number.

Proposition 1 (To find the greatest common measure of two numbers)**.**
Let AB and CD be two numbers with AB the less. We want to find the greatest common measure of AB and CD. Either AB can measure CD or it cannot. If AB measures CD then it is the greatest common measure because it measures itself and itself is the greatest magnitude that can measure itself.

If AB does not measure CD we can repeatedly subtract the lesser from the greater until the remainder does measure CD. The remainder will be either some number or 1 (unity, numero uno). If the remainder is 1 then AB and CD are incommenurable (A.K.A. coprime, relatively prime, RSA cnadidates).

When I think of the Euclidean GCD algorithm I think of learning long division when I was a child. Remember, drawing the frame over the dividend and then placing the divisor on the left. Then you would try to think how many times you could multiply the left number (the divisor) until you got a remainder that was less then the divisor. This process is called Euclidean Division.

In future chapters we will see how valuable the GCD algorithm is.

```
func GCD(a int32, b int32) int32 {
        var u int32
        var v int32
        var t int32
        var x int32

        if a < 0 && a < -math.MaxInt32 {
                fmt.Println("GCD: integer overflow")
                a = -a
        }
        if b < 0 && b < -math.MaxInt32 {
                fmt.Println("GCD: integer overflow")
                b = -b
        }
        if b == 0 {
                x = a
        } else {
                u = a
                v = b
                for v != 0 {
                        t = u % v
                        u = v
                        v = t
                }
                x = u
        }
        return x
}
```

3.2 Extended Greatest Common Divisor

Euclid's algorithm is great but we often want to know what the coeffiecents as well. What I mean is, what if we want to know the x and y that satisfy $ax + by = \gcd(a, b)$

```
func XGCD(a int32, b int32) (int32, int32, int32) {
        var u, v, u0, v0, u1, v1, u2, v2, q, r int32
        var aneg, bneg int32

        if a < 0 {
                if a < -math.MaxInt32 {
                        fmt.Println("XGCD: integer overflow")
                }
                a = -a
                aneg = 1
        }

        if b < 0 {
                if b < -math.MaxInt32 {
                        fmt.Println("XGCD: integer overflow")
                }
                b = -b
                bneg = 1
        }

        u1 = 1
        v1 = 0
        u2 = 0
        v2 = 1
        u = a
        v = b

        for v != 0 {
                q = u / v
                r = u % v
                u = v
                v = r
                u0 = u2
                v0 = v2
                u2 = u1 - q*u2
                v2 = v1 - q*v2
                u1 = u0
                v1 = v0
        }
        if aneg != 0 {
                u1 = -u1
        }
        if bneg != 0 {
                v1 = -v1
        }
        return u, u1, v1
}
```

4 And then there were groups

This chapter is kind of cyclic.