Number Theory and Abstract Algebra for Programmers

Noel Niles

December 3, 2017

Contents

1 Preface **1**

2 Introduction **3**

3 Greatest Common Divisor **5**
3.1 Definitions and Properties 5
3.2 Implementations of the Greatest Common Divisor Algorithm 6
3.2.1 Euclid's Algorithm 6
3.2.2 Recursive Euclidean Algorithm with mod () 7
3.2.3 Recursive GCD with Subtraction Operator 8
3.3 Imperitive GCD 8
3.4 Extended Greatest Common Divisor 9

4 And then there were groups **11**

5 Outstanding in the Field **13**

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.

However, it won't all be hooey. When I make strong conjectures I will layout and explain the proof. Most of this will be familiar stuff, albeit in a new light, but there may be some new things thrown in.

3 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.

3.1 Definitions and Properties

If u and v are two non-zero integers, the *greatest common divisor* $\gcd(u, v)$ is the greatest number that evenly divides both u and v. Every number divides zero so if $u = v = 0$ the definition doesn't make sense instead by convention we say $\gcd(0, 0) = 0/$. From the definition it follows

$$\gcd(u, v) = \gcd(v, u) \tag{3.1}$$
$$\gcd(u, v) = \gcd(-v, u) \tag{3.2}$$
$$\gcd(u, 0) = |u| \tag{3.3}$$

From the above equations we can assume that u and v are are positive integers. By the *Fundamental Theorem of Arithmetic* every positive integers can be factored into the form

$$u = 2^{u_2}3^{u_3}5^{u_5}7^{u_7}\ldots = \prod_{p\,\text{prime}} p^{u_p} \tag{3.4}$$

Related to the GCD function is the *lowest common multiple (LCM)* function. The LCM function is the one you learned in school when adding fractions. LCM looks very similar to GCD.

$$\gcd(u, v) = \prod_{p\,\text{prime}} p^{\min(u_p, v_p)} \tag{3.5}$$
$$\operatorname{lcm}(u, v) = \prod_{p\,\text{prime}} p^{\max(u_p, v_p)} \tag{3.6}$$

Using the definition of the GCD and LCM functions we can derive several useful identities.

$$\gcd(u,v)w = \gcd uv, uw, \qquad if w \geq 0 \tag{3.7}$$
$$\mathrm{lcm}(u,v)w = \mathrm{lcm}(uw, vw), \qquad if w \geq 0 \tag{3.8}$$
$$u \cdot v = \gcd(u,v) \cdot \mathrm{lcm}(u,v) \qquad if u, v \geq 0 \tag{3.9}$$
$$\gcd(\mathrm{lcm}(u,v), \mathrm{lcm}(u,w)) = \mathrm{lcm}(u, \gcd(v,w)) \tag{3.10}$$
$$\mathrm{lcm}(\gcd(u,v), gcd(u,w)) = \mathrm{lcm}(u, gcd(v,w)) \tag{3.11}$$

If $gcd(u,v) = 1$ we say that u is relatively prime or coprime to v. Coprime numbers and their properties are very import in group theory and will be discussed more later. Coprime numbers also arise in the RSA and other encryption schemes. Therefore it is important as a programmer to understand greatest Common divisor and be able to implement it efficiently and correctly. In the following sections we will discuss various ways to implement the greatest common divisor algorithm and discuss which methods are the best in practice.

3.2 Implementations of the Greatest Common Divisor Algorithm

3.2.1 Euclid's Algorithm

Every textbook on number theory that I have ever read begins with a discussion of a GCD algorithm that is attributed to Euclid. Many algorithm and discrete math classes also begin with Euclid's algorithm. Euclid's algorithm is often used to introduce recursive functions to beginning programmers. The popularity of this method is due to it's usefulness and age. Euclid's GCD method can be found in *Elements* Book 7, Propositions 1 and 2, (*c.* 300 B.C) but he probably didn't invent it. It has survived over 2200 years to the present day with suprisingly few improvements by modern algebra methods.

In

Proposition 1 (To find the greatest common measure of two positive numbers)**.**
The greatest common measure of two lengths can be found by repeatedly subtracting the smaller of the two from the larger until the smaller divides the larger.

Proof. Let A, C be two positive integers. We want to find the greatest common divisor of A and C. If $C \mid A$ we are done becasue $C \mid C$ and C is the largest divisor of itself. Therefore C is the greatest divisor of itself and it divides A making it the greatest common divisor.

If $C \nmid A$ then continually substract the smaller from the larger until some number remains that divides the previous one. This will eventually terminate on some number or unity. If the result is unity then it will certainly divide the previous number and A, C are coprime.

Let E be the remainder of A divided by C. And let F be the remainder of C divided by E. Since F divides E and E divides $C - F$, F also divides $C - F$ meaning F divides itself and F divides C. And C divides $A - E$; therefore F also divides $A - E$, but it also divides E therefore it also divides E therefor it divides A. Thus, F is a common divisor of A and C.

Now we must show that F is the *greatest* common divisor. Suppose there exists a number $G > F$ that divides both A and C. G divides C and C divides $A - E$ and G divides $A - E$, so G also divides the whole of A so it divides the remainder E. But E divides $C - F$ so G divides $C - F$ and G divides the whole of C so it divides the remainder F. This leads to a contradiction with a larger number dividing a smaller number. Therefore F is the largest number that will divide both A and C. □

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.

The Euclidean GCD algorithm lends itself naturally to recursion. Below are two recursive versions of Euclid's GCD algorithm. The second one is due to Dijkstra and uses substraction rather than the modulus operator which should be more efficient. However these are presented only for reference. For serious work with large numbers we can't use recursion. I tested both recurisive versions on my machine. The second version crash with a segmentation fault because the recursion tree greatest too deep.

3.2.2 Recursive Euclidean Algorithm with mod ()

The basis of this algorithm is the fact that

$$0 < n \leq m \implies \gcd(m,n) = \begin{cases} n & \text{if n divides m without remainder} \\ \gcd(n, remainder\frac{m}{n}) & \text{otherwise} \end{cases}$$

Notice that in this version I replaced the subtraction mentioned used in Euclid's original with the modulus operator. This is possible because division is just repeated subtraction and we don't ever use the quotient just the remainder. Below is the implementation in Go.

```
// RGCD is the recurisive gcd algorithm. This shouldn't be used.
func RGCD(m int32, n int32) int32 {
        if n == 0 {
                return m
        }
        return RGCD(n, m%n)
}
```

3.2.3 Recursive GCD with Subtraction Operator

The next version uses subtraction which is a simpler operation, but it also requires more operations which makes the recursion tree too deep for this implementation to work in the worst case.

```
// This is a slightly better GCD function, but it still
// shouldn't be used because of the recursion.
func RGCD2(a int32, b int32) int32 {
        if a == b {
                return a
        } else if a > b {
                return RGCD2(a - b, b)
        } else {
                return RGCD2(a, b - a)
        }
}
```

3.3 Imperitive GCD

A more memory friendly version of the Euclidean Algorithm is below. Notice that it uses tempory variables u, v, r, x. These are used so that we don't modify the original values. There is also some error checking code because this is the method I would choose in practice.

```
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
}
```

There are quicker GCD algorithms.

3.4 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.

5 Outstanding in the Field

And then Galois got shot. Damn!