Number Theory Notes

Coding Club

March 13, 2024

Mohammed Alshamsi
2021004826
mo.alshamsi@aurak.ac.ae

Department of Computer Science and Engineering
American University of Ras Al Khaimah
2023–24

Contents

1 Introduction

Welcome to the first set of “lecture notes” for the AURAK Coding Club. The idea is to introduce a useful topic that lends itself easily to coding applications. As the title of this document might indicate, these topics will be mathematical in nature, but most (if not all) of it will be intuitive material that doesn’t require much background knowledge.

1.1 How to Read this Document

This is essentially just a short monologue about elementary number theory. We’ll go over the most basic definitions and results, and give some examples. This is by no means meant to give you a comprehensive introduction; it’s just a quick tour to get you started. Check the bibliography

Since this is a coding club and not a math club, I’ll be sure to bring up coding applications whenever it makes sense. Any code I write at those points will be in the C language. If you’ve taken (or are taking) CSCI 112, you’ll be able to follow along without much trouble. Furthermore, there will be coding exercises. You’re free to solve them in any language.

If you don’t know any coding, check Appendix C for a quick intro to C.

1.2 Basics

The main player we’ll be concerned with is the set of integers

$$Z = \{\dots, -2, -1, 0, 1, 2, \dots\}.$$

We’re not going to be dealing with other sets of numbers (rational, irrational, or imaginary) in here. Still, there’s a lot we can do. Let’s first make a very simple definition.

Definition 1.1 (Even and Odd).

- An integer n is *even* if $n = 2k$ for some integer k.
- An integer n is *odd* if $n = 2k + 1$ for some integer k.

I hope it’s easy to see why these definitions are the way they are. Choose any integer k, multiply it by 2, and you’ve an even integer. If you add 1 after that, you guarantee that the result is odd.

We may as well try a quick proof with these definitions. Proofs give insight as to *why* something is true. If you’re interested in properly understanding the material presented here and how each concept follows from another, you should check some books on the topic, like [1]. Now, let’s use the above definition to prove the following fact.

Proposition 1.2.
If n is an odd integer, then n^2 is an odd integer.

In the following proof, we’ll take for granted that the sums and products of integers are also integers.

Proof. Let n be an odd integer. This means $n = 2k + 1$ for some k, by Definition 1.1.

$$n^2 = (2k+1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1. \tag{1}$$

Let's define a to be $(2k^2 + 2k)$. This is clearly an integer. Substituting a into Equation 1, we have that $n^2 = 2a + 1$. Since a is an integer, by Definition 1.1 again we have that $2a + 1$ is an odd integer. So n^2 is odd. $\square$

So that concludes our introduction.

2 Divisibility

Now we can get into the number theory. We'll first introduce divisibility.

We say an integer a *divides* another integer b if the value $k = \frac{b}{a}$ is also an integer. For instance, 5 divides 15 because $\frac{15}{5} = 3$, and 3 is an integer. 6 doesn't divide 15 because $\frac{15}{6} = 2.5$, and 2.5 obviously isn't an integer.

Let's codify everything in a clean definition.

Definition 2.1 (Divisibility).

A nonzero integer a is said to be a *divisor* of an integer b if $b = ak$ for some integer k.

- When a divides b, we write "$a \mid b$".
- When a does not divide b, we write "$a \nmid b$".

Notice that we stated it as $b = ak$, rather than $k = \frac{b}{a}$. This is equivalent, but is a little more convenient to work with.

Example.

- $5 \mid 15$ because $15 = 5 \cdot 3$, and 3 is an integer.
- $6 \nmid 15$ because $15 = 6 \cdot 2.5$, and 2.5 is not an integer.
  In other words, there is no *integer* k for which $15 = 6k$.
- For all n, $n \mid 0$. (Why?)
- $-7 \mid 35$ because $35 = -7 \cdot -5$, and -5 is an integer.

Divisibility is a type of *relation*, and has the properties of reflexivity (for all n, $n \mid n$) and transitivity (if $a \mid b$ and $b \mid c$, then $a \mid c$). We may explore relations further in a future document.

2.1 The Division Algorithm

This is a more precise restatement of something learnt in grade school. At some point in school, you probably divided numbers and expressed a *remainder* as part of the answer. For example, if we divide 17 by 5, the quotient is 3 and the remainder is 2. So we can write

$$17 = 5 \cdot 3 + 2.$$

Theorem 2.2 (The Division Algorithm[1]).

For integers a and m with $m > 0$, there exist unique integers q and r such that

$$a = mq + r, \tag{2}$$

where $0 \leqslant r < m$. We may write $a \bmod m$ to refer to this unique r.

Equation 2 may be a little tough to wrap one's head around, so let's do a few examples.

Example.

- If $a = 17$ and $m = 5$, then $17 = 5 \cdot 3 + 2$. Note that $0 \leqslant 2 < 5$. Also, $17 \bmod 5 = 2$.
- If $a = -26$ and $m = 2$, then $-26 = 2 \cdot -13 + 0$. Note that $0 \leqslant 0 < 5$. Also, $-26 \bmod 2 = 0$.
- If $a = 18$ and $m = 7$, then $18 = 7 \cdot 2 + 4$.
- If $a = -17$ and $m = 5$, then $-17 = 5 \cdot -4 + 3$.

There are a few important points to notice. The first is that the definition allows for r to be 0; this case is precisely when $m \mid a$. The second is that making a negative changes the values of q and r.

At this point we can discuss some coding. In the C programming language, the % operator gives you the remainder of a division:

```
int a = 17, m = 5;
int r = a % m;
printf("%d",r);
```

The second line here sets r to be the remainder of a/m, which is 2. So the output is 2.

The % operator differs from the mod operator in how it handles negative numbers. For instance, let's take $a = -17$ and $m = 5$. Theorem 2.2 tells us that $a \bmod m = 3$. However, if we run the following code, we'll get an output of -2 instead, since the remainder of a/m is -2 and not 3.

```
int a = -17, m = 5;
int r = a % m;
printf("%d",r);
```

Luckily, it's easy to account for this. I'll leave that to you to figure out.

Exercise 1.

Using Theorem 2.2, investigate how q and r behave differently when a is negative, compared to when a is positive.

Exercise 2.

Write a mod function that correctly gives the value of r according to Theorem 2.2.

[1](not an actual algorithm.)

2.2 The Caesar Cipher

The most well-known application of number theory is in cryptography, and at this point, we can present a discussion of the Caesar Cipher. To give a brief introduction, cryptography involves "hiding" a message by transforming it into something that can't be understood, called a *cipher text*. The transformation algorithm is called an *encryption* algorithm, and uses a special value called a *key* to perform the encryption. Only those who know this key[2] can decrypt the cipher text into the original message.

There's many techniques for encryption, and the Caesar cipher is among the simplest. All it involves is "shifting" the alphabet by some number of letters. The number of letters is your key. For instance, if the key is 3, then A maps to D and is replaced by it; B is replaced by E, and so on. Someone who knows the key can decrypt by shifting backward by the same amount, so that D maps to A and so on.

Now for a more precise statement of the algorithm. Many "stream ciphers" — which encrypt each character separately[3] — look like this.

Algorithm 1 (Caesar Cipher).

1. [Choose key.] Choose an integer k between 0 and 25, inclusive.
2. [Encrypt.] For each character, let x be its numerical value. Assign the value $(x+k) \bmod 26$ to x. Replace the character by the letter corresponding to the new value of x.

Example. Below is the entire alphabet with $k = 3$.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Here is the message we want to encrypt:

I HAVE INVENTED A NEW SALAD, TELL THE GREEKS.

For each character, we find it on the top row and replace it by the character below it. This is the encryption step. So when encrypted, "I" gets replaced by "L"; "H" by "K", and so on.

L KDYH LQYHQWHG D QHZ VDODG, WHOO WKH JUHHNV.

Decryption goes the other way: locate the character on the bottom row and replace it by the character above it.

Exercise 3.

Implement Algorithm 1 in a programming language of your choice.

Exercise 4.

It's clear enough that there are only 26 possible keys. Write a program that decrypts a cipher text by trying out all possible keys. (This is called a brute-force approach.)

[2]We'll see in Section 3.3 that the "sender" in this case can encrypt a message and not be able to decrypt it.

[3]Another category is "block ciphers", which map between blocks of n characters rather than just one at a time. In a sense, stream ciphers are the special case of $n = 1$.

2.3 The GCD

Definition 2.3 (Greatest Common Divisor). Let a, b, and c be integers. If $c \mid a$ and $c \mid b$, then c is a *common divisor* of a and b.

The largest such c is called the *greatest common divisor* of a and b, and is denoted $\gcd(a, b)$.

The following theorem is fairly important. Most notably, it's used in the proofs to several bigger theorems.

Theorem 2.4 (Bézout's Identity).
Let a, b, and d be integers with $d = \gcd(a, b)$. For each multiple of d, there exists a pair of integers x and y such that $ax + by$ is equal to this multiple.

A few examples to clarify this:

Example.

- The GCD of 18 and 5 is 1. We can choose $x = 2$ and $y = -7$ to get $18x + 5y = 1$.
- The GCD of 42 and 10 is 2. We can choose $x = 1$ and $y = -4$ to get $42x + 10y = 2$.

The GCD can be computed using Euclid's algorithm. There are a few variants, notably this.

Algorithm 2 (Euclidean Algorithm).
Given two integers m and n, find $\gcd(m, n)$.

1. [Find remainder.] Divide m by n and let r be the remainder.
2. [Is it zero?] If r is 0, the algorithm terminates; n is the answer.
3. [Reduce.] Set m to n, then n to r, and go back to Step 1.

Exercise 5.
Implement Algorithm 2.

2.4 Prime Numbers

Definition 2.5 (Prime Number).
A prime number p is an integer that has no divisors apart from 1 and p.

The first few prime numbers are

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, \ldots$$

We can enumerate prime numbers using the so-called Sieve of Eratosthenes algorithm. The algorithm can be optimized in a few ways, but let's stick with the naïve approach for clarity's sake.

Algorithm 3 (Sieve of Eratosthenes).
Given a positive integer n greater than 1, generate a list of all prime numbers less than or equal to n.

1. [Initialize.] Create a list of consecutive integers from 2 to n. Let $p = 2$.

2. [Mark composites.] Mark all multiples of p up to n, except p itself.
3. [Iterate.] If there is an unmarked integer greater than p in the list, set p to be the smallest such integer, and go to Step 2. Otherwise, terminate; unmarked values are prime, and marked values are composite.

Exercise 6.

Implement Algorithm 3.

The next two theorems are quite important.

Theorem 2.6 (Euclid's Lemma).

If a prime number p divides the product ab of two integers a and b, then p must divide at least one of a or b.

Theorem 2.7 (Fundamental Theorem of Arithmetic).

Every integer greater than 1 can be represented uniquely as a product of prime powers.

The following definition will be relevant in the next section.

Definition 2.8 (Relatively Prime Numbers).

Let a and b be integers. If $\gcd(a, b) = 1$ — that is, if a and b have no common divisors other than 1 — then a and b are said to be relatively prime.

Obviously, every prime number is relatively prime to all other positive integers.

There's an interesting function called Euler's totient function, usually denoted by $\phi(n)$.

Definition 2.9 (Euler's Totient Function).

Let n be an integer. $\phi(n)$ counts how many of the positive integers up to n are relatively prime to n.

It has a few interesting properties. We'll need two of them later on:

Proposition 2.10.

- Whenever n is prime, $\phi(n) = n - 1$.
- For any two relatively prime numbers m and n, $\phi(mn) = \phi(m)\phi(n)$.

3 Modular Arithmetic

The division algorithm (Theorem 2.2) allows us to relate numbers that, when expressed in terms of Equation 2, have the same r.

Definition 3.1 (Congruence Modulo m). For integers a, b, and m, if $m \mid (a - b)$, then we say that a is congruent to b modulo m, and write $a \equiv b \pmod{m}$.

The intuitive idea is that if you divide a by m and b by m, you'll get the same remainder. That is, $a \bmod m = b \bmod m$.[4]

[4] Yes, we now have two different uses of the word "mod". The one in Theorem 2.2 gives a number. Definition 3.1 gives a relation between two numbers, and is used with the $\equiv$ symbol. Here's an example of a property:

$$a \bmod m \equiv a \pmod{m} \qquad \text{but } a \bmod m \text{ isn't necessarily equal to } a.$$

Congruence is another example of a relation, as with divisibility. It is reflexive ($a \equiv a \pmod{m}$), symmetric ($a \equiv b \pmod{m}$ implies $b \equiv a \pmod{m}$), and transitive (if $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$).[5]

We'll give some examples then discuss basic properties.

Example.

- $9 \equiv 21 \pmod{6}$ because $6 \mid (21 - 9)$. In terms of Theorem 2.2, we can say that $21 = 6 \cdot 3 + 3$ and $9 = 6 \cdot 1 + 3$, and emphasize that the remainders are the same.
- $-17 \equiv 4 \pmod{7}$ because $6 \mid (4 - (-17))$. In terms of Theorem 2.2, $-17 = 7 \cdot -3 + 4$ and $4 = 7 \cdot 0 + 4$. Again, the remainders are the same.

For properties, we have three important ones. You can prove them using the definition if you'd like.

Proposition 3.2 (Modular Arithmetic). Suppose that $a \equiv b \pmod{m}$. Then, the following is true for all integers k.

- $a + k \equiv b + k \pmod{m}$.
- If $c \equiv d \pmod{m}$, then $ac \equiv bd \pmod{m}$.
- $a^k \equiv b^k \pmod{m}$.

We can define the following now.

Definition 3.3 (Modular Multiplicative Inverse).

Given relatively prime integers a, m, there exists an integer a^{-1} such that $a^{-1}a \equiv 1 \pmod{m}$. We call a^{-1} the *modular multiplicative inverse* of a.

Now we know enough to discuss more applications.

3.1 The Affine Cipher

The affine cipher is another encryption algorithm, of which the Caesar cipher was only a special case. We could've introduced this directly after the Caesar cipher, but the method of decryption requires modular arithmetic, so we left it until now.

Recall that the numerical equivalents of the English letters are as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Algorithm 4 (Affine Cipher Encryption).

1. [Choose key.] Choose an integer $0 < a < 26$ relatively prime to 26, and any integer $0 \leqslant b < 26$.
2. [Encrypt.] For each letter, take its numerical value x. Find the integer $0 \leqslant y < 26$ such that $y \equiv ax + b \pmod{26}$. Replace by the letter corresponding to y.

[5]These three properties together mean that congruence modulo m is an equivalence relation.

Exercise 7.

Implement Algorithm 4.

Above, we used a formula $ax + b \pmod{26}$ to encrypt each character; the key is the pair (a, b). For decryption, we also find a pair (c, d). The values of c and d will depend on the original choices for a and b. We'll derive the exact relation now.

We know that a must be relatively prime to 26. Since the prime factors of 26 are 2 and 13, this means a cannot have 2 or 13 as a factor. This leaves the following as possible choices for a:

$$1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25$$

For now, let's assume $b = 0$, so that $ax \equiv y \pmod{26}$. Later we'll consider the effect of a non-zero b. Now find the modular multiplicative inverse for every one of the above choices. For example, if $a = 3$, then $a^{-1} = 9$, because $3 \cdot 9 \equiv 1 \pmod{26}$. The reason we want to find a^{-1} is because if $ax \equiv y \pmod{26}$, then $a^{-1}ax \equiv x \equiv a^{-1}y \pmod{26}$.

We can tabulate all possible pairs in this way:

a	1	3	5	7	9	11	15	17	19	21	23	25
a^{-1}	1	9	21	15	3	19	7	23	11	5	17	25

Now let's consider the effect of a non-zero b. Suppose that $ax + b \equiv y \pmod{26}$. Then $ax \equiv y - b \pmod{26}$, and so $x \equiv a^{-1}y - a^{-1}b \pmod{26}$. This tells us that the decryption pair (c, d) is equal to $(a^{-1}, -a^{-1}b)$. Modulo 26, of course.

Exercise 8.

Based on this discussion, come up with the decryption algorithm and implement it.

3.2 Residues and the Chinese Remainder Theorem

"Residue" is just another word for "remainder", but is usually used in the context of having *multiple* moduli that are all relatively prime to one another. Suppose you have an integer x, and a set of moduli $m_1, m_2, \ldots, m_k$.

By taking $x \bmod m_1$, you get a *residue* u_1; needless to say, $x \equiv u_1 \pmod{m_1}$. Take $x \bmod m_2$ and you have u_2, and so on until u_k. The tuple[6]

$$(u_1, u_2, \ldots, u_k)$$

is called the modular representation of x.

Before stating the Chinese Remainder Theorem, let's first present an example, since the above was probably a little notation-heavy.

Example.

Let's take a small example, with $k = 3$ for three moduli. The only restriction on these moduli is that they should be relatively prime. For example, we could take

$$m_1 = 8, m_2 = 21, m_3 = 5.$$

[6](u_1, u_2) is a pair or a couple, and (u_1, u_2, u_3) is a triple, as you know. The word "tuple" is a more general term to represent those and larger ones. Note that tuples are *ordered*, so there's a notion of first, second, etc.

Now choose any x. Let's go with $x = 127$, and take $x \bmod m_i$ for $1 \leqslant i \leqslant 3$:

$$\begin{aligned} u_1 &= x \bmod m_1 &&= 127 \bmod 8 &&= 7 \\ u_2 &= x \bmod m_2 &&= 127 \bmod 21 &&= 1 \\ u_3 &= x \bmod m_3 &&= 127 \bmod 5 &&= 2 \end{aligned}$$

So, the modular representation of x here is $(7, 1, 2)$. The theorem below essentially tells us that 127 is the only number — between 0 and $m_1 m_2 m_3 - 1 = 839$ inclusive — that has the representation $(7, 1, 2)$.

Of course, it's worth reading and understanding the theorem's statement properly.

Theorem 3.4 (Chinese Remainder Theorem). Let $m_1, m_2, \ldots, m_k$ be positive integers that are relatively prime in pairs; that is,

$$\gcd(m_i, m_j) = 1 \quad \text{when } i \neq j.$$

Let $m = m_1 m_2 \ldots m_k$, and let $a, u_1, u_2, \ldots, u_k$ be integers. Then there is exactly one integer x that satisfies the conditions

$$a \leqslant x < a + m, \qquad \text{and} \quad x \equiv u_i \pmod{m_i} \quad \text{for } 1 \leqslant i \leqslant k.$$

The integer a here allows for a certain offset: you don't have to take 0 to $m - 1$ inclusive, and can instead shift this m-sized window by any integer. If you shift it enough, of course, the x will change.

3.3 RSA

We're now at the last application for this topic: the RSA encryption scheme. Unlike the Caesar (Section 2.2) and affine (Section 3.1) ciphers — called symmetric because they use one key — RSA is asymmetric, so it has a pair of keys instead. One is the public key and the other is the private key. Both are generated by the same individual, which is to be the receiver of an encrypted message.[7] The public key may be shared to anyone, while the private key is known only to this receiver. Anyone who wishes to send a message to the receiver uses the public key to encrypt it. The cipher text here can only be decrypted using the private key, meaning only the receiver can decrypt it and read the message.

RSA is yet another stream cipher, encrypting one character at a time.

Algorithm 5 (RSA Encryption).

1. [Choose key.] Choose two primes p and q, and an integer e such that $(p-1)(q-1)$ and e are relatively prime.
2. [Encrypt.] For each letter, take its numerical value x, and replace it with the letter corresponding to with $(x^e \bmod pq)$.

[7]It follows that if you want to have two-way communication, you need two pairs of keys.

For decryption, let $y = x^e \bmod pq$; that is, the value we calculated in Step 3. We need the integer d for which $ed \equiv 1 \pmod{(p-1)(q-1)}$.[8] Simply take $y^d \bmod pq$, and you have x again. To spell things out more explicitly:

$$x = y^d \bmod pq = (x^e \bmod pq)^d \bmod pq = x^{ed} \bmod pq.$$

The use of $(p-1)(q-1)$ may seem odd. This value is equal to $\phi(pq)$, see Proposition 2.10.

Going back to the public and private key, the former is the pair (pq, e) while the latter is (pq, d). What makes RSA secure is that to find d, one first needs to know what p and q are, and in order to find p and q, one must factor the given product pq. Factorization has no known "easy" solution[9]. Just to illustrate, I ask that you try factoring 221 quickly.

Now to explain why all of this works. We'll need the following theorem:

Theorem 3.5 (Euler's Theorem).

For integers a and n, if they are relatively prime, then

$$a^{\phi(n)} \equiv 1 \pmod{n}, \quad \text{or equivalently } a^{\phi(n)+1} \equiv a \pmod{n}.$$

This theorem is a generalization of Fermat's Little Theorem, which originally stated that n specifically be prime. In the context of RSA, it tells us that $x^{\phi(pq)} \equiv 1 \pmod{pq}$, which implies that $x^{k \cdot \phi(pq)+1} \equiv x \pmod{pq}$ by applying Proposition 3.2.

Now for a proof for the correctness of the decryption scheme. We begin by assuming $ed \equiv 1 \pmod{(p-1)(q-1)}$.

Proof. By Theorem 3.5, we know that

$$x^{p-1} \equiv 1 \pmod{p} \quad \text{and} \quad x^{q-1} \equiv 1 \pmod{q}.$$

Applying Proposition 3.2, we have

$$x^{k(p-1)(q-1)+1} \equiv x \pmod{p} \quad \text{and} \quad x^{k(p-1)(q-1)+1} \equiv x \pmod{q}.$$

Since $ed \equiv 1 \pmod{\phi(pq)}$, there is an integer k such that $ed = k\phi(pq) + 1$. That is, $ed = k(p-1)(q-1) + 1$. We may substitute this to get

$$x^{ed} \equiv x \pmod{p} \quad \text{and} \quad x^{ed} \equiv x \pmod{q}.$$

It follows that $x^{ed} \equiv x \pmod{pq}$ by Theorem 3.4. □

[8] d here is the multiplicative inverse of e modulo $(p-1)(q-1)$, see Section 3.3.

[9] It's an NP-hard problem. Maybe this is why I struggled with factorizing in school?

Further Reading

Everything we discussed makes for a very minimalistic overview of elementary number theory; there wasn't much in the way of explanations, proofs, examples, or exercises. These things are all available in dedicated textbooks on the matter. Below are five that I referenced while preparing this document, and which include expositions on a great deal of other material.

[1] Chapters 1–4 in this book roughly correspond to what we covered. We did skip the basis representation theorem due to lack of relevance; generating functions due to difficulty; and combinatorics due to the fact that it's better suited for a dedicated exposition. (Hint, hint.)

George E. Andrews. *Number Theory*. Dover Publications, 1994. ISBN: 9780486682525

[2] Chapter 2 here is on direct proofs, but doubles as an extended example where the basics of number theory are developed, culminating with a proof to Fermat's Little Theorem. There's also a "bonus" section that goes into more detail on RSA.

Jay Cummings. *Proofs: A Long-form Mathematics Textbook*. LongFormMath.com, 2021. ISBN: 9798595265973

[3] Chapter 4 is on number theory. A few of the sections here cover some interesting material that we haven't included. Other chapters cover some other math relevant to computer science.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. 3rd ed. Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-55802-5

[4] Section 4.3.2 is on modular arithemtic, and starts with introducing residues and the Chinese Remainder Theorem. Our precise statement of the theorem (3.4) is based on the one here.

Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 9780201896848]

[5] This book gives a comprehensive overview of cryptographic techniques, including the mathematics and theoretical computer science background needed to understand them. (For instance, why the factoring problem makes RSA as secure as it is.) The proof to RSA's correctness in Page 10 is from Section 8.2.1 of this book.

Alfred J. Menezes et al. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 9781439821916

C Crash Course in C

C.1 Installation

On Windows, the least painful way for a beginner to get started with C is probably Visual Studio. Download the community edition here and run the installer. Then follow these steps to set up a C "project". (There's not much to it.) Once you have the `.c` file, you can just type your C code and press the F5 key to compile and run it.

On Linux, your distribution probably comes with GCC already. Supposing you have a file called `hello.c`, run the command "`gcc hello.c -o hello`". This will output an executable file `hello`, which you can run by typing "`./hello`". If you don't have GCC installed somehow, you'll have to do so using your distribution's package manager. Each one's different, so you'll have to search how to do it on the Internet.

For Mac, follow this guide: https://www.cs.auckland.ac.nz/~paul/C/Mac/

C.2 Basic Structure

Below is a C program that prints the message `Hello world!`.

```
#include<stdio.h>
int main() {
    // prints some text
    printf("Hello world!\n");
    return 0;
}
```

The `#include<stdio.h>` directive allows us to use certain standard "functions" which let us input and output text. The one we'll see most is `printf` for output, but occasionally we'll take input from the user using `scanf`.

Execution of the program starts inside of `main`. The line starting with `//` is a comment, which is written in English by the programmer to explain the code. Comments are ignored by the compiler.

We have two statements here. The first is `printf("Hello world!\n");`. What's happening here is that `printf` is being given an input `"Hello world!\n"`. Inputs to a function like `printf` are delimited by parentheses `(...)`, as you can see. In this case, the input is a string of characters; strings are delimited by double quotes `""`. The text in the string is obvious enough, but what about the `\n`? This is saying "start a new line here." The string ends after that in this case, but if you write some more text after the `\n`, like

```
printf("Hello world!\nHow are you?");
```

and run it, you'll get the below output:

```
Hello world!
How are you?
```

For the `return 0;` statement, we'll see that in C.6.

C.3 Data

There's a few important data types you need to keep in mind. The first is `int`, which is an integer. You create an integer like this:

```
int my_int;
```

The naïve explanation is that this will find a place in the computer's memory and label it `my_int`[10]. Whenever you use the name `my_int`, you'll be referring to whatever is stored in this spot in memory. For `int`, the size of this spot is 4 bytes, or 32 bits. Each bit can be a 0 or a 1, so a single `int` can store any one of $2^{32} = 4294967296$ possible values.

Obviously, you want to give a value to `my_int`. This can be done using the assignment operator, `=`. If you type `my_int = 5`, then the spot reserved in memory for `my_int` will contain the value `5`.

A shorthand for declaring and assigning a value to a variable is `int my_int = 5;`. (You can use other data types, names, and values, of course.)

`char` is another data type, and stores a single character. For example,

```
char letter = 'a';
```

Since each character has a corresponding integer value[11], `char` effectively boils down to just being a smaller `int`, but does have its uses. We'll cover those Section C.4.

`float` lets you store real numbers, or at least approximations to them. For example,

```
float x = 3.14159;
```

C.3.1 Arithmetic

This is probably the best place to fit in arithmetic. We have five major operators: `+`, `-`, `*`, `/`, and `%`. The first four are respectively addition, subtraction, multiplication and division; these work with integers and floats. The `%` is the remainder operator, and is exclusive to integer types. Check Section 2.1 for a proper coverage of that.

When dividing integers, the result will be another integer, so you won't always get an exact result. For instance, `17/5` gives you `3` rather than `3.4`.

C.3.2 Type Casting

You can convert between data types; this is known as type-casting. Here's an example.

```
float a = 3.14159;
int b = (int)3.14159;
```

Since data types' nature may differ, you get that type-casting often means you lose some data. For instance, `b` in the example loses the decimal points and contains 3. Usually type-casting to `int` is done for this oft-desirable property.

[10]This is called a declaration.
[11]Look up an ASCII table.

C.3.3 Arrays and Pointers

Your computer has several gigabytes of memory in its RAM. You can think of the RAM as an ordered list of bytes, starting at the 0'th byte and counting up. This gives an unambiguous way to refer to individual bytes, to access or change their values.

In C, you can define an *array* which has the same property. Here's an example:

```
int my_arr[5];
```

This finds space in memory for *five* integers in a single "block". That's 20 consecutive bytes for this array, since each integer is 4 bytes.

`my_arr[0]` refers to the first integer in this array, and `my_arr[4]` refers to the last one. (The number refers to the integers rather than the bytes.)

Like how there's a shorthand for declaring and initializing a variable, you can declare and initialize an array in one go:

```
int my_arr[5] = { 42, 4, 90, 64, 493 };
```

This is like doing `my_arr[0] = 42; my_arr[1] = 4`, etc.

The name `my_arr` by itself isn't one of these five integers. Rather, it's the memory address of the first integer. You can imagine `my_arr[3]` as doing the following:

1. Go to the memory address given by `my_arr`. This is the location of the first integer.
2. Skip forward by 12 bytes, thus skipping three integers (`my_arr[0]`, `my_arr[1]`, `my_arr[2}`).
3. We're now at the *location* of `my_arr[3]`, and can examine or change the value stored here.

Names like `my_arr` that refer to memory addresses are called *pointers*.

If you want to check the value pointed to by a pointer, you can use `*`. In the below example, since `my_arr` points to the memory address of the first element of the array, `*my_arr` must refer to the value stored at that address. That is, `*my_arr` is the same as saying `my_arr[0]`.

```
int my_arr[5] = { 42, 4, 90, 64, 493 };
*my_arr = 5; // same as my_arr[0] = 5
```

To declare a pointer, we can say `int* my_pointer = 5;`. This declares a pointer to an *integer*, specifically. Pointers are always the same size (64 bits, on a 64-bit machine) regardless of what they're pointing to; but you still need to specify what's being pointed to. This lets you perform pointer arithmetic, by referring not to individual bytes, but to individual integers (in this case):

```
*(my_arr+1) = 5; // same as my_arr[1] = 5
```

If you have a value, say `int x = 5;` and want to get its address, you can use the `&` operator. `&x` gives you an integer pointer containing the value of `x`. You can use this to define a pointer that points to `x`:

```
int* p = &x;
```

Pointers are usually fairly difficult to work with and have many pitfalls; but that's an outline of the basics.

C.4 Strings, Input, and Output

A string in C is simply an array of `char` values, terminated by the null character (denoted by `\0`). Here's an example:

```
char string[50] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

This kind of annoying to type, of course, so there's a shorthand:

```
char string[50] = "Hello";
```

The two statements are equivalent. You may worry that the size of the array is 50 so if we print `string`, fifty characters would be printed. This isn't a problem; the `\0` indicates when to stop when reading this string.

The `f` in `printf` means "formatted". This alludes to the fact that you can print strings in a nice, readable format. Usually, `printf` takes several inputs. The first is the string itself, which can contain text as well as certain placeholders. These placeholders will be replaced in order by the second, third, ... inputs of `printf`. It's best to illustrate this with an example:

```
int x = 5;
printf("The value of x is %d!!", x);
```

The portions "`The value of x is` " and "`!!`" are printed as-is. The `%` indicates a placeholder, and the `d` means that this placeholder is an integer. The placeholder will be replaced by the second argument of `printf`, in this case `x`; and we know `x` is 5, so the program will print the following:

```
The value of x is 5!!
```

There's a few other choices for placeholder, and you may also specify a "padding" so that no matter what is printed, it's guaranteed to take up some amount of characters. You can check an online reference for the details; there's no need to recount them here. However, we're likely to use the `%s` placeholder often, so I'll give an example of that.

```
char string[50] = "Hello world!"
printf("%s",string);
```

This prints `Hello world!`, as you can imagine.

For user input, we use `scanf`. It's fairly similar to `printf` in that the first argument is a string showing the format and containing placeholders. These placeholders mean what data type is read; the value is placed into the second/third/... argument of `scanf`. One key difference is that these later arguments are addresses. So if you want to read a value into an integer, you do the following:

```
int x;
scanf("%d",&x);
```

This will take in the user input and put it in `x`.

C.5 Control Structures

So far, our programs have been straightforward — execute one statement and move on to the next. If "tracing" the flow of the program, you need only place a finger on the first statement and move down. You know that the statements will execute in that order.

Control structures allow for the program to have a more sophisticated flow, where you will need to lift your finger on occasion in order to skip steps or go back to a previous step. The simplest method of control is the goto. Here's a program that repeatedly prints A:

```
#include<stdio.h>
int main() {
    printf("I'm going to scream: ");
mylabel:
    printf("A");
    goto mylabel;
}
```

mylabel is, as the name suggests, a label. By itself it doesn't do anything, and simply marks a specific location in the program. The goto mylabel; statement means to go back to mylabel and continue execution from there.

The next structure is the if statement. This lets us evaluate some condition, and if it is true, execute a statement (or series of statements). Here's an example.

```
#include<stdio.h>
    int main() {
    printf("Input an integer: ");
    int x;
    scanf("%d",&x);
    if (x % 2 == 0)
        printf("Your integer is even.");
    else
        printf("Your integer is odd.");
}
```

Here we introduce a new operator, the ==. This checks to see if the left side is equal to the right side. If it isn't equal, the expression evaluates to 0. Otherwise, it evaluates to 1. The if executes the following statement whenever the value within the parentheses is not 0. In other words, if we had (for instance) written if(1) (or any other nonzero value), the statement would *always* be executed.

The statement following the else executes only if the condition in the preceding if is false.

Besides the ==, there are other operators. != checks for inequality. > checks if the value in the left is greater than the value on the right; and vice versa for <. >= is "greater than or equal to", and <= is "less than or equal to".

Furthermore, we have three other operators that act on expressions, allowing us to compose larger ones from smaller ones. The first is && means "and", and can be used in this manner:

```
if ((x > 10) && (x <= 15))
```

This will check if x is *both* greater than 10, *and* less than or equal to 15; only in that case will it return a 1. The "or" operator `||` works the same way, but returns a 1 whenever at least one of the operands is 1. The third operator is the "not" operator `!`, which inverts the value of the following expression; if it's a 1, it'll become a 0, and vice versa. For instance,

```
!(x > 10)
```

is true if and only if $x \leqslant 10$.

`if` and `else` only execute the following statement. If you want to have multiple statements be executed if a condition is (not) true, then you can wrap them in braces:

```
if (condition) {
    // statement 1;
    // statement 2;
    // ...
}
```

Truth be told, `goto` and `if` are enough to implement the bigger control structures (loops). However, those structures do offer some syntactic sugar, meaning they make programs easier to reason about and write. Suppose you want to repeat a set of statements as long as some condition is true. With `goto` and `if`, you can do the following.

```
loop:
    // your statement(s) here...
    if (condition) goto loop;
```

This is a little tedious, so there's a shorthand for it in the form of `while` loops:

```
while (condition)
    // your statement here...
```

Like with `if`, you can wrap multiple statements in braces to have them all be looped.

We'll cover one other form, which is the `for` loop. Typically, this is used when you know how how many times you need to repeat a set of statements. The part in the parentheses is more complicated, so we'll illustrate with an example:

```
for (int i = 1; i <= 10; i = i + 1) {
    // statements
}
```

Notice the two semicolons. The part before the first semicolon is executed once only. The second part is executed at the start of each iteration. The last part is executed at the end of each iteration. A `for` loop is more or less equivalent to either of the following constructs.

```
int i = 1;
while (i <= 10) {
    // statements
    i = i + 1;
}
```

```
    int i = 1;
cond:
    if (i <= 10) goto loop;
    else goto endloop;
loop:
    // statements
    i = i + 1;
endloop:
    // rest of program
```

C.6 Functions

Programming would be a lot more tedious and repetitive if you couldn't bundle your code together into bigger pieces — into units composed of several statements that together perform a single specific task. These units are called functions. You can write the code for a function once, and later on *call* this function to perform a task; usually giving it some inputs, and then taking an output from it to use elsewhere.

Here's an example of a function:

```
void say_hi(int x) {
    printf("Hello!\nYour number is %d",x);
}
```

The `void` keyword means that this function has no output. It takes in a single input of type `int`, and performs some task with it (in this case printing it inside of a string).

We may use this function by writing `say_hi(5)`, for instance. As long as what's within the parentheses is a single integer, there are no problems.

Here's a function that returns the square of an integer.

```
int square(int x) {
    return x * x;
}
```

Now we're doing something different; we aren't printing anything, but are instead *returning* a value. When using this, we simply say `square(5)`, but that's not enough if we want to *use* the value in some way. Here are two ways we can use it.

```
int four_squared = square(4);

int input;
printf("Input a number: ");
scanf("%d, &input);
printf("The square of %d is %d",input,square(input));
```

C.6.1 Calls by Reference, Calls by Value

In the above examples, the inputs given to the functions created a new "local" copy of the variable. If this variable were to be altered, the original one would not be affected. See this example:

```
void double(int x) {
    x = 2 * x;
}
int main() {
int input = 5;
double(input);
printf("%d",input);
}
```

This would print 5 rather than 10, because the call to `double` creates a copy of `input` called `x`. We could've named it `input` rather than `x`, and nothing would change. The key thing is that this copy starts with the same value as `input` does. But after this initialization, it has no relation to `input`; doubling `x` would not double `input`. This is called passing by value.

Now consider this example.

```
void double(int &x) {
    x = 2 * x;
}
int main() {
    int input = 5;
    double(input);
    printf("%d",input);
}
```

This would print 10. The only difference is that we used `&x` rather than `x` in the parameter. This means that when passing a variable, its location will be given, so that `x` refers to the original variable rather than a copy.

This concludes the introduction to C.