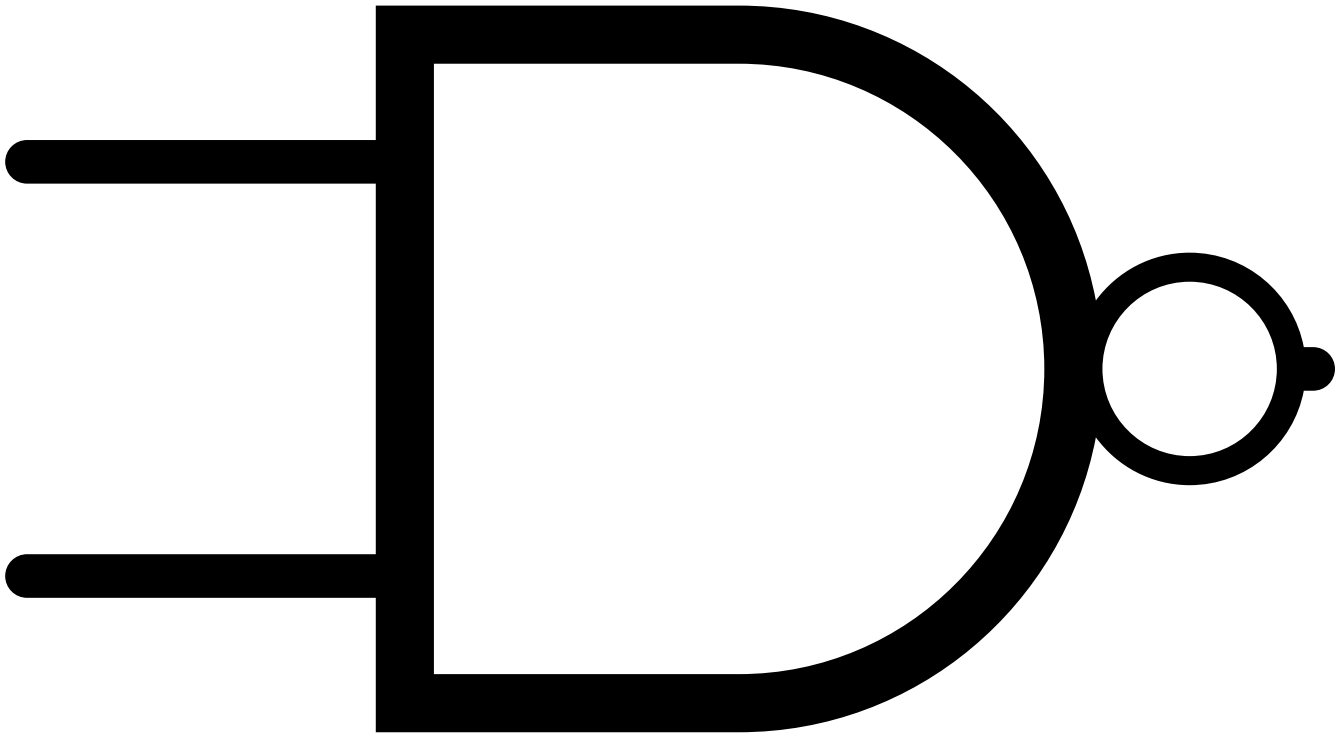
UMBC
Training Centers
6996 Columbia Gateway Drive
Suite 100
Columbia, MD 21045
Tel: 443-692-6600
http://www.umbctraining.com

Discrete Math & Version Control



Course # TCPRG4003

Rev. 05/14/2021

This page intentionally left blank.

Contents

But people have always been dimly aware of the problem with the start of things. They wonder how the snowplough driver gets to work, or how the makers of dictionaries look up the spelling of words.

Terry Pratchett, *Hogfather*

Chapter 1

Introduction

Objectives

This course is designed to give a novice the mathematical tools they need to become a programmer. It is not exhaustive, nor is it overly concerned with mathematical rigor. It is also built to introduce the most important tool of *version control* to a new developer. Upon completion, a graduate should be able to do the following tasks:

- Prove the equality of two propositions using truth tables
- Manipulate logarithms
- Manipulate mathematical sets
- Count in binary and hexadecimal
- View the history of a project in `git`
- Create commits in `git`
- Create and switch between branches in `git`
- Clone and create new projects in `git`

Contents

Prerequisites

This material assumes familiarity with arithmetic, and the POSIX command line for working with git. Required tools are enumerated in Chapter 2.

A beginning student should be able to navigate the file system, create, edit, and remove files using the command line.

History

While the history of Mathematics is older than civilization[1], mathematics as it applies to Computer Science is much more recent. While the foundations were laid in the 6th century BCE by the Greeks Thales and Pythagoras, most of the discoveries did not begin until the 18th century CE, and the crowning capstone of the Church-Turing Hypothesis was proven only in the 1930s.

As such, the limits of computer science and its mathematics are still being explored.

`git` was developed in 2005 by Linus Torvalds, creator of the Linux OS kernel. This was in response to licensing issues regarding the current tool that was used to manage the Linux kernel. Within a short time, it became the dominant version control system used by programmers and developers.

While this course may be specific to `git`, all of the concepts of version control are knowledge-transferrable to other version control systems.

[1] The oldest mathematical artifacts date to 20,000BCE

This page intentionally left blank.

Mathematics is the second-cheapest profession, requiring only paper and pencil and a wastebasket (I'll leave unnamed the profession that makes do without the wastebasket).

George Pôlya

Chapter 2

Tools

Paper, Pencil, and Wastebasket

Sketching out possible solutions and attempting to draw logical connections requires fluid, erasable thought. Be prepared to throw away work that may be effort directed down the wrong path.

Despite this being an opening course for computer science, very little is needed of computers for the mathematical portion. It is more concerned with relationships between sets and numbers than pure calculation.

Calculator

A calculator that is capable of calculating x^y, e^y, and $\log x$ will be essential to performing certain algebraic calculations. A computer desktop calculator should be sufficient.

`git`

`git` does not require an Internet connection, GitHub account, or a GUI. A terminal is all that is necessary. This course does assume that the student has command-line literacy, specifically with a POSIX shell.

GitHub is an extremely useful collaboration tool, but is not required. It is a third-party service than can be very useful for a team of developers or authors.

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

Lewis Carroll, *Through the Looking-Glass*

CHAPTER 3

Logic

Equality

The equality symbol, $=$, means for the terms on the left and right hand side to be *the same*. Not that they are similar or equivalent, but the exact same value. Equality is a mathematical guarantee that the two sides are the same in all respects save perhaps the symbols used to represent them. There is only one "number 6", while it may be written as 6 or $1 + 5$ or $\sqrt[3]{216}$ or VI, it is still the same number.

In mathematical terms, that means that the left-hand side can be substituted for the right-hand side in any other expression, and vice versa. $1 + 8 \times 2 = 3 \times 6 - 1$ implies that wherever $1 + 8 \times 2$ appears, the mathematical phrase $3 \times 6 - 1$ could be written instead. It is vital in mathematics that the exact phrase be substituted, usually with parentheses.

Low-level mathematics is a "substitution game" like those equations. Only the exact phrases shown to be equal may be substituted, and that equality is found through rigorous or exhaustive proof.

Propositions

A **proposition** is a statement that is true or false, exclusively. It does not matter whether the statement is true or false, only that it is in one of those two states. Some examples of propositions are:

- 91 is divisible by 7.
- Today is Thursday.
- It is raining outside.
- George Washington is the president of Colombia.

The following sentences are *not* propositions:

- Is it Thursday?
- $x^2 - 5x + 9$
- Eat your beans.
- This statement is false.

This course only deals with propositions. Other forms of statements that are not propositions will not be considered.[1]

[1] The non-propositions shown are a question, an equation, a command, and a paradox.

Conjunction

Consider the statement "Today is Thursday, and it is raining.".

It is a proposition; it can be true or false. Note that, by the English understanding of the phrase, it would only be true if both today is Thursday, and it is raining.

Conjunction: Between two propositions P and Q, the new proposition $P \wedge Q$, which is true if and only if both P and Q are true. This is also called an "AND" statement, as it is read as "P and Q."

It is important to note that the AND statement is, itself, a proposition. Since AND works on propositions, and itself produces a proposition, that means that additional AND statements can be chained together, two at a time.

"(Today is Thursday, and it is raining), and George Washington is the president of Colombia."

Truth Tables

Early proofs are best solved via **truth tables**: a table that exhaustively lists every possible combination of input values. If any two columns in the table are identical, then those two statements are equal or equivalent.

Here is an example of the truth table for the AND statement:

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Note that both middle rows are present. There will be certain operations that do not work the same in either direction, so it is important that the truth table carries both cases. For example, $8 - 4 \neq 4 - 8$

To show that $P \wedge Q = Q \wedge P$, the following truth table is constructed:

P	Q	$P \wedge Q$	$Q \wedge P$
T	T	T	T
T	F	F	F
F	T	F	F
F	F	F	F

Since the two columns are the same, $P \wedge Q = Q \wedge P$. As such, $P \wedge Q$ can be substituted for $Q \wedge P$.

Disjunction

Disjunction: Of two propositions P and Q, the new proposition $P \vee Q$, which is false if and only if both P and Q are false. This is also called an "OR" statement, as it is read as "P or Q."

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

This is somewhat unintuitive compared to the English language version. The statement "It is raining or it is Thursday," tends to carry the implication that *only* one of the constituent statements is true. In logic, however, both statements may also be true to yield a true disjunction.

Negation

Unlike the other operations seen so far, **negation** involves only a single proposition, not two.

Negation: Of a propositions P, the new proposition $\neg P$, which is false if and only if P is true. This is also called an "NOT" statement, as it is read as "not P."

P	$\neg P$
T	F
F	T

It is easy to fall into the trap of improperly negating an English sentence. The negation of "Today is Thursday," is *not* "Today is Friday," or "Yesterday was Thursday," but rather "Today is not Thursday."

Prefixing "it is not the case that" in front of an English sentence will generally do the correct job of negating it.

Tautology and Contradiction

Using the operations introduced so far, it is possible to create propositions that are true no matter what. These propositions are very useful, as they allow derivation of additional rules of logic.

"Today is Thursday or today is not Thursday."

Tautology: A proposition that is always true, regardless of its constituent propositions.

P	$\neg P$	$P \vee \neg P$
T	F	T
F	T	T

It is also possible to create propositions that are false no matter what. These **contradictions** are useful in certain proofs, where the result of a contradiction from an initial assumption shows that the assumption must be false.

"Today is Thursday and today is not Thursday."

Contradiction: A proposition that is always false, regardless of its constituent propositions.

P	$\neg P$	$P \wedge \neg P$
T	F	F
F	T	F

Conditional and Biconditional

Logical proofs often require the use of chains of logical deduction. One of the most useful tools towards this goal is a variant of an OR statement.

"If George Washington is the president of Colombia, then 91 is divisible by 7."

Conditional: Between two propositions P and Q, the new proposition $P \implies Q$, which is false if and only if both P is true and Q is false. This is also called an "IF" statement, as it is read as "if P then Q."

P	Q	$P \implies Q$
T	T	T
T	F	F
F	T	T
F	F	T

Note that it *does not matter* that a conditional's two parts be related. The only arbiter of truth in a conditional is that the if-part is true and the then-part is false.

There is a logical form of equality that is a shorthand for two conditionals pointing in different directions.

"Today is Thursday if and only if it is raining."

Biconditional: Between two propositions P and Q, the new proposition $P \iff Q$, which is false only if P and Q have different truth

values. This is also called an “IFF” statement, short for “if and only if”.

P	Q	$P \iff Q$
T	T	T
T	F	F
F	T	F
F	F	T

The biconditional is equivalent to the following conjuction of conditionals:

P	Q	$P \implies Q$	$Q \implies P$	$P \implies Q \wedge Q \implies P$	$P \iff Q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	T	F	F	F
F	F	T	T	T	T

Exclusive Or

The implied English-language version of the OR statement is worth considering:

"Either George Washington is the president of Colombia, or 91 is divisible by 7."

Here, the implication is that one of the two propositions is true, but not *both*.

Exclusive Disjunction: Between two propositions P and Q, the new proposition $P \oplus Q$, which is true if and only if both P and Q have different truth values. This is also called an "XOR" statement, as it is read as "P exclusive-or Q."

P	Q	$P \oplus Q$
T	T	F
T	F	T
F	T	T
F	F	F

Arithmetic Logic Notation

Sometimes, in the interest of space or expediency, it can be helpful to reduce the number of symbols used when writing propositions. This is done by removing the $\wedge$ symbol entirely, using $+$ instead of $\vee$, and using a postfix $'$ instead of a prefix $\neg$.

$$P \wedge (Q \vee \neg R) = P(Q + R')$$

This notation is very handy as it reflects the parallel arithmetic meaning, if truth uses the value 1 and false uses the value 0. Conjunction looks like multiplication, which matches the logical AND table. Disjunction yields a 0 value only if all its inputs are 0.

Exercises

1. Circle all following statements that are propositions.

   (a) $2 + 7 = 9$

   (b) The Queen is secretly a lizard person.

   (c) $2 + 2 = 17$

   (d) $5x + 3 = 8$

   (e) Read the book *Catch 22*.

   (f) It is not the case that the Queen is secretly a lizard person.

   (g) *Single Ladies* is the best music video of all time.

   (h) No lizard people were unmasked by the publication of this book.

   (i) Taylor Swift thinks *Single Ladies* is the best music video of all time.

   (j) This statement is a proposition.

2. Mark the following statements as true or false.

   (a) Ben Franklin is the president of Peru.

   (b) Paris is the capital of France.

   (c) Today is Sunday.

   (d) Paris is the capital of France and Ben Franklin is the president of Peru.

   (e) Ben Franklin is the president of Peru or Paris is the capital of France.

(f) If then Paris is the capital of France, then Ben Franklin is the president of Peru.

(g) If Ben Franklin is the president of Peru, then Ben Franklin is the president of Peru.

(h) Paris is the capital of France if and only if today is Sunday.

(i) Ben Franklin is the president of Peru if and only if today is Sunday.

(j) If it is Sunday, then Ben Franklin is the president of Peru.

(k) Ben Franklin is the president of Peru or Ben Franklin is the president of Ecuador.

3. Are the following propositions tautologies, contradictions, or neither?

(a) $P \wedge \neg P$

(b) $P \vee \neg P$

(c) $(P \vee \neg Q) \implies Q$

(d) $(P \vee Q) \implies (P \wedge Q)$

(e) $(P \implies Q) \iff (\neg Q \implies \neg P)$

(f) $(P \implies Q) \implies (Q \implies P)$

4. Prove the following tautologies using truth tables.

(a) $P \implies Q = \neg Q \implies \neg P$

(b) $P \vee Q = Q \vee P$

(c) $P \wedge Q = Q \wedge P$

(d) $(P \vee Q) \vee R = P \vee (Q \vee R)$

(e) $(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$

(f) $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$

(g) $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$

(h) $P \vee (Q \implies R) = (P \vee Q) \implies (P \vee R)$

(i) $\neg(P \vee Q) = \neg P \wedge \neg Q$

(j) $\neg(P \wedge Q) = \neg P \vee \neg Q$

(k) $P = \neg\neg P$

(l) $P \implies P$

(m) $P \iff P$

(n) $Q \implies (P \implies Q)$

(o) $(P \wedge Q) \implies (P \vee Q)$

5. Construct truth tables for the following propositions.

(a) $\neg(\neg P \implies P)$

(b) $(\neg P \implies P) \iff P$

(c) $(P \implies Q) \vee (Q \implies P)$

(d) $(P \implies Q) \implies (Q \implies P)$

(e) $(P \vee Q) \implies (Q \iff P)$

(f) $((P \wedge \neg Q) \implies Q) \implies \neg P$

(g) $P \wedge ((P \implies Q) \wedge (P \implies \neg Q))$

(h) $(P \vee Q) \iff \neg(\neg P \wedge \neg Q)$

(i) $(P \wedge Q) \implies (P \implies Q)$

(j) $(Q \vee R) \implies ((R \implies P) \vee (Q \implies P))$

(k) $(Q \vee R) \implies ((P \implies Q) \vee (P \implies R))$

(l) $(P \wedge \neg(Q \vee R)) \iff \neg(P \implies Q)$

6. An island's population is made up of the Tutus, who always tell the truth, and Fufus, who always lie. In a number of distinct instances listed below, you meet two islanders, A and B. Determine which group each islander is from based on the statement you received.

   (a) A says "I am a Fufu but B is not."

   (b) A says "I am a Fufu or B is a Tutu."

   (c) A says "It is not the case that B being a Fufu implies I am a Tutu."

7. Let P be the proposition "Rasputin is alive" and Q be the proposition "Anastasia is dead." Translate the following logical statements into English.

   (a) $P \implies Q$

   (b) $\neg Q \implies \neg P$

   (c) $Q \vee \neg P$

The Analytical Engine weaves algebraical patterns just as the Jacquard-loom weaves flowers and leaves.

Ada Byron King, Duchess of Lovelace

Chapter 4

Algebra

Exponents

Multiplication is shorthand for repeated addition.

$3 + 3 + 3 + 3 + 3 = 3 \times 5$

Exponents are shorthand for repeated multiplication. In the following example, 3 is the **base**, and 5 is the **exponent**.

$3 \times 3 \times 3 \times 3 \times 3 = 3^5$

Multiplication is **commutative**, which is another way of saying that multiplying two numbers is the same no matter which one comes first: $3 \times 5 = 5 \times 3$. This is not the case with exponentiation.

$$2^3 \neq 3^2$$
$$2 \times 2 \times 2 \neq 3 \times 3$$

Exponentiation has a number of identities.

$$x^a x^b = x^{(a+b)} \qquad x^a y^a = (xy)^a$$
$$(x^a)^b = x^{(ab)} \qquad x^{(a-b)} = \frac{x^a}{x^b}$$
$$x^{\frac{a}{b}} = \sqrt[b]{x^a} = \sqrt[b]{x}^a \qquad x^{-a} = \frac{1}{x^a}$$

Logarithms

Given a known value of exponent, a inverse operation known as a root can be defined:

$$\begin{aligned} x^5 &= 243 \\ \sqrt[5]{x^5} &= \sqrt[5]{243} \\ x &= \sqrt[5]{243} \\ x &= 3 \end{aligned}$$

For an unknown value of exponent, the inverse operation is the **logarithm**:

$$\begin{aligned} 3^n &= 243 \\ \log_3 3^n &= \log_3 243 \\ n &= \log_3 243 \\ n &= 5 \end{aligned}$$

Just like taking a root requires knowing what exponent is being inverted (the Nth root), so too does taking a logarithm requires knowing what base is being inverted. This is called the **base** of the logarithm.

Logarithms have a lot of useful mathematical properties.

$$\log_b 1 = 0 \qquad \log_b \frac{x}{y} = \log_b x - \log_b y$$
$$\log_b b = 1 \qquad \log_b(xy) = \log_b x + \log_b y$$
$$\log_b x^n = n \log_b x \qquad \log_k x = \frac{\log_b x}{\log_b k}$$

To convert a logarithm from some base k to a different base b:

$$\log_k x = \frac{\log_b x}{\log_b k}$$

This identity is useful because it allows easy moving between logarithm bases. There are three bases of interest.

$\log_{10}$ is the base-ten logarithm.

$\log_2$ is the base-two logarithm.

$\log_e$ or $\ln$ is the base-e logarithm, where $e \approx 2.718281828$[1]

When talking about roots, the radical $\sqrt{}$ by itself implies the square root. When discussing logarithms, the function $\log$ by itself tends to imply the base-two logarithm when programming[2].

Logarithms roughly function as an "order-of-magnitude" function that

[1] This number is very important to mathematicians, and it pops up in mathematics all over the place.

[2] In mathematics, base-e is implied, and in engineering, base-ten

indicate the number of digits (less one) for a number in the given numeric base.

$$\log_{10} 1000 = 3$$
$$\log_2 256 = 8$$
$$\log_7 49 = 2$$

Factorials

How many ways are there to arrange three marbles in a row? Consider that the first-positioned marble has three possible choices. After that, the second position would only have two possible remaining choices, and then the final position has its marble already determined. By multiplying these values together, the number of arrangements is found, which is six.

$$3 \times 2 \times 1 = 6$$

R G B B R G G R B
R B G B G R G B R

This sort of operation, of multiplying numbers that descend down to 1, is known as a **factorial** operation, and is written $3!$. The factorial function grows very quickly:

3!	6
4!	24
5!	120
6!	720
7!	5040
8!	40320

This factorial is the number of ways to arrange or permute n objects, as illustrated earlier. So, $1! = 1$, as there is only one way to arrange one object. Similarly but confusingly, there is also only one way to arrange zero objects, which means $0! = 1$

Modulo Arithmetic

When dividing numbers, it is possible to end up with a decimal value.

$$17 \div 4 = 4.25$$

However, it also possible to express the result as a remainder.

$$17 \div 4 = 4\text{R}1$$

The **modulo operation** between two integer values is an operation that yields the remainder between the two. The programming symbol for this binary operation is the percent sign (%)[3].

$$17 \,\%\, 4 = 1$$

Since computers tend to perform better using only integers, having a remainder operation like the modulo makes it easy to to division using only integer inputs and outputs.

It is important to note that $a \,\%\, b \neq b \,\%\, a$.

This modulo arithmetic is similar to how 12-hour clocks work: 6 hours after 8:00 is not 14:00 but 2:00. All operations are performed modulo-12.

[3] Mathematicians tend to use the word "mod", such as $17 \mod 4 = 1$

Completely Pedantic Aside

In computing, the terms “remainder” and “modulo” are used interchangably. This is something that mathematicians may make more distinct from each other, in that “modulo” is always a nonnegative number, while “remainder” may be a negative value.

$$-17 \,\%\, 4 = -1$$

$$-17 \mod 4 = 3$$

Exercises

1. Calculate the following values using a calculator:

   (a) $\log_{10} 98$

   (b) $\ln 45$

   (c) $\log_2 1024$

   (d) $\log_{10} 1024$

   (e) $\ln 1024$

   (f) $\log_5 27$

   (g) $\log_9 982383$

2. Simplify the following using the various logarithm rules.

   (a) $\log_3 9x^4 - \log_3(3x)^2$

   (b) $\ln e^2$

   (c) $\log_2 \frac{8x^2}{y} + \log_2 2xy$

   (d) $\ln(e^2 \ln e^3)$

3. Calculate the following values.

   (a) $10!$

   (b) $\frac{10!}{9!}$

   (c) $\frac{10!}{5!}$

   (d) $\frac{6!}{(6-3)!3!}$

4. Solve the following for the unknown variable.

   (a) $\log_3 x = 4$

   (b) $\log_m 81 = 4$

   (c) $\log_z 1000 = 3$

   (d) $\log_2 \frac{x}{2} = 5$

   (e) $\log_3 y = 5$

   (f) $\log_2 4x = 5$

5. Solve the following for x.

   (a) $2\log_b 4 + \log_b 5 - \log_b 10 = \log_b x$

   (b) $\log_b 26 - \log_b 52 = \log_b x$

   (c) $\log 8 + \log x^2 = \log x$

When you have mastered numbers, you will in fact no longer be reading numbers, any more than you read words when reading books. You will be reading meanings.

W. E. B. Du Bois

Chapter 5

Numbers

Classifications

It is useful to subdivide numbers into different classifications, as these subgroups can have useful properties. In computer science, certain groups of these have features that make them amenable to storage and computation.

Natural Numbers

The **natural numbers**, whole numbers, or counting numbers, are increasing numbers starting with 0: $0, 1, 2, 3, 4, \ldots$. No fractional or negative values are allowed[1]. The natural numbers are often referred to using the symbol $\mathbb{N}$.

Integers

Integers are the natural numbers extended to negative values. -5, 5, 0, and $-1,928,739$ are all examples of integers. Integers do not have fractional parts. Note that every natural number is also an integer. The set of all integers is referred to using the symbol $\mathbb{Z}$, which stands for *zahlen*, the German word for 'number'.

[1] Some overeager teachers may suggest that there exists a sub-category known as the **whole numbers**, consisting of the natural numbers sans 0. This subset is entirely irrelevant.

Rational Numbers

The **rational numbers** are any numbers that can be expressed as a fraction of two integers. Note that all integers are also rational numbers, such as 6/3 or 8/1. Other examples of rational numbers include -1/2, 17/3, 3.989, and 0/87.

Any number with a finite number of decimal digits will always be a rational number. As an example, 3.98 can be represented as 398/100. Also, if a number's decimal digits have a repeating pattern, such as $0.333\ldots$, it is also going to be a rational number.

To be exactingly specific, the denominator of the fraction making up a rational number is not allowed to be 0. The set of all rational numbers (which obviously includes all integers) is referred to using the symbol $\mathbb{Q}$, which stands for 'quotient', the result of a division.

Real Numbers

The **real numbers** include all representable numbers, such as roots, infinite decimal points, irrational numbers, and the like. $\sqrt{2}$ and π are examples of real numbers that are not rational. Note that numbers with repeating decimals, such as $0.142857142857\ldots$ are more specifically rational numbers, in this case 1/7. All rational numbers are real numbers.

The set of real numbers is referred to using $\mathbb{R}$.

Prime Numbers

A **prime number** is a special type of natural number greater than one that is only divisible by 1 or itself[2]. The first few prime numbers are:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, \ldots$$

There are an infinite amount of prime numbers, just as there are infinitely many natural numbers. Any natural number that is not prime is called a **composite number**. Prime numbers are very important in programming, used for encryption.

Every single natural number has **exactly one prime factorization**, or collection of prime numbers raised to powers that, when multiplied together, that yield that number. This property is important enough that it is given the title **Fundamental Theorem of Arithmetic**.

$$1999 = 1999$$

(is prime)

$$2000 = 2^4 \times 5^3$$

$$2001 = 3 \times 23 \times 29$$

This **Unique Prime Factorization** can be used to distinguish when two numbers have factors in common. As an example:

[2] 0 and 1 are excluded from the definition intentionally

$$2^n \mod 3 \neq 0$$

Bases

A number can be written in a variety of different ways, without changing the underlying value. The number 6 can be written as VI, ||||||, or six. These different representations do not change the '6'-ness of the number.

$$\mathrm{VI} + \mathrm{IV} = \mathrm{X}$$
$$|||||| + |||| = ||||||||||$$
$$\text{six + four} = \text{ten}$$
$$6 + 4 = 10$$
$$6 + 4 = 12_8$$

With 10 fingers on our hands, the **base** of our number system is ten.

In a positional number system, each digit stands for a different value based on its place in the number. In '427', the digit 4 represents a larger value (four hundred) than the digit 7 (seven), despite the fact that $4 < 7$. The value of each place increases by a factor of 10 as each digit is moved to the left.

$$\begin{aligned} 427 &= 400 + 20 + 7 \\ &= (4 \times 10 \times 10) + (2 \times 10) + (7) \end{aligned}$$

However, the phrase "factor of 10" fails to make sense unless a positional number system is already assumed! Instead, try expressing it as

the word “ten”.

$$\begin{aligned} 427 &= 400 + 20 + 7 \\ &= (4 \times \text{ten} \times \text{ten}) + (2 \times \text{ten}) + (7) \end{aligned}$$

Using a number other than “ten” results in a number system with that base. For clarity, the base of a number is often written as a subscript. With no subscript, base-ten is assumed. A single digit will be the same in any base.

$$\begin{aligned} 427_8 &= 400_8 + 20_8 + 7 \\ &= (4 \times \text{eight} \times \text{eight}) + (2 \times \text{eight}) + (7) \\ &= (256_{10}) + (16_{10}) + 7 \\ &= 279_{10} \end{aligned}$$

Note that $427_8 = 279_{10}$. That means that these two representations are the **same value**. There is no “conversion” or “difference” between the two; they are the same number because they have been shown to be equal. Numbers written in different bases are just different representations of the same value.

Different bases use different sets of digits. For instance, in base-eight, it makes no sense to talk about the digits 8 or 9: $7_8 + 1_8 = 10_8$

Similarly, for a base greater than ten, new digits are needed. Alphabetic letters tend to be used (A = ten, B = eleven, C = twelve, etc.): $9_{16} + 2_{16} = \text{B}_{16}$

In computer science, there are three important bases in addition to base-ten. These bases are often indicated with prefixes rather than subscripts when programming.

1. base-two, prefixed with 0b: 0b101101
2. base-eight, prefixed with 0 (sometimes 0o): 055 or 0o55
3. base-sixteen, prefixed with 0x: 0x2D

Binary

Base-two numbering, also called **binary**, is an important number system to programming. It uses only two digits, 0 and 1, to represent all possible numbers. Even though people will say "binary numbers," remember that this really means "binary-formatted numbers." The underlying number is still the same value.

$$\begin{aligned} \text{0b00111101} = &(1 \times \text{two} \times \text{two} \times \text{two} \times \text{two} \times \text{two}) \\ &+ (1 \times \text{two} \times \text{two} \times \text{two} \times \text{two}) \\ &+ (1 \times \text{two} \times \text{two} \times \text{two}) \\ &+ (1 \times \text{two} \times \text{two}) \\ &+ 1 \\ &= 61 \end{aligned}$$

Binary is a useful base because it can be encoded easily in hardware,

using the presence of a current or voltage to represent 1, and the lack of such to represent 0.

Binary	Decimal
0b0001	1
0b0010	2
0b0011	3
0b0100	4
0b0101	5
0b0110	6
0b0111	7
0b1000	8

Hexadecimal

Base-sixteen numbering, called **hexadecimal** or **sexagesimal**, is a high-level way of referring to numbers during programming that maps easily onto binary encoding. It uses all ten digits, 0-9, as well as the letters A-F for the values ten through fifteen, respectively. Even though people will say "hexadecimal numbers," remember that this really means "hexadecimal-formatted numbers." The underlying number is still the same value.

Binary takes many digits to represent a number, but that number can be encoded much more efficiently (four times more efficiently) in hexadecimal.

$$\begin{aligned} \texttt{0x3D} &= (3 \times \text{sixteen}) + \texttt{0xD} \\ &= \texttt{0b00111101} \\ &= 61 \end{aligned}$$

Hexadecimal is now the lingua franca of numbers in programming. Note that each quartet of binary numbers corresponds one-to-one with a hexadecimal digit:

Hexadecimal	Binary
0x7	0b0111
0x8	0b1000
0x9	0b1001
0xA	0b1010
0xB	0b1011
0xC	0b1100
0xD	0b1101
0xE	0b1110
0xF	0b1111

This correspondence does not hold true for decimal and hexadecimal, which is why many programmers “talk” in hexadecimal when it comes to low-level discussion of the computer (memory addresses, sizes, etc.). The correspondence is why hexadecimal is used!

Hexadecimal	Decimal
0x08	8
0x09	9
0x0A	10
0x0B	11
0x0C	12
0x0D	13
0x0E	14
0x0F	15
0x10	16
0x11	17
0x12	18

Octal

Base-eight numbering, also known as **octal**, is no longer frequently used in programming. On older 12-bit systems[3], octal was a useful representation of the contents of a word, using three octal digits per word.

In modern-day systems that use 8 bits in a byte, octal does not fit well and has thus fallen out of favor. It is still used in some cases, such as file permissions, but on the whole is less used than binary or hexadecimal. It is important to note that because octal is indicated only with a leading zero, it is risky to use leading 0s on any decimal numbers.

[3] such as the PDP-8

Octal	Decimal
005	5
006	6
007	7
010	8
011	9
012	10
013	11

Radix Economy

Which base is the best to use? It really comes down to communication. When expressing a value where the specific bits are important, binary makes the most sense. When expressing a location in computer memory, hexadecimal makes more sense, as it can encode a binary number with fewer digits. And most things involving the real world should be in decimal, as that is what humans are used to.

It is easy to see that a value that can be compactly represented in hexadecimal, such as $0xFACE$, would take many more digits in binary: $0b1111101011001110$. But, the more compact representation requires a larger variety of digits to choose from, complicating the number system. What base would be the most efficient in terms of expressing numbers versus needing distinct digits? This can be measured using **radix economy**, the the number of digits multiplied by the base.

$$E(b, N) = b\lfloor \log_b(N) + 1 \rfloor$$

Here, expressing the number N the base b yields a particular radix economy. Consider the radix economy of the first few numbers (all numbers written in base-ten):

Note how binary is consistently better in efficiency than the larger bases. Interestingly, base-three (**ternary**) is actually more efficient than binary by a small amount, when averaged over various ranges of numbers.

In fact, the base that would be most efficient when measuring radix economy would be e.

$$E(b, N) = b\lfloor \log_b(N) + 1 \rfloor \approx b \log_b(N) = b\frac{\ln N}{\ln b} = \ln N \frac{b}{\ln b}$$

And this number is minimum when $\frac{b}{\ln b}$ is minimum, which occurs at e. As ternary is the closest integer base to e, it has the best radix economy of integers[4].

[4]Thus, in theory, a computer built in ternary rather than binary would be cheaper and faster due to using fewer circuits and components. The one known example of a (balanced) ternary computer, the Сетунь, was manufactured between 1959 and 1965. Its binary replacement ran as quickly, but at 2.5 times the cost.

Number	E(10)	E(2)	E(16)	E(3)
1	10	2	16	3
2	10	4	16	3
3	10	4	16	6
4	10	6	16	6
5	10	6	16	6
6	10	6	16	6
7	10	6	16	6
8	10	8	16	6
9	10	8	16	9
10	20	8	16	9
11	20	8	16	9
12	20	8	16	9
13	20	8	16	9
14	20	8	16	9
15	20	8	16	9
16	20	10	32	9
17	20	10	32	9
18	20	10	32	9
19	20	10	32	9
20	20	10	32	9
21	20	10	32	9
22	20	10	32	9
23	20	10	32	9
24	20	10	32	9
25	20	10	32	9
26	20	10	32	9

Binary Operations

All numbers are ultimately stored in binary in a computer. A collection of circuits are storing either “on” or “off” values, representing the 1 and 0 of binary digits. So, all values on a computer are eventually stored in binary form. It is said that these values or data are encoded in binary.

This means that mathematical binary operations are critical to how a computer works. Thankfully, since there are only two digits, all of these binary operations can be fully enumerated easily.

Since the underlying value is the same in binary, hexadecimal, or decimal, only the efficiency of the computer and its storage need be considered when it comes to the representations of values. The computer can translate from its internal binary representation to an external decimal one when required. However, it is necessary that programmers understand the underlying binary logic that drives the computer.

These operations and algorithms do not require a full understanding of binary other than addition. Standard mathematical algorithms will work perfectly well in binary just as they do in decimal, because the underlying values are still the same.

Addition

Addition may be the only unituitive operation in binary, since people are more used to decimal notation. Remember that $1 + 1 = 10$ in binary. Since there are only two digits, the full addition table is as follows.

$+$	0	1
0	0	1
1	1	10

When adding larger numbers, remember to carry the result of any additions. But the overall algorithm for adding becomes much simpler with just two digits.

	0^1	0^1	1^1	1^1	1^1	1	0^1	1
$+$	0	1	0	1	0	1	0	1
	1	0	0	1	0	0	1	0

Note that no more intimate knowledge of binary is needed, the standard algorithm for adding works just as well on binary as it does on decimal.

Multiplication

With only two digits (one of which is zero), multiplication becomes very simple in binary.

$\times$	0	1
0	0	0
1	0	1

Multiplying large numbers does result in a high amount of additions, however. Once again, only the standard algorithm for multiplication needs to be used. In fact, only in the addition portion does it matter that the numbers are written in binary.

							0	0	1	1	1	1	0	1
$\times$							0	1	0	1	0	1	0	1
							0	0	1	1	1	1	0	1
					0	0	1	1	1	1	0	1		
			0	0	1	1	1	1	0	1				
$+$	0	0	1	1	1	1	0	1						
	0	1	0	1	0	0	0	1	0	0	0	0	0	1

It is important to note that when multiplying two binary numbers, the bitwidth of the product is as wide as the sum of the factors' bitwidths. Multiplying two 16-bit numbers will result in a product that fits in 32 bits.

Bitwise AND

It is extremely useful to combine two binary numbers using the multiplication table, but without multiplying the two. This corresponds exactly to the logical AND truth table, and thus is called **bitwise AND**, as it ANDs together each corresponding pair of bits from a given binary

number. An ampersand (&) is used as the binary operator.

&	0	1
0	0	0
1	0	1

Unlike multiplication, this operation is very fast, and the bitwidth of the result is the same as the bitwidths of the inputs. Just like a logical AND, the result is 1 if and only if the two inputs are both 1.

	0	0	1	1	1	1	0	1
&	0	1	0	1	0	1	0	1
	0	0	0	1	0	1	0	1

Bitwise OR

Just as bitwise AND combines two bits like a logical AND table, there exists a corresponding **bitwise OR**, that combines bits like a logical OR truth table. The result is 0 if and only if the two inputs are both 0.

|	0	1
0	0	1
1	1	1

The symbol for bitwise OR is the pipe(|).

	0	0	1	1	1	1	0	1
|	0	1	0	1	0	1	0	1
	0	1	1	1	1	1	0	1

Bitwise XOR

While there is an exclusive OR in logic, it is little-used by mathematicians. Programmers, however, use this logic extensively, and thus there exists a **bitwise XOR** operator, with its own table. The caret character (^) is used to indicate this operation.

^	0	1
0	0	1
1	1	0

The idea behind the operation is that it detects that the inputs are different. The result is 1 if and only if the inputs are different.

	0	0	1	1	1	1	0	1
^	0	1	0	1	0	1	0	1
	0	1	1	0	1	0	0	0

Bitwise NOT

Just as the logical negation operation inverts the true/false value of a proposition, so too does the **bitwise NOT** operator reverse the bits of a value. The tilde, $\sim$, is used to indicate this operation. Note that it only operates on a single value, rather than two.

$\sim$	0	1
	1	0

Storing Data in Binary

All storage in a computer has a specific bitwidth: a number may be encoded in eight bits, sixteen, thirty-two, or even sixty-four bits. Each bit in a number naturally corresponds to the next power of two, just as in decimal in corresponds to the next power of ten.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
128	64	32	16	8	4	2	1	
0	0	1	1	1	1	0	1	= 61

Letters and Text

Any English letters or words are stored by assigning each letter a specific numeric value. The most common such mapping is known as **Unicode**, which contains more than 128,000 characters across different languages and writing systems. For instance, the capital letter A is represented by the number 65. To encode the word “ABC”, the computer would store 65-66-67.

Negative Numbers

So far, only natural numbers have been described in binary. Negative values (thereby covering integers) need to be encoded in some fashion.

Negative integers are encoded in binary using **Two's Complement**. This means that the most significant bit (the bit position with the greatest value) is actually treated as a negative instead of positive value.[5]

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
−128	64	32	16	8	4	2	1	
1	0	1	1	1	1	0	1	= -67

The benefit of this system is that operations do not need any additional structure or rules to work, yet the entire number is stored using only binary digits, and no extra sign.

	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	1	0	1	1	1	1	0	1	= -67
+	0	1	0	1	0	1	0	1	= 85
	0	0	0	1	0	0	1	0	= 18

Floating-Point Values

It is impossible for computers to represent all possible real numbers. This is largely due to the constraints of space in storing a number (the bitwidth). While it is possible to store arbitrary rational numbers, most numeric storage actually abbreviates the rational numbers even further to a small, encodable subset in a given bitwidth.

The most common form of encoding rational numbers is called **IEEE Floating-Point**. This takes a set of bits and encodes three things using

[5] Two's complement storage is actually one of a number of methods of encoding negative numbers, but is the most common on most hardware platforms.

different sets of bits:

1. Whether the number is positive or negative (the **sign**)
2. Where the decimal point should be placed (the **exponent**)
3. The digits of the number, sans decimal point (the **mantissa**)

This is based on scientific notation, where numbers are written as follows:

$$6.022 \times 10^{23}$$

The integer exponent may be positive or negative, which moves the decimal point to the left or the right by the specified amount. The decimal mantissa indicates the numeric value of the number.

Note that floating-point numbers have limited precision. If a very large value is added to a very small value, and significance of the smaller value is lost in the presence of the larger value.

$$5.9724 \times 10^{24} + 9.1093 \times 10^{-31} = 5.9724 \times 10^{24}$$

Additionally, not all values can be represented in floating-point, just as not all rational number can be written in decimal notation with a fixed number of digits.

$$1/7 = 0.142857\ldots \neq 0.14285714285714285$$

While it seems like these are a number of points against floating-point numbers, they are extremely efficient when it comes to performing arithmetic operations on rational numbers. Many programmers, however, will choose to use integer values when possible to avoid these potential precision issues.

Exercises

1. Classify the following numbers, in descending priority of Prime, Natural, Integer, Rational, or Real.

1. $8 \div 4$
2. $8 \div 3$
3. $3 - 8$
4. $\sqrt[3]{8}$
5. $4 \times \sqrt{8}$
6. $\log_4 8$
7. $\log_8 3$
8. $\log_3 8$
9. $83!$
10. $84!$
11. 83
12. $8 \mod 4$
13. $4 \mod 8$
14. $e^{\pi} - \pi$

2. Circle all prime numbers.

1. 13
2. 8928376
3. 91
4. 0x91
5. 81
6. 101
7. 36963936903
8. 11111101
9. 11111011
10. 0b11111101
11. 0x11111011
12. 9112380612381630 1
13. 2
14. 1
15. -1

3. Memorize the values of the hexadecimal letters.

4. Memorize the powers of 2 up through 32,768.

5. Complete the chart.

Decimal	Binary	Hexadecimal
10		0xA
32	0b100000	
	0b1010101	0x55
67		
	0b1100001	
		0x101
128		
	0b1111111	
		0xAC
17		
	0b10101010	
		0xE

6. Calculate the following values in their binary representation.

$$\begin{array}{rr} & 10110 \\ + & 1001 \\ \hline \end{array} \qquad \begin{array}{rr} & 10110 \\ - & 1001 \\ \hline \end{array} \qquad \begin{array}{rr} & 10110 \\ \times & 1001 \\ \hline \end{array} \qquad \begin{array}{rr} & 10110 \\ \div & 1001 \\ \hline \end{array}$$

$$\begin{array}{rr} & 10110 \\ \& & 1001 \\ \hline \end{array} \qquad \begin{array}{rr} & 10110 \\ | & 1001 \\ \hline \end{array} \qquad \begin{array}{rr} & 10110 \\ \hat{} & 1001 \\ \hline \end{array}$$

7. Calculate the following values in their binary representation.

1011010	1011010	1011010	1011010
$+$ 1111	$-$ 1111	$\times$ 1111	$\div$ 1111

1011010	1011010	1011010
& 1111	| 1111	^ 1111

8. Calculate the following values in their hexadecimal representation.

0xDEAF	0xDEAF	0xDEAF	0xDEAF
$+$ 0xABBA	$-$ 0xABBA	$\times$ 0xABBA	$\div$ 0xABBA

0xDEAF	0xDEAF	0xDEAF
& 0xABBA	| 0xABBA	^ 0xABBA

9. Rewrite these values in binary, then calculate their value using that representation.

300	200	100
$+$ 400	$-$ -300	$\times$ -200

10. Calculate the following values using a calculator.

1. $\log_2 0b101101111$
2. $\log_{10} 0b101101111$
3. $\log_{16} 0b101101111$
4. $\log_2 0xABBAACCA$
5. $\log_{10} 0xABBAACCA$
6. $\log_{16} 0xABBAACCA$

This page intentionally left blank.

Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.

Leopold Kronecker

CHAPTER 6

Sets

Definition

Sets are the fundamental unit or object in mathematics. All other constructs: numbers, fractions, matrices, and more—are derived from sets.

A **set** is an unordered collection of objects. To indicate the elements of a set, curly braces enumerate the collected objects.

$$A = \{1, 18.3, \text{apple}, \pi, \text{the crown jewels}\}$$

Elements in sets are considered unique; duplicates are not counted, so that means that $\{1, 1, 1\} = \{1\}$.

It is also possible to describe the contents of a set using a special form of notation, rather than enumerating every single member. It involves a variable and a description of valid values for that variable.

$$B = \{x \mid x \text{ is even}\}$$

This is read as “The set of all x such that x is even.”

To denote that an object belongs to a set, a new proposition is defined, that of set membership.

Set Membership

It is said that x **is an element of the set** A if x is an object in A, written $x \in A$. If x is not an object in A, it can be said that $x \notin A$.

$$2 \in \{x \mid x \text{ is even } \}$$

$$2 \notin \{x \mid x \text{ is odd } \}$$

Once again, all members of a set are considered unique, so there's no concept of "the number of times" that an element is in a set; it is either a member of a set or it is not a member of that set.

Just as propositions can be combined to form new propositions, so too can sets.

Set Intersection

Intersection: Of two sets A and B, the new set $A \cap B$, which contains only the objects that are in both A and B.

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

There are a number of identities and facts when it comes to set intersection.

$$A \cap A = A$$
$$A \cap \{\} = \{\}$$
$$A \cap B = B \cap A$$
$$A \cap (B \cap C) = (A \cap B) \cap C$$

Empty Set

It is possible for a set to contain no elements at all:

$$\{2, 4, 6, 8\} \cap \{1, 3, 5, 7\} = \{\}$$

This special set containing no elements is called the **empty set**, and is sometimes written as $\emptyset$. Because all empty sets are equal to each other, it is often referred to as *the* empty set.

Set Union

Union: Of two sets A and B, the new set $A \cup B$, which contains only the objects that are in either A or B.

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

Set unions have a number of identities to aid in their manipulation.

$$A \cup A = A$$
$$A \cup \{\} = A$$
$$A \cup B = B \cup A$$
$$A \cup (B \cup C) = (A \cup B) \cup C$$

Combining set union with set intersection reveals other important relationships.

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \qquad A \cup (A \cap B) = A$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \qquad A \cap (A \cup B) = A$$

Subsets

There exist other operations on sets that will yield additional propositions.

Subset: Of a set B, a set A such that every element $a \in A$ implies that $a \in B$[1]. This is phrasable as a proposition $A \subseteq B$.

To show that one set is a subset of the other, assume that some arbitrary element $a \in A$ exists. Then show that $a \in B$. This is sufficient to show that $A \subseteq B$.

Example: Proove for $A = \{x \mid x \text{ is even}\}$ and $B = \{x \mid x \in \mathbb{Z}\}$, $A \subseteq B$.

Proof: Let x be an even number. Since an even number must be an integer, $x \in B$. Thus $\{x \mid x \text{ is even}\} \subseteq \{x \mid x \text{ is an integer}\}$.

But wait! Is $\{\} \subseteq A$? Because proving a subset relationship seems to require *assuming* that something exists in the subset. How can this work when nothing exists in the subset?

Such a proof requires a bit more math than has been introduced so far, but the result is that, yes, for any set, the empty set is a subset of it.

[1] Which could be written as $A \subseteq B \iff (a \in A \implies a \in B)$

Equality

Now that subsets are defined, it is possible to define when two sets are equal, by using a subset relation in both directions.

Equality: Between two sets A and B, the proposition $A = B$, given by $A \subseteq B \wedge B \subseteq A$.

While this seems like a funny definition of equality, it is useful in mathematical proofs to determine when two sets are equal. Assume that an arbitrary element a exists in set A, then show that it also exists in B, thus satisfying $A \subseteq B$. Then do the reverse, showing that $B \subseteq A$. That is sufficient to showing that the two sets are equal.

Set Difference

Difference: Between two sets A and B, the new set $A - B$[2], which contains only the objects in A that are not in B.

$A - B = \{x \mid x \in A \wedge x \notin B\}$

[2] sometimes writtes $A \setminus B$

Cardinality

It is possible to talk about the number of elements in a set. This is known as the **cardinality** of the set. This is simple for finite sets.

$$\Big|\{1, 2, 3, 4, 5, baker\}\Big| = 6$$

The vertical bars are the operator that mean "the cardinality of the set."

For non-finite sets, the result of this operation gets a little unusual, mostly because there are mathematically different types of infinity.

The basic form of infinity is referred to as $\aleph_0$. This term, read "aleph-null," is the cardinality of the natural numbers. The natural numbers, $\mathbb{N}$, is a set of numbers whose cardinality is infinite. This leads to occasionally confusing results, as $\aleph_0 = \aleph_0 + 1 = \frac{\aleph_0}{2}$

$$|\mathbb{N}| = \aleph_0$$

Products

Given a pair of sets, it is possible to define a **Cartesian Product** or **cross product** on the two, which generates a new set made up of unusual elements.

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$$

$$\{2, 4, 6\} \times \{1, 3\} = \{(2, 1), (2, 3), (4, 1), (4, 3), (6, 1), (6, 3)\}$$

The expression (a, b) is a new type of mathematical object known as an **ordered pair**. Unlike sets, ordered pairs have a first and second element to them, and therefore an ordering. Sets are unordered, ordered pairs are ordered.

Ordered pairs are equal only if the first elements are equal and the second elements are equal:

$$(a, b) = (c, d) \iff a = c \wedge b = d$$

$$(3, 7) \neq (7, 3)$$

$$(5, 7 - 5) = (2 + 3, 2)$$

Note that the cardinality of a cross product is the product of the cardinalities:

$$|A| \times |B| = \left|A \times B\right|$$

It is possible to create a cross product across multiple sets, resulting in an **ordered N-tuple**, where **N** is the number of sets.

$$A \times B \times C = \{(a, b, c) \mid a \in A \land b \in B \land c \in C\}$$

A set crossed with the empty set yields the empty set.

Combinations & Permutations

Imagine a set of three objects:

$$\{green, red, blue\}$$

A reasonable question to ask is "If two elements were selected from the set to form a subset, how many possible subsets are there?" By testing each possibile arrangement, it can be shown that there are three such subsets:

$$\{green, red\} \qquad \{green, blue\} \qquad \{red, blue\}$$

For a larger starting set, the number of subsets also grows. This growth follows a simple formula that is based on the number of elements of the original set (n), and the size of the resultant subsets (k). This is called the number of **combinations**, given by:

$$C(n, k) = \frac{n!}{(n-k)!k!}$$

This is often written as $\binom{n}{k}$, read as "n choose k."

Suppose, instead of making subsets, that one wanted to extract the elements of the set in a particular order. How many ways are there to extract two elements from the set of three objects?

$$(green, red) \qquad (green, blue) \qquad (red, blue)$$
$$(red, green) \qquad (blue, green) \qquad (blue, red)$$

Note that these are ordered pairs, not sets! Sets do not have ordering to them. The number of these orderings is called the **permutation**. It is given by the equation:

$$P(n, k) = \frac{n!}{(n-k)!}$$

There is no such shorthand (other than $P(n, k)$) to indicate a permutation. The number of permutations is obviously equal to or higher than the number of combinations.

Power Sets

The **power set** of a given set is a set containing all the subsets of that set. Note that this will include both the original set as an element, as well as the empty set as an element.

$$\mathcal{P}(A) = \{a \mid a \subseteq A\}$$

$$\mathcal{P}(\{a, b, c\}) = \big\{\{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\big\}$$

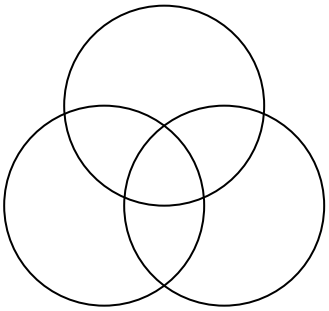
The cardinality of a power set is exactly two to the power of the original set's cardinality, as each element doubles the possible number of subsets.

$$\big|\mathcal{P}(A)\big| = 2^{|A|}$$

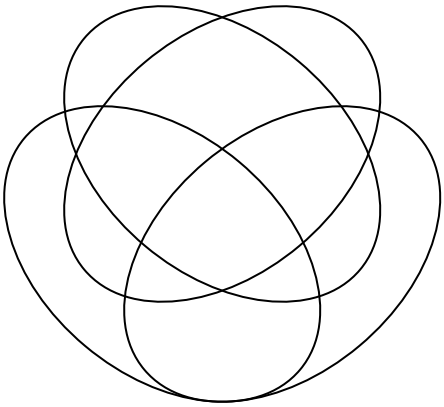
This concept is important as it will reflect how possibilities grow as new variables are added.

Diagrams

When discussing sets and their relations, a picture can be very useful. For instance, many people are introduced to **Venn diagrams**[3] to show the relationship between sets.



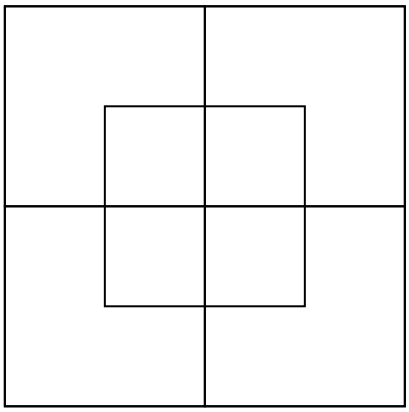
But, Venn diagrams are limited in that it is difficult to scale them for multiple sets.



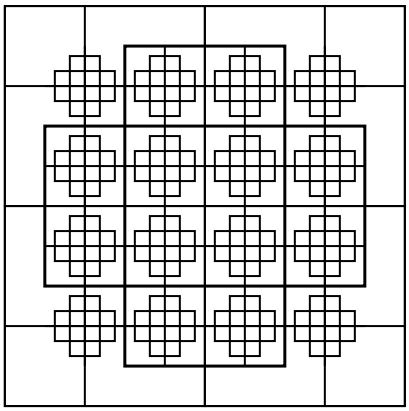
Instead, when it comes to trying to track multiple sets of on/off aspects, a **Carroll diagram**[4] is generally clearer and easier to work with.

[3] Created by John Venn in 1880, as a special case of Euler diagrams.

[4] Created by Lewis Carroll (yes, that one) in 1896.



Carroll diagrams extend to multiple aspects quite easily.



Exercises

1. Are the following propositions true in all cases?

    1. $A \subseteq B \iff B \subseteq A$
    2. $((A \subseteq B) \land (B \subseteq C)) \iff A \subseteq C$
    3. $(a \in A) \land (A \subseteq B) \implies a \in B$
    4. $(a \in A) \land (a \in B) \iff a \in (A \cup B)$
    5. $(a \in A) \land (a \in B) \iff a \in (A \cap B)$

2. Determine the resultant sets.

    1. $\{2, 4, 6, 8\} \cup \{1, 3, 5, 7\}$
    2. $\{2, 4, 6, 8\} \cap \{1, 3, 6, 8\}$
    3. $\{x \mid x \text{ is even}\} \cap \{x \mid x \text{ is odd}\}$
    4. $\{x \mid x \text{ is even}\} \cup \{x \mid x \text{ is odd}\}$
    5. $\{x \mid x \text{ is even}\} \cap \{2, 4, 6, 8\}$
    6. $\{x \mid x \text{ is even}\} \cup \{2, 4, 6, 8\}$

3. Prove the following identities:

1. $A \cup B = B \cup A$
2. $A \cup (B \cup C) = (A \cup B) \cup C$
3. $A \cap B = B \cap A$
4. $A \cap (B \cap C) = (A \cap B) \cap C$

3. Given an infinite number of red, green, and blue marbles, how many different combinations of *three* marbles contain at least one green one?

4. A sack contains 10 marbles labeled 1-10. If three marbles are pulled out, what are the odds that all the numbers are even? What are the odds that the sum of the numbers is even?

5. Calculate the following combinations. Note that the result is undefined if $n < k$.

$\binom{n}{k}$	n					
	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

6. What is the cardinality of set of current students?

7. What is the cardinality of the power set of current students?

8. Draw a Venn diagram represeting the sets of dogs, four-legged animals, and fish.

9. Draw a Carroll diagram representing the sets of cats, animals with tails, and insects.

10. Draw a Carroll diagram represeting the sets of cats, dogs, mammals, and animals with tails.

This page intentionally left blank.

I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git.

Linus Torvalds

Chapter 7

git

Introduction

`git` is a powerful **Version Control System**. It is designed to track different versions of projects over time, organize history, and elevate conflicts to appropriate users.

`git` has a huge variety of command-line flags and options to make interactive use easy, usually accomplishing a specific goal with a single command. This course will not cover the majority of these options; instead concentrating on the core commands rather than all available options.

Repositories and Working Copies

A **repository** (sometimes referred to as a **repo**) is a collection of all history of a directory tree. This tracks what an older version looked like, who changed it, and why.

A **working copy** (or **WC** for short) is a snapshot of one moment in a repository's history; all its files and their contents. It specifically means that these are now literal files on disk, the working copy is an active, editable directory tree.

Any communication or queries about a project's history must be answered by the repository. Given a file in that history, the repository can be queried as to how the file changes over time, to show differences between versions.

This also means that the repository can look at a working copy's version of a file, and the latest historical version of a file, and compare the two. This is called a **diff**, short for difference.

Repository/WC Interaction

There are a number of commands that go between a working copy and a repository. A simple one is retrieving a file from the repository and putting it into the working copy. This is called a **checkout**. The file is being checked out from the repository (no, it never needs to be returned, only a copy of the repository's version is checked out).

It is possible to checkout individual files or directory trees, from any historcial version.

On the other hand, it may be desirable to put a file in the working copy into the repository. This operation is referred to as a **commit**.

Starting `git`

Help

The `git` program operates by using many different commands to interact with the working copy and its repository. Every command will be of the form `git <command>`.

Quick help (which just lists available flags and options) can be accessed by passing a `-h` flag to any given command. A full manual page for the given command can be viewed by passing the `--help` option.

```
$ git status -h
$ git status --help
```

Configuration

To effectively use git, it first needs some basic configuration set up. While there are many configurable options that can be set in the home directory file `.gitconfig`, only the user's name and email are required to be set. These can be written in the file directly, but it is easier to use the command line to do so:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "sample@example.org"
```

This configuration will only ever be done once per user per machine.

It is a global configuration that persists between various repositories on the given machine.

First Steps

Clones

Most work is done in pre-existing repositories. The act of retrieving a repository from somewhere else, and extracting the current working copy, is called **cloning**, and it is accomplished with the `git clone` command. A URL is passed as an argument that gives the location of the pre-existing repository.

```
$ git clone share/instructor_share/poetry
$ cd poetry
```

The act of cloning will copy the entire repository and the latest working copy into a directory of the same name as the final part of the URL. The protocols supported are `file`, `http/s`, `ssh`, and a custom `git` protocol.

To demonstrate how git tracks content across different commits, some content is necessary. Here, some text from a public domain poem will be tracked[1].

`git status` is an important command that will be used frequently. It will compare the last known snapshot in the repository with the current working copy. Its output will show that `chaos` is an untracked file, which means that git currently does not have a snapshot of that file.

[1] *The Chaos*, by Gerard Nolst Trenité, 1922

```
$ cat >chaos
Dearest creature in creation
Studying English pronunciation
        I will teach you in my verse
        Sounds like corpse, corps, horse and worse.

$ git status
```

Adding and Committing

The next step is to make git aware of the file that has content, so that it can track any changes made. This is a two-step process: the file is first added to a staging area using `git add`, and then to commit that staging area (called the **index**) using `git commit`. After each step, run `git status` to see how git sees the file and its changes.

```
$ git add chaos
$ git status
$ git commit -m 'Add poem on chaos'
$ git status
```

The `git commit` command may take a `-m` flag along with a commit message. If this flag is not passed, then git will drop the user into an editor to write a commit message. Upon commit, the file as added is now a part of the repository, and git is now able to perform additional operations on it.

All commits must have a commit message. It is recommended that every commit message starts with a present-tense verb, such as “Create”, “Merge”, “Add”, “Modify”, etc. It can be difficult to write good commit messages for every commit.

Change

When a file changes in the working copy, git will be able to tell the difference between it and the repository. A second verse will be added to the poem.

```
$ cat >>chaos
I will keep you, Susy, busy,
Make your head with heat grow dizzy;
        Tear in eye, your dress you'll tear;
        Queer, fair seer, hear my prayer.

$ git status
```

Staging Changes

`git status` now indicates that the file in question has been "modified". This means that the file in the repository and the file in the working copy no longer match exactly. This is a signal to the user that some **local changes** have been made in the working copy. Just like adding the file for the first time to the index, these changes must be added to the index before they are committed.

```
$ git add chaos
$ git status
```

Committing Changes Using an Editor

As mentioned, without a `-m` flag, `git commit` will drop the user into a text editor to write the commit message. All commits must have a commit message. Which text editor is used can be configured with `git config`.

```
$ git config core.editor
$ git config core.editor nano
$ git commit
$ git status
```

Moving and Removing Files

git tracks content primarily, and files/directories secondarily. As such, when the content in a file moves, git is able to detect it.

```
$ mkdir gnt
$ mv chaos gnt
$ ls
$ git status
```

Right now, all that git knows is about the files it has been told to track. So, all it is able to detect is that the file has been removed. Its new location must be added, and the old location must be removed. `git rm` will remove a file's entry from the repository at that location.

```
$ git add gnt/chaos
$ git status
$ git rm chaos
$ git status
```

Finally, this change can be committed.

```
$ git commit
$ git status
```

This entire process can be accomplished with the `git mv` command. This long approach was taken to show what is happening under the hood of git.

```
$ git mv chaos gnt/chaos
$ git commit
```

Logging

Revisions

Each snapshot of history is called a **revision**. While it might be intuitive to number these revisions as 1, 2, 3, the distributed nature of version control requires that all users agree on a unique way to refer to a specific revision. As such, all revisions are referred to by a hash identifier[2].

A hash identifier is a string of forty letters and numbers, for example `121828abf077aa366690ed153eb15713863be709`.

Since these revision IDs are somewhat unwieldly, any unique prefix of the hash identifier may be used to identify a specific revision. For example, the previous hash identifier could be abbreviated as `121828`, `121828ab`, or even `1218`, if no other revision started with `1218`.

`git` does keep track of the order of commits, and their timestamps. These revision IDs are used to refer to specific snapshots in that historical order.

NOTE: Revision IDs are unique to the file, author, commit message, and timestamp of the commit. As such, any new commits made by different users or the instructor will have different revision IDs than what may be shown in this text.

[2] Specifically, a SHA-1 hash of the commit.

Viewing History

Once some commits are in place, it is possible to review that history. `git` is flexible when it comes to querying history, using `git log`.

```
$ git status
$ git log
```

`git log` with no options will show the full history, from the current revision back to its oldest ancestor. Each revision is shown as its revision ID, the date and time it was committed, the author, and the commit message.

To see which files are associated (and what happened to them) with each commit, pass the `name-status` option to `git log`. This option shows each file as well as how the commit altered that file:

A	Added
D	Deleted
M	Modified

If a slimmer view of history is desired, the `oneline` option will compress each commit down to a single line of revision ID and the first line of the commit message. This can be handy when dealing with a large history.

```
$ git log --name-status
$ git log --oneline
```

It is also possible to restrict the range of commits. Passing in a revision ID will only show the history up to that revision ID. The `since`

option will only show commits since a certain point in time.

```
$ git log 8ed623
$ git log --since=2017-01-01
```

As a user, `git log` is generally usable as-is. It has a variety of ways to filter, restrict, or view commits, which are generally more for tool-writers and scripts than for most end users.

Checking Out From the Repository

The most important command for moving files from the repository to the working copy is `git checkout`. Checking out is like the reverse of `commit`, which moves files to the repository from the working copy. The `checkout` command moves a file or files from the repository to the working copy.

The simplest form of `checkout` is to "reset" the local changes to a file. Whether a working copy's file was edited or even removed, a `checkout` will restore the last version that the repository knew about (the most recent commit).

```
$ cat /dev/null >gnt/chaos
$ git status
$ git checkout gnt/chaos
$ cat gnt/chaos
```

Note that the `git status` command will give a hint as to how to discard the uncommitted changes to the file. This command can be very destructive!

All local changes are thrown away, unable to be recovered. If a large number of changes have been made in a file, but not committed, it is possible to lose all of them with a single `checkout` command.

```
$ cat >>gnt/chaos
Pray, console your loving poet,
Make my coat look new, dear, sew it!
        Just compare heart, hear and heard,
        Dies and diet, lord and word.

Sword and sward, retain and Britain
(Mind the latter how it's written).
        Made has not the sound of bade,
        Say-said, pay-paid, laid but plaid.

$ git checkout gnt/chaos
```

Moving Through History

A working copy is merely a snapshot in time, and commits store those snapshots. This means that it is possible to change the working copy to a specific commit. This is useful for writing patches against older versions of the software.

Since this involves moving files from the repository to the working copy, the `checkout` command is used. Using a specific revision ID, it is possible to "go back in time" to exactly how the working copy looked when the commit was made.

```
$ git checkout 8ed623
$ cat chaos
$ git checkout main
$ cat gnt/chaos
```

In this example, `main` is a special kind of checkout target known as a **branch**. `main` has so far been the most recent commit made to the

repository.

Branches

A repository can contain an unlimited number of timelines of work, known as **branches**. Each branch is an alternate timeline that may track a different history of commits.

Branch Creation

git maintains a set of branches, a list of which may be viewed with `git branch`. This command is also the way to create new branches.

```
$ git branch
$ git branch rime
$ git branch
$ git checkout rime
$ git branch
$ git status
```

As mentioned before, `git checkout` can fetch all the relevent files in the repository belonging to a given branch. Checking out a branch also "activates" the branch, so that any further work now on the working copy will be associated with the currently checked-out branch.

Switching Working Copies

Every commit has a corresponding working copy associated with it. It is possible to retrieve all parts of that working copy from the repository

```
$ cat >stc/rime_of_the_ancient_mariner
It is an ancient Mariner
And he stoppeth one of three.
`By thy long grey beard and glittering eye,
Now wherefore stopp'st thou me?

$ git add stc/rime_of_the_ancient_mariner
$ git commit
$ cat >>stc/rime_of_the_ancient_mariner
The Bridegroom's doors are opened wide,
And I am next of kin;
The guests are met, the feast is set:
May'st hear the merry din.'

$ git add stc/rime_of_the_ancient_mariner
$ git commit
```

using the pre-existing `git checkout` command. By checking out a branch, the entire current working copy is swapped out for that new one.

```
$ git checkout main
$ git log
$ ls *
$ git checkout xanadu
$ git log
$ ls *
```

Merging Branches

Whenever two lines of development are to be combined, the result is a **merge**. Since content can only exist inside a commit, this merging of

content usually results in a **merge commit**, where all the changes from both branches are present. The `git merge` command will merge these two lines of development.

```
$ git checkout main
$ git merge xanadu
```

It is possible to merge more than two lines of development at once, called an **octopus merge**, but that is outside the scope of this course.

Fast-Forward Merges

`git` will attempt to do the least amount of work possible when merging two branches. If one of the branches being merged is a direct ancestor of the other, the result is a **fast-forward**. The older branch is "moved up" to the newer one, with no additional merge commits needing to be created.

Conflicts

Occasionally, performing a merge will introduce conflicts, when two lines of development disagree about what content should exist in the resultant working copy.

```
$ git checkout main
$ git merge sonnets
```

When this happens, `git` makes no effort to solve the conflict. Rather, it defers to the user to resolve the conflict. If a file can be merged without conflict, git will do so. However, if the same lines are modified in the same file, there is nothing to do but defer to the person most capable of resolving which set should be included.

Any area of conflict is marked with <<<<<<< / ======= / >>>>>>> sections. When the conflict is resolved, the file in questions should be added to the index and committed.

Tags

It can be useful to **tag** a specific commit. A tag is a kind of branch that does not move; it is attached to that specific commit. The commit that a branch's tip indicates changes with each commit to that branch, while a tag always indicates the same commit.

To create a tag, use the `git tag` command.

```
$ git tag version-3.0
$ git tag
$ git tag version-2.0 8ed623
$ git tag
```

When invoked with no arguments, the `tag` command lists all tags present in the repository. When invoked with a single argument, it creates a new tag using that argument as the name of the new tag. It is even possible to tag a specific commit by passing in the revision ID as a second argument.

Just like a branch, a tag may be revisited via `checkout`. Multiple tags can indicate the same commit. Unlike branches, all tag checkouts result in a **detached HEAD**.

Detached HEADs

HEAD is a special reference to the current commit in the working copy. Whenever a new commit is made, or a `checkout` is executed, the

HEAD changes.

A commit may be made with any other commit as the parent. A branch only indicates a single commit, not all its ancestors.

```
$ git checkout 8ed623
$ git status
$ touch empty_file
$ git add empty_file
$ git commit
$ git checkout main
```

Internals

The internal structure of repositories can yield a much greater degree of fluency than memorizing commands.

SHA1 Hashes

The first item to understand is that the revision IDs of commits are really hashes, called SHA1 hashes, of information about the commit. This information is plain text, that is fed through a one-way algorithm that generates the SHA1 hash, the revision ID. This information can be viewed with the `git cat-file` command. This command is not commonly used in day-to-day work, and is only appropriate for this kind of low-level vision into the inner workings of git.

`git cat-file` must take both an operation and a SHA1. The first operation, `-t`, will print out the type of the SHA1, as SHA1s are generated for multiple different types of objects in git. The `-p` operation will print out the object that corresponds to the SHA1.

Commit Objects

In this case, `ffceae5` is a **commit object**. The entire contents of this commit object are as shown using the `-p` flag. Note that the commit object references two other SHA1 values: a parent and a tree.

```
$ git cat-file -t 609e59
$ git cat-file -p 609e59
```

The parent object is merely another commit object. This is what allows git to trace its way back through history, by internally viewing each commit object and following its parent. Merge commits have two parents:

```
$ git cat-file -t d267a4
$ git cat-file -p d267a4
```

Tree Objects

A **tree object** in git is a directory listing. It shows file permissions, the type of content, a filename, and a SHA1 for each piece of content in the directory. That content may be another tree (a subdirectory), or a blob object.

```
$ git cat-file -t 2348d0
$ git cat-file -p 2348d0
$ git cat-file -p 649f01
$ git cat-file -p 532455
```

Blob Objects

Any file content is stored in a **blob object**. The entire contents of the file are stored in this object.

```
$ git cat-file -t a094f9
$ git cat-file -p a094f9
```

This SHA1 only applies to the content in the file, not the name of the file, its permissions, or location (these are tracked by the tree object).

Annotated Tag Objects

The fourth kind of git object is an **annotated tag object**. This is a special form of tag that has a commit message with it.

```
$ git cat-file -t 3c4de3
$ git cat-file -p 3c4de3
```

Only tags that have a commit message or are signed get stored in an object. Other tags are merely tracked in a file of tags. This object stores a SHA1 reference to the object being tagged, along with the commit message and the tag author.

Storage

All git objects are stored in the `.git` directory at the top level of the repository. This directory also stores project-specific configurations, branch and tag information, as well as scriptable hooks into the git process.

```
$ ls .git
$ ls .git/objects/
$ cat .git/config
$ ls .git/refs/
$ ls .git/refs/heads/
$ ls .git/refs/tags/
$ cat .git/HEAD
$ ls .git/hooks/
```

.gitconfig

A number of configuration options and settings may be set in an appropriate configuration file. The global file is `~/.gitconfig` in the user's home directory. This human-readable file contains aliases, settings, and the like for all systemwide git usage by that user.

Each repository has a `.git/config` file that contains overrides for that specific project. So, a given project could be set up to use a different author name, aliases, remote users, and settings. There is even a systemwide `/etc/gitconfig` that can be used to set up system defaults.

The Index

The **index** in git is a staging area for the next commit to be made. When a commit is made, all the content in the index is turned into the next commit's working copy. This means that some local changes may be added to the index, and others left out. This affords a large degree of

flexibility to creating commits in git.

Remotes

git allows repositories to communicate with each other. Just as a working copy can communicate with a repository, so too can repositories communicate with each other. A different set of verbs is used for this communication. These other repositories are referred to from the perspective of the currently active repository, so the other repositories are called **remote repositories** or **remotes**.

Remotes are intended to track the same commit information between the different repositories. They are not intended to track completely different projects, but rather the same project[3].

Repositories are a collection of commits, tags, and branches. When repositories communicate, it involves copying those commits, tags, and branches from one repository to another. All such moves are relative to the repository where the commands are executed.

It is possible to have as many remotes as desired. This can be useful when working with multiple contributors. It is very typical to have just a single remote. This course will assume a single remote.

Each remote is made up of a short name and a URL. The short name is visible only to the local repository. The URL is the location of the remote repository.

To view a list of the remotes that are configured, use the command `git remote`. The short name for each remote is displayed. To see

[3] `submodule` is used to track different projects from within a given project.

the URLs as well, pass in the `-v` option.

```
$ git remote
$ git remote -v
```

Remotes can be added using the same command, using the `add` subcommand. This subcommand requires the short name and URL. Using the short name, they can also be renamed or removed using the appropriate subcommands.

```
$ git remote add oxford git://example.org/oxford-poetry
$ git remote -v
$ git remote rename oxford example
$ git remote -v
$ git remote remove example
$ git remote -v
```

Remote Branches

One of the most useful aspects of remotes is that, like all repositories, they contain branches, in this case **remote branches**. While all local branches can be viewed with `git branch`, a list of all remote branches can be viewed with `git branch -r`.

```
$ git branch
$ git branch -r
```

This may permit the user to merge between local and remote branches by specifying the remote branch, such as `oxford/sonnets`. As `oxford/sonne` is a valid reference to a line of development in git, it can be used in any other such command in git.

```
$ git log oxford/sonnets
$ git diff oxford/sonnets
$ git checkout oxford/sonnets
$ git merge oxford/sonnets
```

Pushing and Fetching

To **fetch** from a remote means to retrieve copies of all new commits, tags, and branches from that remote, and copy them into the current repository. Note that copying commits will not create any conflicts. All commits are unique. It is possible for the two repos to disagree as to where tags are attached. The `git fetch` command will retrieve all such items.

```
$ git fetch
$ git status
```

Just as `git checkout` and `git commit` move content between the repository and the working copy, `fetch` and `push` move content between repositories.

`git fetch` will retrieve all content from a remote repository into the current repository. `git push` will send all content from the current repository to the remote one. Note that it is possible for the remote repository to be ahead of the local one. In that case, the `push` will fail.

Since a `git fetch` does nothing to affect the working copy, it is often followed up by a `git merge` to merge content from the remote branch. This is common enough that a single command can perform both the fetch and merge, called `git pull`. So, normally, to commu-

nicate with a remote repository, only `push` and `pull` are used.

Creation

git repositories are created for the first time (not cloned!) using the `git init` command. This will create the `.git` directory and all the structure necessary to start using git for the given project.

```
$ cd
$ mkdir vcs-demo
$ cd vcs-demo
$ git init
```

`git init` will create and initialize the repository. This involves creating a `.git` subdirectory in the current directory which will track all of the repository data, such as changes, branches, tags, remotes, and content. All git commands that work with a repository must be at this level or deeper within the directory tree. `git` will recursively look upward until it finds a directory containing this `.git` directory. This top-level directory where the `.git` directory lives is sometimes called the **root** of the git repository.

The initial repository created will have no commits and no history.

Rebasing

One of the most frightening and powerful tools in git is its ability to rewrite history, called **rebasing**. It involves moving the commit-arrows around in a given project, possibly editing content along the way.

Rebasing should only be done with content that has not been shared with others! The act of rebasing changes the revision IDs of the commits, content, and trees, meaning that executing such rewrites on shared history will result in nigh-irreconcilable conflicts. As such, only an individual user, working on unpublished commits, should employ the `git rebase` command to rewrite history.

Exercises

1. Find out how to limit the number of log items in `git log` to just the last 5 commits.

2. Find out how to view only the commits made by a specific person in `git log`.

3. Find the original version of `chaos` as a blob object by tracing the way back from the current commit.

4. Find all commits where a file was modified instead of added or removed.

The mathematician's patterns, like the painter's or the poet's, must be beautiful; the ideas, like the colours or the words, must fit together in a harmonious way. Beauty is the first test: there is no permanent place in this world for ugly mathematics.

Godfrey Hardy

Appendix A

Proofs

True mathematics is not the act of sitting down with a page-long equation and solving it for x. Rather, it is the act of showing and proving logical concepts. If elementary mathematics is $2 + 2 =?$ and algebra is $2 + x = 4$, then what mathematicans do is *prove* that $2 + 2 = 4$.

The simplest form of proof is the truth table, which shows that two propositions are equal. From there, it is possible to prove additional propositions without use of a truth table, relying solely on the property of equality; being able to substitute one side of an equation for the other.

There are numerous methods of proof, often tailored for specific domains. The proof for an equation in calculus will be different from one for geometry, both structurally and in the techniques used.

Theorem: If two sides AC and BC of triangle ABC are congruent, the angles A and B opposite those sides are congruent.

Proof: Construct CD bisecting angle ACB. $ACD = BCD$. $CD = CD$. Therefore, triangles ACD and BCD are congruent. Therefore, angles A and B are congruent.

Theorem: $A \cap B \subset A \cup B$.

Proof: Let $a \in A \cap B$. Therefore, $a \in A \wedge a \in B$. Since $a \in A$, $a \in A \cup B$. Since this was true for any element a, $a \in A \cap B \implies a \in A \cup B$. Therefore, $A \cap B \subset A \cup B$.

Proofs have not been heavily used in this material, the facts are presented without support. Learning to write proofs for effective use is a

skill in and of itself, honed with practice in mathematical studies.

This page intentionally left blank.

All men are mortal. Socrates was mortal. Therefore, all men are Socrates.

Woody Allen

APPENDIX B

First-Order Logic

Propositional Logic is fairly simple and straightforward. Most mathematics uses a slightly more complex form of logic known as First-Order Logic. This involves the addition of a new type of statement known as an **open predicate**, one that involves variables that may be substituted for to generate an actual proposition. For instance:

x is even.

This statement is not a proposition until x has a definite value, such as 56 or $19/3$ or even $pollywog$. These predicates are often expressed such that they look like a function:

$$\begin{aligned} P(x) &= x \text{ is even} \\ Q(x) &= x \text{ is odd} \end{aligned} \tag{B.1}$$

This is also what allows the definition of sets:

$$\begin{aligned} K &= \{x \mid P(x)\} \\ &= \{x \mid x \text{ is even}\} \\ &= \{\ldots, -4, -2, 0, 2, 4, 6, \ldots\} \end{aligned} \tag{B.2}$$

The $\mid$ reads "such that", so $\{x \mid x < 2\}$ would read as "the set of all x such that x is less than 2."

Predicates also yield a new set of operators: the existential and universal qualifiers, which turn an open predicate into a proposition.

Universal Qualifier

The universal qualifier is an operation that takes an open predicate and yields a proposition.

$\forall x P(x)$

This reads as "for all x, $P(x)$ yields a true proposition." This is the same as saying that for all possible inputs, $P(x)$ would yield a tautology.

$\forall x(P(x) \wedge Q(x)) = \forall x P(x) \wedge \forall x Q(x)$

Existential Qualifier

The existential qualifier takes an open predicate and yields a proposition.

$\exists x P(x)$

This reads as "there exists an x such that $P(x)$ yields a true proposition." This is the same as saying that $P(x)$ is true in one or more circumstances, but not necessarily all.

$\exists x(P(x) \wedge Q(x)) \implies \exists x P(x) \wedge \exists x Q(x)$

Exercises

For the following exercises, indicate which conclusions (if any) follow from the given statements. Remember to derive conclusions from the statements, not gut intuition!

1. Statements: All cars are red. No red things are cheap.

    I. All cars are cheap.

    II. Red things are not cheap.

2. Statements: Some kings are emperors. All emperors are powerful.

    I. All kings are powerful.

    II. All powerful people are kings.

3. Statements: Most directors are American. Some Americans are Texan.

    I. Some Texans are American.

    II. Some directors are Texan.

4. Statements: No flubbers are junko. Krobplet is a flubber.

    I. Krobplet is not a junko.

    II. All flubbers are not Krobplet.

5. Statements: All plants are flowers. All flowers are stinky.

I. All plants are stinky.

II. All stinky things are plants.

III. All flowers are plants.

IV. Some stinky things are plants.

6. Statements: All terrorists are murderers. All terrorists are criminals.

   I. Either all criminals are murderers or all murderers are criminals.

   II. Some murderers are criminals.

   III. Generally, criminals are murderers.

   IV. Crime and murder go together.

7. Statements: Some ducks are feathered. Some feathery things are dinosaurs. All dinosaurs are awesome.

   I. Some awesome things are feathered.

   II. Some awesome things are ducks.

   III. Some dinosaurs are ducks.

8. Statements: All gerplotten are neinblinken. Some neinblinken are verboten. Some rammsteinen are verboten.

   I. Some rammsteinen are neinblinken.

   II. Some verboten are gerplotten.

   III. Some gerplotten are rammsteinen.

9. Statements: Some musicians are deaf. All deaf people are blind. All blind people are excellent dancers.

I. Some excellent dancers are musicians.

II. Some blind people are musicians.

III. Some deaf people are musicians.

IV. Some musicians are excellent dancers.

10. Statements: No marines are soldiers. No soldiers are sailors. Some marines are sailors.

I. No sailor is a marine.

II. No sailor is a soldier.

III. Some soldiers are marines.

IV. All sailors are marines.