

Lecture 1: Introduction

CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

matthew.mckague@qut.edu.au



Outline

Unit overview

What is mathematics?

Modular Arithmetic

Exponents and logarithms

Outline

Unit overview

What is mathematics?

Modular Arithmetic

Exponents and logarithms

Discrete Structures

Really should be called something like *Intro to thinking about computers*. This subject is about:

- ▶ Understanding and applying rigorous mathematical definitions
- ▶ Modelling a problem using mathematical language
- ▶ Applying known solutions to modelled problems
- ▶ Thinking carefully in a rigorous way

Teaching team — Administrative Team

Matthew McKague: Unit Coordinator

- ▶ General enquiries
- ▶ matthew.mckague@qut.edu.au
- ▶ @matthew.mckague on Slack

Mukhtar Hussain : Marking coordinator

- ▶ Enquiries about marking
- ▶ m5.hussain@qut.edu.au

We aim for a 1 business day turnaround for communication.

Teaching Team — Tutors

- ▶ Mukhtar Hussain
- ▶ Jayamine Alupotha
- ▶ Niluka Amarasinghe
- ▶ Alan Yu
- ▶ Calvin Johnson

Tutorials

- ▶ Please watch the lectures *before* coming to tutorials
- ▶ Tutors will go over example exercises
- ▶ Time for questions
- ▶ One tutorial will be recorded and available through Blackboard
- ▶ You are welcome to attend any tutorial time, including online. Preference to registered students if space is short
- ▶ Online tutorials through BlackBoard Ultra Collaborate. Link in Blackboard sidebar.

Assessment

- ▶ Mathematical foundations quiz (on weeks 1 to 5): (30%)
- ▶ Discrete structures project (40%)
- ▶ Problem solving in selected topics (30%)

Important dates

Assignment due dates:

- ▶ Quiz: due Friday 22 April (week 6)
- ▶ Project: due Wednesday 25 May (week 11)
- ▶ Mini projects: due Wednesday 22 June (exam period)

Census day:

- ▶ Last day to withdraw without penalty: Friday, 1 April

Blackboard

Go to Blackboard for:

- ▶ Lecture/tutorial recordings
- ▶ Lecture slides/scribbles
- ▶ Tutorial questions
- ▶ Readings
- ▶ Weekly quizzes
- ▶ Checking your marks
- ▶ Announcements

Additional material

For those who would like more explanation or additional practice:

- ▶ Videos on specific topics
- ▶ Problem worksheets

are available on Blackboard in the relevant week.

Readings

Recommended readings for each lecture come from these two textbooks:

- ▶ *Algebra & Geometry* by Mark V. Lawson
- ▶ *Mathematics of Discrete Structures for Computer Science* by Gordon J. Pace.

Both of these are available for download as a PDF from the QUT library. You will not be tested on material from the readings. They are there to help you understand the lecture content and give more details.

Extras

- ▶ Each topic we cover is an entire area by itself, so we'll just be scratching the surface. I'll give some starting points for you to explore these topics on your own.
- ▶ The tutorials also have *stretch questions* for those who want more of a challenge. These are not testable material.

This is an extra. You can safely ignore this, but it could be very interesting!

Topics

The course is divided into three section:

- ▶ Mathematical foundations
- ▶ Discrete structures
- ▶ Selected topics

Lectures

Mathematical foundations:

1. Abstractions, definitions, modular arithmetic, exponents
 2. Data representations, bits, bit strings
 3. Logic, proofs, axioms recursion
 4. Sets
 5. Predicate logic
- Term break —

Discrete structures:

6. Relations, functions
7. Graphs
8. Trees
9. Digraphs

Selected topics:

10. Finite state automata and regular languages
11. Linear algebra
12. Probability and Bayesian reasoning
13. Program correctness

Readings

- ▶ Lawson: Section 5.4

Outline

Unit overview

What is mathematics?

Modular Arithmetic

Exponents and logarithms

Solving a problem...

Suppose you have the following problem:

- ▶ You have two baskets of apples
- ▶ You know how many apples are in each basket
- ▶ You want to know how many apples there would be if you put all apples into one basket

How to solve this problem?

Extraneous properties

There are lots of properties of this situation:

- ▶ Size and shape of each basket and apple
- ▶ Position and orientation of each apple and each basket
- ▶ ...

and of the change that we are making:

- ▶ Change in position and orientation of each apple and basket

We can use an *abstraction* to simplify the problem by throwing away all the information that is not needed.

Abstractions

Abstractions capture the relevant properties of the situation:

- ▶ x = number of apples in each basket A before moving them
- ▶ y = number of apples in each basket B before moving them
- ▶ z = number of apples in final basket after moving

and of *relationships between* those properties

- ▶ $z = x + y$

Once can then use properties of the *abstraction* to work out the solution.

Mathematical approach problem solving

1. Analyse the problem for relevant properties
2. Create (or use an existing) abstraction of the situation
3. Solve the problem in the abstraction
4. Translate the solution back into the real situation

Natural numbers

The usual abstraction for problems involving counting is the *natural numbers*, i.e. $0, 1, 2 \dots$:

- ▶ Create an abstraction: assign a number to each basket counting the apples
- ▶ Solve the problem in the abstraction: add the numbers together according to the rules of arithmetic
- ▶ Translate the solution back: the sum will tell us how much is in the final basket

Note: the natural numbers are completely abstract. There is no such thing as 1 in the real world.

How abstractions can go wrong

Our abstraction might not work! Some possible reasons:

- ▶ Abstraction might not be exact. E.g. Baskets hold a maximum number of apples. The final basket may have *fewer* apples than the sum of the original baskets because some fall out.
- ▶ Abstraction might not have enough information or the wrong information. Eg: model each basket as either empty or not empty. This is not enough information to determine the number of apples in the final basket.
- ▶ Abstraction might have too much information, making it harder to solve the problem. E.g. giving a list of colours of all apples in basket rather than just the number.

Having the right abstraction makes a huge difference in problem solving!

Abstractions on their own

We often study abstractions without reference to any particular problem or real situation. In this case we call the abstraction a *mathematical theory*. These usually consist of:

- ▶ A collection of *mathematical objects*, eg. numbers, operations (addition, subtraction)
- ▶ *Axioms*: statements about how objects are related to each other, eg. $a + b = b + a$

Additionally, we can derive new relationships by applying logic to the axioms.

The natural numbers

We can define the the natural numbers like so:

Objects: $0, 1, 2, 3, \dots, +$

Some of the Axioms:

1. for all natural numbers a and b , $a + b$ is a natural number
2. for all natural numbers a and b , $a + b = b + a$
3. for all natural numbers a , b , and c , $(a + b) + c = a + (b + c)$
4. for all natural numbers a , $a + 0 = a$

We can combine axioms with logic to create new true statements about natural numbers. Eg. $0 + a = a$ from 2. and 4. above.

Mathematical objects

What are these mathematical objects?

- ▶ We can't say *anything* about what they are! 2 might be the number of apples in a basket, the number of eyes on my face, etc.. Because these are abstract objects, they could correspond to many different concrete objects.
- ▶ The *only* thing we can say about mathematical objects is how they relate to other objects. That is what axioms are for.
- ▶ Formally, objects are just *symbols*. They have no meaning, they are just names.

You might think you know what 2 is, but formally it is just a short name for the number after the number after 0. 0 is just the number such that when you add it to another number you get that same number.

Relationships between objects

What do relationships between objects look like?

- ▶ Formally, relationships between objects are given by *propositions*
- ▶ Roughly, propositions are statements that can be true or false
- ▶ Axioms for a mathematical theory are propositions that we assert to be true for the objects in the theory

Applying theories to real situations

A mathematical theory applies to a real situation if you can:

- ▶ Match up every object in the theory to something in the real situation (at least hypothetically). Eg. for every natural number we can imagine a basket of apples with that amount of apples in it, addition corresponds to combining two baskets into one.
- ▶ All of the axioms in the theory remain true in the real situation. Eg. combining a basket with a apples and one with 0 apples gives you a basket with a apples.

In this case you can determine properties of the real situation. Eg. combining three baskets with 1, 2 and 3 apples in them will always give a basket with 6 apples. We don't have to check.

Mathematicians would say that the real situation is a *model* for the theory.

Theories that don't apply

If a situation does not obey all axioms in a mathematical theory then the theory does not apply. Consider:

- ▶ Each number corresponds to the number of members in a committee
- ▶ Addition corresponds to combining all members of two committees into one committee

But this does not obey the axioms of the natural numbers! Eg:

- ▶ Alice, Bob and Charlie form the committee for creating new assessment items. This corresponds to the number 3.
- ▶ Alice, and Dave form the committee for revising extension request policies. This corresponds to the number 2.
- ▶ We add these committees together to create a new committee of Alice, Bob, Charlie and Dave. This corresponds to the number 4.
- ▶ So $3 + 2 = 4$!

This is not a model for the natural numbers.

An analogy

Suppose two people are playing a board game:

- ▶ The rules of the game (say, chess) are like a mathematical theory
- ▶ This particular game is like a model for chess
- ▶ If they instead play according to some other rules then this is not a game of chess

Truth in mathematics

Statements in mathematics are always relative to a particular mathematical theory:

- ▶ A statement may be true in one theory and not in another: eg $a \times b = b \times a$ for real numbers, but not for matrices.
- ▶ A true statement in a theory says nothing about a real situation if it is not a model for the theory.

Within a theory and its models, truth is absolute:

- ▶ The rules of logic guarantee that true statements in a theory are true in *every* model of the theory.

It's possible for a mathematical theory to be *inconsistent* in which case every statement in the theory is true, but also every statement in the theory is false! Inconsistent theories cannot have any models.

What do mathematicians do?

Mathematicians usually do some of these things:

- ▶ Create new abstractions (often abstractions of other abstractions!)
- ▶ Develop techniques for solving problems in particular abstractions
- ▶ Determine properties of objects in particular abstractions (proving theorems)
- ▶ Apply abstractions to model and solve problems (applied mathematics)

Terminology for models

There is more than one meaning for “model”:

- ▶ A system that embodies the rules in a mathematical theory
- ▶ A mathematical theory that is an abstraction of a particular system (a *mathematical model*)

These two meanings are kind of opposite! From now on, we will usually use the second meaning.

Mathematicians actually restrict “model” to refer to sets of elements and relations that obey the axioms of a mathematical theory. I.e. models of mathematical theories are more math.

How does this relate to computers?

We use computers in an analogous way to mathematics:

1. Transform input into bits using a *data representation*
2. Operate on inputs according to rules embodied in a program
3. Transform bits back into outputs

All of these steps are basically maths!

- ▶ Bits are symbols without meaning, i.e. mathematical objects. Data representations link these symbols to meaningful properties of the problem.
- ▶ Step 2 gives the relationship between the input and the output. The program is an implementation of the rules of some mathematical theory.

Good programmers will reuse code, and mathematicians have thousands of years of code development to choose from!

Outline

Unit overview

What is mathematics?

Modular Arithmetic

Exponents and logarithms

Modular Arithmetic

We'll practice some ideas around mathematical theories by looking at *modular arithmetic* which is very useful in computer science:

- ▶ Computers don't do normal arithmetic, they do modular arithmetic
- ▶ 2's complement representation for negative numbers depends on modular arithmetic
- ▶ Useful for modeling many things, e.g. indices for ring buffers
- ▶ Foundational for many special topics such as cryptography

For brevity, all numbers and variables will be integers for the rest of this section.

Mathematical definitions

In mathematics we very often define new terms:

- ▶ A definition creates a short name for some concept
- ▶ Purely for brevity: definitions do not make it possible to express new things, they just make it easier.
- ▶ Analogous to how we define functions or use temporary variables when programming for brevity and to make code easier to read.

Important: we often use common English words in definitions, but the English meaning is usually not the same as the mathematical meaning. Only the mathematical meaning will apply.

Example definition: Divides

Definition

Let a, b be integers. If there exists another integer c such that $ac = b$, we say that a *divides* b and write $a|b$. Equivalently, we say b is *divisible* by a .

In definitions, we use italics to emphasise the words that are being defined. The definition will always refer to previously defined terms or basic objects (in this case, multiplication and equality).

Divides

The previous definition says that these all have exactly the same meaning:

- ▶ $a|b$
- ▶ a divides b
- ▶ b is divisible by a
- ▶ There exists some integer c such that $ac = b$

We can use them interchangeably. The first three are defined in the definition. They are shortcut names for the last one.

Using definitions

We can use definitions in two ways: replace a term with its definition, or replace the definition with the term.

- ▶ If we see $2 \cdot 3 = 6$ then we could instead say $2|6$ (here 3 plays the role of c)
- ▶ If we see $2|6$ we could instead say “there is some integer c such that $2c = 6$ ”

These are the *only* way that we can use definitions. The defined term is just a stand in for its definition.

Parity

- ▶ Integers have one of two different *parities*:
 - ▶ x is *even* means $2|x$
 - ▶ x is *odd* means $2|(x - 1)$
- ▶ Some properties:
 - ▶ $\text{even} \pm \text{even} = \text{even}$
 - ▶ $\text{even} \pm \text{odd} = \text{odd}$
 - ▶ $\text{odd} \pm \text{odd} = \text{even}$
- ▶ So two numbers have the same parity if their difference is even

Clock arithmetic

- ▶ Now it's 10 o'clock. What time is it in 5 hours?

3 o'clock

- ▶ Time *wraps around* at 12
- ▶ Adding any multiple of 12 hours does not change the o'clock
- ▶ Example: is 27 hours from now the same o'clock as 39 hours from now?
Yes: 39 is a multiple of 12 away from 27
- ▶ So a and b are the same o'clock if 12 divides $a - b$.

Modular arithmetic

Modular arithmetic is an abstraction of parity and clock arithmetic.

- ▶ Parity is arithmetic modulo 2
- ▶ Clocks use arithmetic modulo 12
- ▶ More generally, we can have arithmetic modulo n for any positive integer n .

Modular arithmetic creates *cyclic groups*. Groups are a cornerstone of mathematics.

Modular equivalence

Modular arithmetic works by replacing equality with *modular equivalence*, also called *modular congruence*.

Definition

If $a - b$ is divisible by n we say a and b are *equivalent modulo n* and write

$$a \equiv b \pmod{n}$$

Properties of modular arithmetic

Modular equivalence plays nicely with addition, subtraction and multiplication. Suppose that $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$. Then:

- ▶ $a + c \equiv b + d \pmod{n}$
- ▶ $a - c \equiv b - d \pmod{n}$
- ▶ $ac \equiv bd \pmod{n}$.

This means we can use many tools from normal arithmetic with modular arithmetic.

If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then. $a \equiv c \pmod{n}$

This means that we can free replace one number with an equivalent one.

mod operator

For computing, we might want to store only small numbers. We have a special notation for this:

Definition

$a \bmod n$ is the smallest non-negative b such that $a \equiv b \pmod{n}$

- ▶ Equivalently, $a \bmod n$ is the remainder you get when you divide a by n .
- ▶ Example: $17 \bmod 4 = 1$ because $17 = 4(4) + 1$
- ▶ In Python and many other languages, this is the `%` operator
- ▶ Careful! In some languages, $a \% n$ is negative if a is negative

Divides revisited

Let's practice using definitions to get a useful property of the mod operator.

Lemma

Let a and b be integers. If $a \bmod b = 0$ then $b|a$.

A *lemma* is little statement that is true in a mathematical theory, derived from its axioms. We can show that it is true by using a *proof* which shows the steps necessary to get from the axioms to the statement. We can also use other lemmas in the proof if we have already proved them.

Proof of lemma

Proof.

Let integers a and b be given such that $a \bmod b = 0$. Then from the definition of the mod operator we have:

$$a \equiv 0 \pmod{b}.$$

From the definition of modular equivalence we have:

$$b \mid (a - 0).$$

From a well known lemma we see that $a - 0 = a$ for any integer, so $b \mid a$. □

This is frequently used to determine divisibility or test if a number is even when programming.

Examples of modular arithmetic

- ▶ Parity, clocks, day of the week
- ▶ In RSA encryption we use arithmetic modulo pq where p, q are prime
- ▶ CPU integer arithmetic is always modulo 2^n where n is the number of bits (also in many programming languages)
- ▶ Indices for ring buffers

The **RSA cryptosystem** was the first practical public key encryption scheme. It is still frequently used.

Modular arithmetic in Python

```
>>> 10 % 7                                # Use the % operator for mod
3
>>> 12 % 7
5
>>> (10 + 12) % 7                          # % evaluated before + so () needed
1
>>> (10 % 7 + 12 % 7) % 7                  # % plays nicely with +
1
>>> 10 * 12 % 7                            # * and % evaluated left to right
1
>>> (10 % 7) * (12 % 7) % 7                # % plays nicely with *
1
>>>
```

Outline

Unit overview

What is mathematics?

Modular Arithmetic

Exponents and logarithms

Exponents

- ▶ Notation: a^3 means $a \cdot a \cdot a$
- ▶ a^n means multiply a together n times
- ▶ a is called the *base*
- ▶ n is called the *exponent*

Exponents can be defined where n is not integer. Real and even complex values of n can be used.

Laws of exponents

- ▶ $(ab)^n = a^n \cdot b^n$
- ▶ $a^m \cdot a^n = a^{m+n}$
- ▶ $a^{m-n} = \frac{a^m}{a^n}$ (when $a \neq 0$)
- ▶ $a^{-n} = \frac{1}{a^n}$ (when $a \neq 0$)
- ▶ $a^0 = 1$
- ▶ $(a^m)^n = a^{m \cdot n}$

Important exponents in Computer Science

In computer science you'll see a lot of powers of 2. For example, we have prefixes

- ▶ *kilo-* means multiply by 2^{10}
- ▶ *mega-* means multiply by $2^{20} = (2^{10})^2$
- ▶ *giga-* means multiply by $2^{30} = (2^{10})^3$
- ▶ *tera-* means multiply by $2^{40} = (2^{10})^4$
- ▶ *peta-* means multiply by $2^{50} = (2^{10})^5$
- ▶ *exa-* means multiply by $2^{60} = (2^{10})^6$

For example, one kilobit is $1024 = 2^{10}$ bits.

We will always use these multipliers when talking about bits/bytes.

More important exponents

So many powers of 2!

- ▶ $8 = 2^3$ bits in a byte
- ▶ $32 = 2^5$ or $64 = 2^6$ bit processors
- ▶ $256 = 2^8 = 2^{2^3}$ possible 8-bit characters

Some earlier computers had 12, 24, or 36 bit CPUs.

Exponent examples

- ▶ $2^{10} = 1024$ is the number of bytes in a kilobyte
- ▶ $2^3 = 8$ is the number of bits in a byte
- ▶ $2^{10} \cdot 2^3 = 2^{10+3} = 8192$ is the number of bits in a kilobyte

Logarithms

- ▶ logarithms are the *inverse* of exponents
- ▶ If $n = \log_a x$ then $a^n = x$
- ▶ So \log_a tells what exponent is needed to make x from a :

$$a^{\log_a x} = x$$

Laws of logarithms

- ▶ $\log_a 1 = 0$
- ▶ $\log_a a = 1$
- ▶ $\log_a (x \cdot y) = \log_a x + \log_a y$
- ▶ $\log_a x^y = y \log_a x$
- ▶ $\log_a \frac{1}{y} = -\log_a y$
- ▶ $\log_a \frac{x}{y} = \log_a x - \log_a y$
- ▶ $\log_b x = (\log_b a) \cdot \log_a x$

Base transformation law

- ▶ $\log_a x = \frac{\log_b x}{\log_b a}$
- ▶ Calculators usually have only base 10 and base e
- ▶ Use base transformation to calculate \log_2 etc.

Example: address lines (1)

- ▶ Each address line is 1 bit
- ▶ n bits means 2^n possible addresses
- ▶ How many address lines required for 65536 addresses?

$$\log_2 65536 = 16$$

- ▶ How many address lines required for 4096 kilobytes (one address per byte)?

$$\log_2(4096 * 2^{10}) = (\log_2 4096) + 10 = 12 + 10 = 22$$

- ▶ How many address lines required for 5000 bytes (one address per byte)?

$$\log_2 5000 = 12.28771 \dots ???$$

Ceiling and floor

- ▶ 12.28771... address lines doesn't make sense
- ▶ introduce the *ceiling* and *floor*
- ▶ Ceiling: round *up*. $\lceil a \rceil$ is the next integer *above* a
- ▶ Floor: round *down*. $\lfloor a \rfloor$ is the next integer *below* a
- ▶ We need $\lceil \log_2 5000 \rceil = 13$ address lines

Exponents and logs in Python

```
>>> import math                # log functions need to be imported
>>> math.log2(8)                # log2 means log base 2
3.0
>>> 2 ** 3                      # ** is exponentiation
8
>>> math.log2(2 ** 3)          # log2 returns a float (decimal)
3.0
>>> math.log2(2 ** (3 + 2))
5.0
>>> math.log2(2 ** 3) + math.log2(2 ** 2)
5.0
>>> math.log10(100)            # log base 10
2.0
>>> math.log2(100) / math.log2(10) # base transformation
2.0
>>> (2 ** 3) ** 4
4096
>>> 2 ** (3 ** 4)              # ** operator is not associative!
2417851639229258349412352
```