Lecture 1: Introduction

CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

cab203@qut.edu.au

Outline

Outline

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

Unit Coordinator

- Matthew McKague (cab203@qut.edu.au)

Super-Tutor

- Jayamine Alupotha (cab203@qut.edu.au)

Teaching Team — Tutors

- Niluka Amarasinghe
- Mir Ali Rezazadeh Baee
- Jayamine Alupotha
- Alan Yu
- Mitchell Johnson
- Elham Hormozi
- James Barker
- Maria Vargas Duque
- Marcus van Egmond

Tutorials

- Tutors will go over example exercises
- Time for questions
- One tutorial will be recorded and available alongside the lecture recordings
- Public holiday Fri 2 April — Please attend another tutorial this week

Assessment

- Three sets of problem solving tasks to build skills with concepts (20%)
- Applied project incorporating multiple concepts (50%)
- Cumulative final exam (30%)

Important dates

Assignment due dates:

- Due: 26 March 2021 (End of Week 4)
- Due: 23 April 2021 (End of Week 7)
- Due: 14 May 2021 (End of Week 10)

Project:

- Due: 4 June 2021 (End of week 13)

Final exam:

- During exam period: 11–26 June 2021
- Schedule released early May (Check HiQ)

Blackboard

Go to Blackboard for:

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

Additional material

For those who would like more explanation or additional practice:

- Videos on specific topics
- Problem worksheets

are available on Blackboard

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!

Outline

Topics

- Bits, bit strings
- Set theory, relations, functions
- Logic and proofs
- Graphs and trees
- Regular languages and finite state automata
- Linear algebra

Lectures

1. Modular arithmetic, exponents, bit strings
2. Data representations
3. Set theory, syllogisms
4. Propositional logic
5. Predicate logic

- Break (1 week)

6. Proof Methods
7. Induction, Program correctness
8. Relations, functions, recursion
9. Graphs
10. Directed Graphs
11. Regular languages, finite state automata
12. Linear algebra
13. Review

Readings

- Lawson: Section 5.4

Outline

Divides

Definition 1.1 (Divides)
If, for some pairs of integers a, b, there exists another integer c such that $ac = b$, we say that a **divides** b, with the notation $a|b$
E.g. For $2|14$, we can say "two divides 14" or "14 is divisible by 2"

Parity

- Two different *parities*:
  - *Even*: divisible by 2
  - *Odd*: remainder 1 when divided by 2
- Some properties:
  - even $\pm$ even $=$ even
  - even $\pm$ odd $=$ odd
  - odd $\pm$ odd $=$ 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

- Parity and clock arithmetic are *modular* arithmetic
- Parity: arithmetic modulo 2
- Clock: arithmetic modulo 12
- More generally, arithmetic modulo $n \in \mathbb{N}$ (whole number n)
- When $a - b$ is divisible by n we say

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

a and b are *equivalent* modulo n

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

Properties of modular arithmetic

- Suppose that $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$. Then:
  - $a + c \equiv b + d \pmod{n}$
  - $a - c \equiv b - d \pmod{n}$
  - $ac \equiv bd \pmod{n}$.
- If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then $a \equiv c \pmod{n}$

mod operator

- For computing, we might want to store only small numbers
- $a \mod n$ is the remainder when n divides a. E.g.

$$15 \mod 12 = 3$$

- so $a \mod n$ returns the smallest positive b such that $a \equiv b \pmod{n}$
- In Python and many other languages, this is the % operator
- Careful! In some languages, $a\%n$ is negative if a is negative

Examples of modular arithmetic

- Parity, clocks, day of the week
- In RSA encryption we use arithmetic modulo pq where p, q are prime
- In C/C++, unsigned integer types use arithmetic modulo 2^n where n is the number of bits
- Indices for ring buffers

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

Modular arithmetic in Python

```
>>> 7 % 3                       # Use the % operator for mod
1
>>> (12 * 23) % 5               # Use parentheses for clarity
1
>>> (12 % 5) * (23 % 5)
6
>>> (12 % 5) * (23 % 5) % 5
1
>>>
```

Outline

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)?

$$\log_2 5000 = 12.28771\ldots ???$$

Ceiling and floor

- $12.28771\ldots$ 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!
241785163922925834941235 2
```

Outline

Bits

All data in computers is stored in *bits*.

- A bit is something that has 2 *states*
- Usually label the two states “0” and “1”

Examples of bits

Some examples:

- A mathematical variable x that can take values in $\{0, 1\}$
- A wire that has either low or high voltage
- A mechanical device with two possible positions
- Position of an on/off switch
- Heads or tails on a coin

Uses of bits

Everything that happens in a computer involves bits:

- Inputting data
- Storing data
- Manipulating data (arithmetic, etc.)
- Outputting data

We need to know about *representations* of any possible data as bits, and how to manipulate data as bits.

Bits are also a fundamental unit of *information*, and play an important role in information theory.

Bit strings

Bits are small. We usually use many bits together in *strings*.
Examples:

- 0 (a single bit is also a bit string)
- 00000
- 10101110001
- 11111111111111111111111111111111

Bit strings have a *length* which is just the number of symbols.

Bit string notation

Some notation:

- We'll use a bar over variables to indicate a string: $\overline{x}$
- The set of all strings of length n is $\{0,1\}^n$ (also called n-bit strings)
- *All* bit strings of all lengths are members of $\{0,1\}^*$
- The jth bit in $\overline{x}$ is $\overline{x}_j$ (j goes from 0 to $n-1$)
- We count from the *right*, so $\overline{x}_0$ is the furthest right.

(We'll go over the $\{\cdot\}$ notation when we discuss Sets)

Number of bit strings

How many different bit strings of length n are there?

- Two choices for $\overline{x}_0$, 0 and 1
- *for each* choice of $\overline{x}_0$ there are two choices for $\overline{x}_1$, 4 total
- For each of the four choices for $\overline{x}_1\overline{x}_0$ there are again two choices for $\overline{x}_2$, total of 8.
- In general there will be 2^n possible bit strings of length n

Bit operations

Two types of bit operations:

- Operations on a single bit or pairs of bits
- Operations on bit strings

Operations on bits can also be thought of as logical operators, which we'll talk about later.

NOT

NOT flips a bit: 0 becomes 1 and vice versa.

x	$\sim x$
0	1
1	0

AND

AND is like multiplication.

x	y	$x \,\&\, y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

OR is like addition, but what do you do with 2? Squash to 1.

x	y	$x \mid y$
0	0	0
0	1	1
1	0	1
1	1	1

XOR

XOR is another kind of addition, but squash 2 to 0, like parity.

x	y	$x \hat{} y$
0	0	0
0	1	1
1	0	1
1	1	0

$(\{0,1\}, \hat{}, \&)$ is a *Field*, equivalent to integers modulo 2.

Bit-wise operations

We can apply bit operations *bit-wise* to strings of the same length. If $\overline{z} = \overline{x} \,\&\, \overline{y}$ then

$$\overline{z}_j = \overline{x}_j \,\&\, \overline{y}_j$$

We just perform the operation on pairs of bits. Other operations work similarly.

$\{0,1\}^n$ is a group under bit-wise ^.

Bit shifts

We can move bits around in a string.

- Left shift by m bits drops the leftmost m bits and adds m 0's on the right:

$$011010 << 1 = 110100$$

$$100001 << 3 = 001000$$

- Right shift by m bits drops the rightmost m bits and adds m 0's on the left:

$$011011 >> 1 = 001101$$

$$100000 >> 3 = 000100$$

There are also *left rotations* where bits dropped on the left appear on the right, and analogously *right rotations*.

Concatenation

We can also *concatenate* bit strings, which joins them together. If $\overline{x}$ is an n-bit string and $\overline{y}$ is a m-bit string, then $\overline{z} = \overline{xy}$ is a $(n+m)$-bit string.
Example: $\overline{x} = 000$ and $\overline{y} = 11$ then $\overline{xy} = 00011$.

The set of all bit strings $\{0,1\}^*$ forms a *monoid* under concatenation.

Bit manipulation

Most CPUs work on 8, 32, or 64 bits at a time, not individual bits. So we can't manipulate a single bit directly, but sometimes we want to. But we can do this using bitwise operators.

Some things we might want to do:

- Turn some bits on (set them to 1) aka *set* the bit
- Turn some bits off (set them to 0) aka *clear* the bit
- Flip some bits (NOT them)
- Test if a bit is 0

This is really common in systems programming (operating systems, networking, etc.) and in embedded systems (microcontrollers).

Bitmasks

We use the concept of a *bitmask* (or just mask) to manipulate groups of bits. Assume 4 bit strings with $\overline{x} = 1100$.

- Mask for bits 0 and 2: $\overline{m} = 0101$
- Turn on bits 0 and 2:

$$\overline{x} \,|\, \overline{m} = 1101$$

- Turn off bits 0 and 2:

$$\overline{x} \,\&\, (\sim \overline{m}) = 1000$$

- Turn off all bits except 0 and 2:

$$\overline{x} \,\&\, \overline{m} = 0100$$

- Flip bits 0 and 2:

$$\overline{x} \,\hat{}\, \overline{m} = 1001$$

Bit strings and masks in Python

```
>>> 0b101                       # 0b prefix for binary constants
5
>>> bin(0b101)                  # bin() gives binary representation
'0b101'                         # comes out as a string
>>> print(bin(5))
0b101
>>> print(0b101)                # printed as integer by default
5
>>> bin(0b1100 & 0b1010)        # AND
'0b1000'
>>> bin(0b1100 | 0b1010)        # OR
'0b1110'
>>> bin(0b1100 ^ 0b1010)        # XOR
'0b110'
>>> bin(~0b10)                  # NOT all bits
'-0b11'                         # This is a negative number,
                                # explained next week
```

Bit strings and masks in Python

```
>>> ctrlmask = 1 << 3          # create a mask for bit 3
>>> x = 0
>>> x = x ^ ctrlmask           # flip bit 3
>>> bin(x)
'0b1000'
>>> x = x ^ ctrlmask           # flip bit 3 again
>>> bin(x)
'0b0'
>>> x = x & ~ctrlmask          # turn off bit 3
>>> bin(x)
'0b0'
>>> x = x | ctrlmask           # turn on bit 3
>>> bin(x)
'0b1000'
>>> ctrl = (x & ctrlmask) != 0  # test bit 3 (on = true)
>>> ctrl
True
```