CAB203 | 2022 S1

Dr Matthew McKague | Notes for CAB203 at the Queensland University of Technology

Table of Contents

CAB203: Discrete Structures

In trying to solve complex problems, a powerful approach is to transform the problem into a simpler model by abstracting away some of the less important details. Once in this more abstract form, powerful mathematical techniques (developed over centuries) can be brought to bear. For computing related problems, the most relevant mathematical concepts and techniques come from the field of discrete mathematics, and include arithmetic, logic, set theory, graph theory and functions. This unit

demonstrates how these mathematical concepts and techniques can be used to model and solve real-world problems. The unit also supports subsequent units: CAB301 where algorithms involving graphs are introduced and CAB402 where the mathematical notion of a function provides the basis for alternative programming paradigms.

Week 1: Foundations

Extraneous Properties

Suppose we 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 would we go about solving this?
It's first important to note that there is a lot of useless information given to us when trying to solve this problem.

For example the

- Size and shape of each basket and apple
- Position and orientation of each apple and each basket
- Change in position and orientation of each apple and basket

We can use a process called abstraction to simplify the problem by throwing away all the information that is not needed.

Abstractions

Abstractions allow us to capture the relevant properties of any given situation. Let's take a look at how we would abstract the problem given above.

We first start by extracting all important information needed from the problem and assigning each piece to an arbitrary value.

- 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 the final basket after moving

Now that we've extracted this information we can go ahead and write an equation that will allow us to solve for z.

$$z = x + y$$

One thing to remember when solving problems using abstraction is that abstraction may not work 100% of the time. In reference to the question above this could be because:

- The abstraction may not be exact. Baskets only hold a maximum number of apples. The final basket may have fewer apples than the sum of the original baskets because some apples may have fallen out.
- Abstractions may not have enough information or the wrong information. In our abstraction we assumed that the final basket was empty, what if it wasn't?
- The abstraction may have too much information making it harder to solve the problem.

Abstractions on their own

Often abstractions are studied without reference to any particular problem or real situation. In these cases we call an abstraction a mathematical theory. These usually consist of:

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

Additionally, new relationships can be derived by applying logic to the axioms

Mathematical Problem Solving Approaches

When solving mathematical problems we tend to follow a 4 step plan

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

If we were to walk through the problem above outlining these steps it would look something like this:

- 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

Axioms

As said before an axiom is a statement about how mathematical objects relate to each other. Let's take a look at some examples of 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 then combine axioms with logic to create new true statements about natural numbers.

$$0 + a = a \quad (\text{this is derived from statement 2 and 4})$$

Mathematical Objects

Mathematical objects simply put are just abstractions.

- 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.

So what do these relationships look like then?

- Formally, relationships between objects are given by propositions
- Propositions are statements that can be either true or false
- Axioms for mathematical theory for example 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 (or at least hypothetically). For example, for every natural number we can imagine a basket of apples with that amount of apples in it.

- All of the axioms in the theory remain true in the real situation. For example, combining a basket with *a* apples and one with 0 apples gives you a basket with *a* apples.

Using the apple example we can determine properties of the real situation. Combining three baskets with 1, 2, and 3 apples in them will always give a basket with 6 apples. Mathematicians would say that the real situation is a model for the theory.

There are cases however where a theory may not apply. Let's take a look at an example of one that doesn't obey all axioms. Imagine a situation where each number corresponds to the number of members in a committee:

- 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 our equation ends up as $3 + 2 = 4$.

Truth in Mathematics

Statements in mathematics are always relative to a particular mathematical theory. It is possible for a statement to be true in one theory, but false for another. An example of this being $ab = ba$, this is true for real numbers however, false 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 however, truth is absolute. This means that, by the rules of logic, any true statements in a theory will be true in every model of that theory.

Computers and Mathematics

Computers are used in an similar 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

We can take this one step further by saying:

- Bits are symbols without meaning with data representations linking these symbols to meaningful properties of the problem. This is no different to how mathematical objects work.

- Step 2 gives the relationship between the input and the output. Therefor meaning that the program is an implementation of the rules of some mathematical theory.

Parity

Parity is just another way of saying whether something is even or odd.
Integers can have one of two different parities:

- x is *even* means $2 \mid x$ (Divisible by 2)
- x is *odd* means $2 \mid (x-1)$

Some properties of parity include:

- even $\pm$ even = even
- even $\pm$ odd = odd
- odd $\pm$ odd = even

Clock Arithmetic

Clock arithmetic is a number system where once a number reaches 12 anything that gets added above will be wrapped back to the initial starting point as it is on a 12 hour clock. To see an example of this let's assume it's 10 o'clock, what time will it be if we add 5 hours? 3 o'clock. Using clock arithmetic we can also see that adding any multiple of 12 hours will not change the o'clock value. If it's 10 o'clock and we wait 12 hours, it will still be 10 o'clock (excusing the fact that AM/PM are a thing).

Modular Arithmetic

Modular arithmetic is an abstraction of parity and clock arithmetic. We know parity uses modulo 2 as there can only be 2 values, even or odd, and we know that clocks use modulo 12 as there are only 12 hours on the clock. Therefor, modular arithmetic says we can have arithmetic modulo n for any positive integer n.

Modular arithmetic works by replacing equality with modular equivalence, also called modular congruence. If $a - b \mid n$ then a and b are equivalent *modulo n* and write

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

What this means is 2 things have the same parity if their difference is divisible by n. Another way of looking at this is if 2 things have an equivalent mod n value, then they have the same parity. Let's see an example using

$$5 \equiv 3 \pmod 2$$

Here we see $5 - 3 = 2$ which is divisible by 2 showing us that 5 and 3 have the same parity. We can also see that $5 \mod 2 = 1$ and $3 \mod 2 = 1$ therefor proving that they are modular equivalent.

Modular arithmetic also plays nicely with addition, subtraction and multiplication. Let's look at an example using $a \equiv b + c \equiv d \pmod n$ remembering the axioms we showed before.

- $a + c \equiv b + d \pmod n$
- $a - c \equiv b - d \pmod n$
- $ac \equiv bd \pmod n$

So let's see how these would work with actual values. In the below examples we will be using mod 5.

$$2 \equiv 7 \qquad (5 \mid 7 - 2)$$

$$4 \equiv -1 \qquad (5 \mid 4 - (-1))$$

$$2 + 4 \equiv 6 \equiv 1 \qquad (5 \mid 6 - 1)$$

$$7 + 4 \equiv 11 \equiv 1 \qquad (5 \mid 11 - 1)$$

These are all valid because all answer evenly divide into 5.

Modular arithmetic can be found being used in many places in the real world such as RSA encryption, CPU integer arithmetic and indices for ring buffers.

Modular Operations

For computing, we might want to store only small numbers. We can achieve this using the mod operator (commonly tied to the % symbol in many programming languages). The mod operator works such that $a \mod n$ is the smallest non-negative b such that $a \equiv b \pmod n$. In other words, $a \mod n$ is the remainder you get when you divide a by n.

Lemma

A lemma is a proposition 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

Let's create a lemma and a proof to go along with it to see how all these concepts work together.

Lemma: Let a and b be integers. If $a \mod b = 0$ then $b \mid a$.

Proof:
Let integers a and b be given such that $a \mod b = 0$. 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 proof is frequently used to determine divisibility or test if a number is even when programming.

Modular Arithmetic in Python

As mentioned before, if we want to find the mod of something in python we need to use the % symbol.

```
>>> 10 % 7
3
>>> 12 % 7
5
>>> (10 + 12) % 7          # % is evalutated before + so () are needed
1
>>> (10 % 7 + 12 % 7) % 7
1
>>> 10 * 12 % 7            # * and % are evaluated left to right
1
```

Exponents

An exponent is a value that refers to the number of times a number is multiplied by itself. It is notated as such a^n where a is the base and n is the exponent. An example of this follows

$$a^3 = a \cdot a \cdot a$$

Let's take a look at some of the laws of exponents:

- $(ab)^n = a^n \cdot b^n$
- $a^m \cdot a^n = a^{m+n}$
- ParseError: KaTeX parse error: \tag works only in display equations
- ParseError: KaTeX parse error: \tag works only in display equations
- $a^0 = 1$
- $(a^m)^n = a^{m \cdot n}$

In computer science we use many prefixes based around the powers of 2. These multipliers will always be used when talking about bits/bytes.

Prefix	Multiplied by
Kilo	2^{10}
Mega	2^{20} or $(2^{10})^2$
Giga	2^{30} or $(2^{10})^3$
Tera	2^{40} or $(2^{10})^4$
Peta	2^{50} or $(2^{10})^5$
Exa	2^{60} or $(2^{10})^6$

For example:

- $2^{10} = 1024$ 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. In a simple way it allows us to find the amount of one number we need to multiply to get another. If $n = \log_a x$ then $a^n = x$. For example the logarithm, or inverse, of $2^3 = 8$ would be $\log_2 8 = 3$.

Let's take a look at some of the 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

Most calculators only calculate in base 10 and base e so what happens if we want to calculate in a different base? We can use something called the base transformation law which is written as

$$\log_a x = \frac{\log_b x}{\log_b a}$$

So let's see a working example of this:

$$\log_8 64 = \frac{\log_2 64}{\log_2 8} = \frac{6}{3} = 2$$

And by double checking our answer we can see that it is true

$$8^2 = 64$$

We can take this one step further and show logarithms in a real world example when using memory and address lines. If

- Each address line is 1 bit
- *n* bits means 2^n possible states per address (2^n states containing 0 or 1, or $\{0,1\}^n$ per address)

How many address lines are required for 65536 addresses?

$$\log_2 65536 = 16$$

How many address lines are required for 4069 kilobytes (one address per byte)?

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

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

$$\log_2 5000 = 12.28771...$$

This example becomes a little tricky because we cant have 12.28.. of a byte, we would need to have 13. So how would we expresses this? We can use this notion of ceilings and floors which act as rounding functions that always only go one way.

Term	Definition	Example
Ceiling	Will always round up to the next integer	$\lceil a \rceil$ is the next integer above a
Floor	Will always round down to the next integer	$\lfloor a \rfloor$ is the next integer below a

So to express the example we had before

$$\lceil \log_2 5000 \rceil = 13$$

Exponents and Logs in Python

To use special math functions in python we first need to `import math`. This library gives us access to a majority of the mathematical functions we'll need to use.

```
>>> import math
>>>
>>> math.log2(8)                    # log2 means log base 2
3.0
>>> 2 ** 3                          # ** is exponentiation
8
>>> math.log2(2 ** 3)
3.0
>>> math.log10(100)                 # log10 means log base 10
2.0
>>> math.log2(100) / math.log2(10)  # base transformation for base 10
2.0
```

Week 2: Data Representations

Bits

Bits are the smallest unit of measurement used to store data on computers. A bit has 2 states generally labeled as *0* or *1*. Due to bits being small we usually string them together to create bit strings which can represent larger values.

Bit-string notation:

- When representing a string we put a bar overtop of the variable: $\overline{x}$
- The set of all strings of length n is $\{0,1\}^n$ (i.e. each bit in a bit string is either 0 or 1)
- All bit strings, no matter length, are members of $\{0,1\}^*$
- The *j*th bit in $\overline{x}$ is $\overline{x}_j$

- When working with bit strings we count from right to left. This means that $\overline{x}_0$ is the far most right value.

Operators

Operators, or operations, are mathematical objects we use to transform other objects. An example of this is the $+$ operator, $+$ is a binary operator that takes in two objects and transforms them into a third object. These operators are commonly used in mathematical theories and used axioms to specify these relationships between objects.

Bit Operators

Bit operators allow us to manipulate and change bits in certain ways. There are two types of bit operations: operations on a single bit or pairs of bits and operations on a bit string. Below are a few common operands to know.

NOT (bit flip)

The NOT operand flips the value stored in a bit (changing 0 to 1 or 1 to 0).

x	$\sim x$
0	1
1	0

AND

The AND operand returns 1 if all values are 1, and 0 otherwise. At base value the AND operand works similar to multiplication.

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

OR

The OR operand returns 1 as long as one of the values is 1, and 0 otherwise.

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

XOR

The XOR operand returns 1 as long as only one of the values is 1, and 0 otherwise. Another way of looking at it is addition modulo 2.

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

Bitwise Operations

We can apply bitwise operations on bit-strings of the same length. So we can say, if $\overline{z} = \overline{x} \& \overline{y}$ then:

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

Bit Shifting

Using a method called bit-shifting we can shift bits to the left or right of their original position. When we shift bits the positions that become available are replaced with 0's.

Left shift:

- $011010 \ll 1 = 110100$
- $100001 \ll 3 = 001000$

Right shift:

- $011011 \gg 1 = 001101$
- $100000 \gg 3 = 000100$

Bit Concatenation

Bit concatenation works the way we'd suspect it to work. We can concatenate bit-strings together to create new bit-strings with the length of both combined. So if $\overline{x}$ is an *n-bit* string and $\overline{y}$ is an *m-bit* string, then $\overline{z} = \overline{xy}$ is a *(n+m)-bit* string. If we say $\overline{x} = 000$ and $\overline{y} = 11$ we can say $\overline{xy} = 00011$.

Bit Manipulation

The majority of CPUs now days operate, manipulate and work with 8, 32, 64 and sometimes even 86 bits at a time, rarely ever working with individual bits. So what happens if we want to interact with an individual bit? We can use bitwise operators to do this.

Bit Mask

So say we have a bit-string and want to access certain individual bits how would we go about this? Well we can use something called a bit mask. A bit mask is a special bit string we can use in junction with our bitwise operators to manipulate the original bit string we have. Within a bit mask we put 0's for the places we don't want to change and 1's for the places we do.

Let's assume:

- A 4 bit string with $\overline{x} = 1100$
- A bit mask for bits 0 and 2: $\overline{m} = 0101$

Below are some examples of how we can use bit masks:

- Turn on bits 0 and 2: $\overline{x} \mid \overline{m} = 1101$
- 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: $\overline{x} \hat{} \overline{m} = 1001$

Bitwise Operations with Python

So how would we go about doing all these bitwise calculations in Python? There are a few important things to note first when using bit masks and bit-strings in python:

- We need to prefix our bitmask/bit-strings with `0b`, this tells the variable that we want a binary constant.
- We can use the `bin()` function to give the binary representation of a value.
- Python will by default remove any trailing 0's. For example, `0b1100 ^ 0b1010` will produce `110` instead of `0110`. The value is still there, it isn't lost, it's just hidden from sight.

```
>>> 0b101
5
>>> bin(5)
'0b101'
>>> bin(0b101)
'0b101'

>>> bin(0b1100 & 0b1010)       # AND operator
'0b1000'
>>> bin(0b1100 | 0b1010)       # OR operator
'0b1110'
>>> bin(0b1100 ^ 0b1010)       # XOR operator
'0b110'
>>> bin(~0b101)                # NOT operator
'-0b110'                       # Okay... that's not what we expected
```

We will look into why the NOT bitwise operator isn't visually representing our value properly in the coming weeks. For now a quick fix is to simply AND our value with a bytes worth of 1 bits.

```
>>> bin(~0b101)
'-0b110'
>>> bin(~0b101 & 0xFF)
'0b11111010'                   # We will still see the leadning 1's telling us this valu
```

Bit Mask Operations with Python

We can create bit masks in python and use them to manipulate other bit-strings.

```
>>> bitMask = 1 << 3      # Here we create a bit mask for bit 3 using bit shifting (0b10
>>> x = 0b1010
>>>
>>> bin(x ^ bitMask)      # Use XOR to flip bit 3
'0b10'                    # Visually represented as '0b0010'
>>>
>>> bin(x & ~bitMask)     # Use a combination of AND and NOT to turn off bit 3
'0b10'                    # Visually represented as '0b0010'
>>>
>>> bin(x | (1 << 2))       # Use OR to turn on bit 2
'0b1110'
```

Characters

At their core, characters are just symbols. These can be anything from numerals and mathematical symbols to characters from writing systems and punctuation. Characters can also be unprintable characters, things such as newline, tabs and spaces.

Encodings

To represent our characters as bit-strings we first need an encoding. An encoding must have:

- A set of characters to represent (e.g. the Latin alphabet, mathematical symbols)
- A length n for our bit-strings
- A mapping from characters to $0, 1^n$
- Exactly one unique bit-string for each character. No two characters can share the same bit-string

Let's look at an example to see how we could encode the characters {a, b, +, 8} using 2-bit strings.

Bit-string	Character
00	a
01	b
10	+
11	8

Although this encoding works, it's not very useful or good for a couple of reasons such as:

- The set of characters provided aren't very useful
- There is no logical ordering to how we encoded said characters
- Different types of characters are mixed together (alphanumerical, numerical, mathematical symbols)

So what's an optimal way to encode characters? A common way to order bit-strings is through lexicographic ordering. This type of ordering is done by comparing strings one bit at a time working from left to right and ordering them numerically such that 0 comes before 1. If one string is longer than the other, then the shorter string pads its missing values with spaces (which come before 0). Here are a few examples to showcase lexicographic ordering:

- 000 comes before 100
- 000 comes before 001
- 011 comes before 100
- 01 comes before 010

ASCII

ASCII is an old encoding used for the English text and characters. It's composed of 7-bit strings and holds a total of 128 characters (upper and lowercase Latin characters, numbers, punctuation,

mathematical symbols, spaces, newline, etc). It also contained encodings for special characters like BEL, ESC and NUL which were used by teleprinters. Here is an example of an 8-bit US-ASCII chart:

Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

UNICODE

UNICODE is a modern collection of encoding system that supports nearly every character in most other writing systems. It contains, as of the time of writing, "144,697 characters covering 159 modern and historic scripts as well as symbols, emoji, and non-visual control and formatting codes." (Wikipedia, 2022). UNICODE works by assigning a code point (a hexidecimal string) to each character. There are many different encodings used but the most common are:

- UTF32: UTF32 is a fixed-length encoding which encodes using exactly 32-bits per code point
- UTF16: UTF16 is a variable-length encoding which uses 1 or 2 16-bit strings per code point
- UTF8: UTF8 is a variable-length encoding which uses 1-4 8-bit strings per code point. It was designed to be backwards compatible with the 7-bit ASCII encoding. UTF8 is the most common encoding used due to its backwards compatibility with ASCII. Python strings are encoded with UTF8 by default.

Character Strings

Generally when we go to type or do things on a computer we need to use more than just individual characters at a single time. It's because of this reason that we have strings. Strings start with a sequence of characters with each character representative of its individual bit-string. All the individual bit-strings for each character are then pushed together to create one single long string of bits.

There are however some subtleties when interpreting these bit strings that we need to be careful of. When it comes to fixed-length encodings such as UTF32 or ASCII the only thing we really need to know is when to stop reading the string (i.e. we need to know where the end of the string is). However, for variable-length encodings like UTF8, we not only need to know when to stop reading the string, we also need to know when a particular character ends.

Every computer language has its own way of storing and interpreting strings. However, there are two main methods the computer can solve the above problem:

1. By storing the number of characters along with a string (most modern languages accomplish the problem this way).
2. By using null-terminating strings. After the last character, the computer will store bits `00000000` (the ASCII binary for NUL) to signal the end.

Null-terminator example:

Character	Bit-string
h	`01101000`
e	`01100101`
l	`01101100`
l	`01101100`
o	`01101111`
NUL	`00000000`
hello	`01101000 01100101 01101100 01101100 01101111 00000000`

Binary Representation of Numbers

When counting numbers we usually use a base-10 counting system. That's to say, starting at position 0 at the right-most of the numeral position, position j receives a multiplier of 10^j. We then add up all the values and that's how we count.

Let's see an example of this using the value 5302:

10^3	10^2	10^1	10^0
5	3	0	2

We can see $5 \cdot 10^3 + 3 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0 = 5302$.

When working with numbers in binary we use a base-2 counting system instead. Similar to how position 0 starts at the right-most of the numeral position, position j receives a multiplier of 2^j. We then add up all values to get our number in binary.

Let's see an example of this using the value 203:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	1	1

We can see $2^7 + 2^6 + 2^3 + 2^1 + 2^0 = 128 + 64 + 8 + 2 + 1 = 203$

Here is another example of how base-10 converts to base-2 using 4-bits:

Base-10	**Base-2**	**Base-10**	**Base-2**
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Say we want to store positive numbers 0-225 as 8-bit strings, how would we represent this mathematically?

$$\sum_{j=0}^{7} 2^j \overline{x}_j$$

What we're saying here in this small statement is, starting with $j = 0$ and repeating until $j = 7$ what is the sum of $2^j \overline{x}_j$.

Binary Addition

So how would we add binary values to each other? We do so as we would with any number

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$1 + 1 = 10$$

So now let's convert our binary to base-10 values to see if we did our addition correctly:

2^4	2^3	2^2	2^1	2^0	**Total**
0	1	0	1	0	$2^3 + 2^1 = 10$
0	1	1	0	0	$2^3 + 2^2 = 12$
1	0	1	1	0	$2^4 + 2^2 + 2^1 = 22$

or

2^3	2^2	2^1	2^0	**Total**
0	1	1	0	$2^2 + 2^1 = 6$
0	1	1	1	$2^2 + 2^1 + 2^0 = 7$
1	1	0	1	$2^3 + 2^2 + 2^0 = 13$

We can express this mathematically by saying, for *n*-bit binary numbers $\overline{x}$ and $\overline{y}$, the sum in binary $\overline{z}$ is an *n* + 1-bit binary number where:

$$\overline{z}_j = \overline{x}_j \hat{} \overline{y}_j \hat{} \overline{c}_j$$

$$\overline{c}_{j+1} = (\overline{x}_j \& \overline{y}_j) \mid (\overline{x}_j \& \overline{c}_j) \mid (\overline{y}_j \& \overline{c}_j)$$

The bit string $\overline{c}$ represents the carried bits (take $\overline{c}_0$ to be 0). The equation for $\overline{c}_{j+1}$ says when $\overline{x}_j + \overline{y}_j + \overline{c}_j$ equals either 2 or 3 then $\overline{c}_{j+1} = 1$, otherwise $\overline{c}_{j+1} = 0$.

Binary Addition in CPUs

General in CPUs, integers have a fixed number of bits, these are usually 8, 32, or 64. So what happens when an integer proceeds the number of given bits? Let's see a working example using 8-bit numbers:

Let's assume $\overline{x} = 10000000$ (128) and $\overline{y} = 10000001$ (129). Following our addition rules we would get $\overline{x} + \overline{y} = 100000001$ which is a 9-bit string with the value 257. In this instance, because we've proceeded our 8-bit capacity, the leftmost bit gets dropped and we get left with 00000001 (1). We can actually see that this works the same way as modding our value by 2^8 as $257 \mod 256 = 1$.

So by seeing the above example we can say that CPUs implement integer arithmetic modulo 2^n where n is simply the number of bits.

Negative Numbers in Binary

In programming we have this idea of unsigned and signed integers. A signed integer refers to an integer that can hold both negative and positive values while an unsigned integer can only hold positive values. There are a few ways we can represent negative numbers in binary however we'll be using 2's complement encoding for now (which corresponds to signed integer values). In 2's complement encoding the final bit represents the sign of the number (0: positive; 1: negative). This allows us to represent all values ranging from -2^{n-1} to $2^{n-1} - 1$. To convert a negative number x to its positive equivalent, we can take the modulus of x with 2^n, where n is the bit length. We can also do this by converting x to a positive value and subtracting it from 2^n. Let's see an example using both of these methods where $x = -1$ and $n = 8$:

$$-1 \mod 2^8 = 255 \qquad \text{(Method 1)}$$

$$2^8 - 1 = 255 \qquad \text{(Method 2)}$$

From this we can tell that $-1 \equiv 255 \pmod{256}$ when using 2's complement.

We can also double check this using binary values. We know that, using 8-bit values, the binary representation of 255 is `0b11111111`. We can put this in a 2's complement binary table:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	1	1	1

And from that we can see: $-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = -128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -128 + 127 = -1$

Let's see another example table of 2's compliment using 4-bits knowing that we can represent all values from -2^{4-1} (-8) through to $2^{4-1} - 1$ (7):

Values: -2^{4-1}

-2^3	2^2	2^1	2^0	**base-10**
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Values: $2^{4-1} - 1$

-2^3	2^2	2^1	2^0	**base-10**
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5

-2^3	2^2	2^1	2^0	**base-10**
0	1	1	0	6
0	1	1	1	7

We can also create a table to see how 2's complement interpretation (signed integers) differs from normal binary interpretation (unsigned integers). We'll use 3 bit values for simplicity in the table below:

Bit-string	2's Complement (signed)	binary (unsigned)
000	0	0
001	1	1
010	2	2
011	3	3
100	-4	4
101	-3	5
110	-2	6
111	-1	7

Hexadecimal

Hexadecimal is a base-16 number system (working the same as base-10 and base-2) where the numeral in position j gets a multiplier of 16^j.

Here is the Hexadecimal conversion chart:

Symbol	Bit-string	Base-10	Symbol	Bit-string	Base-10
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11

Symbol	Bit-string	Base-10	Symbol	Bit-string	Base-10
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

So if we wanted to write the hexadecimal string 3B as an 8-bit binary string we would simply concatenate the 4-bit string for 3 (`0b0011`) and B (`0b1011`):

$$00111011$$

We can also convert hexadecimal to base-10 using base-16 conversion. Let's say we wanted to convert the hexadecimal string 3B to base-10:

16^1	16^0
3 = `0b0011` = 3 (base 10)	B = `0b1011` = 11 (base 10)

So we would end up with $3 \cdot 16^1 + 11 \cdot 16^0 = 48 + 15 = 63$.

We use hexadecimal numbers in computer science for a variety of reasons. On one hand they are shorter and more compact than bit-strings due to each hex numeral being exactly 4 bits. They also make it relatively easy to work out bits by hand and there are only 14 symbols we need to keep track of. We can see an example of hexadecimal values used when talking about IP address:

- Binary: 11000000.10101000.00000000.00000001
- Base-10: 192.168.0.1
- Hex: C0:A8:00:01

Scientific Notation in Binary

Scientific notation is a way that we can represent very large numbers and very small numbers in a more simple and compact way. Let's take the example of the integer 100000000, we can also write this as 10^8. We can also see this is true for the integer 0.0000001 which we can represent as 10^{-8}.

We write scientific notation using the following guideline:

$$\pm a.bc \cdot 10^e$$

Where:

- $\pm$ is the sign
- $a.bc$ are the values being used (significant digits)
- e is the exponent
- 10 is the base (the base always matches the base the significant digits and exponent are written in)

Although scientific notation is usually written in base-10 we can also write it in base-2 when working with binary:

$$\pm a.bc \cdot 2^e$$

Where:

- $\pm$ is the sign
- $a.bc$ are bits
- e is the exponent (written in binary)
- 2 is the base (we don't need to encode this as it's understood to be 2 by default)

Floating Point Values

The standard for floating point numbers is IEEE 754 which supports 16/23/64-bit values. Using 16-bits IEEE 754 would look like:

$$\overbrace{0}^{s}\ \overbrace{10101}^{e}\ \overbrace{1010101010}^{f}$$

Where:

- s encodes the sign (1-bit)
- e encodes the exponent (5-bits)
- f encodes the significant digits dropping the leading 1 (10-bits)

Let's see an example of this and encode $1.1010 \cdot 2^{10}$ using floating point standard:

- We know this is a positive number
- The significant digits are 11010
- The exponent is 10 (2)

$$\overbrace{0}^{s}\ \overbrace{10000}^{e}\ \overbrace{1010000000}^{f}$$

Integers and Floats in Python

Python has two basic number types:

- Integers (int): Defaults to 32-bit 2's complement but will internally allocate more memory as it's needed.
- Floats (float): IEEE 754 Double precision: 64-bit floating point

Here are some examples of using integers and floats in Python:

```
# No matter how big or small the value is they are all classified as ints
>>> type(10000000000000000000000000000000000000000)
<class 'int'>
>>> type(100000000)
<class 'int'>
>>> type(1)
<class 'int'>

>>> type(1.0)               # Decimals stored as float values
>>> 5 / 2                   # Division will always return a float
2.5
>>> 5 // 2                  # Floor division will always return an integer
2

>>> type(5 / 2)
<class 'float'>
>>> 5 // 2
<class 'int'>

>>> '{:x}'.format(43722)    # Can format integers as Hex
'aaca'
>>> 0xaaca                  # And vice-verse
43722
```

Week 3: Propositional Logic

Recursive Definition

When defining a type of object, sometimes it may actually be easier to define it in terms of itself, that is to say we define a function that uses the function inside of itself. This is called a recursive definition and is used in many applications such as the factorial function, fibonacci sequence and more. A recursive definition needs at least one base case and at least one recursive case. The base case(s) are evaluated without any reference to the object (i.e. solved without calling the function). While the recursive cases are the cases that will refer back to the definition of the object (i.e. the cases where the function calls itself).

For example, the fibonacci sequence for n can be defined as:

$$f(n) = f(n-1) + f(n-2)$$

We can also define $f(n)$ recursively as:

$$f(n) = \begin{cases} 1 & : n = 1 \\ 1 & : n = 2 \\ f(n-1) + f(n-2) & : n > 2 \end{cases}$$

We can define these recursive definitions in python using functions. Below is an example of how we would implement the fibonacci sequence recursive definition as a function in Python:

```
def F(n):
    if n == 1: return 1
    elif n == 2: return 1
    else: return F(n-1) + F(n-2)
```

Propositions

If we remember back to week one we mentioned that propositions were statements which evaluate to be either true or false. To build complicated and large propositions we general use the help of logical connectives. So let's take a look at some propositions to get a feel of how they work:

- Pineapple belongs on pizza could be true or false
- Humans need to drink water is true
- This sentence is false is neither true or false, so it's not a valid proposition

Usually in computer science, and maths, we use symbols such as x or y to stand in for propositions:

- x = Pineapple belongs on pizza
- y = Humans need to drink water

Propositions can be used in many places, not just in math:

- Propositions from math:
  - $2 \equiv 4 \pmod 2$
  - $54 + 43 = 97$
- Propositions from the world:
  - Apples fall from trees
  - The sun is always shining
- Complex propositions

- It is day if and only if the sun is out
- Humans drink water and humans eat food

Atomic vs Compound Propositions

There are two different types of propositions: atomic or compound. Atomic propositions are propositions that can't be broken down into more propositions. While compound propositions are propositions that are made up of 2 or more atomic propositions. Here are a few examples:

- The sun is shining. This is atomic as it can't be broken down.
- If it is raining then I will go inside. This is a compound proposition as it contains two atomic propositions: 'It is raining' and 'I will go inside'.

Logical Operators

Logical operators, or logical connectives, are connecters we can use to combine two atomic propositions together to create compound propositions. Here are a few common logical operators:

- NOT symbolised by $\neg$
- AND symbolised by $\wedge$
- OR symbolised by $\vee$
- XOR symbolised by $\oplus$
- IF...THEN symbolised by $\rightarrow$
- IF AND ONLY IF (implies) symbolised by $\leftrightarrow$

Logical NOT

NOT operates on a single proposition and gives the negation of it. For example:

- $p =$ Humans need to drink water
- $\neg p =$ Humans don't need to drink water

It's important to note that NOT will always give the exact logical opposite. For example, '$\neg$ He is tall' will be 'He is not tall', which is different from 'He is short'.

p	$\neg p$
T	F
F	T

Logical AND

$p \wedge q$ is true only when both p and q are true.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Logical OR

$p \vee q$ is true as long as one of p or q is true.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Logical XOR

$p \oplus q$ is true only when one of p or q is true.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Logical IF...THEN (Implies)

$p \rightarrow q$ signifies that q must be true whenever p is. However, when p is false, q can be whatever. That's to say, when p is false, then $p \rightarrow q$ is always true.

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Logical IF and only IF

$p \leftrightarrow q$ returns true if and only if p and q have the same truth value. This can also be written as $(p \rightarrow q) \wedge (q \rightarrow p)$ in long form.

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

Boolean Formulas

Boolean formulas are strings of symbols that defines the way we build compound propositions. Formulas are defined by these rules:

- T (true), F (false) and lower case letters are all formulas
- If A and B are formulas, then so are:
  - $\neg A$
  - $(A \wedge B)$
  - $(A \vee B)$
  - $(A \oplus B)$
  - $(A \rightarrow B)$
  - $(A \leftrightarrow B)$

Some examples of formulas may be:

- $\neg p$
- $(p \vee q) \rightarrow (q \oplus p)$
- $(T \vee (p \leftrightarrow F))$
- $T \wedge p$

Classifying Formulas

There are three basic kinds of which will always have the same behaviour no matter what truth values we insert:

- Tautologies: These formulas are always true
- Contradictions: These formulas are always false
- Contingent Formulas: These formulas are true or false depending on the variables
- Satisfiable Formulas: These formulas are either tautologies or contingent formulas

Classification	Always true	Sometimes true	Always false
Tautology	True	False	False

Classification	Always true	Sometimes true	Always false
Contingent	False	True	False
Contradiction	False	False	True
Satisfiable	True	True	False

Tautology

A tautology is always true. Examples may include:

- T
- $\neg F$
- $A \vee \neg A$
- $\neg(A \wedge \neg A)$
- $(A \wedge (A \to B)) \to B$

Contingent

Contingent formulas are sometimes true, sometimes false. Contingent formulas are also satisfiable. Examples may include:

- $A \vee B$
- $A \leftrightarrow B$

Contradiction

Contradictions are always false. Contradiction formulas are not satisfiable. Examples may include:

- F
- $\neg T$
- $A \wedge \neg A$
- $(A \wedge (A \to B)) \to \neg B$

Logically Equivalent Formulas

Two formulas are logically equivalent if they both have the same truth value. For example, $A \to B$ is logically equivalent to $\neg A \vee B$ as they both have the same truth values.

Let's create up a truth table for these two formulas and prove that they are equivalent.

A	B	$\neg A$	$A \rightarrow B$	$\neg A \vee B$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

What we can say then is $A \rightarrow B \equiv \neg A \vee B$

When we have two logically equivalent formulas we can substitute one formula for the other. So let's suppose we have $A \vee C$, if $A \equiv B$ then $A \vee C \equiv B \vee C$.

Logical Operators in Python

We can use logical operators in Python by using the equivalent keywords such as `and`, `or`, `not` and more. It's important to note that `True` and `False` in Python must use a capital letter.

```
>>> True
True
>>> False
False
>>> True and False
False
>>> True and True
True
>>> True or False
True
>>> not True
False
>>> True ^ False
True
```

Python doesn't have an inbuilt function for IF...THEN but we can simply create our own using logical equivalence:

```
def ifThen(x, y):
    return (not x) or y

ifThen(True, False) # False
```

Week 4: Set Theory

Set Theory

Set theory is the mathematical theory of sets. Sets, S, are collections of unique things/items/elements which contain no order of relevance. We can say an item is a member or element of a set using the notation:

$$x \in S$$

and we can say an item is not an element or member of a set using the notation:

$$x \notin S$$

Some mathematical sets have already been defined for us an we can use them to help define our sets. Below are a few common ones to know:

- $\mathbb{Z}$: A set of integers (negative and positive)
- $\mathbb{Z}^+$: A set of integers (positive; starts at 1)
- $\mathbb{Z}_{\geq 0}$: A set of integers (positive; starts at 0)
- $\mathbb{N}$: A set of natural numbers
- $\mathbb{R}$: A set of real numbers (decimals)
- $\mathbb{Q}$: A set of rational numbers (fractions)
- $\varnothing$: An empty set

There are a few ways we can define a set. The first way is to define the set by listing all its elements:

$$SMALLPRIMES = \{1, 2, 3, 5, 7\}$$

The second way to define a set is by using set-builder notation. In the example below we create a base set $x \in \mathbb{Z}$ that we draw from. These values are then added to our set if they match the condition $x = y^2$ for some $y \in \mathbb{Z}$.

$$SQUARES = \{x \in \mathbb{Z} : x = y^2 \text{ for some } y \in \mathbb{Z}\}$$

The final way we can define a list is with an implied pattern. If our list contains a pattern of some sort we can show the first few examples of that pattern and using … we can imply that the next values in the list follow said pattern. It is suggested not to use these however as they can often lead confusion to the reader if the pattern is implied or interpreted incorrectly.

$$EVENS = \{2, 4, 6, 8, ...\}$$

Membership

As explained before, membership is the concept of an item being a member of a set. This just means that if an item is part of a set, it is a member of said set. We can check whether an element or item is in a set in a few ways all depending on how the set is given to us. If the set, *S*, is given to us explicitly,

that is to say we know all the values in the set, we can simply just check if x is in the list. If the set, *S*, is given to us in set-builder notation we can check if x is a member by checking if it satisfies the set-builders condition. Finally if the set, *S*, is given to us with implied values we can check if x is in it by checking whether x satisfies the implied condition. Let's see a few examples of this:

- $5 \in \{1, 2, 3, 4, 5, 6, 7\}$ because it is in the explicit list
- $8 \in \{x \in \mathbb{Z} : x \equiv 0 \pmod 2\}$ because $(8 - 0) \mid 2$ therefore $x \equiv 0 \pmod 2$
- $10 \in \{2, 4, 6, ...\}$ because it follows the implied condition (even numbers)

Equality of Sets

If any two given sets contain the same elements then we can consider them as equal. We can say $S = T$ when:

- Every element $x \in S$ is in T
- And every element $x \in T$ is in S

Let's see a few examples of set equality:

- $\{1, 2, 3, 4, 5\}$ and $\{5, 4, 3, 2, 1\}$ are equal as order doesn't matter
- $\{6666\}$ and $\{6\}$ are equal as we don't count duplicates (only unique values)
- $\{x \in \mathbb{Z} : x^2 = 4\}$ and $\{2, -2\}$ are the same as the second list contains all possible answers for the set-builders notation defined in the first set.

Set Sizes

We can represent the size of a set using $|S|$ notation. $|S|$ counts the number of unique elements in a set. For example:

- $|\{1, 2, 3, 4, 5\}| = 5$
- $|\{6666\}| = 1$
- $|\{+, -, \div, \times\}| = 4$

Subsets

Subsets are smaller sets that are contained inside of bigger sets. We can say that A is a subset of B if every $x \in A$ is in B. That is to say, if all elements in A are also contained in B then A is a subset of B. The notation for this is:

$$A \subseteq B$$

A proper subset is a subset that is not equal to its parent set. That is to say, A is a proper subset of B if A does not equal B. The notation for this is:

$$A \subset B$$

We can also say:

$$A \subset B \equiv A \subseteq B \wedge A \neq B$$

Finally, a set containing all subsets of another set, S, is called the power set and is written as such that $P(S)$.

$$P(\{1,2\}) = \{\varnothing, \{1\}, \{2\}, \{1,2\}\}$$

Referring back to earlier when we looked at set equality, we can no re-define our definition and say $S = T$ when:

- $S \subseteq T$
- $T \subseteq S$

More Set-builder Notation

We can define set-builder sets using two general notations:

1. Set comprehension
2. Set replacement

In set comprehension we specify a subset of elements that match some pre-defined condition.

$$\{x \in S : \phi(x)\}$$

We will talk about what the ϕ symbol means in the following weeks but for now just just imagine it as a statement specifically relating to a set (i.e. evaluates to either true or false. An example of set comprehension is:

$$SQUARES = \{x \in \mathbb{Z} : x = y^2 \text{ for some } y \in \mathbb{Z}\}$$

In set replacement we apply a function to each element of a set and create a set using those returned values.

$$\{f(x) : x \in S\}$$

Using the example above we can create a set replacement variant as such:

$$SQUARES = \{x^2 : x \in \mathbb{Z}\}$$

Set Operations

We can operate on sets just as we can with numbers. Some typical operations we can do on sets are:

- Union ($\cup$): With this operator we grab each value where x is such that x is in A or B.

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

- Intersection ($\cap$): With this operator we grab each value where x is such that x is in A and B.

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

- Difference ($\setminus$): With this operator we can remove items in one set from another. That is to say if, $A = \{1, 2, 3, 4\}$ and $B = \{1, 2, 5\}$ then $A \setminus B = \{3, 4\}$.

$$A \setminus B = \{x : x \in A \wedge x \notin B\}$$

Universe Sets and Compliments

A universe set, U, is a set of all elements that relate to a particular context. This means that U is very context heavy and must either be well defined for the audience or heavily assumed based on context. Once U is defined it is assumed that all elements after, unless stated, are members of said universe. Using universe sets we can define compliments. The compliment of a set is a set that includes every element of the defined universal set that is not contained in the given set. For example:

$$\overline{S} = \{x \in U : x \notin S\}$$

Characteristic Vectors

Assuming our universe isn't too big we can represent our sets as bit-strings, also know as characteristic vectors. So we can say:

- Let $n = |U|$ (let n equal the size of our universe)
- Each element of U is given a number such that $U = \{e_1, e_2, ...e_n\}$
- $\chi_S = \chi_{S_1}\chi_{S_2}...\chi_{S_n}$ where

$$\chi_{S_j} = \begin{cases} 0 & e_j \notin S \\ 1 & e_j \in S \end{cases}$$

So let's see a few examples of this with our universe defined such that $U = \{1, 2, 3, 4, 5\}$:

- $\chi_{\{1,3,4,5\}}$ = `0b10111`
- $\chi_{\{1,3,5\}}$ = `0b10101`

- $\chi_{\{3\}}$ = `0b00100`

$U = \{1, 2, 3, 4, 5\}$

- $\chi_{\{1,3,4,5\}}$ = `0b10111`

Python and Sets

Python allows us to define sets using `{}` and allows us to do operations on these using their respective commands. Here is a small demonstration of sets in Python:

```
>>> S = {1, 2, 3}; T = {1, 2, 3, 4, 5}
>>> S.add(7); print(S)
{1, 2, 3, 7}
>>> S.remove(7); print(S)
{1, 2, 3}

>>> S.union(T)
{1, 2, 3, 4, 5}
>>> S.intersection(T)
{1, 2, 3}
>>> S.issubset(T)
True
>>> T.issubset(S)
False
>>> S.issubset({1, 2, 4})
False

>>> 1 in S
True
>>> 7 in S
False

>>> S - T
set() # empty set
>>> T - S
{4, 5}

# We can also use set-builder notation in python
>>> S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> def p(x): return x % 2 == 0
>>> { x for x in S if p(x) } # x such that x is divisble by 2
{0, 2, 4, 6, 8, 10}
```

The Zermelo-Fraenkel Set Theory

The Zermelo-Fraenkel set theory is a set of nine axioms, although we'll only look at seven, created by two mathematicians, Ernst Zermelo and Abraham Fraenkel. The seven axioms in this set theory

define to us how sets should behave. We will take a look at this informally for now as the math behind it contains things we haven't learnt yet.

1. Axiom of extensionality: If two sets contain the same elements then they are equal to each other.
2. Axiom of regularity: For every non-empty set S, S must contain at least one element y such that S and y are disjoint. That is to say, sets cannot have membership to themselves (be a member of themselves) and cannot be empty as there is only one empty set $\varnothing$.
3. Axiom schema of specification: If we have a set, S, and a formula, $\phi(x)$, then there exists a subset that contains exactly all elements of S satisfying $\phi(x)$ ($\{x \in S : \phi(x)\}$).
4. Axiom of pairing: Given sets $S_1, S_2, ...$, then there exists a set T such that $S_1, ..., S_n \in T$. That is to say, if $S_1, S_2, ...$ are sets, then there exists another set containing all elements of every S_j.
5. Axiom schema of replacement: If we have a set, S, and $f(x)$ is a function on S, then there exists a subset that contains $f(x)$ for every $x \in S$ ($\{f(x) : x \in S\}$).
6. Axiom of infinity: If we have a set, S, and $\varnothing \in S$ with $S_j = \{S_{j-1}\}$, then there exists a set T containing every S_j. Here is a small preview of S:

$$
\begin{aligned}
S_0 &= \varnothing \\
S_1 &= \{\varnothing\} \\
S_2 &= \{\{\varnothing\}\} \\
S_3 &= \{\{\{\varnothing\}\}\} \\
&\vdots \\
T &= \{\varnothing, S_1, S_2, ...\} \\
T &= \{\varnothing, \{\varnothing\}, \{\{\varnothing\}\}, \{\{\{\varnothing\}\}\}, ...\}
\end{aligned}
$$

7. Axiom of power set: For any given set S there exists a set T containing every possible subset of S.

Syllogisms and Set-theory

Syllogisms are simply a way to create deductive reasoning about sets. We first start by creating a premises which is assumed to be true and then apply some valid argument to it. Finally we draw a conclusion based off of our premises and argument. Below is an example of this:

$$\frac{\begin{array}{l}\text{All humans are mortal. (major premise)}\\ \text{Socrates is human. (minor premise)}\end{array}}{\text{Socrates is mortal. (conclusion)}}$$

Example taken from QUT slides

So now let's re-create the example above but using set-theory notation instead.

$$\frac{\begin{array}{l}\text{HUMANS} \subseteq \text{MORTALS}\\ s \in \text{HUMANS}\end{array}}{s \in \text{MORTALS}}$$

Example taken from QUT slides

We can then abstract this information to create a generic valid syllogism type assuming the relation between A, B, and x remain true:

$$\frac{\begin{array}{l}A \subseteq B\\ x \in A\end{array}}{x \in B}$$

We must remember when creating syllogisms that not everything following the above notation will be true. The example below shows how our arguments can follow this notation but still be incorrect.

$$\frac{\begin{array}{l}x \in A \cap B\\ y \in B\end{array}}{y \in A}$$

$$\frac{\begin{array}{l}\text{Einstein was a genius and a physicist}\\ \text{I am a physicist}\end{array}}{\text{I am a genius}}$$

Example taken from QUT slides

Week 5: Predicate Logic

Logical Implication

Logical implication is the process of deriving a true proposition from another true proposition. We can represent logical implication using the notation:

$$A \models B$$

This is equivalent to saying that $A \rightarrow B$ is a tautology. It's important to note that logical implication aren't symmetrical and we can't always make substitutions. We only substitute using logical implication if the left hand side is known to be true.

There are 2 general ways that we can find logical implications, either through truth tables or the use of proofs. When using a truth table we can prove that $A \models B$ by checking every line where $A = T$ that $B = T$.

Let's show that $A \wedge B \models A$ is true using a truth table:

A	B	$A \wedge B$	$A \wedge B \models A$
T	T	T	T
T	F	F	F
F	T	F	F
F	F	F	F

Here is another example of showing $(A \rightarrow B) \wedge A \models B$ is true using a truth table:

A	B	$A \rightarrow B$	$(A \rightarrow B) \wedge A$	$(A \rightarrow B) \wedge A \models B$
T	T	T	T	T
T	F	F	F	F
F	T	T	F	F
F	F	T	F	F

Proofs

A proof is a list of formulas stepping through the process of showing how a mathematical statement logically guarantees its conclusion. That is to say, a proof is a list of steps mathematically showing a statement is correct. A proof starts with a some premises and ensures all other formulas on the proof are:

- Logically equivalent to the formula above it
- Logically implied by a formula above it
- Is the AND of some formula above it
- Logically implied by the AND of some formulas above it

A proof will create a new logical implication $P \models Q$ where P is the AND of all premises listed and Q is the last line of the proof. A proof tells us that assuming all the premises are true, then the conclusion must also be true. Let's see an example of a proof:

Proof: $(M \vee \neg H) \wedge H \models M$

1	$M \vee \neg H$	premise
2	H	premise
3	$\neg H \vee M$	$(A \vee B \equiv B \vee A)$
4	$\neg\neg H$	$(A \equiv \neg\neg A)$
5	$(\neg H \vee M) \wedge H$	$((3) \text{ and } (4))$
6	M	$\text{logical implication of } (5) \text{ using } (A \vee B) \wedge \neg A \models B$

Predicate Logic

A predicate is a proposition that takes in one or more variables as arguments and returns a boolean value based on some logic using those arguments. For example, $P(x) \rightarrow \{T, F\}$ where P is the predicate and x is the parameter or argument taken by P. It's good to note that the variables we pass into predicates are different to the variables we pass into formulas. Predicate parameters can stand in for any value while variables in formulas stand in for propositions which must evaluate to either true or false.

We can use small predicates to form complex predicates just as we could with propositions due to predicates evaluating to a boolean value:

- $A(x) \wedge B(x)$
- $A(x) \rightarrow B(y)$
- $(A(x) \rightarrow B(y)) \wedge A(x)$

Quantifiers

Quantifiers are symbols that allow us to give some information about the parameters used in predicates. Quantifiers are useful as they allows us to form propositions out of predicates without the need to define a boolean value for their parameters. There are two kinds of quantifiers:

1. Existential quantification (∃): Used to say something exists.
2. Universal quantification (∀): Used to say something about all of a certain value.

When we use a quantifier we always need to specify which set our values are coming from and if not we must specify with context which universe our values come from.

Existential Quantification

The existential quantifier is used to say "there exists" and is represented by the $\exists$ symbol. For example, say we wanted to write that there exists a value $x \in \mathbb{Z}$ where $x^2 = 4$. We can write that using our existential quantifier as such:

$$\exists x \in \mathbb{Z}(x^2 = 4)$$

This doesn't tell us what x actually represents, just that there exists some value x that matches our condition.

Universal Quantification

The universal quantifier on the other hand is used to say "for all" and is represented by the $\forall$ symbol. For example, say we wanted to write that for every $x \in \mathbb{Z}$, x^2 is a non-negative. We can do so as such using our universal quantifier:

$$\forall x \in \mathbb{Z}(x^2 \geq 0)$$

In the above example it doesn't matter which value from $\mathbb{Z}$ we use to fill in our predicate we will always get a true statement.

Predicates, Parameters and Quantifiers

When we use quantifiers to specify parameters in predicates the values the quantifier is referring to can no longer be filled in with our own values. For example, in $\exists x P(x)$ the x is filled in by $\exists x$ so we can't use our own value for x anymore.

Due to predicates being able to take in more than one value we can sometimes end up having quantified parameters and free parameters (values that aren't quantified). In an instance like that we can only use our own values for variables that haven't been quantified. Just remember, we can have multiple quantifiers in a statement but only one quantifier per value.

For example, $A(y) = \exists x p(x, y)$:

- x is quantified over and as such we can't fill it in
- y is not quantified over so we can fill it with our own value
- Due to y not being quantified the truth value of $A(y)$ depends on the value of y

Truth Values with Universal Quantification

When we universally quantify over finite sets we have to check all values in the set to determine if the predicate is fully quantifiable. For example:

$$\forall x \in \{0, 1\}(x^2 = x)$$

We can check if this is true by working through each element in our set and checking the condition:

$$0^2 = 0$$
$$1^2 = 1$$

But to class this as false we only need to find one value that doesn't work with our condition.

Truth Values with Existential Quantification

The same can be said when existentially quantifying over a finite set however everything is flipped. Only one value that works needs to be found to show something is true:

$$\exists x \in \{0, 1\}(x^2 = 1)$$

This is true as $1^2 = 1$. However, if we want to show the statement is false we need to check every possible value:

$$\exists x \in \{0, 1\}(x^2 = 2)$$

We can class this as false as every value is false when put through our condition:

$$0^2 \neq 2$$
$$1^2 \neq 2$$

Truth Values with Infinite Sets

For an infinite set when we use an existential quantifier to prove something is true we do it much in the same way as with a finite set. We simply need to find a single value to say it's true:

$$\exists x \in \mathbb{Z}(x^2 = 1)$$

This is true as $1^2 = 1$.

But what if we wan't to show something is false? How do we check if every value in our infinite set doesn't match our condition? We can generally get around this by using different properties of math. For example:

$$\exists x \in \mathbb{Z}(x^2 = -1)$$

This would be false as we know that $\forall x \in \mathbb{Z}(x^2 \geq 0)$ thus showing $x^2 \neq 1$.

Everything is much the same for universal quantifiers except flipped. It's harder to prove everything is true as we need to check all values while we only need to find one value thats false to prove the predicate is false.

Necessary and Sufficient Conditions

When working with predicates we can break conditions down into necessary conditions and sufficient conditions. For example, when $\forall x(p(x) \rightarrow q(x))$ we generally say:

- $p(x)$ is a sufficient condition for $q(x)$
- $q(x)$ is a necessary condition for $p(x)$

Another example could be when $\forall x(p(x) \leftrightarrow q(x))$

- $p(x)$ is a necessary and sufficient condition for $q(x)$
- $q(x)$ is a necessary and sufficient condition for $p(x)$

Boolean Formulas Revisited

When $U = \{T, F\}$ and $A(x, y)$ is a boolean formula with x and y being some formula then we can say:

- "A is a tautology" means $\forall x, yA(x, y)$
- "A is a contradiction" means $\forall x, y\neg A(x, y)$
- "A is satisfiable" means $\exists x, yA(x, y)$

Logical Equivalence with Predicates

Much like with our boolean formulas, predicates can have logical equivalences. Here are a few examples:

- $\neg(\forall xp(x)) \equiv \exists x\neg p(x)$: It is not the case that all humans are male is equivalent to saying there exists some human which is not male.
- $\neg(\exists xp(x)) \equiv \forall x\neg p(x)$: It is not the case that there exists a human which can fly is equivalent to saying all humans can't fly
- $\forall x(p(x) \wedge q(x)) \equiv (\forall xp(x)) \wedge (\forall xq(x))$
- $\exists x(p(x) \vee q(x)) \equiv (\exists xp(x)) \vee (\exists xq(x))$

Example taken from QUT slides

Logical Implication with Predicates

Just as we'd suspect, all previous logical implications also work with predicates. Here are a few examples:

- $(\forall x \in Sp(x)) \land (y \in S) \models p(y)$: All humans are mortal and Socrates is human implies Socrates is mortal
- $p(x) \land (x \in S) \models \exists y \in Sp(y)$: Socrates is male and human implies there exists some human who is male
- $(\forall x \in Sp(x)) \land (S \neq \varnothing) \models \exists y \in Sp(y)$: All humans are mortal and there exists humans implies there exists a human who is mortal

Example taken from QUT slides

Predicates and Python

Python implements predicates simply through the use of functions. Any function without side effects (only changes local state) that returns a boolean value can be used as a predicate. For example:

```
def p(x, y):
    return x >= y

p(1, 2) # False
p(2, 1) # True
```

Python also doesn't directly implement quantifiers but we can substitute them for for-loops and if-statements. For example, $\exists x \in Sp(x)$:

```
S = {-2, -1, 0, 1, 2}

def p(x):
    return x > 0

def exists(S):
    for i in S:
        if p(i): return True

    return False
```

The same is true for $\forall x \in Sp(x)$:

```
S = {0, 1, 2, 3, 4, 5}

def p(x):
    return x >= 0

def forall(S):
    for i in S:
        if not p(i): return False

    return True
```

Week 6: Relations and Functions

Tuple

We can define an ordered pair as (a, b). Unlike sets, pairs can contain duplicates and are ordered. We can say that $(a_1, ..., a_n)$ is a tuple of n size where order matters. It's important to note that:

$$(a, b) \neq (b, a) \text{ unless } a = b$$

$$(a, a) \neq (a)$$

Cartesian Product

The cartesian product of two sets, A and B, is a collection of pairs for every combination of A and B. That is to say:

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

We then say the size of $A \times B$ is given by $|A \times B| = |A||B|$.

For example:

$$A = \{a, b, c\} \quad B = \{1, 2, 3\}$$

	a	b	c
1	$(a, 1)$	$(b, 1)$	$(c, 1)$
2	$(a, 2)$	$(b, 2)$	$(c, 2)$
3	$(a, 3)$	$(b, 3)$	$(c, 3)$

Some examples of when we might use a cartesian product:

- $\mathbb{R}^2$ describes points on a 2-dimensional plane
- $\{0,1\}^2$ is equivalent to the set of bit strings of length n
- $KEYS \times VALUES$ can be used to describe a key-value relationship in a hash map
- $\{0,...,1919\} \times \{0,...,1079\}$ encodes (x,y) coordinates on a 1080p screen.

Tuples in Python

Creating tuples in python is nice and easy as the notation is equivalent to its math variant. It's important to note that you cannot modify a tuple in python.

```
>>> myTuple = (1, 2)        # Create a tuple
>>> myTuple[0]              # Index a tuple
1
>>> myTuple[0] = 2          # Tuples in python are immutable so this will cause an error
```

Relations

A relation is a fixed length collection of tuples where each entry of the tuple is taken from a set.

- A relation on $A_1 ... A_n$ is a subset of $A_1 \times ... \times A_n$
- A binary relation between A and B is a subset of $A \times B$
- A binary relation between A and A is called a relation over A

Some properties of relations are:

- Symmetric
- Reflexive
- Transitive
- Anti-symmetric
- Irreflexive

Symmetry

A binary relation $R \subseteq A \times A$ is symmetric if:

$$\forall a, b \in A \times A \ (aRb \leftrightarrow bRa)$$

That is to say that, whenever we have (a,b) we also have (b,a).

Anti-Symmetry

A binary relation $R \subseteq A \times A$ is anti-symmetric if:

$$\forall x, y \in A \left((xRy \land yRx) \rightarrow x = y\right)$$

This can also be said as:

$$\forall x, y \in A \left(x \neq y \rightarrow \neg(xRy \land yRx)\right)$$

That is to say, if x and y are different, then we can't have both xRy and yRx.

Reflexivity

A binary relation $R \subseteq A \times A$ is reflexive if:

$$\forall a \in A \; aRa$$

Irreflexivity

A binary relation $R \subseteq A \times A$ is irreflexive if:

$$\forall a \in A \; (x, x) \notin R$$

Transitivity

A binary relation $R \subseteq A \times A$ is transitive if:

$$\forall a, b, c \in A \left((aRb \land bRc) \rightarrow aRc\right)$$

Equivalence Relations

An equivalence relation $R \subseteq A \times A$ separates a set into equivalence classes. These are subsets of A that are all related by the relation. For example, the equivalence relation given by $a \equiv b \pmod 2$ gives two equivalence classes, the even and odd numbers.

Partial Ordering

A partial ordering on a set A is a binary relation over A which is:

- Reflexive
- Transitive
- Anti-symmetric

Partial orderings allow us to capture the idea of one thing coming before another. If the relation is irreflexive then we call it a strict partial ordering.

Total Ordering

A total ordering on a set A is a partial ordering R over A that also has the property:

$$\forall x, y \in A \ (xRy \vee yRx)$$

What this means is that we can always compare any two elements of A. If the relation is irreflexive then we call it a strict total ordering.

Functions

A function is a relation f between A and B where for each $a \in A$ there is exactly one $b \in B$ such that $(a, b) \in f$.

$$((a, b) \in f \wedge (a, c) \in f) \rightarrow b = c$$

We can write this as:

$$f : A \rightarrow B$$

Where $f(a)$ is the unique $b \in B$ such that $(a, b) \in f$.

Usually it's said that a function $f : A \rightarrow B$ is defined on all of A meaning:

$$\forall a \in A \ \exists! b \in B \ (a, b) \in f$$

The ! notation simply means unique. So in the above definition we're saying there exists a unique b for every a.

Domain and Range

For a function $f : A \rightarrow B$, we call A the domain of f and B the co-domain. The set $\{f(x) : x \in A\}$ is called the range of f.

The domain is a set of every element for which f is defined. This essentially means that the domain the set of possible inputs to f while the range is the set of all possible outputs.

Composing Functions

Let's imagine we have two functions: $f : A \rightarrow B$ and $g : B \rightarrow C$. Instead of first running f then g we can combine the two using ∘ (of) notation. For example:

$$g \circ f : A \rightarrow C$$

The above definition is given by:

$$(g \circ f)(x) = g(f(x)) \qquad (\text{called g of f of x})$$

Or more formally:

$$g \circ f = \{(x, z) \in A \times C : \exists y \in B \ \ (x, y) \in f \wedge (y, z) \in g\}$$

Inverses

Some functions, not all, have an inverse version. What this means is, given $f : A \to B$, there may exists an inverse $f^{-1} : B \to A$ such that:

$$\forall x \in A \ (f^{-1} \circ f(x) = x)$$

It's important to note that the range of f must match the domain of f^{-1} much like the range of f^{-1} must match the domain of f.

For example:

$$f = \{(1, a), (2, b), (3, c)\}$$

$$A = \{1, 2, 3\} \quad B = \{a, b, c\}$$

$$f^{-1} = \{(b, 2), (a, 1), (c, 3)\}$$

$$f^{-1} \circ f(1) = f^{-1}(a) = 1$$

Functions in Python

The functions we use in Python have a different meaning to the functions we use in mathematics.

- Every computable mathematical function can be written as a function in Python
- Functions in Python that are side-effect free (no global state changes) and deterministic (no randomness) are also functions in the mathematical sense
- Calling a function twice is the same as calling it once

Partial Functions

Sometimes we only care about a subset of a domain, not the entire domain itself. A partial function f from a set S and a set T is a function from a subset of S to T. This essentially means that a partial function is a function mapped to a subset of some set.

Partial Functions in Python

We can implement partial functions in python with the use of dictionaries (or hash-map).

```
>>> f = {'a': 1, 'b': 2, 'c': 3}          # Create a dictionary
>>> f[1]                                  # Given the value, returns the key
'a'
>>> f['c']                                # Given the key, returns the value
3
>>> f['d'] = 4                            # Set new items
>>> f.keys()                              # Get the domain
dict_keys(['a', 'b', 'c', 'd'])
>>> f.values()                            # Get the range
dict_values([1, 2, 3, 4])
```

Sequences

When we care about the order for a set of items we use sequences. We usually implement sequences as lists or arrays depending on the language being used.

We define a sequence of length n as a function $x : \{1...n\} \rightarrow S$ for some set S. Typically sequences are notated as x_j rather than $x(j)$ and are typically written as such:

$$[1, 2, 5, 1, 3]$$

Sequences in Python

We can implement sequences in Python through the use of the list data type.

```
>>> s = [1, 2, 5, 1, 3]          # Create a list
>>> s[4]                         # Retrieve item at index 4 (with left most index being 0
3
>>> s[2] = 7                     # Replace item at index 2 with the value 7
>>> s.append(9)                  # Will append the value 9 to the end of the list
>>> s.pop()                      # Will remove and return the last item in the list
9
>>> len(s)                       # Returns the length of a list
5
>>> 6 in s                       # Check if an item is in a list
False
>>> a, b, c = s                  # Can unpack, or spread, the values of a list across a s
>>> print(a, b, c)
1 2 5
>>> d = {x : x*x for x in range(1,10) if x % 2 == 0}
{2: 4, 4: 16, 6: 36, 8: 64}
```

More on Tuples in Python

```
>>> t = (1, 2)         # Create a tuple
>>> s = (1,)           # Tuples with a single element need a leading comma
>>> u = t + (3,)       # Concatenation with tuples
(1, 2, 3)
>>> u[2]               # Access the item in a tuple at the provided index
3
>>> a, b, c = u        # Can unpack, or spread, the values of a tuple across a set of va
>>> print(a, b, c)
1 2 3
```

Tuples vs Sequences

Tuples and sequences are two very similar types and are both ways of expressing the same thing. Generally we use tuples for short, fixed length data where each position often has a different meaning and lists for variable length data.

In Python:

- Lists are mutable while tuples are immutable
- Use lists for variable length data
- Use tuples for fixed length data where the entries are related (e.g. x-y coordinates)

Mutability

In programming some objects are able to change while some are static. We call objects that can change over time mutable while objects that can't immutable. Operations on immutable types return a new object while operations on mutable types change the object.

Hashability

Hashable types can be identified by a hash value which is used internally by certain containers.

- Mutable types are not hashable
- Immutable types are generally, but not always, hashable
- immutable containers are hashable only if they contain only hashable elements

Week 7: Graphs

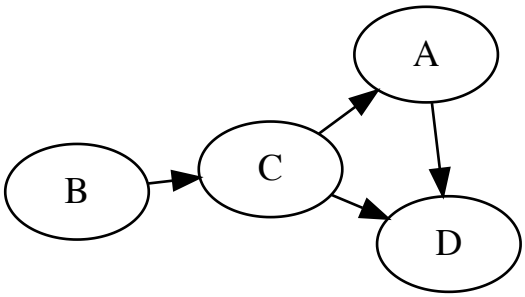
Graphs

A graph G is a pair (V, E) where the set V is the vertices or nodes of G and the set $E \subseteq V \times V$ is the edges of G. For every $(u, v) \in E$, we also have $(v, u) \in E$ as well as there being no edges such that (v, v). This means that graphs have an irreflexive, symmetric relation.

The graph described above is called a loop-free undirected graph which refers to its irreflexivity and symmetry.

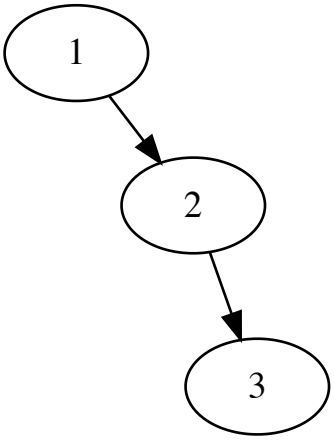
Graph Visualisation

We generally visualise graphs as a set of points (vertices) connect by lines (edges). The arrangement of the vertices doesn't matter and can be chosen arbitrarily.



An example of this including our notation could be:

$$V \in \{1, 2, 3\} \quad E = \{(1, 2), (2, 1), (2, 3), (3, 2)\}$$



Graph Definitions

There are a few rules and definitions we can follow when it comes to graphs:

- If $(u, v) \in E$ then we can say that u and v are adjacent and are neighbours.
- The neighbourhood of a vertex u is the set of u's neighbours:

$$N_G(u) = \{v \in V : (u, v) \in E\}$$

- The neighbourhood of a set of vertices $S \subseteq V$ is the set of all neighbours of all vertices in U:

$$N_G(S) = \{v \in V : \exists u \in S, (u, v) \in E\}$$

- If $e = (u, v) \in E$ then we can say that e is incident with u and v
- Due to our graphs being symmetric, that being for each $(u, v) \in E$ we also have $(v, u) \in E$, we can call these pairs the same edge and when counting they are considered as one.

Graphs in Python

```
V = { 1, 2, 3, 4, 5 }
E = { (1, 2), (2,1), (2,3), (3,2), (3,1), (1,3), (1,4), (4,1), (4,5), (5,4) }
G = (V, E)

def N(G, u):
    V, E = G
    return { v for v in V if (u,v) in E }

def NS(G, S):
    V, E = G
    return { v for v in V for u in S if (u,v) in E }

print("Vertices", V)
print("Edges", E)
print("Neighbours of 1", N(G, 1))
print("Neighbours of {1,5}", NS(G, {1,5}))
```

Degrees

The degree of a vertex u is the size of it's neighbourhood:

$$d(u) = |N_u|$$

For every edge (u, v), 1 is added to the degree of u and the degree of v:

$$2|E| = \sum_{u \in V} d(u)$$

Since the sum of the degrees is even, that must mean that there is an even number of vertices with odd degrees. We call this the handshaking lemma.

Paths

A path is a sequence of vertices $v_1, v_2, ..., v_j$ where:

- No vertex appears more than once

- v_k is adjacent to v_{k+1} for each $k = 1, ...j - 1$
- The length of the path is $j - 1$ (the number of edges)

We can also say that an edge (s, t) is in a path of $s = v_k$ and $t = v_{k+1}$ for some k.

Cycles

A cycle is similar to a path however we start and end at the same place. This is equivalent to saying that $v_1 = v_j$.

- The length of the cycle is $j - 1$ (the number of edges and number of vertices)
- Due to a cycle essentially being a loop, we don't care about which vertex is the starting point as A, B, C, A is the same as B, C, A, B
- Sometimes cycles don't duplicate the starting/ending vertex and instead just state "the cycle A, B, C" as opposed to "the path A, B, C".

Connectedness

If there exists a path from u to v for every $u, v \in V$, then we can say that G is connected. If this is false then we can say that G is disconnected. We can also say that, if for every $u, v \in S \subseteq V$ there exists a path between u and v and there are no paths to any vertices outside of S, then S is called a connected component of G.

Distance

Finally we can say the distance between vertices u and v is the length of the shortest path from u to v

Calculating Distances

$$V_j = \begin{cases} V & : j = 0 \\ V_{j-1} \setminus D_{j-1} & : j > 1 \end{cases}$$

$$D_j = \begin{cases} \{u\} & : j = 0 \\ N_{V_j}(D_{j-1}) & : j > 0 \end{cases}$$

```
V = { 1, 2, 3, 4, 5 }
E = { (1, 2), (2,1), (2,3), (3,2), (3,1), (1,3), (1,4), (4,1), (4,5), (5,4) }
G = (V, E)

def N(G, u):
    V, E = G
    return { v for v in V if (u,v) in E }

def NS(V, E, S):
    return { v for v in V for u in S if (u,v) in E }

def distanceClasses(V, E, u):
    V0 = V
    D = [ { u } ]
    return distanceClassesR(V0, E, D)

def distanceClassesR(V, E, D):
    Vnew = V - D[-1]

    if len(Vnew) == 0:
            return D

    Dnew = D + [ NS(Vnew, E, D[-1]) ]
    return distanceClassesR(Vnew, E, Dnew)

print(distanceClasses(V, E, 1))
```

Bipartite Graphs

A bipartite graph is a set of graph vertices split into two disjoint sets (A, B) in such a way that no two graph vertices within the same set are adjacent. If this is the case (A, B) can be called a bipartition of G.

In the distance finding method we simply set $A = D_0 \cup D_2 \cup D_4 ...$ and $B = D_1 \cup D_3 \cup D_5 ....$ Then if no two vertices in the same distance set (D_j) are adjacent we can call (A, B) a bipartition of G. We can also say a graph G is bipartite if and only if G has no cycles of odd length.

Distance Finding Revisited

We can summarise the shortest paths algorithm as:

- Set $D_0 = \{u\}$ (starting vertex)
- Set $D_1 =$ neighbours of u
- Set $D_2 =$ neighbours of d_1 that we haven't seen before
- ...

In this algorithm we process vertices and add them to some D_j in order of distance where the closes vertices to u come first.

Breadth First Traversal/Search

In the breadth first traversal algorithm we process the vertices in a graph in order of distance from some initial vertex. The difference here compared to our previous distance calculating algorithm is that we do some processing on the vertices and we don't bother to keep track of the sets D_j. So if we step through this algorithm:

- Set $D_0 = \{u\}$ and process u
- Set $D_1 =$ neighbours of u, and process them
- Set $D_2 =$ neighbours of d_1 that we haven't seen before and process them
- ...

```
V = { 1, 2, 3, 4, 5 }
E = { (1, 2), (2,1), (2,3), (3,2), (3,1), (1,3), (1,4), (4,1), (4,5), (5,4) }
G = (V, E)

def NS(V, E, S):
    return { v for v in V for u in S if (u,v) in E }

def BFS(V, E, u):
    D = {u}
    BFSR(V, E, D)

def BFSR(V, E, D):
    for v in D:
            print(v)
    Vnew = V - D
    if len(Vnew) == 0: return
    Dnew = NS(Vnew, E, D)
    BFSR(Vnew, E, Dnew)
```

Depth First Traversal

In the depth first traversal algorithm we pick the path that will take us the furthest from u and then when we can't go any further we backtrack and repeat until all paths are traversed.

```
V = { 1, 2, 3, 4, 5 }
E = { (1, 2), (2,1), (2,3), (3,2), (3,1), (1,3), (1,4), (4,1), (4,5), (5,4) }
G = (V, E)

def N(G, u):
    V, E = G
    return { v for v in V if (u,v) in E }

def depthFirst(V, E, u):
    T = {u}                                 # Set of vertices already seen
    depthFirstR(V, E, u, T)

def depthFirstR(V, E, u, T):
    print(u)                                # Process vertex u
    if len(T) == len(V): return T           # Check if we've seen all vertices
    Nu = N((V, E), u) - T                   # Neighbours not already seen
    T.update(Nu)                            # Update set of vertices already seen
    for v in Nu:
        T.update(depthFirstR(V, E, v, T))   # Add vertices seen
    return T
```

Week 8: Trees

Trees

A tree is a connected graph that does not include cycles. Some important things to note about trees is that:

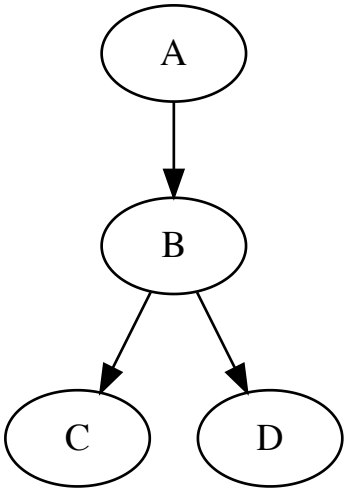
- A tree will always have $|V| - 1$ edges
- There will always be a unique path between any two vertices in a tree

We call a graph that has disconnections without any cycles a forest. In these types of graphs each connected component is a tree.

Rooted Trees

In a tree graph we can identify a special vertex as the root. This allows us to gain some additional organisation to our tree graph.

- The parent of the vertex is the first vertex in the unique path to the root
- The ancestors of a vertex is its parent, its parent's parent and so on until we reach the root
- The root vertex has no parents
- Any vertex in our graph can be the root therefore we must specify it.
- We call the parent of v $P(v)$



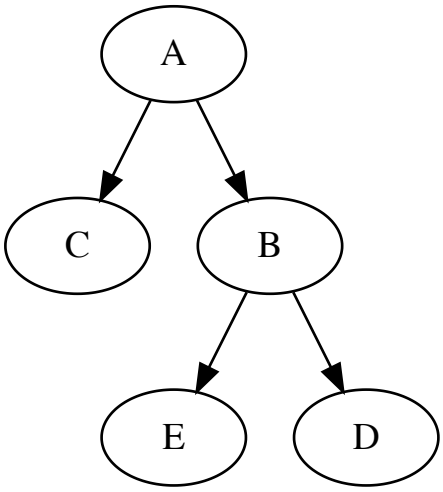
Recursive Definition of Ancestors

Let's say we have vertex v, we would then call all ancestors of v $A(v)$.

- If v is the root, then $A(v) = \varnothing$
- If v is not the root, then $A(v) = \{P(v)\} \cup A(P(v))$

In the below example, the root is chosen to be A with:

- B's ancestors are A
- D's ancestors are A and B



Rooted Trees 2

We can also say that:

- A vertex's children are all of its neighbours excluding its parent
- All descendants of a vertex are considered its children, all of its children's children and so on

- We call a vertex without children a leaf

Recursive Definition of Descendents

Let's say we have vertex v, we would then call the set of children of v $C(v)$ and the set of descendents $D(v)$.

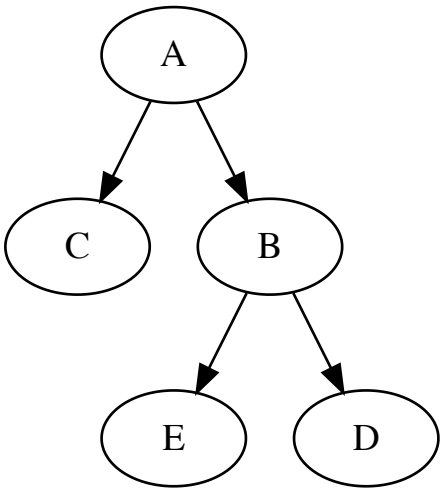
- If v is the root, then $C(v) = N_v$, otherwise $C(v) = N_v \setminus \{P(v)\}$

We can caluclate the descendents by:

$$D(v) = C(v) \cup \bigcup_{u \in C(v)} D(u)$$

So looking back on the graph we saw earlier we can now say that:

- A's children are C and B
- B's descendants are E and D
- The leaves are C, E, and D



Sub-Graphs

A sub-graph is a smaller graph contained inside of another graph. We can say the sub-graph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

An induced sub-graph is a sub-graph E' which contains all edges of E that are incident with the vertices contained in V'. We write this as $G(V')$.

$$G(V') := (V', \{(s, t) \in E : s, t \in V'\})$$

Sub-Trees

As well as sub-graphs we also have sub-trees. If we have a rooted tree, we can then define sub-trees.

- A sub-tree of a rooted tree on vertex v is the induced sub-graph on v and its descendent
- A sub-tree will always be a rooted tree, in this instance v is the root

Spanning Tree

We can identify trees inside of graphs.

- A sub-graph $T = (V, E')$ of a connected graph $G = (V, E)$ which is also a tree is called a spanning tree.
- If the graph has a spanning tree then we know it must also be connected. This means that the spanning tree gives a unique path between any two vertices in G

BFS

```
# Code referenced from Week 8 Topic 2

def NS(V, E, S):
    return { v for u in S for v in V if (u, v) in E }

def N(V, E, u):
    return { v for v in V if (u, v) in E }

# @brief Helper method to get the next item in a set
#
# @param S Set of items
# @returns the next item in the set
def arbitrary(S):
    return next(iter(S))

# @brief Function to return the parent of each vertex
#
# @param V A graph
# @param E Set of edges
# @param r The root
# @returns parents A set of all found parents
def spanTree(V, E, r):
    parents = { r: None }
    spanTreeR(V - {r}, E, {r}, parents)
    return parents

# @brief Recursive version of the function `spanTree`
#
# @param V A sub-graph
# @param E Set of edges
# @param D The previous distance class
# @param parents The parents of all found vertices
def spanTreeR(V, E, D, parents):
    Dnew = NS(V, E, D)
    if not Dnew: return
    for v in Dnew:
        parents[v] = arbitrary(N(D, E, v))
    spanTreeR(V - Dnew, E, Dnew, parents)
```

Parents to Paths

```
# Code referenced from Week 8 Topic 2

def path(parents, v):
    u = parents[v]
    if u == None: return [v]
    return path(parents, u) + [v]
```