COMP110: Principles of Computing

asic Principles for Computation

FALMOUTH
UNIVERSITY

Worksheet 1

Worksheet 1

- Due **tomorrow**!
- Support workshop **later this afternoon**

Worksheet submission

- `https://github.com/falmouth-games-academy/comp110-worksheet-1`
- If the YouTube uploader within SpaceChem does not work, save the video to disk and upload to YouTube manually
- If this also doesn't work, use e.g. OBS to record videos and upload them (I only need to see your solutions running)
- If **this** doesn't work, record your screen with your phone!

Worksheet submission

- `https://github.com/falmouth-games-academy/comp110-worksheet-1`
- Don't forget to upload your **save file**, add your **playlist link**, and open a **pull request**
- Please make sure I can figure out who you are! If it's not obvious from your username, please put your **name or student number** in the title of your pull request
- Any problems, send me a message via email or slack

Binary notation

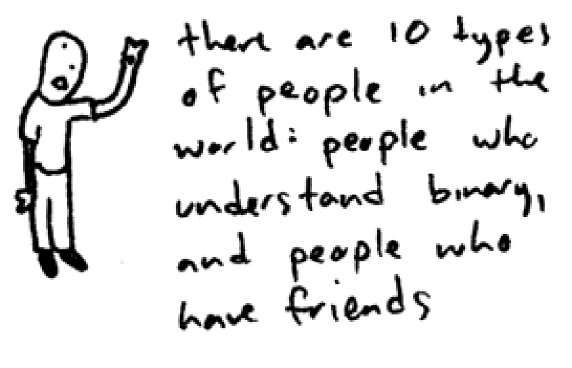


Image credit: http://www.toothpastefordinner.com

How we write numbers

How we write numbers

- We write numbers in **base 10**

How we write numbers

- We write numbers in **base 10**
- We have 10 **digits**: $0, 1, 2, \ldots, 8, 9$

How we write numbers

- We write numbers in **base 10**
- We have 10 **digits**: $0, 1, 2, \ldots, 8, 9$
- When we write 6397, we mean:

How we write numbers

- We write numbers in **base 10**
- We have 10 **digits**: $0, 1, 2, \ldots, 8, 9$
- When we write 6397, we mean:
  - Six thousand, three hundred and ninety seven

How we write numbers

- We write numbers in **base 10**
- We have 10 **digits**: $0, 1, 2, \dots, 8, 9$
- When we write 6397, we mean:
  - Six thousand, three hundred and ninety seven
  - (Six thousands) and (three hundreds) and (nine tens) and (seven)

How we write numbers

- We write numbers in **base 10**
- We have 10 **digits**: $0, 1, 2, \ldots, 8, 9$
- When we write 6397, we mean:
  - Six thousand, three hundred and ninety seven
  - (Six thousands) and (three hundreds) and (nine tens) and (seven)
  - $(6 \times 1000) + (3 \times 100) + (9 \times 10) + (7)$

How we write numbers

- We write numbers in **base 10**
- We have 10 **digits**: $0, 1, 2, \ldots, 8, 9$
- When we write 6397, we mean:
  - Six thousand, three hundred and ninety seven
  - (Six thousands) and (three hundreds) and (nine tens) and (seven)
  - $(6 \times 1000) + (3 \times 100) + (9 \times 10) + (7)$
  - $\left(6 \times 10^3\right) + \left(3 \times 10^2\right) + \left(9 \times 10^1\right) + \left(7 \times 10^0\right)$

How we write numbers

- We write numbers in **base 10**
- We have 10 **digits**: $0, 1, 2, \ldots, 8, 9$
- When we write 6397, we mean:
  - Six thousand, three hundred and ninety seven
  - (Six thousands) and (three hundreds) and (nine tens) and (seven)
  - $(6 \times 1000) + (3 \times 100) + (9 \times 10) + (7)$
  - $(6 \times 10^3) + (3 \times 10^2) + (9 \times 10^1) + (7 \times 10^0)$
Thousands	Hundreds	Tens	Units
6	3	9	7

Binary

Binary

- Binary notation works the same, but is **base 2** instead of **base 10**

Binary

- Binary notation works the same, but is **base 2** instead of **base 10**
- We have 2 **digits**: 0, 1

Binary

- Binary notation works the same, but is **base 2** instead of **base 10**
- We have 2 **digits**: 0, 1
- When we write 10001011 in binary, we mean:

Binary

- Binary notation works the same, but is **base 2** instead of **base 10**
- We have 2 **digits**: 0, 1
- When we write 10001011 in binary, we mean:
  $$(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

Binary

- Binary notation works the same, but is **base 2** instead of **base 10**
- We have 2 **digits**: 0, 1
- When we write 10001011 in binary, we mean:
  $$(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4)$$
  $$+ (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$
  $$= 2^7 + 2^3 + 2^1 + 2^0$$

Binary

- Binary notation works the same, but is **base 2** instead of **base 10**
- We have 2 **digits**: 0, 1
- When we write 10001011 in binary, we mean:

$$\begin{aligned} &(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) \\ &+ (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &= 2^7 + 2^3 + 2^1 + 2^0 \\ &= 128 + 8 + 2 + 1 \text{ (base 10)} \end{aligned}$$

Binary

- Binary notation works the same, but is **base 2** instead of **base 10**
- We have 2 **digits**: 0, 1
- When we write 10001011 in binary, we mean:

$$
\begin{aligned}
&(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) \\
&+ (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\
&= 2^7 + 2^3 + 2^1 + 2^0 \\
&= 128 + 8 + 2 + 1 \text{ (base 10)} \\
&= 139 \text{ (base 10)}
\end{aligned}
$$

Why binary?

Why binary?

- Modern computers are **digital**

Why binary?

- Modern computers are **digital**
- Based on the flow of current in a circuit being either **on** or **off**

Why binary?

- Modern computers are **digital**
- Based on the flow of current in a circuit being either **on** or **off**
- Hence it is natural to store and operate on numbers in base 2

Why binary?

- Modern computers are **digital**
- Based on the flow of current in a circuit being either **on** or **off**
- Hence it is natural to store and operate on numbers in base 2
- The binary digits 0 and 1 correspond to **off** and **on** respectively

Converting to binary

https://www.youtube.com/watch?v=OezK_zTyvAQ

Bits, bytes and words

Bits, bytes and words

- A **bit** is a binary digit

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary
- A **word** is the number of bits that the CPU works with at once

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary
- A **word** is the number of bits that the CPU works with at once
  - 32-bit CPU: 32 bits = 1 word

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary
- A **word** is the number of bits that the CPU works with at once
  - 32-bit CPU: 32 bits = 1 word
  - 64-bit CPU: 64 bits = 1 word

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary
- A **word** is the number of bits that the CPU works with at once
  - 32-bit CPU: 32 bits = 1 word
  - 64-bit CPU: 64 bits = 1 word
- An n-bit word can store a number between 0 and $2^n - 1$

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary
- A **word** is the number of bits that the CPU works with at once
  - 32-bit CPU: 32 bits = 1 word
  - 64-bit CPU: 64 bits = 1 word
- An n-bit word can store a number between 0 and $2^n - 1$
  - $2^{16} - 1 = 65,535$

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary
- A **word** is the number of bits that the CPU works with at once
  - 32-bit CPU: 32 bits = 1 word
  - 64-bit CPU: 64 bits = 1 word
- An n-bit word can store a number between 0 and $2^n - 1$
  - $2^{16} - 1 = 65,535$
  - $2^{32} - 1 = 4,294,967,295$

Bits, bytes and words

- A **bit** is a binary digit
  - Can store a 0 or 1 (i.e. a boolean value)
  - The smallest possible unit of information
- A **byte** is 8 **bits**
  - Can store a number between 0 and 255 in binary
- A **word** is the number of bits that the CPU works with at once
  - 32-bit CPU: 32 bits = 1 word
  - 64-bit CPU: 64 bits = 1 word
- An n-bit word can store a number between 0 and $2^n - 1$
  - $2^{16} - 1 = 65,535$
  - $2^{32} - 1 = 4,294,967,295$
  - $2^{64} - 1 = 18,446,744,073,709,551,615$

Other units

Other units

- A **nibble** is 4 **bits**

Other units

- A **nibble** is 4 **bits**
- A **kilobyte** is 1000 or 1024 **bytes**

Other units

- A **nibble** is 4 **bits**
- A **kilobyte** is 1000 or 1024 **bytes**
  - $10^3 = 1000 \approx 1024 = 2^{10}$

Other units

- A **nibble** is 4 **bits**
- A **kilobyte** is 1000 or 1024 **bytes**
  - $10^3 = 1000 \approx 1024 = 2^{10}$
- A **megabyte** is 1000 or 1024 **kilobytes**

Other units

- A **nibble** is 4 **bits**
- A **kilobyte** is 1000 or 1024 **bytes**
  - $10^3 = 1000 \approx 1024 = 2^{10}$
- A **megabyte** is 1000 or 1024 **kilobytes**
- A **gigabyte** is 1000 or 1024 **megabytes**

Other units

- A **nibble** is 4 **bits**
- A **kilobyte** is 1000 or 1024 **bytes**
  - $10^3 = 1000 \approx 1024 = 2^{10}$
- A **megabyte** is 1000 or 1024 **kilobytes**
- A **gigabyte** is 1000 or 1024 **megabytes**
- A **terabyte** is 1000 or 1024 **gigabytes**

Other units

- A **nibble** is 4 **bits**
- A **kilobyte** is 1000 or 1024 **bytes**
  - $10^3 = 1000 \approx 1024 = 2^{10}$
- A **megabyte** is 1000 or 1024 **kilobytes**
- A **gigabyte** is 1000 or 1024 **megabytes**
- A **terabyte** is 1000 or 1024 **gigabytes**
- ...

Addition with carry

In base 10:

$$
\begin{array}{ccccc}
 & 1 & 2 & 3 & 4 \\
+ & 5 & 6 & 7 & 8 \\
\hline
\end{array}
$$

Addition with carry

In base 10:

$$
\begin{array}{rcccc}
 & 1 & 2 & 3 & 4 \\
+ & 5 & 6 & 7_1 & 8 \\
\hline
 & & & & 2
\end{array}
$$

Addition with carry

In base 10:

$$\begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ + & 5 & 6_1 & 7_1 & 8 \\ \hline & & & 1 & 2 \end{array}$$

Addition with carry

In base 10:

$$\begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ + & 5 & 6_1 & 7_1 & 8 \\ \hline & & 9 & 1 & 2 \end{array}$$

Addition with carry

In base 10:

$$
\begin{array}{ccccc}
 & 1 & 2 & 3 & 4 \\
+ & 5 & 6_1 & 7_1 & 8 \\
\hline
 & 6 & 9 & 1 & 2
\end{array}
$$

Addition with carry

In base 2:

$$\boxed{1 + 1 = 10 \qquad 1 + 1 + 1 = 11}$$

$$\begin{array}{ccccccccc} & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

Addition with carry

In base 2:

$$\boxed{1+1=10 \qquad 1+1+1=11}$$

$$\begin{array}{cccccccccc} & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ \hline & & & & & & & & 1 \end{array}$$

Addition with carry

In base 2:

$$\boxed{1+1=10 \qquad 1+1+1=11}$$

$$\begin{array}{cccccccccc} & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 0 & 0 & 1_1 & 1 & 1 \\ \hline & & & & & & & 0 & 1 \end{array}$$

Addition with carry

In base 2:

$$\boxed{1 + 1 = 10 \qquad 1 + 1 + 1 = 11}$$

$$\begin{array}{ccccccccc} & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 0 & 0_1 & 1_1 & 1 & 1 \\ \hline & & & & & & 1 & 0 & 1 \end{array}$$

Addition with carry

In base 2:

$$\boxed{1 + 1 = 10 \qquad 1 + 1 + 1 = 11}$$

$$\begin{array}{ccccccccc} & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 0_1 & 0_1 & 1_1 & 1 & 1 \\ \hline & & & & & 0 & 1 & 0 & 1 \end{array}$$

Addition with carry

In base 2:

$$\boxed{1+1=10 \qquad 1+1+1=11}$$

$$\begin{array}{cccccccccc}
 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
+ & 0 & 0 & 1 & 0_1 & 0_1 & 1_1 & 1 & 1 \\
\hline
 & & & & 1 & 0 & 1 & 0 & 1
\end{array}$$

Addition with carry

In base 2:

$$\boxed{1 + 1 = 10 \qquad 1 + 1 + 1 = 11}$$

$$\begin{array}{ccccccccc} & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ + & 0 & 0_1 & 1 & 0_1 & 0_1 & 1_1 & 1 & 1 \\ \hline & & & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

Addition with carry

In base 2:

$$\boxed{1 + 1 = 10 \qquad 1 + 1 + 1 = 11}$$

$$\begin{array}{ccccccccc}
 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
+ & 0_1 & 0_1 & 1 & 0_1 & 0_1 & 1_1 & 1 & 1 \\
\hline
 & & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}$$

Addition with carry

In base 2:

$$\boxed{1 + 1 = 10 \qquad 1 + 1 + 1 = 11}$$

$$\begin{array}{ccccccccc}
 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
+ & 0_1 & 0_1 & 1 & 0_1 & 0_1 & 1_1 & 1 & 1 \\
\hline
 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}$$

Hexadecimal notation

Hexadecimal notation

- Other number bases than 2 and 10 are also useful

Hexadecimal notation

- Other number bases than 2 and 10 are also useful
- Hexadecimal is **base 16**

Hexadecimal notation

- Other number bases than 2 and 10 are also useful
- Hexadecimal is **base 16**
- Uses extra digits:
  - `A`=10, `B`=11, ..., `F`=15

Hexadecimal notation

- Other number bases than 2 and 10 are also useful
- Hexadecimal is **base 16**
- Uses extra digits:
  - A=10, B=11, …, F=15

Hex	Dec	Hex	Dec	Hex	Dec
00	0	10	16	F0	240
01	1	11	17	F1	241
⋮	⋮	⋮	⋮	⋮	⋮
09	9	19	25	F9	249
0A	10	1A	26	FA	250
0B	11	1B	27	FB	251
0C	12	1C	28	FC	252
0D	13	1D	29	FD	253
0E	14	1E	30	FE	254
0F	15	1F	31	FF	255

2’s Complement

Modular arithmetic

Modular arithmetic

- Arithmetic **modulo** N

Modular arithmetic

- Arithmetic **modulo** N
- Numbers "wrap around" between 0 and $N-1$

Modular arithmetic

- Arithmetic **modulo** N
- Numbers "wrap around" between 0 and $N - 1$
- E.g. modulo 16:

Modular arithmetic

- Arithmetic **modulo** N
- Numbers "wrap around" between 0 and $N - 1$
- E.g. modulo 16:
  - $14 + 7 = 5$

Modular arithmetic

- Arithmetic **modulo** N
- Numbers "wrap around" between 0 and $N-1$
- E.g. modulo 16:
  - $14 + 7 = 5$
  - $4 - 7 = 13$

Modulo operator

Modulo operator

- Present in many programming languages (including C++, C#, Python) as `%`

Modulo operator

- Present in many programming languages (including C++, C#, Python) as `%`
- `a % b` gives the **remainder** of `a` divided by `b`

Modulo operator

- Present in many programming languages (including C++, C#, Python) as `%`
- `a % b` gives the **remainder** of `a` divided by `b`
- E.g. `21 % 16` gives 5

Modulo operator

- Present in many programming languages (including C++, C#, Python) as `%`
- `a % b` gives the **remainder** of `a` divided by `b`
- E.g. `21 % 16` gives 5
- Useful for wrapping around e.g. loop indexes or screen coordinates

2's complement

2's complement

- How can we represent negative numbers in binary?

2's complement

- How can we represent negative numbers in binary?
- Represent them modulo 2^n (for n bits)

2's complement

- How can we represent negative numbers in binary?
- Represent them modulo 2^n (for n bits)
- I.e. represent $-a$ as $2^n - a$

2's complement

- How can we represent negative numbers in binary?
- Represent them modulo 2^n (for n bits)
- I.e. represent $-a$ as $2^n - a$
- Instead of an n-bit number ranging from 0 to $2^n - 1$, it ranges from -2^{n-1} to $+2^{n-1} - 1$

2's complement

- How can we represent negative numbers in binary?
- Represent them modulo 2^n (for n bits)
- I.e. represent $-a$ as $2^n - a$
- Instead of an n-bit number ranging from 0 to $2^n - 1$, it ranges from -2^{n-1} to $+2^{n-1} - 1$
- E.g. 16-bit number ranges from -32768 to $+32767$

2's complement

- How can we represent negative numbers in binary?
- Represent them modulo 2^n (for n bits)
- I.e. represent $-a$ as $2^n - a$
- Instead of an n-bit number ranging from 0 to $2^n - 1$, it ranges from -2^{n-1} to $+2^{n-1} - 1$
- E.g. 16-bit number ranges from -32768 to $+32767$
- Note that the left-most bit can be interpreted as a **sign** bit: 1 if negative, 0 if positive or zero

Converting to 2's complement

Converting to 2's complement

- Convert the absolute value to binary

Converting to 2's complement

- Convert the absolute value to binary
- Invert all the bits (i.e. change $0 \leftrightarrow 1$)

Converting to 2's complement

- Convert the absolute value to binary
- Invert all the bits (i.e. change $0 \leftrightarrow 1$)
- Add 1

Converting to 2's complement

- Convert the absolute value to binary
- Invert all the bits (i.e. change $0 \leftrightarrow 1$)
- Add 1
- (This is equivalent to subtracting the number from 2^n... why?)

Converting to 2's complement

- Convert the absolute value to binary
- Invert all the bits (i.e. change $0 \leftrightarrow 1$)
- Add 1
- (This is equivalent to subtracting the number from 2^n... why?)
- This is also the process for converting back from 2's complement, i.e. doing it twice should give the original number

Why 2's complement?

Why 2's complement?

- Allows all addition and subtraction to be carried out modulo 2^n without caring whether numbers are positive or negative

Why 2's complement?

- Allows all addition and subtraction to be carried out modulo 2^n without caring whether numbers are positive or negative
- In fact, subtraction can just be done as addition

Why 2's complement?

- Allows all addition and subtraction to be carried out modulo 2^n without caring whether numbers are positive or negative
- In fact, subtraction can just be done as addition
- I.e. $a - b$ is the same as $a + (-b)$, where a and $-b$ are just n-bit numbers

Worksheet 2

Worksheet 2

Due next Friday!
Online quiz on LearningSpace

FALMOUTH
UNIVERSITY

Algorithms

What is an algorithm?

What is an algorithm?

A **sequence of instructions** which can be followed **step by step** to perform a **(computational) task**.

Algorithms historically

Algorithms historically

- Named after Muhammad ibn Musa al-Khwarizmi (c. 780–850), Persian mathematician

Algorithms historically

- Named after Muhammad ibn Musa al-Khwarizmi (c. 780–850), Persian mathematician
- Used in mathematics to describe steps for calculations

Algorithms historically

- Named after Muhammad ibn Musa al-Khwarizmi (c. 780–850), Persian mathematician
- Used in mathematics to describe steps for calculations
  - E.g. Euclid's algorithm for finding the greatest common divisor of two numbers

Algorithms historically

- Named after Muhammad ibn Musa al-Khwarizmi (c. 780–850), Persian mathematician
- Used in mathematics to describe steps for calculations
  - E.g. Euclid's algorithm for finding the greatest common divisor of two numbers
- Computers developed as machines for carrying out mathematical algorithms

Programs vs algorithms

Programs vs algorithms

- A program is **specific** to a particular programming language and/or machine

Programs vs algorithms

- A program is **specific** to a particular programming language and/or machine
- An algorithm is **general**

Programs vs algorithms

- A program is **specific** to a particular programming language and/or machine
- An algorithm is **general**
- An algorithm must be **implemented** as a program before a computer can run it

Programs vs algorithms

- A program is **specific** to a particular programming language and/or machine
- An algorithm is **general**
- An algorithm must be **implemented** as a program before a computer can run it
- An algorithm generally performs **one task**, whereas a program may perform **many**

Programs vs algorithms

- A program is **specific** to a particular programming language and/or machine
- An algorithm is **general**
- An algorithm must be **implemented** as a program before a computer can run it
- An algorithm generally performs **one task**, whereas a program may perform **many**
  - E.g. Microsoft Word is not an algorithm, but it implements many algorithms

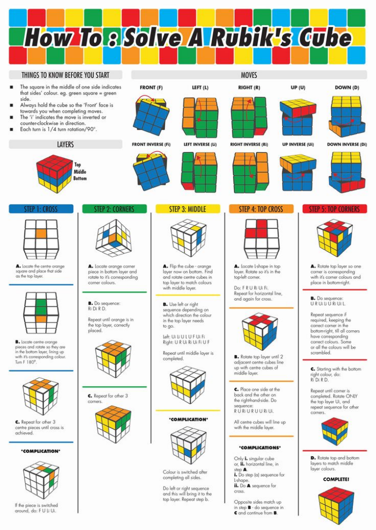
Programs vs algorithms

- A program is **specific** to a particular programming language and/or machine
- An algorithm is **general**
- An algorithm must be **implemented** as a program before a computer can run it
- An algorithm generally performs **one task**, whereas a program may perform **many**
  - E.g. Microsoft Word is not an algorithm, but it implements many algorithms
  - E.g. it implements an algorithm for determining where to break a line of text, how much space to add to centre a line, etc.

Algorithms outside computing

1. Preheat the oven to 180C, gas 4.
2. Beat together the eggs, flour, caster sugar, butter and baking powder until smooth in a large mixing bowl.
3. Put the cocoa in separate mixing bowl, and add the water a little at a time to make a stiff paste. Add to the cake mixture.
4. Turn into the prepared tins, level the top and bake in the preheated oven for about 20-25 mins, or until shrinking away from the sides of the tin and springy to the touch.
5. Leave to cool in the tin, then turn on to a wire rack to become completely cold before icing.
6. To make the icing: measure the cream and chocolate into a bowl and carefully melt over a pan of hot water over a low heat, or gently in the microwave for 1 min (600w microwave). Stir until melted, then set aside to cool a little and to thicken up.
7. To ice the cake: spread the apricot jam on the top of each cake. Spread half of the ganache icing on the top of the jam on one of the cakes, then lay the other cake on top, sandwiching them together.
8. Use the remaining ganache icing to ice the top of the cake in a swirl pattern. Dust with icing sugar to serve.

Algorithms outside computing



Why algorithms?

Why algorithms?

- Allow for common computations to have **common solutions**

Why algorithms?

- Allow for common computations to have **common solutions**
- **Algorithm strategies** give widely applicable approaches for solving problems

Why algorithms?

- Allow for common computations to have **common solutions**
- **Algorithm strategies** give widely applicable approaches for solving problems
- Can **prove** mathematically that an algorithm does what it is supposed to

Why algorithms?

- Allow for common computations to have **common solutions**
- **Algorithm strategies** give widely applicable approaches for solving problems
- Can **prove** mathematically that an algorithm does what it is supposed to
- Can reason about the **complexity** (time, space etc) of an algorithm — and place **lower bounds** on the best possible algorithm

Why algorithms?

- Allow for common computations to have **common solutions**
- **Algorithm strategies** give widely applicable approaches for solving problems
- Can **prove** mathematically that an algorithm does what it is supposed to
- Can reason about the **complexity** (time, space etc) of an algorithm — and place **lower bounds** on the best possible algorithm
- **Computability** theory lets us reason about what computations are and are not possible