# 11: NUMERICAL REPRESENTATIONS

COMP110 PRINCIPLES OF COMPUTING

# WORKSHEETS

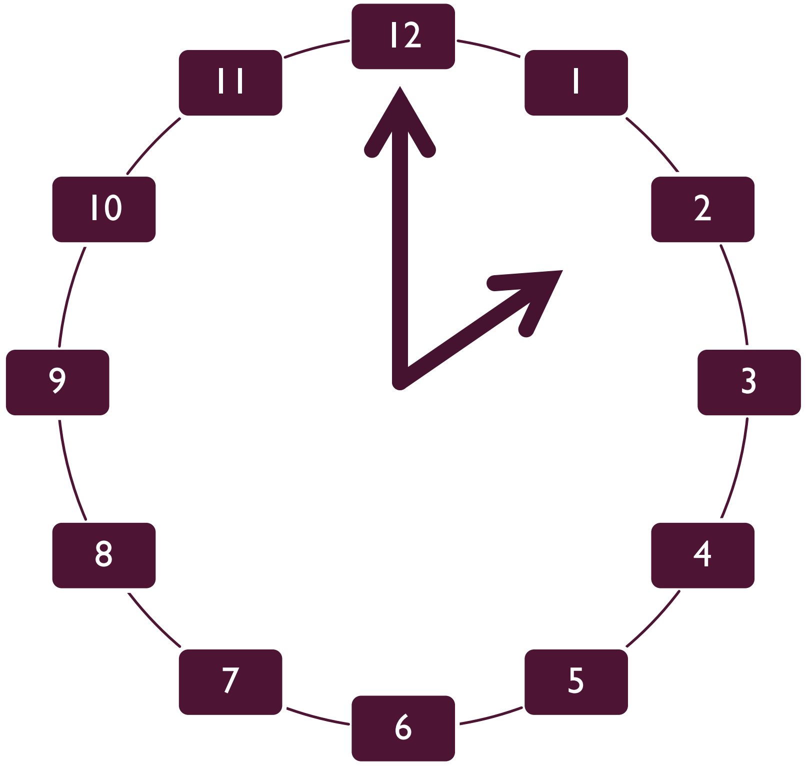
- **Worksheet 7 (Recursion):** due this Wednesday
- **Worksheet 8 (Floating point numbers and vectors):** due next Wednesday
- **Worksheet 9 (TIS-100):** released this Friday, due after xmas break
  - This worksheet will ask you to play through a few levels of TIS-100 (Zachtronics, 2015)
  - TIS-100 is currently in the Steam sale for £2.49 until 1st December (normal price £4.99)

# TODAY'S SESSION

- We have seen before how to represent **non-negative integers** in the digital computer, using **binary notation**
- Today we extend to two other useful types of numbers
  - Negative integers
  - Real numbers

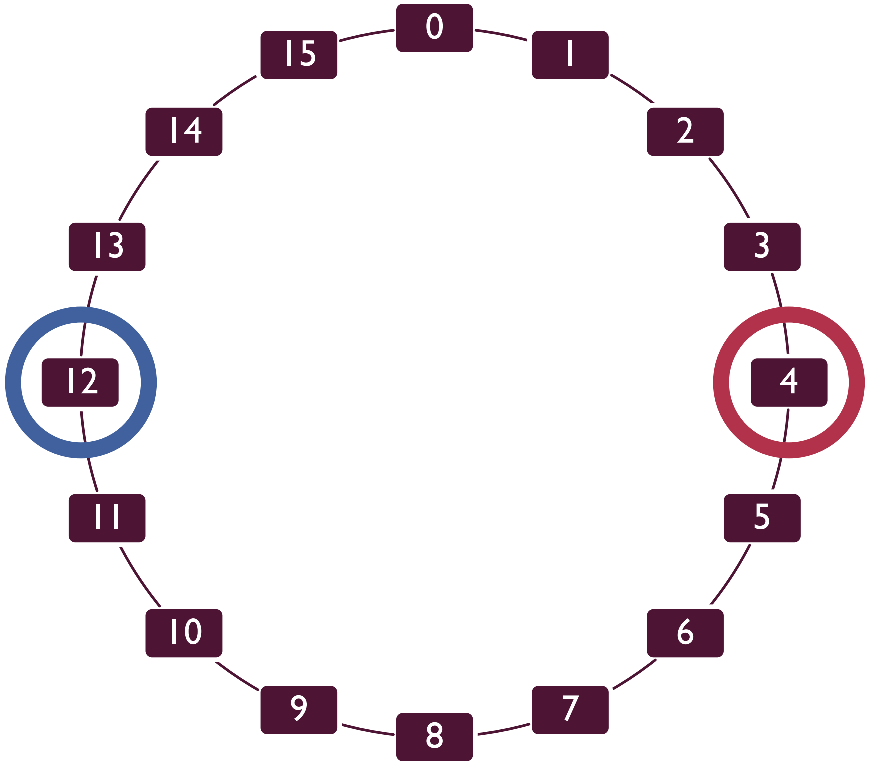
# MODULAR ARITHMETIC

# TELLING THE TIME




- On a 12 hour clock, hours wrap around between 1 and 12
- For example, 9pm + 5 hours = 2am

# MODULAR ARITHMETIC




- Arithmetic "modulo $N$"
- Numbers wrap around between $0$ and $N - 1$
- For example, modulo 16:
  - $12 + 6 = 2$
  - $4 - 5 = 15$

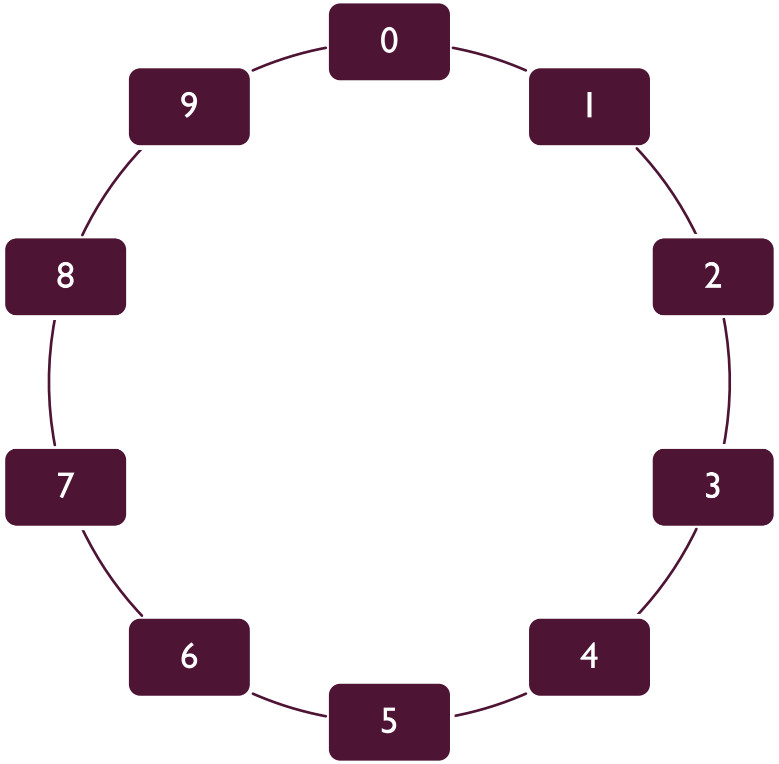
# MODULAR ARITHMETIC AND DIVISION

- Arithmetic modulo $N$: take the remainder from division by $N$
- For example:
  - $\mathbf{12 + 6 = 18}$
  - $\mathbf{18} \div \mathbf{16} = 1$, remainder $\mathbf{2}$
  - Equivalently, $\mathbf{18} = \mathbf{16} \times 1 + \mathbf{2}$
  - Therefore modulo $\mathbf{16}$, $\mathbf{12 + 6 = 2}$

# THE MODULO OPERATOR

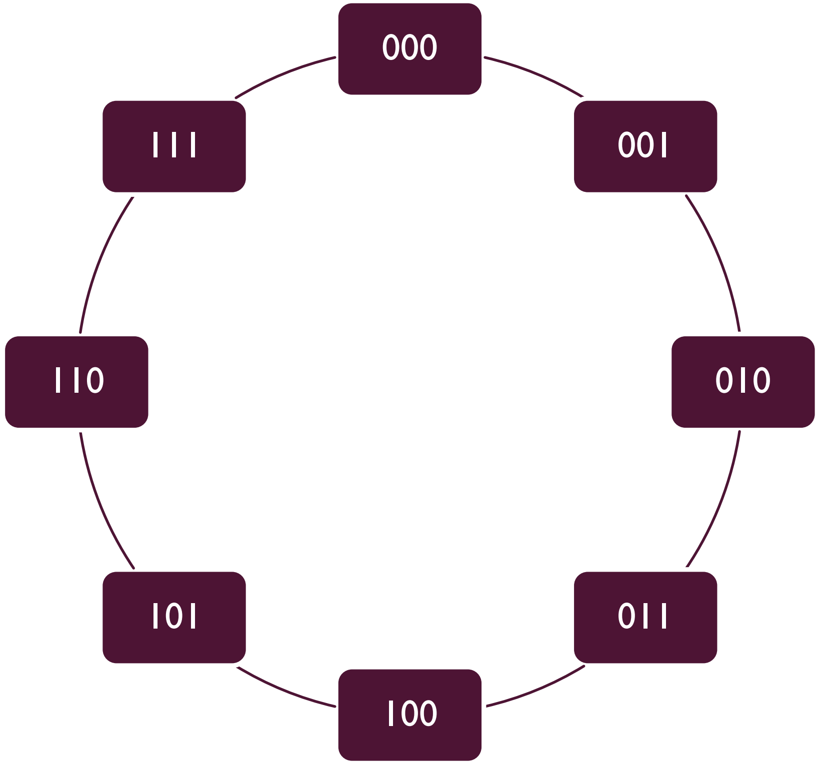
- Most languages (C++, C#, Python, Java, Javascript, …) have a % operator
- For positive numbers, $a \,\%\, b$ gives the **remainder** from dividing $a \div b$
- E.g. $18 \,\%\, 16 = 2$
- Useful for calculating **wrap-around** array indices, screen coordinates etc.

# MODULO 10, 100, …




- The remainder of dividing a positive number by 10 is just the **last digit** of the number
- E.g. $12 \% 10 = 2 \quad 327 \% 10 = 7$
- Similarly, modulo $100$, we take the last 2 digits, etc.

# MODULO 2, 4, 8, 16, …



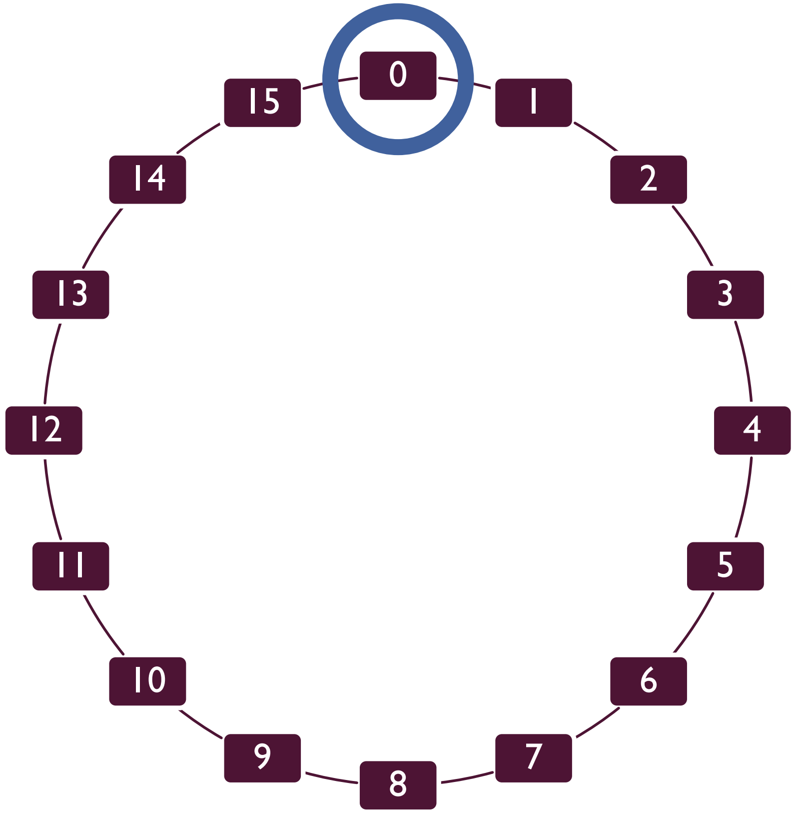

- The same principle applies to powers of 2, but in binary
- Modulo 2 = take the last bit
- Modulo 8 = take the last 3 bits
- Modulo $2^n$ = take the last $n$ bits

# OVERFLOW

- Suppose we add together two 32-bit integers
- What happens if the result is larger than $2^{32} - 1$?
- Answer: only the last 32 bits are kept – the rest are thrown away
- This is called an **overflow**
- This means that 32-bit arithmetic is arithmetic modulo $2^{32}$

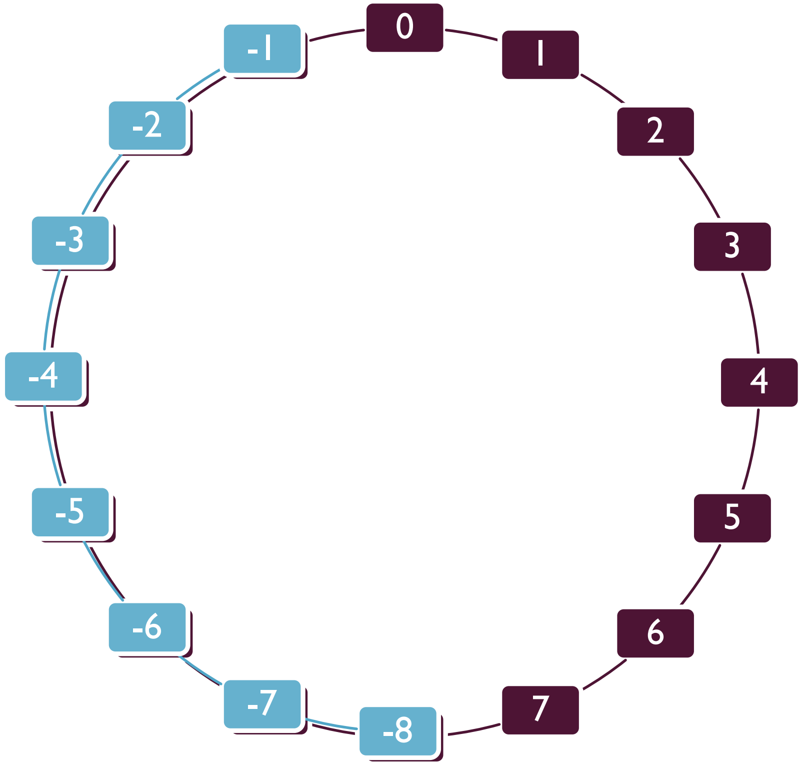
# 2'S COMPLEMENT

# WHAT ABOUT NEGATIVE NUMBERS?




- Negative numbers fit with our "wrapping around" picture as well
- E.g. modulo 16, $-1$ should correspond to $15$
- In general, modulo $N$, a negative number $-k$ is equivalent to $N - k$

# REPRESENTING SIGNED INTEGERS




- For **unsigned** integers (as we've seen before), $N$ bits are used to represent numbers from $0$ to $2^N - 1$
- For **signed** integers, we instead represent numbers from $-2^{N-1}$ to $2^{N-1} - 1$
- The "second half" of the range is now used to represent negative numbers
- This is called **2's complement**

# RANGES OF INTEGERS

| Bits | Unsigned range | | Signed range | |
|---|---|---|---|---|
| $N$ | $0$ | $2^N - 1$ | $-2^{N-1}$ | $2^{N-1} - 1$ |
| 8 | $0$ | $255$ | $-128$ | $127$ |
| 16 | $0$ | $65535$ | $-32768$ | $32767$ |
| 32 | $0$ | $4294967295$ | $-2147483648$ | $2147483647$ |

# WHY 2'S COMPLEMENT?

- Modulo $2^N$, subtracting $-k$ is the same as adding $2^N - k$
  - E.g. modulo 16:
  - $4 - 5 = -1 = 15$
  - $4 + (16 - 5) = 4 + 11 = 15$
- This means that the **same circuits** can be used to add and subtract signed and unsigned numbers
- Also, the "second half" of the $N$-bit numbers are exactly those where the first bit is $1$ – therefore the first bit becomes a "**sign bit**"

# SCIENTIFIC NOTATION

# INTEGER POWERS

If $a \neq 0$ is a real number and $b > 0$ is an integer, then

$$a^b = \underbrace{a \times a \times \cdots \times a}_{b \text{ times}}$$

$$a^0 = 1$$

$$a^{-b} = \frac{1}{a^b} = \frac{1}{\underbrace{a \times a \times \cdots \times a}_{b \text{ times}}}$$

# POWERS OF 10

$$10^6 = 1\underbrace{000\,000}_{6\text{ zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

$$10^{-1} = \frac{1}{10} = 0.1$$

$$10^{-6} = \frac{1}{1\,000\,000} = 0.\underbrace{000\,00}_{5\text{ zeroes}}1$$

# MULTIPLYING BY POWERS OF 10

- Multiplying by $10^n$ is the same as moving the decimal point $n$ places to the right (adding zeroes if necessary)
  - $3.14159 \times 10^2 = 314.159$
  - $4.2 \times 10^5 = 42000$
- If $n$ is negative, the decimal point moves to the left instead:
  - $123.45 \times 10^{-2} = 1.2345$
  - $42 \times 10^{-4} = 0.0042$

# SCIENTIFIC NOTATION

- A convenient way of writing very large or very small numbers

$$a \times 10^b$$

**Mantissa**
A real number, positive or negative, with $1 \leq |a| < 10$ (i.e. with a single non-zero digit before the decimal point)

**Exponent**
An integer, can be positive, negative or zero

# EXAMPLES

- Light year: $9.461 \times 10^{15}$ metres
- Planck's constant: $6.626 \times 10^{-34}$ Joules
- Socrative time! Student login → room code **FALCOMPED**

# SCIENTIFIC NOTATION IN CODE

- Instead of writing $\times 10$, we write e (for exponent)
- `double lightYear = 9.461e15;`
- `double planck = 6.626e-34;`

# FLOATING POINT NOTATION

# “DECIMAL” POINTS IN BINARY

- In decimal, the digits after the point represent negative powers of $10$
  - E.g. $0.45 = \frac{4}{10} + \frac{5}{100}$
- The same principle can be used in binary, but with powers of 2
  - E.g. $0.101 = \frac{1}{2} + \frac{0}{4} + \frac{1}{8}$

# FLOATING POINT NOTATION

- Like scientific notation, but in base 2 rather than base 10

**Mantissa**
A real number, positive or negative, with $1 \leq |a| < 2$ (i.e. with a single 1 before the point)

$$a \times 2^b$$

**Exponent**
An integer, can be positive, negative or zero

# STORING THE MANTISSA

- **IEEE754** is the most common standard for storing floating point numbers
- Mantissa is a positive or negative number, with a 1 before the point
- Sign is stored as a single bit (0 = positive, 1 = negative)
- We know the bit before the point is a 1, so it is not actually stored
- Instead we just store the bits after the point

# STORING THE EXPONENT

- Exponent is a signed integer
- Not stored in 2's complement!
- Instead, stored as an integer with a **bias** added
- E.g. bias = 127, we store exponent + 127
- Exponents of 000…0 and 111…1 are reserved, more on this later

| Exponent | Stored as |
|---|---|
| -126 | 00000001 (1) |
| … | … |
| -1 | 01111110 (126) |
| 0 | 01111111 (127) |
| 1 | 10000000 (128) |
| … | … |
| 127 | 11111110 (254) |

# IEEE754 FLOATING POINT FORMATS

| Type | Sign | Exponent | Mantissa | Total |
|---|---|---|---|---|
| Single precision | 1 bit | 8 bits<br>Bias 127 | 23 bits | 32 bits |
| Double precision | 1 bit | 11 bits<br>Bias 1023 | 52 bits | 64 bits |
| Extended precision | 1 bit | 15 bits<br>Bias 16383 | 64 bits | 80 bits |

# IEEE754 FLOATING POINT FORMATS

- C# (and many other languages) have `float` and `double` types for single and double precision, respectively
- Literals are interpreted as `float` if they end in `f`, otherwise `double`
  - `3.14f` is a `float`
  - `3.14` is a `double`
- Python's `float` type is actually double precision
- Extended precision is not generally used in programs, but is used internally on the CPU for intermediate calculation results

# EXAMPLE (SINGLE PRECISION)

`0 10000001 10100000000000000000000`

**Sign**
0, so the number is positive

**Exponent**
10000001 = 129
Bias is 127
So actual exponent is
$129 - 127 = 2$

**Mantissa**
1.101000…
Don't forget the 1 in front of the point!

$$1.101 \times 2^2$$
$$= 110.1$$
$$= 6.5 \text{ (decimal)}$$

# SPECIAL FLOATING POINT NUMBERS

- Zero is stored by setting the mantissa and exponent to all 0s
- There is a notion of "signed 0" -- $+0$ and $-0$ both exist
- An exponent of all 1s represents infinity or NaN ("not a number"), depending on the mantissa

# INFINITY AND NAN

- `float.PositiveInfinity > x` is true for all finite x
- `float.NegativeInfinity < x` is true for all finite x
- `float.NaN > x`, `float.NaN < x`, `float.NaN == x` are **false** for all x
- Similarly for `double.PositiveInfinity` etc
- Can check for these with `float.IsNaN`, `float.IsInfinity` etc
- Infinity is useful wherever you want a "very large" value
- Infinity and NaN can also arise from calculations (especially division by 0)

# NUMERICAL PRECISION

# NUMERICAL PRECISION

- Back to scientific notation for a moment…
- It is common to write the mantissa with a set number of **decimal places**, rounding as needed
- $3.142$ (3 decimal places) can represent any number between $3.1415$ and $3.1425$ – a range of $0.001$
- So $3.142 \times 10^6$ could be any number between $3141500$ and $3142500$ – a range of $1000$
- The larger the exponent, the larger the possible **rounding error**

# FLOATING POINT PRECISION

- Floating point numbers have a limited number of bits for the mantissa
- Therefore they are also subject to rounding error
- And the margin of error gets larger as the exponent gets larger
- This can have real impacts, e.g. physics calculations getting less precise as we move further from the origin

# LIMITATIONS OF DECIMALS

- Not all numbers can be represented exactly in decimal notation in a finite number of digits
- E.g. $\frac{1}{3} = 0.333333 \ldots$ has an infinite decimal expansion
- The same is true in binary
- In fact, $\frac{1}{b}$ can **only** be represented in a finite number of bits **if** $b$ is a power of 2
- So some numbers which **can** be represented easily in decimal **can't** be represented exactly in binary, for example $\frac{1}{10}$

# TESTING FOR EQUALITY

- Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea
- E.g. in most languages, `0.1 + 0.2 == 0.3` is actually false!
- Better to check for **approximate equality**: calculate the difference between the two numbers, and check it is smaller than some small threshold values
- Unity has this built in: `Mathf.Approximately(0.1 + 0.2, 0.3)` will return true as you would expect

# OTHER REPRESENTATIONS

- **Fixed point**: take an $N$-bit integer and insert a point at a pre-set position
  - Was much faster on old hardware without dedicated floating point support (think 80s/90s)
  - Can be more precise in some situations
  - Range of numbers that can be stored is typically very narrow
- **Decimal type**: uses base 10 internally (i.e. stores numbers as arrays of decimal digits rather than as binary)
  - 0.1 + 0.2 does what you would expect
  - No hardware support, so much slower than floating point

# SUMMARY

- Modern computers have efficient ways of storing **signed integers** and **real numbers**
- **IEEE754** gives a good tradeoff between **storage space**, **range** and **precision**
- … however it is important to understand the limitations of its precision

# WORKSHOP