Exercise walk-through

13.3.1 The mantissa-exponent trade-off, normalisation, and overflow and underflow

A-Level Computer Science

You must be able to

- understand the effects of changing the allocation of bits to **mantissa 尾数** and **exponent 指数** in a floating-point representation
- normalise a floating-point number, and give the **reasons** for normalisation
- understand how **overflow** and **underflow** can occur
- understand that a binary representation is only an approximation, and that this gives rise to **rounding errors 舍入误差**

1 The method

Method — Mantissa bits buy precision; exponent bits buy range

The total number of bits is fixed, so the two can only grow at each other's expense. That single fact answers the whole question:

- **more mantissa bits** – more significant figures, so greater **precision** and smaller rounding errors; but fewer exponent bits, so a **smaller range** of magnitudes.
- **more exponent bits** – a **larger range**, both larger and smaller magnitudes representable; but fewer mantissa bits, so **less precision**, and the values are more coarsely spaced.

Method — Mantissa bits buy precision; exponent bits buy range

A three-mark answer names **both effects** and **both directions**. “More mantissa is more accurate” is one mark of three.

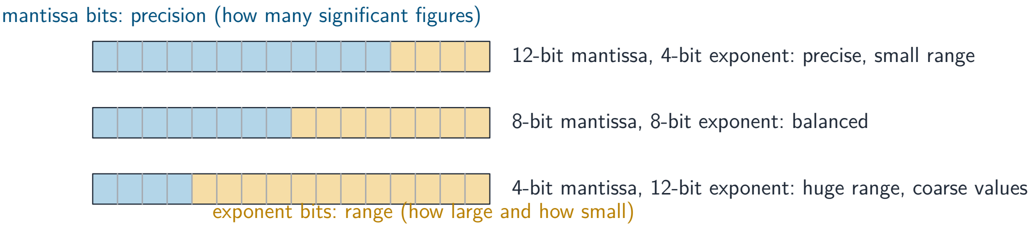
Worked example. *In a 10-bit mantissa, 6-bit exponent format, state the largest positive number.*

The largest mantissa is 0111111111 $= 1 - 2^{-9}$, just under 1, and the largest exponent is 011111 $= 31$. So the largest value is just under 2^{31}.

The smallest positive **normalised** number takes the smallest normalised mantissa, 0100000000 $= 0.5$, with the most negative exponent, 100000 $= -32$: $0.5 \times 2^{-32} = 2^{-33}$.

Method — Mantissa bits buy precision; exponent bits buy range

In a 10-bit mantissa, 6-bit exponent format, state the largest positive number.



the same 16 bits every time: more of one means less of the other

Method — Normalisation, and why it exists

A **normalised** mantissa starts 01 for a positive number and 10 for a negative one: the first bit after the sign must be the **opposite** of the sign bit.

To normalise: shift the mantissa **left** until that holds, and **decrease** the exponent by one for every place shifted. The value is unchanged, because each left shift doubles the mantissa and each decrement of the exponent halves the result.

Worked example. *Normalise mantissa 00011000 with exponent 00000101.*

The mantissa is $+0.1875$ and the exponent is 5, so the value is $0.1875 \times 32 = 6$.

Two left shifts give 01100000 $= 0.75$, and the exponent falls by 2 to 3. Check: $0.75 \times 8 = 6$. Same number.

The **reasons** for normalising, and a question asking for them wants the reason, not the rule:

- it gives the **greatest possible precision** for the bits available, because no mantissa bit is wasted holding a leading zero that carries no information;

Method — Normalisation, and why it exists

- it makes the representation **unique**, so one value has exactly one stored form, which lets two numbers be compared directly.

Method — Normalisation, and why it exists

A normalised mantissa starts 01 for a positive number and 10 for a negative one: the first bit after the sign…

before	`0.0011010`	exponent 4 (two wasted leading zeros)
	↓ shift left 2, exponent -2	
after	`0.1101000`	exponent 2 (normalised — same value)

Method — Overflow and underflow both come from the exponent

- **Overflow**: the result of a calculation is **larger than the largest** value the format can hold; the exponent would need to be bigger than its bits allow.
- **Underflow**: the result is a non-zero value **smaller than the smallest** the format can hold, too close to zero for the exponent to express, so it is stored as **zero**.
- Both are limits of the **exponent's** range, not the mantissa's. Blaming the mantissa, or defining overflow as "too many digits", is the answer the mark scheme is looking out for.
- Widening the exponent postpones both, at the cost of precision – which is the trade-off above, seen from the other side.

Method — Why a binary representation is only an approximation

A denary fraction is exact in binary only when its denominator is a power of 2. 0.5, 0.25 and 0.75 are exact; 0.1 is the recurring binary fraction $0.000110011\ldots$, which has to be cut off wherever the mantissa ends.

- **rounding errors** then accumulate over many operations, which is why `0.1 + 0.2` is not exactly `0.3`;
- so a real value is never tested with =. Test that the **difference** from the expected value is smaller than a small tolerance. A modulus function is **not** on the 9618 insert, so a question that needs one defines it; without one, test both directions, `IF X - 0.3 < 0.000001 AND 0.3 - X < 0.000001 THEN`;
- subtracting two nearly equal values leaves a result made almost entirely of error.
- Where exactness is required – currency, above all – use **fixed-point 定点** or **BCD** instead.

2 Practice

Practice 2.1 ▪ 2 marks

State what a **normalised** mantissa looks like for a positive number and for a negative number.

Solution 2.1

Worked steps

- positive: the sign bit is 0 and the next bit is 1, so it begins 01 **B1**
- negative: the sign bit is 1 and the next bit is 0, so it begins 10 **B1**
- in both cases the bit after the sign is the **opposite** of the sign bit

Practice 2.2 ▪ 2 marks

A number is stored with mantissa 01100000 and exponent 00000010.
Calculate its denary value.

Solution 2.2

Method: Normalisation, and why it exists — p.7

Worked steps

- mantissa 01100000 $= 0.5 + 0.25 = 0.75$; exponent 00000010 $= 2$ M1
- $0.75 \times 2^2 = \mathbf{3}$ A1

Practice 2.3 ▪ 1 mark

State **one** reason why floating-point numbers are normalised.

Solution 2.3

Method: Normalisation, and why it exists — p.7

Worked steps

- any one: it gives the greatest possible **precision** for the bits available, because no mantissa bit is wasted on a leading zero; or it makes the representation **unique**, so one value has one stored form **B1**

Practice 2.4 ▪ 2 marks

State what is meant by **overflow** and by **underflow**.

Solution 2.4

Method: Overflow and underflow both come from the exponent — p.10

Worked steps

- **overflow**: the result is larger than the largest value the format can represent **B1**
- **underflow**: the result is a non-zero value smaller than the smallest the format can represent, so it is stored as zero **B1**

Practice 2.5 ▪ 1 mark

State which part of a floating-point number, the mantissa or the exponent, determines the **range** of values that can be stored.

Solution 2.5

Method: Mantissa bits buy precision; exponent bits buy range — p.4

Worked steps

- the **exponent** B1

3 Exam-style

Exam-style 3.1 ▪ 3 marks

A system uses a fixed total of 16 bits for each floating-point number.
Describe the effect on the numbers that can be stored of **increasing** the number of bits allocated to the mantissa and reducing the number allocated to the exponent.

Solution 3.1

Worked steps

- **precision increases**: more mantissa bits means more significant figures, so values are stored more accurately and rounding errors are smaller **B1**
- **range decreases**: fewer exponent bits means the largest and smallest magnitudes that can be stored both move towards 1 **B1**
- the total is fixed, so one can only be gained at the other's expense; overflow and underflow therefore occur for values that the old allocation could have held **B1**

Exam-style 3.2 ▪ 6 marks

A number is stored with mantissa 00011000 and exponent 00000101.

Exam-style 3.2 (a) ▪ 2 marks

Calculate its denary value.

Solution 3.2 (a)

Method: Normalisation, and why it exists — p.7

Worked steps

mantissa 00011000 $= 0.125 + 0.0625 = 0.1875$; exponent $= 5$ **M1**

- $0.1875 \times 2^5 = 0.1875 \times 32 = \mathbf{6}$ **A1**

Exam-style 3.2 (b) ▪ 3 marks

Write the number in **normalised** form, giving the new mantissa and the new exponent.

Solution 3.2 (b)

Worked steps

the mantissa must begin 01, so shift **left** by **2** places M1

- new mantissa = **01100000** A1
- the exponent **decreases** by 2, from 5 to 3, so it is **00000011** A1
- shifting right, or increasing the exponent, changes the value and scores nothing

Exam-style 3.2 (c) ▪ 1 mark

Show that your normalised form has the same value.

Solution 3.2 (c)

Worked steps

$0.75 \times 2^3 = 0.75 \times 8 = 6$, the same value as in (a) **B1**

Exam-style 3.3 ▪ 3 marks

Explain why **overflow** and **underflow** are caused by the **exponent** rather than by the mantissa.

Solution 3.3

Method: Overflow and underflow both come from the exponent — p.10

Worked steps

- the mantissa fixes the **precision** – how many significant figures are kept – and it always represents a value between -1 and $+1$ B1
- the **exponent** fixes the **magnitude**, by saying what power of 2 the mantissa is multiplied by, so it alone decides how large or how small the number can be B1
- overflow happens when a result needs an exponent larger than the exponent field can hold, and underflow when it needs one more negative than the field can hold; running out of mantissa bits causes a loss of accuracy, not overflow B1

Exam-style 3.4 ▪ 3 marks

A program adds `0.1` to `0.2` and tests whether the result is equal to `0.3`. The test fails.

Exam-style 3.4 (a) ▪ 2 marks

Explain why.

Solution 3.4 (a)

Method: Why a binary representation is only an approximation — p.11

Worked steps

0.1 and 0.2 are **recurring** binary fractions, because their denominators are not powers of 2, so neither can be stored exactly **B1**

- each is rounded to fit the mantissa, and the two small errors survive into the sum, so the result differs from the stored value of `0.3` in its last bits **B1**

Exam-style 3.4 (b) ▪ 1 mark

State how the test should be written instead.

Solution 3.4 (b)

Worked steps

test that the **difference** from `0.3` is smaller than a small tolerance rather than testing for zero, for example `IF X - 0.3 < 0.000001 AND 0.3 - X < 0.000001 THEN`

B1

- a modulus function is not on the 9618 insert, so a question needing one supplies it

Exam-style 3.5 ▪ 3 marks

A banking system stores currency amounts. Explain why floating-point representation is **not** suitable, and state what should be used instead.

Solution 3.5

Method: Why a binary representation is only an approximation — p.11

Worked steps

- currency values such as 0.10 are recurring binary fractions and cannot be stored exactly in floating-point **B1**
- the rounding errors accumulate over many transactions, so totals drift and accounts fail to balance – an error of any size is unacceptable in money **B1**
- use **fixed-point** representation, or **BCD**, in which each denary digit is stored exactly **B1**