

Topic 13: Data Representation

The real IEEE 754 format, what floating-point arithmetic gets wrong, and a space-saving trick for hashing.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 13.2 File organisation and access (hashing), 13.3 Floating-point numbers. Try the challenge questions on the last pages before you read the answers.

IEEE 754: how real computers store reals

The syllabus uses a simple mantissa-and-exponent format. Nearly every real processor instead uses **IEEE 754**:

	Sign	Exponent (biased)	Fraction	Bias
Single (32-bit)	1 bit	8 bits	23 bits	127
Double (64-bit)	1 bit	11 bits	52 bits	1023

A normal value is $(-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exponent} - \text{bias}}$. The leading 1 is always present, so it is not stored (the **implicit bit**), giving a free extra bit of precision. The exponent is stored with a **bias** so it needs no sign bit.

Example: 6.5. $6.5 = 110.1$ in binary $= 1.101 \times 2^2$. Exponent field $= 2 + 127 = 129 = 10000001$. Fraction = 101 then zeros.

```
sign  exponent  fraction
  0    10000001  101000000000000000000000 = 0x40D00000
```

Special values and rounding error

- **Exponent all 1s:** fraction 0 means $\pm\infty$ (e.g. 1/0.0); fraction non-zero means **NaN** ("not a number", e.g. 0/0.0). NaN is not equal to anything, not even itself.
- **Exponent all 0s:** zero (note **+0** and **-0** both exist) or tiny *subnormal* numbers that fill the gap near zero.
- **Precision is finite:** a double holds about 15–16 significant decimal digits; the gap between 1.0 and the next double (machine epsilon) is $2^{-52} \approx 2.2 \times 10^{-16}$.

Spacing between representable numbers *grows* with magnitude. Above $2^{53} \approx 9 \times 10^{15}$ a double cannot represent every integer: consecutive doubles are 2 apart, so $1e16 + 1$ rounds back to $1e16$. This produces **catastrophic cancellation**: $(1e16 + 1) - 1e16$ evaluates to **0** instead of 1. When subtracting nearly equal numbers, the leading digits cancel and only the rounding noise remains. Numerical programmers reorder calculations to avoid it.

Bloom filters: "definitely not" or "probably yes"

Hashing (13.2) maps a key to a storage location. A **Bloom filter** uses hashing to answer "have I seen this item before?" in a tiny amount of memory. It is a bit array of m bits, all 0, plus k hash functions.

- **Add** an item: compute its k hashes and set those k bits to 1.
- **Query** an item: if *any* of its k bits is 0, it is **definitely not** in the set. If all are 1, it is **probably** in the set.
- **No false negatives**, but **false positives** can happen when other items happened to set the same bits.

Browsers have used Bloom filters to quickly rule out safe URLs before checking a blocklist, and databases use them to avoid unnecessary disk reads. Each item costs only a few bits, far less than storing the item itself.

Challenge questions

1. Encode -0.75 as a 32-bit IEEE 754 number. Give the sign, exponent and fraction fields.
2. Decode the 32-bit pattern 0x41200000.
3. A Bloom filter has $m = 8$ bits and $k = 2$ hash functions. "cat" hashes to bits 1 and 5; "dog" to bits 3 and 5. After adding both, query "emu" (bits 1 and 3) and "fox" (bits 2 and 6). What does each query return?

Worked answers

Only look once you have had a proper go at each question.

1. $0.75 = 0.11$ in binary $= 1.1 \times 2^{-1}$.

Sign = 1; exponent = $-1 + 127 = 126 = 01111110$; fraction = 1 followed by 22 zeros. Result $1\ 01111110\ 1000000000000000000000 = 0xBF400000$.

2. $0100\ 0001\ 0010\ 0000\dots$ = sign 0, exponent $10000010 = 130$, fraction $01000\dots$. Value = 1.01 (binary) $\times 2^{130-127} = 1.25 \times 8 = \mathbf{10.0}$.

3. After adding both, bits 1, 3 and 5 are set.

"emu": bits 1 and 3 are both set, so it answers **probably in the set**, a *false positive* since emu was never added.

"fox": bit 2 is 0, so it is **definitely not** in the set.