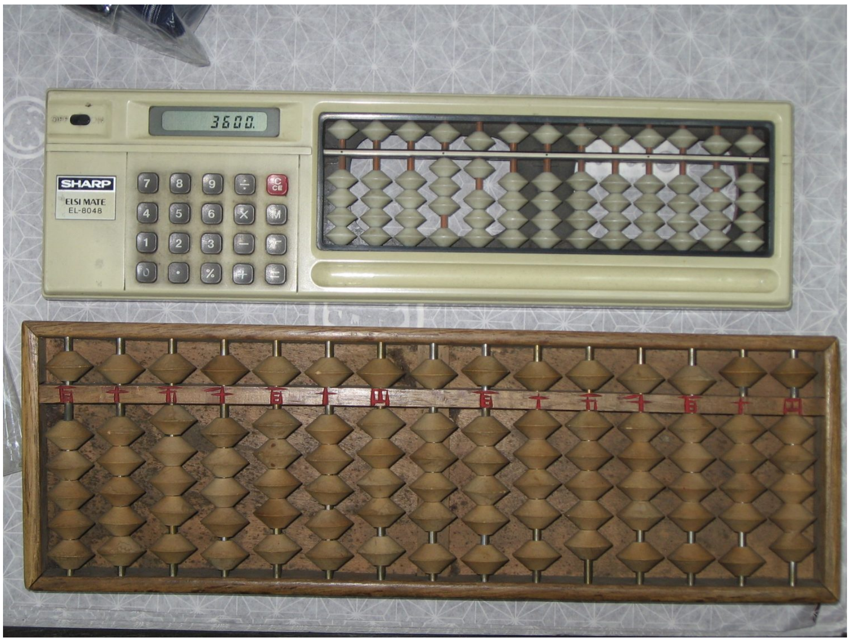
Information representation

A-Level Computer Science

Number systems

The three **number systems** 数制 you must use:

- **denary** 十进制 (decimal, base 10) —uses digits 0–9. Place values are powers of ten.
- **binary** 二进制 (base 2) —uses 0 and 1. Place values are powers of two. Every **byte** 字节 is 8 **bits** 位.
- **hexadecimal** 十六进制 (base 16) —uses 0–9 then A–F for 10–15. Each hex **digit** 数位 stands for exactly 4 bits.



An abacus represents numbers by place value —the same idea behind decimal, binary and hexadecimal

Image: Syced, CC0 (commons.wikimedia.org)

Conversions

Denary → binary: keep dividing by 2 and record the remainders, read bottom-up. Or subtract the largest **place value** 位值 (power of 2) that fits.

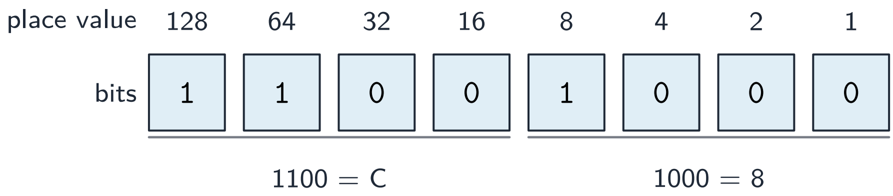
Example: 558_{10}: $558 = 512 + 32 + 8 + 4 + 2 = 2^9 + 2^5 + 2^3 + 2^2 + 2^1$. In 12 bits: `0010 0010 1110`.

Binary → hex: group the bits into **nibbles** 半字节 (4 bits) from the right and convert each. `0010 0010 1110` → `2 2 E` → `22E`.

Hex → binary: replace each hex digit with its 4-bit pattern. **Hex → denary**: multiply each digit by its place value. `22E` $= 2 \times 256 + 2 \times 16 + 14 = 558$.

Worked example. Convert denary 200 to 8-bit binary, then to hexadecimal.

$200 = 128 + 64 + 8$, so the binary is `11001000`. In nibbles, `1100 1000` = 12 and 8, i.e. C and 8, so the hexadecimal is `C8`.



$$200 = 128 + 64 + 8 = \texttt{11001000}_2 = \texttt{C8}_{16}$$

Reading 200 from its place values, then grouping the bits into nibbles to get hex C8

Binary vs decimal prefixes

Two prefix families look similar but differ —decimal (powers of 10) and binary (powers of 2):

Decimal (SI)	Binary (memory)
kilo $= 10^3$	kibi (Ki) $= 2^{10} = 1024$
mega $= 10^6$	mebi (Mi) $= 2^{20}$
giga $= 10^9$	gibi (Gi) $= 2^{30}$
tera $= 10^{12}$	tebi (Ti) $= 2^{40}$

So a tebibyte (TiB) is slightly more than a terabyte (TB). A ”1 TB” drive holds 10^{12} bytes, but an operating system that reports in TiB shows a smaller number.

Binary arithmetic

Binary addition

Add column by column from the right, carrying as in denary:

Bit A	Bit B	Carry in	Sum bit	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	1	0	0	1
1	1	1	1	1

Overflow 溢出 happens when the result needs more bits than the **register** 寄存器 can hold —the carry-out of the leftmost column is the overflow bit.

Binary subtraction

The usual way is **two's complement 补码** addition: to do $A - B$, form the two's complement of B (invert every bit and add 1), then add, and discard any final carry-out.

To subtract 00011110 from 01100100 (unsigned 8-bit):

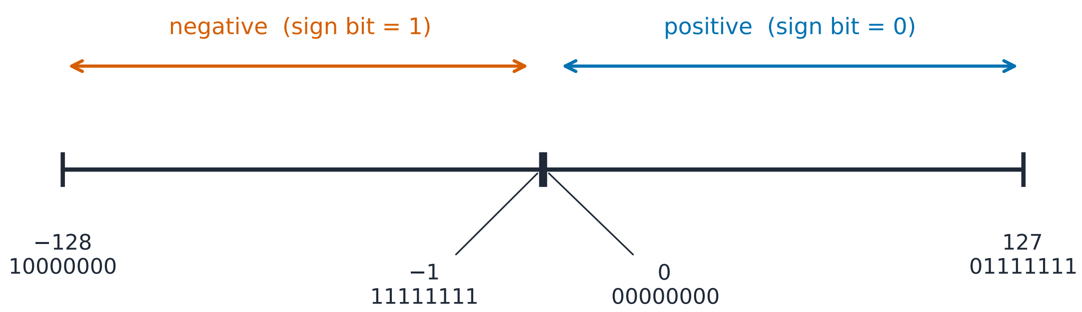
- two's complement of 00011110: invert $\rightarrow$ 11100001, add 1 $\rightarrow$ 11100010.
- add to 01100100: result 1 01000110 (9 bits) —discard the leading 1 $\rightarrow$ 01000110 = 70_{10}. Check: $100 - 30 = 70$.

Two's complement signed integers

In an n-bit two's-complement number:

- the **most significant bit 最高有效位** (MSB) is the **sign bit 符号位**: 0 = positive, 1 = negative.
- to read a negative number: invert every bit, add 1, then negate.

So 11100010 is negative; invert $\rightarrow$ 00011101, add 1 $\rightarrow$ 00011110 = 30, so it is -30. This is a **signed integer 有符号整数** (unlike an **unsigned 无符号** one). The range for n bits is -2^{n-1} to $+2^{n-1} - 1$; for 8 bits, -128 (10000000) to $+127$ (01111111).



8-bit two's complement: the sign bit splits the range into negative (-128 to -1) and positive (0 to 127)

Worked example. What denary value does the 8-bit two's-complement number 10110100 represent?

The MSB is 1, so it is negative. Invert $\rightarrow$ 01001011, add 1 $\rightarrow$ 01001100 = 76, so the value is -76. Check with place values: $-128 + 32 + 16 + 4 = -76$.

Overflow in signed arithmetic happens when the true result falls outside this range — spotted when the sign bit flips wrongly (two positives giving a negative, or two negatives giving a positive).

One's complement

Before two's complement, an older scheme called **one's complement** 反码 represented a negative number by simply **inverting every bit** of the positive —there is no "add 1" step.

- $+30 = 00011110$, so in one's complement $-30 = 11100001$ (just the inverse).
- Drawback: it has **two zeros** —00000000 (+0) and 11111111 (−0) —which wastes a bit pattern and makes arithmetic awkward.

Two's complement (invert **and** add 1) removes the negative zero: it has a single zero and lets addition and subtraction use the same circuit. That is why modern computers store signed integers in two's complement, not one's complement.

Binary Coded Decimal (BCD)

In **BCD** 二进码十进数, each denary digit is written as its own 4-bit pattern. The number 93 is `1001 0011` in BCD —not binary 93 (01011101). Each nibble uses only 0–9; patterns 1010–1111 are invalid.

BCD reading: `0010 0111 0101` → 2, 7, 5 → 275.

Use: calculators, digital clocks, and devices that show denary digits —each digit drives a **7-segment display** 七段显示器. Currency code often uses BCD to avoid the rounding errors of converting fractions like 0.1 to binary.

A seven-segment display shows one denary digit, often driven by BCD

Image: The Pi Hut, Product image (thepihut.com)

Hexadecimal —practical uses

Hex is a compact way to write binary (1 hex digit = 4 bits):

1	0	1	0		1	1	1	1	→ AF
nibble = A					nibble = F				

A byte is two nibbles; each nibble is one hex digit

- **memory addresses** 内存地址 in low-level programming —`0x7FFE`.
- **colour values** in HTML/CSS —`#FF8800`.
- **MAC addresses** —`AC:DE:48:00:11:22`.

Hex does not change the stored data —it just makes binary easier for humans.

Character codes

Computers store text as numbers; each character has a numeric **code point** 码点 set by a **character set** 字符集.

ASCII

- **ASCII** uses 7 bits —128 code points. Basic Latin letters, digits, punctuation, and control codes.
- **Extended ASCII** uses 8 bits —256 code points; the lower 128 match ASCII, the upper 128 vary by region.

character	code (denary)	binary
`A`	65	01000001
`a`	97	01100001
`0`	48	00110000
`(space)`	32	00100000

Each character is stored as a number —a few ASCII code points in denary and binary

Unicode

- **Unicode** is a universal character set covering almost every script, plus symbols and emoji.
- common **encodings** 编码: **UTF-8** (1–4 bytes, ASCII-compatible), **UTF-16** (2 or 4 bytes), **UTF-32** (fixed 4 bytes).

Why Unicode beats ASCII

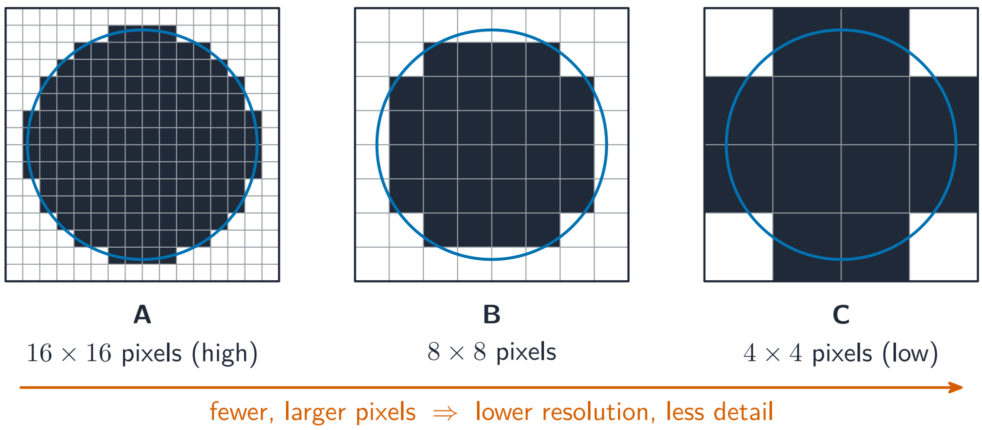
- it represents **far more characters** (every script, emoji); ASCII covers only basic English.
- files are **portable** with no code-page confusion, and allow **multilingual** text in one document.

- trade-off: Unicode files are usually **larger** for English-only text.

Bitmap images

A **bitmap** 位图 image (also called a **bitmapped image**) stores the colour of every **pixel** 像素 in a grid. At the start of the file a **file header** 文件头 records the image's metadata —its width, height and colour depth —so software knows how to read the pixel data that follows.

- **image resolution** 图像分辨率: the bitmap's own size, width × height in pixels (e.g. 1920 × 1080).
- **screen resolution** 屏幕分辨率: the width × height the **display** can show. If an image's resolution is larger than the screen it is scaled down to fit; a low-resolution image looks blocky when stretched onto a higher-resolution screen.
- **colour depth** 颜色深度 (**bit depth** 位深度): bits per pixel. 1 bit → black/white; 8 bits → 256 colours; 24 bits → 16.7 million ("true colour").



The same image stored at three resolutions, from high (A) to low (C): fewer, larger pixels mean less detail

File size

$$\text{size in bits} = \text{width} \times \text{height} \times \text{bit depth}.$$

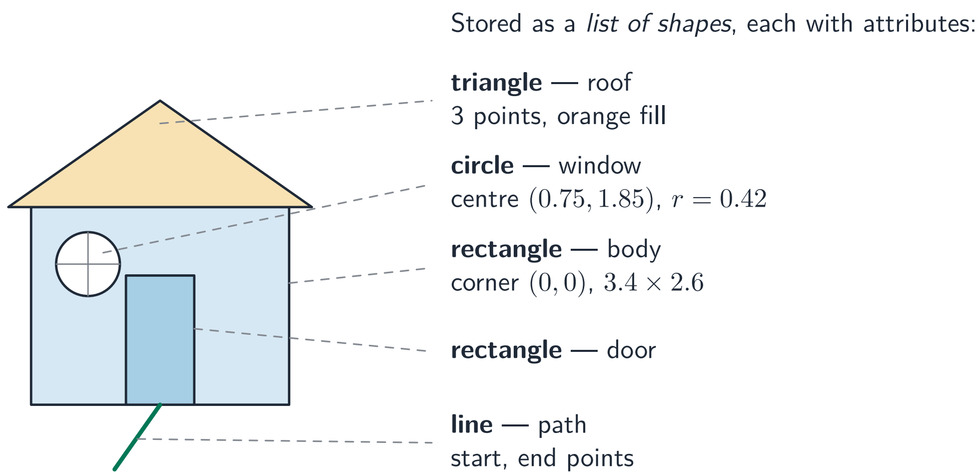
Divide by 8 for bytes, by 1024 for KiB, etc. Example: a 3000 × 2000 image at 24 bpp is $3000 \times 2000 \times 24 = 1.44 \times 10^8$ bits ≈ 17.2 MiB.

Changing settings

- **lower resolution** → smaller file, less detail (looks blocky when enlarged).
- **lower colour depth** → smaller file, but smooth shades show banding.
- **higher** of either → larger file, better quality.

Vector graphics

A **vector graphic** 矢量图形 stores the **instructions** to draw the image as a **drawing list** 绘图列表—an ordered list of **drawing objects** 绘图对象 (geometric **primitives** 图元: lines, curves, polygons, circles). Each drawing object has **properties** 属性 such as colour, fill, line width and position (coordinates). To show it, the program **renders** 渲染 the drawing list at any resolution needed.

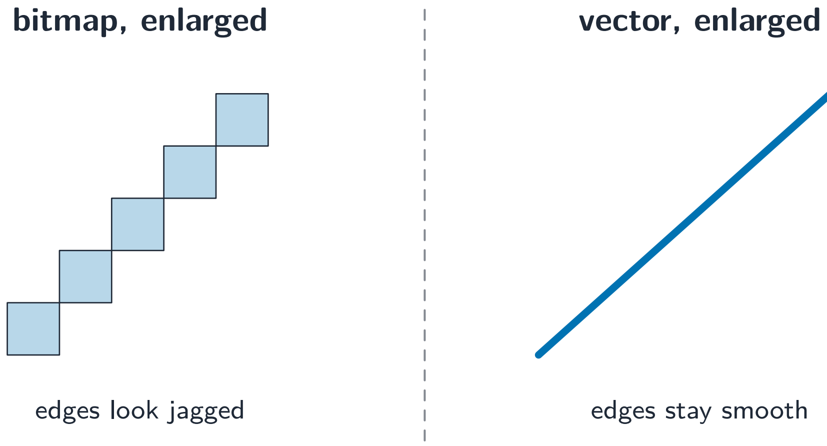


A vector image is built from labelled geometric shapes, each with attributes

Bitmap vs vector

Task	Better choice	Why
Photograph	Bitmap	Complex pixel-level detail can't be described as shapes.
Logo, icon, sign	Vector	Sharp edges; scales to any size without blur.
Engineering drawing	Vector	Precise geometry and scaling.
Painting, texture	Bitmap	Smooth tonal detail per area.

Vector advantage: it **scales without losing quality** —a vector logo stays sharp at any size, while a bitmap blurs when enlarged. **Vector disadvantage**: it cannot describe arbitrary pixel detail (photographs).

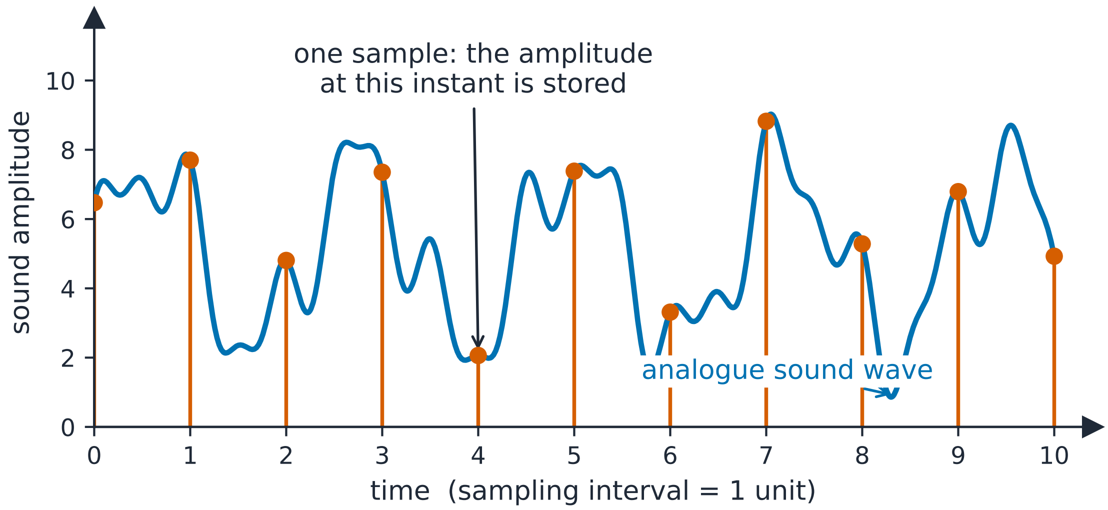


Enlarged, a bitmap's pixels turn jagged; a vector stays smooth at any size

Sound

A continuous wave of **analogue data** 模拟数据 (the sound) is converted into **digital data** 数字数据 by **sampling** 采样:

- **sampling rate** 采样率—samples per second (Hz). CD quality is 44.1 kHz.
- **sampling resolution** 采样分辨率 (bit depth) —bits per sample's **amplitude** 振幅. CD quality is 16 bits.



Sampling a sound wave: its amplitude is read at each time interval

File size

$$\text{size in bits} = \text{sampling rate} \times \text{resolution} \times \text{duration} \times \text{channels}.$$

A 10-second stereo CD clip: $44100 \times 16 \times 10 \times 2 \approx 1.8$ MiB.

Changing settings

- **higher sampling rate** → captures higher pitches, larger file.
- **higher sample resolution** → finer amplitude steps, less **quantisation 量化** noise, larger file.
- **lower** of either → smaller file, clear quality loss.

(The sampling rate must be at least twice the highest frequency you want to keep.)

Compression

Compression 压缩 reduces file size, saving storage and transmission **bandwidth 带宽**. Two kinds:

- **lossless 无损**—the original data is recovered exactly (text, programs, ZIP/PNG).
- **lossy 有损**—some detail is dropped for much smaller files (JPEG, MP3, video).

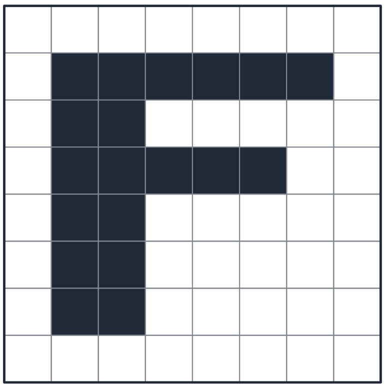
When to use which

- **lossless** for documents, source code, medical images —anything needing exact data.
- **lossy** for streaming media. **Real-time video streaming** uses lossy compression because it must send huge amounts of data in real time over limited bandwidth; lossless would not shrink it enough. Raw HD video is gigabytes per minute, so without compression the picture would keep freezing.

Lossless methods

- **run-length encoding 行程编码** (RLE): store ”the next n values are x” instead of repeating x. Great for flat areas; useless for noisy data.
- **dictionary methods 字典编码** (ZIP, PNG): replace repeated byte sequences with a short reference. Good for text and code.
- **Huffman coding 霍夫曼编码**: give short codes to common symbols and long codes to rare ones, bringing the average code length near the data's **entropy 熵**.

8×8 bitmap



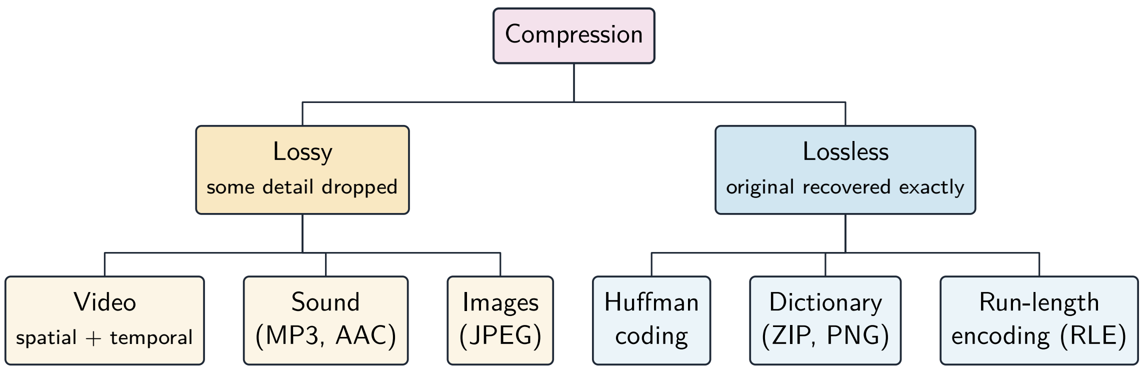
binary	RLE code
00000000	8W
01111110	1W 6B 1W
01100000	1W 2B 5W
01111100	1W 5B 2W
01100000	1W 2B 5W
01100000	1W 2B 5W
01100000	1W 2B 5W
00000000	8W

Key: 1 = black (B), 0 = white (W). RLE stores *count* + *colour*: e.g. 01111110 → 1W 6B 1W

Run-length encoding of the letter F in an 8×8 black-and-white grid

Lossy methods

- **images** (JPEG): drop fine detail and colour differences the eye barely sees.
- **sound** (MP3, AAC): drop pitches we hear less well, and quiet sounds hidden by louder ones.
- **video** combines **spatial** 空间 compression (within each frame, like JPEG) with **temporal** 时间 compression (most frames store only the differences from the previous frame).



Compression methods: lossless versus lossy, with common examples

Exam tips

- Show working for base conversions: denary → binary by place values, binary → hexadecimal in **nibbles** (groups of 4 bits).
- For **two's complement** the MSB is negative; to negate, **invert and add 1**; watch for **overflow** when the sign bit flips wrongly.
- Distinguish **bitmap** (pixels; file size = width × height × colour depth) from **vector** (drawing commands; scales without loss).

- Sound file size depends on **sample rate** $\times$ **bit depth** $\times$ **time** —more of each means better quality but a bigger file.
- Compare **lossless vs lossy** compression and give a use for each.