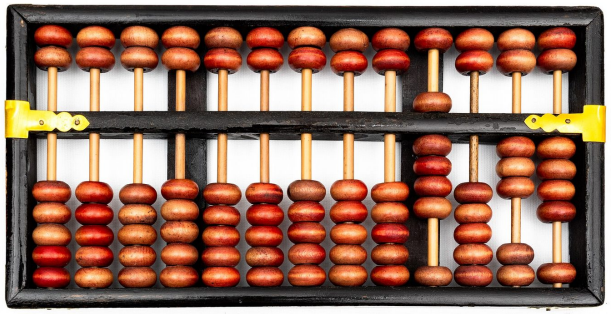
Information representation

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

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.

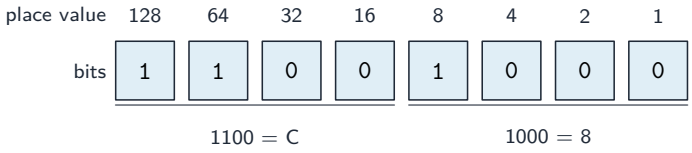
Example: 558_{10}: $558 = 512 + 32 + 8 + 4 + 2 = 2^9 + 2^5 + 2^3 + 2^2 + 2^1$. In 12 bits: `0010 0010 1110`.

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

How many bits?

Exam questions fix the **register width** 寄存器宽度 (8, 12 or 16 bits). Pad with leading zeros to that width: 558 in 12 bits is `0010 0010 1110`, never `10 0010 1110`.

To find the **minimum number of bits** that can store a value, ask which place values you need:

- an unsigned integer from 0 to $2^n - 1$ needs n bits: 200 needs 8 bits (the top is 255), 1000 needs 10 bits (the top is 1023), 16 needs 5 bits (4 bits stop at 15).
- a signed two's-complement integer from -2^{n-1} to $2^{n-1} - 1$ needs n bits: -200 needs 9 bits, because 8 bits stop at -128.
- one hexadecimal digit needs 4 bits, one BCD digit needs 4 bits, and one ASCII character needs 7 bits (8 for extended ASCII).

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.

Worked example. Add the 8-bit unsigned integers 10110101 and 01101100, and comment on the result.

$10110101 + 01101100 = 1\,00100001$. The answer needs 9 bits, so it does not fit in an 8-bit register: overflow has occurred. A full answer names the error and says why, using the word size the question gave: "Overflow: the true result (289) is larger than the largest value an 8-bit register can hold (255), so the carry out of the most significant bit is lost and the stored result (00100001 = 33) is wrong."

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 → 11100001, add 1 → 11100010.
- add to 01100100: result 1 01000110 (9 bits) — discard the leading 1 → 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 → 00011101, add 1 → 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).

The same bits mean different numbers depending on the agreed reading. As an **unsigned** integer every bit is a place value, so 8 bits run from 0 to 255; as a **signed** two's-complement integer the top bit is the sign, so the same 8 bits run from −128 to +127. The pattern 11111111 is 255 read one way and −1 read the other — nothing in the bits themselves says which.

8 bits	as unsigned every bit a place value	as signed top bit is the sign
00000000	0	0
01111111	127	+127
10000000	128	−128
11111111	255	−1
range for n bits	$0 \ldots 2^n - 1$	$-2^{n-1} \ldots 2^{n-1} - 1$

The same byte read as unsigned and as signed: only the agreed interpretation tells them apart *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 → 01001011, add 1 → 01001100 = 76, so the value is −76. Check with place values: $-128 + 32 + 16 + 4 = -76$.

Worked example. Write −108 as a 12-bit two's-complement integer.

Start from +108 in 12 bits: $108 = 64 + 32 + 8 + 4$, so `0000 0110 1100`. Invert every bit: `1111 1001 0011`. Add 1: `1111 1001 0100`. Check with place values, where the top bit is worth $-2^{11} = -2048$: $-2048 + 1024 + 512 + 256 + 128 + 16 + 4 = -108$. ✓

For 12 bits the range is −2048 (`1000 0000 0000`) to +2047 (`0111 1111 1111`). Questions that ask for the smallest and largest values want these two patterns, so learn the rule: the most negative number is a 1 followed by zeros; the most positive is a 0 followed by ones.

An **arithmetic shift** 算术移位 moves every bit left or right but keeps the sign: a shift right by one place halves the value and copies the sign bit into the empty space on the left, so a negative number stays negative (`1111 1001 0100` shifted right three places is `1111 1111 0010`, which is −14: $-108/8 = -13.5$, and a shift right rounds down). A shift left doubles the value. Shifts belong to the assembly instruction set in topic 4, but this question is asked with the number work here.

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 "justify" answer must link the use to a property of BCD: each denary digit has its own 4 bits, so a digit can be sent straight to its display, or added digit by digit, with no conversion of the whole number; and a decimal fraction such as 0.10 is stored exactly, which a binary fraction cannot do.

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

Hexadecimal — practical uses

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

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

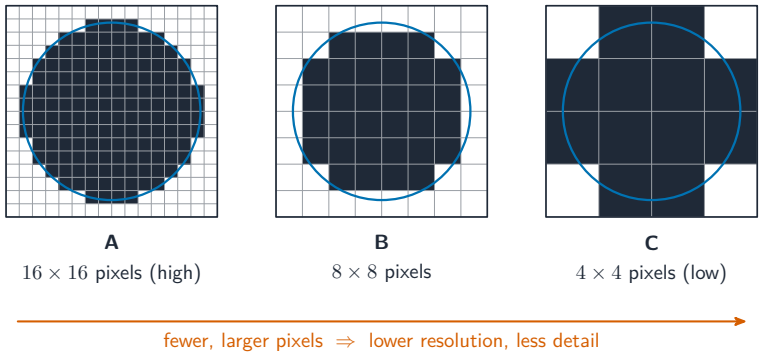
- it represents **far more characters** (every script, emoji); ASCII covers only basic English.
- files are **portable** with no code-page confusion, and allow **multilingual** text in one document.
- trade-off: Unicode files are usually **larger** for English-only text.

When a question asks for **differences**, give them in pairs with numbers: ASCII uses 7 bits (extended ASCII 8), so 128 (256) characters; Unicode uses up to 32 bits (UTF-8 uses 1 to 4 bytes), so more than a million code points. ASCII covers basic English only; Unicode covers every script, and its first 128 code points are the ASCII ones. In UTF-8 an English letter still takes 1 byte, so a 40-letter English file name is 40 bytes in ASCII and in UTF-8 alike, while a Chinese character takes 3 bytes.

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").

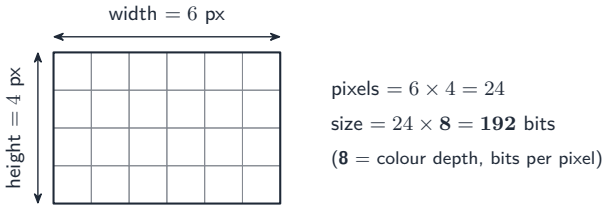


The same image stored at three resolutions, from high (A) to low (C): fewer, larger pixels mean less detail

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.



The same formula on small numbers: count the pixels, then multiply by the colour depth

State the units you used. The mark scheme accepts 1 MB = 10^6 bytes (the SI prefix) or 1 MiB = 1024×1024 bytes (the binary prefix), as long as your working shows which one; the same image is 18.0 MB or 17.2 MiB. Add the size of the file header if the question gives one.

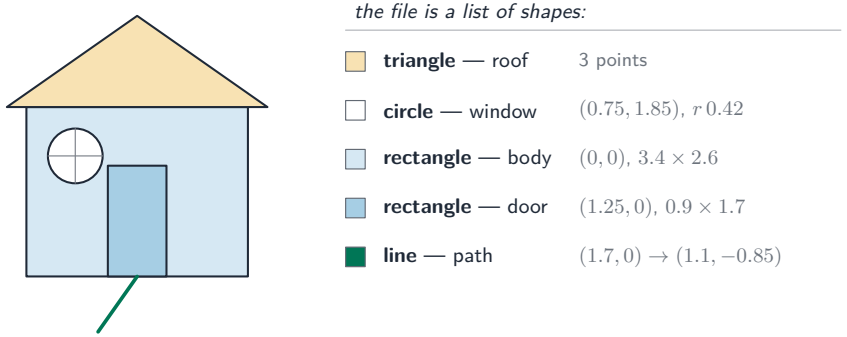
A video is a sequence of bitmap images, each one a **frame** 帧. Before compression its size is the size of one frame × the **frame rate** 帧率 (frames per second) × the duration in seconds: 30 frames per second of 1920 × 1080 pixels at 24 bits is $30 \times 1920 \times 1080 \times 24 \approx 1.5 \times 10^9$ bits, about 187 MB, for every second. That is why video is always compressed.

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.



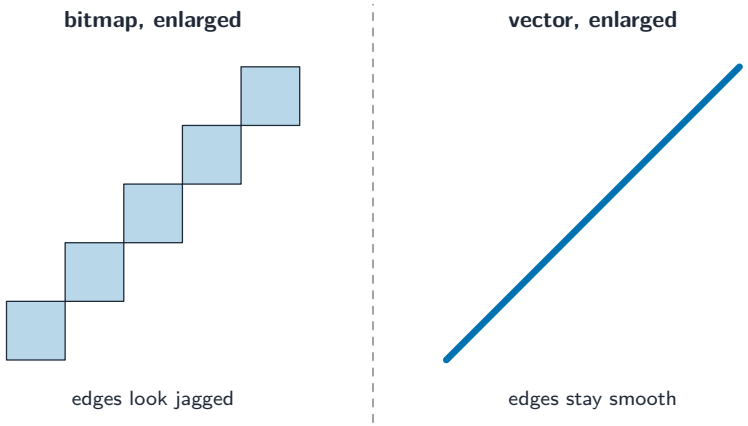
A vector image is built from labelled geometric shapes, each with attributes

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).

A "justify" answer links the choice to the task. "The logo must appear on a business card and on a billboard, so it should be a vector graphic: it is stored as drawing objects and is re-rendered sharply at any size, whereas a bitmap would show its pixels when enlarged." For a photograph the argument runs the other way: there are no shapes to describe, so every pixel's colour must be stored.

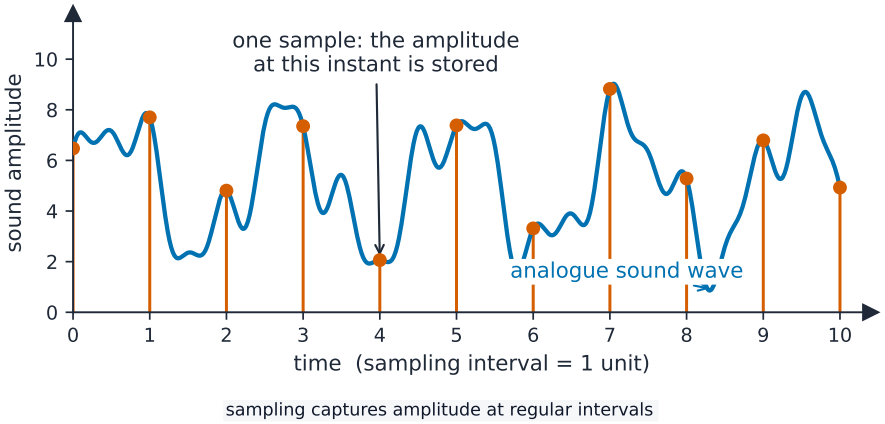


Enlarged, a bitmap's pixels turn jagged; a vector stays smooth at any size

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

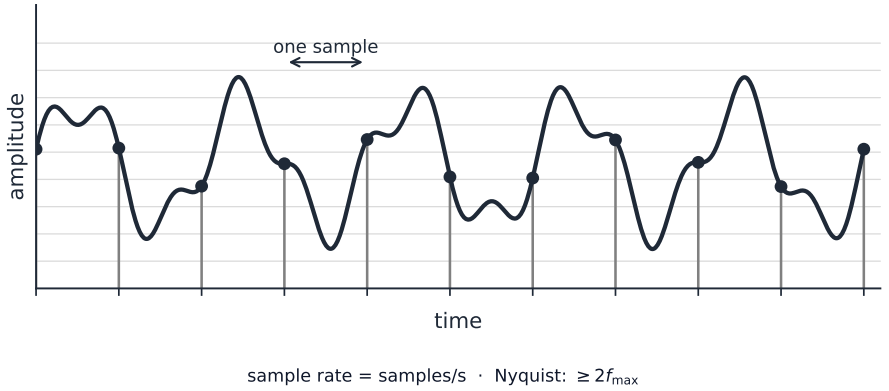
$$\text{size in bits} = \text{sampling rate} \times \text{resolution} \times \text{duration} \times \text{channels}.$$

A 10-second stereo CD clip: $44100 \times 16 \times 10 \times 2 = 14\,112\,000$ bits ≈ 1.68 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.)



Sample rate is samples per second; the Nyquist rule is why it must be at least twice the highest frequency kept

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.

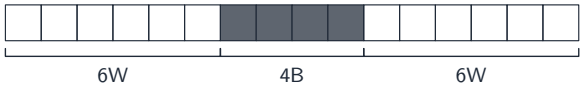
A "justify" answer names the method, then the reason from the situation: "Lossless, because the spreadsheet must be restored exactly; a single changed value would make the accounts wrong." Or: "Lossy, because the photographs are viewed on a phone screen where the dropped detail is not visible, and the smaller files upload faster and use less storage."

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.

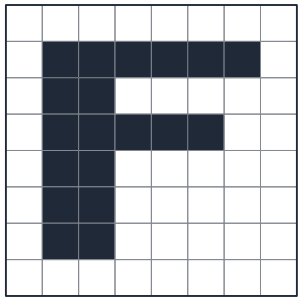
How each kind of file is compressed:

- **text file**: dictionary methods and Huffman coding turn repeated words and common characters into short codes. Text must stay lossless, because one changed character changes the meaning.
- **bitmap image**: RLE for runs of identical pixels (icons, diagrams, black-and-white scans); lossy JPEG for photographs, or a lower colour depth or resolution.
- **vector graphic**: the drawing list is already small; remove drawing objects that are not needed, store coordinates to fewer decimal places, or apply a lossless method such as ZIP to the file.
- **sound file**: lossy MP3 or AAC removes what the ear cannot hear; a lower sampling rate or resolution is also lossy; lossless formats keep every sample and shrink the file much less.



16 pixels
⇒ **6W 4B 6W**
(3 runs, not 16)

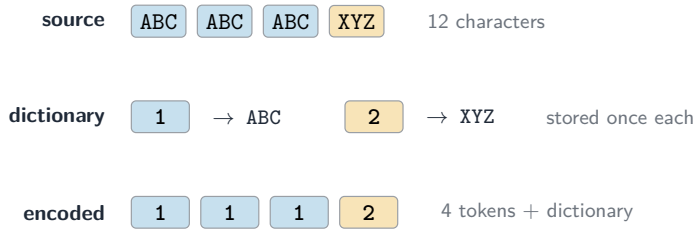
Run-length encoding on a single row: 16 pixels become 3 runs



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

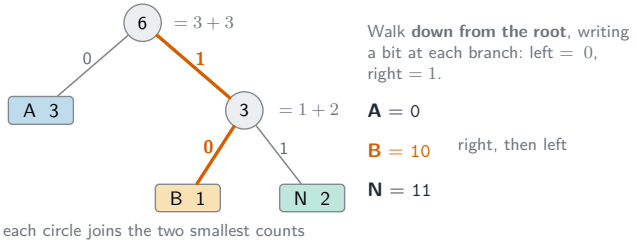
Key: 1 = black (B), 0 = white (W). RLE stores *count* + *colour*: e.g. 01111110 → 1W 6B 1W

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



The decoder rebuilds the dictionary as it reads, so the original returns exactly — **lossless** (ZIP, PNG).

Dictionary coding: each repeated sequence is stored once, and every occurrence becomes a short index

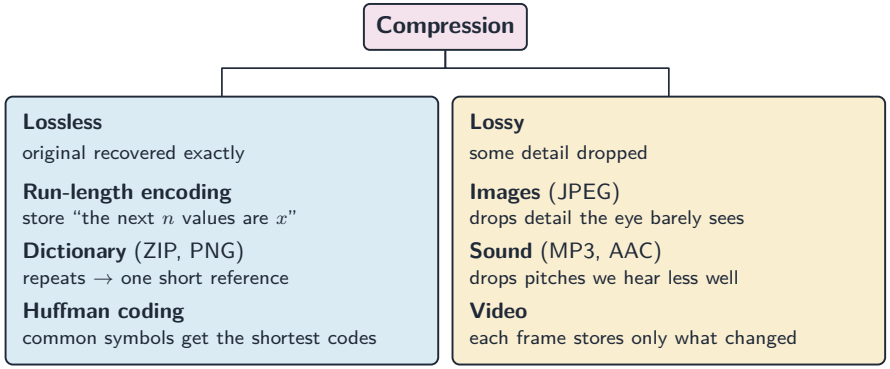


BANANA becomes 10 0 11 0 11 0 — **10 bits**, against 12 for fixed 2-bit codes: the commonest letter is nearest the root.

Huffman coding: the commonest symbol gets the shortest code, so BANANA needs 10 bits instead of 12

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

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
bit	a single binary digit, 0 or 1
byte	a group of 8 bits
binary prefix	a multiplier that is a power of 2 (kibi = 1024) rather than a power of 10 (kilo = 1000)
two's complement	a way of representing signed integers in which the most significant bit has a negative place value
overflow	the result of a calculation is too large to be represented in the number of bits available
Binary Coded Decimal	each denary digit is stored as its own 4-bit binary pattern
character set	the set of characters a computer can represent, each with its own binary code
pixel	the smallest element of a bitmap image, storing one colour value
image resolution	the number of pixels in an image, given as width by height
screen resolution	the number of pixels a display can show, given as width by height
colour depth	the number of bits used to store the colour of one pixel
sampling rate	the number of samples of the sound taken per second
sampling resolution	the number of bits used to store the amplitude of one sample
lossless compression	compression from which the original data can be recovered exactly
lossy compression	compression that permanently removes some data, so the original cannot be recovered
run-length encoding	replacing a run of repeated values with one value and a count

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** × **bit depth** × **time** — more of each means better quality but a bigger file.
- Compare **lossless vs lossy** compression and give a use for each.

Common mistakes

- Explaining an overflow with "the answer was greater than 255" or "it has 9 bits". State the word size the question gave, then say the result cannot be represented in it.
- Making a negative number by setting the top bit to 1 and leaving the rest (sign and magnitude). Two's complement means invert every bit of the positive value, then add 1.
- Forgetting to pad a converted number to the register width the question asks for.
- Mixing bits and bytes in a file-size calculation. Work in bits, divide by 8 once, and say whether you used 1000 or 1024.
- Answering "describe" in everyday words ("the picture gets worse"). Use the syllabus terms: fewer colours, banding, lower image resolution, larger pixels.