

Computer Science for A Level

OCR H446 — Student Book

First edition · 2026

Adriano G Platania

Aligned against OCR A Level Computer Science H446, specification version 2.7 (March 2024)

Terms of use

This book may be read, downloaded, and shared free of charge for personal study and for classroom use. It may not be resold or redistributed for commercial purposes. This book is intended as a supplement to the textbooks officially adopted by your school. Please use it alongside your prescribed course materials and follow your teacher's guidance.

Table of Contents

ASSESSMENT AT A GLANCE	6
HOW TO USE THE BOOK.....	6
<u>SECTION 1 — PROCESSORS, INPUT, OUTPUT AND STORAGE (SPEC 1.1).....</u>	<u>7</u>
1.1 STRUCTURE AND FUNCTION OF THE PROCESSOR	7
1.2 THE FETCH–DECODE–EXECUTE CYCLE.....	8
1.3 PERFORMANCE AND PIPELINING.....	8
1.4 PROCESSOR ARCHITECTURES.....	9
1.5 CISC, RISC, GPUS AND PARALLEL SYSTEMS.....	9
1.6 INPUT, OUTPUT AND STORAGE	9
INPUT DEVICES	9
OUTPUT DEVICES	10
STORAGE INTRODUCTION.....	10
WORKED EXAMPLE 1 — SELECTING AND JUSTIFYING DEVICES.....	12
SECTION 1 EXERCISES.....	13
QUICK CHECK.....	13
PRACTICE	13
EXAM-STYLE	13
CHECK YOURSELF	14
KEY FACTS	15
<u>SECTION 2 — SOFTWARE, SOFTWARE DEVELOPMENT AND PROGRAMMING LANGUAGES (SPEC 1.2).....</u>	<u>16</u>
2.1 SYSTEMS SOFTWARE AND THE OPERATING SYSTEM	16
MEMORY MANAGEMENT: PAGING, SEGMENTATION AND VIRTUAL MEMORY	16
INTERRUPTS AND ISRS.....	16
SCHEDULING	16
TYPES OF OPERATING SYSTEM.....	17
BIOS, DEVICE DRIVERS AND VIRTUAL MACHINES	17
2.2 APPLICATIONS, UTILITIES AND SOFTWARE LICENSING.....	17
2.3 TRANSLATORS AND THE PATH FROM SOURCE TO EXECUTION	17
2.4 SOFTWARE DEVELOPMENT METHODOLOGIES.....	18
2.5 PROGRAMMING PARADIGMS AND LANGUAGE LEVELS.....	19
2.6 LITTLE MAN COMPUTER (LMC)	19
WORKED EXAMPLE 1 — INPUT TWO NUMBERS AND OUTPUT THEIR SUM.....	20
WORKED EXAMPLE 2 — CONTINUOUSLY OUTPUT THE LARGER OF TWO INPUTS.....	21
WORKED EXAMPLE 3 — A COUNTDOWN LOOP	22
2.7 ADDRESSING MODES.....	23
WORKED EXAMPLE 4 — IMMEDIATE VERSUS DIRECT ADDRESSING	23
2.8 OBJECT-ORIENTED LANGUAGES IN THE SPECIFICATION	23
WORKED EXAMPLE 5 — ENCAPSULATION USING AN ACCOUNT CLASS.....	25
SECTION 2 EXERCISES.....	26
QUICK CHECK.....	26
PRACTICE	26
EXAM-STYLE	26
CHECK YOURSELF	27
KEY FACTS	28

SECTION 3 — EXCHANGING DATA (SPEC 1.3)	29
3.1 COMPRESSION	29
3.2 ENCRYPTION AND HASHING	29
3.3 DATABASES	29
NORMALISATION TO 3NF	30
WORKED EXAMPLE 1 — NORMALIZING AN ENROLMENT TABLE	31
CAPTURING, SELECTING, MANAGING AND EXCHANGING DATA	32
SQL	32
WORKED EXAMPLE 2 — SELECTING AND JOINING RECORDS	32
WORKED EXAMPLE 3 — INSERTING, UPDATING AND DELETING RECORDS	33
REFERENTIAL INTEGRITY	33
TRANSACTIONS, LOCKING AND REDUNDANCY	34
3.4 NETWORKS AND THE INTERNET	35
NETWORK HARDWARE	35
THREATS AND PROTECTION	36
3.5 WEB TECHNOLOGIES	36
WORKED EXAMPLE 4 — HTML, CSS AND JAVASCRIPT WORKING TOGETHER	37
SEARCH ENGINES AND PAGERANK	37
WORKED EXAMPLE 5 — ONE PAGERANK CALCULATION, THREE ROUNDS	38
SECTION 3 EXERCISES	38
QUICK CHECK	38
PRACTICE	39
CHECK YOURSELF	40
KEY FACTS	41
 SECTION 4 — DATA TYPES, DATA STRUCTURES AND BOOLEAN ALGEBRA (SPEC 1.4)	 42
4.1 PRIMITIVE DATA TYPES AND NUMBER REPRESENTATION	42
POSITIVE BINARY AND HEXADECIMAL	42
WORKED EXAMPLE 1 — CONVERTING BETWEEN DENARY, BINARY AND HEXADECIMAL	42
BINARY INTEGER ARITHMETIC	43
WORKED EXAMPLE 2 — BINARY ADDITION	43
SIGN-AND-MAGNITUDE AND TWO'S COMPLEMENT	43
WORKED EXAMPLE 3 — REPRESENTING A NEGATIVE NUMBER IN TWO'S COMPLEMENT	44
FLOATING POINT	45
WORKED EXAMPLE 4 — READING A FLOATING-POINT VALUE	45
BITWISE OPERATIONS AND MASKS	45
WORKED EXAMPLE 5 — APPLYING BITWISE MASKS	45
CHARACTER SETS AND UNICODE ENCODING	46
UNICODE ENCODING FORMATS	46
WORKED EXAMPLE 6 — COMPARING UTF-8, UTF-16 AND UTF-32	47
4.2 DATA STRUCTURES	48
OPERATIONS ON STRUCTURES	48
WORKED EXAMPLE 7 — ARRAY-BACKED STACK	50
WORKED EXAMPLE 8 — CIRCULAR QUEUE — REAR WRAPS AROUND	51
4.3 BOOLEAN ALGEBRA, TRUTH TABLES AND CIRCUITS	52
KARNAUGH MAPS	53
D-TYPE FLIP-FLOPS AND ADDERS	53
SECTION 4 EXERCISES	54
QUICK CHECK	54
PRACTICE	54

EXAM-STYLE	54
CHECK YOURSELF	55
KEY FACTS	56
SECTION 5 — LEGAL, MORAL, CULTURAL AND ETHICAL ISSUES (SPEC 1.5)	57
5.1 COMPUTING-RELATED LEGISLATION.....	57
5.2 MORAL, ETHICAL, SOCIAL AND CULTURAL ISSUES.....	57
5.3 WRITING AN EXTENDED RESPONSE	58
SECTION 5 EXERCISES.....	58
QUICK CHECK.....	58
PRACTICE	58
EXAM-STYLE	58
CHECK YOURSELF	59
KEY FACTS	60
SECTION 6 — ELEMENTS OF COMPUTATIONAL THINKING (SPEC 2.1)	61
6.1 THINKING ABSTRACTLY	61
6.2 THINKING AHEAD.....	61
6.3 THINKING PROCEDURALLY	61
6.4 THINKING LOGICALLY	61
6.5 THINKING CONCURRENTLY	61
SECTION 6 EXERCISES.....	62
QUICK CHECK.....	62
PRACTICE	62
EXAM-STYLE	62
CHECK YOURSELF	63
KEY FACTS	64
SECTION 7 — PROBLEM SOLVING AND PROGRAMMING (SPEC 2.2)	65
7.1 CORE PROGRAMMING CONSTRUCTS	65
WORKED EXAMPLE 1 — SEQUENCE + SELECTION + ITERATION	65
7.2 RECURSION	66
WORKED EXAMPLE 2 — RECURSIVE FACTORIAL	66
7.3 VARIABLES, SCOPE, MODULARITY AND PARAMETER PASSING	66
7.4 IDES AND DEBUGGING	66
7.5 OBJECT-ORIENTED TECHNIQUES	67
WORKED EXAMPLE 3 — INHERITANCE AND POLYMORPHISM	68
7.6 COMPUTATIONAL METHODS	69
WHAT MAKES A PROBLEM SUITABLE FOR A COMPUTATIONAL METHOD?	69
SECTION 7 EXERCISES.....	69
QUICK CHECK.....	69
PRACTICE	69
EXAM-STYLE	69
CHECK YOURSELF	70
KEY FACTS	71
SECTION 8 — ALGORITHMS (SPEC 2.3).....	72

8.1 DESIGNING AND ANALYSING ALGORITHMS	72
8.2 ALGORITHMS FOR DATA STRUCTURES.....	73
WORKED EXAMPLE 1 — SINGLY LINKED LIST: POINTER UPDATES	74
WORKED EXAMPLE 2 — BREADTH-FIRST TREE TRAVERSAL.....	75
WORKED EXAMPLE 3 — DEPTH-FIRST POST-ORDER TREE TRAVERSAL	76
8.3 SEARCHING AND SORTING	76
WORKED EXAMPLE 4 — BINARY SEARCH	78
WORKED EXAMPLE 5 — SIMPLE QUICKSORT — CLEAR BUT NOT IN-PLACE.....	80
8.4 GRAPH TRAVERSAL, DIJKSTRA AND A*	82
WORKED EXAMPLE 6 — DIJKSTRA FROM A TO E.....	83
SECTION 8 EXERCISES	84
QUICK CHECK.....	84
PRACTICE	84
EXAM-STYLE	84
CHECK YOURSELF	85
KEY FACTS	86
 APPENDIX A — OCR-STYLE PSEUDOCODE QUICK REFERENCE	 87
ASSIGNMENT, INPUT AND OUTPUT	87
SELECTION AND ITERATION	87
FUNCTIONS, PROCEDURES AND ARRAYS.....	87
OPERATORS AND USEFUL IDEAS	87
 APPENDIX B — THE PROGRAMMING PROJECT (NEA)	 89
ANALYSIS — 10 MARKS	89
DESIGN — 15 MARKS.....	89
DEVELOPMENT AND TESTING — 25 MARKS	89
EVALUATION AND DOCUMENTATION — 20 MARKS	89
 APPENDIX C — SPECIFICATION COVERAGE CHECKLIST	 90
 REFERENCES AND VERIFICATION NOTES.....	 92
VERIFICATION AND EDITION NOTES.....	92
REFERENCES AND FURTHER READING.....	92

About this book

This book is written for students taking OCR A Level Computer Science H446. It assumes secure GCSE knowledge and then builds the extra precision A-level questions demand: accurate technical vocabulary, worked methods, traceable algorithms, and enough programming detail to connect theory to code. Each section follows the OCR specification order. The main text explains the ideas; worked examples make the methods concrete; “Common misconception” boxes target mistakes that lose marks; and the exercises move from retrieval to explanation and extended response. The Student Edition deliberately does not include a full answer key.

Coverage notes. This edition was rebuilt against OCR H446 specification version 2.7. It explicitly includes the areas that are easy to miss in shorter revision guides: programming paradigms, high- and low-level languages, Little Man Computer (LMC), addressing modes, network hardware and models, binary/hexadecimal arithmetic, floating-point arithmetic, data-structure operations, tree traversal and OCR-style pseudocode.

Legislation notes. OCR H446 specification v2.7 still names the Data Protection Act 1998. For current UK law, data protection is governed principally by the Data Protection Act 2018 and the UK GDPR; the Data (Use and Access) Act 2025 amends parts of that framework rather than replacing it. This book therefore distinguishes the wording students see in the OCR specification from the current legal position.

Assessment at a glance

Component	Assessment	Weighting
01 Computer systems	140 marks · 2 h 30 min written paper · no calculator	40%
02 Algorithms and programming	140 marks · 2 h 30 min written paper · no calculator	40%
03/04 Programming project	70 marks · non-exam assessment	20%

OCR uses short factual questions, multi-step calculations and traces, scenario-based explanations, code/pseudocode questions, and extended-response questions where the quality of discussion is assessed. Recent public papers continue to examine LMC, addressing modes and the distinction between assembly and high-level languages.

How to use the book

- Do the retrieval starter before reading the section.
- Write or trace the code: programming ideas become easier when you execute them.
- For “explain” questions, link a technical cause to a consequence in the scenario.
- For 9-mark discussions, develop both sides, use the scenario, and finish with a justified judgement.
- Use the specification checklist at the back before final revision

This book has no answer key. At the end of each section, the quick revision page lets you test yourself. Try the Check yourself questions from memory first, then use the Key facts list on the next page to check your own answers.

Section 1 — Processors, input, output and storage (spec 1.1)

Before you start — retrieve from GCSE: What are the roles of the CPU, RAM and secondary storage? What does a register hold? What is the von Neumann architecture?

1.1 Structure and function of the processor

The processor repeatedly fetches instructions from memory, decodes them and executes them. At A Level you must name the components and describe how values move between them.

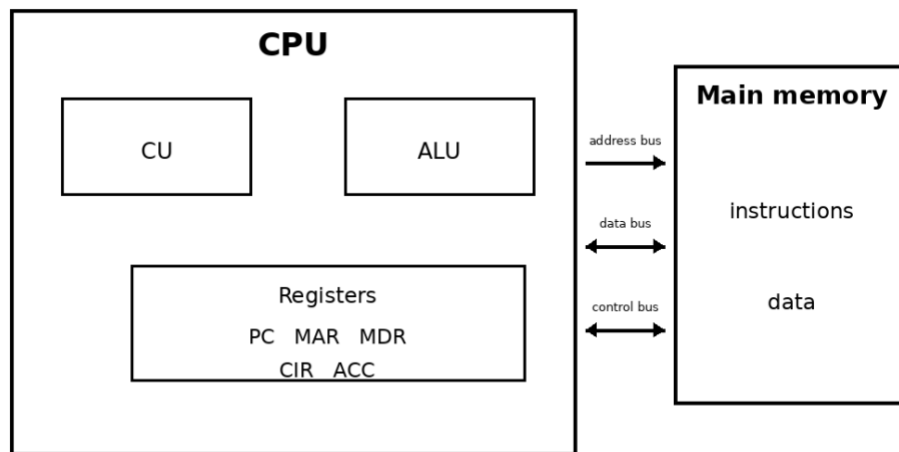


Fig 1.1 — CPU components, registers and buses

Component	Role
ALU	Performs arithmetic, comparisons and Boolean/bitwise operations.
Control Unit (CU)	Decodes instructions and sends control signals that coordinate the processor and other components.
PC	Holds the address of the next instruction to fetch.
MAR	Holds the address in memory currently being accessed.
MDR	Holds data or an instruction moving to or from memory.
CIR	Holds the current instruction while it is decoded and executed.
ACC	Stores intermediate arithmetic/logic results in accumulator-based architectures.
Address bus	Carries addresses, usually from processor to memory/devices.
Data bus	Carries data and instructions between components; normally bidirectional.
Control bus	Carries control signals such as read/write and timing signals.

1.2 The fetch–decode–execute cycle

Fetch

The address of the next instruction is copied from the **Program Counter (PC)** into the **Memory Address Register (MAR)**.

The **PC is incremented** so that it points to the next instruction in sequence.

The address held in the MAR is sent to memory along the **address bus**, and the Control Unit sends a memory-read signal along the **control bus**.

The instruction is retrieved from memory and transferred along the **data bus** into the **Memory Data Register (MDR)**.

The instruction is copied from the MDR into the **Current Instruction Register (CIR)**.

Decode

The **Control Unit (CU)** decodes the instruction held in the CIR, identifying the operation (opcode) and any required operand information.

Execute

7. The instruction is executed. If it is a taken branch instruction, the PC is updated with the branch target address rather than continuing with the next sequential instruction.

Exam language. Remember the register sequence: **PC → MAR; memory → MDR → CIR**. The PC is incremented during the fetch stage, but a taken branch can subsequently change its value.

1.3 Performance and pipelining

Clock speed is the number of clock cycles per second. More cores can execute independent instruction streams at the same time when software can be parallelised. Cache is small, fast memory close to the processor that reduces the time spent waiting for frequently needed instructions and data.

Common misconception. A higher clock speed does not mean a faster computer. Clock speed only counts cycles per second; what a processor achieves in each cycle depends on its architecture, its cores and whether the data it needs is already in cache.

Pipelining overlaps stages of different instructions. While instruction 1 executes, instruction 2 can decode and instruction 3 can fetch. This increases instruction throughput; it does not usually reduce the latency of a single instruction. A branch can force prefetched or partly processed instructions on the wrong path to be discarded, reducing the benefit.

Common misconception. Pipelining is not the same as having more cores. Pipelining keeps a single core busy by overlapping the stages of different instructions, while multicore runs separate instruction streams side by side, and a question about one will not accept an answer about the other.

1.4 Processor architectures

Von Neumann architecture stores instructions and data in the same memory and uses shared pathways. Harvard architecture has separate instruction and data memories/buses, so an instruction and a data value can be accessed at the same time. Contemporary processors often use a modified Harvard approach: separate instruction and data caches close to the CPU, with a unified main memory beyond them.

1.5 CISC, RISC, GPUs and parallel systems

Feature	CISC	RISC
Instruction set	Larger, with complex instructions and many addressing forms.	Smaller and simpler, designed for efficient decoding/execution.
Program length	Often fewer instructions.	Often more instructions for the same task.
Hardware/compiler	More complexity in processor hardware.	Simpler instruction design places more work on the compiler.
Pipelining	Variable instruction behaviour can complicate pipelines.	Regular instruction design is generally easier to pipeline.
Typical examples	x86 desktop/server family.	ARM family in phones, embedded devices and increasingly desktops/servers.

A GPU contains many simpler processing units and is designed for high-throughput parallel work. Graphics rendering is one example; matrix operations for machine learning, simulation and video processing are others. A parallel system divides a problem so multiple processors/cores work at the same time. Performance depends on how much of the problem is genuinely parallel and how much communication is required.

1.6 Input, output and storage

Questions often give a scenario and ask you to **select and justify an appropriate device**. Do not simply name a device: link its features to the requirements of the situation. Useful factors include **accuracy, speed, durability, capacity, portability, cost, power consumption and the environment in which it will be used**.

Input devices

Input devices turn something in the world into data. The exam reaches for the devices a keyboard cannot be: sensors where no person is present, barcode and RFID readers where speed and accuracy matter, MICR where a forgery must fail.

Technology	Strengths	Limitations / typical uses
Keyboard, mouse, joystick	Fast, accurate entry of free text and commands by a trained user.	Needs a person present and able; poor hands-free, gloved or wet; repetitive-strain risk in long use; desktops and laptops.
Touchscreen	Direct manipulation with no separate pointer; a sealed front suits kiosks and phones.	Gloves and water interfere; precision limited by screen size; adds cost and power draw; phones, tablets, ticket machines.

Sensors (temperature, light, pressure, motion)	Continuous, objective readings, faster and more consistent than a person; work where a person cannot.	Analogue readings need an ADC and calibration; a faulty sensor feeds wrong data straight into a control loop; greenhouses, weather stations, phones.
Barcode and QR reader	Near-instant entry of coded data with very low error; readers are cheap.	Reads only what the label carries; dirty or damaged barcodes fail, though QR codes tolerate some damage; retail, ticketing, asset tracking.
RFID and NFC	No line of sight needed, and many tags read at once (RFID); NFC suits close-range payment and pairing.	Tags and readers cost more than barcodes; metal and water detune tags; stock control, access cards, contactless payment.
Magnetic stripe and MICR	MICR characters are fast to read and hard to forge, so banks trust them; stripes are cheap to produce.	Stripes wear, copy easily and hold little data; cheques (MICR) and legacy cards (stripes).
Scanner, camera, microphone	Capture a whole document, scene or sound as data; OCR can recover editable text.	OCR accuracy falls on poor print or handwriting; files are large; background noise defeats speech; archiving, passport gates, voice interfaces.

Output devices

Output devices turn data back into something a person or a machine can act on. Where the output changes the physical world, the device is an actuator and the whole system is a control system: the computer's digital decision leaves through driver circuitry and moves a motor, a valve or a heater.

Technology	Strengths	Limitations / typical uses
Monitor (LCD, OLED, e-ink)	Fast, sharp, colour output for interactive work; e-ink stays readable in sunlight and draws almost no power when static.	LCD and OLED need constant power; OLED can burn in on static images; e-ink refreshes slowly and in limited colour; general use and e-readers.
Laser printer	Fast, crisp text and low cost per page at volume.	Higher purchase cost and footprint; photo quality below inkjet; offices and exam paperwork.
Inkjet printer	Cheap hardware and good photo quality.	Slow, expensive ink, heads clog when idle; home use.
3D printer	Turns a CAD model into a physical object in-house.	Slow, material limits, finishing needed; prototyping and design work.
Speakers and headphones	Immediate audio feedback; headphones keep it private.	No use to a deaf or hard-of-hearing user, so an audio alert needs a visual or haptic partner; noise disturbs others; media, alerts, screen readers.
Actuators (motors, valves, heaters)	Let a computer change the physical world; the output side of every control system.	Need driver circuitry, and a DAC where the control signal is analogue rather than simply switched on and off; a stuck or oversized actuator acts on wrong data decisively; greenhouses, robots, braking systems.

Storage introduction

Computers use primary memory and secondary storage to hold data and instructions. RAM provides fast, temporary working memory and is volatile, meaning its contents are lost when power is removed. Secondary storage, such as HDDs, SSDs and optical discs, is non-volatile and retains data without power.

Technology	Strengths	Limitations / typical uses
------------	-----------	----------------------------

Magnetic	Very high capacity and low cost per GB.	Moving parts; slower random access; desktops, servers, backups.
Flash/solid state	Fast, silent, shock resistant, low power.	Higher cost per GB; finite write endurance; laptops, phones, embedded devices.
Optical	Cheap removable media and useful for distribution/archival niches.	Lower capacity and speed than modern alternatives.
RAM	Fast working memory, directly accessed by CPU.	Volatile: contents are lost without power.
ROM	Non-volatile memory for firmware or fixed instructions/data.	Not used as general read/write working memory.
Cloud storage	Presents remote storage as accessible over a network; supports sharing and off-site copies.	Depends on connectivity, provider availability, security and ongoing service.

Common misconception. Cloud storage is not the same as virtual memory. Cloud storage provides access to data held on remote servers over a network. Virtual memory is an operating-system technique that uses secondary storage to extend the available address space beyond physical RAM, although accessing data on secondary storage is much slower than accessing RAM.

Worked example 1 — selecting and justifying devices

Choosing an appropriate input, output or storage device depends on the requirements of the situation. The most suitable device is not necessarily the fastest or most expensive; it is the one whose properties best meet those requirements.

Scenario A – An automated greenhouse

A greenhouse must monitor temperature and automatically activate a heater when the temperature falls below a set value, even when nobody is present.

Recommended devices: Temperature sensor (input) and heater controlled by an actuator or switching circuit (output).

Justification: The sensor continuously measures the temperature without human intervention. The computer compares the reading with the required temperature and switches the heater on when necessary. This allows the greenhouse to maintain suitable conditions automatically.

Scenario B – A supermarket checkout

A supermarket processes hundreds of products every hour and needs to minimise errors when identifying items.

Recommended device: Barcode reader (input).

Justification: A barcode reader captures product codes quickly and accurately without requiring staff to type them manually. This reduces input errors and speeds up transactions.

Scenario C – A bank processing cheques

A bank needs to read information printed on cheques quickly and reliably, including cheques that may have been stamped or marked during processing.

Recommended device: MICR reader (input).

Justification: Magnetic Ink Character Recognition (MICR) reads characters printed using magnetic ink. These characters can be recognised reliably by automated processing equipment, even when some ordinary visual markings are present. The specialised printing also makes casual alteration more difficult, although it does not eliminate forgery.

Scenario D – A school storing video recordings

A school needs to store several terabytes of lesson recordings while keeping costs low. The recordings are accessed only occasionally.

Recommended technology: Magnetic hard disk drive (HDD).

Justification: HDDs provide high storage capacity at a relatively low cost per gigabyte. Although slower than SSDs for random access, they are suitable when capacity and cost are more important than maximum access speed.

Scenario E – A laptop used while travelling

A laptop must start quickly, load applications efficiently and withstand frequent movement.

Recommended technology: Solid-state drive (SSD).

Justification: SSDs offer fast data access and contain no moving mechanical parts, making them more resistant to physical shocks than HDDs. Although they generally cost more per gigabyte, their speed and durability make them suitable for portable computers.

Trace it: Each scenario begins with a requirement that determines the most suitable technology. The greenhouse needs unattended monitoring and control, so sensors and actuators are appropriate. The supermarket prioritises fast, accurate data entry, making a barcode reader suitable. The bank needs reliable automated character recognition, so MICR is appropriate. The school prioritises capacity and cost, favouring magnetic storage, while the travelling laptop prioritises speed and resistance to physical shocks, favouring solid-state storage. The key is to connect the device's technical properties to the specific needs of the situation.

Exam language. When justifying a device, identify the requirement, name a suitable device and explain how its properties meet that requirement. A device name alone is not a complete justification.

Section 1 exercises

Quick check

1. During the fetch stage, which registers hold (a) the address being accessed, (b) the fetched instruction, and (c) the address of the next instruction?
2. State the difference between the data bus and address bus.
3. What is the difference between von Neumann and Harvard architecture?

Practice

1. Describe the fetch stage of the fetch–decode–execute cycle using registers and buses.
2. Explain why increasing the number of CPU cores may not double the speed of a program.
3. A wearable medical sensor needs secondary storage. Justify flash storage rather than magnetic storage.
4. A library self-service machine issues loans without staff. Identify one input device and one output device the machine needs and justify each against the scenario.

Exam-style

1. A game console manufacturer is choosing processor architecture, number of cores, cache size and storage. Explain how each choice can affect performance and cost. [9]
2. Compare CISC and RISC processors for an embedded battery-powered device. [6]
3. A car park replaces its ticket machines with an automatic barrier that reads vehicle number plates. (a) Name one input device and one output device the system needs. [2] (b) Explain why reading the number plate suits this situation better than issuing each driver a barcoded ticket, and why the barrier needs an actuator rather than a screen. [4]

Section 1 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. Name the register that holds the address of the next instruction. What happens to that address if the current instruction is a branch?
2. Why can pipelining lose work after a branch?
3. A colleague claims doubling the core count will halve a program's runtime. What must you establish before agreeing?
4. How do pipelining and multicore parallelism differ?

Key facts

- PC gives the address of the next instruction; MAR holds the current memory address; MDR carries the value; CIR holds the current instruction; ACC stores working results.
- Fetch uses address, data and control buses; a branch changes the PC from its normal next-instruction value.
- Clock speed, cores and cache affect performance, but benefits depend on workload and bottlenecks.
- Pipelining overlaps instruction stages and can lose work after a branch.
- von Neumann shares instruction/data memory; Harvard separates them; contemporary CPUs often blend the ideas.
- GPU and multicore performance come from parallel work, not simply “faster clock = faster computer”.

Section 2 — Software, software development and programming languages (spec 1.2)

Before you start — retrieve from GCSE: What does an operating system do? What is the difference between a compiler and an interpreter? What is a high-level language?

2.1 Systems software and the operating system

An operating system provides a platform for applications and manages hardware resources. Its responsibilities include processor scheduling, memory management, file management, device management, security/user accounts, user interfaces and handling interrupts.

Memory management: paging, segmentation and virtual memory

Paging divides a process's virtual address space into fixed-size pages and RAM into equally sized page frames. A page table records which frame holds each page. Because pages can occupy any free frames, a program does not need one contiguous block of physical RAM.

Segmentation divides a program into variable-sized logical sections such as code, stack or data. Segments match the structure of the program but can suffer from external fragmentation. Virtual memory keeps pages that are not currently needed on secondary storage. It allows programs to use an address space larger than physical RAM, but frequent page transfers cause thrashing and severe slowdown.

Common misconception. Paging and virtual memory are not the same thing. Paging is the scheme that divides memory into fixed-size pages and frames, and virtual memory is one use of it: keeping the pages a program is not using on secondary storage so its address space can exceed the RAM installed.

Interrupts and ISRs

An interrupt is a signal that requests CPU attention. The processor checks for interrupts at the end of each fetch–decode–execute cycle, which is why a running instruction is allowed to finish first.

Examples include keyboard input, a timer, completion of an I/O operation and software exceptions. The CPU saves the contents of its registers onto a stack so it can resume later, identifies the interrupt, runs an interrupt service routine (ISR), restores the previous state and continues the interrupted program. Priority and masking determine which interrupts can be serviced immediately.

Scheduling

Algorithm	How it works	Trade-off
Round robin	Processes take fixed time slices in turn.	Responsive and fair; context-switch overhead.
First come first served	Runs jobs in arrival order, normally to completion.	Simple; a long job can delay everything behind it.
Shortest job first	Chooses the shortest total job.	Good average waiting time if durations are known; long jobs may starve.
Shortest remaining time	Pre-emptive SJF: a shorter arriving job may interrupt the current one.	Responsive to short jobs; starvation risk and more switching.
Multi-level feedback queues	Several priority queues; processes can move between queues based on behaviour.	Flexible and practical; more complex to configure.

Types of operating system

Type	Key idea	Example context
Multitasking	One user runs many programs apparently at once.	Desktop/laptop OS.
Multi-user	Many users share one computer/system resources.	Server or mainframe.
Distributed	Several networked computers are presented/managed as a coordinated system.	Cluster/distributed service.
Embedded	Purpose-built for a device, often with constrained resources.	Router, appliance, industrial controller.
Real-time	Guarantees responses within defined timing constraints.	Airbag, industrial control, medical device.

BIOS, device drivers and virtual machines

Firmware starts the machine, performs initial hardware checks and begins loading the operating system. The specification uses the term BIOS; modern PCs often use UEFI firmware, but the exam concept is the same startup role. A device driver translates generic OS requests into commands for a particular hardware device.

A virtual machine is software that behaves like a machine. A system VM can run one operating system inside another or on a hypervisor. A process VM can execute intermediate code: for example, bytecode is translated/executed by a suitable runtime on each platform.

2.2 Applications, utilities and software licensing

Applications perform tasks for the user: word processing, CAD, accounting, browsers and games. Utilities maintain, protect or optimise a system: backup, compression, anti-malware, encryption or disk tools. In a scenario, justify an application by matching its features to the user's requirements.

Open-source software makes source code available under a licence that permits specified forms of use, study and modification. Closed source/proprietary software normally distributes executables under a licence while keeping source code private. Neither model automatically guarantees quality, security or price; compare support, control, customisation, auditability, vendor dependence and licensing cost.

2.3 Translators and the path from source to execution

Translator / tool	Purpose
Assembler	Converts assembly mnemonics into machine code.
Compiler	Translates a whole high-level program (or unit) into target/object code before execution.
Interpreter	Translates/executes source or intermediate instructions at run time.
Linker	Combines object modules with required library code and resolves references.
Loader	Places the executable into memory, performs final loading tasks and starts execution.
Libraries	Pre-written, tested code a program can call (for example, maths or graphics routines). The linker resolves references to the library routines the program needs.

A typical compilation pipeline: lexical analysis removes whitespace and comments, turns the remaining characters into tokens and records identifiers in a symbol table; syntax analysis checks tokens against the grammar and builds a structural representation; code generation produces target/object instructions; optimisation changes the code to improve speed or resource use without changing its behaviour.

Common misconception. A compiled program does not need its compiler to run, and an interpreter does not avoid translation. The compiler translates once and leaves behind a file that runs on its own; the interpreter translates every time the program runs, which is why it must still be installed on the machine running it.

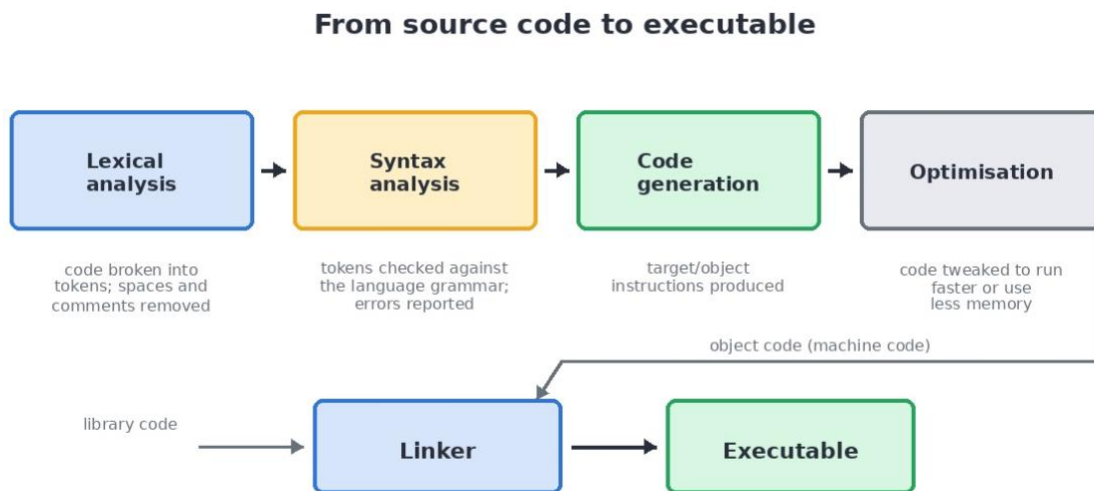


Fig 2.1 — The stages of compilation, then linking

Exam language. “Compiler = faster” is not a complete comparison. Explain when translation happens, what output is produced, how errors are reported, whether the translator is needed at run time, portability and optimisation.

2.4 Software development methodologies

Method	Approach	Best fit / limitation
Waterfall	Sequential stages with sign-off before the next stage.	Stable requirements, heavy documentation; expensive to revisit earlier decisions.
Agile	Short iterations delivering working increments with frequent stakeholder feedback.	Changing requirements; needs regular collaboration.
Extreme programming (XP)	Agile practices such as pair programming, test-first development, continuous integration and very small releases.	Small skilled teams and rapidly changing code; intensive collaboration.
Spiral	Repeated cycles driven by risk analysis.	Large/high-risk projects; management overhead.
RAD	Rapid prototyping and user workshops; prototypes are refined quickly.	When users discover requirements by seeing/using prototypes.

The development process still involves analysis, design, implementation/development, testing, deployment and evaluation/maintenance. Methodologies change how those stages are ordered, repeated and documented.

2.5 Programming paradigms and language levels

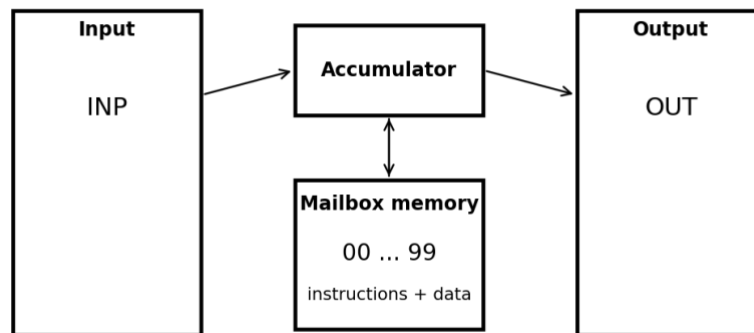
A programming paradigm is a general style for organising computation. Procedural programming organises instructions into procedures/functions operating on data. Object-oriented programming organises

data and behaviour into objects created from classes. Other paradigms exist, but OCR expects you to understand why different paradigms suit different problems rather than treat one as universally superior.

Low-level / assembly	High-level
Close to processor instructions and hardware resources.	More abstract; syntax and data structures are closer to the problem domain.
Usually processor/instruction-set specific.	Often portable when a suitable compiler/interpreter exists.
Fine control and low overhead are possible.	Faster development, easier maintenance and better readability.
Translated by an assembler.	Translated by a compiler/interpreter or via intermediate code.

Assembly language replaces binary opcodes and addresses with mnemonics and labels. It is useful where hardware control, precise timing, very small memory footprint or specialist processor features matter. The cost is portability and programmer productivity.

2.6 Little Man Computer (LMC)



LMC models a stored-program computer: the same memory holds instructions and data.

Fig 2.2 — The LMC model

Little Man Computer is a teaching model of a simple stored-program computer. Numbered mailboxes represent memory; the accumulator is the working register; input and output provide simple I/O. OCR H446 section 1.2.4 explicitly requires students to follow and write simple programs using the Little Man Computer instruction set.

Mnemonic	Meaning	What changes
INP	Input a value	Input value enters the accumulator.
OUT	Output	Current accumulator value is output.
LDA label	Load	Copy value from a mailbox into the accumulator.
STA label	Store	Copy accumulator into a mailbox.
ADD label	Add	Add mailbox value to accumulator.

SUB label	Subtract	Subtract mailbox value from accumulator.
BRA label	Branch always	PC changes to target label.
BRZ label	Branch if zero	Branch when accumulator is zero.
BRP label	Branch if non-negative	Branch when accumulator is zero or positive.
HLT	Halt	Stop execution.
DAT value	Data location	Reserve a mailbox, optionally with an initial value.

The usual confusion: BRP branches when the accumulator is zero too, because zero is non-negative. BRZ branches only on zero. Some LMC simulators also display numeric opcodes. OCR exam questions use mnemonics, so this book uses mnemonics only.

Worked example 1 — input two numbers and output their sum

The following LMC program accepts two numbers, adds them together and outputs the result.

```

    INP
    STA FIRST
    INP
    ADD FIRST
    OUT
    HLT
FIRST DAT 0

```

Suppose the user enters 12 followed by 7.

Instruction	What happens	Accumulator
INP	The first input, 12, enters the accumulator.	12
STA FIRST	The value 12 is copied into the mailbox FIRST.	12
INP	The second input, 7, replaces the accumulator's value.	7
ADD FIRST	The stored value 12 is added to 7.	19
OUT	The accumulator value, 19, is output.	19
HLT	The program stops.	19

Output: 19

Trace it: The first INP places 12 in the accumulator. STA FIRST saves this value in the mailbox labelled FIRST, without changing the accumulator. The second INP replaces the accumulator's contents with 7, but the original value of 12 remains stored in memory. ADD FIRST retrieves the stored value and adds it to the accumulator: $7 + 12 = 19$. OUT displays 19, and HLT stops the program.

Notice that STA copies a value into memory, while ADD uses the value stored there. Without STA FIRST, the first input would be lost when the second INP replaces the accumulator's contents.

Worked example 2 — continuously output the larger of two inputs

The following LMC program repeatedly accepts two numbers, compares them and outputs the larger value.

```
START    INP
          STA A
          INP
          STA B
          SUB A
          BRP BLARGER
          LDA A
          OUT
          BRA START
BLARGER  LDA B
          OUT
          BRA START
A        DAT 0
B        DAT 0
```

The program compares the numbers by subtracting the first input (A) from the second input (B).

- If $B - A$ is zero or positive, B is greater than or equal to A, so the program branches to BLARGER.
- If $B - A$ is negative, A is larger, so the branch is skipped.

Example A — the second number is larger

Suppose the user enters 12 followed by 20.

Instruction	What happens	Accumulator
INP	Read the first input, 12.	12
STA A	Store 12 in mailbox A.	12
INP	Read the second input, 20.	20
STA B	Store 20 in mailbox B.	20
SUB A	Calculate $20 - 12$.	8
BRP BLARGER	8 is positive, so the branch is taken.	8
LDA B	Load the larger value, 20.	20
OUT	Output 20.	20
BRA START	Return to START for another pair of inputs.	20

Output: 20

Example B — the first number is larger

Now suppose the user enters 20 followed by 12.

Instruction	What happens	Accumulator
INP	Read the first input, 20.	20
STA A	Store 20 in mailbox A.	20
INP	Read the second input, 12.	12
STA B	Store 12 in mailbox B.	12
SUB A	Calculate $12 - 20$.	-8*
BRP BLARGER	The result is negative, so the branch is not taken.	-8*
LDA A	Load the larger value, 20.	20
OUT	Output 20.	20
BRA START	Return to START for another pair of inputs.	20

Output: 20

The negative result is shown mathematically. Some LMC simulators represent negative results differently internally, but the branch decision follows the negative condition.

Trace it: The first input is stored in mailbox A and the second in mailbox B. After the second input, the accumulator contains B. SUB A subtracts A from B, leaving the difference in the accumulator. The instruction BRP BLARGER checks that result. If it is zero or positive, execution jumps to BLARGER, where LDA B loads the second number and OUT displays it. Otherwise, execution continues with LDA A, which loads and outputs the first number

Both paths finish with `BRA START`, returning execution to the beginning. Unlike the first worked example, this program has no `HLT` instruction. It continues processing pairs of inputs until execution is stopped externally.

Important: `BRP` includes zero. If both inputs are equal, the subtraction gives zero, so the program takes the branch to `BLARGER`. This still produces the correct result because both numbers are the same.

Trace discipline. When tracing LMC programs, record the instruction being executed, the accumulator's value, any changes to memory and the output produced. Pay particular attention to branch instructions: they determine which instruction executes next. Labels such as `START` and `BLARGER` identify locations in the program; they are not instructions themselves.

Worked example 3 — a countdown loop

The following LMC program uses a loop to count down from 4 to 1. It outputs each number and stops when the counter reaches zero.

```

LOOP   LDA COUNT
        OUT
        SUB ONE
        STA COUNT
        BRZ END
        BRA LOOP
END     HLT
COUNT DAT 4
ONE     DAT 1

```

The mailbox `COUNT` initially stores 4, while `ONE` stores the value 1, which is subtracted during each iteration.

Iteration	COUNT at start	Value output	COUNT after subtraction	BRZ END
1	4	4	3	Not taken
2	3	3	2	Not taken
3	2	2	1	Not taken
4	1	1	0	Taken

Output: 4, 3, 2, 1

Trace it: The program begins at `LOOP`, where `LDA COUNT` copies the current value of `COUNT` into the accumulator. Initially, this is 4. `OUT` displays the accumulator's value without changing it.

`SUB ONE` subtracts 1 from the accumulator, reducing its value from 4 to 3. `STA COUNT` then stores the updated value back into the mailbox `COUNT`. Importantly, storing the value does not change the accumulator, so `BRZ END` can still test whether the result is zero.

Since the accumulator contains 3, `BRZ END` is not taken. The instruction `BRA LOOP` returns execution to the beginning of the loop.

The process repeats, outputting 3, then 2, then 1. On the fourth iteration, `SUB ONE` reduces the accumulator from 1 to 0, and `STA COUNT` stores 0. This time, `BRZ END` branches to `END`, where `HLT` stops execution.

Notice that **0 is never output**. The `OUT` instruction executes before the subtraction and zero check.

Consequently, the final value displayed is 1.

Understanding the accumulator: The accumulator is the working register used for arithmetic and data transfer. Values that must be retained between iterations are stored in memory mailboxes declared using `DAT`. Instructions such as `LDA` and `STA` transfer values between the accumulator and these mailboxes.

Labels such as `LOOP` and `END` make the program easier to read. The assembler translates these labels into numeric memory addresses so that branch instructions can locate their targets.

2.7 Addressing modes

An addressing mode tells the processor how to obtain an operand. OCR requires students to understand four addressing modes. LMC primarily demonstrates direct addressing; the other modes can be illustrated using generic assembly-language examples.

Mode	Meaning	Example idea
Immediate	The actual operand value is included in the instruction.	ADD #5 adds the literal value 5.
Direct	The instruction specifies the memory address containing the operand.	ADD FIVE in LMC adds the value stored in the mailbox labelled FIVE.
Indirect	The specified memory location contains another address, which identifies where the operand is stored.	LOAD (40) retrieves a value from the memory address stored at address 40.
Indexed	The effective memory address is calculated using a base address and an index or offset.	Useful for accessing successive array elements.

Worked example 4 — immediate versus direct addressing

In generic assembly notation:

ADD #5

The processor adds the literal value 5 to the accumulator. The value is part of the instruction.

In standard LMC, the equivalent operation uses direct addressing:

ADD FIVE

FIVE DAT 5

Here, FIVE labels a mailbox containing the value 5. The ADD instruction reads that mailbox and adds its contents to the accumulator.

Trace it: Suppose the accumulator initially contains 3. With immediate addressing, ADD #5 adds the literal value 5 directly to the accumulator, giving $3 + 5 = 8$.

With direct addressing, ADD FIVE first accesses the mailbox labelled FIVE, retrieves the value stored there (5), and adds it to the accumulator. The result is also $3 + 5 = 8$.

Both instructions produce the same result, but they obtain the operand differently. With immediate addressing, the value is contained in the instruction itself. With direct addressing, the instruction identifies a memory location containing the value.

Notice: ADD #5 is illustrative generic assembly notation, not a valid instruction in standard LMC. Standard LMC does not provide immediate, indirect or indexed addressing instructions.

Common misconception. Immediate addressing does not mean accessing fast memory. It means the operand value is included in the instruction itself. Direct addressing supplies the location of the operand, while indirect addressing supplies a location containing another address.

2.8 Object-oriented languages in the specification

Object-oriented programming (OOP) organises programs around objects that combine data with the operations performed on that data.

A **class** defines the attributes and methods that its objects will have. An **object** is an instance of a class, with its own state.

The main concepts are:

- **Attributes:** Variables that store an object's state.
- **Methods:** Procedures or functions that define an object's behaviour.
- **Encapsulation:** Combining attributes and methods within a class while controlling access to its internal state.

- **Inheritance:** Allowing a subclass to inherit and extend the attributes and methods of a parent class.
- **Polymorphism:** Allowing the same method call to produce different behaviour depending on the type of object receiving it.

Common misconception. Encapsulation is not simply placing variables and methods inside a class. An important part of encapsulation is controlling access to an object's internal state. Private attributes cannot be accessed directly from outside the class; public methods provide a controlled interface through which the state can be read or modified.

Common misconception. Encapsulation is not simply putting things inside a class. It means keeping the state private and letting the outside world reach it only through methods you control, which is why private balance and getBalance() in the listing below are the whole point rather than decoration.

Worked example 5 — encapsulation using an Account class

The following OCR-style pseudocode defines an Account class with a private attribute called balance. The class provides public methods for depositing money and retrieving the current balance.

```
class Account
    private balance

    public procedure new(startBalance)
        balance = startBalance
    endprocedure

    public procedure deposit(amount)
        if amount > 0 then
            balance = balance + amount
        endif
    endprocedure

    public function getBalance()
        return balance
    endfunction
endclass
```

The class contains three methods:

- `new(startBalance)` is the constructor. It runs when an object is created and initialises balance using the supplied starting value.
- `deposit(amount)` is a public procedure that increases the balance only when the amount is greater than zero.
- `getBalance()` is a public function that returns the current balance without modifying it.

Tracing the Account class

Four statements execute in sequence. Follow how the private attribute balance changes after each statement.

Statement	What happens	Balance after	Result
<code>myAccount = new Account(50)</code>	Creates an Account object and initialises its balance to 50.	50	Object assigned to <code>myAccount</code>
<code>myAccount.deposit(25)</code>	The condition <code>amount > 0</code> is true, so 25 is added to the balance.	75	No return value
<code>myAccount.deposit(-10)</code>	The condition <code>amount > 0</code> is false, so the deposit is ignored.	75	No return value
<code>myAccount.getBalance()</code>	The function reads and returns the current balance.	75	75

Trace it: The first statement creates an object called `myAccount`. Its constructor, `new(50)`, initialises the private attribute balance to 50.

The second statement calls `deposit(25)`. Since 25 is greater than zero, the condition is true and the balance increases from 50 to 75.

The third statement calls `deposit(-10)`. This time, the condition is false because -10 is not greater than zero. The body of the if statement is skipped, leaving the balance unchanged at 75. The procedure does not return a value, so the caller receives no explicit indication that the deposit was rejected.

Finally, `getBalance()` returns 75. Unlike `deposit()`, this is a function because it returns a value to the caller.

Why this demonstrates encapsulation: The attribute balance is declared private, so code outside the class cannot access or modify it directly. Instead, external code uses the public methods. The `deposit()` procedure controls how deposits change the balance, while `getBalance()` allows the balance to be read without exposing it for direct modification.

Notice: The `deposit()` method rejects amounts of zero or less, but the constructor does not validate the initial balance. This means an Account object could be created with a negative starting balance. A more robust implementation would validate `startBalance` before assigning it to the private attribute balance.

Exam language. When tracing an object-oriented program, identify the object's initial state, follow each method call in order, check whether conditions are satisfied, and record any changes to attributes. Remember that a procedure does not return a value, whereas a function returns a result.

Section 2 exercises

Quick check

1. What type of translator converts assembly language?
2. Name the four addressing modes required by OCR.
3. What does BRZ do in LMC?
4. State one difference between a real-time OS and a merely fast OS.
5. State the difference between a class and an object.
6. Which object-oriented feature lets a subclass reuse the attributes and methods of a parent class?
7. State two factors you would compare when choosing between open-source and closed-source software.
8. Name the scheduling algorithm in which a shorter job arriving can interrupt the job currently running.

Practice

1. Trace the LMC sum program for inputs 7 and 4, recording accumulator values after each arithmetic instruction.
2. Write LMC that inputs a number and outputs twice the number.
3. Compare a compiler and interpreter for a school programming environment.
4. Explain how paging differs from segmentation.
5. The Account class has no way to take money out. Write, in Exam Reference Language, a public withdraw procedure for the class. It must refuse any withdrawal that would leave the balance below zero.
6. A web server handles many short requests and occasional long report generations. Explain why round robin suits this workload better than first come first served.

Exam-style

1. A manufacturer is writing firmware for a small controller. Discuss reasons for and against using assembly rather than a high-level language. [9]
2. Write an LMC program that inputs three numbers and outputs the largest of the three, then halts. Annotate your branches. [8]
3. A hospital system has fixed safety requirements and must produce extensive evidence for regulators. Compare two development methodologies and justify a choice. [9]

Section 2 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. A program's address space is larger than installed RAM. Which memory-management feature makes this workable, and what symptom appears if it is overused?
2. After an interrupt is raised, why might the CPU not service it immediately?
3. In LMC, what is the difference between BRZ and BRP? Give an accumulator value for which they behave differently.
4. ADD #5: which addressing mode is this, and how does it differ from direct addressing?

Key facts

- Paging maps virtual pages to RAM frames; segmentation uses variable logical sections; virtual memory swaps needed pages between RAM and secondary storage.
- Interrupts run an ISR and then resume the interrupted program; schedulers decide which ready process gets CPU time.
- Assembler translates assembly; compiler translates before execution; interpreter translates/executes at run time; linker joins modules; loader places the program in memory; libraries supply pre-written code the program can call.
- OCR requires waterfall, agile, XP, spiral and RAD, plus the ability to justify a method for a scenario.
- LMC uses INP, OUT, LDA, STA, ADD, SUB, BRA, BRZ, BRP, HLT and DAT.
- Addressing modes: immediate = value; direct = address; indirect = address of address; indexed = base plus index.
- Open source publishes the source code under a licence permitting study and modification; closed source distributes executables and keeps the source private. Neither model guarantees quality, security or price.
- Scheduling: round robin gives fixed time slices; first come first served runs in arrival order; shortest job first and shortest remaining time favour short jobs and can starve long ones; multi-level feedback queues move processes between priority queues.
- A class defines attributes (state) and methods (behaviour), and an object is an instance of it. Encapsulation keeps state private behind methods; inheritance lets a subclass extend a parent; polymorphism lets the same call behave differently depending on the object.

Section 3 — Exchanging data (spec 1.3)

Before you start — retrieve from GCSE: What is the difference between lossy and lossless compression? What is an IP address? What does encryption do?

3.1 Compression

Lossless compression allows the original data to be reconstructed exactly; it is required for program files, text and other data where a changed bit could change meaning. Lossy compression permanently discards information judged less important to perception, which can give greater size reduction for images, audio and video.

Run-length encoding (RLE) stores repeated runs as a count and value. It is effective when long runs occur and can increase size when data changes constantly. Dictionary coding replaces recurring sequences with shorter references to entries in a dictionary; the dictionary and references together represent the original data.

3.2 Encryption and hashing

Symmetric encryption uses the same secret key for encryption and decryption. It is fast and suitable for large amounts of data, but sender and receiver need a secure way to share the key. Asymmetric encryption uses a public/private key pair. Data encrypted for a recipient using the recipient's public key can be decrypted with the recipient's private key.

Hashing maps arbitrary input to a fixed-size digest. A cryptographic hash is designed to be one-way and to change drastically when the input changes. Password systems should store salted password hashes rather than plaintext passwords; a stolen hash does not directly reveal the original password, although attackers can still test guesses. Hash tables use a (not necessarily cryptographic) hash function to choose a storage index.

Common misconception. A signature is not simply “encrypting the whole message with the private key”. In practice a digest of the message is signed using the private key and verified using the public key.

3.3 Databases

A flat-file database stores records in a single table or file without relationships between separate tables. This can lead to duplicated data and difficulties when records need to be updated.

A relational database organises data into related tables. Each table represents an entity, such as a student or course, while primary and foreign keys establish relationships between tables. Relational design helps reduce duplication, maintain data consistency and enforce integrity rules.

Term	Meaning
Primary key	An attribute, or combination of attributes, that uniquely identifies each record in a table.
Candidate key	An attribute, or combination of attributes, that could uniquely identify each record. One
Foreign key	Attribute that references a key in another table.
Secondary key	A non-primary attribute used to search for records. Its values do not have to be unique.
Index	An additional data structure that speeds up record retrieval, usually at the cost of extra storage and maintenance when data changes.
Entity relationship model	A model showing entities, their attributes and the relationships between them.

Common misconception. A foreign key is not necessarily “the other table’s primary key copied exactly” in every database design; it is a field whose values reference a candidate/primary key in another table.

Normalisation to 3NF

Normalisation is the process of organising data in relational tables to reduce unnecessary duplication and prevent insertion, update and deletion anomalies.

- **First Normal Form (1NF):** Each field contains a single atomic value, there are no repeating groups, and each record can be uniquely identified.
- **Second Normal Form (2NF):** The table is in 1NF, and every non-key attribute depends on the whole of each candidate key, not just part of a composite key.
- **Third Normal Form (3NF):** The table is in 2NF, and no non-key attribute depends transitively on a candidate key through another non-key attribute.

The practical goal is to store each fact in the appropriate place, reducing duplicated information and improving data consistency.

Worked example 1 — normalizing an enrolment table

A college stores student enrolments in a single table:

StudentID	StudentName	CourseID	CourseName	Tutor
317	Ada Lateef	CS01	Computer Science	Dr Khan
317	Ada Lateef	PH01	Physics	Ms Evans
318	Ben Carter	CS01	Computer Science	Dr Khan

The composite primary key is (StudentID, CourseID), since a student may enrol on several courses, and each course may contain several students.

Step 1 – Check 1NF

Each field contains one value, there are no repeating groups, and the composite key uniquely identifies each enrolment. The table therefore satisfies 1NF.

Step 2 – Check 2NF

The table is not in 2NF because some attributes depend on only part of its composite primary key:

- StudentID → StudentName
- CourseID → CourseName, Tutor

These are partial dependencies. StudentName depends only on StudentID, while CourseName and Tutor depend only on CourseID.

To remove these dependencies, split the table into three related tables.

STUDENT

StudentID (PK)	StudentName
317	Ada Lateef
318	Ben Carter

COURSE

CourseID (PK)	CourseName	Tutor
CS01	Computer Science	Dr Khan
PH01	Physics	Ms Evans

ENROLMENT

StudentID (PK, FK)	CourseID (PK, FK)
317	CS01
317	PH01
318	CS01

The two attributes in ENROLMENT form a composite primary key. Each is also a foreign key referencing its corresponding table.

Step 3 – Check 3NF

The three tables are now in 2NF. Assuming each course has one assigned tutor and there are no additional dependencies between non-key attributes, none contains a transitive dependency. They therefore satisfy 3NF.

Trace it: In the original table, Ada Lateef's name appears twice because she takes two courses. Similarly, Computer Science and Dr Khan appear twice because two students are enrolled on the same course. This duplication creates an update anomaly. If the tutor for Computer Science changes, every matching enrolment record must be updated. Missing one would leave inconsistent information. After normalisation, each student and course is stored once. The ENROLMENT table records the relationships between them. Changing the tutor for CS01 requires only one update in COURSE. Notice that normalisation does not remove relationships. It preserves them using primary and foreign keys.

Capturing, selecting, managing and exchanging data

Data can be captured using forms, sensors, scanners, file imports or APIs.

Validation checks whether data meets defined rules, such as an acceptable range or format.

Verification checks whether data has been copied or entered accurately.

Selecting data involves retrieving records or fields that match specified criteria. Managing data includes inserting, updating and deleting records, creating backups, controlling access and enforcing integrity constraints.

Data can be exchanged through files, database connections, APIs and network services. Appropriate formats and protocols should preserve data meaning, consistency and security.

SQL

Structured Query Language (SQL) is used to retrieve and manipulate data in relational databases.

The following examples use two tables: `Student(studentID, name)` and `Result(resultID, studentID, score)`. In `Result`, `studentID` is a foreign key referencing `Student`.

Worked example 2 — selecting and joining records

A college wants to retrieve the names and scores of students who achieved at least 80 marks, ordered from highest to lowest.

Consider the following records.

Student

studentID	name
101	Maya Patel
102	Ben Carter
103	Sofia Khan

Result

resultID	studentID	score
1	101	84
2	102	72
3	103	91
4	101	65

SQL query

```
SELECT Student.name, Result.score
FROM Student
JOIN Result
  ON Student.studentID = Result.studentID
WHERE Result.score >= 80
ORDER BY Result.score DESC;
Query result
name      score
Sofia Khan 91
Maya Patel 84
```

Trace it: `FROM Student` identifies the starting table. `JOIN Result` combines records where the `studentID` values match, linking each result to the correct student.

The `WHERE` condition keeps only records with scores of 80 or above. Ben Carter's score of 72 and Maya Patel's score of 65 are excluded.

`ORDER BY Result.score DESC` sorts the remaining records from highest to lowest score. Finally, `SELECT` determines which columns appear in the output: the student's name and score.

Notice that Maya Patel has two results, but only her qualifying score of 84 appears. The query filters individual result records rather than selecting every result belonging to a student who has scored 80 or more.

Exam language. When tracing a SQL query, identify which tables are joined, how records are matched, which records satisfy the `WHERE` condition, and how the final results are ordered.

Worked example 3 — inserting, updating and deleting records

SQL can also modify the contents of database tables.
The following statements demonstrate three common operations.

Part A – INSERT: adding a record

```
INSERT INTO Student (studentID, name)
VALUES (317, 'Ada Lateef');
```

This adds a new record to the Student table with studentID 317 and the name Ada Lateef, assuming the ID is not already in use.

Part B – UPDATE: modifying a record

```
UPDATE Student
SET name = 'Ada Lateef-Ng'
WHERE studentID = 317;
```

This changes the name of the student whose ID is 317. Other student records remain unchanged.

Part C – DELETE: removing records

```
DELETE FROM Result
WHERE score IS NULL;
```

This deletes every record in the Result table whose score is NULL, subject to any applicable database constraints.

NULL represents a missing or unknown value, not zero. The condition must use IS NULL, rather than = NULL.

Trace it: INSERT INTO creates a new record containing the specified values. The student with ID 317 is added to Student.

UPDATE changes an existing record. The WHERE clause restricts the change to the student whose ID is 317.

Without that condition, the statement would attempt to change the names of every student in the table.

DELETE removes records that satisfy its condition. In this example, every result with a NULL score is removed.

If the condition were omitted, the statement would attempt to delete every record in Result.

These operations affect stored data, unlike SELECT, which retrieves data without modifying the records.

Important: Deleting records with NULL scores may remove assessments that have not yet been marked. A real system should confirm that such records are genuinely unwanted before deleting them.

Exam language. Always check the WHERE clause when tracing UPDATE and DELETE statements. It determines which records are modified or deleted. Without a WHERE clause, an UPDATE affects every row, while a DELETE attempts to remove every row in the table.

Referential integrity

Referential integrity ensures that foreign-key values reference valid records in the related table, unless NULL is permitted.

For example, a studentID stored in Result must identify an existing student in Student.

If someone attempts to delete a student who still has related results, the DBMS follows the referential action defined for that relationship. It may reject the deletion, automatically delete the related results (ON DELETE CASCADE), or set their foreign keys to NULL (ON DELETE SET NULL), if permitted.

This prevents the database from retaining invalid references between related records.

Common misconception. Referential integrity does not mean that related records can never be deleted. It means the database must handle such operations according to the constraints defined for the relationship.

Term	Meaning
Primary key	Attribute(s) that uniquely identify a record.
Candidate key	Any attribute or set of attributes that could uniquely identify a record; one candidate is chosen to be the primary key.
Foreign key	Attribute that references a key in another table.
Secondary key	Non-primary field used/indexed for searching; values need not be unique.
Index	Additional structure that speeds retrieval at the cost of storage and update work.
Entity relationship model	Diagram/model showing entities, attributes/keys and relationships.

Common misconception. A foreign key is not necessarily “the other table’s primary key copied exactly” in every database design; it is a field whose values reference a candidate/primary key in another table.

Transactions, locking and redundancy

ACID	Guarantee
Atomicity	All of a transaction happens or none does.
Consistency	A completed transaction leaves database rules/invariants satisfied.
Isolation	Concurrent transactions do not interfere with each other; each behaves as if it were running on its own.
Durability	Once committed, changes survive a system failure such as a power loss.

Record locking helps prevent conflicting changes when multiple users attempt to update the same data simultaneously, ensuring that transactions do not interfere with one another.

Data redundancy refers to storing multiple copies of the same data. Controlled redundancy can improve availability, reliability and recovery, particularly in backup and replication systems. However, unnecessary duplication within a relational database can lead to inconsistencies and update anomalies. Normalisation aims to minimise this type of redundancy by ensuring that each fact is stored in the appropriate place.

3.4 Networks and the internet

A network connects devices so they can exchange data and share resources. Protocols are agreed rules for communication; standards allow hardware and software from different vendors to interoperate.

The TCP/IP stack: four layers, each with one job

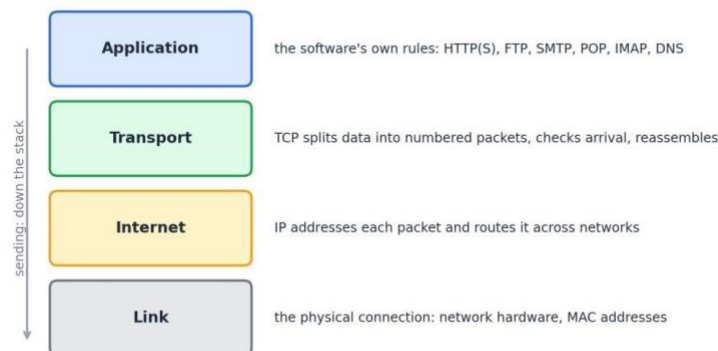


Fig 3.1 — The TCP/IP stack

Protocol layering separates responsibilities. An application can use a transport service without knowing how a particular Wi-Fi radio sends bits. During transmission, layers add control information as needed; the receiver removes this information in reverse order. DNS is a distributed naming system that resolves domain names to IP addresses.

Concept	Key distinction
LAN vs WAN	A Local Area Network (LAN) connects devices within a limited geographical area, such as a building or campus. A Wide Area Network (WAN) connects networks across larger geographical areas, often using telecommunications providers' infrastructure.
Packet switching	Data is divided into packets that are transmitted independently and may follow different routes to their destination. Network links can be shared efficiently.
Circuit switching	A dedicated communication path or channel is established for the duration of a session. This provides predictable resource availability but can be inefficient when the connection is idle.
Client–server	Servers provide resources or services to clients. This allows centralised management, security and administration.
Peer-to-peer	Devices can both request and provide resources directly to one another. This reduces dependence on central servers but can make management and security more difficult.

Network hardware

Device	Role
NIC (Network Interface Card)	Provides the hardware interface that allows a device to connect to a network, typically using a MAC address at the data-link layer.
Switch	Connects devices within a LAN and forwards frames to the appropriate ports using MAC address information.
Router	Connects different networks and forwards IP packets between them using routing information.
Wireless access point	Allows wireless devices to connect to a network using Wi-Fi.

Modem/access equipment	Converts data to and from the signalling format required by a particular access technology, such as DSL or cable, enabling connectivity to an internet service provider (ISP).
-------------------------------	--

Threats and protection

Network threats include malware, unauthorised access, interception, spoofing, denial-of-service attacks and social engineering.

Firewalls filter network traffic according to defined rules. Proxy servers handle requests on behalf of clients and may provide filtering, caching or logging. Encryption protects the confidentiality of transmitted data, while authenticated security protocols can also provide integrity and authentication.

Effective network security relies on multiple layers of protection, including strong authentication, least-privilege access, regular software updates, backups and continuous monitoring.

3.5 Web technologies

HTML (HyperText Markup Language) defines the structure and content of a web page. CSS (Cascading Style Sheets) controls its appearance, while JavaScript adds programmable behaviour, such as responding to user interactions or changing page content.

Client-side processing takes place in the user's browser. It can provide fast responses and reduce the number of requests sent to the server. However, it cannot be relied upon for security because users can inspect or modify code running in their browsers.

Server-side processing is required for authoritative validation, secure database operations and protecting sensitive information, such as API keys and authentication credentials.

Worked example 4 — HTML, CSS and JavaScript working together

The following web page demonstrates how HTML, CSS and JavaScript perform different roles to produce an interactive result.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Results page</title>

  <style>
    .warning {
      font-weight: bold;
    }
  </style>
</head>
<body>
  <h1 id="title">Results</h1>
  <p class="warning">Provisional</p>

  <script>
    document.getElementById("title").textContent =
      "Results are in";
  </script>
</body>
</html>
```

What appears in the browser

The page initially defines a heading containing "Results" and a paragraph containing "Provisional". When the JavaScript executes, it changes the heading's text.

The resulting page displays:

Results are in

Provisional

The second line appears in bold because of the CSS rule.

Trace it: HTML creates the page's structure. The `<h1>` element defines a heading with the identifier `title`, while the `<p>` element defines a paragraph with the class `warning`.

CSS controls the presentation. The selector `.warning` identifies elements belonging to the `warning` class, and `font-weight: bold` makes their text bold. It does not change the text itself.

JavaScript then modifies the page's content. `document.getElementById("title")` locates the heading whose `id` is `title`. The `textContent` property replaces its original text, "Results", with "Results are in".

Notice that the JavaScript runs in the browser after the HTML elements have been created, because the `<script>` appears near the end of the document body. The change happens without requesting a new page from the server.

This demonstrates the three distinct roles: **HTML provides structure, CSS controls presentation, and JavaScript provides behaviour.**

Common misconception. JavaScript does not have to change a page's appearance through CSS. It can also modify HTML content directly through the Document Object Model (DOM). In this example, JavaScript changes the heading's text, while CSS independently controls the paragraph's appearance.

Search engines and PageRank

Search engines discover pages through links and other sources, crawl pages they can access, and build an index. When someone searches, a ranking system orders relevant results.

PageRank is one algorithm that estimates a page's importance from links pointing to it. Modern search engines use many additional ranking signals.

In a simplified PageRank model, an incoming link contributes more if it comes from a page with a high score. However, each linking page divides its contribution among its outgoing links.

$$PR(A) = (1 - d) + d \times [PR(T_1)/L(T_1) + PR(T_2)/L(T_2) + \dots]$$

Where:

$PR(A)$ is the PageRank score of page A.

$T_1, T_2, \dots$ are the pages that link to A.

$L(T)$ is the number of outgoing links from page T.

d is the damping factor, commonly set to 0.85.

For example, a page with ten outgoing links divides its link contribution into ten equal shares.

The damping factor $d = 0.85$ models a person following a link with probability 0.85 and jumping to another page with probability 0.15. The term $(1 - d)$ gives every page a baseline score.

This formula uses a scale on which the final scores add up to the number of pages. Another common version divides the baseline by the number of pages so that the scores add up to 1.

PageRank is calculated iteratively: start with an estimated score for each page, apply the formula repeatedly, and continue until the scores change very little.

In the worked example below, update A, then B, then C, always using the newest score available. Keep full precision in the calculations and round only the displayed answers to three decimal places.

Worked example 5 — one PageRank calculation, three rounds

Three pages link as follows: $A \rightarrow B$ and C ; $B \rightarrow C$; $C \rightarrow A$. Therefore, $L(A) = 2$, $L(B) = 1$ and $L(C) = 1$. Start with $PR(A) = PR(B) = PR(C) = 1$ and use $d = 0.85$.

Round	$PR(A)$	$PR(B)$	$PR(C)$
Start	1.000	1.000	1.000
1	1.000	0.575	1.064
2	1.054	0.598	1.106
3	1.090	0.613	1.135

Trace it: Start with $PR(A) = PR(B) = PR(C) = 1.000$. In each round, calculate A, then B, then C, using the newest scores available.

Round 1

- **Page A:** Only C links to A, and C has one outgoing link. $PR(A) = 0.15 + 0.85 \times (1.000/1) = 1.000$.
- **Page B:** Only A links to B. A has two outgoing links, so $PR(B) = 0.15 + 0.85 \times (1.000/2) = 0.575$.
- **Page C:** Both A and B link to C. Use the newly calculated score for B: $PR(C) = 0.15 + 0.85 \times (1.000/2 + 0.575/1) = 1.06375 \approx 1.064$.

Round 2

Repeat the calculations using the latest available scores.

- **Page A:** $PR(A) = 0.15 + 0.85 \times 1.06375 = 1.054$.
- **Page B:** $PR(B) = 0.15 + 0.85 \times (1.0541875/2) = 0.598$.
- **Page C:** $PR(C) = 0.15 + 0.85 \times (1.0541875/2 + 0.5980296875) = 1.106$.

Round 3

Repeat the same process, again retaining full precision during calculations.

- **Page A:** $PR(A) \approx 1.090$.
- **Page B:** $PR(B) \approx 0.613$.
- **Page C:** $PR(C) \approx 1.135$.

After each round, the ranking remains $C > A > B$, although the scores continue to change. Page C ranks highest because it receives contributions from both A and B. Page B ranks lowest because its only incoming link comes from A, which divides its contribution between two pages.

The calculations would continue until the scores change very little between successive rounds.

Notice: Because this example updates scores one page at a time, the intermediate totals do not necessarily add up to 3. As the calculations converge, the total approaches 3. If all pages were updated simultaneously using only scores from the previous round, the intermediate results would differ, although both methods converge to the same final PageRank scores.

Section 3 exercises

Quick check

1. State one difference between symmetric and asymmetric encryption.

2. What does referential integrity prevent?
3. Name the four ACID properties.
4. Which network device routes packets between networks?

Practice

1. Normalise an order table containing orderID, customerID, customerName, customerPostcode, customerTown, productID, productName and quantity to 3NF.
2. A table PRODUCT(productID, productName, price) stores products. Write SQL to list productName and price for products costing less than 20, ordered by price from highest to lowest.
3. Explain why a browser form must be validated again on the server.
4. Compare client–server and peer-to-peer architectures for a small school file-sharing system.

Exam-style

1. A ticketing system receives two purchases for the final seat almost simultaneously. Explain how transactions, isolation and locking should prevent a double sale. [6]
2. Discuss security controls for a school moving a local database to an internet-accessible service. [9]

Section 3 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. Why is lossless compression essential for program files but often unnecessary for photographs?
2. Why should a password system store have salted password hashes rather than reversibly encrypted passwords?
3. What does referential integrity prevent? If a deletion would break it, give two actions a DBMS might take.
4. Why is client-side validation alone never sufficient for security?

Key facts

Lossless compression reconstructs the original exactly; lossy compression discards information for greater reduction.

Symmetric encryption uses one shared secret; asymmetric uses a public/private pair; hashing produces a one-way digest.

3NF means non-key attributes depend on the key, the whole key, and nothing but the key.

ACID and record locking protect transactions; referential integrity protects relationships.

TCP/IP uses layers; DNS maps names; routers join networks; switches forward inside a LAN.

Client-side code is convenient but not authoritative for security; server-side checks remain necessary.

Section 4 — Data types, data structures and Boolean algebra (spec 1.4)

Before you start — retrieve from GCSE: Convert 45 to binary. What is a Boolean value? What is LIFO?

4.1 Primitive data types and number representation

Type	Typical meaning
Integer	A whole number, positive, negative or zero.
Real / floating point	A number that can represent fractional values, usually with limited precision.
Character	A single character from a character set, such as A, 7 or ?.
String	A sequence of characters, such as "Computer Science".
Boolean	A logical value: True or False.

Positive binary and hexadecimal

Binary is a base-2 number system using only 0 and 1. Each position represents a power of two, starting with 2^0 on the right.

Hexadecimal is base 16, using digits 0–9 and letters A–F. Each hexadecimal digit represents exactly four binary bits. To convert binary to hexadecimal, group the bits into fours from the right, adding leading zeros if necessary.

Worked example 1 — converting between denary, binary and hexadecimal

Convert 173_{10} into binary and hexadecimal, then convert $3F_{16}$ into denary.

Part A – Denary to binary

Find the powers of two that add up to 173.

$$173 = 128 + 32 + 8 + 4 + 1$$

Place value: 128 64 32 16 8 4 2 1

Binary bit: 1 0 1 0 1 1 0 1

$$173_{10} = 10101101_2$$

Part B – Binary to hexadecimal

Group the binary digits into sets of four, starting from the right.

Binary: 1010 1101

Hexadecimal: A D

$$10101101_2 = AD_{16}$$

Part C – Hexadecimal to denary

$$3F_{16} = (3 \times 16) + (15 \times 1)$$

$$= 48 + 15$$

$$= 63_{10}$$

Trace it: In Part A, each binary digit is 1 if its corresponding power of two contributes to 173, otherwise it is 0. In Part B, split the eight bits into two groups: 1010 represents 10 (A in hexadecimal), while 1101 represents 13 (D). In Part C, the hexadecimal digits have place values of 16 and 1. Since F represents 15, the result is $3 \times 16 + 15 = 63$.

Binary integer arithmetic

Binary addition follows the same principle as denary addition but uses only two digits. Working from right to left, carry into the next column whenever the total reaches two.

The key rules are:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 1 = 10_2 \text{ (write 0, carry 1)}$$

$$1 + 1 + 1 = 11_2 \text{ (write 1, carry 1)}$$

With a fixed number of bits, overflow occurs when the mathematical result lies outside the range that can be represented. Subtraction can be performed by adding the two's-complement negative of the second number.

Worked example 2 — binary addition

Add 54 and 29 using 8-bit binary.

```

Carry-in: 01111000
           00110110  (54)
         + 00011101  (29)
         -----
           01010011  (83)
  
```

The carry-in row shows which columns receive a carry from the column immediately to their right. It is not a third number being added.

Trace it: Begin with the rightmost bit, the least significant bit (LSB), and work towards the left. Whenever two 1s are added, write 0 and carry 1 into the next column. If three 1s are added, including a carry, write 1 and carry 1.

For example, in the third column from the right, $1 + 1 = 10_2$, so write 0 and carry 1. In the next column, $0 + 1 + 1$ (the carry) = 10_2 , so write another 0 and carry 1. Continue until all eight columns have been processed. The result is 01010011_2 , which equals $64 + 16 + 2 + 1 = 83$ in denary. This confirms that $54 + 29 = 83$. Since 83 fits within the 8-bit representation, no overflow occurs.

Sign-and-magnitude and two's complement

Sign-and-magnitude represents negative numbers using the most significant bit (MSB) as a sign: 0 for positive and 1 for negative. The remaining bits represent the magnitude. This method has two representations of zero (+0 and -0), making arithmetic less convenient.

Two's complement avoids this problem and allows positive and negative integers to be handled using the same binary addition circuitry.

In an 8-bit two's-complement representation, the bit weights are:

-128 64 32 16 8 4 2 1

This gives a representable range of -128 to +127.

To convert a positive binary number to its negative equivalent in two's complement, invert every bit and add 1.

Worked example 3 — representing a negative number in two's complement

Represent -75 using 8-bit two's complement.

Step 1: Write +75 in binary.

```
01001011
```

Step 2: Invert every bit (0 $\leftrightarrow$ 1).

```
10110100
```

Step 3: Add 1.

```
10110100
+      1
+-----
10110101
```

Result: $10110101_2 = -75_{10}$

Trace it: First, convert positive 75 into 8-bit binary. Since $75 = 64 + 8 + 2 + 1$, its binary representation is 01001011.

Next, invert all eight bits, changing every 0 into 1 and every 1 into 0. This produces 10110100. Adding 1 gives 10110101, the two's-complement representation of -75.

To verify the result, apply the signed bit weights:

```
Bits:      1   0   1   1   0   1   0   1
Weights: -128 64 32 16   8   4   2   1
Value = -128 + 32 + 16 + 4 + 1
       = -75
```

The most significant bit has a weight of -128, not +128. Adding the weights of all the bits set to 1 confirms that the representation is correct.

Floating point

OCR floating-point questions typically use a binary mantissa and exponent, both represented in two's complement. The mantissa contains the significant bits, while the exponent determines the position of the binary point. The binary point is assumed to be immediately after the mantissa's sign bit.

A normalised positive mantissa begins 0.1, while a normalised negative mantissa begins 1.0. Normalisation maximises the precision available for the mantissa width.

Worked example 4 — reading a floating-point value

Mantissa: 01010000
Exponent: 0010

Mantissa = $0.1010000_2 = 0.625$
Exponent = +2

Value = 0.625×2^2
= 0.625×4
= 2.5

Trace it: The mantissa begins 01, so it represents a normalised positive number. Its fractional bits give 0.625. The exponent is +2, so multiply by $\backslash(2^2\backslash)$, equivalent to moving the binary point two places to the right. The resulting binary number is 10.10000_2 , which equals 2.5 in denary.

To add floating-point numbers, first align their exponents by shifting the mantissa of the number with the smaller exponent to the right, increasing its exponent accordingly. Add the mantissas, then normalise the result and adjust the exponent.

Negative mantissas are represented in two's complement. Subtraction can therefore be performed by adding a negative value after aligning the exponents.

Shifting a mantissa may discard low-order bits, introducing rounding or truncation errors. Overflow occurs when a result exceeds the representable range; underflow occurs when a non-zero result is too small in magnitude to be represented.

Precision versus range. More mantissa bits increase precision. More exponent bits increase range. With a fixed total word length, increasing one usually reduces the other.

Bitwise operations and masks

AND can clear unwanted bits, OR can set selected bits, XOR can toggle bits, and shifts move bit patterns. A logical left shift by one multiplies an unsigned integer by two if no significant bit is lost; a right shift divides by two with loss of remainder. Masks select bit fields or set/clear flags.

Worked example 5 — applying bitwise masks

value = 10110110
mask = 00001111

AND = 00000110 # preserve low nibble
OR = 10111111 # set masked bits to 1
XOR = 10110001 # toggle masked bits

Trace it: The mask 00001111 targets the four rightmost bits (the low nibble).

AND clears the four leftmost bits and preserves the low nibble.

OR sets the four rightmost bits to 1.

XOR toggles those four bits, changing each 0 to 1 and each 1 to 0.

Character sets and Unicode encoding

A **character set** maps characters, such as letters, digits, punctuation marks and symbols, to numeric codes that computers can represent in binary.

ASCII (American Standard Code for Information Interchange) is an early character encoding standard that uses 7 bits to represent 128 characters, including English letters, digits, punctuation and control characters. Although widely used in early computer systems, ASCII cannot represent the many writing systems and symbols used worldwide.

Unicode was developed to overcome these limitations. It assigns a unique number, called a *code point*, to characters from writing systems around the world, as well as mathematical symbols, punctuation and emoji.

However, assigning a number to a character is not the same as deciding how that number should be stored. Unicode therefore supports different encoding formats, including UTF-8, UTF-16 and UTF-32.

How did UTF-8, UTF-16 and UTF-32 develop?

The original Unicode standard, introduced in 1991, was designed around 16-bit character values. This initially seemed sufficient to represent the characters needed by most writing systems.

However, 16 bits provide only $2^{16} = 65,536$ possible values, and it became clear that Unicode would need a larger code space.

UTF-8 was developed in 1992 to represent Unicode characters efficiently in byte-oriented systems while maintaining compatibility with ASCII. Unicode subsequently expanded its code space, and UTF-16 was introduced to support additional characters using pairs of 16-bit units when necessary. UTF-32 provides a fixed-width representation using one 32-bit unit per code point.

The three formats are not successive replacements for one another. They are alternative methods of representing the same Unicode code points.

Unicode encoding formats

Encoding	Storage per Unicode code point	Main characteristic
UTF-8	1–4 bytes	Variable-length encoding; particularly efficient for ordinary English text and compatible with ASCII.
UTF-16	2 or 4 bytes	Uses one or two 16-bit units; offers a balance between storage efficiency and text processing in some systems.
UTF-32	4 bytes	Fixed-length encoding; simplifies access to individual code points but often uses more memory.

Worked example 6— comparing UTF-8, UTF-16 and UTF-32

Consider storing the word `dog`.

Each letter belongs to the ASCII character set and therefore requires only one byte in UTF-8.

Character	UTF-8	UTF-16	UTF-32
d	1 byte	2 bytes	4 bytes
o	1 byte	2 bytes	4 bytes
g	1 byte	2 bytes	4 bytes
Total	3 bytes	6 bytes	12 bytes

Now consider the word `café`, where the final character is the single Unicode character `é`.

Encoding	Calculation	Total
UTF-8	1 + 1 + 1 + 2	5 bytes
UTF-16	2 + 2 + 2 + 2	8 bytes
UTF-32	4 + 4 + 4 + 4	16 bytes

In UTF-8, the character `é` requires two bytes rather than one because its Unicode code point lies outside the ASCII range.

Finally, consider the emoji 🐱.

Encoding	Storage
UTF-8	4 bytes
UTF-16	4 bytes
UTF-32	4 bytes

This emoji requires four bytes in all three encodings, although those bytes represent the code point differently.

Trace it: The word `dog` requires only three bytes in UTF-8 because each letter can be represented using one byte. UTF-16 requires six bytes because each letter occupies one 16-bit unit, while UTF-32 requires twelve bytes because every code point occupies 32 bits.

For `café`, UTF-8 requires an additional byte for `é`, giving a total of five bytes. UTF-16 still requires two bytes per character, while UTF-32 uses four bytes per character.

The emoji example shows that UTF-8 is not always smaller. Its storage requirements depend on the Unicode code points being represented.

Common misconception. UTF-32 is not a newer or more advanced version of UTF-8, and UTF-16 is not simply 16-bit ASCII. All three formats can represent the full Unicode character repertoire. They differ in how the code points are encoded and how much storage they require. Likewise, UTF-8 does not allocate fewer bytes simply because a word is short. For ordinary English text, each ASCII character requires one byte, whether the word is `dog` or `computer`. The number of bytes depends on the characters present, not the length of the word itself.

Exam language. When calculating text storage, identify the encoding, determine how many bytes each character requires, and add the values. Do not assume that UTF-8 always uses one byte per character or that UTF-16 always uses exactly two bytes.

A character set maps symbols to numeric codes. ASCII is a small historical set used heavily in early systems and protocols. Unicode provides code points for writing systems and symbols worldwide; encodings such as UTF-8 store those code points as bytes. Do not say Unicode is simply “16-bit ASCII”.

4.2 Data structures

Structure	Shape / operations	When useful
Array	Fixed-size indexed collection; may be 1D, 2D or 3D.	Fast indexed access and predictable layout.
Record	Named fields, possibly different types.	Represent one structured entity.
List	Ordered collection with flexible length.	General sequence where insert/remove is required.
Tuple	Ordered immutable collection.	Fixed grouped values that should not change.
Linked list	Nodes contain data and link(s) to other nodes.	Insert/delete without shifting an array; sequential access.
Stack	LIFO: push, pop, peek.	Call stack, undo, depth-first processing.
Queue	FIFO: enqueue, dequeue.	Scheduling, buffering, breadth-first processing.
Tree / BST	Hierarchical nodes; BST orders smaller left/larger right.	Hierarchies, ordered search when balanced.
Graph	Vertices connected by directed/undirected, weighted/unweighted edges.	Networks, routes, dependencies.
Hash table	Hash maps key to index/bucket.	Expected $O(1)$ lookup with a good hash and load factor.

Operations on structures

OCR expects more than definitions: you should be able to create, traverse, add and remove data. For array-backed stacks/queues, manage top/front/rear indices and detect underflow/overflow. For linked structures, update links in the correct order so nodes are not lost. For a BST, insertion follows comparisons until an empty child is reached.

A linked list: each node stores data and a pointer to the next node



After inserting 30 between 12 and 47:

set 30.next to 47, then set 12.next to 30.

Only pointers change; the existing data values do not move.

Fig 4.1 — A linked list, and an insertion done purely with pointers

Stack (LIFO) and queue (FIFO)

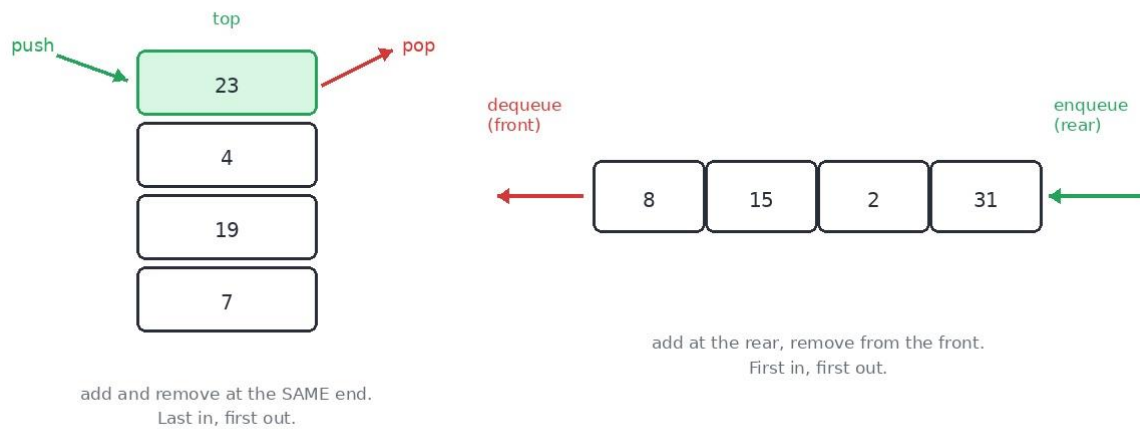
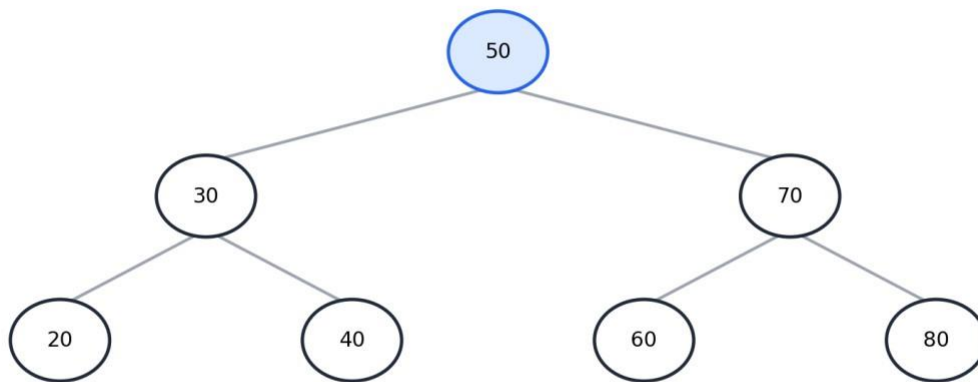


Fig 4.2 — The stack and the queue

A binary search tree built by inserting 50, 30, 70, 20, 40, 60, 80



the rule: smaller values go left, larger go right, at every node.
An in-order traversal (left, node, right) visits: 20 30 40 50 60 70 80 – sorted.

Fig 4.3 — A binary search tree

A hash table: the hash function turns a key into an index

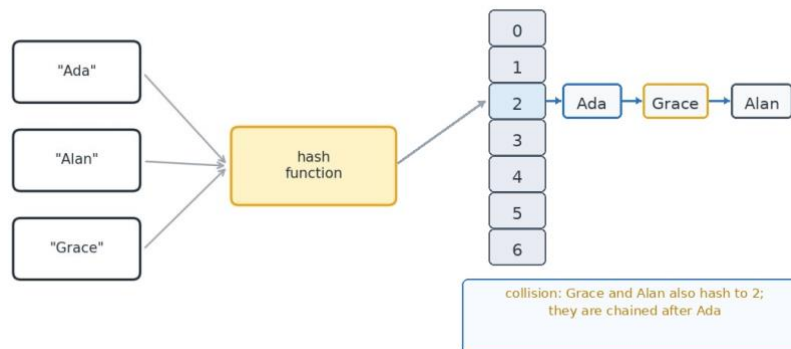


Fig 4.4 — A hash table, including a collision handled by chaining

Worked example 7 — Array-backed stack

A stack follows the **Last In, First Out (LIFO)** principle: the last item added is the first item removed. The following pseudocode implements a stack using an array and two operations:

- **push(item)** adds an item to the top of the stack.
- **pop()** removes and returns the item at the top of the stack.

The variable `top` stores the index of the current top item. Initially, `top = -1`, indicating that the stack is empty. `MAX` represents the array's capacity.

```
procedure push(item)
    if top == MAX - 1 then
        print("overflow")
    else
        top = top + 1
        stack[top] = item
    endif
endprocedure

function pop()
    if top == -1 then
        return null
    endif
    item = stack[top]
    top = top - 1
    return item
endfunction
```

Tracing the stack

Suppose the stack has a maximum capacity of three items (`MAX = 3`), with valid array indices 0, 1 and 2. Initially, `top = -1` and the stack is empty.

Operation	Stack contents (bottom → top)	top after	Result
Initial state	Empty	-1	—
push(12)	[12]	0	12 added
push(25)	[12, 25]	1	25 added
push(37)	[12, 25, 37]	2	37 added
push(40)	[12, 25, 37]	2	Overflow
pop()	[12, 25]	1	Returns 37
pop()	[12]	0	Returns 25
pop()	Empty	-1	Returns 12
pop()	Empty	-1	Returns null

Trace it: Initially, `top = -1`, indicating that the stack contains no items.

When `push(12)` executes, the procedure first checks whether the stack is full. Since it is not, `top` increases from `-1` to `0`, and 12 is stored at `stack[0]`.

The next two calls, `push(25)` and `push(37)`, increase `top` to 1 and 2 respectively. The stack now contains three items and is full.

When `push(40)` is attempted, the condition `top == MAX - 1` is true because `top = 2` and `MAX - 1 = 2`. The procedure reports **overflow**, and 40 is not added.

The first `pop()` retrieves the item at `stack[2]`, which is 37. The value of `top` then decreases from 2 to 1.

Subsequent calls remove 25 and 12 in that order, demonstrating the LIFO principle.

After removing 12, `top` becomes `-1`, indicating that the stack is empty. A further call to `pop()` returns null, signalling underflow because there is no item available to remove.

Notice that `pop()` does not physically erase the removed value from the array. Instead, it decreases `top`, so the removed item is no longer considered part of the stack. A later `push()` may overwrite that array position.

Common misconception. Overflow occurs when an item is added to a full stack, while underflow occurs when an attempt is made to remove an item from an empty stack. These are different error conditions.

Exam language. When tracing an array-backed stack, keep track of top after every operation. Remember that push() increments top before storing the item, while pop() retrieves the item before decrementing top. Always check for overflow or underflow before performing the operation.

Worked example 8 — Circular queue — rear wraps around

A circular queue follows the **First In, First Out (FIFO)** principle: the first item added is the first item removed.

Unlike a linear array-backed queue, a circular queue can reuse available positions at the beginning of the array when the rear reaches the end.

The following pseudocode adds an item to a circular queue:

```
procedure enqueue(item)
    if count == MAX then
        print("full")
    else
        queue[rear] = item
        rear = (rear + 1) MOD MAX
        count = count + 1
    endif
endprocedure
```

The queue uses three variables:

- **rear** stores the index of the next available position for insertion.
- **count** records how many items are currently stored.
- **MAX** specifies the maximum number of items the queue can hold.

The expression $(\text{rear} + 1) \text{ MOD } \text{MAX}$ makes the queue circular. When rear reaches the final array position, the next increment wraps it back to index 0.

Tracing the circular queue

Suppose $\text{MAX} = 4$, giving valid array indices 0, 1, 2 and 3.

The queue initially contains A and B at indices 1 and 2, following an earlier removal that freed index 0. The next insertion position is $\text{rear} = 3$, and $\text{count} = 2$.

Operation	Index 0	Index 1	Index 2	Index 3	rear after	count after
Initial state	Empty	A	B	Empty	3	2
enqueue(C)	Empty	A	B	C	0	3
enqueue(D)	D	A	B	C	1	4
enqueue(E)	D	A	B	C	1	4

The final operation fails because the queue is full.

Trace it: Initially, $\text{rear} = 3$ and $\text{count} = 2$, so the queue has space for two more items.

When enqueue(C) executes, the condition $\text{count} == \text{MAX}$ is false because 2 is less than 4. The value C is stored at $\text{queue}[3]$.

The next instruction updates rear:

$\text{rear} = (3 + 1) \text{ MOD } 4 = 0$

The remainder is 0, so rear wraps around to the beginning of the array. The value of count then increases to 3.

When enqueue(D) executes, D is stored at $\text{queue}[0]$. The rear index becomes $(0 + 1) \text{ MOD } 4 = 1$, and count increases to 4.

Finally, enqueue(E) checks whether the queue is full. Since $\text{count} == \text{MAX}$ is now true, the procedure prints "full". No item is added, and neither rear nor count changes.

Although the items occupy array positions 1, 2, 3 and 0, their logical order remains $A \rightarrow B \rightarrow C \rightarrow D$. The next item removed would be A, preserving the FIFO principle.

Common misconception. The rear index does not necessarily identify the last occupied position. In this implementation, it identifies the position where the next item would be inserted. The variable count determines whether the queue is full, even when rear has wrapped around.

Exam language. When tracing a circular queue, track rear and count after every insertion. Remember that MOD returns the remainder after division, allowing the index to wrap from the final array position back to 0. A full queue must reject further insertions until an item is removed.

Array	Direct indexed access; traverse by index.	Insert/delete in the middle usually shifts later elements.
Record	Access a named field.	Fields are defined by the record type; update field values directly.
List	Traverse in sequence or by supported index.	Append/insert/remove according to the list implementation.
Tuple	Read values in order/by index.	Immutable: create a new tuple rather than changing membership.
Linked list	Follow links from the head node.	Insert/delete by rewiring links; finding a position may require traversal.
Graph	Traverse neighbours using adjacency representation.	Add/remove vertices and edges, updating the matrix/list consistently.
Stack / queue	Access only the permitted end(s).	push/pop or enqueue/dequeue.
Tree / BST	Depth/breadth traversals; BST search follows comparisons.	BST insert follows left/right comparisons; deletion handles leaf, one-child and two-child cases.
Hash table	Hash key to bucket/slot, then resolve collisions.	Insert/remove at the hashed location, maintaining chains/probe rules.

Common misconception. A list in a programming language is not automatically the same thing as the abstract linked-list data structure. Always use the structure defined by the question.

An adjacency matrix uses $O(V^2)$ space and gives fast edge lookup. An adjacency list stores neighbours for each vertex and is compact for sparse graphs. A hash collision occurs when different keys map to the same index; chaining stores multiple entries in a bucket, while open addressing/probing searches for another slot.

4.3 Boolean algebra, truth tables and circuits

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

NOT inverts a value. Boolean expressions can describe conditions in programs and logic circuits. A truth table lists every possible input combination and the resulting output; a logic-gate diagram is a circuit representation of the same expression.

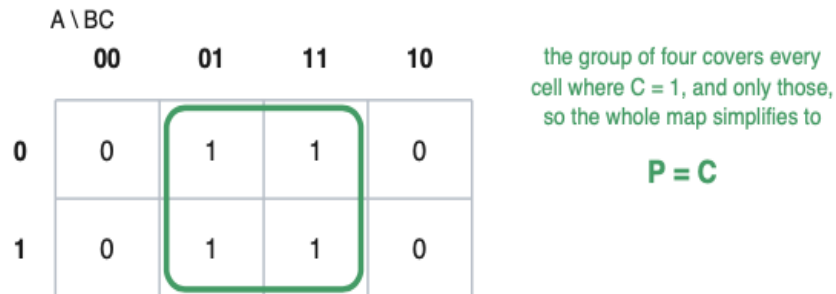
Law	Example
Commutation	$A \text{ AND } B = B \text{ AND } A$; $A \text{ OR } B = B \text{ OR } A$
Association	$(A \text{ AND } B) \text{ AND } C = A \text{ AND } (B \text{ AND } C)$
Distribution	$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$
Double negation	$\text{NOT}(\text{NOT } A) = A$
De Morgan	$\text{NOT}(A \text{ AND } B) = \text{NOT } A \text{ OR } \text{NOT } B$; $\text{NOT}(A \text{ OR } B) = \text{NOT } A \text{ AND } \text{NOT } B$

You should also be able to move between an expression, a truth table and a logic-gate diagram. For example, $P = (A \text{ AND } B) \text{ OR } (\text{NOT } C)$ is drawn with A and B entering an AND gate, C entering a NOT gate, and those two outputs entering an OR gate.

Karnaugh maps

A Karnaugh map arranges truth-table outputs in Gray-code order so adjacent cells differ by one variable. Group adjacent 1s in rectangles whose size is a power of two. Make groups as large as possible; wrapping and overlapping are allowed. A variable that changes inside a group disappears from that term; OR the remaining terms together.

A Karnaugh map groups 1s to simplify a Boolean expression



rule: groups must be rectangles of 1, 2, 4 or 8 ones; columns follow Gray code order (00 01 11 10)

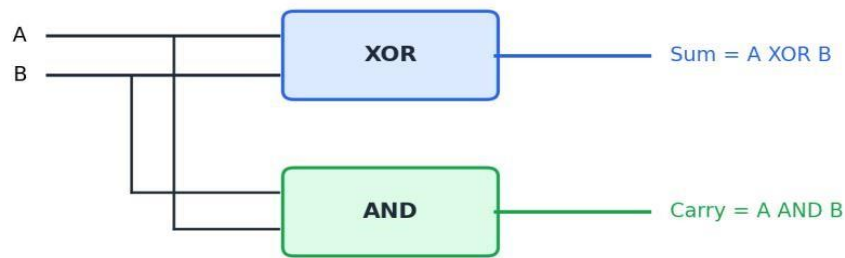
Fig 4.5 — A Karnaugh map simplifying a three-variable expression to $P = C$

Common misconception. Gray code is the arrangement principle used for Karnaugh-map rows and columns. Adjacent Gray-code values differ in exactly one bit; ordinary binary numerical order does not guarantee this at every boundary.

D-type flip-flops and adders

A D-type flip-flop stores one bit. On its active clock edge, Q takes the value at D and then holds it until the next active edge. Registers are built from many flip-flops. A half adder adds two bits: $\text{Sum} = A \text{ XOR } B$ and $\text{Carry} = A \text{ AND } B$. A full adder accepts A, B and carry-in, and produces a sum plus carry-out; chaining full adders creates multi-bit adders.

The half adder: XOR gives the sum, AND gives the carry



1 + 1 in binary is 10: sum 0, carry 1. Exactly what these two gates produce.
Two half adders plus an OR gate make a full adder, which also accepts a carry in.

Fig 4.6 — The half adder

Section 4 exercises

Quick check

1. Convert 0x7B to denary and binary.
2. State the 8-bit two's-complement range.
3. Which traversal of a BST produces values in sorted order?
4. What are the sum and carry outputs of a half adder for A=1, B=1?

Practice

1. Add 101101_2 and 011011_2 and state whether 6-bit unsigned overflow occurs.
2. Represent -37 in 8-bit two's complement.
3. The floating-point value has mantissa 00101100 and exponent 0011, both in two's complement. Normalise the mantissa and give the adjusted exponent.
4. Describe how to insert a node after a named node in a singly linked list without losing the rest of the list.

Exam-style

1. A road network has millions of junctions, but each junction connects to only a few roads. Justify an adjacency-list representation rather than an adjacency matrix. [5]
2. For $F(A,B,C)$, the output is 1 for input combinations 001, 011, 110 and 111, and 0 for the other four combinations. Simplify F using a Karnaugh map and verify the simplified expression against the truth table. [6]

Section 4 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. Explain why the MSB in 8-bit two's complement has weight -128 , and state the representable range.
2. With a fixed word length, why does adding exponent bits reduce precision?
3. When would you choose a hash table over a balanced BST, and why is hash lookup expected $O(1)$ rather than guaranteed $O(1)$?
4. Why must Karnaugh-map columns follow Gray-code order?
5. A half adder receives $A = 1$ and $B = 1$. State the sum and carry, and name the gates that produce them.

Key facts

- Know integer, real/floating point, character, string and Boolean data types.
- Hexadecimal groups binary into nibbles; two's complement uses a negative MSB weight.
- Floating point trades precision (mantissa) against range (exponent); arithmetic can introduce rounding and overflow/underflow.
- Stacks are LIFO; queues FIFO; linked lists trade random access for efficient pointer-based insertion/deletion.
- Balanced BST search is $O(\log n)$, but a badly skewed tree can degrade to $O(n)$. Hash tables are expected $O(1)$, not guaranteed $O(1)$.
- K-maps use Gray-code adjacency; half adders use XOR for sum and AND for carry.

Section 5 — Legal, moral, cultural and ethical issues (spec 1.5)

Before you start — retrieve: What is personal data? What does copyright protect? Give one possible risk of collecting large amounts of personal information.

This section rewards applied discussion. Learn what each law is for but also practise weighing consequences for different stakeholders.

5.1 Computing-related legislation

Law named by OCR	What students should understand
Data Protection Act 1998 (specification wording)	Purpose of data-protection rules and limits on how organisations collect, store, secure, retain and use personal data. In current UK law, use DPA 2018/UK GDPR principles.
Computer Misuse Act 1990	Unauthorised access, access with intent to commit further offences, and unauthorised acts intended to impair operation/data.
Copyright, Designs and Patents Act 1988	Protection of software and other creative works; copying/distribution/licensing without permission.
Regulation of Investigatory Powers Act 2000	Specification context for interception/monitoring of communications and powers/obligations around surveillance. Current UK surveillance law has also developed beyond RIPA.

Exam note — data protection legislation. The OCR H446 specification (v2.7) refers to the Data Protection Act 1998. However, current UK data protection law is based on the Data Protection Act 2018 and UK GDPR, as amended by the Data (Use and Access) Act 2025. When answering examination questions, distinguish between the legislation named in the specification and the current legal framework, applying the relevant data protection principles to the scenario.

5.2 Moral, ethical, social and cultural issues

Issue	Opportunities / benefits	Risks / questions
Computers in the workforce	Automation can improve safety, productivity and create new skilled roles.	Displacement, deskilling, unequal transition costs and dependence on automated systems.
Automated decision making	Consistency and ability to process large datasets quickly.	Bias in data, opacity, appeal/accountability and inappropriate proxies.
Artificial intelligence	New capabilities in medicine, accessibility, productivity and science.	Misuse, concentration of power, errors, labour effects, environmental cost and accountability.
Environment	Optimisation, smart grids, remote work and efficient logistics.	Energy use, data centres, device manufacture, mining and e-waste.
Censorship and the internet	Protection from illegal/harmful material and platform safety.	Freedom of expression, over-blocking and who controls the rules.
Monitoring behaviour / personal information	Fraud detection, safeguarding and personalised services.	Privacy, chilling effects, profiling, security breaches and function creep.
Piracy / offensive communications	Enforcement protects creators and users from harm.	Enforcement can be intrusive or overbroad and can affect legitimate speech/use.
Layout, colour, character sets	Inclusive localisation reaches more users.	Assuming one language, reading direction, name format or colour meaning can exclude users.

5.3 Writing an extended response

1. Identify the stakeholders and the exact technology in the scenario.
2. Develop benefits with technical detail and a consequence.
3. Develop risks with technical detail and a consequence.
4. Use law where it applies; do not name laws at random.
5. Compare the weight of the arguments in this scenario.
6. Finish with a justified conclusion, not “there are pros and cons”.

Exam language. A 9-mark answer is not a list. Develop a chain: point → explanation → impact on a stakeholder → counterargument/condition → judgement.

Section 5 exercises

Quick check

1. Which Act criminalises unauthorised access?
2. Give one data-protection principle relevant to a school holding student records.
3. Give one cultural consideration when choosing interface colours.

Practice

1. A company uses historical recruitment data to train an automated screening model. Explain two ethical risks and one mitigation.
2. Explain one benefit and one environmental cost of cloud computing.

Exam-style

1. A city proposes live facial recognition in public spaces. Discuss legal, ethical and social issues. [9]
2. A school plans to monitor all student internet traffic and retain it for five years. Discuss the benefits, risks and safeguards. [9]

Section 5 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. Which Act criminalises unauthorised access to computer systems?
2. Give one computing example that may be legal yet ethically questionable, and one that is illegal.
3. Name the four laws named in the H446 specification and give a one-line purpose for each.
4. In a 9-mark answer, when should you name a law and when should you avoid doing so?

Key facts

- Know the purpose of the four laws named in the H446 specification, while applying current data-protection principles.
- Ethics questions are scenario questions: identify stakeholders, benefits, harms, uncertainty and safeguards.
- Automated systems can be consistent yet still reproduce bias from data or design choices.
- Cultural design includes language, reading direction, names, symbols, colour conventions and character sets.
- A high-level discussion weighs evidence and conditions before reaching a justified conclusion.

Common misconception. Legal does not mean ethical, and ethical does not mean legal. A strong answer can distinguish compliance from wider consequences.

Section 6 — Elements of computational thinking (spec 2.1)

Before you start — retrieve: What is decomposition? What is an algorithm? Give one reason programmers reuse existing components or libraries.

Computational thinking is not “thinking like a computer”. It is a set of habits for representing problems and designing solutions that a computer can execute.

6.1 Thinking abstractly

Abstraction removes irrelevant detail while preserving what matters for the purpose. A model is useful because it is simpler than reality, but every abstraction loses information. You should be able to say what has been included, what has been omitted and why the omission is safe for the task.

Common misconception. Abstraction removes irrelevant detail for a purpose; decomposition splits a problem into smaller parts. They often work together, but they are not the same idea.

6.2 Thinking ahead

Thinking ahead identifies inputs, outputs and preconditions before implementation. A precondition must be true before a component is used: binary search, for example, requires data ordered according to the search key. Caching stores data likely to be reused in a faster location; the benefit is reduced access time, while drawbacks include stale copies, memory use and cache-management complexity. Reusable components reduce development time and improve reliability when interfaces are clear.

6.3 Thinking procedurally

Break the problem into components, identify the parts of the solution, place steps in the required order and identify sub-procedures. This turns a vague requirement into a structure that can be implemented and tested.

6.4 Thinking logically

Identify decision points, express the conditions precisely and determine the path followed for each outcome. Logic errors often come from missing combinations or using conditions that overlap unexpectedly.

6.5 Thinking concurrently

Identify parts that can happen at the same time. Concurrency can improve responsiveness or throughput, but shared data may require synchronisation and coordination. Some tasks are inherently sequential because a later step depends on the earlier result.

Scenario clue	Thinking element
Throwing away irrelevant map detail	Abstractly
Inputs, outputs, preconditions, caching, reuse	Ahead
Breaking into ordered sub-problems	Procedurally
Conditions and decision paths	Logically
Independent tasks at the same time	Concurrently

Section 6 exercises

Quick check

1. Define abstraction and explain why deliberately omitting detail can make a model more useful.
2. Give one benefit and one drawback of caching.
3. Which thinking element identifies decision points?

Practice

1. A route-planning program ignores building colour and roof material. Explain why this is abstraction rather than missing data.
2. A video editor applies the same filter to thousands of independent frames. Explain the benefit and one risk of concurrent processing.

Exam-style

1. A hospital appointment system has multiple interacting requirements. Show how all five computational-thinking elements could guide its design. [9]

Section 6 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. A route planner includes journey times but omits scenery. Why is this abstraction rather than missing data?
2. Define precondition and give one example other than “the data must be sorted”.
3. Give one example of a task that is inherently sequential and explain why.
4. Which thinking element covers identifying inputs, outputs, preconditions and reusable components?

Key facts

- Abstraction deliberately removes detail for a purpose.
- Thinking ahead identifies inputs, outputs, preconditions, caching opportunities and reusable components.
- Thinking procedurally decomposes and orders the solution.
- Thinking logically identifies decisions and exact conditions.
- Thinking concurrently finds independent work and recognises synchronisation/overhead.

Section 7 — Problem solving and programming (spec 2.2)

Before you start — retrieve: What is sequence, selection and iteration? What is a subroutine? What is meant by the scope of a variable?

7.1 Core programming constructs

Sequence executes instructions in order. Selection chooses a path based on a condition. Iteration repeats a block: count-controlled loops repeat a known number of times; condition-controlled loops repeat while/until a condition changes.

Worked example 1 — sequence + selection + iteration

```
total = 0
for i = 0 to values.length - 1
    if values[i] > 0 then
        total = total + values[i]
    endif
next i
print(total)
```

Suppose values = [4, -2, 7, 0].

i	values[i]	Positive?	total after iteration
0	4	Yes	4
1	-2	No	4
2	7	Yes	11
3	0	No	11

Trace it: First, total is set to 0 (sequence). The for loop visits each array element (iteration). The if statement adds an element only when it is greater than zero (selection). The values 4 and 7 are included; -2 and 0 are skipped. The program outputs 11.

7.2 Recursion

A recursive subroutine calls itself on a smaller/simpler version of the problem. A correct recursive design has a base case that returns without another self-call and a recursive case that moves towards that base case.

Worked example 2 — recursive factorial

For a non-negative integer n , $\text{factorial}(n)$ is the product of all positive integers from 1 to n . By definition, $\text{factorial}(0) = 1$.

```
function factorial(n)
  if n <= 1 then
    return 1
  else
    return n * factorial(n - 1)
  endif
endfunction
```

The example assumes n is a non-negative integer. Negative inputs are outside the definition of factorial and would need to be rejected in a more robust implementation.

Suppose we call $\text{factorial}(3)$. Follow the calls downwards first, then the returns back up. Multiplication takes place as the recursive calls return, after the base case is reached.

Call	n	Is $n \leq 1$?	What it is waiting for	What it returns
$\text{factorial}(3)$	3	No	$3 \times \text{factorial}(2)$	6
$\text{factorial}(2)$	2	No	$2 \times \text{factorial}(1)$	2
$\text{factorial}(1)$	1	Yes	Nothing – base case	1

Trace it: $\text{factorial}(3)$ cannot finish until $\text{factorial}(2)$ returns. In turn, $\text{factorial}(2)$ waits for $\text{factorial}(1)$. The call $\text{factorial}(1)$ reaches the base case and returns 1 immediately. The pending calculations then complete in reverse order: $2 \times 1 = 2$, followed by $3 \times 2 = 6$.

At the deepest point, three stack frames exist: $\text{factorial}(3)$ and $\text{factorial}(2)$ are suspended, while $\text{factorial}(1)$ executes. Each frame holds information such as the parameter n and the return location. As calls return, their stack frames are removed.

Deep recursion can exhaust the call stack. Some recursive algorithms can be rewritten iteratively; an iterative factorial uses constant extra memory, but other problems (such as tree traversal or backtracking) may require an explicit stack. Recursion is particularly natural for trees, divide-and-conquer and backtracking.

7.3 Variables, scope, modularity and parameter passing

A local variable exists within a subroutine/block and reduces unintended interactions. A global variable is accessible more widely and can make dependencies harder to trace. Modularity splits a solution into cohesive functions/procedures with clear interfaces, improving reuse, testing and maintenance.

Pass by value gives a subroutine a copy of the argument value, so assigning to the parameter does not alter the caller's variable. Pass by reference gives access to the caller's storage, so changes can be visible outside the subroutine. Languages differ in their exact parameter semantics; answer using the model stated by the question.

Common misconception. Pass by reference is not the same as using a global variable. A by-reference parameter allows a subroutine to change the caller's argument while keeping the interface explicit.

7.4 IDEs and debugging

Tool	Use
Breakpoint	Pause execution at a chosen line.
Step into / step over	Execute in controlled steps to follow program flow.
Watch / variable inspector	Observe values as they change.

Call stack	See active subroutine calls.
Syntax/error highlighting	Identify likely syntax/type problems early.

7.5 Object-oriented techniques

Object-oriented programming (OOP) organises a system into interacting objects, each combining data (attributes) with behaviour (methods).

Three important OOP principles are:

Encapsulation: Protects an object's internal state by controlling access through its methods.

Inheritance: Allows a subclass to inherit and extend the attributes and methods of a parent class, representing an *is-a* relationship.

Polymorphism: Allows the same method call to produce different behaviour depending on the type of object receiving it.

OOP is particularly useful when a problem involves entities with their own state and related operations. However, it is not always the most appropriate programming approach; the choice depends on the problem being solved.

Worked example 3 — inheritance and polymorphism

The following OCR-style pseudocode defines an Animal class and a Dog subclass. The subclass inherits behaviour from Animal but overrides the speak() method to provide its own implementation.

```
class Animal
    private name

    public procedure new(givenName)
        name = givenName
    endprocedure

    public function getName()
        return name
    endfunction

    public procedure speak()
        print("...")
    endprocedure
endclass

class Dog inherits Animal
    public procedure new(givenName)
        super.new(givenName)
    endprocedure

    public procedure speak()
        print(getName() + " says Woof")
    endprocedure
endclass
```

Creating objects and calling their methods

Consider the following statements:

```
pet = new Animal("Milo")
pet.speak()
```

```
pet = new Dog("Rex")
pet.speak()
```

Output

```
...
Rex says Woof
```

Tracing the program

Statement	What happens	Output
pet = new Animal("Milo")	Creates an Animal object with the name Milo.	None
pet.speak()	Calls the speak() method defined in Animal.	...
pet = new Dog("Rex")	Creates a Dog object. Its constructor calls the parent constructor to initialise the name.	None
pet.speak()	Calls the overridden speak() method defined in Dog.	Rex says Woof

Trace it: The first statement creates an Animal object named Milo. Its constructor stores the name in the private attribute name. When pet.speak() executes, the method defined in Animal prints "...".

The next statement creates a Dog object named Rex. Because Dog inherits from Animal, it can reuse methods provided by the parent class. The instruction super.new(givenName) calls the parent constructor, initialising the inherited name attribute.

When pet.speak() executes again, the program uses the speak() method defined in Dog, rather than the version inherited from Animal. This is **method overriding**: the subclass provides a different implementation of a method defined in its parent class.

The Dog implementation calls getName(), which it inherits from Animal. This retrieves the private name without accessing the attribute directly, demonstrating encapsulation.

Why this demonstrates polymorphism: The same method call, pet.speak(), produces different results depending on whether pet refers to an Animal or a Dog object. The appropriate method implementation is selected according to the object's actual class.

Common misconception. Inheritance and polymorphism are related but are not the same concept. Inheritance allows a subclass to reuse and extend behaviour from a parent class. Polymorphism allows the same operation to behave differently depending on the object receiving it.

7.6 Computational methods

What makes a problem suitable for a computational method?

A problem is a good candidate when its relevant information can be represented as data, the required operations/constraints can be stated precisely, inputs and outputs can be identified, and a finite procedure can make useful progress with available time and memory. Some problems may be computable in principle but impractical at realistic sizes, which is why efficiency and heuristics matter.

Method	Idea / example
Problem recognition	Identify that a problem can be represented and solved computationally.
Decomposition	Break it into smaller solvable parts.
Divide and conquer	Split into similar subproblems, solve them, combine results; merge sort is a classic example.
Abstraction	Ignore irrelevant detail to produce a useful model.
Backtracking	Build a candidate solution and undo choices that lead to failure; puzzles and constraint search.
Data mining	Find patterns/relationships in large datasets.
Heuristics	Use a rule/estimate to guide search or find a good solution faster.
Performance modelling	Predict system behaviour under expected loads before deploying the real system.
Pipelining	Overlap stages of work to improve throughput.
Visualisation	Represent data graphically so patterns are easier for humans to perceive.

Section 7 exercises

Quick check

1. State the difference between a function and a procedure in typical pseudocode.
2. What two parts must a recursive solution have?
3. What does a breakpoint do?

Practice

1. Trace factorial(5), showing the sequence of calls and returns.
2. Give a scenario where pass by reference is useful and explain one risk.
3. Explain why backtracking suits a Sudoku solver.

Exam-style

1. Compare iterative and recursive solutions for traversing a tree, referring to clarity and memory. [6]
2. A simulation models thousands of interacting vehicles. Explain where OOP helps and where concurrency creates risks. [9]

Section 7 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. What two components must every recursive subroutine have, and what goes wrong without a base case?
2. Explain the difference between pass by value and pass by reference, with one risk of each.
3. Name three IDE debugging tools and state what each helps you inspect.
4. Give one situation where OOP fits the problem well and one where it does not.

Key facts

- Sequence, selection and iteration are the core control constructs.
- Recursion needs a base case and progress toward it; calls consume stack space.
- Prefer clear local scope and modular interfaces; know pass-by-value versus pass-by-reference.
- IDEs support breakpoints, stepping, watches and stack inspection.
- OOP: class/object, attributes/methods, encapsulation, inheritance and polymorphism.
- Computational methods include divide and conquer, backtracking, data mining, heuristics, performance modelling, pipelining and visualisation.

Section 8 — Algorithms (spec 2.3)

Before you start — retrieve: What is linear search? Why does binary search require sorted data? What do LIFO and FIFO mean?

8.1 Designing and analysing algorithms

An algorithm is a finite, unambiguous method for solving a problem. Good design starts from the requirements and data: correctness comes first, then time, space, simplicity and maintainability. The “best” algorithm depends on input size, ordering, memory limits and operations required.

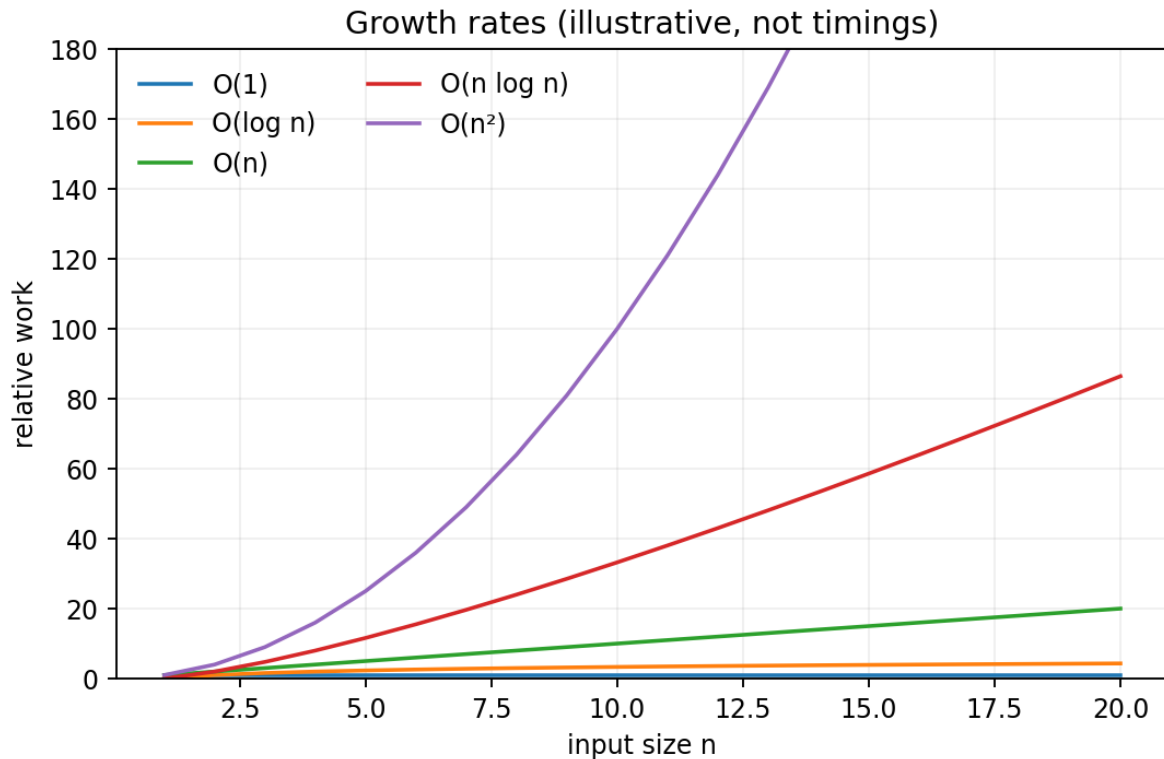


Fig 8.1 — How common growth rates separate as n increases

Class	Meaning	Example
$O(1)$	Constant growth.	Array index access; average hash lookup.
$O(\log n)$	Problem shrinks by a constant factor each step.	Binary search.
$O(n)$	Work proportional to input size.	Linear search.
$O(n \log n)$	Linearithmic growth.	Merge sort; average quicksort.
$O(n^2)$	Quadratic polynomial growth.	Simple bubble/insertion worst-case.
$O(n^k)$	Polynomial for a fixed k .	Many nested-loop algorithms.
$O(2^n)$	Exponential growth.	Brute-force subset-style search.

Big O describes growth, not exact seconds. Constants and hardware matter for small inputs, but growth dominates as n becomes large. Space complexity applies the same idea to additional memory. Algorithms often trade time for space.

Common misconception. “Polynomial” is a family such as $O(n^2)$, $O(n^3)$, etc. $O(n^2)$ is polynomial, but polynomial does not mean only quadratic.

8.2 Algorithms for data structures

Stacks and queues can support insertion and removal in **$O(1)$** time when implemented appropriately. Traversing a linked list takes $O(n)$ time because nodes must be visited sequentially. However, inserting a node after a known node takes $O(1)$ time, provided no additional search is needed.

Searching a balanced Binary Search Tree (BST) takes $O(\log n)$ time because each comparison allows the search to continue down one subtree. In a skewed tree, where nodes form a chain, searching may take $O(n)$ time.

Tree traversal algorithms visit every node in a particular order:

- **Breadth-first traversal (BFS)** visits nodes level by level, typically using a queue.
- **Depth-first traversal (DFS)** explores a subtree before returning to visit another, using recursion or an explicit stack.
- **Post-order traversal** is a depth-first traversal in which the left subtree is visited first, followed by the right subtree, and finally the node itself.

Worked example 1 — singly linked list: pointer updates

A singly linked list consists of nodes, each containing data and a pointer to the next node. The following procedures insert or remove a node immediately after a specified node.

```

Procedure insertAfter(node, newNode)
    newNode.next = node.next
    node.next = newNode
endprocedure

```

```

procedure deleteAfter(node)
    if node.next is not null then
        node.next = node.next.next
    endif
endprocedure

```

Tracing the linked list

Suppose the initial list contains three nodes:

A → C → D → null

We want to insert a new node B immediately after A.

Step 1 – Insert B

insertAfter(A, B)

First, newNode.next = node.next makes B point to C.

Then, node.next = newNode changes A's pointer to B.

The list becomes:

A → B → C → D → null

Step 2 – Delete the node after A

deleteAfter(A)

The procedure updates A's pointer to skip B and point directly to C.

The list becomes:

A → C → D → null

Operation	Pointer change	Resulting list
Initial state	No changes	A → C → D
Insert B after A	B.next = C, then A.next = B	A → B → C → D
Delete after A	A.next = C	A → C → D

Trace it: When inserting B, the order of the pointer updates is essential. The first instruction, newNode.next = node.next, preserves the link to C by making B point to it. Only then does node.next = newNode change A's pointer to B.

If A's pointer were changed first, the original reference to C could be lost, disconnecting the remaining nodes from the list.

During deletion, node.next = node.next.next bypasses the node immediately after A. In this example, B is removed from the list, but C and D remain connected.

Notice that deleting a node from the list does not necessarily erase it from memory immediately. The procedure removes the link to that node; memory reclamation depends on the programming language and its memory-management system.

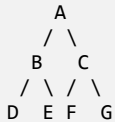
Common misconception. Insertion after a known node takes $O(1)$ time, but finding that node may require $O(n)$ time. The complexity of the overall operation therefore depends on whether the insertion position is already known.

Worked example 2 — breadth-first tree traversal

Breadth-first traversal visits nodes level by level, starting at the root. A queue ensures that nodes are processed in the order they are discovered.

```
procedure breadthFirst(root)
  create empty queue
  enqueue(root)
  while queue is not empty
    node = dequeue()
    visit(node)
    for each child of node
      enqueue(child)
    next child
  endwhile
endprocedure
```

Consider the following binary tree:



Tracing the queue

Assume children are processed from left to right.

Step	Node visited	Queue after processing
Start	–	[A]
1	A	[B, C]
2	B	[C, D, E]
3	C	[D, E, F, G]
4	D	[E, F, G]
5	E	[F, G]
6	F	[G]
7	G	Empty

Traversal order: A → B → C → D → E → F → G

Trace it: Initially, the root node A is added to the empty queue.

The first iteration removes A using dequeue() and visits it. Its children, B and C, are then added to the rear of the queue.

The next iteration removes B, leaving C at the front. B's children, D and E, are added to the rear. The queue now contains C, D and E.

Next, C is removed and visited. Its children, F and G, are added to the queue. The remaining nodes are then removed and visited in order: D, E, F and G.

Because the queue follows the First In, First Out (FIFO) principle, nodes discovered earlier are processed before nodes discovered later. This ensures that all nodes at one level are visited before moving to the next level.

The algorithm finishes when the queue becomes empty.

Exam language. When tracing breadth-first traversal, record the queue after each node is processed. Remember that dequeue() removes an item from the front, while enqueue() adds new children to the rear.

Worked example 3 — depth-first post-order tree traversal

Post-order traversal is a depth-first technique that visits a node only after visiting its left and right subtrees.

The order is:

Left subtree → Right subtree → Node

```
procedure postOrder(node)
  if node is not null then
    postOrder(node.left)
    postOrder(node.right)
    visit(node)
  endif
endprocedure
```

Consider the same binary tree used in Worked example 2:



Tracing the recursive calls

Step	Node visited	Explanation
1	D	Visit the left child of B.
2	E	Visit the right child of B.
3	B	Both subtrees of B have been visited.
4	F	Visit the left child of C.
5	G	Visit the right child of C.
6	C	Both subtrees of C have been visited.
7	A	Both subtrees of the root have been visited.

Traversal order: D → E → B → F → G → C → A

Trace it: The procedure begins at the root A, but does not visit it immediately. Instead, it recursively calls `postOrder(node.left)`, moving into the left subtree rooted at B.

From B, the procedure moves to D. Since D has no children, its recursive calls receive null and return without visiting anything. D is then visited.

Execution returns to B and moves into its right subtree, visiting E. Once both D and E have been visited, the procedure finally visits B.

The same process occurs in the right subtree of A. F is visited first, followed by G, and then C.

Finally, execution returns to A. Since both its left and right subtrees have been completely traversed, A is visited last.

This illustrates the defining feature of post-order traversal: **a parent node is visited only after all nodes in its subtrees have been visited.**

Each recursive call uses a stack frame to preserve the current node and the point to which execution must return. The call stack therefore allows the algorithm to return to parent nodes after exploring their children.

Common misconception. Depth-first traversal does not always produce the same visiting order. Pre-order visits the node before its subtrees, in-order visits it between the left and right subtrees, and post-order visits it after both subtrees.

Exam language. When tracing post-order traversal, remember **Left → Right → Node**. Do not write down a parent node until both its subtrees have been fully processed.

8.3 Searching and sorting

Algorithm	Typical complexity / condition	Key idea
Linear search	$O(n)$	Check items one by one; no sorting required.
Binary search	$O(\log n)$; data must be sorted	Compare middle item and discard half the remaining range.
Bubble sort	$O(n^2)$ worst/typical simple form	Swap adjacent out-of-order pairs over repeated passes.

Insertion sort	$O(n^2)$ worst; can be efficient for small/nearly sorted data	Grow a sorted prefix by inserting each next value.
Merge sort	$O(n \log n)$, $O(n)$ auxiliary storage in standard implementation	Split, recursively sort halves, merge.
Quicksort	Average $O(n \log n)$, worst $O(n^2)$	Partition around a pivot, recursively sort partitions.

Worked example 4 — binary search

Binary search is an efficient algorithm for finding a target value in a **sorted array**. Instead of checking every element, it repeatedly examines the middle element and eliminates half of the remaining search space. The following pseudocode searches an array and returns the index of the target value. If the target is not found, it returns -1.

```
function binarySearch(a, target)
    low = 0
    high = a.length - 1

    while low <= high
        mid = (low + high) DIV 2

        if a[mid] == target then
            return mid
        endif

        if a[mid] < target then
            low = mid + 1
        else
            high = mid - 1
        endif
    endwhile

    return -1
endfunction
```

The algorithm uses three variables:

- **low** identifies the first index in the current search range.
- **high** identifies the last index in the current search range.
- **mid** identifies the middle index, calculated using integer division (DIV).

Tracing binary search

Suppose we want to find the value **29** in the following sorted array:

Index: 0 1 2 3 4 5 6 7 8

Value: 3 7 11 16 22 29 34 41 56

The array contains nine elements, so initially **low = 0** and **high = 8**.

Iteration	low	high	mid	a[mid]	Action
1	0	8	4	22	22 is less than 29; set low = 5.
2	5	8	6	34	34 is greater than 29; set high = 5.
3	5	5	5	29	Target found; return 5.

Result: Index 5

Trace it: Initially, the search covers the entire array, from index 0 to index 8.

During the first iteration, $mid = (0 + 8) \text{ DIV } 2 = 4$. The value at index 4 is 22. Since 22 is smaller than 29, the target cannot be in the left-hand portion of the sorted array. The algorithm therefore sets $low = mid + 1 = 5$, eliminating indices 0–4.

During the second iteration, $mid = (5 + 8) \text{ DIV } 2 = 6$. The value at index 6 is 34. Since 34 is greater than 29, the target must lie to the left of index 6. The algorithm sets $high = mid - 1 = 5$.

Now both low and high equal 5. The next middle index is therefore 5, where the value is 29. The condition $a[mid] == target$ is true, so the function returns 5.

Notice that the algorithm locates the target using only **three comparisons**, rather than examining every element sequentially.

If the target were absent, the search would eventually reach a point where $low > high$. The loop would terminate, and the function would return -1 to indicate that the value was not found.

Time complexity: Binary search has a worst-case time complexity of $O(\log n)$ because each unsuccessful comparison eliminates approximately half of the remaining elements. A successful match on the first comparison takes $O(1)$ time.

Common misconception. Binary search does not work reliably on an unsorted array. Its efficiency depends on knowing that values to the left of the middle are smaller and values to the right are larger, assuming ascending order.

Exam language. When tracing binary search, record low, high and mid after each comparison. Remember that DIV performs integer division, discarding any fractional part. Array indices normally begin at 0 unless stated otherwise.

Worked example 5 — Simple quicksort — clear but not in-place

```
function quicksort(a)
```

Quicksort is a divide-and-conquer sorting algorithm. It selects a value called the **pivot**, partitions the remaining values into two groups, recursively sorts each group and combines the results.

In this simplified implementation, the first element is selected as the pivot:

```
function quicksort(a)
    if a.length <= 1 then
        return a
    endif

    pivot = a[0]

    smaller = values after a[0] that are < pivot
    larger = values after a[0] that are >= pivot

    return quicksort(smaller) + [pivot] + quicksort(larger)
endfunction
```

Here, the two partitioning statements are high-level pseudocode describing how sublists are formed, rather than complete executable instructions. The + operator represents list concatenation.

The algorithm follows four steps:

1. **Choose a pivot:** Select the first element of the array.
2. **Partition:** Place values smaller than the pivot into smaller, and values greater than or equal to it into larger.
3. **Recurse:** Apply quicksort to both sublists.
4. **Combine:** Concatenate the sorted smaller sublist, the pivot and the sorted larger sublist.

Tracing quicksort

Suppose we want to sort the following array into ascending order:

[7, 3, 9, 3, 5, 8]

Step 1 – Choose the first pivot

The first element is 7.

```
Pivot:      7
Smaller:    [3, 3, 5]
Larger:     [9, 8]
```

The values are divided into two groups based on their comparison with 7. Notice that both occurrences of 3 are retained.

Step 2 – Sort the smaller sublist

Apply quicksort recursively to [3, 3, 5].

```
Pivot:      3
Smaller:    []
Larger:     [3, 5]
```

The smaller sublist is empty, so it has reached the base case.

The larger sublist [3, 5] must still be sorted:

```
Pivot:      3
Smaller:    []
Larger:     [5]
```

Both resulting sublists contain zero or one element, so no further partitioning is required.

Combining the results gives:

[3, 3, 5]

Step 3 – Sort the larger sublist

Return to the original larger sublist [9, 8].

Choose 9 as the pivot:

```
Pivot:      9
Smaller:    [8]
Larger:     []
```

Both sublists have reached the base case.

Combining the results gives:

[8, 9]

Step 4 – Combine the sorted sublists

The original pivot was 7. Place it between the two sorted sublists.

```
Sorted smaller: [3, 3, 5]
Original pivot:  7
Sorted larger:  [8, 9]
```

Final result: [3, 3, 5, 7, 8, 9]

Trace it: Quicksort begins by selecting 7 as the pivot. All values smaller than 7 are placed in one sublist, while values greater than or equal to 7 are placed in another.

The algorithm then calls itself recursively on each sublist. For [3, 3, 5], the first pivot is 3. The remaining 3 belongs in the larger sublist because the condition is $\geq$ pivot, not $>$ pivot. This ensures that duplicate values are preserved.

Each recursive call continues until its sublist contains zero or one element. Such a sublist is already sorted, so the function returns it immediately.

As the recursive calls return, the smaller sorted sublist, pivot and larger sorted sublist are concatenated.

The final result is [3, 3, 5, 7, 8, 9].

Notice that the algorithm does not simply move the pivot into its final position and stop. It must also sort the two sublists, which is why recursion is essential to this implementation.

Time complexity: Quicksort has an average time complexity of $O(n \log n)$ when partitions are reasonably balanced. In the worst case, such as repeatedly selecting the smallest or largest element as the pivot, its time complexity becomes $O(n^2)$.

Common misconception. Quicksort does not require the pivot to be the middle element. The pivot may be selected in several ways, including choosing the first, last or a random element. The choice affects how balanced the partitions are and therefore the algorithm's performance.

Exam language. When tracing quicksort, identify the pivot at each recursive call, record which values enter the smaller and larger partitions, and follow the recursive calls until the base cases are reached. Then combine the sorted sublists in the correct order.

Quicksort detail. Quicksort can be implemented in-place, rearranging elements within the original array. However, this simplified teaching version creates additional sublists during partitioning and therefore requires extra memory. The amount of additional storage depends on the implementation and recursion depth

8.4 Graph traversal, Dijkstra and A*

Depth-first search explores along a path before backtracking and uses a stack (explicitly or through recursion). Breadth-first search explores in layers and uses a queue; in an unweighted graph, BFS discovers a shortest path in number of edges.

Dijkstra's algorithm finds shortest/cheapest paths when all edge weights are non-negative. Give the start distance 0 and all others infinity. Repeatedly choose the unvisited node with the smallest tentative distance, relax its outgoing edges, then finalise it. Once a node is finalised under these conditions, its shortest distance is known.

A search:* A* extends shortest-path searching by using a heuristic, $h(n)$, which estimates the remaining cost from node n to the goal. It selects the node with the smallest value of $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path found so far and $h(n)$ is the estimated remaining cost.

An *admissible heuristic* never overestimates the true remaining cost. With appropriate handling of repeated nodes, this allows A* to find an optimal path while potentially exploring fewer nodes than Dijkstra's algorithm.

Worked example 6 — Dijkstra from A to E

Seven roads join five places. A to B costs 4, A to C costs 2, B to C costs 1, B to D costs 5, C to D costs 8, C to E costs 10, and D to E costs 2. Start at A. Give A a distance of 0 and every other place infinity. At each step, finalise the unvisited place with the smallest distance, then relax the roads leading out of it.

Step	Finalised	A	B	C	D	E
Start	—	0	∞	∞	∞	∞
1	A	0	4	2	∞	∞
2	C	0	3	2	10	12
3	B	0	3	2	8	12
4	D	0	3	2	8	10
5	E	0	3	2	8	10

All distances are measured from A. Infinity means that no route has yet been discovered.

Shortest route: A → C → B → D → E

Total cost: 10

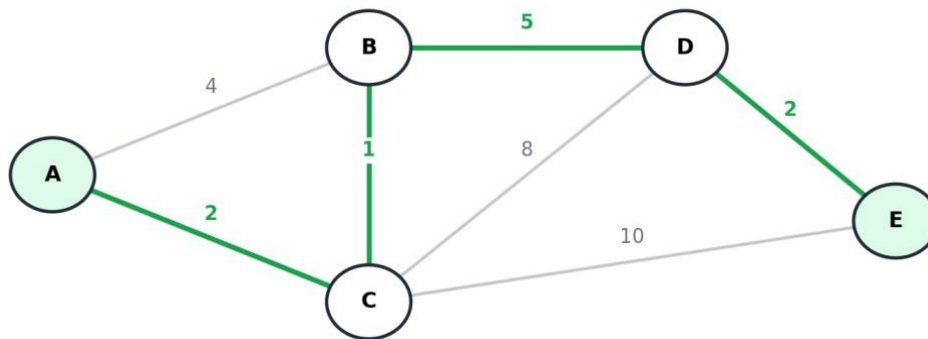
Trace it: Three distances improve after they are first calculated. B initially has a distance of 4 via the direct road from A, but step 2 discovers that A → C → B costs only 3. D initially has a distance of 10 through C, but step 3 finds a shorter route through B, reducing its distance to 8. E initially has a distance of 12 through C, but step 4 discovers a shorter route through D, reducing its distance to 10.

The shortest route from A to E is therefore A → C → B → D → E, with a total cost of 2 + 1 + 5 + 2 = 10.

This demonstrates *relaxation*: when a shorter route to a node is discovered, its tentative distance is updated.

A distance remains provisional until the node is finalised, at which point its shortest distance from the starting node is known, provided all edge weights are non-negative.

Dijkstra's algorithm: cheapest route from A to every node



shortest A to E is not the direct-looking A-C-E (12) but A-C-B-D-E (2+1+5+2 = 10).
Dijkstra finds it by always expanding the cheapest unvisited node.

Fig 8.2 — Dijkstra's algorithm on a weighted graph

Common misconception. A* is not “an approximate version of Dijkstra”. The heuristic guides which node to explore next. With a suitable admissible heuristic it can still return an optimal path.

Section 8 exercises

Quick check

1. State the precondition of binary search.
2. Give the average and worst-case time complexities of quicksort.
3. Which data structure drives breadth-first traversal?
4. What condition do edge weights need for Dijkstra's standard algorithm?

Practice

1. For a sorted list of 1024 items, estimate the maximum number of halvings needed by binary search.
2. Trace insertion sort on [7, 3, 5, 2].
3. A binary tree has root 40; its left child is 20 with children 10 and 30; its right child is 60 with children 50 and 70. Give the post-order and breadth-first traversals.

Exam-style

1. A system must sort ten million records but has tight memory limits. Compare merge sort and an in-place quicksort, including time and space. [8]
2. Explain why a route planner might prefer A* to Dijkstra and the condition its heuristic should satisfy. [6]

Section 8 — quick revision

Try these from memory first, then use Key facts to check your own answers.

Check yourself

1. Why does binary search require sorted data?
2. State quicksort's average and worst-case complexities and describe an input/pivot pattern that can trigger the worst case.
3. Which data structure drives breadth-first traversal and which drives depth-first traversal? Justify both.
4. What must be true of edge weights for Dijkstra's standard algorithm, and what does an admissible A* heuristic guarantee?

Key facts

- Big O compares growth; distinguish constant, logarithmic, linear, polynomial and exponential classes.
- Binary search is $O(\log n)$ but requires sorted data.
- Merge sort is $O(n \log n)$ with extra merge storage; quicksort averages $O(n \log n)$ but can reach $O(n^2)$.
- Breadth-first traversal uses a queue; depth-first traversal uses a stack/recursion. OCR names post-order tree traversal explicitly.
- Dijkstra assumes non-negative edge weights; A* adds $g+h$ and needs a suitable admissible heuristic for optimality.

Appendix A — OCR-style pseudocode quick reference

OCR questions use a consistent exam reference language. Exact syntax can vary slightly in student answers if the logic is unambiguous, but you should be comfortable reading and writing the following patterns.

Assignment, input and output

```
name = input("Name")
score = int(input("Score"))
print(name)
print(score)
```

Selection and iteration

```
if score >= 80 then
    grade = "A"
elseif score >= 70 then
    grade = "B"
else
    grade = "C or below"
endif

for i = 0 to 9
    print(i)
next i

while total < 100
    total = total + inputValue
endwhile
```

Functions, procedures and arrays

```
function maximum(a, b)
    if a > b then return a else return b endif
endfunction

procedure swapByReference(a:byRef, b:byRef)
    temp = a
    a = b
    b = temp
endprocedure

array scores[10]
scores[0] = 42
```

Operators and useful ideas

Type	Examples
Arithmetic	+ - * / DIV MOD
Comparison	== != < <= > >=
Boolean	AND OR NOT
String	length, substring/slice, concatenate, upper/lower – exact functions will be clear from the question context.

OCR convention: use = for assignment, == for equality and != for “not equal”. When pass-by-reference matters, OCR marks the parameter with :byRef.

When a question supplies unfamiliar functions, read their description rather than guessing from a language you know. Pseudocode is there to test the algorithm, not Python/Java syntax trivia.

.

Appendix B — The programming project (NEA)

The programming project is 20% of the A level and carries 70 marks. OCR allocates 10 marks to analysis, 15 to design, 25 to development and testing, and 20 to evaluation and documentation. The evidence should show the problem-solving journey as well as the finished program.

Analysis — 10 marks

- Define the problem and stakeholders/users.
- Investigate existing solutions and justify the need for your system.
- State measurable success criteria that can be tested later.
- Identify constraints, inputs, outputs, data and likely algorithms.

Design — 15 marks

- Decompose the solution into modules/classes and data structures.
- Design algorithms before coding; use pseudocode/flowcharts where useful.
- Design validation, file/database structures and interfaces.
- Plan testing, including normal, boundary and invalid cases.

Development and testing — 25 marks

- Build iteratively and explain important technical decisions.
- Show evidence of code evolving, problems found and fixes made.
- Test modules as they are built, not only at the end.
- Use version control/backups and keep evidence while it exists.

Evaluation and documentation — 20 marks

- Test every success criterion with evidence.
- Gather relevant user feedback.
- Explain limitations honestly and identify realistic future improvements.
- Evaluate maintainability and how the solution could be adapted.

One habit worth marks. Write the documentation as you go. Reconstructing months of design and testing from memory at the end is difficult and usually produces thin evidence.

Authenticity and use of AI. The programming project must be the student's own work. Any assistance, external code, sources or use of AI tools must be acknowledged in accordance with current OCR and JCQ requirements. Students and teachers must complete the applicable authentication declarations. Always consult the latest OCR guidance before submitting the project.

Appendix C — Specification coverage checklist

Use this as a final audit. Tick each line only when you can explain it without looking at the book and answer a scenario question about it.

✓	Spec	You can explain and apply...
☐	1.1.1	Processor components/registers/buses; fetch–decode–execute; performance; pipelining; von Neumann/Harvard/contemporary
☐	1.1.2	CISC/RISC; GPUs; multicore and parallel systems
☐	1.1.3	Input/output/storage scenarios; magnetic/flash/optical; RAM/ROM; virtual storage
☐	1.2.1	OS purpose; paging/segmentation/virtual memory; interrupts/ISR; scheduling; OS types; BIOS; drivers; virtual machines
☐	1.2.2	Applications/utilities; open vs closed source; assemblers/compilers/interpreters; compilation stages; linkers/loaders/libraries
☐	1.2.3	Waterfall, agile, XP, spiral, RAD; merits/drawbacks; writing/following algorithms
☐	1.2.4	Programming paradigms; procedural; assembly and LMC; immediate/direct/indirect/indexed addressing; OOP concepts
☐	1.3.1	Lossy/lossless; RLE/dictionary coding; symmetric/asymmetric encryption; hashing uses
☐	1.3.2	Relational/flat file; keys; ER modelling; normalisation/indexing; capture/select/manage/exchange; 3NF; SQL; referential integrity; ACID/locking/redundancy
☐	1.3.3	Networks/protocols/standards; TCP/IP; DNS; layering; LAN/WAN; packet/circuit switching; threats/security; hardware; client-server/P2P
☐	1.3.4	HTML/CSS/JavaScript; indexing; PageRank; client/server processing
☐	1.4.1	Primitive types; binary/hex; signed numbers; binary arithmetic; floating-point representation/normalisation/arithmetic; bitwise masks; ASCII/Unicode
☐	1.4.2	Arrays, records, lists, tuples; linked lists, graphs, stacks, queues, trees, BSTs, hash tables; create/traverse/add/remove
☐	1.4.3	Boolean logic; algebra laws; K-maps; gates/truth tables; D flip-flops; half/full adders
☐	1.5.1	Data protection; Computer Misuse; Copyright, Designs and Patents; RIPA — purpose and application
☐	1.5.2	Workforce; automated decisions; AI; environment; censorship; monitoring; personal information; piracy/offensive communications; cultural design
☐	2.1.1	Thinking abstractly
☐	2.1.2	Thinking ahead, caching, reusable components
☐	2.1.3	Thinking procedurally

☐	2.1.4	Thinking logically
☐	2.1.5	Thinking concurrently
☐	2.2.1	Sequence/iteration/selection; recursion; scope; modularity; functions/procedures; value/reference; IDE; OOP techniques
☐	2.2.2	Problem recognition/decomposition; divide and conquer; abstraction; backtracking; data mining; heuristics; performance modelling; pipelining; visualisation
☐	2.3.1	Algorithm design/suitability; time/space; Big O; data-structure algorithms; tree BFS/post-order DFS; bubble/insertion/merge/quick; Dijkstra/A*; binary/linear search

Acknowledgement of AI assistance

AI tools, including ChatGPT and Claude, were used during the preparation of this book to assist with editing, improving explanations, developing worked examples and reviewing technical accuracy and consistency.

The author directed the development of the material and retains responsibility for its content, accuracy and educational approach. AI-generated suggestions were reviewed and revised before inclusion.

References and verification notes

Verification and edition notes

The content of this book was reviewed in October 2026 against the OCR A Level Computer Science H446 qualification materials and specification version 2.7 (March 2024). Publicly available OCR question papers, mark schemes and examiner reports were also consulted to check assessment emphasis, including Little Man Computer (LMC), addressing modes and programming language concepts.

The OCR A Level H446/02 examination paper is titled *Algorithms and programming*, while *Algorithms and problem solving* is the title of the corresponding AS Level H046/02 paper. Little Man Computer is included under Section 1.2.4, *Types of Programming Language*, of the H446 specification.

The specification coverage checklist included in this book is intended as a teaching and revision guide, rather than a reproduction of the OCR specification. Topic descriptions have been paraphrased and expanded to provide accessible explanations, worked examples and opportunities for independent study. Where legislation or technology has changed since the original specification was published, the book distinguishes between the terminology used in the specification and the current technical or legal context. This book is an independent educational resource. It is not published, approved or endorsed by OCR.

References and further reading

American National Standards Institute (1986) *Coded Character Set — 7-Bit American National Standard Code for Information Interchange (ASCII)*. ANSI X3.4-1986. New York: ANSI.

Brin, S. and Page, L. (1998) ‘The anatomy of a large-scale hypertextual Web search engine’, *Computer Networks and ISDN Systems*, 30(1–7), pp. 107–117. Available at: [https://doi.org/10.1016/S0169-7552\(98\)00110-X](https://doi.org/10.1016/S0169-7552(98)00110-X).

Brookshear, J.G. and Brylow, D. (2019) *Computer Science: An Overview*. 13th edn. Harlow: Pearson.

Connolly, T. and Begg, C. (2015) *Database Systems: A Practical Approach to Design, Implementation and Management*. 6th edn. Harlow: Pearson.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2022) *Introduction to Algorithms*. 4th edn. Cambridge, MA: MIT Press.

Dijkstra, E.W. (1959) ‘A note on two problems in connexion with graphs’, *Numerische Mathematik*, 1, pp. 269–271. Available at: <https://doi.org/10.1007/BF01386390>.

Ecma International (2024) *ECMA-262: ECMAScript 2024 Language Specification*. 15th edn. Available at: <https://ecma-international.org/publications-and-standards/standards/ecma-262/> (Accessed: 15 October 2026).

Eddy, W. (2022) *RFC 9293: Transmission Control Protocol (TCP)*. RFC Editor. Available at: <https://www.rfc-editor.org/rfc/rfc9293> (Accessed: 15 October 2026).

Great Britain (1988) *Copyright, Designs and Patents Act 1988*, c. 48. Available at: <https://www.legislation.gov.uk/ukpga/1988/48> (Accessed: 15 October 2026).

Great Britain (1990) *Computer Misuse Act 1990*, c. 18. Available at: <https://www.legislation.gov.uk/ukpga/1990/18> (Accessed: 15 October 2026).

Great Britain (1998) *Data Protection Act 1998*, c. 29. Available at: <https://www.legislation.gov.uk/ukpga/1998/29> (Accessed: 15 October 2026).

Great Britain (2000) *Regulation of Investigatory Powers Act 2000*, c. 23. Available at: <https://www.legislation.gov.uk/ukpga/2000/23> (Accessed: 15 October 2026).

Great Britain (2018) *Data Protection Act 2018*, c. 12. Available at: <https://www.legislation.gov.uk/ukpga/2018/12> (Accessed: 15 October 2026).

Great Britain (2025) *Data (Use and Access) Act 2025*, c. 18. Available at: <https://www.legislation.gov.uk/ukpga/2025/18> (Accessed: 15 October 2026).

- Hart, P.E., Nilsson, N.J. and Raphael, B. (1968) 'A formal basis for the heuristic determination of minimum cost paths', *IEEE Transactions on Systems Science and Cybernetics*, 4(2), pp. 100–107. Available at: <https://doi.org/10.1109/TSSC.1968.300136>.
- Heathcote, P.M. and Heathcote, R.S.U. (2016) *OCR AS and A Level Computer Science*. PG Online.
- Hoare, C.A.R. (1962) 'Quicksort', *The Computer Journal*, 5(1), pp. 10–16.
- IEEE (2019) *IEEE 754-2019: IEEE Standard for Floating-Point Arithmetic*. New York: IEEE.
- Information Commissioner's Office (n.d.) *Guide to the UK GDPR*. Available at: <https://ico.org.uk/for-organisations/uk-gdpr-guidance-and-resources/> (Accessed: 15 October 2026).
- Karnaugh, M. (1953) 'The map method for synthesis of combinational logic circuits', *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5), pp. 593–599.
- Kurose, J.F. and Ross, K.W. (2021) *Computer Networking: A Top-Down Approach*. 8th edn. Harlow: Pearson.
- Matthes, E. (2023) *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. 3rd edn. San Francisco, CA: No Starch Press.
- MDN Web Docs (n.d.) *JavaScript Guide*. Available at: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide> (Accessed: 15 October 2026).
- Mockapetris, P. (1987) *RFC 1034: Domain Names — Concepts and Facilities*. RFC Editor. Available at: <https://www.rfc-editor.org/rfc/rfc1034> (Accessed: 15 October 2026).
- OCR (2024) *A Level Computer Science H446: Specification*. Version 2.7, March 2024. Cambridge: Oxford Cambridge and RSA Examinations. Available at: <https://www.ocr.org.uk/Images/170844-specification-accredited-a-level-gce-computer-science-h446.pdf> (Accessed: 15 October 2026).
- OCR (n.d.-a) *AS and A Level Computer Science H046, H446: Assessment materials*. Available at: <https://www.ocr.org.uk/qualifications/as-and-a-level/computer-science-h046-h446-from-2015/assessment/> (Accessed: 15 October 2026).
- OCR (n.d.-b) *AS and A Level Computer Science H046, H446: Qualification overview*. Available at: <https://www.ocr.org.uk/qualifications/as-and-a-level/computer-science-h046-h446-from-2015/> (Accessed: 15 October 2026).
- OWASP Foundation (2025) *OWASP Top 10:2025*. Available at: <https://top10.owasp.org/2025/> (Accessed: 15 October 2026).
- Page, L., Brin, S., Motwani, R. and Winograd, T. (1999) *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford, CA: Stanford InfoLab.
- Patterson, D.A. and Hennessy, J.L. (2020) *Computer Organization and Design: The Hardware/Software Interface*. 6th edn. Cambridge, MA: Morgan Kaufmann.
- Postel, J. (1981a) *RFC 791: Internet Protocol*. RFC Editor. Available at: <https://www.rfc-editor.org/rfc/rfc791> (Accessed: 15 October 2026).
- Postel, J. (1981b) *RFC 793: Transmission Control Protocol*. RFC Editor. Available at: <https://www.rfc-editor.org/rfc/rfc793> (Accessed: 15 October 2026).
- Python Software Foundation (n.d.-a) *The Python Language Reference*. Available at: <https://docs.python.org/3/reference/> (Accessed: 15 October 2026).
- Python Software Foundation (n.d.-b) *The Python Standard Library*. Available at: <https://docs.python.org/3/library/> (Accessed: 15 October 2026).
- Python Software Foundation (n.d.-c) *The Python Tutorial*. Available at: <https://docs.python.org/3/tutorial/> (Accessed: 15 October 2026).
- Ramalho, L. (2022) *Fluent Python*. 2nd edn. Sebastopol, CA: O'Reilly Media.
- Rouse, G., Pitt, J. and O'Byrne, S. (2015) *OCR A Level Computer Science*. London: Hodder Education.
- Sedgewick, R. and Wayne, K. (2011) *Algorithms*. 4th edn. Boston, MA: Addison-Wesley.
- Shannon, C.E. (1948) 'A mathematical theory of communication', *The Bell System Technical Journal*, 27(3), pp. 379–423; 27(4), pp. 623–656.

- Silberschatz, A., Korth, H.F. and Sudarshan, S. (2020) *Database System Concepts*. 7th edn. New York: McGraw-Hill.
- Sommerville, I. (2016) *Software Engineering*. 10th edn. Harlow: Pearson.
- Stallings, W. (2019) *Computer Organization and Architecture: Designing for Performance*. 11th edn. Harlow: Pearson.
- Tanenbaum, A.S. and Bos, H. (2015) *Modern Operating Systems*. 4th edn. Harlow: Pearson.
- Tanenbaum, A.S. and Wetherall, D.J. (2011) *Computer Networks*. 5th edn. Harlow: Pearson.
- Turing, A.M. (1936) ‘On computable numbers, with an application to the Entscheidungsproblem’, *Proceedings of the London Mathematical Society*, s2-42(1), pp. 230–265.
- Unicode Consortium (2026) *The Unicode Standard, Version 18.0.0*. South San Francisco, CA: The Unicode Consortium. Available at: <https://www.unicode.org/versions/Unicode18.0.0/> (Accessed: 15 October 2026).
- van Rossum, G., Warsaw, B. and Coghlan, N. (2001) *PEP 8: Style Guide for Python Code*. Python Software Foundation. Available at: <https://peps.python.org/pep-0008/> (Accessed: 15 October 2026).
- W3C (n.d.) *CSS Specifications*. Available at: <https://www.w3.org/Style/CSS/> (Accessed: 15 October 2026).
- WHATWG (n.d.) *HTML Living Standard*. Available at: <https://html.spec.whatwg.org/multipage/> (Accessed: 15 October 2026).
- Wing, J.M. (2006) ‘Computational thinking’, *Communications of the ACM*, 49(3), pp. 33–35. Available at: <https://doi.org/10.1145/1118178.1118215>.
- Yergeau, F. (2003) *RFC 3629: UTF-8, a transformation format of ISO 10646*. RFC Editor. Available at: <https://www.rfc-editor.org/rfc/rfc3629> (Accessed: 15 October 2026).