IB Diploma Programme

Chuyên luyện thi IELTS/SAT → AP / IB / A-levels

8020 ENGLISH

IB Diploma Programme Computer Science

Giáo trình luyện thi chuyên sâu

Song ngữ Anh–Việt

8020 ENGLISH

5A/2 Trần Phú, P.4, Q.5, TP.HCM · luyenthi8020.com

Thầy Tường 0332 747 169 · Cô Nancy 0983 422 692

Lời mở đầu · Foreword

Cuốn sách này thuộc bộ giáo trình luyện thi chương trình quốc tế của **8020 English (8020 Education)** — trung tâm luyện thi trọng điểm **IELTS · SAT · AP · IB · A-level · IGCSE** và săn học bổng du học. Toàn bộ nội dung được biên soạn bởi đội ngũ **Giáo sư · Tiến sĩ · Thạc sĩ** tốt nghiệp các trường top đầu thế giới, bám sát khung chương trình chính thức, trình bày đầy đủ lý thuyết, ví dụ mẫu, hình minh hoạ và hệ thống bài tập có lời giải chi tiết.

This book is part of the 8020 English international exam-prep series: complete English theory with Vietnamese explanations, worked examples, illustrations and fully-solved practice, aligned to the official framework.

Thành tích 8020 English

9.0 IELTS cao nhất	**1570** SAT cao nhất
5/5 AP — điểm tuyệt đối	**90/100** IB Diploma
100% GV $\geq$ 8.0 IELTS / 1500 SAT	**CMU** Tiến sĩ ĐH #1 Hoa Kỳ về CS

Đội ngũ tiêu biểu: Thầy Châu Cát Tường (Academic Head, ThS Western Sydney, top 1% IELTS 8.5) · TS Nguyễn Anh Huy (Carnegie Mellon, cựu kỹ sư Amazon) · TS Nguyễn Phước Trường (Toán, ĐH Klagenfurt) · cùng đội ngũ giảng viên SAT 1500–1600, IELTS 8.0–8.5.

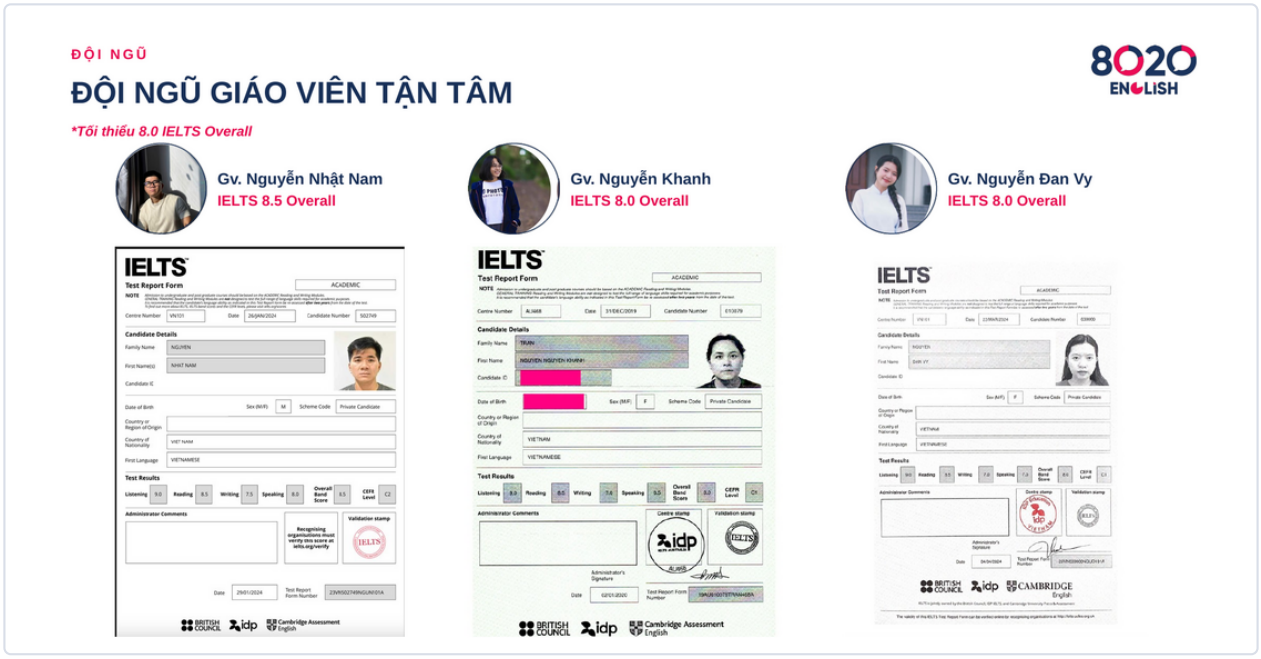
Kết quả học viên — điểm thi thật: IELTS 8.0–9.0 · SAT 1550 / 1570 / 1590 · AP 5/5 (Calculus BC, Microeconomics) · IB 90/100 (Economics) · Duolingo 110 · PTE 90 · TOEFL 120. **Bảng vàng SAT:** Khánh Đăng 1510 · Thảo Nguyên 1520 · Tâm Anh 1530.

"Cuối cùng đã đậu vào trường top như mong muốn." — Phan Thị Thanh Hằng, IELTS Overall 8.5.

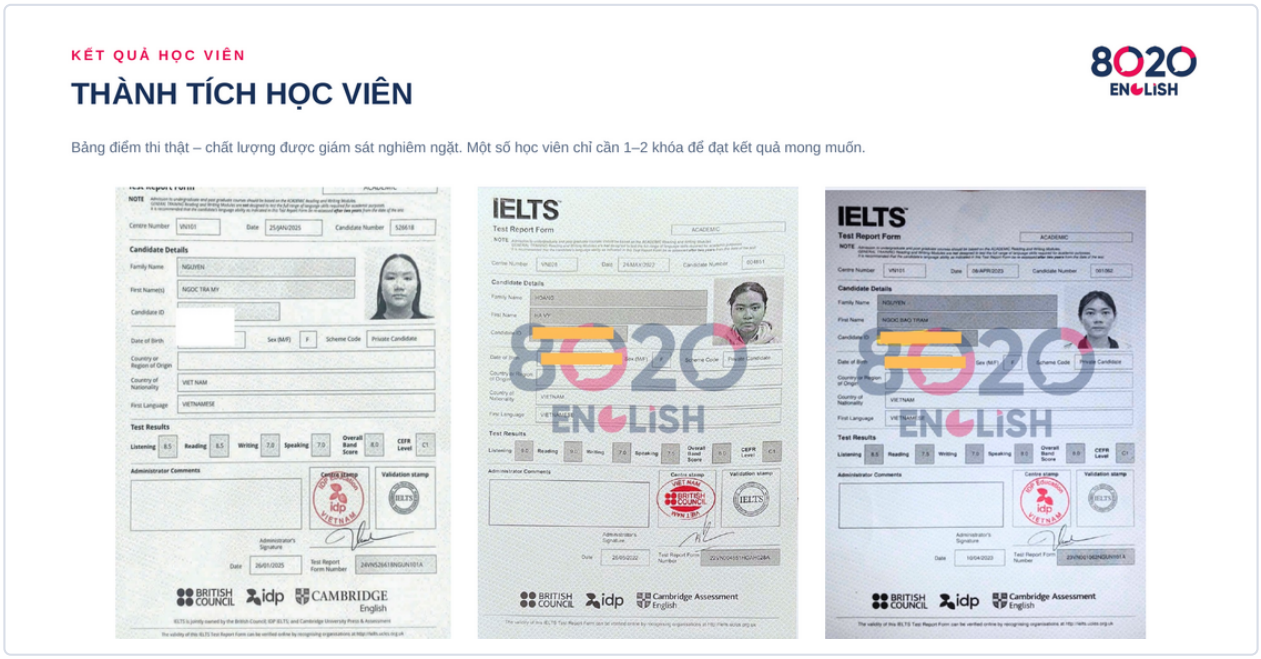
Cách dùng sách hiệu quả: mỗi Unit gồm (1) **Lý thuyết trọng tâm** tiếng Anh kèm **giải thích tiếng Việt**; (2) **Ví dụ mẫu** có lời giải; (3) **Hệ thống bài tập** kèm **lời giải chi tiết**. Hãy tự làm bài trước khi xem lời giải để đạt hiệu quả tối đa.

THÀNH TÍCH THỰC TẾ · 8020 ENGLISH

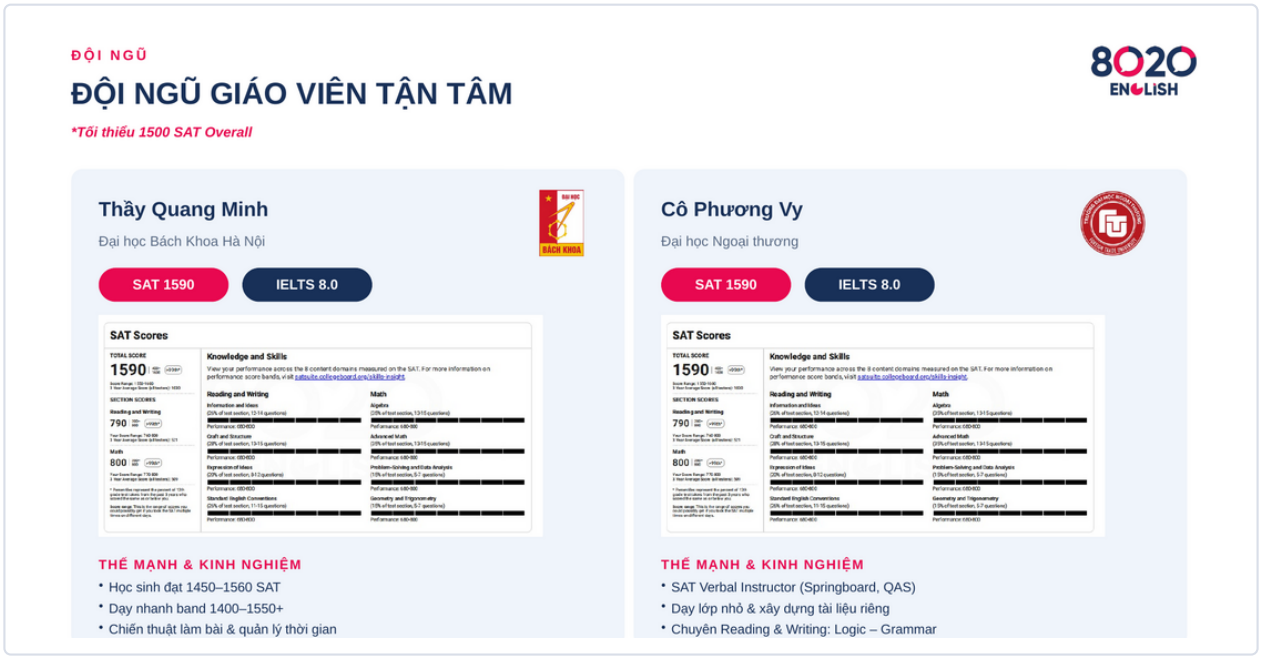
Điểm thi thật · Bằng & bảng điểm thật của giảng viên và học viên 8020 English — **Điểm thật · Thầy thật · Kết quả thật**



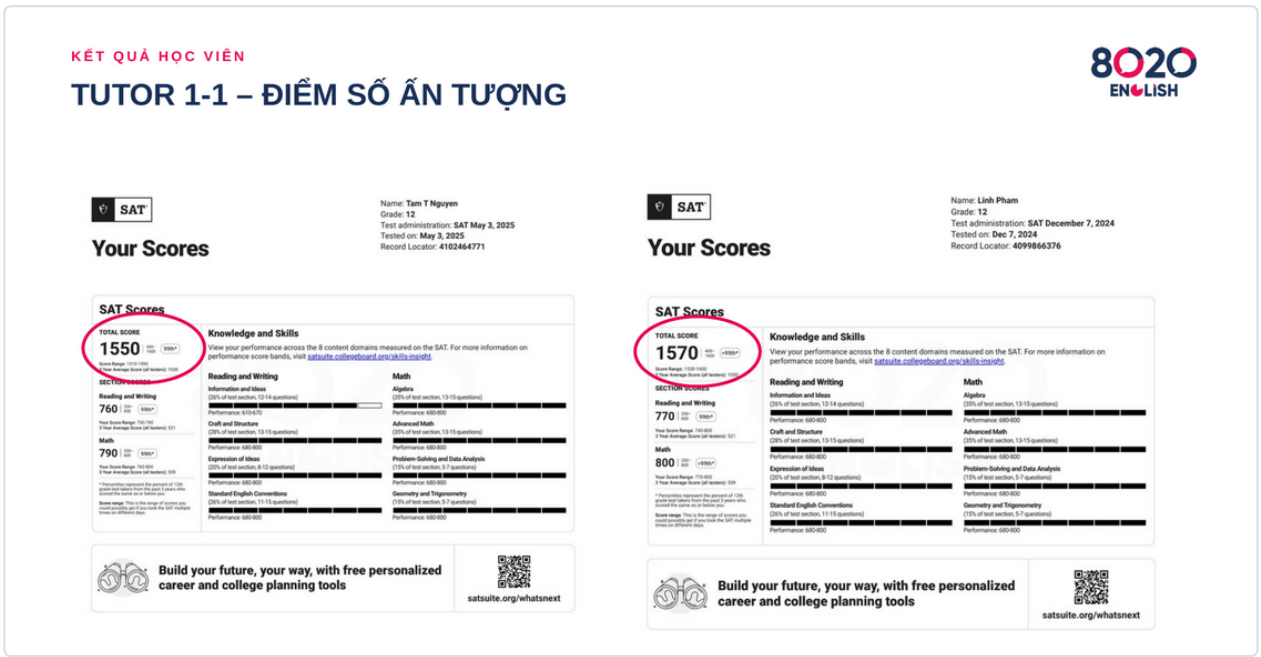
Giảng viên 8020 — IELTS 8.5 & 8.0 Overall (British Council · IDP · Cambridge)



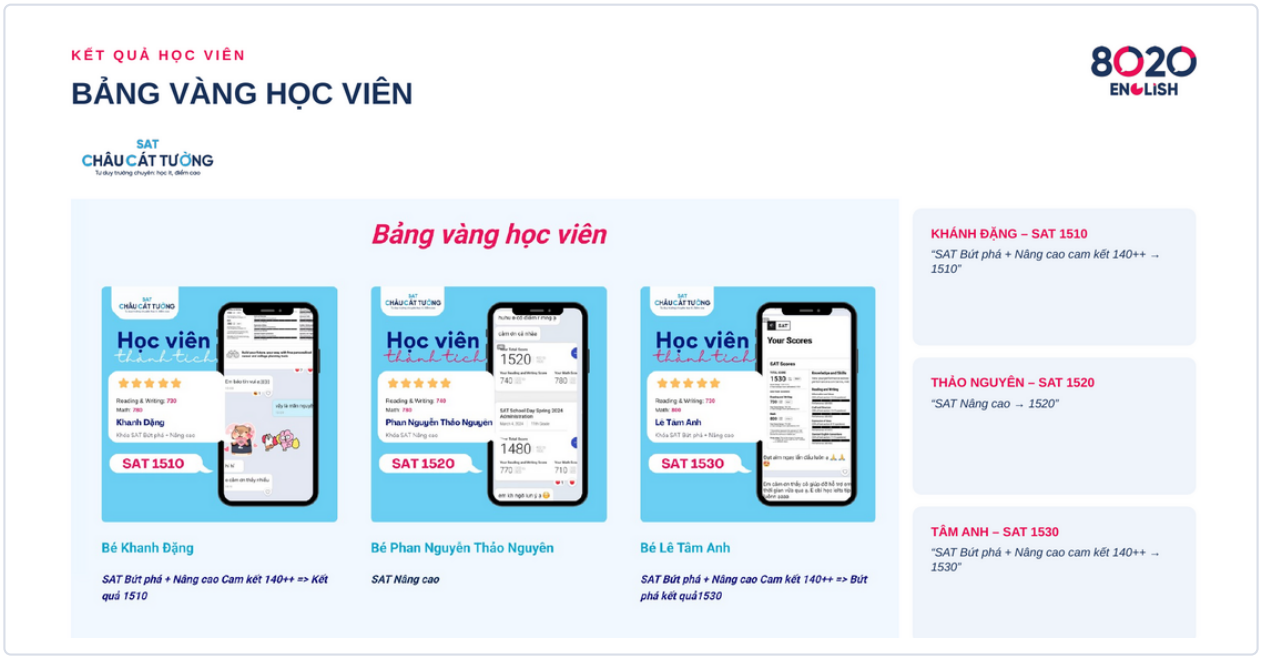
Học viên 8020 — IELTS 8.0 Overall (bảng điểm thật)



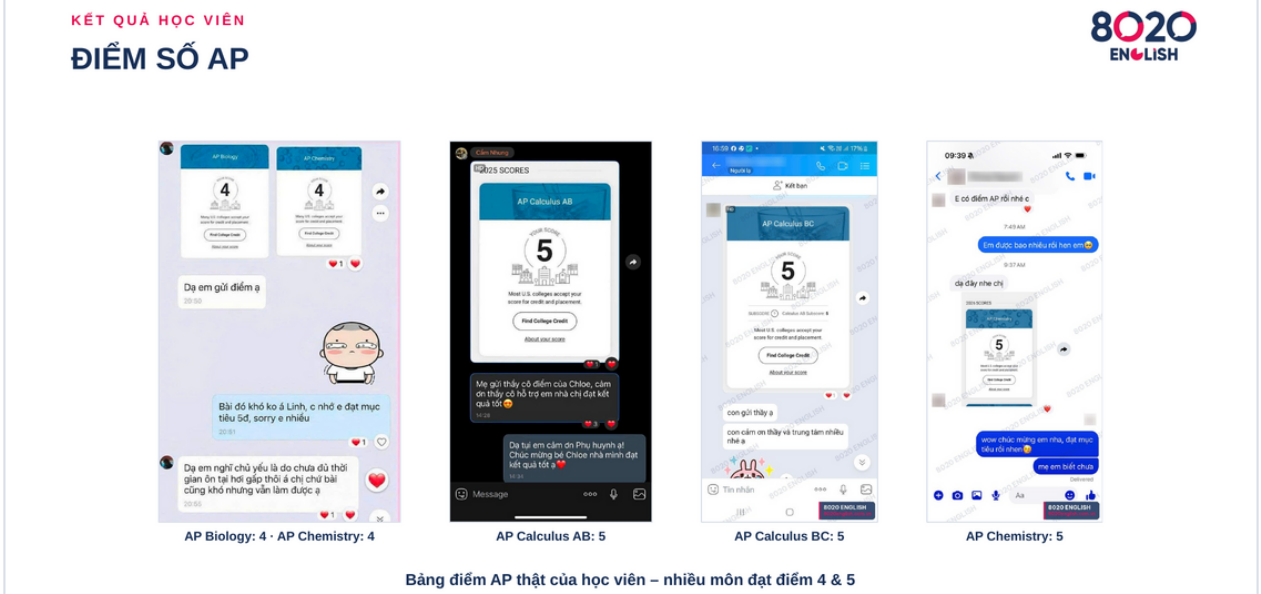
Giảng viên 8020 — SAT 1590 (ĐH Bách Khoa Hà Nội & ĐH Ngoại thương)



Học viên 8020 — SAT 1550 & 1570 (College Board)



Bảng vàng SAT — Học viên đạt 1510 · 1520 · 1530



Điểm AP thật — Calculus AB/BC 5/5 · Biology & Chemistry 4–5 (College Board)

Mục lục · Contents

System Fundamentals

Mục tiêu Unit: Hệ thống công nghệ thông tin (IT system) và các bên liên quan; vòng đời phát triển hệ thống (SDLC) từ phân tích đến đánh giá; phương pháp thu thập yêu cầu và nghiên cứu khả thi; chiến lược kiểm thử; thiết kế giao diện người–máy (HCI); độ tin cậy, sao lưu & phục hồi thảm hoạ; và các vấn đề đạo đức–xã hội của công nghệ thông tin.

Course map. Core topics **1–4** (System fundamentals, Computer organization, Networks, Computational thinking) are studied by **both SL and HL** students. Topics **5–7** (Abstract data structures, Resource management, Control) are the **HL extension** only. Every student also studies **one** Option in depth; Chapter 8 develops Databases and Object-oriented programming fully, with compact treatments of Modelling & Simulation and Web Science.

1.1 IT systems, components and stakeholders

An **IT system** is the combination of **hardware**, **software**, **data**, **procedures**, and **people** that work together to process information for a purpose. This five-part definition is examinable: removing any single component means what remains is no longer an IT system in the full sense. A computer sitting unused on a desk, with no data loaded, no procedures for operating it, and nobody using it, is just hardware — not a system.

Khái niệm cốt lõi

The **five components** of an IT system:

- **Hardware** — the physical devices (CPU, storage, input/output devices, network equipment).
- **Software** — the programs that instruct the hardware (system software and application software).
- **Data** — raw facts and figures the system stores and processes; once given context/meaning it becomes **information**.
- **Procedures** — the documented steps for operating the system correctly and safely (e.g. login protocols, backup schedules).
- **People** — everyone who interacts with or depends on the system.

VN **GIẢI THÍCH TIẾNG VIỆT**

Một **hệ thống IT** không chỉ là phần cứng và phần mềm, mà còn gồm **dữ liệu**, **quy trình** (procedures) và **con người**. Thiếu một trong năm thành phần này thì không còn là một hệ thống IT hoàn chỉnh — đây là điểm hay bị bỏ sót khi trả lời câu hỏi định nghĩa trong đề thi.

Stakeholders are anyone with an interest in a system's success or outcome. Different stakeholder groups typically have *competing* priorities, and IB exam questions frequently ask you to identify or reconcile these conflicts.

Stakeholder	Primary interest
End users	Ease of use, reliability, minimal disruption to their daily work
Managers/owners	Cost control, return on investment, meeting business goals on schedule
IT/technical staff	Maintainability, compatibility with existing infrastructure, manageable workload
Systems analysts/designers	Accurately capturing requirements and translating them into a workable design
Customers/clients (external)	Data privacy, service availability, value for money

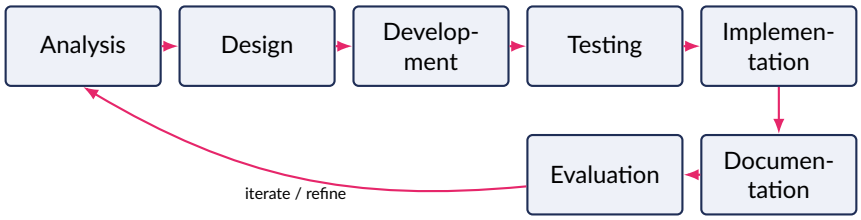
Ví dụ mẫu · Worked example

A hospital wants a new appointment-booking system. The IT department wants to reuse the existing database server to save money; the receptionists (end users) want a system that requires minimal retraining; the hospital director wants the project finished before the next fiscal year. Identify one potential conflict between these stakeholders.

The IT department's preference to reuse older infrastructure may conflict with the director's deadline if the older server cannot be configured quickly enough, or may conflict with end users if reusing legacy software means keeping an unfamiliar, outdated interface rather than a modern one that would need retraining. Resolving such conflicts is a core task of the **systems analyst** during the analysis stage.

1.2 The system life cycle: analysis and design

Every IT system passes through a predictable sequence of stages known as the **System Development Life Cycle (SDLC)**. Although in practice organisations often loop back to earlier stages (an **iterative** approach), IB exams expect you to know the stages and what happens within each.



The System Development Life Cycle (SDLC). Real projects frequently loop back to earlier stages as new requirements emerge.

Analysis begins with a **feasibility study**, which determines whether a proposed system is worth pursuing along four dimensions, followed by detailed **requirements gathering**.

Feasibility study dimensions.

- **Technical feasibility** — can the required hardware/software actually be built or acquired with current technology and in-house skills?
- **Economic feasibility** — do the projected benefits (cost savings, revenue) outweigh the development and running costs (cost–benefit analysis)?
- **Schedule feasibility** — can the system realistically be delivered within the required timeframe?
- **Legal feasibility** — does the system comply with data protection, copyright, and other applicable law?

Requirements-gathering methods: **interviews** (in-depth, but time-consuming and can be biased by leading questions); **questionnaires** (reach many people quickly and cheaply, but responses may be shallow or have low return rates); **direct observation** (reveals what people *actually* do, not just what they say, but people may change behaviour when watched — the **Hawthorne effect**); **document review** (examining existing forms, reports, manuals to understand current data flows).

VN GIẢI THÍCH TIẾNG VIỆT

Nghiên cứu khả thi có 4 khía cạnh: **kỹ thuật** (làm được không?), **kinh tế** (lợi ích có lớn hơn chi phí không?), **tiến độ** (kịp thời hạn không?), **pháp lý** (có tuân thủ luật không?). **Thu thập yêu cầu**: phỏng vấn (sâu nhưng tốn thời gian), khảo sát (nhanh, rẻ, nhưng nông), quan sát trực tiếp (thấy hành vi thật nhưng người bị quan sát có thể thay đổi hành vi — hiệu ứng Hawthorne), xem tài liệu hiện có.

Design translates requirements into a technical blueprint: data structures and file/database design, algorithm design (pseudocode/flowcharts), the user interface, and the hardware/software specification needed to run the system. Design decisions made here are far cheaper to change than defects discovered after development, which is why thorough design reduces overall project risk and cost.

(!) Lỗi thường gặp: Students often conflate **analysis** (what problem are we solving, and is it worth solving?) with **design** (how, specifically, will we solve it?). An exam answer describing a data-flow diagram or a proposed algorithm belongs to **design**; a discussion of feasibility or interviewing stakeholders belongs to **analysis**.

1.3 Development, testing and test data

Development is writing, configuring or customising the software and setting up the specified hardware. **Testing** then verifies the system behaves as intended *before* real users depend on it.

Levels of testing (smallest to largest scope): **unit testing** (one module/function in isolation); **integration testing** (do modules work correctly together?); **system testing** (does the whole system meet its specification?); **acceptance testing** (do real users/clients agree the system is ready for release?).
Test data categories — every input field should be tested with all three:

- **Normal data** — typical, valid values the system should accept and process correctly.
- **Boundary data** — values exactly at, and just outside, the edge of an accepted range (e.g. for ages 0–120: test $0, 120, -1, 121$).
- **Erroneous data** — invalid data of the wrong type or format, which the system should *reject gracefully* rather than crash on (e.g. entering `"abc"` into a numeric age field).

Ví dụ mẫu · Worked example

A form field accepts integer test scores from 0 to 100 inclusive. Give one example each of normal, boundary, and erroneous test data, and state the expected system response for each.
Normal: 75 — accepted and stored as a valid score. **Boundary:** $0, 100$ (should be accepted) and $-1, 101$ (should be rejected) — these values sit exactly on or just past the limits and are the values most likely to expose an *off-by-one* coding error. **Erroneous:** `"ninety"` or `"7O"` (letter O instead of zero) — the system should display a clear error message and refuse to store the value, rather than crashing.

VN **GIẢI THÍCH TIẾNG VIỆT**
Ba loại dữ liệu kiểm thử: **dữ liệu bình thường** (giá trị hợp lệ điển hình), **dữ liệu biên** (giá trị ngay tại và ngay ngoài giới hạn — dễ phát hiện lỗi "lệch một" off-by-one), **dữ liệu sai** (sai kiểu/định dạng, hệ thống phải từ chối một cách nhẹ nhàng chứ không được crash). Các cấp độ kiểm thử: unit → integration → system → acceptance.

Black-box vs white-box testing

Black-box testing checks that outputs are correct for given inputs *without* looking at the internal code — the tester treats the program as a sealed box. **White-box testing** examines the internal code structure directly, designing test cases to exercise every logical path/branch at least once.

1.4 Implementation, changeover and documentation

Implementation (changeover) is the process of moving from the old system to the new one. The choice of changeover method is a classic exam topic because it always involves a **risk vs cost** trade-off.

Changeover methods.

Method	Description	Risk / cost
Direct	Old system stopped, new one starts immediately	Cheapest, *highest risk*
Parallel	Both systems run side by side for a period	Safest, most expensive (double running costs)
Phased	New system introduced module by module	Moderate risk; slower, staggered rollout
Pilot	New system trialled at one site/department first	Contains risk to a small area before wider rollout

Ví dụ mẫu · Worked example

A hospital is replacing its patient-record system. Lives are at stake if the new system fails. Which changeover method is most appropriate, and why?

Parallel running is best: the old and new systems operate side by side so that if the new system produces an error, staff can immediately fall back on the reliable old system. The extra cost of running two systems simultaneously is justified by the extremely high risk of patient harm from a direct changeover failure.

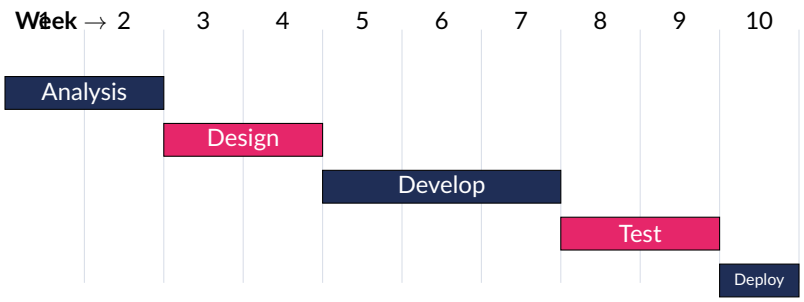
VN **GIẢI THÍCH TIẾNG VIỆT**

Bốn phương pháp chuyển đổi hệ thống: **trực tiếp** (rẻ nhưng rủi ro cao nhất — hệ cũ dừng, hệ mới chạy ngay), **song song** (an toàn nhất nhưng tốn kém nhất — chạy cả hai hệ cùng lúc), **theo giai đoạn** (từng phần một, rủi ro vừa phải), **thí điểm** (thử ở một địa điểm nhỏ trước khi nhân rộng). Bài thi hay hỏi: hệ thống rủi ro cao (bệnh viện, ngân hàng) ⇒ chọn song song hoặc thí điểm.

Documentation comprises **technical documentation** (system design, code comments, data dictionaries, network diagrams — for future developers/maintainers) and **user documentation** (manuals, quick-start guides, tutorials, FAQs — for end users). **Evaluation** closes the loop: the finished system is compared against the *original* requirements specification, assessing whether it is efficient, reliable, easy to use, appropriate to its purpose, and asking users/clients for structured feedback.

1.5 Project management: Gantt charts and critical path

Large systems projects are planned and tracked using project-management tools that make dependencies and deadlines visible to every stakeholder.



A simplified Gantt chart: horizontal bars show each task's start week and duration, making overlaps and the overall timeline visible at a glance.

Gantt chart vs critical path

A **Gantt chart** is a horizontal bar chart showing when each task starts, how long it lasts, and which tasks overlap — useful for scheduling and progress tracking. A **critical path** is the longest sequence of *dependent* tasks from start to finish; delaying any task on the critical path delays the entire project, whereas tasks with **slack/float** (not on the critical path) can be delayed slightly without affecting the finish date.

VN **GIẢI THÍCH TIẾNG VIỆT**

Biểu đồ Gantt thể hiện thời điểm bắt đầu/kết thúc và độ dài từng công việc bằng các thanh ngang. **Đường găng (critical path)** là chuỗi công việc phụ thuộc nhau dài nhất — trễ bất kỳ công việc nào trên đường găng sẽ làm trễ toàn dự án; công việc không nằm trên đường găng có "thời gian dự trữ" (slack/float).

1.6 Human--computer interaction and interface design

HCI is the study of how people interact with computer systems, aiming to make interfaces effective, efficient and satisfying to use.

Usability principles commonly assessed in IB CS:

- **Learnability** — how quickly can a new user become productive?
- **Efficiency** — how quickly can an experienced user complete tasks?
- **Memorability** — can an occasional user return after a break without relearning everything?
- **Error prevention/recovery** — does the interface stop mistakes before they happen, and help the user recover gracefully when they occur (clear error messages, undo)?
- **Satisfaction** — is the experience pleasant, not merely functional?

User-centred design involves real target users throughout development (needs analysis, prototyping, usability testing, iteration) rather than designing in isolation and testing only at the end.

VN **GIẢI THÍCH TIẾNG VIỆT**

Nguyên tắc HCI: **dễ học** (learnability), **hiệu quả** (efficiency), **dễ nhớ** (memorability), **ngăn ngừa/khắc phục lỗi**, **sự hài lòng**. **Thiết kế lấy người dùng làm trung tâm**: đưa người dùng thật vào suốt quá trình phát triển (phân tích nhu cầu, làm bản mẫu, kiểm thử khả dụng), không chỉ kiểm tra ở bước cuối.

Ví dụ mẫu · Worked example

A banking app redesigns its transfer screen, adding a confirmation dialog ("You are about to send $500 to John. Confirm?") before any transfer completes. Which HCI principle does this primarily address?

This primarily addresses **error prevention**: catching a mistaken amount or recipient *before* an irreversible action (sending money) takes place, rather than only allowing the user to notice and fix the mistake afterwards.

1.7 Reliability, robustness, backup and disaster recovery

(!) Lỗi thường gặp: **Reliability** and **robustness** are frequently confused. **Reliability** means the system consistently performs correctly under the conditions it was designed for (measured, e.g., as mean time between failures). **Robustness** means the system copes *gracefully* with unexpected input or conditions outside normal use (invalid input, a failed network connection, corrupted data) without crashing.

VN GIẢI THÍCH TIẾNG VIỆT

Độ tin cậy (reliability): hệ thống hoạt động đúng trong điều kiện thiết kế bình thường. **Độ bền/độ chịu lỗi (robustness)**: hệ thống vẫn "sống sót", không bị treo/crash khi gặp tình huống bất thường (input sai, mất kết nối mạng). Hai khái niệm này khác nhau và hay bị nhầm lẫn trong đề thi.

Backup strategies protect against data loss: a **full backup** copies everything (slow to create, fast to restore); an **incremental backup** copies only data changed since the *last* backup of any kind (fast to create, but restoring requires the last full backup plus every incremental since); a **differential backup** copies all data changed since the last *full* backup (a middle ground — faster to restore than incremental, slower to create than incremental). The **3-2-1 rule** is a common guideline: keep at least 3 copies of data, on 2 different media types, with 1 copy stored off-site.

A **disaster recovery plan (DRP)** documents how an organisation restores IT services after a major incident (fire, flood, cyberattack, hardware failure). Two key metrics: **Recovery Point Objective (RPO)** — the maximum tolerable amount of *data* loss, measured in time (e.g. an RPO of 1 hour means backups must run at least hourly); **Recovery Time Objective (RTO)** — the maximum tolerable *downtime* before the system must be back online.

Ví dụ mẫu · Worked example

A company backs up its database every night at midnight only (full backups). A server crash occurs at 3 p.m. What is the maximum possible data loss, and how could the backup strategy be improved?

Since the last backup was at midnight and the crash occurred at 3 p.m., up to **15 hours** of data (all transactions between midnight and 3 p.m.) could be lost — this is the company's effective **RPO** under the current strategy. Adding frequent **incremental backups** throughout the day (e.g. every hour) would shrink the RPO to about one hour, at the cost of slightly more complex restoration (apply the full backup, then each incremental in order).

1.8 IT in a social context

Computer science does not exist in a vacuum: systems raise recurring ethical and societal issues that IB CS expects you to discuss with specific, system-linked examples rather than generic statements.

Issue	Description
Privacy	Collection, storage and use of personal data without consent or beyond stated purpose
Security	Protecting data/systems from unauthorised access, theft, or damage
Digital divide	Unequal access to technology/connectivity based on income, geography, age, or disability
Intellectual property	Software piracy, copyright infringement, patenting of algorithms
Algorithmic bias	Systematically unfair outcomes caused by unrepresentative training data or flawed rules
Environmental impact	E-waste, energy/water consumption of data centres, device manufacturing footprint
Employment impact	Automation displacing certain jobs while creating demand for new technical skills

VN **GIẢI THÍCH TIẾNG VIỆT**

Các vấn đề đạo đức–xã hội thường gặp: **quyền riêng tư**, **bảo mật**, **khoảng cách số** (digital divide — chênh lệch tiếp cận công nghệ), **sở hữu trí tuệ** (vi phạm bản quyền phần mềm), **thiên vị thuật toán** (algorithmic bias — kết quả không công bằng do dữ liệu huấn luyện thiếu đại diện), **tác động môi trường** (rác thải điện tử, tiêu thụ năng lượng của trung tâm dữ liệu). Khi trả lời, luôn gắn vấn đề với ví dụ hệ thống cụ thể trong đề bài, không trả lời chung chung.

Ví dụ mẫu · Worked example

A facial-recognition system trained mostly on one demographic performs poorly and unfairly for other groups. Which social/ethical issue does this best illustrate, and which SDLC stage could have mitigated it?

This illustrates **algorithmic bias** — the system produces systematically unfair outcomes because its training data was not representative. It could have been mitigated during **analysis/design** (specifying diverse, representative training data as an explicit requirement) and during **testing** (evaluating accuracy separately across demographic groups rather than relying on one overall accuracy figure).

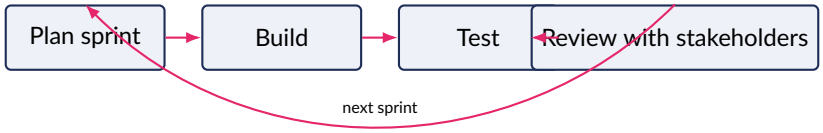
VN **GIẢI THÍCH TIẾNG VIỆT**

Ôn tập nhanh Unit 1: (1) Hệ thống IT = phần cứng + phần mềm + dữ liệu + quy trình + con người; (2) SDLC: phân tích → thiết kế → phát triển → kiểm thử → triển khai → tài liệu hoá → đánh giá; (3) 4 khía cạnh khả thi: kỹ thuật/kinh tế/tiến độ/pháp lý; (4) 3 loại dữ liệu kiểm thử: bình thường/biên/sai; (5) 4 phương pháp chuyển đổi: trực tiếp/song song/giai đoạn/thí điểm — chọn theo mức rủi ro; (6) reliability $\neq$ robustness; (7) full/incremental/differential backup và RPO/RTO.

1.9 Development methodologies: Waterfall and Agile

The **Waterfall model** follows the SDLC stages strictly in sequence — each stage is completed and signed off before the next begins, producing thorough documentation and a predictable schedule, but making it costly to accommodate requirement changes discovered late (going back to an earlier "finished" stage is disruptive and expensive).

Agile methodologies instead deliver working software in short, repeated cycles called **sprints** (typically 1–4 weeks), each producing a small but usable increment of the system. After every sprint the team reviews progress with stakeholders and adapts the next sprint's priorities — welcoming changing requirements even late in development, rather than resisting them. A daily short "stand-up" meeting keeps the team synchronised.



One Agile sprint cycle: a short, repeating loop of planning, building, testing, and stakeholder review.

Aspect	Waterfall	Agile
Requirements	Fixed early, changes costly later	Expected to evolve; revisited each sprint
Delivery	One large release at the end	Frequent small, working increments
Documentation	Extensive, formal, upfront	Lighter-weight, evolves alongside code
Best suited to	Well-understood, stable requirements (e.g. safety-critical/regulated systems)	Projects with uncertain or fast-changing requirements

Ví dụ mẫu · Worked example

A start-up is building a mobile app for a brand-new market where user needs are not yet well understood and are expected to change based on early feedback. Which methodology is more appropriate, and why?

Agile is more appropriate: since user needs are uncertain and likely to change, the team benefits from releasing small increments quickly, gathering real user feedback each sprint, and adjusting direction — a **Waterfall** approach would lock in a detailed requirements specification upfront that is very likely to be wrong or outdated before the single, large final release, wasting significant development effort correcting course only at the very end.

VN **GIẢI THÍCH TIẾNG VIỆT**

Waterfall (mô hình thác nước): các giai đoạn SDLC thực hiện tuần tự, giai đoạn trước xong mới sang giai đoạn sau — tài liệu đầy đủ, tiến độ dễ dự đoán, nhưng khó thay đổi yêu cầu muộn. **Agile**: giao sản phẩm theo từng **sprint** ngắn (1–4 tuần), liên tục nhận phản hồi và điều chỉnh — phù hợp khi yêu cầu chưa rõ ràng hoặc thay đổi nhanh.

1.10 Outsourcing, insourcing and total cost of ownership

Insourcing develops/maintains a system using an organisation's own staff and infrastructure; **outsourcing** contracts an external company to do this instead.

Outsourcing advantages: access to specialist expertise the organisation lacks in-house; potentially lower cost (economies of scale, cheaper labour markets); the organisation can focus on its core business. **Outsourcing disadvantages:** less direct control over quality/timeline; ongoing dependency on the external supplier; potential communication/time-zone barriers; data-security and confidentiality risks in sharing sensitive systems/data externally. **Total cost of ownership (TCO)** sums *all* costs of a system over its useful lifetime, not just the initial purchase/build price: it includes purchase/development cost, licensing, training, ongoing maintenance/support, upgrades, and eventual decommissioning. Comparing options by purchase price alone can be misleading if their ongoing costs differ substantially.

Ví dụ mẫu · Worked example

System A costs \$40,000 upfront with \$6,000/year maintenance. System B costs \$25,000 upfront with \$14,000/year maintenance. Over a 5-year lifetime, which has the lower total cost of ownership?

System A: $40{,}000 + (6{,}000 \times 5) = 40{,}000 + 30{,}000 = 70{,}000$. System B: $25{,}000 + (14{,}000 \times 5) = 25{,}000 + 70{,}000 = 95{,}000$. Despite System B's *lower purchase price*, System A has the lower **total cost of ownership** over 5 years (\$70,000 vs \$95,000) — a decision based on upfront price alone would have chosen the more expensive option in the long run.

VN **GIẢI THÍCH TIẾNG VIỆT**

Outsourcing: thuê công ty bên ngoài phát triển/bảo trì hệ thống (tiếp cận chuyên môn, có thể rẻ hơn, nhưng ít kiểm soát trực tiếp và có rủi ro bảo mật dữ liệu). **Insourcing**: tự làm bằng nhân lực nội bộ. **Tổng chi phí sở hữu (TCO)**: cộng TẤT CẢ chi phí trong suốt vòng đời hệ thống (mua/xây dựng + cấp phép + đào tạo + bảo trì + nâng cấp), không chỉ giá mua ban đầu — so sánh chỉ dựa vào giá ban đầu có thể dẫn đến quyết định sai.

1.11 Legislation: data protection and computer misuse

Most jurisdictions regulate how organisations may collect, store, and use personal data, and criminalise unauthorised access to computer systems.

Data protection principles (common to most data-protection laws): personal data must be collected for a specified, legitimate purpose; kept accurate and up to date; kept no longer than necessary; kept secure against unauthorised access or loss; and processed only with an appropriate legal basis (e.g. consent). **Computer misuse offences** typically criminalise: unauthorised access to a computer system (even without causing damage); unauthorised access with intent to commit a further crime; and unauthorised modification of data/programs (e.g. deploying malware).

Ví dụ mẫu · Worked example

An employee at a company, out of curiosity, uses their valid login credentials to browse a colleague's private HR file, which they have no work-related reason to access. Have they broken computer misuse law, even though they used a legitimate password?

Yes — **unauthorised access** is defined by whether the person was authorised to access *that specific data for that purpose*, not merely whether they possessed valid login credentials for the system in general. Using a legitimate account to view data outside the scope of one's job role is still unauthorised access, and most computer misuse laws would treat this as an offence

regardless of whether any further harm was caused.

VN GIẢI THÍCH TIẾNG VIỆT

Luật bảo vệ dữ liệu: dữ liệu cá nhân phải được thu thập đúng mục đích, chính xác, không lưu quá lâu, bảo mật, và có cơ sở pháp lý (như sự đồng thuận). **Luật chống lạm dụng máy tính**: cấm truy cập trái phép vào hệ thống (kể cả khi dùng mật khẩu hợp lệ nhưng sai mục đích), truy cập trái phép nhằm phạm tội khác, và sửa đổi trái phép dữ liệu/chương trình (như phát tán mã độc).

1.12 Health and safety in IT use

Prolonged computer use carries physical and psychological risks that organisations have a duty to manage through good **ergonomic** design and workplace policy.

Repetitive strain injury (RSI) — pain/damage to muscles, tendons, and nerves from repeated movements (e.g. constant typing/mouse use), mitigated by ergonomic keyboards/mice, wrist rests, and regular breaks. **Eye strain** — discomfort from prolonged screen focus and reduced blinking, mitigated by the "20-20-20" habit (every 20 minutes, look at something 20 feet away for 20 seconds), appropriate screen brightness, and anti-glare measures. **Poor posture/back pain** — from incorrectly adjusted chairs/desks/monitor height, mitigated by adjustable, ergonomically designed furniture and monitor placement at eye level.

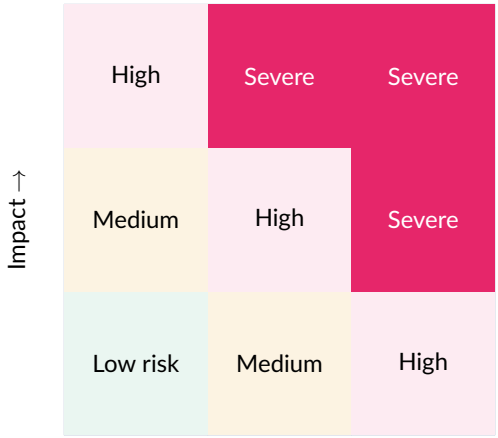
VN GIẢI THÍCH TIẾNG VIỆT

Hội chứng căng cơ lặp lại (RSI): đau/tổn thương cơ, gân, dây thần kinh do chuyển động lặp lại (gõ phím, dùng chuột liên tục) — giảm thiểu bằng bàn phím/chuột công thái học, nghỉ giải lao đều đặn. **Mỏi mắt**: do nhìn màn hình liên tục — áp dụng quy tắc "20-20-20" (mỗi 20 phút, nhìn xa 20 feet trong 20 giây). **Đau lưng/tư thế xấu**: do ghế/bàn/màn hình không điều chỉnh đúng — dùng nội thất công thái học có thể điều chỉnh.

1.13 IT project risk management

Every IT project carries risk — the chance that some event will negatively affect cost, schedule, or quality. **Risk management** is the disciplined process of identifying risks in advance, estimating how serious each is, and planning a response, rather than reacting only after something goes wrong.

A risk is typically scored by combining its estimated **likelihood** (how probable is it?) and **impact** (how severe would the consequences be?). Common risk responses: **avoid** (change the plan to eliminate the risk entirely); **mitigate** (take action to reduce likelihood or impact, e.g. extra testing, redundant hardware); **transfer** (shift the consequence elsewhere, e.g. insurance, outsourcing a risky component to a specialist); **accept** (consciously decide the risk is low enough to tolerate without special action, usually reserved for low-likelihood, low-impact risks).



A simple risk matrix: overall risk severity rises with both likelihood and impact.

Ví dụ mẫu · Worked example

A project team identifies a risk: "the third-party payment gateway API might change its specification before launch, breaking our integration." Suggest one action for each of the four risk responses (avoid, mitigate, transfer, accept) that could apply to this specific risk.

Avoid: choose a different, more stable payment provider with a long history of not making breaking changes, removing the specific risk entirely. **Mitigate:** build the integration using an abstraction layer that isolates payment-gateway-specific code, so a future API change only requires updating one small module rather than the whole system. **Transfer:** choose a payment provider whose contract guarantees compensation/support if their API change causes the client's system to fail. **Accept:** if the provider has an excellent track record of API stability and the cost of any of the above actions outweighs the (low) likelihood of a breaking change, the team may simply monitor the provider's changelog and accept the residual risk.

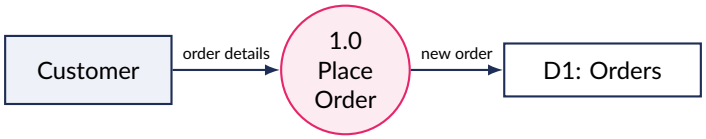
VN **GIẢI THÍCH TIẾNG VIỆT**

Quản lý rủi ro dự án IT: xác định rủi ro trước, ước lượng theo **khả năng xảy ra** và **mức độ ảnh hưởng**, rồi lập kế hoạch ứng phó. Bốn chiến lược ứng phó: **tránh** (loại bỏ hoàn toàn nguyên nhân rủi ro), **giảm thiểu** (hành động để giảm khả năng/ảnh hưởng), **chuyển giao** (chuyển hậu quả sang bên khác, ví dụ bảo hiểm, thuê ngoài), **chấp nhận** (rủi ro đủ thấp để không cần hành động đặc biệt).

1.14 Systems analysis diagrams: data flow and use case

Analysts use standardised diagrams to communicate requirements precisely between stakeholders, analysts, and developers, reducing the ambiguity of plain prose descriptions.

A **data flow diagram (DFD)** models how data moves through a system: **external entities** (people/systems outside the system boundary that supply or receive data), **processes** (transform incoming data into outgoing data), **data stores** (where data is held, e.g. a file or database table), and **data flows** (arrows showing the direction data moves). A **use case diagram** instead models *who* (an **actor**) can do *what* (a **use case**) with a system, useful for capturing functional requirements from a user's perspective early in analysis.



A simple level-1 data flow diagram fragment: an external entity supplies data to a process, which writes to a data store.

Ví dụ mẫu · Worked example

A library wants a system where a "Member" can search the catalogue and borrow books, while a "Librarian" can additionally add new books and manage fines. Sketch (in words) the use cases and actors involved.

Two **actors**: `Member` and `Librarian`. **Use cases** connected to `Member`: "Search Catalogue," "Borrow Book." **Use cases** connected to `Librarian`: "Add New Book," "Manage Fines," and (since a librarian can do everything a member can, plus more) potentially also "Search Catalogue" and "Borrow Book" on behalf of a member. This diagram communicates, at a glance, exactly which functionality each type of user requires — valuable for scoping the system's requirements before any design work begins.

VN **GIẢI THÍCH TIẾNG VIỆT**

Sơ đồ luồng dữ liệu (DFD): mô hình hoá dữ liệu di chuyển qua hệ thống — thực thể ngoài (external entity), tiến trình (process, biến đổi dữ liệu), kho dữ liệu (data store), và luồng dữ liệu (mũi tên). **Sơ đồ use case**: mô hình hoá AI (actor) có thể làm GÌ (use case) với hệ thống — hữu ích để nắm bắt yêu cầu chức năng từ góc nhìn người dùng ngay từ giai đoạn phân tích.

1.15 Prototyping

Prototyping builds an early, partial, working model of a system to clarify requirements and gather feedback before committing to full-scale development.

Throwaway prototyping builds a rough, disposable model (e.g. clickable screen mockups) purely to clarify requirements and gather feedback — the prototype itself is then discarded, and the real system is built separately from scratch, informed by what was learned. **Evolutionary prototyping** instead starts with a basic working version and progressively refines/extends the *same* codebase over time until it becomes the final, complete system — closely related in spirit to Agile's iterative delivery. The main advantage of prototyping in general is catching misunderstood or missing requirements *early*, when they are cheap to fix, rather than after full development is already complete.

Ví dụ mẫu · Worked example

A client struggles to describe exactly how they want a new booking system's calendar screen to look and behave. Explain how throwaway prototyping could help before committing to full development.

A developer could quickly build a rough, non-functional (or minimally functional) mockup of the calendar screen — perhaps just clickable images showing different possible layouts — and show it to the client for feedback within days rather than weeks. The client, seeing something concrete rather than a written description, can much more easily say "actually I'd prefer the week view by default" or "I need a way to filter by staff member here" — catching such requirements *before* the real, fully-functional system is built, avoiding a costly rebuild later. The mockup itself is then discarded once requirements are clarified.

VN GIẢI THÍCH TIẾNG VIỆT

Làm bản mẫu (prototyping): xây dựng mô hình sơ khai, hoạt động một phần để làm rõ yêu cầu và thu thập phản hồi trước khi phát triển đầy đủ. **Bản mẫu dùng một lần (throwaway)**: xây rồi bỏ, chỉ để làm rõ yêu cầu. **Bản mẫu tiến hoá (evolutionary)**: phát triển dần trên CÙNG một bản mẫu cho đến khi trở thành hệ thống hoàn chỉnh — gần giống tinh thần Agile. Lợi ích chính: phát hiện yêu cầu còn thiếu/hiểu sai SỚM, khi chi phí sửa còn thấp.

1.16 Version control and configuration management

As software projects grow and involve multiple developers, tracking exactly what changed, when, and by whom becomes essential to avoid chaos.

A **version control system (VCS)** records a history of changes to a project's files over time, allowing developers to review past versions, understand who changed what and why (via commit messages), and safely revert an unwanted change. A **repository** stores this full history; a **commit** is a saved snapshot of changes at a point in time. **Branching** lets a developer work on a new feature or fix in an isolated copy of the project without affecting the main, stable version; **merging** later combines that branch's changes back into the main line once complete and tested. When two developers have changed the *same* part of a file differently, a **merge conflict** occurs, requiring a human decision about which change (or what combination) should be kept.

Ví dụ mẫu · Worked example

Two developers, working on separate branches, both independently modify the same line of a shared configuration file in different ways. Explain what happens when they try to merge both branches back into the main project, and why this cannot always be resolved automatically.

When both branches are merged, the version control system detects that the very same line was changed differently in each branch — a **merge conflict**. The system cannot automatically decide which developer's change is "correct" (both are valid, deliberate edits, just incompatible with each other), so it flags the conflict and requires a human developer to manually review both versions and decide how to resolve it: keep one change, keep the other, or manually combine them into a new version that satisfies both intentions. This human judgment is necessary precisely because the conflict reflects a genuine difference in intended behaviour, not a technical error the software could resolve algorithmically.

VN GIẢI THÍCH TIẾNG VIỆT

Hệ thống quản lý phiên bản (VCS): ghi lại lịch sử thay đổi tệp dự án theo thời gian — **repository** (kho lưu trữ) chứa toàn bộ lịch sử, **commit** là một bản lưu tại một thời điểm. **Nhánh (branch)**: làm việc trên bản sao riêng biệt mà không ảnh hưởng bản chính; **gộp (merge)**: kết hợp thay đổi từ nhánh vào bản chính. **Xung đột gộp (merge conflict)**: xảy ra khi hai người sửa CÙNG một chỗ khác nhau — cần con người quyết định cách giải quyết.

1.17 Practice

Bài tập luyện tập

Explain one advantage of using a version control system over simply emailing zipped copies of a project's source code back and forth between developers.

Lời giải chi tiết

A version control system maintains one authoritative, continuously-updated history of every change with clear attribution (who changed what, when, and typically why via a commit message), and provides built-in tools for safely combining (merging) different developers' simultaneous changes and identifying/resolving genuine conflicts between them. Emailing zipped copies back and forth provides no such structured history, makes it very easy to accidentally overwrite a colleague's recent changes, and gives no reliable way to see exactly what changed between two versions or to revert cleanly to an earlier one.

Bài tập luyện tập

Explain why "branching" before starting work on a risky new feature is good practice, even if the feature is ultimately abandoned.

Lời giải chi tiết

Working on a separate **branch** isolates all experimental changes from the main, stable version of the project — if the risky feature is abandoned, the branch can simply be discarded without ever having affected the main line at all, and the stable version remains completely unaffected throughout the experiment. Without branching, risky, unfinished changes made directly to the main project could break it for every other developer relying on that main version, even if the experiment is later abandoned.

Bài tập luyện tập

A company's economic feasibility study estimates a new CRM system will cost \$80,000 to build and save \$25,000 per year thereafter. A rival system costs \$50,000 to build but only saves \$12,000 per year. Calculate the payback period for each and state which is preferable if the system will only be used for 3 years total.

Lời giải chi tiết

System 1 payback $= \dfrac{80{,}000}{25{,}000} = 3.2$ years. System 2 payback $= \dfrac{50{,}000}{12{,}000} \approx 4.17$ years. Over a 3-year total lifetime, **neither system fully pays for itself** — but System 1 recovers more of its cost within 3 years ($25{,}000 \times 3 = 75{,}000$ recovered out of $80{,}000$, a shortfall of only $5{,}000$) compared with System 2 ($12{,}000 \times 3 = 36{,}000$ recovered out of $50{,}000$, a shortfall of $14{,}000$) — System 1 is the less financially unfavourable option for this specific 3-year timeframe, illustrating why payback period alone should always be considered alongside the system's actual expected lifetime.

Bài tập luyện tập

Distinguish between "verification" (Unit 8's usage, checking a model was built to specification) and "validation" as used in a general software testing context, applied to a temperature-conversion function.

Lời giải chi tiết

Verification would check that the temperature-conversion function was implemented exactly as its design specification stated (e.g. if the spec required the formula $F = C \times 9/5 + 32$, verification confirms the code actually implements this precise formula, matching the design

regardless of whether the design itself is the most useful one). **Validation** would instead check whether the function's real-world outputs are actually correct/useful (e.g. testing that converting $0°$C genuinely returns $32°$F, and more broadly whether the tool meets the actual needs of whoever will use it) — a function could pass verification (correctly implementing a flawed specification) while still failing validation if the specification itself was wrong.

Bài tập luyện tập

A school's SDLC feasibility study for a new library system reports: "the school can afford the software licence and the librarians can be trained within the existing timetable." This is primarily assessing:

(A) Technical feasibility
(B) Economic and schedule feasibility
(C) Legal feasibility
(D) Operational testing

Lời giải chi tiết

Affordability is **economic feasibility**; fitting training into the existing timetable is **schedule feasibility**. Neither statement concerns whether the technology can physically be built (technical) or whether it is legally permitted. **(B)**.

Bài tập luyện tập

A small clinic switches from paper records to a new digital system *overnight*, deleting the paper files immediately to save storage space. One week later, a software bug corrupts several patient files with no backup available. Identify the changeover mistake and propose a better method.

Lời giải chi tiết

The clinic used **direct changeover** and additionally destroyed the fallback (paper records) prematurely — both increase risk. A better strategy is **parallel running** (keep the paper system live alongside the digital one for several weeks) combined with **regular backups** before phasing out paper records, so recovery is possible if the new system fails.

Bài tập luyện tập

Which test data category is essential to include when testing an input field that only accepts ages from 0 to 120?

(A) Only normal values (e.g. 30)
(B) Only erroneous values (e.g. "abc")
(C) Normal, boundary (0, 120, -1, 121) and erroneous data
(D) Only values supplied by the client

Lời giải chi tiết

Thorough testing requires **normal** data (typical valid input), **boundary** data (values at and just outside the limits), and **erroneous** data (invalid type, e.g. text) to confirm the system rejects it gracefully. **(C)**.

Bài tập luyện tập

Distinguish between **unit testing** and **integration testing**, giving an example of each for an online shopping system.

Lời giải chi tiết

Unit testing checks one isolated module, e.g. testing only the function that calculates sales tax on a single item. **Integration testing** checks that multiple modules work correctly *together*, e.g. verifying that the shopping-cart module correctly passes the final total to the payment module and that the payment confirmation correctly updates the inventory module.

Bài tập luyện tập

A company wants to know exactly how staff currently process paper invoices, including steps staff might not think to mention. Which requirements-gathering method is most appropriate, and what is its main drawback?

(A) Questionnaire; low response detail

(B) Direct observation; people may alter behaviour when watched

(C) Document review only; may be outdated

(D) Interview only; time-consuming for large teams

Lời giải chi tiết

Direct observation reveals the real, complete process (including undocumented workarounds staff may forget to mention in an interview), but its main drawback is the **Hawthorne effect** — people may work more carefully or differently simply because they know they are being watched. **(B)**.

Bài tập luyện tập

Explain, using an example, the difference between **black-box** and **white-box** testing.

Lời giải chi tiết

Black-box testing treats the program as a sealed unit: the tester enters inputs and checks that the outputs match expectations, without examining the code (e.g. entering an age of 150 and checking that an error message appears). **White-box testing** requires access to the source code, designing test cases so that every branch/path of the logic is executed at least once (e.g. writing a specific input to force the `else` branch of an `if` statement to run, to confirm it behaves correctly).

Bài tập luyện tập

Which changeover method contains risk to a small area of an organisation before a wider rollout?

(A) Direct

(B) Parallel

(C) Phased

(D) Pilot

Lời giải chi tiết

Pilot changeover trials the new system at one site, branch, or department first, limiting the impact of any problems before a full rollout. **(D)**.

Bài tập luyện tập

A retail chain rolls out a new point-of-sale system to one store at a time over six months, learning from each store before moving to the next. Name this changeover method and give one advantage over a fully direct changeover across all stores simultaneously.

Lời giải chi tiết

This is **phased changeover** (rolled out module-by-module/store-by-store). One advantage over a full direct changeover: problems discovered in early stores can be fixed before affecting the remaining stores, so the overall risk of a chain-wide failure is much lower, even though the total rollout takes longer.

Bài tập luyện tập

Explain the difference between technical documentation and user documentation, with one example of each.

Lời giải chi tiết

Technical documentation is written for developers/maintainers and includes system design diagrams, data dictionaries, and commented source code (e.g. an entity-relationship diagram of the database). **User documentation** is written for end users and includes manuals, quick-start guides, and FAQs (e.g. a one-page guide showing how to reset a forgotten password).

Bài tập luyện tập

On a project Gantt chart, "Testing" cannot start until "Development" finishes, and any delay in either task delays the whole project. What term describes the sequence Analysis → Design → Development → Testing → Deployment if every task depends directly on the one before it?

(A) Slack path
(B) Critical path
(C) Parallel path
(D) Iteration path

Lời giải chi tiết

A sequence of tasks where every task depends on the previous one, and any delay pushes back the whole project finish date, is by definition the **critical path**. **(B)**.

Bài tập luyện tập

A project has two independent task chains: Chain A takes 8 weeks; Chain B takes 5 weeks; both must finish before the final "Deploy" task can start. What is Chain B's **slack** (float), and which chain is on the critical path?

Lời giải chi tiết

The project cannot deploy until *both* chains finish, so the overall timeline is set by the longer chain, Chain A (8 weeks) — this is the **critical path**. Chain B finishes in 5 weeks but does not need to, since Deploy waits for Chain A anyway; Chain B therefore has **3 weeks of slack** $(8 - 5 = 3)$ — it could be delayed by up to 3 weeks without affecting the overall project finish date.

Bài tập luyện tập

A food-delivery app's interface lets an experienced user reorder a previous meal with one tap, while a first-time user is guided step-by-step through menu browsing. Which two HCI usability principles are being addressed for the two respective user types?

(A) Learnability (new user) and efficiency (experienced user)

(B) Memorability (new user) and error prevention (experienced user)

(C) Efficiency (new user) and learnability (experienced user)

(D) Satisfaction only, for both

Lời giải chi tiết

Step-by-step guidance for a first-time user targets **learnability** (getting a new user productive quickly); a one-tap shortcut for a returning user targets **efficiency** (letting an experienced user complete a task as fast as possible). **(A)**.

Bài tập luyện tập

Explain why "user-centred design" typically produces a more usable interface than a design created entirely by engineers without user input.

Lời giải chi tiết

User-centred design involves real target users throughout the process — in needs analysis (understanding what users actually need, not what developers assume), in prototyping (getting early feedback before major investment), and in usability testing (catching confusing or inefficient design choices before release). An engineer-only design risks embedding assumptions about how users think that do not match how real users actually behave, leading to a technically correct but frustrating or inefficient interface that only surfaces problems after release, when they are far more expensive to fix.

Bài tập luyện tập

A weather-monitoring station is advertised as "reliable." A user later complains that when a sensor cable is accidentally unplugged, the whole system crashes and must be manually restarted. Which quality is actually missing, and how does it differ from reliability?

(A) Robustness; the system fails to handle an unexpected condition gracefully

(B) Reliability; the system fails to handle an unexpected condition gracefully

(C) Efficiency; the system is too slow

(D) Portability; the system cannot run on other hardware

Lời giải chi tiết

Robustness is missing: the system cannot cope gracefully with an unexpected condition (a disconnected sensor) and crashes instead of, e.g., logging an error and continuing with the remaining sensors. **Reliability** concerns consistent correct behaviour under normal, expected operating conditions — a different property from handling abnormal conditions gracefully. **(A)**.

Bài tập luyện tập

A company performs a full backup every Sunday at midnight and an **incremental** backup every other night at midnight. The server fails on Thursday at 6 a.m. List, in order, which backups must be restored, and estimate the maximum possible data loss.

Lời giải chi tiết

Restoration order: the **Sunday full backup** first, then every **incremental** backup in chronological order (Monday, Tuesday, Wednesday nights) up to the most recent one before the failure. Since the last backup ran at midnight Wednesday/Thursday and the failure occurred at 6 a.m. Thursday, the maximum possible data loss is the transactions made between midnight and 6 a.m. Thursday — approximately **6 hours** of data.

Bài tập luyện tập

Explain one advantage and one disadvantage of **incremental** backups compared with **differential** backups.

Lời giải chi tiết

Advantage of incremental: each individual backup is faster to create and smaller, since it only stores changes since the very last backup (of any type). **Disadvantage of incremental:** restoring is slower and more complex, because you must apply the last full backup *and every* incremental backup since, in the correct order — if any one incremental in the chain is corrupted or missing, the restore fails. (A differential backup instead grows larger each day but only ever needs the last full backup plus the single most recent differential to restore.)

Bài tập luyện tập

A hospital's disaster recovery plan states an RPO of 15 minutes and an RTO of 1 hour. Explain what each of these figures commits the hospital to.

Lời giải chi tiết

An **RPO (Recovery Point Objective) of 15 minutes** means the hospital can tolerate losing at most 15 minutes' worth of data if a failure occurs — implying backups/replication must run at least every 15 minutes. An **RTO (Recovery Time Objective) of 1 hour** means that, however the failure happens, IT services must be fully restored and operational again within 1 hour of the incident — this drives investment in fast failover systems, spare hardware, or cloud backup infrastructure.

Bài tập luyện tập

A ride-sharing app's pricing algorithm is later found to charge systematically higher fares in low-income neighbourhoods, because its training data came mostly from wealthier areas. Which issue does this illustrate, and suggest one way the SDLC could have caught it earlier.

Lời giải chi tiết

This illustrates **algorithmic bias** caused by unrepresentative training data. Testing could have caught it earlier by evaluating the algorithm's output *separately* across different neighbourhood/income groups during the **testing** stage, rather than checking only an overall average fare accuracy — a group-level accuracy check would have exposed the systematic disparity before deployment.

Bài tập luyện tập

A rural community has slow, expensive internet access compared with a nearby city. Students in the rural community struggle to complete online coursework as a result. Which social issue does this best illustrate?

(A) Digital divide
(B) Intellectual property infringement
(C) Algorithmic bias
(D) Robustness failure

Lời giải chi tiết

Unequal access to technology/connectivity based on geography is the definition of the **digital divide**. **(A)**.

Bài tập luyện tập

A large data centre operator advertises "carbon neutral by 2030." List two concrete environmental costs of data centre operation that such a pledge would need to address.

Lời giải chi tiết

Two concrete costs: (1) **energy consumption** — servers and, critically, cooling systems draw enormous amounts of electricity continuously, and unless that electricity comes from renewable sources it produces significant carbon emissions; (2) **hardware manufacturing and e-waste** — servers and storage devices are replaced every few years, consuming raw materials (rare metals) to manufacture and generating electronic waste that is difficult to recycle safely. (Water consumption for cooling is also an acceptable answer.)

Bài tập luyện tập

A company distributes free software but embeds code that secretly copies user files to a remote server without consent. Identify *two* distinct ethical/legal issues raised by this scenario.

Lời giải chi tiết

Two distinct issues: (1) **privacy** — personal files are being collected without the user's knowledge or consent, beyond any stated purpose of the software; (2) **security** — the unauthorised transfer of files represents a security breach (data exfiltration), and depending on jurisdiction

may also breach **data protection law**, a separate legal-feasibility concern.

Bài tập luyện tập

Explain why "evaluation," as the final SDLC stage, should be measured against the *original* requirements specification rather than against the developers' own opinion of the finished system.

Lời giải chi tiết

The purpose of evaluation is to judge whether the system actually solves the problem it was commissioned to solve. If developers judge success only by their own standards, they may (consciously or not) evaluate the system against what they *built* rather than what was *needed* — a system can be well engineered yet still fail to meet the client's real requirements (e.g. missing a feature a stakeholder considered essential). Comparing against the original requirements specification, ideally with structured stakeholder feedback, keeps the evaluation objective and client-focused.

Bài tập luyện tập

A university deploys a new online enrolment system directly (no parallel running) the week before term starts, without informing academic advisors of the change. On day one thousands of students cannot enrol due to an unanticipated server load, and there is no fallback. List two separate SDLC failures illustrated here.

Lời giải chi tiết

Two separate failures: (1) an inappropriate **changeover method** — **direct changeover** for a high-stakes, high-traffic system with no fallback was excessively risky; parallel running or a phased/pilot rollout (e.g. one faculty first) would have contained the damage. (2) Inadequate **testing**, specifically missing **system/load testing** — the server load at real scale (thousands of simultaneous students) was evidently never tested before go-live, a stress condition that should have been part of the test plan alongside normal/boundary/erroneous data testing.

Bài tập luyện tập

Differentiate between **validation** and the general SDLC **testing** process using an example of a student-ID input field limited to exactly 8 digits.

Lời giải chi tiết

Validation is an automatic check built into the software itself, applied to every piece of data entered — e.g. a rule that rejects any student ID that is not exactly 8 digits long, catching format errors as data is typed. Broader **testing** is the wider development-stage process of confirming the whole system works as intended, of which checking that this validation rule itself works correctly (using boundary/erroneous test data such as a 7-digit or 9-digit ID) is just one small part. In short: validation runs automatically *within* the live system; testing is the development activity used to confirm validation (and everything else) is correctly implemented.

Bài tập luyện tập

A company's economic feasibility study shows the new system will cost \$120,000 to build and save \$30,000 per year in reduced labour costs. After how many full years does the system pay for itself (ignoring any figures beyond simple payback)?

Lời giải chi tiết

Payback period $= \dfrac{\text{Cost}}{\text{Annual saving}} = \dfrac{120\,000}{30\,000} = 4$ years. The system pays for itself after **4 full years** of operation, after which the annual \$30,000 saving becomes net benefit.

Bài tập luyện tập

Explain why "legal feasibility" might make an otherwise technically and economically attractive system unfeasible, using data protection law as an example.

Lời giải chi tiết

A system might be entirely achievable with current technology (technically feasible) and profitable to run (economically feasible), yet still be unfeasible if it would require collecting, storing, or transferring personal data in ways that violate data protection law (e.g. storing citizens' health data on servers outside a jurisdiction that legally requires such data to remain within national borders). In that case, the organisation cannot lawfully deploy the system as designed regardless of its technical or financial merits, so the design must be revised (or the project abandoned) purely on legal grounds.

Bài tập luyện tập

A new inventory system is planned with a critical path of 20 weeks (Analysis → Design → Development → Testing → Deployment, each depending on the last). Development is delayed by 3 weeks due to a staffing shortage. What is the effect on the overall project, and why?

Lời giải chi tiết

Because **Development** lies on the **critical path** (every subsequent task directly depends on it finishing), a 3-week delay in Development pushes back Testing, then Deployment, by the same 3 weeks — the whole project finish date slips from 20 weeks to **23 weeks**, with no slack available to absorb the delay. (Had the delay instead occurred in a task with slack/float not on the critical path, the overall finish date could have been unaffected.)

Bài tập luyện tập

A government agency is building a system to calculate legally mandated tax payments, where requirements are fixed by law and unlikely to change during development, and full audit documentation is legally required at every stage. Which methodology, Waterfall or Agile, is more appropriate, and why?

Lời giải chi tiết

Waterfall is more appropriate here: the requirements are fixed and well understood (set by law), so there is little benefit from Agile's iterative re-planning, while Waterfall's emphasis on

thorough, sequential documentation at every stage directly supports the legal requirement for a complete audit trail. Agile's strength — adapting quickly to changing/uncertain requirements — is not needed when the requirements are stable and externally fixed.

Bài tập luyện tập

Explain one risk of outsourcing the development of a hospital's confidential patient-record system to an external company, beyond simple cost.

Lời giải chi tiết

One significant risk is **data confidentiality and security**: sharing sensitive patient data (or system access) with an external company widens the number of people/organisations who could potentially mishandle or leak it, and the hospital has less direct day-to-day control over the external company's security practices, staff vetting, and internal safeguards than it would over its own staff — a serious concern given the sensitivity of medical records and the legal obligations (data protection law) attached to them.

Bài tập luyện tập

Explain the difference between a data store and a data flow in a data flow diagram (DFD), giving an example of each for an online food-ordering system.

Lời giải chi tiết

A **data flow** is the movement of data between components — represented by an arrow — such as a customer's order details travelling from the "Customer" external entity to the "Process Order" process. A **data store** is a place where data is held/persisted, such as an "Orders" database table that the "Process Order" process writes the confirmed order into for later retrieval — a data store does not move data anywhere itself, it simply holds it until some process reads from or writes to it.

Bài tập luyện tập

A ride-sharing app has two actors: "Rider" (can request a ride, rate a driver) and "Driver" (can accept a ride, mark a trip complete). Sketch, in words, the use cases connected to each actor in a use case diagram.

Lời giải chi tiết

Rider actor connects to use cases: "Request Ride" and "Rate Driver." **Driver** actor connects to use cases: "Accept Ride" and "Mark Trip Complete." Each actor is linked only to the use cases relevant to their own role, giving stakeholders an immediate, unambiguous visual summary of exactly what functionality each type of user needs the system to support — useful for confirming with the client that no required functionality has been missed before design begins.

Bài tập luyện tập

Explain one advantage of evolutionary prototyping over throwaway prototyping, and one situation where throwaway prototyping would be preferred instead.

Lời giải chi tiết

Advantage of evolutionary prototyping: since the same codebase is refined into the final product, no development effort building the prototype is wasted — every improvement made while gathering feedback directly contributes to the finished system, unlike a throwaway prototype which is discarded entirely once it has served its feedback-gathering purpose. **Situation favouring throwaway prototyping instead:** when requirements are so unclear or the interface design so uncertain that the team expects to attempt and reject several very different approaches quickly (e.g. testing three completely different navigation layouts with users) — building each as quick, disposable, low-effort mockups is far faster than trying to evolve a single "real" codebase through several complete redesigns.

Bài tập luyện tập

Explain why a client reviewing a rough clickable prototype is more likely to identify a missing requirement than a client reviewing only a written specification document.

Lời giải chi tiết

A written specification requires the client to mentally imagine how the finished system will actually look, feel, and behave — many people struggle to notice a missing feature or an awkward workflow purely from abstract prose, especially non-technical clients unfamiliar with reading formal specifications. A **prototype** gives the client something concrete to directly interact with (even if rough or partially functional), closely mimicking the real experience of using the finished system — this makes it far easier to notice, in the moment, "wait, how would I actually do X here?" or "I expected a button here," surfacing missing or misunderstood requirements much earlier and more reliably than reading a document alone.

Bài tập luyện tập

Two software licensing options are compared over a 4-year period. Option X: \$15,000 upfront, \$2,000/year support. Option Y: \$5,000 upfront, \$6,000/year support. Calculate the total cost of ownership for each over 4 years and state which is cheaper overall.

Lời giải chi tiết

Option X: $15{,}000 + (2{,}000 \times 4) = 15{,}000 + 8{,}000 = 23{,}000$. Option Y: $5{,}000 + (6{,}000 \times 4) = 5{,}000 + 24{,}000 = 29{,}000$. Option X has the lower **total cost of ownership** (\$23,000 vs \$29,000) over 4 years, despite its much higher upfront price — comparing only the upfront cost would have wrongly favoured Option Y.

Bài tập luyện tập

An employee uses their own valid work login to read a colleague's personal medical file stored on the company system, purely out of personal curiosity and with no work justification. Explain why this is a computer misuse offence despite the employee having a legitimate password.

Lời giải chi tiết

Authorisation under computer misuse law is defined by whether a person is permitted to access *that specific data for that specific purpose*, not merely whether they hold valid credentials for the system in general. Since the employee had no legitimate work reason to view that

particular file, their access was **unauthorised** in substance even though their login itself was valid — this is a textbook example of exceeding authorised access, which most computer misuse legislation explicitly criminalises regardless of whether the data was subsequently misused further.

Bài tập luyện tập

A call-centre worker develops persistent wrist pain after months of continuous keyboard and mouse use with no breaks. Name this condition and suggest two concrete workplace changes to reduce the risk for other staff.

Lời giải chi tiết

This is **repetitive strain injury (RSI)**. Two concrete changes: (1) provide **ergonomic equipment** — a wrist rest and an ergonomically shaped keyboard/mouse that reduce awkward joint angles during repetitive movements; (2) enforce **regular scheduled breaks** (e.g. a short break every hour) so that muscles and tendons are not subjected to continuous repetitive strain for extended, uninterrupted periods.

Bài tập luyện tập

Explain why the ”20-20-20 rule” is recommended for office workers who spend most of the day looking at a screen, and identify which health issue it targets.

Lời giải chi tiết

The 20-20-20 rule (every 20 minutes, look at something roughly 20 feet away for at least 20 seconds) targets **eye strain**. Continuously focusing on a nearby screen keeps the eye’s focusing muscles tensed and reduces the natural blink rate (leading to dryness); periodically refocusing on a distant object relaxes those muscles and gives the eyes a brief natural break, reducing the cumulative strain of a long screen-focused work session.

Bài tập luyện tập

A software company currently uses Waterfall for all projects. A new client project has vague, likely-to-change requirements because it targets an emerging market the client does not yet fully understand. Explain one specific risk of using Waterfall for this particular project.

Lời giải chi tiết

Under **Waterfall**, requirements are gathered and effectively ”locked in” during the early analysis/design stages, with development proceeding sequentially from that fixed specification. If the client’s understanding of the market (and therefore the real requirements) changes significantly after this point — which is likely here, given the requirements are described as vague and evolving — the team risks building an entire, fully-documented system based on an early specification that no longer matches what is actually needed, requiring a costly restart or major rework rather than a small, cheap course-correction, which an Agile approach would have allowed much earlier.

Bài tập luyện tập

A hospital IT project identifies a risk with **low likelihood** but **catastrophic impact**: a total power failure during surgery-critical system operation. Which risk response is most appropriate, and why is "accept" clearly inappropriate here despite the low likelihood?

Lời giải chi tiết

Mitigate (or a combination of mitigate and transfer) is most appropriate: installing an uninterruptible power supply (UPS) and backup generators directly reduces the impact of a power failure even if it does occur. "Accept" is inappropriate here because acceptance is only suitable when the *combination* of likelihood and impact is low enough to tolerate — even though likelihood is low, the impact (potential loss of life during surgery) is so catastrophic that the overall risk remains unacceptable, and the cost of mitigation (backup power) is easily justified against that severity.

Bài tập luyện tập

Explain the difference between "transferring" a risk and "mitigating" a risk, using the example of a company's risk that its cloud storage provider might suffer a data breach.

Lời giải chi tiết

Mitigating this risk means the company takes direct action to reduce the likelihood or impact of the breach itself — e.g. requiring the provider to use strong encryption, or maintaining an independent backup so a breach at the provider does not mean permanent data loss. **Transferring** this risk means shifting the financial/legal consequence to another party without necessarily reducing the chance of the breach happening — e.g. purchasing cyber-insurance so that, if a breach does occur, the financial cost of the resulting damages or fines is covered by the insurer rather than falling entirely on the company.

IB DIPLOMA PROGRAMME • Unit 2

Computer Organization

Mục tiêu Unit: Kiến trúc CPU (thanh ghi, ALU, CU, bus), chu trình fetch–decode–execute, phân cấp bộ nhớ & lưu trữ thứ cấp, mạch logic & đại số Boole, và biểu diễn dữ liệu (nhị phân, hex, bù hai, dấu phẩy động, mã hoá ký tự/ảnh/âm thanh, nén dữ liệu).

2.1 CPU architecture and registers

The **Central Processing Unit (CPU)** follows the classic **von Neumann architecture**: a single memory space stores both instructions and data, fetched one at a time over a shared **bus**. The CPU contains the **Control Unit (CU)**, which sequences and coordinates instruction execution, and the **Arithmetic Logic Unit (ALU)**, which performs arithmetic and logical operations.

Key registers. **PC** (Program Counter) — holds the address of the *next* instruction to fetch. **MAR** (Memory Address Register) — holds the address currently being accessed in memory. **MDR** (Memory Data Register) — holds data/instructions being transferred to or from memory. **CIR** (Current Instruction Register) — holds the instruction currently being decoded/executed. **ACC** (Accumulator) — stores the results of ALU operations. **Buses:** the **address bus** carries memory addresses (one-directional, CPU → memory); the **data bus** carries the actual data/instructions (bidirectional); the **control bus** carries control signals (e.g. read/write, clock, interrupt signals).

Simplified CPU architecture: the CU sequences fetch–decode–execute; the ALU performs arithmetic/logic; registers hold operands and addresses; buses connect the CPU to memory.

VN **GIẢI THÍCH TIẾNG VIỆT**

CPU gồm **CU** (điều khiển, giải mã lệnh) và **ALU** (tính toán số học/logic). Các thanh ghi quan trọng: **PC** (địa chỉ lệnh kế tiếp), **MAR**/**MDR** (địa chỉ/dữ liệu đang truy cập bộ nhớ), **CIR** (lệnh hiện tại), **ACC** (kết quả ALU). Ba loại bus: **địa chỉ** (một chiều), **dữ liệu** (hai chiều), **điều khiển** (tín hiệu đồng bộ, ngắt).

The fetch–decode–execute cycle:

1. **Fetch:** the address in PC is copied to MAR → the instruction at that memory address is copied into MDR → then into CIR → PC is incremented (so it already points to the *next* instruction).
2. **Decode:** the CU interprets the opcode held in CIR, identifying the operation and any operands.
3. **Execute:** the instruction is carried out — e.g. the ALU computes a result, which may be written back to a register or to memory.

This cycle repeats continuously while the CPU is powered, driven by the **system clock**. **Clock speed** (measured in GHz) is the number of cycles per second; higher clock speed generally means faster execution, but is not the only factor — the number of cores, cache size, and **pipelining** (overlapping the fetch/decode/execute of successive instructions) also matter.

Ví dụ mẫu · Worked example

Place the following steps in the correct order of the fetch–execute cycle: (i) CU decodes the opcode in CIR; (ii) PC is incremented; (iii) instruction is copied from memory into MDR then CIR; (iv) instruction is carried out (possibly using the ALU).

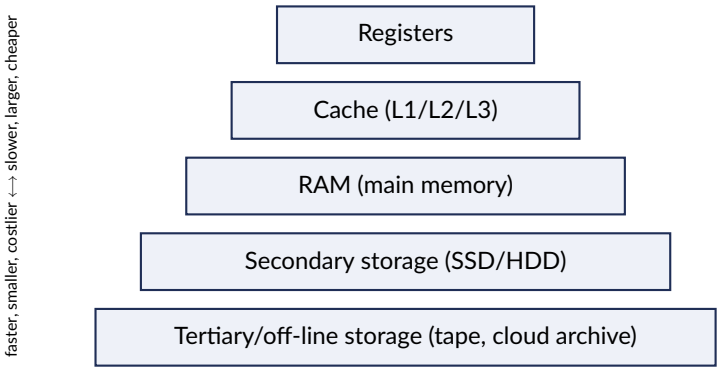
Correct order: (iii) fetch the instruction into MDR/CIR → (ii) PC incremented → (i) decode → (iv) execute. In short: **Fetch (iii, ii) → Decode (i) → Execute (iv)**. Incrementing PC *during* the fetch stage (not after execute) is essential, since a jump/branch instruction executed later may need to *overwrite* PC — if PC were incremented after execute, a branch's new target address would be wrongly overwritten.

VN **GIẢI THÍCH TIẾNG VIỆT**

Chu trình **fetch–decode–execute**: **Fetch** (lấy lệnh từ bộ nhớ theo địa chỉ trong PC, đưa vào CIR, rồi tăng PC lên 1 ngay lập tức); **Decode** (CU giải mã lệnh); **Execute** (thực thi lệnh, có thể dùng ALU). **Tốc độ xung nhịp (clock speed)** đo bằng GHz — nhanh hơn thường nghĩa là xử lý nhanh hơn, nhưng số lõi (core), bộ nhớ đệm (cache) và kỹ thuật **pipelining** (chồng lấn các giai đoạn của nhiều lệnh liên tiếp) cũng ảnh hưởng lớn.

2.2 Memory hierarchy and storage

Memory is organised in a hierarchy that trades **speed** against **cost per byte** and **capacity**.



The memory hierarchy: each level down trades speed for capacity and lower cost per byte.

RAM (Random Access Memory) is **volatile** working memory that loses its contents when power is removed; it holds the currently-running program and its data. **ROM** (Read-Only Memory) is **non-volatile** and stores permanent instructions that survive power loss (e.g. the bootstrap/BIOS/firmware needed to start the computer). **Cache** is small, very fast memory built close to (or on) the CPU, storing recently/frequently used instructions and data so the CPU need not wait for slower RAM as often.

VN **GIẢI THÍCH TIẾNG VIỆT**

Phân cấp bộ nhớ đánh đổi **tốc độ** với **dung lượng/chi phí**: thanh ghi (nhanh nhất, nhỏ nhất) → cache → RAM → ổ đĩa (SSD/HDD) → lưu trữ ngoại tuyến. **RAM** là bộ nhớ tạm (mất dữ liệu khi tắt nguồn); **ROM** lưu trữ vĩnh viễn (chương trình khởi động BIOS/firmware). **Cache** nhỏ nhưng rất nhanh, lưu dữ liệu/lệnh hay dùng gần đây.

Secondary storage technologies

HDD (Hard Disk Drive) — magnetic platters spun at high speed, read/written by a moving head; cheap per gigabyte, but slower and mechanically fragile. **SSD** (Solid-State Drive) — flash memory with no moving parts; much faster, more durable, and more power-efficient, but costlier per gigabyte. **Optical** (CD/DVD/Blu-ray) — data etched as pits read by laser; cheap, portable, but low capacity and slow. **Magnetic tape** — extremely cheap per gigabyte and durable for decades, but strictly **sequential** access (must wind through the tape) — used almost exclusively for tertiary/archival backup, never for active programs.

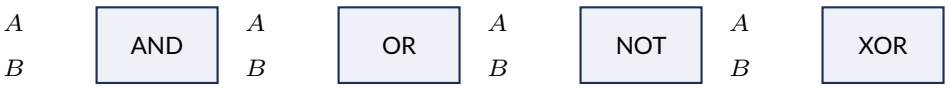
Ví dụ mẫu · Worked example

Explain, with one reason, why cache memory is kept small even though it is much faster than RAM.

Cache uses fast but expensive SRAM technology (compared with the cheaper DRAM used for RAM), and manufacturing cost per byte rises sharply with speed. A larger cache would also take slightly longer to search on every access, eroding some of its speed advantage. So cache is deliberately kept small (a few megabytes) to stay both fast *and* affordable, holding only the most frequently used instructions/data rather than everything.

2.3 Logic gates and Boolean algebra

Digital circuits are built from **logic gates**, each implementing one elementary Boolean function of one or two binary inputs.



Truth values

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

NOT: $\overline{0} = 1,\ \overline{1} = 0.$

Ví dụ mẫu · Worked example

Construct the truth table for $X = (A$ AND $B)$ OR (NOT C) and evaluate it for $A = 1, B = 0, C = 0$.
Substitute directly: A AND $B = 1$ AND $0 = 0$. NOT C = NOT $0 = 1$. So $X = 0$ OR $1 = 1$.

A	B	C	A AND B	$\overline{C}$	X
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	1

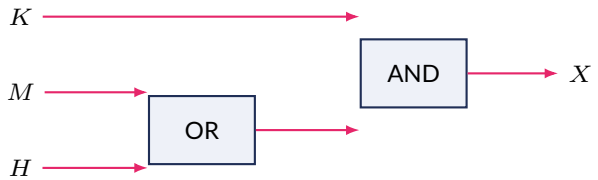
De Morgan's Laws: $\overline{A \text{ AND } B} = \overline{A}$ OR $\overline{B}$, and $\overline{A \text{ OR } B} = \overline{A}$ AND $\overline{B}$. **NAND** = NOT(AND); **NOR** = NOT(OR); **XNOR** = NOT(XOR), true exactly when inputs are *equal*. NAND (and separately NOR) is **functionally complete** — any Boolean circuit can be built using only NAND gates, which is why NAND gates are the physical building block of most real integrated circuits.

VN **GIẢI THÍCH TIẾNG VIỆT**

Cổng logic: AND (cả hai đều 1 thì kết quả 1), OR (một trong hai là 1 thì kết quả 1), NOT (đảo giá trị), XOR (kết quả 1 khi hai đầu vào **khác nhau**), XNOR (kết quả 1 khi hai đầu vào **giống nhau**). **Định luật De Morgan** cho phép chuyển đổi giữa AND/OR khi đảo dấu — rất hay xuất hiện trong đề thi rút gọn biểu thức Boole. NAND là cổng "vạn năng" — có thể dựng mọi mạch logic chỉ từ NAND.

Ví dụ mẫu · Worked example

Design a logic circuit (as a Boolean expression) for a security system that unlocks a door ($X = 1$) only when a valid keycard is swiped ($K = 1$) **AND** either the manager override is active ($M = 1$) **OR** it is during business hours ($H = 1$).
$X = K$ AND $(M$ OR $H)$. Truth-table check at $K{=}1, M{=}0, H{=}1$: M OR $H = 0$ OR $1 = 1$; $X = 1$ AND $1 = 1$ (door unlocks, correctly, since it is business hours). At $K{=}0, M{=}1, H{=}1$: $X = 0$ AND $(\ldots) = 0$ (no keycard ⇒ door stays locked regardless of the other conditions) — matching the requirement that a valid keycard is always mandatory.



Logic circuit for $X = K$ AND $(M$ OR $H)$: the OR gate combines manager override and business hours, then AND with a valid keycard.

The half adder and full adder

A **half adder** adds two single bits A, B, producing a **Sum** ($= A$ XOR B) and a **Carry** ($= A$ AND B). A **full adder** extends this to also accept a carry-in C_{in} from a previous column, producing Sum $= A \oplus B \oplus C_{in}$ and Carry-out $= (A$ AND $B)$ OR $(C_{in}$ AND $(A$ XOR $B))$. Chaining full adders (a **ripple-carry adder**) lets a CPU's ALU add multi-bit binary numbers.

VN **GIẢI THÍCH TIẾNG VIỆT**

Bộ cộng bán phần (half adder): cộng 2 bit, cho Sum (XOR) và Carry (AND). **Bộ cộng toàn phần (full adder)**: cộng thêm carry từ cột trước. Ghép nhiều bộ cộng toàn phần lại tạo thành mạch cộng số nhị phân nhiều bit trong ALU.

2.4 Data representation: number systems

Computers store everything as binary (base 2). **Hexadecimal** (base 16) is a human-friendly shorthand because each hex digit maps exactly to a 4-bit nibble, and **denary** (base 10) is our everyday number system.

Ví dụ mẫu · Worked example

Convert binary `11010110` to hexadecimal and denary.

Split into nibbles: `1101 0110` = `D 6`, so the hex value is `D6`. Denary: $(1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 128 + 64 + 16 + 4 + 2 = 214$. Check via hex: $D6_{16} = 13 \times 16 + 6 = 208 + 6 = 214$. ✓

Ví dụ mẫu · Worked example

Convert denary 183 to binary using repeated division by 2.

$183 \div 2 = 91$ r 1 $\quad 91 \div 2 = 45$ r 1 $\quad 45 \div 2 = 22$ r 1 $\quad 22 \div 2 = 11$ r 0

$11 \div 2 = 5$ r 1 $\quad 5 \div 2 = 2$ r 1 $\quad 2 \div 2 = 1$ r 0 $\quad 1 \div 2 = 0$ r 1

Reading remainders bottom-to-top: `10110111`. Check: $128 + 32 + 16 + 4 + 2 + 1 = 183$. ✓

VN **GIẢI THÍCH TIẾNG VIỆT**

Đổi **nhị phân** ↔ **hex**: gom mỗi 4 bit thành 1 chữ số hex. Đổi **denary** → **nhị phân**: chia liên tục cho 2, ghi lại số dư, rồi đọc số dư từ dưới lên trên. Đổi **nhị phân** → **denary**: nhân mỗi bit với luỹ thừa của 2 tương ứng vị trí rồi cộng lại.

2.5 Two's complement and binary arithmetic

Computers represent signed (negative) integers using **two's complement**: to negate a number, invert every bit then add 1.

Ví dụ mẫu · Worked example

Represent -45 in 8-bit two's complement.

$45 =$ `00101101`. Invert all bits: `11010010`. Add 1: `11010011`. So $-45 =$ `11010011` in 8-bit two's complement (the leading 1 signals a negative value; the range for 8 bits is -128 to 127).

Ví dụ mẫu · Worked example

In 8-bit two's complement, what denary value does `11111011` represent?

Leading bit is 1, so the value is negative. Invert: `00000100`; add 1: `00000101` $= 5$. So the original value is -5.

For n-bit two's complement, the representable range is -2^{n-1} to $2^{n-1} - 1$. Addition works identically to unsigned binary addition — the hardware needs no special negative-number logic, which is precisely why two's complement is the universal standard. **Overflow** occurs in signed addition when two numbers of the *same sign* produce a result of the opposite sign (e.g. adding two positives and getting a negative) — this signals the true result did not fit in the available bits.

Ví dụ mẫu · Worked example

Add 53 and 79 in 8-bit two's complement and check for overflow. (53 = `00110101`, 79 = `01001111`.)

```
  00110101    (53)
+ 01001111    (79)
------------
  10000100
```

The 8-bit result `10000100` has a leading 1, so it is interpreted as *negative* even though both inputs were positive — this is **overflow**: the true sum $53 + 79 = 132$ exceeds the 8-bit signed maximum of 127 and cannot be correctly represented.

VN **GIẢI THÍCH TIẾNG VIỆT**

Bù hai (two's complement) biểu diễn số âm: đảo tất cả các bit rồi cộng thêm 1. Với n bit, phạm vi là -2^{n-1} đến $2^{n-1} - 1$. **Tràn số (overflow)**: cộng hai số cùng dấu mà kết quả lại đổi dấu — nghĩa là kết quả thật vượt quá phạm vi biểu diễn của số bit hiện có.

2.6 Floating point, character, image and sound representation

Real (non-integer) numbers are commonly stored in **floating-point** form, conceptually similar to scientific notation: a **sign** bit, a **mantissa** (significant digits), and an **exponent** (position of the binary point), following standards such as IEEE 754. A larger mantissa gives more **precision**; a larger exponent field gives a wider **range** of magnitudes — for a fixed total number of bits, these two trade off against each other.

Character encoding: ASCII uses 7 (or 8) bits per character, covering 128–256 symbols (English letters, digits, basic punctuation). **Unicode** (e.g. UTF-8/UTF-16) uses a variable number of bytes per character, covering virtually every writing system in the world, including Vietnamese diacritics and emoji. **Image representation:** an image is a grid of **pixels**; **colour depth** (bits per pixel) determines how many distinct colours each pixel can show (e.g. 24-bit colour $= 2^{24} \approx 16.7$ million colours); **resolution** (pixels wide $\times$ pixels tall) determines detail. File size (uncompressed) = width $\times$ height $\times$ bits-per-pixel. **Sound representation:** analogue sound is converted to digital by **sampling** — measuring amplitude at regular intervals (**sample rate**, in Hz) and storing each sample with a given **bit depth**. Higher sample rate and bit depth capture more fidelity but produce larger files.

Ví dụ mẫu · Worked example

An uncompressed image is 800×600 pixels with 24-bit colour. Calculate its file size in megabytes (MB), using 1 MB $= 8 \times 10^6$ bits.

Total bits $= 800 \times 600 \times 24 = 11{,}520{,}000$ bits. File size $= \dfrac{11{,}520{,}000}{8 \times 10^6} = 1.44$ MB.

VN **GIẢI THÍCH TIẾNG VIỆT**

ASCII mã hoá ký tự bằng 7–8 bit (chỉ đủ chữ cái Anh cơ bản); **Unicode** dùng số byte thay đổi, mã hoá được mọi hệ chữ viết (kể cả tiếng Việt có dấu). Kích thước ảnh chưa nén = rộng $\times$ cao $\times$ số bit/điểm ảnh. Âm thanh số hoá bằng **lấy mẫu (sampling)**: tần số lấy mẫu và độ sâu bit càng cao thì chất lượng càng tốt nhưng file càng lớn.

Lossy vs lossless compression

Lossless compression (e.g. ZIP, PNG, FLAC) reduces file size while allowing the *exact* original data to be reconstructed — essential for text, program files, and technical images. **Lossy compression** (e.g. JPEG, MP3, most video codecs) achieves much greater size reduction by permanently discarding data judged least perceptible to humans (e.g. very high/low audio frequencies, subtle colour gradients) — the original cannot be perfectly reconstructed, but the reduction in size is far larger, which is why it is preferred for photos, music and video where some quality loss is acceptable.

VN **GIẢI THÍCH TIẾNG VIỆT**

Nén không mất dữ liệu (lossless): khôi phục lại chính xác dữ liệu gốc (dùng cho văn bản, chương trình). **Nén mất dữ liệu (lossy)**: bỏ bớt dữ liệu ít quan trọng để giảm kích thước nhiều hơn (dùng cho ảnh/nhạc/video — con người khó nhận ra sự khác biệt).

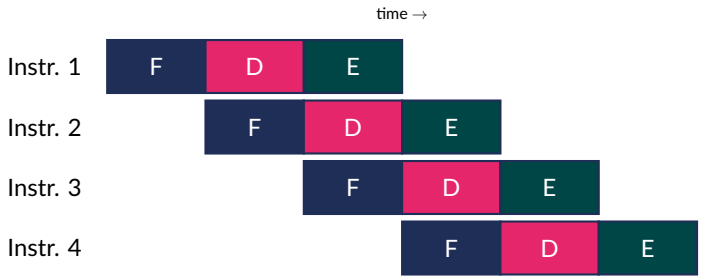
VN **GIẢI THÍCH TIẾNG VIỆT**

Ôn tập nhanh Unit 2: (1) Chu trình fetch–decode–execute và các thanh ghi PC/MAR/MDR/CIR/ACC; (2) phân cấp bộ nhớ: thanh ghi → cache → RAM → lưu trữ thứ cấp → lưu trữ ngoại tuyến; (3) cổng logic AND/OR/NOT/XOR/NAND/NOR và định luật De Morgan; (4) đổi nhị phân–hex–denary thành thạo; (5) bù hai và nhận diện tràn số (overflow); (6) dấu phẩy động = dấu + định trị + số mũ; (7) nén lossy (ảnh/nhạc) vs lossless (văn bản/chương trình).

2.7 CPU performance: pipelining and multicore processing

Real CPUs rarely execute the fetch–decode–execute cycle in strict, one-instruction-at-a-time isolation; several performance techniques overlap or duplicate work to increase throughput.

Pipelining overlaps the stages of *successive* instructions: while instruction 2 is being decoded, instruction 1 can already be executing, and instruction 3 can already be fetched — much like an assembly line. This increases instruction **throughput** (instructions completed per second) without requiring a faster clock, though a **pipeline hazard** (e.g. a branch instruction whose outcome is not yet known) can force the pipeline to stall or discard incorrectly-fetched instructions. **Multicore** processors instead duplicate entire CPU cores on one chip, allowing genuinely independent instruction streams (e.g. different processes, or a single program split into parallel threads) to execute truly simultaneously — but software must be specifically written to exploit multiple cores; a purely sequential program gains no benefit from extra cores.



A 3-stage pipeline (Fetch, Decode, Execute): once full, one instruction completes execution every clock cycle instead of every three.

Ví dụ mẫu · Worked example

A non-pipelined CPU takes 3 clock cycles per instruction (fetch, decode, execute, one per cycle, with no overlap). A pipelined version overlaps these stages as shown above. Estimate the number of cycles to complete 6 instructions under each design, once the pipeline is full.

Non-pipelined: 6 instructions $\times 3$ cycles each $= 18$ cycles (no overlap at all). **Pipelined:** the pipeline takes 3 cycles to fill (first instruction's fetch, decode, execute), then one *additional* instruction completes every single cycle thereafter: $3 + (6 - 1) = 8$ cycles. Pipelining more than **halves** the time here, without any change to the clock speed itself — purely from overlapping independent stages of different instructions.

VN **GIẢI THÍCH TIẾNG VIỆT**

Pipelining: chồng lấn các giai đoạn (fetch/decode/execute) của nhiều lệnh liên tiếp như dây chuyền lắp ráp — tăng thông lượng lệnh mà không cần tăng xung nhịp; **hazard** (như lệnh rẽ nhánh chưa biết kết quả) có thể làm gián đoạn pipeline. **Đa lõi (multicore)**: nhân bản toàn bộ lõi CPU để xử lý thật sự song song — nhưng phần mềm phải được viết để tận dụng nhiều lõi, chương trình tuần tự thuần tuý không được lợi gì từ việc thêm lõi.

2.8 Karnaugh maps for Boolean simplification

A **Karnaugh map (K-map)** is a grid-based tool for simplifying Boolean expressions by visually grouping adjacent 1s (minterms) in a truth table, exploiting the fact that adjacent cells differ in only one variable.

Ví dụ mẫu · Worked example

Simplify $F(A, B, C)$ given the truth table below, using a K-map.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Reading the table, $F = 1$ exactly when $B = 1$ (rows where $B = 1$: rows 3,4,7,8 all have $F = 1$; every row with $B = 0$ has $F = 0$), regardless of A or C. Grouping these four adjacent 1s together (all differing freely in A and C but sharing $B = 1$) simplifies directly to $F = B$ — confirmed since C and A both vary across the group while F stays 1, meaning neither variable is needed in the simplified expression.

K-map grouping rule: valid groups must have a size that is a power of 2 ($1, 2, 4, 8, \ldots$); the larger the group, the more variables are eliminated from that term. A variable that takes *both* values (0 and 1) within a group is eliminated from that group's simplified term; a variable that stays *constant* across the whole group is kept (in its true or complemented form accordingly).

VN GIẢI THÍCH TIẾNG VIỆT

Bản đồ Karnaugh (K-map): công cụ dạng lưới để rút gọn biểu thức Boole bằng cách nhóm các ô có giá trị 1 liền kề nhau (chỉ khác nhau đúng 1 biến). Nhóm càng lớn (luỹ thừa của 2: 1, 2, 4, 8...) thì càng loại bỏ được nhiều biến khỏi biểu thức rút gọn — biến thay đổi giá trị trong nhóm sẽ bị loại, biến giữ nguyên giá trị sẽ được giữ lại.

2.9 RISC vs CISC instruction sets

CISC (Complex Instruction Set Computer, e.g. x86) provides many specialised instructions, some performing multi-step operations in a single instruction — reducing the number of instructions per program, but each instruction may take multiple clock cycles and the CPU hardware is more complex. **RISC** (Reduced Instruction Set Computer, e.g. ARM) provides a small set of simple instructions, each designed to execute in (close to) a single clock cycle — programs need more instructions overall, but simpler, more uniform instructions are easier to pipeline efficiently and consume less power, which is why RISC (ARM) architectures dominate battery-powered mobile devices.

VN GIẢI THÍCH TIẾNG VIỆT

CISC: tập lệnh phức tạp, mỗi lệnh có thể làm nhiều việc (giảm số lệnh nhưng mỗi lệnh tốn nhiều chu kỳ, phần cứng phức tạp) — ví dụ x86. **RISC**: tập lệnh đơn giản, mỗi lệnh gần như 1 chu kỳ (cần nhiều lệnh hơn nhưng dễ pipelining, tiết kiệm năng lượng) — ví dụ ARM, phổ biến trên thiết bị di động dùng pin.

2.10 Cache mapping and locality of reference

Cache is effective because real programs exhibit **locality of reference**: **temporal locality** (recently accessed data/instructions are likely to be accessed again soon, e.g. inside a loop) and **spatial locality**

(data near a recently accessed address is likely to be accessed soon too, e.g. the next array element). Cache exploits both by keeping recently used data and loading it in small contiguous blocks (**cache lines**) rather than one byte at a time.

Cache mapping techniques decide where a block of main memory may be placed within the cache: **direct mapping** — each memory block maps to exactly *one* specific cache line (simple, fast to check, but two frequently-used blocks that map to the *same* line will repeatedly evict each other, called **thrashing**); **fully associative mapping** — a block can go in *any* cache line (best flexibility, minimises this kind of collision, but requires checking every line simultaneously, which is expensive in hardware); **set-associative mapping** — a practical compromise, where memory blocks map to one of a small *set* of lines (e.g. any of 4), balancing flexibility against hardware cost.

Ví dụ mẫu · Worked example

Explain why a program repeatedly looping over a small array benefits enormously from cache, referencing both temporal and spatial locality.

Temporal locality: each element of the small array is accessed again on every loop iteration, so after the very first iteration loads the array into cache, every subsequent iteration can be served directly from the much faster cache rather than slower RAM. **Spatial locality:** since array elements are stored contiguously in memory, loading one element as part of a cache line automatically brings several *neighbouring* elements into cache at the same time, so accessing them shortly afterwards is also a fast cache hit rather than a slow RAM access.

VN **GIẢI THÍCH TIẾNG VIỆT**

Tính cục bộ tham chiếu (locality of reference): **cục bộ thời gian** (dữ liệu vừa dùng có khả năng được dùng lại sớm, như trong vòng lặp) và **cục bộ không gian** (dữ liệu gần địa chỉ vừa truy cập có khả năng được dùng sớm, như phần tử mảng kế tiếp). **Ánh xạ cache**: trực tiếp (mỗi khối bộ nhớ chỉ ánh xạ vào đúng 1 dòng cache — dễ xảy ra tranh chấp/thrashing), liên kết đầy đủ (khối có thể vào bất kỳ dòng nào — linh hoạt nhất nhưng phần cứng phức tạp), liên kết theo tập hợp (thoả hiệp thực tế giữa hai cách trên).

2.11 Von Neumann vs Harvard architecture, and GPUs

The **von Neumann architecture** (assumed throughout this chapter) stores both instructions and data in the *same* memory space, accessed over the *same* bus — simple and flexible (programs can be treated as data, e.g. loaded, modified, compiled), but creating the **von Neumann bottleneck**: the CPU cannot fetch an instruction and read/write data in the same cycle, since both share one bus. The **Harvard architecture** uses *physically separate* memories and buses for instructions and data, allowing both to be accessed simultaneously — faster in principle, but less flexible (harder to treat code as modifiable data), and more complex/costly to build. Many modern CPUs use a hybrid: von Neumann main memory, but separate Harvard-style *instruction* and *data* caches close to the CPU for speed.

A **GPU** (Graphics Processing Unit) contains hundreds or thousands of simpler cores optimised for performing the *same* operation on many different pieces of data simultaneously (**SIMD**: Single Instruction, Multiple Data) — ideal for graphics rendering (the same shading calculation applied to millions of pixels) and for other highly parallel workloads such as training machine-learning models. A **CPU** instead has fewer, more powerful cores optimised for fast execution of varied, sequential, branching general-purpose tasks. Neither replaces the other: a modern computer uses the CPU for general control/logic and offloads suitable, massively parallel sub-tasks to the GPU.

Ví dụ mẫu · Worked example

Explain why applying the same brightness adjustment to every one of a photo's 12 million pixels is much faster on a GPU than on a CPU, even though a single CPU core may individually be faster than a single GPU core.

The brightness adjustment is an identical, independent calculation repeated across every pixel — exactly the **SIMD** pattern GPUs are built for. A GPU can apply that identical calculation to thousands of pixels *simultaneously* across its many simple cores, while a CPU with only a handful of cores must process the 12 million pixels in far fewer parallel "batches," taking many more sequential rounds even though each individual CPU core might complete its own single calculation slightly faster. The sheer number of parallel execution units available on the GPU outweighs any per-core speed advantage the CPU might have.

VN **GIẢI THÍCH TIẾNG VIỆT**

Kiến trúc von Neumann: lệnh và dữ liệu dùng chung 1 vùng nhớ và 1 bus — đơn giản, linh hoạt nhưng gây "nút thắt cổ chai von Neumann" (không thể lấy lệnh và đọc/ghi dữ liệu cùng lúc). **Kiến trúc Harvard**: bộ nhớ và bus lệnh/dữ liệu TÁCH RIÊNG — nhanh hơn nhưng kém linh hoạt hơn, phức tạp/tốn kém hơn. **GPU**: hàng trăm/nghìn lõi đơn giản, xử lý CÙNG một phép tính trên NHIỀU dữ liệu cùng lúc (SIMD) — lý tưởng cho đồ hoạ và học máy; **CPU**: ít lõi mạnh hơn, phù hợp tác vụ tuần tự, rẽ nhánh phức tạp.

2.12 Address bus width and addressable memory

The **address bus** carries memory addresses from the CPU to memory; its **width** (number of wires/bits) directly determines the maximum amount of memory the CPU can ever address, independent of how much RAM is physically installed.

An address bus of n bits can represent 2^n distinct addresses. If each address refers to one byte, the maximum **addressable memory** is 2^n bytes. This is why 32-bit systems are limited to addressing at most 2^{32} bytes (4 GiB) of memory regardless of how much RAM is physically installed — additional installed RAM beyond that limit simply cannot be addressed/used by a strictly 32-bit CPU/OS combination, which is precisely why 64-bit systems (with a vastly larger 2^{64} address space) became necessary as installed RAM capacities grew.

Ví dụ mẫu · Worked example

A CPU has a 16-bit address bus, with each address referring to one byte. Calculate the maximum amount of memory this CPU can address, in bytes and in kibibytes (KiB, where 1 KiB $= 2^{10}$ bytes).

Maximum addresses $= 2^{16} = 65{,}536$. Since each address refers to one byte, maximum ad-

dressable memory $= 65{,}536$ bytes $= \dfrac{65{,}536}{1024} =$ **64 KiB**. Even if far more physical RAM were installed, this CPU could never access more than 64 KiB of it, because there are simply no larger addresses it is capable of generating.

VN **GIẢI THÍCH TIẾNG VIỆT**

Độ rộng bus địa chỉ (address bus width): số bit của bus địa chỉ quyết định số lượng địa chỉ bộ nhớ tối đa CPU có thể đánh địa chỉ, với n bit thì đánh địa chỉ được 2^n vị trí (thường tính bằng byte). Đây là lý do hệ thống 32-bit chỉ đánh địa chỉ tối đa 2^{32} byte (4 GiB) dù RAM lắp vào có nhiều hơn — phần RAM vượt quá giới hạn này không thể được CPU 32-bit truy cập.

2.13 Instruction format: opcode and operand

A machine-code instruction is typically split into an **opcode** (identifying *which* operation to perform, e.g. ADD, LOAD, JUMP) and one or more **operands** (the data or addresses the operation acts on).

Ví dụ mẫu · Worked example

An instruction set uses a fixed 16-bit instruction format: 4 bits for the opcode, and 12 bits for a single operand (a memory address). State (a) how many distinct opcodes this format can support, and (b) the maximum directly-addressable memory this operand field allows.

(a) With 4 opcode bits, there are $2^4 = 16$ distinct opcodes possible — enough for a small instruction set (e.g. LOAD, STORE, ADD, SUB, JUMP, and a handful more). (b) With 12 operand bits used as a memory address, up to $2^{12} = 4096$ distinct memory locations can be directly referenced by a single instruction — a real design trade-off, since widening the opcode field to support more instruction types would necessarily narrow the operand field, reducing the directly addressable memory range (or vice versa), for a fixed total instruction width.

VN **GIẢI THÍCH TIẾNG VIỆT**

Một lệnh mã máy thường gồm **opcode** (mã lệnh — cho biết THỰC HIỆN phép toán gì) và **operand** (toán hạng — dữ liệu/địa chỉ mà lệnh tác động lên). Với độ rộng lệnh cố định, số bit dành cho opcode và operand phải đánh đổi lẫn nhau: nhiều bit opcode hơn cho phép nhiều loại lệnh hơn nhưng ít bit hơn cho địa chỉ operand, và ngược lại.

2.14 Firmware and the boot process

Between the moment a computer is powered on and the moment its full operating system is ready for use, a carefully sequenced startup process must occur — before an OS is even loaded into RAM, *something* must already be running to load it.

Firmware is software permanently stored in ROM (or flash memory that behaves similarly, retaining data without power) directly on a hardware device, providing low-level control that survives power loss — the **BIOS** (Basic Input/Output System) or its modern successor **UEFI** (Unified Extensible Firmware Interface) is the firmware responsible for a computer's startup sequence. The **boot process**: (1) firmware (BIOS/UEFI) runs first directly from ROM, performs a **POST** (Power-On Self-Test) checking essential hardware is present and functioning; (2) firmware locates and loads a small **bootloader** program from designated storage; (3) the bootloader loads the full operating system kernel into RAM and transfers control to it; (4) the OS initialises drivers, services, and finally the user interface.

Ví dụ mẫu · Worked example

Explain why the very first code executed when a computer is powered on cannot be loaded from a hard disk, and must instead run directly from ROM/firmware.

Loading a program from a hard disk requires the CPU to already know *how* to communicate with disk-controller hardware — but immediately after power-on, RAM is empty and no operating system (which would normally provide this capability) has been loaded yet. There must be *some* minimal, permanent starting instructions available the instant power is applied, with no dependency on anything else first being loaded — this is exactly the role of **firmware** stored in ROM, which retains its content without power and is built to run immediately from a fixed, known starting address the CPU always begins executing from.

VN **GIẢI THÍCH TIẾNG VIỆT**

Firmware: phần mềm lưu vĩnh viễn trong ROM/flash trên thiết bị phần cứng, cung cấp điều khiển cấp thấp ngay cả khi mất điện. **BIOS/UEFI**: firmware chịu trách nhiệm khởi động máy tính. **Quy trình khởi động**: firmware chạy trước (kiểm tra phần cứng — POST) → nạp **bootloader** → bootloader nạp nhân hệ điều hành vào RAM → hệ điều hành khởi tạo driver/dịch vụ/giao diện người dùng.

2.15 Utility software and system maintenance

Alongside application software, an operating system relies on **utility software** — programs performing routine maintenance tasks that keep the system running efficiently and reliably.

Disk defragmentation: reorganises fragmented files on a mechanical HDD (Unit 2, earlier) so that each file's data is stored in physically contiguous sectors rather than scattered across the disk — since a fragmented file requires the read/write head to jump between many locations, defragmentation restores faster sequential access (this is unnecessary, and can even reduce lifespan, on an SSD, which has no moving head and near-instant access to any location). **Antivirus/anti-malware software**: scans files and running processes against known malware signatures (and increasingly, suspicious behaviour patterns) to detect and remove malicious software. **Compression utilities**: reduce file size for storage/transmission (Unit 2's lossy/lossless distinction). **Backup utilities**: automate the full/incremental/differential backup strategies discussed in Unit 1.

Ví dụ mẫu · Worked example

Explain why running a disk defragmentation utility on an SSD provides little benefit and may even be actively harmful, unlike on a traditional HDD.

Defragmentation's benefit on an **HDD** comes from reducing the physical distance the read-/write head must travel between a file's scattered fragments — since an SSD has **no moving parts** and can access any memory location in roughly the same very short time regardless of physical arrangement, fragmentation causes essentially no meaningful performance penalty on an SSD to begin with. Worse, flash memory (used in SSDs) has a limited number of write cycles before cells wear out — unnecessarily rewriting and moving data during defragmentation consumes some of this finite write endurance for no real speed benefit, which is why modern operating systems specifically avoid running defragmentation on detected SSDs.

VN **GIẢI THÍCH TIẾNG VIỆT**

Phần mềm tiện ích: **chống phân mảnh đĩa (defragmentation)** — sắp xếp lại file bị phân mảnh trên HDD để tăng tốc (KHÔNG cần và có thể HẠI cho SSD vì không có bộ phận chuyển động và giới hạn số lần ghi). **Diệt virus**: quét tệp/tiến trình theo dấu hiệu mã độc đã biết. **Nén dữ liệu**: giảm kích thước lưu trữ/truyền tải. **Sao lưu tự động**: thực hiện chiến lược backup đầy đủ/tăng dần/khác biệt.

2.16 Practice

Bài tập luyện tập

The hexadecimal value 2F equals which denary number?

(A) 47
(C) 29
(B) 74
(D) 92

Lời giải chi tiết

$2F_{16} = 2 \times 16 + 15 = 32 + 15 = 47$. **(A)**.

Bài tập luyện tập

Convert denary 201 to 8-bit binary and to hexadecimal.

Lời giải chi tiết

$201 = 128 + 64 + 8 + 1 = 2^7 + 2^6 + 2^3 + 2^0$, so binary is 11001001. Splitting into nibbles: 1100 $1001 = \text{C}\ 9$, so hex is $\text{C}9$. Check: $C9_{16} = 12 \times 16 + 9 = 192 + 9 = 201$. ✓

Bài tập luyện tập

In 8-bit two's complement, what denary value does 11111011 represent?

(A) -3
(C) -5
(B) 251
(D) 5

Lời giải chi tiết

Leading bit is 1, so the value is negative. Invert: 00000100; add 1: $00000101 = 5$. So the original value is -5. **(C)**.

Bài tập luyện tập

Represent -100 in 8-bit two's complement.

Lời giải chi tiết

$100 = 01100100$. Invert: 10011011. Add 1: 10011100. So $-100 = 10011100$ in 8-bit two's complement. Check: leading bit 1 (negative); invert $10011100 \rightarrow 01100011$; add 1 $\rightarrow 01100100 = 100$. ✓

Bài tập luyện tập

Simplify $\overline{A \text{ OR } B}$ using De Morgan's Law.

(A) $\overline{A}$ OR $\overline{B}$
(B) $\overline{A}$ AND $\overline{B}$
(C) A AND B
(D) A OR B

Lời giải chi tiết

De Morgan: $\overline{A \text{ OR } B} = \overline{A}$ AND $\overline{B}$. **(B).**

Bài tập luyện tập

Simplify the Boolean expression $Y = (A \text{ AND } B) \text{ OR } (A \text{ AND } \overline{B})$.

Lời giải chi tiết

Factor out A: $Y = A$ AND $(B$ OR $\overline{B})$. Since B OR $\overline{B}$ is always true (a value is either 1 or 0), $Y = A$ AND $1 = A$. So the expression simplifies to just $Y = A$.

Bài tập luyện tập

Place the following in the correct order of the fetch–execute cycle: (i) CU decodes the opcode in CIR, (ii) PC is incremented, (iii) instruction copied from memory into MDR then CIR, (iv) instruction is carried out (possibly using the ALU).

Lời giải chi tiết

Correct order: (iii) fetch the instruction into MDR/CIR → (ii) PC incremented → (i) decode → (iv) execute. In short: **Fetch (iii, ii)** → **Decode (i)** → **Execute (iv)**.

Bài tập luyện tập

Explain, with one reason, why cache memory is small even though it is much faster than RAM.

Lời giải chi tiết

Cache uses fast but expensive SRAM technology (compared to cheaper DRAM used for RAM), and manufacturing cost per byte rises sharply with speed. A larger cache would also take longer to search, reducing its speed advantage — so cache is deliberately kept small (a few MB) to stay both fast *and* affordable, holding only the most frequently-used data/instructions.

Bài tập luyện tập

An SSD and an HDD both offer 1 TB capacity at a similar retail price. Explain one technical reason the SSD is faster, and one reason it may be preferred for a laptop specifically.

Lời giải chi tiết

Speed reason: an SSD uses flash memory with no mechanical moving parts, so data can be accessed almost instantly at any location; an HDD must physically move a read/write head and wait for the correct part of a spinning platter to rotate into position, which takes mea-

surably longer, especially for random (non-sequential) access. **Laptop reason:** SSDs have no moving parts, making them more resistant to damage from being dropped or jolted while a laptop is carried around, and they also consume less power (longer battery life) and produce no mechanical noise, compared with an HDD's spinning platters and moving head.

Bài tập luyện tập

Which gate is described as "functionally complete," meaning any Boolean circuit can be built using only that gate type?

(A) AND
(B) OR
(C) NAND
(D) XOR

Lời giải chi tiết

NAND (and separately NOR) is functionally complete — every other logic gate's function can be reproduced using only NAND gates, which is why real integrated circuits are built almost entirely from NAND gates. **(C)**.

Bài tập luyện tập

Construct the truth table for $Z = \overline{A}$ AND B and state for which input combination(s) $Z = 1$.

Lời giải chi tiết

A	B	$\overline{A}$	$Z = \overline{A}$ AND B	
0	0	1	0	
0	1	1	1	$\leftarrow Z = 1$
1	0	0	0	
1	1	0	0	

$Z = 1$ only when $A = 0$ and $B = 1$.

Bài tập luyện tập

A half adder is fed inputs $A = 1, B = 1$. State the Sum and Carry outputs.

Lời giải chi tiết

Sum $= A$ XOR $B = 1$ XOR $1 = 0$. Carry $= A$ AND $B = 1$ AND $1 = 1$. So the half adder outputs Sum $= 0$, Carry $= 1$ — correctly representing $1 + 1 = 10_2$ (denary 2).

Bài tập luyện tập

Add 01110101 and 00100110 (unsigned 8-bit binary) and give the result in binary and denary.

Lời giải chi tiết

```
  01110101   (117)
+ 00100110   (38)
-----------
```

```
  10011011
```

$10011011_2 = 128 + 16 + 8 + 2 + 1 = 155$. Check: $117 + 38 = 155$. ✓ (No overflow issue arises here since these are treated as unsigned values within the 8-bit range 0–255.)

Bài tập luyện tập

Add two 8-bit two's complement numbers representing $+90$ and $+50$. Show that the result overflows.

Lời giải chi tiết

$90 =$ `01011010`, $50 =$ `00110010`.

```
  01011010   (+90)
+ 00110010   (+50)
-----------
  10001100
```

The result `10001100` has a leading 1, so it is read as *negative*, even though both operands were positive — this is **overflow**, since the true sum $90+50 = 140$ exceeds the 8-bit signed maximum of 127.

Bài tập luyện tập

An uncompressed image is 1024×768 pixels with 8-bit (256-colour) depth. Calculate its file size in megabytes (1 MB $= 8 \times 10^6$ bits).

Lời giải chi tiết

Total bits $= 1024 \times 768 \times 8 = 6{,}291{,}456$ bits. File size $= \dfrac{6{,}291{,}456}{8 \times 10^6} \approx 0.786$ MB.

Bài tập luyện tập

A sound is sampled at 44,100 Hz with 16-bit depth, in stereo (2 channels), for 30 seconds. Estimate the total uncompressed file size in megabytes (1 MB $= 8 \times 10^6$ bits).

Lời giải chi tiết

Bits per second per channel $= 44{,}100 \times 16 = 705{,}600$ bits/s. For 2 channels: $705{,}600 \times 2 = 1{,}411{,}200$ bits/s. Over 30 seconds: $1{,}411{,}200 \times 30 = 42{,}336{,}000$ bits. File size $= \dfrac{42{,}336{,}000}{8 \times 10^6} \approx 5.29$ MB.

Bài tập luyện tập

Explain why increasing an image's colour depth from 8-bit to 24-bit increases both realism and file size, and quantify the increase in file size for a fixed resolution.

Lời giải chi tiết

Colour depth determines how many distinct colours each pixel can represent: 8-bit allows $2^8 = 256$ colours, while 24-bit allows $2^{24} \approx 16.7$ million colours, letting subtle gradients and shading be represented far more realistically instead of banding into visible steps. Since file size (uncompressed) is proportional to bits-per-pixel, moving from 8-bit to 24-bit exactly **triples** the file size for the same resolution (24 is 3 times 8), with no change to the number of pixels.

Bài tập luyện tập

Explain why a text document is normally compressed losslessly, while a photograph is often compressed lossily, using the consequence of data loss in each case.

Lời giải chi tiết

A text document's meaning depends on every single character being exactly correct — losing or altering even one character (through lossy compression) could change a word, a number, or a piece of code and make the document wrong or unusable, so **lossless** compression is required. A photograph, by contrast, is interpreted by the human eye, which cannot perceive very fine, high-frequency detail or subtle colour differences — **lossy** compression can discard exactly this imperceptible detail, achieving a much smaller file size with no noticeable difference in the viewed image.

Bài tập luyện tập

Why does ASCII fail to represent Vietnamese text correctly (e.g. "Tiếng Việt"), and what encoding solves this problem?

(A) ASCII is too slow; use compression instead

(B) ASCII has too few code points for Vietnamese diacritics; use Unicode (e.g. UTF-8)

(C) ASCII cannot store lowercase letters; use hexadecimal

(D) This is a hardware, not encoding, problem

Lời giải chi tiết

Standard ASCII only defines 128 characters — the basic unaccented Latin alphabet, digits and punctuation — with no code points for Vietnamese diacritical and tone marks (e.g. the accented letters in "Tiếng Việt"). **Unicode** (commonly implemented as UTF-8) defines code points for virtually every character in every writing system, including all Vietnamese tone marks and modified letters, correctly representing such text. **(B)**.

Bài tập luyện tập

A CPU's clock speed is increased from 2.5 GHz to 3.5 GHz, but a benchmark shows less than the expected 40% increase in program speed. Suggest one reason clock speed alone does not fully determine performance.

Lời giải chi tiết

Clock speed measures cycles per second, but useful work per cycle also depends on other factors: **memory/cache performance** (if the CPU frequently stalls waiting for data from RAM, a faster clock cannot help during those stalls), the efficiency of **pipelining** and branch prediction,

and whether the program can actually make use of multiple **cores** in parallel. A faster clock only speeds up the parts of execution that are not already bottlenecked elsewhere, so overall speed-up is often less than the raw clock-speed ratio would suggest.

Bài tập luyện tập

Design a Boolean expression for a fire-alarm system that sounds an alarm ($X = 1$) if smoke is detected ($S = 1$) **OR** if temperature is critically high ($T = 1$) **AND** the system is armed ($A = 1$). Evaluate X for $S = 0, T = 1, A = 0$.

Lời giải chi tiết

$X = S$ OR $(T$ AND $A)$. Substituting $S = 0, T = 1, A = 0$: T AND $A = 1$ AND $0 = 0$; $X = 0$ OR $0 = 0$ — the alarm does *not* sound, correctly, since no smoke was detected and the system was not armed even though the temperature was high.

Bài tập luyện tập

Convert hexadecimal `7B` to binary and denary.

Lời giải chi tiết

Each hex digit → 4 bits: $7 =$ `0111`, $B(11) =$ `1011`, giving binary `01111011`. Denary: $7 \times 16 + 11 = 112 + 11 = 123$.

Bài tập luyện tập

Explain why floating-point representation, unlike fixed-point integer representation, involves a trade-off between range and precision for a fixed total number of bits.

Lời giải chi tiết

A floating-point number is stored as a sign, a **mantissa** (the significant digits) and an **exponent** (how far to shift the binary point), similar to scientific notation. For a fixed total bit budget, allocating more bits to the exponent allows representing much larger or smaller magnitudes (wider **range**), but leaves fewer bits for the mantissa, reducing how many significant digits can be stored (lower **precision**) — and vice versa. This is why standards like IEEE 754 fix a specific split (e.g. 8 exponent bits and 23 mantissa bits for 32-bit single precision) as an engineering compromise between range and precision.

Bài tập luyện tập

Two 4-bit unsigned binary numbers `1011` and `0110` are added. Show the result and state whether a carry-out occurs beyond 4 bits.

Lời giải chi tiết

`1011` $= 11$, `0110` $= 6$; $11 + 6 = 17$. In binary: $17 =$ `10001`, which needs 5 bits. Within a strict 4-bit register, only the low 4 bits `0001` would be stored, with a **carry-out of 1** generated beyond the 4th bit — signalling that the true result did not fit in 4 bits (unsigned overflow).

Bài tập luyện tập

A 5-stage pipelined CPU takes 5 cycles to fill, after which one instruction completes every cycle. Estimate the number of cycles needed to complete 20 instructions, and compare with a non-pipelined CPU taking 5 cycles per instruction with no overlap.

Lời giải chi tiết

Pipelined: $5 + (20 - 1) = 24$ cycles. **Non-pipelined:** $20 \times 5 = 100$ cycles. Pipelining reduces the time from 100 cycles to just 24 cycles for the same 20 instructions — more than a 4-fold speed-up, achieved purely by overlapping independent instruction stages rather than increasing clock speed.

Bài tập luyện tập

Explain why a branch (conditional jump) instruction is particularly disruptive to a pipelined CPU, using the term "pipeline hazard."

Lời giải chi tiết

A pipeline continuously fetches and begins processing the *next* instructions in memory order, assuming execution will simply continue sequentially. A **branch** instruction may redirect execution to a completely different address depending on a condition that is not resolved until the **execute** stage — by which point several *wrong* instructions (fetched under the assumption of no branch) may already be partway through the pipeline. This is a **pipeline hazard**: those incorrectly-fetched instructions must be discarded (a "pipeline flush"), wasting the cycles already spent on them and temporarily reducing the pipeline's throughput advantage.

Bài tập luyện tập

Simplify $F(A, B, C)$ using a K-map, given that $F = 1$ only for the rows $(A, B, C) = (0, 0, 1)$ and $(1, 0, 1)$, and $F = 0$ for every other combination.

Lời giải chi tiết

The two rows where $F = 1$ are $(0, 0, 1)$ and $(1, 0, 1)$ — both share $B = 0$ and $C = 1$, while A takes both values (0 and 1) across the group. Since A varies freely within this group of 2, it is eliminated; B and C stay constant (at 0 and 1 respectively) and are kept. The simplified expression is $F = \overline{B}$ AND C.

Bài tập luyện tập

A smartphone manufacturer chooses an ARM (RISC) processor over an x86 (CISC) processor for battery life reasons. Explain the architectural reason RISC processors are generally more power-efficient.

Lời giải chi tiết

RISC processors use a small set of simple, uniform instructions that each typically execute in close to a single clock cycle, requiring simpler control-unit hardware to decode and execute them (compared with CISC's larger, more complex instructions and decoding logic). Simpler hardware consumes less power per operation and enables more efficient pipelining. Although

RISC programs require more individual instructions to accomplish the same task, the reduced hardware complexity and per-instruction power draw makes RISC processors generally more energy-efficient overall — an important advantage for battery-powered devices like smartphones.

Bài tập luyện tập

A program processes a large 2D array by iterating column-by-column instead of row-by-row, even though the array is stored in memory row-by-row (each row contiguous). Explain, using spatial locality, why this is likely to run slower.

Lời giải chi tiết

Because the array is stored row-by-row in memory, accessing consecutive elements of the *same row* is a sequential, contiguous memory access — exploiting **spatial locality**, since each cache line loaded also brings in several useful neighbouring elements from that row. Iterating *column-by-column* instead jumps to a different row for every single access, meaning consecutive accesses are scattered far apart in memory; each access is likely to require loading a brand-new cache line (mostly containing data not needed again soon), causing far more cache misses and much slower execution than the row-by-row access pattern, despite processing exactly the same data.

Bài tập luyện tập

Explain why direct-mapped cache can suffer from ”thrashing” even when the cache overall is far from full, using an example of two frequently-used memory blocks that map to the same cache line.

Lời giải chi tiết

In **direct mapping**, each memory block is assigned to exactly *one* specific cache line determined by its address, regardless of how many other cache lines happen to be free. If two blocks that are both used frequently (e.g. alternately, inside a loop) both happen to map to the *same* single cache line, loading one will always evict the other — even though plenty of other cache lines sit completely unused. The CPU ends up repeatedly reloading both blocks from slower RAM on almost every access (**thrashing**), despite the cache having ample spare capacity elsewhere, simply because direct mapping gives each block no alternative location to go.

Bài tập luyện tập

Explain the ”von Neumann bottleneck” and state one architectural feature of modern CPUs that reduces its practical impact.

Lời giải chi tiết

The **von Neumann bottleneck** arises because instructions and data share the same memory and the same bus, so the CPU cannot fetch the next instruction and read/write data at exactly the same time — the shared bus becomes a limiting factor on overall throughput. Modern CPUs reduce this impact by using **separate instruction and data caches** (a Harvard-style split) close to the CPU, so that most fetches and data accesses are served from these separate, simultaneously-accessible caches rather than contending for the single shared path to main

memory on every single access.

Bài tập luyện tập

A machine-learning engineer trains a neural network that requires the same mathematical operation to be applied to millions of numbers simultaneously. Explain why they would choose to run this on a GPU rather than a CPU.

Lời giải chi tiết

Training a neural network involves applying the same repeated calculations (e.g. matrix multiplications) across enormous numbers of independent values simultaneously — precisely the **SIMD** (Single Instruction, Multiple Data) workload GPUs are purpose-built for, with hundreds or thousands of simple cores executing the same operation on different data in parallel. A CPU's much smaller number of cores, though individually more capable for varied, branching tasks, would have to process this highly repetitive, massively parallel workload in far fewer simultaneous "batches," making training dramatically slower than on a GPU that can exploit the full available parallelism.

Bài tập luyện tập

State one advantage and one disadvantage of fully associative cache mapping compared with direct mapping.

Lời giải chi tiết

Advantage: a memory block can be placed in *any* free cache line, eliminating the forced collisions (thrashing) that direct mapping can suffer when two frequently-used blocks happen to map to the same single line — this generally gives a higher cache hit rate. **Disadvantage:** because a block could be in any line, checking whether data is already cached requires comparing the requested address against *every* line in the cache simultaneously, which requires considerably more complex (and expensive/power-hungry) comparison hardware than direct mapping's single, fixed lookup.

Bài tập luyện tập

Explain why a program written using only a single sequential thread of execution gains no speed benefit from being run on an 8-core CPU instead of a 1-core CPU.

Lời giải chi tiết

Multiple cores provide the *capacity* for several independent instruction streams to execute simultaneously, but this capacity can only be used if the software is actually structured into multiple independent tasks/threads that can run concurrently. A purely sequential program executes one single, unbroken chain of instructions where each instruction may depend on the result of the previous one — there is nothing independent to distribute across the extra 7 cores, which therefore sit idle. Only software specifically written (or compiled) to split its work into genuinely parallel threads can benefit from additional cores.

Bài tập luyện tập

A microcontroller has a 10-bit address bus, with each address referring to one byte. Calculate the maximum memory it can address, in bytes.

Lời giải chi tiết

Maximum addresses $= 2^{10} = 1024$. Since each address refers to one byte, the maximum addressable memory is **1024 bytes** (1 KiB) — this microcontroller could never use more than 1 KiB of memory no matter how much RAM were physically connected to it.

Bài tập luyện tập

Explain why moving from a 32-bit to a 64-bit address bus was necessary as computers began shipping with more than 4 GiB of RAM.

Lời giải chi tiết

A 32-bit address bus can generate at most 2^{32} distinct addresses, corresponding to a hard maximum of 4 GiB of directly addressable memory (assuming byte-addressing) — no matter how much physical RAM is installed beyond this, the CPU simply has no valid address to refer to that extra memory, so it is unusable. Moving to a **64-bit** address bus expands the addressable space to 2^{64} bytes, an astronomically larger limit that comfortably exceeds any RAM capacity currently practical to install, resolving the 4 GiB ceiling and allowing systems to make full use of much larger RAM capacities.

Bài tập luyện tập

An instruction format uses 6 bits for the opcode and 10 bits for a single operand address, in a fixed 16-bit instruction. State how many distinct instructions this format supports and the maximum directly addressable memory.

Lời giải chi tiết

With 6 opcode bits: $2^6 = 64$ distinct instructions are supported. With 10 operand bits used as an address: $2^{10} = 1024$ distinct memory locations can be directly referenced by a single instruction.

Bài tập luyện tập

Explain the trade-off a computer architect faces when designing a fixed-width instruction format's split between opcode bits and operand bits.

Lời giải chi tiết

For a fixed total instruction width, every bit allocated to the **opcode** field increases the number of distinct instructions/operations the instruction set can support, but leaves fewer bits for the **operand** field — reducing the range of memory addresses (or the size of immediate values) a single instruction can directly reference. Conversely, allocating more bits to the operand field allows referencing a wider range of memory directly, but constrains the total number of distinct opcodes/instructions available. Architects must balance these competing needs based on the intended use of the processor — a rich instruction set versus a large directly-addressable range

from a single instruction.

Bài tập luyện tập

Explain the role of POST (Power-On Self-Test) in the boot process, and what typically happens if it detects a critical hardware fault.

Lời giải chi tiết

POST runs as one of the very first actions of the firmware (BIOS/UEFI) immediately after power-on, checking that essential hardware components (RAM, CPU, basic input devices) are present and functioning correctly *before* any further boot steps proceed. If POST detects a critical fault (e.g. no RAM detected, or a failed processor check), the boot process typically halts immediately, often signalling the specific problem through a pattern of beeps or an error code/display message — since continuing to boot an operating system on top of definitively broken essential hardware would be pointless or could risk further damage/data corruption.

Bài tập luyện tập

Explain why firmware, rather than a regular installed application, is used to control very low-level hardware startup tasks.

Lời giải chi tiết

A regular installed application depends on the operating system already being loaded and running to provide it with services (file access, memory management, hardware drivers) — but at the very start of the boot process, none of that exists yet. **Firmware** is stored directly in ROM/flash memory on the hardware itself, requiring no operating system or prior software to already be running, and is specifically designed to be the very first code the CPU executes — making it the only viable place to put code responsible for initiating the boot process itself, before any OS-dependent application could possibly run.

Bài tập luyện tập

Explain why antivirus software that only checks files against a database of known malware signatures can fail to detect brand-new (”zero-day”) malware.

Lời giải chi tiết

Signature-based antivirus detection works by comparing a file’s characteristics against a database of patterns from *previously identified* malware — it can only recognise threats it has already been told about. A brand-new piece of malware that has not yet been discovered, analysed, and added to the signature database has, by definition, no matching entry to compare against, so purely signature-based scanning would not flag it as a threat. This is why modern antivirus software increasingly supplements signature matching with **behaviour-based** detection (flagging suspicious actions, such as a program suddenly attempting to encrypt many files rapidly, even without a matching known signature).

Bài tập luyện tập

A user notices their HDD-based laptop has become noticeably slower at opening large files after months of regular use, with no other changes. Suggest a likely cause and an appropriate utility software fix.

Lời giải chi tiết

A likely cause is **file fragmentation**: over months of creating, deleting, and resizing files, large files may have become split into many non-contiguous pieces scattered across the HDD, forcing the read/write head to move between many locations to read what should be one continuous file. Running a **disk defragmentation** utility would reorganise these scattered fragments back into contiguous blocks, restoring faster sequential read performance — though this diagnosis and fix specifically apply to a mechanical **HDD**; an SSD-based laptop slowing down would need a different explanation, since fragmentation barely affects SSD performance.

Bài tập luyện tập

Convert the denary number 58 to binary, then to hexadecimal, showing your working for both.

Lời giải chi tiết

Binary: $58 = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1$, giving `00111010`. Hexadecimal: split into nibbles `0011 1010` = `3 A`, so hex is `3A`. Check: $3A_{16} = 3 \times 16 + 10 = 48 + 10 = 58$. ✓

Bài tập luyện tập

Explain why a truth table with 4 input variables (A, B, C, D) has exactly 16 rows, and state the general formula for n variables.

Lời giải chi tiết

Each variable can independently take one of 2 values (0 or 1), and a truth table must list every possible *combination* of all variables' values — with 4 independent variables, the total number of combinations is $2 \times 2 \times 2 \times 2 = 2^4 = 16$. In general, for n boolean variables, the truth table has exactly $\mathbf{2^n}$ rows, since each additional variable doubles the number of combinations that must be listed.

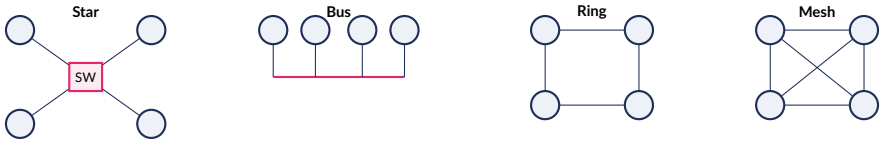
Networks

Mục tiêu Unit: Mạng LAN/WAN, các tô pô mạng và thiết bị phần cứng mạng; mô hình OSI 7 tầng & TCP/IP 4 tầng; giao thức mạng, chuyển mạch gói/kênh; địa chỉ IP/MAC và mô hình client–server/P2P; băng thông & độ trễ; bảo mật mạng cơ bản.

3.1 Network fundamentals, topologies and hardware

A **LAN** (Local Area Network) covers a small area (one building or site) and is usually privately owned and managed. A **WAN** (Wide Area Network) spans large geographic distances — often countries or continents — typically using third-party infrastructure (e.g. the internet, leased lines). A **topology** is the arrangement (physical layout or logical data flow) of devices on a network.

Topology	Description	Trade-off
Star	All devices connect to a central switch/hub	Reliable (one cable failure does not affect others), but the central switch is a single point of failure
Bus	All devices share one central cable	Cheap to install; one cable break can disable the whole segment
Ring	Devices connected in a closed loop, data passes node to node	Predictable performance under load, but one broken link can disrupt the ring (unless dual-ring)
Mesh	Every device connects to many/all others	Highly fault-tolerant (many alternate paths); expensive and complex to cable



The four classic network topologies.

VN **GIẢI THÍCH TIẾNG VIỆT**

LAN phạm vi nhỏ (1 toà nhà), **WAN** phạm vi rộng (nhiều thành phố/quốc gia). **Tô pô hình sao** phổ biến nhất hiện nay vì an toàn (1 dây đứt không sập cả mạng), còn **tô pô bus/ring** rẻ hơn nhưng dễ bị ảnh hưởng toàn mạng khi đứt cáp; **mesh** chịu lỗi tốt nhất nhưng đắt và phức tạp nhất.

Network hardware. **NIC** (Network Interface Card) — lets a device physically connect to a network, holding its unique **MAC address**. **Hub** — broadcasts incoming data to *every* connected port (outdated, wastes bandwidth). **Switch** — forwards data only to the specific port whose device matches the destination MAC address (efficient, standard for modern LANs). **Router** — connects different networks together and forwards data (packets) between them based on **IP address**, typically also connecting a LAN to a WAN/the internet. **Access point** — allows wireless (Wi-Fi) devices to join a wired network.

VN **GIẢI THÍCH TIẾNG VIỆT**

NIC: card mạng, giữ địa chỉ MAC duy nhất. **Hub**: gửi dữ liệu tới TẤT CẢ cổng (lãng phí băng thông, lỗi thời). **Switch**: chỉ gửi tới đúng cổng có địa chỉ MAC đích (hiệu quả hơn). **Router**: kết nối các mạng khác nhau, định tuyến theo địa chỉ IP (thường nối LAN với Internet). **Access point**: cho thiết bị Wi-Fi kết nối vào mạng có dây.

3.2 The OSI and TCP/IP models

The **OSI model** splits network communication into 7 conceptual layers so that hardware/software from different vendors can interoperate reliably. Data passed down each layer is wrapped with that layer's own header (**encapsulation**); the receiving device unwraps it layer by layer (**de-encapsulation**).

7 · Application — HTTP, HTTPS, FTP, SMTP, DNS
6 · Presentation — encryption, compression, data formatting
5 · Session — opens/maintains/closes a communication session
4 · Transport — TCP/UDP; segmentation, reliability, flow control
3 · Network — IP addressing and routing between networks
2 · Data Link — MAC addressing, framing, switches
1 · Physical — bits, cables, radio signals, connectors

The OSI 7-layer model. TCP/IP condenses this into 4 layers: Application (OSI 5–7), Transport (OSI 4), Internet (OSI 3), and Network Access (OSI 1–2).

VN **GIẢI THÍCH TIẾNG VIỆT**

Mô hình **OSI** có 7 tầng, giúp các thiết bị/hãng khác nhau "nói chuyện" được với nhau. Mẹo nhớ: **A**pplication, **P**resentation, **S**ession, **T**ransport, **N**etwork, **D**ata Link, **P**hysical. Mỗi tầng đóng gói dữ liệu bằng header riêng (**encapsulation**); bên nhận tháo từng lớp ra (**de-encapsulation**). Mô hình **TCP/IP** thực tế chỉ dùng 4 tầng, gộp 3 tầng trên của OSI thành 1 tầng Application.

Ví dụ mẫu · Worked example

At which OSI layer does a switch primarily operate, and at which layer does a router primarily operate? Justify each.

A **switch** forwards frames using **MAC addresses** — this is the defining function of the **Data Link layer (2)**. A **router** forwards packets between different networks using **IP addresses** — this is

the defining function of the **Network layer (3)**. This is why a switch alone cannot connect a home LAN to the internet, but a router can.

3.3 Protocols and packet switching

A **protocol** is an agreed set of rules for communication.

Protocol	Purpose
HTTP / HTTPS	Transfers web pages; HTTPS adds TLS/SSL encryption for security
FTP	Transfers files between client and server
SMTP	Sends outgoing email between mail servers
POP3 / IMAP	Retrieves email from a mail server to a client (POP3 typically downloads and removes; IMAP synchronises and keeps mail server-side)
TCP	Reliable, connection-oriented delivery: guarantees packets arrive, in order, retransmitting any lost packet
UDP	Unreliable, connectionless delivery: faster, lower overhead, no guarantee of arrival or order (used for live video/voice where speed matters more than perfection)
DNS	Translates human-readable domain names (e.g. `example.com`) into IP addresses
DHCP	Automatically assigns IP addresses to devices joining a network

VN **GIẢI THÍCH TIẾNG VIỆT**

HTTP/HTTPS: tải trang web (HTTPS có mã hoá TLS/SSL). **FTP**: truyền file. **SMTP**: gửi email; **POP3/IMAP**: nhận email. **TCP**: truyền tin cậy, có xác nhận, truyền lại gói bị mất — chậm hơn nhưng chắc chắn. **UDP**: truyền nhanh, không đảm bảo, không truyền lại — dùng cho video/thoại trực tiếp. **DNS**: đổi tên miền thành địa chỉ IP. **DHCP**: tự động cấp phát địa chỉ IP.

Most data networks use **packet switching**: a message is split into packets, each routed independently (possibly via different paths) and reassembled at the destination — efficient for bursty traffic and resilient to a single link failure, unlike **circuit switching** (a dedicated path reserved for the whole session, as in traditional telephony, wasting capacity during silent pauses but guaranteeing a fixed data rate throughout).

Ví dụ mẫu · Worked example

A 45 MB file is sent over a connection with a bandwidth of 15 Mbps. Estimate the transmission time (1 byte $= 8$ bits).

File size in bits: $45 \times 8 = 360$ Mb. Time $= \dfrac{360 \text{ Mb}}{15 \text{ Mbps}} = 24$ seconds.

Ví dụ mẫu · Worked example

A file of 900 MB must be transferred within 2 minutes. What minimum bandwidth (in Mbps) is required? (1 byte $= 8$ bits; ignore overhead.)

File size in bits: $900 \times 8 = 7200$ Mb. Time available: $2 \times 60 = 120$ s. Minimum bandwidth

$$= \frac{7200 \text{ Mb}}{120 \text{ s}} = 60 \text{ Mbps}.$$

VN **GIẢI THÍCH TIẾNG VIỆT**

Chuyển mạch gói (packet switching): chia dữ liệu thành các gói nhỏ, mỗi gói có thể đi đường khác nhau rồi ráp lại ở đích — hiệu quả và chịu lỗi tốt hơn **chuyển mạch kênh (circuit switching)** (giữ một đường cố định suốt phiên liên lạc). Công thức tính thời gian truyền: Thời gian $= \frac{\text{Kích thước tệp}}{\text{Băng thông}}$ — nhớ đổi cùng đơn vị (bit/byte, Mb/MB).

(!) **Lỗi thường gặp:** **Bandwidth is measured in bits per second (bps/Mbps/Gbps); file size is usually given in bytes (B/MB/GB).** Always convert to the same unit before dividing — a common exam error is dividing megabytes directly by megabits per second without the $\times 8$ conversion, giving an answer 8 times too large.

3.4 IP addressing, MAC addresses and the client--server model

An **IPv4** address is a 32-bit number, usually written as four decimal octets (0–255) separated by dots, e.g. `192.168.1.1`. This gives roughly 4.3 billion possible addresses, which is now insufficient for the number of internet-connected devices worldwide — **IPv6** (128-bit addresses, written in hexadecimal groups) is the long-term replacement, providing an effectively inexhaustible address space. Private address ranges (e.g. `192.168.0.0/16`, `10.0.0.0/8`) are reused inside LANs and are not routable on the public internet; a router performs **NAT** (Network Address Translation) to let many private devices share one public IP address.

A **MAC address** (Media Access Control address) is a 48-bit identifier burned into a NIC by its manufacturer, globally unique, and used for *local* delivery (Data Link layer). An **IP address** identifies a device's location on a *network* for routing purposes (Network layer) and can change (e.g. a laptop gets a new IP address on a different Wi-Fi network, but keeps the same MAC address).

VN **GIẢI THÍCH TIẾNG VIỆT**

Địa chỉ **IPv4** gồm 32 bit, viết thành 4 nhóm số thập phân (0–255). **IPv6** (128 bit) ra đời vì IPv4 sắp hết địa chỉ. **Địa chỉ MAC** (48 bit, gắn cố định vào card mạng) dùng để giao tiếp cục bộ; **địa chỉ IP** dùng để định tuyến qua nhiều mạng và có thể thay đổi.

In the **client–server** model, clients request services/resources from one or more central servers — simpler to manage, secure, back up, and control access centrally, but the server(s) can become a bottleneck or single point of failure. In **peer-to-peer (P2P)**, each device (peer) can act as both client and server, sharing resources directly with other peers — no single point of failure and scales well as more peers join, but harder to secure/manage/audit centrally.

VN **GIẢI THÍCH TIẾNG VIỆT**

Mô hình **client–server**: máy khách yêu cầu dịch vụ từ máy chủ trung tâm (dễ quản lý, bảo mật, nhưng server có thể là điểm lỗi trung tâm). Mô hình **ngang hàng (P2P)**: mỗi máy vừa là client vừa là server (không có điểm lỗi trung tâm, dễ mở rộng, nhưng khó quản lý/bảo mật hơn).

3.5 Network security and cloud computing

Firewall — monitors and filters incoming/outgoing traffic against a set of rules, blocking unauthorised access while permitting legitimate traffic. **Encryption** — scrambles data so it is unreadable without the correct key, protecting data in transit (e.g. TLS/SSL used by HTTPS) and at rest. **VPN** (Virtual Private Network) — creates an encrypted "tunnel" over a public network, letting a remote user securely access a private network as if directly connected. **Authentication** methods — passwords, biometrics, and **two-factor authentication (2FA)** (something you know *plus* something you have/are) reduce the risk of unauthorised access even if one factor is compromised.

VN GIẢI THÍCH TIẾNG VIỆT

Firewall: lọc lưu lượng ra/vào theo quy tắc. **Mã hoá (encryption)**: biến dữ liệu thành dạng không đọc được nếu thiếu khoá giải mã. **VPN**: tạo đường hầm mã hoá qua mạng công cộng để truy cập mạng riêng an toàn. **Xác thực hai yếu tố (2FA)**: kết hợp "thứ bạn biết" (mật khẩu) và "thứ bạn có/là" (điện thoại, vân tay) để tăng bảo mật.

Cloud computing delivers computing resources (storage, processing power, software) over the internet from remote data centres rather than local hardware, typically billed on usage (**pay-as-you-go**). Advantages include scalability (resources can grow/shrink on demand), reduced upfront hardware cost, and access from anywhere with internet connectivity; disadvantages include dependency on internet connectivity, ongoing subscription costs, and reduced direct control over where and how data is physically stored (a factor in privacy/legal-feasibility decisions from Unit 1).

Ví dụ mẫu · Worked example

A school moves its student-record system to a cloud provider. State one benefit and one risk of this decision.

Benefit: the school avoids the upfront cost of buying and maintaining its own servers, and the provider handles scaling, patching, and physical security of the hardware. **Risk:** the school's sensitive student data now depends on a third party's security practices and on a reliable internet connection — if the provider suffers a breach, or the school's internet connection fails, staff may lose access to critical records exactly when they are needed.

VN GIẢI THÍCH TIẾNG VIỆT

Điện toán đám mây (cloud computing): thuê tài nguyên tính toán/lưu trữ qua Internet thay vì mua phần cứng riêng, trả tiền theo mức sử dụng. Ưu điểm: linh hoạt mở rộng, giảm chi phí đầu tư ban đầu. Nhược điểm: phụ thuộc kết nối Internet, mất quyền kiểm soát trực tiếp nơi lưu dữ liệu.

VN GIẢI THÍCH TIẾNG VIỆT

Ôn tập nhanh Unit 3: (1) LAN nhỏ, WAN rộng; sao/bus/ring/mesh và đánh đổi rủi ro; (2) OSI 7 tầng (nhớ APSTNDP) và TCP/IP 4 tầng; switch = tầng 2 (MAC), router = tầng 3 (IP); (3) TCP tin cậy > UDP nhanh; (4) băng thông = kích thước/thời gian, đổi đúng đơn vị bit/byte; (5) IPv4 32-bit, MAC 48-bit cố định; (6) client–server vs P2P; (7) firewall/mã hoá/VPN/2FA bảo vệ mạng.

3.6 Subnetting basics

A **subnet mask** divides an IP address into a **network portion** (identifying which network a device belongs to) and a **host portion** (identifying the specific device within that network). **Subnetting**

splits one larger network into several smaller ones, improving organisation, security (isolating traffic between departments), and efficient use of address space.

A subnet mask such as `255.255.255.0` (also written `/24`) means the first 24 bits are the network portion and the remaining 8 bits are the host portion, allowing $2^8 = 256$ addresses (minus 2 reserved: the network address and the broadcast address, giving 254 usable host addresses). A smaller host portion (e.g. `/28`, 4 host bits) supports fewer devices per subnet ($2^4 = 16$, minus 2 reserved $= 14$ usable) but allows more, smaller subnets to be created from the same address block.

Ví dụ mẫu · Worked example

A network uses the address block `192.168.1.0` with subnet mask `255.255.255.192` (i.e. `/26`). How many usable host addresses does each resulting subnet provide?

A `/26` mask leaves $32 - 26 = 6$ bits for hosts. Total addresses per subnet $= 2^6 = 64$. Subtracting the 2 reserved addresses (network address and broadcast address): $64 - 2 =$ **62** usable host addresses per subnet.

VN **GIẢI THÍCH TIẾNG VIỆT**

Mặt nạ mạng con (subnet mask) chia địa chỉ IP thành phần **mạng** và phần **máy chủ**. **Chia mạng con (subnetting)** tách một mạng lớn thành nhiều mạng nhỏ hơn — dễ quản lý, cách ly lưu lượng theo phòng ban, dùng địa chỉ hiệu quả hơn. Số địa chỉ khả dụng mỗi mạng con $= 2^{(\text{số bit máy chủ})} - 2$ (trừ địa chỉ mạng và địa chỉ broadcast).

3.7 Wireless networking

Wi-Fi (based on the IEEE 802.11 family of standards) lets devices connect to a LAN over radio waves via an **access point**, identified to users by its **SSID** (Service Set Identifier, the network's visible name). Successive Wi-Fi standards (e.g. 802.11n, ac, ax/"Wi-Fi 6") have progressively increased maximum theoretical bandwidth and improved handling of many simultaneous devices.

Wireless security relies on encrypting traffic between device and access point: **WEP** (Wired Equivalent Privacy) is an old, now cryptographically **broken** standard that should never be used. **WPA2** and the newer **WPA3** use much stronger encryption and are the current standards for securing home/enterprise Wi-Fi. An access point broadcasting its SSID with *no* password (open network) allows anyone in range to join and, on many older configurations, to potentially intercept other users' unencrypted traffic.

VN **GIẢI THÍCH TIẾNG VIỆT**

Wi-Fi (chuẩn IEEE 802.11) kết nối thiết bị vào LAN qua sóng radio thông qua **access point**, nhận diện bằng **SSID** (tên mạng). **WEP** đã bị bẻ khoá, không nên dùng; **WPA2/WPA3** là chuẩn mã hoá hiện tại, an toàn hơn nhiều. Mạng Wi-Fi mở (không mật khẩu) cho phép bất kỳ ai trong phạm vi kết nối, rủi ro bị nghe lén dữ liệu.

3.8 Network performance: latency, jitter and throughput

Bandwidth is the theoretical *maximum* data rate a connection can carry (e.g. in Mbps). **Throughput** is the *actual* data rate achieved in practice, which is often lower than bandwidth due to overhead, congestion, or interference. **Latency** is the delay between sending and receiving data (e.g. measured by `ping` round-trip time) — critical for real-time applications like gaming and video calls, where even high bandwidth cannot compensate for high latency. **Jitter** is the *variation* in latency over time — inconsistent delay disrupts real-time audio/video far more than a consistently higher (but stable) latency would, since it is harder for the receiving application to smoothly buffer and play back irregularly-arriving data.

Ví dụ mẫu · Worked example

A gamer has a connection with very high bandwidth (500 Mbps) but complains of "lag" (noticeable delay between pressing a button and seeing the result on-screen) during online games, despite large files downloading almost instantly. Explain this apparent contradiction.

File downloads benefit primarily from high **bandwidth** — total data volume matters more than delay for a one-off transfer. Online gaming instead depends heavily on low, consistent **latency** (and low **jitter**): every input must round-trip to the game server and back with minimal, stable delay for the game to feel responsive. A connection can have enormous bandwidth (explaining fast downloads) while still suffering from high or variable latency (e.g. due to distance to the server, network congestion, or an unstable Wi-Fi signal) — the two metrics are largely independent, and gaming performance depends on the one that high bandwidth does not fix.

VN **GIẢI THÍCH TIẾNG VIỆT**

Băng thông (bandwidth): tốc độ dữ liệu TỐI ĐA về mặt lý thuyết. **Thông lượng (throughput)**: tốc độ dữ liệu THỰC TẾ đạt được (thường thấp hơn băng thông). **Độ trễ (latency)**: thời gian trễ giữa gửi và nhận — quan trọng với game/gọi video. **Jitter**: độ trễ KHÔNG ỔN ĐỊNH theo thời gian — gây gián đoạn âm thanh/video trực tiếp nhiều hơn cả độ trễ cao nhưng ổn định.

3.9 Cloud service models

IaaS (Infrastructure as a Service) — rents raw virtualised hardware (servers, storage, networking); the customer installs and manages their own operating system and software (most control, most management responsibility). **PaaS** (Platform as a Service) — provides a ready-to-use platform (OS, runtime, development tools) on which customers deploy their own applications, without managing the underlying servers. **SaaS** (Software as a Service) — delivers a complete, ready-to-use application over the internet (e.g. web-based email, office suites); the customer manages nothing beyond their own data and account settings (least control, least management responsibility).

Ví dụ mẫu · Worked example

A company wants to run a fully custom database engine with specific low-level configuration, while another company just wants to use an off-the-shelf web-based email service for staff. Identify the most appropriate cloud service model for each.

The first company needs deep, low-level control over the operating environment for its custom database engine — best served by **IaaS**, renting virtualised hardware and installing/configuring everything itself. The second company wants a ready-made application with no infrastructure

or platform management at all — best served by **SaaS**, simply using a complete hosted email application.

VN GIẢI THÍCH TIẾNG VIỆT

IaaS: thuê phần cứng ảo hoá thô, tự cài hệ điều hành/phần mềm (kiểm soát nhiều nhất). **PaaS**: nền tảng sẵn sàng (hệ điều hành, công cụ phát triển) để triển khai ứng dụng riêng. **SaaS**: phần mềm hoàn chỉnh dùng ngay qua Internet (email, bộ văn phòng trực tuyến) — kiểm soát ít nhất nhưng tiện lợi nhất.

3.10 Error detection: parity and checksums

Data can be corrupted during transmission (electrical interference, signal degradation over distance, collisions). **Error detection** techniques let the receiver notice that corruption occurred, so it can request retransmission.

A **parity bit** is one extra bit appended to a block of data, set so that the total number of `1`s (including the parity bit) is always even (**even parity**) or always odd (**odd parity**). The receiver recounts the `1`s; if the parity does not match what is expected, at least one bit was corrupted. A simple parity bit can *detect* a single-bit error but cannot detect an even number of simultaneous bit errors (they cancel out) and cannot identify *which* bit was corrupted. A **checksum** sums up blocks of the transmitted data (often modulo some fixed value) and sends this sum alongside the data; the receiver independently recomputes the sum and compares it to the transmitted checksum — a mismatch indicates an error, and checksums generally catch a wider range of error patterns than a single parity bit.

Ví dụ mẫu · Worked example

Using **even parity**, what parity bit should be appended to the 7-bit data `1011001`, and what does the receiver do if it later receives `10110010` but recomputes and finds an odd number of 1s?

Count the 1s in `1011001`: there are four (positions with 1: 1,0,1,1,0,0,1 → four 1s). Since four is already even, the parity bit appended is `0`, giving the transmitted byte `10110010` (8 bits, still an even total of four 1s). If the receiver recomputes the count on a *received* byte and finds an **odd** number of 1s instead of the expected even number, this signals that at least one bit was corrupted during transmission, and the receiver should request retransmission of that data.

VN GIẢI THÍCH TIẾNG VIỆT

Bit chẵn lẻ (parity bit): 1 bit thêm vào để tổng số bit 1 luôn chẵn (even parity) hoặc luôn lẻ (odd parity) — phát hiện được lỗi 1 bit nhưng KHÔNG phát hiện được lỗi ở số bit chẵn (chúng triệt tiêu nhau) và không biết bit nào sai. **Checksum**: tính tổng các khối dữ liệu rồi gửi kèm, bên nhận tính lại và so sánh — phát hiện được nhiều kiểu lỗi hơn parity bit đơn.

3.11 The internet, ISPs and content delivery networks

The **internet** is a global **WAN of WANs** — a network of interconnected networks with no single central owner. **Internet Service Providers (ISPs)** connect individual homes/businesses to this global network, and themselves interconnect at large **internet exchange points**, forming the internet's high-capacity **backbone**.

A **Content Delivery Network (CDN)** stores copies (**caches**) of frequently-requested content (images, video, web pages) on servers distributed geographically close to end users, rather than serving every single request from one distant, central server. When a user requests content, they are automatically routed to a nearby CDN server rather than the original server potentially located on another continent. This reduces **latency** (physically shorter round trips), reduces load on the original server, and improves resilience (if one CDN server or region fails, requests can be routed to another).

Ví dụ mẫu · Worked example

A streaming video company's servers are located only in one country, yet users worldwide report smooth playback with minimal buffering. Explain how a CDN makes this possible.

Without a CDN, every user's video stream would need to travel all the way to the single distant server and back, incurring high latency and competing for that server's limited bandwidth — especially painful for users physically far from that one location. With a **CDN**, popular video content is **cached** on servers distributed around the world; a user in a distant country is served from a nearby CDN server holding a copy of the same content, dramatically reducing the physical distance (and therefore latency) the data must travel, and spreading the load across many servers instead of overwhelming one.

VN **GIẢI THÍCH TIẾNG VIỆT**

Internet: mạng của các mạng (WAN of WANs), không có chủ sở hữu trung tâm. **ISP** (nhà cung cấp dịch vụ Internet) kết nối hộ gia đình/doanh nghiệp vào mạng toàn cầu. **Mạng phân phối nội dung (CDN)**: lưu bản sao nội dung phổ biến trên các server đặt gần người dùng ở nhiều nơi trên thế giới, giảm độ trễ và tải cho server gốc, tăng độ tin cậy khi một server gặp sự cố.

3.12 VLANs and network redundancy

A **VLAN** (Virtual LAN) lets a single physical switch/network be logically divided into multiple separate, isolated networks, as if they were on entirely separate physical hardware — combining the security/traffic-management benefits of subnetting (Unit 3, earlier) with the flexibility of not needing separate physical switches and cabling for each department.

Devices on different VLANs cannot communicate directly without going through a **router** (or a switch with routing capability) — exactly as if they were on physically separate networks — even though they may share the very same physical switch and cabling. This lets an organisation, for example, keep "Guest Wi-Fi" traffic logically isolated from "Finance" traffic on the same physical office network infrastructure, without running separate cables to every device.

Ví dụ mẫu · Worked example

A company has one physical switch serving both the sales team and the finance team. Explain how a VLAN achieves logical separation between them without installing a second physical switch.

Each port on the switch is configured (in software) to belong to a specific VLAN — e.g. ports 1–12 assigned to "Sales VLAN" and ports 13–24 assigned to "Finance VLAN." The switch then treats traffic from each VLAN as if it were on an entirely separate network: broadcast traffic from a Sales device is not sent to Finance ports, and vice versa, even though both sets of devices are physically plugged into the very same switch. Communication between the two VLANs (if needed at all) must pass through a router, which can apply security policies (e.g.

blocking Sales devices from reaching Finance servers) exactly as it would between two physically separate networks.

VN GIẢI THÍCH TIẾNG VIỆT

VLAN (mạng LAN ảo): chia một switch/mạng vật lý thành nhiều mạng logic tách biệt, như thể chúng nằm trên phần cứng riêng biệt — kết hợp lợi ích bảo mật/quản lý lưu lượng của việc chia mạng con mà không cần switch và dây cáp riêng cho từng phòng ban. Thiết bị khác VLAN muốn giao tiếp phải đi qua router, dù dùng chung switch vật lý.

3.13 RAID: redundant storage

RAID (Redundant Array of Independent Disks) combines multiple physical disks into one logical unit to improve performance, fault tolerance, or both — directly relevant to network servers that must remain available even if one disk fails.

RAID 0 (striping): splits data evenly across multiple disks with *no* redundancy — improves read-/write speed (multiple disks work in parallel) but a single disk failure loses *all* data, since no copy exists anywhere. **RAID 1 (mirroring):** writes an identical copy of all data to two (or more) disks simultaneously — if one disk fails, the mirrored copy on the other disk still has everything, at the cost of effectively halving usable storage capacity for the same total disk space purchased. **RAID 5:** stripes data across three or more disks *plus* distributed parity information, allowing the array to survive any *single* disk failure (the failed disk's data can be reconstructed from the remaining disks and parity) while using storage more efficiently than full mirroring.

Ví dụ mẫu · Worked example

A company server stores critical data across 4 disks of 1 TB each. Compare the usable storage capacity and fault tolerance of configuring them as RAID 0 versus RAID 1.

RAID 0: all 4 disks combine for a full $4 \times 1 = 4$ TB of usable storage (no redundancy), but if *any single* disk fails, all data across the entire array is lost, since data is split (striped) across all disks with no duplicate copy anywhere. **RAID 1:** data is mirrored, typically usable capacity is roughly half the total raw capacity (e.g. 2 TB usable from 4 TB raw, mirrored in pairs), but the array can survive the failure of a disk in each mirrored pair without losing any data, since a full duplicate copy exists elsewhere in the array.

VN GIẢI THÍCH TIẾNG VIỆT

RAID: kết hợp nhiều ổ đĩa vật lý thành một khối logic để tăng hiệu suất và/hoặc độ tin cậy. **RAID 0 (striping)**: chia đều dữ liệu, tăng tốc độ nhưng KHÔNG có dự phòng — 1 ổ hỏng là mất hết dữ liệu. **RAID 1 (mirroring)**: sao chép dữ liệu giống hệt sang ổ khác — chịu được lỗi 1 ổ nhưng giảm một nửa dung lượng khả dụng. **RAID 5**: chia dữ liệu + thông tin chẵn lẻ (parity) phân tán trên $\geq$3 ổ — chịu được lỗi 1 ổ mà vẫn tiết kiệm dung lượng hơn RAID 1.

3.14 Quality of Service (QoS)

When a network's total demand approaches or exceeds its available bandwidth, not all traffic can be treated equally without some applications suffering unacceptable degradation.

Quality of Service (QoS) lets a network prioritise certain types of traffic over others during congestion, rather than treating every packet identically on a first-come-first-served basis. For example, a router might be configured to prioritise voice-call and video-conferencing packets (highly sensitive to latency/jitter, Unit 3 earlier) over a background file download (which can tolerate being slightly delayed without any noticeable negative effect, since the user is not watching it in real time). Without QoS, a large background download could saturate available bandwidth and cause a simultaneous video call to stutter badly, even though delaying the file download by a few extra seconds would have gone completely unnoticed.

Ví dụ mẫu · Worked example

An office network is shared by video-conferencing calls and a large scheduled backup transferring files overnight. Explain how configuring QoS to prioritise the video-conferencing traffic improves the experience for both types of traffic overall.

Without QoS, both types of traffic compete equally for the same limited bandwidth — if the backup happens to be running during business hours, it could consume enough bandwidth to cause noticeable video/audio stuttering during calls, which is highly disruptive to real-time communication. With **QoS** configured to prioritise video-conferencing traffic, the router ensures video/audio packets are transmitted promptly even when the network is busy, forcing the backup transfer to slow down and use only the bandwidth left over — the backup simply takes a little longer to finish (a delay of minutes that goes entirely unnoticed, since nobody is watching it in real time), while call quality remains smooth throughout.

VN **GIẢI THÍCH TIẾNG VIỆT**

Chất lượng dịch vụ (QoS): cho phép mạng ưu tiên loại lưu lượng nào đó khi mạng bị tắc nghẽn, thay vì xử lý mọi gói tin như nhau theo thứ tự đến trước. Ví dụ: ưu tiên gói tin gọi video (nhạy cảm với độ trễ) hơn tải file nền (có thể chậm một chút mà không ai để ý) — giúp cả hai loại lưu lượng đều có trải nghiệm tốt hơn tổng thể so với không có QoS.

3.15 Firewalls and packet filtering in more depth

Building on the brief firewall introduction earlier in this chapter, it is worth understanding more precisely how a firewall decides what to block.

A **packet-filtering firewall** examines each packet's header information — source/destination IP address, source/destination port number, and protocol — against a configured set of rules, allowing or blocking the packet accordingly, without inspecting the packet's actual content/payload. A **stateful firewall** goes further, tracking the state of active connections (e.g. remembering that a particular outgoing request was made, so it can correctly allow the corresponding incoming reply through, while still blocking unsolicited incoming traffic that does not correspond to any request the internal network actually made). **Port numbers** identify which specific service/application on a device a packet is intended for (e.g. port 80 for HTTP, port 443 for HTTPS, port 25 for SMTP) — a firewall commonly blocks all ports except those explicitly required for permitted services, minimising the network's exposed "attack surface."

Ví dụ mẫu · Worked example

A company's firewall is configured to allow outgoing traffic on port 443 (HTTPS) and incoming traffic only if it is a reply to a connection the internal network itself initiated, while blocking all unsolicited incoming connections on every port. Explain how this configuration protects the

internal network while still allowing normal web browsing.

Allowing **outgoing** traffic on port 443 lets internal users initiate secure HTTPS connections to any external website as needed. The **stateful** tracking of connections means that when a reply comes back from a website the internal network itself requested, the firewall recognises it as legitimately corresponding to an internal request and allows it through — but any *unsolicited* incoming connection attempt (e.g. an external attacker trying to directly reach an internal computer without any internal request having been made first) is blocked outright, since it does not match any tracked outgoing request. This lets normal web browsing work completely unaffected, while significantly reducing the network's exposure to unsolicited external attacks.

VN **GIẢI THÍCH TIẾNG VIỆT**

Tường lửa lọc gói (packet-filtering firewall): kiểm tra header gói tin (IP nguồn/đích, cổng, giao thức) theo quy tắc cấu hình, không xem nội dung gói tin. **Tường lửa có trạng thái (stateful)**: theo dõi trạng thái kết nối đang hoạt động — cho phép phản hồi hợp lệ của yêu cầu đã gửi đi, nhưng chặn kết nối đến không được yêu cầu. **Số cổng (port)**: xác định dịch vụ cụ thể (80 = HTTP, 443 = HTTPS) — tường lửa thường chặn mọi cổng trừ những cổng thực sự cần thiết, giảm bề mặt tấn công.

3.16 Practice

Bài tập luyện tập

At which OSI layer does a switch primarily operate, using MAC addresses to forward frames?

(A) Physical (1)
(B) Data Link (2)
(C) Network (3)
(D) Transport (4)

Lời giải chi tiết

Switches forward frames based on **MAC addresses**, which is the defining function of the **Data Link** layer. **(B)**.

Bài tập luyện tập

A video-call app splits a live stream into small units, each sent independently and possibly via different routes, then reassembled at the receiver. This is an example of:

(A) Circuit switching
(B) Packet switching
(C) Direct changeover
(D) Parallel running

Lời giải chi tiết

Independent routing and reassembly of small data units is the definition of **packet switching**. **(B)**.

Bài tập luyện tập

A file of 900 MB must be transferred and the requirement is that it completes within 2 minutes. What minimum bandwidth (in Mbps) is required? (1 byte $= 8$ bits; ignore overhead.)

Lời giải chi tiết

File size in bits: $900 \times 8 = 7200$ Mb. Time available: $2 \times 60 = 120$ s. Minimum bandwidth $= \dfrac{7200 \text{ Mb}}{120 \text{ s}} = 60$ Mbps.

Bài tập luyện tập

Explain one advantage of a star topology over a bus topology for a school computer lab.

Lời giải chi tiết

In a **star** topology each computer has its own dedicated cable to the central switch, so if one cable or computer fails, the rest of the network keeps working normally. In a **bus** topology all devices share a single central cable, so a single break can disable communication for every device on that segment — a much larger impact for a busy computer lab.

Bài tập luyện tập

A 250 MB software update must download in under 100 seconds. What minimum bandwidth, in Mbps, is required?

Lời giải chi tiết

File size in bits: $250 \times 8 = 2000$ Mb. Minimum bandwidth $= \dfrac{2000 \text{ Mb}}{100 \text{ s}} = 20$ Mbps.

Bài tập luyện tập

Explain why UDP is preferred over TCP for a live video call, despite UDP offering no guarantee that every packet arrives.

Lời giải chi tiết

A live video call needs data to arrive *quickly* to feel real-time; TCP's guarantee of reliable, in-order delivery requires waiting for retransmission of any lost packet, which would introduce noticeable delay/stutter. **UDP** sends data with minimal overhead and no waiting for lost-packet retransmission — an occasional dropped packet just causes a brief, barely noticeable glitch in video/audio quality, which is far less disruptive to the live experience than the delay TCP's reliability would cause.

Bài tập luyện tập

Which device connects two different networks together, forwarding data based on IP address?

(A) Hub
(B) Switch
(C) Router
(D) NIC

Lời giải chi tiết

A **router** operates at the Network layer, using IP addresses to forward packets between distinct networks (e.g. connecting a home LAN to the internet). **(C)**.

Bài tập luyện tập

Explain why a hub is considered less efficient and less secure than a switch on the same LAN.

Lời giải chi tiết

A **hub** broadcasts every incoming frame out to *all* connected ports regardless of the intended destination, wasting bandwidth (every device processes traffic not meant for it) and reducing security (any device on the hub can see traffic intended for another device). A **switch** inspects the destination MAC address and forwards each frame only to the specific port where that device is connected, using bandwidth more efficiently and preventing other devices from casually observing traffic not addressed to them.

Bài tập luyện tập

State the correct order of OSI layers from Layer 1 to Layer 7.

Lời giải chi tiết

1 Physical, 2 Data Link, 3 Network, 4 Transport, 5 Session, 6 Presentation, 7 Application. (Mnemonic: "**P**lease **D**o **N**ot **T**hrow **S**ausage **P**izza **A**way".)

Bài tập luyện tập

A company's private LAN uses addresses in the range `192.168.0.0/16`. Explain why these devices can still communicate with the public internet despite using private, non-routable addresses.

Lời giải chi tiết

The company's router performs **Network Address Translation (NAT)**: it rewrites the source address of outgoing packets from each device's private IP address to the router's single public IP address (tracking which internal device made which request), and reverses the process for returning traffic — so many devices with private IP addresses can share one public IP address to reach the internet, without every internal device needing its own globally routable address.

Bài tập luyện tập

Distinguish between a MAC address and an IP address, including which OSI layer each is associated with.

Lời giải chi tiết

A **MAC address** is a 48-bit identifier permanently assigned to a network interface card by its manufacturer, globally unique, and used for local delivery within a network segment — associated with the **Data Link layer (2)**. An **IP address** identifies a device's location on a network for the purpose of routing data between different networks, can change depending on which network a device joins, and is associated with the **Network layer (3)**.

Bài tập luyện tập

A user types `www.example.com` into a browser. Which protocol/service translates this into the numerical IP address needed to actually locate the server?

(A) DHCP
(B) DNS
(C) FTP
(D) SMTP

Lời giải chi tiết

DNS (Domain Name System) translates human-readable domain names into the numerical IP addresses that routers actually use to locate and reach a server. **(B)**.

Bài tập luyện tập

Explain the role of DHCP when a laptop joins a new Wi-Fi network for the first time.

Lời giải chi tiết

DHCP (Dynamic Host Configuration Protocol) automatically assigns the joining device an available IP address (plus related settings such as the subnet mask, default gateway, and DNS server address) from the network's address pool, without requiring the user to manually configure networking settings — this is why a laptop can join almost any Wi-Fi network and get online within seconds.

Bài tập luyện tập

A music-sharing application lets any user's computer directly send song files to any other user's computer, with no central file server. Which network model does this describe, and give one disadvantage of this model compared with a client–server approach.

Lời giải chi tiết

This describes a **peer-to-peer (P2P)** model. One disadvantage compared with client–server: it is much harder to centrally manage, secure, back up, or audit content and access, since there is no single controlling server — for example, no central authority can easily verify file integrity or enforce access permissions across all peers.

Bài tập luyện tập

Explain why HTTPS is preferred over plain HTTP for an online banking website, referencing the OSI/TCP-IP model layer where the relevant protection is applied.

Lời giải chi tiết

HTTPS adds TLS/SSL **encryption** (operating conceptually at the Presentation/Session layers of OSI, layered under the Application-layer HTTP itself) so that data exchanged between the browser and bank's server — including login credentials and account details — cannot be read or tampered with by anyone intercepting the traffic in between. Plain **HTTP** sends all data as unencrypted plain text, meaning a network eavesdropper could directly read passwords and financial information, an unacceptable risk for banking.

Bài tập luyện tập

A company installs a firewall between its internal network and the internet. Explain what the firewall does and why it does not, by itself, guarantee complete security.

Lời giải chi tiết

A **firewall** inspects incoming and outgoing traffic against a configured set of rules, blocking traffic that appears unauthorised or suspicious (e.g. traffic from a known malicious IP address, or on an unexpected port) while allowing legitimate traffic through. It does not guarantee complete security because it cannot stop every threat: e.g. it cannot prevent an employee from being tricked by a **phishing** email into voluntarily revealing a password, cannot inspect content hidden inside *encrypted* traffic without additional configuration, and cannot protect against malware already introduced via a USB drive or a legitimately permitted connection.

Bài tập luyện tập

A start-up chooses to host its web application on a cloud provider rather than buying its own servers. Give one financial and one technical reason for this choice.

Lời giải chi tiết

Financial reason: the start-up avoids a large upfront capital cost of purchasing servers, instead paying for cloud resources on a flexible, pay-as-you-go basis that scales with actual usage — important for a start-up with limited initial funding and uncertain future demand. **Technical reason:** the cloud provider can automatically scale resources up during traffic spikes and down during quiet periods, and handles hardware maintenance, redundancy, and security patching, which a small start-up team would otherwise need to manage itself.

Bài tập luyện tập

Explain the difference between full mesh and star topology in terms of fault tolerance and cabling cost, and suggest one scenario where mesh would be justified despite its cost.

Lời giải chi tiết

In a **star** topology, every device has one link to a central switch — if the switch fails, the entire network goes down, and cabling cost is proportional to the number of devices. In a full **mesh** topology, every device connects directly to every other device, so there is no single point of failure and multiple alternate paths exist if any one link fails — but cabling cost grows much faster (roughly with the square of the number of devices), making it very expensive at scale. Mesh is justified in critical infrastructure where any downtime is unacceptable, e.g. connecting core routers in a national telecommunications backbone, where the extra cost of full redundancy is worth avoiding catastrophic, wide-scale outages.

Bài tập luyện tập

A company enforces two-factor authentication (2FA) for remote logins. Explain how this improves security compared with a password alone.

Lời giải chi tiết

A password alone is "something you know," which can be stolen (e.g. through phishing, a data breach, or being guessed). **Two-factor authentication** additionally requires a second, independent factor — typically "something you have" (e.g. a one-time code sent to a registered phone or generated by an authenticator app) or "something you are" (biometrics). An attacker who obtains the password alone still cannot log in without also possessing the second factor, substantially reducing the risk of unauthorised access even after a password is compromised.

Bài tập luyện tập

A remote employee connects to their company's internal network from home using a VPN. Explain what protection the VPN provides that a normal internet connection would not.

Lời giải chi tiết

A **VPN** creates an **encrypted tunnel** between the employee's device and the company network over the public internet, so that any data travelling between them — even if intercepted on the public internet in between — cannot be read or altered by a third party. Without a VPN, the same traffic would travel as ordinary (and potentially unencrypted, depending on the application) packets across the public internet, exposing it to interception, especially on unsecured networks such as public Wi-Fi.

Bài tập luyện tập

An organisation moving from IPv4 to IPv6 explains the change as "a scaling issue, not a security issue." Justify this statement using the size of each address space.

Lời giải chi tiết

IPv4 addresses are 32 bits, giving at most $2^{32} \approx 4.3$ billion unique addresses — a number now insufficient for the world's internet-connected devices (computers, phones, and the growing number of IoT devices). **IPv6** addresses are 128 bits, giving 2^{128} (an astronomically large number) of possible addresses — effectively unlimited for the foreseeable future. The motivation for the change is therefore primarily **running out of unique addresses to allocate** (a scaling/capacity issue), not a flaw in IPv4's security model as such (though IPv6 does also include some built-in security improvements as a secondary benefit).

Bài tập luyện tập

A network uses subnet mask `255.255.255.240` (i.e. `/28`). Calculate the number of usable host addresses per subnet.

Lời giải chi tiết

A `/28` mask leaves $32 - 28 = 4$ bits for hosts. Total addresses $= 2^4 = 16$; subtracting the 2 reserved addresses (network and broadcast): $16 - 2 =$ **14** usable host addresses per subnet.

Bài tập luyện tập

Explain why a company might deliberately subnet its single large office network into separate subnets for "Finance," "Guest Wi-Fi," and "Engineering," even though all three could technically

share one flat network.

Lời giải chi tiết

Subnetting provides both **security** and **traffic management** benefits: isolating "Guest Wi-Fi" onto its own subnet prevents guest devices from directly reaching sensitive internal systems (e.g. Finance's servers) even if a guest device is compromised, since traffic between subnets can be filtered/blocked by a router or firewall. It also contains broadcast traffic and network problems (e.g. a device malfunctioning and flooding traffic) to a single subnet rather than affecting the entire company, and allows different security/access policies to be applied per department.

Bài tập luyện tập

Explain why an open (password-free) Wi-Fi network at a café poses a greater security risk to users than a WPA2/WPA3-secured network, even if both provide identical internet speed.

Lời giải chi tiết

On a WPA2/WPA3-secured network, traffic between each device and the access point is **encrypted**, so even another user connected to the same network cannot easily read data intended for someone else. On an **open network**, this encryption is absent (or, in some configurations, uses a shared key known to everyone), making it far easier for another device on the same network to intercept and read unencrypted traffic — potentially exposing passwords, messages, or other sensitive data sent by users who assume their connection is private.

Bài tập luyện tập

A video call has a stable but somewhat high latency of 150ms; another call has an average latency of 60ms but frequently spikes unpredictably between 20ms and 300ms. Explain which is likely to feel worse to users, referencing "jitter."

Lời giải chi tiết

The second call, despite a lower *average* latency, is likely to feel worse because of high **jitter** — the large, unpredictable variation in delay makes it very difficult for the receiving application to smoothly buffer and play back audio/video, causing stuttering, garbled audio, or frozen video frames. A consistently higher but **stable** latency (the first call) is easier to compensate for with a fixed buffering delay, producing a smoother (if slightly more delayed) experience overall — this is why jitter, not just average latency, is a critical metric for real-time communication quality.

Bài tập luyện tập

A start-up wants to deploy a web application without managing any server operating systems, but still needs to write and deploy custom application code (not just use an existing off-the-shelf app). Which cloud service model best fits this need?

(A) IaaS
(B) PaaS
(C) SaaS
(D) None; only physical servers can do this

Lời giải chi tiết

PaaS (Platform as a Service) provides a ready-to-use runtime/platform on which the start-up can deploy its own custom application code, without needing to install, patch, or manage the underlying operating system or physical servers themselves (unlike IaaS, which would require exactly that). **(B)**.

Bài tập luyện tập

Explain why bandwidth alone is a poor predictor of how ”fast” a network connection will feel for browsing many small web pages that each require multiple quick round-trips to the server.

Lời giải chi tiết

Loading a typical web page often involves many small, sequential requests (HTML, then images, stylesheets, scripts, each potentially requiring a separate round trip) rather than one single large continuous transfer. Because each round trip must wait for the request to travel to the server and the response to travel back, overall page-load ”feel” is dominated by **latency** (the delay per round trip) multiplied by the number of round trips required, not by **bandwidth** (which mainly limits how fast large, continuous transfers can complete). A connection with very high bandwidth but high latency can therefore feel sluggish browsing many small pages, even though it would transfer one enormous file quickly.

Bài tập luyện tập

Using **odd parity**, determine the parity bit that should be appended to the data `1100010`.

Lời giải chi tiết

Count the 1s in `1100010`: there are three (positions 1,2,6). For **odd parity**, the total count (including the parity bit) must be odd. Three is already odd, so the parity bit is `0`, giving the transmitted byte `11000100` (still three 1s, an odd total).

Bài tập luyện tập

Explain why a single parity bit cannot detect an error where exactly *two* bits in the same block are corrupted.

Lời giải chi tiết

A parity bit only checks whether the *total count* of 1s in the block is even or odd, not which specific bits are 1. If exactly two bits flip (e.g. a 0 becomes a 1, and separately a different 1 becomes a 0, or two 0s both become 1s), the overall count of 1s changes by an even amount (either $+2$, -2, or 0 net change) — meaning the parity (even or odd) of the total count is unchanged from what was expected. The receiver’s parity check will therefore incorrectly show ”no error,” failing to detect that two bits were actually corrupted.

Bài tập luyện tập

Explain why a news website with a global audience would use a CDN, referencing both latency and server load.

Lời giải chi tiết

Without a CDN, every reader worldwide would request the same articles/images directly from one (or a few) origin servers, meaning distant readers suffer high **latency** (their requests must physically travel much further), and the origin server must handle the *full* global request volume alone, risking overload during traffic spikes (e.g. breaking news). A **CDN** caches popular content on servers distributed close to readers worldwide, so each reader is served from a nearby location (much lower latency) and the request load is spread across many distributed servers instead of concentrating entirely on the origin server, improving both speed and resilience.

Bài tập luyện tập

Explain why the internet is described as having ”no single central owner,” and state what this implies about how it is intentionally designed to survive localised failures.

Lời giải chi tiết

The internet is a network of thousands of independently-operated networks (ISPs, universities, companies, governments) that agree to interconnect using common protocols (TCP/IP), rather than being built or controlled by any single organisation. Because it is highly interconnected with many redundant paths between networks (rather than one central hub through which everything must pass), a failure or outage in one region or provider does not bring down the whole internet — traffic can typically be rerouted through alternative paths, similar in principle to how a mesh topology tolerates individual link failures.

Bài tập luyện tập

Explain why a school might configure its ”Student Devices” and ”Staff Devices” into separate VLANs on the same physical network switches, rather than purchasing entirely separate physical switches for each group.

Lời giải chi tiết

VLANs provide the same logical isolation and security benefits as physically separate networks (student devices cannot directly reach staff-only resources, and broadcast traffic is contained within each VLAN) without the school needing to install, cable, and maintain two entirely separate sets of physical switches — a significantly cheaper and simpler solution while still achieving the desired separation, since any necessary communication between the two VLANs can be tightly controlled by routing it through a single router with appropriate security rules.

Bài tập luyện tập

A video-editing workstation prioritises maximum read/write speed for large temporary files and has robust separate backups elsewhere, so disk redundancy is not a priority. Which RAID level is most appropriate, and why?

(A) RAID 0, for maximum speed with no redundancy overhead

(B) RAID 1, for maximum redundancy

(C) RAID 5, for a balance of speed and redundancy

(D) No RAID; use one disk only

Lời giải chi tiết

RAID 0 stripes data across multiple disks with no redundancy, maximising read/write speed since multiple disks can be accessed in parallel — appropriate here specifically because the workstation does not need built-in redundancy (it already has separate backups elsewhere) and prioritises raw performance for large temporary editing files. **(A)**.

Bài tập luyện tập

Explain why RAID 1 (mirroring) does not, by itself, protect against a user accidentally deleting an important file.

Lời giải chi tiết

RAID 1 protects against *physical disk failure* by maintaining an identical mirrored copy of data on a second disk — but it operates below the level of individual files, simply copying whatever changes are written. If a user deletes a file, that deletion is itself immediately and faithfully **mirrored** to the second disk as well, since RAID 1 does not distinguish between a legitimate write and an accidental deletion — it just replicates whatever change occurs. Genuine protection against accidental deletion requires a separate **backup** strategy (Unit 1) with distinct point-in-time copies, not RAID mirroring alone.

Bài tập luyện tập

A company needs both good fault tolerance (survive one disk failing) and efficient use of storage capacity across 5 disks, and can tolerate slightly more complex hardware. Which RAID level best fits, and why is it preferred over RAID 1 here?

Lời giải chi tiết

RAID 5 best fits: it can survive the failure of any single disk (reconstructing lost data from parity information distributed across the remaining disks) while using storage more efficiently than RAID 1, which sacrifices roughly half of total raw capacity to full mirroring regardless of how many disks are used. With 5 disks, RAID 5 provides the effective capacity of 4 disks (one disk's worth is used for distributed parity) plus single-disk fault tolerance, compared with RAID 1's much heavier capacity cost for the same number of disks.

Bài tập luyện tập

Explain why QoS is generally unnecessary on a home network with abundant bandwidth relative to actual usage, but becomes important on a congested office network.

Lời giải chi tiết

QoS solves the problem of *deciding which traffic to prioritise when there is not enough bandwidth for everyone at once* — if a home network's available bandwidth comfortably exceeds everything the household actually uses simultaneously, every application can already be served promptly without needing to prioritise any traffic over any other, so QoS provides little practical benefit. On a **congested office network** where total demand regularly approaches or exceeds available bandwidth, some traffic *must* be delayed during peak usage — QoS ensures that delay falls on traffic that can tolerate it (e.g. a background transfer) rather than on traffic that cannot (e.g. a live video call), which matters precisely because congestion is a real, recurring problem in that

environment.

Bài tập luyện tập

A network administrator configures QoS to give the lowest priority to a particular streaming-video website during office hours. Explain the likely business reason for this configuration.

Lời giải chi tiết

A likely reason is to ensure that recreational streaming traffic does not compete for bandwidth with business-critical applications (e.g. video conferencing with clients, cloud-based work tools, VoIP phone calls) during working hours — by deliberately giving this specific traffic the lowest priority, the administrator ensures that if bandwidth becomes constrained, streaming video is the first traffic to be slowed down, protecting the responsiveness of applications the business actually depends on for productive work.

Bài tập luyện tập

Explain the difference between a packet-filtering firewall and a stateful firewall, using the example of an internal computer's web browser awaiting a reply to a page request.

Lời giải chi tiết

A basic **packet-filtering firewall** examines each packet independently against fixed rules (e.g. source/destination IP, port) with no memory of prior traffic — it would need an explicit rule permitting incoming traffic on the relevant port to let the webpage's reply through, which could also allow *any* other incoming traffic on that same port, not just genuine replies. A **stateful firewall** instead remembers that the internal browser *itself initiated* the specific outgoing request, and permits only the corresponding incoming reply traffic tied to that exact request through — any other incoming traffic on the same port that does not match a request the internal network actually made is still blocked, providing tighter, context-aware security.

Bài tập luyện tập

Explain why blocking all network ports except those explicitly required (e.g. only 80 and 443) is considered good security practice, using the term "attack surface."

Lời giải chi tiết

Each open port represents a potential entry point through which an attacker might attempt to reach a service running on that port, especially if that service has an undiscovered vulnerability — collectively, all of a system's open ports and reachable services form its **attack surface**. By blocking every port except those genuinely required for permitted services (e.g. 80/443 for standard web traffic), an organisation minimises the number of potential entry points available to an attacker, reducing the attack surface as much as possible while still allowing all legitimately needed functionality to continue working normally.

Bài tập luyện tập

A 320 MB video file must stream smoothly over a connection guaranteeing at least 8 Mbps. Calculate the minimum time (in seconds) needed to transfer the file, and state one reason actual playback might still stutter despite this calculation.

Lời giải chi tiết

File size in bits: $320 \times 8 = 2560$ Mb. Minimum time $= \dfrac{2560 \text{ Mb}}{8 \text{ Mbps}} = 320$ seconds. Despite this bandwidth calculation, actual playback could still stutter due to **jitter** — variation in latency that disrupts the smooth, steady arrival of data needed for real-time playback, even if the *average* bandwidth is technically sufficient over the whole transfer.

Bài tập luyện tập

Explain why two computers on the same LAN, both configured with private IP addresses in the range `192.168.1.x`, can communicate directly without needing a router, while communication with a server on the public internet requires one.

Lời giải chi tiết

Two devices on the *same* LAN with addresses in the same subnet range can exchange data directly at the Data Link layer using **MAC addresses** via a switch, since they are both part of the same local network segment with no need to cross into a different network. Reaching a server on the public **internet** requires leaving the local private network entirely and being routed across potentially many intermediate networks based on **IP address** — this inter-network routing (and the necessary translation of private to public addressing via NAT) is specifically the role of a **router**, which a simple switch-connected LAN communication does not need.

Computational Thinking, Problem-Solving and Programming

Mục tiêu Unit: Tư duy giải thuật (phân rã, trừu tượng hoá, nhận diện mẫu); lưu đồ & giả mã chuẩn IB; biến, kiểu dữ liệu, mảng 1D/2D và xử lý chuỗi; chương trình con (thủ tục/hàm); các thuật toán tìm kiếm/sắp xếp chuẩn; độ phức tạp thuật toán (Big-O); kiểm thử & bảng vết (trace table).

4.1 Computational thinking concepts

Computational thinking is a structured approach to problem-solving that translates a real-world problem into a form a computer can solve.

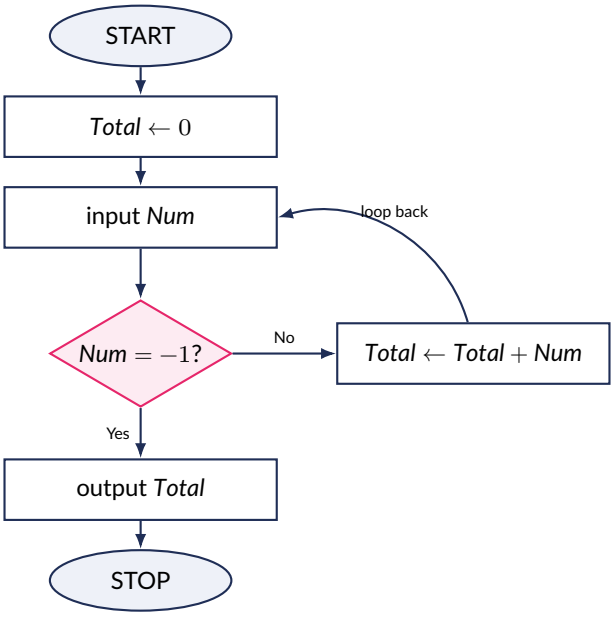
Concept	Meaning
Decomposition	Breaking a complex problem into smaller, more manageable sub-problems that can be solved (and tested) independently
Abstraction	Removing unnecessary detail, keeping only what is relevant to the current problem (e.g. a subway map abstracts away exact geography)
Pattern recognition	Spotting similarities/regularities between problems or within data, so a previously solved technique can be reused
Algorithm design	Producing a precise, ordered, finite sequence of steps that solves the (decomposed, abstracted) problem

VN **GIẢI THÍCH TIẾNG VIỆT**

Tư duy giải thuật gồm 4 trụ cột: **phân rã** (chia bài toán lớn thành phần nhỏ), **trừu tượng hoá** (bỏ chi tiết không cần thiết, giữ lại phần cốt lõi), **nhận diện mẫu** (tìm điểm chung với bài toán đã giải trước đó), **thiết kế giải thuật** (viết ra chuỗi bước giải quyết chính xác, có thứ tự, hữu hạn).

4.2 Flowcharts and IB pseudocode

An **algorithm** is a precise, finite sequence of steps that solves a problem. IB pseudocode uses keywords such as `input`, `output`, assignment `<-`, `if...then...else...end if`, `loop...end loop` (counted `loop I from...to...` or conditional `loop while.../loop until...`), and array notation `List[i]`.



Flowchart: sum numbers entered by the user until the sentinel value -1 is entered.

Ví dụ mẫu · Worked example

Pseudocode for the flowchart above:

```
Total <- 0
input Num
loop while Num <> -1
    Total <- Total + Num
    input Num
end loop
output Total
```

VN **GIẢI THÍCH TIẾNG VIỆT**

Giả mã kiểu IB dùng các từ khoá: `input`, `output`, gán `<-`, `if...then...else...end if`, `loop...end loop`. **Lưu đồ (flowchart):** hình bầu dục = bắt đầu/kết thúc, hình chữ nhật = xử lý, hình thoi = quyết định (rẽ nhánh). Vòng lặp có điều kiện dừng gọi là "sentinel value" (giá trị lính canh), ví dụ -1 trong bài trên.

4.3 Variables, arrays and string manipulation

A **variable** is a named storage location holding a value of a particular **data type** (`INTEGER`, `REAL`, `STRING`, `BOOLEAN`, `CHAR`). A 1D array `List[0..N-1]` stores a fixed-size, indexed collection of same-typed values; a 2D array `Grid[row][col]` models a table/grid, useful for representing spreadsheets, game boards, or matrices.

Common string operations: `LENGTH(S)` (number of characters in `S`); `LEFT(S, n)` (first `n` characters); `RIGHT(S, n)` (last `n` characters); `MID(S, start, n)` (`n` characters beginning at position `start`); concatenation with `+`. IB pseudocode indexes strings and arrays starting from **0** unless the question states otherwise — always check the question's convention.

Ví dụ mẫu · Worked example

Trace the pseudocode below for `Word = "COMPUTER"` (0-indexed):

```
Count <- 0
loop I from 0 to LENGTH(Word) - 1
    if MID(Word, I, 1) = "O" then
        Count <- Count + 1
    end if
end loop
output Count
```

`"COMPUTER"` has 8 characters (indices 0–7): C(0) O(1) M(2) P(3) U(4) T(5) E(6) R(7). The letter `O` occurs at index 1 only, so `Count` increases from 0 to 1 exactly once. Output: `1`.

Ví dụ mẫu · Worked example

Trace this pseudocode for a 2D grid `Grid[0..1][0..2]` $= \{\{4,7,2\},\{9,1,6\}\}$:

```
Total <- 0
loop R from 0 to 1
    loop C from 0 to 2
        Total <- Total + Grid[R][C]
    end loop
end loop
output Total
```

Row 0: $4+7+2=13$. Row 1: $9+1+6=16$. Running total after both rows: $13+16=29$. Output: 29 (this pattern — a loop nested inside another — visits every cell of a 2D grid exactly once).

VN **GIẢI THÍCH TIẾNG VIỆT**

Mảng (array) lưu một dãy giá trị cùng kiểu, truy cập qua chỉ số; mảng 2 chiều `Grid[row][col]` biểu diễn dạng bảng/lưới. **Xử lý chuỗi**: `LENGTH` (độ dài), `LEFT/RIGHT/MID` (cắt chuỗi con), nối chuỗi bằng +. Luôn kiểm tra chỉ số bắt đầu từ 0 hay 1 tuỳ đề bài — đây là lỗi bẫy phổ biến (off-by-one error).

4.4 Subroutines: procedures and functions

A **subroutine** is a named, reusable block of code that can be called from elsewhere in a program, avoiding repeated code and improving readability. A **procedure** performs an action but does not return a value; a **function** performs a computation and **returns** a value to the caller. **Parameters** let a subroutine receive input values each time it is called.

A variable's **scope** is the region of a program where it can be accessed. A **local** variable, declared inside a subroutine, exists only during that call and cannot be accessed from outside it. A **global** variable is accessible throughout the whole program. Overusing global variables makes programs harder to debug (any subroutine could change them unexpectedly) — good practice favours passing values explicitly as parameters instead.

Ví dụ mẫu · Worked example

Write a function `IsEven` that takes an integer parameter and returns `TRUE` if it is even, `FALSE` otherwise; then show it being called.

```
method IsEven(N)
    if N MOD 2 = 0 then
        return TRUE
    else
        return FALSE
    end if
end method

output IsEven(14)    // outputs TRUE
output IsEven(7)     // outputs FALSE
```

`14 MOD 2 = 0`, so `IsEven(14)` returns `TRUE`. `7 MOD 2 = 1`, so `IsEven(7)` returns `FALSE`.

VN **GIẢI THÍCH TIẾNG VIỆT**

Chương trình con: khối lệnh có tên, tái sử dụng được. **Thủ tục (procedure)**: thực hiện hành động, KHÔNG trả về giá trị. **Hàm (function)**: tính toán và TRẢ VỀ một giá trị. **Tham số (parameter)**: giá trị đầu vào mỗi lần gọi. **Phạm vi biến (scope)**: biến cục bộ (local) chỉ tồn tại trong chương trình con; biến toàn cục (global) dùng được khắp chương trình nhưng nên hạn chế để dễ gỡ lỗi.

4.5 Standard searching algorithms

Linear search checks every element in order until the target is found (or the list ends) — works on unsorted data. **Binary search** repeatedly halves a *sorted* array by comparing the target to the middle element — much faster on large data, but *requires* the data to already be sorted.

Ví dụ mẫu · Worked example

Trace a linear search for `Target = 40` in `[12, 7, 40, 3, 18]` (indices 0–4).

```
Index <- 0
Found <- FALSE
loop while Index < 5 AND Found = FALSE
    if List[Index] = Target then
        Found <- TRUE
    else
        Index <- Index + 1
    end if
end loop
```

Index 0: $12 \neq 40$; Index 1: $7 \neq 40$; Index 2: $40 = 40 \rightarrow$ `Found <- TRUE`, loop stops. The target is located at index **2** after 3 comparisons.

Ví dụ mẫu · Worked example

Trace a binary search for `Target = 23` in the sorted array `[2, 7, 12, 18, 23, 31, 40, 55]` (indices 0–7).

Step	Low, High	Mid (index, value)	Comparison
1	0, 7	3, 18	$18 < 23 \Rightarrow$ search right half
2	4, 7	5, 31	$31 > 23 \Rightarrow$ search left half
3	4, 4	4, 23	Match found at index 4

Found in 3 comparisons instead of up to 8 for linear search. (Mid index computed as $\lfloor (\text{Low} + \text{High})/2 \rfloor$ at each step.)

VN **GIẢI THÍCH TIẾNG VIỆT**

Tìm kiếm tuyến tính (linear search): duyệt từng phần tử, dùng được với dữ liệu chưa sắp xếp, độ phức tạp $O(n)$. **Tìm kiếm nhị phân (binary search)**: yêu cầu dữ liệu **đã sắp xếp**, so sánh với phần tử giữa rồi loại bỏ một nửa mỗi lần, độ phức tạp $O(\log n)$ — nhanh hơn nhiều với dữ liệu lớn.

4.6 Standard sorting algorithms

Bubble sort repeatedly steps through the list, swapping adjacent elements that are out of order, until a full pass makes no swaps. **Insertion sort** builds a sorted section at the start of the list one element at a time, inserting each new element into its correct position among those already sorted.

Ví dụ mẫu · Worked example

Bubble sort pseudocode and a full trace of `[5, 2, 4, 1]`:

```
loop I from 0 to N - 2
    loop J from 0 to N - 2 - I
        if List[J] > List[J + 1] then
            Temp <- List[J]
            List[J] <- List[J + 1]
            List[J + 1] <- Temp
        end if
    end loop
end loop
```

$N = 4$. **Pass 1** ($I = 0$, J from 0 to 2): compare $(5, 2) \rightarrow$ swap $\rightarrow [2, 5, 4, 1]$; compare $(5, 4) \rightarrow$ swap $\rightarrow [2, 4, 5, 1]$; compare $(5, 1) \rightarrow$ swap $\rightarrow [2, 4, 1, 5]$. **Pass 2** ($I = 1$, J from 0 to 1): compare $(2, 4)$ no swap; compare $(4, 1) \rightarrow$ swap $\rightarrow [2, 1, 4, 5]$. **Pass 3** ($I = 2$, $J = 0$): compare $(2, 1) \rightarrow$ swap $\rightarrow [1, 2, 4, 5]$. List is now fully sorted: `[1, 2, 4, 5]`.

Ví dụ mẫu · Worked example

Insertion sort pseudocode and a trace of `[9, 4, 6, 2]`:

```
loop I from 1 to N - 1
    Key <- List[I]
    J <- I - 1
    loop while J >= 0 AND List[J] > Key
```

```
        List[J + 1] <- List[J]
        J <- J - 1
    end loop
    List[J + 1] <- Key
end loop
```

$I = 1$: Key$= 4$; compare with $9 > 4$, shift: $[9, 9, 6, 2] \rightarrow$ place Key at index 0: $[4, 9, 6, 2]$. $I = 2$: Key$= 6$; compare with $9 > 6$, shift: $[4, 9, 9, 2] \rightarrow$ compare with 4: not > 6, stop; place Key at index 1: $[4, 6, 9, 2]$. $I = 3$: Key$= 2$; compare $9 > 2$ shift, $6 > 2$ shift, $4 > 2$ shift: $[4, 4, 6, 9] \rightarrow$ place Key at index 0: $[2, 4, 6, 9]$. Fully sorted: `[2, 4, 6, 9]`.

VN **GIẢI THÍCH TIẾNG VIỆT**

Bubble sort: so sánh và đổi chỗ các phần tử liền kề nhiều lượt cho đến khi không còn đổi chỗ nào — độ phức tạp $O(n^2)$. **Insertion sort**: xây dựng phần đã sắp xếp từ đầu danh sách, chèn từng phần tử mới vào đúng vị trí — cũng $O(n^2)$ trong trường hợp xấu nhất nhưng nhanh hơn thực tế với dữ liệu gần như đã sắp xếp.

4.7 Algorithmic efficiency and Big-O notation

Big-O describes how an algorithm's running time (or space use) grows as input size n grows, ignoring constant factors — it measures *scalability*, not raw speed on one specific machine.

$O(1)$	constant	array index access
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	linear search, single loop over n items
$O(n \log n)$	log-linear	efficient sorts (merge sort, quicksort average case)
$O(n^2)$	quadratic	bubble sort, insertion sort, nested loop over n items

Ví dụ mẫu · Worked example

Explain why a single loop that visits every element of an array once is $O(n)$, but a loop nested inside another loop over the same n elements is $O(n^2)$.

A single loop performs one operation for each of the n elements: total work is proportional to n, i.e. $O(n)$. A nested loop performs the inner loop's n operations *for every one* of the n iterations of the outer loop: total work is proportional to $n \times n = n^2$, i.e. $O(n^2)$ — this is exactly the structure of bubble sort's two nested loops, which is why it is quadratic.

(!) Lỗi thường gặp: Two *sequential* (not nested) loops, each visiting all n elements, give $n + n = 2n$ total operations. Big-O drops constant multipliers, so this remains $O(n)$, *not* $O(n^2)$ — only *nested* loops over the same data produce quadratic complexity. Students often mistakenly assume "two loops" automatically means $O(n^2)$.

VN **GIẢI THÍCH TIẾNG VIỆT**

Big-O mô tả tốc độ tăng của thời gian chạy khi kích thước dữ liệu n tăng, KHÔNG phải thời gian chạy thực tế trên một máy cụ thể. Vòng lặp đơn $\rightarrow O(n)$; vòng lặp lồng nhau (2 tầng) trên cùng n phần tử $\rightarrow O(n^2)$; hai vòng lặp NỐI TIẾP (không lồng) $\rightarrow$ vẫn $O(n)$; tìm kiếm nhị phân (chia đôi liên tục) $\rightarrow O(\log n)$.

4.8 Validation, verification and trace tables

Validation is an automatic check applied to data as it is entered, testing whether it is reasonable/-plausible according to a rule (e.g. **range check**, **presence check**, **type check**, **length check**, **format check**). Validation cannot guarantee data is *correct*, only that it is plausible — a **verification** check (e.g. double entry, visual proofreading) is needed to catch data that is valid in form but wrong in fact.

A **trace table** systematically records the value of every relevant variable after each step/iteration of an algorithm, used to manually verify an algorithm's logic or to debug it. Columns are usually the loop/step counter plus one column per variable.

Ví dụ mẫu · Worked example

Complete a trace table for the following pseudocode with input list `[3, 8, 1, 9, 4]`:

```
Max <- List[0]
loop I from 1 to 4
    if List[I] > Max then
        Max <- List[I]
    end if
end loop
output Max
```

I	$List[I]$	*Max* (after check)
—	—	3 (initial)
1	8	8
2	1	8
3	9	9
4	4	9

Final output: `9` (the largest value in the list).

VN **GIẢI THÍCH TIẾNG VIỆT**

Kiểm tra hợp lệ (validation): kiểm tra tự động khi nhập dữ liệu (kiểm tra phạm vi, kiểu, độ dài, định dạng) — chỉ đảm bảo dữ liệu HỢP LÝ, không đảm bảo ĐÚNG. **Kiểm tra xác nhận (verification)**: xác nhận dữ liệu đúng với thực tế (nhập hai lần, kiểm tra bằng mắt). **Bảng vết (trace table)**: ghi lại giá trị từng biến sau mỗi bước để kiểm tra logic thuật toán hoặc gỡ lỗi.

VN **GIẢI THÍCH TIẾNG VIỆT**

Ôn tập nhanh Unit 4: (1) 4 trụ cột tư duy giải thuật: phân rã, trừu tượng hoá, nhận diện mẫu, thiết kế giải thuật; (2) giả mã IB dùng `<-`, `if...end if`, `loop...end loop`; (3) hàm trả về giá trị, thủ tục thì không; (4) linear search $O(n)$ vs binary search $O(\log n)$ (yêu cầu dữ liệu đã sắp xếp); (5) bubble/insertion sort đều $O(n^2)$; (6) vòng lặp lồng nhau $\rightarrow O(n^2)$, vòng lặp nối tiếp vẫn $O(n)$; (7) validation kiểm tra hợp lý, verification kiểm tra đúng thực tế; (8) luôn kiểm tra chỉ số mảng bắt đầu từ 0 hay 1.

4.9 Merge sort: a divide-and-conquer algorithm

Merge sort is a more efficient sorting algorithm than bubble/insertion sort, based on the **divide-and-conquer** strategy: recursively split the list in half until each piece has one element (trivially sorted), then repeatedly **merge** pairs of sorted pieces back together into larger sorted pieces.

Divide-and-conquer solves a problem by: (1) **dividing** it into smaller sub-problems of the same type; (2) **conquering** each sub-problem (often recursively, down to a trivial base case); (3) **combining** the sub-solutions into a solution for the original problem. Merge sort's combine step (merging two already-sorted lists into one sorted list) takes $O(n)$ time at each level, and there are $O(\log n)$ levels of splitting, giving an overall complexity of $O(n \log n)$ — significantly better than bubble/insertion sort's $O(n^2)$ for large n.

Ví dụ mẫu · Worked example

Trace merge sort dividing and merging `[38, 27, 43, 10]`.
Divide: $[38, 27, 43, 10] \rightarrow [38, 27]$ and $[43, 10] \rightarrow$ further divide: $[38], [27]$ and $[43], [10]$ (each a single element, trivially sorted). **Merge level 1:** merge $[38]$ and $[27]$ by comparing front elements: $27 < 38$, so take 27 first, then 38: $[27, 38]$. Merge $[43]$ and $[10]$: $10 < 43$, so $[10, 43]$. **Merge level 2:** merge $[27, 38]$ and $[10, 43]$: compare fronts 27 vs 10: take 10; compare 27 vs 43: take 27; compare 38 vs 43: take 38; only 43 remains: take 43. Result: `[10, 27, 38, 43]` — fully sorted.

VN **GIẢI THÍCH TIẾNG VIỆT**

Merge sort: chiến lược **chia để trị** — chia danh sách làm đôi liên tục đến khi mỗi phần chỉ còn 1 phần tử, rồi **trộn (merge)** từng cặp danh sách đã sắp xếp lại với nhau. Độ phức tạp $O(n \log n)$ — nhanh hơn nhiều so với bubble/insertion sort $O(n^2)$ khi n lớn, nhờ cấu trúc chia-và-trộn hiệu quả.

4.10 Complex conditionals and logical operators

IB pseudocode combines conditions using `AND`, `OR`, and `NOT`, following standard Boolean logic (Unit 2). Complex nested `if` statements let a program branch based on multiple combined conditions.

Ví dụ mẫu · Worked example

Trace this pseudocode, which classifies a triangle by its three side lengths `A`, `B`, `C`, for `A=5, B=5, C=5`:

```
if A = B AND B = C then
    output "Equilateral"
else if A = B OR B = C OR A = C then
    output "Isosceles"
else
    output "Scalene"
end if
```

With $A = 5, B = 5, C = 5$: the first condition `A = B AND B = C` evaluates `5=5 AND 5=5`, both true, so the whole `AND` expression is `TRUE`. The first branch executes, outputting `"Equilateral"`; the remaining `else if/else` branches are skipped entirely once a matching branch has run.

(!) Lỗi thường gặp: In a chain of `if...else if...else if...else`, only the *first* branch whose condition is true ever executes — order matters. Placing a more general condition before a more specific one can silently prevent the specific case from ever being reached; always check conditions from most specific to most general when they overlap.

VN **GIẢI THÍCH TIẾNG VIỆT**

Giả mã IB kết hợp điều kiện bằng `AND`, `OR`, `NOT` theo logic Boole. Trong chuỗi `if...else if...else`, CHỈ nhánh điều kiện ĐÚNG ĐẦU TIÊN được thực thi — thứ tự rất quan trọng, đặt điều kiện tổng quát trước điều kiện cụ thể có thể khiến nhánh cụ thể không bao giờ được chạy tới.

4.11 String algorithms

Beyond basic `LEFT/RIGHT/MID`, several standard algorithms operate on strings by processing them character by character.

Ví dụ mẫu · Worked example

Write pseudocode to check whether a string `Word` is a **palindrome** (reads the same forwards and backwards), and trace it for `Word = "LEVEL"`.

```
IsPalindrome <- TRUE
Left <- 0
Right <- LENGTH(Word) - 1
loop while Left < Right
    if MID(Word, Left, 1) <> MID(Word, Right, 1) then
        IsPalindrome <- FALSE
    end if
    Left <- Left + 1
    Right <- Right - 1
end loop
output IsPalindrome
```

`"LEVEL"` has length 5 (indices 0–4): L(0) E(1) V(2) E(3) L(4). Left=0,Right=4: compare L,L — match. Left=1,Right=3: compare E,E — match. Left=2,Right=2: loop condition `Left < Right` is now false ($2 < 2$ is false), loop ends. `IsPalindrome` remained `TRUE` throughout: output `TRUE` — correctly identifying ”LEVEL” as a palindrome.

Ví dụ mẫu · Worked example

Write pseudocode to reverse a string `Word` character by character, building a new string `Reversed`, and trace it for `Word = "CAT"`.

```
Reversed <- ""
loop I from LENGTH(Word) - 1 to 0 step -1
    Reversed <- Reversed + MID(Word, I, 1)
end loop
output Reversed
```

`"CAT"` has indices C(0) A(1) T(2), length 3. $I = 2$: `Reversed` = ”” + ”T” = ”T”. $I = 1$: `Reversed` = ”T” + ”A” = ”TA”. $I = 0$: `Reversed` = ”TA” + ”C” = ”TAC”. Output: `"TAC"` — the reverse of ”CAT”.

VN **GIẢI THÍCH TIẾNG VIỆT**

Kiểm tra đối xứng (palindrome): so sánh ký tự đầu và cuối, tiến dần vào giữa. **Đảo chuỗi**: duyệt từ ký tự cuối về đầu, nối từng ký tự vào chuỗi kết quả. Đây là các thuật toán xử lý chuỗi

kinh điển thường xuất hiện trong đề thi lập trình.

4.12 2D array algorithms

Beyond simple summation, 2D arrays support richer algorithms that process rows, columns, or the whole grid in specific patterns.

Ví dụ mẫu · Worked example

A 3×3 grid `Matrix[0..2][0..2]` $= \{\{1,2,3\},\{4,5,6\},\{7,8,9\}\}$ must be **transposed** (rows become columns) into a new grid `Result`. Write pseudocode and trace it.

```
loop R from 0 to 2
    loop C from 0 to 2
        Result[C][R] <- Matrix[R][C]
    end loop
end loop
```

Tracing: R=0: C=0: $Result[0][0]$=1; C=1: $Result[1][0]$=2; C=2: $Result[2][0]$=3. R=1: $Result[0][1]$=4; $Result[1][1]$=5; $Result[2][1]$=6. R=2: $Result[0][2]$=7; $Result[1][2]$=8; $Result[2][2]$=9. Final `Result` $= \{\{1,4,7\},\{2,5,8\},\{3,6,9\}\}$ — each original row has become a column, the definition of a matrix transpose.

Ví dụ mẫu · Worked example

Write pseudocode to find the row with the largest row-total in a 2D array `Grid[0..R-1][0..C-1]`, and trace it for `Grid` $= \{\{2,2,2\},\{9,1,1\},\{4,4,4\}\}$ (3 rows, 3 columns).

```
BestRow <- 0
BestTotal <- 0
loop Row from 0 to R - 1
    RowTotal <- 0
    loop Col from 0 to C - 1
        RowTotal <- RowTotal + Grid[Row][Col]
    end loop
    if RowTotal > BestTotal then
        BestTotal <- RowTotal
        BestRow <- Row
    end if
end loop
output BestRow
```

Row 0 total: $2+2+2=6>0$, so `BestTotal <- 6, BestRow <- 0`. Row 1 total: $9+1+1=11>6$, so `BestTotal <- 11, BestRow <- 1`. Row 2 total: $4+4+4=12>11$, so `BestTotal <- 12, BestRow <- 2`. Output: 2 (row index 2 has the largest total, 12).

VN **GIẢI THÍCH TIẾNG VIỆT**

Chuyển vị ma trận (transpose): hàng thành cột, dùng vòng lặp lồng nhau `Result[C][R] <- Matrix[R][C]`. **Tìm hàng/cột có tổng lớn nhất**: tính tổng từng hàng trong vòng lặp trong, so sánh với giá trị lớn nhất đã lưu, cập nhật nếu tìm thấy giá trị lớn hơn — mẫu thuật toán "theo

dõi giá trị tốt nhất" (best-so-far) rất phổ biến.

4.13 Constructing a systematic test plan for an algorithm

Just as Unit 1 introduced normal/boundary/erroneous *data* for testing whole systems, individual *algorithms* should be tested systematically against a deliberately chosen set of cases before being trusted.

A good algorithm test plan includes: (1) a **typical/normal** case with ordinary input; (2) **boundary** cases (e.g. an empty list, a list of exactly one element, the target being the first or last element); (3) cases exercising *every* distinct logical branch (e.g. for a search algorithm: target present, and target absent); (4) cases that could expose classic bugs (e.g. duplicate values, all-identical values, already-sorted or reverse-sorted input for a sorting algorithm).

Ví dụ mẫu · Worked example

Design a test plan (a list of test cases with expected outputs, not full traces) for the `Max`-finding algorithm from Unit 4's trace-table example (which outputs the largest value in a list).

Test case	Expected output / purpose
`[3, 8, 1, 9, 4]` (normal case)	`9` — typical mixed values
`[5]` (single element, boundary)	`5` — smallest possible valid list
`[-2, -8, -1]` (all negative)	`-1` — checks the algorithm does not wrongly assume 0 is a safe starting value
`[7, 7, 7]` (all identical)	`7` — checks ties are handled without error
`[1, 2, 3, 4, 5]` (already ascending)	`5` — maximum correctly found even at the very end

Systematically covering these five cases would catch, for example, the classic "initialise Max to 0" bug identified earlier in this chapter, since the all-negative case would expose it immediately.

VN **GIẢI THÍCH TIẾNG VIỆT**

Kế hoạch kiểm thử thuật toán nên bao gồm: trường hợp bình thường, trường hợp biên (danh sách rỗng/1 phần tử), các nhánh logic khác nhau (tìm thấy/không tìm thấy), và các trường hợp dễ gây lỗi kinh điển (giá trị trùng lặp, toàn giá trị âm, dữ liệu đã sắp xếp sẵn). Kiểm thử có hệ thống giúp phát hiện lỗi như "khởi tạo Max bằng 0" đã nêu trước đó trong chương này.

4.14 Selection sort

Selection sort repeatedly finds the *smallest* remaining unsorted element and swaps it into its correct sorted position, growing a sorted section from the front of the list one element at a time.

Ví dụ mẫu · Worked example

Selection sort pseudocode and a full trace of `[29, 10, 14, 37]`:

```
loop I from 0 to N - 2
```

```
    MinIndex <- I
    loop J from I + 1 to N - 1
        if List[J] < List[MinIndex] then
            MinIndex <- J
        end if
    end loop
    Temp <- List[I]
    List[I] <- List[MinIndex]
    List[MinIndex] <- Temp
end loop
```

$I = 0$: scan indices 1–3 for the smallest: $10 < 29$ (MinIndex→1), 14 not< 10, 37 not< 10. Swap index 0 and 1: $[10, 29, 14, 37]$. $I = 1$: scan indices 2–3: $14 < 29$ (MinIndex→2), 37 not< 14. Swap index 1 and 2: $[10, 14, 29, 37]$. $I = 2$: scan index 3: 37 not< 29. No swap needed: $[10, 14, 29, 37]$. Fully sorted: `[10, 14, 29, 37]`.

Selection sort always performs exactly the same number of *comparisons* regardless of the input's initial order (it always fully scans the remaining unsorted section each pass), giving $O(n^2)$ in the best, average, and worst case alike. This differs from bubble sort (which can exit early if a pass makes no swaps) and insertion sort (whose inner loop can exit early for nearly-sorted data) — selection sort's number of *swaps*, however, is much lower than bubble sort's (at most one swap per outer-loop pass, rather than potentially many), which can matter when swapping large data records is expensive.

VN **GIẢI THÍCH TIẾNG VIỆT**

Selection sort: mỗi lượt tìm phần tử NHỎ NHẤT trong phần chưa sắp xếp rồi đưa vào đúng vị trí đầu phần chưa sắp — xây dựng phần đã sắp xếp từ đầu danh sách. Luôn thực hiện CÙNG số lần so sánh bất kể dữ liệu đầu vào ($O(n^2)$ mọi trường hợp), nhưng số lần ĐỔI CHỖ (swap) ít hơn nhiều so với bubble sort — có lợi khi việc đổi chỗ bản ghi lớn tốn kém.

4.15 Modular design: a worked multi-function program

Modular design decomposes a program into independent subroutines, each handling one clear responsibility, then composes them together to solve the overall problem — directly applying computational thinking's **decomposition** principle (Unit 4, earlier) to actual code structure.

Ví dụ mẫu · Worked example

Design a modular program to compute the average of only the *passing* scores (score ≥ 50) in a list, using separate subroutines for filtering and averaging.

```
method FilterPassing(Scores)
    Passing <- []
    loop I from 0 to LENGTH(Scores) - 1
        if Scores[I] >= 50 then
            Passing <- Passing + [Scores[I]]
        end if
    end loop
    return Passing
end method
```

```
method CalculateAverage(List)
    Total <- 0
    loop I from 0 to LENGTH(List) - 1
        Total <- Total + List[I]
    end loop
    return Total / LENGTH(List)
end method

Scores <- [45, 62, 78, 30, 91, 55]
PassingScores <- FilterPassing(Scores)
output CalculateAverage(PassingScores)
```

`FilterPassing` returns `[62, 78, 91, 55]` (excluding 45 and 30, both below 50). `CalculateAverage` then computes $(62 + 78 + 91 + 55)/4 = 286/4 = 71.5$. Output: `71.5`. Each subroutine can be independently tested (e.g. testing `FilterPassing` alone with a list of all-failing scores, expecting an empty result) before combining them — a key benefit of modular design highlighted already in Unit 1's discussion of unit testing.

VN **GIẢI THÍCH TIẾNG VIỆT**

Thiết kế module hoá: chia chương trình thành các chương trình con độc lập, mỗi cái đảm nhận một trách nhiệm rõ ràng, rồi kết hợp lại để giải quyết bài toán tổng thể — áp dụng trực tiếp nguyên tắc **phân rã** vào cấu trúc mã nguồn thực tế. Mỗi chương trình con có thể được kiểm thử ĐỘC LẬP trước khi kết hợp — đây là lợi ích cốt lõi liên hệ với kiểm thử đơn vị (unit testing) đã học ở Unit 1.

4.16 Understanding exam command words

IB CS exam questions use specific **command words** that indicate precisely how much detail and what type of response is required — misreading the command word is a common, entirely avoidable source of lost marks.

Command word	What is required
State/Identify	A brief answer with no explanation required (e.g. naming an algorithm, a value, a term)
Describe	Give a detailed account, in words, of a process/structure/situation (no justification needed)
Explain	Give a detailed account including *reasons or causes* — "describe *and* justify why"
Compare	Give the similarities *and* differences between two or more things
Distinguish	Focus specifically on the *differences* between two things
Evaluate	Weigh up strengths and weaknesses to reach a balanced, justified conclusion
Outline	Give a brief account of the main points, without going into full explanatory detail

Ví dụ mẫu · Worked example

A question asks "**Explain** why binary search is more efficient than linear search for large sorted datasets." Contrast a response that would only satisfy "**Describe**" with one that fully satisfies "**Explain**."

A **Describe**-level response might state: "Binary search repeatedly checks the middle element and eliminates half the remaining list each time, while linear search checks each element in turn" — this describes *what* each algorithm does, but does not yet address *why* one is more efficient. A full **Explain**-level response must additionally give the *reason*: "...because eliminating half the remaining data at each step means the number of comparisons grows only logarithmically ($O(\log n)$) with the dataset size, whereas linear search's comparisons grow directly in proportion to the dataset size ($O(n)$) — so for large n, binary search requires dramatically fewer comparisons." Only this second version connects the described behaviour to the underlying *reason* it results in better efficiency, which is what "Explain" specifically requires.

VN **GIẢI THÍCH TIẾNG VIỆT**

Các từ khoá lệnh (command words) trong đề thi IB CS quyết định mức độ chi tiết cần viết: **State/Identify** (nêu ngắn gọn, không cần giải thích), **Describe** (mô tả chi tiết, không cần lý do), **Explain** (mô tả VÀ giải thích LÝ DO/NGUYÊN NHÂN), **Compare** (nêu cả điểm giống VÀ khác), **Distinguish** (tập trung vào điểm KHÁC), **Evaluate** (cân nhắc ưu/nhược điểm để đưa ra kết luận có cơ sở). Đọc sai từ khoá lệnh là nguyên nhân phổ biến và hoàn toàn có thể tránh được khi bị mất điểm.

4.17 Practice

Bài tập luyện tập

An array of 1000 sorted elements is searched using binary search. Approximately how many comparisons are needed in the worst case? ($2^{10} = 1024$.)

(A) 10
(B) 100
(C) 500
(D) 1000

Lời giải chi tiết

Binary search is $O(\log_2 n)$. Since $2^{10} = 1024 > 1000$, at most 10 comparisons are needed in the worst case. **(A)**.

Bài tập luyện tập

Trace this pseudocode and state the final output.

```
Result <- 1
loop K from 1 to 4
    Result <- Result * K
end loop
output Result
```

Lời giải chi tiết

$K = 1$: Result $= 1 \times 1 = 1$. $K = 2$: Result $= 1 \times 2 = 2$. $K = 3$: Result $= 2 \times 3 = 6$. $K = 4$: Result $= 6 \times 4 = 24$. Final output: **24** (this computes $4! = 24$).

Bài tập luyện tập

Which algorithm is more appropriate for searching an unsorted list of 50 unique names for a match, and what is its time complexity?

(A) Binary search, $O(\log n)$
(B) Linear search, $O(n)$
(C) Bubble sort, $O(n^2)$
(D) Binary search, $O(n)$

Lời giải chi tiết

Binary search requires **sorted** data; the list is unsorted, so **linear search** must be used, checking each element in turn: $O(n)$. **(B)**.

Bài tập luyện tập

Rewrite the following buggy pseudocode, which should output the largest value in `List[0..N-1]`, and explain the bug.

```
Max <- 0
loop I from 0 to N - 1
    if List[I] > Max then
        Max <- List[I]
    end if
end loop
output Max
```

Lời giải chi tiết

Bug: initialising `Max <- 0` fails if every value in the list is negative (the true maximum would never replace 0). **Fix:** initialise `Max` to the *first* element instead:

```
Max <- List[0]
loop I from 1 to N - 1
    if List[I] > Max then
        Max <- List[I]
    end if
end loop
output Max
```

Bài tập luyện tập

A single loop that visits every element of an array once is $O(n)$. What is the complexity of two consecutive (not nested) loops, each visiting every element once?

(A) $O(n^2)$
(B) $O(2n)$, which simplifies to $O(n)$
(C) $O(\log n)$
(D) $O(1)$

Lời giải chi tiết

Two *sequential* loops give $n + n = 2n$ operations — Big-O drops constant factors, so this is still $O(n)$, not $O(n^2)$ (which requires *nested* loops). **(B)**.

Bài tập luyện tập

Trace the following pseudocode for `Word = "PYTHON"` and state the output.

```
Result <- ""
loop I from 0 to LENGTH(Word) - 1
    Letter <- MID(Word, I, 1)
    if Letter = "O" then
        Result <- Result + "0"
    else
        Result <- Result + Letter
    end if
end loop
output Result
```

Lời giải chi tiết

`"PYTHON"` has letters P-Y-T-H-O-N. Every letter is copied unchanged except `O` (index 4), which is replaced by `"0"` (zero). Result built up: P, PY, PYT, PYTH, PYTH0, PYTH0N. Output: `"PYTH0N"`.

Bài tập luyện tập

Trace bubble sort's first full pass only on `[6, 1, 3, 9, 2]`, showing the array after each swap.

Lời giải chi tiết

Compare $(6, 1)$: swap $\rightarrow [1, 6, 3, 9, 2]$. Compare $(6, 3)$: swap $\rightarrow [1, 3, 6, 9, 2]$. Compare $(6, 9)$: no swap. Compare $(9, 2)$: swap $\rightarrow [1, 3, 6, 2, 9]$. After pass 1: `[1, 3, 6, 2, 9]` — note the largest value, 9, has correctly "bubbled" to its final position at the end of the list, as expected after one full pass.

Bài tập luyện tập

Write a function `Square` that takes one numeric parameter and returns its square, then trace `output Square(5) + Square(3)`.

Lời giải chi tiết

```
method Square(N)
    return N * N
end method
```

`Square(5)` returns 25; `Square(3)` returns 9. `output Square(5) + Square(3)` evaluates to $25 + 9 = 34$.

Bài tập luyện tập

An input field for "month of birth" only accepts integers 1–12. Name the type of validation check this requires, and give a valid piece of erroneous test data.

Lời giải chi tiết

This requires a **range check** (confirming the value falls within an acceptable numeric range). Erroneous test data: a non-numeric value such as `"May"`, or an out-of-range integer such as `13` or `0` (boundary-erroneous), should both be rejected by the validation.

Bài tập luyện tập

Explain why validation alone cannot guarantee that a user's entered date of birth is actually correct, using an example.

Lời giải chi tiết

Validation only checks that data is *plausible* in form (e.g. a valid calendar date, within a sensible range), not that it is factually true. For example, a user could validly enter `01/01/2000` as their date of birth — this passes every format and range check — even if their real birth date is different; only independent **verification** (e.g. checking against an official ID document) could catch this kind of error, since it looks perfectly valid in form.

Bài tập luyện tập

Trace this pseudocode for `List = [10, 20, 30]` and `Grid` is not used; identify what the algorithm computes.

```
Count <- 0
Sum <- 0
loop I from 0 to LENGTH(List) - 1
    Sum <- Sum + List[I]
    Count <- Count + 1
end loop
Average <- Sum / Count
output Average
```

Lời giải chi tiết

$I = 0$: Sum$= 10$, Count$= 1$. $I = 1$: Sum$= 30$, Count$= 2$. $I = 2$: Sum$= 60$, Count$= 3$. Average $= 60/3 = 20$. The algorithm computes the **arithmetic mean** of the list, output: 20.

Bài tập luyện tập

Explain, using decomposition, how you would approach designing a program that manages a school's exam timetable (rooms, times, invigilators, no student clashes).

Lời giải chi tiết

Decomposition breaks the overall problem into independent, more manageable sub-problems, such as: (1) collecting the list of exams and enrolled students for each; (2) assigning rooms

based on capacity; (3) assigning time slots so no student has two exams simultaneously; (4) assigning invigilators to each room/slot without double-booking staff; (5) validating the final timetable for any remaining clashes. Each sub-problem can be designed, coded, and tested largely independently before being combined, which is far more manageable than attempting to solve the entire timetabling problem as one single, monolithic algorithm.

Bài tập luyện tập

Trace a linear search for `Target = 99` in `[4, 15, 8, 23]` (indices 0–3) and state how many comparisons are made before the search concludes the value is absent.

Lời giải chi tiết

Index 0: $4 \neq 99$. Index 1: $15 \neq 99$. Index 2: $8 \neq 99$. Index 3: $23 \neq 99$. All 4 elements checked with no match; the search concludes 99 is **not present** after **4** comparisons (the worst case for linear search on a list of $n = 4$ elements is exactly n comparisons).

Bài tập luyện tập

Explain why binary search cannot be correctly applied to the unsorted list `[15, 3, 42, 8, 27]` without first sorting it, using one concrete failed step as evidence.

Lời giải chi tiết

Binary search assumes that if the middle element is greater than the target, the target (if present) must lie in the *left* half, and vice versa — this assumption only holds if the array is sorted. In `[15, 3, 42, 8, 27]`, searching for `Target = 8`: the middle element (index 2) is 42, which is greater than 8, so binary search would incorrectly discard the *right* half and search only `[15, 3]` — but the actual target, 8, is in the discarded right half. The search would incorrectly report the value as absent.

Bài tập luyện tập

A program declares a variable `Total` inside a function `CalculateSum`, and a completely separate variable also named `Total` outside all functions (global scope). Explain what happens when `CalculateSum` modifies its own `Total`.

Lời giải chi tiết

Because `Total` inside `CalculateSum` is a **local** variable, it exists only within that function's scope and is entirely distinct from the **global** `Total` declared outside — even though they share the same name. Modifying the local `Total` inside the function has **no effect whatsoever** on the global `Total`; each is a separate storage location, and the local one ceases to exist once the function call ends.

Bài tập luyện tập

Insertion sort is often preferred over bubble sort for data that is "almost sorted already." Explain why, referencing what each algorithm does when it encounters elements already in order.

Lời giải chi tiết

Insertion sort's inner loop only shifts elements while they are strictly greater than the current key — if a section of the list is already sorted, the inner loop for those elements exits almost immediately (very few comparisons/shifts), making the algorithm run close to $O(n)$ in the best case (nearly sorted input). **Bubble sort**, in its simplest form, always performs a fixed number of nested-loop comparisons regardless of how sorted the data already is (unless an early-exit "no swaps made" flag is added), so it does not automatically benefit from data being nearly sorted in the same way.

Bài tập luyện tập

A 2D array `Seating[0..2][0..3]` represents a small cinema (3 rows, 4 seats each) where `1` means occupied and `0` means free. Write pseudocode to count the number of free seats.

Lời giải chi tiết

```
FreeCount <- 0
loop R from 0 to 2
    loop C from 0 to 3
        if Seating[R][C] = 0 then
            FreeCount <- FreeCount + 1
        end if
    end loop
end loop
output FreeCount
```

The nested loop visits every one of the $3 \times 4 = 12$ seats exactly once, incrementing `FreeCount` for each seat marked free (`0`).

Bài tập luyện tập

Explain the difference between a **presence check** and a **format check**, giving one example field for each.

Lời giải chi tiết

A **presence check** simply confirms that a required field has *not* been left empty (e.g. an "Email address" field on a registration form must not be blank). A **format check** confirms that entered data follows a specific expected pattern/structure regardless of whether it is present (e.g. an email field must contain an `@` symbol and a domain, such as `name@example.com`, not just any non-empty text).

Bài tập luyện tập

Insertion sort is run on `[5, 5, 5, 5]` (all equal values). State how many swaps/shifts occur during the whole sort, and explain why.

Lời giải chi tiết

Zero shifts occur. Insertion sort's inner loop only shifts an element when the previous element is *strictly greater than* (>) the current key; since every value is equal, the condition `List[J] >`

`Key` is never true for any comparison, so the inner loop exits immediately at every step of the outer loop — the array is confirmed already in order without a single element needing to move.

Bài tập luyện tập

Explain, with reference to Big-O notation, why an algorithm that is $O(n^2)$ can still run faster in practice than an $O(n \log n)$ algorithm for small input sizes, even though Big-O predicts the opposite for large n.

Lời giải chi tiết

Big-O describes the *growth rate* of running time as n becomes large, ignoring constant factors and lower-order terms — but for small n, those ignored constants can dominate. An $O(n^2)$ algorithm with very small constant overhead (e.g. simple operations, no extra memory allocation) may outperform an $O(n \log n)$ algorithm that has larger constant overhead (e.g. the overhead of recursive calls or merging in merge sort) when n is small enough that n^2 has not yet grown large relative to $n \log n$. As n increases beyond some **crossover point**, the $O(n \log n)$ algorithm will always eventually become faster, which is exactly what Big-O guarantees for sufficiently large n.

Bài tập luyện tập

Trace merge sort's divide and merge steps for `[8, 3, 5, 1]`, showing the final sorted result.

Lời giải chi tiết

Divide: $[8, 3, 5, 1] \rightarrow [8, 3]$ and $[5, 1] \rightarrow$ further: $[8], [3]$ and $[5], [1]$. **Merge level 1:** merge $[8]$ and $[3]$: $3 < 8$, so $[3, 8]$. Merge $[5]$ and $[1]$: $1 < 5$, so $[1, 5]$. **Merge level 2:** merge $[3, 8]$ and $[1, 5]$: compare 3 vs 1: take 1; compare 3 vs 5: take 3; compare 8 vs 5: take 5; only 8 remains: take 8. Result: `[1, 3, 5, 8]`.

Bài tập luyện tập

Explain why merge sort's $O(n \log n)$ complexity makes it a better choice than bubble sort's $O(n^2)$ for sorting one million records, using approximate operation counts.

Lời giải chi tiết

For $n = 1{,}000{,}000$: bubble sort's $O(n^2)$ requires roughly $n^2 = 10^{12}$ (one trillion) operations in the worst case. Merge sort's $O(n \log n)$ requires roughly $n \log_2 n \approx 1{,}000{,}000 \times 20 = 2 \times 10^7$ (twenty million) operations, since $\log_2(1{,}000{,}000) \approx 20$. Merge sort therefore needs roughly **50,000 times fewer** operations for this input size — a difference between a sort completing in a fraction of a second versus one that could take hours or longer.

Bài tập luyện tập

Rewrite the triangle-classification pseudocode from this chapter so that it correctly still identifies an equilateral triangle even if the conditions were mistakenly reordered to check for "Isosceles" first. Explain what bug this fixes.

Lời giải chi tiết

If reordered as:

```
if A = B OR B = C OR A = C then
    output "Isosceles"
else if A = B AND B = C then
    output "Equilateral"
else
    output "Scalene"
end if
```

This is buggy: for an equilateral triangle ($A = B = C$), the *first* condition (`A=B OR B=C OR A=C`) is already true, so the program would incorrectly output "Isosceles" and the second, more specific "Equilateral" branch would never be reached. The **fix** is to always test the most *specific* condition (Equilateral, requiring *all* sides equal) *before* the more general condition (Isosceles, requiring only *some* sides equal), exactly as in the chapter's original version — general conditions must always come after, not before, more specific overlapping ones.

Bài tập luyện tập

Trace the palindrome-checking pseudocode from this chapter for `Word = "HELLO"` and state the final output.

Lời giải chi tiết

`"HELLO"` has indices H(0) E(1) L(2) L(3) O(4), length 5. Left=0,Right=4: compare H,O — mismatch, so `IsPalindrome <- FALSE`. Left=1,Right=3: compare E,L — mismatch (IsPalindrome stays FALSE, already set). Left=2,Right=2: loop condition `Left<Right` is false ($2 < 2$), loop ends. Output: `FALSE` — correctly, since "HELLO" reversed is "OLLEH", not the same as the original.

Bài tập luyện tập

Trace the string-reversal pseudocode from this chapter for `Word = "CODE"` and state the final output.

Lời giải chi tiết

`"CODE"` has indices C(0) O(1) D(2) E(3), length 4. $I = 3$: Reversed $=$ "" $+$ "E" $=$ "E". $I = 2$: Reversed $=$ "E"+"D"="ED". $I = 1$: Reversed="ED"+"O"="EDO". $I = 0$: Reversed="EDO"+"C"="EDOC". Output: `"EDOC"`.

Bài tập luyện tập

Explain why merge sort requires additional memory (temporary arrays to hold merged results) in a way that bubble sort or insertion sort does not, and state whether this is a meaningful trade-off in most modern applications.

Lời giải chi tiết

Merge sort's merge step combines two separately-sorted sub-lists into one sorted list by comparing their fronts one at a time — this naturally produces a *new* list, requiring temporary extra

memory space roughly proportional to n (this is called **not in-place**). Bubble sort and insertion sort, by contrast, rearrange elements *within the original array* using only a small constant amount of extra memory (a single temporary variable for swapping), making them **in-place**. In most modern applications with ample available RAM, this extra $O(n)$ memory use is a very reasonable trade-off given merge sort's vastly better $O(n \log n)$ time complexity on large datasets — memory is comparatively cheap, while time is often the more critical constraint.

Bài tập luyện tập

A question asks students to "**Compare** bubble sort and merge sort." Explain why an answer that only lists merge sort's advantages, without mentioning bubble sort at all, would not fully satisfy this command word.

Lời giải chi tiết

"**Compare**" specifically requires identifying *both* similarities *and* differences between the two things named, not simply describing the strengths of one of them in isolation. An answer only listing merge sort's advantages addresses at most one direction of the comparison (why merge sort might be preferred) but omits any direct comparison back to bubble sort's own characteristics (e.g. bubble sort's simplicity, lower memory use, or comparable performance on tiny/already-sorted datasets) — a full "Compare" response should explicitly relate the two algorithms to each other (e.g. "unlike bubble sort's $O(n^2)$..., merge sort achieves $O(n \log n)$, though bubble sort requires no extra memory while merge sort does").

Bài tập luyện tập

Rewrite the instruction "**Describe** how a stack operates" as a full "**Explain**"-level question, and outline what additional content the "Explain" version would require in the answer.

Lời giải chi tiết

An "Explain"-level version might read: "**Explain** why a stack is well suited to implementing an 'undo' feature in a text editor." The "Describe" version only requires outlining *how* a stack works (LIFO, push/pop at the top) without needing to justify anything. The "Explain" version additionally requires connecting this behaviour to the specific *reason* it suits the undo feature — namely, that undo must reverse the *most recently performed* action first, which is exactly the LIFO order a stack naturally provides, so the answer must state *why* this matching of behaviours matters, not merely describe the stack's operations in isolation.

Bài tập luyện tập

Trace the following pseudocode for `List = [12, 45, 7, 23, 8]` and state the final output.

```
Count <- 0
loop I from 0 to LENGTH(List) - 1
    if List[I] MOD 2 = 0 then
        Count <- Count + 1
    end if
end loop
output Count
```

Lời giải chi tiết

Checking each value's remainder when divided by 2 (even if remainder is 0): $12 \bmod 2 = 0$ (even, Count→1); $45 \bmod 2 = 1$ (odd); $7 \bmod 2 = 1$ (odd); $23 \bmod 2 = 1$ (odd); $8 \bmod 2 = 0$ (even, Count→2). Final output: **2** — the list contains exactly 2 even numbers (12 and 8).

Bài tập luyện tập

A "Describe" question asks: "Describe how insertion sort works." A student instead writes only "Insertion sort is efficient for nearly-sorted data because its inner loop exits early." Explain what is missing from this answer relative to the command word used.

Lời giải chi tiết

The student's answer focuses entirely on *why* insertion sort performs well on certain data (a justification, appropriate for an "Explain" command word) but never actually **describes** the algorithm's mechanism itself — how it builds a sorted section from the front of the list, taking each new element (the "key") and shifting larger already-sorted elements rightward until the key's correct position is found. A "Describe" response needs to communicate *what the algorithm does, step by step*, not jump straight to a justification of its performance characteristics — the student has effectively answered a different, unasked "Explain" question instead.

Bài tập luyện tập

Write pseudocode for a function `CountVowels` that counts how many vowels (A, E, I, O, U, case-insensitive is not required — assume all uppercase) appear in a string, and trace it for `"ALGORITHM"`.

Lời giải chi tiết

```
method CountVowels(Word)
    Count <- 0
    loop I from 0 to LENGTH(Word) - 1
        Letter <- MID(Word, I, 1)
        IsVowel <- Letter = "A" OR Letter = "E" OR Letter = "I"
        IsVowel <- IsVowel OR Letter = "O" OR Letter = "U"
        if IsVowel then
            Count <- Count + 1
        end if
    end loop
    return Count
end method
```

`"ALGORITHM"`: A-L-G-O-R-I-T-H-M. Vowels present: A (index 0), O (index 3), I (index 5) — three vowels. `CountVowels("ALGORITHM")` returns **3**.

Bài tập luyện tập

Trace the following pseudocode, which searches for the *second-largest* value in `List = [15, 42, 8, 42, 30]`, and state the final output.

```
Largest <- List[0]
Second <- List[0]
```

```
loop I from 1 to LENGTH(List) - 1
    if List[I] > Largest then
        Second <- Largest
        Largest <- List[I]
    else if List[I] > Second AND List[I] <> Largest then
        Second <- List[I]
    end if
end loop
output Second
```

Lời giải chi tiết

Initial: Largest$=15$, Second$=15$. I=1: $42>15$, so Second$\to 15$(old Largest), Largest$\to 42$. I=2: 8 not>42; is $8>15$ AND $8\neq 42$? No. No change. I=3: 42 not>42 (equal, not strictly greater); is $42>15$ AND $42\neq 42$? Second condition fails since $42=42$ (the <>Largest check correctly excludes counting a duplicate of the largest value as the second-largest). No change. I=4: 30 not>42; is $30>15$ AND $30\neq 42$? Yes — Second$\to 30$. Final output: **30** — correctly the second-largest *distinct* value, ignoring the duplicate 42.

Bài tập luyện tập

Trace the matrix-transpose pseudocode from this chapter on the 2×2 grid `Matrix[0..1][0..1]` $=\{\{5,6\},\{7,8\}\}$ and state the final `Result` grid.

Lời giải chi tiết

R=0: C=0: $Result[0][0]=Matrix[0][0]=5$; C=1: $Result[1][0]=Matrix[0][1]=6$. R=1: $Result[0][1]=Matrix[1][0]=7$; $Result[1][1]=Matrix[1][1]=8$. Final `Result` $=\{\{5,7\},\{6,8\}\}$ — row 0 (5,6) became column 0, and row 1 (7,8) became column 1.

Bài tập luyện tập

Design a test plan (list of test cases with brief justification, not full traces) for the linear search algorithm from this chapter, ensuring both possible outcomes (found / not found) and at least one boundary case are covered.

Lời giải chi tiết

A suitable test plan: (1) **target present, middle of list** — e.g. search for an element known to be at index 2 of a 5-element list, confirms basic correct-match detection; (2) **target present, first element** (boundary) — confirms the search does not skip index 0; (3) **target present, last element** (boundary) — confirms the loop does not terminate one element too early (an off-by-one check); (4) **target absent** — confirms the algorithm correctly reports "not found" rather than looping forever or crashing; (5) **single-element list** (boundary) — the smallest valid input, checked both with the target present and target absent.

Bài tập luyện tập

Explain why testing a sorting algorithm only with already-randomly-shuffled data is insufficient, and name one additional input pattern that should also be tested.

Lời giải chi tiết

Randomly shuffled data alone may never exercise certain edge-case code paths that only appear with specific patterns — for example, some sorting algorithm implementations behave differently (or contain bugs) specifically when the input is **already fully sorted** or **sorted in reverse order**, since these represent the best-case and worst-case scenarios respectively for several algorithms' number of comparisons/swaps. A thorough test plan should explicitly include an **already-sorted** list and a **reverse-sorted** list as additional targeted test cases, alongside random data, to confirm correct behaviour (and reveal performance characteristics) at both extremes, not just the "typical" shuffled case.

Bài tập luyện tập

Trace selection sort's full execution on `[8, 3, 6, 1]`, showing the array after every swap.

Lời giải chi tiết

$I = 0$: scan indices 1–3 for smallest: $3 < 8$ (MinIndex→1), 6 not< 3, $1 < 3$ (MinIndex→3). Swap index 0 and 3: $[1, 3, 6, 8]$. $I = 1$: scan indices 2–3: 6 not< 3, 8 not< 3. MinIndex stays 1, no swap: $[1, 3, 6, 8]$. $I = 2$: scan index 3: 8 not< 6. No swap: $[1, 3, 6, 8]$. Fully sorted: `[1, 3, 6, 8]`.

Bài tập luyện tập

Explain why selection sort always performs the same number of comparisons for a given list size n, regardless of whether the input is already sorted, while insertion sort's number of operations can vary greatly depending on the input order.

Lời giải chi tiết

Selection sort's inner loop always scans the *entire* remaining unsorted section on every pass to find the minimum, regardless of whether that section happens to already be in order — there is no mechanism to detect or exploit existing order, so the comparison count is fixed by n alone. **Insertion sort**'s inner loop instead exits as soon as it finds an element that does not need to shift (i.e. the previous element is not greater than the current key) — for already-sorted input, this means the inner loop exits almost immediately on every outer iteration, giving far fewer total operations than for randomly-ordered or reverse-sorted input, where many shifts are needed.

Bài tập luyện tập

Redesign the modular scores program from this chapter to add a third subroutine, `CountFailing`, that returns how many scores were below 50, without modifying `FilterPassing` or `CalculateAverage`.

Lời giải chi tiết

```
method CountFailing(Scores)
    Count <- 0
    loop I from 0 to LENGTH(Scores) - 1
        if Scores[I] < 50 then
            Count <- Count + 1
```

```
        end if
    end loop
    return Count
end method

output CountFailing([45, 62, 78, 30, 91, 55])   // outputs 2
```

This new subroutine can be added entirely independently, without touching the existing `FilterPassing` or `CalculateAverage` subroutines at all — a direct benefit of modular design: new functionality can be added as new, self-contained modules rather than requiring changes to already-working, already-tested code.

Bài tập luyện tập

Explain one advantage of testing `FilterPassing` and `CalculateAverage` as separate, independent subroutines (unit testing) rather than only testing the combined program's final output.

Lời giải chi tiết

Testing each subroutine independently makes it far easier to **isolate and locate** the source of a bug: if the combined program's final output is wrong, testing only the final result cannot immediately reveal whether the error is in filtering, in averaging, or in how the two are combined. By testing `FilterPassing` on its own (e.g. confirming it correctly returns an empty list when all scores fail) and `CalculateAverage` on its own (e.g. confirming it correctly averages a simple known list), a fault in either subroutine can be identified and fixed directly, without needing to reason about the entire combined program at once.

Abstract Data Structures --- HL

HL ONLY — Higher Level extension topic (not assessed at SL)

Mục tiêu Unit: Ngăn xếp (stack) và hàng đợi (queue) tuyến tính/vòng; danh sách liên kết (linked list) và các thao tác chèn/xoá; cây nhị phân, cây nhị phân tìm kiếm (BST) và ba phép duyệt; đệ quy và ngăn xếp gọi hàm; so sánh cấu trúc dữ liệu tĩnh với động.

5.1 Static vs dynamic data structures

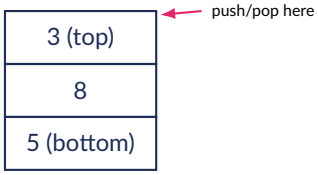
A **static** data structure (e.g. an array) has a size fixed at creation — memory is allocated once and cannot grow, though elements can be overwritten. A **dynamic** data structure (e.g. a linked list) can grow or shrink at run time, allocating new memory as needed. Arrays offer $O(1)$ direct index access but $O(n)$ insertion/deletion in the middle (later elements must shift); dynamic structures offer flexible size and $O(1)$ insertion/deletion *once you have a reference to the right place*, but no direct index access — you must traverse from a starting point.

VN GIẢI THÍCH TIẾNG VIỆT

Cấu trúc dữ liệu tĩnh (mảng): kích thước cố định khi tạo, truy cập chỉ số nhanh $O(1)$, nhưng chèn/xoá giữa mảng tốn $O(n)$ (phải dịch chuyển phần tử). **Cấu trúc dữ liệu động** (danh sách liên kết): có thể tăng/giảm kích thước lúc chạy, chèn/xoá nhanh nếu đã có vị trí, nhưng không truy cập trực tiếp theo chỉ số — phải duyệt từ đầu.

5.2 Stacks

A **stack** is a **LIFO** (Last-In-First-Out) abstract data type, restricted to two core operations: `push` (add an item to the top) and `pop` (remove and return the item at the top). Attempting to `pop` an empty stack is a **stack underflow** error; a stack with a fixed maximum capacity can suffer **stack overflow** if pushed beyond that limit. Stacks are used for undo history, browser back-button history, and managing function calls (the **call stack**).



A stack: both `push` and `pop` operate only at the top.

Ví dụ mẫu · Worked example

A stack starts empty. Trace `push(5)`, `push(8)`, `pop()`, `push(3)`, `pop()`.
push(5): [5] push(8): [5, 8] pop() removes 8 → [5] push(3): [5, 3] pop() removes 3 → [5].
Final stack: `[5]`. Last value popped: `3`.

VN **GIẢI THÍCH TIẾNG VIỆT**

Stack (ngăn xếp): vào sau ra trước (LIFO) — như chồng đĩa, chỉ thao tác ở đỉnh (`push`/`pop`). Dùng cho lịch sử undo, nút back trình duyệt, và quản lý lời gọi hàm (call stack). **Tràn ngăn xếp (overflow)**: push khi đã đầy; **cạn ngăn xếp (underflow)**: pop khi đang rỗng.

5.3 Queues

A **queue** is a **FIFO** (First-In-First-Out) abstract data type with `enqueue` (add to the rear) and `dequeue` (remove from the front). Queues model print spoolers, task/job scheduling buffers, and message queues. A **circular queue** reuses the space freed at the front by wrapping the rear pointer back to index 0 once it reaches the end of a fixed-size array, avoiding the need to shift elements or waste space that a naive linear queue would waste after several dequeues.

Ví dụ mẫu · Worked example

A queue starts empty. Trace `enqueue(A)`, `enqueue(B)`, `dequeue()`, `enqueue(C)`, `dequeue()`. enqueue(A): [A] → enqueue(B): [A, B] → dequeue() removes the front, A: [B] → enqueue(C): [B, C] → dequeue() removes the front, B: [C]. Remaining queue: `[C]`. Items removed, in order: `A, then B` (FIFO order).

VN **GIẢI THÍCH TIẾNG VIỆT**

Queue (hàng đợi): vào trước ra trước (FIFO) — như hàng người xếp hàng, thêm ở cuối (`enqueue`), lấy ra ở đầu (`dequeue`). **Hàng đợi vòng (circular queue)**: dùng lại vị trí trống ở đầu mảng bằng cách quay con trỏ cuối về chỉ số 0, tránh lãng phí bộ nhớ so với hàng đợi tuyến tính thông thường.

5.4 Linked lists

A **linked list** is a chain of **nodes**, each storing a data value plus a **pointer** to the next node; the last node's pointer is `NULL`. Unlike arrays, linked lists do not require contiguous memory and can grow/shrink dynamically one node at a time, but they do not support direct index access — traversal must always start from the **head**.

A singly linked list: each node holds data and a pointer to the next node; the final pointer is `NULL`.

Ví dụ mẫu · Worked example

Pseudocode to traverse and print every value in a linked list starting at `Head`:

```
Current <- Head
loop while Current <> NULL
    output Current.Data
    Current <- Current.Next
end loop
```

Ví dụ mẫu · Worked example

A singly linked list has `Head` pointing to node containing `5`, whose `Next` points to a node containing `9`, whose `Next` is `NULL`. Write pseudocode to insert a new node containing `7` *between* `5` and `9`.

```
NewNode.Data <- 7
NewNode.Next <- Head.Next
Head.Next <- NewNode
```

The new node must first be pointed at what `5` currently points to (`9`) *before* `5`'s pointer is overwritten to point at the new node — reversing this order would lose the reference to `9` and permanently break the chain.

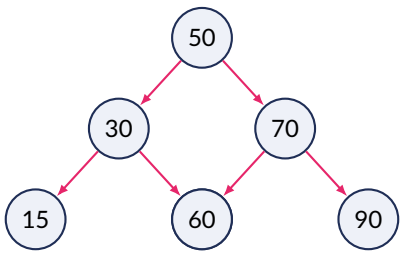
(!) Lỗi thường gặp: When inserting into a linked list, always capture the "next" reference *before* overwriting any pointer. The most common linked-list exam error is updating a pointer in the wrong order, silently orphaning the rest of the list (a memory leak / lost data with no error message).

VN **GIẢI THÍCH TIẾNG VIỆT**

Danh sách liên kết (linked list): mỗi **node** gồm dữ liệu và một con trỏ trỏ tới node kế tiếp; node cuối trỏ tới `NULL`. Ưu điểm so với mảng: chèn/xoá không cần dịch chuyển toàn bộ phần tử và không cần bộ nhớ liên tục; nhược điểm: không truy cập trực tiếp theo chỉ số, phải duyệt từ `Head`. Khi chèn node mới, LUÔN lưu con trỏ "tiếp theo" TRƯỚC khi ghi đè con trỏ khác, nếu không sẽ mất phần còn lại của danh sách.

5.5 Binary trees and traversal

A **binary tree** has nodes with at most two children (**left**, **right**). A **Binary Search Tree (BST)** additionally maintains the invariant that every value in a node's left subtree is less than the node, and every value in its right subtree is greater, enabling fast $O(\log n)$ search, insertion and deletion on a *balanced* tree (an unbalanced, "listlike" tree degrades to $O(n)$).



A binary search tree: every left child $<$ parent $<$ every right child.

Traversal orders (recursive): **Pre-order** = visit Root, then Left subtree, then Right subtree. **In-order** = Left, Root, Right (produces *sorted* order for any valid BST). **Post-order** = Left, Right, Root (useful for safely deleting a tree bottom-up, or evaluating an expression tree).

Ví dụ mẫu · Worked example

For the tree above, list the pre-order, in-order, and post-order traversals.

Pre-order (Root, L, R): $50, 30, 15, 40, 70, 60, 90$.
In-order (L, Root, R): $15, 30, 40, 50, 60, 70, 90$ — notice this is fully sorted, as expected for any BST.
Post-order (L, R, Root): $15, 40, 30, 60, 90, 70, 50$.

Ví dụ mẫu · Worked example

Trace inserting 35 into the BST above (root 50; left subtree 30 with children 15, 40; right subtree 70 with children 60, 90).

Compare 35 with root 50: $35 < 50$, go left to 30. Compare 35 with 30: $35 > 30$, go right to 40. Compare 35 with 40: $35 < 40$, go left — but 40 has no left child, so 35 is inserted there as the new left child of 40. The BST property is preserved: 35 is correctly greater than 30 but less than 40 and less than 50.

VN **GIẢI THÍCH TIẾNG VIỆT**

Cây nhị phân tìm kiếm (BST): con trái $<$ nút cha $<$ con phải. Ba phép duyệt: **Pre-order** (Gốc–Trái–Phải), **In-order** (Trái–Gốc–Phải, cho kết quả **đã sắp xếp** với BST), **Post-order** (Trái–Phải–Gốc). Khi chèn node mới vào BST: so sánh với gốc, đi trái nếu nhỏ hơn, đi phải nếu lớn hơn, lặp lại cho đến khi gặp vị trí trống. Đây là dạng bài rất hay gặp trong đề thi HL Paper 2.

5.6 Recursion

A **recursive** algorithm calls itself with a smaller sub-problem, and must have a **base case** that stops the recursion without a further call. Each call adds a new **stack frame** to the call stack (holding that call's parameters and local variables); without a base case, calls never stop and the stack grows until it **overflows**.

Ví dụ mẫu · Worked example

Trace `Factorial(4)` for the pseudocode:

```
method Factorial(N)
    if N = 0 then
        return 1
    else
        return N * Factorial(N - 1)
    end if
end method
```

`Factorial(4)` $= 4 \times$ Factorial(3) $= 4 \times (3 \times$ Factorial(2)$) = \cdots = 4 \times 3 \times 2 \times 1 \times 1 = 24$. Each call waits on the call stack until `Factorial(0)` returns 1 (the base case), then the multiplications unwind back up the stack.

Ví dụ mẫu · Worked example

Trace `Mystery(5)` where `method Mystery(N)`: if `N <= 1` then return 1, else return `N + Mystery(N - 1)`.

`Mystery(5)` $= 5+4+3+2+1 = 15$ — this recursively computes $\sum_{k=1}^{N} k = \dfrac{N(N+1)}{2}$, the sum of the first N positive integers. Check: $\dfrac{5\times 6}{2} = 15$. ✓

Ví dụ mẫu · Worked example

Write a recursive binary search function over `List[Low..High]` and trace `BinarySearch(List, 23, 0, 7)` on `[2, 7, 12, 18, 23, 31, 40, 55]`.

```
method BinarySearch(List, Target, Low, High)
    if Low > High then
        return -1
    end if
    Mid <- (Low + High) DIV 2
    if List[Mid] = Target then
        return Mid
    else if List[Mid] < Target then
        return BinarySearch(List, Target, Mid + 1, High)
    else
        return BinarySearch(List, Target, Low, Mid - 1)
    end if
end method
```

Call 1: `Low=0, High=7,` Mid$=3$, `List[3]=18 < 23,` recurse right: `BinarySearch(List,23,4,7).` Call 2: `Low=4, High=7,` Mid$=5$, `List[5]=31 > 23,` recurse left: `BinarySearch(List,23,4,4).` Call 3: `Low=4, High=4,` Mid$=4$, `List[4]=23=23,` base case reached: return 4. Final result: index **4**, matching the iterative trace from Unit 4.

(!) Lỗi thường gặp: Forgetting the **base case** (or writing a recursive call that never actually reaches it, e.g. always calling `Factorial(N)` instead of `Factorial(N-1)`) causes infinite recursion and a **stack overflow** — this is the single most common recursion bug tested on exams.

VN **GIẢI THÍCH TIẾNG VIỆT**

Đệ quy: hàm tự gọi lại chính nó với bài toán nhỏ hơn, PHẢI có **trường hợp cơ sở (base case)** để dừng. Mỗi lời gọi được đẩy vào **ngăn xếp gọi hàm (call stack)**; thiếu base case ⇒ tràn ngăn xếp (stack overflow). Tìm kiếm nhị phân đệ quy và duyệt cây đều là các ứng dụng đệ quy kinh điển trong đề HL.

VN **GIẢI THÍCH TIẾNG VIỆT**

Ôn tập nhanh Unit 5: (1) Cấu trúc tĩnh (mảng) vs động (linked list, cây); (2) Stack: LIFO, push/pop ở đỉnh — dùng cho undo, call stack; (3) Queue: FIFO, enqueue/dequeue — hàng đợi vòng tránh lãng phí bộ nhớ; (4) Linked list: node gồm dữ liệu + con trỏ, luôn lưu con trỏ tiếp theo trước khi ghi đè; (5) BST: trái $<$ gốc $<$ phải; in-order cho kết quả đã sắp xếp; (6) Đệ quy luôn cần base case, mỗi lời gọi → 1 stack frame.

5.7 Doubly linked and circular linked lists

A **doubly linked list** extends the singly linked list by giving each node *two* pointers: `Next` (to the following node) and `Previous` (to the preceding node). This allows traversal in *either* direction and makes deleting a known node much simpler (the node's own `Previous` pointer gives direct access to the node before it, without needing to traverse from the head to find it) — at the cost of extra memory per node (one additional pointer) and slightly more complex insertion/deletion logic (two pointers must be updated correctly on each side).

> A **circular linked list** has its final node point back to the **head** instead of to `NULL`, forming a loop. This is useful for applications that naturally cycle (e.g. a music player's "repeat playlist" feature, or round-robin scheduling in Unit 6), since after processing the last node, moving to "next" automatically returns to the first node without special-case code. A circular list can be singly or doubly linked.

Ví dụ mẫu · Worked example

A doubly linked list has nodes $A \leftrightarrow B \leftrightarrow C$ (with A.`Previous` = NULL and C.`Next` = NULL). Write pseudocode to delete node B given only a reference to B itself (not the head).

```
B.Previous.Next <- B.Next
B.Next.Previous <- B.Previous
```

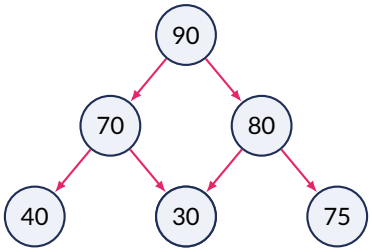
Because B already holds direct references to both its neighbours (via its own `Previous` and `Next` pointers), A and C can be linked directly to each other in just two steps, *without* needing to traverse the list from the head to locate B's neighbours first — this is precisely the advantage a doubly linked list has over a singly linked one for this operation.

VN **GIẢI THÍCH TIẾNG VIỆT**

Danh sách liên kết đôi (doubly linked list): mỗi node có 2 con trỏ (`Next` và `Previous`) — duyệt được cả hai chiều, xoá node dễ hơn (không cần duyệt lại từ đầu để tìm node trước đó), nhưng tốn thêm bộ nhớ. **Danh sách liên kết vòng (circular linked list)**: node cuối trỏ về lại `Head` thay vì `NULL` — phù hợp cho ứng dụng có tính tuần hoàn (phát nhạc lặp lại, lập lịch round-robin).

5.8 Heaps and priority queues

A **priority queue** is an abstract data type where each element has an associated **priority**, and `dequeue` always removes the element with the *highest* priority first, regardless of arrival order (unlike a plain FIFO queue). A **heap** is a common, efficient underlying implementation: a complete binary tree where every parent node's priority is (for a **max-heap**) greater than or equal to both its children's priorities.



A max-heap: every parent's value $\geq$ both its children's values (unlike a BST, left/right order between siblings does not matter).

Unlike a BST, a heap does *not* require left $<$ parent $<$ right — only that a parent is $\geq$ (max-heap) or $\leq$ (min-heap) *both* its children. This weaker condition makes heaps faster to maintain for their specific purpose: both `insert` and `remove-highest-priority` run in $O(\log n)$ time on a heap with n elements. Priority queues built on heaps are used in task schedulers (Unit 6), pathfinding algorithms, and event-driven simulations, wherever "process the most urgent/important item next" is required.

Ví dụ mẫu · Worked example

A hospital triage system uses a priority queue where patients are assigned a severity score (higher = more urgent). Patients arrive in this order with these severities: Patient A (3), Patient B (9), Patient C (5). In what order are they treated, and how does this differ from a plain FIFO queue?

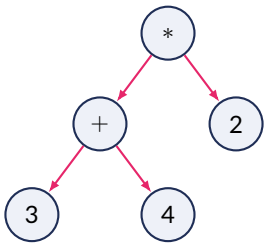
A **priority queue** always removes the highest-priority element first: despite arriving *last*, Patient B (severity 9) is treated **first**, then Patient C (severity 5), then Patient A (severity 3) — treatment order is B, C, A. A plain **FIFO queue** would instead treat them strictly in arrival order (A, B, C), which would be dangerously inappropriate for emergency triage, since it ignores how urgently each patient actually needs care.

VN **GIẢI THÍCH TIẾNG VIỆT**

Hàng đợi ưu tiên (priority queue): phần tử có **độ ưu tiên**; `dequeue` luôn lấy phần tử ƯU TIÊN CAO NHẤT trước, bất kể thứ tự đến. **Heap**: cây nhị phân hoàn chỉnh, node cha $\geq$ (max-heap) hoặc $\leq$ (min-heap) CẢ HAI con — không cần thứ tự trái/phải như BST. Cả thêm và lấy phần tử ưu tiên cao nhất đều chạy $O(\log n)$. Ứng dụng: lập lịch tác vụ, tìm đường đi, mô phỏng theo sự kiện.

5.9 Expression trees

An **expression tree** represents an arithmetic expression as a binary tree: each **leaf** node holds an operand (a number/variable), and each **internal** node holds an operator, with its two children being the sub-expressions it combines.



Expression tree for $(3 + 4) \times 2$: the root operator applies to its two child sub-expressions.

Traversing an expression tree gives different useful notations: **in-order** traversal (with brackets added around each operator's subtree) recovers the familiar **infix** expression, e.g. $(3 + 4) \times 2$; **post-order** traversal gives **postfix** (reverse Polish) notation, e.g. $3\ 4\ +\ 2\ *$, which is convenient for stack-based evaluation (as seen in Unit 5's stack calculator example); **pre-order** traversal gives **prefix** notation, e.g. $*\ +\ 3\ 4\ 2$.

Ví dụ mẫu · Worked example

For the expression tree above, evaluate the expression using a recursive tree-evaluation method, and state the result.

`Evaluate(root)` sees the root is an operator ($*$), so it recursively evaluates the left subtree and right subtree, then combines: `Evaluate(left)` $=$ `Evaluate`($+$ subtree) $= 3 + 4 = 7$ (itself found by recursively evaluating its own leaf children, 3 and 4). `Evaluate(right)` $= 2$ (a leaf, returned directly — the base case of the recursion). Combine: $7 \times 2 = 14$. The tree evaluates to **14**, matching $(3 + 4) \times 2 = 14$.

VN **GIẢI THÍCH TIẾNG VIỆT**

Cây biểu thức (expression tree): nút lá là toán hạng, nút trong là toán tử. Duyệt **in-order** (thêm ngoặc) cho ký hiệu trung tố (infix) quen thuộc; duyệt **post-order** cho ký hiệu hậu tố (postfix) — tiện lợi để tính bằng stack; duyệt **pre-order** cho ký hiệu tiền tố (prefix). Đánh giá cây biểu thức là một ứng dụng đệ quy kinh điển: nút lá là trường hợp cơ sở, nút toán tử kết hợp kết quả đệ quy của hai cây con.

5.10 Hash tables and dictionaries

A **dictionary** (also called a **map** or **associative array**) is an abstract data type storing **key–value pairs**, where each unique key maps to an associated value, and values are retrieved by key rather than by numeric index. A **hash table** is the standard efficient implementation: a **hash function** converts each key into an array index (a "bucket"), allowing near-instant lookup, insertion, and deletion on average.

A good hash function distributes keys roughly evenly across available buckets. A **collision** occurs when two different keys hash to the *same* bucket; common resolution strategies include **chaining** (each bucket holds a small linked list of all entries that hashed there) and **open addressing** (probe for the next free bucket according to a fixed rule). With a well-designed hash function and table size, both `insert` and `lookup` run in $O(1)$ *average* time — dramatically faster than a linear search's $O(n)$ for large datasets — though a poorly-designed hash function causing many collisions can degrade performance toward $O(n)$ in the worst case.

Ví dụ mẫu · Worked example

A simple hash function for string keys sums the ASCII codes of each character and takes the result modulo 5 (table size 5). Compute the bucket index for the key `"CAB"` (ASCII: C$=67$, A$=65$, B$=66$).

Sum $= 67 + 65 + 66 = 198$. Bucket index $= 198 \bmod 5 = 198 - (39 \times 5) = 198 - 195 = 3$. The key `"CAB"` is stored in **bucket 3** — any other key whose character-sum also happens to be $\equiv 3 \pmod 5$ (e.g. a sum of 193, 198, or 203) would collide with it in the same bucket, requiring chaining or another collision-resolution strategy to store both.

VN **GIẢI THÍCH TIẾNG VIỆT**

Từ điển (dictionary/map): kiểu dữ liệu trừu tượng lưu cặp **khoá–giá trị**, truy xuất theo khoá thay vì chỉ số. **Bảng băm (hash table)**: cách cài đặt hiệu quả, dùng **hàm băm** chuyển khoá thành chỉ số mảng — cho phép thêm/tìm gần như $O(1)$ trung bình. **Va chạm (collision)**: hai khoá khác nhau băm ra cùng một chỉ số — giải quyết bằng **chuỗi liên kết (chaining)** hoặc **dò địa chỉ mở (open addressing)**.

5.11 Balanced trees: the AVL intuition

A BST's $O(\log n)$ efficiency depends on the tree remaining reasonably **balanced**; inserting already-sorted data into a plain BST (Unit 5, earlier) can degrade it into a linked-list shape with $O(n)$ search. An **AVL tree** is a self-balancing BST that automatically restores balance after every insertion/deletion.

Each node in an AVL tree tracks a **balance factor** (the height of its left subtree minus the height of its right subtree). If an insertion causes any node's balance factor to become too extreme (outside $-1, 0, 1$), the tree performs a **rotation** — a local rearrangement of a small number of nodes that restores balance while preserving the BST ordering property. Because rotations keep the tree balanced after every change, AVL trees **guarantee** $O(\log n)$ search, insertion, and deletion even in the worst case — unlike a plain BST, which only achieves $O(\log n)$ if it happens to stay balanced.

Ví dụ mẫu · Worked example

Explain, conceptually (without full rotation mechanics), why inserting the sequence $10, 20, 30$ in that order into a plain (non-self-balancing) BST produces a badly unbalanced tree, while an AVL tree would not.

Inserting into a plain BST always attaches a new value as a leaf following the BST comparison rule: 10 becomes the root; $20 > 10$, so it becomes 10's right child; $30 > 20$, so it becomes 20's right child — producing a straight, purely right-leaning chain (effectively a linked list), giving $O(n)$ search despite being technically "a BST." An **AVL tree** would detect, after inserting 30, that the root (10)'s balance factor has become too extreme (all its height is on the right side) and would perform a **rotation** — restoring a balanced shape (e.g. 20 becomes the new root, with 10 and 30 as its two children) that preserves $O(\log n)$ search depth.

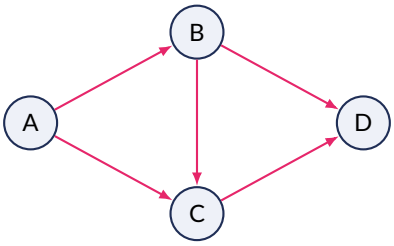
VN **GIẢI THÍCH TIẾNG VIỆT**

Cây AVL: cây nhị phân tìm kiếm TỰ CÂN BẰNG. Mỗi node theo dõi **hệ số cân bằng** (chiều cao cây con trái trừ chiều cao cây con phải); nếu chèn/xoá làm hệ số này vượt ngưỡng, cây thực hiện **phép xoay (rotation)** để khôi phục cân bằng mà vẫn giữ đúng tính chất BST. Nhờ vậy, cây AVL LUÔN đảm bảo $O(\log n)$ cho tìm kiếm/chèn/xoá, kể cả trường hợp xấu nhất — khác với BST thường chỉ đạt $O(\log n)$ nếu cây tình cờ cân bằng.

5.12 Graphs as an abstract data structure

A **graph** generalises a tree: it consists of **vertices** (nodes) connected by **edges**, but unlike a tree, a graph allows cycles and does not require a single root or a strict parent–child hierarchy. Graphs model many real-world networks: road maps, social networks, the web's hyperlink structure, and computer networks themselves (Unit 3).

An **undirected** graph's edges have no direction (e.g. a mutual friendship); a **directed** graph's edges point one way (e.g. a one-way street, or "follows" on social media). A **weighted** graph attaches a numeric cost/distance to each edge (e.g. road distance between cities). Two common representations: an **adjacency matrix** (an $n \times n$ grid where cell $[i][j]$ records whether/how an edge connects vertex i to vertex j — simple and fast to check any specific connection, but wastes memory for graphs with few edges) and an **adjacency list** (each vertex stores a list of only the vertices it directly connects to — more memory-efficient for sparse graphs, but slightly slower to check one specific pair).



A small directed graph with 4 vertices and 5 edges — notice, unlike a tree, vertex D is reachable via two different paths, and there is no single "root."

Ví dụ mẫu · Worked example

Represent the graph above as (a) an adjacency matrix and (b) an adjacency list.

(a) Adjacency matrix (row → column means an edge exists from row vertex to column vertex; 1 = edge, 0 = no edge):

	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

(b) Adjacency list: `A: [B, C], B: [C, D], C: [D], D: []`. For this graph (only 5 edges out of a possible 12 directed pairs among 4 vertices), the adjacency list uses noticeably less memory than the largely-empty 4×4 matrix — a general pattern for **sparse** graphs (relatively few edges compared with the maximum possible).

Graph traversal algorithms visit every reachable vertex from a starting point, generalising tree traversal (Unit 5, earlier) to structures that may contain cycles. **Breadth-First Search (BFS)** uses a **queue**: visit the start vertex, then all its direct neighbours, then all of *their* unvisited neighbours, and so on, layer by layer outward — naturally finds the *shortest path* (fewest edges) from the start vertex in an unweighted graph. **Depth-First Search (DFS)** uses a **stack** (or recursion): explore as far as possible down one path before backtracking to try alternatives — better suited to tasks like exhaustively exploring every possible path, or detecting cycles. Both must track **visited** vertices to avoid looping forever around a cycle.

Ví dụ mẫu · Worked example

Trace a BFS starting from vertex A on the graph above (edges: A→B, A→C, B→C, B→D, C→D). Initialise queue with A; mark A visited. Dequeue A, visit its neighbours B and C (both unvisited): enqueue both, mark both visited. Queue: [B, C]. Dequeue B, visit its neighbours C (already visited, skip) and D (unvisited): enqueue D, mark visited. Queue: [C, D]. Dequeue C, visit its neighbour D (already visited, skip). Queue: [D]. Dequeue D, no unvisited neighbours. Queue empty, BFS complete. **Visit order: A, B, C, D.** Notice this is exactly the order vertices were first discovered, layer by layer outward from A — the queue (FIFO, Unit 5) is what enforces this "widen outward evenly" behaviour.

VN **GIẢI THÍCH TIẾNG VIỆT**

Đồ thị (graph): tổng quát hoá cây — gồm **đỉnh (vertex)** và **cạnh (edge)**, CÓ THỂ có chu trình, không nhất thiết có gốc. Đồ thị **có hướng** vs **vô hướng**; đồ thị **có trọng số** gắn giá trị số cho

mỗi cạnh. Biểu diễn: **ma trận kề** (đơn giản nhưng tốn bộ nhớ nếu ít cạnh) hoặc **danh sách kề** (tiết kiệm bộ nhớ hơn với đồ thị thưa). **BFS** (dùng queue) duyệt theo từng lớp, tìm đường đi ngắn nhất (ít cạnh nhất); **DFS** (dùng stack/đệ quy) đi sâu hết một nhánh rồi mới quay lại — cả hai cần đánh dấu đỉnh ĐÃ THĂM để tránh lặp vô hạn khi có chu trình.

5.13 Comparing abstract data structures

Choosing the right ADT for a task is a key design skill: each structure trades off differently between access patterns, ordering, and efficiency.

ADT	Access pattern	Typical use
Stack	LIFO, only at top	Undo history, call stack, expression evaluation
Queue	FIFO, rear in / front out	Print spooling, task buffering, BFS
Linked list	Sequential, traversal from head	Dynamic-size ordered data, frequent insert/delete
BST	Ordered via comparisons	Fast search/insert/delete on ordered data ($O(\log n)$ if balanced)
Heap	Priority-based	Priority queues, always-need-the-max/min
Hash table	Direct via key	Fast key-based lookup, no ordering needed
Graph	Arbitrary connections	Networks, maps, relationships, dependencies

Ví dụ mẫu · Worked example

A ride-sharing app needs to find the shortest route (fewest road segments) between a driver's current location and a passenger's pickup point, given a map of intersections and roads. Which ADT and which traversal algorithm are most appropriate, and why?

The road map is naturally modelled as a **graph** (intersections as vertices, roads as edges). To find the shortest path in terms of *fewest road segments* (an unweighted shortest path), **Breadth-First Search (BFS)** is the appropriate traversal, since BFS explores outward layer by layer and is guaranteed to reach any vertex via the fewest possible edges before exploring vertices further away — exactly the "shortest hop-count path" property required here. (If the requirement were shortest *distance*, using a weighted graph, a more advanced weighted shortest-path algorithm building on these same graph foundations would be used instead.)

VN **GIẢI THÍCH TIẾNG VIỆT**

Lựa chọn đúng **ADT** là kỹ năng thiết kế quan trọng: **stack** cho undo/gọi hàm, **queue** cho hàng đợi công việc, **linked list** cho dữ liệu có thứ tự cần chèn/xoá thường xuyên, **BST** cho tìm kiếm nhanh trên dữ liệu có thứ tự, **heap** cho hàng đợi ưu tiên, **hash table** cho tra cứu nhanh theo khoá, **graph** cho mạng lưới/quan hệ/bản đồ.

5.14 Array-based vs pointer-based (dynamic) implementations

Every ADT discussed in this chapter can be implemented in more than one way; a key exam skill is recognising the trade-offs between a fixed-size array-based implementation and a dynamically-linked (pointer-based) implementation.

An **array-based** implementation (e.g. a stack stored in a fixed-size array with a "top" index variable) is simple, and benefits from **spatial locality** (Unit 2) since elements sit contiguously in memory — but its maximum size must be decided in advance, and a full array causes a **stack overflow** (or requires a slow, manual resize: allocate a new, larger array and copy every existing element across). A **pointer-based (linked)** implementation (e.g. a stack built from linked-list nodes, Unit 5 earlier) can grow dynamically, limited only by total available memory rather than a pre-declared size — but uses slightly more memory per element (storing a pointer alongside each value) and loses the cache-friendly spatial locality of a contiguous array.

Ví dụ mẫu · Worked example

A fixed-size array-based stack of capacity 5 is full: `[12, 7, 9, 3, 20]` with `Top = 4` (0-indexed). Explain what happens if `push(15)` is attempted, and what a program should do to handle this correctly.

Attempting `push(15)` on an already-full fixed-size array would be a **stack overflow** condition — there is no array slot available to store the new value. A correctly-written program must explicitly **check** whether the stack is full (e.g. `if Top = MaxSize - 1`) *before* attempting to push, and if so, either reject the operation with an appropriate error/message, or (in a more advanced implementation) allocate a new, larger array, copy all 5 existing elements across, and only then insert the new value — silently attempting to write past the end of a fixed array without such a check would corrupt adjacent memory or crash the program.

VN **GIẢI THÍCH TIẾNG VIỆT**

Cài đặt bằng mảng: đơn giản, tận dụng tính cục bộ không gian (nhanh hơn về bộ nhớ đệm) nhưng kích thước tối đa phải xác định trước — đầy mảng gây tràn ngăn xếp (stack overflow) hoặc phải cấp phát lại mảng lớn hơn và sao chép (chậm). **Cài đặt bằng con trỏ (linked)**: phát triển động, chỉ giới hạn bởi bộ nhớ khả dụng, nhưng tốn thêm bộ nhớ cho con trỏ và mất tính cục bộ không gian.

5.15 The Set abstract data type

A **set** is an unordered collection of *unique* values — no duplicates are permitted, and (unlike an array or list) there is no meaningful concept of "position" or index.

Core set operations: `add` (insert a value, with no effect if it is already present); `remove`; `contains` (test membership); and set-theory operations **union** (all elements in either set), **intersection** (only elements in *both* sets), and **difference** (elements in one set but not the other). Sets are frequently implemented internally using a hash table (this chapter, earlier) for near-$O(1)$ average membership testing, since a set's core requirement — "is this value already present?" — is exactly what a hash table answers efficiently.

Ví dụ mẫu · Worked example

Given Set A $= \{2, 4, 6, 8\}$ and Set B $= \{4, 8, 10\}$, compute $A \cup B$ (union), $A \cap B$ (intersection), and $A - B$ (difference).

Union $A \cup B = \{2, 4, 6, 8, 10\}$ (every element appearing in either set, with duplicates like 4 and 8 counted only once each, since a set never holds duplicates). **Intersection** $A \cap B = \{4, 8\}$ (only elements present in *both* sets). **Difference** $A - B = \{2, 6\}$ (elements in A that are *not* also in B).

VN **GIẢI THÍCH TIẾNG VIỆT**

Tập hợp (set): bộ sưu tập KHÔNG có thứ tự, không có phần tử trùng lặp. Các phép toán: `add/remove/contains`, và các phép toán tập hợp: **hợp (union)**, **giao (intersection)**, **hiệu (difference)**. Set thường được cài đặt bằng bảng băm để kiểm tra thành viên gần như $O(1)$.

5.16 Practice

Bài tập luyện tập

A queue starts empty. Trace `enqueue(A)`, `enqueue(B)`, `dequeue()`, `enqueue(C)`, `dequeue()`. What remains in the queue, and in what order were items removed?

Lời giải chi tiết

enqueue(A): [A] → enqueue(B): [A, B] → dequeue() removes the front, A: [B] → enqueue(C): [B, C] → dequeue() removes the front, B: [C]. Remaining queue: `[C]`. Items removed, in order: `A, then B` (FIFO order).

Bài tập luyện tập

For the BST in this chapter (root 50; left subtree 30/15/40; right subtree 70/60/90), which traversal order will always output the values in ascending sorted order?

(A) Pre-order
(B) In-order
(C) Post-order
(D) None of them

Lời giải chi tiết

In-order traversal (Left, Root, Right) visits every node smaller than the root before the root, and every node larger than the root after it — recursively, this yields values in strictly ascending order for any valid BST. **(B)**.

Bài tập luyện tập

A recursive method is defined as: `method Mystery(N)`: if `N <= 1` then return 1, else return `N + Mystery(N - 1)`. What does `Mystery(5)` return, and what well-known formula does this compute?

Lời giải chi tiết

`Mystery(5)` $= 5 + 4 + 3 + 2 + 1 = 15$ — this recursively computes $\sum_{k=1}^{N} k = \dfrac{N(N+1)}{2}$, the sum of the first N positive integers. Check: $\dfrac{5 \times 6}{2} = 15$. ✓

Bài tập luyện tập

A singly linked list has `Head` pointing to node containing `5`, whose `Next` points to a node containing `9`, whose `Next` is `NULL`. Write pseudocode to insert a new node containing `7` *between* `5` and `9`.

Lời giải chi tiết

```
NewNode.Data <- 7
NewNode.Next <- Head.Next
Head.Next <- NewNode
```

The new node must first be pointed at what `5` currently points to (`9`) *before* `5`'s pointer is overwritten to point at the new node — reversing this order would lose the reference to `9` and break the chain.

Bài tập luyện tập

Which abstract data structure is most appropriate for implementing an ”undo” feature in a text editor, and why?

(A) Queue (FIFO)
(B) Stack (LIFO)
(C) Binary search tree
(D) Linked list traversed only forward

Lời giải chi tiết

Undo must reverse the *most recent* action first, which is exactly LIFO behaviour: each edit is `push`ed onto a stack, and ”undo” calls `pop()` to retrieve and reverse the last action. **(B)**.

Bài tập luyện tập

Trace `push(1)`, `push(2)`, `push(3)`, `pop()`, `pop()`, `push(9)` on an initially empty stack. State the final stack contents (top to bottom).

Lời giải chi tiết

push(1): [1]. push(2): [1,2]. push(3): [1,2,3]. pop() removes 3: [1,2]. pop() removes 2: [1]. push(9): [1,9]. Final stack, top to bottom: `9`, `1` (9 is on top, having been pushed last).

Bài tập luyện tập

Explain why a circular queue is more memory-efficient than a simple linear queue implemented in a fixed-size array, after several enqueue/dequeue operations.

Lời giải chi tiết

In a simple **linear queue**, once items are dequeued from the front, those array slots become unused, but the rear pointer keeps moving forward — eventually the rear pointer reaches the end of the array and no more items can be enqueued, even though slots at the front are free. A **circular queue** instead wraps the rear pointer back to index 0 once it passes the last array index, reusing the freed front slots — allowing the same fixed-size array to be used indefinitely without wasting the space vacated by earlier dequeues.

Bài tập luyện tập

Insert the value 55 into the BST with root 50 (left subtree 30 with children 15, 40; right subtree 70 with children 60, 90). Show the comparisons made and the final position of the new node.

Lời giải chi tiết

Compare 55 with root 50: $55 > 50$, go right to 70. Compare 55 with 70: $55 < 70$, go left to 60. Compare 55 with 60: $55 < 60$, go left — 60 has no left child, so 55 is inserted there as the new left child of 60. Resulting in-order traversal remains sorted: $15, 30, 40, 50, 55, 60, 70, 90$.

Bài tập luyện tập

Write recursive pseudocode for an in-order traversal of a binary tree given a node reference, assuming each node has `Left`, `Data`, and `Right` fields.

Lời giải chi tiết

```
method InOrder(Node)
    if Node <> NULL then
        InOrder(Node.Left)
        output Node.Data
        InOrder(Node.Right)
    end if
end method
```

The base case is reaching a `NULL` child (an empty subtree), at which point the method simply returns without any output — this naturally stops the recursion at every leaf's children.

Bài tập luyện tập

Explain why searching an unbalanced BST that has degenerated into a single chain (every node has only a right child, no left children at all) is no longer $O(\log n)$, and state its actual complexity.

Lời giải chi tiết

The $O(\log n)$ efficiency of BST search relies on each comparison eliminating roughly *half* of the remaining nodes, which only happens when the tree is reasonably **balanced**. If every node has only a right child, the tree is effectively a linked list in disguise — each comparison eliminates only the single current node, not half the remaining data. Searching such a degenerate tree therefore requires checking up to every node in the worst case, giving complexity $O(n)$, identical to a simple linear search.

Bài tập luyện tập

Explain what would happen if the base case were removed entirely from the `Factorial` function in this chapter, and name the resulting runtime error.

Lời giải chi tiết

Without the base case (`if N = 0 then return 1`), every call to `Factorial(N)` would always call `Factorial(N-1)` with no condition to ever stop — the recursion would continue indef-

initely (even going into negative numbers), each call adding another frame to the call stack. Since the call stack has a finite maximum size, this uncontrolled growth would eventually exhaust it, causing a **stack overflow** error and crashing the program.

Bài tập luyện tập

A print spooler processes documents in the order they were sent to the printer. Which ADT models this behaviour, and name its two core operations.

Lời giải chi tiết

A **queue (FIFO)** models this behaviour, since the first document sent should be the first one printed. Its two core operations are `enqueue` (adding a new print job to the rear of the queue as it arrives) and `dequeue` (removing and printing the job currently at the front of the queue).

Bài tập luyện tập

Given post-order traversal $15, 40, 30, 60, 90, 70, 50$ of a BST, identify the root of the tree and explain your reasoning.

Lời giải chi tiết

In a **post-order** traversal (Left, Right, Root), the root of the *entire* tree is always visited **last** — so the final value in the sequence, **50**, must be the root. (This matches the BST used throughout this chapter: root 50, left subtree 30/15/40, right subtree 70/60/90.)

Bài tập luyện tập

Compare the worst-case time complexity of searching a value in: (a) an unsorted array using linear search; (b) a balanced BST; (c) a degenerate (linked-list-shaped) BST. State each in Big-O notation.

Lời giải chi tiết

(a) Unsorted array, linear search: $O(n)$ — every element may need checking. (b) Balanced BST: $O(\log n)$ — each comparison eliminates roughly half the remaining nodes. (c) Degenerate BST (effectively a linked list): $O(n)$ — no nodes are eliminated faster than one at a time, identical to linear search despite superficially being "a tree."

Bài tập luyện tập

A stack-based calculator evaluates the postfix expression `3 4 + 2 *` by pushing numbers and, on seeing an operator, popping two values, applying the operator, and pushing the result. Trace this evaluation.

Lời giải chi tiết

Push 3: [3]. Push 4: [3, 4]. See +: pop 4 and 3, compute $3 + 4 = 7$, push 7: [7]. Push 2: [7, 2]. See $*$: pop 2 and 7, compute $7 \times 2 = 14$, push 14: [14]. Final stack: [14] — the expression evaluates to **14**, matching the equivalent infix $(3 + 4) \times 2 = 14$.

Bài tập luyện tập

Explain why a stack (rather than a queue) naturally models the sequence of nested function calls in a program.

Lời giải chi tiết

When a function calls another function, the calling function must **pause and wait** for the called function to finish before it can resume — and this must happen in strict reverse order of calling (the most recently called function must finish *first* before control can return to whichever function called it). This "last call must finish first" behaviour is exactly **LIFO**, which a stack directly implements: each new call `pushes` a frame; each return `pops` it, always affecting the most recently added (innermost) call first. A queue's FIFO behaviour would instead resume calls in the order they were *originally* made, which does not match how nested function calls actually unwind.

Bài tập luyện tập

A doubly linked list has nodes $X \leftrightarrow Y \leftrightarrow Z$. Given only a reference to node Y, write pseudocode to delete it, and explain why this would require traversing from the head in a *singly* linked list instead.

Lời giải chi tiết

```
Y.Previous.Next <- Y.Next
Y.Next.Previous <- Y.Previous
```

This works directly because Y already stores references to both its neighbours (X via `Previous`, Z via `Next`). In a **singly** linked list, each node only stores a reference to the *next* node, not the previous one — so given only Y, there would be no way to find X (the node pointing *to* Y) directly; the list would need to be traversed from the `Head` until the node whose `Next` equals Y is found, an $O(n)$ operation instead of the doubly linked list's $O(1)$ deletion once Y is known.

Bài tập luyện tập

Explain one practical use of a circular linked list, and state what makes it "circular" rather than a standard singly linked list.

Lời giải chi tiết

One practical use: a music player's "repeat playlist" feature, where after the last song finishes, playback should automatically continue from the first song again without special-case code to detect "end of list, go back to start." What makes it **circular** is that the *last* node's `Next` pointer references the **head** node instead of `NULL` — so traversal can continue indefinitely in a loop, rather than reaching a dead end that the program must explicitly detect and handle.

Bài tập luyện tập

Insert the value `85` into the max-heap shown in this chapter (root 90; left subtree 70 with children 40, 65; right subtree 80 with children 30, 75), following the rule that a new value is added as a new leaf and then swapped upward with its parent while it is greater than that

parent. Describe the final position of 85.

Lời giải chi tiết

The new value 85 is first added as a new leaf (e.g. as a child of 80, the next available position in the complete tree). Since $85 > 80$ (its parent), they are swapped: 85 moves up to where 80 was (as a child of the root, 90), and 80 becomes 85's child. Comparing again: is $85 > 90$ (the root)? No, so the swapping ("bubbling up") stops here. Final position: **85 becomes a child of the root (90)**, with 80 now demoted to be 85's child — the max-heap property (parent $\geq$ both children) is preserved throughout.

Bài tập luyện tập

Explain why a priority queue implemented with a heap is more efficient than one implemented by simply keeping an unsorted list and linearly scanning for the highest-priority element every time `dequeue` is called.

Lời giải chi tiết

Scanning an unsorted list for the highest-priority element requires checking every element, an $O(n)$ operation for *every single* `dequeue` call. A **heap** maintains a partial ordering (parent $\geq$ children) that lets both `insert` and `remove-highest-priority` be performed in only $O(\log n)$ time, by following a single path from root to leaf (or leaf to root) rather than examining every element. For an application performing many dequeue operations on a large priority queue, this difference — $O(\log n)$ versus $O(n)$ per operation — becomes very significant as the number of elements grows.

Bài tập luyện tập

Draw (describe in words) the expression tree for $(5-2)\times(1+4)$ and give its post-order (postfix) traversal.

Lời giải chi tiết

The root is $\times$, with a left subtree representing $5-2$ (root $-$, children 5 and 2) and a right subtree representing $1+4$ (root $+$, children 1 and 4). **Post-order** traversal (Left, Right, Root) visits: left subtree first in post-order $(5, 2, -)$, then right subtree in post-order $(1, 4, +)$, then finally the root $(\times)$. Full postfix result: `5 2 - 1 4 + *`.

Bài tập luyện tập

Using the stack-based postfix evaluation method from Unit 5's earlier example, evaluate the postfix expression `5 2 - 1 4 + *` obtained in the previous question.

Lời giải chi tiết

Push 5: [5]. Push 2: [5,2]. See −: pop 2 and 5, compute $5 - 2 = 3$, push 3: [3]. Push 1: [3,1]. Push 4: [3,1,4]. See +: pop 4 and 1, compute $1 + 4 = 5$, push 5: [3,5]. See $*$: pop 5 and 3, compute $3 \times 5 = 15$, push 15: [15]. Final result: **15**, matching the infix calculation $(5 - 2) \times (1 + 4) = 3 \times 5 = 15$. ✓

Bài tập luyện tập

Explain why post-order (not pre-order or in-order) traversal is the natural choice for evaluating an expression tree using a stack-based calculator.

Lời giải chi tiết

A stack-based postfix evaluator needs to see *both operands of an operation before it sees the operator itself*, so it can pop them off the stack and combine them when the operator is finally read. **Post-order** traversal (Left, Right, Root) visits an operator's two operand subtrees completely *before* visiting the operator node itself, guaranteeing every operand needed for a given operation has already been pushed onto the stack by the time that operator is encountered — exactly matching how postfix (reverse Polish) notation is meant to be evaluated. Pre-order would output the operator *before* its operands are known, and in-order does not group operators and operands in an order a simple left-to-right stack scan could directly evaluate.

Bài tập luyện tập

Using the hash function from this chapter (sum of ASCII codes, modulo 5), compute the bucket index for the key `"BAD"` (ASCII: B$=66$, A$=65$, D$=68$), and state whether it collides with `"CAB"` from the worked example (bucket 3).

Lời giải chi tiết

Sum $= 66 + 65 + 68 = 199$. Bucket index $= 199 \bmod 5$: $199 - (39 \times 5) = 199 - 195 = 4$. `"BAD"` hashes to **bucket 4**, which is different from `"CAB"`'s bucket 3 — so no collision occurs between these two specific keys.

Bài tập luyện tập

Explain why a hash table generally offers faster average-case lookup than a BST, and one advantage a BST retains that a hash table does not.

Lời giải chi tiết

A well-designed **hash table** computes a key's bucket directly via its hash function, giving $O(1)$ *average*-case lookup — no comparisons against other stored keys are needed at all, unlike a BST's $O(\log n)$, which requires traversing down the tree comparing at each node. However, a **BST** retains the ability to easily retrieve data in **sorted order** (via in-order traversal) and to efficiently answer range queries (e.g. "all keys between 10 and 50") — a hash table has no inherent ordering between keys, since hashing deliberately scatters keys across buckets with no relationship to their relative order.

Bài tập luyện tập

Explain what a "collision" is in a hash table, and describe how chaining resolves it.

Lời giải chi tiết

A **collision** occurs when two different keys are hashed by the hash function to the exact same bucket index. **Chaining** resolves this by allowing each bucket to hold not just one entry, but a small linked list of *all* entries that have hashed to that bucket — when a collision occurs, the

new key-value pair is simply appended to that bucket's linked list, rather than overwriting the existing entry. Looking up a key that collided with others still works correctly, but requires a short linear search through that specific bucket's small linked list to find the matching key, rather than one direct array access.

Bài tập luyện tập

Explain why a plain (non-self-balancing) BST built by inserting values in strictly *ascending* order always degrades to $O(n)$ search, using the BST insertion rule.

Lời giải chi tiết

The standard BST insertion rule always compares a new value with existing nodes and follows the "greater → go right" branch when the new value exceeds the current node. If values are inserted in strictly *ascending* order, every single new value is greater than *every* value already in the tree, so each new node is always attached as the right child of the previously-inserted node — producing a tree that is really just a straight chain leaning entirely to the right (structurally identical to a linked list). Searching such a tree requires potentially visiting every single node, exactly like a linear search, giving $O(n)$ rather than the $O(\log n)$ a balanced tree would provide.

Bài tập luyện tập

Explain the practical benefit of an AVL tree's guaranteed $O(\log n)$ worst-case performance for an application that repeatedly inserts already-sorted or near-sorted data (e.g. timestamped log entries arriving in time order).

Lời giải chi tiết

As shown in this chapter, inserting already-sorted data into a *plain* BST produces a degenerate, linked-list-shaped tree with $O(n)$ search — exactly the kind of data pattern (timestamped log entries, naturally arriving in increasing time order) that would trigger this worst case in practice, not just in theory. An **AVL tree** automatically detects the resulting imbalance after each insertion and performs rotations to restore balance, guaranteeing $O(\log n)$ search performance is *maintained* even under this realistic, commonly-occurring "already sorted" insertion pattern — making it a much safer choice than a plain BST for applications where sorted or near-sorted insertion order is likely.

Bài tập luyện tập

For the graph with edges A→B, A→C, B→C, B→D, C→D (as in this chapter), trace a DFS starting from vertex A, always choosing to visit the *first* unvisited neighbour in alphabetical order, backtracking when stuck.

Lời giải chi tiết

Visit A (mark visited). From A, go to first unvisited neighbour, B (mark visited). From B, go to first unvisited neighbour, C (mark visited). From C, go to its neighbour D (unvisited, mark visited). From D, no unvisited neighbours — backtrack to C (no more unvisited neighbours) — backtrack to B (neighbour D already visited) — backtrack to A (neighbours B, C already visited). DFS complete. **Visit order: A, B, C, D** — in this particular small graph DFS happens to produce the same order as BFS, but in general DFS plunges as deep as possible down one path first,

while BFS spreads outward evenly layer by layer.

Bài tập luyện tập

Explain why an adjacency list is more memory-efficient than an adjacency matrix for a social network with 1 million users, where the average user is only directly connected to about 200 others.

Lời giải chi tiết

An **adjacency matrix** for 1 million vertices would require a $1{,}000{,}000 \times 1{,}000{,}000$ grid — one trillion cells — regardless of how many actual connections exist, the vast majority of which would be 0 (no connection), since each user connects to only about 200 out of a million possible others. An **adjacency list** instead only stores, for each user, the (roughly 200) users they are actually connected to — using memory proportional to the *actual number of edges* (roughly 200 million total, across all users) rather than to the square of the number of vertices, making it vastly more memory-efficient for this kind of **sparse** graph (a graph with far fewer edges than the maximum possible).

Bài tập luyện tập

Explain why BFS, rather than DFS, is the correct choice when the goal is specifically to find the shortest path (fewest edges) between two vertices in an unweighted graph.

Lời giải chi tiết

BFS explores the graph strictly layer by layer outward from the start vertex — visiting every vertex exactly one edge away before any vertex two edges away, then every vertex two edges away before any vertex three edges away, and so on. This guarantees that the *first time* BFS reaches any given target vertex, it has done so via the fewest possible number of edges. **DFS** instead plunges as deep as possible down one arbitrary path before backtracking, and may easily reach the target vertex via a long, roundabout path long before it would have found a much shorter path through a different branch — DFS provides no such shortest-path guarantee.

Bài tập luyện tập

A company needs a data structure to store employee records so that a specific employee can be retrieved almost instantly by their unique employee ID number, with no need to ever list employees in ID order. Which ADT from this chapter's summary table is most appropriate, and why not a BST?

Lời giải chi tiết

A **hash table** is most appropriate: since lookups are always by the unique employee ID (the key) and there is no stated need to retrieve employees in sorted order, a hash table's $O(1)$ average-case lookup by key is both sufficient and faster than a BST's $O(\log n)$ lookup. A **BST** would be the better choice only if the application also needed to efficiently retrieve employees in *sorted ID order* or answer range queries (e.g. "all employees with ID between 1000 and 2000") — capabilities a hash table does not provide, but which are not required in this scenario.

Bài tập luyện tập

Explain why a pointer-based (linked) implementation of a queue is generally preferred over a fixed-size array-based implementation when the maximum number of items the queue will ever hold is genuinely unpredictable in advance.

Lời giải chi tiết

A fixed-size array-based queue requires committing to a maximum capacity when the program is written; if actual demand exceeds this limit, the queue overflows and further `enqueue` operations must be rejected (or require an expensive resize-and-copy operation). A **pointer-based (linked)** implementation instead allocates a new node for each new element as it is enqueued, allowing the queue to grow dynamically to whatever size is actually needed at runtime, limited only by total available memory rather than a size decided in advance — a clear advantage whenever the maximum realistic size genuinely cannot be predicted ahead of time.

Bài tập luyện tập

Given Set X $= \{A, C, E, G\}$ and Set Y $= \{C, E, H\}$, compute $X \cup Y$, $X \cap Y$, and $Y - X$.

Lời giải chi tiết

Union $X \cup Y = \{A, C, E, G, H\}$. **Intersection** $X \cap Y = \{C, E\}$ (elements in both). **Difference** $Y - X = \{H\}$ (elements in Y but not in X — note C and E are excluded since they also appear in X).

Bài tập luyện tập

Explain why adding the value `"apple"` twice to a set results in the set containing `"apple"` only once, and contrast this with adding the same value twice to a queue.

Lời giải chi tiết

A **set**, by definition, only ever stores *unique* values — its `add` operation checks whether the value is already present and has no effect if it is, so adding `"apple"` a second time leaves the set unchanged, still containing exactly one `"apple"`. A **queue** has no such uniqueness restriction: `enqueue("apple")` performed twice simply adds two *separate* entries of `"apple"` to the queue, both of which will eventually be dequeued in turn — a queue is concerned with *order* of processing, not with whether values are unique.

Bài tập luyện tập

A binary tree has root 40, with left child 20 (whose children are 10 and 30) and right child 60 (whose children are 50 and 70). Is this a valid BST? Justify your answer using the BST property.

Lời giải chi tiết

Yes, this is a valid **BST**. Checking the property (every left subtree $<$ parent $<$ every right subtree) at each node: root 40 — left subtree $\{10, 20, 30\}$ are all < 40, right subtree $\{50, 60, 70\}$ are all > 40. Node 20 — left child 10 $< 20 <$ right child 30. Node 60 — left child 50 $< 60 <$ right child 70. Every node satisfies the BST ordering property, so this is indeed a valid BST (and, being a fully-filled shape at this size, an already *balanced* one).

Bài tập luyện tập

Explain the base case and recursive case of a recursive method `SumList(List, Index)` intended to sum all elements of `List` from `Index` to the end, and write its pseudocode.

Lời giải chi tiết

```
method SumList(List, Index)
    if Index >= LENGTH(List) then
        return 0
    else
        return List[Index] + SumList(List, Index + 1)
    end if
end method
```

The **base case** (`Index >= LENGTH(List)`) handles running off the end of the list, returning 0 since there is nothing left to add. The **recursive case** adds the current element to the sum of everything from the *next* index onward, computed by a smaller recursive call — each call moves `Index` strictly closer to the base case, guaranteeing the recursion eventually terminates.

Bài tập luyện tập

A stack-based undo system has performed 4 actions in order: Type, Bold, Underline, Delete. The user presses undo twice. Trace which actions remain "active" and in what order the undone actions were reversed.

Lời giải chi tiết

Each action is `pushed` as it occurs: stack (bottom to top) is `[Type, Bold, Underline, Delete]`. Pressing undo calls `pop()`, removing and reversing the top item first: first undo pops and reverses **Delete**; second undo pops and reverses **Underline**. Actions reversed, in order: **Delete, then Underline** (LIFO order — the most recent actions are undone first). Remaining stack (still "active," not undone): `[Type, Bold]`.

Bài tập luyện tập

Explain why a linked list's traversal to reach the k-th element is $O(n)$ in the worst case, while an array's access to its k-th element is $O(1)$, and state the practical implication for an algorithm requiring frequent random access by index.

Lời giải chi tiết

An **array** stores elements at contiguous, directly-computable memory addresses, so accessing index k requires a single calculation (base address plus k times element size) regardless of k's value — $O(1)$. A **linked list** has no such direct addressing; reaching the k-th node requires starting at the `Head` and following exactly k `Next` pointers one at a time, so in the worst case (a large k) this takes time proportional to n — $O(n)$. An algorithm that frequently needs random access by index (e.g. repeatedly jumping to arbitrary positions) would perform far better backed by an **array** than a linked list, despite the linked list's advantages for frequent insertion/deletion.

Resource Management --- HL

HL ONLY — Higher Level extension topic (not assessed at SL)

> **Mục tiêu Unit:** Vai trò của hệ điều hành; quản lý tiến trình & các thuật toán lập lịch CPU (FCFS, SJF, Round Robin, Priority); quản lý bộ nhớ (phân trang, phân đoạn, bộ nhớ ảo, thay thế trang); trình biên dịch/thông dịch/hợp ngữ và các giai đoạn biên dịch; deadlock & tranh chấp tài nguyên.

6.1 The role of the operating system

The **operating system (OS)** manages hardware resources (CPU time, memory, storage, I/O devices) and provides a stable, consistent platform on which application software runs, without every application needing to control hardware directly.

> Core OS responsibilities: **process management** (creating, scheduling, and terminating running programs); **memory management** (allocating and reclaiming RAM safely between processes); **file management** (organising data into files/directories on storage); **device management** (mediating access to I/O devices via drivers); **security** (user accounts, permissions, authentication); providing a **user interface** (command-line or graphical).

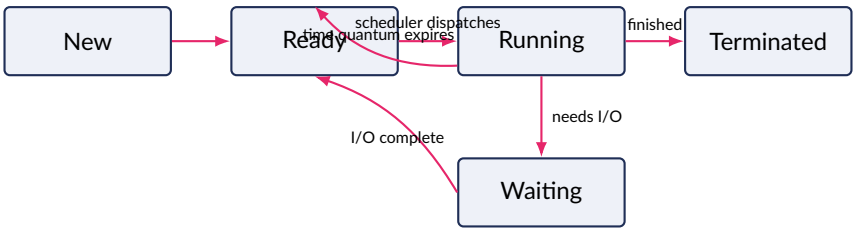
> VN **GIẢI THÍCH TIẾNG VIỆT**
> **Hệ điều hành** quản lý tài nguyên phần cứng: CPU, bộ nhớ, thiết bị vào/ra, tệp tin, và cung cấp một nền tảng ổn định cho các ứng dụng chạy mà không cần tự điều khiển phần cứng trực tiếp.

Multitasking lets one CPU (or core) appear to run several programs at once by rapidly switching between them (**time-slicing**), fast enough that it appears simultaneous to a human user. **Multiprocessing** uses multiple physical CPUs/cores to genuinely execute multiple tasks in parallel at the same instant.

> VN **GIẢI THÍCH TIẾNG VIỆT**
> **Đa nhiệm (multitasking)**: 1 CPU luân phiên xử lý nhiều chương trình rất nhanh (chia lát thời gian — time-slicing), tạo *cảm giác* chạy song song. **Đa xử lý (multiprocessing)**: nhiều lõi/CPU thực sự xử lý song song tại cùng một thời điểm.

6.2 Process states and scheduling

A process moves through a well-defined set of states during its lifetime.



Process state transitions: New → Ready → Running, with possible detours to Waiting (blocked on I/O) before returning to Ready, and finally Terminated.

Ready means a process has everything it needs to run except CPU time. **Running** means it currently has the CPU. **Waiting** (blocked) means it cannot proceed until an external event completes (e.g. disk I/O finishes), and is *not* simply waiting for the CPU. A **scheduler** decides which *ready* process the CPU runs next.

VN **GIẢI THÍCH TIẾNG VIỆT**

Trạng thái tiến trình: New (mới) → Ready (sẵn sàng) → Running (đang chạy) → (Waiting nếu cần I/O, rồi quay lại Ready) → Terminated (kết thúc). **Ready**: đã sẵn sàng, chỉ còn chờ CPU. **Waiting**: đang bị chặn bởi sự kiện bên ngoài (ví dụ chờ đọc đĩa), KHÔNG chỉ đơn thuần chờ CPU.

Scheduling algorithms.

- **FCFS** (First-Come-First-Served): runs processes strictly in arrival order — simple, but can cause the *convoy effect* (many short jobs stuck waiting behind one long job).
- **SJF** (Shortest-Job-First): runs the process with the shortest burst time next — minimises average waiting time among non-preemptive algorithms, but can **starve** long jobs indefinitely if short jobs keep arriving.
- **Round Robin**: each ready process gets a fixed **time quantum** in rotation before moving to the back of the queue — fair and responsive for interactive systems, but incurs **context-switch** overhead (time spent saving/restoring process state at each switch).
- **Priority scheduling**: the highest-priority ready process runs first — can starve low-priority jobs unless **aging** is used (gradually increasing the priority of jobs that have waited a long time).

Ví dụ mẫu · Worked example

Three processes arrive at time 0 with burst times $P_1 = 6$, $P_2 = 8$, $P_3 = 3$. Compare average waiting time for FCFS (in arrival order) vs SJF.

FCFS (P_1, P_2, P_3): P_1 starts at 0 (wait 0); P_2 starts at 6 (wait 6); P_3 starts at 14 (wait 14). Average wait $= \dfrac{0+6+14}{3} = \dfrac{20}{3} \approx 6.67$.

SJF (shortest first: P_3, P_1, P_2): P_3 starts at 0 (wait 0); P_1 starts at 3 (wait 3); P_2 starts at 9 (wait 9). Average wait $= \dfrac{0+3+9}{3} = 4$.

SJF gives the *lower* average waiting time here — this is a general, provable property of SJF among non-preemptive scheduling algorithms.

Ví dụ mẫu · Worked example

Four processes arrive at time 0 with burst times $P_1 = 5$, $P_2 = 2$, $P_3 = 1$, $P_4 = 4$, run with **Round Robin** using a time quantum of 2. Determine the order processes complete.

Round-robin cycles through the ready queue, giving each process at most 2 time units per turn:

Turn 1: P_1 runs 2 (remaining 3), P_2 runs 2 (remaining 0, **finishes**), P_3 runs 1 (remaining 0, **finishes**), P_4 runs 2 (remaining 2).

Turn 2: P_1 runs 2 (remaining 1), P_4 runs 2 (remaining 0, **finishes**).
Turn 3: P_1 runs 1 (remaining 0, **finishes**).
Completion order: P_2, P_3, P_4, P_1 — notice the two shortest jobs ($P_3 = 1$, $P_2 = 2$) finish very early despite not being first in line, because Round Robin gives every process a turn quickly rather than making short jobs wait behind long ones.

VN **GIẢI THÍCH TIẾNG VIỆT**

FCFS: chạy theo thứ tự đến trước, đơn giản nhưng có thể gây hiệu ứng đoàn tàu (convoy effect — job ngắn bị kẹt sau job dài). **SJF**: chạy job ngắn nhất trước, giảm thời gian chờ trung bình nhưng có thể "bỏ đói" job dài. **Round Robin**: chia đều một lượng tử thời gian cố định cho từng tiến trình theo vòng, công bằng nhưng tốn chi phí chuyển ngữ cảnh (context switch). **Priority**: ưu tiên job quan trọng trước, cần kỹ thuật "aging" để tránh bỏ đói job ưu tiên thấp.

6.3 Memory management: paging and virtual memory

Paging divides logical (program) memory into fixed-size **pages**, mapped to equally-sized physical **frames** in RAM via a **page table**. This allows **virtual memory** to give each program the illusion of more memory than physically exists: pages not currently needed are **swapped** out to disk, and brought back in on demand.

A **page fault** occurs when a program accesses a page that is not currently loaded in RAM, forcing the OS to pause the process and load that page from disk — this is far slower than a normal memory access, so excessive page faults (**thrashing**) severely degrade performance. Given a page size S and logical address A: **page number** $= \lfloor A/S \rfloor$, and **offset** $= A - (\text{page number} \times S)$.

Ví dụ mẫu · Worked example

With a page size of 1024 bytes, find the page number and offset for logical address 2050.
Page number $= \left\lfloor \dfrac{2050}{1024} \right\rfloor = 2$. Offset $= 2050 - (2 \times 1024) = 2050 - 2048 = 2$. So logical address 2050 is **page 2, offset 2**.

Ví dụ mẫu · Worked example

With a page size of 4096 bytes, find the page number and offset for logical address 9000.
Page number $= \left\lfloor \dfrac{9000}{4096} \right\rfloor = 2$ (since $2 \times 4096 = 8192 \leq 9000 < 3 \times 4096 = 12288$). Offset $= 9000 - 8192 = 808$. So the address is **page 2, offset 808**.

VN **GIẢI THÍCH TIẾNG VIỆT**

Phân trang (paging): chia bộ nhớ thành các **trang (page)** kích thước cố định, ánh xạ tới các **khung (frame)** trong RAM qua bảng trang. **Bộ nhớ ảo (virtual memory)** cho phép chương trình "tưởng" có nhiều RAM hơn thực tế bằng cách hoán đổi (swap) trang ra đĩa. **Page fault**: trang cần dùng không có trong RAM, phải nạp lại từ đĩa (chậm) — nếu xảy ra quá thường xuyên gọi là **thrashing**. Công thức: số trang $= \lfloor A/S \rfloor$, offset $= A - (\text{số trang} \times S)$.

Page replacement: FIFO vs LRU

When RAM is full and a new page must be loaded, the OS must choose an existing page to evict. **FIFO** (First-In-First-Out) evicts whichever page has been in memory the *longest*, regardless of how recently it was used — simple, but can evict a page that is still frequently needed. **LRU** (Least Recently Used) evicts the page that has gone *longest without being accessed*, on the reasonable assumption that data unused for a long time is less likely to be needed again soon — generally performs better than FIFO, but requires extra bookkeeping to track access recency.

VN **GIẢI THÍCH TIẾNG VIỆT**

Thay thế trang FIFO: loại trang đã ở trong bộ nhớ lâu nhất, bất kể có đang được dùng hay không. **Thay thế trang LRU**: loại trang lâu nhất KHÔNG được truy cập — thường hiệu quả hơn FIFO vì phản ánh đúng khả năng trang đó sẽ còn được dùng lại hay không.

Segmentation

Segmentation divides a program's memory into logically meaningful, *variable-sized* segments (e.g. one segment for code, one for the stack, one for global data), unlike paging's fixed-size, logically arbitrary chunks. Segmentation matches how programmers naturally think about memory, but variable sizes make it more prone to **external fragmentation** (free memory scattered in gaps too small individually to hold a new segment). Many modern systems combine both (**segmented paging**).

VN **GIẢI THÍCH TIẾNG VIỆT**

Phân đoạn (segmentation): chia bộ nhớ thành các đoạn có kích thước THAY ĐỔI theo ý nghĩa logic (mã lệnh, ngăn xếp, dữ liệu toàn cục) — dễ hiểu với lập trình viên hơn nhưng dễ gây phân mảnh ngoài (external fragmentation) hơn so với phân trang.

6.4 Translators: compilers, interpreters, assemblers

Compiler: translates the *entire* source program into machine code *before* execution — the resulting program runs fast, but any source change requires full recompilation, and errors are typically reported all together after compilation. **Interpreter**: translates and executes source code *line by line* at runtime — slower overall execution (translation overhead repeats every run), but easier to debug interactively, since execution stops immediately at the exact faulty line (e.g. Python). **Assembler**: translates assembly-language mnemonics into machine code on a (roughly) one-to-one basis.

VN **GIẢI THÍCH TIẾNG VIỆT**

Trình biên dịch (compiler): dịch toàn bộ mã nguồn thành mã máy TRƯỚC khi chạy — chạy nhanh nhưng sửa code phải biên dịch lại toàn bộ. **Trình thông dịch (interpreter)**: dịch và chạy TỪNG DÒNG một — chạy chậm hơn nhưng dễ gỡ lỗi (dừng ngay tại dòng lỗi). **Hợp ngữ (assembler)**: dịch mã hợp ngữ (assembly) sang mã máy gần như 1-1.

Stages of compilation: (1) **Lexical analysis** — breaks source code into tokens (keywords, identifiers, operators), discarding whitespace/comments, and flags invalid symbols. (2) **Syntax analysis (parsing)** — checks tokens conform to the language's grammar rules, building a parse tree; a missing semicolon or unbalanced bracket is caught here. (3) **Semantic analysis** — checks the parsed code actually makes logical sense (e.g. type checking: adding a string to an integer without conversion is a semantic error even though it may be syntactically valid). (4) **Code generation** — produces the target machine code. (5) **Optimization** — improves the generated code's speed/size without changing its behaviour (e.g. removing unreachable code, simplifying expressions computed at compile time).

Ví dụ mẫu · Worked example

For the line of code `result = totalCost / itemCount;` classify each of the following errors by compilation stage: (a) a missing semicolon at the end of the line; (b) dividing by a variable that was declared as a `String`, not a number.

(a) A missing semicolon breaks the required grammatical structure of a statement — this is caught during **syntax analysis**. (b) Dividing by a `String` is grammatically valid in form (it looks like a normal division statement) but is logically meaningless given the variable's declared type — this is caught during **semantic analysis** (type checking).

VN **GIẢI THÍCH TIẾNG VIỆT**

Các giai đoạn biên dịch: **phân tích từ vựng** (tách mã nguồn thành token) → **phân tích cú pháp** (kiểm tra đúng ngữ pháp, ví dụ thiếu dấu chấm phẩy) → **phân tích ngữ nghĩa** (kiểm tra ý nghĩa logic, ví dụ kiểm tra kiểu dữ liệu) → **sinh mã máy** → **tối ưu hoá** (cải thiện tốc độ/kích thước mà không đổi hành vi chương trình).

6.5 Deadlock and concurrency

A **deadlock** occurs when two or more processes are each waiting for a resource held by another, so none can ever proceed. Four conditions must *all* hold simultaneously for deadlock to occur:

Necessary conditions for deadlock: (1) **Mutual exclusion** — at least one resource can only be used by one process at a time. (2) **Hold and wait** — a process holding at least one resource is waiting to acquire additional resources held by others. (3) **No pre-emption** — a resource cannot be forcibly taken from a process; it must be released voluntarily. (4) **Circular wait** — a closed chain of processes exists, each waiting for a resource held by the next. Breaking *any one* of these four conditions prevents deadlock.

Ví dụ mẫu · Worked example

Process A holds Printer 1 and is waiting for Printer 2; Process B holds Printer 2 and is waiting for Printer 1. Explain why this is a deadlock and suggest one way it could have been prevented.

This is a **deadlock**: A cannot proceed without Printer 2 (held by B), and B cannot proceed without Printer 1 (held by A) — a circular wait with no possibility of either process releasing what it holds until it gets what it needs, which never happens. One prevention strategy: enforce a fixed global **ordering** on resource requests (e.g. always request Printer 1 before Printer 2) so that circular wait becomes structurally impossible — B would then have had to request Printer 1 first, before Printer 2, avoiding the cycle entirely.

VN **GIẢI THÍCH TIẾNG VIỆT**

Deadlock (bế tắc): hai hay nhiều tiến trình chờ tài nguyên do tiến trình khác giữ, không ai tiến được. Bốn điều kiện cần đồng thời xảy ra: loại trừ tương hỗ, giữ và chờ, không tước đoạt, chờ vòng tròn — phá vỡ MỘT trong bốn điều kiện này là đủ để ngăn deadlock (ví dụ: quy định thứ tự xin tài nguyên cố định để loại bỏ chờ vòng tròn).

VN **GIẢI THÍCH TIẾNG VIỆT**

Ôn tập nhanh Unit 6: (1) OS quản lý CPU/bộ nhớ/thiết bị/tệp; đa nhiệm (1 CPU luân phiên) khác đa xử lý (nhiều lõi song song thật); (2) trạng thái tiến trình New→Ready→Running→(Waiting)→Terminated; (3) FCFS đơn giản nhưng convoy effect; SJF tối ưu thời gian chờ trung bình; Round Robin công bằng cho hệ tương tác; Priority cần aging; (4) phân trang: số trang $= \lfloor A/S \rfloor$; FIFO vs LRU thay thế trang; (5) compiler dịch hết rồi chạy, interpreter dịch-chạy từng dòng; 5 giai đoạn biên dịch; (6) deadlock cần đủ 4 điều kiện, phá 1 điều kiện là ngăn được.

6.6 Threads vs processes

A **process** is an independent running program with its own private memory space; processes cannot directly access each other's memory (enforced by the OS), making them robust but relatively expensive to create and to switch between. A **thread** is a lightweight unit of execution *within* a process; multiple threads in the same process **share** that process's memory space, making them cheaper to create and to switch between, and allowing them to communicate and share data directly and efficiently — but also meaning one thread's error (e.g. corrupting shared data) can more easily affect other threads in the same process, whereas a crashing process does not directly corrupt another process's separate memory.

Multithreading lets a single application perform several tasks concurrently within one process (e.g. a word processor's UI remaining responsive while a document autosaves in the background, both as separate threads of the same program). Because threads share memory, switching between threads of the *same* process is faster (a smaller **context switch**) than switching between entirely separate processes, which must swap out an entire private memory context.

Ví dụ mẫu · Worked example

A web browser uses one thread to render a web page's visuals and a separate thread to continue downloading page resources in the background, both within the same browser process. Explain why this improves user-perceived responsiveness compared with doing both tasks strictly one after another in a single thread.

If rendering and downloading happened strictly sequentially within one thread, the entire page would appear frozen/blank while a slow resource downloads, since the single thread cannot do anything else until that download completes. By splitting the work into **two threads** sharing the same process, the rendering thread can continue displaying already-downloaded content and remain responsive to user scrolling/interaction *while* the download thread works independently in the background — the user perceives the browser as responsive even though a slow network resource is still loading.

VN GIẢI THÍCH TIẾNG VIỆT

Tiến trình (process): chương trình chạy độc lập, có vùng nhớ riêng — chuyển đổi giữa các tiến trình tốn kém hơn nhưng an toàn hơn (lỗi tiến trình này không ảnh hưởng tiến trình khác). **Luồng (thread)**: đơn vị thực thi nhẹ hơn BÊN TRONG một tiến trình, CHIA SẺ vùng nhớ với các luồng khác cùng tiến trình — chuyển đổi nhanh hơn, giao tiếp dễ hơn, nhưng lỗi một luồng dễ ảnh hưởng luồng khác cùng tiến trình.

6.7 Disk scheduling algorithms

When multiple processes request data from the same hard disk at around the same time, a **disk scheduler** decides the order in which to service these requests, since the physical read/write head can only be in one place at a time, and unnecessary head movement (**seek time**) wastes time.

FCFS (First-Come-First-Served) disk scheduling services requests strictly in arrival order — simple and fair, but can cause excessive head movement if requests happen to arrive in a "scattered" order across the disk. **SSTF** (Shortest Seek Time First) always services whichever pending request is *physically closest* to the head's current position — minimises total head movement in the short term, but can indefinitely **starve** a request that happens to be located far from a cluster of closer, constantly-arriving requests. **SCAN** ("elevator algorithm") moves the head steadily in one direction, servicing every request it passes, until it reaches the end of the disk, then reverses direction — like a lift servicing every floor along its current direction of travel, this avoids the starvation risk of SSTF while still keeping head movement reasonably efficient.

Ví dụ mẫu · Worked example

The disk head is currently at track 50. Pending requests are for tracks 20, 90, 55, 40, 75 (in that arrival order). Using **SSTF**, determine the order requests are serviced.

From track 50, distances to each pending request: 20 ($|50-20|=30$), 90 (40), 55 (5), 40 (10), 75 (25). The *closest* is 55 (distance 5) — serviced first. From 55, remaining distances: 20 (35), 90 (35), 40 (15), 75 (20); closest is 40 — serviced next. From 40: 20 (20), 90 (50), 75 (35); closest is 20 — serviced next. From 20: 90 (70), 75 (55); closest is 75 — serviced next, leaving 90 last. Service order: **55, 40, 20, 75, 90** — notice this ignores arrival order entirely, always jumping to whichever pending request is currently nearest.

VN GIẢI THÍCH TIẾNG VIỆT

Lập lịch đĩa (disk scheduling): quyết định thứ tự phục vụ yêu cầu đọc/ghi đĩa để giảm thời gian di chuyển đầu đọc (seek time). **FCFS**: theo thứ tự đến, đơn giản nhưng có thể di chuyển đầu đọc nhiều. **SSTF**: luôn phục vụ yêu cầu GẦN NHẤT hiện tại — giảm di chuyển nhưng có thể bỏ đói yêu cầu ở xa. **SCAN** (thuật toán thang máy): đầu đọc di chuyển một hướng, phục vụ mọi yêu cầu gặp phải, rồi đảo hướng — tránh bỏ đói mà vẫn hiệu quả.

6.8 Virtual machines and hypervisors

A **virtual machine (VM)** is a software-based emulation of a complete computer, running its own guest operating system as if it were separate physical hardware, even though it actually shares the underlying physical machine's resources with other VMs.

A **hypervisor** is the software layer that creates and manages VMs, allocating each one a share of the physical CPU, memory, and storage, and keeping VMs isolated from one another (one VM cannot directly access another VM's memory, similar in spirit to how the OS isolates separate processes). **Type 1 (bare-metal) hypervisors** run directly on the physical hardware, without a separate host operating system underneath — typically used in data centres/servers for better performance and security. **Type 2 (hosted) hypervisors** run as an application *on top of* a conventional host operating system (e.g. running a Linux VM inside a virtualisation app on a Windows laptop) — simpler to set up, but with somewhat more performance overhead from the extra software layer beneath it.

Ví dụ mẫu · Worked example

A cloud provider runs 20 separate customers' virtual servers on one powerful physical server, each customer believing they have their own dedicated machine. Explain the role of the hypervisor in making this possible and secure.

The **hypervisor** creates and manages 20 separate VMs on the one physical server, allocating each a defined share of the physical CPU cycles, memory, and storage, and presenting each with the illusion of dedicated, private hardware. Critically, the hypervisor also enforces **isolation** between VMs: a customer's VM cannot directly read or interfere with another customer's VM's memory or data, even though they physically share the same underlying hardware — this isolation is what allows a cloud provider to safely host multiple, mutually-untrusted customers on shared physical infrastructure.

VN **GIẢI THÍCH TIẾNG VIỆT**

Máy ảo (VM): mô phỏng phần mềm của một máy tính hoàn chỉnh, chạy hệ điều hành khách riêng dù chia sẻ phần cứng vật lý với các VM khác. **Hypervisor**: lớp phần mềm tạo và quản lý VM, phân chia CPU/bộ nhớ/lưu trữ và CÁCH LY các VM với nhau. **Type 1 (bare-metal)**: chạy trực tiếp trên phần cứng — hiệu năng/bảo mật tốt hơn, dùng cho server. **Type 2 (hosted)**: chạy như một ứng dụng trên hệ điều hành host — dễ cài đặt hơn nhưng có thêm overhead.

6.9 Processing modes: batch, interactive and real-time

Different applications require fundamentally different scheduling philosophies from the OS, depending on how and when results are needed.

Batch processing collects a large group of jobs and processes them together, without a user waiting interactively for immediate results (e.g. an overnight payroll run processing every employee's pay at once) — efficient for large volumes of similar work, but unsuitable when an immediate response is needed. **Interactive processing** responds to a user's input essentially immediately, supporting a back-and-forth dialogue (e.g. a word processor responding to each keystroke) — prioritises responsiveness for a human user over raw processing efficiency. **Real-time processing** (Unit 7) must respond within a guaranteed, bounded time limit, as discussed for embedded control systems — the strictest of the three, since even interactive systems tolerate occasional small, variable delays that a real-time system cannot.

Ví dụ mẫu · Worked example

A bank runs its overnight interest-calculation job across all 2 million customer accounts starting at 1 a.m., completing before branches open at 9 a.m. Identify the processing mode and explain why it is appropriate here rather than interactive processing.

This is **batch processing**: a large, uniform volume of work (identical interest calculations across every account) is processed together as one big job, with no individual customer waiting interactively for their own personal result. This is appropriate because there is no need for an immediate, per-customer interactive response at 1 a.m. — processing all accounts together as one efficient overnight batch, well before the 9 a.m. deadline, is a far more efficient use of computing resources than handling each account as a separate real-time interactive request.

VN GIẢI THÍCH TIẾNG VIỆT

Xử lý theo lô (batch processing): gom nhiều công việc xử lý cùng lúc, không có người dùng chờ kết quả ngay lập tức (tính lương qua đêm) — hiệu quả với khối lượng lớn công việc giống nhau. **Xử lý tương tác (interactive)**: phản hồi gần như ngay lập tức theo thao tác người dùng. **Xử lý thời gian thực (real-time)**: phải phản hồi trong giới hạn thời gian đảm bảo — nghiêm ngặt nhất trong ba loại.

6.10 Interrupts and interrupt handling

An **interrupt** is a signal sent to the CPU indicating that an event requiring immediate attention has occurred — from hardware (e.g. a keypress, a disk finishing a read, a network packet arriving) or from software (e.g. a program requesting an OS service, or an error condition).

On receiving an interrupt, the CPU: (1) finishes its *current* instruction (interrupts are not mid-instruction); (2) saves the current process's state (registers, PC) so it can resume later exactly where it left off; (3) looks up the appropriate **interrupt service routine (ISR)** — a small piece of code specifically written to handle that type of interrupt — using an **interrupt vector table** (a lookup table mapping interrupt types to their ISR's address); (4) executes the ISR; (5) restores the saved state and resumes the interrupted process exactly where it left off. Interrupts let the CPU respond promptly to events without wasting time continuously **polling** (repeatedly checking "has anything happened yet?" in a loop) — a much more efficient approach when events are relatively infrequent and unpredictable.

Ví dụ mẫu · Worked example

Explain why using interrupts to detect a keypress is more efficient than the CPU repeatedly polling the keyboard in a tight loop thousands of times per second, "just in case" a key has been pressed.

Polling would force the CPU to spend a large amount of processing time repeatedly checking the keyboard status, even during the (typically very long, relative to CPU speed) intervals between actual keypresses — this is wasted computation that could otherwise be used running other processes. Using **interrupts** instead, the CPU can get on with other useful work entirely, and the keyboard hardware itself only signals the CPU (via an interrupt) at the exact moment a key is actually pressed — the CPU's attention is requested only when genuinely needed, rather than constantly "checking in" on the small chance something has happened.

VN GIẢI THÍCH TIẾNG VIỆT

Ngắt (interrupt): tín hiệu gửi tới CPU báo có sự kiện cần xử lý ngay (bàn phím, đĩa xong việc, lỗi). CPU xử lý ngắt: hoàn tất lệnh hiện tại → lưu trạng thái tiến trình → tra **bảng vector ngắt** tìm **chương trình xử lý ngắt (ISR)** → chạy ISR → khôi phục trạng thái và tiếp tục tiến trình bị ngắt. Ngắt hiệu quả hơn **polling** (liên tục kiểm tra "có gì xảy ra chưa?") vì CPU chỉ bị làm phiền

đúng lúc cần thiết.

6.11 Memory fragmentation

Poorly-managed memory allocation over time leads to **fragmentation** — wasted or unusable memory — of two distinct kinds.

External fragmentation (introduced with segmentation, earlier in this chapter): free memory exists in total, but is scattered across many small, non-contiguous gaps, none individually large enough to satisfy a new request even though their *sum* would be sufficient. **Internal fragmentation**: a process is allocated a fixed-size block (e.g. one page) larger than it actually needs, wasting the unused remainder *within* its own allocated block — this can occur even with paging's fixed-size blocks, whenever a process's actual data does not exactly fill a whole number of pages (the last, partially-used page wastes the remainder). Paging trades external fragmentation (largely eliminated, since any free frame fits any page) for a small amount of internal fragmentation (wasted space within the last page of each process); segmentation has the opposite trade-off.

Ví dụ mẫu · Worked example

A process needs 4500 bytes of memory, and the system uses 2048-byte pages. Calculate how many pages must be allocated, and the resulting internal fragmentation.

Pages needed $= \left\lceil \frac{4500}{2048} \right\rceil = \lceil 2.197 \rceil = 3$ pages (partial pages must still be allocated as a whole page). Total memory allocated $= 3 \times 2048 = 6144$ bytes. Internal fragmentation (wasted space) $= 6144 - 4500 =$ **1644 bytes** — memory reserved for this process but never actually used, since only whole pages can be allocated even though the process's true need falls partway through the third page.

VN **GIẢI THÍCH TIẾNG VIỆT**

Phân mảnh ngoài (external fragmentation): tổng bộ nhớ trống đủ nhưng bị chia nhỏ, rải rác thành nhiều khoảng không đủ lớn cho yêu cầu mới. **Phân mảnh trong (internal fragmentation)**: một tiến trình được cấp một khối kích thước CỐ ĐỊNH lớn hơn nhu cầu thực tế, lãng phí phần dư BÊN TRONG khối đã cấp — xảy ra ngay cả với phân trang khi trang cuối không được dùng hết. Phân trang giảm phân mảnh ngoài nhưng vẫn có chút phân mảnh trong; phân đoạn thì ngược lại.

6.12 Memory allocation strategies

When variable-sized memory requests must be fitted into available free blocks (relevant to segmentation, earlier in this chapter), the OS must choose *which* free block to use for each request.

First-fit: allocate the request to the *first* free block found that is large enough, scanning from the start of memory each time — fast to compute, but can leave many small, awkward leftover fragments near the start of memory. **Best-fit**: search *all* free blocks and choose the *smallest* one that is still large enough — minimises the leftover wasted space for this individual allocation, but requires scanning every free block (slower) and can still leave many very small, barely-usable leftover fragments scattered throughout memory over time. **Worst-fit**: deliberately choose the *largest* available free block — leaves a larger, potentially more *useful* leftover fragment after allocation (rather than a tiny sliver), at the cost of using up large blocks quickly, potentially leaving no large block available for a future large request.

Ví dụ mẫu · Worked example

Free memory blocks of sizes 100, 500, 200, and 300 (KB) are available, in that order. A new request for 150 KB arrives. State which block first-fit, best-fit, and worst-fit would each choose.

First-fit: scans from the start and picks the *first* block ≥ 150 KB — skips the 100 KB block (too small), picks the **500 KB** block (leaving 350 KB free there). **Best-fit**: examines all blocks ≥ 150 KB (500, 200, 300) and picks the *smallest* suitable one, the **200 KB** block (leaving only 50 KB free there — a tight fit). **Worst-fit**: picks the *largest* available block, the **500 KB** block (leaving 350 KB free, a large, more flexible leftover for future requests).

VN **GIẢI THÍCH TIẾNG VIỆT**

First-fit: chọn khối trống ĐẦU TIÊN đủ lớn — nhanh nhưng có thể để lại nhiều mảnh nhỏ. **Best-fit**: chọn khối trống NHỎ NHẤT vẫn đủ lớn — giảm lãng phí cho lần cấp phát này nhưng chậm hơn và vẫn có thể để lại nhiều mảnh rất nhỏ. **Worst-fit**: chọn khối trống LỚN NHẤT — để lại phần dư lớn hơn, hữu ích hơn, nhưng dùng hết khối lớn nhanh, có thể thiếu khối lớn cho yêu cầu tương lai.

6.13 Deadlock detection and recovery

Rather than *preventing* deadlock by structurally eliminating one of the four necessary conditions (earlier in this chapter), some systems instead allow deadlock to potentially occur, but actively **detect** and **recover** from it when it does.

Deadlock detection: the OS periodically examines resource-allocation data (which process holds/waits for which resource) looking for a circular-wait pattern — often modelled as a **resource-allocation graph**, where a cycle in the graph indicates deadlock. **Recovery** strategies once deadlock is detected: **process termination** (kill one or more of the deadlocked processes, forcibly freeing their held resources so others can proceed — simple, but the killed process loses all unsaved work); **resource pre-emption** (forcibly take a resource away from one process and give it to another, rolling the first process back to an earlier safe state). Detection-and-recovery trades the overhead of prevention's stricter operating rules for occasional, more disruptive recovery action when deadlock does occur — appropriate when deadlock is expected to be rare.

Ví dụ mẫu · Worked example

An OS's deadlock detection algorithm finds a cycle: Process A holds Resource 1 and waits for Resource 2; Process B holds Resource 2 and waits for Resource 1. Describe one recovery action the OS could take, and one consequence of that action.

The OS could **terminate Process B**, forcing it to release Resource 2 — Resource 2 then becomes available for Process A, which can proceed and eventually release Resource 1. **Consequence**:

any unsaved work or progress made by Process B is lost entirely, and Process B (or the user relying on it) will need to restart that work from scratch — a real cost of this simple recovery strategy, which is why **prevention** (avoiding deadlock ever occurring in the first place) is often preferred for critical systems where losing a process's work is unacceptable.

VN **GIẢI THÍCH TIẾNG VIỆT**

Phát hiện deadlock: định kỳ kiểm tra dữ liệu cấp phát tài nguyên để tìm mẫu chờ vòng tròn (thường dùng đồ thị cấp phát tài nguyên — chu trình trong đồ thị báo hiệu deadlock). **Khôi phục**: **chấm dứt tiến trình** (đơn giản nhưng mất công việc chưa lưu) hoặc **tước đoạt tài nguyên** (lấy lại tài nguyên, đưa tiến trình về trạng thái an toàn trước đó). Cách tiếp cận này đánh đổi chi phí phòng ngừa nghiêm ngặt để lấy hành động khôi phục gây gián đoạn hơn khi deadlock thực sự xảy ra.

6.14 Practice

Bài tập luyện tập

Four processes arrive at time 0 with burst times $P_1 = 4$, $P_2 = 1$, $P_3 = 8$, $P_4 = 2$. Using SJF (non-preemptive), what is P_3's waiting time?

(A) 0
(C) 8
(B) 7
(D) 15

Lời giải chi tiết

SJF order by burst time: $P_2(1), P_4(2), P_1(4), P_3(8)$. Start times: P_2 at 0, P_4 at 1, P_1 at 3, P_3 at 7. P_3 waits until all shorter jobs finish: waiting time $= 7$. **(B)**.

Bài tập luyện tập

Which process state describes a process that is loaded and eligible to run, but is not currently being executed because the CPU is busy with another process?

(A) New
(C) Running
(B) Ready
(D) Waiting

Lời giải chi tiết

Ready means the process has everything it needs and is waiting only for CPU time; **Running** means it currently has the CPU; **Waiting** means it is blocked on I/O, not just waiting for the CPU. **(B)**.

Bài tập luyện tập

With a page size of 4096 bytes, find the page number and offset for logical address 9000.

Lời giải chi tiết

Page number $= \left\lfloor \frac{9000}{4096} \right\rfloor = 2$ (since $2 \times 4096 = 8192 \leq 9000 < 3 \times 4096 = 12288$). Offset $= 9000 - 8192 = 808$. So the address is **page 2, offset 808**.

Bài tập luyện tập

A programmer wants to test small snippets of code interactively, seeing results immediately after each line, and is not concerned about final execution speed. Which translator best suits this workflow, and why?

(A) Compiler, because it is faster overall

(B) Interpreter, because it executes and reports results line by line

(C) Assembler, because it works with mnemonics

(D) None; translators are unrelated to interactive testing

Lời giải chi tiết

An **interpreter** translates and runs code one line/statement at a time, giving immediate feedback after each line — ideal for interactive testing, even though the resulting execution is slower than compiled code. **(B)**.

Bài tập luyện tập

Explain why Round Robin scheduling is considered fairer than FCFS for an interactive multi-user system, using the idea of a time quantum in your answer.

Lời giải chi tiết

In **FCFS**, a process that arrives first — even a very long one — monopolises the CPU until it finishes, forcing every later process (however short) to wait its entire duration (the *convoy effect*). **Round Robin** instead gives each ready process only a fixed **time quantum** before moving to the next process in a circular queue; this guarantees every process gets regular, bounded access to the CPU, keeping the system responsive for all interactive users rather than favouring whichever process merely arrived earliest.

Bài tập luyện tập

Explain why priority scheduling can lead to starvation, and describe how ”aging” fixes this.

Lời giải chi tiết

Under pure priority scheduling, a low-priority process only ever runs once no higher-priority process is ready — if higher-priority processes keep arriving continuously, a low-priority process could theoretically wait forever, which is called **starvation**. **Aging** fixes this by gradually *increasing* the priority of a process the longer it waits, so that eventually even an initially low-priority process's priority rises high enough to guarantee it gets scheduled, without needing to abandon priority scheduling altogether.

Bài tập luyện tập

Three processes arrive at time 0 with burst times $P_1 = 10$, $P_2 = 2$, $P_3 = 3$. Calculate the average waiting time under FCFS (arrival order) and under SJF, and state which is lower.

Lời giải chi tiết

FCFS (P_1, P_2, P_3): P_1 waits 0, P_2 waits 10, P_3 waits 12. Average $= \dfrac{0+10+12}{3} = \dfrac{22}{3} \approx 7.33$. **SJF** ($P_2, P_3, P_1$ by burst time $2, 3, 10$): P_2 waits 0, P_3 waits 2, P_1 waits 5. Average $= \dfrac{0+2+5}{3} = \dfrac{7}{3} \approx 2.33$. **SJF gives the lower average waiting time** — placing the long job (P_1) last prevents it from forcing the two short jobs to wait through its entire burst time.

Bài tập luyện tập

Explain the difference between a page fault and normal page access, and state one consequence of frequent page faults ("thrashing").

Lời giải chi tiết

A **normal page access** finds the required page already loaded in a RAM frame, so the CPU can read/write it directly at RAM speed. A **page fault** occurs when the required page is *not* currently in RAM, forcing the OS to pause the requesting process and load the page from much slower secondary storage (disk) before continuing. If page faults happen very frequently (**thrashing**) — e.g. because too many processes compete for too little RAM — the system can spend more time swapping pages in and out than doing useful work, causing severe overall performance degradation.

Bài tập luyện tập

Compare FIFO and LRU page replacement using an example: RAM holds pages loaded in order A, B, C (A oldest). Page D must now be loaded and one page evicted. Page B was just accessed a moment ago; A and C have not been accessed in a long time. Which page does each algorithm evict?

Lời giải chi tiết

FIFO evicts whichever page has been in memory the *longest*, regardless of recent use — that is **Page A** (loaded first). **LRU** evicts whichever page has gone the *longest without being accessed* — since B was just accessed, and A/C have not been accessed recently, LRU would evict either A or C depending on which of those two was accessed *least* recently; it would specifically *not* evict B, since B has clearly been used most recently of the three.

Bài tập luyện tập

Classify the following as a lexical, syntax, or semantic error: a program declares `Age` as an integer, then later executes `Age = Age + "5"` (adding a string literal to an integer variable without conversion).

Lời giải chi tiết

The statement is grammatically well-formed (an assignment with a plus operator between two operands) — the problem is that the operand types are incompatible for direct addition given their declared types. This is a **semantic error** (caught during semantic analysis / type checking), not a lexical or syntax error.

Bài tập luyện tập

Explain one advantage of an interpreter over a compiler specifically during the software *development* phase (not for the finished product).

Lời giải chi tiết

During development, code changes constantly and programmers need fast feedback on whether a change works. An **interpreter** translates and runs code line by line, so it can immediately execute (and report an error in) the exact line just written, without waiting for the entire program to be translated first — letting a developer test small changes almost instantly. A **compiler**, by contrast, must retranslate the *entire* program before it can be run at all, which is slower for this rapid edit-test-edit cycle even though the compiled program ultimately runs faster once finished.

Bài tập luyện tập

Two processes, A and B, each need both a scanner and a printer to complete their task. A has acquired the scanner and is waiting for the printer; B has acquired the printer and is waiting for the scanner. Identify which of the four deadlock conditions is most directly illustrated by this exact situation, and explain.

Lời giải chi tiết

This most directly illustrates **circular wait**: A is waiting on a resource (printer) held by B, and B is waiting on a resource (scanner) held by A, forming a closed cycle of two processes each waiting on the other. (The scenario also relies on the other three conditions holding — mutual exclusion of each device, hold-and-wait behaviour, and no pre-emption of the devices — but the circular chain of dependency itself is the condition most specifically shown here.)

Bài tập luyện tập

Suggest one practical operating-system strategy that prevents deadlock by breaking the ”hold and wait” condition specifically.

Lời giải chi tiết

Require every process to request *all* the resources it will need **at once**, before starting execution, rather than acquiring some resources and then later requesting more while still holding the first ones. If a process cannot obtain every resource it needs immediately, it acquires none and must wait/retry later — this directly eliminates ”hold and wait,” since a process is never simultaneously holding one resource while blocked waiting for another.

Bài tập luyện tập

Explain why segmentation is more prone to external fragmentation than paging, using the concept of segment/page size.

Lời giải chi tiết

Paging divides memory into *fixed-size* pages/frames, so any free frame can hold any page — free space is never "the wrong shape" for a waiting page. **Segmentation** uses *variable-sized* segments matched to logical program units (code, stack, data); as segments are loaded and removed over time, the free memory left behind is scattered in gaps of varying, unpredictable sizes. A new segment may fail to fit into any single gap even if the *total* free memory is more than enough, because no single gap is large enough on its own — this scattered, unusable free space is **external fragmentation**.

Bài tập luyện tập

A single-core CPU appears to run a web browser, a music player, and a word processor "at the same time." Explain, using the term "time quantum," how this is actually achieved.

Lời giải chi tiết

A single core can only truly execute one process at any given instant. The OS achieves the *appearance* of simultaneous execution through **multitasking**: the scheduler rapidly switches the CPU between the browser, music player, and word processor, giving each one a short **time quantum** of CPU time in turn (often just milliseconds), then performing a **context switch** to the next process. Because these switches happen so fast relative to human perception, all three applications appear to run simultaneously, even though only one is genuinely executing at any single instant.

Bài tập luyện tập

Explain why switching between two threads of the *same* process is generally faster than switching between two entirely separate processes.

Lời giải chi tiết

Switching between separate **processes** requires the OS to save and later restore an entire private memory context (memory mappings, open file handles, and more) for each process, since each process has its own completely separate address space. Switching between **threads** of the *same* process only requires saving/restoring each thread's own small execution state (its registers, stack pointer, program counter), since the shared memory space, open files, and other process-wide resources do not need to be swapped out at all — this smaller context switch is significantly cheaper and faster.

Bài tập luyện tập

Explain one risk of multithreading that does not apply to using entirely separate processes, using the concept of shared memory.

Lời giải chi tiết

Because threads within the same process **share** the same memory space, a bug in one thread — such as writing incorrect data to a shared variable, or corrupting a data structure while another thread is reading it at the same time (a **race condition**) — can directly corrupt data used by other threads in that process, or cause unpredictable, hard-to-reproduce bugs. Separate **processes** do not share memory directly (enforced by the OS), so an error in one process cannot directly corrupt another process's private memory in the same way, making processes more robustly isolated from each other's failures.

Bài tập luyện tập

A disk head is at track 30. Pending requests (in arrival order) are for tracks 10, 65, 35, 80. Using SSTF, determine the full service order.

Lời giải chi tiết

From track 30: distances are 10 (20), 65 (35), 35 (5), 80 (50). Closest: **35** (distance 5) — serviced first. From 35: remaining distances are 10 (25), 65 (30), 80 (45); the smallest is 25, so **10** is serviced next. From 10: remaining distances are 65 (55) and 80 (70); the smaller is 55, so **65** is serviced next, leaving **80** last. Service order: **35, 10, 65, 80**.

Bài tập luyện tập

Explain why SSTF disk scheduling can lead to starvation of a distant request, using a scenario where new nearby requests keep arriving.

Lời giải chi tiết

SSTF always chooses whichever *currently pending* request is physically closest to the disk head, without any regard for how long a request has already been waiting. If new requests keep arriving in the region where the head currently is (or nearby), those newly-arrived, closer requests will repeatedly be chosen ahead of an older but more distant request — the distant request could, in principle, wait indefinitely as long as a steady stream of closer requests keeps taking priority over it, a form of **starvation**. The SCAN algorithm avoids this specific problem by committing to sweep past every pending request in the current direction before reversing, guaranteeing every request is eventually serviced within one full sweep.

Bài tập luyện tập

Explain why running several VMs on one physical server is more resource-efficient than dedicating a separate physical server to each workload, for workloads that individually only use a small fraction of a server's total capacity.

Lời giải chi tiết

If each workload only uses, say, 15% of a dedicated physical server's capacity, running it on its own separate physical machine leaves roughly 85% of that expensive hardware sitting idle and wasted. Running multiple such workloads as separate **VMs** on one shared physical server, managed by a **hypervisor**, allows the underlying hardware's capacity to be pooled and allocated dynamically across all the VMs — several lightly-loaded workloads can share one physical machine's resources far more fully, substantially reducing the amount of idle, wasted hardware

capacity (and the associated cost of purchasing, powering, and maintaining many separate underused physical servers).

Bài tập luyện tập

Explain why a Type 1 (bare-metal) hypervisor generally offers better performance than a Type 2 (hosted) hypervisor.

Lời giải chi tiết

A **Type 2** hypervisor runs as an application on top of a full host operating system, meaning every VM's requests to the underlying hardware must pass through an additional software layer (the host OS) before even reaching the hypervisor itself — adding extra processing overhead at each step. A **Type 1** hypervisor runs directly on the physical hardware with no separate host OS underneath it at all, so VM requests reach the hypervisor (and hence the real hardware) more directly, with one less layer of software overhead — this is why performance-critical data-centre and server environments typically use Type 1 hypervisors.

Bài tập luyện tập

A university's exam-result processing system compiles and finalises all students' results together once, several days after the exam period ends, rather than as each script is marked. Identify the processing mode and give one advantage of this choice.

Lời giải chi tiết

This is **batch processing**: all results are gathered and processed together as one large job, rather than each individual result being processed and released the moment it becomes available. One advantage: processing all results together allows the university to apply consistency checks and any necessary grade moderation/scaling *across the entire cohort at once* before any results are released, which would be far more difficult (or produce inconsistent results across students) if each result were processed and finalised independently and immediately as it came in.

Bài tập luyện tập

Explain why a hard real-time system (e.g. an airbag controller) cannot simply be built using ordinary interactive-processing scheduling, even though interactive systems also aim to respond quickly.

Lời giải chi tiết

Interactive processing aims for responses that *feel* fast to a human user, but tolerates occasional small, variable delays (e.g. a brief lag while an application is busy) without this being considered a failure. A **hard real-time** system instead requires an *absolute guarantee* that a response will occur within a strict deadline, every single time, because a late response (however rare) constitutes complete system failure with potentially catastrophic consequences. Ordinary interactive scheduling provides no such formal guarantee — it optimises for good *typical* responsiveness, not a guaranteed *worst-case* bound — so hard real-time systems require specialised real-time scheduling algorithms specifically designed to provide provable worst-case timing guarantees.

Bài tập luyện tập

Free memory blocks of sizes 250, 100, 400, and 150 (KB) are available, in that order. A request for 120 KB arrives. State which block first-fit and best-fit would each select.

Lời giải chi tiết

First-fit: scanning from the start, the 250 KB block is the first one ≥ 120 KB (the 100 KB block is skipped as too small), so **first-fit selects the 250 KB block** (leaving 130 KB free). **Best-fit**: examining all blocks ≥ 120 KB (250, 400, 150), the smallest suitable one is the **150 KB block** (leaving only 30 KB free — the tightest possible fit among the eligible blocks).

Bài tập luyện tập

Explain one disadvantage of best-fit memory allocation, despite it minimising wasted space for each individual allocation.

Lời giải chi tiết

Although best-fit minimises the leftover space for any single allocation, it tends to create many very small leftover fragments scattered throughout memory over time (since it deliberately picks the tightest-fitting block, the remainder is often too small to usefully satisfy any future request) — contributing significantly to **external fragmentation** in the long run. Best-fit also requires scanning *every* free block to find the smallest suitable one, making it slower to compute than first-fit, which can stop as soon as it finds the first block that fits.

Bài tập luyện tập

Explain why a system relying on deadlock detection and recovery (rather than prevention) might be an acceptable design choice for a non-critical desktop application, but a poor choice for a hospital's life-support control system.

Lời giải chi tiết

Detection-and-recovery accepts that deadlock *can* occasionally occur, then deals with it after the fact — usually by terminating a process and losing its unsaved work, or rolling back to an earlier state. For a **non-critical desktop application**, an occasional deadlock causing one program to close and lose unsaved changes is inconvenient but rarely catastrophic. For a **hospital life-support system**, however, a process controlling critical equipment being forcibly terminated (or briefly stalled while deadlock is detected and resolved) could directly endanger a patient's life — here, deadlock **prevention** (structurally guaranteeing deadlock can never occur in the first place, by eliminating one of the four necessary conditions) is essential, since even a brief, rare deadlock is an unacceptable risk.

Bài tập luyện tập

Explain why a resource-allocation graph containing a cycle indicates a deadlock, using the definition of circular wait.

Lời giải chi tiết

A **resource-allocation graph** represents each process and resource as a node, with edges showing which process holds which resource and which process is waiting for which resource. A **cycle** in this graph means there exists a closed chain: Process 1 waits for a resource held by Process 2, which waits for a resource held by Process 3, ..., which eventually waits for a resource held by Process 1 again — this is precisely the definition of **circular wait**, one of the four necessary conditions for deadlock (Unit 6, earlier). If such a cycle exists (and each resource involved can only be held by one process at a time), none of the processes in the cycle can ever proceed, confirming a genuine deadlock.

Bài tập luyện tập

Five processes arrive at time 0 with burst times $P_1 = 3, P_2 = 5, P_3 = 2, P_4 = 7, P_5 = 1$. Calculate the average waiting time using SJF (non-preemptive).

Lời giải chi tiết

SJF order by burst time: $P_5(1), P_3(2), P_1(3), P_2(5), P_4(7)$. Start times: P_5 at 0, P_3 at 1, P_1 at 3, P_2 at 6, P_4 at 11. Waiting times: $P_5 = 0$, $P_3 = 1$, $P_1 = 3$, $P_2 = 6$, $P_4 = 11$. Average $= \dfrac{0+1+3+6+11}{5} = \dfrac{21}{5} = 4.2$.

Bài tập luyện tập

Explain why an operating system uses virtual memory rather than simply requiring every program to fit entirely within physical RAM.

Lời giải chi tiết

Requiring every program to fit entirely in physical RAM would strictly limit the size and number of programs that could run to whatever RAM happens to be physically installed — a serious constraint given RAM is expensive relative to secondary storage. **Virtual memory** lets the OS present each program with the illusion of a large, contiguous address space, transparently swapping less-currently-needed pages out to slower secondary storage (disk) and only keeping actively-needed pages in RAM at any moment — allowing more programs (or larger programs) to run than physical RAM alone could hold, at the cost of occasional slower page-fault-driven disk access when a needed page must be swapped back in.

Bài tập luyện tập

With a page size of 2048 bytes, find the page number and offset for logical address 5000.

Lời giải chi tiết

Page number $= \left\lfloor \dfrac{5000}{2048} \right\rfloor = 2$ (since $2 \times 2048 = 4096 \leq 5000 < 3 \times 2048 = 6144$). Offset $= 5000 - 4096 = 904$. Address is **page 2, offset 904**.

Bài tập luyện tập

Explain why an operating system uses separate "Ready" and "Waiting" states rather than merging them into one single "Not Running" state.

Lời giải chi tiết

Merging these into a single state would hide critically important information the scheduler needs: a **Ready** process has everything it needs and is only waiting for CPU time — the scheduler could select it to run at any moment. A **Waiting** process is blocked on an external event (e.g. disk I/O) and *cannot* usefully run even if given the CPU right now — selecting it would simply waste a CPU time slot on a process that cannot make progress. Keeping these as distinct states lets the scheduler correctly select only from truly Ready processes, avoiding wasted CPU allocations to processes that are not actually able to proceed yet.

Bài tập luyện tập

Explain the purpose of the ”interrupt vector table” in interrupt handling, using an example of a keyboard interrupt versus a disk-read-complete interrupt.

Lời giải chi tiết

The **interrupt vector table** maps each distinct *type* of interrupt to the memory address of the specific **interrupt service routine (ISR)** responsible for handling it. When a *keyboard* interrupt occurs, the CPU looks up the keyboard interrupt's entry in this table to find and jump to the keyboard-handling ISR (e.g. reading the pressed key and storing it in a buffer); when a *disk-read-complete* interrupt occurs instead, the CPU looks up a completely different entry in the same table, jumping to a different ISR appropriate for disk operations (e.g. copying the read data into the requesting process's memory). This table lets the CPU respond correctly and quickly to whichever specific type of interrupt has occurred, without needing to check every possible interrupt type in sequence.

Bài tập luyện tập

A process needs 10,000 bytes of memory, and the system uses 4096-byte pages. Calculate how many pages must be allocated and the resulting internal fragmentation.

Lời giải chi tiết

Pages needed $= \left\lceil \frac{10{,}000}{4096} \right\rceil = \lceil 2.44 \rceil = 3$ pages. Total memory allocated $= 3 \times 4096 = 12{,}288$ bytes. Internal fragmentation $= 12{,}288 - 10{,}000 =$ **2288 bytes**.

Bài tập luyện tập

Explain why reducing the page size (e.g. from 4096 bytes to 512 bytes) would reduce internal fragmentation, and state one disadvantage of using a very small page size.

Lời giải chi tiết

Internal fragmentation is, at most, one page's worth of wasted space per process (the unused remainder of the last, partially-filled page) — a **smaller** page size directly caps this maximum possible waste to a smaller amount, since the largest possible ”leftover” within any single page shrinks along with the page size itself. However, a very small page size means any given process needs *many more* pages to hold the same total amount of data, which increases the size of the **page table** needed to track all those mappings, and increases the overhead of managing many more, smaller page transfers between RAM and disk — a real trade-off between minimising

wasted space and minimising management overhead.

Bài tập luyện tập

Explain why segmentation, unlike paging, cannot suffer internal fragmentation in the same way, yet is more prone to external fragmentation.

Lời giải chi tiết

Segmentation allocates memory in *variable-sized* segments sized to exactly match each logical unit's actual need (e.g. a segment sized precisely for a program's stack or code), so there is no fixed block size that a segment could be smaller than — eliminating internal fragmentation almost entirely (a segment's size is chosen to be no larger than needed). However, because segments have variable sizes, as they are allocated and freed over time, the memory freed by removed segments leaves gaps of inconsistent, unpredictable sizes scattered throughout memory — exactly the **external fragmentation** problem discussed earlier in this chapter, since a new segment may not fit into any single existing gap even if enough total free memory exists across several smaller gaps.

Control --- HL

HL ONLY — Higher Level extension topic (not assessed at SL)

> **Mục tiêu Unit:** Hệ thống nhúng; cảm biến & bộ truyền động; vòng lặp phản hồi (feedback loop) vòng kín/vòng hở; giám sát (monitoring) so với điều khiển (control); hệ thống thời gian thực cứng/mềm; ghi nhật ký dữ liệu & tần số lấy mẫu; các nghiên cứu tình huống điều khiển.

7.1 Embedded and control systems

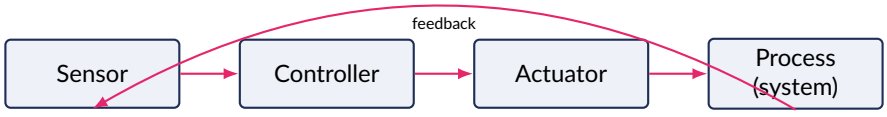
An **embedded system** is a computer built into a larger device to perform one dedicated function, rather than being a general-purpose computer (e.g. a car's engine-management controller, a washing machine's cycle controller, a digital thermostat, a pacemaker). Embedded systems typically run a small, fixed piece of software, often on modest hardware chosen specifically for reliability, low power use, and cost, rather than raw performance.

> **Monitoring** means a system *reads* values from the environment and records/reports them, without changing anything (e.g. a weather station logging temperature every hour). **Control** means the system *uses* readings to actively change the environment (e.g. a thermostat switching a heater on or off in response to temperature). Many real systems do *both* at once — monitoring is a necessary first step, but control additionally requires an action.

> VN **GIẢI THÍCH TIẾNG VIỆT**
>
> **Hệ thống nhúng**: máy tính được tích hợp vào một thiết bị lớn hơn để thực hiện một chức năng chuyên biệt duy nhất (bộ điều khiển động cơ ô tô, máy giặt, máy điều nhiệt, máy tạo nhịp tim), thường dùng phần cứng nhỏ gọn, tiết kiệm năng lượng. **Giám sát (monitoring)**: chỉ ĐỌC và ghi lại dữ liệu, không can thiệp. **Điều khiển (control)**: dùng dữ liệu đọc được để CHỦ ĐỘNG thay đổi môi trường — nhiều hệ thống làm cả hai.

7.2 Sensors, actuators and feedback loops

Sensors convert physical conditions into data the system can read (temperature, light, pressure, motion/PIR, humidity, proximity, sound). **Actuators** let the system act back on the physical world (motors, valves, heaters, relays, speakers, LEDs). A **closed-loop (feedback) system** continuously measures its own output and feeds that measurement back into the controller to adjust behaviour; an **open-loop system** has no feedback path at all — it runs a fixed, pre-programmed action regardless of the actual outcome.



Closed-loop (feedback) control: the sensor continually measures the process output and reports it back to the controller.

Ví dụ mẫu · Worked example

Pseudocode for a thermostat (a classic closed-loop control system) with target temperature 21 °C:

```
Target <- 21
loop
    CurrentTemp <- READSENSOR()
    if CurrentTemp < Target then
        HEATER <- "ON"
    else
        HEATER <- "OFF"
    end if
end loop
```

The loop repeats indefinitely, continually reading the sensor (feedback) and adjusting the actuator (heater) — this is monitoring *and* control combined, and its closed-loop nature means the system automatically self-corrects if the room temperature drifts for any reason (an open window, more people entering the room, etc).

Ví dụ mẫu · Worked example

A garden sprinkler runs for exactly 10 minutes every day at 6 a.m., regardless of recent rainfall or current soil moisture. Classify this as open-loop or closed-loop, and redesign it as the other type.

This is **open-loop**: the controller runs a fixed action (10 minutes) with no sensor feeding back the actual outcome (soil moisture), so it wastes water after rain and may under-water during a heatwave. A **closed-loop** redesign would add a **soil-moisture sensor**: the controller reads current soil moisture and runs the sprinkler only until a target moisture level is reached (or skips watering entirely if already moist), continuously adapting to real conditions instead of blindly following a fixed schedule.

VN **GIẢI THÍCH TIẾNG VIỆT**

Cảm biến (sensor): chuyển đại lượng vật lý (nhiệt độ, ánh sáng, áp suất, chuyển động) thành dữ liệu số. **Bộ truyền động (actuator)**: tác động ngược lại thế giới vật lý (động cơ, van, rơ-le, loa). **Vòng kín (closed-loop/feedback)**: liên tục đo đầu ra và phản hồi để điều chỉnh — tự sửa sai, như máy điều nhiệt. **Vòng hở (open-loop)**: không có phản hồi, chạy một hành động cố định bất kể kết quả thực tế — ví dụ hẹn giờ tưới cây cố định bất kể độ ẩm đất.

(!) Lỗi thường gặp: A system can **measure** something and still be open-loop if that measurement is never fed back to influence the controller's future actions. The defining test for closed-loop is not "does it have a sensor?" but "does the sensor's *output* change the controller's *future behaviour*?"

7.3 Real-time systems and data logging

A **real-time system** must respond to input within a guaranteed time limit, not merely "quickly."

Hard real-time: missing a deadline constitutes a system **failure**, with potentially severe consequences (e.g. an airbag controller that deploys even a fraction of a second late, an aircraft flight-control system, industrial safety shutoffs). **Soft real-time**: occasional missed deadlines degrade quality but do not cause catastrophic failure (e.g. video streaming that occasionally buffers, a live scoreboard update arriving a moment late).

VN **GIẢI THÍCH TIẾNG VIỆT**

Hệ thống thời gian thực: phải phản hồi trong giới hạn thời gian đảm bảo, không chỉ "nhanh". **Thời gian thực cứng (hard)**: trễ hạn = hệ thống thất bại, hậu quả nghiêm trọng (túi khí ô tô, hệ thống điều khiển bay). **Thời gian thực mềm (soft)**: trễ hạn chỉ làm giảm chất lượng, không gây thảm hoạ (xem video trực tuyến bị giật một chút).

Data logging automatically records sensor readings at a chosen **sampling rate** (readings per unit time). A higher sampling rate captures more detail/accuracy (better able to catch short bursts of change) but consumes more storage and power; a lower sampling rate saves resources but may miss important short-lived events entirely.

Ví dụ mẫu · Worked example

A weather station logs temperature every 5 seconds, continuously, for 1 hour. How many readings are recorded?

1 hour $= 3600$ seconds. Number of readings $= \dfrac{3600}{5} = 720$ readings.

Ví dụ mẫu · Worked example

A soil-moisture data logger samples every 15 minutes for 24 hours. How many readings does it store, and what happens if the sampling interval is instead reduced to every 1 minute?

At 15-minute intervals: $\dfrac{24 \times 60}{15} = 96$ readings per day. At 1-minute intervals: $24 \times 60 = 1440$ readings per day — roughly **15 times** more data, giving far finer detail (better chance of catching short bursts of change) at the cost of much greater storage use and battery/power drain.

VN **GIẢI THÍCH TIẾNG VIỆT**

Ghi nhật ký dữ liệu (data logging): tự động ghi lại số đọc cảm biến theo một **tần số lấy mẫu (sampling rate)**. Tần số cao hơn → chi tiết/chính xác hơn nhưng tốn bộ nhớ và năng lượng hơn; tần số thấp hơn → tiết kiệm nhưng có thể bỏ lỡ sự kiện ngắn hạn quan trọng.

7.4 Case studies in control systems

Traffic light system: sensors (inductive loops or cameras) detect vehicle presence at each approach; the controller applies fixed or adaptive timing rules to decide which direction gets a green light, and for how long — a **closed-loop** design (using real-time vehicle counts) generally reduces congestion better than a fixed-timer (**open-loop**) design that ignores actual traffic. **Washing machine:** sensors detect water level, drum temperature, and load balance; the controller sequences the wash/rinse/spin cycle, actuating the water valve, heater and motor — largely closed-loop for water level and temperature, but the overall cycle length is often simply a pre-set open-loop timer once a program is chosen. **Autonomous greenhouse:** soil-moisture, light, and temperature sensors feed a controller that actuates irrigation valves, grow lights, and roof vents — a fully closed-loop system that continuously adapts to real plant conditions rather than following a fixed schedule.

VN GIẢI THÍCH TIẾNG VIỆT

Đèn giao thông: cảm biến phát hiện xe, bộ điều khiển quyết định thời gian đèn xanh — thiết kế vòng kín (dùng số liệu xe thực tế) giảm ùn tắc tốt hơn vòng hở (hẹn giờ cố định). **Máy giặt**: cảm biến mực nước/nhiệt độ/cân bằng tải, vòng kín cho nước và nhiệt độ nhưng tổng thời gian chu trình thường là hẹn giờ vòng hở. **Nhà kính tự động**: cảm biến độ ẩm đất/ánh sáng/nhiệt độ điều khiển van tưới, đèn, cửa thông gió — hoàn toàn vòng kín.

Ví dụ mẫu · Worked example

A city upgrades its traffic lights from a fixed 60-second cycle at every intersection to a system using real-time vehicle-count sensors that adjust green-light duration dynamically. Explain, using the terms ”open-loop” and ”closed-loop,” why this upgrade should reduce congestion.

The original fixed 60-second cycle is **open-loop**: it allocates the same green-light time to every direction regardless of how many vehicles are actually waiting, wasting green time on a nearly-empty road while a busy road remains stuck on red. The upgraded system is **closed-loop**: real-time vehicle-count sensors continuously feed back the actual traffic level to the controller, which can allocate *more* green time to the direction with more waiting vehicles and less time to an empty approach — automatically adapting to real, changing traffic conditions rather than a rigid, one-size-fits-all schedule.

VN GIẢI THÍCH TIẾNG VIỆT

Ôn tập nhanh Unit 7: (1) Hệ thống nhúng = máy tính chuyên biệt trong thiết bị lớn hơn; (2) giám sát chỉ ĐỌC, điều khiển thì HÀNH ĐỘNG; (3) vòng kín có phản hồi (tự sửa sai), vòng hở không có phản hồi (hành động cố định); (4) thời gian thực cứng: trễ = thất bại; thời gian thực mềm: trễ chỉ giảm chất lượng; (5) tần số lấy mẫu cao hơn = chi tiết hơn nhưng tốn tài nguyên hơn; (6) luôn kiểm tra xem cảm biến có thực sự ẢNH HƯỞNG hành vi tương lai của bộ điều khiển hay không để phân biệt vòng kín/vòng hở.

7.5 Proportional control

The thermostat example earlier in this chapter uses **on/off (bang-bang) control**: the actuator is either fully ON or fully OFF depending on a single threshold. This is simple but can cause the controlled value to repeatedly overshoot and undershoot the target (oscillation), since the actuator gives no information about *how far* the current value is from the target.

Proportional control instead adjusts the actuator's output in proportion to the **error** (the difference between the target value and the current measured value): a large error produces a large corrective action, while a small error produces a small, gentler correction as the system approaches the target. This generally reaches and holds the target more smoothly than simple on/off control, reducing oscillation — though pure proportional control can still leave a small permanent gap between the target and actual value (a **steady-state error**), which more advanced control (e.g. full PID control, beyond this course) can further reduce.

Ví dụ mẫu · Worked example

A robot's motor speed is controlled proportionally to reduce distance-to-target error: MotorSpeed $=$ $K \times$ Error, where Error $=$ TargetDistance $-$ CurrentDistance and $K = 2$. Calculate the motor speed when the error is 10 units, and again when the error has shrunk to 2 units, and explain the resulting behaviour.

At Error $= 10$: MotorSpeed $= 2 \times 10 = 20$ (fast, since the robot is far from its target). At Error $= 2$: MotorSpeed $= 2 \times 2 = 4$ (much slower, since the robot is nearly at its target). The motor automatically slows down as the robot approaches its destination, rather than moving at full speed until suddenly stopping — this smoother approach reduces the risk of overshooting the target compared with simple on/off control, which would run the motor at full speed right up until the exact target is reached.

VN **GIẢI THÍCH TIẾNG VIỆT**

Điều khiển bật/tắt (on/off): bộ truyền động chỉ BẬT hoặc TẮT hoàn toàn — đơn giản nhưng dễ gây dao động (vọt lố rồi lại thiếu). **Điều khiển tỷ lệ (proportional control)**: điều chỉnh đầu ra TỶ LỆ với sai số (Error = mục tiêu − giá trị hiện tại) — sai số lớn thì điều chỉnh mạnh, sai số nhỏ thì điều chỉnh nhẹ, giúp tiệm cận mục tiêu êm hơn, ít dao động hơn so với bật/tắt đơn giản.

7.6 IoT and edge computing in control systems

The **Internet of Things (IoT)** refers to everyday physical devices (thermostats, door locks, fitness trackers, industrial sensors) embedded with sensors/actuators and network connectivity, allowing them to be monitored and controlled remotely and to share data with other systems over the internet.

Cloud-based control sends sensor data over the internet to a remote server for processing, which then sends control instructions back — flexible and powerful (can use large-scale computation, combine data from many devices), but adds **latency** (round trip to a distant server) and depends entirely on network connectivity being available and fast enough. **Edge computing** instead processes data *locally*, close to (or on) the device itself, before only sending a summary or exceptions to the cloud — reducing latency and bandwidth use, and allowing the system to keep functioning (at least for critical, time-sensitive decisions) even if the internet connection is temporarily lost.

Ví dụ mẫu · Worked example

A factory's safety system must stop a machine within 50 milliseconds of detecting a hazard, but the factory's internet connection is occasionally unreliable. Explain why an edge-computing design is more appropriate than a cloud-based design for this specific safety function.

A hard real-time safety deadline of 50 milliseconds could easily be missed if the decision de-

pended on a round trip to a distant cloud server — network delay or an unreliable/unavailable connection could cause the deadline to be missed entirely, an unacceptable risk for a safety system. An **edge computing** design processes the hazard-detection logic locally, directly at (or very near) the machine, so the stop decision does not depend on external network connectivity at all — guaranteeing the response time is met even during a temporary internet outage, while still potentially sending summary data to the cloud afterwards for logging/analysis when connectivity is available.

VN GIẢI THÍCH TIẾNG VIỆT

IoT (Internet vạn vật): thiết bị vật lý hàng ngày gắn cảm biến/bộ truyền động và kết nối mạng, cho phép giám sát/điều khiển từ xa. **Điều khiển qua đám mây**: gửi dữ liệu lên server từ xa xử lý rồi gửi lệnh về — mạnh mẽ nhưng có độ trễ và phụ thuộc kết nối mạng. **Điện toán biên (edge computing)**: xử lý dữ liệu NGAY TẠI thiết bị — giảm độ trễ, vẫn hoạt động được (với quyết định quan trọng, khẩn cấp) ngay cả khi mất kết nối Internet tạm thời.

7.7 Actuator types

Different actuators suit different control tasks, depending on whether continuous, precisely-positioned, or simple on/off movement is required.

DC motor — rotates continuously at a speed roughly proportional to applied voltage; simple and cheap, but does not inherently know or control its own exact position/angle without an additional sensor (e.g. an encoder) feeding back rotation data. **Servo motor** — a motor combined with built-in position feedback and control circuitry, allowing it to rotate to (and hold) a specific, precise angle on command — ideal where exact positioning matters (e.g. a robotic arm joint, an automatic camera gimbal). **Stepper motor** — rotates in fixed, discrete increments ("steps") per electrical pulse received, allowing precise, repeatable positioning by simply counting pulses sent, without needing a separate position sensor — common in 3D printers and precision instruments. **Relay/solenoid** — a simple electrically-controlled switch or linear push/pull actuator, suited to basic on/off actions (e.g. switching a heater circuit on or off, or an electronic door lock).

Ví dụ mẫu · Worked example

A robotics student needs a robot arm's elbow joint to move to and hold a precise, specific angle. Which actuator type is most suitable, and why would a plain DC motor alone be a poor choice?

A **servo motor** is most suitable, since it is specifically designed with built-in feedback to move to, and hold, a precise commanded angle. A plain **DC motor** alone only controls rotational *speed* (roughly proportional to voltage) and continues rotating as long as voltage is applied — it has no inherent awareness of its current angular position, so without extra external sensors and control logic, it cannot reliably be commanded to "stop at exactly 47 degrees and hold that position," which is exactly what a servo motor is built to do.

VN GIẢI THÍCH TIẾNG VIỆT

Động cơ DC: quay liên tục, tốc độ tỉ lệ với điện áp — đơn giản, rẻ, nhưng không tự biết vị trí góc quay chính xác. **Động cơ servo**: có phản hồi vị trí tích hợp, quay tới và giữ đúng một góc cụ thể — dùng cho cánh tay robot, gimbal camera. **Động cơ bước (stepper motor)**: quay theo

từng bước rời rạc cố định mỗi xung điện — định vị chính xác mà không cần cảm biến riêng, dùng trong máy in 3D. **Relay/solenoid**: công tắc điều khiển điện đơn giản, phù hợp hành động BẬT/TẮT cơ bản.

7.8 Fail-safe and fail-secure design

Every control system must be designed with a deliberate answer to "what happens when something goes wrong?" — power loss, sensor failure, or a software crash.

A **fail-safe** design defaults to the *safest possible state* for people/property when a failure occurs, even if that state is inconvenient (e.g. a level-crossing barrier that automatically **lowers** if it loses power or its control signal, since a barrier stuck open during a power failure could allow a car onto the tracks in front of an oncoming train). A **fail-secure** (or fail-locked) design instead defaults to the state that best protects *security*, even at the cost of convenience or safety in some contexts (e.g. an electronic door lock that stays **locked** if it loses power, prioritising preventing unauthorised entry over allowing people to exit freely). The correct choice depends entirely on which risk — physical safety or security breach — is more critical for that specific system.

Ví dụ mẫu · Worked example

A fire-exit door uses an electronic lock. Explain why this door should almost certainly be designed **fail-safe** (unlocking on power loss) rather than fail-secure (locking on power loss), even though fail-secure might seem more "secure."

If the building loses power during a fire (a very plausible scenario, since fires often damage electrical systems), a **fail-secure** design would lock the fire exit shut exactly when people most urgently need to escape — trading a security benefit for an unacceptable, potentially fatal safety risk. A **fail-safe** design instead automatically unlocks the door if power is lost, prioritising occupants' ability to escape during an emergency over the comparatively minor risk of the door being briefly unsecured during a power outage — for a life-safety exit, physical safety must take priority over security.

VN **GIẢI THÍCH TIẾNG VIỆT**

Thiết kế an toàn khi lỗi (fail-safe): mặc định về trạng thái AN TOÀN NHẤT cho con người khi có sự cố (ví dụ: barie đường sắt tự động HẠ XUỐNG khi mất điện). **Thiết kế bảo mật khi lỗi (fail-secure)**: mặc định về trạng thái BẢO MẬT NHẤT khi có sự cố (ví dụ: khoá cửa điện tử tự động KHOÁ khi mất điện). Lựa chọn đúng tuỳ vào rủi ro nào (an toàn tính mạng hay an ninh) nghiêm trọng hơn với hệ thống cụ thể đó.

7.9 Case study: sensor fusion in autonomous vehicles

An autonomous vehicle's control system must build a reliable picture of its surroundings by combining data from *multiple different types* of sensors, since no single sensor type is reliable in every condition — a technique called **sensor fusion**.

Cameras provide rich visual detail (reading road signs, lane markings, traffic-light colour) but perform poorly in darkness, fog, or direct glare. **Radar** reliably measures distance and speed of other objects even in poor weather/visibility, but provides much coarser detail about an object's exact shape. **LiDAR** (laser-based distance sensing) builds a very precise 3D map of surrounding shapes and distances, but is more expensive and can be degraded by heavy rain/snow. Combining all three (**sensor fusion**) lets the control system cross-check readings against each other, so a temporary weakness in one sensor type (e.g. a camera blinded by sun glare) can be compensated for by the others (e.g. radar still reliably detecting a vehicle ahead) — a direct application of **robustness** (Unit 1) in a safety-critical control system.

Ví dụ mẫu · Worked example

An autonomous car's camera is temporarily unable to see a pedestrian clearly due to heavy glare from low sun. Explain how sensor fusion allows the vehicle's control system to still detect and react to the pedestrian safely.

Even though the **camera**'s visual detection is degraded by glare, the vehicle's **radar** and/or **LiDAR** sensors are not affected by visible-light glare in the same way, and can still detect the pedestrian's presence, distance, and movement using entirely different physical principles (reflected radio waves or laser pulses rather than visible light). The control system's **sensor fusion** logic combines readings from all sensor types, weighting or cross-checking them against each other, so a temporary weakness in one specific sensor does not necessarily mean the overall system loses awareness of the hazard — this redundancy across genuinely different sensor technologies is precisely what makes the overall system more **robust** than relying on any single sensor type alone.

VN **GIẢI THÍCH TIẾNG VIỆT**

Hợp nhất cảm biến (sensor fusion): kết hợp dữ liệu từ NHIỀU LOẠI cảm biến khác nhau (camera, radar, LiDAR) vì không loại cảm biến nào đáng tin cậy trong MỌI điều kiện. Camera cho chi tiết hình ảnh nhưng kém trong bóng tối/chói sáng; radar đo khoảng cách/tốc độ tốt trong thời tiết xấu nhưng kém chi tiết; LiDAR cho bản đồ 3D chính xác nhưng đắt và kém trong mưa to. Kết hợp cả ba giúp hệ thống vẫn hoạt động đúng dù một loại cảm biến tạm thời bị suy giảm — ứng dụng trực tiếp của tính **robustness** trong hệ thống điều khiển an toàn.

7.10 Analogue-to-digital conversion

Physical quantities (temperature, sound, light) vary **continuously** (analogue), but a digital computer can only store discrete numeric values. An **Analogue-to-Digital Converter (ADC)** bridges this gap, and the choices made in this conversion directly determine how faithfully the digital data represents the original physical signal.

An ADC's **sampling rate** (Unit 7, earlier) determines how often the continuous signal is measured; its **resolution** (in bits) determines how many discrete levels each individual measurement can be rounded to. A signal that changes rapidly requires a *higher* sampling rate to avoid missing important short-lived variations (undersampling can make rapid changes appear as a false, much slower pattern — a distortion effect called **aliasing**). Higher resolution reduces **quantisation error** (the small rounding error introduced whenever a continuous value is forced into the nearest available discrete level), at the cost of more bits (storage) per sample.

Ví dụ mẫu · Worked example

A temperature sensor's ADC has 8-bit resolution over a measurement range of 0 to 51.2°C. Calculate the smallest temperature change (quantisation step) the ADC can distinguish.

An 8-bit ADC has $2^8 = 256$ discrete levels across the full range. Smallest distinguishable step $= \frac{51.2°\text{C}}{256} = 0.2°\text{C}$. Any actual temperature difference smaller than 0.2°C cannot be distinguished by this ADC — two slightly different real temperatures within the same 0.2°C "bucket" would be recorded as numerically identical digital readings.

VN **GIẢI THÍCH TIẾNG VIỆT**

Chuyển đổi tương tự sang số (ADC): chuyển tín hiệu vật lý LIÊN TỤC thành giá trị số RỜI RẠC. **Tần số lấy mẫu** thấp quá có thể gây **răng cưa (aliasing)** — tín hiệu biến đổi nhanh bị hiểu sai thành biến đổi chậm giả tạo. **Độ phân giải (resolution)** cao hơn giảm **sai số lượng tử hoá (quantisation error)** nhưng tốn nhiều bit lưu trữ hơn mỗi mẫu.

7.11 Case study: smart home automation

A smart home system integrates many individual sensors and actuators under one coordinating controller, illustrating how the concepts of this chapter combine in a single real system.

Ví dụ mẫu · Worked example

A smart home system: a motion sensor and a light sensor jointly control hallway lighting (turn lights on only if motion is detected *and* ambient light is below a threshold); a smart thermostat maintains temperature using feedback from a temperature sensor; a water-leak sensor triggers an emergency shutoff valve and an immediate phone alert. Identify the type of control (open/closed loop) for each subsystem, and classify the water-leak subsystem's real-time requirement.

The **hallway lighting** subsystem is **closed-loop** with respect to light level (continuously sensing ambient light to decide whether to act) but note the "motion AND darkness" rule is itself just a combined condition, not full continuous feedback on the lighting's own effect. The **thermostat** is a classic **closed-loop** system (continuously measuring and correcting temperature, exactly as in this chapter's earlier thermostat example). The **water-leak shutoff** is **hard real-time**: a delayed response to a detected leak could allow significant, costly water damage to accumulate, so the valve must close and the alert must fire within a strictly bounded time of detection, every time, without exception.

VN **GIẢI THÍCH TIẾNG VIỆT**

Nhà thông minh kết hợp nhiều cảm biến/bộ truyền động dưới một bộ điều khiển trung tâm: đèn hành lang (cảm biến chuyển động + ánh sáng), máy điều nhiệt (vòng kín kinh điển), cảm biến rò rỉ nước (thời gian thực cứng — vì phản hồi trễ có thể gây thiệt hại lớn). Đây là ví dụ thực tế tổng hợp các khái niệm: cảm biến/bộ truyền động, vòng kín/vòng hở, và phân loại thời gian thực.

7.12 Redundancy in control systems

Safety-critical control systems often cannot rely on a single sensor or actuator, since any single point of failure could be catastrophic — **redundancy** deliberately duplicates critical components.

Redundancy means including backup/duplicate components so the system continues operating correctly even if one component fails. Common patterns: **dual redundancy** (two identical components; if one fails, the other takes over — but if the two disagree, the system cannot tell which is faulty); **triple modular redundancy (TMR)** (three identical components; the system uses a **majority vote** — if one disagrees with the other two, the majority is trusted and the outlier is flagged as faulty, allowing the system to continue operating correctly *and* identify which unit failed). Aircraft flight-control computers commonly use triple (or greater) redundancy specifically because a single computer error, undetected, could be catastrophic.

Ví dụ mẫu · Worked example

An aircraft uses triple modular redundancy for its critical altitude sensor: three independent sensors report readings of $10{,}050$ ft, $10{,}048$ ft, and $9{,}200$ ft. Explain how the system determines the correct altitude and identifies the faulty sensor.

The system compares all three readings: two of them ($10{,}050$ and $10{,}048$ ft) closely agree with each other, while the third ($9{,}200$ ft) differs drastically from both. Using a **majority vote**, the system trusts the two closely-agreeing readings as correct (taking, e.g., their average or one of them as the working altitude value) and flags the third sensor as **faulty** — the aircraft's control system continues operating safely using the trusted majority reading, while maintenance can later investigate and repair/replace the identified faulty sensor, all without ever exposing the flight to an unverified, potentially dangerous altitude reading.

VN **GIẢI THÍCH TIẾNG VIỆT**

Dự phòng (redundancy): bao gồm thành phần dự phòng/nhân bản để hệ thống vẫn hoạt động đúng dù một thành phần hỏng. **Dự phòng kép**: hai thành phần giống nhau, nhưng không biết cái nào sai nếu chúng bất đồng. **Dự phòng ba (TMR)**: ba thành phần, dùng **biểu quyết đa số** — nếu một cái khác biệt với hai cái kia, đa số được tin dùng và thành phần lệch bị đánh dấu lỗi. Máy bay dùng TMR cho máy tính điều khiển bay vì một lỗi đơn lẻ không bị phát hiện có thể gây thảm hoạ.

7.13 Case study: automated traffic light control

Ví dụ mẫu · Worked example

Design simplified pseudocode for a traffic-light controller at a junction with vehicle-presence sensors on each of two roads (Road A, main road; Road B, side road), giving Road A priority but switching to Road B if a vehicle waits there for too long.

```
RoadALight <- "GREEN"
RoadBLight <- "RED"
WaitTimeB <- 0

loop
    VehicleOnB <- READSENSOR(RoadBSensor)
    if VehicleOnB = TRUE then
        WaitTimeB <- WaitTimeB + 1
    else
        WaitTimeB <- 0
    end if
```

```
    if WaitTimeB > 30 then
        RoadALight <- "RED"
        RoadBLight <- "GREEN"
        WaitTimeB <- 0
    end if
end loop
```

This is a **closed-loop** control system: the `RoadBSensor` reading continuously feeds back into the controller's decision, dynamically adjusting the actuator (the traffic lights) rather than following a rigid, fixed timer regardless of actual conditions — directly combining this chapter's sensor/actuator/feedback-loop concepts into one realistic worked system.

VN **GIẢI THÍCH TIẾNG VIỆT**

Ví dụ điều khiển đèn giao thông: cảm biến phát hiện xe ở đường phụ (Road B), nếu xe chờ quá lâu (biến đếm `WaitTimeB` vượt ngưỡng) thì hệ thống chuyển đèn xanh sang đường phụ — đây là hệ thống **vòng kín** vì tín hiệu cảm biến liên tục phản hồi vào quyết định của bộ điều khiển, thay vì dùng hẹn giờ cố định bất kể tình huống thực tế.

7.14 Practice

Bài tập luyện tập

A rain gauge simply records rainfall amounts to a log file every hour, with no other device reacting to the readings. Is this monitoring, control, or both?

(A) Monitoring only
(B) Control only
(C) Both monitoring and control
(D) Neither

Lời giải chi tiết

The system only **reads and records** data — it does not use the readings to change anything in the environment. This is **monitoring only**. **(A)**.

Bài tập luyện tập

Explain, using the terms *sensor*, *controller*, and *actuator*, how a closed-loop system differs from an open-loop system.

Lời giải chi tiết

Both systems use a **controller** to decide what an **actuator** should do. The key difference is feedback: in a **closed-loop** system, a **sensor** continually measures the actual output of the process and reports it back to the controller, which adjusts its actions accordingly (self-correcting). In an **open-loop** system there is no sensor feeding results back — the controller runs a fixed action (e.g. a timer) regardless of the actual outcome, so it cannot correct for errors or changing conditions.

Bài tập luyện tập

A patient monitor in an ICU must trigger an alarm within 2 seconds of detecting an irregular heartbeat, every time, without exception. What type of system is this?

(A) Soft real-time
(B) Hard real-time
(C) Batch processing
(D) Open-loop only

Lời giải chi tiết

Missing the deadline here could be fatal, so *every* deadline must be met without exception — this is the defining feature of a **hard real-time** system. **(B)**.

Bài tập luyện tập

A soil-moisture data logger samples every 15 minutes for 24 hours. How many readings does it store, and give one advantage and one disadvantage of increasing the sampling rate to every 1 minute.

Lời giải chi tiết

At 15-minute intervals: $\frac{24 \times 60}{15} = 96$ readings per day. Increasing the rate to every 1 minute gives $24 \times 60 = 1440$ readings/day — an **advantage** is much finer detail (better chance of catching short bursts of change, more accurate trend analysis); a **disadvantage** is far greater storage use and power/battery drain (roughly $15\times$ more data to store and transmit).

Bài tập luyện tập

A video-streaming service occasionally has to briefly lower video resolution when network conditions worsen, rather than freezing completely. Classify this requirement as hard or soft real-time, and justify.

Lời giải chi tiết

This is **soft real-time**: an occasional missed deadline (insufficient data arriving in time for the next frame at full quality) results only in a temporary, tolerable *degradation* of quality (lower resolution), not a catastrophic system failure. A hard real-time failure would instead be unacceptable regardless of how "graceful" the degradation — soft real-time systems are specifically designed to tolerate this kind of occasional shortfall.

Bài tập luyện tập

A washing machine's water-level sensor stops the water valve exactly when the drum reaches the correct level for the selected wash size, then proceeds through a wash/rinse/spin sequence of pre-set, fixed durations. Identify which part of this system is closed-loop and which part is open-loop.

Lời giải chi tiết

The **water-level control** is **closed-loop**: the sensor continuously measures the actual water level and feeds this back to stop the valve at exactly the right point, adapting to whatever the real water level currently is. The **wash/rinse/spin sequence timing** is **open-loop**: once a

wash programme is selected, each stage runs for a fixed, pre-programmed duration regardless of any actual measured outcome (e.g. it does not check whether clothes are actually clean before ending the wash stage).

Bài tập luyện tập

Explain why an embedded system in a pacemaker is designed with hard real-time constraints, and describe the likely consequence of even a very rare missed deadline.

Lời giải chi tiết

A pacemaker's function is to deliver an electrical pulse to regulate the heartbeat at precisely the moment the heart's natural rhythm requires it — a delayed or missed pulse can directly threaten the patient's life within seconds. Because the cost of a missed deadline is catastrophic and irreversible, the system is designed with **hard real-time** guarantees: even a single missed or excessively late deadline, however rare, is treated as a complete system failure, not merely a quality reduction, because in this context there is no such thing as an acceptable "slightly late" response.

Bài tập luyện tập

A smart irrigation controller only ever waters a garden for a fixed 20 minutes at 7 a.m. every day, based on a schedule set once by the installer, never adjusted afterwards. A neighbour's system instead uses live soil-moisture readings to decide both whether and how long to water each day. Compare the two systems' likely water efficiency and explain why.

Lời giải chi tiết

The fixed-schedule system is **open-loop**: it waters for the same 20 minutes regardless of whether it rained yesterday, how hot the weather is, or how moist the soil already is — likely wasting water after rain and possibly under-watering during a heatwave. The neighbour's **closed-loop** system continuously measures real soil moisture and adapts both the decision to water and the duration to actual current conditions, watering only when and as much as genuinely needed. The closed-loop system is therefore very likely **more water-efficient**, since it responds to the real, changing state of the garden rather than following a rigid, one-size-fits-all schedule.

Bài tập luyện tập

A factory's automated safety system must cut power to a machine within 50 milliseconds of detecting an operator's hand entering a dangerous zone. State the type of real-time system this requires, and explain what would make the associated sensor and actuator "monitoring *and* control," not monitoring alone.

Lời giải chi tiết

This requires a **hard real-time** system: a missed or late deadline could result in serious injury, an unacceptable outcome regardless of how rarely it might occur. The sensor detecting the hand is only **monitoring** on its own; the system becomes **monitoring *and* control** because the reading is actively used to trigger an **actuator** (the power cut-off mechanism) that changes the physical state of the machine — the defining feature of control is this active response, not

merely detecting the hand's presence.

Bài tập luyện tập

Explain why a data logger recording earthquake vibrations would need a much higher sampling rate than one recording daily outdoor temperature, referencing how quickly each phenomenon changes.

Lời giải chi tiết

The **sampling rate** must be fast enough to capture the phenomenon's fastest meaningful changes — a general principle is that if a signal changes significantly within a certain time window, samples must be taken more often than that window to avoid missing (or badly misrepresenting) the change. Earthquake vibrations oscillate extremely rapidly (many times per second), so a logger must sample hundreds or thousands of times per second to accurately capture the waveform. Outdoor temperature changes gradually over minutes to hours, so sampling once every few minutes is more than sufficient to capture its trend accurately — sampling it as fast as an earthquake sensor would be unnecessary and would waste storage with no benefit, since nothing meaningful changes between such closely-spaced samples.

Bài tập luyện tập

A greenhouse controller reads a light sensor every second but the reading is only ever displayed on a screen for staff to view manually; nothing in the greenhouse automatically responds to the reading. Later, the greenhouse is upgraded so that the same reading automatically opens or closes a shading screen. Explain the classification change.

Lời giải chi tiết

In the original version, the system only **monitors**: light level is read and displayed, but nothing in the system itself changes as a direct, automatic result of the reading (staff might act on it, but that is a human decision, not the system controlling anything). After the upgrade, the same reading is fed back into a controller that actuates the shading screen automatically — the system now performs **monitoring and control**, and specifically becomes a **closed-loop** system, since the sensor's output now directly and automatically influences the actuator's future behaviour without requiring a human in the loop.

Bài tập luyện tập

A proportional heating controller uses HeaterPower $= K \times$ Error with $K = 5$, where Error $=$ TargetTemp $-$ CurrentTemp. Calculate the heater power when the room is 8°C below target, and when it is only 1°C below target, and explain the behavioural difference from simple on/off control.

Lời giải chi tiết

At Error $= 8$: HeaterPower $= 5 \times 8 = 40$ (units of power) — a strong response since the room is far from target. At Error $= 1$: HeaterPower $= 5 \times 1 = 5$ — a much gentler response as the room nears target temperature. Unlike simple **on/off** control (which would run the heater at full power right up until the exact target temperature is reached, then suddenly switch off — likely overshooting due to residual heat), **proportional control** tapers the heater's output down

smoothly as the room approaches the target, reducing the risk of overshoot and the resulting oscillation around the target temperature.

Bài tập luyện tập

Explain one disadvantage of simple on/off (bang-bang) control compared with proportional control, using the concept of "oscillation."

Lời giải chi tiết

On/off control only ever fully switches the actuator ON or OFF based on a single threshold, with no information about *how close* the system is to the target. As a result, the controlled value often **overshoots** past the target before the "OFF" signal takes effect (e.g. residual heat continues warming a room briefly after a heater switches off), then drifts back down until it triggers "ON" again, overshooting again in the other direction — this repeated overshoot/undershoot cycle is called **oscillation**. Proportional control reduces this problem by tapering the actuator's output down as the error shrinks, approaching the target more smoothly rather than swinging fully on and fully off around it.

Bài tập luyện tập

A wearable fitness tracker analyses raw heart-rate sensor data directly on the device to detect an irregular heartbeat pattern, only sending a summary alert to a cloud server if something concerning is found. Identify whether this is an edge-computing or cloud-computing design, and give one advantage of this choice for this specific use case.

Lời giải chi tiết

This is an **edge computing** design: the time-critical analysis (detecting an irregular heartbeat) happens locally on the device itself, rather than requiring every raw sensor reading to be sent to a distant cloud server for processing. One advantage for this use case: the device can detect and respond to a potentially urgent health event with minimal delay and *without depending on an internet connection being available at that moment* — critical for a safety-relevant, time-sensitive alert, whereas a cloud-only design could be delayed or fail entirely if connectivity were temporarily lost exactly when needed most.

Bài tập luyện tập

Explain one disadvantage of edge computing compared with cloud computing for a control system, using the idea of available processing power.

Lời giải chi tiết

Edge devices are typically small, low-power embedded systems with limited processing power, memory, and storage compared with large-scale cloud data centres. This means edge computing is well suited to relatively simple, fast, time-critical local decisions, but is poorly suited to computationally intensive analysis (e.g. training a large machine-learning model, or combining and analysing data from thousands of devices at once) — such heavy processing is generally better performed in the **cloud**, which can allocate far greater computational resources than any single small embedded device could provide locally.

Bài tập luyện tập

A 3D printer must move its print head to precise, repeatable X-Y positions many times per print, without needing a separate position sensor. Which actuator type is most appropriate, and why?

(A) DC motor, because it is the cheapest option

(B) Servo motor, because it moves continuously

(C) Stepper motor, because it moves in precise, countable increments per pulse

(D) Relay, because it only needs two states

Lời giải chi tiết

A **stepper motor** moves in fixed, discrete steps for each electrical pulse it receives, allowing the controller to achieve precise, repeatable positioning simply by counting how many pulses have been sent — without requiring a separate feedback sensor, unlike a servo motor's built-in feedback loop. **(C)**.

Bài tập luyện tập

Explain why a simple relay would be an inappropriate actuator choice for smoothly and precisely controlling a robot arm's joint angle.

Lời giải chi tiết

A **relay** is fundamentally a simple electrically-controlled switch, capable only of an on/off (or open/closed) state — it provides no means of controlling a continuous range of positions or angles, only a binary state change. Precisely controlling a robot arm's joint angle requires an actuator capable of moving to, and holding, a wide range of specific intermediate positions (such as a servo or stepper motor) — a task a simple two-state relay is structurally incapable of performing.

Bài tập luyện tập

A bank vault's electronic lock is designed to remain **locked** if it loses power. Explain why fail-secure, rather than fail-safe, is the appropriate design choice here.

Lời giải chi tiết

For a bank vault, the primary risk being managed is **unauthorised access to valuable/sensitive contents**, not the physical safety of people trapped inside during normal operating conditions (a vault is not a life-safety fire exit that people need to escape through in an emergency). A **fail-secure** design — remaining locked during a power failure — correctly prioritises preventing a security breach during a vulnerable moment (a power outage that might otherwise be exploited by an intruder), which is the dominant risk for this specific type of system, unlike a fire-exit door where the safety risk of trapping occupants dominates.

Bài tập luyện tập

Explain why designing every safety-critical door in a building identically fail-secure would be inappropriate, using the contrast between a fire exit and a server-room door.

Lời giải chi tiết

A **fire exit** must prioritise occupant safety above all else — trapping people inside during a power failure (which a fail-secure design would risk) could be fatal, so it should be **fail-safe** (unlocking on power loss). A **server-room door**, by contrast, primarily protects valuable equipment and sensitive data from unauthorised access, with no urgent need for people to be inside during a power failure — here **fail-secure** (remaining locked on power loss) correctly prioritises security. Applying one single design choice (fail-safe or fail-secure) uniformly to every door in a building, without considering each door's specific risk profile, would create an unacceptable outcome for at least one of these two very different use cases.

Bài tập luyện tập

Explain why an autonomous vehicle relying on *only* a camera sensor (no radar or LiDAR) would be considered less robust than one using sensor fusion, referencing a specific weather condition.

Lời giải chi tiết

A camera-only system depends entirely on visible light and clear optical conditions to function correctly; in **heavy fog** (or similarly, heavy rain, snow, or darkness), a camera's ability to detect and classify obstacles is severely degraded or may fail altogether, since visibility itself is physically reduced regardless of camera quality. A system using **sensor fusion** additionally relies on **radar**, which uses radio waves largely unaffected by fog/rain/darkness and can still reliably detect the distance and speed of obstacles even when the camera's view is severely compromised — providing a level of continued safe operation that a camera-only system structurally cannot guarantee under the same conditions.

Bài tập luyện tập

Explain why sensor fusion, rather than simply installing the single "best" sensor type available, is the standard approach in autonomous vehicle design.

Lời giải chi tiết

No single sensor type is reliable under *every* condition an autonomous vehicle might encounter — each has specific weaknesses (cameras struggle in glare/darkness, radar has coarse resolution, LiDAR is degraded by heavy precipitation) that are largely *independent* of each other's weaknesses. Relying on even the objectively "best" single sensor type still leaves the vehicle vulnerable during precisely the conditions that sensor type handles poorly. **Sensor fusion** combines fundamentally different sensing technologies so that each sensor's weaknesses are very unlikely to occur *simultaneously* with another sensor's weakness in the same condition — providing robust perception across a much wider range of real-world driving conditions than any single sensor, however good, could achieve alone.

Bài tập luyện tập

A pressure sensor's ADC has 10-bit resolution over a range of 0 to 1023 kPa. Calculate the smallest pressure change this ADC can distinguish.

Lời giải chi tiết

A 10-bit ADC has $2^{10} = 1024$ discrete levels. Smallest distinguishable step $= \dfrac{1023 \text{ kPa}}{1024} \approx 0.999$ kPa ≈ 1 kPa. Two real pressures differing by less than approximately 1 kPa would be recorded as the same digital reading by this ADC.

Bài tập luyện tập

Explain why sampling a rapidly oscillating signal too slowly can cause ”aliasing,” making the digital data appear to represent a much slower, false pattern.

Lời giải chi tiết

If the sampling rate is too low relative to how quickly the actual signal is changing, the samples captured may happen to catch the signal at similar points in its cycle each time (e.g. always near a peak), because the sampling interval and the signal's own cycle length are of a similar order — the sequence of recorded values can then trace out a smooth, slowly-varying pattern that does not represent the signal's true (much faster) behaviour at all. This false, misleadingly slow-looking pattern reconstructed from too-infrequent samples is **aliasing** — the only reliable fix is to sample fast enough, relative to the signal's true rate of change, to capture its real behaviour faithfully.

Bài tập luyện tập

Explain why the water-leak shutoff subsystem in the smart-home case study must be treated as hard real-time, while the hallway-light subsystem is not similarly critical.

Lời giải chi tiết

A delayed response to a detected water leak allows water damage to actively accumulate every additional second the valve remains open — a missed or late deadline directly translates into worsening, costly physical damage, exactly the defining feature of a **hard real-time** requirement. The hallway-light subsystem, by contrast, responding a fraction of a second late to motion/darkness causes at most a very minor, harmless inconvenience (a brief delay before the light turns on) — an occasional small delay here has no serious consequence, so it does not need the strict, guaranteed-deadline behaviour that the water-leak subsystem requires.

Bài tập luyện tập

Explain why increasing an ADC's resolution from 8-bit to 12-bit reduces quantisation error, and state the cost of this improvement.

Lời giải chi tiết

An 8-bit ADC divides its measurement range into $2^8 = 256$ discrete levels, while a 12-bit ADC divides the *same* range into $2^{12} = 4096$ discrete levels — each individual level therefore represents a much smaller range of the original analogue value, so rounding any given analogue reading to its nearest available digital level introduces a smaller maximum **quantisation error**. The cost is that each sample now requires more bits (12 instead of 8) to store, increasing the total data/storage size for the same number of samples, and increasing the data-processing/transmission bandwidth required.

Bài tập luyện tập

A safety system uses dual redundancy (two identical sensors) rather than triple redundancy for cost reasons. Explain a key limitation of dual redundancy if the two sensors ever disagree.

Lời giải chi tiết

With only **two** sensors, if their readings disagree, the system has no reliable way to determine *which* of the two sensors is the faulty one — both readings are equally "outvoted" by the other, with no third, independent reading to break the tie. **Triple modular redundancy** solves exactly this limitation: with three sensors, a majority (two agreeing readings) can be trusted over the single outlier, allowing the system to both continue operating correctly *and* identify which specific unit is faulty — a capability dual redundancy alone cannot provide.

Bài tập luyện tập

Three redundant pressure sensors in a chemical plant report 102 kPa, 101 kPa, and 150 kPa. State which reading the system should trust using majority voting, and identify the faulty sensor.

Lời giải chi tiết

The readings 102 kPa and 101 kPa closely agree with each other, while 150 kPa differs substantially from both — using **majority voting**, the system trusts the closely-agreeing pair (around 101–102 kPa) as the correct pressure, and flags the sensor reporting **150 kPa** as **faulty**, allowing continued safe operation using the trusted majority reading while the faulty sensor is scheduled for repair.

Bài tập luyện tập

In the traffic-light pseudocode from this chapter, explain what would happen if the line `WaitTimeB <- 0` inside the `else` branch were accidentally deleted, using a trace of a vehicle briefly appearing and then leaving Road B.

Lời giải chi tiết

Without resetting `WaitTimeB` to 0 when no vehicle is detected, the counter would never reset once it starts incrementing — even if a vehicle only waits briefly and then leaves Road B entirely, `WaitTimeB` would remain at whatever value it last reached and continue incrementing further on any *future* occasion a vehicle appears again, rather than restarting from zero for each new waiting vehicle. This bug would cause Road B to incorrectly get priority (its light turning green) much sooner than actually justified, based on an inflated, never-reset wait count rather than the true continuous waiting time of a vehicle currently present.

Bài tập luyện tập

Explain why the traffic-light system in this chapter is described as closed-loop rather than open-loop, referencing the specific line of pseudocode responsible.

Lời giải chi tiết

The system is **closed-loop** because the line `VehicleOnB <- READSENSOR(RoadBSensor)` continuously feeds a real, current sensor reading back into the controller's ongoing decision-making, and this feedback directly determines the controller's future actions (whether to switch the lights) — the defining test for closed-loop control established earlier in this chapter. An **open-loop** version would instead switch the lights purely based on a fixed timer (e.g. "switch every 60 seconds") with no sensor reading involved at all, regardless of whether any vehicle was actually present or waiting on Road B.

Bài tập luyện tập

A smart irrigation system logs soil moisture every 10 minutes for 12 hours. Calculate the number of readings taken, and explain one reason the system might additionally log the exact timestamp of each reading, not just the moisture value itself.

Lời giải chi tiết

Number of readings $= \dfrac{12 \times 60}{10} = 72$ readings. Logging the exact **timestamp** alongside each value allows the readings to later be correctly ordered and analysed as a time series (e.g. identifying exactly when soil moisture dropped sharply, correlating readings with a specific watering event or weather change) — without a timestamp, a bare sequence of moisture values would lose all information about exactly when each reading was taken, making meaningful trend analysis over time impossible.

Bài tập luyện tập

Explain why a home smoke detector is designed as a hard real-time, closed-loop-like monitoring-and-control system, referencing both properties.

Lời giải chi tiết

It is **hard real-time** because a delayed alarm in the event of an actual fire could directly endanger lives — there is no acceptable "slightly late" response when a genuine fire is detected, exactly the defining feature of hard real-time systems. It performs **monitoring and control** (rather than monitoring alone) because it does not merely record smoke levels for later review — it actively uses the sensor reading to immediately trigger an actuator (the alarm siren), taking direct action based on the sensed condition rather than just passively logging it.

Bài tập luyện tập

An automatic car wash uses a fixed sequence of stages (soap, scrub, rinse, dry), each running for a preset duration regardless of how dirty the car actually is. Classify this as open-loop or closed-loop, and suggest one sensor that could be added to make it closed-loop.

Lời giải chi tiết

This is **open-loop**: each stage runs for a fixed, pre-programmed duration with no feedback about the car's actual current cleanliness influencing the controller's behaviour. Adding a **dirt/-turbidity sensor** (measuring how much dirt is present in the rinse water, for example) could make the system **closed-loop**: the controller could extend the scrub/rinse stage automatically

if the sensor detects the car is still significantly dirty, and shorten it if the car is already clean — continuously adapting to the real, measured condition of the car rather than following a fixed schedule regardless of actual need.

Bài tập luyện tập

Explain why a spacecraft's life-support control system would use triple modular redundancy for its oxygen-level sensor rather than a single sensor, even though this adds cost and weight.

Lời giải chi tiết

A single oxygen sensor failing (giving an incorrect reading, or failing silently) in a spacecraft could lead the control system to make a catastrophically wrong decision (e.g. failing to increase oxygen supply during a genuine shortage, or wasting limited oxygen reserves reacting to a false low reading) — with astronauts' lives directly at stake, this is an unacceptable single point of failure. **Triple modular redundancy** allows the system to use a majority vote to both continue operating correctly if one sensor fails or gives an outlying reading, and to identify which specific sensor is faulty for repair/replacement — the added cost and weight is considered justified given the extreme consequences of an undetected sensor failure in this life-critical context.

Option: Object-Oriented Programming and Databases

Mục tiêu Unit: Tổng quan 4 phần tự chọn (Option) của IB CS. Đi sâu vào **lập trình hướng đối tượng (OOP)**: lớp, đối tượng, đóng gói, kế thừa, đa hình, sơ đồ lớp UML; và **cơ sở dữ liệu quan hệ**: khoá chính/khoá ngoại, chuẩn hoá 1NF–3NF, SQL, sơ đồ ER. Tổng quan gọn về **mô hình hoá & mô phỏng** và **khoa học Web**.

The IB CS syllabus offers **four Options**; a student studies **one** in depth for the exam. This chapter develops **Object-Oriented Programming** and **Databases** in full depth (two of the most commonly studied options), and gives a compact but complete treatment of **Modelling & Simulation** and **Web Science** for orientation and the "Options" section of Paper 1/2.

8.1 Classes, objects and attributes

Object-oriented programming (OOP) organises a program around **objects** — self-contained units that bundle together data and the operations on that data — rather than around a sequence of instructions acting on separate data structures.

Class vs object

A **class** is a blueprint/template that defines what **attributes** (data fields) and **methods** (behaviours/functions) its objects will have — a class itself is not a "thing" in the running program, only a description. An **object** (or **instance**) is a specific, concrete realisation of a class, with its own actual values stored in its attributes. Many objects can be created from the same class, each with independent attribute values but sharing the same defined methods.

Ví dụ mẫu · Worked example

Define a `Car` class with attributes `Make`, `Speed`, and a method to accelerate; then create two independent objects from it.

```
class Car
    private Make
    private Speed

    constructor(M)
        Make <- M
        Speed <- 0
    end constructor

    public method Accelerate(Amount)
        Speed <- Speed + Amount
```

```
    end method

    public method GetSpeed()
        return Speed
    end method
end class

Car1 <- new Car("Toyota")
Car2 <- new Car("Honda")
Car1.Accelerate(30)
output Car1.GetSpeed()   // outputs 30
output Car2.GetSpeed()   // outputs 0
```

`Car1` and `Car2` are two independent **objects** of the same `Car` **class**: calling `Accelerate` on `Car1` changes only `Car1`'s own `Speed` attribute, leaving `Car2`'s `Speed` at its initial value of 0 — each object maintains its own separate copy of the class's attributes.

VN **GIẢI THÍCH TIẾNG VIỆT**

Lớp (class): bản thiết kế định nghĩa thuộc tính (dữ liệu) và phương thức (hành vi) — bản thân lớp không phải là một "thực thể" đang chạy. **Đối tượng (object/instance)**: một thực thể cụ thể được tạo ra từ lớp, có giá trị thuộc tính RIÊNG của mình. Nhiều đối tượng có thể được tạo từ cùng một lớp, mỗi đối tượng độc lập với nhau về dữ liệu nhưng dùng chung các phương thức đã định nghĩa.

8.2 Encapsulation

Encapsulation bundles an object's data (attributes) together with the methods that operate on it, and restricts direct outside access to that data — typically by marking attributes `private` and providing controlled `public` methods (often called **getters** and **setters**) to read or modify them.

Access modifiers: `private` — accessible only from within the class's own methods; `public` — accessible from anywhere the object is visible, including outside code. Encapsulation's benefits: **data integrity** (a setter method can validate a new value before accepting it, rejecting invalid data that direct access could not prevent); **information hiding** (external code depends only on the public interface, not on internal implementation details, so the internal implementation can later be changed freely without breaking external code that uses the class); reduced risk of accidental interference between unrelated parts of a large program.

Ví dụ mẫu · Worked example

Add a `SetSpeed` method to the `Car` class that validates the new speed cannot be negative, demonstrating encapsulation's data-integrity benefit.

```
public method SetSpeed(NewSpeed)
    if NewSpeed >= 0 then
        Speed <- NewSpeed
    else
        output "Error: speed cannot be negative"
    end if
end method
```

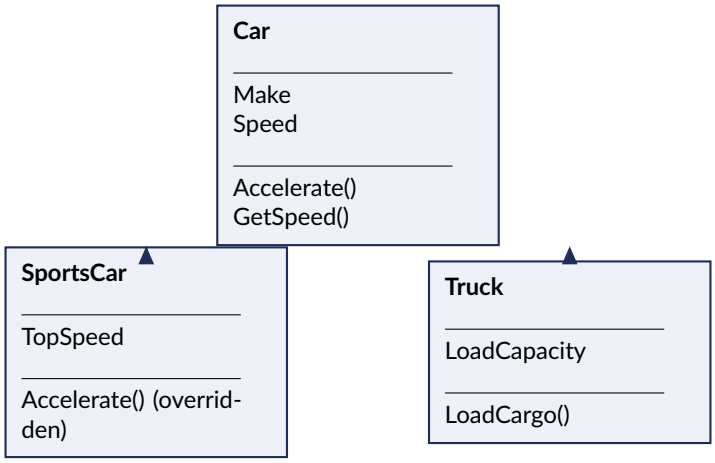
Because `Speed` is `private`, external code *cannot* bypass this method to set an invalid negative speed directly — it must go through `SetSpeed`, which enforces the validation rule. If `Speed` were instead `public`, any external code could set `Car1.Speed <- -50` directly, with no validation ever applied.

VN **GIẢI THÍCH TIẾNG VIỆT**

Đóng gói (encapsulation): gộp dữ liệu và phương thức lại, giới hạn truy cập trực tiếp (thuộc tính `private`, truy cập qua phương thức `public` như getter/setter). Lợi ích: **đảm bảo tính toàn vẹn dữ liệu** (setter có thể kiểm tra giá trị hợp lệ trước khi gán), **che giấu thông tin** (mã bên ngoài chỉ phụ thuộc vào giao diện công khai, cho phép thay đổi cách cài đặt bên trong mà không ảnh hưởng mã bên ngoài).

8.3 Inheritance

Inheritance lets a new class (the **subclass/child class**) reuse and extend the attributes and methods of an existing class (the **superclass/parent class**), modelling an "is-a" relationship (e.g. a `SportsCar` *is a* `Car`).



A UML class diagram: `SportsCar` and `Truck` both inherit from `Car` (arrow points to the superclass), each adding their own attributes/methods and `SportsCar` overriding `Accelerate()`.

Ví dụ mẫu · Worked example

Define `SportsCar` as a subclass of `Car`, adding a `TopSpeed` attribute and **overriding** `Accelerate` to cap speed at `TopSpeed`.

```
class SportsCar inherits Car
    private TopSpeed

    constructor(M, T)
        super(M)
        TopSpeed <- T
    end constructor

    public method Accelerate(Amount)
        Speed <- Speed + Amount
        if Speed > TopSpeed then
```

```
            Speed <- TopSpeed
        end if
    end method
end class

MyCar <- new SportsCar("Ferrari", 200)
MyCar.Accelerate(250)
output MyCar.GetSpeed()   // outputs 200
```

`SportsCar` **inherits** `Make`, `Speed`, and `GetSpeed()` directly from `Car` without rewriting them, calling `super(M)` in its constructor to reuse `Car`'s own constructor logic. It **overrides** `Accelerate` with its own version that additionally caps speed at `TopSpeed` — so accelerating by 250 from a standing start (0) is capped at 200, not 250, since `SportsCar`'s own `Accelerate` method (not `Car`'s) is the one actually called.

VN **GIẢI THÍCH TIẾNG VIỆT**

Kế thừa (inheritance): lớp con (subclass) dùng lại và mở rộng thuộc tính/phương thức của lớp cha (superclass), mô hình hoá quan hệ "là một loại của" (is-a). `super(...)` gọi lại constructor/phương thức của lớp cha. **Ghi đè (overriding)**: lớp con định nghĩa lại một phương thức đã có ở lớp cha với hành vi riêng — khi gọi phương thức đó trên đối tượng lớp con, phiên bản của lớp con được ưu tiên thực thi.

8.4 Polymorphism and abstraction

Polymorphism ("many forms") lets objects of *different* classes respond to the *same* method call, each in its own class-specific way — typically through method overriding combined with a shared superclass or interface.

Ví dụ mẫu · Worked example

Given `Car`, `SportsCar` (overriding `Accelerate`), and a plain `Truck` (using `Car`'s original `Accelerate` unmodified), trace calling `Accelerate(50)` on each, starting all from `Speed = 0`.

```
Vehicles <- [new Car("Kia"), new SportsCar("Ferrari", 40), new Truck("Volvo")]
loop I from 0 to LENGTH(Vehicles) - 1
    Vehicles[I].Accelerate(50)
    output Vehicles[I].GetSpeed()
end loop
```

The identical call `Accelerate(50)` produces *different* results depending on the object's actual class: plain `Car` outputs `50` (no cap); `SportsCar` (TopSpeed 40) outputs `40` (capped by its overridden method); `Truck` (using `Car`'s unmodified method, assuming no override) outputs `50`. The calling code did not need to know which specific subclass each object was — this is **polymorphism**: the same method call automatically produces the behaviour appropriate to each object's actual class.

Abstraction in OOP means exposing only the relevant details of an object through its public interface, hiding unnecessary internal complexity — closely related to encapsulation, but abstraction is about *conceptual simplicity* (what an object does) while encapsulation is about *access control* (protecting how it does it). An **abstract class** cannot itself be instantiated (no objects can be created directly from it) and exists purely to be inherited from, typically defining method signatures that every subclass *must* implement in its own way — guaranteeing a consistent public interface across many different subclasses while leaving the actual implementation to each one.

VN **GIẢI THÍCH TIẾNG VIỆT**

Đa hình (polymorphism): các đối tượng thuộc lớp KHÁC NHAU phản hồi KHÁC NHAU với CÙNG một lời gọi phương thức — mã gọi không cần biết đối tượng cụ thể thuộc lớp con nào. **Trừu tượng hoá (abstraction)** trong OOP: chỉ hiển thị phần cần thiết qua giao diện công khai, ẩn đi độ phức tạp bên trong. **Lớp trừu tượng (abstract class)**: không thể tạo đối tượng trực tiếp, chỉ để lớp con kế thừa và buộc phải cài đặt riêng các phương thức đã khai báo.

8.5 Composition and design considerations

Composition models a "has-a" relationship: one class contains an object of another class as one of its attributes, rather than inheriting from it (e.g. a `Car` *has a* `Engine`, but a `Car` *is not a* kind of `Engine`).

(!) Lỗi thường gặp: A common design error is using **inheritance** where **composition** would be more appropriate. Ask "is a *X* truly a specific kind of *Y*?" (inheritance/is-a) versus "does *X* simply *contain* or *use* a *Y*?" (composition/has-a). Modelling `Car inherits Engine` would be incorrect design — a car is not a type of engine, it merely *contains* one.

Ví dụ mẫu · Worked example

Model a `Car` that *has an* `Engine` using composition, rather than inheritance.

```
class Engine
    private HorsePower

    constructor(HP)
        HorsePower <- HP
    end constructor

    public method GetPower()
        return HorsePower
    end method
end class

class Car
    private CarEngine

    constructor(HP)
        CarEngine <- new Engine(HP)
    end constructor

    public method DescribeEngine()
```

```
        output "This car has " + CarEngine.GetPower() + " horsepower"
    end method
end class
```

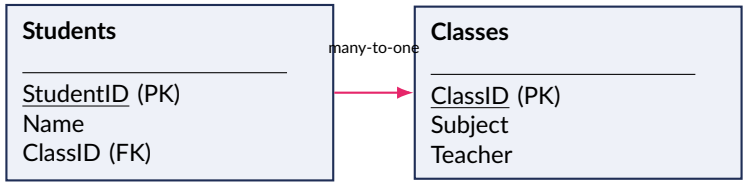
Here, `Car` does not inherit from `Engine` — instead, a `Car` object *contains* its own `Engine` object as one of its private attributes (`CarEngine`), correctly modelling the "has-a" relationship rather than incorrectly implying a car is a specialised type of engine.

VN **GIẢI THÍCH TIẾNG VIỆT**

Kết hợp (composition): mô hình hoá quan hệ "có một" (has-a) — một lớp CHỨA đối tượng của lớp khác làm thuộc tính, thay vì kế thừa. Lỗi thiết kế phổ biến: dùng kế thừa khi lẽ ra nên dùng kết hợp — luôn tự hỏi "X có PHẢI LÀ một loại Y không?" (kế thừa) hay "X chỉ đơn giản là CÓ/DÙNG một Y?" (kết hợp).

8.6 Relational databases: tables, keys and relationships

A **relational database** organises data into **tables** (relations), where each row is a **record** (tuple) and each column is a **field** (attribute). A **primary key** is a field (or combination of fields) that uniquely identifies each record in a table; a **foreign key** is a field in one table that references the primary key of another table, creating a **relationship** between them.



A simple entity-relationship (ER) diagram: each Student record references one Class via the `ClassID` foreign key, which matches a Class's primary key.

Referential integrity requires that every foreign-key value in a table must either match an existing primary-key value in the referenced table, or be null (indicating no relationship) — the database should reject an attempt to insert a foreign key value that does not correspond to any existing record (e.g. assigning a student to a `ClassID` that does not exist), preventing "orphaned" references. Relationships can be **one-to-many** (one class has many students, as above), **one-to-one** (each record in one table relates to exactly one record in another), or **many-to-many** (typically implemented using a separate "junction" table holding pairs of foreign keys, e.g. students and clubs, where each student can join many clubs and each club has many students).

VN **GIẢI THÍCH TIẾNG VIỆT**

Cơ sở dữ liệu quan hệ: dữ liệu tổ chức thành **bảng**, mỗi dòng là một **bản ghi**, mỗi cột là một **trường**. **Khoá chính (primary key)** định danh duy nhất mỗi bản ghi; **khoá ngoại (foreign key)** ở một bảng tham chiếu tới khoá chính của bảng khác, tạo **quan hệ** giữa các bảng. **Toàn vẹn tham chiếu**: giá trị khoá ngoại phải khớp với một khoá chính đang tồn tại (hoặc để trống) — ngăn tham chiếu "mồ côi". Quan hệ có thể là một–nhiều, một–một, hoặc nhiều–nhiều (dùng bảng trung gian).

8.7 Normalization

Normalization is a systematic process of organising a database's tables to reduce data redundancy and avoid update anomalies (where the same fact, stored in multiple places, could become inconsistent if only some copies are updated).

1NF (First Normal Form): every field holds a single, atomic value (no repeating groups or lists crammed into one field), and each record is uniquely identifiable. **2NF (Second Normal Form):** the table is in 1NF, *and* every non-key field depends on the *entire* primary key, not just part of it (only relevant when the primary key is composite/made of multiple fields) — eliminates **partial dependency**. **3NF (Third Normal Form):** the table is in 2NF, *and* no non-key field depends on *another non-key field* rather than directly on the primary key — eliminates **transitive dependency**.

Ví dụ mẫu · Worked example

A table `StudentPhones(StudentID, Name, Phone1, Phone2, Phone3)` stores up to three phone numbers per student in separate columns. Explain why this violates 1NF and redesign it correctly.

This violates **1NF** because it effectively stores a *repeating group* (a list of phone numbers) spread awkwardly across multiple similarly-named columns — students with only one phone number waste two empty columns, and a student needing a fourth number cannot be accommodated without changing the table structure. **Correct redesign:** split into two tables — `Students(StudentID, Name)` and `Phones(PhoneID, StudentID, PhoneNumber)`, where `StudentID` in `Phones` is a foreign key referencing `Students`. Now each student can have any number of phone numbers, each phone number is a single atomic value in its own record, and no columns are wasted.

Ví dụ mẫu · Worked example

A table `OrderItems(OrderID, ProductID, ProductName, Quantity)` has composite primary key (`OrderID`, `ProductID`). `ProductName` depends only on `ProductID`, not on the full composite key. Explain the 2NF violation and redesign it.

This is a **partial dependency**: `ProductName` depends on only *part* of the composite primary key (`ProductID` alone), not the whole key (`OrderID`, `ProductID`) — violating **2NF**. If a product's name ever needs correcting, it would need to be updated in *every* order row containing that product, risking inconsistency. **Correct redesign:** split into `Products(ProductID, ProductName)` and `OrderItems(OrderID, ProductID, Quantity)`, where `ProductID` in `OrderItems` is a foreign key. Now `ProductName` is stored exactly once per product, regardless of how many orders include it.

Ví dụ mẫu · Worked example

A table `Employees(EmployeeID, DepartmentID, DepartmentName)` stores the department's name directly alongside each employee. Explain the 3NF violation and redesign it.

`DepartmentName` depends on `DepartmentID` (a *non-key* field), not directly on the primary key `EmployeeID` — this is a **transitive dependency**, violating **3NF**. If a department is renamed, every employee row referencing that department must be updated, risking inconsistency if some rows are missed. **Correct redesign:** split into `Departments(DepartmentID, DepartmentName)` and `Employees(EmployeeID, DepartmentID)`, where `DepartmentID` in `Employees` is a foreign key referencing `Departments`. The department name is now stored

exactly once, in one place, and updating it there automatically applies everywhere it is referenced.

VN **GIẢI THÍCH TIẾNG VIỆT**

Chuẩn hoá (normalization): quy trình tổ chức lại bảng để giảm dư thừa dữ liệu và tránh bất thường khi cập nhật. **1NF**: mỗi trường chỉ chứa MỘT giá trị nguyên tố, không có nhóm lặp lại. **2NF**: 1NF + mọi trường không khoá phụ thuộc vào TOÀN BỘ khoá chính (không phụ thuộc bộ phận — chỉ liên quan khi khoá chính gồm nhiều trường). **3NF**: 2NF + không trường không khoá nào phụ thuộc vào trường không khoá KHÁC (không phụ thuộc bắc cầu).

8.8 SQL: querying and modifying data

SQL (Structured Query Language) is the standard language for defining, querying, and modifying relational database tables.

Statement	Purpose
`SELECT`	Retrieve data: columns, from which table(s), filtered by `WHERE`, ordered by `ORDER BY`
`INSERT`	Add a new record to a table
`UPDATE`	Modify existing record(s) matching a condition
`DELETE`	Remove record(s) matching a condition
`JOIN`	Combine rows from two or more tables based on a related field (e.g. a foreign key matching a primary key)

Aggregate functions used with `SELECT`: `COUNT()` (number of matching rows), `SUM()` (total of a numeric field), `AVG()` (mean average), `MAX()`/`MIN()` (largest/smallest value), often combined with `GROUP BY` to aggregate per category.

Ví dụ mẫu · Worked example

Write an SQL query to list the name and grade of every student scoring above 80, highest first.

```
SELECT Name, Grade
FROM Students
WHERE Grade > 80
ORDER BY Grade DESC;
```

Ví dụ mẫu · Worked example

Using the `Students(StudentID, Name, ClassID)` and `Classes(ClassID, Subject, Teacher)` tables from earlier, write an SQL query to list each student's name alongside the subject they are enrolled in.

```
SELECT Students.Name, Classes.Subject
FROM Students
JOIN Classes ON Students.ClassID = Classes.ClassID;
```

The `JOIN...ON` clause combines each `Students` row with the matching `Classes` row where the foreign key `Students.ClassID` equals the referenced primary key `Classes.ClassID` — without this join, `Students` alone has no `Subject` column to display directly.

Ví dụ mẫu · Worked example

Write an SQL query to count how many students are enrolled in each `ClassID`, listing only classes with more than 20 students.

```
SELECT ClassID, COUNT(StudentID) AS NumStudents
FROM Students
GROUP BY ClassID
HAVING COUNT(StudentID) > 20;
```

`GROUP BY ClassID` groups all student rows by their class before counting; `HAVING` (not `WHERE`) is required here because the filter condition applies to the *aggregated* count, not to individual rows — `WHERE` filters rows *before* grouping, while `HAVING` filters groups *after* aggregation.

Ví dụ mẫu · Worked example

Write SQL statements to insert a new student, then update their class, then delete a student who has left the school.

```
INSERT INTO Students (StudentID, Name, ClassID)
VALUES (1042, "Minh Anh", 7);

UPDATE Students
SET ClassID = 9
WHERE StudentID = 1042;

DELETE FROM Students
WHERE StudentID = 1042;
```

(!) Lỗi thường gặp: Always include a `WHERE` clause with `UPDATE` and `DELETE` unless you genuinely intend to affect *every* row in the table. Omitting `WHERE` from `DELETE FROM Students;` would delete *every single student record* in the table, not just one — a catastrophic and common real-world mistake.

VN **GIẢI THÍCH TIẾNG VIỆT**

SQL: ngôn ngữ truy vấn cơ sở dữ liệu chuẩn. `SELECT...WHERE...ORDER BY`: truy vấn dữ liệu có điều kiện, sắp xếp. `JOIN`: kết hợp dữ liệu từ nhiều bảng qua khoá liên kết. `GROUP BY...HAVING`: nhóm dữ liệu rồi lọc theo điều kiện TỔNG HỢP (khác với `WHERE` lọc từng dòng TRƯỚC khi nhóm). `INSERT/UPDATE/DELETE`: thêm/sửa/xoá dữ liệu — LUÔN dùng `WHERE` với UPDATE/DELETE trừ khi thực sự muốn áp dụng cho TOÀN BỘ bảng.

8.9 ACID properties (brief)

Database transactions (a group of operations that must all succeed or all fail together) are expected to satisfy **ACID** properties: **Atomicity** (a transaction is all-or-nothing — if any part fails, the entire transaction is rolled back, leaving no partial changes); **Consistency** (a transaction only ever moves the database from one valid state to another, never violating defined rules like referential integrity); **Isolation** (concurrent transactions do not interfere with each other's intermediate results); **Durability** (once a transaction is confirmed complete/"committed," its changes survive even a subsequent power failure or crash).

Ví dụ mẫu · Worked example

A bank transfer moves $100 from Account A to Account B, requiring two updates: subtract $100 from A, then add $100 to B. Explain, using **atomicity**, what must happen if the system crashes immediately after the first update but before the second.

Atomicity guarantees the transaction is treated as a single indivisible unit: if it does not complete *entirely*, the database must roll back *any* partial changes already made, restoring Account A's original balance as if the transfer had never been attempted. Without atomicity, a crash at this exact moment would leave $100 permanently deducted from A but never credited to B — the money would simply vanish from the system, a serious and unacceptable financial error.

VN **GIẢI THÍCH TIẾNG VIỆT**

ACID: **Atomicity** (giao dịch phải trọn vẹn — thất bại 1 phần thì huỷ toàn bộ), **Consistency** (luôn chuyển CSDL từ trạng thái hợp lệ này sang trạng thái hợp lệ khác), **Isolation** (các giao dịch đồng thời không can thiệp lẫn nhau), **Durability** (giao dịch đã hoàn tất thì tồn tại vĩnh viễn, kể cả khi mất điện/crash ngay sau đó).

8.10 Option: Modelling and simulation (overview)

A **model** is a simplified representation of a real system, deliberately omitting irrelevant detail so the essential behaviour can be studied or predicted. A **simulation** runs that model forward over (simulated) time to observe behaviour, without the cost, danger, or impracticality of experimenting on the real thing (e.g. flight simulators, climate models, epidemic-spread models).

Validation checks whether a model's behaviour accurately reflects real-world behaviour, typically by comparing simulation output against real observed data. **Verification** checks whether the model/program was built correctly according to its own design specification (i.e. free of implementation bugs), independent of whether the design itself is realistic. A **deterministic** model always produces the exact same output for the same input (no randomness); a **stochastic** model incorporates randomness/probability, so repeated runs with identical inputs can produce different outputs (e.g. modelling random chance events like customer arrival times).

Ví dụ mẫu · Worked example

A traffic simulation's predicted congestion patterns are compared against a month of real traffic-camera data and found to match closely. Separately, the simulation's source code is checked line-by-line against its design document to confirm it was implemented correctly. Identify which activity is validation and which is verification.

Comparing the simulation's output against **real-world observed data** is **validation** — checking whether the model accurately represents reality. Checking the **source code against its design specification** is **verification** — checking whether the model was built correctly according to its intended design, regardless of whether that design itself is realistic. A model can pass verification (correctly implements its design) yet still fail validation (the design itself does not match reality), and vice versa.

VN **GIẢI THÍCH TIẾNG VIỆT**

Mô hình: bản mô phỏng đơn giản hoá của hệ thống thực. **Mô phỏng**: chạy mô hình theo thời gian để quan sát hành vi mà không cần thử nghiệm trên hệ thống thật. **Xác thực (validation)**: mô hình có phản ánh đúng thực tế không (so với dữ liệu thật)? **Kiểm chứng (verification)**: mô

hình có được xây dựng ĐÚNG theo thiết kế không (không có lỗi lập trình)? **Mô hình tất định**: luôn cho cùng kết quả với cùng đầu vào; **mô hình ngẫu nhiên (stochastic)**: có yếu tố xác suất, kết quả có thể khác nhau giữa các lần chạy.

8.11 Option: Web science (overview)

The web operates on a **client–server** architecture over **HTTP/HTTPS**: a browser (client) requests a page; a web server responds with content, typically built from **HTML** (structures content), **CSS** (styles/formats it), and **JavaScript** (adds interactivity/behaviour). Hyperlinks connect pages into a vast **directed graph**.

Search engines use automated programs called **crawlers** (or "spiders") to systematically discover and index web pages by following hyperlinks from page to page. A **ranking algorithm** then orders search results by estimated relevance/importance — historically influenced by ideas like **link-based authority** (a page linked to by many other important pages is itself likely important, similar in spirit to the original PageRank algorithm), alongside many other modern signals (content relevance, user engagement, freshness). Since **HTTP is stateless** (each request is treated independently, with no built-in memory of previous requests), **cookies** are small pieces of data a server asks the browser to store and resend with every subsequent request, allowing the server to recognise the same client across multiple requests (e.g. keeping a user logged in, or remembering the contents of a shopping cart).

Ví dụ mẫu · Worked example

Explain why a shopping website would lose track of a user's shopping-cart contents between page loads if cookies were completely disabled in the user's browser.

HTTP is stateless: by default, the server treats each incoming page request as entirely independent, with no automatic memory linking it to any previous request from the same user. Normally, a **cookie** issued on the first request is sent back by the browser with every subsequent request, letting the server recognise "this is the same shopping session" and retrieve the associated cart contents. With cookies disabled, the server has no reliable way to link a new page request back to the earlier one where items were added to the cart, so (without some alternative session-tracking mechanism) the cart would appear empty on every new page load.

VN **GIẢI THÍCH TIẾNG VIỆT**

Web hoạt động theo mô hình **client–server** qua **HTTP/HTTPS**. **HTML** = cấu trúc, **CSS** = định dạng, **JavaScript** = tương tác. **Crawler**: chương trình tự động thu thập trang web bằng cách theo các siêu liên kết. **Thuật toán xếp hạng**: sắp xếp kết quả tìm kiếm theo mức độ liên quan/uy tín (ví dụ dựa trên số lượng/chất lượng liên kết trỏ tới trang). **Cookie**: giúp lưu trạng thái vì HTTP vốn không lưu trạng thái (stateless) — ví dụ giữ giỏ hàng, phiên đăng nhập.

8.12 NoSQL databases (brief comparison)

Not all data fits neatly into rigid, predefined tables. **NoSQL** ("Not Only SQL") databases offer alternative data models better suited to certain workloads.

Document databases (e.g. storing JSON-like documents) let each record have a flexible, even different, set of fields without requiring every record to share an identical rigid schema — convenient when data naturally varies in structure (e.g. product listings with very different attributes per category). **Key-value stores** are essentially large-scale, distributed hash tables (Unit 5) — extremely fast simple lookups by key, ideal for caching or session storage. Compared with a **relational database** (this chapter's main focus), NoSQL databases typically sacrifice some of the strict consistency/referential-integrity guarantees (ACID) in exchange for easier horizontal scaling across many servers and more flexible, schema-less data storage — an appropriate trade-off for some large-scale web applications, but not for applications (e.g. banking) where strict transactional consistency is essential.

Ví dụ mẫu · Worked example

A social media platform needs to store billions of simple "user ID → session token" lookups with extremely fast read/write speed, while a bank needs strict guarantees that a money transfer is never partially applied. Identify the more appropriate database type for each, and justify.

For the social media session lookups, a **key-value store** (a NoSQL type) is most appropriate: the data is a simple, massive-scale key-to-value mapping requiring extremely fast lookups, with no need for complex relationships between records or strict transactional guarantees across multiple records. For the bank's money transfer, a **relational database** with full **ACID** guarantees is essential: the strict **atomicity** guarantee (Unit 8, earlier) that a transaction is entirely all-or-nothing is critical to prevent money disappearing or being duplicated during a transfer — a property NoSQL databases often relax in exchange for scalability, making them unsuitable for this specific financial use case.

VN **GIẢI THÍCH TIẾNG VIỆT**

NoSQL: mô hình dữ liệu thay thế cho bảng quan hệ cứng nhắc. **Cơ sở dữ liệu tài liệu (document)**: mỗi bản ghi có cấu trúc linh hoạt, không cần giống nhau hoàn toàn. **Kho khoá–giá trị (key-value store)**: về bản chất là bảng băm quy mô lớn, tra cứu cực nhanh. So với cơ sở dữ liệu quan hệ, NoSQL thường đánh đổi bớt tính nhất quán/toàn vẹn chặt chẽ (ACID) để đổi lấy khả năng mở rộng ngang dễ dàng hơn — phù hợp cho web quy mô lớn nhưng KHÔNG phù hợp cho ứng dụng cần đảm bảo giao dịch nghiêm ngặt như ngân hàng.

8.13 Common design patterns (brief)

A **design pattern** is a reusable, well-tested general solution to a commonly recurring design problem in object-oriented software — not specific code to copy, but a proven structural approach.

Singleton pattern: ensures a class has *exactly one* instance throughout the entire program, with a single global point of access to it (e.g. a single shared configuration-settings object that every part of a program reads from, avoiding inconsistent copies). **Factory pattern:** centralises object creation logic in one dedicated method/class, rather than scattering the specific details of "how to construct a particular object" throughout the codebase — useful when the exact class to instantiate depends on some runtime condition (e.g. a `ShapeFactory` that creates a `Circle`, `Square`, or `Triangle` object depending on a parameter, without the calling code needing to know each class's specific constructor details).

Ví dụ mẫu · Worked example

A drawing application must create different shape objects (`Circle`, `Square`, `Triangle`) based on which toolbar button the user clicks, without the toolbar code needing to know the details of each shape class's constructor. Explain how a Factory pattern helps here.

Instead of the toolbar's click-handling code directly containing logic like `if ButtonName = "Circle" then new Circle(...) else if ButtonName = "Square" then new Square(...)` scattered throughout the interface code, a **Factory** method (e.g. `CreateShape(ButtonName)`) centralises this decision in one place: the toolbar code simply calls `CreateShape(ButtonName)` and receives back the correct object, without needing to know or repeat each shape class's specific construction details itself. If a new shape type is added later, only the factory method needs updating, rather than searching through scattered creation logic across the whole codebase.

VN **GIẢI THÍCH TIẾNG VIỆT**

Mẫu thiết kế (design pattern): giải pháp tổng quát, đã được kiểm chứng, có thể tái sử dụng cho một vấn đề thiết kế thường gặp — không phải đoạn mã cụ thể để sao chép mà là cách tiếp cận cấu trúc đã được chứng minh hiệu quả. **Singleton**: đảm bảo một lớp CHỈ CÓ DUY NHẤT một thực thể trong toàn chương trình. **Factory**: tập trung logic tạo đối tượng vào một nơi duy nhất, hữu ích khi lớp cụ thể cần tạo phụ thuộc vào điều kiện lúc chạy chương trình.

8.14 Database indexes and views

As tables grow large, two further database features become essential for performance and usability.

An **index** is an auxiliary data structure (conceptually similar to a book's index, or in practice often implemented using a B-tree — a wide, shallow, balanced tree related to the BSTs of Unit 5) built on one or more columns, allowing the database to locate matching rows without scanning the entire table. A `SELECT` query filtering on an **indexed** column can jump almost directly to matching rows ($O(\log n)$-like performance); without an index, the database must check every single row (a slow **full table scan**, $O(n)$). Indexes speed up reads, but slightly slow down `INSERT`/`UPDATE`/`DELETE` (since the index itself must also be updated) and use additional storage — so indexes are typically added only to columns frequently used in `WHERE` clauses or joins, not to every column. A **view** is a saved, named `SELECT` query that behaves like a virtual table: querying the view re-runs its underlying query each time, without duplicating the actual data — useful for simplifying complex, frequently-repeated queries or restricting which columns/rows certain users can see.

Ví dụ mẫu · Worked example

A `Students` table with 2 million rows is frequently queried by `StudentID` (the primary key, automatically indexed) but also very frequently searched by `Surname` (not indexed). Explain the performance problem this causes and how to fix it.

Searching by `StudentID` benefits from the automatic primary-key index, giving fast lookups. But searching by `Surname` with no index forces the database to perform a **full table scan** — checking all 2 million rows one by one to find matches — which is slow and grows worse as the table grows. Adding an explicit **index** on the `Surname` column would let the database locate matching surnames far more directly, dramatically speeding up this frequent query type, at the small cost of slightly slower inserts/updates (since the new index must also be maintained) and some additional storage for the index itself.

VN **GIẢI THÍCH TIẾNG VIỆT**

Chỉ mục (index): cấu trúc dữ liệu phụ (thường dựa trên cây B-tree) giúp tìm dòng khớp mà không cần quét toàn bộ bảng — tăng tốc `SELECT` nhưng làm chậm `INSERT/UPDATE/DELETE` một chút và tốn thêm bộ nhớ. **View**: truy vấn `SELECT` đã lưu, hoạt động như bảng ảo — không nhân bản dữ liệu thật, hữu ích để đơn giản hoá truy vấn phức tạp hay giới hạn quyền xem dữ liệu.

8.15 Multiple inheritance and interfaces

Some OOP languages allow a class to inherit from *more than one* superclass simultaneously (**multiple inheritance**); many others deliberately avoid this in favour of **interfaces**.

Multiple inheritance lets a class combine behaviour from several superclasses at once (e.g. a `FlyingCar` inheriting from both `Car` and `Aircraft`) but risks the **diamond problem**: if two superclasses both define a method with the same name (differently), the subclass inherits an ambiguous conflict — which version should apply? An **interface** sidesteps this: it defines a set of method *signatures* that any implementing class must provide its own implementation for, with *no* actual inherited implementation to conflict over. A class can implement *multiple* interfaces safely (gaining multiple sets of required behaviours) while still inheriting actual implemented code from only a *single* superclass — avoiding the diamond problem entirely while still supporting a form of "multiple inheritance" of behaviour contracts.

Ví dụ mẫu · Worked example

Explain the "diamond problem" using a `FlyingCar` class that inherits from both `Car` and `Aircraft`, where both superclasses separately define a method `Stop()` with different implementations.

If `FlyingCar` inherits from *both* `Car` (whose `Stop()` applies brakes) and `Aircraft` (whose `Stop()` cuts engine thrust and deploys a parachute), calling `myFlyingCar.Stop()` creates a genuine ambiguity: which superclass's `Stop()` implementation should actually run? Different languages resolve this ambiguity differently (e.g. by inheritance order, or requiring the programmer to explicitly specify), but the underlying problem — an inherited method existing in conflicting versions from multiple parents — is exactly the **diamond problem**, and is a central reason some OOP languages (e.g. Java) disallow multiple inheritance of implemented classes altogether, permitting multiple *interfaces* instead.

VN **GIẢI THÍCH TIẾNG VIỆT**

Đa kế thừa (multiple inheritance): một lớp kế thừa từ NHIỀU lớp cha cùng lúc — có nguy cơ gặp **vấn đề kim cương (diamond problem)** khi hai lớp cha có cùng tên phương thức nhưng cài đặt khác nhau, gây mơ hồ không biết dùng bản nào. **Interface**: chỉ khai báo CHỮ KÝ phương thức (không có cài đặt), lớp con BẮT BUỘC tự cài đặt riêng — một lớp có thể triển khai NHIỀU interface an toàn (không xung đột) trong khi chỉ kế thừa cài đặt thật từ MỘT lớp cha duy nhất.

8.16 Method overloading vs overriding

Two frequently-confused OOP concepts both let a method name be reused, but in fundamentally different ways.

Method overriding (this chapter, earlier) occurs *between* a superclass and subclass: the subclass redefines a method with the *same* name and parameters, replacing the superclass's behaviour when called on a subclass object. **Method overloading** occurs *within a single class*: multiple methods share the *same name* but have *different parameter lists* (a different number and/or type of parameters) — the correct version to call is automatically chosen based on the arguments actually supplied at the call site, allowing one conceptual operation (e.g. "add") to sensibly handle several different input situations under one shared, intuitive name.

Ví dụ mẫu · Worked example

A `Calculator` class overloads an `Add` method: one version takes two integers, another takes a list of numbers. Show both, and trace two different calls.

```
class Calculator
    public method Add(A, B)
        return A + B
    end method

    public method Add(NumberList)
        Total <- 0
        loop I from 0 to LENGTH(NumberList) - 1
            Total <- Total + NumberList[I]
        end loop
        return Total
    end method
end class

Calc <- new Calculator()
output Calc.Add(3, 5)              // calls the two-integer version: outputs 8
output Calc.Add([1, 2, 3, 4])       // calls the list version: outputs 10
```

Even though both methods share the name `Add`, the calculator automatically selects the correct version based on what was passed in: two individual numbers invoke the first version, while a single list argument invokes the second — this is **overloading**, entirely distinct from overriding, which instead involves a subclass replacing a superclass's method of the exact same signature.

VN **GIẢI THÍCH TIẾNG VIỆT**

Ghi đè (overriding): xảy ra GIỮA lớp cha và lớp con — lớp con định nghĩa lại phương thức CÙNG tên và tham số. **Nạp chồng (overloading)**: xảy ra TRONG một lớp — nhiều phương thức CÙNG tên nhưng KHÁC danh sách tham số; phiên bản đúng được chọn tự động dựa trên đối số truyền vào lúc gọi. Đây là hai khái niệm khác nhau, dễ bị nhầm lẫn.

8.17 Client-server database architecture

In practice, a database rarely runs in isolation — it typically operates as the **server** side of a client-server system (Unit 3), with applications acting as clients issuing queries to it.

A **Database Management System (DBMS)** is the software that manages a database, handling query execution, enforcing constraints (primary/foreign keys, data types), controlling concurrent access (Unit 8's ACID isolation property), and managing security/permissions — applications never manipulate the raw stored data files directly; they always go through the DBMS. In a typical **three-tier architecture**: the **presentation tier** (user interface, e.g. a web browser or app) sends requests to the **application/logic tier** (business logic, e.g. a web server processing the request), which in turn queries the **data tier** (the DBMS and its underlying database) and returns results back up through the same layers. Separating these tiers means, for example, the underlying database technology could be changed without needing to rewrite the user interface, since the interface only ever communicates with the application tier, never directly with the database.

Ví dụ mẫu · Worked example

Explain why a banking app's mobile interface never connects directly to the bank's database, instead always going through an intermediate application server.

Connecting the client (mobile app) directly to the database would expose the database's connection details and structure directly to every user's device, creating serious **security** risks (e.g. a compromised or reverse-engineered app could potentially be manipulated to run unauthorised queries directly against sensitive financial data) and would make it far harder to enforce consistent business rules (e.g. daily withdrawal limits) across every possible client. Routing all requests through an intermediate **application/logic tier** lets the bank centrally enforce security checks, business rules, and data validation in one controlled place, before any request ever reaches the actual database — the mobile app only ever "asks" the application server to perform actions on its behalf, never touching the database directly.

VN **GIẢI THÍCH TIẾNG VIỆT**

DBMS: phần mềm quản lý cơ sở dữ liệu — thực thi truy vấn, ràng buộc khoá, kiểm soát truy cập đồng thời, bảo mật. Ứng dụng KHÔNG BAO GIỜ thao tác trực tiếp file dữ liệu thô, luôn thông qua DBMS. **Kiến trúc 3 tầng**: tầng giao diện (presentation) → tầng logic ứng dụng (application) → tầng dữ liệu (data/DBMS) — tách tầng giúp thay đổi công nghệ CSDL mà không cần viết lại giao diện người dùng, và tập trung kiểm soát bảo mật/quy tắc nghiệp vụ ở tầng ứng dụng.

8.18 Practice

Bài tập luyện tập

A `Students` table has a repeating group storing three phone numbers in one field. Which normal form does this violate?

(A) 1NF
(B) 2NF
(C) 3NF
(D) None; this is fully normalized

Lời giải chi tiết

1NF requires every field to hold a single, atomic value — a repeating group (multiple phone numbers crammed into one field) violates 1NF and should be split into a separate related table. **(A)**.

Bài tập luyện tập

A traffic simulation predicts congestion patterns that closely match real observed traffic data collected over a month. This comparison against real-world data is an example of:

(A) Verification
(B) Validation
(C) Normalization
(D) Encapsulation

Lời giải chi tiết

Comparing the model's output against real-world observations checks whether the model accurately represents reality — this is **validation** (verification would instead check that the simulation code correctly implements its design specification). **(B)**.

Bài tập luyện tập

In the `SportsCar inherits Car` example, which OOP principle is demonstrated by `Speed` and `Make` being declared `private` and only accessible via methods?

(A) Inheritance
(B) Polymorphism
(C) Encapsulation
(D) Normalization

Lời giải chi tiết

Restricting direct access to an object's internal data, exposing it only through defined methods, is the definition of **encapsulation**. **(C)**.

Bài tập luyện tập

Explain briefly why HTTP cookies are necessary given that HTTP itself is a stateless protocol.

Lời giải chi tiết

HTTP treats every request independently, with no built-in memory of previous requests from the same user — without help, a server could not tell that two requests came from the same logged-in visitor. **Cookies** are small pieces of data the server asks the browser to store and resend with every subsequent request, letting the server recognise the same client across requests (e.g. keeping a user logged in or a shopping cart intact) despite HTTP itself having no memory of past interactions.

Bài tập luyện tập

A table `Books(ISBN, Title, AuthorID, AuthorName)` stores the author's name directly alongside every book. Identify the normalization violation and redesign the table(s) correctly.

Lời giải chi tiết

`AuthorName` depends on `AuthorID` (a non-key field), not directly on the primary key `ISBN` — this is a **transitive dependency**, violating **3NF**. **Redesign:** split into `Authors(AuthorID, AuthorName)` and `Books(ISBN, Title, AuthorID)`, with `AuthorID` in `Books` as a foreign key referencing `Authors` — the author's name is now stored once, correctly updating everywhere automatically if it ever changes.

Bài tập luyện tập

Write an SQL query to find the average `Grade` of all students in `ClassID = 5`.

Lời giải chi tiết

```
SELECT AVG(Grade) AS AverageGrade
FROM Students
WHERE ClassID = 5;
```

The `WHERE` clause filters to only rows where `ClassID = 5` *before* the `AVG()` aggregate function computes the mean of the remaining `Grade` values.

Bài tập luyện tập

Explain the difference between `WHERE` and `HAVING` in SQL, using an example query that counts orders per customer and only shows customers with more than 5 orders.

Lời giải chi tiết

`WHERE` filters individual *rows before* any grouping/aggregation occurs; `HAVING` filters *groups after* aggregation has been computed. For "customers with more than 5 orders": `WHERE` cannot be used for this condition because "more than 5 orders" is a property of the *aggregated group* (a count), not of any single row — the correct query is:

```
SELECT CustomerID, COUNT(OrderID) AS NumOrders
FROM Orders
GROUP BY CustomerID
HAVING COUNT(OrderID) > 5;
```

Bài tập luyện tập

A junior programmer models a `Manager` class as inheriting from an `Employee` class, and separately models a `Department` class as inheriting from a `Manager` class (reasoning: "a department has a manager"). Explain why this second inheritance relationship is a design error, and correct it.

Lời giải chi tiết

Inheritance should only be used for a genuine "is-a" relationship, and a **Department is not a kind of Manager** — the programmer's own reasoning ("a department *has* a manager") actually describes a **has-a** relationship instead. The correct design uses **composition**: `Department` should contain a reference to a `Manager` object as one of its own attributes (e.g. a `DepartmentHead` attribute of type `Manager`), not inherit from `Manager`. Using inheritance here would incorrectly give every `Department` object all of a `Manager`'s own personal attributes and behaviours (e.g. a salary, a start date) as if a department itself were a type of employee, which makes no logical sense.

Bài tập luyện tập

Given the `Vehicles` polymorphism example in this chapter, explain what would happen if `Truck` also **overrode** `Accelerate` to add a "towing weight" penalty, without changing the loop code that calls `Vehicles[I].Accelerate(50)`.

Lời giải chi tiết

Because of **polymorphism**, the exact same loop code — which simply calls `Accelerate(50)` on each object without knowing or checking its specific class — would automatically invoke *whichever* `Accelerate` method belongs to each object's actual class at runtime. If `Truck` now has its own overridden `Accelerate` (applying a towing penalty), that version would automatically run for `Truck` objects, while `Car` and `SportsCar` objects would continue using their own respective versions — no change to the calling loop is required at all, which is precisely the benefit polymorphism provides: new/changed behaviour in a subclass integrates seamlessly with existing code written against the shared superclass interface.

Bài tập luyện tập

Explain why an `Employees` table with primary key `EmployeeID` and non-key fields `FirstName`, `LastName`, `Salary` is already in 3NF (assuming no other complicating fields), and what would need to be true for it to *violate* 3NF.

Lời giải chi tiết

This table is in **3NF** because every non-key field (`FirstName`, `LastName`, `Salary`) depends directly and only on the primary key `EmployeeID`, with no field depending on another *non-key* field. It would **violate** 3NF if, for example, a `TaxBracket` field were added whose value was actually determined by `Salary` (a non-key field) rather than independently by `EmployeeID` — that would be a transitive dependency (`EmployeeID` → `Salary` → `TaxBracket`), requiring `TaxBracket` to be split into its own table if it needs to be stored explicitly at all.

Bài tập luyện tập

Explain the "Isolation" property of ACID transactions using an example of two customers simultaneously trying to book the very last seat on a flight.

Lời giải chi tiết

Isolation guarantees that concurrently-running transactions do not interfere with each other's intermediate, uncommitted results. If two customers simultaneously attempt to book the last remaining seat, isolation ensures the database processes these overlapping transactions in a way that behaves *as if* they had run one strictly after the other (even though they were actually concurrent) — so only one booking transaction can successfully reserve the seat and decrement the seat count, while the second transaction correctly sees the seat is no longer available and fails/rolls back, rather than both transactions reading "1 seat available" simultaneously and both successfully booking the same physical seat.

Bài tập luyện tập

A students' club-membership system needs to record that each student can join many clubs, and each club can have many student members. Explain how this many-to-many relationship should be implemented in a relational database, since neither table alone can hold a simple single foreign key for the other.

Lời giải chi tiết

A many-to-many relationship requires a separate **junction (linking) table**, e.g. `Memberships(StudentID, ClubID)`, where `StudentID` is a foreign key referencing `Students` and `ClubID` is a foreign key referencing `Clubs`. Each row in `Memberships` represents one specific student-club pairing; a student who joins 3 clubs simply has 3 rows in this table (one per club), and a club with 50 members has 50 rows (one per student) — this avoids needing a variable, unbounded number of foreign-key columns directly in either the `Students` or `Clubs` table, which a simple single foreign key could not accommodate.

Bài tập luyện tập

Explain why a stochastic (rather than deterministic) model would be more appropriate for simulating customer arrivals at a supermarket checkout, compared with modelling a fixed, predictable factory conveyor belt.

Lời giải chi tiết

Customer arrival times at a checkout are inherently **random** and unpredictable in their exact timing — a **stochastic** model, incorporating randomness (e.g. drawing arrival intervals from a probability distribution), can realistically capture this variability and simulate a range of plausible outcomes (busy periods, queue build-up) across repeated runs. A factory conveyor belt moving at a fixed, constant, mechanically-controlled speed behaves predictably and identically every time under the same conditions, making a **deterministic** model (always producing the same output for the same input) an appropriate and sufficient choice for that specific system, since no meaningful randomness needs to be captured.

Bài tập luyện tập

Explain why search engine ranking cannot rely purely on keyword matching (how many times a search term appears on a page), using the concept of link-based authority.

Lời giải chi tiết

Relying purely on keyword frequency is easily abused: a low-quality page could simply repeat a popular search term many times to rank highly, even if its content is irrelevant or unreliable (a manipulation technique sometimes called "keyword stuffing"). **Link-based authority** instead treats being linked to by other pages — especially other important, trusted pages — as a signal of a page's genuine relevance/importance, on the reasoning that other page authors would not link to low-quality content. Combining this with keyword relevance produces far more resistant, higher-quality rankings than keyword frequency alone, since manipulating a page's own keyword content is much easier for a bad actor than genuinely earning many incoming links from other reputable websites.

Bài tập luyện tập

Write an SQL query to delete all students from the `Students` table who belong to `ClassID = 12`, and explain the risk of forgetting the `WHERE` clause.

Lời giải chi tiết

```
DELETE FROM Students
WHERE ClassID = 12;
```

If the `WHERE ClassID = 12` clause were omitted, the statement would become `DELETE FROM Students;`, which deletes **every single record** in the `Students` table, not just those in class 12 — a catastrophic, usually irreversible data-loss error (unless a backup exists), which is why careful review of `WHERE` clauses before executing `DELETE`/`UPDATE` statements is essential practice.

Bài tập luyện tập

Explain why a class implementing an abstract `Shape` class's required `CalculateArea()` method for both a `Circle` and a `Rectangle` subclass is an example of both abstraction and polymorphism working together.

Lời giải chi tiết

Abstraction is shown by the abstract `Shape` class defining *that* every shape must provide a `CalculateArea()` method (a consistent public interface), without specifying *how* — hiding the detail that the actual formula differs completely between shapes. **Polymorphism** is shown when code calls `CalculateArea()` on a list of mixed `Shape` objects (some `Circle`, some `Rectangle`): the identical method call automatically executes each object's own specific formula (πr^2 for a circle, width $\times$ height for a rectangle) without the calling code needing to check or know which specific subclass each object is — the two principles combine to let new shape subclasses be added later without ever having to modify code that already works with the general `Shape` interface.

Bài tập luyện tập

Explain why a NoSQL key-value store would be a poor choice for a bank's core account-balance and transaction-history system, referencing ACID properties.

Lời giải chi tiết

A bank's transactional operations (e.g. transferring money between accounts) require strict **ACID** guarantees — especially **atomicity** (a transfer must be fully completed or fully rolled back, never partially applied) and **consistency** (the database must never end up in an invalid state, such as money appearing from nowhere). Many NoSQL databases deliberately relax some of these strict guarantees in exchange for easier horizontal scalability and higher raw throughput, which is an acceptable trade-off for less critical data (e.g. caching or session tokens) but is generally unacceptable for core financial transaction data, where a relational database's strong ACID guarantees are essential to prevent serious, potentially catastrophic accounting errors.

Bài tập luyện tập

A product catalogue for an online marketplace needs to store wildly different attributes for different product categories (e.g. "screen size" for TVs, "page count" for books, "shoe size" for footwear) without forcing every product record to have the same fixed set of columns. Which type of NoSQL database is best suited to this, and why would a rigid relational table be awkward here?

Lời giải chi tiết

A **document database** is best suited here, since each product's record (document) can hold its own flexible, varying set of fields appropriate to its specific category, without needing to conform to one single, fixed schema shared by every product type. A rigid **relational table** would be awkward because it requires every row to share the exact same fixed set of columns — accommodating every possible product attribute across every category in one table would require an enormous number of mostly-empty (null) columns for any given product, or a complex, harder-to-query set of separate category-specific tables, whereas a document database naturally accommodates this variability record-by-record.

Bài tập luyện tập

Explain how the Singleton pattern would be applied to a program's single, shared application-settings object, and why having *multiple* separate settings objects in different parts of a large program could cause bugs.

Lời giải chi tiết

Using the **Singleton** pattern, the settings class is designed so that only ever *one* instance of it can exist for the entire running program, with every part of the program accessing that same single shared instance rather than creating its own separate copy. If instead *multiple* separate settings objects existed in different parts of a large program, one part of the program could update a setting (e.g. changing the display theme to "dark mode") on *its own* copy, while other parts of the program continued reading from their own, now out-of-sync copies — leading to inconsistent, buggy behaviour where different parts of the application disagree about the current settings, purely because they were never sharing the same single source of truth.

Bài tập luyện tập

Explain why adding an index to every single column of a heavily-written-to table (one with frequent `INSERT`s) is generally a poor idea, despite indexes speeding up `SELECT` queries.

Lời giải chi tiết

Every index added to a table must itself be updated whenever a row is inserted, updated, or deleted, since the index needs to stay accurate and current — so adding many indexes means every single `INSERT` now requires updating not just the table itself but also *every* one of those indexes, substantially slowing down write operations. For a table with frequent writes, indiscriminately indexing every column would impose significant write overhead for relatively little benefit on columns rarely used in `WHERE` clauses or joins — indexes should be added selectively, targeting only the columns that are actually frequently searched/filtered on, balancing read speed against write overhead.

Bài tập luyện tập

A company wants certain staff to see only a `Name` and `Department` column from an `Employees` table (which also contains a confidential `Salary` column), without giving them direct access to the full table. Explain how a view achieves this.

Lời giải chi tiết

A **view** can be created as a saved query such as `SELECT Name, Department FROM Employees;`, and those staff can be granted access only to this *view* rather than to the underlying `Employees` table directly. Querying the view behaves exactly like querying a table containing only `Name` and `Department` — the confidential `Salary` column (and any other columns not included in the view's defining query) is never exposed to users of the view, even though the viewed data is still, behind the scenes, coming from the same single underlying `Employees` table, without duplicating any data.

Bài tập luyện tập

Explain how implementing multiple *interfaces* (e.g. `Flyable` and `Drivable`) avoids the diamond problem that multiple *inheritance* from two full classes could cause, for the same `FlyingCar` example.

Lời giải chi tiết

An **interface** only declares method *signatures* (e.g. `Flyable` declares that any implementing class must provide its own `Fly()` method, and `Drivable` declares that any implementing class must provide its own `Drive()` method) — it contains *no actual implementation itself* to conflict with anything else. `FlyingCar` can implement *both* `Flyable` and `Drivable`, providing its own single, unambiguous implementation of `Fly()` and its own single implementation of `Drive()` — since neither interface supplies a competing pre-built implementation, there is no ambiguity about "which version" to use, unlike inheriting a *already-implemented* `Stop()` method separately from two different full superclasses.

Bài tập luyện tập

Explain one advantage of using a database view over simply giving every application the same raw `SELECT` query text to copy and reuse.

Lời giải chi tiết

A **view** is defined and stored *once* in the database itself; every application or user simply queries the view by name, rather than each one needing to independently write and maintain its own copy of a complex `SELECT` query (potentially involving several joins and conditions). If the underlying logic ever needs to change (e.g. an additional filter condition, or a change to which tables are joined), updating the view's single definition automatically applies everywhere it is used, without needing to track down and separately edit the same query text duplicated across many different applications — reducing both duplication and the risk of inconsistent copies drifting out of sync with each other.

Bài tập luyện tập

A `Shape` superclass defines `CalculateArea()`, and a `Circle` subclass provides its own version of `CalculateArea()` with the identical name and parameters. Separately, a `Printer` class defines two methods both named `Print`: one taking a string, another taking an integer. Identify which scenario is overriding and which is overloading.

Lời giải chi tiết

The `Shape/Circle` scenario is **overriding**: a subclass (`Circle`) redefines a method with the exact same name and parameters as one already defined in its superclass (`Shape`), replacing the inherited behaviour for `Circle` objects specifically. The `Printer` scenario is **overloading**: within the *same* class, two methods share the name `Print` but differ in their parameter types (string vs integer), with the correct version automatically selected based on what type of argument is actually passed at the call site.

Bài tập luyện tập

Explain why overloaded methods must differ in their parameter list (number or type of parameters), not merely in their return type, using an example of why "same parameters, different return type" would be ambiguous.

Lời giải chi tiết

The correct overloaded method is selected based on what the calling code actually *passes in* as arguments — the compiler/interpreter can inspect the arguments at the call site and match them against each overloaded version's parameter list. If two methods had the *exact same* parameter list but only differed in return type (e.g. one `Add(A, B)` returning an integer and another `Add(A, B)` returning a real number), a call like `Add(3, 5)` would give the system no way to determine which version was intended, since the return type is not knowable from the call itself before the method actually executes — this is why virtually all languages require overloaded methods to differ in their *parameters*, not merely their return type.

Bài tập luyện tập

Explain one advantage of a three-tier (presentation / application / data) architecture over a simple two-tier design where the client application talks directly to the database.

Lời giải chi tiết

A three-tier architecture inserts an **application/logic tier** between the presentation tier and the database, centralising business rules, validation, and security checks in one place that every client request must pass through — this makes it far easier to enforce consistent rules across many different types of clients (web, mobile, desktop) without duplicating that logic in each one, and reduces the security exposure of allowing every client direct access to the database itself. A two-tier design, connecting clients directly to the database, would require duplicating business logic and security checks into every separate client application, increasing both the risk of inconsistency and the attack surface exposed directly to end-user devices.

Bài tập luyện tập

A `LibraryBook` class inherits from a `Media` superclass, and a `DVD` class also inherits from `Media`. Both override a method `GetLoanPeriod()` to return different values. Explain what OOP principle allows a single loop over a mixed list of `LibraryBook` and `DVD` objects to correctly call `GetLoanPeriod()` on each without checking each object's specific type.

Lời giải chi tiết

This is **polymorphism**: since both `LibraryBook` and `DVD` share the common superclass `Media` and each overrides `GetLoanPeriod()` with their own specific implementation, a single loop calling `Item.GetLoanPeriod()` on each object automatically invokes whichever version is appropriate for that object's actual class — the calling code does not need an `if/else if` chain checking "is this a LibraryBook or a DVD?" before deciding which method to call; the correct behaviour is resolved automatically based on each object's real class.

Bài tập luyện tập

A `Payments` table has columns `PaymentID` (primary key), `OrderID` (foreign key), `Amount`, and `Currency`. Write an SQL query to find the total `Amount` of payments for each `Currency`, ordered from highest total to lowest.

Lời giải chi tiết

```
SELECT Currency, SUM(Amount) AS TotalAmount
FROM Payments
GROUP BY Currency
ORDER BY TotalAmount DESC;
```

`GROUP BY Currency` groups all payments by their currency before `SUM(Amount)` totals each group; `ORDER BY TotalAmount DESC` then sorts the resulting currency totals from highest to lowest.

Bài tập luyện tập

A `Movies(MovieID, Title, DirectorName, DirectorBirthYear)` table stores the director's birth year alongside every movie by that director. Identify the normalization violation and redesign the table(s).

Lời giải chi tiết

`DirectorBirthYear` depends on `DirectorName` (itself a non-key field), not directly on the primary key `MovieID` — a **transitive dependency**, violating **3NF**. **Redesign:** split into `Directors(DirectorID, DirectorName, DirectorBirthYear)` and `Movies(MovieID, Title, DirectorID)`, with `DirectorID` in `Movies` as a foreign key referencing `Directors` — each director's birth year is now stored exactly once, regardless of how many movies they have directed.

Bài tập luyện tập

Explain why a class marking all of its attributes `public` (directly accessible, with no getter/setter methods) undermines the benefit of encapsulation, even if the class still technically groups data and methods together.

Lời giải chi tiết

Encapsulation's real benefit comes from *restricting* direct access to an object's internal data, forcing all interaction through controlled methods that can validate input and hide internal implementation details. A class with every attribute marked `public` still technically bundles data and methods in one class, but external code can freely read *and overwrite* its attributes

directly, bypassing any validation logic entirely (e.g. setting a `Speed` attribute to an invalid negative value with no check) and becoming tightly coupled to the exact internal attribute names/structure — losing both the data-integrity and information-hiding benefits that make encapsulation valuable in the first place.

Đồng hành cùng 8020 English

Cảm ơn bạn đã học cùng bộ giáo trình quốc tế 8020. **Điểm thật · Thầy thật · Kết quả thật** — nhiều học viên chỉ cần 1–2 khoá để đạt kết quả mong muốn.

Chương trình đào tạo tại 8020 English

IELTS Academic/General · **SAT** · PTE · Duolingo · TOEFL | **AP** (College Board) · **IB** Diploma · **Cambridge** IGCSE / A-Level | săn học bổng du học.

Vì sao chọn 8020 English?

- **100% giáo viên** đạt tối thiểu 8.0 IELTS hoặc 1500 SAT — đội ngũ Giáo sư · Tiến sĩ · Thạc sĩ tốt nghiệp trường top đầu thế giới (Carnegie Mellon — #1 Hoa Kỳ về Khoa học Máy tính).
- **Kèm 1–1** mọi độ tuổi; lớp sĩ số nhỏ 2–5 bạn (đa số dưới 10 bạn/lớp).
- Lộ trình cá nhân hoá, theo sát tiến độ từng học viên; lớp OFFLINE tại TP.HCM & ONLINE toàn quốc.

Bảng vàng học viên

AP 5/5 — Calculus AB/BC	**AP** 4–5 — Biology, Chemistry	**IB** 90/100 — Economics
SAT 1590 / 1570 / 1550	**IELTS** 8.0–9.0	**Duolingo** 110 · **PTE** 90

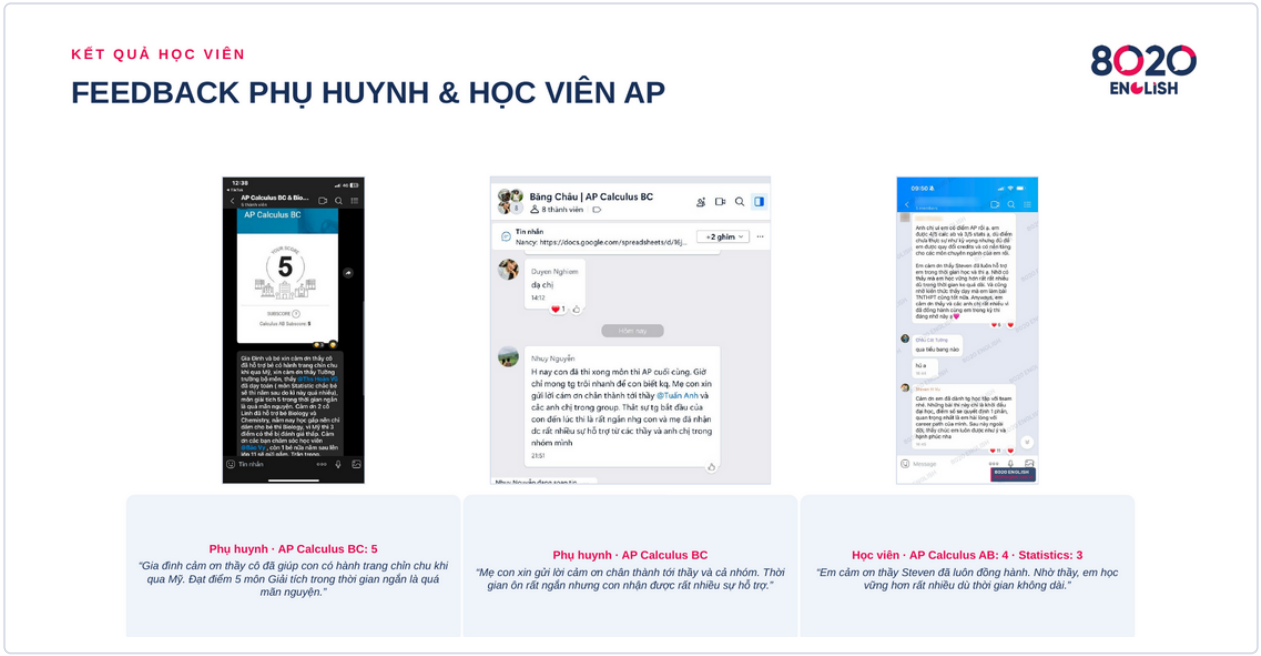
Cảm nhận học viên & phụ huynh

> *"Bạn đã cải thiện được tư duy Writing — kỹ năng từng là nỗi sợ lớn nhất — và cuối cùng đã đậu vào trường top như mong muốn. Thái độ học tập cực kỳ nghiêm túc; ngay cả khi nằm viện vẫn cố gắng học đều đặn."*
>
> — Phan Thị Thanh Hằng · IELTS Overall 8.5

> *"Cần tất cả kỹ năng trên 7.5 để xét vào trường top 1 Anh Quốc về Y học (chương trình UKFPO của NHS) — và đã đạt mục tiêu sau khi học Writing 1–1."*
>
> — Trần Hoàng Bảo Ngọc · IELTS Overall 8.0 (Writing 6.0 → 7.5)

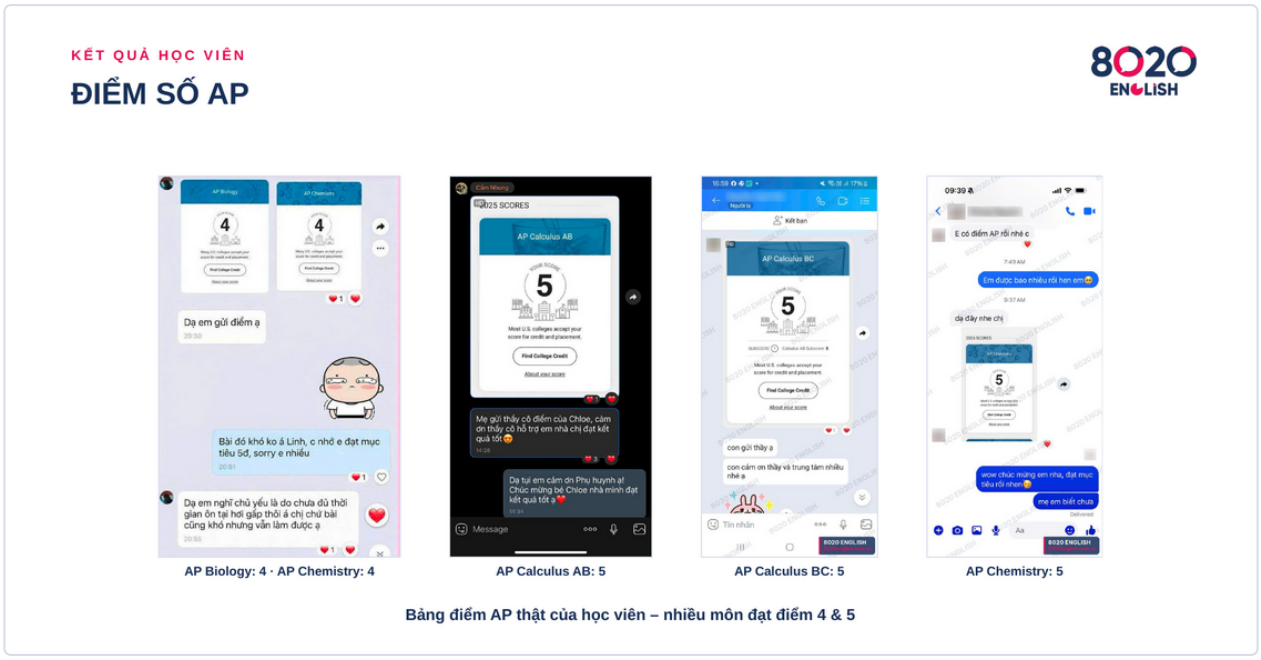
> *"Gia đình và bé xin cảm ơn thầy cô đã giúp con có hành trang chỉn chu khi qua Mỹ. Đạt điểm 5 môn Giải tích trong thời gian ngắn là quá mãn nguyện."*
>
> — Phụ huynh · AP Calculus BC 5/5

> *"Em cảm ơn thầy đã luôn đồng hành. Nhờ thầy, em học vững hơn rất nhiều dù thời gian ôn không dài."*
>
> — Học viên · AP Calculus AB 4

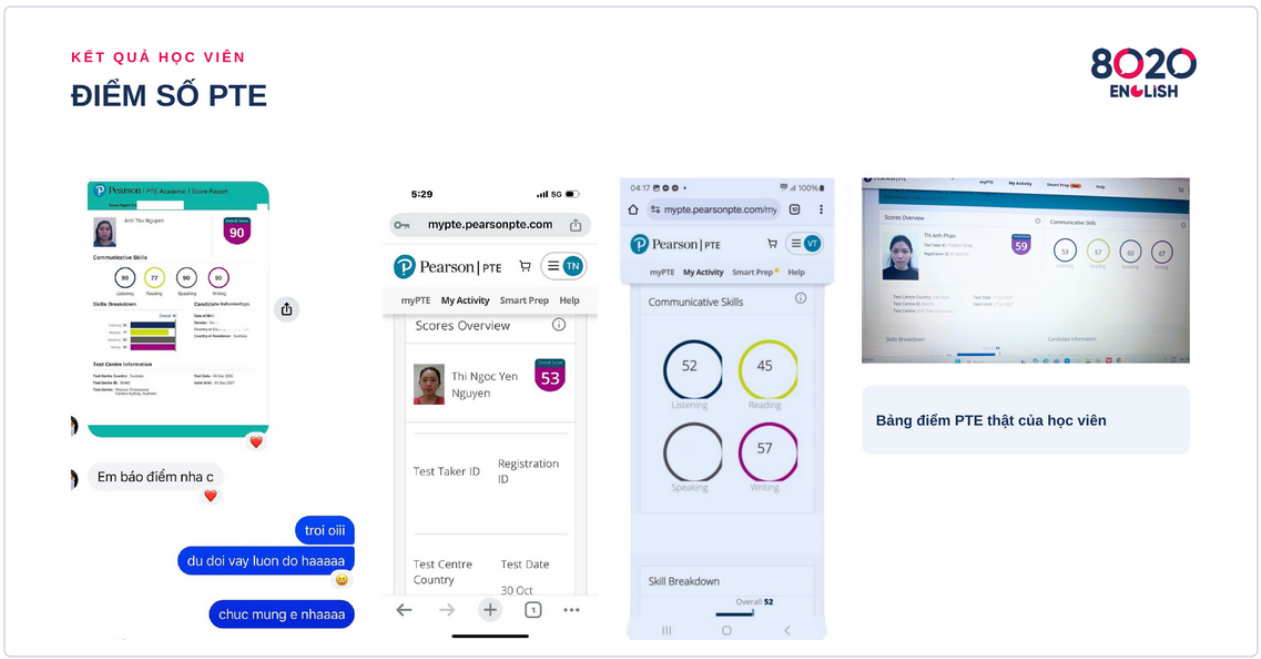


Feedback phụ huynh & học viên AP — nhiều môn đạt điểm 4 & 5

Điểm thi thật của học viên



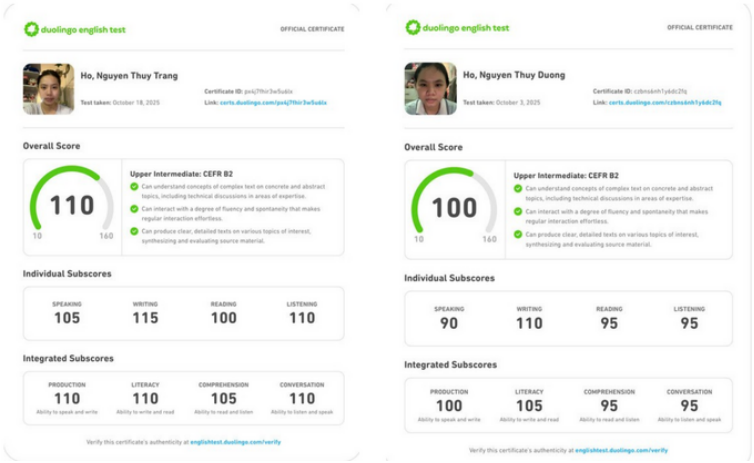
Bảng điểm AP thật — Calculus AB/BC 5/5 · Biology & Chemistry 4–5 (College Board)



Học viên 8020 — PTE Academic 90

KẾT QUẢ HỌC VIÊN

ĐIỂM SỐ DUOLINGO



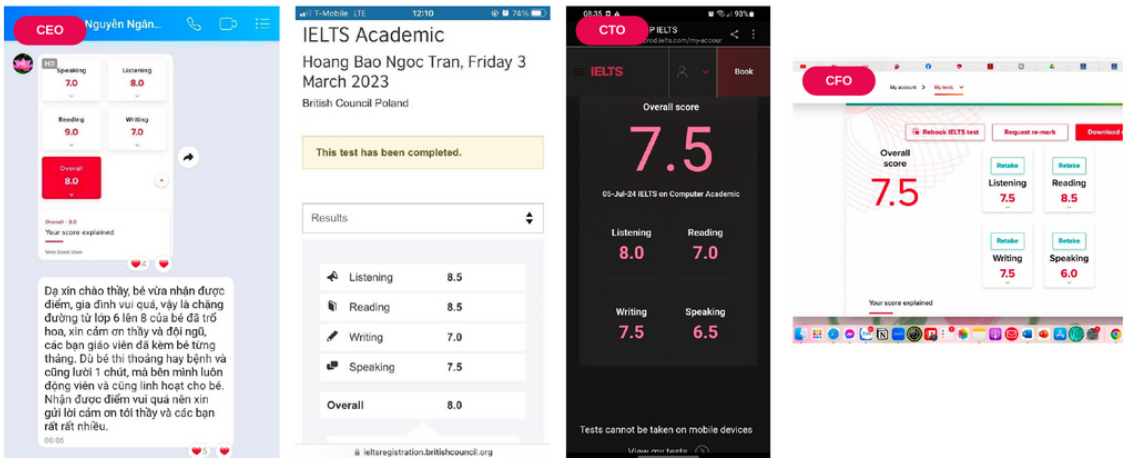
Duolingo English Test – 110 & 100

Học viên 8020 — Duolingo English Test 110 & 100

KẾT QUẢ HỌC VIÊN

THÀNH TÍCH HỌC VIÊN

Bảng điểm thi thật – chất lượng được giám sát nghiêm ngặt. Một số học viên chỉ cần 1–2 khóa để đạt kết quả mong muốn.



Học viên 8020 — IELTS 8.0 / 7.5 / 8.5 (báo điểm thật)