Data Structures & Algorithms

Personal Notes for Labs (TA Work)

Henry Baker

2025

Hertie School

This document provides a self-contained treatment of data structures and algorithms: computational complexity, common data structures, algorithm analysis, divide-and-conquer, dynamic programming, graph algorithms, greedy algorithms, and intractability.

Contents

DSA Lecture Notes: Week 1
Introduction to Scrum and Agile Development

This introductory week establishes a foundational metaphor for understanding complex, adaptive systems-both biological and organisational. We begin with W. Grey Walter's pioneering work on cybernetic tortoises, which demonstrated how simple feedback loops can produce surprisingly sophisticated emergent behaviour. This concept directly informs our understanding of Scrum: a framework that harnesses complexity through iterative feedback rather than attempting to control it through rigid, top-down planning.

1 Emergent Behaviour: Grey Walter's Tortoises

W. Grey Walter (1910–1977) was a British neurophysiologist and roboticist whose work profoundly influenced both neuroscience and artificial intelligence. His 1953 book *The Living Brain* documented experiments with simple autonomous robots that exhibited remarkably complex behaviour.

1.1 The Machina Speculatrix

In the early 1950s, Walter constructed a series of three-wheeled tortoise-shaped robots he called *machina speculatrix* (literally "the machine that watches"). These robots-nicknamed Elmer and Elsie-were groundbreaking in their simplicity and the complexity of behaviour they produced.

Machina Speculatrix Design

Each tortoise contained only two sensory inputs and two motors:

- **Photocell**: detected light intensity
- **Bump sensor**: detected physical contact
- **Steering motor**: controlled direction
- **Drive motor**: controlled forward/backward movement

The control logic was equally minimal: seek moderate light (avoiding both darkness and bright light), and respond to obstacles by backing up and turning. No explicit programming of "behaviour" was included.

Despite this minimal design, the tortoises exhibited behaviours that appeared purposeful and even lifelike:

- **Exploration**: wandering through the environment, apparently "curious"
- **Phototaxis**: moving toward light sources, then circling around them

- **Obstacle avoidance**: navigating around barriers without explicit mapping
- **Self-recognition**: when placed in front of a mirror with a light on their "nose", they would exhibit a distinctive "dance" as they responded to their own reflection
- **Social behaviour**: when multiple tortoises interacted, they appeared to "recognise" each other and engage in complex group dynamics

Emergent Behaviour

Emergent behaviour occurs when a system exhibits properties or patterns that are not explicitly programmed or present in any individual component, but arise from the interactions between components and their environment.

Walter's tortoises demonstrated that complex, apparently intelligent behaviour can emerge from simple rules combined with environmental feedback-no central "controller" or explicit behavioural programming required.

1.2 Implications for Complex Systems

Walter's work established several principles that remain relevant to understanding complex adaptive systems:

1. **Feedback over planning**: The tortoises had no "plan" for how to behave; their behaviour emerged from continuous feedback loops with the environment.
2. **Simple rules, complex outcomes**: Minimal components and simple interaction rules can produce sophisticated, adaptive behaviour.
3. **Environment as computation**: Much of the "intelligence" in the system was distributed between the robot and its environment, not located solely in the robot's "brain".
4. **Robustness through adaptation**: The tortoises could handle novel situations not because they had been programmed for every eventuality, but because their feedback mechanisms allowed continuous adaptation.

These principles directly inform our understanding of effective software development methodologies. Just as Walter's tortoises achieved complex behaviour through simple feedback mechanisms rather than elaborate pre-programming, modern agile methodologies achieve project success through iterative feedback rather than comprehensive upfront planning.

2 Scrum: Harnessing Complexity

Scrum is a lightweight framework for managing complex work, particularly software development. Rather than attempting to predict and plan every aspect of a project upfront, Scrum embraces uncertainty and uses rapid feedback cycles to adapt continuously.

"This concept of a harness to help coordinate independent processors via feedback loops, while having the feedback be reality-based from real data coming from the environment is central to human groups achieving higher level behavior than any individual can achieve on their own."

This quote captures the essence of Scrum's philosophy: like Walter's tortoises, a Scrum team achieves outcomes beyond what any individual member could accomplish alone, not through rigid command-and-control hierarchies, but through structured feedback mechanisms that enable emergent coordination.

2.1 Historical Context

The term "Scrum" comes from rugby, where it refers to the method of restarting play after an infringement-a tight formation where the team works together to move the ball forward. The metaphor emphasises teamwork, adaptability, and collective effort toward a shared goal.

Scrum was formalised in the mid-1990s by Jeff Sutherland and Ken Schwaber, drawing on earlier work in lean manufacturing (particularly the Toyota Production System), empirical process control theory, and iterative development practices. The framework was designed to address the chronic problems of traditional "waterfall" development:

- Projects delivered late and over budget
- Final products that didn't meet actual user needs
- Inability to respond to changing requirements
- Developer burnout and disengagement
- Integration nightmares when combining work from multiple teams

2.2 Scrum Principles

Core Scrum Principles

- **Inspect and adapt**: Continuously examine outcomes and adjust processes accordingly
- **Embrace uncertainty**: Accept that surprises will occur (and that this is valuable information, not failure)
- **Short development cycles**: Keep iterations brief (typically 1–4 weeks) to enable rapid feedback
- **Small, achievable goals**: Break work into discrete, completable chunks rather than monolithic deliverables
- **Frequent re-evaluation**: Regularly reassess priorities based on new information
- **Regular check-ins**: Maintain team alignment through structured communication rituals
- **Transparency**: Make work visible to all stakeholders
- **Self-organisation**: Teams determine how to accomplish work, not managers

Common Misconception

Scrum is not "no planning"-it is *continuous planning*. Traditional waterfall approaches front-load all planning before development begins. Scrum distributes planning throughout the project, with detailed planning happening just before work begins (when information is freshest) and high-level planning being continuously refined based on empirical evidence.

3 The Agile Manifesto

Scrum exists within the broader context of the Agile movement. In February 2001, seventeen software developers met at a ski resort in Utah and produced the *Manifesto for Agile Software Development*, a brief document that articulated shared values underlying their various methodologies.

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	over	processes and tools
Working software	over	comprehensive documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Note the careful phrasing: the manifesto does not say that processes, documentation, contracts, and plans are worthless. It establishes a hierarchy of values-when trade-offs must be made, prefer the items on the left.

3.1 The Twelve Principles

Behind the manifesto lie twelve principles that elaborate on these values:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity-the art of maximising the amount of work not done-is essential.
11. The best architectures, requirements, and designs emerge from self-organising teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Agile vs Scrum

Agile is a philosophy and set of values about how software should be developed.

Scrum is a specific framework that implements agile principles through defined roles, events, and artefacts.

Other agile methodologies include Kanban, Extreme Programming (XP), Crystal, and Feature-Driven Development. Scrum is the most widely adopted, but it is not synonymous with Agile.

4 The Scrum Framework

Scrum provides a minimal but complete framework for managing iterative development. It consists of three categories of elements: roles (who), events (when), and artefacts (what).

4.1 Scrum Roles

A Scrum team consists of exactly three roles. This constraint is intentional-it prevents the proliferation of specialised roles that can create bottlenecks and diffuse accountability.

4.1.1 Product Owner

The Product Owner is responsible for maximising the value of the product and the work of the Development Team. Key responsibilities include:

- **Managing the Product Backlog**: Maintaining the prioritised list of features, enhancements, and fixes
- **Expressing backlog items clearly**: Ensuring the team understands what needs to be built and why
- **Prioritising work**: Ordering the backlog to optimise value delivery
- **Stakeholder communication**: Serving as the primary interface between the team and external stakeholders
- **Acceptance decisions**: Determining whether work meets the definition of "done"

The Product Owner must be a single individual (not a committee) with sufficient authority to make decisions about the product. They represent the voice of the customer and the business.

4.1.2 Scrum Master

The Scrum Master is responsible for ensuring Scrum is understood and enacted. They serve the team by:

- **Facilitating Scrum events**: Ensuring meetings happen and are productive
- **Removing impediments**: Clearing obstacles that slow the team's progress
- **Coaching the team**: Helping team members understand and apply Scrum effectively
- **Protecting the team**: Shielding the team from external interruptions and scope creep during sprints
- **Promoting self-organisation**: Helping the team develop its ability to manage itself

The Scrum Master is explicitly *not* a traditional project manager. They have no authority to assign tasks or make decisions for the team. Their role is servant-leadership: enabling the team's success rather than directing their work.

4.1.3 Development Team

The Development Team consists of professionals who do the work of delivering a potentially releasable increment of product at the end of each Sprint. Characteristics include:

- **Self-organising**: The team decides how to accomplish work; no one tells them how to turn backlog items into increments of functionality
- **Cross-functional**: The team collectively possesses all skills needed to create the product increment
- **No titles**: Regardless of individual specialisations, all members are "Developers"
- **No sub-teams**: Work is not siloed; the whole team is accountable for the whole increment
- **Optimal size**: Typically 3–9 people (large enough for diverse skills, small enough for effective coordination)

Team Accountability

In Scrum, accountability operates at the team level, not the individual level. If the team fails to deliver the sprint goal, the *team* failed-not any individual member. This encourages collaboration over hero culture and creates psychological safety for experimentation and learning.

4.2 Scrum Events

Scrum prescribes five events (sometimes called "ceremonies"). Each creates a formal opportunity for inspection and adaptation. All events are time-boxed-they have a maximum duration that cannot be extended.

4.2.1 The Sprint

The Sprint is the heartbeat of Scrum-a time-boxed iteration (typically 1–4 weeks, with 2 weeks being most common) during which a "Done", usable, potentially releasable product increment is created.

Sprint Characteristics

- **Fixed duration**: Sprints are always the same length; consistency aids planning and measurement
- **Immutable goal**: Once the sprint begins, the Sprint Goal cannot be changed
- **Protected scope**: No changes that endanger the Sprint Goal are allowed during the sprint
- **Continuous**: A new sprint starts immediately after the previous one ends
- **Cancellable**: Only the Product Owner can cancel a sprint, and only if the Sprint Goal becomes obsolete

4.2.2 Sprint Planning

Sprint Planning initiates the sprint by defining what can be delivered and how. The entire Scrum team collaborates. For a two-week sprint, this meeting is time-boxed to four hours maximum.

The meeting addresses two questions:

1. **What can be done this sprint?** The Product Owner presents the highest-priority backlog items. The Development Team forecasts how much they can accomplish based on their past performance (velocity) and current capacity.
2. **How will the work get done?** The Development Team decomposes selected backlog items into tasks, typically of one day or less. This creates the Sprint Backlog.

By the end of Sprint Planning, the team has a Sprint Goal-a coherent objective that provides guidance on why the increment is being built.

4.2.3 Daily Scrum

The Daily Scrum (often called "standup" because participants traditionally stand to keep the meeting brief) is a 15-minute time-boxed event for the Development Team to synchronise and plan the next 24 hours.

Each team member typically addresses three questions:

1. What did I accomplish yesterday toward the Sprint Goal?
2. What will I work on today toward the Sprint Goal?
3. Are there any impediments blocking my progress?

Daily Scrum Anti-patterns

The Daily Scrum is *not*:

- A status report to management
- A problem-solving session (those happen after the standup)
- An opportunity to assign or reassign tasks
- A meeting that requires the Scrum Master or Product Owner to run (the Development Team owns it)

If the Daily Scrum regularly runs over 15 minutes, or if team members are speaking to the Scrum Master rather than to each other, these are signals that the meeting has drifted from its purpose.

4.2.4 Sprint Review

The Sprint Review is held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed. The team demonstrates what was accomplished, and stakeholders provide feedback. For a two-week sprint, this is time-boxed to two hours.

Key activities include:

- The Product Owner explains what has been "Done" and what has not
- The Development Team demonstrates the work and answers questions
- The Product Owner discusses the current Product Backlog state
- The group collaborates on what to do next
- The meeting results in a revised Product Backlog for the next Sprint

4.2.5 Sprint Retrospective

The Sprint Retrospective is an opportunity for the Scrum Team to inspect itself and create a plan for improvements. It occurs after the Sprint Review and before the next Sprint Planning. For a two-week sprint, this is time-boxed to 90 minutes.

The team discusses:

1. What went well during the Sprint?
2. What could be improved?
3. What will we commit to improving in the next Sprint?

Continuous Improvement

The Retrospective is where the "inspect and adapt" principle applies to the team's own processes, not just the product. This institutionalised reflection is one of Scrum's most powerful features-it builds continuous improvement into the team's operating rhythm.

Teams that skip or rush retrospectives often find their velocity plateaus. The retrospective is an investment in the team's future effectiveness.

4.3 Scrum Artefacts

Scrum's artefacts represent work or value. They are designed to maximise transparency of key information.

4.3.1 Product Backlog

The Product Backlog is an ordered list of everything that is known to be needed in the product. It is the single source of requirements for any changes to be made to the product.

Characteristics:

- **Dynamic**: The backlog constantly evolves as the product and its environment change
- **Never complete**: As long as a product exists, its backlog exists
- **Ordered by value**: Higher-priority items are typically more detailed and refined
- **Owned by Product Owner**: The Product Owner is responsible for content, availability, and ordering

Product Backlog items typically include features, functions, requirements, enhancements, and fixes. Each item has a description, order, estimate, and value. Higher-order items are clearer and more detailed; lower-order items may be deliberately vague until they move up the backlog.

4.3.2 Sprint Backlog

The Sprint Backlog is the set of Product Backlog items selected for the Sprint, plus a plan for delivering them and achieving the Sprint Goal. It makes visible all the work the Development Team identifies as necessary to meet the Sprint Goal.

The Sprint Backlog is a forecast by the Development Team about what functionality will be in the next Increment and the work needed to deliver that functionality. It is owned entirely by the Development Team and is updated throughout the Sprint as work is completed and new work is discovered.

4.3.3 Product Increment

The Increment is the sum of all Product Backlog items completed during a Sprint and all previous Sprints. At the end of a Sprint, the new Increment must be "Done"-meaning it is in a usable condition and meets the team's Definition of Done.

Definition of Done

The Definition of Done (DoD) is a shared understanding of what it means for work to be complete. It typically includes criteria such as:

- Code is written and reviewed
- Unit tests pass
- Integration tests pass
- Documentation is updated
- Code is merged to the main branch
- Product Owner has accepted the work

The DoD creates transparency by ensuring everyone has a shared understanding of "complete". Without an explicit DoD, "done" becomes subjective and technical debt accumulates.

5 Why This Matters for Data Structures and Algorithms

You might wonder why a course on data structures and algorithms begins with software development methodology. The connection is threefold:

5.1 Algorithms Exist in Context

Algorithms do not exist in a vacuum. They are implemented within software systems, developed by teams, and deployed to solve real problems. Understanding how professional software is developed helps you:

- Write code that others can understand and maintain
- Break complex algorithmic problems into manageable pieces
- Estimate how long implementation will take
- Collaborate effectively with other developers

5.2 Iterative Problem-Solving

The "inspect and adapt" philosophy of Scrum mirrors effective algorithmic problem-solving:

1. Start with a simple, potentially inefficient solution
2. Test it against your understanding of the problem
3. Identify bottlenecks and areas for improvement
4. Refine the solution based on empirical evidence

5. Repeat until the solution meets requirements

This iterative approach-rather than attempting to design a perfect algorithm from scratch-is how experienced practitioners actually solve algorithmic problems.

5.3 Managing Complexity

Both Scrum and algorithm design are fundamentally about managing complexity:

- **Decomposition**: Breaking complex problems into smaller, tractable sub-problems
- **Abstraction**: Hiding unnecessary detail behind clean interfaces
- **Empiricism**: Making decisions based on measured evidence rather than speculation
- **Feedback loops**: Using results to inform future decisions

Grey Walter's tortoises achieved complex behaviour through simple feedback mechanisms. Scrum teams achieve complex software through structured iteration. And you will achieve elegant algorithmic solutions through the same principles: break the problem down, implement something, measure it, and refine.

The Unifying Theme

Whether designing autonomous robots, managing software projects, or implementing algorithms, the same insight applies: **complex adaptive behaviour emerges from simple rules combined with feedback from reality**.

This course will teach you the vocabulary and techniques for analysing algorithms. But the meta-skill-iterative refinement through empirical feedback-transcends any particular algorithmic technique.

6 Introduction to Computing

Understanding how computers represent and manipulate data at the lowest level is fundamental to data structures and algorithms. The efficiency of algorithms often depends critically on how data is stored and accessed in memory, how arithmetic operations are performed at the hardware level, and what the inherent limitations of numerical representation are. This week, we build the foundation upon which all subsequent algorithmic analysis rests.

7 What is a Computer?

At its core, a computer is a machine that manipulates symbols according to a set of rules. These symbols are represented as bits-the fundamental unit of information in computing.

7.1 Bits: The Fundamental Unit

Definition: Bit

A **bit** (binary digit) is the smallest unit of data in computing, representing exactly one of two possible states. We denote these states as:

- Logical: `TRUE` / `FALSE`
- Numerical: `1` / `0`
- Physical: `HIGH VOLTAGE` / `LOW VOLTAGE`, `ON` / `OFF`

All of these representations are isomorphic-they encode exactly the same information.

The power of bits lies in their simplicity and reliability. A continuous signal is inherently susceptible to noise and degradation, but a discrete binary signal only needs to distinguish between two states. This makes digital systems remarkably robust: small variations in voltage do not change the interpretation of a bit.

Why Bits Matter for Algorithms

Every data structure you will encounter-arrays, linked lists, trees, hash tables-ultimately reduces to patterns of bits in memory. Understanding bit-level representation helps you:

- Analyse the true memory footprint of data structures
- Understand overflow and precision errors in numerical algorithms
- Design efficient bit manipulation algorithms (e.g., bit vectors, bloom filters)
- Reason about cache efficiency and memory access patterns

7.2 Logic Gates

Logic gates are the physical building blocks that implement Boolean operations on bits. They are constructed from transistors and form the foundation of all digital circuits.

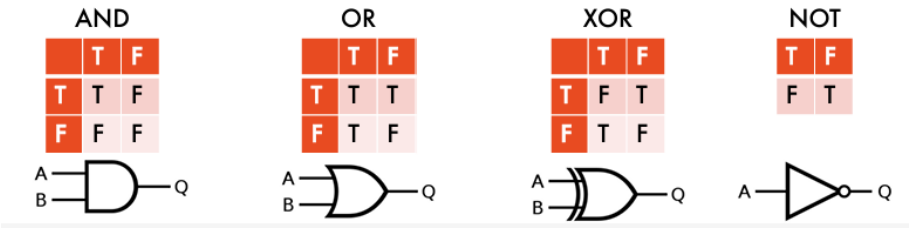

Figure 1: Truth tables for fundamental logic gates. A useful mnemonic: read "A is 1, *[gate name]* B is 1" to determine when the output is 1. For AND: "A is 1 *and* B is 1"; for OR: "A is 1 *or* B is 1".

The fundamental gates are:

- **AND gate**: Output is 1 only when both inputs are 1. Denoted $A \wedge B$ or $A \cdot B$.
- **OR gate**: Output is 1 when at least one input is 1. Denoted $A \vee B$ or $A + B$.
- **NOT gate** (inverter): Output is the opposite of the input. Denoted $\neg A$ or $\overline{A}$.
- **XOR gate** (exclusive or): Output is 1 when inputs are different. Denoted $A \oplus B$.

Boolean Algebra Identities

These identities are useful for simplifying logical expressions and understanding circuit design:

$A \wedge 0 = 0$	$A \vee 0 = A$	(Identity)
$A \wedge 1 = A$	$A \vee 1 = 1$	(Identity)
$A \wedge A = A$	$A \vee A = A$	(Idempotence)
$A \wedge \neg A = 0$	$A \vee \neg A = 1$	(Complement)
$\neg(\neg A) = A$		(Double negation)
$\neg(A \wedge B) = \neg A \vee \neg B$	$\neg(A \vee B) = \neg A \wedge \neg B$	(De Morgan's laws)

7.2.1 The NAND Gate: A Universal Building Block

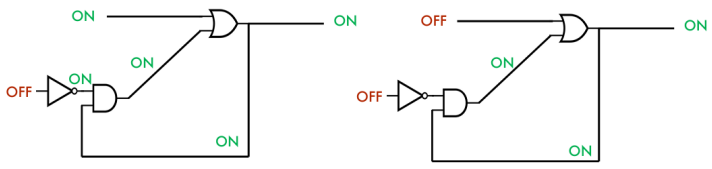

Figure 2: The NAND gate symbol and truth table. NAND outputs 0 only when both inputs are 1-it is the negation of AND.

The NAND gate ("NOT AND") is particularly important because it is functionally complete: any Boolean function can be implemented using only NAND gates. This is why NAND gates are the most commonly used gates in integrated circuit design.

Functional Completeness of NAND

A set of logic gates is **functionally complete** if any Boolean function can be expressed using only gates from that set. NAND is functionally complete because:

- NOT: $\neg A = A \text{ NAND } A$
- AND: $A \wedge B = (A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$
- OR: $A \vee B = (A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)$

Since NOT, AND, and OR can express any Boolean function, so can NAND alone.

8 Hexadecimal Notation

While binary is the native language of computers, it is cumbersome for humans. Hexadecimal (base-16) provides a compact representation that maps cleanly to binary.

Definition: Hexadecimal

The **hexadecimal** (hex) number system is a positional numeral system with base 16. The digit set is:

$$\mathcal{H} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

where $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, $F = 15$ in decimal.

Hexadecimal numbers are typically prefixed with `0x` (e.g., `0x3A`) to distinguish them from decimal.

Why Hexadecimal?

Since $16 = 2^4$, each hexadecimal digit corresponds to exactly 4 binary digits (bits). This makes conversion between hex and binary trivial, while being four times more compact than binary representation.

8.1 Conversion Reference

Hex	Decimal	Binary	Hex	Decimal	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

8.2 Conversion Procedures

8.2.1 Hexadecimal to Binary

Replace each hex digit with its 4-bit binary equivalent:

Example: Convert `0x3A` to binary.

- `3` $\rightarrow 0011_2$
- `A` $\rightarrow 1010_2$
- Combined: `0x3A` $= 0011\,1010_2$

8.2.2 Binary to Hexadecimal

Group binary digits into sets of four from the right, padding with leading zeros if necessary:

Example: Convert 10111011_2 to hexadecimal.

- Group: $1011\,1011$
- $1011_2 =$ `B`
- $1011_2 =$ `B`
- Combined: $10111011_2 =$ `0xBB`

8.2.3 Hexadecimal to Decimal

Expand using positional notation:

Example: Convert `0x3A` to decimal.

$$\texttt{0x3A} = 3 \times 16^1 + 10 \times 16^0 = 48 + 10 = 58_{10}$$

9 Memory

Memory is where computers store information. Understanding memory organisation is crucial for analysing the space complexity of algorithms and understanding how data structures are laid out.

9.1 Flip-Flops: The Basis of Memory

Definition: Flip-Flop

A **flip-flop** is a bistable electronic circuit that can store one bit of information. It maintains its state (0 or 1) until explicitly changed by an input signal. Flip-flops are constructed from logic gates arranged with feedback loops.

Memory can be conceptualised as an organised collection of flip-flops:

- A **row of flip-flops** forms a sequence of bits: $[\text{bit}_0, \text{bit}_1, \text{bit}_2, \ldots, \text{bit}_{n-1}]$
- Each bit or group of bits has a unique **address**-a numerical identifier used to locate it
- Memory is organised hierarchically: bits $\rightarrow$ bytes $\rightarrow$ words $\rightarrow$ pages

9.2 Random Access Memory (RAM)

Random Access

Random Access Memory (RAM) allows any memory location to be accessed directly in constant time $O(1)$, regardless of its address. This is in contrast to **sequential access** memory (like magnetic tape), where accessing an element requires traversing all preceding elements.

The "random" in RAM refers to the ability to access any location equally quickly-not to randomness in the statistical sense.

This constant-time access property is fundamental to the performance of array-based data structures and is assumed in most algorithm analysis.

9.3 Memory Units and Addressing

Definition: Byte

A **byte** is a unit of digital information consisting of 8 bits:

$$1 \text{ byte} = 8 \text{ bits}$$

The byte is the standard addressable unit in most modern computer architectures.

KB vs kB vs Kb: A Common Source of Confusion

Memory units have two competing conventions:

- **Binary prefixes** (IEC standard): 1 KiB (kibibyte) $= 2^{10} = 1024$ bytes
- **SI prefixes**: 1 kB (kilobyte) $= 10^3 = 1000$ bytes

Additionally, note the case distinction:

- Uppercase B = bytes (e.g., KB, MB, GB)
- Lowercase b = bits (e.g., kb, Mb, Gb)

Network speeds are typically measured in bits per second (e.g., 100 Mbps), while storage is measured in bytes (e.g., 500 GB). A "100 Mbps" connection transfers at most $100/8 = 12.5$ MB per second.

Unit	Binary (Traditional)	SI (Decimal)
Kilobyte	$2^{10} = 1024$ bytes	$10^3 = 1000$ bytes
Megabyte	$2^{20} = 1{,}048{,}576$ bytes	$10^6 = 1{,}000{,}000$ bytes
Gigabyte	$2^{30} \approx 1.07 \times 10^9$ bytes	10^9 bytes

9.4 Memory Addressing

In a byte-addressable machine, each byte has a unique address. For a memory with 2^n addressable locations, we need n bits for addresses.

```
Address:    0x00  0x01  0x02  0x03  0x04  0x05  0x06  0x07
          +---+---+---+---+---+---+---+---+
Data:     |byte0|byte1|byte2|byte3|byte4|byte5|byte6|byte7|
          +---+---+---+---+---+---+---+---+
```

Example: In a 1 KB memory:

- Total bytes: 1024
- Address range: 0 to 1023 (or `0x000` to `0x3FF`)
- Bits needed for addressing: $\lceil \log_2 1024 \rceil = 10$ bits

To access the 10th byte (address 10), we retrieve bits 80–87 (since byte 10 contains bits $10 \times 8 = 80$ through $10 \times 8 + 7 = 87$).

10 Data Representation

All data in a computer-text, numbers, images, programs-must ultimately be represented as sequences of bits. This section examines how different types of data are encoded.

10.1 Text Representation

10.1.1 ASCII (American Standard Code for Information Interchange)

Definition: ASCII

ASCII is a character encoding standard that assigns numerical codes to 128 characters:

- **Bit length**: 7 bits per character (often stored in 8 bits with a leading zero)
- **Range**: 0–127
- **Coverage**: English letters (A–Z, a–z), digits (0–9), punctuation, and control characters

Examples:

- `'A'` = 65 = 01000001_2 = `0x41`

- `'a'` = 97 = 01100001_2 = `0x61`
- `'0'` = 48 = 00110000_2 = `0x30`

Notice that uppercase and lowercase letters differ by exactly 32 (2^5), which means converting case requires only flipping bit 5.

10.1.2 Unicode

Definition: Unicode

Unicode is a universal character encoding standard that aims to represent every character from every writing system. Key points:

- **Code points**: Over 140,000 characters across 150+ scripts
- **Notation**: Code points written as U+XXXX (e.g., U+0041 for 'A')
- **Encodings**:
  - **UTF-8**: Variable length (1–4 bytes), backwards compatible with ASCII
  - **UTF-16**: Variable length (2 or 4 bytes)
  - **UTF-32**: Fixed length (4 bytes per character)

UTF-8: The Web's Encoding

UTF-8 is the dominant encoding on the web because:

- ASCII text is valid UTF-8 (backwards compatibility)
- Common characters use fewer bytes (efficient for English)
- Self-synchronising: you can find character boundaries by examining any byte

10.2 Integer Representation

10.2.1 Unsigned Binary Integers

The simplest integer representation interprets a sequence of bits as a base-2 number.

Definition: Unsigned Integer

An **unsigned n-bit integer** interprets a bit sequence $b_{n-1}b_{n-2}\ldots b_1b_0$ as:

$$\text{value} = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Range: 0 to $2^n - 1$

Unsigned Integer Range

For n bits, an unsigned integer can represent values from 0 to $2^n - 1$:

- 8 bits: 0 to 255
- 16 bits: 0 to 65,535
- 32 bits: 0 to 4,294,967,295 ($\approx 4.3 \times 10^9$)
- 64 bits: 0 to 18,446,744,073,709,551,615 ($\approx 1.8 \times 10^{19}$)

Example: The 8-bit sequence 1011_2 represents:

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}$$

10.2.2 Two's Complement for Signed Integers

To represent negative numbers, we need a scheme that:

1. Uses a fixed number of bits
2. Makes arithmetic operations simple (ideally using the same circuitry as unsigned)
3. Has a unique representation for zero

Two's complement achieves all three goals and is the standard for signed integer representation.

Definition: Two's Complement

In **two's complement** representation for n bits:

- The most significant bit (MSB) b_{n-1} is the **sign bit**: 0 for non-negative, 1 for negative
- The value is computed as:

$$\text{value} = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

- **Range**: -2^{n-1} to $2^{n-1} - 1$

Two's Complement Range

For n bits in two's complement:

- 8 bits: -128 to 127
- 16 bits: $-32{,}768$ to 32,767
- 32 bits: $-2{,}147{,}483{,}648$ to 2,147,483,647 ($\approx \pm 2.1 \times 10^9$)

Note the asymmetry: there is one more negative value than positive values.

Converting a positive number to its negative (and vice versa):

1. **Invert all bits** (flip 0s to 1s and 1s to 0s)
2. **Add 1** to the result

Example: Represent -6 in 4-bit two's complement.

1. Start with $+6 = 0110_2$
2. Invert all bits: 1001_2
3. Add 1: $1001_2 + 1 = 1010_2$
4. Result: $-6 = 1010_2$

Verification using the formula:

$$1010_2 = -1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6\checkmark$$

Two's Complement Heuristics

Useful shortcuts for working with two's complement:

1. If the sign bit is 0, the remaining bits give the magnitude directly
2. If the sign bit is 1, find the magnitude by inverting and adding 1 (same procedure as negation)
3. A number and its negation share the same bits from the rightmost 1 onwards; bits to the left of this are complements:
   - $+6 = 0\mathbf{110}$
   - $-6 = 1\mathbf{010}$

   (The rightmost '10' is the same; the bits to the left are complementary)

10.2.3 Excess-K (Biased) Notation

Excess notation is another way to represent signed integers, primarily used for exponents in floating-point numbers.

Definition: Excess-K Notation

In **excess-K** (or biased) notation, a value v is stored as the unsigned binary representation of $v + K$, where K is a fixed bias.

To decode: value = stored unsigned value $- K$

For n bits, typically $K = 2^{n-1}$ or $K = 2^{n-1} - 1$.

Example: Excess-8 notation with 4 bits

The bias is $K = 8$, so zero is stored as $8 = 1000_2$.

Value	Stored (unsigned)	Binary
$+7$	$7+8=15$	1111
$+6$	$6+8=14$	1110
$+1$	$1+8=9$	1001
0	$0+8=8$	**1000**
-1	$-1+8=7$	0111
-7	$-7+8=1$	0001
-8	$-8+8=0$	**0000**

Why Use Excess Notation?

The key advantage of excess notation is that the ordering of stored values matches the ordering of actual values when treated as unsigned integers. This makes comparison operations simpler in hardware-important for comparing floating-point exponents quickly.

11 Binary Arithmetic

11.1 Binary Addition

Binary addition follows the same principles as decimal addition, but with only two digits:

$$\begin{aligned} 0+0&=0\\ 0+1&=1\\ 1+0&=1\\ 1+1&=10_2 \quad (0 \text{ with carry } 1) \end{aligned}$$

Example: Add $00111001_2 + 01101101_2$ (57 + 109 in decimal)

$$\begin{array}{r} 00111001\\ +01101101\\ \hline 10100110 \end{array}$$

Result: $10100110_2 = 166_{10}$ ✓

11.2 Binary Subtraction via Two's Complement

One of the elegant properties of two's complement is that subtraction can be performed using addition:

$$A - B = A + (-B) = A + (\text{two's complement of } B)$$

This means the same hardware circuit can perform both addition and subtraction.

11.3 Overflow

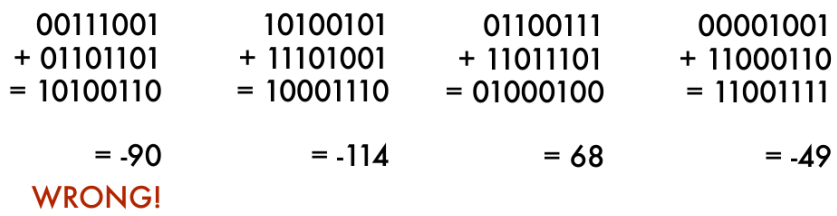

Figure 3: Overflow occurs when the result of an arithmetic operation exceeds the representable range. When adding two 8-bit numbers that produce a 9-bit result, the leftmost bit is truncated, leading to an incorrect answer within the 8-bit representation.

Definition: Overflow

Overflow occurs when an arithmetic operation produces a result outside the representable range for the given number of bits. The result "wraps around" due to the fixed bit width.

Detecting Overflow

Overflow detection differs for signed and unsigned arithmetic:

Unsigned overflow: Occurs when there is a carry out of the most significant bit.

Signed (two's complement) overflow: Occurs when:

- Adding two positive numbers yields a negative result, OR
- Adding two negative numbers yields a positive result

Equivalently: overflow occurs when the carry into the sign bit differs from the carry out of the sign bit.

Example: Signed overflow with 4-bit two's complement

Compute $7 + 3$:

- $7 = 0111_2$
- $3 = 0011_2$

$$\begin{array}{r} 0111 \\ +0011 \\ \hline 1010 \end{array}$$

The result 1010_2 is interpreted in two's complement as:

$$1010_2 = -1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6$$

We added two positive numbers (7 and 3) and got a negative result (-6). This is **signed overflow**-the true answer (10) exceeds the maximum representable value (7) for signed 4-bit integers.

Overflow in Programming

Many programming languages do not automatically check for integer overflow:

- In C/C++, signed overflow is undefined behaviour; unsigned overflow wraps around
- Python's arbitrary-precision integers cannot overflow (but are slower)
- Java's integers overflow silently with wrap-around

Overflow bugs have caused serious software failures, including the Ariane 5 rocket explosion (1996).

12 Floating-Point Representation

Integers can only represent whole numbers. For fractional values and very large or very small numbers, we use floating-point representation.

12.1 The Concept of Floating Point

Definition: Floating-Point Representation

A **floating-point number** represents a real number in the form:

$$(-1)^s \times m \times 2^e$$

where:

- s is the **sign bit** (0 = positive, 1 = negative)
- m is the **mantissa** (or significand), representing the significant digits
- e is the **exponent**, determining the magnitude

This is analogous to scientific notation in decimal (e.g., 6.022×10^{23}), but in base 2.

12.2 8-Bit Floating Point (Pedagogical Example)

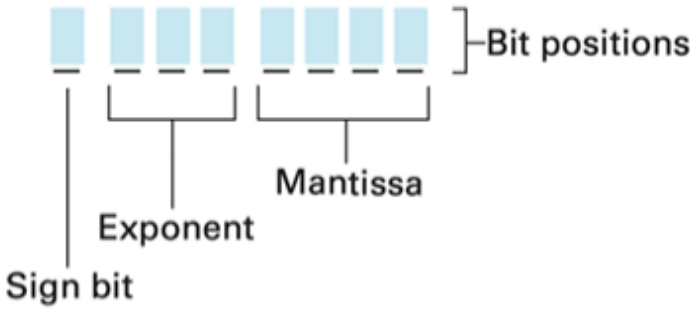

Figure 4: Structure of an 8-bit floating-point format: 1 sign bit, 3 exponent bits (using excess-4 notation), and 4 mantissa bits. Real-world formats use more bits but follow the same principles.

Consider a simplified 8-bit floating-point format:

- **1 sign bit**: Position 7 (leftmost)
- **3 exponent bits**: Positions 4–6, using excess-4 notation
- **4 mantissa bits**: Positions 0–3

12.2.1 Normalised Form and the Hidden Bit

In **normalised** floating-point numbers, the mantissa is adjusted so that the leading digit (before the radix point) is always 1. Since this leading 1 is always present, it can be implicit (not stored), giving us an extra bit of precision "for free."

Thus, a stored mantissa of 1011 actually represents 1.1011_2.

12.2.2 Conversion: Binary Floating Point to Decimal

01101011	10001000	01111111	00111100
+.1011	-.1000	+.1111	+.1100
+10.11	-.00001000	+111.1	+.011
2 + ½ + ¼	$-1/32$	+7 + ½	¼ + $1/8$
2 ¾	$-1/32$	7 ½	$3/8$

Figure 5: Examples of converting 8-bit floating-point representations to decimal values. The process involves extracting the sign, decoding the exponent (subtracting the bias), and reconstructing the mantissa with the implicit leading 1.

Example 1: Convert 01101011_2 to decimal.

1. **Sign bit**: 0 (positive)
2. **Exponent bits**: $110_2 = 6_{10}$; subtract bias: $6 - 4 = 2$
3. **Mantissa bits**: 1011; with implicit leading 1: 1.1011_2
4. **Value**: $1.1011_2 \times 2^2 = 110.11_2$

Converting 110.11_2 to decimal:

$$110.11_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 2 + 0 + 0.5 + 0.25 = 6.75$$

Example 2: Convert 10001000_2 to decimal.

1. **Sign bit**: 1 (negative)
2. **Exponent bits**: $000_2 = 0_{10}$; subtract bias: $0 - 4 = -4$
3. **Mantissa bits**: 1000; with implicit leading 1: 1.1000_2
4. **Value**: $-(1.1000_2 \times 2^{-4}) = -0.00011_2$

Converting -0.00011_2 to decimal:

$$-0.00011_2 = -(1 \times 2^{-4} + 1 \times 2^{-5}) = -(0.0625 + 0.03125) = -0.09375$$

12.2.3 Conversion: Decimal to Binary Floating Point

Example: Convert $1\frac{1}{3}$ to our 8-bit format.

Step 1: Convert to binary.

$1\frac{1}{3} = 1.333\ldots$

For the fractional part, multiply by 2 repeatedly:

$$\begin{aligned} 0.333\ldots \times 2 = 0.666\ldots &\quad \rightarrow \quad 0 \\ 0.666\ldots \times 2 = 1.333\ldots &\quad \rightarrow \quad 1 \\ 0.333\ldots \times 2 = 0.666\ldots &\quad \rightarrow \quad 0 \\ \vdots \text{ (repeating)} & \end{aligned}$$

So $1\frac{1}{3} = 1.\overline{01}_2$ (repeating pattern).

Step 2: Normalise.

Already normalised as $1.010101\ldots_2 \times 2^0$

Step 3: Extract components.

- Sign: 0 (positive)
- Exponent: 0; stored as $0 + 4 = 4 = 100_2$
- Mantissa: First 4 bits after the decimal: 0101 (truncated from $010101\ldots$)

Result: $0\,100\,0101_2 = 01000101_2$

Verification: Converting back: $1.0101_2 \times 2^0 = 1.3125_{10}$

Note the error: $1.333\ldots - 1.3125 = 0.020\overline{8}$. This demonstrates the inherent precision limitation of floating point.

12.3 IEEE 754 Standard

Real-world computing uses the IEEE 754 standard for floating-point arithmetic.

IEEE 754 Formats

Single Precision (32-bit):

- 1 sign bit
- 8 exponent bits (excess-127)
- 23 mantissa bits (+1 implicit)
- Precision: 6–9 significant decimal digits
- Range: $\approx \pm 3.4 \times 10^{38}$

Double Precision (64-bit):

- 1 sign bit
- 11 exponent bits (excess-1023)
- 52 mantissa bits (+1 implicit)
- Precision: 15–17 significant decimal digits
- Range: $\approx \pm 1.8 \times 10^{308}$

12.4 Floating-Point Precision Issues

Floating-Point Gotchas

Floating-point arithmetic has several counter-intuitive behaviours that can cause bugs:

1. Representation error: Many decimal fractions cannot be represented exactly in binary.

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

2. Comparison failures:

```
>>> (3 - 2.9) <= 0.1
False  # 3 - 2.9 is slightly greater than 0.1!
```

3. Absorption: Adding a very small number to a very large number may have no effect.

```
>>> 1e16 + 1 == 1e16
True  # The 1 is too small to affect the large number
```

4. Non-associativity: $(a + b) + c \neq a + (b + c)$ in floating point.

Best Practices for Floating-Point Comparison

Never compare floating-point numbers for exact equality. Instead:

```
import math

# Use a tolerance (epsilon)
def approx_equal(a, b, rel_tol=1e-9, abs_tol=0.0):
    return abs(a - b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)

# Or use math.isclose (Python 3.5+)
math.isclose(0.1 + 0.2, 0.3)  # True
```

13 CPU Architecture

Understanding CPU architecture provides context for how algorithms are actually executed and why certain operations are faster than others.

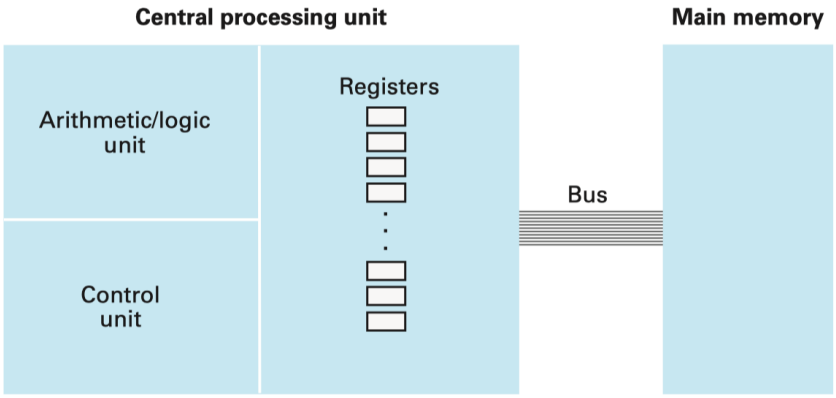

Figure 6: Simplified CPU architecture showing the main components: the Arithmetic Logic Unit (ALU), Control Unit, registers, and their connection to main memory via the system bus.

13.1 CPU Components

13.1.1 Arithmetic Logic Unit (ALU)

The ALU is the computational core of the CPU, performing:

- **Arithmetic operations**: Addition, subtraction, multiplication, division
- **Logical operations**: AND, OR, NOT, XOR, and other bitwise operations
- **Comparison operations**: Testing equality, less than, greater than

13.1.2 Control Unit

The Control Unit orchestrates the operation of the CPU:

- **Instruction Fetch**: Retrieves the next instruction from memory
- **Instruction Decode**: Interprets what operation the instruction requires
- **Execution Control**: Directs the ALU, manages data flow, and coordinates components

13.1.3 Registers

Registers: The Fastest Storage

Registers are small, extremely fast storage locations within the CPU itself. They hold:

- Data currently being processed
- Memory addresses being accessed
- The program counter (address of the next instruction)
- Status flags (overflow, zero result, carry, etc.)

Register access is essentially instantaneous compared to memory access (typically 1 cycle vs. 100+ cycles for main memory).

13.1.4 Memory Hierarchy

Data moves between different levels of storage, each with different speed/capacity trade-offs:

Level	Typical Size	Access Time
Registers	64–256 bytes	< 1 ns
L1 Cache	32–64 KB	1–4 ns
L2 Cache	256 KB – 1 MB	4–10 ns
L3 Cache	4–50 MB	10–40 ns
Main Memory (RAM)	8–64 GB	50–100 ns
SSD Storage	256 GB – 4 TB	10–100 μs
HDD Storage	1–10 TB	1–10 ms

This hierarchy is crucial for algorithm performance: algorithms that access memory in predictable patterns ("cache-friendly") can be orders of magnitude faster than those with scattered access patterns.

13.2 The Stored-Program Concept

The von Neumann Architecture

The **stored-program concept** (von Neumann architecture) is the fundamental principle that both data and programs are stored in the same memory as sequences of bits. This means:

- Programs can be loaded and modified like any other data
- Programs can generate and execute other programs
- The same memory and processing hardware handles both code and data

An instruction in memory consists of:

- **Opcode**: Specifies the operation (e.g., ADD, LOAD, STORE, JUMP)

- **Operands**: Specify the data or addresses to operate on

13.3 The Machine Cycle

The CPU operates in a continuous cycle:

1. **Fetch**: Retrieve the next instruction from memory (address in program counter)
2. **Decode**: Interpret the instruction to determine the operation and operands
3. **Execute**: Perform the operation using the ALU or other components
4. **Repeat**: Increment the program counter and return to step 1

Modern CPUs use **pipelining** to overlap these stages for different instructions, and **multiple cores** to execute instructions in parallel, achieving billions of operations per second.

14 Relevance to Data Structures and Algorithms

Why This Matters

Understanding low-level computing concepts is essential for DSA because:

Space complexity analysis:

- An `int` typically uses 4 bytes; a `double` uses 8 bytes
- An array of n integers uses $4n$ bytes plus overhead
- Pointer/reference sizes depend on architecture (4 or 8 bytes)

Time complexity in practice:

- Array access is $O(1)$ because of random-access memory
- Cache-efficient algorithms can be 10–100x faster than cache-unfriendly ones with the same big-O complexity
- Integer arithmetic is faster than floating-point; bitwise operations are fastest

Correctness:

- Integer overflow can cause incorrect results or security vulnerabilities
- Floating-point comparison requires tolerance-based approaches
- Character encoding matters for string algorithms

15 Summary

This chapter established the foundational concepts of how computers represent and process information:

- **Bits and Logic Gates**: All computation reduces to Boolean operations on bits
- **Hexadecimal**: A compact notation for binary data
- **Memory**: Organised collections of bits with addresses; RAM provides $O(1)$ access
- **Text Representation**: ASCII for basic characters; Unicode for universal coverage
- **Integer Representation**: Unsigned for non-negative; two's complement for signed; excess notation for biased values
- **Binary Arithmetic**: Addition/subtraction with overflow considerations
- **Floating Point**: Approximate representation of real numbers with inherent precision limitations
- **CPU Architecture**: The fetch-decode-execute cycle operating on data in registers and memory

These concepts form the substrate upon which all data structures are built and all algorithms execute. A deep understanding of this level enables you to make informed decisions about data representation, predict performance characteristics, and avoid subtle bugs in numerical computation.

16 Common Data Structures

This week bridges the theoretical foundations of computation with the practical data structures you will use throughout your programming career. We begin by formalising what it means for a function to be computable, leading us to the Turing machine-a simple yet powerful abstraction that defines the limits of algorithmic computation. Understanding these limits helps us appreciate both what algorithms can achieve and what lies forever beyond their reach.

We then turn to the fundamental data structures-arrays, linked lists, stacks, queues, and trees-that form the building blocks of virtually every program. The choice of data structure profoundly affects algorithm efficiency, and understanding how these structures are stored in memory illuminates why certain operations are fast while others are slow.

17 Functions and Computability

Definition: Function

A **function** $f : X \to Y$ is a mapping from a set X (the domain) to a set Y (the codomain or range). For each element $x \in X$, there exists exactly one corresponding element $f(x) \in Y$.

More precisely, a function is a relation $f \subseteq X \times Y$ such that:

1. For every $x \in X$, there exists some $y \in Y$ with $(x, y) \in f$.
2. If $(x, y_1) \in f$ and $(x, y_2) \in f$, then $y_1 = y_2$.

The concept of a function is central to programming: every program can be viewed as implementing a function that maps inputs to outputs. However, a crucial question arises: can every mathematically definable function be computed by a program?

The answer, perhaps surprisingly, is **no**. There exist well-defined mathematical functions that no algorithm can compute. To make this precise, we need to formalise what "computation" means.

Not All Functions Are Computable

There are infinitely more functions than there are possible programs. This follows from a counting argument:

- The set of all programs is countable (each program is a finite string of symbols)
- The set of all functions $\mathbb{N} \to \{0, 1\}$ is uncountable (by Cantor's diagonal argument)

Therefore, "most" functions cannot be computed by any program. This is not a limitation of current technology-it is a fundamental mathematical fact.

18 Turing Machines

The Turing machine, introduced by Alan Turing in 1936, provides a precise mathematical definition of computation. Despite its simplicity, it captures the full power of any physically realisable computer.

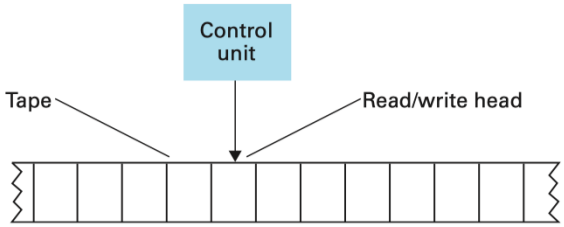

Figure 7: Schematic of a Turing machine. The machine reads and writes symbols on an infinite tape, moves left or right, and transitions between states according to a finite set of rules.

Definition: Turing Machine

A **Turing machine** consists of:

1. An **infinite tape** divided into cells, each containing a symbol from a finite alphabet Γ (including a blank symbol $\sqcup$)
2. A **head** that can read and write symbols, and move left or right
3. A **finite set of states** Q, including a start state q_0 and halt states
4. A **transition function** $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$

At each step, based on the current state and the symbol under the head, the transition function determines:

- The new state to enter
- The symbol to write (replacing the current symbol)
- The direction to move the head (Left or Right)

A Turing machine typically has a finite number of meaningful states that encode the "phase" of computation. For example, a machine performing addition might have states such as:

- `START` - initial state, reading the first operand
- `ADD` - performing addition on current digit
- `CARRY` - propagating a carry to the next digit
- `OVERFLOW` - handling overflow condition
- `RETURN` - moving back to output position
- `HALT` - computation complete, machine stops

The power of the Turing machine lies not in the complexity of individual operations (each step is trivially simple) but in the ability to chain arbitrarily many steps together. This simple model can simulate any algorithm that can be expressed in any programming language.

18.1 The Church–Turing Thesis

The Church–Turing Thesis

Any function that can be computed algorithmically can be computed by a Turing machine.

This is a thesis, not a theorem-it cannot be mathematically proved because "algorithmically computable" is an informal notion. However, the thesis is supported by:

- Every formal model of computation ever proposed (lambda calculus, recursive functions, cellular automata, quantum computers) has been shown equivalent in power to Turing machines
- No one has ever found a counterexample-a function computable by some other means but not by a Turing machine

The Church–Turing thesis guides the design and analysis of algorithms. If you can describe an algorithm in any reasonable computational model, you can be confident it is Turing-computable. Conversely, if something is not computable by a Turing machine, no programming language or computer architecture can compute it.

18.2 Computable Functions

What makes a function computable? The following criteria characterise functions that a Turing machine (or equivalently, any programming language) can compute:

Criteria for Computability

A function $f : X \to Y$ is **computable** if there exists a Turing machine M such that:

1. **Finite instructions**: Given any input $x \in X$ encoded on the tape, M follows a finite, unambiguous sequence of steps.
2. **Halts on valid inputs**: For every x in the domain, M eventually reaches a halt state with $f(x)$ on the tape.
3. **Behaviour on invalid inputs**: For inputs not in the domain, M either never halts or enters an error state (this behaviour is permitted but not required to be consistent).

Computable functions have two important characteristics:

- **Deterministic**: Given the same input, the machine follows the exact same sequence of steps and produces the same output. There is no randomness in the transition function (though we can simulate randomness if provided with a random input).
- **Algorithmic**: The computation is specified by clear, finite instructions that could in principle be followed mechanically. There is no appeal to intuition, creativity, or infinite processes.

18.3 Turing Completeness

Definition: Turing Completeness

A computational system is **Turing complete** if it can simulate any Turing machine. Equivalently, it can compute any computable function, given enough time and memory.

A system is Turing complete if and only if it supports:

1. **Conditional branching**: The ability to execute different instructions based on data values (e.g., `if` statements)
2. **Arbitrary memory access**: The ability to read and write to an unbounded memory store
3. **Repetition**: Either loops (`while`, `for`) or recursion

Practical Implications of Turing Completeness

Most modern programming languages-Python, Java, C++, JavaScript, Haskell-are Turing complete. This means:

- Any algorithm that can be expressed in one language can be expressed in any other
- The choice of language does not affect what is computable, only what is convenient
- Even seemingly simple systems (e.g., Excel with iterative calculation, Conway's Game of Life, certain card games) can be Turing complete

Python, for instance, is Turing complete because it provides:

```
# Conditional branching
if condition:
    do_something()

# Loops (repetition)
while running:
    process()

for item in collection:
    handle(item)

# Recursion (alternative repetition mechanism)
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

# Arbitrary memory access (via data structures)
memory = {}
memory[address] = value
```

Turing Completeness Has Consequences

Turing completeness is a double-edged sword. If a system is Turing complete, it inherits all the limitations of Turing machines:

- Programs can loop forever (non-halting behaviour)
- It is impossible to automatically verify arbitrary properties of programs
- Security vulnerabilities can arise from unexpected computational power

Some systems (e.g., configuration languages, SQL without recursion) are deliberately not Turing complete to avoid these issues.

19 The Halting Problem

The halting problem is the most famous undecidable problem in computer science. It demonstrates a fundamental limitation on what computers can determine about their own behaviour.

The Halting Problem

Problem statement: Given a description of a program P and an input I, determine whether P will eventually halt (stop executing) when run on input I, or whether it will run forever.

Undecidability: There is no algorithm that solves the halting problem for all possible program-input pairs. That is, no program $H(P, I)$ can exist that:

- Returns `HALTS` if $P(I)$ terminates
- Returns `LOOPS` if $P(I)$ runs forever

for all choices of P and I.

Before proving undecidability, let us see that some specific programs can be analysed:

Example: A Non-Halting Program

```
x = 1
while x != 0:
    x = x + 1
```

Analysis:

- Initial value: $x = 1$
- Loop condition: continue while $x \neq 0$
- Loop body: increment x by 1
- Since x starts at 1 and increases, it never equals 0. The loop runs **forever**.

Example: A Halting Program

```
x = 0
while x != 0:
    x = x + 1
```

Analysis:

- Initial value: $x = 0$
- Loop condition: continue while $x \neq 0$
- Since $x = 0$ initially, the condition is false from the start. The loop body never executes, and the program **halts immediately**.

While specific programs can often be analysed, the key result is that no general algorithm works for all programs.

19.1 Proof of Undecidability via Self-Reference

The proof that the halting problem is undecidable uses a clever self-referential argument-a diagonalisation technique similar to Cantor's proof that the real numbers are uncountable.

Proof by Contradiction

Suppose, for the sake of contradiction, that a halting oracle H exists. That is, $H(P, I)$ is a program that:

- Returns `True` if program P halts on input I
- Returns `False` if program P runs forever on input I

We construct a paradoxical program D (for "diagonal") as follows:

```
def D(P):
    """A␣paradoxical␣program␣that␣defies␣the␣halting␣oracle."""
    if H(P, P):       # If P halts when given itself as input...
        while True:   # ...then loop forever
            pass
    else:             # If P loops forever when given itself...
        return        # ...then halt immediately
```

Now we ask: **What happens when we run $D(D)$?**-that is, we feed D its own source code as input.

Case 1: Suppose $D(D)$ halts.

- Then $H(D, D)$ returns `True` (since the oracle correctly predicts halting)
- But then D enters the `while True` loop and runs forever
- Contradiction: $D(D)$ both halts and runs forever

Case 2: Suppose $D(D)$ runs forever.

- Then $H(D, D)$ returns `False` (since the oracle correctly predicts non-halting)
- But then D executes `return` and halts
- Contradiction: $D(D)$ both runs forever and halts

Both cases lead to contradiction. Therefore, our assumption that H exists must be false. **No halting oracle can exist.** □

Why Self-Reference Works

The key insight is that programs can inspect and manipulate other programs (including themselves) because programs are just data-strings of characters that can be passed as input.

This "programs are data" principle is fundamental:

- Compilers take programs as input
- Interpreters execute programs represented as data
- Debuggers analyse running programs
- The halting problem asks us to analyse programs about programs

Self-reference creates the paradox: we build a program whose behaviour contradicts any prediction about it.

Implications for Software Development

The undecidability of the halting problem has practical consequences:

- **No perfect bug detector**: You cannot write a program that detects all infinite loops
- **No perfect optimiser**: Determining if code is dead (unreachable) is undecidable in general
- **No perfect verifier**: Automatically proving that a program satisfies its specification is impossible in general

This does not mean analysis tools are useless-they can catch many bugs-but they must either be incomplete (miss some bugs), unsound (report false positives), or require human assistance.

20 Data Types

Having established the theoretical foundation of computation, we now turn to the practical question of how data is represented and organised in programs. Data types define both how values are stored in memory and what operations are valid on those values.

20.1 Primitive Data Types

Integers come in two main varieties:

- **Unsigned integers**: Represent non-negative whole numbers. An n-bit unsigned integer can represent values from 0 to $2^n - 1$.
- **Signed integers**: Represent positive and negative whole numbers, typically using two's complement representation. An n-bit signed integer can represent values from -2^{n-1} to $2^{n-1} - 1$.

Floating-point numbers represent fractions and very large or small numbers using scientific notation encoded in binary. The IEEE 754 standard defines formats like 32-bit (float) and 64-bit (double) with a sign bit, exponent, and mantissa. Floating-point arithmetic involves trade-offs between range and precision (covered in Week 2).

Text is represented using character encodings:

- **ASCII**: 7-bit encoding for 128 characters (English letters, digits, punctuation, control characters)
- **Unicode**: Variable-width encoding supporting over 140,000 characters from virtually all writing systems. UTF-8 is the most common encoding, using 1–4 bytes per character.

20.2 Lists

In Python, the **list** is the central way to organise collections of items. Lists are ordered, mutable sequences with several important properties:

- **Ordered**: Elements have positions; the first element is at index 0 (the head), the last at index -1 (the tail)
- **Heterogeneous**: A single list can contain elements of different types (integers, strings, other lists, etc.)
- **Dynamic**: Lists can grow and shrink; elements can be added or removed at any position

List Comprehensions

Python's list comprehensions provide a concise syntax for creating lists through transformation and filtering:

```
# Basic transformation: apply function to each element
squares = [x**2 for x in range(10)]

# With filtering: include only elements satisfying a condition
evens = [x for x in range(20) if x % 2 == 0]

# Combined: transform and filter
valid_scores = [process(v) for v in scores if v is not None]
```

List comprehensions are not just syntactic sugar-they are often faster than equivalent `for` loops because the iteration is implemented in C within the Python interpreter.

20.3 Arrays

An **array** is a specialised data structure for storing collections of elements of the same type in contiguous memory. Unlike Python lists, arrays have fixed-size elements, enabling efficient random access.

Figure 8: A 2D array with shape $(3, 4)$-3 rows and 4 columns. The values 1 through 12 are stored contiguously in memory, with metadata tracking the shape.

Arrays carry **metadata** that describes their structure:

```
shape = (3, 4)  # 3 rows, 4 columns
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

How multi-dimensional arrays map to linear memory depends on the **memory layout**:

- **Row-major order** (C, NumPy default): Consecutive elements of a row are adjacent in memory. Element (i, j) is at position $i \cdot \text{cols} + j$.
- **Column-major order** (Fortran, MATLAB): Consecutive elements of a column are adjacent. Element (i, j) is at position $j \cdot \text{rows} + i$.

NumPy Array Fundamentals

NumPy provides efficient multi-dimensional arrays for numerical computing:

```
import numpy as np

# 1D array (vector)
v = np.array([1, 2, 3])
v.shape  # (3,) - tuple with one element

# 2D array (matrix)
m = np.array([[1, 2], [3, 4]])
m.shape  # (2, 2)

# Accessing elements
m[0, 1]  # 2 (row 0, column 1)

# Slicing
m[:, 0]  # array([1, 3]) - first column
m[1, :]  # array([3, 4]) - second row
```

Note the difference between a 1D array with shape `(3,)` and a 2D array with shape `(3, 1)` or `(1, 3)`-they have the same elements but different behaviours in operations like matrix multiplication.

20.4 Hash Tables

A **hash table** (called `dict` in Python) provides fast key-value lookup using a hash function.

Definition: Hash Table

A hash table stores key-value pairs using the following mechanism:

1. A **hash function** $h : K \to \{0, 1, \ldots, n-1\}$ maps each key to an index in an underlying array of size n
2. To store pair (k, v): compute $i = h(k)$ and store v at position i
3. To retrieve value for key k: compute $i = h(k)$ and return the value at position i

Collision handling: Different keys may hash to the same index. Common strategies include:

- **Chaining**: Each array position holds a linked list of all pairs that hash there
- **Open addressing**: On collision, probe subsequent positions until an empty slot is found

Hash Table Performance

With a good hash function and appropriate table size:

- **Average case**: $O(1)$ for insertion, deletion, and lookup
- **Worst case**: $O(n)$ if all keys hash to the same position (pathological)

Python's `dict` uses sophisticated techniques to maintain near-constant time operations in practice.

21 Stacks and Queues

Stacks and queues are restricted data structures-lists with constrained access patterns. These constraints, far from being limitations, make the structures simpler to reason about and enable important algorithms.

21.1 The Stack

Definition: Stack

A **stack** is a linear data structure supporting two operations:

- **Push**: Add an element to the top of the stack
- **Pop**: Remove and return the top element

Stacks follow the **LIFO** (Last In, First Out) principle: the most recently added element is the first to be removed.

Intuition: Think of a stack of books or plates. You can only add to or remove from the top. Attempting to access items below the top requires first removing all items above them.

Metaphor: Leaving breadcrumbs while exploring a maze. Each step forward pushes a breadcrumb; each step backward pops one to retrace your path.

Applications of Stacks

- **Function call stack**: When a function calls another, the return address is pushed; on return, it is popped
- **Expression evaluation**: Parsing and evaluating arithmetic expressions
- **Undo functionality**: Each action is pushed; undo pops and reverses
- **Depth-first search**: Exploring graphs using backtracking
- **Bracket matching**: Checking that parentheses are balanced

In Python, you can use a list as a stack with `append()` to push and `pop()` to pop:

```
stack = []
stack.append(1)   # Push 1: stack = [1]
stack.append(2)   # Push 2: stack = [1, 2]
stack.append(3)   # Push 3: stack = [1, 2, 3]
top = stack.pop()   # Pop: top = 3, stack = [1, 2]
```

This is efficient because Python lists are implemented as dynamic arrays that grow from the end, making `append()` and `pop()` both $O(1)$ amortised operations.

21.2 Queues

Definition: Queue

A **queue** is a linear data structure supporting two operations:

- **Enqueue**: Add an element to the back (tail) of the queue
- **Dequeue**: Remove and return the front (head) element

Queues follow the **FIFO** (First In, First Out) principle: elements are processed in the order they arrived.

Intuition: Think of a queue of people waiting at a shop. New arrivals join at the back; the person at the front is served first.

Metaphor: A buffer of tasks to accomplish. Tasks enter the buffer as they arrive and are processed in arrival order.

Applications of Queues

- **Breadth-first search**: Exploring graphs level by level
- **Task scheduling**: Processing jobs in arrival order
- **Print spooling**: Documents printed in submission order
- **Message passing**: Asynchronous communication between processes
- **Buffering**: Smoothing out differences in processing rates

Python List as Queue: Inefficient!

While you can use a Python list as a queue with `append()` and `pop(0)`, this is inefficient:

```
queue = []
queue.append(1)   # Enqueue: O(1) - adds to end
queue.append(2)
first = queue.pop(0)   # Dequeue: O(n) - shifts all elements!
```

The `pop(0)` operation is $O(n)$ because every remaining element must be shifted left to fill the gap. For a proper queue, use `collections.deque`:

```
from collections import deque
queue = deque()
queue.append(1)      # O(1)
queue.popleft()      # O(1) - efficient!
```

22 Trees

Trees are hierarchical data structures that model relationships with a natural parent-child structure. They appear throughout computer science, from file systems to parsing to search algorithms.

Definition: Tree

A **tree** is a connected, acyclic graph consisting of:

- **Nodes**: The elements of the tree
- **Edges**: Connections between nodes (each node except the root has exactly one parent)
- **Root**: The topmost node with no parent
- **Leaves** (terminal nodes): Nodes with no children
- **Internal nodes**: Nodes with at least one child

The **depth** of a node is its distance from the root (number of edges on the path). The **height** of the tree is the maximum depth of any node-equivalently, the length of the longest path from root to leaf.

Common Tree Types

- **Binary tree**: Each node has at most two children (left and right)
- **Binary search tree (BST)**: Binary tree where left subtree contains smaller values, right subtree contains larger values. Enables $O(\log n)$ search in balanced trees.
- **Balanced trees** (AVL, Red-Black): BSTs with guaranteed $O(\log n)$ height through rebalancing
- **Heaps**: Complete binary trees satisfying the heap property (parent $\geq$ children for max-heap). Used for priority queues.
- **Tries**: Trees for storing strings, where each path from root represents a prefix

Trees provide efficient operations for many problems:

- Searching, inserting, and deleting in sorted data: $O(\log n)$ for balanced BSTs
- Representing hierarchical data (file systems, organisation charts)
- Expression parsing (syntax trees)
- Decision processes (game trees, decision trees in ML)

23 Storage and Memory

Understanding how data structures are stored in memory is crucial for analysing algorithm performance. The choice between different storage strategies-contiguous versus linked-has profound implications for operation efficiency.

23.1 Pointers

Definition: Pointer

A **pointer** is a variable that stores the memory address of another value. Instead of containing data directly, a pointer "points to" where the data is located.

In low-level terms:

- Memory is a large array of bytes, each with a unique address
- A pointer is an integer (the address) stored in memory, representing the location of another piece of data
- **Dereferencing** a pointer means accessing the data at the address it contains

Pointers enable:

- **Indirection**: Referring to data without copying it
- **Dynamic data structures**: Linked lists, trees, graphs where connections are represented by pointers

- **Sharing**: Multiple variables can point to the same data
- **Dynamic memory allocation**: Creating data structures whose size is determined at runtime

In Python, pointers are implicit. Variables are references (pointers) to objects, though Python hides the memory addresses from you. When you write `x = [1, 2, 3]`, the variable `x` holds a pointer to the list object.

23.2 Garbage Collection

When programs allocate memory dynamically, they must eventually release it. **Garbage collection** automates this process.

Definition: Garbage Collection

Garbage collection is automatic memory management that identifies and reclaims memory no longer in use by the program. The collector:

1. Maintains a record of all active variables and their associated memory locations
2. Traces which memory locations are reachable-accessible through some chain of references from active variables
3. Reclaims unreachable memory-locations that no variable can access

Why Garbage Collection Matters

- **Prevents memory leaks**: Forgetting to free memory causes programs to consume ever more memory
- **Prevents dangling pointers**: Freeing memory still in use causes crashes and security vulnerabilities
- **Simplifies programming**: Developers focus on logic, not memory bookkeeping

Python, Java, and JavaScript use garbage collection. C and C++ require manual memory management (or smart pointers).

23.3 Passing by Value vs. Reference

When you pass a variable to a function, what does the function receive? This is one of the most important distinctions in programming language semantics.

Pass by Value vs. Pass by Reference

Pass by value: The function receives a copy of the argument's value. Modifications inside the function do not affect the original variable.

Pass by reference: The function receives a reference (pointer) to the original data. Modifications inside the function do affect the original.

Consider this Python example:

```
def increment_value(x):
    x = x + 1  # Creates a new integer object
    return x

a = 1
output = increment_value(a)
print(a, output)  # Output: 1 2
```

Here, `a` remains 1 because integers are immutable in Python. The line `x = x + 1` creates a new integer object rather than modifying the original.

Now consider a mutable object:

```
def append_item(lst):
    lst.append(4)  # Modifies the list in place

my_list = [1, 2, 3]
append_item(my_list)
print(my_list)  # Output: [1, 2, 3, 4]
```

The list is modified because `lst` and `my_list` refer to the same object. Python uses "pass by object reference": the function receives a copy of the reference, not a copy of the object.

Python's Pass-by-Object-Reference

Python is neither purely pass-by-value nor pass-by-reference:

- The reference is copied (like pass-by-value)
- But the reference points to the same object (enabling reference-like behaviour)

Immutable objects (int, str, tuple): Behave like pass-by-value because modification creates a new object

Mutable objects (list, dict, set): Behave like pass-by-reference because in-place modification affects the original

To avoid modifying the original, explicitly copy: `lst.copy()` or `list(lst)`

Common Pitfall: Mutable Default Arguments

A notorious Python gotcha:

```
def add_item(item, lst=[]):   # Default list created ONCE
    lst.append(item)
    return lst

print(add_item(1))   # [1]
print(add_item(2))   # [1, 2] -- unexpected!
```

The default list is created once when the function is defined, not each time it is called. Use `None` as default instead:

```
def add_item(item, lst=None):
    if lst is None:
        lst = []
    lst.append(item)
    return lst
```

23.4 Memory Storage Strategies

Data structures can be stored in memory using two fundamental strategies, each with distinct trade-offs.

23.4.1 Contiguous Storage (Arrays)

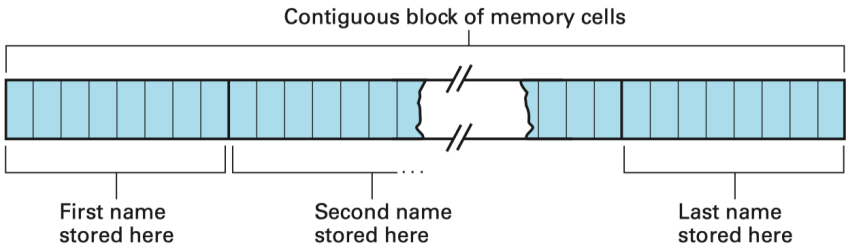

Figure 9: Contiguous storage: elements occupy consecutive memory addresses. Knowing the base address and element size, any element can be accessed directly by computing its address.

In **contiguous storage**, elements are stored in consecutive memory locations:

- **Random access in $O(1)$**: To access element i, compute address = base + $i\times$ element_size
- **Cache-friendly**: Consecutive elements are loaded together into CPU cache, improving performance for sequential access
- **Fixed element size required**: All elements must have the same size (or you need additional indirection)

However, contiguous storage has significant drawbacks for dynamic operations:

- **Insertion is $O(n)$**: Inserting at position i requires shifting all subsequent elements
- **Deletion is $O(n)$**: Deleting at position i requires shifting all subsequent elements to fill the gap
- **Resizing is expensive**: Growing beyond allocated capacity requires allocating new memory and copying all elements

23.4.2 Linked Storage (Linked Lists)

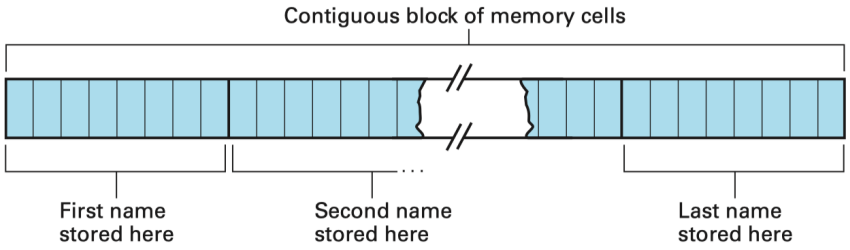

Figure 10: Linked storage: each node contains data and a pointer to the next node. Nodes can be anywhere in memory; the pointers connect them logically.

In **linked storage**, each element (node) contains data plus a pointer to the next element:

- **Dynamic size**: No pre-allocated capacity needed; grows and shrinks naturally
- **Flexible element sizes**: Each node can contain different amounts of data
- **Efficient insertion/deletion**: Once you have a pointer to a node, inserting or deleting nearby takes $O(1)$

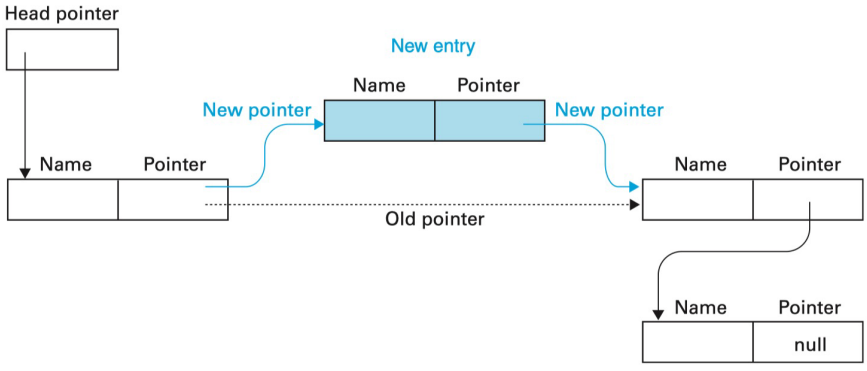

Figure 11: Insertion and deletion in a linked list. To remove a node, redirect the previous node's pointer to skip it. To insert, create a new node and update two pointers.

To **delete** a node: change one pointer (the previous node's "next" pointer skips the deleted node)

To **insert** a node: allocate a new node and change two pointers (previous node points to new node; new node points to what was previously next)

However, linked lists sacrifice random access:

- **Access is $O(n)$**: To reach element i, you must traverse from the head through i nodes
- **Cache-unfriendly**: Nodes are scattered in memory, causing cache misses
- **Memory overhead**: Each node requires extra space for the pointer(s)

Choosing Between Arrays and Linked Lists

Operation	Array	Linked List
Access by index	$O(1)$	$O(n)$
Search (unsorted)	$O(n)$	$O(n)$
Insert at beginning	$O(n)$	$O(1)$
Insert at end	$O(1)$ amortised	$O(1)$ with tail pointer
Insert in middle	$O(n)$	$O(1)$ after finding position
Delete	$O(n)$	$O(1)$ after finding position
Memory overhead	Low	Higher (pointers)
Cache performance	Excellent	Poor

Use arrays when: Random access is frequent, size is predictable, cache performance matters

Use linked lists when: Frequent insertions/deletions at arbitrary positions, size varies dramatically

23.4.3 Storing Stacks

Stacks can be efficiently implemented using contiguous storage with a **stack pointer**:

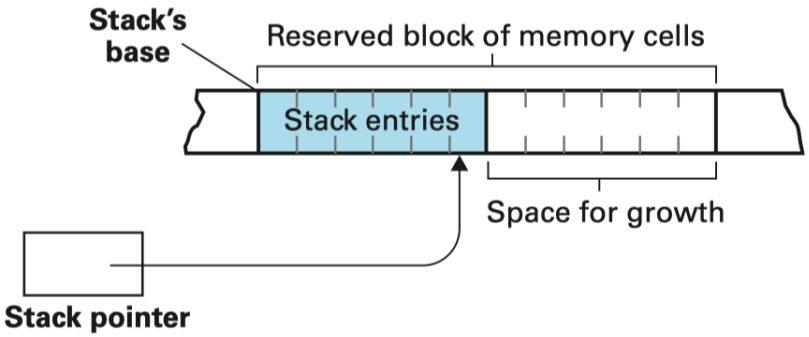

Figure 12: Stack storage using an array. The stack pointer indicates the top of the stack. Push increments the pointer and writes; pop reads and decrements.

- Reserve a contiguous block of memory for the stack
- Maintain a **stack pointer** indicating the current top
- **Push**: Increment pointer, write value at new top position
- **Pop**: Read value at top position, decrement pointer

Both operations are $O(1)$ and access only the end of the array, making this extremely efficient. The call stack in most programming languages works this way, which is why "stack overflow" errors occur when recursion is too deep-the reserved stack memory is exhausted.

23.4.4 Storing Queues

Queues present a challenge: both ends need efficient access. A clever solution is the **circular queue** (or ring buffer):

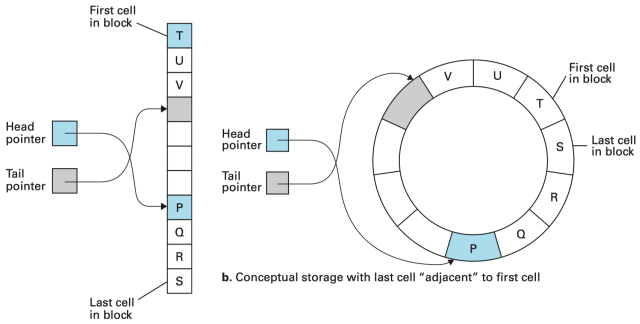

Figure 13: Circular queue implementation. The array wraps around: after the last position comes the first. Head and tail pointers track the front and back of the queue.

- Reserve a fixed-size array
- Maintain two pointers: **head** (front, where dequeue occurs) and **tail** (back, where enqueue occurs)
- When a pointer reaches the end of the array, it wraps around to the beginning
- **Enqueue**: Write at tail, advance tail (with wraparound)
- **Dequeue**: Read at head, advance head (with wraparound)

Circular Queue Implementation

With array size n:

- Enqueue: `tail = (tail + 1) % n`
- Dequeue: `head = (head + 1) % n`
- Empty when: `head == tail`
- Full when: `(tail + 1) % n == head`

Note: We sacrifice one slot to distinguish empty from full (alternatively, maintain a count). This gives $O(1)$ for both operations with fixed memory.

The circular queue elegantly solves the problem of efficiently accessing both ends of a sequence without shifting elements. It is used extensively in operating systems (for I/O buffers), networking (packet queues), and producer-consumer scenarios.

24 Summary

Week 3 Key Concepts

Theory of Computation:

- Not all mathematical functions are computable
- Turing machines formalise what "computable" means
- The Church–Turing thesis: any algorithm can be expressed as a Turing machine
- Turing completeness: a system that can simulate any Turing machine
- The halting problem is undecidable-no general algorithm can determine if arbitrary programs terminate

Data Structures:

- **Arrays**: Contiguous storage, $O(1)$ random access, $O(n)$ insertion/deletion
- **Linked lists**: Pointer-based, $O(n)$ access, $O(1)$ insertion/deletion at known positions
- **Stacks**: LIFO, push/pop at one end only
- **Queues**: FIFO, enqueue at back, dequeue at front
- **Hash tables**: $O(1)$ average-case lookup using hash functions
- **Trees**: Hierarchical structure, $O(\log n)$ operations when balanced

Memory Concepts:

- Pointers enable indirection and dynamic data structures
- Garbage collection automates memory management
- Pass-by-value copies data; pass-by-reference shares data
- Storage strategy (contiguous vs. linked) determines operation efficiency

The concepts from this week form the vocabulary for algorithm analysis. When we say an algorithm is $O(n)$ or $O(\log n)$, we are implicitly assuming certain data structure operations take certain amounts of time. Understanding why array access is $O(1)$ while linked list access is $O(n)$ is essential for choosing the right data structure for a given problem.

25 Programming Paradigms

Programming paradigms are fundamental ways of thinking about and structuring code. Just as natural languages shape how we express ideas, programming paradigms shape how we express algorithms. Understanding paradigms matters for data structures and algorithms because different paradigms naturally suit different problems: functional programming excels at recursive data structures, object-oriented programming shines for modelling complex systems, and declarative approaches simplify data manipulation tasks.

This week explores the journey from low-level machine operations to high-level abstractions, then surveys the four major programming paradigms. We conclude with software engineering principles-coupling, cohesion, and information hiding-that guide how we structure code regardless of paradigm.

26 Low-Level to High-Level Abstractions

Programming languages exist on a spectrum from machine-level instructions to human-readable abstractions. Understanding this hierarchy illuminates why different languages feel so different to use, and why some operations are fast while others are slow.

26.1 Assembly: Unitary Operations

At the lowest level, programs consist of individual instructions that directly control the processor. Assembly language provides a thin veneer over raw machine code, using human-readable mnemonics for operations.

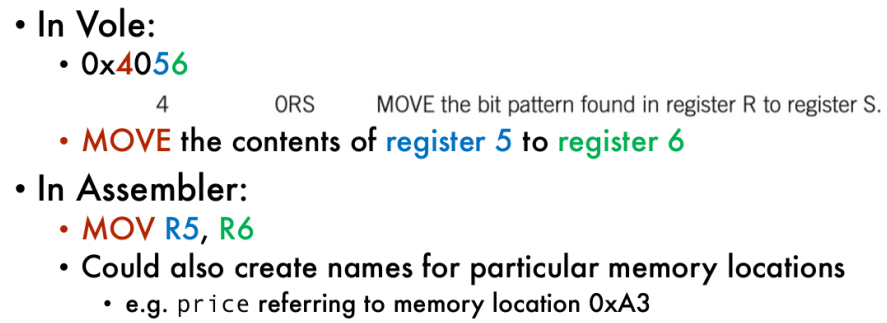

Figure 14: Assembly language example. Each line represents a single machine instruction: moving data between registers, performing arithmetic, or controlling program flow. The cryptic mnemonics (`MOV`, `ADD`, `JMP`) map directly to processor operations.

- **Moving data around memory**: The fundamental operations involve loading values into registers, storing them back to memory, and transferring between locations.
- **Machine-specific instructions**: Assembly code written for an Intel x86 processor will not run on an ARM processor-the instruction sets differ fundamentally.
- **Low-level concepts**: The programmer must manage registers, memory addresses, and processor flags directly.

Why Assembly Still Matters

Though rarely written by hand today, assembly language remains relevant for:

- Understanding what high-level code compiles to (performance analysis)
- Writing device drivers and operating system kernels
- Embedded systems with extreme resource constraints
- Security research and reverse engineering

26.2 Primitives: High-Level Instructions

Primitives are the basic building blocks provided by a programming language-data types, control structures, and fundamental operations. They abstract away the underlying machine operations.

- **Common tasks abstracted**: Rather than writing dozens of assembly instructions to add two numbers, you write `x + y`.
- **Each expression is a series of operations**: A simple statement like `total = price * quantity` may translate to many machine instructions.
- **Parsing trees for grammar**: Languages define formal grammars; the compiler or interpreter parses code into abstract syntax trees before execution.

High-Level Primitives

High-level primitives are **machine-independent**-they work identically regardless of the underlying processor architecture. This abstraction enables:

- **Portability**: The same source code runs on different machines
- **Productivity**: Programmers think in terms of the problem, not the hardware
- **Safety**: The language can prevent common errors (buffer overflows, type mismatches)

The cost is a layer of translation between your code and the machine.

26.3 Compilers: Translation to Machine Code

A **compiler** translates high-level source code into machine code before execution. The resulting binary file runs directly on the processor.

- **Translates primitives to machine code**: Takes machine-independent high-level code and converts it into machine-dependent operations.
- **Produces self-contained binaries**: The entire program is compiled once, producing an executable that runs without the original source code.

- **Target-specific backends**: To compile for different architectures (x86, ARM, etc.), the compiler needs separate code generation modules for each target.
- **Optimisation opportunities**: Because the compiler sees the entire program at once, it can perform extensive optimisations-inlining functions, reordering instructions, eliminating dead code.

Compiled Languages

Examples: C, C++, Rust, Go, Fortran, Haskell (via GHC).

Advantages: Fast execution, optimised machine code, no runtime dependency on source.

Disadvantages: Slower development cycle (compile-run-debug), platform-specific binaries, less flexible at runtime.

26.4 Interpreters: Direct Execution

An **interpreter** executes source code directly, reading and performing instructions on the fly without a separate compilation step.

- **Direct execution**: Reads high-level code and performs operations immediately, without producing a separate binary.
- **Line-by-line processing**: The interpreter handles each statement as it encounters it, enabling interactive execution.
- **Just-In-Time (JIT) compilation**: Modern interpreters often use a hybrid approach-compiling frequently-executed code paths to machine code during execution for better performance.

Performance considerations:

- Compilers can perform more optimisation because they analyse the entire program in advance.
- Compilers know the data types at compile time, avoiding runtime type checks. For example, knowing a variable is a floating-point number eliminates checks for what kind of data type it is.
- Interpreters trade execution speed for flexibility and development speed.

Advantages of interpretation:

- **Interactive development**: Execute code snippets immediately (REPL environments)
- **Flexibility**: Modify and test code without recompilation
- **Platform independence**: The same source runs anywhere the interpreter exists

Interpreted Languages

Examples: Python, JavaScript, Ruby, R, MATLAB.

Many "interpreted" languages actually use bytecode compilation: Python compiles to `.pyc` bytecode, which the Python Virtual Machine then interprets. This is a middle ground between pure interpretation and full compilation.

27 Four Programming Paradigms

A **programming paradigm** is a fundamental style of programming that provides a conceptual framework for structuring code. Most modern languages support multiple paradigms, but each has historical roots in a particular way of thinking.

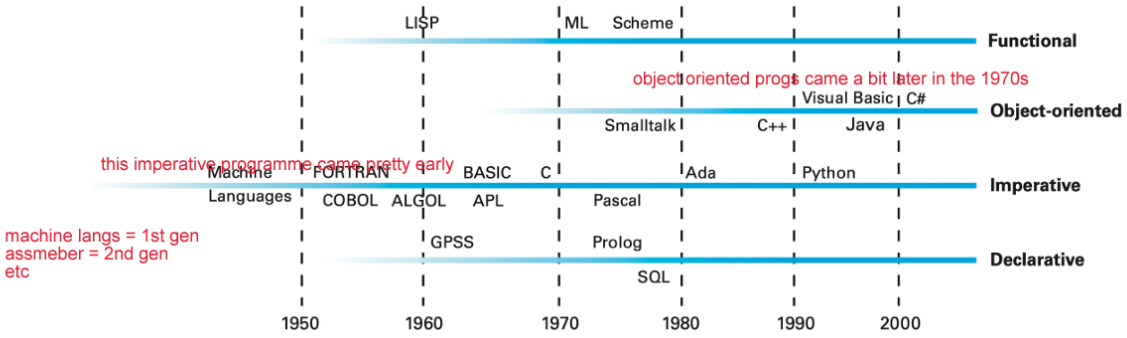

Figure 15: The four major programming paradigms. Imperative programming specifies step-by-step instructions. Declarative programming describes the desired result. Functional programming builds programs from composable functions. Object-oriented programming models systems as interacting objects.

Paradigm Summary

Imperative	"Follow these steps" - explicit control flow
Declarative	"Here is the problem" - describe what, not how
Functional	"Compose these functions" - transform data through functions
Object-Oriented	"Model these objects" - encapsulate state and behaviour

27.1 Imperative Programming

"Tell the machine the steps to take."

Imperative programming is the oldest and most direct paradigm. You write explicit instructions that modify program state, step by step, in the order they should execute.

- **Explicit control flow**: The programmer specifies exactly what happens at each step.
- **Requires precise knowledge**: You must know the exact sequence of operations to solve the problem.

- **Historical examples**: FORTRAN (1957, still used for numerical computing), COBOL (1959, still running banking systems), C (1972, systems programming).

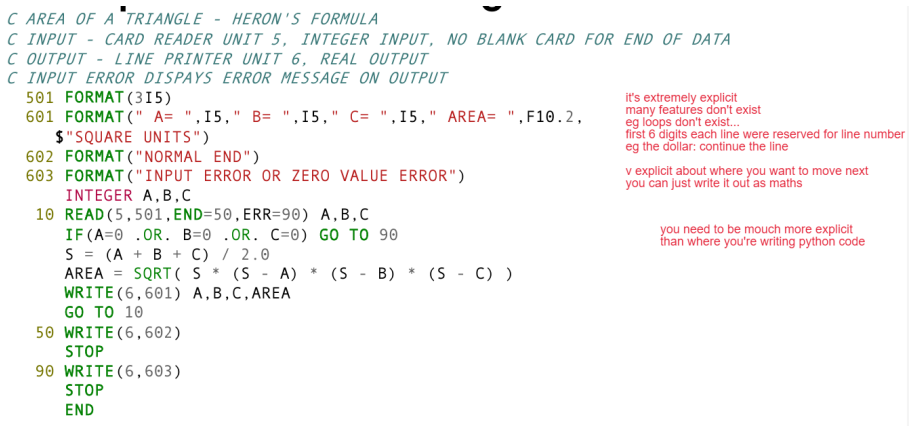

Figure 16: Fortran 77 code example. Originally written on punch cards, Fortran pioneered structured imperative programming. The numbered lines and explicit `GOTO` statements reflect the step-by-step nature of early imperative code. Remarkably, much high-performance numerical computing still relies on Fortran libraries today.

Characteristics of Imperative Programming

Imperative programs are characterised by:

1. **Sequential execution**: Statements execute in order (unless control flow directs otherwise)
2. **Mutable state**: Variables can be reassigned; the program's state changes over time
3. **Explicit control structures**: Loops (`for`, `while`), conditionals (`if/else`), jumps (`break`, `continue`)
4. **Assignment statements**: The fundamental operation is assigning values to variables

27.2 Declarative Programming

"Tell the machine what problem to solve."

Declarative programming inverts the imperative approach: rather than specifying how to compute something, you describe what you want. The system figures out the implementation details.

Imperative: "Follow these steps" vs **Declarative:** "Here is the problem"

- **Problem specification**: You must precisely define the problem, not the solution procedure.
- **Central to data science**: Much of what data scientists do is declarative-specify transformations, let the system optimise execution.
- **Agent-based modelling** exemplifies declarative thinking:

- Define exactly how one agent interacts with other agents
- Set up the environment they interact in
- Watch emergent behaviour unfold

Example: SQL (Declarative)

In SQL, you write what data you want, not how to obtain it. The database query optimiser determines the best execution plan.

```sql
SELECT *
FROM Book
WHERE price > 100.00
ORDER BY title;
```

```sql
SELECT Book.title AS Title,
       count(*) AS Authors
FROM Book
JOIN Book_author
ON Book.isbn = Book_author.isbn
GROUP BY Book.title;
```

```sql
SELECT isbn,
       title,
       price,
       price * 0.06 AS sales_tax
FROM Book
WHERE price > 100.00
ORDER BY title;
```

Figure 17: SQL query example. The query specifies desired columns, tables, join conditions, and filters-but says nothing about which indexes to use, how to order the joins, or how to scan the tables. The database engine makes these decisions.

Example: dplyr (Declarative Data Manipulation)

The R package `dplyr` provides a declarative grammar for data manipulation, letting you chain transformations without specifying implementation.

• (See dbplyr documentation)

```r
flights %>%
  group_by(month, day) %>%
  summarise(delay = mean(dep_delay)) %>%
  show_query()
#> Warning: Missing values are always
removed in SQL.
#> Use `AVG(x, na.rm = TRUE)` to silence
this warning #> <SQL>
#> SELECT `month`, `day`, AVG(`dep_delay`)
AS `delay`
#> FROM `nycflights13::flights`
#> GROUP BY `month`, `day`
```

R	SQL
inner_join()	SELECT * FROM x JOIN y ON x.a = y.a
left_join()	SELECT * FROM x LEFT JOIN y ON x.a = y.a
right_join()	SELECT * FROM x RIGHT JOIN y ON x.a = y.a
full_join()	SELECT * FROM x FULL JOIN y ON x.a = y.a

Figure 18: dplyr example. The pipe operator (`%>%`) chains declarative verbs: `filter`, `select`, `mutate`, `summarise`. You describe the transformation; dplyr handles execution.

27.3 Functional Programming

"Define entities which accept input and provide output."

Functional programming builds programs by composing small, independent functions. Each function takes inputs and produces outputs without modifying external state.

- **Composition**: Construct complex programs by combining simpler functions.
- **Mathematical foundation**: Functions in functional programming behave like mathematical functions-same input always yields same output.

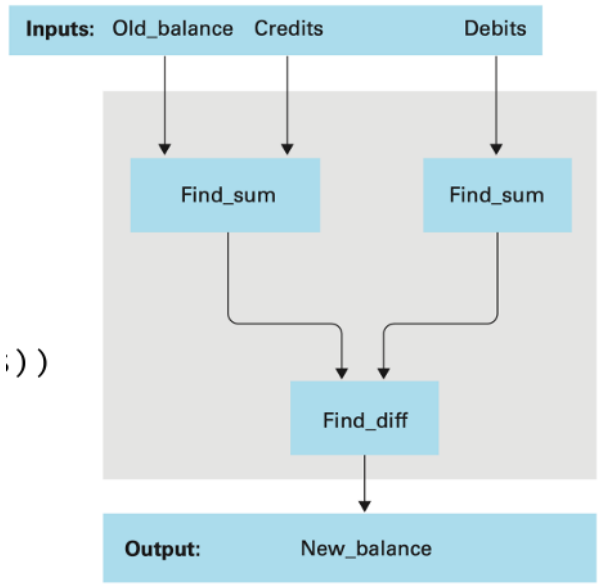

Figure 19: Functional approach to computing a balance. The expression `(Find_diff (Find_sum Old_balance Credits) (Find_sum Debits))` composes three functions. Data flows through the composition: credits and debits are summed separately, then differenced with the old balance.

```
Total_credits = sum of all Credits
Temp_balance = Old_balance + Total_credits
Total_debits = sum of all Debits
Balance = Temp_balance - Total_debits
```

Figure 20: Imperative approach to the same calculation. Variables are initialised, then mutated through loops. The state changes at each step, and the final result depends on the sequence of mutations.

27.3.1 Pure Functions

A **pure function** has two defining properties: it produces the same output for the same input, and it has no side effects.

Definition: Pure Function

A function f is **pure** if:

1. **Deterministic**: For any input x, $f(x)$ always returns the same value
2. **No side effects**: Calling $f(x)$ does not modify any state outside the function (no changing global variables, no database writes, no console output)

A function that reads from or writes to external state is **impure**.

Pure functions offer profound advantages:

- **Formal provability**: Pure functions can be mathematically reasoned about. Since they always produce the same output for the same input, you can prove properties about them-essential for mission-critical systems.

- **Modularity**: Each function does one thing. When testing or debugging, you need only consider the inputs and outputs, not hidden context or global state. Impure functions force you to reason about whether the surrounding context is correct.
- **Composability**: Pure functions combine freely. Because they have no side effects, you can compose them without worrying about execution order or interference. This enables powerful code reuse and leads to more readable, maintainable programs.

Pure Functions in Practice

Many functions you write daily are pure:

```
def square(x):
    return x * x  # Pure: same input, same output, no side effects

def add(a, b):
    return a + b  # Pure
```

These are impure:

```
total = 0
def add_to_total(x):
    global total
    total += x    # Impure: modifies global state
    return total

def get_random():
    return random.random()  # Impure: different output each call
```

27.3.2 Loops vs Recursion

Functional programming prefers **recursion** over **loops** because loops inherently involve mutable state.

Loops are not purely functional:

```
# Iterative Fibonacci - NOT functional
a = 0
b = 1
for term in range(50):
    print(f'term: {term} / number: {a}')
    (a, b) = (b, a + b)
```

- The variables `a` and `b` are reassigned (mutated) in each iteration.
- The loop must track which iteration we are on-state disconnected from the calculation itself.
- The calculation depends not just on the input but on the current iteration state.
- In purely functional code, you would never look outside the function itself.

Recursion is functional:

```
# Recursive Fibonacci - functional
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

- No side effects or state mutations-the function simply calls itself with different arguments.
- Each recursive call works with its own parameters; no shared mutable state.
- The code mirrors the mathematical definition: $F(n) = F(n-1) + F(n-2)$.
- Multiple function instances may execute (conceptually) simultaneously.

Recursion Performance

The naive recursive Fibonacci has exponential time complexity $O(2^n)$ because it recomputes the same values many times. Practical functional programming uses techniques like **memoisation** (caching results) or **tail recursion** (which compilers can optimise to loops) to achieve efficiency. The conceptual purity of recursion does not require sacrificing performance.

27.3.3 Anonymous and Higher-Order Functions

Anonymous functions (often called **lambda functions**) are functions defined without a name. They provide a concise way to create simple functions for single use, typically as arguments to other functions.

Definition: Higher-Order Function

A **higher-order function** is a function that either:

1. Takes one or more functions as arguments, and/or
2. Returns a function as its result

Higher-order functions are central to functional programming, enabling powerful abstractions like `map`, `filter`, and `reduce`.

Example: A higher-order function that applies a function then multiplies by 3:

```
def times3(a, b, function):
    return 3 * function(a, b)
```

Here, `times3` takes two numbers (`a` and `b`) and a function as arguments. It applies the provided function to `a` and `b`, then multiplies the result by 3.

```
# Using a lambda to add, then multiply by 3
add_then_times3 = times3(2, 4, lambda x, y: x + y)  # (2 + 4) * 3 = 18

# Using a lambda to subtract, then multiply by 3
sub_then_times3 = times3(2, 4, lambda x, y: x - y)  # (2 - 4) * 3 = -6
```

The anonymous functions `lambda x, y: x + y` and `lambda x, y: x - y` are defined inline without names. This avoids cluttering the namespace with one-off function definitions.

Power of Higher-Order Functions

Higher-order functions enable:

- **Customisation without boilerplate**: Pass different behaviours without defining separate named functions
- **Code reuse**: Write general algorithms parameterised by specific operations
- **Declarative style**: Express what transformation to apply, not how to loop through data

Common higher-order functions: `map(f, xs)`, `filter(pred, xs)`, `reduce(f, xs)`.

27.4 Object-Oriented Programming

Object-oriented programming (OOP) models programs as collections of **objects**-self-contained units that combine data (attributes) and behaviour (methods).

- **Modularisation**: Objects encapsulate related functionality, enabling changes without ripple effects.
- **Self-knowledge**: Objects maintain information about their own state.
- **Communication**: Objects interact by sending messages (method calls) to each other.
- **Actions**: Each object has a defined set of behaviours it can perform.
- **Mutability**: Objects can modify their own internal state.

27.4.1 Classes vs Objects

Definition: Class and Object

A **class** is a blueprint or template that defines:

- What data (attributes) instances will hold
- What behaviours (methods) instances can perform
- Initial values and default behaviours

An **object** (or **instance**) is a concrete realisation of a class:

- Created by instantiating the class
- Has its own copy of the instance attributes
- Shares method definitions with other instances of the same class

Analogy: A class is like an architectural plan for a house. It specifies the layout, materials, and features. An object is an actual house built from that plan-each house has the same structure but may have different paint colours, furniture, and occupants.

- **Class = blueprint**: Defines all possible configurations of the thing. When a class is defined, no memory is allocated.
- **Object = instantiated class**: A specific version with concrete data. Memory is allocated when the object is created.

27.4.2 Classes and Methods in Python

Constructor

The constructor method `__init__` is called automatically when an object is created. It initialises the object's attributes.

```
class Example:
    def __init__(self, attribute):
        self.attribute = attribute
```

Attributes

Attributes are variables belonging to an object. They hold data that can vary between instances and change over time.

```
class Example:
    def __init__(self):
        self.attribute = 'value'
```

Methods

Methods are functions defined inside a class. They define what the object can do-either to itself or to other objects.

```
class Example:
    def method(self):
        return 'I␣am␣a␣method'
```

Static Methods

Static methods do not require access to instance or class data. They are marked with the `@staticmethod` decorator and behave like regular functions that happen to live inside a class.

```
class Example:
    @staticmethod
    def static_method():
        return 'I␣am␣a␣static␣method'
```

Properties (Getters/Setters)

Properties provide controlled access to attributes, allowing validation or computation when getting or setting values.

```
class Example:
    def __init__(self, value):
        self._attribute = value

    @property
```

```
def attribute(self):
    return self._attribute

@attribute.setter
def attribute(self, value):
    self._attribute = value
```

27.4.3 Core Properties of OOP

Inheritance

Definition: Inheritance

Inheritance allows a new class (the child or subclass) to extend or modify an existing class (the parent or superclass). The child class inherits all attributes and methods of the parent, and can:

- Add new attributes or methods
- Override inherited methods with new implementations
- Extend inherited methods by calling the parent's version

Example: A `Regression` class might define data processing common to all regression analyses. A `RidgeRegression` class can extend `Regression` to add regularisation-specific behaviour, reusing the common code.

Polymorphism

Definition: Polymorphism

Polymorphism ("many forms") allows objects of different classes to respond to the same method call in different ways. Each object interprets the message according to its own class definition.

This enables designing interfaces that work with multiple types. You can write code that calls `.fit()` and `.predict()` on any model object, regardless of whether it is a linear regression, random forest, or neural network-each implements these methods according to its own algorithm.

Example: In scikit-learn, every model class implements `.fit()` and `.predict()`. The internal implementations vary dramatically, but the interface is uniform.

Encapsulation

Definition: Encapsulation

Encapsulation restricts access to an object's internal state, distinguishing between:

- **Public interface**: Methods and attributes intended for external use. The developer promises to maintain this interface.
- **Private internals**: Implementation details hidden from external code. These may change without notice.

Encapsulation means you interact with objects only through their promised capabilities, not their internal mechanisms. You do not need to understand how `.fit()` works internally-only what it does.

- **Public**: Stable interface the developer commits to maintaining.
- **Private**: Implementation details that may change; not to be relied upon.

In Python, private attributes and methods are conventionally denoted with a leading underscore (`_attribute`) or double underscore (`__attribute`). This is a convention, not enforced by the language.

28 Software Development Practices

Beyond paradigms, effective software development requires principles for organising code. This section covers literate programming, test-driven development, and the interrelated concepts of coupling, cohesion, and information hiding.

28.1 Literate Programming

Literate programming, introduced by Donald Knuth in the 1980s, represents a philosophical shift: code is primarily for humans to read, not machines to execute.

- **Self-documenting code**: Rather than adding comments to obscure code, write code that is inherently clear.
- **Structure and intelligibility**: Focus on making the logic obvious from the code itself.
- **Building tools for humans**: We are not just instructing computers-we are building software systems that others must understand, maintain, and extend.
- **Readability over cleverness**: Simple, obvious code is better than clever, obscure code.

Literate Programming Principles

- Choose clear, descriptive names for variables and functions
- Keep functions short and focused on one task
- Structure code to match the logical structure of the problem
- Comments should explain why, not what-the code shows what

28.2 Test-Driven Development (TDD)

Test-Driven Development inverts the traditional development cycle: write tests before writing code.

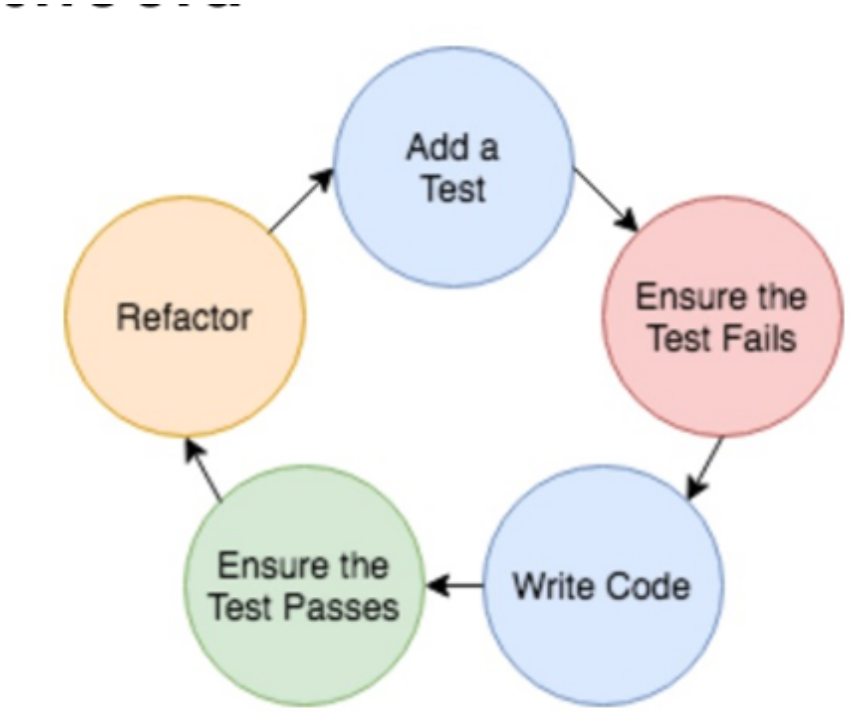

Figure 21: The TDD cycle: Red-Green-Refactor. First write a failing test (Red), then write minimal code to pass the test (Green), then improve the code while keeping tests passing (Refactor). This cycle repeats for each new piece of functionality.

Each test should be short and clear, following the **Given-When-Then** pattern:

- **GIVEN**: What are the initial conditions? (Setup)
- **WHEN**: What action is being performed? (Execution)
- **THEN**: What is the expected behaviour? (Assertion)

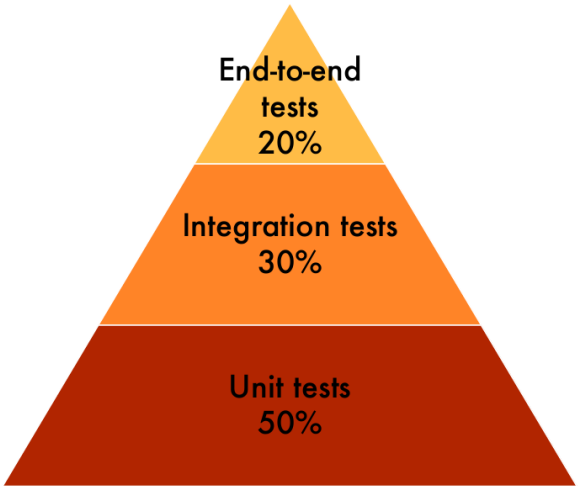

Figure 22: The testing pyramid. **Unit tests** (base): Test individual functions or methods in isolation-fast, numerous, and focused. **Integration tests** (middle): Test that multiple units work together correctly-do outputs of one module align with inputs of another? **End-to-end (E2E) tests** (top): Test the complete application as a user would experience it-slow but comprehensive.

Why TDD?

- **Design pressure**: Writing tests first forces you to think about interfaces and use cases
- **Confidence**: Tests catch regressions when you modify code
- **Documentation**: Tests demonstrate how code is intended to be used
- **Incremental progress**: Each passing test is a small, verified step forward

28.3 Coupling

Coupling measures how strongly different parts of a system depend on each other. High coupling means changes in one module often require changes in others-making the system fragile and hard to maintain.

The Coupling Principle

Minimise coupling between modules.

Low coupling means modules can be:

- Understood independently
- Modified without cascading changes
- Tested in isolation
- Reused in different contexts

High coupling is where bugs hide-interdependencies create unexpected interactions.

Coupling exists on a spectrum from loose (good) to tight (problematic). The following types are ordered from loosest to tightest:

28.3.1 Data Coupling (Loosest)

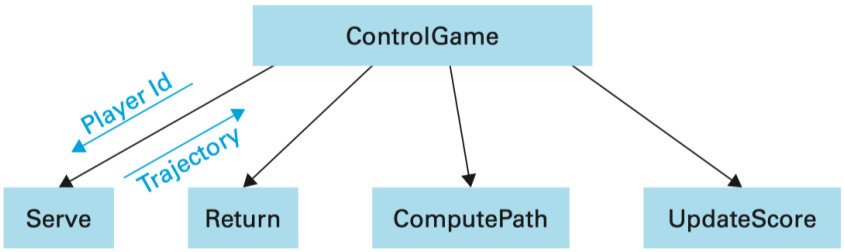

Figure 23: Data coupling: modules share only the data necessary for the task. Module A passes specific values to Module B, which processes them and returns a result. Neither module needs to know about the other's internal structure.

Data coupling occurs when modules communicate by passing only the data needed for the task-nothing more. This is the loosest, most desirable form of coupling.

```
def calculate_tax(income, rate):
    """Data coupling: receives only the values needed."""
    return income * rate

# Calling module passes only required data
tax = calculate_tax(50000, 0.2)
```

28.3.2 Stamp Coupling

Stamp coupling occurs when modules share a composite data structure (e.g., an object or dictionary), but each module uses only part of it.

```
def process_order(order):
    """Stamp coupling: receives entire order, uses only some fields."""
    return order['quantity'] * order['price']
```

```
# The function receives more data than it needs
order = {'customer': 'Alice', 'quantity': 5, 'price': 10.0, 'date': '2024-01-01'}
total = process_order(order)
```

The problem: if the `order` structure changes, all functions receiving it may need review, even if they do not use the changed fields.

28.3.3 Control Coupling

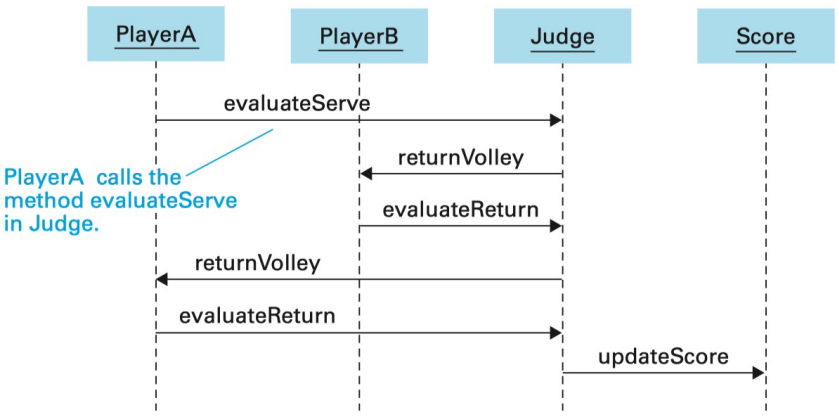

Figure 24: Control coupling: Module A passes a control flag that determines how Module B behaves. Module B must interpret the flag and branch accordingly, creating tighter interdependency than pure data coupling.

Control coupling occurs when one module controls another's behaviour by passing control information (flags, commands, mode parameters).

```
def format_output(data, output_type):
    """Control coupling: behaviour depends on control flag."""
    if output_type == 'json':
        return json.dumps(data)
    elif output_type == 'xml':
        return to_xml(data)
    elif output_type == 'csv':
        return to_csv(data)
```

The receiving module must understand and act on the control information, creating tighter coupling. The caller must know what flags are valid.

28.3.4 Common Coupling

Common coupling occurs when multiple modules share access to the same global data. Changes to the global state can affect any module that uses it, making behaviour unpredictable.

```
# Global variable - source of common coupling
config = {'debug': True, 'max_retries': 3}

def module_a():
    if config['debug']:  # Reads global
        print("Debug mode")
```

```
def module_b():
    config['max_retries'] = 5  # Modifies global - affects module_a!
```

Dangers of Global State

Common coupling through global variables is particularly dangerous because:

- Any module can modify the shared state
- Effects are non-local and hard to trace
- Testing requires careful setup and teardown of global state
- Concurrent access can cause race conditions

28.3.5 Content Coupling (Tightest)

Content coupling occurs when one module directly accesses or modifies another module's internal data or code. This is the tightest, most problematic form of coupling.

```
class ModuleA:
    def __init__(self):
        self._internal_state = 100  # Intended to be private

class ModuleB:
    def manipulate(self, module_a):
        # Content coupling: directly accessing internal state
        module_a._internal_state = 200  # Violates encapsulation!
```

Content coupling violates encapsulation entirely. If `ModuleA`'s internal structure changes, `ModuleB` breaks.

28.4 Cohesion

Cohesion measures how strongly related the elements within a single module are. High cohesion means everything in a module serves a unified purpose-making the module easier to understand, test, and maintain.

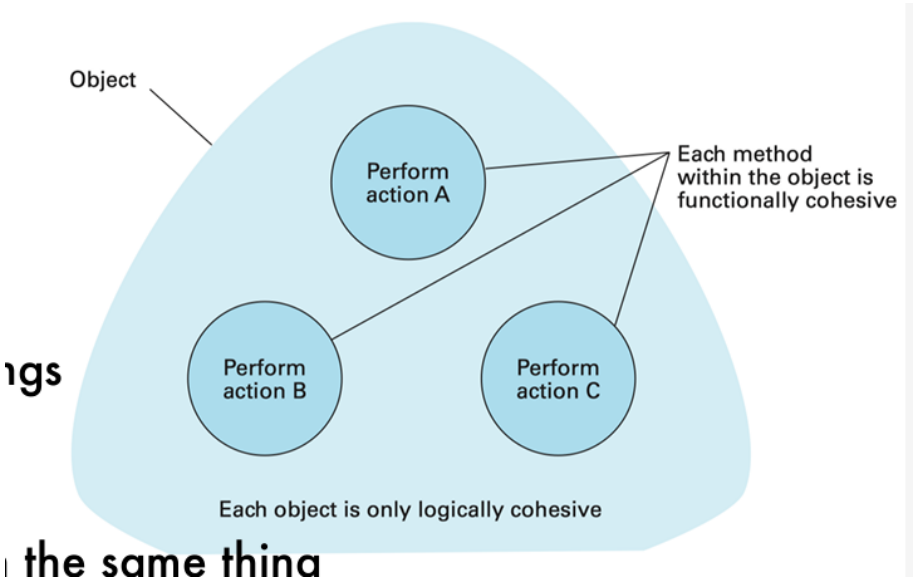

Figure 25: Cohesion spectrum from low (coincidental) to high (functional). Higher cohesion means the module's elements are more closely related in purpose. Aim for the right side of this spectrum.

The Cohesion Principle

Maximise cohesion within modules.

High cohesion means a module does one thing well. Its elements are strongly related, making the module:

- Easy to name (if you struggle to name it, cohesion may be low)
- Easy to understand without examining other modules
- Easy to test with focused test cases
- Easy to reuse in other contexts

Types of cohesion, from lowest (worst) to highest (best):

28.4.1 Coincidental Cohesion (Lowest)

Elements are grouped arbitrarily with no meaningful relationship. This often results from poor organisation or "utility" classes that accumulate unrelated functions.

```
class Utilities:
    """Coincidental␣cohesion:␣unrelated␣functions␣grouped␣together."""
    def parse_date(self, s): ...
    def calculate_tax(self, income): ...
    def send_email(self, to, body): ...
    def compress_image(self, img): ...
```

28.4.2 Logical Cohesion

Elements perform similar categories of tasks but are otherwise unrelated. They are grouped because they "do similar things" at an abstract level.

```
class InputHandler:
    """Logical cohesion: all handle 'input' but are unrelated."""
    def read_file(self, path): ...
    def read_keyboard(self): ...
    def read_network(self, socket): ...
    def read_sensor(self, device): ...
```

28.4.3 Temporal Cohesion

Elements are grouped because they execute at the same time, not because they are logically related.

```
def startup():
    """Temporal cohesion: things that happen at startup."""
    load_config()
    connect_database()
    initialise_cache()
    start_logging()
    warm_up_ml_model()
```

28.4.4 Procedural Cohesion

Elements are grouped because they follow a specific sequence, but each step may serve different purposes.

```
def process_order():
    """Procedural cohesion: follows a sequence."""
    validate_input()
    check_inventory()
    calculate_total()
    process_payment()
    send_confirmation()
```

28.4.5 Communicational Cohesion

Elements operate on the same data but perform different operations on it.

```
class CustomerReport:
    """Communicational cohesion: all work with customer data."""
    def __init__(self, customer):
        self.customer = customer

    def get_purchase_history(self): ...
    def calculate_lifetime_value(self): ...
    def get_preferences(self): ...
```

28.4.6 Sequential Cohesion

Elements form a pipeline where output of one becomes input to the next.

```
class DataPipeline:
    """Sequential␣cohesion:␣data␣flows␣through␣stages."""
    def extract(self, source): ...    # Returns raw data
    def transform(self, raw): ...     # Returns cleaned data
    def load(self, cleaned): ...      # Stores final result
```

28.4.7 Functional Cohesion (Highest)

All elements contribute to a single, well-defined task. This is the ideal.

```
class StackCalculator:
    """Functional␣cohesion:␣everything␣serves␣stack-based␣calculation."""
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        return self.stack.pop()

    def add(self):
        self.push(self.pop() + self.pop())
```

Coupling and Cohesion Trade-off

Coupling and cohesion are related but distinct:

- **High cohesion** (good): Elements within a module are strongly related
- **Low coupling** (good): Modules have minimal dependencies on each other

These goals usually align: modules with high cohesion tend to have cleaner interfaces, reducing coupling to other modules. However, excessive pursuit of cohesion can lead to many tiny modules with high coupling between them. Balance is key.

28.5 Information Hiding

Information hiding, introduced by David Parnas in 1972, is the principle that modules should conceal their internal details from other modules.

The Information Hiding Principle

Not all parts of a system need to know everything.

Each module should:

- Expose a minimal, stable interface
- Hide implementation details that might change
- Protect internal state from external modification

This enables changing a module's internals without affecting code that uses it.

Why hide information?

- **Reduced complexity**: Users of a module need only understand its interface, not its implementation.
- **Change isolation**: Internal changes do not propagate to other modules.
- **Error prevention**: External code cannot corrupt internal state.
- **Modularity**: Hidden details can be optimised or replaced without affecting users.

Global Variables and Information Hiding

Global variables violate information hiding-they expose state to the entire program. Any module can read or modify a global variable, creating implicit dependencies and making behaviour difficult to reason about.

Prefer passing data explicitly through function parameters and return values.

Public vs Private in Python

Python uses naming conventions to indicate visibility:

```
class Model:
    def __init__(self):
        self.public_attribute = 1       # Public: part of the interface
        self._private_attribute = 2     # Private by convention
        self.__mangled_attribute = 3    # Name-mangled (harder to access)

    def fit(self, X, y):
        """Public method: part of the promised interface."""
        self._internal_fit(X, y)

    def _internal_fit(self, X, y):
        """Private method: implementation detail, may change."""
        pass
```

- **No underscore**: Public. The developer promises to maintain this interface.
- **Single underscore** (`_name`): Private by convention. External code should not rely on it.
- **Double underscore** (`__name`): Name-mangled to `_ClassName__name`. Harder to access accidentally, but still not truly private.

Information Hiding and Encapsulation

Information hiding and encapsulation are related but distinct:

- **Information hiding** is a design principle: decide what to reveal and what to conceal
- **Encapsulation** is a mechanism: bundling data and methods, with access controls

Encapsulation is one way to implement information hiding, but information hiding is the broader goal. You can have encapsulation without information hiding (if you expose all internals as public).

29 Connecting Paradigms to Data Structures and Algorithms

Programming paradigms are not merely academic distinctions-they fundamentally shape how we implement and think about data structures and algorithms.

Paradigm-Algorithm Connections

- **Imperative**: Natural for in-place algorithms (sorting arrays, modifying graphs). Loop-based traversals and explicit state management.
- **Functional**: Natural for recursive algorithms (tree traversals, divide-and-conquer). Immutable data structures and compositional design.
- **Object-Oriented**: Natural for encapsulating data structure operations (stack, queue, linked list classes). Polymorphism enables generic algorithms.
- **Declarative**: Natural for query and transformation operations (filtering, mapping, aggregating). Lets the system optimise execution.

Practical implications:

- **Recursion vs iteration**: Functional thinking helps with recursive data structures (trees, linked lists) and algorithms (merge sort, quicksort). Understanding both approaches lets you choose the clearer implementation.
- **Encapsulation and ADTs**: Object-oriented principles underpin abstract data types (ADTs). A stack is defined by its operations (`push`, `pop`, `peek`), not its implementation (array vs linked list).
- **Coupling and algorithm design**: Low-coupling principles guide modular algorithm design. Helper functions should communicate through clean interfaces, not shared state.
- **Cohesion and code organisation**: High-cohesion principles suggest keeping related algorithms together. A `GraphAlgorithms` module containing BFS, DFS, and shortest-path algorithms has communicational cohesion (all operate on graphs).

Understanding paradigms equips you to choose the right tool for each problem and to write code that others can understand, test, and maintain.

Data Structures & Algorithms: Wk 5
Basics of Algorithm Analysis

30 Analysis I: Correctness & Efficiency

When analysing any algorithm, we ask two fundamental questions:

1. **Correctness**
    - Does the algorithm produce the correct output for all valid inputs?
    - Can we guarantee that the assignment we generate is stable?
2. **Efficiency**
    - Can we generate the assignment quickly?
    - Can we do so without using up too much storage space?
    - How does performance scale as input size grows?

30.1 Constant Time Operations

How do we define 'fast'? The key concept is **constant time complexity**, denoted $O(1)$.

Constant Time - $O(1)$

An operation has **constant time complexity** if it takes a fixed amount of time (a fixed number of operations) to complete, regardless of the size of the data structure involved.

More precisely, an operation is $O(1)$ if there exists a constant $c > 0$ such that the operation completes in at most c steps for any input size n.

In this context, 'fast' means that the operation's time will be negligible and largely unaffected by the scale of data. Different data structures have different efficiencies-they are fast at different operations.

Why Constant Time Matters

When building efficient algorithms, we want each individual operation within a loop to be $O(1)$. This allows us to focus solely on how many times the loop executes (as a function of n), rather than worrying about the cost of each iteration varying with input size.

30.1.1 Arrays: Fast Random Access

Arrays store elements in **contiguous** memory locations.
Fast: Retrieving an element by index (random access)
Slow: Adding or removing elements (especially in the middle)
Why Array Indexing is $O(1)$
Retrieving data from an array given an index requires just a few fixed steps:

1. Get the memory location of the first element (base address)
2. Add the index value to that memory location (simple arithmetic)
3. Load data from the computed memory location

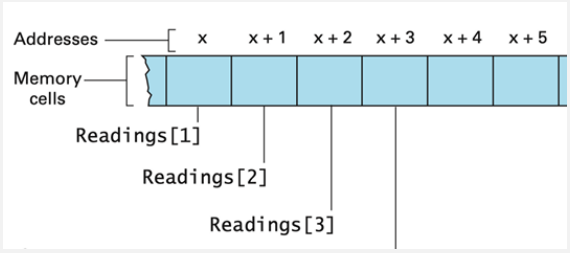

Figure 26: Array indexing: computing the memory address of element i requires only adding the index to the base address-a constant-time operation regardless of array size.

The size of the array doesn't matter: accessing element 5 in a 10-element array takes the same time as accessing element 5 in a 10-million-element array. It's just one addition operation.
But other operations are slow:

- **Adding an element:** Arrays have fixed allocated memory. To add beyond capacity, you must allocate a new larger array and copy all elements-$O(n)$ in the worst case.
- **Removing an element:** Removing from anywhere but the end requires shifting all subsequent elements to fill the gap-also $O(n)$.

30.1.2 Queues: Fast Head/Tail Operations

Queues operate on the First In, First Out (FIFO) principle-you only ever interact with the head (front) and tail (back).
Fast: Adding to tail (enqueue), removing from head (dequeue)
Slow: Accessing an arbitrary element in the middle
Why Enqueue/Dequeue are $O(1)$
Both operations require just a few fixed steps:

1. Find the head/tail pointer
2. Load/store data at the location pointed to by the pointer
3. Update the pointer

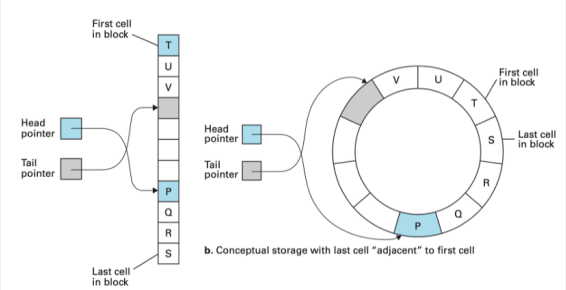

Figure 27: Queue operations: enqueue adds to the tail, dequeue removes from the head. Both operations only interact with the endpoints via pointers, making them $O(1)$ regardless of queue length.

Size independence: The queue's length doesn't affect these operations-we're only interacting with the endpoints, not traversing the structure.
But accessing arbitrary elements is slow: Finding or removing an element in the middle requires iterating through the queue from the head-$O(n)$ in the worst case.

Arrays vs Queues: Complementary Strengths

- **Arrays** are fast ($O(1)$) at accessing **arbitrary** elements when the index is known (random access)
- **Queues** are fast ($O(1)$) at accessing/modifying the **first/last** element without needing to know indices

Arrays excel at the middle (random access); Queues excel at the ends (sequential processing).

This complementarity is crucial for algorithm design: choosing the right data structure for each operation ensures that each step runs in constant time.

31 Analysis II: Big O Notation

Big O notation describes the upper bound of the runtime complexity of an algorithm as a function of the input size n. It captures the **worst-case** growth rate-it's deliberately conservative.

Big O Notation - Formal Definition

A function $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that for all $n > n_0$:

$$T(n) \leq c \cdot f(n)$$

Unpacking this definition:

- $T(n)$ is the actual runtime (or operation count) of the algorithm
- $f(n)$ is a simpler reference function (e.g., n, n^2, $\log n$)
- The constants c and n_0 are 'slack'-we don't care about small inputs or constant factors
- The inequality must hold for all n beyond some threshold n_0

Intuition: From some point onwards, the true runtime $T(n)$ is bounded above by some constant multiple of $f(n)$. We're capturing the order of magnitude of growth, not the exact value.

Big O embeds the idea that what we care about is the big picture: how does runtime scale as n gets large? The constants don't matter-they're implementation details (hardware speed, programming language, etc.).

Efficiency Criterion

An algorithm is 'efficient' if its worst-case runtime grows at a polynomial rate as the input size n increases.

Polynomial growth (n^k for constant k) is considered tractable. Exponential growth (r^n for $r > 1$) quickly becomes impractical for large inputs.

31.1 Why We Drop Constants

With correct implementation, each iteration takes constant time (K operations):

- We set an upper bound on iterations: no more than n^k iterations (where k depends on the algorithm)
- So total operations: no more than $K \times n^k$

Ultimately, we don't care what this constant K is:

- K is 'too deep'-it involves the exact operations of the computer, compiler optimisations, cache behaviour, etc.
- n is what we can change and what varies by data input

This is why we drop constants in Big O notation.

31.2 Choosing the Dominant Term

When expressing Big O, we keep only the highest-order term:

Example 1:

$$T(n) = 32n^2 + 17n + 32$$

- $T(n)$ is $O(n^2)$
- It's also technically $O(n^3)$, $O(n^4)$, etc.-Big O is an upper bound-but we want the most informative (tightest) bound.

Example 2:

$$S(n) = n \cdot T(n-1)$$

- Given $T(n) = O(n^2)$ above, $S(n)$ is $O(n^3)$

Example 3:

$$T(n) = 36n^2 + 5n + 1034 \log n + 7834n \log n$$

- $T(n)$ is $O(n^2)$ (since n^2 dominates $n \log n$)

Example 4:

$$T(n) = 12n + 0.00001 \cdot \exp(n) + n \log n$$

- $T(n)$ is $O(\exp(n))$ (exponential dominates all polynomial terms, no matter how small the constant)

31.3 Common Families of Complexity

Complexity Hierarchy

From fastest to slowest growth (for large n):

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^k) \subset O(2^n) \subset O(n!)$$

31.3.1 Constant Time - $O(1)$

- Operations that don't depend on input size
- Examples: array indexing, hash table lookup (average case), stack push/pop

31.3.2 Logarithmic Time - $O(\log n)$

- For every $d > 0$: $\log n = O(n^d)$ (logarithms grow slower than any polynomial)
- **Base doesn't matter:** $O(\log_a n) = O(\log_b n)$ because change of base introduces only a constant factor: $\log_a n = \frac{\log_b n}{\log_b a}$
- Examples: binary search, balanced tree operations

31.3.3 Linear Time - $O(n)$

- Runtime proportional to input size (after dropping constants)
- Example 1: Finding the maximum of an array-must examine each element once
- Example 2: Merging two sorted lists $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$ into a sorted whole-$O(n + m)$ operations

31.3.4 Quadratic Time - $O(n^2)$

- Typical when enumerating all pairs of elements
- Example 1: Given n points in the plane, find the closest pair by checking all pairs
  - Note: A distance matrix is symmetric, so you'd only need half the comparisons-but this factor of 2 is a constant, which we drop
- Example 2: Examining every element of an $n \times n$ matrix

31.3.5 Polynomial Time - $O(n^k)$

- Example: Given a graph, are there k nodes such that no two are joined by an edge? (Independent set of size k)
  - Approach: Enumerate all k-sized subsets and check each
  - Checking each subset takes $O(k^2)$ (check all pairs within the subset)
  - Number of subsets: $\binom{n}{k} \leq \frac{n^k}{k!}$
  - Total: $O\left(k^2 \cdot \frac{n^k}{k!}\right) = O(n^k)$ (since k is constant)

31.3.6 Exponential Time - $O(r^n)$

- For every $r > 1$ and $d > 0$: $n^d = O(r^n)$ (exponentials dominate all polynomials)
- Exponentials grow extremely fast; typically impractical for $n > 30$–50
- Example: Brute-force enumeration over all subsets (2^n subsets)

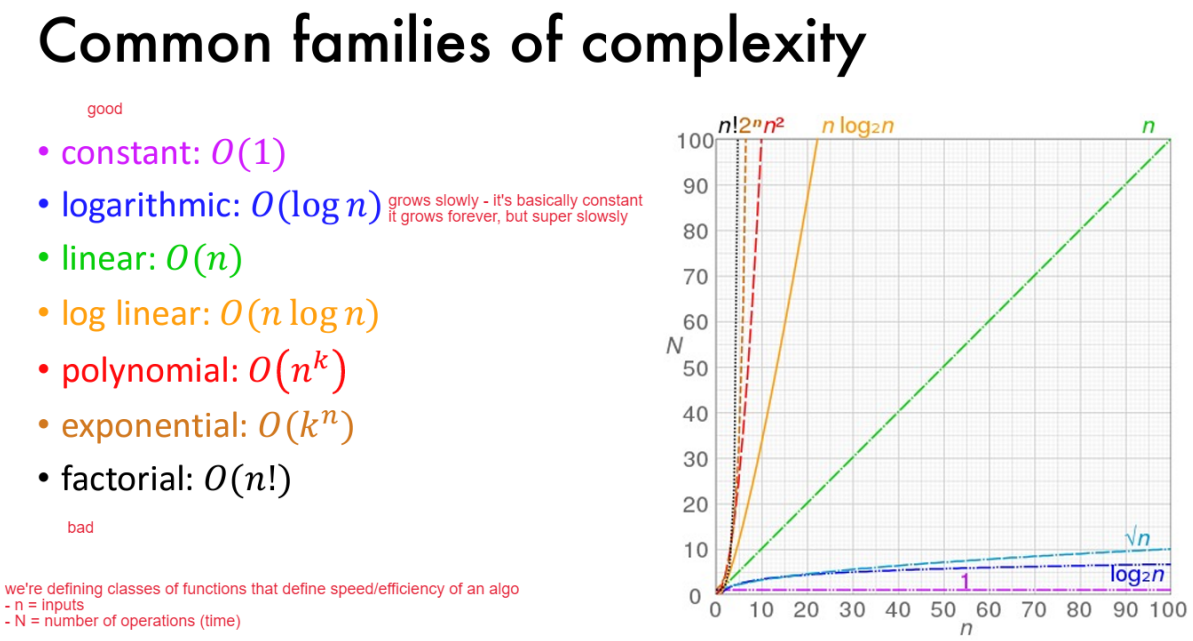

Figure 28: Growth rates of common complexity families. Note how polynomial functions (n, n^2, n^3) grow much more slowly than exponential functions (2^n) as n increases. The vertical axis uses a logarithmic scale to fit all curves; even so, exponential growth quickly dominates.

32 Stable Matching Problem

The stable matching problem is a classic algorithmic problem with wide applicability:

- Matching US medical residents to hospitals (NRMP)
- French students to universities
- Routing internet traffic
- Organ donation matching (e.g., kidney exchanges)

32.1 Problem Setup

Stable Matching Problem

Input:

- A set of n students and n hospitals (or more generally, two disjoint sets)
- Each student ranks all hospitals in order of preference
- Each hospital ranks all students in order of preference

Output: A matching (one-to-one assignment) of students to hospitals.

Goal: The matching must be stable-there should be no unstable pairs.

Unstable pair: A student S and hospital H form an **unstable pair** if:

- S prefers H to their current assigned hospital, AND
- H prefers S to their current assigned student

Essentially, both would agree to abandon their current assignments to be together-this creates instability.

Stable assignment: A matching with no unstable pairs.

32.2 Worked Example

Consider matching three students (Amy, Barry, Charlotte) to three hospitals (Xi'an, York, Zürich).

1. **Students rank hospitals:**

	1st	2nd	3rd
Amy	York	Xi'an	Zürich
Barry	Xi'an	York	Zürich
Charlotte	Xi'an	York	Zürich

Figure 29: Student preference lists: each row shows one student's ranking of hospitals from most preferred (left) to least preferred (right).

2. **Hospitals rank students:**

	1st	2nd	3rd
Xi'an	Amy	Barry	Charlotte
York	Barry	Amy	Charlotte
Zürich	Amy	Barry	Charlotte

Figure 30: Hospital preference lists: each row shows one hospital's ranking of students from most preferred (left) to least preferred (right).

3. **Finding stability is non-trivial:**

   Is (Charlotte → Xi'an), (Barry → York), (Amy → Zürich) stable?

	1st	2nd	3rd
Xi'an	Amy	Barry	Charlotte
York	Barry	Amy	Charlotte
Zürich	Amy	Barry	Charlotte

	1st	2nd	3rd
Amy	York	Xi'an	Zürich
Barry	Xi'an	York	Zürich
Charlotte	Xi'an	York	Zürich

Figure 31: An **unstable** matching: Xi'an would prefer Barry over Charlotte, and Barry would prefer Xi'an over York. Both would benefit from switching-this is an unstable pair.

Is (Amy → Xi'an), (Barry → York), (Charlotte → Zürich) stable?

	1st	2nd	3rd
Xi'an	Amy	Barry	Charlotte
York	Barry	Amy	Charlotte
Zürich	Amy	Barry	Charlotte

	1st	2nd	3rd
Amy	York	Xi'an	Zürich
Barry	Xi'an	York	Zürich
Charlotte	Xi'an	York	Zürich

Figure 32: A **stable** matching: no student-hospital pair would mutually prefer to switch. For any potential pair, at least one party prefers their current assignment.

33 Stable Roommate Problem: A Cautionary Example

Not every matching problem has a stable solution! The **stable roommate problem** demonstrates this.

33.1 Problem Setup

- $2n$ people, each with a complete ranking over all others
- Goal: Pair everyone into roommate pairs with no unstable pairs

	1st	2nd	3rd
Amy	Barry	Charlotte	Darren
Barry	Charlotte	Amy	Darren
Charlotte	Amy	Barry	Darren
Darren	Amy	Barry	Charlotte

Figure 33: The stable roommate problem with 4 people: no stable pairing exists. For example, if we pair (A–B) and (C–D), then B and C form an unstable pair. Every possible pairing has at least one unstable pair-see slides for all permutations.

No Solution Exists!

Unlike the stable matching problem (with two distinct groups), the stable roommate problem (matching within a single group) may have **no stable solution at all**.

This highlights why the bipartite structure of the hospital-student problem matters: the Gale-Shapley algorithm below exploits this structure to guarantee a stable solution always exists.

34 Gale-Shapley Algorithm

The Gale-Shapley algorithm (also called the **Deferred Acceptance algorithm**) solves the stable matching problem. The key insight is that we assign tentative matches and only know the final stable solution at the end-hence 'deferred'.

Gale-Shapley Algorithm

Algorithm 1: Gale-Shapley Algorithm

```
while some student is free and hasn't applied to every hospital do
    Choose such a student S;
    H ← first hospital on S's list to whom S hasn't yet applied;
    if H is free then
        Tentatively assign S and H as matched;
    else if H prefers S to its current student S' then
        Tentatively assign S and H as matched;
        Set S' to be free;
    else
        H rejects S (and S moves to next hospital on list);
    end
end
```

Key insight: Students propose (in order of their preferences, from top to bottom), and hospitals respond. This gives the proposing side an advantage in the final stable matching.

34.1 Correctness Analysis

We prove correctness in three parts: the algorithm terminates, everyone gets matched, and all matches are stable.

34.1.1 Does the Algorithm Halt?

- Students propose to hospitals in decreasing order of preference (they never revisit a hospital)
- **Once a hospital is matched, it never becomes unmatched**-it only 'trades up' to a more preferred student
- This monotonicity puts a bound on the algorithm

Claim: The algorithm halts after at most n^2 iterations.

Reasoning:

- Outer bound: n students
- Inner bound: each student applies to at most n hospitals
- Worst case: each student applies to every hospital before all are matched

Gale-Shapley Complexity

The Gale-Shapley algorithm runs in $O(n^2)$ iterations (and $O(n^2)$ time with proper implementation-see Section 34.2).

34.1.2 Does Everyone Get Matched?

Claim: All students and hospitals get matched.

Proof (by contradiction):

1. Suppose, for contradiction, that some student S is never matched when the algorithm terminates
2. Then some hospital H must also be unmatched (since there are equal numbers)
3. But if H is unmatched and free, it was never applied to
4. But S applied to every hospital (since S remained unmatched and kept proposing)
5. Contradiction! Therefore, everyone gets matched.

34.1.3 Are All Matches Stable?

Claim: The final matching contains no unstable pairs.

Proof (by contradiction):

Suppose (S, H) is an unstable pair in the final matching-meaning S is matched to H^* (not H), H is matched to S^* (not S), yet both S and H would prefer each other.

Two cases:

Case 1: S never applied to H

- Students apply in order of preference, from top to bottom
- If S never applied to H, then S was matched before reaching H on their list
- Therefore, S prefers their current match H^* to H
- Contradiction: (S, H) is not unstable

Case 2: S applied to H

- H rejected S (either immediately or later when trading up)
- H only rejects in favour of a more preferred student
- Therefore, H prefers their final match S^* to S
- Contradiction: (S, H) is not unstable

In both cases, we reach a contradiction. Therefore, no unstable pairs exist.

The Logic:

- **Students go down in quality:** They systematically try their top choices first, settling for less preferred options only when rejected.
- **Hospitals only increase in quality:** They can only trade up to more preferred students, never down.
- **Proposer advantage:** The proposing side (students) ends up with their best possible stable partner, while the receiving side (hospitals) ends up with their worst stable partner (among all stable matchings).

34.2 Efficient Implementation

We've established that Gale-Shapley runs in $O(n^2)$ iterations. But how long does each iteration take?

The goal: Each iteration should take $O(1)$ time, regardless of n.

This is where our earlier discussion of arrays vs queues becomes crucial:

- We need $O(1)$ access to find the next hospital a student should apply to
- We need $O(1)$ access to compare which of two students a hospital prefers
- We need $O(1)$ operations to manage the pool of unmatched students

34.2.1 The Challenge: Comparing Preferences

When a hospital H must decide between its current student S' and a new applicant S, it needs to quickly determine which it prefers.

If we store preferences as a simple list (e.g., H's preferences = [Amy, Barry, Charlotte]), then determining whether H prefers S to S' requires searching through the list-$O(n)$ per comparison!

34.2.2 The Solution: Inverse Preference Tables

Create an **inverse preference table** for each hospital: instead of listing students in preference order, store each student's rank directly.

	1st	2nd	3rd
Xi'an	Amy	Barry	Charlotte
York	Barry	Amy	Charlotte
Zürich	Amy	Barry	Charlotte

	Amy	Barry	Charlotte
Xi'an	1	2	3
York	2	1	3
Zürich	1	2	3

Figure 34: Inverse preference table: for each hospital, we store each student's rank directly. Now comparing preferences is $O(1)$: just compare inverse[S] vs inverse[S'].

Note: This is still the hospital's preference information-we're just reorganising it for efficient lookup. It tells us nothing about students' preferences.

Example:

- inverse[Amy] at Xi'an = 1 (Amy is Xi'an's first preference)
- inverse[Barry] at Xi'an = 2
- To check if York prefers Amy or Barry: compare inverse[Amy] = 2 vs inverse[Barry] = 1
- Since $1 < 2$, York prefers Barry

This makes comparison $O(1)$: direct array indexing instead of list searching.

34.2.3 Complete Data Structure Design

Figure 35: Complete efficient implementation: combining queues for managing unmatched students with arrays for preference lookups. The key line "if H prefers S to current student S'" becomes a single $O(1)$ array comparison using the inverse preference table.

Analysing each operation:

- **Get next unmatched student:** Dequeue from head-$O(1)$
- **Get student's next hospital to try:** Array index into student's preference list-$O(1)$
- **Check if hospital is free:** Array lookup-$O(1)$
- **Compare student preferences for hospital:** Inverse table lookup-$O(1)$
- **Update matching arrays:** Array assignment-$O(1)$
- **Return rejected student to pool:** Enqueue at tail-$O(1)$

Every operation is $O(1)$, so each iteration is $O(1)$.

Why the Queue Matters

What if we used a boolean array instead of a queue to track unmatched students?

`is_matched[S] = True` if student S is matched.

Each iteration would require scanning the entire array to find an unmatched student-$O(n)$ per iteration. With n^2 iterations, total time becomes $O(n^3)$ instead of $O(n^2)$.

The queue ensures we always have $O(1)$ access to an unmatched student.

34.3 Final Analysis

Gale-Shapley Algorithm Summary

With correct implementation using arrays (for $O(1)$ random access) and queues (for $O(1)$ head/tail operations):

- At most n^2 iterations
- Each iteration takes constant time K (some fixed number of $O(1)$ operations)
- Total: at most $K \times n^2$ operations

Time complexity: $O(n^2)$

We don't care about the constant K-it depends on implementation details (hardware, language, etc.). We care about n, which varies with the problem size.

This is why we drop constants in Big O notation.

Summary of Gale-Shapley Analysis:

1. **Correctness:** Proven via three claims-termination, complete matching, stability
2. **Efficiency:** $O(n^2)$ time with proper data structures
3. **Key insight:** Choosing the right data structure for each operation (arrays for random access, queues for sequential processing, inverse tables for preference comparison) ensures each iteration is $O(1)$

Data Structures & Algorithms: Wk 6
More Algorithm Analysis

This week builds on the foundations of Big O notation from Week 5, presenting formal properties of asymptotic analysis and applying them through four detailed case studies. The key pedagogical goal is to develop the skill of algorithmic thinking: recognising structure in problems and exploiting it to improve efficiency.

35 Properties of Big O

Recall from Week 5 that Big O notation captures the upper bound on an algorithm's growth rate. These properties allow us to manipulate and combine Big O expressions algebraically, which is essential for analysing complex algorithms built from simpler components.

Formal Properties of Big O Notation

Let f, f_1, f_2, g, g_1, g_2, and h be functions from $\mathbb{N} \to \mathbb{R}^+$. The following properties hold:

1. Reflexivity: f is $O(f)$.

Every function is an upper bound for itself. Formally, we can choose $c = 1$ and any $n_0 \geq 1$ in the Big O definition.

2. Constants are absorbed: If f is $O(g)$ and $c > 0$, then cf is $O(g)$.

Multiplying a function by a positive constant doesn't change its complexity class. This formalises why we "drop constants" in Big O analysis. If $f(n) \leq c_1 \cdot g(n)$ for large n, then $cf(n) \leq (c \cdot c_1) \cdot g(n)$, which is still $O(g)$.

3. Products combine multiplicatively: If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 \cdot f_2$ is $O(g_1 \cdot g_2)$.

This is crucial for analysing nested loops: if an outer loop runs $O(n)$ times and each iteration does $O(n)$ work, the total is $O(n \cdot n) = O(n^2)$.

4. Sums take the maximum: If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.

When adding complexities, only the dominant term matters. This is why $O(n^2 + n) = O(n^2)$-the quadratic term dominates.

5. Transitivity: If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

Upper bounds compose: if f grows no faster than g, and g grows no faster than h, then f grows no faster than h. This allows us to chain bounds through intermediate functions.

These properties are not just theoretical-they are the tools we use repeatedly when analysing algorithms. The product rule handles nested structures; the sum rule handles sequential code blocks; transitivity lets us substitute bounds we've already established.

36 Algorithm Analysis Guidelines

Before diving into case studies, here is a general framework for improving algorithms:

The Iterative Improvement Process

1. **Start simple:** Implement the naive brute-force solution first
2. **Analyse weaknesses:** Identify why it is inefficient-what work is being repeated? What structure is being ignored?
3. **Exploit structure:** Find properties of the problem that allow shortcuts
4. **Iterate:** Repeat until performance meets requirements

Questions to guide analysis:

- What are the specifics of the problem? What constraints exist?
- How can I make the problem easier for myself?
- Is there exploitable structure? (The same patterns recur across many problems)
- What data do I actually need to keep? Can I work with a subset?
- How can I efficiently search through the solution space?
- Does the data need to be sorted? Can I create structure that makes later operations faster?
- Does this resemble a problem I've seen before with a known efficient solution?

The following four case studies demonstrate this process in action.

37 Case Study 1: Fibonacci Sequence

The Fibonacci sequence is defined recursively:

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0, \quad F(1) = 1$$

The first few terms are: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

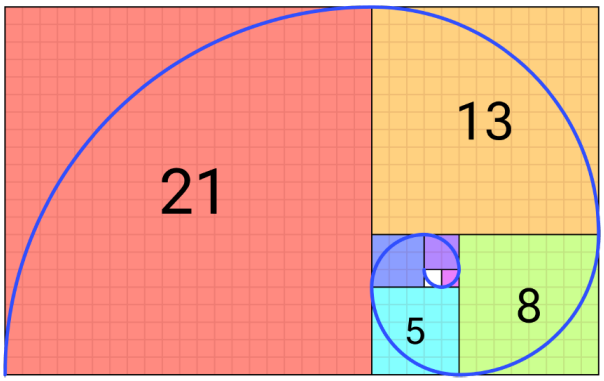

Figure 36: The Fibonacci sequence: each term is the sum of the two preceding terms. This simple recurrence relation appears throughout mathematics, computer science, and nature.

37.1 Naive Approach: Direct Recursion

The most straightforward implementation directly translates the mathematical definition into code:

Algorithm 2: Naive Recursive Fibonacci

```
def fib1(n):
    if n ≤ 1 then
        return n;
    end
    return fib1(n − 2) + fib1(n − 1);
```

How it works:

- Base cases: $F(0) = 0$ and $F(1) = 1$ are returned directly
- Recursive case: for $n \geq 2$, compute both predecessors and sum them
- This is a direct translation of the mathematical definition (see functional programming principles in Week 4)

37.1.1 Analysing Recursion via Recursion Trees

To understand the time complexity, we visualise the function calls as a **recursion tree**:

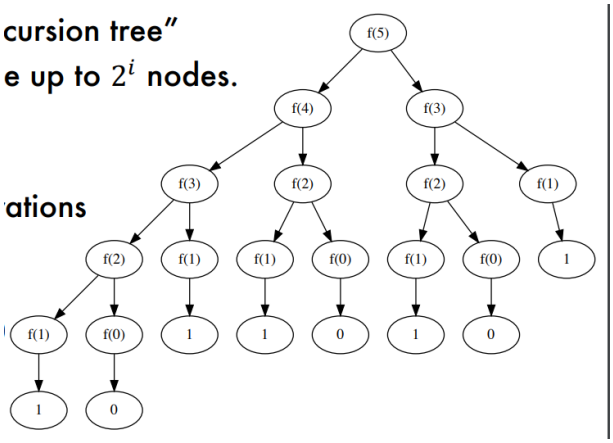

Figure 37: Recursion tree for computing $F(n)$. Each node represents a function call; children represent the recursive calls made. The tree grows exponentially wide, and the same values (e.g., $F(2)$, $F(3)$) are computed many times at different positions.

Analysing the tree structure:

- At level i (counting from 0 at the root), there are up to 2^i nodes
- The tree has depth approximately n (from level 0 to level $n - 1$)
  - Level 0: the initial call $F(n)$
  - Level $n - 1$: the deepest calls reaching the base cases
- Total nodes: up to $2^n - 1$ (a complete binary tree)

- Each node does constant work: one addition, one comparison

Complexity of Naive Fibonacci

Time complexity: $O(2^n)$

The recursion tree has up to $2^n - 1$ nodes, each performing $O(1)$ work. Therefore, total time is $O(2^n)$-exponential in the input.

Space complexity: $O(n)$

Space is determined by the maximum depth of the call stack at any moment, which equals the tree's height. The deepest path has n levels, so at most n stack frames exist simultaneously. Each frame stores local variables and the return address-constant space per frame.

The Hidden Inefficiency

The fundamental problem with this approach is **redundant computation**: the same Fibonacci values are calculated many times.

For example, when computing $F(5)$:

- $F(3)$ is computed twice
- $F(2)$ is computed three times
- $F(1)$ is computed five times

We are paying exponential time to compute values we already know. The recursive structure forgets what it has already computed.

37.2 Improvement 1: Memoisation (Caching Values)

The insight: **store computed values so we only calculate each $F(k)$ once**.

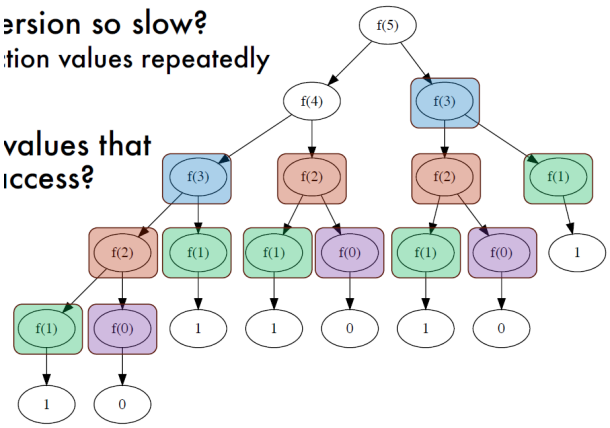

Figure 38: Memoisation visualised: instead of recomputing values, we store them in an array. Each Fibonacci number is computed exactly once and retrieved from the cache thereafter. The tree effectively collapses to a linear chain.

Algorithm 3: Iterative Fibonacci with Memoisation

```
def fib2(n):
    if n ≤ 1 then
        return n;
    end
    arr = [0] * (n + 1);
    arr[1] = 1;
    for i in range (2, n + 1) do
        arr[i] = arr[i − 2] + arr[i − 1];
    end
    return arr[n];
```

How it works:

- Create an array of size $n + 1$, initialised to zeros
- Set the base case: $arr[1] = 1$ (and $arr[0] = 0$ is already set)
- Iterate from $i = 2$ to n, computing each Fibonacci number from the previous two
- Each $F(k)$ is computed exactly once and stored for future reference

This approach is called **dynamic programming**-solving a problem by breaking it into overlapping subproblems and storing their solutions.

Complexity of Memoised Fibonacci

Time complexity: $O(n)$

The loop executes exactly $n - 1$ times. Each iteration performs:

- Two array lookups: $O(1)$ each (random access)
- One addition: $O(1)$
- One array assignment: $O(1)$

Total: $O(n)$ iterations $\times$ $O(1)$ per iteration $= O(n)$.

Space complexity: $O(n)$

We allocate an array of size $n + 1$ to store all intermediate values.

We have reduced time complexity from exponential to linear-a dramatic improvement! For $n = 50$, the naive approach would require approximately $2^{50} \approx 10^{15}$ operations, while the memoised version needs only about 50.

Room for Further Improvement

We're storing the entire array of $n+1$ values, but look closely at the algorithm: to compute $F(i)$, we only ever access $F(i - 1)$ and $F(i - 2)$. We never look back further than two positions.

We're using $O(n)$ space to store values we'll never need again.

37.3 Improvement 2: Constant Space

Since we only need the last two values at any point, we can reduce space from $O(n)$ to $O(1)$:

Algorithm 4: Space-Optimised Iterative Fibonacci

```
def fib3(n):
    if n ≤ 1 then
        return n;
    end
    a, b = 0, 1;
    for i in range (2, n + 1) do
        c = a + b;
        a = b;
        b = c;
    end
    return b;
```

How it works:

- Maintain only three variables: $a = F(i-2)$, $b = F(i-1)$, $c = F(i)$
- After computing $c = a + b$, shift the window: $a \leftarrow b$, $b \leftarrow c$
- At iteration i: a holds $F(i-2)$, b holds $F(i-1)$, and we compute $c = F(i)$

Complexity of Space-Optimised Fibonacci

Time complexity: $O(n)$

Same as before: one loop of $n-1$ iterations, each doing $O(1)$ work.

Space complexity: $O(1)$

We use only three variables (a, b, c) regardless of input size-constant space.

Fibonacci Optimisation Journey

Approach	Time	Space
Naive recursion	$O(2^n)$	$O(n)$
Memoisation	$O(n)$	$O(n)$
Space-optimised	$O(n)$	$O(1)$

Key insights:

1. **Avoid redundant work:** Memoisation eliminates repeated computation
2. **Keep only what you need:** Recognising that only the last two values matter reduces space from $O(n)$ to $O(1)$

This pattern-identify repeated work, cache results, then minimise cache size-recurs throughout algorithm design.

38 Case Study 2: Binary Search

Problem: Given a **sorted** array and a target value, find the index of the target (or determine it's not present).

Example: Find element 33 in a sorted array.

38.1 Naive Approach: Linear Search

The simplest approach: examine every element until we find the target.

Figure 39: A sorted array. Linear search would check each element from left to right until finding the target-potentially examining all n elements.

Analysis: This takes $O(n)$ time in the worst case (target is last or absent). It completely ignores the fact that the array is sorted!

Wasted Structure

The array is **sorted**-this is valuable structure we're not exploiting. When we look at an element and it's too small, we know the target must be to the right. When it's too big, the target must be to the left.

Linear search ignores this, treating a sorted array no differently than an unsorted one.

38.2 Binary Search: Divide and Conquer

Key insight: Compare the target against the middle element. This immediately eliminates half the search space.

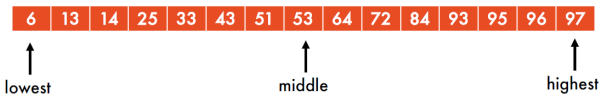

Figure 40: Binary search setup: identify the lowest index (`lo`), highest index (`hi`), and middle index (`mid`). We compare the target against the element at `mid`.

The algorithm: Compare the target against the middle element:

- If the target is smaller → search the left half
- If the target is larger → search the right half
- If equal → found!

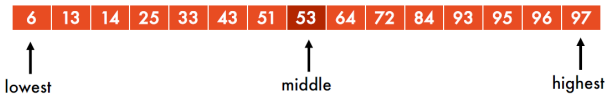

Figure 41: First comparison: target 33 is greater than middle element 27, so we eliminate the left half and search only the right portion.

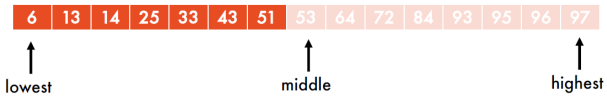

Figure 42: Second comparison: update `lo` to just past the old middle. The search space is now half its original size.

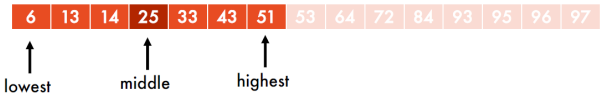

Figure 43: Find the new middle of the reduced search space. Each iteration halves the remaining elements.

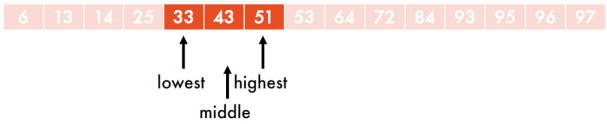

Figure 44: Third comparison: target 33 is less than middle element 43, so we eliminate the right half and search only the left portion.

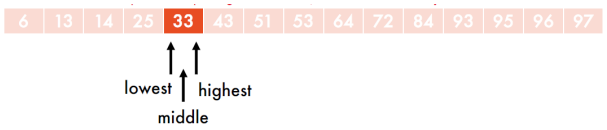

Figure 45: Final state: all three pointers (`lo`, `mid`, `hi`) converge on the same element. If this element equals the target, we've found it; otherwise, the target is not in the array.

38.2.1 Algorithm

Algorithm 5: Binary Search Algorithm

```
def binarySearch(A, x):
    lo = 1;
    hi = n;
    while lo ≤ hi do
        mid = floor((lo + hi)/2);
        if x < A[mid] then
            hi = mid − 1;
        else
            if x > A[mid] then
                lo = mid + 1;
            else
                return mid;
            end
        end
    end
    return −1;
```

Parameters:

- A: the sorted array to search
- x: the target value
- Returns: index of x in A, or -1 if not found

38.2.2 Correctness

Loop invariant: If x is in A, then x is in $A[lo : hi]$ (the current search range).

Termination:

- The quantity $(hi - lo)$ strictly decreases each iteration (we always shrink the range)
- Eventually $hi < lo$, and the loop exits

Correctness:

- If we find $A[mid] = x$, we return the correct index
- If the loop exits with $hi < lo$, the search space is empty, so $x \notin A$

38.2.3 Efficiency

Binary Search Complexity

Time complexity: $O(\log n)$

After k iterations, the remaining search space has size at most:

$$(hi - lo + 1) \leq \frac{n}{2^k}$$

The algorithm terminates when this becomes less than 1, i.e., when:

$$\frac{n}{2^k} < 1 \implies 2^k > n \implies k > \log_2 n$$

Therefore, the maximum number of iterations is $\lfloor \log_2 n \rfloor + 1 = O(\log n)$.

Each iteration does $O(1)$ work (one comparison, one arithmetic operation, one array access).

Space complexity: $O(1)$

We only maintain a constant number of variables: `lo`, `hi`, `mid`.

Binary Search: Key Result

Binary search achieves $O(\log n)$ time complexity-dramatically faster than linear search's $O(n)$.

The power of logarithms: To search 1 billion elements:

- Linear search: up to 1,000,000,000 comparisons
- Binary search: at most $\log_2(10^9) \approx 30$ comparisons

This is a factor of 33 million improvement!

38.2.4 Applications of Binary Search

Binary search is a **canonical building block** that appears throughout computer science:

- **Dictionary lookup:** Finding a word in a physical dictionary (open to the middle, decide which half)
- **Debugging code:** Is the bug before or after line K? Repeat with binary elimination
- **Resource allocation:** Exponential backoff in networking-try a value, if too high/low, adjust
- **Database indexing:** B-trees use binary search within nodes
- **Game playing:** Twenty questions is essentially binary search over possibilities

Binary Search Requires Sorted Data

Binary search **only works on sorted data**. If the input is unsorted, you must either:

1. Sort it first (typically $O(n \log n)$), then binary search
2. Use linear search ($O(n)$)

For a single search, linear search ($O(n)$) beats sort-then-binary-search ($O(n \log n + \log n)$). But for many searches on the same data, sorting once and searching many times is far more efficient.

39 Case Study 3: Sorting

Sorting is one of the most fundamental algorithmic problems. Unlike the previous examples where we were exploiting existing structure, here we are creating structure: transforming an unordered list into an ordered one.

Figure 46: The sorting problem: given an unordered array (top), produce a sorted array (bottom) with elements arranged from least to greatest.

39.1 Naive Approach: BogoSort

The simplest conceivable "algorithm": guess and check.

Algorithm concept:

1. Randomly permute the array
2. Check if it's sorted
3. If not, repeat

Figure 47: BogoSort: randomly shuffle the array and check if it's sorted. This is extremely unlikely to produce the correct order quickly.

Correctness: There are exactly $n!$ possible permutations of n elements. One of them is the sorted order. Eventually (with probability 1), we'll stumble upon it.

BogoSort Complexity

Time complexity: $O(n!)$ **expected**

- There are $n!$ possible permutations
- Each random shuffle has probability $\frac{1}{n!}$ of being correct
- Expected number of shuffles: $n!$
- Checking if sorted takes $O(n)$ (one pass through the array)
- Total expected time: $O(n \cdot n!) = O(n!)$ (since $n!$ dominates)

Space complexity: $O(1)$

In-place shuffling requires only constant extra space.

Factorial Time is Catastrophically Slow

To appreciate how bad $O(n!)$ is, consider Stirling's approximation:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

This shows $n!$ is $O(n^n)$-combining both polynomial and exponential growth!

For just $n = 20$ elements: $20! \approx 2.4 \times 10^{18}$ operations.

Even at 1 billion operations per second, this would take about 77 years. For $n = 25$, it would take longer than the age of the universe.

Why is BogoSort so terrible? It treats sorting as pure guesswork, ignoring all structure. It doesn't consider:

- What properties a sorted list has
- How individual swaps might move us toward a solution
- How to systematically build up the correct order

39.2 MergeSort: Divide and Conquer

MergeSort exploits a key insight: **merging two sorted lists is fast** ($O(n)$). The strategy is to recursively divide the problem until we have trivially sorted sublists (single elements), then merge them back together.

Background: Merging Sorted Lists is $O(n)$

Given two sorted lists $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$, we can merge them into a single sorted list in $O(n+m)$ time.

```
Merge(A, B):
    i = 0, j = 0
    C = [0] * (n + m)
    while (i < n and j < m):
        if (A[i] <= B[j]):
            C[i+j] = A[i]
            i = i + 1
        else:
            C[i+j] = B[j]
            j = j + 1
    # Copy remaining elements from whichever list is not exhausted
```

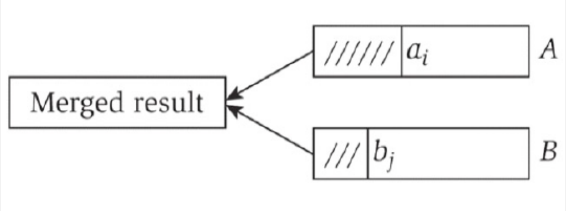

Figure 48: Merging two sorted lists: maintain pointers to the current element of each list. At each step, compare the two current elements and append the smaller to the output. The key insight is that we only ever compare "current" elements-we never need to look back.

Why it's $O(n+m)$:

- We process each element exactly once
- Each comparison moves one element to the output
- Each operation (comparison, copy, increment) is $O(1)$

This is only possible because the lists are sorted! If they weren't, we'd need to search for the minimum element each time, making it $O((n+m)^2)$.

39.2.1 MergeSort Algorithm

MergeSort works in two phases:

1. **Divide:** Recursively split the array into halves until reaching single-element arrays (which are trivially sorted)
2. **Conquer:** Merge the sorted subarrays back together, building up larger sorted arrays

Step-by-step example:

1. **Initial split into pairs:** Compare adjacent pairs and order them.

Figure 49: First pair: compare M and E. Since E < M alphabetically, swap to get (E, M).

Figure 50: Result of first pair comparison: E and M are now in sorted order.

Figure 51: Second pair: compare R and G. Since G < R, swap to get (G, R).

Figure 52: Result: both pairs in the first half are now sorted.

2. **Merge sorted pairs into sorted quadruples:**

Figure 53: Merge (E, M) and (G, R) into (E, G, M, R). We only compare current elements of each list-this is why merging is efficient.

3. **Continue with the other half:**

Figure 54: Sort the next pair: compare S and E to get (E, S).

Figure 55: Sort another pair: compare O and R to get (O, R).

Figure 56: Merge (E, S) and (O, R) into (E, O, R, S).

4. **Merge sorted quadruples:**

Figure 57: Merge (E, G, M, R) and (E, O, R, S) into (E, E, G, M, O, R, R, S)-the first half is now fully sorted.

5. **Process the second half similarly:**

Figure 58: Begin sorting the second half of the original array.

Figure 59: Sort pairs in the second half.

Figure 60: Merge sorted pairs into quadruples.

Figure 61: Continue merging in the second half.

Figure 62: Merge quadruples in the second half.

6. **Continue building larger sorted sublists:**

Figure 63: Both halves are now sorted. One final merge remains.

Figure 64: Performing the final merge operation.

Figure 65: Final result: the entire array is now sorted.

39.2.2 Correctness

- **Base case:** Single-element arrays are trivially sorted
- **Inductive step:** If two subarrays are sorted, merging them produces a sorted result
- **Termination:** Each recursive call works on a strictly smaller array; eventually we reach single elements

39.2.3 Efficiency

MergeSort Complexity

Time complexity: $O(n \log n)$

Analysis via recursion levels:

- At level k (counting from 0 at the leaves), we have 2^k subarrays
- Each subarray at level k has size $n/2^k$
- Merging all subarrays at level k processes all n elements exactly once: $O(n)$ work per level
- Number of levels: $\log_2 n$ (we halve the problem size each time)
- Total work: $O(n) \times O(\log n) = O(n \log n)$

Alternative view-recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

(Two subproblems of half size, plus linear merge time.) This recurrence solves to $T(n) = O(n \log n)$.

Space complexity: $O(n)$

Merging requires a temporary array to store the merged result.

Sorting Comparison

Algorithm	Time	Space
BogoSort	$O(n!)$	$O(1)$
MergeSort	$O(n \log n)$	$O(n)$

Key insight: MergeSort exploits the structure of sorted sublists. Rather than random guessing, it systematically builds up sorted portions and efficiently combines them.

Note: $O(n \log n)$ is asymptotically optimal for comparison-based sorting-no algorithm that compares elements pairwise can do better in the worst case. This is proven via information-theoretic arguments.

40 Case Study 4: Sieve of Eratosthenes

Problem: Find all prime numbers up to n.

A prime number $p > 1$ has no divisors other than 1 and itself.

40.1 Naive Approach: Trial Division

For each number k from 2 to n, check if any number from 2 to $k - 1$ divides it evenly.

Analysis:

- For each of n numbers, we perform up to n divisions
- Total: $O(n^2)$ operations

Slight improvement: We only need to check divisors up to $\sqrt{k}$, since if $k = a \times b$ with $a \leq b$, then $a \leq \sqrt{k}$. This reduces to $O(n\sqrt{n})$, but we can do much better.

40.2 Sieve of Eratosthenes

The Sieve of Eratosthenes exploits the structure of prime numbers: **every composite number is a multiple of some smaller prime**.

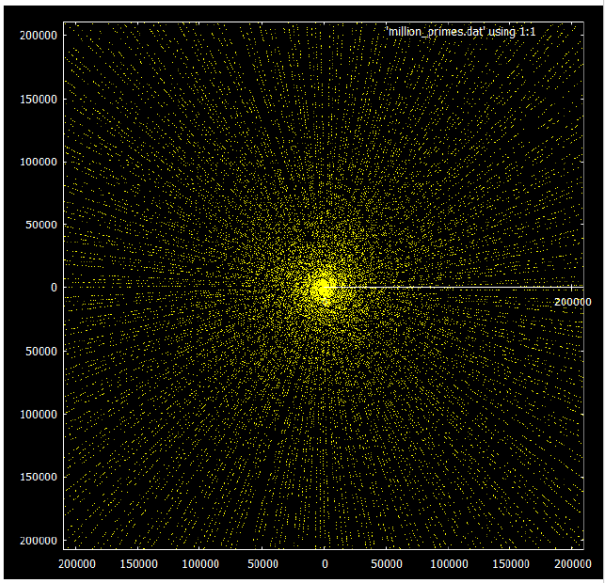

Figure 66: Prime numbers plotted in polar coordinates with radius $= p$ and angle $= p$ radians. The emerging spiral patterns reveal deep structure in the distribution of primes. The Sieve of Eratosthenes exploits a different structural property: multiples of primes.

40.2.1 Algorithm

1. Create a list of all integers from 2 to n, initially all unmarked (potentially prime)
2. Start with the smallest unmarked number $p = 2$
3. Mark all multiples of p (starting from p^2) as composite
    - We start at p^2 because smaller multiples $(2p, 3p, \ldots, (p-1)p)$ were already marked by smaller primes
4. Find the next unmarked number; set it as the new p
5. Repeat until $p > \sqrt{n}$ (all remaining unmarked numbers are prime)

Why only check up to $\sqrt{n}$? If a number $m \leq n$ is composite and its smallest prime factor is p, then $p \leq \sqrt{m} \leq \sqrt{n}$. So all composites are marked by the time we've processed all primes up to $\sqrt{n}$.

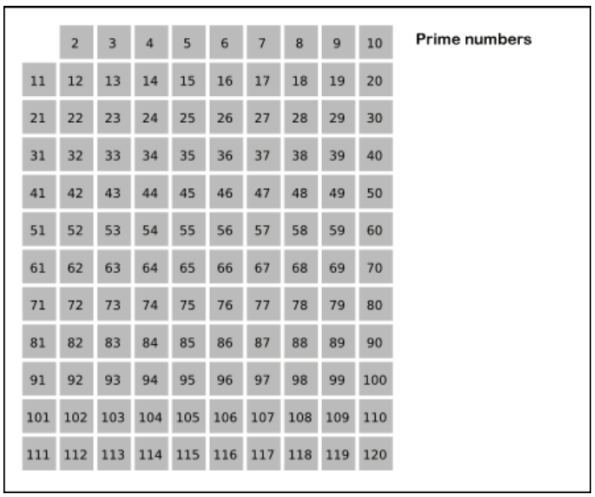

Figure 67: The Sieve of Eratosthenes in action (see `https://upload.wikimedia.org/wikipedia/commons/9/94/Animation_Sieve_of_Eratosth.gif` for an animated version). Starting from 2, we mark all its multiples. Then find 3 (unmarked), mark its multiples. Then 5, 7, etc. Numbers remaining unmarked are prime.

Structural insight: As p increases, we mark fewer and fewer new numbers. Multiples of 2 eliminate half the candidates; multiples of 3 eliminate a third of what remains; and so on. We're leveraging the fact that the definition of primality encodes information about multiples.

40.2.2 Correctness

A number remains unmarked if and only if it has no prime factors less than itself-i.e., if and only if it is prime.

- If m is composite, it has a prime factor $p < m$, and m is marked when we process p
- If m is prime, no smaller prime divides it, so it is never marked

40.2.3 Efficiency

Sieve of Eratosthenes Complexity

Time complexity: $O(n \log \log n)$

This is a subtle analysis. The number of operations is:

$$\sum_{p \leq n, p \text{ prime}} \frac{n}{p} = n \sum_{p \leq n} \frac{1}{p}$$

By Mertens' theorem, $\sum_{p \leq n} \frac{1}{p} \approx \log \log n$, giving $O(n \log \log n)$.

For practical purposes, $\log \log n$ grows extremely slowly-for $n = 10^9$, $\log \log n \approx 3$. So the algorithm is effectively linear.

Space complexity: $O(n)$

We maintain a boolean array of size n.

Sieve of Eratosthenes: Key Result

The Sieve of Eratosthenes finds all primes up to n in $O(n \log \log n)$ time-nearly linear.

Key insight: Rather than testing each number independently, we exploit the multiplicative structure of integers. Marking multiples is far cheaper than testing primality directly.

This is a paradigm shift: instead of asking "is m prime?" for each m, we ask "what numbers does the prime p eliminate?"

Complexity Notation

The complexity $O(n \log \log n)$ is sometimes loosely stated as $O(n \log n)$ in introductory treatments. While technically $\log \log n = o(\log n)$, the difference is small for practical input sizes, so $O(n \log n)$ serves as a conservative upper bound.

For rigorous analysis, the $O(n \log \log n)$ bound is correct and follows from the distribution of prime numbers (specifically, the Prime Number Theorem and Mertens' second theorem).

41 Summary: The Art of Algorithm Design

The four case studies illustrate a common theme: **efficiency comes from exploiting structure**.

Algorithm Design Principles

1. **Identify redundant work:** Are we computing the same thing multiple times? (Fibonacci)
2. **Exploit ordering:** Does sorted data allow faster operations? (Binary search)
3. **Create useful structure:** Can we organise data to make later operations cheaper? (MergeSort)
4. **Use problem-specific properties:** What mathematical structure can we leverage? (Sieve)

The common pattern:

$$\text{Naive approach} \xrightarrow{\text{identify weakness}} \text{Insight about structure} \xrightarrow{\text{exploit it}} \text{Efficient algorithm}$$

Problem	Naive	Improved	Key Insight
Fibonacci	$O(2^n)$	$O(n)$	Cache computed values
Search	$O(n)$	$O(\log n)$	Halve search space each step
Sort	$O(n!)$	$O(n \log n)$	Merge sorted sublists
Primes	$O(n^2)$	$O(n \log \log n)$	Mark multiples, not test divisors

Each improvement represents not just a faster algorithm, but a deeper understanding of the problem's structure.

DS&A Lecture Notes: Wk 7
Optimisation

Chapter Overview

This chapter covers numerical optimisation methods that form the backbone of machine learning algorithms. Understanding these techniques is essential for training models effectively, as most ML problems reduce to finding parameters that minimise a loss function. The concepts here underpin everything from linear regression to deep neural networks.

Central Learning Objective: Constrained Optimisation

$$\theta^* = \arg\min_{\theta \in \Theta} \mathcal{L}(\theta)$$

Where Θ is the *constrained* parameter space.

42 Local vs Global Minima

42.1 Local Minimum

A point is a local minimum if the function value is greater or equal in every **direction** within some neighbourhood.

Local Minimum

$$\exists \delta > 0, \forall \theta \in \Theta \text{ s.t. } |\theta - \theta^*| < \delta, \mathcal{L}(\theta^*) \leq \mathcal{L}(\theta)$$

Breaking down this definition:

- $\exists \delta > 0$: there exists a positive number δ (the neighbourhood radius).
- $\forall \theta \in \Theta$ s.t. $|\theta - \theta^*| < \delta$: for all θ in the set Θ such that the absolute difference between θ and θ^* is less than δ.
- $\mathcal{L}(\theta^*) \leq \mathcal{L}(\theta)$: the value of the function $\mathcal{L}$ evaluated at θ^* is less than or equal to the value of the function $\mathcal{L}$ evaluated at θ.

42.2 Global Minimum

A point is a global minimum if the function value is larger at every other **location** in the entire domain.

Global Minimum

$$\forall \theta \in \Theta, \mathcal{L}(\theta^*) \leq \mathcal{L}(\theta)$$

- For all θ in the set Θ.
- The value of the function $\mathcal{L}$ evaluated at θ^* is less than or equal to the value of the function $\mathcal{L}$ evaluated at θ.

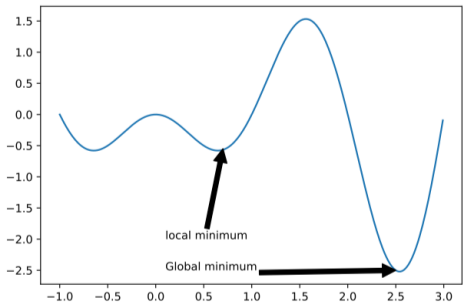

Figure 68: Local versus global minima: a function may have multiple local minima, but only one global minimum (the lowest point overall).

43 Characterising a Local Minimum

43.1 One-Dimensional Case

In one dimension, a local minimum is characterised by two conditions:

1. **First derivative = 0**

$$\frac{d}{d\theta}\mathcal{L}(\theta)\bigg|_{\theta^*} = 0$$

Evaluate the derivative of the loss function at θ^*; at a minimum, the function is neither increasing nor decreasing in that infinitesimal region.

2. **Second derivative ≥ 0 (positive)**

$$\frac{d^2}{d\theta^2}\mathcal{L}(\theta)\bigg|_{\theta^*} \geq 0$$

The second derivative is the "gradient of the gradient" - the rate of change of the rate of change. It should be positive because this indicates that the slope is increasing as we move away from θ^*: we are on an upward movement in any direction. A move in any direction would make the slope become larger (more positive or less negative).

Geometrically, these conditions ensure the function is locally **bowl-shaped** at the minimum.

43.2 High-Dimensional Case

1. **First derivative: Gradient = 0**

Gradient

The **gradient** is the derivative along every parameter dimension. If θ is p-dimensional, then the gradient of $\mathcal{L}$ with respect to θ is a p-dimensional vector:

$$\nabla \mathcal{L}(\theta) = \left(\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \ldots, \frac{\partial \mathcal{L}}{\partial \theta_p} \right)$$

This vector represents the rate of change of $\mathcal{L}$ with respect to each component of θ. The gradient points in the direction of steepest ascent.

At a local minimum, the gradient must be zero:

$$\nabla \mathcal{L}(\theta)\big|_{\theta^*} = \nabla \mathcal{L}(\theta^*) = \mathbf{0}$$

If the gradient were non-zero, we could move a little bit in the negative gradient direction and obtain a smaller value.

2. **Second derivative: Hessian must be "positive-like"**

Hessian Matrix

The **Hessian** is a square matrix of second-order partial derivatives of a scalar-valued function. It describes the local curvature of the function's surface.

For a function $f(\mathbf{x})$ where $\mathbf{x}$ is a vector of variables:

$$H_{ij} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}$$

where H_{ij} represents the second partial derivative of f with respect to x_i and x_j.

The Hessian matrix provides important information about the **local behaviour** of the function:

- **Positive-definite** Hessian $\Rightarrow$ local minimum
- **Negative-definite** Hessian $\Rightarrow$ local maximum
- **Indefinite** Hessian $\Rightarrow$ saddle point

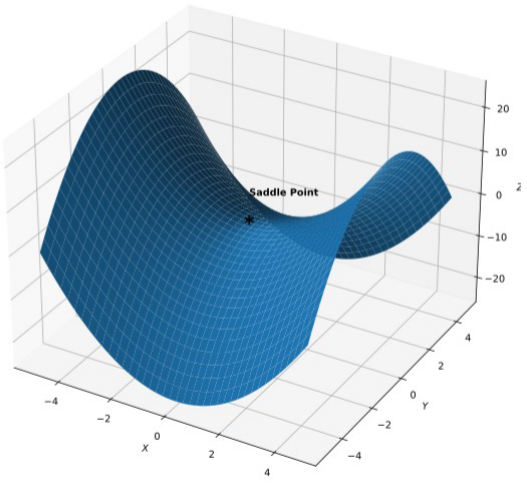

Figure 69: A saddle point is not a local minimum: the Hessian is indefinite. While the function curves upward in the x-direction, it curves downward in the y-direction; moving along y would reduce the loss.

For a local minimum, we require a positive (semi)definite Hessian - meaning a move in any direction should be going uphill:

$$H^* = \nabla^2 \mathcal{L}(\theta)\big|_{\theta^*} = \nabla^2 \mathcal{L}(\theta^*) \succeq 0$$

43.3 Quadratic Forms and Definiteness

To understand what "positive definite" means precisely, we need the concept of quadratic forms.

Quadratic Form

The **quadratic form** $\mathbf{x}^T A\mathbf{x}$ represents a scalar value obtained by multiplying a vector $\mathbf{x}$ by a matrix A and then by $\mathbf{x}^T$ (the transpose). This is called "quadratic" because it involves the square of the variables in $\mathbf{x}$.

- **Vector x:** A column vector of variables.
- **Matrix A:** A symmetric matrix whose coefficients determine the interactions between variables.

The structure $\mathbf{x}^T A\mathbf{x}$ arises from matrix multiplication compatibility: $\mathbf{x}^T$ is a row vector, A is a matrix, and $\mathbf{x}$ is a column vector, yielding a scalar. The result summarises how the variables interact according to the coefficients in A.

Matrix Definiteness

Positive Definite: A symmetric matrix A is positive definite if for any non-zero vector $\mathbf{x}$:

$$\mathbf{x}^T A\mathbf{x} > 0 \quad \text{for all } \mathbf{x} \neq \mathbf{0}$$

Positive Semidefinite: A symmetric matrix A is positive semidefinite if:

$$\mathbf{x}^T A\mathbf{x} \geq 0 \quad \text{for all } \mathbf{x}$$

Negative Definite: A symmetric matrix A is negative definite if:

$$\mathbf{x}^T A\mathbf{x} < 0 \quad \text{for all } \mathbf{x} \neq \mathbf{0}$$

Indefinite: A symmetric matrix A is indefinite if there exist vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ such that:

$$\mathbf{x}_1^T A\mathbf{x}_1 > 0 \quad \text{and} \quad \mathbf{x}_2^T A\mathbf{x}_2 < 0$$

That is, the quadratic form can take both positive and negative values depending on direction.

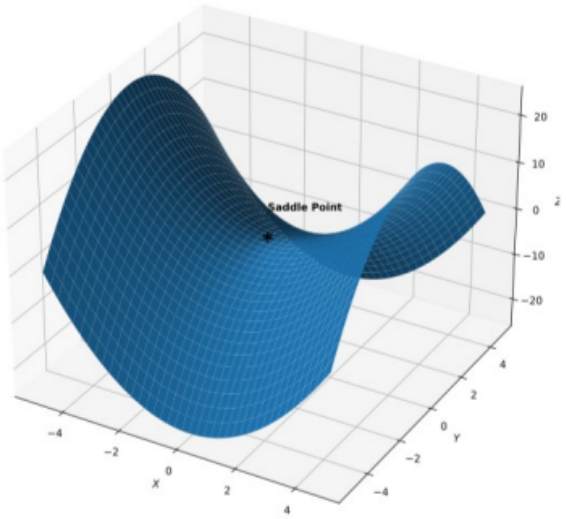

Figure 70: Another view of a saddle point: H is indefinite. While the surface curves upward in one direction, any move along the perpendicular direction would further reduce the loss function.

Local Minimum Conditions

For a point θ^* to be a local minimum:

1. $\nabla \mathcal{L}(\theta^*) = \mathbf{0}$ (gradient is zero)
2. $\nabla^2 \mathcal{L}(\theta^*) \succeq 0$ (Hessian is positive semidefinite)

Geometrically: the function is locally **bowl-shaped** at the minimum.

44 Taylor Expansion and the Logic of Local Minima

Why This Matters

Taylor series provides the formal proof that positive definiteness of the Hessian characterises a minimum:

- Taylor series allows us to locally approximate a function
- Its second-order term is the Hessian
- At a critical point, the first-order term (gradient) is zero
- For sufficiently small perturbations, higher-order terms become negligible

This leaves us with just the second-order/quadratic term. Requiring the function to increase in all directions (≥ 0) then yields the positive definiteness condition.

The intuition: for θ to be a minimum, any small move in any direction must result in a higher loss. We formalise this using Taylor expansion.

44.1 Taylor Expansion of a Function

We take a Taylor expansion around a possible minimum θ.

Taylor Expansion

The Taylor expansion allows us to **locally approximate** a function $f(x+\epsilon)$ around a point x using the function's value and derivatives at x.

The first few terms of the Taylor expansion of the loss function with a small perturbation ϵ in any direction:

$$\mathcal{L}(\theta+\epsilon) \approx \mathcal{L}(\theta) + \nabla\mathcal{L}(\theta)\cdot\epsilon + \frac{1}{2}\epsilon^T\nabla^2\mathcal{L}(\theta)\,\epsilon + \cdots$$

Hessian in Taylor Expansion

Crucially, $\nabla^2\mathcal{L}(\cdot) = H$ (the Hessian is the matrix of second-order partial derivatives).

So we can write:

$$\mathcal{L}(\theta+\epsilon) \approx \mathcal{L}(\theta) + \nabla\mathcal{L}(\theta)\cdot\epsilon + \frac{1}{2}\epsilon^T H\epsilon + \cdots$$

44.2 At a Critical Point

At a critical point (possible minimum or maximum), the gradient is zero:

$$\nabla\mathcal{L}(\theta) = \mathbf{0}$$

This means the first-order term vanishes, leaving:

$$\mathcal{L}(\theta+\epsilon) \approx \mathcal{L}(\theta) + \frac{1}{2}\epsilon^T H\epsilon + \cdots$$

For sufficiently small ϵ, we can neglect higher-order terms, leaving just the quadratic term.

44.3 Condition for a Minimum

For θ to be a minimum, moving a small distance away in any direction (ϵ) should result in a higher loss:

$$\mathcal{L}(\theta + \epsilon) \geq \mathcal{L}(\theta)$$

Substituting our approximation:

$$\frac{1}{2}\epsilon^T H\epsilon \geq 0$$

Since $\frac{1}{2} > 0$, this simplifies to:

$$\epsilon^T H\epsilon \geq 0$$

This condition must hold for all non-zero vectors ϵ, which is precisely the definition of **positive semidefiniteness**.

44.4 Positive Definiteness as Characterisation

The above derivation shows that **positive semidefiniteness of the Hessian at a point θ is a necessary condition for that point to be a local minimum**.

For a *strict* local minimum (where nearby points have strictly higher function values), we require *positive definiteness*: $\epsilon^T H\epsilon > 0$ for all $\epsilon \neq \mathbf{0}$.

45 Convexity

Convexity determines when a **local** minimum is also a **global** minimum.

Convexity and Global Optimality

1. Positive (semi)definite H at a point $\Rightarrow$ local minimum
2. Convex function $\Rightarrow$ any local minimum is also a global minimum

45.1 Formal Definition

Convex Function

A function f is **convex** if for all x, y in its domain and all $\lambda \in [0, 1]$:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

This states that the function value at any convex combination of x and y is less than or equal to the convex combination of the function values.

Geometrically: the line segment connecting any two points on the curve lies above or on the curve. Any two points above the curve can be connected by a straight line that does not cross the curve.

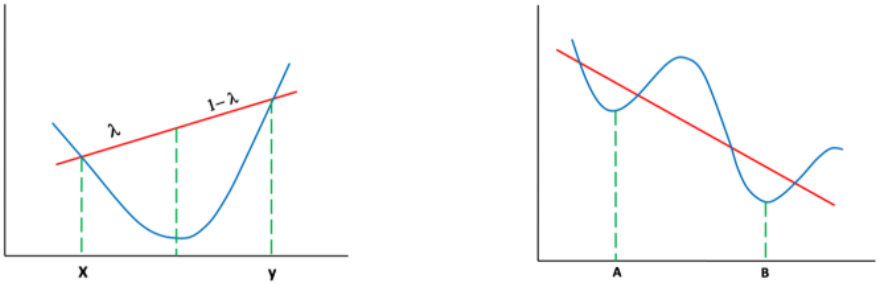

Figure 71: Left: convex function (line segment lies above curve). Right: non-convex function (the function crosses the line segment).

This "bowl" shape ensures that any point within the bowl cannot be higher than points on the line segment connecting any two points on the graph.

45.2 Why Convexity Matters

For loss functions in machine learning, the bowl shape ensures that **if you find a minimum, it is the lowest point in the entire "bowl"** - the global minimum.

Convexity is crucial because:

- It guarantees that a local minimum is also a global minimum
- In high-dimensional optimisation (common in ML), finding the global minimum is computationally challenging
- Convex functions make this feasible: once a local minimum is found, no further search is necessary

This is why convex loss functions are highly desirable for tasks like linear regression, logistic regression, and support vector machines.

45.3 Local vs Global Minima: The Role of PSD Hessians

Characterising Minima via the Hessian

1. **Local Minimum:** Occurs at any point where the Hessian matrix is positive semidefinite (PSD) - the quadratic form $\mathbf{z}^T \nabla^2 f(x) \mathbf{z} \geq 0$ holds for any non-zero vector **z at that point**.
2. **Global Minimum (via Convexity):** For a function to be convex (and thus have any local minimum be global), the Hessian must be PSD **everywhere** in the domain.
3. **Unique Global Minimum:** Requires positive **definiteness** (PD) everywhere, giving strict convexity.

Non-Convex Functions Can Have Global Minima

A function need not be convex to have a global minimum. Consider a surface with two valleys where one is slightly lower than the other. This function has a unique global minimum but is not convex (since the Hessian is not PSD everywhere).

The key insight is that convexity provides a **guarantee**: if you find any local minimum, you know it is global. Without convexity, you might find a local minimum but have no way to verify whether a better solution exists elsewhere.

45.4 Finding Global Minima of Non-Convex Functions

For non-convex functions, several strategies can help find the global minimum:

- It *may* be possible to find the global minimum of a non-convex function, but there are no guarantees.
- Consider a surface with two deep valleys where one is slightly lower. Depending on your starting point, gradient-based methods could converge to either valley, and both would appear to be good solutions.
- **Random restarts**: Run your (local) optimisation algorithm multiple times from different starting points, record the objective value at each minimum, then select the lowest as an estimate of the global minimum.
- Under certain weaker assumptions (e.g., that the objective is smooth), random restarts with sufficiently many trials can provide probabilistic guarantees of approaching the global minimum.

45.5 OLS as a Concrete Example

Let us examine Ordinary Least Squares to see convexity in action.

45.5.1 OLS Loss Function

$$\mathcal{L}(\beta) = \frac{1}{2}(X\beta - y)^T(X\beta - y)$$

Expanding:

$$\mathcal{L}(\beta) = \frac{1}{2}(\beta^T X^T X\beta - \beta^T X^T y - y^T X\beta + y^T y)$$

Since $(X\beta - y)^T = \beta^T X^T - y^T$, the middle terms $\beta^T X^T y$ and $y^T X\beta$ are equal scalars and can be combined.

45.5.2 Gradient of OLS Loss

$$\nabla\mathcal{L}(\beta) = X^T X\beta - X^T y$$

(We drop constant factors that do not affect the location of minima.)

45.5.3 Hessian of OLS Loss

$$\nabla^2 \mathcal{L}(\beta) = X^T X$$

This is the covariance-like matrix of the features.

45.5.4 Convexity of OLS

For a minimum, we need the Hessian to be positive semidefinite. For OLS:

If X has full column rank, then $X^T X$ is positive semidefinite (in fact, positive definite).

Proof sketch: For any vector $\mathbf{z}$, we have $\mathbf{z}^T X^T X \mathbf{z} = \|X\mathbf{z}\|^2 \geq 0$, with equality only when $X\mathbf{z} = \mathbf{0}$. If X has full column rank, then $X\mathbf{z} = \mathbf{0}$ implies $\mathbf{z} = \mathbf{0}$, so $\mathbf{z}^T X^T X \mathbf{z} > 0$ for all $\mathbf{z} \neq \mathbf{0}$.

Therefore, given that X is full rank, **OLS is a convex problem**; any local minimum is the global minimum.

Why We Like OLS

With certain assumptions in place (full column rank of X), OLS provides a convex loss function. Finding the minimum through gradient descent (or analytically) guarantees we have found the unique global optimum.

When X is not of full rank:
If there is **linear dependence** among columns, there may not be a unique global solution. Linear dependence means there exist combinations of columns leading to redundancy, allowing infinitely many solutions that minimise the loss function.
This scenario arises in **overfitting**: despite infinitely many models perfectly interpolating the training data, selecting the most appropriate model requires additional criteria or regularisation to ensure good generalisation.

Summary: Convexity, PSD, and Minima
Positive Semi-Definiteness and Convexity: If a function's Hessian is PSD at every point, the function is convex. The function "curves upward" or is flat in all directions.
Convexity and Minima: Convexity ensures any local minimum is global, eliminating isolated "valleys" common in non-convex functions.
Uniqueness: For strictly convex functions (Hessian PD everywhere), there is exactly one unique global minimum. If the Hessian is merely PSD, the minimum may not be unique, but all minima form a convex set and are global.
Practical Implication: Knowing a problem is convex simplifies optimisation - any minimum found is guaranteed to be global.

46 First-Order Methods

When to Use First-Order Methods

When closed-form solutions are unavailable, we turn to first-order optimisation methods. For instance:

- If X is not of full rank, leading to an underdetermined system
- When the function is complex and difficult to evaluate directly (e.g., neural networks)

These *numerical* methods, which rely primarily on gradient information, offer a way to iteratively search for a solution even without a straightforward *analytical* solution.

First-order methods leverage only the first derivative (the **gradient**) to find minima. The gradient gives the direction of steepest ascent; by moving opposite to it, the algorithm seeks to reduce the function's value.

46.1 General Procedure

1. **Start with an initial guess** (θ_0): random or based on some heuristic.
2. **Update your guess**:
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_t \cdot \mathbf{d}_t$$
where:
   - η_t = **step size / learning rate** (a positive scalar)
   - $\mathbf{d}_t$ = **descent direction** (a vector indicating which way to update)
3. **Repeat until convergence** (many criteria exist: e.g., when the change in loss falls below a threshold, or when gradient magnitude is small).

Note: We have decomposed the update step ϵ from the Taylor expansion into η (magnitude) and $\mathbf{d}$ (direction).

46.2 Descent Direction

46.2.1 Deriving the Descent Direction via Taylor Expansion

Consider the Taylor expansion of $\mathcal{L}(\theta)$ at point θ with a small move $\eta\mathbf{d}$:

$$\begin{aligned}\mathcal{L}(\theta + \eta\mathbf{d}) &\approx \mathcal{L}(\theta) + \nabla\mathcal{L}(\theta) \cdot (\eta\mathbf{d}) + \cdots \\ &\approx \mathcal{L}(\theta) + \eta \cdot \nabla\mathcal{L}(\theta) \cdot \mathbf{d} + \cdots\end{aligned}$$

The change in $\mathcal{L}$ is primarily determined by $\eta \cdot \nabla\mathcal{L}(\theta) \cdot \mathbf{d}$, where:

- η is a scalar controlling step size (we want this large enough for progress)
- $\nabla\mathcal{L}(\theta) \cdot \mathbf{d}$ is a dot product determining direction (we want to minimise this)

46.2.2 Optimal Direction

To minimise the dot product $\nabla\mathcal{L}(\theta) \cdot \mathbf{d}$, we choose $\mathbf{d}$ opposite to $\nabla\mathcal{L}(\theta)$:

Descent Direction

$$\mathbf{d} = -\nabla\mathcal{L}(\theta)$$

The dot product is minimised when the two vectors point in opposite directions (the cosine of the angle is -1). Thus, choosing $\mathbf{d} = -\nabla\mathcal{L}(\theta)$ ensures we move in the direction that decreases $\mathcal{L}$ most steeply.

46.2.3 Gradient Descent Update Rule

Gradient Descent

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \nabla\mathcal{L}(\theta_t)$$

By choosing direction $\mathbf{d} = -\nabla\mathcal{L}(\theta)$, gradient descent leverages local slope information to iteratively move towards the minimum.

46.3 Step Size / Learning Rate

Choosing the learning rate η is more nuanced than choosing the direction.

Too large $\Rightarrow$ failure to converge:

- Updates may overshoot the minimum, potentially causing divergence
- Even without divergence, oscillations around the minimum can prevent settling at the optimal point

Too small $\Rightarrow$ slow convergence / getting stuck:

- Many more iterations required, becoming computationally expensive
- Risk of getting stuck in local minima or plateau regions (especially for non-convex problems)

46.3.1 Example 1a: $f(x) = x^2$

- Convex function, so any local optimum is global
- Minimum at $x = 0$

Running gradient descent from $x = 4$ with learning rate $\eta = 0.2$:

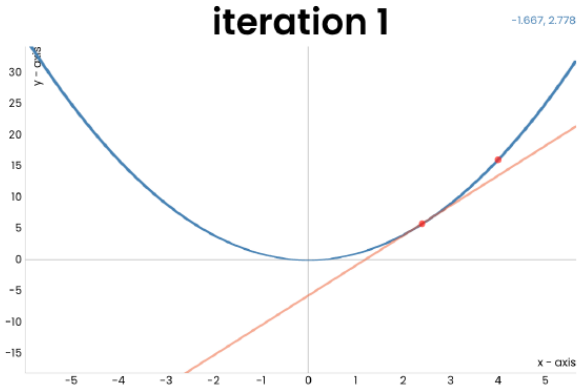

Figure 72: Gradient descent on $f(x) = x^2$, iteration 1: starting at $x = 4$ with $\eta = 0.2$. The algorithm computes the gradient and takes a step towards the minimum.

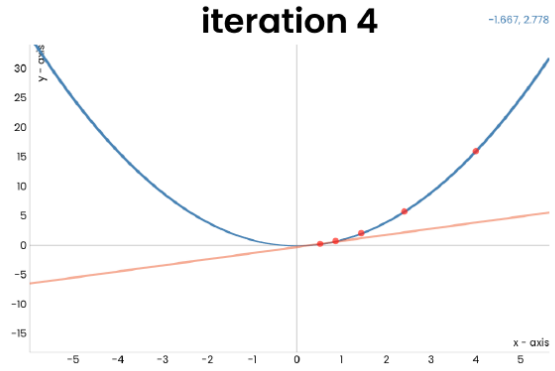

Figure 73: Iteration 4: continued progress towards the minimum at $x = 0$.

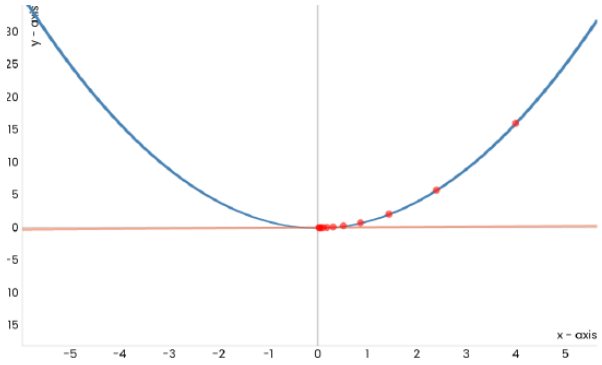

Figure 74: Iteration 10: the algorithm has nearly converged to the minimum. The learning rate is small enough to avoid overshooting.

46.3.2 Example 1b: $f(x) = x^2$ with Large Learning Rate

Same function, but with $\eta = 1.0$:

Figure 75: Large learning rate ($\eta = 1.0$), iteration 0: starting position.

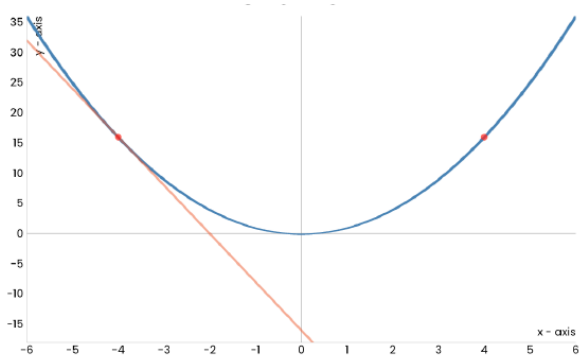

Figure 76: Iteration 1: the algorithm overshoots to the opposite side of the minimum.

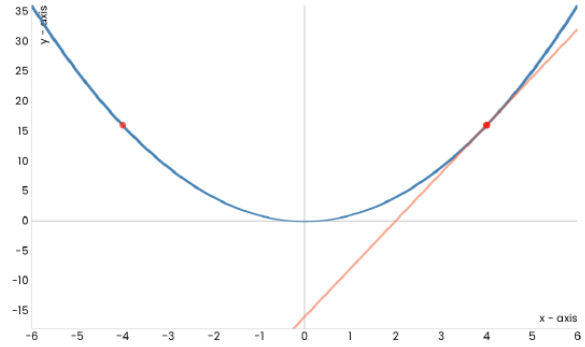

Figure 77: Iteration 2: oscillating back. With $\eta = 1.0$, the algorithm will never converge - it ping-pongs between $x = 4$ and $x = -4$ indefinitely.

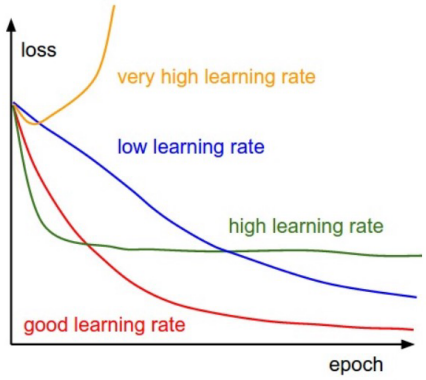

Figure 78: Learning curves showing loss vs iteration. We want **smooth descent**. With a high learning rate (orange), the loss plateaus as the algorithm oscillates without making progress.

46.3.3 Example 2a: Non-Convex Cubic

Consider $f(x) = x + 2x^2 + 0.4x^3$:

- Not convex
- Global minimum: $x = -\infty$ (where $f(-\infty) = -\infty$)
- But there is a local minimum

Running gradient descent from $x = 2$ with $\eta = 0.25$:

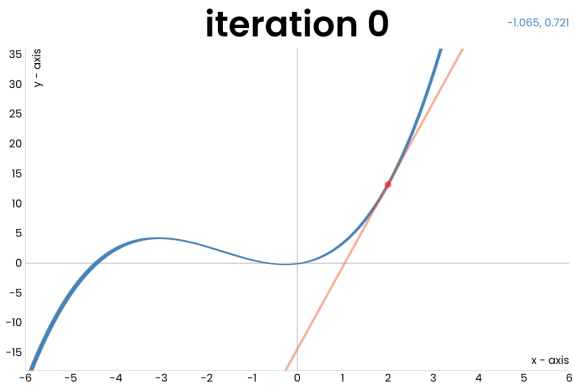

Figure 79: Cubic function, early iterations: starting at $x = 2$.

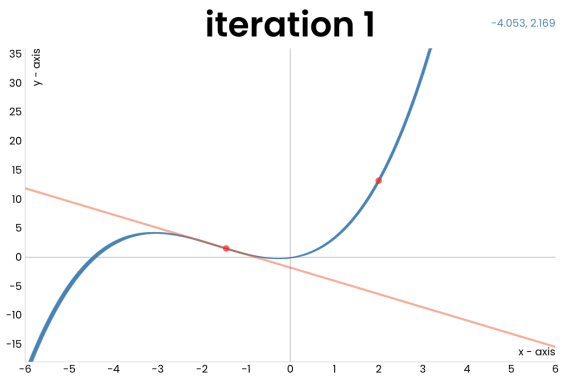

Figure 80: The algorithm descends into the local valley.

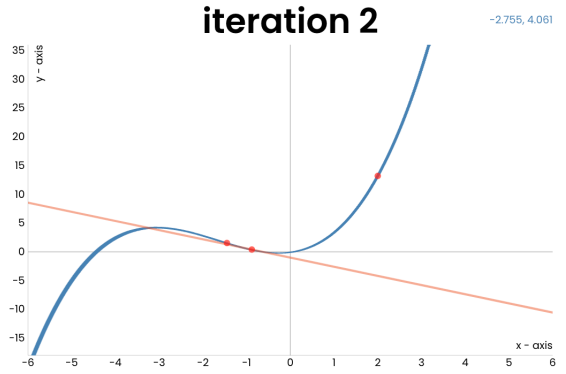

Figure 81: Approaching the local minimum.

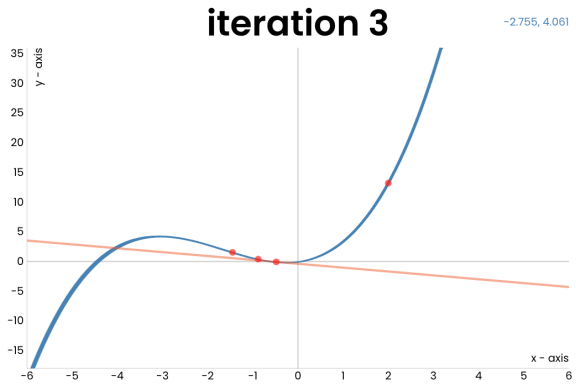

Figure 82: Converged to local minimum. The algorithm does **not** find the global minimum (at $-\infty$) because the learning rate is too small to escape the local valley.

46.3.4 Example 2b: Same Cubic, Different Starting Point

Same function and learning rate, but starting at $x = -4$:

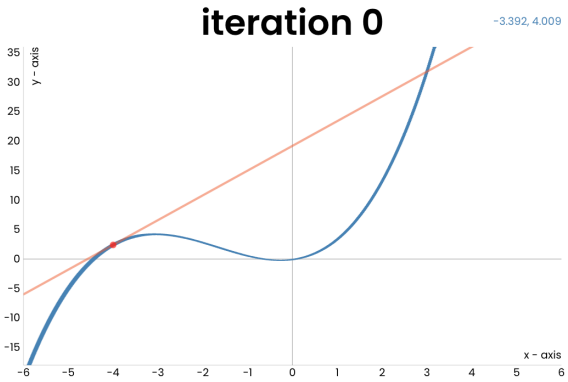

Figure 83: Starting at $x = -4$: on the left side of the local minimum.

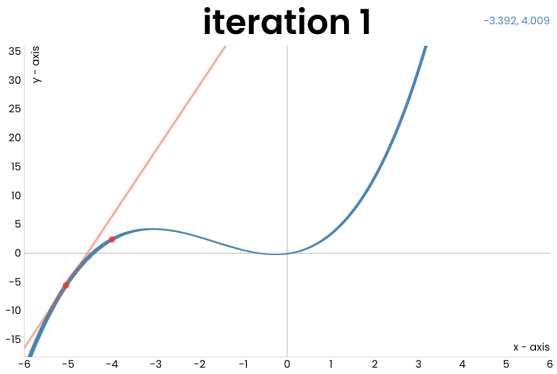

Figure 84: The gradient points left (towards $-\infty$), so the algorithm moves that direction.

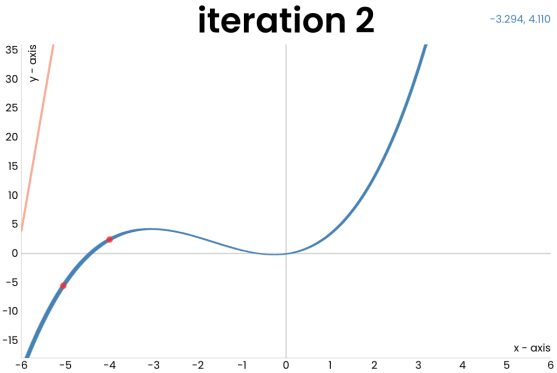

Figure 85: The algorithm continues towards $-\infty$, finding the global minimum. With the same η, the starting point determined which minimum was found.

Non-Convexity and Initialisation

For non-convex functions, the starting point critically determines what the algorithm converges to. This motivates **randomised restarts**: running gradient descent from multiple random starting points to increase the chance of finding the global minimum.

46.4 Choosing a Constant Learning Rate

If the gradient is sufficiently smooth - specifically, "Lipschitz smooth" with constant L:

$$\|\nabla\mathcal{L}(\theta^*) - \nabla\mathcal{L}(\theta)\| \leq L\,\|\theta^* - \theta\|$$

Then if we choose $\eta \leq \frac{2}{L}$, gradient descent is guaranteed to converge.

Intuitively:

- **Smoother function** (smaller L) $\Rightarrow$ larger steps permissible
- **More rapidly varying function** (larger L) $\Rightarrow$ smaller steps required

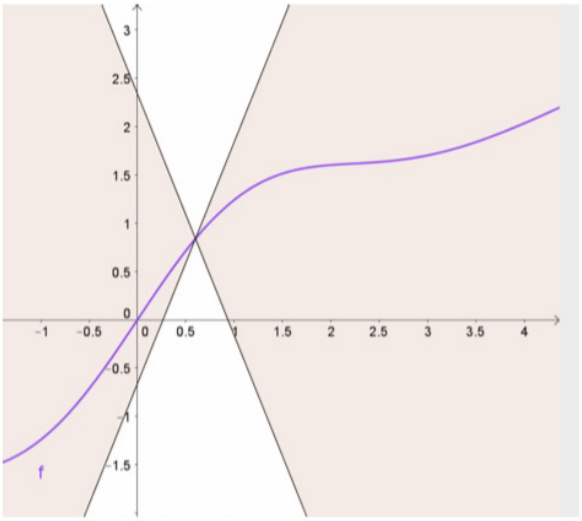

Figure 86: Smooth vs non-smooth functions: smoother functions allow larger step sizes without overshooting. The Lipschitz constant L quantifies this smoothness.

46.5 Choosing a Variable Learning Rate

46.5.1 Exact Line Search (Steepest Descent)

At each iteration, find the step size that minimises the loss along the search direction:

$$\eta^* = \arg\min_{\eta>0} \mathcal{L}(\theta + \eta\mathbf{d})$$

For certain problem classes (e.g., quadratic functions), this has a closed-form solution:

$$\eta^* = -\frac{\mathbf{d}^T\nabla\mathcal{L}(\theta_t)}{\mathbf{d}^T\nabla^2\mathcal{L}(\theta_t)\mathbf{d}}$$

where:

- $\nabla\mathcal{L}(\theta_t)$: Gradient
- $\nabla^2\mathcal{L}(\theta_t)$: Hessian

Optimal Step Size Intuition

The optimal step size is related to the **ratio of the gradient to the Hessian**:

- The **gradient** provides direction and magnitude of steepest ascent
- The **Hessian** captures the curvature of the loss surface

This adapts the learning rate based on local curvature: larger steps in flatter regions, smaller steps in steeper/more curved regions.

For OLS specifically:

$$\eta^* = -\frac{\mathbf{d}^T(X^TX\beta - X^Ty)}{\mathbf{d}^TX^TX\mathbf{d}}$$

46.5.2 Inexact Line Search

Exact line search can be computationally expensive. Inexact methods find a "good enough" step size.

Armijo Backtracking:

- Start with a relatively large step size η
- Iteratively scale back by factor c (where $0 < c < 1$)
- Stop when a sufficient decrease condition is met:

$$\mathcal{L}(\theta + \eta\mathbf{d}) \leq \mathcal{L}(\theta) + c \cdot \eta\nabla\mathcal{L}(\theta)^T\mathbf{d}$$

This balances decrease in function value with computational efficiency.

46.5.3 Convergence Rates and Condition Number

The **condition number** κ measures how the output can change for small input changes:

$$\kappa = \frac{\sigma_{\max}}{\sigma_{\min}}$$

where $\sigma_{\max}$ and $\sigma_{\min}$ are the maximum and minimum singular values of the Hessian. The condition number indicates how "round" the loss surface bowl is:

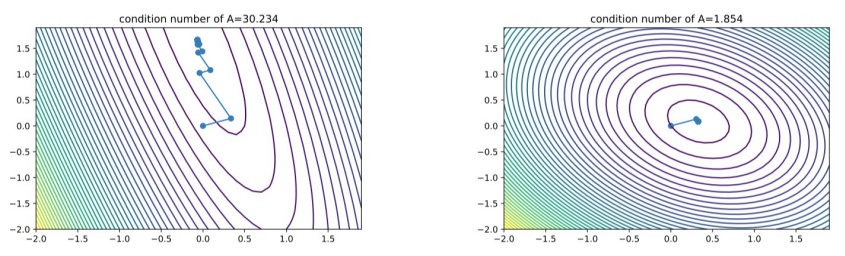

Figure 87: High condition number ($\kappa = 1000$): the bowl is elongated, causing inefficient zigzagging paths. Low condition number ($\kappa \approx 1$): the bowl is circular, allowing direct paths to the minimum.

- **High** κ: ill-conditioned problem; different directions have vastly different curvatures, causing oscillations and slow convergence
- **Low** κ: well-conditioned; curvature is similar in all directions, leading to stable, fast convergence

For steepest descent on OLS, the **convergence rate** is:

$$\left(\frac{\kappa - 1}{\kappa + 1}\right)^2$$

This tells us the error reduction factor per iteration:

- $\kappa \approx 1$: convergence rate ≈ 0 (very fast convergence)
- $\kappa = 100$: convergence rate ≈ 0.96 (slow - only 4% error reduction per iteration)

46.6 Momentum

Momentum incorporates **past gradient information** to inform the current update direction. Inspired by physical momentum: a ball rolling downhill accumulates speed.

Benefits:

1. Helps navigate through **flat regions** of the loss landscape
2. Overcomes **oscillations** in steep valleys
3. Dampens sensitivity to noisy or temporarily misleading gradients

46.6.1 Momentum Update Rules

1. **Momentum accumulation (autoregressive):**

$$m_t = \beta m_{t-1} + \nabla \mathcal{L}(\theta_{t-1})$$

where $\beta \in [0, 1)$ is the momentum coefficient. If $\beta = 0$, this reduces to pure gradient descent.

2. **Parameter update:**

$$\theta_t = \theta_{t-1} - \eta m_t$$

46.6.2 Understanding Momentum Accumulation

Expanding the momentum term recursively:

$$\begin{aligned} m_t &= \beta m_{t-1} + \nabla \mathcal{L}(\theta_{t-1}) \\ &= \beta^2 m_{t-2} + \beta \nabla \mathcal{L}(\theta_{t-2}) + \nabla \mathcal{L}(\theta_{t-1}) \\ &= \sum_{i=0}^{t-1} \beta^i \nabla \mathcal{L}(\theta_{t-i-1}) \end{aligned}$$

The momentum is an **exponentially weighted moving average** of past gradients, with recent gradients weighted more heavily.

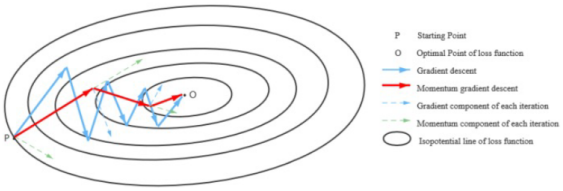

Figure 88: Momentum "smooths out" the optimisation path by averaging over past gradients, reducing oscillations and accelerating convergence in consistent directions.

46.7 Nesterov Accelerated Momentum (NAG)

NAG improves on standard momentum by **anticipating the future position** of parameters before computing the gradient.

With standard momentum, we compute the gradient at the current position then add momentum. This can lead to overshooting. NAG instead:

1. **Look ahead:** $\hat{\theta}_{t+1} = \theta_t + \beta m_t$
2. **Compute gradient at the anticipated position:**

$$m_{t+1} = \beta m_t - \eta_t \nabla \mathcal{L}(\theta_t + \beta m_t)$$

The gradient is computed as if we had already taken the momentum step.

3. **Update parameters:** $\theta_{t+1} = \theta_t + m_{t+1}$

This incorporates curvature information from "ahead," often leading to better convergence.

46.8 Summary: First-Order Methods

First-order methods use gradient information in various ways:

- Gradients at **current** location (basic gradient descent)
  - With **constant** learning rate
  - With **variable** learning rate (exact or inexact line search)
- Gradients at **past** locations (momentum)
- Gradients at **anticipated future** locations (Nesterov momentum)

Exact vs Inexact Line Search
Exact Line Search: Finds the step size that exactly minimises the objective along the search direction. Can be expensive for complex functions.
Inexact Line Search: Finds a step size that sufficiently reduces the objective but does not minimise exactly. Generally less expensive as it only requires meeting certain criteria.

But first-order methods use only gradient information. There is unused second-order (Hessian) structure that critically affects performance (recall condition numbers and optimal learning rates). This motivates second-order methods.

47 Second-Order Methods

47.1 Newton's Method

Newton's method uses the **Hessian** to account for curvature, dramatically speeding up convergence.

Newton's Method

$$\theta_{t+1} = \theta_t - H_t^{-1} \nabla \mathcal{L}(\theta_t)$$

Or with optional line search:

$$\theta_{t+1} = \theta_t - \eta_t (\nabla^2 \mathcal{L}(\theta_t))^{-1} \nabla \mathcal{L}(\theta_t)$$

The Hessian informs both **direction** and **step size**.

Key properties:

- Uses both first and second derivatives
- Faster convergence, especially near the optimum
- Performance contingent on ability to compute and invert the Hessian

47.1.1 Algorithm

1. **Initialise** θ with an initial guess
2. **Compute gradient** $\nabla \mathcal{L}(\theta_t)$
3. **Compute Hessian** $\nabla^2 \mathcal{L}(\theta_t)$
4. **Solve for Newton direction** $\mathbf{d}_t$:
$$\mathbf{d}_t = -(\nabla^2 \mathcal{L}(\theta_t))^{-1} \nabla \mathcal{L}(\theta_t)$$
5. **Optional line search** for η
6. **Update:** $\theta_{t+1} = \theta_t + \eta \mathbf{d}_t$
7. **Repeat** until convergence

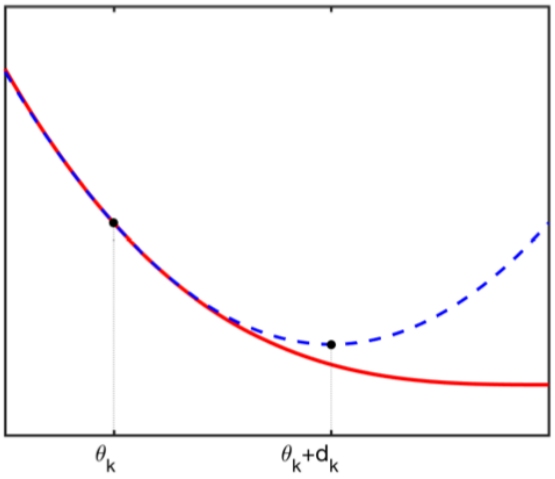

Figure 89: Newton's method iteration 1: the quadratic approximation (dashed) guides the step towards the minimum more directly than gradient descent.

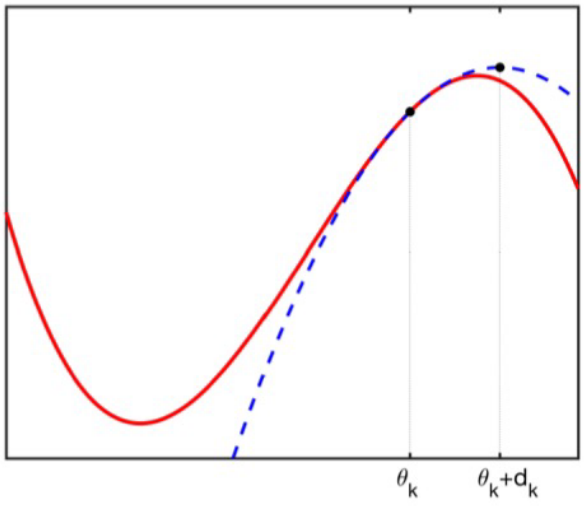

Figure 90: Newton's method iteration 2: for quadratic functions, Newton's method finds the exact minimum in one step.

47.1.2 Derivation via Taylor Series

Step 1: Taylor expansion including Hessian

$$\mathcal{L}(\theta_t + \epsilon) \approx \mathcal{L}(\theta_t) + \nabla\mathcal{L}(\theta_t) \cdot \epsilon + \frac{1}{2}\epsilon^T \nabla^2\mathcal{L}(\theta_t)\epsilon$$

Here ϵ is the step vector from θ_t towards the minimum (not an error term).

Step 2: Minimise the quadratic approximation

Take the gradient of the approximation with respect to ϵ:

$$\nabla_\epsilon \mathcal{L}_{\text{approx}} = \nabla\mathcal{L}(\theta_t) + \nabla^2\mathcal{L}(\theta_t)\epsilon$$

Set to zero:

$$\mathbf{0} = \nabla\mathcal{L}(\theta_t) + \nabla^2\mathcal{L}(\theta_t)\epsilon$$

Step 3: Solve for ϵ

$$\epsilon = -(\nabla^2\mathcal{L}(\theta_t))^{-1}\nabla\mathcal{L}(\theta_t)$$

This gives the Newton update rule:

Newton Update

$$\theta_{t+1} = \theta_t - (\nabla^2\mathcal{L}(\theta_t))^{-1}\nabla\mathcal{L}(\theta_t)$$

The Hessian informs both direction and step size by approximating how the gradient changes as parameters adjust.

47.1.3 Newton's Method Applied to OLS

For OLS with loss $\mathcal{L}(\beta) = \|X\beta - y\|^2$:

- Gradient: $\nabla\mathcal{L}(\beta) = 2X^T(X\beta - y)$
- Hessian: $\nabla^2\mathcal{L}(\beta) = 2X^TX$ (constant - no β dependence)

Since the Hessian is constant, Newton's method converges in **one step**:

$$\begin{aligned}
\beta_1 &= \beta_0 - (X^TX)^{-1}(X^TX\beta_0 - X^Ty) \\
&= \beta_0 - (X^TX)^{-1}X^TX\beta_0 + (X^TX)^{-1}X^Ty \\
&= \beta_0 - \beta_0 + (X^TX)^{-1}X^Ty \\
&= (X^TX)^{-1}X^Ty
\end{aligned}$$

Newton's Method and OLS

$$\beta_1 = (X^TX)^{-1}X^Ty$$

This is the familiar OLS closed-form solution! For OLS, Newton's method finds the exact solution in one iteration, regardless of starting point.

47.1.4 Difficulties with Newton's Method

While efficient for simple problems like OLS, Newton's method faces challenges:

- **Computational cost:** Computing and inverting the Hessian is $O(p^3)$ for p parameters
- **Memory requirements:** Storing the $p \times p$ Hessian may be impractical for large models
- **Numerical stability:** Inversion is unstable if the Hessian is nearly singular (multicollinearity, more features than observations)

These difficulties motivate quasi-Newton methods.

47.2 BFGS (A Quasi-Newton Method)

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm approximates the Hessian iteratively rather than computing it directly.

BFGS Key Idea

Rather than compute H exactly, build up approximations as optimisation proceeds. We know from gradient descent that the Hessian is not strictly necessary for convergence - it just improves efficiency. A rough approximation is often good enough.

47.2.1 Algorithm

Initialisation: $B_0 = I_p$ (identity matrix - assumes the surface is initially "round")

BFGS Update:

$$B_{t+1} = B_t + \frac{y_t y_t^T}{y_t^T s_t} - \frac{(B_t s_t)(B_t s_t)^T}{s_t^T B_t s_t}$$

where:

- $y_t = \nabla \mathcal{L}(\theta_t) - \nabla \mathcal{L}(\theta_{t-1})$: gradient difference
- $s_t = \theta_t - \theta_{t-1}$: parameter difference

This update ensures $B_{t+1} s_t = y_t$ (the secant equation), meaning the approximate Hessian correctly predicts the observed gradient change.

47.2.2 Technical Conditions

For BFGS to work well:

1. Line search should satisfy Wolfe conditions (ensuring sufficient decrease and positive definiteness of the approximation)
2. The function should be twice continuously differentiable with positive definite Hessian near the optimum

Under these conditions, BFGS approximates $\nabla^2 \mathcal{L}(\theta)$ well enough for efficient convergence.

47.2.3 Efficiency Improvements

- **Limited-memory BFGS (L-BFGS):** Stores only a limited number of recent updates, reducing memory from $O(p^2)$ to $O(mp)$ where m is small (typically 5–20). Suitable for high-dimensional problems.
- **Scaling:** Adjusting the initial approximation B_0 based on early iterations can improve convergence.

For problems small enough to hold in memory, BFGS should be the default optimisation algorithm. It provides near-Newton efficiency without requiring explicit Hessian computation.

47.3 Summary: Second-Order Methods

Second-order methods exploit curvature information:

- **Curvature information** allows intelligent step size and direction adjustments
- **Adaptive steps:** larger steps in flat regions, smaller steps in curved regions - more direct paths to the minimum

Why Line Search is Still Sometimes Used

Even though second-order methods inherently inform step size via curvature, line search can still be beneficial:

- The quadratic approximation may be inaccurate far from the optimum
- Line search provides a safety mechanism against poor steps
- For non-convex functions, the Hessian may not be positive definite everywhere

Better approximation to the loss function:

- The Taylor series to second order gives a **quadratic approximation**
- This is usually closer to the true loss than a linear (first-order) approximation
- Enables more accurate prediction of how a step will affect the loss

Faster convergence:

- Fewer iterations due to better step selection
- Better at navigating plateaus and saddle points

Trade-offs:

- Computational cost of Hessian computation/inversion
- Memory requirements for storing the Hessian

- Quasi-Newton methods (BFGS) provide a practical middle ground

When to Use What
Pure Newton: Excellent when Hessian is cheap to compute and invert (small problems, constant Hessian like OLS)
BFGS: General-purpose for medium-sized problems; benefits of second-order without explicit Hessian
First-order (gradient descent): When problems are too large for second-order methods, or when only gradients are available (e.g., neural networks trained with backpropagation)

48 Stochastic Gradient Descent (SGD)

Why Return to First-Order?

After discussing the advantages of second-order methods, we return to first-order methods. Why? Because with small data batches, we cannot estimate H well enough. The noise in stochastic gradient estimates makes second-order information unreliable.

Motivation for SGD
SGD enables rapid updates to model parameters, facilitating faster convergence than methods requiring the entire dataset for each update. It is the foundation of neural network training.
By updating parameters based on gradients from data subsets, SGD handles large-scale datasets effectively and provides a practical approximation to minimising the true (population) risk.

48.1 The Structure in Our Loss Functions

SGD exploits the fact that empirical risk is a **sum over individual losses**:

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(x_i;\theta))$$

Therefore, the gradient is also a sum:

$$\nabla\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \ell(y_i, f(x_i;\theta))$$

48.2 Relationship to Population Risk

Empirical risk approximates population risk:

$$\mathbb{E}[\mathcal{L}(\theta)] = \int \ell(y, f(x;\theta))p(x,y)\,dx\,dy$$

Key insight: if (x_i, y_i) are i.i.d. samples from $p(x, y)$, then:

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\nabla_\theta \ell(y_i, f(x_i;\theta))\right] = \mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B}\nabla_\theta \ell(y_b, f(x_b;\theta))\right]$$

for any batch size $B \leq n$. The expected value of a mini-batch gradient equals the true gradient!

SGD Intuition

Stochastic gradient descent uses a subset of data to estimate the gradient. On average, this estimate equals the population gradient - so on average, SGD moves in the right direction.

48.3 SGD Algorithm

1. Take a small sub-sample (mini-batch) of size B from the data
2. **Estimate gradient on that sub-sample:**

$$\nabla \mathcal{L}_B(\theta) = \frac{1}{B}\sum_{b=1}^{B}\nabla_\theta \ell(y_b, f(x_b;\theta))$$

3. **Update model parameters:**

$$\theta_{t+1} = \theta_t - \eta_t \nabla \mathcal{L}_B(\theta_t)$$

4. **Repeat until convergence**

48.4 Practical Considerations

Choice of batch size B:

- Small batches: more updates per epoch, higher variance in gradient estimates
- Large batches: fewer updates, lower variance, but higher memory usage
- Optimal choice is empirical and problem-dependent

Sampling: Typically sample without replacement within an epoch. After all samples are used, shuffle and repeat.

Epochs: One epoch = one pass over the entire dataset. Training usually involves multiple epochs.

SGD Hyperparameters

1. Number of epochs
2. Mini-batch size B
3. Learning rate η (and schedule)

These require tuning for each problem.

48.5 SGD Applied to OLS

For OLS, the gradient for a single observation (x, y) is:

$$\nabla \ell(y, x\beta) = x^T(x\beta - y)$$

With batch size $B = 1$ (online/incremental learning):

$$\beta_{t+1} = \beta_t - \eta_t \cdot x_t^T(x_t\beta_t - y_t)$$

This is the **Least Mean Squares (LMS)** algorithm, a classic adaptive filter.

Why "Least Mean Squares"?

It is called Least *Mean* Squares because although individual steps may not move in the optimal direction (due to noise from using single samples), *on average* the algorithm moves towards the minimum.

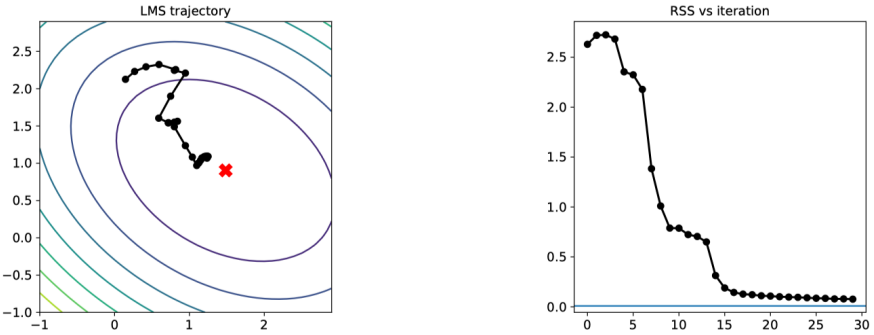

Figure 91: LMS/SGD path: unlike gradient descent, individual steps may not always decrease the loss, but on average the algorithm progresses towards the minimum.

48.6 Choosing a Learning Rate for SGD

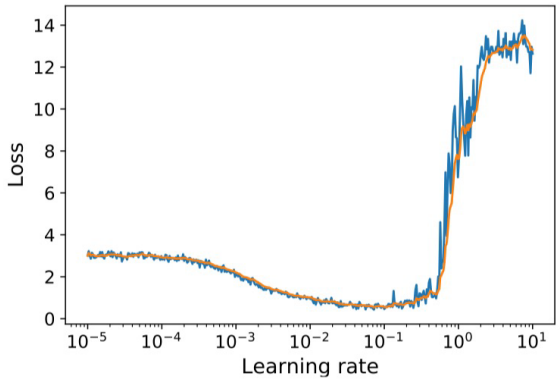

Figure 92: Learning rate effects: too large causes oscillation/divergence; too small causes slow convergence or getting stuck.

48.6.1 Learning Rate Schedules

Adjust η over time: start high for rapid progress, decrease for fine-tuning near convergence.

Common schedules:

- **Time-based decay:** $\eta_t = \eta_0/(1 + kt)$ or $\eta_t = \eta_0 e^{-kt}$
- **Step decay:** Decrease by a factor at specified intervals
- **Performance-based:** Decrease when improvement stalls

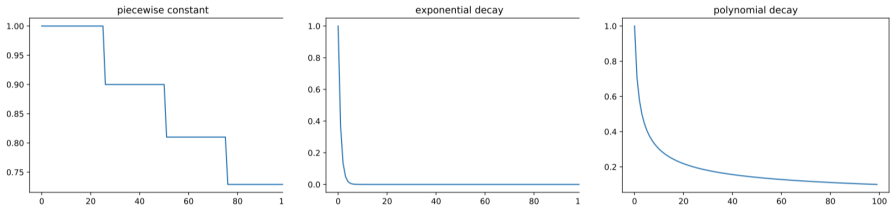

Figure 93: Learning rate schedules: various strategies for decreasing the learning rate over training iterations.

48.6.2 Robbins–Monro Conditions

For guaranteed convergence of stochastic approximation:

Robbins–Monro Conditions

1. Learning rate approaches zero:
$$\lim_{t\to\infty} \eta_t = 0$$
2. Learning rates are square-summable but not summable:
$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

Equivalently: $\frac{\sum_t \eta_t^2}{\sum_t \eta_t} \to 0$

Interpretation:

- Learning rate must decrease to zero for convergence
- But not too quickly - must explore the parameter space sufficiently ($\sum \eta_t = \infty$)
- Sum of squares converging controls variance and ensures stability

A common choice satisfying these conditions: $\eta_t = c/t$ for some constant c.

48.7 Advanced SGD Techniques

Iterate Averaging / Stochastic Weight Averaging:

- Average parameters from multiple iterations to reduce variance
- Stochastic Weight Averaging (SWA) averages over the later training phase, often finding wider optima with better generalisation

Incorporating Second-Order Information:

- Estimate the Hessian on larger batches, then use it
- In practice, noise in stochastic gradients makes this less effective
- Computing/inverting Hessians remains expensive

Preconditioning: Modify gradients to account for ill-conditioning and anisotropic loss surfaces.

Adaptive Learning Rates:

- **AdaGrad:** Per-parameter learning rates based on historical gradient magnitudes; good for sparse data
- **RMSProp:** Uses exponential moving average of squared gradients; addresses AdaGrad's aggressive decay
- **Adam:** Combines momentum (first moment) with adaptive rates (second moment); widely used default

49 Summary

Optimisation Methods Summary

First-Order Methods (Gradient Descent, SGD):

- Simple and easy to implement
- May be slower due to using only local gradient information
- Find solutions to OLS but require many iterations
- SGD enables training on large datasets

Second-Order Methods (Newton, BFGS):

- Account for curvature: faster convergence
- Newton's method solves OLS in one iteration
- BFGS provides second-order benefits without explicit Hessian
- Computationally expensive for large problems

Further Resources
For visualisations of optimisation algorithms in action, see: `https://github.com/jiupinjia/Visualize-Optimization-Algorithms`

50 Divide and Conquer

51 Overview

Divide and Conquer is one of the most powerful algorithmic paradigms in computer science. The strategy decomposes complex problems into simpler subproblems, solves each recursively, and combines the results. This approach underlies many of the most efficient algorithms we use daily.

The paradigm consists of three steps:

1. **Divide** - Break the problem into smaller subproblems of the same type. Typically, we divide a problem of size n into subproblems of size $n/2$, though other divisions are possible.
2. **Conquer** - Solve each subproblem recursively. When subproblems become sufficiently small (the base case), solve them directly.
3. **Combine** - Merge the solutions of subproblems into a solution for the original problem.

The Divide and Conquer Pattern

For most divide and conquer algorithms:

- **Division**: Split problem of size n into two subproblems of size $n/2$
- **Division cost**: $O(1)$ or $O(n)$ depending on the algorithm
- **Combination cost**: $O(n)$ - this is typically where the real work happens
- **Resulting complexity**: Often $O(n \log n)$

51.1 Tools for Analysing Divide and Conquer Algorithms

To effectively understand and solve divide and conquer problems:

1. **Identify the Base Case**: Determine the simplest case that can be solved directly without further division.
2. **Determine the Division Strategy**: Understand how to partition the problem into subproblems.
3. **Design the Combination Step**: Create an efficient method for integrating solutions of subproblems.
4. **Analyse Complexity**: Use recurrence relations and the Master Theorem to determine time and space complexity.

51.2 Recurrence Relations

Recurrence Relation

A **recurrence relation** is an equation that expresses the nth term of a sequence as a function of preceding terms. For divide and conquer algorithms, recurrences capture the recursive structure of the algorithm's running time.

The general form for divide and conquer recurrences is:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- a = number of subproblems created at each recursive call
- b = factor by which the problem size is reduced
- $f(n)$ = cost of the work done outside the recursive calls (dividing and combining)

For the common case where we divide into two equal halves and the combination step is linear:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

This recurrence arises in many fundamental algorithms including MergeSort, and its solution is $T(n) = O(n \log n)$.

52 MergeSort

MergeSort is the canonical example of divide and conquer. It achieves optimal $O(n \log n)$ comparison-based sorting by recursively dividing an array, sorting each half, and merging the sorted halves.

MergeSort Summary

Strategy:

- **Divide**: Split the array into two halves
- **Conquer**: Recursively sort each half
- **Combine**: Merge the two sorted halves into a single sorted array

Complexity: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Space: $O(n)$ auxiliary space for the merge step

52.1 Motivation: Why Do We Sort?

Sorting is fundamental to computer science and appears ubiquitously:

- **Organising data** - Filesystems sort files by name, size, type, or modification date. Libraries organise books by call number. This organisation enables efficient retrieval and improves user experience.
- **Ranking and display** - Search engines (Google), social media feeds (Facebook, Instagram), and recommendation systems all rely on sorting items by relevance scores to display ranked results.
- **Enabling other algorithms** - Sorting is a preprocessing step for many algorithms: binary search requires sorted data, closest pair algorithms benefit from sorted coordinates, and database query optimisation relies heavily on sorted indices.
- **Data analysis** - Finding medians, percentiles, detecting duplicates, and computing order statistics all become tractable once data is sorted.

52.2 The MergeSort Process

```
MERGE-SORT(L)

IF (list L has one element)
    RETURN L.

Divide the list into two halves A and B.
A ← MERGE-SORT(A).  ⟵ T(n / 2)
B ← MERGE-SORT(B).  ⟵ T(n / 2)
L ← MERGE(A, B).    ⟵ Θ(n)
RETURN L.
```

Figure 94: MergeSort recursively divides the array until reaching single elements, then merges sorted subarrays back together.

MergeSort works by recursively sorting the left and right halves of an array and then merging these sorted halves to create a sorted whole.

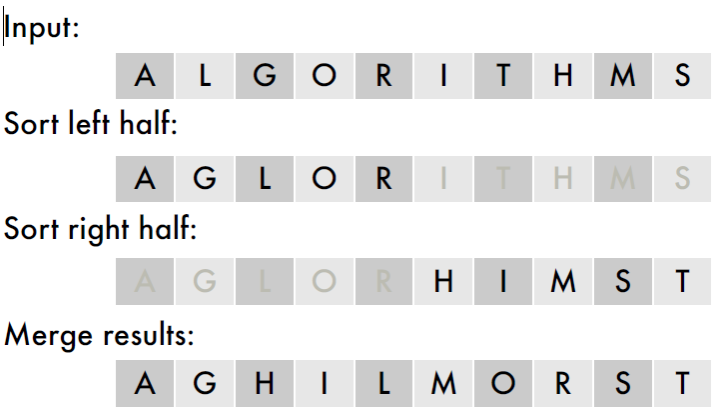

Figure 95: Merging two sorted halves: compare leading elements, take the smaller, and advance that pointer. The merge produces a sorted combined array.

Consider sorting the string "ALGORITHMS":

1. **Divide** into two halves: "ALGOR" and "ITHMS"
2. **Recursively sort** each half, yielding: "AGLOR" and "HIMST"
3. **Merge** these sorted halves:
   - Compare "A" and "H" → take "A"
   - Compare "G" and "H" → take "G"
   - Compare "L" and "H" → take "H"
   - Continue until all elements are merged
4. **Result**: "AGHILMORST"

52.3 Implementation

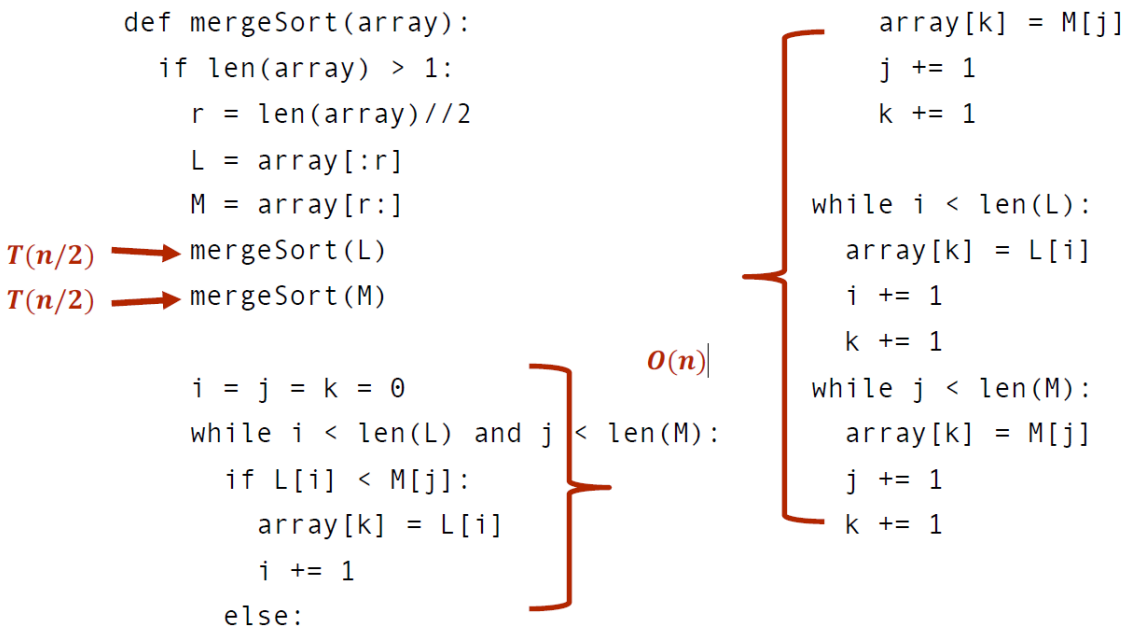

Figure 96: MergeSort pseudocode showing the recursive structure: divide at the midpoint, recursively sort each half, then merge.

```
def mergeSort(array):
    if len(array) > 1:
        mid = len(array) // 2
        L = array[:mid]       # Left half
        R = array[mid:]       # Right half

        mergeSort(L)          # T(n/2)
        mergeSort(R)          # T(n/2)

        # Merge step: O(n)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                array[k] = L[i]
                i += 1
            else:
                array[k] = R[j]
                j += 1
            k += 1

        # Copy remaining elements from L
        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

        # Copy remaining elements from R
```

```
        while j < len(R):
            array[k] = R[j]
            j += 1
            k += 1
```

52.4 Time Complexity Analysis

The recurrence relation for MergeSort captures its recursive structure:

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + O(n) & \text{if } n > 1 \end{cases}$$

The terms represent:

- $T\left(\lfloor \frac{n}{2} \rfloor\right)$ - time to sort the left half
- $T\left(\lceil \frac{n}{2} \rceil\right)$ - time to sort the right half
- $O(n)$ - time to merge the two sorted halves (at most $n - 1$ comparisons)

52.4.1 Recursion Tree Intuition

Before applying the Master Theorem, we can build intuition from the recursion tree:

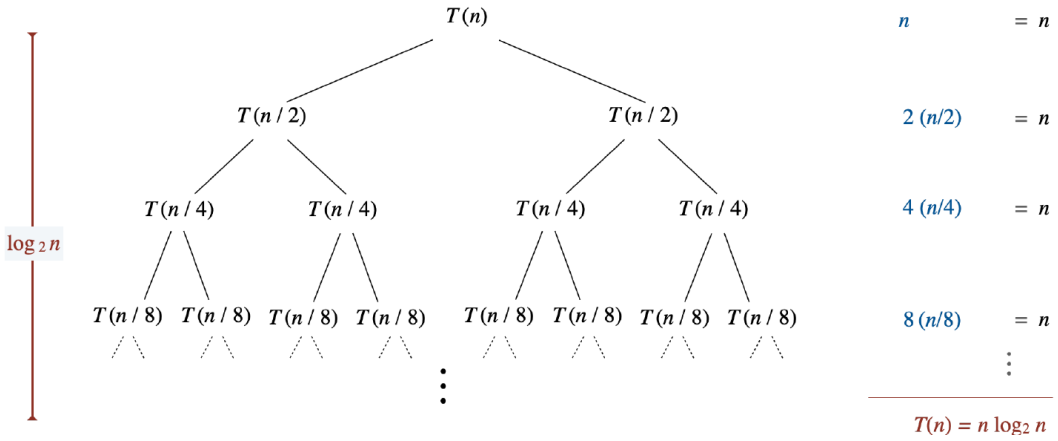

Figure 97: MergeSort recursion tree. Each level performs $O(n)$ total work (across all nodes at that level). With $\log n$ levels, the total work is $O(n \log n)$.

At each level of the recursion tree:

- Level 0: 1 problem of size $n \to O(n)$ merge work
- Level 1: 2 problems of size $n/2 \to O(n)$ total merge work
- Level 2: 4 problems of size $n/4 \to O(n)$ total merge work
- $\vdots$
- Level $\log n$: n problems of size $1 \to$ base case

Since there are $\log n$ levels, each contributing $O(n)$ work, the total is $O(n \log n)$.

52.4.2 Master Theorem Application

Master Theorem for MergeSort

For MergeSort, the recurrence is $T(n) = 2T(n/2) + O(n)$, giving us:

- $a = 2$ (two subproblems)
- $b = 2$ (each subproblem is half the size)
- $f(n) = O(n)$ (linear merge cost)

Computing $\log_b a = \log_2 2 = 1$, we have $f(n) = \Theta(n^{\log_b a}) = \Theta(n^1)$.

By Case 2 of the Master Theorem (when $f(n) = \Theta(n^{\log_b a})$):

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

52.4.3 Formal Inductive Proof

Proof that MergeSort is $O(n \log n)$

Claim: $T(n) \leq cn \log n$ for some constant $c > 0$ and all $n \geq 2$.

Base Case ($n = 1$): A single-element array is trivially sorted with $T(1) = 0$ comparisons.

Inductive Hypothesis: Assume $T(k) \leq ck \log k$ for all $k < n$.

Inductive Step: For an array of size n, let $n' = \lfloor n/2 \rfloor$ and $n'' = \lceil n/2 \rceil$, so $n = n' + n''$.

The recurrence gives us:

$$T(n) \leq T(n') + T(n'') + n$$

By the inductive hypothesis:

$$T(n) \leq cn' \log n' + cn'' \log n'' + n$$

Since $n' \leq n'' \leq n/2 + 1$, we have $\log n' \leq \log n''$, so:

$$T(n) \leq cn' \log n'' + cn'' \log n'' + n = c(n' + n'') \log n'' + n = cn \log n'' + n$$

Since $n'' \leq \lceil n/2 \rceil < n$ for $n > 2$, we have $\log n'' < \log n$, and for sufficiently large c:

$$T(n) \leq cn \log n$$

This completes the induction, proving $T(n) = O(n \log n)$.

53 The Sorting Lower Bound

Having established that MergeSort achieves $O(n \log n)$ complexity, a natural question arises: can we do better? The answer, for comparison-based sorting, is no.

Sorting Lower Bound Theorem

Any deterministic, comparison-based sorting algorithm must make $\Omega(n \log n)$ comparisons in the worst case.

This means $O(n \log n)$ is **optimal** for comparison-based sorting - MergeSort achieves this bound.

Understanding the Lower Bound

This theorem establishes a **lower bound** on the **worst case**:

- **Lower bound**: No algorithm can do better than $\Omega(n \log n)$
- **Worst case**: This bound applies to the hardest inputs

Note: Some algorithms (like Bubble Sort) achieve $O(n)$ on already-sorted inputs, but this does not violate the theorem since it concerns worst-case behaviour. The theorem says: for any comparison-based algorithm, some input will require $\Omega(n \log n)$ comparisons.

53.1 The Decision Tree Model

The proof uses the **decision tree model**, where we assume:

- Elements can only be accessed through pairwise comparisons
- Comparisons are the only operations that "cost" time
- The algorithm's behaviour can be represented as a binary tree of comparison decisions

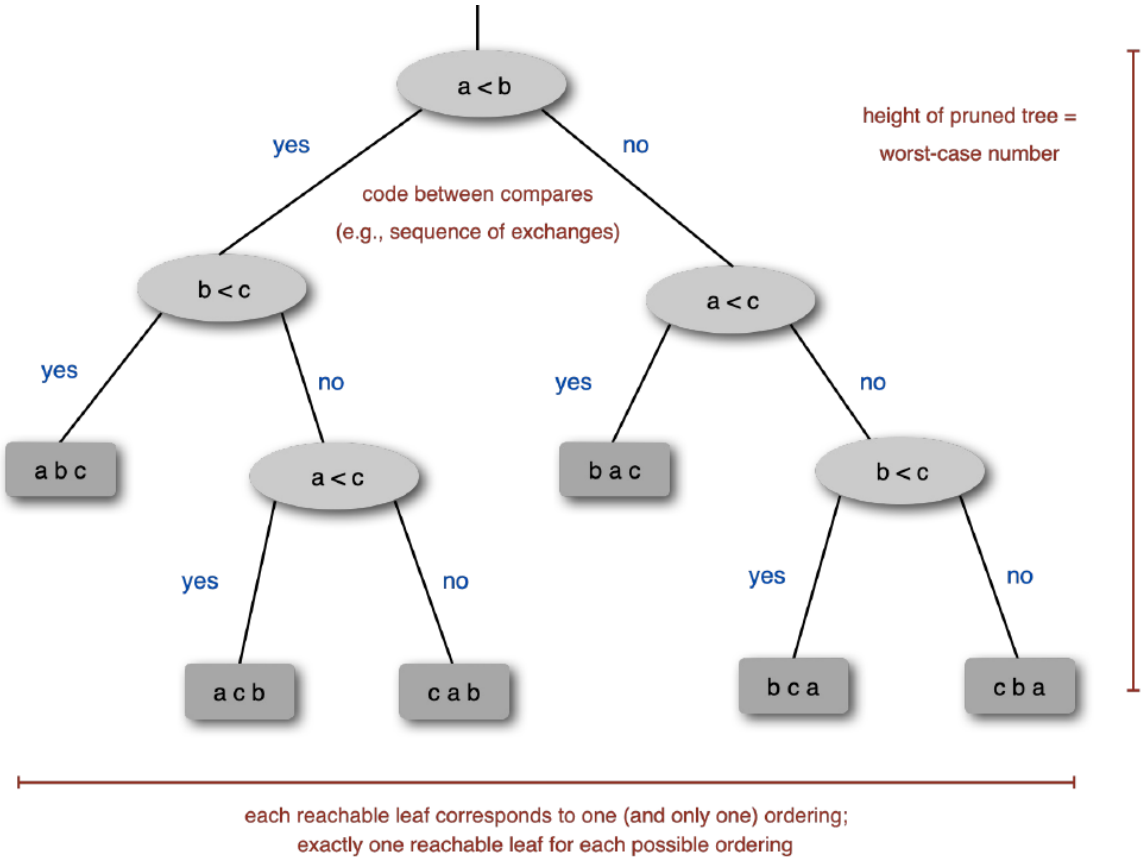

Figure 98: Decision tree for sorting three elements $\{a, b, c\}$. Each internal node represents a comparison; each leaf represents a permutation (sorted order). The tree must have at least $n! = 6$ leaves for $n = 3$ elements.

Key observations about the decision tree:

- Each internal node represents a comparison (e.g., "is $a < b$?")
- Each leaf represents a unique permutation of the input
- The path from root to leaf traces the comparisons made for that input
- The **height** of the tree equals the worst-case number of comparisons

53.2 The Lower Bound Proof

Proof of the $\Omega(n \log n)$ Lower Bound

Setup: Consider sorting n distinct elements. There are $n!$ possible permutations (orderings) of the input.

Key insight: A correct sorting algorithm must distinguish between all $n!$ permutations, since each requires a different sequence of swaps to sort. Therefore, the decision tree must have at least $n!$ leaves.

Binary tree height: A binary tree with L leaves has height at least $\lceil \log_2 L \rceil$. Since our tree needs $L \geq n!$ leaves:

$$\text{Height} \geq \log_2(n!)$$

Stirling's approximation: Using $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, we get:

$$\log_2(n!) = \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = n\log_2 n - n\log_2 e + O(\log n)$$

Therefore:

$$\log_2(n!) = \Theta(n \log n)$$

Conclusion: Any comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparisons in the worst case.

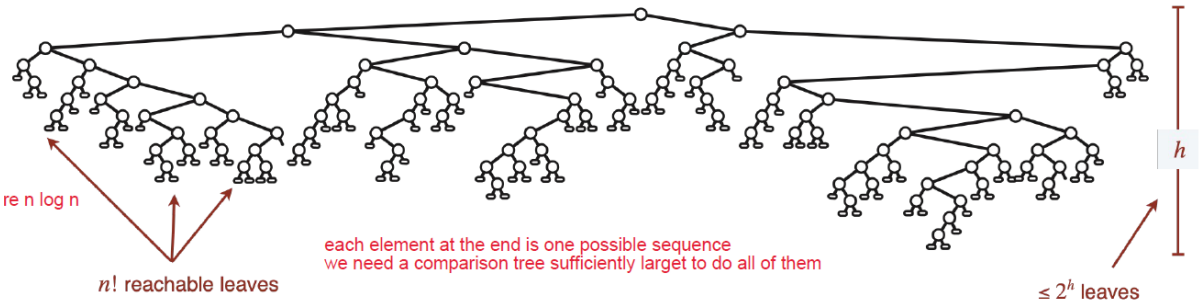

Figure 99: The relationship between tree height, number of leaves, and the sorting lower bound. A binary tree of height h has at most 2^h leaves, so $2^h \geq n!$ implies $h \geq \log_2(n!) = \Omega(n \log n)$.

Implications of the Lower Bound

- MergeSort, HeapSort, and (average-case) QuickSort are **asymptotically optimal** among comparison-based sorts
- To beat $O(n \log n)$, we must use non-comparison-based methods (Counting Sort, Radix Sort, Bucket Sort) which exploit additional structure in the data
- The bound assumes only comparisons provide information about element ordering

54 Counting Inversions

Counting Inversions

An **inversion** is a pair of elements that are "out of order" relative to the sorted sequence. Counting inversions measures how far a list is from being sorted.

Formal definition: Given a sequence $a_1, a_2, \ldots, a_n$, an inversion is a pair (i, j) where $i < j$ but $a_i > a_j$.

Key insight: Inversions count the number of swaps needed to sort (not just elements out of place).

54.1 Motivation and Applications

Counting inversions has numerous applications:

- **Ranking similarity**: Compare how similar two rankings are (e.g., comparing movie preferences between users for recommendation systems)
- **AUC-ROC in ML**: The Area Under the ROC Curve is computed using inversion counting - it measures how well a classifier's predictions match actual labels
- **Kendall tau distance**: A statistical measure of correlation between rankings
- **Sorting complexity**: The number of inversions equals the number of swaps Bubble Sort would perform

54.2 Naive Approach

The brute-force approach compares every pair:

```
def count_inversions_naive(arr):
    count = 0
    n = len(arr)
    for i in range(n):
        for j in range(i + 1, n):
            if arr[i] > arr[j]:
                count += 1
    return count
```

This requires $\binom{n}{2} = O(n^2)$ comparisons - one for each pair.

54.3 Divide and Conquer Approach

We can count inversions in $O(n \log n)$ time using a modification of MergeSort:

1. **Divide**: Split the list into two halves
2. **Conquer**: Recursively count inversions within each half

3. **Combine**: Count inversions between the two halves (cross-inversions)
4. **Return**: Sum of left inversions + right inversions + cross-inversions

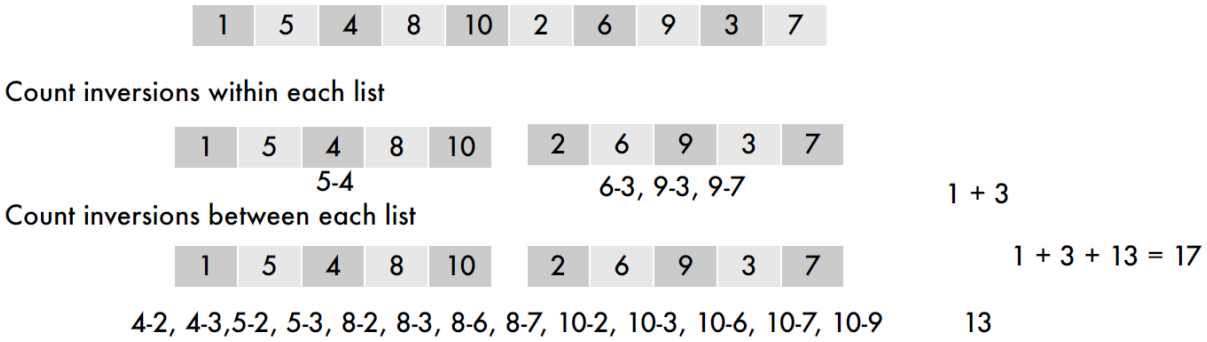

Figure 100: Inversions fall into three categories: those within the left half, those within the right half, and cross-inversions between halves. Divide and conquer handles each category.

The key insight is that counting cross-inversions becomes easy **if both halves are sorted**.

54.4 Sort-and-Count Algorithm

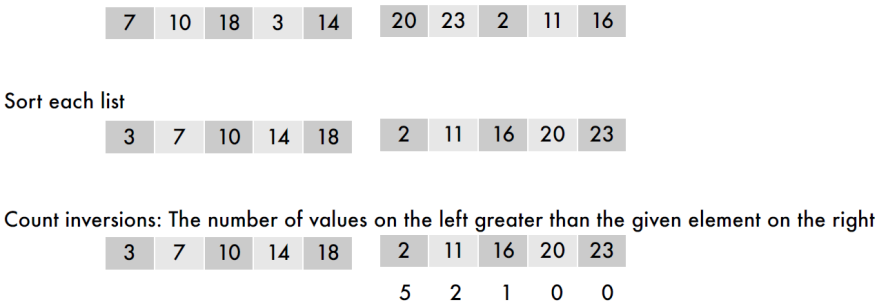

Figure 101: With sorted halves, counting cross-inversions reduces to: when taking an element from the right half during merge, count how many elements remain in the left half (all form inversions with it).

The algorithm merges while counting:

- Scan both sorted lists A and B from left to right
- Compare current elements a_i and b_j
- If $a_i \leq b_j$: element a_i is not inverted with any remaining element in B; append a_i to output
- If $a_i > b_j$: element b_j is inverted with **every remaining element** in A; add $|A| - i$ to the inversion count and append b_j to output

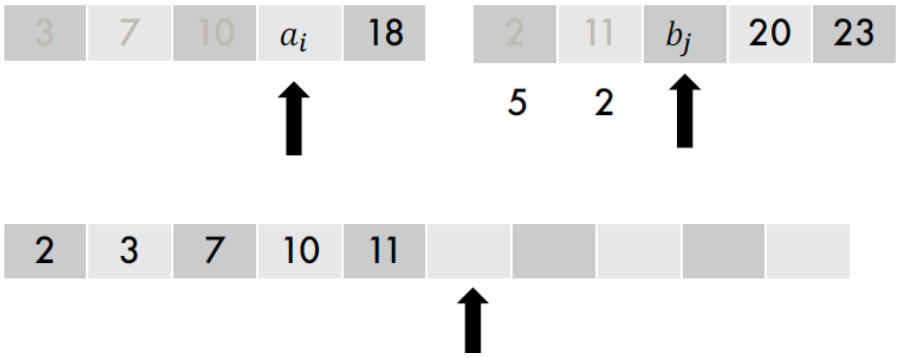

Figure 102: Merge-and-count in action: when $b_j < a_i$, all remaining elements in A (from position i onwards) form inversions with b_j. Here, when taking 11 from B, elements $\{a_i, 18\}$ remaining in A each form an inversion.

Sort-and-Count Insight

The crucial observation: since both halves are sorted, if $b_j < a_i$, then b_j is also less than $a_{i+1}, a_{i+2}, \ldots$ - so we can count all these inversions in $O(1)$ rather than checking each pair individually.

This is why sorting enables efficient counting: we leverage the sorted structure to avoid redundant comparisons.

54.5 Algorithm and Analysis

SORT-AND-COUNT(L)

IF (list L has one element)
  RETURN $(0, L)$.

Divide the list into two halves A and B.
$(r_A, A) \leftarrow$ SORT-AND-COUNT(A). ⟵ $T(n/2)$
$(r_B, B) \leftarrow$ SORT-AND-COUNT(B). ⟵ $T(n/2)$
$(r_{AB}, L) \leftarrow$ MERGE-AND-COUNT(A, B). ⟵ $\Theta(n)$

RETURN $(r_A + r_B + r_{AB}, L)$.

Figure 103: Sort-and-Count pseudocode. The algorithm returns both the inversion count and the sorted list, enabling the divide and conquer structure.

The algorithm simultaneously sorts and counts inversions:

Input: List L

Output:

1. Number of inversions in L

2. L in sorted order (needed for parent recursive calls)

Sort-and-Count Complexity

The recurrence relation is:

$$T(n) \leq \begin{cases} O(1) & \text{if } n = O(1) \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + O(n) & \text{if } n > 1 \end{cases}$$

The $O(n)$ term covers merging and counting cross-inversions (both linear in the merge step).

This is identical to MergeSort's recurrence, so Sort-and-Count runs in $O(n \log n)$ time.

54.5.1 Worked Example

Consider the array $[2, 3, 8, 6, 1]$:

1. **Divide**: $[2, 3]$ and $[8, 6, 1]$
2. **Left half** $[2, 3]$: Already sorted, 0 inversions
3. **Right half** $[8, 6, 1]$:
   - Split into $[8]$ and $[6, 1]$
   - $[6, 1]$ has 1 inversion: $(6, 1)$
   - Merging $[8]$ with $[1, 6]$: when taking 1, one element (8) remains, so 1 inversion; when taking 6, one element (8) remains, so 1 inversion
   - Right half total: $1 + 1 + 1 = 3$ inversions
4. **Merge** $[2, 3]$ with $[1, 6, 8]$:
   - Take 1: two elements remain in left $\{2, 3\}$, so 2 cross-inversions
   - Take 2: no inversion
   - Take 3: no inversion
   - Take 6, 8: no inversions
   - Cross-inversions: 2
5. **Total**: $0 + 3 + 2 = 5$ inversions

Verification: The inversions are $(2, 1), (3, 1), (8, 6), (8, 1), (6, 1)$ - indeed 5 pairs.

55 Selection: Finding the kth Smallest Element

The **selection problem** asks: given an unsorted array of n elements, find the kth smallest element.

Selection Problem

Special cases:

- $k = 1$: Find the minimum - trivially $O(n)$
- $k = n$: Find the maximum - trivially $O(n)$
- $k = \lceil n/2 \rceil$: Find the median

Key insight: Selection is fundamentally easier than sorting. We can find any order statistic in $O(n)$ time without sorting the entire array.

55.1 Applications

Selection algorithms are crucial in:

- **Order statistics**: Finding percentiles, quartiles, and medians
- **Database queries**: "Find records in the top 10th percentile"
- **Machine learning**: Outlier detection, feature selection, computing robust statistics
- **Algorithm design**: Many algorithms need medians or approximate medians as subroutines

55.2 Approaches and Their Complexities

1. **Sort then index** - $O(n \log n)$
   - Sort the array using MergeSort or similar
   - Return element at position k
   - Simple but suboptimal: we do more work than necessary
2. **Binary heap approach** - $O(n + k \log n)$ or $O(n \log k)$
   - Build a min-heap in $O(n)$ time
   - Extract minimum k times, each extraction costs $O(\log n)$
   - Alternatively, maintain a max-heap of size k while scanning
3. **QuickSelect** - $O(n)$ expected, $O(n^2)$ worst case
   - Partition-based approach, similar to QuickSort
   - Only recurse on the partition containing the kth element
   - Randomised pivot selection gives expected linear time
4. **Median of Medians** - $O(n)$ worst case
   - Deterministic linear-time selection
   - Guarantees a good pivot, eliminating worst-case quadratic behaviour

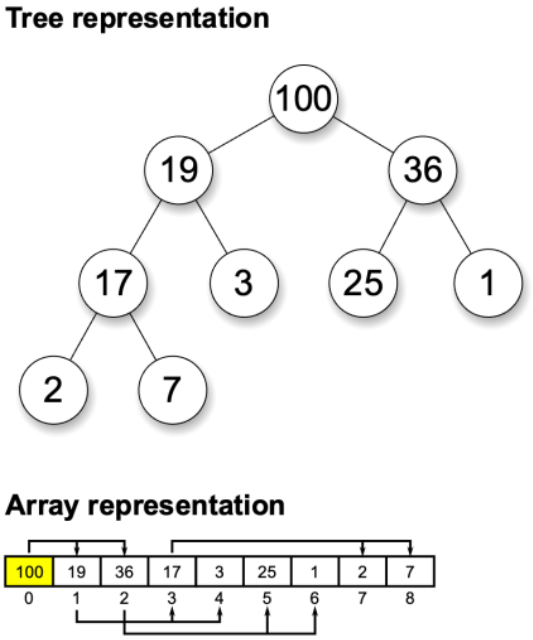

Figure 104: A binary heap structure. In a max-heap, each parent is larger than its children. Heaps enable efficient extraction of extremal elements.

55.3 3-Way Partitioning

The foundation of QuickSort and QuickSelect is the **partitioning** operation.

3-Way Partitioning

Given an array and a pivot element p, 3-way partitioning rearranges elements into three groups:

- **Left**: Elements $< p$
- **Middle**: Elements $= p$
- **Right**: Elements $> p$

The pivot ends up in its **correct sorted position**. This operation runs in $O(n)$ time and $O(1)$ space.

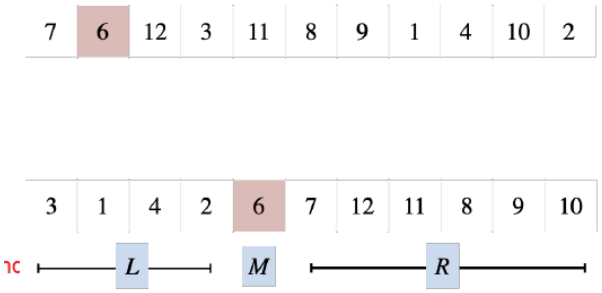

Figure 105: 3-way partitioning: elements are rearranged so all elements less than pivot p are on the left, elements equal to p are in the middle, and elements greater than p are on the right.

55.3.1 The Partitioning Algorithm

1. **Choose pivot**: Select element p (often randomly or swap chosen element to first position)
2. **Initialise pointers**:
   - lt (less than): boundary for elements $< p$, starts at position 0
   - gt (greater than): boundary for elements $> p$, starts at position $n - 1$
   - i: current element being examined
3. **Partitioning loop**: Scan with index i from left to right:
   - If $A[i] < p$: swap $A[i]$ with $A[lt]$, increment both lt and i
   - If $A[i] > p$: swap $A[i]$ with $A[gt]$, decrement gt (do not increment i - the swapped element needs examination)
   - If $A[i] = p$: just increment i
4. **Terminate**: When $i > gt$, the array is partitioned

3-Way Partitioning Complexity

Time: $O(n)$ - each element is examined at most twice (once when i reaches it, possibly once more if swapped from the right)

Space: $O(1)$ - partitioning is done in-place using only pointer variables

3-way partitioning is particularly efficient when there are many duplicate elements, as all duplicates of the pivot are grouped together and excluded from recursive calls.

55.4 QuickSelect Algorithm

QuickSelect uses partitioning to find the kth smallest element without fully sorting:

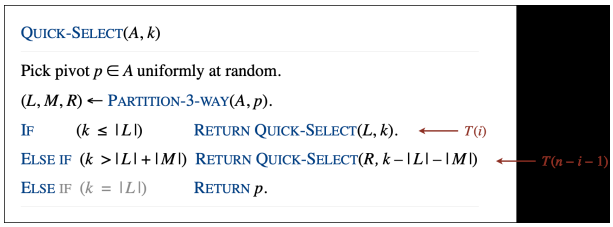

Figure 106: QuickSelect partitions around a random pivot, then recurses only into the partition containing position k. Unlike QuickSort, only one recursive call is made.

1. **Choose a random pivot** p
2. **3-way partition** into L (elements $< p$), M (elements $= p$), R (elements $> p$)
3. **Recurse on the relevant partition**:
   - If $k \leq |L|$: recurse on L to find the kth smallest
   - If $|L| < k \leq |L| + |M|$: the pivot p is the answer (it's in position k)
   - If $k > |L| + |M|$: recurse on R to find the $(k - |L| - |M|)$th smallest

QuickSelect vs QuickSort

The key difference: QuickSort recurses on both partitions; QuickSelect recurses on only one.

This single-recursion structure is why QuickSelect achieves $O(n)$ expected time rather than $O(n \log n)$.

55.5 Analysis of Randomised QuickSelect

55.5.1 Intuition: The Stick-Breaking Analogy

Consider breaking a stick at a random point:

- On average, you get pieces of size 1/4 and 3/4 (not 1/2 and 1/2)
- In the worst case for selection, we recurse on the larger piece
- So we expect to work on a 3/4 fraction of the array each iteration

55.5.2 Recurrence and Solution

Let $T(n)$ be the expected number of comparisons to select from an array of size n.

In the worst case over all k values:

$$T(n) \leq T\left(\frac{3n}{4}\right) + cn$$

where cn is the cost of partitioning.

Expanding this recurrence:

$$T(n) \leq cn + \frac{3}{4}cn + \left(\frac{3}{4}\right)^2 cn + \left(\frac{3}{4}\right)^3 cn + \cdots$$

This is a geometric series:

$$T(n) \leq cn \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = cn \cdot \frac{1}{1 - 3/4} = 4cn$$

QuickSelect Expected Complexity

Randomised QuickSelect runs in $O(n)$ expected time.

More precisely, the expected number of comparisons is at most $4n$.

Worst case: $O(n^2)$ if we consistently choose bad pivots (e.g., always the maximum element). However, randomisation makes this exponentially unlikely.

Summary: Selection Complexities

Algorithm	Expected	Worst Case
Sort then index	$O(n \log n)$	$O(n \log n)$
Heap-based	$O(n \log k)$	$O(n \log k)$
Randomised QuickSelect	$O(n)$	$O(n^2)$
Median of Medians	$O(n)$	$O(n)$

56 Closest Pair of Points

The **closest pair problem** asks: given n points in 2D, find the pair with minimum Euclidean distance.

Closest Pair Problem

Input: n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in the plane

Output: The pair of points with minimum distance

Applications:

- Nearest-neighbour matching in causal inference
- k-nearest neighbour algorithms
- Collision detection in graphics/games
- Clustering algorithms

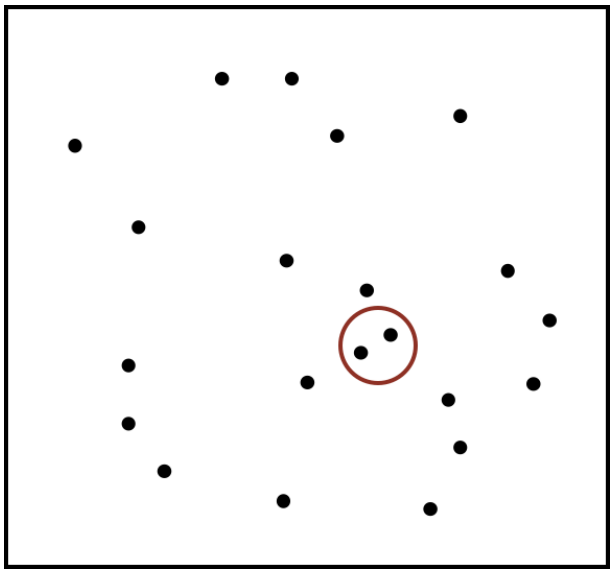

Figure 107: The closest pair problem in 2D: among all $\binom{n}{2}$ pairs of points, find the pair with minimum Euclidean distance.

56.1 Brute Force Approach

The naive approach checks all pairs:

```
def closest_pair_naive(points):
    min_dist = infinity
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            d = distance(points[i], points[j])
            if d < min_dist:
                min_dist = d
                closest = (points[i], points[j])
    return closest
```

This requires $\binom{n}{2} = O(n^2)$ distance computations.

56.2 Divide and Conquer Solution

We can solve this in $O(n \log n)$ time:

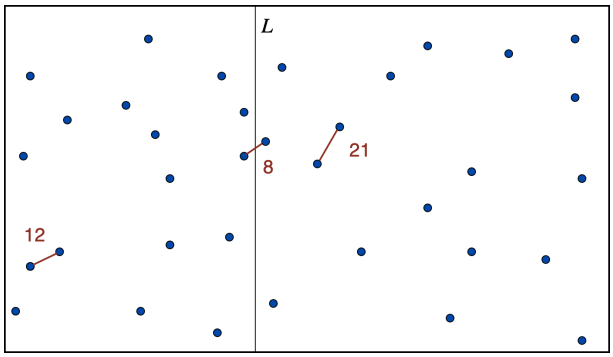

Figure 108: Divide and conquer for closest pair: split points by a vertical line L, recursively find closest pairs in each half (distances 12 and 21), then check for closer pairs straddling the line (distance 8).

1. **Sort points** by x-coordinate: $O(n \log n)$ preprocessing
2. **Divide**: Draw a vertical line L through the median x-coordinate, splitting points into left and right halves
3. **Conquer**: Recursively find the closest pair in each half
   - Let δ_1 = closest distance in left half
   - Let δ_2 = closest distance in right half
   - Let $\delta = \min(\delta_1, \delta_2)$
4. **Combine**: Check for closer pairs that straddle the dividing line
   - Only points within distance δ of the line can be part of such a pair
   - This defines a "strip" of width 2δ centred on L

The Combine Step Challenge

The naive combine step would check all pairs of points in the strip - potentially $O(n^2)$ if all points lie near the dividing line.

The clever insight: after sorting strip points by y-coordinate, each point needs only be compared with a **constant number** of neighbours.

56.3 Why Only 7 Neighbours?

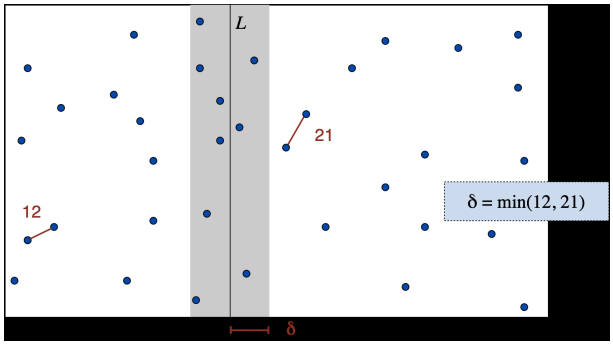

Figure 109: In the combine step, we only consider points within distance δ of the dividing line. After sorting these by y-coordinate, each point needs comparison with at most 7 subsequent points.

Consider a point p in the strip. Any point closer than δ to p must lie within a $2\delta \times \delta$ rectangle (width 2δ for the strip, height δ above p).

This rectangle can be divided into 8 cells of size $\delta/2 \times \delta/2$. By the definition of δ, each cell contains **at most one point** (two points in the same cell would be closer than δ, contradicting our recursive solutions).

Therefore, at most 7 other points can be within distance δ of p in the strip. Checking 7 neighbours per point gives $O(n)$ comparisons for the combine step.

56.4 Complete Algorithm

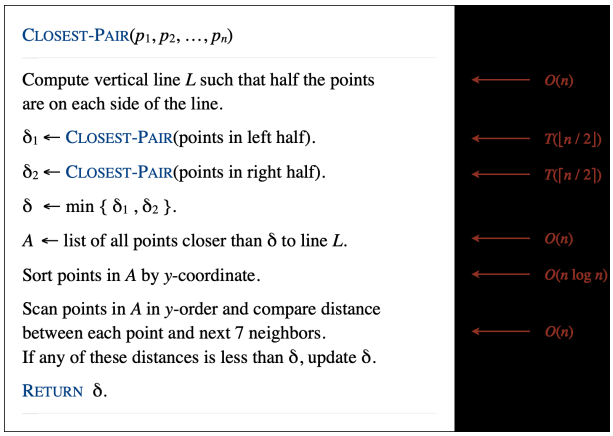

Figure 110: Complete closest pair algorithm with complexity annotations. The sorting within the strip can be done in $O(n)$ per level by maintaining a separate list sorted by y-coordinate.

Closest Pair Complexity

The recurrence relation is:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 3 \\ T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + O(n) & \text{if } n > 3 \end{cases}$$

The $O(n)$ combine step includes:

- Finding the dividing line: $O(1)$ (median of sorted list)
- Building the strip: $O(n)$
- Sorting strip by y-coordinate: $O(n)$ (using merge from pre-sorted lists)
- Checking pairs in strip: $O(n)$ (constant neighbours per point)

By the Master Theorem: $T(n) = O(n \log n)$.

Closest Pair Summary

The closest pair algorithm achieves $O(n \log n)$ complexity, matching the sorting lower bound. This is optimal for comparison-based algorithms since we must at least "see" all points.

Key techniques:

- Divide by median x-coordinate
- Recursively solve subproblems
- Clever combine: only check strip, only check 7 neighbours

57 Computational Geometry Applications

Divide and conquer is particularly powerful for geometric problems. Many problems that seem to require $O(n^2)$ or worse can be solved in $O(n \log n)$ using this paradigm.

Problem	Brute Force	Divide & Conquer
Closest Pair	$O(n^2)$	$O(n \log n)$
Farthest Pair	$O(n^2)$	$O(n \log n)$
Convex Hull	$O(n^2)$	$O(n \log n)$
Delaunay Triangulation	$O(n^4)$	$O(n \log n)$
Voronoi Diagram	$O(n^4)$	$O(n \log n)$
Euclidean MST	$O(n^2)$	$O(n \log n)$

Table 1: Complexity improvements from divide and conquer in computational geometry. The dramatic improvement for Delaunay/Voronoi from $O(n^4)$ to $O(n \log n)$ is particularly striking.

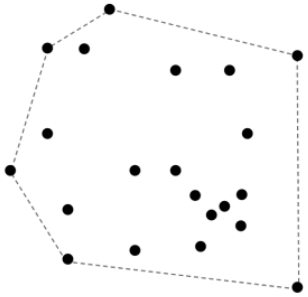

Figure 111: Convex hull: the smallest convex polygon containing all points. Can be computed in $O(n \log n)$ using divide and conquer (e.g., the "gift wrapping" or Graham scan algorithms).

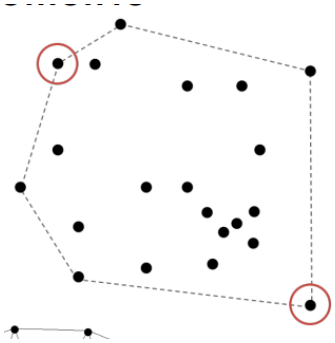

Figure 112: Farthest pair problem (computing the diameter): find the two points with maximum distance. The farthest pair always lies on the convex hull, enabling an $O(n \log n)$ algorithm.

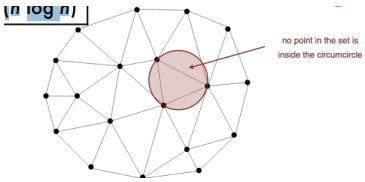

Figure 113: Delaunay triangulation: connects points such that no point lies inside the circumcircle of any triangle. Used in mesh generation, terrain modelling, and as a dual to Voronoi diagrams.

58 The Master Theorem: General Theory

The Master Theorem provides a systematic method for solving recurrence relations of the divide and conquer form.

58.1 The General Recurrence

Master Theorem Setup

Consider recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^c)$$

where:

- $a \geq 1$: number of subproblems at each recursive level
- $b > 1$: factor by which problem size shrinks
- $c \geq 0$: exponent in the cost of dividing/combining

From these parameters, we can derive:

- **Depth of recursion tree**: $1 + \log_b n$ levels
  The problem size at level i is n/b^i. We reach base case when $n/b^i = 1$, i.e., $i = \log_b n$.
- **Size of subproblem at level i**: n/b^i
  Each level divides by b.
- **Number of subproblems at level i**: a^i
  Each problem spawns a subproblems, so level i has a^i subproblems.
- **Work at level i**: $a^i \cdot (n/b^i)^c = n^c \cdot (a/b^c)^i$
  This counts all a^i subproblems, each of size n/b^i with cost $(n/b^i)^c$.

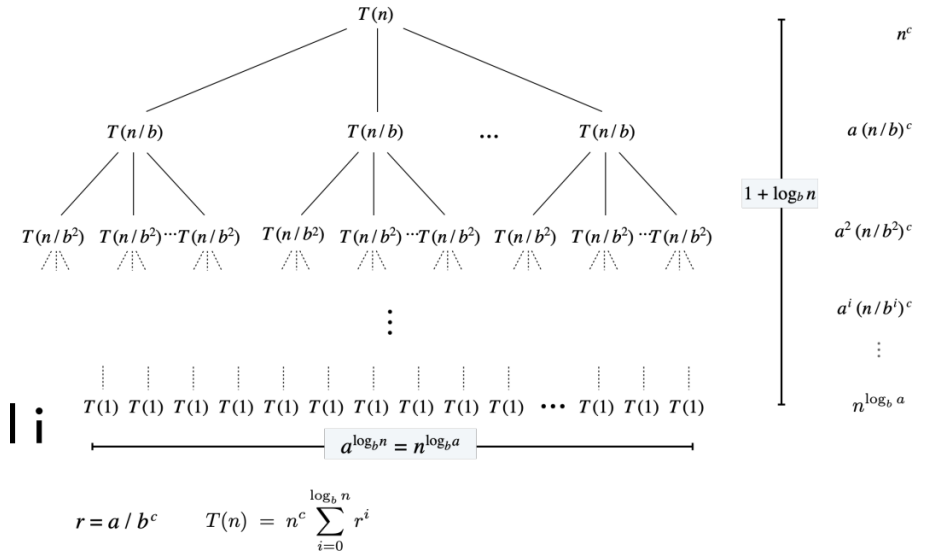

Figure 114: Recursion tree for the general divide and conquer recurrence. The total work is $T(n) = n^c \sum_{i=0}^{\log_b n} r^i$ where $r = a/b^c$.

58.2 The Critical Ratio

The key quantity is the **ratio** $r = a/b^c$, which determines whether work increases or decreases as we descend the tree:

Total Work Formula

The total work across all levels is:

$$T(n) = n^c \sum_{i=0}^{\log_b n} r^i \quad \text{where } r = \frac{a}{b^c}$$

This is a geometric series. Its behaviour depends on whether $r < 1$, $r = 1$, or $r > 1$.

58.3 The Three Cases

Master Theorem: Three Cases

Let $r = a/b^c$. The solution to $T(n) = aT(n/b) + O(n^c)$ is:

1. **Case 1** ($r < 1$, equivalently $c > \log_b a$): Work decreases down the tree

   $$T(n) = O(n^c)$$

   Root-dominated: Most work happens at the top level.

2. **Case 2** ($r = 1$, equivalently $c = \log_b a$): Work is equal at each level

   $$T(n) = O(n^c \log n)$$

   Balanced: Work is evenly distributed, with $\log n$ levels contributing equally.

3. **Case 3** ($r > 1$, equivalently $c < \log_b a$): Work increases down the tree

   $$T(n) = O(n^{\log_b a})$$

   Leaf-dominated: Most work happens at the bottom (base cases).

58.3.1 Case 1: Root Domination ($r < 1$)

When $a < b^c$, the work at each level forms a decreasing geometric series. The sum is dominated by the first (largest) term:

$$\sum_{i=0}^{\log_b n} r^i \leq \sum_{i=0}^{\infty} r^i = \frac{1}{1-r} = O(1)$$

Thus $T(n) = O(n^c)$.

Example: $T(n) = T(n/2) + n^2$ has $a = 1$, $b = 2$, $c = 2$. Since $1 < 2^2 = 4$, we have $r < 1$, so $T(n) = O(n^2)$.

58.3.2 Case 2: Equal Distribution ($r = 1$)

When $a = b^c$, each level contributes equally:

$$\sum_{i=0}^{\log_b n} r^i = \sum_{i=0}^{\log_b n} 1 = 1 + \log_b n = O(\log n)$$

Thus $T(n) = O(n^c \log n)$.

Example: MergeSort has $T(n) = 2T(n/2) + n$ with $a = 2$, $b = 2$, $c = 1$. Since $2 = 2^1$, we have $r = 1$, so $T(n) = O(n \log n)$.

58.3.3 Case 3: Leaf Domination ($r > 1$)

When $a > b^c$, the work at each level forms an increasing geometric series. The sum is dominated by the last (largest) term:

$$\sum_{i=0}^{\log_b n} r^i = \frac{r^{1+\log_b n} - 1}{r - 1} = O(r^{\log_b n}) = O\left(\frac{a^{\log_b n}}{b^{c \log_b n}}\right) = O\left(\frac{n^{\log_b a}}{n^c}\right)$$

Thus $T(n) = n^c \cdot O(n^{\log_b a - c}) = O(n^{\log_b a})$.

Example: Binary search tree traversal has $T(n) = 2T(n/2) + 1$ with $a = 2$, $b = 2$, $c = 0$. Since $2 > 2^0 = 1$, we have $r > 1$, so $T(n) = O(n^{\log_2 2}) = O(n)$.

Master Theorem Quick Reference

Given $T(n) = aT(n/b) + O(n^c)$:

Condition	Interpretation	Result
$c > \log_b a$	Root-dominated	$T(n) = O(n^c)$
$c = \log_b a$	Balanced	$T(n) = O(n^c \log n)$
$c < \log_b a$	Leaf-dominated	$T(n) = O(n^{\log_b a})$

59 Parallelisation of Divide and Conquer

Divide and conquer algorithms are naturally suited to parallel execution: once a problem is divided into independent subproblems, these can be solved simultaneously on different processors.

Parallelisation Principle

Key insight: Parallelisation improves wall-clock time, not total work.

If you have p processors, you can potentially achieve a p-fold speedup, but the total number of operations remains the same. Parallelisation is about redistribution of work, not reduction of work.

59.1 Embarrassingly Parallel Problems

An "embarrassingly parallel" task divides naturally into independent subtasks requiring minimal coordination:

1. **High independence**: Subtasks do not depend on each other's results during execution
2. **Minimal communication**: Little or no data exchange between parallel tasks
3. **Simple combination**: Merging results incurs low overhead
4. **Linear scalability**: Adding processors proportionally reduces time

59.1.1 Example: Training a Random Forest

Random forests exemplify embarrassingly parallel computation:

- **Independent trees**: Each decision tree is trained independently on a bootstrap sample
- **No communication**: Trees do not share information during training
- **Simple aggregation**: Final predictions combine via majority voting or averaging

With p processors and T trees, training time reduces from $O(T \cdot \text{tree\_cost})$ to $O(\lceil T/p \rceil \cdot \text{tree\_cost})$.

Parallelisation Overhead

Parallelisation is not free:

- **Communication cost**: Distributing data and collecting results takes time
- **Synchronisation**: Waiting for the slowest subtask (load balancing issues)
- **Amdahl's Law**: Serial portions of the algorithm limit maximum speedup

The cost of distributing and combining work corresponds to the $O(n^c)$ term in our recurrence. For parallelisation to be worthwhile, this overhead must be small relative to the parallel gains.

60 Matrix Multiplication

Matrix multiplication is a fundamental operation with applications throughout scientific computing, machine learning, and computer graphics. The standard algorithm has cubic complexity, but divide and conquer approaches can improve upon this.

60.1 Standard Matrix Multiplication

For two $n \times n$ matrices A and B, the product $C = AB$ has entries:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

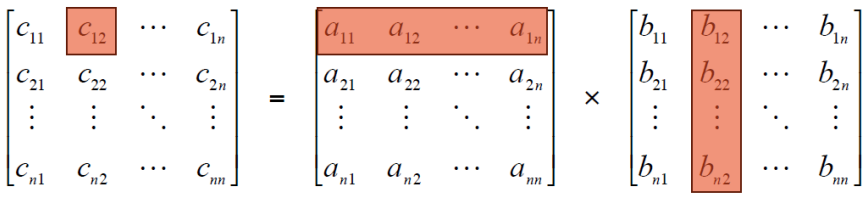

Figure 115: Computing one entry of the product matrix: C_{12} is the dot product of row 1 of A with column 2 of B, requiring n multiplications and $n-1$ additions.

Standard Matrix Multiplication Complexity

Computing $C = AB$ for $n \times n$ matrices:

- Each entry C_{ij} requires n multiplications and $n-1$ additions: $O(n)$
- There are n^2 entries to compute
- **Total**: $O(n^3)$ arithmetic operations

60.2 Block Matrix Multiplication

We can view matrix multiplication in terms of submatrices (blocks):

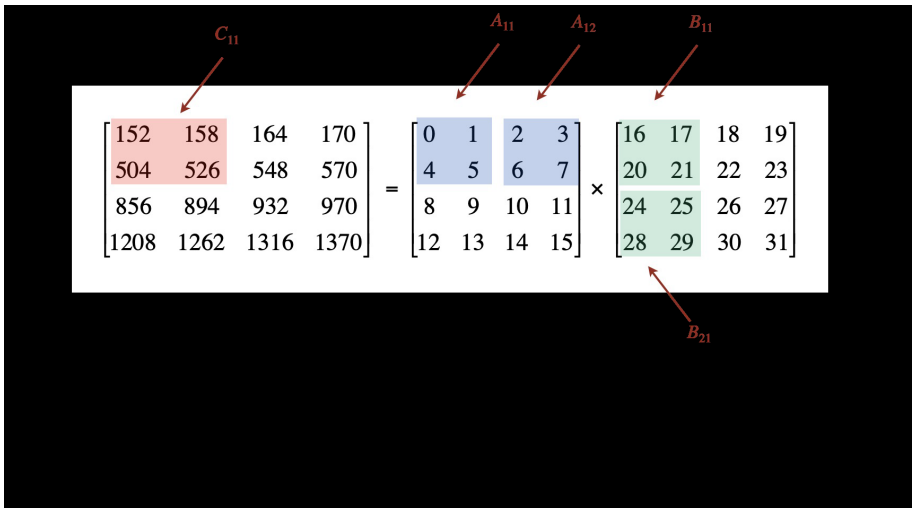

Figure 116: Block matrix multiplication: partition each matrix into four $(n/2) \times (n/2)$ blocks. The product can be computed using 8 recursive multiplications of half-sized matrices plus additions.

Partition A, B, and C into four $(n/2) \times (n/2)$ blocks:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

The block products are:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

This gives the recurrence:

$$T(n) = 8T(n/2) + O(n^2)$$

Here $a = 8$ (eight recursive multiplications), $b = 2$ (half-sized subproblems), and $c = 2$ (matrix additions are $O(n^2)$).

Since $\log_2 8 = 3 > 2 = c$, by Case 3 of the Master Theorem:

$$T(n) = O(n^{\log_2 8}) = O(n^3)$$

This naive divide and conquer approach gives no improvement over the standard algorithm.

60.3 Strassen's Algorithm

Strassen's Key Insight

Strassen (1969) discovered that the four block products can be computed using only **7 multiplications** instead of 8, at the cost of more additions.

Since multiplications dominate the complexity, this reduces the exponent.

Strassen's algorithm computes seven "helper" matrices:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Then the result blocks are:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Strassen's Algorithm Complexity

The recurrence is:

$$T(n) = 7T(n/2) + O(n^2)$$

With $a = 7$, $b = 2$, $c = 2$:

$$\log_2 7 \approx 2.807 > 2 = c$$

By Case 3 of the Master Theorem:

$$T(n) = O(n^{\log_2 7}) = O(n^{2.807})$$

This is a significant improvement over $O(n^3)$ for large matrices.

Strassen's Algorithm in Practice

While asymptotically faster, Strassen's algorithm has practical limitations:

- Higher constant factors make it slower for small matrices
- Numerical stability can be slightly worse
- Memory access patterns are less cache-friendly
- Typically only used for matrices larger than ~500–1000

Modern implementations often use Strassen at top levels and switch to standard multiplication for smaller blocks.

60.4 Significance of Matrix Multiplication Complexity

Matrix multiplication is a fundamental primitive. Many important operations have the **same complexity** as matrix multiplication:

- Matrix squaring: $A^2 = A \times A$
- Matrix inversion: A^{-1}
- Determinant computation: $\det(A)$
- Matrix rank
- Solving linear systems: $Ax = b$
- LU decomposition
- Least squares: $\min_x \|Ax - b\|^2$

Any improvement in matrix multiplication complexity automatically improves all these operations. Currently, the best known algorithm achieves approximately $O(n^{2.373})$, though the constant factors make it impractical.

61 Summary: The Divide and Conquer Paradigm

Divide and Conquer: Key Takeaways

The Pattern:

1. Divide the problem into smaller subproblems
2. Conquer each subproblem recursively
3. Combine solutions into the final answer

When to Use:

- Problem has natural recursive structure
- Subproblems are independent
- Combining solutions is efficient

Analysis Tool: The Master Theorem handles recurrences $T(n) = aT(n/b) + O(n^c)$

Algorithm	Recurrence	Complexity	Improvement Over Naive
MergeSort	$2T(n/2) + O(n)$	$O(n \log n)$	vs $O(n^2)$
Counting Inversions	$2T(n/2) + O(n)$	$O(n \log n)$	vs $O(n^2)$
QuickSelect	$T(3n/4) + O(n)$	$O(n)$	vs $O(n \log n)$
Closest Pair	$2T(n/2) + O(n)$	$O(n \log n)$	vs $O(n^2)$
Strassen's	$7T(n/2) + O(n^2)$	$O(n^{2.807})$	vs $O(n^3)$

Table 2: Summary of divide and conquer algorithms covered, their recurrences, complexities, and improvements over naive approaches.

The divide and conquer paradigm demonstrates a powerful principle: by carefully structuring how we decompose and recombine problems, we can achieve dramatic algorithmic improvements. The $O(n \log n)$ algorithms for sorting, counting inversions, and closest pair all exploit the same recursive structure, while QuickSelect shows that when we only need partial information (the kth element rather than full sorted order), we can do even better.

62 Graphs

63 What is a Graph?

A graph G is a mathematical structure used to model pairwise relations between objects. It consists of two components:

1. **Vertices (or nodes),** V: These represent the objects in the graph.
2. **Edges,** E: These are the connections or relationships between pairs of objects (nodes) in the graph.

$G = (V, E)$ (a tuple object, with 2 lists)

Size: $n = |V|$, $m = |E|$

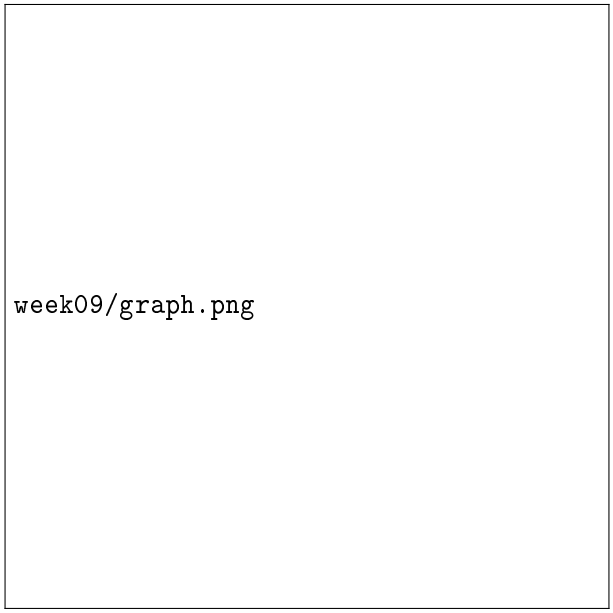

Figure 117: An undirected graph with 8 vertices and 11 edges.

For the graph in Figure 117:

$$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$
$$E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6, 7-8\}$$
$$m = 11$$
$$n = 8$$

Graph - Formal Definition

A **graph** is an ordered pair $G = (V, E)$ where:

- V is a set of **vertices** (or nodes)
- E is a set of **edges**, which are 2-element subsets of V (for undirected graphs) or ordered pairs from $V \times V$ (for directed graphs)

The **order** of a graph is $|V| = n$ (number of vertices).

The **size** of a graph is $|E| = m$ (number of edges).

Graphs can be used to model a wide variety of systems in many fields, including computer networks, social networks, biological networks, and transportation networks, among others.

Edges can be **directed** or **undirected**, representing the nature of the relationship (unidirectional or bidirectional), and can also have **weights** assigned to them, which can represent the strength or capacity of the connection.

Note: you can also use graphs to express measures of similarity as a network - e.g. kernels; strength of ties between units, which can then be analysed as a network.

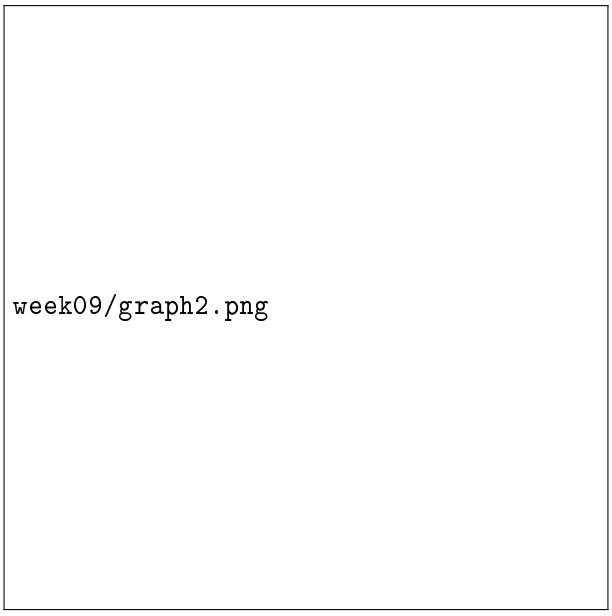

Figure 118: A more complex graph illustrating various connection patterns.

64 Representation of a Graph

There are two primary ways to represent a graph in a computer: the **adjacency matrix** and the **adjacency list**. The choice between them depends on the graph's characteristics and the operations you need to perform.

Table 3: Graph Types with Nodes and Edges - Real-World Examples

Graph Type	**Nodes**	**Edges**
Communication	Telephone, Computer	Fibre Optic Cable
Circuit	Gate, Register, Processor	Wire
Mechanical	Joint, Rod, Beam, Spring	Physical connections
Financial	Stock, Currency	Transactions
Transportation	Street Intersection, Airport	Highway, Airway Route
Internet	Class C Network	Connection
Game	Board Position	Legal Move
Social	Person, Actor	Friendship, Movie Cast
Neural Network	Neuron	Synapse
Protein Network	Protein	Protein-Protein Interaction
Molecule	Atom	Bond

64.1 Adjacency Matrix

An $n \times n$ matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

- **Representation**: Each row and column represent a vertex. If there is an edge between vertex i and vertex j, then the matrix entry $A[i][j] = 1$; otherwise, $A[i][j] = 0$.

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

  This is the adjacency matrix for the graph in Figure 117.

- **Space Complexity**: The space complexity is $O(n^2)$ regardless of the number of edges, making it **less space-efficient for sparse graphs** (graphs with few edges relative to the number of vertices).

- **Time Complexity**:
  - Checking if an edge exists between two vertices is $O(1)$, as you only need to look at the corresponding matrix element.
  - Identifying all edges takes $O(n^2)$ because you must inspect each entry in the matrix.

Adjacency Matrix Complexity

Space: $O(n^2)$

Check if edge exists: $O(1)$

Enumerate all edges: $O(n^2)$

Best for: **dense graphs**, frequent edge queries

64.2 Adjacency List

Represents a graph as a node-indexed array of lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

Figure 119: Adjacency list representation of the graph from Figure 117. Each vertex maps to a list of its neighbours.

Note: the order of the nodes within the linked lists is not important.

- **Representation**: The graph is represented as an array of lists (or linked lists). Each list at array index i contains a list of nodes that are adjacent to vertex i.
- **Space Complexity**: The space complexity is $O(m+n)$, where m is the number of edges and n is the number of vertices. This makes it **more space-efficient** than an adjacency matrix, **especially for sparse graphs**.
- **Time Complexity**:
  - Checking if an edge exists between vertex i and j can take $O(\text{degree}(i))$, as it requires a search through the list at index i.
  - Identifying all edges takes $O(n+m)$ because each vertex and each edge is explored once.

*Additionally, as a linked list, it allows for **easy addition of new nodes** by simply adding a pointer, which can be done in constant time.*

Adjacency List Complexity

Space: $O(n+m)$

Check if edge exists: $O(\text{degree}(i))$

Enumerate all edges: $O(n+m)$

Best for: **sparse graphs**, graph traversal algorithms

64.3 Choosing Between the Two

Graph Representation Summary

Adjacency Matrix - use when:

- Graph is **dense** (edge-to-vertex ratio is high)
- **Frequent edge existence checks** are needed ($O(1)$ lookup)
- Algorithm needs to **frequently access edge weights**

Adjacency List - use when:

- Graph is **sparse** (few edges relative to vertices)
- Need to **explore edges of specific nodes** (e.g., BFS, DFS)
- Memory efficiency is important

Adjacency Matrix:

- Preferable for **dense graphs** where the edge-to-vertex ratio is high.
- Better when **frequent edge existence checks are needed**, as these can be done in constant time.
- Simpler for implementing algorithms that need to **frequently access edge weights**, like many algorithms in numerical simulations and optimisations.

Adjacency List:

- Preferable for **sparse graphs** with few edges relative to vertices.
- More space-efficient when the graph is large and mostly consists of empty connections.
- Often **faster for algorithms that explore the edges of specific nodes**, such as most graph traversal algorithms like **DFS and BFS**, especially when the node degree is much less than the number of nodes.

65 Paths and Connectivity

65.1 Paths

Path - Formal Definition

A **path** in an undirected graph is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that for each pair of consecutive vertices v_i and v_{i+1}, there is an edge $\{v_i, v_{i+1}\} \in E$.

A path is **simple** if each vertex appears at most once (no repeated vertices).

The **length** of a path is the number of edges it contains, which is $k - 1$ for a path with k vertices.

- **Path**: A sequence of nodes connected by edges.
- **Simple Path**: A path where **each node is distinct** - i.e., it does not contain any repeated vertices. This excludes the possibility of looping back to a previously visited vertex within the same path.

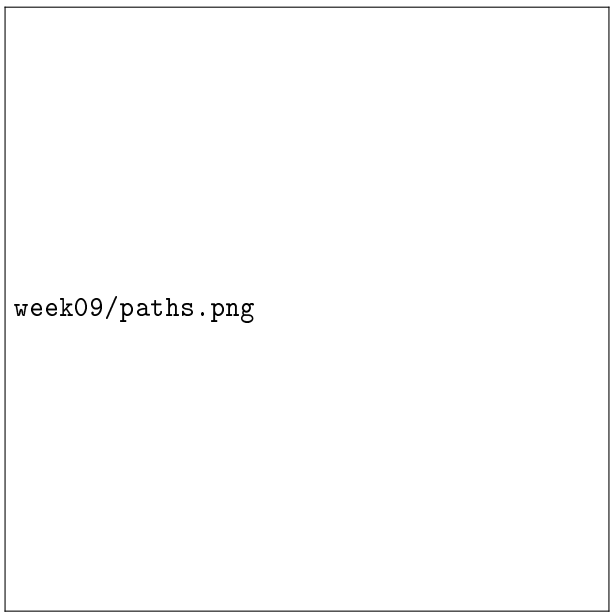

Figure 120: A simple path through a graph - each vertex is visited at most once.

65.2 Connectivity

- **Connected Graph**: An undirected graph is connected if there is **at least one path between any two** distinct nodes in the graph. This means that you can **start at any vertex and reach any other vertex** through some sequence of edges.
- **Components**: If an undirected graph is not connected, it can be decomposed into **connected components**. Each component is a maximal connected subgraph, meaning that within a component, any two vertices are connected by paths, and no additional edges or vertices outside of the component can be included without losing the property of connectivity.

Properties of Paths and Connectivity

- **Fundamental Property**: In a connected graph, there exists at least one path between any two nodes. This property is crucial for many algorithms in graph theory, such as those used for network analysis, routing, and resource distribution.
- **Graph Traversals**: Depth-First Search (DFS) and Breadth-First Search (BFS) are two fundamental graph traversal techniques that can be used to explore all vertices and edges of a graph to determine connectivity, find connected components, or simply to visit all nodes for various computational purposes.

65.3 Cycles

Cycle - Formal Definition

A **cycle** is a path $v_1, v_2, \ldots, v_k$ where $v_1 = v_k$ (the path starts and ends at the same vertex).

A cycle is **simple** if it does not contain any repeated vertices or edges, except that the starting and ending vertex is the same, and $k \geq 4$ (at least 3 distinct vertices).

- **Cycle**: A path that **starts and ends at the same node**.
- **Simple Cycle**: A cycle that **does not contain any repeated** vertices or edges, except that the starting and ending vertex is the same.

Figure 121: A simple cycle - the path returns to its starting vertex without repeating any intermediate vertex.

66 Trees

A tree is an **undirected** graph that is **connected** and **acyclic**. This means that there is **exactly one path between any two vertices**, and no path leads back to its starting vertex except the

trivial path.

From any point, once you start moving, you are **monotonically moving away** from your starting location - you cannot return without retracing your steps.

Tree Characterisation

Any **two** of these properties of an undirected graph G imply the third:

1. **Connected**: There is a path between every pair of vertices
2. **Acyclic**: The graph contains no cycles
3. $n-1$ **edges**: A graph with n vertices has exactly $n-1$ edges

66.1 Properties of Trees

- **Connected**: There is a path between every pair of vertices in the tree. No vertex is isolated.
- **Acyclic**: The graph does not contain any cycles, ensuring there is only one path between any two nodes.
- **Edge Count** $n-1$: A tree with n vertices always has $n-1$ edges.
  - This is the minimal number of edges needed to connect all vertices without creating a cycle, and it is a key property that helps in characterising trees.

Figure 122: A tree - an undirected, connected, acyclic graph.

66.2 Types of Trees

- **Rooted Trees**: Trees can also be discussed as rooted, where one vertex is designated as the root, and the directionality can be implied (though not explicitly directed) from the

root outward. This hierarchy allows for representing structures like organisational charts, decision trees, and family trees.

Figure 123: Rooted trees - the same tree structure with different vertices designated as the root.

Applications of rooted trees include:

- Decision trees
- Binary search trees
- GUI architectures
- Library Dewey Decimal system
- (many other applications)

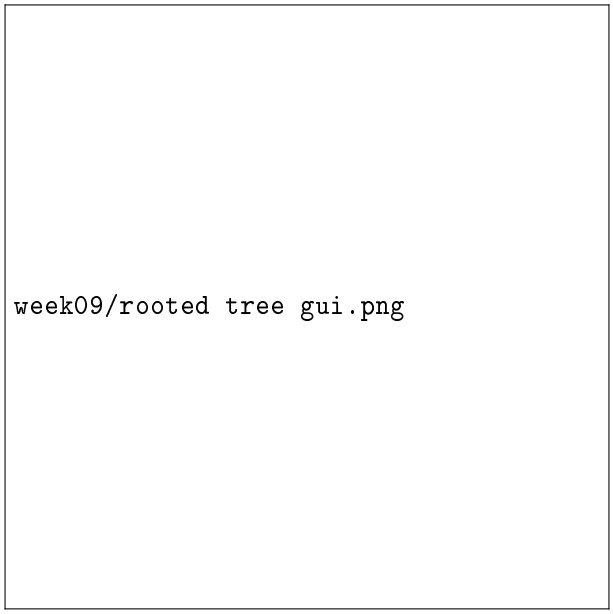

Figure 124: GUI hierarchy as a rooted tree; this is how HTML hierarchy works - each element is a node with parent-child relationships.

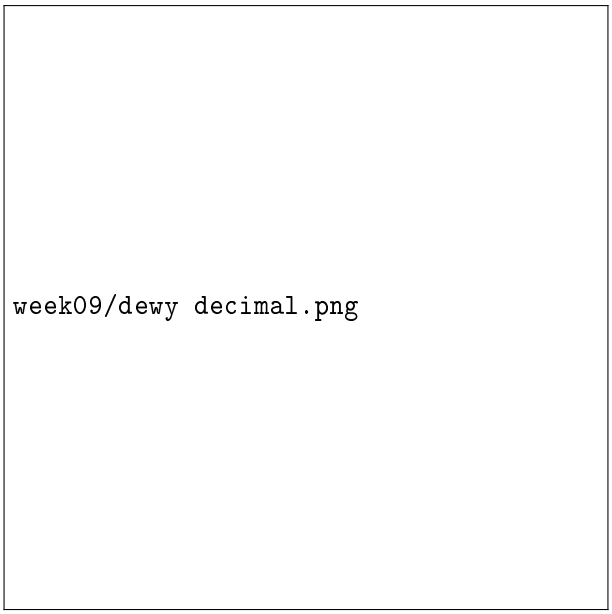

Figure 125: Dewey Decimal system as a rooted tree - hierarchical classification of knowledge.

- **Binary Trees**: A popular type of rooted tree where each node has at most two children. This is widely used in computer science, particularly for binary search trees and binary heaps.
- **Spanning Trees**: A spanning tree of a graph is a subgraph that includes all the vertices of the original graph, is a tree, and is connected. This concept is crucial in network design and optimisation algorithms, ensuring minimal connection without redundancy.

66.3 Applications of Trees

- **Data Structures**: Trees form the basis of several fundamental data structures, including binary search trees, heaps, and B-trees, all crucial for efficient searching, sorting, and indexing operations.
- **Network Routing**: Tree structures are utilised in routing algorithms to ensure efficient and cycle-free paths for data communication.
- **Algorithm Design**: Trees are used in a variety of algorithms, particularly those that involve hierarchical structures, such as recursive algorithms and divide-and-conquer strategies.

67 s-t Connectivity Problems

These problems focus on the existence and properties of paths between two specific nodes in a graph.

67.1 s-t Connectivity Problem

- **Problem Statement**: Given two nodes, s and t, determine whether there is a path connecting them.

- **Purpose**: This problem helps in understanding whether two points in a network are accessible to each other.
- **Methods of Solution**:
  - **Depth-First Search (DFS)** or **Breadth-First Search (BFS)** can be used in an unweighted graph to find whether a path exists between s and t.
  - These methods will traverse the graph starting from node s and attempt to reach node t, confirming connectivity if t is reached.

67.2 s-t Shortest Path Problem

- **Problem Statement**: Given two nodes, s and t, determine the shortest path between them, typically in terms of the least number of edges (unweighted) or minimum path weight (weighted).
- **Purpose**: This problem is critical for routing and navigation, where the goal is to find the most efficient route between points.
- **Methods of Solution**:
  - **BFS**: For unweighted graphs, BFS finds the shortest path in terms of number of edges.
  - **Dijkstra's Algorithm**: Efficient for finding the shortest paths from a single source node s to all other nodes in a graph with non-negative edge weights.
  - **Bellman-Ford Algorithm**: Handles graphs with negative weights and computes the shortest paths from a single source s to all other vertices.
  - **Floyd-Warshall Algorithm**: Computes shortest paths between all pairs of vertices, useful for dense graphs.
  - **A* Search Algorithm**: Utilises heuristics to dramatically speed up the search for a shortest path, useful in practical applications like GPS navigation.

67.3 Applications of s-t Connectivity Problems

- **Solving a Maze**: This can be seen as an s-t connectivity problem where s is the start of the maze and t is the exit. The solution involves finding a path through the maze from start to finish.
- **Finding a Route**: Similar to solving a maze but typically involves navigating a network, such as road maps for vehicle routing or links in a network topology.
- **Calculating Kevin Bacon Numbers**: This is an application of the s-t shortest path problem in a social network where actors are vertices and edges are films they have appeared in together. The Kevin Bacon number of an actor is the shortest path length to Kevin Bacon in this network.

68 Breadth-First Search (BFS)

68.1 BFS Algorithm Overview

The BFS algorithm starts from the source node s, exploring all its immediate neighbours, then for each of those neighbours, it explores their unvisited neighbours, and so on. It proceeds level by level, hence exploring the graph in "waves."

BFS Key Properties

- Explores graph **level by level** (by distance from source)
- Uses a **queue** (FIFO) data structure
- Finds **shortest paths** in unweighted graphs
- Time complexity: $O(n + m)$
- Space complexity: $O(n)$ for the visited array and queue

68.1.1 BFS Algorithm Steps

1. **Initialisation**: Start with a queue that contains only the source node s and mark s as visited.
2. **Exploration**:
   - Dequeue an element from the front of the queue (say v).
   - Visit all unvisited neighbours of v, mark them as visited, and enqueue them.
   - Repeat until the queue is empty.

68.1.2 Level Sets in BFS

- **Level Set** L_0: Contains only source node s.
- **Level Set** L_1: Contains all neighbours of L_0.
- **Level Set** L_2: Contains all nodes that are two edges away from s. These nodes are the neighbours of nodes in L_1 that have not been visited before or included in previous levels.
  - Formally: all nodes that do not belong to $L_0 \cup L_1$, and have an edge to a node in L_1
- **Level Set** L_{i+1}: All nodes that do not belong to $\bigcup_{j \leq i} L_j$, and have an edge to a node in L_i

Each subsequent level set L_i contains all nodes that are exactly i edges away from s.

BFS Correctness

Theorem: There is a path from s to t if and only if t appears in some level set L_i.

Proof idea: BFS systematically explores all reachable nodes by expanding outwards from s. Every node connected to s will eventually be discovered. Conceptualise this as a contagion spreading through the network - every connected node will be reached.

68.2 Properties and Benefits of BFS

- **Path Existence (Correctness)**: There is a path from s to t if and only if t appears in one of the level sets during the BFS execution.
- **Shortest Path Guarantee**: BFS guarantees that the first time a node is visited, the path to that node from s is the shortest possible path in terms of the number of edges (this only applies to unweighted graphs).
- **Completeness**: BFS is complete, meaning that if there is a solution or if the node t is reachable from s, BFS will definitely find it.
- **Time Complexity**: The time complexity of BFS is $O(n + m)$, where n is the number of vertices and m is the number of edges in the graph. This is because each vertex and each edge will be explored in the worst-case scenario.

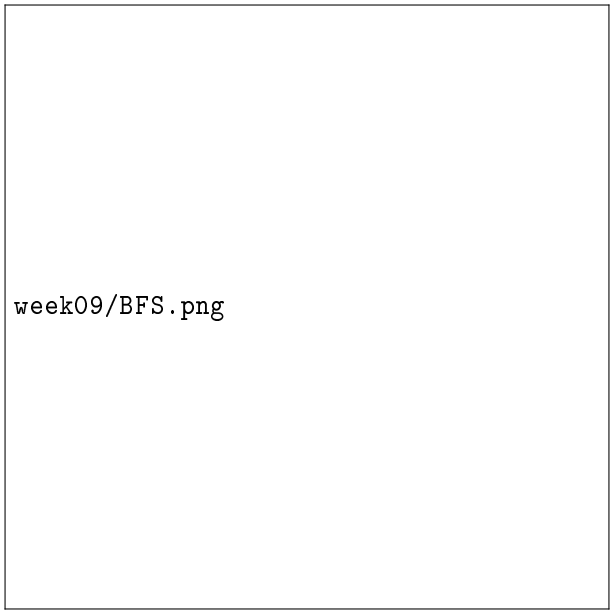

Figure 126: BFS exploration pattern - nodes are visited in order of their distance from the source.

68.3 Analysis of BFS

BFS Tree Property

Let T be a BFS tree of $G = (V, E)$, and let x-y be an edge of G. Then the **levels of x and y differ by at most 1**.

This property ensures:

- The BFS tree is correctly structured according to shortest path distances
- Each edge connects vertices at the same level or adjacent levels
- The path from s to any vertex in the BFS tree is a shortest path

Implications of the Property

- **Ensures BFS Tree Structure**: This property ensures that the BFS tree is correctly structured according to the shortest path distances from the source vertex. Each edge in the original graph connects vertices that are either at the same level or at adjacent levels.
- **Shortest Path**: Because each vertex is visited at its earliest possible level, the path from the source vertex to any other vertex in the BFS tree is the shortest path in terms of the number of edges.
- **Graph Analysis**: This characteristic of BFS is particularly useful in applications such as finding the shortest path in unweighted graphs, analysing the structure of networks, and more.

Figure 127: BFS tree structure: the solid black lines show the BFS-discovered shortest paths between s and other vertices; the dotted lines show other edges in the graph that connect vertices within or between adjacent levels.

68.3.1 Time Complexity of BFS

BFS runs in $O(m+n)$ time, if the graph is given by its adjacency list representation, where:

- m is the number of edges
- n is the number of vertices

This complexity is derived from the way BFS explores each vertex and edge of the graph exactly once in the case of an adjacency list representation.

Adjacency List Representation:

- **Vertex Exploration**: Each vertex is enqueued and dequeued exactly once. The enqueuing and dequeuing operations take constant time, i.e., $O(1)$, and since there are n vertices, this part of the algorithm takes $O(n)$ time.
- **Edge Exploration**: Each edge in the graph is considered exactly once during the entire run of BFS. When a vertex is dequeued, BFS examines each adjacent vertex connected by an edge. Since each edge is examined exactly once and there are m edges, this part of the algorithm takes $O(m)$ time.

Detailed Breakdown

1. **Initialisation**: BFS initialises a queue and a boolean array (or similar structure) to keep track of visited vertices, which takes $O(n)$ time.
2. **Processing Vertices**: As BFS processes each vertex exactly once, and each vertex enqueue/dequeue operation takes $O(1)$, the total time taken for all vertices is $O(n)$.
3. **Processing Edges**: Since the adjacency list of each vertex contains all its adjacent vertices (edges), and each edge in the graph is examined exactly once throughout the BFS run, the total time taken for all edges is $O(m)$.

Why $O(n+m)$?

1. Consider node i: there are degree(i) connected edges
2. Total time processing edges is $\sum_{i \in V} \text{degree}(i) = 2m$
   - **Each edge i-j is counted twice**: once in degree(i) and once in degree(j)
3. Even when $m = 1$, we still have to initialise and check each vertex, giving $O(n)$

Why Each Edge is Counted Twice

In an undirected graph represented by an adjacency list, each edge is stored twice: once in the list of each of the two vertices it connects.

For example, if there is an edge between vertices i and j, this edge will appear in the adjacency list of i and in the adjacency list of j. Therefore, summing degrees gives $2m$ because each edge contributes 2 to the total degree count.

In **sparse graphs**, where the number of edges m is much less than the possible maximum $\frac{n(n-1)}{2}$ for undirected graphs, the n term can dominate because you must account for potentially disconnected vertices or those with fewer connections.
In **dense graphs**, as m approaches $\frac{n(n-1)}{2}$, the number of edges m significantly influences the time complexity, reflecting the extensive connectivity among the vertices.

Practical Implications

- **Efficiency**: BFS is particularly efficient on **sparse graphs** where m is much less than n^2 (the number of edges in a complete graph). **In such cases, $O(m+n)$ (from using BFS on an adjacency list) is a better bound than $O(n^2)$, which you might get using an adjacency matrix**.
- **Applications**: BFS is used in scenarios requiring the exploration of all vertices reachable from a certain vertex, finding shortest paths in unweighted graphs, and in various algorithms requiring level order traversal (like in networking, pathfinding in games, and more).

68.4 Example Applications of BFS

68.4.1 Flood Fill

Given an image, colour a contiguous region of one colour into another colour.

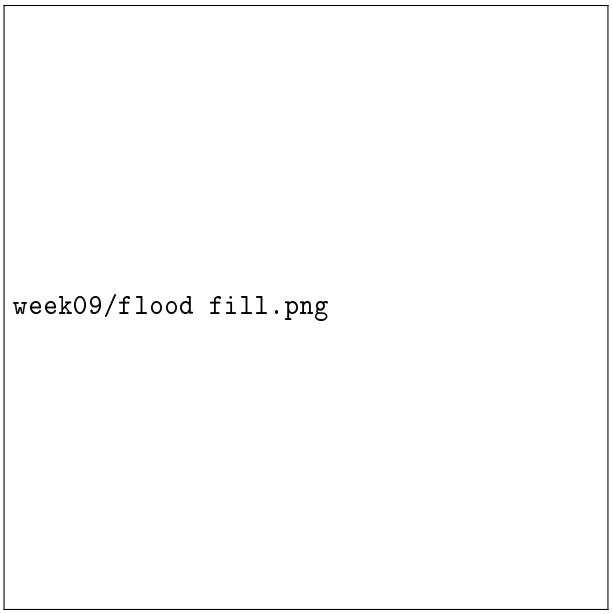

Figure 128: Flood fill algorithm - colouring a contiguous region. Each pixel is a node; edges connect adjacent pixels of the same colour.

- **Node**: pixel
- **Edge**: exists if two adjacent pixels are the same colour
- **Blob**: connected component of pixels of the same colour

68.4.2 K-Nearest Neighbours Classification

Identifying some region where all the labels are the same for any test point in that region. It would have the same classification, so you can think of this as a flood fill over the decision regions.

68.4.3 Application in Experimental Design

1. **Systematic Sampling**: Starting from a randomly chosen unit, BFS can be used to generate a tree where each level represents a set of units that are equidistant from the starting point. This method ensures that all units within the same level are similarly distant from the starting conditions, potentially controlling for confounding variables related to proximity or similarity.
2. **Uniform Distribution**: By treating each level set L_i with the same experimental condition (either treatment or control), researchers can ensure that the assignment is not biased by the network structure of the data. This is particularly useful in cases where the connectivity of units might influence the outcome (e.g., in social networks or spatially distributed units).
3. **Coverage and Separation**: BFS ensures that treatment is distributed broadly across the network, covering diverse connections and reducing the risk of clustered treatment effects. This spread helps in evaluating the treatment's impact across a variety of contexts and connections between units.
4. **Graph-Based Data Analysis**: In settings where the units are connected in a graph-like structure (e.g., social networks, geographic locations with adjacency, etc.), using BFS helps to understand how treatments might propagate through the network, which is useful for understanding diffusion of effects, peer influence, and network externalities.

week09/bfs app.png

Figure 129: BFS for experimental design: start at randomly allocated s; then each level set is coloured in reference to that point - this gives good coverage of treatment across the network structure.

68.4.4 Explore All Connected Components

To find all nodes connected to s:

- **BFS**: explores by distance from source
- **DFS**: explores as far as possible along each branch before backtracking

Both methods will find all connected nodes; the difference is in the order of exploration.

69 Depth-First Search (DFS)

Keep getting distracted by the first undiscovered thing you find.

Both BFS and DFS explore the whole graph; the difference is in the order in which they do so. Neither is inherently more efficient - it depends on what you want to achieve. In general, BFS is more useful when shortest paths matter.

The key data structure for DFS is the **stack**:

- As you traverse, you add other connected vertices to the stack
- If you reach a vertex with no unvisited adjacent vertices, you backtrack by popping the stack (or returning from a recursive call) to explore the next vertex

Whereas BFS leverages the queue.

69.1 DFS Outline

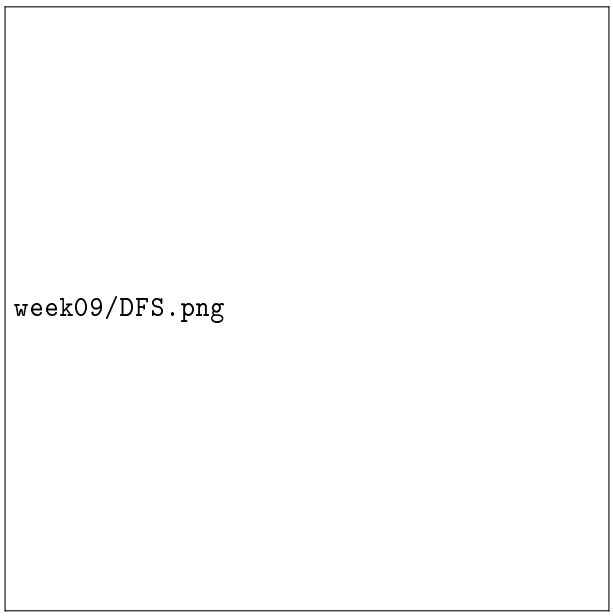

Figure 130: DFS exploration pattern - the algorithm goes as deep as possible before backtracking.

69.1.1 Recursive Definition

Figure 131: Recursive DFS - the call stack implicitly manages the traversal order.

```
function DFS(vertex v)
    mark v as visited
    visit v

    for each vertex u adjacent to v
        if u is not visited
            DFS(u)
```

69.1.2 Non-Recursive Definition

Figure 132: Non-recursive (iterative) DFS using an explicit stack.

```
function DFS(start_vertex)
    create a stack
    push start_vertex onto the stack
    mark start_vertex as visited

    while stack is not empty
        v = pop the stack
        visit v

        for each vertex u adjacent to v
            if u is not visited
                mark u as visited
                push u onto the stack
```

The algorithm starts at a selected node (the root in the case of trees, or any node in the case of graphs) and explores as far as possible along each branch before backtracking:

- Visit an adjacent, unvisited vertex, mark it as visited, and push it on a stack (or move into it recursively).
- Continue moving to an adjacent unvisited vertex, marking new vertices and pushing them onto the stack as you go.
- If you reach a vertex with no unvisited adjacent vertices, you backtrack by popping the stack (or returning from a recursive call) to explore the next vertex.

69.2 Properties of DFS

DFS Key Properties

- Explores graph by going **as deep as possible** before backtracking
- Uses a **stack** (LIFO) - either explicit or via recursion
- Time complexity: $O(n + m)$ (same as BFS)
- Space complexity: $O(n)$ in the worst case (path length)
- More memory-efficient than BFS for sparse, deep graphs

- **Memory Efficiency**: DFS uses less memory than BFS in sparse graphs because it holds a single path from the root node rather than storing all the nodes at a given depth level.
- **Path Finding**: DFS is often used when we need to find a path in the graph, especially if the path is likely to be deep.
- **Complexity (same as BFS)**: $O(n + m)$, where n is the number of vertices and m is the number of edges in the graph. This is because each vertex and each edge is explored exactly once.

69.3 Applications of DFS

- **Topological Sorting**: DFS is useful for ordering vertices such that every directed edge from vertex u to vertex v implies that u comes before v in the ordering.
- **Cycle Detection**: DFS can be used to detect cycles in a graph, which is important in many applications, including detecting deadlocks in concurrent systems.
- **Component Finding**: DFS helps in identifying the connected components in a graph.
- **Solving Puzzles**: Such as mazes or logical problems where exploring to the deepest level might yield a solution.

70 BFS vs DFS

DFS is particularly useful in applications such as topological sorting, detecting cycles, and finding strongly connected components. It is also used to simulate scenarios where complete exploration of one path or option is needed before moving to another (e.g., solving mazes, puzzle games).

When to Use BFS vs DFS

Use BFS when:

- Task requires exploring the graph **level by level**
- **Shortest path** or minimum moves are of interest
- Solution is likely **close to the starting point**

Use DFS when:

- **Complete exploration** of a path is necessary before alternatives (backtracking)
- **Memory is constrained** (DFS can be more space-efficient)
- Graph is **deep** and solutions are potentially **far from root**
- Need to detect cycles or perform topological sorting

70.1 Intuitive Differences

Here we consider the recursive form of DFS:

- **BFS**:
  - BFS explores all the nodes at the present "depth" (distance from the start node) before moving on to nodes at the next depth level.
  - It expands outward from the starting point and is implemented using a **queue (First In, First Out - FIFO)**. This ensures that nodes are explored in the order they are discovered.
  - Think of it as going in waves at each level set: filling up the queue → emptying the queue → filling up the queue → emptying the queue
- **DFS**:
  - In contrast, DFS dives as deep as possible into the graph before backtracking to explore other paths.
  - It is typically implemented using a **stack (Last In, First Out - LIFO)**, or through **recursion, which inherently uses a call stack**.
  - This means DFS will follow a path from the starting node to an end node, then backtrack and explore other paths from previously visited nodes.
  - It adds all the adjacent nodes to a stack but doesn't explore them until it has reached the end of its current path.

70.2 Key Differences in Operation

1. **Node Exploration Order**:
   - **BFS**: Nodes are explored in layers based on their distance from the start node. It uses a queue to keep track of the next node to explore. BFS adds all adjacent nodes to the queue and explores the oldest node first, ensuring that the distance from the start node increases gradually.

- **DFS**: Nodes are explored as deeply as possible before backtracking. Using a stack (or recursion), DFS adds one adjacent node and moves deeper immediately, exploring as far as possible along each branch before backtracking.

2. **Properties and Use Cases**:

   - **BFS** is particularly useful for finding the shortest path in unweighted graphs and for scenarios where you need to explore all options uniformly, like in the case of level-wise processing required in algorithms like the minimum spanning tree or for layer-by-layer computation in neural networks.
   - **DFS** is advantageous in scenarios involving exhaustive searches, as in puzzle games (e.g., solving mazes or the Knight's tour problem) where a solution involves exploring all possible configurations. DFS is also useful for topological sorting, and in scenarios where space complexity might be a constraint since iterative DFS can be more space-efficient than BFS.

3. **Performance**:

   - **Time complexity is the same:** $O(m+n)$, where m is the number of edges, and n is the number of vertices. However, the actual performance may depend on the graph's structure and the specific needs of the application, such as whether the problem requires visiting every node or just ensuring a particular condition is met (like finding a cycle).
   - **Space complexity differs: iterative DFS can be slightly more space-efficient than BFS**, especially in sparse graphs where backtracking happens frequently and the stack depth remains relatively small compared to the number of nodes that would have to be kept in the queue for BFS.

70.3 Practical Implications

The choice between BFS and DFS can significantly impact the performance and correctness of graph-based algorithms depending on the specific requirements of the application:

- **BFS** might be more suitable for networking applications or GPS navigation systems where finding the shortest route is essential.
- **DFS** could be more appropriate for applications involving the exploration of complex branchings, such as syntax tree exploration in compilers or backtracking algorithms in constraint satisfaction problems.

71 Bipartite Graphs

A bipartite graph is a special class of graph where the set of vertices can be divided into **two disjoint subsets** such that **no two vertices within the same subset are adjacent**.

This type of graph can be coloured using just two colours:

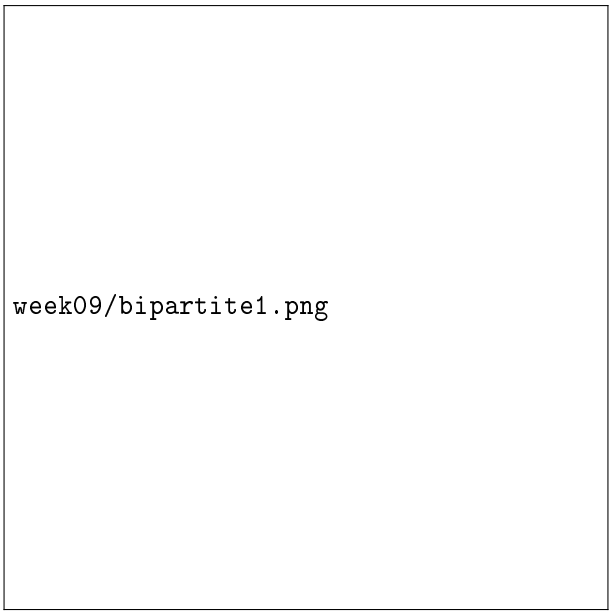

Figure 133: An uncoloured bipartite graph - can you see how to partition the vertices into two groups?

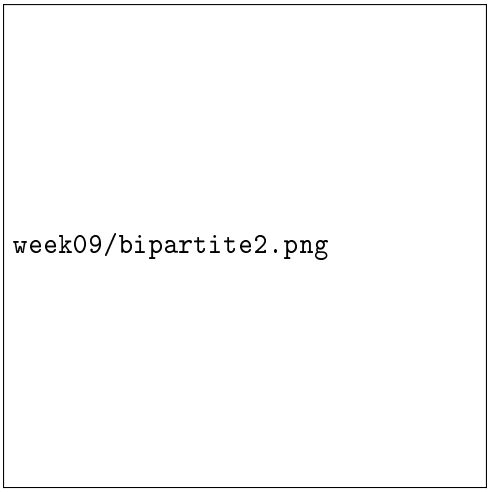

Figure 134: The same graph with a valid 2-colouring - vertices are partitioned into red and blue sets with no edges within either set.

71.1 Properties of Bipartite Graphs

Bipartite Graph Characterisation

A graph G is bipartite if and only if:

1. It can be **2-coloured** (no adjacent vertices share a colour)
2. It contains **no odd-length cycles**
3. Its vertices can be partitioned into two sets U and V where every edge connects a vertex in U to one in V

Important: Trees are always bipartite (they have no cycles at all).

- **Two-Colour Property**: A graph is bipartite if and only if it can be coloured using two colours such that no two adjacent vertices share the same colour.
  - This implies that the graph does **not contain any odd-length cycles**, which would require at least three colours to colour the graph properly.
- **Subsets**: If you can split the graph's vertices into two groups U and V such that **every edge connects a vertex in U to one in V** (and no edge connects vertices within the same group), the graph is bipartite.
- **Trees are always bipartite**: Because trees are acyclic, you can always alternate colours along any path from the root, ensuring that no two adjacent vertices share the same colour.

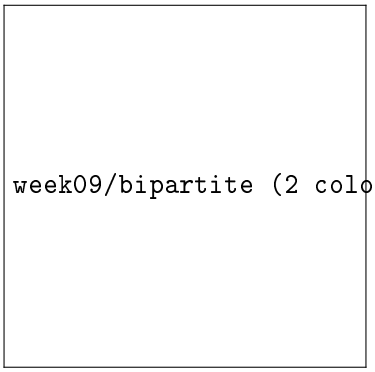

Figure 135: A bipartite graph - successfully 2-coloured.

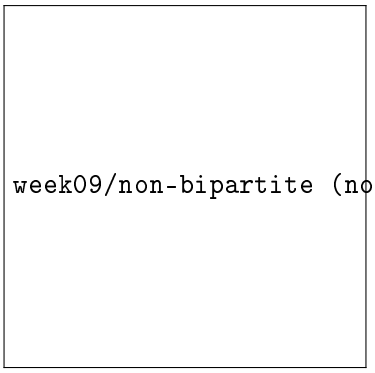

Figure 136: A non-bipartite graph - contains an odd-length cycle (triangle), making 2-colouring impossible.

71.2 Checking if a Graph is Bipartite

You can determine whether a graph is bipartite using graph traversal techniques like Depth-First Search (DFS) or Breadth-First Search (BFS):

1. **Start from any vertex** and colour it with one colour.
2. **Use BFS/DFS** to traverse the graph:
   - When visiting a vertex, colour it with the alternate colour of its parent.
   - *If you find a vertex that has already been coloured with the same colour as the current vertex, the graph is not bipartite.*
3. **Continue** until all vertices are coloured or a contradiction is found.

Bipartite Detection

A graph is **not bipartite** if and only if it contains an **odd-length cycle**.

During BFS/DFS traversal, if you ever find an edge connecting two vertices at the **same level** (same colour), the graph contains an odd cycle and is not bipartite.

71.3 Applications of Bipartite Graphs

1. **Stable Matching**:
   - In scenarios like the stable marriage problem or matching medical residents to hospitals, bipartite graphs are used to model preferences on both sides (e.g., applicants and hospitals), ensuring that matches are stable and no two individuals would prefer each other over their current matches.
2. **Scheduling**:
   - Bipartite graphs are ideal for scheduling problems where you need to match two distinct groups, such as machines and jobs, ensuring that no machine is overloaded or that prerequisites are respected.
3. **Causal Inference**:

- In experimental design and statistics, causal inference models can use bipartite graphs to differentiate between treatment and control groups, ensuring that the influence of treatments can be distinctly measured without interference.

4. **Network Flows**:
   - Many network flow problems, such as the maximum flow problem, can be represented as bipartite graphs, where the flow from a source to a sink through intermediary nodes (representing two distinct sets) needs to be maximised.
5. **Data Clustering**:
   - In data science, clustering algorithms can use bipartite graphs to categorise data into two groups based on different attributes or relationships, useful in recommendation systems or for segmenting markets.

71.4 BFS and Bipartiteness

BFS naturally layers a graph by levels of distance from a starting node - this structure can determine if a graph is bipartite by checking for the presence of odd-length cycles.

- **Initialisation**:
  - Start BFS from node i, marking it with one colour (say, red).
  - Initialise a queue and enqueue the starting node.
- **BFS Traversal**:
  - While the queue is not empty, dequeue a node u.
  - For each unvisited adjacent node v of u:
    - If v is not coloured, assign it the opposite colour of u and enqueue v.
    - If v is already coloured and shares the same colour as u, then the graph cannot be bipartite.
- **Layer Formation**:
  - As BFS proceeds, it forms layers $L_1, L_2, \ldots, L_k$ where each layer L_j contains all nodes that are j edges away from the starting node i.
  - No two nodes within the same layer should be connected by an edge if the graph is bipartite.

71.5 Analysing the Results

Let G be the graph, and let $L_0, L_1, \ldots, L_k$ be the layers produced by BFS starting at node i.

Bipartiteness via BFS

Theorem: A graph G is bipartite if and only if there are no intra-layer edges in any BFS traversal.

If no intra-layer edges exist $\Rightarrow$ G is bipartite:

- All edges connect nodes from adjacent layers only
- This alignment confirms the graph is bipartite because it effectively assigns two colours such that no two adjacent nodes share the same colour
- No odd-length cycles can exist

If an intra-layer edge exists $\Rightarrow$ G is not bipartite:

- An edge connecting nodes within the same layer indicates an odd-length cycle
- To connect within the same layer, a cycle must "dip" down to a lower layer and return, creating an odd cycle

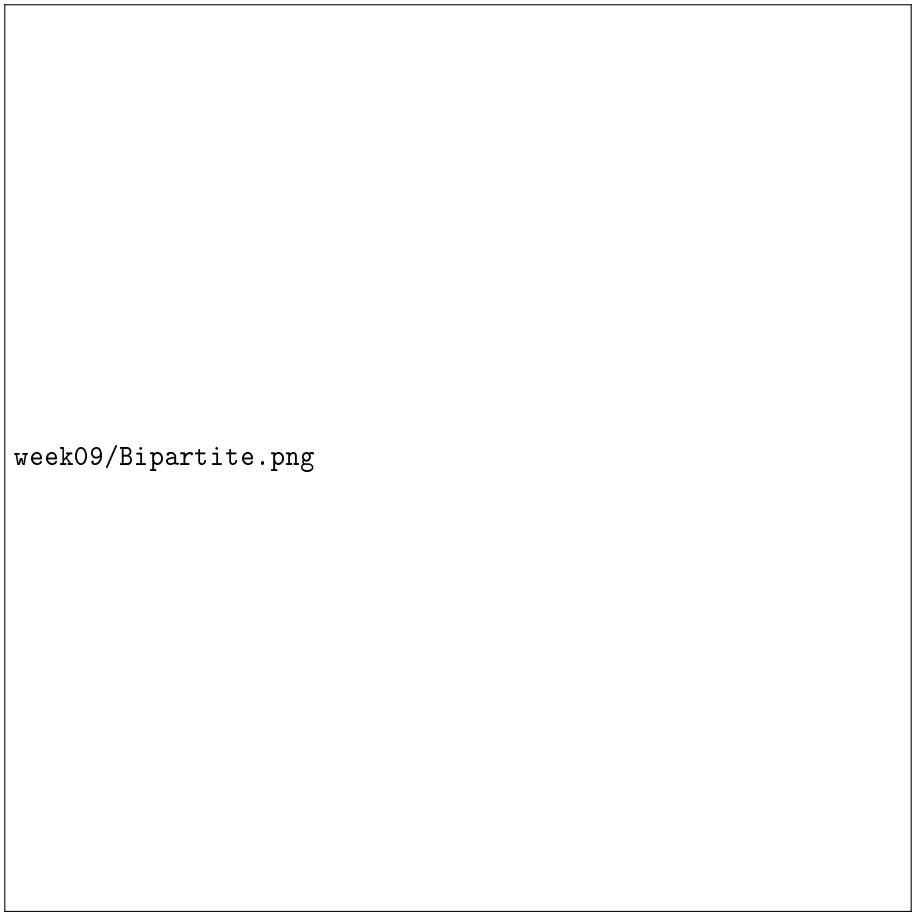

Figure 137: Left: Bipartite graph with no intra-layer edges. Right: Non-bipartite graph with an intra-layer edge (indicating an odd cycle).

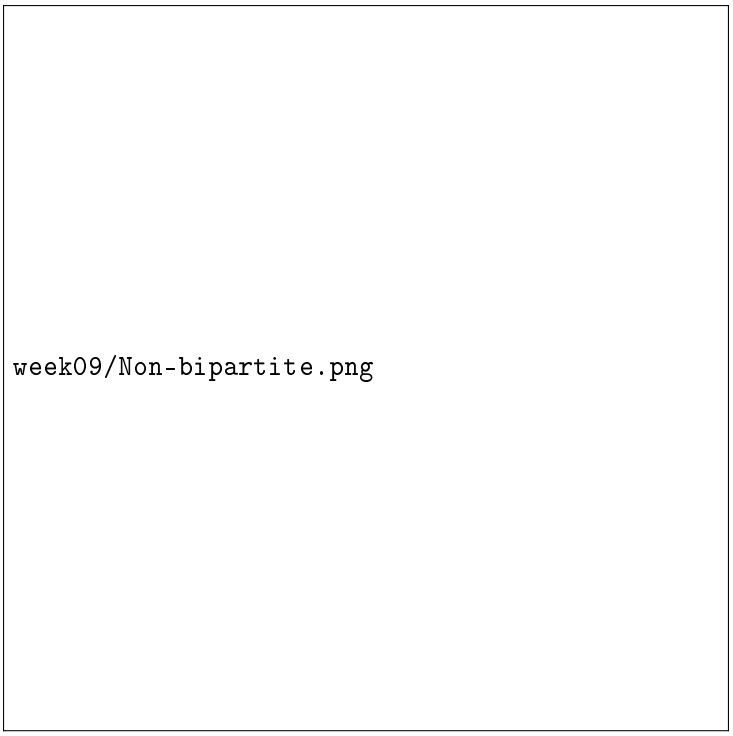

Figure 138: BFS layers of a non-bipartite graph: the red edge connects two vertices within the same layer L_3, revealing an odd-length cycle.

We are trying to colour each level a single colour; we can have connections between levels, but not within levels.

72 Directed Graphs

In a directed graph (or **digraph**), edges have a direction: $i \to j$ does **not** imply that $j \to i$.

72.1 Directed Acyclic Graphs (DAGs)

A **Directed Acyclic Graph (DAG)** is a directed graph with no directed cycles. DAGs are particularly important because they can encode causal relationships and dependency structures.

DAG Properties

A Directed Acyclic Graph (DAG) has:

- **No directed cycles**: There is no path $v_1 \to v_2 \to \cdots \to v_k \to v_1$
- **At least one source**: A vertex with no incoming edges
- **At least one sink**: A vertex with no outgoing edges
- **A topological ordering**: Vertices can be linearly ordered respecting edge directions

Acyclic: There is no directed path $i \to j \to k \to \cdots \to i$ that returns to its starting vertex.

72.2 Topological Ordering

A **topological ordering** (or topological sort) is a concept applied to DAGs to provide a linear ordering of its vertices such that for every directed edge $v_i \to v_j$, vertex v_i comes before v_j.

Topological order encodes *causal precedence* in a causal DAG.

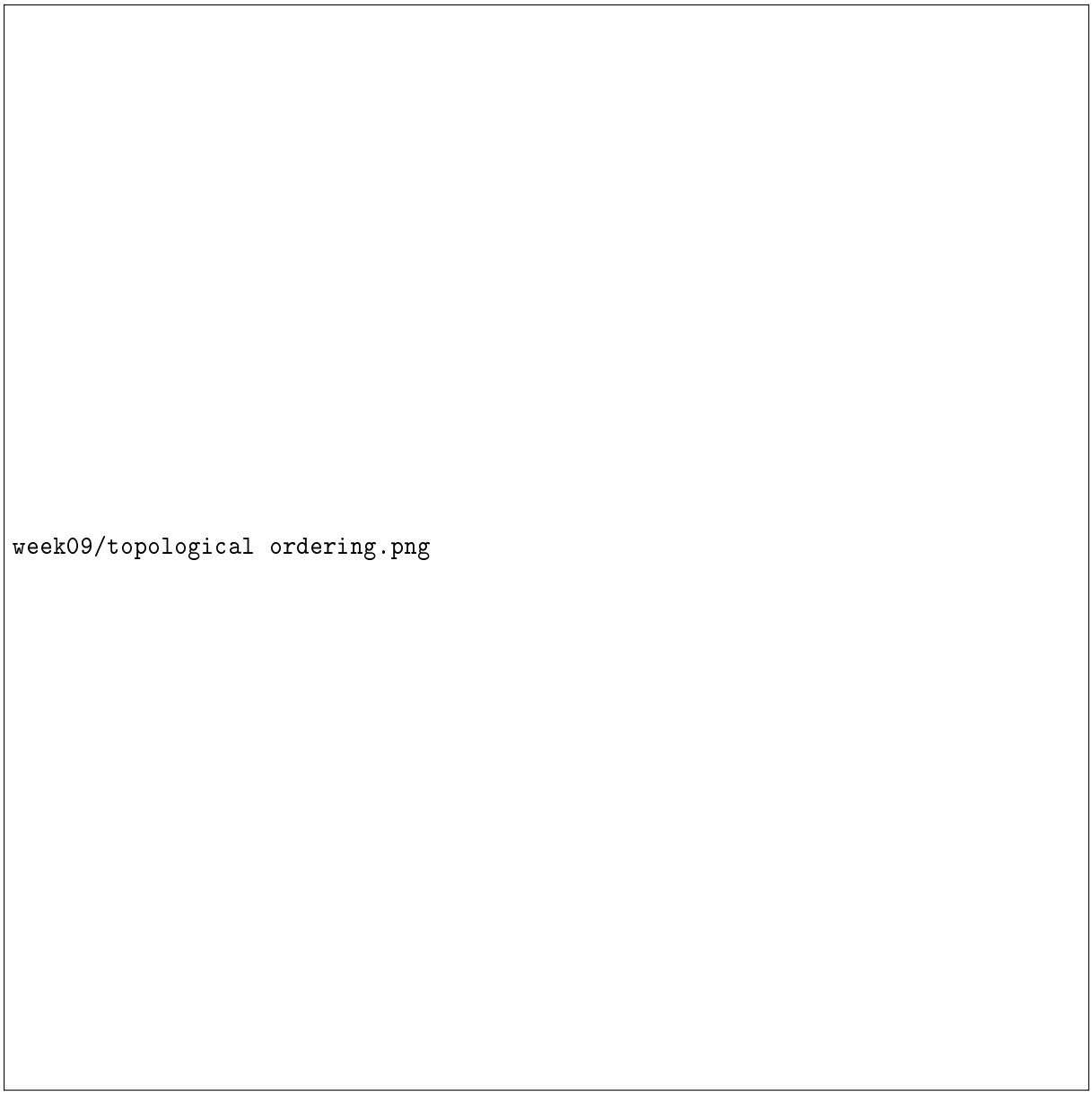

Figure 139: Left: A standard DAG representation. Right: The same DAG with vertices arranged in topological order - all edges point from left to right.

72.2.1 Properties of Topological Ordering

- **Directed Acyclic Graph (DAG)**: Topological ordering only applies to DAGs, as the presence of a cycle would make such an ordering impossible.
- **Uniqueness**: A topological order is not necessarily unique; a DAG might have several valid topological sorts.

- **Algorithms**: The common algorithms to find a topological order are Kahn's Algorithm and DFS-based methods.

72.2.2 Kahn's Algorithm for Topological Sorting

Kahn's algorithm builds the topological order by repeatedly choosing vertices with no incoming edges.

1. **Identify Vertices with No Incoming Edge**: These will be the starting points for the sorting.
2. **Remove Chosen Vertex and Its Edges**: Once a vertex with no incoming edges is selected, it is removed from the graph along with all its outgoing edges.
3. **Repeat Process**: Continue removing vertices until all have been ordered.

72.2.3 DFS-Based Algorithm for Topological Sorting

1. **Perform DFS**: Run DFS from every unvisited node.
2. **Track the Finish Times**: Record the finish times for each node.
3. **Order by Finish Times**: Once all nodes have been visited, sort them by decreasing finish time, which gives a topological order.

72.2.4 Applications of Topological Ordering

- **Task Scheduling**: In project scheduling or task management where tasks have dependencies, a topological sort provides an order in which to perform the tasks without conflicts.
- **Causal Inference**: In DAGs that represent causal relationships, a topological sort can give the order in which events must occur or be considered.
- **Package Dependency Resolution**: In package managers for software development, topological sorting can be used to decide the order in which packages or libraries should be installed to satisfy dependencies.

72.2.5 Topological Ordering Implies DAG

Topological Ordering ⇔ DAG

Theorem: A directed graph G has a topological ordering if and only if G is a DAG.

Proof ($\Rightarrow$): If G has a topological ordering, then G is acyclic.

Suppose for contradiction that G has both a topological ordering $v_1, v_2, \ldots, v_n$ and a cycle C.

1. Let $v_{c_1}, v_{c_2}, \ldots, v_{c_k}$ be the vertices in cycle C, indexed according to their positions in the topological ordering.
2. For each edge $v_{c_i} \to v_{c_{i+1}}$ in the cycle, the topological ordering requires $c_i < c_{i+1}$.
3. But since C is a cycle, there must be an edge from v_{c_k} back to v_{c_1}, requiring $c_k < c_1$.
4. This contradicts $c_1 < c_2 < \cdots < c_k$.

Therefore, if G has a topological ordering, it cannot have a cycle.

week09/contradiction.png

Figure 140: The backward edge (bottom) cannot exist in a valid topological ordering - it would create a contradiction.

DAGs have source and sink nodes

DAGs must have at least one vertex with no entering edges (the "source" vertex), and at least one vertex with no outgoing edges (the "sink" vertex). This property is useful for setting up topological orderings (see Kahn's algorithm).

- If a DAG had no source, every vertex would have an incoming edge, which would necessarily create a cycle.

72.2.6 Kahn's Algorithm (Detailed)

Initialisation: Start by scanning the graph to initialise the following:

1. `count[i]`: An array to maintain the count of incoming edges for each vertex i in the graph. This takes $O(m)$ time, where m is the number of edges, since you have to potentially look at every edge in the graph.
2. `S`: A set or list to keep track of all nodes with no incoming edges, which can be identified during the initial scan. Finding these nodes will take $O(n)$ time, where n is the number of vertices in the graph.

The overall time for this initialisation step is $O(m+n)$, since you are scanning each edge once and each vertex once.

Process of Creating Topological Order:

- **While `S` is not empty:**
  - Remove a node i from `S`.
  - Add i to the end of the topological ordering.
  - For each node j such that there is an edge $i \to j$:
    - Decrement `count[j]` by 1 (since we are effectively removing the edge $i \to j$ from the graph).
    - If `count[j]` becomes zero, add j to `S` because j now has no incoming edges.

Complexity Analysis:

Each removal operation is $O(1)$.

The removal of each edge is done exactly once, and each edge can only cause one decrement operation, so the **complexity of processing all edges is** $O(m)$.

Since each node is inserted into `S` once and removed from `S` once, the **complexity associated with processing all nodes is** $O(n)$.

Kahn's Algorithm Complexity

Overall complexity: $O(m+n)$

This reflects the initial graph scan ($O(n+m)$) and the subsequent edge removal and node processing steps.

The initialisation is the most expensive computational part - we must scan through the entire graph to set this up.

72.2.7 Causal Inference Application

Suppose you want to measure the causal effect of v_i on v_j. What nodes should you control for?

Figure 141: Causal DAG with topological ordering - when measuring the effect of v_i on v_j, be careful about which variables to control for.

Causal Inference Warning

You should **not** control for v_k such that $k > j$ in the topological ordering.

Controlling for a node that appears after j (i.e., a descendant of j) can introduce **collider bias** or open a back-door path, which can distort the causal estimate.

72.3 Finding d-Separation

In the context of causal inference, **d-separation** is a concept used to determine whether a set X of variables provides evidence about the independence between two other sets of variables, say A and Y. If A and Y are d-separated by X, then A is conditionally independent of Y given X, and in a regression model where Y is regressed on A and X, the coefficient of A would be expected to be zero if the only pathways between A and Y are blocked by X.

How to Find d-Separations:

The basic idea behind algorithms for finding d-separations in a graph involves the following steps:

1. **Identify All Paths Between A and Y**: In a directed graph (often a DAG in causal models), you would enumerate all paths between the nodes in sets A and Y.
2. **Analyse Conditional Independencies**: Determine whether each path is blocked or not when conditioning on set X. A path is considered blocked if it contains a non-collider that is in X or if it contains a collider that is not in X and has no descendants in X.
3. **Consider All Nodes Y**: Run the analysis for each variable that could be an outcome of interest in the graph to see if it is d-separated from A by X.

Complexity of Finding d-Separations:

The complexity of finding d-separations is often $O(m + n)$ where m is the number of edges and n is the number of nodes in the graph. This is because checking for d-separation between two

nodes in a DAG usually involves examining paths between the nodes, and in the worst case, you might need to traverse all the nodes and edges.

Algorithmic Approaches:

Specific algorithms, like the one mentioned by Neapolitan (2003) and the work of Geiger, Verma, and Pearl (1989), provide structured methods for efficiently computing these d-separations. These works typically provide pseudocode or step-by-step procedures for traversing the DAG and checking for conditional independencies.

In practical terms, these methods can be applied in statistical software using algorithms that traverse causal networks to aid in the design of statistical models and in the interpretation of regression coefficients, particularly in fields such as epidemiology, economics, and social sciences where causal inference plays a critical role in understanding relationships between variables.

73 Min-Cut and Max-Flow

73.1 Flow Networks

Flow networks are a specialised type of directed graph that facilitate modelling and solving various types of network flow problems.

Think of the network as a system of pipes - we want to know how water flows through each pipe. The capacity of each pipe corresponds to the edge weight.

Flow Network - Formal Definition

A **flow network** is defined by a tuple $G = (V, E, s, t, c)$ where:

- V is the set of vertices
- E is the set of directed edges
- $s \in V$ is the **source** vertex (where flow originates)
- $t \in V$ is the **sink** vertex (where flow terminates)
- $c : E \to \mathbb{R}_{\geq 0}$ is the **capacity function** assigning a non-negative capacity to each edge

For each edge $e \in E$: $c(e) \geq 0$

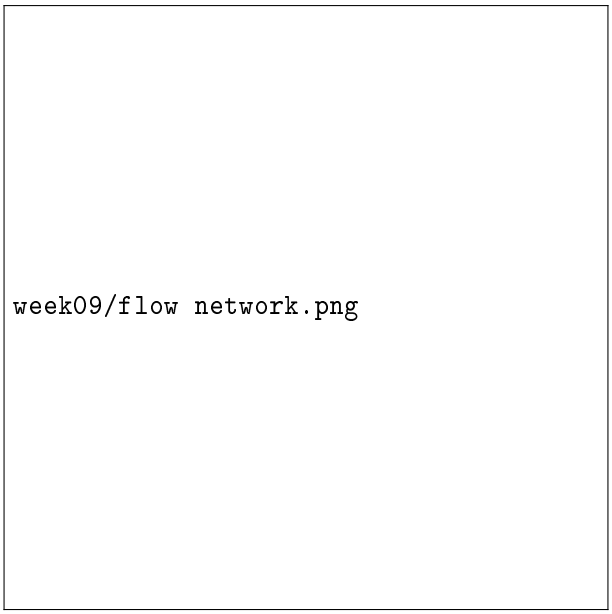

Figure 142: A flow network with source s, sink t, and edge capacities shown as weights.

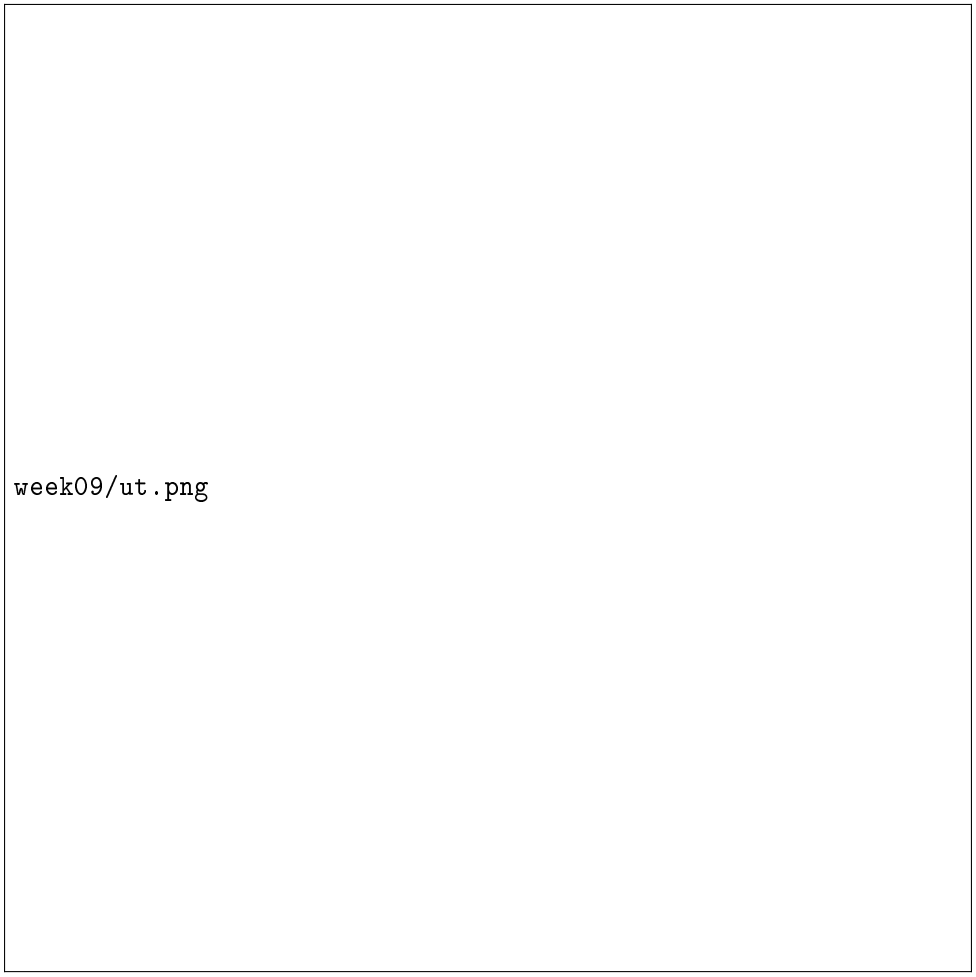

Figure 143: A larger flow network example: each edge label indicates capacity. The goal is to find the maximum flow from s to t.

Properties and Usage:

- **Edge Direction**: Edges in flow networks are directed, meaning that flow can only move in the specified direction from one vertex to another.
- **Conservation of Flow**: Except for the source, which "produces" flow, and the sink, which "consumes" flow, each vertex in a flow network has the property that the total flow into the vertex equals the total flow out of the vertex. This is known as the conservation of flow or Kirchhoff's first law in the context of electrical networks.

73.2 Graph Cuts

s-t Cut - Formal Definition

An **s-t cut** in a flow network is a partition of the vertices into two disjoint subsets A and B such that:

- $A \cup B = V$ and $A \cap B = \emptyset$
- $s \in A$ (source is in A)
- $t \in B$ (sink is in B)

The **capacity of a cut** (A, B) is:

$$\text{cap}(A, B) = \sum_{e \text{ from } A \text{ to } B} c(e)$$

where the sum is over all edges that cross from A to B.

Note: Edges from B to A do **not** contribute to the cut capacity.

The cut is the cumulative capacity of all the "pipes" at the point at which we cut.

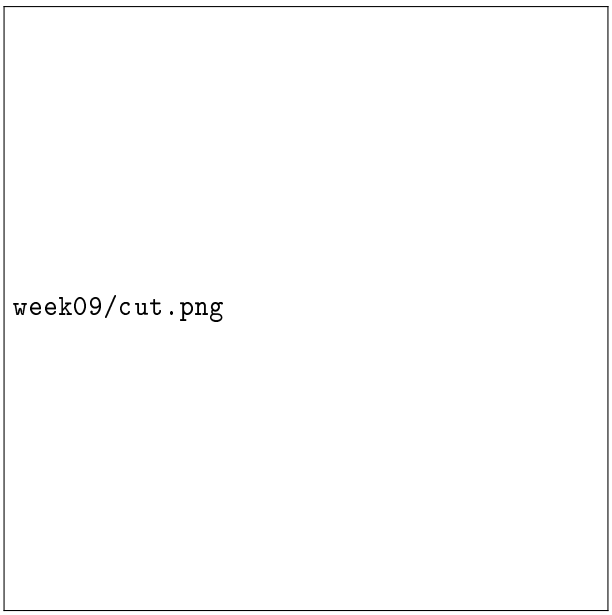

Figure 144: An s-t cut: the cut capacity includes only edges crossing from A to B, not edges from B to A.

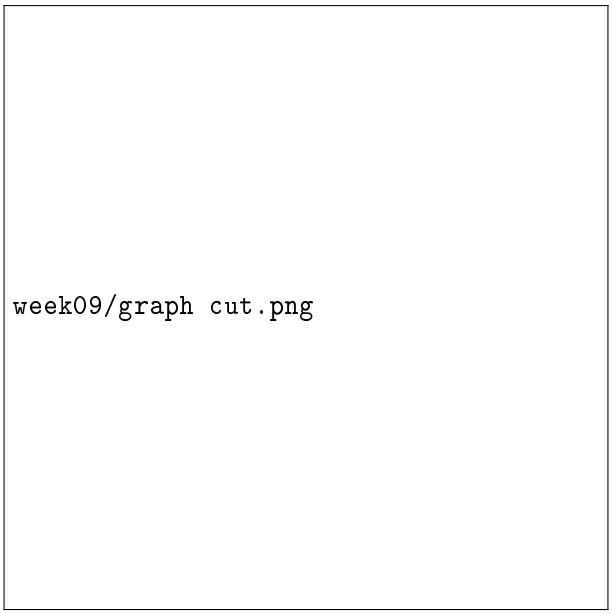

Figure 145: Another view of graph cuts - the dashed line shows the partition between sets A and B.

73.3 Min-Cut Problem

The **minimum cut** (min-cut) is the s-t cut with the smallest capacity among all possible cuts.

Min-Cut Problem

Goal: Find the s-t cut (A, B) that minimises $\text{cap}(A, B)$.

Key insight: To determine the min-cut, you typically compute the max-flow first. This relationship is grounded in the **Max-Flow Min-Cut Theorem**.

After computing maximum flow, you can find the minimum cut in $O(m)$ time.

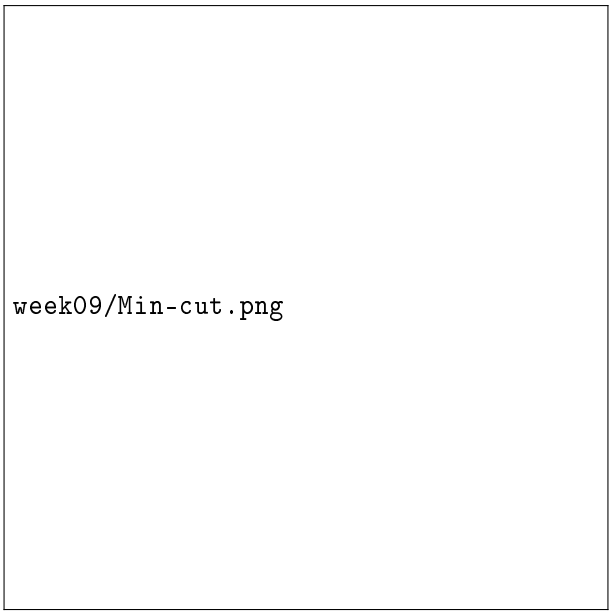

Figure 146: The minimum cut represents the bottleneck capacity of the network - the maximum flow that can pass from s to t.

Applications:

- **Network Design**: Understanding the limitations and bottleneck of a network.
- **Reliability Analysis**: Evaluating the robustness of a system against failures.
- **Image Segmentation**: In computer vision, graph cuts are used to separate objects from the background.
- **Project Management**: Identifying critical paths and constraints in project planning.

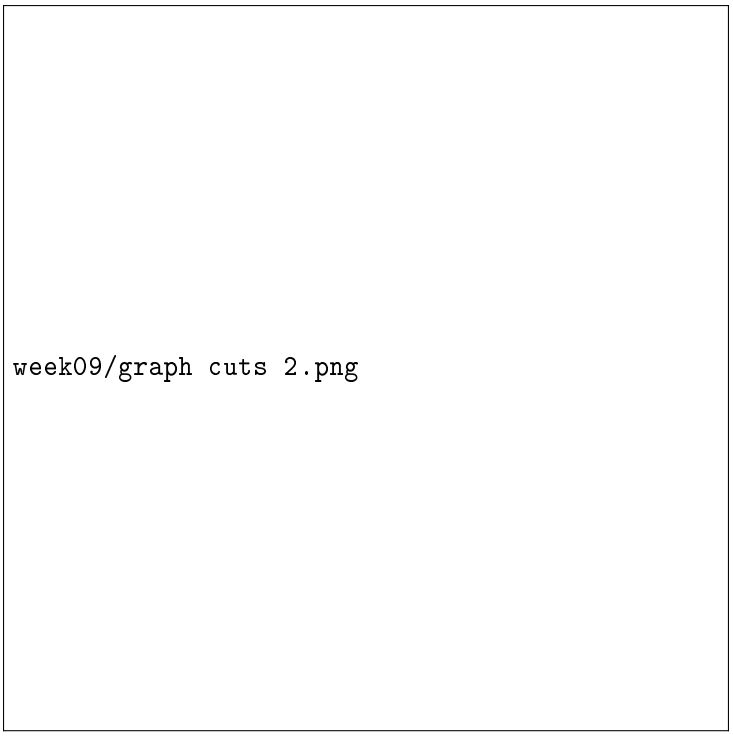

Figure 147: Graph cuts in image segmentation - pixels are nodes, and edges connect adjacent pixels with weights based on similarity.

73.4 Max-Flow Problem

The **maximum flow problem** asks for the maximum possible flow that can be sent from a source node s to a sink node t in a flow network, subject to capacity constraints on the edges and flow conservation at each node.

s-t Flow - Formal Definition

An **s-t flow** f is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies:

1. **Capacity Constraint**: For each edge $e \in E$:

$$0 \leq f(e) \leq c(e)$$

2. **Flow Conservation**: For every vertex $v \in V \setminus \{s, t\}$:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

(flow in equals flow out)

The **value** of the flow is:

$$|f| = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ into } s} f(e)$$

(net flow leaving the source)

Max-Flow Intuition

"How high can I turn up the water pressure at the source that this pipe network can support?"

The maximum flow equals the total amount of "stuff" that can be pushed from s to t per unit time, respecting all capacity constraints.

Note on flow value: The flow value $|f|$ represents the net flow leaving the source (which equals the net flow entering the sink, by conservation). Typically, there is no flow entering the source, so $|f| = \sum_{e \text{ out of } s} f(e)$.

Algorithms to Solve Max-Flow:

- **Ford-Fulkerson Method**: Uses augmenting paths and works well in practice. However, its time complexity is not polynomial in the size of the graph for irrational capacities.
- **Edmonds-Karp Algorithm**: An implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths, which leads to a polynomial runtime.
- **Dinic's Algorithm**: Relies on repeatedly constructing level graphs and finding blocking flows, with better average performance.

Applications of Max-Flow:

- **Networking**: Finding the maximum throughput in a network - e.g., internet routing
- **Transportation**: Determining the most efficient way to route goods through a transportation network.
- **Project Management**: Identifying critical paths in project scheduling.
- **Matching Problems**: Solving problems in bipartite graphs for matching resources to tasks.

73.5 Ford-Fulkerson Algorithm

73.5.1 Naive Algorithm (Fails)

Why the Naive Approach Fails

The naive greedy approach fails because once you choose a path and push flow through it, you can never decrease the flow through that path.

You need the ability to "undo" or redirect flow to fix suboptimal early choices.

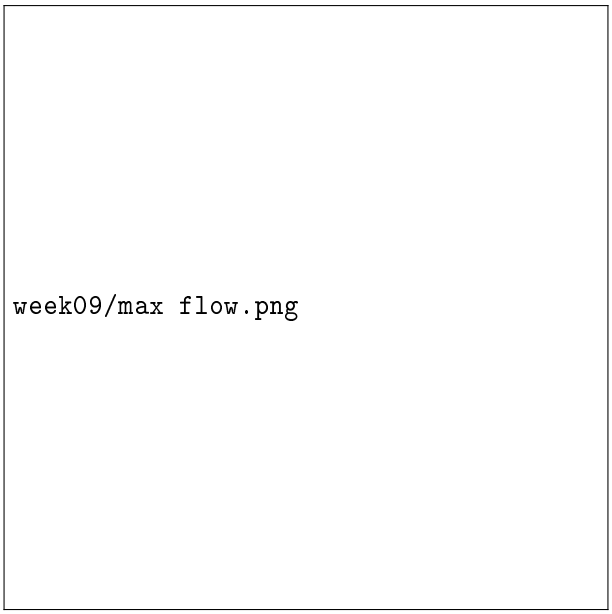

Figure 148: A flow network where the naive greedy approach can fail to find the maximum flow.

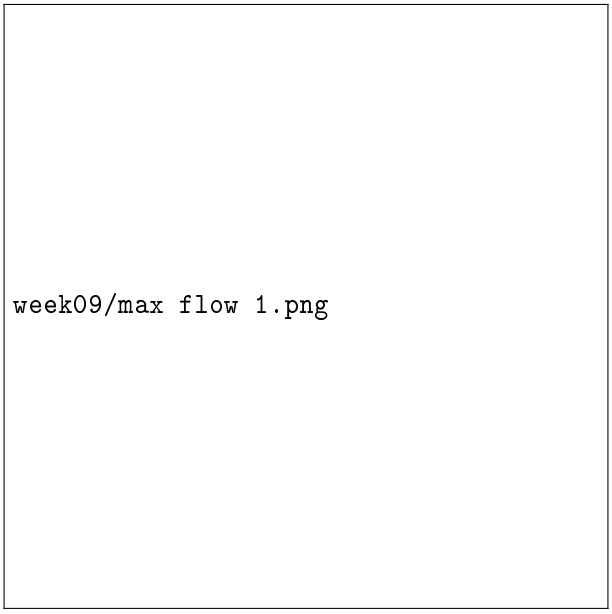

Figure 149: Step 1: Start with $f(e) = 0$ for each edge $e \in E$.

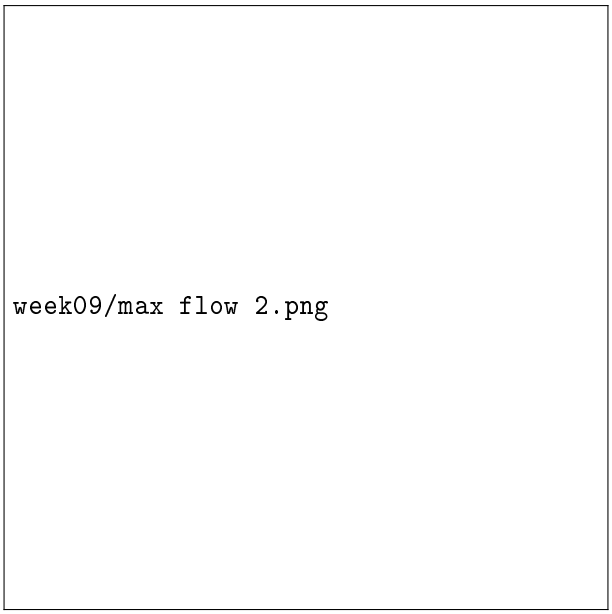

Figure 150: Step 2: Find an $s \to t$ path where each edge has $f(e) < c(e)$, then augment flow along that path.

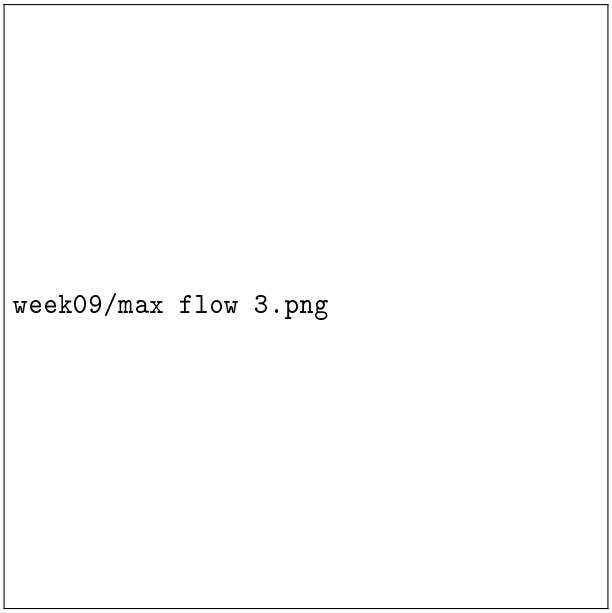

Figure 151: Step 3: Augment flow along the chosen path.

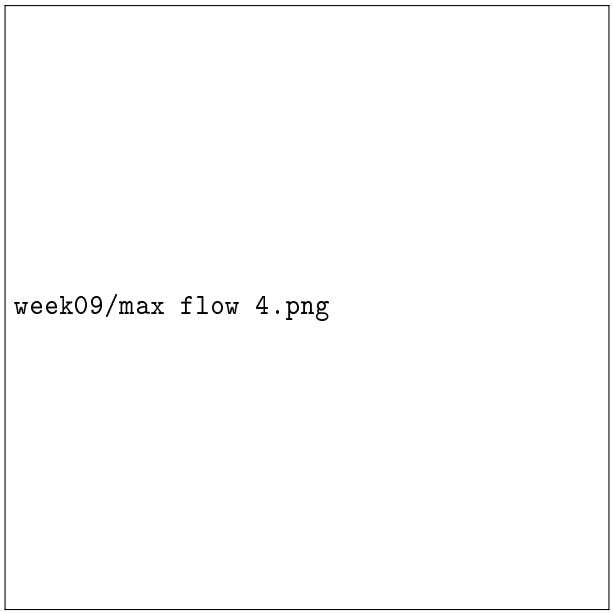

Figure 152: Step 4: Repeat until you get stuck (no more augmenting paths with available capacity).

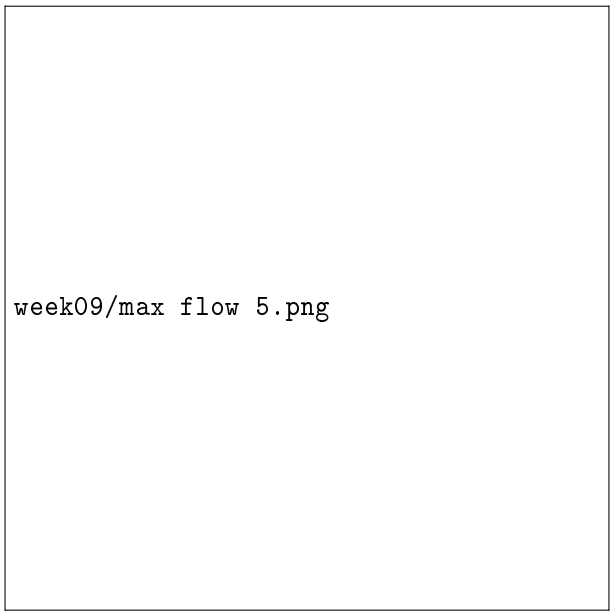

Figure 153: Naive result: ending flow = 16.

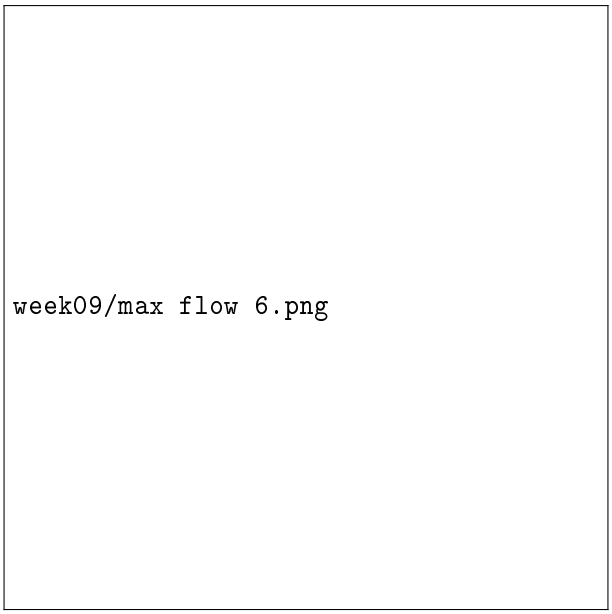

Figure 154: Optimal result: maximum flow = 19. The naive approach missed 3 units of flow!

The naive approach fails because once you choose a path, you can never decrease the flow through that path. We need the ability to go back and "fix" suboptimal choices.

73.5.2 Residual Flow / Residual Network

The **residual network** shows how much more flow can be pushed through the network and the directions in which it can be pushed. This network adjusts dynamically as the flow changes.

Residual Network - Formal Definition

Given a flow network $G = (V, E, s, t, c)$ and a flow f, the **residual network** $G_f = (V, E_f)$ is defined as follows:

For each edge $e = (u \to v)$ in G:

Forward edge (remaining capacity):

$$c_f(u \to v) = c(e) - f(e)$$

Include this edge in E_f if $c_f(u \to v) > 0$.

Backward edge (ability to "undo" flow):

$$c_f(v \to u) = f(e)$$

Include this edge in E_f if $f(e) > 0$.

The residual capacity formula:

$$c_f(e') = \begin{cases} c(e) - f(e) & \text{if } e' \text{ is a forward edge of } e \in E \\ f(e) & \text{if } e' \text{ is a backward edge of } e \in E \end{cases}$$

1. **Original Edge and Flow**:

- Let $e = (i \to j)$ be a directed edge in the original flow network G with a capacity $c(e)$ and a flow $f(e)$.

Figure 155: Original edge with capacity $c(e)$ and current flow $f(e)$.

2. **Residual Capacity**:

- The **residual capacity** of an edge e is defined as $c_f(e) = c(e) - f(e)$. This value represents the additional amount of flow that can still be pushed through edge e without exceeding its capacity.

Figure 156: Residual network showing both forward capacity ($c(e) - f(e)$) and backward capacity ($f(e)$).

3. **Reverse Edge**:

- For each edge $e = (i \to j)$ in G, the residual network includes a reverse edge $e_{\text{rev}} = (j \to i)$. This reverse edge allows for the possibility of reducing the flow that was previously sent through e, effectively "undoing" some of the flow if it enhances the overall solution.
- The capacity of the reverse edge e_{rev} is set to $f(e)$, which is the current flow through edge e.

4. **Key Property**:
   - A flow f' in the residual network G_f can be combined with the flow f in G to form a new valid flow in G. If f' represents an increase or decrease along certain paths, $f + f'$ reflects the adjusted flow in G that accommodates these changes.

73.5.3 Augmenting Path

An **augmenting path** is the key concept for finding maximum flow.

Augmenting Path

An **augmenting path** P is a simple path from s to t in the residual network G_f.

The **bottleneck capacity** of P is the minimum residual capacity among all edges in P:

$$\text{bottleneck}(P) = \min_{e \in P} c_f(e)$$

Augmentation: When an augmenting path is found, the flow can be increased by the bottleneck capacity:

- For forward edges $e \in P$: $f'(e) = f(e) + \text{bottleneck}(P)$
- For backward edges in P (corresponding to original edge e): $f'(e) = f(e) - \text{bottleneck}(P)$

Figure 157: An augmenting path in the residual network - the bottleneck determines how much flow can be pushed.

The new flow f' after applying the augmentation along path P increases the total flow from s to t by the amount of the bottleneck capacity:

$$|f'| = |f| + \text{bottleneck}(P)$$

This increment continues until no more augmenting paths can be found in the residual network, at which point the flow f is maximum.

73.5.4 Ford-Fulkerson Algorithm

Ford-Fulkerson Algorithm

1. **Initialise**: Set $f(e) = 0$ for each edge $e \in E$
2. **While** there exists an augmenting path P from s to t in residual graph G_f:
   - Compute bottleneck capacity of P
   - Augment flow along P
   - Update residual capacities
3. **Return** flow f (this is the maximum flow)

1. **Initialisation**:
   - Start with $f(e) = 0$ for each edge e in the set of edges E, implying that initially, there is no flow through the network.
2. **Finding Augmenting Paths**:
   - Look for a path from the source s to the sink t in the residual network such that each edge along the path has positive residual capacity.
3. **Augmenting Flow**:
   - Once an augmenting path is found, increase the flow along the path by the bottleneck capacity (the smallest residual capacity on any edge in the path).
4. **Repeat the Process**:
   - Continue finding augmenting paths and increasing the flow until no more augmenting paths can be found.
5. **Algorithm Termination**:
   - When you can no longer find any augmenting paths, the flow f is a maximum flow, and the algorithm terminates.

Key Points:

- **Residual Graph**: The residual graph reflects the capacity left for each edge to carry more flow. It changes as flows are augmented along paths.
- **Bottleneck Capacity**: For each augmenting path, the flow is increased by the bottleneck capacity, which is the maximum additional flow that can be pushed through the path without violating capacity constraints.
- **Edge Direction**: The algorithm considers reverse edges to allow for the possibility of "undoing" or reducing flow if it leads to an improved overall solution.
- **Complexity**: While the Ford-Fulkerson method is intuitive and often efficient in practice, its runtime depends on the magnitude of the flow. If capacities are integers, the algorithm runs in $O(m \cdot |f_{\max}|)$, where $|f_{\max}|$ is the value of the maximum flow.

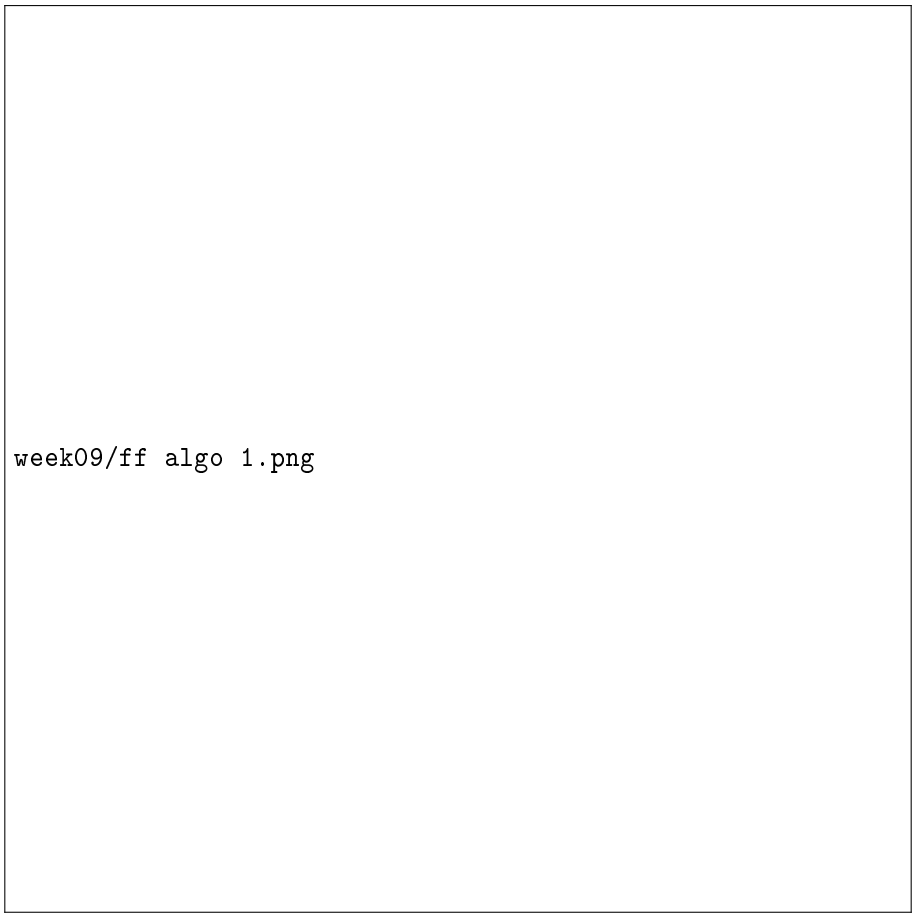

Figure 158: Ford-Fulkerson Step 1: Initial flow network with all flows set to zero.

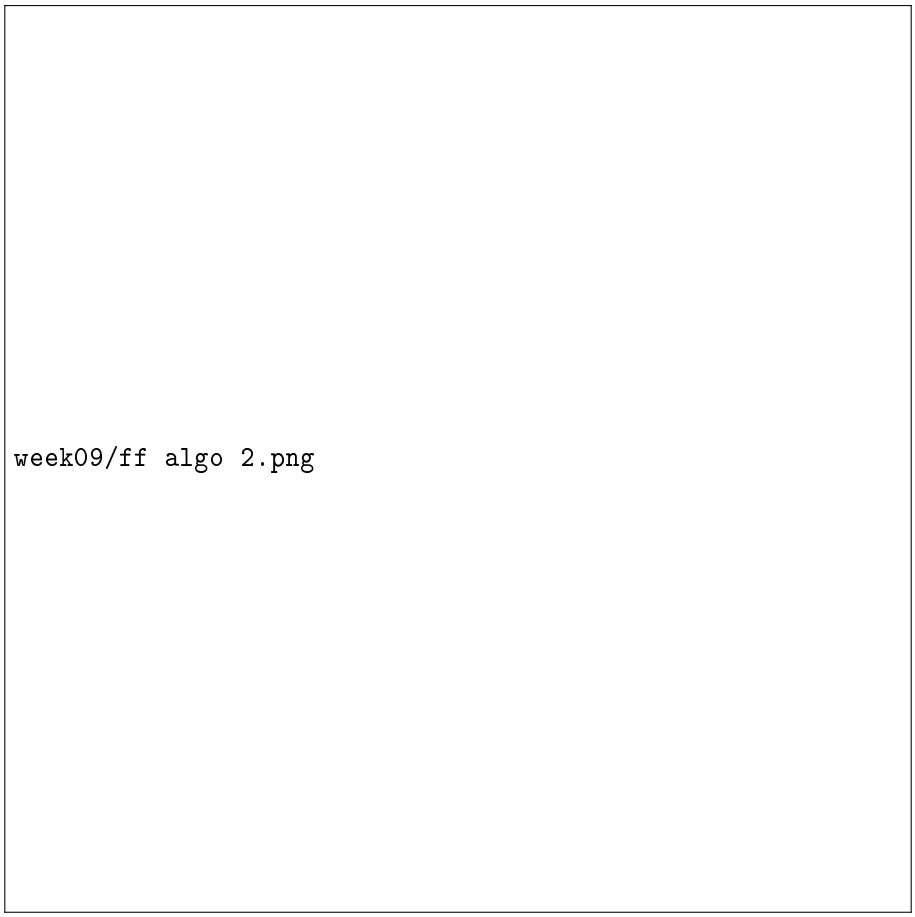

Figure 159: Ford-Fulkerson Step 2: Find an augmenting path and compute its bottleneck.

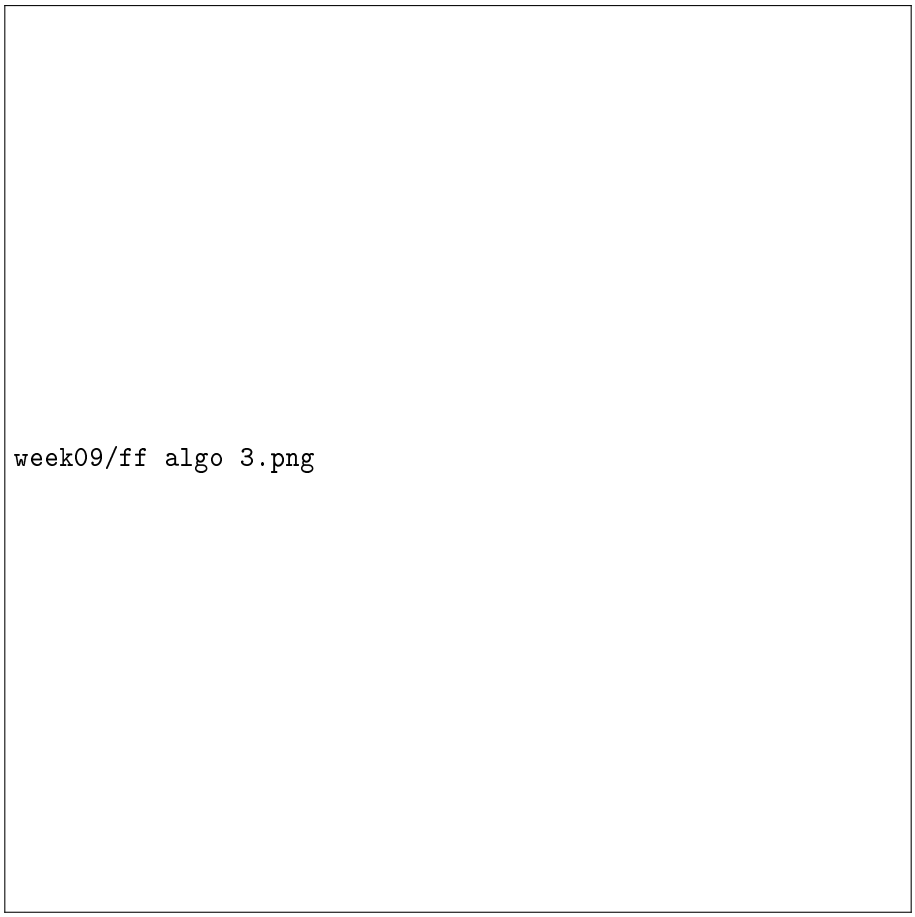

Figure 160: Ford-Fulkerson Step 3: Augment flow along the path and update residual capacities.

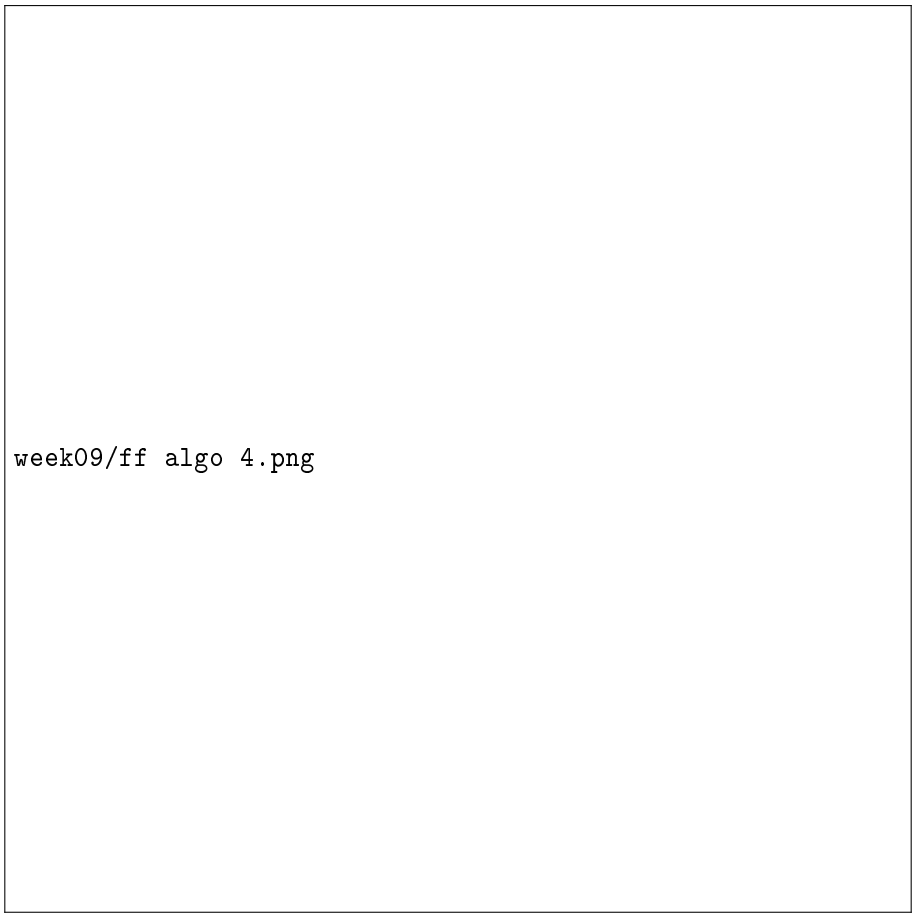

Figure 161: Ford-Fulkerson Step 4: Continue finding augmenting paths in the updated residual network.

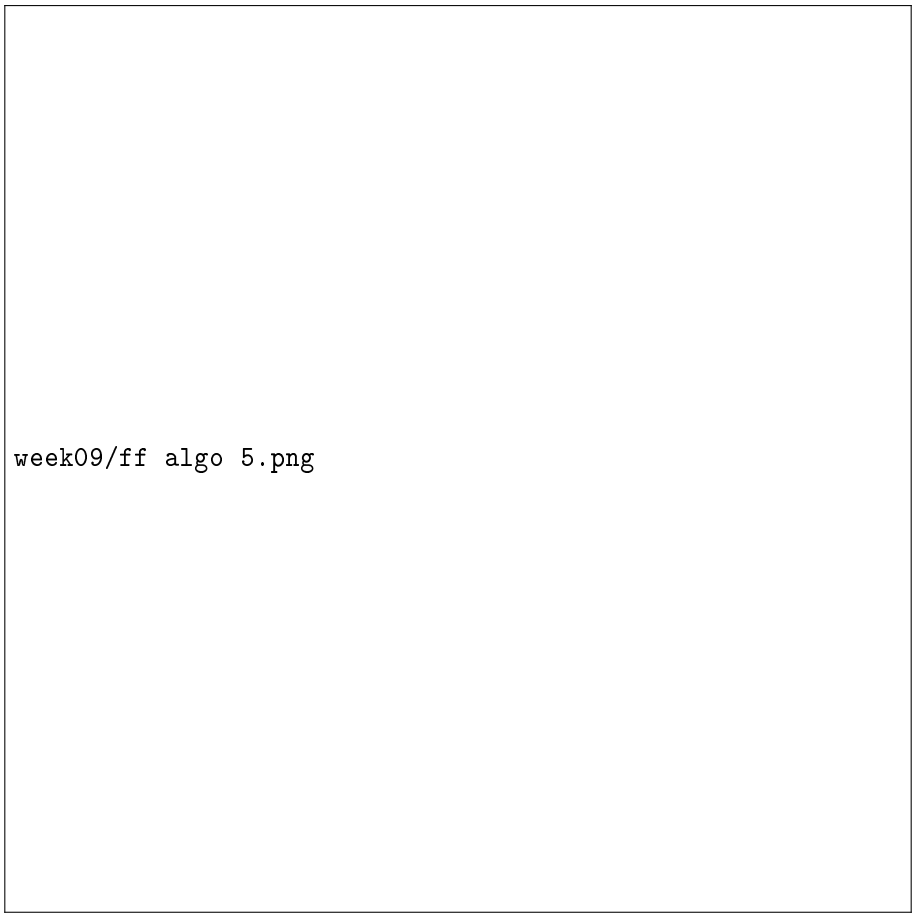

Figure 162: Ford-Fulkerson Step 5: Another augmentation using a path that includes a backward edge.

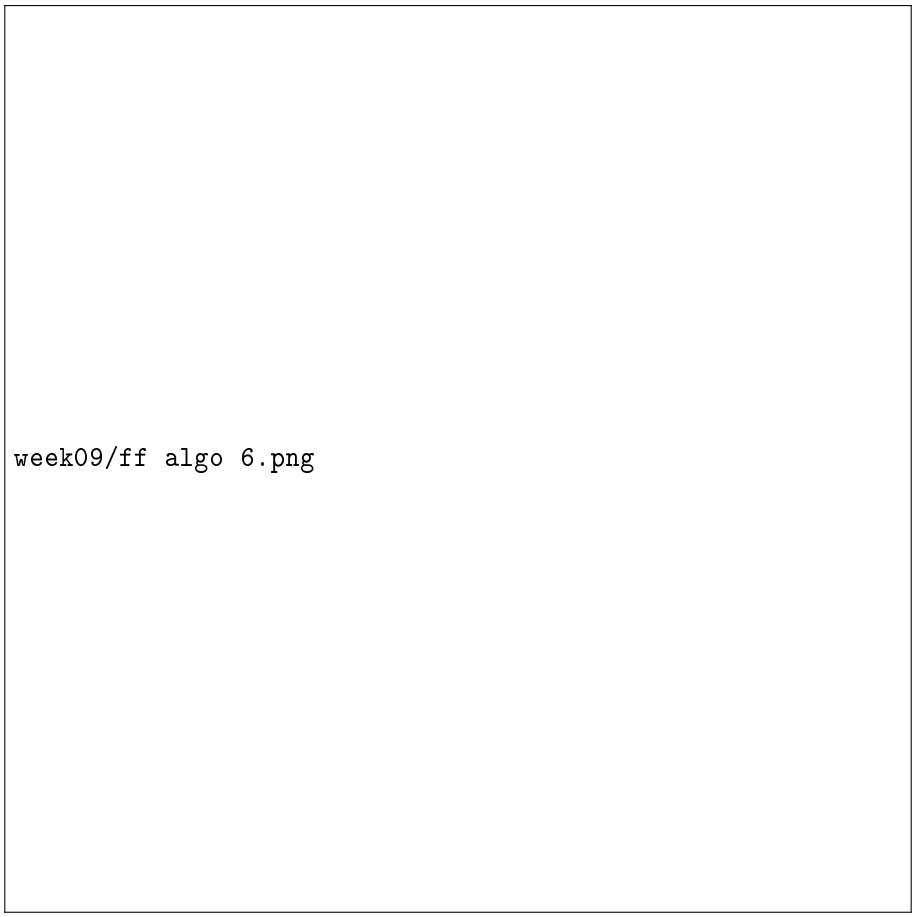

Figure 163: Ford-Fulkerson Final: No more augmenting paths exist - maximum flow achieved.

73.5.5 Edmonds-Karp Algorithm - Using BFS to Choose Paths

The **Edmonds-Karp algorithm** is an implementation of Ford-Fulkerson that uses Breadth-First Search (BFS) to choose paths, ensuring systematic selection of the **shortest** augmenting paths in terms of the number of edges.

Ford-Fulkerson: Almost a Good Algorithm!

Ford-Fulkerson can misbehave if paths are chosen badly - in the worst case, you could have an **exponential number of iterations**!

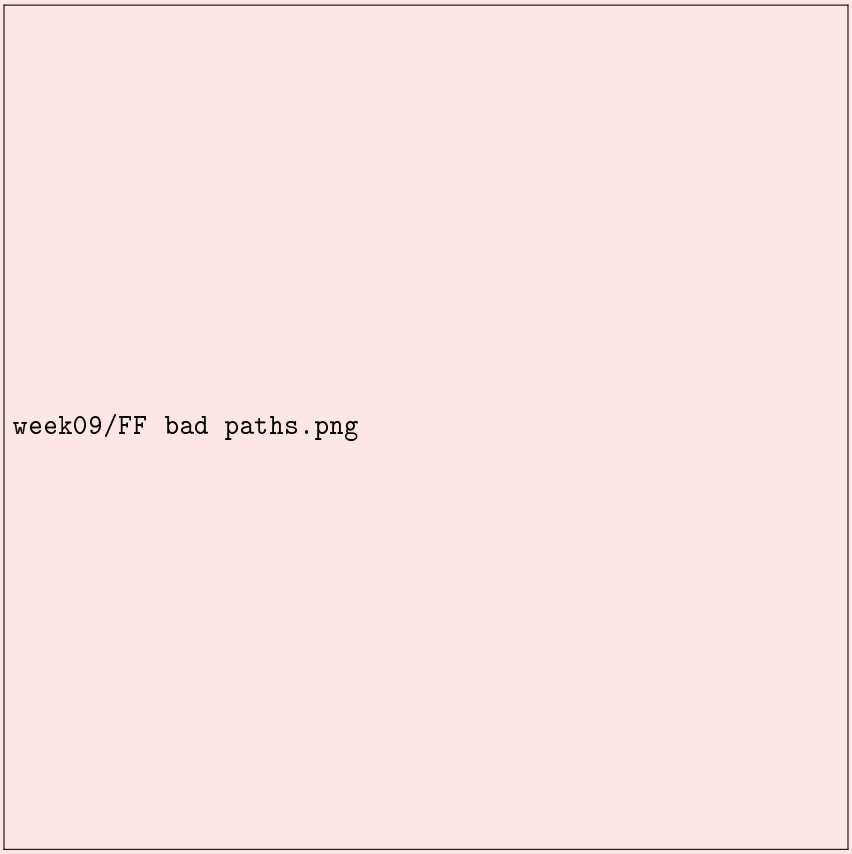

Figure 164: Bad path selection in Ford-Fulkerson - alternating between two paths can lead to exponentially many iterations.

The solution: use BFS to always find the **shortest** augmenting path.

Figure 165: Edmonds-Karp uses BFS to find the shortest augmenting path (fewest edges).

BFS for Augmenting Paths

BFS is utilised to find the shortest augmenting paths because it explores all vertices at the present depth before moving on to vertices at the next depth level, ensuring the path found has the minimum number of edges:

1. **Augmenting Path Lengths**: When BFS is used, the length of the shortest augmenting path, measured in edges, never decreases between iterations. This is because once all shortest paths of a given length have been augmented, any subsequent augmenting paths must necessarily be longer.
2. **Increment in Path Lengths**: After exhausting all augmenting paths of a given length, the only remaining paths (if any) must involve more edges, hence after m augmenting paths of a given length, the shortest augmenting path must increase in length.

Edmonds-Karp Complexity

Per iteration: $O(m)$ (BFS to find shortest augmenting path)

Total iterations: $O(mn)$ (path length increases, bounded by n, and at most m augmentations per length)

Overall complexity: $O(m^2n)$

This is a polynomial-time algorithm, unlike vanilla Ford-Fulkerson which can be exponential.

Complexity Analysis

- **Iteration Complexity**: In each iteration, BFS is used to find the shortest augmenting path, which runs in $O(m)$ time where m is the number of edges. BFS is optimal for this part of the algorithm because it efficiently handles the exploration of all vertices and edges.

- **Total Number of Iterations**: The total number of iterations required by the algorithm is bounded by the number of edges m times the maximum number of distinct shortest path lengths, which in the worst case is $n-1$ (where n is the number of vertices). Hence, the complexity becomes $O(m^2n)$.

Discussion on Efficiency and Improvements

- **Efficiency**: While the $O(m^2n)$ time complexity is a significant improvement over the unrestricted Ford-Fulkerson approach, especially for networks where the capacities are not excessively large relative to the number of edges, it can still be suboptimal for very large or dense networks.
- **Faster Approaches**: Algorithms like Dinic's algorithm and Push-Relabel provide better performance in many scenarios. Dinic's algorithm, for example, uses a combination of BFS for constructing a "level graph" and DFS for finding blocking flows, resulting in an $O(n^2m)$ complexity, which can be more efficient especially in dense graphs.

73.6 Max-Flow Min-Cut Theorem

Max-Flow Min-Cut Theorem

In any flow network, the **maximum value of an s-t flow** equals the **minimum capacity of an s-t cut**.

$$\max_f |f| = \min_{(A,B)} \text{cap}(A,B)$$

Proof sketch:

1. **Flow $\leq$ Cut**: For any flow f and any cut (A,B), we have $|f| \leq \text{cap}(A,B)$. This is because all flow from s to t must cross the cut, and cannot exceed the cut's capacity.
2. **Equality at optimum**: When Ford-Fulkerson terminates with no augmenting path, define A as the set of vertices reachable from s in the residual network. Then:
   - $s \in A$ and $t \notin A$ (otherwise there would be an augmenting path)
   - Every edge from A to $B = V \setminus A$ is saturated: $f(e) = c(e)$
   - Every edge from B to A has zero flow: $f(e) = 0$

   Therefore $|f| = \text{cap}(A,B)$, achieving equality.

Finding the Min-Cut from Max-Flow

After computing max-flow (using Ford-Fulkerson, Edmonds-Karp, or Dinic's algorithm), you can find the min-cut in $O(m)$ time:

Method:

1. Construct the final residual graph G_f
2. Use BFS/DFS from s to find all vertices reachable in G_f
3. Let A = set of reachable vertices, $B = V \setminus A$
4. The edges crossing from A to B in the original graph form the min-cut

Process for Finding Min-Cut:

1. **Compute Residual Graph**: After finding the max-flow, construct the residual graph G_f.
   - Each edge $e = (u \to v)$ has residual capacity $c_f(e) = c(e) - f(e)$
2. **Identify Reachable Vertices**: Run BFS/DFS from source s *in the residual graph* G_f - mark all vertices reachable from s.
3. **Form the Min-Cut**:
   - A = set of all vertices reachable from s in G_f
   - $B = V \setminus A$
   - The edges crossing from A to B in the original graph G form the min-cut

The max-flow min-cut theorem is one of the fundamental results in combinatorial optimisation, connecting two seemingly different problems and providing both an algorithm (Ford-Fulkerson finds both max-flow and min-cut) and a certificate of optimality (the min-cut proves the flow is maximal).

DS&A Lecture Notes: Wk 10
Greedy Algorithms

74 Greedy Algorithm Outline

Greedy Algorithms at a Glance

Greedy algorithms build solutions incrementally, making the locally optimal choice at each step. They succeed when local optimality leads to global optimality-a property that must be proven for each specific problem.

Key characteristics:

1. Process input in a particular order (typically sorted by some criterion)
2. Make irrevocable decisions-once a choice is made, it is never reconsidered
3. Succeed when the **greedy choice property** holds

The core principle of a greedy algorithm is to build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit or is the most "greedy".

This approach makes decisions from the given solution domain based on some local optimum criterion, hoping to find a global optimum by the end.

1. **Processes input in a particular order** - typically sorted based on a certain criterion relevant to the problem!
2. **Makes irrevocable decisions**

Greedy algorithms make **local optimum choices**.

As a result, greedy algorithms **only succeed (find the global optimum solution) when they never make decisions that are irreversibly wrong**.

It is the existence of an appropriate ordering criterion structure that lies behind success (/failure) of a greedy algorithm.

Greedy Choice Property

A problem exhibits the **greedy choice property** if a globally optimal solution can be arrived at by making locally optimal (greedy) choices. More formally:

At each decision point, the choice that appears best at the moment can be extended to a globally optimal solution.

To prove a greedy algorithm is correct, one typically shows:

1. **Greedy choice property**: There exists an optimal solution that includes the greedy choice
2. **Optimal substructure**: After making the greedy choice, the remaining subproblem is also optimally solvable by the same greedy strategy

Common proof techniques include:

- **Greedy stays ahead**: Show that at each step, the greedy solution is at least as good as any other
- **Exchange argument**: Show any optimal solution can be transformed into the greedy solution without worsening it

There is no simple definition of a "greedy" algorithm. They are *'myopic'*-they take action now, without worrying about the future.

- **Greedy algorithms stay ahead:** involves proving that at each step of the algorithm, the solution is at least as good as, if not better than, any other possible solution up to that point.
- **Structural:** rely on the structural properties of the problem. This means the problem must be structured in such a way that local optimisation leads to global optimisation.

75 Cashier's Algorithm

Cashier's Algorithm Summary

Problem: Find the optimal way to give change using the fewest coins possible.

Structure: Sort coins by denomination amount (descending).

Solution: Repeatedly add the largest denomination coin that does not exceed the remaining amount.

Complexity: $O(k)$ where k is the number of denominations (assuming amounts are bounded).

Caveat: Only optimal for certain coin systems (e.g., UK/US coins). Fails for arbitrary denominations.

Problem: optimal way to give change using the fewest coins possible.

Algorithm: Adds the largest value coin that does not take us past the desired amount.

Structure: sort coins by denomination amount.

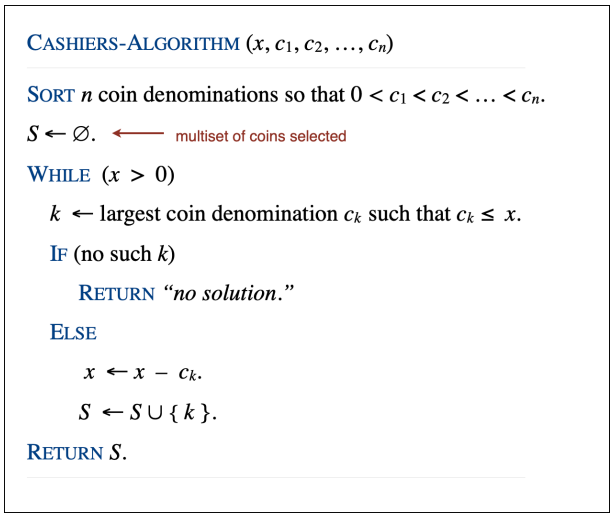

Figure 166: Cashier's algorithm pseudocode: greedily select the largest denomination that fits

Conditions:

Where this is optimal (correct + efficient) depends on the coin denominations. (See slides for examples of non-optimal solutions.)

- c_k = denomination of coin k
- $N(c_k)$ = number of coins with value c_k

When Greedy Fails for Coin Change

The greedy algorithm for coin change is **not always optimal**. It depends entirely on the structure of the coin denominations.

Counterexample: Consider coins $\{1, 3, 4\}$ and target amount 6.

- Greedy: $4 + 1 + 1 = 6$ (3 coins)
- Optimal: $3 + 3 = 6$ (2 coins)

For arbitrary coin systems, dynamic programming is required to guarantee optimality.

Conditions for Greedy Optimality in Coin Change

The greedy algorithm is optimal when the coin denominations form a **canonical coin system**. Informally, this requires:

1. Each larger denomination c_k should not be expressible as a sum of fewer smaller denomination coins
2. No combination of smaller coins provides a "cheaper" way (fewer coins) to make up any amount achievable with a single larger coin

More precisely, the greedy algorithm is optimal if for all amounts x, the greedy solution uses no more coins than any other solution.

Standard UK/US denominations $\{1, 5, 10, 25, 100\}$ (or $\{1, 2, 5, 10, 20, 50, 100, 200\}$ for UK) satisfy these conditions.

The table below shows bounds on coin counts that any optimal solution must satisfy for US coins:

Table 4: Optimal solutions for coin change with US denominations must satisfy these bounds

k	c_k	All Optimal Solutions Must Satisfy	Max Value of $c_1, \ldots, c_{k-1}$ in Optimal Solution
1	1	$N(1) \leq 4$	-
2	5	$N(5) \leq 1$	4
3	10	$N(5) + N(10) \leq 2$	$4 + 5 = 9$
4	25	$N(25) \leq 3$	$20 + 4 = 24$
5	100	No limit	$75 + 24 = 99$

The table captures the key insight: for US coins, you never need more than 4 pennies (because 5 pennies = 1 nickel), never more than 1 nickel (because 2 nickels = 1 dime), and so on. These bounds ensure that the greedy approach of always taking the largest coin works correctly.

76 Activity Selection (Earliest-Finish-Time-First Algorithm)

Activity Selection Summary

Problem: Select the maximum number of mutually compatible (non-overlapping) activities.

Structure: The finishing time is the crucial constraint-it determines how many subsequent activities can fit.

Solution: Sort activities by finish time; greedily select the next compatible activity.

Complexity: $O(n \log n)$ (dominated by sorting).

Optimality: Provably optimal via "greedy stays ahead" argument.

76.1 Problem Setup: Interval Scheduling / Activity Selection

- Job j starts at s_j and finishes at f_j

- Two jobs are compatible if they do not overlap
- Goal: Find the largest subset of mutually compatible jobs

This equals selecting the maximum number of mutually compatible activities from a given set.

This problem is significant in scenarios where you want to schedule jobs, tasks, or events such that they do not overlap in time, and the goal is to maximise the number of tasks that can be completed.

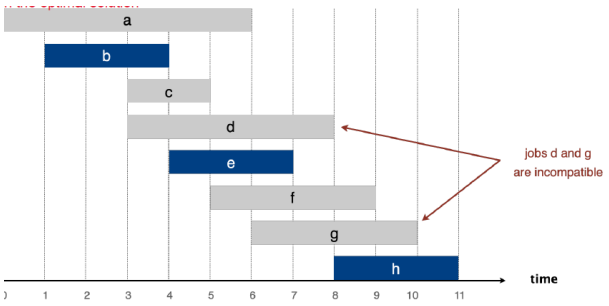

Figure 167: Interval scheduling problem: select the maximum number of non-overlapping intervals

76.2 Solution: Earliest-Finish-Time-First Algorithm

Structure: Consider jobs in order of earliest finish time: i.e., ascending order of f_j.

See counterexamples on slide for why other orderings (shortest job, earliest start, fewest conflicts) fail.

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

$S \leftarrow \emptyset$. ← set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.

Figure 168: Earliest-finish-time-first algorithm pseudocode

76.2.1 Process

1. **Sorting** - all activities by their finish times. This sorting is essential because it allows the greedy choice strategy.
2. **Selection**
   - Starting with the first activity in the sorted list (the one that finishes the earliest), you select it for the schedule.
   - For each subsequent activity, you check if its start time s_j is greater than or equal to the finish time f_{j*} of the last selected activity.

NB: Job j is compatible with S iff $s_j \geq f_{j^*}$

Key Intuition

Having ordered by finish time, if you know the finish time of the last selected job, the only thing you need to check is the start time of candidate jobs.

76.3 Complexity Analysis

$O(n \log n)$

1. **Sorting the activities by finish time** ($O(n \log n)$): Sorting is typically the most computationally intensive part of this algorithm.
2. **Iterating through the sorted activities** ($O(n)$): Once the activities are sorted, you only need to go through them once to determine which to include in the final selection (only comparison criterion is $s_j \geq f_{j-1}$). This iteration takes linear time, $O(n)$, since each activity is considered exactly once for inclusion.

NB: The overall time complexity of the algorithm is dominated by the sorting step, making it $O(n \log n)$.

Complexity

Earliest-finish-time-first algorithm: $O(n \log n)$

76.4 Optimality

Correctness of Earliest-Finish-Time-First

Claim: The earliest-finish-time-first algorithm produces an optimal solution.

Proof sketch (Greedy Stays Ahead):

Let $i_1, i_2, \ldots, i_k$ be the activities selected by greedy (in order), and let $j_1, j_2, \ldots, j_m$ be activities in some optimal solution (also ordered by finish time).

We prove by induction that for all r: $f(i_r) \leq f(j_r)$.

Base case: $r = 1$. The greedy algorithm selects the activity with the earliest finish time, so $f(i_1) \leq f(j_1)$.

Inductive step: Assume $f(i_r) \leq f(j_r)$. Since j_{r+1} is compatible with j_r in the optimal solution, we have $s(j_{r+1}) \geq f(j_r) \geq f(i_r)$. Thus j_{r+1} is also compatible with i_r. The greedy algorithm considers all activities compatible with i_r and selects the one with the earliest finish time, so $f(i_{r+1}) \leq f(j_{r+1})$.

Since greedy "stays ahead" at each step, it must select at least as many activities as optimal: $k \geq m$. Thus greedy is optimal.

Structure: The reason greedy works here is that the finishing time is the crucial structure: the finishing time is what restricts the amount of jobs that can be run-this is the constraint.

Solution: Sort jobs by finish time, at each point choose the next compatible job in the list.

(Greedy algorithms generally have the structure where you sort by something, then crawl along the sorted list to select on some criteria.)

77 Interval Partitioning Problem (Resource Allocation)

Resource Allocation Algorithm Summary

Problem: Find the fewest number of resources (classrooms) to host intervals (lectures).

Structure: Compatible lectures to a given classroom depend on the *current lecture's starting time* and *all classrooms' finishing times*.

Solution: Sort jobs by *start time*, assign the next job to a free classroom (or start a new one).

Complexity: $O(n \log n)$ using a priority queue for classroom availability.

77.1 Problem Setup

Objective: Assign a set of intervals (lectures) to the minimum number of resources (classrooms) such that no two intervals overlap in the same resource.

Input: A set of lectures, each defined by a start time s_j and a finish time f_j.

Sorting: (Unlike the earliest-finish-time-first approach used in activity selection) consider lectures in ascending order of their *start times* s_j.

Room Allocation:

- Initialise an array or list of classrooms. Each classroom keeps track of the end time of the last lecture scheduled in that room.
- For each lecture:
  - Check available classrooms to find one where the last scheduled lecture finishes before the current lecture starts.
  - If such a classroom is found, assign the lecture there and update the classroom's finish time.
  - If no existing classroom can accommodate the lecture, open a new classroom, assign the lecture to it, and set its finish time.

Output: The total number of classrooms used provides the solution to the problem.

77.2 Algorithm

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$. ← number of allocated classrooms

FOR $j = 1$ TO n

  IF (lecture j is compatible with some classroom)

    Schedule lecture j in any such classroom k.

  ELSE

    Allocate a new classroom $d + 1$.

    Schedule lecture j in classroom $d + 1$.

    $d \leftarrow d + 1$.

RETURN schedule.

Figure 169: Earliest-start-time-first algorithm for interval partitioning

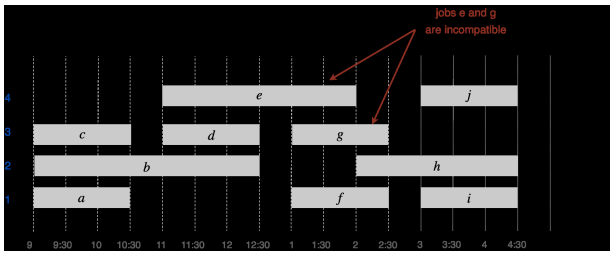

Figure 170: Incorrect approach: not ordered by start time leads to suboptimal classroom usage

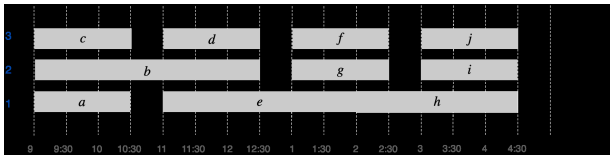

Figure 171: Correct approach: ordering by start time yields minimum classrooms

Priority Queue
A specialised data structure that operates much like a regular queue or list, but with an added feature: each element has a "priority" associated with it.

Priority Queue Key Properties

- Makes it easy to find a min or a max: $O(1)$
- Think of it as a partial sorting of your data
- Typically stored as a binary heap

1. **Find Maximum/Minimum**:
   - **Complexity**: $O(1)$
   - **Description**: Returns only. Constant time because max (/min) element is always at a known position (e.g., the root of the heap).
2. **Delete Maximum/Minimum**:
   - **Complexity**: $O(\log n)$
   - **Description**: Removes & returns max (/min) priority element from queue. Logarithmic due to the need to reorganise the heap to maintain its properties after removal.
3. **Insert**:
   - **Complexity**: $O(\log n)$
   - **Description**: Element is initially inserted at the lowest possible level of the heap and then "bubbled up" to restore heap order, takes log time.
4. **Increase Key** (/ Decrease Key for a min-heap):
   - **Complexity**: $O(\log n)$
   - **Description**: Increases the priority of an element in the queue. After increasing, the heap order might be violated, so the element may need to be moved up in the tree, taking logarithmic time.
5. **Meld**:
   - **Complexity**: $O(n)$
   - **Description**: Combines two priority queues into one, preserving the properties of the priority queue. This operation generally involves building a new heap from all the elements, which takes linear time.

How Priority Queues Work: Heaps

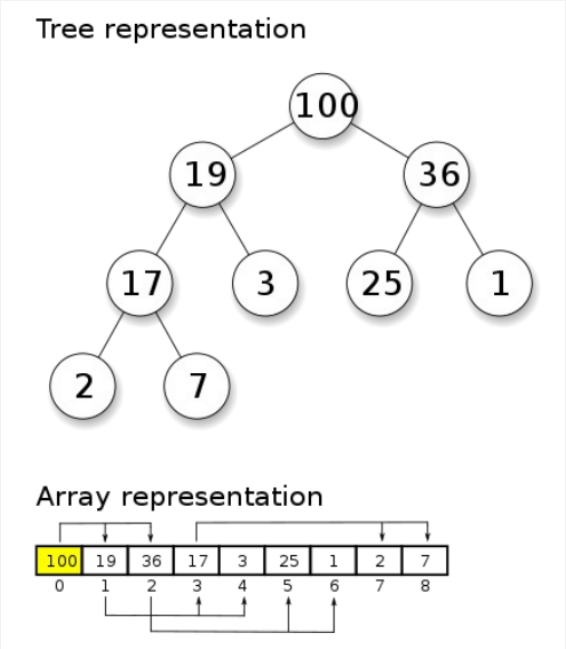

Figure 172: Binary heap structure: parent nodes satisfy heap property with children

Priority queues are often implemented using a **heap**, which is a complete binary tree where every parent node has a value greater than or equal to (in a max-heap) or less than or equal to (in a min-heap) the values of its children. This structure ensures that the element with the highest priority can always be found quickly.

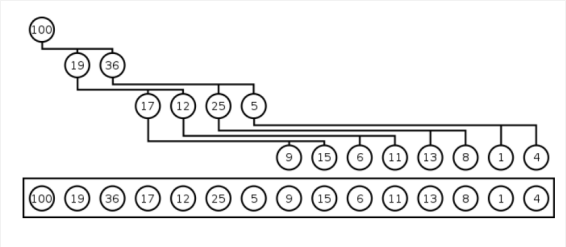

Figure 173: Binary heap stored as an array: children of node i are at positions $2i+1$ and $2i+2$

- **Partial Sorting**: The heap does not fully sort its elements; it only partially sorts them to satisfy the heap property. This means that while the root node will always contain the maximum (or minimum) value, the rest of the tree does not necessarily follow a strict order.
- **Tree Traversal**: Inserting or removing elements requires "walking" the tree-either up or down-to ensure the structure remains a valid heap.

Implementation Details

- **Binary Tree in an Array**: The binary tree is typically represented using an array. The children of the node at position i in the array are found at positions $2i+1$ and $2i+2$, and the parent of any node (except the root) is found at position $\lfloor (i-1)/2 \rfloor$.
- **Choice of Min or Max**: When designing a heap, you decide whether to make it a min-heap or a max-heap depending on whether you want quick access to the minimum or maximum element.

77.3 Complexity Analysis

1. **Sorting Lectures**:
   - **Operation**: Sort all lectures by their *start time* s_j.
   - **Complexity**: $O(n \log n)$, where n is the number of lectures.
   - **Purpose**: This allows the algorithm to process lectures in the order they begin, facilitating efficient scheduling.
2. **Using a Priority Queue**: for *classroom availability*
   - **Operation**: Maintain a priority queue to keep track of the earliest time a classroom becomes available (keyed by finish times of lectures).
   - **Complexity for Various Operations**:
     - **Inserting a New Classroom**: $O(\log n)$ for each operation.
     - **Scheduling a Lecture (Increase-key from k to f_j)**: $O(\log n)$, necessary when a lecture is scheduled in an existing classroom to *update the classroom's availability* time.
     - **Checking Classroom Availability (Find-minimum)** - check if lecture j is compatible by comparing s_j to `Find-minimum`: $O(1)$, to quickly find the earliest available classroom.
3. **Classroom Allocation**:
   - **Operation**: For each lecture, use the priority queue to find the first available classroom that can accommodate it:
     - If the classroom's finish time (from `Find-minimum`) is less than or equal to the lecture's start time, schedule the lecture in this classroom and update the finish time (`Increase-key`).
     - If no classroom is available, insert a new classroom into the priority queue.
   - **Complexity**: The total number of operations in the priority queue is proportional to n, with each operation taking $O(\log n)$.

Overall Complexity

- **Sorting**: $O(n \log n)$.
- **Priority Queue Operations**:
  - Total number of priority queue operations: n
  - Each takes $O(\log n)$

  Each lecture involves a check (`Find-minimum`), potentially an `Increase-key`, and occasionally inserting a new classroom, each taking $O(\log n)$. With n lectures, the total complexity of all priority queue operations collectively also sums up to $O(n \log n)$.
- **Together:** $O(n \log n)$

Complexity

Earliest-start-time-first algorithm (interval partitioning): $O(n \log n)$

77.4 Optimality

The **depth** of a set of open intervals refers to the maximum number of intervals that overlap at any single point in time.

Here, the number of classrooms required is equal to the depth.

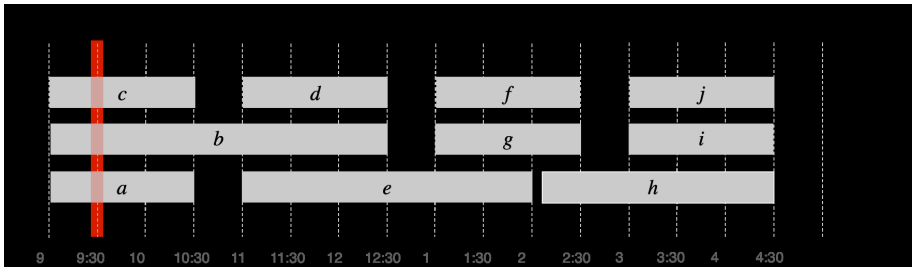

Figure 174: The depth (maximum overlap) provides a lower bound on classrooms needed; the greedy algorithm achieves this bound

The algorithm efficiently utilises the minimum number of classrooms needed to accommodate all lectures without overlap, corresponding exactly to the theoretical minimum inferred from the intervals' depth.

By using this algorithm, one can guarantee that no more resources (classrooms) than absolutely necessary are used.

Structure

Having ordered by start time, we only need to compare the next item's start time against previous items' finish time(s) (made available through a priority queue).

Correctness of Interval Partitioning

Claim: Earliest-start-time-first never schedules two incompatible lectures in the same classroom, and uses the minimum number of classrooms.

Proof:

- Let d be the number of classrooms the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture (j) that is incompatible with all $d-1$ other classrooms.
- Therefore, these $d-1$ lectures each end after s_j.
- We sorted by start-time, so the incompatible lectures start no later than s_j.
- So we have d overlapping lectures at time $s_j + \epsilon$.
- This means the depth of the interval set is at least d, which is a lower bound on any solution.
- Since our algorithm uses exactly d classrooms, it is optimal.

77.5 Summary

- **Problem**: Find the fewest number of classrooms to host lectures.
- **Structure**: Compatible lectures to a given classroom depend on the current lecture's starting time and all classrooms' finishing times.
- **Solution**: Sort jobs by start time, assign the next job to the free classroom (or start a new one).

78 Greedy Algorithms on Graphs

We now turn to greedy algorithms that operate on graphs. The two classic examples are Dijkstra's algorithm for shortest paths and algorithms for finding minimum spanning trees.

79 Dijkstra's Algorithm

Dijkstra's Algorithm Summary

Problem: Find shortest path from source node s to every other node.

Structure: The shortest edge between the explored and unexplored set is always a valid part of the solution.

Solution: Add the shortest edge between explored and unexplored set at each iteration.

Complexity: $O((m+n)\log n)$ with binary heap; $O(m + n\log n)$ with Fibonacci heap.

Requirement: All edge weights must be non-negative.

Intuition: This is essentially BFS for a weighted graph-because each weight/distance is unique, each node becomes its own level set.

Non-Negative Weights Required

Dijkstra's algorithm **requires all edge weights to be non-negative**. For graphs with negative edge weights, use the Bellman-Ford algorithm instead.

See the Wikipedia page for good visualisations of Dijkstra's algorithm in action.

79.1 Problem Setup

Single-pair/Single-source shortest path problem:

Given a directed graph $G = (V, E)$:

- edge lengths/weights $l_e \geq 0$
- source $s \in V$
- destination $t \in V$

Single-pair: find shortest path $s \to t$

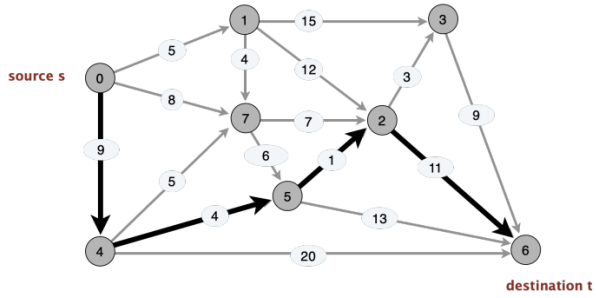

Figure 175: Single-pair shortest path problem: find the minimum-weight path from s to t

Single-source: find shortest path $s \to$ every node

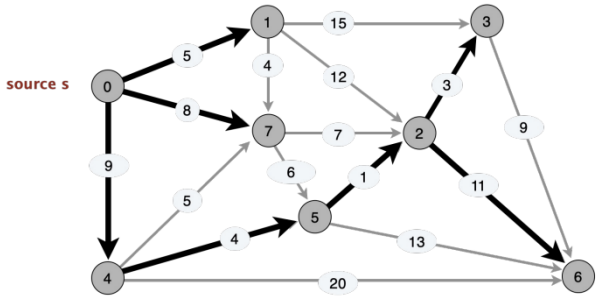

Figure 176: Single-source shortest paths: Dijkstra's algorithm finds shortest paths from s to all other nodes

79.2 Algorithm

Intuition: Maintain a set of explored nodes about which we know shortest paths from source to periphery; repeatedly expand that set by one node (the shortest edge between explored and unexplored).

Essentially this is BFS, but where each level set is just 1 node, because distances are unique.

1. **Initialisation**:
   - Start with a set S of explored nodes, initially containing only the source node s.
   - Initialise the distance to the source node $d[s]$ as 0, since the distance from a node to itself is zero.
   - For all other nodes u in the graph, set the initial distances $d[u]$ to infinity, representing that they are initially unreachable from the source.
2. **Node Selection**: repeatedly choose a single unexplored node.
   - In each iteration, select the node $v \notin S$ (i.e., a node not yet in the set of explored nodes) that minimises the sum of the distance from the source to a node u in S and the weight of the edge from u to v.
   - This is expressed as:
   $$\pi(v) = \min_{e=(u,v):u \in S} (d[u] + l_e)$$
   Where:

- u: An element we already know the shortest path to (with known length $d[u]$)
- v: An element we do not yet know the shortest path to
- l_e: The weight of the edge from node u to node v

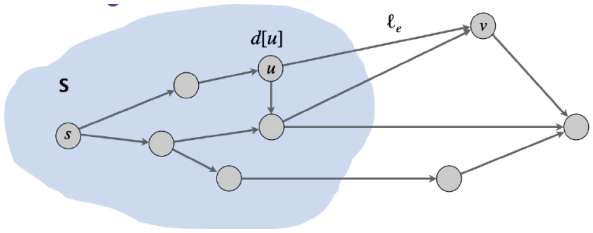

Figure 177: Considering adding node v to the explored set S: we examine all edges crossing the frontier

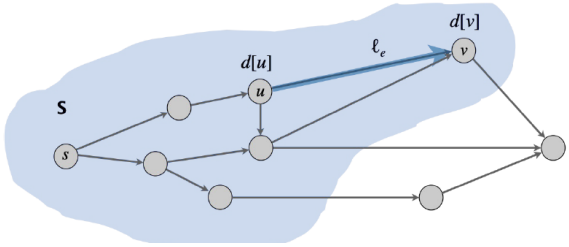

Figure 178: The greedy choice: select the node v that minimises $d[u] + l_{(u,v)}$ over all frontier edges

3. **Update Distances**:
   - Add the selected node v to the set S.
   - Set $d[v]$ to $\pi(v)$, the calculated minimum distance.
4. **Path Recovery**:
   - To reconstruct the shortest path from the source node to any other node v, maintain a predecessor (or parent) pointer pred[v] for each node. Update pred[v] to the node u that achieved the minimum in $\pi(v)$.
5. **Termination**:
   - The algorithm repeats the selection and update steps until all nodes are included in S, meaning that the shortest paths from the source to all nodes have been determined.

Key Points to Remember

- **Priority Queue**: To efficiently find the node that minimises $\pi(v)$, it is common to use a priority queue (often implemented as a min-heap). The priority queue holds all nodes not yet included in S, with priorities corresponding to their current shortest known distances $\pi[v]$.

Clarification on $\pi[v$

$]The priority queue stores \pi[v]$ values only for nodes adjacent to S (the frontier). Nodes not yet adjacent to any explored node have $\pi[v] = \infty$. As S grows, more nodes become adjacent and get finite $\pi[v]$ values.

Crucially, $\pi[v]$ is the *best known* distance so far, not the true shortest distance (until v is added to S).

- **Edge Relaxation**: The operation of checking and updating the shortest path estimate $\pi[v]$ is known as *relaxation*. For each neighbouring node v of u, if $\pi[u] + l_{uv} < \pi[v]$, then update $\pi[v]$ to $\pi[u] + l_{uv}$ and set $\text{pred}[v] = u$.
- **Complexity**: The computational complexity of Dijkstra's algorithm depends on the implementation. With a binary heap as the priority queue, the complexity is $O((V + E) \log V)$, where V is the number of vertices and E is the number of edges in the graph.

79.3 Correctness

Correctness of Dijkstra's Algorithm

Claim: For each node $u \in S$, $d[u]$ is the length of the shortest $s \rightsquigarrow u$ path.

Proof by induction on $|S|$:

Base Case: When $S = \{s\}$, we have $d[s] = 0$, which is trivially correct (the shortest path from s to itself has length 0).

Inductive Step: Assume the claim holds for all nodes currently in S. We show it holds when we add node v to S.

Let v be the node added to S with $d[v] = \pi(v)$. Suppose for contradiction that there exists a shorter path P from s to v.

Let (x, y) be the first edge in P that crosses from S to $V \setminus S$ (i.e., $x \in S$ and $y \notin S$).

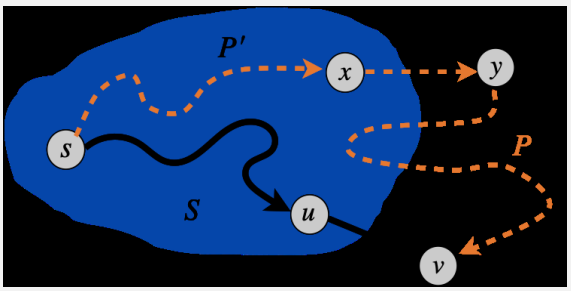

Figure 179: Path validation: if a shorter path P existed, its first edge (x, y) leaving S would have been chosen instead of (u, v)

Then:

- By the inductive hypothesis, $d[x]$ is the true shortest distance to x.
- The length of P is at least $d[x] + l_{xy} +$ (length from y to v).
- Since edge weights are non-negative, length of $P \geq d[x] + l_{xy} \geq \pi(y)$.
- But we chose v because $\pi(v) \leq \pi(y)$ for all $y \notin S$.
- So length of $P \geq \pi(y) \geq \pi(v) = d[v]$.

This contradicts the assumption that P is shorter than $d[v]$. Therefore, $d[v]$ is the true shortest distance to v.

79.4 Efficiency and Implementation

Efficiency Improvements in Dijkstra's Algorithm

Efficiency improvements are achieved by using a *priority queue* and *maintaining a dynamic set of shortest path estimates* $\pi[v]$ *for each node* v *not in* S.

Priority Queue and $\pi[v]$

The algorithm maintains $\pi[v]$ for each node v:

- Priority queue stores unexplored nodes ($v \notin S$), using $\pi[\cdot]$ as priority keys.
- For each, it explicitly maintains $\pi[v]$ instead of computing it directly.
- Crucially initialising $\pi[v] = \infty$ for $v \neq s$.
- Then we use the following update rule:

$$\pi[v] = \min(\pi[v], \pi[u] + l_e)$$

This can also be expressed as:

$$\pi[v] = \begin{cases} \pi[u] + l_e & \text{if there is an edge } (u, v) \text{ with } u \in S \text{ and } v \notin S \text{ of length } l_e \\ \infty & \text{if } v \text{ is not adjacent to any } u \in S \end{cases}$$

Meaning that:

- We are effectively comparing newly estimated $\pi[v] = \pi[u] + l_e$ values for nodes now neighbouring S vs all 'far' / non-adjacent nodes valued at $\pi[v] = \infty$.
- For each $v \notin S$, $\pi(v)$ can only decrease (because set S increases).
- Suppose u is added to S, and there is an edge $e = (u, v)$ leaving u, then it suffices to use the above update rule.
- Once u is deleted from the queue, $\pi[u]$ = length of shortest $s \to u$ path.

```
DIJKSTRA (V, E, ℓ, s)
FOREACH v ≠ s : π[v] ← ∞, pred[v] ← null; π[s] ← 0.
Create an empty priority queue pq.
FOREACH v ∈ V : INSERT(pq, v, π[v]).
WHILE (IS-NOT-EMPTY(pq))
  u ← DEL-MIN(pq).
  FOREACH edge e = (u, v) ∈ E leaving u:
    IF (π[v] > π[u] + ℓe)
      DECREASE-KEY(pq, v, π[u] + ℓe).
      π[v] ← π[u] + ℓe ; pred[v] ← e.
```

Figure 180: Dijkstra's algorithm pseudocode with priority queue operations

1. **Initialisation**:
   - For each node $v \neq s$, initialise $\pi[v] = \infty$ (indicating that the shortest path from s to v is unknown at the start).
   - For the source node s, $\pi[s] = 0$.
2. **Priority Queue Usage**:
   - Nodes $v \notin S$ (i.e., nodes that have not been fully explored) are stored in a priority queue. The priority for each node is given by $\pi[v]$, which represents the *best known* shortest path from s to v at any given time.

- The node with the minimum $\pi[v]$ value is chosen next for exploration (using `Delete-min` operation).

3. **Updating** $\pi[v]$:
   - When a node u is added to S, for each outgoing edge $e = (u, v)$, update $\pi[v]$ if $\pi[u] + l_{uv} < \pi[v]$ where l_{uv} is the weight of the edge from u to v. This is done using the `Decrease-key` operation in the priority queue.
   - This ensures that $\pi[v]$ always holds the shortest path length from s to v that has been discovered so far.

Dijkstra's Algorithm Complexity

Priority Queue Operations:

- **Insertion**: $O(\log n)$ - Inserting a node into the priority queue.
- **Delete-min**: $O(\log n)$ - Removing the node with the smallest $\pi[v]$, which is the next node to be explored.
- **Decrease-key**: $O(\log n)$ - Updating the priority of a node in the queue when a shorter path to it is found.

Operation Counts:

- Each of the n nodes is inserted once and deleted once: $O(n)$ insertions and $O(n)$ delete-mins.
- Each of the m edges can trigger at most one decrease-key operation: $O(m)$ decrease-keys.

Overall Complexity with Binary Heap:

$$O(n \log n + m \log n) = O((n + m) \log n)$$

Since typically $m \geq n$ for connected graphs, this simplifies to $O(m \log n)$.

With Fibonacci Heap: Decrease-key is $O(1)$ amortised, giving:

$$O(n \log n + m)$$

Dijkstra's Algorithm Complexity

- **Binary Heap**: $O((n + m) \log n) = O(m \log n)$
- **Fibonacci Heap**: $O(m + n \log n)$

Implementation Variants

The efficiency can vary depending on the choice of priority queue:

- **Binary Heap**: Provides $O(\log n)$ for key operations as discussed.
- **Fibonacci Heap**: Can improve the `Decrease-key` operation to $O(1)$ amortised time, reducing the overall complexity to $O(m + n \log n)$.

Importance of priority queue for efficiency in Dijkstra's algorithm:

1. Streamlines the process of selecting the next node to explore.
2. Ensures that the algorithm efficiently progresses towards finding the shortest path by always considering the most promising unexplored node.

79.5 Summary

Problem: Find shortest path from node s to every other node.

Structure: The shortest edge between explored and unexplored set is always a valid part of the solution.

Solution: Add shortest edge between explored and unexplored set at each iteration.

This is basically BFS for a weighted graph-because each weight/distance is unique, each node becomes its own level set.

80 Cycle-Cut Intersection

A **cut** is a partition of the nodes into two nonempty subsets, S and $V \backslash S$, where $S \cap (V \backslash S) = \emptyset$.

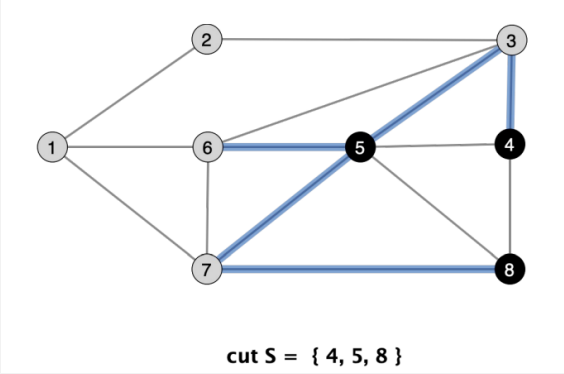

Figure 181: A cut partitions the vertex set into two non-empty subsets

The **cutset** of a cut S is the set of edges with exactly one endpoint in S.

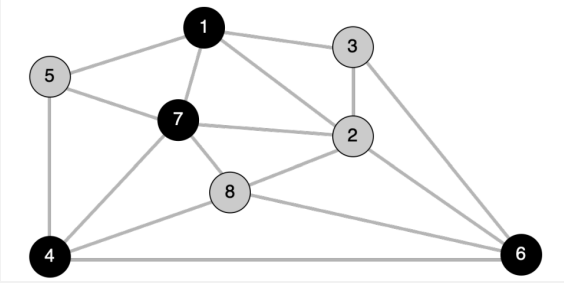

Figure 182: Cutset example: for cut $S = \{1, 3, 6, 7\}$, the cutset contains all edges crossing between S and $V \setminus S$

Cycle-Cut Intersection Property

A cycle and a cutset intersect in an **even number of edges**.

Intuition: You have to go in and out of the cut set, so it will always be even.

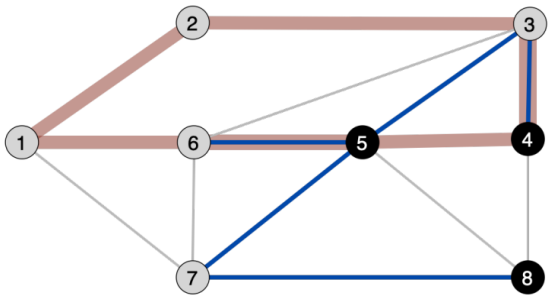

Figure 183: Cycle-cut intersection: every cycle crosses a cutset an even number of times (entering and leaving)

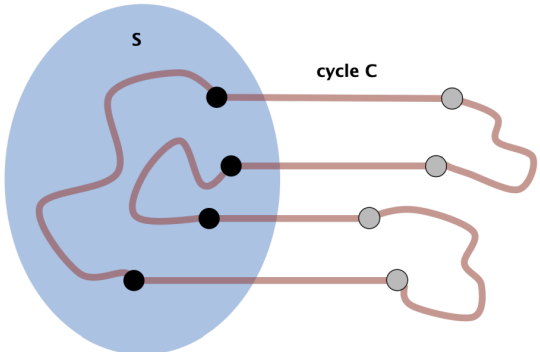

Figure 184: Another example of cycle-cut intersection: the cycle enters and exits the cut region equally often

81 Spanning Trees and Minimum Spanning Trees

81.1 Spanning Tree: General Characteristics

A spanning tree of a graph $G = (V, E)$ is a subgraph $H = (V, T)$ where:

- V is the set of vertices, and T is a subset of E, the set of edges from the original graph G.
  - V remains the same (all nodes included), but $T \subseteq E$
- H must satisfy two key properties:
  1. **Acyclic**: H does not contain any cycles.
  2. **Connected**: H includes all vertices of G and there is a path between any pair of vertices in H.

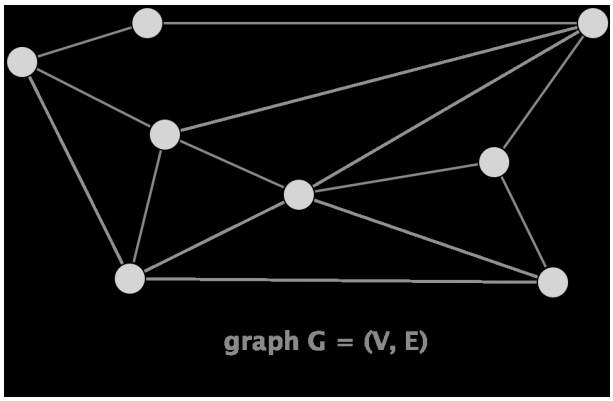

Figure 185: A spanning tree connects all vertices using a subset of edges, forming no cycles

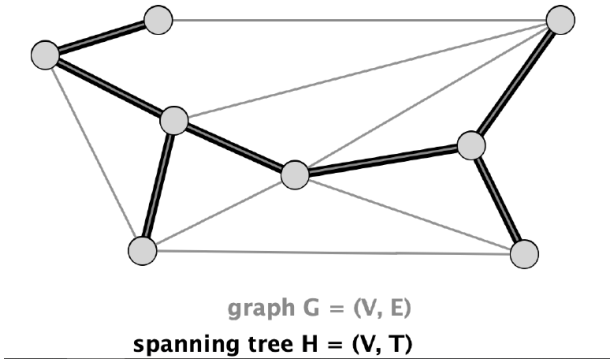

Figure 186: Different spanning trees of the same graph: the same vertices connected by different edge subsets

Characteristics of Spanning Trees

1. **Minimality**: The spanning tree is minimal with respect to the edges. It contains the minimum number of edges required to connect all the vertices, which is $|V| - 1$ edges for a connected graph with $|V|$ vertices.
2. **(Non-)Uniqueness**: The spanning tree is not necessarily unique. A graph may have multiple spanning trees, depending on the structure of the graph and the edges available.
3. **Maximum Spanning Trees (MSTs)**: In weighted graphs, you can further classify spanning trees as minimum or maximum spanning trees, where the sum of the edge weights is minimised or maximised, respectively.

Uses of Spanning Trees

- **Network Design**: In networking, spanning trees are crucial for designing efficient and loop-free networks. Protocols like the Spanning Tree Protocol (STP) are used in Ethernet networks to prevent loop formation by dynamically building spanning trees.
- **Cluster Analysis**: In data science, algorithms that construct spanning trees can be used for clustering analysis, helping to identify natural groupings of data points by treating them as connected graphs.

- **Optimisation**: Various optimisation problems involve finding spanning trees that minimise or maximise certain attributes, such as cost, distance, or time.

Constructing a Spanning Tree (NB: NOT MST)

To construct a spanning tree from a graph:

1. **Start from any vertex** if using a depth-first search (DFS) or breadth-first search (BFS) approach.
2. **Add edges one by one** while avoiding the creation of cycles and ensuring that every vertex gets connected.
3. **Stop when all vertices are included** and exactly $|V| - 1$ edges have been added.

Equivalent Characterisations of Spanning Trees

For a subgraph $H = (V, T)$ of a connected graph $G = (V, E)$, the following are equivalent:

1. H is a spanning tree (connected and acyclic)
2. H is connected and has exactly $|V| - 1$ edges
3. H is acyclic and has exactly $|V| - 1$ edges
4. H is minimally connected (removing any edge disconnects it)
5. H is maximally acyclic (adding any edge creates a cycle)

Properties that follow:

- **Contains exactly $V-1$ edges**: The minimum needed to connect V vertices without cycles.
- **Removal of any edge disconnects it**: Each edge is a bridge; there is no redundancy.
- **Addition of any edge creates a cycle**: Any two nodes already have a unique path connecting them.

81.2 Minimum Spanning Tree (MST)

MST Definition

Given a connected, undirected graph $G = (V, E)$ with edge costs c_e, a **Minimum Spanning Tree** (V, T) is a spanning tree of G such that the sum of the edge costs $\sum_{e \in T} c_e$ is minimised.

MSTs are useful in network design, clustering, and other applications where minimal cost connectivity must be maintained.

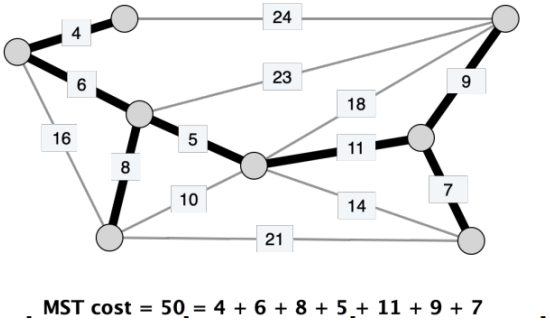

Figure 187: A minimum spanning tree: the spanning tree with the smallest total edge weight

Definition and Properties

- **Minimum Spanning Tree**: An MST is a subset of the edges of a connected, weighted graph that connects all the vertices together, without any cycles, and with the minimum possible total edge weight.
- **Edge Costs**: In a weighted graph, each edge has an associated cost. The goal of the MST is to ensure that the sum of the costs of the edges included in the tree is minimised.

Computational Challenge

- **Brute Force Feasibility**: As per **Cayley's theorem**, a complete graph on n nodes has n^{n-2} spanning trees. This exponential growth in the number of spanning trees makes brute force computation infeasible for anything but the smallest graphs. For example, even a modestly sized graph of 10 nodes would have over 100 million different spanning trees.

Efficient Algorithms for MST

To find the MST, several efficient algorithms are commonly used:

- **Kruskal's Algorithm**: Adds edges in order of increasing weight, using a disjoint-set (or union-find) data structure to ensure that no cycles are formed.
- **Prim's Algorithm**: Starts with a single vertex and grows the MST one edge at a time, always choosing the minimum weight edge that adds a new vertex to the tree.
- **Borůvka's Algorithm**: Another approach that is effective especially in parallel computation contexts.

These algorithms typically run in $O(E \log E)$ or $O(E \log V)$ time, which is feasible for large graphs.

MST and the One-Nearest-Neighbour Graph

- **One-Nearest-Neighbour Graph**: A graph where each vertex is connected to its closest neighbour based on edge cost or some other metric.
- **Important caveat**: An MST does not necessarily contain all the edges from a one-nearest-neighbour graph. The MST ensures overall minimal weight, which means it might skip over the nearest neighbour of a particular vertex if including a slightly more distant neighbour yields lower total cost.

Q: For which changes in the edge costs in graph G will every Minimum Spanning Tree (MST) in G also be an MST in the modified graph G'?

1. $\{c'_e = c_e + 17\}$: Adding a constant value to each edge cost c_e does not affect the relative ordering of edge costs. Therefore, the minimum spanning tree (MST) in the original graph G will remain the same in the modified graph G'. All MSTs in G will also be MSTs in G'.

2. $\{c'_e = 17 \times c_e\}$: Multiplying each edge cost c_e by a constant factor 17 changes the scale of the edge costs but maintains their relative ordering. Again, this does not change the structure of the MSTs in the graph. All MSTs in G will still be MSTs in G'.

3. $\{c'_e = \log(c_e)\}$: Taking the logarithm of the edge costs c_e alters the scale of the costs but also preserves their relative ordering. The logarithm function is monotonic, meaning that it maintains the same order of values. Therefore, the structure of the MSTs will not change in the modified graph G'. All MSTs in G will also be MSTs in G'.

All of the above.
In each case, the changes to the edge costs preserve the relative ordering of the costs. As long as something *monotone* for defining distances between nodes, you are maintaining the MST (the person closest to me is always the person closest to me no matter what scale we use).

81.3 Fundamental Cycle

Key Structure for MST Algorithms

The fundamental cycle is one piece of structure that is leveraged in algorithms to find an MST.

When you have a spanning tree $H = (V, T)$ of a graph $G = (V, E)$, it includes all vertices and is acyclic.

1. **Adding a Non-Tree Edge**: If you add an edge e that is not part of the spanning tree T (i.e., $e \in E$ but $e \notin T$), you will create a cycle. This cycle is unique to the addition of e and is called a **fundamental cycle**.

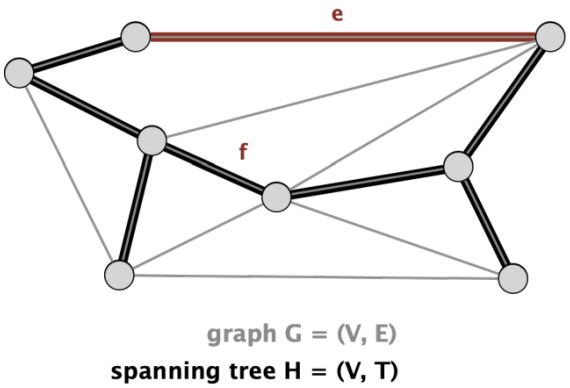

Figure 188: Fundamental cycle: adding edge e (not in tree T) creates exactly one cycle

2. **Properties**:
    - Since T is a tree and hence acyclic, adding e to T connects two nodes in T that previously had exactly one path between them (since trees have a unique path between any two nodes). This addition forms a loop, thereby creating a cycle.
3. **Cycle-Based Spanning Tree Alteration**:
    - If you want to maintain a spanning tree after adding e and creating a cycle C, you can remove any other edge f from C (not necessarily the added edge e). The resulting graph $T \cup \{e\} - \{f\}$ will still be a spanning tree.
4. **Implication for MST**:
    - If the weight c_e of the added edge e is less than the weight c_f of any other edge f in the cycle, then the original tree T was not minimal, as replacing f with e results in a spanning tree with a smaller total weight.

81.4 Fundamental Cutset

Second Key Structure for MST Algorithms

The fundamental cutset is the second piece of structure that is leveraged in MST algorithms.

A fundamental cutset arises when considering the removal of an edge from a spanning tree:

1. **Removing a Tree Edge**: If you remove an edge f from the spanning tree T, the tree splits into two connected components. This division creates a **cutset**, which consists of all the edges that have one endpoint in each of the resulting components.

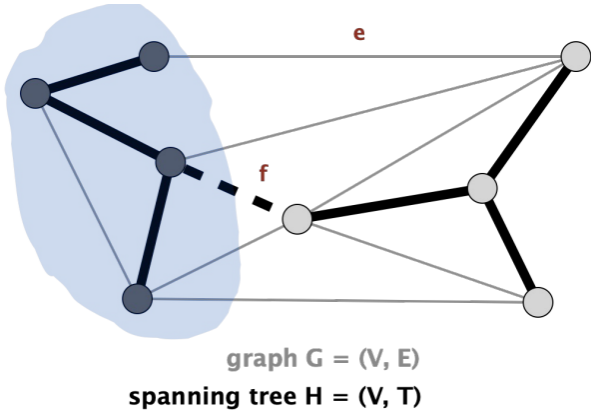

Figure 189: Fundamental cutset: removing edge f splits the tree into two components; the cutset contains all edges crossing between them. If $c_e < c_f$, then (V, T) was not an MST

2. **Cutset-Based Spanning Tree Alteration**:
    - Adding any edge e from this cutset back to the tree (replacing f) will reconnect the two components without creating a cycle, thus forming another spanning tree.
3. **Implication for MST**:

- If any edge e in the cutset has a lower weight than f, then again, T was not minimal. Replacing f with e would yield a spanning tree with a lower total weight.

Both fundamental cycles and cutsets provide mechanisms to explore alternative spanning trees and are particularly useful in optimising and adjusting existing trees. They are central to algorithms like **Kruskal's** and **Prim's** for finding MSTs, where decisions about edge inclusion or exclusion are made based on the properties of these cycles and cutsets.

81.5 Greedy Algorithm to Find MST: Red-Blue Rules

Cut Property for MST

Cut Property: For any cut S of the graph, the minimum weight edge e crossing the cut is in *some* MST.

Proof: Suppose $e = (u, v)$ is the minimum weight edge crossing cut S, and suppose T is an MST that does not contain e. Since T is a spanning tree, there is a unique path from u to v in T. This path must cross the cut S at some edge $f \neq e$ (since $u \in S$ and $v \notin S$).

Consider $T' = T \cup \{e\} - \{f\}$. This is still a spanning tree (we added one edge and removed one from the resulting cycle). Since $c_e \leq c_f$ (as e is the minimum crossing the cut), we have $\text{cost}(T') \leq \text{cost}(T)$. Since T was an MST, T' is also an MST, and it contains e.

Cycle Property for MST

Cycle Property: For any cycle C in the graph, the maximum weight edge e in C is in *no* MST (assuming unique edge weights).

Proof: Suppose e is the maximum weight edge in cycle C, and suppose T is an MST containing e. Removing e from T creates two components. Since C is a cycle containing e, there must be another edge $f \in C$ connecting these two components.

Consider $T' = T - \{e\} \cup \{f\}$. This is still a spanning tree. Since $c_f < c_e$ (as e is the maximum in the cycle), we have $\text{cost}(T') < \text{cost}(T)$, contradicting that T was an MST.

These properties lead to the **Red-Blue algorithm**:

Red Rule: Max Cost in a Cycle → prevents cycles

- Let C be a cycle with no red edges.
- Select an uncoloured edge of C of max cost, colour it red.

Blue Rule: Min Cost in a Cutset → encourages connectivity & minimisation

- Let D be a cutset with no blue edges.
- Select an uncoloured edge in D of min cost, colour it blue.

Overall:

- Apply red and blue rules until all edges are coloured.
- The blue edges form an MST!

- Terminate when $n-1$ edges are blue.

These are the fundamental building blocks for any spanning tree optimisation process; the choice between rules to apply gives different algorithms!

Red Rule: Max Cost in a Cycle

- **Purpose**: Prevents the creation of cycles within the eventual MST. By colouring the max cost edge in any cycle red, the rule effectively ensures that this edge will **not be included** in the MST.
- **Operation**: Whenever a cycle is formed or identified in the graph, and it contains uncoloured edges, the red rule selects the uncoloured edge with the maximum cost and colours it red. This edge is thereby excluded from consideration for inclusion in the MST.

Blue Rule: Min Cost in a Cutset

- **Purpose**: Encourages connectivity and minimisation of the spanning tree's weight by adding the minimum cost edges from cutsets.
  - Leverages the concept of the fundamental cutset, where removing a tree edge creates two components; the lowest cost edge that reconnects these components is crucial for maintaining a minimal structure.
- **Operation**: In any cutset that forms from the current set of tree edges (i.e., the edges included in the MST construction so far), the rule selects the uncoloured edge with the minimum cost and colours it blue. These blue edges are then **candidates for inclusion in the MST**.

Algorithm Progression

- **Application of Rules**: The algorithm alternates between these two rules, systematically colouring edges and effectively deciding which edges are included in the MST (blue) and which are not (red). This approach will incrementally build up the MST by adding the necessary connections (blue edges) while avoiding unnecessary, costly cycles (red edges).
- **Termination**: The process continues until all edges are coloured or, more precisely, **until $n-1$ edges are coloured blue** for a graph with n vertices, at which point an MST has been formed.
  - Remember from before: a spanning tree has $V-1$ edges.

Key Points

- **Difference in Algorithms**: The choice of when to apply each rule can vary, leading to different algorithms or variations in the order in which edges are evaluated and coloured. This flexibility can affect the intermediate steps but will generally lead to the same MST assuming all edges are considered appropriately.
- **Greedy Strategy**: Both rules embody the greedy methodological framework-making the most cost-effective choice available at each step.
  - Blue rule directly selects the minimum cost edge from cutsets to ensure minimal connection costs.
  - Red rule avoids costly cycles by removing the most expensive link from consideration.

81.6 Prim's Algorithm

Prim's algorithm is a special case of the greedy algorithm that repeatedly applies the blue rule.

Prim's Algorithm Summary

Problem: Find the minimum spanning tree of a weighted, connected graph.

Strategy: Grow the MST from a single starting vertex by repeatedly adding the minimum-weight edge that connects a vertex in the tree to a vertex outside.

Complexity: $O(m \log n)$ with binary heap; $O(m + n \log n)$ with Fibonacci heap.

Key insight: This is essentially Dijkstra's algorithm, but instead of tracking distance from source, we track the minimum edge weight to connect to the growing tree.

1. **Initialisation**: ($S = \{s\}$ for any node s, $T = \emptyset$)
   - Start with a set S containing a single arbitrary node s from the graph. This node serves as the starting point for the MST.
   - Initialise T, the set of edges in the MST, to be empty.
2. **Algorithm Execution**:
   (a) Add to T a min-cost edge with exactly one endpoint in S.
   (b) Add the other endpoint to S.
   (c) Repeat $n - 1$ times.
   - **Repeat** $n - 1$ times, where n is the number of vertices in the graph:
     - From the set of edges that have one endpoint in S (the set of nodes already included in the MST) and one endpoint not in S, select the edge (u, v) that has the minimum cost and where $u \in S$ and $v \notin S$.
     - Add this minimum cost edge to T.
     - Add the endpoint v (not previously in S) to S.
   - This step ensures that in each iteration, the tree grows by one edge and one vertex, gradually covering all vertices by connecting the least costly edge to the growing tree.

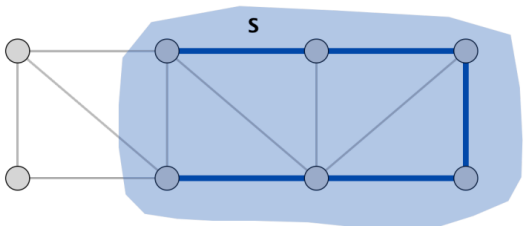

Figure 190: Prim's algorithm visualisation: the tree (blue) grows by adding the minimum-weight edge to an unexplored vertex

3. **Termination**:
   - The algorithm stops after adding $n - 1$ edges since a spanning tree for a graph with n vertices always contains $n - 1$ edges.

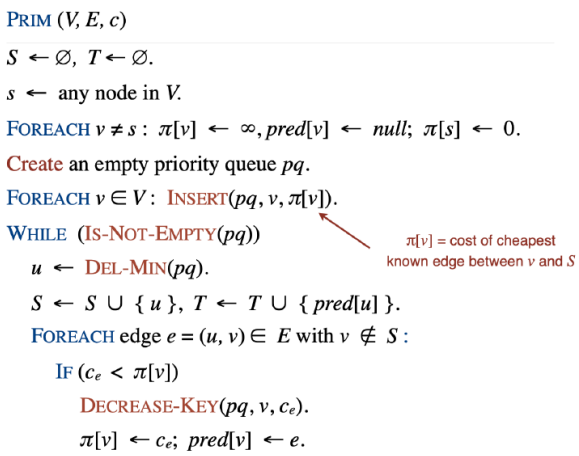

Figure 191: Prim's algorithm pseudocode: similar structure to Dijkstra but tracking edge costs rather than path distances

Implementation Details

- **Priority Queue**:
  - A *dynamic* priority queue is crucial for efficiently retrieving the minimum cost edge at each step.
  - The priority queue typically stores **all the edges crossing the cut** (edges with one endpoint in S and the other not in S).
  - Update priorities (edge costs) as new vertices are added to S.
  - When a vertex is added to S, all new edges connecting this vertex to any vertices outside S need to be considered for addition to the priority queue, or the priorities need updating if they already exist in the queue.
- **Complexity**:
  - The running time primarily **depends** on how the priority queue is implemented.
  - With a **binary heap**, the complexity is $O(m \log n)$, where m is the number of edges and n is the number of vertices. Each edge insertion and extraction operation (insert and delete-min) takes logarithmic time.
  - If an advanced priority queue structure like a **Fibonacci heap** is used, the complexity can be reduced further, especially in terms of the decrease-key operations, which are more efficient in such structures.

Prim's Algorithm Complexity

Operations:

- Each vertex is added to S exactly once: n delete-min operations.
- Each edge is examined at most twice (once from each endpoint): $O(m)$ decrease-key or insert operations.

With Binary Heap:

$$O(n \log n + m \log n) = O(m \log n)$$

(since $m \geq n - 1$ for connected graphs)

With Fibonacci Heap:

$$O(n \log n + m) = O(m + n \log n)$$

Comparison with Dijkstra's Algorithm

- **Similarities**:
  - Both algorithms use a priority queue to manage vertices/edges.
  - Both grow a set starting from a single vertex by adding the "cheapest" next step to a growing structure (the shortest path tree for Dijkstra's, the MST for Prim's).
- **Differences**:
  - Dijkstra's algorithm tracks **distance from the source vertex** to all other vertices, adjusting path lengths based on newly discovered paths.
  - Prim's algorithm tracks **minimum edge weight to connect** to the tree-it is solely concerned with expanding the MST by connecting the nearest vertex not yet in the tree at each step, without reconsidering previously made decisions.

81.7 Kruskal's Algorithm

(Skipped in lectures)

Kruskal's algorithm is another greedy MST algorithm that applies the blue rule differently:

1. Sort all edges by weight (ascending).
2. For each edge in order: if it connects two different components (does not create a cycle), add it to the MST.
3. Use a union-find data structure to efficiently check connectivity.

Complexity: $O(m \log m) = O(m \log n)$ (dominated by sorting).

81.8 MSTs in Experimental Design

Building a Similarity Graph

1. **Data Representation**:
    - Consider each unit (subject, plot of land, etc.) as a vertex in a graph.
    - Calculate distances or dissimilarities between all pairs of units based on relevant characteristics or measurements. These characteristics could be baseline variables like pre-treatment measurements, demographic information, or other relevant metrics.
2. **Edge Weights**:
    - The edges between any two vertices (units) in this graph are weighted by their similarity or dissimilarity. For example, weights could be calculated using a distance measure that reflects how different the units are from each other. The weight might be a simple Euclidean distance for quantitative measurements, or more complex measures tailored to the data, such as kernel-based distances.
3. **Kernel Matrix**:
    - A kernel matrix can be used to represent similarities or dissimilarities among all pairs of units. This matrix can then be transformed into a graph, where each element represents the weight of the edge between two nodes.

Using MST for Group Assignment

1. **Constructing the MST**:
    - Use an algorithm (like Prim's or Kruskal's) to construct an MST from this weighted graph.
    - The MST will connect all the units in such a way that the total dissimilarity (or similarity) is minimised without forming any loops.
2. **Binary Spanning Tree (BST) for Treatment and Control**:
    - Once the MST is constructed, it can be utilised to split the units into treatment and control groups.
    - A simple method might involve choosing a starting node and alternating assignments along the tree, ensuring that connected nodes (likely similar to each other) are assigned to different groups to balance the characteristics across groups.
    - Alternatively, more complex rules might be applied to decide splits based on optimising certain balance metrics.

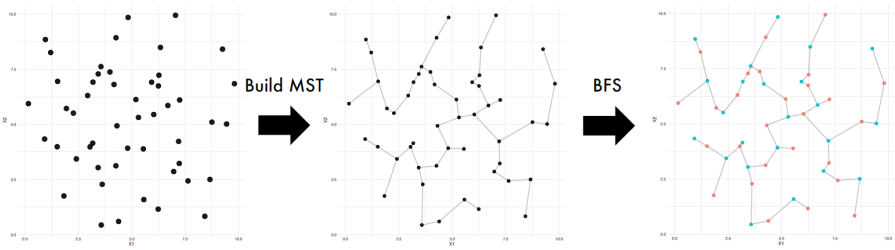

Figure 192: Using MST for experimental design: similar units (connected in the MST) are assigned to different treatment groups to ensure balance

Advantages of Using MST in Experimental Design

- **Balance and Similarity**: MSTs help in achieving a high degree of balance between treatment and control groups by leveraging the inherent structure of the data.
- **Reduction of Bias**: By carefully constructing groups that minimise internal dissimilarity, the influence of confounding variables is reduced, leading to more reliable and valid results.
- **Flexibility**: This method can be adapted to various types of data and different measures of similarity or dissimilarity, making it versatile across disciplines.

81.9 Summary: Minimum Spanning Trees

MST Summary

Problem: Find the *connected*, *acyclic* subgraph (spanning tree) with the lowest total edge cost.

Structure being leveraged:

- Fundamental cycles and fundamental cutsets
- Cut property: min-weight edge crossing any cut is in some MST
- Cycle property: max-weight edge in any cycle is in no MST

Solution: Iteratively apply:

- Red rule (cycle property): Exclude max-cost edge in cycles
- Blue rule (cut property): Include min-cost edge in cutsets

Algorithms:

- **Prim's**: Grow tree from one vertex (repeated blue rule)
- **Kruskal's**: Sort edges, add if no cycle (combines both rules)

Complexity: $O(m \log n)$ for both algorithms with standard data structures.

Problem: Find the *connected*, *acyclic* subgraph with the lowest total cost.

Structure being leveraged in MST algorithms:

- Fundamental cycles & Fundamental cutsets
- Allows for greedy steps to change one spanning tree into a 'smaller' spanning tree.

Solution: Iteratively find fundamental cycles (find a non-element of the MST) or fundamental cutsets (find an element of the MST).

82 Intractability

83 Introduction and Motivation

Throughout this course, we have studied algorithms that solve problems efficiently-greedy algorithms, divide-and-conquer strategies, dynamic programming, and graph algorithms. These represent the "easy" problems: those solvable in polynomial time. But not all problems are so tractable.

This week, we confront the limits of efficient computation. We will encounter problems for which no fast algorithm is known, and for which there is strong theoretical evidence that no fast algorithm exists. Understanding these limits is crucial: it tells us when to stop searching for an efficient exact algorithm and instead pursue approximations, heuristics, or problem restructuring.

Why Intractability Matters

Understanding computational intractability allows us to:

- Recognise when a problem is fundamentally hard (not just lacking a clever solution)
- Justify using approximation algorithms or heuristics
- Identify when problem constraints can be relaxed to enable tractability
- Understand the theoretical foundations of cryptography and security

83.1 Classifying Computational Problems

We can broadly classify problems by their computational complexity:

1. **Tractable (polynomial time):** Problems solvable in time $\mathcal{O}(n^k)$ for some constant k. These are computationally feasible even for large inputs.
2. **Intractable (exponential time):** Problems requiring time $\mathcal{O}(c^n)$ for some constant $c > 1$. These become infeasible very quickly as input size grows.

Some problems are provably exponential due to their combinatorial nature:

- **Decision problems with exhaustive search requirements:** Determining whether a program halts within k steps, or solving position-based strategy games like $n \times n$ chess or checkers. These require examining an immense number of possibilities, each affected by the sequence of moves leading to them.
- **Strategic games:** In chess or checkers, a minor change early in the game can drastically alter the outcome. Solving these often requires considering nearly all possible future moves, leading to a combinatorial explosion.

However, many problems **defy easy classification**:

- **The Halting Problem:** It is undecidable whether arbitrary programs will halt. For specific instances, it might be provable, but no general algorithm exists.

- **Problems in the "grey zone":** Many optimisation and search problems have no known polynomial-time algorithm, yet we cannot prove they require exponential time.

Implications for Algorithm Design

If a problem is known to be exponential, researchers focus on heuristic or approximate solutions rather than exact ones. Conversely, proving that a problem previously thought to be exponential can actually be solved in polynomial time would revolutionise fields like cryptography, logistics, and artificial intelligence.

84 Complexity Class P: Polynomial-Time Algorithms

Definition: Complexity Class P

The complexity class **P** (Polynomial time) consists of all decision problems that can be solved by a deterministic Turing machine in time $\mathcal{O}(n^k)$ for some constant k, where n is the size of the input.

Equivalently, a problem is in P if there exists an algorithm that:

1. Always produces the correct answer (yes/no for decision problems)
2. Runs in time bounded by a polynomial function of the input size

Class P: Key Characteristics

P represents the class of "efficiently solvable" problems:

- Running time is $\mathcal{O}(n^k)$ for some constant k
- Algorithms scale reasonably as input size increases
- Considered tractable and computationally feasible

84.1 What Makes Polynomial Time "Efficient"?

The definition of polynomial time is deliberately broad and robust:

- **Scalability:** Polynomial-time algorithms grow at a manageable rate. Doubling the input size increases running time by a factor of at most 2^k, which remains bounded.
- **Practical constants:** In practice, polynomial algorithms tend to have small constants and small degrees-$3n^2$ rather than $10^8 n^2$; $\mathcal{O}(n^2)$ rather than $\mathcal{O}(n^{100})$.
- **Contrast with exponential:** An algorithm running in 2^n time becomes infeasible even for modest inputs. For $n = 100$, $2^{100} \approx 10^{30}$ operations-far beyond any computer's capability.

84.2 Easy vs Hard: A Surprising Dichotomy

Many problems that appear similar have vastly different complexities:

In P (tractable)	Probably not in P
Shortest Path	Longest Path
Minimum Cut	Maximum Cut
2-SAT	3-SAT
Planar 4-Colouring	Planar 3-Colouring
Bipartite Vertex Cover	General Vertex Cover
Matching	3D-Matching
Primality Testing	Integer Factorisation
Linear Programming	Integer Linear Programming

This table reveals a remarkable pattern: small changes to problem definitions can shift them from tractable to (apparently) intractable. Understanding why this happens is the central goal of complexity theory.

85 Polynomial-Time Reductions

Reductions are the fundamental tool for comparing the difficulty of computational problems. The core idea is simple: if we can transform any instance of problem X into an instance of problem Y, then solving Y also solves X.

Definition: Polynomial-Time Reduction

Problem X **polynomial-time reduces** to problem Y, written $X \leq_p Y$, if arbitrary instances of X can be solved using:

1. A polynomial number of standard computational steps, plus
2. A polynomial number of calls to an oracle that solves problem Y

Equivalently, there exists a polynomial-time computable function f such that for any instance x of problem X:

$$x \text{ is a YES-instance of } X \iff f(x) \text{ is a YES-instance of } Y$$

Polynomial Reduction: Key Insight

If $X \leq_p Y$, then:

- Y **is at least as hard as** X-a polynomial-time solution to Y yields a polynomial-time solution to X
- **Contrapositive:** If X cannot be solved in polynomial time, then neither can Y

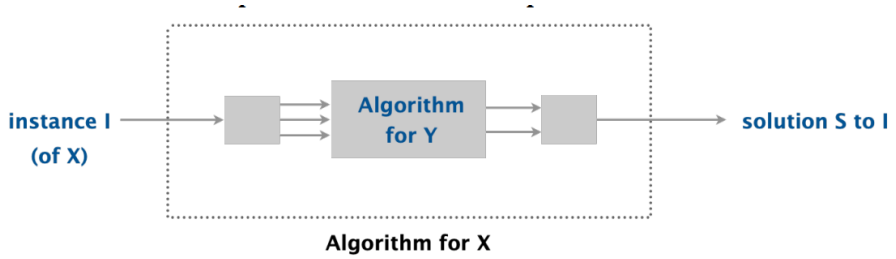

Figure 193: Polynomial-time reduction from problem X to problem Y. An instance of X is transformed (in polynomial time) into an instance of Y. The oracle solves Y, and the solution is transformed back to answer X.

85.1 The Reduction Process

A polynomial-time reduction from X to Y involves three steps:

1. **Transform the input:** Convert any instance of X into an instance of Y. This transformation must run in polynomial time.
2. **Solve the transformed problem:** Use the oracle (or algorithm) for Y to solve the transformed instance.
3. **Interpret the solution:** Convert the solution back to the context of X. This step must also run in polynomial time.

85.2 Understanding the Direction of Reductions

The notation $X \leq_p Y$ can be confusing. Let us clarify what it does and does not imply:

Common Misconceptions About Reductions

Given $X \leq_p Y$:

- **FALSE:** "If X can be solved in polynomial time, then so can Y"
  Why wrong: The reduction only goes one direction. Y could be harder than X.
- **FALSE:** "X can be solved in polynomial time if and only if Y can"
  Why wrong: This assumes bidirectional reduction. Y might have other, harder aspects not captured by X.
- **TRUE:** "If X cannot be solved in polynomial time, then neither can Y"
  Why true: If Y were polynomial-time solvable, we could solve X in polynomial time via the reduction-contradicting our assumption.
- **FALSE:** "If Y cannot be solved in polynomial time, then neither can X"
  Why wrong: X might be easier than Y and have alternative solution methods.

Intuition: Think of Y as a powerful engine and X as a machine that uses this engine. If the engine (Y) works efficiently, the machine (X) works efficiently. But a broken engine tells us nothing about whether we could build the machine differently.

85.3 Polynomial Equivalence

When reductions exist in both directions, the problems are equally hard:

Definition: Polynomial Equivalence

If both $X \leq_p Y$ and $Y \leq_p X$, then $X \equiv_p Y$ (X and Y are **polynomially equivalent**).

In this case, X can be solved in polynomial time if and only if Y can be solved in polynomial time.

Polynomial equivalence partitions problems into equivalence classes of equal difficulty.

86 Classic Reductions: Vertex Cover and Independent Set

We now demonstrate reductions through two fundamental graph problems that turn out to be polynomially equivalent.

86.1 Independent Set

Definition: Independent Set Problem

Input: A graph $G = (V, E)$ and an integer k.

Question: Does there exist a subset $S \subseteq V$ with $|S| \geq k$ such that no two vertices in S are adjacent?

An **independent set** is a set of vertices with no edges between any pair of them.

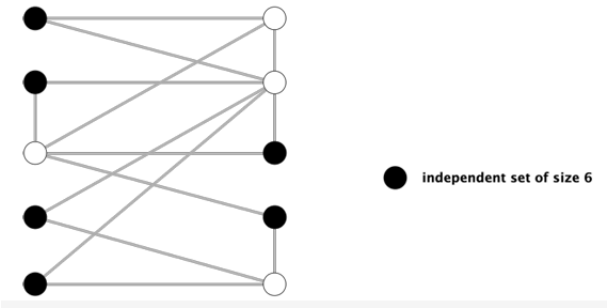

Figure 194: An independent set in a graph. The highlighted vertices form an independent set of size 6-no two highlighted vertices share an edge. This graph has an independent set of size ≥ 6 but not of size ≥ 7.

86.2 Vertex Cover

Definition: Vertex Cover Problem

Input: A graph $G = (V, E)$ and an integer k.

Question: Does there exist a subset $S \subseteq V$ with $|S| \leq k$ such that every edge in E has at least one endpoint in S?

A **vertex cover** is a set of vertices that "covers" all edges-every edge touches at least one vertex in the set.

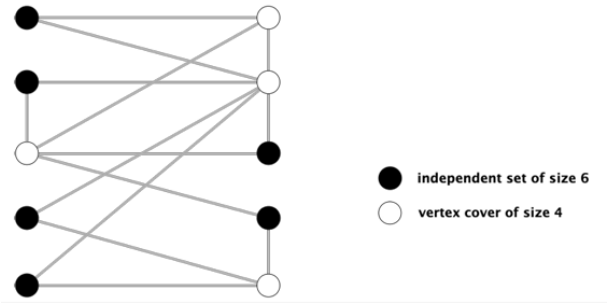

Figure 195: A vertex cover in a graph. The highlighted vertices form a vertex cover of size 4-every edge has at least one endpoint among the highlighted vertices. This graph has a vertex cover of size ≤ 4 but not of size ≤ 3.

86.3 Intuitive Comparison

Guard Analogy for Vertex Cover and Independent Set

Imagine edges as hallways and vertices as rooms:

- **Vertex Cover:** Place guards in rooms such that every hallway is watched by at least one guard. Minimise the number of guards.
- **Independent Set:** Select rooms for a party such that no two selected rooms share a hallway (guests cannot see each other). Maximise the number of rooms.

The key insight: in both problems, selected vertices must connect only to non-selected vertices.

86.4 The Complement Relationship

These problems are complements of each other:

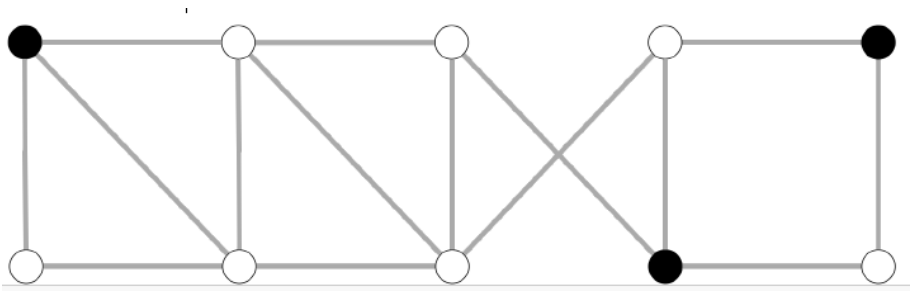

Figure 196: The complement relationship between vertex cover and independent set. In this graph, the white vertices form a vertex cover (every edge has at least one white endpoint), while the black vertices form an independent set (no two black vertices are adjacent). Together, they partition the vertex set.

Theorem: Vertex Cover $\equiv_p$ Independent Set

Let $G = (V, E)$ be a graph and let $S \subseteq V$. Then:

$$S \text{ is a vertex cover} \iff V \setminus S \text{ is an independent set}$$

Proof:

($\Rightarrow$) Suppose S is a vertex cover. Consider any edge $(u, v) \in E$. Since S covers all edges, at least one of u or v is in S. Therefore, u and v cannot both be in $V \setminus S$. Since this holds for every edge, no two vertices in $V \setminus S$ are adjacent, so $V \setminus S$ is an independent set.

($\Leftarrow$) Suppose $V \setminus S$ is an independent set. Consider any edge $(u, v) \in E$. Since $V \setminus S$ contains no adjacent vertices, at least one of u or v must be in S. Since this holds for every edge, S covers all edges, so S is a vertex cover.

Reduction: Vertex Cover $\leftrightarrow$ Independent Set

Corollary: G has a vertex cover of size $\leq k$ if and only if G has an independent set of size $\geq |V| - k$.

This gives us polynomial-time reductions in both directions:

- **Vertex Cover $\leq_p$ Independent Set:** To check if G has a vertex cover of size $\leq k$, check if G has an independent set of size $\geq |V| - k$.
- **Independent Set $\leq_p$ Vertex Cover:** To check if G has an independent set of size $\geq k$, check if G has a vertex cover of size $\leq |V| - k$.

Therefore: **Vertex Cover $\equiv_p$ Independent Set**

These are mirror problems: solving one immediately solves the other by taking the complement. The reduction is trivial-no additional computation beyond complementing the vertex set.

87 Set Cover and Its Relationship to Vertex Cover

87.1 The Set Cover Problem

Definition: Set Cover Problem

Input:

- A universal set $U = \{1, 2, \ldots, n\}$ of elements
- A collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U
- An integer k

Question: Does there exist a selection of k or fewer subsets from $\mathcal{S}$ whose union equals U?

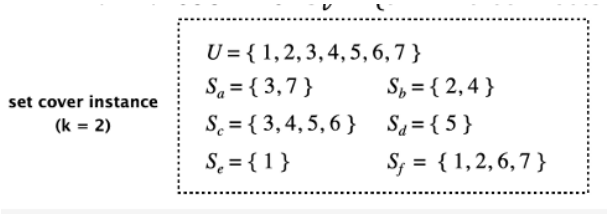

Figure 197: A Set Cover instance with $U = \{1, 2, 3, 4, 5, 6, 7\}$ and six subsets. The question asks whether we can cover all elements using $k = 2$ or fewer subsets. Here, $S_c \cup S_f = \{3, 4, 5, 6\} \cup \{1, 2, 6, 7\} = U$, so yes.

Practical example: Consider building a software system with n required capabilities. Each available software package provides some subset of these capabilities. Set Cover asks: can we achieve all n capabilities using at most k software packages?

87.2 Reduction: Vertex Cover $\leq_p$ Set Cover

We now show that Vertex Cover reduces to Set Cover, demonstrating that Set Cover is at least as hard as Vertex Cover.

Theorem: Vertex Cover $\leq_p$ Set Cover

Given a Vertex Cover instance $(G = (V, E), k)$, construct a Set Cover instance as follows:

- Universal set: $U = E$ (the set of all edges)
- For each vertex $v \in V$, create subset $S_v = \{e \in E : v \text{ is an endpoint of } e\}$
- The collection is $\mathcal{S} = \{S_v : v \in V\}$
- Use the same integer k

Claim: G has a vertex cover of size $\leq k$ if and only if the constructed Set Cover instance has a solution of size $\leq k$.

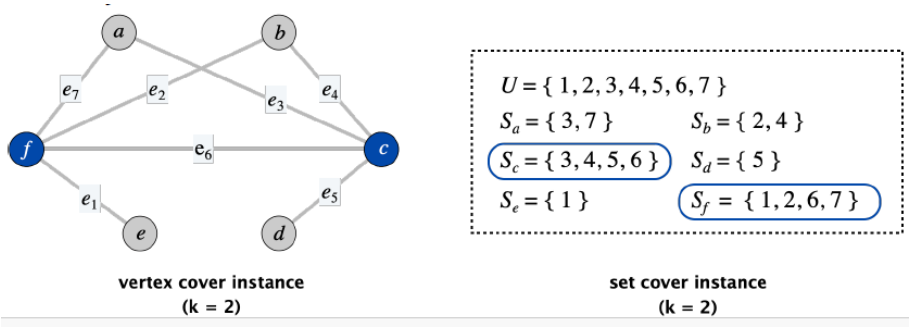

Figure 198: Reduction from Vertex Cover to Set Cover. Each vertex becomes a subset containing its incident edges. The graph on the left has vertices $\{a, b, c, d, e, f\}$ and edges $\{e_1, \ldots, e_7\}$. Selecting vertices $\{a, c\}$ as a vertex cover corresponds to selecting subsets $\{S_a, S_c\}$ in the Set Cover instance, which together cover all edges.

Proof of correctness:

($\Rightarrow$) If $V' \subseteq V$ is a vertex cover of size k, then selecting $\{S_v : v \in V'\}$ covers all edges. Each edge $e = (u, v)$ is covered by S_u or S_v (or both), since at least one endpoint is in V'.

($\Leftarrow$) If $\{S_{v_1}, \ldots, S_{v_k}\}$ covers $U = E$, then $\{v_1, \ldots, v_k\}$ is a vertex cover. Each edge e is in some S_{v_i}, meaning v_i is an endpoint of e.

Set Cover is More General

Set Cover has strictly more representational power than Vertex Cover. In the reduction:

- Each edge appears in exactly two subsets (its two endpoints)
- General Set Cover instances can have elements appearing in any number of subsets

You cannot represent an arbitrary Set Cover instance as a Vertex Cover instance. Thus Vertex Cover $\leq_p$ Set Cover, but Set Cover $\not\leq_p$ Vertex Cover (via this type of reduction).

88 Satisfiability and the 3-SAT Problem

The Boolean Satisfiability problem (SAT) holds a special place in complexity theory as the first problem proven NP-complete.

88.1 Preliminaries: Boolean Logic

Boolean Logic Terminology

- A **literal** is a Boolean variable x or its negation $\neg x$
- A **clause** is a disjunction (OR) of literals: e.g., $(x \vee \neg y \vee z)$
- A clause is satisfied if at least one of its literals is true
- A formula in **Conjunctive Normal Form (CNF)** is a conjunction (AND) of clauses:
$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (\neg x_3 \vee x_1)$$
- A CNF formula is satisfied if all its clauses are satisfied

88.2 The SAT and 3-SAT Problems

Definition: SAT and 3-SAT

SAT (Boolean Satisfiability):

Input: A Boolean formula Φ in CNF.

Question: Does there exist an assignment of truth values to the variables that makes Φ true?

3-SAT: SAT restricted to formulas where each clause contains exactly 3 literals.

Example: Consider the 3-SAT formula:

$$\Phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

Is there an assignment making Φ true? Setting $x_1 = x_2 = x_3 =$ True satisfies all three clauses.

Why 3-SAT is Hard

With n variables, there are 2^n possible truth assignments. Checking all of them requires exponential time. No polynomial-time algorithm is known for 3-SAT, and it is strongly believed that none exists.

Note: 2-SAT (each clause has exactly 2 literals) is solvable in polynomial time-a striking example of how a small change in problem definition can dramatically affect complexity.

88.3 Reduction: 3-SAT $\leq_p$ Independent Set

This reduction demonstrates a powerful technique: encoding logical constraints as graph structure.

Theorem: 3-SAT $\leq_p$ Independent Set

Given a 3-SAT formula Φ with k clauses, construct a graph G as follows:

1. For each clause, create 3 vertices (one per literal in the clause)
2. Connect all three vertices within each clause (forming a triangle)
3. Connect each literal vertex to all vertices representing its negation in other clauses

Claim: Φ is satisfiable if and only if G has an independent set of size k.

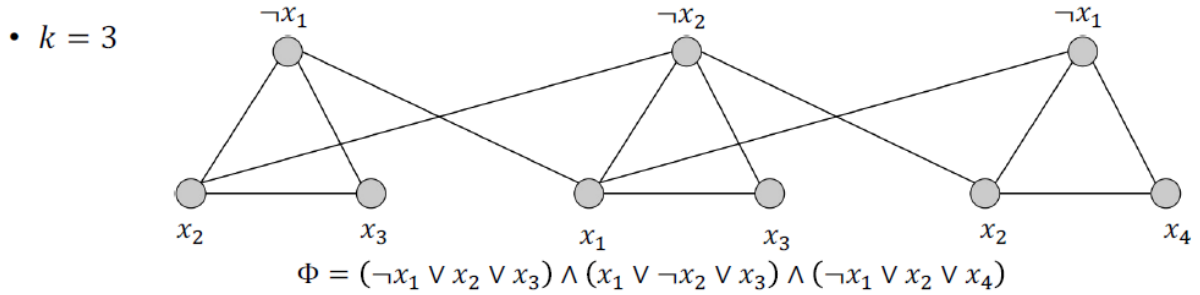

Figure 199: Reduction from 3-SAT to Independent Set. Each clause becomes a triangle of three vertices. Edges within triangles ensure we pick at most one literal per clause. Edges between contradictory literals (e.g., x_1 and $\neg x_1$) ensure consistency: if we include x_1 in our independent set, we cannot include $\neg x_1$. An independent set of size k (one vertex per clause) corresponds to a satisfying assignment.

Intuition: The independent set constraints encode logical consistency:

- Triangle edges force us to pick at most one literal per clause
- Cross-clause edges prevent picking both x and $\neg x$
- An independent set of size k picks exactly one true literal per clause-a satisfying assignment

88.4 Transitivity of Reductions

Chain of Reductions

Polynomial-time reductions are transitive: if $X \leq_p Y$ and $Y \leq_p Z$, then $X \leq_p Z$.

This gives us:

$$\text{3-SAT} \leq_p \text{Independent Set}$$
$$\text{Independent Set} \equiv_p \text{Vertex Cover}$$
$$\text{Vertex Cover} \leq_p \text{Set Cover}$$

Therefore: 3-SAT $\leq_p$ Set Cover

This chain demonstrates that Set Cover is at least as hard as 3-SAT.

89 Reduction Strategies

Having seen several reductions, we can identify common patterns:

Common Reduction Techniques

1. **Simple equivalence:** Problems that are complements or duals of each other.
   Example: Independent Set $\equiv_p$ Vertex Cover
2. **Special case to general case:** Embed a restricted problem into a more general framework.
   Example: Vertex Cover $\leq_p$ Set Cover (graphs are special cases of set systems)
3. **Gadget encoding:** Construct "gadgets"-graph or set structures that encode logical constraints.
   Example: 3-SAT $\leq_p$ Independent Set (triangles encode clause satisfaction; cross-edges encode consistency)
4. **Transitivity:** Chain existing reductions together.
   Example: 3-SAT $\leq_p$ Set Cover via Independent Set and Vertex Cover

90 Decision, Search, and Optimisation Problems

Many problems come in three flavours:

- **Decision:** Does a solution of a given quality exist? (Yes/No answer)
- **Search:** Find a solution of a given quality (if one exists)
- **Optimisation:** Find the best possible solution

Example with Vertex Cover:

- **Decision:** Does G have a vertex cover of size $\leq k$?
- **Search:** Find a vertex cover of size $\leq k$ in G
- **Optimisation:** Find a minimum vertex cover in G

Theorem: Decision $\equiv_p$ Search $\equiv_p$ Optimisation

For Vertex Cover (and many similar problems), all three versions are polynomially equivalent.

Decision $\leq_p$ Search: If we can find a vertex cover of size $\leq k$, we can certainly decide whether one exists (just check if the search succeeds).

Search $\leq_p$ Decision: Given a decision oracle, we can find a vertex cover by:

1. Use binary search to find the minimum k such that a cover of size $\leq k$ exists
2. For each vertex v: check if $G - v$ has a cover of size $\leq k - 1$
3. If yes, include v in the cover and recurse on $G - v$ with target $k - 1$
4. If no for all neighbours of some uncovered edge, backtrack

Optimisation $\leq_p$ Search: Use binary search with the search algorithm.

Search $\leq_p$ Optimisation: An optimal solution is certainly a valid solution.

This equivalence means that proving hardness for the decision version implies hardness for search and optimisation as well.

91 Complexity Class NP: Efficient Verification

91.1 Certificates and Verification

The class NP captures problems where solutions can be verified efficiently, even if finding them is hard.

Definition: Certifier

A **certifier** for a problem is an algorithm $C(x, y)$ that takes:

- An instance x of the problem
- A **certificate** (or **witness**) y

and outputs YES or NO.

The certifier is **polynomial-time** if C runs in time polynomial in $|x|$ (the certificate y may be polynomial in $|x|$ as well).

Example: Composite Numbers

- **Instance:** A large integer n
- **Certificate:** A divisor d with $1 < d < n$
- **Certifier:** Check that $n \mod d = 0$

The certifier runs in polynomial time (division is efficient), even though finding d might be hard.

91.2 The Class NP

Definition: Complexity Class NP

A decision problem is in **NP** (Nondeterministic Polynomial time) if there exists a polynomial-time certifier.

Equivalently: a problem is in NP if, for every YES-instance, there exists a certificate of polynomial size that can be verified in polynomial time.

Class NP: The Verification Perspective

A problem is in NP if:

"Given a proposed solution, I can quickly check whether it is correct."

- We do not require that solutions can be found quickly
- We only require that solutions can be verified quickly
- This creates an asymmetry: verification may be much easier than discovery

Why "Nondeterministic"? An alternative characterisation: NP consists of problems solvable in polynomial time by a nondeterministic Turing machine-one that can "guess" the right certificate and then verify it. This is equivalent to the certifier definition.

91.3 Examples of Problems in NP

Vertex Cover is in NP:

- **Certificate:** A subset $S \subseteq V$ of vertices
- **Verification:** Check that $|S| \leq k$ and every edge has an endpoint in S
- This takes $\mathcal{O}(|V| + |E|)$ time-polynomial

3-SAT is in NP:

- **Certificate:** An assignment of truth values to all variables
- **Verification:** Check that each clause has at least one true literal
- This takes $\mathcal{O}(n \cdot k)$ time where n is the number of variables and k is the number of clauses

Independent Set is in NP:

- **Certificate:** A subset $S \subseteq V$ of vertices
- **Verification:** Check that $|S| \geq k$ and no two vertices in S are adjacent
- This takes $\mathcal{O}(|S|^2)$ time in the worst case-polynomial

91.4 What is NOT in NP?

Not every problem is in NP. Consider these variants of the longest path problem:

1. **"Is the longest simple path $\geq k$?" - In NP**

   Certificate: a path of length $\geq k$. Verification: check it is simple and has length $\geq k$.

2. **"Is the longest simple path $\leq k$?" - Not obviously in NP**

   A YES answer means no path exceeds length k. There is no obvious short certificate for this-we would need to argue about all possible paths.

3. **"Find the longest simple path" - Not a decision problem**

   NP is defined for decision problems (YES/NO answers). Optimisation problems are handled separately.

NP is About YES-Instances

NP captures problems where YES-instances have short, verifiable certificates. Problems asking for non-existence ("there is no...") or universal properties ("for all...") may not be in NP.

The class **co-NP** captures problems where NO-instances have short certificates. Whether NP = co-NP is an open question.

92 P, NP, and EXP: The Complexity Landscape

92.1 The Three Main Classes

The Complexity Hierarchy

- **P:** Decision problems solvable in polynomial time
- **NP:** Decision problems verifiable in polynomial time
- **EXP:** Decision problems solvable in exponential time ($\mathcal{O}(2^{n^k})$ for some k)

92.2 Known Relationships

Complexity Class Inclusions

- **P $\subseteq$ NP**: If we can solve a problem in polynomial time, we can certainly verify solutions in polynomial time (just solve and compare).
- **NP $\subseteq$ EXP**: Any NP problem can be solved in exponential time by trying all possible certificates and verifying each one.
- **P $\neq$ EXP**: This is proven-there exist problems requiring exponential time that cannot be solved in polynomial time (e.g., certain generalised games).

92.3 The P vs NP Question

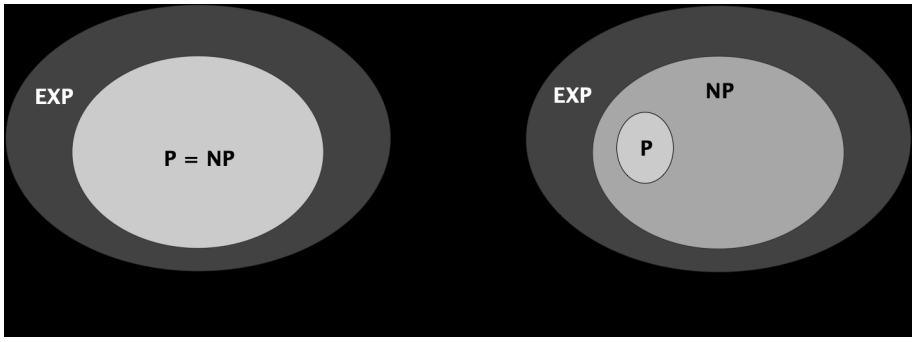

Figure 200: Two possible worlds. **Left:** If P = NP, the classes collapse-every problem with efficiently verifiable solutions also has an efficient algorithm. **Right:** If P $\neq$ NP (the prevailing belief), P is strictly contained in NP, and there exist problems that can be verified but not solved efficiently.

The P vs NP Problem

Question: Does P = NP?

Translation: Is every problem whose solutions can be verified quickly also solvable quickly?

Status: One of the seven Millennium Prize Problems. Unsolved since formally posed in 1971. Most researchers believe P $\neq$ NP, but no proof exists.

If P = NP:

- Every problem with efficiently checkable solutions would be efficiently solvable
- Modern cryptography would collapse (e.g., RSA relies on factoring being hard)
- Optimisation, scheduling, and logistics problems would become tractable
- Mathematical theorem proving could be automated

If P $\neq$ NP:

- There is a fundamental gap between finding and verifying
- Cryptography remains secure (assuming appropriate problems are hard)
- We must accept approximations and heuristics for hard problems

93 NP-Completeness: The Hardest Problems in NP

93.1 Definition

Definition: NP-Complete

A problem Y is **NP-complete** if:

1. $Y \in$ NP (solutions can be verified in polynomial time)
2. For every problem $X \in$ NP: $X \leq_p Y$ (every NP problem reduces to Y)

The second condition means Y is **NP-hard**: at least as hard as every problem in NP.

Thus: NP-complete = NP $\cap$ NP-hard.

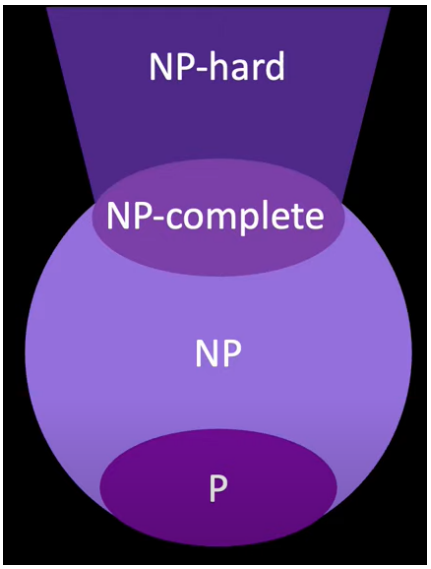

Figure 201: Relationship between complexity classes (assuming P $\neq$ NP). NP-complete problems sit at the boundary of NP-the hardest problems within NP. NP-hard problems are at least as hard as NP-complete but may lie outside NP entirely.

NP-Complete: Key Properties

If Y is NP-complete:

- Y is a "universal" hard problem-solving Y efficiently solves all of NP
- If $Y \in P$, then P = NP (every NP problem reduces to Y, so all become polynomial)
- Contrapositive: if P $\neq$ NP, then no NP-complete problem is in P

93.2 The Cook-Levin Theorem

How do we know NP-complete problems exist? The foundational result:

Cook-Levin Theorem (1971)

SAT (Boolean Satisfiability) is NP-complete.

This was proven by showing that any computation of a nondeterministic Turing machine can be encoded as a SAT formula. Thus every problem in NP reduces to SAT.

Once we have one NP-complete problem, we can prove others NP-complete via reduction:

Proving NP-Completeness

To prove problem Y is NP-complete:

1. Show $Y \in \text{NP}$ (exhibit a polynomial-time certifier)
2. Show $X \leq_p Y$ for some known NP-complete problem X

By transitivity, if X is NP-complete and $X \leq_p Y$, then every NP problem reduces to Y.

93.3 Examples of NP-Complete Problems

Thousands of problems are known to be NP-complete, spanning diverse domains:

- **Logic:** SAT, 3-SAT, Circuit-SAT
- **Graph theory:** Vertex Cover, Independent Set, Clique, Graph Colouring, Hamiltonian Path/Cycle
- **Sets and numbers:** Set Cover, Subset Sum, Partition, Bin Packing
- **Scheduling:** Job Shop Scheduling, Multiprocessor Scheduling
- **Optimisation:** Travelling Salesman (decision version), Knapsack (decision version)

The fact that such diverse problems are all equivalent (in terms of polynomial-time solvability) is remarkable and suggests a deep underlying structure.

94 NP-Hard: Beyond NP

Definition: NP-Hard

A problem Y is **NP-hard** if every problem in NP polynomial-time reduces to Y.

Note: Y need not be in NP itself. NP-hard problems may be:

- Decision problems outside NP (no efficient verification)
- Optimisation problems (not decision problems at all)
- Undecidable problems (no algorithm can solve them)

NP-Hard vs NP-Complete

- **NP-complete** = NP-hard AND in NP
- **NP-hard** = at least as hard as NP-complete, but possibly harder

Every NP-complete problem is NP-hard, but not every NP-hard problem is NP-complete.

Examples of NP-hard problems not in NP:

- **The Halting Problem:** Given a program and input, does the program halt? This is undecidable-no algorithm can solve it for all inputs. It is NP-hard (every NP problem can be reduced to it) but not in NP (not even decidable, let alone verifiable).
- **Optimisation versions:** "Find the minimum vertex cover" is NP-hard but not a decision problem. The decision version ("Is there a vertex cover of size $\leq k$?") is NP-complete.

Common Misconception

NP-hard does not mean "harder than NP-complete." The distinction is about type, not difficulty:

- NP-complete problems are the hardest problems within NP
- NP-hard problems are at least as hard as NP-complete, but may lie outside NP

94.1 The Landscape of Hard Problems

Most natural problems in NP fall into one of two categories:

1. Known to be in P (efficiently solvable)
2. Known to be NP-complete (probably not efficiently solvable)

A few problems remain in limbo, with neither polynomial algorithms nor NP-completeness proofs:

- **Integer factorisation:** Given n, find its prime factors
- **Graph isomorphism:** Given two graphs, are they structurally identical?
- **Discrete logarithm:** Given g, h, p, find x such that $g^x \equiv h \pmod{p}$

These problems are believed to be intermediate-harder than P but not NP-complete. If P $\neq$ NP, such intermediate problems must exist (Ladner's theorem).

95 Coping with NP-Complete Problems

When faced with an NP-complete problem in practice, giving up is not an option. Several strategies can yield useful results:

95.1 Approximation Algorithms

Approximation Approach

Accept a solution that is provably close to optimal, even if not optimal itself.

Example: For Vertex Cover, a simple greedy algorithm achieves a 2-approximation-it finds a cover at most twice the size of the minimum.

Approximation algorithms provide **guarantees**: the solution quality is bounded relative to optimal.

95.2 Heuristics and Metaheuristics

When guarantees are not essential, heuristics can find good solutions quickly:

- **Greedy heuristics:** Make locally optimal choices
- **Local search:** Start with a feasible solution and iteratively improve it
- **Simulated annealing:** Allow occasional worse moves to escape local optima
- **Genetic algorithms:** Evolve a population of solutions

These methods often work well in practice but offer no worst-case guarantees.

95.3 Special Cases and Problem Structure

Many NP-complete problems become tractable under restrictions:

- **Vertex Cover on trees:** Solvable in linear time (dynamic programming)
- **2-SAT:** Polynomial time (unlike 3-SAT)
- **Planar graphs:** Many problems are easier on planar graphs
- **Bounded treewidth:** Many NP-complete problems are polynomial on graphs with bounded treewidth

Practical Wisdom

Before applying general-purpose methods to an NP-complete problem:

1. Check if your specific instances have exploitable structure
2. Consider whether approximate solutions suffice
3. Evaluate the actual instance sizes you face-exponential algorithms may be acceptable for small n

95.4 Exponential Algorithms Done Well

Even when polynomial time is impossible, we can still optimise:

- **Branch and bound:** Prune the search space using bounds
- **Dynamic programming over subsets:** Often achieves $\mathcal{O}(2^n \cdot \text{poly}(n))$ rather than naive $\mathcal{O}(n!)$
- **Parameterised complexity:** Identify parameters that, when small, make the problem tractable

95.5 Connection to Machine Learning

The intractability of finding optimal solutions motivates many machine learning techniques:

- **Gradient descent** finds local optima, not global optima-but this often suffices
- **Regularisation** trades optimality for generalisation
- **Ensemble methods** combine multiple suboptimal solutions

Just as we accept local minima in neural network training, we can accept approximate solutions to NP-complete problems. The key insight: a good-enough solution found quickly often beats an optimal solution found too late.

96 Summary

Complexity Classes Summary

- **P** - Problems solvable in polynomial time ("easy to solve")
- **NP** - Problems verifiable in polynomial time ("easy to check")
- **NP-complete** - The hardest problems in NP; if any is in P, then P = NP
- **NP-hard** - At least as hard as NP-complete; may be outside NP
- **EXP** - Problems solvable in exponential time ("really hard")

Known: $\text{P} \subseteq \text{NP} \subseteq \text{EXP}$ and $\text{P} \neq \text{EXP}$.

Unknown: Whether $\text{P} = \text{NP}$ (most believe $\text{P} \neq \text{NP}$).

Practical Takeaways

1. **Recognise intractability:** If your problem is NP-complete, do not waste time seeking an efficient exact algorithm.
2. **Prove hardness via reduction:** Show $X \leq_p Y$ where X is known NP-complete to establish Y is NP-hard.
3. **Exploit structure:** Special cases may be tractable even when the general problem is not.
4. **Accept approximation:** A good solution now often beats a perfect solution never.
5. **Understand the landscape:** P, NP, NP-complete, and NP-hard form a coherent framework for reasoning about computational difficulty.

Data Structures & Algorithms: Assignment 2

Henry Baker

January 5, 2026

Question 1

Pull request URL: `https://github.com/lenafm/calculator_app/pull/12`

Question 2

Going through the algorithm step-by-step:
(NB just to define the algorithm's process I am consciously using bullet points without full sentences for clarity)

The function `check_array` does the following:

- takes a parameter (here: `input`)
- `input` is presumably a list array (given function name `check_array`)
- it iterates over every element in `input` (here: `idx`)
- if the first character of the element (`[0]`) is `1` then this function transforms the whole element to be `None`.
- It then returns the newly modified array.

Having stored values `1_3, 5_2` under the variable `original_array`, this code chunk then passes the variable to the `check_array` function. Lists in python are passed by reference and the modification is done in-place, so when we modify `input` inside the function, we make modifications at the memory location referred to by `original_array`. So `original_array` is modified in place. Line 9 also establishes another variable `new_array` that refers to the same memory location (where the changes made by the function are stored). Thus both variables now 'point' (see below) to the same modified list. The output given from this code chunk is:
`[None "5_2"]`
`[None "5_2"]`.

This behaviour is ultimately dissimilar to R and is because in python, variables that hold objects (like lists here) don't actually contain the objects themselves but references to them in memory. Thus when we pass a list to a function, we're passing a reference to that list, not a separate copy of the list (...I'm guessing this is what R does). This means that if we modify the list inside the function, the changes will reflect on the original list because both the original reference (outside the function) and the function's parameter reference (inside the function) point to the same list object in memory. This behaviour is under girded by pointers.

Considering this from the perspective of pointers:

A pointer is when a memory location is encoded and stored in memory itself; variables hold pointers to memory locations. So here we first store the function's algorithm operations in memory and assign a pointer with `check_array` variable. We then store `["1_3" "5_2"]` in memory and a pointer is held to that location by `original_array` variable. As covered above, when we pass `original_array` to the `check_array` function we are passing a reference to the list's memory location - as the function iterates over each element according to its index reference it is like passing a pointer to the array. The function then modifies the list and stores those results directly in place at that memory location. When line 9 establishes another variable, it has now set up 2 pointers pointing at the same memory location

> NB there's no opportunity for 'garbage collection' to free up memory as both `original_array` variable and `new array` variable continue pointing at their memory location so they remain active variables.

Question 3

This represents a brute-force approach. There are ways to make it more efficient. However given our definition of efficiency where *'an algorithm is efficient if its runtime is polynomials'* it is efficient.

The algorithm does as follows:

1. **line 2**: sets up an outer loop that iterates through each element of the array:
   - each selected index of the array is then the starting point of the contiguous subarray (whose sum we will be taking).
2. **line 4:** sets up an inner loop that iterates through the subarray starting from current element of outer loop (i) to the end of the array itself.
   - The variable j is the end index of the subarray being considered. This loop expands the subarray one element at a time as it iterates through the subarray.
   - it adds the value of the current element (j) to the `sum_subarray` variable (line 5).
3. **Lines 6 & 7**: checks if the sum of the currently considered subarray between i and j is the largest value found so far (line 6). If so it saves it as `max_sum` (line 7)

This therefore accumulates all the possible sums of all possible contiguous permutations of the elements in the array and saves the iteratively largest sum encountered at that point in the algorithm. The algorithm halts when the outer loop has completed (i.e., it has indexed through the entire length of the array).

To determine whether it is efficient or not, we take the worst case (we are seeking an upper bound). The outer loop: runs n times, for each of these n iterations, the inner loop runs. The inner loop depends on the position of the outer loop as it runs $n - i$ times, and i is set by the outer loop. In the first iteration ($i = 0$), the inner loop runs n times; in the second iteration ($i = 1$), it runs $n - 1$ times; in the third iteration ($i = 2$), it runs $n - 2$ times, etc., until the final iteration ($i = n - 1$), where it runs 1 time. Thus, to find the upper bound on the total operations in nested loops, we sum the iterations of each inner loop for each possible value of the outer

loop. Formally: $n+(n-1)+(n-2)+\ldots+1$. This combinatorics series can be simplified to $\frac{n\cdot(n+1)}{2}$.

In big O notation, we ignore constants to focus on the polynomials of n. In this case, to determine the order, we have an $n \times n$ element, giving us n^2. In big O notation, this becomes $O(n^2)$, denoting that its complexity quadratically increases with the size of the input array. Here we are in polynomial time where $d = 2$. An algorithm is efficient if its runtime is polynomial; thus, it is efficient.

> Nevertheless, this represents a brute force approach because there is unexploited structure here. Unless there are negative numbers, the greatest sum would simply be the very first iteration. If we wanted to be more efficient, we could take a running sum to determine the location of negative values within the array, to be able to build our sum to maximise around these negative values.

I used ChatGPT to help explain the behaviour of pointers in Q2. I initially thought it was wrong and ran the code on my laptop to get the same result. I also used ChatGPT in my CSS stylings for the flask app