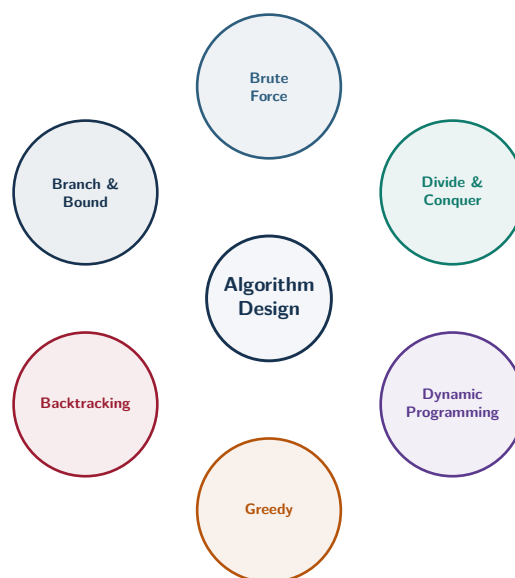


Problem Solving Strategies and Computational Approaches

Consolidated Reference Note

A single, de-duplicated treatment of the full course: computational thinking, recursion and search, time-complexity and asymptotic analysis, and the five algorithm design paradigms — brute force, divide-and-conquer, dynamic programming, greedy, backtracking, and branch-and-bound — followed by the P/NP complexity classes.

Synthesised from Handouts 1–9 and Lab Sheets 01–10, with every overlapping topic merged into one authoritative section and every quantitative lab problem worked in full.



How to use this note

What this document is

The SCS1304 material arrives as ten lecture handouts and ten lab sheets. Between them there is a great deal of repetition: binary search is introduced three separate times, best/worst/average-case analysis is developed twice, the Fibonacci sequence appears in four different chapters, and the coin-change and knapsack problems each show up under two different paradigms. This note collapses all of that into *one* linear treatment. Each idea is developed once, in the place where it makes most sense, and cross-referenced from everywhere else.

Where the original material went

Every chapter opens with a [Sources] line telling you which handouts and lab sheets it consolidates, so you can always trace a section back to the lecture slides it came from.

Consolidation map

Topic	Appeared in	Consolidated here
Computational thinking	H1	Ch. 1
Pseudocode conventions	H2	Ch. 1 §1.5
Sequential / binary search	H2, H3, H4(I), H5, LS03, LS05	Ch. 2 §2.2–2.3 (mechanics), Ch. 3 §3.4 (case analysis), Ch. 7 §7.2 (recurrence)
Fibonacci sequence	H2, H6, LS03, LS08, LS09	Ch. 2 §2.4 (definition), Ch. 8 §8.3 (DP treatment)
Recursion & its types	H2, LS04	Ch. 2 §2.5–2.8
Best/worst/average case	H3, H4(I)	Ch. 3
Big- O , Ω , Θ , o	H3, H4(I), H4(II)	Ch. 4
Loop-counting recipes	H4(I), LS05, LS06	Ch. 4 §4.7
Mathematical induction	H5	Ch. 5 §5.5
Brute force	H5, LS07, LS09	Ch. 6
Merge sort / quick sort	H5, LS07, LS08, LS09	Ch. 7 §7.3–7.4
Strassen's algorithm	H5, LS08, LS09	Ch. 7 §7.5
Maximum subarray	LS07	Ch. 7 §7.6
Dynamic programming	H6, LS08, LS09	Ch. 8
Greedy algorithms, MST	H7, LS10	Ch. 9
Knapsack (both variants)	H7	Ch. 8 §8.6 (0/1), Ch. 9 §9.6 (fractional)
Coin change	H7	Ch. 9 §9.3, contrasted with DP
Backtracking	H8	Ch. 10
Branch and bound	H9	Ch. 11
TSP by reduced cost matrix	H9	Ch. 11 §11.9
P, NP, NP-hard, NP-complete	H7, LS10	Ch. 12

How to read a chapter

Each chapter uses a consistent set of visual signals.

Definition

Formal statements you are expected to be able to reproduce verbatim in an exam.

Key idea

The single sentence that, if you remember nothing else from the section, lets you reconstruct the rest.

Worked Example 0.1 | Sample

A complete solution with every intermediate step shown. Numbers here match the lab sheets wherever possible.

Pitfall

A mistake that is easy to make and expensive to make — usually a place where two similar-looking concepts get confused.

Exam technique

How the idea is typically examined, and what markers look for.

Consolidation note

A pointer back to the original handout, or a note about how two handouts disagreed and how the disagreement was resolved.

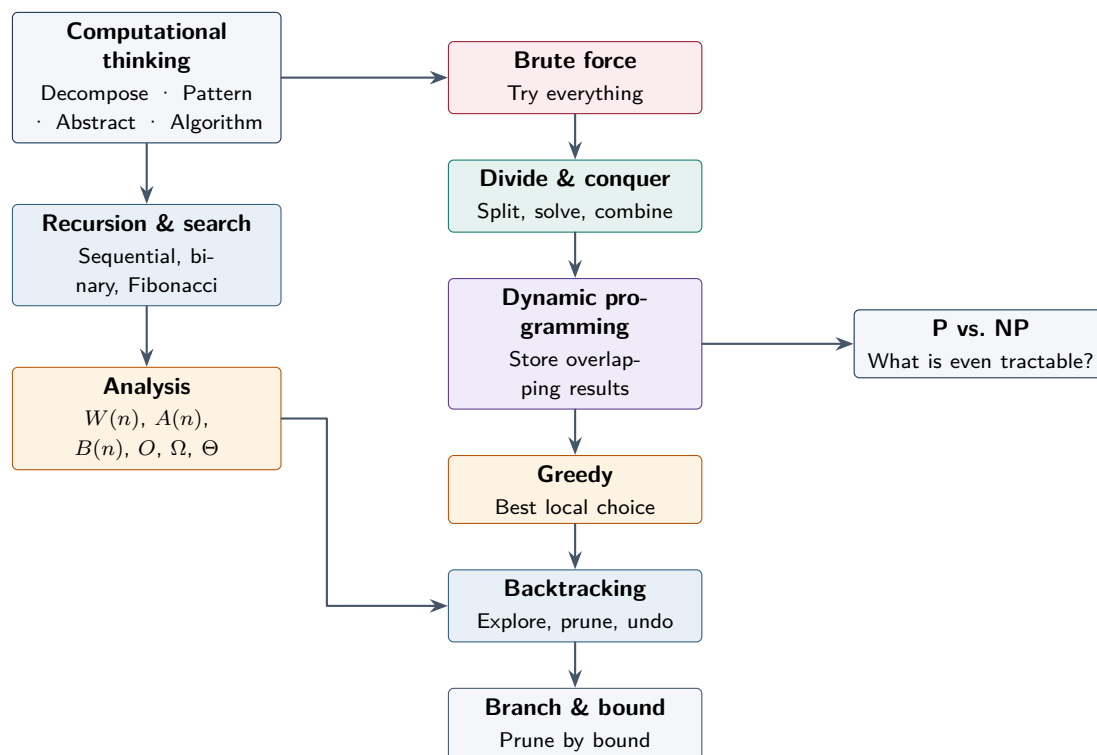
Notation used throughout

Symbol	Meaning
n	Input size — number of array elements, list length, matrix dimension, graph vertices
$T(n)$	Running time as a function of input size (the exact count of basic operations)
$W(n)$	Worst-case time complexity: maximum number of basic operations over all inputs of size n
$A(n)$	Average-case time complexity: expected number of basic operations
$B(n)$	Best-case time complexity: minimum number of basic operations
$\lg n$	$\log_2 n$. Used throughout; the base is irrelevant inside asymptotic notation
$\lfloor x \rfloor$	Floor: greatest integer $\leq x$
$\lceil x \rceil$	Ceiling: least integer $\geq x$
$S[1..n]$	Array S indexed from 1 to n (pseudocode convention — <i>not</i> 0-based)
V, E	Vertex set and edge set of a graph; $ V $ and $ E $ their sizes
$\in$	Set membership. $g(n) \in O(f(n))$ reads “ g is in big-oh of f ”

A note on “=” versus “ $\in$ ”

Handout 4 writes both $T(n) = O(n^2)$ and $g(n) \in O(n^2)$. These mean the same thing, but $\in$ is the honest notation: $O(n^2)$ is a *set of functions*, and T is a member of it. The $=$ form is an entrenched abuse of notation. Use whichever your lecturer uses, but understand it as membership — that is what makes statements like “ $n^3 \notin O(n^2)$ ” natural.

The one-page course map



Everything to the left of the paradigms is the *vocabulary* — how to describe a problem and how to measure a solution. Everything to the right is a *strategy* for producing one. The final chapter explains why, for some problems, no efficient strategy is known to exist at all.

Contents

How to use this note	i
I Foundations	1
1 Computational Thinking	2
1.1 What computational thinking is — and is not	2
1.2 The four pillars	2
1.2.1 Decomposition	2
1.2.2 Pattern recognition	4
1.2.3 Abstraction	4
1.2.4 Algorithmic thinking	5
1.3 Problem, parameter, instance, solution	6
1.4 Evaluation: the fifth, unnumbered pillar	6
1.5 Describing algorithms: pseudocode and flowcharts	7
1.5.1 Pseudocode conventions	7
1.5.2 Reference-parameter notation	7
1.5.3 Flowchart symbols	8
1.6 Where computational thinking meets this course	9
2 Searching, Sequences and Recursion	10
2.1 Why start with searching	10
2.2 Sequential (linear) search	10
2.2.1 How it behaves	11
2.3 Binary search	11
2.3.1 The idea	11
2.3.2 Iterative version	12
2.3.3 Recursive version	12
2.3.4 Sequential versus binary: the comparison count	13
2.4 The Fibonacci sequence	14
2.4.1 Why the sequence matters here	14
2.4.2 Recursive and iterative implementations	14
2.4.3 The golden ratio	15

2.5	Recursion	16
2.5.1	The two mandatory parts	16
2.5.2	Recursion and the call stack	16
2.5.3	Thinking recursively: the Matryoshka doll	17
2.6	Types of recursion	17
2.6.1	Direct versus indirect	18
2.6.2	The four kinds of direct recursion	18
2.7	Tracing recursive functions	20
2.7.1	Trace tables — for linear recursion	21
2.7.2	Trace trees — for tree recursion and for showing structure	21
2.8	Applications of recursion	23
2.9	Recursion versus iteration	24
2.9.1	When loops alone are not enough	24
2.10	Best practices and pitfalls	24
II	Analysis of Algorithms	26
3	Time Complexity: Basic Operations and the Three Cases	27
3.1	Why analyse an algorithm at all	27
3.2	The two ingredients: input size and basic operation	27
3.2.1	Input size	27
3.2.2	Basic operation	28
3.3	The three cases	29
3.3.1	Worst case — the pessimistic scenario	29
3.3.2	Average case — the realistic scenario	30
3.3.3	Best case — the optimistic scenario	31
3.3.4	Side-by-side	32
3.4	Worked case analyses: sequential search	32
3.4.1	Worst case	32
3.4.2	Best case	32
3.4.3	Average case, assuming x is definitely in the array	32
3.4.4	Average case, allowing x to be absent	33
3.5	Every-case time complexity	33
3.6	Space complexity	34
3.7	Empirical analysis versus theoretical analysis	35
3.8	Correctness — the analysis that comes first	35

4	Asymptotic Notation	37
4.1	The problem asymptotic notation solves	37
4.2	Big- O : asymptotic upper bound	37
4.2.1	Proving membership	38
4.2.2	Tight bounds	39
4.3	Big- Ω : asymptotic lower bound	40
4.4	Θ : asymptotic tight bound	41
4.5	Small- o and small- ω	42
4.5.1	The five relations at a glance	43
4.6	Do not confuse the notations with the three cases	43
4.7	Simplification and analysis rules	44
4.7.1	Simplification rules	44
4.7.2	Structural analysis rules	44
4.8	Loop analysis: a complete recipe	45
4.8.1	The eight patterns	45
4.8.2	Worked loop analyses	45
4.9	Growth rates and what they mean in practice	50
4.9.1	The standard hierarchy	50
4.9.2	Execution times: the reality check	50
4.9.3	Faster hardware does not change complexity class	51
5	Recurrence Relations and Mathematical Induction	53
5.1	Why recursive algorithms need different machinery	53
5.2	Reading a recurrence off an algorithm	54
5.3	Method 1: Iteration (repeated substitution)	54
5.4	Method 2: The recursion tree	55
5.5	Method 3: The Master Theorem	56
5.6	Method 4: Substitution (guess and verify by induction)	56
5.7	Mathematical induction	57
5.7.1	The domino principle	57
5.7.2	Worked induction proofs	58
5.7.3	Common summation formulas	59
5.8	Floor and ceiling	59
5.9	A worked catalogue of recurrences	60

III	Algorithm Design Paradigms	62
6	Brute Force	63
6.1	The idea	63
6.2	The canonical illustration: the padlock	63
6.3	Time complexities of brute-force algorithms	64
6.3.1	How fast these grow	64
6.4	Advantages and disadvantages	64
6.5	Optimising brute force	65
6.6	Alternatives to brute force	65
6.7	Applications where brute force is genuinely used	66
6.8	The travelling salesman problem	66
6.8.1	Counting the routes	67
6.9	Brute force on the maximum subarray problem	68
6.9.1	The brute-force approach	68
6.9.2	Counting the subarrays	69
7	Divide and Conquer	72
7.1	The strategy	72
7.2	Binary search revisited as divide and conquer	73
7.3	Merge sort	73
7.3.1	The idea	73
7.3.2	The merge step in detail	74
7.3.3	Complexity	75
7.3.4	Properties: stability and in-place	75
7.3.5	Advantages, disadvantages and applications	76
7.3.6	External merge sort	76
7.3.7	Worked merge sort traces	77
7.4	Quick sort	78
7.4.1	The idea	79
7.4.2	Pivot selection	80
7.4.3	Complexity	80
7.4.4	Advantages, disadvantages and applications	81
7.4.5	Worked quick sort traces	81
7.5	Strassen's matrix multiplication	83
7.5.1	The conventional algorithm	83

7.5.2	The divide-and-conquer version	84
7.5.3	Strassen's insight	84
7.5.4	Comparison and limitations	86
7.6	Divide and conquer for the maximum subarray problem	87
7.6.1	The idea	87
7.6.2	Complexity	88
7.7	Further divide-and-conquer algorithms	89
8	Dynamic Programming	91
8.1	The idea	91
8.1.1	The jigsaw analogy	91
8.2	The two conditions	91
8.2.1	Optimal substructure, illustrated	92
8.2.2	Overlapping subproblems, illustrated	92
8.3	Divide and conquer versus dynamic programming	93
8.4	The DP procedure	93
8.5	The two implementations	94
8.5.1	Memoisation (top-down)	94
8.5.2	Tabulation (bottom-up)	95
8.5.3	Memoisation versus tabulation	95
8.5.4	Dimensions of memoisation	96
8.6	Fibonacci: the complete DP treatment	96
8.7	Longest Common Subsequence	99
8.8	The 0/1 Knapsack problem	100
8.9	Coin change by dynamic programming	101
8.10	Advantages, disadvantages and applications	101
9	Greedy Algorithms	104
9.1	The idea	104
9.1.1	Historical note	104
9.2	When greedy works: the two properties	105
9.2.1	The climbing analogy	105
9.3	Characteristics of greedy algorithms	105
9.4	The coin change problem	106
9.4.1	When greedy succeeds	106
9.4.2	When greedy fails	106

9.5	The activity selection problem	107
9.6	The knapsack problems	109
9.6.1	Fractional knapsack — greedy is optimal	109
9.6.2	0/1 knapsack — greedy fails	110
9.7	Spanning trees	110
9.7.1	Applications of MSTs	111
9.8	Kruskal's algorithm	111
9.8.1	Complexity	112
9.9	Prim's algorithm	112
9.9.1	Complexity	113
9.10	Kruskal versus Prim	113
9.10.1	A complete worked comparison	113
9.10.2	Why a graph can have several MSTs	115
9.11	Greedy versus dynamic programming	116
9.12	Heuristics	116
9.13	Applications of greedy algorithms	117
9.13.1	Huffman coding in brief	117
9.14	Advantages and disadvantages	118
9.14.1	Prim's algorithm: specific advantages and disadvantages	118
10	Backtracking	120
10.1	The idea	120
10.1.1	The maze analogy	120
10.2	Backtracking versus plain recursion	121
10.2.1	The three-move cycle	121
10.3	The state space tree	121
10.4	The general algorithm	122
10.5	The N-Queens problem	123
10.5.1	The backtracking strategy	123
10.5.2	4-Queens, fully traced	123
10.6	Sudoku	125
10.7	Hamiltonian paths and cycles	125
10.7.1	Decision problem and optimisation problem	126
10.7.2	Constraints for the backtracking search	126
10.8	Graph colouring	127
10.9	The Knight's Tour	127

10.10	Optimising backtracking	128
10.11	Applications, advantages and limitations	128
11	Branch and Bound	130
11.1	The idea	130
11.1.1	Which technique for which problem	130
11.2	Backtracking versus branch and bound	130
11.3	Vocabulary	131
11.4	When to apply branch and bound	132
11.5	Basic concepts	132
11.5.1	Generating a state space tree	132
11.5.2	Evaluating a candidate solution	132
11.5.3	The bounding function	132
11.6	Types of solution representation	133
11.7	The three classifications	133
11.7.1	FIFO branch and bound	134
11.7.2	LIFO branch and bound	135
11.7.3	Least-cost branch and bound	135
11.8	Least-cost branch and bound on the TSP	136
11.8.1	The problem	136
11.9	Branch and bound on the 0/1 knapsack problem	139
11.10	Comparison of the three exhaustive techniques	140
11.11	Applications	141
IV	Complexity Classes and Reference Material	142
12	P, NP, NP-Hard and NP-Complete	143
12.1	Why this chapter exists	143
12.2	Tractable and intractable	143
12.3	The complexity classes	143
12.3.1	Deterministic and nondeterministic algorithms	144
12.4	The P versus NP question	145
12.5	Classic NP-complete and NP-hard problems	145
12.6	How NP-hard problems are handled in practice	146
12.7	Lab Sheet 10 Q6 — worked solution	146
A	Lab Sheet Worked Solutions	149

Index of lab questions	149
A.1 Lab Sheet 01 — Order-placement flowchart	151
A.2 Lab Sheet 02 — Pattern recognition and decomposition	153
A.2.1 Q1 — Examination results analysis	153
A.2.2 Q2 — Functional decomposition of a hospital appointment system	155
A.3 Lab Sheet 03 — Fibonacci growth and searching	155
A.3.1 Part 2 — Growth analysis	156
A.3.2 Part 3 — Applying search algorithms to Fibonacci data	157
A.4 Lab Sheet 05 Q4–Q5 — Basic operations in applied scenarios	158
A.5 Lab Sheet 02 Q4 and 03 Part 4 — LeetCode practice	159
B Master Reference Tables	161
B.1 Choosing a paradigm	161
B.2 The six paradigms side by side	162
B.3 Complexity of every algorithm in the course	163
B.4 Sorting algorithm properties	164
B.5 Asymptotic notation quick reference	164
B.5.1 Simplification and structural rules	164
B.5.2 Loop patterns	165
B.6 Recurrences and the Master Theorem	165
B.7 Summation formulas	166
B.8 Growth-rate reference	166
B.9 Exam checklist	167

Part I

Foundations

Computational Thinking

[Sources: Handout 1 (complete); Handout 2 §“Problem Description”; Lab Sheets 01 and 02]

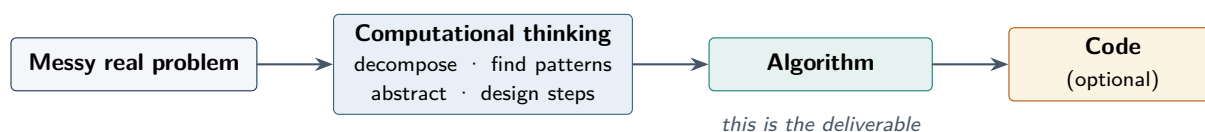
1.1 What computational thinking is — and is not

Computational thinking is a disciplined way of approaching a problem so that the end product is an **algorithm**: an unambiguous, finite sequence of steps that a human or a machine can follow to produce a required result.

The emphasis on the *end product* is what separates computational thinking from problem solving in general. Many problem-solving processes end in a decision or an insight. Computational thinking ends in a procedure — something repeatable, describable, and transferable.

Key idea

Computational thinking is not coding. Coding is one possible way of *expressing* the algorithm that computational thinking produces. The whole of SCS1304 is conducted in pseudocode and flowcharts precisely to keep the two apart.



1.2 The four pillars

Handout 1 presents four thinking processes. They are not sequential stages so much as four lenses you keep switching between, but the order below is the usual one.



1.2.1 Decomposition

Decomposition

Breaking a complex problem into smaller, more manageable sub-problems that can be understood, solved, and tested independently.

Why it works: a problem of size n that you cannot hold in your head becomes k problems of size n/k that you can. It also unlocks four practical benefits that Handout 1 stresses:

1. **Comprehension.** Smaller problems are examinable in detail.
2. **Prioritisation.** You can allocate time to sub-problems by importance and urgency rather than treating the whole as one undifferentiated lump.
3. **Delegation.** Independent sub-problems can be handed to different people — this is the basis of modular software teams.
4. **Testability.** A small part with a clear contract can be verified; a monolith can only be tried.

Worked Example 1.1 | Decomposing a criminal investigation

“Solve this crime” is not actionable. Decomposed, it becomes a set of questions each of which *is*:

- What crime was committed?
- When was it committed?
- Where was it committed?
- What physical evidence exists?
- Were there witnesses?
- Have similar crimes occurred recently?

Each is independently investigable; the answers recombine into a solution.

Worked Example 1.2 | Decomposing an online food-delivery platform

Lab Sheet 01 asks for at least six modules of a food-delivery system. A defensible decomposition, with the two things markers look for — *responsibility* and *interfaces* — made explicit:

Module	Primary responsibility	Data handled	Interacts with
User & Identity	Registration, login, session, roles	Credentials, profiles, tokens	All modules (authorisation)
Restaurant & Catalogue	Restaurant profiles, menus, availability, pricing	Menu items, prices, opening hours	Search, Cart, Order
Cart & Ordering	Cart state, order creation, cancellation window	Cart lines, order records	Catalogue, Payment, Restaurant Ops
Payment	Card and cash flows, refunds, gateway integration	Transactions, receipts	Ordering, third-party gateways
Restaurant Operations	Accept/reject, preparation status	Order status transitions	Ordering, Delivery, Notifications
Delivery & Dispatch	Rider assignment, routing, live tracking	Rider location, routes, ETAs	Restaurant Ops, Notifications
Notifications	Push/SMS/email on state change	Message queue, templates	Every state-changing module
Admin & Analytics	Monitoring, reporting, dashboards	Aggregated metrics	Read-only from all

Answering the follow-up questions. *Why decomposition matters at scale:* independent modules can be developed, tested, deployed and scaled separately — the Delivery module can be given more servers at peak dinner hours without touching Payment. *Highest load:* Delivery & Dispatch and Notifications, because they emit events continuously for the whole life of an order, whereas Payment fires once. *Most critical:* Payment — a failure there is both a revenue loss and a trust loss, and it is the module with the least tolerance for partial correctness. *If built as one component:* any change forces a full redeploy, one team blocks another, a fault anywhere takes the whole platform down, and the codebase becomes untestable in isolation.

1.2.2 Pattern recognition

Pattern recognition

Identifying similarities, differences, trends and repetitions among the sub-problems, so that one solution can be reused across many instances.

Handout 1’s contrast between square and triangular numbers is worth keeping, because it shows that patterns come in different depths.

First 5 square numbers	First 5 triangular numbers
$1 \times 1, 2 \times 2, 3 \times 3, 4 \times 4, 5 \times 5$	1; 1+2; 1+2+3; 1+2+3+4; 1+2+3+4+5
<i>One-stage pattern:</i> multiply each number by itself. A single loop suffices.	<i>Two-stage pattern:</i> first <i>generate</i> the sequence $1..k$, then <i>sum</i> it. A nested structure, or an accumulator carried across iterations.

The triangular case is the more instructive one: recognising *that* there is a pattern is not the same as recognising *what kind* of pattern it is. The computational shape of the solution follows from the second, not the first.

Pattern recognition in the lab sheets

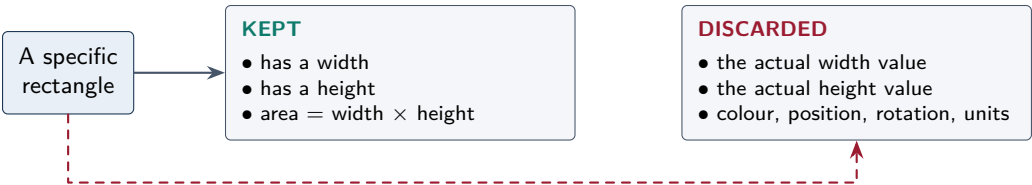
Lab Sheet 02 Q1 is a pure pattern-recognition exercise on examination data — see the fully worked solution in Appendix A, §A.2. Lab Sheet 03 does the same for Fibonacci-shaped user growth.

1.2.3 Abstraction

Abstraction

Deliberately discarding detail that is irrelevant to the problem at hand, so that what remains is a general model applicable to every instance.

The canonical example from Handout 1 is the rectangle-area module. Abstracting a rectangle for the purpose of computing area yields exactly three facts:



The discarded values are not lost — they become *parameters*. That is the whole trick: abstraction converts specific values into named inputs, which is what makes one procedure serve infinitely many instances.

Worked Example 1.3 | Abstraction in a racing game

Question. A game simulates cars on a race track. Explain how abstraction is applied.

Answer. Retain only what affects the player’s experience of driving:

- *Controls* collapse to forward/backward/steer; the clutch, gearbox, handbrake and ignition are discarded because they do not change the core loop.
- *Geometry* collapses to low-polygon models; the real car has millions of surfaces, the model needs a silhouette and a collision volume.
- *Physics* collapses to a small number of parameters (grip, mass, acceleration curve); tyre temperature, fuel consumption and engine wear are discarded — cars simply have infinite fuel.

Each discarded detail is a deliberate judgement that its absence does not change the answer to the question the system exists to answer.

Key idea

Abstraction is a road map. A road map shows roads, junctions and towns because those determine a route. It omits road width, surface material and building heights because those do not. A map that showed everything would be the territory — and useless.

1.2.4 Algorithmic thinking**Algorithmic thinking**

Expressing the solution as a finite, ordered, unambiguous sequence of steps that transforms a defined set of inputs into a defined set of outputs, and that produces the same output every time it is run on the same input.

The five properties an algorithm must have:

Property	Meaning
Finiteness	Terminates after a finite number of steps for every valid input
Definiteness	Every step is precisely and unambiguously specified
Input	Zero or more well-defined inputs
Output	One or more well-defined outputs, related to the inputs by the problem specification
Effectiveness	Every operation is basic enough to be carried out exactly, in finite time

Algorithmic thinking is not “the answer”

It is easy to slip into solving the instance instead of the problem. If asked to find the largest of {7, 3, 9}, the answer “9” is a solution to an *instance*. “Set `max` to the first element; for each remaining element, if it exceeds `max`, replace `max`” is a solution to the *problem*. Only the second is an algorithm.

1.3 Problem, parameter, instance, solution

[Handout 2 introduces this vocabulary and it is used for the rest of the course.]

The four terms

Problem A question to be answered, usually stated with unassigned variables.

Parameters

Those unassigned variables.

Instance The problem with specific values assigned to all parameters.

Solution The answer to the question asked, for a particular instance.

Worked Example 1.4 | The searching problem

Problem. Determine whether the number x is in the list S of n numbers.

Parameters. S, n, x — three of them.

Instance. $S = [10, 7, 11, 5, 13, 8]$, $n = 6$, $x = 8$.

Solution to that instance. “Yes, x is in S ” (specifically at position 6).

Algorithm. Starting with the first item of S , compare x with each item in sequence until x is found or S is exhausted.

This vocabulary matters because complexity analysis is always stated *per input size*, and “input size” is a function of the parameters. Choosing the wrong parameter as the size gives the wrong complexity.

Problem	Natural input size n
Searching / sorting an array	Number of elements
Traversing a list	Length of the list
Multiplying matrices	Number of rows and columns
Graph algorithms	Number of vertices $ V $ and edges $ E $
Arithmetic on a large integer	Number of digits (<i>not</i> the value)

1.4 Evaluation: the fifth, unnumbered pillar

Handout 1 lists evaluation alongside the four pillars, and it deserves emphasis because it is what the rest of this course is about. Once you have an algorithm, three questions remain:

1. **Is it correct?** Does it produce the right output for *every* valid input, including the edge cases — empty input, single element, all elements equal, largest possible value?
2. **Is it efficient?** How does the running time and memory grow as n grows? (Chapters 3–5.)
3. **Is it the best available?** Is there a fundamentally different strategy with better asymptotic behaviour? (Chapters 6–11.)

Key idea

The course objective, in the handout’s own words, is “to devise a solution by answering not just *a* problem, but with the *best* answer.” Evaluation is where “best” gets defined.

1.5 Describing algorithms: pseudocode and flowcharts

[Handout 2 §“Algorithmic Description”. This section is the reference for all pseudocode in the rest of the note.]

An algorithm can be described in a natural language (ambiguous), a programming language (cluttered with syntax), or pseudocode (the compromise this course uses).

1.5.1 Pseudocode conventions

The course pseudocode resembles C/C++/Java but relaxes several rules deliberately.

Feature	In C/C++/Java	In course pseudocode
Array indexing	Always starts at 0	Any integer range: $S[1..n]$, $S[2..n]$, $S[\text{low}..high]$
Array size	Fixed at compile time	Variable-sized: <code>keytype S[2..n]</code> is legal
Comparisons	<code>low <= x && x <= high</code>	$low \leq x \leq high$
Swapping	<code>temp = x; x = y; y = temp;</code>	“exchange x and y ”
Types	Concrete (<code>int</code> , <code>char</code>)	Arbitrary: <code>index</code> , <code>number</code> , <code>keytype</code> , <code>bool</code>
Control structures	<code>for</code> , <code>while</code> , <code>do</code>	Also <code>repeat (n times) { ... }</code>

1-based indexing

Handout 2 indexes arrays from 1, and every complexity formula in the course (especially $A(n) = (n+1)/2$ for sequential search) assumes it. If you silently switch to 0-based indexing while deriving a formula, your off-by-one errors will compound. Pick 1-based for exam work.

1.5.2 Reference-parameter notation

Handout 2’s search algorithms return their result through a reference parameter, written `Index &location`. In C the ampersand is the reference operator; the effect is that the caller’s variable is updated in place rather than a value being returned.

Notation reminder

```
1 procedure seqsearch(n, S, x, &location)    // last parameter is OUTPUT
```

Throughout this note, output parameters are marked `&` exactly as in the handouts, but where it improves clarity the equivalent `return` form is also given.

1.5.3 Flowchart symbols

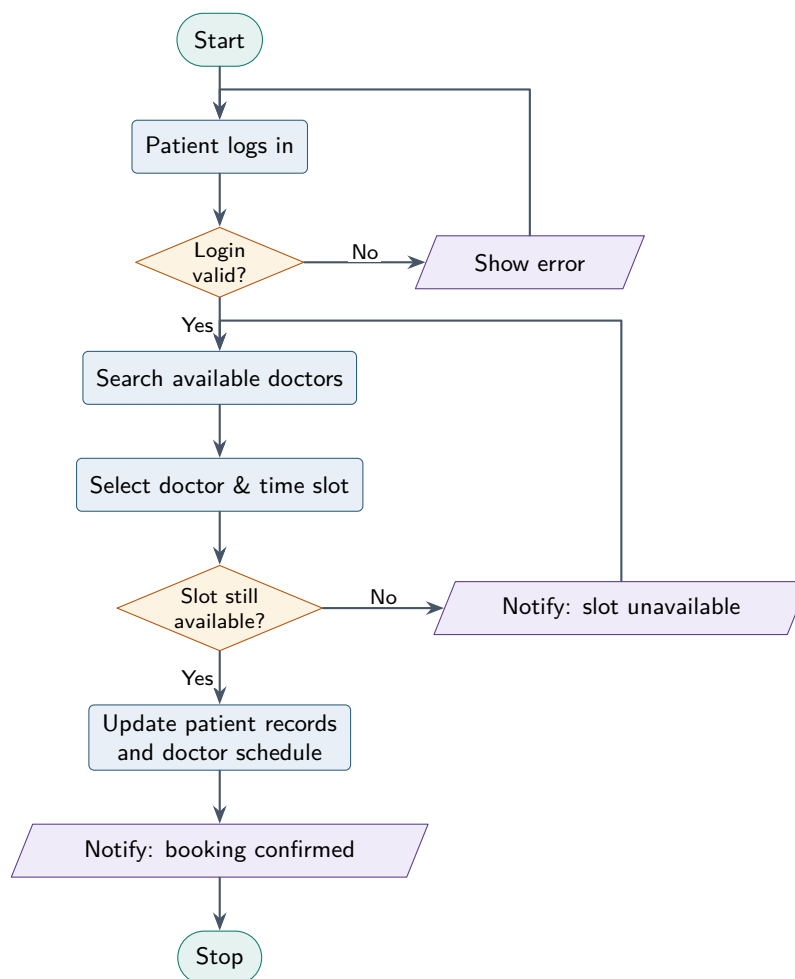
The course requires standard symbols. Lab Sheets 01 and 02 both mark on symbol correctness.



Three rules that markers check for:

1. Every decision diamond has exactly two labelled exits (*Yes/No* or *True/False*). An unlabelled exit is an error.
2. Every path eventually reaches a terminator. No dangling arrows.
3. Flow direction is top-to-bottom and left-to-right unless an arrowhead says otherwise — and loops back must always carry an arrowhead.

Worked Example 1.5 | Flowchart — “Book Appointment”



Note the two failure loops. Both return the user to a productive state rather than terminating — a flowchart that dead-ends on failure loses marks.

Exam technique

For the larger Lab Sheet 01 order-placement flowchart, the five mandatory decision points are: login failure, user cancellation before confirmation, card-payment failure, restaurant rejection, and rider unavailability. Each must be a diamond with two labelled exits, and each failure branch must have a defined destination (retry, alternative payment, refund, or cancel-with-notification). The full worked version is in Appendix A, §A.1.

1.6 Where computational thinking meets this course

The remaining chapters are, in effect, a systematic study of the fourth pillar. Chapters 3–5 supply the vocabulary for *evaluating* a designed algorithm. Chapters 6–11 supply five reusable patterns for the design step itself:

Paradigm	Which pillar it leans on	The recurring shape
Brute force	Algorithm design only	Enumerate every candidate; test each
Divide & conquer	Decomposition	Split into <i>independent</i> parts, solve, combine
Dynamic programming	Decomposition + pattern recognition	Split into <i>overlapping</i> parts; remember answers
Greedy	Pattern recognition + abstraction	Identify a local rule that is provably globally safe
Backtracking	Decomposition + abstraction	Build incrementally; abandon (prune) invalid partial solutions
Branch and bound	All four	Backtracking plus a numerical bound that prunes harder

Chapter summary

- Computational thinking produces an *algorithm*, not an answer and not code.
- Four pillars: decomposition (break it up), pattern recognition (what repeats), abstraction (what to ignore), algorithm design (the ordered steps). Evaluation is the implicit fifth.
- Vocabulary: a *problem* has *parameters*; fixing the parameters gives an *instance*; an *instance* has a *solution*. Complexity is always measured against input size, which is derived from the parameters.
- An algorithm must be finite, definite, effective, and have well-defined inputs and outputs.
- Course pseudocode: 1-based arrays, variable-sized arrays, mathematical comparisons, arbitrary types, & for output parameters.
- Flowcharts: standard symbols, two labelled exits per diamond, every path reaches a terminator.

Searching, Sequences and Recursion

[Sources: Handout 2 (complete); Handout 3 §“Best case”; Handout 5 §“Binary Search Algorithm Design”; Lab Sheets 03, 04, 05]

Consolidation

Binary search appears in three handouts: Handout 2 gives the iterative version, Handout 3 gives its best case, and Handout 5 gives the recursive version and its worst-case recurrence. All three are merged here in §2.3. The recurrence solution is deferred to Chapter 5 where the machinery for solving recurrences is developed.

2.1 Why start with searching

Searching is the smallest problem in the course that still has all the interesting parts: two genuinely different algorithms, a clean best/worst/average case split, a data precondition that changes everything (sortedness), and a natural recursive formulation. Almost every idea in Chapters 3–5 is introduced by analysing one of the two search algorithms below.

2.2 Sequential (linear) search

The sequential search problem

Problem. Is the key x in the array S of n keys?

Inputs. A positive integer n ; an array S of keys indexed 1 to n ; a key x .

Output. The location of x in S , or 0 if x is not in S .

Precondition. None. S may be in any order.

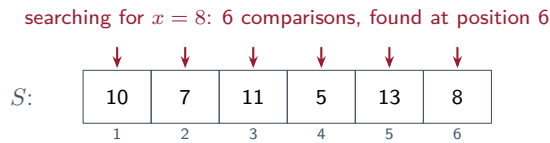
Algorithm 2.1 — Sequential Search [H2]

```

1  procedure seqsearch(int n, const keytype S[], keytype x, index &location)
2      location ← 1
3      while location ≤ n and S[location] ≠ x do
4          location ← location + 1
5      end while
6      if location > n then
7          location ← 0          // x was not found
8      end if
9  end procedure

```

2.2.1 How it behaves



The number of comparisons depends entirely on *where* x sits:

Case	When it occurs	Comparisons
Best	$x = S[1]$	1
Worst	$x = S[n]$, or $x \notin S$	n
Average (given $x \in S$)	Uniform over positions	$(n+1)/2$

The derivations of these are in Chapter 3; here we only need the shape of the answer.

Key idea

Sequential search cannot be improved without extra information about S . If S is unsorted, any correct algorithm must in the worst case inspect every element — otherwise an adversary could hide x in the one slot you skipped.

2.3 Binary search

The binary search problem

Problem. Is the key x in the array S of n keys?

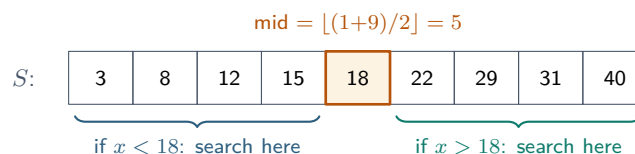
Inputs. A positive integer n ; an array S **sorted in non-decreasing order**, indexed 1 to n ; a key x .

Output. The location of x in S , or 0 if x is not in S .

Precondition. S must be sorted. This is the whole reason the algorithm works.

2.3.1 The idea

Compare x with the middle element. Because S is sorted, this single comparison eliminates *half* the remaining candidates:



2.3.2 Iterative version

Algorithm 2.2 — Binary Search, iterative [H2]

```

1  procedure binsearch(int n, const keytype S[], keytype x, index &location)
2      index low, high, mid
3      low ← 1; high ← n; location ← 0
4      while low ≤ high and location = 0 do
5          mid ← (low + high) / 2      // INTEGER division: floor((low+high)/2)
6          if x = S[mid] then
7              location ← mid
8          else if x < S[mid] then
9              high ← mid - 1
10         else
11             low ← mid + 1
12         end if
13     end while
14 end procedure

```

2.3.3 Recursive version

Algorithm 2.3 — Binary Search, recursive [H5]

```

1  index location(index low, index high)
2      index mid
3      if low > high then
4          return 0      // BASE CASE: not found
5      else
6          mid ← (low + high) / 2      // integer division
7          if x = S[mid] then
8              return mid      // BASE CASE: found
9          else if x < S[mid] then
10             return location(low, mid - 1) // recurse on left subarray
11         else
12             return location(mid + 1, high) // recurse on right subarray
13         end if
14     end if
15
16 // caller:
17 locationout ← location(1, n)

```

Here S , n and x are treated as globals so that only the changing parameters low and $high$ are passed — this is the handout’s convention and it keeps the recurrence relation clean.

Integer division and the two “mids”

$mid \leftarrow (low + high)/2$ is *integer* division, i.e. $\lfloor (low + high)/2 \rfloor$ — the **lower mid**. Some variants use $\lceil \cdot \rceil$, computed as $\lfloor (low + high + 1)/2 \rfloor$ — the **upper mid**. For $low = 0, high = 5$: lower mid = 2, upper mid = 3.

Use the lower mid for plain binary search. The upper mid is needed when searching for the *last* occurrence of a repeated key or an upper bound, where the lower mid can cause an infinite loop.

2.3.4 Sequential versus binary: the comparison count

Handout 2 tabulates the two algorithms when x is larger than every array item (the worst case for both).

Array size n	Sequential search	Binary search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

The binary column is $\lfloor \lg n \rfloor + 1$. Going from a thousand to four billion elements — a factor of four million — costs binary search twenty-two extra comparisons.

	Sequential search	Binary search
Precondition	None	Array must be sorted
Best case	1 comparison	1 comparison (item is at mid)
Worst case	n comparisons	$\lfloor \lg n \rfloor + 1$ comparisons
Average case	$\approx (n+1)/2$	$\approx \lg n$
Paradigm	Iteration / brute force	Divide and conquer
Best used when	Small n , unsorted data, one-off search	Large n , already-sorted data, many searches

Exam technique

“Is binary search always better?” — No. If the data is unsorted, you must first sort it at $O(n \lg n)$, which is worse than one $O(n)$ sequential search. Binary search wins only when the array is *already* sorted, or when the sorting cost is amortised over many searches.

Worked Example 2.1 | Search traces

(a) **Sequential search for 176 in** [105, 78, 210, 56, 132, 89, 305, 176, 44, 250].

Compare against positions 1, 2, ...: $105 \neq 176$, $78 \neq 176$, $210 \neq 176$, $56 \neq 176$, $132 \neq 176$, $89 \neq 176$, $305 \neq 176$, $176 = 176$ ✓

8 comparisons.

(b) **Binary search for 176 in the sorted version** [44, 56, 78, 89, 105, 132, 176, 210, 250, 305], $n = 10$.

Iter.	low	high	mid	$S[\text{mid}]$	Action
1	1	10	$\lfloor 11/2 \rfloor = 5$	105	$176 > 105 \Rightarrow \text{low} \leftarrow 6$
2	6	10	$\lfloor 16/2 \rfloor = 8$	210	$176 < 210 \Rightarrow \text{high} \leftarrow 7$
3	6	7	$\lfloor 13/2 \rfloor = 6$	132	$176 > 132 \Rightarrow \text{low} \leftarrow 7$
4	7	7	$\lfloor 14/2 \rfloor = 7$	176	found at position 7

4 comparisons — against 8 for the sequential search on the same data.

2.4 The Fibonacci sequence

Fibonacci sequence

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \text{ for } n \geq 2.$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

Two indexing conventions in this course

Handout 2 uses $f_0 = 0$, $f_1 = 1$. Lab Sheets 03, 08 and 09 use a shifted version — Lab Sheet 03 starts “Month 1 $\rightarrow$ 1, Month 2 $\rightarrow$ 1”, and Lab Sheets 08/09 define $F(1) = 1$, $F(2) = 2$. These generate the same sequence of *values* but attach different *indices*. **Always restate the base cases you are using before you compute.** A mis-shifted index is the single most common source of lost marks on these questions.

2.4.1 Why the sequence matters here

Fibonacci is the course’s running example for three separate lessons:

1. It is the simplest non-trivial *recurrence relation* (this chapter).
2. Its naive recursive implementation is the canonical example of *exponential blow-up from overlapping subproblems* (Chapter 8).
3. Its iterative/tabulated version is the canonical example of *dynamic programming* (Chapter 8).

2.4.2 Recursive and iterative implementations

Algorithm 2.4 — n th Fibonacci term, recursive [H2]

```

1  int fib(int n)
2    if n ≤ 1 then
3      return n
4    else
5      return fib(n-1) + fib(n-2)
6    end if

```

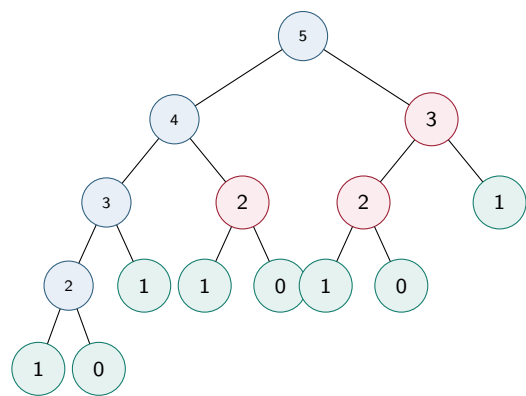
Algorithm 2.5 — n th Fibonacci term, iterative [H2]

```

1  int fib2(int n)
2    index i
3    int f[0..n]
4    f[0] ← 0
5    if n > 0 then
6      f[1] ← 1
7      for i ← 2 to n do
8        f[i] ← f[i-1] + f[i-2]
9      end for
10   end if
11   return f[n]

```

The recursion tree for `fib(5)` shows immediately why the first version is catastrophic:



Red nodes are *recomputations* of a subproblem already solved elsewhere in the tree. `fib(3)` is computed twice; `fib(2)` three times; `fib(1)` five times.

The recursive version makes $2f_{n+1} - 1$ calls in total, which grows exponentially ($\approx 1.618^n$). The iterative version makes exactly $n - 1$ additions. For $n = 50$ this is the difference between roughly 4×10^{10} operations and 49.

Consolidation note

This is the motivating example for the whole of Chapter 8. The iterative version above is literally *tabulation* — bottom-up dynamic programming — even though Handout 2 does not yet call it that.

2.4.3 The golden ratio

Golden ratio φ

$$\varphi = \frac{1 + \sqrt{5}}{2} \approx 1.6180339887 \dots, \qquad \lim_{n \rightarrow \infty} \frac{f_{n+1}}{f_n} = \varphi.$$

f_{n+1}/f_n	2/1	3/2	5/3	8/5	13/8	21/13	34/21
value	2.000	1.500	1.667	1.600	1.625	1.615	1.619

The ratio oscillates above and below φ , converging rapidly. The relevance to this course is that φ is the growth rate of the naive recursive Fibonacci — i.e. $\Theta(\varphi^n)$, an exponential-time algorithm. The appearance of φ in sunflower spirals, pine cones and nautilus shells is a pleasant aside, not examinable content.

2.5 Recursion

Recursion

A technique in which a function solves a problem by calling itself on a *smaller* instance of the same problem, until it reaches a **base case** that can be answered without further recursion.

2.5.1 The two mandatory parts

1. Base case

The smallest, simplest instance — one that can be decomposed no further and is answered directly. Typically: empty string, empty list, empty tree, zero, one.
Without it, the recursion never stops.

2. Recursive step

Decomposes a larger instance into one or more *strictly smaller* instances, solves them by recursive calls, and recombines the results.
Without progress toward the base case, the recursion never stops either.

Worked Example 2.2 | Factorial, traced

$$n! = n \times (n - 1)!, \quad 0! = 1.$$

```
factorial(5)
  = 5 * factorial(4)
    = 5 * 4 * factorial(3)
      = 5 * 4 * 3 * factorial(2)
        = 5 * 4 * 3 * 2 * factorial(1)
          = 5 * 4 * 3 * 2 * 1 * factorial(0)
            = 5 * 4 * 3 * 2 * 1 * 1 = 120
```

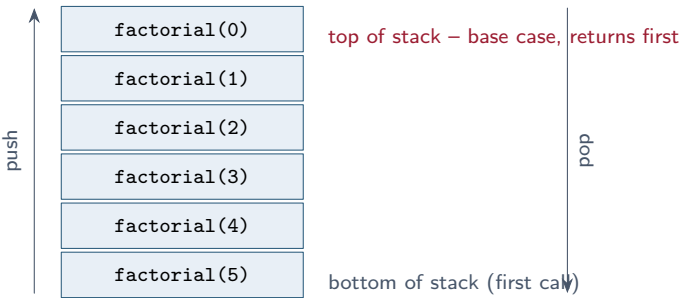
winding:

calls go down

unwinding: values return up

2.5.2 Recursion and the call stack

Each active call occupies a **stack frame** holding its parameters, local variables and return address. Frames are pushed on call and popped on return — Last In, First Out.



This is why deep recursion causes **stack overflow**: the stack has a fixed memory limit, and each frame consumes some of it.

2.5.3 Thinking recursively: the Matryoshka doll

Handout 2's nesting-doll analogy is the cleanest statement of recursive decomposition available.

Counting nested dolls [H2]

```

1  doll-count ← 0
2
3  procedure count-NestingDolls(doll)
4    if doll can be separated then
5      smaller-doll ← separate(doll)
6      increase doll-count by 1
7      count-NestingDolls(smaller-doll)    // RECURSIVE STEP
8    else
9      increase doll-count by 1           // BASE CASE: solid doll
10   end if
11 end procedure

```

Each opening makes the problem *literally* smaller, and there is a doll that cannot be opened — so termination is guaranteed. Those are exactly the two conditions for correct recursion.

Worked Example 2.3 | Loop version of the doll count

```

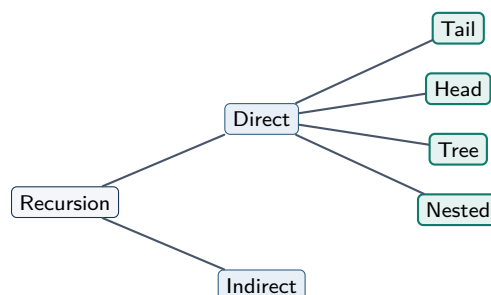
1  doll-count ← 0
2  current ← outermost-doll
3  while current can be separated do
4    doll-count ← doll-count + 1
5    current ← separate(current)
6  end while
7  doll-count ← doll-count + 1    // count the final solid doll

```

This works because the recursion is *linear* — one call per level, no branching. Linear recursion always converts to a loop mechanically. Tree recursion does not (see §2.9).

2.6 Types of recursion

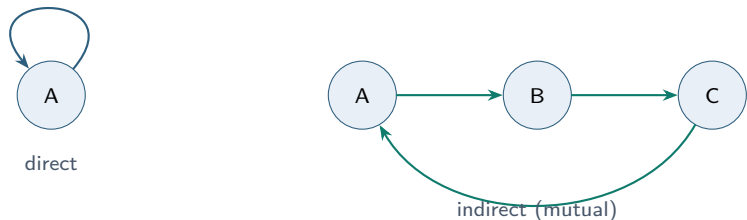
[Handout 2. This is the section most likely to be examined by definition.]



2.6.1 Direct versus indirect

Direct and indirect recursion

Direct recursion: a function calls itself explicitly within its own body.
Indirect recursion: function *A* calls *B*, which (possibly through further functions) calls *A* again, forming a cycle.



2.6.2 The four kinds of direct recursion

Type	One-line rule	Signature shape
Tail	Work happens first; the recursive call is the <i>last</i> statement executed	<code>...; return f(smaller)</code> — nothing after the call
Head	The recursive call happens <i>first</i> ; all work happens on the way back up	<code>f(smaller); ...work...</code> — nothing before the call
Tree	One call generates <i>more than one</i> recursive call	<code>return f(a) + f(b)</code>
Nested	A recursive call is used <i>as an argument</i> to another recursive call	<code>return f(f(x))</code>

Tail recursion

Tail recursion

A recursive function in which the recursive call is the last statement executed — nothing remains to be done after it returns.

Sum of first *N* naturals — tail recursive

```
1 function sum(n, accumulator)
2   if n = 0 then
3     return accumulator
4   end if
5   return sum(n - 1, accumulator + n)  // LAST statement -- nothing after
6 end function
```

Because nothing is pending, the current stack frame is no longer needed once the call is made. Compilers can therefore reuse it — **tail call optimisation** (TCO) — turning the recursion into a loop and reducing stack use from $O(n)$ to $O(1)$.

Factorial — tail recursive form

```

1  function fact(n, acc)
2    if n ≤ 1 then return acc
3    return fact(n - 1, n * acc)
4  end function
5  // fact(4, 1) -> fact(3, 4) -> fact(2, 12) -> fact(1, 24) -> 24

```

Head recursion**Head recursion**

A recursive function in which the recursive call is the first statement — no operation is performed before it. All the work is done during the unwinding phase.

Head recursion: printing 1 to n in ascending order

```

1  procedure fun(n)
2    if n > 0 then
3      fun(n - 1)      // FIRST statement -- nothing before it
4      print n         // work happens on the way back up
5    end if
6  end procedure

```

Calling `fun(3)` prints 1 2 3, not 3 2 1 — the prints happen during unwinding, so they occur in reverse order of the calls.

The loop equivalent

```

1  procedure fun_loop(n)
2    for i ← 1 to n do
3      print i
4    end for
5  end procedure

```

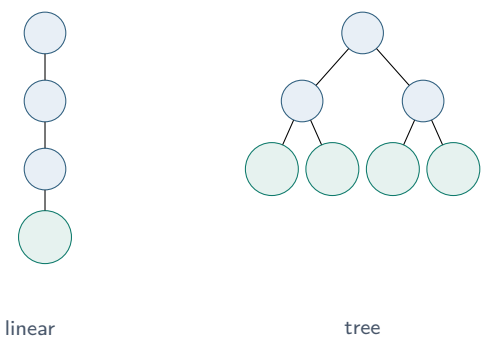
Key idea

Head vs. tail in one sentence: *tail* recursion does its work on the way *down*, *head* recursion does its work on the way *up*. That is why head recursion cannot benefit from tail-call optimisation — the frame is still needed when the call returns.

Linear versus tree recursion**Linear and tree recursion**

Linear recursion: the function makes exactly one recursive call per invocation. The call graph is a chain.

Tree recursion: the function makes more than one recursive call per invocation. The call graph branches.



`fib(n)` is the standard tree-recursive example: each call spawns `fib(n-1)` and `fib(n-2)`. Tree recursion is where exponential blow-up comes from, and where dynamic programming pays off.

Nested recursion

Nested recursion

A recursive call appears as an *argument* to another recursive call — “recursion inside recursion”.

The Ackermann-style nested pattern

```
1 function fun(n)
2   if n > 100 then
3     return n - 10
4   else
5     return fun( fun(n + 11) )    // the inner call's RESULT is the outer argument
6   end if
7 end function
```

Nested recursion is difficult to reason about because the depth of recursion is not a simple function of n — it depends on values computed *during* the recursion. It is included in the syllabus for completeness rather than for use.

Type	Recursive call position	Calls per level	Classic example
Tail	Last statement	1	Tail factorial, tail sum
Head	First statement	1	Print ascending, reverse traversal
Tree	Anywhere	≥ 2	Fibonacci, merge sort, binary tree traversal
Nested	Inside another call's arguments	varies	Ackermann function
Indirect	In another function that calls back	1 (per function)	Mutual even/odd tests

2.7 Tracing recursive functions

[Lab Sheet 04 is entirely devoted to this. Two tools are required: the trace table and the trace tree.]

2.7.1 Trace tables — for linear recursion

A trace table has one row per call, with columns for the call, the parameter values, and the return value. Fill the parameter columns downward as the calls wind, then fill the return column *upward* as they unwind.

Worked Example 2.4 | double_factorial(8)

```
1 def double_factorial(n):
2     if n ≤ 1: return 1
3     else:     return n * double_factorial(n - 2)
```

Step	Function call	<i>n</i>	Returns	Value
1	double_factorial(8)	8	8 × df(6)	8 × 48 = 384
2	double_factorial(6)	6	6 × df(4)	6 × 8 = 48
3	double_factorial(4)	4	4 × df(2)	4 × 2 = 8
4	double_factorial(2)	2	2 × df(0)	2 × 1 = 2
5	double_factorial(0)	0	base case: 1	1

Answer: double_factorial(8) = 8 × 6 × 4 × 2 × 1 = 384.

Note that rows 1–5 are written top-down as the calls are made, but the *Value* column is completed bottom-up. This is the discipline the lab sheet is testing.

2.7.2 Trace trees — for tree recursion and for showing structure

A trace tree draws one node per call, with children for each recursive call, labelled with both the call and its eventual return value.

Worked Example 2.5 | sum_squares(5)

```
1 def sum_squares(n):
2     if n == 0: return 0
3     else:     return n*n + sum_squares(n - 1)
```

sum_squares(5) = 55

↓

sum_squares(4) = 30

↓

sum_squares(3) = 14

↓

sum_squares(2) = 5

↓

sum_squares(1) = 1

↓

sum_squares(0) = 0

Unwinding: $0 \rightarrow 1^2+0 = 1 \rightarrow 2^2+1 = 5 \rightarrow 3^2+5 = 14 \rightarrow 4^2+14 = 30 \rightarrow 5^2+30 = 55$.
 Check: $1 + 4 + 9 + 16 + 25 = 55$ ✓ (and $\frac{n(n+1)(2n+1)}{6} = \frac{5 \cdot 6 \cdot 11}{6} = 55$).

Worked Example 2.6 | Bacteria growth

```

1 function bacteria_growth(P, n):
2     if n = 0:                return P
3     else if P ≥ 2000000:     return bacteria_growth(P / 2, n - 1)
4     else:                    return bacteria_growth(P * 2, n - 1)

```

Trace for `bacteria_growth(1000, 5)`:

Step	Function call	P	n	Return
1	<code>bacteria_growth(1000, 5)</code>	1000	5	<code>bg(2000, 4)</code>
2	<code>bacteria_growth(2000, 4)</code>	2000	4	<code>bg(4000, 3)</code>
3	<code>bacteria_growth(4000, 3)</code>	4000	3	<code>bg(8000, 2)</code>
4	<code>bacteria_growth(8000, 2)</code>	8000	2	<code>bg(16000, 1)</code>
5	<code>bacteria_growth(16000, 1)</code>	16000	1	<code>bg(32000, 0)</code>
6	<code>bacteria_growth(32000, 0)</code>	32000	0	32 000 (base case)

Answers to the sub-questions. (iii) Initial population: 1000. (iv) Base case: $n = 0$ — return P unchanged. (v) Recursive case, $P < 2,000,000$: double the population and decrement the hour counter: `bg(P*2, n-1)`. (vi) Recursive case, $P \geq 2,000,000$: halve the population and decrement the hour counter: `bg(P/2, n-1)`. (vii) Value returned at the base case: the current P . (viii) Result: $1000 \times 2^5 = \mathbf{32,000}$. (ix) Modified version with a floor of 10:

```

1 function bacteria_growth(P, n, minP):
2     if n = 0:                return P
3     else if P ≥ 2000000:
4         newP ← P / 2
5         if newP < minP then newP ← minP end if // clamp at the floor
6         return bacteria_growth(newP, n - 1, minP)
7     else:
8         return bacteria_growth(P * 2, n - 1, minP)

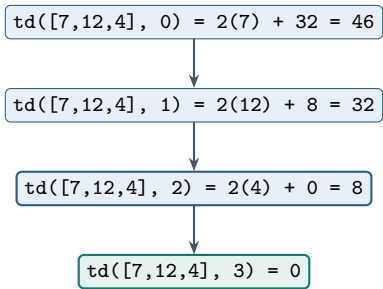
```

Trace for `bacteria_growth(1000, 10, 100)`: since P starts at 1000 and doubles, it never enters the decline branch within 10 hours: $1000 \rightarrow 2000 \rightarrow 4000 \rightarrow 8000 \rightarrow 16000 \rightarrow 32000 \rightarrow 64000 \rightarrow 128000 \rightarrow 256000 \rightarrow 512000 \rightarrow \mathbf{1,024,000}$, and $1,024,000 < 2,000,000$ throughout, so the clamp is never exercised. Result: $1000 \times 2^{10} = \mathbf{1,024,000}$.

Worked Example 2.7 | Delivery drone total distance

```
1 function total_distance(d, index):
2     if index == length(d): return 0
3     else: return 2 * d[index] + total_distance(d, index + 1)
```

Trace tree for `total_distance([7, 12, 4], 0)`:



(iv) Distances: [7, 12, 4]. (v) Base case: `index == length(d)` — past the end of the array, return 0. (vi) Recursive case: twice the current distance (out and back) plus the total for the remaining destinations. (vii) Value at base case: 0. (viii) Number of recursive calls: 3 recursive calls after the initial one (indices 1, 2, 3), i.e. 4 invocations in total. (ix) Total distance: $2(7 + 12 + 4) = 46$ metres.

2.8 Applications of recursion

Domain	Where recursion appears
Searching and sorting	Binary search, merge sort, quick sort, tree and graph traversal
Mathematics	Factorial, Fibonacci, GCD, power, combinatorial counting
Compiler design	Recursive-descent parsing of nested language constructs
Computer graphics	Fractals, the Mandelbrot set, procedural terrain, L-systems
Artificial intelligence	Recursive neural networks; game-tree search
Operating systems	Directory tree traversal; the filesystem is itself a recursive structure
Algorithm paradigms	Divide and conquer, dynamic programming, backtracking, branch and bound — all four are naturally recursive

2.9 Recursion versus iteration

	Recursion	Iteration (loops)
Mechanism	Function calls itself; the call stack tracks state	Explicit loop counter; state in variables
Memory	$O(\text{depth})$ stack frames	$O(1)$ typically
Speed	Function-call overhead per level	No call overhead
Readability	Excellent when the problem is self-similar	Excellent when the problem is a simple repetition
Code length	Usually shorter	Usually longer for tree-shaped problems
Risk	Stack overflow if depth is large	Infinite loop if the condition never fails
Debugging	Harder — state is spread across frames	Easier — state is in front of you

2.9.1 When loops alone are not enough

A loop repeats *sequentially*. A tree *branches*. Printing a directory structure is the standard illustration: at each directory you must descend into an unknown number of subdirectories, finish each of them completely, and then come back and continue — and the “come back and continue” requires remembering where you were, at every level simultaneously. Recursion gets that bookkeeping free from the call stack.

Key idea

You *can* always convert recursion to iteration — but for tree-shaped problems you must build your own explicit stack to do it. Pure loops without an auxiliary stack are not enough. Problems that are naturally recursive: nested directories, parsing mathematical expressions, Tower of Hanoi, depth-first search in graphs, and every backtracking problem in Chapter 10.

2.10 Best practices and pitfalls

Practice	Why
Identify base case(s) first	The base case is the termination condition. Write it before the recursive step.
Ensure progress	Every recursive call must move <i>strictly</i> closer to a base case; otherwise the recursion never terminates even with a valid base case.
Cover <i>all</i> base cases	Fibonacci needs two ($n = 0$ and $n = 1$). Covering only one is a common and fatal error.
Watch memory	Deep recursion consumes stack. Consider tail recursion or memoisation.
Test across input sizes	Recursive performance can differ dramatically between $n = 10$ and $n = 40$ — test both.

The three classic recursion failures

1. **Stack overflow.** Recursion depth exceeds available stack memory. Cause: base case unreachable, or depth genuinely too large for the machine.
2. **Missing or incomplete base case.** Either no base case at all, or the problem needs several and only some are covered.
3. **Redundant or incorrect recursive calls.** Arguments that do not progress toward the base case, or the same subproblem recomputed exponentially often (the Fibonacci disease — cured by memoisation, Chapter 8).

Chapter summary

- **Sequential search:** no precondition, $B(n) = 1$, $W(n) = n$, $A(n) = (n+1)/2$. Cannot be improved on unsorted data.
- **Binary search:** requires a *sorted* array; $W(n) = \lfloor \lg n \rfloor + 1$. Uses divide and conquer. Beware the two “mid” conventions.
- **Fibonacci:** $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$. Ratio tends to $\varphi \approx 1.618$. Naive recursion is $\Theta(\varphi^n)$; the iterative version is $\Theta(n)$.
- **Recursion** needs exactly two parts: a base case, and a recursive step that makes progress toward it.
- **Types:** direct (tail, head, tree, nested) and indirect. Tail = call last, work first. Head = call first, work last. Tree = more than one call. Nested = call inside a call’s arguments.
- **Tracing:** trace *table* for linear recursion (fill parameters down, returns up); trace *tree* for branching recursion.
- **Recursion vs. iteration:** recursion trades memory and speed for clarity on self-similar problems. Tree-shaped problems need an explicit stack to convert to loops.

Part II

Analysis of Algorithms

Time Complexity: Basic Operations and the Three Cases

[Sources: Handout 3 (complete); Handout 4 Part I §§1–5; Lab Sheets 05, 06]

Consolidation

Handouts 3 and 4(I) cover best/worst/average-case analysis in almost identical terms, with Handout 3 giving the motivation and Handout 4(I) giving the formal definitions and the sequential-search derivations. Both are merged here. The asymptotic *notation* (O , Ω , Θ) that both handouts start to introduce is deferred entirely to Chapter 4, so that this chapter can concentrate on exact counting.

3.1 Why analyse an algorithm at all

Three reasons, in order of practical importance:

1. **To choose.** Given two algorithms for the same problem, analysis tells you which will still be usable when the input grows.
2. **To predict.** Analysis tells you what resources to provision — how much memory, how many machines, whether a real-time deadline is achievable.
3. **To improve.** The act of analysing an algorithm usually reveals where the work is concentrated. Algorithms tend to become shorter, simpler and more elegant during analysis.

Key idea

Analysis must be *independent of the machine*. It cannot depend on CPU speed, operating system, compiler or programming language — otherwise the answer changes every time hardware changes. This is why we count *operations*, not seconds.

3.2 The two ingredients: input size and basic operation

3.2.1 Input size

Input size

A parameter n that characterises the amount of data the algorithm must process, and against which the operation count is expressed.

Setting	Input size n
Array processing	Number of elements in the array
List processing	Length of the list
Matrix operations	Number of rows and columns ($n \times m$, so two parameters)
Graph algorithms	Number of vertices $ V $ and edges $ E $
String processing	Length of the string
Numerical algorithms	Number of digits (or bits) in the input value

3.2.2 Basic operation

Basic operation

The dominant operation — the one executed most frequently, and to which the total work done by the algorithm is roughly proportional. The time complexity is defined as the number of times the basic operation is executed, as a function of n .

Algorithm	Basic operation
Searching an array	Comparison: $S[i] = \text{key}$
Sorting an array	Comparison: $S[i] > S[j]$ (sometimes: the swap)
Matrix multiplication	Multiplication: $A[i][k] * B[k][j]$
Summing array elements	Addition: $\text{sum} \leftarrow \text{sum} + S[i]$
Graph traversal	Edge examination

Worked Example 3.1 | Identifying the basic operation

```

1 for i ← 1 to n do
2   sum ← sum + A[i]
3 end for

```

Basic operation: the addition $\text{sum} + A[i]$. It executes exactly n times. Therefore $T(n) = n$, which is $O(n)$.

Note that the loop also performs n increments of i , $n+1$ comparisons of i against n , and n array accesses. We ignore all of these: they are each within a constant factor of the additions, so counting only the additions gets the growth rate right. *Choosing one representative operation is the whole point of the “basic operation” idea.*

Do not count every instruction

Students frequently try to count every assignment, comparison and increment. This is both laborious and pointless — the constant factor it produces is discarded by asymptotic notation anyway. Pick the single dominant operation and count that.

Exam technique

When a question says “identify the basic operation first” (as Lab Sheets 05 and 06 both do), the expected answer is a specific line of the given pseudocode, named explicitly. “The `PRINT` statement in the innermost loop” is a complete answer; “the loops” is not.

3.3 The three cases

For a fixed input size n , different inputs cause different amounts of work. Sequential search on ten elements takes one comparison if the key is first and ten if it is last. Three summary statistics capture this spread.

The three time-complexity functions

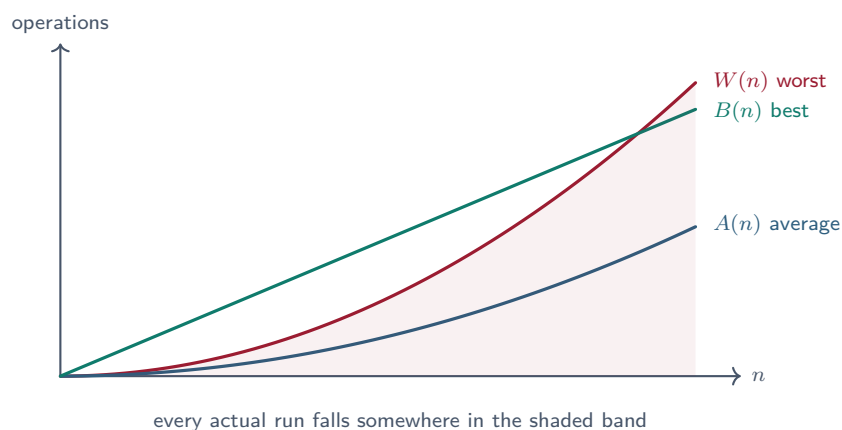
For a given algorithm and input size n :

$W(n)$ the **worst-case time complexity**: the *maximum* number of times the basic operation is performed, over all inputs of size n .

$A(n)$ the **average-case time complexity**: the *expected* number of times the basic operation is performed, given a probability distribution over inputs of size n .

$B(n)$ the **best-case time complexity**: the *minimum* number of times the basic operation is performed, over all inputs of size n .

Always $B(n) \leq A(n) \leq W(n)$.



3.3.1 Worst case — the pessimistic scenario

Key idea

$W(n)$ answers: *in the least favourable circumstances, how badly could this perform?* It gives an **upper bound** and therefore a **guarantee**.

Why it is the default choice in practice:

Reason	Explanation
Reliability	Behaviour is predictable even under adversarial input; the system stays stable.
Resource allocation	You can provision memory and processing power for the worst demand, not the typical one.
Risk management	In safety-critical systems, worst-case bottlenecks must be identified at design time, not discovered in production.
Performance guarantee	A conservative estimate is a safety net — especially valuable when the input distribution is unknown or variable.
Comparison	Gives an unambiguous, distribution-free metric for choosing between algorithms.

Algorithm	What triggers the worst case	$W(n)$
Linear search	Key is last, or absent entirely	n
Binary search	Key absent; every level halved to the end	$\lfloor \lg n \rfloor + 1$
Quick sort	Pivot is always the smallest or largest element $\Rightarrow$ maximally unbalanced partitions	n^2
Bubble sort	Array in exactly reverse order	n^2
Basic graph shortest path	Must traverse all vertices and edges	$ V ^2$

Exam technique

Lab Sheet 03 asks for three examples each of mission-critical, life-critical and financial applications where worst-case analysis matters. *Mission-critical*: air-traffic control, satellite attitude control, nuclear reactor monitoring, railway signalling. *Life-critical medical*: infusion-pump dosage control, pacemaker firmware, radiotherapy dose planning, ventilator control. *Financial*: high-frequency trading order matching, real-time fraud screening, interbank settlement (RTGS), ATM transaction authorisation. In each case a single slow response is unacceptable, so an average-case guarantee is worthless — only the worst case counts.

3.3.2 Average case — the realistic scenario

Key idea

$A(n)$ answers: *what should I expect on a typical run?* It requires a probability distribution over inputs, and is only as good as that distribution.

The procedure is always the same:

1. Enumerate all possible inputs of size n (or all distinguishable classes).
2. Assign each a probability, *using all available information*.
3. Compute the operation count for each.
4. Take the weighted sum.

Do not assume uniformity out of laziness

Handout 4(I) is emphatic here. If an array holds first names and you search for names drawn at random from a population, common names will be searched far more often than rare ones — the slots are *not* equally likely. Assuming a uniform distribution when you have information to the contrary produces a wrong answer. Assume uniformity only when you genuinely have no reason to prefer one input over another.

An average is only “typical” if the variance is small

An average can be called typical only if actual cases do not deviate much from it, i.e. only if the standard deviation is small. An algorithm that takes 1 operation half the time and $2n$ operations the other half has an average of about n — but n never actually happens.

3.3.3 Best case — the optimistic scenario**Key idea**

$B(n)$ answers: *under ideal conditions, how fast could this go?* It gives a **lower bound**, and is the least useful of the three.

Why it is misleading if used alone:

1. **Unrealistic expectations.** The best case is usually a rare, highly idealised arrangement.
2. **Not representative.** Real inputs are not arranged to satisfy it.
3. **Distracting.** Focusing on it means neglecting the common and difficult cases the algorithm will actually face.
4. **Dangerous for planning.** Provisioning resources on the best case guarantees under-provisioning.

Guaranteeing a lower bound tells you nothing useful: an algorithm with $B(n) = 1$ may still take years in the worst case.

Algorithm	What triggers the best case	$B(n)$
Linear search	Key is the first element	1
Binary search	Key is the middle element	1
Bubble sort	Array already sorted; one pass, no swaps	n
Merge sort	<i>No best case</i> — always divides and merges fully	$n \lg n$

3.3.4 Side-by-side

	Best $B(n)$	Average $A(n)$	Worst $W(n)$
Scenario	Optimistic	Realistic	Pessimistic
Provides	Lower bound	Expected behaviour	Upper bound
Typical use	Theoretical benchmark	Predicting throughput	Guarantees, provisioning
Difficulty	Easy	Hard — needs a distribution	Moderate
How often used	Rarely	Sometimes	Almost always
Main weakness	Unrepresentative	Depends on assumed distribution	Can be overly pessimistic
Asymptotic partner	Ω	Θ	O

$O/\Omega/\Theta$ are not the same as worst/best/average

The last row above reflects how this course pairs them, and it is a useful mnemonic — but it is a convention of usage, not a definition. O , Ω and Θ are bounds on *any* function; you can perfectly well write “the best case of quick sort is $\Theta(n \lg n)$ ” or “the worst case of merge sort is $\Omega(n \lg n)$ ”. See Chapter 4 §4.6.

3.4 Worked case analyses: sequential search

[Handout 4(I). These derivations are examinable in full.]

Throughout: **basic operation** = the comparison of an array item with x ; **input size** = n , the number of items in the array.

3.4.1 Worst case

The basic operation is done at most n times, which happens when x is the last item in the array, or when x is not in the array at all. Therefore

$$W(n) = n$$

3.4.2 Best case

Since $n \geq 1$ there is at least one pass through the loop. If $x = S[1]$, there is exactly one pass regardless of n . Therefore

$$B(n) = 1$$

3.4.3 Average case, assuming x is definitely in the array

Assume the items of S are distinct, and that we have no reason to prefer one slot over another — so the probability that x is in the k th slot is $1/n$ for each k with $1 \leq k \leq n$. If x is in the k th

slot, the basic operation is performed exactly k times. Hence

$$A(n) = \sum_{k=1}^n \left(k \times \frac{1}{n} \right) = \frac{1}{n} \sum_{k=1}^n k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \boxed{\frac{n+1}{2}}$$

As expected: on average about half the array is searched.

3.4.4 Average case, allowing x to be absent

Let p be the probability that x is in the array. If it is, it is again equally likely to be in any slot, so the probability that x is in the k th slot is p/n . The probability that x is not in the array at all is $1 - p$, and in that case the loop makes n passes. Therefore

$$A(n) = \sum_{k=1}^n \left(k \times \frac{p}{n} \right) + n(1 - p) = \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1 - p) = \boxed{\frac{p(n+1)}{2} + n(1 - p)}$$

Two sanity checks:

- $p = 1$ (always present): $A(n) = \frac{n+1}{2}$ — matches the previous result. ✓
- $p = \frac{1}{2}$ (present half the time): $A(n) = \frac{n+1}{4} + \frac{n}{2} = \frac{3n}{4} + \frac{1}{4}$ — about three-quarters of the array is searched on average. ✓

Exam technique

The step “ $\sum_{k=1}^n k = n(n+1)/2$ ” is used constantly in this course. Learn it, along with $\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$ and $\sum_{k=0}^n 2^k = 2^{n+1} - 1$.

3.5 Every-case time complexity

Some algorithms perform the same number of basic operations regardless of the input values — only the size matters. Their best, average and worst cases coincide.

Every-case time complexity

If $B(n) = A(n) = W(n)$ for all n , the algorithm has an **every-case time complexity**, written simply $T(n)$.

Algorithm	$T(n)$	Why every-case
Sum the elements of an array	n	Must touch every element
Find the maximum of an array	$n - 1$	Must compare every element to the current max
Matrix addition ($n \times n$)	n^2	Every cell is added exactly once
Merge sort	$n \lg n$	Always splits fully and always merges fully
Naive matrix multiplication	n^3	Three fully-executed nested loops

Key idea

Merge sort is the important case here. It has *no* best case in a useful sense: handing it an already-sorted array does not help, because the divide phase and the merge phase both run to completion regardless. That predictability is exactly what makes it attractive (Chapter 7).

3.6 Space complexity

Time is not the only resource.

Space complexity

The amount of memory an algorithm requires as a function of the input size, *excluding* the memory occupied by the input itself. This excess is called **auxiliary space**.

Algorithm	Auxiliary space	Why
Sequential / binary search (iterative)	$O(1)$	A handful of index variables
Binary search (recursive)	$O(\lg n)$	One stack frame per level of recursion
Merge sort	$O(n)$	Temporary array needed for merging
Quick sort	$O(\lg n)$ average, $O(n)$ worst	Recursion stack only; partitioning is in place
Bubble / insertion / selection sort	$O(1)$	In-place swaps
Dynamic programming (tabulation)	$O(n)$ or $O(nW)$	The DP table

In-place algorithm

An algorithm that transforms its input using only a small, *constant* amount of additional storage. Quick sort is in place; merge sort is not.

Exam technique

Why in-place matters: memory-constrained systems (embedded devices, IoT sensors); very large datasets where copying is prohibitively expensive; and cache performance — an in-place algorithm keeps working on the same memory and so benefits from cache locality, which is a large part of why quick sort beats merge sort in practice despite the worse worst case.

3.7 Empirical analysis versus theoretical analysis

	Theoretical (complexity) analysis	Empirical analysis
Method	Count basic operations as a function of n	Run the code on real data and measure
Depends on	Nothing but the algorithm	CPU, memory, OS, compiler, language, data
Tells you	How performance <i>scales</i>	How fast it <i>is</i> , here, today
Strength	Machine-independent, predictive, valid for all n	Captures constant factors and cache effects that theory hides
Weakness	Hides constants that can dominate for small n	Not transferable to other machines or larger n

The two are complements, not alternatives. Complexity analysis tells you that $O(n \lg n)$ will beat $O(n^2)$ eventually; empirical analysis tells you where “eventually” begins — which for small n is frequently never.

3.8 Correctness — the analysis that comes first

Handout 4(I) lists correctness alongside complexity and empirical analysis, and it belongs first: a fast wrong answer is worthless.

Check	What to verify
Termination	Every loop makes progress toward its exit condition; every recursion approaches a base case
Edge cases	$n = 0$, $n = 1$, all elements equal, key absent, key at each boundary, maximum representable values
Loop invariant	A property true before the loop, preserved by each iteration, and which implies correctness at exit
Precondition	Any assumption the algorithm makes (e.g. “ S is sorted” for binary search) is actually guaranteed

Chapter summary

- Analysis counts **basic operations** as a function of **input size**, so that the result is independent of hardware, OS and language.
- Pick *one* dominant operation. Do not count everything.
- $W(n)$ (max), $A(n)$ (expected), $B(n)$ (min), with $B \leq A \leq W$. $W(n)$ is the default because it gives a guarantee.
- $A(n)$ requires a probability distribution and is only as trustworthy as that distribution. An average is “typical” only when the variance is small.
- $B(n)$ is a theoretical benchmark; using it alone is misleading.
- **Sequential search:** $B(n) = 1$, $W(n) = n$, $A(n) = \frac{n+1}{2}$ (if present), $A(n) = \frac{p(n+1)}{2} +$

$n(1 - p)$ (if present with probability p).

- **Every-case complexity** $T(n)$ exists when the three coincide — e.g. merge sort, matrix addition, finding a maximum.
- **Space complexity** counts auxiliary memory. In-place means $O(1)$ auxiliary space.

Asymptotic Notation

[Sources: Handout 3 §“Popular Notations”; Handout 4 Part I §§6–12; Handout 4 Part II (complete); Lab Sheets 05, 06]

4.1 The problem asymptotic notation solves

Chapter 3 produced exact counts such as $W(n) = n$ and $A(n) = \frac{n+1}{2}$. Exact counts are unwieldy and, worse, misleadingly precise: they depend on implementation choices that do not affect how the algorithm *scales*. Consider

$$0.1n^2 + n + 100.$$

n	$0.1n^2$	n	100	total
10	10	10	100	120
20	40	20	100	160
50	250	50	100	400
100	1 000	100	100	1 200
1 000	100 000	1 000	100	101 100

At $n = 10$ the constant dominates. By $n = 1000$ the quadratic term is 99% of the total. This function is not a pure quadratic, but eventually it *behaves like* one, and for classification purposes that is all that matters.

Key idea

Asymptotic notation describes the behaviour of a function as $n \rightarrow \infty$, ignoring constant factors and lower-order terms. It answers “how does this scale?” and deliberately refuses to answer “how fast is this?”

Asymptote

An asymptote of a curve is a line the curve approaches arbitrarily closely but never touches. *Asymptotic* analysis is analysis of the tail of the graph — the region where n is very large.

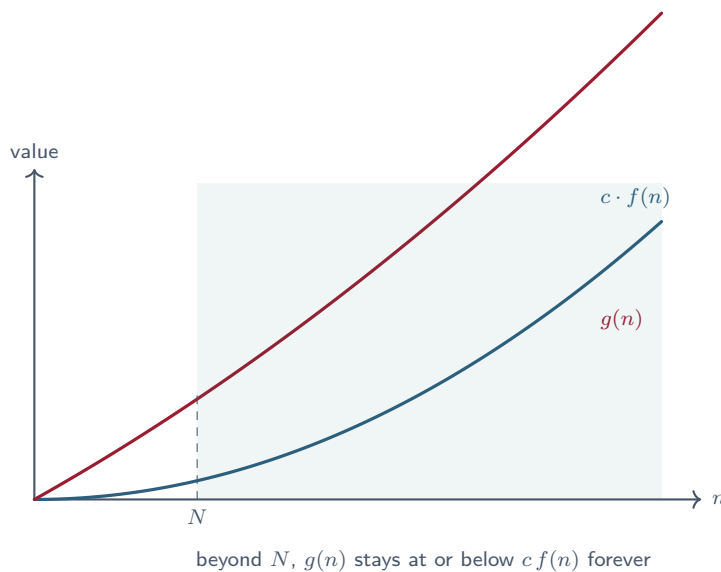
4.2 Big- O : asymptotic upper bound

Big- O

For a given complexity function $f(n)$, $O(f(n))$ is the *set* of complexity functions $g(n)$ for which there exist **some** positive real constant c and **some** non-negative integer N such that, for all $n \geq N$,

$$g(n) \leq c \cdot f(n).$$

We write $g(n) \in O(f(n))$ and say “ $g(n)$ is big-oh of $f(n)$ ”.



Three things about this definition deserve emphasis:

1. **“Some” c and N , not unique ones.** You need only exhibit one pair that works. Different students will produce different valid pairs.
2. **“For all $n \geq N$ ”, not “for all n ”.** g may start out above cf ; all that matters is that it eventually falls below and stays there.
3. **$O(f(n))$ is a set.** $g(n) \in O(f(n))$ is a membership claim.

4.2.1 Proving membership

Worked Example 4.1 | $n^2 + 10n \in O(n^2)$

Attempt 1. Take $c = 11$, $N = 1$. For all $n \geq 1$ we have $10n \leq 10n^2$, so

$$n^2 + 10n \leq n^2 + 10n^2 = 11n^2. \quad \checkmark$$

Attempt 2. Take $c = 2$, $N = 10$. For all $n \geq 10$ we have $10n \leq n^2$ (since $n \geq 10$), so

$$n^2 + 10n \leq n^2 + n^2 = 2n^2. \quad \checkmark$$

Both are correct. Neither is “more right”. The definition says *some* c and *some* N ; producing one pair is a complete proof.

Worked Example 4.2 | $5n^2 \in O(n^2)$

Take $c = 5$, $N = 0$. For all $n \geq 0$, $5n^2 \leq 5n^2$. $\checkmark$

Worked Example 4.3 | $T(n) = \frac{n(n-1)}{2} \in O(n^2)$

For all $n \geq 0$,

$$\frac{n(n-1)}{2} \leq \frac{n^2}{2}.$$

Take $c = \frac{1}{2}$ and $N = 0$. ✓

Worked Example 4.4 | $n \in O(n^2)$

Take $c = 1$, $N = 1$. For all $n \geq 1$, $n \leq 1 \cdot n^2$. ✓

This is true but *loose* — see §4.2.2.

Worked Example 4.5 | $n^3 \notin O(n^2)$ — proof by contradiction

Suppose $n^3 \in O(n^2)$. Then there exist $c > 0$ and $N \geq 0$ with

$$n^3 \leq cn^2 \quad \text{for all } n \geq N.$$

Divide both sides by n^2 (positive for $n \geq 1$):

$$n \leq c \quad \text{for all } n \geq N.$$

But c is a fixed constant and n grows without bound, so this fails for any $n > \max(N, c)$. Contradiction. Therefore $n^3 \notin O(n^2)$. ■

Exam technique

The “divide both sides and derive a contradiction” template above is the standard way to prove a *non-membership*. Learn its three lines: assume membership, divide by the smaller function, observe that a variable cannot be bounded by a constant.

4.2.2 Tight bounds

All of the following are true:

$$3n^2 + 5n + 2 \in O(n^2), \quad 3n^2 + 5n + 2 \in O(n^3), \quad 3n^2 + 5n + 2 \in O(n^4), \quad \dots$$

because any larger function is also a valid upper bound. But only the first is **tight**, and by convention that is the one we state.

Key idea

Always report the *tightest* (smallest, closest) upper bound you can justify. “This algorithm is $O(2^n)$ ” is technically true of merge sort and completely useless.

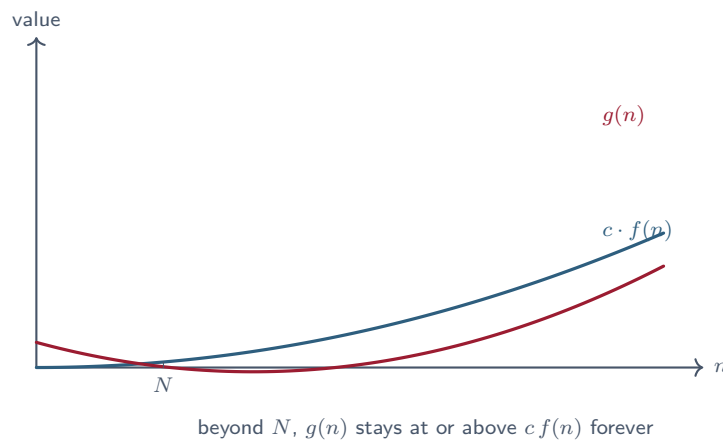
4.3 Big-Ω: asymptotic lower bound

Big-Ω

For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity functions $g(n)$ for which there exist some positive real constant c and some non-negative integer N such that, for all $n \geq N$,

$$g(n) \geq c \cdot f(n).$$

The mirror image of big- O : g may start below cf , but eventually rises above it and stays there. If $g(n)$ is the time complexity of an algorithm, f being a lower bound means the algorithm can never run *faster* than f in the long run.



Worked Example 4.6 | $n^2 + 10n \in \Omega(n^2)$

Take $c = 1$, $N = 0$. For all $n \geq 0$, $n^2 + 10n \geq n^2$. ✓

Worked Example 4.7 | $T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$

For $n \geq 2$ we have $n - 1 \geq n/2$, so

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \cdot \frac{n}{2} = \frac{1}{4}n^2.$$

Take $c = \frac{1}{4}$, $N = 2$. ✓

Worked Example 4.8 | $n^3 \in \Omega(n^2)$

For all $n \geq 1$, $n^3 \geq 1 \cdot n^2$. Take $c = 1$, $N = 1$. ✓

Worked Example 4.9 | $n \notin \Omega(n^2)$ — proof by contradiction

Suppose $n \in \Omega(n^2)$. Then there exist $c > 0$, $N \geq 0$ with $n \geq cn^2$ for all $n \geq N$. Divide both sides by cn :

$$\frac{1}{c} \geq n \quad \text{for all } n \geq N,$$

which is impossible since n is unbounded. Contradiction. ■

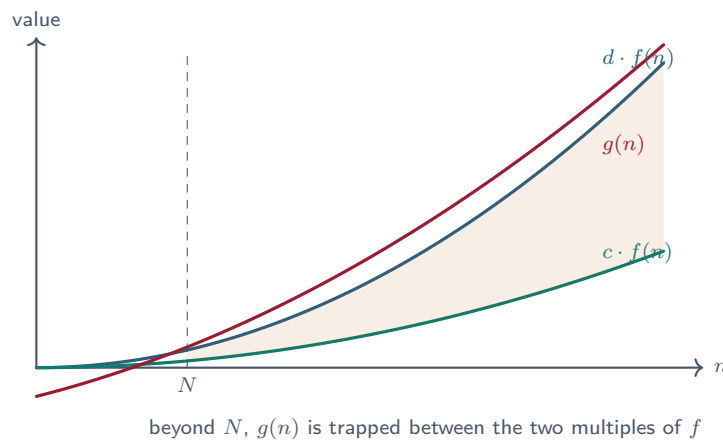
4.4 Θ : asymptotic tight bound

Big- Θ

For a given complexity function $f(n)$, $\Theta(f(n))$ is the set of complexity functions $g(n)$ for which there exist positive real constants c and d and a non-negative integer N such that, for all $n \geq N$,

$$c \cdot f(n) \leq g(n) \leq d \cdot f(n).$$

Equivalently: $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

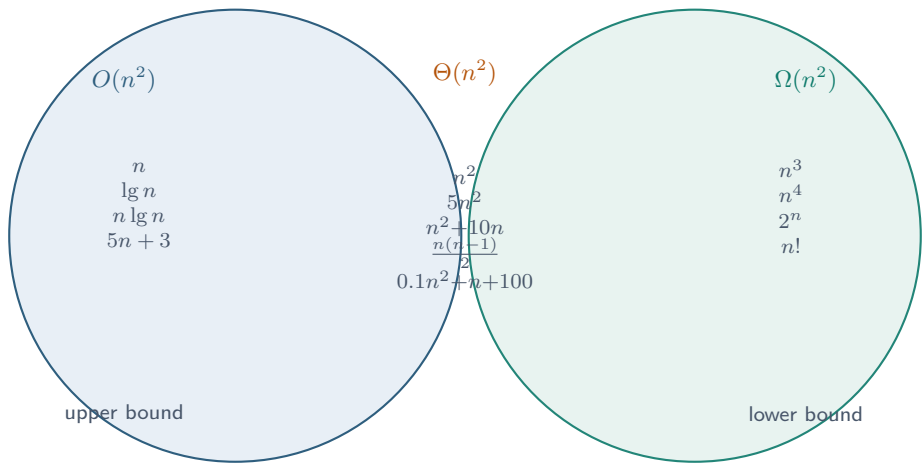


Worked Example 4.10 | $T(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$

From the two worked examples above, $T(n) \in O(n^2)$ with $c = \frac{1}{2}$, and $T(n) \in \Omega(n^2)$ with $c = \frac{1}{4}$, $N = 2$. Combining, for all $n \geq 2$:

$$\frac{1}{4}n^2 \leq \frac{n(n-1)}{2} \leq \frac{1}{2}n^2.$$

So $T(n) \in \Theta(n^2)$ with $c = \frac{1}{4}$, $d = \frac{1}{2}$, $N = 2$. ✓



4.5 Small-*o* and small- ω

Small-*o*

$o(f(n))$ is the set of complexity functions $g(n)$ satisfying: for **every** positive real constant c there exists a non-negative integer N such that, for all $n \geq N$,

$$g(n) \leq c \cdot f(n).$$

"Some c " versus "every c " — the entire difference

	Big- O	Small- o
Quantifier	There exists <i>some</i> $c > 0 \dots$	For <i>every</i> $c > 0 \dots$
Meaning	g grows no faster than f (up to a constant)	g grows <i>strictly</i> slower than f
Analogue	$\leq$	$<$
Tightness	May be tight	Never tight — the bound is guaranteed loose
Example	$2n^2 + 5 \in O(n^2)$	$n \in o(n^2)$
Ratio test	$g(n)/f(n)$ stays bounded	$g(n)/f(n) \rightarrow 0$

Little- o is the *stronger* statement: it asserts a much larger gap between the growth rates.

Worked Example 4.11 | $n \in o(n^2)$

Let $c > 0$ be arbitrary. We need N such that $n \leq cn^2$ for all $n \geq N$. Dividing by cn : $\frac{1}{c} \leq n$. So it suffices to take any $N \geq 1/c$.
Concretely, if $c = 0.00001$ then $N = 100,000$ works: for all $n \geq 100,000$, $n \leq 0.00001n^2$. Since such an N exists for *every* c , $n \in o(n^2)$. ✓

Worked Example 4.12 | $n \notin o(5n)$ — proof by contradiction

Suppose $n \in o(5n)$. Then the defining property must hold for *every* positive c — in particular for $c = \frac{1}{6}$. That would require

$$n \leq \frac{1}{6} \cdot 5n = \frac{5}{6}n \quad \text{for all } n \geq N,$$

which is false for every $n > 0$. Contradiction. ■

The moral: n and $5n$ differ only by a constant factor, so neither grows strictly faster than the other. Little- o correctly refuses to separate them.

Small- ω

Dually, $g(n) \in \omega(f(n))$ if for every positive c there exists N such that $g(n) \geq cf(n)$ for all $n \geq N$. It is the strict version of Ω , i.e. the analogue of $>$.

4.5.1 The five relations at a glance

Notation	Numeric analogue	Meaning	Limit test ($n \rightarrow \infty$)
$g \in O(f)$	$g \leq f$	g grows no faster than f	$\lim g/f < \infty$
$g \in o(f)$	$g < f$	g grows strictly slower	$\lim g/f = 0$
$g \in \Theta(f)$	$g = f$	same growth rate	$0 < \lim g/f < \infty$
$g \in \Omega(f)$	$g \geq f$	g grows no slower than f	$\lim g/f > 0$
$g \in \omega(f)$	$g > f$	g grows strictly faster	$\lim g/f = \infty$

The analogy is imperfect

Unlike numbers, functions are not totally ordered: there exist f and g where none of the five relations holds (e.g. $f(n) = n$ and $g(n) = n^{1+\sin n}$). This never arises in this course, but it is why the notation exists as set membership rather than as an ordering.

4.6 Do not confuse the notations with the three cases

Handouts 3 and 4(II) both present the pairing “ Ω = best case, Θ = average case, O = worst case”. This is a useful mnemonic for how the notations are *usually deployed* in this course, but it is not what they mean.

The mnemonic (how they are used)	The reality (what they mean)
O describes the worst case	O is an <i>upper bound on any function</i> .
Ω describes the best case	Ω is a <i>lower bound on any function</i> .
Θ describes the average case	Θ means the upper and lower bounds <i>coincide</i> .

Both of these statements are correct and both are commonly written:

- “Merge sort is $\Theta(n \lg n)$ in *all* cases” — Θ applied to the worst case.

- “Quick sort’s best case is $\Omega(n \lg n)$ ” — Ω applied to the best case, as the handouts use it.
- “Quick sort’s worst case is $O(n^2)$ ” — O applied to the worst case.

Exam technique

Follow the handouts’ convention in exam answers (Ω /best, Θ /average, O /worst), since that is what is being marked — but understand that the case and the bound are independent choices. If a question asks you to *define* O , give the formal “there exist c and N ” definition, not “the worst case”.

4.7 Simplification and analysis rules

4.7.1 Simplification rules

	Rule	Example
1	Drop constant factors	$5n \Rightarrow O(n)$; $\frac{n^2}{3} \Rightarrow O(n^2)$
2	Drop lower-order terms	$n^2 + n + 10 \Rightarrow O(n^2)$
3	Keep only the fastest-growing term	$5n^3 + n + 4 \Rightarrow O(n^3)$
4	The base of a logarithm is irrelevant	$O(\log_2 n) = O(\log_{10} n) = O(\ln n) = O(\lg n)$

Rule 4 holds because $\log_a n = \frac{\log_b n}{\log_b a}$, and $\frac{1}{\log_b a}$ is a constant — which rule 1 then discards. Mathematics defaults to base e ; computing conventionally uses base 2 and writes $\lg n$. Inside asymptotic notation it makes no difference.

Rules 1–3 apply to *addition*, not *multiplication*

$n \cdot \lg n$ cannot be simplified to $O(n)$ or $O(\lg n)$. Dropping non-dominant terms is a rule about *sums*. When complexities are *multiplied* — as they are for nested loops — both factors are kept. $O(n \lg n)$ is a genuine, irreducible complexity class sitting strictly between $O(n)$ and $O(n^2)$.

4.7.2 Structural analysis rules

Structure	Rule	Example
Sequential statements	Add , keep the dominant term	A $O(n)$ loop followed by an $O(n^2)$ loop is $O(n) + O(n^2) = O(n^2)$
Nested loops	Multiply	Two nested loops each running n times: $O(n) \times O(n) = O(n^2)$
Consecutive loops	Add	Two separate loops each running n times: $O(n) + O(n) = O(n)$
if/else	Take the maximum of the two branches	if branch $O(n)$, else branch $O(n^2) \Rightarrow O(n^2)$ worst case
Function call	Substitute the function’s own complexity	A $O(\lg n)$ call inside an n -iteration loop $\Rightarrow O(n \lg n)$

4.8 Loop analysis: a complete recipe

[This section merges the loop patterns from Handout 4(I) with the exercises of Lab Sheets 05 and 06. It is the single most heavily examined part of the course.]

Key idea

Two questions answer every loop-complexity problem: (1) *How many times does each loop run, as a function of n ?* (2) *Are the loops nested (multiply) or consecutive (add)?*

4.8.1 The eight patterns

	Pattern	Complexity	Why
1	Simple statement, no loop or recursion	$O(1)$	Fixed work
2	for i <- 1 to n	$O(n)$	n iterations
3	for i <- 0 to n step c (constant c)	$O(n)$	n/c iterations; constant discarded
4	for i <- 1 to n: i <- i*2 (or i/2)	$O(\lg n)$	Index multiplied/divided by a constant $\Rightarrow \lg n$ iterations
5	Two nested loops, both n	$O(n^2)$	$n \times n$
6	Nested loops of sizes n and m	$O(nm)$	Independent bounds multiply
7	Outer n , inner $\lg n$ (or vice versa)	$O(n \lg n)$	Multiply
8	Triangular: for i<-1 to n { for j<-1 to i }	$O(n^2)$	$\sum_{i=1}^n i = \frac{n(n+1)}{2}$

Pattern 8 is the classic trap

A triangular double loop performs only about *half* the work of a square one — $\frac{n(n+1)}{2}$ instead of n^2 . But $\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$, and rules 1–2 discard both the $\frac{1}{2}$ and the $\frac{n}{2}$. So the answer is still $O(n^2)$. Halving the work does *not* change the complexity class.

4.8.2 Worked loop analyses

Worked Example 4.13 | Lab Sheet 05 Q1 — all four snippets

(a) Triple nested loop.

```

1  FOR i FROM 0 TO n-1
2    FOR j FROM 0 TO n-1
3      FOR k FROM 0 TO n-1
4        PRINT i, j, k

```

Basic operation: the PRINT. Each loop runs n times and they are nested, so the count is $n \times n \times n = n^3$. $O(n^3)$.

(b) Two consecutive loops.

```

1  FOR i FROM 1 TO n
2    PRINT i
3  FOR j FROM 1 TO n^2
4    PRINT j

```

Basic operation: the PRINT. The loops are *consecutive*, so *add*: $n + n^2$. Drop the lower-order term. $O(n^2)$.

(c) Nested accumulation.

```

1  sum = 0
2  FOR i FROM 0 TO n-1
3    FOR j FROM 0 TO n-1
4      sum = sum + 1

```

Basic operation: the addition. Executed $n \times n = n^2$ times. $O(n^2)$.

(d) Halving loop.

```

1  i = n
2  WHILE i > 1
3    PRINT i
4    i = i / 2

```

Basic operation: the PRINT. Values of i : $n, n/2, n/4, \dots, 2$. After k iterations $i = n/2^k$; the loop stops when $n/2^k \leq 1$, i.e. $k \geq \lg n$. $O(\lg n)$.

Worked Example 4.14 | Lab Sheet 06 Part A — all six snippets

(1) Triangular loop.

```

1  FOR i FROM 0 TO n-1
2    FOR j FROM 0 TO i
3      PRINT i, j

```

Inner loop runs $i + 1$ times for each i , so the total is $\sum_{i=0}^{n-1} (i + 1) = \sum_{k=1}^n k = \frac{n(n+1)}{2}$. $O(n^2)$.

(2) Three independent bounds.

```

1  FOR i FROM 0 TO n-1
2    FOR j FROM 0 TO m-1
3      FOR k FROM 0 TO p-1
4        PRINT i, j, k

```

$n \times m \times p$ iterations. $O(nmp)$.

(3) Conditional inner loop.

```

1  FOR i FROM 0 TO n-1
2    FOR j FROM 0 TO i
3      IF (i % 2 == 0) PRINT i, j
4      ELSE FOR k FROM 0 TO j: PRINT i, j, k

```

For *even* i : constant work per (i, j) pair, contributing $O(n^2)$ overall. For *odd* i : the innermost loop runs $j + 1$ times, so that i contributes $\sum_{j=0}^i (j + 1) = \Theta(i^2)$. Summing over the odd $i < n$: $\sum \Theta(i^2) = \Theta(n^3)$. The odd branch dominates. $O(n^3)$.

(4) Doubling outer, linear inner.

```

1  i = 1
2  WHILE i < n
3    FOR j FROM 0 TO i: PRINT i, j
4    i = i * 2

```

The outer loop takes $i = 1, 2, 4, \dots, 2^k$ with $2^k < n$. The inner loop runs about i times for each. Total work

$$1 + 2 + 4 + \dots + 2^k = 2^{k+1} - 1 < 2n.$$

This is a geometric series, so the *last* term dominates the whole sum. $O(n)$.

Pitfall

It is tempting to say “outer is $\lg n$, inner is n , so $O(n \lg n)$ ”. That is wrong: the inner loop only reaches n iterations on the *final* pass. Whenever a loop bound doubles, sum the geometric series rather than multiplying by the maximum.

(5) Divisibility-filtered triple loop.

```

1  FOR i FROM 1 TO n
2    FOR j FROM 1 TO i*i
3      IF (j % i == 0)
4        FOR k FROM 0 TO j: PRINT i, j, k

```

For a fixed i , the middle loop runs i^2 times but the guard $j \% i == 0$ passes only for $j = i, 2i, \dots, i^2$ — that is i values. For each such j the innermost loop runs about j times, so the innermost work for this i is

$$\sum_{t=1}^i (ti) = i \cdot \frac{i(i+1)}{2} = \Theta(i^3).$$

The middle loop’s own iterations cost $\Theta(i^2)$, which is dominated. Summing over i :

$$\sum_{i=1}^n \Theta(i^3) = \Theta(n^4). \quad O(n^4).$$

(6) Harmonic inner loop.

```

1  FOR i FROM 0 TO n-1
2    k = i
3    WHILE k < n
4      PRINT i, k
5      k = k + i + 1

```

For a given i , k starts at i and increases by $i + 1$ each step, so it runs about $\frac{n-i}{i+1}$ times. Summing:

$$\sum_{i=0}^{n-1} \frac{n-i}{i+1} \leq n \sum_{i=0}^{n-1} \frac{1}{i+1} = n \cdot H_n = \Theta(n \ln n).$$

$O(n \lg n)$. (The harmonic number $H_n = \sum_{k=1}^n 1/k \approx \ln n$.)

Worked Example 4.15 | Lab Sheet 06 Part E — add_them

```

1  int add_them(int n, int A[])
2    index i, j, k
3    j ← 0
4    for i ← 1 to n do j ← j + A[i]      // loop 1
5    k ← 1
6    for i ← 1 to n do k ← k + k        // loop 2
7    return j + k

```

(1) Output for $n = 5$, $A = [2, 5, 3, 7, 8]$.

Loop 1 sums the array: $j = 2 + 5 + 3 + 7 + 8 = 25$.

Loop 2 doubles k each iteration starting from 1: $k = 2, 4, 8, 16, 32$, so $k = 2^5 = 32$.

Return $j + k = 25 + 32 = 57$.

(2) Time complexity. Both loops execute exactly n times and are *consecutive*, so $T(n) = n + n = 2n$, giving $\Theta(n)$ — a linear-time algorithm.

(3) An improved version. The second loop computes 2^n by repeated doubling. If exponentiation is available, or by using fast exponentiation by squaring, the whole loop collapses:

```

1  int add_them_v2(int n, int A[])
2    j ← 0
3    for i ← 1 to n do j ← j + A[i]      // still O(n) -- unavoidable
4    k ← power(2, n)                     // O(1) if hardware shift; O(lg n)
        by squaring
5    return j + k

```

The summation loop cannot be improved — every element must be read at least once, so $\Omega(n)$ is a lower bound for the problem. The result is still $\Theta(n)$ overall, but with roughly half the operations. This is a constant-factor improvement, not a complexity-class improvement — which is exactly the distinction the question is testing.

Worked Example 4.16 | Lab Sheet 06 Part D — classifying a polynomial

$$f(n) = 3n^2 + 10n \log n + 1000n + 4 \log n + 9999.$$

Growth-rate ordering of the terms: $\log n \prec n \prec n \log n \prec n^2$. The fastest-growing term is $3n^2$. Discarding the constant factor, $f(n) \in \Theta(n^2)$ — **option (e)**.

Worked Example 4.17 | Lab Sheet 06 Part B — three algorithms from scratch**1. Largest number in a list of n numbers.**

```

1 function findMax(n, S[1..n])
2   max ← S[1]
3   for i ← 2 to n do
4     if S[i] > max then max ← S[i]
5   end for
6   return max

```

Basic operation: the comparison $S[i] > \text{max}$. It executes exactly $n - 1$ times for *every* input — this is an every-case complexity. $T(n) = n - 1 \in \Theta(n)$.

2. The m smallest numbers in a list of n numbers.

```

1 function mSmallest(n, S[1..n], m)
2   for i ← 1 to m do                      // m passes
3     minIndex ← i
4     for j ← i+1 to n do                  // scan the unsorted remainder
5       if S[j] < S[minIndex] then minIndex ← j
6     end for
7     exchange S[i] and S[minIndex]
8   end for
9   return S[1..m]

```

Basic operation: the comparison $S[j] < S[\text{minIndex}]$. Pass i performs $n - i$ comparisons, so

$$W(n, m) = \sum_{i=1}^m (n - i) = mn - \frac{m(m+1)}{2} \in O(mn).$$

When m is a constant this is $\Theta(n)$; when $m = n$ it degenerates to selection sort at $\Theta(n^2)$.

3. All 3-element subsets of a set of n elements.

```

1 procedure allTriples(n, S[1..n])
2   for i ← 1 to n-2 do
3     for j ← i+1 to n-1 do
4       for k ← j+1 to n do
5         print S[i], S[j], S[k]
6       end for
8     end for

```

Basic operation: the print. The strict inequalities $i < j < k$ mean each subset is generated

exactly once, so the count is

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{6} \in \Theta(n^3).$$

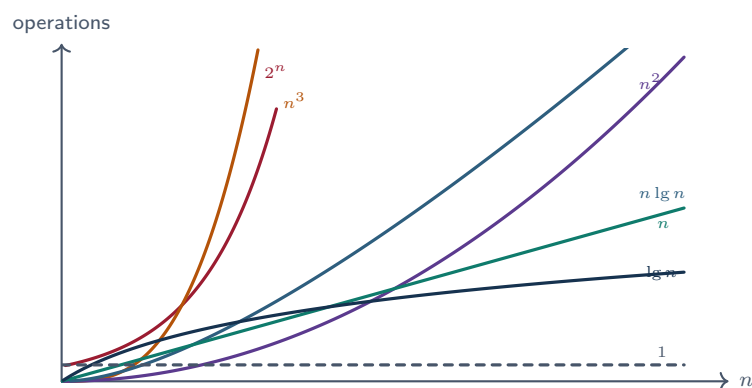
This is again an every-case complexity, and it is optimal: any algorithm that must *output* $\binom{n}{3}$ subsets needs at least that many steps.

4.9 Growth rates and what they mean in practice

4.9.1 The standard hierarchy

$$O(1) \prec O(\lg n) \prec O(n) \prec O(n \lg n) \prec O(n^2) \prec O(n^3) \prec O(2^n) \prec O(n!)$$

Class	Name	Behaviour	Typical algorithm
$O(1)$	Constant	Independent of n	Array index; stack push/pop; hash lookup
$O(\lg n)$	Logarithmic	Barely grows even for huge n	Binary search; balanced-BST lookup
$O(n)$	Linear	Doubling n doubles the work	Sequential search; array sum
$O(n \lg n)$	Linearithmic	Slightly worse than linear	Merge sort, quick sort (avg.), heap sort
$O(n^2)$	Quadratic	Doubling n quadruples the work	Bubble, insertion, selection sort; all-pairs comparison
$O(n^3)$	Cubic	Doubling n multiplies work by 8	Naive matrix multiplication; Floyd–Warshall
$O(2^n)$	Exponential	Each extra element doubles the work	Subset enumeration; naive Fibonacci
$O(n!)$	Factorial	Unusable beyond $n \approx 12$	Permutation generation; brute-force TSP



4.9.2 Execution times: the reality check

Assume one basic operation takes 1 nanosecond (10^{-9} s).

n	$\lg n$	n	$n \lg n$	n^2	2^n
10	3 ns	10 ns	33 ns	100 ns	1 μ s
100	7 ns	100 ns	664 ns	10 μ s	4×10^{13} yr
1,000	10 ns	1 μ s	10 μ s	1 ms	—
10^6	20 ns	1 ms	20 ms	17 min	—
10^9	30 ns	1 s	30 s	32 yr	—

Key idea

This table is the whole argument for studying algorithm design. At $n = 10^9$ the difference between $O(n \lg n)$ and $O(n^2)$ is the difference between half a minute and a human lifetime — on the same hardware. No amount of faster hardware closes that gap, because the gap grows with n .

4.9.3 Faster hardware does not change complexity class**Worked Example 4.18 | Lab Sheet 06 Part C — the effect of a 1000× faster machine**

An old computer solves an instance of size $n = 1000$ in one minute. A new computer is 1000 times faster. What size instance can the new machine handle in one minute?

Let $n_{\text{old}} = 1000$ and let n_{new} be the new size. In one minute the new machine does 1000× as many operations, so we need $T(n_{\text{new}}) = 1000 \cdot T(n_{\text{old}})$.

(a) $T(n) = n$.

$$n_{\text{new}} = 1000 \times 1000 = \mathbf{1,000,000}.$$

A 1000× speed-up buys a 1000× larger instance.

(b) $T(n) = n^3$.

$$n_{\text{new}}^3 = 1000 \cdot 1000^3 \Rightarrow n_{\text{new}} = 1000 \cdot \sqrt[3]{1000} = 1000 \times 10 = \mathbf{10,000}.$$

A 1000× speed-up buys only a 10× larger instance.

(c) $T(n) = 10^n$.

$$10^{n_{\text{new}}} = 1000 \cdot 10^{1000} = 10^3 \cdot 10^{1000} = 10^{1003} \Rightarrow n_{\text{new}} = \mathbf{1003}.$$

A 1000× speed-up buys *three* extra elements.

$T(n)$	Old size	New size	Gain
n	1 000	1 000 000	×1000
n^3	1 000	10 000	×10
10^n	1 000	1 003	+3

Key idea

This is the most important single result in the chapter. **Hardware improvements give multiplicative gains; algorithmic improvements give gains that grow with n .** For an exponential-time algorithm, a thousandfold faster machine is worth almost nothing — you must change the algorithm.

Worked Example 4.19 | Lab Sheet 05 Q6 — choosing an algorithm for a growing system

A student information system currently holds 1 000 records, expected to grow substantially. Three candidates: A is $O(n^2)$, B is $O(n \lg n)$, C is $O(\lg n)$.

(A) Ranking for large n (most to least efficient): $C \prec B \prec A$, i.e. $O(\lg n)$ is best, then $O(n \lg n)$, then $O(n^2)$.

(B) Growth of A versus B. A's curve is a parabola — doubling n quadruples the work. B grows only slightly faster than linearly, because $\lg n$ increases by 1 each time n doubles. The vertical gap between them widens without limit as n increases.

(C) At n from 1 000 to 100 000.

	$n = 1,000$	$n = 100,000$	Growth factor
A: n^2	10^6	10^{10}	$\times 10,000$
B: $n \lg n$	$\approx 10^4$	$\approx 1.7 \times 10^6$	$\times 170$
C: $\lg n$	≈ 10	≈ 17	$\times 1.7$

C is most suitable *if the operation can be expressed as a search on sorted or indexed data*. Where a full pass over the records is unavoidable (e.g. generating a report), B is the realistic target. A should be rejected: a $100\times$ growth in data causes a $10,000\times$ growth in work.

(D) Why small-data performance misleads. For small n the constant factors dominate: an $O(n^2)$ algorithm with a tiny constant can easily beat an $O(n \lg n)$ one with a large constant. Asymptotic behaviour describes the *tail* of the curve; below the crossover point the ordering can be reversed. Choosing on small-data benchmarks selects for the wrong property.

(E) Beyond a million records. Only $O(\lg n)$, $O(n)$ and $O(n \lg n)$ remain viable. At $n = 10^6$, an $O(n^2)$ algorithm requires 10^{12} operations — about 17 minutes at 1 ns per operation, for a single query. Anything at or above quadratic must be eliminated at design time.

Chapter summary

- **Big- O :** $g \in O(f)$ if $\exists c > 0, N \geq 0$ such that $g(n) \leq cf(n)$ for all $n \geq N$. Upper bound.
- **Big- Ω :** same with $\geq$. Lower bound.
- **Big- Θ :** $cf(n) \leq g(n) \leq df(n)$. Tight bound; $\Theta = O \cap \Omega$.
- **Small- o :** the same inequality but for *every* c — strictly slower growth, never a tight bound. o is to O as $<$ is to $\leq$.
- Report the **tightest** bound you can justify.
- **Simplify:** drop constants, drop lower-order terms, keep the fastest-growing term, ignore the log base. These rules apply to *sums*, never to products — $n \lg n$ is irreducible.
- **Structure:** nested loops multiply; consecutive loops add; **if/else** takes the max.
- **Loop patterns:** index $+c \Rightarrow O(n)$; index $\times c$ or $\div c \Rightarrow O(\lg n)$; triangular double loop $\Rightarrow O(n^2)$; geometric bounds $\Rightarrow$ sum the series, do not multiply.
- **Hierarchy:** $1 \prec \lg n \prec n \prec n \lg n \prec n^2 \prec n^3 \prec 2^n \prec n!$
- Faster hardware gives a multiplicative gain in instance size for polynomial algorithms and an *additive* gain for exponential ones. Only better algorithms change the class.

Recurrence Relations and Mathematical Induction

[Sources: Handout 5 §§“Time Complexity of Recursive Algorithms”, “Mathematical Induction”, and the binary-search / merge-sort derivations]

Consolidation

Handout 5 develops recurrences and induction inline, as they are needed for binary search and merge sort. Because the same machinery is reused for quick sort, Strassen’s algorithm and the maximum-subarray problem, it is collected here into one chapter and referenced from all of them.

5.1 Why recursive algorithms need different machinery

For a non-recursive algorithm, counting basic operations is a matter of adding up loop iterations (Chapter 4 §4.8). A recursive algorithm cannot be counted that way: its cost is defined *in terms of its own cost on smaller inputs*. The result is a **recurrence relation**, and solving it requires its own toolkit.

Recurrence relation

An equation that defines a sequence by expressing each term as a function of one or more earlier terms, together with one or more **initial conditions** (base cases) that anchor the recursion.

Algorithm	Recurrence	Solution
Binary search	$W(n) = W(n/2) + 1, W(1) = 1$	$\lg n + 1$
Merge sort	$W(n) = 2W(n/2) + n - 1,$ $W(1) = 0$	$\Theta(n \lg n)$
Quick sort (worst)	$T(n) = T(n - 1) + n, T(1) = 0$	$\Theta(n^2)$
Quick sort (best)	$T(n) = 2T(n/2) + n$	$\Theta(n \lg n)$
Naive matrix mult.	$T(n) = 8T(n/2) + \Theta(n^2)$	$\Theta(n^3)$
Strassen	$T(n) = 7T(n/2) + \Theta(n^2)$	$\Theta(n^{\lg 7}) \approx \Theta(n^{2.81})$
Naive Fibonacci	$T(n) = T(n - 1) + T(n - 2) + 1$	$\Theta(\varphi^n)$
Maximum subarray (D&C)	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \lg n)$

5.2 Reading a recurrence off an algorithm

Key idea

Every divide-and-conquer recurrence has the shape

$$T(n) = \underbrace{a}_{\text{number of subproblems}} \cdot T\left(\underbrace{n/b}_{\text{size of each subproblem}}\right) + \underbrace{f(n)}_{\text{cost to divide and combine}}$$

Identify a , b and $f(n)$ by inspection, then solve.

Algorithm	a	b	$f(n)$	Where the cost is
Binary search	1	2	$\Theta(1)$	One comparison to divide
Merge sort	2	2	$\Theta(n)$	The merge
Quick sort (balanced)	2	2	$\Theta(n)$	The partition
Naive matrix mult.	8	2	$\Theta(n^2)$	Submatrix additions
Strassen	7	2	$\Theta(n^2)$	The 18 additions/-subtractions

Note the contrast between merge sort and quick sort: both split into two halves, but merge sort’s expensive step is the *combine* (merging) while quick sort’s is the *divide* (partitioning). The recurrence is the same shape either way.

5.3 Method 1: Iteration (repeated substitution)

Expand the recurrence a few times, spot the pattern after k steps, then choose k to reach the base case.

Worked Example 5.1 | Binary search: $W(n) = W(n/2) + 1$, $W(1) = 1$

Assume $n = 2^k$ for a power of two.

Step 1 — compute small cases.

$$\begin{aligned} W(1) &= 1 \\ W(2) &= W(1) + 1 = 2 \\ W(4) &= W(2) + 1 = 3 \\ W(8) &= W(4) + 1 = 4 \\ W(16) &= W(8) + 1 = 5 \end{aligned}$$

Step 2 — spot the pattern.

$$W(2^k) = k + 1.$$

Step 3 — substitute back. Since $n = 2^k$, we have $k = \lg n$, so

$$W(n) = \lg n + 1$$

For general n (not a power of two) the answer is $W(n) = \lfloor \lg n \rfloor + 1$, which is $\Theta(\lg n)$.

Worked Example 5.2 | Quick sort worst case: $T(n) = T(n-1) + n$, $T(1) = 0$

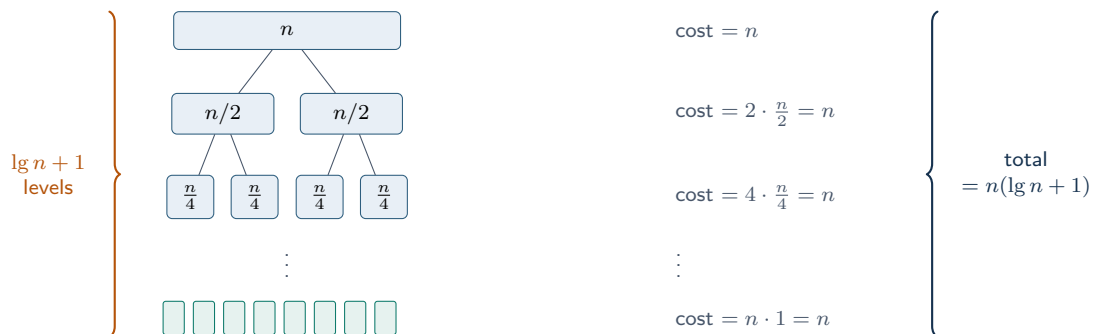
Expanding:

$$\begin{aligned}
 T(n) &= T(n-1) + n \\
 &= [T(n-2) + (n-1)] + n \\
 &= T(n-3) + (n-2) + (n-1) + n \\
 &\vdots \\
 &= T(1) + 2 + 3 + \cdots + n \\
 &= 0 + \left(\frac{n(n+1)}{2} - 1 \right) = \frac{n^2 + n - 2}{2} \in \Theta(n^2).
 \end{aligned}$$

This is exactly the unbalanced-partition case: each partition removes only one element, so the recursion depth is n and each level costs $\Theta(n)$.

5.4 Method 2: The recursion tree

Draw the tree of recursive calls, compute the cost at each *level*, and sum the levels. This is the most reliable method for $T(n) = aT(n/b) + f(n)$ and it makes the answer visually obvious.

Worked Example 5.3 | Merge sort: $T(n) = 2T(n/2) + n$ 

Every level costs exactly n , because halving the subproblem size doubles the number of subproblems. There are $\lg n + 1$ levels, since the size sequence $n, n/2, n/4, \dots, 1$ has $\lg n + 1$ entries. Therefore

$$T(n) = n(\lg n + 1) = n \lg n + n \in \Theta(n \lg n)$$

Key idea

The recursion tree makes the essential question visible: *is the work concentrated at the root, spread evenly across levels, or concentrated at the leaves?*

- Evenly spread (merge sort) $\Rightarrow \Theta(f(n) \cdot \text{depth})$
- Root-dominated $\Rightarrow \Theta(f(n))$
- Leaf-dominated (naive matrix multiplication) $\Rightarrow \Theta(n^{\log_b a})$

Those three cases are exactly the Master Theorem below.

5.5 Method 3: The Master Theorem

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants and let

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Compare $f(n)$ with the **critical exponent** $n^{\log_b a}$:

Case 1 If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$ (leaves dominate), then $T(n) = \Theta(n^{\log_b a})$.

Case 2 If $f(n) = \Theta(n^{\log_b a})$ (balanced), then $T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3 If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ (root dominates) and $af(n/b) \leq cf(n)$ for some $c < 1$ and large n , then $T(n) = \Theta(f(n))$.

Recurrence	a	b	$n^{\log_b a}$	$f(n)$	Result
$T(n) = T(n/2) + 1$	1	2	$n^0 = 1$	1	Case 2: $\Theta(\lg n)$
$T(n) = 2T(n/2) + n$	2	2	$n^1 = n$	n	Case 2: $\Theta(n \lg n)$
$T(n) = 8T(n/2) + n^2$	8	2	n^3	n^2	Case 1: $\Theta(n^3)$
$T(n) = 7T(n/2) + n^2$	7	2	$n^{2.807}$	n^2	Case 1: $\Theta(n^{\lg 7})$
$T(n) = 2T(n/2) + n^2$	2	2	n	n^2	Case 3: $\Theta(n^2)$
$T(n) = 4T(n/2) + n$	4	2	n^2	n	Case 1: $\Theta(n^2)$

Key idea

$\log_b a$ is where all the information lives. It is the exponent you get from the *leaves* of the recursion tree: there are $a^{\log_b n} = n^{\log_b a}$ of them. Strassen's whole contribution was reducing a from 8 to 7, which drops $\log_2 a$ from 3 to 2.807.

The Master Theorem does not apply to everything

It requires a and b to be *constants* and the subproblems to be of *equal* size. It does not apply to $T(n) = T(n-1) + n$ (subtraction, not division), or to $T(n) = T(n-1) + T(n-2) + 1$ (unequal subproblems). Use iteration or a recursion tree for those.

5.6 Method 4: Substitution (guess and verify by induction)

Guess the form of the answer, then prove it by induction. This is the only method that is always available, and it is why the next section on induction matters.

Worked Example 5.4 | Verify $T(n) = 2T(n/2) + n$ is $O(n \lg n)$

Guess. $T(n) \leq cn \lg n$ for some constant $c > 0$ and all $n \geq 2$.

Inductive step. Assume it holds for $n/2$:

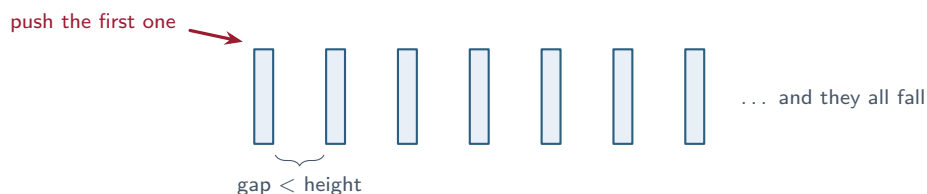
$$\begin{aligned}
 T(n) &= 2T(n/2) + n \\
 &\leq 2 \cdot c \frac{n}{2} \lg \frac{n}{2} + n && \text{(inductive hypothesis)} \\
 &= cn(\lg n - 1) + n \\
 &= cn \lg n - cn + n \\
 &\leq cn \lg n && \text{provided } c \geq 1.
 \end{aligned}$$

Base case. For $n = 2$: $T(2) = 2T(1) + 2$. With $T(1)$ constant, choosing c large enough makes $T(2) \leq c \cdot 2 \lg 2 = 2c$ hold. ■

5.7 Mathematical induction

[Handout 5. Induction underpins the substitution method and the correctness proofs for greedy algorithms in Chapter 9.]

5.7.1 The domino principle



Two conditions make the whole row fall:

1. You knock over the first domino.
2. The dominoes are spaced so that whenever the n th falls, it knocks over the $(n+1)$ st.

An induction proof has exactly these two parts.

Proof by mathematical induction

To prove a statement $P(n)$ holds for all integers $n \geq n_0$:

Induction base

Prove $P(n_0)$ directly.

Induction hypothesis

Assume $P(n)$ is true for an arbitrary $n \geq n_0$.

Induction step

Using the hypothesis, prove $P(n+1)$.

Conclude that $P(n)$ holds for all $n \geq n_0$.

Induction verifies; it does not discover

It is never possible to *derive* a true statement using induction. You must first guess the statement — typically by computing several cases and spotting the pattern — and only then can induction confirm it. If the guess is wrong, the induction step simply fails.

5.7.2 Worked induction proofs

Worked Example 5.5 | $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ for all $n \geq 1$

Base ($n = 1$). LHS = 1. RHS = $\frac{1 \cdot 2}{2} = 1$. ✓

Hypothesis. Assume $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ for some $n \geq 1$.

Step. Show it holds for $n + 1$:

$$\begin{aligned} \sum_{i=1}^{n+1} i &= \left(\sum_{i=1}^n i \right) + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) && \text{(hypothesis)} \\ &= \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}, \end{aligned}$$

which is the formula with n replaced by $n + 1$. ■

Worked Example 5.6 | $\sum_{i=0}^n 2^i = 2^{n+1} - 1$ for all $n \geq 0$

Base ($n = 0$). LHS = $2^0 = 1$. RHS = $2^1 - 1 = 1$. ✓

Step.

$$\sum_{i=0}^{n+1} 2^i = (2^{n+1} - 1) + 2^{n+1} = 2 \cdot 2^{n+1} - 1 = 2^{n+2} - 1. \quad \blacksquare$$

This identity is what makes the geometric-series argument in Chapter 4 §4.8 pattern 4 work: the sum of a doubling series is dominated by its final term.

Worked Example 5.7 | **Binary search:** $W(2^k) = k + 1$

Claim. For the recurrence $W(1) = 1$, $W(n) = W(n/2) + 1$, we have $W(2^k) = k + 1$ for all $k \geq 0$.

Base ($k = 0$). $W(2^0) = W(1) = 1 = 0 + 1$. ✓

Hypothesis. Assume $W(2^k) = k + 1$.

Step.

$$W(2^{k+1}) = W\left(\frac{2^{k+1}}{2}\right) + 1 = W(2^k) + 1 = (k + 1) + 1 = k + 2,$$

which is the claim with k replaced by $k + 1$. ■

Substituting $n = 2^k$, i.e. $k = \lg n$, gives $W(n) = \lg n + 1$.

5.7.3 Common summation formulas

Formula	Where it appears
$\sum_{i=1}^n 1 = n$	Any simple loop
$\sum_{i=1}^n i = \frac{n(n+1)}{2}$	Triangular loops; sequential-search average case; selection sort; bubble sort
$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$	Triple triangular loops
$\sum_{i=0}^n 2^i = 2^{n+1} - 1$	Doubling loops; complete binary trees; subset enumeration
$\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1} \quad (r \neq 1)$	General geometric series
$\sum_{i=1}^n \frac{1}{i} = H_n \approx \ln n + 0.577$	Harmonic loops; quick sort average case
$\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$	Enumerating 3-element subsets

5.8 Floor and ceiling

Recurrences for real algorithms produce $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ rather than $n/2$, because array indices are integers.

Floor and ceiling

$\lfloor x \rfloor$ = the greatest integer $\leq x$, $\lceil x \rceil$ = the least integer $\geq x$.

x	2.0	2.3	2.5	2.9	-2.3
$\lfloor x \rfloor$	2	2	2	2	-3
$\lceil x \rceil$	2	3	3	3	-2

Two facts used constantly:

- Integer division in most languages computes $\lfloor \cdot \rfloor$ for positive operands. `mid <- (low+high)/2` is $\lfloor (low + high)/2 \rfloor$.
- Splitting n items into two parts gives $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, and $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ exactly. This is why merge sort works for odd n with no special case: one half simply has one extra element.

Key idea

For asymptotic purposes floors and ceilings can be dropped: they change the answer by at most a constant. Keep them only when computing an exact count, e.g. $W(n) = \lfloor \lg n \rfloor + 1$ for binary search.

5.9 A worked catalogue of recurrences

Worked Example 5.8 | Solve $T(n) = 2T(n/2) + n - 1$, $T(1) = 0$ (merge sort, exact)

Handout 5 gives merge sort's exact worst case as $T(n) = 2T(n/2) + n - 1$, where the $n - 1$ is the maximum number of comparisons to merge two halves totalling n elements.

Assume $n = 2^k$. Expanding:

$$\begin{aligned}
 T(2^k) &= 2T(2^{k-1}) + 2^k - 1 \\
 &= 2 \left[2T(2^{k-2}) + 2^{k-1} - 1 \right] + 2^k - 1 \\
 &= 4T(2^{k-2}) + 2^k - 2 + 2^k - 1 \\
 &= 4T(2^{k-2}) + 2 \cdot 2^k - (1 + 2) \\
 &\vdots \\
 &= 2^k T(1) + k 2^k - (2^k - 1) \\
 &= k 2^k - 2^k + 1.
 \end{aligned}$$

Substituting $n = 2^k$, $k = \lg n$:

$$T(n) = n \lg n - n + 1 \in \Theta(n \lg n)$$

Check at $n = 4$: $4(2) - 4 + 1 = 5$. Direct computation: $T(2) = 2T(1) + 1 = 1$; $T(4) = 2T(2) + 3 = 2 + 3 = 5$. ✓

Worked Example 5.9 | Solve $T(n) = T(n-1) + T(n-2) + 1$, $T(0) = T(1) = 1$ (naive Fibonacci)

This is not of Master-Theorem form. Note that $T(n) > f_n$, the n th Fibonacci number, and $f_n = \Theta(\varphi^n)$ where $\varphi = \frac{1+\sqrt{5}}{2}$. In fact $T(n) = 2f_{n+1} - 1$ exactly.

n	5	10	20	30	40	50
calls $T(n)$	15	177	21,891	2.7×10^6	3.3×10^8	4.1×10^{10}

Each increment of n multiplies the work by roughly 1.618. This is the concrete justification for dynamic programming (Chapter 8).

Worked Example 5.10 | Solve $T(n) = T(n/2) + \Theta(1)$ versus $T(n) = 2T(n/2) + \Theta(1)$

These look similar and behave completely differently.

One subproblem ($a = 1$, $b = 2$, $f = \Theta(1)$): $n^{\log_2 1} = n^0 = 1 = \Theta(f(n))$, so Case 2 gives $\Theta(\lg n)$ — binary search.

Two subproblems ($a = 2$, $b = 2$, $f = \Theta(1)$): $n^{\log_2 2} = n$, and $f(n) = 1 = O(n^{1-\epsilon})$, so Case 1 gives $\Theta(n)$ — e.g. traversing a balanced binary tree.

Discarding one half of the data (binary search) is exponentially better than visiting both halves (traversal), even though the code differs by a single recursive call.

Exam technique

Exam questions on recurrences almost always take one of three forms:

1. “Derive the recurrence for this recursive algorithm” — identify a , b and $f(n)$ from the code and state the base case.
2. “Solve this recurrence” — iterate three or four times, spot the pattern, substitute $n = 2^k$, convert back with $k = \lg n$.
3. “Prove by induction that ...” — always write the three headings *Base*, *Hypothesis*, *Step* explicitly. Marks are attached to the structure, not just the algebra.

Chapter summary

- A recursive algorithm’s cost is a **recurrence relation**: recursive terms plus a divide-and-combine cost, anchored by base cases.
- Standard form: $T(n) = aT(n/b) + f(n)$, where a is the number of subproblems, b the shrink factor, $f(n)$ the divide-and-combine cost.
- Four solution methods: **iteration** (expand and spot the pattern), **recursion tree** (cost per level $\times$ number of levels), **Master Theorem** (compare $f(n)$ with $n^{\log_b a}$), and **substitution** (guess, then prove by induction).
- Master Theorem: leaves dominate $\Rightarrow \Theta(n^{\log_b a})$; balanced $\Rightarrow \Theta(n^{\log_b a} \lg n)$; root dominates $\Rightarrow \Theta(f(n))$. It requires equal-sized subproblems and constant a , b .
- **Induction** has three parts: base, hypothesis, step. It verifies a guess; it never produces one.
- Memorise $\sum i = \frac{n(n+1)}{2}$, $\sum i^2 = \frac{n(n+1)(2n+1)}{6}$, $\sum 2^i = 2^{n+1} - 1$.
- Floors and ceilings are dropped for asymptotics and kept for exact counts. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$.

Part III

Algorithm Design Paradigms

Brute Force

[Sources: Handout 5 §§1–17; Lab Sheets 07 and 09]

6.1 The idea

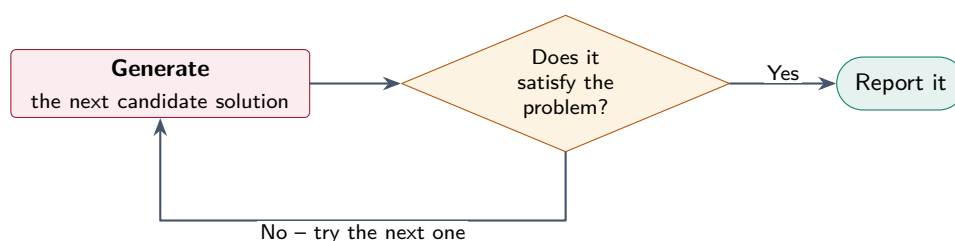
Brute force

A problem-solving approach that generates *every* candidate solution and tests each one, applying no strategic insight about the structure of the problem. Also called exhaustive search.

The name comes from military usage, where it described a direct, unconcealed, overwhelming assault rather than a stratagem. The computing sense preserves the meaning exactly: overwhelm the problem with effort rather than cleverness.

Key idea

Imagine a castle with a locked gate and no key. Strategy would mean climbing the walls, finding a hidden entrance, or picking the lock. Brute force means hitting the gate until it breaks — no subtlety, no plan, just power.



6.2 The canonical illustration: the padlock

A four-digit combination padlock, each dial 0–9. You have forgotten the combination and do not want to buy a new lock. Set all dials to 0 and try 0000, 0001, 0002, ... until it opens.

Worst case: $10^4 = 10,000$ attempts. There is no cleverness available — the lock gives no partial feedback — so brute force is not merely acceptable here, it is optimal.

Key idea

Brute force is the right answer when the problem provides *no exploitable structure*. It is the wrong answer whenever structure exists and is ignored.

6.3 Time complexities of brute-force algorithms

Complexity	Task	Example
$O(n!)$	Generate all permutations of n elements	Brute-force travelling salesman; array permutation
$O(2^n)$	Enumerate all subsets of an n -element set	0/1 knapsack by exhaustive search; subset sum
$O(n^m)$	Check all possibilities for m variables each taking n values	Constraint satisfaction over a fixed domain
$O(m^n)$	Try all passwords of length n over an alphabet of m characters	Password cracking
$O(n^2)$	Check all pairs	All-pairs comparison, plagiarism detection
$O(n^3)$	Check all contiguous subarrays and sum each	Naive maximum subarray

6.3.1 How fast these grow

n	n^2	2^n	$n!$
5	25	32	120
10	100	1,024	3,628,800
15	225	32,768	1.31×10^{12}
20	400	1.05×10^6	2.43×10^{18}
25	625	3.36×10^7	1.55×10^{25}

At one million evaluations per second, $n = 20$ takes about a second for 2^n and about *77 thousand years* for $n!$.

6.4 Advantages and disadvantages

Advantages	Disadvantages
Simple to describe, implement and reason about	Scales terribly — often exponential or factorial
Guaranteed to find a solution if one exists	Wasteful use of computation
Works for small instances and one-off problems	Too slow for real-time or interactive systems
Provides a correctness baseline for testing optimised algorithms	Exploits no problem-specific structure or heuristic
Needs no proof of a clever property	Cannot be used where the search space is unbounded

Key idea

Brute force is the algorithm you write *first* — to establish that the problem is well-defined and to generate correct answers for small inputs against which a faster algorithm can be tested. It is rarely the algorithm you ship.

6.5 Optimising brute force

When a complete search is genuinely necessary, four techniques help without changing the fundamental approach.

Technique	What it does	Effect
Pruning	Eliminate infeasible candidates as early as possible, before they are fully constructed	Cuts whole branches of the search space (this is the basis of Chapter 10)
Heuristics	Order the search so promising regions are explored first	Finds a good solution sooner; does not reduce the worst case
Memoisation	Cache intermediate results so identical subproblems are not recomputed	Turns exponential into polynomial when subproblems overlap (Chapter 8)
Parallelism	Distribute independent candidates across cores or machines	Constant-factor speedup equal to the number of workers

Parallelism does not defeat exponential growth

Dividing a 2^n search across 1000 machines reduces the work per machine to $2^n/1000 \approx 2^{n-10}$. You have bought *ten* extra elements of problem size — the same additive-gain phenomenon seen in Chapter 4. For exponential problems, only a better algorithm helps.

6.6 Alternatives to brute force

Handout 5 lists five families, which is a preview of the rest of this course.

Technique	Core move	Travel-planning analogy [H5]
Divide and conquer	Split into independent parts, solve, combine	Plan each city separately, then stitch the itinerary together
Greedy	Take the best immediate option at every step	Always choose the cheapest next flight
Dynamic programming	Solve overlapping subproblems once and remember	Save route calculations for reuse
Reduction	Transform into a different problem with a known efficient algorithm	Convert travel planning into a map-search problem
Approximation	Accept a provably near-optimal answer, quickly	Get a near-best itinerary without checking every option

Technique	Everyday analogy	Representative algorithms
Divide & conquer	Clean a messy room by dividing it into sections	Merge sort, quick sort, binary search, Strassen
Greedy	Walk home always taking the road that looks shortest now	Dijkstra, Huffman coding, Kruskal, Prim
Dynamic programming	Study from saved notes rather than relearning each topic	Knapsack, longest common subsequence, Floyd–Warshall
Reduction	Convert currency because the shop accepts only one	Subset sum $\rightarrow$ knapsack; many NP-completeness proofs
Approximation	Pack luggage quickly rather than trying every arrangement	Set cover, vertex cover, TSP approximations

6.7 Applications where brute force is genuinely used

Domain	Use
Cryptography	Attacking weak encryption by trying all keys — and, defensively, sizing key spaces so that brute force is infeasible
Bioinformatics	Comparing DNA sequences to find mutations; aligning protein structures
Puzzle solving	Exhaustively exploring moves in chess endgames, Sudoku, Rubik’s cube
Software testing	Exhaustive testing over all input combinations for small parameter spaces
Competitive programming and interviews	Frequently sufficient within the time limit for the given constraints
Baseline verification	Generating known-correct outputs to validate an optimised implementation

Exam technique

“Brute force attacks in cyber security” are the security-facing side of the same idea: an attacker tries every possible password. The defence is not a cleverer algorithm but a larger search space (longer passwords, bigger alphabet) plus rate limiting. If a password supports only digits and is four characters long, there are $10^4 = 10,000$ combinations — trivially brute-forced. Each extra character multiplies the space by 10.

6.8 The travelling salesman problem

Travelling Salesman Problem (TSP)

Given a set of cities and the distance between every pair, find the shortest possible route that visits every city exactly once and returns to the starting city.

The brute-force method:

1. Count the total number of routes.
2. Enumerate every route.

3. Compute the total distance of each.
4. Select the minimum.

6.8.1 Counting the routes

Fix the starting city (any tour can be rotated to start anywhere). The remaining $n - 1$ cities can be visited in $(n - 1)!$ orders. If direction does not matter, each tour is counted twice, giving $(n - 1)!/2$ distinct tours.

Cities n	Routes $(n - 1)!$	Distinct tours $(n - 1)!/2$
5	24	12
8	5,040	2,520
10	362,880	181,440
12	39,916,800	19,958,400
15	8.7×10^{10}	4.4×10^{10}
20	1.2×10^{17}	6.1×10^{16}

Worked Example 6.1 | Lab Sheet 09 Q1 — route planning with 8 locations

A logistics company's route planner examines every possible route before choosing the shortest.

(a) Which strategy is this? Brute force (exhaustive search). It enumerates every permutation of delivery locations, computes each total distance, and keeps the minimum. No structural property of the distance matrix is exploited.

(b)(i) Number of routes for 8 locations. Interpreting the question as “all orderings of 8 locations”:

$$8! = 40,320.$$

If one location is a fixed depot and the other 7 are permuted, the count is $7! = 5,040$; halving for direction gives 2,520 distinct tours. *State your interpretation* — all three are defensible and markers accept any of them if justified.

(b)(ii) Execution time at 2 million routes per second.

$$\frac{40,320}{2 \times 10^6} = 0.0202 \text{ seconds} \approx 20 \text{ ms.}$$

(With $7!$: $5040/2 \times 10^6 = 2.52 \text{ ms.}$)

Eight locations is trivial. The point of the question is what happens next:

Locations	Routes $n!$	Time at $2 \times 10^6/\text{s}$
8	40,320	0.02 s
10	3,628,800	1.8 s
12	4.79×10^8	4 minutes
15	1.31×10^{12}	7.6 days
20	2.43×10^{18}	38,500 years

(c) Two advantages and two disadvantages.

Advantages: (i) guaranteed to find the true optimum — no approximation error; (ii) trivially simple to implement and to verify, with no assumptions about the distance metric (it works

even if distances are asymmetric or violate the triangle inequality).

Disadvantages: (i) factorial growth makes it unusable past about a dozen locations — exactly the range a real delivery fleet operates in; (ii) it wastes enormous effort re-evaluating routes that share long common prefixes, and cannot respond within a real-time dispatch window.

(d) Recommended alternative. TSP is NP-hard (Chapter 12), so no known exact algorithm is polynomial. A layered recommendation:

- *Exact, for moderate n :* **branch and bound** (Chapter 11) or Held–Karp dynamic programming at $O(n^2 2^n)$ — still exponential, but $n = 20$ becomes feasible where $n!$ is not.
- *Practical, for real fleets:* a **greedy** nearest-neighbour construction (Chapter 9) to get an initial tour quickly, improved by a local-search heuristic such as 2-opt.
- *With provable quality:* a minimum-spanning-tree-based **approximation** algorithm, which guarantees a tour within a bounded factor of optimal in polynomial time.

Justification: real routing also carries extra constraints — time windows, vehicle capacity, driver hours — which exact methods handle badly and heuristics handle naturally.

Consolidation note

TSP recurs three more times in this note: as an application of greedy heuristics (§9.13), as the standard branch-and-bound example (Chapter 11), and as the canonical NP-hard problem (Chapter 12).

6.9 Brute force on the maximum subarray problem

[Lab Sheet 07 Q2. The divide-and-conquer and linear solutions follow in Chapter 7 §7.6.]

Maximum subarray problem

Given an array $A[1..n]$ of numbers (which may be negative), find the contiguous subarray with the largest sum.

6.9.1 The brute-force approach

Enumerate every contiguous subarray, sum each, keep the maximum.

Brute force, $\Theta(n^3)$

```

1  function maxSubarrayBrute(A[1..n])
2      best ← -infinity
3      for i ← 1 to n do                // start index
4          for j ← i to n do            // end index
5              sum ← 0
6              for k ← i to j do        // sum the slice
7                  sum ← sum + A[k]
8              end for
9              if sum > best then
10                 best ← sum; bestI ← i; bestJ ← j
11             end if
12         end for

```

```

13  end for
14  return best, bestI, bestJ

```

Brute force with a running sum, $\Theta(n^2)$

```

1  function maxSubarrayBrute2(A[1..n])
2      best ← -infinity
3      for i ← 1 to n do
4          sum ← 0
5          for j ← i to n do
6              sum ← sum + A[j]           // extend, don't re-sum
7              if sum > best then best ← sum; bestI ← i; bestJ ← j
8          end for
9      end for
10     return best, bestI, bestJ

```

6.9.2 Counting the subarrays

The number of contiguous subarrays of an n -element array is

$$\sum_{i=1}^n (n - i + 1) = \sum_{k=1}^n k = \frac{n(n+1)}{2}.$$

For $n = 10$ that is 55 subarrays. The triple-loop version additionally re-sums each subarray, giving $\Theta(n^3)$ additions overall; the running-sum version does one addition per subarray, giving $\Theta(n^2)$.

Worked Example 6.2 | Lab Sheet 07 Q2 — retail profit/loss data

$$A = [250, -120, 340, -180, 520, -90, 160, -240, 430, -110]$$

(A) The approach. For every pair (i, j) with $i \leq j$, compute $\sum_{k=i}^j A[k]$ and retain the largest. There are $\frac{n(n+1)}{2}$ such pairs. With naive re-summation the cost is $\Theta(n^3)$; with a running sum it drops to $\Theta(n^2)$.

(B) The maximum subarray. Working with prefix sums $P_0 = 0$, $P_j = \sum_{k \leq j} A[k]$:

j	0	1	2	3	4	5	6	7	8	9	10
$A[j]$	—	250	-120	340	-180	520	-90	160	-240	430	-110
P_j	0	250	130	470	290	810	720	880	640	1070	960

The subarray sum for $[i..j]$ is $P_j - P_{i-1}$, maximised by taking the largest P_j and the smallest earlier P_{i-1} . The smallest prefix is $P_0 = 0$ and the largest is $P_9 = 1070$, giving

$$\boxed{\text{maximum subarray sum} = 1070, \quad \text{subarray } A[1..9]}$$

i.e. $250 - 120 + 340 - 180 + 520 - 90 + 160 - 240 + 430 = 1070$.

The whole array sums to 960, so including the final -110 is worse; and every proper prefix sums to less than 1070. The optimum runs from day 1 to day 9.

(C) Subarrays examined and comparisons.

$$\text{subarrays} = \frac{10 \times 11}{2} = 55.$$

The running-sum version performs one addition and one comparison per subarray: 55 additions and 55 comparisons. The triple-loop version performs $\sum_{i \leq j} (j - i + 1) = \binom{n+2}{3} = \binom{12}{3} = 220$ additions.

(D) Why it degrades, and what to use instead. The $\Theta(n^2)$ (or $\Theta(n^3)$) cost means a $10\times$ larger dataset costs $100\times$ (or $1000\times$) more. For daily data over ten years ($n \approx 3650$) the quadratic version needs about 6.7 million operations and the cubic version about 8×10^9 — the latter is already impractical.

Better alternatives:

Algorithm	Complexity	Idea
Brute force (triple loop)	$\Theta(n^3)$	Re-sum every subarray
Brute force (running sum)	$\Theta(n^2)$	Extend rather than re-sum
Divide and conquer	$\Theta(n \lg n)$	Best is in the left half, right half, or crosses the midpoint
Kadane's algorithm	$\Theta(n)$	At each element, either extend the current best-ending-here or start fresh

Kadane's algorithm is the recommended solution: one pass, constant extra space.

Kadane's algorithm, $\Theta(n)$ — for reference

```

1 function kadane(A[1..n])
2   best ← A[1]; current ← A[1]
3   for i ← 2 to n do
4     current ← max( A[i], current + A[i] )  // extend, or restart here
5     best ← max( best, current )
6   end for
7   return best

```

Key idea

The progression $n^3 \rightarrow n^2 \rightarrow n \lg n \rightarrow n$ on one problem is the entire course in miniature. Each step comes from noticing that the previous version repeats work: re-summing slices, then re-examining overlapping halves, then re-deciding a choice that a single running value already determines.

Chapter summary

- **Brute force** generates every candidate and tests each. Simple, guaranteed correct, and usually far too slow.
- Typical complexities: $O(n!)$ (permutations), $O(2^n)$ (subsets), $O(m^n)$ (passwords), $O(n^2)$ (pairs).
- It is the right choice when the problem offers *no structure* to exploit, when n is small, or when a correctness baseline is needed.
- **Optimisations:** pruning, heuristics, memoisation, parallelism. Only the first two change the search; none changes the complexity class.

- **Alternatives:** divide and conquer, greedy, dynamic programming, reduction, approximation — Chapters 7–11.
- **TSP:** $(n - 1)!$ routes (or $(n - 1)!/2$ distinct tours). Feasible to about $n = 12$; NP-hard in general.
- **Maximum subarray:** $\frac{n(n+1)}{2}$ contiguous subarrays. Brute force is $\Theta(n^3)$ or $\Theta(n^2)$; divide and conquer is $\Theta(n \lg n)$; Kadane's is $\Theta(n)$.

Divide and Conquer

[Sources: Handout 5 §§18–119; Handout 2 (binary search); Lab Sheets 07, 08, 09]

7.1 The strategy

Divide and conquer

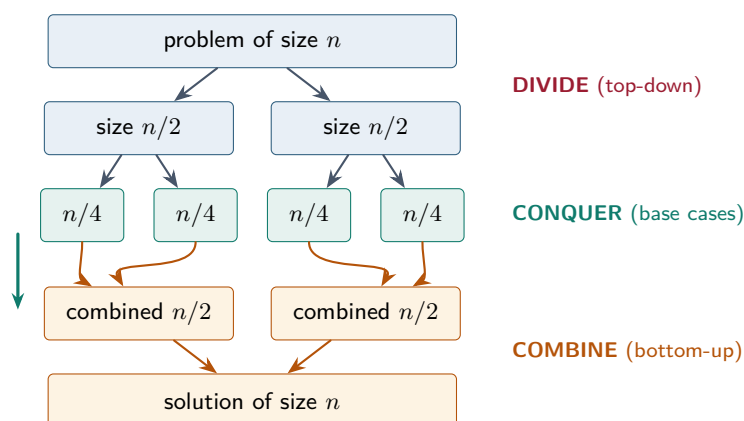
A *top-down* strategy with three phases:

Divide Split an instance of the problem into two or more smaller instances. If the smaller instances are still too large, divide them further, until the instances are small enough to solve directly.

Conquer Solve the smaller instances — usually by recursion, since they are instances of the same problem.

Combine Assemble the solutions of the smaller instances into a solution of the original.

The name comes from the military tactic of bringing the whole of one's force to bear on a fraction of the enemy in sequence, rather than engaging the entire opposing force at once — a favourite of Napoleon's.



Key idea

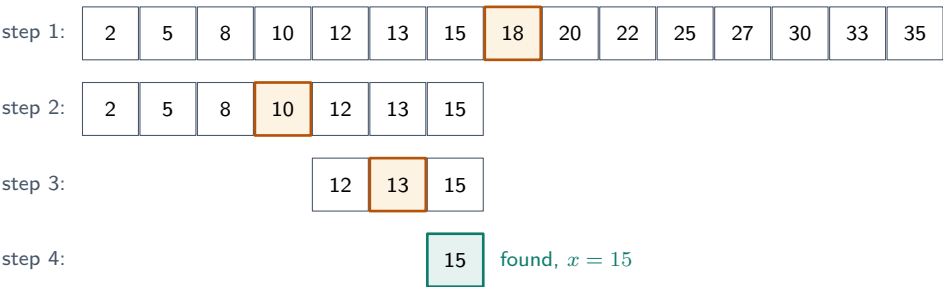
The defining requirement is that the subproblems are **independent** — solving one gives no information about another, and none is solved twice. When subproblems *overlap*, divide and conquer wastes exponential effort and dynamic programming (Chapter 8) is the right tool.

7.2 Binary search revisited as divide and conquer

Binary search (Chapter 2 §2.3) is the simplest divide-and-conquer algorithm, and an unusual one:

Phase	What happens
Divide	Compare x with $S[mid]$; this splits the array into two halves
Conquer	Recurse into <i>one</i> half only — the other is discarded
Combine	Nothing to do; the sub-answer <i>is</i> the answer

Because $a = 1$ (one subproblem, not two) and the combine step is free, the recurrence is $W(n) = W(n/2) + 1$ and the solution is $W(n) = \lfloor \lg n \rfloor + 1$ — derived in full in Chapter 5.



7.3 Merge sort

7.3.1 The idea

Merging means combining two *already sorted* arrays into one sorted array. Merge sort applies this repeatedly:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves.

The recursion bottoms out at arrays of size 1, which are trivially sorted.

Algorithm 7.1 — Merge sort

```
1 procedure mergesort(int n, keytype S[1..n])
2   if n > 1 then
3     h ← floor(n/2)
4     m ← n - h
5     copy S[1..h] to U[1..h]           // DIVIDE
6     copy S[h+1..n] to V[1..m]
7     mergesort(h, U)                   // CONQUER
8     mergesort(m, V)
9     merge(h, m, U, V, S)              // COMBINE
10  end if
11 end procedure
```

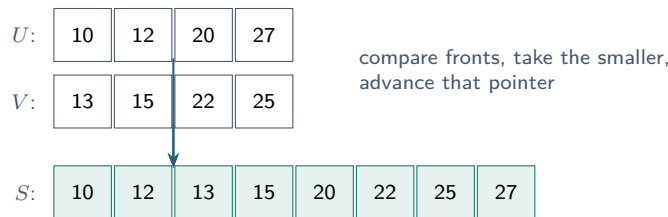
Algorithm 7.2 — Merge

```
1 procedure merge(int h, int m, const keytype U[1..h], const keytype V[1..m],
2               keytype S[1..h+m])
3   index i, j, k
4   i ← 1; j ← 1; k ← 1
5   while i ≤ h and j ≤ m do
6     if U[i] < V[j] then           // BASIC OPERATION: this comparison
7       S[k] ← U[i]; i ← i + 1
8     else
9       S[k] ← V[j]; j ← j + 1
10    end if
11    k ← k + 1
12  end while
13  if i > h then
14    copy V[j..m] to S[k..h+m]     // left half exhausted
15  else
16    copy U[i..h] to S[k..h+m]    // right half exhausted
17  end if
18 end procedure
```

Odd-length arrays need no special case

$h = \lfloor n/2 \rfloor$ and $m = n - h = \lceil n/2 \rceil$, and $h + m = n$ exactly. If n is odd, one half simply gets one extra element. The complexity is unaffected.

7.3.2 The merge step in detail



Step	U (remaining)	V (remaining)	Comparison	S so far
1	10 12 20 27	13 15 22 25	10 < 13	10
2	12 20 27	13 15 22 25	12 < 13	10 12
3	20 27	13 15 22 25	20 > 13	10 12 13
4	20 27	15 22 25	20 > 15	10 12 13 15
5	20 27	22 25	20 < 22	10 12 13 15 20
6	27	22 25	27 > 22	10 12 13 15 20 22
7	27	25	27 > 25	10 12 13 15 20 22 25
8	27	—	V empty: copy rest	10 12 13 15 20 22 25 27

Seven comparisons for eight elements — the maximum possible is $h + m - 1 = 7$.

Key idea

Merging two sorted lists of total length n requires at most $n - 1$ comparisons, because each comparison places exactly one element, and the last element is placed free when one list

runs out.

7.3.3 Complexity

$$W(n) = \underbrace{2 W(n/2)}_{\text{two halves}} + \underbrace{n - 1}_{\text{merge}}, \quad W(1) = 0$$

By the Master Theorem ($a = 2$, $b = 2$, $f(n) = \Theta(n)$, $n^{\log_2 2} = n$): Case 2, so $W(n) \in \Theta(n \lg n)$. The exact solution (Chapter 5) is

$$W(n) = n \lg n - n + 1.$$

Case	Complexity	Why
Best	$\Omega(n \lg n)$	Even an already-sorted array is fully divided and fully merged
Average	$\Theta(n \lg n)$	Typical input requires the same recursive division and merging
Worst	$O(n \lg n)$	Reverse order still requires the same divide and merge operations
Auxiliary space	$O(n)$	The temporary arrays U and V used during merging

Key idea

Merge sort has an *every-case* complexity. Its performance is completely independent of the initial arrangement of the data. That predictability is its main selling point — and its main cost is the $O(n)$ extra memory.

7.3.4 Properties: stability and in-place

Stable sorting

A sorting algorithm is **stable** if equal elements retain their original relative order after sorting.

Worked Example 7.1 | Stability, illustrated

Records: [(Alice,85), (Bob,92), (Charlie,85), (David,78)], sorted by score.
Stable (merge sort): [(David,78), (Alice,85), (Charlie,85), (Bob,92)] — Alice still precedes Charlie, as in the input.
Unstable (quick sort): [(David,78), (Charlie,85), (Alice,85), (Bob,92)] — Charlie has jumped ahead of Alice.
Stability matters when sorting by a secondary key: sorting by name and then stably by score leaves records grouped by score and alphabetised within each score.

In-place sorting

A sorting algorithm is **in-place** if it rearranges elements within the original array using only a small constant amount of additional storage — $O(1)$ auxiliary space.

Stability and in-place are independent properties

All four combinations exist:

	In-place	Not in-place
Stable	Insertion sort, bubble sort	Merge sort
Unstable	Quick sort , heap sort, selection sort	(rare)

7.3.5 Advantages, disadvantages and applications

Advantages	Disadvantages
Stable — preserves the order of equal elements Guaranteed $O(n \lg n)$ in every case Simple, regular divide-and-conquer structure Naturally parallel — subproblems are independent and share no data, so they can run on separate cores Excellent for linked lists (merging needs no random access)	Not in-place — needs $O(n)$ extra memory Slower than quick sort in practice on arrays Poorer cache locality — data is copied between arrays Extra allocation cost for the temporary arrays
Application	Why merge sort
Sorting very large datasets	Predictable $n \lg n$ regardless of the data
External sorting	Data too large for memory — see below
Sorting linked lists	No random access needed; can be done with $O(1)$ extra space on lists
Parallel and distributed sorting	Independent subproblems map naturally onto cores or machines
Counting inversions	The merge step counts out-of-order pairs for free

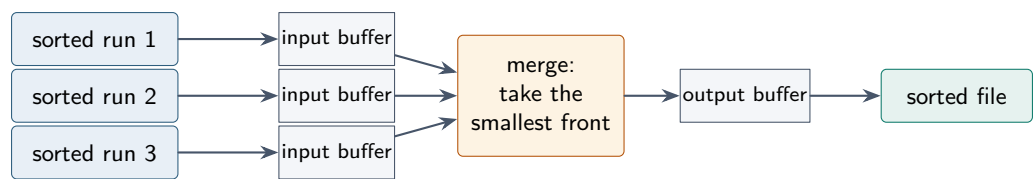
7.3.6 External merge sort

When the dataset does not fit in memory:

1. **Create sorted runs.** Read as much data as fits in memory (say 100 MB), sort that chunk with any internal sort, and write the sorted *run* back to disk. Repeat until all data has been processed.
2. **Merge the runs.** Read a small block from the front of each run into an input buffer. Repeatedly: find the smallest front element across all buffers, write it to the output buffer;

when an input buffer empties, read the next block from that run; when the output buffer fills, write it to disk.

3. Repeat until a single fully sorted file remains.



7.3.7 Worked merge sort traces

Worked Example 7.2 | Lab Sheet 07 Q1 — delivery IDs

[45, 18, 72, 11, 39, 56, 8, 91, 27, 63, 14, 50] (n = 12)

(B) Split–merge tree.

(With $h = \lfloor n/2 \rfloor$, a 3-element array splits as 1 + 2.)

(A) Merge trace table — bottom-up, with comparison counts.

Level	Merge	Result	Comps
4	[18] + [72]	[18, 72]	1
4	[39] + [56]	[39, 56]	1
4	[91] + [27]	[27, 91]	1
4	[14] + [50]	[14, 50]	1
3	[45] + [18, 72]	[18, 45, 72]	2
3	[11] + [39, 56]	[11, 39, 56]	1
3	[8] + [27, 91]	[8, 27, 91]	1
3	[63] + [14, 50]	[14, 50, 63]	2
2	[18, 45, 72] + [11, 39, 56]	[11, 18, 39, 45, 56, 72]	5
2	[8, 27, 91] + [14, 50, 63]	[8, 14, 27, 50, 63, 91]	5
1	[11, 18, 39, 45, 56, 72] + [8, 14, 27, 50, 63, 91]	[8, 11, 14, 18, 27, 39, 45, 50, 56, 63, 72, 91]	11
Total			31

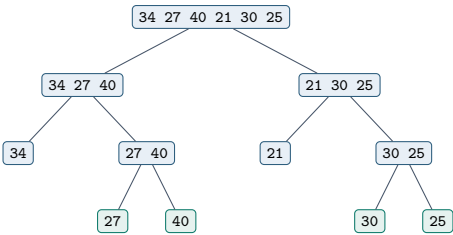
(C) Total comparisons: 31. Cross-check against the upper bound: at each merge, at most (combined size – 1) comparisons. Summing the upper bounds gives 4(1) + (2 + 2 + 2 + 2) + (5 + 5) + 11 = 4 + 8 + 10 + 11 = 33, and the actual count 31 is just below it because two merges exhausted one side early. The exact-formula estimate $n \lg n - n + 1 = 12(3.585) - 12 + 1 \approx 32$ is likewise consistent.

(D) Time complexity. Best, average and worst are all $\Theta(n \lg n)$. Merge sort is insensitive to the initial ordering because the divide phase always splits to single elements regardless of the data, and the merge phase always processes every element at every level. No arrangement of input can shorten either phase — unlike quick sort, where the data determines the partition balance.

Worked Example 7.3 | Lab Sheet 08 B — Candidate IDs

[34, 27, 40, 21, 30, 25] ($n = 6$)

(1) Divide and merge phases.



Merge	Result	Comparisons
[27] + [40]	[27, 40]	1
[30] + [25]	[25, 30]	1
[34] + [27, 40]	[27, 34, 40]	2
[21] + [25, 30]	[21, 25, 30]	1
[27, 34, 40] + [21, 25, 30]	[21, 25, 27, 30, 34, 40]	4
Total		9

(2) Estimated comparisons for 1000 records using $T(n) = n \log_2 n$.

$$\log_2 1000 = \frac{\ln 1000}{\ln 2} \approx 9.966, \quad T(1000) \approx 1000 \times 9.966 \approx \mathbf{9,966} \text{ comparisons.}$$

Using the tighter exact formula $n \lg n - n + 1$ gives $9966 - 1000 + 1 = 8,967$, which is the true worst case. Both are acceptable; state which you used.

(3) Parallel merge sort across 10 servers. The divide phase creates subproblems that share no data, so they can be sorted simultaneously with no coordination. With 10 servers each sorting 1/10 of the districts:

- Each server sorts $n/10$ records in $\Theta(\frac{n}{10} \lg \frac{n}{10})$ time, all ten running concurrently.
- A final merge phase combines the ten sorted runs. A 10-way merge costs $\Theta(n \lg 10)$, or a binary merge tree of depth $\lg 10 \approx 3.3$ costs $\Theta(n \lg 10)$ as well.

The overall wall-clock time drops from $\Theta(n \lg n)$ to roughly $\Theta(\frac{n}{10} \lg \frac{n}{10} + n \lg 10)$. The gain is a constant factor close to the number of servers — not a change in complexity class — but for election-night throughput a $\sim 8\times$ speedup is decisive. Merge sort is particularly suited to this because the subproblems are independent; quick sort's partitions are also independent but their *sizes* are unpredictable, which makes load balancing harder.

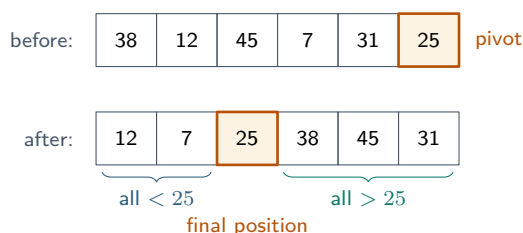
7.4 Quick sort

7.4.1 The idea

Quick sort achieves the same divide-and-conquer benefits as merge sort *without* additional storage. Instead of splitting the array in the middle and doing work during the merge, it does the work up front during the split.

Pivot and split point

A **pivot** is a chosen array value. The **partition** step rearranges the array so that every element smaller than the pivot is to its left and every element greater is to its right. The pivot then occupies its final sorted position — the **split point** — and the array is divided there for the recursive calls.



Algorithm 7.3 — Quick sort with Lomuto partition (last element as pivot)

```

1  procedure quicksort(keytype S[], index low, index high)
2    if low < high then
3      p ← partition(S, low, high)      // DIVIDE: places pivot at its final index
4      quicksort(S, low, p - 1)         // CONQUER left
5      quicksort(S, p + 1, high)        // CONQUER right
6    end if                             // COMBINE: nothing to do
7  end procedure
8
9  function partition(keytype S[], index low, index high) returns index
10   pivot ← S[high]
11   i ← low - 1
12   for j ← low to high - 1 do
13     if S[j] ≤ pivot then
14       i ← i + 1
15       exchange S[i] and S[j]
16     end if
17   end for
18   exchange S[i+1] and S[high]
19   return i + 1
20 end function

```

Key idea

Merge sort does no work in the divide and all the work in the combine. Quick sort does all the work in the divide and *no* work in the combine. That is the whole difference between them.

7.4.2 Pivot selection

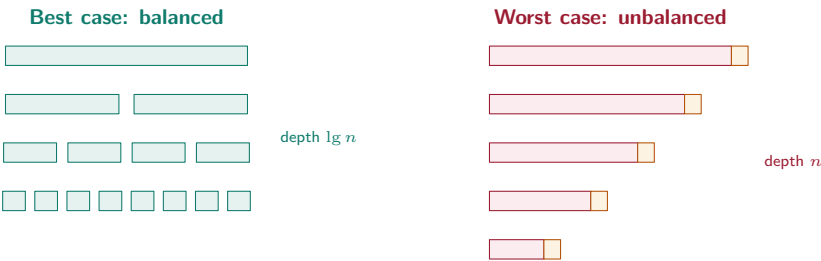
Strategy	Behaviour	Risk
First element	Simple	Worst case on already-sorted input
Last element	Simple (Lomuto’s standard choice)	Worst case on already-sorted input
Middle element	Simple	Worst case on specially-constructed input
Random element	Expected $\Theta(n \lg n)$ regardless of the input	Requires a random-number source
Median-of-three	Take the median of first, middle and last	Very robust in practice; slight overhead

Worked Example 7.4 | Median-of-three

For $[90, \dots, 50, \dots, 80]$ the three candidates are first = 90, middle = 50, last = 80. Sorted: 50, 80, 90. The median is **80**, so 80 becomes the pivot. This avoids the pathological case where the first or last element happens to be extreme.

7.4.3 Complexity

Case	Complexity	When
Best	$\Omega(n \lg n)$	Pivot splits the array into roughly equal halves at every level. Recurrence $T(n) = 2T(n/2) + n$.
Average	$\Theta(n \lg n)$	Random data. Even fairly unbalanced splits (e.g. 10:90) still give logarithmic depth.
Worst	$O(n^2)$	Pivot is always the smallest or largest element — e.g. an already-sorted array with a first/last-element pivot. Recurrence $T(n) = T(n - 1) + n$.
Auxiliary space	$O(\lg n)$ avg, $O(n)$ worst	The recursion stack only; partitioning is in place.



7.4.4 Advantages, disadvantages and applications

Advantages	Disadvantages
In-place — $O(1)$ auxiliary space beyond the recursion stack	Worst case is $O(n^2)$ with a poor pivot
Cache-friendly — works within one array, giving excellent locality	Not stable — swaps ignore original positions
Fastest general-purpose sort for large random data when stability is not required	Not a good choice for small datasets (overhead dominates)
Tail recursive — the second recursive call can be optimised away	Performance depends on pivot choice
Low constant factors	

Key idea

Quick sort is *usually* faster than merge sort in practice on arrays — not because of a better complexity class (it is worse) but because in-place partitioning means far better cache behaviour and no allocation. Merge sort wins on linked lists, when stability is required, and when the $O(n \lg n)$ guarantee must be absolute.

7.4.5 Worked quick sort traces

Worked Example 7.5 | Lab Sheet 08 A — polling-station arrival times

[245, 120, 375, 180, 320, 150]

(1) **The role of the pivot.** The pivot is the reference value against which every other element is compared during partitioning. Elements $\leq$ the pivot are moved to its left and elements $>$ the pivot to its right. When partitioning ends, the pivot is at its *final sorted position* — the split point — and is never moved again. The array is then divided at that point and the two sides are sorted recursively and independently.

(2) **First two recursive levels** (last element as pivot, Lomuto partition).

Level 0. Range [0..5], pivot = 150. Scanning: 245 > 150 (no move), 120 $\leq$ 150 (moves left), 375 > 150, 180 > 150, 320 > 150. Then the pivot is swapped into place.

Before	[245, 120, 375, 180, 320, 150]
After	[120, 150 , 375, 180, 320, 245]
Split point	index 1
Left partition	[120] — size 1, already sorted
Right partition	[375, 180, 320, 245]

Level 1. Range [2..5] = [375, 180, 320, 245], pivot = 245. Scanning: 375 > 245, 180 $\leq$ 245 (moves left), 320 > 245. Pivot swapped into place.

Before	[375, 180, 320, 245]
After	[180, 245 , 320, 375]
Split point	index 3 (global)
Left partition	[180] — sorted
Right partition	[320, 375]

Global array after two levels: [120, 150, 180, 245, 320, 375] — already sorted, though the algorithm does not know this and performs one more partition on [320,375] with pivot 375.

(3) Worst-case comparisons for 5000 records. In the worst case each partition removes one element:

$$(n - 1) + (n - 2) + \cdots + 1 = \frac{n(n - 1)}{2} = \frac{5000 \times 4999}{2} = \mathbf{12,497,500}$$

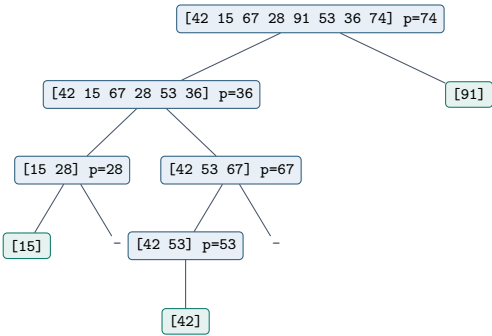
comparisons — approximately 1.25×10^7 , or $\Theta(n^2)$. For contrast, the average case is about $1.39 n \lg n \approx 8.5 \times 10^4$ comparisons: a factor of roughly 150 worse.

(4) One technique to avoid the worst case. *Randomised pivot selection* (or median-of-three). Choosing the pivot at random makes the probability of repeatedly hitting the extreme element vanishingly small, so the expected running time is $\Theta(n \lg n)$ *regardless of the input arrangement*. This matters here specifically because election records arriving ordered by time are likely to be *already nearly sorted* — precisely the input that triggers quick sort’s worst case with a first- or last-element pivot.

Worked Example 7.6 | Lab Sheet 09 Q2D — full quick sort recursion tree

[42, 15, 67, 28, 91, 53, 36, 74], last element as pivot

Level	Subarray (indices)	Pivot	After partition	Split at
0	[0..7]: 42 15 67 28 91 53 36 74	74	42 15 67 28 53 36 74 91	index 6
1	[0..5]: 42 15 67 28 53 36	36	15 28 36 42 53 67	index 2
1	[7..7]: 91	—	base case	—
2	[0..1]: 15 28	28	15 28	index 1
2	[3..5]: 42 53 67	67	42 53 67	index 5
3	[0..0]: 15	—	base case	—
3	[3..4]: 42 53	53	42 53	index 4
4	[3..3]: 42	—	base case	—



Final sorted array: [15, 28, 36, 42, 53, 67, 74, 91].

Notice that the tree is *lopsided* — depth 4 rather than the ideal 3 — because the last-element pivots happened to sit off-centre. This is the behaviour that a randomised or median-of-three pivot is designed to prevent.

Worked Example 7.7 | Lab Sheet 09 Q2E — counting the merge sort recursion structure

For the same 8-element array [42, 15, 67, 28, 91, 53, 36, 74] under merge sort:

(I) Number of levels in the recursion tree. Each level halves the subarray size: $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. The number of *division* steps is $\lg 8 = 3$, so the tree has $\lg n + 1 = 4$ levels counting the root level (size 8) and the leaf level (size 1).

(II) Total number of subarrays created during the divide phase. Level 0: 1 subarray of size 8. Level 1: 2 of size 4. Level 2: 4 of size 2. Level 3: 8 of size 1.

$$1 + 2 + 4 + 8 = 2^{\lg n + 1} - 1 = 2n - 1 = \mathbf{15} \text{ subarrays.}$$

(III) Number of merge operations. Every internal node of the tree corresponds to one merge. There are $15 - 8 = 7$ internal nodes, i.e. $n - 1 = 7$ merge operations: 4 merges of size-1 pairs, 2 merges of size-2 pairs, 1 final merge of the two halves.

General result: for n a power of two, merge sort creates $2n - 1$ subarrays and performs $n - 1$ merges across $\lg n + 1$ levels.

7.5 Strassen's matrix multiplication

7.5.1 The conventional algorithm

For $n \times n$ matrices A and B , the product $C = AB$ has entries

$$c_{kl} = \sum_{m=1}^n a_{km} b_{ml}.$$

Conventional matrix multiplication

```

1  procedure matmul(A[1..n][1..n], B[1..n][1..n], C[1..n][1..n])
2    for i ← 1 to n do
3      for j ← 1 to n do
4        C[i][j] ← 0
5        for k ← 1 to n do
6          C[i][j] ← C[i][j] + A[i][k] * B[k][j]    // BASIC OPERATION
7        end for
8      end for
9    end for
10 end procedure

```

Three nested loops of n iterations each: n^3 multiplications and $n^3 - n^2$ additions. $T(n) = \Theta(n^3)$.

Rectangular matrices

For A of size $n \times m$ and B of size $m \times p$, the product C is $n \times p$ and the cost is $\Theta(nmp)$ — one multiplication for each of the nmp triples (i, k, j) . This answers the exercise posed in Handout 4(I).

7.5.2 The divide-and-conquer version

Partition each $n \times n$ matrix into four $\frac{n}{2} \times \frac{n}{2}$ blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then block multiplication gives

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} & C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} & C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

That is **8** multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and **4** additions, giving

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Master Theorem: $n^{\log_2 8} = n^3$, and $f(n) = n^2 = O(n^{3-\epsilon})$, so Case 1 applies and $T(n) = \Theta(n^3)$.
No improvement — the recursive version is exactly as slow as the loop version.

7.5.3 Strassen's insight

Volker Strassen (1969) found a way to obtain all four blocks of C using only **seven** multiplications instead of eight, at the cost of extra additions and subtractions (which are cheap: $\Theta(n^2)$, versus $\Theta(n^3)$ for multiplication).

Strassen's seven products

$$\begin{aligned} P_1 &= A_{11}(B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12})B_{22} \\ P_3 &= (A_{21} + A_{22})B_{11} \\ P_4 &= A_{22}(B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21})(B_{11} + B_{12}) \end{aligned}$$

and then

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 & C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 & C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

	Multiplications	Additions/subtractions	Recurrence
Conventional (recursive)	8	4	$T(n) = 8T(n/2) + \Theta(n^2)$
Strassen	7	18	$T(n) = 7T(n/2) + \Theta(n^2)$

Master Theorem with $a = 7$, $b = 2$: $n^{\log_2 7} = n^{2.807\dots}$, and $f(n) = n^2 = O(n^{2.807-\epsilon})$, so Case 1 gives

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$$

Key idea

Trading one multiplication for fourteen extra additions looks like a bad deal — and at one level it is. The point is that the saving *compounds through the recursion*. Removing one of eight recursive calls at every level changes the number of leaves from $8^{\lg n} = n^3$ to $7^{\lg n} = n^{2.807}$. Additions are only $\Theta(n^2)$ work, which is dominated.

n	n^3 (conventional)	$n^{\lg 7}$ (Strassen)	Ratio
2	8	7	1.14
4	64	49	1.31
8	512	343	1.49
16	4,096	2,401	1.71
64	262,144	117,649	2.23
1024	1.07×10^9	2.82×10^8	3.80

Worked Example 7.8 | Strassen on 2×2 matrices, in full

$$A = \begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}, \quad B = \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

Step 1 — the seven products.

Product	Expression	Value
$P_1 = A_{11}(B_{12} - B_{22})$	$1 \times (8 - 2)$	6
$P_2 = (A_{11} + A_{12})B_{22}$	$(1 + 3) \times 2$	8
$P_3 = (A_{21} + A_{22})B_{11}$	$(7 + 5) \times 6$	72
$P_4 = A_{22}(B_{21} - B_{11})$	$5 \times (4 - 6)$	-10
$P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$	$(1 + 5) \times (6 + 2)$	48
$P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$	$(3 - 5) \times (4 + 2)$	-12
$P_7 = (A_{11} - A_{21})(B_{11} + B_{12})$	$(1 - 7) \times (6 + 8)$	-84

Step 2 — assemble C .

$$C_{11} = P_5 + P_4 - P_2 + P_6 = 48 - 10 - 8 - 12 = 18$$

$$C_{12} = P_1 + P_2 = 6 + 8 = 14$$

$$C_{21} = P_3 + P_4 = 72 - 10 = 62$$

$$C_{22} = P_5 + P_1 - P_3 - P_7 = 48 + 6 - 72 + 84 = 66$$

$$C = \begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$$

Step 3 — verify against the conventional method.

$$c_{11} = 1(6) + 3(4) = 18 \quad \checkmark$$

$$c_{12} = 1(8) + 3(2) = 14 \quad \checkmark$$

$$c_{21} = 7(6) + 5(4) = 62 \quad \checkmark$$

$$c_{22} = 7(8) + 5(2) = 66 \quad \checkmark$$

Operation count for this example.

	Conventional	Strassen
Multiplications	8	7
Additions/subtractions	4	18

For a 2×2 example Strassen is clearly *worse* in total operations. It becomes advantageous only when applied recursively to large matrices, because the saved multiplication is a saved *recursive call* on an $\frac{n}{2} \times \frac{n}{2}$ matrix — an $\Theta(n^{2.807})$ -sized saving — whereas the extra additions cost only $\Theta(n^2)$.

Worked Example 7.9 | Conventional multiplication, 3×3 — Lab Sheet 09 Q3(a) method

$$A = \begin{pmatrix} 2 & 1 & 3 \\ 0 & 4 & 1 \\ 5 & 2 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 & 2 \\ 3 & 1 & 1 \\ 0 & 2 & 4 \end{pmatrix}$$

Each entry $c_{ij} = \sum_{k=1}^3 a_{ik}b_{kj}$:

$$\begin{aligned} c_{11} &= 2(1) + 1(3) + 3(0) = 5 & c_{12} &= 2(0) + 1(1) + 3(2) = 7 & c_{13} &= 2(2) + 1(1) + 3(4) = 17 \\ c_{21} &= 0(1) + 4(3) + 1(0) = 12 & c_{22} &= 0(0) + 4(1) + 1(2) = 6 & c_{23} &= 0(2) + 4(1) + 1(4) = 8 \\ c_{31} &= 5(1) + 2(3) + 0(0) = 11 & c_{32} &= 5(0) + 2(1) + 0(2) = 2 & c_{33} &= 5(2) + 2(1) + 0(4) = 12 \end{aligned}$$

$$C = \begin{pmatrix} 5 & 7 & 17 \\ 12 & 6 & 8 \\ 11 & 2 & 12 \end{pmatrix}$$

Operation count: $3^3 = 27$ multiplications and $3^3 - 3^2 = 18$ additions.

7.5.4 Comparison and limitations

Aspect	Conventional	Strassen
Divide and conquer	Can be written recursively (8 subproblems) but gains nothing	7 subproblems — the reduction <i>is</i> the algorithm
Multiplications	n^3	$n^{\log_2 7} \approx n^{2.807}$
Additions	$n^3 - n^2$	More per level (18 vs. 4), but only $\Theta(n^2)$ overall
Time complexity	$\Theta(n^3)$	$\Theta(n^{2.807})$
Numerical stability	Good	Worse — subtractions can amplify floating-point error
Practical crossover	Better for small n	Better only above roughly $n = 64$ – 128
Matrix size	Any	Requires n to be a power of 2 (pad with zeros otherwise)
Memory	Modest	Higher — many intermediate submatrices

Why Strassen is rarely used in production

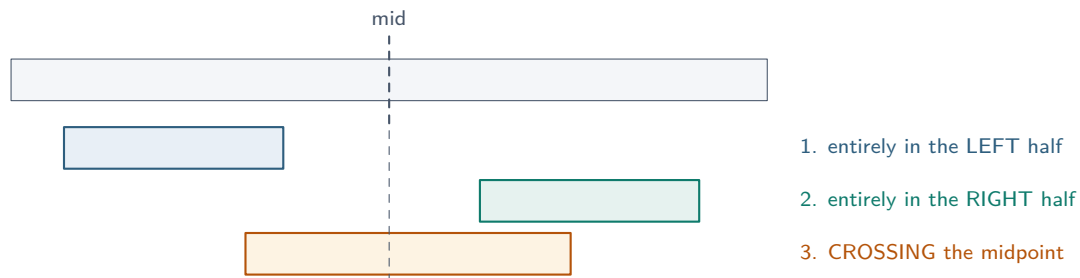
Three reasons: (i) the constant factor and the extra additions mean it loses to a well-tuned conventional implementation for the matrix sizes most programs actually use; (ii) the subtractions degrade numerical stability for floating-point data; (iii) conventional multiplication has excellent cache behaviour and maps well onto vector hardware, which Strassen's irregular structure does not. It remains theoretically important as the first proof that $\Theta(n^3)$ was not optimal.

7.6 Divide and conquer for the maximum subarray problem

[Lab Sheet 07 Q3. Continues from Chapter 6 §6.9.]

7.6.1 The idea

Split the array at the midpoint. The maximum subarray must be in exactly one of three places:



Cases 1 and 2 are solved recursively. Case 3 is solved in linear time: the best crossing subarray is the best suffix of the left half joined to the best prefix of the right half, and both can be found with a single scan outward from the midpoint.

Algorithm 7.4 — Maximum subarray, divide and conquer

```

1  function maxSubarray(A, low, high)
2      if low = high then
3          return A[low]                                // BASE CASE
4      end if
5      mid ← floor((low + high) / 2)
6      leftMax ← maxSubarray(A, low, mid)                // CONQUER left
7      rightMax ← maxSubarray(A, mid+1, high)            // CONQUER right
8      crossMax ← maxCrossing(A, low, mid, high)         // COMBINE
9      return max(leftMax, rightMax, crossMax)
10 end function
11
12 function maxCrossing(A, low, mid, high)
13     sum ← 0; leftBest ← -infinity
14     for i ← mid downto low do                          // scan left from mid
15         sum ← sum + A[i]
16         if sum > leftBest then leftBest ← sum
17     end for
18     sum ← 0; rightBest ← -infinity
19     for j ← mid+1 to high do                          // scan right from mid

```

```

20     sum ← sum + A[j]
21     if sum > rightBest then rightBest ← sum
22   end for
23   return leftBest + rightBest
24 end function

```

7.6.2 Complexity

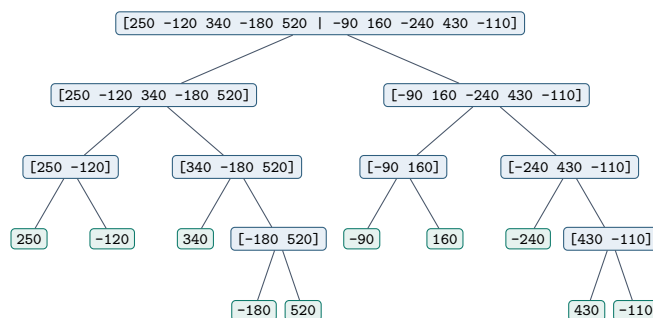
$$T(n) = \underbrace{2T(n/2)}_{\text{two halves}} + \underbrace{\Theta(n)}_{\text{crossing scan}} \implies T(n) \in \Theta(n \lg n)$$

by Master Theorem Case 2 — the same recurrence as merge sort.

Worked Example 7.10 | Lab Sheet 07 Q3 — divide and conquer on the profit data

$A = [250, -120, 340, -180, 520, -90, 160, -240, 430, -110]$, $n = 10$

(B) Recursion tree.



(C) The top-level computation. The midpoint splits the array as $L = A[1..5]$ and $R = A[6..10]$.

Left half $L = [250, -120, 340, -180, 520]$. The best contiguous sum is the whole of L : $250 - 120 + 340 - 180 + 520 = \mathbf{810}$. (Every proper subarray is smaller: e.g. $[340, -180, 520] = 680$, $[520] = 520$.)

Right half $R = [-90, 160, -240, 430, -110]$. The best is $[430]$ alone = **430**. (Check: $[160] = 160$; $[160, -240, 430] = 350$; $[430, -110] = 320$.)

Crossing the midpoint. Best suffix of L , scanning left from index 5: 520 ; $520 - 180 = 340$; $340 + 340 = 680$; $680 - 120 = 560$; $560 + 250 = 810$. Maximum = **810** (the whole left half).

Best prefix of R , scanning right from index 6: -90 ; $-90 + 160 = 70$; $70 - 240 = -170$; $-170 + 430 = 260$; $260 - 110 = 150$. Maximum = **260** (i.e. $A[6..9]$).

Crossing maximum = $810 + 260 = \mathbf{1070}$.

Combine.

$$\max(\underbrace{810}_{\text{left}}, \underbrace{430}_{\text{right}}, \underbrace{1070}_{\text{crossing}}) = \boxed{1070}$$

achieved by $A[1..9]$ — exactly the answer the brute-force method produced in Chapter 6. ✓

(D) Time complexity. $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$. It beats brute force because the crossing scan is linear and is performed only $\lg n$ times per element, instead of every subarray being examined independently.

(E) Comparison.

	Brute force	Divide and conquer
Main idea	Enumerate every contiguous subarray and sum it	Split at the midpoint; the answer lies left, right, or across
Complexity	$\Theta(n^3)$ naive, $\Theta(n^2)$ with a running sum	$\Theta(n \lg n)$
Small datasets	Perfectly fine; lower constant factor	Recursion overhead may make it slower
Large datasets	Impractical — $n = 10^4$ needs 10^8 operations	Comfortable — $n = 10^4$ needs $\approx 1.3 \times 10^5$
Advantages	Trivial to write and verify; no recursion	Far better scaling; parallelisable halves
Limitations	Quadratic or cubic growth	More complex; still beaten by Kadane's $\Theta(n)$

7.7 Further divide-and-conquer algorithms

Algorithm	Complexity	Divide / combine
Binary search	$\Theta(\lg n)$	Divide by comparison; no combine
Merge sort	$\Theta(n \lg n)$	Divide by position; combine by merging
Quick sort	$\Theta(n \lg n)$ avg	Divide by partitioning; no combine
Strassen	$\Theta(n^{2.807})$	Divide into blocks; combine by additions
Maximum subarray	$\Theta(n \lg n)$	Divide at midpoint; combine via crossing scan
Closest pair of points	$\Theta(n \lg n)$	Divide by a vertical line; combine by checking a strip
Karatsuba integer multiplication	$\Theta(n^{1.585})$	Split digits; 3 multiplications instead of 4
Fast Fourier Transform	$\Theta(n \lg n)$	Split by even/odd index; combine with twiddle factors
Tower of Hanoi	$\Theta(2^n)$	Move $n - 1$, move 1, move $n - 1$

Key idea

Karatsuba and Strassen share the same trick, and it is worth naming: *reduce a , the number of recursive calls*, even at the cost of more cheap work in $f(n)$. Since the answer is $\Theta(n^{\log_b a})$ whenever the leaves dominate, shaving one recursive call changes the exponent itself.

Chapter summary

- **Divide and conquer** = divide into *independent* subproblems, conquer recursively, combine. Top-down.
- Recurrence shape $T(n) = aT(n/b) + f(n)$; solve with the Master Theorem.
- **Binary search**: $a = 1$, no combine, $\Theta(\lg n)$.
- **Merge sort**: work in the *combine*; $\Theta(n \lg n)$ in every case; stable; *not* in-place ($O(n)$ auxiliary); ideal for linked lists, external sorting and parallelisation.
- **Quick sort**: work in the *divide*; $\Theta(n \lg n)$ average but $O(n^2)$ worst; *in-place*; *not* stable;

fastest in practice on arrays. Use randomised or median-of-three pivots.

- **Stable** $\neq$ **in-place**: they are independent properties.
- **Strassen**: 7 multiplications instead of 8 per level $\Rightarrow \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$. Numerically less stable; worth it only for large n .
- **Maximum subarray**: left, right, or crossing $\Rightarrow T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$. Still beaten by Kadane's $\Theta(n)$.
- Divide and conquer fails when subproblems *overlap* — use dynamic programming instead.

CHAPTER 8

Dynamic Programming

[Sources: Handout 6 (complete); Lab Sheets 08 and 09]

8.1 The idea

Dynamic programming

An optimisation technique that solves a problem by breaking it into *overlapping* subproblems, solving each subproblem exactly once, and storing its solution so that it can be reused instead of recomputed.

The name is historical — Richard Bellman coined it in the 1950s, choosing “programming” in the sense of *scheduling* or *planning*, not coding. Nothing about the technique involves programming languages.

Key idea

Dynamic programming is what you get when you take recursion and add *memory*. The entire gain comes from never solving the same subproblem twice.

8.1.1 The jigsaw analogy

Solving a jigsaw	The DP concept
Group pieces by colour or pattern (edges, sky, trees)	Decomposition into subproblems
Once the sky is finished, set it aside and never redo it	Memoisation — store the solved subproblem
Combine finished sections into larger ones	Tabulation — build up from small to large
Never re-test a piece you have already placed	Avoiding redundant computation

Randomly trying pieces to see what fits is brute force. Systematically grouping, solving and remembering is dynamic programming.

8.2 The two conditions

Dynamic programming applies when — and only when — the problem has both of the following properties.

Optimal substructure

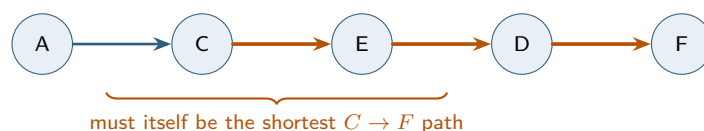
An optimal solution to the problem can be constructed from optimal solutions to its subproblems.

Overlapping subproblems

The recursive decomposition produces the *same* subproblems repeatedly, so that storing their solutions saves computation.

8.2.1 Optimal substructure, illustrated

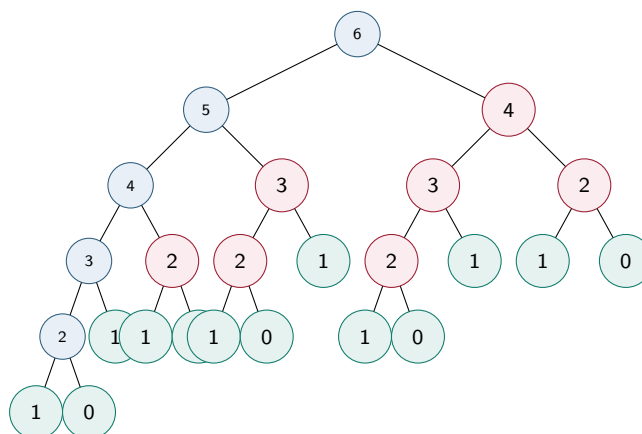
The shortest-path problem has it. If the shortest path from A to F passes through C , then the portion of that path from C to F must itself be a shortest path from C to F .



Why: if there were a shorter path from C to F , substituting it would give a shorter path from A to F , contradicting the assumption that the original was optimal. Standard algorithms exploiting this include Floyd–Warshall and Bellman–Ford.

8.2.2 Overlapping subproblems, illustrated

The recursion tree for $\text{fib}(6)$ (Chapter 2 §2.4) recomputes $\text{fib}(3)$ three times and $\text{fib}(2)$ five times. Computing $\text{fib}(50)$ naively requires roughly 4×10^{10} calls — and there are only 50 distinct subproblems.



Red = a subproblem already computed elsewhere in the tree.

Not every recursive problem is a DP problem

Recursion is necessary but not sufficient. If the subproblems are *independent* — as in merge sort and quick sort — there is nothing to reuse, and divide and conquer is the correct

paradigm. Merge sort and quick sort are not dynamic programming problems, even though both are recursive.

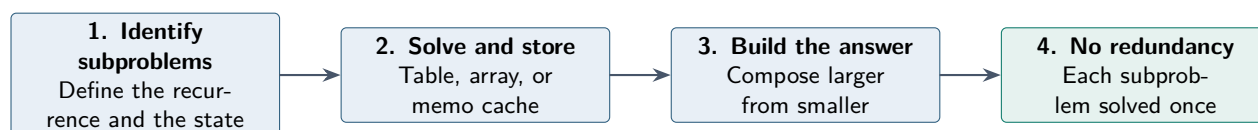
8.3 Divide and conquer versus dynamic programming

	Divide and conquer	Dynamic programming
Subproblems	Independent — disjoint, no shared work	Overlapping — the same subproblems recur
Approach	Split, solve separately, combine	Solve each subproblem once, store, reuse
Reuse	None — past results are discarded	Central — past results are the whole point
Solution built from	Combining independent results	Stored solutions of smaller overlapping subproblems
Direction	Top-down only	Top-down (memoisation) <i>or</i> bottom-up (tabulation)
Memory	Lower — usually only the recursion stack	Higher — a table or cache is required
Repeated work	Occurs if subproblems overlap	Eliminated by construction
Typical use	Sorting, searching, matrix multiplication	Optimisation and counting problems
Examples	Merge sort, quick sort, binary search, Strassen	Fibonacci, knapsack, LCS, shortest paths
Implementation	Generally simpler	Generally more complex

Key idea

One question decides between them: *do the subproblems share work?* If splitting the problem produces subproblems that never look at the same data, use divide and conquer. If it produces the same subproblem repeatedly, use dynamic programming.

8.4 The DP procedure



Recognising a DP problem

Signs that dynamic programming applies:

1. **Overlapping subproblems** — the same subproblem appears many times.
2. **Optimal substructure** — the optimum is built from sub-optima.
3. **Naturally recursive** — the problem is easy to express recursively.
4. **Memoisation opportunity** — a recursive solution is obviously improved by caching.

5. **Sequential dependencies** — the answer depends on results of earlier steps (sequences, strings, arrays, graphs).
6. **Optimisation or counting** — “maximise”, “minimise”, “how many ways”.

8.5 The two implementations

Memoisation — top-down

Start from the original problem and recurse downward.
 Before computing a subproblem, check the cache.
 If cached, return immediately; otherwise compute, store, return.
“Recursion with a memory.”

Tabulation — bottom-up

Start from the smallest subproblems.
 Fill a table iteratively, in order of increasing size.
 Stop when the table entry for the original problem is filled.
“Build the answer piece by piece.”

8.5.1 Memoisation (top-down)

Memoisation

From the Latin *memorandum* (“to be remembered”), shortened to *memo*. The technique of caching the result of a function call so that a later call with the same arguments returns the stored value instead of recomputing it.

How it works:

1. A recursive function calls itself on smaller subproblems.
2. Before recursing, it checks whether the result is already in the cache.
3. A base case terminates the recursion.
4. If the result is not cached, it is computed, stored, and then returned.

Fibonacci by memoisation

```

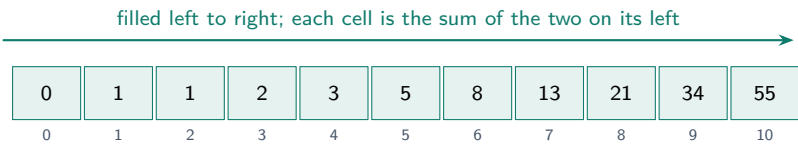
1 memo ← empty map
2
3 function fib(n)
4   if n ≤ 1 then return n           // BASE CASE
5   if memo contains n then return memo[n] // CACHE HIT -- no recomputation
6   result ← fib(n-1) + fib(n-2)      // compute
7   memo[n] ← result                 // store
8   return result
9 end function

```

8.5.2 Tabulation (bottom-up)

Fibonacci by tabulation

```
1 function fib(n)
2   int table[0..n]
3   table[0] ← 0           // BASE CASES filled directly
4   table[1] ← 1
5   for i ← 2 to n do      // ITERATIVE, increasing order
6     table[i] ← table[i-1] + table[i-2]
7   end for
8   return table[n]
9 end function
```



8.5.3 Memoisation versus tabulation

	Memoisation	Tabulation
Direction	Top-down	Bottom-up
Mechanism	Recursion	Iteration (loops)
Storage	Cache filled as results are computed	Table filled from smallest to largest
Coverage	Solves only the subproblems actually needed	Usually computes all subproblems
Overhead	Function-call and stack overhead	None — no recursion
Speed	Slower by a constant factor	Usually faster
Stack risk	Can overflow for deep recursion	No recursion, so no risk
When preferred	When many subproblems are never needed (sparse dependency)	When all subproblems must be solved anyway
Ease of writing	Often easier — mirrors the recurrence directly	Requires working out a correct fill order

The distinguishing feature is recursion, not the data structure

Memoisation may use an array; tabulation may use a hash map. That does not make one the other. **The difference is that memoisation computes values by recursion and tabulation by iteration.** Everything else is incidental.

Key idea

Is memoisation the same as caching? Memoisation is a *specific kind* of caching. “Caching” covers any storage of data for future use (HTTP caches, disk caches); memoisation specifically means caching a *function’s return value keyed on its arguments*.

8.5.4 Dimensions of memoisation

The number of cache dimensions equals the number of parameters that change between recursive calls.

Type	Changing parameters	Example
1D	One	Fibonacci; climbing stairs; coin change (amount only)
2D	Two	Longest common subsequence of two strings; 0/1 knapsack (item index, remaining capacity); edit distance
3D	Three	Longest common subsequence of <i>three</i> strings; some interval-DP variants

8.6 Fibonacci: the complete DP treatment

Method	Time	Space	Calls for $n = 50$
Naive recursion	$\Theta(\varphi^n) \approx \Theta(1.618^n)$	$O(n)$ stack	$\approx 4 \times 10^{10}$
Memoisation	$\Theta(n)$	$O(n)$ cache + $O(n)$ stack	99
Tabulation	$\Theta(n)$	$O(n)$ table	0 (iterative)
Tabulation, two variables	$\Theta(n)$	$O(1)$	0

Fibonacci with $O(1)$ space — the final optimisation

```
1 function fib(n)
2   if n ≤ 1 then return n
3   a ← 0; b ← 1
4   for i ← 2 to n do
5     c ← a + b; a ← b; b ← c    // only the last two values are ever needed
6   end for
7   return b
8 end function
```

Key idea

Reducing exponential time to linear is the headline. But notice the second optimisation: because the recurrence only looks back two positions, the full table is unnecessary and $O(1)$ space suffices. *Always ask how far back the recurrence reaches* — it often collapses the space requirement dramatically.

Worked Example 8.1 | Lab Sheet 08 D — report transmission paths

The number of transmission paths satisfies

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

(1) Overlapping subproblems, and why DP beats plain recursion. A subproblem overlaps when the identical subproblem instance is required by more than one branch of the

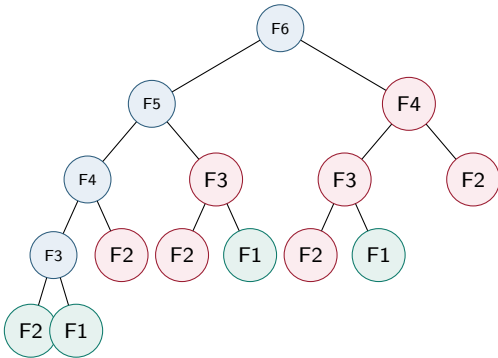
recursion. Here $F(5)$ is needed to compute both $F(6)$ and $F(7)$, $F(4)$ is needed by $F(5)$ and $F(6)$, and so on. A plain recursive solution recomputes each from scratch, giving $\Theta(\varphi^n)$ work for only n distinct subproblems. Dynamic programming stores each result on first computation, so total work becomes $\Theta(n)$.

(2) $F(8)$ by tabulation.

n	1	2	3	4	5	6	7	8
$F(n)$	1	2	3	5	8	13	21	34
Computed as	base	base	2+1	3+2	5+3	8+5	13+8	21+13

$F(8) = 34$

(3) Recursion tree for $F(6)$ and its repeated subproblems.



Three (or more) repeated subproblems, as required:

- $F(4)$ is computed **twice** — once under $F(5)$, once directly under $F(6)$.
- $F(3)$ is computed **three times**.
- $F(2)$ is computed **five times**.
- $F(1)$ is computed **three times**.

(4) Complexity comparison.

	Complexity	Reason
Naive recursion	$\Theta(\varphi^n)$, $\varphi \approx 1.618$	Each call spawns two more; the tree has exponentially many nodes
Dynamic programming	$\Theta(n)$ time, $\Theta(n)$ space (or $\Theta(1)$ with two variables)	Each of the n distinct subproblems is evaluated exactly once

DP is more efficient because the number of *distinct* subproblems is only n , while the number of *calls* in the naive version is exponential. Storing each result converts repeated recomputation into a single table lookup.

Worked Example 8.2 | Lab Sheet 09 Q4 — warehouse robot rack planning

A robot moves either one or two racks at a time. $R(n)$ is the number of ways to reach rack

n :

$$R(n) = R(n-1) + R(n-2), \quad R(1) = 1, R(2) = 2.$$

(A) Why DP rather than divide and conquer.

- *Independent subproblems* (divide and conquer's requirement) means the subproblems share no work — as in merge sort, where the two halves are disjoint. That is *not* the case here: computing $R(n-1)$ requires $R(n-2)$, which is also required directly by $R(n)$.
- *Overlapping subproblems* is exactly what this recurrence exhibits: the dependency graph of R is a chain in which every node feeds two successors. Divide and conquer would recompute these; DP stores them.
- *Optimal substructure* holds: the number of ways to reach rack n is determined entirely by the counts for racks $n-1$ and $n-2$, with no dependence on how those were reached.

Divide and conquer would give $\Theta(\varphi^n)$; DP gives $\Theta(n)$.

(B) Recursion tree for $R(6)$. Structurally identical to the $F(6)$ tree above. Repeated subproblems: $R(4)$ twice, $R(3)$ three times, $R(2)$ five times.

(C) $R(8)$ by memoisation (top-down). Call sequence, with cache hits marked:

Call	Action	Result
$R(8)$	needs $R(7)$, $R(6)$	—
$R(7)$	needs $R(6)$, $R(5)$	—
$R(6)$	needs $R(5)$, $R(4)$	—
$R(5)$	needs $R(4)$, $R(3)$	—
$R(4)$	needs $R(3)$, $R(2)$	—
$R(3)$	needs $R(2)$, $R(1)$	—
$R(2)$	base case	2, stored
$R(1)$	base case	1, stored
$R(3)$	returns $2 + 1$	3, stored
$R(4)$	$R(3) = 3$ (cache hit) + $R(2) = 2$ (hit)	5, stored
$R(5)$	$5 + 3$ (both hits)	8, stored
$R(6)$	$8 + 5$ (both hits)	13, stored
$R(7)$	$13 + 8$ (both hits)	21, stored
$R(8)$	$21 + 13$ (both hits)	34

Final memoisation table:

n	1	2	3	4	5	6	7	8
memo [n]	1	2	3	5	8	13	21	34

Without memoisation the naive recursion makes **41** function invocations to reach $R(8)$; with memoisation it makes **13**, of which only 6 do any arithmetic — the rest are base cases or cache hits.

(D) $R(8)$ by tabulation (bottom-up).

n	1	2	3	4	5	6	7	8
$dp[n]$	1	2	3	5	8	13	21	34
From	base	base	$dp_2 + dp_1$	$dp_3 + dp_2$	$dp_4 + dp_3$	$dp_5 + dp_4$	$dp_6 + dp_5$	$dp_7 + dp_6$

$R(8) = 34$

Which approach is more suitable here? Tabulation. Every subproblem $R(1) \dots R(n)$ is required to compute $R(n)$ — none can be skipped — so memoisation’s ability to solve only the needed subproblems buys nothing, while its recursion overhead and stack-overflow risk remain. Tabulation is iterative, has no call overhead, and can be reduced to $O(1)$ space by keeping only the last two values.

8.7 Longest Common Subsequence

Longest Common Subsequence (LCS)

Given two sequences X and Y , find the longest sequence that appears as a subsequence (not necessarily contiguous) of both.

The recurrence, for $X[1..i]$ and $Y[1..j]$:

$$L[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i - 1][j - 1] + 1 & \text{if } X[i] = Y[j] \\ \max(L[i - 1][j], L[i][j - 1]) & \text{otherwise} \end{cases}$$

Worked Example 8.3 | $X = \text{ABCB DAB}, Y = \text{BDCABA}$

		Y					
		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$L[7][6] = 4$. One LCS of length 4 is **BCBA** (also BDAB, BCAB).
Complexity: $\Theta(mn)$ time and $\Theta(mn)$ space — versus $\Theta(2^m)$ for the brute-force approach of testing every subsequence of X against Y .

Key idea

LCS is a **2D memoisation** problem: two parameters (i and j) change between recursive calls, so the cache is a two-dimensional table. Extending to three strings gives a 3D table and $\Theta(mnp)$ complexity.

Applications: file-difference tools (`diff`), version control merges, DNA and protein sequence alignment, plagiarism detection, spell checking (the closely related edit-distance problem).

8.8 The 0/1 Knapsack problem

0/1 Knapsack

Given n items, item i having weight w_i and value v_i , and a knapsack of capacity W , choose a subset of items maximising total value subject to total weight $\leq W$. Each item may be taken *entirely or not at all* — no fractions, and exactly one copy of each item is available.

This is the variant greedy cannot solve

The *fractional* knapsack (Chapter 9 §9.6.1) yields to a greedy algorithm. The 0/1 knapsack does not: the indivisibility destroys the greedy-choice property. Dynamic programming guarantees the optimum for 0/1; greedy does not.

The recurrence, where $K[i][c]$ is the best value using the first i items with capacity c :

$$K[i][c] = \begin{cases} 0 & \text{if } i = 0 \text{ or } c = 0 \\ K[i-1][c] & \text{if } w_i > c \text{ (item does not fit)} \\ \max(\underbrace{K[i-1][c]}_{\text{skip item } i}, \underbrace{K[i-1][c-w_i] + v_i}_{\text{take item } i}) & \text{otherwise} \end{cases}$$

Worked Example 8.4 | 0/1 knapsack, $W = 7$

Item	Weight w_i	Value v_i
1	1	1
2	3	4
3	4	5
4	5	7

DP table (rows = items considered, columns = capacity):

$i \backslash c$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4	5	6	6	9
4	0	1	1	4	5	7	8	9

Reading the answer. $K[4][7] = 9$.

Recovering the chosen items (trace backwards from $K[4][7]$): $K[4][7] = K[3][7] = 9$, so item 4 was *not* taken. $K[3][7] = 9 \neq K[2][7] = 5$, so item 3 *was* taken; move to $K[2][7-4] = K[2][3] = 4$. $K[2][3] = 4 \neq K[1][3] = 1$, so item 2 *was* taken; move to $K[1][3-3] = K[1][0] = 0$. Stop.

Selected: items 2 and 3, total weight $3 + 4 = 7$, total value $4 + 5 = 9$.

Why greedy fails here. By value-per-weight, item 1 ranks 1.0, item 2 ranks 1.33, item 3 ranks 1.25, item 4 ranks 1.4. Greedy takes item 4 first (weight 5, value 7), leaving capacity 2, into which only item 1 fits — total value **8**, one short of the optimum 9.

Complexity: $\Theta(nW)$ time and $\Theta(nW)$ space.

$\Theta(nW)$ is “pseudo-polynomial”, not polynomial

W is a *value*, not an input size. Writing W requires only $\lg W$ bits, so $\Theta(nW)$ is exponential in the number of bits of the input. This is why 0/1 knapsack is NP-hard (Chapter 12) despite having a table-filling algorithm.

8.9 Coin change by dynamic programming

Minimum-coin change

Given denominations $C = \{c_1, \dots, c_k\}$ (unlimited supply of each) and a target amount W , find the smallest number of coins summing to exactly W .

$$dp[a] = \begin{cases} 0 & a = 0 \\ 1 + \min_{c \in C, c \leq a} dp[a - c] & a > 0 \end{cases}$$

Worked Example 8.5 | $C = \{1, 4, 5\}$, $W = 8$

Amount a	0	1	2	3	4	5	6	7	8
$dp[a]$	0	1	2	3	1	1	2	3	2
Coin used	—	1	1	1	4	5	1	1	4

Working through $a = 8$: $dp[8] = 1 + \min(dp[7], dp[4], dp[3]) = 1 + \min(3, 1, 3) = 1 + 1 = 2$, via the coin 4. Tracing back: $8 \rightarrow 4 \rightarrow 0$, so the solution is $\{4, 4\}$.
The greedy algorithm gets this wrong. Greedy takes the largest coin ≤ 8 , namely 5, leaving 3, then three 1s: $\{5, 1, 1, 1\}$ — **4 coins**. DP finds $\{4, 4\}$ — **2 coins**.
Complexity: $\Theta(W \cdot |C|)$ time, $\Theta(W)$ space.

Consolidation note

The greedy version of coin change, and the conditions under which it *does* give the optimum, are covered in Chapter 9 §9.4.

8.10 Advantages, disadvantages and applications

Advantages	Disadvantages
Guarantees the optimal solution when optimal substructure holds	Increased memory usage — the table may be large
Eliminates redundant computation — often exponential $\rightarrow$ polynomial	Harder to design — finding the right state and recurrence is the difficult part
Simplifies complex problems into a clear recurrence	Depends on problem structure — useless without overlapping subproblems
Solutions are reusable and easy to verify by table inspection	Computes subproblems that may never be needed (tabulation)

Algorithm / application	What DP contributes
Longest Common Subsequence	$\Theta(mn)$ instead of exponential subsequence enumeration
Floyd–Warshall (all-pairs shortest paths)	$\Theta(V ^3)$ by reusing intermediate-vertex results
Bellman–Ford (single-source shortest paths)	Handles negative edge weights by relaxing edges $ V - 1$ times
0/1 Knapsack	$\Theta(nW)$ instead of $\Theta(2^n)$ subset enumeration
Matrix chain multiplication	Optimal parenthesisation in $\Theta(n^3)$
Edit distance / spellcheckers	Minimum insert/delete/substitute operations
DNA sequence alignment	Needleman–Wunsch and Smith–Waterman are DP
Google Maps route computation	Shortest-path computation over a road network
Compilers	Optimal register allocation
Stock trading algorithms	Optimal buy/sell timing
Resource allocation and budgeting	Knapsack-style constrained optimisation
Project scheduling; Tower of Hanoi	Optimal orderings and move counts

Frequently asked short questions

Should I use tabulation or memoisation?

Tabulation when *all* subproblems must be solved anyway — it avoids recursion overhead. Memoisation when only part of the subproblem space is reachable, since it evaluates lazily.

Why is tabulation faster?

It is iterative, so there is no function-call overhead and no call-stack management.

Why is memoisation called top-down?

It starts from the full problem and breaks it down, consulting the cache on the way; tabulation starts from the base cases and builds up.

Is memoisation the same as caching?

It is a specific form of caching, applied to function return values keyed on arguments.

Chapter summary

- **Dynamic programming** = recursion + memory. Solve each overlapping subproblem once, store it, reuse it.
- Two required conditions: **optimal substructure** (optimum built from sub-optima) and **overlapping subproblems** (the same subproblem recurs).
- Divide and conquer needs *independent* subproblems; DP needs *overlapping* ones. Merge sort and quick sort are **not** DP.
- **Memoisation** is top-down and recursive; **tabulation** is bottom-up and iterative. The difference is recursion vs. iteration, not array vs. map.
- Cache dimension = number of parameters that change: 1D (Fibonacci), 2D (LCS, knapsack), 3D (three-string LCS).
- **Fibonacci**: $\Theta(\varphi^n) \rightarrow \Theta(n) \rightarrow O(1)$ space.

- **LCS:** $\Theta(mn)$. **0/1 knapsack:** $\Theta(nW)$ (pseudo-polynomial). **Coin change:** $\Theta(W|C|)$.
- DP guarantees optimality where greedy does not — e.g. 0/1 knapsack and coin change with awkward denominations.

Greedy Algorithms

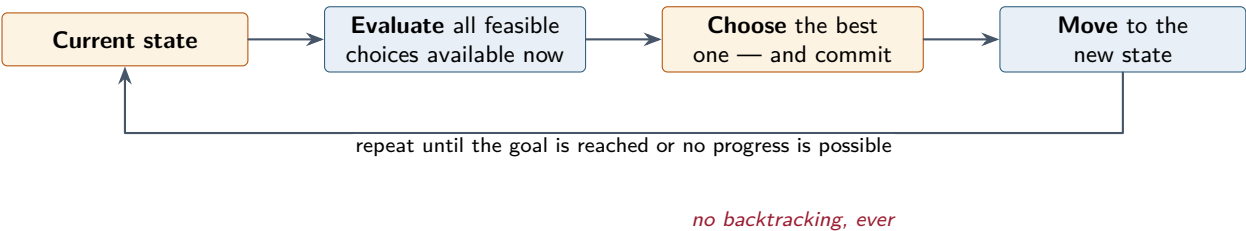
[Sources: Handout 7 (complete); Lab Sheet 10]

9.1 The idea

Greedy algorithm

An algorithm that builds a solution step by step, at each stage choosing the option that looks best *right now* — the locally optimal choice — without reconsidering earlier decisions and without looking ahead.

The Oxford English Dictionary defines *greedy* as “having an excessive desire for something, often without considering the consequences.” The computing sense keeps that meaning exactly: take the best immediate option and never look back.



9.1.1 Historical note

The principle predates the name. The coin-change problem — make change using the fewest coins by repeatedly taking the largest coin that fits — has been used for centuries and works for some currency systems but not all. The term *greedy algorithm* entered computer science during the development of algorithm-design theory in the 1960s and 1970s, by which time the key results already existed:

Year	Author	Algorithm
1930	Vojtěch Jarník	Minimum spanning tree (later rediscovered as Prim’s)
1956	Joseph Kruskal	Minimum spanning tree by edge sorting
1957	Robert C. Prim	Minimum spanning tree by tree growth
1956–59	Edsger W. Dijkstra	Single-source shortest paths (and independently rediscovered Prim’s algorithm in 1959)
1952	David Huffman	Optimal prefix-free codes for data compression

9.2 When greedy works: the two properties

Greedy-choice property

A globally optimal solution can be reached by making a locally optimal (greedy) choice at each step, without reconsidering previous choices and without exploring all possible options.

Optimal substructure

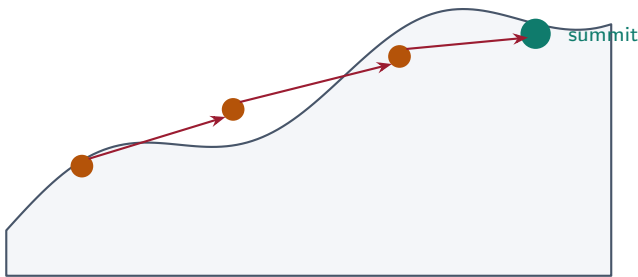
An optimal solution to the problem contains within it optimal solutions to its subproblems. (Same definition as in Chapter 8.)

Key idea

Greedy-choice property + Optimal substructure $\implies$ Guaranteed global optimum

If *either* property fails, the greedy algorithm may still produce a good answer — but it is no longer guaranteed to produce the best one.

9.2.1 The climbing analogy



Greedy choice: the biggest step available now always keeps me on a path to the top — I never need to come back down.

Optimal substructure: once I am partway up, the best route onward does not depend on how I got here.

9.3 Characteristics of greedy algorithms

Characteristic	Detail
Optimisation-oriented	Aims at a maximum or minimum (profit, cost, count) — but guarantees the optimum only when the greedy-choice property holds
Fast and efficient	Typically $O(n)$ or $O(n \lg n)$, since a single pass suffices after sorting
No backtracking	Once a choice is made it is never revisited or undone
Simple to implement	No exploration of alternatives; no table to maintain
Low memory	No subproblem results are stored

9.4 The coin change problem

Coin change

Given denominations C (unlimited supply of each) and an amount W , use the fewest coins totalling exactly W . The greedy strategy: repeatedly take the largest denomination not exceeding the remaining amount.

9.4.1 When greedy succeeds

Worked Example 9.1 | $C = \{1, 2, 5\}$, $W = 13$

Step	Remaining	Coin chosen	Reason
1	13	5	largest ≤ 13
2	8	5	largest ≤ 8
3	3	2	largest ≤ 3
4	1	1	largest ≤ 1
5	0	—	done

Solution $\{5, 5, 2, 1\}$ — **4 coins**, and 4 is provably the minimum.

Worked Example 9.2 | $C = \{1, 2, 5, 10, 20, 50, 100\}$, $W = 36$

Greedy: 20 (leaving 16), 10 (leaving 6), 5 (leaving 1), 1 (leaving 0). Solution $\{20, 10, 5, 1\}$ — **4 coins**, which is optimal.
This denomination set is the one used by most real currencies, and greedy is optimal for it. That is why the strategy feels natural: every cashier in the world uses it.

9.4.2 When greedy fails

Worked Example 9.3 | $C = \{1, 4, 5\}$, $W = 8$

Greedy	Optimal
Take 5 (remaining 3)	
Take 1 (remaining 2)	Take 4 (remaining 4)
Take 1 (remaining 1)	Take 4 (remaining 0)
Take 1 (remaining 0)	
$\{5, 1, 1, 1\}$ — 4 coins	$\{4, 4\}$ — 2 coins

The greedy choice of 5 was locally best but globally wrong: it left a remainder that could only be made up with 1s. The greedy-choice property does not hold for this denomination set.

Worked Example 9.4 | $C = \{1, 3, 4\}$, $W = 6$ Greedy: $4 + 1 + 1$ — **3 coins**. Optimal: $3 + 3$ — **2 coins**.

Dynamic programming (Ch. 8 §8.9)	Greedy
Always produces the optimal solution, if one exists	Produces the optimal solution only for denomination sets with the greedy-choice property
$\Theta(W \cdot C)$ time, $\Theta(W)$ space	$\Theta(C \lg C + W/c_{\min})$ time, $O(1)$ space
Considers all combinations implicitly	Commits to one path

Key idea

The coin-change problem is the best single illustration in the course of why greedy needs justification. The algorithm is identical in all four examples above; only the *denominations* differ, and that alone decides whether the answer is optimal.

9.5 The activity selection problem

Activity selection

Given n activities, each with a start time s_i and finish time f_i , select the maximum number of activities that can be performed by a single resource, such that no two selected activities overlap.

Key idea

Greedy choice: always select the activity that *finishes earliest* among those still compatible. Finishing early leaves the most time available for everything that follows.

Algorithm 9.1 — Activity selection

```

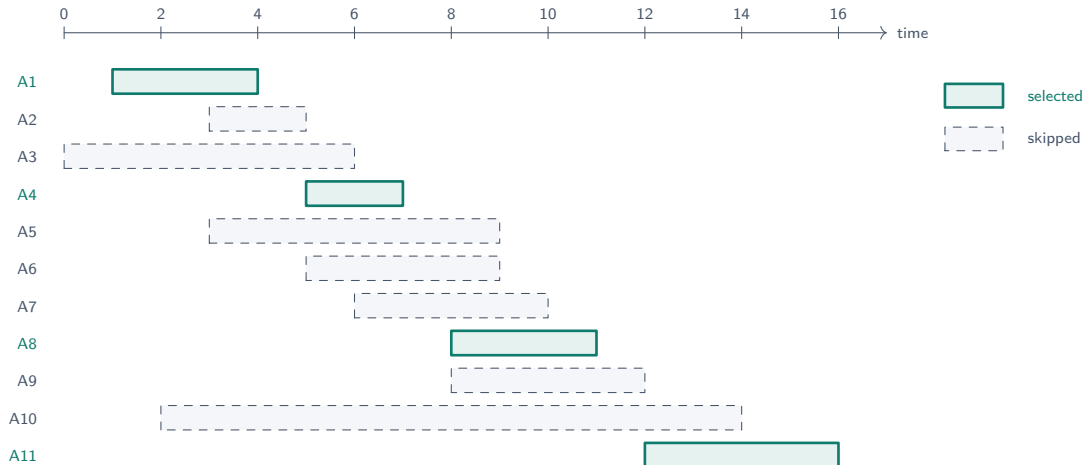
1  function activitySelection(activities[1..n])
2      sort activities by finish time, ascending           //  $O(n \lg n)$ 
3      selected  $\leftarrow$  { activities[1] }                // the earliest finisher
4      lastFinish  $\leftarrow$  activities[1].finish
5      for i  $\leftarrow$  2 to n do
6          if activities[i].start  $\geq$  lastFinish then      // compatible?
7              add activities[i] to selected
8              lastFinish  $\leftarrow$  activities[i].finish
9          end if                                         // else skip it
10     end for
11     return selected
12 end function

```

Worked Example 9.5 | Eleven activities

Activity	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}
Start s_i	1	3	0	5	3	5	6	8	8	2	12
Finish f_i	4	5	6	7	9	9	10	11	12	14	16

(Already sorted by finish time.)



Step	Activity	Start	Last finish	Decision
1	A_1	1	—	select (earliest finisher); lastFinish $\leftarrow 4$
2	A_2	3	4	$3 < 4$: skip
3	A_3	0	4	$0 < 4$: skip
4	A_4	5	4	$5 \geq 4$: select ; lastFinish $\leftarrow 7$
5	A_5	3	7	skip
6	A_6	5	7	skip
7	A_7	6	7	skip
8	A_8	8	7	$8 \geq 7$: select ; lastFinish $\leftarrow 11$
9	A_9	8	11	skip
10	A_{10}	2	11	skip
11	A_{11}	12	11	$12 \geq 11$: select ; lastFinish $\leftarrow 16$

Result: $\{A_1, A_4, A_8, A_{11}\}$ — **4 activities**, which is optimal.Complexity: $\Theta(n \lg n)$, dominated by the sort; the scan itself is $\Theta(n)$.**Why not “shortest activity first”?**

It seems plausible and it is wrong. Consider three activities: $(0, 10)$, $(9, 11)$, $(10, 20)$. Shortest-first picks $(9, 11)$, which blocks both of the others — one activity selected. Earliest-finish picks $(0, 10)$ then $(10, 20)$ — two activities. Nor does “earliest start first” work: a single very long activity starting at time 0 blocks everything. *Only earliest finish time is provably optimal.*

9.6 The knapsack problems

	Fractional knapsack	0/1 knapsack
Rule	Items may be divided; take any fraction	Items are indivisible; take an item entirely or not at all
Copies	One copy of each item	Exactly one copy of each item
Solved optimally by	Greedy (by value density)	Dynamic programming
Greedy result	Always optimal	Not always optimal
Complexity	$\Theta(n \lg n)$	$\Theta(nW)$

9.6.1 Fractional knapsack — greedy is optimal

Key idea

Greedy choice: sort items by *value per unit weight* v_i/w_i , descending. Take whole items in that order until an item does not fit, then take the fraction of it that fills the remaining capacity exactly.

Algorithm 9.2 — Fractional knapsack

```
1 function fractionalKnapsack(items[1..n], capacity W)
2   sort items by value/weight, descending           //  $O(n \lg n)$ 
3   total  $\leftarrow$  0; remaining  $\leftarrow$  W
4   for i  $\leftarrow$  1 to n do
5     if items[i].weight  $\leq$  remaining then
6       total  $\leftarrow$  total + items[i].value         // take the whole item
7       remaining  $\leftarrow$  remaining - items[i].weight
8     else
9       total  $\leftarrow$  total + items[i].value * (remaining / items[i].weight)
10      break                                         // knapsack is now full
11    end if
12  end for
13  return total
14 end function
```

Worked Example 9.6 | Fractional knapsack, $W = 50$

Item	Weight	Value	Value/weight
1	10	60	6.0
2	20	100	5.0
3	30	120	4.0

Already sorted by density.

Step	Action	Remaining capacity	Value so far
1	Take all of item 1 (weight 10)	40	60
2	Take all of item 2 (weight 20)	20	160
3	Take $20/30 = 2/3$ of item 3	0	$160 + \frac{2}{3}(120) = \mathbf{240}$

Maximum value = **240**. Greedy is provably optimal here: because items are divisible, every unit of capacity is filled with the highest-value material still available, so no exchange could improve the total.

9.6.2 0/1 knapsack — greedy fails

Worked Example 9.7 | The same greedy rule on an indivisible instance

Items: $(w, v) = (1, 1), (3, 4), (4, 5), (5, 7)$; capacity $W = 7$.
Densities: 1.00, 1.33, 1.25, 1.40. Greedy order: item 4, item 2, item 3, item 1.

- Take item 4 (weight 5, value 7). Remaining capacity 2.
- Item 2 (weight 3) does not fit. Item 3 (weight 4) does not fit.
- Take item 1 (weight 1, value 1). Remaining capacity 1.

Greedy total: 8. The DP solution (Chapter 8 §8.8) is items 2 and 3, weight $3 + 4 = 7$, **value 9**.
Greedy fails because indivisibility means the “leftover” capacity is wasted — and greedy has no mechanism for reconsidering the item that caused the waste.

9.7 Spanning trees

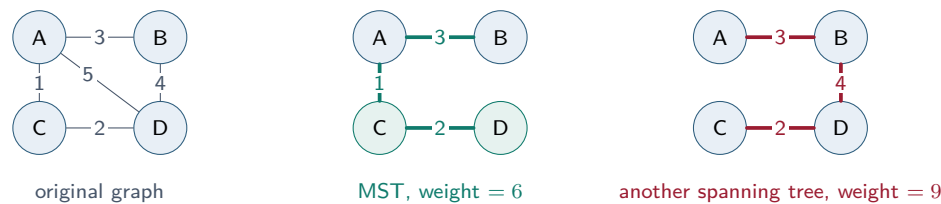
Spanning tree

A spanning tree of a connected, undirected graph $G = (V, E)$ is a subgraph that:

1. is **connected** — every vertex of G is reachable;
2. is **acyclic** — it contains no cycles, so it is a tree;
3. contains **all** $|V|$ vertices of G ;
4. has exactly $|V| - 1$ edges — the minimum number needed to stay connected.

Minimum spanning tree (MST)

A spanning tree whose total edge weight is minimum among all spanning trees of G . A graph may have several distinct MSTs, all of the same total weight.



9.7.1 Applications of MSTs

- Computer network design (minimum cabling cost)
- Telephone and fibre-optic network layout
- Electrical power distribution networks
- Water and gas pipeline networks
- Road and highway network planning
- Railway network design
- Spanning Tree Protocol (loop-free Ethernet)
- Network routing infrastructure
- Hierarchical clustering in machine learning
- Image segmentation in computer vision
- Approximation algorithms for TSP
- Circuit and PCB design
- Sensor network design
- Distribution and logistics planning

9.8 Kruskal's algorithm

Key idea

Kruskal thinks in edges. Sort all edges by weight, then repeatedly add the smallest edge that does not create a cycle. Also described as the “avoid cycles” method.

Algorithm 9.3 — Kruskal's MST

```

1  function kruskalMST(G = (V, E))
2      sort E in non-decreasing order of weight      // O(E lg E) -- dominant step
3      T ← empty set
4      make-set(v) for every v in V                  // each vertex its own
           component
5      for each edge (u,v) in sorted order do
6          if find(u) ≠ find(v) then                  // different components => no
           cycle
7              add (u,v) to T
8              union(u, v)                            // merge the two components
9          end if                                     // else DISCARD -- would form a
           cycle
10         if |T| = |V| - 1 then break                // spanning tree complete
11     end for
12     return T
13 end function

```

Union-Find (disjoint-set)

Kruskal needs to know whether two vertices are already connected. The **union-find** structure supports two operations:

find(x) Return a representative of x 's component. Two vertices are in the same component iff their representatives are equal.

union(x, y)
Merge the two components containing x and y .

With path compression and union by rank, both run in effectively constant amortised time.

9.8.1 Complexity

Step	Cost	Note
Sorting all edges	$O(E \lg E)$	Dominant step α is the inverse Ackermann function — effectively constant
Union–find operations	$O(E \alpha(V)) \approx O(E)$	
Total time	$O(E \lg E) = O(E \lg V)$	since $E < V^2$, $\lg E < 2 \lg V$
Space	$O(V)$	Parent array $O(V)$ + rank array $O(V)$

9.9 Prim’s algorithm

Key idea

Prim thinks in vertices. Start from any vertex and grow a single tree, repeatedly adding the cheapest edge that connects the tree to a vertex not yet in it. Also described as the “build tree” method.

Cut

A **cut** in a graph is a partition of the vertices into two sets; the edges *crossing* the cut are those with one endpoint in each set. At every step Prim’s algorithm forms the cut (vertices in the tree | vertices not in the tree) and picks the minimum-weight edge crossing it.

Algorithm 9.4 — Prim’s MST

```

1  function primMST(G = (V,E), start s)
2    for each v in V do
3      key[v] ← infinity           // cheapest known edge weight connecting v to
                                   the tree
4      parent[v] ← nil
5    end for
6    key[s] ← 0
7    mstSet ← empty
8    while |mstSet| < |V| do
9      u ← vertex not in mstSet with the smallest key           // priority queue
10     add u to mstSet
11     for each neighbour v of u do
12       if v not in mstSet and weight(u,v) < key[v] then
13         key[v] ← weight(u,v)
14         parent[v] ← u
15       end if
16     end for
17   end while
18   return the edges (v, parent[v]) for all v ≠ s
19 end function

```

Key idea

key[v] is the *minimum edge weight needed to connect vertex v to the growing MST*. It is not a distance from the start vertex — that is Dijkstra’s algorithm, which differs from Prim’s by exactly this one detail.

9.9.1 Complexity

Implementation	Time	Best for
Adjacency matrix, linear scan for the minimum	$O(V^2)$	Dense graphs ($E \approx V^2$)
Adjacency list with binary min-heap	$O(E \lg V)$	Sparse graphs
Adjacency list with Fibonacci heap	$O(E + V \lg V)$	Theoretical optimum
Space	$O(V)$	key, parent and mstSet arrays

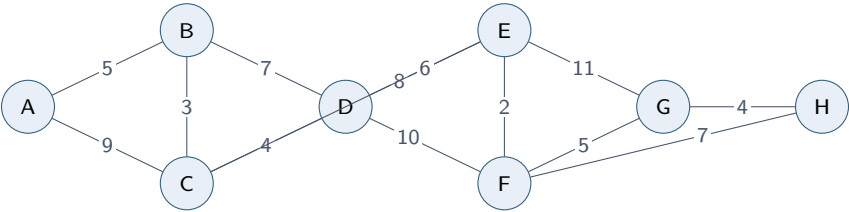
9.10 Kruskal versus Prim

	Kruskal	Prim
Thinks in terms of	Edges — sort all, add the smallest safe one	Vertices — grow one tree from a start vertex
Greedy choice	Globally smallest remaining edge that creates no cycle	Smallest edge crossing the current cut
Intermediate state	A <i>forest</i> of several components that eventually merge	A single connected tree that grows
Data structure	Union–find (disjoint sets)	Priority queue (min-heap)
Needs a start vertex	No	Yes (any vertex)
Time	$O(E \lg E)$	$O(V^2)$ or $O(E \lg V)$
Better for	Sparse graphs (few edges to sort)	Dense graphs
Disconnected graphs	Produces a minimum spanning <i>forest</i> naturally	Produces only the component containing the start vertex

Both are greedy, and both rely on the same underlying justification — the **cut property**: for any cut of the graph, the minimum-weight edge crossing that cut belongs to some MST. Kruskal applies it implicitly by never closing a cycle; Prim applies it explicitly at every step.

9.10.1 A complete worked comparison

Worked Example 9.8 | Kruskal and Prim on the same graph



$|V| = 8, |E| = 13$, so the MST will have $|V| - 1 = 7$ edges.
Kruskal — edges sorted by weight, in the order considered:

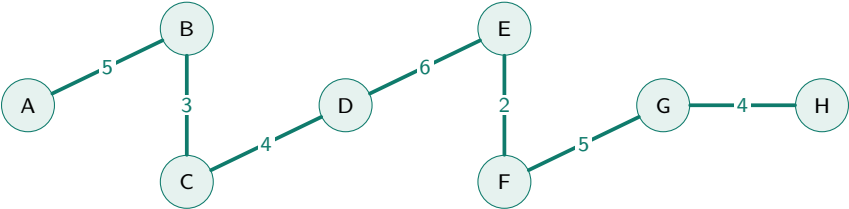
Order	Edge	Weight	Decision
1	E-F	2	ACCEPT
2	B-C	3	ACCEPT
3	C-D	4	ACCEPT
4	G-H	4	ACCEPT
5	A-B	5	ACCEPT
6	F-G	5	ACCEPT
7	D-E	6	ACCEPT (7th edge — tree complete)
8	B-D	7	REJECT — B and D already connected (cycle)
9	F-H	7	REJECT — cycle
10	C-E	8	REJECT — cycle
11	A-C	9	REJECT — cycle
12	D-F	10	REJECT — cycle
13	E-G	11	REJECT — cycle

MST weight = $2 + 3 + 4 + 4 + 5 + 5 + 6 = \boxed{29}$

Prim — starting at A, in the order the edges were selected:

Step	Vertex added	Edge used	Weight	Tree so far
1	B	A-B	5	{A, B}
2	C	B-C	3	{A, B, C}
3	D	C-D	4	{A, B, C, D}
4	E	D-E	6	{A, B, C, D, E}
5	F	E-F	2	{A, B, C, D, E, F}
6	G	F-G	5	{A, B, C, D, E, F, G}
7	H	G-H	4	all 8 vertices

MST weight = $5 + 3 + 4 + 6 + 2 + 5 + 4 = \boxed{29}$



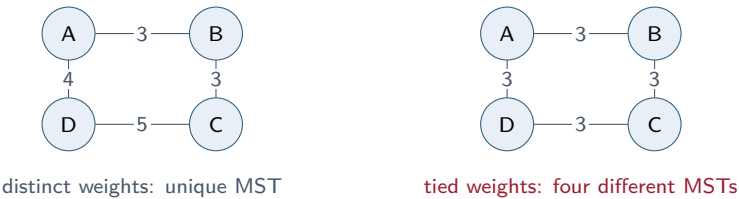
Minimum spanning tree, total weight 29

Observation. The two algorithms selected the edges in *different orders* — Kruskal took E-F first, Prim took A-B first — but arrived at the same tree of weight 29. That is typical, and it is a good self-check: if your two answers have different totals, one of them is wrong.

Presenting MST answers

- Lab Sheet 10 is explicit about what must be shown:
- *For Kruskal*: “a complete list of the edges, indicating which edges you take and which (if any) you reject”. So list **all** edges in sorted order and annotate each ACCEPT/REJECT — do not just list the seven you kept.
 - *For Prim*: “list the edges selected by the algorithm in the order they were selected”, and where the question says “starting at A”, state the order in which *vertices* were considered as well as the edges included.
 - Always state the final total weight.

9.10.2 Why a graph can have several MSTs



Algorithm	Why multiple MSTs can arise
Prim's	The starting vertex can be chosen arbitrarily, <i>and</i> when several edges crossing the cut have equal weight, different choices give different trees. The starting vertex alone does not create multiple MSTs — ties do.
Kruskal's	There is no starting vertex, but when several edges have equal weight the sorted order among them is arbitrary, and different orders can produce different trees.

Key idea

Ties in edge weight are the *only* source of multiple MSTs. If all edge weights are distinct, the MST is unique — and both Kruskal and Prim will find that same unique tree regardless of implementation details.

9.11 Greedy versus dynamic programming

Property	Dynamic programming	Greedy
Core principle	Principle of optimality: build the optimum from optimal sub-solutions	Best local (feasible) choice at each step
Decision making	Considers all choices, evaluates subproblems, then picks the best combination	Makes one choice per step, commits immediately, never reconsiders
Subproblems explored	Solves overlapping subproblems (once each, with memoisation)	Follows one path only; no alternatives explored
Backtracking	Not needed, but alternatives are implicitly considered via the recurrence	Never; decisions are final
Optimality guarantee	Always optimal if the principle of optimality holds	Optimal only if greedy-choice property <i>and</i> optimal substructure both hold
Time complexity	Usually higher — often $O(n^2)$ or $O(nW)$	Usually lower — $O(n \lg n)$ or $O(n)$
Space complexity	Higher — a table of subproblem results	Lower — nothing stored
Approach style	Bottom-up (tabulation) or top-down (memoisation)	Top-down only, single forward pass
When to use	When local choices cannot be trusted to lead to the global optimum	When the problem structure guarantees local choice = global optimum

Key idea

The philosophical difference in one line each:

- **Dynamic programming asks:** “What is the best way to solve this, considering all the ways the smaller pieces could combine?” — exhaustive but smart, since memoisation removes redundancy.
- **Greedy asks:** “What is the best thing to do right now?” — fast but risky, since it assumes today’s best choice will not hurt tomorrow.

9.12 Heuristics

Heuristic

A practical problem-solving approach that uses experience or rules of thumb to make good decisions quickly when time, information or computational resources are limited. Heuristics do not guarantee an optimal solution.

Situation	Heuristic	Result
Finding a friend in a crowd	Search where they are most likely to be (entrance, food court) rather than randomly	Usually faster, but not guaranteed
Playing chess	Capture the opponent's most valuable piece; protect your king	Often a good move, rarely the best move in the whole game
Routing a delivery	Always go to the nearest unvisited stop	A reasonable tour quickly; typically 25% worse than optimal

Greedy heuristics often produce good approximate solutions in reasonable time, which makes them valuable for large or NP-hard problems where the exact optimum is unreachable. Their quality is measured by the **approximation ratio** — the ratio of the heuristic's answer to the true optimum.

9.13 Applications of greedy algorithms

Application	Greedy rule
Memory allocation (first-fit, best-fit)	Place a block in the first (or smallest adequate) free region
CPU scheduling	Choose the next process by a local priority rule to maximise utilisation
Bin packing	Place each item into a bin using a local rule, minimising bins used. Applied to disk placement and container loading
Travelling salesman (heuristic)	Always travel to the nearest unvisited city
Fractional knapsack	Take items in decreasing value-per-weight order
Minimum spanning trees	Kruskal's and Prim's algorithms
Dijkstra's shortest path	Repeatedly finalise the unvisited vertex with the smallest tentative distance
Egyptian fractions	Repeatedly subtract the largest unit fraction that fits, e.g. $\frac{47}{154} = \frac{1}{7} + \frac{1}{11} + \frac{1}{14}$
Coin change	Take the largest coin not exceeding the remainder (optimal for some denomination sets only)
Huffman coding	Repeatedly merge the two least frequent symbols to build an optimal prefix-free code

9.13.1 Huffman coding in brief

Huffman coding

A lossless data-compression technique that assigns short binary codes to frequent symbols and long codes to rare ones. Build a binary tree by repeatedly removing the two lowest-frequency nodes and joining them under a new parent whose frequency is their sum; the path from the root to each leaf is that symbol's code.

The greedy choice — always merge the two *least* frequent nodes — is provably optimal: it guarantees that the rarest symbols end up deepest in the tree, hence with the longest codes, minimising the expected code length.

9.14 Advantages and disadvantages

Advantages	Disadvantages
Simplicity — easier to describe, understand and implement than DP or backtracking	Not always optimal — prioritising local optima can give a suboptimal global result (e.g. 0/1 knapsack)
Efficiency — one pass through the data; less time and memory	Hard to prove optimal — requires establishing both the greedy-choice property and optimal substructure
Optimal for many problems — Kruskal, Prim, Huffman, activity selection	Sensitive to input order — the processing order can change the result
Good approximations — even without a guarantee, the answer is usually decent and fast	Limited applicability — fails on problems with complex constraints or interdependent choices
Building block — used inside branch and bound, or as a warm start for optimisation	

9.14.1 Prim's algorithm: specific advantages and disadvantages

Advantages	Disadvantages
Guarantees the MST for a connected weighted graph	Slower on very large or dense graphs due to the number of edges processed
$O(E \lg V)$ with a binary heap	Requires a priority queue, using extra memory
Simple to understand and implement	The choice of starting vertex can yield a different MST when ties exist (though the total weight is unchanged)
Performs well on dense graphs with an adjacency matrix	Works only on <i>connected</i> graphs; on a disconnected graph it finds only the component containing the start vertex, producing a minimum spanning <i>forest</i> rather than a tree

Chapter summary

- A **greedy algorithm** makes the locally best feasible choice at each step and never reconsiders.
- It is guaranteed optimal only when both the **greedy-choice property** and **optimal substructure** hold.
- **Coin change**: greedy is optimal for some denomination sets ($\{1, 2, 5\}$, real currencies) and not others ($\{1, 4, 5\}$, $\{1, 3, 4\}$). DP is always optimal.
- **Activity selection**: choose earliest *finish* time. $\Theta(n \lg n)$. Provably optimal; shortest-first and earliest-start-first are not.
- **Fractional knapsack**: greedy by value/weight is optimal. **0/1 knapsack**: greedy fails; use DP.
- **Spanning tree**: connected, acyclic, all $|V|$ vertices, exactly $|V| - 1$ edges. **MST**: minimum total weight.

- **Kruskal** = edges + union-find, $O(E \lg E)$, better for sparse graphs. **Prim** = vertices + priority queue, $O(V^2)$ or $O(E \lg V)$, better for dense graphs. Both rely on the cut property.
- Multiple MSTs arise **only** from ties in edge weight.
- Greedy vs. DP: greedy is faster and simpler but conditional; DP is slower and memory-hungry but always optimal when optimal substructure holds.

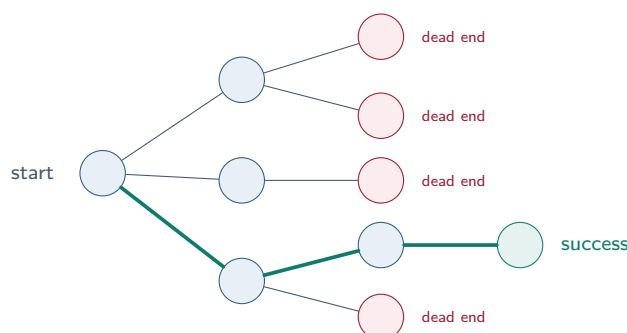
Backtracking

[Sources: Handout 8 (complete)]

10.1 The idea

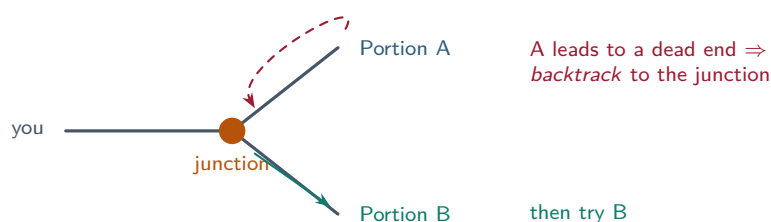
Backtracking

A technique for searching a large solution space by building a candidate solution *incrementally*, one component at a time, and abandoning (“backtracking” from) a partial candidate as soon as it becomes clear that it cannot be completed into a valid solution.



10.1.1 The maze analogy

You are in a maze. At a junction you have two options, leading into portion A and portion B.



If a junction has more than two choices, the rule generalises: try each in turn, and if you get stuck, return to the junction and try the next. If every choice from every junction leads to a dead end, there is no solution.

Key idea

Backtracking = brute force + early abandonment. It explores the same conceptual search space as exhaustive search, but discards entire branches the moment a partial candidate violates a constraint — which in practice removes the overwhelming majority of the space.

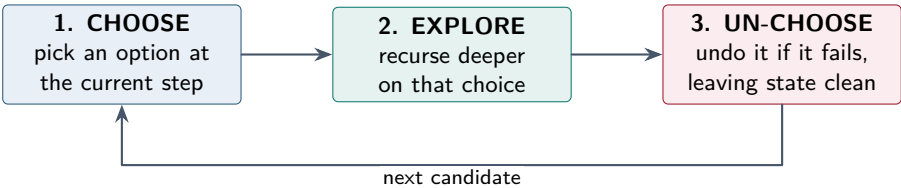
10.2 Backtracking versus plain recursion

	Plain recursion	Backtracking
Goal	Compute a value or traverse a structure	Search for solution(s) satisfying constraints
Explores multiple branches?	Not necessarily	Yes, systematically
Undoes state?	No	Yes — that is the defining move
Examples	Factorial, Fibonacci, tree height, quick sort	N-Queens, Sudoku, permutations and subsets, maze solving, graph colouring

Key idea

Recursion is a *mechanism* — a function calling itself. Backtracking is a *strategy* that happens to be implemented with recursion. Every backtracking algorithm is recursive; most recursive algorithms are not backtracking.

10.2.1 The three-move cycle



The “un-choose” step is what distinguishes backtracking. After exploring a path and returning, all changes made by the choice must be reverted so that the next candidate starts from a clean state.

10.3 The state space tree

State space tree

A tree in which the root is the initial state, each edge is a choice, and each node is a partial (or complete) candidate solution. Leaves are either complete solutions or dead ends.

Backtracking traverses the state space tree by **depth-first search** (DFS) — it goes as deep as possible along one branch before returning.

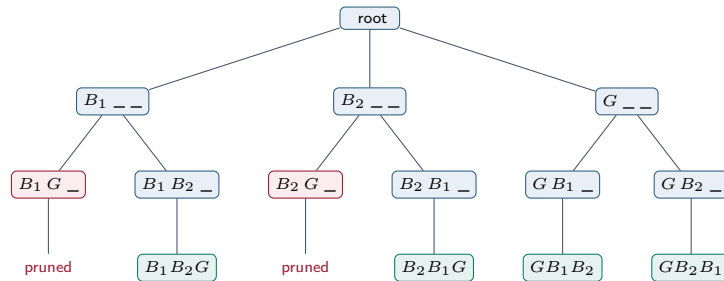
	Depth-first search (DFS)	Breadth-first search (BFS)
Order	Explores one branch fully before the next	Explores all nodes at depth d before depth $d + 1$
Structure	Stack (or the call stack)	Queue
Memory	$O(\text{depth})$	$O(\text{width})$ — can be far larger
Used by	Backtracking	Branch and bound (FIFO variant)

Worked Example 10.1 | State space tree: two boys and one girl on three benches

Problem. Arrange 2 boys (B_1, B_2) and 1 girl (G) on three benches.

Constraint. The girl must not be on the middle bench.

Total arrangements without the constraint: $3! = 6$.



Valid solutions: 4 — $B_1B_2G, B_2B_1G, GB_1B_2, GB_2B_1$. The two branches placing G in the middle are pruned as soon as the second position is chosen, without ever constructing the third position. That is the saving.

Backtracking is not for optimisation

Handout 8 is emphatic: dynamic programming is for optimisation problems (a single best answer); backtracking is for problems where *multiple* solutions exist and all — or any — of them are wanted. Backtracking *can* find the best solution by exploring everything and keeping the best, but it is not designed for that. For optimisation, use DP, greedy, or branch and bound.

10.4 The general algorithm

Algorithm 10.1 — Generic backtracking template

```

1 procedure BACKTRACK(current_state)
2   if is_solution(current_state) then
3     record current_state           // and optionally stop here
4     return
5   end if
6
7   for each candidate in generate_candidates(current_state) do
8     new_state ← apply_choice(current_state, candidate) // 1. CHOOSE
9     if is_valid(new_state) then // prune here
10      BACKTRACK(new_state) // 2. EXPLORE
11    end if
12    undo_choice(current_state, candidate) // 3. UN-CHOOSE
13  end for
14 end procedure

```

Component	Purpose
<code>is_solution(state)</code>	Tests whether the state is a complete, valid solution
<code>generate_candidates(state)</code>	Produces the choices available for extending the partial solution
<code>apply_choice(state, c)</code>	Builds the extended state
<code>is_valid(state)</code>	Tests whether the partial solution still satisfies all constraints. Returning false prunes the entire subtree
<code>undo_choice(state, c)</code>	The backtracking step: reverts the change so other candidates can be tried from the same state

10.5 The N-Queens problem

N-Queens

Place N queens on an $N \times N$ chessboard so that no two attack each other — no two queens share a row, a column, or a diagonal.

10.5.1 The backtracking strategy

- Place queens one row at a time, starting from the top row.
- When placing a queen in a row, check for conflicts with all queens already placed.
- If a column in the current row is safe, place the queen there and move to the next row.
- If no column in the row is safe, **backtrack**: undo the previous row’s placement and try the next column there.

Key idea

The row-by-row structure builds in a constraint for free: by placing exactly one queen per row, the row constraint can never be violated. Only columns and diagonals need to be checked. This is a small but characteristic backtracking design decision — encode constraints into the *structure* of the search so they need not be tested.

10.5.2 4-Queens, fully traced

Worked Example 10.2 | The 4-Queens problem

Q	x	x	x
x	x		
x		x	
x			x

Step 1: Q_1 at (0,0)

Q	x	x	x
x	x	Q	x
x	x	x	x
x	x		x

Step 2: Q_2 at (1,2)

Q	x	x	x
x	x	x	Q
x		x	
x			x

Step 4: retry Q_2 at (1,3)

Q			
			Q
	Q		
x	x	x	x

Step 5: Q_3 at (2,1)

x	Q	x	x
x	x	x	
		x	x
	x		

Step 7: Q_1 at (0,1)

x	Q	x	x
x	x	x	Q
		x	x

Step 8: Q_2 at (1,3)

	Q		
			Q
Q			
	x	x	x

Step 9: Q_3 at (2,0)

	Q		
			Q
Q			
		Q	

Step 10: Q_4 at (3,2) **SOLVED**

Narrative of the search.

Step	Action
0	Initialise an empty 4×4 board.
1	Place Q_1 at $(0, 0)$. Cells attacked by it are marked $\times$. Move to row 1.
2	Row 1: columns 0 and 1 are attacked; place Q_2 at $(1, 2)$. Move to row 2.
3	Row 2: <i>every</i> column is attacked. Backtrack — remove Q_2 from $(1, 2)$.
4	Row 1 still has a safe cell: place Q_2 at $(1, 3)$. Move to row 2.
5	Row 2: place Q_3 at $(2, 1)$. Move to row 3.
6	Row 3: no safe cell. Backtrack — remove Q_3 . Row 2 has no other safe cell, so backtrack again — remove Q_2 . Row 1 has no further safe cell, so backtrack again — remove Q_1 from $(0, 0)$ and move it to the next column.
7	Place Q_1 at $(0, 1)$. Move to row 1.
8	Place Q_2 at $(1, 3)$. Move to row 2.
9	Place Q_3 at $(2, 0)$. Move to row 3.
10	Place Q_4 at $(3, 2)$. All four queens placed — solution found .

Output format. Queens are shown as Q, empty cells as . :

```
.Q..
...Q
Q...
..Q.
```

The 4-Queens problem has exactly **two** solutions; the other is the mirror image of this one.

Complexity. Time $O(N!)$ in the worst case — there are N choices in row 1, at most $N - 1$ compatible ones in row 2, and so on. Auxiliary space $O(N)$ for the board representation (one column index per row) plus the recursion stack.

Algorithm 10.2 — N-Queens

```

1 function solveNQueens(board, col_of_row[], row)
2   if row = N then
3     return true                                // all queens placed
4   end if
5   for col ← 0 to N-1 do
6     if isSafe(col_of_row, row, col) then
7       col_of_row[row] ← col                    // CHOOSE
8       if solveNQueens(board, col_of_row, row+1) then // EXPLORE
9         return true
10      end if
11      col_of_row[row] ← nil                     // UN-CHOOSE (backtrack)
12    end if
13  end for
14  return false                                // no safe column in this row
15 end function
16
17 function isSafe(col_of_row, row, col)
18   for r ← 0 to row-1 do
19     c ← col_of_row[r]
20     if c = col then return false                // same column
21     if abs(r - row) = abs(c - col) then return false // same diagonal
22   end for

```

```
23     return true
24 end function
```

10.6 Sudoku

Sudoku

Fill a 9×9 grid so that each row, each column, and each 3×3 subgrid contains every digit from 1 to 9 exactly once. (Illustrated below on a 4×4 grid with digits 1–4 and 2×2 subgrids.)

Stage	Action
1. Choice	Pick an empty cell and write a candidate digit in it.
2. Validation	Check whether the grid is still legal — no duplicate in the row, column or subgrid. If legal, move to the next empty cell. If not, try the next digit in <i>this</i> cell.
3. Backtracking	If no digit works in this cell, undo the <i>previous</i> cell’s digit and try the next value there — recursively, as far back as necessary.
4. Solution	Repeat until every cell is filled legally.

2	1	3	
			2
3			
		4	

1 conflicts – try 2 instead

2	4	3	1
1	3	4	2
3	1	2	4
4	2	1	3

complete and legal

10.7 Hamiltonian paths and cycles

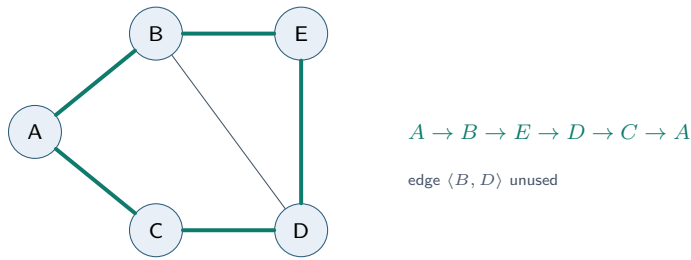
Hamiltonian path and cycle

A **Hamiltonian path** visits every vertex of a graph exactly once. A **Hamiltonian cycle** is a Hamiltonian path that returns to its starting vertex — a closed loop visiting every vertex exactly once. A graph containing one is called a *Hamiltonian graph*.

Named after Sir William Rowan Hamilton, who in 1857 invented a puzzle game based on finding such a cycle.

A Hamiltonian cycle need not use all edges

It must visit every *vertex* exactly once; it may omit edges entirely. In the graph below, $A \rightarrow B \rightarrow E \rightarrow D \rightarrow C \rightarrow A$ is a Hamiltonian cycle even though the edge $\langle B, D \rangle$ is never used.



10.7.1 Decision problem and optimisation problem

Form	Statement
Decision problem	“Given a path, is it a Hamiltonian cycle of this graph?” — answerable yes/no
Optimisation problem	“Given graph G , find a Hamiltonian cycle.” — requires a search

10.7.2 Constraints for the backtracking search

The candidate path is built one vertex at a time, subject to:

1. Vertices i and $i + 1$ in the path must be **adjacent** in the graph.
2. The last vertex must be adjacent to the first (to close the cycle).
3. Vertex i must not already appear among the first $i - 1$ vertices of the path.

With an adjacency-matrix representation, checking adjacency is $O(1)$.

Worked Example 10.3 | Hamiltonian cycle by backtracking, traced

Consider a graph on vertices $\{1, 2, 3, 4, 5\}$ and start the tour at vertex 1.

Step	Action
1	Build a path from vertex 1 forward, e.g. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$. Vertex 5 has no edge back to 1 — dead end .
2	Backtrack to the deepest node with an unexplored alternative — here vertex 3 — and take a different next vertex.
3	The new path also reaches a dead end. Backtrack again and exhaust all options at that level.
4	Continue backtracking upward until a node with a genuinely new option is found — here vertex 2.
5	One of the resulting paths closes back to vertex 1: a Hamiltonian cycle is recorded. Continue exploring to find any others.
6	The search terminates having found (in this example) two distinct Hamiltonian cycles.

Complexity. In the worst case the state-space tree has

$$1 + n + n^2 + \cdots + n^{n-1} = \frac{n^n - 1}{n - 1}$$

nodes, giving $T(n) = O(n^n)$ — exponential time. Constraint checking prunes most of this in practice, but the worst case is unavoidable: deciding whether a Hamiltonian cycle exists is NP-complete (Chapter 12).

10.8 Graph colouring

Graph colouring

Assign a colour to every vertex of a graph so that no two adjacent vertices share a colour, using the *minimum* number of colours.

The backtracking algorithm assigns colours one vertex at a time, checking after each assignment that no neighbour already has that colour. On conflict it tries the next colour; when all colours fail for a vertex, it backtracks.

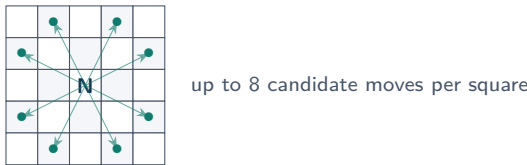
Application	Mapping to the problem
Timetabling and exam scheduling	Vertices are exams; edges join exams sharing a student; colours are time slots
Frequency assignment in wireless networks	Vertices are transmitters; edges join those that would interfere; colours are frequencies
Map colouring	Vertices are regions; edges join regions sharing a border; colours are map colours
Register allocation in compilers	Vertices are variables; edges join variables live at the same time; colours are CPU registers
Chip and PCB design	Vertices are components; edges are conflicts; colours are layers

10.9 The Knight's Tour

Knight's Tour

Move a knight around a chessboard so that it visits every square exactly once.

The backtracking algorithm tries each of the (up to eight) legal knight moves from the current square, marking squares as visited. If the knight becomes stuck without having visited every square, it backtracks and tries a different move sequence.



up to 8 candidate moves per square

10.10 Optimising backtracking

Technique	What it does	Example
Pruning	Identify and eliminate branches early that cannot lead to a solution	Stop exploring a Sudoku configuration the instant a rule is violated
Heuristics	Order candidates so the most promising branches are explored first	In N-Queens, place queens in positions least likely to cause conflicts; in Sudoku, fill the most-constrained cell first
Memoisation	Cache explored states so equivalent configurations are not re-derived	Cache visited states in puzzle solvers where the same position is reachable by different move orders
Constraint propagation	Deduce forced values before searching	In Sudoku, if a cell has only one legal digit, fill it without branching

Key idea

Pruning is the single most important optimisation, and it is what makes backtracking practical at all. The earlier a constraint is checked, the larger the subtree removed. Checking a constraint at depth 2 of a tree with branching factor 8 and depth 10 eliminates roughly $8^8 \approx 1.7 \times 10^7$ nodes at a cost of one test.

10.11 Applications, advantages and limitations

Application area	Examples
Puzzle solving	Sudoku, crosswords, N-Queens
Combinatorial generation	All permutations, combinations and subsets of a set
Constraint satisfaction	Graph colouring, timetabling, job assignment
Pathfinding	Mazes, rat-in-a-maze, Knight's Tour
Game solving	Exploring possible moves in strategy games
Decision making	Subset sum and similar multiple-choice problems
String processing	Generating valid strings, e.g. balanced parentheses
Optimisation	Maximising profit or minimising cost under constraints (though DP or branch and bound are usually preferable)

Advantages	Limitations
Completeness — explores all valid possibilities, so it never misses a solution	Combinatorial explosion — the number of combinations can grow factorially. If one TSP path takes 1 microsecond to evaluate, checking every path through 20 points takes nearly two millennia
Efficiency — strong constraints allow many options to be eliminated quickly	Complexity — defining the choices, constraints and stopping criteria correctly can be difficult
Adaptability — the tree structure is generic, so the same template applies to very different problems	No convergence guarantee — may not terminate usefully when no solution exists, or may loop if the problem is poorly specified

Chapter summary

- **Backtracking** builds candidate solutions incrementally and abandons a partial candidate as soon as it cannot be completed — brute force plus early pruning.
- The defining move is **choose** → **explore** → **un-choose**. The un-choose (state reversal) is what separates backtracking from plain recursion.
- The search space is a **state space tree**, traversed depth-first.
- Backtracking is for problems with *multiple* solutions; it is not an optimisation technique.
- **N-Queens**: one queen per row; check columns and diagonals; $O(N!)$ time, $O(N)$ space. The 4-Queens problem has 2 solutions.
- **Hamiltonian cycle**: visits every vertex exactly once and returns to the start; may omit edges; $O(n^n)$ worst case; the decision version is NP-complete.
- Other classics: Sudoku, graph colouring, Knight's Tour, maze solving, subset generation.
- **Optimise with**: pruning (most important), heuristic ordering, memoisation, constraint propagation.
- The fundamental limit is combinatorial explosion — addressed further by branch and bound (Chapter 11).

Branch and Bound

[Sources: Handout 9 (complete); Handout 8 §“Algorithm Design Techniques”]

11.1 The idea

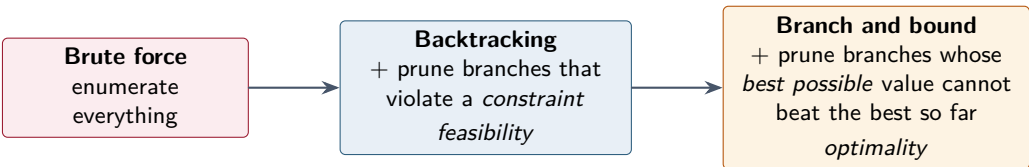
Branch and bound

A systematic method for solving *optimisation* problems — problems about finding the best solution among many possible options. The problem is divided into smaller parts (**branching**), and those parts are evaluated to decide whether to explore them further or discard them (**bounding**). The bound uses logical tests to eliminate suboptimal paths early, speeding up the search.

Key idea

Unlike backtracking, which focuses on **feasibility** — finding *any* solution that satisfies the problem’s constraints — branch and bound aims to find the **optimal** solution by efficiently exploring the entire solution space.

Consider finding the shortest travelling path through several cities. The algorithm explores routes (branching) and uses bounds — such as the minimum distance already travelled — to discard longer routes early in the process.



11.1.1 Which technique for which problem

Handout 9 draws the line explicitly:

Backtracking	8-Queens problem; knapsack using backtracking; Hamiltonian path
Branch and bound	Travelling Salesman Problem

11.2 Backtracking versus branch and bound

Both generate a state space tree. The difference is *how* the tree is generated and *why* nodes are discarded.

	Backtracking	Branch and bound
Goal	Feasibility — find any (or all) solutions satisfying the constraints	Optimality — find the best solution
Tree traversal	Follows DFS	Follows BFS in its classic form (see the note below)
Pruning basis	A constraint is violated	The bound cannot beat the best solution so far
Extra machinery	None beyond the constraints	A bounding function and a record of the best value so far
Problem type	Constraint satisfaction	Discrete and combinatorial optimisation
Typical use	N-Queens, Sudoku, graph colouring, Hamiltonian path	TSP, job assignment, knapsack (exact)

“B&B follows BFS” needs one qualification

Handout 9 states that backtracking follows DFS whereas branch and bound follows BFS, and in BFS all nodes of the same level are added before moving to the next level. That is true of the *classic* (FIFO) formulation, and it is the contrast the exam is testing. But the same handout then lists **three** search techniques for B&B — LC, BFS and DFS — so B&B is not restricted to breadth-first. The honest summary is: *backtracking is always depth-first; branch and bound may be breadth-first (FIFO), depth-first (LIFO), or least-cost, and it is the presence of the bounding function — not the traversal order — that makes it branch and bound.*

11.3 Vocabulary

Live node, E-node, dead node

Live node

A node that has been generated but whose children have not all been generated yet.

E-node The *expanding* node — the live node currently being expanded.

Dead node

A generated node that is not to be expanded further, either because it is a leaf or because it has been killed by the bounding function.

Feasible and optimal solution

A **feasible solution** is defined by the problem states that satisfy all the given constraints. An **optimal solution** is a feasible solution which produces the best value of a given objective function.

11.4 When to apply branch and bound

Condition	Meaning
Discrete optimisation	The variables belong to a discrete set — integers, subsets, permutations, yes/no choices
Combinatorial optimisation	The objective function is maximised or minimised over a finite but enormous set of combinations
NP-hard problems	B&B provides an optimal solution to an NP-hard problem by exploring the entire search space, identifying candidate solutions step by step

Consolidation note

NP-hardness (Chapter 12) classifies problems that are, in a sense, as hard as the hardest problems in NP. Many optimisation problems in computer science — the shortest path in a graph, the minimum spanning tree — have a finite number of feasible solutions and *can* be solved in polynomial time. The problems B&B targets are the ones that cannot: they require, in the worst case, examining all possible permutations.

11.5 Basic concepts

11.5.1 Generating a state space tree

As with backtracking, B&B generates a state space tree to search the solution space of a given problem instance efficiently.

11.5.2 Evaluating a candidate solution

Unlike backtracking, B&B needs two additional ingredients to evaluate a candidate:

1. **A bound assigned to each node.** A way to assign a bound on the best value of the given criterion function to each node in the state space tree. It is produced by adding further components to the partial solution given by that node.
2. **The best value obtained so far.** This describes the *upper* bound for a maximisation problem and the *lower* bound for a minimisation problem.

11.5.3 The bounding function

Bounding function

A heuristic function that evaluates the lower and upper bounds on the possible solutions at each node. The bound values are used to steer the search along the partial solutions leading to an optimal solution.

If a node does **not** produce a solution better than the best solution obtained so far, it is abandoned without further exploration and the algorithm branches to another path. The desired solution is the value of the best solution produced so far.

Which direction the bound must point

Problem	Bound needed	Kill the node when
Minimisation (e.g. TSP)	<i>Lower</i> bound on the cost achievable in the subtree	bound $\geq$ best complete cost found so far
Maximisation (e.g. knapsack)	<i>Upper</i> bound on the value achievable in the subtree	bound $\leq$ best complete value found so far

The bound must always be *optimistic*. If it ever promises less than the subtree can actually deliver, the subtree containing the true optimum may be discarded and the algorithm returns a wrong answer.

11.6 Types of solution representation

The solution of a branch-and-bound problem can be represented in two ways.

Representation	Description
Variable size solution	The subset of the given set that gives the optimised solution. If a combination from $\{A, B, C, D\}$ is required and A and B together give the best solution, the answer is $\{A, B\}$.
Fixed size solution	A string of 0s and 1s, where the digit in position i indicates whether the i th element is included. The same answer becomes $\{1, 1, 0, 0\}$.

Worked Example 11.1 | Job selection

Jobs = $\{j_1, j_2, j_3, j_4\}$.

Variable size solution. The subsets of jobs are listed directly:

$$S_1 = \{j_1, j_4\}, \quad S_2 = \{j_1, j_2, j_4\}$$

Fixed size solution. For S_2 — the first job is completed, the second and third are not, and the fourth is completed:

$$S_1 = \{1, 0, 0, 1\}, \quad S_2 = \{1, 1, 0, 1\}$$

11.7 The three classifications

The branch-and-bound method is classified into three types based on the order in which the state space tree is searched.

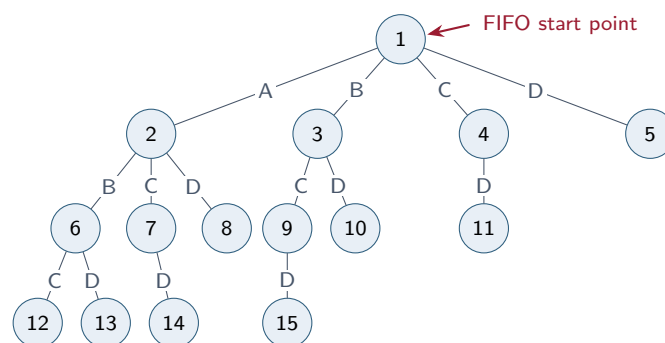
Variant	Data structure	Behaviour
FIFO branch and bound	Queue	Breadth-first search. All elements at a level are searched, then the next level, starting with the first child of the first node at the previous level
LIFO branch and bound	Stack	Nodes are pushed as they are generated; after all nodes of a level have been added, the topmost element is popped and explored
Least-cost (LC) branch and bound	Priority queue	Uses a heuristic cost function to compute bound values at each node; the node with the least cost value is selected as the next E-node

Key idea

The first two compute a cost at each node but *do not use it* to decide what to explore next — they follow a fixed traversal order. Least-cost B&B uses the cost to choose the next E-node, which is why it is generally faster than both LIFO and FIFO.

11.7.1 FIFO branch and bound

For the set $\{A, B, C, D\}$, the state space tree is generated breadth-first: consider element A first, then B , then C , then finally D .



Reading the tree:

Node	Meaning and expansion
Route 1–2–8	Elements A and D are used; B and C are not considered
Node 3	Element B considered, A not considered $\Rightarrow$ two options remain, C and D : create nodes 9 and 10
Node 4	Element C considered, A and B not $\Rightarrow$ one option remains, D : create node 11
Node 5	Only D considered; A, B, C not $\Rightarrow$ no elements left, no expansion
Node 6	Elements A and B considered $\Rightarrow$ two options, C and D : create nodes 12 and 13
Node 7	Elements A and C considered, B not $\Rightarrow$ one option, D : create node 14
Node 8	Elements A and D considered $\Rightarrow$ no elements left, no expansion
Node 9	Elements B and C considered, A not $\Rightarrow$ one option, D : create node 15

11.7.2 LIFO branch and bound

The same set, explored with a stack. Nodes are pushed as they are generated; after all nodes of a level have been added, the topmost element is popped and explored.

Step	Action	Effect
1	Expand node 1; push nodes 2, 3, 4, 5	Node 5 is on top
2	Pop node 5. It is the last element D	No scope for expansion
3	Pop node 4; expand. Element D is considered	Node 6 pushed
4	Pop node 6. It is element D	No scope for expansion
5	Pop node 3 (element B); expand	Nodes 7 and 8 pushed, working on C and D
6	Pop node 8 (element D)	No scope for expansion
7	Pop node 7 (element C); expand	Node 9 pushed, working on D
8	Pop node 9 (element D)	No scope for expansion
9	Pop node 2 (element A); expand	Nodes 10, 11, 12 pushed, working on B , C , D

All nodes are explored using the stack, following the LIFO principle.

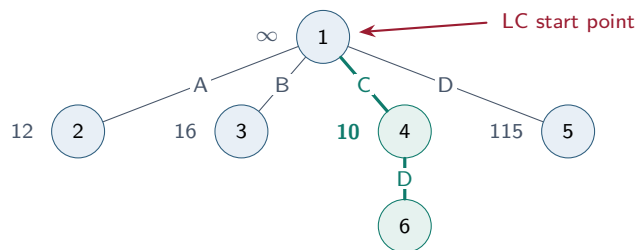
11.7.3 Least-cost branch and bound

Key idea

Consider a node whose cost has been determined. If this value is greater than U_0 — the best solution value found so far — this node and its children cannot give a better solution. The node is killed and its branches are not explored. This is what prevents the algorithm from exploring cases that are not worth it.

Worked Example 11.2 | LC selection on $\{A, B, C, D\}$

Node 1 has cost ∞ and expands into four nodes named 2, 3, 4 and 5. Assume the costs of nodes 2, 3, 4 and 5 are 12, 16, 10 and 115 respectively.



The node with minimum cost is node 4, so node 4 is explored. Node 4 is element C ; the only element that remains unexplored is D (we already decided not to select A and B). Node 4 therefore expands to a single node — call it node 6 — for element D . Node 6 has no elements left to explore, so there is no further scope for expansion.

Hence $\{C, D\}$ is the optimal way to choose for the least cost.

11.8 Least-cost branch and bound on the TSP

[Handout 9, slides 37–63. This is the fully worked example the course examines.]

Consolidation note

TSP was introduced in Chapter 6 §6.8 as the standard brute-force disaster $((n - 1)!$ routes) and appears again in Chapter 12 as an NP-hard problem. This section is the payoff: an exact method that solves it by examining a small fraction of those routes.

11.8.1 The problem

There are n cities and a travelling salesman. The salesman should start from a source city, travel through all the other cities *only once*, and at last travel back to the source city — in such a manner that the total travelling cost or distance is minimum.

Worked Example 11.3 | TSP by least-cost branch and bound

The cost matrix. Five cities, with an *asymmetric* cost matrix (the cost of $i \rightarrow j$ need not equal $j \rightarrow i$):

	1	2	3	4	5
1	∞	14	4	10	20
2	14	∞	7	8	7
3	4	5	∞	4	16
4	11	7	9	∞	2
5	18	7	17	4	∞

Step 1 — reduce the cost matrix.

Key idea

To **reduce** a matrix, do **row reduction** then **column reduction**. For row reduction, find the minimum number in each row and subtract it from every number in that row. For column reduction, find the minimum in each column and subtract it from every number in that column. Every row and every column must end up containing at least one zero.

Row reduction. Row minima are 4, 7, 4, 2, 4.

$$\text{Row reduction cost} = 4 + 7 + 4 + 2 + 4 = 21$$

	1	2	3	4	5	
1	∞	10	0	6	16	4
2	7	∞	0	1	0	7
3	0	1	∞	0	12	4 row minima
4	9	5	7	∞	0	2
5	14	3	13	0	∞	4

Column reduction. Column minima of the row-reduced matrix are 0, 1, 0, 0, 0.

Column reduction cost = 0 + 1 + 0 + 0 + 0 = 1

	1	2	3	4	5	
1	∞	9	0	6	16	
2	7	∞	0	1	0	this is the
3	0	0	∞	0	12	reduced matrix
4	9	4	7	∞	0	
5	14	2	13	0	∞	
	0	1	0	0	0	column minima

Reduced cost at the root (Node 1) = 21 + 1 = 22

This 22 is a **lower bound**: no tour can cost less, because every tour must use one entry from each row and each column, and we have subtracted 22 in total from quantities that must be paid regardless.

Step 2 — the rule for a child node.

Key idea

- To compute the reduced cost for the node corresponding to edge (i, j) :
1. Change all entries in **row** i and **column** j to ∞ , to avoid any other edge leaving i or entering j .
 2. Set $(j, 1)$ to ∞ , to prevent premature use of the edge $j \rightarrow 1$ back to the start.
 3. Reduce all rows and columns of the resulting matrix (except rows and columns that are entirely ∞). Call the total reduction r .
 4. The cost of the child node is then

$\underbrace{\text{cost}(j)}_{\text{child}}$

=

$\underbrace{\text{cost}(i)}_{\text{parent}}$

+

$\underbrace{M[i][j]}_{\text{entry in the parent's reduced matrix}}$

+

$\underbrace{r}_{\text{reduction cost of the new matrix}}$

Step 3 — expand the root.

Node	Edge	Computation	Cost
12	(1, 2)	$22 + 9 + 0$	31
13	(1, 3)	$22 + 0 + 7$	29
14	(1, 4)	$22 + 6 + 2$	30
15	(1, 5)	$22 + 16 + 4$	42

For example, node 13 takes edge (1, 3). The entry $M[1][3] = 0$ in the root's reduced matrix. Blanking row 1 and column 3, setting $(3, 1) = \infty$, and reducing costs a further 7. So $29 = 22 + 0 + 7$.

Least cost among the live nodes is **29 at node 13**, so node 13 becomes the next E-node.

Step 4 — expand node 13.

Node	Edge	Computation	Cost
132	(3, 2)	$29 + 0 + 2$	31
134	(3, 4)	$29 + 0 + 2$	31
135	(3, 5)	$29 + 12 + 4$	45

This is where LC-BB differs from a greedy dive

It is tempting to continue downward from node 132. **Do not.** The rule is: among *all* live nodes anywhere in the tree, find the least cost node to explore and branch from it. Live nodes now: 12=31, 14=30, 15=42, 132=31, 134=31, 135=45. Among the values 31, 31, 31, 45, 30, 42, the least is **30 at node 14** — which is one level *higher* in the tree. So node 14 is the next E-node.

Diving greedily from node 132 instead would terminate at the tour $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$ with cost **31** — a feasible tour, but *not* the optimum.

Step 5 — expand node 14.

Node	Edge	Computation	Cost
142	(4, 2)	$30 + 4 + 11$	45
143	(4, 3)	$30 + 7 + 7$	44
145	(4, 5)	$30 + 0 + 0$	30

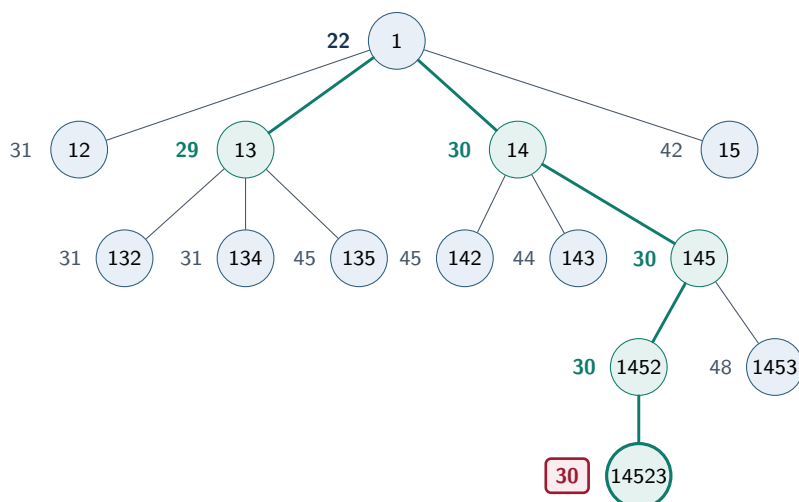
Least among all live nodes is 30 at node 145.

Step 6 — expand node 145, then node 1452.

Node	Edge	Computation	Cost
1452	(5, 2)	$30 + 0 + 0$	30
1453	(5, 3)	$30 + 11 + 7$	48
14523	(2, 3)	$30 + 0 + 0$	30

Node 14523 is a complete tour, and its cost 30 is the least among all leaf values (31, 31, 31, 45, 45, 44, 30, 48, 42). The search terminates.

The state space tree.



The solution. Node 14523, and back to 1:

$$1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$$

with the costs taken from the *original* adjacency matrix:

$$\text{Final cost} = 10 + 2 + 7 + 7 + 4 = \boxed{30}$$

Verification. Exhaustive enumeration of all $(n - 1)! = 24$ tours confirms 30 is the true optimum, and that it is unique. Branch and bound reached it after generating only **13** nodes instead of examining 24 complete tours — and the saving grows enormously with n .

The three places students lose marks on this question

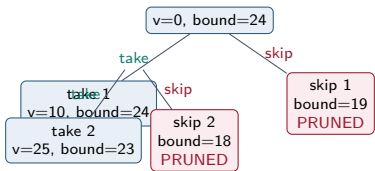
1. **Forgetting to set $(j, 1) = \infty$** when building a child matrix. Without it the algorithm may close the tour early and return a sub-tour rather than a full cycle.
2. **Diving instead of doing a true least-cost search.** After expanding a node, compare *every* live node in the tree, not just the children you have just created. In this example that is the difference between the answer 30 and the answer 31.
3. **Reading the final cost off the reduced matrices.** The bound at the leaf happens to equal the tour cost here, but you must state the tour and add up the edges from the *original* cost matrix to justify it.

11.9 Branch and bound on the 0/1 knapsack problem

[The DP solution is in Chapter 8 §8.8; the greedy failure is in Chapter 9.]

Knapsack is a *maximisation* problem, so the bound must be an **upper** bound. The natural choice is the **fractional knapsack** value of the remaining items, computable greedily in $\Theta(n \lg n)$ (Chapter 9 §9.6.1). It is always at least as good as the best 0/1 solution, since allowing fractions can only help — exactly the optimism a valid bound requires.

Element	Choice for 0/1 knapsack
Branching	At level i , two children: take item i , or leave item i
Bound	Current value + fractional-knapsack value of the remaining items in the remaining capacity
Prune when	$\text{Bound} \leq \text{best complete value so far}$, or the accumulated weight exceeds capacity
Solution form	Fixed size — a string of 0s and 1s, one per item



Illustrative shape only. Once a complete packing of value 22 exists, any node whose upper bound is ≤ 22 is discarded unexplored.

11.10 Comparison of the three exhaustive techniques

	Brute force	Backtracking	Branch and bound
Problem type	Any	Constraint satisfaction; find all/any solutions	Optimisation
Search order	Any	Depth-first	FIFO, LIFO, or least-cost
Pruning basis	None	Feasibility (constraint violated)	Optimality (bound cannot beat the best so far)
Needs a bound?	No	No	Yes
Finds	All solutions	All or any solution	The single best solution
Typical use	Small instances; baseline	N-Queens, Sudoku, graph colouring	TSP, 0/1 knapsack, job assignment, integer programming
Worst case	Full search space	Full search space	Full search space
Typical case	Full search space	Much smaller	Smaller still

All three are exponential in the worst case

Pruning improves the *typical* case, often dramatically, but does not change the worst-case complexity. If the bounding function is weak, or the instance is adversarial, branch and bound degenerates to brute force. Its value is empirical, not asymptotic — which is precisely why it is the practical tool for NP-hard problems rather than a refutation of their hardness.

11.11 Applications

Problem	Branching / bounding
Travelling salesman	Branch on the next city; bound by the reduced cost matrix
0/1 knapsack	Branch on take/skip; bound by the fractional solution
Job assignment	Branch on which worker takes the next job; bound by row/column minima
Integer linear programming	Branch on a fractional variable ($x \leq \lfloor v \rfloor$ or $x \geq \lceil v \rceil$); bound by the LP relaxation
Scheduling and timetabling	Branch on the next task; bound by remaining processing time
Facility location	Branch on open/closed facilities; bound by relaxed assignment costs

Chapter summary

- **Branch and bound** solves *optimisation* problems: branch to divide, bound to discard. Backtracking prunes on **feasibility**; B&B prunes on **optimality**.
- Backtracking always follows DFS. Classic B&B follows BFS, but the three recognised variants are **FIFO** (queue, BFS), **LIFO** (stack, DFS) and **least-cost** (priority queue, cheapest E-node first).
- Vocabulary: **live node** (generated, not fully expanded), **E-node** (currently expanding), **dead node** (killed or a leaf).
- The **bounding function** must be optimistic: a *lower* bound for minimisation, an *upper* bound for maximisation.
- Solutions are written in **variable size** form $\{A, B\}$ or **fixed size** form $\{1, 1, 0, 0\}$.
- **TSP by LC-BB**: reduce the matrix (row then column) to get the root bound; for a child on edge (i, j) blank row i and column j , set $(j, 1) = \infty$, reduce, and add. Always branch from the least-cost *live* node anywhere in the tree — not from the node you just created.
- Worked example: root bound 22, optimal tour $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$, cost $10 + 2 + 7 + 7 + 4 = 30$.
- Worst case remains exponential; the benefit is practical, not asymptotic.

Part IV

Complexity Classes and Reference Material

P, NP, NP-Hard and NP-Complete

[Sources: Handout 7 §§8–16; Lab Sheet 10 Q6]

12.1 Why this chapter exists

Chapters 6–11 presented five design paradigms. A natural question follows: *for which problems does a good algorithm exist at all?* For a large and important family of problems — travelling salesman, 0/1 knapsack, graph colouring, Hamiltonian cycle — decades of effort have produced no efficient algorithm, and no proof that none exists. This chapter names that situation.

12.2 Tractable and intractable

Category	Complexities	Practical meaning
Polynomial time	$O(1)$, $O(\lg n)$, $O(n)$, $O(n \lg n)$, $O(n^2)$, $O(n^3)$	Remains usable as n grows. Considered <i>tractable</i>
Non-polynomial time	$O(2^n)$, $O(n!)$	Becomes impractical rapidly. Considered <i>intractable</i>

The dividing line at “polynomial” is somewhat arbitrary — an $O(n^{100})$ algorithm is useless in practice — but it is robust: polynomial time is closed under composition, and is independent of the particular model of computation. That robustness is what makes it the right boundary for a theory.

12.3 The complexity classes

Class P

The set of decision problems solvable in polynomial time — that is, in $O(n^k)$ for some constant k — by a deterministic algorithm.

Class NP

The set of decision problems whose proposed solutions can be *verified* in polynomial time. NP stands for **Nondeterministic Polynomial time**: if you are handed a candidate solution (a **certificate**), you can check its validity quickly. NP does *not* stand for “non-polynomial”.

Key idea

The Sudoku analogy. Solving a hard Sudoku from scratch is difficult. But if someone hands you a completed grid, checking that it is correct takes seconds — scan each row, column and box. *Hard to solve, easy to verify* is exactly what NP captures.

NP-hard

A problem is **NP-hard** if it is at least as hard as every problem in NP: any NP problem can be transformed (“reduced”) into it in polynomial time. No polynomial-time algorithm is known that solves an NP-hard problem optimally in general. An NP-hard problem *need not itself be in NP* — it need not even be a decision problem.

NP-complete

$$\text{NP-complete} = \text{NP-hard} \cap \text{NP}$$

The hardest problems that are still *in* NP: verifiable in polynomial time, and at least as hard as everything else in NP. If a polynomial-time algorithm were found for any single NP-complete problem, every problem in NP could be solved in polynomial time — because all NP-complete problems reduce to one another in polynomial time.

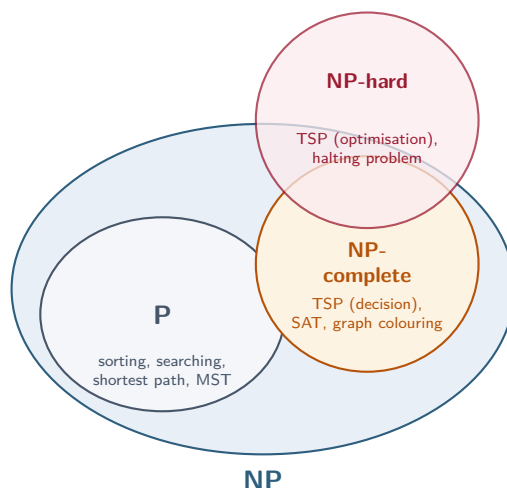


Diagram assumes $P \neq NP$. If $P = NP$, P and NP-complete would coincide with NP.

12.3.1 Deterministic and nondeterministic algorithms

Deterministic algorithm

Follows one specific sequence of steps. Given the same input, it always does the same thing. This is what real computers execute.

Nondeterministic algorithm

A theoretical model that can “guess” the correct solution and then verify it in polynomial time. It is a device for *defining* NP, not something that can be built.

12.4 The P versus NP question

Key idea

Is $P = NP$? That is: if a solution can be *verified* quickly, can it also be *found* quickly? Nobody knows. It is one of the seven Millennium Prize Problems and the most famous open question in computer science.

If $P = NP$

Every quickly-verifiable problem would also be quickly solvable. Consequences: most public-key cryptography would collapse (factoring and discrete-log-based schemes rely on the gap); optimisation, logistics, scheduling and protein folding would become tractable; mathematical proof search would be automatable.

If $P \neq NP$

The gap between finding and checking is real and permanent. This is what most researchers believe, and what modern cryptography implicitly assumes.

$NP \neq NP\text{-complete}$

NP is the *group*; NP-complete problems are the hardest, most representative members of that group. Every problem in P is also in NP (if you can solve it quickly you can certainly verify it quickly), but not every NP problem is NP-complete.

12.5 Classic NP-complete and NP-hard problems

Problem	Statement
Travelling salesman	Find the shortest route visiting a set of cities exactly once and returning to the start
0/1 Knapsack	Choose items to maximise value under a weight limit
Graph colouring	Colour the vertices so no two adjacent vertices share a colour, using the fewest colours
Hamiltonian path / cycle	Find a path (or cycle) visiting every vertex exactly once
Subset sum	Determine whether some subset of a set of numbers sums to a target value
Boolean satisfiability (SAT)	Determine whether a Boolean formula can be made true — the first problem proved NP-complete (Cook, 1971)
Vertex cover, set cover, clique	Classic graph and set optimisation problems
Bin packing, job-shop scheduling	Classic operations-research problems

Key idea

Three of these are algorithms you have already met from the other side: TSP appears in Chapters 6 and 11, 0/1 knapsack in Chapters 8 and 9, and Hamiltonian cycle in Chapter 10. Their NP-hardness is precisely *why* those chapters resorted to exponential search, pseudo-polynomial DP, or heuristics.

12.6 How NP-hard problems are handled in practice

Strategy	What it gives up	Example
Approximation algorithms	Optimality, but with a <i>provable</i> bound on how far off it can be	MST-based 2-approximation for metric TSP; greedy set cover
Heuristics	Any guarantee at all — but usually fast and good	Greedy nearest-neighbour, simulated annealing, genetic algorithms, 2-opt local search
Special-case algorithms	Generality — exploit structure in the instances you actually face	Planar graph colouring; TSP on Euclidean points; knapsack with small weights
Exponential with pruning	Speed on large inputs, but exact on small ones	Branch and bound, backtracking (Chapters 10–11)
Pseudo-polynomial DP	Nothing, provided the numeric parameter is small	0/1 knapsack in $\Theta(nW)$

Approximation ratio

For a minimisation problem, an algorithm has approximation ratio ρ if for every instance it returns a solution of cost at most ρ times the optimum. A 2-approximation for TSP is guaranteed never to produce a tour more than twice the length of the shortest one.

Why greedy matters here

Chapter 9 noted that a greedy algorithm “does not always guarantee the optimal solution but often provides a good approximate solution”. This chapter supplies the reason that matters: for NP-hard problems there *is* no known efficient optimal algorithm, so a fast greedy heuristic with a measurable approximation ratio is often the best available option, not a compromise.

12.7 Lab Sheet 10 Q6 — worked solution

[Answer each with “some”, “all”, “none” or “unknown”.]

Statement	Answer	Reason
(a) Problems in P are also in NP	all	$P \subseteq NP$: if you can solve it in polynomial time, you can verify a certificate in polynomial time (just re-solve it)
(b) Problems in NP-hard are at least as hard and maybe harder than all problems in NP	all	This is the definition of NP-hard
(c) Problems in NP-complete are at least as hard and maybe harder than all problems in NP	all	NP-complete problems are NP-hard by definition, so the same statement applies
(d) Problems in NP-complete are at least as hard and maybe harder than all problems in P	all	Since $P \subseteq NP$ and NP-complete is at least as hard as all of NP, it is at least as hard as all of P
(e) Problems in NP-complete are also in NP-hard	all	NP-complete = NP-hard $\cap$ NP, so every NP-complete problem is NP-hard
(f) Problems in NP-hard are also in NP-complete	some	Only those NP-hard problems that additionally lie in NP. Others — e.g. the halting problem, or the TSP <i>optimisation</i> version — are NP-hard but not in NP
(g) Problems in NP are also in NP-hard	some	The NP-complete ones are. Whether <i>all</i> NP problems are NP-hard depends on whether $P = NP$; under the standard assumption $P \neq NP$, problems in $P \subsetneq NP$ are not NP-hard

Part (g) is the subtle one

If $P = NP$ then every non-trivial problem in NP would be NP-hard, making the answer “all”. Since P vs. NP is open, an equally defensible answer to (g) is **unknown**. State your assumption: “*some*, assuming $P \neq NP$ ” is the safest full answer.

Chapter summary

- **Polynomial** time ($n, n \lg n, n^2, n^3$) is tractable; **exponential** and **factorial** time ($2^n, n!$) is not.
- **P**: solvable in polynomial time. **NP**: *verifiable* in polynomial time (“Nondeterministic Polynomial”, not “non-polynomial”).
- **NP-hard**: at least as hard as every problem in NP; need not be in NP. **NP-complete** = NP-hard $\cap$ NP.
- $P \subseteq NP$. Whether $P = NP$ is the great open question; solving any one NP-complete problem in polynomial time would solve all of them.
- Classic NP-complete problems: TSP (decision), 0/1 knapsack, graph colouring, Hamiltonian path/cycle, subset sum, SAT.
- In practice, NP-hard problems are attacked with approximation algorithms (provable bounds), heuristics (no bounds), special-case algorithms, branch and bound, or pseudo-polynomial DP.

Lab Sheet Worked Solutions

Where every lab question is answered

Most quantitative lab problems are worked *in place*, inside the chapter that develops the relevant technique — that is the point of consolidating. This appendix holds the remainder, plus an index so nothing is lost.

Lab sheet	Topic	Solved in
LS01	Abstraction, decomposition, flowchart	Ch. 1 §1.2 (modules table); §A.1 below (flowchart)
LS02 Q1	Pattern recognition on exam data	§A.2 below
LS02 Q2	Functional decomposition (hospital)	§A.2 below
LS02 Q3	“Book Appointment” flowchart	Ch. 1, Worked Example 1.5
LS02 Q4	LeetCode practice	§A.5 below
LS03 Part 1	Sequential vs. binary search traces	Ch. 2, Worked Example 2.1
LS03 Part 2	Fibonacci user-growth analysis	§A.3 below
LS03 Part 3	Searching Fibonacci data	§A.3 below
LS04	Trace tables and trace trees	Ch. 2 §2.7, Worked Examples 2.2–2.5
LS05 Q1	Loop complexity (4 snippets)	Ch. 4, Worked Example 4.11
LS05 Q2–Q3	Linear and binary search traces	Ch. 2, Worked Example 2.1
LS05 Q4–Q5	Basic-operation identification	§A.4 below
LS05 Q6	Choosing an algorithm as data grows	Ch. 4, Worked Example 4.15
LS06 A	Loop complexity (6 snippets)	Ch. 4, Worked Example 4.12
LS06 B	Writing and analysing 3 algorithms	Ch. 4, Worked Example 4.14
LS06 C	1000× faster computer	Ch. 4, Worked Example 4.14
LS06 D	Classifying a polynomial	Ch. 4, Worked Example 4.13
LS06 E	<code>add_them</code>	Ch. 4, Worked Example 4.13
LS07 Q1	Merge sort trace and comparisons	Ch. 7, Worked Example 7.5
LS07 Q2	Brute-force maximum subarray	Ch. 6, Worked Example 6.3
LS07 Q3	Divide-and-conquer maximum subarray	Ch. 7, Worked Example 7.11
LS08 A	Quick sort on election data	Ch. 7, Worked Example 7.8
LS08 B	Merge sort; parallel merge sort	Ch. 7, Worked Example 7.6
LS08 C	Strassen’s algorithm	Ch. 7, Worked Examples 7.9–7.10
LS08 D	DP tabulation, $F(8)$	Ch. 8, Worked Example 8.1
LS09 Q1	Brute-force route planning	Ch. 6, Worked Example 6.2
LS09 Q2	Merge sort / quick sort / counting	Ch. 7, Worked Examples 7.7–7.8
LS09 Q3	Matrix multiplication and Strassen	Ch. 7, Worked Examples 7.9–7.10
LS09 Q4	Memoisation vs. tabulation, $R(8)$	Ch. 8, Worked Example 8.2
LS10 Q1–Q5	Kruskal and Prim on networks	Ch. 9 §9.10.1
LS10 Q6	NP-completeness quiz	Ch. 12 §12.7

Lab sheets with image-based data

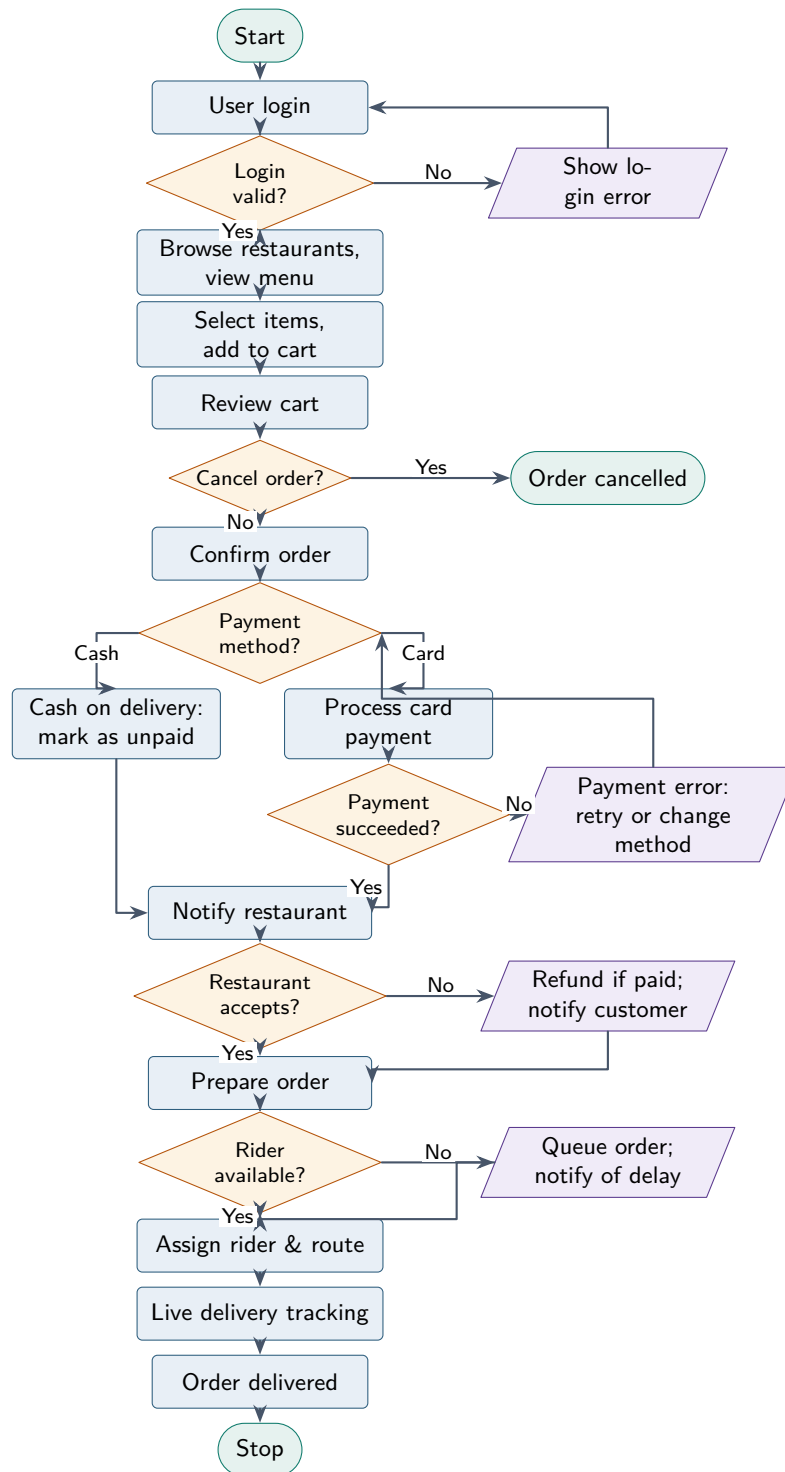
Lab Sheet 08 Part C, Lab Sheet 09 Part 3 and all of Lab Sheet 10 present their matrices and graphs as *images*, whose exact values could not be extracted for this note. The worked examples in Chapters 7 and 9 therefore use representative instances of the same size and structure. **The method transfers exactly** — substitute your own numbers into the same tables and the same steps.

A.1 Lab Sheet 01 — Order-placement flowchart

[The high-level requirements and module decomposition are in Chapter 1 §1.2.]

The flowchart must cover: user login, restaurant browsing, menu viewing, item selection, cart review, order confirmation, a payment-method decision, cash and card payment paths, cancellation before final confirmation, restaurant notification, order preparation, delivery assignment, delivery tracking and order completion — with five mandatory decision points.

The chart below uses a single main column with all exception handling on the right, which keeps every flow line straight and un-crossed.



Mandatory decision point	Handling
Login fails	Show an error and return to the login step (with a retry limit in a real system)
User cancels before confirmation	Terminate cleanly at “Order cancelled”; no restaurant notification is sent
Card payment fails	Show an error, offer a retry or an alternative payment method, and return to the payment-method decision
Restaurant rejects the order	Refund any card payment, notify the customer, and offer alternative restaurants
No rider available	Queue the order and notify the customer of the delay; re-check availability rather than cancelling

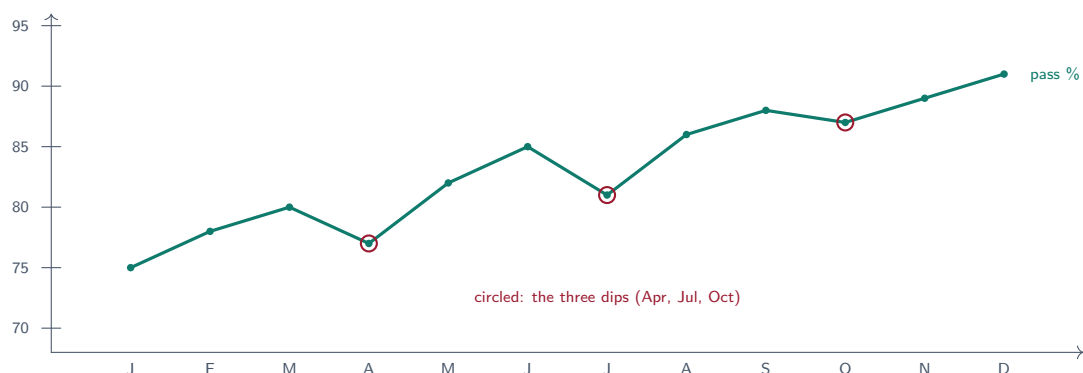
Exam technique

Two things that lose marks here: (i) a failure branch that simply stops with no message or recovery; (ii) an unlabelled diamond exit. Every diamond must have exactly two exits, both labelled, and every path must reach a terminator.

A.2 Lab Sheet 02 — Pattern recognition and decomposition

A.2.1 Q1 — Examination results analysis

Month	Students	Avg. assignment	Avg. final exam	Pass %
January	120	68	70	75
February	125	70	72	78
March	130	72	74	80
April	128	69	71	77
May	135	74	76	82
June	140	75	78	85
July	138	73	75	81
August	145	76	79	86
September	150	78	81	88
October	148	77	80	87
November	152	79	82	89
December	155	80	84	91



(A) Highest and lowest pass percentages, and the pattern. Highest: **December, 91%**. Lowest: **January, 75%**. The dominant pattern is a *sustained upward trend* across the year, from 75% to 91% — a rise of 16 percentage points. Superimposed on it is a *regular three-month dip*: April (−3 from March), July (−4 from June) and October (−1 from September). Each dip is followed by recovery to a new high, so the trend is a rising sawtooth rather than a smooth climb. The dips fall at the start of each quarter after April, suggesting an academic-calendar cause — a new term, holiday period, or module transition.

(B) Trend in student numbers. Enrolment rises from 120 to 155, an increase of 35 students (29%). The rise is not monotonic: April (128 vs. 130), July (138 vs. 140) and October (148 vs. 150) each show a small decline — *the same three months as the pass-rate dips*. This correlation is the most valuable pattern in the dataset: whatever causes students to drop out in those months also depresses performance.

(C) Consistency of assignment and exam marks. The two move together throughout, with the final exam average consistently **2–4 marks above** the assignment average. The gap is remarkably stable:

Month	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Gap (exam – assignment)	2	2	2	2	2	3	2	3	3	3	3	4

This indicates that (i) continuous assessment is a *reliable predictor* of exam performance, so assignment marks can be used as an early-warning signal; and (ii) the slight widening of the gap (2 → 4) suggests students are increasingly well-prepared for exams relative to coursework, or that exam difficulty eased slightly.

(D) Prediction for next January. Two competing patterns must be reconciled:

- *Trend extrapolation*: the average month-on-month rise is $(91 - 75)/11 \approx 1.45$ percentage points, predicting ≈ 92 – 93% .
- *Seasonal dip*: January begins a new academic year, and the previous January was the year's minimum.

Prediction: pass percentage ≈ 88 – 90% ; average final exam mark ≈ 84 – 85 . The reasoning: the underlying trend is strongly positive, but a new intake and a term boundary make a small dip likely, mirroring April, July and October. A confident single point estimate would be **89% and 84 marks**, with the caveat that year-boundary effects are not observable in a single year of data.

(E) Two strategies to improve performance.

1. **Targeted intervention at the quarter boundaries.** Since April, July and October show simultaneous drops in both enrolment and performance, schedule revision sessions, catch-up clinics and re-engagement contact in the weeks preceding those months. The dips are predictable, so the intervention can be too.
2. **Early-warning system based on assignment marks.** Because assignment averages track exam averages with a stable 2–4 mark offset, a student's assignment performance predicts their exam outcome. Flag any student whose assignment mark falls more than one standard deviation below the cohort mean and offer support *before* the exam rather than after.

(F) How pattern recognition supports decision-making. Patterns turn a table of numbers into a basis for action, because they convert observations into *predictions*. Concretely: having identified the three-month dip cycle, the department can decide to **move the mid-term assess-**

ment two weeks earlier in each quarter so that struggling students are identified before the dip rather than during it. Without the pattern, the same data supports only the retrospective observation “April was worse than March”.

A.2.2 Q2 — Functional decomposition of a hospital appointment system

(A) Major functions and their sub-functions.

Major function	Sub-functions
Patient registration	(i) Capture and validate personal and contact details; (ii) Generate a unique patient ID; (iii) Create login credentials; (iv) Verify identity documents
Appointment booking	(i) Search doctors by specialty, location and availability; (ii) Display available time slots; (iii) Reserve and confirm a slot; (iv) Issue booking confirmation
Appointment cancellation and rescheduling	(i) Retrieve the patient’s existing bookings; (ii) Release the slot back to the doctor’s schedule; (iii) Apply cancellation-window rules; (iv) Notify the doctor and the patient
Medical records management	(i) Create and update consultation records; (ii) Attach test results and prescriptions; (iii) Enforce access control and audit logging; (iv) Retrieve patient history
Doctor schedule management	(i) Define working hours and slot durations; (ii) Block leave and unavailable periods; (iii) Publish the schedule; (iv) Handle emergency overrides
Notifications	(i) Send booking confirmations; (ii) Send appointment reminders; (iii) Alert on cancellation or reschedule
Billing and payments	(i) Generate invoices; (ii) Process payments and refunds; (iii) Handle insurance claims
Reporting and administration	(i) Utilisation and no-show reports; (ii) User and role management; (iii) System configuration

(B) How functional decomposition manages complexity.

Decomposition converts one problem that nobody can hold in their head into a set of problems that each have a clear boundary, a clear input and a clear output. The benefits are: independent development and testing; parallel work by different team members; localised change (a fix in one function does not risk another); and comprehensibility, since each function can be understood on its own.

Concrete illustration from the decomposition above. Consider the requirement “a patient must not be able to double-book the same time slot”. Without decomposition this rule would be tangled through the whole booking flow. With decomposition it belongs entirely to one sub-function — *Reserve and confirm a slot* — which can be given a precise contract (“atomically reserve slot s for patient p , or fail”), implemented once, unit-tested in isolation, and reused by both the booking and the rescheduling functions. A later change to the rule (say, allowing double-booking for emergencies) touches exactly one place.

A.3 Lab Sheet 03 — Fibonacci growth and searching

[Note the indexing convention: this lab sheet uses Month 1 $\rightarrow$ 1, Month 2 $\rightarrow$ 1, Month 3 $\rightarrow$ 2, i.e. $a_1 = a_2 = 1$. This is *not* Handout 2’s $f_0 = 0$ convention. See the pitfall in Chapter 2 §2.4.]

Month	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
New user groups	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

A.3.1 Part 2 — Growth analysis

(1) Users in specific months.

$$6\text{th month} = 5 + 3 = \mathbf{8}, \quad 8\text{th month} = 13 + 8 = \mathbf{21}.$$

(2) Total users by the end of the 8th month.

$$1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 = \mathbf{54}$$

(Useful identity: $\sum_{i=1}^n f_i = f_{n+2} - 1$. Here $f_{10} - 1 = 55 - 1 = 54$. ✓)

(3) Users in the 12th month. $89 + 55 = 144$.

(4) **Average monthly growth rate, months 6 to 12.** Two defensible readings — state which you use:

- *Average absolute growth:* $\frac{144 - 8}{12 - 6} = \frac{136}{6} \approx \mathbf{22.7}$ new user groups per month.
- *Average compound growth rate:* $\left(\frac{144}{8}\right)^{1/6} - 1 = 18^{1/6} - 1 \approx 0.619 = \mathbf{61.9\%}$ per month.

The second is the more meaningful figure for capacity planning, and it converges to $\varphi - 1 \approx 61.8\%$ — the growth rate of the Fibonacci sequence itself.

(5) Total users, months 1 to 12.

$$1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 + 55 + 89 + 144 = \mathbf{376}$$

Check with the identity: $f_{14} - 1 = 377 - 1 = 376$. ✓

(6) **A tribonacci variant.** If each month's growth is the sum of the previous *three* months, starting 1, 1, 2:

Month	1	2	3	4	5	6
Value	1	1	2	$1+1+2=4$	$1+2+4=7$	$2+4+7=13$

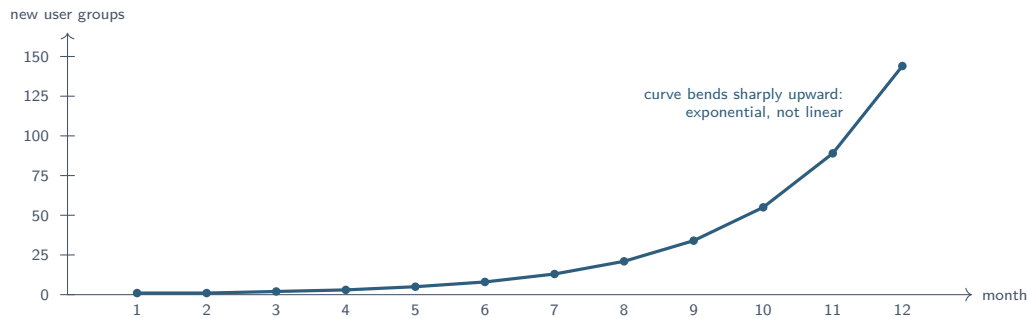
Comparison: the tribonacci 6th month is **13** against the Fibonacci **8** — 62.5% higher. The tribonacci growth constant is approximately 1.839 versus $\varphi \approx 1.618$, so the gap widens rapidly with n . For capacity planning this matters enormously: a system sized on the Fibonacci assumption would be under-provisioned within months if growth actually followed the tribonacci pattern.

(7) Month with the highest growth rate (months 1–12). Month-on-month increases:

Month	2	3	4	5	6	7	8	9	10	11	12
Increase	0	1	1	2	3	5	8	13	21	34	55

Month 12, with an increase of **55** user groups. Note that the sequence of increases is itself the Fibonacci sequence shifted by two — a direct consequence of $f_n - f_{n-1} = f_{n-2}$.

(8) Graph and observed trends.



How to construct it: plot month on the x -axis and new user groups on the y -axis, and join the points.

Trends observed: the curve is flat for the first five months and then rises steeply — the signature of **exponential growth** with ratio $\varphi \approx 1.618$. Months 1–5 add only 12 user groups in total; month 12 alone adds 55. The practical implication is that infrastructure planning must be based on the *growth rate*, not on recent absolute numbers: a server provisioned in month 5 for “twice current load” would be saturated by month 7.

A.3.2 Part 3 — Applying search algorithms to Fibonacci data

The 15-month list is $[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]$ — already sorted in non-decreasing order, so both search algorithms apply.

(I) Sequential search for 34. Compare positions 1, 2, 3, ... in turn against 34: 1, 1, 2, 3, 5, 8, 13, 21 all fail; position 9 holds 34. ✓

Position **9**, **9** comparisons.

(II) Binary search for 144.

Iter.	low	high	mid	$S[\text{mid}]$	Action
1	1	15	$\lfloor 16/2 \rfloor = 8$	21	$144 > 21 \Rightarrow \text{low} \leftarrow 9$
2	9	15	$\lfloor 24/2 \rfloor = 12$	144	found at position 12

Position **12**, **2** comparisons.

(III) Finding the month in which users reach 233.

Sequential search: scan from position 1, comparing each value with 233. It is found at position 13 after **13 comparisons**.

Binary search:

Iter.	low	high	mid	$S[\text{mid}]$	Action
1	1	15	8	21	$233 > 21 \Rightarrow \text{low} \leftarrow 9$
2	9	15	12	144	$233 > 144 \Rightarrow \text{low} \leftarrow 13$
3	13	15	14	377	$233 < 377 \Rightarrow \text{high} \leftarrow 13$
4	13	13	13	233	found at position 13

4 comparisons — versus 13 for the sequential search.

	Sequential search	Binary search
Advantages	Works on unsorted data; trivial to implement; optimal when the target is near the front; no preprocessing	Far fewer comparisons — $O(\lg n)$ vs. $O(n)$; the gap widens dramatically with n
Disadvantages	$O(n)$ — unusable for large datasets	Requires the data to be sorted; more complex to implement; poor cache locality on very large arrays
Best here	Only if the list were unsorted, or very short	Recommended — the growth data is inherently sorted (it is monotonically increasing), so the precondition is free

(IV) Range search: months where users fall between 21 and 233.

Run two binary searches — one for the *lower* boundary and one for the *upper*.

Lower boundary (first position with value ≥ 21). $low = 1$, $high = 15$, $mid = 8$, $S[8] = 21$ — an exact match at position 8. Because the list is strictly increasing from position 3 onward, position 8 is the first such position. **1 comparison.**

Upper boundary (last position with value ≤ 233). From the trace in part (III), 233 is at position 13, found in 4 comparisons.

Result: positions **8 to 13** inclusive, i.e. **months 8–13**, with values 21, 34, 55, 89, 144, 233 — six months.

Steps in general. (i) Binary-search for the lower bound: whenever $S[mid] \geq \text{low target}$, record mid as a candidate and move $high \leftarrow mid - 1$ to look further left. (ii) Binary-search for the upper bound: whenever $S[mid] \leq \text{high target}$, record mid and move $low \leftarrow mid + 1$ to look further right. (iii) The answer is the index range between the two recorded positions. Total cost $O(\lg n)$ — two logarithmic searches rather than a linear scan.

The duplicate 1 at positions 1 and 2

The list is *non-decreasing* rather than strictly increasing, because $f_1 = f_2 = 1$. Plain binary search still works (it will find a position holding the key), but for a range query you must use the boundary-seeking variants above — and this is exactly where the upper-mid convention from Chapter 2 §2.3 becomes necessary.

A.4 Lab Sheet 05 Q4–Q5 — Basic operations in applied scenarios

Worked Example A.1 | Q4 — Counting likes on posts

```

1 FOR each post IN posts
2   totalLikes = totalLikes + post.likes

```

(a) **Basic operation.** The addition `totalLikes + post.likes`.

(b) **Executions for n posts.** Exactly n — once per post, with no early exit and no nesting. This is an *every-case* complexity: $B(n) = A(n) = W(n) = n$.

(c) **Time complexity.** $\Theta(n)$ — linear.

(d) **Effect of doubling the number of posts.** The execution time *doubles*. In a linear algorithm, work is directly proportional to input size: $T(2n) = 2n = 2 \cdot T(n)$. Going from 1

million to 2 million posts takes twice as long, not four times — which is exactly the property that makes linear algorithms safe to deploy at scale.

Worked Example A.2 | Q5 — All-pairs plagiarism comparison

```

1 FOR i FROM 0 TO n-1
2   FOR j FROM 0 TO n-1
3     Compare(student[i], student[j])

```

- (a) **Basic operation.** The call `Compare(student[i], student[j])`.
 (b) **Number of comparisons.** The loops are nested and each runs n times, so the basic operation executes $n \times n = n^2$ times.
 (c) **Time complexity.** $\Theta(n^2)$ — quadratic.
 (d) **Why it becomes expensive.** Doubling the class size *quadruples* the work:

Class size n	Comparisons n^2	Time at 1 ms per comparison
50	2 500	2.5 s
100	10 000	10 s
500	250 000	4.2 minutes
1 000	1 000 000	16.7 minutes
5 000	25 000 000	6.9 hours

For a university-wide check across 5 000 submissions the naive approach takes most of a day, and document comparison costs far more than 1 ms in reality.

Two improvements worth stating.

1. **Halve the work immediately.** Comparison is symmetric and self-comparison is meaningless, so j should run from $i+1$ rather than 0. That gives $\binom{n}{2} = \frac{n(n-1)}{2}$ comparisons — still $\Theta(n^2)$, but with half the constant, and it removes n spurious perfect matches on the diagonal.
2. **Reduce the candidate set.** Compute a cheap fingerprint (e.g. a hash of each document's k -word shingles) in $\Theta(n)$ time, group documents by shared fingerprints, and run the expensive pairwise comparison only within groups. This is the technique real plagiarism detectors use, and it turns a quadratic problem into a near-linear one for typical inputs.

A.5 Lab Sheet 02 Q4 and 03 Part 4 — LeetCode practice

These parts ask for screenshots of completed LeetCode problems, so no written answer is possible here. What is worth recording is which technique each problem is testing, so the practice connects to the syllabus.

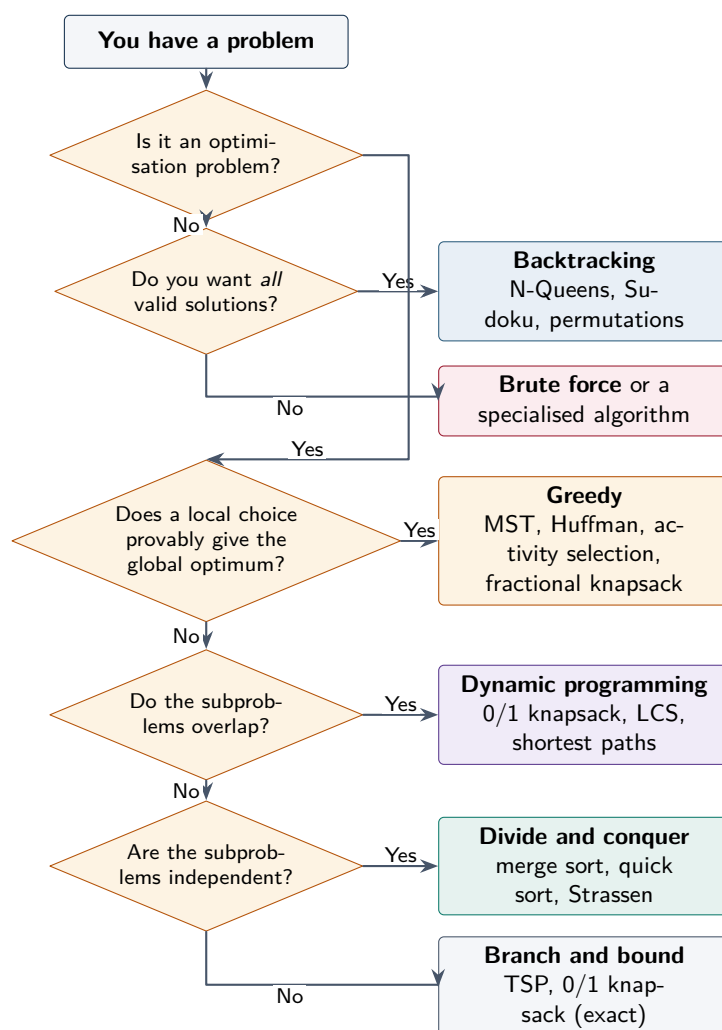
Problem	Technique	Optimal approach
Two Sum	Hashing	One pass storing seen values in a hash map: $\Theta(n)$ time, $\Theta(n)$ space. The naive nested loop is $\Theta(n^2)$
Palindrome Number	Arithmetic	Reverse half the digits; $\Theta(\lg n)$ in the value, $O(1)$ space. Avoid string conversion
Roman to Integer	Linear scan	Add each numeral's value, subtracting when a smaller numeral precedes a larger one: $\Theta(n)$
Longest Common Prefix	Vertical scan	Compare character i across all strings; stop at the first mismatch: $\Theta(S)$ where S is total input length
Valid Parentheses	Stack	Push openers, pop and match on closers; valid iff the stack is empty at the end: $\Theta(n)$
Binary Search	Divide and conquer	Chapter 2 §2.3; $\Theta(\lg n)$
Fibonacci Number	Dynamic programming	Chapter 8 §8.6; $\Theta(n)$ time, $O(1)$ space with two variables
Search Insert Position	Modified binary search	Return <code>low</code> when the loop exits — that is the insertion point; $\Theta(\lg n)$

Exam technique

Two of these are directly examinable course content. *Search Insert Position* is the standard variant question on binary search: when the key is absent, the value of `low` at loop exit is exactly where the key would go. *Fibonacci Number* is the DP question in disguise — the accepted solution must be $\Theta(n)$, not the naive recursion.

Master Reference Tables

B.1 Choosing a paradigm



Read the “Yes” branch of the first question carefully

The path from “Is it an optimisation problem? — Yes” runs down the *left* column through three further questions. Only if all three fail do you arrive at branch and bound.

B.2 The six paradigms side by side

Paradigm	Core move	Requires	Typical cost	Examples
Brute force	Enumerate everything	Nothing	$O(2^n)$, $O(n!)$	Padlock, exhaustive TSP
Divide & conquer	Split, solve, combine	<i>Independent</i> subproblems	$\Theta(n \lg n)$ typically	Merge sort, quick sort, binary search, Strassen
Dynamic programming	Store and reuse subproblem answers	<i>Overlapping</i> subproblems + optimal substructure	$\Theta(n^2)$, $\Theta(nW)$	Fibonacci, LCS, 0/1 knapsack, Floyd–Warshall
Greedy	Best local choice, never revised	Greedy-choice property + optimal substructure	$\Theta(n \lg n)$	Kruskal, Prim, Huffman, activity selection, fractional knapsack
Backtracking	Build incrementally; prune on <i>constraint</i> violation	Constraints to prune on	$O(n!)$ worst	N-Queens, Sudoku, Hamiltonian cycle, graph colouring
Branch & bound	Prune on <i>bound</i> vs. incumbent	An optimistic bounding function	$O(2^n)$ worst	TSP, 0/1 knapsack (exact), integer programming

B.3 Complexity of every algorithm in the course

Algorithm	Best	Average	Worst	Space
<i>Searching</i>				
Sequential search	$\Omega(1)$	$\Theta(n)$	$O(n)$	$O(1)$
Binary search (iterative)	$\Omega(1)$	$\Theta(\lg n)$	$O(\lg n)$	$O(1)$
Binary search (recursive)	$\Omega(1)$	$\Theta(\lg n)$	$O(\lg n)$	$O(\lg n)$
<i>Sorting</i>				
Bubble sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$\Omega(n \lg n)$	$\Theta(n \lg n)$	$O(n \lg n)$	$O(n)$
Quick sort	$\Omega(n \lg n)$	$\Theta(n \lg n)$	$O(n^2)$	$O(\lg n)$
Heap sort	$\Omega(n \lg n)$	$\Theta(n \lg n)$	$O(n \lg n)$	$O(1)$
<i>Matrix and numeric</i>				
Matrix addition		$\Theta(n^2)$ every case		$O(n^2)$
Conventional matrix mult.		$\Theta(n^3)$ every case		$O(n^2)$
Strassen		$\Theta(n^{\lg 7}) \approx \Theta(n^{2.807})$		$O(n^2)$
Fibonacci (naive recursion)		$\Theta(\varphi^n) \approx \Theta(1.618^n)$		$O(n)$
Fibonacci (DP)		$\Theta(n)$		$O(n)$ or $O(1)$
<i>Maximum subarray</i>				
Brute force (triple loop)		$\Theta(n^3)$		$O(1)$
Brute force (running sum)		$\Theta(n^2)$		$O(1)$
Divide and conquer		$\Theta(n \lg n)$		$O(\lg n)$
Kadane's algorithm		$\Theta(n)$		$O(1)$
<i>Dynamic programming</i>				
Longest common subsequence		$\Theta(mn)$		$O(mn)$
0/1 knapsack		$\Theta(nW)$		$O(nW)$
Coin change (min coins)		$\Theta(W C)$		$O(W)$
Floyd–Warshall		$\Theta(V^3)$		$O(V^2)$
<i>Greedy</i>				
Activity selection		$\Theta(n \lg n)$		$O(n)$
Fractional knapsack		$\Theta(n \lg n)$		$O(1)$
Kruskal's MST		$O(E \lg E) = O(E \lg V)$		$O(V)$
Prim's MST (matrix)		$O(V^2)$		$O(V)$
Prim's MST (heap)		$O(E \lg V)$		$O(V)$
Huffman coding		$\Theta(n \lg n)$		$O(n)$
<i>Backtracking and exhaustive</i>				
N-Queens	—	—	$O(N!)$	$O(N)$
Hamiltonian cycle	—	—	$O(n^n)$	$O(n)$
Brute-force TSP		$\Theta(n!)$		$O(n)$
Subset enumeration		$\Theta(2^n)$		$O(n)$

B.4 Sorting algorithm properties

Algorithm	Stable?	In-place?	Worst case	Use when
Bubble sort	Yes	Yes	$O(n^2)$	Teaching only
Insertion sort	Yes	Yes	$O(n^2)$	n small, or data nearly sorted
Selection sort	No	Yes	$O(n^2)$	Minimising the number of writes
Merge sort	Yes	No	$O(n \lg n)$	Stability required; linked lists; external sorting; parallel
Quick sort	No	Yes	$O(n^2)$	General-purpose in-memory array sorting
Heap sort	No	Yes	$O(n \lg n)$	Guaranteed $n \lg n$ with $O(1)$ space

B.5 Asymptotic notation quick reference

Notation	Definition ($\forall n \geq N$)	Quantifier on c	Analogue
$O(f)$	$g(n) \leq cf(n)$	$\exists c > 0$	$\leq$
$\Omega(f)$	$g(n) \geq cf(n)$	$\exists c > 0$	$\geq$
$\Theta(f)$	$cf(n) \leq g(n) \leq df(n)$	$\exists c, d > 0$	$=$
$o(f)$	$g(n) \leq cf(n)$	$\forall c > 0$	$<$
$\omega(f)$	$g(n) \geq cf(n)$	$\forall c > 0$	$>$

B.5.1 Simplification and structural rules

Rule	Example
Drop constant factors	$5n \Rightarrow O(n)$
Drop lower-order terms	$n^2 + n + 10 \Rightarrow O(n^2)$
Ignore the logarithm base	$O(\log_2 n) = O(\ln n)$
Never simplify a product	$n \lg n$ stays $n \lg n$
Sequential statements: add , keep the dominant	$O(n) + O(n^2) = O(n^2)$
Nested loops: multiply	$O(n) \times O(n) = O(n^2)$
if/else : take the maximum branch	$\max(O(n), O(n^2)) = O(n^2)$

B.5.2 Loop patterns

Pattern	Complexity	Tell
for i <- 1 to n	$O(n)$	Index increases by a constant
for i <- 1 to n step c	$O(n)$	Constant step; constant discarded
i <- i * 2 or i <- i / 2	$O(\lg n)$	Index multiplied/divided by a constant
Two nested n -loops	$O(n^2)$	Multiply
Nested n -loop and m -loop	$O(nm)$	Independent bounds
Outer n , inner $\lg n$	$O(n \lg n)$	Multiply
Triangular: inner runs to i	$O(n^2)$	$\sum i = \frac{n(n+1)}{2}$
Inner bound doubles each outer pass	$O(n)$	Sum the geometric series , do not multiply
Inner runs $\approx n/i$ times	$O(n \lg n)$	Harmonic sum $H_n \approx \ln n$

B.6 Recurrences and the Master Theorem

For $T(n) = aT(n/b) + f(n)$, compare $f(n)$ with $n^{\log_b a}$:

Case	Condition	Result	Intuition
1	$f(n) = O(n^{\log_b a - \epsilon})$	$\Theta(n^{\log_b a})$	Leaves dominate
2	$f(n) = \Theta(n^{\log_b a})$	$\Theta(n^{\log_b a} \lg n)$	Every level costs the same
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$ (plus regularity)	$\Theta(f(n))$	Root dominates

Recurrence	a	b	$f(n)$	Solution
$T(n) = T(n/2) + 1$	1	2	1	$\Theta(\lg n)$
$T(n) = 2T(n/2) + n$	2	2	n	$\Theta(n \lg n)$
$T(n) = 2T(n/2) + 1$	2	2	1	$\Theta(n)$
$T(n) = 2T(n/2) + n^2$	2	2	n^2	$\Theta(n^2)$
$T(n) = 4T(n/2) + n$	4	2	n	$\Theta(n^2)$
$T(n) = 7T(n/2) + n^2$	7	2	n^2	$\Theta(n^{2.807})$
$T(n) = 8T(n/2) + n^2$	8	2	n^2	$\Theta(n^3)$
$T(n) = T(n-1) + n$	not Master form			$\Theta(n^2)$
$T(n) = T(n-1) + T(n-2)$	not Master form			$\Theta(\varphi^n)$

B.7 Summation formulas

Identity	Where it is used
$\sum_{i=1}^n i = \frac{n(n+1)}{2}$	Triangular loops; sequential-search average; selection and bubble sort; number of contiguous subarrays
$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$	Triple triangular loops
$\sum_{i=0}^n 2^i = 2^{n+1} - 1$	Doubling loops; subset counts; nodes in a complete binary tree
$\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}$	General geometric series
$H_n = \sum_{i=1}^n \frac{1}{i} \approx \ln n + 0.577$	Harmonic loops; quick sort average case
$\binom{n}{k} = \frac{n!}{k!(n-k)!}$	Counting subsets of a fixed size
$\sum_{i=1}^n f_i = f_{n+2} - 1$	Cumulative Fibonacci totals

B.8 Growth-rate reference

n	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	3.3	10	33	100	1 024	3.6×10^6
20	4.3	20	86	400	1.0×10^6	2.4×10^{18}
50	5.6	50	282	2 500	1.1×10^{15}	3.0×10^{64}
100	6.6	100	664	10 000	1.3×10^{30}	9.3×10^{157}
1 000	10.0	1 000	9 966	10^6	—	—
10^6	19.9	10^6	2.0×10^7	10^{12}	—	—
10^9	29.9	10^9	3.0×10^{10}	10^{18}	—	—

B.9 Exam checklist

Question type	What to write
“Identify the basic operation”	Name the specific line: “the comparison $S[i] > \max$ on line 3”. Not “the loops”.
“Find the time complexity”	State the basic operation, count its executions as a function of n , show the summation, then apply the simplification rules. Show the arithmetic.
“Define big- O ”	Give the formal statement with $\exists c > 0, \exists N \geq 0, \forall n \geq N$. Do <i>not</i> answer “the worst case”.
“Prove $g \in O(f)$ ”	Exhibit one specific c and one specific N and verify the inequality. One valid pair is a complete proof.
“Prove $g \notin O(f)$ ”	Assume membership, divide both sides by the smaller function, observe that a variable cannot be bounded by a constant.
“Prove by induction”	Write the three headings <i>Base</i> , <i>Hypothesis</i> , <i>Step</i> explicitly — marks attach to the structure.
“Trace this recursive function”	Trace <i>table</i> for linear recursion (parameters down, returns up); trace <i>tree</i> for branching recursion. Label every node with its return value.
“Trace merge sort / quick sort”	Show every split and every merge (or partition), with the array state after each. Count comparisons in a separate column.
“Apply Kruskal / Prim”	Kruskal: list all edges in sorted order with ACCEPT/REJECT. Prim: list vertices <i>and</i> edges in selection order. Both: state the total weight.
“Fill in the DP table”	Show the recurrence first, then the table, then trace back to recover the actual solution — not just the optimal value.
“Draw the state space tree”	Mark pruned branches explicitly and say <i>why</i> each was pruned.
“Compare X and Y”	Use a table with named rows (complexity, space, stability, when to use). Prose comparisons lose structure marks.

Three habits worth forming

1. **State your conventions.** Fibonacci indexing, 0-based vs. 1-based arrays, whether n counts elements or bits — ambiguity in the question is resolved by you stating your reading, not by guessing the marker’s.
2. **Show the intermediate count before simplifying.** Writing $\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \in O(n^2)$ earns the working marks; writing $O(n^2)$ alone does not.
3. **Sanity-check numerical answers.** Merge sort comparisons should be just under $n \lg n$; MST weight should be identical whichever algorithm you used; a DP table’s last cell should be reachable by tracing back through legal moves.

Index

algorithm, 2
approximation ratio, 117
auxiliary space, 34
average-case time complexity, 29

base case, 16
best-case time complexity, 29
bounding, 130
branching, 130

certificate, 143
critical exponent, 56
cut, 112
cut property, 113

depth-first search, 121

every-case time complexity, 33

feasible solution, 131

Hamiltonian cycle, 125
Hamiltonian path, 125

in-place, 76
initial conditions, 53

lower mid, 12

Nondeterministic Polynomial time, 143
NP-hard, 144

optimal solution, 131

partition, 79
pivot, 79

recurrence relation, 53
reduce, 136

split point, 79
stable, 75
stack frame, 16
stack overflow, 16

tail call optimisation, 18
tight, 39

union-find, 111
upper mid, 12

worst-case time complexity, 29