CS231

Algorithmic Problem Solving

University of Waterloo · term 2026_05

Consolidated Course Notes

Everything you need to learn in this course,
organized one chapter per lecture module.

These notes synthesize all 11 lecture modules into exam-ready definitions, paradigm *sketches*, *checklists*, *recipes*, worked recurrences, and complexity results. They mirror the structure of the online course.

Material © University of Waterloo / CS231 course staff.
Personal study copy — do not redistribute.

Contents

The CS231 framework: one method, repeated **3**

1 Fundamentals **5**
1.1 Course goals and core definitions . . . 5
1.2 Specifying a problem . . . 5
1.3 Types of data . . . 6
1.4 Problem types . . . 6
1.5 Paradigms and exhaustive search . . . 6
1.6 Key Python ideas . . . 7

2 Order notation **8**
2.1 Running-time model . . . 8
2.2 Formal definitions . . . 8
2.3 Categories of growth and recipes . . . 8
2.4 Math review . . . 9

3 Algorithm analysis **10**
3.1 Pseudocode and the model of computation . . . 10
3.2 TSP by exhaustive search . . . 10
3.3 Counting facts for exhaustive search . . . 11

4 Greedy approach **12**
4.1 The greedy paradigm . . . 12
4.2 Logic background and proving (in)correctness . . . 12
4.3 Concrete algorithms (and when greedy fails) . . . 13

5 Divide-and-conquer **14**
5.1 The paradigm and checklist . . . 14
5.2 Recurrence relations . . . 14
5.3 Worked examples and TSP . . . 15

6 Dynamic programming **17**
6.1 The paradigm . . . 17
6.2 Optimal substructure . . . 17
6.3 Worked recurrences . . . 18

7 Hardness of problems **19**
7.1 Complexity, upper and lower bounds . . . 19
7.2 Lower bounds . . . 19
7.3 Reductions, NP, and completeness . . . 20

8 Compromising on time **22**
8.1 Backtracking . . . 22
8.2 Branch-and-bound . . . 23

9 Compromising on correctness **24**
9.1 Approximation algorithms and ratio bounds 24
9.2 Heuristics 25

10 Changing the rules **26**
10.1 Special instances 26
10.2 Randomized algorithms 26
10.3 Fixed-parameter tractability (FPT) 27
10.4 Online algorithms 27

11 More to explore **28**
11.1 Other models of computation 28
11.2 Classifying problems 28

A Complexity & growth cheat-sheet **29**

The CS231 framework: one method, repeated

CS231 is organized around a single recurring workflow. Almost every topic in the course is an instance of this five-stage pipeline:

1. **Specify the problem.** Turn a vague *case study* into a precise *problem* (input specification + output specification), made *general* and guaranteed to *make sense*. Classify it: optimization / decision / search / counting / enumeration.
2. **Choose a paradigm.** A systematic technique for designing an algorithm. Each paradigm ships with a *sketch* (general outline) and a *checklist* (the problem-specific blanks to fill in).
3. **Analyze.** Express worst-case running time as a function of input size and reduce it to order notation (O, Ω, Θ); for recursive algorithms, set up and solve a *recurrence*.
4. **Decide if it is good enough.** Establish lower bounds and *hardness* (P, NP, NP-hard, NP-complete) to learn whether a fast, correct algorithm is even possible.
5. **Compromise when it is not.** If a problem is NP-complete, trade away a guarantee: exact-but-slow (*time*), feasible-with-a-bound (*correctness*), or *change the rules* (special instances, randomization, parameters, online).

The golden rule of paradigms

Use the *specified* paradigm even when a cleverer shortcut exists — you are graded on applying the method, not on being smarter than it.

The paradigm toolbox at a glance

Paradigm	Mod.	Idea	Speed	Correctness story
Exhaustive search	1,3	Try every possibility	Slow ($n!$ or 2^n)	Trivially correct
Greedy	4	Grab the locally-best option, never undo	Fast	Needs optimal substructure + greedy choice; course proves only *incorrectness*
Divide-and-conquer	5	Split → recurse → combine	Medium	Via recurrences
Dynamic programming	6	Store solutions to smaller instances, build bottom-up	Often poly.	Optimal substructure + overlapping subproblems
Backtracking	8	Build partial solutions in a search tree, prune, undo	Exp. but pruned	Prune only subtrees that cannot contain a (better) solution
Branch-and-bound	8	Backtracking + a bounding function	Exp. but pruned	Same as backtracking
Approximation	9	Feasible solution with a provable ratio bound	Poly.	Within factor $R(n)$ of optimal
Heuristics	9	Local moves (hill climbing), no guarantee	Varies	None — may stick at a local optimum
Randomized / FPT / online	10	Change the rules of the game	Varies	Las Vegas (always correct) vs. Monte Carlo (likely correct)

How to use these notes. Each chapter follows the same rhythm as the course: the paradigm's *sketch* and *checklist*, the key *definitions* and *recipes*, worked results, and a boxed *Key takeaways* summary. The growth and complexity cheat-sheet in Appendix A collects every running-time result.

Chapter 1

Fundamentals

Translating real-life case studies into precisely specified problems, classifying problem and data types, and applying the exhaustive-search paradigm (with a Python review).

1.1 Course goals and core definitions

CS231 teaches you to use computers to solve problems. Two foundational terms:

Definition: algorithm & paradigm

An **algorithm** is a step-by-step procedure for solving a problem.
A **paradigm** is a systematic approach (a basic technique) for designing algorithms, applicable to many problems.

The course teaches you to translate real-life problems into computer-understandable ones, use paradigms to design algorithms, compare algorithms before coding, write Python code, recognize when a problem is unlikely to be solved in reasonable time, and find "good enough" solutions to such problems.

1.2 Specifying a problem

Definition: problem

A **problem** is a way of associating an input and an output. It includes an **input specification** (the types of data used) and an **output specification** (how the input relates to what is produced). The definition makes *no* claim about how (or whether) the problem can be solved. A good problem must be (1) as general as possible and (2) make sense — every valid input has a corresponding output.

Recipe: specifying a problem

- Make the problem general.
- Form the input specification using the most important data.
- Form the output specification using one or more easily measurable goals.
- Ensure there is output specified for any data satisfying the input specification.

Related terms. An **instance** is specific data satisfying the input specification. A **solution** to an instance satisfies the output specification; an algorithm *consumes* an instance and *produces* output, and *solves* the problem if it produces a solution for any instance ("a" solution, since several may exist).

Questions to consider when forming a problem.

- Can the input set or sequence be empty? If so, add "False" as a possible output if needed.
- Can there be more than one solution? If so, use "a" instead of "the".
- Can ordered items repeat? If so, use *nondecreasing*/*nonincreasing* instead of *increasing*/*decreasing*.

(A *set* has distinct items; a *sequence* may have duplicates and positions 0 to length − 1.)

1.3 Types of data

Input data types: numbers, strings, sets, sequences, grids, trees, graphs. **Output types** commonly include an ordering of items, a categorization, or a subset.

- **Grid:** a rectangular (2-D) arrangement; rows $0 \ldots (\#\text{rows} - 1)$, columns $0 \ldots (\#\text{columns} - 1)$.
- **Tree:** zero or more nodes; a nonempty rooted tree has a root; non-root nodes have a parent via an edge (child, siblings, ancestors, descendants, leaf, internal node, path of length = #edges = #nodes − 1, subtree). *Rooted, unordered unless stated otherwise.*
- **Graph: vertices** and **edges**, optionally with weights/colours. *Simple, undirected unless stated otherwise.*

Measuring instance size (loosely, the size in bits): number $\to n$; string $\to n$ (length); set $\to n$ (#elements); sequence $\to n$ (length); grid $\to r$ rows, c columns; tree $\to n$ nodes; graph $\to n$ vertices, m edges. Constant-size items are ignored when several types are present.

1.4 Problem types

One case study yields many problems, differing by goal/output:

- **Optimization:** find the *best* entity.
- **Decision:** output yes/no — add a bound B ("Is there a tour of weight $\leq B$?").
- **Search:** output an entity satisfying conditions ("or False if none exists").
- **Counting:** output the *number* of such entities.
- **Enumeration:** output *all* such entities.

Recipe: defining an optimization problem

(1) Define a *feasible solution*; (2) give a way to compute a numerical value for each feasible solution; (3) specify *maximization* or *minimization*. An **optimal solution** reaches the goal value. Sub-types: **constructive** (return an optimal solution) and **evaluation** (return only its *value*).

The running case study (trip planning) yields **TSP** (Travelling Salesperson Problem): input a complete weighted graph, output a *tour* (an ordering of all vertices forming a cycle) of minimum weight.

1.5 Paradigms and exhaustive search

A paradigm comes with a **sketch** (general outline) and a **checklist** (problem-specific definitions and processes). Not all paradigms work for all problems; one paradigm may yield several algorithms; results may be slow or incorrect.

Sketch for exhaustive search

- Generate all possibilities.
- Extract information from one possibility at a time.
- Determine the solution from the extracted information.

For optimization (e.g. TSP) all possibilities must be processed; for some decision problems you may stop early. Possibilities are commonly *orderings*, *categorizations*, or *subsets* of elements.

1.6 Key Python ideas

- **Functions:** `def`, indented body, `return`; contract in the docstring.
- **Importing/testing:** `import check` (e.g. `check.expect`); course modules via `from module import *` (`grids.py`, `trees.py`, `graphs.py`, `equiv.py`).
- **Lists:** non-mutating (`len`, indexing, `in`, `+`, `sorted`, slices) vs. mutating (`append`, `insert`, `pop`, `remove`, `sort`). `remove`/`sort` mutate and return `None`.
- **Copying:** `=` aliases; use `list(...)`, `copy.copy`, or `copy.deepcopy` to avoid mutating inputs.

Key takeaways

- A *problem* = input specification + output specification; it says nothing about solvability.
- Always make problems general and ensure every valid input has an output.
- Distinguish *instance*, *solution*, *feasible solution*, *optimal solution*.
- One case study ⇒ many problem types (optimization, decision, search, counting, enumeration).
- *Exhaustive search* = try all possibilities — universally applicable, easy to prove correct, but often factorial/exponential.

Chapter 2

Order notation

A math-grounded development of asymptotic notation (O, Ω, Θ) for categorizing running times, plus the running-time model.

2.1 Running-time model

An algorithm's running time is a function $f(n)$ of instance size n (multi-variable, e.g. $f(m, n)$, when sized by several quantities). For size k, three representatives reduce all instances to one value:

- **Average case:** $f(k) = \sum_{I \in \mathcal{I}} \Pr[I] \cdot \text{time}(I)$, over all instances $\mathcal{I}$ of size k.
- **Best case:** the smallest running time on any instance of size k.
- **Worst case:** the largest running time on any instance of size k.

Crucial warning

Any notation (O, Ω, Θ) can classify *any* case (best/worst/average). Do **not** equate worst case with O or best case with Ω.

2.2 Formal definitions

Order notation uses a real constant $c > 0$ and an integer threshold $n_0 \geq 1$ (which ignores small-n behaviour); constants may be less than 1.

Definition: O, Ω, Θ

- $f(n) \in O(g(n))$ if $\exists\, c > 0,\ n_0 \geq 1$ with $f(n) \leq c\, g(n)$ for all $n \geq n_0$.
- $f(n) \in \Omega(g(n))$ if $\exists\, c > 0,\ n_0 \geq 1$ with $f(n) \geq c\, g(n)$ for all $n \geq n_0$.
- $f(n) \in \Theta(g(n))$ if $\exists\, c_1, c_2 > 0,\ n_0 \geq 1$ with $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

Thus Θ means both O and Ω: $f \in \Theta(g) \iff f \in O(g)$ and $f \in \Omega(g)$. For a Θ proof, show O and Ω separately — choose c above the dominant coefficient for O, below it for Ω (e.g. $3n^2 - 5n + 6 \in \Theta(n^2)$).

2.3 Categories of growth and recipes

Functions group roughly by growth: **constant** $(1,\ 201)$, **logarithmic** $(\log_2 n)$, **linear** $(n, 5n)$, **quadratic** (n^2), **exponential** (2^n); others $(n \log n,\ 1/n)$ need no name. The **dominant** term grows largest in n. A **simple function** (course term) has no multiplicative constants or additive terms.

Recipe: categorizing with Θ (one variable)

Use a simple function: remove all multiplicative constants and keep only the dominant term (constants give $\Theta(1)$). **Multiple variables:** keep *all* possibly-dominant terms, since relative

variable sizes may be unknown.

Comparison. For simple g smaller than h: $f \in O(g) \Rightarrow f \in O(h)$ and $f \in \Omega(h) \Rightarrow f \in \Omega(g)$.
Combining (sums/products). Keep dominant term(s); the result is Θ only if *all* chosen categories are Θ, else O (smallest simple function). Products multiply categories first.
Relative sizes. If $m \in \Theta(g(n))$, substitute $g(n)$ for m; $m \in O(g(n))$ may let you drop m-terms; $m \in \Omega(g(n))$ may let you drop terms m dominates. Variables stay variables (never fix a value).

2.4 Math review

- **Floors/ceilings:** $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$.
- **Logs/exponents** ($\log = \log_2$): $n^a n^b = n^{a+b}$, $(n^a)^b = n^{ab}$, $\log_a x = \frac{\log_b x}{\log_b a}$, $\log xy = \log x + \log y$, $\log x^k = k \log x$; $\log^2 n = (\log n)^2$.
- **Summations:** arithmetic $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$; geometric $\sum_{i=1}^{n} c^{i-1} = \frac{1-c^n}{1-c}$.
- **Counting:** permutations $n!$; assignments to k categories k^n; subsets 2^n; combinations $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!}$.

Key takeaways

- Running time is a function $f(n)$; best/worst/average are just different representatives.
- O, Ω, Θ are defined via c (and c_1, c_2) and n_0; Θ = both O and Ω.
- Any notation can describe any case — they are *not* paired.
- Categorize by dropping constants and keeping dominant term(s); Θ only if all inputs are Θ.

Chapter 3

Algorithm analysis

Describing algorithms in pseudocode and analyzing worst-case running time with order notation, applied to exhaustive search and TSP.

3.1 Pseudocode and the model of computation

The goal is to compare algorithms' running times *without* coding them. A **model of computation** defines operations and costs; the **Random Access Machine (RAM)** is one. In practice the course uses **pseudocode**: simple operations (assignment, simple arithmetic/Boolean ops, moving to another line, returning a value) each cost $\Theta(1)$.

Conventions (for pseudocode you *read*): italics for pseudocode; function names in CAPITALS; variables Capitalized; assignment $\leftarrow$; functions written in full (*APPEND*(L,4), *LENGTH*(L)); *for each ... in* for sequence loops; simple list ops allowed, powerful ones (`sorted`, `reverse`) not.

Recipe: worst-case running time

- Break each block into blocks.
- Determine a bound on each block individually.
- Retain all dominant costs.
- Use Θ if all costs are in Θ *and* can occur simultaneously; use O otherwise.

Supporting rules. A line of constantly many constant-time steps is $\Theta(1)$. *Never* assume list/dict operations (`sort`, `map`, `filter`...) are constant-time. **Branching:** cost of the executed branch *plus* the conditions evaluated to reach it. **Looping:** sum over iterations of (iteration management $\Theta(1)$ + body).

- **Common situation 1:** every iteration costs $\Theta(g(n)) \Rightarrow$ loop = (#iterations)$\times\Theta(g(n))$.
- **Common situation 2:** iteration i costs $\Theta(i) \Rightarrow$ loop = $\Theta(k^2)$ for k iterations (since $\sum i = \frac{n(n+1)}{2}$).

3.2 TSP by exhaustive search

TSP is *constructive optimization*: return the minimum-weight tour. Checklist: possibilities = all tours (all vertex permutations, *ORDERINGS*); information = cost of one tour (*TOURCOST*); solution = minimum-so-far. Both *ORDERINGS* and the number of orderings are $\Theta(n!)$, and *TOURCOST* is $\Theta(n^2)$ on a complete graph, so the total is

$$\boxed{\Theta\left(n^2\, n!\right)} \quad = \quad (\#\text{possibilities}) \times (\text{cost per possibility}).$$

This factorial blow-up is why TSP is expensive; correctness is trivial (try all, keep the best).

3.3 Counting facts for exhaustive search

- $2^n < n! < n^n$, i.e. $n! \in O(n^n)$ and $n! \in \Omega(2^n)$.
- $\log n! \in \Omega(n \log n)$.
- $\binom{n}{k} \in \Theta(n^k)$.

Example: **3-COLOURING** has 3^n possibilities (3 colour choices per vertex), each checked so adjacent vertices differ.

Key takeaways

- Pseudocode lets you analyze before coding; simple ops are $\Theta(1)$, list/dict ops are not.
- Apply the four-step recipe and the two common loop situations.
- Exhaustive-search cost = (#possibilities)×(cost per possibility) — often factorial/exponential.
- Exhaustive-search TSP is $\Theta(n^2 n!)$; remember $2^n < n! < n^n$ and $\binom{n}{k} \in \Theta(n^k)$.

Chapter 4

Greedy approach

Build a solution step by step by grabbing whatever looks best now — plus the logic to prove (or disprove) that such choices are correct.

4.1 The greedy paradigm

A **greedy algorithm** builds a solution step by step, often using an ordering to decide, and *never undoes* a decision.

Sketch for greedy

Process data; loop to build the solution step by step (use information to make a decision; make updates to information).

Checklist for greedy

- Process for preprocessing data.
- Definition of the steps needed to build a solution.
- Definition of the information used for a decision.
- Definition of the criteria for a decision.
- Process for making updates to information.

Decisions choose a min/max value or are **arbitrary** (any one, by an unspecified method — distinct from *random*). Ties break arbitrarily unless stated; an algorithm that breaks ties arbitrarily is correct *only if it is correct for every way of breaking ties*. Greedy TSP runs in $\Theta(n^2)$ but a counterexample shows it can be non-optimal.

4.2 Logic background and proving (in)correctness

Tools: OR/AND/NOT; predicates $P(x)$; **existential** / **universal** statements (negation swaps the two and exchanges AND/OR); implication, **converse**, **contrapositive** (equivalent to the original); *modus ponens*; proof by contradiction; universal via a **generic element**; existential by **exhibiting** a witness.

Recipe: proving an algorithm *not* correct (a counterexample)

Exhibit one instance x: (1) show x is a valid instance; (2) show the algorithm outputs y; (3) show a correct solution z is better than y.

Definition: when greedy is correct

A greedy algorithm is correct only if both hold:

- **Optimal Substructure Property:** an optimal solution can be formed from optimal solutions to smaller instances.
- **Greedy Choice Property:** there is an optimal solution consistent with each greedy choice made.

This course *omits* formal proofs of the Greedy Choice Property and defers Optimal Substructure to a later module; the **focus is counterexamples** of incorrectness, which need only a single instance.

4.3 Concrete algorithms (and when greedy fails)

- **Scheduling Activities:** maximize non-overlapping activities. Shortest-first is *incorrect*; the correct criterion is **earliest finish time**.
- **Making Change:** largest-denomination-first is *incorrect* — for $C = \{1, 7, 10\}$, $x = 14$, greedy gives $10 + 1 + 1 + 1 + 1$ (five coins) vs. optimal $7 + 7$.
- **Fractional Knapsack:** maximize $\sum x_i v_i$ s.t. $\sum x_i w_i \leq W$, $0 \leq x_i \leq 1$. Correct criterion: **non-increasing value-to-weight ratio**, $\Theta(n \log n)$ (sorting dominates).
- **Dijkstra** (single-source cheapest paths): keep K (known), U (unknown), *special*(v); repeatedly move the cheapest special vertex into K and update. $O(m + n \log n)$; also works on directed graphs.
- **Minimum Spanning Tree: Kruskal** ($O(m \log m)$, add cheapest cycle-free edges) and **Prim** ($O(m \log n)$, grow a tree by cheapest connecting edge).

Versus exhaustive search: greedy explores *some* feasible solutions, has *narrow* applicability, but is *fast*.

Key takeaways

- The five-item checklist (preprocess, steps, information, criteria, updates) designs any greedy algorithm.
- Arbitrary tie-breaking demands correctness under *every* tie-break.
- Correctness needs Optimal Substructure + Greedy Choice; the course proves only *incorrectness* via one counterexample (x, y, z).
- Greedy is correct for Fractional Knapsack, Selection Sort, Dijkstra, and MST — but fails for Making Change and shortest-activity scheduling.

Chapter 5

Divide-and-conquer

Break an instance into smaller instances of the same problem, solve those recursively, and combine — with running time analyzed via recurrences.

5.1 The paradigm and checklist

Definition: divide-and-conquer

Divide	the instance into one or more smaller instances of the same problem.
Conquer	the smaller instance(s) via recursive calls.
Combine	the sub-solutions into a solution to the original instance.

It relies on **recursive definitions** (≥ 1 recursive case $+ \geq 1$ base case covering all sizes).

Checklist for divide-and-conquer

(1) Definition of smaller instances; (2) definition of base cases; (3) process for dividing; (4) process for combining results.

Divide/combine trade-off

A cheap combine forces a costly divide and vice versa. Splitting by *position* (cheap divide) with a *merge* (costly combine) gives **Mergesort**; **Binary search** compares the splitting element to the target and returns the relevant sub-instance's solution.

5.2 Recurrence relations

Definition: recurrence

A **recurrence** expresses worst-case time $T(n)$: base cases for small n; for larger n, a recursive case combining divide + conquer ($T(k)$, $k < n$) + combine. Recurrence base cases depend only on *size* n (cases too small for the recursive case) — unlike recursion base cases (solved without recursive calls).

Binary search (with slices): $T(n) \leq T(\lfloor n/2 \rfloor) + \Theta(n)$; in-place: $T(n) \leq T(\lfloor n/2 \rfloor) + \Theta(1)$.

Method 1 — Iteration (recursion tree)

Recipe: iteration method

Apply the definition of $T(n)$ repeatedly; express it after k iterations; pick k that reduces the smaller term to the base case; write $T(n)$ in closed form.

E.g. $T(n) = T(n-1) + 5,\ T(1) = 4 \Rightarrow T(n) = 5n - 1$; $T(n) = T(\lceil n/2 \rceil) + 1,\ T(1) = 3 \Rightarrow \Theta(\log n)$.

Method 2 — Master method

Recipe: master method

For $T(n) = aT(n/b) + f(n)$ $(a \geq 1,\ b > 1)$: compute $x = n^{\log_b a}$; compare $f(n)$ to x:

- f *smaller* than $x \Rightarrow T(n) \in \Theta(x)$;
- f *bigger* than $x \Rightarrow T(n) \in \Theta(f(n))$;
- f *same size* as $x \Rightarrow T(n) \in \Theta(x \log n)$.

$4T(n/2) + n \Rightarrow x = n^2 \Rightarrow \Theta(n^2)$; $T(n/2) + n \Rightarrow x = 1 \Rightarrow \Theta(n)$; $2T(n/2) + 4n \Rightarrow x = n \Rightarrow \Theta(n \log n)$.

Definition: Master Theorem (exact)

For $T(n) = aT(n/b) + f(n)$, $a \geq 1,\ b > 1$:

- if $f(n) \in O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$;
- if $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $af(n/b) \leq cf(n)$ for some $c < 1$ and large n, then $T(n) \in \Theta(f(n))$;
- if $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \log n)$.

Method 3 — Substitution (induction)

Recipe: substitution method

Guess an upper bound for $T(n)$; substitute the guess for T on smaller values; simplify the right side to prove the bound; check base cases last. Use a placeholder constant c; if none works, subtract a lower-order term (e.g. $cn^2 - dn$).

$T(n) = 2T(\lfloor n/2 \rfloor) + 1 \Rightarrow T(n) \leq 6n - 1 \in O(n)$; $2T(\lfloor n/2 \rfloor) + 21 \Rightarrow O(n \log n)$.

5.3 Worked examples and TSP

- **MAXIMUM SUBTOTAL:** combine = max(left, right, max suffix of left+max prefix of right); $T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n \log n)$.
- **SELECTION** (jth largest, median-of-medians): $T(n) \leq T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + O(n)$ — *not* solvable by the Master method; substitution gives $\Theta(n)$.
- **Naive matrix multiply** is $\Theta(n^3)$ (Strassen improves it); Karatsuba speeds up integer PRODUCT.

On TSP: divide-and-conquer gets stuck at the divide step (removing an edge/vertex breaks completeness and recombining tours is unclear). It suits decomposable structures (paths, trees: node height = 1 + max subtree height) but generally not graph problems. Overall: *medium* applicability, *medium* speed.

Key takeaways

- Divide → conquer → combine, driven by a 4-item checklist; balance divide vs. combine cost.
- Running time is a recurrence; recurrence base cases depend only on size n.
- Three solving methods: iteration, Master method (compare f to $x = n^{\log_b a}$), substitution (guess + induction).
- The Master method is fast but does not apply to all recurrences (e.g. SELECTION).

Chapter 6

Dynamic programming

Build up stored solutions to smaller instances, bottom-up — the right tool when a problem has the Optimal Substructure Property.

6.1 The paradigm

Dynamic programming (DP) solves smaller instances, stores them in a table, and looks them up to solve larger instances. ("Programming", coined by Bellman in the 1950s, has nothing to do with computers.) DP works **bottom-up**; the course does not credit top-down memoization.

Sketch for dynamic programming

```
Create a table
Loop over table entries
    Fill in base cases
    Fill in non-base cases
Extract solution
```

Checklist for dynamic programming

1. Definition of the solution in terms of solutions to smaller instances (the recurrence — the hardest step).
2. Definition of the information stored in each table entry.
3. Definition of base cases and their values.
4. Definition of the shape of the table(s).
5. Definition of the order of evaluation.
6. Process for extracting the solution from the table.

The key to *order of evaluation* is that every entry an entry depends on is already filled. Base cases need not be real instances (empty sequences, negative indices); triangular tables arise when $i < j$ or $M[i, j] = M[j, i]$.

6.2 Optimal substructure

DP applies when subproblems **overlap** (naive divide-and-conquer recomputes pieces) and the problem has the **Optimal Substructure Property**.

Recipe: proving optimal substructure (contradiction, generic instance)

1. From the optimal solution O for an arbitrary instance I, construct smaller instances.
2. Decompose O into pieces, one per smaller instance.

3. Show that if any piece O' is not optimal for its smaller instance I', then O is not optimal for I.

Analysis ≈ (table size) × (cost per entry); filling the table usually dominates.

6.3 Worked recurrences

Matrix-chain multiplication (M_i is $d_i \times d_{i+1}$):

$$m[i,j] = \min_{i \le k < j} \{\, m[i,k] + m[k+1,j] + d_i\, d_{k+1}\, d_{j+1} \,\}, \quad m[i,i] = 0,$$

upper-triangular table filled diagonal-by-diagonal; $O(n^3)$.

Longest common subsequence (X length m, Y length n):

$$L[i,j] = \begin{cases} L[i-1,j-1]+1 & x_{i-1} = y_{j-1} \\ \max\{L[i-1,j],\ L[i,j-1]\} & \text{otherwise,} \end{cases}$$

$L[0,j] = L[i,0] = 0$; $(m{+}1) \times (n{+}1)$ table; $\Theta(mn)$.

All-pairs cheapest paths (Floyd–Warshall) — $\text{cost}(i,j,k)$ uses intermediates of index $< k$:

$$\text{cost}(i,j,k) = \min\{\, \text{cost}(i,j,k{-}1),\ \text{cost}(i,k{-}1,k{-}1) + \text{cost}(k{-}1,j,k{-}1) \,\},$$

base $k = 0$: edge weight (∞ if absent); $\Theta(n^3)$.

0/1 Knapsack — $C[i,j]$ is the greatest value for weight bound j using objects up to i:

$$C[i,j] = \max\{\, C[i-1,j],\ C[i-1,\ j-w_i] + v_i \,\},$$

$n \times (W{+}1)$ table; answer $C[n,W]$; $\Theta(nW)$ (pseudo-polynomial).

Trees/graphs: tree height = 1 + max child height (leaves are base cases, height 0); evaluate leaves-to-root. DP suits special graphs like trees because general graphs (TSP) recombine poorly.

Key takeaways

- DP needs *optimal substructure*; build bottom-up, store smaller solutions, look them up.
- The 6-item checklist is the core method; the recurrence is the hardest step.
- Running time ≈ (table size) × (cost per entry).
- Recurrences: matrix-chain $O(n^3)$, LCS $\Theta(mn)$, Floyd–Warshall $\Theta(n^3)$, knapsack $\Theta(nW)$, tree height linear. Store traceback info to reconstruct solutions.

Chapter 7

Hardness of problems

Prove a problem is "as good as it can get": lower bounds, and classifying problems as tractable (P) or hard (NP-hard / NP-complete) using reductions.

7.1 Complexity, upper and lower bounds

The **complexity** of a problem is the worst-case cost of its *best* algorithm. Complexity $\Theta(f(n))$ needs both an **upper bound** $O(f(n))$ (some algorithm achieves it) and a **lower bound** $\Omega(g(n))$ (every algorithm needs it). A bound on an *algorithm* is not a bound on the *problem*.

Definition: P and tractability

An algorithm runs in **polynomial time** if its time is $O(n^c)$ for a constant c. The class **P** is the decision problems solvable in polynomial time. A problem is **tractable** if known to be in P, **intractable** otherwise. Polynomial time is chosen because the class is robust (closed under sums, products, composition).

7.2 Lower bounds

Decision trees (information theory). For **comparison-based** algorithms $(=, \neq, <, >, \leq, \geq)$, a decision tree has comparisons at internal nodes and outputs at leaves; worst-case comparisons = height. A binary tree of height h has $\leq 2^h$ leaves, so:

Information-theory lower bound

If a problem has ℓ possible outputs, any comparison-based algorithm needs $\Omega(\log \ell)$ time in the worst case. Searching n values: $\Omega(\log n)$; sorting ($n!$ outputs): $\Omega(\log n!) = \Omega(n \log n)$.

Adversary arguments ("twenty questions"). The algorithm asks questions to shrink the set of consistent instances; the adversary answers to keep it large.

Recipe: adversary lower bound

1. Specify an adversary strategy.
2. Determine a number of steps T any correct algorithm must take.
3. Show that after $T - 1$ steps, ≥ 2 consistent instances remain that yield *different* outputs — so the adversary can force an error.

SORTING needs $\geq \lfloor \log n! \rfloor$ comparisons; MINIMUM needs $\geq \lceil n/2 \rceil$ (or $n - 1$) comparisons.

7.3 Reductions, NP, and completeness

Definition: reduction

Problem A **reduces** to problem B if we can solve A using an algorithm for B at most polynomially many times plus polynomial extra time ("A is no harder than B"). Consequences: *Easy* $B \Rightarrow$ *Easy* A (if $B \in \mathsf{P}$ then $A \in \mathsf{P}$); *Hard* $A \Rightarrow$ *Hard* B. If both directions reduce, the problems are **equivalent**.

Definition: NP and verification

A **polynomial-time verification algorithm** takes an instance plus a polynomial-size **certificate** and outputs "Yes" iff it is a yes-instance with a valid certificate. **NP** is the class of decision problems verifiable in polynomial time with a polynomial-size certificate ("NP" = *nondeterministic* polynomial).

Recipe: membership in NP

(1) Give a polynomial-size certificate per yes-instance; (2) give a poly-time verifier; (3) show it says "Yes" for any yes-instance + certificate; (4) show no false certificate fools it on a no-instance.

Definition: NP-hard / NP-complete

$\mathsf{P} \subseteq \mathsf{NP}$ (proven); whether $\mathsf{P} = \mathsf{NP}$ is open. Y is **NP-hard** if every $X \in \mathsf{NP}$ reduces to Y; Y is **NP-complete** if $Y \in \mathsf{NP}$ *and* Y is NP-hard. If any NP-complete problem is in P, then $\mathsf{P} = \mathsf{NP}$.

Recipe: proving NP-completeness of Z

1. Prove $Z \in \mathsf{NP}$.
2. Select a known NP-complete Y.
3. Give a poly-time f mapping each instance of Y to an instance of Z.
4. Prove: x a yes-instance for $Y \Rightarrow f(x)$ a yes-instance for Z.
5. Prove: $f(x)$ a yes-instance for $Z \Rightarrow x$ a yes-instance for Y.
6. Prove f runs in polynomial time.

Pitfall: reduce *from* known-hard Y *to* Z — never reverse it.

Examples. VERTEX COVER (NP-complete via INDEPENDENT SET): $(G, k) \mapsto (G, n - k)$. **TSP DECISION** (via HAMILTONIAN CYCLE): original edges weight 0, added edges weight 1, target 0.

Key takeaways

- Distinguish bounds on *algorithms* from bounds on *problems*.
- Decision trees give $\Omega(\log \ell)$ ($\Omega(n \log n)$ for sorting); adversary arguments are more general.
- Reduction $A \to B$: *Easy* $B \Rightarrow$ *Easy* A, *Hard* $A \Rightarrow$ *Hard* B.
- P= poly-time solvable; NP= poly-time verifiable; $\mathsf{P} \subseteq \mathsf{NP}$.
- NP-hard = everything in NPreduces to it; NP-complete = NP-hard $\cap$ NP. Reduce *from* a

known-hard problem in the correct direction.

Chapter 8

Compromising on time

Two exact paradigms — backtracking and branch-and-bound — that always produce a correct answer to NP-complete problems even though running time may be exponential.

8.1 Backtracking

Because NP-complete problems likely admit no poly-time algorithm, we compromise on time (this chapter) or correctness (next). Driving example: **LONG PATH** (does G have a path of length $\geq k$?). We want to generate *only useful* possibilities while *never missing* a correct solution.

Definition: partial solutions & candidates

A **partial solution** corresponds to all solutions consistent with it (empty path → all paths; "a" → all paths starting at a). It is a **candidate** if it has the right form to be a solution. Extending divides a partial solution's candidate set into subsets; correctness requires the union of an extension's candidate sets to equal the parent's (no candidate lost).

The search forms an implicit **search tree**, explored depth-first; at a base case (leaf) the search **backtracks** to the nearest incompletely-explored node. Unlike greedy, decisions can be undone.

Sketch for backtracking

Start with the initial partial solution at the root; initialize extra information; at the current node — stop if it is a candidate; stop if it cannot be extended; recursively process each child in order; update the extra information; update the output; backtrack.

Checklist for backtracking

Definition of: a partial solution; the partial solution at the root; the extra information; the process for the root; the process for a non-root; determining a candidate; determining it cannot be extended; extending a partial solution; updating the extra information; determining the output. (The root is a non-recursive wrapper; non-roots recurse.)

All four problem types share the skeleton: *decision* (True/False); *search* (first candidate, else False); *enumeration* (set of all candidates); *optimization* (keep the **best-so-far**; a candidate is one longer than best-so-far).

Reducing nodes explored (time $\leq$ nodes $\times$ cost/node): **fix an ordering** so each candidate is generated at most once (but still at least once); **collapse equivalent solutions**; **store richer state**.

8.2 Branch-and-bound

Definition : branch-and-bound

Adds a **bounding function** to backtracking: for *maximization* an *upper* bound on any candidate extending the current partial solution; for *minimization* a *lower* bound. If the bound is worse than the best-so-far, **backtrack immediately**, pruning the whole subtree. (For INTEGER KNAPSACK the bound is current value plus $v_k/w_k \times$ remaining weight — a fractional relaxation.)

Key takeaways

- **Backtracking** builds useful partial solutions in an implicit, recursively-explored search tree, undoing decisions on backtrack — correct but possibly exponential.
- The checklist is the reusable recipe; one skeleton serves decision, search, enumeration, optimization.
- Pruning, best-so-far, and fixed orderings cut nodes; orderings must remove duplicates without dropping any candidate.
- **Branch-and-bound** adds a bounding function (upper for max, lower for min) to prune subtrees that cannot beat the best-so-far.

Chapter 9

Compromising on correctness

Trade correctness for speed: approximation algorithms (with provable guarantees) or heuristics (with none).

9.1 Approximation algorithms and ratio bounds

Definition: approximation algorithm

An algorithm for an optimization problem guaranteeing that (1) the solution is *feasible* and (2) there is a *provable bound* on how close it is to optimal. We care more about quality than speed.

Let A = approximate value, O = optimal value. For minimization focus on A/O; for maximization on O/A.

Definition: ratio bound

An approximation algorithm has **ratio bound** $R(n)$ if for any instance of size n, $\max\{\frac{A}{O}, \frac{O}{A}\} \leq R(n)$. If $R(n)$ is constant it is a **constant ratio bound**; a problem with a poly-time constant-ratio approximation is in **APX**.

To disprove a ratio bound: one counterexample with $\max\{\frac{A}{O}, \frac{O}{A}\} > R(n)$ (or a *family* for an order-notation bound). The greedy max-degree vertex cover has *no* constant ratio bound (a family gives $\Theta(\log k)$).

Recipe: analyzing an approximation algorithm

(1) Choose a new measure M; (2) bound A in terms of M; (3) bound O in terms of M; (4) combine so M cancels.

- **Vertex cover (2-approx):** repeatedly pick any uncovered edge, add *both* endpoints, remove incident edges. With $|E|$ chosen edges: $A = 2|E|$ and $O \geq |E|$, so $A/O \leq 2$.
- **Metric/Euclidean TSP (2-approx):** MST $\to$ full walk (each edge twice) $\to$ shortcut. With M = MST cost: $A \leq 2M$ and $O \geq M$, so $A/O \leq 2$.
- **Job scheduling (makespan):** assign each job to the least-loaded machine so far.

Inapproximability. An *inapproximability result* shows that a certain approximation existing would imply $\mathsf{P} = \mathsf{NP}$. Theorem: *if general TSP is in APX, then* $P = NP$ (via HAMILTONIAN CYCLE).

9.2 Heuristics

Definition: heuristic

A **heuristic** has *no guarantees* — possibly non-optimal, possibly slow (the key difference from approximation algorithms). **Hill climbing**: start with an arbitrary feasible solution, repeatedly make a small improving change, stop at a local optimum. **Steepest ascent** picks the single best-improving change. Both can get stuck at a local optimum; escaping generally requires occasionally moving to a worse solution.

Key takeaways

- Approximation guarantees feasibility *and* $\max\{\frac{A}{O}, \frac{O}{A}\} \leq R(n)$ (A/O for min, O/A for max).
- Prove a bound via a third measure M (vertex cover and metric TSP both give ratio 2); disprove with a counterexample or family.
- General TSP is inapproximable unless $\mathsf{P} = \mathsf{NP}$.
- Heuristics like hill climbing give *no* guarantees and can be trapped at local optima.

Chapter 10

Changing the rules

Relaxing four standing assumptions — no extra instance information, deterministic algorithms, compromising on NP-complete problems, and full input knowledge — can make computation easier (or harder).

10.1 Special instances

A NP-complete problem may still have a poly-time algorithm on special instances.

Definition: pseudo-polynomial; weak vs. strong NP-completeness

An algorithm polynomial in the *largest integer* of the input runs in **pseudo-polynomial time**. Because instance size measures representation *length*, not numeric value, KNAPSACK's $O(nW)$ DP need not be polynomial in n. A NP-complete problem is **weakly NP-complete** if a pseudo-polynomial algorithm exists, **strongly** otherwise.

Special parameters / graph classes: 2-COLOURING is polynomial though 3-COLOURING is NP-complete; VERTEX COVER with $k = 1$ is $\Theta(mn)$; trees, bounded-treewidth, and planar graphs often make NP-complete graph problems tractable.

10.2 Randomized algorithms

Definition: randomized algorithm; expected time

A **randomized algorithm** makes some steps randomly, so results/times may vary across runs. **Expected running time** averages over executions *on a single instance* (no distribution on instances). An expected value is $\sum_{\text{events}}(\text{value}) \cdot (\text{probability})$.

- **Las Vegas:** *always correct*, expected poly time. (Random-pivot SELECTION/SORTING: a pivot is "good" if each sub-instance is $\leq 3/4$ the size, probability 1/2.)
- **Monte Carlo:** *always poly time*, correct with high probability. Decision problems have *false positives* / *false negatives*; **one-sided error** has only one kind (e.g. PRIMALITY: no false negatives). **Boosting:** one-sided per-iteration error $p \Rightarrow$ error p^k after k iterations; two-sided error reduced by majority vote. Las Vegas $\rightarrow$ Monte Carlo is easy; there is *no known* general Monte Carlo $\rightarrow$ Las Vegas conversion.

10.3 Fixed-parameter tractability (FPT)

Definition: fixed-parameter tractable

A **parameterized problem** designates a parameter p to "take the blame" for hardness. An algorithm is **fixed-parameter tractable** if it runs in $O(g(p)\,n^{O(1)})$, where g depends on p only (not n).

Bounded search tree: a tree of height h with $\leq c$ children per node has $O(c^h)$ nodes. VERTEX COVER by cover size k: branch on the two endpoints of an uncovered edge, height $\leq k$, giving $O(2^k n^{O(1)})$ — FPT.

Definition: kernelization

Kernelization replaces an instance by a smaller **kernel** of the same problem so that (1) the kernel is a yes-instance iff the original is, and (2) the kernel size is bounded by a function of p. Solving = forming the kernel then brute-forcing it (still FPT). For VERTEX COVER by k: any vertex of degree $> k$ must be in the cover.

Classes: FPT ("easy") $\subseteq$ W[1] $\subseteq$ W[2] $\subseteq \cdots$; hardness uses **parameterized reductions** (parameter of B a function of that of A, built in FPT time), *not* polynomial-time reductions.

10.4 Online algorithms

Definition: online algorithm; competitive ratio

An **offline** algorithm has all input before computing; an **online** algorithm must decide before all input is known (e.g. rent-or-buy). It has **competitive ratio** c if its cost is at most c times the best offline algorithm's cost on any input.

A lower bound ℓ is proved by a specific instance with (online)/(best offline) $\geq \ell$. Randomization helps against an *oblivious adversary* (choices fixed before the run).

Key takeaways

- Restricting inputs (integer sizes, parameter values, graph classes) can make NP-complete problems tractable; pseudo-polynomial $\Rightarrow$ weakly NP-complete.
- **Las Vegas:** always correct, expected poly time. **Monte Carlo:** always poly time, likely correct; repetition drives error toward p^k.
- **FPT** confines hardness to a parameter: $O(g(p)\,n^{O(1)})$ — via bounded search trees or kernelization; hardness via parameterized reductions.
- **Online** algorithms decide without the future; quality = competitive ratio vs. the best offline algorithm.

Chapter 11

More to explore

A survey broadening the course's lens beyond the RAM model and worst-case complexity.

11.1 Other models of computation

- **Turing machine** — infinite tape + finite states; rests on the **Church–Turing thesis**. Variants: alternating, probabilistic, oracle.
- **Circuit models** — Boolean circuits (OR/AND/NOT gates), measured by *size* and *depth*; generalized to arithmetic circuits.
- **Parallel computation** — synchronous processes sharing memory (minimum of distinct numbers in constant time with one processor per comparison).
- **Distributed computation** — asynchronous message-passing; *consensus* is typically impossible deterministically but tractable with randomization.
- **Quantum computation** — qubits/registers; **Grover's algorithm** searches n possibilities in $O(\sqrt{n})$ with high probability.

11.2 Classifying problems

Beyond input-size / worst-case / P-vs-NP:

- **Measures** — output-sensitive analysis, amortized analysis.
- **Polynomial-time hierarchy** — from P/NP/co-NPvia alternating Turing machines; collapses to Pif P = NP. **PSPACE** contains the whole hierarchy.
- **Approximation** — relative error $\frac{|A-O|}{O} \leq \epsilon(n)$; classes FPTAS $\subseteq$ PTAS $\subseteq$ APX.
- **Randomized** — ZPP $\subseteq$ RP $\subseteq$ BPP; P $\subseteq$ ZPP $\subseteq$ RP $\subseteq$ NP.
- **Parallel/space** — NC $\subseteq$ L $\subseteq$ NL $\subseteq$ P.
- **Computability** — some problems (the **Halting Problem**) cannot be solved at all.

Key takeaways

- The RAM/worst-case framework is one choice among many; alternative models reframe design and cost.
- Knowing more about your data enables specialized techniques and measures.
- Complexity is a rich web of classes whose separations largely hinge on the open P = NP question.

Appendix A

Complexity & growth cheat-sheet

Growth ordering (slowest → fastest)

$$1 \prec \log n \prec n \prec n \log n \prec n^2 \prec n^3 \prec \cdots \prec 2^n \prec n! \prec n^n$$

with the course facts $2^n < n! < n^n$, $\log n! \in \Omega(n \log n)$, and $\binom{n}{k} \in \Theta(n^k)$.

Algorithm running times from the course

Problem / algorithm	Module	Running time
Exhaustive-search TSP	3	$\Theta(n^2 \cdot n!)$
3-colouring (exhaustive)	3	$\Theta(3^n \cdot \text{check})$
Greedy TSP	4	$\Theta(n^2)$
Fractional knapsack (greedy)	4	$\Theta(n \log n)$
Dijkstra (cheapest paths)	4	$O(m + n \log n)$
MST — Kruskal / Prim	4	$O(m \log m)$ / $O(m \log n)$
Mergesort, max-subtotal	5	$\Theta(n \log n)$
Selection (median-of-medians)	5	$\Theta(n)$
Matrix multiply (naive)	5	$\Theta(n^3)$
Matrix-chain (DP)	6	$O(n^3)$
LCS (DP)	6	$\Theta(mn)$
Floyd–Warshall (DP)	6	$\Theta(n^3)$
0/1 knapsack (DP)	6	$\Theta(nW)$ (pseudo-poly)
Vertex cover (FPT, bounded tree)	10	$O(2^k\, n^{O(1)})$

Master Theorem (quick form)

For $T(n) = aT(n/b) + f(n)$ with $x = n^{\log_b a}$: $f \prec x \Rightarrow \Theta(x)$; $f \succ x \Rightarrow \Theta(f)$ (with regularity); $f \asymp x \Rightarrow \Theta(x \log n)$.

The complexity ladder

P $\subseteq$ NP; NP-hard = everything in NPreduces to it; NP-complete = NP-hard $\cap$ NP. To prove hardness, reduce *from* a known-hard problem *to* the new one.

Material © University of Waterloo / CS231 course staff. Personal study copy — do not redistribute. Generated from the consolidated course notes; for verbatim per-unit notes see `notes/`, for figures, code, and PDFs see `resources/`.