Week 1 — Algorithms, Complexity & Divide-and-Conquer

Félix Chavelli felix.chavelli@inria.fr

February 18, 2026 · Semester 2

PSL

Today's agenda

Welcome & Course Overview

Review: Algorithms & Complexity

Asymptotic Notation — Formal Definitions

Divide-and-Conquer

Solving Recurrences & The Master Theorem

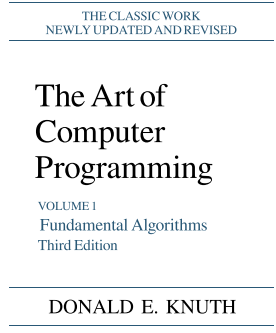
Summary & What's Next

Part 1

Welcome & Course Overview

Welcome to Semester 2

- Welcome back! S1 laid the foundations:
  - Variables, types, control flow
  - Functions, recursion, OOP basics
  - Big-$\mathcal{O}$ basics, intro to DP
  - Arrays, linked lists, stacks, queues
- S2 takes you to the next level:
  - Deeper algorithmic theory
  - Advanced data structures
  - Graph algorithms
  - Numerical computing
- **Icebreaker:** What was the hardest concept from S1?



D. Knuth, *TAOCP*

The Teaching Team

Lecturer

Félix Chavelli

felix.chavelli@inria.fr

TA (Wk 1–6)

Emmanouil Sylligardos

emmanouil.sylligardos@inria.fr

TA (Wk 7–15)

Magali Parrino

magali.parrino@inria.fr

Contact Policy

Email with `[BAI-TAOCP2]` in the subject line

All materials on **Moodle**: `https://moodle.psl.eu/course/view.php?id=37247`

Why This Course?

Motivation

"A Data Scientist who understands the math but ignores the machine will produce inefficient solutions."

- Training a neural network = matrix multiplications + graph traversal + optimization
- GPT-scale models require algorithmic thinking at scale
- S1 gave you the tools — S2 teaches you to build with them efficiently

The three pillars of S2

15-Week Roadmap

Wk	Topic	CLRS
1	**Algorithms, Complexity & D&C** *today*	Ch. 1–4
2	Advanced OOP & Design Patterns	—
3	Heaps, Heapsort & Priority Queues	Ch. 6
4	Quicksort & Linear-time Sorts	Ch. 7–8
5	Hash Tables	Ch. 11
6	BSTs & Red-Black Trees	Ch. 12–13
7	Advanced Dynamic Programming	Ch. 15
8	Greedy Algorithms & Amortized Analysis	Ch. 16–17
9	Graphs: BFS, DFS, Topological Sort	Ch. 22–23
10	MST, Union-Find, Dijkstra, Bellman-Ford	Ch. 24
11	A*, Floyd-Warshall	Ch. 25
12	Numerical Methods (integration, roots, ODEs)	—
13	Matrix Operations & Linear Algebra	Ch. 28
14–15	*Group Project & Presentations*	—

Assessment

Grading

Final exam (written, 1 h 30)	**60 %**
Group project (weeks 14–15)	**40 %**

- **Project:** teams of 2–3 students
- Topics revealed later in the semester
- Deliverables: code + oral presentation

Weekly Warm-up Quizzes

- 5–10 min at the **start** of each lecture (from week 2)
- Covers last week's material
- Not graded!
- Purpose: self-assessment, stay on track

A Note on LLMs

- LLMs are good at generating content similar to their training data
- $\Rightarrow$ usually quite good at simple exercises
- However:
  - You won't learn if the LLM solves problems for you
  - Solutions may be hard to understand
  - They may use constructs outside the scope of this class
  - They can be wrong (hallucinations)

Policy

Not recommended, except when you are confident enough to **fully verify and understand** the output.

You learn by doing, not by prompting.

What S1 Covered — Quick Recap

#	Topic
1	Introduction: C / C++ / Python, binary representation, variables, types
2	Structured programming: control flow, functions, recursion
3	Algorithm quality: Big-$\mathcal{O}$, Python collections (`list`, `dict`, `set`)
4	Memory management: stack vs heap, pointers, `malloc`/`new`
5	OOP basics: classes, encapsulation, constructors, `self`/`this`
6	Data structures: arrays, linked lists, stacks, queues, deques
7	Algorithmic techniques: brute force, greedy, DP (intro)
8	Exceptional situations: exceptions, RAII, smart pointers
10	I/O: streams, files, buffering, serialization
—	Sorting (intro), testing, functional programming, Git, regex

→ **Today we deepen:** formal asymptotic analysis, divide-and-conquer, master theorem.

Part 2

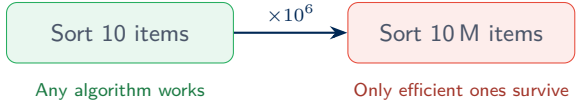
Review: Algorithms & Complexity

What is an Algorithm?

Definition (CLRS Ch. 1)

An **algorithm** is a well-defined computational procedure that takes some value (or set of values) as **input** and produces some value (or set of values) as **output** in a **finite** number of steps.

- **Correctness:** does it solve the problem for *all* valid inputs?
- **Efficiency:** how does it scale as input grows?
- Same problem, very different scale:



Insertion Sort — Revisited

Input: Array $A[0 \ldots n{-}1]$
Output: A sorted in non-decreasing order

1: **for** $i \leftarrow 1$ **to** $n{-}1$ **do**
2: $\quad key \leftarrow A[i]$
3: $\quad j \leftarrow i - 1$
4: $\quad$ **while** $j \geqslant 0$ **and** $A[j] > key$ **do**
5: $\quad\quad A[j{+}1] \leftarrow A[j]$
6: $\quad\quad j \leftarrow j - 1$
7: $\quad$ **end while**
8: $\quad A[j{+}1] \leftarrow key$
9: **end for**

Walk-through:

$[\mathbf{5}, 2, 4, 6, 1, 3]$

$[2, \mathbf{5}, 4, 6, 1, 3]$

$[2, 4, \mathbf{5}, 6, 1, 3]$

$[2, 4, 5, \mathbf{6}, 1, 3]$

$[\mathbf{1}, 2, 4, 5, 6, 3]$

$[1, 2, \mathbf{3}, 4, 5, 6]$ ✔

Best case: $\Theta(n)$
Worst case: $\Theta(n^2)$

Insertion Sort — Detailed Analysis

Cost per line (let t_i = number of times the **while** test runs for index i):

Line	Cost	Times
`for` $i \leftarrow 1$ to $n-1$	c_1	n
$key \leftarrow A[i]$	c_2	$n-1$
$j \leftarrow i-1$	c_3	$n-1$
`while` $j \geqslant 0$ and $A[j] > key$	c_4	$\sum_{i=1}^{n-1} t_i$
$A[j+1] \leftarrow A[j]$	c_5	$\sum_{i=1}^{n-1}(t_i-1)$
$j \leftarrow j-1$	c_6	$\sum_{i=1}^{n-1}(t_i-1)$
$A[j+1] \leftarrow key$	c_7	$n-1$

✓ Best case (already sorted)

Each $t_i = 1$ (while test fails immediately)

$T(n) = (c_1+c_2+c_3+c_4+c_7)\,n - \text{const}$

$\Rightarrow \Theta(n)$

✗ Worst case (reverse sorted)

Each $t_i = i$, so $\sum t_i = \frac{n(n-1)}{2}$

$T(n) = \frac{c_4+c_5+c_6}{2}\,n^2 + \ldots$

$\Rightarrow \Theta(n^2)$

Why Efficiency Matters

	$n = 10^3$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(n)$	10^3	10^6	10^9
$\mathcal{O}(n \log n)$	10^4	10^7	10^{10}
$\mathcal{O}(n^2)$	10^6	10^{12}	10^{18}
$\mathcal{O}(2^n)$	10^{301}	heat death of universe	

Key Idea

The difference between $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^2)$ is the difference between **6 milliseconds** and **16 minutes** for $n = 10^6$ operations at 10^9 ops/sec.

Part 3

Asymptotic Notation — Formal Definitions

Recall from S1 — Informal View

- $\mathcal{O}(g(n))$ — "grows **at most** as fast as $g(n)$" (upper bound)
- $\Omega(g(n))$ — "grows **at least** as fast as $g(n)$" (lower bound)
- $\Theta(g(n))$ — "grows **exactly** as fast as $g(n)$" (tight bound)

Now: formal definitions with quantifiers

Big-$\mathcal{O}$: Upper Bound

$\mathcal{O}$-notation (CLRS 3.1)

$f(n) = \mathcal{O}(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that:

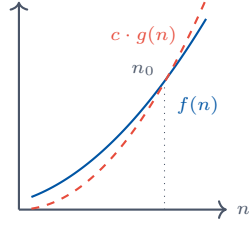
$$0 \leqslant f(n) \leqslant c \cdot g(n) \qquad \forall\, n \geqslant n_0$$

Example: Show that $2n^2 + 3n + 1 = \mathcal{O}(n^2)$.

For $n \geqslant 1$: $3n \leqslant 3n^2$ and $1 \leqslant n^2$

$$2n^2 + 3n + 1 \leqslant 2n^2 + 3n^2 + n^2 = 6n^2$$

$\Rightarrow$ Choose $c = 6$, $n_0 = 1$ ✔



Ω and Θ: Lower & Tight Bounds

Ω-notation

$f(n) = \Omega(g(n))$ if $\exists\, c > 0,\ n_0 > 0$ such that $f(n) \geqslant c \cdot g(n)$ for all $n \geqslant n_0$.

Θ-notation

$f(n) = \Theta(g(n))$ if and only if $f(n) = \mathcal{O}(g(n))$ **and** $f(n) = \Omega(g(n))$.

Equivalently: $\exists\, c_1, c_2 > 0,\ n_0$ such that $c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$ for all $n \geqslant n_0$.

Key Idea

Θ gives a **tight** characterization: f and g grow at the same rate (up to constants).

Little-o and Little-ω (briefly)

- $f(n) = o(g(n))$: f grows strictly slower than g

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

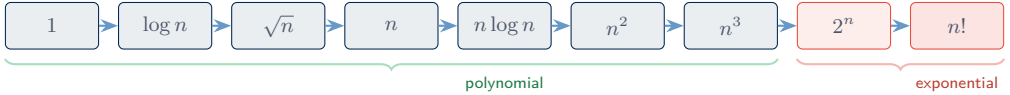
- $f(n) = \omega(g(n))$: f grows strictly faster than g

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Analogy:

$$\begin{array}{lcll} f = \mathcal{O}(g) & \sim & f \leqslant g & \text{(asymptotically)} \\ f = o(g) & \sim & f < g & \\ f = \Omega(g) & \sim & f \geqslant g & \\ f = \omega(g) & \sim & f > g & \\ f = \Theta(g) & \sim & f = g & \end{array}$$

Growth Rate Ranking



Quick quiz — What is the complexity of...

1. Looking up a key in a hash table?
2. Sorting n elements with merge sort?
3. Checking all pairs in an array of size n?
4. Generating all subsets of a set of size n?

Growth Rate Ranking — Answers

1. **Looking up a key in a hash table?** $\mathcal{O}(1)$ on average

   Hash function maps the key directly to a bucket index. Worst case is $\mathcal{O}(n)$ if all keys collide, but with a good hash function this is extremely unlikely.

2. **Sorting n elements with merge sort?** $\Theta(n \log n)$

   The array is split in half at each level ($\log n$ levels), and each level does $\Theta(n)$ work for merging.

3. **Checking all pairs in an array of size n?** $\Theta(n^2)$

   Two nested loops: $\displaystyle\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$.

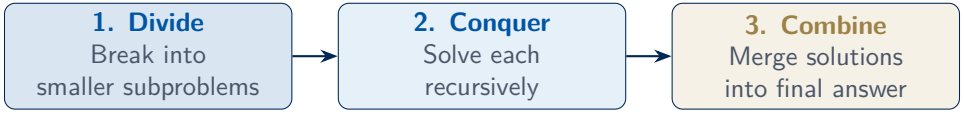
4. **Generating all subsets of a set of size n?** $\Theta(2^n)$

   Each element is either included or excluded $\Rightarrow 2^n$ subsets. Just *enumerating* them already takes $\Theta(2^n)$.

Part 4

Divide-and-Conquer

The Divide-and-Conquer Paradigm



- A design pattern for algorithms — one of three paradigms this semester
- Works when the problem has **recursive structure**
- Key question: how fast can we **combine**?

Key Idea

The efficiency of D&C depends on the **balance** between the number of subproblems, their size, and the cost of combining.

Merge Sort — The Algorithm

Merge-Sort(A, p, r)**:**

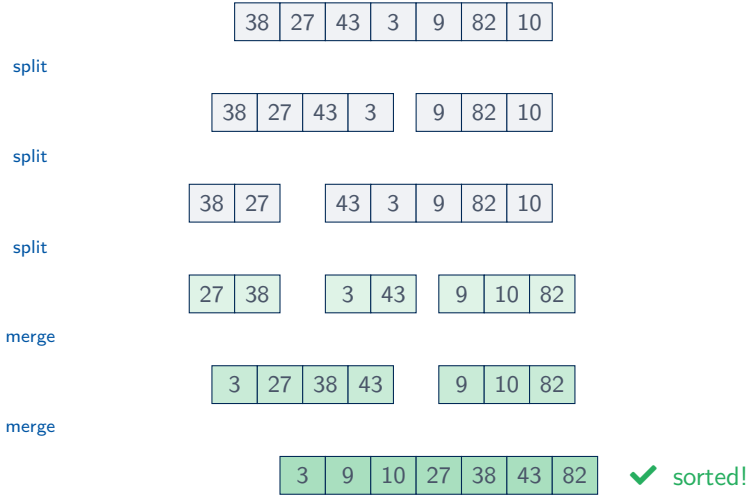
1: **if** $p < r$ **then**
2: $q \leftarrow \lfloor (p + r)/2 \rfloor$
3: MERGE-SORT(A, p, q)
4: MERGE-SORT$(A, q{+}1, r)$
5: MERGE(A, p, q, r)
6: **end if**

Divide: compute midpoint $\mathcal{O}(1)$
Conquer: 2 subproblems of size $n/2$
Combine: merge in $\Theta(n)$

Merge(A, p, q, r)**:**

1: Create arrays $L \leftarrow A[p \dots q]$,
$R \leftarrow A[q{+}1 \dots r]$
2: $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow p$
3: **while** $i < |L|$ **and** $j < |R|$ **do**
4: **if** $L[i] \leqslant R[j]$ **then**
5: $A[k] \leftarrow L[i]$; $i \mathrel{+}= 1$
6: **else**
7: $A[k] \leftarrow R[j]$; $j \mathrel{+}= 1$
8: **end if**
9: $k \mathrel{+}= 1$
10: **end while**
11: Copy remaining elements of L or R into A

Merge Sort — Walk-through



Merge Sort — Analysis

- **Recurrence:**

$$T(n) = \underbrace{2\,T(n/2)}_{\text{2 subproblems}} + \underbrace{\Theta(n)}_{\text{merge}} \qquad T(1) = \Theta(1)$$

- **Solution:** $T(n) = \Theta(n \log n)$
  - $\log n$ levels in the recursion tree
  - $\Theta(n)$ total work at each level

	Insertion	Merge
Best	$\Theta(n)$	$\Theta(n \log n)$
Worst	$\Theta(n^2)$	$\Theta(n \log n)$
Average	$\Theta(n^2)$	$\Theta(n \log n)$

Key Idea

Merge sort **guarantees** $\Theta(n \log n)$ in all cases, but uses extra memory. Insertion sort wins for **small** or **nearly sorted** inputs.

Binary Search — Another D&C Example

Input: Sorted array $A[0 \ldots n-1]$, target x
Output: Index of x, or -1

```
1: ℓ ← 0,  r ← n − 1
2: while ℓ ⩽ r do
3:     m ← ⌊(ℓ + r)/2⌋
4:     if A[m] = x then
5:         return m
6:     else if A[m] < x then
7:         ℓ ← m + 1
8:     else
9:         r ← m − 1
10:    end if
11: end while
12: return −1
```

Analysis:

$$T(n) = T(n/2) + \Theta(1)$$

$$\Rightarrow T(n) = \Theta(\log n)$$

- **Divide:** compare with middle element
- **Conquer:** recurse on one half only
- **Combine:** nothing to do!

Maximum Subarray Problem (optional)

Problem: Given an array of integers, find the contiguous subarray with the largest sum.

Example: $[-2, 1, -3, \mathbf{4}, \mathbf{-1}, \mathbf{2}, \mathbf{1}, -5, 4] \rightarrow$ answer $= 6$

D&C approach:

1. **Divide** at the midpoint
2. **Conquer:** find max subarray in left half and right half
3. **Combine:** find max subarray *crossing* the midpoint (scan from middle outward)
4. Return the maximum of the three

$$T(n) = 2\,T(n/2) + \Theta(n) \quad \Rightarrow \quad \Theta(n \log n)$$

Key Idea

Preview: Kadane's algorithm solves this in $\Theta(n)$ using dynamic programming — we'll revisit this in Week 7!

Part 5

Solving Recurrences & The Master Theorem

Why Recurrences?

- Every recursive algorithm yields a recurrence for its running time
- We need tools to **solve** these recurrences $\rightarrow$ closed-form $\Theta(\cdot)$
- Three methods:
  1. **Substitution:** guess + prove by induction
  2. **Recursion tree:** draw the tree, sum costs
  3. **Master theorem:** "cookbook" formula

Method 1: Substitution

Claim: $T(n) = 2\,T(n/2) + n$ is $\mathcal{O}(n \log n)$.

Proof by strong induction: Assume $T(k) \leqslant c\,k \log k$ for all $k < n$.

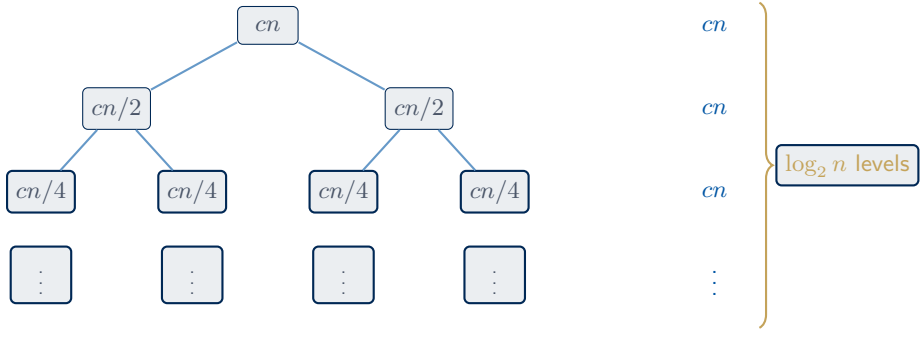
$$\begin{aligned} T(n) &= 2\,T(n/2) + n \\ &\leqslant 2 \cdot c \cdot \tfrac{n}{2} \cdot \log(n/2) + n \\ &= c\,n\,\log n - c\,n\,\log 2 + n \\ &= c\,n\,\log n - n(c \log 2 - 1) \\ &\leqslant c\,n\,\log n \qquad \text{for } c \geqslant \frac{1}{\log 2} \end{aligned}$$

Caveat

The substitution method requires you to **guess** the answer first! The recursion tree or master theorem help you make the right guess.

Method 2: Recursion Tree

Visual approach for $T(n) = 2\,T(n/2) + cn$



Total: $cn \times \log_2 n = \Theta(n \log n)$

Method 3: The Master Theorem

A "cookbook" for recurrences

Master Theorem

For a recurrence of the form $T(n) = a\,T(n/b) + f(n)$ with $a \geqslant 1$, $b > 1$, let $c_{\text{crit}} = \log_b a$:

Case	Condition	Result
1	$f(n) = \mathcal{O}(n^{c_{\text{crit}} - \varepsilon})$ for some $\varepsilon > 0$	$T(n) = \Theta(n^{c_{\text{crit}}})$
2	$f(n) = \Theta(n^{c_{\text{crit}}})$	$T(n) = \Theta(n^{c_{\text{crit}}} \log n)$
3	$f(n) = \Omega(n^{c_{\text{crit}} + \varepsilon})$ + regularity	$T(n) = \Theta(f(n))$

Regularity: there exists $\alpha < 1$ s.t. for all sufficiently large n, $a\,f(n/b) \leqslant \alpha\,f(n)$

Key Idea

Intuition: compare the cost of the **leaves** ($n^{\log_b a}$) vs. the cost of the **root** ($f(n)$). Whichever dominates determines the answer.

Master Theorem — Worked Examples

Recurrence	a	b	$n^{\log_b a}$	Case	Result
$T(n) = 2T(n/2) + \Theta(n)$	2	2	n^1	2	$\Theta(n \log n)$
→ Merge sort					
$T(n) = T(n/2) + \Theta(1)$	1	2	$n^0{=}1$	2	$\Theta(\log n)$
→ Binary search					
$T(n) = 7T(n/2) + \Theta(n^2)$	7	2	$n^{2.81\ldots}$	1	$\Theta(n^{\log_2 7})$
→ Strassen's algorithm (preview — week 13!)					
$T(n) = 4T(n/2) + \Theta(n^3)$	4	2	n^2	3	$\Theta(n^3)$

Limitations

The master theorem does **not** apply when there is no polynomial gap between $f(n)$ and $n^{\log_b a}$.

Example: $T(n) = 2T(n/2) + n \log n$ — use the recursion tree method instead $\Rightarrow \Theta(n \log^2 n)$.

Part 6

Summary & What's Next

Summary — Key Takeaways

1. **Asymptotic notation** is a formal mathematical tool — not just "drop the constants"
   - $\mathcal{O}$ (upper), Ω (lower), Θ (tight), with quantifiers
2. **Divide-and-Conquer:** Divide $\rightarrow$ Conquer recursively $\rightarrow$ Combine
   - Merge sort: $\Theta(n \log n)$ guaranteed
   - Binary search: $\Theta(\log n)$
3. **The Master Theorem:** cookbook for $T(n) = aT(n/b) + f(n)$
   - Compare $f(n)$ with $n^{\log_b a}$ — three cases
4. Always **verify theory with practice** $\rightarrow$ that's what the lab is for!

What's Next — Week 2

→ Next week: Advanced OOP — Inheritance, Polymorphism & Design Patterns

- Inheritance: simple & multiple (MRO in Python, access specifiers in C++)
- Polymorphism: virtual functions, duck typing
- Abstract classes / interfaces
- Operator overloading: building a `Vector` class
- Design patterns: Strategy, Iterator, Decorator

To prepare: Review your S1 notes on classes, constructors, and encapsulation.

? Questions?

References

- **Cormen, Leiserson, Rivest, Stein** — *Introduction to Algorithms*, 4th ed., MIT Press, 2022. **Chapters 1–4.**

PSL UNIVERSITÉ PARIS

PSL University · Bachelor of Science in AI · 2025–2026