Lecture 3 – Asymptotic Analysis and Algorithmic Correctness: Θ/O/Ω, Invariants, and Empirical Validation

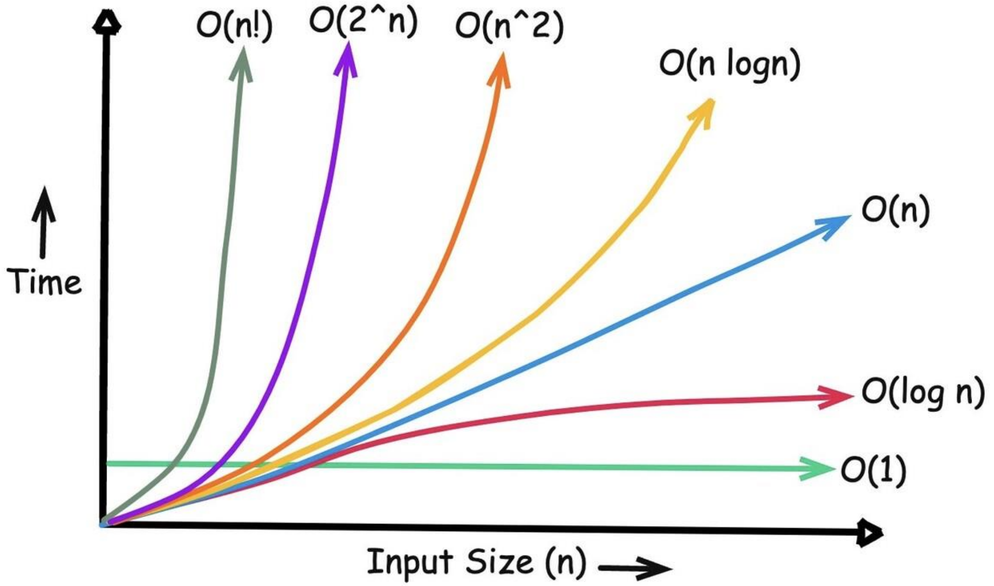
NURSULTAN KHAIMULDIN

N.KHAIMULDIN@ASTANAIT.EDU.KZ

Content

1. Looking back at the last course
2. Extending asymptotic notation
3. Correctness of algorithms
4. Loop invariants
5. Empirical validation
6. Amortized analysis

Looking back at the last course



Upper bound on growth rate (worst-case scale behavior)

Compare functions as $n \to \infty$; ignore constants & lower-order terms

$f(n) \in O(g(n)) \Leftrightarrow \exists\ c>0,\ n_0$ s.t. $\forall\ n \geq n_0$: $f(n) \leq c \cdot g(n)$

$7n \to O(n)$; $3n^2 + 10n \to O(n^2)$

$n^3 + 100n \log n + 5 \to O(n^3)$

$O(1)$ (constant), $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$

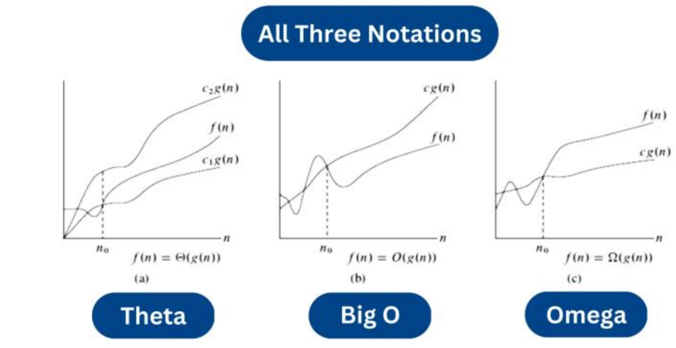
Worst-case runtime, space, comparisons, I/O ops (as a function of n)

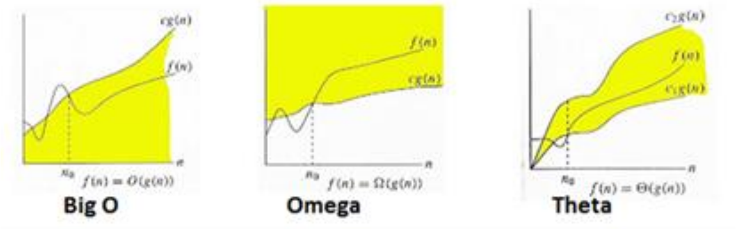
Example:

- Linear search: $O(n)$
- Binary search: $O(\log n)$
- Insertion sort (worst-case): $O(n^2)$
- Merge sort: $O(n \log n)$

ASTANA IT UNIVERSITY

Extending asymptotic notation





O is only an **upper bound,** but we also need **lower** and **tight** bounds.

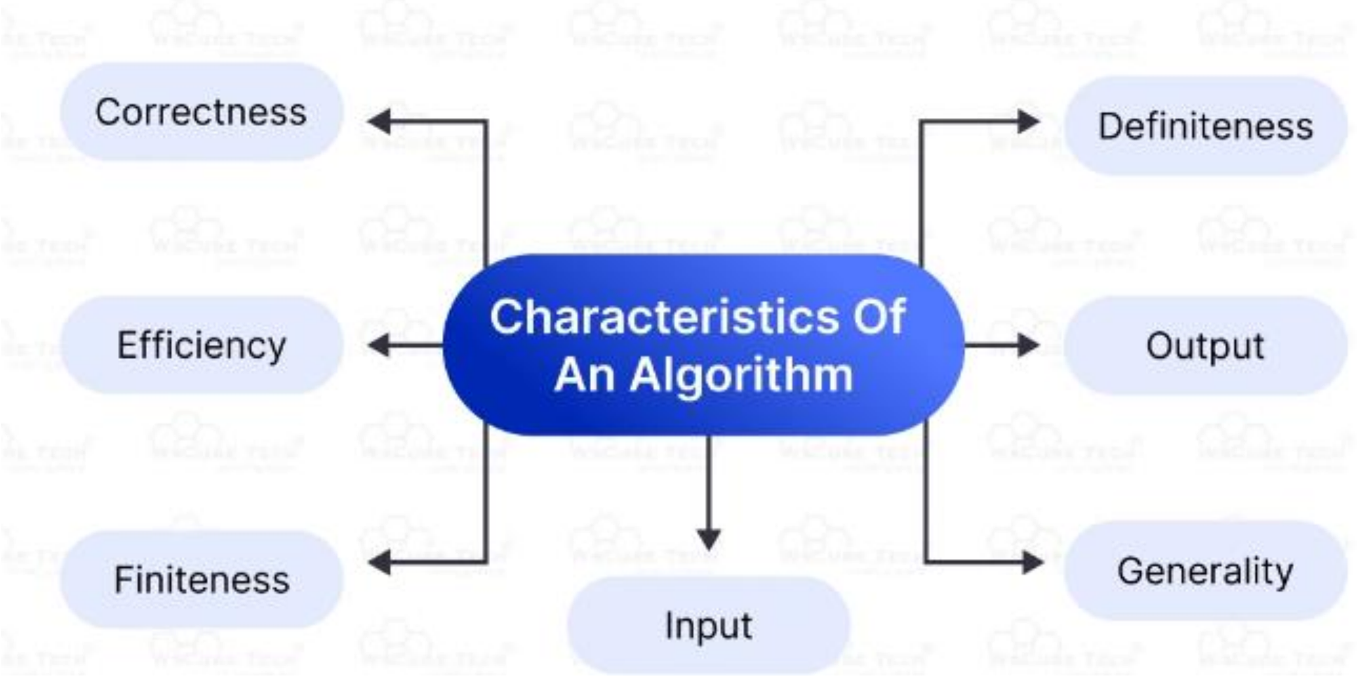
Core set:

- **Ω(g):** lower bound (eventually $\geq c \cdot g$)
- **Θ(g):** tight bound (sandwiched: $c_1 g \leq f \leq c_2 g$)
- **o(g):** strictly smaller; **ω(g):** strictly larger

Understanding:

- **O**: “It won’t be worse than this (eventually).”
- **Ω**: “It can’t be better than this (eventually).”
- **Θ**: “It’s this, up to constants.”

Correctness of algorithms



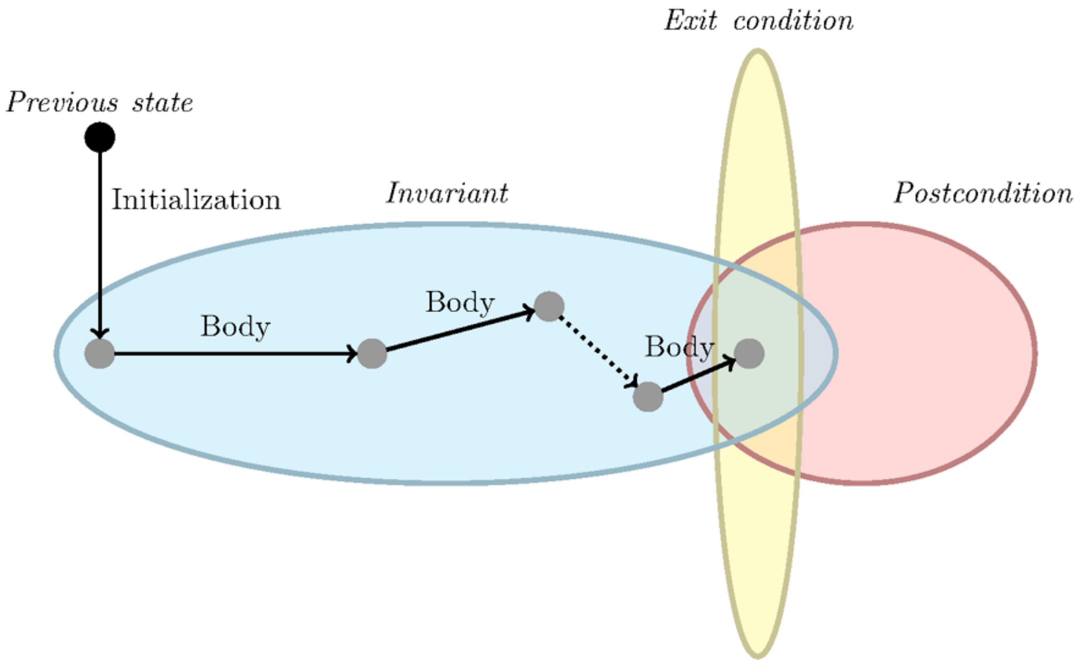
For any allowed input, the algorithm **halts** and the result **meets the spec**.

Write the spec first:

- **Preconditions** – what you assume (e.g., array is sorted, graph is unweighted).
- **Postconditions** – what must be true after (e.g., output is a permutation and is sorted).
- **Effects** – which state may change (no hidden side effects).

Proof obligations:

1. **Initialization** – your invariant holds before the loop/recursion.
2. **Preservation** – one step keeps the invariant true.
3. **Progress/Termination** – some measure strictly decreases (problem size shrinks).
4. **Postcondition** - invariant + exit condition ⇒ required result.



Loop invariants

A loop invariant is a statement about your program's state that is **true before the loop starts** and **remains true after every iteration**.

It captures "what is already correct" at each step, so you can **prove the loop's result** when it finishes.

How it proves correctness:

1. **Initialization:** it's true before the first iteration.
2. **Maintenance:** one iteration keeps it true.
3. **Termination:** when the loop ends, the invariant **plus the exit condition** implies the goal (postcondition).

Formulating a good invariant:

- Describe the **portion already correct** (a sorted prefix)
- Capture **where the work remains** (the unsorted suffix)
- Be **strong enough** to imply the goal at exit, but **simple enough** to check each step

Examples:

- **Insertion Sort:** "Prefix A[0..i] is sorted; all elements belong to final set."
- **Selection Sort:** "Prefix A[0..i-1] holds the i smallest elements in order."

  Binary Search: "If the target exists, it is inside [lo..hi]; each step shrinks this interval."
- **BFS:** "Queue contains the current frontier; discovered nodes have correct levels."

Empirical validation

Empirical validation is the **measurement-based check** that an algorithm's **real-world behavior** matches its **theoretical expectations** (scaling trend, resource use).

Measuring:

- Runtime, CPU time
- Operation counts, allocations/memory
- Cache/branches from profilers

Purpose:

- Confirms or challenges asymptotic predictions in practice
- Reveals **constants, cross-over points, variance**, and **system effects** (cache, JIT/GC, branch prediction, memory layout)
- Builds **credibility and reproducibility** for your results

How to work:

- **Form a hypothesis**: state expected growth (e.g., "~ n log n"), the **case** (worst/average/best), and **metrics** (time, comparisons, memory).
- **Control the environment**: same machine, release/optimized build, stable settings; **warm up** for JITed runtimes; minimize background load.
- **Design inputs**: include **random**, **best/worst**, **adversarial**, and **realistic** datasets; fix **seeds** for reproducibility.
- **Scale & repeat**: grow input size by **doubling**; run **multiple trials** per size; report **median** and **spread**. Avoid I/O in timed code.
- **Analyze trends**: check curve **shape** and **doubling behavior**; optionally normalize (e.g., time per n or per n·log n) to test fit; note **cross-overs**.
- **Report transparently**: list hardware/OS, compiler/JVM and flags, dataset description, seeds, number of trials; include clear plots and a compact table.

Amortized analysis

A method to bound the **average cost per operation over any worst-case sequence** of operations.

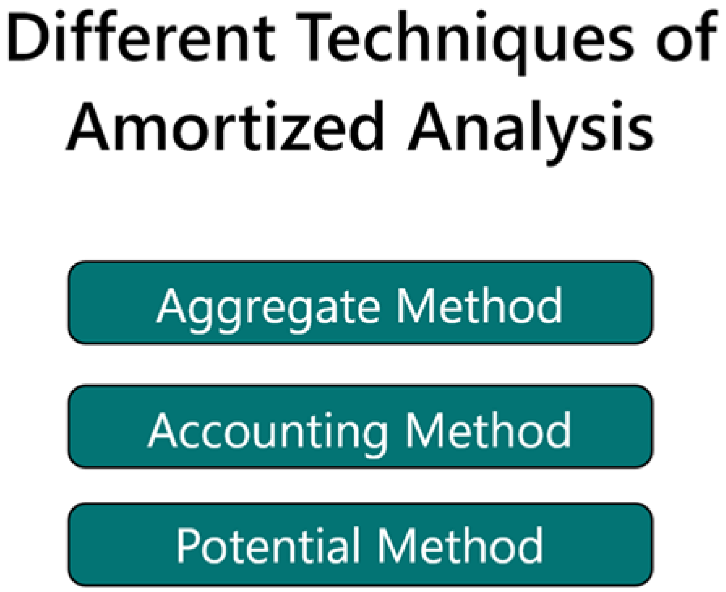
Not probabilistic: **different from average-case** (no input distribution; guarantee holds for every sequence).

Different techniques:

- **Aggregate method:** Bound the **total** cost of *m* operations, then divide by *m*.
- **Accounting method:** Charge some ops extra “credits” that pay for rare expensive ops later; credits never go negative.
- **Potential method:** Define a potential (energy) on the data structure’s state; amortized cost = actual work ± change in potential.

How to work:

1. **Specify the operation sequence & policy** (e.g., array doubles when full; hash table rehashes at load factor α).
2. **Choose a technique** (aggregate/accounting/potential) and a simple **cost model** (e.g., element moves, comparisons).
3. Prove the bound over any sequence:
   - Aggregate: show total work ≤ K·m → amortized ≤ K.
   - Accounting: assign per-op charges so credits cover future expensive steps.
   - Potential: pick a potential that never goes negative and drops when an expensive step happens.
4. **State the result clearly:** worst-case per op vs **amortized** per op; note assumptions (growth factor, load factor).
5. **Check for correctness:** construct adversarial sequences and (optionally) measure to see the predicted average holds.



Literature

Algorithms (4th ed.), Robert Sedgewick & Kevin Wayne

1.4 Analysis of Algorithms – worst-case guarantees, randomized guarantees, and an intro to amortized analysis via resizing arrays

3.2 Binary Search in an Ordered Array – iterative and recursive versions, with correctness notes

Introduction to Algorithms (4th ed.) – Cormen, Leiserson, Rivest, Stein (CLRS)

Ch. 3.1 O-, Ω-, Θ-notation – formal asymptotic notation.

Ch. 2 Insertion Sort – loop invariants and how they prove correctness.

Ch. 2: Best/Worst/Average-case discussion – why worst-case is often the focus.

Algorithms: Design, Techniques, and Analysis – M. H. Alsuwaiyel

1.8 Time complexity – O, Ω, Θ (and related ideas).

1.12 Worst-Case and Average-Case Analyses;

1.13 Amortized Analysis

Grokking Algorithms – Aditya Bhargava (Manning)

Ch. 1 Big-O (intro) and *Ch. 4 Quicksort* – clear average- vs worst-case explanation.

ASTANA IT UNIVERSITY

Thank you