Growth Functions

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why growth matters

An algorithm that works on a small example may become unusable when its input grows. Growth analysis asks how required work, memory, communication, or another resource depends on the size and shape of the input. It supports machine-independent comparison while deliberately ignoring many implementation details.

Asymptotic analysis and measurement are complementary. Analysis explains long-run scaling under a model; experiments reveal constants, memory behaviour, compiler effects, and the input range that matters in practice.

2 Learning goals

After studying this note, you should be able to:

- choose meaningful input-size parameters and a cost model;
- distinguish worst-case, average-case, expected, and amortised claims;
- derive a cost function from loops, calls, and common sums;
- use O, Ω, Θ, o, and ω precisely;
- prove a simple asymptotic bound from its definition;
- compare standard growth classes and practical crossover points;
- analyse time and auxiliary space separately;
- expand a simple divide-and-conquer recurrence; and
- design timing experiments without presenting them as proofs.

3 Prerequisites

You should understand basic algebra, functions, powers, logarithms, loops, arrays, and recursion. The preceding orientation note introduced inputs, representations, and the unit-cost RAM model.

4 Input size and cost models

Before counting work, decide what the input size means. For sorting, n is normally the number of records. A graph may require two parameters, $|V|$ and $|E|$. A matrix multiplication involves dimensions, not merely one generic n. For an integer x, the representation length is $\lfloor \log_2 x \rfloor + 1$ bits when $x > 0$.

This last distinction prevents misleading claims. An algorithm taking $O(x)$ steps is exponential in the bit length of x, not polynomial in it. Such a bound is often called **pseudo-polynomial** when it is polynomial in numeric values but not in the length of their encoding.

A **cost model** states which operations are counted. Possibilities include comparisons, array accesses, arithmetic operations on machine words, allocated words, block transfers, messages, or calls to an expensive subroutine. A result is meaningful only together with its model.

5 Cases, probability, and sequences

Inputs of the same size can produce different costs:

- the **worst-case** cost is the maximum over inputs of size n;
- the **best-case** cost is the minimum;
- the **average-case** cost is an expectation over a stated probability distribution on inputs; and
- the **expected cost of a randomized algorithm** fixes an input and averages over the algorithm's random choices.

These expectations are not interchangeable. "Random-looking test data" is not a mathematical input distribution, and random pivot selection is not an assumption that the original input is random.

Amortised analysis uses no probability. It bounds the total cost of every allowed operation sequence and assigns that total across the operations. A single operation may remain expensive even when the amortised cost per operation is small.

Each of these labels applies to a cost function, not directly to a line of code. State the size parameter first, then say which inputs or random choices the function quantifies over.

6 From code to a cost function

Count executions rather than multiplying visible loop bounds mechanically. Consider:

```
for i = 1 to n:
    for j = 1 to i:
        do constant work
```

The number of executions of the inner operation is

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

Therefore the running time is $\Theta(n^2)$. Two nested loops happen to give a quadratic result here, but the sum explains why; if the inner bound were $\lfloor \log i \rfloor$, the result would be different.

Consecutive phases add. Nested independent work usually multiplies. Conditional branches require a case assumption: a worst-case analysis uses the most costly feasible branch, while an expected analysis needs probabilities.

7 A small mathematical toolkit

Several sums recur throughout algorithm analysis:

$$\sum_{i=1}^{n} 1 = n, \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2),$$

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1, \qquad \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n).$$

The geometric series explains dynamic-array resizing and levels of complete trees. The harmonic sum appears in probabilistic analyses.

Useful logarithm rules are

$$\log(xy) = \log x + \log y, \qquad \log(x^a) = a \log x,$$

and

$$\log_b n = \frac{\log_a n}{\log_a b}.$$

Changing a fixed logarithm base multiplies by a constant, so the base normally disappears inside asymptotic notation.

8 Asymptotic definitions

Assume f and g are non-negative for sufficiently large n. Formally, $O(g)$ is a set of functions. The notation $f \in O(g)$ is therefore clearest, although textbooks conventionally write $f = O(g)$ as one-way shorthand. It is not an algebraic equality: from $f = O(g)$ one may not infer $g = O(f)$.

8.1 Upper, lower, and tight bounds

We have $f \in O(g)$ if constants $c > 0$ and n_0 exist such that

$$0 \le f(n) \le cg(n) \quad \text{for all } n \ge n_0.$$

We have $f \in \Omega(g)$ if constants $c > 0$ and n_0 exist such that

$$0 \le cg(n) \le f(n) \quad \text{for all } n \ge n_0.$$

Finally, $f \in \Theta(g)$ when both bounds hold. Equivalently, positive c_1, c_2, n_0 exist with

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \quad \text{for all } n \ge n_0.$$

Worst case says which cost function is under discussion; Big O says how a function is bounded. Big O does not itself mean worst case.

8.2 Strict asymptotic relations

We have $f \in o(g)$ if, for **every** constant $c > 0$, some n_0 makes

$$0 \le f(n) < cg(n) \quad \text{for all } n \ge n_0.$$

Thus f becomes smaller than every positive constant multiple of g. Conversely, $f \in \omega(g)$ if, for every $c > 0$, eventually $f(n) > cg(n)$.

When $g(n) > 0$ eventually and the relevant limit exists,

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

implies $f \in o(g)$. A finite positive limit implies $f \in \Theta(g)$, and an infinite limit implies $f \in \omega(g)$. The constant-based definitions still apply when a ratio limit does not exist.

9 Worked proof from the definition

Let $f(n) = 3n^2 + 7n + 20$. For every $n \ge 1$,

$$3n^2 + 7n + 20 \le 3n^2 + 7n^2 + 20n^2 = 30n^2.$$

Choosing $c_2 = 30$ and $n_0 = 1$ proves $f \in O(n^2)$. Also $f(n) \ge 3n^2$, so $c_1 = 3$ and $n_0 = 1$ prove $f \in \Omega(n^2)$. Hence

$$3n^2 + 7n + 20 \in \Theta(n^2).$$

The constants need only work eventually; they need not be smallest possible.

10 Common growth classes

For fixed $k > 1$, $a > 1$, and $d > 1$,

$$1 \prec \log n \prec (\log n)^k \prec n \prec n \log n \prec n^d \prec a^n \prec n!.$$

Here $f \prec g$ informally denotes $f \in o(g)$. The condition $d > 1$ is essential: for $d = 1$, $n \log n$ grows faster than $n^d = n$.

Growth	Informal name	Typical source
$\Theta(1)$	constant	one indexed array access
$\Theta(\log n)$	logarithmic	repeatedly halve a range
$\Theta(n)$	linear	inspect every item once
$\Theta(n \log n)$	linearithmic	balanced divide and conquer
$\Theta(n^2)$	quadratic	inspect every pair
$\Theta(2^n)$	exponential	enumerate all subsets
$\Theta(n!)$	factorial	enumerate all permutations

Dominant terms determine a sum's asymptotic class when terms are eventually non-negative. Thus

$$n^3 + 100n \log n + 10^6 \in \Theta(n^3).$$

This does not make lower-order terms or constants irrelevant at finite sizes.

11 Recurrences from first principles

A recurrence describes a recursive call tree. It must include a base case. For powers of two, consider

$$T(1) = d, \qquad T(n) = 2T(n/2) + cn \quad (n > 1).$$

At level i, there are 2^i subproblems of size $n/2^i$. Their non-recursive work totals

$$2^i c \frac{n}{2^i} = cn.$$

There are $\log_2 n$ non-leaf levels, plus n constant-cost leaves. Therefore

$$T(n) = cn \log_2 n + dn = \Theta(n \log n).$$

For arbitrary n, floors and ceilings change constants but not this asymptotic result. A later note gives the Master theorem; expanding a few levels first is safer than applying a memorised formula to a recurrence that does not fit it.

12 Worked comparison: constants versus growth

Suppose two implementations are modelled by

$$T_A(n) = 50n \log_2 n, \qquad T_B(n) = n^2.$$

Although $T_A \in o(T_B)$, A is faster only when $50 \log_2 n < n$. At $n = 128$, the left side is 350, so B wins in this model. At $n = 512$, it is 450, so A wins. Asymptotic order predicts eventual behaviour; an exact model or measurement locates the relevant crossover.

13 Time, space, and trade-offs

Analyse different resources separately. Iteratively summing an array takes $\Theta(n)$ time and $\Theta(1)$ auxiliary space. A direct recursive version still takes $\Theta(n)$ time but uses $\Theta(n)$ call-stack space. An algorithm may deliberately spend memory to save recomputation, or spend time to reduce storage.

Also distinguish **auxiliary space** from total storage. If an input array already occupies $\Theta(n)$ words, saying an in-place scan uses $\Theta(1)$ auxiliary space does not mean the entire computation occupies constant memory.

14 Measurement and asymptotic analysis

A useful timing experiment should:

- vary input size over a substantial range;
- state how inputs and cases are generated;
- separate setup from the operation being measured where appropriate;
- repeat trials and report variability;
- keep environments comparable;
- label axes and units;
- consider logarithmic axes for wide ranges; and
- explain deviations from the analytical model.

A straight line on a log-log plot can suggest polynomial growth, with its slope suggesting an exponent. It cannot prove an asymptotic bound: measurements cover finitely many inputs, and different functions can look similar over a restricted range.

15 Connections

- Dynamic arrays in the next note use a geometric sum for amortised analysis.
- Merge sort, quicksort, and selection lead to recurrences and expected-cost calculations.
- Graph algorithms commonly require bounds in both $|V|$ and $|E|$.
- Succinct structures and string algorithms make word size and bit complexity explicit.
- Heuristic methods require empirical comparisons because asymptotic cost alone says nothing about solution quality.

16 Common mistakes

- Treating $f = O(g)$ as a symmetric algebraic equality or an exact runtime.
- Saying "Big O means worst case." Case selection and function bounds are different.
- Giving a loose upper bound when a tight bound is available.
- Measuring numeric magnitude instead of input representation length.
- Multiplying loop limits without checking dependencies.
- Confusing average-case, expected randomized, and amortised analysis.
- Omitting recursion-stack or auxiliary-space costs.
- Assuming the asymptotically better algorithm wins for every practical n.
- Presenting a fitted timing curve as proof.

17 Self-check

1. Prove $7n + 4 \in \Theta(n)$ with explicit constants.
2. Is $n \in O(n^2)$? Is $n^2 \in O(n)$? Explain from the definition.
3. Why is writing $f \in O(g)$ conceptually clearer than $f = O(g)$?
4. Distinguish worst-case, average-case, randomized expected, and amortised cost.
5. Order $(\log n)^2$, $n \log n$, $n^2 + 100n$, n^3, and 2^n.
6. What is the bit length of a positive integer x, asymptotically?
7. Why does every level of $T(n) = 3T(n/3) + n$ contribute $\Theta(n)$ non-recursive work?
8. Give an algorithm whose iterative and recursive forms have equal time but different space costs.

18 Revision summary

- Define input size, case, and cost model before counting.
- Big O is an eventual upper bound, Omega a lower bound, and Theta a tight bound.
- Little-o and little-ω express strict asymptotic separation.
- Common sums translate executions into growth functions.
- Constants and lower-order terms vanish asymptotically but matter for finite workloads.
- Expected and amortised claims average over fundamentally different things.
- Time and auxiliary space require separate analyses.
- Recurrence trees expose how recursive work is distributed across levels.
- Experiments test implementations; they do not prove asymptotic results.

19 Implementations and hands-on exploration

- Python `timeit` calibrates repetitions for short Python statements and functions. Keep input construction outside the timed statement when the algorithm alone is the object of study.
- Google Benchmark provides parameterised C++ microbenchmarks, fixtures, counters, and repeated runs. Compiler optimisation and cache state can still make a correct benchmark answer a narrower question than intended.
- Criterion.rs performs warm-up, sampling, statistical analysis, and regression comparisons for Rust. Statistical confidence concerns measured runs, not an asymptotic proof or a guarantee on every input.
- OpenJDK JMH provides a harness for Java and other JVM microbenchmarks, with samples for avoiding dead-code elimination and warm-up errors. Run its generated benchmark project rather than treating an ordinary method timer as equivalent.

19.1 Small experiments

1. Instrument the triangular nested loop. After outer iteration i, assert the exact invariant `count == i*(i+1)/2`; for $n = 2^k$, also print `count/n^2` and `count/n`. Which ratio approaches a constant?
2. For powers of two, implement dummy work whose counters execute exactly $50n \log_2 n$ and n^2 operations. Assert both counts before timing the loops, then compare the analytical and measured crossover points and explain any difference.

20 Sources and further study

See the chapters on growth of functions and analysing algorithms in Cormen et al. (2022), and the introductory algorithm-analysis discussion in Kleinberg and Tardos (2006).

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.

Kleinberg, Jon, and Éva Tardos. 2006. *Algorithm Design*. Pearson.