

Course Algorithms and Data Structures · Lecture 3

Time Complexity: Big-O, Best and Worst Case

Textbook: A. Levitin, Sections 2.1–2.2

Why not just use a stopwatch?

The running time in seconds depends on things that have nothing to do with the algorithm:

- how fast the computer is
- the programming language and compiler
- what else is running at the same moment

So we **count the basic operations**, and see how the count **grows** when the input grows.

Input size and basic operation

Input size n

how big the problem is: the number of elements in an array, characters in a text, names in a phone book

Basic operation

the one that takes the most time in total, usually in the innermost loop: a comparison, an addition

C(n) = the number of basic operations for input size n

Example 1: sum of an array

```
ALGORITHM Sum(A[0..n − 1])
    s ← 0
    for i ← 0 to n − 1 do
        s ← s + A[i]
    return s
```

$$C(n) = n$$

Twice as many elements, twice as much work.

Example 2: are there two equal elements?

```
ALGORITHM UniqueElements(A[0..n − 1])
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

$$C(n) = (n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$$

Two nested loops: twice as many elements, four times as much work.

Example 3: binary search

Every comparison throws away half of the remaining elements:

$$C(n) \approx \log_2 n$$

10

comparisons for n = 1,000

20

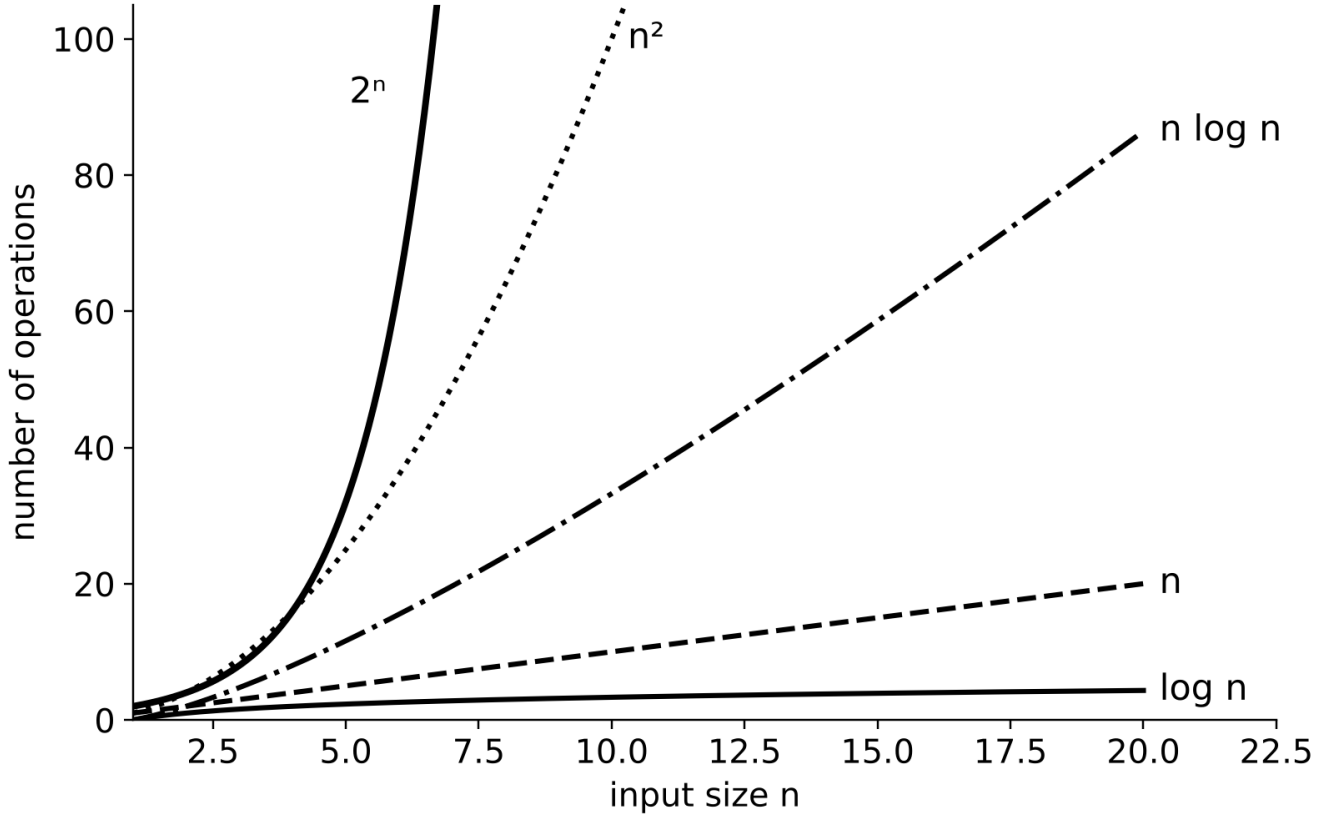
comparisons for n = 1,000,000

Order of growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
10	3.3	10	33	100	1,024
100	6.6	100	664	10,000	$1.3 \cdot 10^{30}$
1,000	10	1,000	9,966	10^6	$1.1 \cdot 10^{301}$
10^6	20	10^6	$2 \cdot 10^7$	10^{12}	—

For small n every algorithm is fast. The difference shows up when n is large.

How the common functions grow



What happens when n doubles?

Order	When n doubles, the work…
log n	grows by one step
n	doubles
n^2	grows 4 times
2^n	is squared

Big-O notation

$C(n) \in O(g(n))$ means: for large n, C(n) grows **no faster than** g(n), up to a constant factor.

1. Keep only the fastest-growing term.
2. Drop the constant factor in front of it.

$$\mathbf{3n^2 + 5n + 2 \in O(n^2)\ n - 1 \in O(n)}$$

The formal definition

$t(n) \in O(g(n))$ if there are a constant $c > 0$ and a number n_0 such that $t(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Example: $3n^2 + 5n + 2 \leq 4n^2$ for all $n \geq 6$, so $c = 4$ and $n_0 = 6$.

Levitin also uses Ω (“grows at least as fast”) and Θ (“grows exactly as fast”). For now Big-O is enough.

The most common classes, from fast to slow

Big-O	Name	Example
$O(1)$	constant	access A[i]
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	sum, sequential search
$O(n \log n)$	n log n	merge sort (week 11)
$O(n^2)$	quadratic	two nested loops, simple sorts
$O(2^n)$	exponential	trying all subsets

Quick rules for code

- One loop over n elements: **O(n)**
- Two **nested** loops over n elements: **O(n²)**
- A loop that halves n each time: **O(log n)**
- Two parts one after another: take the bigger one
- Steps that don’t depend on n: **O(1)**

Best, worst and average case

Sequential search in an array of n elements:

Case	When	Comparisons
Best	K is the first element	1
Worst	K is last or not there	n
Average	K is there, any position equally likely	(n + 1)/2

We usually use the **worst case**: it is a guarantee for every input.

Common mistakes

- Measuring in seconds instead of counting operations
- Keeping constants: O(2n) and O(n + 5) are just O(n)
- “Two loops means $O(n^2)$”: only if they are **nested**
- Judging an algorithm by its best case

Summary

- We measure an algorithm by counting basic operations as a function of the input size n.
- Only the order of growth matters; Big-O keeps the fastest-growing term and drops constants.
- From fast to slow: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$.
- The worst case gives a guarantee, so it is the one we usually use.

Review questions

1. Why do we count operations instead of measuring the running time in seconds?
2. What is the Big-O of $5n^2 + 3n + 10$? Of $100n + 7$? Explain the rules you used.
3. What are the best and the worst case of sequential search in an array of 100 elements? How many comparisons does each need?

Reading: Levitin, Sections 2.1–2.2; Bhargava, Grokking Algorithms, Chapter 1.