



Complexity and Big-O

Algorithms | Counting operations, O / Omega / Theta, time vs space, best / average / worst case, amortised cost, recurrences and the master theorem

#Algorithms #BigO #Complexity #MasterTheorem

What complexity measures

Planning a road trip		Analysing an algorithm
Distance to drive	≈	Input size n
Kilometres per hour of your car	≈	Speed of the machine (constant factor, ignored)
Route that doubles length when the map doubles	≈	O(n): linear growth
Route that visits every pair of towns	≈	O(n²): quadratic growth
Worst traffic jam on the way	≈	Worst-case running time
Fuel tank size you need	≈	Space complexity

WHERE THE ANALOGY BREAKS A faster car cuts every trip by the same factor; a better algorithm changes how the trip grows with distance, which no faster car can match for large n.

Before Complexity analysis, can you explain...

	Not yet	Partly	Yes
MUST KNOW Exponents and logarithms log2(1 024) = 10, 2¹⁰ ≈ 10³; Learn: maths: algebra and notation refresher	☐	☐	☐
MUST KNOW Loops and recursion in Python Learn: programming: Python for AI	☐	☐	☐
HELPFUL Sums of series 1 + 2 + ... + n = n(n+1)/2	☐	☐	☐
ADVANCED Limits f/g → constant means f = Theta(g)	☐	☐	☐

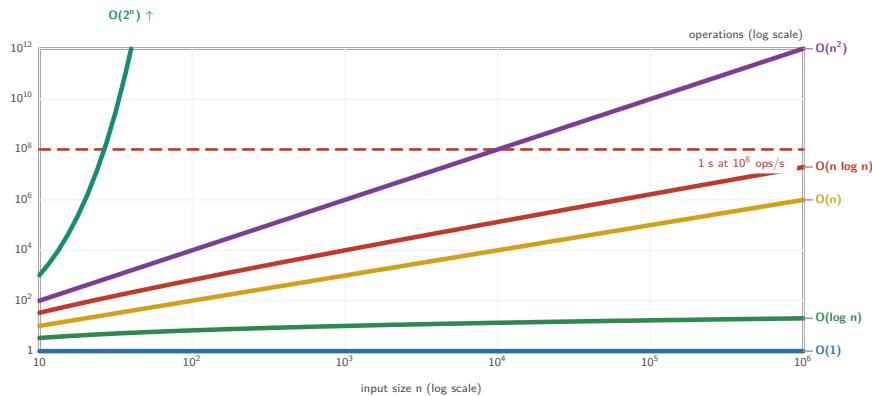
Complexity counts **elementary operations** (or memory cells) as a function of the **input size n**, then keeps only the fastest-growing term and drops constants: 3n² + 20n + 7 is **Theta(n²)**. The answer predicts how the cost **scales** (double n, what happens?), not the run time in seconds.

Notation	Definition	Reads as
O	$f(n) = O(g(n)) \iff \exists c > 0, n_0 : f(n) \leq c g(n) \forall n \geq n_0$	grows no faster than g (upper bound)
Omega	$f(n) = \Omega(g(n)) \iff \exists c > 0, n_0 : f(n) \geq c g(n) \forall n \geq n_0$	grows at least as fast as g (lower bound)
Theta	$f = \Theta(g) \iff f = O(g) \wedge f = \Omega(g)$	same growth rate (tight)
o	$\lim_{n \rightarrow \infty} f(n)/g(n) = 0$	strictly slower than g

where f(n) = cost of the algorithm on inputs of size n (operations or memory cells); g(n) = reference growth function (n, n log n, n²...) (same as f); c = positive constant factor (none); n₀ = size from which the inequality must hold (elements)

⚠ WARNING
Big-O is an **upper bound**, so binary search is O(n) too (true but useless). In interviews and papers 'O' is used loosely for the tight bound; when precision matters say Theta.

⚙️ Growth classes in numbers



class · example	n = 10	n = 1000	n = 10^6
$O(1)$ array index, hash lookup	1 10 ns	1 10 ns	1 10 ns
$O(\log n)$ binary search	3.32 33.2 ns	9.97 99.7 ns	20 199 ns
$O(n)$ linear scan, max of a list	10 100 ns	1000 10 μ s	10^6 10 ms
$O(n \log n)$ merge sort, heap sort	33 332 ns	9966 99.7 μ s	2.0×10^7 199 ms
$O(n^2)$ bubble sort, all pairs	100 1 μ s	10^6 10 ms	10^{12} 2.78 h
$O(2^n)$ all subsets	1024 10.2 μ s	1.1×10^{301} > age of universe	9.9×10^{301029} > age of universe

Counts and run times at 10^8 simple operations/s (approx. compiled code; Python 10-100× slower)

Class	Typical algorithm	n solvable in about 1 s
$O(1)$	array index, hash lookup (average)	any
$O(\log n)$	binary search, balanced-tree lookup	any
$O(n)$	one scan, counting, two pointers	approx. 10^8
$O(n \log n)$	merge sort, heap sort, Dijkstra with a heap	approx. $10^6 - 10^7$
$O(n^2)$	all pairs, insertion sort, naive DP on two strings	approx. 10^4
$O(n^3)$	Floyd-Warshall, naive matrix multiply	approx. 500
$O(2^n)$	all subsets, brute-force SAT	approx. 25 - 27
$O(n!)$	all permutations (brute-force TSP)	approx. 11

💡 TIP

Read the constraint first in a coding problem: $n \leq 20$ invites $O(2^n)$, $n \leq 5\,000$ allows $O(n^2)$, $n \leq 2 \cdot 10^5$ demands $O(n \log n)$ or better.

Count the loops

```
def pairs(a):
    # 0(n^2) time, 0(1) space
    c = 0
    for i in range(len(a)):
        # n
        for j in range(i + 1, len(a)):
            # n-1, n-2, ... 0
            c += a[i] + a[j]
    return c
    # n(n-1)/2 additions

def halve(n):
    # 0(log n): n halves each turn
    steps = 0
    while n > 1:
        n //= 2
        steps += 1
    return steps

def all_subsets(a):
    # 0(n * 2^n) time and output
    out = [[]]
    for x in a:
        out += [s + [x] for s in out]
    return out
```

🕒 Which case, which resource

Best case	Average case	Worst case
cheapest input of size n	expected cost over an input distribution	most expensive input of size n
insertion sort on sorted data: $\Theta(n)$	insertion sort on random data: $\Theta(n^2)$	insertion sort on reversed data: $\Theta(n^2)$
quicksort, perfect pivots: $\Theta(n \log n)$	quicksort, random pivot: $\Theta(n \log n)$ expected	quicksort, sorted input + first-element pivot: $\Theta(n^2)$
hash lookup, no collision: $\Theta(1)$	hash lookup, load factor α : $\Theta(1 + \alpha)$	hash lookup, all keys in one bucket: $\Theta(n)$

Expected vs average — Expected cost averages over the algorithm's own random choices (any input); average-case cost averages over an assumed input distribution.

Auxiliary space — Memory used beyond the input: merge sort $O(n)$, heap sort $O(1)$, recursive DFS $O(\text{depth})$ for the call stack.

In-place — Uses $O(1)$ (sometimes $O(\log n)$) extra memory and rewrites the input; quicksort and heap sort are in place, merge sort on arrays is not.

Output-sensitive — Cost depends on the output size k too: listing all subsets is $\Theta(n \cdot 2^n)$ because the output has that size.

- **Time-space trade-off** memoisation spends $O(n)$ memory to turn exponential recursion into $O(n)$ time; a hash set spends $O(n)$ memory to make duplicate checks $O(1)$.
- **Word RAM model** arithmetic on $O(\log n)$ -bit numbers is $O(1)$; big integers, strings and float accumulations are not free (string concat in a loop is $O(n^2)$).
- **Python traps** `x in list` is $O(n)$, `x in set` $O(1)$ average; `list.pop(0)` is $O(n)$, use `collections.deque`; slicing `a[i:j]` copies $O(j-i)$.

Amortised analysis

PROBLEM A dynamic array starts with capacity 1 and doubles when full. What is the total cost of $n = 16$ appends, counting 1 per write and 1 per element copied?

GIVEN

- Writes: one per append, so 16
- Resizes happen when size reaches 1, 2, 4, 8 (before appending the next element)

1 Count the copies at each resize

$1 + 2 + 4 + 8 = 15$

Why: each resize copies every current element into the new block

2 Add the writes

$15 + 16 = 31$

Why: every append writes its element once

3 Generalise

$\text{copies} \leq 1 + 2 + \dots + n/2 < n$, total $< 3n$

Why: a geometric series is dominated by its last term

ANSWER 31 operations, below $3n = 48$: amortised $O(1)$ per append

CHECK Growing by a constant (+10 slots) instead of a factor gives $n^2/20$ copies: amortised $O(n)$.

Method	Idea	Example
Aggregate	bound the total over n operations, divide by n	n appends cost $< 3n$
Accounting	charge each cheap op extra credit that pays later work	append pays 3: 1 write + 2 saved for future copies
Potential	$\Phi(\text{state}) \geq 0$; amortised = actual + change in Φ	$\Phi = 2 \cdot \text{size} - \text{capacity}$ for a doubling array

NOTE

Amortised $O(1)$ is a guarantee over any sequence; it is not an average over random inputs. A single append can still take $\Theta(n)$, which matters for real-time systems (latency spikes).

Recurrences and the master theorem

Critical exponent of a divide-and-conquer recurrence

FORMULA $E = \log(a) / \log(b)$

WHERE

- E = critical exponent $\log_b(a)$: the leaves of the recursion tree number n^E (none)
- a = number of sub-problems per call (count)
- b = factor by which the input shrinks per call (count)

WORKED

$a = 7$
 $b = 2$
 $E = \log(7) / \log(2)$
 $E = 2.807$

RESULT $E = 2.807$

READ IT Strassen: $T(n) = 7T(n/2) + \Theta(n^2)$; $E \approx 2.81 > 2$, so the leaves dominate and $T = \Theta(n^{2.81})$, better than the naive n^3 .

MISTAKES

- Compare $f(n)$ with n^E polynomially: $n \log n$ vs n is not a polynomial gap, so case 3 does not apply to $T = 2T(n/2) + n \log n$ (answer $\Theta(n \log^2 n)$).
- Sub-problems must have equal size n/b ; $T(n) = T(n/3) + T(2n/3) + n$ needs a recursion tree ($\Theta(n \log n)$).

Case	Condition	Result
1 (leaves dominate)	$f(n) = O(n^{E-\epsilon})$	$T(n) = \Theta(n^E)$
2 (balanced)	$f(n) = \Theta(n^E \log^k n)$	$T(n) = \Theta(n^E \log^{k+1} n)$
3 (root dominates)	$f(n) = \Omega(n^{E+\epsilon})$, $af(n/b) \leq cf(n)$	$T(n) = \Theta(f(n))$

where $T(n)$ = cost on an input of size n , with $T(n) = a T(n/b) + f(n)$ (operations); $f(n)$ = work done outside the recursive calls (split + combine) (operations); E = critical exponent $\log_b a$ (none); ϵ = any positive constant (a polynomial gap) (none); k = power of the log factor, $k \geq 0$ (none); c = constant below 1 (regularity condition) (none)

Algorithm	Recurrence	Case	Cost
Binary search	$T(n/2) + 1$	2, $k = 0$	$\Theta(\log n)$
Merge sort	$2T(n/2) + n$	2, $k = 0$	$\Theta(n \log n)$
Karatsuba multiply	$3T(n/2) + n$	1, $E \approx 1.58$	$\Theta(n^{1.58})$
Tree traversal	$2T(n/2) + 1$	1, $E = 1$	$\Theta(n)$
Quickselect (good pivots)	$T(n/2) + n$	3	$\Theta(n)$
Naive Fibonacci	$T(n-1) + T(n-2) + 1$	not master form	$\Theta(\phi^n)$, $\phi \approx 1.618$

⚠ Traps and recap

YOU MAY HEAR

" $O(n)$ is always faster than $O(n^2)$ "

BUSTED

✓ ACTUALLY

Only beyond a crossover size

100n beats $n^2/2$ only for $n > 200$; that is why library sorts switch to insertion sort for runs of about 16-64 elements.

YOU MAY HEAR

"Hash tables are $O(1)$ "

BUSTED

✓ ACTUALLY

$O(1)$ on average, $O(n)$ in the worst case

Adversarial keys can all collide; Python randomises string hashing per process to make that hard.

YOU MAY HEAR

"Two nested loops always mean $O(n^2)$ "

BUSTED

✓ ACTUALLY

Count total iterations, not loop depth

A sliding window moves its inner pointer at most n times in total: $O(n)$. A loop that halves its bound inside a linear loop gives $O(n \log n)$.

YOU MAY HEAR

"Recursion is free in memory"

BUSTED

✓ ACTUALLY

Each pending call holds a stack frame

Recursive DFS on a $10^{6 \text{ nodes}}$ path uses $O(n)$ stack; Python's default recursion limit is 1 000.

Tick a box only when you could explain the point to someone else



Count operations as a function of n

keep the dominant term, drop constants



Amortised $\neq$ average

a bound on the total cost of any sequence



Constraints tell the target

$n \leq 20 \rightarrow 2^n$, $n \leq 10^4 \rightarrow n^2$, $n \leq 10^6 \rightarrow n \log n$



State the case

best, average (input distribution), expected (random choices), worst



Master theorem

compare $f(n)$ with $n^{\lceil \log_b a \rceil}$: leaves, balanced or root dominates

⊕ INTERVIEW QUESTION

Q: 'What is the complexity of $s = s + c$ in a loop over n characters in Python?' A: $\Theta(n^2)$ in the worst case, since each concatenation may copy the whole string; build a list and `''.join` it for $\Theta(n)$.

Q1 Simplify $5n^3 + 100n^2 \log n + 7$ in Theta notation.

A $\Theta(n^3)$: the cubic term dominates for large n .

Q2 Is binary search $O(n)$?

A Yes, Big-O is only an upper bound; its tight bound is $\Theta(\log n)$.

Q3 Amortised cost of append on a doubling dynamic array?

A $O(1)$: n appends copy fewer than n elements in total.

Q4 Master theorem: $T(n) = 4T(n/2) + n$?

A $E = \log_2 4 = 2 > 1$, case 1 $\rightarrow \Theta(n^2)$.

Q5 Master theorem: $T(n) = 2T(n/2) + n^2$?

A $E = 1$, $f = n^2$ dominates polynomially and regularity holds, case 3 $\rightarrow \Theta(n^2)$.

Q6 Expected vs average-case complexity?

A Expected averages over the algorithm's random choices for every input; average-case averages over a distribution of inputs.

Q7 Space complexity of recursive merge sort on an array?

A $\Theta(n)$ for the merge buffer plus $\Theta(\log n)$ of call stack.

Q8 Largest n for an $O(n^2)$ algorithm in about one second of compiled code?

A Approx. 10^4 (10^8 operations); in pure Python roughly 3 000.

Q9 Why is `list.pop(0)` slow in Python?

A It shifts the $n-1$ remaining elements: $\Theta(n)$. `deque.popleft()` is $O(1)$.