

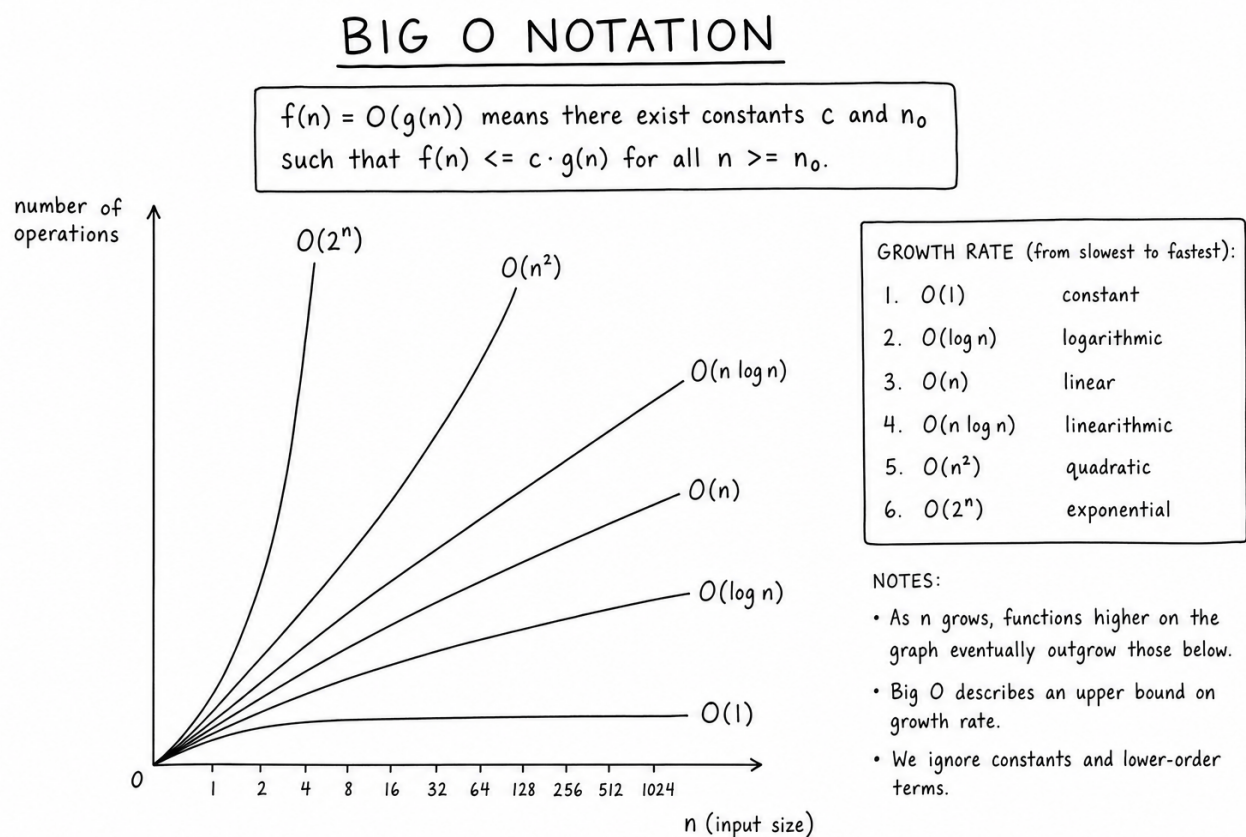
Big-O Notation

[Read this note online](#)

Big-O notation describes how the resource requirements of an algorithm, usually time or space, scale with the size of the input. It's the universal language for talking about algorithm efficiency. Whether you're choosing between sorting algorithms, estimating database query cost, or evaluating whether your code can handle production-scale data, you're reasoning in Big-O terms.

The notation strips away constant factors and lower-order terms, focusing on how an algorithm scales asymptotically. $O(n^2)$ means quadratic growth: doubling the input quadruples the work. $O(n \log n)$ means slightly worse than linear. $O(2^n)$ means brute-force exponential blowup that becomes intractable past a few dozen items. Knowing where each common algorithm sits on this scale is half of algorithm selection.

Big-O gives an asymptotic upper bound. The input model and the operation being counted must be specified before complexity labels are compared.



Big-O notation growth rate curves comparison modern textbook illustration

The Formal Definition

$f(n) = O(g(n))$ means there exist positive constants c and n_0 such that:

In words: $f(n)$ grows no faster than $g(n)$ up to a constant factor, for all sufficiently large n . This is an upper bound on growth rate. Constants and lower-order terms drop out because they become negligible as n grows.

Big-O is purely about asymptotic behavior. $3n^2 + 100n + 9999 = O(n^2)$, the constant 9999 and the linear term $100n$ become tiny compared to $3n^2$ once n is large enough. The 3 also drops because constants don't matter in Big-O.

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$

Common Complexity Classes

From fastest to slowest:

$O(1)$: constant. Hash table lookup, array index access. The runtime doesn't depend on input size. $O(\log n)$: logarithmic. Binary search, balanced BST operations. Doubling the input adds only one step. $O(n)$: linear. Single-pass loops, linear search. Time scales proportionally with input size. $O(n \log n)$: linearithmic. Merge sort, heapsort, quicksort (average case). The provably optimal class for comparison-based sorting. $O(n^2)$: quadratic. Bubble sort, naive pair comparisons. Doubling input quadruples work. $O(n^3)$: cubic. Naive matrix multiplication, Floyd-Warshall. Doubling input multiplies work by 8. $O(2^n)$: exponential. Brute-force subset search, naive recursive Fibonacci. Becomes intractable past $n \approx 30$. $O(n!)$: factorial. Brute-force traveling salesman, all permutations. Intractable past $n \approx 12$.

Worked Example: Linear vs Quadratic

Consider two ways to find duplicates in a list of n items.

Naive (quadratic): compare every pair. $\binom{n}{2} \approx n^2/2$ comparisons. $O(n^2)$.

Hash-based (linear): insert each item into a set; if it's already there, you've found a duplicate. n insertions, each $O(1)$ on average. $O(n)$.

For $n = 10,000$, the quadratic version does ~50 million comparisons. The linear version does ~10,000 set operations. Same problem, different complexity class, drastically different runtime. This is the kind of choice Big-O analysis informs.

```
for i in range(n): for j in range(i+1, n): if items[i] == items[j]: return True
```

seen = set() for x in items: if x in seen: return True seen.add(x)

Big-O vs Big-Theta vs Big-Omega

Three asymptotic notations get used together:

In practice, programmers say "Big-O" to mean tight bound (Θ) most of the time, even though that's technically imprecise. When precision matters (e.g., proving lower bounds for sorting), use Ω and Θ explicitly.

$O(g(n))$: upper bound. Function grows no faster than $g(n)$. $\Omega(g(n))$: lower bound. Function grows at least as fast as $g(n)$. $\Theta(g(n))$: tight bound. Function grows exactly at the rate of $g(n)$ (both upper and lower bound match).

Plotting the Complexity Classes

Different complexity classes look almost identical for small inputs but diverge dramatically as input grows. On a logarithmic y-axis, the differences become clear: each class shows up as a line with a distinct slope. Linear and constant complexities sit at the bottom; exponential and factorial blow up off the top.

The implication: for small inputs ($n < 100$), almost any algorithm works. For medium inputs ($n < 10^6$), $O(n \log n)$ becomes essential. For large inputs ($n > 10^9$), even $O(n)$ might be too slow, and you need sublinear or distributed algorithms.

Time Complexity vs Space Complexity

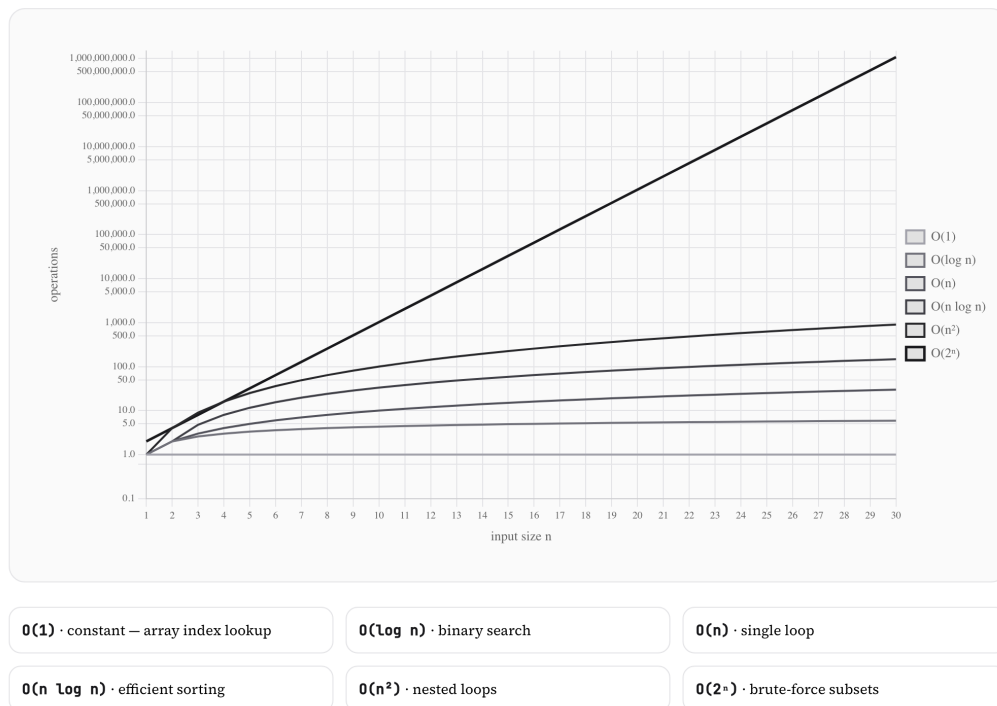
Big-O describes both time and space requirements. **Time complexity** measures how runtime scales with input. **Space complexity** measures memory consumption.

An $O(n)$ time algorithm with $O(1)$ space is usually preferred over an $O(n)$ time algorithm with $O(n)$ space (same speed, less memory). But sometimes you trade space for time: hash tables use $O(n)$ extra space to enable $O(1)$ lookups.

Always specify which complexity you're discussing. "This algorithm is $O(n^2)$ " is ambiguous without context, it could mean time, space, or both.

How fast do common complexities grow?

Operations counted versus input size n . The y-axis is logarithmic — every grid line is 10× the previous one.



Big-O complexity comparison chart on logarithmic scale

Best, Average, and Worst Case

The same algorithm can have different complexities depending on the input. Quicksort is $O(n^2)$ worst case (when the pivot consistently splits poorly), $O(n \log n)$ average case (random or shuffled input), and $O(n \log n)$ best case (balanced splits).

When you say "quicksort is $O(n \log n)$," you usually mean average case. Production code often cares about worst case (denial-of-service attacks exploit worst-case inputs). Algorithm analysis courses cover all three.

Hash tables are another classic example: $O(1)$ average case for lookups, $O(n)$ worst case if all keys hash to the same bucket (essentially unavoidable with good hash functions but possible with adversarial input).

Amortized Analysis

Some operations are usually fast but occasionally slow. The amortized cost averages the slow ones across many fast ones. Dynamic array (vector) appending is the classic example, most appends are $O(1)$, but occasionally the array doubles in size, requiring $O(n)$. Amortized over many appends, the cost is $O(1)$.

Amortized analysis matters when picking data structures. A dynamic array's "amortized $O(1)$ append" is what makes it usable in practice despite occasional reallocations. Similar reasoning applies to hash table resizing, splay tree access, and many advanced data structures.

Reference Card: Common Algorithms by Complexity

A quick mental reference for the algorithms you'll meet most often:

$O(1)$: hash table operations, array indexing, stack/queue push/pop, doubly linked list insert/delete given pointer. $O(\log n)$: binary search, balanced BST operations, heap insert/delete-min, exponentiation by squaring. $O(n)$: array traversal, linear search, building a heap, single graph traversal step. $O(n \log n)$: merge sort, heapsort, quicksort (average), FFT, kd-tree construction. $O(n^2)$: bubble sort, insertion sort, selection sort, naive nearest-neighbor brute force. $O(n^3)$: naive matrix multiply, Floyd-Warshall all-pairs shortest paths, naive 3D point cloud algorithms. $O(2^n)$: brute-force subset enumeration, naive recursive Fibonacci, NP-hard exact algorithms. $O(n!)$: brute-force traveling salesman, all-permutation enumeration.

Algorithms by Big-O class

A reference card for the standard textbook algorithms. Knowing where each one sits on the complexity scale is half of algorithm selection.

COMPLEXITY	NAME	EXAMPLES	VERDICT
$O(1)$	Constant	Hash table lookup, array index	Best possible
$O(\log n)$	Logarithmic	Binary search, balanced BST operations	Excellent
$O(n)$	Linear	Array traversal, linear search	Good
$O(n \log n)$	Linearithmic	Merge sort, quicksort (avg), heapsort	Standard for sorting
$O(n^2)$	Quadratic	Bubble sort, insertion sort, naïve pair check	Avoid for large n
$O(n^3)$	Cubic	Naïve matrix multiply, Floyd-Warshall	Only small n
$O(2^n)$	Exponential	Brute-force subset enumeration, naïve fib	Intractable past $n \approx 30$
$O(n!)$	Factorial	Brute-force traveling salesman, all permutations	Intractable past $n \approx 12$

Rule of thumb: if your input could plausibly hit a million, you need $O(n \log n)$ or better. If it's bounded under a few thousand, $O(n^2)$ is usually fine. Above $O(n^2)$, every additional polynomial degree imposes a hard ceiling on input size.

Common algorithms by Big-O complexity class reference table

Why Constants Matter in Practice (Even Though Big-O Ignores Them)

Big-O ignores constants for theoretical tidiness, but constants matter enormously in practice. An $O(n \log n)$ algorithm with constant 1000 might be slower than an $O(n^2)$ algorithm with constant 1 for small inputs. The crossover happens at the n where $1000 \cdot n \log n = n^2$, which is around $n \approx 10,000$.

This is why insertion sort beats quicksort for very small arrays (production sort implementations switch to insertion sort below $n \approx 10$ or so). It's also why constant-factor improvements ("optimization") matter for hot code paths even when the asymptotic complexity is unchanged.

Common Mistakes with Big-O

Treating Big-O as exact runtime. Big-O describes asymptotic behavior; actual runtime depends on constants, hardware, and inputs. $O(n)$ doesn't mean "linear in milliseconds", it means linear in operation count, asymptotically. **Ignoring lower-order terms when they matter.** For small

inputs, lower-order terms dominate. Big-O is a long-run prediction, not a short-run guarantee. **Confusing time and space complexity.** An algorithm can be fast and memory-hungry, or slow and memory-efficient. Always specify which. **Conflating average and worst case.** "Quicksort is $O(n \log n)$ " is the average case. The worst case is $O(n^2)$, which can matter for adversarial inputs. **Reading nested loops as automatic $O(n^2)$.** Two nested loops over different ranges might be $O(n + m)$, not $O(nm)$. Read the actual iteration counts. **Misjudging recursive complexity.** Recursive algorithms need recurrence-relation analysis (Master theorem, recursion tree). Eyeballing the code often produces wrong complexity estimates.

A Brief History of Big-O Notation

Big-O notation predates computer science. German mathematician Paul Bachmann introduced it in 1894 in his treatise on number theory, and Edmund Landau popularized it in 1909. The original use was in analytic number theory for describing the asymptotic growth of error terms in approximations.

Donald Knuth introduced Big-O to computer science in his *The Art of Computer Programming* series, starting in the 1960s and 1970s. Knuth's adoption made Big-O the standard language for algorithm analysis, replacing earlier ad-hoc notations.

Today every undergraduate computer science curriculum teaches Big-O analysis. Coding interviews universally test it. Algorithm research papers use it as the basic vocabulary for comparing performance. After 130+ years, Big-O remains the dominant framework for asymptotic analysis across mathematics and computer science.

The Master Theorem

For divide-and-conquer recurrences of the form $T(n) = aT(n/b) + f(n)$, the master theorem gives the asymptotic complexity directly:

The master theorem covers most divide-and-conquer recurrences (merge sort, quicksort, binary search, fast Fourier transform). For recurrences that don't fit, recursion-tree analysis or substitution proofs handle the general case.

If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$: $T(n) = \Theta(n^{\log_b a})$. If $f(n) = \Theta(n^{\log_b a})$: $T(n) = \Theta(n^{\log_b a} \log n)$. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and the regularity condition holds: $T(n) = \Theta(f(n))$.

P vs NP and Computational Complexity Classes

Beyond Big-O, computational complexity theory groups problems into classes by inherent difficulty. **P** is the class of problems solvable in polynomial time. **NP** is the class where solutions can

be verified in polynomial time. The famous P vs NP question asks whether every problem with a quickly-checkable solution can also be quickly solved.

NP-complete problems (traveling salesman, satisfiability, graph coloring) are the hardest in NP, if any NP-complete problem has a polynomial-time algorithm, all NP problems do. Most computer scientists believe $P \neq NP$, but no proof exists. The question remains one of the seven Millennium Prize Problems with a \$1 million reward for resolution.

Worked Example: Recursive Complexity Analysis

Analyze the complexity of merge sort. The recurrence is $T(n) = 2T(n/2) + O(n)$, two recursive calls on halves plus a linear merge step.

By the master theorem with $a = 2, b = 2, f(n) = O(n)$: $\log_b a = 1$, and $f(n) = \Theta(n^1)$ matches the boundary case. So $T(n) = \Theta(n \log n)$. Merge sort runs in $O(n \log n)$ time.

The same analysis applies to many divide-and-conquer algorithms: binary search ($T(n) = T(n/2) + O(1)$, giving $O(\log n)$), Karatsuba multiplication ($T(n) = 3T(n/2) + O(n)$, giving $O(n^{\log_2 3})$), and the FFT ($T(n) = 2T(n/2) + O(n)$, giving $O(n \log n)$).

When Big-O Is Misleading

Big-O ignores constants and lower-order terms. For real-world inputs, those can dominate. Three classic cases where Big-O misleads:

Big-O is the right tool for choosing between fundamentally different algorithm classes. For optimizing within a class, you need profiling, benchmarks, and an understanding of the underlying hardware.

Galactic algorithms: $O(n \log \log n)$ integer multiplication is asymptotically faster than $O(n \log n)$ FFT-based multiplication, but the constants are so large that no practical input triggers the crossover. **Cache effects:** two $O(n^2)$ algorithms can have wildly different real-world speeds depending on memory access patterns. Cache-friendly traversal beats cache-hostile traversal by $10\times$ - $100\times$. **Small inputs:** for $n < 30$, insertion sort beats quicksort despite the worse asymptotic complexity. Production sort routines hybridize for this reason.

Worked Example: Analyzing Two-Sum

The classic interview problem: given an array and a target, find two indices whose values sum to the target. Two solutions:

Brute force: nested loop over all pairs. Each pair is checked in $O(1)$; there are $\binom{n}{2} \approx n^2/2$ pairs. Total: $O(n^2)$ time, $O(1)$ space.

Hash table: single pass through the array. For each element x , check whether target $-x$ is already in a hash table; if so, found the pair. Otherwise add x to the hash table. Each iteration is $O(1)$ average; total iterations: n . Total: $O(n)$ time, $O(n)$ space.

The hash-table version trades extra memory for dramatically better runtime. For $n = 10^6$, brute force does 5×10^{11} operations (hours). The hash version does 10^6 operations (milliseconds). This is the textbook lesson on space-time trade-offs in algorithm design.

Common Cheat Sheet: Big-O for Standard Data Structures

Most data structures have characteristic complexities for their main operations. A quick reference:

Knowing this table by heart lets you pick the right data structure for any problem in seconds. It's the practical basis for algorithm selection.

Array: indexed access $O(1)$; search $O(n)$; insert/delete at end $O(1)$ amortized; insert/delete in middle $O(n)$. **Linked list:** indexed access $O(n)$; search $O(n)$; insert/delete with pointer $O(1)$. **Hash table:** insert, lookup, delete all $O(1)$ average, $O(n)$ worst case. **Balanced BST (red-black, AVL):** insert, lookup, delete all $O(\log n)$ worst case. **Heap:** peek minimum $O(1)$; insert and extract-minimum $O(\log n)$. **Stack/queue:** push, pop, peek all $O(1)$. **Trie:** insert and search of a string of length k is $O(k)$, independent of total stored strings.

How Different Languages Express Big-O

The asymptotic complexity of an algorithm is independent of programming language, but the constant factors aren't. Python is typically 10-50× slower than C for the same algorithm, but a Python implementation of $O(n \log n)$ sorting still beats a C implementation of $O(n^2)$ sorting for large inputs.

This is why Big-O is the right tool for choosing algorithms across languages. A million-element sort in well-implemented Python merge sort takes seconds; a million-element sort in C bubble sort takes hours. The asymptotic class dominates language-specific constants for any non-trivial input size.

Big-O in Database Query Performance

Database queries have characteristic complexities. A full table scan is $O(n)$. An indexed lookup on a B-tree is $O(\log n)$. A nested-loop join is $O(n \cdot m)$. A hash join is $O(n + m)$ average case.

Query optimizers in databases (PostgreSQL, MySQL, BigQuery) compute estimated costs for dif-

ferent execution plans using Big-O reasoning combined with row-count statistics. Understanding the asymptotic complexity of indexes, joins, and aggregations is what separates working DBAs from struggling ones. EXPLAIN plans show you exactly which complexity class each operation uses.

The related [OpenStax mathematics textbooks](#) provide additional explanations and exercises for the mathematical definitions used here.

Counting a Nested Loop

A loop runs i times for each i from 1 through n . Its total iterations are $1 + 2 + \dots + n = n(n + 1)/2$, which has order n squared. The inner loop does not run n times on every pass, yet the leading growth is still quadratic.

An upper bound need not be tight: a linear-time function is also $O(n^2)$. Theta notation can state a matching growth rate when both upper and lower asymptotic bounds hold. For small inputs, constants and implementation costs can still outweigh the asymptotic comparison.

The points to check in this example are:

- Worst-case, average-case and amortized costs describe different analyses.
- A bound needs an input-size definition.
- Discarding constants is an asymptotic simplification, not a claim that constants never affect running time.

Check Your Understanding

Try the questions before reading the answers.

Question 1. Is $n(n + 1)/2$ in $O(n^2)$?

Answer. Yes; it is also in $\Theta(n^2)$.

Question 2. Does $O(n^2)$ prove a function cannot be $O(n)$?

Answer. No. An upper bound alone does not prove tightness.

Practice Questions

Work each question before reading its solution. The set runs from direct recall and substitution to the applied questions that exams actually use to separate grades. The downloadable PDF includes these questions and their solutions.

Question 1. Rank these functions from slowest-growing to fastest-growing: n^2 , $\log n$, 2^n , $5n$, $n \log n$.

Solution. $\log n < 5n < n \log n < n^2 < 2^n$. **Logarithms** grow slower than any linear term, the extra $\log n$ factor beats a constant multiplier, polynomials beat both, and exponentials beat every polynomial eventually. "Eventually" is the key word: for small n , 2^n can be smaller than $5n$.

Question 2. Simplify $O(3n^2 + 10n + 7)$ and explain why the dropped parts are dropped.

Solution. $O(n^2)$. Big-O describes growth as n gets large, and there the n^2 term dominates: at $n = 1000$, $3n^2$ is 3,000,000 while $10n + 7$ is barely 10,000. Constant factors are dropped because Big-O compares shapes of growth, not machine speeds.

Question 3. A loop runs i from 1 to n , and inside it a second loop runs j from 1 to m . What is the time complexity of the pair?

Solution. $O(nm)$: the inner body executes $n \times m$ times. It collapses to $O(n^2)$ only when m is proportional to n . Reading nested loops as automatic $O(n^2)$ is the classic mistake; count the actual iterations.

Question 4. Why is binary search $O(\log n)$, and roughly how many steps does it need on a sorted array of 1,000,000 items?

Solution. Each comparison halves the remaining range, so the number of steps is how many times you can halve n before reaching 1, which is $\log_2 n$. Since $2^{20} \approx 1,000,000$, binary search needs about 20 comparisons. Linear search would need up to 1,000,000.

Question 5. One function runs an $O(n)$ loop, then a second separate $O(n)$ loop. Is the function $O(n^2)$?

Solution. No. Sequential work adds: $O(n) + O(n) = O(2n) = O(n)$. Complexities multiply only when one loop runs inside the other. Sequential loops add, nested loops multiply.

Question 6. Merge sort is $O(n \log n)$ and bubble sort is $O(n^2)$. For an array of 100,000 items, estimate how many times more basic operations bubble sort performs.

Solution. Bubble sort: about 10^{10} operations. Merge sort: about $100,000 \times 17 \approx 1.7 \times 10^6$, since $\log_2 100,000 \approx 17$. The ratio is roughly 6,000 times. That is the difference between a program that finishes in milliseconds and one you kill after minutes.

Question 7. Hash table lookup is usually quoted as $O(1)$ average but $O(n)$ worst case. Where does the worst case come from?

Solution. Collisions. If every key lands in the same bucket, the "table" degrades into one long chain that must be scanned linearly. Good hash functions make this astronomically unlikely, which is why the average case is $O(1)$, but the worst case is real and adversarial inputs can trigger it.

Question 8. Summing an array iteratively uses $O(1)$ extra space. A naive recursive sum uses $O(n)$ extra space. Why?

Solution. Each recursive call adds a frame to the call stack, and the recursion goes n levels deep before unwinding, so the peak stack usage grows linearly. The iterative version reuses one accumulator variable. Time complexity is identical; space is not.

Question 9. Which grows faster: $n^{1.5}$ or $n \log n$?

Solution. $n^{1.5} = n \cdot n^{0.5}$ against $n \cdot \log n$, so compare $n^{0.5}$ with $\log n$. Any positive power of n eventually outgrows $\log n$, so $n^{1.5}$ grows faster. At $n = 10^6$: $n^{0.5} = 1000$ versus $\log_2 n \approx 20$.

Question 10. An $O(n^2)$ algorithm processes $n = 1000$ items in 0.01 seconds on your machine. Estimate the time for $n = 10^6$, and state the lesson.

Solution. Scaling n by 1000 scales n^2 by $1000^2 = 10^6$, so the time becomes about $0.01 \times 10^6 = 10,000$ seconds, nearly 3 hours. The lesson: complexity, not hardware, decides whether an algorithm survives growth. A faster computer buys a constant factor; a better algorithm changes the curve.

FAQs

What is Big-O notation?

A mathematical notation that describes how an algorithm's resource requirements (usually time or space) scale with input size, in the limit as input grows large. It strips away constants and lower-order terms to focus on the dominant growth rate.

What does $O(n)$ mean?

Linear time complexity. The number of operations grows proportionally with the input size n . Doubling the input doubles the work. Examples: linear search through an array, single-pass aggregation.

What does $O(\log n)$ mean?

Logarithmic time complexity. The number of operations grows like the logarithm of input size. Doubling the input adds just one extra step. Examples: binary search, balanced binary search tree operations.

Why do we ignore constants in Big-O?

Because Big-O describes asymptotic behavior, what happens as input grows toward infinity. Constants and lower-order terms become negligible at scale. The dominant term tells you how the algorithm fundamentally scales, which matters more for choosing between algorithms than implementation-level constants.

What's the difference between Big-O, Big-Theta, and Big-Omega?

Big-O is an upper bound (no faster than). Big-Omega is a lower bound (no slower than). Big-Theta is a tight bound (exactly this rate). In casual programming use, 'Big-O' often means 'Big-Theta,' but in formal computer science the distinction matters for proofs and lower-bound analysis.

How do I find the Big-O of an algorithm?

Identify the dominant operation count as a function of input size. Single loops over n items: $O(n)$. Nested loops: typically $O(n^2)$. Recursion: use recurrence relations or the Master theorem. Practice with standard algorithms until the patterns become recognizable on sight.

What is amortized complexity?

The average cost per operation when most operations are cheap and a few are expensive. Dynamic arrays (vectors) have amortized $O(1)$ append because rare resizing operations get absorbed into many cheap appends. Useful when worst-case complexity is misleading about typical behavior.

What does $O(1)$ mean?

Constant time complexity. The runtime doesn't depend on input size. Examples: array indexing, hash table lookup (average case), stack push/pop. The fastest possible complexity class, runtime stays bounded as input grows.

What does $O(n^2)$ mean?

Quadratic time complexity. The number of operations grows like the square of input size. Doubling the input quadruples the work. Examples: bubble sort, naive pair-wise comparisons, brute-force collision detection.

What's the difference between time complexity and space complexity?

Time complexity describes how runtime scales; space complexity describes how memory consumption scales. The same algorithm can be $O(n)$ in time but $O(1)$ in space (in-place algorithms) or $O(n)$ in time and $O(n)$ in space (algorithms using extra storage). Always specify which.

Why is $O(n \log n)$ considered fast?

Because it's only slightly worse than linear for practical input sizes. For $n = 1,000,000$, $n \log n$ is about 20 million operations, while n^2 would be 10^{12} operations, six orders of magnitude slower. $n \log n$ is also the proven optimal complexity for comparison-based sorting, so it's the gold standard for many algorithms.

How is Big-O used in interviews?

Coding interviews routinely ask 'what's the time complexity of your solution?' Knowing the standard Big-O of common algorithms (sorting, searching, graph traversal) and being able to analyze your own code is a baseline expectation. Practice analyzing every solution you write until it becomes second nature.

What is the master theorem in algorithm analysis?

A formula that gives the asymptotic complexity of divide-and-conquer recurrences $T(n) = aT(n/b) + f(n)$ in three cases based on how $f(n)$ compares to $n^{\log_b a}$. It covers most standard divide-and-conquer algorithms (merge sort, binary search, FFT) and is the fastest way to analyze them.

What's the difference between P and NP?

P is the class of problems solvable in polynomial time. NP is the class of problems whose solutions can be verified in polynomial time. $P \subseteq NP$ is obvious; whether $P = NP$ is one of the most important open questions in computer science (and a Millennium Prize Problem with a \$1M reward).