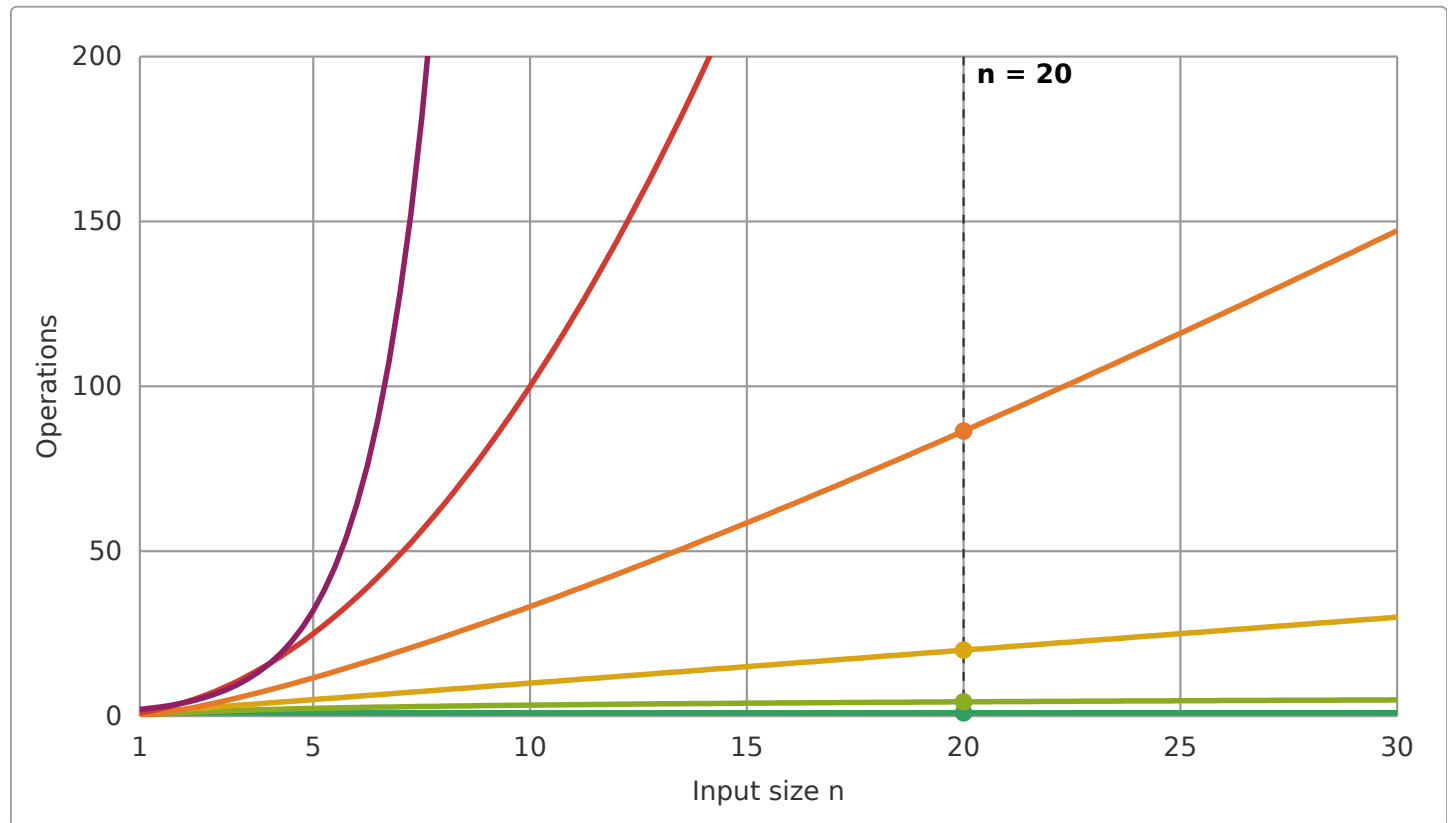


Big O cheat sheet

Time and space complexity for common data structures, sorting algorithms, graph and tree algorithms, and the interview staples that sit between them.

Complexity fundamentals

Big O describes how the work an algorithm does grows as the input size n grows. It ignores constants and lower-order terms, so it tells you which algorithm wins on large inputs, not which one is faster on ten items.



Approximate operations for the chosen n

Complexity	Name	Operations
$O(1)$	Constant	1
$O(\log n)$	Logarithmic	4.3
$O(n)$	Linear	20
$O(n \log n)$	Linearithmic	86.4
$O(n^2)$	Quadratic	400
$O(2^n)$	Exponential	1,048,576

Logarithms are base 2. The chart shows n from 1 to 30 and clips at 200 operations, so the steepest curves leave the plot early. Type a larger n to see the table values diverge.

$O(1)$: constant

Work does not depend on input size.

```
def first(items):  
    return items[0]
```

$O(\log n)$: logarithmic

Each step discards half of what remains.

```
def binary_search(items, target):  
    lo, hi = 0, len(items) - 1  
    while lo <= hi:  
        mid = (lo + hi) // 2  
        if items[mid] == target:  
            return mid  
        if items[mid] < target:  
            lo = mid + 1  
        else:  
            hi = mid - 1  
    return -1
```

O(n): linear

One pass over the input.

```
def total(items):
    result = 0
    for x in items:
        result += x
    return result
```

O(n log n): linearithmic

Sorting dominates, then one linear pass.

```
def has_duplicates(items):
    items = sorted(items)
    return any(a == b for a, b in zip(items, items[1:]))
```

O(n²): quadratic

A loop inside a loop over the same input.

```
def has_pair_sum(items, target):
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
            if items[i] + items[j] == target:
                return True
    return False
```

O(2ⁿ): exponential

Every item doubles the number of branches.

```
def count_subsets(items, target, i=0):
    if i == len(items):
        return 1 if target == 0 else 0
    take = count_subsets(items, target - items[i], i + 1)
    skip = count_subsets(items, target, i + 1)
    return take + skip
```

Color key

O(1)

O(log n)

O(n)

O(n log n)

O(n²)

O(2ⁿ)

N/A or not applicable

Common data structure operations

Data structure	Time complexity								Space complexity
	Average (or amortized, see notes)				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack ^a	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue ^a	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-linked list ^b	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-linked list ^b	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip list ^c	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)	O(n log n)
Hash table ^d	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary search tree ^e	N/A	O(h)	O(h)	O(h)	N/A	O(n)	O(n)	O(n)	O(n)
Cartesian tree ^f	N/A	O(log n)	O(log n)	O(log n)	N/A	O(n)	O(n)	O(n)	O(n)
B-tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Red-black tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Splay tree ^g	N/A	O(log n)	O(log n)	O(log n)	N/A	O(n)	O(n)	O(n)	O(n)
AVL tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
KD tree ^h	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)	O(n)

Average, amortized, and worst case

These three words describe different things, so the tables do not treat them as interchangeable.

Worst case

The most work a single operation can need, on the least favorable input. It holds no matter what the data looks like.

Average (expected) case

The cost averaged over a stated assumption, such as random keys, random insertion order, or a hash function that spreads keys evenly. If the assumption fails, the bound can fail with it.

Amortized case

The cost per operation when you average over a whole sequence of operations, with no randomness involved. An occasional expensive operation (like a resize) is paid for by many cheap ones, but a single operation can still be slow.

Where a bound is tight, it is written with Θ ; worst-case columns use O as an upper bound. Implementation details and assumptions behind specific rows are in the notes below.

- Stack and queue.** Push, pop, enqueue, and dequeue are $O(1)$ when built on a linked list, a circular buffer, or a dynamic array used at one end (a dynamic array's push is amortized $O(1)$). A queue on a plain array that shifts every element after removing from the front, such as Python's `list.pop(0)`, takes $O(n)$ per dequeue.
- Linked lists.** Insertion and deletion are $O(1)$ only when you already hold a reference to the position: the head, or the node before it in a singly-linked list. Finding that position takes $O(n)$.

- c. **Skip list.** Search, insertion, and deletion are expected bounds that come from random level choices; the worst case is $O(n)$. Space is $O(n)$ in expectation; $O(n \log n)$ is the worst case when levels are capped at $\log n$.
- d. **Hash table.** $\Theta(1)$ is the expected cost assuming a good hash function that spreads keys evenly and a load factor kept bounded by resizing. If many keys collide, a single operation can take $O(n)$. Insertion is also amortized, because a resize rehashes every key.
- e. **Binary search tree.** An ordinary BST costs $O(h)$ for search, insertion, and deletion, where h is its height. That is $\Theta(\log n)$ when the tree is balanced or keys arrive in random order, and $O(n)$ in the worst case, such as inserting sorted keys. Self-balancing trees (AVL, red-black, B-tree) keep h at $O(\log n)$ and so give a worst-case guarantee. A BST does not support access by position.
- f. **Cartesian tree.** The $\Theta(\log n)$ figures assume random priorities, as in a treap. A tree built from adversarial or already-ordered input can degrade to $O(n)$.
- g. **Splay tree.** $\Theta(\log n)$ is an amortized bound over a sequence of operations. A single operation can take $O(n)$, even though the average over many operations is $O(\log n)$.
- h. **KD tree.** The $\Theta(\log n)$ figures assume points that keep the tree reasonably balanced. Nearest-neighbor search in high dimensions can degrade toward $O(n)$.

More data structures

These structures answer different questions than the table above, so each lists the operations it is actually used for. In the trie rows, m is the key length.

Data structure	Operation	Time	Space	Notes
Dynamic array	Access by index	$O(1)$	$O(n)$	Python <code>list</code> , Java <code>ArrayList</code> , C++ <code>vector</code> . Capacity grows geometrically, so most appends are cheap and an occasional append copies everything.
	Append	$O(1)$ amortized, $O(n)$ worst		
	Insert or delete in the middle	$O(n)$		
	Remove last	$O(1)$		
	Search (unsorted)	$O(n)$		
Binary heap	Peek min or max	$O(1)$	$O(n)$	Stored in an array. Heaps give fast access to one extreme, not sorted order or fast search.
	Insert	$O(\log n)$ worst, $O(1)$ average*		
	Extract min or max	$O(\log n)$		
	Build from n items	$O(n)$		
	Update or delete at a known position	$O(\log n)$		
	Search for a value	$O(n)$		
Trie	Insert	$O(m)$	$O(n \cdot m)$ nodes	Cost depends on key length, not on how many keys are stored. Child arrays multiply space by the alphabet size; hash-map children save space.
	Search	$O(m)$		
	Delete	$O(m)$		
	Find all keys with a prefix	$O(m + k)$ for k results		
Disjoint-set union (Union-Find)	Make set	$O(1)$	$O(n)$	With path compression and union by rank or size, find and union cost amortized $O(\alpha(n))$, which is at most 4 for any practical n . Without both, a find can take $O(n)$.
	Find	$O(\alpha(n))$ amortized		
	Union	$O(\alpha(n))$ amortized		
Monotonic stack	Push or pop	$O(1)$	$O(n)$	A stack that keeps its items in sorted order by popping anything that breaks the order. Each item is pushed and popped at most once, so problems like next greater element take $O(n)$ in total, not $O(n^2)$.
	Peek	$O(1)$		
	Process n items (whole pass)	$O(n)$ amortized		
Deque	Push or pop at either end	$O(1)$	$O(n)$	Ring-buffer deques (C++ <code>deque</code> , Java <code>ArrayDeque</code>) allow $O(1)$ indexing. Python <code>collections.deque</code> uses linked blocks, so indexing near the middle is $O(n)$. A monotonic deque solves sliding-window maximum in $O(n)$.
	Peek either end	$O(1)$		
	Access by index	$O(1)$ ring buffer, $O(n)$ linked blocks		
	Search	$O(n)$		

Priority queue implementations

A priority queue is an interface, not a single structure. The cost of each operation depends on what sits underneath.

Implementation	Peek	Insert	Extract min or max	Decrease key	Notes
----------------	------	--------	--------------------	--------------	-------

Unsorted array	$O(n)$	$O(1)$	$O(n)$	$O(1)$ at a known index	Fine for small n or when extracts are rare.
Sorted array	$O(1)$	$O(n)$	$O(1)$ from the sorted end	$O(n)$	Fine when extracts far outnumber inserts.
Binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	The default choice. Python (<code>heapq</code>), Java (<code>PriorityQueue</code>), and C++ (<code>priority_queue</code>) use it. Most do not support decrease-key directly.
Fibonacci heap	$O(1)$	$O(1)$ amortized	$O(\log n)$ amortized	$O(1)$ amortized	Best asymptotics for Dijkstra and Prim, but large constants make binary heaps faster in practice.

- * Inserting into a binary heap built from random data moves up a constant number of levels on average. The worst case is still $O(\log n)$.

Graph representations

A graph has V vertices and E edges. The representation you pick decides which operations are cheap. Here $\deg(v)$ is the number of edges at vertex v .

Operation	Adjacency list	Adjacency matrix	Edge list
Space	$O(V + E)$	$O(V^2)$	$O(E)$
Check whether edge (u, v) exists	$O(\deg(u))$ *	$O(1)$	$O(E)$
Iterate neighbors of v	$O(\deg(v))$	$O(V)$	$O(E)$
Iterate all edges	$O(V + E)$	$O(V^2)$	$O(E)$
Add edge	$O(1)$	$O(1)$	$O(1)$
Remove edge	$O(\deg(u))$ *	$O(1)$	$O(E)$
Add vertex	$O(1)$	$O(V^2)$ **	$O(1)$
Best for	Sparse graphs and traversals such as BFS, DFS, and Dijkstra.	Dense graphs and algorithms that test many edges, such as Floyd-Warshall.	Algorithms that sort or scan every edge, such as Kruskal and Bellman-Ford.

- * Storing each vertex's neighbors in a hash set makes edge lookup and removal $O(1)$ on average, at the cost of more memory per vertex.
- ** Growing a matrix means copying it. Preallocating capacity for the maximum number of vertices avoids the copy.
- For an undirected graph, adjacency lists and edge lists store each edge once or twice depending on the implementation; the asymptotic bounds are the same.

The same graph three ways

An undirected graph with four vertices and edges 0-1, 0-2, 1-3, and 2-3.

Adjacency list

```
0: [1, 2]
1: [0, 3]
2: [0, 3]
3: [1, 2]
```

Adjacency matrix

```

  0 1 2 3
0 [ 0 1 1 0 ]
1 [ 1 0 0 1 ]
2 [ 1 0 0 1 ]
3 [ 0 1 1 0 ]
```

Edge list

```
(0, 1)
(0, 2)
(1, 3)
(2, 3)
```

Array sorting algorithms

Algorithm	Time complexity			Space complexity
	Best	Average	Worst	Auxiliary, worst
Quicksort ^a	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n^2)$	$O(\log n)$
Mergesort	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n \log n)$	$O(n)$
Timsort	$\theta(n)$	$\theta(n \log n)$	$\theta(n \log n)$	$O(n)$
Heapsort ^b	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n \log n)$	$O(1)$
Bubble sort ^c	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	$O(1)$
Insertion sort	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	$O(1)$
Selection sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$O(1)$
Tree sort ^d	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n^2)$	$O(n)$
Shell sort ^e	$\Omega(n \log n)$	Depends on gaps	$O(n^2)$	$O(1)$
Bucket sort ^f	$\theta(n+k)$	$\theta(n+k)$	$O(n^2)$	$O(n+k)$
Radix sort	$\theta(nk)$	$\theta(nk)$	$\theta(nk)$	$O(n+b)$

Counting sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	$O(n+k)$
Cubesort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

Notation in this table

- $\Theta(f)$ is a tight bound: the cost grows like f and no slower or faster. It is used wherever the bound is known to be tight.
- $O(f)$ is an upper bound and $\Omega(f)$ is a lower bound. They are used where only one side is established.
- Best, average, and worst name the input, not the bound type. Quicksort, for example, is $\Theta(n \log n)$ in the best and average cases and $\Theta(n^2)$ in the worst case.
- Space is auxiliary space beyond the input array.

- Quicksort space.** $O(\log n)$ when the recursion handles the smaller partition first. A naive implementation can need $O(n)$ stack space in the worst case.
- Heapsort.** The $\Theta(n \log n)$ best case holds for distinct keys. Input where every key is equal can finish in $O(n)$.
- Bubble sort.** The $\Theta(n)$ best case needs an early exit when a pass makes no swaps. Without it, every case is $\Theta(n^2)$.
- Tree sort.** Assumes inserting into a plain binary search tree, which degrades to $\Theta(n^2)$ on already-sorted input. A self-balancing tree makes the worst case $\Theta(n \log n)$.
- Shell sort.** The running time depends on the gap sequence. Shell's original gaps give $O(n^2)$ in the worst case; sequences such as Pratt's give $O(n (\log n)^2)$. Average-case bounds are not known for most sequences.
- Bucket sort.** The $\Theta(n+k)$ average assumes input spread roughly evenly across buckets. If most items land in one bucket, the cost is dominated by sorting that bucket, up to $O(n^2)$.

Algorithm complexity

A quick reference for the searching, graph, heap, and interval algorithms that come up most in interviews. Detailed breakdowns follow in the next section.

Algorithm	Best	Average	Worst	Space
Linear search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary search	$O(1)$	$O(\log n)$	$O(\log n)^*$	$O(1)^*$
BFS	$O(V + E)$	$O(V + E)$	$O(V + E)$	$O(V)$
DFS	$O(V + E)$	$O(V + E)$	$O(V + E)$	$O(V)^*$
Heap insert	—	$O(\log n)$	$O(\log n)$	$O(1)$ auxiliary
Heap peek	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Heap extract min or max	—	$O(\log n)$	$O(\log n)$	$O(1)$ auxiliary
Merge intervals	—	$O(n \log n)$	$O(n \log n)$	$O(n)^*$
Union-Find operation	—	Amortized $O(\alpha(n))$	—	$O(n)$ total structure
Dijkstra with binary heap	—	—	$O((V + E) \log V)^*$	$O(V + E)$

- * Complexity depends on the implementation and assumptions. Binary search takes $O(\log n)$ time only on sorted random-access data; iterative binary search uses $O(1)$ auxiliary space. BFS and DFS space depends on the graph representation and traversal implementation. Dijkstra's bound assumes a standard adjacency-list implementation with a binary heap.
- Heap insert is $O(\log n)$ as an upper bound. On random data the expected cost is $O(1)$.
- Merge intervals is dominated by sorting the intervals; the $O(n)$ space is the output list, plus any space the sort needs.
- Union-Find reaches amortized $O(\alpha(n))$ only with path compression and union by rank or size. α is the inverse Ackermann function, which stays below 5 for any input that fits in memory.

Graph and tree algorithms

V is the number of vertices, E the number of edges, n the number of tree nodes, and h the tree height (about $\log n$ when balanced, up to n when skewed). Graph bounds assume an adjacency list unless noted.

Traversal and search

Algorithm	Time	Space	Notes
Breadth-first search (BFS)	$O(V + E)$	$O(V)$	Uses a queue. Finds shortest paths by edge count. $O(V^2)$ with an adjacency matrix.
Depth-first search (DFS)	$O(V + E)$	$O(V)$	Uses recursion or an explicit stack, so depth can reach V . Basis for cycle detection and topological sort. $O(V^2)$ with an adjacency matrix.
Binary search on a sorted array	$O(\log n)$	$O(1)$ iterative, $O(\log n)$ recursive	Requires sorted, random-access data. Also works on any monotonic yes/no condition, not just arrays.
Binary search tree lookup	$O(h)$	$O(1)$ iterative	$O(\log n)$ when balanced, $O(n)$ when skewed.
Preorder, inorder, postorder traversal	$O(n)$	$O(h)$	Recursion or stack depth is the height. Inorder on a binary search tree visits keys in sorted order.

Level-order traversal	$O(n)$	$O(w)$	Uses a queue; w is the widest level, which can be about $n/2$ in a complete tree.
Morris traversal	$O(n)$	$O(1)$	Inorder or preorder with no stack. Temporarily rewires tree pointers, so it is not safe for concurrent readers.

Heap operations

For a binary heap holding n items.

Operation	Time	Notes
Peek min or max	$O(1)$	Read the root.
Insert	$O(\log n)$	Append at the end, then sift up. $O(1)$ on average for random data.
Extract min or max	$O(\log n)$	Swap the root with the last item, remove it, then sift down.
Build heap from an array	$O(n)$	Sift down from the last parent to the root. Faster than n separate inserts, which cost $O(n \log n)$.
Change priority or delete at a known index	$O(\log n)$	Finding the index without a position map costs $O(n)$.
Search for an arbitrary value	$O(n)$	A heap only orders parents against children.
Heapsort	$O(n \log n)$	$O(1)$ extra space. See the sorting table.

Shortest-path algorithms

Algorithm	Edge weights	Time	Space	Notes
BFS	Unweighted	$O(V + E)$	$O(V)$	Single source.
0-1 BFS	0 or 1 only	$O(V + E)$	$O(V)$	Single source. Uses a deque instead of a heap.
Shortest path in a DAG	Any, including negative	$O(V + E)$	$O(V)$	Single source. Relax edges in topological order.
Dijkstra with binary heap	Non-negative	$O((V + E) \log V)$	$O(V + E)$ *	Single source. The usual choice for sparse graphs.
Dijkstra with array	Non-negative	$O(V^2)$	$O(V)$	Single source. Beats the heap version on dense graphs.
Dijkstra with Fibonacci heap	Non-negative	$O(E + V \log V)$	$O(V + E)$	Single source. Best bound, rarely fastest in practice.
Bellman-Ford	Negative allowed	$O(V \cdot E)$	$O(V)$	Single source. Detects negative cycles reachable from the source.
Floyd-Warshall	Negative allowed, no negative cycles	$O(V^3)$	$O(V^2)$	All pairs. Works on an adjacency matrix.

- * The space includes the adjacency list. A heap that allows duplicate entries instead of decrease-key can hold up to E items; an indexed heap with decrease-key needs only $O(V)$ beyond the graph. The $O((V + E) \log V)$ bound assumes a standard adjacency-list implementation.
- Other space figures count extra space beyond the input graph.

Minimum spanning trees

These algorithms take a connected, undirected, weighted graph. On a disconnected graph they produce a minimum spanning forest.

Algorithm	Time	Space	Notes
Kruskal	$O(E \log E)$	$O(V + E)$	Sort edges, then add each one that joins two components, using Union-Find. $O(E \log E)$ equals $O(E \log V)$ because E is at most V^2 . Works from an edge list.
Prim with binary heap	$O(E \log V)$	$O(V + E)$	Grows one tree from a start vertex. Good for sparse graphs.
Prim with array	$O(V^2)$	$O(V)$	Good for dense graphs stored as a matrix.
Prim with Fibonacci heap	$O(E + V \log V)$	$O(V + E)$	Best bound, rarely fastest in practice.
Borůvka	$O(E \log V)$	$O(V)$	Each round adds the cheapest edge leaving every component. Components at least halve each round, so it needs at most $\log V$ rounds. Easy to parallelize.

Coding patterns

Interview problems tend to reuse a handful of patterns. Knowing the typical cost of each lets you estimate a solution before you write it.

These are typical complexities, not guarantees. The real cost depends on the problem, on how you implement the pattern, and on any extra work such as sorting the input first. Always check the bound against your own code.

Coding pattern	Typical time complexity	Typical auxiliary space	Typical problems and caveats
Two pointers	$O(n)$	$O(1)$	Pair sum on a sorted array, removing duplicates in place, palindrome check. If the input is not sorted and the pattern needs it, add $O(n \log n)$ for the sort.
Sliding window	$O(n)$	$O(1)$ or $O(n)$	Longest substring or subarray under a constraint. Space is $O(1)$ for a running sum or count, and grows when the window stores its contents, such as a character map or a deque for the window maximum.
Prefix sums	$O(n)$ preprocessing; $O(1)$ per range sum	$O(n)$	Many range-sum queries on a fixed array. Updates to the array force a rebuild, or a Fenwick or segment tree instead.

Hash map frequency counting	Expected $O(n)$ total	$O(n)$	Anagrams, duplicates, two-sum. Relies on expected $O(1)$ hash operations; heavy collisions can make a single operation $O(n)$.
Monotonic stack	$O(n)$ total	$O(n)$	Next greater element, daily temperatures, largest rectangle in a histogram. Each item is pushed and popped at most once.
Backtracking and subsets	$O(2^n)$ output generation	Depends on implementation	Recursion depth alone is $O(n)$. Storing every subset needs $O(n \cdot 2^n)$ space, and copying each subset into the result adds a factor of n to the time.
Permutations	$O(n \cdot n!)$ to generate all permutations	Depends on output and implementation	There are $n!$ permutations of length n . Recursion depth is $O(n)$; storing all of them takes $O(n \cdot n!)$.
Dynamic programming	Number of states $\times$ transitions per state	Depends on memoization or tabulation	Knapsack with n items and capacity W has $n \cdot W$ states with two transitions each, so $O(n \cdot W)$ time. Tabulation often lets you keep only the previous row, cutting space to $O(W)$. Memoization also uses recursion stack space.

Reading the notation

- $O(\dots)$ describes an asymptotic upper bound.
- $\Omega(\dots)$ describes an asymptotic lower bound.
- $\Theta(\dots)$ describes a tight asymptotic bound.
- n is the input size. Graph algorithms use V for vertices and E for edges. Some algorithms use extra parameters, such as k or m , as noted in each table.
- $\alpha(n)$ is the inverse Ackermann function. It grows so slowly that it is at most 4 for any practical input.
- Average (expected) cost depends on a stated assumption, such as random input or a well-behaved hash function. Amortized cost is the average per operation over a worst-case sequence of operations and involves no randomness. Worst-case cost is the most a single operation can take. See the notes under the data structure table.
- Tables use Θ where a bound is known to be tight, O for upper bounds, and Ω for lower bounds.
- N/A means the operation is not applicable in that table's context.

Complexities are quick-reference summaries. Actual performance depends on the implementation, the input, and the assumptions behind each operation, so check the linked references when exact guarantees matter.