DSA Notes: Beginner to Advanced (with Examples)

WHAT DSA IS AND HOW TO USE THESE NOTES

What DSA Is

Data structures and algorithms (DSA) are two halves of one skill. A **data structure** is a way of organizing data in memory so that certain operations are efficient: arrays, hash maps, linked lists, trees, heaps, and graphs are examples. An **algorithm** is a step-by-step procedure that solves a problem, such as searching, sorting, or finding a shortest path. The choice of structure often determines which algorithms are possible and how fast they run, so the two are always studied together.

Why It Matters

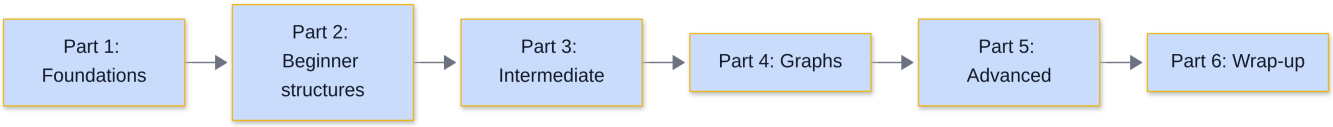
- **Interviews:** Coding interviews commonly test whether you can pick a suitable structure, reason about its cost, and write a correct solution under time pressure.
- **University courses:** DSA courses build the vocabulary (Big-O, recursion, trees, graphs, dynamic programming) that later subjects rely on.
- **Everyday programming:** Knowing why a lookup is fast or slow helps you avoid designs that stop working as input grows.

How These Notes Are Organized

The notes move from beginner to advanced across six parts:

Part	Focus	Main topics
1. Foundations	How to think about algorithms	Big-O, Omega, Theta; best/average/worst case; space complexity; growth rates; recursion basics; approaching a problem
2. Beginner structures and algorithms	Everyday tools	Arrays, strings, hashing, linked lists, stacks, queues, searching, basic sorting
3. Intermediate	Efficient and recursive techniques	Merge/quick/heap/counting sort, backtracking, trees, BSTs, heaps, tries, bit manipulation
4. Graphs	Relationships and networks	Representation, BFS/DFS, cycles, topological sort, shortest paths, spanning trees, Union-Find, grids
5. Advanced	Optimization and theory	Dynamic programming, greedy, divide and conquer, advanced trees, string algorithms, amortized analysis, NP-completeness
6. Wrap-up	Consolidation	Complexity cheat sheet, pattern table, 12-week roadmap, interview tips, practice list

Learning Roadmap



Each part assumes the one before it. Part 1 gives you the tools to compare solutions; Parts 2 and 3 supply the structures; Part 4 applies them to graphs; Part 5 covers techniques that combine them; Part 6 helps you review and plan practice.

How to Use Each Topic

Topics are intended to follow a consistent pattern: the concept, an intuition for why it works, a worked example with a small trace or diagram, code, time and space complexity, common pitfalls, and one or two practice problems. Code is written in Python, with short notes on Java or C++ equivalents where they help.

Suggested Study Habits

These are suggestions, not requirements:

1. Trace each example by hand before reading the code.
2. Re-implement the code from memory without looking.
3. Attempt the practice problems before checking any solution.
4. State the time and space complexity of every solution you write.
5. Revisit earlier parts regularly; later topics reuse them heavily.

Beginners should follow the parts in order. If you already know the basics, skim Parts 1 and 2 and use the cheat sheet in Part 6 to find gaps.

ANALYZING ALGORITHMS: ASYMPTOTIC NOTATION

Why We Analyze Algorithms

Two programs can solve the same problem yet behave very differently as the input grows. Timing a program on one machine is unreliable, because the result depends on hardware, language and input. Instead, we count how the number of basic steps grows with the input size, usually written *n*. This lets us compare algorithms independently of the machine.

The Three Notations

Notation	Meaning	Intuition
Big-O, O(f(n))	Upper bound	Grows no faster than f(n)
Omega, Ω(f(n))	Lower bound	Grows at least as fast as f(n)
Theta, Θ(f(n))	Tight bound	Bounded above and below by f(n)

Constant factors and lower-order terms are dropped. A routine that performs 3n + 5 steps is O(n), and it is also Θ(n), because the growth is linear from both sides. Big-O is the notation used most often in interviews, but it is only an upper bound: a linear algorithm is technically also $O(n^2)$, though that statement is not useful. Aim to give the tightest bound you can justify.

Best, Average and Worst Case

The same algorithm can take different amounts of work on different inputs of the same size. We describe this with three cases:

- **Best case:** the input that makes the algorithm fastest.
- **Average case:** the expected cost over typical inputs (this requires an assumption about how inputs are distributed).
- **Worst case:** the input that makes the algorithm slowest. Interviews and courses usually quote this unless told otherwise.

Notation and case are separate ideas. Big-O, Omega and Theta describe bounds on a growth function; best, average and worst describe which input scenario that function represents. You can state a Big-O bound for the best case or an Omega bound for the worst case.

Worked Example: Linear Search

```python
def linear_search(arr, target):
    for i in range(len(arr)):      # runs at most n times
        if arr[i] == target:       # one comparison per step
            return i
    return -1
```

Take `arr = [7, 3, 9, 4]`, so n = 4.

Target	Comparisons	Case
7	1	Best: first element
9	3	Somewhere in the middle
4	4	Worst: last element
8	4	Worst: not present

- **Best case:** one comparison, which is constant, so $\Theta(1)$.
- **Worst case:** n comparisons, so $\Theta(n)$, and therefore $O(n)$.
- **Average case:** if the target is equally likely to be at any position, about $n/2$ comparisons are made, which is still $\Theta(n)$ once the constant is dropped.

So we can say linear search is $O(n)$ overall, $\Omega(1)$ overall, and $\Theta(n)$ in the worst case. No single Theta bound covers every input, which is why the case must be named.

How to Count Steps

1. Identify the input size *n*.
2. Count how many times the most frequent operation runs, usually inside the deepest loop.
3. Multiply the counts of nested loops; add the counts of sequential blocks.
4. Keep the dominant term and discard constants.

For example, two sequential loops of n steps each give $2n$, which is $O(n)$. A loop nested inside another, both running n times, gives n^2, which is $O(n^2)$.

Common Pitfalls

- Treating Big-O as "the worst case". It is a bound, not a case.
- Forgetting that the input size may be more than one variable, such as n and m.
- Keeping constants and lower-order terms in the final answer.

Practice

1. State the best-case and worst-case cost of finding the maximum in an unsorted list of n numbers. Does the case change the answer?
2. Count the steps for a double loop where the inner loop runs from 0 to i, and give its Big-O (a suggestion: sum the inner counts).

GROWTH RATES AND SPACE COMPLEXITY

Why Growth Rates Matter

Big-O describes how running time or memory grows as the input size *n* grows. Constant factors matter little once *n* is large, so the shape of the growth curve decides whether a solution is practical. Learning to rank the common growth rates lets you judge an approach before you write any code.

Common Growth Rates

The table lists the usual complexities from slowest-growing (best) to fastest-growing (worst), with a typical example for each. The examples are standard illustrations drawn from topics covered later in these notes.

Rank	Notation	Name	Typical example
1	O(1)	Constant	Array access by index; hash map lookup (average)
2	O(log n)	Logarithmic	Binary search
3	O(n)	Linear	Linear search; one pass over an array
4	O(n log n)	Linearithmic	Merge sort; heap sort
5	$O(n^2)$	Quadratic	Bubble, selection and insertion sort; nested loops
6	$O(2^n)$	Exponential	Generating all subsets; naive recursive Fibonacci
7	O(n!)	Factorial	Generating all permutations

Intuition. Doubling *n* leaves O(1) unchanged, adds one step to O(log n), doubles O(n), quadruples $O(n^2)$, and squares the work of $O(2^n)$. This is why an $O(n^2)$ idea that is fine for a few hundred items may be too slow for a million, while O(n log n) usually remains comfortable.

Reading Complexity from Code

A few rules of thumb (suggested heuristics, not exhaustive):

- A single loop over the input is typically O(n).
- Two nested loops over the input are typically $O(n^2)$.
- A loop that halves the remaining range each step is O(log n).
- A recursion that makes two calls on a problem of size n−1 tends toward $O(2^n)$.
- Drop constants and lower-order terms: O(3n + 10) is O(n), and $O(n^2 + n)$ is $O(n^2)$.

```python
def has_duplicate_slow(a):      # O(n^2) time, O(1) extra space
    for i in range(len(a)):
        for j in range(i + 1, len(a)):
            if a[i] == a[j]:
                return True
    return False

def has_duplicate_fast(a):      # O(n) time, O(n) extra space
    seen = set()
    for x in a:
        if x in seen:
            return True
        seen.add(x)
    return False
```

The two functions solve the same problem and illustrate a recurring theme: spending memory can buy speed.

Space Complexity

Space complexity measures how much memory an algorithm uses as a function of input size. It is usually expressed as **auxiliary space**: the extra memory beyond the input itself. Always state which convention you are using.

Sources of memory cost:

- **New data structures**, such as a set, array or table of size proportional to n (O(n)), or a 2D table (O(n·m)).
- **The recursion call stack.** Each active call occupies a stack frame, so recursion that goes d levels deep uses O(d) space even if it allocates nothing else. A recursive factorial of n uses O(n) stack space.
- **Output storage**, which some analyses count and others exclude.

Approach	Time	Extra space
Nested-loop duplicate check	$O(n^2)$	O(1)
Hash-set duplicate check	O(n)	O(n)
Iterative loop summing a list	O(n)	O(1)
Recursive sum of a list	O(n)	O(n) stack

Common Pitfalls

- Ignoring the call stack when calling a solution “O(1) space.”
- Treating average-case O(1) hash lookups as guaranteed worst-case bounds.
- Keeping constants and lower-order terms in a final Big-O answer.
- Judging speed only on small inputs, where all growth rates look similar.

Practice Problems

1. Rank these from fastest to slowest growing: O(n log n), O(n), $O(2^n)$, O(log n), $O(n^2)$.

2. Give the time and auxiliary space of a function that builds a list of all pairs (i, j) from an array of length *n*, and explain how the answer changes if the output is not counted.

RECURSION BASICS AND PROBLEM-SOLVING APPROACH

Concept and Intuition

Recursion is a technique in which a function solves a problem by calling itself on a smaller version of the same problem. Every correct recursive function has two parts:

- **Base case**: the smallest input, answered directly without further calls.
- **Recursive case**: a step that reduces the input and combines the sub-result into the answer.

If the base case is missing or the input never shrinks, the calls never stop and the program fails with a stack overflow (in Python, a `RecursionError`).

Worked Example 1: Factorial

`n! = n × (n-1)!`, with `0! = 1` as the base case.

```python
def factorial(n):
    if n == 0:          # base case
        return 1
    return n * factorial(n - 1)   # recursive case
```

Call-stack trace for `factorial(3)` (calls push frames downward; returns unwind upward):

Step	Action	Stack (top at right)
1	call factorial(3)	f(3)
2	call factorial(2)	f(3), f(2)
3	call factorial(1)	f(3), f(2), f(1)
4	call factorial(0), base case returns 1	f(3), f(2), f(1), f(0)
5	f(1) returns 1 × 1 = 1	f(3), f(2), f(1)
6	f(2) returns 2 × 1 = 2	f(3), f(2)
7	f(3) returns 3 × 2 = 6	f(3)

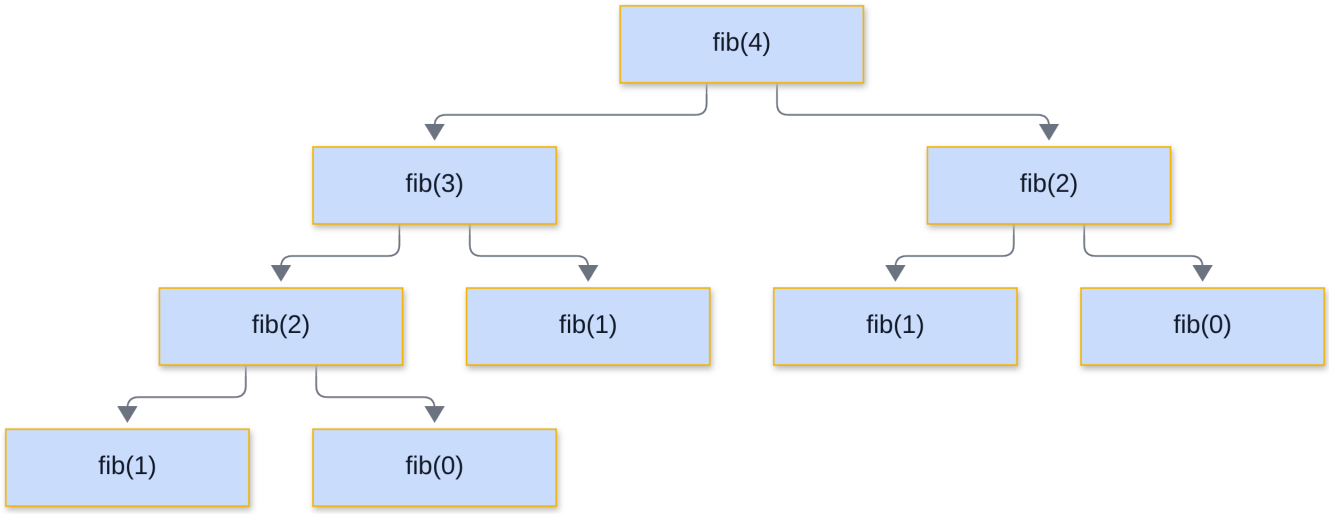
Complexity: time O(n), because there are n+1 calls. Space O(n), because the call stack holds up to n+1 frames at once.

Worked Example 2: Fibonacci

Each term is the sum of the previous two: `F(0)=0`, `F(1)=1`, `F(n)=F(n-1)+F(n-2)`.

```python
def fib(n):
    if n < 2:           # base cases: 0 and 1
        return n
    return fib(n - 1) + fib(n - 2)
```

The call tree for `fib(4)` shows the problem: it branches twice per call.



`fib(2)` is computed twice, and the repetition grows rapidly with n. The naive version takes exponential time ($O(2^n)$, a standard upper bound) with O(n) stack depth. Storing results already computed (memoization, covered in the dynamic programming sections) removes the repeated work and brings time down to O(n).

Common Pitfalls

- Forgetting the base case, or writing one the input can skip over (for example testing `n == 0` when `n` may be negative).
- Not shrinking the input on each call.
- Recomputing overlapping subproblems, as in naive Fibonacci.
- Ignoring stack depth: deep recursion costs O(depth) space.

A Repeatable Problem-Solving Framework

The following steps are a suggested routine, not a fixed rule; use them on any problem in these notes.

1. **Restate** the problem in your own words and identify inputs and outputs.
2. **Clarify constraints**: input size, value ranges, empty or duplicate inputs.
3. **Work a small example** by hand, including an edge case.
4. **Start with brute force** and state its complexity.
5. **Find the bottleneck** and choose a data structure or pattern that removes it.
6. **Code** cleanly, naming the base cases or invariants first.
7. **Test** with your example, then edge cases (empty, single element, duplicates).
8. **Analyze** final time and space complexity.

Practice Problems

1. Write a recursive function that returns the sum of the digits of a non-negative integer, and draw its call stack for one input of your choice.
2. Write a recursive `power(x, n)` for non-negative integer `n`, then state its time and space complexity.

ARRAYS: CORE OPERATIONS AND PATTERNS

Concept and Intuition

An array stores elements in contiguous memory, so the element at index `i` is reached by direct address arithmetic. That makes access O(1), but it makes changes in the middle costly, because neighbouring elements must shift to keep the block contiguous. Python's `list` is a dynamic array; Java uses `int[]` or `ArrayList`, and C++ uses `vector`.

Core Operations

```python
a = [4, 8, 15, 16]
for x in a: print(x)          # traversal, O(n)
a.insert(2, 99)               # insert at index 2 -> [4, 8, 99, 15, 16], O(n)
a.pop(1)                      # delete index 1 -> [4, 99, 15, 16], O(n)
```

Trace of `insert(2, 99)` on `[4, 8, 15, 16]`: 16 and 15 each shift one slot right, then 99 is placed at index 2. Appending at the end is amortized O(1); inserting or deleting at the front is O(n).

Two-Pointer Technique

Use two indices that move toward each other or in the same direction to avoid a nested loop. Example: in a **sorted** array, find a pair summing to a target.

```python
def pair_sum(a, target):
    l, r = 0, len(a) - 1
    while l < r:
        s = a[l] + a[r]
        if s == target: return (l, r)
        if s < target: l += 1
        else: r -= 1
    return None
```

If the sum is too small, only moving `l` right can increase it; if too large, moving `r` left decreases it. Time O(n), space O(1).

Prefix Sum

Precompute `p[i]` as the sum of the first `i` elements; then the sum of `a[l..r]` is `p[r+1] - p[l]`.

```python
p = [0]
for x in a: p.append(p[-1] + x)
```

For `[2, 4, 6, 8]`, `p = [0, 2, 6, 12, 20]`, and the sum of indices 1..2 is `12 - 2 = 10`. Build is O(n); each query is O(1); extra space is O(n).

Complexity Summary

Operation	Time
Access by index	O(1)
Search (unsorted)	O(n)
Insert/delete in middle	O(n)
Append	O(1) amortized

Common Pitfalls

- Off-by-one errors in loop bounds and prefix-sum indices.
- Modifying a list while iterating over it.
- Applying two pointers to unsorted data without a justification.
- Forgetting empty-array cases.

Practice Problems

1. Remove duplicates from a sorted array in place (two pointers).
2. Range Sum Query – Immutable (prefix sum).

KADANE'S ALGORITHM: MAXIMUM SUBARRAY

Concept and Intuition

The maximum subarray problem asks for the largest sum of any non-empty contiguous block of an array. Kadane's algorithm solves it in one pass. At each index it decides whether to extend the best subarray ending at the previous index or start fresh at the current element. If the running sum becomes a burden (negative), carrying it forward can only hurt, so we discard it.

Worked Example and Trace

Take the illustrative array `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`. Let `cur` be the best sum ending at the current index and `best` the best seen so far.

i	x	cur = max(x, cur + x)	best
0	-2	-2	-2
1	1	1	1
2	-3	-2	1
3	4	4	4
4	-1	3	4
5	2	5	5
6	1	6	6
7	-5	1	6
8	4	5	6

The answer is 6, from the subarray `[4, -1, 2, 1]`.

Code (Python)

PYTHON

```python
def max_subarray(nums):
    cur = best = nums[0]
    for x in nums[1:]:
        cur = max(x, cur + x)
        best = max(best, cur)
    return best
```

Initialising from `nums[0]` rather than `0` keeps the result correct when every element is negative. In Java or C++, the same loop works with `Math.max` or `std::max`; initialise with the first element, not with `0`.

Complexity

- **Time:** O(n), a single pass.
- **Space:** O(1), two variables.

Common Pitfalls

- Starting `best` at 0, which wrongly returns 0 for all-negative arrays.
- Forgetting that the subarray must be contiguous and non-empty.
- Returning only the sum when the problem asks for indices; track a start index that resets whenever you start fresh.

Practice Problems

1. Maximum Subarray: implement the function above and test it on an all-negative array.
2. Extension: modify the code to also return the start and end indices of the best subarray.
3. Challenge: Maximum Product Subarray, where you must track both the maximum and minimum product ending at each index.

STRINGS AND THE SLIDING WINDOW

Concept and Intuition

A string is a sequence of characters. In Python, strings are immutable, so "changing" one creates a new string. Most string problems reduce to scanning with one or two indices and keeping a small summary (counts, a set, or a window) of what you have seen.

Reverse, Palindrome, Anagram

```python
def reverse(s):            # O(n) time, O(n) space
    return s[::-1]

def is_palindrome(s):      # two pointers, O(n) time, O(1) extra space
    i, j = 0, len(s) - 1
    while i < j:
        if s[i] != s[j]:
            return False
        i += 1
        j -= 1
    return True

from collections import Counter
def is_anagram(a, b):      # O(n) time, O(k) space (k distinct chars)
    return Counter(a) == Counter(b)
```

For `"level"` the pointers compare l/l, e/e, then meet at v, so it is a palindrome. For `"listen"` and `"silent"` the character counts match, so they are anagrams.

Sliding Window: Longest Substring Without Repeating Characters

Intuition: keep a window `[left, right]` containing no duplicates. Expand `right`; when a duplicate appears, move `left` past its previous occurrence.

```python
def longest_unique(s):
    last, left, best = {}, 0, 0
    for right, ch in enumerate(s):
        if ch in last and last[ch] >= left:
            left = last[ch] + 1
        last[ch] = right
        best = max(best, right - left + 1)
    return best
```

Trace on `"abcabcbb"`:

right	char	left	window	best
0	a	0	a	1
1	b	0	ab	2
2	c	0	abc	3
3	a	1	bca	3
4	b	2	cab	3
5	c	3	abc	3
6	b	5	cb	3
7	b	7	b	3

Result: 3.

Complexity

Reverse and palindrome checks are O(n) time. The sliding window is O(n) time because each index moves forward only; space is O(min(n, alphabet size)).

Pitfalls

- Forgetting the `last[ch] >= left` check, which lets stale entries shrink the window wrongly.
- Ignoring case or punctuation in palindromes when the problem requires it.
- Building strings with repeated `+=` in loops; prefer `"".join`.

Practice

1. Valid Palindrome (ignore non-alphanumerics).
2. Group Anagrams.

Java/C++ note: Java strings are immutable, so use `StringBuilder` for in-place reversal; in C++, `std::string` is mutable and can be reversed in place.

HASHING: MAPS, SETS AND COLLISIONS

Concept and Intuition

A hash map stores key-value pairs; a hash set stores keys only. A hash function converts a key into an array index (a bucket), so lookup jumps straight to the right place instead of scanning. Think of a coat check: the ticket number tells you the hook without searching every coat. On average, insert, search and delete take O(1) time.

Worked Example: Two-Sum

Find indices of two numbers that add to a target. Brute force checks every pair in $O(n^2)$. With a map from value to index, we check whether `target - x` was already seen.

Trace for `nums = [2, 7, 11, 15]`, `target = 9`:

Step	x	Needed	Map before	Result
0	2	7	{}	store 2→0
1	7	2	{2:0}	found: [0, 1]

PYTHON

```python
def two_sum(nums, target):
    seen = {}
    for i, x in enumerate(nums):
        if target - x in seen:
            return [seen[target - x], i]
        seen[x] = i
    return []
```

Frequency Counting

PYTHON

```python
from collections import Counter

def freq(items):
    counts = {}
    for x in items:
        counts[x] = counts.get(x, 0) + 1
    return counts   # or Counter(items)
```

For `"banana"` the result is `{b:1, a:3, n:2}`. The same idea solves anagram checks and duplicate detection (a set suffices for the latter).

Collisions

Two keys can map to the same bucket. Two standard remedies:

- **Chaining:** each bucket holds a list of entries; colliding keys are appended.

- **Open addressing:** on collision, probe other slots (linear probing, quadratic probing, double hashing) until a free one is found.

Tables also resize (rehash) when the load factor grows, keeping chains or probe sequences short.

Complexity

Operation	Average	Worst
Insert / search / delete	O(1)	O(n)
Space	O(n)	O(n)

The worst case occurs when many keys collide in one bucket. Two-sum and frequency counting run in O(n) time and O(n) space.

Common Pitfalls

- Using mutable keys (Python lists) – they are unhashable.
- Inserting before checking in two-sum, which can pair an element with itself.
- Assuming iteration order or O(1) in the worst case.

Java/C++: `HashMap`/`HashSet` and `unordered_map`/`unordered_set`.

Practice

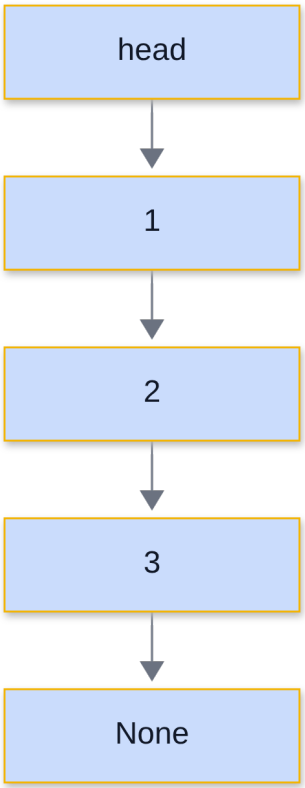
1. Contains Duplicate.
2. Group Anagrams.

LINKED LISTS: STRUCTURE AND REVERSAL

Concept and Intuition

A linked list stores items as separate **nodes**, each holding a value and a reference to another node. Unlike an array, nodes need not sit next to each other in memory, so you reach element *k* by following links from the head rather than by index.

- **Singly linked list:** each node has `val` and `next`. Traversal goes one way only.
- **Doubly linked list:** each node also has `prev`, so you can walk backward and delete a known node without searching for its predecessor. The cost is one extra pointer per node and more links to keep consistent.



In a doubly linked list, each arrow between neighbors exists in both directions (`next` forward, `prev` backward).

Node Definitions (Python)

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class DNode:
    def __init__(self, val=0):
        self.val = val
        self.prev = None
        self.next = None
```

In Java or C++ the structure is the same: a class or struct with a value and pointer or reference fields (`null` / `nullptr` instead of `None`).

Reversing a Singly Linked List

Idea: walk the list and flip each `next` pointer to point at the previous node. Use three names: `prev`, `curr`, and a saved `nxt`.

```python
def reverse(head):
    prev, curr = None, head
    while curr:
        nxt = curr.next   # 1. save the rest of the list
        curr.next = prev  # 2. flip the link
        prev = curr       # 3. advance prev
        curr = nxt        # 4. advance curr
    return prev           # new head
```

Trace for 1→2→3:

Step	prev	curr	List state
start	None	1	1→2→3
1	1	2	1→None, 2→3
2	2	3	2→1→None, 3
3	3	None	3→2→1→None

Complexity

Time O(n), extra space O(1). Access by position is O(n); insert or delete at the head is O(1).

Common Pitfalls

- Overwriting `curr.next` before saving it, which loses the rest of the list.
- Returning `head` instead of `prev`; the old head is now the tail.
- Forgetting to handle an empty list or a single node (the loop above handles both).
- In doubly linked lists, updating `next` but not `prev` (or the reverse).

Practice

1. Reverse a linked list recursively.
2. Reverse only the nodes between positions *left* and *right*.

LINKED LIST PATTERNS: CYCLES AND MERGING

Floyd's Cycle Detection

Concept. A linked list has a cycle if following `next` pointers never reaches `None`. Floyd's tortoise-and-hare algorithm detects this using two pointers: `slow` moves one node per step, `fast` moves two.

Intuition. On a cycle, the fast pointer gains one node on the slow pointer each step, so it must eventually land on it. If there is no cycle, `fast` reaches `None` first.

```python
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True
    return False
```

Trace. List 1→2→3→4, with 4 linking back to 2.

Step	slow	fast
start	1	1
1	2	3
2	3	2
3	4	4

At step 3 the pointers meet, so a cycle exists.

Merging Two Sorted Lists

Intuition. Use a dummy head node and a `tail` pointer. Repeatedly attach the smaller front node of the two lists, then attach whatever remains.

```python
def merge(a, b):
    dummy = tail = ListNode(0)
    while a and b:
        if a.val <= b.val:
            tail.next, a = a, a.next
        else:
            tail.next, b = b, b.next
        tail = tail.next
    tail.next = a or b
    return dummy.next
```

Trace. Merging 1→3→5 and 2→4: pick 1, 2, 3, 4, then the remainder 5, giving 1→2→3→4→5.

Complexity

Algorithm	Time	Space
Floyd detection	O(n)	O(1)
Merge (n and m nodes)	O(n + m)	O(1), since nodes are relinked

Common Pitfalls

- Checking only `fast.next` and not `fast`, which raises an error on `None`.
- Comparing values instead of node identity (`is`) when detecting a meeting.
- Forgetting to attach the leftover list after the loop.
- Using `<` rather than `<=`, which can break stability for equal values.

Practice Problems

1. Find the node where the cycle begins (Linked List Cycle II).
2. Merge Two Sorted Lists, then extend the idea to merging K sorted lists (covered later with heaps).

STACKS AND MONOTONIC STACKS

Concept and Intuition

A stack is a last-in, first-out (LIFO) structure: the most recently pushed item is the first one popped. Think of a pile of plates. In Python, a plain `list` works: `append` pushes, `pop` removes the top, and `stack[-1]` peeks. Push, pop and peek are all O(1).

Balanced Parentheses

Every closing bracket must match the most recent unmatched opening bracket, which is exactly what the top of a stack holds.

```python
def is_balanced(s):
    pairs = {')': '(', ']': '[', '}': '{'}
    stack = []
    for ch in s:
        if ch in '([{':
            stack.append(ch)
        elif ch in pairs:
            if not stack or stack.pop() != pairs[ch]:
                return False
    return not stack
```

Trace for `"{[()]}"`: push `{`, push `[`, push `(`; `)` pops `(` (match); `]` pops `[`; `}` pops `{`. The stack is empty, so the result is True. For `"(]"`, the `]` pops `(`, which does not match, so the result is False.

Next Greater Element and the Monotonic Stack

For each element, find the first larger value to its right. A brute-force scan is $O(n^2)$. A monotonic stack keeps indices whose values are in decreasing order; when a larger value arrives, it resolves every smaller value on top.

```python
def next_greater(nums):
    res = [-1] * len(nums)
    stack = []  # indices, values decreasing
    for i, x in enumerate(nums):
        while stack and nums[stack[-1]] < x:
            res[stack.pop()] = x
        stack.append(i)
    return res
```

Trace for `[2, 1, 5, 3]`:

i	x	Action	Stack (values)	res
0	2	push	[2]	[-1,-1,-1,-1]
1	1	push	[2,1]	unchanged
2	5	pop 1, pop 2, push	[5]	[5,5,-1,-1]
3	3	push	[5,3]	[5,5,-1,-1]

Complexity

- Balanced parentheses: O(n) time, O(n) space.
- Next greater element: O(n) time, because each index is pushed and popped at most once; O(n) space.

Common Pitfalls

- Popping from an empty stack; check `stack` first.
- Forgetting the final `not stack` check, which misses unclosed openers like `"((("`.
- Storing values instead of indices when you need positions or distances.
- Using the wrong comparison (< versus <=) when duplicates exist.

Java/C++ note: prefer `ArrayDeque` in Java and `std::stack` in C++.

Practice Problems

1. Valid Parentheses with three bracket types (variation: return the index of the first mismatch).
2. Daily Temperatures: days until a warmer temperature, using the monotonic stack.

QUEUES, DEQUES AND CIRCULAR QUEUES

Concept and Intuition

A **queue** is a first-in, first-out (FIFO) structure: items leave in the order they arrived, like people in a line. A **deque** (double-ended queue) allows insertion and removal at both ends. A **circular queue** is a fixed-capacity queue stored in an array whose indices wrap around, so freed slots at the front are reused.

Structure	Add	Remove	Best fit
Queue	back	front	BFS, task scheduling, buffering
Deque	both ends	both ends	sliding-window helpers, palindrome checks, undo/redo
Circular queue	back (if not full)	front	fixed-size buffers, streaming data

Worked Example: Circular Indexing

Use an array of capacity 4 with `head` (front index) and `count`. The tail slot is `(head + count) % 4`.

1. Enqueue A, B, C, D: they occupy indices 0–3, and `count = 4` (full).
2. Dequeue twice: `head = 2`, `count = 2`.
3. Enqueue E: slot `(2 + 2) % 4 = 0`, reusing index 0. Enqueue F goes to index 1.

Without the modulo, the queue would report full even though space is free.

Code (Python)

PYTHON

```python
from collections import deque

q = deque()          # queue or deque
q.append(1); q.popleft()      # FIFO use
q.appendleft(0); q.pop()      # deque use

class CircularQueue:
    def __init__(self, k):
        self.a = [None] * k
        self.head = 0
        self.count = 0
    def enqueue(self, x):
        if self.count == len(self.a): return False
        self.a[(self.head + self.count) % len(self.a)] = x
        self.count += 1
        return True
    def dequeue(self):
        if self.count == 0: return None
        x = self.a[self.head]
        self.head = (self.head + 1) % len(self.a)
        self.count -= 1
        return x
```

In Java, use `ArrayDeque` or `Queue`; in C++, use `std::queue` and `std::deque`.

Complexity

Enqueue, dequeue, and deque end operations are O(1) with `deque` or the circular array. Space is O(n). Avoid `list.pop(0)` in Python, which is O(n).

Common Pitfalls

- Using a plain list as a queue, making removal from the front slow.
- Confusing full and empty states; tracking `count` avoids this ambiguity.
- Forgetting the modulo when advancing `head`.

Practice

1. Implement a queue using two stacks.
2. Design a circular deque with fixed capacity.

SEARCHING: LINEAR, BINARY AND SEARCH ON ANSWER

Concept and Intuition

Linear search checks items one by one until the target is found or the data ends. It works on any collection, sorted or not. **Binary search** requires sorted data (or any monotonic condition). Each step compares the middle element with the target and discards half of the remaining range.

Binary search on answer applies the same idea when you are not searching an array but a range of candidate answers. If a yes/no test `feasible(x)` flips from false to true exactly once as `x` grows, you can binary search for the boundary.

Code

PYTHON

```python
def linear_search(a, t):
    for i, v in enumerate(a):
        if v == t:
            return i
    return -1

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

In Java and C++, `lo + (hi - lo) / 2` also avoids integer overflow, which Python does not suffer from.

Worked Trace

Search for 7 in `[1, 3, 5, 7, 9, 11]`:

Step	lo	hi	mid	a[mid]	Action
1	0	5	2	5	5 < 7, so lo = 3
2	3	5	4	9	9 > 7, so hi = 3
3	3	3	3	7	found, return 3

Binary Search on Answer

Example: find the smallest integer speed `k` such that a task can be finished within a limit, where a higher `k` never makes it harder. Define `feasible(k)`, then search for the first true value:

```python
def first_true(lo, hi, feasible):
    while lo < hi:
        mid = lo + (hi - lo) // 2
        if feasible(mid):
            hi = mid        # mid might be the answer
        else:
            lo = mid + 1    # answer is larger
    return lo
```

The loop keeps the answer inside `[lo, hi]` and shrinks until one candidate remains.

Complexity

Algorithm	Time	Space
Linear search	O(n)	O(1)
Binary search	O(log n)	O(1)
Search on answer	O(log R × cost of feasible)	O(1)

Here R is the size of the candidate range.

Common Pitfalls

- Using binary search on unsorted or non-monotonic data.
- Mixing `lo <= hi` loops with `hi = mid` updates, which can loop forever.
- Off-by-one errors: `mid - 1` and `mid + 1` belong with inclusive bounds; `hi = mid` belongs with `lo < hi`.
- Choosing a range that excludes the true answer.

Practice

1. Return the first and last position of a target in a sorted array with duplicates.
2. Compute the integer square root of `n` using search on answer.

BASIC SORTING ALGORITHMS

Concept and Intuition

Basic sorting algorithms are simple, quadratic-time methods that build intuition for how ordering works. All three below sort in place by comparing and moving elements.

- **Bubble sort** repeatedly swaps adjacent out-of-order pairs, so the largest unsorted value "bubbles" to the end each pass.
- **Selection sort** finds the minimum of the unsorted part and swaps it into the next position.
- **Insertion sort** grows a sorted prefix by inserting each new element into its correct place, like sorting playing cards in your hand.

Worked Trace

Input: `[5, 2, 4, 1]`

Algorithm	Step-by-step state
Bubble	Pass 1: `[2,4,1,5]` → Pass 2: `[2,1,4,5]` → Pass 3: `[1,2,4,5]`
Selection	Min 1 swapped to front: `[1,2,4,5]` (then remaining passes change nothing)
Insertion	Insert 2: `[2,5,4,1]` → insert 4: `[2,4,5,1]` → insert 1: `[1,2,4,5]`

Code (Python)

```python
def bubble_sort(a):
    n = len(a)
    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swapped = True
        if not swapped:
            break

def selection_sort(a):
    for i in range(len(a) - 1):
        m = i
        for j in range(i + 1, len(a)):
            if a[j] < a[m]:
                m = j
        a[i], a[m] = a[m], a[i]

def insertion_sort(a):
    for i in range(1, len(a)):
        key, j = a[i], i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
```

Java and C++ versions use the same loops with array indexing and a temporary variable for swaps.

Complexity and Stability

Algorithm	Best	Average	Worst	Space	Stable
Bubble (with early exit)	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No (swaps can reorder equal keys)
Insertion	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes

Common Pitfalls

- Forgetting the early-exit flag in bubble sort, losing its O(n) best case.
- Using >= in insertion sort's shift condition, which breaks stability.
- Off-by-one errors in inner loop bounds.

Practice Problems

1. Sort an array of records by key and verify that equal keys keep their original order.
2. Count the number of swaps bubble sort performs on a given array.

MERGE SORT AND QUICK SORT

Concept and Intuition

Both algorithms use divide and conquer: split the problem, solve the parts recursively, and combine the results. Merge sort does the easy split (the middle) and the hard work when combining. Quick sort does the hard work when splitting (partitioning around a pivot) and the combine step is free.

Merge Sort

Split the array in half, sort each half, then merge two sorted halves by repeatedly taking the smaller front element.

Trace for `[5, 2, 4, 1]`: split into `[5, 2]` and `[4, 1]`; these become `[2, 5]` and `[1, 4]`; merging compares 2 vs 1 (take 1), 2 vs 4 (take 2), 5 vs 4 (take 4), then 5, giving `[1, 2, 4, 5]`.

```python
def merge_sort(a):
    if len(a) <= 1:
        return a
    mid = len(a) // 2
    L, R = merge_sort(a[:mid]), merge_sort(a[mid:])
    out, i, j = [], 0, 0
    while i < len(L) and j < len(R):
        if L[i] <= R[j]:      # <= keeps it stable
            out.append(L[i]); i += 1
        else:
            out.append(R[j]); j += 1
    return out + L[i:] + R[j:]
```

Quick Sort

Choose a pivot, partition so smaller elements lie left and larger right, then recurse on both sides.

Lomuto partition trace for `[4, 7, 2, 5]`, pivot = 5 (last): scan 4 (≤5, swap into slot 0), 7 (skip), 2 (≤5, swap with 7 → `[4, 2, 7, 5]`). Place pivot at index 2: `[4, 2, 5, 7]`. Recurse on `[4, 2]` and `[7]`.

```python
def quick_sort(a, lo=0, hi=None):
    if hi is None: hi = len(a) - 1
    if lo < hi:
        p = a[hi]; i = lo
        for j in range(lo, hi):
            if a[j] <= p:
                a[i], a[j] = a[j], a[i]; i += 1
        a[i], a[hi] = a[hi], a[i]
        quick_sort(a, lo, i - 1)
        quick_sort(a, i + 1, hi)
```

Complexity

Algorithm	Best	Average	Worst	Extra space
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n) typical stack

Common Pitfalls

- A fixed last-element pivot on already sorted input gives $O(n^2)$; use a random or median-of-three pivot.
- Many equal keys can also degrade simple partitioning.
- Using < instead of <= in the merge breaks stability.
- Slicing in Python adds copying overhead.

Practice

1. Count inversions in an array using merge sort.
2. Find the k-th largest element with quickselect (partition only one side).

Java/C++: `Arrays.sort` on objects (Java) and `std::stable_sort` (C++) are merge-style stable sorts.

HEAP SORT, COUNTING SORT AND STABILITY TABLE

Heap Sort

Concept. Heap sort builds a max heap from the array, then repeatedly swaps the root (the largest element) with the last unsorted position and restores the heap on the shrunken range.

Intuition. A max heap always exposes the largest remaining value at index 0. Moving it to the end, one position at a time, builds the sorted suffix in place.

Trace on `[4, 10, 3, 5]`: heapify gives `[10, 5, 3, 4]`. Swap root with last: `[4, 5, 3, 10]`, sift down: `[5, 4, 3, 10]`. Swap: `[3, 4, 5, 10]`. Sorted.

```python
def heap_sort(a):
    n = len(a)
    def sift(i, size):
        while True:
            big, l, r = i, 2*i+1, 2*i+2
            if l < size and a[l] > a[big]: big = l
            if r < size and a[r] > a[big]: big = r
            if big == i: return
            a[i], a[big] = a[big], a[i]
            i = big
    for i in range(n//2 - 1, -1, -1):
        sift(i, n)
    for end in range(n-1, 0, -1):
        a[0], a[end] = a[end], a[0]
        sift(0, end)
```

Complexity. Time O(n log n) in all cases; space O(1). **Pitfall:** wrong child indices or forgetting to pass the reduced heap size to `sift`.

Counting Sort

Concept. For integers in a small range 0..k, count occurrences, then rebuild the output from the counts. No comparisons are made.

```python
def counting_sort(a, k):
    count = [0] * (k + 1)
    for x in a: count[x] += 1
    return [v for v in range(k + 1) for _ in range(count[v])]
```

Example: `[2, 1, 2, 0]` gives counts `[1, 1, 2]`, output `[0, 1, 2, 2]`. Time O(n + k), space O(k). **Pitfall:** a large k makes it impractical; negative values need an offset. This simple version does not preserve record order; the stable variant uses prefix sums of counts.

Complexity and Stability Table

Algorithm	Best	Average	Worst	Space	Stable
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Quick sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n) typical	No
Heap sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Counting sort	O(n + k)	O(n + k)	O(n + k)	O(k)	Yes (stable variant)

Practice

1. Sort an array with heap sort and trace the first three swaps.
2. Use counting sort on exam scores from 0 to 100.

RECURSION AND BACKTRACKING: SUBSETS AND PERMUTATIONS

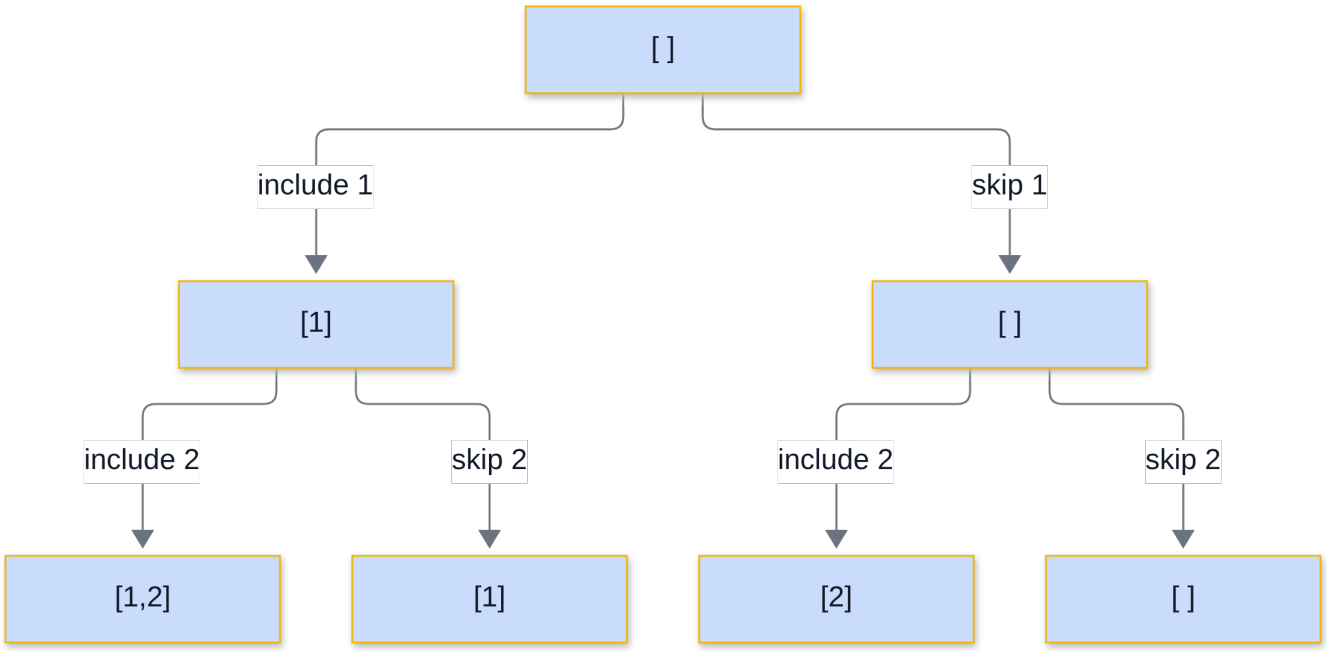
Backtracking is recursion that builds a candidate solution one choice at a time, and **undoes** each choice (the "backtrack") before trying the next. It systematically explores a decision tree of possibilities.

The Template

1. Keep a *path* (the choices made so far).
2. If the path is a complete answer, record a **copy** of it.
3. For each available choice: make it, recurse, then undo it.

Subsets

For `[1,2,3]`, each element is either included or excluded, giving 2^n subsets. The decision tree below shows the include/exclude choice for the first two elements.




```python
def subsets(nums):
    res, path = [], []
    def dfs(i):
        if i == len(nums):
            res.append(path[:])   # copy
            return
        path.append(nums[i])      # choose
        dfs(i + 1)                # explore
        path.pop()                # un-choose
        dfs(i + 1)                # skip
    dfs(0)
    return res
```

Permutations

Here every position may take any unused element, giving n! results. A `used` array marks elements already in the path.

```python
def permutations(nums):
    res, path = [], []
    used = [False] * len(nums)
    def dfs():
        if len(path) == len(nums):
            res.append(path[:])
            return
        for i, x in enumerate(nums):
            if used[i]:
                continue
            used[i] = True; path.append(x)
            dfs()
            path.pop(); used[i] = False
    dfs()
    return res
```

Trace for `[1,2,3]`: choose 1, then 2, then 3 → record `[1,2,3]`; backtrack to 2's level, choose 3 then 2 → `[1,3,2]`; and so on.

Complexity

Problem	Results	Time	Extra space
Subsets	2^n	O(n·2^n)	O(n) recursion
Permutations	n!	O(n·n!)	O(n)

The factor n comes from copying each result.

Common Pitfalls

- Appending `path` instead of `path[:]`, so all results alias one list.
- Forgetting to undo state (`pop`, `used[i] = False`).
- Duplicate inputs produce duplicate results unless you sort and skip repeats.

Practice

1. Generate subsets of a list that contains duplicates, without duplicate results.
2. Generate all permutations of a string.

BACKTRACKING: N-QUEENS AND SUDOKU

Concept

N-Queens asks for a placement of N queens on an N×N board so that no two attack each other: no shared row, column, or diagonal. Sudoku asks for a grid filled so every row, column, and box contains each digit once. Both are constraint-satisfaction problems solved by backtracking: make a choice, check constraints, recurse, and undo the choice on failure.

Intuition

Place one queen per row. This removes row conflicts by construction, so only columns and diagonals need checking. Pruning means rejecting a partial placement the moment it violates a constraint, instead of completing it and then discovering the failure. Cells on the same diagonal share `row - col`; cells on the same anti-diagonal share `row + col`. Sets of used columns and diagonals give constant-time checks.

Worked trace (N = 4)

- Row 0: try column 0. Safe.
- Row 1: columns 0 and 1 are attacked; column 2 is safe.
- Row 2: every column is attacked, so backtrack and move row 1 to column 3.
- Row 2: column 1 is safe. Row 3: no safe column, so backtrack further.
- Eventually row 0 moves to column 1, giving a full solution.

Code (Python)

```python
def solve_n_queens(n):
    cols, d1, d2 = set(), set(), set()
    board, res = [], []
    def place(r):
        if r == n:
            res.append(board[:]); return
        for c in range(n):
            if c in cols or r-c in d1 or r+c in d2:
                continue          # prune
            cols.add(c); d1.add(r-c); d2.add(r+c)
            board.append(c)
            place(r+1)
            board.pop()
            cols.remove(c); d1.remove(r-c); d2.remove(r+c)
    place(0)
    return res
```

The sudoku idea

Find an empty cell, try digits 1–9, keep a digit only if the row, column, and 3×3 box allow it, then recurse; if no digit works, reset the cell and return failure to the caller. A useful suggestion is to choose the empty cell with the fewest legal digits first, which prunes much more.

Complexity

Unpruned N-Queens is exponential; pruning shrinks the search sharply but the worst case stays exponential. Space is O(N) for the recursion and sets. Sudoku is likewise exponential in the worst case.

Pitfalls

- Forgetting to undo state after recursion.
- Recording the live `board` list as a solution instead of a copy (`board[:]`), so later backtracking changes saved results.
- Mixing up the two diagonal formulas.

Practice

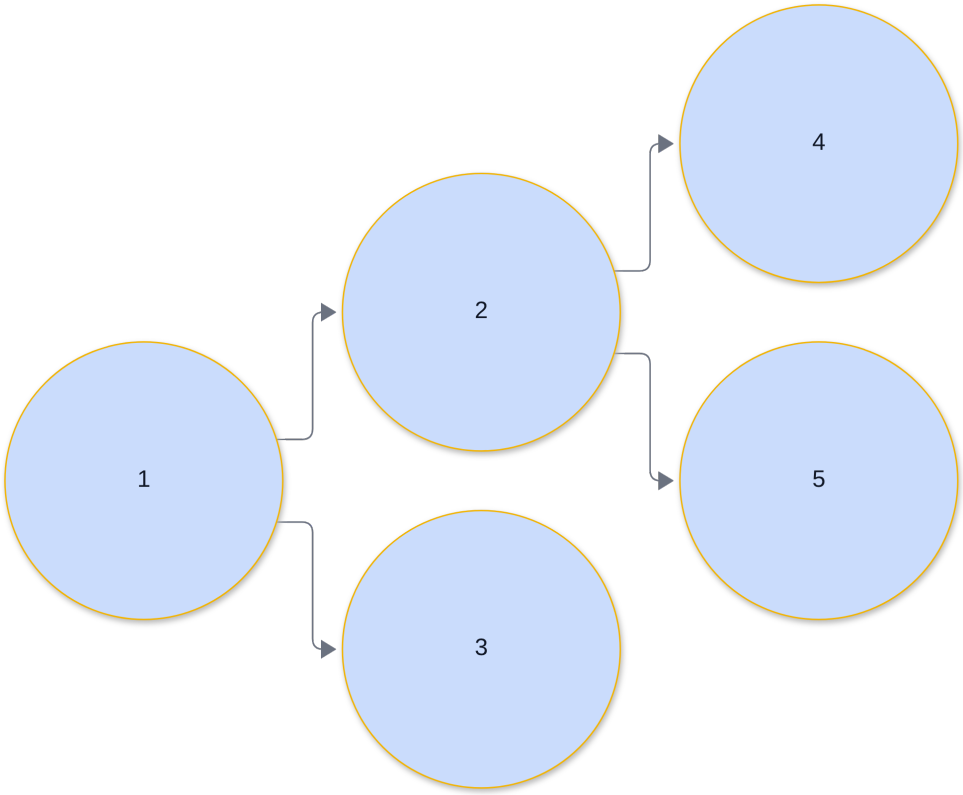
1. Count the solutions for N = 6.
2. Write a sudoku validity checker, then extend it into a solver.

BINARY TREES AND TRAVERSALS

Concept and Intuition

A binary tree is a hierarchical structure in which each node holds a value and at most two children, called left and right. The topmost node is the root; a node with no children is a leaf. Traversal means visiting every node exactly once in a defined order. The four standard orders are preorder (node, left, right), inorder (left, node, right), postorder (left, right, node) and level order (row by row, top to bottom).

Worked Example



For this tree the traversals produce:

Traversal	Order	Output
Preorder	node, left, right	1 2 4 5 3
Inorder	left, node, right	4 2 5 1 3
Postorder	left, right, node	4 5 2 3 1
Level order	by depth	1 2 3 4 5

Code (Python)

```python
from collections import deque

class Node:
    def __init__(self, val, left=None, right=None):
        self.val, self.left, self.right = val, left, right

def inorder(n, out):            # recursive
    if n:
        inorder(n.left, out); out.append(n.val); inorder(n.right, out)

def preorder(n, out):
    if n:
        out.append(n.val); preorder(n.left, out); preorder(n.right, out)

def postorder(n, out):
    if n:
        postorder(n.left, out); postorder(n.right, out); out.append(n.val)

def inorder_iter(root):         # iterative, explicit stack
    out, stack, cur = [], [], root
    while cur or stack:
        while cur:
            stack.append(cur); cur = cur.left
        cur = stack.pop()
        out.append(cur.val)
        cur = cur.right
    return out

def level_order(root):          # queue-based
    out, q = [], deque([root] if root else [])
    while q:
        n = q.popleft(); out.append(n.val)
        if n.left: q.append(n.left)
        if n.right: q.append(n.right)
    return out
```

Preorder and postorder iterative versions follow the same stack idea. Java and C++ versions use `Deque`/`std::stack` and `Queue`/`std::queue` in the same way.

Complexity

Every traversal visits each of the n nodes once, so time is O(n). Recursive space is O(h), where h is the tree height (O(log n) if balanced, O(n) if skewed); level order uses O(w) space, where w is the maximum width.

Common Pitfalls

- Forgetting the `None` base case.
- Confusing preorder with postorder placement of the visit step.
- Not advancing to `cur.right` in iterative inorder.

- Starting level order with a null root.

Practice

1. Return the postorder traversal iteratively.
2. Return level order as a list of lists, one per level.

TREE HEIGHT, DIAMETER AND LCA

Concept and Intuition

Three common tree questions build on recursion over subtrees. **Height** is the number of edges (or nodes, by convention) on the longest path from a node down to a leaf. **Diameter** is the longest path between any two nodes; it need not pass through the root. The **lowest common ancestor (LCA)** of nodes p and q is the deepest node that has both as descendants (a node counts as its own descendant). In edge counting, the height of an empty tree is -1; the code below uses node counts, so an empty tree is 0.

Worked Example

Consider a tree: 1 has children 2 and 3; 2 has children 4 and 5.

- Height (nodes): nodes 4 and 5 give 1, node 2 gives 2, node 3 gives 1, root gives 3.
- Diameter: at node 2, left height 1 + right height 1 = 2 edges (4-2-5). At root, left height 2 + right height 1 = 3 edges (4-2-1-3). The answer is 3.
- LCA(4,5) = 2, because they lie in different subtrees of 2. LCA(4,3) = 1. LCA(2,4) = 2.

Code

PYTHON

```python
def height(node):
    if not node: return 0
    return 1 + max(height(node.left), height(node.right))

def diameter(root):
    best = 0
    def h(node):
        nonlocal best
        if not node: return 0
        l, r = h(node.left), h(node.right)
        best = max(best, l + r)   # edges through this node
        return 1 + max(l, r)
    h(root)
    return best

def lca(node, p, q):
    if not node or node is p or node is q:
        return node
    l = lca(node.left, p, q)
    r = lca(node.right, p, q)
    return node if l and r else (l or r)
```

The LCA logic assumes both nodes exist in the tree. Java and C++ versions are structurally identical, using a `TreeNode` class or struct.

Complexity

Operation	Time	Space
Height	O(n)	O(h) recursion
Diameter (single pass)	O(n)	O(h)
LCA (binary tree)	O(n)	O(h)

Here h is tree height: O(log n) if balanced, O(n) if skewed.

Common Pitfalls

- Recomputing height at every node inside diameter, which gives $O(n^2)$.
- Mixing node-count and edge-count conventions.
- Assuming the diameter passes through the root.
- Using the LCA code when p or q may be missing.

Practice Problems

1. Maximum Depth of Binary Tree.
2. Diameter of Binary Tree.
3. Lowest Common Ancestor of a Binary Tree.

BINARY SEARCH TREES: INSERT, SEARCH, DELETE, VALIDATE

Concept and Intuition

A binary search tree (BST) is a binary tree where, for every node, all keys in the left subtree are smaller and all keys in the right subtree are larger (this note assumes distinct keys). Each comparison discards one subtree, much like binary search on a sorted array. An inorder traversal of a BST visits keys in sorted order, which is also the basis for validation.

Worked Example: Insert and Search

Insert 8, 3, 10, 1, 6, 14 in that order. 8 is the root; 3 goes left; 10 goes right; 1 goes left of 3; 6 goes right of 3; 14 goes right of 10. Searching for 6: compare with 8 (go left), with 3 (go right), find 6.

PYTHON

```python
class Node:
    def __init__(self, key):
        self.key, self.left, self.right = key, None, None

def insert(root, key):
    if not root: return Node(key)
    if key < root.key: root.left = insert(root.left, key)
    elif key > root.key: root.right = insert(root.right, key)
    return root

def search(root, key):
    while root and root.key != key:
        root = root.left if key < root.key else root.right
    return root
```

Delete: Three Cases

1. **Leaf:** remove it. Deleting 1 simply clears 3's left link.
2. **One child:** replace the node with its child. Deleting 10 (after 14 is its only child) promotes 14.
3. **Two children:** copy the inorder successor (smallest key in the right subtree) into the node, then delete that successor from the right subtree. Deleting 3 (children 1 and 6): the successor is 6, so 3 becomes 6, and the old 6 leaf is removed.

PYTHON

```python
def delete(root, key):
    if not root: return None
    if key < root.key: root.left = delete(root.left, key)
    elif key > root.key: root.right = delete(root.right, key)
    else:
        if not root.left: return root.right
        if not root.right: return root.left
        s = root.right
        while s.left: s = s.left
        root.key = s.key
        root.right = delete(root.right, s.key)
    return root
```

Validate

Checking only a node against its direct children is wrong. Pass down an allowed range:

```python
def is_valid(n, lo=float('-inf'), hi=float('inf')):
    if not n: return True
    if not (lo < n.key < hi): return False
    return is_valid(n.left, lo, n.key) and is_valid(n.right, n.key, hi)
```

Complexity

Operation	Balanced	Skewed
Insert/Search/Delete	O(log n)	O(n)
Validate	O(n)	O(n)

Space is O(h) for recursion, where h is the height.

Pitfalls

- Validating only parent–child pairs.
- Forgetting to reassign returned subtrees in recursion.
- Sorted insertion order produces a skewed tree.

Practice

- Find the kth smallest element in a BST.
- Convert a sorted array into a height-balanced BST.

HEAPS AND PRIORITY QUEUES

Concept

A **heap** is a complete binary tree stored in an array that satisfies the heap property. In a **min-heap** every parent is less than or equal to its children, so the smallest element sits at the root. In a **max-heap** every parent is greater than or equal to its children, so the largest sits at the root. A **priority queue** is the abstract structure (insert, peek, remove the best element); a heap is its standard implementation.

Intuition

Because the tree is complete, no pointers are needed. With 0-based indexing, node `i` has children at `2i+1` and `2i+2`, and its parent is at `(i-1)//2`. Each operation repairs the property along a single root-to-leaf path, so it costs the tree height, O(log n).

Worked trace: heapify

Build a min-heap from `[9, 5, 6, 2, 3]`. Start at the last non-leaf index, `n//2 - 1 = 1`, and sift each node down.

Step	Node	Action	Array
0	–	start	[9, 5, 6, 2, 3]
1	i=1 (5)	children 2, 3; smallest is 2; swap	[9, 2, 6, 5, 3]
2	i=0 (9)	children 2, 6; swap with 2	[2, 9, 6, 5, 3]
3	i=1 (9)	children 5, 3; swap with 3	[2, 3, 6, 5, 9]

The result is a valid min-heap.

Code

```python
def sift_down(a, i, n):
    while True:
        s, l, r = i, 2*i+1, 2*i+2
        if l < n and a[l] < a[s]: s = l
        if r < n and a[r] < a[s]: s = r
        if s == i: return
        a[i], a[s] = a[s], a[i]
        i = s

def build_min_heap(a):
    for i in range(len(a)//2 - 1, -1, -1):
        sift_down(a, i, len(a))
```

Python's `heapq` provides a min-heap; Java uses `PriorityQueue` and C++ uses `priority_queue` (a max-heap by default).

Why building a heap is O(n)

Sifting a node down costs at most its height. In a heap of n nodes, about n/2 nodes are leaves (height 0), n/4 have height 1, n/8 have height 2, and so on. The total work is proportional to the sum over h of (n / 2^(h+1)) · h, and the series Σ h/2^h converges to a constant, so the total is O(n), not O(n log n). Inserting elements one at a time, by contrast, costs O(n log n).

Complexity

Operation	Time
Peek	O(1)
Insert / remove root	O(log n)
Build heap	O(n)

Space is O(1) extra when heapifying in place.

Pitfalls

- Looping heapify from index 0 upward instead of from the last non-leaf downward.
- Forgetting bounds checks on child indexes.
- Expecting the array to be sorted; only the parent-child order is guaranteed.

Practice

1. Heapify `[7, 1, 8, 3, 4, 2]` into a min-heap and write each step.
2. Convert the min-heap code above into a max-heap.

HEAP APPLICATIONS: TOP-K AND MERGE K SORTED LISTS

Concept and Intuition

Two classic heap applications avoid sorting everything. **Top-K** asks for the K largest (or smallest) items in a collection. **Merge K sorted lists** combines K already-sorted sequences into one sorted sequence. In both cases a small heap holds only the candidates that matter right now.

Top-K Elements

To find the K largest values, keep a **min-heap of size K**. The root is the smallest of the current top K, so it is the first to be evicted. For each new value, push it; if the heap exceeds K items, pop the root.

Trace for `nums = [5, 1, 8, 3, 9]`, K = 3: heap becomes [5], [1,5], [1,5,8]; add 3 and pop 1 → [3,5,8]; add 9 and pop 3 → [5,8,9]. The answer is {5, 8, 9}.

PYTHON

```python
import heapq

def top_k_largest(nums, k):
    heap = []
    for x in nums:
        heapq.heappush(heap, x)
        if len(heap) > k:
            heapq.heappop(heap)
    return sorted(heap, reverse=True)
```

Merge K Sorted Lists

Place the head of each list in a min-heap as a tuple `(value, list_index, element_index)`. Repeatedly pop the smallest, append it to the output, and push the next element from the same list. The index in the tuple breaks ties so Python never compares unlike objects.

PYTHON

```python
def merge_k_sorted(lists):
    heap, out = [], []
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst[0], i, 0))
    while heap:
        val, i, j = heapq.heappop(heap)
        out.append(val)
        if j + 1 < len(lists[i]):
            heapq.heappush(heap, (lists[i][j+1], i, j+1))
    return out
```

Heap vs. Naive Sorting

Task	Naive sort	Heap-based	Heap space
Top-K of n items	O(n log n)	O(n log k)	O(k)
Merge K lists, N total items	O(N log N)	O(N log k)	O(k)

The heap wins when K is much smaller than n, and it works on streams because it never needs all data at once.

Common Pitfalls

- Using a max-heap of size K for top-K largest; a min-heap is needed so the weakest candidate sits at the root.
- Python's `heapq` is a min-heap only; negate values for max-heap behavior.
- Omitting the tie-breaking index in merge tuples.

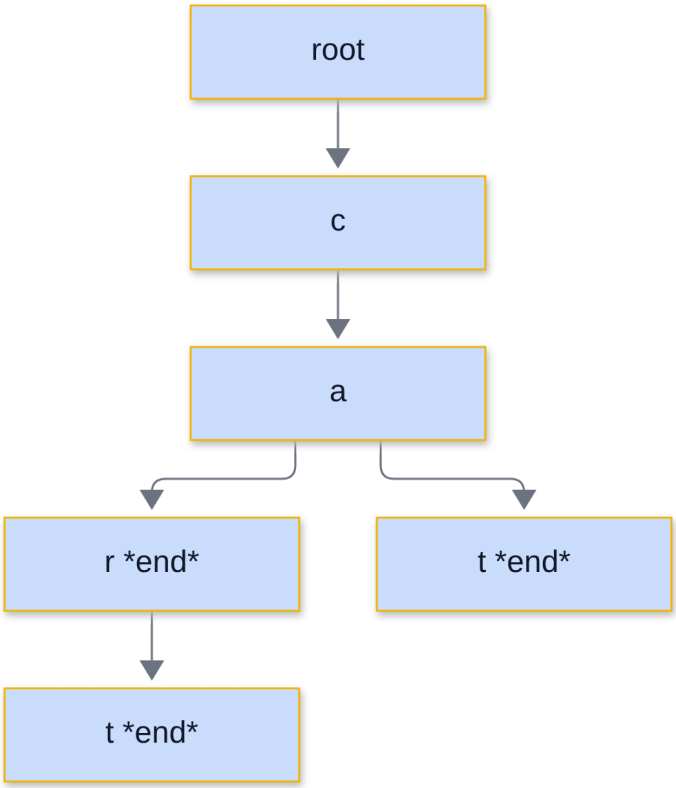
Practice Problems

1. Kth Largest Element in an Array.
2. Merge k Sorted Lists.

TRIES: PREFIX SEARCH AND AUTOCOMPLETE

Concept and Intuition

A trie (prefix tree) stores a set of strings by sharing common prefixes. Each node represents one character along a path from the root; a flag on a node marks that a complete word ends there. Words such as `car`, `cart` and `cat` share the path `c-a`, then branch. Because lookups follow one character at a time, cost depends on the word length, not on how many words are stored.



Worked Example

Insert `car`, `cart`, `cat`. Inserting `car` creates nodes c, a, r and marks r as a word end. Inserting `cart` reuses c-a-r and adds t. Inserting `cat` reuses c-a and adds a new t branch.

- `search("ca")`: path exists, but the end flag is not set, so the result is False.
- `startsWith("ca")`: path exists, so the result is True.
- `autocomplete("car")`: walk to node r, then collect words below it: `car`, `cart`.

Code (Python)

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:
            node = node.children.setdefault(ch, TrieNode())
        node.is_end = True

    def _walk(self, s):
        node = self.root
        for ch in s:
            if ch not in node.children:
                return None
            node = node.children[ch]
        return node

    def search(self, word):
        n = self._walk(word)
        return n is not None and n.is_end

    def starts_with(self, prefix):
        return self._walk(prefix) is not None

    def autocomplete(self, prefix):
        out, start = [], self._walk(prefix)
        def dfs(node, path):
            if node.is_end:
                out.append(path)
            for ch in sorted(node.children):
                dfs(node.children[ch], path + ch)
        if start:
            dfs(start, prefix)
        return out
```

In Java or C++, children are often an array of 26 pointers for lowercase letters instead of a dictionary.

Complexity

Operation	Time	Notes
insert / search / prefix	O(L)	L = string length
autocomplete	O(L + size of subtree)	visits all matching nodes
Space	O(total characters) worst case	prefix sharing reduces it

Common Pitfalls

- Treating a found prefix as a found word; always check the end flag.
- Forgetting to mark the end flag when inserting.
- Fixed 26-slot arrays waste memory for sparse data.

Practice

1. Implement Trie (insert, search, startsWith).
2. Design an autocomplete that returns at most three suggestions for a prefix.

BIT MANIPULATION ESSENTIALS

Concept and Intuition

Bit manipulation works directly on the binary representation of integers. Operations are single machine instructions, so they are fast and use almost no extra memory. The core operators in Python are & (AND), | (OR), ^ (XOR), ~ (NOT), << and >> (shifts).

XOR is the key tool: `a ^ a = 0`, `a ^ 0 = a`, and XOR is commutative and associative. Equal values therefore cancel in any order.

XOR Trick: Single Number

In an array where every value appears twice except one, XOR all elements; the pairs cancel and the unique value remains.

Trace for `[4, 1, 2, 1, 2]`: 0^4=4, 4^1=5, 5^2=7, 7^1=6, 6^2=4. Result: 4.

```python
def single_number(nums):
    result = 0
    for x in nums:
        result ^= x
    return result
```

Power of Two

A positive power of two has exactly one set bit, for example `8 = 1000`. Subtracting 1 flips that bit and sets all lower bits (`0111`), so `n & (n - 1)` clears the lowest set bit and gives 0 only in this case.

```python
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0
```

Counting Set Bits (Brian Kernighan)

Repeatedly clear the lowest set bit and count how many iterations are needed.

```python
def count_set_bits(n):
    count = 0
    while n:
        n &= n - 1
        count += 1
    return count
```

For `13 = 1101`: 1101 → 1100 → 1000 → 0000, so the count is 3.

Complexity

Task	Time	Space
Single number	O(n)	O(1)
Power of two	O(1)	O(1)
Count set bits	O(k), k = set bits	O(1)

Java and C++ use the same operators; in those languages integers have fixed width, which matters for ~ and shifts.

Common Pitfalls

- Forgetting `n > 0` in the power-of-two check, since `0 & -1` is 0.
- Operator precedence: write `(n & (n - 1)) == 0` with parentheses.
- Python integers are unbounded, so `~x` yields a negative number (`-x - 1`); mask if you need fixed width.
- The single-number trick requires every other value to appear exactly twice.

Practice Problems

1. Single Number: solve it with XOR, then explain why it fails if duplicates appear three times.
2. Number of 1 Bits: implement it with and without Kernighan's method.

GRAPHS: REPRESENTATION

Concept

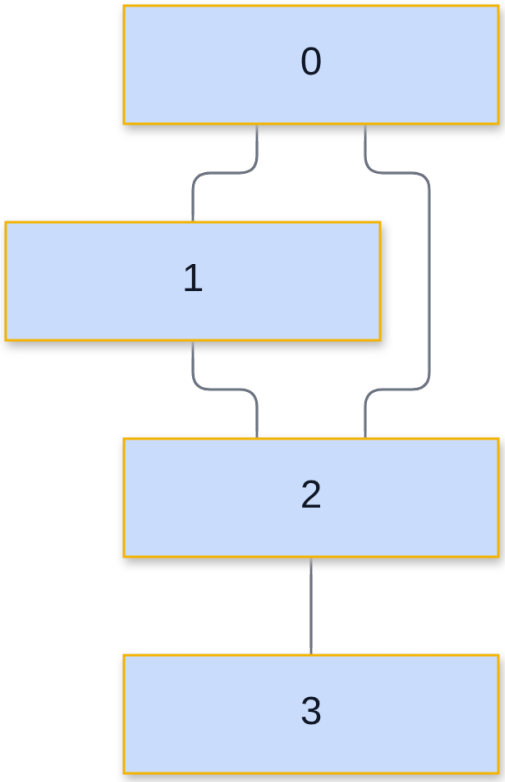
A graph is a set of **vertices** (nodes) connected by **edges**. Edges may be *directed* (one-way) or *undirected* (two-way), and may carry *weights* such as distance or cost. Before running BFS, DFS, or shortest-path algorithms, you must choose how to store the graph. The two standard choices are the adjacency list and the adjacency matrix.

Intuition

Think of a social network. An **adjacency list** is each person's contact list: it records only the friends that exist. An **adjacency matrix** is a full attendance grid with every pair of people, marked yes or no. The list is compact when connections are few; the grid answers "are A and B connected?" instantly.

Worked Example

Consider an undirected graph with 4 vertices (0 to 3) and edges 0-1, 0-2, 1-2, 2-3.



Adjacency list: 0 → [1, 2]; 1 → [0, 2]; 2 → [0, 1, 3]; 3 → [2].

Adjacency matrix (1 = edge present):

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Because the graph is undirected, the matrix is symmetric.

Code (Python)

```python
from collections import defaultdict

n = 4
edges = [(0, 1), (0, 2), (1, 2), (2, 3)]

adj = defaultdict(list)          # adjacency list
mat = [[0] * n for _ in range(n)]  # adjacency matrix
for u, v in edges:
    adj[u].append(v); adj[v].append(u)
    mat[u][v] = mat[v][u] = 1
```

For weighted graphs, store `(neighbor, weight)` pairs in the list, or the weight in the matrix cell. In Java use `List<List<Integer>>`; in C++ use `vector<vector<int>>`.

Trade-offs

Aspect	Adjacency list	Adjacency matrix
Space	O(V + E)	$O(V^2)$
Edge exists? (u, v)	O(deg(u))	O(1)
Iterate neighbors	O(deg(u))	O(V)
Best for	Sparse graphs	Dense graphs, Floyd-Warshall

Common Pitfalls

- Adding only one direction of an undirected edge.
- Using a matrix for a huge sparse graph, wasting memory.
- Forgetting that vertex labels may not be 0..n-1; map them to indices first.

Practice

1. Build both representations for a directed graph from an edge list.
2. Compute the in-degree and out-degree of every vertex using each representation.

BFS, DFS AND CONNECTED COMPONENTS

Concept and Intuition

Breadth-first search (BFS) visits vertices in order of distance from the start, using a queue. Depth-first search (DFS) follows one path as far as it can, then backtracks, using recursion (the call stack) or an explicit stack. Both visit every reachable vertex once, so each runs in O(V + E) time on an adjacency list, with O(V) extra space for the visited set and the queue or stack.

Worked Example: One Graph, Two Traversals

Use an undirected graph with adjacency lists (neighbors in ascending order):

```
1: [2, 3]
2: [1, 4]
3: [1, 4]
4: [2, 3]
5: [6]
6: [5]
```

Traversal from 1	Order visited	Trace idea
BFS	1, 2, 3, 4	Queue: [1] → [2,3] → [3,4] → [4]
DFS	1, 2, 4, 3	Go 1→2→4, then 4→3 (via 4's neighbor), then backtrack

Vertices 5 and 6 are never reached from 1. They form a second connected component.

Code (Python)

```python
from collections import deque

def bfs(graph, start, seen):
    order, q = [], deque([start])
    seen.add(start)
    while q:
        u = q.popleft()
        order.append(u)
        for v in graph[u]:
            if v not in seen:
                seen.add(v)
                q.append(v)
    return order

def dfs(graph, u, seen, order):
    seen.add(u)
    order.append(u)
    for v in graph[u]:
        if v not in seen:
            dfs(graph, v, seen, order)

def components(graph):
    seen, result = set(), []
    for node in graph:
        if node not in seen:
            order = []
            dfs(graph, node, seen, order)
            result.append(order)
    return result
```

On the example graph, `components` returns `[[1, 2, 4, 3], [5, 6]]`: two components.

When Each Fits

- **BFS:** shortest path in unweighted graphs, fewest steps, level-by-level processing.
- **DFS:** connectivity, cycle detection, topological sort, exhaustive path exploration.
- **Components:** either works; start a new traversal from each unvisited vertex.

Pitfalls

- Marking visited when *popping* instead of when *enqueuing* in BFS can add duplicates.
- Deep recursion in DFS can overflow Python's call stack; use an explicit stack for large inputs.
- Forgetting to loop over all vertices misses disconnected components.

Java/C++ note: use `ArrayDeque`/`std::queue` for BFS; the logic is identical.

Practice

1. Count connected components in an undirected graph.
2. Find the minimum number of edges between two vertices using BFS.

CYCLE DETECTION AND TOPOLOGICAL SORT

Concept and Intuition

A **cycle** is a path that returns to its starting vertex. Cycle detection differs by graph type. In an **undirected** graph, a cycle exists if a DFS meets an already-visited vertex that is not the node it just came from. In a **directed** graph, a cycle exists only if DFS meets a vertex that is still *on the current recursion path*. Merely seeing a visited vertex is not enough, because two branches can legitimately reach the same node.

A **topological sort** orders the vertices of a directed acyclic graph (DAG) so every edge u → v places u before v. Think of course prerequisites or build tasks. If a cycle exists, no valid order exists, so topological sort doubles as a cycle detector.

Worked Example

Tasks 0..3 with edges 0→1, 0→2, 1→3, 2→3. In-degrees: 0:0, 1:1, 2:1, 3:2.

1. Queue = [0]. Pop 0, output 0; in-degrees of 1 and 2 drop to 0, so both are enqueued.
2. Pop 1, output 1; node 3 drops to 1.
3. Pop 2, output 2; node 3 drops to 0, enqueue.
4. Pop 3. Order: 0, 1, 2, 3.

If the output has fewer vertices than the graph, a cycle blocked the rest.

Code

```python
from collections import deque

def has_cycle_undirected(n, adj):
    seen = [False] * n
    def dfs(u, parent):
        seen[u] = True
        for v in adj[u]:
            if not seen[v]:
                if dfs(v, u): return True
            elif v != parent:
                return True
        return False
    return any(not seen[s] and dfs(s, -1) for s in range(n))

def has_cycle_directed(n, adj):
    state = [0] * n  # 0 new, 1 on path, 2 done
    def dfs(u):
        state[u] = 1
        for v in adj[u]:
            if state[v] == 1: return True
            if state[v] == 0 and dfs(v): return True
        state[u] = 2
        return False
    return any(state[s] == 0 and dfs(s) for s in range(n))

def topo_kahn(n, adj):
    indeg = [0] * n
    for u in range(n):
        for v in adj[u]: indeg[v] += 1
    q = deque(i for i in range(n) if indeg[i] == 0)
    order = []
    while q:
        u = q.popleft(); order.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0: q.append(v)
    return order if len(order) == n else []

def topo_dfs(n, adj):
    seen, order = [False] * n, []
    def dfs(u):
        seen[u] = True
        for v in adj[u]:
            if not seen[v]: dfs(v)
        order.append(u)  # post-order
    for s in range(n):
        if not seen[s]: dfs(s)
    return order[::-1]
```

The DFS version assumes a DAG; run the directed cycle check first.

Complexity

All four routines run in O(V + E) time with O(V) extra space (queue, state array, or recursion stack).

Common Pitfalls

- Using the undirected parent check on directed graphs, or plain visited flags for directed cycles.
- Forgetting to start from every unvisited vertex in disconnected graphs.
- Not reversing the DFS post-order.
- Parallel edges in undirected multigraphs need edge-ID tracking rather than a parent-vertex check.

Practice

1. Course Schedule (can all courses be finished?).
2. Course Schedule II (return a valid order).

SHORTEST PATHS: DIJKSTRA, BELLMAN-FORD, FLOYD-WARSHALL

Concept and Intuition

Shortest-path algorithms find minimum-cost routes in a weighted graph. All three methods here rely on **relaxation**: if `dist[u] + w(u,v) < dist[v]`, update `dist[v]`. They differ in which graphs they accept and how many relaxations they perform.

Algorithm	Problem solved	Graph conditions	Time	Space
Dijkstra (binary heap)	Single source	Non-negative weights	O((V+E) log V)	O(V)
Bellman-Ford	Single source	Negative edges allowed; detects negative cycles	O(V·E)	O(V)
Floyd-Warshall	All pairs	Negative edges allowed, no negative cycles	$O(V^3)$	$O(V^2)$

Worked Relaxation (Dijkstra)

Take edges A→B 4, A→C 1, C→B 2, B→D 1, source A. Start with dist A=0, others ∞.

1. Pop A (0): relax B=4, C=1.
2. Pop C (1): B becomes 1+2=3 (improved from 4).
3. Pop B (3): D=4.
4. Pop D (4): done.

Once a node is popped, its distance is final, which is valid only because weights are non-negative.

Code (Python)

```python
import heapq

def dijkstra(graph, src):
    dist = {u: float('inf') for u in graph}
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue  # stale entry
        for v, w in graph[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                heapq.heappush(pq, (dist[v], v))
    return dist

def bellman_ford(n, edges, src):
    dist = [float('inf')] * n
    dist[src] = 0
    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            return None  # negative cycle
    return dist

def floyd_warshall(d):  # d: V x V matrix, d[i][i]=0
    n = len(d)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if d[i][k] + d[k][j] < d[i][j]:
                    d[i][j] = d[i][k] + d[k][j]
    return d
```

Intuition for the Others

- **Bellman-Ford:** a shortest path has at most V−1 edges, so V−1 rounds of relaxing every edge suffice. An improvement in round V signals a negative cycle.
- **Floyd-Warshall:** dynamic programming over allowed intermediate vertices; after iteration k, paths may pass through vertices 0..k.

Common Pitfalls

- Using Dijkstra with negative edges gives wrong answers.
- Forgetting the stale-entry check wastes work.
- Floyd-Warshall requires the k loop outermost.
- Adding to ∞ in other languages may overflow; use a safe sentinel.

Practice

1. Network Delay Time (Dijkstra).
2. Cheapest Flights Within K Stops (Bellman-Ford style, limited rounds).

MINIMUM SPANNING TREES: KRUSKAL AND PRIM

Concept

A **minimum spanning tree (MST)** of a connected, weighted, undirected graph is a set of V−1 edges that connects every vertex, contains no cycle, and has the smallest possible total weight. Both algorithms below are greedy: they repeatedly take the cheapest edge that is safe to add.

Intuition

The *cut property* justifies the greed: for any split of the vertices into two groups, the cheapest edge crossing the split belongs to some MST. Kruskal grows a forest by sorting edges; Prim grows a single tree outward from a start vertex.

Worked example

Graph with vertices A, B, C, D and edges: A-B 1, B-C 2, A-C 3, C-D 4, B-D 5.

Kruskal (sort edges, skip any that close a cycle, checked with Union-Find):

Edge	Weight	Decision
A-B	1	take
B-C	2	take
A-C	3	skip (A and C already connected)
C-D	4	take

Total weight = 7, using 3 edges (V−1).

Prim from A: the cheapest edge leaving {A} is A-B (1); leaving {A,B} is B-C (2); leaving {A,B,C} is C-D (4). The result is the same tree.

Code (Python)

```python
def kruskal(n, edges):          # edges: (w, u, v)
    parent = list(range(n))
    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x
    total, used = 0, []
    for w, u, v in sorted(edges):
        ru, rv = find(u), find(v)
        if ru != rv:
            parent[ru] = rv
            total += w; used.append((u, v, w))
    return total, used

import heapq
def prim(adj, start=0):         # adj[u] = [(w, v), ...]
    seen, heap, total = set(), [(0, start)], 0
    while heap:
        w, u = heapq.heappop(heap)
        if u in seen: continue
        seen.add(u); total += w
        for nw, v in adj[u]:
            if v not in seen:
                heapq.heappush(heap, (nw, v))
    return total
```

Complexity

- Kruskal: O(E log E) for sorting; Union-Find operations are near-constant. Space O(V).
- Prim with a binary heap: O(E log V). Space O(V + E).

Kruskal suits sparse graphs and edge lists; Prim suits adjacency lists and dense graphs.

Common pitfalls

- Running either algorithm on a disconnected graph yields a spanning *forest*, not a tree.
- Forgetting to skip already-visited vertices when popping from Prim's heap.
- Confusing MST with shortest paths: an MST does not minimize distance between specific pairs.

Practice

1. Compute the MST weight of a small graph by hand using both algorithms and confirm they agree.
2. Solve a LeetCode-style "Min Cost to Connect All Points" problem.

UNION-FIND AND GRID PROBLEMS

Concept and Intuition

Union-Find (also called Disjoint Set Union, DSU) tracks a collection of non-overlapping groups. It supports two operations: `find(x)` returns the representative (root) of the group containing `x`, and `union(a, b)` merges the two groups. Each group is stored as a tree, and the root identifies the group. Two items are connected exactly when their roots match. It is the structure behind Kruskal's algorithm from the previous section.

Two optimizations keep the trees shallow:

- **Path compression:** during `find`, point every visited node directly at the root.
- **Union by rank:** attach the shorter tree beneath the taller one, so height grows only when two equal-rank trees merge.

Code (Python)

PYTHON

```python
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.count = n          # number of groups

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)
        if ra == rb:
            return False
        if self.rank[ra] < self.rank[rb]:
            ra, rb = rb, ra
        self.parent[rb] = ra
        if self.rank[ra] == self.rank[rb]:
            self.rank[ra] += 1
        self.count -= 1
        return True
```

Worked Example: Number of Islands

Given a grid of `'1'` (land) and `'0'` (water), count groups of land cells joined horizontally or vertically. Map cell `(r, c)` to index `r * cols + c`. Start with one group per land cell, then union each land cell with its land neighbor to the right and below. The final `count` is the island count.

For the grid `11 / 01`, there are three land cells, so `count = 3`. Union `(0,0)-(0,1)` gives 2; union `(0,1)-(1,1)` gives 1. One island.

```python
def num_islands(grid):
    R, C = len(grid), len(grid[0])
    dsu = DSU(R * C)
    water = 0
    for r in range(R):
        for c in range(C):
            if grid[r][c] == '0':
                water += 1
                continue
            if r + 1 < R and grid[r+1][c] == '1':
                dsu.union(r*C + c, (r+1)*C + c)
            if c + 1 < C and grid[r][c+1] == '1':
                dsu.union(r*C + c, r*C + c + 1)
    return dsu.count - water
```

A DFS or BFS flood fill solves the same problem in O(R·C) time as well.

Complexity

Operation	Time	Space
find / union (both optimizations)	nearly constant amortized (inverse Ackermann)	O(n)
Number of islands	about O(R·C)	O(R·C)

Common Pitfalls

- Comparing parents instead of roots; always call `find`.
- Forgetting to subtract water cells when every cell starts as its own group.
- Wrong index mapping (`r * cols + c`, not `r * rows + c`).

Practice Problems

1. Number of Provinces (connected groups from an adjacency matrix).
2. Redundant Connection (the edge whose `union` returns `False`).

Java/C++: the same arrays (`parent`, `rank`) and methods translate directly.

DYNAMIC PROGRAMMING: FOUNDATIONS AND 1D PROBLEMS

Concept and Intuition

Dynamic programming (DP) solves a problem by breaking it into overlapping subproblems and storing each answer so it is computed once. It applies when a problem has **optimal substructure** (the best answer is built from best answers to smaller pieces) and **overlapping subproblems** (the same pieces recur). Naive Fibonacci recursion is the classic case: `fib(n)` recomputes `fib(n-2)` many times.

Memoization vs Tabulation

Aspect	Memoization (top-down)	Tabulation (bottom-up)
Style	Recursion plus a cache	Loop filling a table
Computes	Only states actually reached	All states in order
Risk	Recursion depth limits	Must choose the correct fill order
Space tricks	Harder	Easy to keep only recent values

State-Design Framework

1. **State:** what does `dp[i]` mean? Write it as a sentence.
2. **Transition:** how does `dp[i]` depend on smaller states?
3. **Base cases:** the smallest states with known answers.
4. **Order:** fill so dependencies are ready.
5. **Answer:** which state holds the result?

Climbing Stairs

Count ways to reach step `n` taking 1 or 2 steps. State: `dp[i]` = ways to reach step `i`. Transition: `dp[i] = dp[i-1] + dp[i-2]`. Trace for n=4: dp = 1, 1, 2, 3, 5 (indices 0–4), so the answer is 5.

```python
def climb(n):
    a, b = 1, 1
    for _ in range(n - 1):
        a, b = b, a + b
    return b if n > 0 else 1
```

House Robber

Maximize the sum of non-adjacent values. State: `dp[i]` = best loot from the first `i` houses. Transition: `dp[i] = max(dp[i-1], dp[i-2] + nums[i-1])`.

```python
def rob(nums):
    prev, cur = 0, 0
    for x in nums:
        prev, cur = cur, max(cur, prev + x)
    return cur
```

For `[2, 7, 9, 3, 1]` the result is 12 (2 + 9 + 1).

Coin Change

Fewest coins to make `amount`. State: `dp[a]` = fewest coins for amount `a`. Transition: `dp[a] = min(dp[a-c] + 1)` over coins `c ≤ a`.

```python
def coin_change(coins, amount):
    INF = float('inf')
    dp = [0] + [INF] * amount
    for a in range(1, amount + 1):
        for c in coins:
            if c <= a:
                dp[a] = min(dp[a], dp[a - c] + 1)
    return -1 if dp[amount] == INF else dp[amount]
```

Complexity

- Climbing stairs and house robber: O(n) time, O(1) space with rolling variables.
- Coin change: O(amount × coins) time, O(amount) space.

Pitfalls

- Vague state definitions; write the meaning first.
- Missing base cases or unreachable-state handling (coin change).
- Greedy thinking: largest-coin-first can fail.

Practice

- Min Cost Climbing Stairs.
- House Robber II (circular houses).

Java/C++ equivalents use the same array loops; use a large sentinel instead of infinity.

DYNAMIC PROGRAMMING: 2D, LIS AND GRIDS

Moving from 1D to 2D State

When one index cannot describe a subproblem, add a second. The state becomes `dp[i][j]`, usually meaning "the best answer using the first `i` items of one input and the first `j` of another" (or a grid cell at row `i`, column `j`). Define the state in words first, then the base cases, then the transition.

Longest Common Subsequence (LCS)

Intuition: if the last characters match, they extend the LCS of the shorter prefixes; otherwise drop one character from either string and keep the better result.

- `dp[i][j] = dp[i-1][j-1] + 1` if `a[i-1] == b[j-1]`
- otherwise `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`

Filled table for `a = "ABC"`, `b = "AC"` (row 0 and column 0 are base cases of 0):

	""	A	C
""	0	0	0
A	0	1	1
B	0	1	1
C	0	1	2

The answer is `dp[3][2] = 2` ("AC").

```python
def lcs(a, b):
    dp = [[0]*(len(b)+1) for _ in range(len(a)+1)]
    for i in range(1, len(a)+1):
        for j in range(1, len(b)+1):
            if a[i-1] == b[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[-1][-1]
```

Edit Distance

`dp[i][j]` is the minimum operations to turn the first `i` characters of `a` into the first `j` of `b`. Base cases: `dp[i][0] = i`, `dp[0][j] = j`. If characters match, copy `dp[i-1][j-1]`; otherwise take `1 + min(` insert `dp[i][j-1]`, delete `dp[i-1][j]`, replace `dp[i-1][j-1])`.

0/1 Knapsack

`dp[i][w]` is the best value using the first `i` items with capacity `w`. Either skip item `i`, or take it if it fits: `max(dp[i-1][w], dp[i-1][w-wt] + val)`. Iterate `w` downward if compressing to one row.

Longest Increasing Subsequence (LIS)

Simple DP: `dp[i]` is the LIS ending at `i`, equal to `1 + max(dp[k])` for `k < i` with `nums[k] < nums[i]`. This is $O(n^2)$. Patience sorting with binary search reaches O(n log n).

Grid DP

For unique paths moving right or down, `dp[r][c] = dp[r-1][c] + dp[r][c-1]`, with the first row and column set to 1. Min-path-sum uses `grid[r][c] + min(top, left)`.

Complexity

Problem	Time	Space
LCS, edit distance	O(nm)	O(nm), or O(m) with rolling rows for LCS length
0/1 knapsack	O(nW)	O(nW), or O(W)
LIS	$O(n^2)$ or O(n log n)	O(n)
Grid DP	O(rc)	O(rc), or O(c)

Common Pitfalls

- Off-by-one errors between table indices and string indices.
- Forgetting base rows and columns.
- Iterating capacity upward in 1D knapsack, which reuses items.
- Confusing subsequence with substring.

Practice

1. Longest Common Subsequence; Edit Distance.
2. Partition Equal Subset Sum (a knapsack variant); Minimum Path Sum.

GREEDY ALGORITHMS AND DIVIDE AND CONQUER

Greedy Algorithms

Concept. A greedy algorithm builds a solution step by step, always taking the choice that looks best right now and never revisiting it. It works only when the problem has the *greedy-choice property* (a locally best move can be extended to a global optimum) and *optimal substructure*.

Intuition. Think of packing a day with as many meetings as possible: finishing early leaves the most room for the rest.

Activity Selection (Interval Scheduling)

Given intervals (start, end), choose the maximum number that do not overlap. Sort by end time, then take each interval whose start is at or after the last chosen end.

```python
def max_activities(intervals):
    intervals.sort(key=lambda x: x[1])
    count, last_end = 0, float('-inf')
    for s, e in intervals:
        if s >= last_end:
            count += 1
            last_end = e
    return count
```

Trace for (1,3), (2,5), (4,7), (6,8): take (1,3); skip (2,5) since 2 < 3; take (4,7); skip (6,8) since 6 < 7. Result: 2.

Complexity: O(n log n) time for sorting, O(1) extra space.

Huffman Coding

Huffman coding assigns shorter bit codes to frequent symbols. Put all symbols in a min-heap by frequency; repeatedly remove the two smallest nodes, merge them into a parent whose frequency is their sum, and push it back. The final tree gives codes: left edge 0, right edge 1.

```python
import heapq
def huffman(freq):  # freq: {char: count}
    heap = [[w, [c, ""]] for c, w in freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo, hi = heapq.heappop(heap), heapq.heappop(heap)
        for p in lo[1:]: p[1] = "0" + p[1]
        for p in hi[1:]: p[1] = "1" + p[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return dict(heap[0][1:])
```

Complexity: O(n log n) for n distinct symbols; O(n) space.

When Greedy Fails

Coin change with coins {1, 3, 4} and target 6: greedy takes 4, then 1, 1 (three coins). The optimum is 3 + 3 (two coins). Taking the largest coin first is not safe here, so use dynamic programming (Section 32).

Pitfalls: sorting by the wrong key (start time or duration instead of end time); assuming greedy works without a justification.

Divide and Conquer

Divide the problem into smaller independent subproblems, solve each recursively, then combine. Merge sort and binary search are examples. Running time is often described by a recurrence such as $T(n) = 2T(n/2) + O(n)$, which gives $O(n \log n)$.

Practice: (1) Non-overlapping Intervals. (2) Construct a Huffman code for frequencies a:5, b:9, c:12 and verify that the two smallest merge first.

CHEAT SHEET AND INTERVIEW-PREP ROADMAP

How to Use This Page

This final section condenses the notes into three tools: a complexity cheat sheet, a pattern-recognition table, and a 12-week roadmap. Complexities are the standard textbook values; verify them against the earlier sections when in doubt.

Complexity Cheat Sheet

Structure	Access	Search	Insert	Delete
Array	O(1)	O(n)	O(n)	O(n)
Linked list	O(n)	O(n)	O(1) at known node	O(1) at known node
Hash map (average)	n/a	O(1)	O(1)	O(1)
Balanced BST	n/a	O(log n)	O(log n)	O(log n)
Binary heap	O(1) top	O(n)	O(log n)	O(log n) top

Sort	Average	Worst	Stable
Bubble / Insertion	$O(n^2)$	$O(n^2)$	Yes
Selection	$O(n^2)$	$O(n^2)$	No
Merge	O(n log n)	O(n log n)	Yes
Quick	O(n log n)	$O(n^2)$	No
Heap	O(n log n)	O(n log n)	No
Counting	O(n + k)	O(n + k)	Yes

Pattern-Recognition Table

Clue in the problem	Technique
Sorted input, find a target	Binary search
Contiguous subarray or substring	Sliding window, prefix sum, Kadane
Pair or complement lookup	Hash map, two pointers
Next greater or smaller element	Monotonic stack
Top-K or streaming minimum	Heap
All combinations or arrangements	Backtracking
Shortest path, unweighted	BFS
Weighted shortest path	Dijkstra (non-negative weights)
Dependencies or ordering	Topological sort
Overlapping subproblems	Dynamic programming
Dynamic connectivity	Union-Find

12-Week Roadmap (a suggested plan)

Weeks	Focus	Practice problems
1	Big-O, recursion	Factorial, Fibonacci
2	Arrays, strings	Two Sum, Maximum Subarray, Longest Substring Without Repeating Characters
3	Hashing, linked lists	Valid Anagram, Reverse Linked List, Linked List Cycle
4	Stacks, queues, searching	Valid Parentheses, Binary Search
5	Sorting	Merge Sort implementation, Sort an Array
6	Backtracking	Subsets, Permutations, N-Queens
7	Trees, BSTs	Level Order Traversal, Validate BST, LCA
8	Heaps, tries, bits	Top K Frequent Elements, Single Number
9	Graphs	Number of Islands, Course Schedule
10	Shortest paths, MST	Network Delay Time
11	Dynamic programming	Climbing Stairs, Coin Change, LCS
12	Mock interviews, review	Mixed timed sets

Interview Tips

- Restate the problem and confirm constraints.
- Start with a brute-force solution, then optimize.
- State time and space complexity aloud.
- Test with small and edge cases before declaring done.