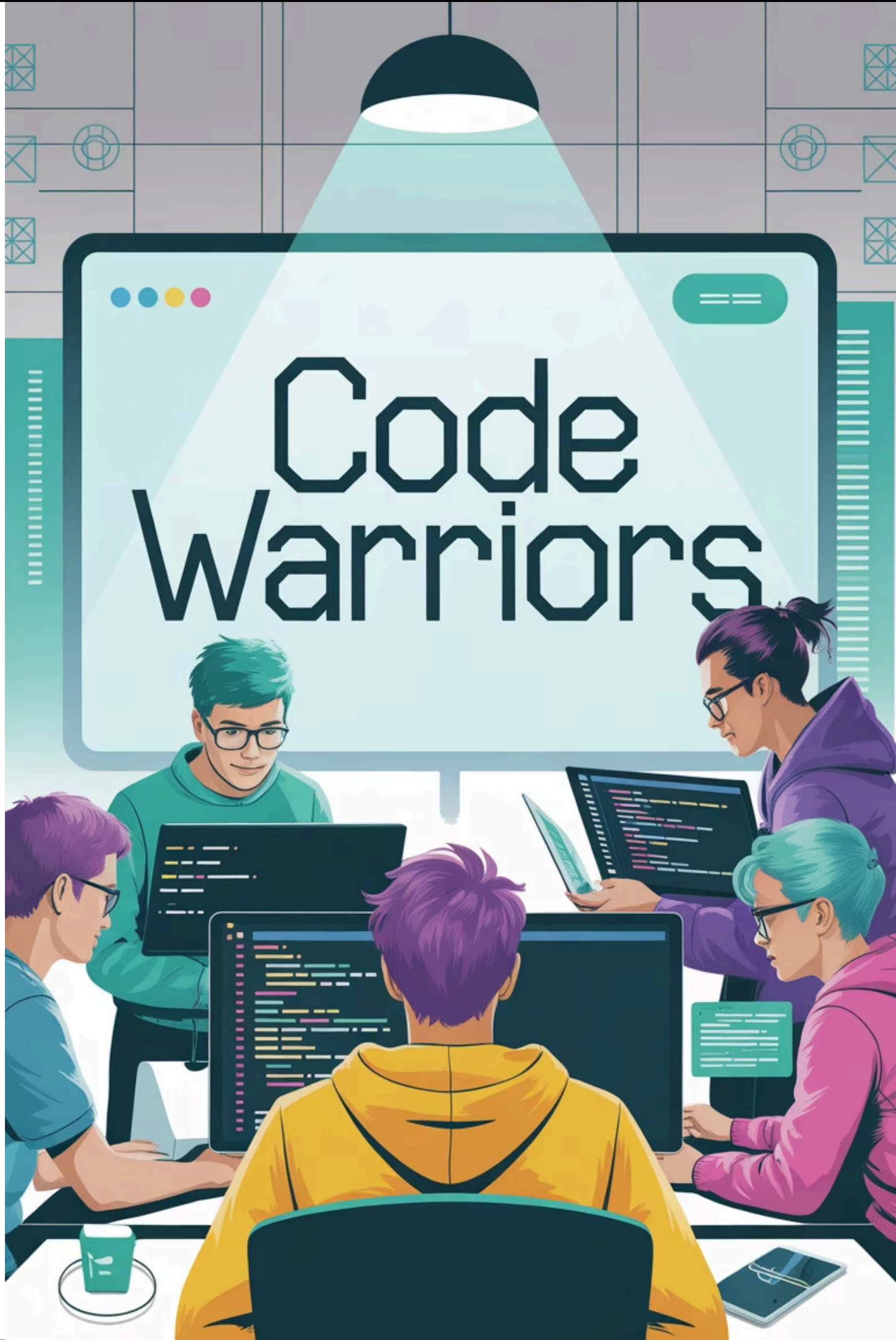
# DSA for Placements (AI-assisted)

Real systems, real patterns — with local LLM help

**Karthikeyan NG**
Madurai 2025



# Today's Plan (60m)

01

## Why DSA matters in real systems

Quick overview of how data structures appear in production systems you'll work with (5 minutes)

02

## Rules & scoreboard

Setting up our competitive coding environment and success metrics (5 minutes)

03

## Programming Challenges

Choose 2 from: LRU Cache, Bracket Validator, Two Sum, Min Stack, Grid BFS - hands-on coding with AI assistance (35 minutes)

04

## Live demo with Continue + Ollama

See local LLM integration in action for real-time coding assistance (10 minutes)

05

## Wrap-up session

Complexity analysis notes and repository artifacts for future reference (5 minutes)

## Why DSA matters (fast mapping)

Understanding data structures isn't just about passing interviews—these patterns power the systems you'll build in your career. Let's connect the dots between classroom theory and production reality.

### Caches → LRU/LFU

Feed scrolling, API rate limiting, and memory management all rely on efficient cache eviction policies you'll implement.

### Editors/IDEs → Stacks/Tries

Every undo/redo operation uses stacks, while autocompletion features leverage trie data structures for fast prefix matching.

### Maps/Chat → Graphs/Queues

Navigation apps use graph algorithms for routing, while chat applications implement queues for message fan-out to multiple users.

### Search/Retrieval → Hashing/Sets

Database joins and deduplication operations rely heavily on hash tables and set operations for optimal performance.

### Campus Apps → BFS/DP

Finding shortest paths between buildings uses BFS, while course scheduling optimization problems require dynamic programming solutions.

## Rules & Scoreboard

Let's establish our coding competition framework to maximize learning and simulate real interview conditions. This structured approach will help you develop both technical skills and time management.

### Tests First Approach

Always start by asking the LLM to draft comprehensive edge-case tests. This test-driven development approach mirrors industry best practices and helps catch bugs early.

### Timeboxing Strategy

Strict 15-20 minute limit per challenge. This mimics interview pressure and teaches you to prioritize the most important aspects of your solution first.

### Pair Programming

Navigator crafts AI prompts and guides strategy, while driver writes code. Switch roles between challenges to practice both skill sets.

## Scoring Criteria

- **Tests pass**: All edge cases handled correctly
- **Time to green**: How quickly you achieve working solution
- **Clean code**: Readable, well-structured implementation
- **Extensions**: Optional micro-benchmarking and memory analysis

# Local AI Setup — Ollama + Continue

Setting up your local LLM environment gives you AI assistance without sending your code to external servers. This setup respects privacy while providing powerful coding support during practice and interviews.

## Installation Steps

Download and install Ollama from [ollama.com](https://ollama.com) for your operating system (macOS, Linux, or Windows).

Pull your preferred models based on your laptop's capabilities:

```
ollama pull phi3:mini # very light, fast
ollama pull llama3.2 # balanced
# optional if your laptop is strong:
ollama pull mistral
ollama pull codellama
ollama run deepseek-r1
```

## Verification

Start the Ollama server and verify everything works:

```
ollama list
curl -s http://127.0.0.1:11434/api/tags | head -c 200
```

You should see your installed models listed and receive a JSON response from the API endpoint.

# Configure VS Code Continue (Provider: Ollama)

The Continue extension transforms VS Code into an AI-powered coding environment. It provides context-aware suggestions and can help debug code, write tests, and explain complex algorithms.

1

### Install Extension

Search for "Continue" in the VS Code Extensions marketplace and install it. The extension will add AI chat and inline suggestions to your editor.

2

### Configure Provider

Set Provider to Ollama and select your preferred model. Models are switchable during class based on performance needs.

3

### Test Setup

Open the Continue chat panel and verify it can communicate with your local Ollama instance.

**Example Configuration**

```
{
  "models": {
    "default": {
      "provider": "ollama",
      "model": "llama3.2:3b-instruct"
    }
  }
}
```

**Pro tip**: Use phi3:mini if responses are too slow on your hardware.

## Model Presets (speed vs quality)

Choosing the right model is crucial for maintaining productivity during coding sessions. Different models offer various tradeoffs between response speed, quality, and resource usage.

1

### phi3:mini

**Fastest option** - Extremely lightweight and responsive. Perfect for generating tests, small code snippets, and quick explanations. Ideal for older laptops or when speed is priority.

2

### llama3.2:3b-instruct

**Balanced choice** - Better reasoning capabilities while remaining relatively fast. Good for complex problem-solving and detailed code analysis. Recommended for most users.

3

### mistral:7b-instruct

**High quality** - Superior reasoning and code generation, but requires stronger hardware (16GB+ RAM). Use when you need the best possible assistance quality.

**Dynamic switching tip**: You can change models mid-session in Continue based on your current needs—use lighter models for quick tasks and heavier models for complex problem solving.

## Troubleshooting (quick)

Common setup issues and their solutions to keep your AI-assisted coding session running smoothly.

- **Connection refused error**
  Ensure `ollama serve` is running in the background. Check that the Ollama desktop app is started or manually run the server command.

- **Model not found**
  Run `ollama pull <model-name>` to download the missing model. Verify with `ollama list` to see all available models.

- **Slow responses**
  Switch to a smaller model like phi3:mini or llama3.2:3b-instruct. Reduce prompt length, or close other resource-intensive applications. Ensure you have adequate RAM/GPU for the model chosen.

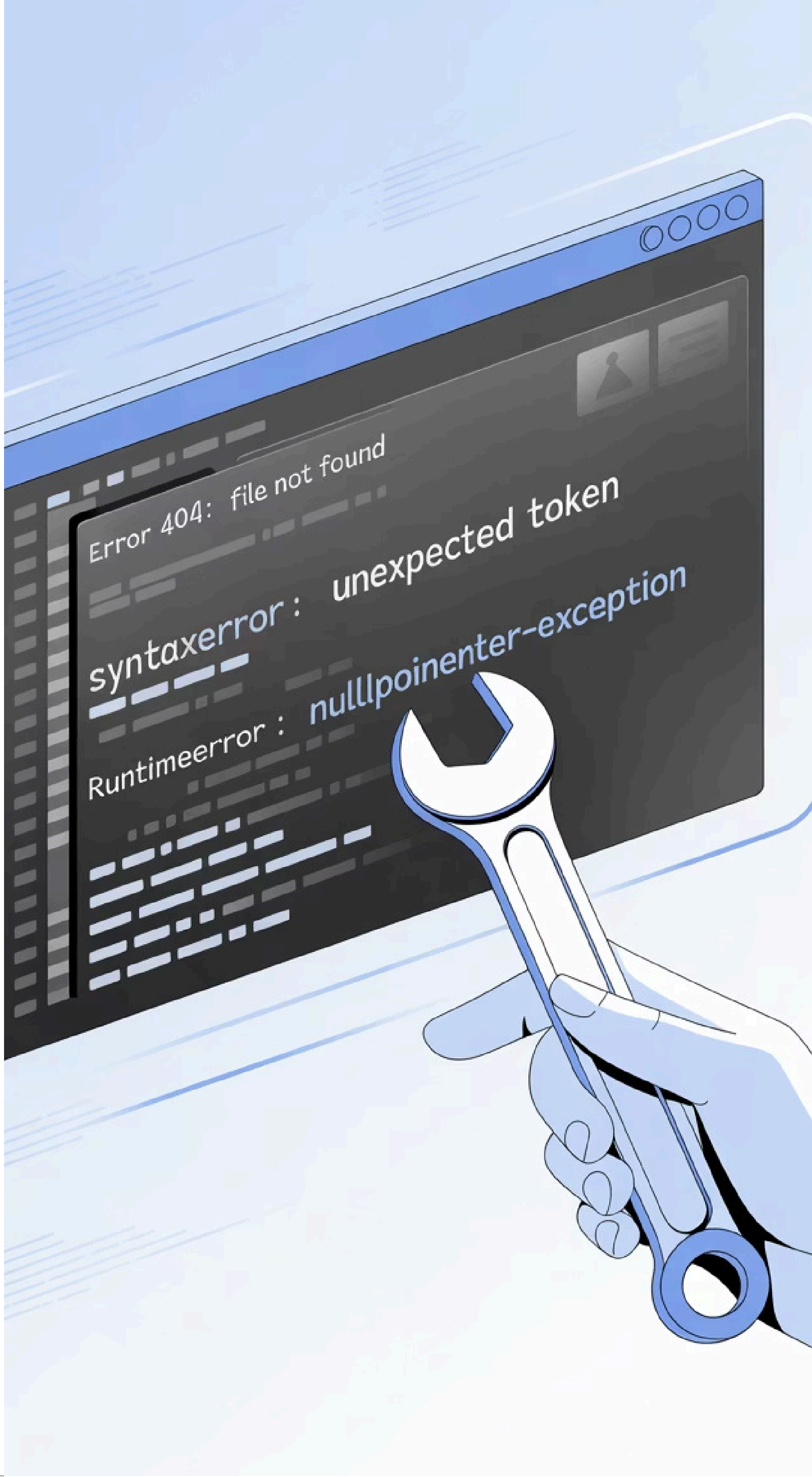
- **Out of memory errors**
  Use a smaller model or restart Ollama to clear memory. Monitor system resources (especially RAM/VRAM) during model loading. Consider upgrading your hardware if consistently encountering this issue.

- **Continue Extension Unresponsive**
  If the Continue extension freezes or doesn't respond, try reloading the VS Code window (Command Palette: `Developer: Reload Window`). If the issue persists, restart VS Code or reinstall the extension. Check the VS Code Output panel for any error messages related to Continue.

- **VS Code Integration Problems**
  If Continue isn't appearing in the sidebar or its commands aren't working, ensure the extension is enabled. Check for potential conflicts with other VS Code extensions. Verify your user or workspace `settings.json` for any Continue-specific configurations that might be incorrect.

- **High CPU/GPU Usage**
  If Ollama or Continue is consuming excessive CPU/GPU resources, consider switching to a smaller model. Close other demanding applications. For Ollama, monitor usage with `ollama ps` and check Ollama documentation for any resource limiting configurations.

- **Incorrect Model Behavior / Hallucinations**
  If the model is providing irrelevant or incorrect responses, try a different prompt engineering approach. Switch to a higher quality model (e.g., mistral:7b-instruct) for more complex tasks. Ensure the model has been fully pulled and isn't corrupted.


Error 404: file not found
syntaxerror: unexpected token
Runtimeerror: nullpointenter-exception

# Run Tests Quickly

Efficient test execution is crucial for rapid feedback during coding challenges. Here are the fastest ways to run your test suite and verify your solutions.

## VS Code Task Runner

The quickest method using the integrated terminal:

**Terminal → Run Task → "Session 2: Run Challenges (unittest)"**

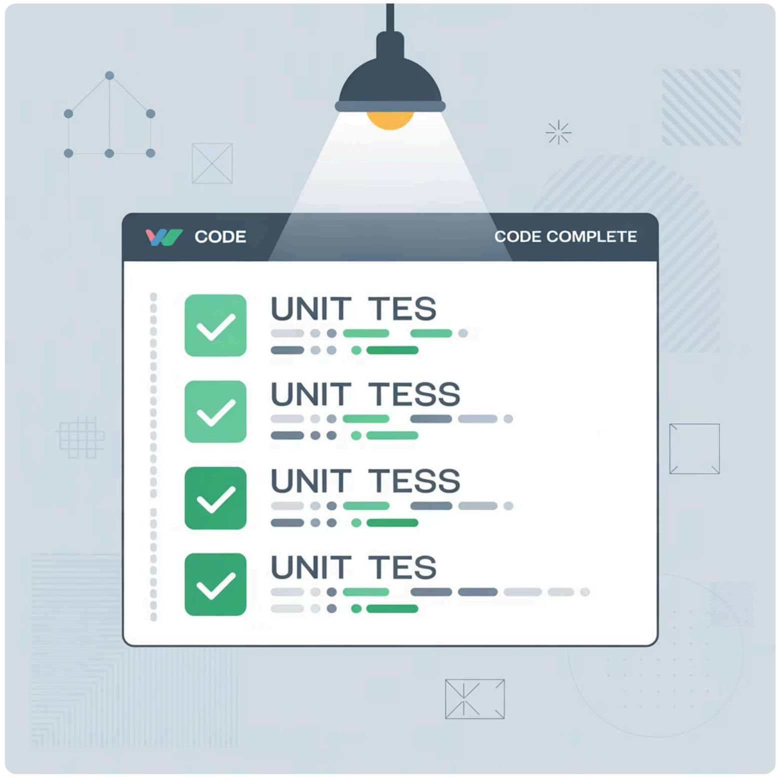
This pre-configured task runs all challenge tests with proper discovery and verbose output.

## Command Line Interface

For direct terminal access:

```
python -m unittest discover -s sessions/session02_dsa/content/challenges -p 'test_*.py' -v
```

This command discovers and runs all test files matching the pattern, providing detailed output for debugging failed assertions.



Both methods provide the same comprehensive test results with color-coded pass/fail indicators and detailed error messages for quick debugging.

## Let's Code!

Now that you're fully equipped with your local AI coding environment, it's time to put theory into practice. With AI assistance at your fingertips, you're ready to tackle any challenge.

### Ollama Ready

Local LLMs are installed and prepared to run powerful models directly on your machine.

### Continue Integrated

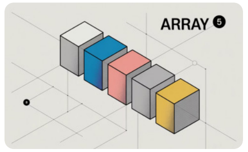
Your VS Code is enhanced with context-aware AI suggestions and debugging capabilities.

### Tests Enabled

Efficient test execution methods are set up for rapid feedback and solution validation.

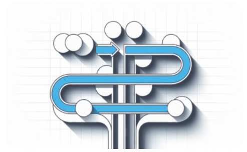
# Big-O Snack (Quick Intuition)

Master the essential complexity classes that appear in 90% of interview questions. Understanding when each complexity occurs helps you choose the right approach and communicate effectively with interviewers.



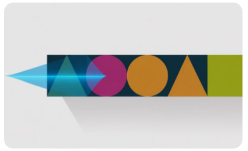
### O(1) Constant Time

Direct access operations like array indexing, hash table lookups, or stack push/pop.



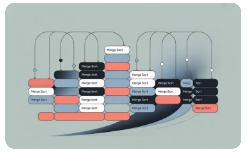
### O(log n) Logarithmic Time

Algorithms that repeatedly halve the problem size, such as binary search or balanced tree operations.



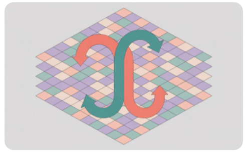
### O(n) Linear Time

Operations that require visiting each element once, like linear search or BFS/DFS traversals.



### O(n log n) Log-linear Time

Efficient sorting algorithms (e.g., merge sort, quicksort) and divide-and-conquer approaches.



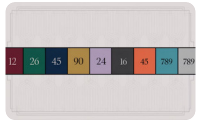
### O($n^2$) Quadratic Time

Algorithms involving nested loops over the same data set, like bubble sort or comparing all pairs.

**Quick recognition rules**: Nested loops with the same range often suggest O($n^2$). Divide-and-conquer typically yields O(n log n). Remember that space-time tradeoffs can let you use extra memory to achieve O(1) per operation, like in LRU cache implementations.

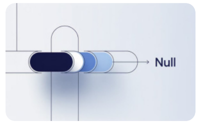
# Data Structures at a Glance

Your toolkit for solving algorithmic problems efficiently. Each data structure excels in specific scenarios—knowing when to use which one separates good developers from great ones.



### List/Array

**O(1) indexing, O(n) mid insert/delete**
Perfect for buffers, random access patterns, and when you need predictable memory layout.

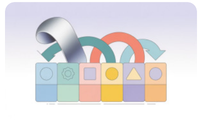


### Linked List

**O(1) end operations with pointers**
Ideal for queues, logs, and scenarios with frequent insertions/deletions at known positions.

### Stack/Queue/Deque

**O(1) push/pop operations**
Essential for undo/redo systems, BFS frontiers, and managing LIFO/FIFO workflows.

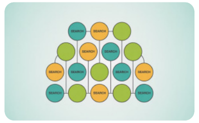


### Hash Map/Set

**O(1) average lookup/insert**
Go-to for caches, deduplication, joins, and any frequent lookup scenarios.

### Heap/Priority Queue

**O(log n) push/pop of extremes**
Perfect for scheduling systems, finding top-k elements, and maintaining priority-ordered data.



### Tree/BST (balanced)

**O(log n) search/insert**
Excellent for ordered queries, range operations, and maintaining sorted data with dynamic updates.

# Patterns at a Glance

Recognizing these algorithmic patterns quickly is your secret weapon in coding interviews. Most problems are variations of these fundamental approaches—master the pattern, solve the problem.

### Two Pointers

**O(n)** - Finding pairs, removing duplicates in sorted arrays, palindrome checking

```
# Finding a pair with sum target
l, r = 0, len(arr) - 1
while l < r:
    if arr[l] + arr[r] == target: return True
    if arr[l] + arr[r] < target: l += 1
    else: r -= 1
```

### Sliding Window

**O(n)** - Longest/shortest substring problems, fixed-size window maximums

```
# Max sum subarray of size k
curr_sum = sum(arr[:k])
max_sum = curr_sum
for i in range(k, len(arr)):
    curr_sum += arr[i] - arr[i-k]
    max_sum = max(max_sum, curr_sum)
```

### Hashing

**O(n)** - Frequency maps, seen sets, anagram detection, complement finding

```
# Counting character frequencies
counts = {}
for char in s:
    counts[char] = counts.get(char, 0) + 1
# Check frequency: print(counts['a'])
```

### Binary Search

**O(log n)** - Sorted data searches, answer space optimization problems

```
# Search in sorted array
low, high = 0, len(arr) - 1
while low <= high:
    mid = (low + high) // 2
    if arr[mid] == target: return mid
    elif arr[mid] < target: low = mid + 1
    else: high = mid - 1
```

### BFS/DFS

**O(V+E)** - Graph traversal, shortest unweighted paths, connected components

```
# DFS (recursive)
def dfs(node, visited):
    if node in visited: return
    visited.add(node)
    for neighbor in node.neighbors: dfs(neighbor, visited)
# BFS (queue based)
from collections import deque
q = deque([start_node]); visited = {start_node}
while q:
    node = q.popleft()
    for neighbor in node.neighbors:
        if neighbor not in visited:
            visited.add(neighbor); q.append(neighbor)
```

### Dynamic Programming

**O(states)** - Overlapping subproblems, optimization with memoization or tabulation

```
# Fibonacci sequence (tabulation)
dp = [0] * (n + 1)
dp[0], dp[1] = 0, 1
for i in range(2, n + 1): dp[i] = dp[i-1] + dp[i-2]
# result is dp[n]
```

### Greedy

**Varies** - Local optimal choices leading to global optimum (prove this works!)

```
# Coin Change (greedy, for specific coin sets)
coins = sorted(coins, reverse=True)
count = 0
for coin in coins:
    count += amount // coin
    amount %= coin
# If amount > 0, cannot make exact change
```

# DP Primer: Memoization vs Tabulation

Dynamic Programming transforms exponential problems into polynomial ones by avoiding redundant calculations. Understanding both approaches gives you flexibility in implementation and optimization.

## Core Principles

Dynamic Programming is applicable when problems exhibit two key characteristics:

- **Overlapping Subproblems**: The same smaller problems are solved multiple times.
- **Optimal Substructure**: The optimal solution to the problem contains optimal solutions to its subproblems.

## Two Approaches to DP

### Memoization (Top-Down)

A recursive approach enhanced with caching. It's often easier to write and understand as it follows the natural decomposition of the problem. Ideal for prototyping and when not all subproblems need to be computed.

### Tabulation (Bottom-Up)

An iterative approach that builds solutions from base cases upwards. It provides better control over computation order and space usage, making it preferred in production code for predictable performance.

## Fibonacci Sequence: A Concrete Example

### Memoization (Top-Down)

We start from the desired `n` and recursively break it down, storing results in a cache (e.g., a dictionary or array) to avoid recomputing.

```
memo = {}
def fib_memo(n):
    if n <= 1: return n
    if n in memo: return memo[n]
    memo[n] = fib_memo(n-1) + fib_memo(n-2)
    return memo[n]
# Warning: Without memoization, recursive calls become exponential!
```

### Tabulation (Bottom-Up)

We build an array (DP table) starting from base cases (fib(0), fib(1)) and iteratively calculate up to `n`, filling the table as we go.

```
dp = [0] * (n + 1)
dp[0], dp[1] = 0, 1 # Base cases
for i in range(2, n + 1):
    dp[i] = dp[i-1] + dp[i-2]
# result is dp[n]
```

# Binary Search on Answer Space

This powerful technique extends binary search beyond sorted arrays to optimization problems. When you can verify if a solution is "good enough," you can binary search for the optimal answer.

01

## Identify Monotonicity

Verify that feasibility is monotonic—if answer X works, then all answers ≥ X also work (or vice versa).

02

## Define Bounds

Set realistic low and high bounds for your answer space. These bounds should guarantee that the optimal answer lies within this range.

03

## Create Predicate Function

Implement a function that checks if a given answer is feasible. This becomes your "comparison" function for binary search.

04

## Binary Search Logic

Use standard binary search: `mid = (low + high) // 2`, test `predicate(mid)`, and adjust bounds accordingly.

### Classic Examples

- **Minimum capacity to ship packages in D days**: Can we ship with capacity X?
- **Minimum speed to eat all bananas in H hours**: Can we finish at speed X?
- **Kth smallest element in matrix**: How many elements ≤ X?

**Complexity**: O(log(range) × verification_cost)

# Edge-Case Bank (use for any challenge)

Comprehensive edge case testing separates successful candidates from those who miss corner cases. Use this checklist for every coding problem to ensure robust solutions.

### Size Boundaries

- Empty inputs (e.g., `len(arr) == 0`, `""`)
- Single element collections (e.g., `len(arr) == 1`, `[5]`)
- Minimum capacity constraints (capacity = 1)
- Maximum size inputs

### Data Patterns

- All identical elements (e.g., `[1,1,1,1]`, `"aaaa"`)
- Sorted and reverse-sorted inputs (e.g., `[1,2,3,4]`, `[4,3,2,1]`)
- Alternating patterns (e.g., `[1,0,1,0]`)
- Long runs of same values

### Boundary Conditions

- First/last index access
- Start/end positions blocked
- Off-by-one errors in loops
- Wraparound scenarios

### Special Values

- Negative numbers and zeros (e.g., `0`, `-1`, `-100`)
- Integer limits (e.g., `sys.maxsize`, `-sys.maxsize` for fixed-size int contexts)
- Empty strings, lists, dicts
- Invalid operation attempts (e.g., division by zero)

### Python-Specific Edge Cases

- `None` values (e.g., `[1, None, 3]`, dictionary keys/values being `None`)
- String encoding issues (e.g., Unicode characters, different encodings)
- Floating-point precision (e.g., `0.1 + 0.2 != 0.3`, `float('inf')`, `float('-inf')`, `float('nan')`)
- Mutable default arguments in function definitions
- Generator exhaustion (iterating over a generator multiple times)

**Pro tip**: Always test exception paths like popping from empty stack or accessing missing keys. These edge cases often reveal logical flaws in your implementation.

# Python Pitfalls (quick)

Avoid these common Python traps that can derail your coding interview. Understanding these subtleties demonstrates Python fluency and prevents debugging headaches.

→ **Mutable Default Arguments**

Never use `def func(arr=[]):` - the same list is reused across calls! Use `arr=None` and create new list inside function.

→ **Shallow vs Deep Copy**

`list(x)` creates shallow copy - nested objects are still shared. Use `copy.deepcopy(x)` for complete independence of nested structures.

→ **Integer Division Gotchas**

`//` truncates toward negative infinity, not zero. `-3 // 2 = -2`, not `-1`. Use `int(-3 / 2)` for truncation toward zero.

→ **Dictionary/Set Key Requirements**

Keys must be immutable (hashable). Lists and dictionaries can't be keys, but tuples can. This affects how you structure data in hash-based solutions.

## Complexity Cheatsheet (common ops)

Quick reference for the most frequently used operations in coding interviews. Memorize these complexities to make optimal algorithmic choices under pressure.

**O(1)**

### Hash Map Operations

Average case for insert, lookup, delete. Worst case can degrade to O(n) with poor hash function or many collisions.

**O(1)**

### Stack/Queue Ops

Push, pop, peek operations on properly implemented stacks and queues using arrays or linked lists.

**O(mn)**

### Grid BFS

Time complexity for BFS on m×n grid. Space complexity also O(mn) for visited set and queue storage.

**O(1)**

### LRU Cache

Both get and put operations using HashMap + doubly-linked list combination. Space complexity O(capacity).

Remember that average case and worst case can differ significantly. During interviews, always clarify which case you're analyzing and discuss potential optimizations.

# Prompt patterns for DSA (local LLM)

Craft effective prompts to get maximum value from your local AI assistant. These proven patterns help you generate better tests, debug faster, and understand complex algorithms efficiently.

### Tests First Pattern

*"Generate 6 edge-case tests for [CHALLENGE]. Include capacity=1, updates, repeats, and boundary conditions."*

Always start with comprehensive test generation to ensure your solution handles all scenarios.

### Constraint Pattern

*"Solve in O(n) time, O(1) extra space if possible. Avoid sorting for the hash table approach."*

Specify complexity requirements upfront to get optimal algorithmic approaches.

### Debug Pattern

*"Given this failing test case, propose a minimal diff to fix the bug without changing the overall approach."*

Focus on targeted fixes rather than complete rewrites when debugging.

### Explanation Pattern

*"Summarize the time/space complexity and describe one real-world usage example in 5 lines or less."*

Get concise explanations perfect for interview discussions.

# Quick Win Challenge — Reverse String

This challenge is designed to be a quick, confidence-building exercise, taking only about 5 minutes. It helps you get comfortable with the AI-driven workflow before tackling more complex problems.

### Problem Statement

Reverse a given string. You can either reverse it in-place (if mutable) or return a reversed copy.

### Estimated Time

5 minutes

### Purpose

Build confidence and familiarize yourself with the AI interaction patterns for problem-solving.

### Simple Test Cases

- `""` -> `""`
- `"a"` -> `"a"`
- `"hello"` -> `"olleh"`
- `"racecar"` -> `"racecar"`

### Hint

Consider Python's string slicing `[::-1]` for a quick solution, or the two-pointer approach for an in-place reversal.

This quick exercise will warm up your interaction pattern with the AI, ensuring you're ready for more substantial challenges.

# Programming Challenge — LRU Cache

Implement a Least Recently Used cache with O(1) operations. This fundamental data structure appears in many systems and tests your ability to combine hash tables with doubly-linked lists effectively.

## API Requirements

- `get(key) -> int`: Return value or -1 if missing
- `put(key, value)`: Insert/update key-value pair
- Both operations must be O(1) average time

## Implementation Strategy

Use HashMap for O(1) key lookup combined with doubly-linked list for O(1) removal/insertion. Alternative: Python's OrderedDict provides built-in LRU functionality.

## Edge Cases to Test

- Capacity = 1 scenarios
- Overwriting existing keys
- Repeated gets (updates recency)
- Correct eviction order

## Real-World Usage

Web caches, API rate limiting, database buffer pools, and mobile app memory management all use LRU eviction policies.

```
class LRUCache:
    def __init__(self, capacity: int):
        ...
    def get(self, key: int) -> int:
        ...
    def put(self, key: int, value: int) -> None:
        ...
```

### Live Prompt — LRU Cache

You are a Python unit test generator. Write 6 edge-case tests for LRUCache with API get(key)->int (-1 if missing) and put(key, value). Cover: capacity=1, overwrite existing key, repeated gets (updates recency), eviction order. Return unittest code with a TestCase named TestLRUCache only.

# Programming Challenge — Bracket Validator

Validate balanced brackets using stack operations. This classic problem tests your understanding of LIFO data structures and proper nesting validation—a skill used in parsers and compilers.

**1 Algorithm Flow**

Push opening brackets onto stack, pop when closing bracket matches top of stack

**2 Validation Rules**

Each closing bracket must match the most recent unmatched opening bracket

**3 Final Check**

Stack must be empty at end—no unmatched opening brackets remain

## Input/Output

**Input**: String containing only `(){}[]` characters
**Output**: Boolean indicating if brackets are properly balanced and nested

## Critical Edge Cases

- Empty string (valid)
- Single character (invalid)
- Wrong order: `")("`
- Cross-nesting: `"([)]"`
- Mixed valid nesting: `"{[()]}"`

```
def is_valid_brackets(s: str) -> bool:
    ...
```

### Live Prompt — Bracket Validator

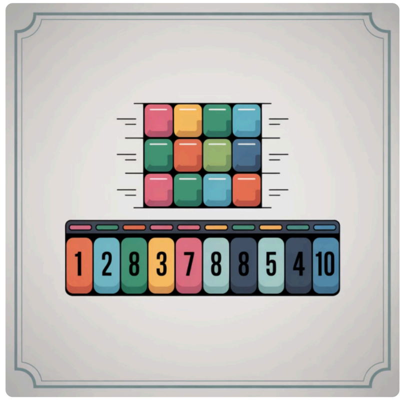
Write 6 unit tests for is_valid_brackets(s: str) -> bool. Include: empty string, single char, ")(", mixed nested valid, cross-nesting invalid, long balanced. Output Python unittest code defining TestBrackets only.

# Programming Challenge — Two Sum (and variants)

Find two numbers in an array that sum to a target value. This fundamental problem demonstrates hash table usage and introduces important concepts like index handling and multiple solution approaches.

## Problem Statement

Given an array of integers and a target sum, return the indices i, j where `nums[i] + nums[j] = target` and `i < j`. Return None if no valid pair exists.



## Solution Approaches

- **Hash Map O(n)**: Store complements in hash table for O(1) lookup
- **Two Pointers O(n log n)**: Sort first, then use two pointers technique
- **Brute Force O(n²)**: Check all pairs (not recommended but good to mention)

## Advanced Variant

**Stream mode**: Process numbers one at a time and yield index pairs as soon as valid pairs are found (useful for large datasets).

## Implementation Considerations

- Handle duplicate values correctly
- Ensure `i < j` ordering constraint
- Consider negative numbers and zero
- What if multiple valid pairs exist? (usually return first found)

```
def two_sum_indices(nums: list[int], target: int) -> tuple[int, int] | None:
    ...
```

### Live Prompt — Two Sum

Write unit tests for two_sum_indices(nums: list[int], target: int) -> tuple[int,int] | None. Include: multiple pairs (pick first valid indices order i None. Output Python unittest code defining TestTwoSum only.

# Programming Challenge — Min Stack (O(1) min)

Design a stack that supports push, pop, top, and retrieving minimum element in constant time. This challenge tests your ability to maintain auxiliary data structures for optimization.

1

### Core Stack Operations

`push(x)`, `pop()`, `top()` work exactly like normal stack with O(1) time complexity.

2

### Minimum Tracking

`get_min()` returns current minimum in O(1) time, even after elements are popped.

3

### Space-Time Tradeoff

Use extra space to maintain minimum information alongside regular stack data.

## Implementation Strategies

### Two Stack Approach

Maintain separate stack for minimums. Push to min_stack only when new minimum found, ensuring min_stack.top() is always current minimum.

### Pairs Approach

Store (value, current_min) pairs in single stack. Each element knows the minimum at the time it was pushed.

## Critical Edge Cases

- Duplicate minimum values
- Negative numbers as minimums
- Long sequences with changing minimums
- Operations on empty stack (should raise IndexError)

```
class MinStack:
    def push(self, x: int) -> None: ...
    def pop(self) -> None: ...
    def top(self) -> int: ...
    def get_min(self) -> int: ...
```

### Live Prompt — Min Stack

Write unit tests for MinStack with push, pop, top, get_min (all O(1)). Include: duplicates, negatives, min updates after pops, and IndexError on pop/top/get_min when empty. Output Python unittest code defining TestMinStack only.

## Programming Challenge — Grid Shortest Path (BFS)

Find the shortest path in a grid from top-left to bottom-right corner using breadth-first search. This problem combines graph traversal with 2D coordinate handling—essential skills for many technical interviews.

### Grid Setup

**0 = walkable cell**
**1 = wall/obstacle**
Move in 4 directions: up, down, left, right

### BFS Implementation

Use queue for frontier, visited set to avoid cycles. Track distance/steps as you explore outward from start.

### Return Value

Return shortest path length as integer, or -1 if no path exists between start and destination.

### Algorithm Details

BFS guarantees shortest path in unweighted grids. Start from (0,0), explore all reachable cells level by level until reaching (m-1, n-1). Track visited cells to avoid infinite loops.

### Edge Cases to Consider

- Start position blocked (immediate return -1)
- End position blocked (return -1)
- Single cell grid (return 0 if walkable)
- Narrow corridors and multiple valid paths
- Rectangular vs square grids

**Complexity**: O(m×n) time for exploring all cells, O(m×n) space for queue and visited set.

```
def shortest_path_len(grid: list[list[int]]) -> int:
    ...
```

### Live Prompt — Grid BFS

Write unit tests for shortest_path_len(grid: list[list[int]]) -> int. Include: simple 3x3 path length, blocked start, blocked end, no path, rectangular grid, tie paths choose shortest. Output Python unittest code defining TestGridBFS only.

# Demo: Continue + Ollama

Watch the complete AI-assisted coding workflow in action. This live demonstration shows how to leverage local LLM capabilities to accelerate development while maintaining code quality and understanding.

## 1. Setup Verification

**Action:** Confirm your Ollama provider connection and model selection (e.g., `llama3.2:3b-instruct` or `phi3:mini` based on system performance).

**Prompt (Internal thought):** "Is Ollama running? Is the chosen model loaded?"

**Expected Output:** Ollama server status shows "Running", model loaded successfully in Continue.

**Common Issue:** "Model not found" or "Connection refused" – troubleshoot Ollama server or model download.

## 2. Test-Driven Development

**Action:** Generate comprehensive unit tests first.

**Prompt:** "Generate comprehensive Python unit tests for the `shortest_path_len(grid: list[list[int]]) -> int` function for a 2D grid BFS problem. Include cases for a simple 3x3 path, blocked start, blocked end, no path, rectangular grid, and scenarios with multiple valid paths where the shortest must be chosen. Output only `unittest` code defining `TestGridBFS`."

**Expected Output:** A Python `unittest` class with multiple `test_...` methods. Run tests to see red status.

**Common Issue:** Tests don't cover all edge cases; AI might generate slightly incorrect assertions.

## 3. Implementation Phase

**Action:** Use AI assistance to implement the algorithm.

**Prompt:** "Implement the `shortest_path_len` function using Breadth-First Search (BFS) for the given grid problem. The grid contains 0s for walkable cells and 1s for walls/obstacles. Allow movement in 4 directions: up, down, left, right. Return the shortest path length as an integer, or -1 if no path exists. Target O(m×n) time and space complexity."

**Expected Output:** A Python function correctly implementing the BFS logic.

**Common Issue:** AI generates a solution with minor bugs (e.g., incorrect boundary checks) or a suboptimal approach if constraints aren't explicit.

## 4. Debug and Refine

**Action:** When tests fail, use AI to identify and fix issues iteratively.

**Prompt:** "The `test_no_path` test case fails for grid `[[0, 1], [1, 0]]`. My code returns `2` but the expected output is `-1`. Provide the minimal diff to fix this error in the `shortest_path_len` function." (Paste specific failing test output or a stack trace if helpful).

**Expected Output:** Small, targeted code modifications (diffs) to correct the failing logic.

**Common Issue:** AI might not fully understand the context without the entire code, or suggest changes that introduce new bugs.

## 5. Analysis and Documentation

**Action:** Request complexity analysis and real-world usage examples to complete the learning cycle.

**Prompt:** "Provide a detailed time and space complexity analysis for the final `shortest_path_len` BFS implementation. Additionally, suggest 2-3 real-world applications where a grid shortest path algorithm could be used."

**Expected Output:** Big O notation for time and space complexity, and relevant examples like robot navigation or game AI pathfinding.

**Common Issue:** Analysis might be too generic or lack specific details relevant to the implementation.

**Key Insight**: The AI assists your thinking process rather than replacing it. You guide the approach, verify the logic, and understand the tradeoffs—skills that directly transfer to interview success.

## Resources & Links

Your complete toolkit for continuing this AI-assisted DSA journey. These resources support both immediate practice and long-term interview preparation.

### Testing Framework

**VS Code Task**: "Session 2: Run Challenges (unittest)"
**Direct CLI**: Unittest discovery in challenges directory
Quick feedback loop for iterative development

### Code Repository

**Location**: sessions/session02_dsa/content/challenges/
Contains all challenge templates, test files, and solution examples with detailed comments

### Prompt Library

**Quick Reference**: This session's prompt patterns
**Extended Collection**: Challenge-specific READMEs
Save your best prompts for future use

### Python Standard Library

**Essential modules:** collections.deque, heapq, bisect, Counter
Master these built-ins for efficient implementations

## Next Steps

- **Commit your work**: Tests + code + complexity notes in README
- **Archive your prompts**: Keep successful prompt patterns in your repo
- **Optional challenge**: Micro-benchmark your LRU implementation and analyze O(1) average vs worst-case performance
- **Practice regularly**: Use this setup for daily coding practice with AI assistance