LESSON 64

Recursive Algorithms and Functions

80 minutes (2 periods) | Advanced Algorithms and Data Structures | Outcome 1.5 | Outcome 1.6 | Outcome 1.7 | Outcome 2.7 | Outcome 2.9 | Outcome 2.10 | Outcome 2.20

WHAT THIS LESSON DOES

In this lesson, you'll learn the essentials of recursive algorithms and functions. Explore how recursion solves problems by breaking them into smaller subproblems, understand base cases and recursive calls, and implement examples like factorial and Fibonacci in Python.

CURRICULUM OUTCOMES

Outcome	What the specification asks for
1.5	evaluate alternative solutions to computational problems
1.6	explain the operation of a variety of algorithms
1.7	develop algorithms to implement chosen solutions
2.7	implement algorithms using a programming language to solve a range of problems
2.9	assemble existing algorithms or create new ones that use functions (including recursive), procedures, and modules
2.10	explain the common measures of algorithmic efficiency using any algorithms studied
2.20	identify and fix/debug warnings and errors in computer code and modify as required

WHAT STUDENTS SHOULD BE ABLE TO DO

- Grasp the fundamental concepts of recursion, including base cases and recursive calls.
- Develop the ability to implement recursive functions for common problems like factorial and Fibonacci.
- Analyse the differences between recursive and iterative approaches in terms of efficiency and structure.
- Recognise recursion limits and potential issues like stack overflow, along with debugging strategies.
- Apply recursive techniques to solve practical programming challenges.

BEFORE THE LESSON

- Confirm Python and VS Code launch and run a simple script on the room machines, since students create a RecursionLesson folder and recursion.py from scratch.
- Run fibonacci(1000) and sys.getrecursionlimit() yourself first so you know exactly what the RecursionError looks like on your setup before students hit it.

Recursive Algorithms and Functions

HOW THE LESSON RUNS

Introduction
Understanding Recursion Basics
Difference Between Iterative and Recursive
Implementing Recursive Factorial
Iterative Factorial and Comparison
Implementing Recursive Fibonacci
Iterative Fibonacci and Comparison
Recursion Limits and Stack Overflow
Debugging Stack Overflow
Challenge
Wrap Up

TEACHING MOVES

- 2. Understanding Recursion Basics. Draw the two parts on the board before any code: base case as the stop, recursive case as the call to a smaller version. Stress that a missing or unreachable base case means infinite recursion, not an infinite loop.
- 3. Difference Between Iterative and Recursive. Keep this as a spoken contrast, not code. Name the call stack as the cost of recursion and the loop counter as the cost of iteration. Ask students to predict which factorial they think is faster before you show either.
- 4. Implementing Recursive Factorial. Have students trace factorial(3) by hand on paper before running it: show the calls stacking down to the base case, then the multiplications unwinding back up. This is where recursion clicks or does not.
- 5. Iterative Factorial and Comparison. Point out the iterative version has the same output but no stack growth. Ask which they find easier to read, and note the honest answer varies by person.
- 6. Implementing Recursive Fibonacci. Sketch the call tree for fibonacci(5) so students see the same subproblems recomputed many times. This visual sets up why the iterative version and memoisation matter later.
- 7. Iterative Fibonacci and Comparison. Contrast the two variables a and b holding state against the exponential branching of the recursive tree. Get students to run both for a larger n and notice the recursive one hang.
- 8. Recursion Limits and Stack Overflow. Let students trigger the RecursionError themselves and read the message aloud. Explain each call adds a stack frame, so depth, not total work, is what overflows.
- 9. Debugging Stack Overflow. Frame the checklist as questions: is the base case reachable, does every call move toward it, is the recursion genuinely deep or just wrong. Most stack overflows at this level are a broken base case, not deep data.
- 10. Challenge. Before anyone types, have them say the base case aloud: empty list returns 0. Then the recursive case: first element plus recursive_sum of the rest. Ban sum() and loops so recursion is the only route.

WHAT TO WATCH FOR

- Students omit or misdefine the base case and expect the function to stop on its own, then blame the language when it errors. Treat the base case as the first line written, not an afterthought. Have them state it before the recursive case every time, and trace a small input to confirm it is actually reached.
- Students think stack overflow means their computer is out of memory or the code is in an infinite loop. Explain it is the call stack depth limit specifically, around 1000 frames by default. Show sys.getrecursionlimit() so the number is concrete, and distinguish deep recursion from a genuine loop.
- Students assume recursive code is always slower or always better, treating it as a rule. Use the two Fibonacci versions as the counterexample: recursion is concise but here recomputes work exponentially, while iteration is efficient but less elegant. It depends on the problem.

DIFFERENTIATION

- Emerging: If the folder or file setup stalls them, give them recursion.py already created so they focus on the recursion, not VS Code navigation. Have them complete a factorial trace on paper with you before typing any code.
- Developing: Ask them to predict the output of factorial(4) in writing, then run it to check their trace matched the unwind.
- Proficient: Point them at memoisation for the recursive Fibonacci and have them measure the speed difference, the kind of efficiency reasoning that carries into a Coursework Project writeup. Ask them to extend recursive_sum to handle a nested list, and explain their new base and recursive cases back to you.