LESSON 63

Algorithmic Efficiency and Complexity

80 minutes (2 periods) | Advanced Algorithms and Data Structures | Outcome 2.7 | Outcome 2.8 | Outcome 2.9 | Outcome 2.10 | Outcome 1.6 | Outcome 1.7 | Outcome 2.19

WHAT THIS LESSON DOES

In this lesson, you'll explore time and space complexity along with Big O notation. You'll implement sorting algorithms in Python using VS Code, measure their performance, and analyse the results to understand algorithmic efficiency better.

CURRICULUM OUTCOMES

Outcome	What the specification asks for
2.7	implement algorithms using a programming language to solve a range of problems
2.8	apply basic search and sorting algorithms and describe the limitations and advantages of each algorithm
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
1.6	explain the operation of a variety of algorithms
1.7	develop algorithms to implement chosen solutions
2.19	test solutions and decisions to determine their short-term and long-term outcomes

WHAT STUDENTS SHOULD BE ABLE TO DO

- Understand the concepts of time and space complexity in algorithmic efficiency.
- Master Big O notation to describe algorithm performance.
- Implement various sorting algorithms in Python.
- Analyse and compare the performance of algorithms using timing data.
- Evaluate the trade-offs between time and space complexity in algorithm design.

BEFORE THE LESSON

- Confirm VS Code with the Python extension runs on the room machines, and that a script actually executes from the terminal, not just the editor.
- Run sorting.py yourself at all three sizes first. Timings vary by hardware, so know roughly what your room produces before students ask why theirs differ.
- Have the full sorting.py code saved somewhere the class can reach in case typing errors stall the timing runs.

Algorithmic Efficiency and Complexity

HOW THE LESSON RUNS

- Introduction
- Understanding Algorithmic Efficiency
- Time Complexity and Big O Notation
- Space Complexity
- Setting Up Your Project
- Implementing Bubble Sort
- Implementing Insertion Sort
- Implementing Quicksort
- Timing the Algorithms
- Analysing the Results
- Challenge
- Wrap Up

TEACHING MOVES

- 2. Understanding Algorithmic Efficiency. Anchor efficiency to something real: searching a contacts list of ten names versus ten million. Stress that speed and memory both matter, and that the difference only shows at scale.
- 3. Time Complexity and Big O Notation. Emphasise that Big O describes growth rate, not seconds. It ignores hardware and constants deliberately. Ask what happens to O(n) and O(n squared) when n doubles before showing any code.
- 4. Space Complexity. Separate auxiliary space from input space. Point out that the recursion stack itself counts as memory, which is why quicksort is not O(1) even though it looks in-place.
- 6. Implementing Bubble Sort. Walk one short list through the swap by hand before running it. Make sure students see why the inner loop shrinks by i each pass.
- 8. Implementing Quicksort. Highlight the base case first: an empty or single-element list is already sorted. Without it the recursion never stops. Trace the pivot split on a tiny list.
- 9. Timing the Algorithms. Stress the arr.copy() line. Each sort must run on the same starting list, or the second sort receives an already-sorted array and the comparison is meaningless.
- 10. Analysing the Results. Have students compute the ratio of the 5000 time to the 1000 time themselves. Compare it to 25 for the quadratic sorts and about 11 for quicksort. The number matters more than the raw seconds.
- 11. Challenge. Let students research merge sort rather than handing them the code. Ask them to predict the timing before running it, then check whether it tracks quicksort as O(n log n) predicts.

WHAT TO WATCH FOR

- Students read Big O as a measurement in seconds and expect the terminal output to match the notation exactly. Reinforce that Big O is a growth rate. Two O(n squared) algorithms can have very different absolute times but the same shape. The ratios reveal the class, not the clock.
- Students believe a faster time for a small list means an algorithm is better overall. Point at the 100-element row where all three are similar, then the 5000 row where they diverge. Efficiency claims only hold as n grows.
- Students think quicksort uses no extra memory because it looks like it sorts in place. Trace the recursion depth and the new left, middle and right lists it builds. Show that the recursion stack and those lists are the O(log n) and O(n) space.

DIFFERENTIATION

- Emerging: Give them the imports and one working function pasted in, and have them just run the timing block so they see output before writing any sort themselves. Pair them with a student whose environment already runs, so an editor or path problem does not consume the period.
- Developing: Ask them to predict which sort will be slowest at 5000 elements and explain why, before running the code.
- Proficient: Push them to complete the merge sort challenge and produce a short comparison table of measured ratios against Big O predictions, the kind of evidence and analysis that would strengthen a Coursework Project write-up. Ask them to explain why quicksort's worst case is O(n squared) and when that occurs, extending toward Higher Level treatment.