LESSON 10

Systematic Problem-Solving

40 minutes | Foundations of Computer Science | Outcome 1.1 | Outcome 1.3 | Outcome 1.4 | Outcome 1.7 | Outcome 2.5 | Outcome 2.6 | Outcome 2.21

WHAT THIS LESSON DOES

In this lesson, you'll explore systematic problem-solving methods essential for computer science. Learn to break down complex challenges into manageable steps using Polya's four-step approach and decision-making tools, applying these skills to practical computing problems.

CURRICULUM OUTCOMES

Outcome	What the specification asks for
1.1	describe a systematic process for solving problems and making decisions
1.3	solve problems by deconstructing them into smaller units using a systematic approach in an iterative fashion
1.4	solve problems using skills of logic
1.7	develop algorithms to implement chosen solutions
2.5	use pseudo code to outline the functionality of an algorithm
2.6	construct algorithms using appropriate sequences, selections/conditionals, loops and operators to solve a range of problems, to fulfil a specific requirement
2.21	critically reflect on and identify limitations in completed code and suggest possible improvements

WHAT STUDENTS SHOULD BE ABLE TO DO

- Grasp the concept of systematic problem-solving and its importance in computer science.
- Master Polya's four-step method for tackling complex problems logically.
- Develop skills in decision-making using tools like decision trees.
- Apply structured problem-solving techniques to practical computing challenges.
- Cultivate critical thinking and reflective evaluation for improving solutions.

BEFORE THE LESSON

- This is largely a theory and discussion lesson. The only hands-on part is the final activity, which students can do on paper or in a notes app, so no environment setup is essential.
- Have a simple worked example ready to model each Polya step live, ideally the average-of-numbers example the lesson uses, so students see the method in motion before they try it.

Systematic Problem-Solving

HOW THE LESSON RUNS

- Introduction
- Understanding Systematic Problem-Solving
- Polya's Four-Step Method
- Step #1 - Understand the Problem
- Step #2 - Devise a Plan
- Step #3 - Carry Out the Plan
- Step #4 - Look Back
- Decision-Making Processes
- Practical Activity: Apply to a Computing Problem
- Wrap Up

TEACHING MOVES

- 2. Understanding Systematic Problem-Solving. Contrast systematic work with trial-and-error explicitly. Ask for a time they fixed code by randomly changing things and it made it worse. That felt need for structure is what the rest of the lesson answers.
- 3. Polya's Four-Step Method. Name all four steps once, then resist detailing them here. Students will meet each in turn. Stress that Look Back is the step everyone skips and the one that separates a working solution from a good one.
- 4. Step #1 - Understand the Problem. Make them restate the problem in their own words out loud before anything else. Push for inputs, outputs and constraints named separately. Most weak solutions trace back to skipping this.
- 5. Step #2 - Devise a Plan. Insist the plan comes before code. Have them write pseudocode or sketch a flowchart for the average example. Point out a plan can be wrong cheaply on paper but expensively in code.
- 6. Step #3 - Carry Out the Plan. Model testing incrementally with sample data rather than writing everything then running once. Show that catching an error in one piece is far easier than in a finished program.
- 7. Step #4 - Look Back. Focus on edge cases: minimum, maximum and invalid inputs. Ask what breaks the average calculator if the list is empty. This is where robust thinking is trained.
- 8. Decision-Making Processes. Draw one decision tree live on the board with yes/no branches. Connect it to conditional logic they will meet in code, so the diagram is not just abstract but a picture of an if-statement.
- 9. Practical Activity: Apply to a Computing Problem. Note the spec quirk: output is Higher above 10, Even at exactly 10, Lower otherwise. Make students catch that the equal case is a distinct branch before they plan, so they design the condition correctly.
- 10. Wrap Up. Ask which of the four steps they are most tempted to skip and why. Land that the method is transferable to any problem, not just this course.

WHAT TO WATCH FOR

- Students jump straight to writing code, treating Understand and Plan as optional preamble. Refuse to look at code until they can show you a restated problem and a plan. Frame the two paper steps as the fastest route to working code, not a delay.
- In the activity, students read the spec as a simple two-way higher/lower split and miss the exact-equals-10 case entirely. Have them trace the value 10 through their plan by hand before coding. It exposes the missing third branch immediately.
- Look Back is treated as done the moment the program produces one correct answer. Give them a value that breaks it, or ask for an invalid input, and show that one passing test is not the same as a tested solution.

DIFFERENTIATION

- Emerging: Give them a single fixed input to walk through all four steps rather than the general problem, so the method is the focus not the arithmetic. Let them fill in a partly completed plan for the average example instead of starting from blank paper.
- Developing: Ask them to explain their plan back to you before they carry it out, checking they can justify each step rather than just follow it.
- Proficient: Have them extend the activity: add ranges (below 0, 0 to 10, above 10) and draw the decision tree for the extended version. Ask them to apply Polya to a problem from their own coursework thinking, producing written notes they could reuse as evidence in an Applied Learning Task.