Lesson 06: Pseudocode Puzzle - Making Sense of Algorithms (High-Inquiry Version)

Designer/Planner: Todd Edwards
Lesson Title: What Does This Code Do? Conjecturing Before Tracing
Intended Grade Level(s): Grades 8-12 (adaptable)
Content Area: Content-Agnostic Computational Thinking

I. Planning

Lesson Focus / Goals

State the big idea(s) of the lesson. Focus on conceptual understanding. Avoid vague objectives. Specify the knowledge and skills students should demonstrate.

The lesson aims to provide the following for students: - Generate conjectures about what code does BEFORE tracing line-by-line - Distinguish between procedural execution and conceptual meaning - Experience algorithms as patterns and structures, not just step-followers - Defend interpretations of code at multiple levels of abstraction - Understand that "understanding code" means seeing the structure, not just tracing steps

Learning Objectives

Write clear, measurable objectives. Include both procedural and conceptual goals. Consider potential misconceptions students might have.

By the end of the lesson, students will be able to: - Generate plausible conjectures about what code does based on structure and patterns - Describe what code does at multiple levels: procedurally ("it adds 1 five times") and abstractly ("it counts to 5") - Test conjectures by tracing or by trying different inputs - Compare different interpretations and evaluate which is more useful/general - Recognize that tracing is a tool for understanding, not understanding itself - Generalize from specific examples to broader algorithmic patterns

Potential Misconceptions: - Students might think code is too complex to interpret without line-by-line tracing - Students might equate "I can trace it" with "I understand it" - Students might think there's only ONE correct description of what code does - Students might not distinguish between procedural and functional descriptions - Students might think formal tracing is the only way to make sense of algorithms

Standards Alignment

Note: This is a content-agnostic lesson that places math and science students on equal footing. However, the reasoning processes naturally align to core practices in both disciplines.

Standards for Mathematical Practice (Common Core): - **MP7** – Look for and make use of structure.
Students recognize patterns in code structure and algorithmic logic. - **MP8** – Look for and express regularity in repeated reasoning.
Students identify how loops create repeated computational patterns.

NGSS Science and Engineering Practices: - **Using Mathematics and Computational Thinking** – Students trace algorithmic processes step-by-step. - **Developing and Using Models** – Students use trace tables to model computational processes.

II. Implementation

Materials Needed

List all physical and digital resources, manipulatives, and technology needed. For each item listed, provide a brief justification/explanation for its inclusion.

The following materials are used in the lesson: - **Pseudocode handout** with three code samples (no pre-made trace tables) - **Conjecture worksheets** with prompts: "What do you think this does?" "Why?" "How could you test that?" - **Blank paper** for student-designed testing/tracing if they choose - **Chart paper** for posting competing interpretations - **Sticky notes** for peer questions and alternative interpretations - **Optional: Additional input values** for students who want to test generalizations - **NO teacher solution guide** - interpretations emerge from student reasoning

Preparation: Print pseudocode handouts. Prepare conjecture worksheets that invite multiple-level descriptions. Do NOT prepare trace tables or worked solutions. Be ready to honor multiple valid interpretations.

Lesson Flow

(Before-During-After)

Organize your plan using the Before–During–After framework. Include approximate timing, key questions, and anticipated student responses.

Note for instructors: The core inquiry components are (1) conjecture generation before tracing, (2) comparison of multiple interpretations, and (3) the execution vs. understanding discussion. These three elements distinguish interpretation from execution. Some connection-making prompts are optional extensions. Prioritize students proposing, testing, and comparing their own interpretations over covering all reflection questions.

Before: (Launch – 7 min)

1. Display **Code Sample A** (simple but not trivial):

```
SET count = 0
REPEAT 5 times:
    ADD 1 to count
END REPEAT
DISPLAY count
```

2. Ask: “What do you think this code does? Just from reading it—don’t trace yet.”
3. Give 30 seconds silent think time
4. Collect 2-3 conjectures without judgment:
   - “It counts to 5”
   - “It adds numbers”
   - “It repeats something 5 times”
5. Key move: “These are all reasonable interpretations. How could we TEST which description is right? Or are they all right in different ways?”
6. One student suggests tracing—frame it as a choice: “Tracing is ONE tool for testing conjectures. You can also try different inputs, look for patterns, think about structure.”
7. Set up the work: “Today you’ll see 3 code samples. For each, your job is to figure out what it does—however makes sense to you. Then we’ll compare interpretations.”

During: (Explore – 23 min) Problem 1: Conditional (7 min)

```
SET number = 8
IF number > 10:
    DISPLAY "large"
ELSE:
    DISPLAY "small"
END IF
```

- **Conjecture phase (2 min):** Pairs write: “We think this code…”
- **Test phase (2 min):** “How can you verify? Try tracing, or try different values for ‘number’ ”
- **Compare (2 min):** Collect interpretations:
  - Procedural: “It checks if 8 is greater than 10, then displays ‘small’ ”
  - Functional: “It categorizes numbers as large or small”
- **Key question:** “Which description is more useful? Why?”
- Teacher names the tension: “Both are true. One describes WHAT IT DOES with this input. One describes WHAT KIND OF THING IT IS.”

Problem 2: More Complex Loop (8 min)

```
SET n = 4
SET result = 1
```

```
REPEAT n times:
    MULTIPLY result by 2
END REPEAT
DISPLAY result
```

- **Conjecture phase (3 min):** Groups work independently
  - "What do you think this does?"
  - Some might trace, some might just pattern-match
- **Test phase (2 min):** Teacher suggests: "What if n was 3? Or 5? Would that help you see the pattern?"
  - Groups try different inputs if they want
- **Compare interpretations (3 min):**
  - Display 3-4 different group descriptions on board
  - Example range:
    - "It multiplies 2 by itself 4 times" (procedural)
    - "It calculates 2 to the power of n" (abstract/mathematical)
    - "Result keeps doubling" (pattern-based)
  - Ask: "Are these all saying the same thing? Which captures the pattern best?"
  - **Encourage cross-group comparison:** "Does anyone disagree with one of these descriptions? Why? Which description would help you predict what happens when n = 10?"
- Teacher doesn't resolve—leaves tension

Problem 3: Surprising Pattern (8 min)

```
SET a = 1
SET b = 1
REPEAT 5 times:
    SET temp = a + b
    SET a = b
    SET b = temp
    DISPLAY b
END REPEAT
```

- **Conjecture phase (3 min):** This one is genuinely puzzling
  - Groups might struggle—that's okay
  - Might need to trace to see what's happening
- **Pattern recognition (3 min):**
  - Ask: "What numbers does this output?" (1, 2, 3, 5, 8)
  - "Do you recognize this pattern?"
  - If someone knows Fibonacci, great. If not, that's fine too.
  - **Important framing:** "Whether or not you know this pattern's name, you've understood what the code does by seeing the pattern. Naming it 'Fibonacci' doesn't make you understand it better—you already understood it by recognizing the structure."
- **Abstraction question (2 min):**
  - "How would you describe what this code IS, not just what it does?"

  - Surface: It's a sequence generator, a pattern-builder
  - Connect: "This is famous in math and nature—spirals, flower petals, rabbit populations"

After: (Meta-Discourse on Understanding Algorithms – 10 min) **Levels of Description (4 min)** - Display all 3 codes side-by-side - Ask: "For each one, we had different ways of describing what it does. What's the difference between these descriptions?" - Surface the distinction: - **Procedural/Execution:** "It checks if $8 > 10$, finds false, goes to else, displays 'small' " - **Functional/Abstract:** "It categorizes numbers" - **Pattern/Mathematical:** "It generates powers of 2" - Key question: "Which level of description is 'better'?" - Guide toward: Depends on your purpose! All are valid. - Abstraction is powerful for seeing connections and generalizing

Execution vs. Understanding (3 min) - "Can you trace code without understanding what it does?" - Yes—you can follow steps mechanically - "Can you understand what code does without tracing every line?" - Yes—by recognizing patterns and structures - Frame: "Understanding a system isn't about following its steps—it's about seeing the structure those steps create."

Connection to Earlier Lessons (2 min) - "Where else have we seen this?" - Headbandz: Following question sequence vs. understanding strategy - Fermi: Calculating vs. modeling - Marble maze: Tracing tests vs. building model - "What's the pattern?" $\rightarrow$ Inquiry lives in interpretation, not execution

Transfer Question (1 min) - "Where else do you encounter 'procedures' or 'algorithms' in math or science?" - Mathematical proofs, chemical reactions, data analysis protocols - "When is it important to understand the procedure? When is it important to understand the meaning?"

III. Assessment

Formative Assessment

Describe how you will check for understanding during and after the lesson.

During Conjecture Phases: - Observe whether students generate interpretations before tracing - Listen for multiple levels of description (procedural, functional, abstract) - Note which groups seek teacher validation vs. test their own conjectures - Check if students distinguish "what it does" from "what kind of thing it is"

During Comparison Discussions: - Monitor whether students recognize that different descriptions can be equally valid - Listen for language about abstraction and generalization - Observe whether students evaluate interpretations by usefulness, not just correctness - Note if students connect code patterns to mathematical/scientific structures

During Meta-Discourse: - Check if students distinguish execution from understanding - Listen for transfer to earlier lessons (Headbandz strategy, Fermi modeling, etc.) - Observe whether students see inquiry as interpretation, not just procedure-following

Exit Ticket: Students receive one more code sample:

```
SET x = 10
WHILE x > 0:
    DISPLAY x
    SUBTRACT 1 from x
END WHILE
```

They answer: 1. "Describe what this code does (procedural level)" 2. "Describe what this code does (abstract/functional level)" 3. "How are these two descriptions different? Which is more useful for seeing patterns?"

Self-Assessment: "I understand this code because I can: [trace it / describe its pattern / predict what happens with different inputs / explain what kind of thing it is]"

IV. Reflection & Next Steps

After teaching: Note what worked well and what didn't. Identify topics or skills to revisit. Record surprising student thinking. Suggest changes for next time.

I will aim to answer the following questions after the lesson has been taught: - Did students conjecture BEFORE tracing, or immediately resort to line-by-line execution? - How comfortable were students with multiple valid interpretations? - Did students distinguish procedural from abstract descriptions on their own? - What language did students use: "the answer" vs. "one way to describe it"? - Did Problem 3 (Fibonacci) genuinely puzzle students? How did they respond to mystery? - Which students recognized patterns without full tracing? What helped them? - Did the meta-discourse about execution vs. understanding land? - Did students connect to earlier lessons spontaneously or only when prompted? - How did students respond to NOT getting teacher-revealed "correct" interpretations? - Which groups struggled most? With conjecture generation? Pattern recognition? Abstraction?

Note: Please attach student handouts and any other printed materials that students will need to complete the lesson.

Student Conjecture & Interpretation Worksheet

Name/Group: ________________________ **Date:** ________________________

Pseudocode Puzzle: Making Sense of Algorithms

Your Challenge: For each code sample, figure out what it does. You can use ANY method that helps you understand: reading carefully, tracing some lines, trying different inputs, looking for patterns, drawing diagrams—whatever makes sense.

Problem 1: Conditional

Code:

```
SET number = 8
IF number > 10:
    DISPLAY "large"
ELSE:
    DISPLAY "small"
END IF
```

Initial Conjecture: What do you think this code does?

How did you figure that out? (Did you trace? Look at structure? Recognize a pattern?)

Describe what it does at TWO levels:

Procedural (what steps): ______________________________

Abstract (what kind of thing): ______________________________

Problem 2: Power Loop

Code:

```
SET n = 4
SET result = 1
REPEAT n times:
    MULTIPLY result by 2
END REPEAT
DISPLAY result
```

Initial Conjecture: What do you think this code does?

Test your conjecture: What if n = 3? What would result be?

Prediction: __________

After testing, revise if needed: This code...

____

Can you describe this using mathematical language?

____

Workspace for tracing/testing (if you want):

____

Problem 3: Mystery Pattern

Code:

```
SET a = 1
SET b = 1
REPEAT 5 times:
    SET temp = a + b
    SET a = b
    SET b = temp
    DISPLAY b
END REPEAT
```

Initial Conjecture: This one is tricky! What do you think happens?

____

Output values: What numbers get displayed? (Trace or test to find out)

____

Pattern recognition: Do you recognize this sequence?

____

What KIND of thing is this code? (A calculator? A counter? A pattern generator? Something else?)

____

Workspace for tracing/testing:

____

Reflection on Understanding

1. For which problem did you understand what it does WITHOUT fully tracing it? How?

____

2. What's the difference between these two statements? - "I can trace this code line-by-line" - "I understand what this code does"

____

____

3. Which level of description is most useful to you: procedural (what steps) or abstract (what pattern)? Why?

____

____

Exit Ticket

New Code:

```
SET x = 10
WHILE x > 0:
    DISPLAY x
    SUBTRACT 1 from x
END WHILE
```

1. Describe what this code does (procedural level – what steps):

____

____

2. Describe what this code does (abstract level – what pattern/function):

____

____

3. How are these two descriptions different? Which is more useful for seeing patterns?

____

____

4. Self-Assessment: I understand this code because I can (check all that apply):

Trace it line-by-line
Describe its pattern
Predict what happens with different inputs
Explain what kind of thing it is
Connect it to math or science concepts SET result = 1 REPEAT n times: MULTIPLY result by 2 END REPEAT DISPLAY result

```
**Trace Table:**

| After Line | n | result | Output |
|------------|---|--------|--------|
| 1 |  |  |  |
| 2 |  |  |  |
| 3 (1st time) |  |  |  |
| 3 (2nd time) |  |  |  |
| 3 (3rd time) |  |  |  |
| 3 (4th time) |  |  |  |
| 4 |  |  |  |

**What does this code do?** (Fill in after teacher explains)

--------------------------------------------------------------------

---

### Exit Ticket

**Trace this code and explain what it does:**

**Code:**
```

SET x = 3 SET y = 7 SET answer = x * y DISPLAY answer “‘

Trace Table:

After Line	x	y	answer	Output
1				
2				
3				
4				

What does this code do?

____

____

V. Further Revision Ideas

These are additional inquiry-enhancing moves suggested through analysis of the lesson’s revision capacity. While not included in this version, they represent growth opportunities for continued development.

Eliminate All Structure - Pure Conjecture (Dimension 3)

- Show code, ask “What does this do?” with NO scaffolding
- Let students struggle with where to even start
- After 3-5 minutes: “What would help you figure this out?”
- Students request tools (tracing, testing inputs, etc.) rather than receiving them
- Makes problem-before-method visceral and student-driven

Have Students Invent Contexts (Dimension 4)

- After understanding what code does technically, ask: “What real-world situation could this model?”
- Example: The conditional could be checking if a patient’s fever is dangerous, if a score passes a threshold, if a bridge load is safe
- Shifts from abstract-only to seeing code as models of phenomena
- Connects computational thinking to scientific/mathematical modeling

Compare Procedural vs. Functional Descriptions Formally (Dimension 1 & 9)

- Create two-column chart for each code: “What it DOES (steps)” vs. “What it IS (pattern/function)”
- Debate: “Which description is more useful? When?”
- Surface explicitly: Multiple valid levels of abstraction exist
- Connect to scientific practice: operational definitions vs. theoretical frameworks

Vary Inputs Systematically (Dimension 2 & 5)

- Don’t just test one alternative value—test a RANGE
- “What if $n = 0$? $n = 1$? $n = 100$?”
- Look for patterns across outputs
- Surfaces generalization and helps students see functional relationships
- Extends ceiling significantly: moving from specific to general understanding

Never Reveal “What It’s Called” (Dimension 8 & 9)

- Don’t name it as “power function” or “Fibonacci sequence”
- Let students invent their own names/categories
- Emphasizes that understanding comes from reasoning, not labeling
- Harder emotionally but epistemically authentic

Create Code from Descriptions (Reversal) (Dimension 6 & 2)

- After interpreting given code, reverse the task
- “Write pseudocode that counts down from 20”

- "Write code that adds up all numbers from 1 to 10"
- Students must make the abstraction→execution translation themselves
- Deepens understanding of algorithms as designed artifacts

Cross-Lesson Integration Protocol (Dimension 10)

- Explicitly compare code-reading to earlier lessons:
  - "How is tracing code like testing the marble maze?"
  - "How is describing what code 'is' vs. what it 'does' like Fermi modeling vs. calculating?"
  - "How is reading code like inferring from graphs?"
- Make the meta-pattern visible: Inquiry lives in interpretation, not execution
- Strengthens transfer

Capacity Analysis Summary

Why This Lesson Has Very High Revision Capacity:

This lesson is **procedurally airtight but epistemically deterministic** in its LOW version. Its constraint signature fills an important niche: - **Lesson 01 (WODB):** authority & closure - **Lesson 02 (Mystery Graphs):** interpretation vs. explanation - **Lesson 03 (Headbandz):** efficiency vs. agency - **Lesson 04 (Marble Maze):** procedure vs. uncertainty - **Lesson 05 (Fermi):** modeling vs. formula-following - **Lesson 06 (Pseudocode): execution vs. interpretation**

Key Insight: "Understanding a system isn't about following its steps—it's about seeing the structure those steps create."

Core Message: Many candidates equate "I can trace it" with "I understand it." This lesson exposes that assumption and reveals that **inquiry lives in abstraction, not accuracy of execution**.

High-Capacity Dimensions for Novice Revision:

1. **Curiosity & Genuine Puzzlement** (HIGH) - "What does this code do?" is inherently puzzling, especially Problem 3; inquiry suppressed by rapid teacher reveal and premature certainty; revision is largely subtractive

2. **Integration of Big Ideas** (HIGH) - Algorithms, iteration, abstraction are foundational; lesson already gestures at generality ("2 to the power of n"); novices can foreground abstraction over execution, connect to mathematical structures

3. **Low Floor / High Ceiling** (HIGH) - Entry fully supported; ceiling capped by fixed inputs/outputs/explanations; novices can vary inputs, ask "what if?", encourage generalization beyond specifics
4. **Connection-Making** (MEDIUM-HIGH) - This lesson connects naturally to Fermi decomposition, marble maze procedures, Headbandz decision trees; novices can make connections explicit, compare "following steps" across contexts
5. **Openness & Multiple Pathways** (MEDIUM-HIGH) - Code can be interpreted at multiple levels (line-by-line, pattern-based, functional); lesson enforces single teacher-revealed interpretation; allowing student-generated descriptions is manageable
6. **Causal Explanation** (MEDIUM-HIGH) - Cause-and-effect explicit (line $\rightarrow$ state change) but explanation procedural not conceptual; novices can ask WHY loop produces exponential growth, connect multiplication to powers, explain invariants

---

Medium Capacity Dimensions:

7. **Student Agency** (MEDIUM) - Students do work but not deciding; teacher determines what code "does", when understanding complete, which interpretations count; letting students propose meanings and defend to peers is doable
8. **Collaboration & Discourse** (MEDIUM) - Work primarily individual; peer interaction limited to checking; novices can have students compare interpretations, debate what program is FOR, justify without tracing every step
9. **Problem Before Method** (MEDIUM) - Problem compelling but method (tracing) front-loaded and treated as only way; requires letting students guess first, tolerating incorrect conjectures, delaying formal tracing—pedagogically challenging but revealing

---

Lower Capacity Dimensions (Still Valid):

10. **Context-Rich / Phenomena-Based** (LOW-MEDIUM) - Code is abstract and intentionally context-free; no external phenomenon to anchor; novices can invent contexts ("What could this model?") but dimension is less central here

---

Why This Is a Strong Capstone Low Anchor:

Three key teaching virtues:

1. **Surfaces beliefs about determinism and understanding** - Many candidates equate “I can trace it” with “I understand it”; this lesson exposes that assumption cleanly
2. **Distinguishes execution from meaning** - Subtle but powerful distinction; deeply relevant to inquiry across disciplines; code always “works” but inquiry comes from interpreting what KIND of thing it is
3. **Completes the set’s epistemic arc** - All 6 lessons together reveal: inquiry isn’t about following procedures accurately, but about **constructing and defending interpretations**

Pedagogical Leverage: This lesson teaches candidates that teaching students to trace code is not teaching them to understand algorithms. The transformation happens when students must INTERPRET structure and pattern, not just EXECUTE steps. Understanding becomes seeing, not following.

Integration Power: This lesson beautifully ties together the entire sequence—showing that the tension between procedure-following and sense-making appears everywhere: reasoning with visuals, graphs, questions, tests, estimates, and code.