AI Algorithm Evaluation: Lessons Learned

Pancake Sorting Challenge Follow-Up

Lucas P. Cordova, Ph.D.

This lecture debriefs the pancake sorting challenge and discusses the key insights about working with AI for algorithmic problem solving.

Table of contents

1 Activity Debrief

1.1 Your Experience with AI and Algorithms

What did we discover together?

Let's reflect on your pancake sorting adventure and extract the key insights about working with AI for algorithmic problem solving.

1.2 Common Challenges Faced

Based on your submissions, typical struggles included:

- AI generated standard sorting instead of flip operations
- Missing edge case handling for empty arrays
- Incorrect output format (just array vs. full result object)
- Flip operations that violated the "from index 0" constraint
- Overly complex solutions for simple cases
- Incomplete or missing flip sequence tracking

Sound familiar?

1.3 The Good News

What worked well across teams (observations):

- Groups with detailed prompts got significantly better results
- TRACE evaluation helped identify issues systematically
- Testing revealed problems that weren't obvious from code reading
- Most teams could improve AI output through iteration
- Reflection led to genuine insights about AI capabilities

2 Prompt Evolution

2.1 The Journey

Let's examine a real example of prompt evolution that demonstrates the power of iterative improvement.

Our scenario: Getting AI to implement proper pancake sorting

2.2 Iteration 1: The Basic Attempt

Student's first prompt:

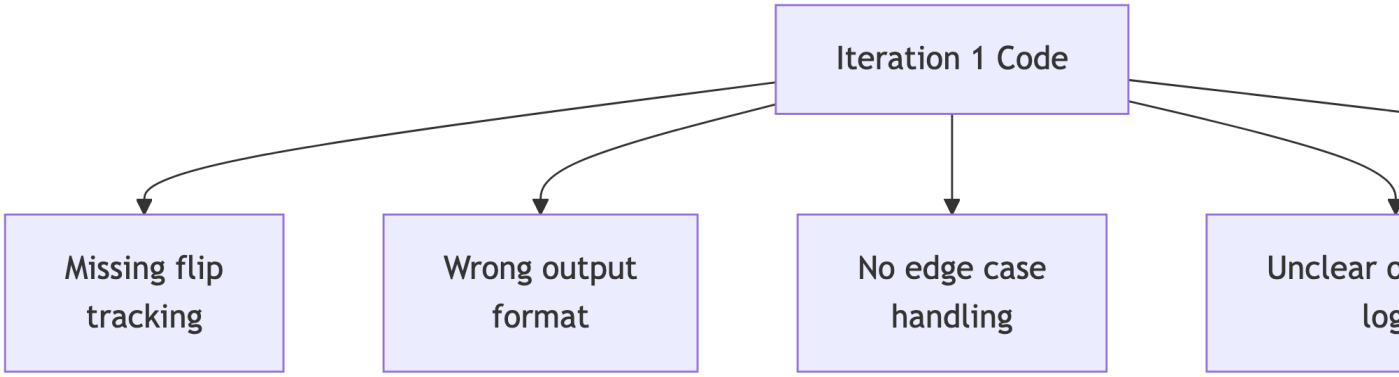
```
Write a pancake sort algorithm in Python.
```

AI Response:

```python
def pancake_sort(arr):
    for i in range(len(arr), 1, -1):
        max_idx = arr.index(max(arr[:i]))
        if max_idx != i-1:
            arr = arr[:max_idx+1][::-1] + arr[max_idx+1:]
            arr = arr[:i][::-1] + arr[i:]
    return arr
```

2.3 Problems with Iteration 1

TRACE Analysis revealed multiple issues:



Test Results:

- **Basic functionality:** FAILED (wrong return format)
- **Duplicates:** FAILED (logic errors)
- **Edge cases:** FAILED (crashes on empty array)
- **Flip validation:** FAILED (no operation tracking)

2.4 Iteration 2: Adding Context and Requirements

Improved prompt:

```
I need a pancake sorting algorithm for a robotic pancake flipper.
The robot can only flip pancakes from the top of the stack down to
any position. The function should:

- Take a list of pancake sizes
- Only use flip operations from index 0 to position k
- Return a dictionary with sorted array, flip sequence, and total flips
```

```
- Handle edge cases like empty arrays
- Work with duplicate values

Include the flip operation as a helper function.
```

2.5 Iteration 2 Results

AI Response (excerpt):

```python
def flip(arr, k):
    return arr[:k+1][::-1] + arr[k+1:]

def pancake_sort(pancakes):
    if len(pancakes) <= 1:
        return {
            'sorted_array': pancakes,
            'flip_sequence': [],
            'total_flips': 0
        }

    arr = pancakes.copy()
    flips = []

    for i in range(len(arr), 1, -1):
        max_idx = arr.index(max(arr[:i]))
        if max_idx != 0:
            arr = flip(arr, max_idx)
            flips.append(max_idx)
        if arr[0] != arr[i-1]:
            arr = flip(arr, i-1)
            flips.append(i-1)

    return {
        'sorted_array': arr,
        'flip_sequence': flips,
        'total_flips': len(flips)
    }
```

2.6 Progress in Iteration 2

Major improvements:

- Proper output format with required keys
- Dedicated flip helper function
- Basic edge case handling
- Flip sequence tracking
- Clear algorithm structure

Remaining issues:

- Still fails some duplicate tests
- Missing detailed validation
- No operations log for debugging
- Algorithm could be more efficient

2.7 Iteration 3: The Refined Solution

Final optimized prompt:

```
Create a comprehensive pancake sorting implementation for a robotic
breakfast system with these exact specifications:

CONTEXT: Robot arm can only flip pancakes from position 0 down to any
position k. This reverses elements from index 0 to k inclusive.

REQUIREMENTS:
- Function signature: pancake_sort(pancakes, options=None)
- Input: list of comparable elements (integers representing sizes)
- Output: dictionary with keys: 'sorted_array', 'flip_sequence',
  'total_flips', 'is_sorted', 'operations_log'

CONSTRAINTS:
- Only flip operations allowed (no swaps, inserts, etc.)
- All flips must start from index 0
- Must preserve all original elements
- Handle duplicates correctly

EDGE CASES:
- Empty array: return empty array, no flips
- Single element: return unchanged, no flips
- Already sorted: minimize flips needed
- All identical: handle efficiently

EXAMPLE:
```

```
Input: [3, 1, 2]
Process: Find max (3) at 0, flip to end → [2, 1, 3]
         Find max (2) at 0, flip to position → [1, 2, 3]
Output: {'sorted_array': [1, 2, 3], 'flip_sequence': [2, 1],
         'total_flips': 2, 'is_sorted': True}

Include comprehensive error checking and detailed comments
explaining the pancake sorting strategy.
```

2.8 Iteration 3 Results

AI delivered a robust solution:

```python
def pancake_sort(pancakes, options=None):
    """
    Pancake sorting algorithm - sorts by flipping from index 0 only.

    Args:
        pancakes: List of comparable elements
        options: Optional dict for algorithm parameters

    Returns:
        Dict with sorted_array, flip_sequence, total_flips,
        is_sorted, operations_log
    """
    # Input validation
    if not isinstance(pancakes, list):
        raise TypeError("Input must be a list")

    if len(pancakes) <= 1:
        return {
            'sorted_array': pancakes.copy(),
            'flip_sequence': [],
            'total_flips': 0,
            'is_sorted': True,
            'operations_log': []
        }

    arr = pancakes.copy()
    flips = []
    operations = []

```

```
    def flip(array, k):
        """Flip elements from index 0 to k inclusive"""
        return array[:k+1][::-1] + array[k+1:]

    # Main pancake sorting algorithm
    n = len(arr)
    for size in range(n, 1, -1):
        # Find position of largest unsorted element
        max_idx = 0
        for i in range(size):
            if arr[i] > arr[max_idx]:
                max_idx = i

        # If largest element is not at front, flip it to front
        if max_idx != 0:
            arr = flip(arr, max_idx)
            flips.append(max_idx)
            operations.append((max_idx, arr.copy()))

        # If largest element is not at correct position, flip to position
        if max_idx != size - 1 or arr[0] > arr[size - 1]:
            arr = flip(arr, size - 1)
            flips.append(size - 1)
            operations.append((size - 1, arr.copy()))

    return {
        'sorted_array': arr,
        'flip_sequence': flips,
        'total_flips': len(flips),
        'is_sorted': all(arr[i] <= arr[i+1] for i in range(len(arr)-1)),
        'operations_log': operations
    }
```

2.9 Test Results: Iteration 3

Comprehensive success:

- **Basic functionality:** PASSED
- **Duplicates:** PASSED
- **Reverse sorted:** PASSED
- **Edge cases:** PASSED
- **Flip validation:** PASSED

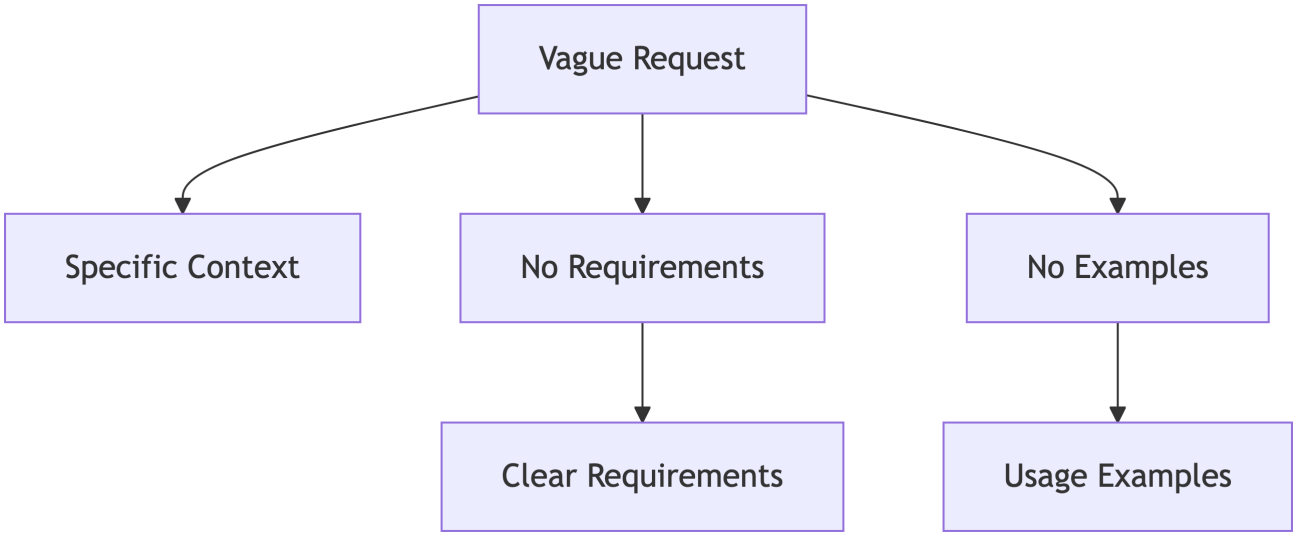
- **Identical elements:** PASSED

Performance: Efficient $O(n^2)$ implementation with proper flip tracking

3 Key Improvements Between Iterations

3.1 Iteration 1 → 2: Adding Context

What changed:



Impact:

- AI understood the domain (robotic pancake flipper)
- Generated appropriate constraints (flip from index 0)
- Included required output format
- Added basic edge case handling

3.2 Iteration 2 → 3: Comprehensive Specification

Major enhancements:

- **Detailed examples** with step-by-step walkthrough
- **Explicit edge cases** with expected behaviors
- **Precise input/output specifications**
- **Algorithm explanation** requirements
- **Error handling** requirements

- **Performance considerations**

The difference: Moving from "what" to "exactly how and why"

3.3 The Transformation Pattern

Common progression we observed:

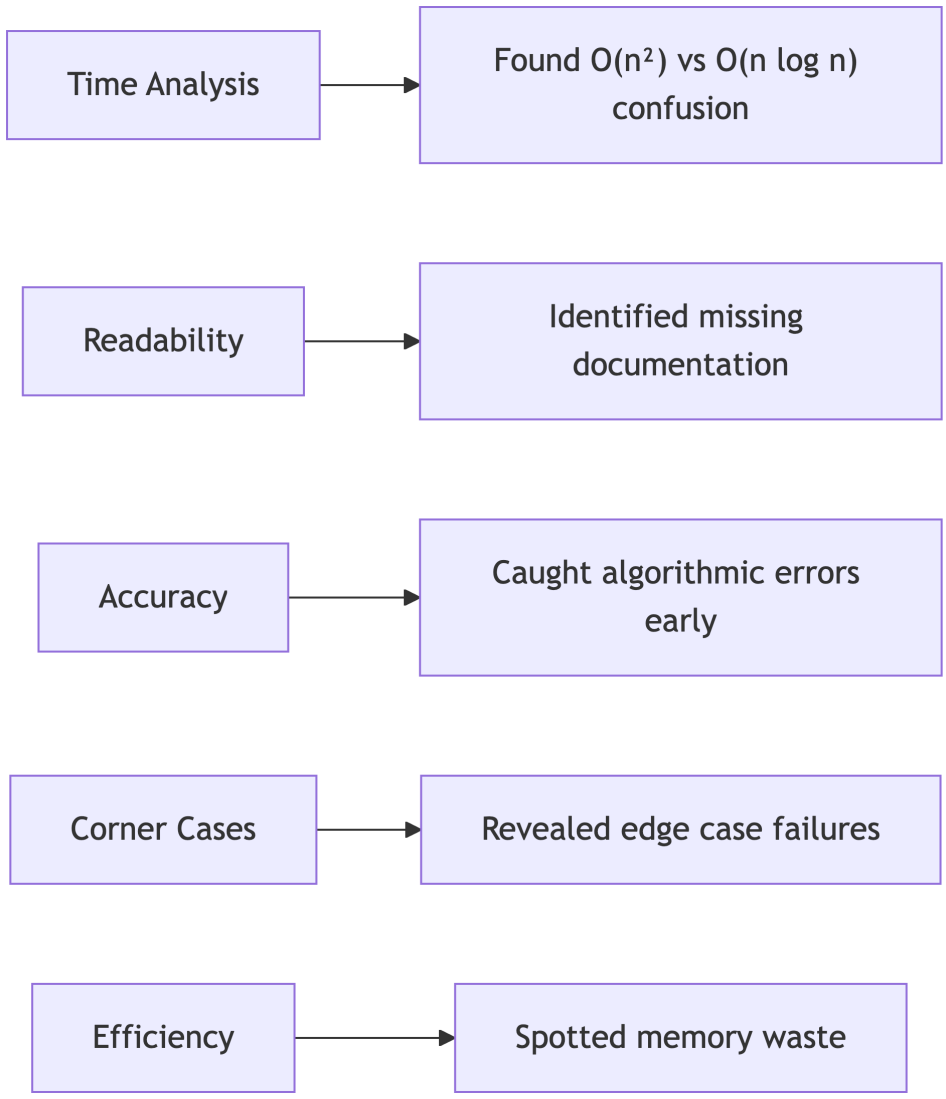
1. **Basic request** → Generic, often wrong solution
2. **Context added** → Domain-appropriate but incomplete
3. **Specifications detailed** → Robust, testable solution

Key insight: Specificity and context are multiplicative, not additive

4 Critical Insights

4.1 TRACE Method Effectiveness

What your submissions are revealing:



The TRACE method caught issues that manual inspection missed

4.2 Evident AI Strengths

Where AI excelled:

- **Pattern recognition** in algorithmic structures
- **Code generation speed** for well-specified problems
- **Syntax correctness** and basic error handling
- **Documentation** when specifically requested
- **Multiple approaches** when asked for alternatives

4.3 Evident AI Limitations

Consistent weak points:

- **Algorithm selection** without clear guidance
- **Edge case imagination** - needs explicit examples
- **Optimization decisions** without performance requirements
- **Testing strategy** - rarely suggests comprehensive tests
- **Domain knowledge** assumptions can be wrong

4.4 The Prompt Engineering Takeaways

What separates effective from ineffective prompts:

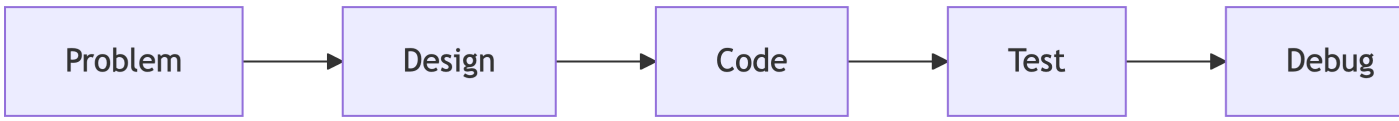
- **Specific examples** with expected outputs
- **Constraint explanation** with reasoning
- **Edge case enumeration** with handling requirements
- **Output format specification** with all required fields
- **Algorithm strategy** hints when appropriate

It's not just being detailed - it's being strategically detailed

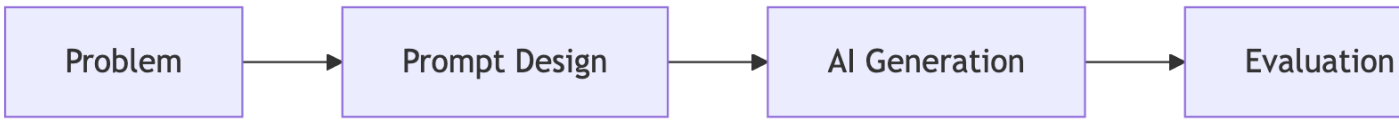
5 Broader Implications for Coding

5.1 The New Coding Workflow

Traditional approach:



AI-Assisted approach:



Key difference: Prompt design becomes as important as algorithm design

5.2 Skills That Matter More Now

Rising in importance:

- **Specification writing** - precise requirement articulation
- **Evaluation frameworks** - systematic quality assessment
- **Testing strategy** - comprehensive validation approaches
- **Prompt engineering** - effective AI communication
- **Critical analysis** - distinguishing good from adequate code

Still essential but changing:

- **Algorithm knowledge** - for evaluation and debugging
- **Code reading** - for AI output assessment
- **System design** - for integration planning

5.3 Quality Assurance Evolution

New responsibilities:

- **AI output validation** replaces some manual testing
- **Prompt version control** for reproducible results
- **Evaluation criteria definition** for consistent quality
- **AI model selection** for different problem types
- **Bias detection** in AI-generated solutions

6 Takeaways for Future Projects

6.1 The Prompt Engineering Playbook

For algorithmic problems, always include:

1. **Domain context** - what real problem this solves
2. **Operation constraints** - what's allowed and forbidden
3. **Input/output examples** - concrete illustrations
4. **Edge cases** - boundary conditions and expected handling
5. **Performance expectations** - time/space complexity needs
6. **Output format** - exact structure required
7. **Validation requirements** - how success is measured

6.2 The Evaluation Checklist

Before trusting any AI-generated algorithm:

- Apply systematic evaluation (TRACE or similar)
- Run comprehensive test suite including edge cases
- Verify algorithmic complexity claims
- Check for constraint violations
- Validate output format completeness
- Assess code maintainability and documentation

6.3 When to Use AI for Algorithms

Good candidates:

- Well-understood problem domains
- Standard algorithmic patterns with twists
- Implementation of known algorithms
- Code translation between languages
- Performance optimization of existing code

Proceed with caution:

- Novel algorithmic research
- Critical performance requirements
- Complex constraint satisfaction
- Domain-specific optimization
- Security-sensitive implementations

6.4 Building on the Relationship with Generative AI

Practice these techniques:

- **Iterative refinement** - start broad, narrow systematically
- **Specification decomposition** - break complex requirements down
- **Example-driven explanation** - show don't just tell
- **Constraint articulation** - be explicit about limitations
- **Quality measurement** - define success criteria upfront