QuickSort Analysis & AI-Generated Code Evaluation

CS 351 - Data Structures and Algorithms

Lucas P. Cordova, Ph.D.

This lecture explores the core of the QuickSort algorithm, focusing on the critical role of partitioning schemes; specifically, the Lomuto and Hoare methods. We compare their design, efficiency, and practical trade-offs, highlighting how partitioning choice impacts performance and implementation complexity. The session also introduces a structured approach to evaluating AI-generated code, emphasizing the importance of critical analysis, prompt engineering, and thorough testing. By the end, you'll understand how to select and analyze partitioning strategies, and how to apply best practices when leveraging AI tools for algorithm development.

Table of contents

0.1 Outline

Part 1: QuickSort Deep Dive

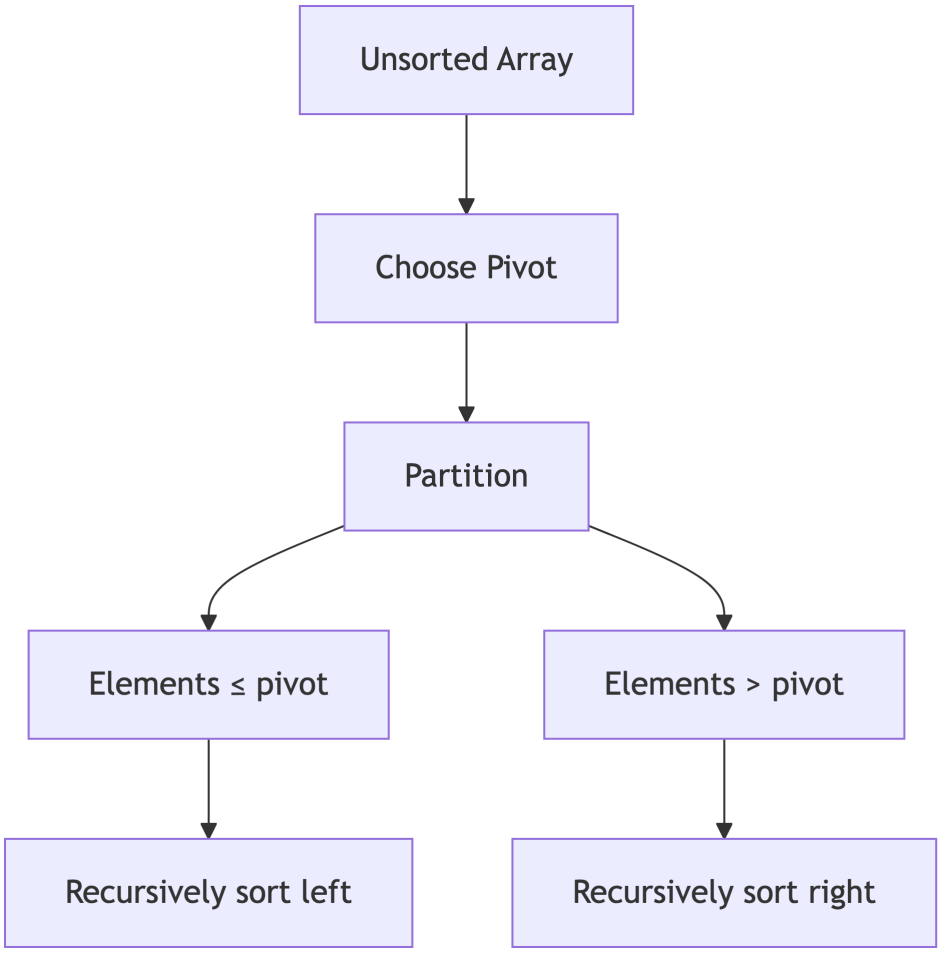
- Partitioning algorithms comparison
- Hands-on algorithm exploration
- Observation and reflection

Part 2: Evaluating AI-Generated Algorithms

- Critical analysis of AI code
- Prompt engineering for algorithms
- Interactive evaluation exercise

1 Part 1: QuickSort Partitioning Algorithms

Understanding the heart of QuickSort: **partitioning**



1.1 The Two Champions: Lomuto vs Hoare

Question for Discussion:

What makes a good partitioning algorithm?

Think about:

- Simplicity of implementation
- Number of swaps
- Cache performance

- Edge case handling

1.2 Lomuto Partitioning Scheme

Invented by: Nico Lomuto (1990s)
Key Idea: Single pointer scanning from left to right

```
def lomuto_partition(arr, low, high):
    pivot = arr[high]  # Choose last element as pivot
    i = low - 1        # Index of smaller element

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

Characteristics:

- Easy to understand and implement
- Always places pivot in final position
- Performs more swaps on average

1.3 Hoare Partitioning Scheme

Invented by: Tony Hoare (1961) - Original QuickSort creator
Key Idea: Two pointers moving toward each other

```
def hoare_partition(arr, low, high):
    pivot = arr[low]  # Choose first (or last) element as pivot
    i = low - 1
    j = high + 1

    while True:
        i += 1
        while arr[i] < pivot:
            i += 1

        j -= 1
        while arr[j] > pivot:
```

```
            j -= 1

        if i >= j:
            return j

        arr[i], arr[j] = arr[j], arr[i]
```

Characteristics:

- Fewer swaps on average
- Pivot not necessarily in final position
- More complex logic

1.4 Comparison Table

Aspect	Lomuto	Hoare
Simplicity	High	Medium
Swaps (average)	More	Fewer
Pivot placement	Exact final position	Approximate
Cache performance	Good	Better
Edge cases	Easier to handle	More complex

1.5 Exploration Time!

Activity 1: Algorithm Visualization

Visit: `https://courses.lpcordova.com/cs351/algoviz/quicksort.html`

Your Mission:

1. **Test QuickSort with Hoare partitioning**

Try these scenarios:

- Randomly generated data (default settings)
- Use own data option
  - Already sorted array: `[1, 2, 3, 4, 5, 6, 7, 8]`
  - Reverse sorted: `[8, 7, 6, 5, 4, 3, 2, 1]`
  - Random: `[3, 7, 1, 9, 4, 6, 8, 2]`
  - Duplicates: `[5, 3, 5, 1, 5, 7, 5]`

1.6 Reflection & Observations

Discussion Questions:

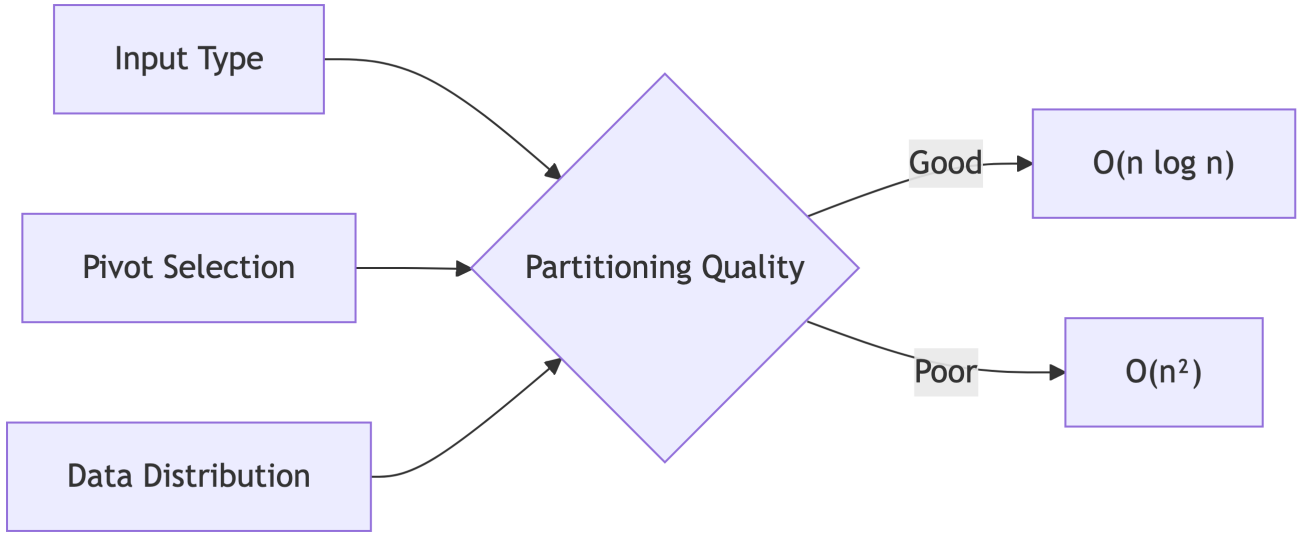
Group Discussion: What patterns did you notice?

Specific observations to consider:

- How did performance differ with sorted vs random data?
- Any unexpected behaviors or peculiarities?

1.7 Key Insights from QuickSort Analysis

Performance Characteristics:



Real-world implications:

- Choice of partitioning affects constant factors and simplicity
- Pivot selection strategy matters more than partitioning scheme
- Understanding algorithms deeply helps in optimization

2 Part 2: Evaluating AI-Generated Algorithms

The New Reality: AI can generate algorithm code instantly, but can we trust it?

Objectives:

- Develop critical evaluation skills
- Learn effective prompt engineering
- Understand AI limitations in algorithmic thinking

2.1 Case Study: AI-Generated Merge Sort

Scenario: You ask an AI to implement Merge Sort

Your prompt: *"Write a merge sort algorithm in Python"*

Let's see what we might get and how to evaluate it...

2.2 AI Response Example

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

2.3 Evaluation Framework

The TRACE Method for Algorithm Evaluation:

Time Complexity Analysis
Readability and Structure
Accuracy and Correctness
Corner Cases and Edge Handling
Efficiency and Space Usage

2.4 T - Time Complexity Analysis

Questions to ask:

- What's the theoretical time complexity?
- Does the implementation match the expected complexity?
- Are there hidden inefficiencies?

For our Merge Sort: - **Expected:** O(n log n) - **Implementation:** Correct recursive structure - **Hidden issue:** Array slicing creates copies (Python-specific)

2.5 R - Readability and Structure

Code Quality Checklist:

- Clear variable names
- Logical function decomposition
- Appropriate comments
- Consistent style

Our example assessment:

- Clear function separation
- Meaningful variable names
- Missing docstrings
- Clean, readable structure

2.6 A - Accuracy and Correctness

Testing Strategy:

```
def test_merge_sort():
    # Basic functionality
    assert merge_sort([3, 1, 4, 1, 5]) == [1, 1, 3, 4, 5]

    # Edge cases
    assert merge_sort([]) == []
    assert merge_sort([1]) == [1]

    # Duplicates
    assert merge_sort([2, 2, 2]) == [2, 2, 2]

    # Already sorted
    assert merge_sort([1, 2, 3, 4]) == [1, 2, 3, 4]
```

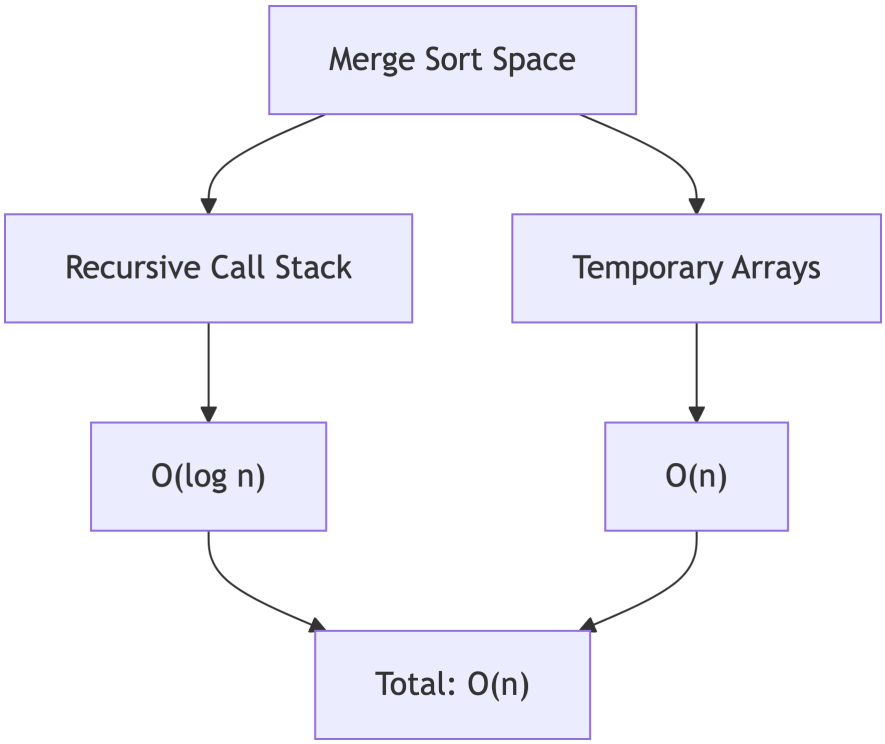
2.7 C - Corner Cases and Edge Handling

Critical scenarios to test:

- Empty arrays
- Single element arrays
- All elements identical
- Already sorted (best case)
- Reverse sorted (worst case for some algorithms)
- Very large datasets

2.8 E - Efficiency and Space Usage

Space Complexity Analysis:



Python-specific considerations:

- List slicing creates copies
- Memory overhead for temporary lists
- Garbage collection impact

2.9 Exercise: Prompt Engineering for Algorithms

Activity 2: Crafting Better Prompts

Basic Prompt: "Write a heap sort algorithm"

Enhanced Prompt: "Write an in-place heap sort algorithm in Python with the following requirements:

- Include time and space complexity analysis
- Add comprehensive docstrings
- Handle edge cases (empty array, single element)
- Include test cases
- Optimize for minimal memory usage
- Add comments explaining the heapify process"

Your turn: Work in pairs to improve this prompt for bubble sort

2.10 Prompt Engineering Strategies

The CLEAR Framework:

Context - Provide background and constraints
Length - Specify desired code length/complexity
Examples - Include input/output examples
Analysis - Request complexity analysis
Requirements - List specific technical requirements

2.11 Example: CLEAR Applied

Context: "I'm implementing sorting algorithms for a computer science course"

Length: "Write a concise but well-documented implementation"

Examples: "Should sort [64, 34, 25, 12, 22, 11, 90] to [11, 12, 22, 25, 34, 64, 90]"

Analysis: "Include time and space complexity in comments"

Requirements: "Use Python, include error handling, optimize for readability"

2.12 Red Flags in AI-Generated Code

Watch out for:

- **Infinite recursion** in recursive algorithms
- **Off-by-one errors** in array indexing
- **Memory leaks** in languages requiring manual management
- **Incorrect complexity claims** in comments
- **Missing edge case handling**
- **Non-optimal implementations** that work but are inefficient

2.13 Interactive Evaluation Exercise

Activity 3: Code Review Challenge

I'll show you an AI-generated selection sort. Use the TRACE method to evaluate it:

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Work in groups:

- Apply TRACE method
- Identify strengths and weaknesses
- Suggest improvements

2.14 Best Practices for AI Algorithm Assistance

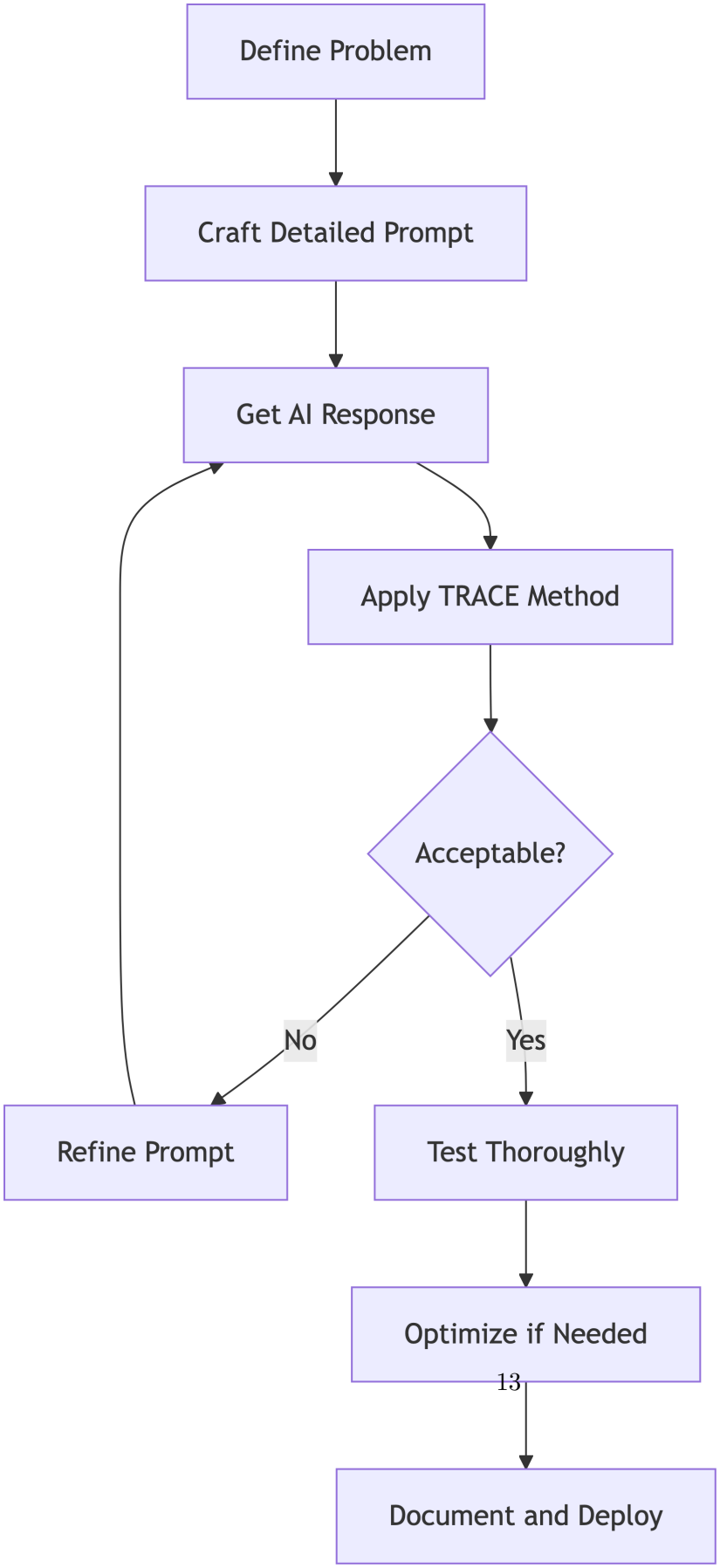
DO:

- Use AI for initial implementation ideas
- Ask for multiple approaches
- Request explanation of design choices
- Verify with your own test cases
- Cross-reference with authoritative sources

DON'T:

- Blindly trust AI output
- Skip manual testing
- Ignore complexity analysis
- Copy without understanding
- Forget to optimize for your specific use case

2.15 Practical Workflow



2.16 Summary: Key Takeaways

QuickSort Insights:

- Partitioning choice affects performance constants
- Hoare generally more efficient, Lomuto more intuitive
- Understanding internals helps optimization decisions

AI Algorithm Evaluation:

- Always apply critical analysis (TRACE method)
- Good prompts yield better results (CLEAR framework)
- AI is a tool, not a replacement for understanding
- Testing and verification are non-negotiable

2.17 Next Steps

For your QuickSort assignment:

1. **Apply today's knowledge** to analyze your implementation
2. **Use TRACE method** to evaluate any AI assistance
3. **Test thoroughly** with various input types
4. **Document your analysis** of partitioning choice

Questions to ponder:

- How might modern library implementations handle these trade-offs?
- What other factors might affect sorting performance in practice?
- How can profiling tools help validate theoretical analysis?