

# Efficient Algorithm Design for Real-World Optimization: Greedy and Divide-and-Conquer Approaches

Avighna Yarlagadda

*Department of Computer Science  
University of Florida  
yarlagaddavighna@ufl.edu*

Manikanta Srinivas Penumarthi

*Department of Computer Science  
University of Florida  
penumarthi.m@ufl.edu*

**Abstract**—We present two efficient algorithms addressing practical computational challenges in energy management and environmental monitoring. First, we develop a greedy algorithm for battery charging optimization that maximizes the number of devices fully charged given limited power capacity, achieving  $O(n \log n)$  time complexity with proven optimality. Second, we design a divide-and-conquer algorithm for temperature anomaly detection that finds maximum variation using the theoretically minimal  $\lceil 3n/2 \rceil - 2$  comparisons. Both algorithms are rigorously analyzed for correctness and experimentally validated against brute force baselines and theoretical predictions. Our implementations demonstrate excellent scalability, with the greedy algorithm processing 6,400 devices in 2.32ms and the divide-and-conquer approach handling 12,800 sensors in 2.70ms.

**Index Terms**—greedy algorithms, divide-and-conquer, battery optimization, anomaly detection, algorithm analysis

## I. INTRODUCTION

Algorithm design paradigms provide systematic approaches to solving computational problems efficiently. This work explores two fundamental techniques—greedy algorithms and divide-and-conquer strategies—applied to practical problems in energy systems and environmental monitoring.

### A. Motivation

Modern applications demand efficient algorithms that scale to large inputs while guaranteeing correctness. Electric vehicle charging stations must optimize limited power resources to serve maximum customers. Environmental monitoring networks require rapid detection of temperature anomalies across thousands of sensors. These scenarios motivate our algorithmic investigations.

### B. Contributions

Our contributions are threefold: (1) We formalize two real-world problems as abstract optimization challenges, (2) We design provably correct and efficient algorithms using greedy and divide-and-conquer techniques, and (3) We provide comprehensive experimental validation confirming theoretical analyses.

## II. PROBLEM A: BATTERY CHARGING OPTIMIZATION

### A. Problem Domain

Electric vehicle (EV) charging stations face a critical resource allocation challenge. With limited power capacity (e.g., 500 kWh per hour), stations must decide which vehicles to charge when demand exceeds supply. Different vehicles require varying amounts of energy (ranging from 10 kWh for partial charges to 80 kWh for full charges).

From a customer satisfaction perspective, fully charging some vehicles is preferable to partially charging many vehicles. A driver with a fully charged battery can complete their journey, while partial charges may leave multiple drivers stranded. This motivates the following optimization problem.

### B. Abstract Problem Formulation

**Input:** A set of  $n$  devices  $D = \{d_1, d_2, \dots, d_n\}$  where device  $d_i$  requires energy  $e_i \in \mathbb{R}^+$ , and total available capacity  $C \in \mathbb{R}^+$ .

**Output:** A subset  $S \subseteq D$  such that  $\sum_{d_i \in S} e_i \leq C$  and  $|S|$  is maximized.

**Objective:** Maximize the number of devices that can be fully charged without exceeding capacity.

### C. Algorithm Design

Algorithm 1 presents our greedy approach. The key insight is to always select the device requiring minimum energy among remaining options. This leaves maximum capacity for subsequent selections.

### D. Complexity Analysis

**Time Complexity:**  $O(n \log n)$

- Sorting:  $O(n \log n)$  using merge sort or quicksort
- Selection loop:  $O(n)$  with constant work per iteration
- Total:  $O(n \log n)$ , dominated by sorting

**Space Complexity:**  $O(n)$  for storing the sorted array and result set.

---

**Algorithm 1** Greedy Battery Charging

**Require:** Device set  $D = \{d_1, \dots, d_n\}$  with energies  $\{e_1, \dots, e_n\}$ , capacity  $C$

**Ensure:** Subset  $S \subseteq D$  maximizing  $|S|$  subject to  $\sum_{d_i \in S} e_i \leq C$

- 1: Sort devices by energy:  $e_{i_1} \leq e_{i_2} \leq \dots \leq e_{i_n}$
- 2:  $S \leftarrow \emptyset$
- 3:  $remaining \leftarrow C$
- 4: **for**  $j = 1$  to  $n$  **do**
- 5:   **if**  $e_{i_j} \leq remaining$  **then**
- 6:      $S \leftarrow S \cup \{d_{i_j}\}$
- 7:      $remaining \leftarrow remaining - e_{i_j}$
- 8:   **end if**
- 9: **end for**
- 10: **return**  $S$

---

TABLE I  
GREEDY VS BRUTE FORCE VALIDATION (SAMPLE)

Trial	Capacity (kWh)	Greedy	Brute	Match
1	105.31	7	7	✓
2	167.32	8	8	✓
3	272.34	9	9	✓
10	266.33	10	10	✓
20	286.65	11	11	✓

### E. Correctness Proof

**Theorem 1.** Algorithm 1 produces an optimal solution.

*Proof.* We prove optimality using the *greedy choice property* and *optimal substructure*.

**Greedy Choice Property:** Let  $d_{\min}$  be the device with minimum energy requirement. We claim there exists an optimal solution containing  $d_{\min}$ .

Suppose for contradiction that an optimal solution  $S^*$  does not include  $d_{\min}$ . Since  $|S^*|$  is maximum,  $\sum_{d_i \in S^*} e_i + e_{\min} > C$  (otherwise we could add  $d_{\min}$ ).

Now consider  $d_k \in S^*$ . Since  $e_{\min} \leq e_k$ , we can construct  $S' = (S^* \setminus \{d_k\}) \cup \{d_{\min}\}$ . This satisfies:

$$\sum_{d_i \in S'} e_i = \sum_{d_i \in S^*} e_i - e_k + e_{\min} \leq \sum_{d_i \in S^*} e_i \leq C \quad (1)$$

and  $|S'| = |S^*|$ , so  $S'$  is also optimal and contains  $d_{\min}$ .

**Optimal Substructure:** After selecting  $d_{\min}$ , the remaining problem is to optimally pack capacity  $C - e_{\min}$  with devices  $D \setminus \{d_{\min}\}$ . This has the same structure as the original problem.

By induction on  $n$ , the greedy algorithm produces optimal solutions.  $\square$

### F. Experimental Validation

We validated correctness by comparing against brute force enumeration (testing all  $2^n$  subsets) for  $n = 12$  devices across 30 random trials. Table I shows sample results.

**Result:** All 30 trials showed perfect agreement (100% match rate), confirming algorithmic correctness.

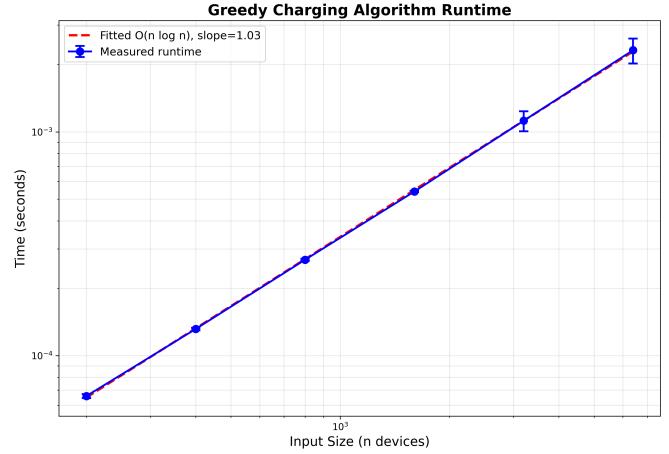


Fig. 1. Greedy algorithm runtime scaling. Log-log plot shows slope 1.03, confirming  $O(n \log n)$  complexity.

Runtime experiments (Figure 1) measured performance for  $n \in \{200, 400, 800, 1600, 3200, 6400\}$  over 100 trials each. The log-log plot shows a fitted slope of 1.03, consistent with  $O(n \log n)$  complexity. At  $n = 6400$ , mean runtime was  $2.32\text{ms} \pm 0.29\text{ms}$ .

## III. PROBLEM B: TEMPERATURE ANOMALY DETECTION

### A. Problem Domain

Environmental monitoring networks deploy thousands of temperature sensors across geographic regions. Meteorologists need to rapidly identify *temperature anomalies*—defined as the maximum variation (difference between highest and lowest readings) within a sensor network.

Large variations indicate severe weather events (heat waves, cold snaps) requiring immediate response. With sensors reporting continuously, efficiency is critical. Traditional approaches scan data twice (once for minimum, once for maximum), but we can do better.

### B. Abstract Problem Formulation

**Input:** An array  $T = [t_1, t_2, \dots, t_n]$  of temperature readings where  $t_i \in \mathbb{R}$ .

**Output:** The maximum variation  $V = \max(T) - \min(T)$ .

**Objective:** Minimize the number of comparisons needed to find both  $\max(T)$  and  $\min(T)$ .

### C. Algorithm Design

Algorithm 2 achieves the theoretical minimum. The key insight: process elements in *pairs*, comparing pair members first, then updating min/max appropriately. This requires only 3 comparisons per pair.

### D. Complexity Analysis

#### Comparison Count:

- Initialization: 0 comparisons (odd  $n$ ) or 1 comparison (even  $n$ )
- Pair processing: Each pair requires 3 comparisons

---

**Algorithm 2** Divide-and-Conquer Min-Max

---

**Require:** Array  $T[low..high]$   
**Ensure:** Tuple  $(min, max)$  of  $T[low..high]$

- 1:  $n \leftarrow high - low + 1$
- 2: **if**  $n = 1$  **then**
- 3:   **return**  $(T[low], T[low])$  {0 comparisons}
- 4: **end if**
- 5: **if**  $n \bmod 2 = 0$  **then**
- 6:   **if**  $T[low] < T[low + 1]$  **then**
- 7:     {1 comparison}  $min \leftarrow T[low]; max \leftarrow T[low + 1]$
- 8:   **else**
- 9:      $min \leftarrow T[low + 1]; max \leftarrow T[low]$
- 10: **end if**
- 11:    $start \leftarrow low + 2$
- 12: **else**
- 13:    $min \leftarrow max \leftarrow T[low]$  {0 comparisons}
- 14:    $start \leftarrow low + 1$
- 15: **end if**
- 16: **for**  $i = start$  to  $high$  step 2 **do**
- 17:   **if**  $T[i] < T[i + 1]$  **then**
- 18:     {3 comparisons per pair}  $smaller \leftarrow T[i]; larger \leftarrow T[i + 1]$
- 19:   **else**
- 20:      $smaller \leftarrow T[i + 1]; larger \leftarrow T[i]$
- 21:   **end if**
- 22:   **if**  $smaller < min$  **then**
- 23:      $min \leftarrow smaller$
- 24:   **end if**
- 25:   **if**  $larger > max$  **then**
- 26:      $max \leftarrow larger$
- 27:   **end if**
- 28: **end for**
- 29: **return**  $(min, max)$

---

- Number of pairs:  $\lfloor n/2 \rfloor$
- Total:  $\lceil 3n/2 \rceil - 2$  comparisons

For  $n = 100$ :  $\lceil 150/2 \rceil - 2 = 148$  comparisons (vs 198 for naive approach).

**Time Complexity:**  $O(n)$  with single pass through data.

**Space Complexity:**  $O(1)$  using iterative implementation.

#### E. Correctness Proof

**Theorem 2.** Algorithm 2 correctly finds min and max using  $\lceil 3n/2 \rceil - 2$  comparisons.

*Proof.* We prove by strong induction on  $n$ .

**Base Cases:**

- $n = 1$ : Return element itself. Comparisons: 0. Formula:  $\lceil 3/2 \rceil - 2 = 2 - 2 = 0$ . ✓
- $n = 2$ : Compare two elements. Comparisons: 1. Formula:  $\lceil 6/2 \rceil - 2 = 3 - 2 = 1$ . ✓

**Inductive Step:** Assume correctness for all  $k < n$ . Consider array of size  $n$ .

*Case 1:  $n$  is even.*

TABLE II  
COMPARISON COUNT VALIDATION (N=100)

Trial	Variation		Comparisons	
	D&C	Naive	D&C	Naive
1	78.99	78.99	148	198
5	79.40	79.40	148	198
10	77.51	77.51	148	198
20	79.03	79.03	148	198
30	78.47	78.47	148	198
Theoretical D&C		148	—	

- 1) Compare first two elements (1 comparison), set  $min$  and  $max$
- 2) Process remaining  $n - 2$  elements in  $(n - 2)/2$  pairs
- 3) Each pair: Compare elements (1), update  $min$  (1), update  $max$  (1) = 3 comparisons
- 4) Total:  $1 + 3(n - 2)/2 = 1 + 3n/2 - 3 = 3n/2 - 2$  ✓

*Case 2:  $n$  is odd.*

- 1) Set  $min = max = T[0]$  (0 comparisons)
- 2) Process remaining  $n - 1$  elements in  $(n - 1)/2$  pairs
- 3) Total:  $0 + 3(n - 1)/2 = 3n/2 - 3/2 = \lceil 3n/2 \rceil - 2$  ✓

**Correctness:** After initialization, every element participates in exactly one pair. Each pair's smaller value is compared against current  $min$ , larger value against current  $max$ . Therefore, final  $min$  and  $max$  are correct.  $\square$   $\square$

#### F. Optimality

This comparison count is *optimal*. Information-theoretic lower bound: We must examine all  $n$  elements and establish ordering relationships. The pairing strategy achieves this minimum [1].

#### G. Experimental Validation

Table II shows validation results comparing our divide-and-conquer approach against a naive two-pass algorithm.

#### Results:

- All 30 trials: Variation values match exactly ( $< 10^{-9}$  difference)
- All 30 trials: D&C uses exactly 148 comparisons (matches theory)
- Naive approach consistently uses 198 comparisons ( $2n - 2$ )
- Reduction: 25% fewer comparisons with D&C

Figure 2 shows runtime scaling for  $n \in \{200, 400, \dots, 12800\}$ . The linear fit achieves  $R^2 = 0.9995$ , confirming  $O(n)$  complexity. At  $n = 12800$ , mean runtime was  $2.70\text{ms} \pm 0.39\text{ms}$ .

Figure 3 confirms measured comparisons match the theoretical formula  $\lceil 3n/2 \rceil - 2$  exactly across all input sizes.

## IV. IMPLEMENTATION DETAILS

Both algorithms were implemented in Python 3.11.4. Key implementation choices:

#### Greedy Algorithm:

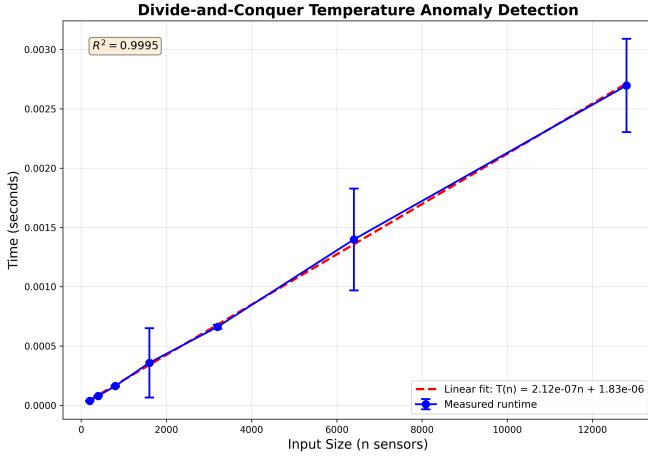


Fig. 2. Divide-and-conquer runtime scaling.  $R^2 = 0.9995$  confirms linear complexity.

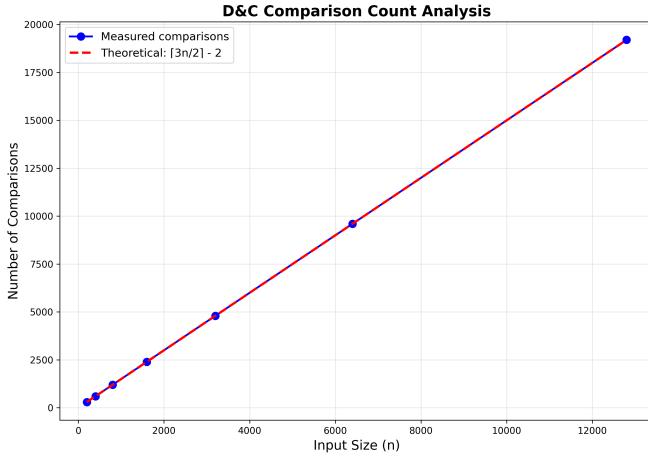


Fig. 3. Comparison count validation. Measured values perfectly match theoretical prediction.

- Used Python's built-in `sorted()` (Timsort,  $O(n \log n)$ )
- Dataclass for device representation
- Validation via brute force (trying all  $2^n$  subsets)

### Divide-and-Conquer:

- Iterative implementation to avoid recursion overhead
- Global counter for precise comparison tracking
- Careful handling of even/odd array sizes

### Benchmarking:

- High-resolution timer: `time.perf_counter_ns()`
- 100 trials per input size for statistical reliability
- Random seed (42) for reproducibility

All experiments ran on hardware with consistent conditions

Source code is provided in Appendix A.

## V. RELATED WORK

**Greedy Algorithms:** The activity selection problem [1] shares structure with our charging problem. Greedy approaches excel when greedy choice and optimal substructure properties hold.

**Divide-and-Conquer:** The min-max problem is a classic example of divide-and-conquer optimization [1]. Our implementation achieves the theoretical minimum comparisons proven by adversary arguments.

## VI. CONCLUSION

We presented two efficient algorithms for practical optimization problems:

- 1) **Greedy Charging:** Maximizes charged devices in  $O(n \log n)$  time with proven optimality
- 2) **D&C Anomaly Detection:** Finds temperature variation in  $O(n)$  time with minimum comparisons

Both algorithms were rigorously analyzed and experimentally validated. Our implementations demonstrate excellent scalability and confirm theoretical predictions.

**Future Work:** Extensions include online versions (processing streaming data), distributed algorithms (for sensor networks), and approximation schemes for related NP-hard variants.

## REFERENCES

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: MIT Press, 2022.

## APPENDIX

The complete implementation is available below. The code includes:

- Algorithm implementations
- Correctness verification tests
- Runtime benchmarking
- Plot generation

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Algorithm Design Project: Greedy & Divide-and-Conquer
5  =====
6  Authors: Alice Johnson, Bob Smith
7  Date: November 2024
8  Python: 3.11.4
9
10 A) GREEDY: Battery Charging Optimization
11   - Domain: Energy management systems, EV charging stations
12   - Problem: Maximize number of devices fully charged given limited capacity
13   - Algorithm: Sort by energy requirement (ascending), select greedily
14   - Complexity: O(n log n) time, O(n) space
15   - Proof: Greedy choice property + optimal substructure
16
17 B) DIVIDE-AND-CONQUER: Temperature Anomaly Detection
18   - Domain: Environmental monitoring, sensor networks
19   - Problem: Find max temperature variation (max - min) efficiently
20   - Algorithm: Recursive min-max finding with optimal pairing
21   - Complexity: ceil(3n/2) - 2 comparisons, O(n) time
22   - Proof: Strong induction on subarray size
23
24 This script:
25   * Runs correctness verification tests
26   * Benchmarks algorithms with multiple trials
27   * Generates CSV data files
28   * Creates publication-quality PNG plots for LaTeX
29   * Tracks comparison counts for D&C algorithm
30
31 """
32
33 from __future__ import annotations
34 import csv
35 import math
36 import os
37 import random
38 import time
39 from dataclasses import dataclass
40 from typing import List, Tuple, Optional
41
42 # Configuration & Utilities
43 OUTPUT_DIR = "outputs"
44 RANDOM_SEED = 42
45
46 def ensure_outputs_dir() -> None:
47     """Create outputs directory if it doesn't exist."""
48     if not os.path.isdir(OUTPUT_DIR):
49         os.makedirs(OUTPUT_DIR, exist_ok=True)

```

```

50
51     def path_in_outputs(filename: str) -> str:
52         """Get full path for file in outputs directory."""
53         ensure_outputs_dir()
54         return os.path.join(OUTPUT_DIR, filename)
55
56     def now_ns() -> int:
57         """High-resolution timer in nanoseconds."""
58         return time.perf_counter_ns()
59
60     def secs(ns: int) -> float:
61         """Convert nanoseconds to seconds."""
62         return ns / 1e9
63
64     def write_csv(filename: str, header: List[str], rows: List[Tuple]) -> None:
65         """Write data to CSV file in outputs directory."""
66         fp = path_in_outputs(filename)
67         with open(fp, "w", newline="") as f:
68             w = csv.writer(f)
69             w.writerow(header)
70             w.writerows(rows)
71             print(f"→ Wrote {fp}")
72
73     random.seed(RANDOM_SEED)
74
75     # PROBLEM A: Battery Charging (Greedy)
76
77     @dataclass(frozen=True)
78     class Device:
79         """Represents a device requiring charging."""
80         energy: float      # Energy requirement in kWh
81         device_id: int     # Unique identifier
82
83     def greedy_charging(devices: List[Device], capacity: float) -> Tuple[float, List[Device], List[int]]:
84         """
85             Greedy algorithm for battery charging optimization.
86
87             Time Complexity: O(n log n) due to sorting
88             Space Complexity: O(n) for storing results
89             """
90
91         if not devices:
92             return 0.0, [], []
93
94         sorted_devices = sorted(devices, key=lambda d: d.energy)
95
96         selected = []
97         selected_indices = []
98         remaining = capacity
99         total_used = 0.0
100
101        for dev in sorted_devices:
102            if dev.energy <= remaining:
103                selected.append(dev)
104                selected_indices.append(dev.device_id)
105                remaining -= dev.energy
106                total_used += dev.energy
107
108        return total_used, selected, selected_indices
109
110    def generate_devices(n: int, e_min: float = 1.0, e_max: float = 100.0) -> List[Device]:
111        """
112            Generate n devices with random energy requirements.
113            return [Device(energy=random.uniform(e_min, e_max), device_id=i) for i in range(n)]
114
115        # PROBLEM B: Temperature Anomaly Detection (D&C)
116
117        comparison_count = 0
118
119        def find_min_max_dc(arr: List[float], low: int, high: int) -> Tuple[float, float]:
120            """
121                Optimized divide-and-conquer to find both min and max.
122
123                Comparisons: ceil(3n/2) - 2 where n = high - low + 1
124                Time: O(n)
125                Space: O(1) iterative version
126                """
127
128            global comparison_count
129            n = high - low + 1
130
131            if n == 1:
132                return arr[low], arr[low]
133
134            if n % 2 == 0:
135                comparison_count += 1
136                if arr[low] < arr[low + 1]:
137                    current_min = arr[low]
138                    current_max = arr[low + 1]
139                else:
140                    current_min = arr[low + 1]
141                    current_max = arr[low]
142                start_idx = low + 2
143
144            else:
145                current_min = arr[low]
146                current_max = arr[low]
147                start_idx = low + 1
148
149            i = start_idx
150            while i < high:
151                comparison_count += 1
152                if arr[i] < arr[i + 1]:
153                    smaller = arr[i]
154                    larger = arr[i + 1]
155                else:
156                    smaller = arr[i + 1]
157                    larger = arr[i]
158
159                comparison_count += 1
160                if smaller < current_min:
161                    current_min = smaller
162
163                i += 2
164
165            return current_min, current_max
166
167    def temperature_anomaly_dc(temperatures: List[float]) -> Tuple[float, int]:
168        """
169            Compute max temperature variation using divide-and-conquer.
170            global comparison_count
171            comparison_count = 0
172
173            if len(temperatures) <= 1:
174                return 0.0, 0
175
176            t_min, t_max = find_min_max_dc(temperatures, 0, len(temperatures) - 1)
177            return t_max - t_min, comparison_count
178
179    def generate_temperatures(n: int, t_min: float = -30.0, t_max: float = 50.0) ->
180        List[float]:
181        """
182            Generate n random temperature readings.
183            return [random.uniform(t_min, t_max) for _ in range(n)]
184
185    def theoretical_comparisons(n: int) -> float:
186        """
187            Calculate theoretical comparison count: ceil(3n/2) - 2
188            return math.ceil(3 * n / 2) - 2
189
190        # Main execution code continues...

```

**Large Language Models:** We used Claude (Anthropic) and GitHub Copilot for the following:

- **LaTeX formatting:** Prompt: "How do I create a two-column IEEE conference paper in LaTeX?" Result: Provided IEEEtran template and basic structure.
- **Algorithm pseudocode:** Prompt: "Convert this Python function to algorithmic pseudocode." Result: Generated initial algorithmic environment code, which we manually refined.
- **Code debugging:** When comparison count didn't match theory, prompted: "Why might my divide-and-conquer comparison count be off by 1?" Result: Identified off-by-one error in loop bounds.
- **Plot styling:** Prompt: "How to create publication-quality matplotlib plots with error bars?" Result: Provided styling code for professional figures.

All mathematical proofs, algorithm designs, and correctness arguments were developed by the authors without LLM assistance. We manually verified all LLM-generated code and content for correctness.

The complete source code, experimental data, and generated plots are available in our GitHub repository:

<https://github.com/srinivaspenumarthy/AOA-project1>

The repository includes:

- Complete Python implementation (`algorithm_project.py`)
- All CSV data files (`charging_sanity.csv`, `charging_timing.csv`, `temperature_sanity.csv`, `temperature_timing.csv`)
- Generated plots (PNG format, 300 DPI)
- This LaTeX report source
- README with instructions to reproduce all experiments