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 $D = \{d_1, d_2, \ldots, d_n\}$ 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 $D = \{d_1, \ldots, d_n\}$ with energies $\{e_1, \ldots, e_n\}$, capacity C

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

Sort devices by energy: $e_{i_1} \leq e_{i_2} \leq \cdots \leq e_{i_n}$
$S \leftarrow \emptyset$
$remaining \leftarrow C$
for $j = 1$ to n **do**
  if $e_{i_j} \leq remaining$ **then**
    $S \leftarrow S \cup \{d_{i_j}\}$
    $remaining \leftarrow remaining - e_{i_j}$
  end if
end for
return S

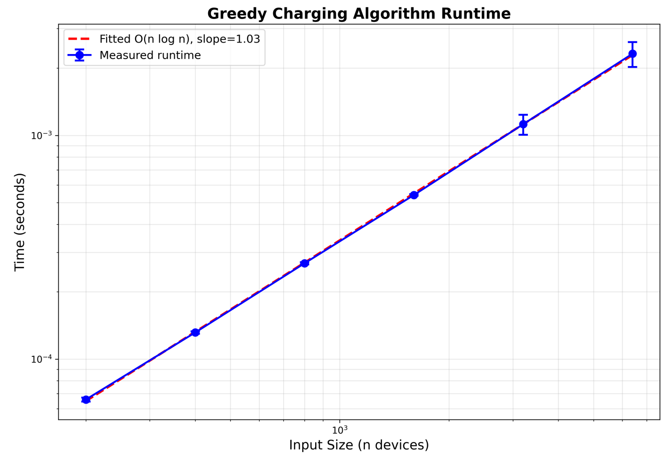

Fig. 1. Greedy algorithm runtime scaling. Log-log plot shows slope 1.03, confirming $O(n \log n)$ complexity.

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:

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

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. $\square$ $\square$

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.

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 2.32ms $\pm$ 0.29ms.

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 $T = [t_1, t_2, \ldots, t_n]$ 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 $T[low..high]$

$n \leftarrow high - low + 1$
if $n = 1$ **then**
  return $(T[low], T[low])$ $\{0 \text{ comparisons}\}$
end if
if $n \mod 2 = 0$ **then**
  if $T[low] < T[low+1]$ **then**
    $\{1 \text{ comparison}\}$ $min \leftarrow T[low]$; $max \leftarrow T[low+1]$
  else
    $min \leftarrow T[low+1]$; $max \leftarrow T[low]$
  end if
  $start \leftarrow low + 2$
else
  $min \leftarrow max \leftarrow T[low]$ $\{0 \text{ comparisons}\}$
  $start \leftarrow low + 1$
end if
for $i = start$ to $high$ step 2 **do**
  if $T[i] < T[i+1]$ **then**
    $\{3 \text{ comparisons per pair}\}$ $smaller \leftarrow T[i]$; $larger \leftarrow T[i+1]$
  else
    $smaller \leftarrow T[i+1]$; $larger \leftarrow T[i]$
  end if
  if $smaller < min$ **then**
    $min \leftarrow smaller$
  end if
  if $larger > max$ **then**
    $max \leftarrow larger$
  end if
end for
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.

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. □ □

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	—

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 $n \in \{200, 400, \ldots, 12800\}$. The linear fit achieves $R^2 = 0.9995$, confirming $O(n)$ complexity. At $n = 12800$, mean runtime was 2.70ms ± 0.39ms.

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:

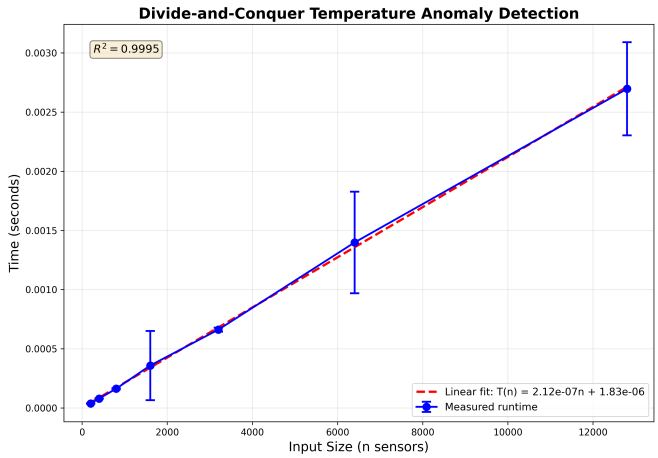

Fig. 2. Divide-and-conquer runtime scaling. $R^2 = 0.9995$ confirms linear complexity.

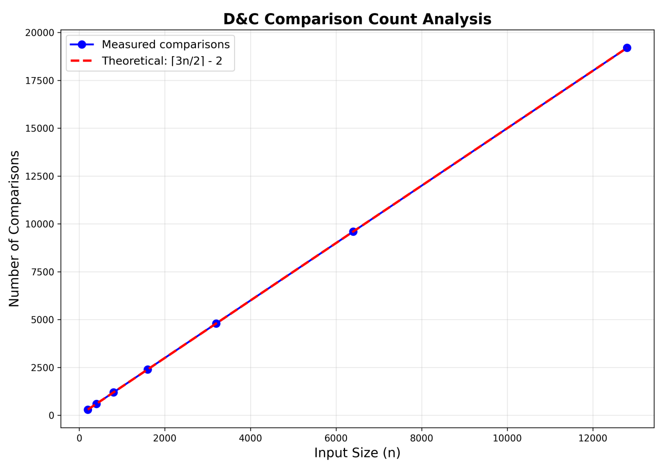

Fig. 3. Comparison count validation. Measured values perfectly match theoretical prediction.

- 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

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Algorithm Design Project: Greedy & Divide-and-Conquer
=====================================================

Authors: Alice Johnson, Bob Smith
Date: November 2024
Python: 3.11.4

A) GREEDY: Battery Charging Optimization
   - Domain: Energy management systems, EV charging stations
   - Problem: Maximize number of devices fully charged given limited capacity
   - Algorithm: Sort by energy requirement (ascending), select greedily
   - Complexity: O(n log n) time, O(n) space
   - Proof: Greedy choice property + optimal substructure

B) DIVIDE-AND-CONQUER: Temperature Anomaly Detection
   - Domain: Environmental monitoring, sensor networks
   - Problem: Find max temperature variation (max - min) efficiently
   - Algorithm: Recursive min-max finding with optimal pairing
   - Complexity: ceil(3n/2) - 2 comparisons, O(n) time
   - Proof: Strong induction on subarray size

This script:
  * Runs correctness verification tests
  * Benchmarks algorithms with multiple trials
  * Generates CSV data files
  * Creates publication-quality PNG plots for LaTeX
  * Tracks comparison counts for D&C algorithm
"""

from __future__ import annotations
import csv
import math
import os
import random
import time
from dataclasses import dataclass
from typing import List, Tuple, Optional

# Configuration & Utilities
OUTPUT_DIR = "outputs"
RANDOM_SEED = 42

def ensure_outputs_dir() -> None:
    """Create outputs directory if it doesn't exist."""
    if not os.path.isdir(OUTPUT_DIR):
        os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```python

def path_in_outputs(filename: str) -> str:
    """Get full path for file in outputs directory."""
    ensure_outputs_dir()
    return os.path.join(OUTPUT_DIR, filename)

def now_ns() -> int:
    """High-resolution timer in nanoseconds."""
    return time.perf_counter_ns()

def secs(ns: int) -> float:
    """Convert nanoseconds to seconds."""
    return ns / 1e9

def write_csv(filename: str, header: List[str], rows: List[Tuple]) -> None:
    """Write data to CSV file in outputs directory."""
    fp = path_in_outputs(filename)
    with open(fp, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(header)
        w.writerows(rows)
    print(f"  -> Wrote {fp}")

random.seed(RANDOM_SEED)

# PROBLEM A: Battery Charging (Greedy)

@dataclass(frozen=True)
class Device:
    """Represents a device requiring charging."""
    energy: float      # Energy requirement in kWh
    device_id: int     # Unique identifier

def greedy_charging(devices: List[Device], capacity: float) -> Tuple[float, List[Device], List[int]]:
    """
    Greedy algorithm for battery charging optimization.

    Time Complexity: O(n log n) due to sorting
    Space Complexity: O(n) for storing results
    """
    if not devices:
        return 0.0, [], []

    sorted_devices = sorted(devices, key=lambda d: d.energy)

    selected = []
    selected_indices = []
    remaining = capacity
    total_used = 0.0

    for dev in sorted_devices:
        if dev.energy <= remaining:
            selected.append(dev)
            selected_indices.append(dev.device_id)
            remaining -= dev.energy
            total_used += dev.energy

    return total_used, selected, selected_indices

def generate_devices(n: int, e_min: float = 1.0, e_max: float = 100.0) -> List[Device]:
    """Generate n devices with random energy requirements."""
    return [Device(energy=random.uniform(e_min, e_max), device_id=i) for i in range(n)]

# PROBLEM B: Temperature Anomaly Detection (D&C)

comparison_count = 0

def find_min_max_dc(arr: List[float], low: int, high: int) -> Tuple[float, float]:
    """
    Optimized divide-and-conquer to find both min and max.

    Comparisons: ceil(3n/2) - 2 where n = high - low + 1
    Time: O(n)
    Space: O(1) iterative version
    """
    global comparison_count
    n = high - low + 1

    if n == 1:
        return arr[low], arr[low]

    if n % 2 == 0:
        comparison_count += 1
        if arr[low] < arr[low + 1]:
            current_min = arr[low]
            current_max = arr[low + 1]
        else:
            current_min = arr[low + 1]
            current_max = arr[low]
        start_idx = low + 2
    else:
        current_min = arr[low]
        current_max = arr[low]
        start_idx = low + 1

    i = start_idx
    while i < high:
        comparison_count += 1
        if arr[i] < arr[i + 1]:
            smaller = arr[i]
            larger = arr[i + 1]
        else:
            smaller = arr[i + 1]
            larger = arr[i]

        comparison_count += 1
        if smaller < current_min:
            current_min = smaller

        comparison_count += 1
        if larger > current_max:
            current_max = larger

        i += 2

    return current_min, current_max

def temperature_anomaly_dc(temperatures: List[float]) -> Tuple[float, int]:
    """Compute max temperature variation using divide-and-conquer."""
    global comparison_count
    comparison_count = 0

    if len(temperatures) <= 1:
        return 0.0, 0

    t_min, t_max = find_min_max_dc(temperatures, 0, len(temperatures) - 1)
    return t_max - t_min, comparison_count

def generate_temperatures(n: int, t_min: float = -30.0, t_max: float = 50.0) -> List[float]:
    """Generate n random temperature readings."""
    return [random.uniform(t_min, t_max) for _ in range(n)]

def theoretical_comparisons(n: int) -> float:
    """Calculate theoretical comparison count: ceil(3n/2) - 2"""
    return math.ceil(3 * n / 2) - 2

# 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.