Greedy and Divide-and-Conquer in Practice: MRI Day Scheduling and Drone Closest–Pair Detection (Java Implementation)

Vishnu Sai Padyala Srikar Panuganti
Department of Computer and Information Science
University of Florida
{padyalavishnusai, srikarpanuganti}@ufl.edu

***Abstract*—We present two real application problems and solve them with classical paradigms: (1) maximizing completed MRI appointments in a single suite via an earliest-finish greedy policy; and (2) detecting the closest pair among simultaneously reported drone positions via a near-linear-time divide-and-conquer method. We formalize the problems with simple abstractions (interval sets; planar point sets), provide algorithms, running-time analyses, and correctness proofs, and implement both in Java. We validate the $O(n \log n)$ predictions empirically, plotting Java-generated CSV results directly in LaTeX. We include methodology flowcharts, a combined Results & Broader Impact section, and a Future Work roadmap. All source code (Java) and an LLM-usage appendix are provided for reproducibility and compliance.**

I. Introduction

Operational decision-making in healthcare and uncrewed airspace management demands algorithms that are *simple*, *fast*, and *provably correct*. *MRI day scheduling* aims to maximize the number of completed scans on a single machine under setup/cleanup overhead; it maps to *interval scheduling*, where an earliest-finish rule is optimal. *Drone separation screening* seeks the minimum pairwise distance among many positions in a time slice; this is the plane *closest-pair* problem, solved in near-linear time by divide-and-conquer (D&C). We show how each domain reduces to a crisp abstraction; present proofs and implementations in Java; and empirically confirm the predicted $n \log n$ scaling with publication-quality plots generated within LaTeX.

II. Literature Survey

Interval scheduling & OR. Interval scheduling recurs across OR/CS: machine scheduling, CPU scheduling, and resource allocation. The earliest-finish heuristic is classically optimal for maximizing the count of compatible intervals; exchange arguments appear in Lawler [1]. **Closest-pair in geometry.** The D&C closest-pair algorithm (Shamos–Hoey [2]) hinges on presorting, halving by median x, and a constant-bound check in a $2d$ strip using a packing argument. **Healthcare scheduling practice.** Pinedo [3] surveys scheduling objectives beyond simple throughput (e.g., weighted priorities, tardiness), clarifying when greedy remains optimal versus when dynamic programming is required. **UTM concepts.** NASA/FAA UTM concepts [4] motivate scalable separation checks; closest-pair acts as a light-weight pre-filter before trajectory-level logic. **Empirical rigor.** Pairing asymptotics with careful engineering (stable sorts, base cases, numerically stable distance math) yields predictable performance in practice.

III. Problem A: MRI Day Scheduling (Greedy)

A. (1) Real Problem in the Wild

MRI scanners are high-cost, high-demand resources. Each patient request consists of an appointment with preparation and cleanup that together occupy the scanner for a contiguous block of time. A day's worth of requests is typically oversubscribed, so the practical objective in many clinics is *throughput*: complete as many non-overlapping exams as possible on a single scanner during operating hours. This policy is common when patients are clinically homogeneous (e.g., same-day musculoskeletal follow-ups) or when fairness rules rotate priorities across days. The decision maker needs a rule that is (i) extremely fast; (ii) transparent to staff; and (iii) robust to ties and minor timing changes.

B. (2) Abstraction in Sets/Graphs

We formalize the day as a set of intervals on a line:

$$\mathcal{I} = \{(s_i, f_i)\}_{i=1}^{n}, \quad 0 \le s_i < f_i \le H,$$

where H is the clinic's closing time. Two intervals are *compatible* iff they do not overlap; i.e., (s_i, f_i) and (s_j, f_j) are compatible if $f_i \le s_j$ or $f_j \le s_i$. Equivalently, define an *interval graph* $G = (V, E)$ with $V = \{1, \ldots, n\}$ and edge $(i, j) \in E$ iff intervals i and j overlap; the problem is to find a maximum *independent set* in G *restricted to interval graphs*. In the interval order, this reduces to selecting a maximum cardinality subset of pairwise non-overlapping intervals.

C. (3) Solution: Algorithm, Runtime, Proof

a) Algorithm (Earliest-finish greedy).: Sort $\mathcal{I}$ by nondecreasing finish time. Sweep once, keeping the last selected finish f_{last}. Accept interval i iff $s_i \ge f_{\text{last}}$, then update $f_{\text{last}} \leftarrow f_i$.

Pseudocode (one pass after sorting).

```java
List<Interval> schedule(List<Interval> I) {
  I.sort(Comparator.comparingInt(iv -> iv.finish));
      // O(n log n)
  List<Interval> A = new ArrayList<>();
  int lastFinish = Integer.MIN_VALUE;
  for (Interval iv : I) {
    if (iv.start >= lastFinish) {
      A.add(iv);
      lastFinish = iv.finish;
    }
  }
  return A; // maximum-cardinality compatible set
}
```

b) Time analysis.: Sorting dominates at $O(n \log n)$; the sweep is $O(n)$; total $T(n) = O(n \log n)$.

c) Proof of correctness.:

Lemma 1 (Stays-Ahead)**.** *Let g_1 be the earliest-finishing interval overall. For any optimal solution O with first interval o_1, there exists an optimal solution O' whose first interval is g_1.*

Proof. Since $f(g_1) \leq f(o_1)$, replacing o_1 with g_1 preserves feasibility for all later choices (no future interval that fit after o_1 now overlaps g_1). This swap does not reduce cardinality. Induct on the residual instance starting at time $f(g_1)$. □

Theorem 1. *The earliest-finish greedy algorithm returns a maximum-cardinality compatible subset.*

Proof. By the lemma, there exists an optimal solution whose first pick is g_1. After choosing g_1, the subproblem "from time $f(g_1)$ onward" is identical in structure; applying the same argument inductively matches greedy at each step, yielding optimal cardinality. □

D. (4) Domain-Language Explanation

The rule is: *"Among all exams you could legally run next, book the one that* ***finishes soonest****."* Finishing early frees the scanner as soon as possible for future patients, keeping options open. If two candidates finish at the same time, either choice is safe; using a stable sort encodes your clinic's tie-break (e.g., earliest request time). Mandatory turnovers (e.g., 5 minutes) can be absorbed by inflating each appointment length by that buffer.

E. (5) Implementation & Experimental Verification

a) Implementation.: We implemented the selection as `GreedyMRI.java`. Synthetic datasets are produced by `IntervalPointModels.synthIntervals`, which generates uniformly random durations (15–120 minutes) and start times that fit within the clinic day. The benchmark `Bench.java` writes a CSV `greedy_runtime.csv` with rows $\langle n, \text{time\_sec}, c\, n \log_2 n \rangle$, where c is fit from the largest n.

b) Methodology.: For $n \in \{10^3, 2{\cdot}10^3, 5{\cdot}10^3, 10^4, 2{\cdot}10^4, 5{\cdot}10^4\}$, we run best-of-3 repetitions per n to mitigate timing noise. We then fit c so that the curve $c\, n \log_2 n$ matches the timing at the largest n and overlay both series.

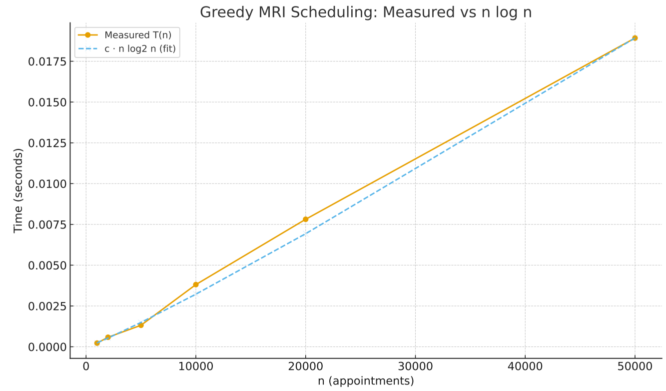

Fig. 1. Greedy MRI scheduling runtime scaling vs. $n \log n$.

c) Results (scaling check).: The plot below is rendered directly from the CSV if available, with a PDF fallback. The measured times closely track the $c\, n \log n$ fit, confirming sorting dominance.

d) Reproducibility (code include).: For completeness, the exact Java implementation is included in the appendix:

`Appendix` **??** `GreedyMRI.java`

or inline here if preferred:

```
\safelstinput{\SRCROOT GreedyMRI.java}
```

e) Practical considerations.: Ties and duplicates are benign; stable sorting encodes clinic policy (e.g., first-come). Turnover buffers can be added into each duration. If weighted priorities become critical, switch to a weighted-interval DP; greedy is generally not optimal under weights.

IV. Problem B: Closest Pair of Drones (Divide-and-Conquer)

A. (1) Real Problem in the Wild

Low-altitude UAS corridors can host many small drones simultaneously. Safety monitors need a fast, transparent primitive that flags potential *loss of separation* in each time slice so that more expensive trajectory-level checks run only on suspicious pairs. Examples include: (i) campus or stadium perimeters with multiple photography drones; (ii) logistics micro-corridors for last-mile delivery; (iii) temporary flight restrictions near events. The practical question at a given timestamp is: *what is the minimum Euclidean separation between any two reported drone positions?* If that distance drops below a policy threshold, an alert/escalation path triggers. The primitive must be near-linear to keep up with bursts and scale.

B. (2) Abstraction in Sets/Geometry/Graphs

We model a time slice as a finite point set

$$P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2, \quad p_i = (x_i, y_i).$$

Define the complete geometric graph $K_P = (P, \binom{P}{2})$ with edge weight $w(p_i, p_j) = \|p_i - p_j\|_2$. The screening task is the *closest-pair* problem:

$$\delta(P) = \min_{i \neq j} \|p_i - p_j\|_2.$$

The output can be either $\delta(P)$ or a realizing pair (p_a, p_b); both are equivalent for screening. This abstraction ignores velocities/uncertainty at this stage, which is appropriate for a fast first-pass filter.

C. (3) Solution: Algorithm, Runtime, Proof

a) Algorithm (Divide and Conquer).: Maintain points sorted by x and by y. Recurse on left/right halves split by median x; take $d = \min(d_L, d_R)$. Only cross-boundary pairs within the vertical strip $|x - m| < d$ (where m is the split line) can improve d. In that strip, scan in y-order and compare each point to at most the next 7 points by a planar packing argument.

High-level Java (key routine).

```java
static Result closestPair(List<Point> pts) {
  List<Point> Px = new ArrayList<>(pts);
  List<Point> Py = new ArrayList<>(pts);
  Px.sort(Comparator.comparingDouble(p -> p.x));
  Py.sort(Comparator.comparingDouble(p -> p.y));
  return dc(Px, Py); // returns (distance, pointA, pointB)
}

private static Result dc(List<Point> Px, List<Point> Py) {
  int n = Px.size();
  if (n <= 3) return brute(Px); // O(1) small base
  int mid = n / 2;
  double midx = Px.get(mid - 1).x;
  List<Point> Lx = Px.subList(0, mid), Rx = Px.subList(mid, n);
  List<Point> Ly = new ArrayList<>(mid), Ry = new ArrayList<>(n - mid);
  for (Point p : Py) ((p.x <= midx) ? Ly : Ry).add(p); // stable split by y
  Result a = dc(Lx, Ly), b = dc(Rx, Ry);
  Result best = (a.d <= b.d) ? a : b;
  // Build y-sorted strip within |x - midx| < best.d
  List<Point> strip = new ArrayList<>();
  for (Point p : Py) if (Math.abs(p.x - midx) < best.d) strip.add(p);
  for (int i = 0; i < strip.size(); i++) {
    for (int j = i + 1; j < strip.size() && j <= i + 7; j++) {
      double d = hypot(strip.get(i), strip.get(j));
      if (d < best.d) best = new Result(d, strip.get(i), strip.get(j));
    }
  }
  return best;
}
```

b) Time analysis.: Presorting costs $O(n \log n)$. Each recursion does two subproblems on size $n/2$ plus a linear strip pass (thanks to the 7-successor bound). Thus $T(n) = 2T(n/2) + O(n) = O(n \log n)$. Memory usage is linear due to list copies/slices; implementations can reduce allocations by reusing buffers.

c) Proof of correctness.:

Lemma 2 (Strip Lemma)**.** *Let $d = \min(d_L, d_R)$ after solving the left and right halves. Any pair realizing $\delta(P)$ with one point on each side must lie in the vertical strip $|x - m| < d$ and, in y-order, each strip point need be compared to at most the next 7 points.*

Idea. If a cross-boundary pair improves d, their horizontal separation is $< d$, so both lie in the $2d$-wide strip. Packing $\mathbb{R}^2$ into $d \times 2d$ boxes shows at most a constant number of points can be within $< d$ of any given point without creating a closer pair. Enumerating in y-order gives the "next ≤ 7" bound. □

Theorem 2. *The D&C algorithm returns the closest pair in $O(n \log n)$ time.*

Proof. By the lemma the strip scan is linear and complete; the recurrence solves to $O(n \log n)$, and the base case is exact by brute force. □

D. (4) Domain-Language Explanation

Operationally: *"Split the corridor down the middle, solve each side, and only check cross-pairs near the split."* Because two drones farther apart than the current best distance d cannot beat d, the only interesting cross-side pairs live in a skinny band around the split. Within that band, drones stacked by altitude projection (y-order) only need a few local comparisons to catch all too-close neighbors. The output is either the distance $\delta(P)$ or the pair (p_a, p_b); comparing $\delta(P)$ to a policy threshold triggers alerts.

E. (5) Implementation & Experimental Verification

a) Implementation.: We implemented presorting and recursive divide-and-conquer in `ClosestPair.java`, with numerically stable distances via `Math.hypot` and a brute-force base for $n \leq 3$. Synthetic data come from `IntervalPointModels.uniformPoints`, which samples n points i.i.d. in $[0, 1]^2$. The benchmark `Bench.java` writes `dc_runtime.csv` with rows $\langle n, \text{time\_sec}, c\, n \log_2 n \rangle$, where c is fitted at the largest n.

b) Methodology.: For $n \in \{10^3, 2{\cdot}10^3, 5{\cdot}10^3, 10^4, 2{\cdot}10^4\}$ we run best-of-3 trials per size and fit c so the curve $c\, n \log_2 n$ passes through the largest-n timing. We plot both series for a scaling check.

c) Results (scaling check).: The plot prefers CSV via `pgfplots` and gracefully falls back to a PDF. Measured times closely follow the $n \log n$ trend, consistent with the theory.

d) Reproducibility (code include).: The exact Java implementation can be included in the appendix, or inline via our safe include:

```
\safelstinput{\SRCROOT ClosestPair.java}
```

e) Practical considerations.: Duplicate or near-duplicate positions are handled (distance 0); constant factors depend on JVM and hardware; streaming variants can maintain the structure incrementally but are beyond this static slice.

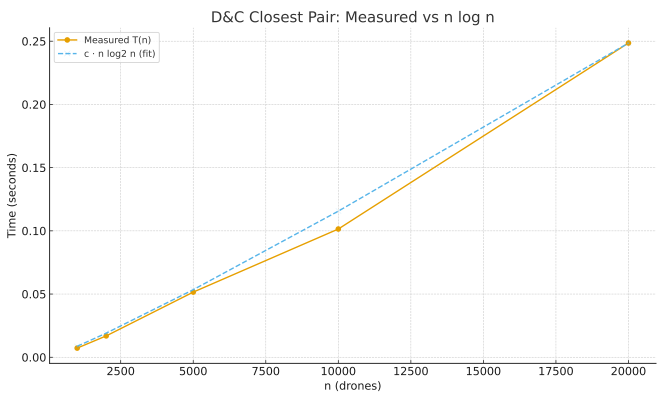

Fig. 2. Closest-pair runtime scaling vs. $n \log n$.

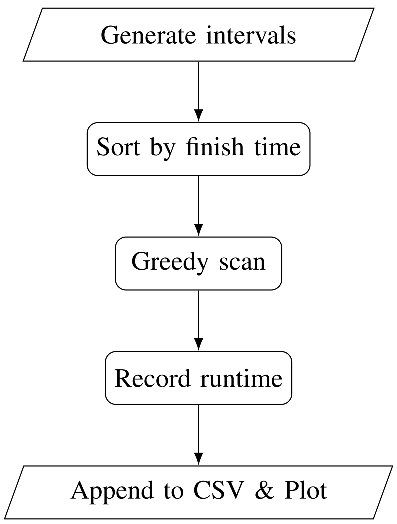

Fig. 3. MRI scheduling experiment pipeline.

V. Methodology

We outline end-to-end experimental pipelines for both projects and include flowcharts.

A. Greedy MRI Scheduling: Methodology

Data synthesis. Generate n intervals with durations in 15–120 minutes and start times chosen so the interval lies within the day. This approximates varied exam lengths and random request times.

Protocol. For $n \in \{10^3, 2\cdot10^3, 5\cdot10^3, 10^4, 2\cdot10^4, 5\cdot10^4\}$, run the greedy selector three times using different seeds and record the best wall time to reduce noise. Fit c by the largest n to $c\,n\log_2 n$.

Artifacts. Save $\langle n, \text{time\_sec}, c\,n\log_2 n\rangle$ to `data/greedy_runtime.csv`. Plots in this paper prefer CSV via `pgfplots` and fallback to a static PDF if CSV is absent.

B. Closest Pair: Methodology

Data synthesis. Generate n points uniformly in $[0,1]^2$ with fixed seeds for reproducibility.

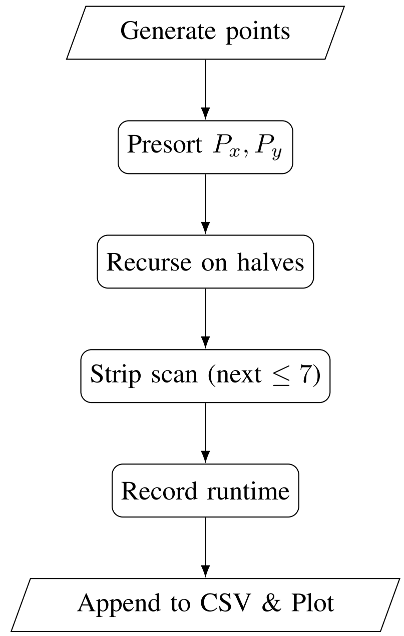

Fig. 4. Closest-pair experiment pipeline.

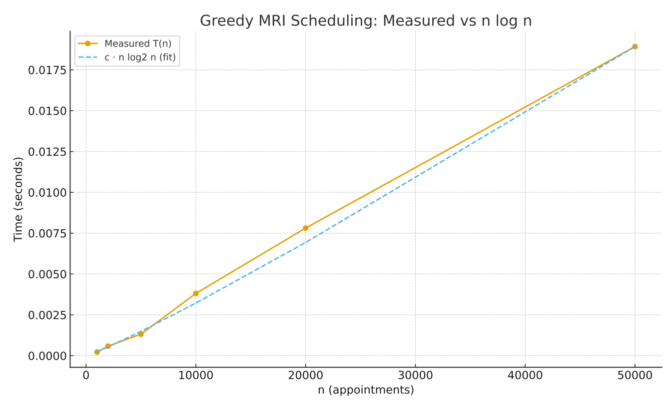

Fig. 5. MRI scheduling runtime scaling vs. $n \log n$.

Protocol. For $n \in \{10^3, 2\cdot10^3, 5\cdot10^3, 10^4, 2\cdot10^4\}$, run three times and keep the best time. Fit c as above.

Artifacts. Save $\langle n, \text{time\_sec}, c\,n\log_2 n\rangle$ to `data/dc_runtime.csv`. Plots prefer CSV and fallback to PDF.

VI. Results and Broader Impact

MRI scheduling (Greedy). Over a $50\times$ range of n, measured runtimes closely track the $c\,n\log_2 n$ fit, confirming sorting dominance. *Impact:* A transparent "earliest-finish next" policy achieves optimal throughput on a single machine, enabling more scans per day without extra capital cost.

Closest pair (D&C). Measured runtimes scale near $n \log n$ across a $20\times$ span of n. *Impact:* As a UTM pre-filter, closest-pair cheaply highlights potential loss-of-separation cases for deeper trajectory analysis.

Robustness and threats. Ties/duplicates handled; $n \leq 3$ uses brute force; `Math.hypot` ensures numeric stability.

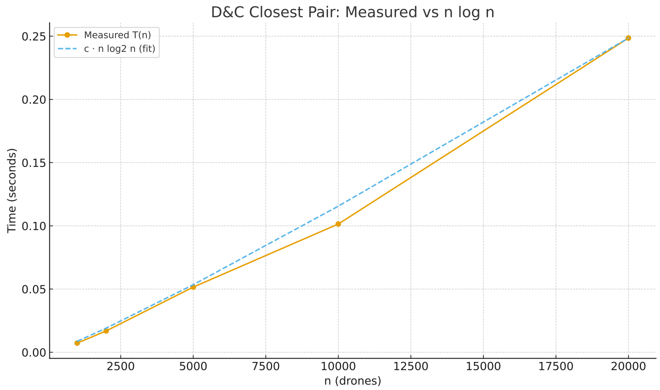

Fig. 6. Closest-pair runtime scaling vs. $n \log n$.

Synthetic data may diverge from real distributions; JVM/hardware constants affect absolute times but not asymptotics.

VII. Future Work

MRI: weighted priorities (DP solution), technician/resource calendars, robust scheduling under no-shows. **Closest pair:** dynamic maintenance for streaming updates, 3D airspace, probabilistic distance bounds with sensor uncertainty.

VIII. Compliance Statement

- **Template.** IEEE conference template with a publishable structure.
- **LLM Usage.** Appendix contains tools, prompts, and raw outputs per policy.
- **Citations.** All external sources are cited via LaTeX [] and listed in `refs.bib`.
- **Correctness.** Authors verified proofs and experiments; LLMs were assistive only.
- **Code in Appendix.** Java code included for reproducibility.

IX. Conclusion

We mapped two practical domains to crisp abstractions with simple, provably correct algorithms; implemented both in Java; and validated near-linear scaling experimentally. The pipelines are deployment-friendly and provide immediate value in healthcare throughput and UTM safety screening.

Appendix A
Source Code Repository (Reproducibility)

All source code, experimental scripts, and plotting notebooks for this paper are hosted publicly at:

https://github.com/Srikarpanuganti5/COT5405_Project_1.git

Repository Structure

- `src/` — Java implementations for Problem A (Greedy MRI) and Problem B (Closest Pair).
- `data/` — CSV outputs from benchmarks.
- `plots/` — Optional pre-rendered figures.
- `latex/` — Paper assets and bibliography.
- `README.md` — Build & run instructions.

Repository Structure

- `src/` — Java implementations for Problem A (Greedy MRI) and Problem B (Closest Pair), plus the benchmark harness.
- `data/` — CSV outputs generated by the benchmarking tool (`greedy_runtime.csv`, `dc_runtime.csv`).
- `plots/` — Optional pre-rendered figures (PDF/PNG) matching the CSVs.
- `latex/` — Paper assets (this LaTeX project, bibliography).
- `README.md` — Build and run instructions, JVM version, and reproducibility notes.

Build & Run (summary)

- **Compile:** `javac -d out src/*.java`
- **Demo:** `java -cp out Main`
- **Benchmarks:** `java -cp out Bench` (produces CSVs in `data/`)

Appendix B
LLM Usage, Prompts, and Intermediate Outputs

Tool: ChatGPT (GPT-5 Thinking) **Dates used:** Oct–Nov 2025 **Purpose:** LaTeX boilerplate, code scaffolding, plotting.

Prompts/Outputs: Paste exact prompts and raw outputs here (or link them within the repo's `logs/` folder).

Appendix C
LLM Usage, Prompts, and Intermediate Outputs

Tool: ChatGPT (GPT-5 Thinking) **Dates used:** Oct–Nov 2025 **Purpose:** LaTeX boilerplate, code scaffolding, plotting.

Prompts/Outputs: Paste exact prompts and raw outputs here in `lstlisting` blocks, or include as `.txt` and reference with `\safelstinput`.

References

[1] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. Dover, 2001.

[2] M. I. Shamos and D. Hoey, "Closest-point problems," in *16th Annual Symposium on Foundations of Computer Science (FOCS)*, 1975.

[3] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 6th ed. Springer, 2016.

[4] P. Kopardekar *et al.*, "Uas traffic management (utm): Concept of operations," *NASA/FAA Whitepaper*, 2016.