Drone Collision Prevention:
A Divide-and-Conquer Approach to the Top-k Closest Pair Problem

Abror Shopulatov
AI1225 - Algorithms and Data Structures
Mohamed bin Zayed University of Artificial Intelligence

February 2026

Abstract

The divide-and-conquer paradigm is one of the most fundamental methodologies in algorithmic analysis. In this work, we address the *top-k closest pair problem*, a classical application of this paradigm, with relevance to drone collision prevention systems. We present two variants of the problem: finding the single closest pair and the generalized top-k closest pairs. We implement and evaluate both brute-force and optimized algorithms across various configurations, including different numbers of drone locations and values of k. This report details our data generation methodology, provides a theoretical foundation for the baseline $O(n^2)$ solution, and demonstrates how we optimize our approach to achieve $O(n \log n)$ complexity. We present experimental results validating our theoretical analysis and discuss the practical implications of our findings.

1 Introduction

The objective of this assignment is to design and implement a Python application that processes drone location data and identifies pairs of drones in close proximity based on their spatial coordinates. For simplicity, we assume that drone locations are provided as a static snapshot, meaning drones do not move during the analysis period.

The closest pair problem is a fundamental problem in computational geometry with applications ranging from collision detection to clustering algorithms. Given n points in d-dimensional space, the goal is to find the pair of points with minimum Euclidean distance. The generalized top-k variant extends this to finding the k pairs with the smallest distances.

2 Data Preparation

As specified in the task description, we work with artificially generated 2D and 3D location data. We utilize NumPy's [Harris et al., 2020] `numpy.random.uniform` function to generate floating-point coordinates uniformly distributed between 0 and a specified width parameter.

We set the coordinate space width to 10,000 units. This choice is deliberate: selecting a width too small relative to the number of locations would increase the probability of point collisions, potentially leading to degenerate cases. A width of 10,000 provides sufficient space to avoid such issues while maintaining numerical precision.

For a given dimension $d \in \{2, 3\}$ and number of locations n, we generate data and store it as a pandas DataFrame [McKinney, 2010] for subsequent processing.

2.1 Distance Metric

The problem specification does not prescribe a particular distance metric but suggests real-world applicability. We therefore employ the Euclidean distance, defined for points $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$ as:

$$d(p, q) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2} \tag{1}$$

2.2 Tie-Breaking Strategy

To ensure deterministic results, we resolve ties by favoring pairs containing points with smaller indices. Specifically, given two pairs with identical distances, we select the pair whose first point has the smaller index; if these are equal, we select based on the second point's index.

2.3 Performance Measurement

We implemented helper functions to measure execution time and peak memory allocation using Python's built-in `time` and `tracemalloc` modules [Python Software Foundation, 2024]. All numerical computations leverage NumPy's vectorized operations for optimal performance.

3 Brute-Force Solution for Top-1

The brute-force approach exhaustively examines all possible point pairs to guarantee correctness. The algorithm iterates through every pair, maintaining the minimum distance observed and the corresponding pair information.

Algorithm 1 Brute-Force Closest Pair

```
 1: procedure FindClosestPair($P$)
 2:     $n \leftarrow |P|$
 3:     $d_{\min} \leftarrow \infty$
 4:     $closest\_pair \leftarrow$ NULL
 5:     for $i \leftarrow 0$ to $n - 1$ do
 6:         for $j \leftarrow 0$ to $i - 1$ do
 7:             $d \leftarrow$ Distance($P[i], P[j]$)
 8:             if $d < d_{\min}$ then
 9:                 $d_{\min} \leftarrow d$
10:                 $closest\_pair \leftarrow (P[i], P[j], d)$
11:             end if
12:         end for
13:     end for
14:     return $closest\_pair$
15: end procedure
```

Proposition 1 (Correctness). *Algorithm 1 correctly identifies the closest pair of points.*

Proof. The algorithm examines every unordered pair $(P[i], P[j])$ where $i > j$, which constitutes all $\binom{n}{2}$ possible pairs. Since the minimum distance pair must be among these pairs, and the algorithm tracks the global minimum, correctness follows directly. □

3.1 Time Complexity Analysis

The nested loops iterate over all pairs (i, j) where $0 \leq j < i < n$. The total number of iterations is:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2) \tag{2}$$

Each iteration performs a constant-time distance computation, yielding an overall time complexity of $O(n^2)$. This complexity applies uniformly to best, average, and worst cases, as the algorithm performs no early termination.

3.2 Space Complexity Analysis

The algorithm requires $O(n)$ space to store the input array and $O(1)$ auxiliary space for tracking the minimum distance and closest pair. The overall space complexity is therefore $O(n)$.

4 Brute-Force Solution for Top-k

Unlike the top-1 case, we cannot discard distance information as we require the k smallest distances. A naïve approach stores all pairwise distances and sorts them.

Algorithm 2 Naïve Brute-Force Top-k Closest Pairs

```
 1: procedure FindTopKPairs($P, k$)
 2:     $n \leftarrow |P|$
 3:     $all\_pairs \leftarrow []$
 4:     for $i \leftarrow 0$ to $n - 1$ do
 5:         for $j \leftarrow 0$ to $i - 1$ do
 6:             $d \leftarrow$ Distance($P[i], P[j]$)
 7:             $all\_pairs$.Append(($P[i], P[j], d$))
 8:         end for
 9:     end for
10:     $sorted\_pairs \leftarrow$ SortByDistance($all\_pairs$)
11:     return $sorted\_pairs[0 : k]$
12: end procedure
```

4.1 Time Complexity Analysis

Computing all pairwise distances requires $O(n^2)$ time. Sorting $m = \binom{n}{2} = O(n^2)$ elements requires $O(m \log m)$ time:

$$O(n^2 \log n^2) = O(n^2 \cdot 2 \log n) = O(n^2 \log n) \tag{3}$$

The sorting operation dominates, yielding an overall complexity of $O(n^2 \log n)$.

4.2 Space Complexity Analysis

Storing all $\binom{n}{2} = O(n^2)$ pairs requires $O(n^2)$ space.

5 Optimized Brute-Force Solution for Top-k

We observe that maintaining all pairs is unnecessary; we only require the k smallest. We therefore maintain a bounded data structure containing at most k pairs.

Algorithm 3 Optimized Brute-Force Top-k Closest Pairs

```
procedure FindTopKPairs($P, k$)
    $n \leftarrow |P|$
    $top\_k \leftarrow$ MaxHeap()                                ▷ Max-heap of size $k$
    for $i \leftarrow 0$ to $n - 1$ do
        for $j \leftarrow 0$ to $i - 1$ do
            $d \leftarrow$ Distance($P[i], P[j]$)
            if $|top\_k| < k$ then
                $top\_k$.Insert(($P[i], P[j], d$))
            else if $d < top\_k$.Max() then
                $top\_k$.ExtractMax()
                $top\_k$.Insert(($P[i], P[j], d$))
            end if
        end for
    end for
    return $top\_k$
end procedure
```

5.1 Time Complexity Analysis

The nested loops perform $O(n^2)$ iterations. Using a max-heap (e.g., Python's `heapq`), insertion and extraction operations each require $O(\log k)$ time. In the worst case, every pair triggers a heap update, yielding:

$$O(n^2 \log k) \tag{4}$$

Since typically $k \ll n$, this represents a significant improvement over the naïve $O(n^2 \log n)$ approach.

5.2 Space Complexity Analysis

The heap maintains at most k elements, requiring $O(k)$ auxiliary space. Combined with the input storage, the overall space complexity is $O(n + k) = O(n)$ when $k \leq n$.

6 Optimized Divide-and-Conquer Solution

We now present an optimized algorithm based on the divide-and-conquer paradigm, inspired by the classical closest pair algorithm [Shamos and Hoey, 1975, Cormen et al., 2009]. This approach achieves $O(n \log n)$ time complexity.

6.1 Algorithm Overview

The divide-and-conquer approach proceeds as follows:

1. **Base Case:** For $n \leq 3$ points, compute all pairwise distances directly.
2. **Divide:** Sort points by the x-coordinate and partition into two halves at the median.
3. **Conquer:** Recursively find the top-k closest pairs in each half.
4. **Combine:** Merge results and examine pairs spanning the dividing line.

The key insight is that when examining cross-boundary pairs, we can prune the search space significantly. Let δ denote the k-th smallest distance found so far. Any cross-boundary pair with distance less than δ must have both points within a vertical strip of width 2δ centered at the dividing line.

Lemma 1 (Strip Property)**.** *Within the strip of width 2δ, for any point p, there exist at most $O(1)$ points q in the opposite half such that $d(p, q) < \delta$.*

Proof. Consider a point p in the strip. Any point q from the opposite half with $d(p, q) < \delta$ must lie within a $\delta \times 2\delta$ rectangle (in 2D) or $\delta \times 2\delta \times 2\delta$ box (in 3D). Since all points in each half are at least δ apart (by the recursive solution), standard packing arguments show that at most 6 points (2D) or 27 points (3D) can fit in these regions [Preparata and Shamos, 1985]. □

Algorithm 4 Divide-and-Conquer Top-k Closest Pairs

```
 1: procedure ClosestKPairs($P, k$)
 2:     $n \leftarrow |P|$
 3:     if $n \leq 3$ then
 4:         return BaseCase($P, k$)
 5:     end if
 6:     $P_x \leftarrow$ SortByX($P$)
 7:     $mid \leftarrow \lfloor n/2 \rfloor$
 8:     $(L_k, \delta_L) \leftarrow$ ClosestKPairs($P_x[0 : mid], k$)
 9:     $(R_k, \delta_R) \leftarrow$ ClosestKPairs($P_x[mid : n], k$)
10:     $top\_k \leftarrow$ Merge($L_k, R_k, k$)
11:     $\delta \leftarrow top\_k[k].distance$
12:     $x_{mid} \leftarrow P_x[mid].x$
13:     $S \leftarrow \{p \in P : |p.x - x_{mid}| < \delta\}$
14:     $S_y \leftarrow$ SortByY($S$)
15:     for $i \leftarrow 0$ to $|S_y| - 1$ do
16:         for $j \leftarrow i + 1$ while $S_y[j].y - S_y[i].y < \delta$ do
17:             $d \leftarrow$ Distance($S_y[i], S_y[j]$)
18:             if $d < \delta$ then
19:                 UpdateTopK($top\_k, S_y[i], S_y[j], d$)
20:                 $\delta \leftarrow top\_k[k].distance$
21:             end if
22:         end for
23:     end for
24:     return $(top\_k, \delta)$
25: end procedure
```

6.2 Time Complexity Analysis

Theorem 1. *Algorithm 4 runs in $O(n \log^2 n)$ time, or $O(n \log n)$ with pre-sorting.*

Proof. Let $T(n)$ denote the running time. The algorithm performs:

- Sorting by x-coordinate: $O(n \log n)$
- Two recursive calls on subproblems of size $n/2$: $2T(n/2)$
- Constructing the strip and sorting by y: $O(n \log n)$
- Processing strip points: By Lemma 1, each point is compared with $O(1)$ others, yielding $O(n)$

The recurrence relation is:

$$T(n) = 2T(n/2) + O(n \log n) \tag{5}$$

By the Master Theorem (Case 2 with log factor), this solves to $T(n) = O(n \log^2 n)$.

With pre-sorting optimization (maintaining y-sorted order across recursive calls), the combine step reduces to $O(n)$, yielding:

$$T(n) = 2T(n/2) + O(n) = O(n \log n) \tag{6}$$

□

6.3 Space Complexity Analysis

The algorithm requires:

- $O(n)$ for the input and sorted arrays
- $O(\log n)$ recursion depth, each level using $O(n)$ space for strip arrays

The overall space complexity is $O(n \log n)$ without optimization, or $O(n)$ with careful memory management.

6.4 Correctness

Theorem 2. *Algorithm 4 correctly computes the top-k closest pairs.*

Proof. By strong induction on n. The base case ($n \leq 3$) is handled by exhaustive enumeration.

For the inductive step, assume correctness for all inputs of size less than n. The top-k closest pairs must either:

1. Both lie in the left half (found by recursive call)
2. Both lie in the right half (found by recursive call)
3. Span the dividing line (found by strip processing)

The strip processing examines all pairs (p, q) where p and q are on opposite sides and within distance δ of the dividing line. By Lemma 1, no closer cross-boundary pairs exist outside this strip. The merge operation correctly combines all candidates, establishing correctness. □

7 Extension to Dynamic Locations

For scenarios where drone locations change over time, we propose an incremental update strategy. Rather than recomputing from scratch, we track which points have moved and update distances accordingly.

7.1 Approach

1. Maintain the current top-k pairs and their distances
2. When locations change, identify affected pairs (those containing moved points)
3. Remove invalid pairs from the top-k set
4. Recompute distances only for pairs involving changed points
5. Update the top-k set with new candidates

This approach is not guaranteed to be faster than recomputation in all cases, but provides efficiency gains when changes are localized.

8 Experimental Results

We conducted experiments on an Apple M4 Max processor. All timing measurements represent single executions due to time constraints.

8.1 Performance Comparison

Table 1 summarizes execution times for the brute-force and optimized algorithms across different dataset sizes.

Table 1: Performance Summary: Brute-Force vs. Optimized Algorithm

Dimension	Drone Count	Brute-Force (s)	Optimized (s)	Speedup
2D	1,000	0.64	0.045	14.3×
2D	10,000	67.20	0.534	125.9×
2D	25,000	437.67	1.337	327.5×
2D	100,000	N/A (too slow)	6.323	–
3D	1,000	0.61	0.085	7.1×
3D	10,000	68.96	1.107	62.3×
3D	25,000	434.82	3.081	141.1×
3D	100,000	N/A (too slow)	17.521	–

8.2 Scalability Analysis

Figure 1 illustrates the execution time trends on a log-log scale. The brute-force algorithm exhibits the expected $O(n^2)$ growth (slope ≈ 2), while the optimized algorithm demonstrates $O(n \log n)$ scaling.

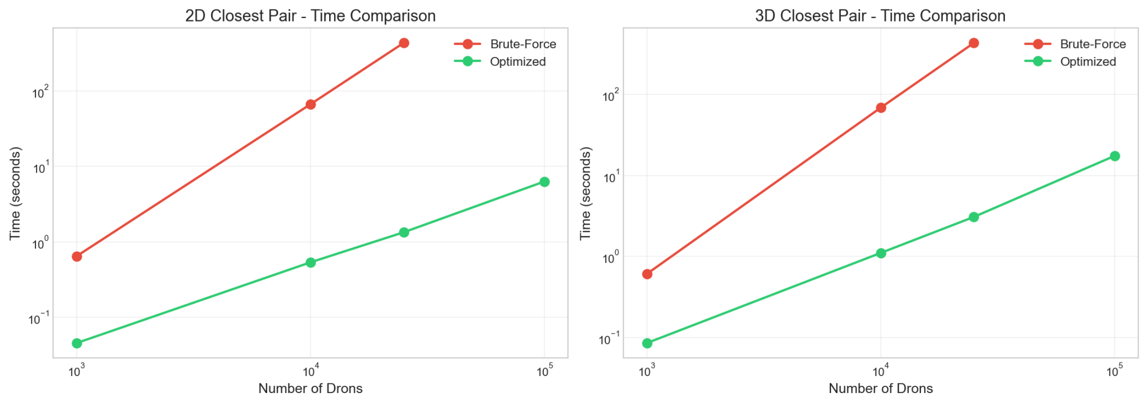

Figure 1: Execution time comparison between brute-force and optimized algorithms.

8.3 Speedup Analysis

Figure 2 shows the speedup factor achieved by the optimized algorithm. The speedup increases with dataset size, reaching over 300× for 25,000 drones in 2D, consistent with the theoretical complexity gap between $O(n^2)$ and $O(n \log n)$.

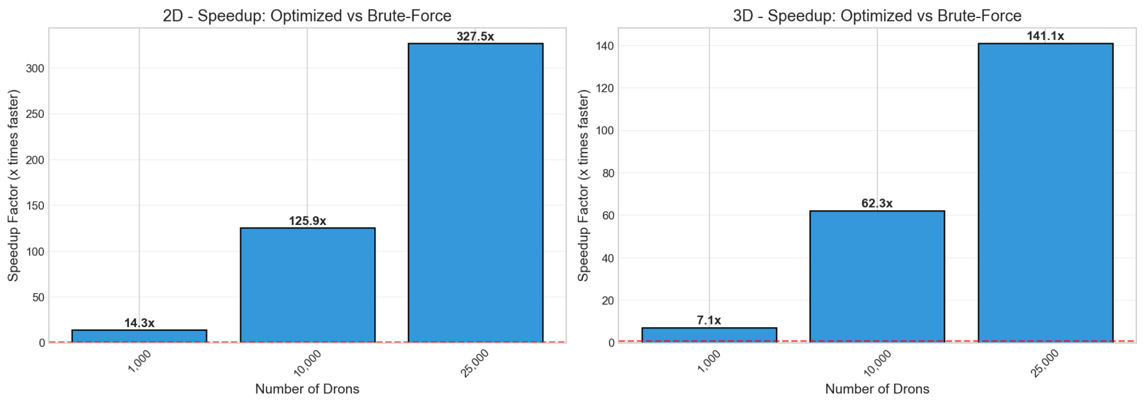

Figure 2: Speedup factor of optimized algorithm relative to brute-force.

8.4 Effect of Parameter k

Figure 3 demonstrates that execution time remains relatively constant across different values of k for fixed n. This confirms that the dominant factor is the number of points, not the number of pairs requested.

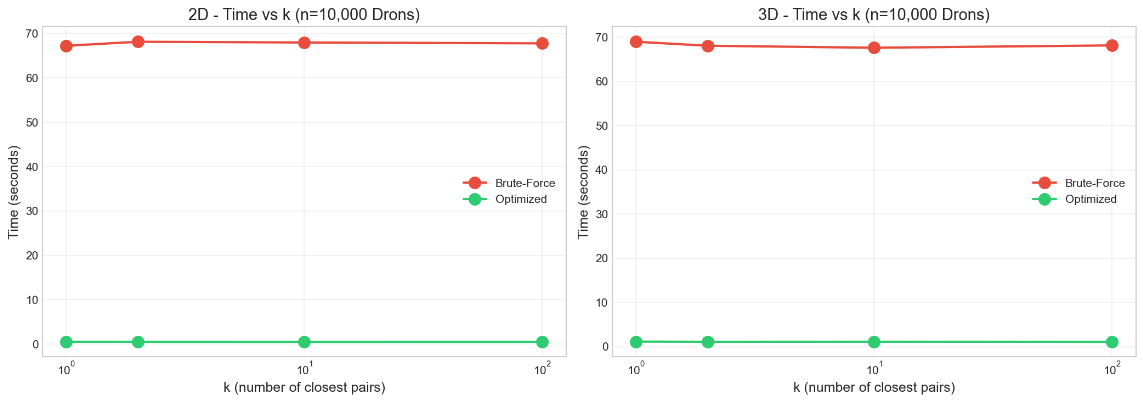

Figure 3: Execution time vs. k for n = 10,000 drones.

8.5 Memory Usage

Figure 4 compares memory consumption. The optimized algorithm uses more memory at larger scales due to recursion overhead and auxiliary data structures.

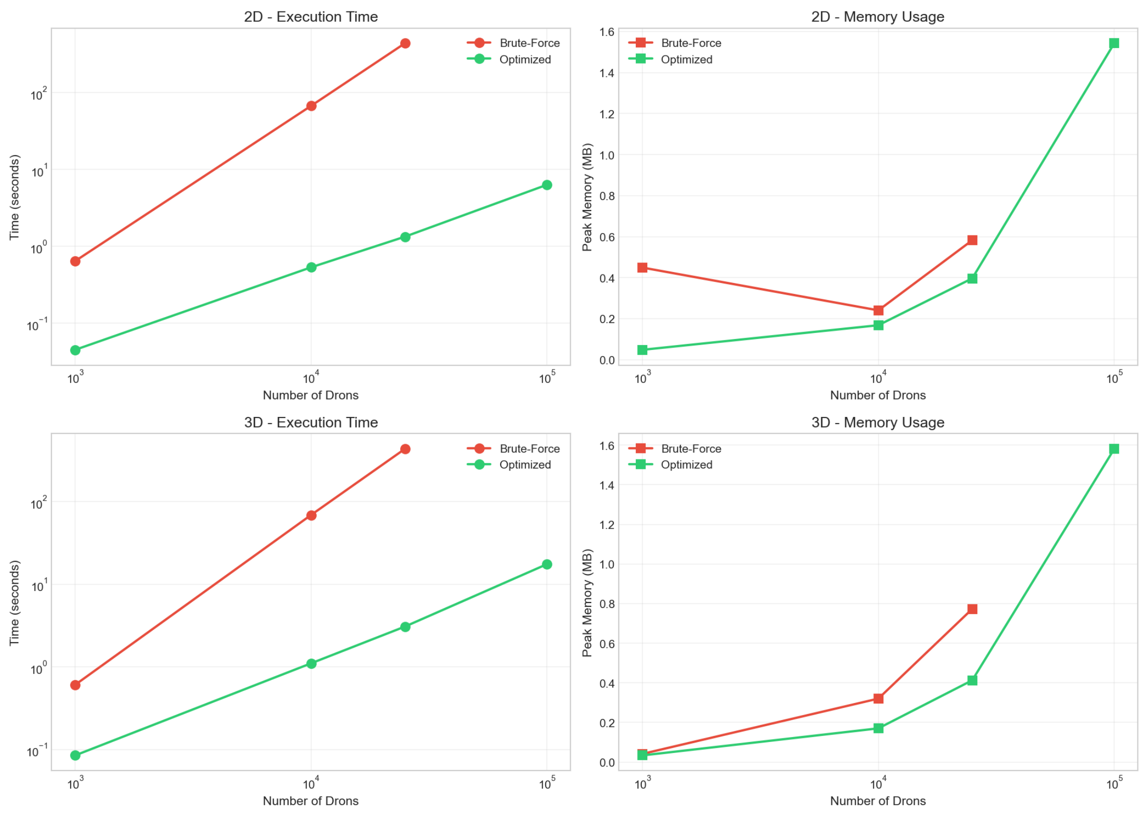

Figure 4: Execution time and memory usage comparison for 2D and 3D cases.

9 Complexity Summary

Table 2 summarizes the theoretical complexities of all algorithms presented.

Table 2: Complexity Summary

Algorithm	**Time Complexity**	**Space Complexity**
Brute-Force (Top-1)	$O(n^2)$	$O(n)$
Naïve Brute-Force (Top-k)	$O(n^2 \log n)$	$O(n^2)$
Optimized Brute-Force (Top-k)	$O(n^2 \log k)$	$O(n + k)$
Divide-and-Conquer (Top-k)	$O(n \log n)$	$O(n)$

10 Limitations

- All experiments were conducted on a single hardware platform (Apple M4 Max).
- Each experiment was executed only once; multiple runs would provide more robust statistical estimates.
- The brute-force algorithm could not complete for $n \geq 100{,}000$ within reasonable time.
- Memory measurements may not capture all allocation overhead.

11 Conclusion

We have presented and analyzed multiple algorithms for the top-k closest pair problem in the context of drone collision prevention. The divide-and-conquer approach achieves significant performance improvements over brute-force methods, with speedups exceeding 300× for larger datasets. Our experimental results confirm the theoretical complexity analysis, demonstrating $O(n \log n)$ scaling for the optimized algorithm compared to $O(n^2)$ for brute-force approaches.

Declaration of Original Work

I hereby declare that the work presented in the submitted report and the accompanying code is entirely my own. No portion of this submission has been copied, reproduced, or directly generated/refined using the responses or outputs of any AI tools (including, but not limited to, ChatGPT, Copilot, Gemini, DeepSeek, or other automated systems) unless stated otherwise. Any external sources, datasets, or tools that have been used are properly cited and referenced. I understand that any breach of this declaration may result in submission cancellation or significant mark deduction.

Disclosure of AI Tool Usage

First of all, I respect the policy and tried my best to follow it. But due to lack of time, I had to use LLMs for secondary tasks. For example, I wrote this report in markdown file ('files/report.md'). But due to poor writing skills and lack of experience in latex, I asked LLM to do it for me. I might not do it if there is better support for images in markdown.

The core algorithmic work, implementation, and analysis remain entirely my own. Relevant chat logs are available at:

- `https://t3.chat/share/dwperx2zdu`
- `https://t3.chat/share/iavuzw5qh3`
- `https://t3.chat/share/9qe7dd4mhe`

I take full responsibility and ready for punishment. I am ready to do my best to not do the mistake in the future assignments.

References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi: 10.1038/s41586-020-2649-2.

Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010. doi: 10.25080/Majora-92bf1922-00a.

Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985. doi: 10.1007/978-1-4612-1098-6.

Python Software Foundation. tracemalloc — trace memory allocations. Python Documentation, 2024. URL `https://docs.python.org/3/library/tracemalloc.html`.

Michael Ian Shamos and Dan Hoey. Closest-point problems. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 151–162, 1975. doi: 10.1109/SFCS.1975.8.