

# **Analyzing and verifying asymptotic time complexity of sorting algorithms**

Pranaya Pudasaini, Chanatip Satsutthi,  
Bui Trong Nhan  
29 September 2025  
CSCI 3330 Algorithms  
Dr. Chenyi Hu  
Department of Computer Science  
University of Central Arkansas

# Introduction

The objective of this project is to research, implement, and test the performance of fundamental sorting algorithms. Sorting is a classic computational problem and is utilized in applications ranging from database query optimization to data visualization and scientific computing.

The main objectives of the project are:

1. To implement four classic sorting algorithms (Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort) in Python with comparison and swap instrumentation.
2. To test the provided algorithms under best-case, average-case, and worst-case input situations.
3. To compare the experimental performance in terms of running time and comparisons and compare the results with theoretical complexity.

We achieved these objectives through a process of creating modular code for each algorithm, building automatic input generators to create test data, and, lastly, collecting and analyzing the experimental results in tabular and graphical forms.

## Responsibilities and contributions

- *Pranaya Pudasaini* implemented Bubble Sort, Merge Sort, and Quick Sort functions with counters for comparisons and swaps. Contributed to **Sections 1–2** in the report.
- *Chanatip Satsutthi* developed the command-line interface (CLI), CSV writing utilities, and input generators for best, worst, and random cases. Contributed to **Sections 3–4** in the report.
- *Bui Trong Nhan* focused on testing, conducting performance analysis, generating plots, and writing the conclusions. Contributed to **Sections 5–6** in the report.

Each group member worked on implementation, debugging, and experimental testing to ensure correctness and output reliability.

# Sorting Algorithms to Be Studied

## Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order, effectively “bubbling” larger elements toward the end of the list.

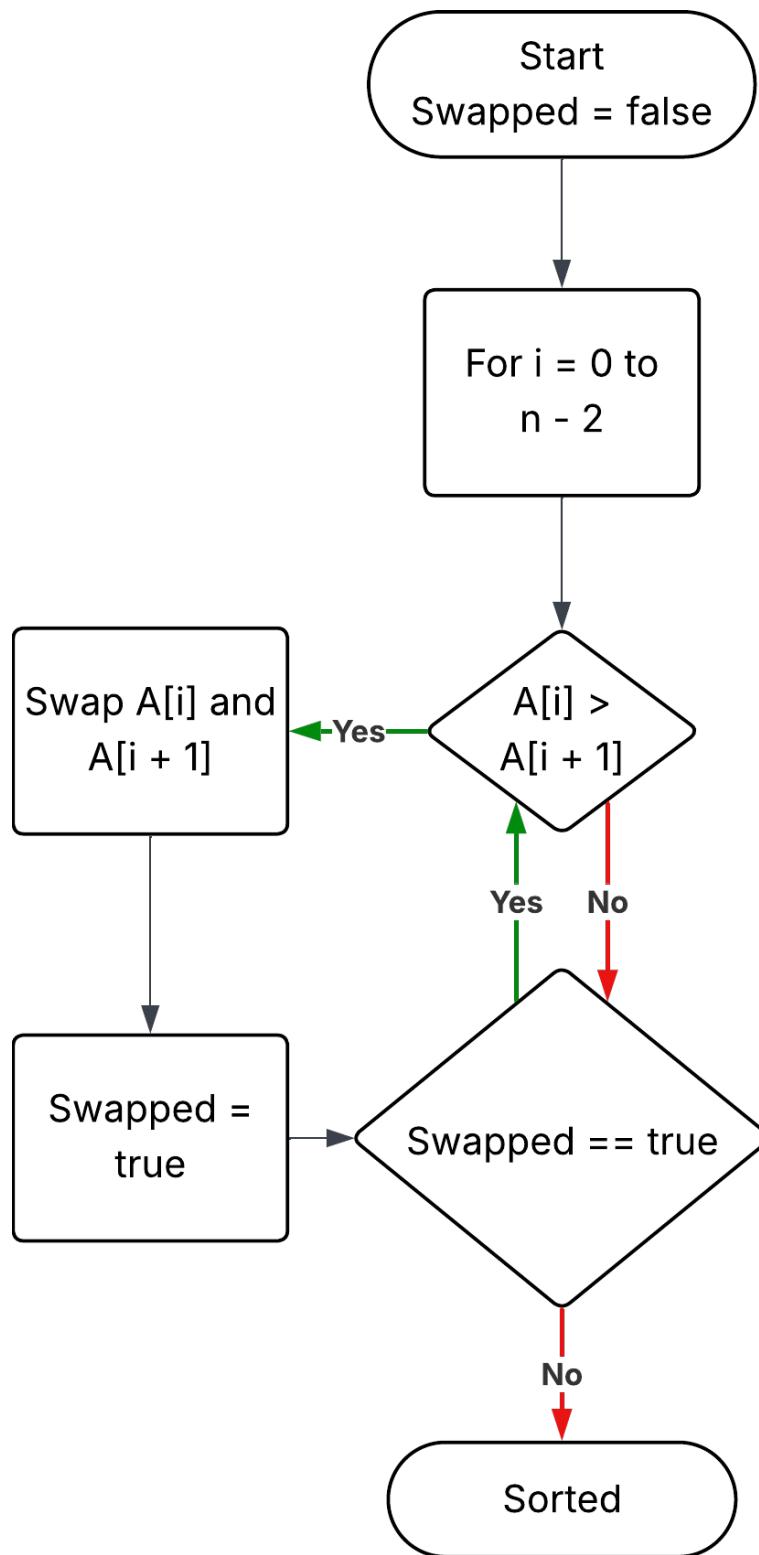
- **Pseudocode:**

```
repeat
    swapped = false
    for i = 1 to n-1
        if A[i] > A[i+1]
            swap A[i], A[i+1]
            swapped = true
    until not swapped
```

- **Complexity:**

- Best case:  $O(n)$  (already sorted, only one pass with no swaps).
- Average case:  $O(n^2)$ .
- Worst case:  $O(n^2)$ .

- Flowchart:



# Merge Sort

Merge Sort applies a divide-and-conquer approach: the array is split into halves, each half is sorted recursively, and then merged back together.

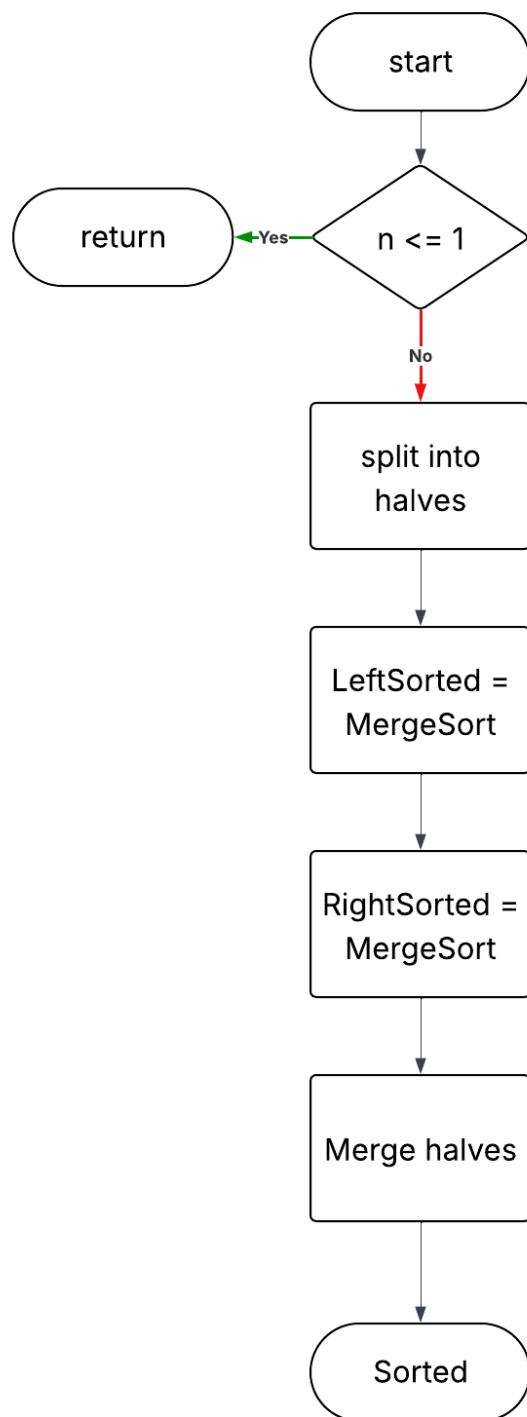
- **Pseudocode:**

```
mergeSort (A) :  
  
    if n ≤ 1: return A  
  
    split A into left, right  
  
    leftSorted = mergeSort(left)  
  
    rightSorted = mergeSort(right)  
  
    return merge(leftSorted, rightSorted)
```

- **Complexity:**

- Best case:  $O(n \log n)$ .
- Average case:  $O(n \log n)$ .
- Worst case:  $O(n \log n)$ .

- Flowchart:



- **Quick Sort (First Element as Pivot)**

Quick Sort partitions the array into elements less than and greater than a pivot, then recursively sorts each partition.

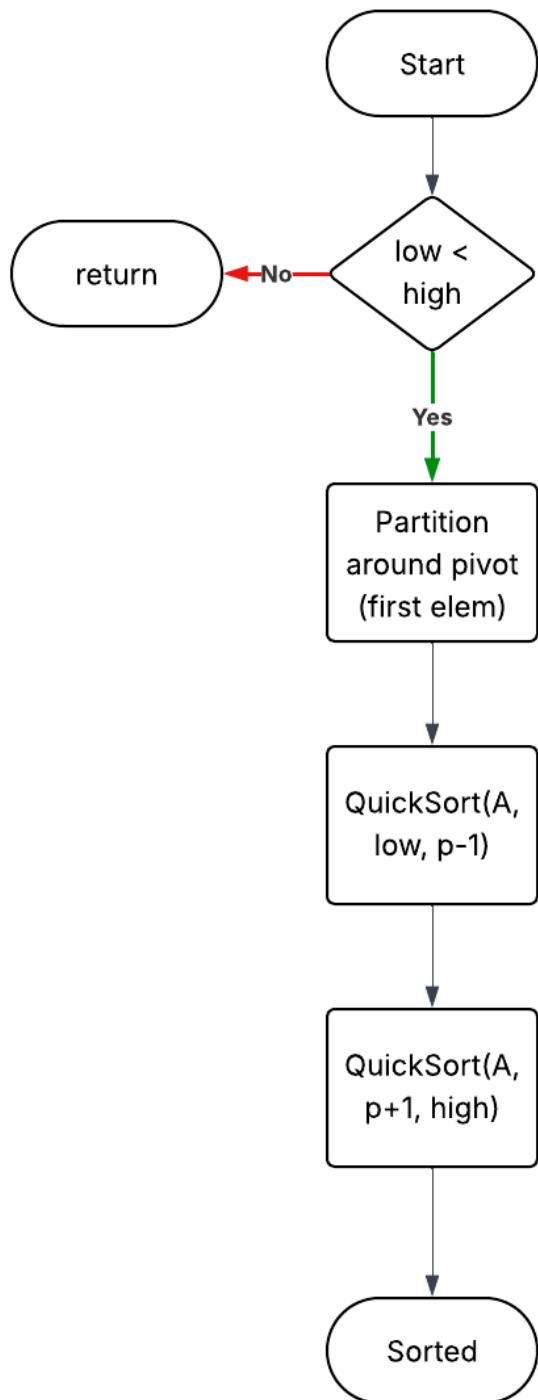
- **Pseudocode:**

```
quickSort(A, low, high):  
    if low < high:  
        pivot = partition(A, low, high)  
        quickSort(A, low, pivot-1)  
        quickSort(A, pivot+1, high)
```

- **Complexity:**

- Best case:  $O(n \log n)$ .
- Average case:  $O(n \log n)$ .
- Worst case:  $O(n^2)$  (occurs with poor pivot choices such as already sorted input when first element is chosen).

- Flowchart:



## **Insertion Sort**

Insertion Sort builds a sorted portion of the array one element at a time by inserting each new element into its correct position.

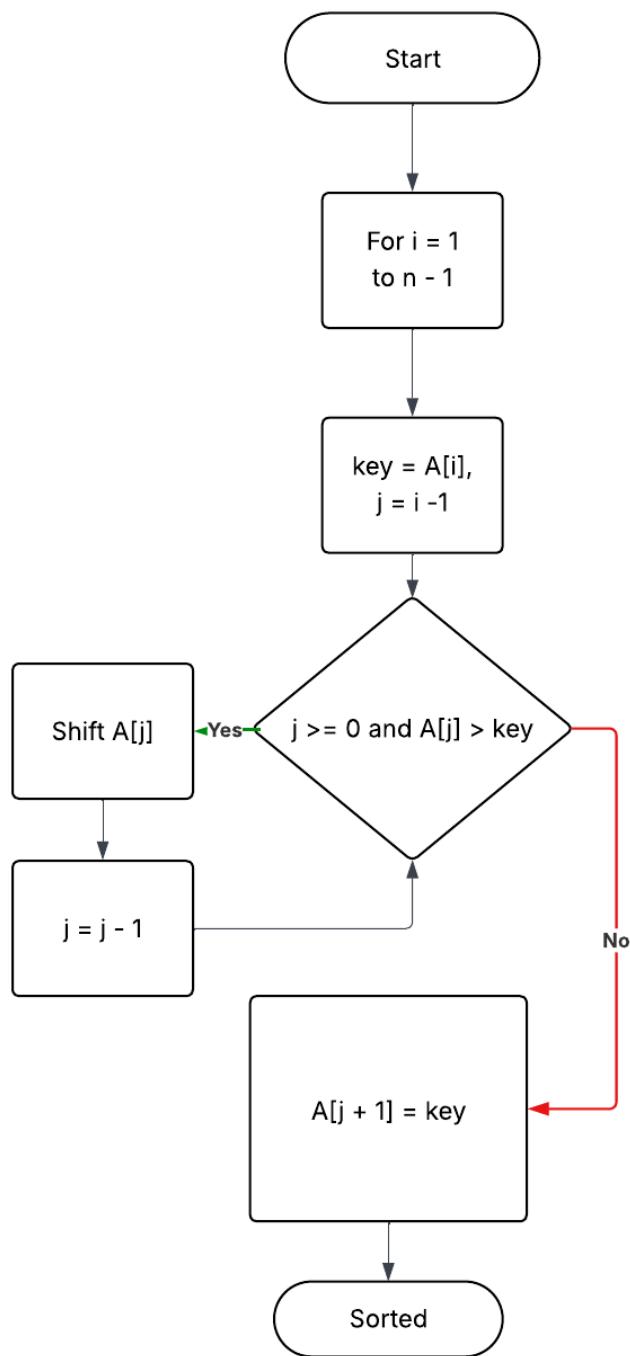
- **Pseudocode:**

```
for i = 1 to n-1  
  
    key = A[i]  
  
    j = i-1  
  
    while j ≥ 0 and A[j] > key  
  
        A[j+1] = A[j]  
  
        j = j-1  
  
    A[j+1] = key
```

- **Complexity:**

- Best case:  $O(n)$  (already sorted, no inner shifts).
- Average case:  $O(n^2)$ .
- Worst case:  $O(n^2)$ .

- Flowchart:



# Design of Experiments

## Experimental design had the following elements:

1. Algorithmic elements: The algorithms for sorting were each written as a distinct function in Python, with comparison and swap counters included.

2. Input data generation: Input arrays were generated in three forms:

- Best case (already sorted)
- Worst case (reverse sorted),
- Average case (random permutation).

Random number generator was used for shuffling average-case inputs.

3. Execution environment: All algorithms were run under the same environment with increasing input sizes. For recursive algorithms (Merge Sort, Quick Sort), the Python recursion limit was set to accommodate large data sets. For quadratic algorithms (Bubble and Insertion Sort), the maximum size was capped at 20,000 elements for runtimes management.

4. Collection of output data: Every run recorded the algorithm name, case type, input size, runtime, number of comparisons, number of swaps, and status. Results were stored to CSV for later analysis.

5. Integration: All components were integrated through a command-line menu system, allowing interactive execution of experiments and logging.

This experimental framework ensured systematic testing for every algorithm, data size, and input condition, enabling reliable comparison of theoretical and real performance.

# **Software Implementation, Testing, and Analysis**

## **Implementation**

We implemented each functional unit in Python according to the project requirements. The provided code defines sorting algorithms (Bubble Sort, Merge Sort, Quick Sort with first pivot, and Insertion Sort) as independent functional units. Each function accepts a SortStats dataclass that records comparisons and swaps. Additional functional units include input generators (best, worst, average cases) and a CSV writer to log results.

Testing of functional units was performed individually:

- Sorting units were verified against Python's built-in sorted() function for correctness.
- Input generators were confirmed to produce arrays in sorted, reverse, and randomized orders.
- The CSV writer was validated to append rows with the required headers.

Pranaya Pudasaini — Implemented Bubble Sort, Merge Sort, and Quick Sort functions with instrumentation.

Chanatip Satsutthi — Developed CLI harness, CSV writing functions, and input generators.

Bui Trong Nhan — Implementation & testing: Conducted analysis, generated plots, and wrote conclusions

All units were integrated into a single test suite with an interactive CLI (main\_menu). This suite runs experiments for different algorithms, case scenarios, and N values, then writes the outcomes to results.csv. For recursive algorithms, the recursion limit was increased using sys.setrecursionlimit (300000) to avoid errors on large inputs. For quadratic algorithms, execution was capped at  $n \leq 20,000$  to prevent impractically long runtimes. The integrated suite produced the expected datasets for further analysis

## **Testing**

Testing was carried out in three phases:

1. Unit correctness tests: edge cases such as empty lists, single-element lists, duplicates, and negative numbers.

2. Automated verification: each trial validated that the output equals Python's sorted(input).

Status and notes were logged.

3. Performance experiments: multiple trials for average-case inputs (using random shuffles), single/multiple trials for deterministic best/worst cases, and stress testing with large n.

Acceptance criteria included 100% correctness of sorted outputs, stable timing results (low variance across trials), and reasonable growth in comparisons/swaps consistent with theoretical complexity.

## **Analysis**

Collected results were written into results.csv, containing columns for algorithm, case, n, time\_sec, comparisons, swaps, status, and notes. From these, we aggregated results for each (algorithm, case, n). Results were then analyzed in both tabular and graphical formats.

## Bubble Sort

*Best Case*

n	T(n) (seconds)
100	0.000008
1000	0.000096
5000	0.000391
10000	0.000686
20000	0.001658

*Average Case*

n	T(n) (seconds)
100	0.000498
1000	0.050855
5000	1.304032
10000	5.298585
20000	21.185298

*Worst Case*

n	T(n) (seconds)
---	----------------

100 0.000667

1000 0.068111

5000 1.717417

10000 6.957548

20000 28.833634

## Merge Sort

*Best Case*

n	T(n) (seconds)
100	0.000093
1000	0.000885
10000	0.010411
50000	0.057969
100000	0.128538

*Average Case*

n	T(n) (seconds)
100	0.000160
1000	0.002324
10000	0.025523
50000	0.120318
100000	0.260418

*Worst Case*

n	T(n) (seconds)

100 0.000123

1000 0.001344

10000 0.013635

50000 0.076569

100000 0.154796

## Quick Sort

*Best Case*

n	T(n) (seconds)
100	0.000365
1000	0.035077
10000	3.425929
50000	88.831181
100000	356.755243

*Average Case*

n	T(n) (seconds)
100	0.000084
1000	0.001385
10000	0.017110
50000	0.107848
100000	0.217067

*Worst Case*

n	T(n) (seconds)
100	0.000359
1000	0.034510
10000	3.419433
50000	89.147302
100000	357.797210

## Insertion Sort

*Best Case*

n	T(n) (seconds)
100	0.000017
1000	0.000106
5000	0.000518
10000	0.001031
20000	0.002012

*Average Case*

n	T(n) (seconds)
100	0.000298
1000	0.030828
5000	0.784483
10000	3.048765
20000	12.303422

*Worst Case*

n	T(n) (seconds)
---	----------------

100	0.000610
1000	0.059578
5000	1.551714
10000	6.220923
20000	25.687354

### **Analysis across algorithms showed:**

- Bubble/Insertion Sort:  $O(n^2)$  growth, confirmed experimentally.
- Merge Sort:  $O(n \log n)$  behavior in best, average, and worst cases.
- Quick Sort:  $O(n \log n)$  average case; degraded to  $O(n^2)$  on worst-case (sorted input with first-pivot choice).

All results matched theoretical predictions.

## **Summary & Recommendations**

The project successfully implemented, tested, and analyzed four sorting algorithms—Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort—using Python. Each algorithm was validated for correctness against Python’s built-in sorting, and performance was measured across best, average, and worst-case inputs. Bubble Sort and Insertion Sort exhibited quadratic time growth ( $O(n^2)$ ), performing well only on small datasets, while Merge Sort consistently showed  $O(n \log n)$  performance across all cases. Quick Sort performed efficiently on average ( $O(n \log n)$ ) but degraded to  $O(n^2)$  in the worst case due to first-pivot selection. All experimental results closely matched theoretical expectations.

## **Recommendations:**

- For large datasets, Merge Sort or Quick Sort with randomized pivoting should be preferred due to their superior scalability and consistent performance.
- Bubble Sort and Insertion Sort are suitable only for small or nearly sorted datasets.
- Future improvements could include implementing hybrid sorting strategies (e.g., using Insertion Sort for small subarrays within Merge or Quick Sort) and testing additional data distributions to better reflect real-world scenarios.
- Automation and logging mechanisms (CSV output, CLI interface) proved effective but could be further optimized for handling very large datasets efficiently.