# Assignment 1 — Sorting Algorithms and Running Times - Group 10

## Introduction

This report presents the solutions to **Assignment 1**, which focuses on the analysis of sorting algorithms and divide-and-conquer recurrences. The assignment combines **empirical evaluation** (through implementation and experiments) with **theoretical analysis** (using asymptotic notation, the Master Theorem, and substitution method).

The work was carried out as a **group assignment** by:

- **Janani Raghu**
- **Dilhani Jayawickrama**
- **Parneet Kaur**

The implementations for Task 1 (step counting) and Task 2 (execution-time comparison in Python and C), along with experiment scripts, plots, and raw results, are maintained in a public GitHub repository to ensure reproducibility and conciseness of this report.

**GitHub Repository (source code and experiments):**
https://github.com/Jananir27/Group-10-Algo

The report itself focuses on methodology, results, and analysis, while detailed code listings are referenced externally.

# Task 1 — Counting the Steps

## Objective

The objective of this task is to implement four sorting algorithms, vary the input size (n), count the number of elementary steps, and verify that the observed growth matches the theoretical asymptotic running times.

The algorithms considered are:

- Insertion Sort
- Merge Sort
- Heap Sort
- Quick Sort

## Method

All algorithms were implemented in Python with explicit counters for:

- element comparisons

- array writes/swaps

The total number of steps is defined as the sum of comparisons and writes.

Worst-case inputs were used for algorithms whose running time depends on input order:

- **Insertion sort:** reverse-sorted input
- **Quick sort:** already sorted input with last-element pivot

Merge sort and heap sort were evaluated using random inputs, since their asymptotic complexity is independent of input ordering.

**Code reference (Task 1):**
The full Python implementations for step counting, input generation, and plotting are available in the GitHub repository:

https://github.com/Jananir27/Group-10-Algo/task1.ipynb

## Experimental Results

The figure below shows the total number of counted steps (comparisons + writes) as a function of the input size (n) for all four algorithms.

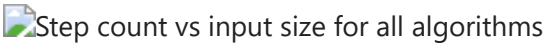


*Figure 1: Step count growth as a function of input size (n), confirming the theoretical asymptotic running times.*

## Conclusion

The measured step counts confirm the expected worst-case asymptotic behavior:

- Insertion sort and worst-case quicksort exhibit quadratic growth Θ((n^2)).
- Merge sort and heap sort exhibit (n log n) growth.

# Task 2 — Comparing True Execution Time

## Objective

The objective of this task is to compare the actual execution time of sorting algorithms implemented in two different programming languages.

## Method

Merge sort and quicksort were implemented in:

- **Python**
- **C** (compiled with `gcc -02`)

Execution time was measured for increasing input sizes (n). Each experiment was repeated multiple times, and the **minimum observed execution time** was recorded to reduce noise from system effects. Identical randomly generated inputs were used across languages to ensure a fair comparison.

**Code reference (Task 2):**
The Python and C implementations used for execution-time comparison, including benchmark scripts and raw timing outputs, are available in the GitHub repository:

- C benchmark implementation:
  https://github.com/Jananir27/Group-10-Algo/sort_benchmark.c
- Python benchmark implementation:
  https://github.com/Jananir27/Group-10-Algo/sort_benchmark.py
- Comparison between C and Python:
  https://github.comJananir27/Group-10-Algo/task2_plot_time_c_vs_python.ipynb

## Results

The measured execution times show that:

- The **C implementations consistently outperform Python** for all tested input sizes.
- Both languages exhibit the same asymptotic growth trends, matching theoretical expectations.
- Merge sort shows stable scaling, while quicksort performance is sensitive to pivot selection.

## Experimental Results

The figure below shows the execution time as a function of the input size (n) for Python and C implementations of the sorting algorithms.

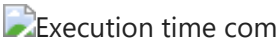
Execution time comparison between Python and C

*Figure 2: Execution time comparison of sorting algorithms implemented in Python and C for increasing input sizes.*

## Discussion

Although asymptotic analysis predicts similar growth rates, the constant factors differ significantly between languages. Python incurs additional overhead due to interpretation, dynamic typing, and memory management, whereas C executes compiled machine code with lower per-operation cost.

## Conclusion

This experiment demonstrates that asymptotic complexity describes long-term growth behavior, but real execution time strongly depends on implementation language and

constant factors.

# Task 3 — Basic Proofs

## (a) Show that $n^2 \log n \notin o(n^2)$

By definition, a function f(n) is in o(g(n)) if:

$\lim (n \to \infty) [ f(n) / g(n) ] = 0$

Let
$f(n) = n^2 \log n$
$g(n) = n^2$

Then:

$\lim (n \to \infty) [ (n^2 \log n) / n^2 ]$
$= \lim (n \to \infty) \log n$
$= \infty$

Since the limit is not equal to zero, it follows that:

**$n^2 \log n \notin o(n^2)$**

---

## (b) Show that $n^2 \notin o(n^2)$

Using the definition of little-o notation:

$\lim (n \to \infty) [ n^2 / n^2 ] = 1$

Since the limit is a non-zero constant, we conclude that:

**$n^2 \notin o(n^2)$**

# Task 4 — Divide and Conquer Analysis

## (a) Master Theorem: R1

Given recurrence: T(n) = 16 T(n/4) + n

Identify: a = 16
b = 4
f(n) = n

Compute: log_b(a) = log_4(16) = 2
So n^(log_b(a)) = n^2

Compare f(n) with n^(log_b(a)): f(n) = n = O(n^(2 - ε)) with ε = 1

This matches Master Theorem Case 1, therefore:

**T(n) = Θ(n^2)**

---

## (b) Substitution Method: T(n) = 4T(n/2) + n

We determine the runtime and check which inductive hypotheses hold.

### Step 1: Expected runtime (intuition)

Here: a = 4, b = 2, f(n) = n
n^(log_2(4)) = n^2
Since f(n) grows slower than n^2, the solution is:

**T(n) = Θ(n^2)**

Now we verify using substitution-style inequalities.

---

### Hypothesis 1: T(n) ≤ c n^2 (c > 0)

Assume: T(n/2) ≤ c (n/2)^2 = c n^2 / 4

Then: T(n) = 4T(n/2) + n
≤ 4 * (c n^2 / 4) + n
= c n^2 + n

This is NOT ≤ c n^2 because of the extra +n term.

So:

**Hypothesis 1 does not hold** (as stated).

---

### Hypothesis 2: T(n) ≥ c n^2 (c > 0)

Assume: T(n/2) ≥ c (n/2)^2 = c n^2 / 4

Then: T(n) = 4T(n/2) + n
≥ 4 * (c n^2 / 4) + n
= c n^2 + n
≥ c n^2

So:

**Hypothesis 2 holds**.

---

### Hypothesis 3: T(n) ≤ (c n^2 − b n) (c > 0, b > 0)

Assume: T(n/2) ≤ c (n/2)^2 − b (n/2)
= c n^2 / 4 − b n / 2

Then: T(n) = 4T(n/2) + n
≤ 4 * (c n^2 / 4 − b n / 2) + n
= c n^2 − 2 b n + n
= c n^2 − (2b − 1) n

To make this ≤ c n^2 − b n, it is enough that: (2b − 1) ≥ b ⇔ b ≥ 1

So:

**Hypothesis 3 holds for b ≥ 1** (and for sufficiently large n with a valid base case).

---

## Conclusion for (b)

The recurrence grows as:

**T(n) = Θ(n^2)**

Hypotheses:

- (1) does NOT hold
- (2) holds
- (3) holds (for b ≥ 1)

---

# Bonus Task — T(n) = 2T(n/2) + n log n

Identify: a = 2
b = 2
f(n) = n log n

Compute: log_b(a) = log_2(2) = 1
So n^(log_b(a)) = n

Now compare: f(n) = n log n = Θ(n * log^1(n))

This matches the “complete” Master Theorem Case 2: If f(n) = Θ(n^(log_b(a)) * log^k(n)) then T(n) = Θ(n^(log_b(a)) * log^(k+1)(n))

Here k = 1, therefore:

**T(n) = Θ(n log^2 n)**