

**Pune Institute of Computer Technology,  
Dhankawadi, Pune**

**Mini Project Report on**

# **Merge Sort**

Submitted by

**Sushilkumar Dhamane**

Roll No: 41123

BE-1

**Vinayak Jamadar**

Roll No: 41137

BE-1

**Sanket Jhavar**

Roll No: 41138

BE-1

Under the guidance of

**Prof. S. W. Jadhav**



**Department of Computer Engineering**

Academic Year 2023-24

# Contents

<b>1</b>	<b>Title: Merge Sort</b>	<b>2</b>
<b>2</b>	<b>Problem Statement</b>	<b>2</b>
<b>3</b>	<b>Abstract</b>	<b>2</b>
<b>4</b>	<b>Learning Objectives:</b>	<b>2</b>
<b>5</b>	<b>Learning Outcomes:</b>	<b>2</b>
<b>6</b>	<b>Software and Hardware Requirements:</b>	<b>2</b>
<b>7</b>	<b>Concept-Related Theory:</b>	<b>3</b>
7.1	Merge Sort Algorithm: . . . . .	3
7.2	Single-Threaded Merge Sort: . . . . .	3
7.3	Multithreaded Merge Sort: . . . . .	4
<b>8</b>	<b>Algorithm</b>	<b>4</b>
8.1	Merge Sort Algorithm: . . . . .	4
8.2	Multithreaded Merge Sort Algorithm: . . . . .	4
<b>9</b>	<b>Output</b>	<b>5</b>
<b>10</b>	<b>Conclusion</b>	<b>7</b>

# 1 Title: Merge Sort

## 2 Problem Statement

Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.

## 3 Abstract

This document explores the concepts and implementations of the merge sort algorithm and its multithreaded counterpart. It outlines the learning objectives, outcomes, and the software and hardware requirements for the exercise. The concept-related theory provides an in-depth understanding of merge sort, including a step-by-step example and its time complexity analysis. The document also delves into the differentiation between single-threaded and multithreaded merge sort, highlighting their key characteristics and use cases. By the end of this exploration, readers will have a comprehensive grasp of the merge sort algorithm, its multithreaded variant, and the factors that differentiate these sorting methods. This knowledge will serve as a foundation for making informed decisions when choosing sorting algorithms for specific applications and hardware configurations.

## 4 Learning Objectives:

- Understand the fundamentals of the merge sort algorithm and its multithreaded implementation.
- Learn how to analyze the time complexity of sorting algorithms in best and worst-case scenarios.
- Gain hands-on experience with both the traditional and multithreaded merge sort algorithms.
- Compare the performance of the two algorithms to determine the advantages and limitations of multithreading in sorting.

## 5 Learning Outcomes:

- Implement the merge sort algorithm in Python.
- Create a multithreaded version of merge sort in Python.
- Analyze and discuss the time complexity of merge sort in best and worst-case scenarios.
- Measure and compare the execution times of the traditional and multithreaded merge sort for various input sizes.
- Understand the impact of multithreading on algorithm performance.

## 6 Software and Hardware Requirements:

- **Software Requirements:**
  - Python (3.x recommended)
  - A text editor or integrated development environment (IDE)
  - Knowledge of basic Python programming
- **Hardware Requirements:**
  - A computer with a multicore processor (for testing multithreaded performance)
  - Sufficient memory for handling the input data

## 7 Concept-Related Theory:

### 7.1 Merge Sort Algorithm:

Merge sort is a classic sorting algorithm that falls into the category of divide-and-conquer algorithms. It works as follows:

**Divide:** The unsorted list is divided into  $n$  sublists, each containing one element. In the context of computer science, this initial division is achieved using recursive calls.

**Conquer:** The sublists are repeatedly merged to produce new sorted sublists. This process continues until there is only one sublist remaining, which is the sorted list. Merging is a fundamental step in merge sort, and it is the reason behind its name.

**Merge Operation:** Merging is the process of taking two sorted sublists and combining them to create a single sorted list. The merging step involves comparing elements from both sublists and placing them in the correct order. This is done by repeatedly selecting the smallest (or largest) element from the two sublists and moving it to the new list until all elements are merged.

**Example of Merge Sort:** Let's consider an example to illustrate how merge sort works. We'll use a simple list of numbers for demonstration:

**Original List:** [38, 27, 43, 3, 9, 82, 10]

**Divide Phase:**

- Divide the list into two halves: [38, 27, 43] and [3, 9, 82, 10]

**Conquer Phase:**

- Continue dividing the sublists until they have only one element. In this case, we further divide [38, 27, 43] into [38], [27], and [43], and [3, 9, 82, 10] into [3], [9], [82], and [10].

**Merge Operation:**

- Merge the divided sublists back together in a sorted manner. Start by merging [38] and [27] into [27, 38], [43] remains unchanged. For the second sublist, merge [3] and [9] into [3, 9], and [82] and [10] into [10, 82].
- Continue merging until you have a fully sorted list. Merge [27, 38] and [43] to get [27, 38, 43], and merge [3, 9] and [10, 82] to get [3, 9, 10, 82].
- Finally, merge the two fully sorted sublists [27, 38, 43] and [3, 9, 10, 82] to obtain the sorted list: [3, 9, 10, 27, 38, 43, 82].

Merge sort guarantees that the result will be a sorted list, and it achieves this in a stable manner, meaning that the relative order of equal elements remains unchanged.

**Time Complexity of Merge Sort:** The time complexity of merge sort is  $O(n \log n)$  in both best and worst-case scenarios. It offers consistent and predictable performance, making it a reliable choice for sorting large datasets. Its divide-and-conquer nature and the merging step ensure that all elements are correctly sorted.

### 7.2 Single-Threaded Merge Sort:

Single-threaded merge sort is the traditional implementation of the merge sort algorithm, where sorting and merging operations are performed sequentially on a single thread or core. It is characterized by the following:

- **Sequential Execution:** In single-threaded merge sort, all operations, including dividing the list and merging sublists, are carried out sequentially by a single thread. Each step is executed one after the other.
- **Performance:** While single-threaded merge sort is efficient for small to moderately-sized datasets, it may not fully exploit the processing power of modern multicore processors. It has a consistent time complexity of  $O(n \log n)$ , which is optimal for many sorting scenarios.
- **Simplicity:** Single-threaded merge sort is relatively straightforward to implement and understand, making it a good choice for educational purposes and simpler applications.

### 7.3 Multithreaded Merge Sort:

Multithreaded merge sort is an enhanced version of merge sort that leverages the power of parallel processing. It splits the list into sublists and sorts them concurrently using multiple threads. Here are the key characteristics:

- **Parallel Execution:** In multithreaded merge sort, the list is divided into sublists, and separate threads are used to sort and merge these sublists concurrently. This parallel execution allows for multiple tasks to be performed simultaneously.
- **Performance:** Multithreaded merge sort can provide significant performance improvements for large datasets, especially on machines with multiple cores. It takes advantage of parallelism to reduce the sorting time, potentially making it faster than single-threaded merge sort.
- **Complexity:** Implementing multithreaded merge sort requires handling thread creation, synchronization, and management, which adds complexity to the algorithm compared to the single-threaded version.

Aspect	Single-Threaded Merge Sort	Multithreaded Merge Sort
Execution Model	Sequential	Parallel
Performance	Efficient for small to moderate datasets	Improved performance for large datasets, especially on multi-core processors
Complexity	Simpler implementation	More complex due to thread management
Hardware Dependence	Less dependent on hardware, performs consistently	Performance depends on the number of available cores, more hardware-dependent
Use Cases	Suitable for smaller datasets, simplicity prioritized	Ideal for large datasets, performance optimization is critical

Table 1: Comparison of Merge Sort Variants

## 8 Algorithm

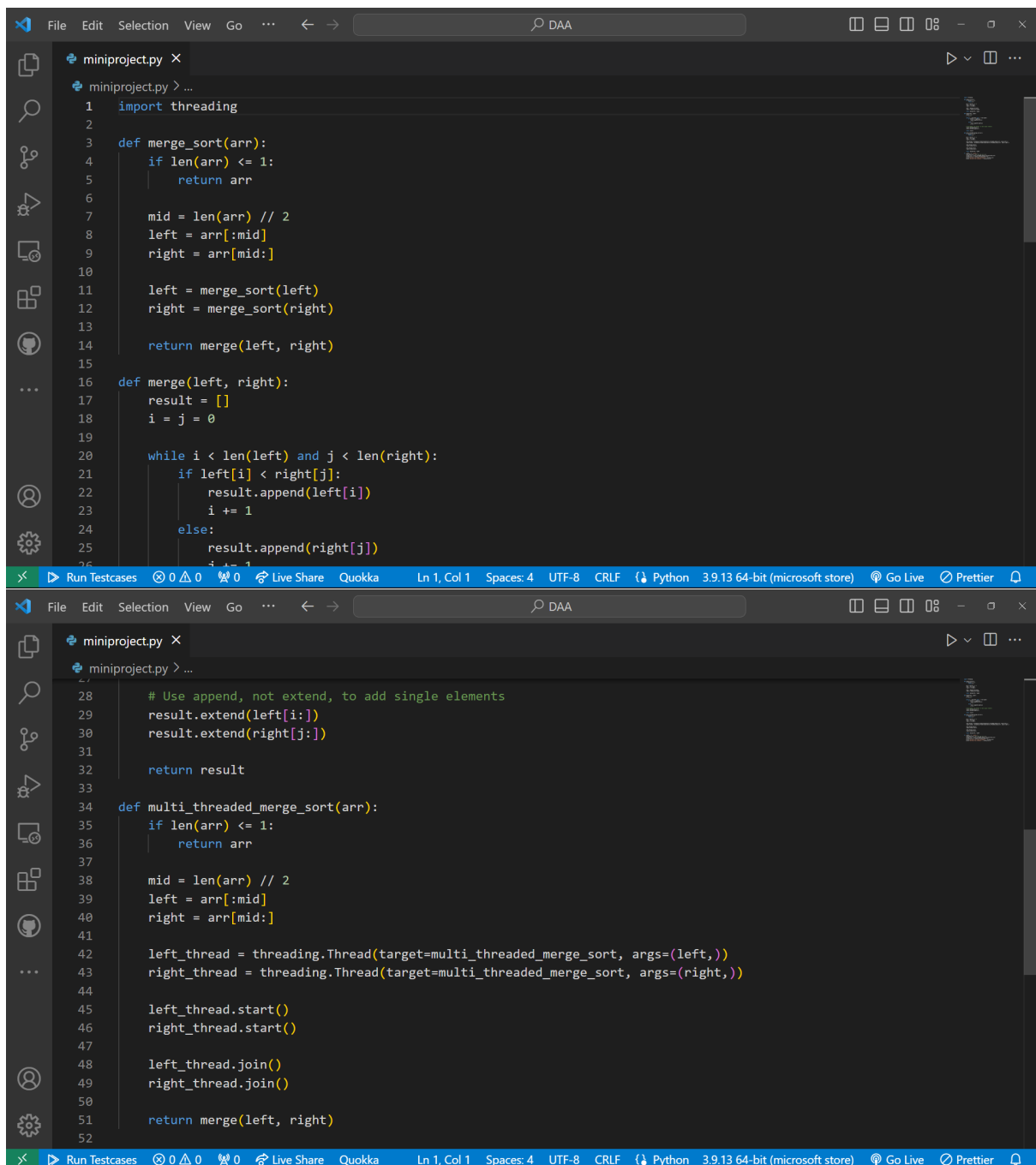
### 8.1 Merge Sort Algorithm:

- If the list contains one or zero elements, it is already sorted.
- Divide the list into two halves.
- Recursively sort each half.
- Merge the sorted halves back into a single sorted list.

### 8.2 Multithreaded Merge Sort Algorithm:

- If the list contains one or zero elements, it is already sorted.
- Divide the list into two halves.
- Create two threads, each responsible for sorting one half.
- Recursively sort each half using threads.
- Merge the sorted halves back into a single sorted list.

## 9 Output



```
miniproject.py x
miniproject.py > ...
1  import threading
2
3  def merge_sort(arr):
4      if len(arr) <= 1:
5          return arr
6
7      mid = len(arr) // 2
8      left = arr[:mid]
9      right = arr[mid:]
10
11     left = merge_sort(left)
12     right = merge_sort(right)
13
14     return merge(left, right)
15
16 def merge(left, right):
17     result = []
18     i = j = 0
19
20     while i < len(left) and j < len(right):
21         if left[i] < right[j]:
22             result.append(left[i])
23             i += 1
24         else:
25             result.append(right[j])
26             j += 1
27
28     # Use extend, not append, to add single elements
29     result.extend(left[i:])
30     result.extend(right[j:])
31
32     return result
33
34 def multi_threaded_merge_sort(arr):
35     if len(arr) <= 1:
36         return arr
37
38     mid = len(arr) // 2
39     left = arr[:mid]
40     right = arr[mid:]
41
42     left_thread = threading.Thread(target=multi_threaded_merge_sort, args=(left,))
43     right_thread = threading.Thread(target=multi_threaded_merge_sort, args=(right,))
44
45     left_thread.start()
46     right_thread.start()
47
48     left_thread.join()
49     right_thread.join()
50
51     return merge(left, right)
52
```

The image shows two screenshots of the Visual Studio Code editor. The top screenshot displays a Python file named `miniproject.py` with the following code:

```
52
53 if __name__ == "__main__":
54     unsorted_list = [12, 11, 13, 5, 6, 7]
55     sorted_list = multi_threaded_merge_sort(unsorted_list)
56     sorted_list_1 = merge_sort(unsorted_list)
57     print("Sorted array (Multithreaded):", sorted_list)
58     print("Sorted array (Normal):", sorted_list_1)
```

The bottom screenshot shows the same editor with the `TERMINAL` panel open. It displays the command `python -u "f:\Programming\DAA\miniproject.py"` and its output:

```
Sorted array (Multithreaded): [5, 6, 7, 12, 11, 13]
Sorted array (Normal): [5, 6, 7, 11, 12, 13]
```

The status bar at the bottom of both screenshots indicates the file is `miniproject.py`, the Python version is `3.9.13 64-bit (microsoft store)`, and the encoding is `UTF-8`.

## 10 Conclusion

In this exploration, we delved into the Merge Sort algorithm and its multithreaded variant, providing a comprehensive understanding of their inner workings. We began by unraveling the fundamental concepts of the traditional Merge Sort. This sorting algorithm, based on the divide-and-conquer principle, has a predictable time complexity, making it a reliable choice for sorting large datasets. We also illustrated the step-by-step execution of Merge Sort using a practical example, highlighting its stability and correctness. In the comparative analysis, we recognized the trade-offs between the two implementations, making it evident that their choice depends on specific use cases and hardware considerations. This exploration is a valuable foundation for making informed decisions when selecting sorting algorithms, whether for simplicity, efficiency, or scalability. In its various forms, Merge Sort remains a significant asset in sorting algorithms, with its stable performance and predictable time complexity.