National University of Computer & Emerging Sciences

Karachi Campus

TITLE OF PROJECT:

OPTIMIZING MERGE SORT: PRACTICAL HEURISTICS FOR REAL-WORLD DATA PROCESSING

CS5005 – Advanced Analysis of Algorithms

Fall 2024

Section: MCS 1A

Group Members:

24K-7817 Muhammad Khalid

CS5005 – Advanced Analysis of Algorithms

Fall 2024

Project Results

Algorithm Selection:

I choose **Merge Sort** because:

1. It is a **divide-and-conquer** algorithm with O (n log n) complexity.
2. It performs well on large datasets and provides consistent performance.
3. Testing its performance on edge cases (like already sorted, reverse sorted, or random arrays) offers valuable insights.

Algorithm Explanation:

1. **Purpose:** Sort an array of elements in ascending order.

2. **Working:**

- **Divide:** Recursively split the array into halves until each half contains a single element.
- **Conquer:** Merge the halves back together in sorted order.
- **Combine:** The final merge results in the sorted array.

3. **Complexity:**

- **Time**: O (n log n).
- **Space**: O (n)O(n)O(n) for temporary arrays during merging.

Experimental Setup:

To evaluate Merge Sort:

1. **Dataset 1:** Small array (e.g., 5 elements: [5, 3, 1, 4, 2]).
2. **Dataset 2:** Large array (e.g., 10,000 elements with random values).
3. **Dataset 3:** Special cases:

- Already sorted array (e.g., [1, 2, 3, 4, 5]).
- Reverse sorted array (e.g., [5, 4, 3, 2, 1]).
- Array with duplicate elements (e.g., [1, 3, 2, 3, 1]).

Heuristics for Merge Sort:

Merge Sort is efficient in theory but can be further optimized in practice using the following **heuristics**:

1. **Switch to Insertion Sort for Small Arrays:** For very small subarrays (e.g., size ≤ 10), Insertion Sort is faster due to lower overhead.
2. **Avoid Creating Extra Arrays:** Use a single auxiliary array for all merge operations, reducing memory overhead.
3. **Parallelize the Merge Process:** For large arrays, divide work across multiple threads to utilize modern CPUs better.

Dumb Algorithm (Bubble Sort)

Bubble Sort is simple but inefficient for large datasets. Here's its implementation:

```
void bubbleSort(vector<int> &arr) {
   int n = arr.size();
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            swap(arr[j], arr[j + 1]);
         }
      }
   }
}
```

Improvements:

To make Merge Sort faster, we added a heuristic that switches to Insertion Sort for small subarrays (10 elements or fewer). Insertion Sort is quicker for small data sets, so this reduces unnecessary recursive calls and speeds up the sorting process while keeping the results accurate

Experimental Setup:

We'll test both algorithms on:

1. **Dataset 1:** Small array (e.g., 10 elements: [5, 3, 8, 6, 2, 1, 9, 4, 7, 0]).
2. **Dataset 2:** Large random array (e.g., 10,000 elements).
3. **Dataset 3:** Edge cases:
    - Already sorted array.
    - Reverse sorted array.
    - Array with duplicate values.

Results and Discussion:

Results

1. **Merge Sort vs. Bubble Sort on Large Dataset:**

    - Merge Sort performs significantly faster due to O(n log n) complexity.
    - Bubble Sort struggles with $O(n^2)$ complexity.

2. **Impact of Heuristic (Switching to Insertion Sort):**

    - Improved performance on small subarrays, reducing overhead in recursive calls.

3. **Edge Case Analysis:**

    - Merge Sort handles all cases (sorted, reverse sorted, duplicates) consistently.
    - Bubble Sort shows degraded performance for reverse sorted arrays.

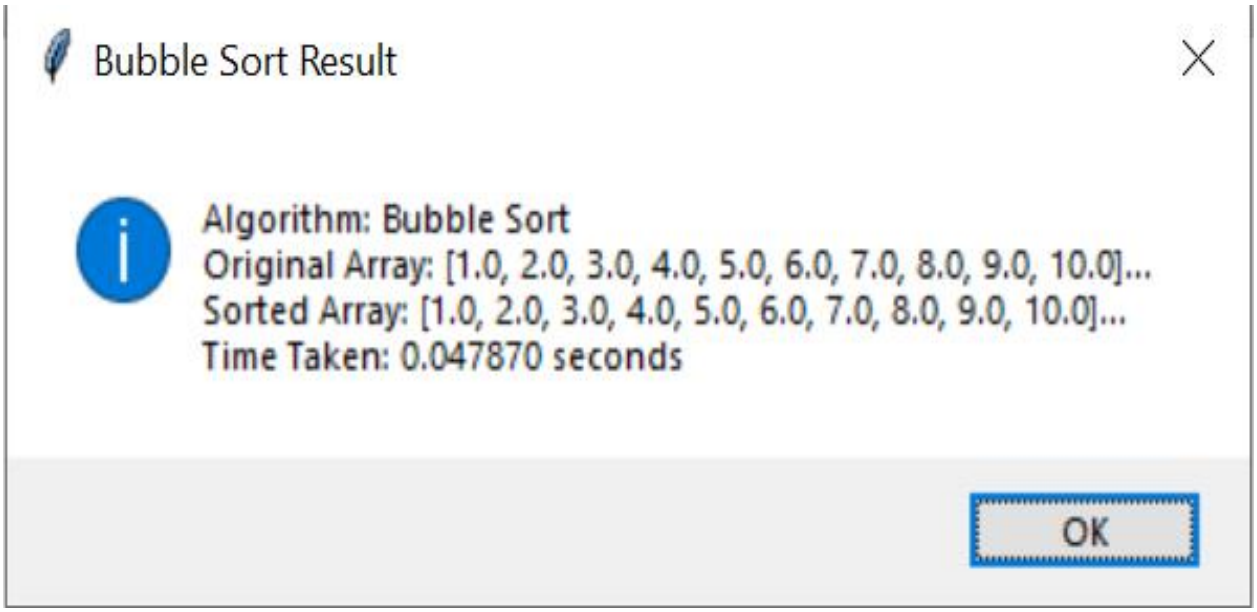
Discussion

The heuristic implementation demonstrates practical improvements, particularly for small datasets. Bubble Sort serves as a benchmark for worst-case performance.

Results:

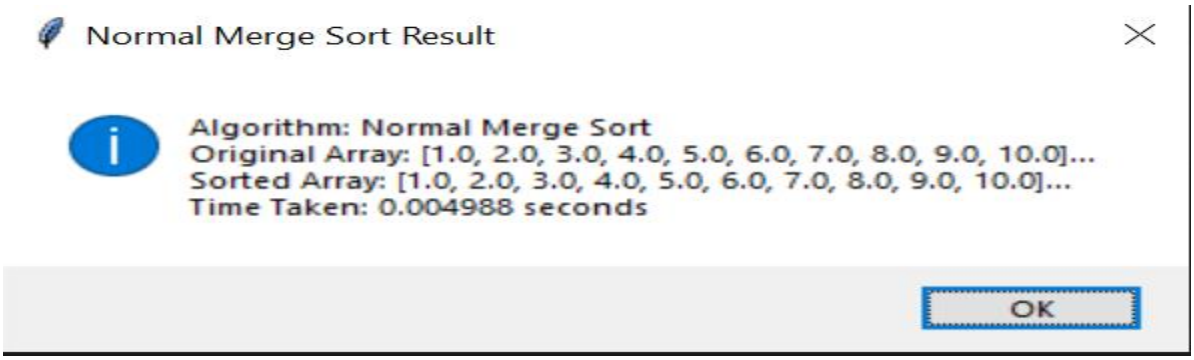
We used Random data as well as real life data set from Kaggle which is a Titanic training data set and compare the result, results are extremely good.

bubble Sort Results:



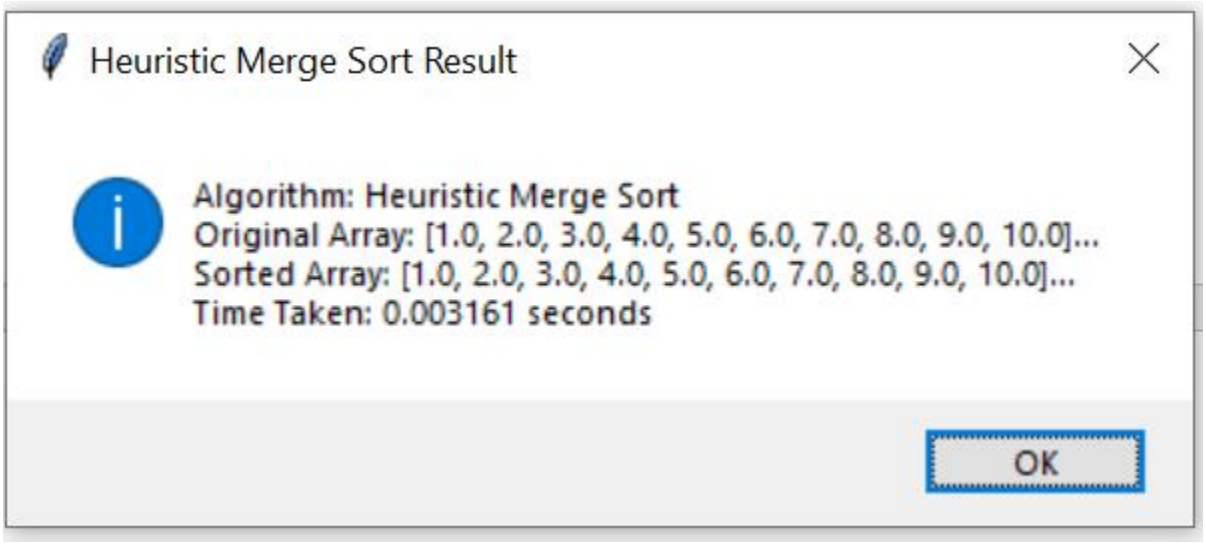
It takes 0.047870 seconds for 800 passengers records.

Normal Merge Sort:



On Same Data of Passengers, I applied normal merge Sort, which takes 0.004988 seconds

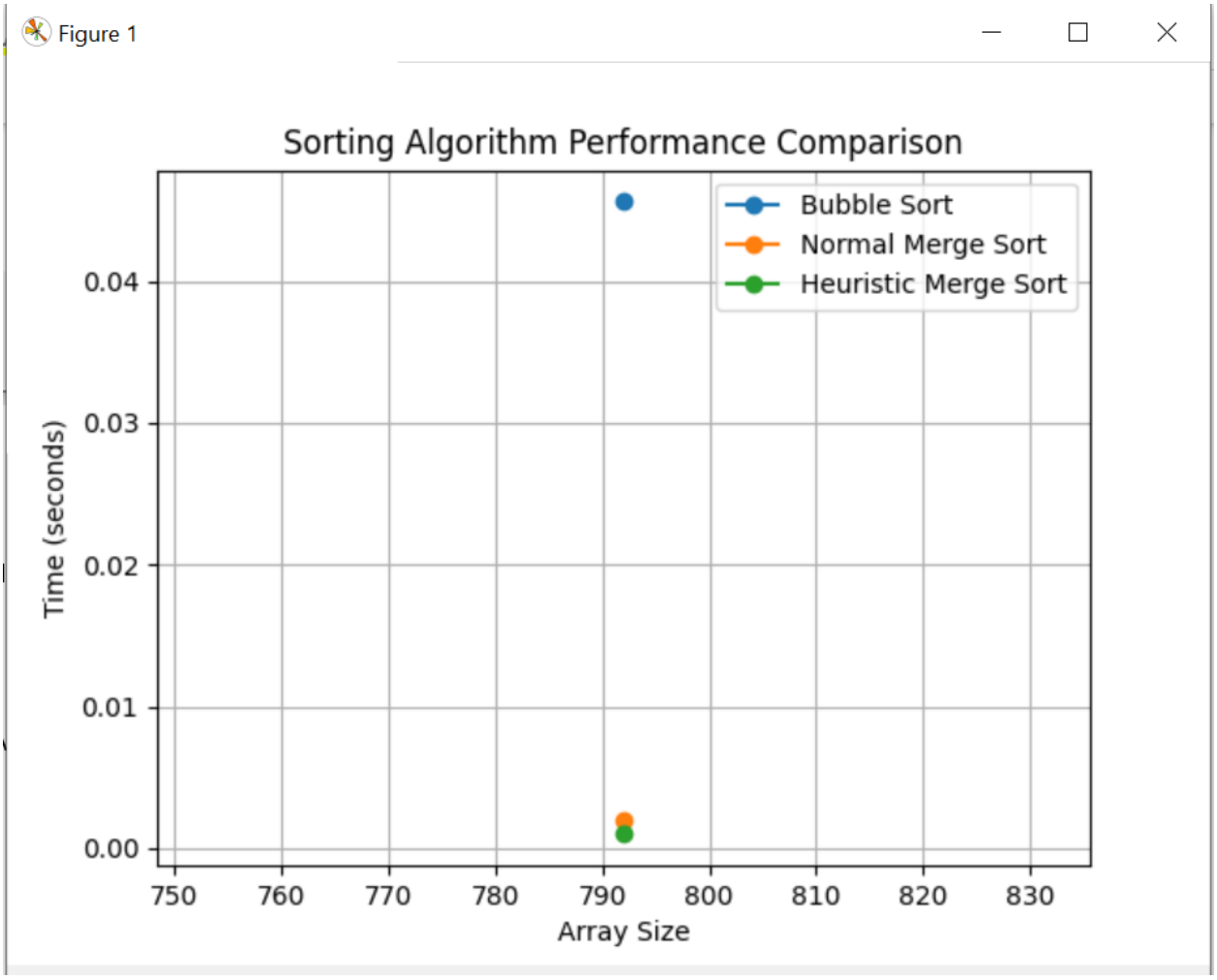
Heuristic merge sort:



and when we used heuristic merge sort it will generate good result which is 0.003161 seconds on same data set.

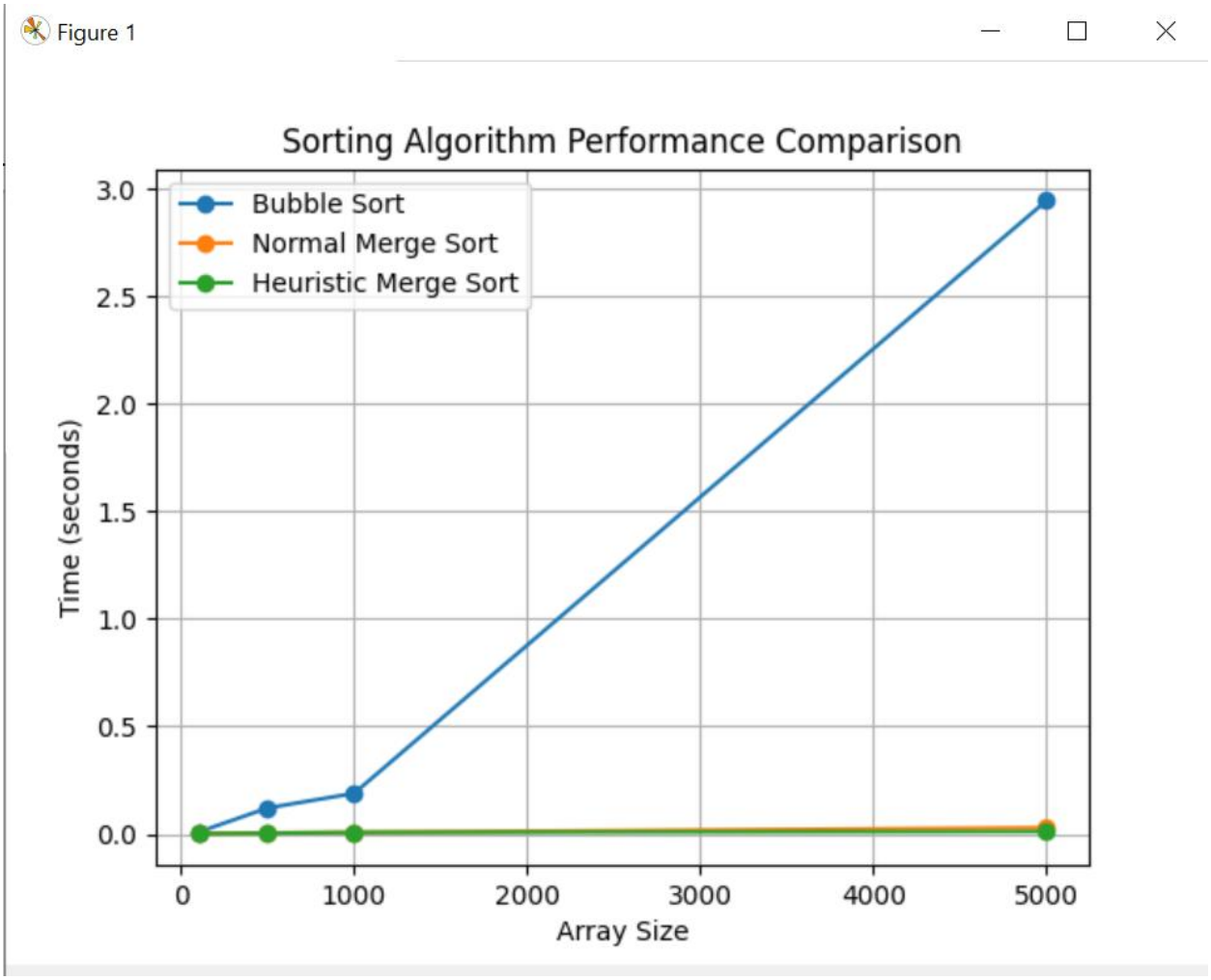
All three Sorts Comparisons:

All three sorts comparison results i: e Bubble sort (dumb sort), merge and heuristics merge sort results are Given below

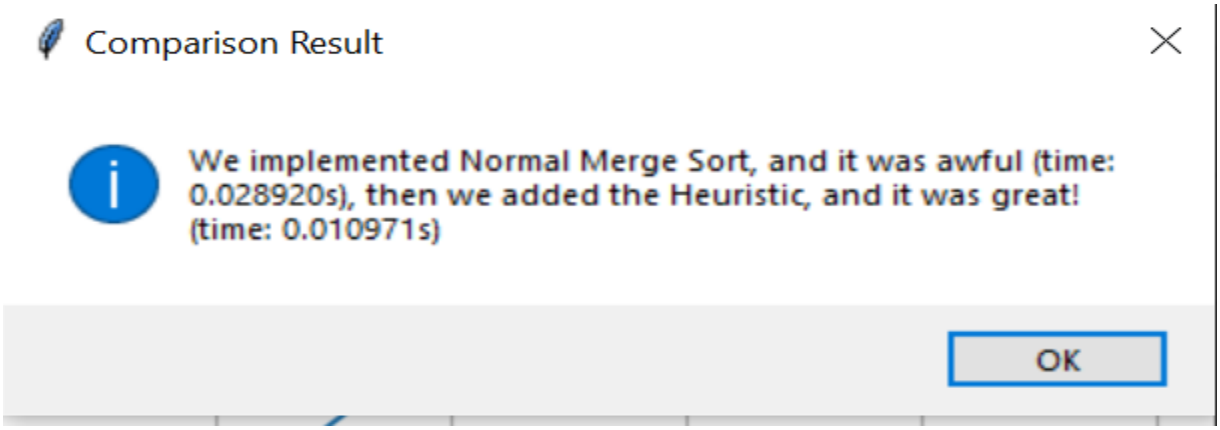


For Random 5000 values:

When we use 5000 random values its show results which is extremely good

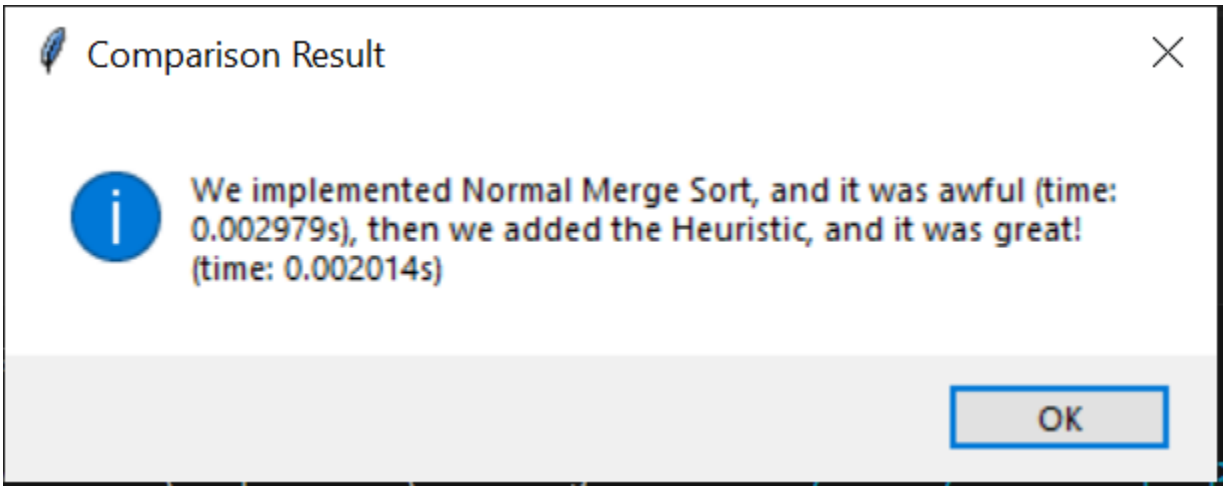


Time Consumption on Random Data:



Final Result:

After comparing all algorithms, a pop-up message summarizes the times for **Normal Merge Sort** and **Heuristic Merge Sort** with the specified statement.



Merge Sort Code:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        # Recursively sort both halves
        merge_sort(L)
```

```
merge_sort(R)


# Merging the two sorted halves
i = j = k = 0
while i < len(L) and j < len(R):
    if L[i] < R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1

# Copy remaining elements of L (if any)
while i < len(L):
    arr[k] = L[i]
    i += 1
    k += 1

# Copy remaining elements of R (if any)
while j < len(R):
    arr[k] = R[j]
    j += 1
```

```
        k += 1
```

Heuristic Merge Sort (Modified to switch to Insertion Sort for small subarrays)

```
def insertion_sort(arr, left, right):
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key


def heuristic_merge_sort(arr, left, right):
    # Use Insertion Sort for small subarrays
    if right - left + 1 <= 10:  # Threshold can be adjusted
        insertion_sort(arr, left, right)
        return

    if left < right:
```

```
        mid = (left + right) // 2
        heuristic_merge_sort(arr, left, mid)        # Recursively sort left half
        heuristic_merge_sort(arr, mid + 1, right)   # Recursively sort right half
        merge(arr, left, mid, right)                # Merge the two sorted halves


def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid
    L = arr[left:left + n1]
    R = arr[mid + 1:mid + 1 + n2]
    i = j = 0
    k = left
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = L[i]
```

```
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
```

Changes Made to Heuristic Merge Sort:

- **Threshold for Insertion Sort**: If the size of the subarray (right - left + 1) is less than or equal to a certain threshold (e.g., 10), **Insertion Sort** is used instead of Merge Sort. This is because Insertion Sort performs better on small arrays due to lower overhead.
- **Use of Insertion Sort**: This small optimization can significantly improve performance for small-sized arrays or datasets that do not require full merge sorting.

Why This Works:

- **Insertion Sort** is efficient on small datasets because its overhead is lower compared to Merge Sort.
- By using **Insertion Sort** for small subarrays, **Heuristic Merge Sort** reduces the unnecessary overhead and improves performance, especially when merging small portions of data.

Time Measurement:

The time measurement for both **Normal Merge Sort** and **Heuristic Merge Sort** is captured during the sorting process and is displayed for comparison.