**Description of the Selected Problem:**

Universities manage vast databases containing thousands of students and alumni records, including names, contact details, and other personal information. A common requirement is to produce alphabetically sorted lists of names for various purposes, such as graduation booklets, alumni fundraising events, event planning, or administrative reports. For example, a graduation ceremony may need a booklet listing all graduating students in alphabetical order, or an alumni office may require a sorted directory for outreach campaigns. Given the large dataset sizes (potentially tens of thousands of records) the sorting process must be fast, reliable, and capable of handling dynamic updates efficiently. Additionally, the system should present results clearly to staff, allowing them to verify correctness and measure performance for optimization. My solution implements two sorting algorithms, Merge Sort and Quick Sort, to address this need, with GUI to input dataset sizes, display sorted names, and compare execution times.

**Description of Both Algorithms:**

The implemented system uses two sorting algorithms - Merge Sort and Quick Sort to sort arrays of names (stored as strings in the format "FirstName LastName"). Below is a description of each algorithm:

1. **Merge Sort**:

Merge Sort is a divide-and-conquer algorithm that recursively splits the input array into smaller subarrays, sorts them, and merges them back into a fully sorted array. MergeSort.java operates as follows:

* **Base Case**: If the array has one or fewer elements, it is already sorted, so return.
* **Divide**: Split the array into two halves, left (from index 0 to middle) and right (from middle to end).
* **Conquer**: Recursively apply Merge Sort to the left and right subarrays.
* **Merge**: Combine the sorted left and right subarrays into the original array (result) by comparing elements and placing them in order. The merge method uses three pointers (i for result, l for left, r for right) to iterate through both subarrays, selecting the smaller element at each step. Remaining elements from either subarray are appended afterward.

1. **Quick Sort:**

Quick Sort is also a divide-and-conquer algorithm that sorts by selecting a pivot and partitioning the array around it. QuickSort.java works as follows:

1. **Base Case**: If the start index is greater than or equal to the end index, stop recursion.
2. **Partition**: Choose the last element as the pivot (pivotValue). Iterate through the subarray from start to end-1, swapping elements smaller than the pivot to the left side. Track the position of the last smaller element (smallerElementIndex). After the loop, place the pivot in its correct position by swapping it with the element at smallerElementIndex + 1. Return the pivot’s final index.
3. **Recurse**: Recursively apply Quick Sort to the subarrays before and after the pivot (excluding the pivot itself).

Additionally, the GUI allows users to input the number of names to sort, generates random names using NameGenerator, runs both algorithms, and displays the first 100 sorted names along with execution times. This setup simulates the university’s need to sort and review large name lists efficiently.

**Theoretical Analysis (Time and Space Complexity, Big-O):**

To determine which algorithm best suits the university database’s needs, we can analyze the time and space complexities of Merge Sort and Quick Sort.

**Merge Sort**

* **Time Complexity**:
  + **Best Case**: O(n log n). The algorithm always divides the array into two halves and merges them, requiring log n levels of recursion, with each level performing O(n) comparisons during merging.
  + **Average Case**: O(n log n). The performance is consistent regardless of input distribution, because the splitting is fixed.
  + **Worst Case**: O(n log n). Even with sorted or reverse-sorted inputs, the algorithm’s steps remain unchanged.
  + **Reason**: The recursive division into halves (log n) and linear merging (n) result in a stable O(n log n) across all cases.
* **Space Complexity**: O(n). Merge Sort requires additional arrays (left and right) during the merge process, making it O(n) additional space. The recursion stack adds O(log n), but the dominant term is O(n) due to the temporary arrays.
* **Advantages**:
  + Guaranteed O(n log n) performance, ideal for predictable sorting of large datasets like university records.
  + Stable sorting (preserves relative order of equal elements), which is less critical for names unless additional sorting criteria apply.
* **Disadvantages**:
  + Higher space usage due to additional arrays, which could be a concern for memory-constrained systems.

**Quick Sort**

* **Time Complexity**:
  + **Best Case**: O(n log n). Occurs when the pivot consistently divides the array into roughly equal halves, leading to log n recursion levels with O(n) work per level.
  + **Average Case**: O(n log n). With random pivots (or random inputs, as in the name generator), partitions are usually balanced, resulting in logarithmic depth.
  + **Worst Case**: O(n²). Occurs with poor pivot choices (already sorted or reverse-sorted arrays), causing unbalanced partitions, leading to n recursion levels with O(n) work each.
  + **Reason**: My implementation uses the last element as the pivot, which risks O(n²) on sorted inputs. Random inputs (like generated names) mitigate this, resulting in O(n log n).
* **Space Complexity**: O(log n) average, O(n) worst case. Quick Sort requires only O(log n) stack space for recursion in balanced cases. In the worst case (unbalanced partitions), the recursion stack can grow to O(n). No additional arrays are needed beyond temporary variables for swapping.
* **Advantages**:
  + In-place sorting minimizes memory usage, beneficial for large datasets on memory-limited systems.
  + Generally faster in practice due to cache efficiency and fewer memory allocations.
  + Average-case O(n log n) suits most real-world scenarios.
* **Disadvantages**:
  + Worst-case O(n²) can occur with poor pivot selection (e.g., sorted alumni lists), although it can be avoided by better pivot strategies.

**Comparison and Suitability:**

**Table with Run-times for 10,000 names:**

|  |  |  |
| --- | --- | --- |
|  | Merge Sort | Quick Sort |
| 1st Run | 6.71ms | 3.69ms |
| 2nd Run | 7.70ms | 8.72ms |
| 3rd Run | 7.09ms | 2.63ms |
| 4th Run | 2.58ms | 2.67ms |
| 5th Run | 2.50ms | 2.44ms |
| 6th Run | 2.62ms | 2.32ms |
| 7th Run | 2.58ms | 2.50ms |
| 8th Run | 2.44ms | 2.55ms |
| 9th Run | 2.53ms | 2.72ms |
| 10th Run | 2.39ms | 2.61ms |
| Average | 3.91ms | 3.29ms |

* **Average**: Quick Sort (3.29ms) is faster than Merge Sort (3.91ms).
* **Variability**: High outliers in early runs (e.g., 7.70ms Merge Sort, 8.72ms Quick Sort in 2nd run), stabilizing to 2.4–2.7ms later.
* **Trend**: Quick Sort generally outperforms, except in cases like the 2nd run where it spikes (8.72ms).

**Table with Run-times for 100,000 names:**

|  |  |  |
| --- | --- | --- |
|  | Merge Sort | Quick Sort |
| 1st Run | 49.93ms | 49.89ms |
| 2nd Run | 41.33ms | 38.90ms |
| 3rd Run | 29.39ms | 33.21ms |
| 4th Run | 31.05ms | 36.48ms |
| 5th Run | 32.23ms | 36.65ms |
| 6th Run | 28.53ms | 32.71ms |
| 7th Run | 28.76ms | 35.91ms |
| 8th Run | 31.31ms | 38.31ms |
| 9th Run | 30.12ms | 40.14ms |
| 10th Run | 28.55ms | 34.79ms |
| Average | 33.12ms | 37.70ms |

* **Average**: Merge Sort (33.12ms) is faster than Quick Sort (37.70ms), opposite of the 10,000-name trend.
* **Variability**: Times range from 28.53ms to 49.93ms (Merge Sort) and 32.71ms to 49.89ms (Quick Sort), with less dramatic outliers than the 10,000 case but still notable (e.g., 1st run ~49ms for both).
* **Trend**: Merge Sort consistently outperforms Quick Sort, with smaller gaps in early runs and larger gaps later (e.g., 28.55ms vs. 34.79ms.

|  |  |  |
| --- | --- | --- |
| Feature | Merge Sort (n = 10,000) | Quick Sort (n = 10,000) |
| Temporary Arrays | Yes (Left and Right subarrays) | No (In-place partitioning) |
| Max Memory Use | ≈ 81.8 KB | ≈ 1.34 KB |
| Recursion Depth | log₂(n) ≈ 14 | log₂(n) ≈ 14 (average case) |
| Stack Frame Size | ~128 bytes | ~96 bytes |
| Garbage Friendly | May allocate many temp arrays | Minimal extra allocations |
| Stability | Stable (preserves order) | Not stable |

**Memory Efficiency**: Quick Sort is significantly more memory-efficient due to its in-place strategy. Merge Sort requires additional arrays at each recursion level, leading to more temporary allocations.

**Suitability**: If memory usage is a constraint, then Quick Sort is preferred. If stability is needed or data is very large, Merge Sort offers a more predictable performance.

**What Changed and Why (10,000 to 100,000 Names):**

Between sorting 10,000 names and 100,000 names, the performance dynamics shift: Quick Sort is faster on average for 10,000 names (3.29ms vs. Merge Sort’s 3.91ms), but Merge Sort outperforms Quick Sort for 100,000 names (33.12ms vs. 37.70ms). This occurs because:

* **Quick Sort’s Pivot Variability**: Quick Sort’s last-element pivot works well for random names at 10,000, but with 100,000 names, the larger array increases the chance of unbalanced partitions, adding comparisons/swaps and increasing times.
* **Merge Sort’s Consistency**: Merge Sort’s O(n log n) performance is unaffected by input distribution, and its predictable memory patterns scale better for larger datasets, despite O(n) array allocation.
* **System Effects**: Both algorithms face JVM warm-up and garbage collection (GC) costs, but these are more noticeable at 100,000 names. Merge Sort’s fixed allocation optimizes over runs, while Quick Sort’s variable partitions cause persistent variability.

**Final Conclusion: Which is Better and When?**

* **Quick Sort** is better for smaller datasets (e.g., <10,000 names) and non-criticaltasks (such as sorting event lists), where low memory usage (~1.34 KB) and faster average speed (3.29ms) make it a highly efficient choice. It performs sorting in-place, using minimal extra memory by avoiding temporary arrays. While it may occasionally spike in runtime due to poor pivot selection (e.g., 8.72ms), its typical performance is excellent for scenarios that prioritize speed and memory efficiency over consistency.
* **Merge Sort** is better for larger datasets (e.g., >100,000 names) and critical applications (like graduation booklets or alumni directories), where consistent performance (average 33.12ms) and predictable scaling (O(n log n)) are crucial. Despite its higher memory usage (~81.8 KB) because of the creation of temporary left and right arrays during each recursive step, it offers stable sorting behavior and avoids negative performance spikes like Quick Sort. This makes it ideal when reliability, stability, and scalability are more important than minimizing memory usage.