Project 2 Sorting Report

Using `std::chrono::high_resolution_clock::now()` I timed each algorithm after supplying each input file via `std::ifstream` and these are the results I obtained below. To calculate average, the times for each input were averaged to the nearest millisecond. Under each input, you will see the correct median listed which was used to verify if each sort returned the correct median. Lastly, worst case quick select input produces an input of 20k elements which was also fed to each algorithm to sort fully or halfway to find the median three times and the average of those three times are also available on the last table. I was timing on MacBook Air, M2, 2022, Apple Silicon M2 chip, 8 GB memory and results may vary across devices.

Input sizes for input 1, 2 and 3 are 1k (1,000)

Algorithm	Input 1 (50492874)	Input 2 (19250688)	Input 3 (70244369)	Average (rounded to nearest ms)
HalSelectionSort	10ms	6ms	10ms	9ms
StandardSort	0ms	0ms	0ms	0ms
MergeSort	2ms	1ms	2ms	2ms
InPlaceMergeSort	1ms	0ms	1ms	1ms
HalfHeapSort	0ms	0ms	0ms	0ms
QuickSelect	0ms	0ms	0ms	0ms
MedianOfMedians	0ms	0ms	0ms	0ms

Input sizes for input 4, 5 and 6 are 31k (31,623)

Algorithm	Input 4 (50173306)	Input 5 (18637175)	Input 6 (70984972)	Average (rounded to nearest ms)
HalSelectionSort	3445ms	3447ms	3488ms	3460ms
StandardSort	1ms	1ms	1ms	1ms
MergeSort	29ms	30ms	29ms	29ms
InPlaceMergeSort	14ms	14ms	15ms	14ms
HalfHeapSort	3ms	3ms	4ms	3ms
QuickSelect	0ms	0ms	0ms	0ms
MedianOfMedians	0ms	0ms	0ms	0ms

Input sizes for input 7, 8 and 9 are 1M (1,000,000)

Algorithm	Input 7 (49971079)	Input 8 (18675104)	Input 9 (70722421)	Average (rounded to nearest ms)
HalSelectionSort	Input too big	Input too big	Input too big	N/A
StandardSort	48ms	53ms	49ms	50ms
MergeSort	1043ms	1041ms	1047ms	1044ms
InPlaceMergeSort	545ms	545ms	548ms	546ms
HalfHeapSort	157ms	159ms	159ms	158ms
QuickSelect	30ms	27ms	29ms	20ms
MedianOfMedians	23ms	23ms	23ms	23ms

Inputs produced by the worst case quickselect input and are of size 20k (20,000)

Algorithm	Trial 1 - 20k	Trial 2 - 20k	Trial 3 - 20k	Average
HalSelectionSort	1156ms	1152ms	1155ms	1154ms
StandardSort	0ms	0ms	0ms	0ms
MergeSort	15ms	15ms	16ms	15ms
InPlaceMergeSort	6ms	6ms	6ms	6ms
HalfHeapSort	1ms	1ms	1ms	1ms
QuickSelect	358ms	356ms	356ms	357ms
MedianOfMedians	0ms	0ms	0ms	0ms

Algorithmic Analysis:

- HalSelectionSort (O(n^2) comparisons): This brute-force method iterates through the list, finding the minimum element and swapping it with the first. While it was kind of simple, its performance suffers for larger datasets.
- StandardSort: As expected, the built-in function shines with its efficient implementation and consistently fast performance across all input sizes.
- Merge Sort and InPlaceMerge Sort (O(n log n) comparisons): These divide-and-conquer algorithms recursively split the list, sort each sub-list, and then merge them. Their time complexity explains the steady performance increase with input size.
- HalfHeapSort (O(n log n) comparisons): This algorithm builds a heap from the first half of the list, then inserts and adjusts the heap for the remaining elements. Its performance is comparable to Merge Sort, demonstrating the versatility of heap-based sorting.
- QuickSelect and MedianOfMedians (O(n) expected time, O(n^2) worst-case): These methods randomly select a pivot element and partition the list around it. While their

average performance is excellent, the worst-case scenario can be significantly slower, as seen with the worst-case QuickSelect input. The 0ms was surprising though.

Observations and Discussion:

- The results mostly align with the expected time complexities. StandardSort, Merge Sort, InPlaceMerge Sort, HalfHeapSort, QuickSelect, and MedianOfMedians all outperform HalSelectionSort, as predicted.
- QuickSelect and MedianOfMedians shine with their O(n) expected time, but their worst-case scenarios are significantly slower. This highlights the importance of choosing the right algorithm for the specific needs of your application.
- It's safe to say that algorithms like Merge Sort and Heap Sort can save significant time on large datasets by stopping the sort once the median is found.

Comparing the Methods:

- Worst-case performance: StandardSort, Merge Sort, and InPlaceMerge Sort are the go-to choices with their guaranteed O(n log n) complexity.
- Average performance: QuickSelect and MedianOfMedians excel, but remember their worst-case is horrendous.
- Large inputs: StandardSort remains efficient, while Merge Sort and InPlaceMerge Sort offer good performance. QuickSelect and MedianOfMedians can be great choices if early termination is utilized effectively.

Interesting and Unexpected Results:

- The small difference between StandardSort, Merge Sort, and InPlaceMerge Sort for larger inputs was surprising. It suggests that the overhead of Merge Sort and InPlaceMerge Sort might be minimal in this specific implementation.
- The consistent good performance of HalfHeapSort was noteworthy. While often overshadowed by other algorithms, it proved to be a strong contender in this analysis.