Scott Johnson

05/06/25

**Analysis Report**

**Introduction:**

This project analyzes the performance of two sorting algorithms: Shell Sort and Quick Sort. The purpose of this study is to understand their efficiency in terms of operation counts and execution time, and to evaluate how closely the empirical results align with their theoretical Big-O complexity. The project also addresses JVM warm-up effects and the impact of algorithm sensitivity to input data.

**High-Level Pseudocode:**

**Shell Sort:**

procedure shellSort(A):

n = length(A)

gap = n / 2

while gap > 0:

for i = gap to n - 1:

temp = A[i]

j = i

while j >= gap and A[j - gap] > temp:

A[j] = A[j - gap]

j -= gap

A[j] = temp

gap = gap / 2

**Quick Sort:**

procedure quickSort(A, low, high):

if low < high:

pivot = partition(A, low, high)

quickSort(A, low, pivot - 1)

quickSort(A, pivot + 1, high)

procedure partition(A, low, high):

pivot = A[high]

i = low - 1

for j = low to high - 1:

if A[j] <= pivot:

i += 1

swap A[i] and A[j]

swap A[i+1] and A[high]

return i + 1

**Big-O Analysis:**

* **Shell Sort:** The time complexity depends on the gap sequence used. With Shell's original gaps, the worst case is ) , but with Hibbard or Sedgewick sequences, the average time complexity can approach .
* **Quick Sort:** Has an average-case time complexity of and a worst-case of ) when the pivot selection is poor. With good pivot strategies the worst-case is mitigated.

**JVM Warm-Up Strategy:**

To mitigate the influence of JVM warm-up, 200 warm-up iterations were performed for each sorting algorithm before benchmarking began. This ensures the JVM is in a steady state and produces consistent timing measurements.

**Critical Operation Counted:**

For Shell Sort, the critical operation is the number of element shifts during insertion in each subarray. For Quick Sort, the critical operation is the number of element comparisons and swaps during partitioning. These operations reflect the core workload that contributes to the time complexity of each algorithm.

**Results Analysis:**

Graphical Results:

Included two graphs, one showing average critical operations vs. input size, another showing average execution time vs. input size for both algorithms.

**Performance Comparison:** Quick Sort consistently outperformed Shell Sort in terms of execution time across all input sizes. However, Shell Sort showed more stable results with smaller standard deviations, especially in smaller arrays.

**Operation vs. Time Comparison:** While both operation counts and execution time increased with input size, Quick Sort's lower operation count more directly translated into better performance. However, due to Quick Sort's recursive overhead, the performance gains were more apparent at larger sizes.

**Coefficient of Variation:** Quick Sort had a higher coefficient of variation in both operation count and execution time. This suggests Quick Sort is more sensitive to input distribution due to its dependence on pivot quality. Shell Sort, while slower, was more consistent.

**Shell Sort Time Spikes:** An unusual spike in Shell Sort's average execution time occurred around input sizes 300–400. This can be attributed to several factors:

* Cache and memory access inefficiencies at specific gap values.
* Inefficiencies in the default gap sequence, leading to more element movements.
* JVM runtime noise, such as garbage collection or optimization delays.
* Branch prediction inconsistencies, where unpredictable conditionals during sorting slow down CPU execution.

These observations reinforce the need to analyze multiple performance metrics rather than relying solely on raw time data.

**Comparison to Big-O Analysis:** Empirical results align with theoretical expectations. Quick Sort displayed -like behavior on average, while Shell Sort showed behavior between and ) depending on the input size.

**Conclusion:**

This analysis shows that Quick Sort is generally more efficient than Shell Sort for larger data sizes, both in terms of execution time and operation count. However, Shell Sort offers more consistent performance due to its iterative nature. The analysis highlights the importance of understanding both algorithm design and real-world performance metrics when selecting a sorting algorithm. JVM warm-up handling, choice of critical operations, and input sensitivity are crucial factors that influence benchmarking accuracy and interpretation.