Impact of Array Configuration on Quicksort Performance: An Empirical Study

Justina Bradshaw
University of the Fraser Valley
Abbotsford, BC
Justina.Bradshaw@student.ufv.ca

Bhupinder Gosal
University of the Fraser Valley
Abbotsford, BC
Bhupinder.Gosal@student.ufv.ca

Noor Sidhu
University of the Fraser Valley
Abbotsford, BC
Eknoor.Sidhu@student.ufv.ca

***Abstract*—This paper presents an empirical evaluation of the time performance of the Quicksort algorithm across varying array types and sizes. We investigate four different pivot selection strategies: first element, last element, middle element, and random element. We implement these strategies in Java and run the algorithm across arrays of various characteristics (size, sorting order). We measure each strategy's performance in terms of elapsed time (in milliseconds) using a system timer. Our findings give insight into how different pivot strategies impact sorting efficiency under varying conditions. The results suggest that pivot position generally affects performance, with the random and middle pivots outperforming the others, regardless of array size.**

Keywords—Quicksort, sorting algorithm, time complexity, pivot selection

I. Project Outline

We designed our project as follows: First, we developed a quicksort algorithm in pseudocode and then translated it into Java code. We provided background information and made predictions regarding its time complexity. Next, we implemented our quicksort algorithm with four pivot strategies: first, last, middle and random. We tested this implementation on arrays of various sizes and types, where "type" refers to the array's initial sorting order (sorted, random, or reverse-sorted), not the data type (e.g., integers). We recorded the performance data as well as system information into excel sheets and created graphs and charts to visualize the results. We compared the effectiveness of different pivot strategies and looked for reasonable inferences based on our collective data. Finally, we drew conclusions based on our observations and insights.

A. Background Information

1) General Overview: Quicksort is a commonly used, fast algorithm to sort arrays. British computer scientist Tony Hoare invented the algorithm in 1959 and formally published it in 1962. It belongs to the category of "divide and conquer" algorithms, which break down an array into smaller, more manageable sub-arrays [2, pp. 10-16].

To begin, quicksort selects a pivot element—typically chosen as the first, middle, last, or a random element. The next step is to partition the array by moving elements smaller than the pivot to the left and elements greater than the pivot to the right. This process creates two sub-arrays, each of which is then recursively sorted. The recursion continues until the base case is reached, which occurs when sub-arrays have a size of zero or one and are therefore already sorted. Once all recursive calls are completed, the array is fully sorted. Since quicksort performs the sorting "in place," it does not require additional memory beyond the recursion stack. This contrasts with merge sort, which requires extra space for merging the sub-arrays [1, pp. 236-238].

2) Visualization: Quicksort can be visualized as a binary search tree, where the root represents the pivot position [1, pp. 236-238]. Each recursive call creates a new level in the tree, with nodes representing pivots and sub-arrays. While the program uses arrays, the tree offers a useful way to conceptualize the recursive process.

To illustrate, consider an unsorted array of positive integers:

A = [3, 6, 1, 5, 2]

Choosing the last element (value = 2) as the pivot, the array is partitioned into two sub-arrays: elements less than 2 and elements greater than 2, with the pivot placed in its correct sorted position.

New Partitions:

- $x < 2$: [1]
- $x > 2$: [3, 6, 5]

After partitioning, the array is rearranged to:
A = [1, 2, 6, 5, 3]
with the value 2 in its correct position at index 1. (Note that this partitioning is not final, as the two sub-arrays still need to be recursively sorted. A breakdown of all recursive calls and base cases is provided in the BST representation.)

Binary Search Tree Visualization

In the tree structure, the pivot (value = 2) serves as the root node. The tree is partitioned into two subtrees:
Tree Structure:

- Root Node: Pivot (value = 2)
- Left Subtree: [1] (already sorted)
- Right Subtree: [3, 6, 5] (initially)

Recursive Calls and Final Tree Visualization:

```
        2
       / \
      1   6
          /
        5
       /
      3
```

There is one initial call and four additional recursive calls. While base cases don't require further recursion, they are counted to indicate the recursion's termination.

First Recursive Call: (on [3, 6, 5])
- Pivot: 6 (last pivot rule)
- Subtree: Elements less than 6: [3, 5]

Second Recursive Call: (on [3, 5])
- Pivot: 5 (last pivot)
- Elements less than 5: [3] (already sorted)
- Base Case: Element 3 (no sorting needed)
- Base Case: Empty subarray from 6 (no sorting needed)

B. *Create Quicksort Algorithm*

1) Pseudocode: We created a high-level English-language algorithm to demonstrate the logic of the program.

2) Java Code: We translated our pseudocode into Java code, which we each ran individually on our separate machines. Curiously, we also translated the Java code into Python to test on one machine to compare time complexity of both implementations. The full pseudocode, Java code, Python code, along with the accompanying README document are available in a separate file hosted on our GitHub respository [https://github.com/bgosal/COMP359-Assignment1.git].

C. *Predictions for Different Pivot Strategies:*

1) *Expectations:* We recognise that the performance of quicksort can vary across different scenarios. We can describe these scenarios using Big-O Notation.
   - *$O(n^2)$* Time Complexity: Quadratic time represents the worst-case scenario occuring when pivot selection leads to unbalanced partitions. When a pivot is poorly chosen (e.g., smallest element in first pivot strategy), each partition removes only one element from consideration, resulting in *n* levels of recursion requiring *n* comparions. Therefore, the total time complexity becomes:

$$O(n \text{ x } n) = O(n^2)$$

   - *O(n log n)* Time Complexity: Linearithmic time represents the average-case scenario (which is optimal for quicksort). This happens when a good pivot choice (e.g., middle pivot in reverse sorted) divides the array into balanced partitions at each recursive call. The recursion depth would be log(n) levels, with n comparisons. Therefore, the time complexity would be:

$$O(n \log n)$$

We predicted the time efficiency of each pivot strategy as follows:

2) *First Pivot:*
   - Sorted Arrays (($O(n^2)$)): Using the first element leads to highly unbalanced partitions, resulting in deep recursion and worst-case performance.
   - Unsorted Arrays (O(n log n)): The first element is more likely to create balanced partitions, reducing recursion depth and achieving average-case performance.
   - Reverse Sorted Arrays ($O(n^2)$): Choosing the first element results in one small and one large partition, leading to deep recursion and worst-case performance.

3) *Last Pivot:*
   - Sorted Arrays ($O(n^2)$): Choosing the last element creates a small and large partition, resulting in highly unbalanced partitions and deep recursion, leading to worst-case performance.
   - Unsorted Arrays (O(n log n)): The last element often results in more balanced partitions compared to the first pivot, leading to average-case performance.
   - Reverse Sorted Arrays ($O(n^2)$): Selecting the last element results in highly unbalanced partitions, leading to deep recursion and worst-case performance.

4) *Middle Pivot:*
   - Sorted Arrays (O(n log n)): Choosing the middle element creates balanced partitions, reducing recursion depth and achieving average-case performance.
   - Unsorted Arrays (O(n log n)): The middle element generally results in balanced partitions, reducing recursion depth and achieving average-case performance.
   - Reverse Sorted Arrays (O(n log n)): Selecting the middle element usually creates balanced partitions, reducing recursion depth and achieving average-case performance.

5) *Random Pivot:*

- Sorted Arrays (O(n log n)): A random pivot avoids systematic bias, creating more balanced partitions on average and reducing recursion depth.
- Unsorted Arrays (O(n log n)): Randomly choosing a pivot creates balanced partitions and prevents unbalanced recursion, resulting in average-case performance.
- Reverse Sorted Arrays (O(n log n)): Random pivoting disrupts order, increasing the chance of balanced partitions and reducing recursion depth.

D. *Test Arrays:* We understand that the type and size of arrays can influence the time complexity of the quicksort algorithm. We also recognize the need to define a set of conditions for the elements in each array. For example, we use integers rather than floats for simplicity, (although floats and other numeric data types can be sorted with quicksort without issue).

We also need to avoid instances where all the array values are identical. In this scenario, the pivot will be the same as every element, leading to inefficient partitioning. Without taking additional measures, the quicksort algorithm would repeatedly process the same elements, potentially leading to an infinite loop [2]. Although arrays with all identical values are uncommon in most scenarios, we wanted to avoid such a case in our program. In developing our quicksort implementation, we considered multiple factors affecting the running time of our program, as discussed in [1, pp. 46-47].

E. *Observations:* We first compared the results of quicksort on the arrays of each size, independently.

Performance for Small Sized Arrays

Sorted:

- First Pivot (0.035 ms): Higher-than-average performance (0.035 ms) but is significantly lower than the last pivot, which was unexpected.
- Last Pivot (0.118 ms): Longest execution time, as anticipated, because it creates highly imbalanced partitions in a sorted array, leading to deep recursion and worst-case performance.
- Middle Pivot (0.015 ms): Much faster run time, as predicted.
- Random Pivot: (0.021 ms): Much faster run time, as predicted.

Unsorted:

- First Pivot (0.022 ms): As expected, the first pivot performs well, likely creating relatively balanced partitions, resulting in average-case performance.
- Last Pivot (0.020 ms): The last pivot strategy performs the best, showing that this method can also result in balanced partitions for random arrays.
- Middle Pivot (0.024 ms): The middle pivot performs well, though it is slightly slower than the first and last pivot strategies, possibly due to some variation in how partitions are balanced.
- Random Pivot (0.024 ms): The random pivot has the same performance as the middle pivot, which aligns with our expectations, as both strategies aim to create balanced partitions.

Reverse Sorted:

- First Pivot (0.023 ms): Performs better than expected, given that a first pivot on a reverse sorted array should lead to highly imbalanced partitions.
- Last Pivot (0.032 ms): Performs much slower than the middle and random pivots, for the same reason we expected the first pivot to be slow—it creates highly unbalanced partitions.
- Middle Pivot (0.021 ms): Performs roughly the same as the random pivot, as expected, for the same reason we expect fast execution on sorted arrays.
- Random Pivot (0.02 ms): Performs roughly the same as the middle pivot, as expected, for the same reason we expect fast execution on sorted arrays.

For an array size of 10, the performance results are shown in Figure 1.

Performance for Medium Sized Arrays

Sorted:

- First Pivot (0.1894 ms): Performs at the slowest execution time, as expected.
- Last Pivot (0.1061 ms): Performs at the second slowest execution time, much closer to the first pivot than the other two, as expected.
- Middle Pivot (0.0475 ms): Performs similar to random pivot, roughly half to a third the speed of the first and last pivots, as expected.
- Random Pivot (0.0513 ms): Performs similar to the middle pivot, roughly half to a third the speed of the other two, as expected.

Unsorted:

- First Pivot (0.0569 ms): Performs well on random arrays, as anticipated, likely creating more balanced partitions.
- Last Pivot (0.0548 ms): Performs well on random arrays, with a slightly lower time than the first pivot, most likely balanced in the partitions.
- Middle Pivot (0.0513 ms): Performs faster than the rest.
- Random Pivot (0.0579 ms): Performs closer to first and last pivot.

Reverse Sorted:

- First Pivot (0.0851 ms): Performs the slowest, poor performance is expected.
- Last Pivot (0.0796 ms): Performs the second slowest, expected to be similar to the first pivot.
- Middle Pivot (0.0337 ms): Performs significantly faster than first and last, similar to random pivot, as expected.
- Random Pivot (0.03722 ms): Performs significantly faster than first and last, similar to the middle pivot, as expected.

For an array size of 100, the performance results are s hown in Figure 2.

Performance for Large Arrays

Sorted:

- First Pivot (3.730 ms): Performs significantly poorly, as anticipated due to highly imbalanced partitions.
- Last Pivot (4.182 ms): Performs the absolute worst, for the same reason as the first pivot.
- Middle Pivot (0.233 ms): Performs significantly better, as anticipated, due to more balanced partitions.
- Random Pivot (0.226 ms): Performs the fastest, as anticipated, for the same reason as the middle pivot.

Random Arrays:

- First Pivot (0.197 ms): Performs the best on random arrays, much faster than we expect on sorted/reverse sorted arrays.
- Last Pivot (0.215 ms): Performs well but not as fast as the first pivot.
- Middle Pivot (0.219 ms): Performs about the same as the last pivot.
- Random Pivot (0.285 ms): Performs at the longest execution time, but not drastically different than the previous pivot points.

Reverse Sorted:

- First Pivot (0.610 ms): Performs poorly, nearly as slow as the last pivot, as anticipated for sorted arrays.
- Last Pivot (0.725 ms): Performs with the longest execution time, for the same reason as the first pivot.
- Middle Pivot (0.172 ms): Performs the fastest, most likely containing more balanced partitions.
- Random Pivot (0.216 ms): Performs fast but not as much as the middle pivot, probably creates a similar number of partitions, with similar levels of recursion.

For an array size of 1000, the performance results are shown in Figure 3.

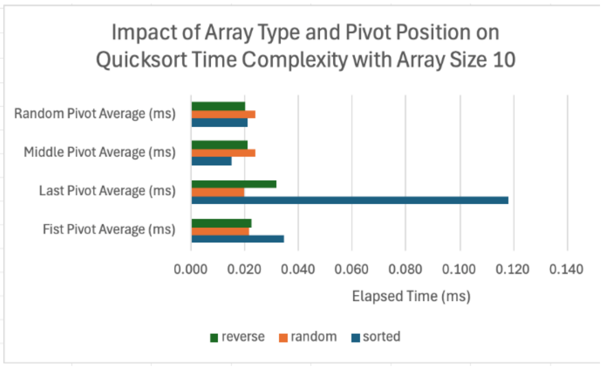

Figure 1: Impact of Array Type and Pivot Position on Quicksort Time Complexity with Array Size 10

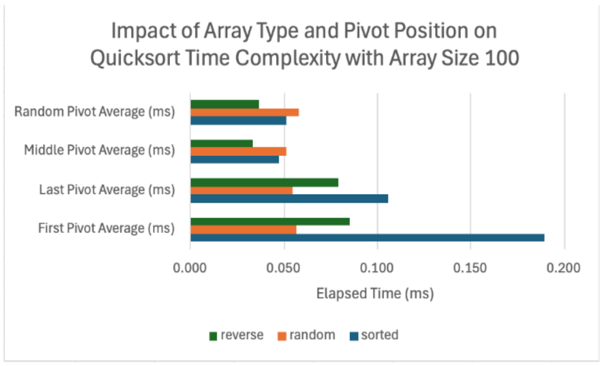

Figure 2: Impact of Array Type and Pivot Position on Quicksort Time Complexity with Array Size 100

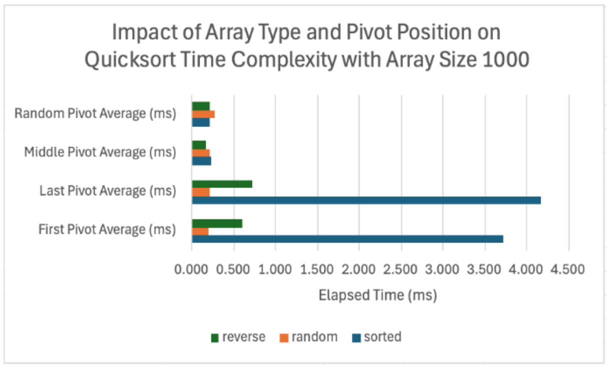

Figure 3: Impact of Array Type and Pivot Position on Quicksort Time Complexity with Array Size 1000

Hardware Considerations

In our investigation, we analyzed how specific hardware variables relate to the overall averaged elapsed time (in ms) for various array types. We focused on two key variables: clock speed (GHz) and RAM (GB).

- Clock Speed (GHz):
  - Example: Each of our machines operates at different clock speeds: 2.1 GHz, 3.2 GHz, and 4.2 GHz. Intuitively, one would expect that the machine with the highest clock speed would exhibit the lowest execution time. However, in the case of the Overall First Pivot Average, the machine running at 4.2 GHz recorded a slower time of 0.219 ms compared to the 3.2 GHz machine, which performed at 0.139 ms.
- RAM (GB):
  - Example: One machine has 8GB of RAM while the other two have 16GB. The common assumption is that machines with double the RAM would execute the quicksort algorithm more efficiently. Surprisingly, our results did not support this expectation; the machine with 8 GB achieved a time of 0.139 ms, which falls between the times of the two 16 GB machines, 0.085 ms and 0.502 ms.

These findings suggest that we could not establish a clear relationship between clock speed, RAM, and overall averaged elapsed time (in ms). For further details, please refer to the tables in the "Combined Analysis" sheet within our Excel file.

Python vs. Java

After interpreting the results from our Java implementation, we curiously decided to translate our code into Python to test on one machine (Intel Core i7-7700K). Our results showed that the Python implementation performs significantly slower on specifically sorted and reverse-sorted arrays. This comparison also displayed how critical pivot selection can be in quicksort. In Python, we noticed that the first pivot approach takes over ten times as long as it does in the Java implementation for reverse-sorted arrays. There is a similar pattern in the last pivot approach. Overall, we can conclude that Python has worse average time complexity executing quicksort when considering first and last pivot selection in arrays that are not randomly sorted. One potential reason for this time difference may be due to the fact that Java is a statically-typed, compiled language. Whereas Python is an interpreted, dynamically-typed language, which generally makes it slower due to the additional overhead of runtime interpretation. For further details, please refer to the graphs in the "(NS) PYTHON Quicksort Timings" sheet within our Excel file.

II. Conclusion

We observed that the first and last pivot strategies performed the worst overall on sorted and reverse-sorted arrays which is consistent with our prediction that they would lead to worst-case time complexity of ($O(n^2)$). The data suggests that this is likely the case, especially for larger arrays. In contrast, middle and random pivot strategies for both sorted and reverse-sorted arrays suggest a more balanced partitioning and therefore tend to exhibit average time complexity ($O(n \log n)$). For random (unsorted) arrays of all sizes, all four pivot strategies appear to execute within average time ($O(n \log n)$).

Curiously, the timing results for arrays of size 10 and 100 (see Figure 1 and Figure 2) are relatively close with only slight differences between pivot strategies. However, as shown in Figure 3, when we examine the results for the largest array (size 1000), there is a dramatic increase in execution time for first and last pivot strategies on sorted and reverse sorted arrays.

One potential explanation for the pronounced difference in first/last poor performance in larger arrays is the deeper recursion caused by highly imbalanced partitions. While our data suggests this is true in smaller arrays of the same type, the impact becomes more severe in larger arrays. Larger arrays are more likely to experience imbalanced partitions which might result in increased memory usage, more cache misses, and slower access times.

It's important to state that we cannot prove the time complexity for our quicksort algorithm. The time complexity is a theoretical measure that helps understand how an algorithm performs as the input size grows. The objective for this assignment was to collect empirical data to make informed observations about quicksort's time efficiency. The fact that all three of us gathered slightly different data suggests that our results depend somewhat on hardware and computational resources. However, given the inconsistencies we observed in clock speed, RAM and our averaged timings, we are unable to determine how hardware specifically influences our findings.

We suggest several ways to improve our empirical study:

- Uilize larger arrays (e.g., 10,000 elements).
- Include a greater variety of sorting types, (e.g., arrays that have duplicate values).
- Control for hardware specifications (e.g., using identical machines)
- Run our program on parallel processors (e.g., implement a parallel version on multi-core processors).

References

[1] M. A. Shaffer, *Data Structures and Algorithm Analysis in Java*, 3rd ed.[Online].Available: https://people.cs.vt.edu/~shaffer/Book/JAVA3elatest.pdf.

[2] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp.10–16,1962.[Online].Available: https://doi.org/10.1093/comjnl/5.1.10