Newman, Michael

Metropolitan State University of Denver

An Exploration of Time Complexity and Big O

Timing Quicksort

# Intro

Quicksort is an algorithm for quickly sorting an array of comparable elements. In this paper, varying implementations are compared and contrasted. Performance is measured by average time complexity over a large set of runs. First, the Hoare and Lomuto partition schemes are compared, then the effect of switching to insertion sort after a cutoff interval is examined, and lastly selecting the median of three values as the pivot is evaluated. A discussion of combinations and the overall performance of quicksort follows.

# Design

All code for this experiment was written in Java. It was executed using a simple, custom testing framework. Each method is encapsulated as a “contender” and a race is broken down into individual laps. The average run time, or time complexity, of each method across all laps is the chief analytical concern.

Every race was standardized to run 1,000 times on an array of size 10,000,000 with random values spanning 0 to 232-1, which is the positive integer space in Java. Values were generated using the Random module provided by the Java Standard Library. Tests were collected in a list and then run in parallel using the Java Collections API. This study was performed on a Mac running MacOS 10.15.2 with a 2.2 GHz 6-Core Intel Core i7 processor with 16 GB in RAM. Timing results will vary on different machines.

The majority of graphs and tables are based on output from R, a programming language suited for data analysis. The source code for all calculations and figures can be found in Appendix A. The raw data can be found in Appendix B. The source code for the testing framework can be found in Appendix C. Note, R automatically calculates outlier values and plots them differently in the box plots it generates.

## Pseudo Code

For 1000 times:

1. Generate random data.
2. Evaluate the contender.
   1. Copy data.
   2. Start timer.
   3. Sort.
   4. End timer.
   5. Assert array is actually sorted (reject entire race if not sorted).

## A review of Quicksort

Quicksort sorts an array of size N with time complexity of O(N log N) and sorts the array in place, meaning that it does not require extra space relative to N (Sedgewick & Wayne, 288). The algorithm involves picking a “pivot” and dividing the array by placing all the values lower than the pivot on one side and all the values higher than the pivot on the other.

Figure 0.1: Partitioning in Subarrays (Sedgwick & Wayne, 288)



Then, this process is repeated on each half of the array. By dividing the array into smaller and smaller subarrays, the algorithm recursively sorts the array.

Insertion sort is not discussed here as it is not a primary source of concern for this study and is considered basic knowledge within the field.

# Hoare VS Lomuto Partitioning

Partitioning is the ‘basic operation’ of the algorithm, the operation that is repeated the most and contributes the most to the runtime. Thus, the efficiency of the algorithm depends most on how partitioning is done in the array. There are two common partitioning algorithms, the Lomuto algorithm and the Hoare algorithm. In Lomuto’s scheme you initialize two pointers at the head of the array, scanning left to right so long as elements are larger than the pivot. When you find an item smaller, you swap it to the head of the array, slowly dividing the subarray into less than and greater than. Once you have completed this process for the subarray, you swap the pivot into place and return.

In Hoare’s scheme, you initialize two pointers at the head and the tail of the subarray. Then, you work right from the head until you find an item larger than the pivot. Similarly, you work left from the tail until you find an item smaller than the pivot. You swap these values, working inwards on the array until the two pointers meet, at which point you swap the pointer into position.

To isolate the differences between the contending algorithms, the contender code varies *only* by the partitioning scheme.

### Results

Figure 1.1

A screenshot of a social media post

Description automatically generated

The Lomuto partitioning scheme demonstrates a lower time complexity under these testing conditions. In fact, the average time complexity of the Hoare partition was slightly worse than the 3rd quartile of the Lomuto partition. With such a significant difference, the Lomuto partition was chosen as the partitioning scheme for the rest of the tests.

This is in contrast with other comparisons. In fact, Sedgwick and Wayne do not introduce the Lomuto partition for favor of the Hoare partition (288). One possible explanation for this discrepancy is the underlying operating system’s array caching strategy. The Hoare partition goes back and forth from high and low indexes while analyzing subarrays. If the start and end of the array are outside of the cached memory block, then data retrieval time add to the time complexity of the algorithm.

# Switching to Insertion sort

With each recursion, quicksort divides the array into more and more sorted arrays. Further, the best-case scenario of insertion sort occurs when the array is *almost* sorted. Thus, a common enhancement to quicksort is to use insertion’s sort best-case scenario as an advantage. The scheme defines a cutoff point after which recursion ceases. This produces an almost sorted array, which is sorted with a call to insertion sort. This dramatically reduces the time complexity of the algorithm as shown below.

There is no agreed upon cut off value which will produce the best result. Sedgwick and Wayne speculate that the value is in between 5-15 but note that the actual value will vary based on machine (296)(Gordon). Interestingly, the Java Standard Library’s DualPiviotQuicksort class defines a cut-off value to Insertion sort at 47 (see line 67 of the class). To determine an appropriate cut-off point, a value range was estimated and then reduced until an approximate value was found.

## Spread 1

Figure 2.1

**A screenshot of a cell phone

Description automatically generated**

This spread was chosen to spread the likely span of values which could lend themselves to substantial improvements of the original method. As demonstrated above, a substantial improvement was seen in the range of 25-100, with a size of 125 producing results similar to not having a cut-off. In fact, partitioning at 50 had a mean runtime of ~2.8e9th nanoseconds, a 19.54% improvement over not partitioning at all (~3.5e9th nanoseconds).

The next range was chosen as 30 to 90 to further narrow down an appropriate partition size.

## Spread 2

Figure 2.2

**A screenshot of a social media post

Description automatically generated**

At this spread, 50 once again emerges as a clear choice for a cut off value. However, upon retrospection, this data set was rejected due to the high time complexities across the board under this sample. This run, labeled as Part2B bellow, demonstrates a dramatic deviation in time complexity.

Figure 2.3

A screenshot of a cell phone

Description automatically generated

To maintain the validity of this study, the same test was run again.

Figure 2.4

A screenshot of a cell phone

Description automatically generated

With a new set of random input data, the algorithm performed within expectations (See figure 2.5 below). Cut off values of 50 and 70 emerged as the lowest average time complexity, taking ~ 2.3e9th nanoseconds.

Table 2.1

**Sample Mean @ 50**: 2,360,682,243 ns

**Sample Mean @ 70:** 2,368,604,084 ns

**Absolute Difference**: 7,921,841 ns

**Relative Distance:** 0.33 %

Taken as a single value, this is an improvement of ~32.06% over the Lomuto partition as measured in spread 1. The next spread was chosen to due to the statistical equivalence of the cut off values of 50 and 70.

#### Sources of Error

Figure 2.5

A screenshot of a cell phone

Description automatically generated

The original dataset (Part2B) remains a chief possible source of error in this project. There are many possible causes, such as CPU congestion during the run, the testing framework code, and the randomness of the test data produced. However, the original race remains included in the final dataset of this paper. Upon examination of the original, deviant dataset, we still observe a trend leading us to the values surrounding 50 and 70. This trend is further observed in the re-run as shown in Figure 2.6. Thus, the discrepancies in time are likely due to external issues and do not correlate to a difference in time complexity.

Figure 2.6

A close up of a map

Description automatically generated A screenshot of a cell phone

Description automatically generated

## Spread 3

Figure 2.7

**A screenshot of a cell phone

Description automatically generated**

In this spread, the lowest average time complexity was at cut off value 60 with a time of 2.6e9th nanoseconds. This represents a 23.94% improvement over the Lomuto partition as measured in spread 1. While this is less of an improvement than the previous partition, the trend points towards a partition size of 60.

Figure 2.8

A close up of a map

Description automatically generated

## Spread 4

Figure 2.9

## **A screenshot of a cell phone Description automatically generated**

In this spread, the same trend is observed. A cut off value of 59 had the lowest time complexity taking ~ 3.05e9th nanoseconds, a 12.47% improvement over Lomuto from part 1. We again observe that this is *less* of an improvement than the last increment. However, over the dataset tested in spread 4, the winner is clear.

Figure 2.10

A close up of a map

Description automatically generated

## Spread 5

Figure 2.11

A screenshot of a social media post

Description automatically generated

In spread 5, we observe the same trend from the earlier spreads. 59 has emerged as the statistical winner with a mean run time of 2.47e9th nanoseconds, a 28.79% increase over Lomuto as measured in spread 1.

## Final Comparison

As a final comparison, cutting off recursion at 59 and never cutting off (Lomuto) was compared in one run.

Figure 2.12

A screenshot of a social media post

Description automatically generated

At the cut off value of 59, an average time complexity of 3.29e9th nanoseconds was observed. Without a cut off value, an average time complexity of 3.65e9th nanoseconds was observed. This represents a 9.68% decrease in time complexity, far less than expected based on the previous runs.

## Discussion

Throughout these spreads to analyze a cut off value, contradictory information arose. In specific, cut off values of the same size produced different results on different races. Further, the decrease in time complexity changed frequently from spread to spread. As the cut off value was homed in on, it was expected that each successive spread would increase the efficiency and outperform Lomuto more with each increment. While there was always *some* decrease in time complexity, it was not uniform. A deeper analysis of the causes of these discrepancies warrants further testing but is beyond the scope of this paper.

Figure 2.13

A screenshot of a cell phone

Description automatically generated

Despite this, we consistently observe a decrease in average time complexity when the algorithm ceases recursion after a cut off. In fact, the lowest average time complexity observed for the Lomuto partition was still higher than the worst average time complexity observed for the cut off scheme.

Figure 2.13

A screenshot of a social media post

Description automatically generated

Table 2.2

|  |  |
| --- | --- |
| **Lomuto** | 3,460,901,971 nanoseconds |
| **Cut off at 59** | 2,936,672,543 nanoseconds |
| **Actual Distance** | 524,229,428 nanoseconds |
| **Relative Distance** | 15.15 % |

Further, when we aggregate all runs of Lomuto and cutting off at 59, a decisive win is observed. Ameliorating the outlier data across a larger spread produces a notable 15 % decrease in time complexity. Thus, this dataset was concluded to be suitable to observe and infer trends in average time complexity.

# Median of Three

A typical worst-case scenario for quicksort happens when the pivot chosen causes no elements to be swapped. This happens when the pivot is the highest indexed element in the array (assuming that the highest index is chosen as the pivot). As there are no values lower than the pivot, the partition does not divide the array into a smaller section and performs no swaps. Instead of simply choosing the highest index, choosing the median of three values will help ameliorate this case. The primary disadvantage of this variation is the overhead of sorting an extra 3 values.

Figure 3.1

A screenshot of a social media post

Description automatically generated

This disadvantage becomes so detrimental that instead of preventing worst-case scenarios, it increases average time complexity. When there is no recursion cut-off value, a performance loss of 2.45 % is observed from Lomuto to a median of 3 partition. This difference is small enough to have arisen from outside sources. However, it suggests that choosing the median of three is not enough to increase the performance of quicksort substantially *by itself.*

# Best Shot: Median of Three and Cut Off at 59

In order to increase the performance of choosing a median of three, the partition scheme was combined with a cut off to insertion sort. The previously determined value of 59 was used in a best-effort attempt to achieve better results.

Figure 4.1

A screenshot of a social media post

Description automatically generated

These observations realize the expected gain in performance from the previous section. Choosing the median of three in conjunction with a cut off to insertion sort produced an 8.53% decrease in time complexity. This combination balances the removal of a possible worst case with the cost of checking 3 extra values.

# Race 5: Best Shot VS java.util.Arrays.sort

As a final test, the best shot algorithm was compared against the Dual Pivot Quicksort implementation in the Java Standard Library, java.util.Arrays.sort.

Figure 5.1

A screenshot of a social media post

Description automatically generated

The library implementation sorted the array in 2.93e9th nanoseconds, a 6.93 % increase in performance from the implementation used here. While this paper’s intent is to understand and analyze possible improvements to quicksort, it is not intended to be comprehensive enough to compete with the Java Standard Library contributors. It is interesting to note that with basic improvements to the standard algorithm, we are able to come incredibly close to the library’s time complexity.

# Discussion

It is clear that there are substantial improvements that can be made on the traditional quicksort algorithm. However, the possible sources of error and future study deserve to be expanded on.

## Sources of Error

This study must be understood with the context of its many sources of possible error. Timing algorithms run time is inherently tricky as clock time does not match the actual time the algorithm was running on the CPU. Thus, CPU congestion can and almost certainly did skew the data. However, this skew was likely ameliorated across the 1000 runs performed for each sort.

The results and data contained within this study should be used for understanding trends in performance, not calculating the expected runtime of any quicksort implementation.

## Future Study

Analysis of the Arrays.sort source code provides many clues on what other improvements can be made to the algorithm. The library uses a dual pivot partitioning scheme, uses merge sort at higher levels, and changes comparing strategies based on the array type. Future studies could use the source code for this study and make improvements until a statistical tie was reached with the library implementation.

Further study of the composition of the random data used to sort would aid in understanding quicksort’s time complexity as well. As stated earlier, the worst case for quicksort happens when it the sub array to partition is entirely less than the pivot, meaning no swaps occur. Future studies should seek to produce data where this is highly likely to occur. Comparative analysis, using this study as a baseline, would provide a deeper understanding of the possible improvements discussed here. Of particular interest would be the effect an almost-sorted data set would have on the median of three strategy. As this strategy is designed to protect against this worse case, its actual protection would be of great interest.

Lastly, the runtime of an algorithm is heavily impacted by the environment that it is run on. Future studies analyzing this algorithm running on different machines would provide a more complete picture of the time complexity of quicksort.

# Conclusion

Several average runtimes of various quicksort implementations were compared, looking for decreases in average time complexity. First, the Lomuto partition sorted the array quicker on average than the Hoare partition, so it was used as the partition scheme for the rest of the tests. Second, the effect of cutting off to insertion sort after a cut off value was examined. A value of 59 was found to be the most effective in reducing time complexity. Then, the effect of choosing a partition from the median of three values was examined. This did not reduce time complexity by itself but did reduce run time when combined with a cut off to insertion sort. Overall, several improvements were made to quicksort to decrease the algorithm’s average time complexity. Further study should focus on delineating best- and worst-case scenarios and seek to understand the effects of data distribution on sort time.