Lillian Garfinkel, Megan Letendre, Jacob van Steyn

Professor Heineman

CS2223 Algorithms Section D20-D01

13 May, 2020

CS2223 D20 Quicksort - Canthon Final Project

Background

The QuickSort algorithm is popular among programmers because of its simplicity and speed. It sorts in-place, meaning it only uses a small amount of memory, and it is considerably fast compared to other sorting algorithms, with a time complexity proportional to O(N2), for an array of length N. Quicksort works by dividing the array to be sorted into two sub arrays. The partition, or place where the division of the array takes place, is determined based on type of QuickSort implementation. The sub arrays are then sorted recursively. Once the two sub arrays have been sorted, the whole array has been sorted.

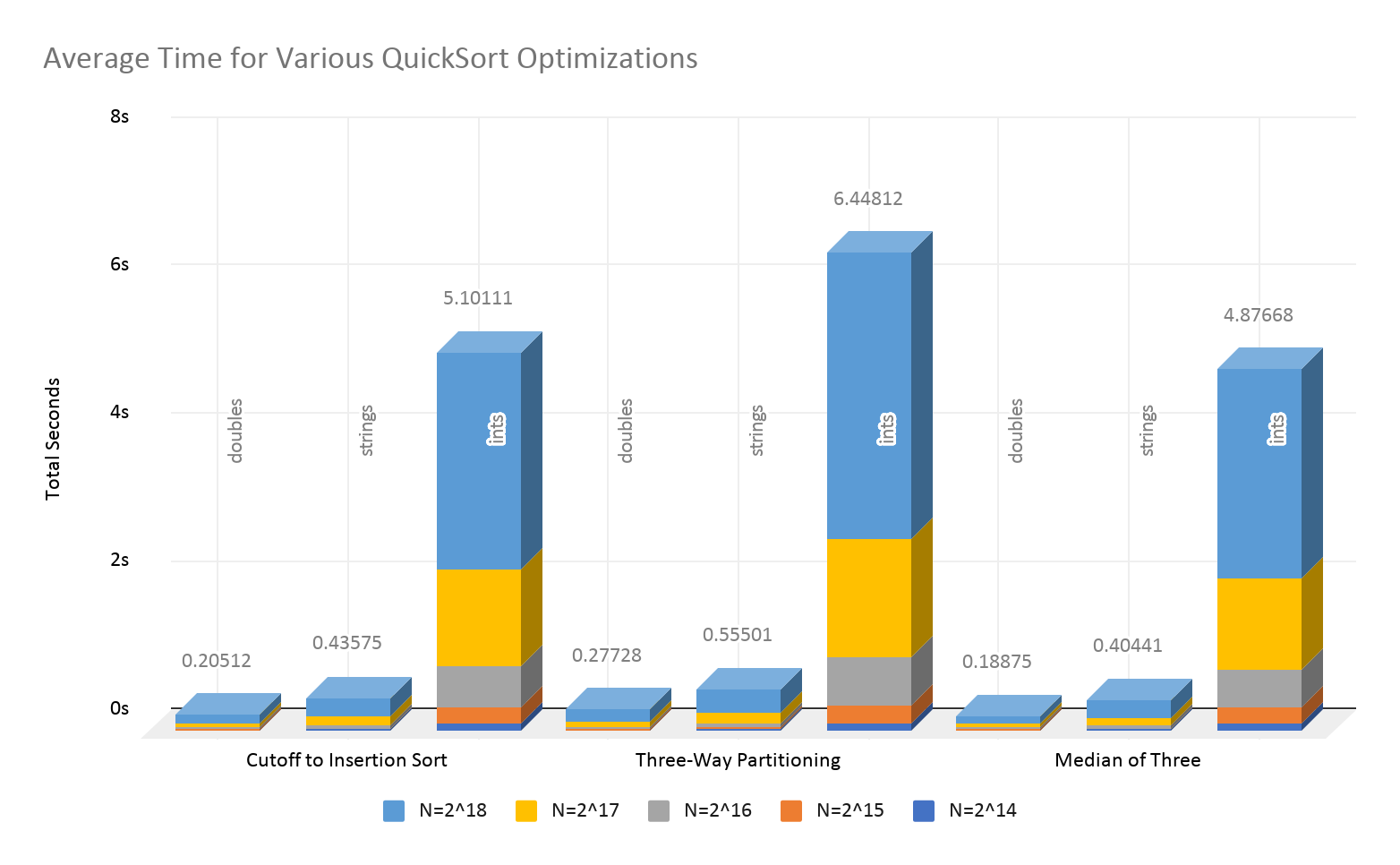
The question of where to partition the array that’s being sorted is a point of contention amongst programmers, as changing the location can have a big effect on efficiency of the algorithm. Due to this, QuickSort has many variations that are useful for different situations. The three that are the focus of this report are cutoff to insertion sort, median-of-three partitioning, and three way partitioning. Each has their own advantages and disadvantages. Cutoff to insertion sort works by determining a cutoff point for the size of the array, at which the sorting algorithm switches from QuickSort to Insertion Sort. . In using median-of-three partitioning, the median value of the first, middle, and last elements of the array is used as the partition point. This is optimal because it creates a more accurate partition versus choosing the first element. Three way partitioning consists of partitioning the array into three sub arrays instead of two. All three of these variations have a runtime of O(N2). All three of these variations are useful for different situations and can aid in the development of an efficient program, when implemented properly.

Q1. Timing Improvements For Each Variation

Each variation of QuickSort, the insertion sort method, median-of-three partitioning, and three way partitioning, are more effective depending on which data type they are performing on. The data sets these three variations were tested on are described as follows:

* **Doubles:** N uniformly random doubles.
* **Strings**: N strings from the English dictionary.
* **Integers:** N integers from StdRandom.uniform(N/k) for k in 2, 3, 4,...16. The invocation to uniform(p) returns a uniformly distributed random integer from 0 up to p-1.

With each different variation of QuickSort, we ran 5 trials with array sizes ranging from 214 to 218, and for each N, we further ran 100 trials for each data type – to see how the different optimizations affected the runtimes for each variation. This is how our runtimes changed depending on the optimization:



For each separate variation, certain trends remained the same concerning the runtime of each of these dataset. First, for each of the three variations of the QuickSort method, the double data type had the lowest runtime whereas the integer dataset had the highest run-time, the Median of Three variation was the most efficient (lowest runtime) for all three data types and the Three-Way Partitioning variation was the least efficient (highest runtime).

Q2. Results Sorting Data Set 3

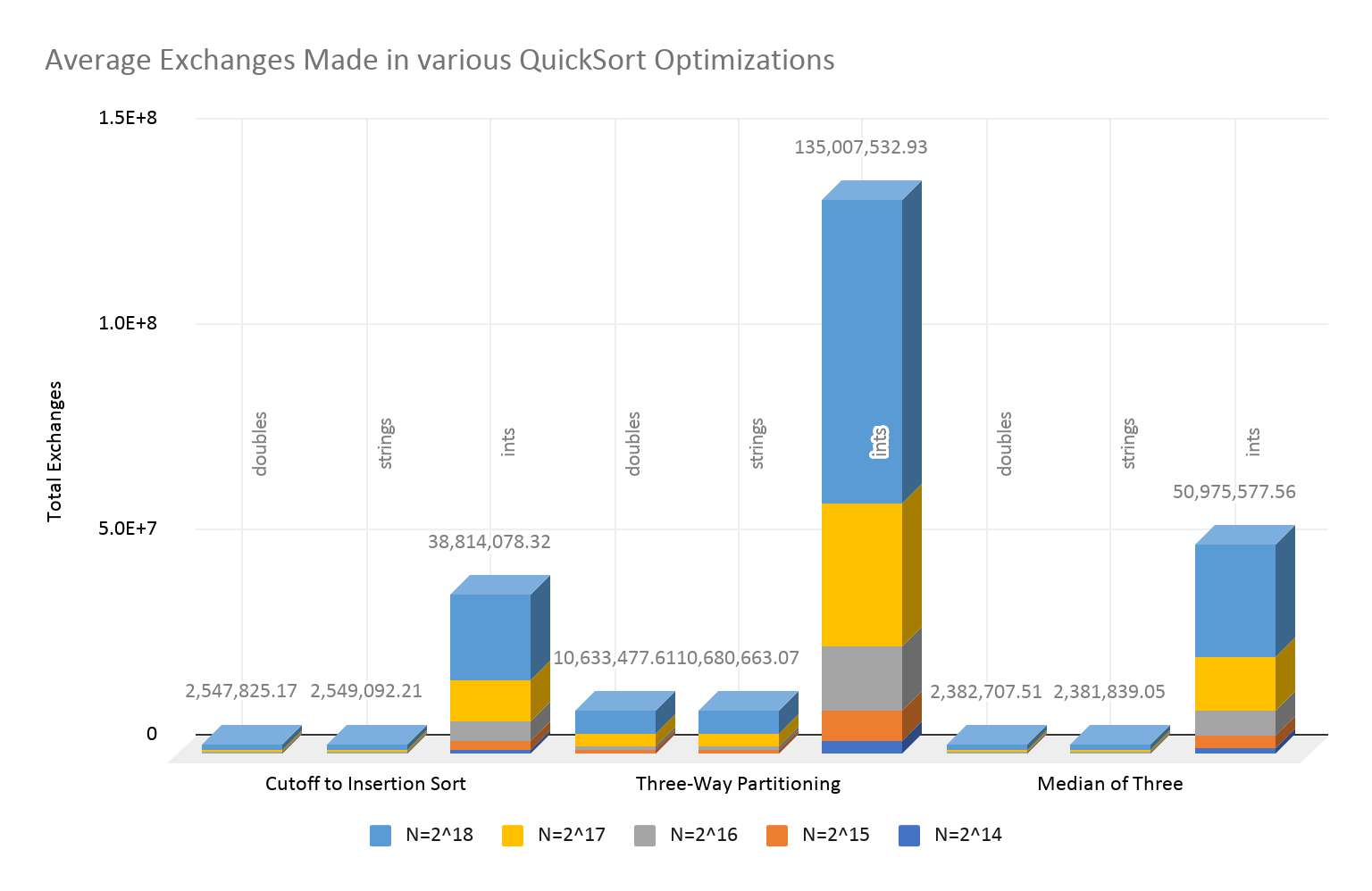
For all three QuickSort variations, sorting the set of integers generated with StdRandom.uniform(N/k) took the most time, the most compares and the most exchanges. This data set is always significantly higher than the other two due to the amount of values within the set itself. Within this specific data set, there are N\*15 values whereas the other two data sets, doubles and strings, have far fewer values. This dataset is aimed at showing the inefficiencies or efficiencies in sorting lists with many repeated values.

Q3. Report on Impact on Number of Exchanges and Comparisons For Each Variation

Each variation of QuickSort, the insertion sort method, median-of-three partitioning, and three way partitioning, cause a difference in the number of compares and exchanges depending on which data type it is performing on. The data sets these three variations were run on are described as follows:

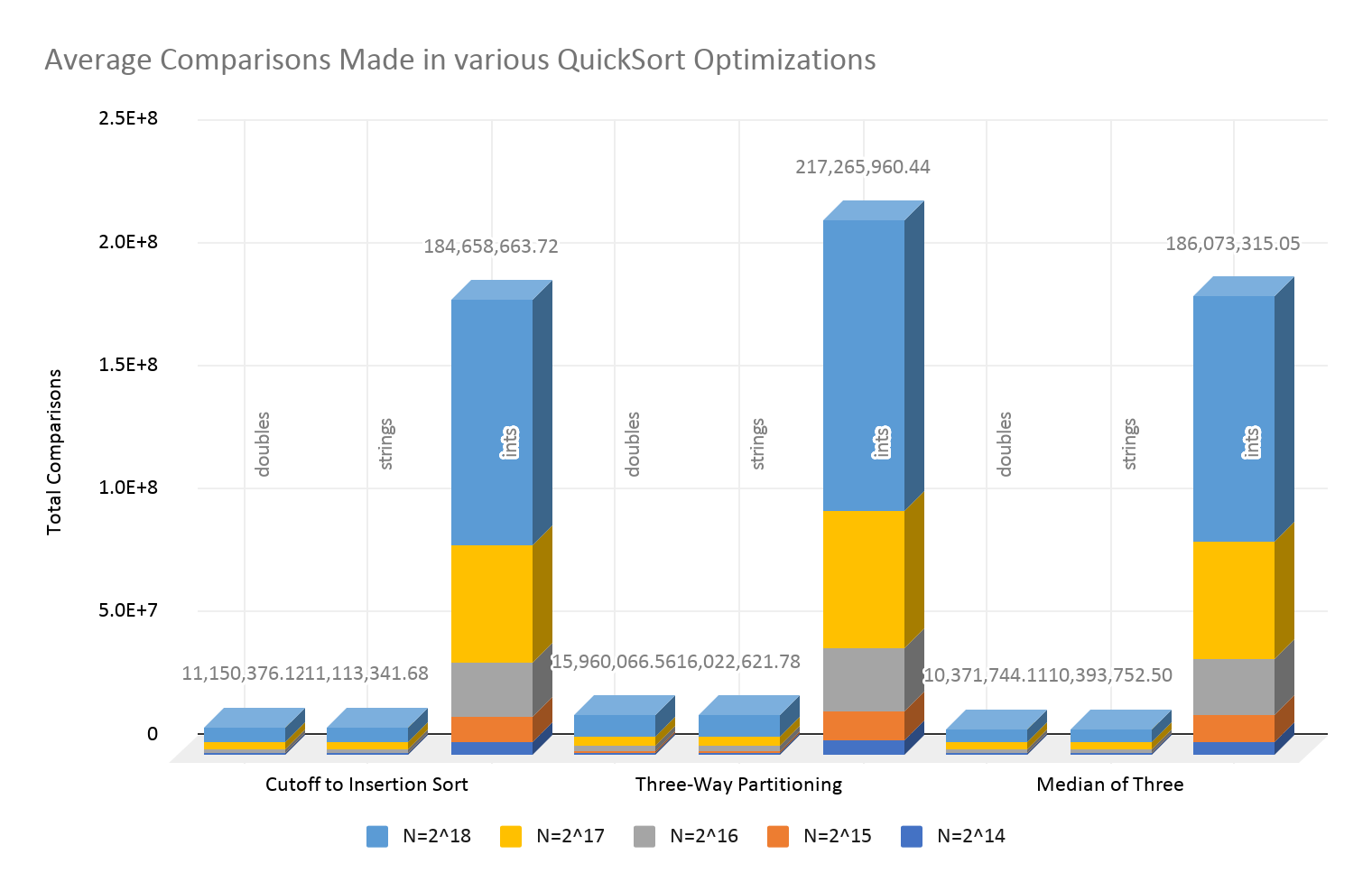
* **Doubles:** N uniformly random doubles.
* **Strings**: N strings from the English dictionary.
* **Integers:** N integers from StdRandom.uniform(N/k) for k in 2, 3, 4,...16. The invocation to uniform(p) returns a uniformly distributed random integer from 0 up to p-1.

With each different variation of QuickSort, we ran 100 trials with each data type to see how the different data types affected the runtimes for each variation. This is how the number of exchanges changed depending on the data type:



For each separate variation, certain trends remained the same concerning the number of exchanges for each of these data types. Each variation had resulted in a different data type having the fewest number of exchanges. For the Cutoff to Insertion Sort variation, the fewest amount of exchanges occurred for the strings data type whereas for the Three-Way Partitioning variation and the Median of Three variation, the fewest amount of exchanges occurred for the doubles data type. The difference in the number of exchanges between the doubles and strings data types was not a large number when compared to the total number of exchanges. For all three variations, the integer data type had significantly more exchanges than the other two data types. Overall, it seems as though the Median of Three variation had the fewest number of exchanges among the doubles and strings data types however the Cutoff Insertion variation had significantly fewer exchanges within the integer data type.

With each different variation of QuickSort, we ran 100 trials with each data type to see how the different data types affected the runtimes for each variation. This is how the number of comparisons changed:



For each separate variation, certain trends remained the same concerning the number of comparisons for each of these data types. Each variation had resulted in a different data type having the fewest number of comparisons. For the Cutoff to Insertion Sort variation and the Three-Way Partitioning variation, the fewest amount of comparisons occurred for the strings data type whereas for the Median of Three variation, the fewest amount of exchanges occurred for the doubles data type. The difference in the number of comparisons between the doubles and strings data types was not a large number when compared to the total number of comparisons. For all three variations, the integer data type had significantly more comparisons than the other two data types. Overall, it seems as though the Median of Three variation had the fewest number of comparisons among the doubles and strings data types however the Cutoff Insertion variation had significantly fewer comparisons within the integer data type, which is the same finding as with the exchanges.

Conclusion

The popularity of the QuickSort algorithm remains to this day due to its simplicity and speed. The number of different variations that can be applied to this algorithm work differently depending on the type of data that it is being implemented on. The three main datasets that were focused on in this analysis (doubles, strings, and integers) provided data that led toward some conclusions concerning these different variations.

For Example, if the input data contains many repeated values, as it would in the integer data set, the Insertion to Cutoff variation has a poor performance, including runtime, exchanges, and comparisons. In this case, using the Median of Three variation seems to work better. As addressed above, the integer data set always had a significantly longer runtime as well as a significantly larger amount of exchanges and comparisons. This is due to the integer data set being larger than the doubles and strings data set.

An optimization we did not test was to combine some of these variations. For example – combining Median of Three with Cutoff to Insertion sort could have improved the overall execution by adding the benefits of both variants together. There are also different variations of QuickSort that we did not get a chance to test due to the limited scope of this project.

Sorting is an essential tool for programs of a wide variety. With data becoming larger and larger, the efficiency of these algorithms is increasingly important. There’s no one algorithm that can do everything – which is why having a variety of options to choose from is important. Overall, the QuickSort algorithm will always be well-known and widely-used because of the convenience and efficiency it provides.

Works Cited

Hoare, C. A. R. Quicksort. *The Computer Journal*, Volume 5, Issue 1, 1962, Pages 10-16, …….<https://doi.org/10.1093/comjnl/5.1.10>

Sedgewick, Robert, and Kevin Wayne. *Algorithms.* Pearson Education, 2011.

Explore our Data

<https://docs.google.com/spreadsheets/d/1a_cVJwWMOKsLbLEcUgur9qgg4Iwui_vL9-Apm9ZpurY/edit?usp=sharing>

Codebase

<https://github.com/Jacobvs/Quicksort-Variations>

Video Report

<https://youtu.be/9lMkeN_o7LU>