Claire Liu, Renee Soika, and Yuqi Wang

October 10, 2020

CS 150-01

Project Report: Sorting Algorithms

1. Introduction

The purpose of this project is to study how the theoretical analysis of sorting algorithms compares with their actual performance. The sorting algorithms explored in this project are selection sort, bubble sort, insertion sort, merge sort, and three versions of quicksort based on the choice of the pivot element (first, random, and median of three). Additional goals include comparing the sorting algorithms with one another, comparing implementations of merge sort and quicksort with the Java standard library's implementations, and comparing the various versions of quicksort with one another. These goals will be accomplished by implementing the sorting algorithms, generating different arrays as test cases, running these test cases through the sorters, collecting the time that it takes for the algorithms to run on different sized and different types of arrays, and analyzing the data collected.

In bubble sort, the array is sorted in passes. In each pass, adjacent elements are compared and exchanged if out of order. Therefore, smaller values bubble up to the top of the array and larger values sink to the bottom. Each time, the algorithm checks to see if the array has been sorted and stops once it has been. The best case, which occurs when the array is already sorted, has a runtime of $O(n)$ because no elements are exchanged, but all the elements are checked. The average case, which is when the array elements are in random order, has a runtime of $O(n^2)$ because each of $O(n)$ elements must bubble across the other $O(n)$ elements. The worst case, which occurs when the array is in reverse order, has a runtime of $O(n^2)$ because every element has to bubble across the whole array. The advantages of bubble sort are that it performs well when the array is already sorted, and it has stable sorting, so the original order will be preserved when there are duplicates. However, bubble sort performs poorly when the array is in reverse order or is large because many elements have to bubble across the array (Xia 2020b).

Selection sort also sorts arrays in passes. In each pass, the next smallest element is selected and placed at the current position. Regardless of the order of the elements in the array, selection sort iterators through the `n` elements in the array `n` times because each pass only sorts one element. Therefore, the best case, average case, and worst case theoretical runtimes are all $O(n^2)$. The advantages of selection sort is that it does not require a lot of extra space in memory—no new arrays are created. However, selection sort always iterates over the array $O(n^2)$ times regardless of its original order. It is also an unstable sorting algorithm, so elements might not appear in the sorted array in the same order as they did in the original array if there are duplicates. Since it has quadratic runtime, selection sort performs poorly for large lists (Xia 2020b).

Insertion sort iterates over the whole array and places each element in the correct position in the sorted portion at the start of the array. It shifts elements down to make room for the next element in the sorted portion if necessary. The best case, which occurs when the array is already sorted, has a runtime of $O(n)$ because no elements are moved, but the algorithm has to check if they are in sorted order. The average case, when the array is half sorted, has a runtime of $O(n^2)$ because $O(n)$ elements have to be shifted for $O(n)$ elements that are inserted. The worst case, when the array is in reverse order, has a runtime of $O(n^2)$ because elements are shifted as all of them are inserted. The advantages of insertion sort are that there is no need for three assignments to sort an element (as are needed in bubble sort and insertion sort), additional memory space required is negligible, and it maintains stable sorting (Xia 2020b).

Merge sort is a divide-and-conquer algorithm in which the array is split in half so that the left half and the right half are sorted recursively. Then the two sorted halves are merged. Merge sort divides and visits every element regardless of the order of the data set. Therefore, the best, average, and worst cases all have $O(nlg(n))$ runtimes. The advantages are that it is a stable sorting algorithm and the guaranteed $O(nlg(n))$ runtime makes it very fast for large datasets as compared to the quadratic sorting algorithms. The disadvantages are that it is slower for smaller data sets and already sorted sets because it goes through all the steps of the algorithm, regardless of initial state, and it uses extra storage space ($O(n)$) in the merge step (Xia 2020b).

Quicksort is a divide-and-conquer algorithm in which a pivot is chosen. The pivot is typically the first element in the array, a random element in the array, or the median of three (the

median value out of the first, last, and middle elements). Then the first and last variables hold indexes marking the endpoints of the region to sort. While first is less than last, a pivot is chosen, then quicksort is applied recursively to sort the part of the array to the left of the pivot and the part of the array to the right of the pivot. The best case, which occurs when the pivot chosen has the median value resulting in balanced partitions, and the average case, which occurs when the pivot results in roughly balanced partitions, both have theoretical runtimes of $O(nlg(n))$. The worst case is when the pivot chosen has the largest or smallest value in the array, resulting in an unbalanced partition and a runtime of $O(n^2)$. Therefore, the selection of the pivot is important. If the first element is the pivot, a major disadvantage is that if the array is sorted or near sorted, quicksort will have worst case runtime because the partition will be uneven. The advantage of choosing the first is that it is simple, requiring no extra swaps. If a random element is the pivot, in most cases a good pivot will be chosen, but randomness requires the program to generate a random number with a generator and does not guarantee a desirable pivot. If the median of three is chosen, there will be good performance in most cases, but the partition will be uneven if the median of three is lower or higher than most of the data. Generally, the advantage of quicksort is that it is fast and requires minimal additional storage space, and the disadvantages are that it is not stable, and if a bad pivot is chosen, it will result in quadratic runtime (Xia 2020b).

2. Approach

Design Architecture

Each sorting algorithm is implemented as a separate class with a generic sorting method (BubbleSorter.java, SelectionSorter.java, InsertionSorter.java, MergeSorter.java) that implements a common interface called Sorter.java. For quicksort, the common components (partitioning and sorting) are implemented in an abstract class called QuickSorter.java, then the three different versions of pivot selection are implemented as concrete classes, QuickSorterFirst.java, QuickSorterRandom.java, and QuickSorterMedian.java, that extend QuickSorter.java. QuickSorter.java also implements Sorter so that it must uphold the contract to have a generic sorting method, making it interchangeable with other sorts, thus simplifying the ExperimentController. Each of these classes was unit tested to ensure proper functionality.

The class IntArrayGenerator.java is where the methods for generating the types of test arrays were, and these methods generated arrays of Integer objects. It encapsulates array

generation so that ExperimentController and ArrayWriter are focused on reading and writing data, respectively. This class was also unit tested.

Working in tandem with IntArrayGenerator.java is ArrayWriter.java. ArrayWriter.java uses the methods from IntArrayGenerator.java and writes arrays to a file called “arrays.txt.” This is where the number of elements to be sorted and the number of trials for each number of elements to be sorted are defined. The resultant “arrays.txt” is organized by type of array, and each type of array is printed ten times, one for each trial. Printing one array for each trial allows ExperimentController to read from the file continuously and ensures that each input array is different so that the random, duplicate, and 90% sorted test cases are more representative of all random sets, not one set. Printing duplicates of the ascending and descending arrays costs more space, especially for large numbers of elements, but the additional size on the order of megabytes was not a problem, and it did not require significantly more time.

In the class ExperimentController.java, a Scanner reads the “arrays.txt” file, copies numbers to an array, and calls and times the single inputted sorting algorithm. ExperimentController is where the number of elements to sort is defined and the total number of trials for each sorting algorithm is defined, which are used in ArrayWriter. ExperimentController also has the Java implementation of the Arrays.sort() method, which runs a Java standard library implementation of merge sort when an array of Objects is sorted and quicksort when an array of primitives are sorted. Since the Java implementation of merge sort uses objects, it is contained in an anonymous Sorter class so that it can share its Integer[]-based experiment code with the other Sorters. The Java implementation of quicksort uses primitives, so a separate version of the experiment code had to be adapted to primitive integer arrays. Since we are comparing the different algorithms, they cannot be run on the same run to avoid caching and other effects. Therefore, the user must manually change the algorithm to run in the parameter of the run method. A file called “time-analysis-<sorting algorithm>.txt” is outputted for each sorting algorithm that is run with columns for the trial name, number of elements, and the average runtime of all trials for that test case.

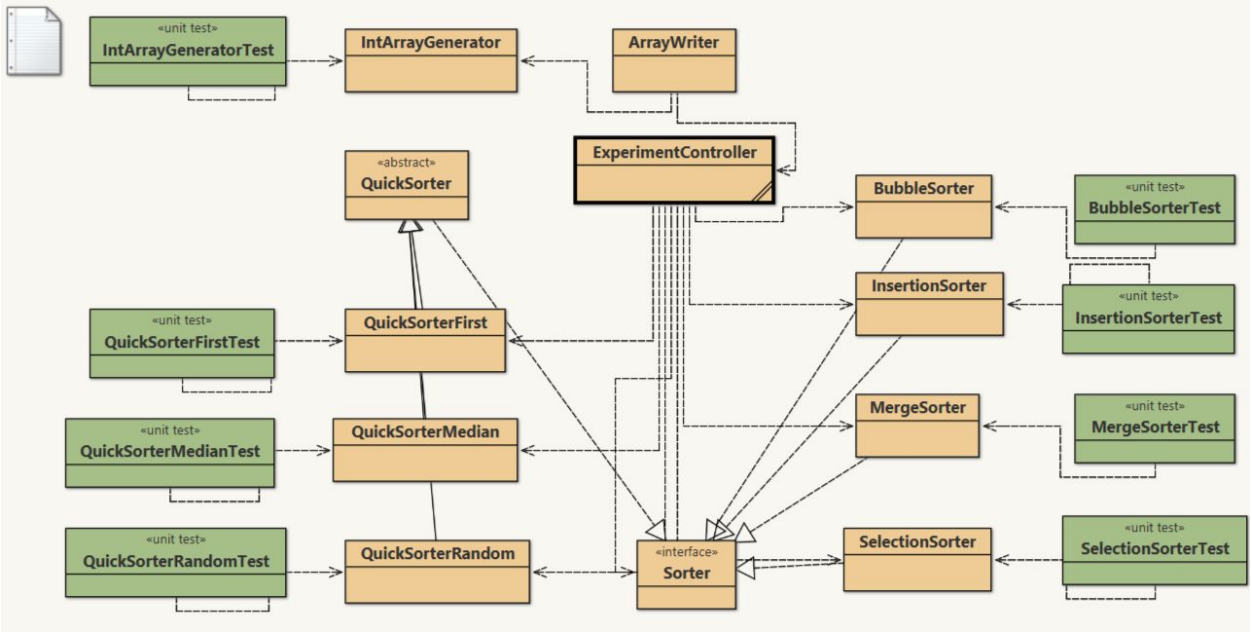

Figure 1: The dependency map of our program.

Design Analysis

We chose to create arrays of Integers to run the algorithms on. Since we knew what the size of each array was going to be and this size would not change over the course of the experiment, there was no need for the arrays to be dynamic ArrayLists. We chose Integers over other data types because they are the easiest to generate using a random number generator, and for the purposes of this project, there was no need to use strings or other data types.

The arrays that we generated included ascending, descending, random, 90% sorted, and many duplicates. Ascending and descending were generated with simple `for` loops with increasing and decreasing counters, respectively. The random test case was generated using an instance of the Random class and the nextInt() method. 90% sorted was generated using a combination of ascending and random—each integer in the array had a 10% chance to be random and a 90% chance to be in ascending order. Many duplicates were generated by using random numbers from 0-50 inclusive, which generates many duplicates as we were generating arrays with numbers of values in the thousands to millions.

The ascending array would be useful in exposing the advantages of algorithms where the initial state of the sorting algorithm would make a difference in the runtime. These algorithms include bubble sort and insertion sort, where minimal work is done if the array is already sorted.

Selection sort and merge sort both do not consider the initial order of the array and go through the whole algorithm regardless, showing a slight disadvantage for these sorting algorithms. To some extent, quicksort does this too, but in an ascending array for quicksort median of three, the median value in the array was chosen every time, which is optimal for quicksort runtime. However, for quicksort where the first value is chosen, this would result in worst case runtime because the array would be broken into subarrays of size 1 and n-1 in every partition.

The descending array would expose the disadvantages of algorithms where the initial state of the sorting algorithm would make a difference in the runtime, like bubble sort and insertion sort, because maximum work has to be done to reverse an array descending order, resulting in worst case runtimes. In comparison, algorithms like selection sort and merge sort that do not consider the initial order of the array would have a slight advantage in that they would be largely unaffected by the reverse order. Again, for quicksort, in the median of three, the median would be chosen every single time, which is optimal. However, for quicksort where the first value is chosen, this would result in worst case runtime because the array would be broken into subarrays of size 1 and n-1 in every partition.

The random array would show the average runtimes for algorithms including bubble sort, insertion sort, and quicksort because there would be a fairly even mix of values in and out of the correct order. For merge sort and selection sort, initial order does not matter, and the random array test case would show this as well, since the runtimes would be comparable to other array test cases. It would also expose the importance of the pivot choice in quicksort. Runtimes for quicksort vary a lot based on the pivot choice. Random or first (which would be a random number in this case) generally would be a good choice to produce best/average case runtime, but in a situation where a value relatively close to the maximum or minimum was chosen as the pivot, the runtime would be closer to the worst case. In the random array case, the median of three choices would be the best because the algorithm would have three random numbers to choose from as the pivot, but in rare cases, where three low or three high numbers were chosen, the worst case runtime could still happen. Ultimately, the random array could show how quicksort's dependence on the pivot choice is sometimes a disadvantage.

The 90% sorted array would expose similar advantages and disadvantages as the ascending array. It will be useful in showing how much initial order affected the runtimes of different sorting algorithms. For the algorithms where initial order does matter, like bubble sort

and insertion sort, it could show how much tolerance for unsorted values an algorithm would have.

The duplicates array could expose the weaknesses of partitioning. For all three versions of quicksort, the duplicates of the pivot would not be evenly distributed between both partitions. As the partitions become more unequal, the runtime would approach the worst case, $O(n^2)$, runtime. However, the runtimes of the other sorting algorithms do not depend on even partitioning, so the duplicates case would be similar to the random case. The duplicates array could also expose a possible advantage in a stable sorting algorithm as perhaps stable sorting algorithms do less work because original order is maintained when there are duplicates, so elements do not have to be moved around as much.

The same arrays were used for different sorting algorithms that were run for the same number of elements, as all the arrays generated were in the "arrays.txt" file. That file was read in the ExperimentController for all the sorters, being overwritten only when the number of elements needed to change. This was intentional, as using the same arrays across the different sorting algorithms would ensure consistency and remove the variable of having different arrays each time, and the analysis of the runtimes could then focus on the algorithms themselves.

3. Methods

The machine we used to run the experiments was a laptop (Dell G7 with 16 GB RAM). The same laptop was used for the data collected to generate the graphs in this report. However, all the experiments were run on different laptops to ensure that the data we were getting from the main testing laptop was consistent with average findings. In total, all the experiments were run on three laptops, though only the data from the Dell was used (chosen arbitrarily).

The timing mechanism in ExperimentController used the current time in milliseconds from System.currentTimeMillis(). Then it simply found the difference between the time recorded when the sorting algorithm began sorting the array and the time recorded when the array was sorted. This way, only the actual sorting for the different sorting algorithms was timed. An almost-identical timing mechanism for primitives was also written because Sorter only took generics, and primitives cannot be used in it. That way, the Java standard library implementation of quicksort (Arrays.sort()) could be timed.

For every sorter inputted, we ran ten trials for every test case, which consisted of one type of array of at least five different lengths. We averaged the times for each of the ten trials to obtain a result for that type of array (random, ascending, descending, etc.) and number of elements. These averages were plotted on the graphs, grouped by the type of array. For most of the sorters we only tested arrays of five different lengths, with two exceptions. We ran arrays of ten different lengths for the Java Standard Library's quicksort because it was sorting elements extremely fast. We wanted to get good data for Figure 13 but comparable in the range of zero to a million elements in Figures 14 to 19 to compare with other sorting algorithms. Additionally, for the ascending and descending arrays in quicksort first, quicksort first would recursively produce partitions of length 1 and n - 1, leading to a stack overflow error for anything over 20,000 elements. So we ran arrays of five different lengths for all the test arrays, but then ran five additional different lengths for the random, 90% sorted, and duplicate test cases with many more elements. This allowed us to see that the duplicate case was slower than the other two, even if it was much faster than the ascending and descending cases.

In general, we decided the number of elements to sort based on how long the different sorting algorithms took. For quadratic sorting algorithms (bubble sort, selection sort, insertion sort), we captured runtimes ranging from 0 ms to around 5,000 ms. This window captured the quadratic shape. For most of the efficient sorting algorithms, we captured runtimes ranging from 0 ms to around 1,000 ms. For quicksort first though, we did increase runtimes to almost 30,000 to see the difference between 90% sorted, random, and duplicates, which we were not able to see in the 0 to 1,000 ms range. For the best cases of the efficient sorting algorithms, at times the runtimes were very low, not exceeding 100 ms, even when the number of items to input exceeded one million.

4. Data and Analysis

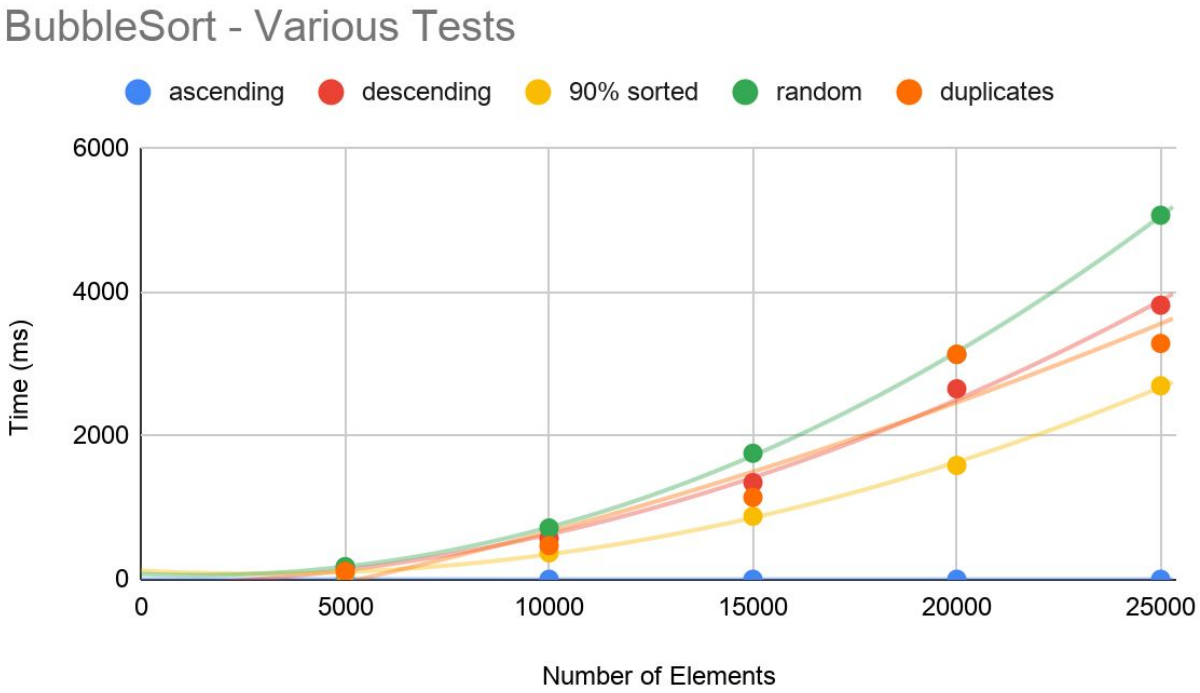

Figure 2: Times for bubble sort on different test arrays.

Figure 2 shows how the runtime of bubble sort is influenced by the original order of the array. Ascending is the best case, taking only $O(n)$ time because the algorithm just had to make one pass to see if the array was sorted. Then, 90% sorted was the next fastest, as many of the elements were already in the correct places; few needed to be moved across the array. Interestingly, descending and duplicates took around the same amount of average time and random took the most time, when it was expected that descending would be the worst case. This may be due to the fact that the random numbers had a greater range of numbers (including negative numbers) that had to be sorted whereas for descending the array values were limited to only the number of items in the array and duplicates was only limited to 0-50. The descending array also had a regular order, even if it was not correctly sorted. In other words, Java or the computer's processor could be performing optimizations when the integers are smaller or more predictable. However, the average and worst cases still ran in $O(n^2)$ time. These results agreed with the theoretical analysis in the introduction.

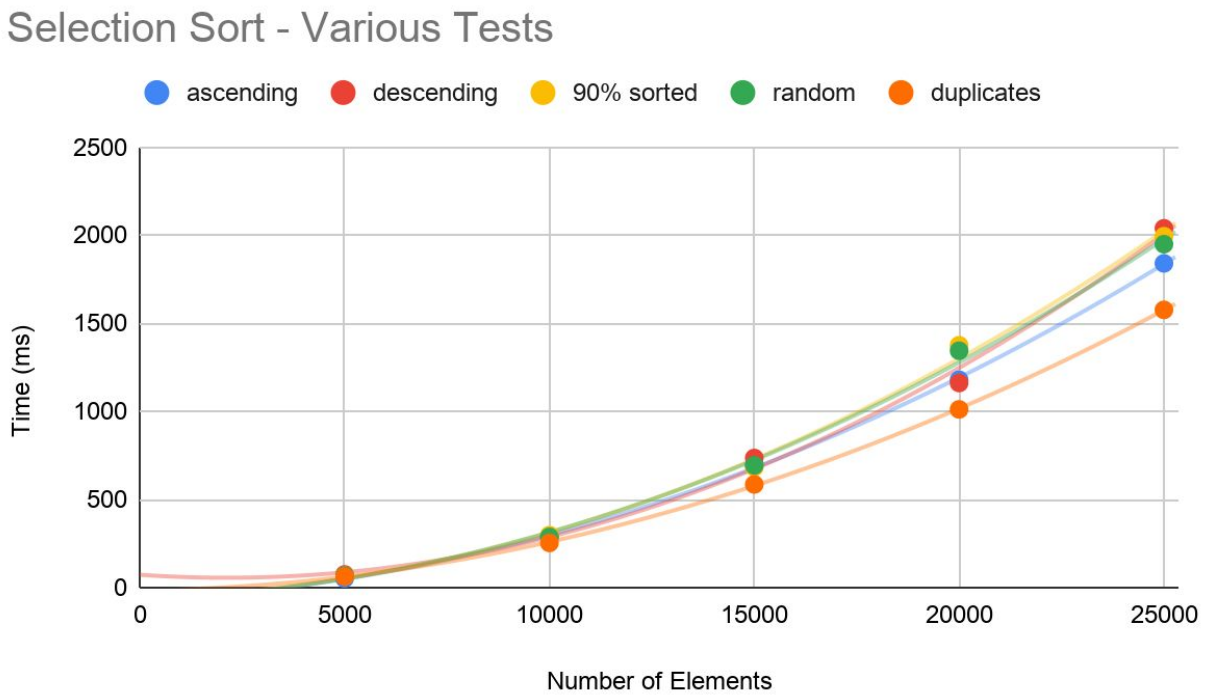

Figure 3: Times for selection sort on different test arrays.

Figure 3 shows how selection sort always searches for the minimum value ahead of every element regardless of the array's original order. All the various tests, ascending, descending, 90% sorted, random, and duplicates were all very close to one another in runtimes—all of their runtimes were $O(n^2)$ as expected. The duplicate test cases performed the best, though not by much. Possibly, Java could cache some of the comparisons between integers, which would make those repeated comparisons in an array with many duplicates faster than a normal comparison. The rest of the test cases were practically indistinguishable. These results agreed with the theoretical analysis in the introduction.

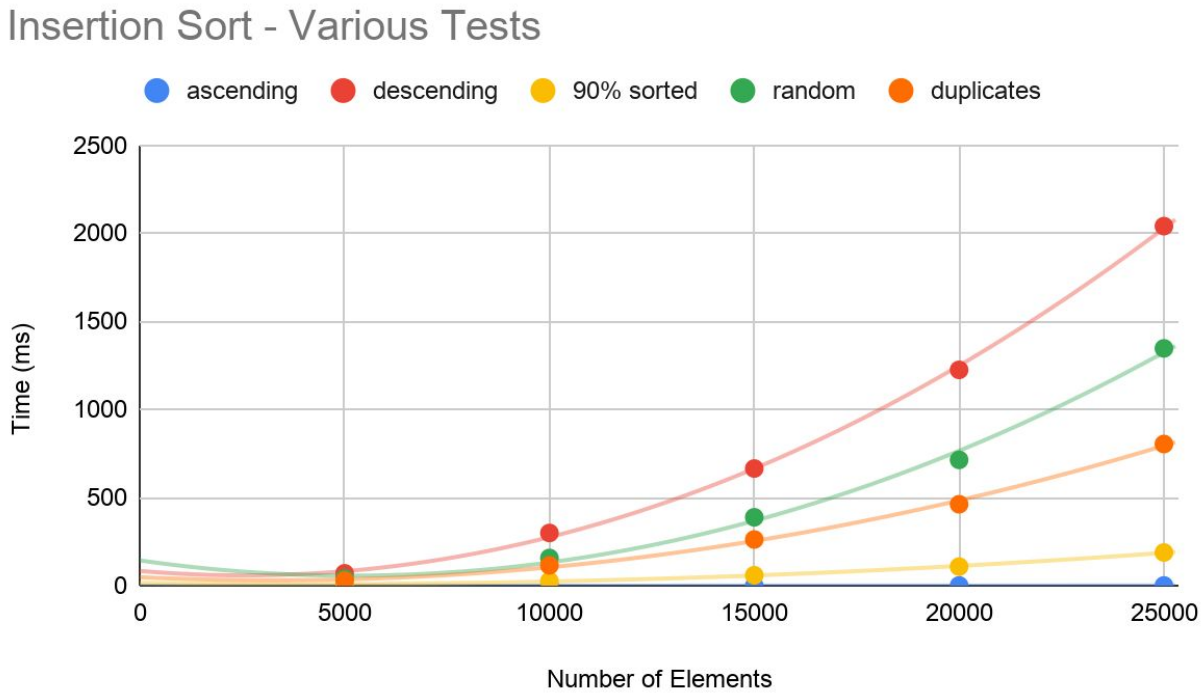

Figure 4: Times for insertion sort on various test arrays.

Figure 4 shows how the runtime of insertion sort is influenced by the original order of the array. In the best case of ascending order and the close-to-best case of 90% sorted, the data produced linear or almost-linear trendlines, confirming that the best case is $O(n)$ when insertion sort needs to move few to no elements. Duplicates and random took in-between/average amounts of time, confirming that the average case is $O(n^2)$ because $O(n)$ elements have to be shifted $O(n)$ times. Descending was the worst case, took the most time, and ran in $O(n^2)$ time as well. This was consistent with the descending case needing to shift the most elements every time a new one was inserted, even if big-O runtime was the same as the random and duplicate cases. These results agreed with the theoretical analysis in the introduction.

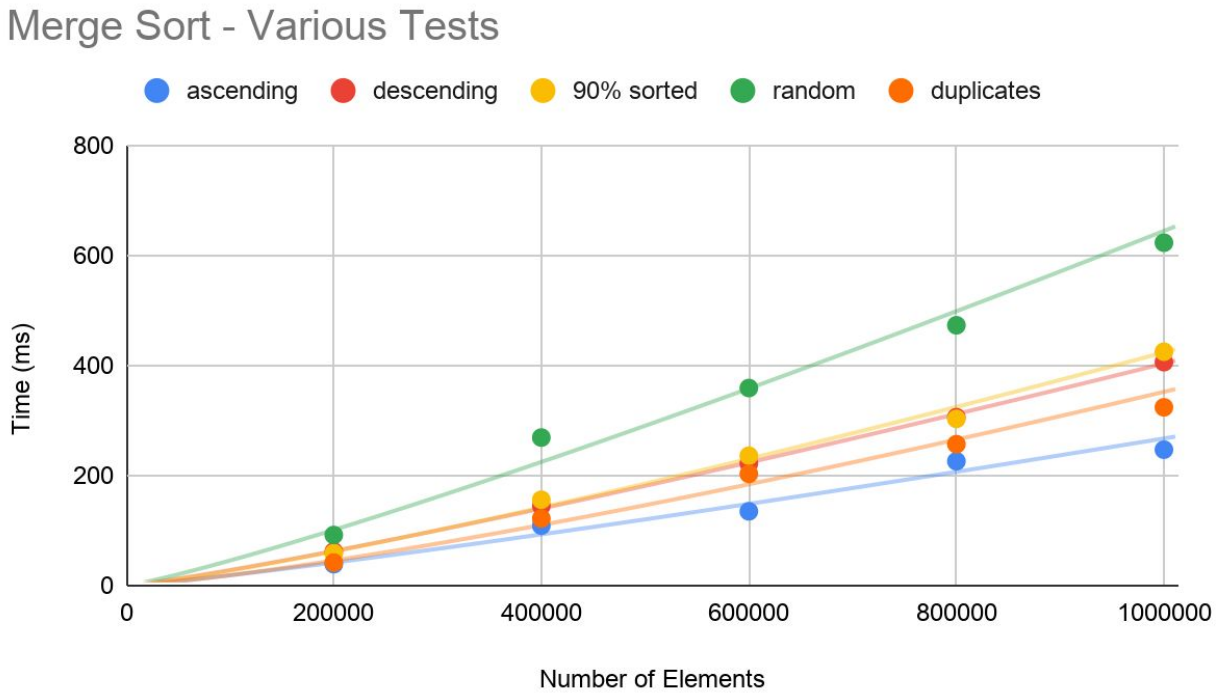

Figure 5: Times for merge sort on various test arrays.

Figure 5 shows that the best case, average case, and worst case for merge sort were all `O(nlg(n))`, which appears almost linear on a graph. Random was the worst case, while descending, 90% sorted, and duplicates were the average cases, and ascending was the best case. Random may have been the worst because random had the greatest range of numbers and the most unpredictable order. The ascending case was likely the fastest because none of the elements needed to change position. Likewise, the duplicates case would have had few elements that needed to be moved—with such a limited range of numbers, many of them would likely be in the correct position to begin with. Theoretically, all of these cases should have resulted in very close runtimes because, as stated in the introduction, the initial order should not affect merge sort, but in practice it does, slightly.

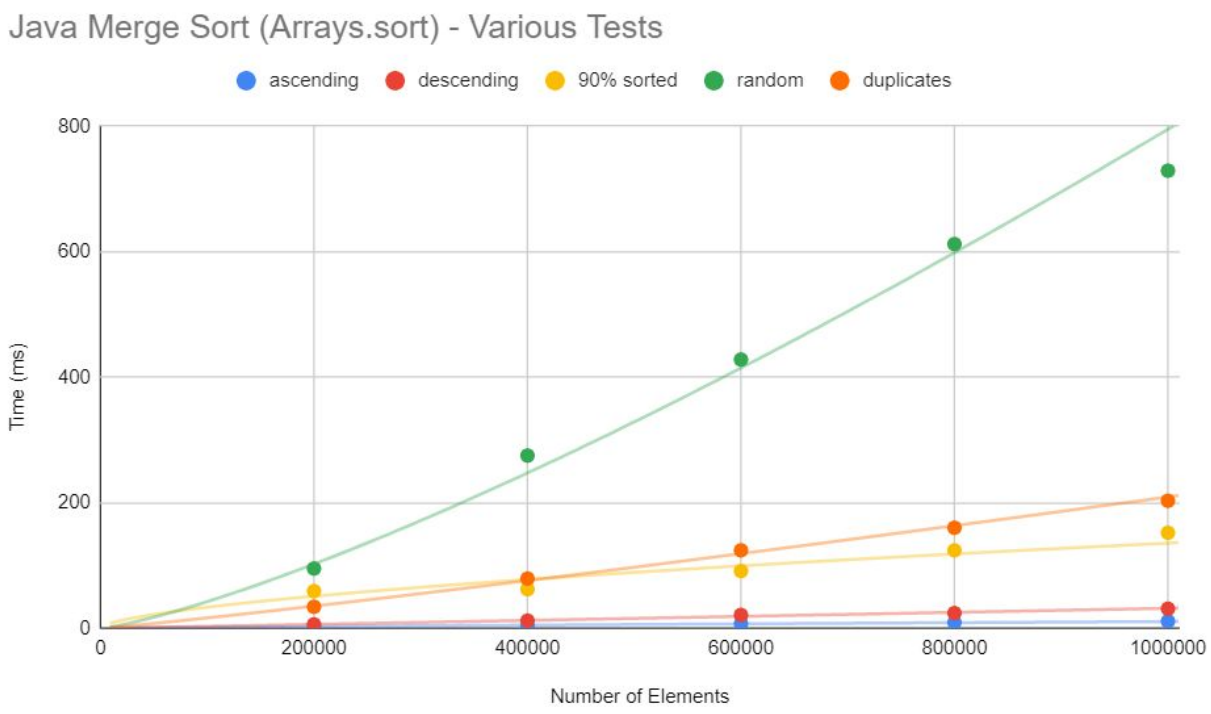

Figure 6: Times for Java Standard Library merge sort on various test arrays.

Figure 6 shows that the best case, average case, and worst case for Java's merge sort implementation were also `O(nlg(n))`. Random was again the worst case, like in our implementation, likely because of the values' unpredictable order that Java may not have been able to optimize. By the same token, the most predictable arrays, the ascending and descending cases, performed the best, just like our implementation of merge sort. The "slope" of the almost-linear curves appear about the same for the random case for both implementations, so they performed about the same for random arrays. For the other cases, the slopes of the Java merge sort curves appear slightly lower, which indicates that Java's version is likely optimized in minor ways compared to our version. These optimizations may be responsible for the lack of consistency that contrasts with the predicted consistency of merge sort in the theoretical analysis.

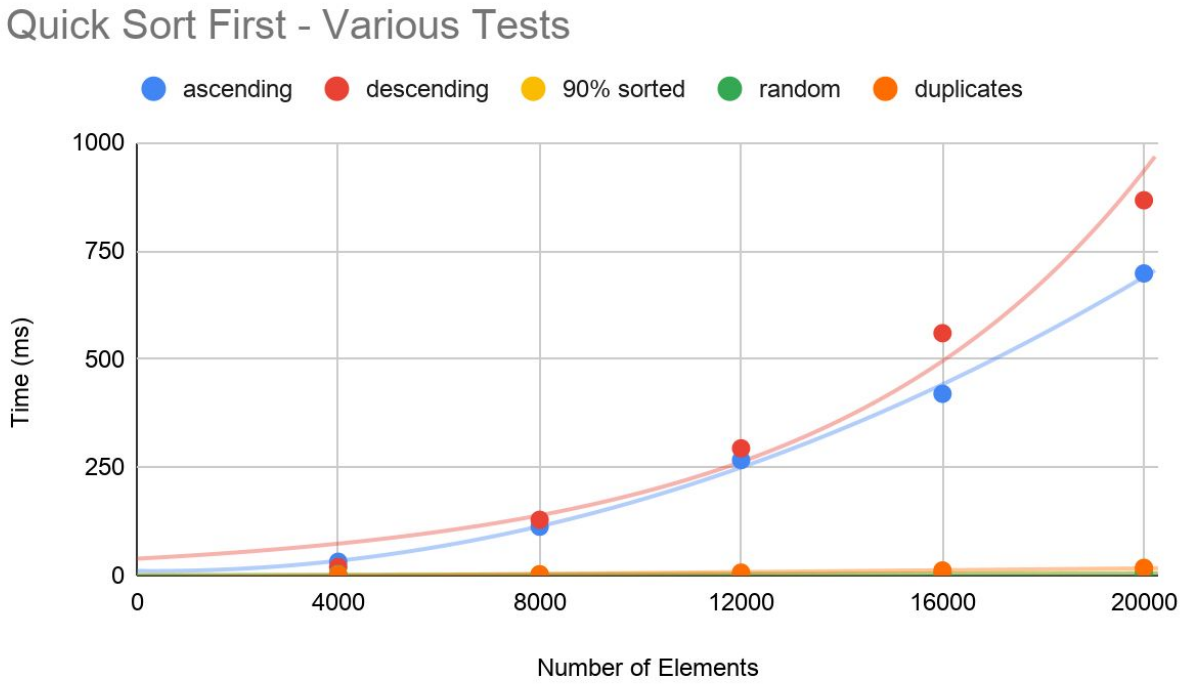

Figure 7: Times for quicksort with first element as the pivot on various test arrays.

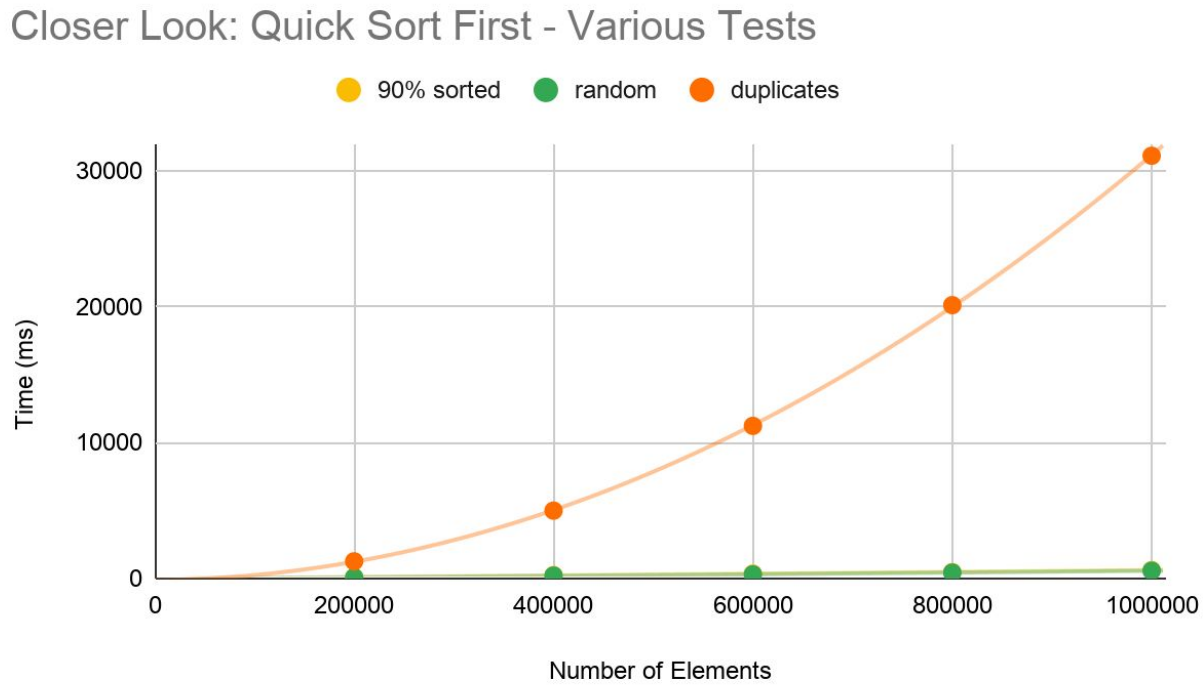

Figure 8: Times for quicksort with first element as the pivot for more elements without the ascending and descending arrays.

Figure 7 shows how when the first element is chosen as the pivot for quicksort for ascending and descending arrays, the worst case runtime of $O(n^2)$ occurs because the partitions only decrease in size by one element. This agreed with our theoretical analysis. Figure 8 shows that 90% sorted and random were the best case with $O(nlg(n))$ runtime. For the random case, there is a good chance that a pivot that will result in a relatively even partition will be chosen. We expected in the theoretical analysis that the 90% sorted case would be similar to the ascending case, but it was actually the one of the best cases, because 10% of the values being random would have made the partitions more balanced, which may have contributed to better performance. The duplicates case was $O(n^2)$, though faster than the ascending or descending arrays. This suggests that having many duplicates creates an uneven partition as the copies of the pivot end up on one side, but the partition is more even than when the pivot is the largest or smallest element.

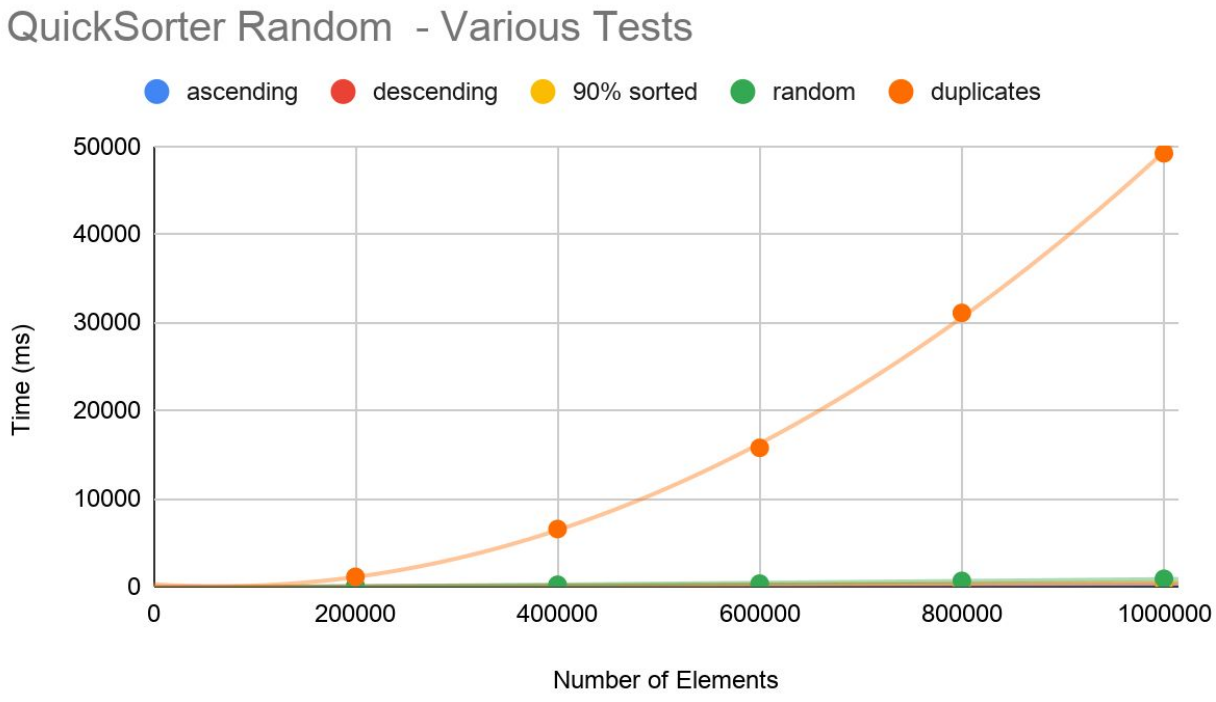

Figure 9: Times for quicksort with random partition on various test arrays.

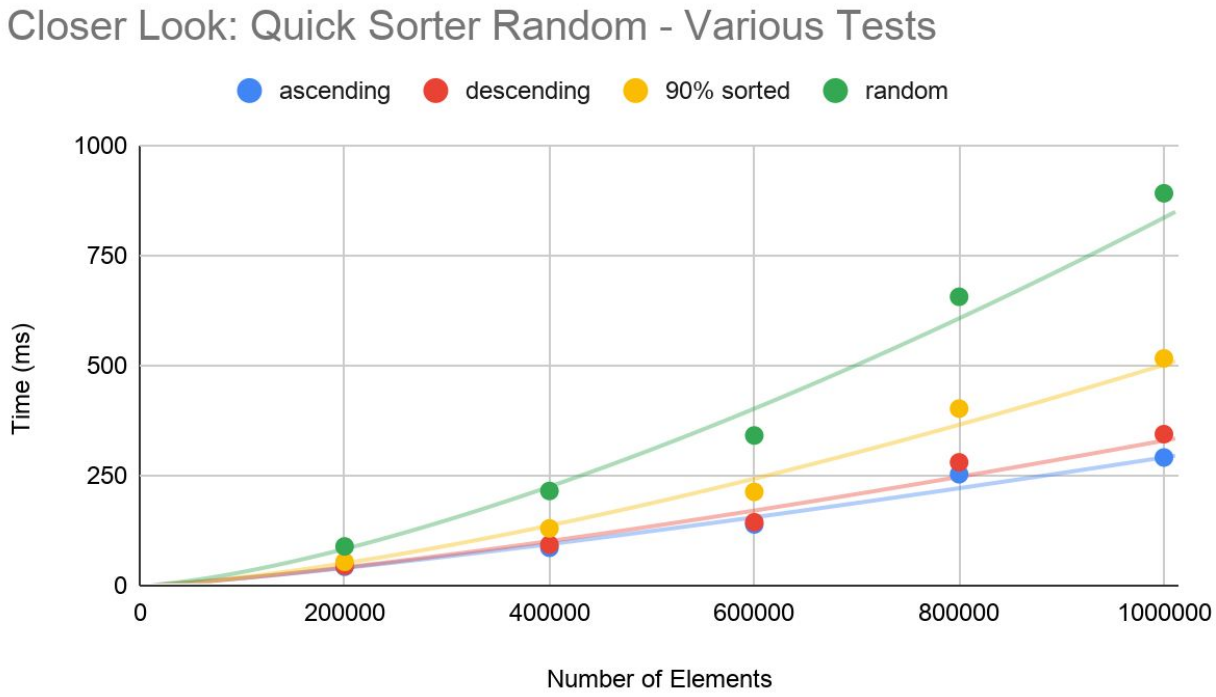

Figure 10: A closer look at Figure 9, omitting the duplicates case.

Figure 9 shows the runtime of quicksort when a random element is chosen as the pivot, and Figure 10 gives a closer look at the data without the worst case of the duplicates array. The duplicates array is the worst case ($O(n^2)$) because the copies of the pivot end up on one side of the partition, making it uneven. This would be the rare case that quicksort random would be the worst case, as specified in the introduction. Random, the average case, also started to approach a slightly more quadratic-looking curve as the elements increased to 1,000,000. Possibly, with up to 1,000,000 random elements, there could have been more duplicates in the arrays than with lower element counts. The average case for quicksort is theoretically $O(nlg(n))$ and though our average case is slightly quadratic, compared to the worst case is closer to $O(nlg(n))$. However, the best cases were ascending, descending, and 90% sorted because there were few to no duplicates, and the pivot value was likely to be around the middle of the data based on probability. These best cases had $O(nlg(n))$ runtime, and they agreed with the theoretical analysis in the introduction.

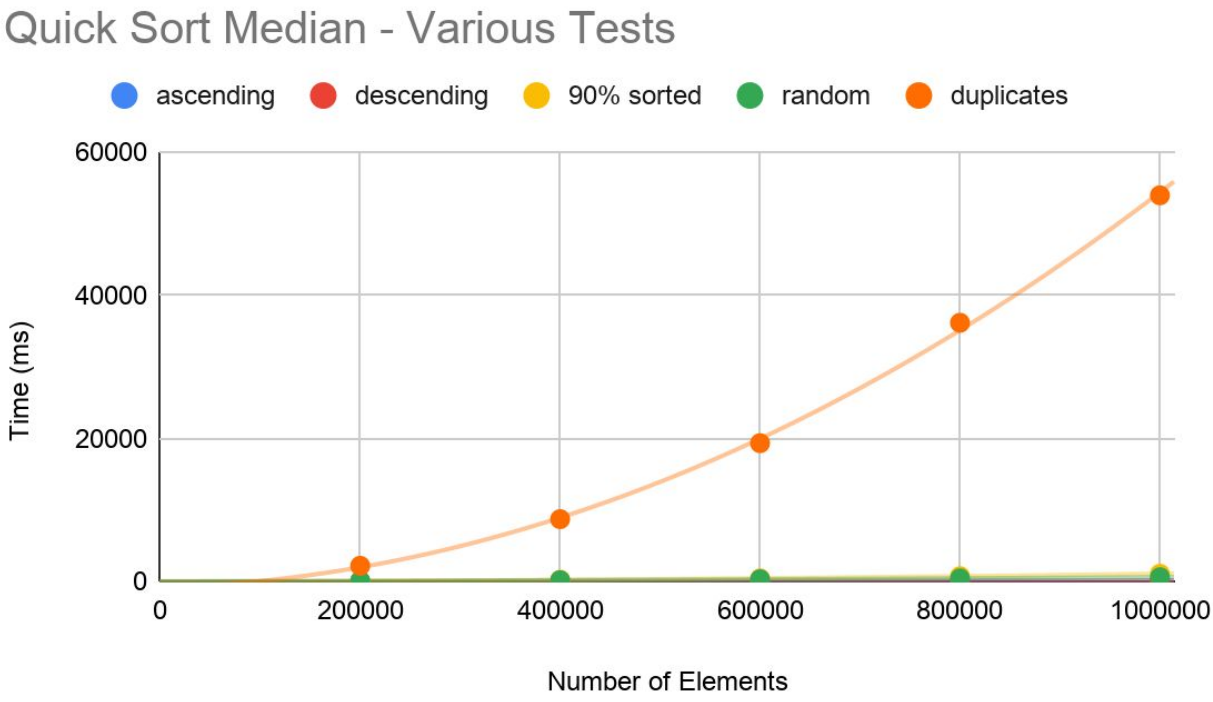

Figure 11: Times for quicksort with the median pivot value on various test arrays.

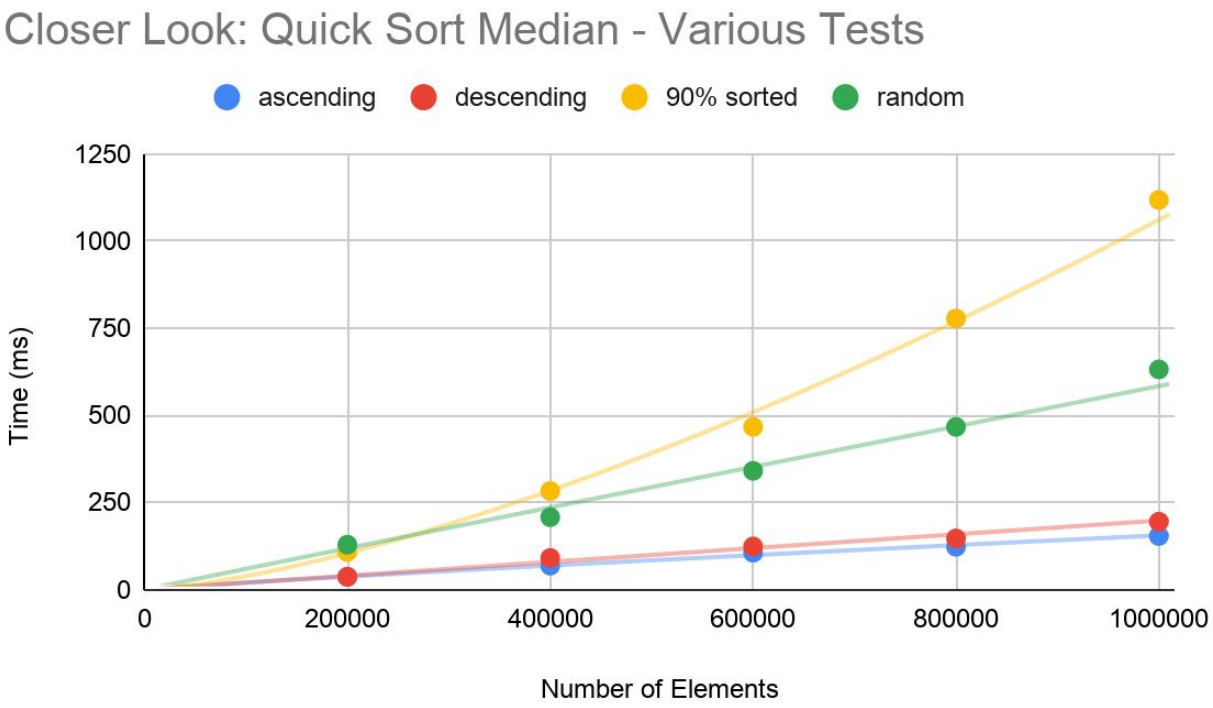

Figure 12: A closer look at Figure 11, omitting the duplicates case.

Figure 11 shows the runtime of quicksort when the median of three is chosen as the pivot, and Figure 12 gives a closer look, omitting the worst case of the duplicates array. The worst case is $O(n^2)$ because the duplicates of the pivot value would end up on one side of the partition, making it uneven. 90% sorted was much faster than the duplicates case but started to appear quadratic at 1,000,000 elements. This could have occurred because at least two of the three pivot choices (the first and last) would be poor in almost all cases, creating an uneven partition if the middle value was random and relatively large or small. Random produced the average case of $O(n \lg(n))$—the first, middle, and last values of the array had just as much chance of being close to the median as any other value in the array. The best cases were the ascending and descending arrays because the exact median was chosen as the pivot every time, resulting in perfectly even partitions and $O(n \lg(n))$ runtime. These results agreed with our theoretical analysis in the introduction.

The best method to select the pivot depends on the original order of the array. Overall, selecting the first element as the pivot was the worst choice—sorting was quadratic for the ascending and descending arrays, and it performed about the same as the random and median pivots for the other test cases. The median and random pivot choices performed similarly, except that random was slightly better for almost-sorted data and median was slightly better for random data. Both of them ran in $O(n \lg(n))$ time for ascending and descending arrays, even if the median pivot was a few milliseconds faster. In practice, the decision between a random pivot and median pivot would depend on the expected initial order of typical data: whether it would usually be almost-sorted or completely random. All of the quicksort versions ran in quadratic time for duplicates, but the first element as the pivot performed the best. However, this case would be too rare to justify picking the first element as the pivot in most circumstances.

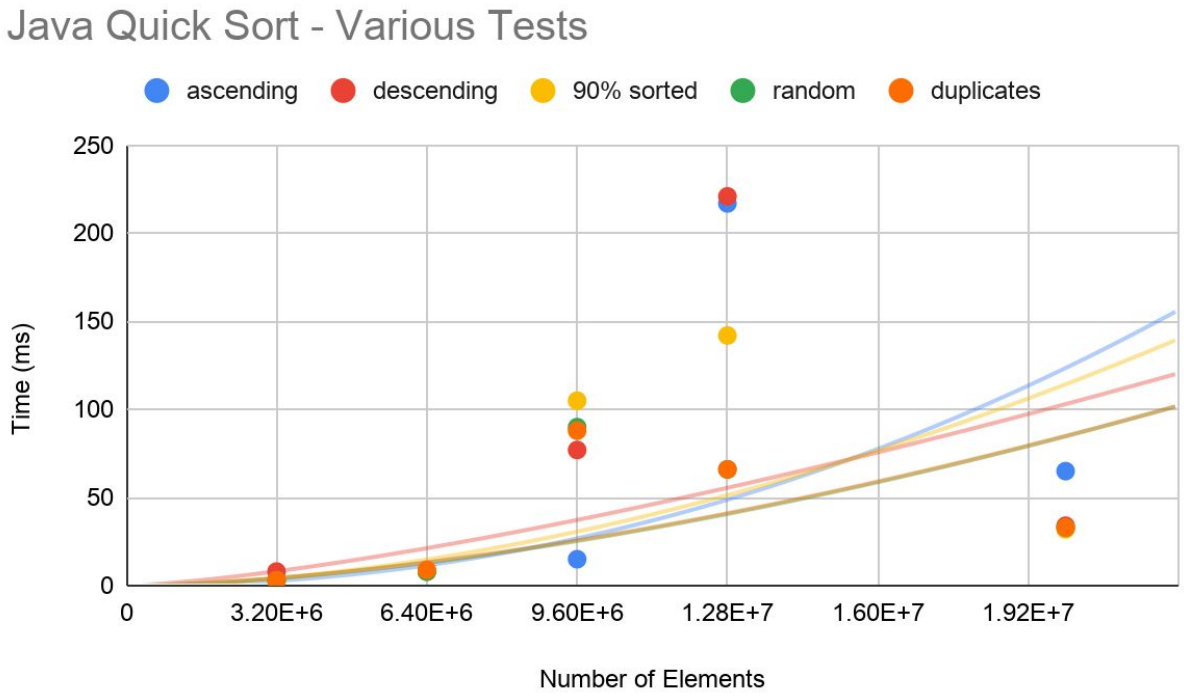

Figure 13: Times for Java Standard Library quick sort on various test arrays.

Figure 13 shows that the data for Java's implementation of quicksort was irregular, even for 20,000,000 elements. This may be because Java's quicksort uses a dual-pivot system and also an adaptation of quicksort that is stable, which could complicate runtimes for various test cases (Tachenov, 2016). Compared to our implementation of quicksort, Java's version was extremely fast. Even the largest data point for Java's quicksort, 221 ms for a descending array of 12,800,000 elements, was as fast or faster than all the data points for our quicksort implementations with 800,000+ elements. According to the Java API for the `Arrays` class (2020), the Java implementation uses a heavily-enhanced version of quicksort with dual pivots. All of the optimizations done on Java's quicksort make it so fast that data from it is highly variable, though the random and duplicates cases appear to be the fastest. Possibly, ascending and descending arrays with around 12,800,000 elements, the largest data points in Figure 13, represent edge cases in Java's complex internal implementation that are less optimized.

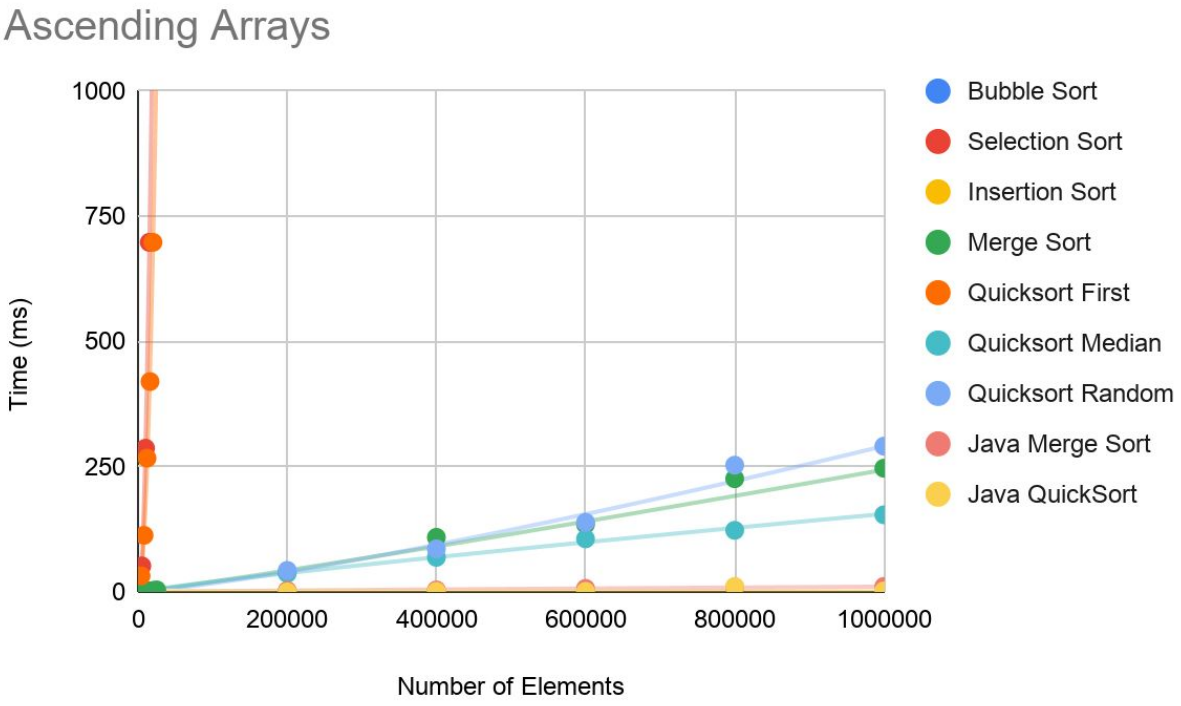

Figure 14: Comparison of different sorting algorithms' performance on ascending arrays.

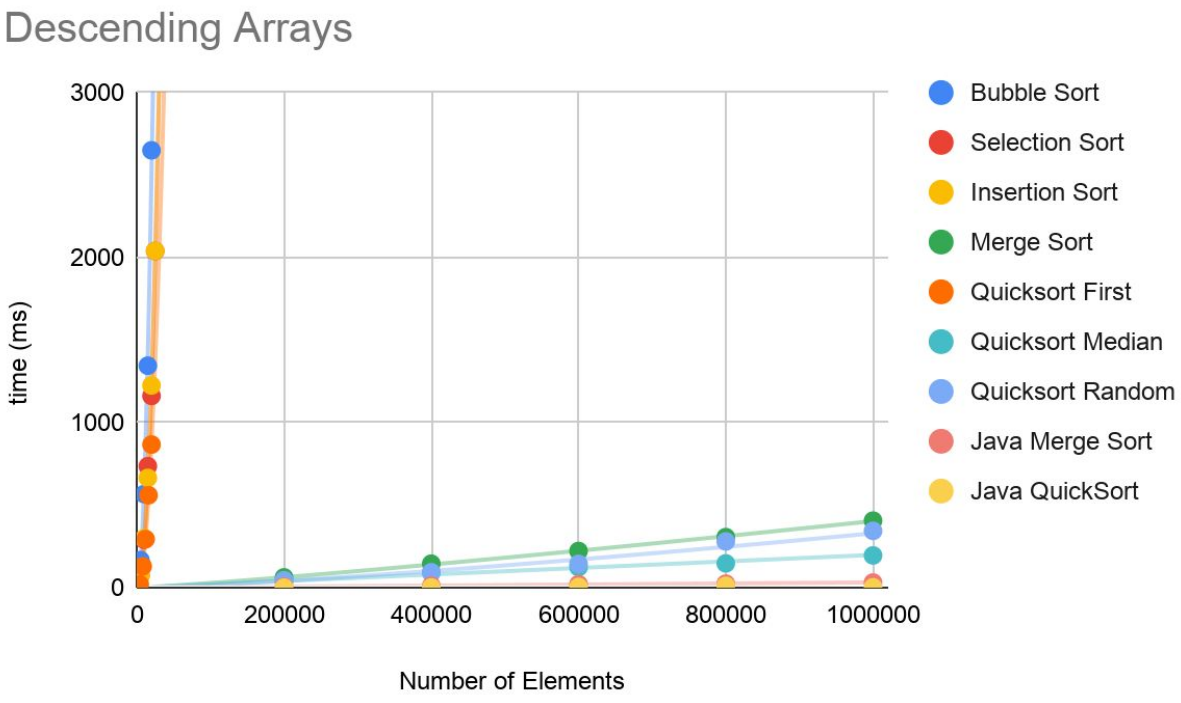

Figure 15: Comparison of different sorting algorithms' performance on descending arrays.

According to Figure 14 and Figure 15, the worst choices to sort an ordered array, either ascending or descending, would be selection sort and quicksort first. Bubble sort and selection sort are also the worst choices for descending arrays, but the best choices for ascending arrays because they run in $O(n)$ time for already-sorted arrays. Another good choice close to the best case would be to use the Java standard library implementation of either merge sort or quicksort because those have been heavily optimized and run in $O(nlg(n))$ time for all or almost all arrays. Of course, merge sort requires $O(n)$ extra space, so quicksort would be better when memory is limited. If those are not possible, then using quicksort median is the best choice because of its $O(nlg(n))$ runtime nature as it chooses the best pivot every time. Average cases are the random and median quicksort implementations and merge sort, which still run in $O(nlg(n))$ but are not as optimized as the Java implementations.

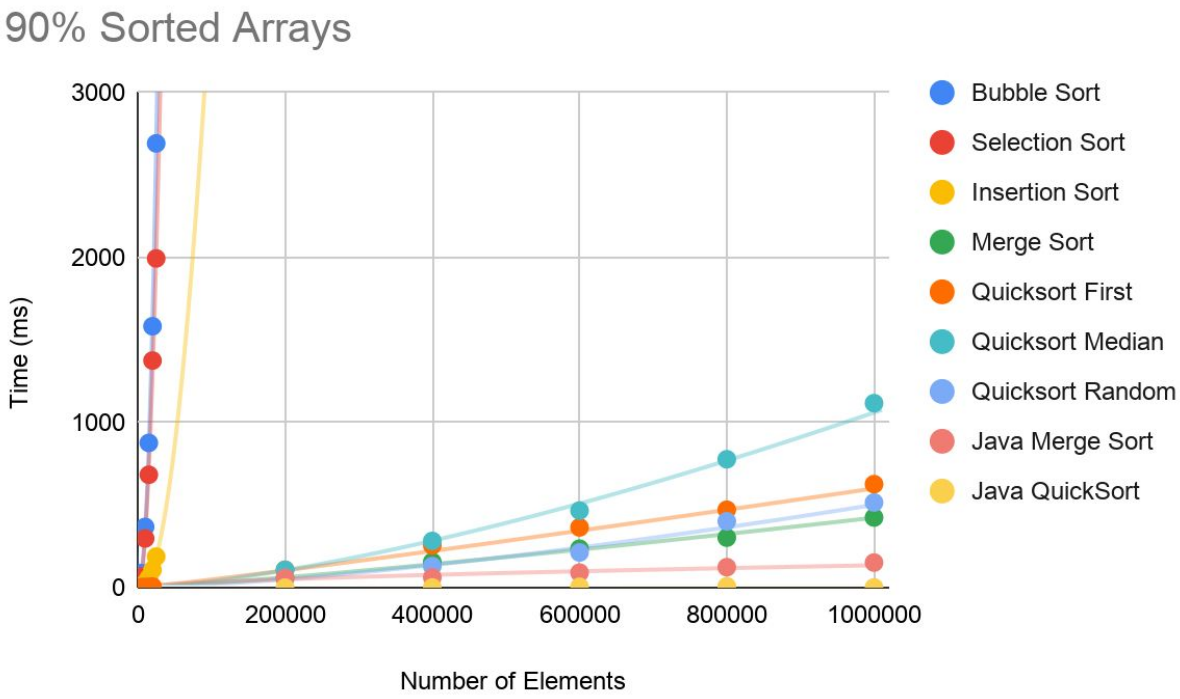

Figure 16: Comparison of different sorting algorithms' performance on 90% sorted arrays.

According to Figure 16, the worst choices for an almost-sorted array are bubble sort and selection sort, closely followed by insertion sort. Those three sorting algorithms all ran in $O(n^2)$

time for the 90% sorted arrays, even though those arrays were close to the best case for bubble sort and insertion sort. Between bubble and insertion sort, Figure 16 shows that insertion sort performed better, even if it still had quadratic runtime—insertion sort has more tolerance for unsorted values than bubble sort. In contrast, the Java Standard Library sorters (merge sort and quicksort) would be the best choices because they have been optimized and ran in $O(nlg(n))$ time. If those are not available, the next best options would be one of the three quicksort versions or our implementation of merge sort because they also ran in $O(nlg(n))$ time, even if they have not been optimized to the same extent that the Java versions have. Since there is some randomness to the almost-sorted case, using merge sort may be better to guarantee $O(nlg(n))$ runtime, even if it requires $O(n)$ extra space and Java's quicksort runs slightly faster in most cases.

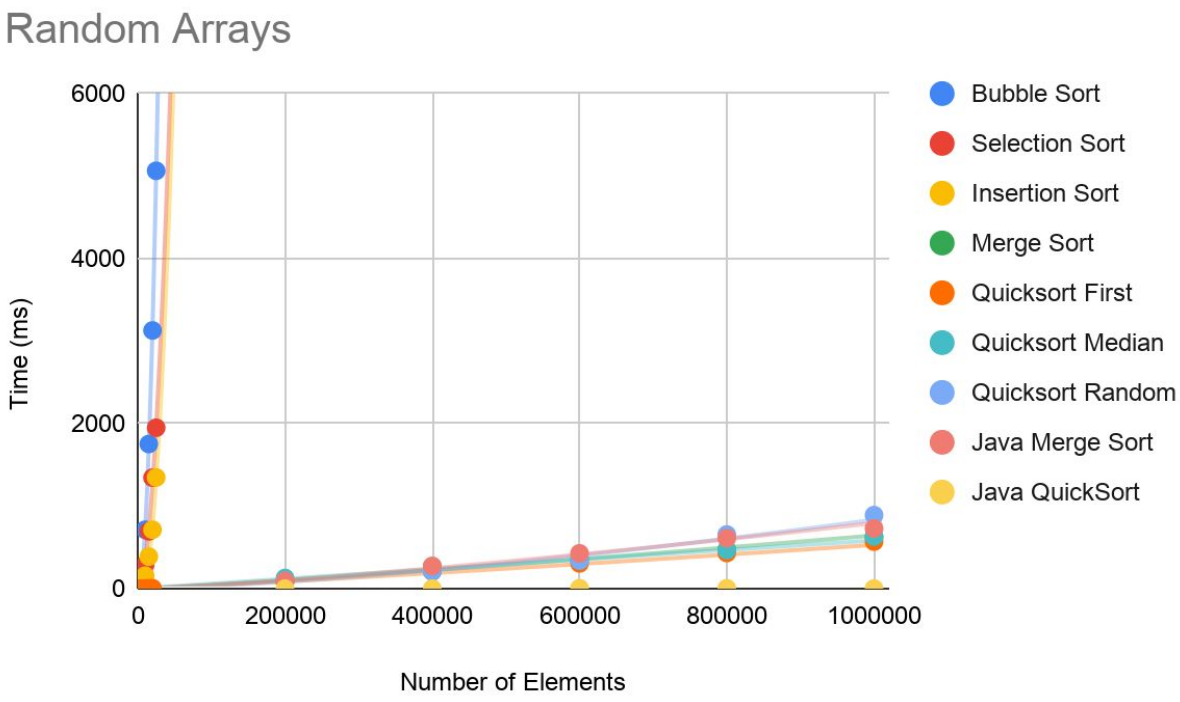

Figure 17: Comparison of different sorting algorithms' performance on random arrays.

According to Figure 17, the worst choices to sort a random array are bubble sort, selection sort, and insertion sort because they ran in $O(n^2)$ time. The Java Standard Library's

version of quicksort would be the best choice, being heavily optimized and having $O(nlg(n))$ average runtime as shown in the graph. If that is not available to use, then quicksort median and quicksort first would be best, followed closely by Java's and our versions of merge sort and quicksort random. Both the first and median pivot selections choose random values in this case, so the construction of a random integer generator in quicksort random would be unnecessary. However, all the merge sort and quicksort versions ran in $O(nlg(n))$ time.

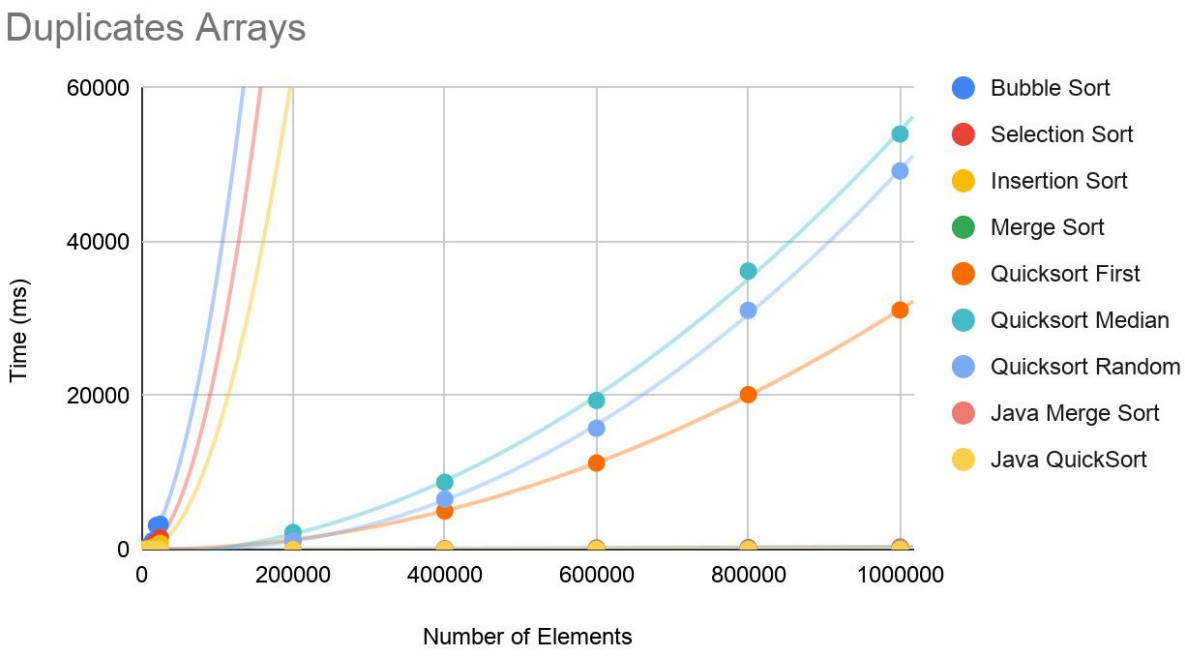

Figure 18: Comparison of different sorting algorithms' performance on duplicates arrays

Figure 18 shows that only merge sort and the enhanced Java implementation of quicksort were able to efficiently sort arrays with many duplicates. Bubble sort, selection sort, and insertion sort would be the worst choices to sort such an array with steeply-increasing $O(n^2)$ runtimes. Likewise, all three of our implementations of quicksort appeared to run in $O(n^2)$ time, even if they were more efficient than the three least-efficient algorithms. Of these versions of quicksort, quicksort first performed the best, likely because its method of selecting a pivot is the simplest. Merge sort guarantees $O(nlg(n))$ time, so even if it requires O(n) extra space, our implementation and Java's implementation would be good choices. Since Java's quicksort also

ran in `O(nlg(n))` time, it appears that it has been optimized to account for cases with many duplicates.

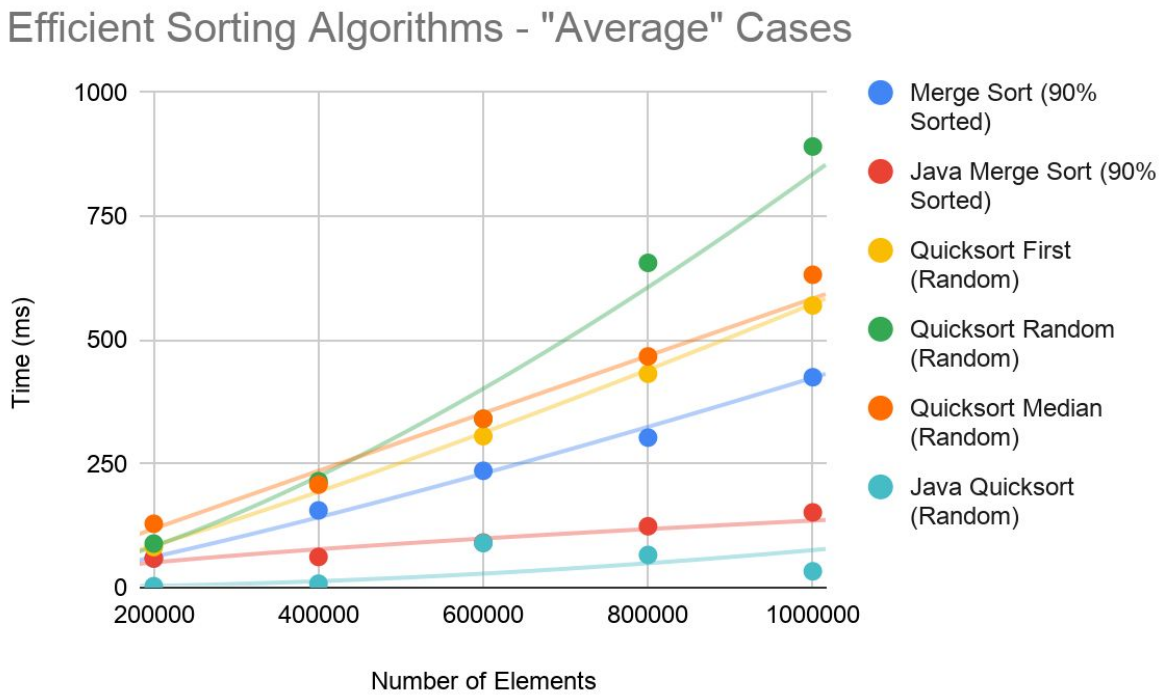

Figure 19: Comparison of efficient sorting algorithms for cases in between their best and worst performance.

Figure 19 compares the efficient sorting algorithms (versions of merge sort and quicksort) for cases where they had middle-range/"average" performance; the cases were not their best case nor their worst case from the data. The middle cases for both merge sorts were descending and 90% sorted, but since descending is more of an edge case than 90% sorted, we chose 90% sorted to represent an "average" case of merge sort. Likewise, random was chosen as the middle-ground case for all of the quicksorts. Java's implementation was the fastest sorting algorithm, closely followed by Java's merge sort. In practice, the optimizations done in Java's libraries outweigh the specifics of custom implementations in terms of speed. Our custom merge sort performed slightly better than the custom quicksort algorithms in "average" cases with guaranteed `O(nlg(n))` performance, though it required `O(n)` extra space. Between the three quicksort algorithms, quicksort first and quicksort median performed better on "average" than

quicksort random. However, quicksort first suffered dramatically for ascending and descending arrays, as we showed earlier. Quicksort median suffered, albeit less dramatically, for 90% sorted cases. Overall, the reliability of merge sort and quicksort random would be more beneficial than the slight advantages of quicksort first and quicksort median in general cases.

5. Conclusion

In general, when the initial state of the arrays being sorted is not known, the best sorting algorithm out of the ones we implemented is merge sort. It was the most consistent because it was not vulnerable to the edge case of many duplicates and showed `O(nlg(n))` runtime across the board. In contrast to quicksort, choosing a good pivot is not a concern, as the array is always split in the middle. Merge sort is also stable. The only major disadvantage is that it does take `O(n)` extra memory. However, memory space is becoming cheaper and cheaper as technology improves, diminishing the potential harmful effects of needed extra memory.

If it is known that there will be few duplicates in the array, then in practice, quicksort random or quicksort median perform extremely well, also showing `O(nlg(n))` runtimes, and do not use any extra memory space. For quicksort first, the simplicity of just picking the first element as the pivot is greatly outweighed by the worst case performance in ascending and descending arrays. The major disadvantage for all the different implementations of quicksort is that there is always the possibility that a bad pivot may be chosen and thus the initial order of the arrays must be considered. It also is an unstable sorting algorithm.

Among the quadratic sorting algorithms of bubble sort, selection sort, and insertion sort, insertion sort is the best. Though it is dependent on the initial state of the array, its worst case is no worse than selection sort's best, average, and worst cases. In line with that, the average and best cases of insertion sort perform better than the average and best cases of selection sort. It is also a stable sorting algorithm whereas insertion sort is not. Selection sort is better than bubble sort though, which has extremely poor performance in most cases, with the exception of an already-sorted ascending array. However, this has very little practical value as there would be very few cases where someone would want to sort an already sorted array.

The only time that someone may want to use a quadratic sorting algorithm may be when the array size is very small (under 500 elements) because their implementation is simpler and

would be less bug-prone. Other than that, using either merge sort or quicksort is almost always preferred.

If stable sorting is desired, then merge sort, insertion sort, or bubble sort can be used. Again, the best option would likely be merge sort for large arrays, insertion sort for small arrays, and almost never bubble sort. If stable sorting is not a concern, then quick sort or selection sort can be used. However, insertion sort still performs better than selection sort across the board, and merge sort tends to be more reliable than quicksort.

The most important knowledge we learned is that Big-Oh notation is really an approximation and an analysis of the general trend of a curve. There can be huge differences of about 1000 ms in the runtimes of algorithms that are all technically $O(nlg(n))$. Similarly, there can be huge differences in the runtimes of algorithms that are all technically $O(n^2)$. For example, bubble sort took around three times longer than the other quadratic sorting algorithms when sorting the same number of elements. Additionally, it would be best to use the Java standard library implementations of merge sort and quicksort in practice as these tend to outperform the user writing their own sorting algorithms. Although Java's implementations are more efficient, it is important to understand the analysis behind the benefits and drawbacks of these algorithms to make informed choices in software design.

6. References

JUnit Team. (2020). *Packages.* Junit4 Javadoc 4.8. https://junit.org/junit4/javadoc/4.8/

Oracle. (2020). *Class Arrays.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

Oracle. (2020). *Class File.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/io/File.html

Oracle. (2020). *Class FileNotFoundException.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/io/FileNotFoundException.html

Oracle. (2020). *Class Random*. Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

Oracle. (2020). *Class PrintWriter.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html

Oracle. (2020). *Class Scanner.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

Tachenov, S. (2016, April 17). Drawback of using Arrays.sort() method in Java instead of QuickSort or MergeSort [Online forum post]. StackOverflow. https://stackoverflow.com/questions/36676433/drawback-of-using-arrays-sort-method-in-java-instead-of-quicksort-or-mergesort/36676801#36676801

Xia, G. (2020). Sorter [Java].

Xia, G. (2020). Sorting [Microsoft PowerPoint].