Analyzing Different Sorting and Searching Algorithms

Zeid Ayssa
University of Texas at Austin
Dept. of Electrical and Computer Engineering
azeid@utexas.edu

Jose Martinez Garcia-Vaso
University of Texas at Austin
Dept. of Electrical and Computer Engineering
carlosgvaso@utexas.edu

Utkarsh Vardan
University of Texas at Austin
Dept. of Electrical and Computer Engineering
uvardan@utexas.edu

ABSTRACT

In this project we will analyze a variety of sorting algorithms and explore how they compare to each other in terms of performance as well as time and space complexity. Additionally, we will also show how different sorting algorithms perform with different test cases and data-set sizes by benchmarking performance using real input test cases. Add to that, we will explore the possibility to visualize each sorting algorithm and demonstrate how effectually different data sets from different starting point can be sorted using different algorithms.

1 INTRODUCTION

The sorting problem is one of the fundamental problems in Computer Science. It consists of obtaining a permutation of a sequence of numbers sorted in non-decreasing order. The problem is defined as follows[3]:

Input: A sequence of n numbers $\langle a_0, a_1, ..., a_{n-1} \rangle$.

Output: A permutation of the input sequence $\langle a'_0, a'_1, ..., a'_{n-1} \rangle$ such that $a'_0 \leq a'_1 \leq ... \leq a'_{n-1}$.

Over the times, there have been many algorithms developed to solve this problem. These algorithms employ a variety techniques, data structures and mathematical properties, which lead to some algorithms performing better than others for different input types. In our research, we focused specifically on the problem of sorting an array of integers, and we studied the performance of nine sorting algorithms when presented with multiple inputs of different sizes and makeups. We compared the performance of the following algorithms:

- Bubble Sort.
- Insertion sort.
- Quicksort.
- Java Collections Framework Arrays sort implementation.
- Mergesort.
- Heapsort.
- Shell Sort.
- Radix Sort.
- Timsort.

In the next subsections, we will introduce the previous algorithms, and we will talk about their theoretical time and space complexities. We will also go over their best, average and worst case inputs.

1.1 Bubble Sort

Bubble sort works by passing the data in the collection by multiple passes. The data is processed from start to end, or left to right. Starting from the first value in the collection, the value is compared to next value, if the value is larger than the next value they are swapped. The largest value is moved to the right. This comparison and swap operation is repeated for each value in the collection until no swaps are performed. Then, the array is in sorted order.

Figure 1 shows how bubble sort works on an example array. In the 1st pass (see *Figure 1 (a)* and *(b)*), we compare 4 with 7, and swap is not performed because 4 is already smaller than 7. Next, 7 is compared and swapped with the next elements: 5 and 5 (*(b)* and *(c)*). Similarly in the third pass *(d)*, 7 is swapped with 6. As shown in *e*, the fourth pass *(e)* yields a sorted array, since no swaps are performed.

Figure 2 shows the time and space complexities of Bubble sort.

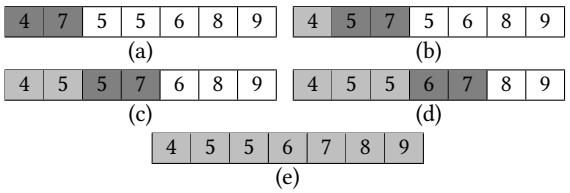


Figure 1: Bubble sort visualization. (a) shows the input array. The comparisons are shown in dark gray, and sorted elements are shown in light gray. (b) shows pass 1 result, (c) shows pass 2 result, (d) shows pass 3 result and (e) shows pass 4 result which returns the array in sorted order.

Time Complexity	
Best Case	$\Omega(n)$
Average Case	$\Theta(n^2)$
Worst Case	$O(n^2)$

Space Complexity	
Worst Case	$O(1)$

Figure 2: Bubble sort time[5] and space complexities[7].

1.2 Insertion Sort

The insertion sort algorithm sorts the the array of integers in-place in a similar way as we would sort a had of cards. It divides the array in a left and right sides, with the goal of ending with all the numbers sorted in the left side and no numbers in the right side. We start with all the numbers in the right side, and we add them one by one to the left side in ascending order. To make sure the numbers are inserted in the correct place, the numbers are compared from right to left with the numbers already in the left side of the array[3].

Insertion performs single pass in the collection of data. All the data on the left side of the item currently been evaluated is know to be sorted and all the data to the right is considered to be unsorted. *Figure 3* shows insertion sort in action. We start with an unsorted array (see *Figure 3 (a)*). The first item in the collection, since there is nothing to the left of 6, it is considered to be sorted. Since 4 is less than 6, we make a swap, and 4 and 6 are considered to be sorted. The rest of the numbers are unsorted. Since 5 is less than 6, we perform a swap between 5 and 6. We continue this operation for 9, since 9 is greater than 6, no swap is performed. Since 7 is less than 9, we swap 7 and 9. Finally, since 8 is less than 9, we again swap 8 and 9. In a single pass, we have sorted the entire array of data using the insertion sort (see *Figure 3 (b)*).

Figure 4 shows the time and space complexities of insertion sort.

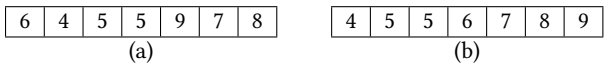


Figure 3: Insertion sort visualization. (a) shows the input array. (b) shows the array after one pass of insertion sort.

Time Complexity	
Best Case	$\Theta(n)$
Average Case	$\Theta(n^2)$
Worst Case	$O(n^2)$

Space Complexity	
Worst Case	$O(1)$

Figure 4: Insertion sort time[3] and space complexities[7].

1.3 Quicksort

Quick sort is a divide and conquer algorithm. It's also one of the most commonly used general purpose sorting algorithm in computer science. Since it is a divide and conquer algorithm, we will be dividing the data into smaller sets. In quick sort, the arrays are not necessarily split in half, rather a pivot value is picked based on the rules or a heuristic. Once a pivot value is picked, all the items in the array smaller than the pivot are placed at the left side of pivot, and entries to the right of the pivot are larger than the pivot. This pivot and partition operation is performed repeatedly on left and right side of partitions until all the items are sorted.

In the unsorted array of data shown in *Figure 5 (a)*, we choose 5 as the pivot. All the entries less than 5 will be moved to the left of 5, and values greater than 5 will be moved to the right of 5. Therefore, 4 and 8 are swapped resulting in *Figure 3 (b)*. Next we choose 2 as the next partition, all the values to the left of 2 are larger, and values to the right are smaller. Therefore, we need to pull 2 out of the array, and place all the values around in their appropriate location as shown in *Figure 5 (c)*. If we pick 3 as pivot, entries to the left of 3 are smaller. Therefore, it is sorted. Similarly, we repeatedly pick to pivot to the right side of 5 until all elements are sorted as displayed in *Figure 5 (d)*.

Figure 6 shows the time and space complexities of insertion sort.

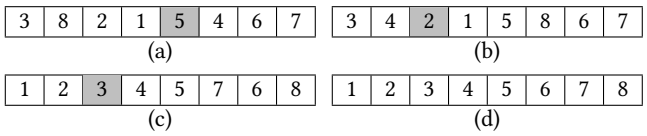


Figure 5: Quicksort visualization. (a) shows the input array. (b) shows the array after using 5 as the pivot. (c) shows the array after uisng 3 as the pivot. (d) shows the sorted array.

Time Complexity	
Best Case	$\Omega(n \lg n)$
Average Case	$\Theta(n \lg n)$
Worst Case	$O(n^2)$

Space Complexity	
Worst Case	$O(\lg n)$

Figure 6: Quicksort time[3] and space complexities[7].

1.4 Java Collections Framework Arrays sort implementation

The Java Collection Array Sort algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch [4]. This algorithm offers $O(n \lg n)$ performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations. According to the latest update on the Array Sort algorithm by Vladimir Yaroslavskiy [8], there has been a lot of changes in order to optimize the Array Sort algorithm. These changes include:

- Pivots are chosen with another step, the 1-st and 5-th candidates are taken as pivots instead of 2-nd and 4-th.
- Splitting into parts is related to the golden ratio.
- Pivot candidates are sorted by combination of 5-element network sorting + insertion sort
- Pivot candidates are sorted by combination of 5-element network sorting + insertion sort
- New backwards partitioning is simpler and more efficient
- Quicksort tuning parameters were updated
- Merging sort is invoked on each iteration from Quicksort
- Heap sort is invoked on the leftmost part
- Heap sort is used as a guard against quadratic time
- Merging sort and pair insertion sort were moved from DualPivotQuicksort class
- Pair insertion sort was simplified and optimized
- New nano insertion sort was introduced for tiny arrays
- Merging sort was fully rewritten
- Optimized merging partitioning is used
- Merging parameters were updated
- Merging of runs was fully rewritten
- Fast version of heap sort was introduced

This list above shows how the Java Collection Array sort algorithm is highly optimized and it uses multiple sorting algorithms underneath to increase performance. Given that this algorithm adapts and uses different sorting algorithms and techniques underneath, it is hard to specify the overall time and space complexity. However, we can get a feel of the complexity by looking and the performance of Quick Sort, Merge Sort, and Heap Sort.

1.5 Mergesort

Merge sort works by recursively splitting the data in half. For example, an array of 10 items would be split in the middle into two sub-arrays of 5 items each. The splitting continues until each sub-array has only one item in it. Since each sub-array has only one item in it, that sub-array is known to be sorted. At this point, the sub-arrays are merged, but the values are put together in sorted order. After each merger, the sorted sub-array doubles in size, and this procedure continues until all the sub-arrays are merged and fully sorted.

Initially, the arrays are recursively split in half. First, the initial array (see *Figure 7 (a)*) is split into 2 sub-arrays of 4 entries each as shown in *Figure 7 (b)*. Next, the sub-arrays are split in arrays of 2 entries each as shown in *Figure 7 (c)*. Then, the resulting sub-arrays are split into 8 sub-arrays of 1 entry each as shown in *Figure 7 (d)*. Because there is only one item in the sub-arrays shown in *Figure 7 (d)*, the entries in the sub-arrays are sorted within their sub-array. For the reconstruction phase, we merge each single entry sub-array back to sub-arrays of 2 entries each by sorting them. Then, 4 and 9 are reconstructed into a sub-array as they are already sorted. 3 and 2 are reconstructed into an array with 2 and 3 in sorted order. 6 and 5 are reconstructed as 5 and 6. 7 and 8 are reconstructed in the same order. The result is shown in *Figure 7 (e)*. Similarly, we merge the resulting sub-arrays of 2 entries each back to 2 sub-arrays of 4 entries each in sorted order, which is shown in *Figure 7 (f)*. In the

final reconstruction step, the 2 remaining sub-arrays are merged into a single sorted array as shown in *Figure 7 (g)*.

Figure 8 shows the time and space complexities of Mergesort.

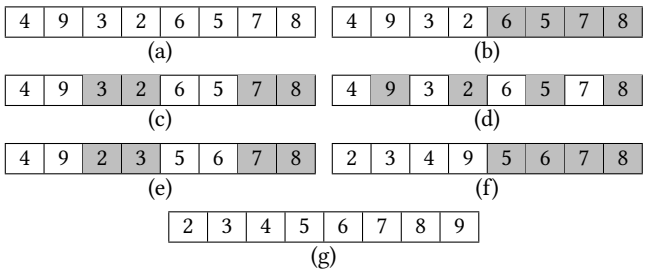


Figure 7: Mergesort visualization. (a) shows the input array. (b), (c) and (d) show how mergesort splits the initial array into sub-arrays. Different sub-arrays are distinguished by the different cell backgrounds. (e), (f) and (g) shows how mergesort joins back the sub-arrays in sorted order.

Time Complexity	
Best Case	$\Omega(n \lg n)$
Average Case	$\Theta(n \lg n)$
Worst Case	$O(n \lg n)$

Space Complexity	
Worst Case	$O(n)$

Figure 8: Mergesort time[3] and space complexities[7].

1.6 Heapsort

Heapsort is comparison based sorting technique based on binary heap data structure. It is similar to selection sort in terms of finding the maximum element, and placing the maximum element in the end. This process is repeated for the remaining elements in an array.

Initially, heap data structure is created using a min-heap or max-heap for the unsorted list of elements. The first element (root node) of the heap is either largest or smallest depending if it is a min or max-heap. We take out first element of the heap, and we place it at the end of the array. Then we reduce the heap size to keep the sorted element outside of the heap. We again make the heap using the remaining elements, and we pick the first element of the heap store in the end of the array. This process is repeated until we have the array completely sorted.

Consider the example shown in *Figure 9*, where we use a max-heap. After building the max-heap, we get the array shown in *Figure 9(b)*. Since 9 is the largest element (root node of the heap), it is removed from the heap, and it is swapped with the last entry of the array as shown in *(c)*. The size of the heap is reduced by one, which is shown by graying out the entries that are no longer part of the heap. Rebuilding the max-heap with the remaining entries, we obtain the array shown in *(d)*. Since 6 is the root of the heap, it is swapped with the last element of the heap, and it is removed from the heap by reducing its size by one (see *(e)*). *(f)* shows the result of rebuilding the max-heap, and *(g)* shows the result of removing the root node of the heap. Continuing this process, we reach a fully sorted array as shown in *Figure 9 (j)*.

Figure 10 shows the time and space complexities of Heapsort.

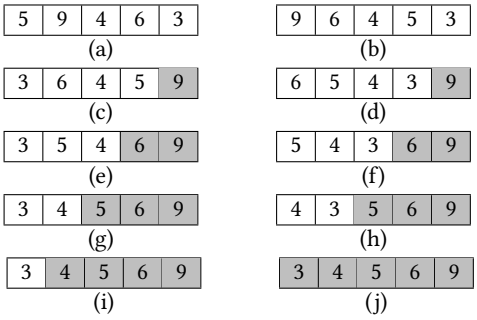


Figure 9: Heapsort visualization. (a) shows the input array. (b), shows how initial array is made into a max-heap. In (c), the root node is swapped with the last element of the array, and the heap size is reduced by one. The elements in the heap are in white, while the elements outside the heap are greyed out. The previous steps are repeated from (d) to (j), until a fully sorted array is produced as shown in (j)

Time Complexity	
Best Case	$\Omega(n \lg n)$
Average Case	$\Theta(n \lg n)$
Worst Case	$O(n \lg n)$

Space Complexity	
Worst Case	$O(1)$

Figure 10: Heapsort time[3] and space complexities[7].

1.7 Shell Sort

Shell sort is a generalization of the insertion sort algorithm. Using shell sort, we compare the elements of the array that are far apart, rather than adjacent. For this purpose, we define the variable *gap*, and we compare the elements that are *gap* number of elements away from each other. This divides the initial array in a *gap* number of interleaved sub-arrays, whose elements are sorted by comparison.

With every iteration, we reduce the gap between the elements being compared, and we sort the sub-arrays. When we sort in the last pass where the $gap = 1$, the array is fully sorted. In the last pass, shell sort behaves like insertion sort.

Consider the example in *Figure 11 (a)*, where we want to sort the array in ascending order. In this example, we start with a value of $gap = 2$, but the initial value of *gap* could be anything, and we reduce the value of gap by dividing it by 2 with each pass. In *(b)*, we show how the initial array is divided in interleaved sub-arrays which are shown in different colors. Next, we sort the sub-arrays individually, as shown in *(c)* and *(d)*. Now, we are done with the iteration, and we can move to the next iteration with $gap = 2/1 = 1$. This means the whole array will be considered now as shown in *(e)*. Now, we would rearrange the elements in sorted order, but the items were already sorted in our example (see *(f)*).

Figure 12 shows the time and space complexities of Shell sort.

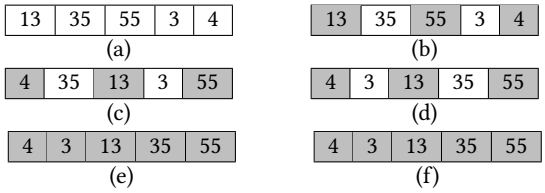


Figure 11: Shell sort visualization. (a) shows the input array. (b), (c) and (d) show how mergesort splits the initial array into sub-arrays. (e), (f) and (g) shows how mergesort joins back the sub-arrays in sorted order.

Time Complexity	
Best Case	$\Omega(n \lg n)$
Average Case	$\Theta(n (\lg n)^2)$
Worst Case	$O(n (\lg n)^2)$

Space Complexity	
Worst Case	$O(1)$

Figure 12: Shell sort time[7] and space complexities[7].

1.8 Radix Sort

Radix sort is not a comparative integer sorting algorithm as the other algorithms that we have seen so far. It instead sorts data lexicographically. In the case that the data is made of integers, Radix sort uses the digits of the integers to sort the data, and it uses Counting sort to help with sorting.

Consider the input array shown in *Figure 13 (a)*. In order to sort elements using radix sort, we use Counting sort using the last digit of each number as the sorting keys (see *(b)*). Sorting the elements, we obtain the array shown in *(c)*. As counting sort is stable sorting algorithm, the elements with the same key will appear in the same order. In the next step, we sort the elements by counting sort using the second to last digit as key (see *(d)*), and we obtain the array shown in *(e)*. Repeating the same procedure with the first digit as key (see *(f)*), we notice that there are few numbers without any number in the first place. In this case, we pad the numbers zeroes as their key. Once we sort the first digit using counting sort, the whole array is sorted as displayed in *(g)*.

Figure 14 shows the time and space complexities of Radix sort.

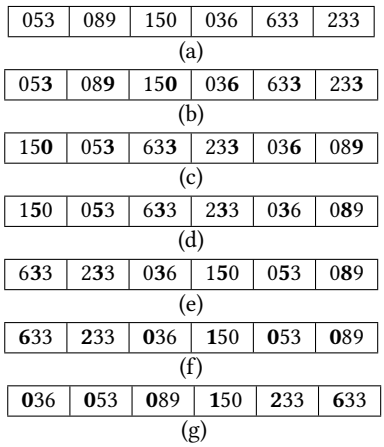


Figure 13: Radix sort visualization. (a) shows the input array. In (b), the sorting key is marked in boldface, and the resulting array after sorting is shown in (c). This procedure is repeated in (d)-(e) and (f)-(g). Notice the numbers have been padded with zeroes on the left, so that they can be sorted.

Time Complexity	
Best Case	$\Omega(nk)$
Average Case	$\Theta(nk)$
Worst Case	$O(nk)$

Space Complexity	
Worst Case	$O(n + k)$

Figure 14: Radix sort time[5] and space complexities[7].

1.9 Timsort

Timsort is a hybrid sorting algorithm. Timsort is a combination of Insertion sort and Mergesort. We basically divide an array into

sub-arrays of length RUN, where RUN is constant. Next, we apply binary insertion sort on each sub-array. Then, we merge the sub-arrays using Mergesort recursively to get the resulting sorted array. Timsort is extremely fast for nearly sorted data sequence. Timsort is also the default sorting algorithm in Java and Python, and it was implemented in 2002 by Tim Peter.

Consider the example shown in *Figure 15*, where an array of 10 elements is sorted using $RUN = 5$. *b* shows how the initial array *(a)* is divided into sub-arrays of size RUN. Using Insertion sort, we sort the sub-arrays as shown in *(c)*. Finally, the sub-arrays are merged back in sorted order using Mergesort. The resulting sorted array is shown in *(d)*.

Figure 16 shows the time and space complexities of Timsort.

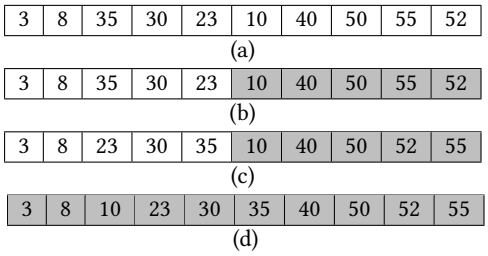


Figure 15: Timsort visualization. (a) shows the input array. (b) shows the array being split into sub-arrays of size RUN. (c) shows the sub-arrays after being sorted using Insertion sort. (d) shows the resulting sorted array after using Mergesort join back the sub-arrays in sorted order.

Time Complexity	
Best Case	$\Omega(n)$
Average Case	$\Theta(n \lg n)$
Worst Case	$O(n \lg n)$

Space Complexity	
Worst Case	$O(n)$

Figure 16: Timsort time[2] and space complexities[7].

2 IMPLEMENTATION

The sorting algorithms and the benchmark tool are written in Java using JDk 11. We have made available the source code of all implementations at the public GitHub repository [1].

2.1 Sorting Algorithms Correctness

As part of adding the sorting algorithms for this project, we also added correctness checks to make sure that the sorting algorithm is producing a sorted result and that the resulting array is a permutation of the original input array. This we we have confidence that the algorithm is correct. Additionally, we added smaller unit tests for each algorithm to make sure we cover the corner cases and validate that the algorithms handle unexpected inputs gracefully.

2.2 Test Cases

In order for us to compare the time complexity of different sorting algorithms, we had to come up with different test cases. We made available all the test cases used in the comparison at our GitHub repository [1].

The naming convention used for test files was "TestCase_DataSize.txt". For example, in the case of sorted in ascending order with 1000 elements the file name would be "testCases/SortedInAscendingOrderCase_1000.txt". This naming convention made it easy for us to identify test cases and parse the output report from the benchmark tool.

2.2.1 Test Cases Generation

Generating test cases is a tedious task if done manually; however, we automated the process of generating different test cases. We have 11 unique test cases that we ran on each algorithm. Additionally, we created the test case generation functions to generate any data size for any particular test case. Also, we tried to include test cases that cover the best, worst, and average cases for our sorting algorithms in order for the comparison to show advantages and disadvantages of different sorting algorithms based on test cases and data sizes.

2.2.2 Test Cases Generation Correctness

For every test case generator function we implemented unit tests to verify the correctness of the generated arrays. Given that the functions can handle any data size, we had to make sure that for large data sizes that correctness still holds. These unit tests gave us confidence that our test cases are what we expect them to be.

2.2.3 Test Cases Included in Results

We have a total of 11 unique test cases. For each test case we created multiple data sizes starting from 100 to 10,000,000 (10 was excluded since it was too small for any comparison). We were not

able to upload the 10,000,000 data size test cases as the files were too large.

The test cases used are:

(1) Sorted In Ascending Order
(2) Sorted In Descending Order
(3) Random Order
(4) Random Order - High Numbers on First Half and Low Numbers on Second Half
(5) Random Order - Low Numbers on First Half and High Numbers on Second Half
(6) Sorted In Ascending Order - High Numbers on First Half and Low Numbers on Second Half
(7) Sorted In Descending Order - High Numbers on First Half and Low Numbers on Second Half
(8) Nearly Sorted In Ascending Order
(9) Nearly Sorted In Descending Order
(10) Same Value Array
(11) Merge Sort Worst Case (This was added since the Merge Sort worst case was not covered by our generic test cases)

2.2.4 *Graphing Tool*

In order to represent the data from the benchmarks, we developed a graphing utility. The tool parses the resulting data files from the benchmarks, and it produces a graphical representation of the data. The data plotted is the execution times of the studied sorting algorithms for different input sizes and input cases. The utility has the ability to normalize the data using the results of any of the algorithms benchmarked. The data is organized by plotting the different input cases (the inputs cases mentioned in the previous section) in separate graphs. Each graph represents time of execution of the algorithms versus the size of the input data for an specific input case.

The tool was implemented using Python 3. It uses the matplotlib plotting library to draw the graphs. The source code of the utility is available in the repository of the project [1].

3 EVALUATION

3.1 Methodology

All of our implementations were written in Java. Initially we thought of measuring the performance of our implementations with a naive approach where we would measure the time either in milliseconds or nanoseconds for sorting the input array. We could for example simply use `System.currentTimeMillis()` or `System.nanoTime()`. However we quickly found out how unreliable the results would be, since we noticed really inconsistent results between runs. As Ponge[6] explains, this kind of benchmarking might be viable in programs written in statically compiled languages like C. However Java runs on a Virtual Machine and it uses *Just-in-time* compilation, so the first time the code is run it is actually being interpreted and then is compiled to native code, depending on the actual platform that is running. Furthermore, the VM tries to use all kinds of different optimization like loop unrolling, inlining functions or on-stack replacements, making it difficult to get consistent results.
We decided to use Java Microbenchmark Harness (JMH)[1] for measuring the performance of our implementations. JMH is an open source benchmarking tool part of the OpenJDK. Although it does not entirely prevent all common pitfalls and inconsistencies introduced by the JVM, it does help mitigating them.
Next step was to use the generated test cases files as inputs for our benchmarks. There are different modes to run benchmarks in JMH. We decided to measure the average time of an operation in microseconds, where an operation is sorting the array for any given algorithm and input array. In this mode, JMH considers an iteration to be a slice of time running as many operations as possible, it measures the time for each operation and averages it. In order to avoid some of the JIT inconsistencies and other JVM optimizations, JMH runs a few warm-up iterations. After that it runs, by default, 5 iterations where the results are actually recorded. For our measuring purposes we decided to run 3 five seconds warm-up iterations and 5 ten seconds actual iterations.

[1] http://openjdk.java.net/projects/code-tools/jmh/

The overall benchmark running time was over 36 hours. This was because we had large data sizes and due to the bad performance of some sorting algorithms in their worst case such as Bubble sort. We had to exclude Insertion and Bubble sort when running the test cases with a data size of 10,000,000.

3.2 Results

We ran the benchmark suite on a machine with an 4-cores and 8 logical processors @ 3.4Ghz and 24Gb of DDR4 RAM @ 3,401Mhz. The benchmark result reports can be found at our GitHub project link[1]. All the algorithms were run serially and nothing was parallelized. The benchmarks suite took around 36 hours where the majority of time was spent on Bubble and Insertion sorts since they were very slow. We ran data size of 10,000,000 for all the algorithms excluding Bubble and Insertions Sort; however, we did not include it since the results were linear with the data size and it was not adding any new information. We do have all the produces results and reports at our public GitHub repository [1]. In the graphs that we will use in this section the title of the graph will be the test case, they X-Axis is the data size and the Y-Axis is the time it took to sort the test case in microseconds.

3.2.1 *Interesting Observations*

In this section we will highlight some of the interesting observations that we found from our benchmark results since we cannot include all our graphs in the report due to space limitations; however, you can refer to the complete results and graphs at our public GitHub repository [1]. In *Figure 17* we can see how bubble sort and insertion sort are very inefficient when it comes to sorting data compared to other sorting algorithms as the size of data increases. This is both bubble sort and insertion sorts worst case. However, they also perform much worse than the other algorithms in their general case, so we had to remove them from some graphs in order to see how the other algorithms performed in comparison with each other.

Another interesting observation is shown in *Figure 18* where the test case is of an array that is already sorted in ascending order which is bubble sort, insertion sort and shell sort best case.

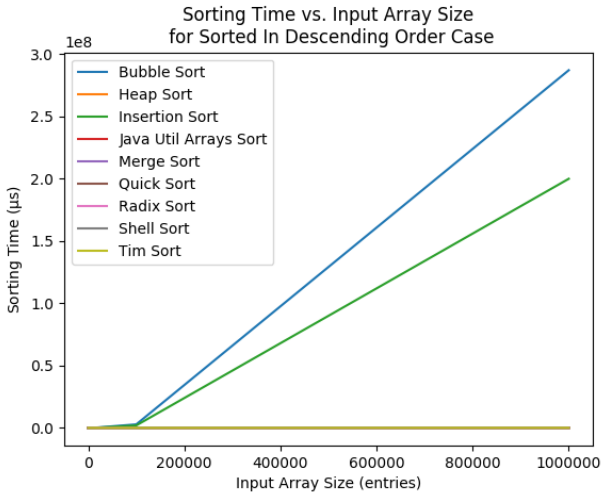


Figure 17: Input case: data is sorted in descending order. All algorithms are included.

We can observe that insertion sort is as fast as the Java Utility Arrays Sort implementation, which is highly optimized, and see how, as expected, other sorting algorithms (like heapsort) still take a longer time to finish, even though the array is already sorted. Since shell sort's best case time complexity is $\Omega(n \lg n)$, it performs similarly to other algorithms like heapsort for which this is an average case input. Another interesting point, was that bubble sort did not perform as well as expected, and we had to take it out of the plot in order to see the details of the other algorithms. Bubble sort was expected to perform in $\Omega(n)$ in this case, but it was the algorithm that took the longest by far for all the input sizes. This was probably caused by a not very efficient implementation of the algorithm.

In the case where the input consists of an array where all the values are the same (see *Figure 19*), Java Util Arrays Sort and insertion sort performed the best. This was expected, since this is a sub-case of the best case scenario of insertion sort (the input is already sorted). Moreover, the optimization of the Java Util Arrays Sort is shown in the results of this input case, since the algorithm performs even better than insertion sort. However, bubble sort did not perform as well as expected for being its best case. It took so long, we had to remove it from the graph in order to see the details. of the other algorithms.

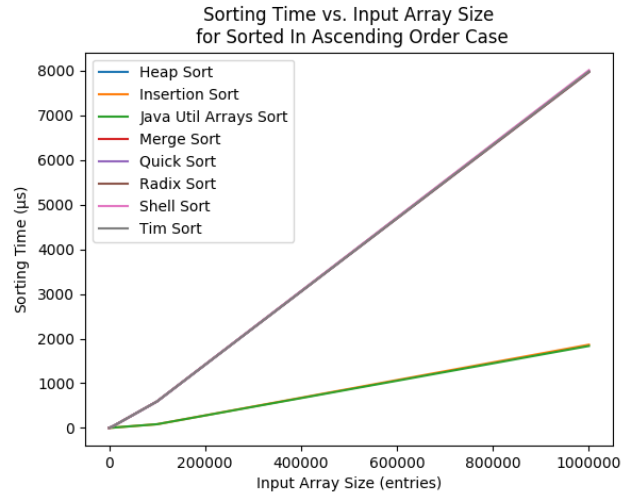


Figure 18: Input case: data is sorted in ascending order. All algorithms included, except bubble sort.

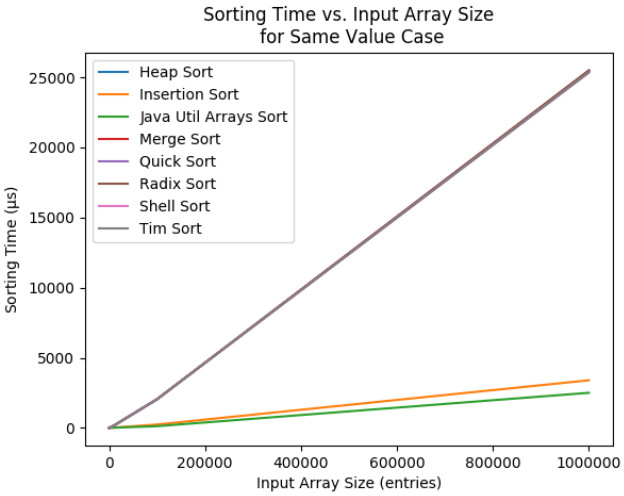


Figure 19: Input case: data is all the same value. All algorithms are included, except bubble sort.

In one of the input cases considered, we crafted the input array to be the worst case input for the mergesort algorithm. This input leads to the most comparisons possible when the mergesort algorithm is run. *Figure 20* shows this results, excluding bubble sort and insertion sort, so that mergesort's results are clearer. As expected, mergesort performed worst than the other algorithms for which this was an average case input. However, it did so by a small constant. This shows why mergesort is such a reliable algorithm. Mergesort has a consistent performance over all types of inputs, performing in $\Theta(n \lg n)$ even in the worst case.

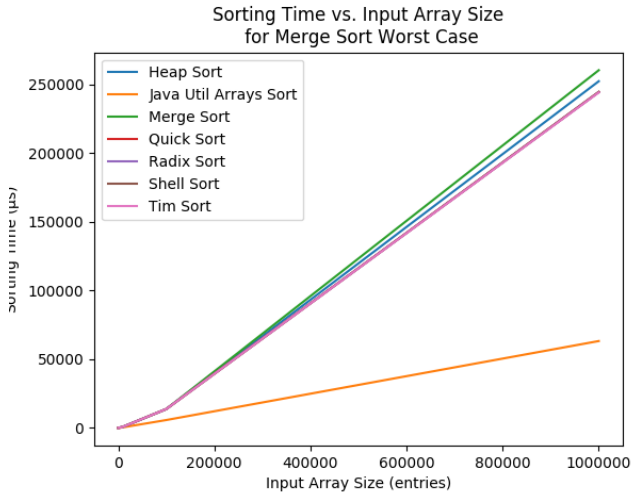
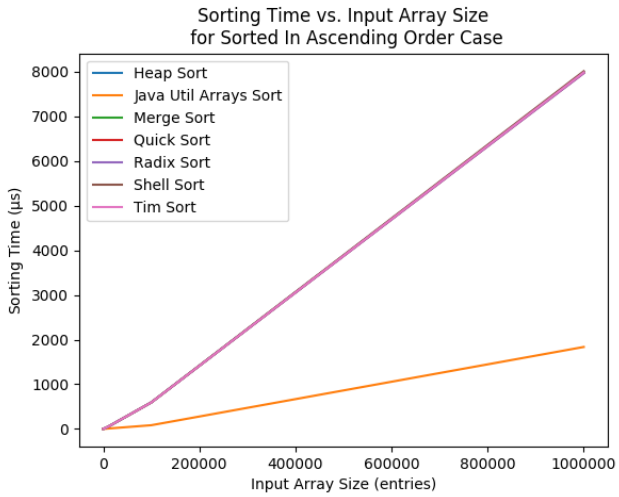
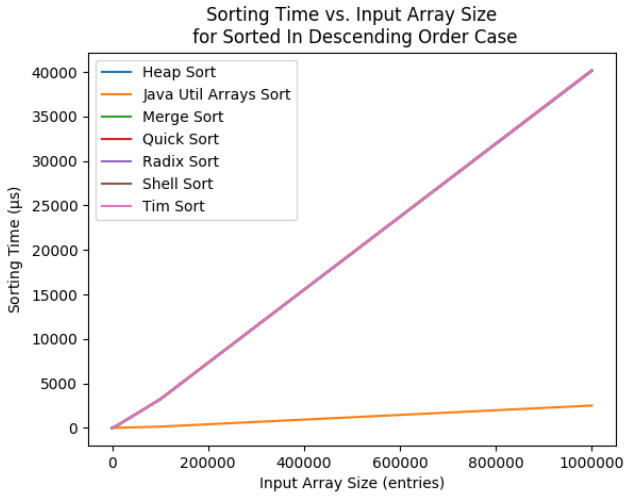


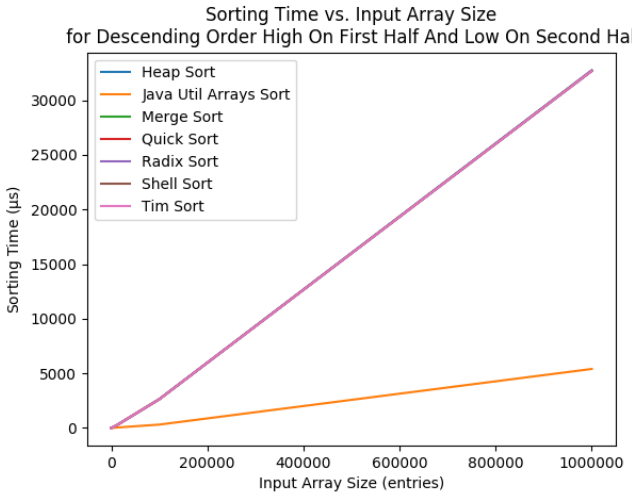
Figure 20: Input case: data generated for mergesort's worst case input. All algorithms are included, except bubble sort and insertion sort.

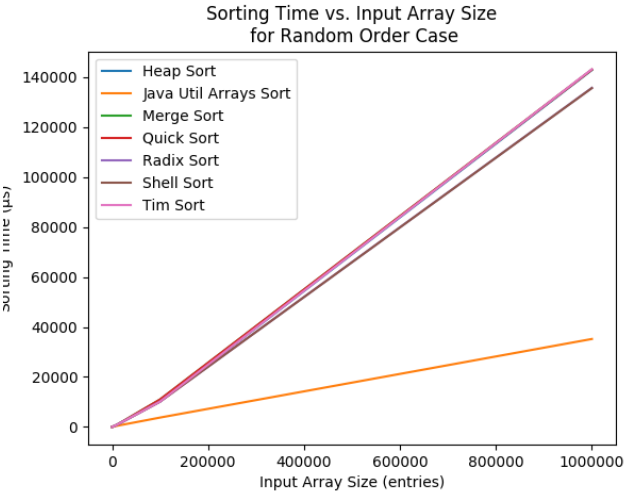
3.2.2 *Final Results Excluding Bubble and Insertion Sort*

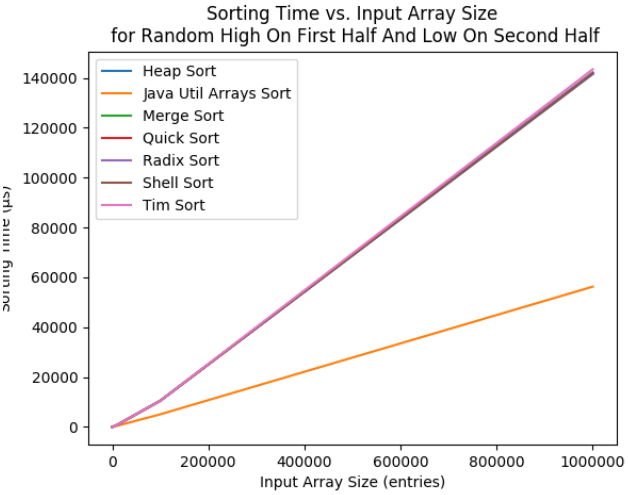
Below are the results for all the test cases for all sorting algorithms except Bubble Sort and Insertion Sort since they are too slow compared to other sorting algorithms and would hide the relative differences between the other algorithms if included.

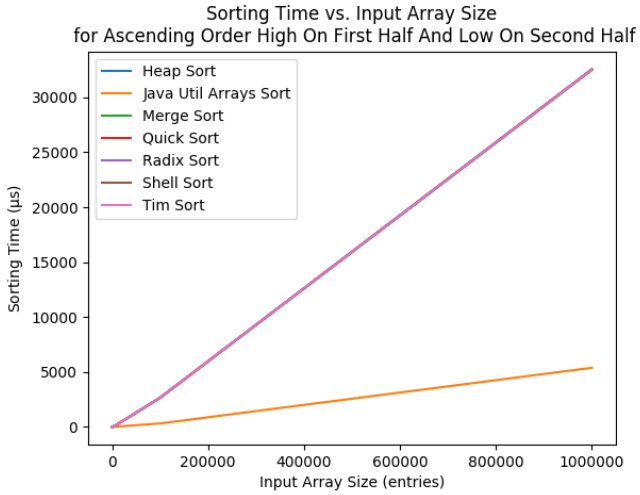


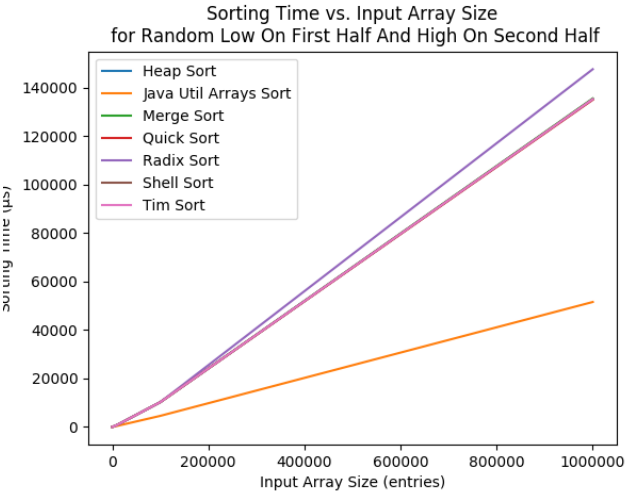

Sorting Time vs. Input Array Size
for Sorted In Descending Order Case
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
Sorting Time (µs)
Input Array Size (entries)

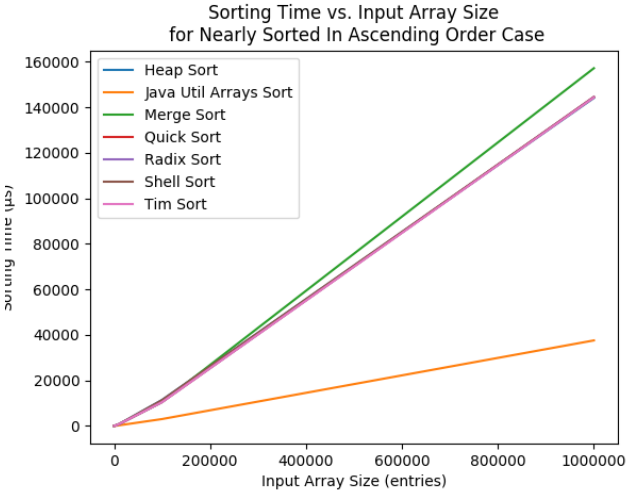

Sorting Time vs. Input Array Size
for Descending Order High On First Half And Low On Second Half
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
Sorting Time (µs)
Input Array Size (entries)

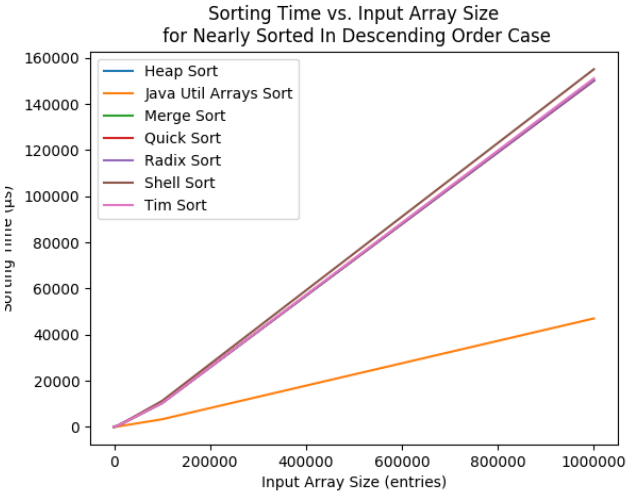

Sorting Time vs. Input Array Size
for Random Order Case
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
Input Array Size (entries)

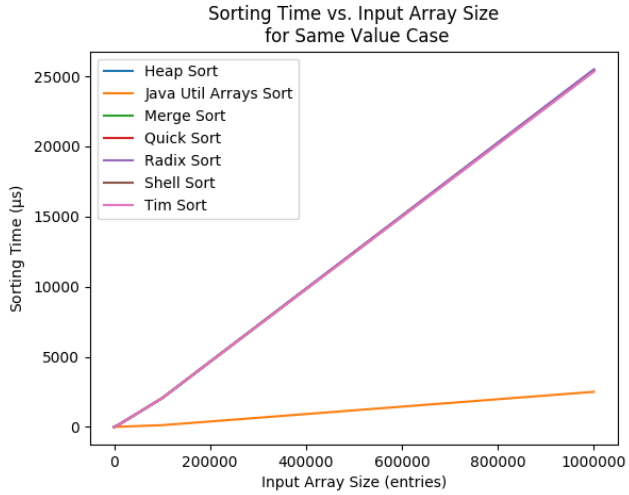

Sorting Time vs. Input Array Size
for Random High On First Half And Low On Second Half
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
Input Array Size (entries)


Sorting Time vs. Input Array Size
for Ascending Order High On First Half And Low On Second Half
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
Sorting Time (µs)
Input Array Size (entries)


Sorting Time vs. Input Array Size
for Random Low On First Half And High On Second Half
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
Input Array Size (entries)


Sorting Time vs. Input Array Size
for Nearly Sorted In Ascending Order Case
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
160000
140000
120000
100000
80000
60000
40000
20000
0
0
200000
400000
600000
800000
1000000
Input Array Size (entries)


Sorting Time vs. Input Array Size
for Nearly Sorted In Descending Order Case
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
160000
140000
120000
100000
80000
60000
40000
20000
0
0
200000
400000
600000
800000
1000000
Input Array Size (entries)


Sorting Time vs. Input Array Size
for Same Value Case
Heap Sort
Java Util Arrays Sort
Merge Sort
Quick Sort
Radix Sort
Shell Sort
Tim Sort
Sorting Time (µs)
25000
20000
15000
10000
5000
0
0
200000
400000
600000
800000
1000000
Input Array Size (entries)

4 CONCLUSION

We realized that is it not a trivial task to benchmark algorithms given all the hardware and software optimizations that we have today. It is very hard and almost impossible to turn all optimizations off; however, we believe that the JMH tool helped in mitigating some of the major pitfalls. Additionally, generating test cases and making sure that data is correct with any data size was also time consuming. Also, knowing what test data to use in order to see interesting results was also not trivial.

From our final results we can see that Java Utility Array sort is the fastest and most performant sorting algorithm compared to all the other algorithms written in Java which should encourage developers to be using it in their applications and not implement their own.

REFERENCES

[1] Project github link. https://github.com/azeid/SortingAndSearchingAlgorithms, April 2019.

[2] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort. *ArXiv:1805.08612*, 2018.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 3rd edition, 2009.

[4] JDK 11 Java SE 11. Java collection framework arrays sort. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html#sort.

[5] Ali Kazim. A comparative study of well known sorting algorithms. *International Journal of Advanced Research in Computer Science*, 8(1), 01 2017.

[6] Julien Ponge. Avoiding benchmarking pitfalls on the jvm. https://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html, July 2014. [Online; posted July-2014].

[7] Eric Rowell. Big-o cheat sheet. http://bigocheatsheet.com/.

[8] Vladimir Yaroslavskiy. The new optimized version of dual-pivot quicksort. http://mail.openjdk.java.net/pipermail/core-libs-dev/2018-January/051000.html, Jan 2018.