Sorting Algorithms Experiment Final Report

By: Team Toast

Alice Li, Skye Hobbs, Henry Stiff, Tinisha Davis

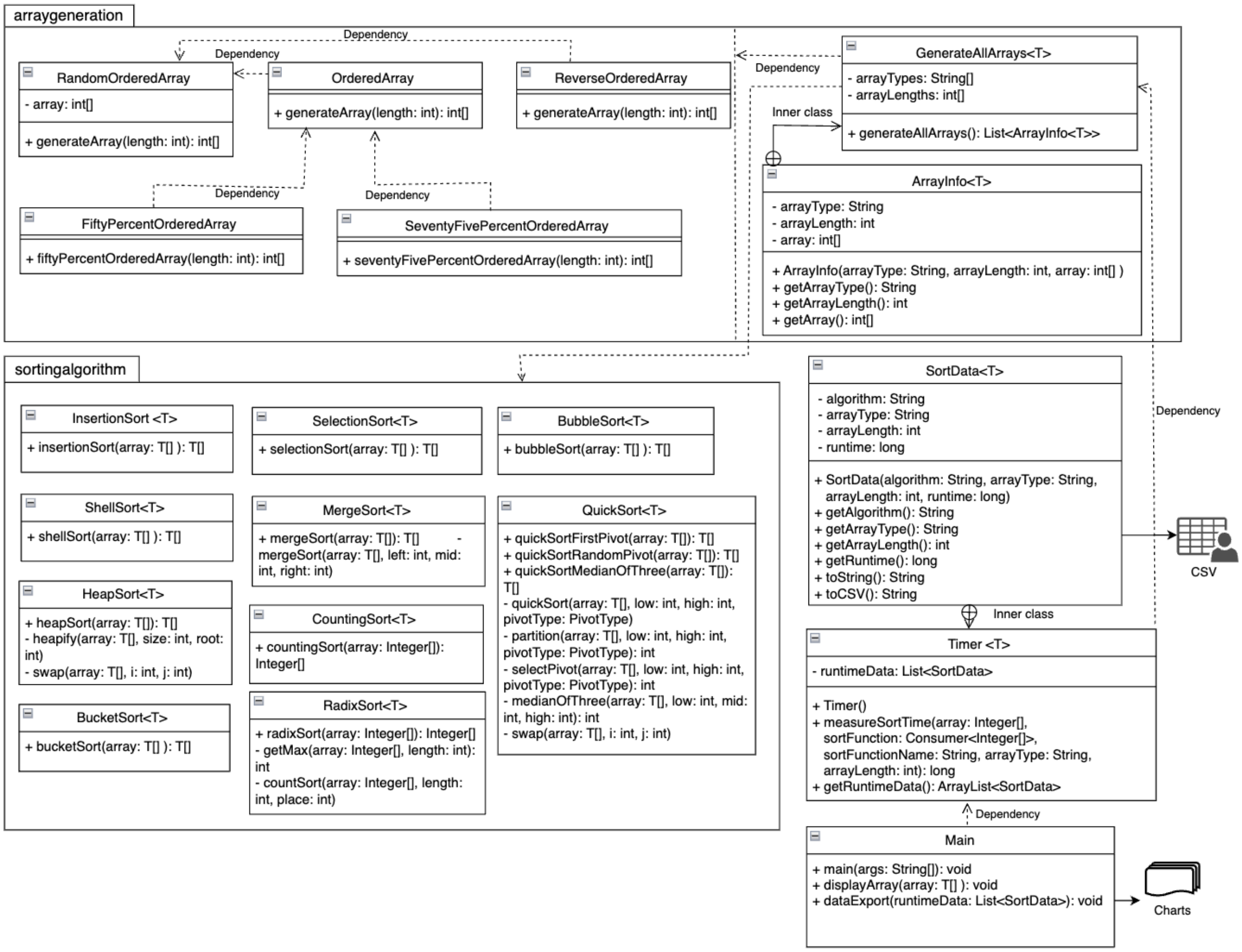
1. Design

1.1 Experiment Design Overview

The purpose of this project is to analyze and compare the efficiencies of various sorting algorithms. The algorithms tested include:

- Insertion Sort
- Selection Sort
- Bubble Sort
- Shell Sort
- Merge Sort
- Quick Sort (First-Pivot, Random Pivot, Median-of-Three Pivot)
- Heap Sort
- Counting Sort
- Bucket Sort
- Radix Sort

1.2 UML Diagram


arraygeneration
Dependency
Dependency
RandomOrderedArray
- array: int[]
+ generateArray(length: int): int[]
OrderedArray
+ generateArray(length: int): int[]
ReverseOrderedArray
+ generateArray(length: int): int[]
Dependency
Dependency
FiftyPercentOrderedArray
+ fiftyPercentOrderedArray(length: int): int[]
SeventyFivePercentOrderedArray
+ seventyFivePercentOrderedArray(length: int): int[]
Dependency
GenerateAllArrays<T>
- arrayTypes: String[]
- arrayLengths: int[]
+ generateAllArrays(): List<ArrayInfo<T>>
Inner class
ArrayInfo<T>
- arrayType: String
- arrayLength: int
- array: int[]
+ ArrayInfo(arrayType: String, arrayLength: int, array: int[])
+ getArrayType(): String
+ getArrayLength(): int
+ getArray(): int[]
sortingalgorithm
InsertionSort <T>
+ insertionSort(array: T[]): T[]
SelectionSort<T>
+ selectionSort(array: T[]): T[]
BubbleSort<T>
+ bubbleSort(array: T[]): T[]
ShellSort<T>
+ shellSort(array: T[]): T[]
MergeSort<T>
+ mergeSort(array: T[]): T[] - mergeSort(array: T[], left: int, mid: int, right: int)
QuickSort<T>
+ quickSortFirstPivot(array: T[]): T[]
+ quickSortRandomPivot(array: T[]): T[]
+ quickSortMedianOfThree(array: T[]): T[]
- quickSort(array: T[], low: int, high: int, pivotType: PivotType)
- partition(array: T[], low: int, high: int, pivotType: PivotType): int
- selectPivot(array: T[], low: int, high: int, pivotType: PivotType): int
- medianOfThree(array: T[], low: int, mid: int, high: int): int
- swap(array: T[], i: int, j: int)
HeapSort<T>
+ heapSort(array: T[]): T[]
- heapify(array: T[], size: int, root: int)
- swap(array: T[], i: int, j: int)
CountingSort<T>
+ countingSort(array: Integer[]): Integer[]
BucketSort<T>
+ bucketSort(array: T[]): T[]
RadixSort<T>
+ radixSort(array: Integer[]): Integer[]
- getMax(array: Integer[], length: int): int
- countSort(array: Integer[], length: int, place: int)
SortData<T>
- algorithm: String
- arrayType: String
- arrayLength: int
- runtime: long
+ SortData(algorithm: String, arrayType: String, arrayLength: int, runtime: long)
+ getAlgorithm(): String
+ getArrayType(): String
+ getArrayLength(): int
+ getRuntime(): long
+ toString(): String
+ toCSV(): String
CSV
Dependency
Inner class
Timer <T>
- runtimeData: List<SortData>
+ Timer()
+ measureSortTime(array: Integer[], sortFunction: Consumer<Integer[]>, sortFunctionName: String, arrayType: String, arrayLength: int): long
+ getRuntimeData(): ArrayList<SortData>
Dependency
Main
+ main(args: String[]): void
+ displayArray(array: T[]): void
+ dataExport(runtimeData: List<SortData>): void
Charts

1.3 Input Array Design

The arrays are generated programmatically with the following design:
14 array sizes in powers of 2, starting from (4) to (32,768).
Random integers between 0 and 40,000 (duplicates allowed).
For each array size, five variations are tested:

- Random Ordered Array
- Ordered Array
- Reverse Ordered Array
- 50% Ordered Array
- 75% Ordered Array

1.4 Metrics

The metrics chosen for evaluating the sorting algorithms are:
Input Size: Array size ranging from 4 to 32,768.
Execution Time: Recorded in microseconds (nanoseconds / 1000) for better precision.

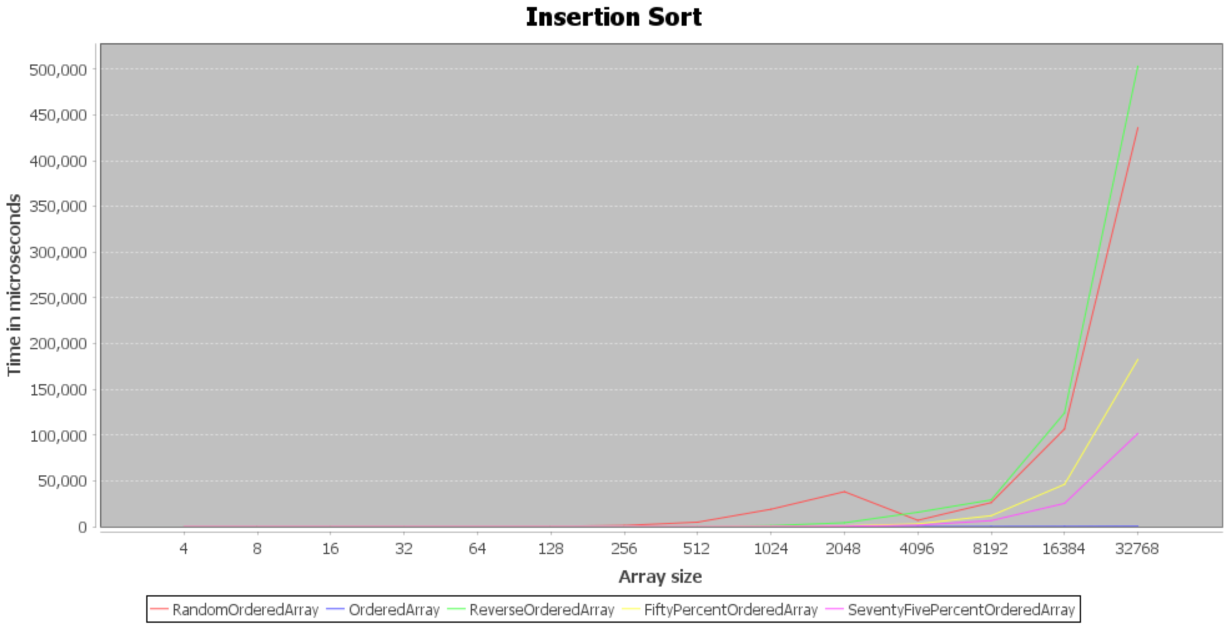
1.5 Results Display

The results are visualized in the following ways:

- Runtime data will be exported to text files.
- Charts will be generated using Java libraries (e.g., JFreeChart) or external tools like Excel.
- Each sorting algorithm will have 5 plots, one for each input type, totaling 60 charts.

2. Experimental Results and Observations

Insertion Sort



Insertion sort is a stable sorting algorithm that performs efficiently on small datasets but becomes increasingly inefficient as the dataset size grows. This theoretical understanding is confirmed by empirical results, which show that while insertion sort is effective for smaller input sizes, its performance degrades significantly with larger datasets.

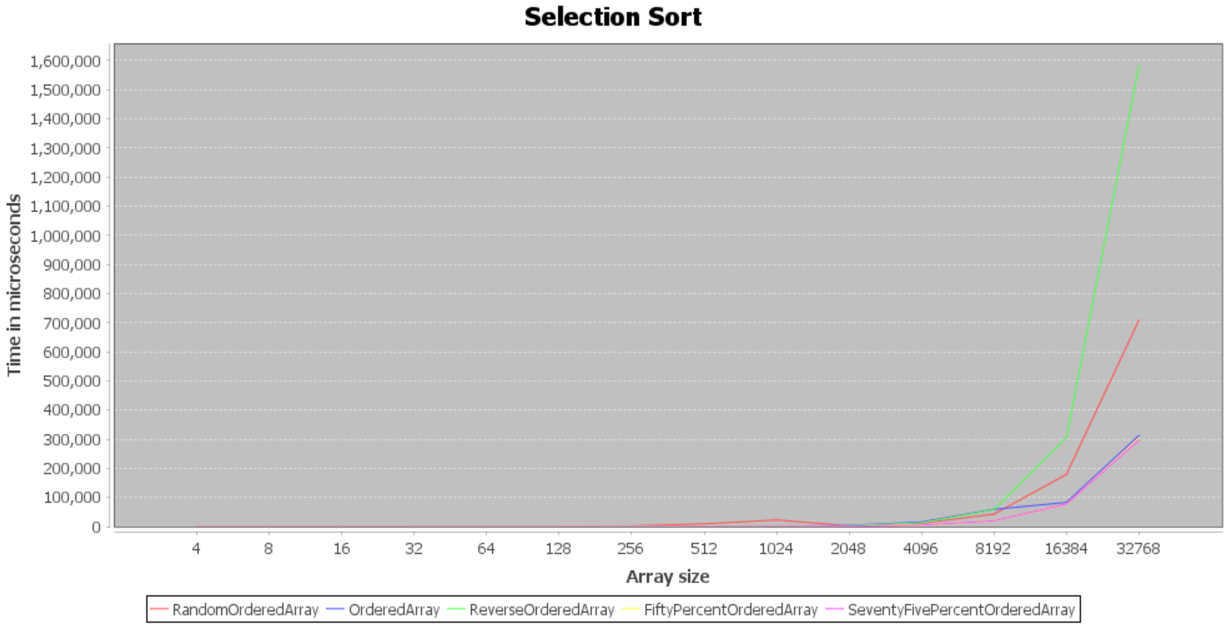
One of the main reasons for implementing insertion sort is its simplicity and stability. It is easy to implement and works well on nearly ordered datasets. The empirical results support this, as both the fully ordered array and the seventy-five percent ordered array demonstrate time efficiency despite the dataset size. This makes insertion sort a viable option for scenarios where data is already sorted or nearly sorted. Additionally, the algorithm was implemented using a generic type to allow for flexibility in future projects.

Among the different input types tested, the randomly ordered array exhibited the most variable performance. This variability arises because insertion sort’s efficiency is highly dependent on the initial order of the data. While ordered datasets allow for a best-case linear time complexity, randomly ordered data leads to frequent comparisons and swaps, making performance less efficient.

The impact of input array types on sorting time can be observed in the runtime graph, which remains relatively uniform, except for a noticeable skew in the randomly ordered array’s data. This uniformity results from insertion sort’s inherent simplicity. When working with ordered or nearly ordered data, insertion sort executes efficiently, whereas random data introduces greater fluctuations in sorting time.

The theoretical complexity of insertion sort is $O(n^2)$ in the average and worst cases, while it achieves $O(n)$ linear time when dealing with fully ordered data. This expected behavior is reflected in the empirical runtime data. The ordered array follows a linear time complexity, whereas all other arrays exhibit an exponential increase in runtime as dataset size grows. These results reinforce the understanding that insertion sort is most effective for small or nearly sorted datasets but becomes impractical for large, unordered datasets.

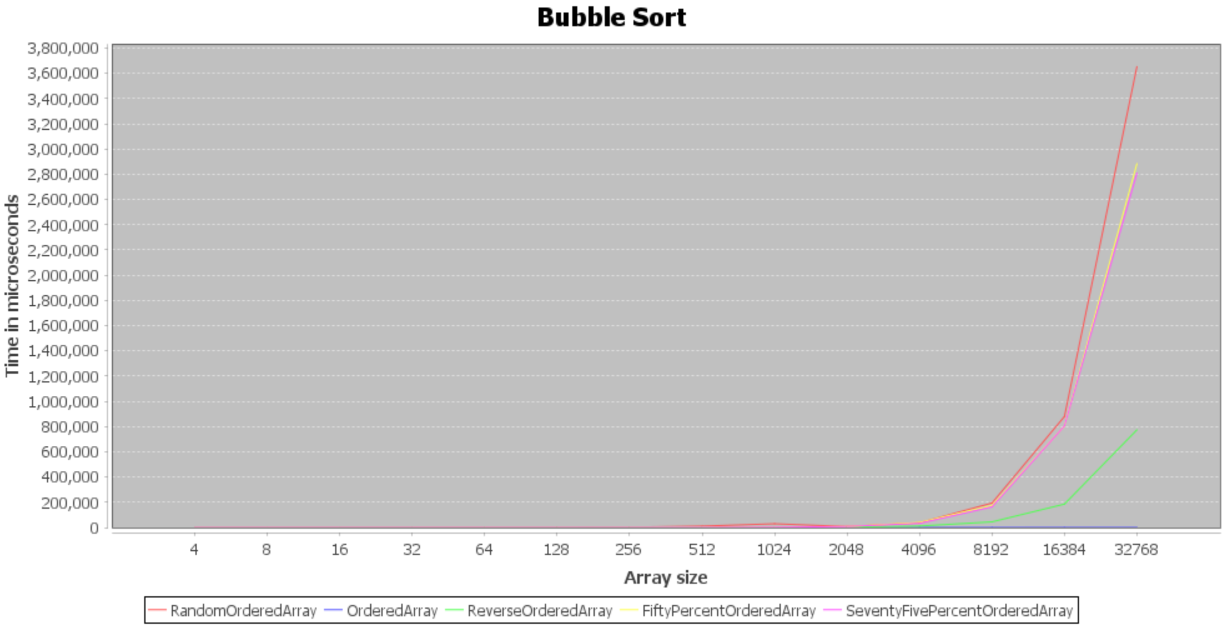
Selection Sort



For an ordered array the empirical results align with the theoretical expectation that Selection Sort will approach O(n) if the array is already sorted. The same is true for the other arrays, their empirical results confirm the theory that Selection sort will have a time complexity of O(n^2). Selection Sort algorithms are great for teaching algorithm analysis because it has a relatively simple implementation. It would also be efficient for small datasets and in environments where using minimal memory is preferred. I don't think Selection Sort is a good algorithm choice for large datasets because it runs at O(n^2) and there are other algorithms that can run a large dataset much faster. While it performs well on already sorted data it doesn't do great with nearly sorted data because it will still run through the entire array, being nearly sorted doesn't make Selection Sort any faster. It is also an inherently unstable sorting algorithm so it isn't suited for any implementation that requires stability.

As the chart shows there is a significant variance between the best-case and worst-case scenarios that were tested. The best-case was an ordered array at O(n) while the worst-case was the reversed ordered array at O(n^2). The ordered array is significantly faster than the algorithm realizes the array is sorted and is able to minimize the number of swaps performed. While the revered ordered array has to perform the absolute maximum number of comparisons and swaps. The random and partially ordered arrays fell in between the two extremes with their runtimes depending on the percentage of partial sorting and the randomization of the array. The empirical runtime of the ordered array shows almost linear growth, it approaches O(n), but I think it could technically be considered O(n^2). The empirical runtime of the other arrays shows quadratic growth which does confirm the expected complexity of O(n^2).

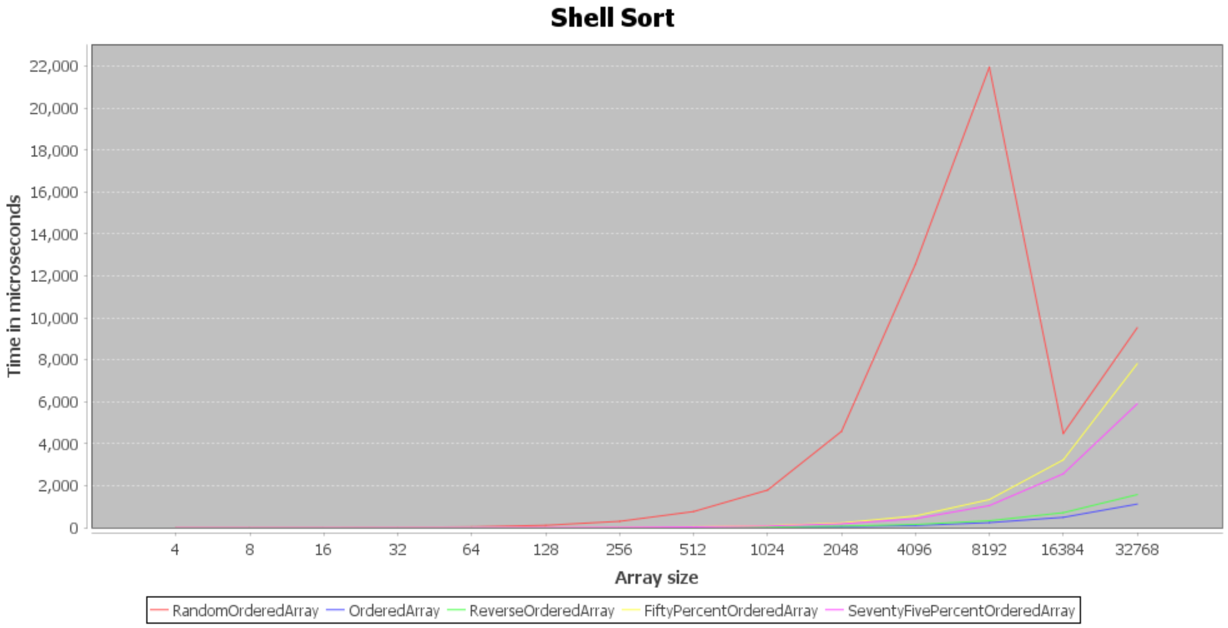
Bubble Sort



The empirical results for all of the arrays matched their respective theoretical results. For ordered arrays the best-case was O(n), while all the other arrays' best-case was O(n^2). I did have the structure of the nested loops set up differently at first but after running it a couple times I found using a break statement was what worked best for this experiment. I was having trouble with accuracy and I'm not exactly sure why but once I changed the nested loops up a bit to include a break statement it ran just fine. Similar to Selection Sort, Bubble Sort is easy to understand and to implement therefore, it is regularly used to teach fundamental sorting concepts. It is not always the best option for sorting algorithms but it can be efficient for very small datasets.

Originally, when coding this sorting algorithm I did have the structure of the nested loops set up differently at first but after running it a couple times I found using a break statement was what worked best for this experiment. I was having trouble with accuracy and I'm not exactly sure why but once I changed the nested loops up a bit to include a break statement it ran just fine. The chart shows that the performance varies significantly between the ordered array and reverse ordered array scenarios. This high variance shows that this is not a good sorting algorithm to use when the input order is unpredictable. For ordered arrays this algorithm will run in linear time, O(n), while reserved ordered arrays significantly impact the sorting time resulting in a quadratic run time, O(n^2). The random and partial ordered arrays fall in between the two extremes but the exact runtime will depend on the percentage of partial sorting and the randomization of the array. If one array is randomized to be almost completely reverse ordered versus one array randomized to be almost completely sorted, then we would see a bigger variance between the two. The empirical runtime with an ordered array shows linear growth which confirms the expected best-case complexity of O(n). The other arrays' empirical runtime shows quadratic growth which confirms the expected complexity for the worst-case, O(n^2).

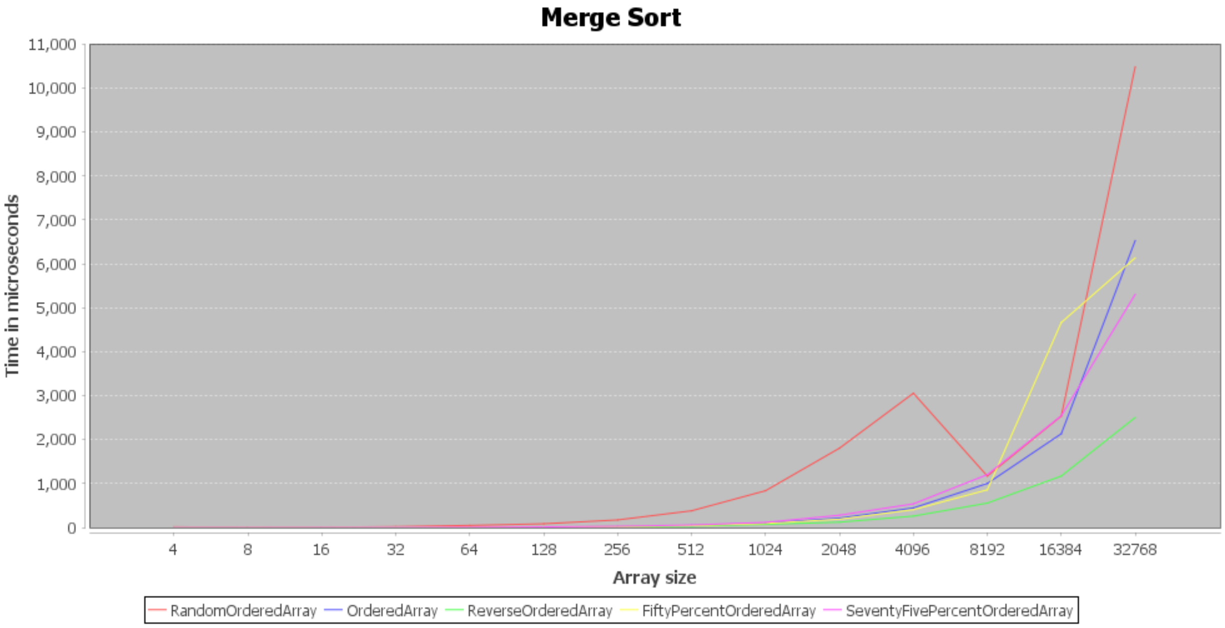
Shell Sort



Each time this test is run there seems to be a large increase in runtime leading up to the size 2^{13} random array. This makes sense for shell sort since the sequence of gap sizes that are chosen affects the runtime, and inefficient gap sizes can cause the algorithm to do inefficient passes over the array. The gap sequence used for this implementation was $n/2^k$ rounded down where k is the number of passes. This was the most simple to implement although it may have caused the notable increase in runtime at specific points. Other gap sequences are more efficient in some cases however they can be more difficult to implement and there are different cases where they slow down. For the sake of simplicity this one was chosen, and the chart shows how the gap sequence can make a difference in runtime. The point where the random ordered array has size 2^{13} always occurs so it isn't really variance, however the random ordered array had the most variable performance.

After multiple tests, there isn't much variance in any of the array types, but of the small amount of variance that there is, the random ordered array input causes the most. The random ordered array has the most significant impact on sorting time because it has the least amount of elements already in order. Because of that, the algorithm has to make more passes and increase runtime. The fifty percent ordered array has the next most significant runtime followed by the seventy five percent ordered array since these arrays have partially random elements that need to be sorted. The expected complexity of this shell sort implementation is $O(n^2)$. However, the results show that the runtime in some specific cases can significantly depend on the gap sequence chosen. Shell sort works similarly to insertion sort, however the runtimes of the two algorithms are significantly different in this experiment which shows the difference that sorting with a gap makes in speeding up the runtime.

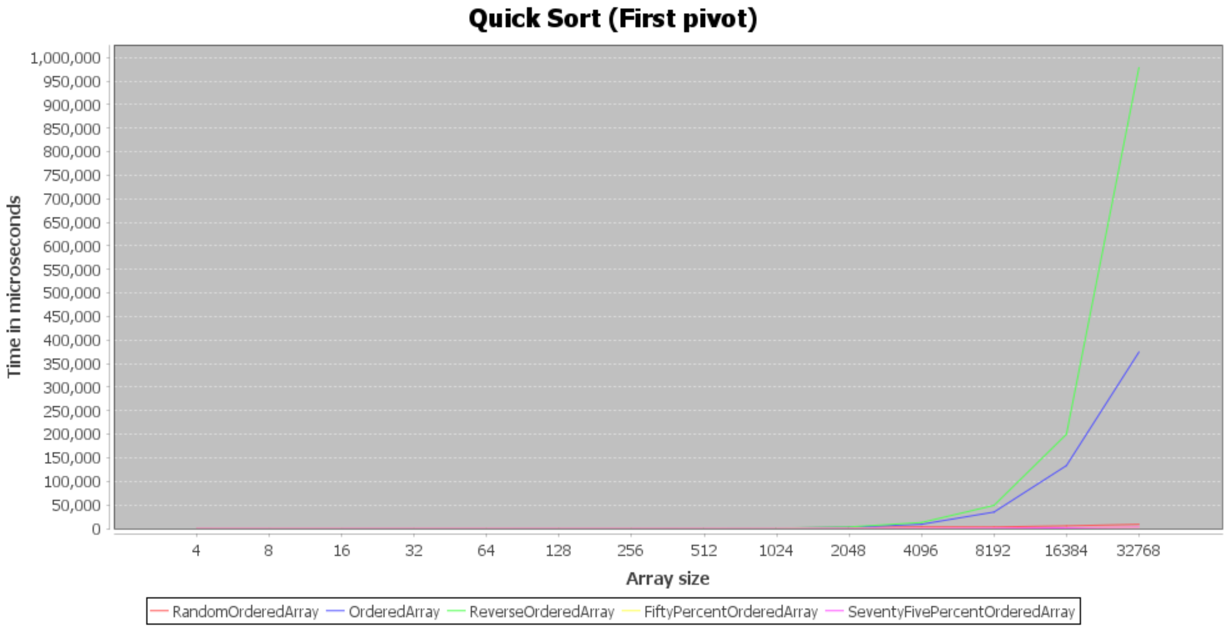
Merge Sort

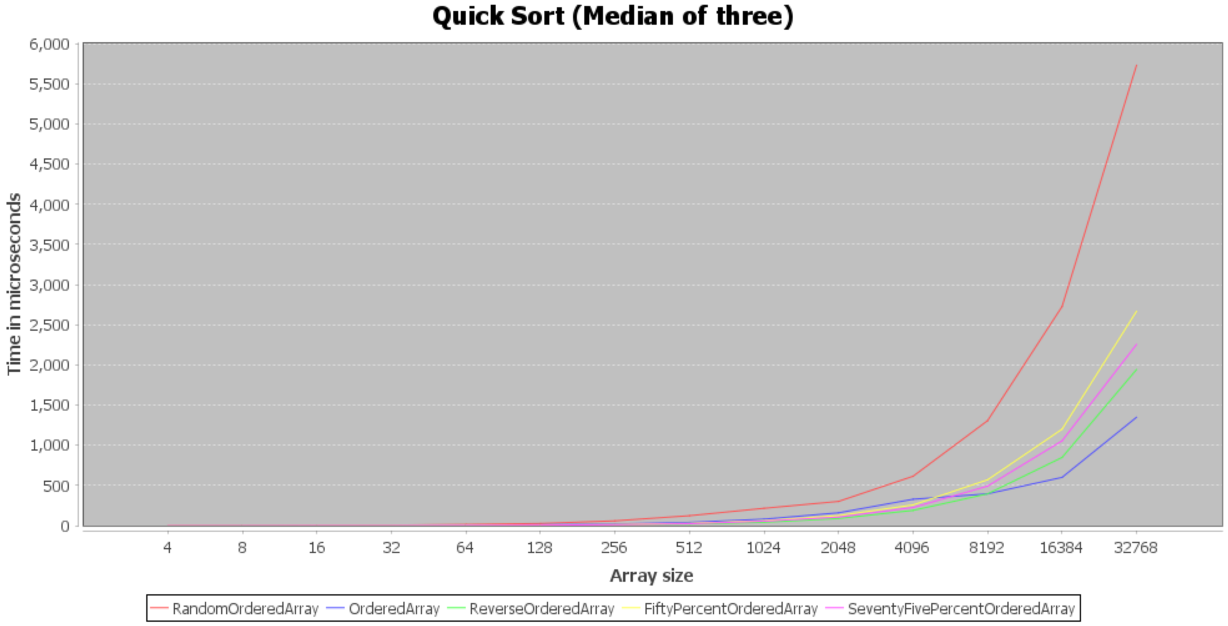


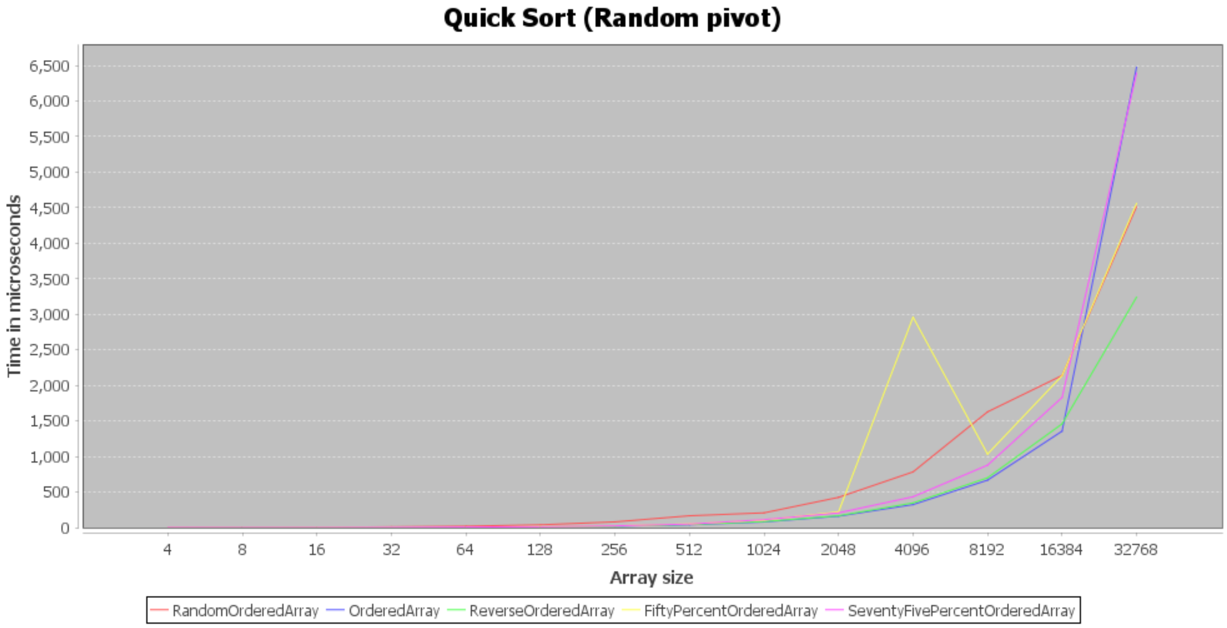
There is little variance in the graph which makes sense considering theoretically there aren't any specific data sets that will cause merge sort to slow down significantly. This implementation for merge sort allows for generic values as input that implement the comparable interface. This was so that in the future we could reuse this algorithm on other projects. The variables for left, right and middle indices in the implementation have descriptive names so the code is easier to understand. Implementing merge sort and using abbreviated variable names ended up making it confusing even though it used less lines.

The most variable performance is in the random ordered array. The random ordered array caused the most impact on sorting time because it frequently required more arrays to be swapped. The other array types have similar runtimes, although on this particular run the graph shows that the fifty percent ordered array took more time than the random ordered array at one point which did not occur on every run of the experiment and is likely due to the random values selected in both arrays on this run. The expected complexity for merge sort is O(n log(n)). The chart shows the runtime increasing at a similar rate for all array types. The runtime is exponentially faster than some other algorithms that were expected to have large runtimes like bubble sort. The runtime is also slower than algorithms like radix sort and counting sort which were expected to be fast.

Quick Sort (3 variations)







Empirical Results vs. Theoretical Results

Theoretically, QuickSort should have the following time complexities:

- Best Case Complexity: $\Omega(n \log n)$ when partitions are perfectly balanced.
- Average Case Complexity: $\Theta(n \log n)$, occurring with reasonably balanced partitions.
- Worst Case Complexity: $O(n^2)$, happening with unbalanced partitions

Our empirical results align with the theoretical expectations:

For example, in First Pivot QuickSort, runtime increases quadratically for reverse-ordered arrays because the largest element is always chosen as the pivot. This forces almost every element to be shifted in each partitioning step, leading to $O(n^2)$ complexity.

For ordered arrays, the smallest pivot is selected in each iteration, but the elements remain in their original positions. This also results in $O(n^2)$ complexity, but the runtime is slightly better than that of reverse-ordered arrays, as fewer swaps occur.
For randomly ordered arrays, the complexity remains $\Theta(n \log n)$ as expected, since partitions tend to be well-balanced.
For Median Pivot and Random Pivot, the results closely follow the theoretical $\Theta(n \log n)$ complexity. Both approaches significantly reduce sorting time compared to First Pivot, as they help maintain more balanced partitions and avoid the worst-case scenario.

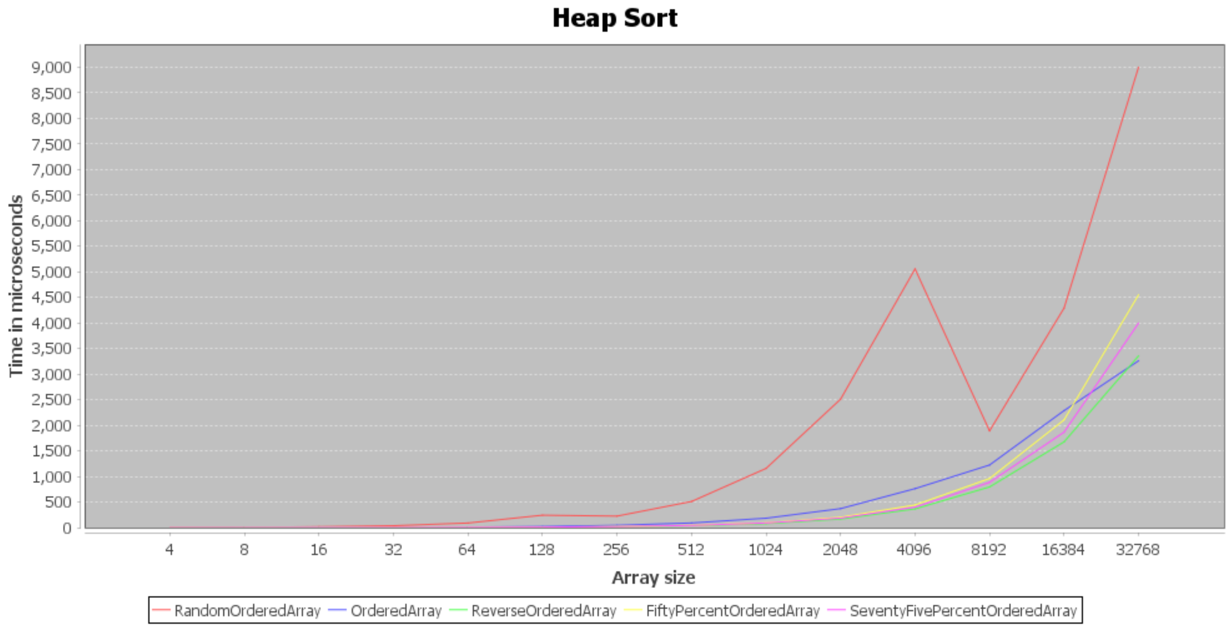
We were surprised that Median Pivot QuickSort ran faster on a reverse-ordered array than on a randomly ordered array. This happens because the median-of-three pivot selection tends to pick a value near the center, creating balanced partitions even in a reverse-sorted array. In contrast, while random arrays usually lead to $\Theta(n \log n)$ performance, occasional unbalanced partitions can cause slight inefficiencies, making the runtime slightly higher in some cases.

Reasons for Implementing Specific Sorting Algorithms

During the implementation, we initially used a recursive approach for QuickSort. However, when sorting large arrays, this led to stack overflow errors, as the recursive calls exceeded the system's stack size limit. This issue was particularly problematic when dealing with highly unbalanced partitions, such as when the pivot consistently selected the smallest or largest element, leading to deep recursive call stacks.

To address this, we switched to an iterative implementation of QuickSort. The iterative version eliminates deep recursion by using an explicit stack (e.g., a manually managed stack structure) instead of relying on function calls. This approach prevents stack overflow and improves stability and memory efficiency, especially for large datasets.
Additionally, the iterative method provides more control over stack usage, making it a more reliable choice when working with constrained memory environments or large-scale sorting operations.

Heap Sort

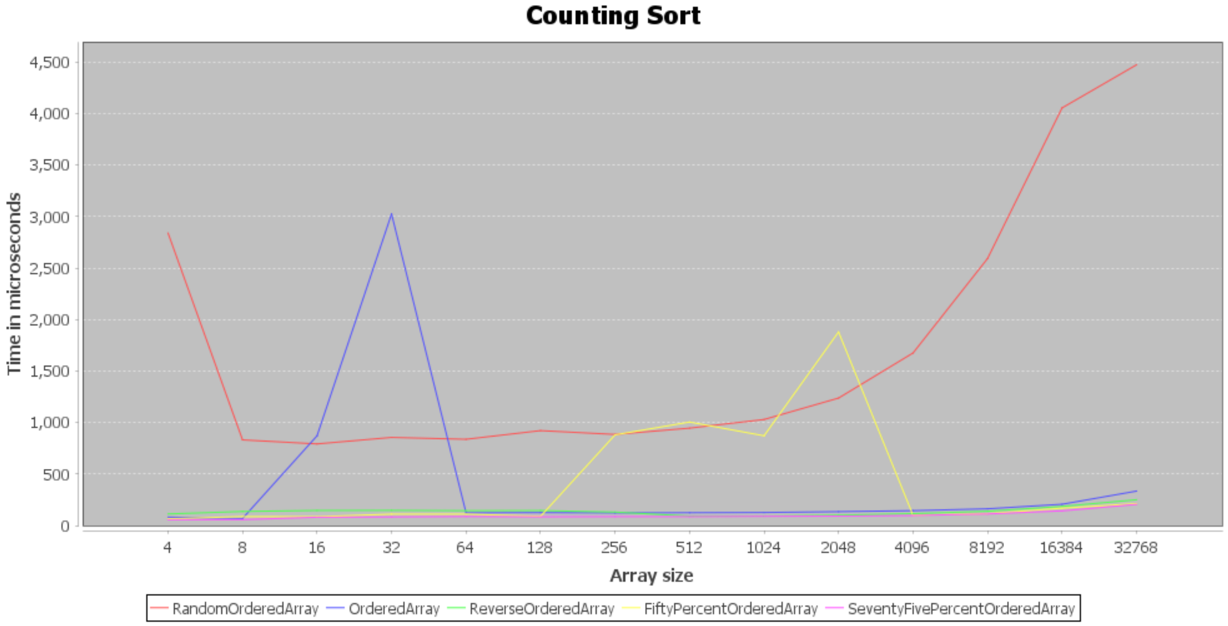


HeapSort consistently achieves O(n log n) performance, making it a reliable choice for worst-case stability, making it suitable for real-time systems and security-sensitive applications. Its in-place sorting mechanism eliminates the memory overhead seen in MergeSort, though frequent memory access and poor cache locality often make it slower in practice.

Despite its stability, empirical results showed a noticeable runtime bump at an array size of 4096 for randomly ordered inputs. This anomaly is likely due to CPU cache effects and memory hierarchy limitations. Since HeapSort relies heavily on tree restructuring and swapping elements between heap levels, excessive cache misses can significantly slow down performance. HeapSort runs consistently across input types, but random arrays take longer due to more swaps and cache inefficiencies. Reverse-ordered, ordered, and partially ordered arrays show similar runtimes, as

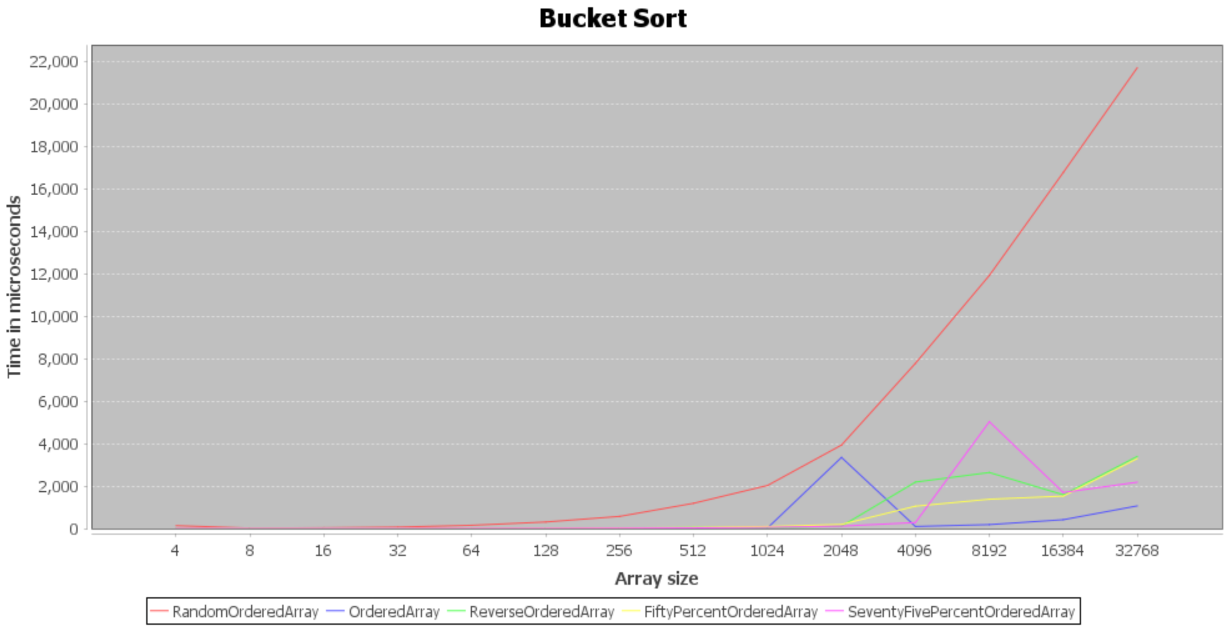
HeapSort always rebuilds the heap. In contrast, sorting algorithms with better locality of reference, such as QuickSort, tend to be less affected by such thresholds.

Counting Sort



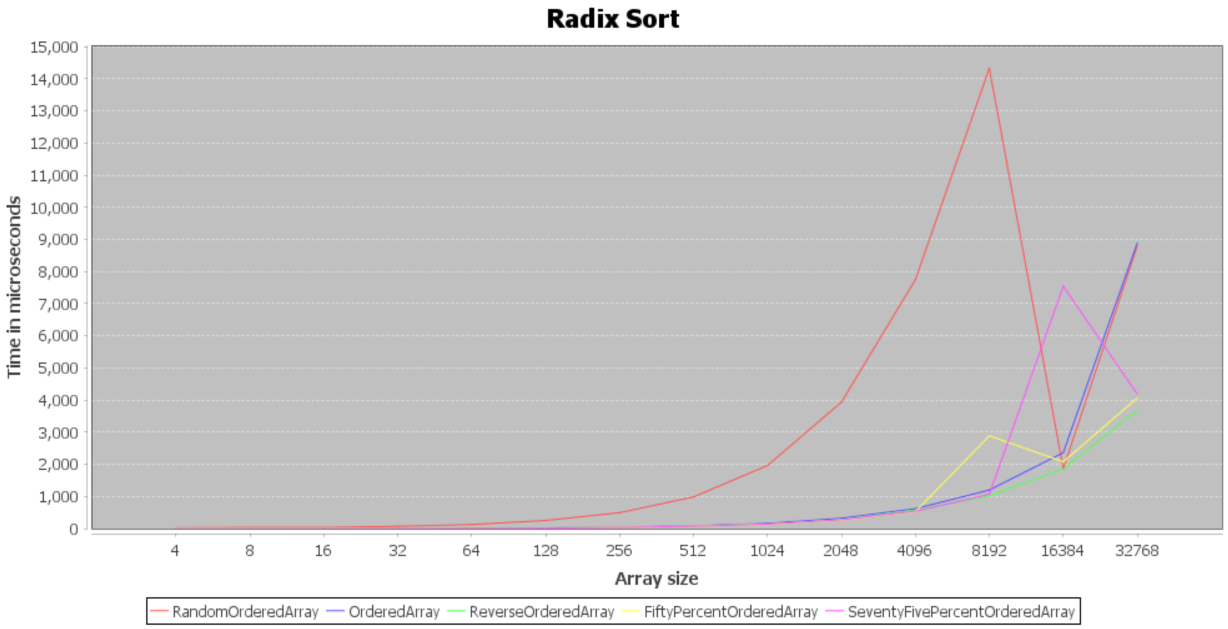
Theoretically, counting sort should take more time based on the maximum value in the data set. In the results we can see that the larger data sets took longer to sort which is likely due to there being a higher chance of there being a wider range of inputs and therefore a larger maximum value. Counting sort relies on the properties of positive integers and as such only works on them. For other algorithms we allowed generic arrays as inputs, however counting sort only allows integers. Since the rest of the algorithms in the project accept generic types, primitive types cannot be used. This algorithm was originally using the primitive type “int” since it should have less overhead, but was changed to the wrapper type “Integer” in order to be compatible with the generic algorithms. On subsequent runs of the experiment, the random ordered array frequently has the most deviation in results. The variations are caused by there randomly being a high maximum number in the input array. There also seems to be more variation in the lower sized arrays which is likely because one of the few numbers could randomly be large, requiring a large internal array to be created for a small dataset. The random ordered array usually has the greatest sorting time, which is expected. The ordered and partially ordered arrays have less runtime which makes sense because they require less operations to be sorted. The expected complexity for counting sort is O(N+M) where M is the difference between the maximum value in the data set. The runtime that we see in the worst case, which is usually the random ordered array, is close to linear and approximately matches the expected complexity.

Bucket Sort



The empirical results are generally consistent with the theoretical expectation that Bucket Sort should have a run time of O(n). However, the deviations suggest that the distribution of the values within the buckets may not be perfectly uniform, especially when the input size grows. This would lead to some buckets being more populated than others which would increase the time spent sorting those individual buckets. This sorting algorithm is very effective if the input data is uniformly distributed and offers the potential for linear time complexity, which makes it very fast granted the ‘right’ data be used. I had to change the number of buckets a few times before I was able to achieve a more even distribution of buckets being used. While I believe there is a way to change this coded algorithm to handle non-uniform data better I struggled to find that solution. The reversed ordered array is where we see the most variable performance and this is because Bucket Sort’s performance is highly dependent on the distribution of the input data. If the data is evenly distributed it will perform very well. If the data is clustered or not evenly distributed then we see the performance degrade significantly. This algorithm will generally run in linear time if the array is ordered, although a random ordered array at times is only slightly slower. Partial ordered arrays are having about the same impact as the other and it runs a bit slower than the random ordered array. The slowest runtime and at times shows nonlinear behavior on the chart. The empirical runtime shows a generally linear trend which matches our expected complexity of O(n).

Radix Sort



Radix sort operates by sorting elements digit by digit, with a time complexity of O(k*(n+b)), where n represents the number of elements, k is the number of digits in the largest number, and b is the base used for digit grouping (e.g., base 10). Theoretically, radix sort proves to be highly efficient when handling fixed data sizes, particularly when k remains small. This theoretical efficiency is supported by empirical evidence, where the dominant factor in runtime is k, rather than the initial order of the given array.

Radix sort is chosen for implementation due to its efficiency in cases where it can be applied, particularly when working with fixed data sizes. Unlike comparison-based sorting algorithms, which often have a worst-case complexity of O(n log n), radix sort runs in linear time under suitable conditions.

Similar to counting sort, radix sort can only be implemented efficiently with primitive data types. Initially, the algorithm was designed to work with int values, but due to compatibility requirements within the project, it had to be adapted to use Integer objects instead. This modification was necessary to maintain consistency with the rest of the project.

Among the different input cases, the randomly ordered array exhibited the most variable performance. The impact of the input array type on sorting time can be best understood by analyzing k, the number of digits in the maximum element. In both ordered and reverse-ordered arrays, k changes in a consistent manner, leading to relatively stable performance. However, in randomly ordered arrays, k exhibits more significant fluctuations, particularly when the average case results in the highest k values. This variability directly influences sorting time, making random inputs the most unpredictable case for radix sort. The theoretical time complexity of radix sort, O(k*(n+b)), is reflected in empirical runtime observations. As n increases, the runtime exhibits a proportional growth, consistent with the expected complexity. Additionally, fluctuations in runtime are observed when k varies, particularly in cases where the dataset contains numbers with differing digit lengths.

3. Discussion and Conclusions

3.1 General Observations

From our analysis, the performance of sorting algorithms varied significantly based on dataset size, scalability, and consistency.

For smaller datasets, Insertion Sort, Selection Sort, and Bubble Sort performed reasonably well due to their simplicity and minimal overhead. Among them, Insertion Sort was the most efficient when data was nearly sorted, while Bubble Sort was the slowest due to excessive swaps. Shell Sort improved on Insertion Sort by reducing the number of swaps, making it more efficient for medium-sized datasets.

For larger datasets, algorithms with O(n log n) complexity scaled the best. Merge Sort, Heap Sort, and Quick Sort consistently performed well due to their efficient divide-and-conquer or heap-based strategies. Among them, Merge Sort was stable but had higher space complexity, while Heap Sort was slower in practice due to higher constant factors in heap operations. Quick Sort (median-of-three pivot) provided the best overall performance with minimal variance.

Regarding performance variability, Quick Sort (first-pivot selection) had the most variable performance, struggling with already sorted or reverse-ordered data, leading to O(n^2) worst-case behavior. Random Pivot Quick Sort showed slight variance but generally performed well. Counting Sort, Bucket Sort, and Radix Sort, which rely on non-comparative sorting, had the least variability, performing consistently in O(n) or O(n + k) complexity but requiring additional space.

In summary, for small datasets, Insertion Sort and Shell Sort were effective. For large datasets, Merge Sort, Heap Sort, and Quick Sort (median-of-three/random pivot) scaled the best. Counting, Bucket, and Radix Sort were highly efficient for specific data types but required additional constraints such as integer-based keys. Quick Sort (first-pivot selection) had the most inconsistent performance, while non-comparative sorting algorithms demonstrated the most stable behavior.

3.2 Trade-offs and Applications

Insertion Sort:
While efficient at sorting small datasets and nearly sorted data it’s worst-case time complexity is O(n^2) and is not efficient for large, randomly ordered datasets. This is a stable

algorithm that’s often used as the base case in more complex sorts like Merge Sort or Quick Sort.

Selection Sort:
This algorithm is easy to implement and performs well on small datasets but it will run with O(n^2) time complexity in all cases and is unstable. This algorithm is great to use in situations where minimizing writes to memory is important due to its minimal number of swaps.

Bubble Sort:
The best-case for bubble sort is O(n) in the case of already sorted data but with most data it will be running at O(n^2). It generally only works well with small datasets and limited scenarios but it’s very inefficient for large datasets and therefore in most cases Insertion Sort would generally be better.

Shell Sort:
For moderately sized datasets this has better performance than Insertion Sort with a time complexity that is better than O(n^2) but not as good as O(n log n). With very large datasets this is slower than other advanced sorting algorithms however, it’s good for embedded systems where memory might be limited.

Merge Sort:
Consistent time complexity of O(n log n) for all cases but can be slightly slower than some cases of Quick Sort. A great choice for large datasets and situations where stability is required. This is a reliable and efficient sorting algorithm for many applications like sorting large files on a disk.

Quick Sort (First-Pivot, Iterative):
While efficient for random data, it suffers from O(n^2) worst-case performance on sorted or reverse-sorted data. Due to its instability, it is not suitable for applications requiring stable sorting, such as database sorting with primary and secondary keys.

Quick Sort (Random Pivot, Iterative):
Offers O(n log n) average-case performance, reducing the risk of worst-case behavior. It is commonly used in general-purpose sorting, such as in standard library implementations.

Quick Sort (Median-of-Three Pivot, Iterative):
Balances partitions better than first-pivot and random-pivot versions, making it more consistent with O(n log n) performance in most cases. It is preferred in performance-sensitive applications such as compilers and real-time systems.

Heap Sort:
Ensures O(n log n) worst-case performance, making it reliable for real-time systems and security-sensitive applications. Though slower than quicksort due to poor cache efficiency, it is preferred in memory-constrained and predictable runtime environments.

Counting Sort:
Only operates on integers. It could theoretically be used to sort other data types if each element to be sorted had an integer key associated with it. The limited type of input is made up for by the fast runtime of this algorithm and the complexity of O(n + m) where m is the maximum value integer in the input array. This algorithm can have variable performance on small datasets, and those with specifically large maximum values. This algorithm is not done in place, so it requires extra memory.

Bucket Sort:
If data is evenly distributed this achieves near linear time complexity, O(n). However, the performance is highly dependent on distribution which severely impacts its runtime. Bucket sort typically requires extra memory to store the buckets themselves. A very adaptable algorithm as it can handle various data types as long as there's defined mapping.

Radix Sort:
This is great for large datasets of strings or integers with fixed-sized keys and it's faster than comparison-based sorting algorithms. It's a stable algorithm that is easily parallelized. However, it's not efficient for sorting floating point numbers or anything that can't be easily mapped to a small number of digits. It requires a significant amount of memory and it's not efficient for small data sets.

3.3 Lessons Learned

Implementing, testing, and analyzing sorting algorithms provided valuable lessons about the critical relationship between algorithm choice and input characteristics. We learned that selecting the right algorithm hinges not only on the general task but also on the specific nature of the data and the application's requirements. For instance, while some algorithms boast a certain time complexity in theory, their actual runtime can vary significantly depending on the implementation and, crucially, the input data's properties like order and distribution. This hands-on experience solidified our understanding of Big O notation, demonstrating how theoretical complexity translates into real-world performance differences.

Furthermore, this process offered deeper insights into the inner workings of each algorithm, highlighting their respective strengths and weaknesses. We gained practical experience measuring method runtime and passing methods as parameters, essential skills for performance analysis. Manipulating input datasets to observe their impact on sorting time complexity reinforced the importance of data organization and preprocessing. Ultimately, this exercise underscored the fact that a thorough understanding of both algorithmic functionality and input characteristics is paramount for efficient and effective sorting.