Analysis

Abdullah

The implementation I used was inspired by the zyBook's in-place swap. And knowing that the QuickSorts would end up using much of the same code I used a lambda expression to make debugging and modifications easier, on top of saving me some time, and hey I now I know how to use a Java lambda expression.

There weren't many other implementations that I'm aware of, so I went and just got to coding.

They sort of did, due to the low sample size , and java giving me inconsistent timings like one of the bugs being that the first sample was way bigger even though it was the smallest array; which I fixed by running the algorithm once before beginning to time them. . . I love java :) But they seem to increase at a near linear scale, and in the worst case it was definitely n^2

I believe they did, due to functions being so fast other factors could have affected the runtime introducing noise, at the very least the worst case was definitely a n^2 as previously stated. This popped up in the worst case for the first element pivot, eventually leading to a stack overflow due to it reaching a stack depth of 3200 ish. Both the median sort pivot and first element pivot suffered when it comes to handling reverse ordered arrays, but when it came to handling ordered arrays it seems the median is the best as it can most efficiently chop up the array.

Random seemed to do the best overall, though I believe if we kept growing the size of the arrays median would come out a bit better when it comes to sorting random arrays. Random also handled every case exceptionally well and wasn't prone to the worst case as badly as the other two.

Abenezer

Bucket Sort's implementation employs dynamic bucket sizing based on the square root of the array length, balancing memory usage with performance. The code handles edge cases effectively, including empty arrays, single-element arrays, and identical element arrays. Using ArrayList for buckets provides efficient memory management for variable bucket sizes. The implementation normalizes value ranges to properly handle negative numbers and prevent division by zero errors. While alternative implementations using fixed bucket counts or LinkedList structures were considered, they were rejected due to poor performance with non-uniform distributions and inferior cache behavior respectively. The implementation achieves the theoretical O(n + k) average case complexity for uniform distributions, while maintaining O(n^2) worst case when elements cluster in a single bucket. Performance varies significantly based on data distribution - excelling with uniform data but degrading with clustered inputs.

Selection Sort uses a straightforward in-place implementation focused on minimal space usage and code simplicity. The algorithm performs a single swap per outer loop iteration to minimize write operations. More complex variations like double-ended selection sort or recursive implementations were rejected to maintain simplicity and avoid additional space overhead. The implementation exactly matches its theoretical O(n^2) time and O(1) space complexity bounds. It performs a consistent n(n-1)/2 comparison regardless of input characteristics, making it highly predictable but inefficient for large datasets. The algorithm shows no sensitivity to input ordering - performing identically on sorted, reverse sorted, or random data.

Shell Sort implements a simple gap sequence (dividing by 2 each time) to achieve predictable though not optimal performance. The in-place implementation maintains space efficiency while moving elements within gaps similar to insertion sort. More sophisticated gap sequences like sedgwick's were considered but rejected to maintain implementation simplicity. The code achieves the theoretical O(n^2) worst case bound for the chosen gap sequence, though a better sequence could theoretically approach O(n log n) at the cost of a simple sequence chosen providing predictable but sub-optimal performance. The algorithm shows particular efficiency with partially sorted arrays while handling reverse sorted arrays competently due to its gap approach.

Armando

The implementation for the Radix Sort algorithm was inspired by a YouTube channel called Geekific. I went with this implementation because the channel does a good job of breaking down complex algorithms into understandable bits with visualizations. This implementation also takes into account negative values. As for expectations, Radix Sort is considered to be better than other comparison-based sorting algorithms for large datasets when keys have several digits, although it is not the best for small datasets. Its Big O is d * (n + b) where d is the number of digits, n is the number of elements, and b is the base of the number system being used. Based on my charts for this algorithm, the slope for the curve as the input grew became less steep for all scenarios. This confirms that it does in fact do better for larger datasets, whereas the smaller input sizes yielded steeper curves. I noticed that the inputs that were fifty percent ordered and random had runtime spikes for the smaller input sizes, which further validates this algorithm's inefficiency for smaller unordered data sets.

The implementation for the Heap Sort algorithm was also inspired by a YouTube channel. I went with their implementation because they visualized the way in which a heap structure is actually used to sort some dataset. Heap sort is known for having a time complexity of n * log(n) for all cases, meaning it can handle large datasets relatively well. The n comes from the height of the heap. My charts for this algorithm did not completely align with the n * log(n) runtime expectation with all array types and it instead demonstrated more of a log(n) runtime in some cases. The reversed and random input arrays seemed to influence the log(n) runtime whereas the others aligned more with the n * log(n) runtime where the slope of the curve gets gradually steeper as the size increases.

The implementation for the Bubble Sort algorithm was inspired by an instructor's code from a CS class. I went with their implementation because they had studied the algorithm and found their implementation to be a valid way of doing it. Bubble sort is considered to be inefficient when the elements are arranged in decreasing order, with a worst-case runtime of n^2. The charts for this algorithm confirmed this expectation since the slope for the curve increased drastically as the input got larger, which aligns with an n^2 runtime. The algorithm performed somewhat sporadically for the ordered array when the size was below 4,000, but it performed as expected for all other array types.

Joe

Counting sort ended up being a little bit faster overall than merge sort did. This matches the theoretical results since counting sort is O(n + k) and merge sort is O(n log n). I'm going to compare those two run times for an array size of 32,768. I will use 40,000 as the range for counting sort below since that is the highest number that will be used in our experiment.

Counting sort, k = 40,000, n + k = 32,768 + 40,000 = 72,768 operations
Merge sort, log n = log 32768 = 15, so n log n = 32,768 * 15 = 491,520

From this, for array sizes not larger than 32,768 and element ranges not larger than 40,000 counting sort is the clear winner. The problem with counting sort is that it is O(n + k). So, to figure out where counting sort will start being less efficient we can look at the numbers above. If we subtract n from the merge sort operations then we get (491,520 - 32,768 = 458,752).
So at an integer range of 458,753 merge sort will start running faster. If we had an extremely large number range of let's say 1 billion then counting sort would be much slower. I will show those calculations below.

n = 32,768
log n = 15
k = 1,000,000,000
Merge sort = 32,768 * 15 = 491,520
Counting sort = 32,768 + 1,000,000,000 = 1,000,032,768

Thus, in that experiment merge sort is running just shy of a half million operations and counting sort is running a little over 1 billion. This matches the theoretical analysis since the integer range makes no difference to merge sort but is critical in the analysis of counting sort.

Insertion sort ran as expected since it runs at O(n^2). For all of the experiments merge sorts highest run time was 4ms, counting sort's highest run time was 3ms and insertion sort's highest run time was 800 ms. In small data sets of 2,048 and below insertion sort was somewhat comparable
to merge sort and insertion sort, but after that it really started to slow down. At an array size of 32,768 insertion sort will run in 32768^2.

32768^2 = 1,073,741,824
From these results we can see that if our number range was 1 billion that at an array size of 32,768 insertions sort and counting sort almost have the same number of operations. So if we were to have an integer range of 2 billion then insertion sort would have roughly half the operations of counting sort.
With that number range, merge sort would be unchanged with its 481,520 operations.

Thus up to a number range of 458,752 counting sort is the clear winner, with a number range above that merge sort takes over and with a range of 2 billion insertion sort is better than counting sort. Thus, if the range is below 458,753 use counting sort and anything higher use merge sort. Insertion sort will always run more slowly than both except with extremely high integer ranges, but that is irrelevant because you just use merge sort in that case anyway