1) Theoretical predictions.

Solution: On randomly ordered data, we expect the following ordering:

Heapsort = Mergesort = Quicksort (deterministic or randomized) < Insertion Sort = Selection Sort

Heapsort and mergesort both have worst-case runtime $\Theta(n \lg n)$, which as we saw in class is the best any comparison-based sorting algorithm can do. Quicksort has worst-case runtime $O(n^2)$ but average-case runtime $O(n \lg n)$. Since we are assuming the data are originally distributed throughout the array uniformly at random (i.e., no initial ordering is more likely than any other), we can imagine that we likely fall into the “average” case and so will see $O(n \lg n)$ performance. Insertion sort and selection sort both have worst-case runtime $\Theta(n^2)$, which is worse than $\Theta(n \lg n)$, so we expect them to take longer.

3) Timing.

Solution: At a high level, the empirical ranking of the five algorithms is much as we expected, based on our theoretical predictions. The three $O(n \lg n)$ algorithms (heapsort, mergesort, and quicksort) are considerably faster than the two $O(n)$ algorithms (insertion sort and selection sort), particularly as $n$ grows large.

![Random Data - linear scale](image)

Our first graph shows runtime as a function of $N$ for the five algorithms, using a linear scale on both the $x$ and $y$ axes. It’s clear that insertion sort and selection sort are much worse than the other three algorithms, and we can see the $n^2$ shape. But it’s very hard to tell whether there are any differences in runtime between the fastest three algorithms, so let’s plot our $y$ axis on a log scale to get a clearer sense of what’s going on.
On a log scale, it becomes apparent that selection sort and insertion sort really are a constant factor apart (recall that on a log scale, a constant difference between two curves on the graph corresponds to a constant factor difference between their numerical values). Similarly, we see that heapsort, mergesort, and quicksort differ only by constant factors, and are much more tightly clustered with one another than insertion sort and selection sort are. But so far we’ve only plotted results up to \( N = 10^6 \). Insertion sort and selection sort can’t handle many more elements than that in a reasonable amount of time, but the faster sorts can.

Going up to \( N = 10^8 \) elements, it’s clear that there are minor constant-factor differences between the three fastest algorithms. Mergesort and quicksort are similar, with quicksort being a little bit faster. Heapsort is
roughly a factor of 2 slower than the other two algorithms, which may not sound like a lot, but it can make a big difference as \( N \) gets very very large.

Some more detailed observations:

- While insertion sort and selection sort have the same asymptotic growth rate, selection sort generally performs much worse than insertion sort. This is because selection sort requires a complete pass through the entire “unsorted” subarray on each iteration of the outer loop, whereas insertion sort may only have to pass through a few elements before locating the correct position in which to insert the next element. Both algorithms have worst-case runtime \( \Theta(n^2) \), but insertion sort typically has better constant factors. Those constant factors can make a big difference in the empirical performance!

- While heapsort, mergesort, and quicksort have the same asymptotic growth rate, there are differences in their relative empirical performance, particularly as \( n \) gets large. Heapsort tends to be a bit slower than mergesort and quicksort, and quicksort tends to be a bit faster than mergesort. This is due to the constant factors that we ignore in the asymptotic analysis.

- On random data, it does not matter which version of quicksort we use: both deterministic quicksort and randomized quicksort will give the same empirical performance. Since our input data is arranged randomly throughout the array, each element is equally likely to be in the last position, and therefore each element is equally likely to be chosen as the pivot by deterministic quicksort. Thus randomized quicksort provides no advantage in this setting.

4) Almost-sorted data.

Solution: Three of our algorithms (selection sort, heapsort, and mergesort) perform similarly with 10-sorted data as they did with random data. Two of our algorithms (insertion sort and quicksort) perform quite differently (depending on the version of quicksort used).

Our first graph shows runtime as a function of \( N \) for the five algorithms (including both deterministic and randomized quicksort), using a linear scale on both the \( x \) and \( y \) axes. It’s clear that selection sort still has
the same $O(n^2)$ performance as with the random data, but it’s hard to see what’s happening with the other algorithms.

On a log scale, some interesting trends become easier to see. Deterministic quicksort (which we were only able to run up to about $N = 50000$ before running into a stack overflow error), very clearly has the same $O(n^2)$ shape as selection sort. Randomized quicksort, on the other hand, continues to run in time $\Theta(n \lg n)$ like mergesort and heapsort. Perhaps the most striking observation is that insertion sort, which took time $O(n^2)$ on random data, now appears to do far better than any of the other algorithms.
Going up to \( N = 10^8 \) elements, we can see that the insertion sort curve actually has a different shape than the heapsort, mergesort, and randomized quicksort curves once \( N \) gets very high. Insertion sort actually appears to take closer to linear time!

Some more detailed observations:

- In order to find the smallest element in the “unsorted” subarray, selection sort must scan across all elements in that subarray. It doesn’t matter what order the elements are in. So selection sort is insensitive to the initial input ordering.

- Similarly, mergesort and heapsort have similar empirical runtimes for 10-sorted data as for random data. Depending on our particular implementation details, there might be some slightly better constant factors for 10-sorted data, particularly for mergesort. In my mergesort implementation, I merge the subarrays from \( lo \) to \( mid \) and from \( mid+1 \) to \( high \) in the array \( to_sort \) by first copying the entire subarray from \( lo \) to \( hi \) into a temporary array. I then walk up each half of the temporary array, copying elements back into \( to_sort \), now in sorted order. But if my counter for the left half of the temporary array reaches \( mid \) before my counter for the right half of the temporary array reaches \( hi \), I know I can stop copying because the remaining elements on the right half will stay where they were in the original array. When the input array is 10-sorted, it’s fairly common that I get to stop copying the right half of my array well in advance of my counter reaching \( hi \). This leads to a nontrivial improvement in the constant factor, but it does not change the asymptotic runtime.

- Insertion sort is much faster on 10-sorted data than on random data. This is because the inner loop of our insertion sort only runs until we find the correct place in which to insert the next element. On random data, we might have to scan all the way down our already-sorted subarray. But when the data is 10-sorted, we only have to scan at most 10 places before we are guaranteed to find the correct position. So the overall runtime for insertion sort becomes linear: even better than mergesort, heapsort, and quicksort! Note that this is not a counterexample to our comparison-based sort \( \Omega(n \lg n) \) lower bound, because in order to get the linear runtime we have to make an assumption about the initial ordering of our input data: not all \( n! \) original permutations of the data are possible.