Programming 2 Assignment

Michael Earl, Gerald Hu

CSCE 221-200

April 11, 2016

Introduction

The purpose of this assignment was to analyze the theoretical and actual performance of several common sorting algorithms discussed in class. The project implemented bubble sort, two "slow sorting" algorithms (insertion and selection sort), and two "fast sorting" algorithms (mergesort and quicksort). The running time of these algorithms was analyzed for several types of data: sorted data, reverse-sorted data, randomized data, and data with many similar elements. Insertion Sort and Bubble Sort ran quickly on already-sorted data, but performed worse in other areas. Out of the slow sorts, Insertion was the best, and Bubble was the worst in most cases. Out of all sorting algorithms, Quicksort and Mergesort performed similarly, but Quicksort was a bit faster. Selection sort ran in $\mathcal{O}(n^2)$ in all cases, insertion sort and bubble sort ran in $\mathcal{O}(n^2)$ in the average and worst cases, but in $\mathcal{O}(n)$ in the best case. Mergesort runs in $\mathcal{O}(n \log n)$ in all cases, and quicksort runs in $\mathcal{O}(n \log n)$ in the best and average cases, but in $\mathcal{O}(n^2)$ in the worst case. We learned that some algorithms perform better in certain situations than others, namely that insertion sort is very good at sorting already sorted and almost sorted sequences, but quicksort is better for just about everything else.

Implementation Details

All sorting algorithms took random-access iterators to the first and last elements of the container, and a comparator.

Bubble Sort is a simple sorting algorithm. It compares a set of two elements that are next to each other, and if the elements in the bubble are out of order, they are swapped. This set traverses the entire container from start to end, then starts from the beginning again, repeating until there are no more swaps to make.

Selection Sort is one of the two "slow" sorting algorithms. It traverses the entire container searching for the smallest element, then swaps that element into the first position. Then it searches the remainder of the container for the next-smallest element, then swaps that element into the first position of the remainder. On its ith traversal, it will swap the smallest element into the ith position. This process continues n times, at which point there is no more "remainder" of the container to traverse.

Insertion Sort is the other of the two "slow" sorting algorithms. It works by keeping an already-sorted area at the beginning of the container. And swapping the current element back through the already-sorted area until it reaches the correct spot. In each traversal, for each step i, if i is not the first element, and if i and $i - 1$ are out of order, the two are swapped, swapping the element into an "already-sorted" portion of the container. Each traversal will grow the already-sorted portion, until all the data is in the sorted portion.

Mergesort is one of the two "fast" sorting algorithms. It splits the input data into two subarrays, and recursively splits each of those subarrays into two subarrays, repeating the splitting process until each subarray is of size<2. These subarrays are then sorted in sets of two, and the resulting sorted data is put into an array and returned. The returned sorted data is merged with more data using the same merge function, until fully-sorted data is returned.

Quicksort is the other of the two "fast" sorting algorithms. It randomly selects an element of the data to use as a "pivot" , and traverses the entire container. It moves all elements less than the pivot to before the pivot's position, and moves all elements greater than the pivot to after the pivot's position, all of which is done in-place. This places the pivot in the proper position. Then, quicksort is recursively called, once on the set of elements less than the pivot, and once on the set of elements greater than the pivot.

The skeleton of timing.cpp was provided by the instructors. The timing function was already implemented for randomly sorted input data; we modified it to analyze sorted and reverse-sorted data, as well as data with many similar elements. The timing function relies on high_resolution_clock; given a sorting function and a number n, the timing function would generate a vector with n elements in it. Then, the timing function would try to sort the vector using the specified sorting algorithm. This process was repeated for increasing sizes of n, and repeated 10 times at each n to ensure an accurate average time. [TODO - what we learned]

Theoretical Analysis

Bubble Sort is $O(n^2)$ average and worst case. It will force the largest element to the end after n comparisons; then it will repeat this for the $n-1$ smallest element, then the $n-2$ smallest element, and so on, repeating n times in total. n traversals and n comparisons/swaps at each traversal results in $O(n^2)$ time. Bubble Sort's best case is $O(n)$, in the case that it's already sorted. It traverses the list once and makes n comparisons, but does not do any swaps or any further traversals.

Selection Sort is $O(n^2)$ in all cases. In searching for the smallest element, it will traverse n elements and make n comparisons, before forcing the smallest element to the start. It then repeats this process for a sublist of size $n-1$, then $n-2...1$. $\sum_{i=0}^{n} i = (i)(i+1)/2$, which is $O(n^2)$ behavior. Furthermore, it will always search for the smallest element, regardless of whether it needs to be swapped or not, and will perform all traversals regardless of what data type it's given.

Insertion Sort is $O(n^2)$ average and worst case, for similar reasons as Selection Sort: it will traverse a list of size 1, then size 2, then 3...n in the worst case. Unlike Selection Sort, Insertion Sort is $O(n)$, in the best case that it is already sorted, as it will only traverse the list once.

Mergesort is $O(nlog(n))$ in all cases. At each step, the problem is split in half, until it reaches the smallest possible subproblem; it will take $O(logn)$ splits to reach the base level, and $O(n)$ to merge all the data back together at each level. There is not a "best case" input type, as mergesort will perform the same operations regardless of the data it's given.

Quicksort is $O(nlog(n))$ average case and best case, and $O(n^2)$ in a rare worst case. The best case of quicksort is $O(nlogn)$ due to an inherent limit on comparison-based sorting algorithms. Assuming an "average case" where the pivots selected give equal (or close to equal) sized sublists, it will take $O(n)$ to parition the sequence around the pivot at every level and $O(logn)$ splits to reach the base level. The worst case is if every pivot chosen is a bad selection, splitting into a subproblem of size 1 and size $(n-1)$, which is then split into a subproblem of size 1 and size $(n-2)$, so on....resulting in $O(n)$ splits, which means $O(n^2)$ time in the worst case.

Experimental Setup

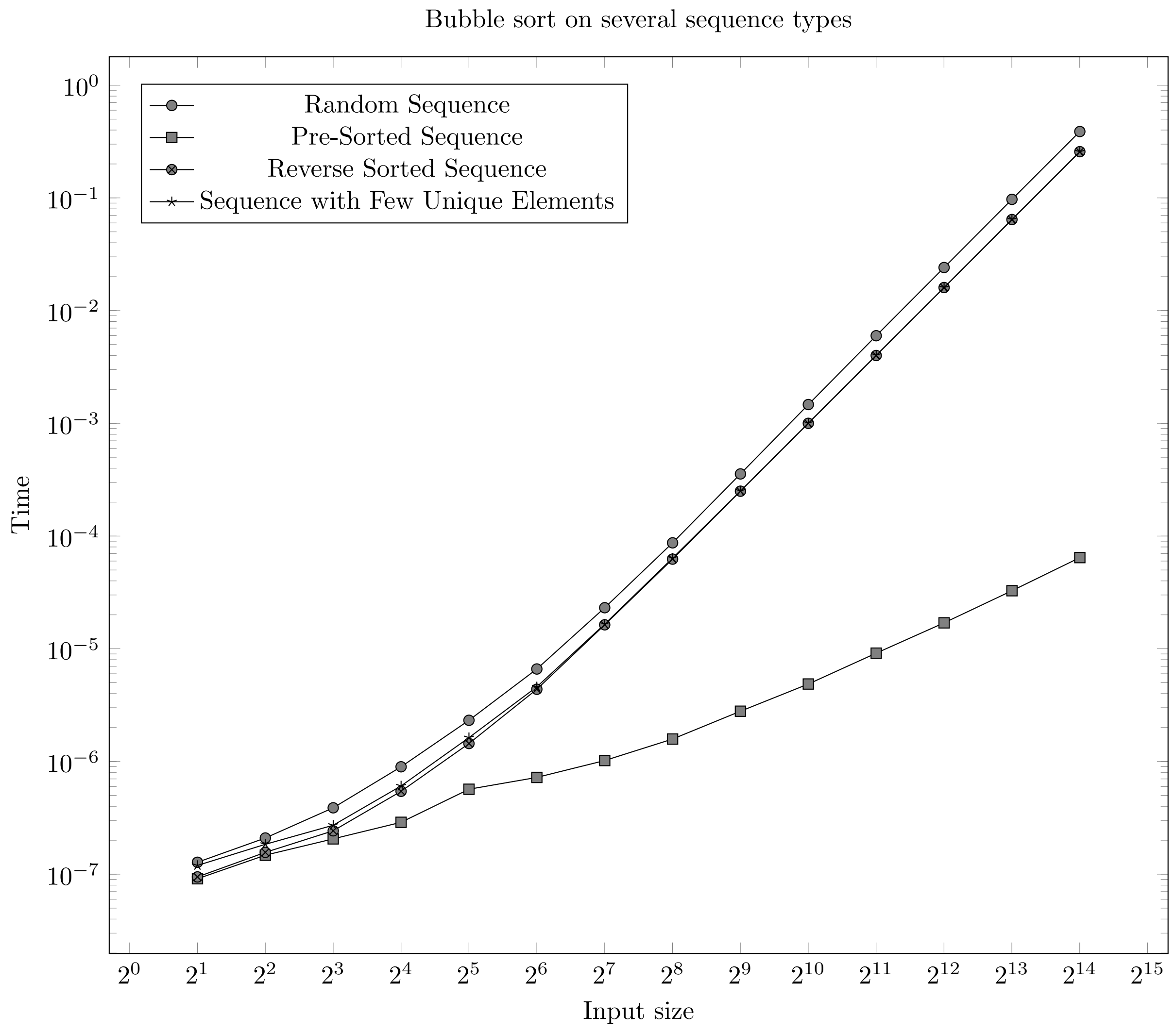
Timing tests were conducted using the provided timing.cpp, compiled with the provided makefile's commands. Compilation was done on Michael's desktop, with G++ version 5.3.0. Compilation was set to the C++14 standard, with the -G flag enabled and O2 optimization level, warnings set to -Wall -Werror (warn all and all warnings treated as compilation errors), and dependencies flagged with -MMD (auto-generate dependencies).

Tests were run on Michael's desktop, which runs Arch Linux x86_64 version 4.4.5-1. It has 8 total gigabytes of RAM. It uses an AMD FX-8350 8-Core processor, with a clock speed of 4 GHz.

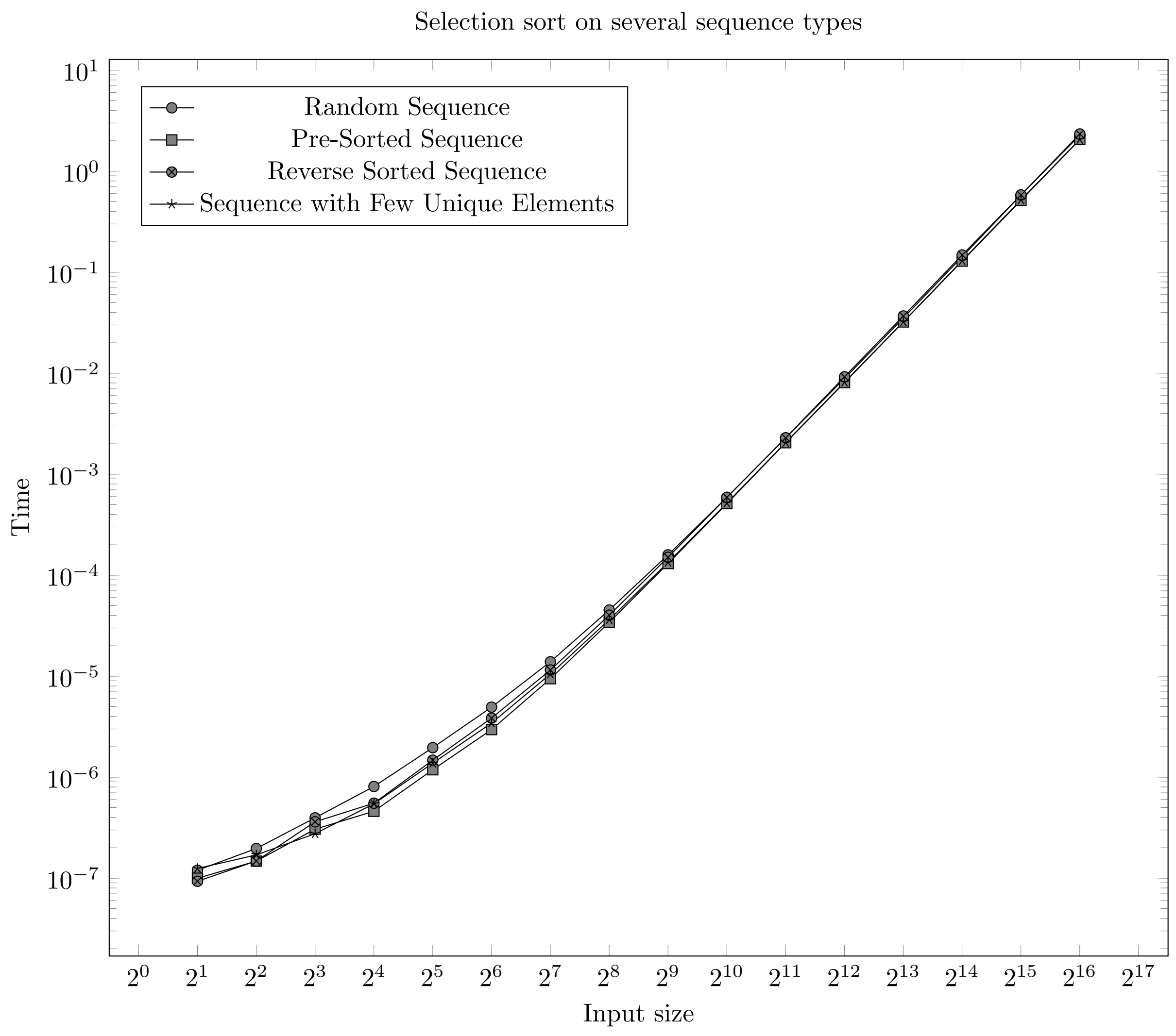
Timing functions output timing results for input sizes that were powers of 2, starting from 2 itself, and ending at a maximum size specified by the user. Each step of the timing was repeated 10 times, and the average of each result taken. Bubble Sort went up to a maximum input size of 32768 (2^{15}) elements; Insertion and Selection sorts went up to a maximum input size of 131072 (2^{17}) elements; and Quicksort and Mergesort went up to a maximum input size of 4194304 (2^{22}) elements.

Four data types were used in testing: sorted data, reverse-sorted data, randomly-sorted data, and data with "few unique elements". For our purposes, "data with few unique elements" was assumed to mean "data where a lot of the elements are the same"; the input data generated consisted mostly of 1s, with a few 2s spaced out in the data.

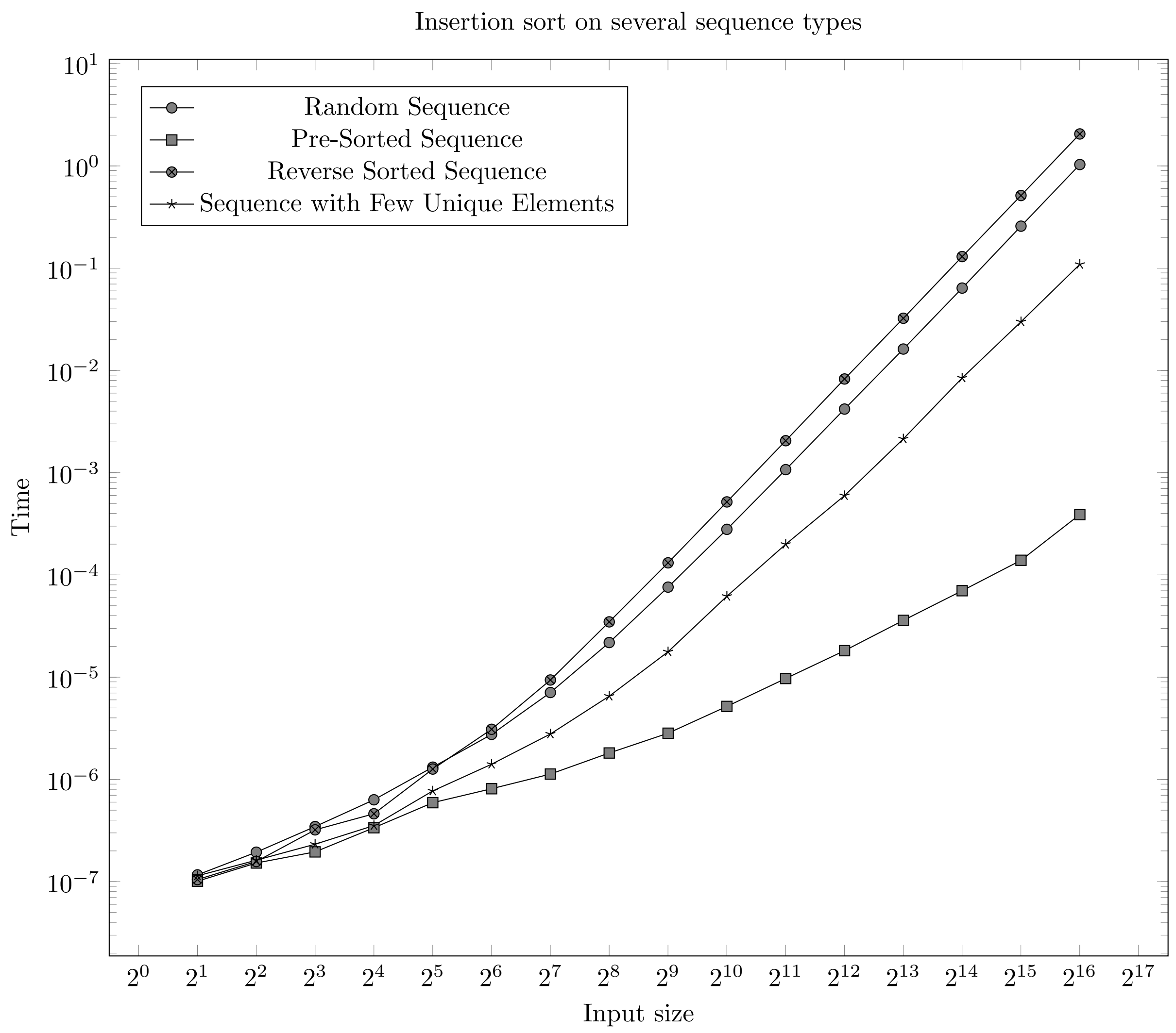
Results and Discussion



Bubble sort performs very similarly on random sequences, reverse sorted sequences, and sequences with few unique elements. It performs nearly *identically* on reverse sorted sequences and sequences with few unique elements. The plots of all three of these seem to follow a quadratic curve, and thus appear to run in $\mathcal{O}(n^2)$ time. The random sequence seems to take slightly longer, most likely because of the bit of extra time it takes to generate the random numbers. On pre-sorted sequences, bubble sort runs very quickly, much more quickly than the other three. The plot for this is clearly linear, which is because bubble sort should only iterate through the sequence once before realizing that it is sorted.

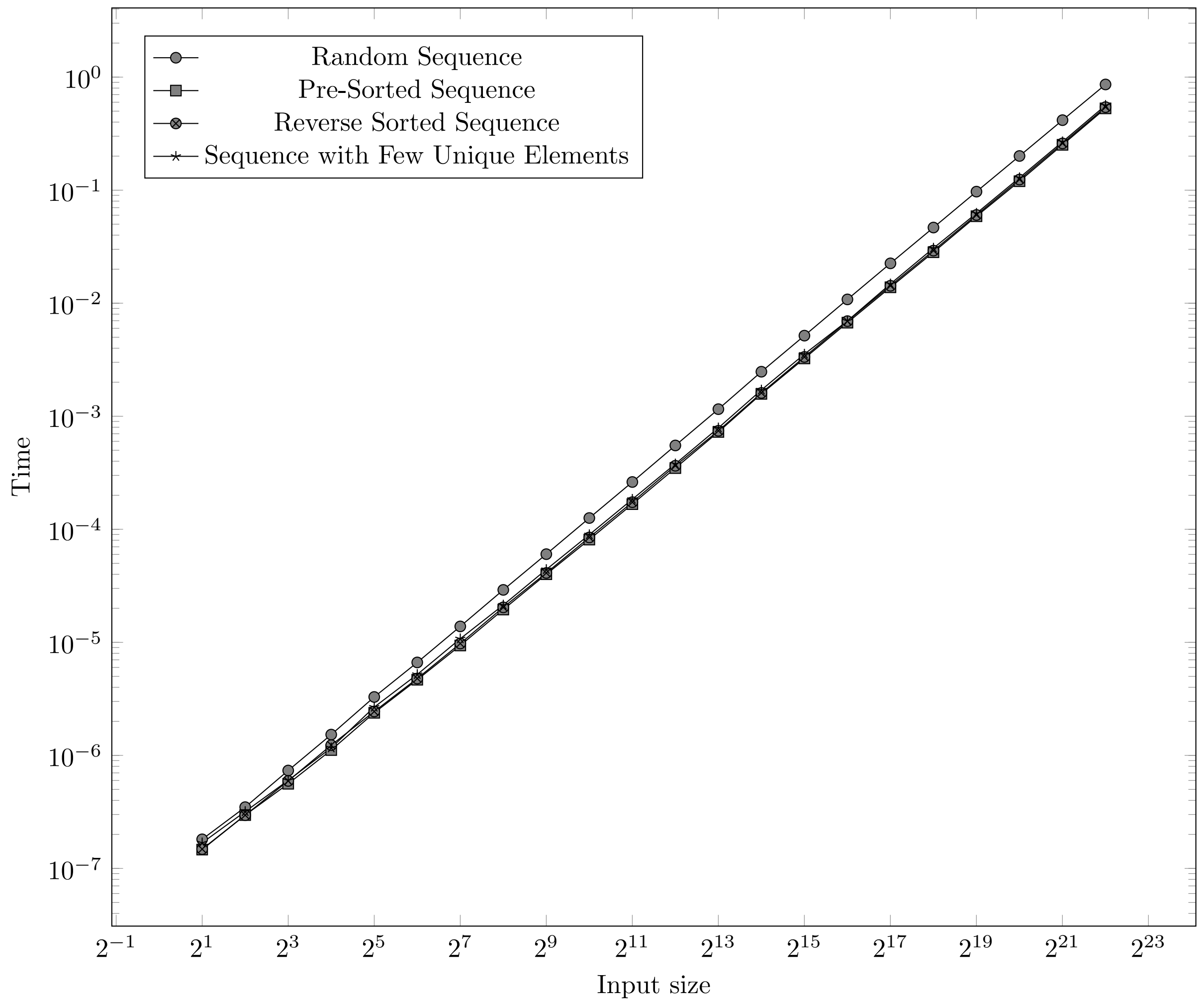


Selection sort performs virtually identically on every type of sequence. This is because no matter the sequence, selection sort will iterate through $n-1$ elements, then $n-2$ elements, and so on, finding the smallest element each time and putting it in the correct position. Its curve is quadratic, so it runs in $\mathcal{O}(n^2)$.

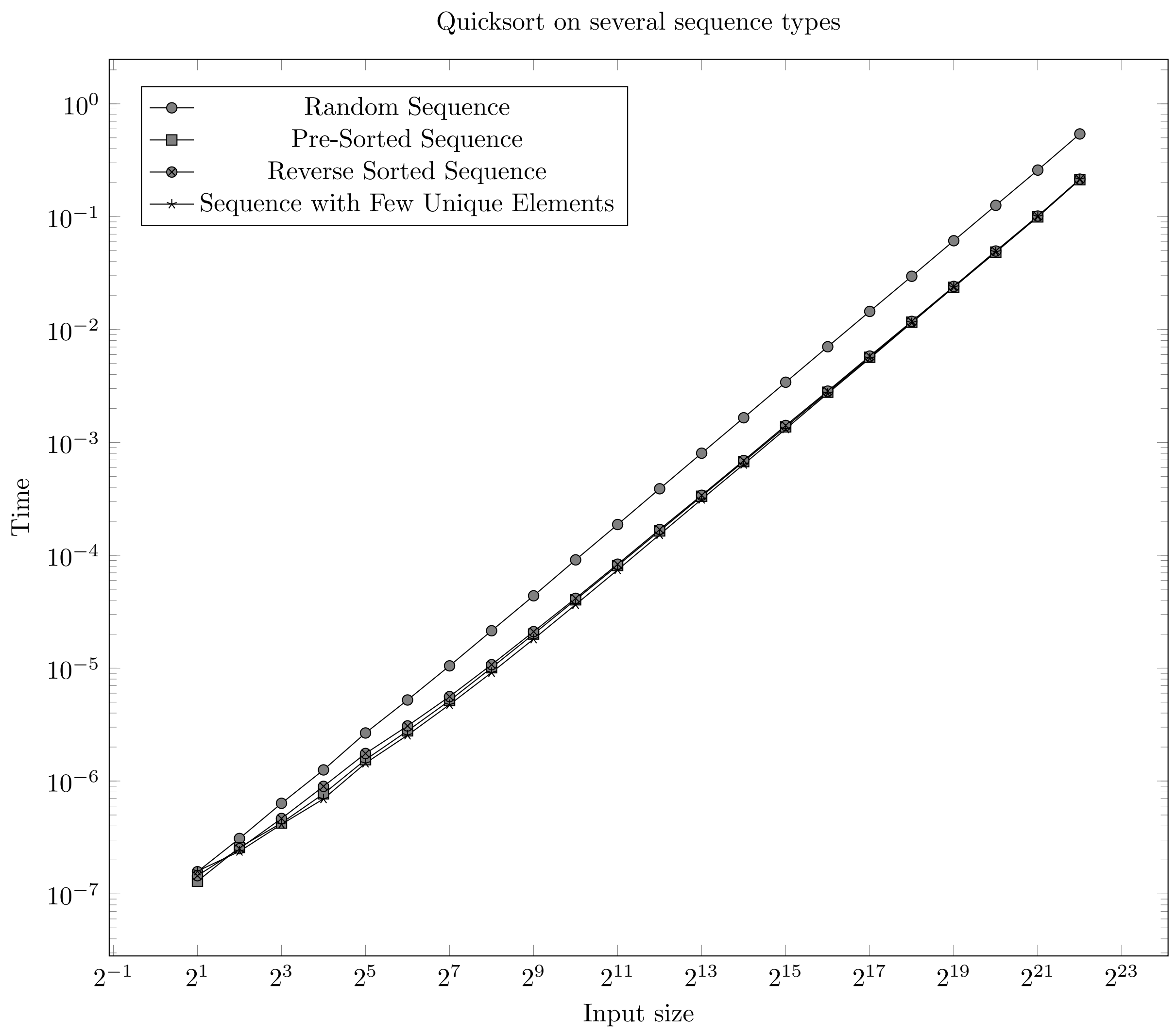


Insertion sort performs fastest on pre-sorted sequences, quite predictably. It simply iterates through the sequence once and doesn't swap anything. The plot is predictably linear, implying $\mathcal{O}(n)$ time complexity on pre-sorted sequences. The other three sequence types all seem to have quadratic curves, implying that insertion sort runs in $\mathcal{O}(n^2)$ on those sequences. These three sequences do have different coefficients, though. Insertion sort is slowest on reverse sorted sequences because it is guaranteed to run in $n - 1 + n - 2 + ... + 1$ operations, which is identical to selection sort. It is slightly faster on random sequences, which are not guaranteed absolutely worst case performance. It runs a bit faster on sequences with few unique elements than it does on random sequences, likely because it doesn't have to make quite as many swaps because so many elements are identical.

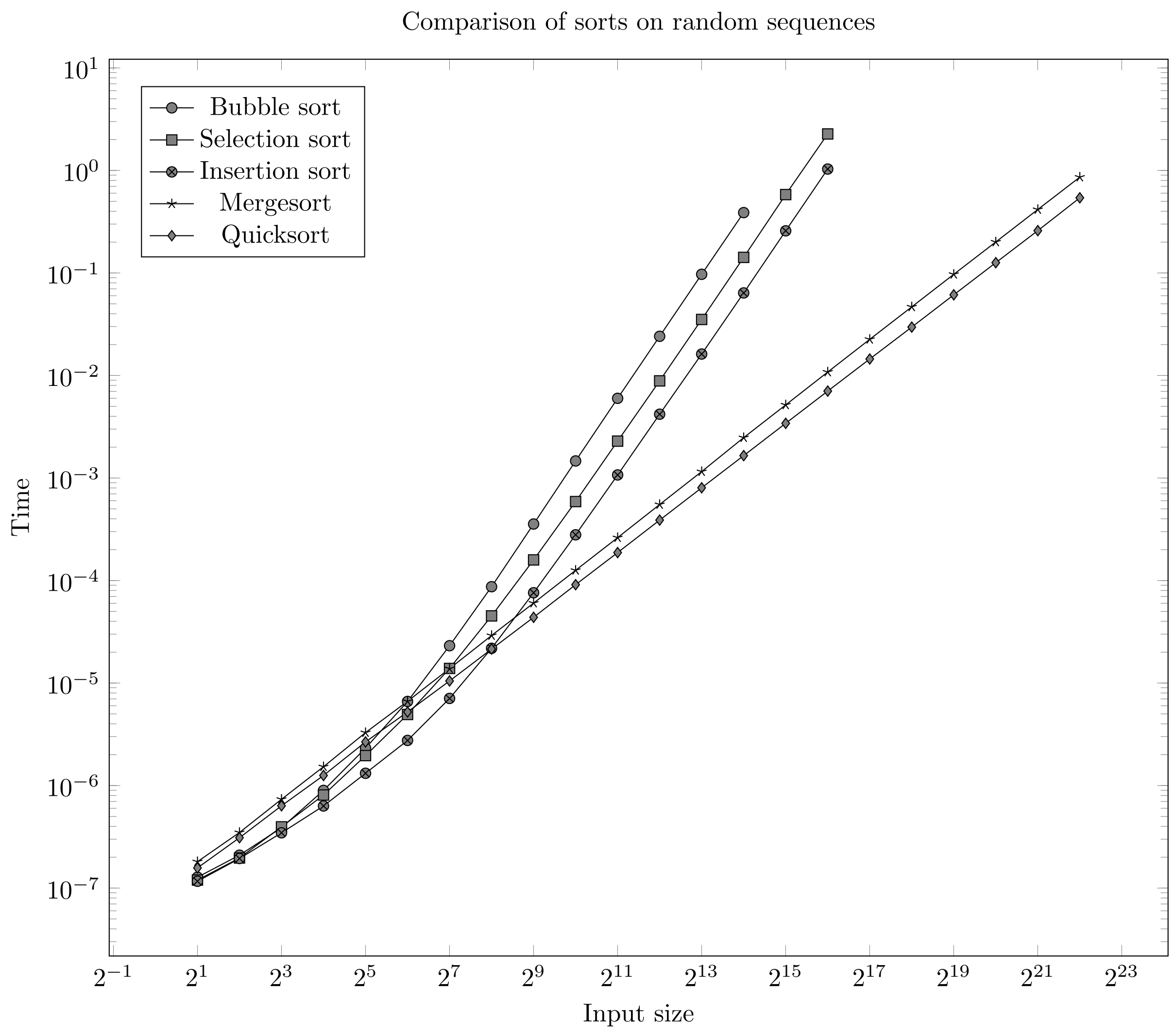
Mergesort on several sequence types



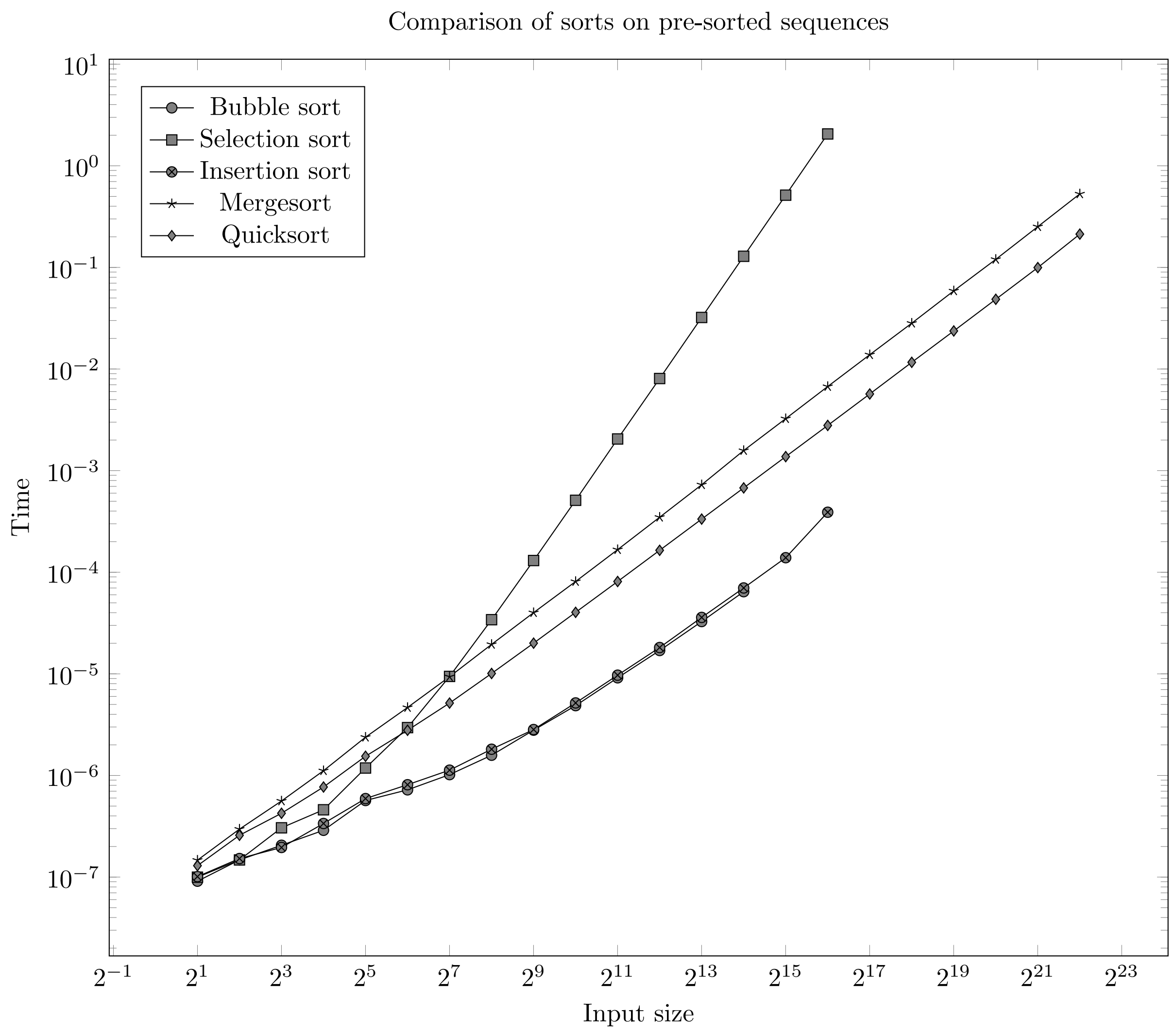
Mergesort runs very similarly on all four sequence types. It is essentially identical for pre-sorted sequences, reverse sorted sequences, and sequences with few unique elements, and slightly slower on random sequences because of the small overhead of generating random numbers. Mergesort performs so similarly on different sequence types is because it does the same steps each time: recursively splitting the sequence in half and then merging the halves back together. The curves *appear* to be linear with a high coefficient, but they actually curve up very slightly because mergesort always runs in $\mathcal{O}(n \log n)$.



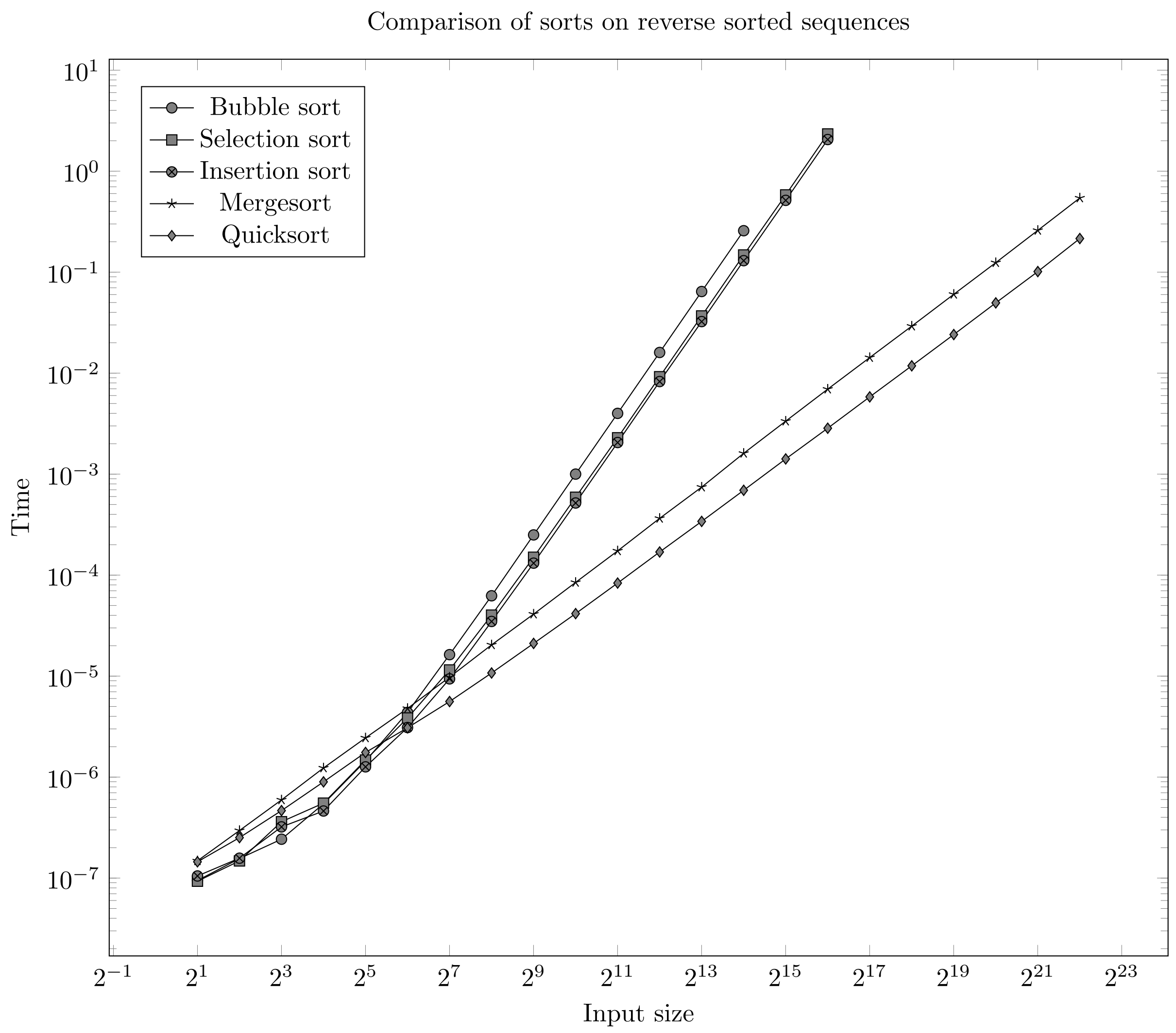
Quicksort runs very similarly on all four sequence types. It is essentially identical for pre-sorted sequences, reverse sorted sequences, and sequences with few unique elements, and slightly slower on random sequences because of the small overhead of generating random numbers. The curves *appear* to be linear with a high coefficient, but they actually curve up very slightly because quicksort almost always runs in $\mathcal{O}(n \log n)$.



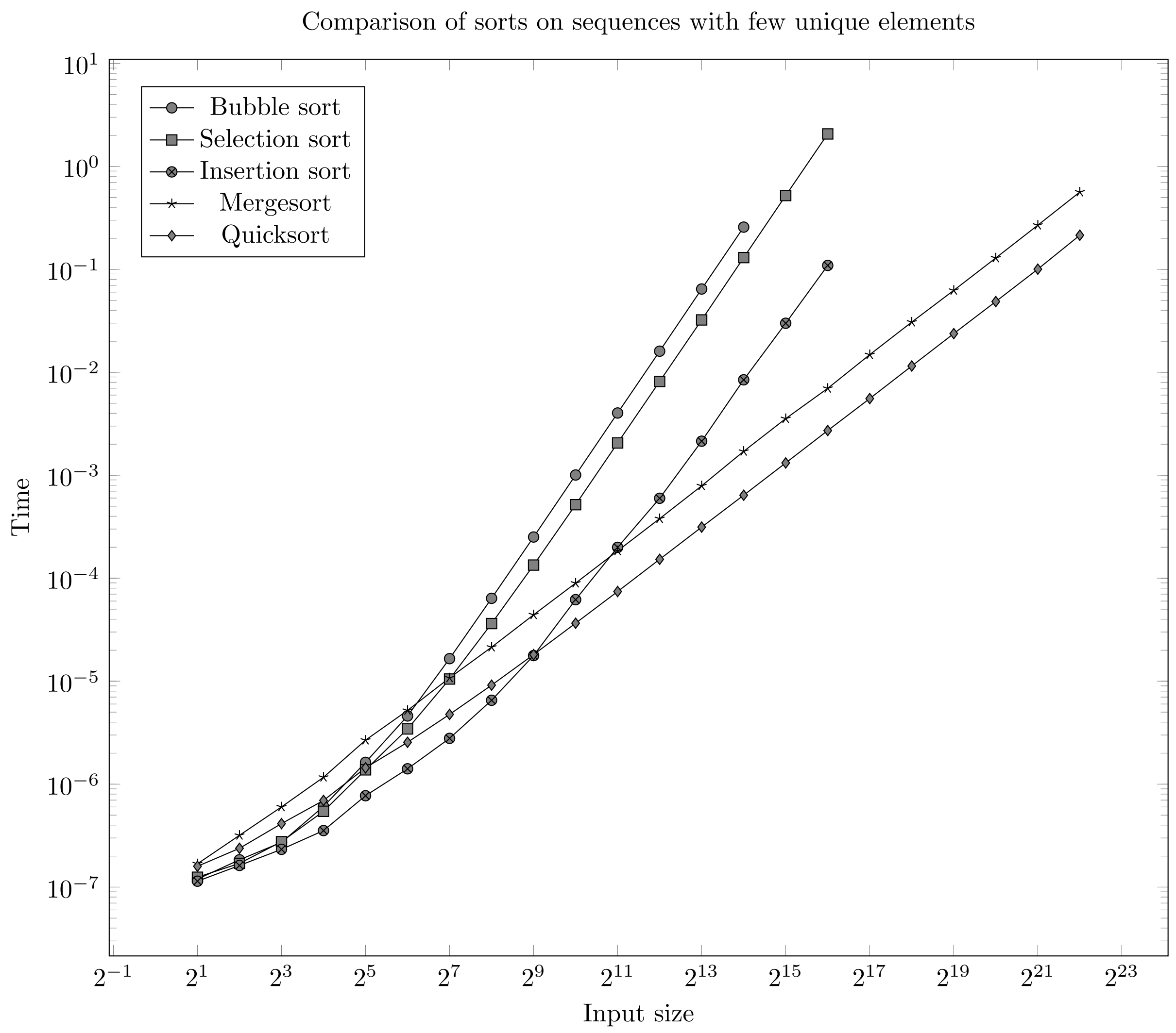
Quicksort and mergesort are clearly the fastest on random sequences (they are both $\mathcal{O}(n \log n)$), although quicksort has slightly better coefficients. Insertion, selection, and bubble sort all take quadratic time (as indicated by their quadratic curves), but insertion sort has the best coefficents and bubble sort has the worst coefficients.



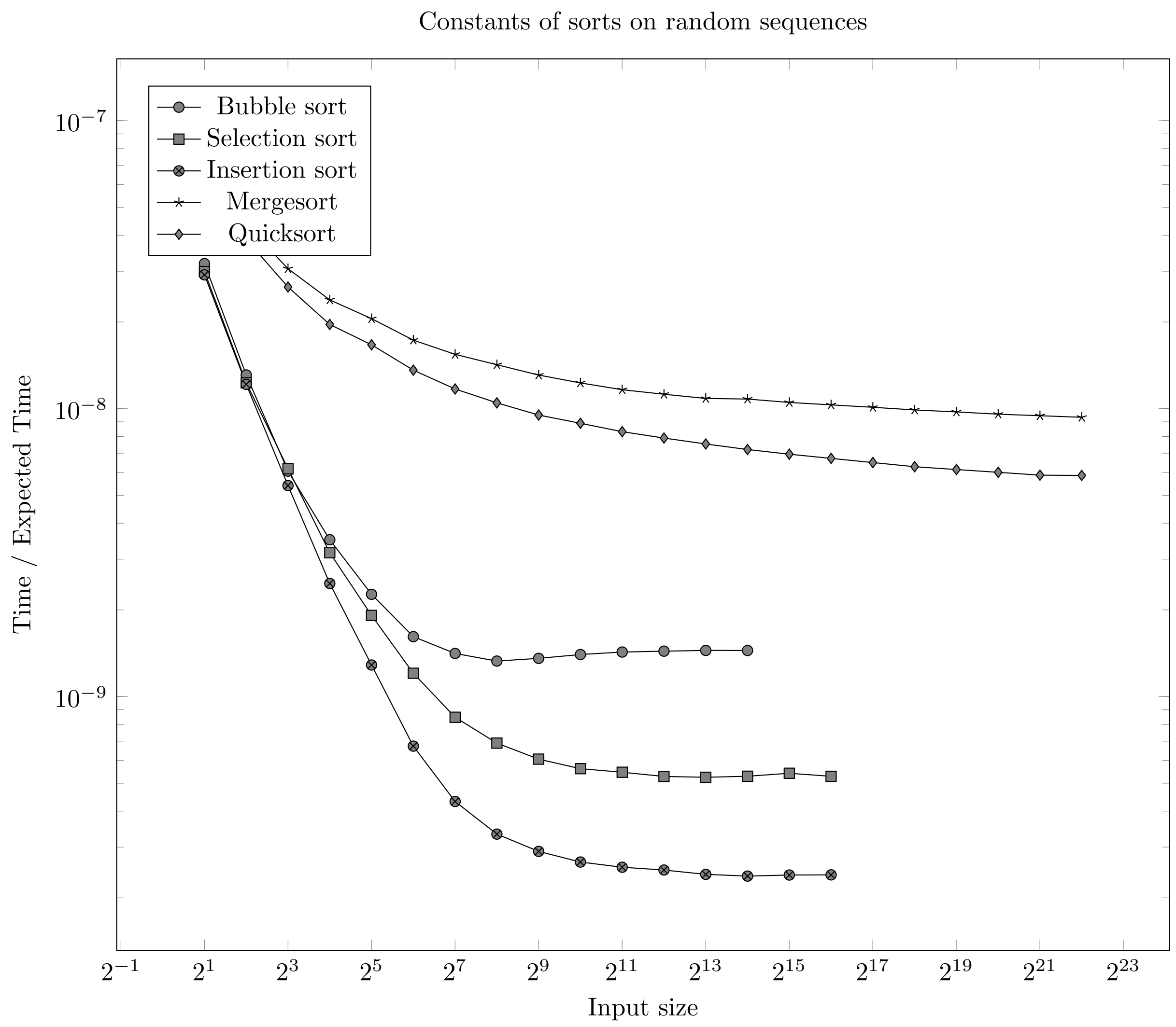
On pre-sorted sequences, insertion sort and bubble sort are the fastest, and their plots are nearly identical. This is because they both simply iterate through the sequence once and stop. Their plots are linear with some mild inconsistencies which can be attributed to changing coefficients, probably due to memory allocation overhead. Quicksort and mergesort are once again very similar, with quicksort being slightly faster again. They don't run any faster than they did on randomized sequences because of their divide and conquer strategies. They are, as usual, $\mathcal{O}(n \log n)$. Selection sort is by far the slowest in this case. While the other slower algorithms performed in linear time on presorted sequences, selection sort did not because it always runs the same way: iterating through the entire array after its current position to find the smallest element and swap it to the right place. Selection sort is, as always, quadratic.



On reverse sorted sequences, mergesort and quicksort perform just like they have on our other sequence types: in $\mathcal{O}(n \log n)$ time with quicksort having better coefficients. Selection and insertion sort perform identically on this sequence type (both quadratically) because insertion sort will have to go through the same number of steps that selection sort does, but in reverse: $1+2+...+n-1$ as opposed to selection sort's $n-1+n-2+...+1$. This occurs because insertion sort will have to move each successive element all the way back through the sequence. Bubble sort is similar, but with worse coefficients.



On sequences with few unique elements, quicksort and mergesort perform as they have with every other sequence type. Insertion sort runs faster than quicksort on sequences with fewer than 2^9 elements. This is likely because insertion has to perform relatively few swaps because large portions of the sequence will already be sorted because there are large patches of identical elements. Eventually, however, its quadratic performance outweighs its good coefficients (which are due to this unique sequence type) and it becomes slower than both quicksort and mergesort. Selection and bubble sort remain considerably slower, however, due to worse coefficients. They don't get any benefits from this sequence type.



The constants of all of these algorithms improve at first and then level out after a certain point. We hypothesize that this is due to runtime optimizations. The effect is more dramatic with the slower algorithms than it is with the faster algorithms, although we are not sure why. The coefficients of the slower sorts aren't really comparable with the coefficients of the quicker sorts because they are asymptotically different ($\mathcal{O}(n^2)$ vs $\mathcal{O}(n \log n)$), so we will compare them separately. For the slow sorts, insertion sort has the best coefficients and bubble sort has the worst, while selection sort is in the middle. Insertion sort has the best coefficients because it often has to traverse smaller portions of the sequence than selection sort and bubble sort. Selection sort is faster than bubble sort because it performs far fewer swaps than bubble does. In fact, it performs at most n swaps, while bubble sort performs at most $\mathcal{O}(n^2)$ swaps. For the fast sorts, quicksort has a better coefficient than mergesort does. We suspect that this is largely due to mergesort's extra overhead for allocating extra memory, which makes the merge step slower than quicksort's pivoting technique.

Conclusion

This assignment showed the differences in runtime between various sorting algorithms, highlighting which ones were theoretically more efficient, and which ones actually performed better in practice. Quicksort, like the discussions in class, consistently performed well and sorted quickly; Insertion Sort and Bubble Sort ran quickly on already-sorted data, but performed worse in other areas. Out of the slow sorts, Insertion was the best, and Bubble was the worst in most cases. Out of all sorting algorithms, Quicksort and Mergesort performed similarly, but Quicksort was a bit faster.